Untitled

September 26, 2019

```
In [1]: # install.packages('pROC')
```

```
package 'pROC' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Rishabh\AppData\Local\Temp\RtmpQzTfqZ\downloaded_packages
```

```
In [2]: library(pROC)
        library(randomForest)
```

```
Warning message:
"package 'pROC' was built under R version 3.6.1"Type 'citation("pROC")' for a citation.

Attaching package: 'pROC'

The following objects are masked from 'package:stats':

    cov, smooth, var
```

```
In [4]: set.seed(420)

In [5]: num.samples<-100

In [6]: # Average man weighs 172 pounds with a standard deviation of 29
        weight<-sort(rnorm(n=num.samples, mean=172, sd=29))

In [9]: weight
```

1. 86.4850488187779 2. 88.0476376160815 3. 111.500638013404 4. 112.697301041632
5. 121.539742996795 6. 122.345331253619 7. 126.533295630791 8. 129.345649093113
9. 129.462679155664 10. 130.173980117571 11. 133.139212421827 12. 133.946055934674
13. 135.808708594103 14. 135.976494512278 15. 137.528240104955 16. 137.530688006059
17. 137.848892655804 18. 137.94284146406 19. 138.34229928744 20. 138.73311849062
21. 140.70378082165 22. 141.303224982559 23. 143.975635912707 24. 144.835483630126
25. 146.469009135704 26. 146.808907189287 27. 149.752762486094 28. 150.360867470725
29. 150.557749722096 30. 152.86059319352 31. 153.602088114026 32. 153.956115962658

33. 155.768679272809 34. 155.995892196836 35. 157.070909503626 36. 157.383578998877 37. 158.645490087063 38. 159.749470190416 39. 161.954138179469 40. 162.950734173502 41. 163.674172527821 42. 163.895352911143 43. 163.926937579932 44. 165.31022856389 45. 165.650776311954 46. 167.331284791247 47. 167.945015182138 48. 168.584214275139 49. 169.39193694741 50. 171.076805195454 51. 173.599779446298 52. 173.657156963664 53. 174.273177037345 54. 175.456145882607 55. 176.283128068799 56. 176.808478231721 57. 176.841306528004 58. 177.298126189843 59. 178.253755914821 60. 178.264690168435 61. 178.380183161192 62. 179.243422237813 63. 179.763614815604 64. 179.785754966362 65. 180.453796644666 66. 181.666278365903 67. 182.68510435637 68. 183.541393664337 69. 185.340102984605 70. 185.793552158117 71. 186.097186578072 72. 186.618316100217 73. 187.034570213726 74. 187.546809134665 75. 187.866267778087 76. 189.114690925971 77. 189.289975572623 78. 189.537164016039 79. 189.630008405486 80. 189.830438712856 81. 189.966761875411 82. 190.286145444458 83. 193.365057551283 84. 194.115124514536 85. 194.447965709302 86. 194.795972526908 87. 194.957150202159 88. 195.213807755448 89. 197.124416042835 90. 199.217113516583 91. 200.194402786787 92. 202.670714463718 93. 203.664952537006 94. 204.95481805755 95. 206.787805425346 96. 210.724476356323 97. 212.068989131613 98. 218.332951870976 99. 227.101403689906 100. 227.682391423944

```
In [7]: obese<-ifelse(test=(runif(n=num.samples)<(rank(weight)/100)),yes = 1, no = 0)
```

```
In [8]: obese
```

1. 0 2. 0 3. 0 4. 0 5. 0 6. 0 7. 0 8. 0 9. 1 10. 1 11. 0 12. 1 13. 0 14. 0 15. 0 16. 0 17. 0 18. 0 19. 0 20. 0 21. 0 22. 0 23. 1 24. 0 25. 1 26. 1 27. 0 28. 0 29. 0 30. 1 31. 1 32. 0 33. 0 34. 1 35. 0 36. 0 37. 0 38. 0 39. 1 40. 1 41. 1 42. 0 43. 0 44. 1 45. 0 46. 0 47. 1 48. 0 49. 0 50. 1 51. 1 52. 1 53. 1 54. 0 55. 0 56. 1 57. 0 58. 0 59. 1 60. 1 61. 1 62. 0 63. 1 64. 1 65. 1 66. 0 67. 1 68. 0 69. 1 70. 1 71. 1 72. 0 73. 0 74. 1 75. 1 76. 1 77. 1 78. 1 79. 0 80. 1 81. 1 82. 1 83. 1 84. 1 85. 1 86. 1 87. 1 88. 1 89. 1 90. 1 91. 1 92. 1 93. 1 94. 1 95. 1 96. 1 97. 1 98. 1 99. 1 100. 1

As we can observe, the **lighter samples** are **mostly 0's (Not Obese)** and the **heavier samples** are **mostly 1's (not obese)**.

```
In [15]: rank(weight)/100
```

