Machine Learning and Statistical Learning

Dealing with unbalanced dataset

AUTHOR
Ewen Gallic

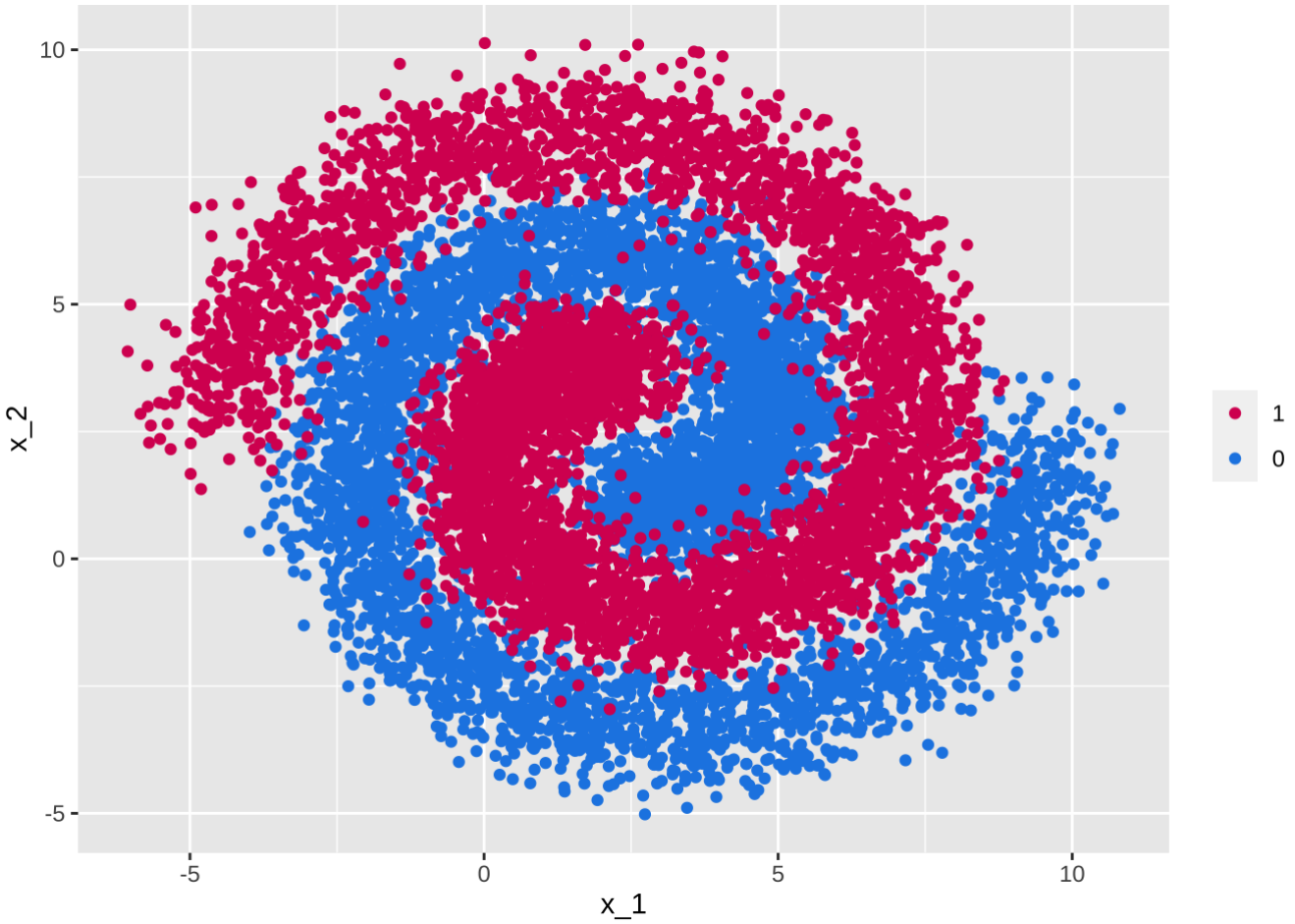
When faced with **unbalanced data** on which to build a classifier, the performance of the model to correctly predict each class is generally poor or even bad. In this notebook, we will consider the case of a **binary classifier**, trained to predict two classes that we will call the **majority class** and the **minority class**. Arbitrarily, we will consider that the minority class will be coded 1, while the majority class will be coded 0. In theory, when the repartition between the two classes is not equal (50% of 0 and 50% of 1), we speak of imbalance. In practice, problems occur when the imbalance is strong, for example 1% of 1 and 99% of 1. It is also possible to have extremely few observations of the minority class. The typical example encountered in machine learning is that of fraud, where more than 99% of observations are non-fraudulent while less than one percent of observations involve fraud.

To illustrate the different concepts, we will use data that we generate. Let us create a dataset with two spirals. To do so, we rely on some code published by Stanislas Morbieu on R-bloggers.

▶ Show the R codes

```
ggplot(data = df, aes(x = x_1, y = x_2)) +
  geom_point(mapping = aes(colour = y)) +
  scale_colour_manual(NULL, values = c("1" = "#D81B60", "0" = "#1E88E5"))
```

Two spirals



Now, let us only keep a few observation from class 1. For example, let us assume that there are 98% of observation of class 0 and only 2% of class 1.

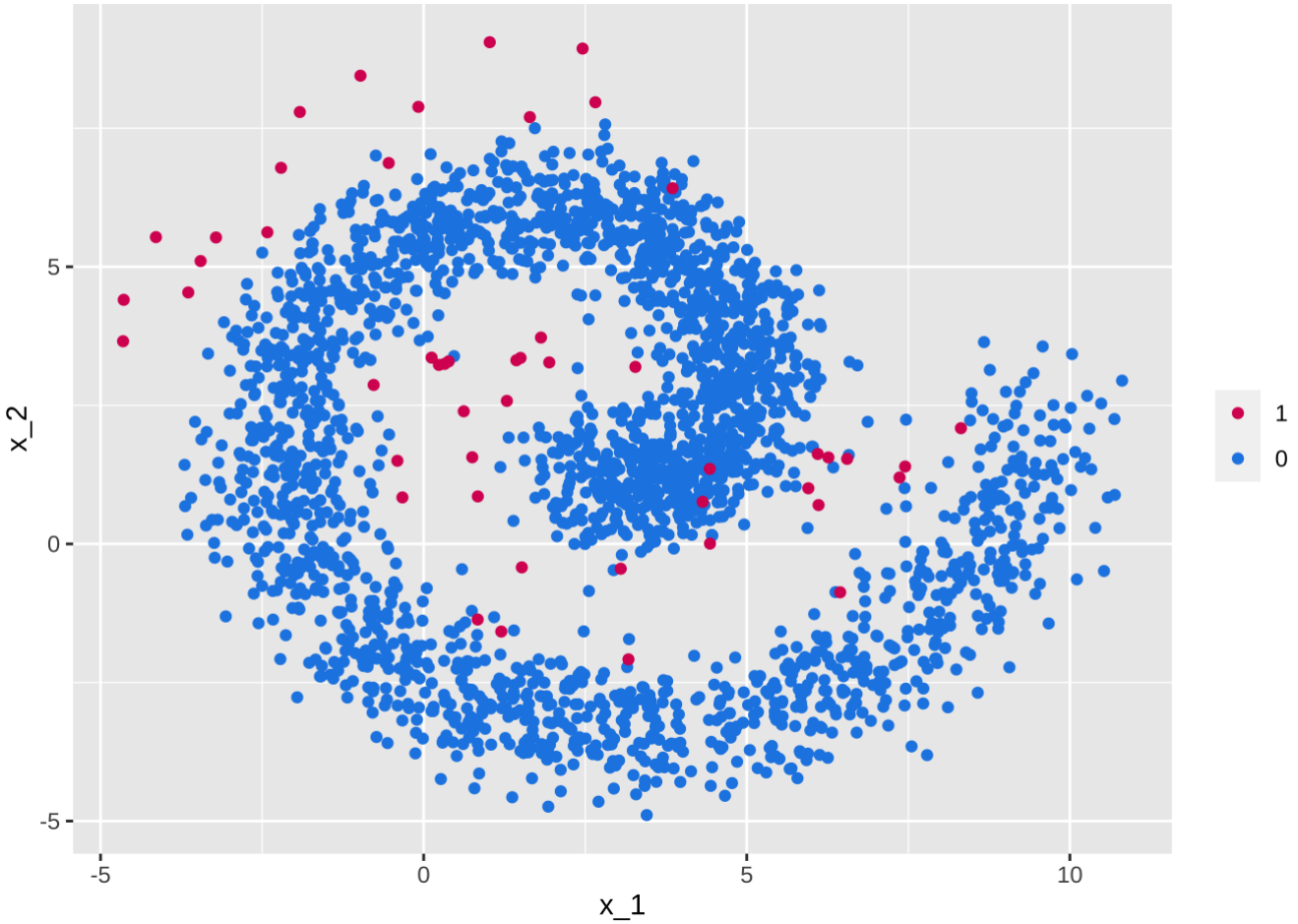
```
imbalance <- .98

df <-
  df %>% filter(y == "0") %>%
  sample_n(size = round(imbalance*n/2)) %>%
  bind_rows(
    df %>% filter(y == "1") %>%
      sample_n(size = round((1-imbalance)*n/2))
  )

df %>% group_by(y) %>% count() %>%
  ungroup() %>% mutate(prop = n / sum(n))
```

```
# A tibble: 2 × 3
  y         n  prop
  <fct> <int> <dbl>
1 0      2450  0.98
2 1        50  0.02
```

```
ggplot(data = df, aes(x = x_1, y = x_2)) +
  geom_point(mapping = aes(colour = y)) +
```

```
  scale_colour_manual(NULL, values = c("1" = "#D81B60", "0" = "#1E88E5"))
```

An imbalanced dataset with 2% of observation of the minority class



Instead of saying that there are 98% of observations from class 0 and 1% of observations from class 1, it is possible to use the **imbalance ratio** that gives the ratio between the number of observations from the majority class and the number of observations from the smallest minority class:

```
imbalance_ratio <- sum(df$y==0) / sum(df$y==1)
imbalance_ratio
```

```
[1] 49
```

The larger the imbalance ratio, the larger the imbalance. For a perfectly balanced dataset, the imbalance ratio is equal to 1.

Building a classifier on an imbalanced dataset

In a first step, let us try to fit a random forest on the raw data. We will train the model on a subsample of the data, and assess its quality of fit both on the same training data and on unseen data (test set). We will consider here two different random forests, and try to select the one that gives the best results.

Let us put 80% of the data in the training set and leave the remaining 20% for the test set.

```
ind_train <- sample(1:nrow(df), size = .8*nrow(df))
df_train <- df[ind_train, ]
df_test <- df[-ind_train, ]
```

A first model

Let us grow a random forest with a first set of hyperparameters (200 trees, 2 variables as candidates to draw from when performing the splits, 20 observation at minimum in terminal nodes)

```
library(randomForest)
mod <-
  randomForest(
  formula = y ~ x_1+x_2,
  data = df_train,
  ntree = 200,
  mtry =2 ,
  nodesize = 20,
  maxnodes = NULL
)
```

Thanks to the `predict` function, we can obtain the estimated probabilities to be classified as 1:

```
pred_prob_train <- predict(mod, newdata = df_train, "prob")
head(pred_prob_train)
```

```
     0    1
1 1.00 0.00
2 0.67 0.33
3 1.00 0.00
4 0.99 0.01
5 1.00 0.00
6 1.00 0.00
```

Based on these probabilities, we can assign a class (0 or 1) to each observation.

```
pred_class_train <- ifelse(pred_prob_train[,"1"] > .5, "1", "0") %>%
  factor(levels = c("0", "1"))
```

Let us have a look at the confusion table on the training set.

	Pred. Negative (0)	**Pred. Positive (1)**
Obs. Negative (0)	True Negative (TN)	False Positive (FP)
Obs. Positive (1)	False Negative (FN)	True Positive (TP)

```
conf_mat_train <-
  caret::confusionMatrix(data = pred_class_train,
                         reference = df_train$y,
                         positive="1")
```

```
conf_mat_train
```

```
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 1959   29
         1    0   12

               Accuracy : 0.9855
                 95% CI : (0.9792, 0.9903)
    No Information Rate : 0.9795
    P-Value [Acc > NIR] : 0.02988

                  Kappa : 0.4477

 Mcnemar's Test P-Value : 1.999e-07

            Sensitivity : 0.2927
            Specificity : 1.0000
         Pos Pred Value : 1.0000
         Neg Pred Value : 0.9854
             Prevalence : 0.0205
         Detection Rate : 0.0060
   Detection Prevalence : 0.0060
      Balanced Accuracy : 0.6463

       'Positive' Class : 1
```

While the accuracy (percentage of correctly predicted individuals) is very high (0.99), we need to keep in mind that 98% of the observations are from class 0. The true negative rate (or specificity) (TN/(TN+FP)) is very high (1.00), but the true positive rate (or sensitivity) (TP/(TP+FN)), on the other hand, is low (0.29).

Let us look at those same metrics based on the predictions made on unseen data.

```
pred_prob_test <- predict(mod, newdata = df_test, type = "prob")
pred_class_test <- ifelse(pred_prob_test[,"1"] > .5, "1", "0") %>%
  factor(levels = c("0", "1"))
```

```
conf_mat_test <-
  caret::confusionMatrix(data = pred_class_test, reference = df_test$y,
                         positive = "1")
conf_mat_test
```

```
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 491   8
         1   0   1
```

```
               Accuracy : 0.984
                 95% CI : (0.9687, 0.9931)
    No Information Rate : 0.982
    P-Value [Acc > NIR] : 0.45445

                  Kappa : 0.1971

 Mcnemar's Test P-Value : 0.01333

            Sensitivity : 0.1111
            Specificity : 1.0000
         Pos Pred Value : 1.0000
         Neg Pred Value : 0.9840
             Prevalence : 0.0180
         Detection Rate : 0.0020
   Detection Prevalence : 0.0020
      Balanced Accuracy : 0.5556

       'Positive' Class : 1
```

Again, the accuracy is very high (0.98). The specificity, *i.e.*, the true negative rate is high (1.00), but the sensitivity, *i.e.*, the true positive rate is low (0.11).

A second model

Let us fit a second model, with different specifications:

```
mod_2 <-
  randomForest(
  formula = y ~ x_1+x_2,
  data = df_train,
  ntree = 200,
  mtry =2 ,
  nodesize = 50,
  maxnodes = NULL
)
```

The predicted probabilities with this second model:

```
pred_prob_train_2 <- predict(mod_2, newdata = df_train, type = "prob")
```

Based on these probabilities, we can assign a class (0 or 1) to each observation.

```
pred_class_train_2 <- ifelse(pred_prob_train_2[,"1"] > .5, "1", "0") %>%
  factor(levels = c("0", "1"))
```

Let us have a look at the confusion table on the training set.

```
conf_mat_train_2 <-
  caret::confusionMatrix(data = pred_class_train_2,
                         reference = df_train$y,
```

```
                        positive="1")
conf_mat_train_2
```

```
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 1959   32
         1    0    9

               Accuracy : 0.984
                 95% CI : (0.9775, 0.989)
    No Information Rate : 0.9795
    P-Value [Acc > NIR] : 0.08616

                  Kappa : 0.3552

 Mcnemar's Test P-Value : 4.251e-08

            Sensitivity : 0.2195
            Specificity : 1.0000
         Pos Pred Value : 1.0000
         Neg Pred Value : 0.9839
             Prevalence : 0.0205
         Detection Rate : 0.0045
   Detection Prevalence : 0.0045
      Balanced Accuracy : 0.6098

       'Positive' Class : 1
```

Let us look at those same metrics based on the predictions made on unseen data.

```
pred_prob_test_2 <- predict(mod_2, newdata = df_test, type = "prob")
pred_class_test_2 <- ifelse(pred_prob_test_2[,"1"] > .5, "1", "0") %>%
  factor(levels = c("0", "1"))
```

```
conf_mat_test_2 <-
  caret::confusionMatrix(data = pred_class_test_2, reference = df_test$y,
                         positive = "1")
conf_mat_test_2
```

```
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 491   8
         1   0   1

               Accuracy : 0.984
                 95% CI : (0.9687, 0.9931)
    No Information Rate : 0.982
    P-Value [Acc > NIR] : 0.4544
```

```
      P-Value [Acc > NIR] : 0.45445

                   Kappa : 0.1971

 Mcnemar's Test P-Value : 0.01333

            Sensitivity : 0.1111
            Specificity : 1.0000
         Pos Pred Value : 1.0000
         Neg Pred Value : 0.9840
             Prevalence : 0.0180
         Detection Rate : 0.0020
   Detection Prevalence : 0.0020
      Balanced Accuracy : 0.5556

       'Positive' Class : 1
```

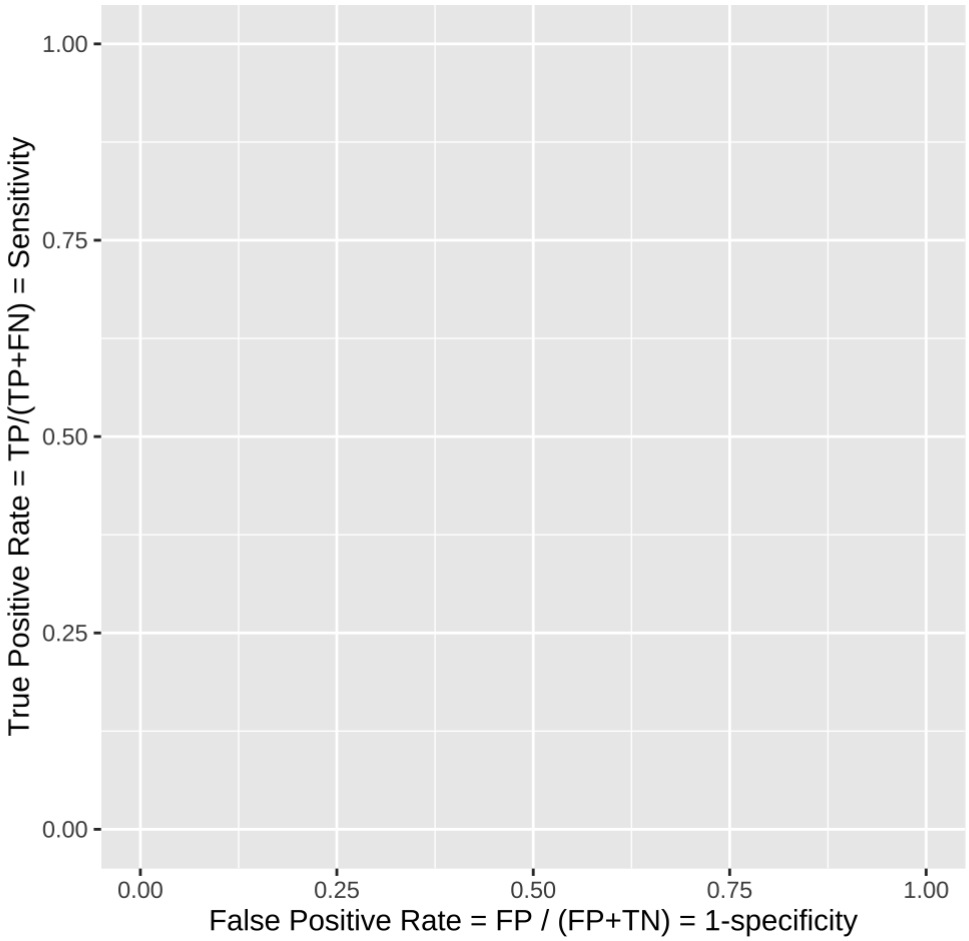
> **Warning**
>
> Which of the two models give the best results? To try to answer this question, let us consider the ROC curve.

ROC Curve

Let us have a look at the **ROC curve**. Recall that the x-axis shows the False Positive Rate (here, fractions of errors for the majority class) while the y-axis show the True Positive Rate (here, fraction of correct predictions for the minority class). The graph reports these two metrics when the threshold τ varies. This threshold is the cut-off point above which class 1 is predicted for the observation: if $\mathbb{P}(Y = 1 \mid X) \geq \tau$, then the observation is classified as 1, 0 otherwise. Varying this threshold will favour the TPR at the expense of the FPR or conversely.

The ROC curve is plotted on a graph with the False positive rate on the x-axis, and the True positive rate on the y-axis:

▶ Show the R codes



With classification threshold $\tau = 0$

If the threshold $\tau = 0$, then every observation is classified as 1. Let us assume that the fraction of class 1 in the sample is γ. The fraction of class 0 is then $1 - \gamma$.

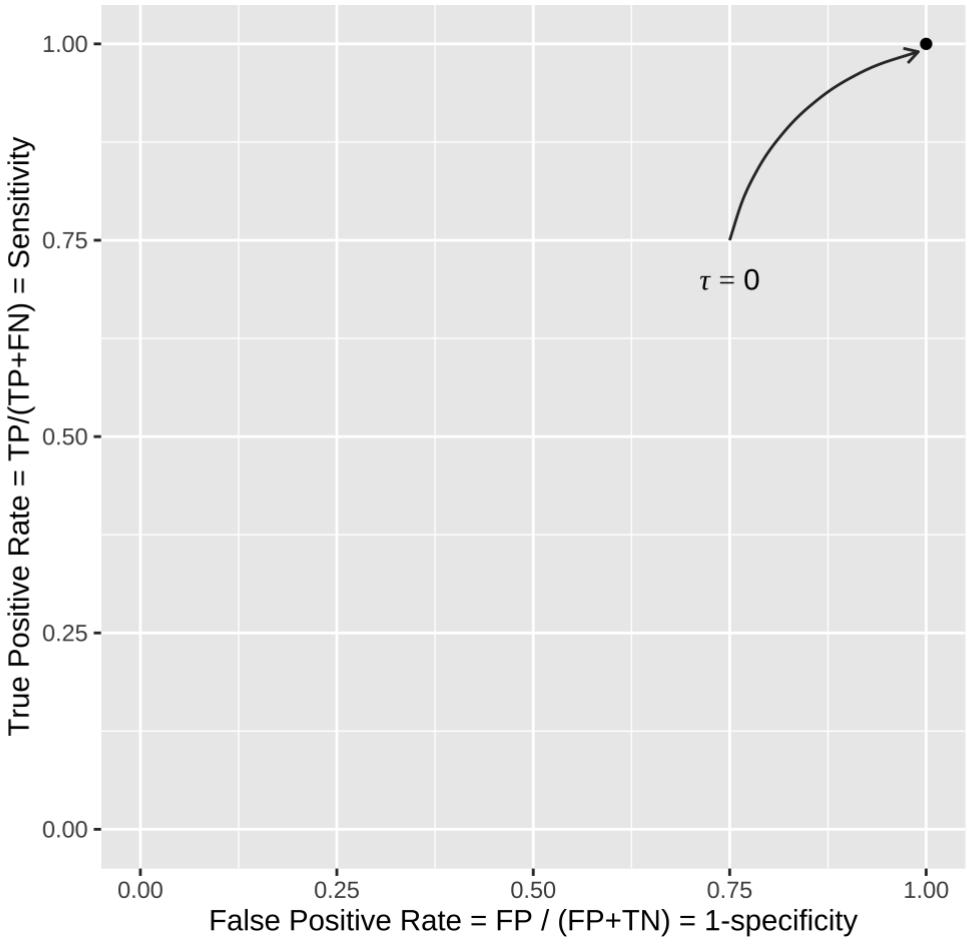
Confusion table (values expressed in proportions) for a classifier that systematically assigns class 1

	Pred. Negative (0)	**Pred. Positive (1)**
Obs. Negative (0)	True Negative (TN) = 0	False Positive (FP) = $1 - \gamma$
Obs. Positive (1)	False Negative (FN) = 0	True Positive (TP) = γ

In such a case:

- $TPR = \frac{TP}{TP+FN} = 1$;
- $FPR = \frac{FP}{FP+TN} = 1$.

