Gradient Methods for Classification

Kieran Morris

2023-11-07

In this document we will explore how optimisation algorithms are employed in logistic regression, demonstrating limitations of simple methods like gradient descent and utilising the options of `optimise` to allow us to employ more complicated newton-type methods for optimisation.

Logistic Regression

Logistic regression, despite its name, is not a regression algorithm - it is in fact a classification algorithm. For the context of this document we will restrict ourselves to binary classification, however this can apply to multiclass classification too.

Suppose we have data set $\{\mathbf{x}_i\}_{i=1}^n \subset \mathbb{R}^d$ and for each $\mathbf{x}_i$ we have an assigned *class* $y_i = y(\mathbf{x}_i) \in \{1, -1\}$. Then we consider parametric model $f : \mathbb{R}^d \to \mathbb{R}$ which we denote by $f = f(\mathbf{x}; w)$, this can be anything, including a generalised linear model or some arbitrary feature transform. Then we consider the following function:

$$\sigma_w(x) = \sigma(f) = \frac{1}{1 + e^{-f}}$$

where σ is the *Sigmoid* function and again we have $f = f(x; w)$. The output of this function is some value between 0 and 1 which can be interpreted as the probability of each $\mathbf{x}$ being assigned a 1. Using this (and some algebraic trickery) we construct the density function:

$$p(y \mid x; w) = \sigma(f(x; w) \cdot y)$$

and then the negative log-likelihood over our observed data:

$$\mathbf{Obj}(\mathbf{w}) = -\sum_{i=1}^{n} \ln \sigma(f(x_i; w) \cdot y_i).$$

For which maximising this essentially finds a maximum likelihood estimate for classification - and as a bonus we get a boundary line where $f(x; w) = 0$ (or equivilently $p(y \mid x; w) = 0.5$). Note that different sources may have different constructions of **Obj** including if $y \in \{0, 1\}$ instead of $y \in \{1, -1\}$, howver our version is **Obj**ectively the prettiest.

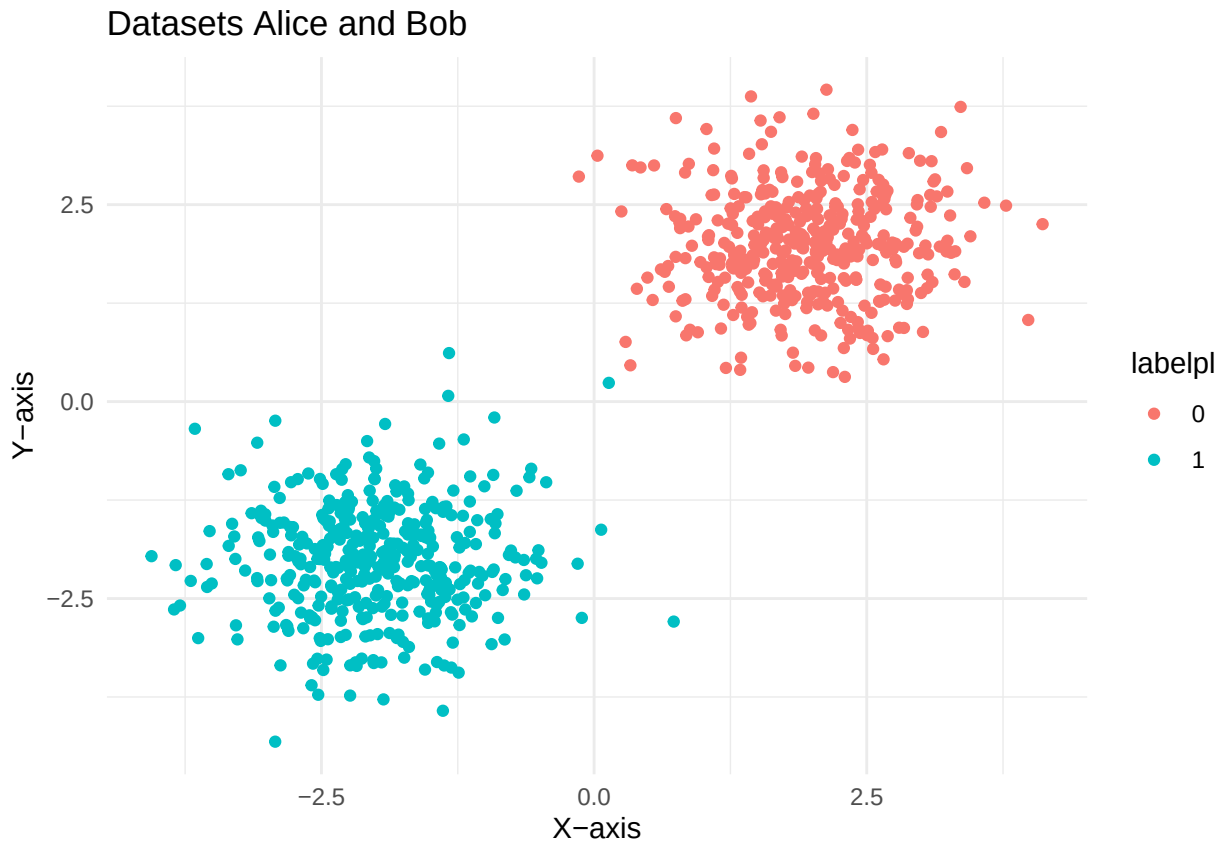
Unlike direct regression problems, there is no exact solution for this optimisation problem, so we must resort to numerical estimation. Fortunately our objective function is a differentiable multivariate function, which we know how to minimise.