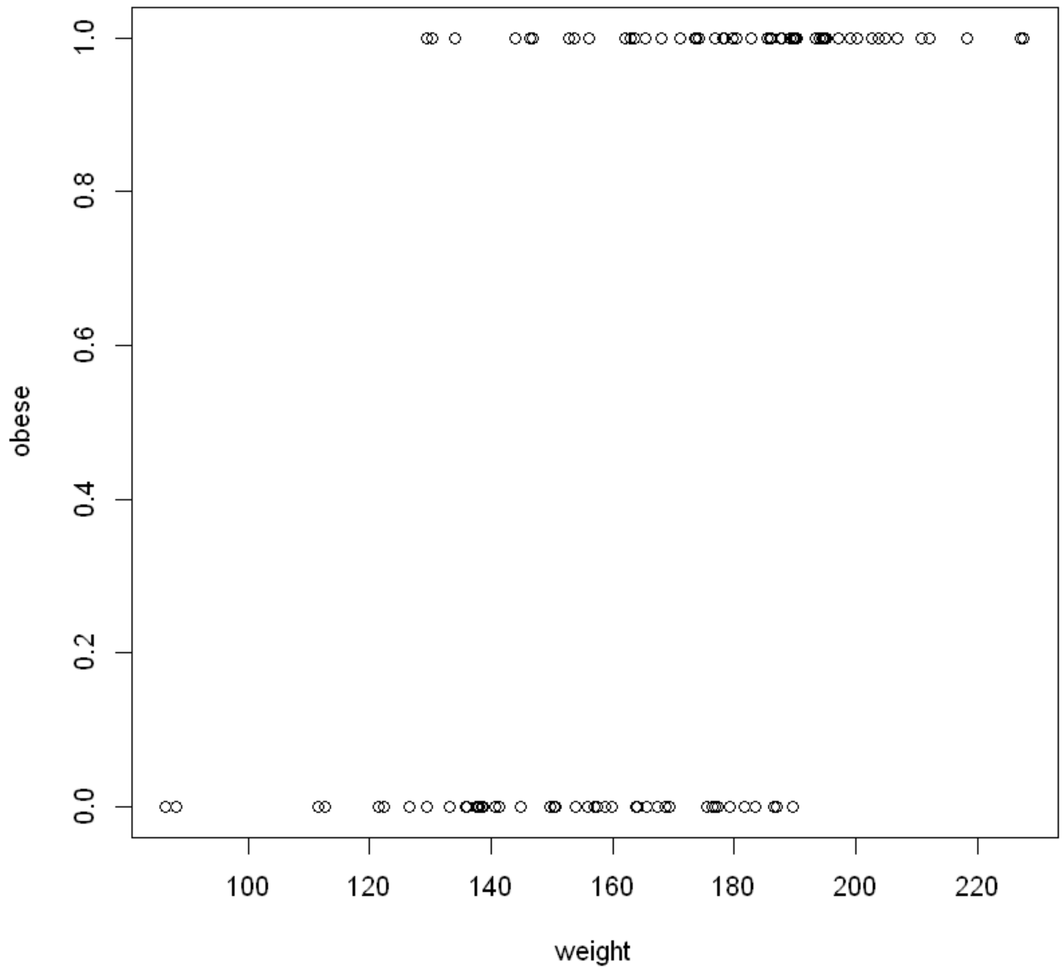
1. 0.01 2. 0.02 3. 0.03 4. 0.04 5. 0.05 6. 0.06 7. 0.07 8. 0.08 9. 0.09 10. 0.1 11. 0.11 12. 0.12 13. 0.13 14. 0.14 15. 0.15 16. 0.16 17. 0.17 18. 0.18 19. 0.19 20. 0.2 21. 0.21 22. 0.22 23. 0.23 24. 0.24 25. 0.25 26. 0.26 27. 0.27 28. 0.28 29. 0.29 30. 0.3 31. 0.31 32. 0.32 33. 0.33 34. 0.34 35. 0.35 36. 0.36 37. 0.37 38. 0.38 39. 0.39 40. 0.4 41. 0.41 42. 0.42 43. 0.43 44. 0.44 45. 0.45 46. 0.46 47. 0.47 48. 0.48 49. 0.49 50. 0.5 51. 0.51 52. 0.52 53. 0.53 54. 0.54 55. 0.55 56. 0.56 57. 0.57 58. 0.58 59. 0.59 60. 0.6 61. 0.61 62. 0.62 63. 0.63 64. 0.64 65. 0.65 66. 0.66 67. 0.67 68. 0.68 69. 0.69 70. 0.7 71. 0.71 72. 0.72 73. 0.73 74. 0.74 75. 0.75 76. 0.76 77. 0.77 78. 0.78 79. 0.79 80. 0.8 81. 0.81 82. 0.82 83. 0.83 84. 0.84 85. 0.85 86. 0.86 87. 0.87 88. 0.88 89. 0.89 90. 0.9 91. 0.91 92. 0.92 93. 0.93 94. 0.94 95. 0.95 96. 0.96 97. 0.97 98. 0.98 99. 0.99 100. 1

```
In [26]: set.seed(100)
         runif(100)
```

1. 0.307766110869125 2. 0.257672501029447 3. 0.552322433330119 4. 0.0563831503968686 5. 0.468549283919856 6. 0.483770735096186 7. 0.812402617651969 8. 0.370320537127554 9. 0.546558595029637 10. 0.170262051047757 11. 0.624996477039531 12. 0.882165518123657

13. 0.28035383997485 14. 0.398487901547924 15. 0.76255108229816 16. 0.669021712383255
17. 0.204612161964178 18. 0.357524853432551 19. 0.359475114848465 20. 0.690290528349578
21. 0.535811153938994 22. 0.710803845431656 23. 0.538348698290065 24. 0.74897222686559
25. 0.420101450523362 26. 0.171420212602243 27. 0.770301609765738 28. 0.881953587755561
29. 0.549096710281447 30. 0.277723756618798 31. 0.488305994076654 32. 0.928505074931309
33. 0.348691981751472 34. 0.954157707514241 35. 0.695274139055982 36. 0.889453538926318
37. 0.180407245177776 38. 0.629390850430354 39. 0.989564136601985 40. 0.130288870073855
41. 0.330660525709391 42. 0.865120546659455 43. 0.777584439376369 44. 0.827303449623287
45. 0.603324356488883 46. 0.491231821943074 47. 0.780358511023223 48. 0.884227027418092
49. 0.207713897805661 50. 0.307085896842182 51. 0.330529848113656 52. 0.19867907022126
53. 0.235694302013144 54. 0.274886660277843 55. 0.591321053681895 56. 0.253390653757378
57. 0.12348723039031 58. 0.229905887041241 59. 0.597575292224064 60. 0.211408555973321
61. 0.463701178086922 62. 0.647101194132119 63. 0.960573092103004 64. 0.676398171577603
65. 0.445148021681234 66. 0.35777378291823 67. 0.455731456167996 68. 0.44541397690773
69. 0.245092589175329 70. 0.694350712001324 71. 0.412237035110593 72. 0.327725868672132
73. 0.57256476697512 74. 0.966999084455892 75. 0.661779022077098 76. 0.624697716906667
77. 0.856653042603284 78. 0.77477888809517 79. 0.834027098724619 80. 0.0915102786384523
81. 0.459525486687198 82. 0.599398155929521 83. 0.919721910730004 84. 0.982824077364057
85. 0.0378025793470442 86. 0.577937400899827 87. 0.73331416817382 88. 0.248742402764037
89. 0.300736524863169 90. 0.733466701582074 91. 0.906954375561327 92. 0.209816768066958
93. 0.358137989183888 94. 0.448299144394696 95. 0.906426433008164 96. 0.389439295744523
97. 0.517459749476984 98. 0.125239087734371 99. 0.0301457457244396 100. 0.771805494558066

```
In [27]: plot(x = weight, y = obese)
```



