Midterm Project Report, Bayes Rules Classifier

Tao(Richie) Lin

October 26, 2020

1 Abstract

This paper is an experiment report from the Machine Learning class with Professor Haralick. By researching, designing, and implementing the experiment, the goal is to understand better the mechanism, application, and limitation of the Bayes rule classifier. The experiment was divided into three major parts: 1) design a generator that creates a random multinomial one-dimensional array. The measurement of space and the class will be made from this array. 2) design a series of programs that implement the Bayes rule classifier with economic gain. 3) design an optimization process and evaluate the result. The result shows that the Bayes conditional probability has a strong correlation to the class having the highest probability when the data is randomly generated. The optimization has small or no impact on the scenario that the score of incorrect assignment is higher than the correct ones in the economic gain matrix.

2 Introduction

As an extremely important component of modern statistics, Bayes Theorem has provided a powerful angle looking at the world with the conditional probability. Additional to the contribution to the statistics, it is also a widely used algorithm in the field of machine learning. In the class of Machine Learning, Professor Haralick has instructed us to accomplish a series of experiments, including understanding the theory of Bayes Theorem, design the experiment procedures, implementing the coding process, evaluating the result.

3 Experiment Process

As a person who started building skillsets from statistical programming, my primary programming language is R. when working on this project. There are a couple of built-in functions in R that provided the short cuts to accomplish the functions that might need a lot of additional steps to do, such as generating uniformly distributed arrays, vectorized computation instead of interactive computation, and Boolean masking subsetting instead of iterative comparison. The experiment was completed in three major steps: generating random multinominal datasets for training and testing; building a program that calculates the expectation of the economic gain based on the Bayes rule; Optimization by adding the marginal change on the correct assignment for a set number of iterations. As one of the characteristic logics in R programming or any statistics programming language, it would be easier to manipulate a dataset rather than building the program from zero. In such a case, I started with making the synthetic dataset generator.

3.1 Building synthetic dataset generator

The synthetic dataset generator is designed to generate a random one-dimensional array with two core parameters: the number of possible values (or class if generating the dependent variable) and the number of observations. The number of possible values Y is used to generate a uniformly distributed array with the range from 0 to 1. The total number of elements in Y. We normalized the array by dividing each element by the sum of them. We fed it into a loop to calculate an array of cumulative probability. The newly generated array will be used as the upper limit of the category.

```
> Y = 3
>   # generate a uniform distribution from 0 to 1, of Y number of elements
>   q_Y = runif(Y)
> q_Y
[1] 0.9760163 0.2848961 0.9788667
>   # normalize the probability
>   q_Y = q_Y / sum(q_Y)
> q_Y
[1] 0.4357645 0.1271983 0.4370372
>   # loop through the distribution, generate the cummulative distribution
>   for (i in 1:length(q_Y)) {
+     if (i == 1) {
+       p_Y = q_Y[i]
+     } else {
+       p_Y = c(p_Y, p_Y[length(p_Y)] + q_Y[i])
+     }
+   }
> p_Y
[1] 0.4357645 0.5629628 1.0000000
```

