Non-linear Classification

Jin Seo Jo

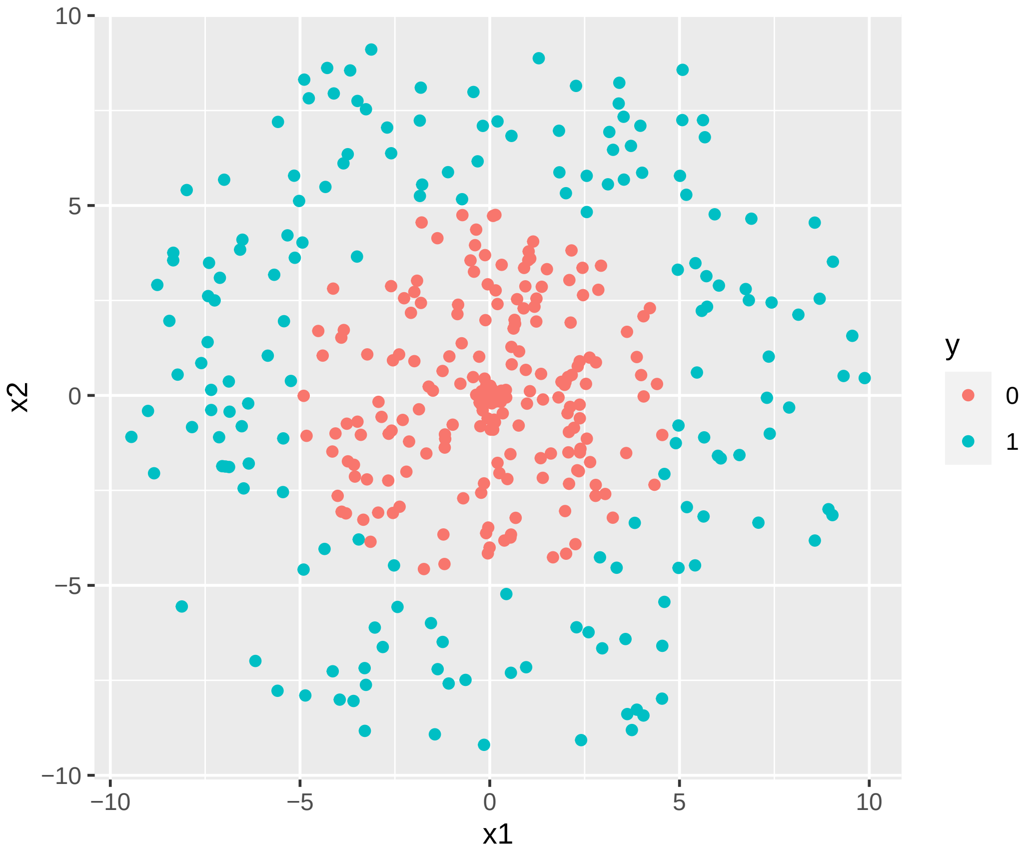
We will begin by loading two packages :

```
library(tidymodels)
library(tidyverse)
```

Non-linear classification

Consider the following data set.

```
set.seed(24601)

n <- 500
dat <- tibble(r = runif(n, max = 10), z1 = rnorm(n), z2 = rnorm(n),
              x1 = r * z1 / sqrt(z1^2 + z2^2), x2 = r * z2 / sqrt(z1^2 + z2^2),
              y = if_else(r < 5, 0, 1)) %>% mutate(y = factor(y)) %>%
  select(x1, x2, y)

split <- initial_split(dat)
train <- training(split)
test <- testing(split)

train %>% ggplot(aes(x1,x2, colour = y)) + geom_point() + coord_fixed()
```



The code looks a bit weird, but it helps to know that points (x_1, x_2) that are simulated according to

- $z_1, z_2 \sim N(0, 1)$
- $x_j = z_j/(z_1^2 + z_2^2)^{1/2}$

are uniformly distributed on the unit circle. (If we add more z_j it will be on the unit sphere!) This means that I've basically chosen a random radius uniformly on $[0, 10]$ and sampled from a point on the circle with radius. If the radius is less than 5 the point is a zero, otherwise it's a one.

This is a nice 2D data set where linear classification will not do a very good job.