The following is an implimentation of Logistic Regression using the built in `glm` function.

Toy Example: Linearly Seperable Data

Consider the following linearly seperable data set:

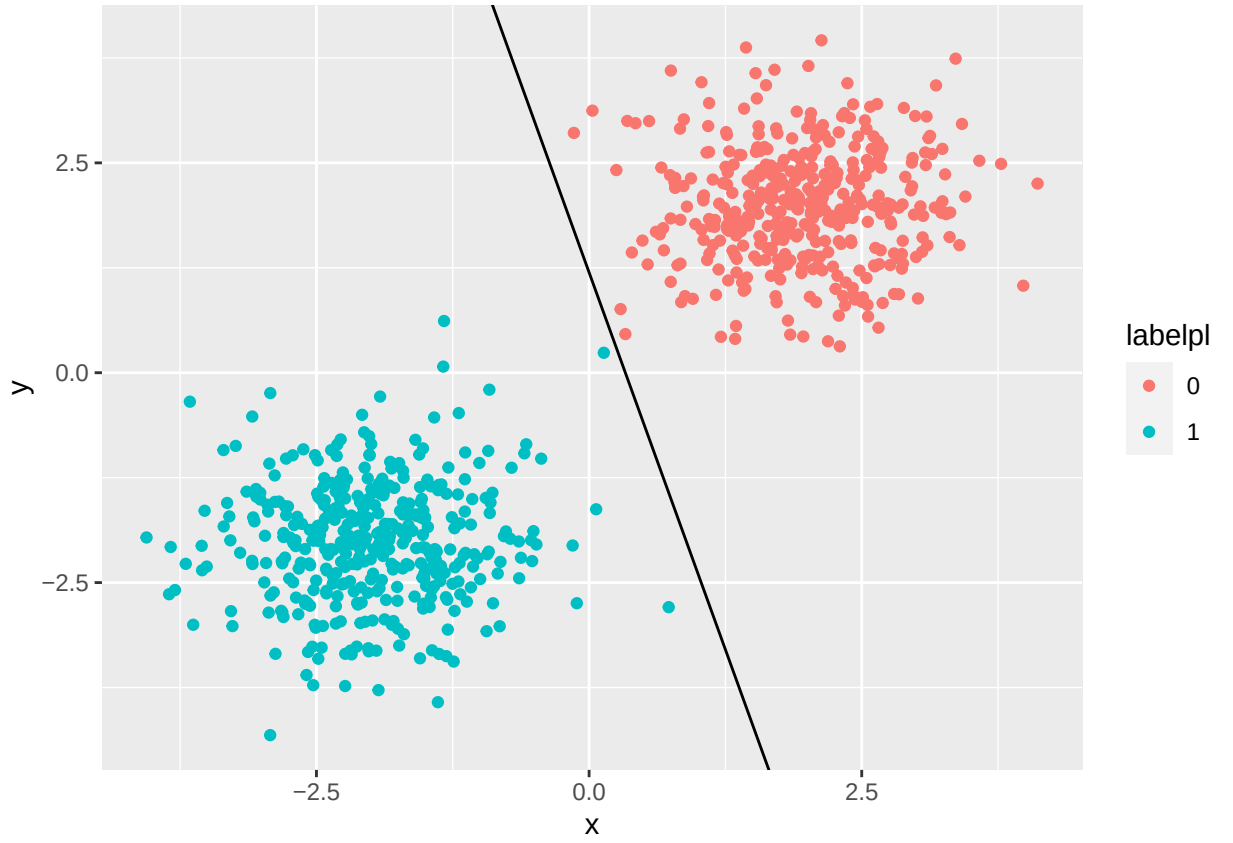
```r
set.seed(11)

# Parameters for the distributions--------------/
mean1 <- c(2, 2)
mean2 <- c(-2, -2)
cov_matrix <- matrix(c(0.5, 0, 0, 0.5), nrow = 2)
n_samples <- 400

#Generate the data using Data_Generator.r-------/
sep_data <- Binary_MVN(mean1, mean2, cov_matrix, n_samples)

#Plot-----------------------------------------/
ggplot(data = sep_data, aes(x = x,y = y,color = labelpl))+
geom_point()+
labs(title = "Datasets Alice and Bob",x = "X-axis",y = "Y-axis") +
theme_minimal()
```



