STATS 790
Project: Implement Naive Bayes classifier from scratch

Doudou Jin – 400174871

4/30/23

Introduction

This project aims to implement three types of Naive Bayes classifier from scratch, including Gaussian Naive Bayes, Multinomial Naive Bayes and Bernoulli Naive Bayes, and compare those classifier with the R build-in Naive Bayes classifier from naivebayes(Majka 2019) package. Naive Bayes is a supervised machine learning algorithm for classification. It is based on the Bayes' Theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

The Naive Bayes classifier has a strong assumption of independence among predictors. Hence, the above formula can be rewritten as:

$$P(y|\mathbf{X}) = \frac{P(\mathbf{X}|y)P(y)}{P(\mathbf{X})} = \frac{P(x_1|y)\cdots P(x_p|y)P(y)}{P(x_1)\cdots P(x_p)} \propto P(x_1|y)\cdots P(x_p|y)P(y)$$

where the $\mathbf{X} = (x_1, x_2, ..., x_p)$ is the set of predictors and y is one of the classes from response variable. Therefore, the $P(y|\mathbf{X})$ is the posterior probability, given the set of predictors, and the $P(y)$ is the prior probability of the particular class. The $P(\mathbf{X}|y)$ are the product of likelihoods of each predictors given the same class and the $P(\mathbf{X})$ are the product of marginal probabilities of all predictors. As the marginal probabilities are fixed for each class, the posterior probability is proportional to the numerator part, which is the product of likelihoods and prior probability.

One main challenge in Naive Bayes is the zero probability problem. It occurs when some discrete values of categorical variables do not present in training set with values of 0, or the frequencies of those categorical variables are zero, the predicted probability for those vaiables will be zero. Therefore, the Laplace smoothing techenique is widely used to handle this situation. The Laplace smoothing parameter allows us to adjust the frequency of categorical variables and more details will be explained in the next section based on different distributions.

Futhermore, the log likelihoods are used to prevent data underflow. As the likelihood values can be very small, multiplying several of these small values can result in products that are too small to be accurately represented in a computer system, leading to data underflow. Moreover, by using the log likelihoods, we can simply sum up the predicted likelihoods for each word to avoid the product of predicted likelihoods to be zero.

Implementation

Gaussian Naive Bayes

In Gaussian Naive Bayes, all predictors are continuous and are assumed to follow the Gaussian distribution, with the probability density function (p.d.f.):

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

There is no need to use Laplace smoothing parameter in Gaussian Naive Bayes. The Laplace smoothing parameter is used when dealing with observations appear in test set but not in training set. Such observations are discrete, such as frequency or categories. However, the values in Gaussian Naive Bayes are all continuous and the likelihoods of response variable y_i are calculated based on the corresponding probability density functions.

Therefore, the algorithm of Gaussian Naive Bayes are:

1. Calculate the log prior probabilities of classes.
2. Separate the training set by classes, then find the mean and variance of predictors in each class.
3. Substitute in the test set and calculate the log likelihoods for every predictors using the p.d.f. of the Gaussian distribution. The corresponding mean and variance are founded in step 2.
4. Sum up log likelihoods and the corresponding prior probability to get the log posterior probability of each class, and then exponentiate the log posterior probabilities.
5. Compare the posterior probabilities. The final predicted class is the class with the largest probability.
6. Compare the predicted class with the true class of the test set, and calculate the misclassification rate.

Multinomial Naive Bayes

In Multinomial Naive Bayes, the values of each predictor represent the occurrences or frequencies of the particular features. It is commonly used for text classification, and so the predictors are the specific words and the values of each cell are the frequencies of words in the text.

Let $N_{x_i|y}$ denote the frequencies of x_i in class y, N_y denotes the total values across all predictors in class y. Then the likelihoods(Kovachi 2018) can be calculated as following:

$$P(x_i|y) = \frac{N_{x_i|y} + \alpha}{N_y + \alpha * k}$$

where the α is the Laplace smoothing parameter(Jayaswal 2020), and is used to avoid zero likelihoods when x_i does not occur in the training set. The variable k is the number of predictors in the training set. In this case, the predicted likelihoods are given as $P(x_i|y) *$ the frequency of x_i in the test set.