Use tidymodels' `logistic_reg()` model with the `"glm"` engine to fit a linear classifier to the training data using `x1` and `x2` as features:

```
spec_linear <- logistic_reg() %>% set_engine("glm")

wf_linear <- workflow() %>% add_formula(y ~ x1 + x2) %>% add_model(spec_linear)
fit_linear <- wf_linear %>% fit(data = train)
fit_linear
```

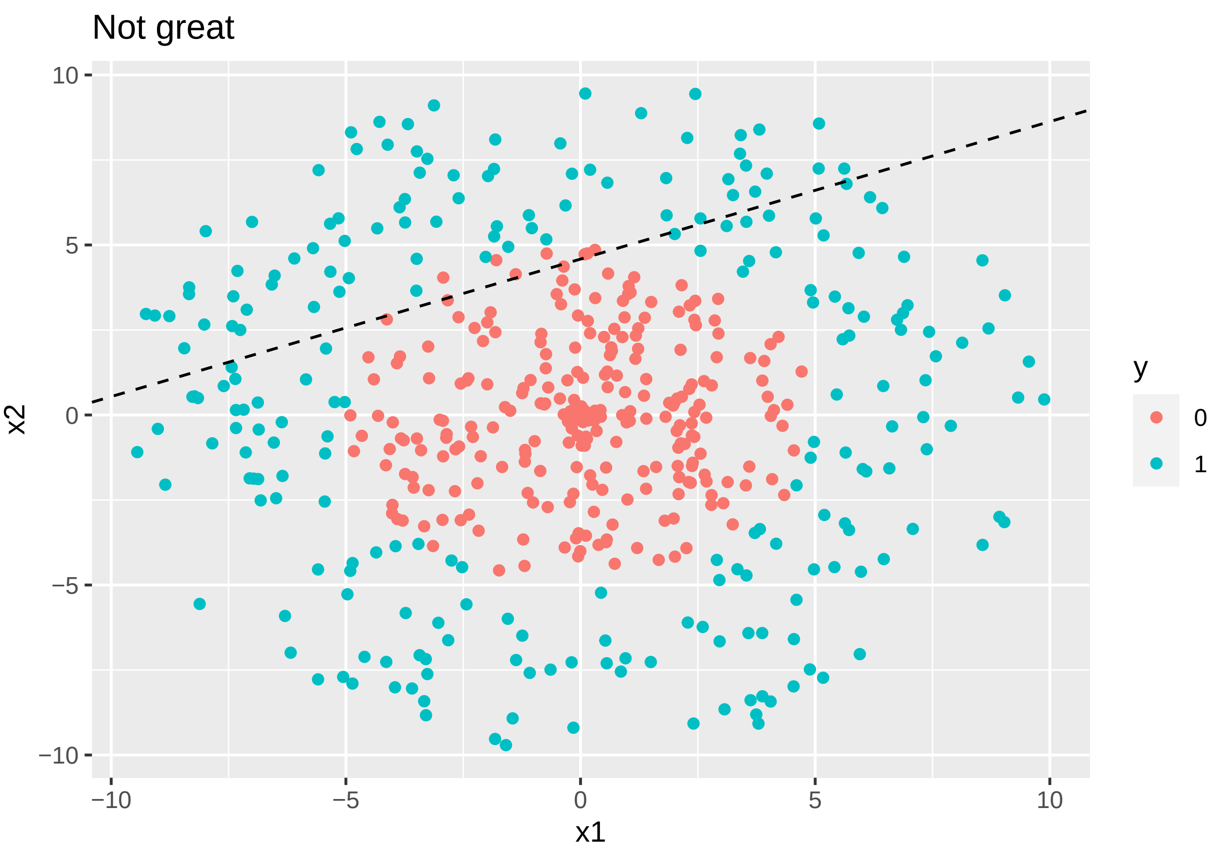
```
## == Workflow [trained] ==========================================================
## Preprocessor: Formula
## Model: logistic_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## y ~ x1 + x2
##
```

```
## -- Model -----------------------------------------------------------------------
##
## Call:  stats::glm(formula = ..y ~ ., family = stats::binomial, data = data)
##
## Coefficients:
## (Intercept)           x1           x2
##    -0.19320     -0.01704      0.04214
##
## Degrees of Freedom: 374 Total (i.e. Null);  372 Residual
## Null Deviance:       517
## Residual Deviance: 514   AIC: 520
```

