Lab 07 - Single Variable Models and Naïve Bayes model

Learning Outcomes

In this lab you will learn how to build a Naïve Bayes model for the KDD dataset. The dataset is available at 'https://github.com/WinVector/zmPDSwR/tree/main/KDD2009'

Single variable models for the KDD dataset

Firstly we read in the dataset, separate the categorical columns from the numerical columns, and train our single variable models to predict the `churn` variable. The code is given in the source file provided below.

Data preparation

In this step, we read in the data, form our training, calibration, and test sets. Also included here is the `calcAUC` function that we need for the subsequent steps.

Load data and form the training, calibration, and test sets

```
library(knitr)
library(ggplot2)
library(ROCR)

# you will need to modify the path variable below
path <- '../data_v2/KDD2009/'
d <- read.table(paste0(path, 'orange_small_train.data.gz'),
                header=T, sep='\t', na.strings=c('NA',''), as.is=FALSE)

churn <- read.table(paste0(path,'orange_small_train_churn.labels.txt'), header=F, sep='\t
d$churn <- churn$V1             #___churn___

set.seed(729375)
d$rgroup <- runif(nrow(d))
dTrainAll <- subset(d, rgroup<=0.9)
dTest <- subset(d, rgroup > 0.9)

outcome <- 'churn'
pos <- '1'

vars <- setdiff(colnames(dTrainAll), c(outcome,'rgroup'))

catVars <- vars[sapply(dTrainAll[,vars], class) %in% c('factor','character')]
numericVars <- vars[sapply(dTrainAll[,vars], class) %in% c('numeric','integer')]

useForCal <- rbinom(n=dim(dTrainAll)[[1]], size=1, prob=0.1) > 0
dCal <- subset(dTrainAll, useForCal)
dTrain <- subset(dTrainAll,!useForCal)
```

Process the categorical variables

```
# perform single variable prediction for a given categorical column
mkPredC <- function(outCol, varCol, appCol) {
  pPos <- sum(outCol==pos)/length(outCol)
  naTab <- table(as.factor(outCol[is.na(varCol)]))
  pPosWna <- (naTab/sum(naTab))[pos]
  vTab <- table(as.factor(outCol), varCol)
  pPosWv <- (vTab[pos,]+1.0e-3*pPos)/(colSums(vTab)+1.0e-3)
  pred <- pPosWv[appCol]
  pred[is.na(appCol)] <- pPosWna
  pred[is.na(pred)] <- pPos
  pred
}

# now go through all the categorical variables in the `catVars` vector
# and perform the predictions. The outputs are stored back into the
# data frame.
for (v in catVars) {
  pi <- paste('pred', v, sep='')
  dTrain[,pi] <- mkPredC(dTrain[,outcome], dTrain[,v], dTrain[,v])
  dCal[,pi] <- mkPredC(dTrain[,outcome], dTrain[,v], dCal[,v])
  dTest[,pi] <- mkPredC(dTrain[,outcome], dTrain[,v], dTest[,v])
}
```

Process the numerical variables

```
# for numerical variables, we convert them into categorical one and
# call the `mkPredC` function above.
mkPredN <- function(outCol, varCol, appCol) {
  cuts <- unique(as.numeric(
    quantile(varCol, probs=seq(0, 1, 0.1), na.rm=T)))
  varC <- cut(varCol, cuts)
  appC <- cut(appCol, cuts)
  mkPredC(outCol, varC, appC)
}

# now go through all the numerical variables in the `numericVars` vector
# and perform the predictions. Again, the outputs are stored back into
# the data frame.
for (v in numericVars) {
  pi <- paste('pred', v, sep='')
  dTrain[,pi] <- mkPredN(dTrain[,outcome], dTrain[,v], dTrain[,v])
  dTest[,pi] <- mkPredN(dTrain[,outcome], dTrain[,v], dTest[,v])
  dCal[,pi] <- mkPredN(dTrain[,outcome], dTrain[,v], dCal[,v])
}
```