The algorithm of Multinomial Naive Bayes are:

1. Calculate the log prior probabilities of classes.
2. Separate the training set by classes, then find the $N_{x_i|y}$ and N_y to calculate the log likelihoods for each predictor.
3. Substitute in the test set and estimate the log likelihoods for every predictor by multiplying the corresponding frequency. Then sum up all log likelihoods and the log prior probabilities to get the log posterior probabilities. Exponentiate the log posterior probabilities.
4. Compare the posterior probabilities. The final predicted class is the class with the largest log probability as logarithm is strictly increasing.
5. Compare the predicted class with the true class of the test set, and calculate the misclassification rate.

Bernoulli Naive Bayes

In Bernoulli Naive Bayes, all predictors are binary values, with 0 or 1 and follow a Bernoulli distribution. The 1 generally represents the presence of the particular feature. The probability mass function (p.m.f.) is given as follow:

$$f(x_i; p) = px_i + (1-p)(1-x_i) \quad \text{for } x_i \in \{0, 1\}$$

The $p = P(1_{x_i}|y)$ is the probability of presence of x_i given the class y. The formula of p is:

$$P(1_{x_i}|y) = \frac{\text{total number of 1 in } x_i \text{ in class y} + \alpha}{\text{total number of 1 in class y for all } x + \alpha * k}$$

The α here is the Laplace smoothing parameter, same as in Multinomial Naive Bayes. The variable k is the number of predictors in the training set.

Moreover, as all predictors in Bernoulli Naive Bayes are binary, with values of either 0 or 1, the calculate process for the probability of presence is the same as in Multinomial Naive Bayes. However, the methods for estimating the predicted likelihoods are different. In Bernoulli Naive Bayes, the predicted likelihoods are estimated using the probability mass function of Bernoulli distribution, while in Multinomial Naive Bayes, they are estimated by multiplying the probabilities and frequencies of predictors.

The algorithm of Bernoulli Naive Bayes are:

1. Calculate the log prior probabilities of classes.
2. Separate the training set by classes, then find $P(1_{x_i}|y)$ for each predictor in all classes.
3. Substitute in the test set and calculate the log likelihoods for every predictors using the p.m.f. of the Bernoulli distribution.
4. Sum up all log likelihoods and the corresponding log prior probability to get the log posterior probability of each class. Exponentiate the log posterior probabilities.
5. Compare the posterior probabilities. The final predicted class is the class with the largest probability.

