Final project

Candidate Numbers: 20317, 12746, 16436, 15464

07/05/2021

Introduction

A stroke is a serious life-threatening medical condition that happens when the blood supply to parts of the brain is cut off. Various symptoms such as the face dropping on one side, a person not being able to lift both arms and slurred speech can indicate that someone has experienced a stroke. Worldwide, cerebrovascular accidents (stroke) is the second leading cause of death. There are various factors which could impact the occurrence of such stroke such as a consumers diet, level of exercise and alchohol consumption. Text

The source of that data set can be obtained via https://www.kaggle.com/fedesoriano/stroke-prediction-dataset (https://www.kaggle.com/fedesoriano/stroke-prediction-dataset). The data set includes 5110 observations and 12 attributes. 11 predictor variables (UNIQUE I.D, GENDER, AGE, HYPERTENSION, HEART DISEASE, EVER MARRIED, WORK TYPE, RESIDENCE TYPE, AVERAGE GLUCOSE LEVEL, BMI, SMOKING STATUS) and the outcome variable STROKE is provided. This investigation aims to apply machine learning methods to the data set and predict the likelihood of a stroke given a person's characteristics. Subsequently, a range of models have been constructed. Examples include:

• linear models • logistic models • model implementing gradient descent • model with trees and a random forest • model using neutral network

The feasibility and implications of these models are discussed before a discussion of possibilities for further work concludes the report.

Data Preprocessing

First we load the data set. Ignoring the dummy variable 'id', we have in total 5110 observations with 10 variables along with the binary outcome variable 'stroke'. The fact that we have much more observations than predictors allow the possibility for further high-dimensional modeling.

```
stroke_data <- read.csv('healthcare_dataset_stroke_data.csv')
dim(stroke_data)
```

```
## [1] 5110   12
```

We have one sample with the response 'other' for the variable 'gender'. While keeping it will make turning it to a binary factor impossible hence complicates data processing, removing one sample out of 5110 observations will only cause minimal impact. Therefore, the observation is dropped.

```
stroke_data %>% count(gender)
```

```
##   gender    n
## 1 Female 2994
## 2   Male 2115
## 3  Other    1
```

```
stroke_data <- stroke_data[!(stroke_data$gender=='Other'),];
stroke_data %>% count(gender);
```

```
##   gender    n
## 1 Female 2994
## 2   Male 2115
```

```
stroke_data1 <- stroke_data;
```

We make sure that all the categorical variables are of type factor and we remove the dummy variable ‘id’ which is not useful for modeling.

```
stroke_data1$gender <- factor(stroke_data1$gender);
stroke_data1$hypertension <- factor(stroke_data1$hypertension);
stroke_data1$heart_disease <- factor(stroke_data1$heart_disease);
stroke_data1$ever_married <- factor(stroke_data1$ever_married);
stroke_data1$work_type <- factor(stroke_data1$work_type);
stroke_data1$Residence_type <- factor(stroke_data1$Residence_type);
stroke_data1$smoking_status <- factor(stroke_data1$smoking_status);
stroke_data1$stroke <- factor(stroke_data1$stroke);
stroke_data1$id <- NULL;
glimpse(stroke_data1)
```

```
## Rows: 5,109
## Columns: 11
## $ gender            <fct> Male, Female, Male, Female, Female, Male, Male, Fema~
## $ age               <dbl> 67, 61, 80, 49, 79, 81, 74, 69, 59, 78, 81, 61, 54, ~
## $ hypertension      <fct> 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1~
## $ heart_disease     <fct> 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0~
## $ ever_married      <fct> Yes, Yes, Yes, Yes, Yes, Yes, Yes, No, Yes, Yes, Yes~
## $ work_type         <fct> Private, Self-employed, Private, Private, Self-emplo~
## $ Residence_type    <fct> Urban, Rural, Rural, Urban, Rural, Urban, Rural, Urb~
## $ avg_glucose_level <dbl> 228.69, 202.21, 105.92, 171.23, 174.12, 186.21, 70.0~
## $ bmi               <chr> "36.6", "N/A", "32.5", "34.4", "24", "29", "27.4", "~
## $ smoking_status    <fct> formerly smoked, never smoked, never smoked, smokes,~
## $ stroke            <fct> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
```

Similarly, missing values in the variable ‘bmi’ are found, but number of observations where BMI is unknown is a relatively small proportion of the total number of observations and imputing such variable might introduce more bias, so we decide to remove these observations.

```
stroke_data1 %>% count(bmi=='N/A') # show numbers of missing values of bmi
```

```
##   bmi == "N/A"    n
## 1        FALSE 4908
## 2         TRUE  201
```

```
stroke_data2<- stroke_data1[!(stroke_data1$bmi=='N/A'),]# removes observations with N/A as the valu
e for BMI
stroke_data2$bmi = as.numeric(as.character(stroke_data2$bmi));
```

We also observed missing values in the variable ‘smoking_status’. However, given the fact the this is a categorical variable and classification models have the capability to treat ‘Unknown’ as a seperate response with no issue, these responses are therefore kept as ‘Unknown’.

```
stroke_data %>% count(smoking_status)
```

```
##    smoking_status    n
## 1 formerly smoked  884
## 2    never smoked 1892
## 3          smokes  789
## 4         Unknown 1544
```

We now split the dataset into training and test data.

```
set.seed(1)
stroke_data_split <- initial_split(stroke_data2, strata = stroke) # 'strata=stroke' makes sure that
the proportion of stroke victims is the same in both the test data and the training data

stroke_data_train <- training(stroke_data_split)
stroke_data_test <- testing(stroke_data_split)
# *****explain test and training data difference******

stroke_data_cv <- vfold_cv(stroke_data_train, v = 10, strata = stroke) # Forming 10 fold cross vali
dation subsets of training data.

# get the recipe
stroke_recipe <- training(stroke_data_split) %>%
  recipe(stroke ~ .) %>%
  prep()
```

Using linear regression to model stroke data

We make sure that the outcome variable is of type numeric.

```
stroke_data_train$stroke <- as.numeric(stroke_data_train$stroke) - 1;
stroke_data_test$stroke <- as.numeric(stroke_data_test$stroke) - 1;
```

We are predicting the likelihood,theta, of a person getting a stoke based on certain characteristics:

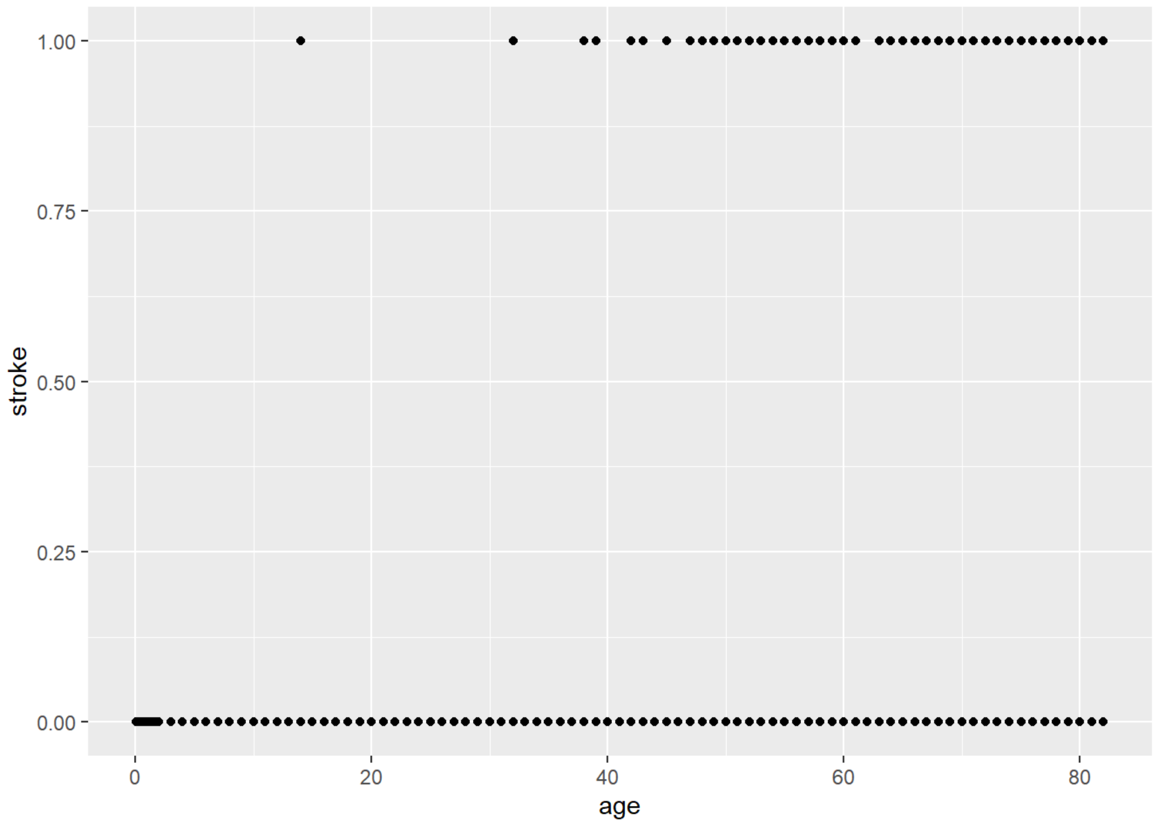
$$\theta_i = Y = \beta_0 + \beta_1 X_1$$

```
## We first try out a simple linear model to predict the likelyhood of a person with certain charac
teristics having a stroke.
linear_model <- lm(stroke~age, data = stroke_data_train)
linear_model
```

```
##
## Call:
## lm(formula = stroke ~ age, data = stroke_data_train)
##
## Coefficients:
## (Intercept)          age
##   -0.046436     0.002073
```

```
## computes the simple linear model using stroke and age on the training dataset
summary(linear_model)
```

```
##
## Call:
## lm(formula = stroke ~ age, data = stroke_data_train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.12354 -0.07379 -0.04062  0.00291  1.01742
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.046436   0.006947  -6.684 2.67e-11 ***
## age          0.002073   0.000143  14.497  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1966 on 3679 degrees of freedom
## Multiple R-squared:  0.05404,    Adjusted R-squared:  0.05378
## F-statistic: 210.2 on 1 and 3679 DF,  p-value: < 2.2e-16
## computes a summary of the model, includes multiple r squared and significance levels.
## The R-squared of the linear model, which can be thought of as “the percentage of variability in
 the response that is explained by the predictor” - r^2=0.05 which is very low and hence not a stro
ng association between the response and predictor variable.
## Relationship is negative, b0 and b1 have different coeffecient signs indicating as one increase
s, the other decreases.
ggplot(stroke_data_train, aes(x=age, y=stroke)) +
  geom_point()
```



```
## Plots the simple linear regression. From the visualisation we can see that simple linear regress
ion is not a good model.
```

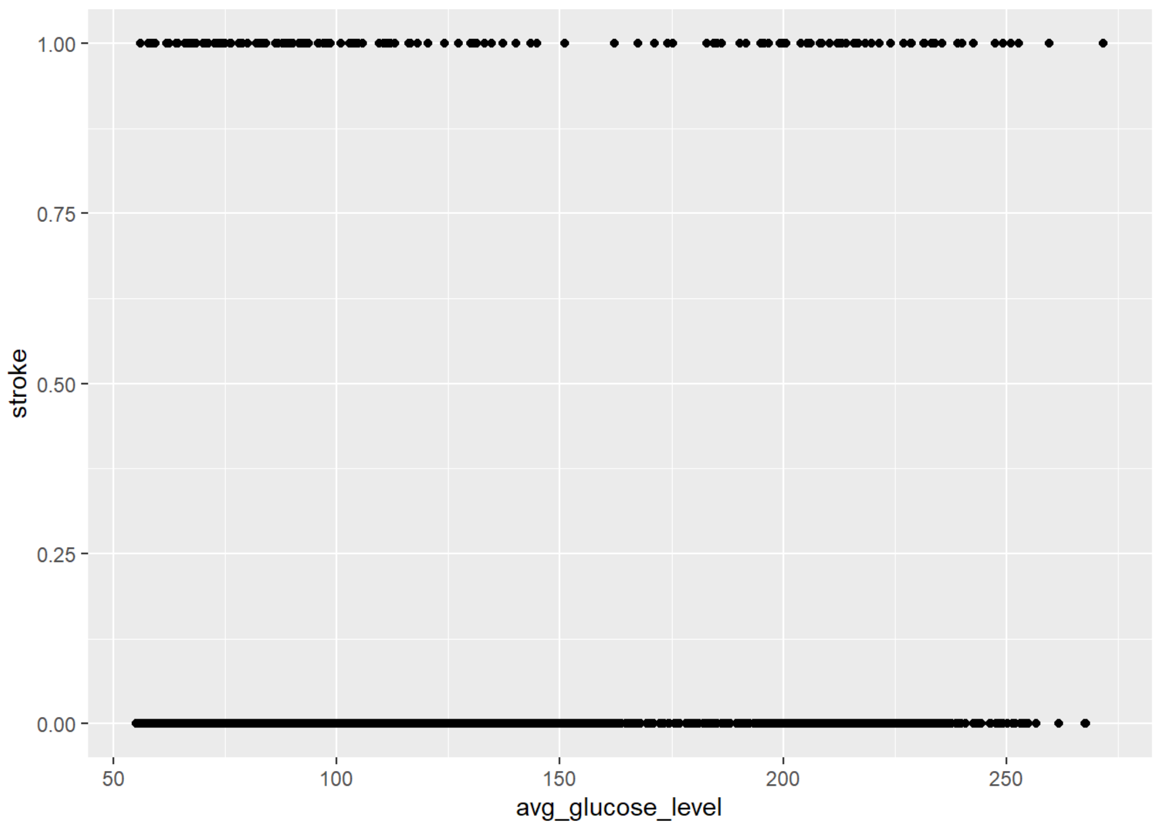
```
linear_model1 <- lm(stroke~avg_glucose_level, data = stroke_data_train)
linear_model1
```

```
##
## Call:
## lm(formula = stroke ~ avg_glucose_level, data = stroke_data_train)
##
## Coefficients:
##       (Intercept)  avg_glucose_level
##        -0.0208321          0.0005988
```

```
## computes the simple linear model using stroke and age on the trainig dataset
summary(linear_model1)
```

```
##
## Call:
## lm(formula = stroke ~ avg_glucose_level, data = stroke_data_train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.13950 -0.04635 -0.03306 -0.02419  0.98723
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -2.083e-02  8.474e-03  -2.458    0.014 *
## avg_glucose_level  5.988e-04  7.361e-05   8.135 5.58e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2003 on 3679 degrees of freedom
## Multiple R-squared:  0.01767,    Adjusted R-squared:  0.0174
## F-statistic: 66.18 on 1 and 3679 DF,  p-value: 5.579e-16
```