▶ Show the R codes



With classification threshold $\tau = 1$

If the threshold $\tau = 1$, then every observation is classified as 0.

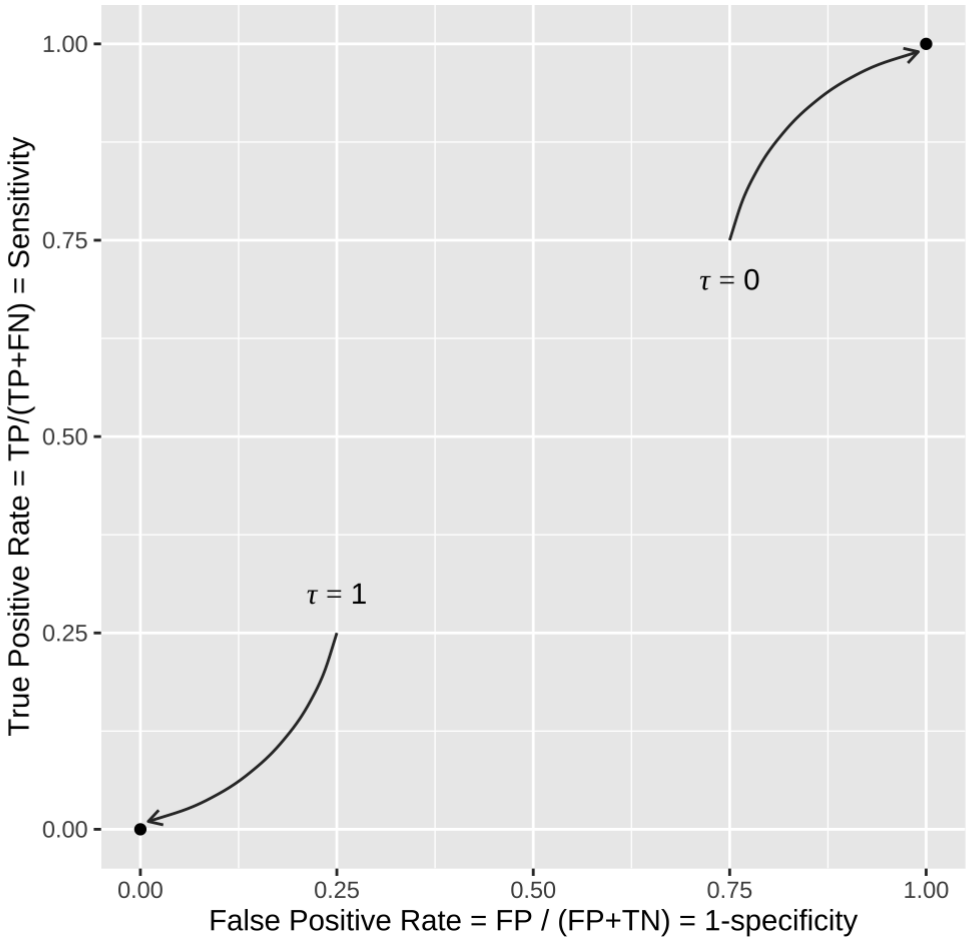
Confusion table (values expressed in proportions) for a classifier that systematically assigns class 0

	Pred. Negative (0)	**Pred. Positive (1)**
Obs. Negative (0)	True Negative (TN) = $1 - \gamma$	False Positive (FP) = 0
Obs. Positive (1)	False Negative (FN) = γ	True Positive (TP) = 0

In such a case:

- $TPR = \frac{TP}{TP+FN} = 0$;
- $FPR = \frac{FP}{FP+TN} = 0$.

▶ Show the R codes



With a perfect classifier

If the classifier perfectly predicts the class of the observations:

- If $\tau = 0$, then the classifier is no longer perfect, and we have:
  - $TPR = 1$,
  - $FPR = 1$;
- If $\tau = 1$, the classifier is not perfect either, and we have:
  - $TPR = 0$,
  - $FPR = 1$;
- For any other value of $0 < \tau < 1$:
- $TPR = \frac{\gamma}{\gamma+0} = 1$,
- $FPR = 0$.

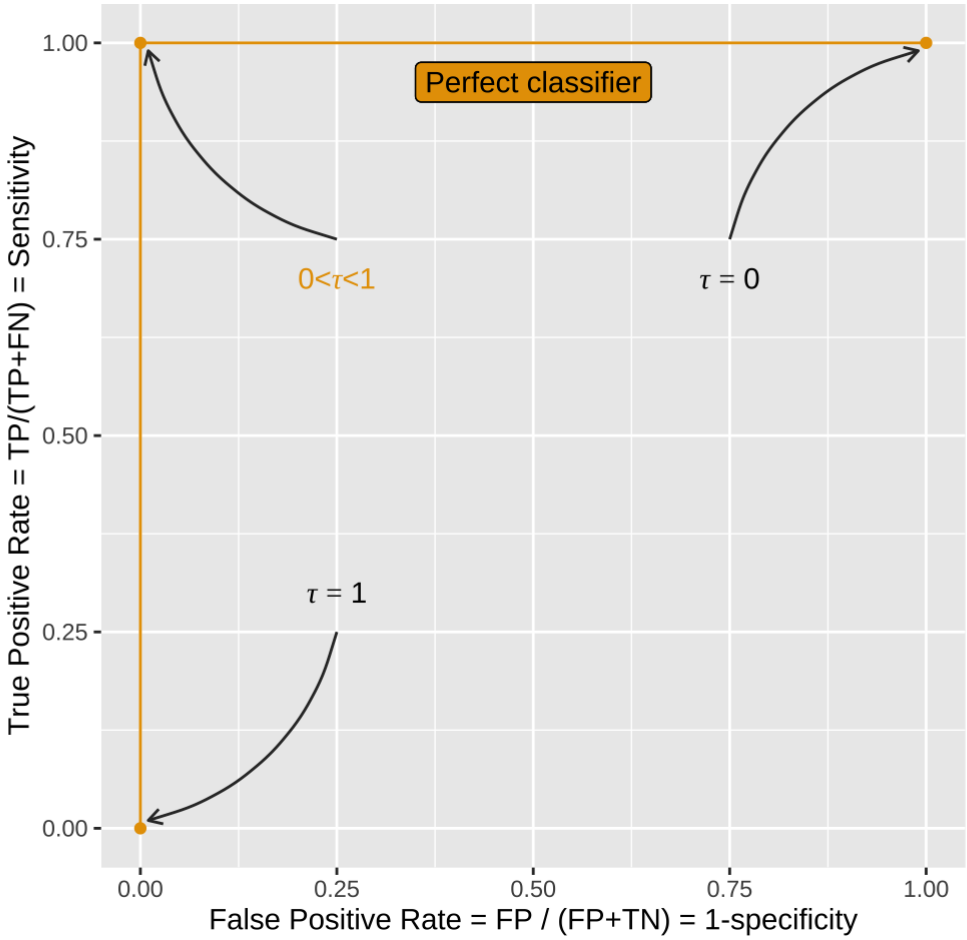
Confusion table (values expressed in proportions) for a classifier that perfectly predicts the classes, for $0 < \tau < 1$

	Pred. Negative (0)	**Pred. Positive (1)**
Obs. Negative (0)	True Negative (TN) = $(1 - \gamma)$	False Positive (FP) =0

	Pred. Negative (0)	Pred. Positive (1)
Obs. Positive (1)	False Negative (FN) = 0	True Positive (TP) = γ

Then, the ROC curve for a perfect classifier will be made of three points: (0,0), (0,1), and (1,1).

▶ Show the R codes



With a classifier that randomly assigns the classes

Now, let us consider the case where the classifier randomly assigns class 1 with a probability p and class 0 with probability $1-p$. In such a case, the expected proportions are as follows:

Confusion table (values expressed in proportions) for a classifier that randomly assigns class 1 with a probability p and class 0 with probability $1-p$

	Pred. Negative (0)	Pred. Positive (1)
Obs. Negative (0)	True Negative (TN) = $(1-p)(1-\gamma)$	False Positive (FP) =$p(1-\gamma)$
Obs. Positive (1)	False Negative (FN) = $(1-p)\gamma$	True Positive (TP) = $p\gamma$

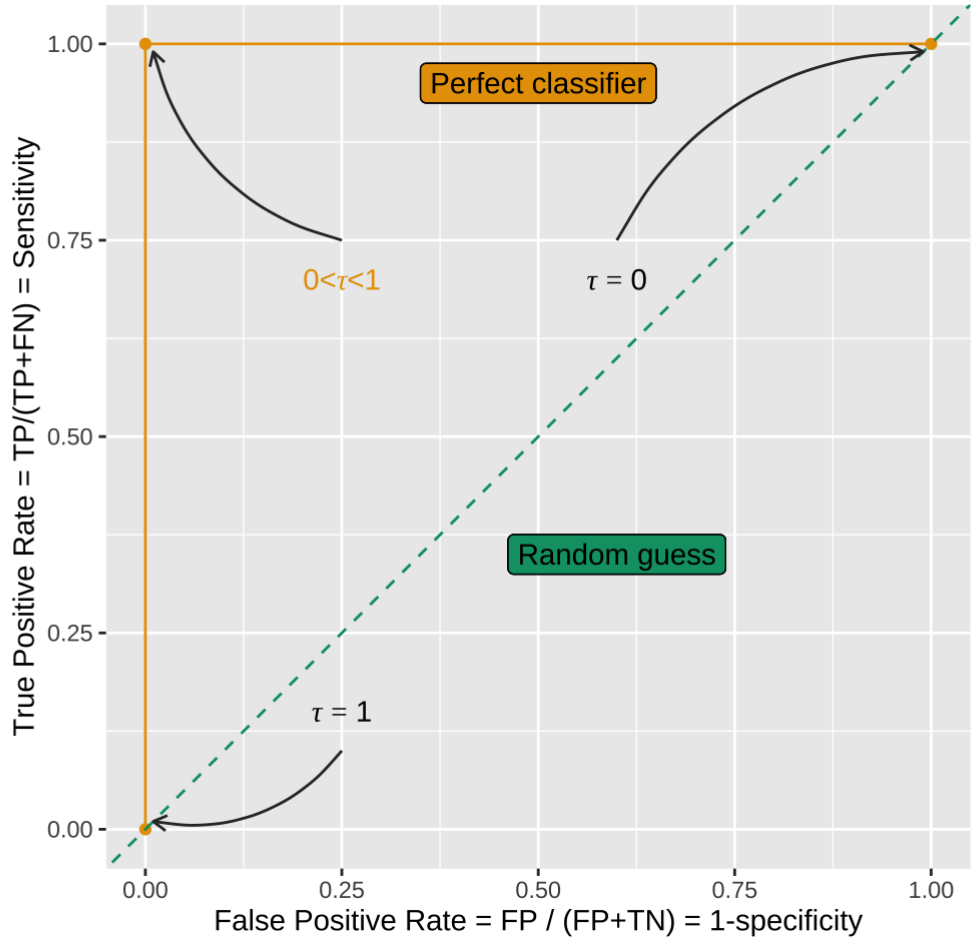
In that case the expected values for the TPR and the FPR are:

- $TPR = \frac{TP}{TP+FN} = \frac{p\gamma}{p\gamma+(1-p)\gamma} = p$;
- $FPR = \frac{FP}{FP+TN} = \frac{p(1-\gamma)}{p(1-\gamma)+(1-p)(1-\gamma)} = p$.

So, on average, with a classifier that randomly assigns a class to the observations, $TPR = FPR = p$. The ROC curve, for such a classifier will thus be a line from (0,0) to (1,1), *i.e.*, a diagonal. The classifier would have no discriminative power.

On the ROC curve, points above that diagonal will correspond to classifiers that provide results that are better than a random guess while points below will correspond to classifiers that provide results that are worse than a random guess.

▶ Show the R codes



The `roc_curve()` function from {yardstick} allows to get the TPR/sensitivity and the TNR/specificity (the FPR is equal to 1-TNR). Let us apply it to the train data firt.

```
library(yardstick)
tibble(observed = df_train$y, prob_class_1 = pred_prob_train[,"1"]) %>%
  roc_curve(truth = observed, prob_class_1, event_level="second")
```

```
# A tibble: 67 × 3
   .threshold specificity sensitivity
        <dbl>       <dbl>       <dbl>
 1   -Inf           0               1
 2      0           0               1
 3      0.005       0.873           1
```

```
 4      0.01        0.910           1
 5      0.015       0.923           1
 6      0.02        0.930           1
 7      0.025       0.938           1
 8      0.03        0.947           1
 9      0.035       0.960           1
10      0.04        0.967           1
# … with 57 more rows
```

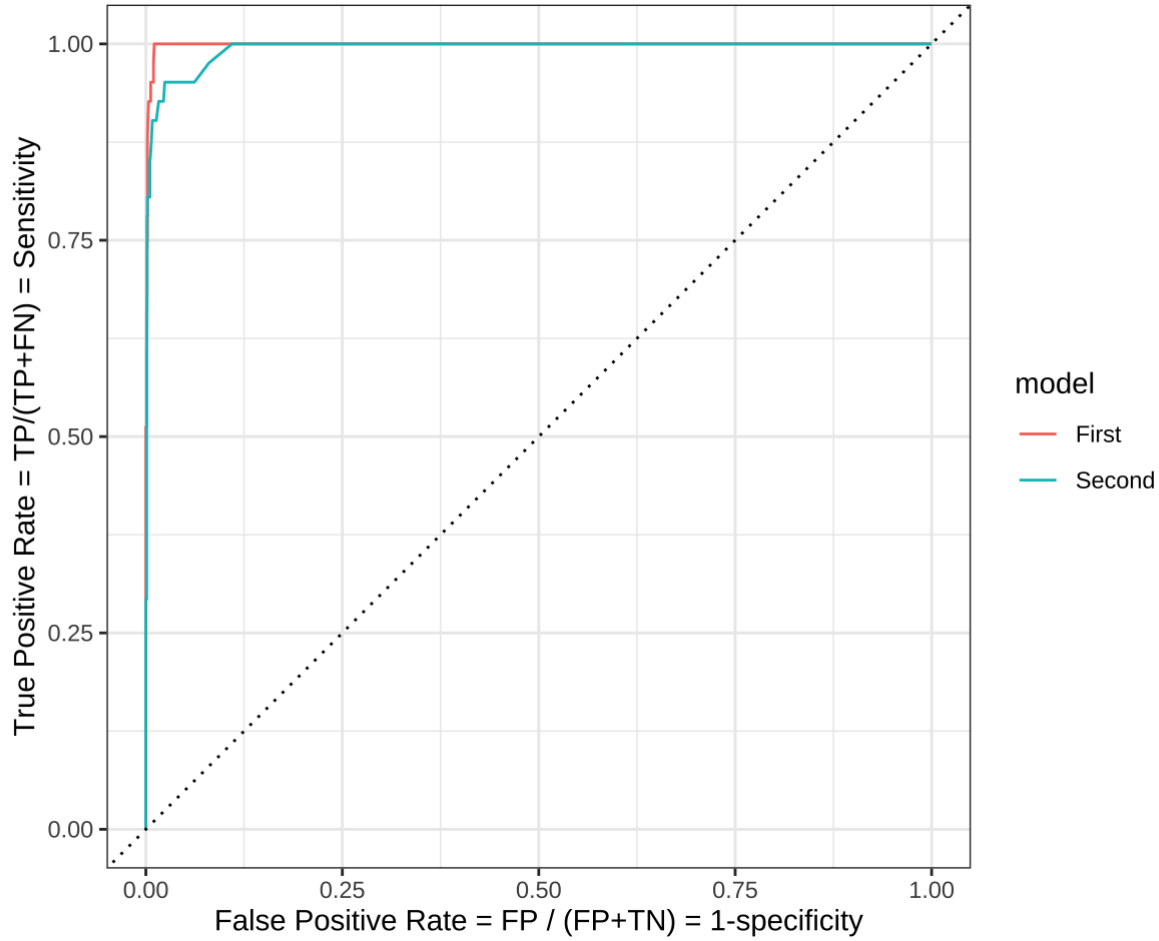
Let us put in a table the predicted probability for each observation, for both models.

```
pred_prob_train_both <-
  tibble(observed = df_train$y, prob_class_1 = pred_prob_train[,"1"],
       model = "First") %>%
  bind_rows(
    tibble(observed = df_train$y, prob_class_1 = pred_prob_train_2[,"1"],
           model = "Second")
  )
```

Then, the `autoplot()` from {yardstick} can be applied to the resulting table to plot the ROC curve.

```
 pred_prob_train_both %>%
  group_by(model) %>%
  roc_curve(truth = observed, prob_class_1, event_level="second") %>%
  autoplot() +
  labs(x = "False Positive Rate = FP / (FP+TN) = 1-specificity",
       y = "True Positive Rate = TP/(TP+FN) = Sensitivity")
```

ROC Curve for the first and the second models, training data



The Area Under the Curve (AUC) can be computed using the `roc_auc()` function from {yardstick}:

```
pred_prob_train_both %>% group_by(model) %>%
  roc_auc(truth = observed, prob_class_1, event_level="second")
```

```
# A tibble: 2 × 4
  model  .metric .estimator .estimate
  <chr>  <chr>   <chr>          <dbl>
1 First  roc_auc binary         0.999
2 Second roc_auc binary         0.994
```

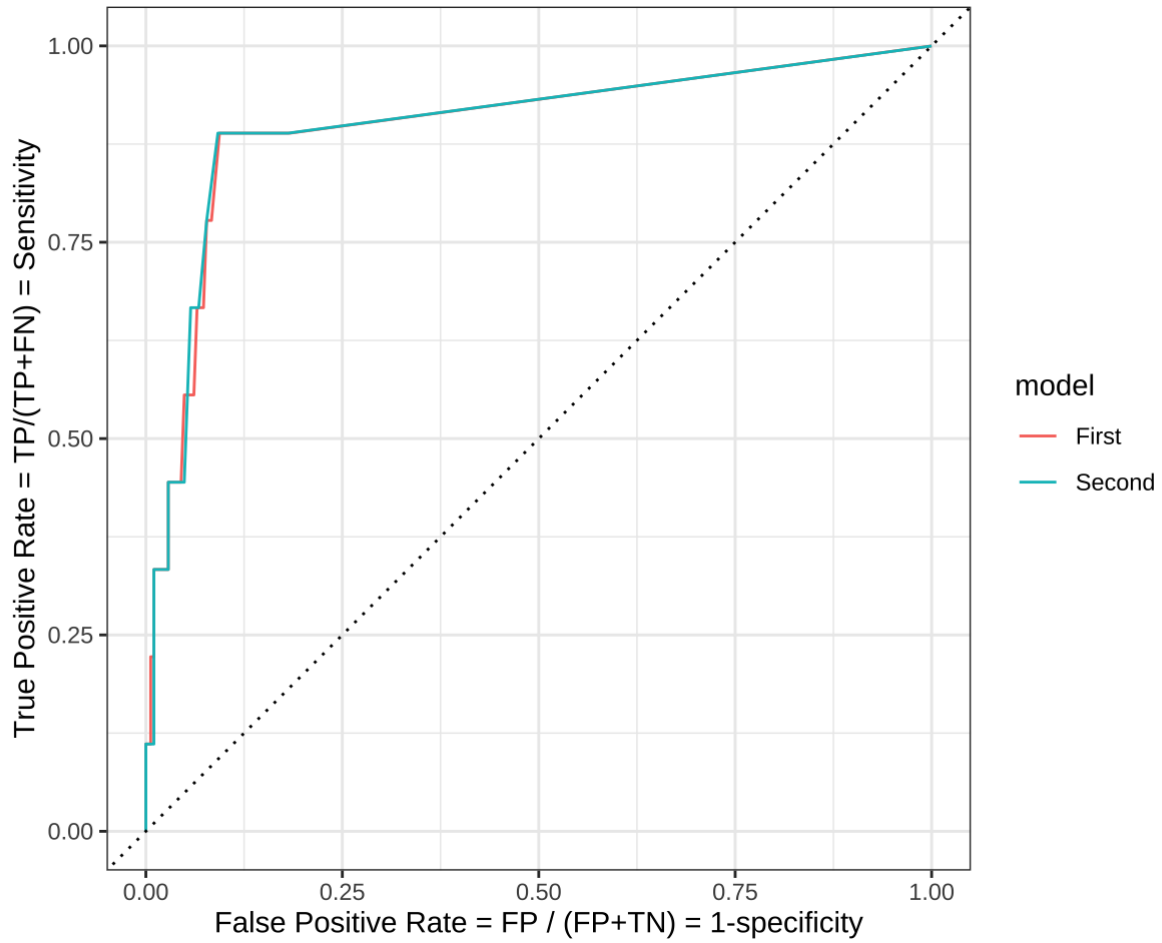
The AUC is really close to 1. The first model seems to provide better results than the second, with regard to the Area Under Curve.

And with the test data:

```
pred_prob_test_both <-
  tibble(observed = df_test$y, prob_class_1 = pred_prob_test[,"1"],
       model = "First") %>%
  bind_rows(
    tibble(observed = df_test$y, prob_class_1 = pred_prob_test_2[,"1"],
           model = "Second")
  )
```

```
pred_prob_test_both %>%
  group_by(model) %>%
  roc_curve(truth = observed, prob_class_1, event_level="second") %>%
  autoplot() +
  labs(x = "False Positive Rate = FP / (FP+TN) = 1-specificity",
       y = "True Positive Rate = TP/(TP+FN) = Sensitivity")
```

ROC Curve for both models, test data



On the test data, the picture is less clear. Let us compute the values:

```
pred_prob_test_both %>% group_by(model) %>%
  roc_auc(truth = observed, prob_class_1, event_level="second")
```

```
# A tibble: 2 × 4
  model  .metric .estimator .estimate
  <chr>  <chr>   <chr>          <dbl>
1 First  roc_auc binary         0.899
2 Second roc_auc binary         0.900
```

Optimal threshold

Let us now turn to finding the best probability threshold for a model. We have seen that a perfect classifier is located on the top-left corner of the plot of the ROC curve (FPR=0, TPR=1).