6. Compare the predicted class with the true class of the test set, and calculate the mis-classification rate.

Code

```
# used packages
library(dplyr)
library(naivebayes)
library(e1071)
library(kernlab)
library(knitr)
library(Rlab)
```

```
# inputs: 1. df: the labeled dataset used for training
#         2. y: the response variable; note: it has to be quoted i.e. "y"
#         3. alpha: the Laplace smoothing parameter
#         4. test_set: the unlabed dataset for testing
#         5. test_ind: the true class for the test set
#         6. method: "Gaussian", "Multinomial", "Bernoulli", correspond
#                  to their specific classifiers

# outputs: 1. probability: the predicted probabilities for all classes
#                          for the test set
#          2. class: the predicted classes for the test set
#          3. table: the contingency table, comparing the predicted
#                    classes and the true classes
#          4. mis_classification: the mis-classification rate of this
#                                 classifier



naiveBayes3 <- function(df, y, alpha, test_set, test_ind, method){
```

```r
log_prior_prob <- function(df, y){

  log_prop <- df %>%
    dplyr::group_by(!!sym(y)) %>%

    # log of proportions
    summarise(log_prop = log(n()/nrow(df)))

  log_prop <- as.data.frame(log_prop)

  colnames(log_prop) <- c("levels", "log_prop")
  return(log_prop)
}

if(method == "Gaussian"){

  # find mean and standard deviation for each predictor
  mean_sd <- function(df, y){

    # remove response variable
    df2 <- df[, !(names(df) %in% y)]

    # calculate mean
    mean_X <- apply(df2, 2, mean)

    # calculate variance
    var_X <- apply(df2, 2, sd)

    # combine mean and variance
    names <- colnames(df2)
```

```
  meanVar_X <- cbind(names, mean_X, var_X)
  colnames(meanVar_X) <- c("predictors", "mean", "sd")
  rownames(meanVar_X) <- NULL

  return(meanVar_X)
}

# likelihoods for all predictors in all classes
# p(x1|y1),p(x2|y1),...,p(xp|yn)
likelihood <- function(df, y){
  mv <- data.frame()
  for(i in levels(df[[y]])){

    # classify response variables
    # calculate mean and variance for each variable for each class
    yi <- df %>% filter(!!sym(y) == i)
    mv_yi <- mean_sd(yi, y)
    mv_yi <- cbind(rep(i, ncol(df)-1), mv_yi)
    mv <- rbind(mv, mv_yi)
  }


  colnames(mv) <- c("levels", "predictors", "mean", "sd")
  mv$levels <- as.factor(mv$levels)
  mv$predictors <- as.factor(mv$predictors)
  mv$mean <- as.numeric(mv$mean)
  mv$sd <- as.numeric(mv$sd)

  return(mv)
}
```

```r
# P(x1|y)*P(x2|y)*...*P(xn|y)
# log(P(x1|y)) + log(P(x2|y)) + ... + log(P(xn|y))
# plug in the values of one observation in test set
pred_log_likelihood <- function(df, y, obs){

  llh_df <- likelihood(df, y)

  p_x_y <- data.frame(levels = factor(), log_likelihoods = numeric())

  for(i in levels(llh_df[,1])){

    # initial P(X|y_i)
    p_x_yi <- 0

    for(j in names(obs)){

      # calculate p(xj|yi)
      xj <- llh_df %>% filter(levels == i & predictors == j)
      xi <- dnorm(obs[[j]], mean = xj[, names(xj) %in% "mean"]
                  ,  sd = xj[, names(xj) %in% "sd"])

      # add up
      p_x_yi <- p_x_yi + log(xi)
    }

    # for each class
    p_x_y <- rbind(p_x_y, data.frame(levels = i, log_likelihoods = p_x_yi))
  }
  return(p_x_y)
}
```

```r
# log likelihoods + log prior probability
# predicted probability for one observation
# exponentiate
pred_prob <- function(df,y, obs){

  # prior probability
  prop_table <- log_prior_prob(df,y)

  # predicted log likelihoods
  pred_llh <- pred_log_likelihood(df, y, obs)

  new_table <- merge(pred_llh, prop_table, by="levels")

  # log P(x1|y) +... log P(xn|y) + log p(y)
  new_table$prob <- new_table$log_likelihoods + new_table$log_prop
  new_table$prob <- exp(new_table$prob)

  prob <- data.frame(t(new_table[["prob"]]))
  colnames(prob) <- c(new_table[,names(new_table) %in% "levels"])

  return(prob)
}

# predict the probabilities for all classes
# for all observations in test set
pred_all <- function(df,y,test_set){

  p_y_x <- apply(test_set, 1, function(row) pred_prob(df, y, row))
  p_y_x <- do.call(rbind, p_y_x)
  return(p_y_x)
}
```

```r
# predict classes for all observations in test set
pred_class <- function(df,y,test_set){
  predictions <- pred_all(df,y,test_set)

  # find the class with largest probability
  class_col <- max.col(predictions)
  class <- colnames(predictions)[class_col]
  return(class)
}

# compare to the true classes
# create a contingency table
compare_pred <- function(df,y,test_set, test_ind){
  pred <- pred_class(df,y,test_set)
  tab <- table(pred, test_ind)
  return(tab)
}

# return probabilities for each observation in each class
predicted_prob <- pred_all(df,y,test_set)

# return predicted class for each observation
predicted_class <- pred_class(df,y,test_set)

# return the contingency table, compared to the true classes
compare_tab <- compare_pred(df,y,test_set, test_ind)

# return the mis-classification rate
accuracy <- 1 - classAgreement(compare_tab)$diag
```

```
  return(list(probability = predicted_prob,
              class = predicted_class,
              table = compare_tab,
              mis_classification = accuracy))


}

# if method = "Multinomial", then use Multinomial Naive Bayes
else if(method == "Multinomial"){

  log_likelihood <- function(df, y, alpha){

    N_x_y <- df %>%
      group_by(!!sym(y)) %>%
      summarise(across(everything(), sum))

    N_x_y_a <- N_x_y[,!names(N_x_y) %in% y] + alpha

    N_y <- rowSums(N_x_y[,!names(N_x_y) %in% y])
    N_y_ak <- N_y + alpha*(ncol(N_x_y_a)-1)