Define functions

There are certainly blocks of code that need to be used again and again. These blocks of code should be put into appropriate functions, e.g., one that calculates the log likelihood and one that calculates the AUC value of a given vector of predictions. We can have a separate function for calculating the *deviance*; however, since *deviance* is just `-2 * logLikelihood` (and since the saturated model usually has 0 log likelihood value and thus 0 deviance), we can simply make use of the log likelihood function directly.

```
# Function to return the AUC value of the predicted vector, ypred,
# compared against the ground truth vector, ytrue.
# Arguments:
# - ypred - must be a vector containing the predicted probabilities.
# - ytrue - must be a vector of the same length containing the ground
#           truth values 1s and 0s (or TRUE and FALSE).
calcAUC <- function(ypred, ytrue) {
    perf <- performance(prediction(ypred, ytrue), 'auc')
    as.numeric(perf@y.values)
}

# Define a function to compute log likelihood so that we can reuse it.
logLikelihood <- function(ypred, ytrue) {
  sum(ifelse(ytrue, log(ypred), log(1-ypred)), na.rm=T)
}
```

Our next steps is to go through the categorical and numerical variables and select the top performers – if they give a large drop in deviance (compared against that of the Null model).

Compute the log likelihood of the Null model

We can use the log likelihood (in fact, the *deviance*) of the Null model to help us select the top performers of the categorical variables and numerical variables.

```
# Compute the likelihood of the Null model on the calibration
# set (for the KDD dataset from previous lecture)
logNull <- logLikelihood(sum(dCal[,outcome]==pos)/nrow(dCal), dCal[,outcome]==pos)

cat("The log likelihood of the Null model is:", logNull)
## The log likelihood of the Null model is: -1178.017
```

Select the top performing variables

Run through categorical variables to select top performers

```
# selCatVars is a vector that keeps the names of the top performing categorical variables
selCatVars <- c()
minDrop <- 10  # may need to adjust this number

for (v in catVars) {
  pi <- paste('pred', v, sep='')
  devDrop <- 2*(logLikelihood(dCal[,pi], dCal[,outcome]==pos) - logNull)
  if (devDrop >= minDrop) {
    cat(sprintf("%6s, deviance reduction: %g\n", v, devDrop))
    selCatVars <- c(selCatVars, pi)
  }
}
## Var205, deviance reduction: 24.2323
## Var206, deviance reduction: 34.4434
## Var210, deviance reduction: 10.6681
## Var218, deviance reduction: 13.2455
## Var221, deviance reduction: 12.4098
## Var225, deviance reduction: 22.9074
## Var228, deviance reduction: 15.9644
## Var229, deviance reduction: 24.4946
```

Run through numerical variables to select top performers

```
# selNumVars is a vector that keeps the names of the top performing numerical variables.
selNumVars <- c()
minDrop <- 10  # may need to adjust this number
for (v in numericVars) {
  pi <- paste('pred', v, sep='')
  devDrop <- 2*(logLikelihood(dCal[,pi], dCal[,outcome]==pos) - logNull)
  if (devDrop >= minDrop) {
    cat(sprintf("%6s, deviance reduction: %g\n", v, devDrop))
    selNumVars <- c(selNumVars, pi)
  }
}
##   Var6, deviance reduction: 13.2431
##   Var7, deviance reduction: 18.685
##  Var13, deviance reduction: 10.0632
##  Var28, deviance reduction: 11.3864
##  Var72, deviance reduction: 12.5353
##  Var73, deviance reduction: 48.2524
##  Var74, deviance reduction: 19.6324
## Var113, deviance reduction: 23.136
## Var126, deviance reduction: 74.9556
## Var140, deviance reduction: 16.1816
## Var144, deviance reduction: 15.9858
## Var189, deviance reduction: 42.3059
```

Combine the two vectors of top performers

```
(selVars <- c(selCatVars, selNumVars))
##  [1] "predVar205" "predVar206" "predVar210" "predVar218" "predVar221"
##  [6] "predVar225" "predVar228" "predVar229" "predVar6"   "predVar7"
## [11] "predVar13"  "predVar28"  "predVar72"  "predVar73"  "predVar74"
## [16] "predVar113" "predVar126" "predVar140" "predVar144" "predVar189"
```

Note that the column corresponding to each element in the vector `selVars` contains the predicted probabilities that can be used directly to get the AUC values. The original variable names (categorical variables or numerical variables (being turned into categorical ones)) can be extracted by stripping off the prefix `pred`.