In the screenshot of n example above, the total number of element Y was set as 3. The uniform distributed array q_Y has a value of 0.9356429, 0.6021019, and 0.4127837. By dividing the sum of all of them, the normalized array has the values of 0.4357645, 0.1271983, 0.4370372. Finally, the cumulative distribution calculated from the loop is 0.4357645, 0.5629628, 1.0000000, which serves as the upper limit of each assignment window. The next step is to input the number of observations, size and generated a uniformly distributed array, d, with the range from 0 to 1 and total elements as size. Each element of d will be fed into a loop of comparison with the values p_Y. In the example of the screenshot above, if the values are greater than 0 but smaller or equal to 4.357645, the label of this element will be assigned as 1; if it is great than the previous upper limit and smaller than 0.5629628, the label will be assigned as 2; same applies to the third circumstances, which the label of the element will be assigned as 3. When the iteration is completed, we will have a one-dimensional array with the possible number value of Y, and the total number of the element as size, the probability of each label being assigned is p_Y. However, as a relatively high level of programming language, R provides a vectorized computational logic, which allows users to apply the manipulation on the entire array level rather than looping through each of them. That character makes our functions hit the performance issue when the size was set more than 1 million. However, when the number of observations in the measurement space was not big enough (in my case, I would prefer to set it at 3 million), some x tuple would not have enough samples to calculate a meaningful prior conditional probability. In such consideration, I decided to go ahead and use one of the powerful preset functions in R: sample(). It generates a randomly distributed one-dimensional array when the given number of possible values of labels, number of observations, and the probability of each possible values.

```
## set the random seed
set.seed(78901)

## set the number of observation as 3 millions
ob = 3000000

## generate the class y and the measurement space of Xs, each of the X will be one feature in the model and will be combined as tuples in the calculation
y <- sample(x = c(1:4), size = ob,
            prob = c(0.25, 0.25, 0.2, 0.3), replace = T)
x1 <- sample(x = c(1:2), size = ob,
             prob = c(.3, .7), replace = T)
x2 <- sample(x = c(1:6), size = ob,
             prob = c(.1, .1, .3, .2,.1,.2), replace = T)
x3 <- sample(x = c(1:3), size = ob,
             prob = c(.1, .19 ,.71), replace = T)
x4 <- sample(x = c(1:3), size = ob,
             prob = c(.57, .23, .2), replace = T)
x5 <- sample(x = c(1:4), size = ob,
             prob = c(.24, .13, .27, .36), replace = T)
x6 <- sample(x = c(1:2), size = ob,
             prob = c(.3, .7), replace = T)


df <- data.frame(y, x1, x2, x3, x4, x5, x6)
```

In the screenshot above, I generated a mea-

surement space with six features: x1 – x6 with the same number of observations but different numbers and probability of the possible labels. They are combined in a data frame with the true class array y.

3.2 Calculating Bayes rule

From the slides, I learned to form the professor that the Bayes rules that are more generally applicable to real-world problem solving are considering the economics gains for every assignment. It was a fascinating idea to me and elevated my level of thinking in applying machine learning techniques in real-world problem-solving. So in building the Bayes rules, I divided the constructions into three parts:

- Building a function that calculates the prior conditional probability $P(d|c)$ and the class probability $P(c)$;
- Building a function that calculates the Bayes posterior conditional probability $P(c|d)$
- Multiply the economics gain to each assignment class's probability for every x tuple for the economic gains.

Before building these functions, I divided the dataset into a training set and testing set.

```
# Building traning and testing set for one time evaluation

## choosing the number of folds we want to divide the dataset on
n_fold = 10

##build an spliting index based on the number of folds assigned earlier
test_set_index <-
  ((i - 1) * 1 / n_fold * nrow(df) + 1) :
    ((i) * 1 / n_fold * nrow(df))

## use the index to split the traning and testing set
df_test <- df[test_set_index, ]
df_train <- df[-test_set_index , ]
```

This step, by providing the number of folds wanted, calculates the staring row and the ending row and saves it as a test set index. We can use the Boolean mask feature in R programming to subset the index's training set and testing set.

The function that calculates the prior conditional probability is called fun_prob() in my code. It takes a data frame with columns named Xn and Y and produces the table that contains the prior probability of each x tuple in every given y.

```
## build a function to calculate prior probability
fun_prob <- function(df){

  # combine all variables into a measurement tuple named
  df$x_tuple <- apply(df[, grep( "y|tuple", names(df),
invert = T)],MARGIN = 1, paste, collapse = ".")

  ob <- nrow(df)

  # use the table function in R to generate the count of
each given y,
  df_cnt <- as.data.frame(table( df$y, df$x_tuple))
  df_cnt <- data.frame(df_cnt, prob_d_c = rep(NA, nrow
(df_cnt)))
  names(df_cnt) <- c("y", "x_tuple", "cnt", "prob_d_c")

  # calculate the prior probability of given class
  for (i in unique(df_cnt$y)) {
    df_cnt[df_cnt$y ==i, "prob_c"] <- sum(df_cnt[df_cnt$y
==i, "cnt"]) / ob
    for (j in unique(df_cnt$x_tuple)){
      df_cnt[df_cnt$x_tuple ==j, "prob_d_c"] <-
        sum(df_cnt[df_cnt$x_tuple ==j, "cnt"]) / sum
(df_cnt[df_cnt$y ==i, "cnt"])
    }
  }


  out <- df_cnt[order(df_cnt[,"x_tuple"], df_cnt[,"y"]),]
  rownames(out) <- 1:nrow(out)
  # the output of this prior conditional probability
calculation function returns 5 columns:
    ## Y as the true class value
    ## x_tuple as the unique x_tuple
    ## cnt as the count of the observations of each
x_tuple in each given y classes
    ## Prob_d_c as the probability of the x_tuple give
each classes
    ## Prob_c as the probablity of each class
  return(out)
}
```