    P_x_y <- lapply(1:nrow(N_x_y_a), function(x) log(N_x_y_a[x,]/N_y_ak[x]))
    P_x_y <- do.call(rbind, P_x_y)

    levels <- levels(df[[y]])

    P_x_y <- cbind(levels, P_x_y)

    return(P_x_y)
  }
```

```r
pred_log_llh <- function(df, y, alpha, obs){

  log_llh <- log_likelihood(df, y, alpha)
  log_llh <- log_llh[, !names(log_llh)=="levels"]

  rowProduct <- function(row, obs){
    row*obs
  }
  pred_P <- apply(log_llh, 1, rowProduct, obs = obs)
  pred_P <- do.call(rbind, pred_P)

  levels <- levels(df[[y]])

  pred_P <- cbind(levels, pred_P)

  return(pred_P)
}



# predict the log posterior probabilities of one observation
pred_prob <- function(df, y, alpha, x){

  # log likelihood
  log_llh <- pred_log_llh(df, y, alpha, x)
  log_llh[["levels"]] <- as.factor(log_llh[["levels"]])

  # log porior probability
  log_prior <- log_prior_prob(df, y)
```

```
  new_table <- merge(log_llh, log_prior, by="levels")

  new_table <- rowSums(new_table[,!names(new_table) %in% "levels"])

  new_table <- exp(new_table)

  pred_P <- data.frame(t(new_table))
  colnames(pred_P) <- c(log_llh[["levels"]])

  return(pred_P)
}

# predict log probabilities for all observations
pred_all_prob <- function(df, y, alpha, test_set){

  p_y_x <- lapply(1:nrow(test_set), function(x) pred_prob(df, y, alpha, test_set[x,]))
  p_y_x <- do.call(rbind, p_y_x)
  return(p_y_x)
}

# predict the classes for all observations in test set
pred_class <- function(df,y, alpha, test_set){
  predictions <- pred_all_prob(df, y, alpha, test_set)

  # find the class with largest probability
  class_col <- max.col(predictions)
  class <- colnames(predictions)[class_col]
  return(class)
}

# compare to the true classes
```

```
  # create a contingency table
  compare_pred <- function(df,y, alpha, test_set, test_ind){
    pred <- pred_class(df,y,alpha, test_set)
    tab <- table(pred, test_ind)
    return(tab)
  }


  # return probabilities for each observation in each class
  predicted_prob <- pred_all_prob(df, y, alpha, test_set)

  # return predicted class for each observation
  predicted_class <- pred_class(df,y, alpha, test_set)

  # return the contingency table, compared to the true class
  compare_tab <- compare_pred(df,y, alpha, test_set, test_ind)

  # return the mis-classification rate
  mis_classification <- 1 - classAgreement(compare_tab)$diag

  return(list(probability = predicted_prob,
              class = predicted_class,
              table = compare_tab,
              mis_classification = mis_classification))

}

# if method = "Bernoulli", then use Bernoulli Naive Bayes
else if(method == "Bernoulli"){

  probabilityOne <- function(df, y, alpha){
```

```
  N_x_y <- df %>%
    group_by(!!sym(y)) %>%
    summarise(across(everything(), sum))

  N_x_y_a <- N_x_y[,!names(N_x_y) %in% y] + alpha

  N_y <- rowSums(N_x_y[,!names(N_x_y) %in% y])
  N_y_ak <- N_y + alpha*(ncol(N_x_y_a)-1)

  P_x_y <- lapply(1:nrow(N_x_y_a), function(x) N_x_y_a[x,]/N_y_ak[x])
  P_x_y <- do.call(rbind, P_x_y)

  levels <- levels(df[[y]])

  P_x_y <- cbind(levels, P_x_y)

  return(P_x_y)
}

# predict the likelihoods for one observation
pred_log_llh <- function(df, y, alpha, obs){

  # P(1_{x_i}|y)
  prob <- probabilityOne(df, y, alpha)
  prob$levels <- as.factor(prob$levels)

  p_x_y <- data.frame(levels = factor(), log_likelihoods = numeric())

  # for each class
  for(i in levels(prob[["levels"]])){
```

```
    p_x_yi <- 0

    for(j in names(obs)){

      pj <- prob %>% filter(levels == i) %>% select(!!sym(j))

      # p.m.f. of Bernoulli distribution
      p_xj_yi <- dbern(obs[[j]], pj[[j]])

      # add up
      p_x_yi <- p_x_yi + log(p_xj_yi)
    }
    p_x_y <- rbind(p_x_y, data.frame(levels = i, log_likelihoods = p_x_yi))
  }
  return(p_x_y)
}


pred_prob <- function(df, y, alpha, x){

  # log likelihood
  log_llh <- pred_log_llh(df, y, alpha, x)
  log_llh[["levels"]] <- as.factor(log_llh[["levels"]])

  # log porior probability
  log_prior <- log_prior_prob(df, y)

  new_table <- merge(log_llh, log_prior, by="levels")

  new_table <- rowSums(new_table[,!names(new_table) %in% "levels"])
```

```
  new_table <- exp(new_table)

  pred_P <- data.frame(t(new_table))
  colnames(pred_P) <- c(log_llh[["levels"]])

  return(pred_P)
}

pred_all_prob <- function(df, y, alpha, test_set){

  p_y_x <- lapply(1:nrow(test_set), function(x) pred_prob(df, y, alpha, test_set[x,]))
  p_y_x <- do.call(rbind, p_y_x)
  return(p_y_x)
}

# predict the classes for all observations in test set
pred_class <- function(df,y, alpha, test_set){
  predictions <- pred_all_prob(df,y, alpha, test_set)

  # find the class with largest probability
  class_col <- max.col(predictions)
  class <- colnames(predictions)[class_col]
  return(class)
}

# compare to the true classes
# create a contingency table
compare_pred <- function(df, y, alpha, test_set, test_ind){
  pred <- pred_class(df,y, alpha, test_set)
  tab <- table(pred, test_ind)
  return(tab)
```

```
    }

    # return probabilities for each observation in each class
    predicted_prob <- pred_all_prob(df,y, alpha, test_set)

    # return predicted class for each observation
    predicted_class <- pred_class(df,y, alpha, test_set)

    # return the contingency table, compared to the true class
    compare_tab <- compare_pred(df, y, alpha, test_set, test_ind)

    # return the mis-classification rate
    mis_classification <- 1 - classAgreement(compare_tab)$diag

    return(list(probability = predicted_prob,
                class = predicted_class,
                table = compare_tab,
                mis_classification = mis_classification))

  }
}
```

