Unit 2 Lecture 4: Classification

September 26, 2023

In this R demo, we explore classification with imbalanced classes in the context of KNN applied to a dataset on credit card default.

Let's first load the tidyverse as well as the default data:

```
# load packages
library(tidyverse)
library(cowplot)    # for plot_grid()
library(stat471)    # for knn(), classification_metrics()

# load default data
default_data <- read_tsv("default.tsv", col_types = "ffdd")
default_data
```

```
## # A tibble: 10,000 x 4
##    default student balance income
##    <fct>   <fct>     <dbl>  <dbl>
##  1 No      No         730. 44362.
##  2 No      Yes        817. 12106.
##  3 No      No        1074. 31767.
##  4 No      No         529. 35704.
##  5 No      No         786. 38463.
##  6 No      Yes        920.  7492.
##  7 No      No         826. 24905.
##  8 No      Yes        809. 17600.
##  9 No      No        1161. 37469.
## 10 No      No           0  29275.
## # i 9,990 more rows
```

Data exploration and visualization

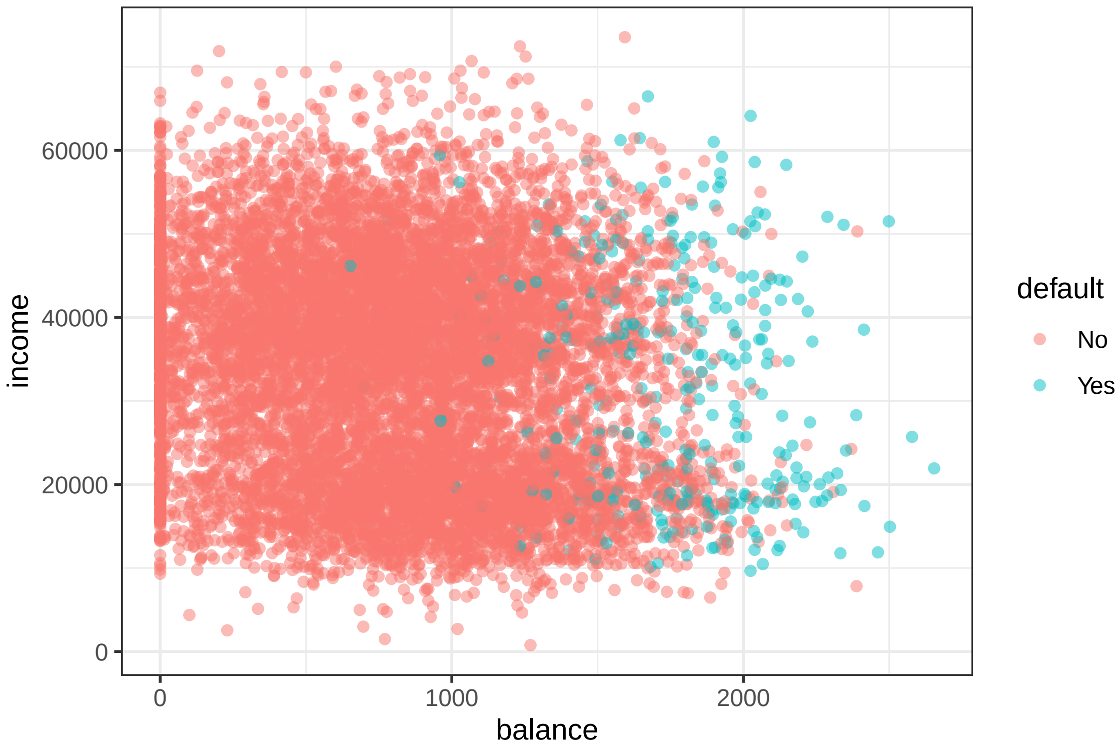
Let's take a look at the default rate in these data:

```
default_data |>
  summarize(default_rate = mean(default == "Yes"))
```