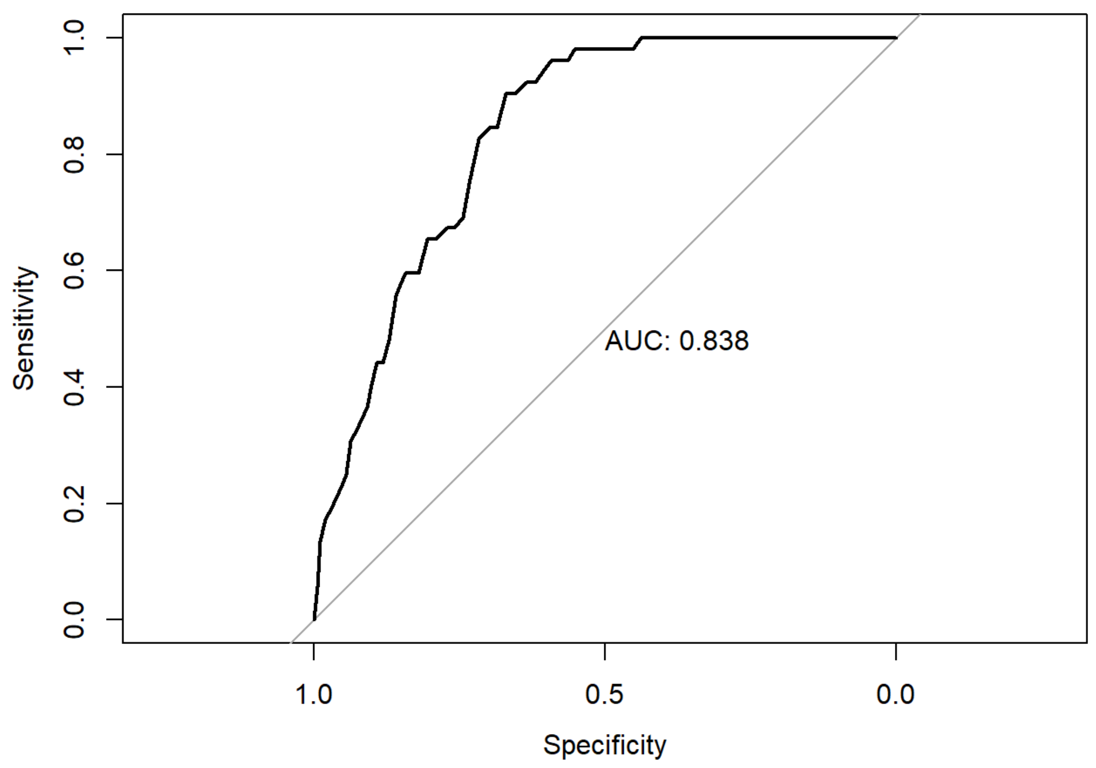
```
## computes a summary of the model, includes multiple r squared and significance levels.
## The R-squared of the linear model is very low.
## Relationship is negative, b0 and b1 have different coeffecient signs indicating as one increase
s, the other decreases.
ggplot(stroke_data_train, aes(x=avg_glucose_level, y=stroke)) +
  geom_point()
```



```
## Plots the simple lineare regression.From the visualisation we can see that simple linear regress
ion is not  a good model.
```

```
## Linear regression on age and stroke
test_prob = predict(linear_model, newdata = stroke_data_test)
## testing the area under the curve to obtain the aaccuracy of the model
test_roc = pROC::roc(stroke_data_test$stroke ~ test_prob, plot = TRUE, print.auc = TRUE)
```

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

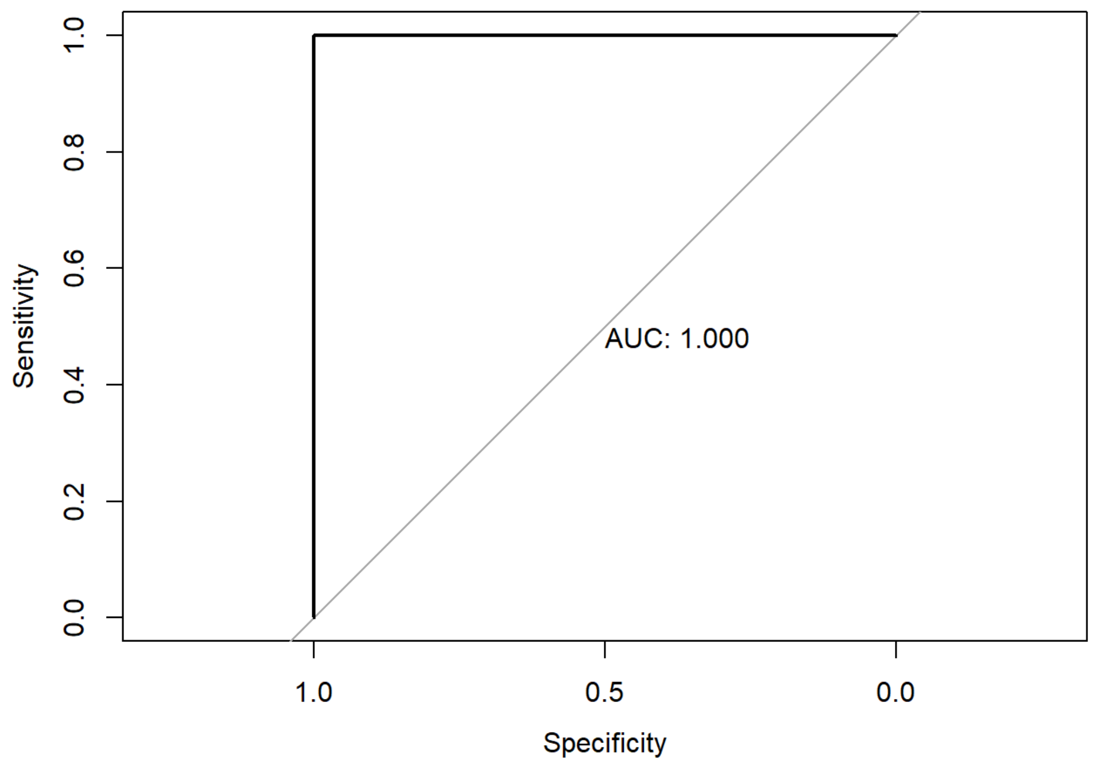


```
## Linear regression on average level of glucose and stroke
test_prob = predict(linear_model1, newdata = stroke_data_test)
## testing the area under the curve to obtain the aaccuracy of the model
test_roc = pROC::roc(stroke_data_test$avg_glucose_level ~ test_prob, plot = TRUE, print.auc = TRUE)
```

```
## Warning in roc.default(response, predictors[, 1], ...): 'response' has more
## than two levels. Consider setting 'levels' explicitly or using 'multiclass.roc'
## instead
```

```
## Setting levels: control = 55.22, case = 55.27
```

```
## Setting direction: controls < cases
```



Using logistic regression to model stroke data

The logistic regression model is a better model than the linear regression model when it comes to classification, where the output is binary. Linear regression is not a good model because it could predict negative values for y, stroke, which is not possible in real life. We are predicting the probability, theta, that a people gets stroke, given some characteristic x:

$$\theta_i = p(x) = P(Y = 1 \mid X = x)$$

The logistic model is as follows:

$$log \frac{p(x)}{(1 - p(x))} = X\beta$$

Which gives:

$$\theta = \frac{1}{(1 + e^{-\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p})}$$

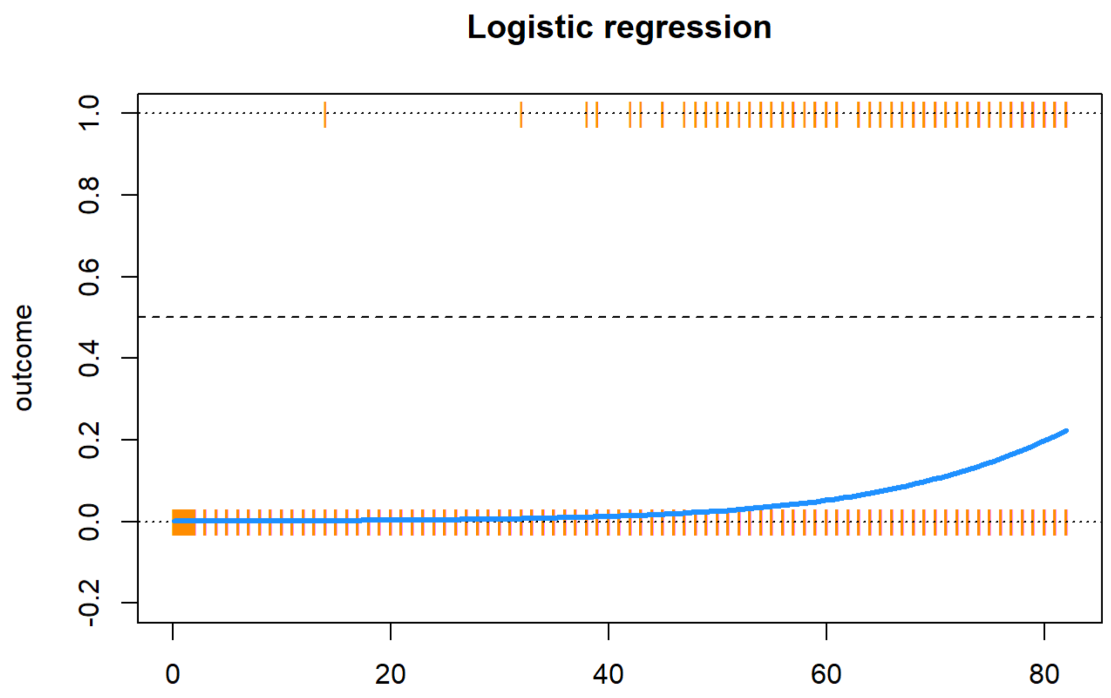
Logistic regression function

```
#Writing a function that allows you to type in the predictor and outcome and fits a logistic model
 based on the variables selected
logistic_regression_model <-function(predictor, outcome)    {
  #function allows user to input the predictor and outcome variables
  outcome <- as.numeric(outcome)
  predictor <-as.numeric(predictor)
  logit <- glm(outcome ~ predictor , data = stroke_data_train, family = "binomial")
  # predicts the logit model using the glm function, where the family is set to binomial
  print(summary(logit)) #Prints a summary of the model, including the estimates for theta and the s
ignificance levels
plot(outcome ~ predictor, data = stroke_data_train, xlab= predictor,
     col = "darkorange", pch = "|", ylim = c(-0.2, 1),
     main= "Logistic regression"  )
#plots the logit regression so we can interpret it
abline(h = 0, lty = 3)
abline(h = 1, lty = 3)
abline(h = 0.5, lty = 2)
curve(predict(logit, data.frame(predictor = x), type = "response"),
     add = TRUE, lwd = 3, col = "dodgerblue")
abline(v = -coef(logit)[1] / coef(logit)[2], lwd = 2)
#a sigmoid curve is fit to the logit model
}
```

Logit on age and stroke

```
logistic_regression_model(stroke_data_train$age,stroke_data_train$stroke)
```

```
##
## Call:
## glm(formula = outcome ~ predictor, family = "binomial", data = stroke_data_train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.7100  -0.3056  -0.1696  -0.0777   3.5570
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -7.368999   0.417517  -17.65   <2e-16 ***
## predictor    0.074629   0.006106   12.22   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1297.8  on 3680  degrees of freedom
## Residual deviance: 1056.9  on 3679  degrees of freedom
## AIC: 1060.9
##
## Number of Fisher Scoring iterations: 7
```