There are many ways to define the optimal threshold value (see Unal (2017)).

For simplicity, let us use the Euclidean distance to find the closest point to (0,1). The optimal threshold $\tau^\star$ thus minimises the following distance:

$$d(\tau) = \sqrt{\text{Sensitivity}(\tau)^2 + \left((1 - \text{Specificity}(\tau)\right)^2}$$

where $\text{Sensitivity} = \frac{TP}{TP+FN}$ and $\text{Specificity} = 1 - \frac{FP}{FP+TN}$.

Let us define a function that computes the Euclidean distance from two points `a` and `b`, where `a` and `b` are provided in a table:

```
eucl_dist_tibble <- function(a,b) sqrt(rowSums(a-b)^2)
```

The distance from every point $(\text{Sensitivity}(\tau), 1 - \text{Specificity}(\tau))$ to the point (0,1) can the be computed as follows:

```
distances <-
  tibble(observed = df_test$y, prob_class_1 = pred_prob_test[,"1"]) %>%
  roc_curve(truth = observed, prob_class_1, event_level="second") %>%
  filter(`.threshold` >0, `.threshold`<1) %>%
  mutate(dist = eucl_dist_tibble(cbind(sensitivity, 1-specificity),
                                 cbind(rep(0, n()), rep(1, n())))) %>%
  arrange(dist)
distances
```

```
# A tibble: 34 × 4
   .threshold specificity sensitivity    dist
        <dbl>       <dbl>       <dbl>   <dbl>
 1      0.015       0.886       0.889 0.00294
 2      0.02        0.906       0.889 0.0174
 3      0.01        0.864       0.889 0.0253
 4      0.005       0.817       0.889 0.0722
 5      0.025       0.916       0.778 0.139
 6      0.03        0.923       0.778 0.145
 7      0.035       0.927       0.667 0.260
 8      0.04        0.935       0.667 0.268
 9      0.045       0.939       0.556 0.383
10      0.055       0.943       0.556 0.387
# … with 24 more rows
```

The optimal threshold obtained with this method is thus:

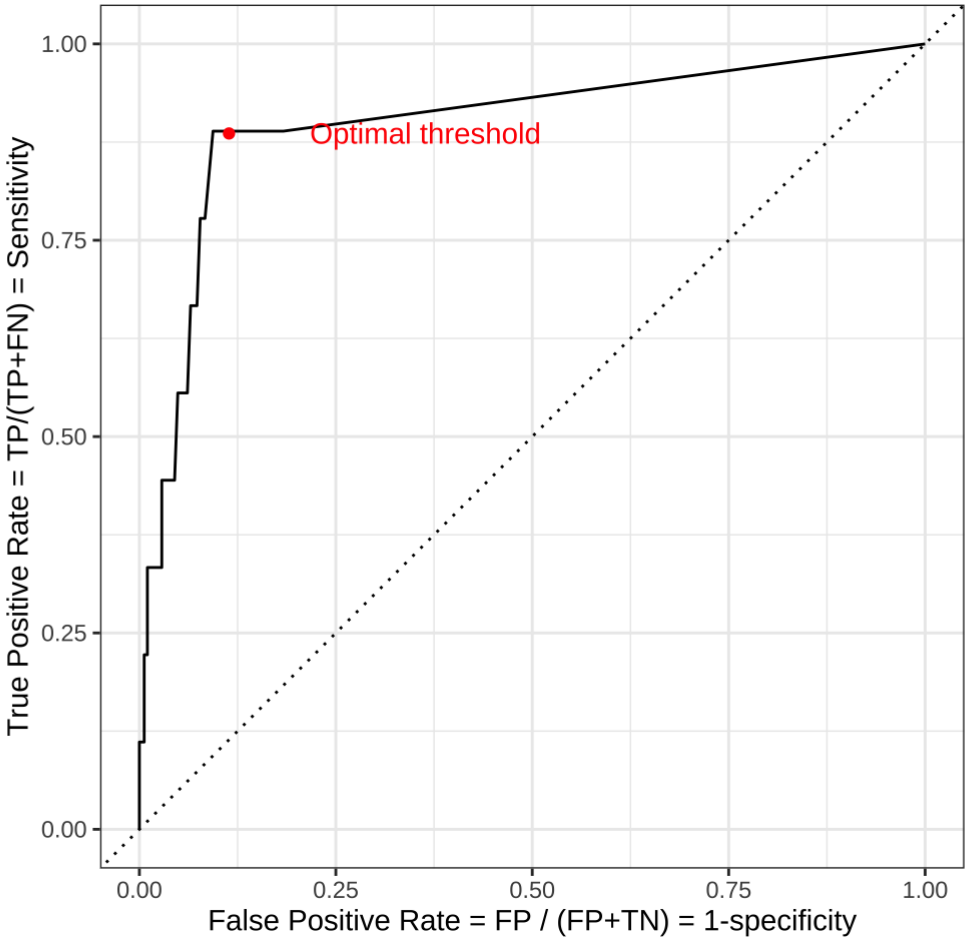
```
optimal_tau <- distances %>% slice(1) %>% .$.threshold
optimal_tau
```

```
  432
0.015
```

```
tibble(observed = df_test$y, prob_class_1 = pred_prob_test[,"1"]) %>%
  roc_curve(truth = observed, prob_class_1, event_level="second") %>%
  autoplot() +
  labs(x = "False Positive Rate = FP / (FP+TN) = 1-specificity",
```

```
       y = "True Positive Rate = TP/(TP+FN) = Sensitivity") +
  geom_point(data = distances %>% slice(1),
             mapping = aes(x = 1-specificity, y = specificity),
             colour = "red") +
  geom_text(data = distances %>% slice(1),
             mapping = aes(x = 1-specificity + .25, y = specificity),
            colour = "red", label = "Optimal threshold")
```

ROC Curve for the logistic model, test data



The predictions made using that probability threshold for the classifier can then computed as follows:

```
pred_class_train_opt <- ifelse(pred_prob_train[,"1"] > optimal_tau, "1", "0") %>%
  factor(levels = c("0", "1"))
```

And the confusion matrix obtained is:

```
caret::confusionMatrix(data = pred_class_train_opt,
                       reference = df_train$y,
                       positive="1")
```

```
Confusion Matrix and Statistics

          Reference
Prediction    0    1
```

```
         0 1821    0
         1  138   41

               Accuracy : 0.931
                 95% CI : (0.919, 0.9417)
    No Information Rate : 0.9795
    P-Value [Acc > NIR] : 1

                  Kappa : 0.3511

 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 1.0000
            Specificity : 0.9296
         Pos Pred Value : 0.2291
         Neg Pred Value : 1.0000
             Prevalence : 0.0205
         Detection Rate : 0.0205
   Detection Prevalence : 0.0895
      Balanced Accuracy : 0.9648

       'Positive' Class : 1
```

And for the test set:

```
pred_class_test_opt <- ifelse(pred_prob_test[,"1"] > optimal_tau, "1", "0") %>%
  factor(levels = c("0", "1"))
```

And the confusion matrix obtained is:

```
caret::confusionMatrix(data = pred_class_test_opt,
                       reference = df_test$y,
                       positive="1")
```

```
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 445   1
         1  46   8

               Accuracy : 0.906
                 95% CI : (0.877, 0.9301)
    No Information Rate : 0.982
    P-Value [Acc > NIR] : 1

                  Kappa : 0.2302

 Mcnemar's Test P-Value : 1.38e-10

            Sensitivity : 0.8889
            Specificity : 0.9063
```

```
         Pos Pred Value : 0.1481
         Neg Pred Value : 0.9978
             Prevalence : 0.0180
         Detection Rate : 0.0160
   Detection Prevalence : 0.1080
      Balanced Accuracy : 0.8976

       'Positive' Class : 1
```

Precision-recall curve

With both the train and test data, looking at the ROC-curve, we might think that our results are pretty good. Let us look at the **precision-recall curve**, to focus specifically on the positive predictions. Recall that:

- Precision = $\frac{TP}{TP+FP}$;
- Recall = $\frac{TP}{TP+FN}$.

These two metrics focus on the positive class, which is the minority class here. With a dataset where there is a fewer number of observation from class 1, we are interested in assessing ability of the classifier to predict class 1. The Precision and recall metrics are thus helpful to have an idea of the performance of the model on the minority class. The **precision-recall** curve plots the precision as a function of the recall. The different values are obtained, as in the case of the ROC curve, by varying the probability threshold τ.

With classification threshold $\tau = 0$

If the threshold $\tau = 0$, the confusion matrix will be as follows:

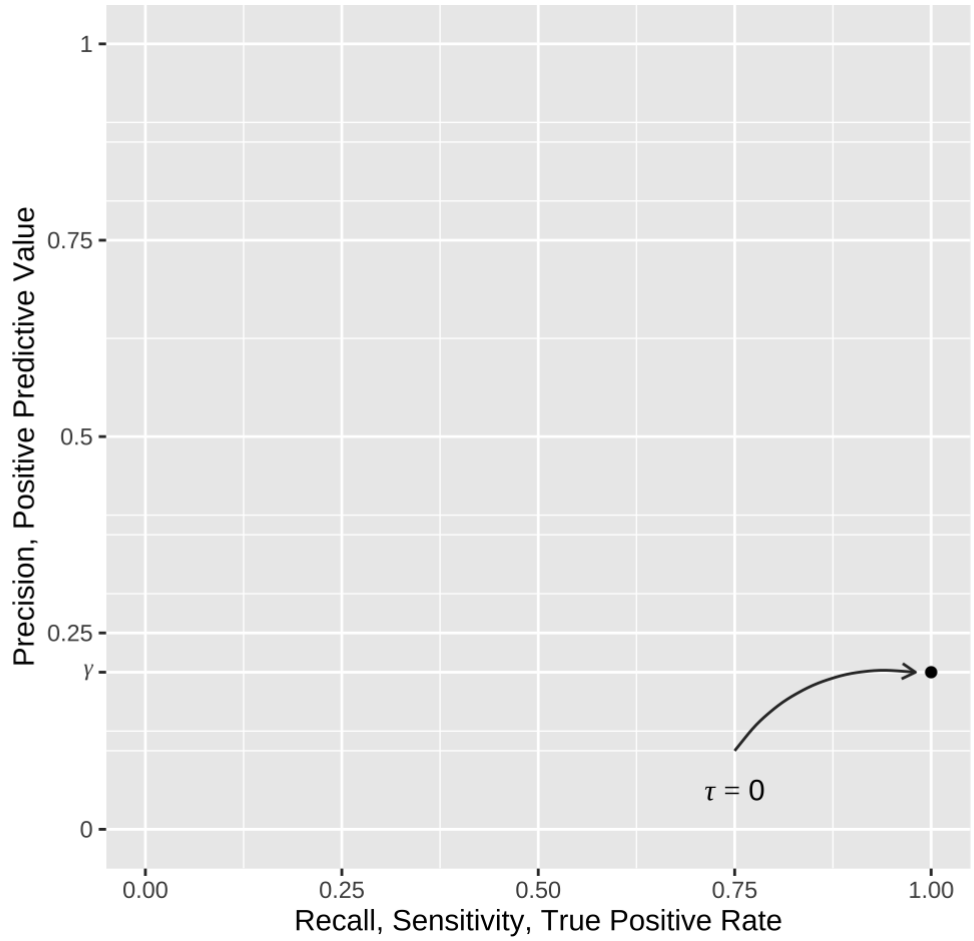
Confusion table (values expressed in proportions) for a classifier, with a threshold $\tau = 0$

	Pred. Negative (0)	**Pred. Positive (1)**
Obs. Negative (0)	True Negative (TN) = 0	False Positive (FP) =$(1-\gamma)$
Obs. Positive (1)	False Negative (FN) = 0	True Positive (TP) = γ

- Precision = $\frac{TP}{TP+FP} = \frac{\gamma}{\gamma+(1-\gamma)} = \gamma$;
- Recall = $\frac{TP}{TP+FN} = \frac{\gamma}{\gamma+0} = 1$.

For example, if $\gamma = 20\%$:

▶ Show the R codes



With classification threshold $\tau = 1$

If the threshold $\tau = 1$, then:

Confusion table (values expressed in proportions) for a classifier, with a threshold $\tau = 0$

	Pred. Negative (0)	**Pred. Positive (1)**
Obs. Negative (0)	True Negative (TN) = $1 - \gamma$	False Positive (FP) = 0
Obs. Positive (1)	False Negative (FN) = γ	True Positive (TP) = 0

Hence:

- Precision = $\frac{TP}{TP+FP}$: undefined.
- Recall = $\frac{TP}{TP+FN} = 0$.

In practice, while the value of τ increases, more and more cases are classified as 0. The proportion of false positive decreases and usually becomes negligible when compared to the proportion of true positive. The precision thus increases and may be equal to 1.

With a perfect classifier

Now, let us turn to a classifier that perfectly predicts the two classes for each case.

Confusion table (values expressed in proportions) for a perfect classifier, with a threshold $0 < \tau < 1$

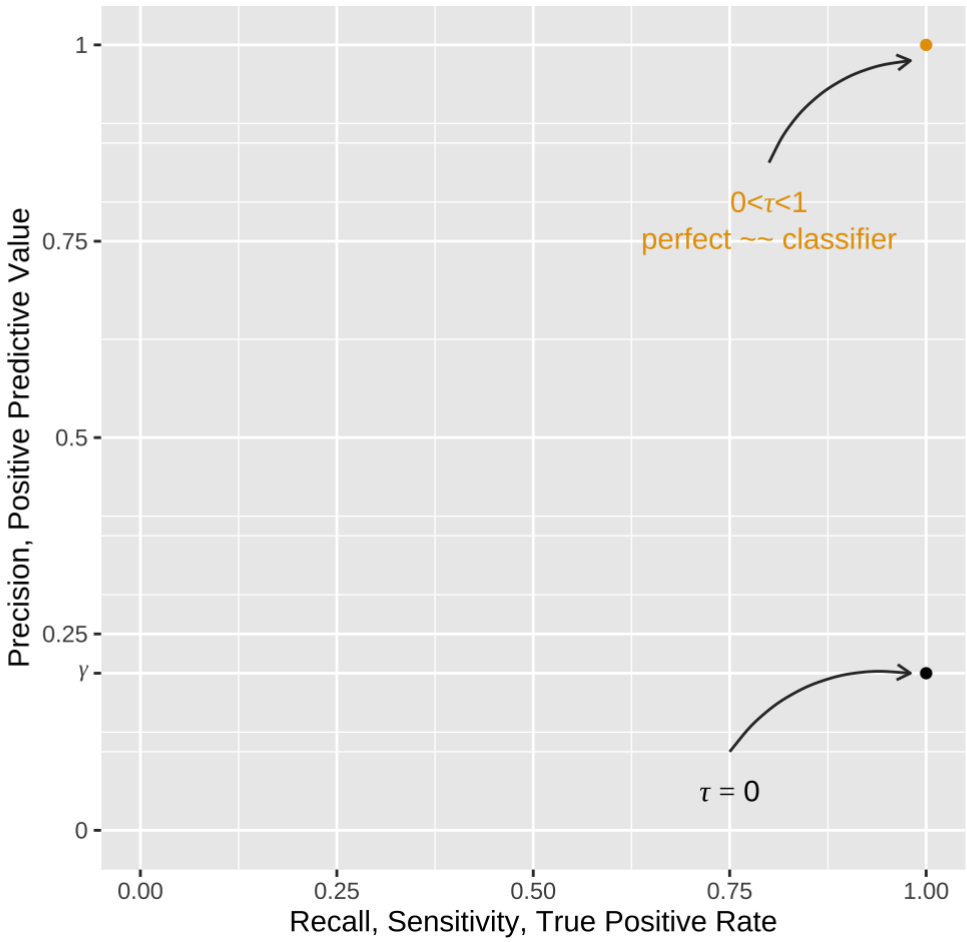
Confusion table (values expressed in proportions) for a perfect classifier, with a threshold $0 < \tau < 1$

	Pred. Negative (0)	**Pred. Positive (1)**
Obs. Negative (0)	True Negative (TN) = $1 - \gamma$	False Positive (FP) = 0
Obs. Positive (1)	False Negative (FN) = 0	True Positive (TP) = γ

In such a case:

- Precision = $\frac{TP}{TP+FP} = 1$;
- Recall = $\frac{TP}{TP+FN} = 1$.

▶ Show the R codes



With a classifier that randomly assigns the classes

If the classifier randomly assigns class 1 with a probability p and class 0 with probability $1 - p$, regardless of τ, we the expected proportions are:

Confusion table (values expressed in proportions) for a perfect classifier, with a threshold $0 < \tau < 1$

	Pred. Negative (0)	**Pred. Positive (1)**
Obs. Negative (0)	True Negative (TN) = $(1 - p)(1 - \gamma)$	False Positive (FP) = $p(1 - \gamma)$

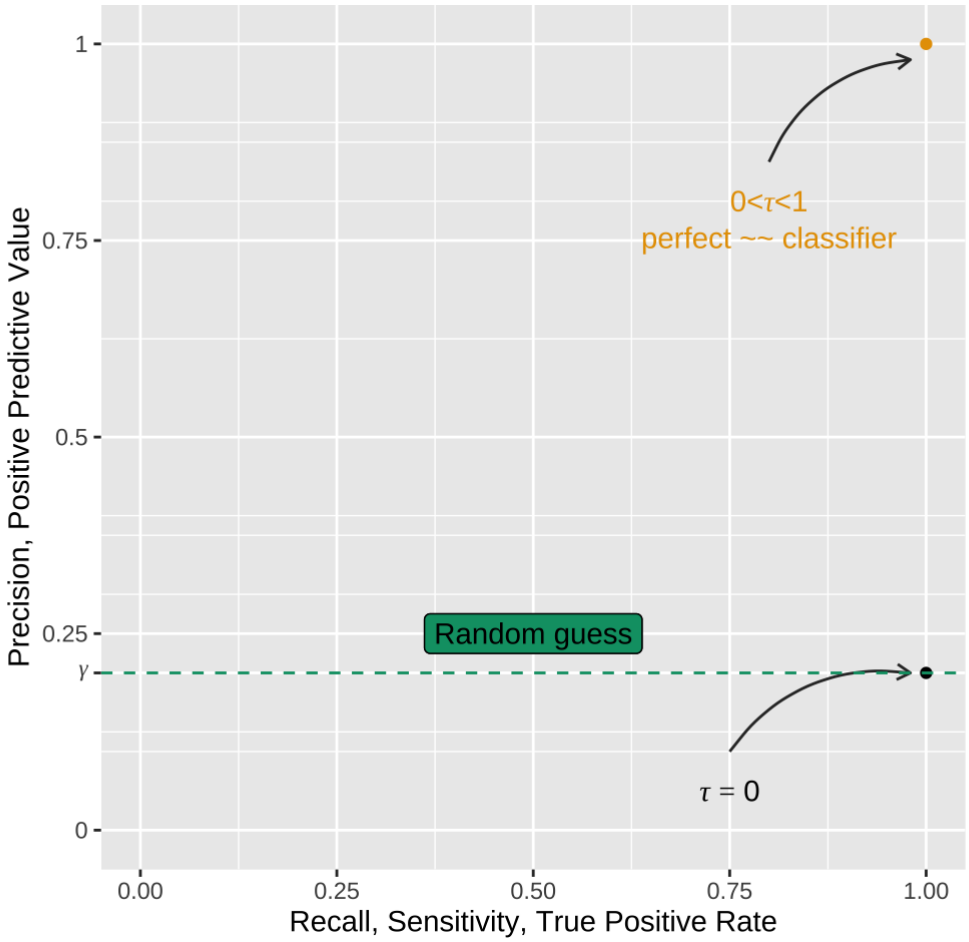
	Pred. Negative (0)	Pred. Positive (1)
Obs. Positive (1)	False Negative (FN) = $(1-p)\gamma$	True Positive (TP) = $p\gamma$

Which leads to:

- Precision = $\frac{TP}{TP+FP} = \frac{p\gamma}{p\gamma+p(1-\gamma)} = \gamma$;
- Recall = $\frac{TP}{TP+FN} = \frac{p\gamma}{p\gamma+(1-p)\gamma} = p$.

Hence, for $0 < \tau < 1$, the point corresponding to a classifier that randomly assigns class 1 with probability p will be on the dashed green line. The x-coordinate of that point will be equal to p (assuming that there is a proportion γ of class 1 individuals in the data).

▶ Show the R codes



With our models

Recall from the confusion matrix (showing proportions) for the first model, on the test sample:

Metric	Training data	Test data
Precision (TP / (TP+FP))	0.98	0.98
Recall/Sensitivity (TP / (TP + FN))	0.11	0.11

And for the second:

Metric	Training data	Test data
Precision (TP / (TP+FP))	0.98	0.98
Recall/Sensitivity (TP / (TP + FN))	0.11	0.11

When faced with an unbalanced dataset and when the model fails to correctly predict the positive class, the ROC curve will not necessarily reflect this if the imbalance is large. The ROC curve will in fact be close to the ROC curve of a perfectly discriminating model. In contrast, the Precision-Recall curve will not be as close as the one corresponding to a perfect classifier. Let us illustrate this.