Now, we will use the **glm()** function to fit a **logistic regression** curve to the data.

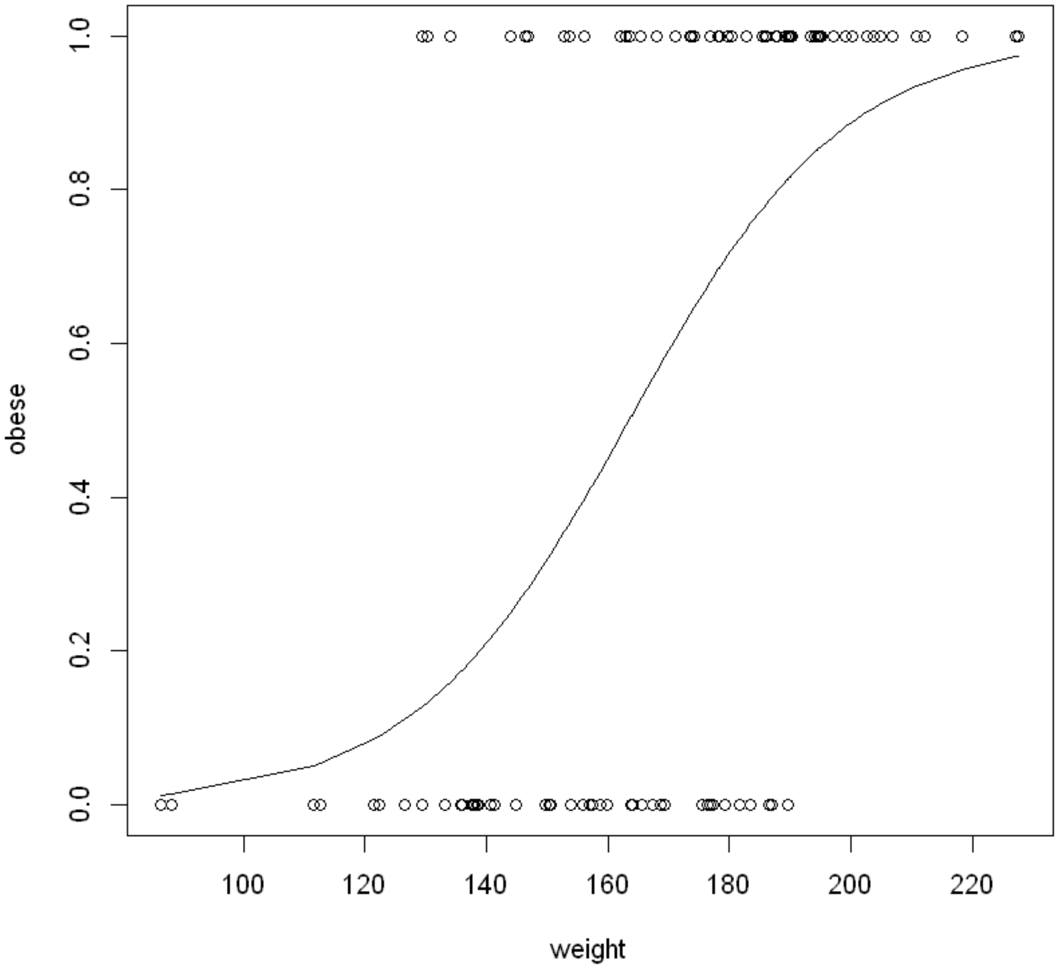
```
In [28]: glm.fit<-glm(obese~weight, family = binomial)

In [31]: summary(glm.fit)


Call:
glm(formula = obese ~ weight, family = binomial)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.8342  -0.7994   0.3874   0.7736   2.0260
```

```
Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept) -9.21184    1.98332  -4.645 3.41e-06 ***
weight       0.05636    0.01177   4.787 1.69e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 137.63  on 99  degrees of freedom
Residual deviance: 101.32  on 98  degrees of freedom
AIC: 105.32

Number of Fisher Scoring iterations: 4
```

In [33]:
```
head(glm.fit$fitted.values)
```

1 0.0129044243892409 **2** 0.0140757942451905 **3** 0.0508243708820616 **4** 0.0541785288702571 **5** 0.0861659649235592 **6** 0.0898091423625073

In [30]:
```
plot(x = weight, y = obese)
lines(weight, glm.fit$fitted.values)
```



The above curve tells us the **predicted probability** that individual is **obese** or **not obese**.