We might be interested in finding how each of these variables performs under the AUC measure. Below are the results:

```
cat("Performance of the top performing single variables on the test set:")
## Performance of the top performing single variables on the test set:
for (v in selVars) {
  # retrieve the original variable name (character location 5 onward)
  orig_v <- substring(v, 5)
  cat(sprintf("Variable %6s: AUC = %g\n", orig_v, calcAUC(dTest[,v], dTest[,outcome]==pos
}
## Variable Var205: AUC = 0.540339
## Variable Var206: AUC = 0.56317
## Variable Var210: AUC = 0.517033
## Variable Var218: AUC = 0.569005
## Variable Var221: AUC = 0.534149
## Variable Var225: AUC = 0.554632
## Variable Var228: AUC = 0.553782
## Variable Var229: AUC = 0.557849
## Variable   Var6: AUC = 0.558554
## Variable   Var7: AUC = 0.565896
## Variable  Var13: AUC = 0.563603
## Variable  Var28: AUC = 0.544406
## Variable  Var72: AUC = 0.514389
## Variable  Var73: AUC = 0.585498
## Variable  Var74: AUC = 0.566515
## Variable Var113: AUC = 0.52901
## Variable Var126: AUC = 0.637506
## Variable Var140: AUC = 0.569325
## Variable Var144: AUC = 0.511947
## Variable Var189: AUC = 0.568254
```

Exercise

You should write functions for display the *confusion matrix*, showing the *precision*, *recall* and *f1 score* formatted in a table so that you can call these functions as often as needed. Try to make your functions *modularised*, i.e., they should not be dependent on global variables defined outside the functions. Try to put in some brief explanation about the arguments passed to the functions. These functions will help to improve the overall presentation of your project.

Naïve Bayes

Naïve Bayes is a method that memorises how each training variable is related to the outcome, and then makes predictions by multiplying together the effects of each variable.

To demonstrate this, let's use a scenario in which we are trying to predict whether somebody is employed based on their level of education, their geographic region, and other variables.

- Naïve Bayes begins by reversing that logic and asking this question: Given that you are employed, what is the probability that you have a high school education?
- From that data, we can then make our prediction regarding employment.

Naïve Bayes Model - Simplified Maths Explanation

-

- The Bayes Rule (single evidence)

$$P(y=T|e_1) = \frac{P(y=T)P(e_1|y=T)}{P(e_1)}$$

- The Naïve assumption: all variables are independent

$$P(e_1,\ldots,e_n|y=T) = P(e_1|y=T)P(e_2|y=T)\cdots P(e_n|y=T)$$

- Therefore,

$$P(y=T|e_1,\ldots,e_n) = \frac{P(y=T)P(e_1|y=T)\cdots P(e_n|y=T)}{P(e_1,\cdots,e_n)}$$

Estimate the posterior

- The terms in the numerator on the right side of the final expression above can be calculated efficiently from the training data.
- We don't have a direct scheme for estimating the denominator term in the Naïve Bayes expression (these are called the joint probability of the evidence), but we can still estimate $P(y = T \mid \text{evidence})$and $P(y = F \mid \text{evidence})$.
- We know by the law of total probability that we should have $P(y = T \mid \text{evidence}) + P(y = F \mid \text{evidence}) = 1$.
- It is enough to pick a denominator such that our estimates add up to 1.

Naïve Bayes Model

Log conversion

For numerical reasons, it is better to convert the products into sums, by taking the log of both sides.