The `pr_curve()` from {yardstick} allows us to get the precision and recall for different values of τ.

```
tibble(observed = df_train$y, prob_class_1 = pred_prob_train[,"1"]) %>%
  pr_curve(truth = observed, prob_class_1, event_level="second")
```

```
# A tibble: 66 × 3
   .threshold recall precision
        <dbl>  <dbl>     <dbl>
 1    Inf     0             1
 2      0.995 0.0732        1
 3      0.97  0.0976        1
 4      0.965 0.146         1
 5      0.9   0.171         1
 6      0.785 0.195         1
 7      0.73  0.220         1
 8      0.65  0.244         1
 9      0.635 0.268         1
10      0.58  0.293         1
# … with 56 more rows
```

And for the second model:

```
tibble(observed = df_train$y, prob_class_1 = pred_prob_train_2[,"1"]) %>%
  pr_curve(truth = observed, prob_class_1, event_level="second")
```

```
# A tibble: 52 × 3
   .threshold recall precision
        <dbl>  <dbl>     <dbl>
 1    Inf     0         1
 2      0.995 0.0732    1
 3      0.975 0.146     1
 4      0.915 0.171     1
 5      0.755 0.195     1
 6      0.705 0.220     1
 7      0.47  0.244     1
 8      0.43  0.268     1
 9      0.415 0.293     1
10      0.4   0.293     0.857
#   with 42 more rows
```

```
# … with 42 more rows
```

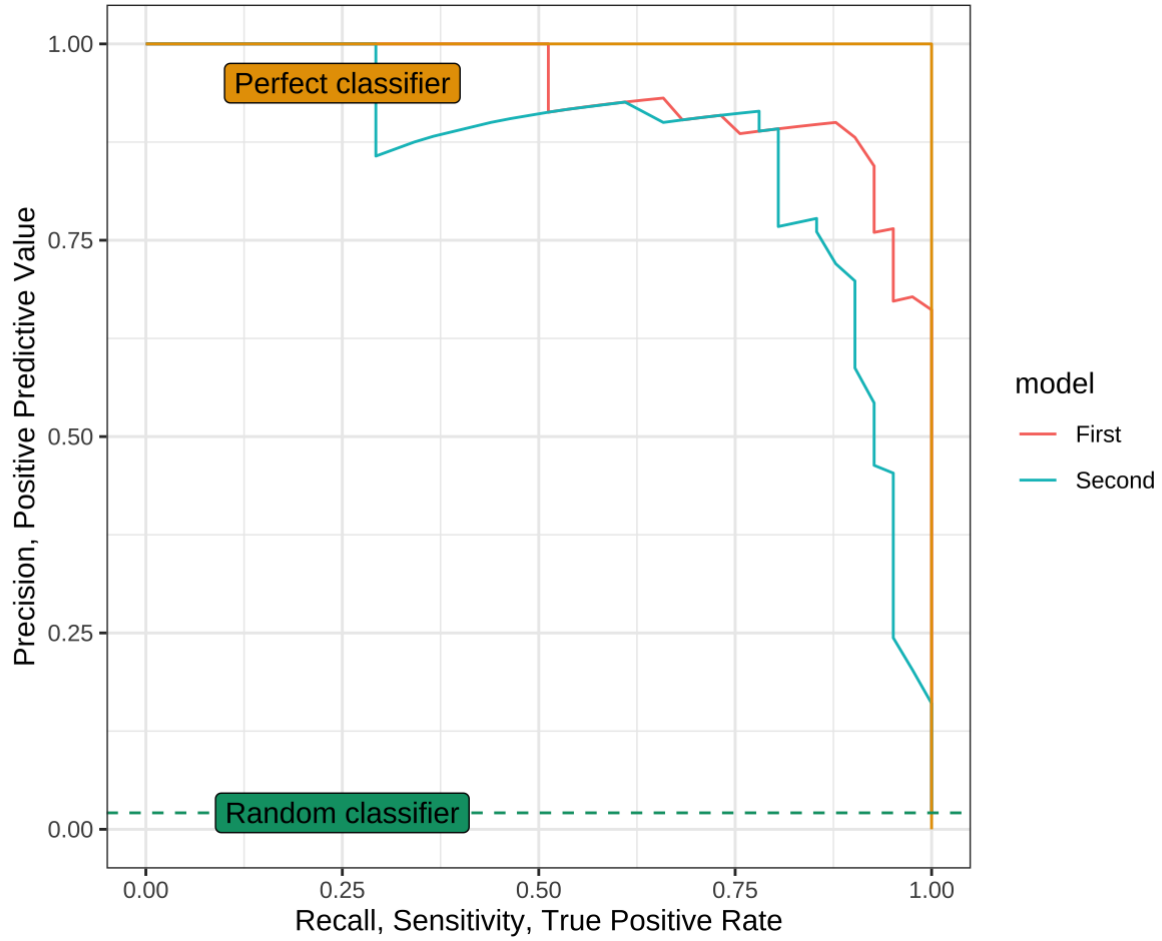
Putting those in a single table:

```
pr_curve_both_train <-
  tibble(observed = df_train$y, prob_class_1 = pred_prob_train[,"1"],
         model = "First") %>%
  bind_rows(
    tibble(observed = df_train$y, prob_class_1 = pred_prob_train_2[,"1"],
           model = "Second")
  ) %>%
  group_by(model) %>%
  pr_curve(truth = observed, prob_class_1, event_level="second")
```

Again, the `autoplot()` function can be used to plot the curve from the obtained table.

```
pr_curve_both_train %>%
  autoplot() +
  geom_line(tibble(recall = c(0,0.9999,1), precision = c(1,1,0)),
            mapping = aes(x = recall, y = precision),
            colour = "#E69F00") +
  geom_hline(yintercept = sum(df_train$y==1) / sum(df_train$y==0), linetype = "dashed"
  geom_label(data = tibble(x = c(.25, 0.25), y = c(sum(df_train$y==1) / sum(df_train$y
                           lab = c("Random classifier", "Perfect classifier")),
             mapping = aes(x=x, y=y, label=lab, fill = lab)) +
  scale_fill_manual(NULL, guide = "none",
                    values = c("Random classifier" = "#009E73",
                               "Perfect classifier" = "#E69F00")) +
  coord_equal() +
  labs(x = "Recall, Sensitivity, True Positive Rate",
       y = "Precision, Positive Predictive Value")
```

Precision-recall for both models, training data



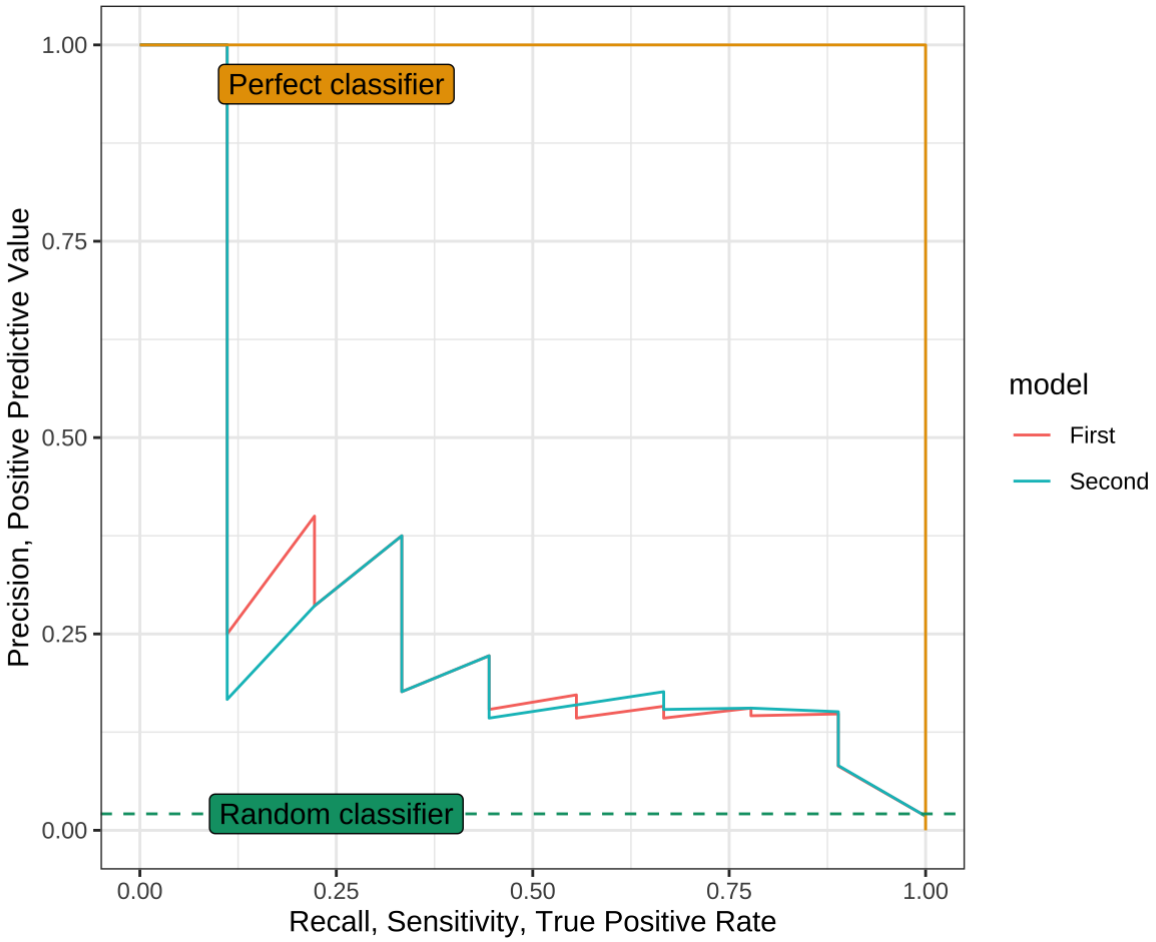
With the test data:

```
pr_curve_both_test <-
  tibble(observed = df_test$y, prob_class_1 = pred_prob_test[,"1"],
         model = "First") %>%
  bind_rows(
    tibble(observed = df_test$y, prob_class_1 = pred_prob_test_2[,"1"],
           model = "Second")
  ) %>%
  group_by(model) %>%
  pr_curve(truth = observed, prob_class_1, event_level="second")
```

```
pr_curve_both_test %>%
  autoplot() +
  geom_line(tibble(recall = c(0,0.9999,1), precision = c(1,1,0)),
            mapping = aes(x = recall, y = precision),
            colour = "#E69F00") +
  geom_hline(yintercept = sum(df_train$y==1) / sum(df_train$y==0), linetype = "dashed"
  geom_label(data = tibble(x = c(.25, 0.25), y = c(sum(df_train$y==1) / sum(df_train$y
                           lab = c("Random classifier", "Perfect classifier")),
             mapping = aes(x=x, y=y, label=lab, fill = lab)) +
  scale_fill_manual(NULL, guide = "none",
                    values = c("Random classifier" = "#009E73",
                               "Perfect classifier" = "#E69F00")) +
  coord_equal() +
```

```
  coord_equal() +
  labs(x = "Recall, Sensitivity, True Positive Rate",
       y = "Precision, Positive Predictive Value")
```

Precision-recall for the both models, test data



Clearly, the performances of the model are much less impressive viwed that way...

F1 Score

With an imbalanced dataset, if we focus on the accuracy, we might select a model that gives very good results on average, but poor results to predict one of the classes. Some metrics are built to take into consideration the predictive capabilities for both classes.

This is the case of the F1 score. It computes the harmonic mean of the precision an sensitivity:

$$\text{F1 score} = 2 \times \frac{\text{Precision} \times \text{Sensitivity}}{\text{Precision} + \text{Sensitivity}} = \frac{2 \times TP}{2 \times TP + FP + FN}.$$

The F1 score values range from 0 (if either precision or sensitivity is equal to 0) to 1 (both precision and sensitivity equal to 1).

With an imbalanced dataset, it may be useful to rely on this metric to assess the quality of fit of a given classifier when performing model selection.

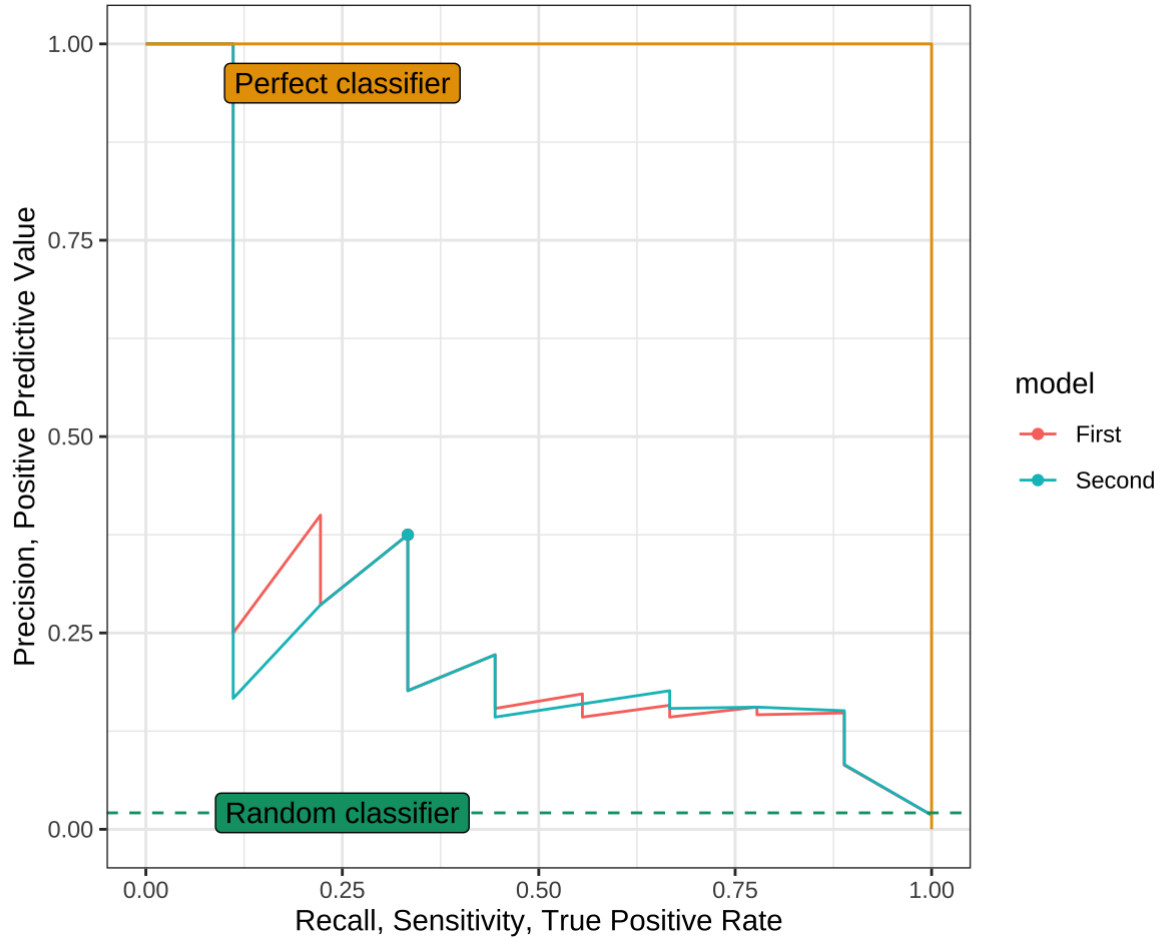
With our two models, we can use the F1-Score to select the optimal threshold:

```
thresholds_F1 <-
  pr_curve_both_test %>%
  mutate(F1_score = 2*precision*recall/ (precision+recall)) %>%
  group_by(model) %>%
  arrange(desc(F1_score)) %>%
  slice(1)
thresholds_F1
```

```
# A tibble: 2 × 5
# Groups:   model [2]
  model  .threshold recall precision F1_score
  <chr>       <dbl>  <dbl>     <dbl>    <dbl>
1 First       0.295  0.333     0.375    0.353
2 Second      0.21   0.333     0.375    0.353
```

```
pr_curve_both_test %>%
  autoplot() +
  geom_line(tibble(recall = c(0,0.9999,1), precision = c(1,1,0)),
            mapping = aes(x = recall, y = precision),
            colour = "#E69F00") +
  geom_point(data = thresholds_F1, mapping = aes(x = recall, y = precision, colour = m
  geom_hline(yintercept = sum(df_train$y==1) / sum(df_train$y==0), linetype = "dashed"
  geom_label(data = tibble(x = c(.25, 0.25), y = c(sum(df_train$y==1) / sum(df_train$y
                           lab = c("Random classifier", "Perfect classifier")),
             mapping = aes(x=x, y=y, label=lab, fill = lab)) +
  scale_fill_manual(NULL, guide = "none",
                    values = c("Random classifier" = "#009E73",
                               "Perfect classifier" = "#E69F00")) +
  coord_equal() +
  labs(x = "Recall, Sensitivity, True Positive Rate",
       y = "Precision, Positive Predictive Value")
```

Precision-recall for the both models, test data



Warning

When faced with an imbalanced dataset, what should we do?

Unfortunately (or maybe, fortunately?), there is no "good" answer to this question. There are multiple ways to address this issue. And as is often the case with machine learning, we have to try different options and we have to make choices.

As of today, there are three ways to deal with imbalanced datasets:

1. data resampling
2. algorithm modifications (not in the scope of this notebook)
3. cost-sensitive learning (not in the scope of this notebook).

In the remainder of this notebook, we will have a look at data resampling methods.

Rebalancing the dataset

We have seen in the first part of this notebook that in the presence of a data sample containing much more observations of the negative class than of the positive class, it is difficult to obtain an algorithm with good predictive capabilities for the minority (positive) class. Yet, obtaining good performance in predicting the minority class may be the precise purpose of the classification (detection of cancer, fraud, recession, ...). In this case, it is possible to re-equillibrate the data set to

decrease the imbalance ratio.

This part of the notebook will present two broad techniques:

1. under-sampling: the number of observations from the majority class is decreased
2. over-sampling: the number of observations from the minority class is increased.

Under-sampling

A way to obtain a balanced dataset consists in sampling from the majority class to decrease the number of observations at hand from this class.There a many techniques to do so.

Random under-sampling

One of the most simple technique is random sampling. All we need to do is to decide the desired imbalance ratio we would like to obtain. For example, if we want that ratio to be equal to 1 (1 minority case for each majority case), we just need to sample as many examples of the majority as there are examples of the minority class.

In the traiing set, the imbalance ratio is:

```
sum(df_train$y==0) / sum(df_train$y==1)
```

```
[1] 47.78049
```

The number of minority cases is:

```
(n_minority <- sum(df_train$y==1))
```

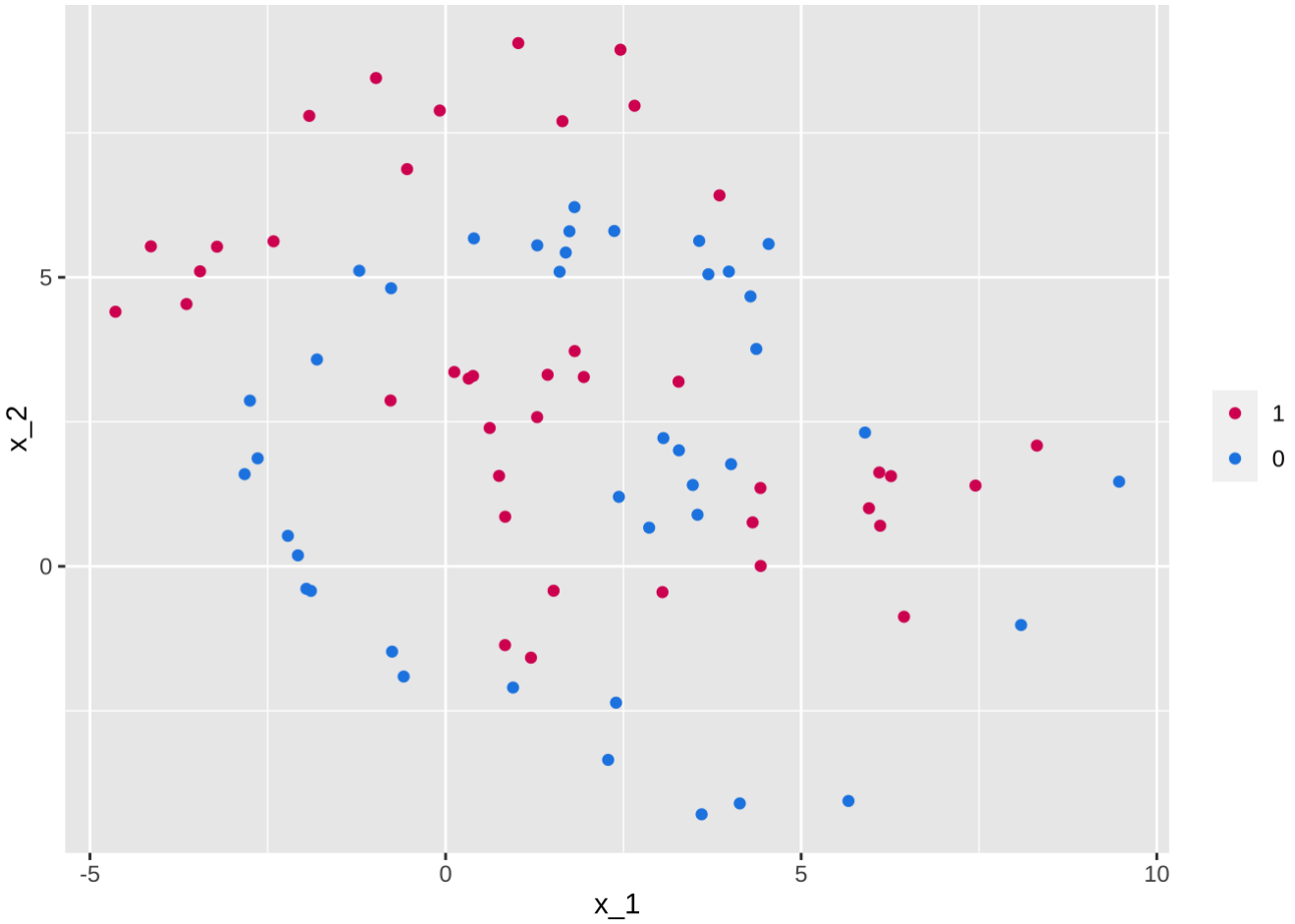
```
[1] 41
```

We will thus randomly sample 41 cases from the majority case, and keep all cases from the minority class.

```
df_train_under_random_1 <-
  df_train %>%
  group_by(y) %>%
  slice_sample(n = n_minority, replace = FALSE) %>%
  ungroup()
```

```
ggplot(data = df_train_under_random_1,
       aes(x = x_1, y = x_2)) +
  geom_point(mapping = aes(colour = y)) +
  scale_colour_manual(NULL, values = c("1" = "#D81B60", "0" = "#1E88E5"))
```

A rebalanced dataset with an imbalance ratio of 1



Undersampling to obtain an imbalance ratio of 1 resulted in a final training set with 2 times the number of minority cases, *i.e.*, 82. This number of observations may be too small, depending on the variability in the data. To have a few more observations on which to train the model, we can accept to have a slightly unbalanced data set.