Comparison between naiveBayes3() and R build-in functions

Gaussian Naive Bayes – Iris Dataset

To compare the implemented Gaussian Naive Bayes with the R build-in gaussian_naive_bayes() in naivebayes package, the iris dataset is used. The iris dataset contains 5 variables and 150 observations. All 4 predictors are continuous, and are the features of the iris flowers. The response variable is the species, including three levels: setosa, versicolor, and virginica. The aim is to classify the species of the iris. We simply split the dataset into training and test sets with a ratio of 4:1.

code

```
# Iris dataset
data("iris")

set.seed(1)

# split into training and test sets, with ratio of 8:2
trainIris <- iris %>% mutate(index=1:nrow(iris)) %>%
  group_by(Species) %>%
  mutate(n=n()) %>%
  sample_frac(size=0.8, weight=n) %>%
  ungroup()

trainIris_ind <- trainIris$index
testIris <- iris[-trainIris_ind, ]
trainIris <- trainIris[, 1:5]
```

```
# implemented Gaussian Naive Bayes classifier
start_time_gNB <- Sys.time()
iris_gNB <- naiveBayes3(df=trainIris, y="Species"
                        , test_set=testIris[,-5]
                        , test_ind = testIris[,5]
                        , method = "Gaussian")
end_time_gNB <- Sys.time()

running_time_gNB <- end_time_gNB - start_time_gNB
```

```
trainIris_ind <- unlist(trainIris[,5])
testIris_R <- as.matrix(testIris[,-5])

# R build-in Gaussian Naive Bayes classifier
```

```r
start_time_gNBR <- Sys.time()
iris_gNBR <- gaussian_naive_bayes(x = trainIris[,-5], y = trainIris_ind)

iris_gNBR_pred <- predict(iris_gNBR, newdata = testIris_R, type = "class")

tabIrisR <- table(iris_gNBR_pred, testIris[,5])
end_time_gNBR <- Sys.time()

running_time_gNBR <- end_time_gNBR - start_time_gNBR
```

```r
# Contingency table of implemented Gausian Naive Bayes
iris_gNB[["table"]]
```

```
            test_ind
pred         setosa versicolor virginica
  setosa         10          0         0
  versicolor      0         10         2
  virginica       0          0         8
```

```r
# Contingency table of R build-in Gaussian Naive Bayes
tabIrisR
```

```
iris_gNBR_pred setosa versicolor virginica
    setosa         10          0         0
    versicolor      0         10         2
    virginica       0          0         8
```

```r
missIrisR <- round(1-classAgreement(tabIrisR)$diag,2)
missIrisgNB <- round(iris_gNB[["mis_classification"]],2)
running_time_gNB <- round(running_time_gNB,2)
running_time_gNBR <- round(running_time_gNBR,2)
compare_gNB <- data.frame(implemented_gNB = c(missIrisgNB, running_time_gNB)
```

```
                          , R_build_in_gNB = c(missIrisR, running_time_gNBR))
rownames(compare_gNB) <- c("mis_classification", "running_time(secs)")
```

