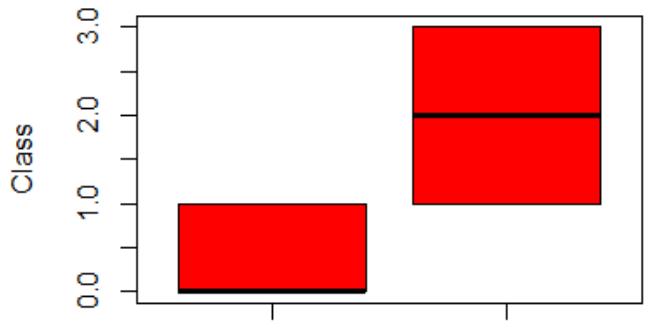
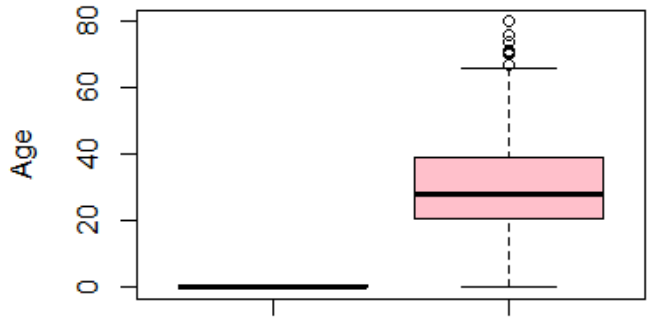
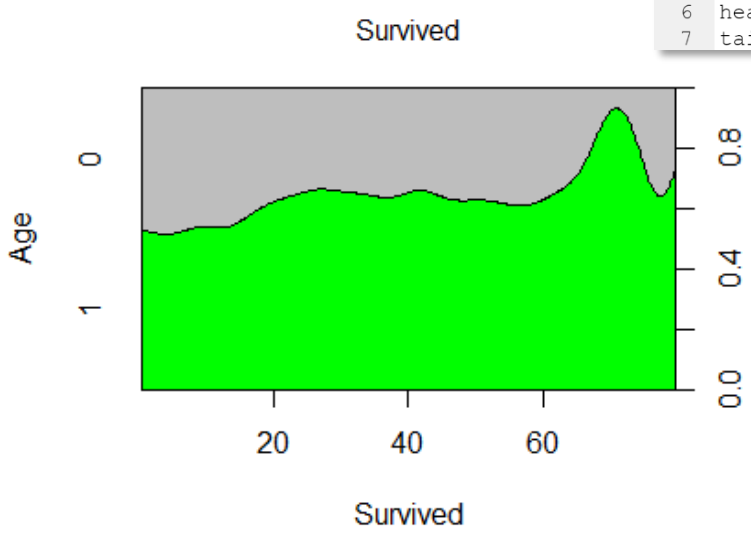
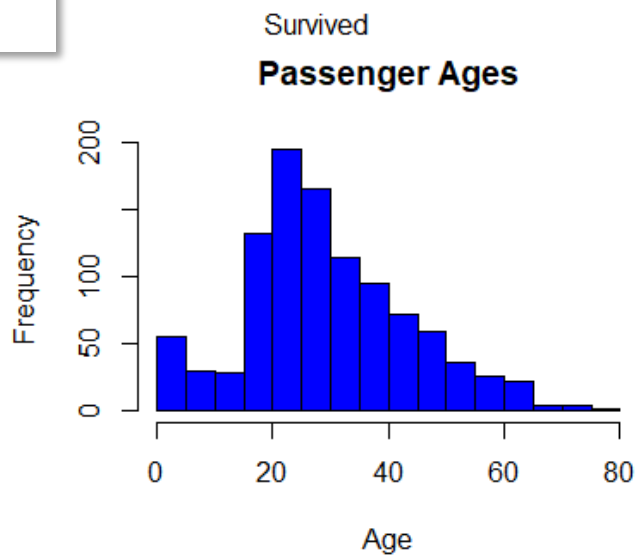
I. **Graphs**:






```
4  summary(df)
5  names(df)
6  head(df)
7  tail(df)
```





II. Logistic Regression:

A. Output:

R:

```
             Estimate Std.
(Intercept)   1.29717    0.
pclass       -0.77993    0.
```

```
pred  0  1
   0 67 36
   1 12 31
> tp <- calc_table[1]
> fn <- calc_table[2]
> fp <- calc_table[3]
> tn <- calc_table[4]
> sens <- (tp/(tp+fn))
> spec <- (tn/(tn+fp))
> # print out
> print(paste("Time taken: ", final_tir
[1] "Time taken:  0.895857095718384"
> print(paste("Accuracy: ", acc1))
[1] "Accuracy:  0.671232876712329"
> print(paste("Specificity: ", spec))
[1] "Specificity:  0.462686567164179"
> print(paste("Sensitivity: ", sens))
[1] "Sensitivity:  0.848101265822785"
>
```

C++:

```
1.29717
-0.779929
```

```
1 //========================================================
2 // Name        : logRegression.cpp
3 // Author      : Kathryn Kingsley KLK170230
4 // Version     : 1.0
5 // Copyright   : Spring 2021
6 // Description : Logistic regression in C++.
7 //========================================================
8
9 #include <iostream>
10 #include <fstream>
11 #include <chrono>
12 #include <math.h>
13 #include <string>
14 #include <C:\Eigen\Dense>
15
16 using Eigen::MatrixXd;
17 using namespace std;
18
19 //for me to follow my code
```

```
<terminated> (exit value: 0) logRegression.exe [C/C++ Application] C
The time taken was: 11.8912
The accuracy is: 0.671233
The sensitivity is: 0.462687
The specificity is: 0.848101
```