For example, if we accept to have an imbalance ratio of 2 (2 observations of the majority class for each observation of the minority class):

```
nrow(df_train)
```

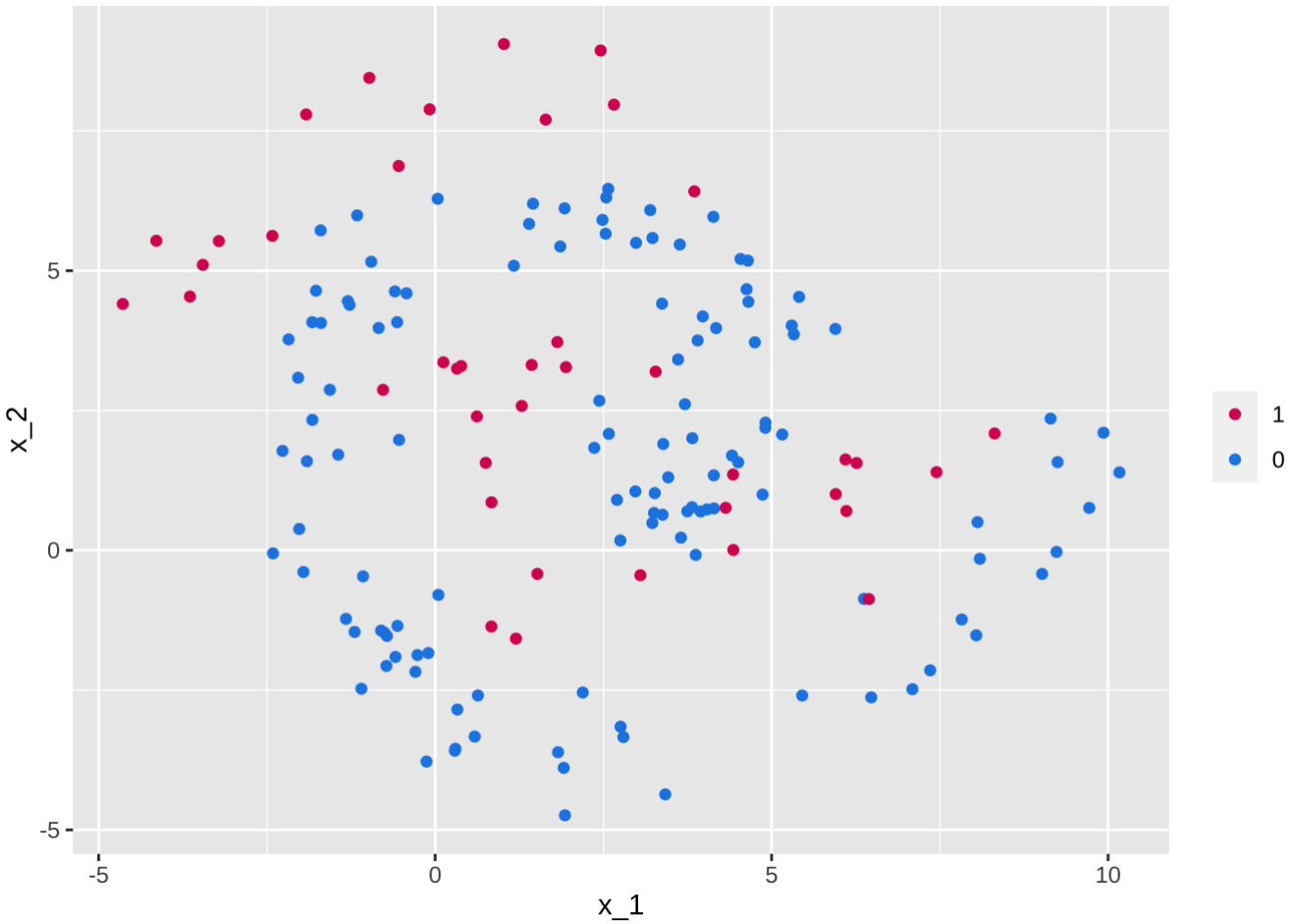
```
[1] 2000
```

```
df_train_under_random_2 <-
  df_train %>%
  # Shuffle observations
  sample_frac(1L) %>%
  group_by(y) %>%
  # Under-sampling in the majority class
  slice_head(n = 3*n_minority) %>%
  ungroup()
```

```
ggplot(data = df_train_under_random_2,
       aes(x = x_1, y = x_2)) +
```

```
  geom_point(mapping = aes(colour = y)) +
  scale_colour_manual(NULL, values = c("1" = "#D81B60", "0" = "#1E88E5"))
```

A rebalanced dataset with an imbalance ratio of 2



When observations are randomly drawn from the majority class, the resulting sample may not be representative of the original data. Removing some individuals may result in removing important information from the training dataset.

Therefore, instead of drawing randomly among the observations of the majority class, some techniques focus on orienting the drawing (by removing redundant individuals, for example), or on creating synthetic individuals among the individuals of the majority class.

Undersampling using KNN

Instead of randomly select observations to keep or remove from the majority class, Mani and Zhang (2003) suggest using the nearest neighbors. They propose 3 different versions.

Note

- NearMiss-1:
  - for each example from the majority class, compute the average distance to **k closest examples among the minority class**
  - select majority examples for which the average distance computed in the previous step is the smallest
- NearMiss-2:

- NearMiss-2:
  - for each example from the majority class, compute the average distance to **k farthest examples among the minority class**
  - select majority examples for which the average distance computed in the previous step is the smallest
- NearMiss-3:
  - for each example from the minority class, identify and select a given number of neighbors among the majority class.
  - for each of the neighbors obtained in the previous step, compute the distance to each example from the the minority class, then, compute the average distance from their 3 closest neighbors
  - select majority examples for which the average distance computed in the previous step is the farthest.

As the NearMiss-1 technique seem to provide poor results, some suggest to select majority examples for which the average distance computed in the previous step is the **highest**. [ref needed]

Let us illustrate with a few R codes how these variants work. First, let us split the sample according to the value of the target variable: the majority class and the minority class.

```
majority_sample <- df_train %>% filter(y == 0) %>% select(x_1, x_2) %>% as.matrix()
nrow(majority_sample)
```

```
[1] 1959
```

```
minority_sample <- df_train %>% filter(y == 1) %>% select(x_1, x_2) %>% as.matrix()
nrow(minority_sample)
```

```
[1] 41
```

Let us say that we want to use $k = 3$ for the KNN:

```
n_neighbors <- 3
```

NearMiss-1

In a first step, for each observation from the minority sample, let us compute the distance (using the Euclidean distance here) to each observation from the minority sample:

```
dists <- fields::rdist(majority_sample, minority_sample)
```

Each row of the obtained matrix gives the distance to each observation from the minority sample (in column):

```
dim(dists)
```

```
[1] 1959   41
```

For a single example from the majority, we identify the closests observations from the minority sample. The distances of the first example to each observation from the minority sample are:

```
dists[1,]
```

```
 [1]  5.560756  8.386229  3.836185  5.257663  4.611472  3.375368  3.465841
 [8]  5.310080  7.891209  5.513237 10.274421  3.288169  3.620326  7.294134
[15]  7.229950  9.132841  3.784633  5.755599  4.298604  6.833199  5.378315
[22]  6.527019  6.739599  8.184762  3.146240  6.750714  3.027914  2.554185
[29]  9.280898  5.349391  4.274212  5.804782  2.894196  2.650842  6.136897
[36]  9.872539  5.302912  5.072795  8.952452  2.533850  4.851702
```

Using the `order()` function, we can get the index of the examples from the minority cases, ordered by descending values of the distance:

```
order(dists[1,])
```

```
 [1] 40 28 34 33 27 25 12  6  7 13 17  3 31 19  5 41 38  4 37  8 30 21 10  1 18
[26] 32 35 22 23 26 20 15 14  9 24  2 39 16 29 36 11
```

The distances to the three closest neighbors are:

```
head(sort(dists[1,]), n_neighbors)
```

```
[1] 2.533850 2.554185 2.650842
```

The average of these distances is:

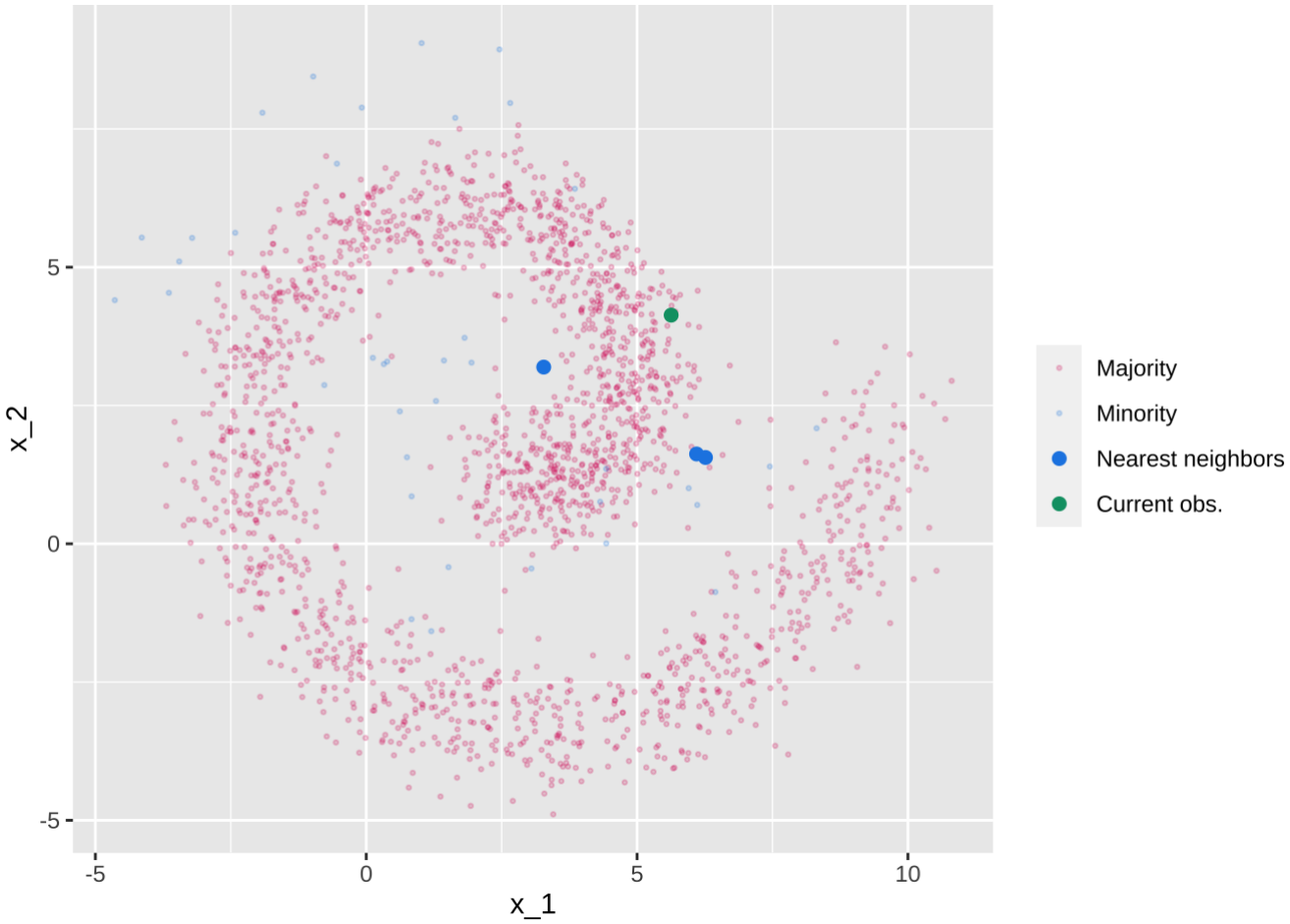
```
mean(head(sort(dists[1,]), n_neighbors))
```

```
[1] 2.579626
```

Visually speaking:

▶ Show the R codes

Computing the average distance from the current observation to the 3-nearest neighbors



Let us compute the average distance to the three nearest neighbors in the minority class for each observation from the majority class:

```
avg_dist <- apply(dists, 1, function(x) mean(head(sort(x), n_neighbors)))
```

Then, we can order the results by descending values of the computed average distance and select the first observations from that list. Let us assume that we to keep only as many observations from the majority class as there are in the minority class:

```
n_to_keep <- n_minority
```

Then the index of the values we keep from the majority class can be obtained as follows:

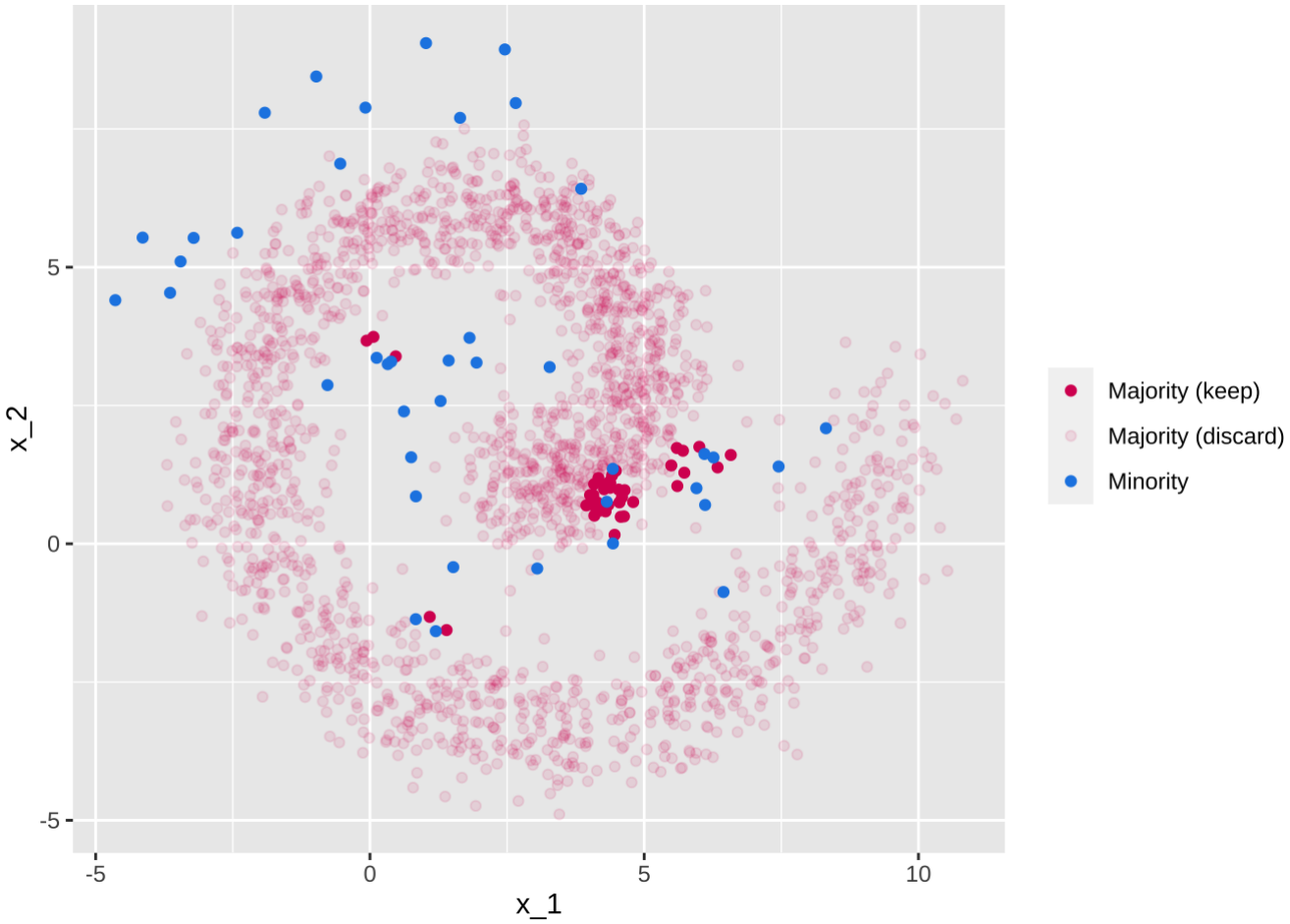
```
ind_to_keep <- order(avg_dist)[1:n_to_keep]
head(ind_to_keep)
```

```
[1] 1874 1942  837  263 1083 1346
```

Let us have a visual representation:

▶ Show the R codes

Observations in the training sample after NearMiss-1 is applied



NearMiss-2

This time, instead of computing the average distance to the closest neighbors, we compute the average distance to the farthest neighbors.

```
dists <- fields::rdist(majority_sample, minority_sample)
dim(dists)
```

```
[1] 1959   41
```

The distance to the farthest neighbors:

```
tail(sort(dists[1,]), n_neighbors)
```

```
[1]  9.280898  9.872539 10.274421
```

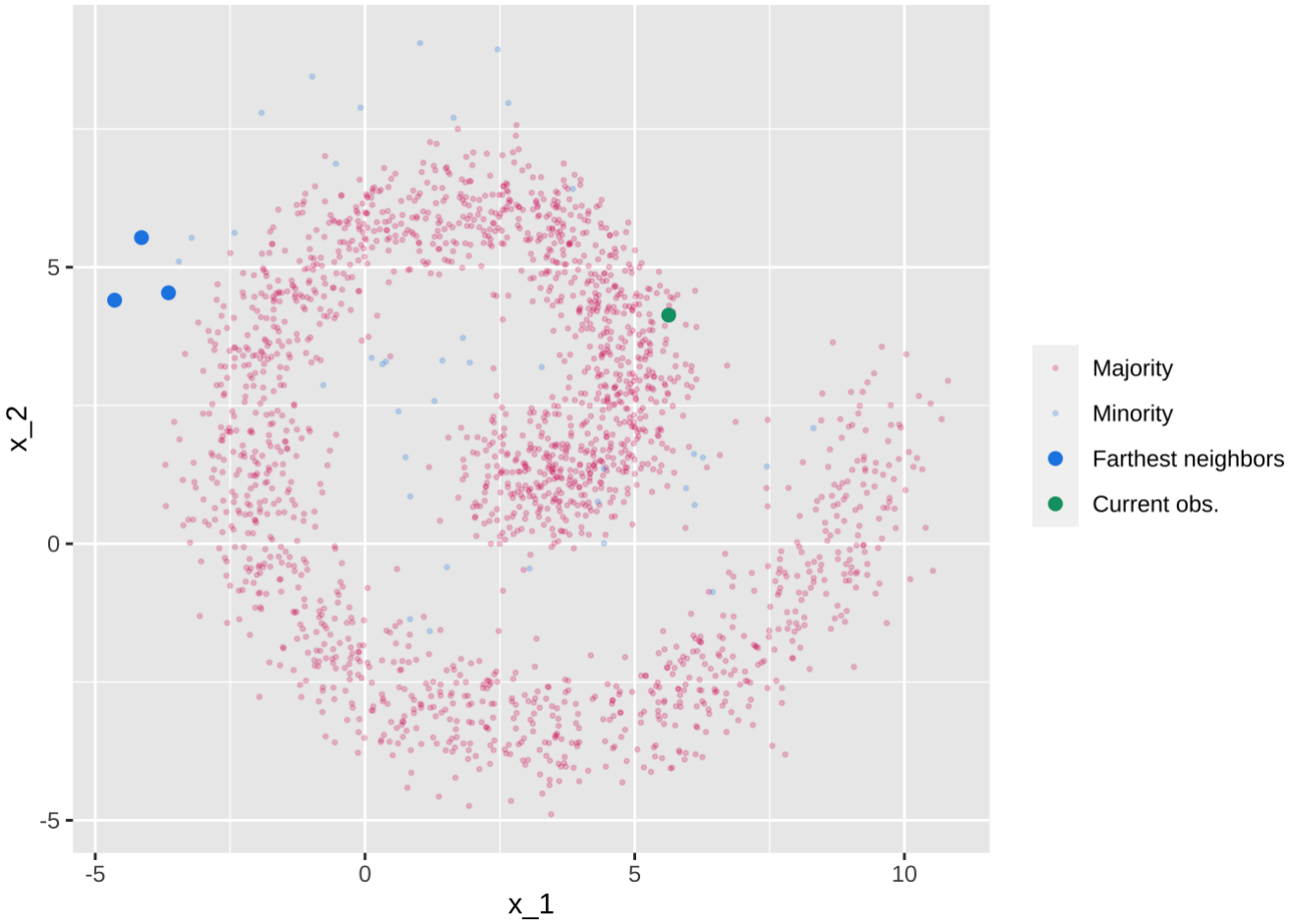
We compute the average:

```
mean(tail(sort(dists[1,]), n_neighbors))
```

```
[1] 9.809286
```

▶ Show the R codes

Computing the average distance from the current observation to the 3-farthest neighbors



We do it for each observation from the majority class:

```
avg_dist_farthest <- apply(dists, 1, function(x) mean(tail(sort(x), n_neighbors)))
```

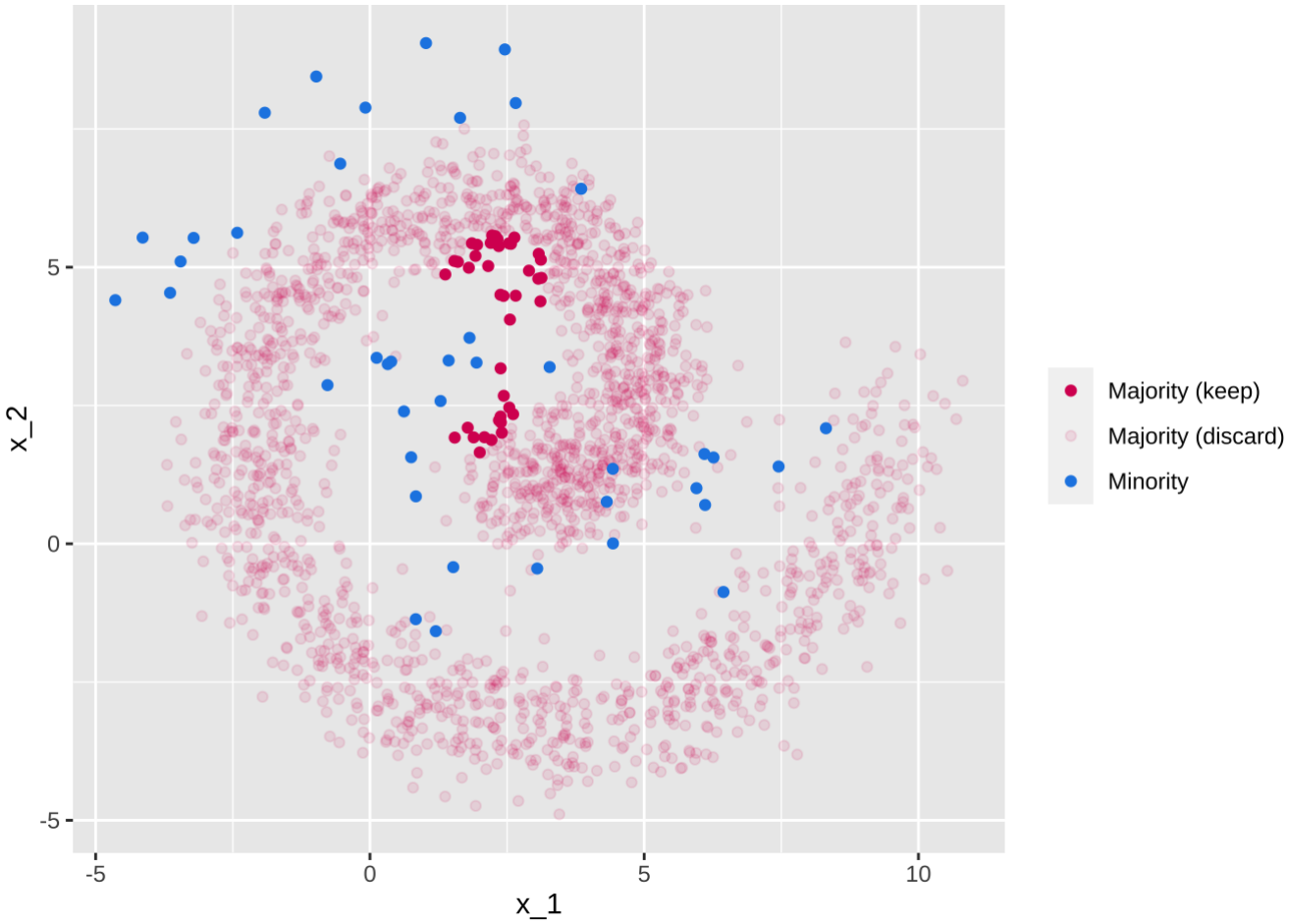
After ordering the results by increasing values, we can select a given number of observations with the smallest distances:

```
ind_to_keep_2 <- order(avg_dist_farthest)[1:n_to_keep]
```

Visually:

▶ Show the R codes

Observations in the training sample after NearMiss-2 is applied



With this solution applied to our spiral dataset, the sampled observations from the majority class seem to be condensed in some regions...