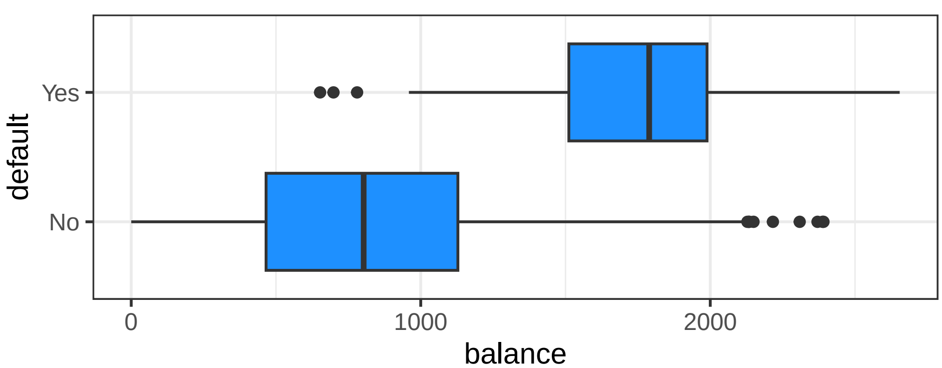
```
## # A tibble: 1 x 1
##   default_rate
##          <dbl>
## 1       0.0333
```

Uh-oh! It looks like we have imbalanced classes.

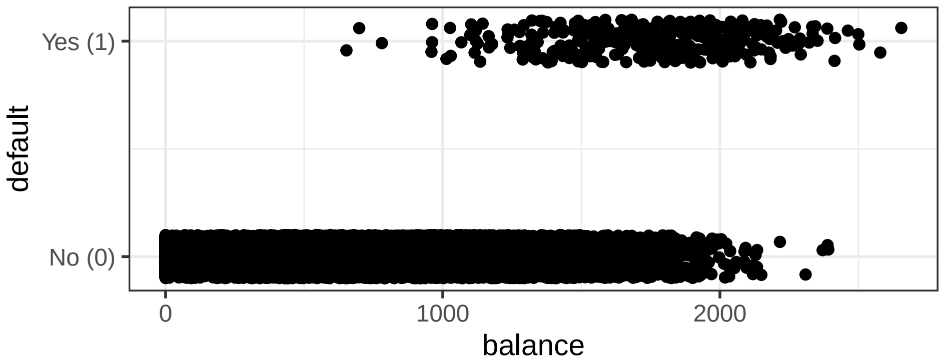
```
# visualize default as a function of `balance` and `income`
default_data |>
  ggplot(aes(x = balance, y = income, colour = default)) +
  geom_point(alpha = 0.5)
```




```
# visualize default as a function of just `balance`
default_data |>
  ggplot(aes(x = balance, y = default)) +
  geom_boxplot(fill = "dodgerblue")
```




```
# another useful visualization of default versus balance is the jitter plot
default_data |>
  ggplot(aes(x = balance, y = as.numeric(default)-1)) +
  geom_jitter(height = 0.1) +
  scale_y_continuous(breaks = c(0,1), labels = c("No (0)", "Yes (1)")) +
  labs(y = "default")
```



K-nearest neighbors classification

Next, split the observations 50%/50% into training and testing (we won't be doing cross-validation today for the sake of time, though in principle we could).

```
set.seed(47102023)                     # set seed for reproducibility
n <- nrow(default_data)
train_samples <- sample(1:n, n/2)       # list of rows to be used for training
default_train <- default_data |>
  filter(row_number() %in% train_samples)
default_test <- default_data |>
  filter(!(row_number() %in% train_samples))
```