B. Assessment

Above are my output metrics for both programs for logistic regression. I created my model in C++ using the Eigen library to help with matrix multiplication. I started by getting all my data into various matrices, and I separated the train and test data immediately into their respective matrices. I then created a sigmoid function that would take a matrix of input, which turned out to be the matrix produced by multiplying data by my ever-changing weights, and return actual probabilities within the range [0,1]. Once that was completed, I created a gradient descent function to find the optimal weights using a learning rate of 0.001. I set the gradient descent function to loop through 50,000 times, and my weights came out the same in both R and C++. Both intercepts were 1.29717 and both pclass coefficients were -0.77993. The difference between linear regression and logistic regression is that logistic regression takes this linear model and assigns probabilities to the observations via the sigmoid function, therefore acting as a classifier. So once the sigmoid function calculated the probability, I put the observations into one of two categories. If the probability was less than 0.5, they were marked as not surviving, and marked as surviving otherwise. I then compared the actual survival rate to the survival rate predicted by my model and used this to make a confusion matrix like that in R. My C++ model was almost the same as the R model in accuracy, specificity, and sensitivity, but it was very different in regards to time.

To calculate the time in R, I started the clock just before calling glm() and stopped the clock immediately after the call. In C++, I used chrono to start the clock right before I called gradientDescent() to calculate my weights and stopped it right after the optimal weights were found. The R logistic regression model took less than a second at 0.896 seconds, and my C++ model took almost 12 seconds to build. Despite the metrics all being the same, I was truly surprised by how long this model took to build. Both R and C++ are interpreted languages, so I thought the times would be very similar, and if

anything, all of R's built logistic regression modeling features would make it slower than my from scratch model. My theory was wrong, however. I do not know if it was all the loops in my gradient descent method or if R uses a different optimization technique entirely, but 12 seconds compared to less than 1 is not even comparable.

III. **Naïve Bayes**

A. Output

R:

```
> print(paste("Time taken: ", final_time))
[1] "Time taken:  0.965866804122925"
> print(paste("Accuracy: ", acc))
[1] "Accuracy:  0.76027397260274"
> print(paste("Specificity: ", spec))
[1] "Specificity:  0.626865671641791"
> print(paste("Sensitivity: ", sens))
[1] "Sensitivity:  0.873417721518987"
>
```

```
A-priori probabilities:
Y
  0   1
0.6 0.4

Conditional probabilities:
   pclass
Y          1         2         3
  0 0.1685185 0.2203704 0.6111111
  1 0.4166667 0.2638889 0.3194444

   sex
Y          0         1
  0 0.1592593 0.8407407
  1 0.6944444 0.3055556

   age
Y       [,1]     [,2]
  0 30.41682 14.21185
  1 28.92060 15.09074
```

C++:

Problems Tasks Console

<terminated> (exit value: 0) naiveBayes.e

```
0.6 0.4
0.168519  0.22037 0.611111
0.416667 0.263889 0.319444
0.159259 0.840741
0.694444 0.305556
The accuracy is: 0.760274
The sensitivity is: 0.626866
The specificity is: 0.873418
The time taken was: 0.000997
```

B. Assessment

I used the Eigen library again for the Naïve Bayes implementation in C++. I divided the data set into train and test matrices, and then I started by calculating the percentage of perished and survived from my test set, called priors. Both R and C++ gave me 60% chance for perished and 40% chance for survived. I used the number of perished divided by the total number of observations. Next, I found the likelihood of the discrete variables, those that were not continuous. This was easy as well. I just counted how many of each class level were in survived and perished, and then divided that by survived or perished respectively. You can see my numbers for the likelihoods of pclass and sex in both the R implementation and the C implementation came out the same. A lot of the algorithm up to this point has been counting, so I used a lot of for loops. Calculating the age likelihood was more difficult since it is continuous. I had to first find the mean and variance. Then I

had to plug the actual age, the mean, and the variance into a formula to calculate the likelihood for any age given. Once all the likelihoods and priors had been calculated, I was ready to plug my test data into my calcRawProbs() function. This function took any give pclass, sex, and age, and would spit out both the probability that they survived and the probability that they perished. Then for comparison of my model with the actual survived or perished outcome, I sorted these probabilities. If the chance of perished was higher, they got a 0 and a 1 otherwise. The accuracy, specificity, sensitivity of the C++ mode exactly matched that of the R version.

I was surprised to see that the C++ algorithm was faster this time! In R, I started the clock just before calling glm() and stopped it directly after. In C++, I started the clock before calculating the apriori values and stopped it after all the likelihoods had been determined. In my code, that was lines 117 to 133. The from scratch version blew R's version out of the water. R clocked in at close to a second, but my implementation did not even get close to that-- about 0.001 seconds!

IV. Conclusion

Even though both R and C++ are interpreted languages, my C++ logistic regression time was very long compared to that of R, so I was certainly shocked to see how fast the Naïve Bayes implementation was. Both algorithms had quite a few for loops, but I think the matrix multiplication really slowed logistic regression down.

Prior to the project, I felt that I had a good understanding of logistic regression and Naïve Bayes machine learning algorithms, but that was not the case. I took for granted how simple R made it seem and really struggled to translate what I could do in R to C++. Now that I have completed this project, I truly feel like I have a firm grasp on both machine learning methods, and I think the amount of time I spent working on this project will help the material stick with me long term.