Logistic regression and Naive Bayes

Erik Van Slyke, Jay Patel
CS 4375.501
Dr. Karen Mazidi
2/16/2020

Logistic Regression:

Logistic regression is a classification algorithm used to assign observations to a discrete set of classes. It is a predictive analysis algorithm and based on the concept of probabilities. In logistic regression the target variable is qualitative, where we want to know what class an observation is in. The target variable is a binary output so that we classify into one class or the other.

R script:

For the R script, the data used was plasma data from library HSAUR, which is a build in R dataset. We want to learn to predict if ECR>20 or not, based on the level of fibrinogen and globulin. Value greater than 20 indicate some possible associations with various health related problems.

The logistic model glm1 uses only fibrinogen as a predictor. After outputting the glm1 model, we see various parameters such as Deviance residuals, coefficients with statistical significance metrics and metrics of model. Residual deviance had lower value than Null deviance which means that the predictors considerably helped the model to reduce errors. Then, we define our sigmoid/logistic function that will take an input matrix and return a vector of sigmoid values for each observation. We initialize Wo and W1 to 1. The data matrix defined has two columns, column 1 is all 1s that will be multiplied by the intercept, and column 2 is fibrinogen which will be multiplied by w1. The code below iterates 500,000 times. In each iteration, it multiplies the data by the weights to get the log likelihood, then runs these values through the sigmoid() function to get a vector of probabilities. It computes the error and then update weights by weight+learning rate*gradient. At last R code outputs the weights. The plot **plot(plasma$fibrinogen, plasma_log_odds, col=plasma$ESR)** in the R script confirms the equation w0 +w1x.

The runtime is calculated using proc.time(). The running time of the R script was 3766 milliseconds. For data exploration, various plots, histograms and correlation values have been shown to understand the given dataset.

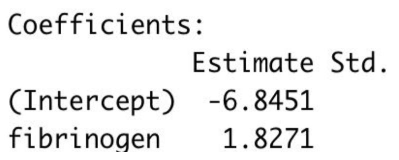
C++ code:

Finding the classifier included using gradient descent in order to compute the weights of the model. In order to find more accurate coefficients for the model we must iterate as much as possible, which takes up most of the execution time. To compute the matrices we needed to implement a matrix multiply method and a transpose matrix method, which I did from scratch. The running time for the C++ implementation is 523 milliseconds which is about 7 times faster than the R script. To calculate these runtimes I used a library called chrono which includes a function high_resolution_clock::now() that returns the system time of the machine down to microseconds. Then taking the difference of the times and outputting it returns the total execution.

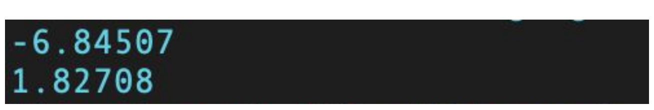
The weights are a 2x1 matrix, which is just a 2d array initialized to one, in which the model is used to be built upon. The data matrix is a Nx2 matrix where N is the number of data points used to train on, the first column is filled with ones and the second filled with the predictor points. The labels vector is just the target values. The inside the loop used to build the model we include the following. Calculate the matrix after multiplying the data matrix with the weights, and then passing that to sigmoid functions and passing that output to the prob_vector. Then the error vector is calculated by subtracting the prob_vector from the labels vector. The new weights for the next generation are then updated by transposing the datamatrix, then multiplying it with the error vector. Multiply that with the learning rate and adding the old weights will give you the new weights. After 500000 runs you will get very accurate weights for the model.

Result:

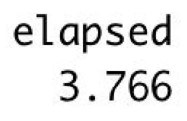
R **C++**



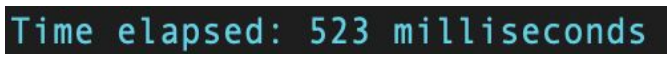
```
Coefficients:
             Estimate Std.
(Intercept)  -6.8451
fibrinogen    1.8271
```



```
-6.84507
1.82708
```



```
elapsed
  3.766
```



```
Time elapsed: 523 milliseconds
```

Naive Bayes:

The Bayes theorem let us examine the probability of an event based on the prior knowledge of any event that related to the former event. Naive Bayes is a popular classification algorithm. The naive Bayes algorithm reduces the complexity of Bayesian classifier by making an assumption of conditional independence over the training dataset.

R script:
In the R script, we started by reading the titanic_project.csv and saved it as df. The df has four characteristics, pclass, survived, sex and age. The data is divided into train and test with first 900 rows in train dataset and rest in test data set. The naive bayes model nb1 is made on train dataset using library e1071 and survived~pclass+sex+age. From the output of the nb1, we can see

parameters such as A-prior and conditional probabilities. The age variable is continuous. The mean for not surviving is 30.4 and for surviving is 28.9. Then we predict the model nb1 with our test dataset. After the test evaluation, the accuracy is 76.02 %, sensitivity is 0.873, and specificity is 0.626. For data exploration, various histograms and plots are shown to understand the nature of data.

