Santander Customer Transaction Prediction

Alex Philip Berger

8/19/2019

Introduction

In this notebook, I take on the **Santander Customer Transaction Prediction** Kaggle Competition (descibed in detail here: https://www.kaggle.com/c/santander-customer-transaction-prediction/overview).

In particular, I demonstrate a simple playbook of how to approach *any* machine learning problem - focusing on neural networks in this example.

I focus on the framework of solving a problem, and not on fine-tuning my model. Last, I upload my final model and participate in the competition.

1. Define problem and assemble the dataset

Problem Type

The problem is of the type *binary classification*.

Context: The task is to predict the value of the 'target' variable in the test set. This variable represents if a customer makes a specific transaction in the future, at Santander Bank.

Assemble data set

To assemble data, I power up *R* and load libraries and support functions (*functions.R*):

```
library(tidyverse)
library(keras)
source("functions.R")
```

Data is given in the link. After some manipulation of original files, I can import data as tibbles:

```
test_data <- get_test_data("feather")
train_data <- get_train_data("feather")
```

Training data looks like this:

```
head(train_data[,1:15])
```

```
## # A tibble: 6 x 15
##   ID_code target var_0 var_1 var_2 var_3 var_4 var_5 var_6 var_7 var_8
##   <chr>    <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 train_0      0  8.93 -6.79 11.9   5.09  11.5 -9.28  5.12  18.6 -4.92
## 2 train_1      0 11.5  -4.15 13.9   5.39  12.4  7.04  5.62  16.5  3.15
## 3 train_2      0  8.61 -2.75 12.1   7.89  10.6 -9.08  6.94  14.6 -4.92
## 4 train_3      0 11.1  -2.15  8.95  7.20  12.6 -1.84  5.84  14.9 -5.86
## 5 train_4      0  9.84 -1.48 12.9   6.64  12.3  2.45  5.94  19.3  6.27
## 6 train_5      0 11.5  -2.32 12.6   8.63  11.0  3.56  4.53  15.2  3.59
## # ... with 4 more variables: var_9 <dbl>, var_10 <dbl>, var_11 <dbl>,
## #   var_12 <dbl>
```

I need to predict *target*, and I have var_0, var_1, . . . , var_199 to do so.

Test data is similar, but without any *target* values. This is the data on which I need to apply my model and upload answers to Kaggle. Thus, I only return to this data at the very end.

2. Choose the measure of success

The measure by which the kaggle competition is decided is *Area Under Receiver Operating Characteristic Curve* (ROC AUC). Another measure typically used together with *ROC AUC* is *Accuracy*, so I also pay attention to that.

However, I note that I have a class-imbalanced problem:

```
table(train_data$target)
```

```
##
##      0      1
## 179902  20098
```

Thus, precision and recall are also important measures, as per 'good practice'. I choose to focus on *ROC AUC* and *Accuracy* in despite of this.

3. Decide the evaluation protocol

I choose to maintain a *hold-out validation set* as my evalutaion protocol. This is the simplest approach.

(Other options are *K-fold cross-validation* or *Iterated K-fold validation*).

4. Prepare data

In this step, data needs to be formatted in a way that can be fed into a machine learning network. In this case, I assume a neural network, and transform data accordingly:

4.1) Format data as tensors

Comment: In this case the tensors will be of rank 2 (e.g., a "2D tensor"): Each single data-point can be encoded as a vector.

```
format_as_tensor <- function(tibble){
  tibble %>%
    select(contains("var_")) %>%
    as.matrix() %>% # 2D tensor, each single data-point can be encoded as a vector
    unname()
}
x_train <- format_as_tensor(train_data)
x_test <- format_as_tensor(test_data)
y_train <- as.numeric(train_data$target) # labels should be numeric
rm(test_data, train_data) # drop tibbles
```

4.2) Normalize data

Comment: We generally want homogoenous data with small values as input to our neural network. While data isn't heterogenous, normalizing achieves both:

```
mean <- apply(x_train, 2, mean) # Normalization value 1
std <- apply(x_train, 2, sd) # Normalization value 2
x_train <- scale(x_train, center = mean, scale = std)
x_test <- scale(x_test, center = mean, scale = std)
rm(mean, std)
```

4.3) Inspect data

- Shape of training data (2D tensor = Matrix):

```
dim(x_train)
```

```
## [1] 200000    200
```