Since the denominator term is the same in both expressions, we can ignore it as all we need is to determine which of the following expressions is greater:

$$\text{score}(P(T|e_1,...,e_n)) = \log(P(y=T)) + \log(P(y=T|e_1)) + ... + \log(P(y=T|e_n))$$

$$\text{score}(P(F|e_1,...,e_n)) = \log(P(y=F)) + \log(P(y=F|e_1)) + ... + \log(P(y=F|e_n))$$

Smoothing

Cromwell's rule – no probability estimate of 0 should ever be used in probabilistic reasoning.

- This is because if you are combining probabilities by multiplication (the most common method of combining probability estimates), then once some term is 0, the entire estimate will be 0 no matter what the values of the other terms are.
- The most common form of smoothing is called *Laplace smoothing*, which counts k

successes out of n trials as a success ratio of $\frac{k+1}{n+1}$and not as a ratio of $\frac{k}{n}$ (defending against the $k = 0$ case).

From single-variable models to Naïve Bayes

All of the single-variable models we have built up to now are estimates of the form

$$\text{model}(e_i) \sim P(y = T|e_i)$$

So our single-variable models can be directly used to build an overall Naïve Bayes model (without any need for additional record keeping).

Build our own Naïve Bayes model

Define our own Naïve Bayes classifier:

```
# proportion of positive instances (`churn` == pos) in the training set.
pPos <- sum(dTrain[,outcome]==pos)/length(dTrain[,outcome])

nBayes <- function(pPos, pf) {
  # proportion of negative instances (`churn` != pos)
  pNeg <- 1 - pPos
  smoothingEpsilon <- 1.0e-5
  scorePos <- log(pPos + smoothingEpsilon) +
    rowSums(log(pf/pPos + smoothingEpsilon))
  scoreNeg <- log(pNeg + smoothingEpsilon) +
    rowSums(log((1-pf)/(1-pPos) + smoothingEpsilon))
  m <- pmax(scorePos, scoreNeg)
  expScorePos <- exp(scorePos-m)
  expScoreNeg <- exp(scoreNeg-m)
  expScorePos/(expScorePos+expScoreNeg)
}
```

Use our Naïve Bayes model for predictions

```
dTrain$nbpredl <- nBayes(pPos, dTrain[,selVars])
dCal$nbpredl <- nBayes(pPos, dCal[,selVars])
dTest$nbpredl <- nBayes(pPos, dTest[,selVars])
cat('The AUC value for the training set is:', calcAUC(dTrain$nbpredl, dTrain[,outcome]==p
## The AUC value for the training set is: 0.6834715
cat('The AUC value for the calibration set is:', calcAUC(dCal$nbpredl, dCal[,outcome]==po
## The AUC value for the calibration set is: 0.679693
cat('The AUC value for the test set is:', calcAUC(dTest$nbpredl, dTest[,outcome]==pos))
## The AUC value for the test set is: 0.6688521
```

Naïve Bayes implementation using the `e1071` package

Alternatively (and it is easier also), we can use the `e1071` library which has the `naiveBayes` function available. All we need to supply are:

- the formula that defines the dependent variable in terms of the independent variables, and
- the training set.

The training step is very simple:

```
library('e1071')

# construct the formula f as a character string
(f <- paste('as.factor(',outcome,' > 0) ~ ', paste(selVars, collapse=' + '), sep=''))
## [1] "as.factor(churn > 0) ~ predVar205 + predVar206 + predVar210 + predVar218 + predVa

# variable `nmodel` below is the model trained from Naïve Bayes
nbmodel <- naiveBayes(as.formula(f), data=dTrain)
```

The prediction step on the calibration set is given below:

```
dTrain$nbpred <- predict(nbmodel, newdata=dTrain, type='raw')[,'TRUE']
dCal$nbpred <- predict(nbmodel, newdata=dCal, type='raw')[,'TRUE']
dTest$nbpred <- predict(nbmodel, newdata=dTest, type='raw')[,'TRUE']

cat('*** Using the "pred" variable for Naïve Bayes:\n')
## *** Using the "pred" variable for Naïve Bayes:
cat('The AUC value for the training set is:', calcAUC(dTrain$nbpred, dTrain[,outcome]==po
## The AUC value for the training set is: 0.6903027
cat('The AUC value for the calibration set is:', calcAUC(dCal$nbpred, dCal[,outcome]==pos
## The AUC value for the calibration set is: 0.6841465
cat('The AUC value for the test set is:', calcAUC(dTest$nbpred, dTest[,outcome]==pos), "\
## The AUC value for the test set is: 0.6867772
```

Using the original variables?