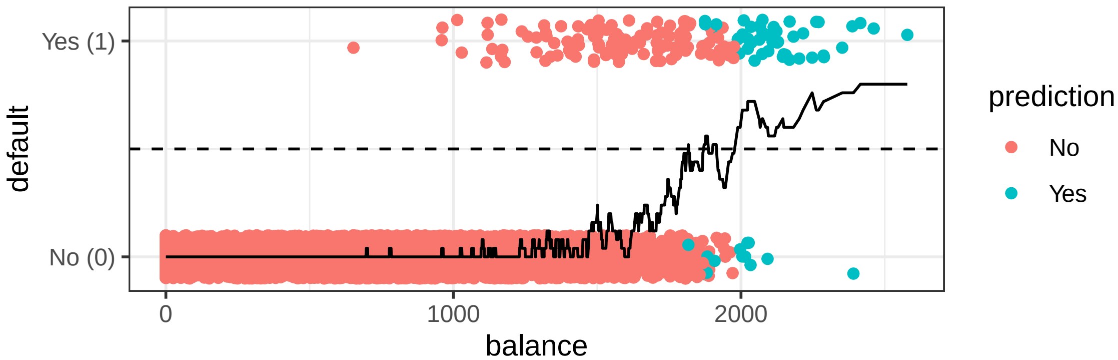
Actually, perhaps we can stratify on `default` before splitting, since we have imbalanced classes:

```
set.seed(47102023)                     # set seed for reproducibility
train_samples <- default_data |>
  mutate(rownum = row_number()) |>
  slice_sample(prop = 0.5, by = default) |>
  pull(rownum)
default_train <- default_data |>
  filter(row_number() %in% train_samples)
default_test <- default_data |>
  filter(!(row_number() %in% train_samples))
```

Next, let's apply KNN with K = 25, using just `balance` as a feature.

```
# apply KNN with K = 25
knn_results <- knn(default ~ balance,
                  training_data = default_train,
                  test_data = default_test,
                  k = 25)

# let's add the predictions and probabilities to the test data
predictions <- knn_results$predictions
probabilities <- knn_results$probabilities |>
  filter(class == "Yes") |>
  pull(probability)
default_test_with_pred <- default_test |>
    mutate(prediction = predictions,
           probability = probabilities)

# visualize the KNN classification
```

```
jitter_plot_with_pred <- default_test_with_pred |>
  ggplot(aes(x = balance)) +
  geom_jitter(aes(y = as.numeric(default)-1, colour = prediction), height = 0.1) +
  geom_line(aes(y = probability)) +
  geom_hline(yintercept = 0.5, linetype = "dashed") +
  scale_y_continuous(breaks = c(0,1), labels = c("No (0)", "Yes (1)")) +
  labs(y = "default")
jitter_plot_with_pred
```



Next let's compute a few performance metrics:

```
# compute misclassification error
default_test_with_pred |>
  summarize(misclassification_error = mean(default != prediction))
```

```
## # A tibble: 1 x 1
##   misclassification_error
##                     <dbl>
## 1                  0.0264
```

```
# calculate the confusion matrix
conf_matrix <- default_test_with_pred |>
  select(default, prediction) |>
  table()
conf_matrix
```

```
##        prediction
## default   No  Yes
##     No  4821   13
##     Yes  119   48
```

```
# calculate precision
TP <- conf_matrix["Yes", "Yes"]
FP <- conf_matrix["No", "Yes"]
precision <- TP / (TP + FP)
precision
```

```
## [1] 0.7868852
```

```
# calculate recall
TP <- conf_matrix["Yes", "Yes"]
FN <- conf_matrix["Yes", "No"]
recall <- TP / (TP + FN)
recall
```

```
## [1] 0.2874251
```

```r
# calculate the F-score
1/(mean(c(1/precision, 1/recall)))
```

```
## [1] 0.4210526
```

```r
# introduce costs associated with misclassifications and computed weighted
#  misclassification error
C_FN <- 2000
C_FP <- 150
default_test_with_pred |>
  summarize(weighted_error = mean(C_FP*(prediction == "Yes" & default == "No") +
                                  C_FN*(prediction == "No" & default == "Yes")))
```

```
## # A tibble: 1 x 1
##   weighted_error
##            <dbl>
## 1           48.0
```

A convenient way to calculate all these metrics at once is the `classification_metrics()` function from the `stat471` package.

```r
classification_metrics(
  test_responses = default_test$default,
  test_predictions = knn_results$predictions,
  C_FP = C_FP,
  C_FN = C_FN
)
```