NearMiss-3

Now let us turn to the NearMiss-3.

In a first step, for each example from the minority class, we need to identify their k-nearest neighbors in the majority class.

Let us assume we want to focus on 5 neighbors in this step (this can vary).

```
n_neighbors_version_3 <- 5
```

Then, let us compute the Euclidean distance between each example from the minority class to the examples from the majority class:

```
dists <- fields::rdist(minority_sample, majority_sample)
dim(dists)
```

```
[1]   41 1959
```

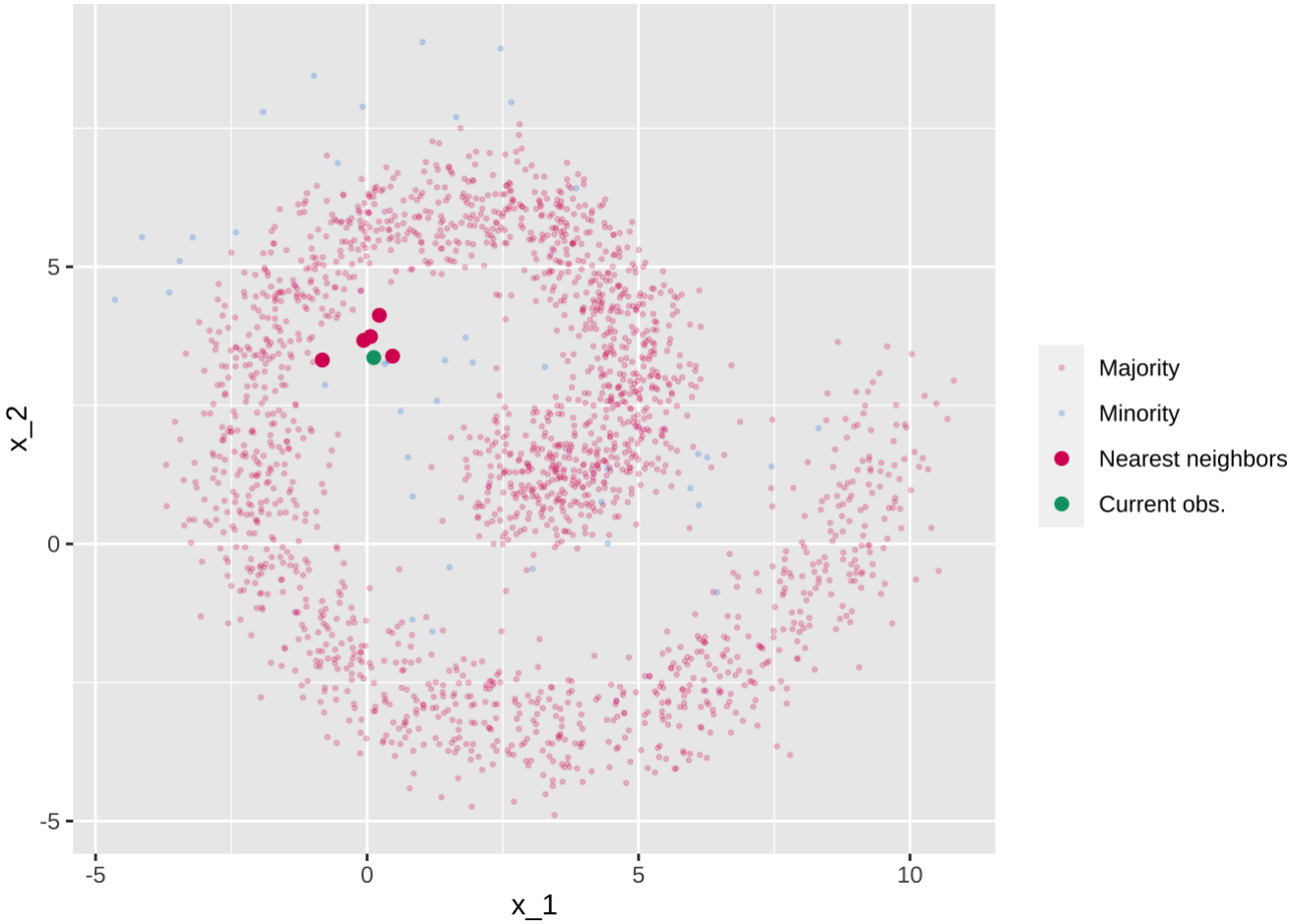
For a single example from the minority class, let us identify its 5-nearest neighbors:

```
order(dists[1,])[1:n_neighbors_version_3]
```

```
[1] 1874 1865 1346  210 1352
```

▶ Show the R codes

5-nearest neighbors from the majority class of the first obs. in the minority class



Let us identify the 5-nearest neighbors in the majority class of each observation from the minority class:

```
ind_to_keep_3 <-
  apply(dists, 1, function(x) order(x)[1:n_neighbors_version_3]) %>%
  as.vector() %>%
  unique()
```

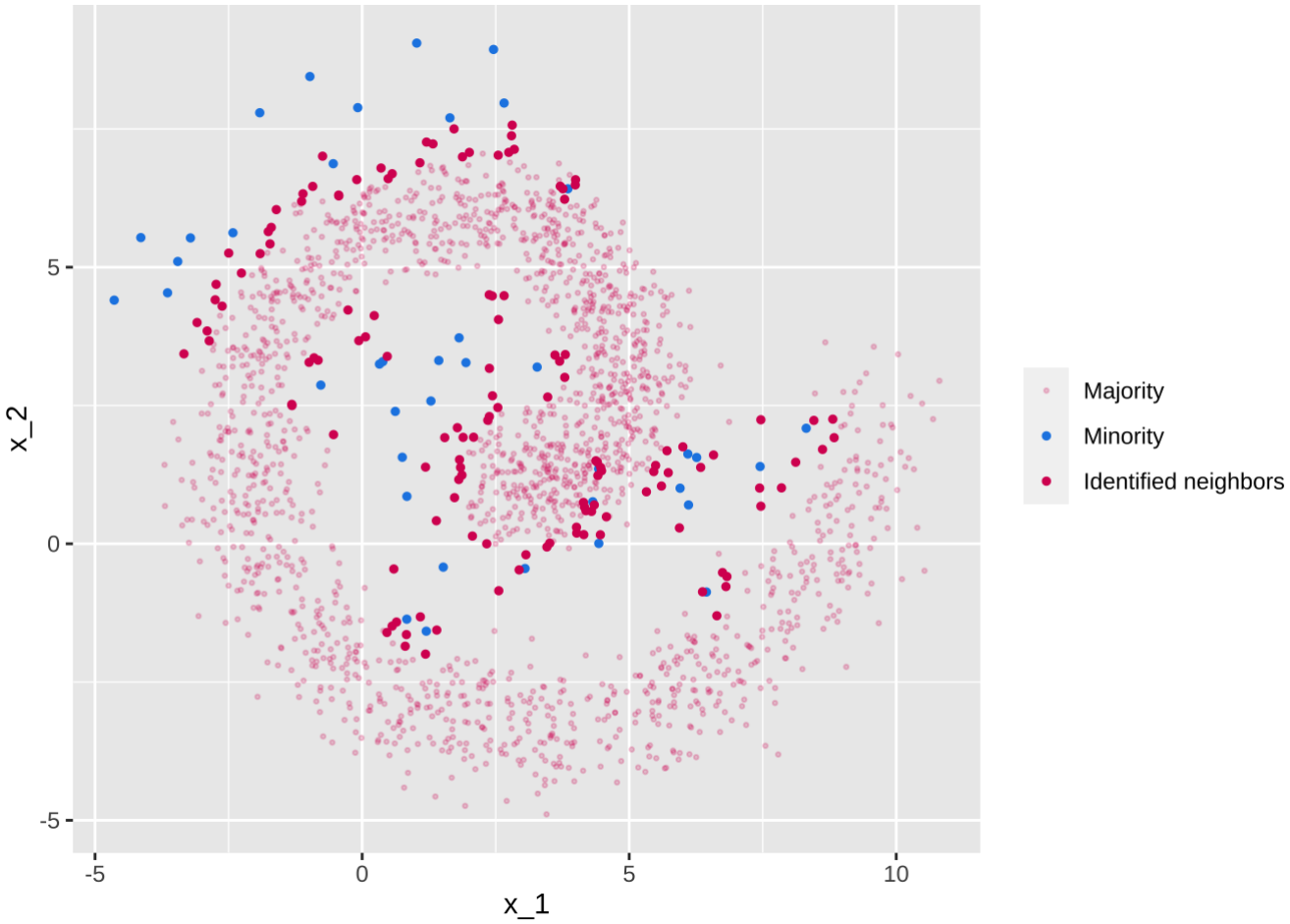
The identified neighbors:

```
majority_sample_neighbors <- majority_sample[ind_to_keep_3, ]
```

From the

▶ Show the R codes

5-nearest neighbors from the majority class of the examples in the minority class



Then, for each of the neighbors we identified, we need to compute the distance to the examples from the minority class, to identify the three closest points from the majority class.

```
dists_2 <- fields::rdist(majority_sample_neighbors, minority_sample)
n_closest <- 3
```

For the first neighbor from the previous step, the 3 closest points from the minority class are the following:

```
head(order(dists_2[1,]), n_closest)
```

```
[1]  8 21  1
```

And the Euclidean distances are:

```
head(sort(dists_2[1,]), n_closest)
```

```
[1] 0.1259316 0.2014562 0.3481431
```

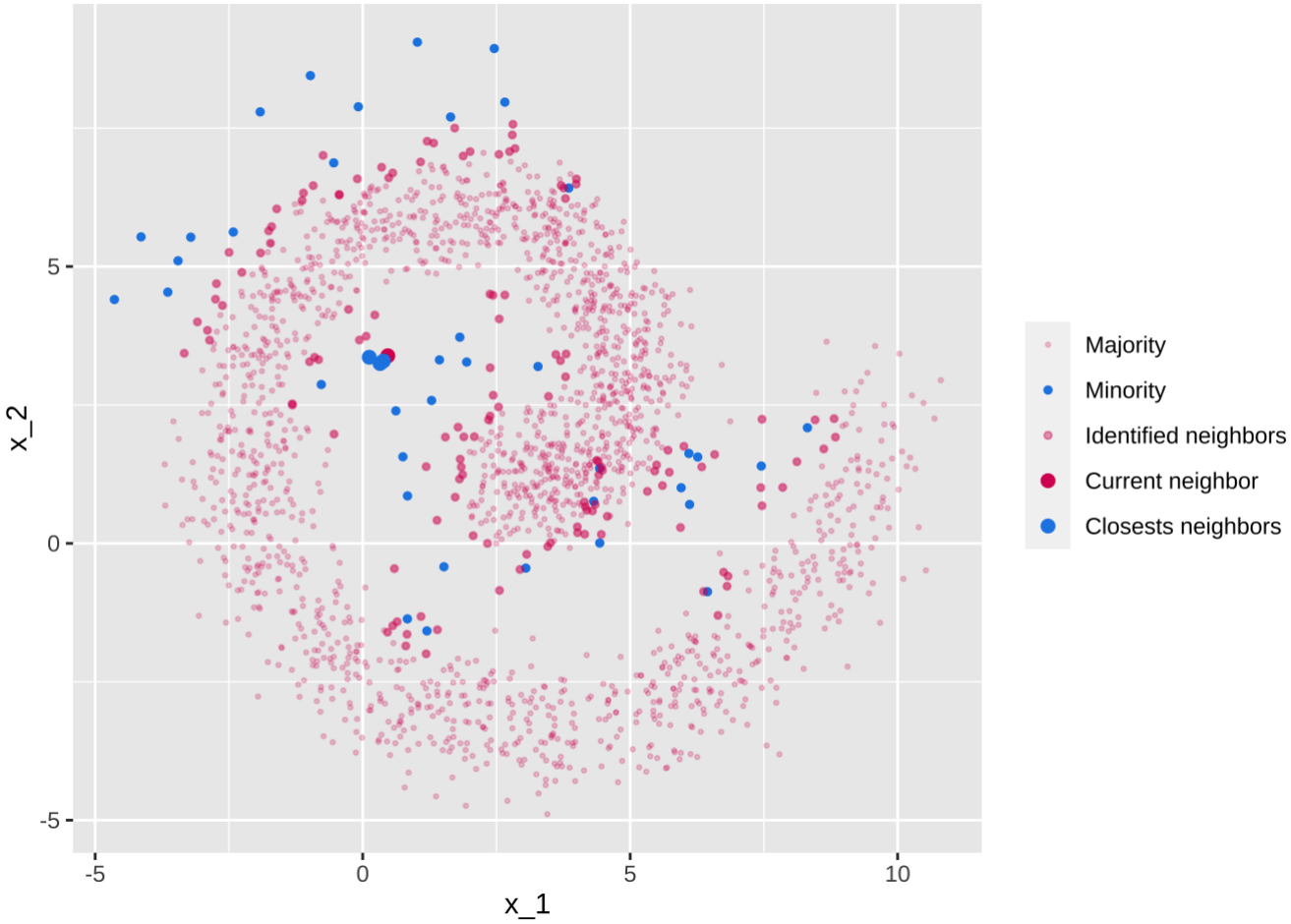
The average of these 3 distances needs to be computed:

```
mean(head(sort(dists_2[1,]), n_closest))
```

```
[1] 0.225177
```

So, for the first identified neighbor (big red dot), we compute the average of the distances to the 3 closest observations from the minority sample (big blue dots):

▶ Show the R codes



For each the neighbors from the majority class, let us compute the average distance to the 3 closest neighbors from the minority class:

```
avg_dist_3 <- apply(dists_2, 1, function(x) mean(head(sort(x), n_closest)))
avg_dist_3
```

```
  [1] 0.2251770 0.5069779 0.4979616 0.8330516 0.8499639 0.9177182 1.2926076
  [8] 1.3161580 1.3170944 1.3635838 0.7123035 0.9729783 1.2604438 1.2729040
 [15] 1.3239252 1.0334778 1.0642164 0.9081734 1.0999691 0.8900858 1.0132932
 [22] 0.8730858 1.1463878 1.2573950 1.1916758 1.2675845 1.3532051 1.5911180
 [29] 1.5432931 1.0571872 0.8105066 0.5749637 0.4850872 0.3579919 0.8148374
 [36] 1.0755783 1.2967133 1.2168434 0.7528816 1.1489869 1.1867125 1.2210328
 [43] 1.1557363 1.4821281 1.3649865 1.0330012 1.5370031 1.0191911 1.1366328
 [50] 1.1874162 1.0300606 0.4745027 0.5465878 0.5171400 0.5440626 0.5462824
 [57] 0.7013093 0.5118003 0.6854913 0.7930765 0.9213224 0.6453580 0.9149546
 [64] 0.8554598 0.9057212 0.7028688 1.1836309 0.9730539 1.1010727 1.1545440
 [71] 1.0045295 1.0773688 0.9591883 1.1410744 1.1503896 0.6555703 0.7212947
 [78] 0.7518483 0.5848023 0.7322752 1.3182153 1.3548690 0.9674024 0.9204197
 [85] 1.3690694 1.3767454 1.2213017 1.2146074 1.3807417 1.2479664 1.1732545
 [92] 1.2840590 1.3243893 1.2944279 0.7114194 0.6790337 1.4304007 1.4393899
```

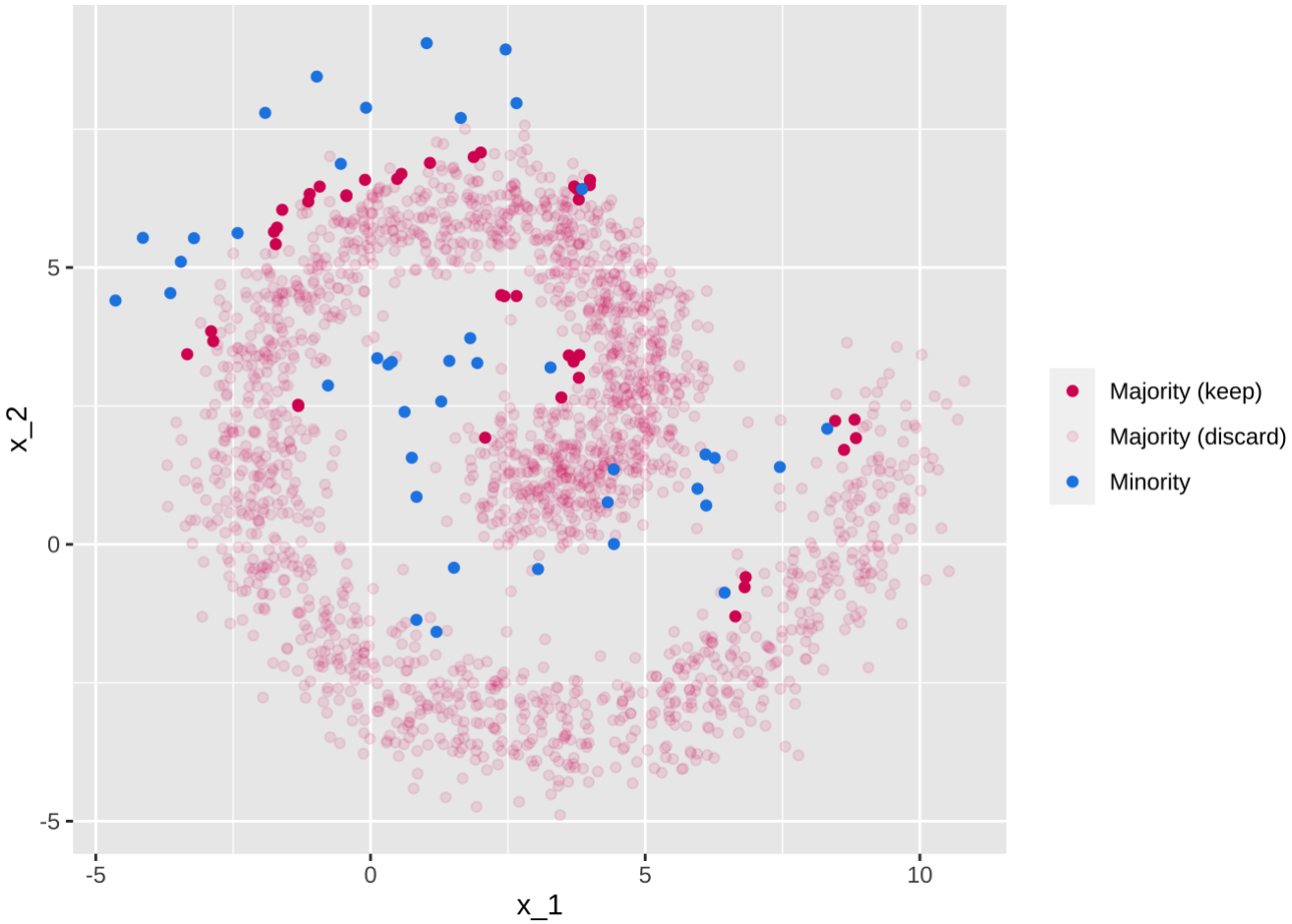
```
 [99] 0.6854008 0.6541362 0.6110025 0.7723911 0.7979115 0.4106465 0.5620973
[106] 0.5556624 1.3393927 0.9559875 1.1287170 1.2407578 1.4899234 1.4645065
[113] 1.5967705 1.6267947 1.5833263 1.1169942 1.1615146 1.0480362 1.1691950
[120] 1.1952582 1.3290517 1.1871077 1.6497306 1.2593037 1.0344329 1.2972478
[127] 1.3719742 1.3961989 1.4833165 1.2795937 1.1726010
```

Let us order these distances from the farthest to the closets, and then keep only the first observations:

```
ind_to_keep_3_neighbors <- order(avg_dist_3, decreasing = TRUE)[1:n_to_keep]
```

▶ Show the R codes

Observations in the training sample after NearMiss-3 is applied



Other techniques

A broader overview about the different undersampling techniques that rely on the KNN algorithm can be found in Beckmann et al. (2015)

Over-sampling

Instead of sampling from the majority class to reduce the number of examples from the majority class, it is possible to consider over-sampling from the minority examples.

SMOTE

A way of over-sampling the data is to create synthetic observations from the minority class. SMOTE (Chawla et al. 2002) (*S*ynthetic *M*inority *O*versampling *Te*chnique) is a popular algorithm to do so. Let us first present how this technique works. Then, we can explore some of its limits.

In a nutshell:

- we decide a proportion α of minority class to reach after the oversampling method is applied
- for each observation from the minority class, synthetic data are created, based on the characteristics from the nearest neighbors.

There are thus two parameters that need to be set prior the technique:

- the proportion of minority class to reach
- the number of nearest neighbors to consider.

To create a synthetic individual for one example x from the minority class, the algorithm works as follows:

1. Randomly select one of the k-nearest neighbors: x'
2. For each of the $j = 1, \ldots, p$ characteristics:
    - compute de distance between the caracteristic x'_j of the neighbor and that of the indivual of interest x'_j, *i.e.*, $x'_j - x'_j$
    - draw a number γ_j from a uniform distribution $\mathcal{U}[0, 1]$
    - multiply γ_j and $x'_j - x'_j$ to obtain the j-th coordinate of the synthetic individual
3. Repeat steps 1 and 2 as many time as necessary to populate for the individual of interest.

> **Tip**
>
> An illustration of the creation of synthetic data is provided a bit further.

Let us do it with R, from scratch:

```
# Number of minority class sample
T <- sum(df_train$y==1)
# Amount of SMOTE N%
perc_over = 300

if(perc_over < 100){
  # randomize the minority class samples as only a random percent of them will be SMOT
  T <- (perc_over/100) * T
  perc_over <- 100
}
perc_over <- perc_over/100
perc_over
```

```
[1] 3
```

If we want 300% more individuals from the minority class, then, for each example we need to create 3 synthetic observations.