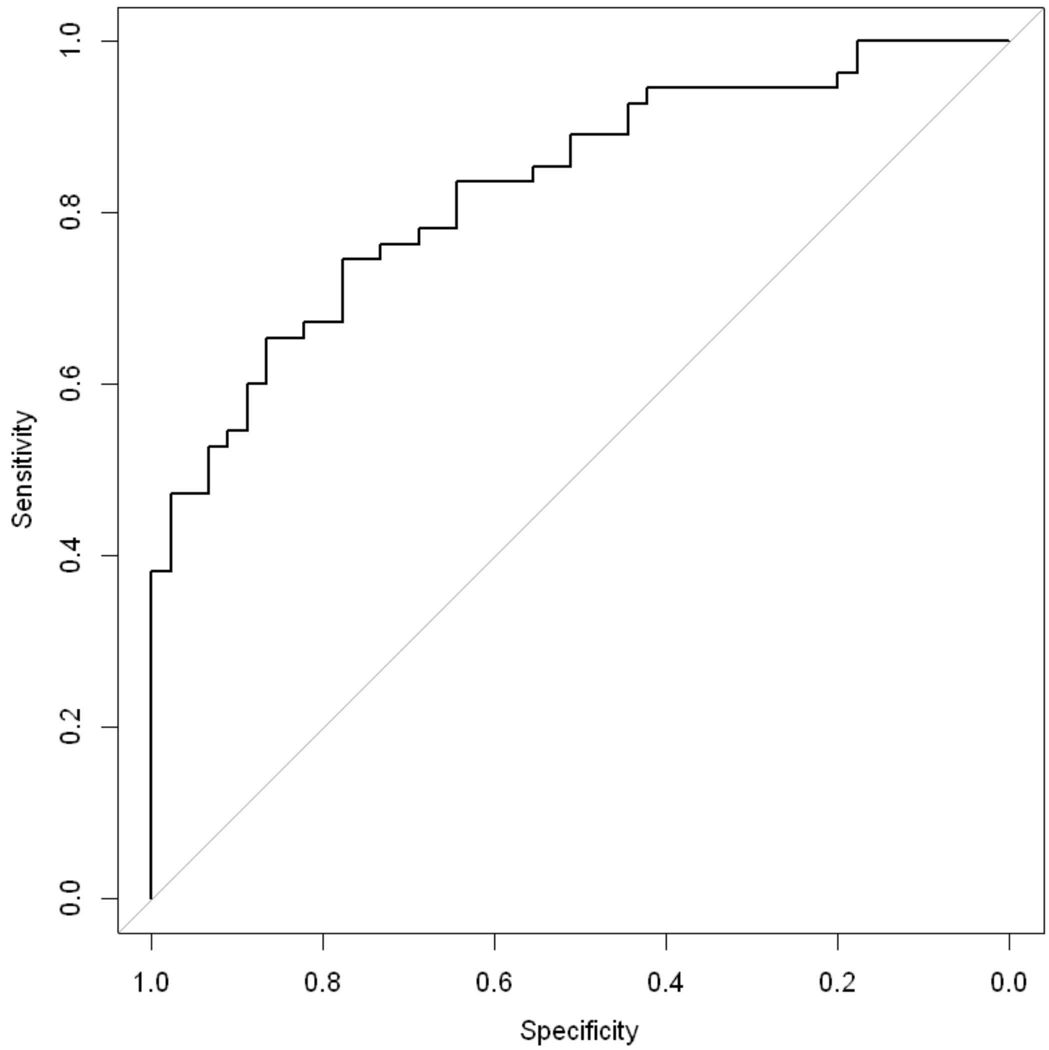
glm.fit\$$fitted.values * *containsy - axiscoordinatesalongthecurveforeachsample.Inotherwords, **$
$glm.fit$
fitted.values contains estimated **probabilities** that each sample is obese.

We will now use the **known classifications** and the **estimated probabilities** to draw an **ROC curve**.

```
In [40]: # par(pty='s')  Used in Rstudio to remove the extra paddings on the side (plot type s
         roc(obese, glm.fit$fitted.values, plot= TRUE)
```

```
Setting levels: control = 0, case = 1
Setting direction: controls < cases
```

```
Call:
roc.default(response = obese, predictor = glm.fit$fitted.values,     plot = TRUE)

Data: glm.fit$fitted.values in 45 controls (obese 0) < 55 cases (obese 1).
Area under the curve: 0.8291
```



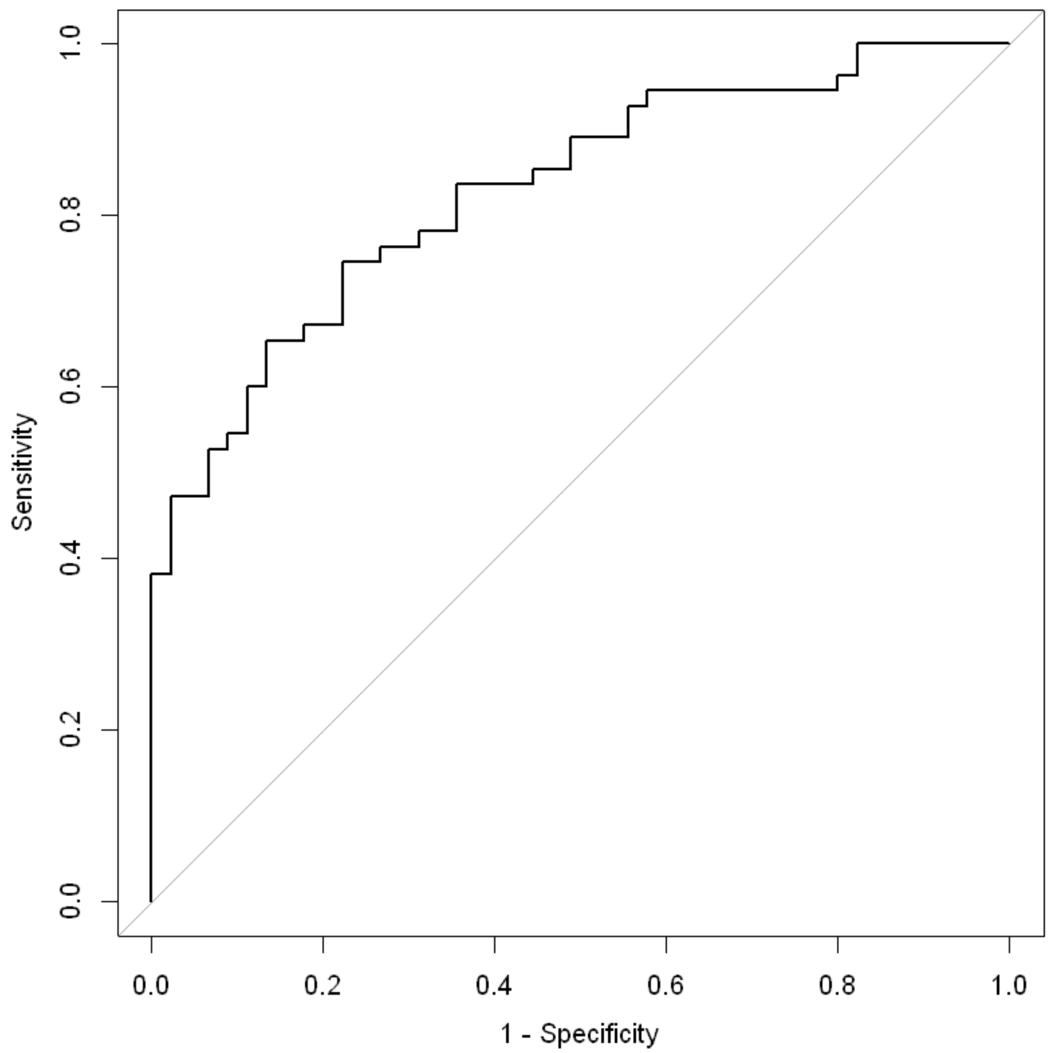
By default, the ROC function plots Specificity on the X-axis instead of 1-Specificity. As a result, X-axis goes from 1 on the left side to 0 on the right side.