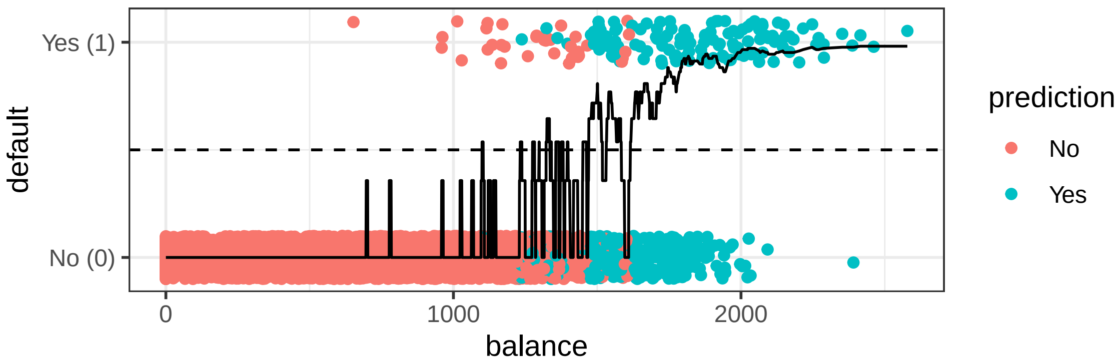
```
## # A tibble: 1 x 5
##   misclass_err w_misclass_err precision recall     F
##          <dbl>          <dbl>     <dbl>  <dbl> <dbl>
## 1       0.0264           48.0     0.787  0.287 0.421
```

Weighted K-nearest neighbors classification

Let's see how the picture changes when we apply KNN with observation weights dictated by the misclassification costs defined above.

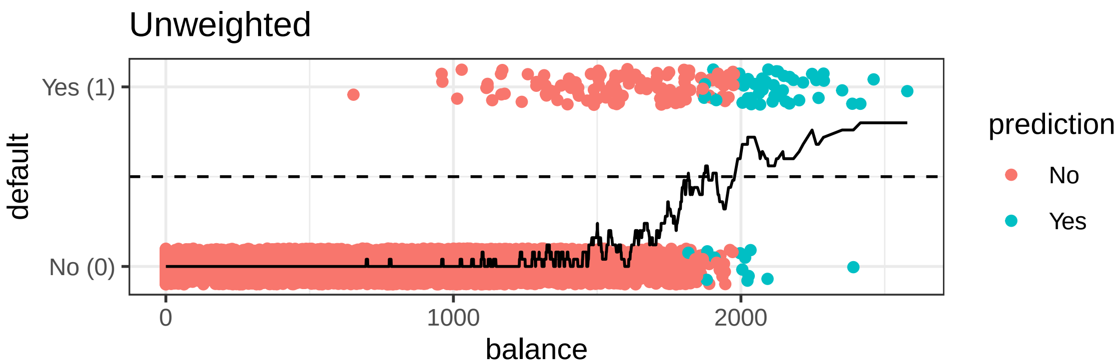
```r
# define weight vector based on misclassification costs
weights <- C_FP * (default_train$default == "No") +
  C_FN * (default_train$default == "Yes")

# rerun KNN with weights
knn_results_weighted <- knn(default ~ balance,
                  training_data = default_train,
                  test_data = default_test,
                  weights = weights,
                  k = 25)

# let's add the predictions and probabilities to the test data
predictions <- knn_results_weighted$predictions
probabilities <- knn_results_weighted$probabilities |>
  filter(class == "Yes") |>
  pull(probability)
default_test_with_pred_weighted <- default_test |>
```

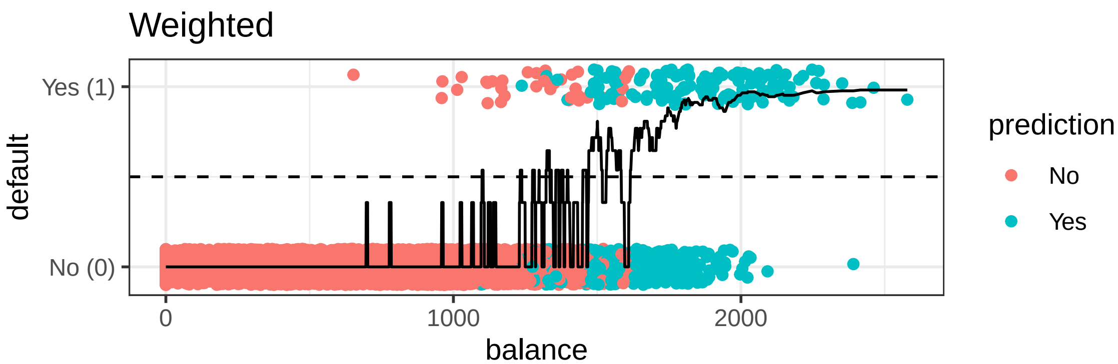
```r
    mutate(prediction = predictions,
           probability = probabilities)

# visualize the KNN classification
jitter_plot_with_pred_weighted <- default_test_with_pred_weighted |>
  ggplot(aes(x = balance)) +
  geom_jitter(aes(y = as.numeric(default)-1, colour = prediction), height = 0.1) +
  geom_line(aes(y = probability)) +
  geom_hline(yintercept = 0.5, linetype = "dashed") +
  scale_y_continuous(breaks = c(0,1), labels = c("No (0)", "Yes (1)")) +
  labs(y = "default")
jitter_plot_with_pred_weighted
```