- Each single data point is a numeric vector:

```
x_test[1:10, 1:5]
```

```
##              [,1]        [,2]       [,3]       [,4]          [,5]
##  [1,]  0.12686819  2.32279483  0.8475948  1.2884287  0.2183204641
##  [2,] -0.70706522  0.71157784  0.2232229 -0.7882906 -1.1588169797
##  [3,] -1.70958134 -2.15565014 -0.2175369  0.1230209 -0.5024387608
##  [4,] -0.70476263  0.07541194  0.4948355 -0.1084653 -1.3754328815
##  [5,]  0.33745677  0.36911244  1.2928606  0.4669221 -1.2166674722
##  [6,] -1.54395912 -0.16386940 -0.7987415  0.1331025  1.9518637644
##  [7,] -0.72943327 -1.10588381 -1.2703621  0.7175438  0.5742334517
##  [8,]  2.17877464 -0.19594312  1.0162119  0.7846405 -0.0003901307
##  [9,] -1.21521462  0.60933204  1.1363227 -0.9893850 -1.4785657084
## [10,] -0.09829252 -1.31040010  1.4971475  1.1244308  0.5097908370
```

- Labels are numeric:

```
str(y_train)
```

```
##  num [1:200000] 0 0 0 0 0 0 0 0 0 0 ...
```

Conclusion: Data is ready for a neural network!

5. Develop a baseline model (1st working model)

Purpose: Build a simple model to determine if we have *statistical power*, that is, can we beat a "dumb" baseline?

To do so, I make the following design choices:

- Last-layer activation: sigmoid (We need a mapping onto the [0, 1] interval, a probability)
- Loss-function: Crossentropy (Common for binary classification problems - it is also a good proxy metric for ROC AUC)

- Optimization configuration: rmsprop (A safe choice in most cases)

I can now implement a simple model (neural network) using the keras package:

5.1 I define the model:

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = dim(x_train)[2]) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Comment: We are looking at the most simple deep learning problem we can ever encounter: The input data are vectors (each obseration is a vector), and the labels are scalars. The above model (3 fully connected dense layers, 2 with relu activations, and 1 with a sigmoid activation to output probabilities) is generally known to perform well on this problem type.

5.2 I compile the model:

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

5.3 I create validation data (by splitting *x_train* in two: one set for *training* and one set for *validating*)

```
set.seed(04651)
validation_prop <- 0.25 # I assign 25 % of the training data to *validation data*.
n <- dim(x_train)[1]
val_indices <- sample(n, n*(1-validation_prop)) # indices for validation data
x_train_train <- x_train[val_indices,]
x_train_validate <- x_train[-val_indices,]
y_train_train <- y_train[val_indices]
y_train_validate <- y_train[-val_indices]
```

The tensors just defined, starting with "partial_" are now the central one in training and validating my solution.