The below code shows the 1-Specificity on the X-axis.

```
In [41]: roc(obese, glm.fit$fitted.values, plot= TRUE, legacy.axes = TRUE)
```

```
Setting levels: control = 0, case = 1
Setting direction: controls < cases
```

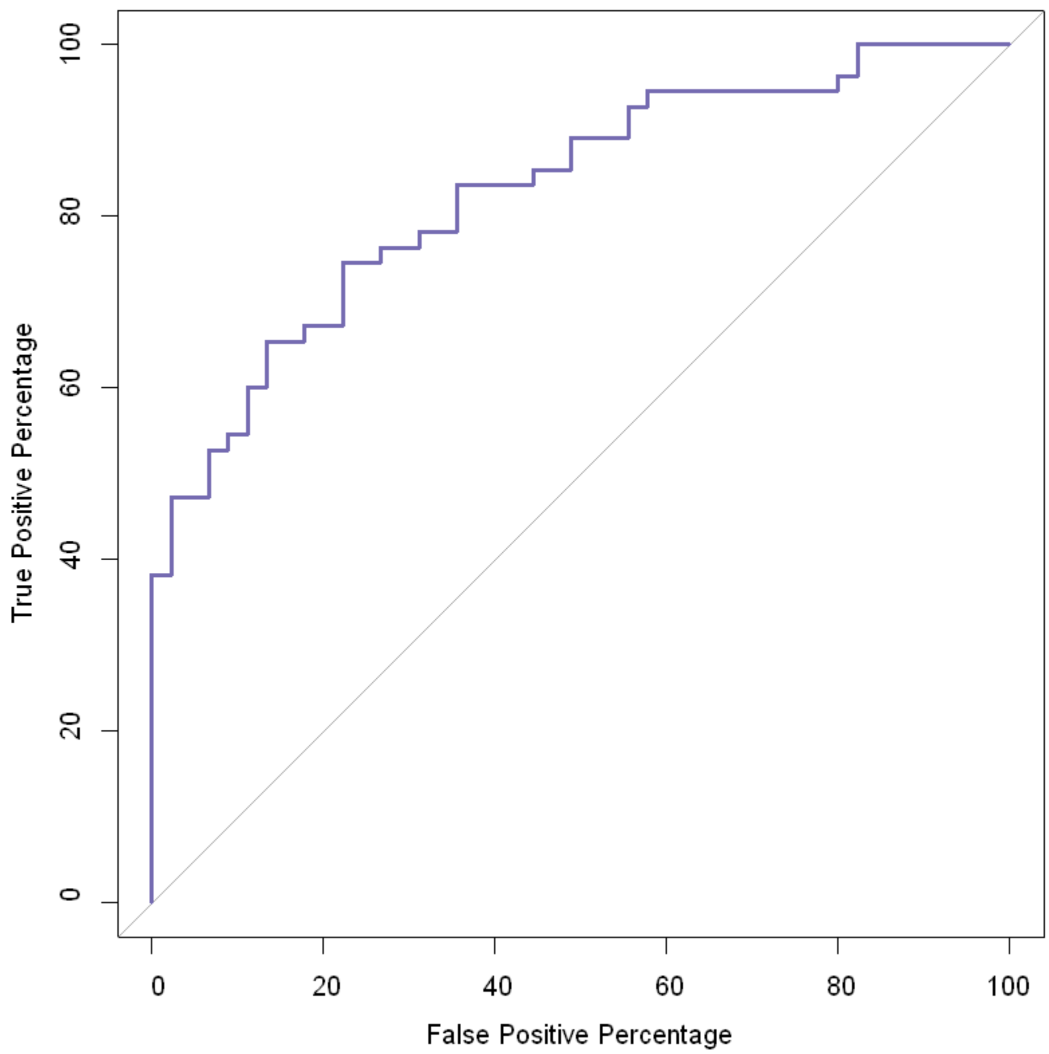
```
Call:
roc.default(response = obese, predictor = glm.fit$fitted.values,     plot = TRUE, legacy.axes =

Data: glm.fit$fitted.values in 45 controls (obese 0) < 55 cases (obese 1).
Area under the curve: 0.8291
```




```
In [53]: # Look at colorbrewer website for colors
         roc(obese, glm.fit$fitted.values, plot= TRUE, legacy.axes = TRUE, percent = TRUE, xlab
             ylab = 'True Positive Percentage', col = '#756bb1', lwd = 3)
```

```
Setting levels: control = 0, case = 1
Setting direction: controls < cases
```

```
Call:
roc.default(response = obese, predictor = glm.fit$fitted.values,     percent = TRUE, plot = TRU

Data: glm.fit$fitted.values in 45 controls (obese 0) < 55 cases (obese 1).
Area under the curve: 82.91%
```



Suppose we are now interested in the **range of thresholds** that resulted in some part of the above **ROC curve**.