Let us select a number of nearest neighbors:

```
k <- 5
```

With our spiral data from earlier, we have $p = 2$ characteristics:

```
num_attrs <- 2
```

Let us identify the k-nearest neighbors thanks to the `knn.index()` function from {FNN}:

```
# Compute the k-nearest neighbors for each minority class sample only
minority_sample <- df_train %>% filter(y == 1) %>% select(x_1, x_2) %>% as.matrix()
# Index of the k-nearest neighbors
k_nearest_neighbors <- FNN::knn.index(minority_sample, k = k)
head(k_nearest_neighbors)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]   21    8   22   37   31
[2,]    9   26   20   24   39
[3,]   17   31    5    8   40
[4,]   19   35   13   15   27
[5,]   37   31   17    8   10
[6,]   12   34   28   25    7
```

We need to initialize a matrix that will contain our observations after the resampling.

```
synthetic_sample <- matrix(ncol = num_attrs, nrow = perc_over*T)
```

Then, we can implement the algorithm this way:

```
for(indiv in 1:nrow(minority_sample)){
  # Populate  for individual `indiv` #
  synthetic_sample_i <- matrix(NA, ncol = num_attrs, nrow = perc_over)
  for(j in 1:perc_over){
    # Chosing one neighbor
    nn <- sample(seq(1, k), size = 1)
    # Generate `perc_over` synthethic observations
    for(attr in 1:num_attrs){
      # For each attribute
      dif <- minority_sample[k_nearest_neighbors[indiv,nn], attr] - minority_sample[in
      gap <- runif(0, 1, n=1)
      synthetic_sample_i[j, attr] <- minority_sample[indiv,attr] + gap*dif
    }
  }
  synthetic_sample[(indiv*perc_over-perc_over+1):(indiv*perc_over),] <- synthetic_samp
}
colnames(synthetic_sample) <- colnames(minority_sample)
head(synthetic_sample)
```

```
            x_1      x_2
[1,]  0.30362151 3.294152
[2,]  0.01705773 3.338812
[3,]  0.33513034 3.321411
[4,] -2.90275216 7.208939
[5,] -2.41632541 6.276632
[6,] -2.16788309 6.804184
```

The new individuals can be put in a tibble:

```
df_synthetic <-
  as_tibble(synthetic_sample) %>%
  mutate(y = 1, colour = "red") %>%
  mutate(y = factor(y, levels = c(0, 1)))
df_synthetic
```

```
# A tibble: 123 × 4
       x_1    x_2 y     colour
     <dbl>  <dbl> <fct> <chr>
 1  0.304   3.29  1     red
 2  0.0171  3.34  1     red
 3  0.335   3.32  1     red
 4 -2.90    7.21  1     red
 5 -2.42    6.28  1     red
 6 -2.17    6.80  1     red
 7  1.86    3.52  1     red
 8  1.87    3.66  1     red
 9  1.85    3.50  1     red
10  3.17   -0.408 1     red
# … with 113 more rows
```

Let us put the training data and the synthetic data in a single tibble:

```
df_train_with_synthetic <-
  df %>% mutate(type = "observed") %>%
  bind_rows(df_synthetic %>% mutate(type = "synthetic"))
```

If we look at how many observations are now in the new dataset:

```
table(df_train_with_synthetic$y)
```

```
   0    1
2450  173
```

And express as proportions:

```
round(prop.table(table(df_train_with_synthetic$y)), 2)
```

```
   0    1
0.93 0.07
```

Recall that the proportions in the training set were:

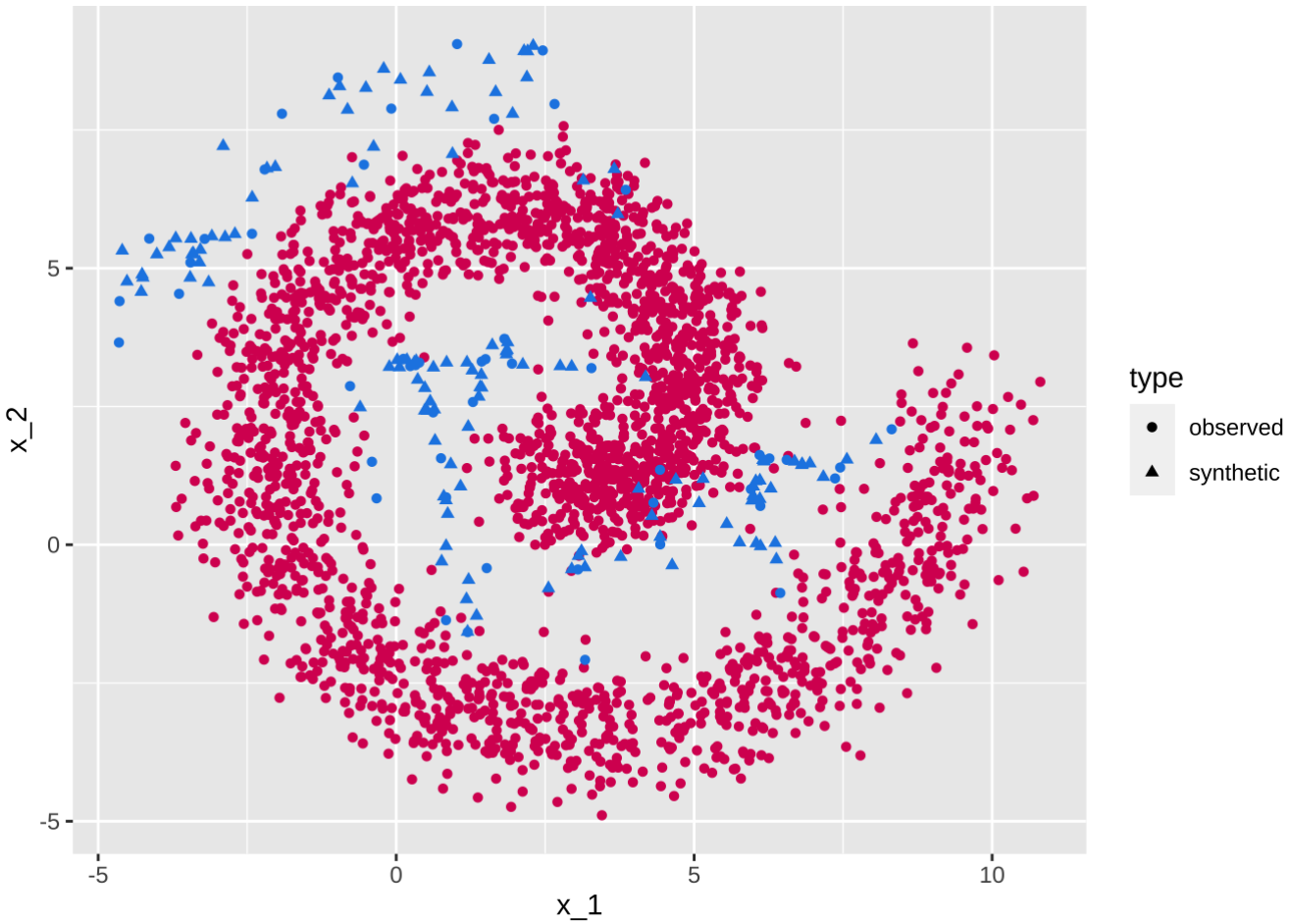
```
round(prop.table(table(df_train$y)), 2)
```

```
   0    1
0.98 0.02
```

Let us visualize the new training set:

```
ggplot(data = df_train_with_synthetic,
       mapping = aes(x = x_1, y = x_2, colour = y, shape = type)) +
  geom_point() +
  scale_colour_manual(NULL, values = c("0" = "#D81B60", "1" = "#1E88E5"), guide = "non
```

Training set with synthetic data obtained after the SMOTE technique



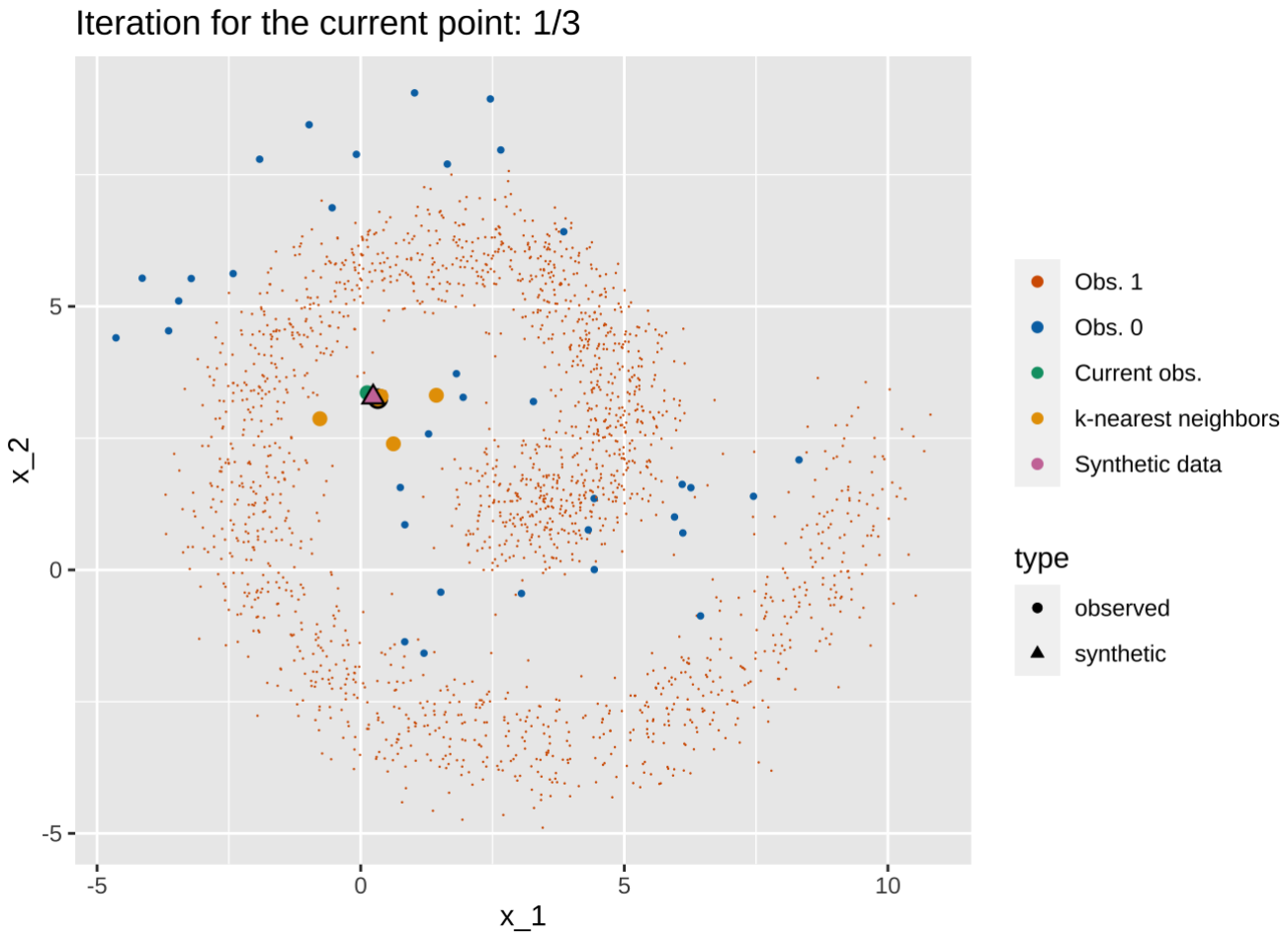
Let us go back to how the synthetic data are obtained, using an illustration.

▶ Show the R codes

The green point represents the current individual from the minority class. We will generate synthetic data using its k nearest neighbors. Among them, one (circled in black) has been randomly picked. A random fraction of the distance from the j-th characteristic of that neighbor to that of the current point has been computed, for each of the $j = 1, 2$ characteriscits (`x_1` and `x_2`). The resulting synthetic observation is represented by a purple triangle with a black contour.

▶ Show the R codes

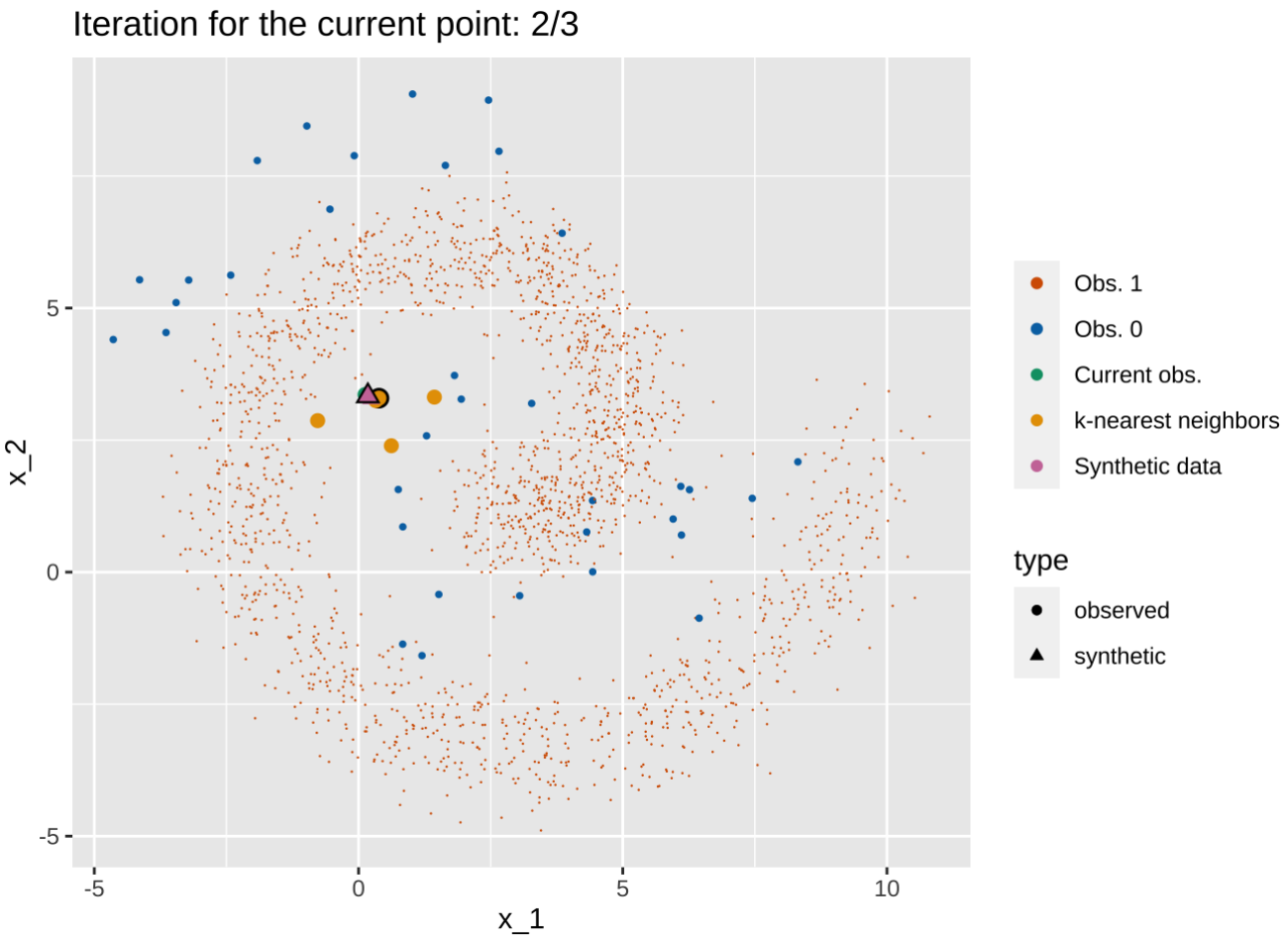
First synthetic observations obtained from a first point from the minority class



Recall that we set `perc_over` to 300. For each observation, we thus generate 3 new points.

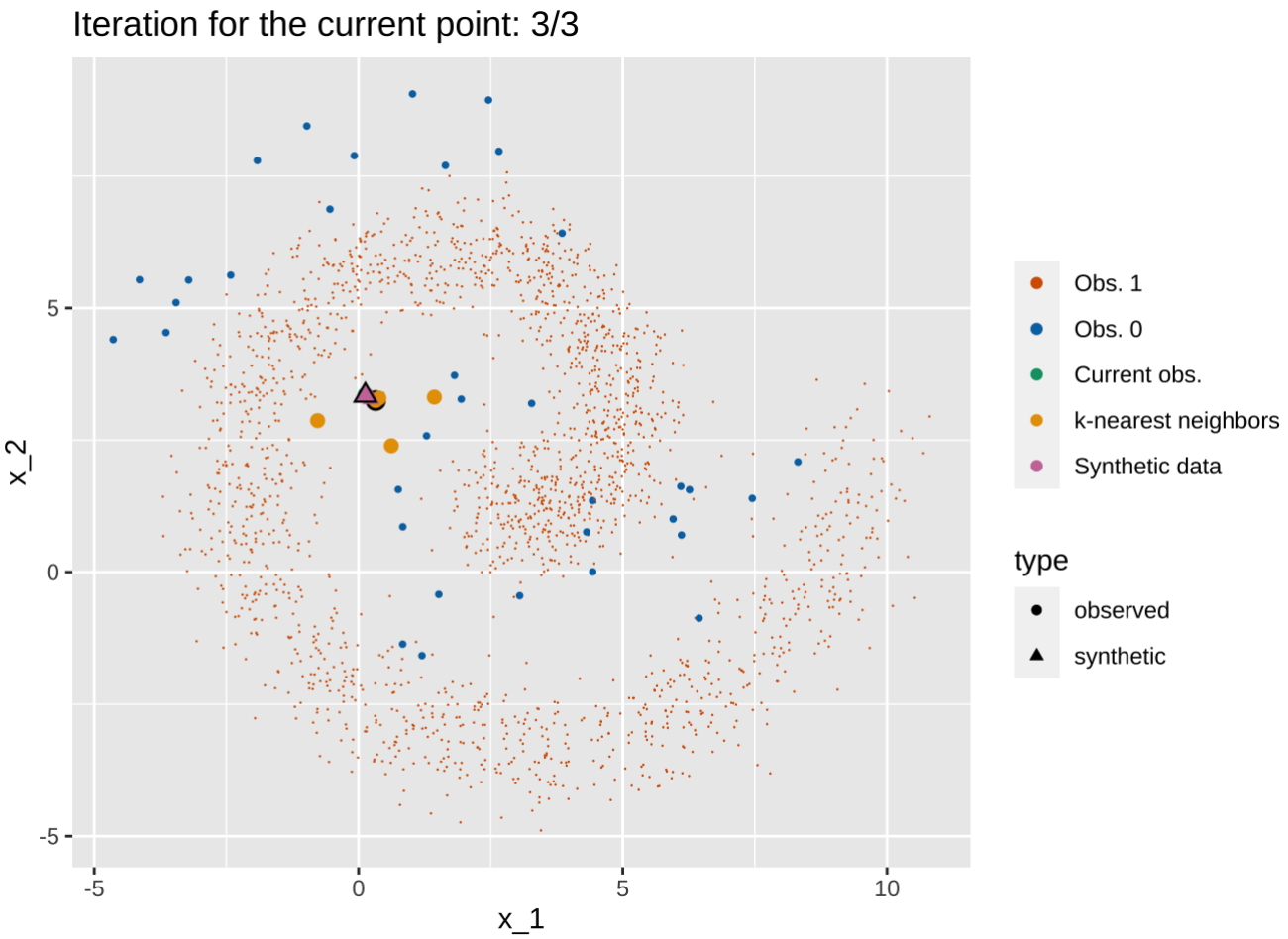
▶ Show the R codes

Second synthetic observations obtained from a first point from the minority class



▶ Show the R codes

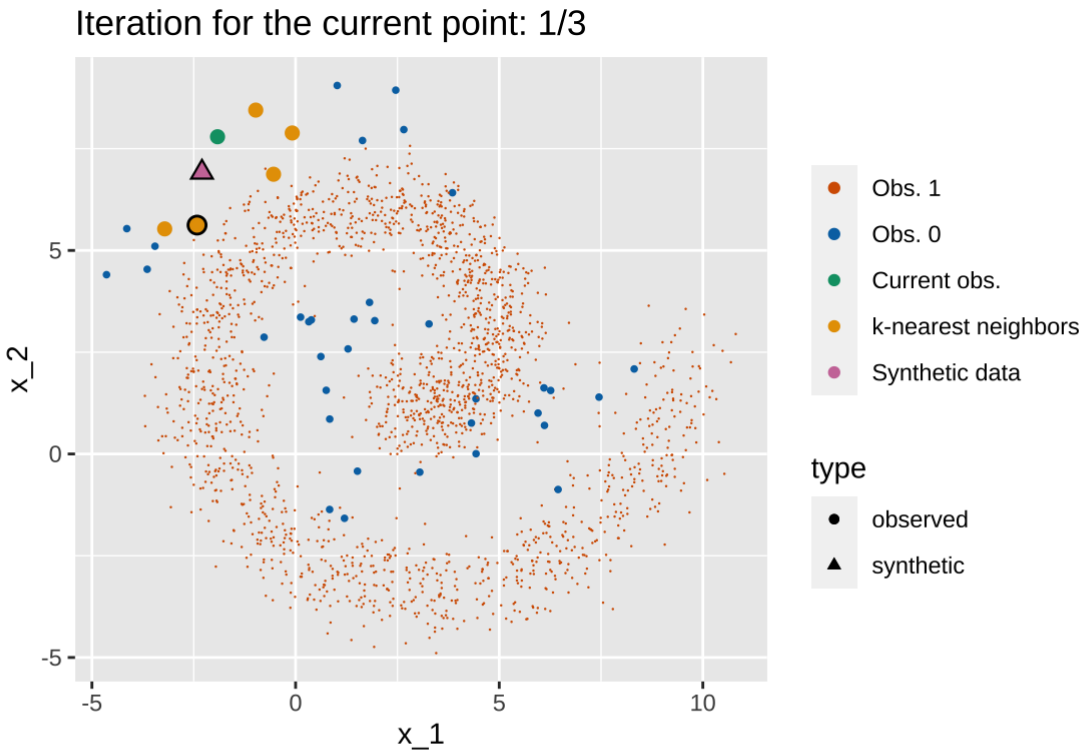
Third synthetic observations obtained from a first point from the minority class