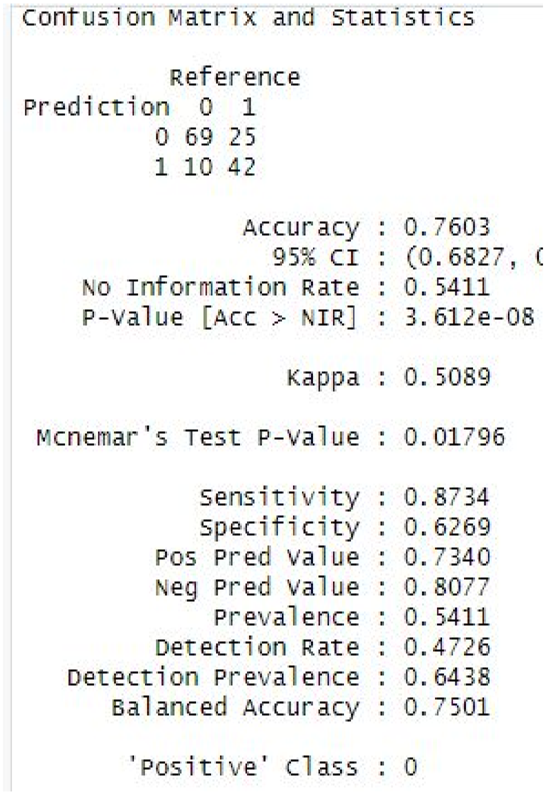
The proc.time() is used to calculate the time. Time elapsed for naive bayes model is 2.12 millisecond.

C++ code:
The C++ implementation of the naive bayes followed the from scratch code closely, using arrays to store the vectors. The output was the same as the Rscript. The execution time of the C++ was very fast, only taking 702 microseconds which is a very significant speedup from the R script taking 2120 milliseconds. The time was measured the same exact way the logistic regression was measured. The time it took to build the model was used as the execution time.

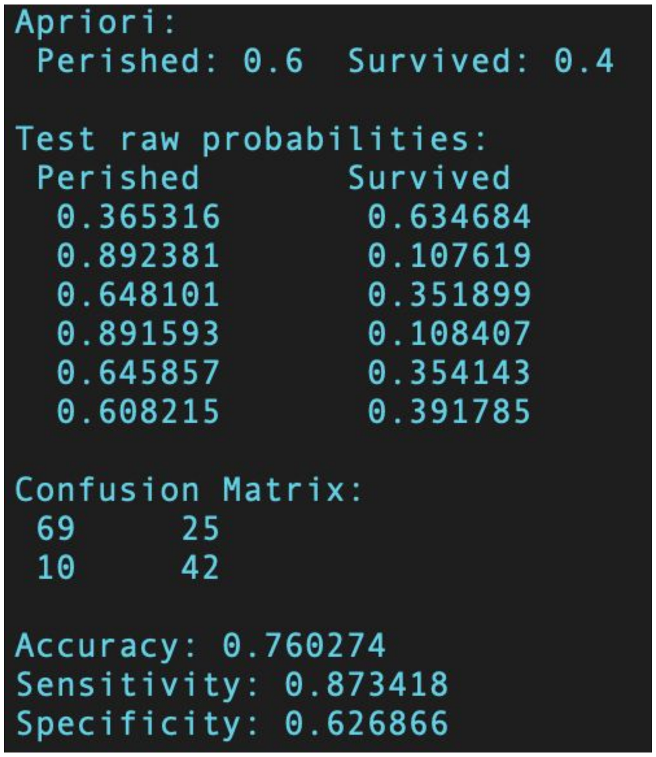
The algorithm starts off by calculating the apriori which was simple enough by averaging each survived and perished count to find the probabilities of each. Next we count the number survived and not survived and assign it to an array. Next we find the likelihoods of each of the predictors in the model. This is done by creating matrices that are NxM, where N is the amount of target variables, and M being the number of predictor variables. Each index in the likelihood array is the likelihood of that predictor variable surviving or not surviving. Then doing the age likelihood was different due to it being continuous. We need to find the average age for surviving and perishing by running through the data, same with the variance. We then can predict all of the raw probabilities for the test data. We multiply all of the likelihoods with the respective aprioris and then dividing by both to get the raw probability. The confusion matrix is done by comparing the raw probabilities to the actual test data to get the number of right and wrongs. The sensitivity, accuracy, and specificity are calculated by using their formulas on the true positives, false positives, false negatives and true negatives.

Result:

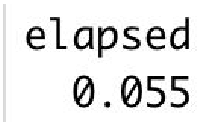
R **C++**



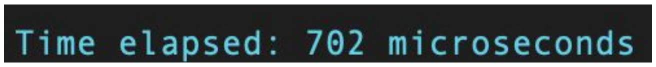
```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 69 25
         1 10 42

               Accuracy : 0.7603
                 95% CI : (0.6827, (
    No Information Rate : 0.5411
    P-Value [Acc > NIR] : 3.612e-08

                  Kappa : 0.5089

 Mcnemar's Test P-Value : 0.01796

            Sensitivity : 0.8734
            Specificity : 0.6269
         Pos Pred Value : 0.7340
         Neg Pred Value : 0.8077
             Prevalence : 0.5411
         Detection Rate : 0.4726
   Detection Prevalence : 0.6438
      Balanced Accuracy : 0.7501

       'Positive' Class : 0
```



```
Apriori:
 Perished: 0.6  Survived: 0.4

Test raw probabilities:
 Perished        Survived
  0.365316        0.634684
  0.892381        0.107619
  0.648101        0.351899
  0.891593        0.108407
  0.645857        0.354143
  0.608215        0.391785

Confusion Matrix:
 69     25
 10     42

Accuracy: 0.760274
Sensitivity: 0.873418
Specificity: 0.626866
```



```
elapsed
  0.055
```



```
Time elapsed: 702 microseconds
```