We can access those **thresholds** by saving the calculations that **roc() function** did in a variable and then create a dataframe that contains all of the **True Positive Percentages** by multiplying the **Sensitivities** by **100** and **False Positive Percentages** by multiplying **1 - Specificities** by **100** and also get the **threshold information**.

```
In [55]: roc.info<-roc(obese, glm.fit$fitted.values, legacy.axes = TRUE)
```

```
Setting levels: control = 0, case = 1
Setting direction: controls < cases
```

```
In [56]: roc.df<- data.frame(tpp = roc.info$sensitivities*100, fpp = (1 - roc.info$specificiti
                             roc.info$thresholds)
```

```
In [57]: head(roc.df)
```

tpp	fpp	thresholds
100	100.00000	-Inf
100	97.77778	0.01349011
100	95.55556	0.03245008
100	93.33333	0.05250145
100	91.11111	0.07017225
100	88.88889	0.08798755

First row of the above dataframe corresponds to the upper right corner of the ROC curve.

```
In [58]: tail(roc.df)
```

	tpp	fpp	thresholds
96	9.090909	0	0.9275222
97	7.272727	0	0.9371857
98	5.454545	0	0.9480358
99	3.636364	0	0.9648800
100	1.818182	0	0.9735257
101	0.000000	0	Inf

Last row of the above dataframe corresponds to the bottom left-hand corner of the ROC curve.

Now, we can islolate the **TPP**, **FPP** and **thresholds** when the True positive rate is between 60 and 80.

```
In [60]: roc.df[roc.df$tpp>60 & roc.df$tpp<80,]
```

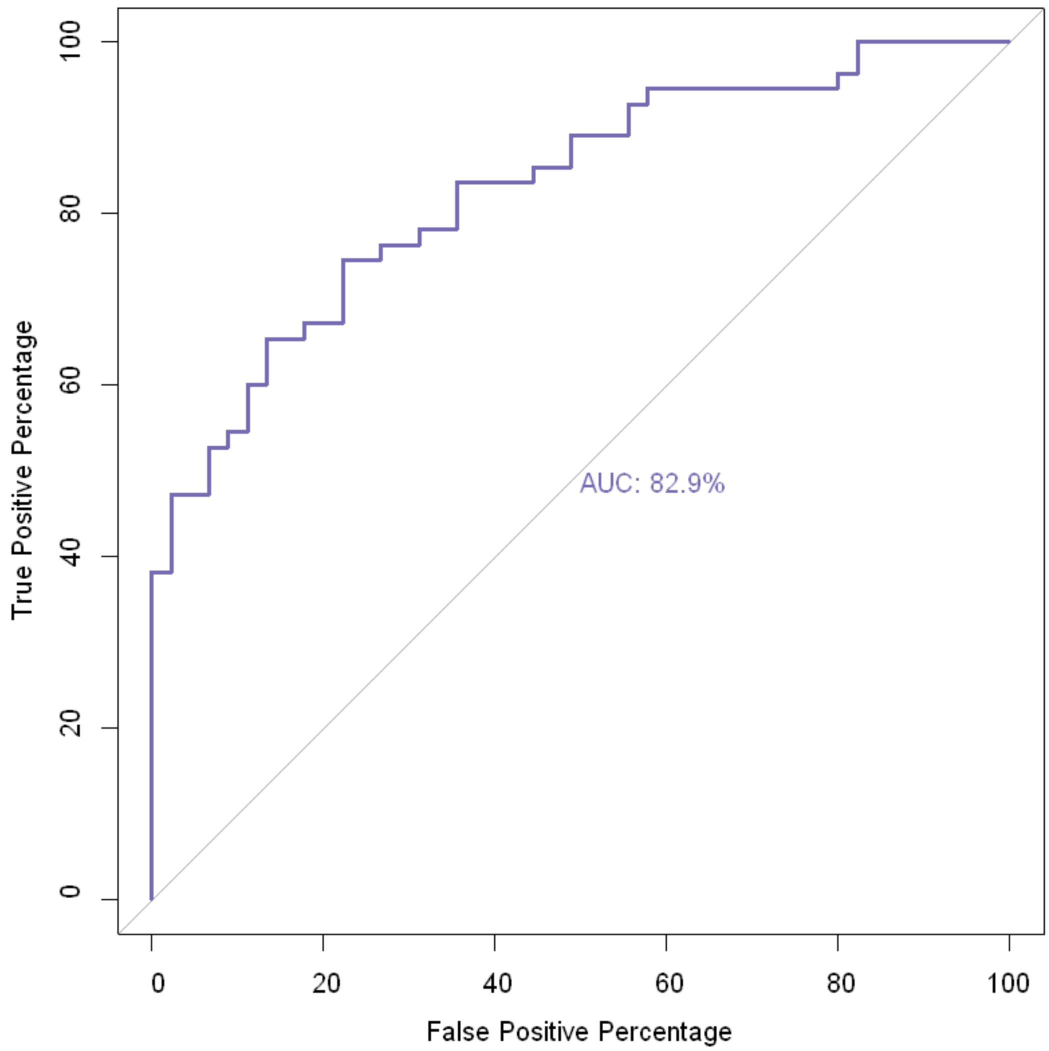
	tpp	fpp	thresholds
42	78.18182	35.55556	0.5049310
43	78.18182	33.33333	0.5067116
44	78.18182	31.11111	0.5166680
45	76.36364	31.11111	0.5287933
46	76.36364	28.88889	0.5429351
47	76.36364	26.66667	0.5589494
48	74.54545	26.66667	0.5676342
49	74.54545	24.44444	0.5776086
50	74.54545	22.22222	0.5946054
51	72.72727	22.22222	0.6227449
52	70.90909	22.22222	0.6398136
53	69.09091	22.22222	0.6441654
54	67.27273	22.22222	0.6556705
55	67.27273	20.00000	0.6683618
56	67.27273	17.77778	0.6767661
57	65.45455	17.77778	0.6802060
58	65.45455	15.55556	0.6831936
59	65.45455	13.33333	0.6917225
60	63.63636	13.33333	0.6975300
61	61.81818	13.33333	0.6982807