The first step is to convert the measurement space to an x tiple by combining all measurement features Xs. There is a handy function called paste(), which string all elements into one string in R. Using the apply() function to cast the paste() on all the elements in each feature, it produces the x tuple with all features combined. Rather than counting the number of appearances in a loop, R has a great function table(), which produces the count of each unique x tuple. When given the second element to the function, in our

case, the true class label y, it can also produce the crosstab of x tuples and y. the result is shown below.

	y	x_tuple	cnt
1	1	1.1.1.1.1.1	80
2	2	1.1.1.1.1.1	134
3	3	1.1.1.1.1.1	56
4	4	1.1.1.1.1.1	76
5	1	1.1.1.1.1.2	175
6	2	1.1.1.1.1.2	363
7	3	1.1.1.1.1.2	83
8	4	1.1.1.1.1.2	143
9	1	1.1.1.1.2.1	45
10	2	1.1.1.1.2.1	76
11	3	1.1.1.1.2.1	19
12	4	1.1.1.1.2.1	39

Using the count from the crosstab, the conditional probability can be easily calculated simply by dividing each true class label's total count. Here is an example of the output:

y	x_tuple	cnt	prob_d_c	prob_c
1	1.1.1.1.1.1	80	0.0006351491	0.2253378
2	1.1.1.1.1.1	134	0.0006351491	0.4597456
3	1.1.1.1.1.1	56	0.0006351491	0.1131559
4	1.1.1.1.1.1	76	0.0006351491	0.2017607
1	1.1.1.1.1.2	175	0.0014024679	0.2253378
2	1.1.1.1.1.2	363	0.0014024679	0.4597456
3	1.1.1.1.1.2	83	0.0014024679	0.1131559
4	1.1.1.1.1.2	143	0.0014024679	0.2017607
1	1.1.1.1.2.1	45	0.0003285887	0.2253378
2	1.1.1.1.2.1	76	0.0003285887	0.4597456
3	1.1.1.1.2.1	19	0.0003285887	0.1131559
4	1.1.1.1.2.1	39	0.0003285887	0.2017607
1	1.1.1.1.2.2	92	0.0007232622	0.2253378
2	1.1.1.1.2.2	187	0.0007232622	0.4597456
3	1.1.1.1.2.2	51	0.0007232622	0.1131559
4	1.1.1.1.2.2	64	0.0007232622	0.2017607

This function is very useful to produce the elements in the Bayes theorem:

$$P(c_n|d) = \frac{P(d|c)P(c)}{P(d,c)} = \frac{P(d|c)P(c)}{\sum_{n=1}^{k} P(d|c_n)P(c_n))}$$

The numerator and denominator element in the expanded form of this conditional probability can be calculated with the function. Build the function to calculate the posterior conditional probability with the economic gain. Based on the Bayes conditional probability:

$$P(c_n|d) = \frac{P(d|c)P(c)}{P(d,c)} = \frac{P(d|c)P(c)}{\sum_{n=1}^{k} P(d|c_n)P(c_n))}$$