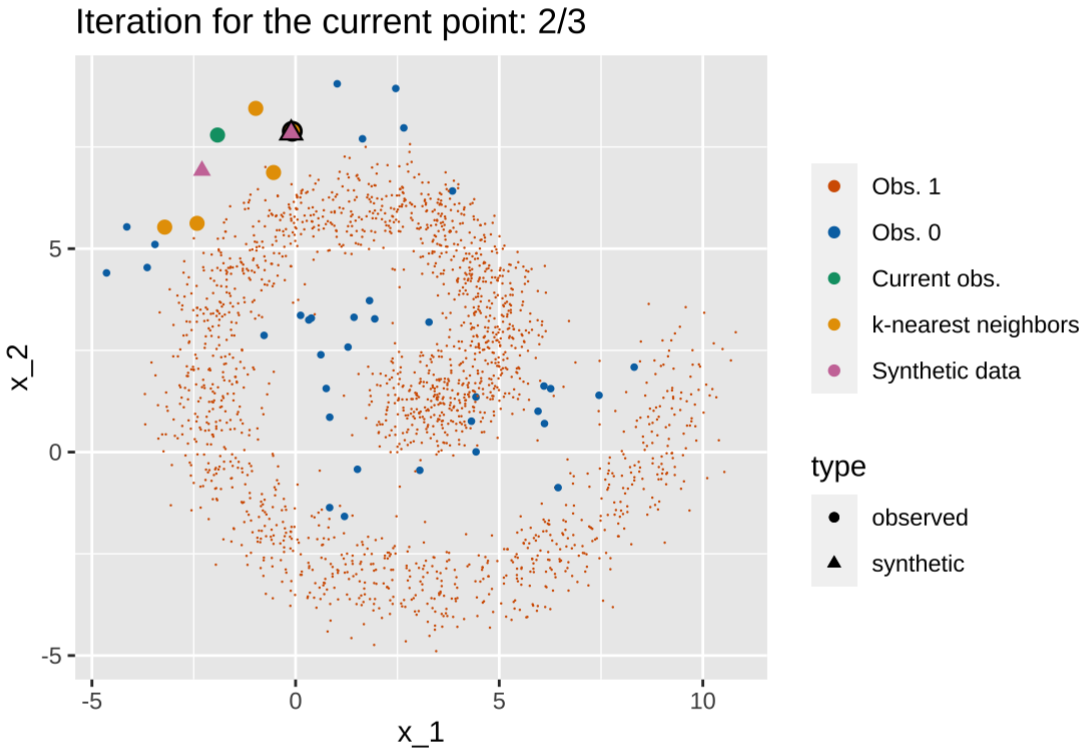


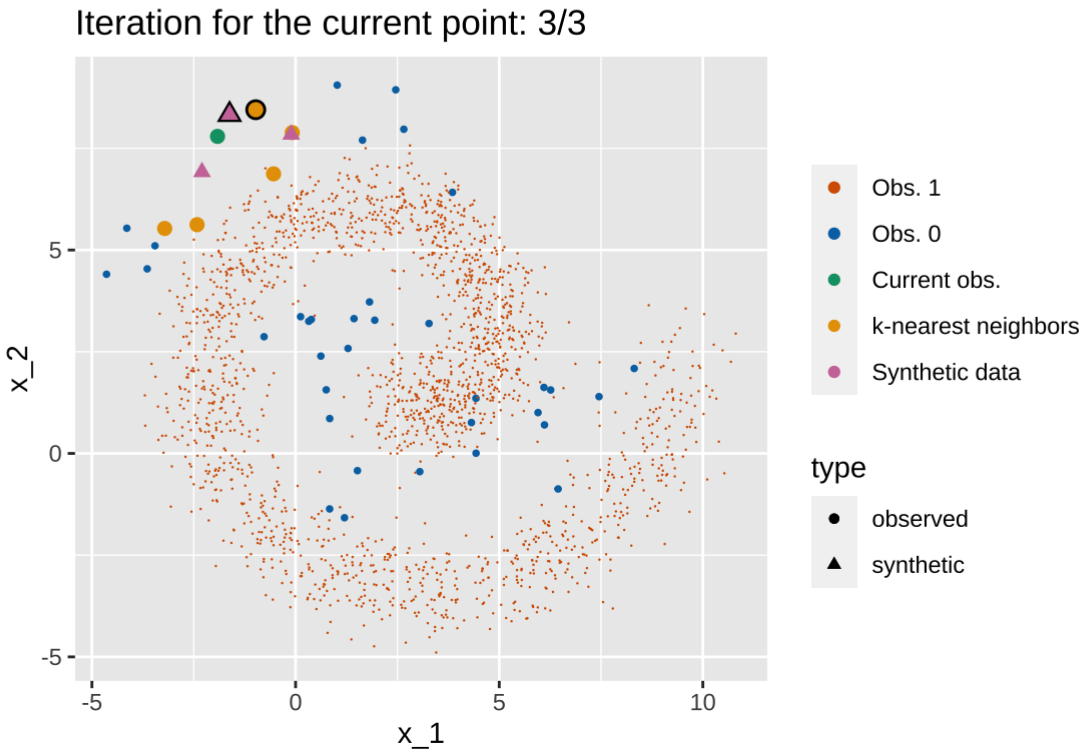
Then, we can turn to another point from the minority class.

▶ Show the R codes

Synthetic observations from a second point from the minority class







Built-in method with R

In R, we can use the `SMOTE()` function from {smotefamily}. It generates synthetic data from the minority class so as to get a perfectly balanced data at the end of the process.

```
gen_data_train <-
  smotefamily::SMOTE(df_train %>% select(-y),
                     target = df_train$y,
                     K = 4)
```

```
df_train_with_synthetic_2 <-
  df_train %>%
  mutate(type = "observed") %>%
  bind_rows(gen_data_train$syn_data %>%
              rename(y = class) %>%
              mutate(type = "synthetic"))
df_train_with_synthetic_2
```

```
# A tibble: 3,886 × 4
      x_1   x_2 y     type
    <dbl> <dbl> <chr> <chr>
 1  5.63   4.13 0     observed
 2  0.122  3.36 1     observed
 3 -0.120  4.57 0     observed
 4  9.24   1.52 0     observed
 5  4.06   5.21 0     observed
 6  3.73   1.55 0     observed
 7 -0.565  4.08 0     observed
 8  2.25  -3.32 0     observed
 9  6.37  -2.99 0     observed
10 -0.349  5.83 0     observed
# … with 3,876 more rows
```

If we look at the proportion of each class in the dataset:

```
prop.table(table(class = df_train_with_synthetic_2$y,
                 type = df_train_with_synthetic_2$type))
```

```
     type
class   observed  synthetic
    0 0.50411734 0.00000000
    1 0.01055069 0.48533196
```

Varying the number of k-nearest neighbors

Let us vary the number of k-nearest neighbors. We can create a small function that creates a balanced dataset using SMOTE given some number `k` of nearest numbers, then estimate the random forest, and returns the predicted probabilities in the test sample.

```
#' SMOTE with `k` nearest neighbors
#' Then train a random forest on the SMOTEd data
#' And make the prediction on the test data
#' @param k number of nearest numbers
get_pred_k <- function(k){
  gen_data_train <-
    smotefamily::SMOTE(df_train %>% select(-y),
                       target = df_train$y,
                       K = k)
  df_train_with_synthetic_2 <-
    df_train %>%
    mutate(type = "observed") %>%
    bind_rows(gen_data_train$syn_data %>%
```

```
                rename(y = class) %>%
                mutate(type = "synthetic")) %>%
    mutate(y = factor(y, levels = c("0", "1")))

  mod_tmp_synthetic <-
    randomForest(
      formula = y ~ x_1+x_2,
      data = df_train_with_synthetic_2,
      ntree = 200,
      mtry =2 ,
      nodesize = 5,
      maxnodes = NULL
    )
  pred_prob_with_synth <- predict(mod_tmp_synthetic, newdata = df_test, type = "prob")
  tibble(observed = df_test$y, pred = pred_prob_with_synth, k = k)
}
```

Let us consider the following values for `k`: 5, 10, 15, and 20. (Here, we only have 41 observations from the minority class, so it might not be a good idea to consider too many neighbors.)

```
predictions_with_smote <-
  map_df(c(5, 10, 15, 20), get_pred_k)
predictions_with_smote
```

```
# A tibble: 2,000 × 3
   observed  pred     k
   <fct>    <dbl> <dbl>
 1 0        0         5
 2 0        0         5
 3 0        0         5
 4 0        0         5
 5 0        0         5
 6 0        0         5
 7 0        0.045     5
 8 0        0         5
 9 0        0.045     5
10 0        0         5
# … with 1,990 more rows
```

For comparison, let us estimate a random forest on the unbalanced dataset:

```
mod_without_synthetic <-
  randomForest(
    formula = y ~ x_1+x_2,
    data = df_train,
    ntree = 200,
    mtry =2 ,
    nodesize = 5,
    maxnodes = NULL
  )
pred_prob_without_synth <- predict(mod_without_synthetic, newdata = df_test, type = "p
```

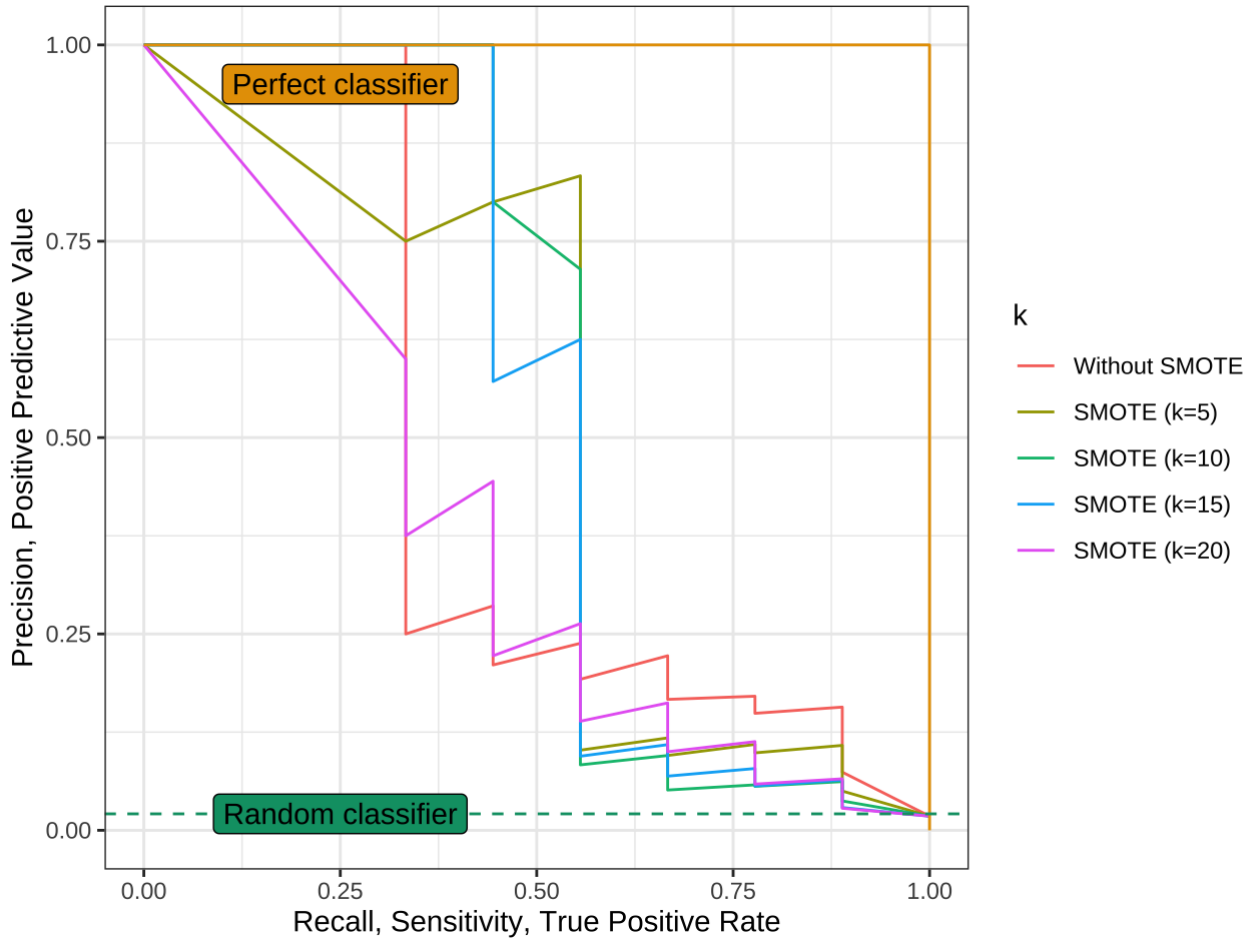
For each of the different models estimated (depending on the number of neighbors, and on the imbalanced dataset), let us compute the precision and sensitivity for varying values of the probability threshold of the classifier.

```
precision_recall <-
  tibble(observed = df_test$y, pred = pred_prob_without_synth, k = -1) %>%
  bind_rows(predictions_with_smote) %>%
  group_by(k) %>%
  pr_curve(truth = observed, pred, event_level="second")
```

We can plot the precision-recall curve:

```
precision_recall %>%
  autoplot() +
  geom_line(tibble(recall = c(0,0.9999,1), precision = c(1,1,0)),
            mapping = aes(x = recall, y = precision),
            colour = "#E69F00") +
  geom_hline(yintercept = sum(df_train$y==1) / sum(df_train$y==0), linetype = "dashed"
  geom_label(data = tibble(x = c(.25, 0.25), y = c(sum(df_train$y==1) / sum(df_train$y
                           lab = c("Random classifier", "Perfect classifier")),
             mapping = aes(x=x, y=y, label=lab, fill = lab)) +
  scale_fill_manual(NULL, guide = "none",
                    values = c("Random classifier" = "#009E73",
                               "Perfect classifier" = "#E69F00")) +
  scale_colour_discrete(labels = c("-1" = "Without SMOTE", "5" = "SMOTE (k=5)",
                                   "10" = "SMOTE (k=10)", "15" = "SMOTE (k=15)",
                                   "20" = "SMOTE (k=20)")) +
  coord_equal() +
  labs(x = "Recall, Sensitivity, True Positive Rate",
       y = "Precision, Positive Predictive Value")
```

Precision-recall for the different random forests, test data, varying number of neighbords considered when using SMOTE



If we use the F1-Score to get the "optimal" threshold for each model, these are the values we obtain:

```
optimal_thresholds <-
  precision_recall %>%
  mutate(F1_score = 2*precision*recall/ (precision+recall)) %>%
  arrange(desc(F1_score)) %>%
  slice(1)
optimal_thresholds
```

```
# A tibble: 5 × 5
# Groups:   k [5]
      k .threshold recall precision F1_score
  <dbl>      <dbl>  <dbl>     <dbl>    <dbl>
1    -1      0.555  0.333     1        0.5
2     5      0.97   0.556     0.833    0.667
3    10      0.99   0.556     0.714    0.625
4    15      1      0.444     1        0.615
5    20      0.98   0.444     0.444    0.444
```

We can compute the AUC of the ROC curve, for each model:

```
tibble(observed = df_test$y, pred = pred_prob_without_synth, k = -1) %>%
  bind_rows(predictions_with_smote) %>%
  group_by(k) %>%
```

```
  group_by(k) %>%
  roc_auc(truth = observed, pred, event_level="second") %>%
  arrange(desc(.estimate))
```

```
# A tibble: 5 × 4
      k .metric .estimator .estimate
  <dbl> <chr>   <chr>          <dbl>
1    -1 roc_auc binary         0.906
2     5 roc_auc binary         0.889
3    20 roc_auc binary         0.863
4    15 roc_auc binary         0.858
5    10 roc_auc binary         0.856
```

As well as the AUC for the Precision-recall curve:

```
auc_pr_all <-
  tibble(observed = df_test$y, pred = pred_prob_without_synth, k = -1) %>%
  bind_rows(predictions_with_smote) %>%
  group_by(k) %>%
  pr_auc(truth = observed, pred, event_level="second") %>%
  arrange(desc(.estimate))
auc_pr_all
```

```
# A tibble: 5 × 4
      k .metric .estimator .estimate
  <dbl> <chr>   <chr>          <dbl>
1    10 pr_auc  binary         0.554
2    15 pr_auc  binary         0.540
3     5 pr_auc  binary         0.507
4    -1 pr_auc  binary         0.452
5    20 pr_auc  binary         0.377
```

Let us use the “optimal” threshold of each model and the predicted probabilities to predict the class.

```
predicted_class_df_test <-
  tibble(observed = df_test$y, pred = pred_prob_without_synth, k = -1) %>%
  bind_rows(predictions_with_smote) %>%
  left_join(
    optimal_thresholds %>% select(k, .threshold)
  ) %>%
  mutate(pred_class = ifelse(pred > .threshold, "1", "0"),
         pred_class = factor(pred_class, levels = c("0", "1")))
predicted_class_df_test
```

```
# A tibble: 2,500 × 5
   observed  pred      k .threshold pred_class
   <fct>    <dbl> <dbl>      <dbl> <fct>
 1 0        0        -1      0.555 0
 2 0        0        -1      0.555 0
 3 0        0        -1      0.555 0
 4 0        0.02     -1      0.555 0
 5 0        0        -1      0.555 0
 6 0        0.115    -1      0.555 0
 7 0        0        -1      0.555 0
```

```
 8 0        0        -1      0.555 0
 9 0        0.005    -1      0.555 0
10 0        0        -1      0.555 0
# … with 2,490 more rows
```

For each model, we can create the confusion table and compute different metrics:

```
confusion_tables <-
  predicted_class_df_test %>%
  group_by(k) %>%
  nest() %>%
  mutate(confusion_table =  map(.x = data,
      .f = ~caret::confusionMatrix(data = .x$pred_class,
                                   reference =  .x$observed, positive = "1")))
```

The confusion table for the model without SMOTE, on the test sample:

```
confusion_tables %>% filter(k==-1) %>% .$confusion_table
```

```
[[1]]
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 491   7
         1   0   2

               Accuracy : 0.986
                 95% CI : (0.9714, 0.9944)
    No Information Rate : 0.982
    P-Value [Acc > NIR] : 0.32177

                  Kappa : 0.3594

 Mcnemar's Test P-Value : 0.02334

            Sensitivity : 0.2222
            Specificity : 1.0000
         Pos Pred Value : 1.0000
         Neg Pred Value : 0.9859
             Prevalence : 0.0180
         Detection Rate : 0.0040
   Detection Prevalence : 0.0040
      Balanced Accuracy : 0.6111

       'Positive' Class : 1
```

And on the model with SMOTE (with k=10`), on the same test sample:

```
confusion_tables %>% filter(k==auc_pr_all$k[1]) %>% .$confusion_table
```

```
[[1]]
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 490   5
         1   1   4

               Accuracy : 0.988
                 95% CI : (0.9741, 0.9956)
    No Information Rate : 0.982
    P-Value [Acc > NIR] : 0.2043

                  Kappa : 0.5658

 Mcnemar's Test P-Value : 0.2207

            Sensitivity : 0.4444
            Specificity : 0.9980
         Pos Pred Value : 0.8000
         Neg Pred Value : 0.9899
             Prevalence : 0.0180
         Detection Rate : 0.0080
   Detection Prevalence : 0.0100
      Balanced Accuracy : 0.7212

       'Positive' Class : 1
```

We seem to be able to better predict the minority class. However, keep in mind that this is a single example. We would need to replicate the whole process a great number of time to have a clue whether or not, on average, with the data at hand, SMOTE is actually helpful.

Note

Notes:

- If you want to go further, you should also try to look at what happens when you use SMOTE on the test set as well. In practice, you should not do so, because you will make predictions for individuals that live in a space where there might actually never be any observations from the minority class.
- When identifying the k-nearest neighbors, the results can be greatly affected if the scale of the different features are large. It may be best to think about normalize the data prior applying the SMOTE technique.

Further comments and readings

Distances with categorical variables

If your dataset includes categorical variables, the distance between two observations is a bit changed. While it would be possible to transform your categorical variable into a set of binary variables (one-hot encoding or dummy encoding), it may not be the best idea.

variables (one-hot encoding or dummy encoding), it may not be the best idea.

Consider the following situation. Assume we have one feature with numerical values and a categorical variable that takes three values: A, B, and C. Let us further assume that we have created two dummy variables: one for class A, and another for class B (class C thus being the reference class). Consider the two examples:

- x, for which the numerical feature is .1 and who belongs to class A: $x = [.1 \quad 1 \quad 0]$
- x', for which the numerical feature is .2 and who belongs to class B: $x' = [.2 \quad 1 \quad 0]$

The Eucledean distance between x and x' this writes:

$$\sqrt{(.1-.2)^2 + (1-0)^2 + (0-1)^2}.$$

The contribution of the categorical variable to the distance is thus $(1-0)^2 + (0-1)^2 = 2$. If the numerical variable has been scaled to get values between 0 and 1, the maximum contribution for that variable is 1. Here, with only 3 classes for the categorical variable, the contribution of that variable to the distance is twice the maximum contribution of the numerical variable. If the number of classes for the categorical variable is higher, the contribution of that variable to the distance will be even higher if we create dummy variables to encode the categorical variable.

To overcome this issue, it is possible to use a different distance measure. Gower's distance (Gower 1971) can be a good solution when the dataset contains categorical variables. Here is how it works, to compute the distance between two observations x and x':

1. for each feature $j = 1, \ldots, p$, compute a score $s_j \in [0, 1]$
    - the closer x and x', the closer s_j to 1
    - the farther x and x', the closer s_j to 0
2. and set a weight δ_j:
    - if x and x' can be compared for their j-th feature, set a weigh $\delta_j = 1$
    - otherwise, in case of missing value, set $\delta_j = 0$
3. The Gower's distance between x and x' is then computed as follows:

$$S = \frac{\sum_{j=1}^{p} s_j \times \delta_j}{\sum_{j=1}^{p} \delta_j}$$

The score s_j depends on the type of the j-th feature:

- for a **quantitative** variable:

$$s_j = 1 - \frac{\mid x_j - x'_j \mid}{\max(\mathbb{x}_j) - \min(\mathbb{x}_j)},$$

  where $\max(\mathbb{x}_j)$ and $\min(\mathbb{x}_j)$ are the maximum and the minimum values of feature j in the training sample.
- for a **qualitative** variable:

$$s_j = \mathbb{1}_{\{x_j = x'_j\}}$$

.

> **Note**
>
> For over-sampling, the SMOTE-NC algorithm allows us to have categorical variables within the data. When creating synthetic data, nothing changes compared to SMOTE for numerical variables. For categorical variables, the value of the synthetic observation is set to the most common among the nearest neighbors.

SMOTE with dicrete numerical variables

If your dataset contains discrete numerical variables, using SMOTE will create synthetic individual in areas where no observation can **never** be found in the original data.

To overcome this issue, one could think of simply casting the variable to a categorical variable. But doing so would make us lose the idea of an order in the values of the variable. A better solution consists in discretizing the variable right after SMOTE.

For example, if the j-th feature is a discrete numerical variable that takes the following values: {1,2,3}, if the synthetic observation has a value of, let us say, 1.2, then we can simply round it to 1.

A nice survey on resampling techniques

A nice survey of the methods available as of 2016 is available in More (2016).

References

Beckmann, Marcelo, Nelson FF Ebecken, Beatriz SL Pires de Lima, et al. 2015. “A KNN Undersampling Approach for Data Balancing.” *Journal of Intelligent Learning Systems and Applications* 7 (04): 104.

Chawla, Nitesh V, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. “SMOTE: Synthetic Minority over-Sampling Technique.” *Journal of Artificial Intelligence Research* 16: 321–57.

Gower, John C. 1971. “A General Coefficient of Similarity and Some of Its Properties.” *Biometrics*, 857–71.

Mani, Inderjeet, and I Zhang. 2003. “kNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction.” In *Proceedings of Workshop on Learning from Imbalanced Datasets*, 126:1–7. ICML.

More, Ajinkya. 2016. “Survey of Resampling Techniques for Improving Classification Performance in Unbalanced Datasets.” *arXiv Preprint arXiv:1608.06048*.

Unal, Ilker. 2017. “Defining an Optimal Cut-Point Value in ROC Analysis: An Alternative Approach.” *Computational and Mathematical Methods in Medicine* 2017.