We can very easily apply logistic regression using the `glm` feature in `R`:

```r
#Run LR---------------------------------------/
logistic_model <- glm(label ~ x + y,
data = sep_data, family = "binomial")
#Retrive boundary coeff-----------------------/
coefficients <- coef(logistic_model)
```

```r
#Plot----------------------------------------/
ggplot(data = sep_data, aes(x = x,y = y,color = labelpl))+
geom_point()+
geom_abline(intercept = -coefficients[1]/coefficients[3],slope = -coefficients[2]/coefficients[3])
```



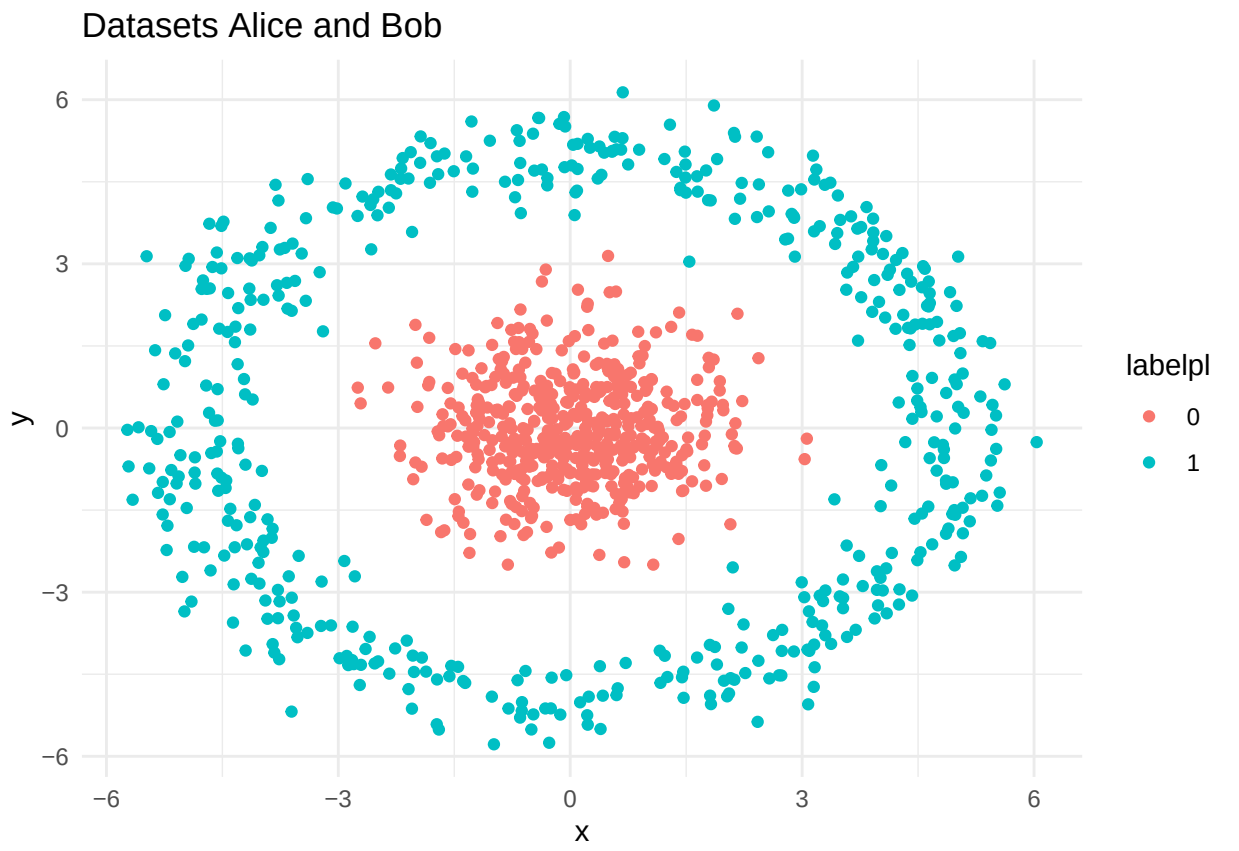

```r
labs(title = "Datasets Alice and Bob: Classified") +
theme_minimal()
```

```
## NULL
```