To calculate each possible assignment class's probability, we need to use the conditional probability multiply the class probability that we wanted to calculate from the training set in the numerator. The denominator would be the joint probability of all possible true class and the conditional probability of the x tuple from the testing set.

```r
fun_test <- function(df_test, prior_df = prior_df, show_e = F) {

  ## calculate the condistional probablity of the testing set with the same function
  test_meta <- fun_prob(df_test)

  # initiate a empty data frame for the result
  result <- data.frame()
  for (i in test_meta$x_tuple) {
    ## for each unique x tuple, multiply the prob_d_c for all values of class with the the probablity of each value of class to calculate the denominator of the bayes theoron
    tmp_set_denom <- test_meta[test_meta$x_tuple == i,]
    denom <- sum(tmp_set_denom$prob_d_c * tmp_set_denom$prob_c)

    ## for each of the values of the class in the traning set, we will subset the x tuple and desired values of class, multiply them to calculate the numerator of the bayes theoron calculation
    out <- data.frame()
    for (j in unique(test_meta$y)) {
      tmp_set_numer <- prior_df[prior_df$y == j & prior_df$x_tuple == i, ]
      numer <- tmp_set_numer$prob_d_c * tmp_set_numer$prob_c

      PT_C_comma_D <- numer / denom

      out <- rbind(out, data.frame(i, j, PT_C_comma_D))
    }
    ## the probablilty of all possible assigning class is calculated in the loop above and stored by stacking the result of all the x tuples into one data fame

    out <- out[order(out$j),]

    ## we multiply the probability of each possible asignment of a x tuple with the economics gain matrix to build the final bayes rule
    ## the result of each x tuple will be sum together as a evaluation of each possible assignment.
    e_out<- colSums(out$PT_C_comma_D * econ_gain)
    ## the assignment with the maximized economic gain will be chosen as the bayes rule class assignment
    e <- max(e_out)
    asgn_class <- which(e_out == max(e_out))

    ##the true class that have the highest probablity in each x tuple will also be saved for the optimization process.
    true_class <- which(tmp_set_denom$cnt == max(tmp_set_denom$cnt))

    # print(data.frame(i, e,asgn_class))


    # then we stack the result of each x tuple for the final output
    result <- rbind(result, data.frame(i, e, asgn_class, true_class))

  }
  result <- result[!duplicated(result),]

  names(result) <- c("x_tuple", "e", "asgn_class", "true_class")

  ## the sum of the expected economics gain is calculated for the evaluation and optimization
  E = sum(result$e)

  ## adding an option to show either the result table of the assignment, or the sum of the economics gain
  if (show_e){
    return(E)
  } else {
    return(result)
  }

}
```

Because the testing set is involved in this calculation step, I named it fun_test(). The first step in this function is to calculate the conditional probability of the testing set. It can be done easily with the function we built earlier. A nested loop is then used to calculate each possible class assignment's probability on each x tuples. Firstly loop through all the x tuples, subset the conditional probability from the training set to calculate the denominator as shared in the calculation in all possible, then loop through y to calculate the numerator in calculating the probability for each possible assignable class.

```
             i j PT_C_comma_D
1 1.3.2.2.4.2 1    0.2380484
2 1.3.2.2.4.2 2    0.4856785
3 1.3.2.2.4.2 3    0.1195387
4 1.3.2.2.4.2 4    0.2131415
```

The screenshot above is the calculation result of the x tuple 1.3.2.2.4.2, given the possible assignment class 1,2,3 and 4. After stacking up the results for each possible assignment class, we multiply the array of probability to the economic gain matrix.

```
> out
             i j PT_C_comma_D
1 1.3.2.2.4.2 1    0.2380484
2 1.3.2.2.4.2 2    0.4856785
3 1.3.2.2.4.2 3    0.1195387
4 1.3.2.2.4.2 4    0.2131415
> ## or for more complex simulation, made a 6 * 6 matrix
 with random selected econ gain scores
> econ_gain <- matrix(
+     c(2,2,2,2,
+       0,2,1,0,
+       3,0,3,0,
+       1,1,0,1), ncol = 4, nrow = 4, byrow = T)
> out$PT_C_comma_D * econ_gain
          [,1]      [,2]      [,3]      [,4]
[1,] 0.4760969 0.4760969 0.4760969 0.4760969
[2,] 0.0000000 0.9713570 0.4856785 0.0000000
[3,] 0.3586162 0.0000000 0.3586162 0.0000000
[4,] 0.2131415 0.2131415 0.0000000 0.2131415
> colSums(out$PT_C_comma_D * econ_gain)
[1] 1.0478546 1.6605954 1.3203916 0.6892384
```