Let's compare this plot with its unweighted counterpart:

```r
plot_grid(
  jitter_plot_with_pred + ggtitle("Unweighted"),
  jitter_plot_with_pred_weighted + ggtitle("Weighted"),
  ncol = 1
)
```





We see that the predicted probabilities increased due to the upweighting of the positive cases, causing more of the test data points to be predicted as positive.

Let's recompute the metrics and compare the weighted and unweighted fits:

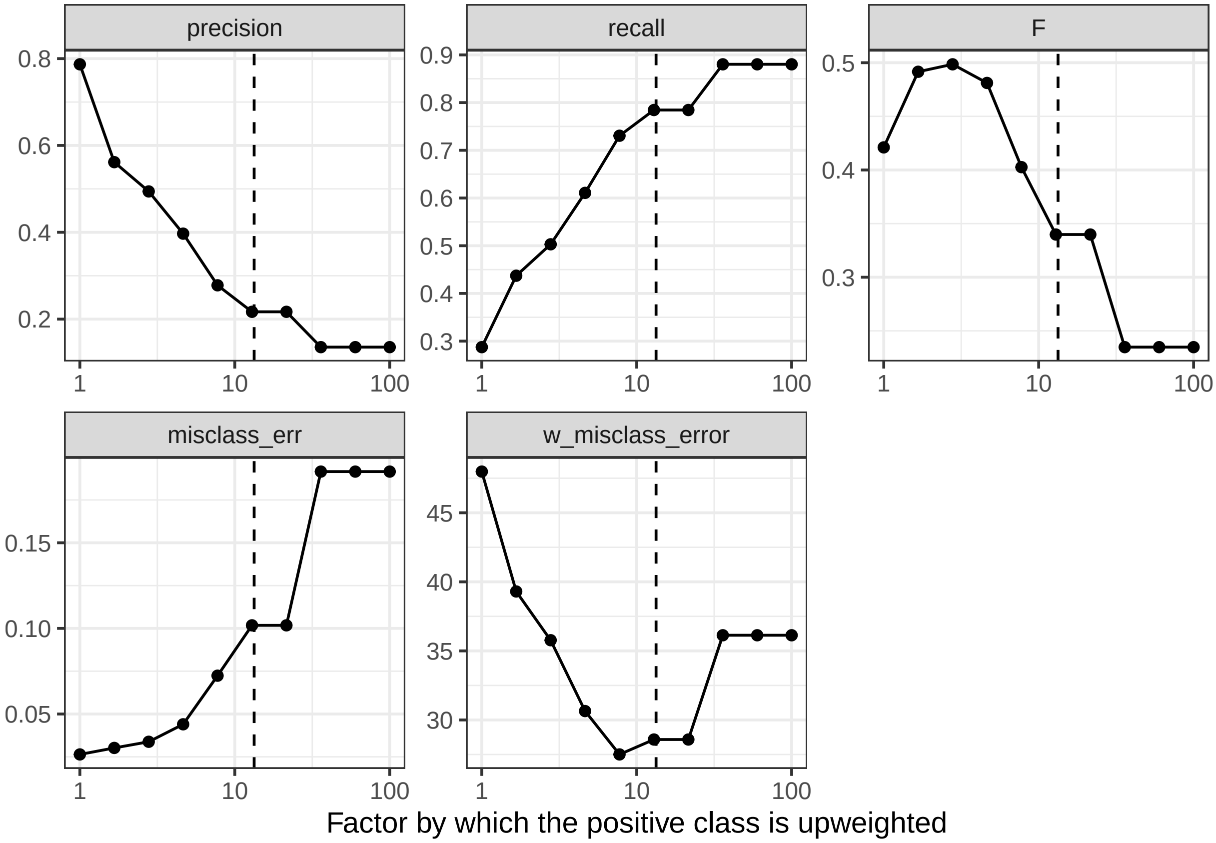
```r
rbind(
  # metrics for unweighted KNN
  classification_metrics(test_responses <- default_test$default,
    test_predictions <- knn_results$predictions,
    C_FP = C_FP,
    C_FN = C_FN
  ) |>
    mutate(weighting = FALSE, .before = 1),
  # metrics for weighted KNN
  classification_metrics(test_responses <- default_test$default,
    test_predictions <- knn_results_weighted$predictions,
    C_FP = C_FP,
    C_FN = C_FN
  ) |>
    mutate(weighting = TRUE, .before = 1)
)
```

```
## # A tibble: 2 x 6
##   weighting misclass_err w_misclass_err precision recall     F
##   <lgl>            <dbl>          <dbl>     <dbl>  <dbl> <dbl>
## 1 FALSE           0.0264           48.0     0.787  0.287 0.421
## 2 TRUE            0.102            28.6     0.217  0.784 0.340
```