Toy Example: Quadratic Feature Transform

Now we try a more complicated example, which employs a feature transform. Consider the following dataset:

```r
#|Generate Data with Data_Generator.r--------------|
Ring_Data <- Binary_RING(500)

#|Plot Data----------------------------------------|
pl <- ggplot(Ring_Data, aes(x = x, y = y, color = labelpl)) +
geom_point() +
labs(title = "Datasets Alice and Bob") +
theme_minimal()

print(pl)
```



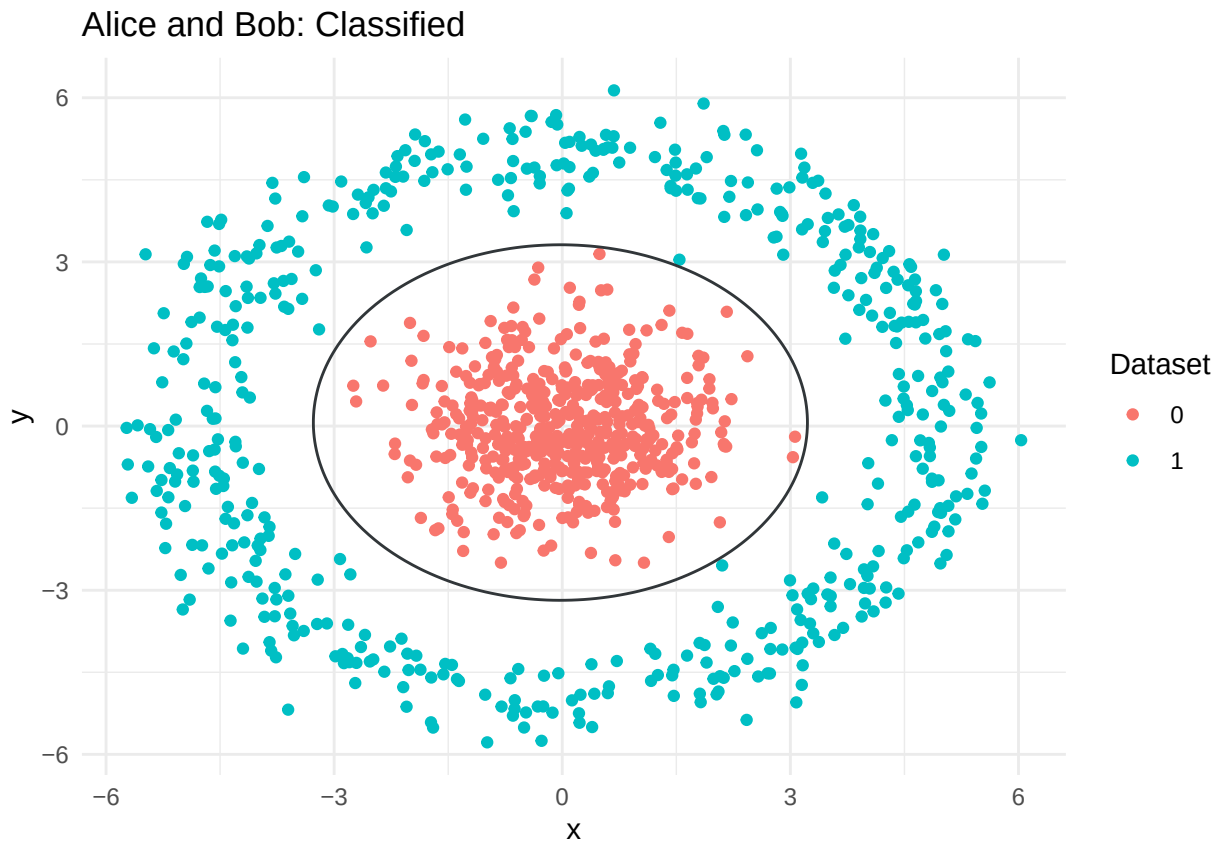
Clearly this data is not linearly seperable, so before anything else we employ the following feature transform:

$$\phi((x_1, x_2)) = (x_1, x_2, x_1^2 + x_2^2),$$

i.e we extend the data into three dimensions - onto a cone specifically. After constructing the feature transform manuaully (it's hard to automate as it requres choice) we run the `glm` command, which stands for *Generalised Linear Model*.

```
#|Feature Transform------------------|
Cone_Data <- Quadratic_Transform(Ring_Data)

#Logistic Regression-----------------|
logistic_model <- glm(label ~ x + y + xy,
data = Cone_Data, family = "binomial")

#Obtain Equation --------------------|
coefficients <- coef(logistic_model)

logistic_function <- function(x, y) {
  return(coefficients[1]+
           coefficients[2]*x+
           coefficients[3]*y+
           coefficients[4]*(x^2+y^2))
}

#Generate contour--------------------|
```

```r
x_values <- seq(-4,4,length.out=100)
y_values <- seq(-4,4,length.out= 100)
grid <- expand.grid(x = x_values,y= y_values)
#Generate (x,y) set
grid$z <- logistic_function(grid$x,grid$y)
#Generate (x,y,z) set

#Plot with contour--------------------/
pl2 <- ggplot(Cone_Data,
  aes(x = x, y = y, color = labelpl)) +
  geom_point() +
  geom_contour(data = grid, aes(x = x, y = y, z = z),
  breaks = 0.5, color = "#313639", linetype = "solid") +
  labs(title = "Alice and Bob: Classified",
       color = "Dataset") +
  theme_minimal()

print(pl2)
```



The function itself returns quite a lot of information, including predictions and residuals. Unfortunately it doesn't return the objective function (although we have that analytically) or the minima that the objective function achieves (although we can derive it via plugging in w to our objective function).

Implimenting the Objective Function

Below we impliment the objective function in `R` as a function of possible parameters $\mathbf{w}$ and w_0 We will use this function when we perform manual optimisation for logistic regression. Note that we include a seperate function to process the data, as it may include a feature transform.

```
Negative_Log_Likelihood_LINEAR <- function(w){
  w_0 <- w[1]
  w_g <- w[-1]
  x_vals <- sep_data[,1:2]
  y_vals <- sep_data$label
  dot_prod <- w_g*t(x_vals)
  linear_comb <- apply(dot_prod,2,sum) + w_0
  for (i in 1:length(y_vals)){
    if (y_vals[i]==0){
      linear_comb[i] = -linear_comb[i]
    }
  }
  log_sigmoid_vals <- log(1/(1+exp(linear_comb)))
  neg_likelihood <- -sum(log_sigmoid_vals)
  return(neg_likelihood)
}
```

Optimising the Objective Function

Gradient Descent

Gradient descent is a pretty slow but fundamental optimisation algorithm, as it forms the basis for complex optimisation like Newton methods and stochastic gradient descent. It builds on the principle that the for some multivariable functional $f : \mathbb{R}^d \to \mathbb{R}$ the gradident ∇f points in the direct of steepest ascent. So recursively taking steps in the *other* direction should converge to a minima.

We formally construct our algorithm as follows:

Suppose we have a continuously differentiable $f : \mathbb{R}^d \to \mathbb{R}$, fix a learning rate $\epsilon > 0$ and and inital point x_0, then we define the following sequence:

$$x_{n+1} = x_n - \epsilon \nabla f$$

Continue this iteration until $|x_n - x_{n-1}| < \epsilon_0$ for some sufficiently small ϵ_0. One thing to note is that we need to have ∇f, for our context we will write out the derivative analytically, however one can utilize the `numDeriv` package or built in `deriv` commands in practice, most of which employs *automatic differentiation* to analytically compute the derivative of a functional.

Lets impliment this in `R` code using the Rosenbrock function as a toy example, which we know has a global minima at $(1, 1)$.

```
#Rosenbrock Function--------------------/
objective_function <- function(x){
  return((1-x[1])^2 + 100*(x[2]-(x[1])^2)^2)}

true_minima <- c(1, 1)

#Gradient Function----------------------/
grad_obj <- function(x){
```