By summing up the economic gain, we chose the assignment with the highest overall economic gain. In the example shown above, each possible assignment's economic gain was calculated as 1.04, 1.66, 1.32, and 0.69. in the example above, the x tuple of 1.3.2.2.4.2 will be assigned to class 2 for having the highest economic gain. After calculating the economic gain for all x tuples, we store the assigned class and the true class, which has the highest P (d —c) probability for the optimization process. Along with these probabilities, we also sum the total economic gain for all x tuples for the evaluation. The result set of the function looks like the screenshot below:

x_tuple	e	asgn_class	true_class
1.1.1.1.1.1	1.509219	2	2
1.1.1.1.1.2	1.606022	2	2
1.1.1.1.2.1	1.419602	2	2
1.1.1.1.2.2	1.636753	2	2
1.1.1.1.3.1	2.121855	2	2
1.1.1.1.3.2	1.581422	2	2
1.1.1.1.4.1	1.215881	2	2
1.1.1.1.4.2	1.372947	2	2
1.1.1.2.1.1	1.946080	2	1
1.1.1.2.1.2	1.623598	2	1
1.1.1.2.1.2	1.623598	2	2
1.1.1.2.2.1	2.337981	2	1
1.1.1.2.2.1	2.337981	2	2
1.1.1.2.2.2	1.434582	2	2
1.1.1.2.3.1	2.108254	2	1
1.1.1.2.3.2	1.995571	2	2
1.1.1.2.4.1	1.808208	2	2
1.1.1.2.4.2	1.459255	2	2
1.1.1.3.1.1	1.866895	2	1
1.1.1.3.1.1	1.866895	2	2
1.1.1.3.1.1	1.866895	2	3

Most of the x tuple in the example above got assigned as class 2 because the overall probability of class 2 is the highest (0.4596163 vs. 0.2251065, 0.1132197, and 0.2020574) also. In the economic gain matrix, the reward of assigning it to class 2 is also pretty high.

3.3 Optimization

The assignment could be rewarded by setting a marginal change value and adding it to the correct assignment's conditional probability, thus further optimized. Here is an example code of optimization In my experiment:

```
## Optimization
# set the number of iteration wanted
iter <- 12

for (i in 1:iter) {
  ## if it is the first round of iteration, set the prior probability to the initial calculation from the training set
  if (i == 1) {
    prior_df = fun_prob(df_train)
  } else {
    ## set a change margin of the probabilityin the prior condistional probability, as instructed by the professor, it should be less than 0.05
    delta <- 0.02

    ## use the testing function built earlier to build the bayes rule classifier
    delta_set <-
      fun_test(df_test = df_test,
               prior_df = prior_df,
               show_e = F)

    ## merge the subset the calculated bayes rule result that had a incorrect assignment, then increase the conditional probability of these x tuples by the marginal change delta
    prior_df <-
      merge(
        prior_df,
        delta_set[delta_set$asgn_class == delta_set$true_class,],
        by.x = c("x_tuple", "y"),
        by.y = c("x_tuple", "true_class"),
        all.x = T
      )

    prior_df$prob_d_c[!is.na(prior_df$asgn_class)] <-
      prior_df$prob_d_c[!is.na(prior_df$asgn_class)] + delta

    prior_df <-
      prior_df[, c('x_tuple', 'y', "cnt", "prob_d_c", "prob_c")]
    # then normalize the condistional probablity by dividing the sum of all possible values of the class
    for (j in unique(prior_df$y)) {
      prior_df[prior_df$y == j, "prob_d_c"] <-
        prior_df[prior_df$y == j, "prob_d_c"] / sum(prior_df[prior_df$y == j, "prob_d_c"])
    }

  }


  result <- list()
  result[[i]] <- fun_test(df_test = df_test,
                          prior_df = prior_df,
                          show_e = F
  )
  print(sum(result[[i]]$asgn_class == result[[i]]$true_class) / nrow(result[[i]]))
  print(sum(result[[i]]$e))
}
```