The classification boundary is $\beta_0 + \beta_1 x_1 + \beta_2 x_2 = 0$ or

$$x_2 = -\frac{\beta_0}{\beta_2} - \frac{\beta_1}{\beta_2} x_1$$

```
coefs <- fit_linear$fit$fit$fit$coefficients
intercept <- -coefs[1] / coefs[3]
slope <- -coefs[2] / coefs[3]
dat %>% ggplot(aes(x1, x2, colour = y))  + geom_point() +
geom_abline(intercept = intercept, slope = slope, linetype =  2) +
ggtitle("Not great")
```



One way to assess how well a classifier did on a binary problem is to use the *confusion matrix*, which cross tabulates the true 0/1 and the predicted 0/1 values on the test set.

This often gives good insight into how classifiers differ. It can be computed using the `yardstick` package (which is part of tidymodels):

```r
conf_mat(data, ## A data.frame with y and .pred
         truth, ## y
         estimate ## .pred (from predict)
         )
```

Compute the confusion matrix for the linear classifier using the `test` data:

```r
cm_linear <- fit_linear %>% predict(test) %>% bind_cols(test) %>%
  conf_mat(truth = y, estimate = .pred_class)

cm_linear
```

```
##           Truth
## Prediction  0  1
##          0 57 48
##          1  2 18
```

Now let's try to fit a non-linear classifier to the data. The first thing we can try is k-Nearest Neighbours using the tidymodels framework.

Fit a k-nearest neighbours classifier to the training data, using cross validation to choose k. Compute the confusion matrix on the test data:

```r
library(kknn)

spec_knn <- nearest_neighbor(mode = "classification", neighbors = tune()) %>%
  set_engine("kknn")

wf_knn <- workflow() %>% add_formula(y ~ x1 + x2) %>% add_model(spec_knn)
folds <- vfold_cv(train)

grid <- grid_regular( neighbors(range = c(1, 40)), levels =  20)
tune_knn <- wf_knn %>% tune_grid(resamples = folds, grid = grid)
best <- select_best(tune_knn, metric = "roc_auc" )

wf_knn <- finalize_workflow(wf_knn, best)
wf_knn
```

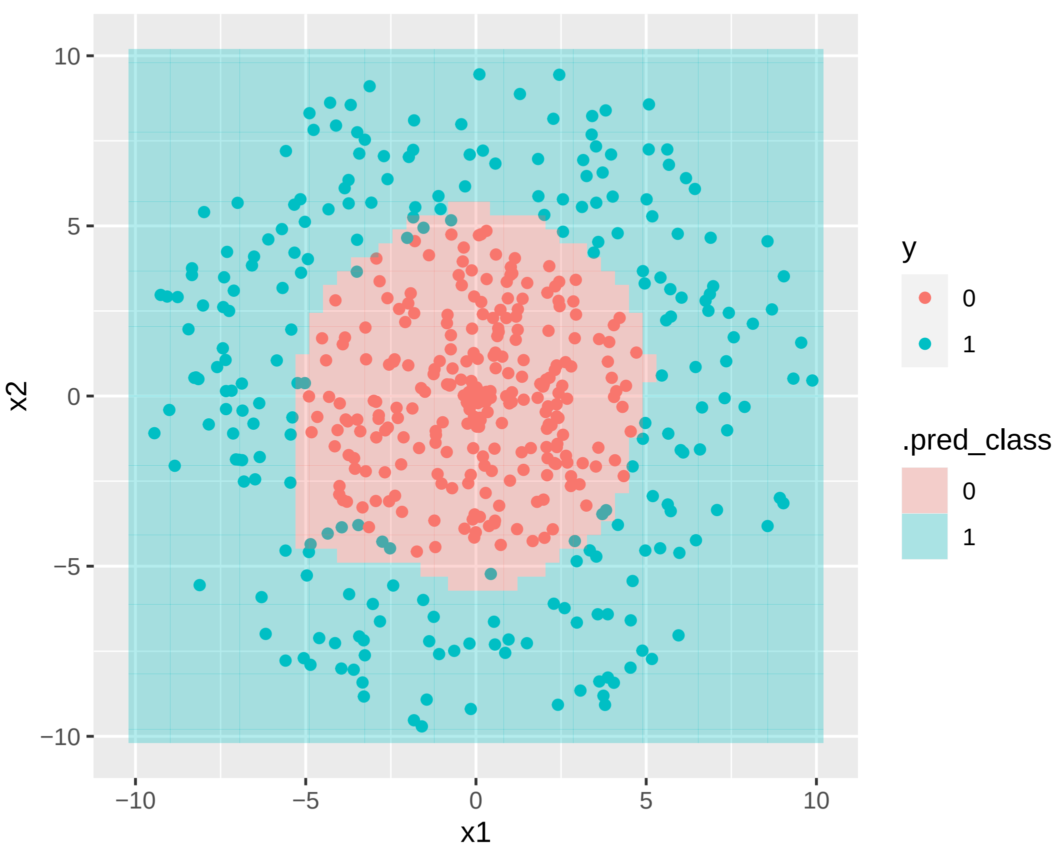
```
## == Workflow ====================================================================
## Preprocessor: Formula
## Model: nearest_neighbor()
##
## -- Preprocessor ----------------------------------------------------------------
## y ~ x1 + x2
##
## -- Model -----------------------------------------------------------------------
## K-Nearest Neighbor Model Specification (classification)
##
## Main Arguments:
##   neighbors = 25
##
## Computational engine: kknn
```

```
fit_knn <- wf_knn %>% fit(train)

cm_knn <- fit_knn %>% predict(test) %>% bind_cols(test) %>%
  conf_mat(truth = y, estimate = .pred_class)
cm_knn
```

```
##           Truth
## Prediction  0  1
##          0 59  7
##          1  0 59
```