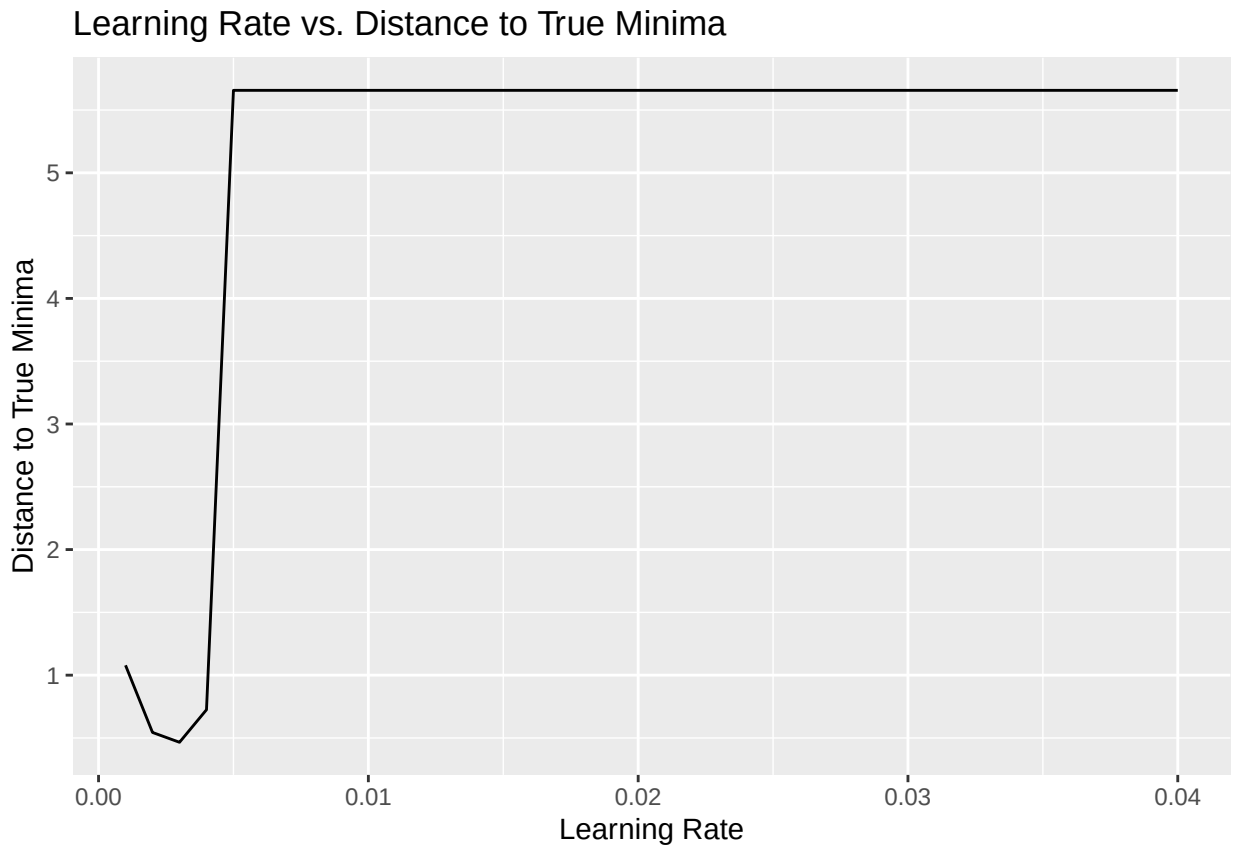
```r
  return(c(400*x[1]^3 - 400*x[1]*x[2]+2*x[1]-2,200*(x[2]-x[1]^2)))}

#Variables for GD-----------------------/
x_0 <- c(-1,1) #Initial point
learnrt <- 0.003 #Learning rate
max_iterations <- 1000 #Upper bound on iterations

#Gradient Descent-----------------------/
gradient_descent(objective_function,grad_obj,x_0,learnrt,max_iterations)
```

```
## [1] 0.771695 0.594419
```

We see that even after 1000 iterations, our primitive gradient descent algorithm is nowhere near the true minima, of course this is only one choice of ϵ, so lets vary it over a range and see how close we get to the true minima.

Note that when gradient descent diverges we assign it the value $(5, 5)$ for presentation purposes. So it is fixed at a distance of $4\sqrt{2}$ from the minima.

```r
#|Range for Learning rates--------------/
learning_rates <- seq(0.001, 0.04, by = 0.001)

#Compute convergence--------------------/
results <- data.frame(learning_rate = numeric(), distance_to_true_minima = numeric())
# Create container
for (lr in learning_rates) {
  x_final <- gradient_descent(objective_function, grad_obj, x_0, lr, max_iterations)
  #GD on each learning rate
  distance_to_true_minima <- sqrt(sum((x_final - true_minima)^2))
  results <- rbind(results, data.frame(learning_rate = lr, distance_to_true_minima = distance_to_true_mi
  #Add distance to convergence to results
}

# Plot Results---------------------------/
ggplot(results, aes(x = learning_rate, y = distance_to_true_minima)) +
  geom_line() +
  labs(title = "Learning Rate vs. Distance to True Minima", x = "Learning Rate", y = "Distance to True M
```



We see that it gets closest to the minima at around $\epsilon = 0.03$, but never actually achieves it. If we try running the same code with $10,000$ iterations we do reach the minima, but only with a very specific ϵ value. It's easy to see that this hyperparameter ϵ can cause a lot of issues in optimisation, this is why most algorithms which build on gradient descent optimise the hyperparameter while running or remove it entirely for something with more accuracy.

I hope the reader can appreciate that any attempt to use just gradient descent on our `Negative_Log_Likelihood` function would be pointless, as it can't even reliably find the root of our toy-example. Instead we move onto newton methods.

