mml87_nb

Kelvin

27 June 2017

```
dag = empty.graph(c("X1", "X2", "Y"))
dag = set.arc(dag, "Y", "X1")
dag = set.arc(dag, "Y", "X2")
cpts = randCPTs(dag, 2, 1)
data = rbn(cpts, 1000)
nVars = ncol(data)
# model = naiveBayes(Y ~ ., data = data)
# model$tables
```

Log likelihood

For NB with multinomial variables, there are two types of parameters $P(Y = y)$ and $P(X_j = x|Y = y)$. The likelihood of Naive Bayes is then

$$\begin{aligned} P(Y|\vec{X}) &= \frac{P(Y)\prod_{j=1}^{m} P(X_j|Y)}{P(\vec{X})} \\ &= \frac{P(Y)\prod_{j=1}^{m} P(X_j|Y)}{\sum_Y P(Y)\prod_{j=1}^{m} P(X_j|Y)} \end{aligned}$$

where $\vec{X} =< X_1, \ldots, X_m >$ is a vector of m input variables. The negative loglikelihood of an entire data set with n observations is

$$\begin{aligned} L &= -\sum_{i=1}^{n} \log P(Y|X) \\ &= -\sum_{i=1}^{n} \left[\log P(Y) + \sum_{j=1}^{m} \log P(X_j|Y) - \log \left(\sum_{Y} P(Y) \prod_{j=1}^{m} P(X_j|Y) \right) \right] \end{aligned}$$

Since it is not known whether or not there is a closed form solution for the determinant of the FIM, we firstly use MLE of NB parameters

$$\begin{aligned} P(Y = y) &= \frac{count(y)}{n} \\ P(X_j = x|Y = y) &= \frac{count(x) \cup count(y)}{count(y)} \end{aligned}$$

The following are implementations of MLE of NB parameters.

MLE implementations

A function to calculate maximum likelihood estimation of nb parameters.

```
# y is the output node
# x is a set of input nodes
# estimated parameters are stored in a list
# smoothing is additive smoothing constant to avoid 0 probabilities
# it can take any other values
```

```r
mle_est_nb = function(y, x, data, smoothing = 1) {
  xIndices = which(colnames(data) %in% x)
  yIndex = which(colnames(data) == y)
  lst = list()
  for (i in 1:length(x)) {
    lst[[i]] = (table(data[, c(yIndex, xIndices[i])]) + smoothing) /
      (rowSums(table(data[, c(yIndex, xIndices[i])])) + smoothing * nlevels(data[, xIndices[i]]))
  } # end for i
  lst[[yIndex]] = (table(data[, yIndex]) + smoothing) / (nrow(data) + smoothing * nlevels(data[, 3]))
  names(lst) = c(x, y)
  return(lst)
}

# (mle_est_nb("Y", c("X1", "X2"), data, 0)) # with no smoothing
pars = mle_est_nb("Y", c("X1", "X2"), data, 1) # with smoothing
```

Fisher information matrix

Define $\theta_0 = P(Y = y)$ and $\theta_j = P(X_j = x|Y = y), \forall j \in [1, m]$. The second derivative of the above negative log likelihood w.r.t each parameter is

$$
\begin{aligned}
\frac{\partial^2 L}{\partial \theta_k^2} &= \sum_{i=1}^{n} \left[\frac{1}{\theta_k^2} - \left(\frac{\prod_{j=0}^{\neq k} \theta_j}{\sum_y \prod_{j=0}^{m} \theta_j} \right)^2 \right] \\
\frac{\partial^2 L}{\partial \theta_k \partial \theta_l} &= \sum_{i=1}^{n} \left[\frac{(\sum_y \prod_{j=0}^{m} \theta_j)(\sum_{j=0}^{\neq k,l} \theta_j) - (\prod_{j=0}^{\neq k} \theta_j)(\prod_{j=0}^{\neq l} \theta_j)}{\left(\sum_y \prod_{j=0}^{m} \theta_j\right)^2} \right]
\end{aligned}
$$

to be a vector of parameters for Naive Bayes, where $\theta_0 = P(Y = y_1)$ and $\theta j1 = P(X_j = x_j|Y = y_1)$ and $\theta j2 = P(X_j = x_j|Y = y_2)$ respectively. Then we have the following

$$
\begin{aligned}
P(X) &= \sum_{Y=y} P(Y = y) \prod_{j=1}^{m} P(X_j|Y = y) \\
\frac{\partial^2 L}{\partial \theta_0^2} &= \sum_{i=1}^{n} \left[\frac{1}{\theta_0^2} - \left(\frac{\prod_{j=1}^{m} \theta_{j1} - \prod_{j=1}^{m} \theta_{j2}}{P(X)} \right)^2 \right] \\
\frac{\partial^2 L}{\partial \theta_{k1}^2} &= \sum_{i=1}^{n} \left[\frac{1}{\theta_{k1}^2} - \left(\frac{\theta_0 \prod_{j=1}^{m, j\neq k} \theta_{j1}}{P(X)} \right)^2 \right] \\
\frac{\partial^2 L}{\partial \theta_{k2}^2} &= \sum_{i=1}^{n} \left[- \left(\frac{(1-\theta_0) \prod_{j=1}^{m, j\neq k} \theta_{j2}}{P(X)} \right)^2 \right]
\end{aligned}
$$

for $k \in [1, m]$.