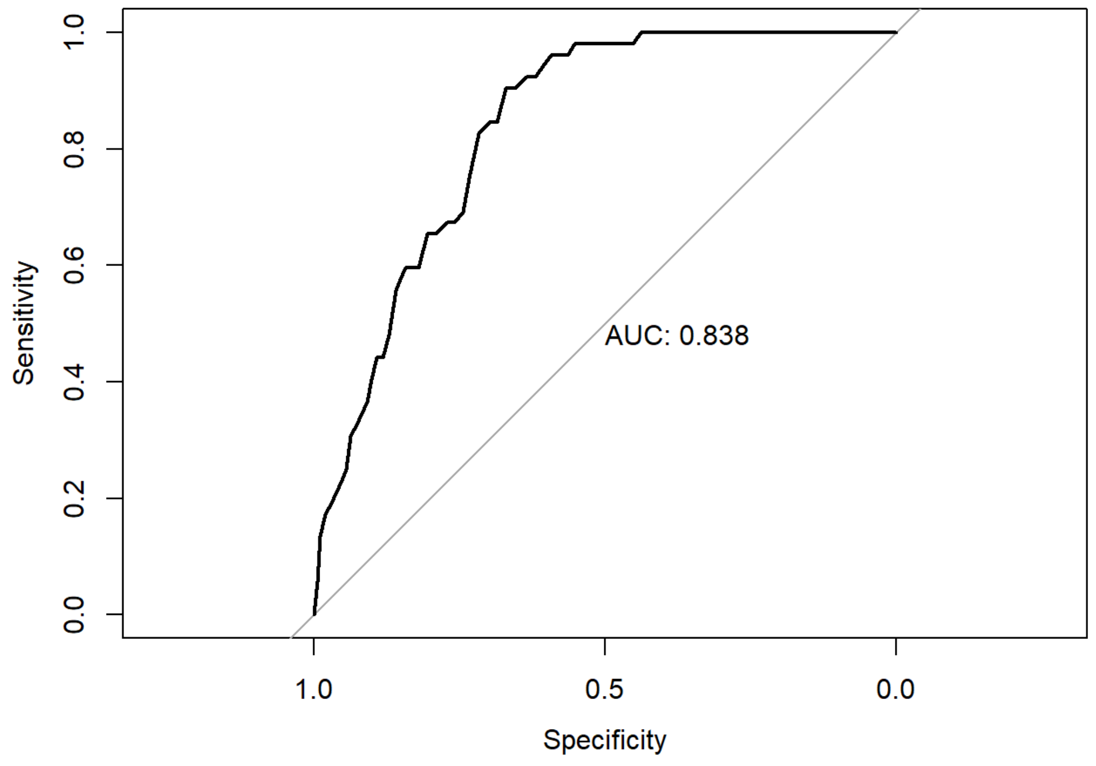


67
80

```
logit <- glm(stroke ~ age , data = stroke_data_train, family = "binomial")
test_prob = predict(logit, stroke_data_test, type = "response") #Here we are predicting the test da
ta using the model created using training data
test_roc = roc(stroke_data_test$stroke ~ test_prob, plot = TRUE, print.auc = TRUE)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```



```
# testing the area under the curve to obtain the accuracy of the model
```

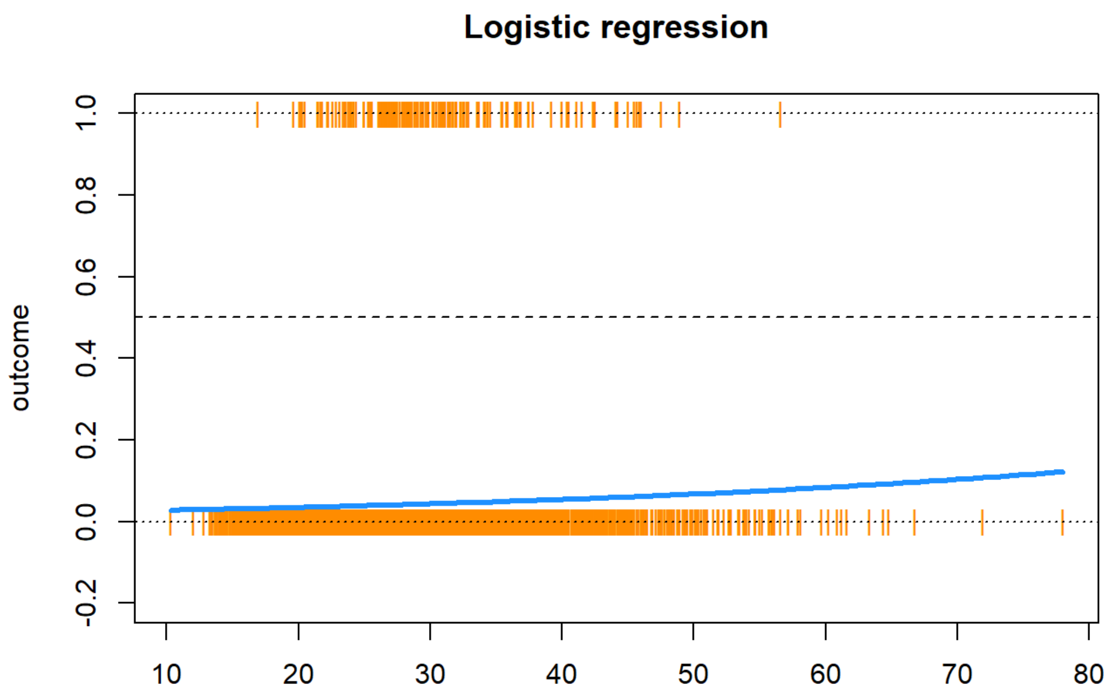
From the coefficient table, we see that age is a good predictor of probability of stroke. Since the coefficient on age is positive, an increase in age increases the probability of stroke. The t value shows that the coefficient is statistically significant under almost 100% significance level.

As can be seen from the ROC curve, this model has an AUC of 0.838. This means that the logit model using age to predict stroke has a good measure of separability. The model is able to distinguish between who is likely to suffer stroke and who is not, based on the age variable.

Logit on bmi and stroke

```
logistic_regression_model(stroke_data_train$bmi,stroke_data_train$stroke)
```

```
##
## Call:
## glm(formula = outcome ~ predictor, family = "binomial", data = stroke_data_train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.5093  -0.3063  -0.2886  -0.2728   2.6232
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -3.80389    0.30494 -12.474   <2e-16 ***
## predictor    0.02342    0.00970   2.414   0.0158 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1297.8  on 3680  degrees of freedom
## Residual deviance: 1292.2  on 3679  degrees of freedom
## AIC: 1296.2
##
## Number of Fisher Scoring iterations: 6
```



The output from

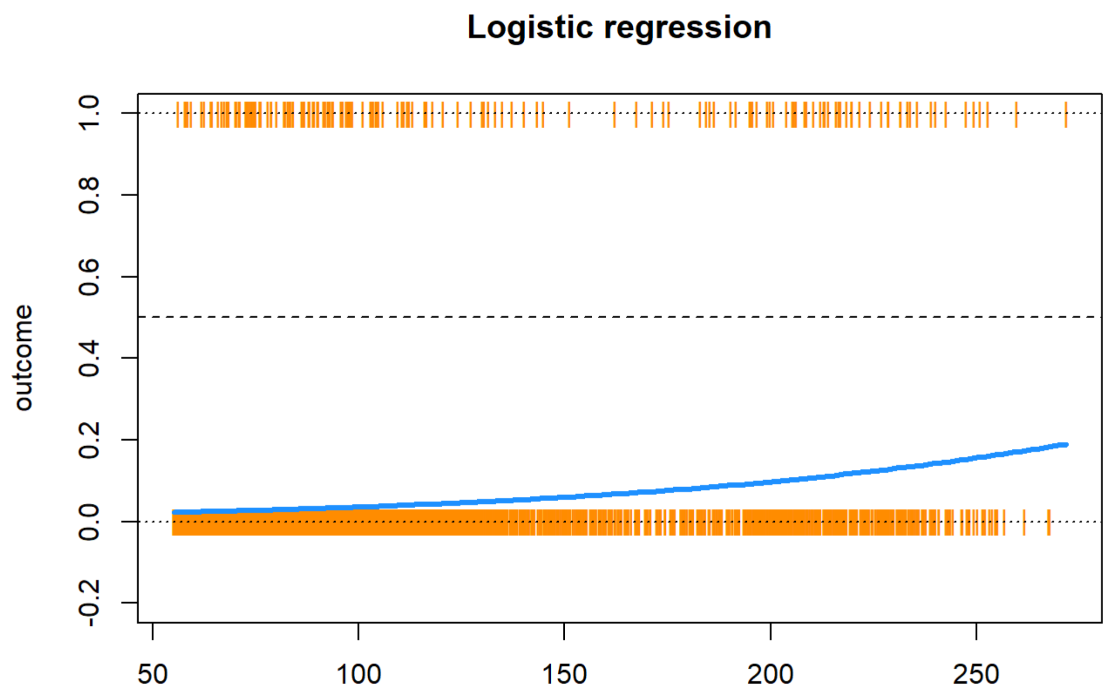
36.6
32.5

this shows that that BMI is a significant predictor of the probability of stroke. Since the coefficient on BMI is positive, an increase in BMI increases the probability of stroke. The t values show that the coefficient on BMI is statistically significant.

Logit on average glucose level and stroke

```
logistic_regression_model(stroke_data_train$avg_glucose_level,stroke_data_train$stroke)
```

```
##
## Call:
## glm(formula = outcome ~ predictor, family = "binomial", data = stroke_data_train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.6367  -0.2863  -0.2545  -0.2352   2.7576
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -4.385872   0.202631 -21.645  < 2e-16 ***
## predictor    0.010804   0.001405   7.689 1.48e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1297.8  on 3680  degrees of freedom
## Residual deviance: 1245.8  on 3679  degrees of freedom
## AIC: 1249.8
##
## Number of Fisher Scoring iterations: 6
```



The output from

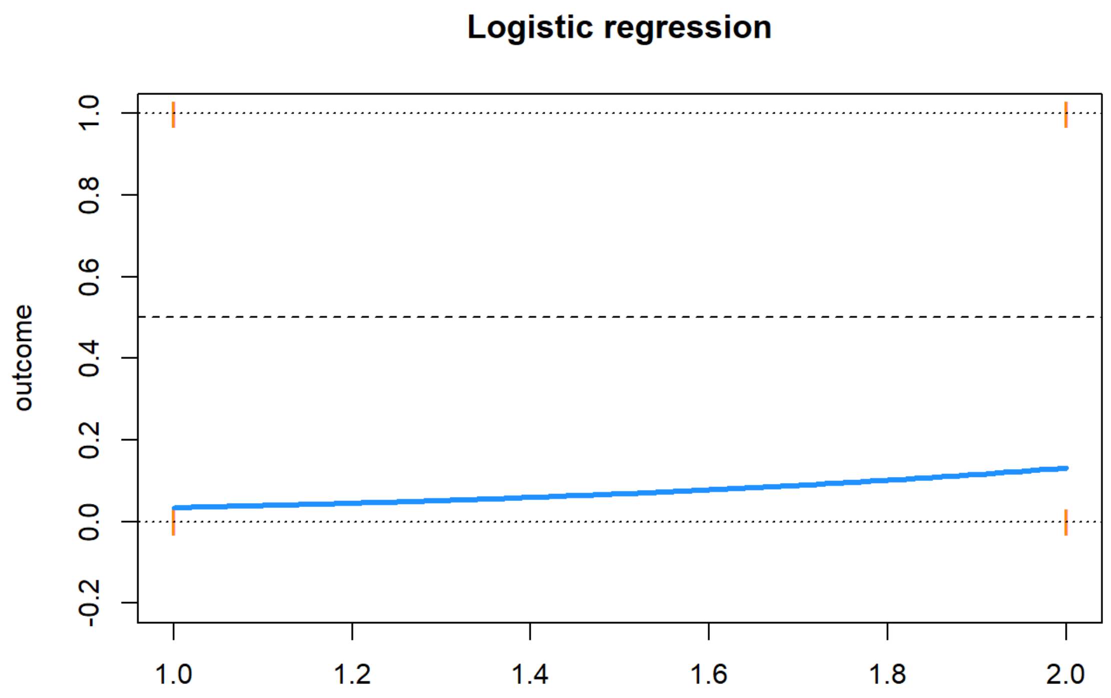
228.69
105.92

this shows that that average glucose level is a significant predictor of the probability of stroke.Since the coefficient on average glucose level is positive, an increase in average glucose level increases the probability of stroke. The t value for both the intercept estimate and the predictor estimate are significant.

Logit on hypertension and stroke

```
logistic_regression_model(stroke_data_train$hypertension,stroke_data_train$stroke)
```