Advanced Methods

We will be applying the following methods:

- BFGS
- Nelder Mead
- Conjugate Gradient
- Simulated Anealing

BFGS

BFGS is a Newton-type method which relies on continuously approximating the hessian matrix, in principle as we converge to the minima, the better the approximation of the hessian we have. After we have an

approximation of the hessian H_n, we find a direction to travel in by solving a system of linear equations. More steps are added to find the best step size - we saw in gradient descent this was a big problem. After taking the appropriate sized step in the appropriate direction, we repeat. BFGS is relatively efficient compared to the following.

Nelder Mead

Nelder Mead is an extremely aesthetically pleasing algorithm, which relies on transforming a simplex across the surface generated by our function. Unfortunately this method is extremely slow to converge and only has advantages in non-differentiable optimisation.

Conjugate Gradient

Conjugate gradient methods are actually designed for linear algebra problems, but as we discovered solving systems of linear equations is quite important in Newton-type methods.

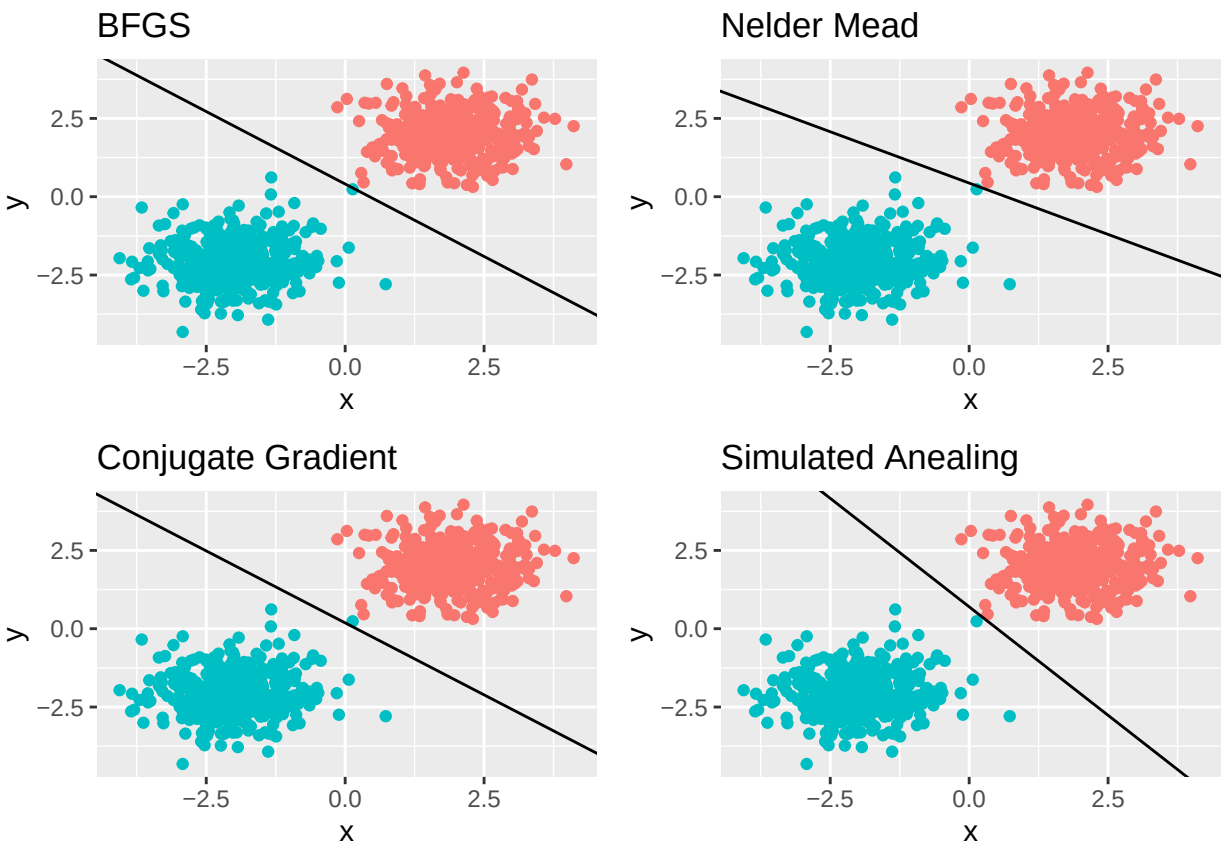
Simulated Anealing

Simulated Anealing is probably the strangest (and in our context longest running) optimisation technique used by `optim`. It uses random sampling to essentially guess solutions, and if it finds a particularly large value, it will search close neighbours of that point for even larger points. The magical thing about simulated annealing is that it can find global minima, whereas all others cannot guarantee this.