Note that adding weighting increased the misclassification error and decreased the precision and F-score. However, it decreased the weighted misclassification error and increased the recall. For any given classification problem, one needs to decide which of these performance metrics is most important. If misclassification costs are available, then arguably the weighted misclassification error is most important.

We can also see what happens as we scan over a range of upweighting factors for the minority class.

```
# logarithmically spaced upweighting factors
upweighting_factors <- exp(seq(log(1), log(100), length.out = 10))
num_weights <- length(upweighting_factors)
# create tibble to store the results
results <- tibble(
  upweighting_factor = numeric(num_weights),
  misclass_err = numeric(num_weights),
  w_misclass_error = numeric(num_weights),
  precision = numeric(num_weights),
  recall = numeric(num_weights),
  `F` = numeric(num_weights)
)
# iterate over upweighting factors
for (weight_idx in 1:num_weights) {
  # define the weights
  upweighting_factor <- upweighting_factors[weight_idx]
  weights <- 1 * (default_train$default == "No") +
    upweighting_factor * (default_train$default == "Yes")
  # run KNN with those weights
  knn_results_weighted <- knn(default ~ balance,
    training_data = default_train,
    test_data = default_test,
    weights = weights,
    k = 25
  )
  # update results tibble
  results[weight_idx,] <- classification_metrics(
    test_responses <- default_test$default,
    test_predictions <- knn_results_weighted$predictions,
    C_FP = C_FP,
    C_FN = C_FN
  ) |>
    mutate(upweighting_factor = upweighting_factor, .before = 1)
}

# plot the results
results |>
  pivot_longer(-upweighting_factor, names_to = "metric", values_to = "value") |>
  mutate(metric = factor(
    metric,
    levels = c("precision", "recall", "F", "misclass_err", "w_misclass_error")
  )) |>
  ggplot(aes(x = upweighting_factor, y = value)) +
  geom_point() +
  geom_line() +
  geom_vline(xintercept = C_FN / C_FP, linetype = "dashed") +
  facet_wrap(~metric, scales = "free") +
  scale_x_log10() +
  labs(
    x = "Factor by which the positive class is upweighted",
    y = element_blank()
  )
```



The dashed vertical line is at an upweighting factor of $C_{\text{FN}}/C_{\text{FP}}$, which is equivalent to weighting the negative cases with C_{FP} and the positive cases with C_{FN}.

Exercises:

- Why is the recall increasing as a function of the factor by which the positive class is upweighted?
- Why is the misclassification error minimized when there is no upweighting?
- Why is the weighted misclassification error minimized roughly when the upweighting corresponds to the misclassification costs C_{FP} and C_{FN}?