```
##
## Call:
## glm(formula = outcome ~ predictor, family = "binomial", data = stroke_data_train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.5303  -0.2613  -0.2613  -0.2613   2.6056
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -4.8306     0.2501 -19.317  < 2e-16 ***
## predictor     1.4701     0.1866   7.879 3.31e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1297.8  on 3680  degrees of freedom
## Residual deviance: 1247.2  on 3679  degrees of freedom
## AIC: 1251.2
##
## Number of Fisher Scoring iterations: 6
```



The output from

1
1

this shows that that hypertension is a significant predictor of the probability of stroke. The intercept estimate tells us that someone who does not have hypertension is much less likely to get stroke. The probability of stroke increases if one has hypertension.

Multi-variate logit model

For ease of interpretation, I used bi-variate logit regression so we can see how each of the variables out of BMI, hypertension and age can be used to predict stroke independently. Now, we are going to look at a logit model fit on all four variables

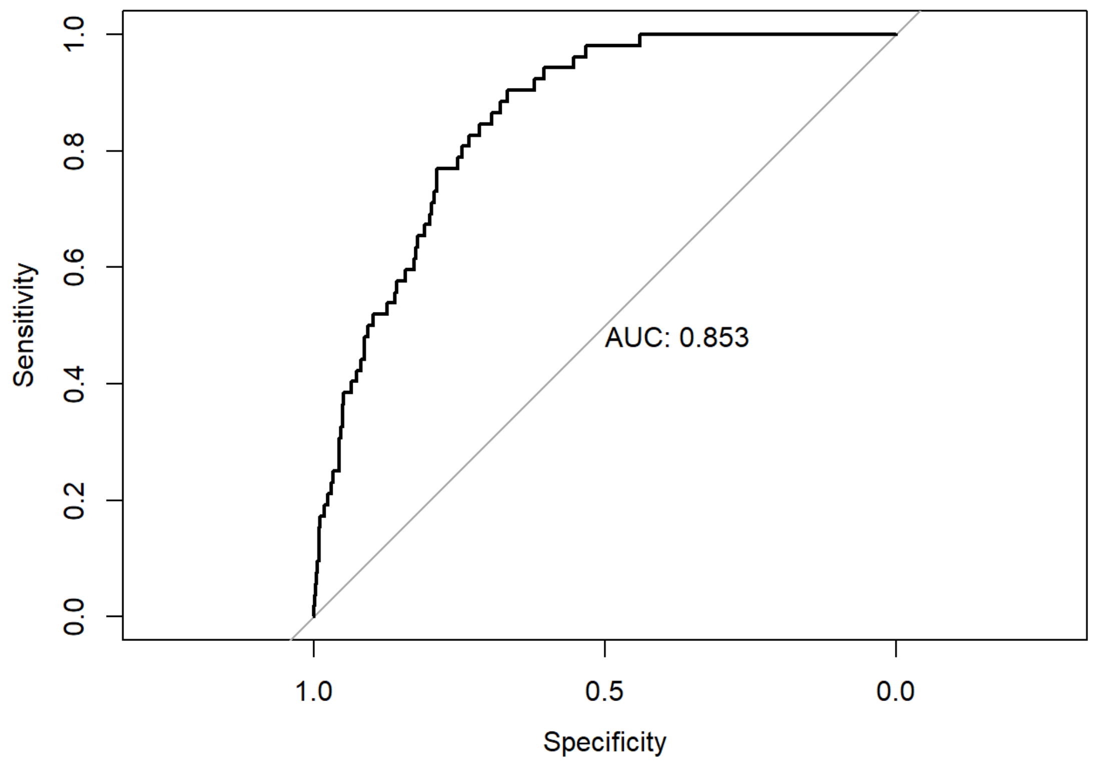
```
multi_logit <- glm(stroke ~ age + hypertension + bmi +avg_glucose_level, data = stroke_data_train,
family = "binomial")
  # predicts the logit model using the glm function, where the family is set to binomial
  print(summary(multi_logit))
```

```
##
## Call:
## glm(formula = stroke ~ age + hypertension + bmi + avg_glucose_level,
##     family = "binomial", data = stroke_data_train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.9951  -0.2987  -0.1612  -0.0758   3.5991
##
## Coefficients:
##                    Estimate Std. Error z value Pr(>|z|)
## (Intercept)       -7.830088   0.632542 -12.379  < 2e-16 ***
## age                0.069602   0.006429  10.826  < 2e-16 ***
## hypertension1      0.528220   0.198815   2.657  0.00789 **
## bmi                0.003900   0.013618   0.286  0.77458
## avg_glucose_level  0.004486   0.001466   3.060  0.00221 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1297.8  on 3680  degrees of freedom
## Residual deviance: 1038.0  on 3676  degrees of freedom
## AIC: 1048
##
## Number of Fisher Scoring iterations: 7
```

```
test_prob = predict(multi_logit, stroke_data_test, type = "response")
test_roc = roc(stroke_data_test$stroke ~ test_prob, plot = TRUE, print.auc = TRUE)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```



From the intercepts, we see that the coefficients on age, hypertension and average glucose level have significant t values. Therefore having a higher value for age, hypertension and average glucose level will increase the probability of a having stroke.

We can see that the AUC is better here (0.853>0.836) than simply using the ‘age’ variable as a predictor. This means that the model is more accurate using all four variables.

Overall, since most of the sample do not have stroke, it is different to make predictions about the probability of stroke as the variance is very high. However, if we manipulate the data and use a smaller subset where a higher proportion have had a stroke before, then we could be potentially overfitting the data on people who have had stroke before.

The pro of a logistic model is that it has a simple probabilistic interpretion. It is easy to implement, efficient to train and interpretable even for a novice. But it tends to underperform when there are non-linear decision boundaries

Implementing a gradient descent on simple linear regression

What is gradient descent? Gradient descent is an iterative optimisation algorithm for finding a local minimum of a function.The algorithm moves in the direction of the steepest descent, which is the negative of the gradient, for some number of steps, until some convergence criteria is satisfied.

Code is inspired by https://www.r-bloggers.com/2017/02/implementing-the-gradient-descent-algorithm-in-r/ (https://www.r-bloggers.com/2017/02/implementing-the-gradient-descent-algorithm-in-r/) and code from seminar 5. We took care to understand and explain every single step in the code.

Gradient descent will be implemented on linear regression:

$$(1)$$
$$y_i = \beta_0 + \beta_1 x_i + \epsilon$$

The loss function is:

$$(2)$$
$$L(\beta) = ||\mathbf{y} - \mathbf{X}\beta||^2$$

The MSE is:

$$(3)$$
$$MSE = (1/n)\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\beta)^2$$

The gradient of the loss function at a given point is:

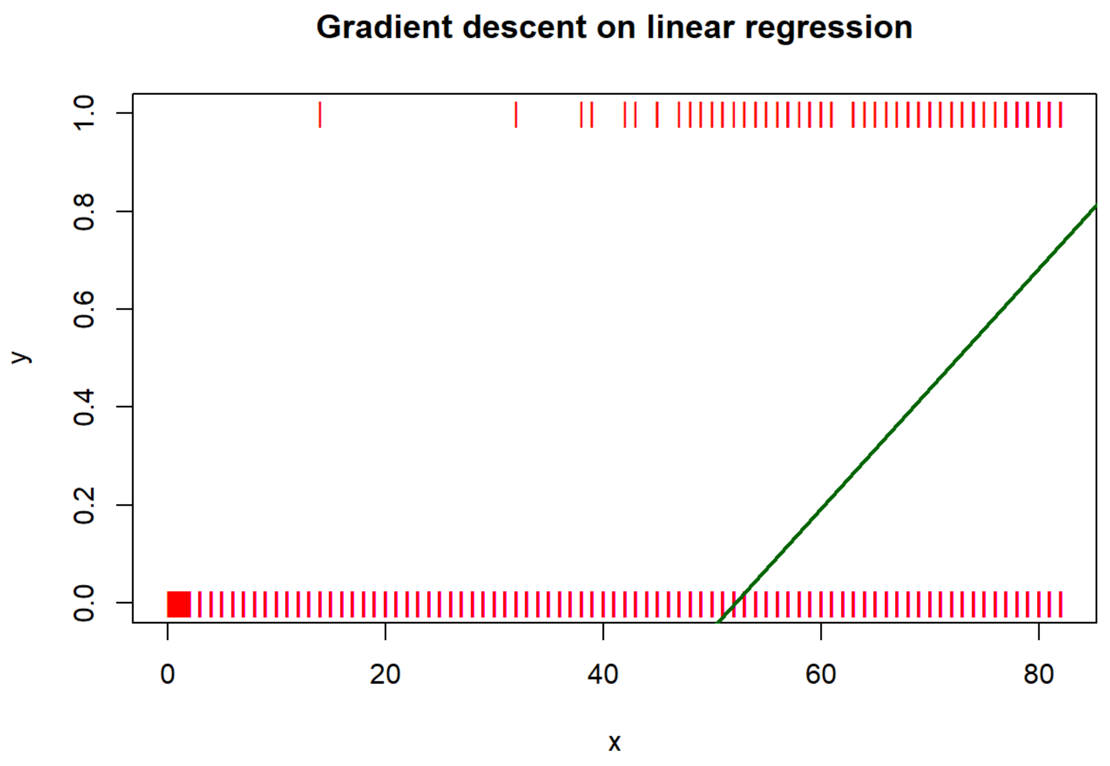
$$(4)$$
$$-2X'(y - X\hat{\beta})$$
$$(5)$$
$$\hat{\beta} = (X'X)^{-1}X'y$$

Gradient descent function

I will be generating a random point then computing the gradient of the loss function at this point, then updating the estimate for beta and then compute the gradient again, until some convergence criteria is satisfied.

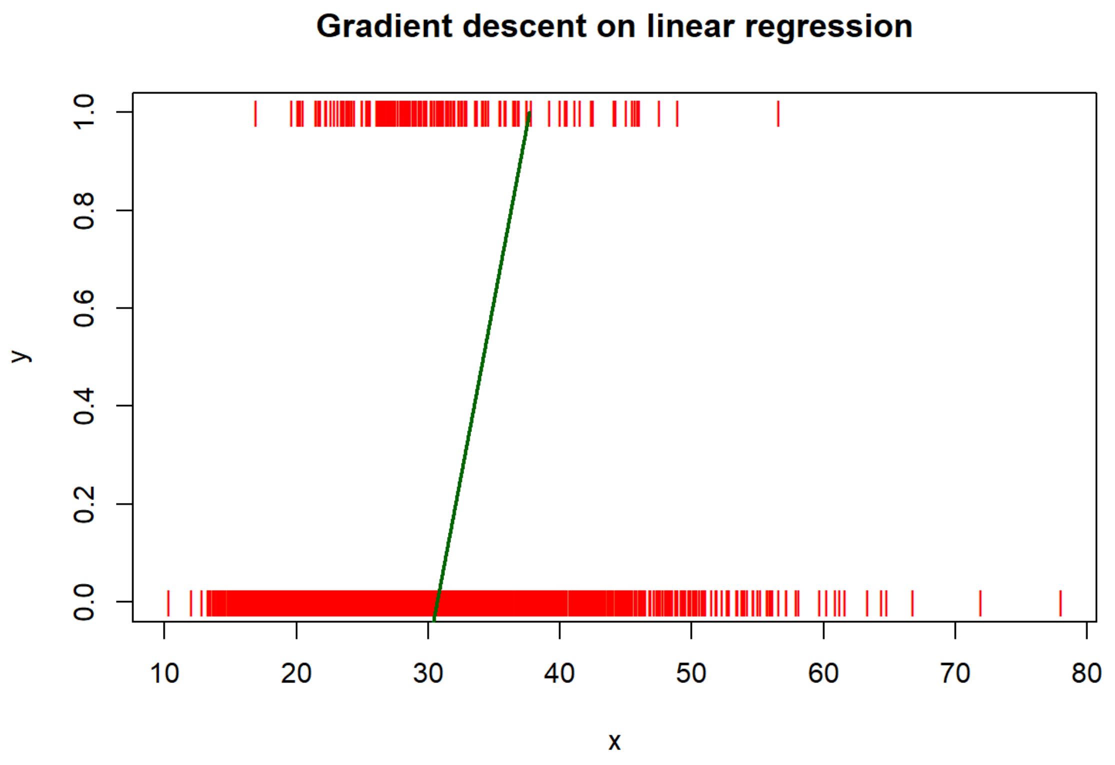
```
gradientDesc <- function(x, y, learn_rate, convergence_threshold, n, max_iter) {
  plot(x, y, col='red',pch='|', main='Gradient descent on linear regression')
  beta0<-runif(1,-10,10) #generating a random number between -10 and 10 as the starting point for b
eta
  c<-runif(1,-10,10) #enerating a random number between -10 and 10 as the starting point for the in
tercept
  yhat<-x*beta0+c # the formula for yhat as given by equation (1) above
  MSE <- sum((y-yhat)^2)/n # the formila for the mean square error as given by equation (3)
  converged = F # whether the model has converged is set to FALSE
  iterations = 0 # no. of iterations is 0
  while(converged == F) {
    # Implement the gradient descent algorithm
    beta_new <- beta0 - learn_rate * ((1 / n) * (sum((yhat - y) * x)))# updating the beta estimate
 by the learning rate, which is the magnitude of the steps the algorithm takes along the slope of t
he loss function
    c_new <- c - learn_rate * ((1 / n) * (sum(yhat - y))) # updating the incept in the same way
    beta0 <- beta_new # replacing the old beta estimate with the new one
    c <- c_new
    yhat <- beta0 * x + c #updating the new yhat using the new beta estimate
    MSE_new <- sum((y - yhat) ^ 2) / n #calculating the new loss function
  if(MSE - MSE_new <= convergence_threshold) {
    # if the difference between the new and old loss function is equal to some convergence threshol
d, then the function has converged, and the optimal intercept and slope are returned
    segments(x0 = 0, y0 = c, x1 = (1-c)/beta0, y1 = 1, col = "darkgreen", lwd=2)
      converged = T
      return(paste("Optimal intercept:", c, "Optimal slope:", beta0))

    }
    iterations = iterations + 1
    if(iterations > max_iter) { #if the number of interations has reached the maximum iterations se
t at the start of function, then the algorithm stops running and outputs the beta and c.
      segments(x0 = 0, y0 = c, x1 = (1-c)/beta0, y1 = 1, col = "darkgreen", lwd=2)  #the line estim
ated by the algorithm is plotted
      converged = T
      return(paste("Optimal intercept:", c, "Optimal slope:", beta0)) # the optimal intercept and s
lope are ouputted
    }
  }

}
```

```
##Gradient Descent for age as the predictor variable
gradientDesc(stroke_data_train$age,stroke_data_train$stroke, n=3682, learn_rate = 0.0001,convergenc
e_threshold = 0.000001, max_iter = 90000)
```