Table 1: Comparison table for Gaussian Naive Bayes

	implemented_gNB	R_build_in_gNB
mis_classification	0.07	0.07
running_time(secs)	10.77	0.38

From the above comparison tables, we can see that the implemented Gaussian Naive Bayes has the same prediction performance as the R build-in function in terms of accuracy. However, the running time of the implemented function is much higher than the R build-in function.

Multinomial Naive Bayes – Spam Dataset

To compare the implemented Multinomial Naive Bayes with the R build-in multinomial_naive_bayes() in naivebayes package, the spam dataset in kernlab(Karatzoglou et al. 2004) (Karatzoglou, Smola, and Hornik 2023) package is used. It contains 58 variables and a total of 4601 emails. The first 1-48 variables are the percentage of words occurred in the emails and 49-54 variables are the percentage of characters. The 55-57 variables are related to the capital letters. The last variable is the response variable with 2 levels: spam and nonspam. For convenience, we randomly pick 100 spam emails and 100 nonspam emails, and simply use the first 1-48 variables. Similar with iris dataset, the simplified spam dataset is split into training and test sets with a ratio of 4:1.

```
data(spam)


set.seed(1)


# 100 spam and 100 non-spam
spam_ind <- which(spam$type == "spam")
nonspam_ind <- which(spam$type == "nonspam")
sampled_spam_ind <- sample(spam_ind, 100)
```

```r
sampled_nonspam_ind <- sample(nonspam_ind, 100)

sampled_ind <- sort(c(sampled_nonspam_ind, sampled_spam_ind))
spam_new <- spam[sampled_ind, ]

# only consider words and convert to integers
spam_new2 <- spam_new[,-c(49:58)]
spam_new2 <- cbind(spam_new2, spam_new[,58])
names(spam_new2)[names(spam_new2) == "spam_new[, 58]"] <- "type"
```

```r
set.seed(1)

# split with a ratio of 8:2
trainSpam_ind <- sample(nrow(spam_new2), 0.8 * nrow(spam_new2))
trainSpam <- spam_new2[trainSpam_ind, ]
testSpam <- spam_new2[-trainSpam_ind, ]
```

```r
# implemented Multinomial Naive Bayes
start_time_mNB <- Sys.time()
spam_mNB <- naiveBayes3(df=trainSpam, y="type", alpha=1
                        , test_set=testSpam[,-49]
                        , test_ind=testSpam[,49]
                        , method = "Multinomial")
end_time_mNB <- Sys.time()
running_time_mNB <- end_time_mNB - start_time_mNB
```

```r
trainSpam_ind <- unlist(trainSpam[,49])
testSpam_R <- as.matrix(testSpam[,-49])

start_time_mNBR <- Sys.time()
# R build=in Multinomial Naive Bayes
spam_mNBR <- multinomial_naive_bayes(x=trainSpam[,-49], y= trainSpam_ind, laplace = 1)
```

```r
spam_mNBR_pred <- predict(spam_mNBR
                          , newdata = testSpam_R
                          , type = "class")

spam_mNBtab <- table(spam_mNBR_pred, testSpam[,49])
end_time_mNBR <- Sys.time()
running_time_mNBR <- end_time_mNBR - start_time_mNBR
```

```r
# Contingency table of implemented Multinomial Naive Bayes
spam_mNB[["table"]]
```

```
         test_ind
pred      nonspam spam
  nonspam      14    1
  spam          1   24
```

```r
# Contingency table of R build-in Multinomial Naive Bayes
spam_mNBtab
```

```
spam_mNBR_pred nonspam spam
       nonspam      14    1
       spam          1   24
```

```r
missSpamR <- round(1-classAgreement(spam_mNBtab)$diag,2)
missSpamgNB <- round(spam_mNB[["mis_classification"]],2)
running_time_mNB <- round(running_time_mNB,2)
running_time_mNBR <- round(running_time_mNBR,2)
compare_mNB <- data.frame(implemented_gNB = c(missSpamgNB, running_time_mNB)
                          , R_build_in_gNB = c(missSpamR, running_time_mNBR))
rownames(compare_mNB) <- c("mis_classification", "running_time(secs)")
```