Optimising for Logistic Regression

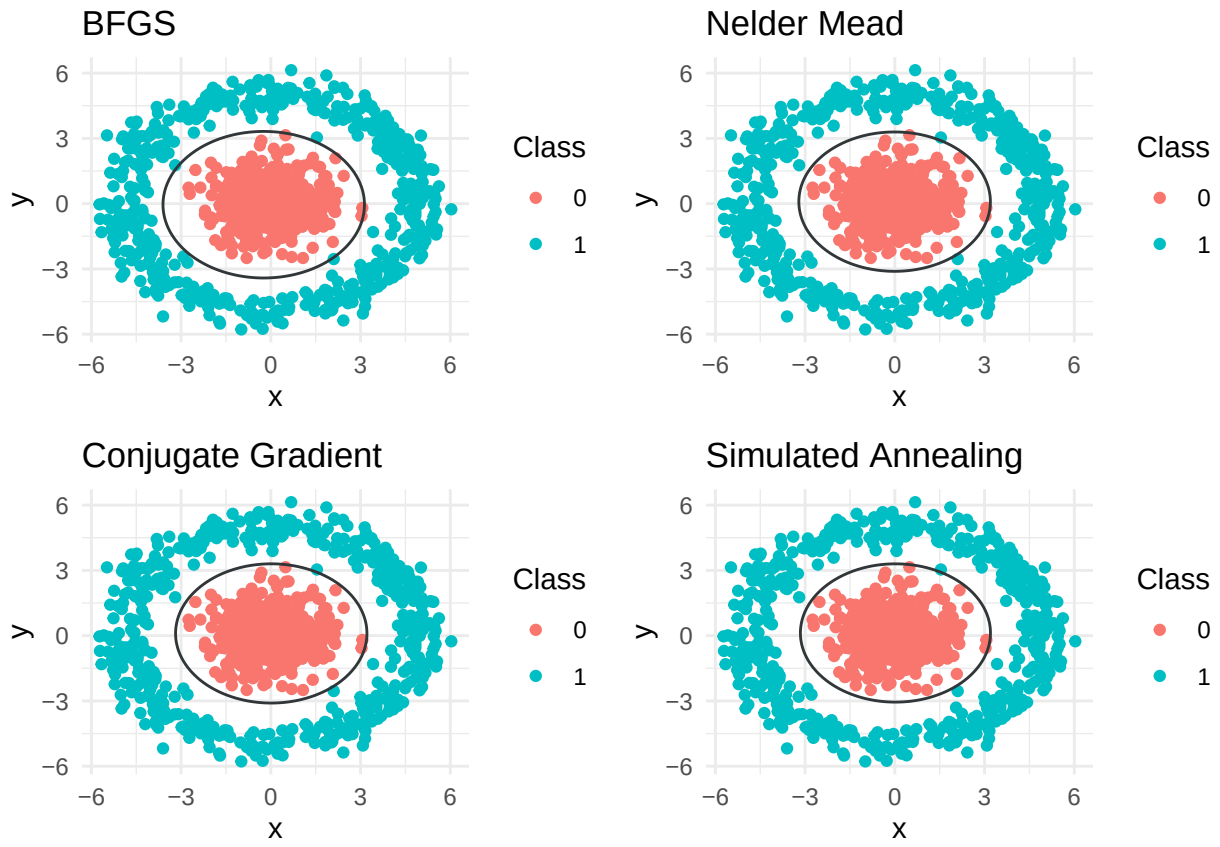
Toy Example: Linearly Seperable

```
## TableGrob (2 x 2) "arrange": 4 grobs
##   z     cells    name           grob
## 1 1 (1-1,1-1) arrange gtable[layout]
## 2 2 (1-1,2-2) arrange gtable[layout]
## 3 3 (2-2,1-1) arrange gtable[layout]
## 4 4 (2-2,2-2) arrange gtable[layout]
```



There is clearly a standout in that simulated annealing is the closest to our result from running `glm` earlier on, however there is still significant deviation. We also see that the other algorithms found the same local minima, but SANN didn't, likely a result of it looking for a global minima.

Toy Example: Quadratic Feature Transform




```
## TableGrob (2 x 2) "arrange": 4 grobs
##   z     cells    name           grob
## 1 1 (1-1,1-1) arrange gtable[layout]
## 2 2 (1-1,2-2) arrange gtable[layout]
## 3 3 (2-2,1-1) arrange gtable[layout]
## 4 4 (2-2,2-2) arrange gtable[layout]
```

Surprisingly BFGS actually got the closest out of all of them to the `glm` estimate for logistic regression in this case. It is also worth noting that simulated annealing, as before, took significantly longer than any of the other optimisation algorithms.