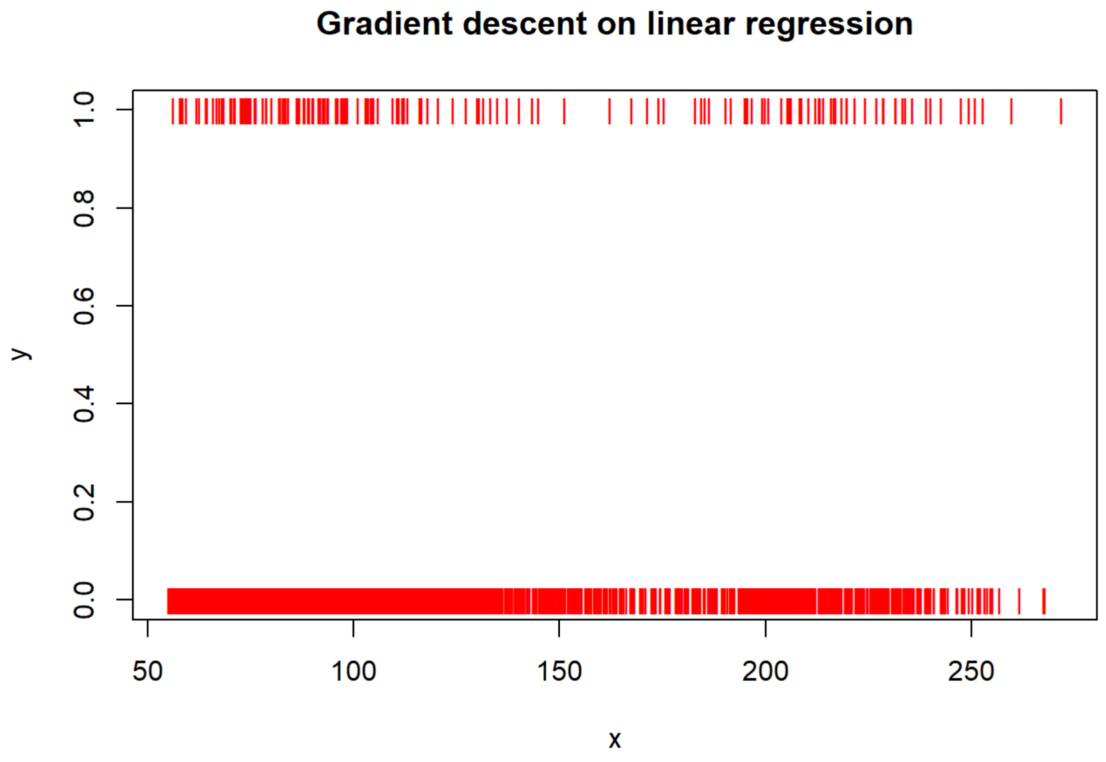
```
## [1] "Optimal intercept: -1.27994697823103 Optimal slope: 0.0245323206182435"
```

```
##Gradient Descent for BMI as the predictor variable
gradientDesc(stroke_data_train$bmi,stroke_data_train$stroke, n=3682, learn_rate = 0.001,convergence
_threshold = 0.0001, max_iter = 900)
```



```
## [1] "Optimal intercept: -4.38303062224729 Optimal slope: 0.142829251244234"
```

```
##Gradient Descent with average glucose level as the predictor variable
gradientDesc(stroke_data_train$avg_glucose_level,stroke_data_train$stroke, n=3682, learn_rate = 0.0
001,convergence_threshold = 0.000001, max_iter = 90000)
```



```
## [1] "Optimal intercept: 0.705212746666183 Optimal slope: -0.00520965220428644"
```

From running these gradient descent models, we find that, despite the relationship between all of the predictor variables and the outcome variable being positive (ie higher age-> higher chance of stroke), sometimes the gradient descent function breaks down and predicts a negative slope. This could be due to the fact that the step size was too large, and the algorithm ‘overshot’ going down the steepest descent. This is one of the key drawbacks of gradient descent- it can veer off in the wrong direction and give a completely wrong estimate!

Also, it is clear that a logistic probablity function is a much better model for this dataset as linear regression predicts negative intercept, which is not sensible.

Trees & Random Forest

By referring to the ideas in lecture on ‘trees’, we think that tree methods might be suitable for this sample. First, we have much more observations than predicting variables here, so this high complexity nature justifies the implementation of trees. Also we observe earlier the limited performance of linear models on this dataset and their incapabilities of handling missing values in a categorical variable, tree methods can be helpful in solving these obstacles.

Here we want outcome variable to be of type factor.

```
stroke_data_train$stroke <- factor(stroke_data_train$stroke);
stroke_data_test$stroke <- factor(stroke_data_test$stroke);
```

Single Tree

First, we train a single classification tree using the ‘rpart’ library. Here we fix tree_depth, which is the maximum level of the tree as 5 in order to obtain a relatively simple and interpretable model.’Cost_complexity’ is the tuning parameter indicating how much we should prune the tree to make it simpler.

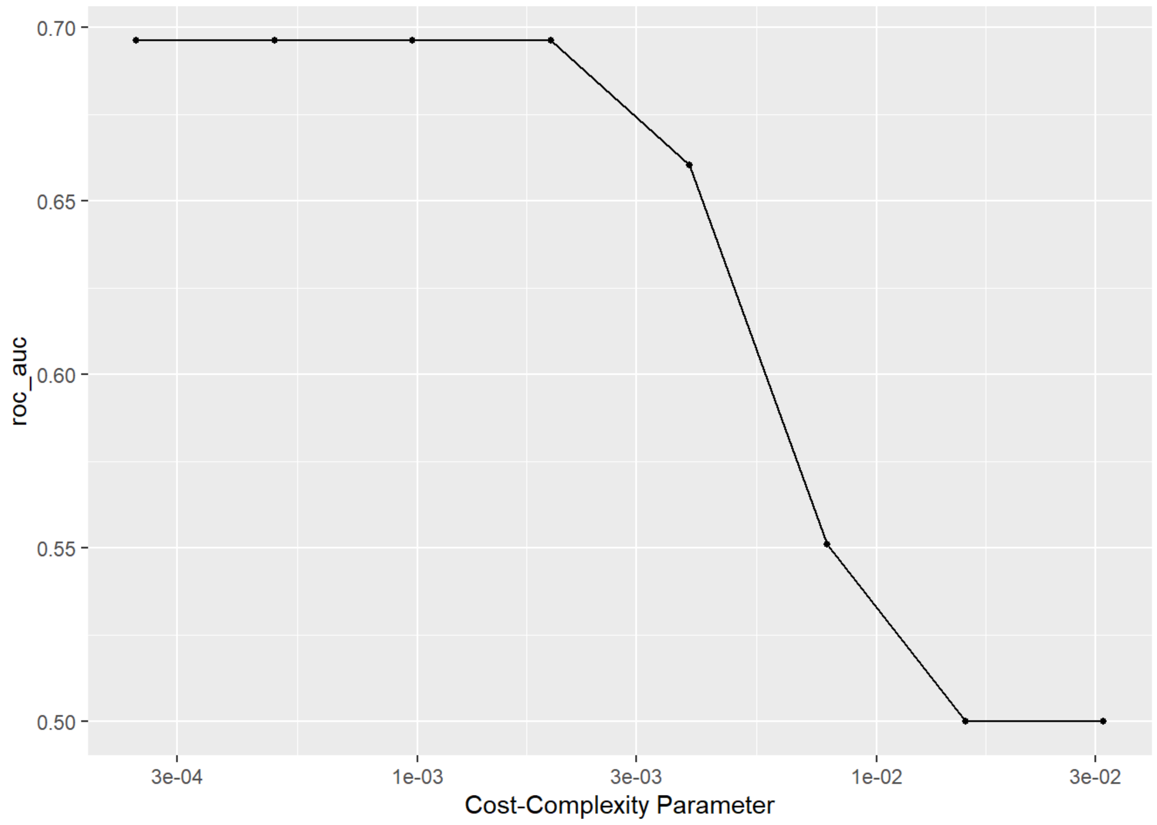
```
stroke_tree <- decision_tree(tree_depth = 5,
                             cost_complexity = tune("C")) %>%
  set_engine("rpart") %>%
  set_mode("classification")
# constructing the workflow
stroke_workflow_tree <- workflow() %>%
  add_recipe(stroke_recipe) %>%
  add_model(stroke_tree)
```

Here we tune the parameter ‘cost_complexity’ using the 10-fold cross validation subsets we formed earlier. We aim to tune over a grid of values of ‘cost_complexity’ and find the optimal value that maximizes ROC_AUC. By referring to the lecture, ROC_AUC refers to the area under the receiver operating characteristic curve, showing the trade-off between false positives and false negatives. We would like to maximize ROC_AUC, so the model is more accurate.

```
set.seed(1)
stroke_fit_tree <- tune_grid(
  stroke_workflow_tree,
  grid = data.frame(C = 2^(-12:-5)),
  stroke_data_cv,
  metrics = metric_set(roc_auc)
  )
```

A plot of ROC_AUC against the grid of values of ‘cost_complexity’ is presented here.

```
stroke_fit_tree %>% autoplot()
```



We select the model with the most optimized value of ‘cost_complexity’ and obtain the overall model specification.

```
stroke_tree_best <- stroke_fit_tree %>%
  select_best(metric = "roc_auc")
stroke_tree_final <- finalize_model(
  stroke_tree,
  stroke_tree_best)
stroke_tree_final
```

```
## Decision Tree Model Specification (classification)
##
## Main Arguments:
##   cost_complexity = 0.000244140625
##   tree_depth = 5
##
## Computational engine: rpart
```

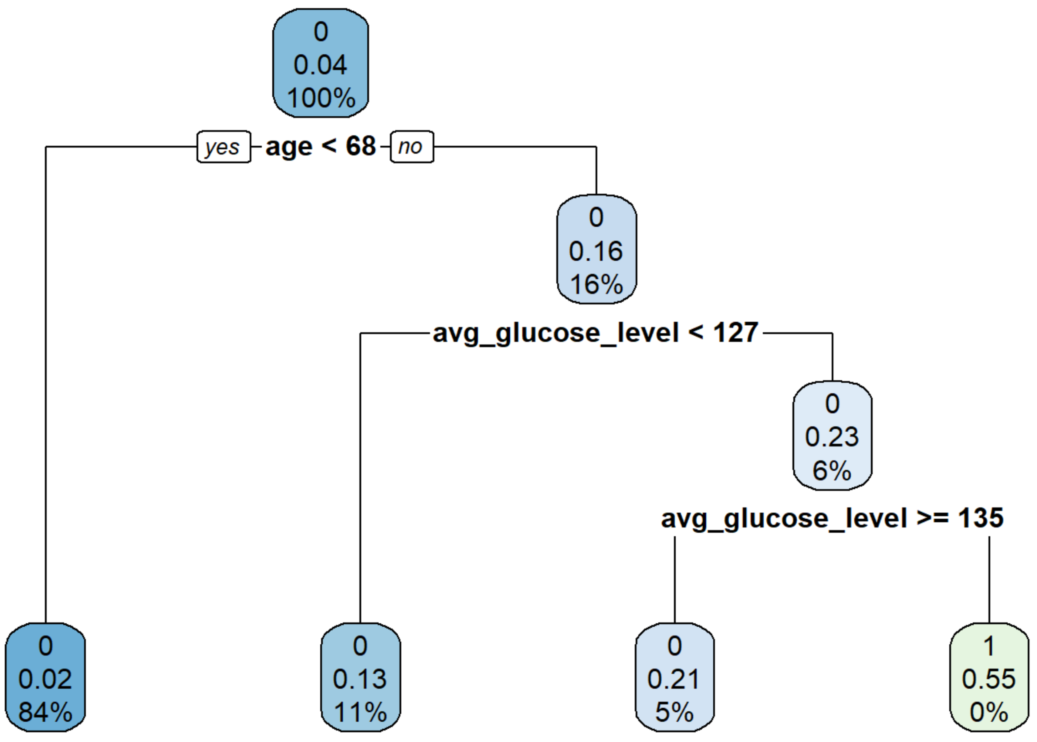
Fitting our model on the test set gives the accuracy and ROC_AUC presented below. The result will be discussed together with results from other tree methods at a later stage.

```
set.seed(1)
stroke_tree_test <-
  stroke_workflow_tree %>%
  update_model(stroke_tree_final) %>%
  last_fit(split = stroke_data_split) %>%
  collect_metrics()
stroke_tree_test
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.956 Preprocessor1_Model1
## 2 roc_auc  binary         0.716 Preprocessor1_Model1
```