First, set a number of iteration. I choose 12

because it reaches the max expected gain after several rounds of experiments. For the first round of the iteration, the prior conditional probability is calculated with the function built earlier. For each round of iteration following, the prior conditional probability is inherited from the previous round. By looking at only the correct assignment from the Bayes rule assignment result, we can now target the x tuple and the correctly assigned y class. Merge the subset with the prior probability table as a filter, add the delta to the prior conditional probability, and then normalize it to make the probability of given x tuple in all y classes sum to 1. By doing so, the correct assignment will be rewarded by the marginal value in each iteration.

Here we have completed the experiment with the Bayes rule classifier.

4 Experimental Result & Conclusions

1. When all the variables were randomly generated and uniformly distributed, the calculated posterior conditional distribution will have a strong correlation with the true class with the highest probability.

2. Because each of the x tuples can be assigned to only one class when the dataset is randomly uniformly distributed, the maximum correct assignment rate would never reach 100

3. Though the correct identification rate's peak will be reached after a few iterations of the optimization process, the expected gain's peak will continue to grow. The optimization result below was from a Bayes rule classifier with the economics gain as an identity matrix with dimensions equals the number of possible classes.

```
"iteration: 1"
"Correct assignment rate: 0.789247311827957"
"Expected gain: 1986.22009557231"
"iteration: 2"
"Correct assignment rate: 0.648387096774194"
"Expected gain: 285.894001610595"
"iteration: 3"
"Correct assignment rate: 0.639784946236559"
"Expected gain: 339.517876249332"
"iteration: 4"
"Correct assignment rate: 0.773118279569892"
"Expected gain: 863.95991484859"
"iteration: 5"
"Correct assignment rate: 0.795698924731183"
"Expected gain: 1307.44827562349"
"iteration: 6"
"Correct assignment rate: 0.795698924731183"
"Expected gain: 1581.45965825494"
"iteration: 7"
"Correct assignment rate: 0.795698924731183"
"Expected gain: 1721.79506304606"
"iteration: 8"
"Correct assignment rate: 0.795698924731183"
"Expected gain: 1805.50101766355"
"iteration: 9"
"Correct assignment rate: 0.795698924731183"
"Expected gain: 1860.07065415301"
"iteration: 10"
"Correct assignment rate: 0.795698924731183"
"Expected gain: 1897.22885757077"
```

4. The economic gain has a significant impact on the assignment. When rewarding the incorrect assignment is multiple times bigger than the correct assignment, it could potentially impair the optimization for the correct assignment rate. Show as the example below:

```
> econ_gain
     [,1] [,2] [,3] [,4]
[1,]  1.2    0    0    0
[2,]  1.2    1    0    0
[3,]  1.2    0    1    0
[4,]  1.2    0    0    1
```

```
"iteration: 1"
"Correct assignment rate: 0.146236559139785"
"Expected gain: 1143.52518310091"
"iteration: 2"
"Correct assignment rate: 0.146236559139785"
"Expected gain: 1143.52518310091"
"iteration: 3"
"Correct assignment rate: 0.146236559139785"
"Expected gain: 691.869140639539"
"iteration: 4"
"Correct assignment rate: 0.146236559139785"
"Expected gain: 1004.39677266129"
"iteration: 5"
"Correct assignment rate: 0.146236559139785"
"Expected gain: 1088.40957696822"
"iteration: 6"
"Correct assignment rate: 0.146236559139785"
"Expected gain: 1110.9936641475"
"iteration: 7"
"Correct assignment rate: 0.146236559139785"
"Expected gain: 1117.06465532472"
"iteration: 8"
"Correct assignment rate: 0.146236559139785"
"Expected gain: 1118.69664220032"
"iteration: 9"
"Correct assignment rate: 0.146236559139785"
"Expected gain: 1119.13534834967"
"iteration: 10"
"Correct assignment rate: 0.146236559139785"
"Expected gain: 1119.25328011025"
```