If we are interested in picking up thresholds in this range, we can do so by picking the one that has an optimal balance of **True Positives** and **False Positives**.

Now, let's show **AUC** on the ROC graph.

```
In [61]: # print.auc= TRUE
         roc(obese, glm.fit$fitted.values, plot= TRUE, legacy.axes = TRUE, percent = TRUE, xlal
             ylab = 'True Positive Percentage', col = '#756bb1', lwd = 3, print.auc= TRUE)
```

```
Setting levels: control = 0, case = 1
Setting direction: controls < cases



Call:
roc.default(response = obese, predictor = glm.fit$fitted.values,     percent = TRUE, plot = TRU

Data: glm.fit$fitted.values in 45 controls (obese 0) < 55 cases (obese 1).
Area under the curve: 82.91%
```



We can also draw and calculate a **partial Area** under the curve. These are useful when you want to focus on the part of the **ROC** curve that only allows for a small number of **False Positives**.

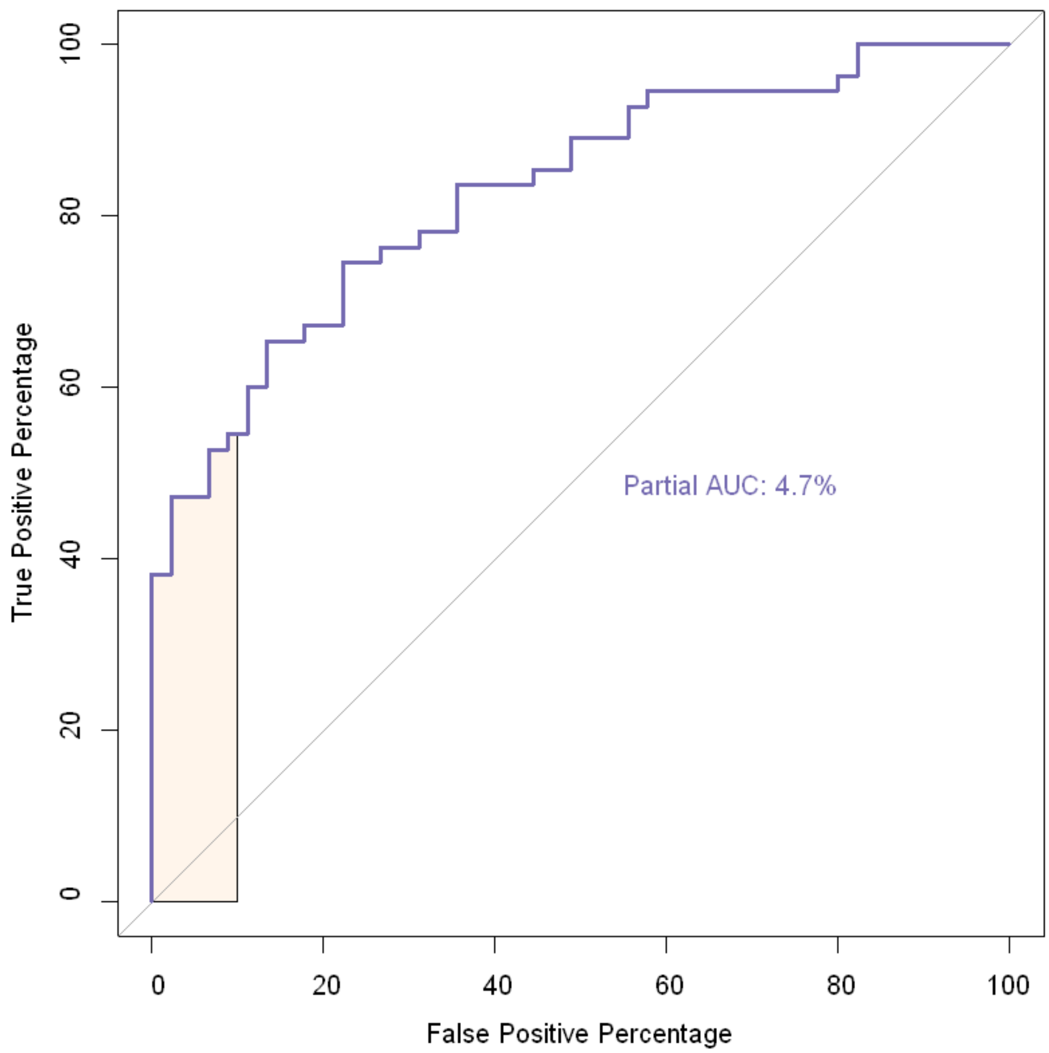
After specifying print.auc = TRUE, we have to specify where along the x-axis we want the AUC to be printed otherwise the text might overlap something important.

Then we set the partial.auc to a range of **specificity values** that we want to focus on. Here, partial.auc = c(100,90). **Note:** 100% specificity corresponds to 0% on our (1-specificity) axis.