Table 2: Comparison table for Multinomial Naive Bayes

	implemented_gNB	R_build_in_gNB
mis_classification	0.05	0.05
running_time(secs)	3.87	0.01

Based on the above comparison tables, we can conclude that the implemented Multinomial Naive Bayes does not perform well in terms of runningg times. The predictivity of this model is still the same as R build-in function, while the running times are 10 times longer than it.

Bernoulli Naive Bayes – Congressional Voting Records Dataset

We use Congressional Voting Records Dataset("Congressional Voting Records" 1987) to compare the implemented Bernoulli Naive Bayes with the R build-in bernoulli_naive_bayes() in naivebayes package. The dataset is downloaded from UCI Machine Learning Repository. It contains 17 variables and 435 records. The first variable is the response variable with 2 levels: democrat and republican. The 2 levels mean that the congressman's parties. The remaining variables are all binary, representing the different votes they made. The value of "f" represents three types of votes, including voted for, paired for, and announced for. The value of "n" denotes the opposite three types of votes: voted against, paired against and announced against. There are 203 records contain missing values, while the missing values mean that they are voted present, or are voted present to avoid conflict of interest, or did not vote or otherwise make a position known.

We simply exclude those records that contain missing values for convenience. The value of "f" is encoded as 1, while the value of "n" is encoded as 0. Then, we split the dataset into training and test sets with a ratio of 4:1.

```
votes <- read.csv("house-votes-84.data"
                  , header=FALSE, na.strings="?"
                  , stringsAsFactors=TRUE)

# remove missing values
votes <- na.omit(votes)
```

```
colnames(votes) <- c("class", "handicapped-infants"
                     , "water_project_cost_sharing"
                     , "adoption_of_the_budget_resolution"
                     , "physician_fee_freeze", "el-salvador-aid"
                     , "religious_groups_in_schools"
                     , "anti_satellite_test_ban"
                     , "aid_to_nicaraguan_contras", "mx_missile"
                     , "immigration", "synfuels_corporation_cutback"
                     , "education_spending", "superfund_right_to_sue"
                     , "crime", "duty_free_exports"
                     , "export_administration_act_south_africa")
```

```
# encoding
# f = 1; n = 0
for(i in 2:ncol(votes)){
  votes[,i] <- as.integer(factor(votes[,i] == "y")) - 1
}
```

```
set.seed(1)

# split
trainVotes_ind <- sample(nrow(votes), 0.8 * nrow(votes))
trainVotes <- votes[trainVotes_ind, ]
testVotes <- votes[-trainVotes_ind, ]
```

```
# implemented Bernoulli Naive Bayes
start_time_bNB <- Sys.time()
votes_bNB <- naiveBayes3(df=trainVotes, y="class", alpha=1
                         , test_set=testVotes[,-1]
                         , test_ind=testVotes[,1]
                         , method = "Bernoulli")
```

```
end_time_bNB <- Sys.time()
running_time_bNB <- end_time_bNB - start_time_bNB
```

```
trainVotes_ind <- unlist(trainVotes[,1])
testVotes_R <- as.matrix(testVotes[,-1])

start_time_bNBR <- Sys.time()

# R build-in Bernoulli Naive Bayes
votes_bNBR <- bernoulli_naive_bayes(x=trainVotes[,-1], y= trainVotes_ind, laplace = 1)


votes_bNBR_pred <- predict(votes_bNBR
                           , newdata = testVotes_R
                           , type = "class")

tabVotesR <- table(votes_bNBR_pred, testVotes[,1])

end_time_bNBR <- Sys.time()
running_time_bNBR <- end_time_bNBR - start_time_bNBR
```

```
# Contingency table of implemented Gausian Naive Bayes
votes_bNB[["table"]]
```

```
            test_ind
pred         democrat republican
  democrat         28          1
  republican        2         16
```

```
# Contingency table of R build-in Gaussian Naive Bayes
tabVotesR
```

```
votes_bNBR_pred democrat republican
```

```
democrat          28           1
republican         2          16
```

```
missVoteR <- round(1-classAgreement(tabVotesR)$diag,2)
missVotegNB <- round(votes_bNB[["mis_classification"]],2)
running_time_bNB <- round(running_time_bNB*60,2)
running_time_bNBR <- round(running_time_bNBR,2)
compare_bNB <- data.frame(implemented_gNB = c(missVotegNB, running_time_bNB)
                          , R_build_in_gNB = c(missVoteR, running_time_bNBR))
rownames(compare_bNB) <- c("mis_classification", "running_time(secs)")
```