FIM implementations

```r
y = "Y"
x = c("X1", "X2")
```

```r
arities = sapply(data, nlevels)
yIndex = which(colnames(data) == y)
xIndices = which(colnames(data) %in% x)

# a function to get corresponding probabilities for each row of data
# only need to specify yIndex since we need to get probs for all Xs given Y
# this function assumes Y has no parents due to the way prob(y) is stored in pars
# the probs need to be stored in the order of p(x_1|y), ..., p(x_m|y), p(y)
extract_probs = function(yIndex, pars, dataPoint) {
  probs = rep(0, ncol(data))
  yValue = dataPoint[[yIndex]]
  ic = 1 # initialize an incremental index to store prob(x|y) in vector probs
  for (xIndex in 1:ncol(data)) {
    if (xIndex != yIndex) {
      xValue = dataPoint[[xIndex]]
      probs[ic] = pars[[xIndex]][xValue, yValue]
      ic = ic + 1
    } # end if
  } # end for each x
  probs[length(probs)] = pars[[yIndex]][yValue]
  return(probs)
}

y = "Y"
x = c("X1", "X2")
arities = sapply(data, nlevels)
yIndex = which(colnames(data) == y)
xIndices = which(colnames(data) %in% x)

px = function(yIndex, xIndices, pars, dataPoint, arities) {
  ss = 0
  for (yValue in 1:arities[yIndex]) {
    mm = pars[[yIndex]][yValue]
    for (j in 1:length(xIndices)) {
      #xValue = data[dataPoint, xIndices[j]]
      xValue = dataPoint[[xIndices[j]]]
      mm = mm * pars[[j]][yValue, xValue]
    } # end for each x_j
    ss = ss + mm
  } # end for each y value
  return(ss)
}

# extract the corresponding probs for each row of data from pars
probs_table = t(apply(data, 1, extract_probs, yIndex = 3, pars = pars))
colnames(probs_table) = colnames(data)

#px(3, c(1,2), pars, data[101,], arities)
allPXs = apply(data, 1, px, yIndex=3,xIndices=c(1,2),pars=pars,arities=arities)

# cbind allPXs to the last column of probs_table with colname "X"
probs_table = cbind(probs_table, X = allPXs)
head(probs_table)
```

```
##             X1        X2         Y         X
## [1,] 0.4565217 0.3652174 0.7714571 0.3864256
## [2,] 0.4565217 0.4547804 0.7714571 0.3019230
## [3,] 0.4565217 0.3652174 0.7714571 0.3864256
## [4,] 0.4565217 0.3652174 0.7714571 0.3864256
## [5,] 0.4565217 0.4547804 0.7714571 0.3019230
## [6,] 0.4565217 0.4547804 0.7714571 0.3019230
```

```
#prod(probs_table[1,])

# FIM is an (M+1) by (M+1) symmetric matrix
# probsPoint is the row of probs in probs_table corresponding to a dataPoint in data
# cellIndex = 1 corresponds to 2nd derivative of L w.r.t. p(y),
# which is stored at the last of the parameter list "pars" and
# 2nd last column of probs_table
diag_entry = function(nVars, diagIndex, probsPoint) {
  if (diagIndex == 1) {
    varIndex = nVars
  } else {
    varIndex = diagIndex - 1
  }
  value = 1 / (probsPoint[[varIndex]] ^ 2) -
  (prod(probsPoint[-c(varIndex, nVars + 1)]) ^ 2) / (probsPoint[[nVars + 1]] ^ 2)
  return(value)
}

#diag_entry(3, 1, probs_table[1,])
#sum(apply(probs_table, 1, diag_entry, nVars = 3, diagIndex = 1))
```

```
# since FIM is symmetric, we only need to fill entries in either above or below diagonal
# above_diag_entry filles entries above the diagonal
# i.e. rowIndex < colIndex, where rowIndex \in [1, m] and colIndex \in [2, m+1]
above_diag_entry = function(nVars, rowIndex, colIndex, probsPoint) {
  # rowVarIndex := 1st var differentiated w.r.t.
  if (rowIndex == 1) {
    rowVarIndex = nVars
  } else {
    rowVarIndex = rowIndex - 1
  }
  # colVarIndex := 2nd var differentiated w.r.t.
  # since colIndex >= 2, col var will never be p(y)
  colVarIndex = colIndex - 1
  px = probsPoint[[nVars + 1]]
  exclRowVar = prod(probsPoint[-c(rowVarIndex, nVars + 1)])
  exclColVar = prod(probsPoint[-c(colVarIndex, nVars + 1)])
  exclBothVar = prod(probsPoint[-c(rowVarIndex, colVarIndex, nVars + 1)])
  return((px * exclBothVar - exclRowVar * exclColVar) / (px ^ 2))
}
```

```
fim = matrix(0, nrow = nVars, ncol = nVars)

for (i in 1:(nVars - 1)) {
  for (j in (i + 1):nVars) {
    fim[i, j] = sum(apply(probs_table, 1, above_diag_entry, nVars = 3, rowIndex = i, colIndex = j))
  }
```

```r
}

fim = fim + t(fim)

for (i in 1:nrow(fim)) {
  diag(fim)[i] = sum(apply(probs_table, 1, diag_entry, nVars = 3, diagIndex = i))
}

fim
```

```
##           [,1]      [,2]      [,3]
## [1,] 4473.5653  910.0078  902.7417
## [2,]  910.0078 4612.2260 1269.4093
## [3,]  902.7417 1269.4093 4366.3081
```

```r
log(det(fim))
```

```
## [1] 25.07474
```