This single classification tree is easily interpretable to a large extent. We can make use of the ‘rpart.plot’ library to get a visual representation of the decision tree. Here we cam observe the decision making path. This model is classifying all candidate with age above 68 and average glucose level above 135 as positive of getting stroke.

```
stroke_workflow_tree_fitted <- stroke_workflow_tree %>%
  update_model(stroke_tree_final) %>%
  parsnip::fit(stroke_data_train); # use the fit function from 'parship' pacakage

tree_fit <- stroke_workflow_tree_fitted %>%
  pull_workflow_fit();
rpart.plot(tree_fit$fit);
```



Bagging

Second, we implement bagging method here. By referring to the lecture slides, we will implement “bootstrap aggregating, i.e. resampling training data and averaging resulting models.”

Note that the original example given in lecture is done with engine ‘C5.0’. However, cost_complexity is only supposed to be a tuning parameter for "‘rpart’ engine.

‘Cost_complexity’ is still the tuning parameter but ‘tree-depth’ is increased to 10 to allow developments of more complicated models.

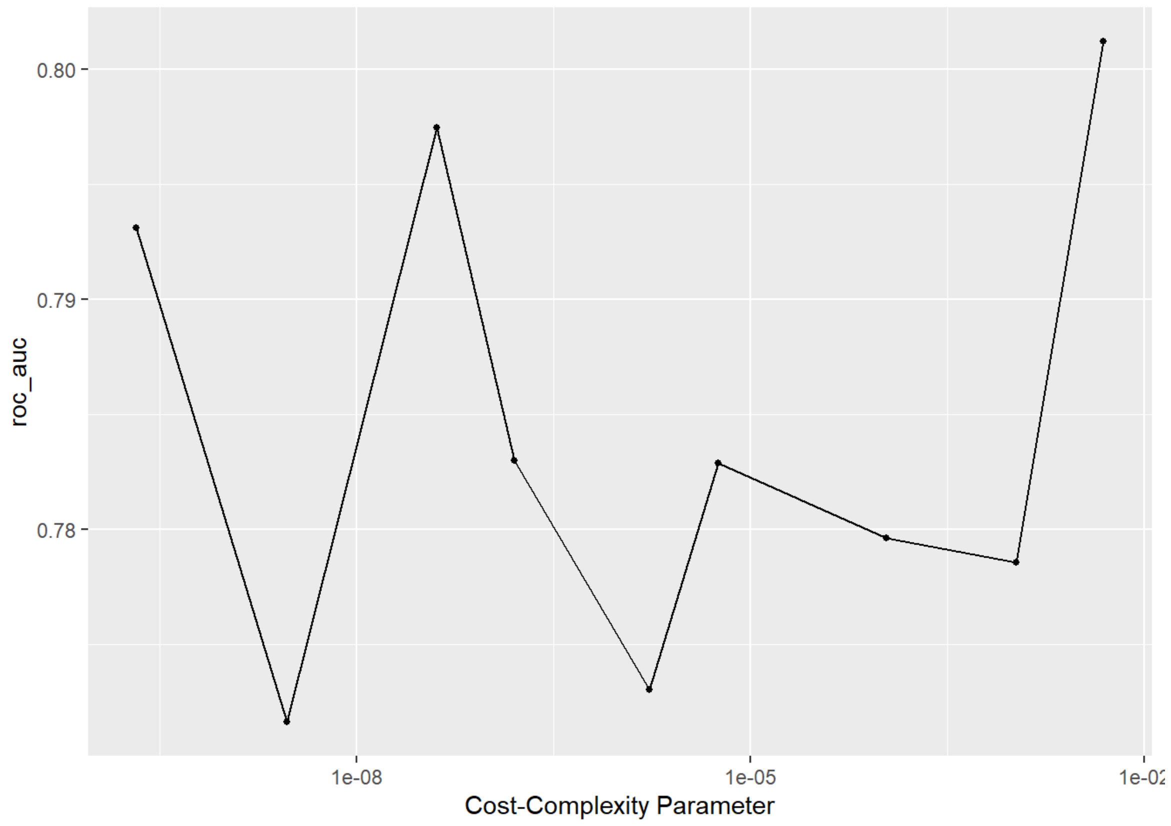
```r
stroke_bag <- bag_tree(tree_depth = 10,
                       cost_complexity = tune()) %>%
  set_engine("rpart") %>%
  set_mode("classification")
# constructing the workflow
stroke_workflow_bag <- workflow() %>%
  add_recipe(stroke_recipe) %>%
  add_model(stroke_bag)
```

Here again, we tune the parameter ‘cost_complexity’ using the 10-fold cross validation subsets we formed earlier. We aim to tune over a grid of values of ‘cost_complexity’ and find the optimal value that maximizes ROC_AUC.

```r
set.seed(1)
## this takes relatively long to run
stroke_fit_bag <- tune_grid(
  stroke_workflow_bag,
  stroke_data_cv,
  metrics = metric_set(roc_auc)
)
```

A plot of ROC_AUC against the grid of values of ‘cost_complexity’ is presented here.

```r
stroke_fit_bag %>% autoplot()
```



We select the model with the most optimized value of ‘cost_complexity’ and obtain the overall model specification.

```
stroke_bag_best <- stroke_fit_bag %>%
  select_best()
stroke_bag_final <- finalize_model(
  stroke_bag,
  stroke_bag_best)
stroke_bag_final
```

```
## Bagged Decision Tree Model Specification (classification)
##
## Main Arguments:
##   cost_complexity = 0.00487080891502241
##   tree_depth = 10
##   min_n = 2
##
## Computational engine: rpart
```

Fitting our bagging model on the test set gives the accuracy and ROC_AUC presented below. The result will be discussed together with other results at a later stage.

```
stroke_bag_test <-
  stroke_workflow_bag %>%
  update_model(stroke_bag_final) %>%
  last_fit(split = stroke_data_split) %>%
  collect_metrics()
stroke_bag_test
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.957 Preprocessor1_Model1
## 2 roc_auc  binary         0.841 Preprocessor1_Model1
```

Random Forest

Third, we implement random forest here. By referring to the lecture slides, we will “randomly drop predictors when resampling” in order to get a better accuracy overall.

Here we fix the ‘trees’ parameter, which is the total number of trees we grow to 100.’Mtry’ is the tuning parameter here, by referring to the official document, it indicates the “number of predictors that will be randomly sampled at each split when creating the tree models.”

```
stroke_rf <-
  rand_forest(trees = 100, mtry = tune()) %>%
  set_mode("classification") %>%
  set_engine("randomForest")
#constructing the workflow
stroke_workflow_rf <- workflow() %>%
  add_recipe(stroke_recipe) %>%
  add_model(stroke_rf)
```

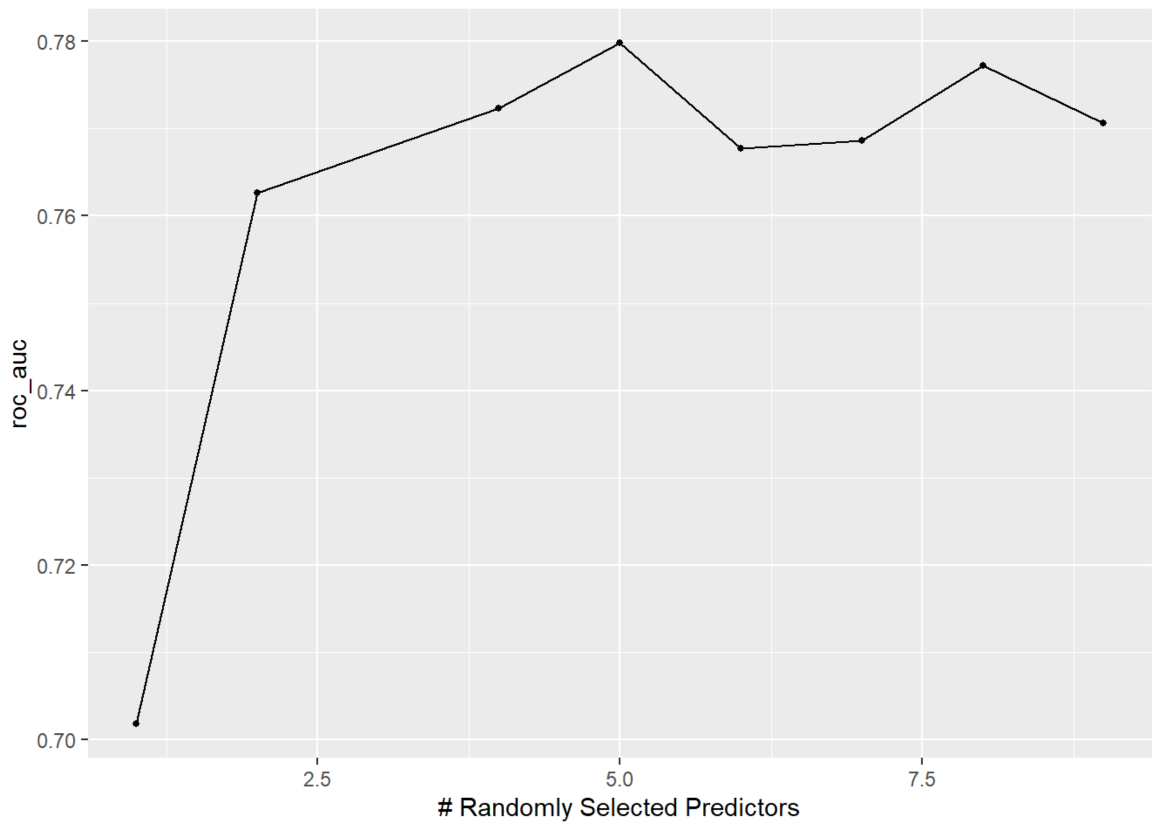
By using the same method as before, we tune the parameter ‘mtry’ using the 10-fold cross validation subsets we formed earlier. We aim to tune over a grid of values of ‘cost_complexity’ and find the optimal value that maximizes ROC_AUC.

```
set.seed(1)
stroke_fit_rf <- tune_grid(
  stroke_workflow_rf,
  stroke_data_cv,
  metrics = metric_set(roc_auc)
)
```

```
## i Creating pre-processing data to finalize unknown parameter: mtry
```

A plot of ROC_AUC against the grid of values of 'mtry' is presented here.

```
stroke_fit_rf %>% autoplot()
```



We select the model with the most optimized value of 'mtry' and obtain the overall model specification.

```
stroke_rf_best <- stroke_fit_rf %>%
  select_best(metric = "roc_auc")
stroke_rf_final <-
  finalize_model(
    stroke_rf,
    stroke_rf_best
  )
stroke_rf_best
```

```
## # A tibble: 1 x 2
##    mtry .config
##   <int> <chr>
## 1     5 Preprocessor1_Model1
```

Fitting our random forest model on the test set gives the accuracy and ROC_AUC presented below.

```
set.seed(1)
stroke_rf_test <-
  stroke_workflow_rf %>%
  update_model(stroke_rf_final) %>%
  last_fit(split = stroke_data_split) %>%
  collect_metrics()
stroke_rf_test
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.958 Preprocessor1_Model1
## 2 roc_auc  binary         0.812 Preprocessor1_Model1
```

Now we collect together the accuracy and ROC_AUC obtained from 3 tree methods.

```
tree_results <- data.frame(model=c('single_tree','bagging','random_forest'),
                 accuracy=c(stroke_tree_test$.estimate[1],
                            stroke_bag_test$.estimate[1],
                            stroke_rf_test$.estimate[1]),
                 ROC_AUC=c(stroke_tree_test$.estimate[2],
                           stroke_bag_test$.estimate[2],
                           stroke_rf_test$.estimate[2]))
tree_results
```

```
##           model  accuracy   ROC_AUC
## 1   single_tree 0.9559902 0.7160065
## 2       bagging 0.9568052 0.8411538
## 3 random_forest 0.9576202 0.8121604
```

As observed, it seems that all three methods obtain relatively high accuracy and fair ROC_AUC. However, it's important to recognize the fact that 95.76% observations in the test set actually have negative stroke outcome. This indicates the idea that, due to the fact that our original dataset is largely imbalanced and biased, no model here can be better than random guesses.