Table 3: Comparison table for Bernoulli Naive Bayes

	implemented_gNB	R_build_in_gNB
mis_classification	0.06	0.06
running_time(secs)	1976.49	0.01

Based on the above comparison tables, the implemented Bernoulli Naive Bayes performs good on classification, and has the same mis-classification rate with the R build-in function. However, the running time of the implemented Bernoulli Naive Bayes is too bad, as it takes over one minute to run.

Discussion

The implemented three types of Naive Bayes classifiers are all have bad running time than the R build-in functions. The prediction performances across all three implemented functions are the same as the R build-in functions.

To decrease the running time of three Naive Bayes classifiers, we can try to make the algorithms more concise. Especially, the Bernoulli Naive Bayes classifier takes the longest time to run and it has really slow performance when calculating the likelihoods following the Bernoulli distribution. The Multinomial Naive Bayes has a best performance on runing times with only about 4 seconds.

Among the three Naive Bayes classifiers implemented, all of them have longer running times compared to the R built-in functions. However, the prediction performance of all three implemented functions is the same to the prediction performance of the R built-in function. To reduce the running time of the Naive Bayes classifier, we can focus on simplifying the algorithm. In particular, the Bernoulli Naive Bayes classifier has the longest running times and is particularly slow in computing the likelihood according to the Bernoulli distribution. On the other hand, the Multinomial Naive Bayes classifier performs the best in terms of running time, taking only about 4 seconds.

Reference

Boos, Dennis D., and Douglas Nychka. 2022. *Rlab: Functions and Datasets Required for ST370 Class*. Manual. https://CRAN.R-project.org/package=Rlab.

"Congressional Voting Records." 1987. UCI Machine Learning. 10.24432/C5C01P.

Jayaswal, Vaibhav. 2020. "Laplace Smoothing in Naïve Bayes Algorithm." https://towardsdatascience.com/laplace-smoothing-in-naïve-bayes-algorithm-9c237a8bdece.

Karatzoglou, Alexandros, Alex Smola, and Kurt Hornik. 2023. *Kernlab: Kernel-based Machine Learning Lab*. Manual. https://CRAN.R-project.org/package=kernlab.

Karatzoglou, Alexandros, Alex Smola, Kurt Hornik, and Achim Zeileis. 2004. "Kernlab – an S4 Package for Kernel Methods in R." *Journal of Statistical Software* 11 (9): 1–20. https://doi.org/10.18637/jss.v011.i09.

Kovachi, J. M. 2018. "Implementing a Multinomial Naive Bayes Classifier from Scratch with Python." https://medium.com/@johnm.kovachi/implementing-a-multinomial-naive-bayes-classifier-from-scratch-with-python-e70de6a3b92e.

Majka, Michal. 2019. *Naivebayes: High Performance Implementation of the Naive Bayes Algorithm in R*. Manual. https://CRAN.R-project.org/package=naivebayes.

Meyer, David, Evgenia Dimitriadou, Kurt Hornik, Andreas Weingessel, and Friedrich Leisch. 2022. *E1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*. Manual. https://CRAN.R-project.org/package=e1071.

ranga_vamsi. 2020. "Naïve Bayes Algorithm -Implementation from Scratch in Python." https://medium.com/@rangavamsi5/naïve-bayes-algorithm-implementation-from-scratch-in-python-7b2cc39268b9.

Wickham, Hadley, Romain François, Lionel Henry, Kirill Müller, and Davis Vaughan. 2023. *Dplyr: A Grammar of Data Manipulation*. Manual. https://CRAN.R-project.org/package=dplyr.

Xie, Yihui. 2015. *Dynamic Documents with R and Knitr*. Second. Boca Raton, Florida: Chapman and Hall/CRC. https://yihui.org/knitr/.