The Naïve Bayes method is suitable for datasets that have mainly categorical variables, as it needs to count the number of instances for the different combinations of levels of the variables in order to get the posterior probability estimated. In the code above, we use the "`pred`" columns which contain probabilities of the predictions. These are all numerical columns that require Naïve Bayes to convert them back to categorical variables. For numerical variables, the common approach is to assume their values follow the Gaussian distribution and then quantize the values into discrete intervals as what we did in the in the `mkPredN` function above. So you might wonder whether we would get similar prediction performance if we use the original variables from the dataset instead of those ones with the prefix `pred`. Let's try that.

```
# Retrieve the original variable names (character location 5 onward)
selVars.orig <- substring(selVars, 5)
cat("High performing variables:", selVars.orig)
## High performing variables: Var205 Var206 Var210 Var218 Var221 Var225 Var228 Var229 Var

(f <- paste('as.factor(',outcome,' > 0) ~ ', paste(selVars.orig, collapse=' + '), sep=''))
## [1] "as.factor(churn > 0) ~ Var205 + Var206 + Var210 + Var218 + Var221 + Var225 + Var2

# training
nbmodel <- naiveBayes(as.formula(f), data=dTrain)

# predicting
dTrain$nbpred <- predict(nbmodel, newdata=dTrain, type='raw')[,'TRUE']
dCal$nbpred <- predict(nbmodel, newdata=dCal, type='raw')[,'TRUE']
dTest$nbpred <- predict(nbmodel, newdata=dTest, type='raw')[,'TRUE']

cat("*** Using the original variables for Naïve Bayes:")
## *** Using the original variables for Naïve Bayes:
cat('The AUC value for the training set is:', calcAUC(dTrain$nbpred, dTrain[,outcome]==po
## The AUC value for the training set is: 0.6509002
cat('The AUC value for the calibration set is:', calcAUC(dCal$nbpred, dCal[,outcome]==pos
## The AUC value for the calibration set is: 0.6418557
cat('The AUC value for the test set is:', calcAUC(dTest$nbpred, dTest[,outcome]==pos), "\
## The AUC value for the test set is: 0.6286057
```

We see a $4 - 5\%$ drop in AUC values for all the three sets. It shows that the predicted probabilities from the single variables do help.

Naïve Bayes is suitable for a large number of features

We can see the improvement on the AUC values when Naïve Bayes combines the top performing single variables together. Naïve Bayes is particularly useful when the dataset has many categorical variables, as demonstrated from the KDD dataset above.

Naïve Bayes is the workhorse method when classifying text documents (as done by email spam detectors).

- Standard model for text documents (usually called *bag-of-words* or *bag-of-k-grams*) can have an extreme number of possible features.
  - In the *bag-of-k-grams* model, we pick a small k (typically 2) and each possible consecutive sequence of k words is a possible feature.
  - Each document is represented as a bag, which is a sparse vector indicating which *k-grams* are in the document.
  - The number of possible features can run into the millions, but each document only has a non-zero value on a number of features proportional to k times the size of the document.

Tasks

-

Write a function that can plot the performance of logistic regression, decision trees, kNN and Naïve Bayes

- Write a function that can display and compare the performance of the above models.
- Incorporate the two functions into your Project.

References

- Best Machine Learning Packages in R: https://www.r-bloggers.com/what-are-the-best-machine-learning-packages-in-r/

Previous
Lab 06 - Explain ML Models using LIME

Next
Lab 08 - Project (Classification)

Last updated 9 days ago