If we look into the prediction result, it's realized that our tree models are predicting negative for almost all of the observations. Although this gives statistically fair accuracy and ROC_AUC, in the context of medical examination, the models are rather useless since they fail to identify majority of the candidates that get stroke, hence they can't be use to identify other people who have the potential of getting stroke either.

```
table(stroke_data_test$stroke)
```

```
##
##    0    1
## 1175   52
```

Threshold Adjustment

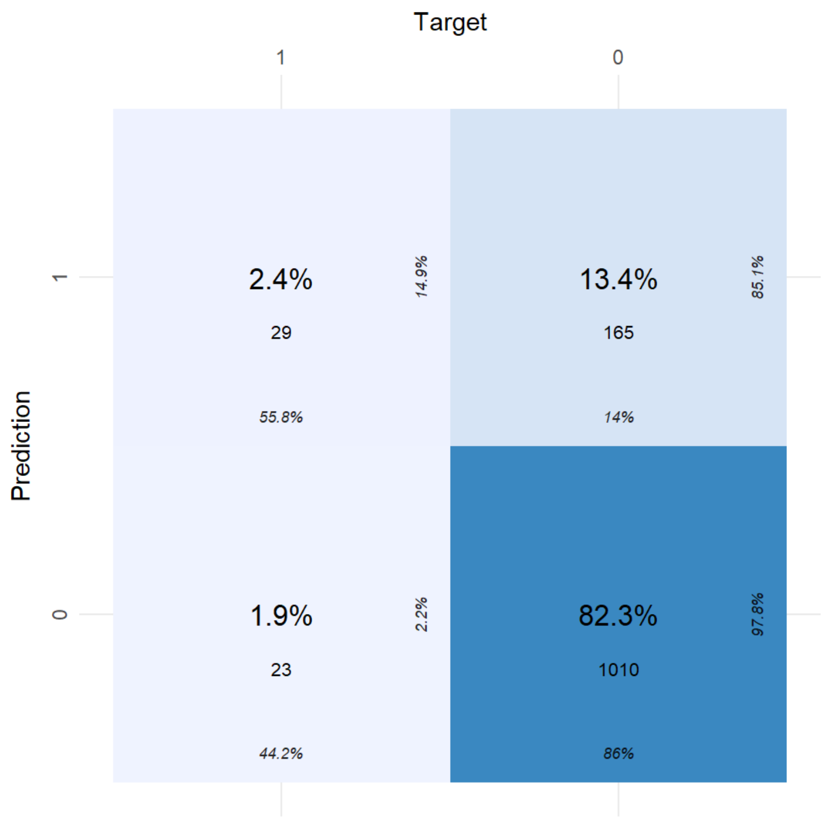
Since our aim is to identify potential stroke from simple variables, i.e. these predicting variables can be obtained by an individual without undertaking any proper medical check, we would like to reduce the false-negatives as much as possible even if it leads to a rise in false-positives. In simple terms, we would rather ask more candidates to undertake proper medical checks than not identifying people with high potential of developing stroke.

One approach we can take is to adjust the threshold of identifying an observation as positive when predicting on the test set. We can calculate the percentage of positive outcomes in the training set, and use it as the adjusted threshold.

```
train_table <- table(stroke_data_train$stroke)
adj_threshold <- train_table[2] / (train_table[1] + train_table[2])
```

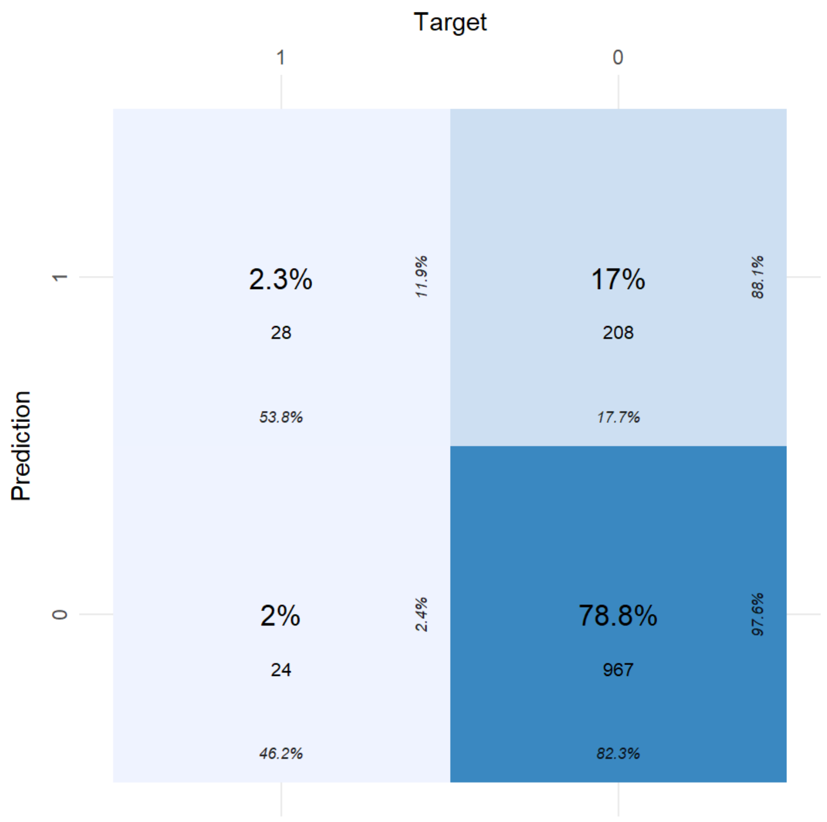
First, we implement the adjusted threshold for the single tree model and plot the confusion matrix.

```
set.seed(1)
a <- stroke_workflow_tree %>% #fit the tree on test set
  update_model(stroke_tree_final) %>%
  last_fit(split = stroke_data_split)
tree_predict <- as.data.frame(a$.predictions) #get the original prediction
tree_predict$pred_adj = tree_predict$.pred_1
tree_predict <- tree_predict %>%
   mutate(across(c("pred_adj"), ~ifelse(.>=adj_threshold, 1, 0))) #re-adjust threshold
tree_predict_tibble <- tibble("actual" = tree_predict$stroke, "prediction" = tree_predict$pred_ad
j);
tree_predict_table <- table(tree_predict_tibble)
cfm_tree <- tidy(tree_predict_table) #construct the confusion matrix
plot_confusion_matrix(cfm_tree, target_col = "actual", prediction_col = "prediction", counts_col =
"n")
```



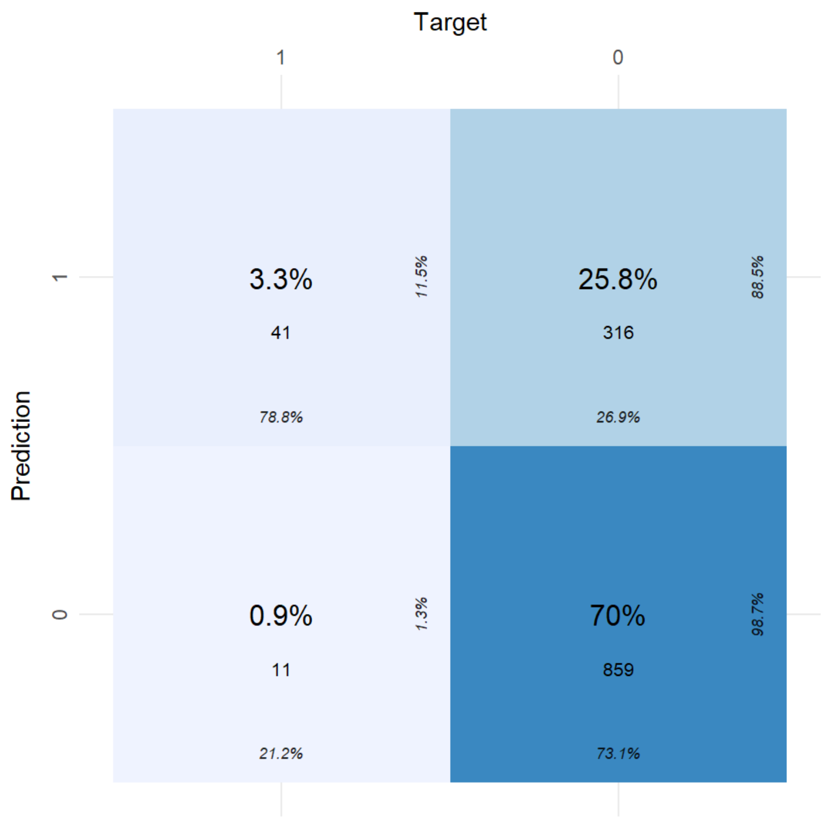
Second, we implement the adjusted threshold for the bagging model and plot the confusion matrix.

```
set.seed(1)
a <- stroke_workflow_bag %>% #fit the bagging model on test set
  update_model(stroke_bag_final) %>%
  last_fit(split = stroke_data_split)
bag_predict <- as.data.frame(a$.predictions) #get the original prediction
bag_predict$pred_adj = bag_predict$.pred_1
bag_predict <- bag_predict %>%
   mutate(across(c("pred_adj"), ~ifelse(.>=adj_threshold, 1, 0))) #re-adjust threshold
bag_predict_tibble <- tibble("actual" = bag_predict$stroke, "prediction" = bag_predict$pred_adj);
bag_predict_table <- table(bag_predict_tibble)
cfm_bag <- tidy(bag_predict_table) #construct the confusion matrix
plot_confusion_matrix(cfm_bag, target_col = "actual", prediction_col = "prediction", counts_col =
"n")
```



Third, we implement the adjusted threshold for the random forest model and plot the confusion matrix.

```
set.seed(1)
a <- stroke_workflow_rf %>% #fit the random forest model on test set
  update_model(stroke_rf_final) %>%
  last_fit(split = stroke_data_split)
rf_predict <- as.data.frame(a$.predictions) #get the original prediction
rf_predict$pred_adj = rf_predict$.pred_1
rf_predict <- rf_predict %>%
   mutate(across(c("pred_adj"), ~ifelse(.>=adj_threshold, 1, 0))) #re-adjust threshold
rf_predict_tibble <- tibble("actual" = rf_predict$stroke, "prediction" = rf_predict$pred_adj);
rf_predict_table <- table(rf_predict_tibble)
cfm_rf <- tidy(rf_predict_table) #construct the confusion matrix
plot_confusion_matrix(cfm_rf, target_col = "actual", prediction_col = "prediction", counts_col =
"n")
```



As observed, although adjusting the threshold decreases the accuracy of all three models, it also helps to reduce the false-negatives. The random forest model successfully reduce the false negative rate and still keeps it below the false positive rate.

So we have a significant increase in sensitivity while sacrificing our models' specificity. Such boost in sensitivity of our models help to identify most of the positives correctly.

Hence, we shall conclude that tree methods are capable of extracting useful information and make meaningful interpretation on this dataset and can potentially be used to identify people with higher risk of getting stroke if we adjust the threshold.

Regarding real-life application, we should always use a model of relatively high sensitivity in context like this. Depending on the actual cost of having false positive, i.e. the cost for further medical checks, we can should use the model that reaches the most suitable balance between sensitivity and specificity.