It's a bit more tricky to plot the decision boundary this time. The easiest option is to predict at a bunch of points:

```
plot_points <- expand_grid(x1 = seq(-10, 10, length.out = 50),
                           x2 = seq(-10, 10, length.out = 50))

pred_plot <- predict(fit_knn, plot_points) %>%
  bind_cols(plot_points)

ggplot() +
  geom_point(data = dat, aes(x1, x2, colour = y)) +
  geom_tile(data = pred_plot, aes(x1,x2, fill = .pred_class), alpha = 0.3) +
  coord_fixed()
```



Finally, let's try a Support Vector Machine. SVMs are a bit like logistic ridge regression except instead of using the logistic loss function, the SVM uses the *hinge loss*.

$$\min \sum_{i=1}^{n} ((2y_i - 1)(\beta_0 + x_i^T \beta), 0)_+ + \lambda \|\beta\|_2^2.$$

There are two advantages to this loss function:

1. It is not sensitive to separation (it actually ignores points that are deep inside the correct region)
2. It can be *kernelized* to allow for non-linear classification.

Usually, when someone talks about an SVM they are talking about the kernelized non-linear variant. For this, we use the Gaussian kernel function

$$k(x, y) = \exp\left(-\frac{1}{2\sigma^2}\|x - y\|_2^2\right),$$

which is implemented in either the `kernlab` or `liquidSVM` packages (we need to have these packages loaded). We will use `kernlab`, but `liquidSVM` is fast for big problems.

The model specification is

```r
spec_svm <- svm_rbf(mode = "classification", rbf_sigma = tune()) %>%
  set_engine("kernlab")

spec_svm %>% translate()
```

```
## Radial Basis Function Support Vector Machine Specification (classification)
##
## Main Arguments:
##   rbf_sigma = tune()
##
## Computational engine: kernlab
##
## Model fit template:
## kernlab::ksvm(x = missing_arg(), data = missing_arg(), kernel = "rbfdot",
##     prob.model = TRUE, kpar = list(sigma = ~tune()))
```

(The `_rbf` stands for "radial basis function" which is another way of referring to the kernel function.)