Then we draw the partial area under the curve by setting auc.polygon = TRUE. Optionally set auc.polygon.col to specify polygon's color. **Note:** Add two digits(22) to the end of RGB numbers to make the color semi-transparent.

```
In [67]: roc(obese, glm.fit$fitted.values, plot= TRUE, legacy.axes = TRUE, percent = TRUE, xlal
             ylab = 'True Positive Percentage', col = '#756bb1', lwd = 3, print.auc= TRUE, prin
            auc.polygon = TRUE, auc.polygon.col = '#fee6ce66')
```

```
Setting levels: control = 0, case = 1
Setting direction: controls < cases



Call:
roc.default(response = obese, predictor = glm.fit$fitted.values,     percent = TRUE, plot = TRU

Data: glm.fit$fitted.values in 45 controls (obese 0) < 55 cases (obese 1).
Partial area under the curve (specificity 100%-90%): 4.727%
```

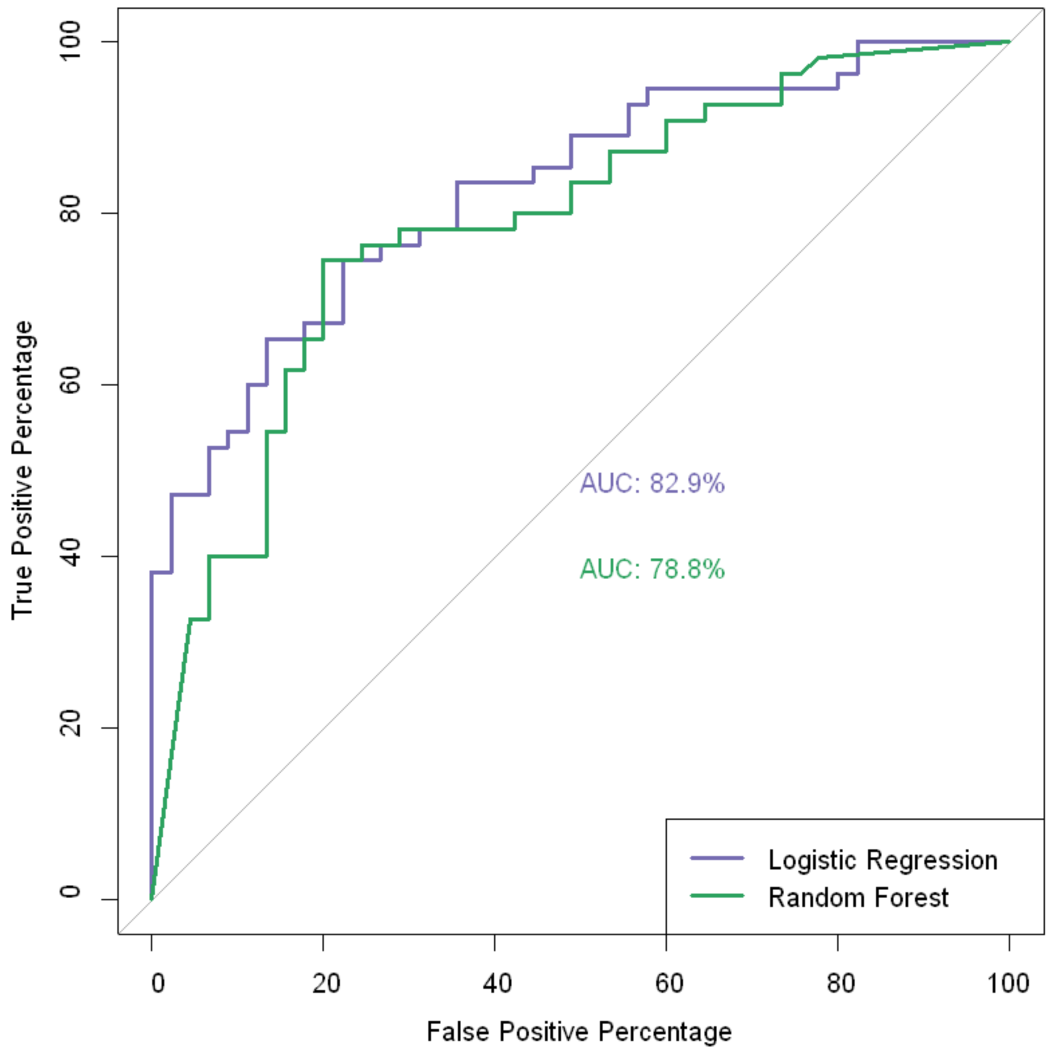


Now, let us try to **overlap two ROC curves** so they are **easy to compare**.

```
In [68]: # Random Forest Classifier
         rf.model<-randomForest(factor(obese)~weight)
```

```
In [74]: roc(obese, glm.fit$fitted.values, plot= TRUE, legacy.axes = TRUE, percent = TRUE, xlab
             ylab = 'True Positive Percentage', col = '#756bb1', lwd = 3, print.auc= TRUE)
         plot.roc(obese, rf.model$votes[,1], percent = TRUE, col = '#2ca25f', lwd = 3, print.au
         legend('bottomright',legend = c('Logistic Regression', 'Random Forest'), col = c('#756
```

```
Setting levels: control = 0, case = 1
Setting direction: controls < cases



Call:
roc.default(response = obese, predictor = glm.fit$fitted.values,     percent = TRUE, plot = TRU

Data: glm.fit$fitted.values in 45 controls (obese 0) < 55 cases (obese 1).
Area under the curve: 82.91%


Setting levels: control = 0, case = 1
Setting direction: controls > cases
```



We pass in the number of trees in the forest that voted correctly. (rf.model$votes[,1])

```
In [79]: head(rf.model$votes[,])
```

0	1
1.0000000	0.00000000
1.0000000	0.00000000
0.9887006	0.01129944
0.9896907	0.01030928
0.9768786	0.02312139
0.9771429	0.02285714

```
In [80]: tail(rf.model$votes[,])
```

	0	1
95	0	1
96	0	1
97	0	1
98	0	1
99	0	1
100	0	1