Neural Network Model

We make sure that the outcome variable is of type integer

```
stroke_data_train$stroke <- as.integer(stroke_data_train$stroke) - 1;
stroke_data_test$stroke <- as.integer(stroke_data_test$stroke) - 1;
```

For the neural network to work it has to work with the categorical variables have to be in numerical form. Therefore, we replace these variables with dummy variables

```
stroke_data_train2 <- dummy_cols(stroke_data_train, select_columns = c("gender", "hypertension", "h
eart_disease", "smoking_status", "ever_married", "work_type", "Residence_type"), remove_first_dummy
= TRUE, remove_selected_columns = TRUE)
stroke_data_test2 <- dummy_cols(stroke_data_test, select_columns = c("gender", "hypertension", "hea
rt_disease", "smoking_status", "ever_married", "work_type", "Residence_type"), remove_first_dummy =
TRUE, remove_selected_columns = TRUE)
```

To understand what the functions above do it is helpful to look for example at the variable gender. In the original training dataset, an observation's gender was determined whether the gender column contain "Male" or "Female". Now, in this new training dataset, the gender of an observation is determined whether the column gender_Male contains a "1" meaning the gender is "Male" or a "0" meaning the gender is "Female.

```
glimpse(stroke_data_train)
```

```
## Rows: 3,681
## Columns: 11
## $ gender            <fct> Male, Male, Female, Female, Male, Male, Female, Fema~
## $ age               <dbl> 67, 80, 49, 79, 81, 74, 78, 61, 79, 50, 64, 60, 52, ~
## $ hypertension      <fct> 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0~
## $ heart_disease     <fct> 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0~
## $ ever_married      <fct> Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Ye~
## $ work_type         <fct> Private, Private, Private, Self-employed, Private, P~
## $ Residence_type    <fct> Urban, Rural, Urban, Rural, Urban, Rural, Urban, Rur~
## $ avg_glucose_level <dbl> 228.69, 105.92, 171.23, 174.12, 186.21, 70.09, 58.57~
## $ bmi               <dbl> 36.6, 32.5, 34.4, 24.0, 29.0, 27.4, 24.2, 36.8, 28.2~
## $ smoking_status    <fct> formerly smoked, never smoked, smokes, never smoked,~
## $ stroke            <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
```

```
glimpse(stroke_data_train2)
```

```
## Rows: 3,681
## Columns: 16
## $ age                           <dbl> 67, 80, 49, 79, 81, 74, 78, 61, 79, 50, ~
## $ avg_glucose_level             <dbl> 228.69, 105.92, 171.23, 174.12, 186.21, ~
## $ bmi                           <dbl> 36.6, 32.5, 34.4, 24.0, 29.0, 27.4, 24.2~
## $ stroke                        <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ gender_Male                   <int> 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0~
## $ hypertension_1                <int> 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0~
## $ heart_disease_1               <int> 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0~
## $ `smoking_status_never smoked` <int> 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1~
## $ smoking_status_smokes         <int> 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0~
## $ smoking_status_Unknown        <int> 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0~
## $ ever_married_Yes              <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1~
## $ work_type_Govt_job            <int> 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0~
## $ work_type_Never_worked        <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ work_type_Private             <int> 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0~
## $ `work_type_Self-employed`     <int> 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1~
## $ Residence_type_Urban          <int> 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1~
```

Divide data in features and outcome

```
set.seed(1)
#First on training dataset
#Shuffle training dataset
rand <- sample(nrow(stroke_data_train2))
stroke_data_train2 <- stroke_data_train2[rand,]
x_train <- cbind(stroke_data_train2[,1:3], stroke_data_train2[,5:16])
y_train <- stroke_data_train2[,4]
#Second on testing dataset
#Shuffle testing dataset
rand <- sample(nrow(stroke_data_test2))
stroke_data_test2 <- stroke_data_test2[rand,]
x_test <- cbind(stroke_data_test2[,1:3], stroke_data_test2[,5:16])
y_test <- stroke_data_test2[,4]
```

Make sure all the dummy variables are of type integer so the algorithm below can work properly.

```
class(x_train$hypertension)
```

```
## [1] "integer"
```

```
x_train$hypertension <- as.integer(x_train$hypertension)
class(x_train$hypertension)
```

```
## [1] "integer"
```

```
x_train$heart_disease <- as.integer(x_train$heart_disease)
x_test$hypertension <- as.integer(x_test$hypertension)
x_test$heart_disease <- as.integer(x_test$heart_disease)
class(y_train)
```

```
## [1] "numeric"
```

Turn datasets from dataframe to matrix in order to feed into algorithm.

```
x_train2 <- data.matrix(x_train)
x_test2 <- data.matrix(x_test)
y_train2 <- data.matrix(y_train)
y_test2 <- data.matrix(y_test)
```

In order to run the Neural Network we will use the package 'keras'. The Keras R interface uses the TensorFlow backend engine by default. For more information about the package refer to: https://cran.r-project.org/web/packages/keras/vignettes/index.html (https://cran.r-project.org/web/packages/keras/vignettes/index.html)

The neural network works the following way:

$$\text{Step 1: We input the matrix } X = [X_1, \dots, X_p]^T$$

$$\text{p being the total number of parameters and each } X_i \text{ is one column of x\_train2 or x\_test2.}$$

$$\text{Step 2: We then run a linear regression on X to produce } Z^2 = \beta_0^1 + X\beta^1$$

$$\text{Note : } \beta^k = [\beta_1^k, \dots, \beta_M^k]^T \text{ (M is the number of units in layer k)}$$

$$\text{Step 3: These results are then introduced into an activation function } a^2 = \sigma(Z^2)$$

$$\text{In this neural network we use two different types of activation functions:}$$

$$\text{relu (Rectified Linear unit): } \sigma(Z^k) = \max\{0, Z^k\}$$

\text{and sigmoid } g(Z^{k}) = \frac{1}{1 + e^{-Z^{k}}

$$\text{The matrix } a^2 \text{ is then passed into the next layer. We go back to step 2 using } a^2 \text{ instead of X.}$$

$$\text{This process is done repetitively until we reach the final layer which outputs } a^L \text{ or equivalently Y, the predicted outcome.}$$

$$\text{In conclusion the neural network does this: } Y = g(\sigma(\sigma(\sigma(\beta_0^1 + X\beta^1))))$$

$$\text{The dropout layer is a method of regularization that randomly does not use a unit of the layer with probability 0.5.}$$

```
#Input units
p <- length(x_train2[1,]) #We are going to input all the features into the neural net. The algorith
m will then create a 80 different variables using these features.
#Output units
y <- length(y_train2[1]) #The predicted stroke value
NN_model <- keras_model_sequential() #We will stack the layers linearly
NN_model %>%
  layer_dense(units = 80, activation = 'relu', input_shape = c(p)) %>%
  layer_dropout(rate = 0.5) %>% #Dropout layer for regularization
  layer_dense(units = 40, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>% #Dropout layer for regularization
  layer_dense(units = 20, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>% #Dropout layer for regularization
  layer_dense(units = y, activation = 'sigmoid')
summary(NN_model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## dense_3 (Dense)                     (None, 80)                      1440
## ________________________________________________________________________________
## dropout_2 (Dropout)                 (None, 80)                      0
## ________________________________________________________________________________
## dense_2 (Dense)                     (None, 40)                      3240
## ________________________________________________________________________________
## dropout_1 (Dropout)                 (None, 40)                      0
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 20)                      820
## ________________________________________________________________________________
## dropout (Dropout)                   (None, 20)                      0
## ________________________________________________________________________________
## dense (Dense)                       (None, 1)                       21
## ================================================================================
## Total params: 5,521
## Trainable params: 5,521
## Non-trainable params: 0
## ________________________________________________________________________________
```

As you can see the network has to find the optimum of 5521 parameters these are all the weight

$$\beta_i^k$$

.

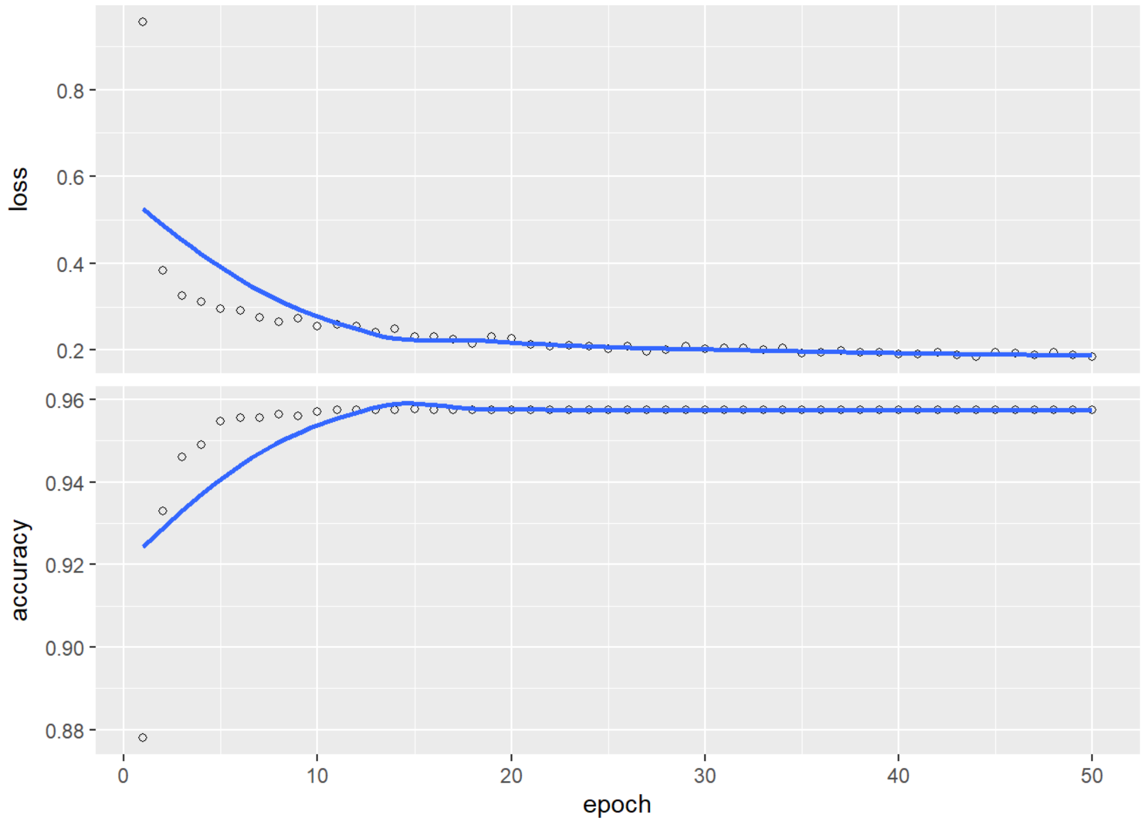
We compile the model with appropriate loss function, optimizer, and metrics. The loss function used here is called binary cross-entropy also known as log loss. This is the same function used in the logistic model. The optimizer algorithm used is stochastic gradient descent. Stochastic gradient descent randomly picks one observation from the whole data set instead of using all the observations to calculate the derivatives. This reduces the running time of the algorithm.

```
NN_model %>% compile(
  loss = 'binary_crossentropy', #Loss function
  optimizer = optimizer_sgd(),
  metrics = c('accuracy') #The objective of the algorithm is to achieve the best possible accuracy
)
```

Train the model for 50 epochs using batches of 100

```
set.seed(1)
history <- NN_model %>% fit(
  x_train2, y_train2,
  epochs = 50, batch_size = 100)
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



The final result on our test set

```
NN_model %>% evaluate(
  x_test2, y_test2)
```

```
##      loss  accuracy
## 0.1792321 0.9576202
```

We see that the model performs with an accuracy beating the rest of the models. Nevertheless, it is the least interpretable.

Conclusion

We used many different models in this project to try and predict the likelihood of stroke for patients. The linear regression model is one of the easiest to interpret and implement. However, since the output is binary, it was not as good a model as the logistic model for fit. It is easier to interpret than the logistic model because the coefficient estimates are easiest to understand. One increment increase in the predictor variable leads to a proportional increase in the outcome variable, with the proportional increase being the coefficient value.

In the logistic model since most of the sample do not have stroke, the curve is not very sigmoid shaped. It does not predict anyone with a high (>0.5) chance of stoke. The logistic model overall is an easy model to implement, and is relatively easy to interpret as the output can be interpretted probabilistically. It tends to underperform when there are non-linear decision boundaries.

The gradient descent model was a bit more difficult to implement than the logistic and OLS models. Sometimes, if the parameters were not set correctly, it gave estimates that were not sensible. For example, the relationship between all of the predictor variables and the outcome variable is clearly positive (ie higher bmi-> higher chance of stroke), but sometimes the model predicted a negative relationship. This means that we have to choose the step size parameter very carefully to ensure that the algorithm does not ‘overshot’ going down the steepest descent, and miss the minimum point. Main disadvantages of the gradient descent model include: having to find the ‘right’ parameters manually and needing a higher computing power when the number of max iterations is set to a very high number.

We found that the tree method was able to extract useful information and we were able to make meaningful interpretations based on this model. It can be used to identify patients at a higher risk of getting a stroke if the threshold is adjusted. However, in real-life application, a model of high sensitivity needs to be used in a medical context such as this one. Depending on the cost of having false positives (ie being predicted to have a stroke when you are extremely unlikely to have one and having to pay for cost of further medical checks), we should use the model that reaches the most suitable balance between sensitivity and specificity

The final model, the deep learning neural network model, is the model with the highest precision. It is however, the least interpretable model out of all of our models. It is extremely difficult for a human to understand the relationships between 141 nodes (ignoring the input variables as nodes), specially if in each layer they are randomly not used with a probability of 0.5.

If we inspect the original dataset, we see that the dataset is very biased.

```
stroke_data %>% count(stroke)
```

```
##   stroke    n
## 1      0 4860
## 2      1  249
```

We see that roughly 95% of the observations did not have a stroke. This means that if we randomly predicted that a person would not have a stroke with 0.95 probability, we would perform better than our interpretable models. The only model that performs better than this model would be the deep learning model but it is very likely that is creating a signal out of noise from the data.

We tried balancing the data through imputation(the code we wrote for this is attached in a separate file) so around half of the observations would have had a stroke but this simply introduced more bias and made our models less accurate and interpretable.

Therefore, we conclude that our results are not significant not because the poor quality of our models but the poor quality of our data. However, in this case the fault is not of the data collector but of the nature of the data at it is very improbable for a person to have a stroke.