Fit a kernelized SVM classifier to the training data and compute it's confusion matrix on the test data. Which of the three classifieres do we prefer?

```r
spec_svm <- svm_rbf(mode = "classification", rbf_sigma = tune()) %>%
  set_engine("kernlab")

wf_svm <- workflow() %>% add_formula(y ~ x1 + x2) %>% add_model(spec_svm)

grid <- grid_regular( rbf_sigma(), levels =  20)
tune_svm <- wf_svm %>% tune_grid(resamples = folds, grid = grid)
best <- select_best(tune_svm, metric = "roc_auc" )

wf_svm <- finalize_workflow(wf_svm, best)
wf_svm
```

```
## == Workflow ====================================================================
## Preprocessor: Formula
## Model: svm_rbf()
##
## -- Preprocessor ----------------------------------------------------------------
## y ~ x1 + x2
##
## -- Model -----------------------------------------------------------------------
## Radial Basis Function Support Vector Machine Specification (classification)
##
## Main Arguments:
##   rbf_sigma = 0.297635144163131
##
## Computational engine: kernlab
```

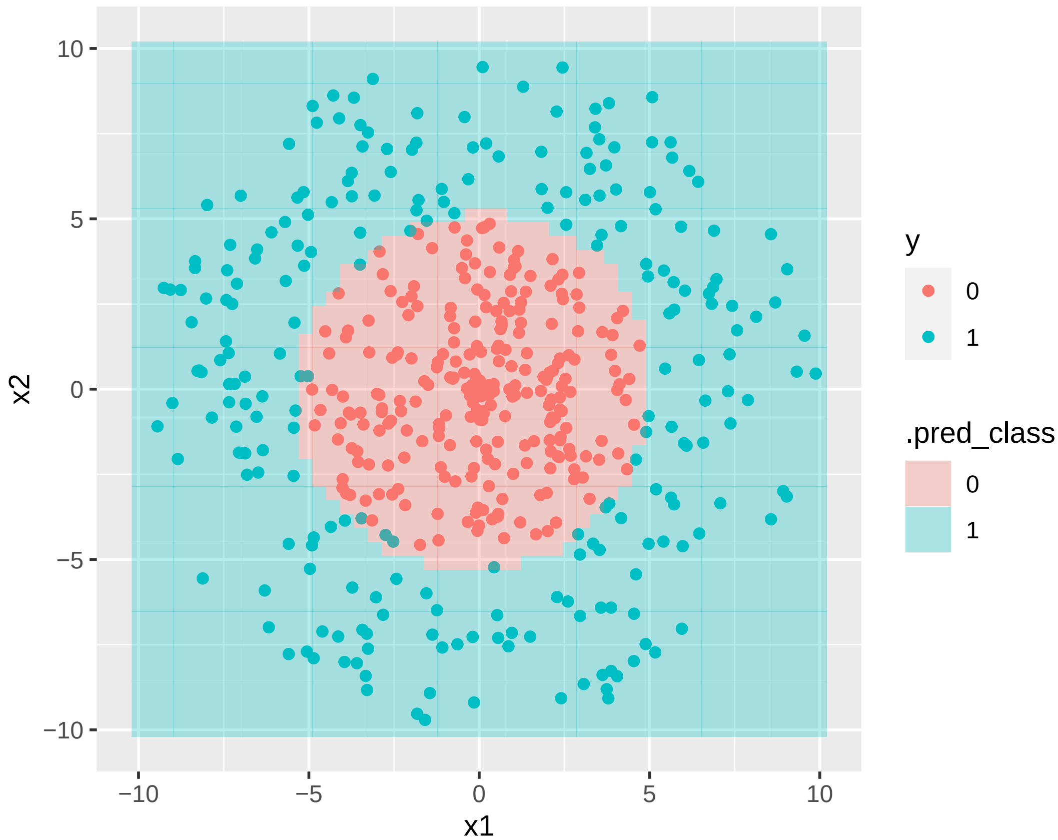
```
fit_svm <- wf_svm %>% fit(train)

cm_svm <- fit_svm %>% predict(test) %>% bind_cols(test) %>%
  conf_mat(truth = y, estimate = .pred_class)
cm_svm
```

```
##            Truth
## Prediction  0  1
##          0 59  4
##          1  0 62
```

Basically the same.

Let's look at the region.

```
plot_points <- expand_grid(x1 = seq(-10, 10, length.out = 50),
                           x2 = seq(-10, 10, length.out = 50))
pred_plot <- predict(fit_svm, plot_points) %>% bind_cols(plot_points)
 ggplot() + geom_point(data = dat, aes(x1, x2, colour = y)) + geom_tile(data = pred_plot, aes(x1,x2, fil
```



In this case, both the kNN and SVM did quite well. The SVM took longer to train, but choosing the penalty parameter is easier for the SVM than chosing the number of neighbours.