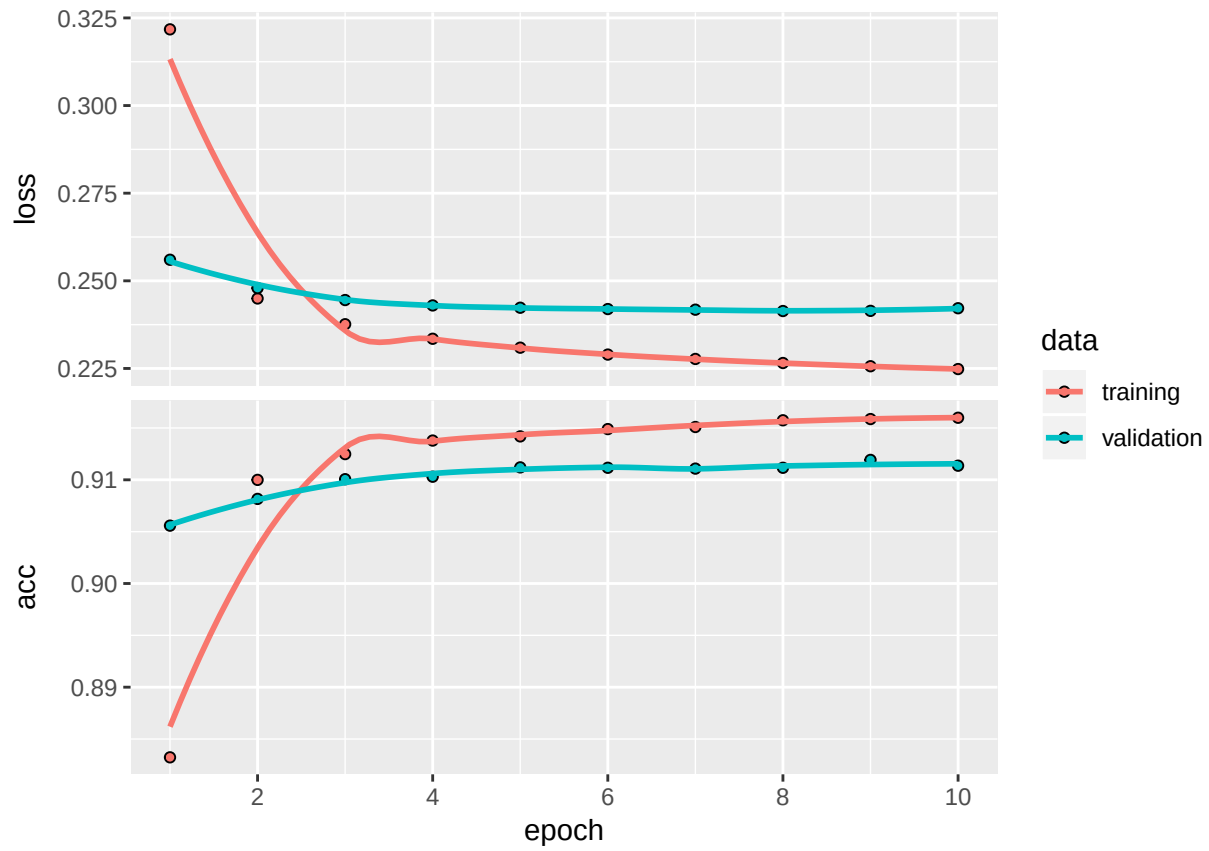
5.4 I fit the model:

```
history <- model %>% fit(
  x_train_train,
  y_train_train,
  epocs = 20, # iterate 20 times through data
  batch_size = 2^9, # update weights through feedback from 512 samples at a time
  validation_data = list(x_train_validate, y_train_validate)
)
```

5.5 I validate the approach:

- First, I plot the training and validation history:

```
plot(history)
```



I observe that loss decreases and accuracy increases with every epoch. The results are (as expected) better in training than in the validation data: We are *overfitting*. Further, the network makes no improvements (in the validation data) after 8th epoch (roughly).

Now, the important question: Does the network display **statistical power**?

First, the accuracy rate in the validation data is ~ 91 %:

```
metrics <- model %>% evaluate(x_train_validate, y_train_validate)
metrics$acc
```

```
## [1] 0.91136
```

This can also be obtained through the history object, or through an explicit calculation:

```
# Accuracy from 'history' object
as.data.frame(history) %>%
  filter(data == "validation", metric == "acc", epoch == max(epoch))
```

```
##   epoch   value metric       data
## 1    10 0.91136    acc validation
```

```
# Accuracy through excplicit calculation
predictions <- model %>% keras::predict_classes(x_train_validate) %>% as.vector()
actuals <- y_train_validate
sum(predictions == actuals) / length(predictions)
```

```
## [1] 0.91136
```

Second, how would a random guess perform? Context: The validation data has 50000 observations. In the above, we predict 2231 positive labels (transactions). In fact, the validation data has 5085 positive labels. I now construct two *random guesses* based on these numbers:

```
random_binary_prediction <- function(n = 100, n_pos = 20){
  random_prediction <- rep(0, n)
  random_prediction[base::sample(x = n, size = n_pos)] <- 1
  return(random_prediction)
}
random_prediction_1 <- random_binary_prediction(n = length(actuals), n_pos = sum(actuals))
random_prediction_2 <- random_binary_prediction(n = length(actuals), n_pos = sum(predictions))
```

The accuracy of the two random predictions:

```
sum(random_prediction_1 == actuals) / length(actuals)
```

```
## [1] 0.817
```

```
sum(random_prediction_2 == actuals) / length(actuals)
```

```
## [1] 0.86336
```

I conclude that the our simple neural network performs 5 % better than a random guess in absolute terms. Thus, there *is* statistical power.

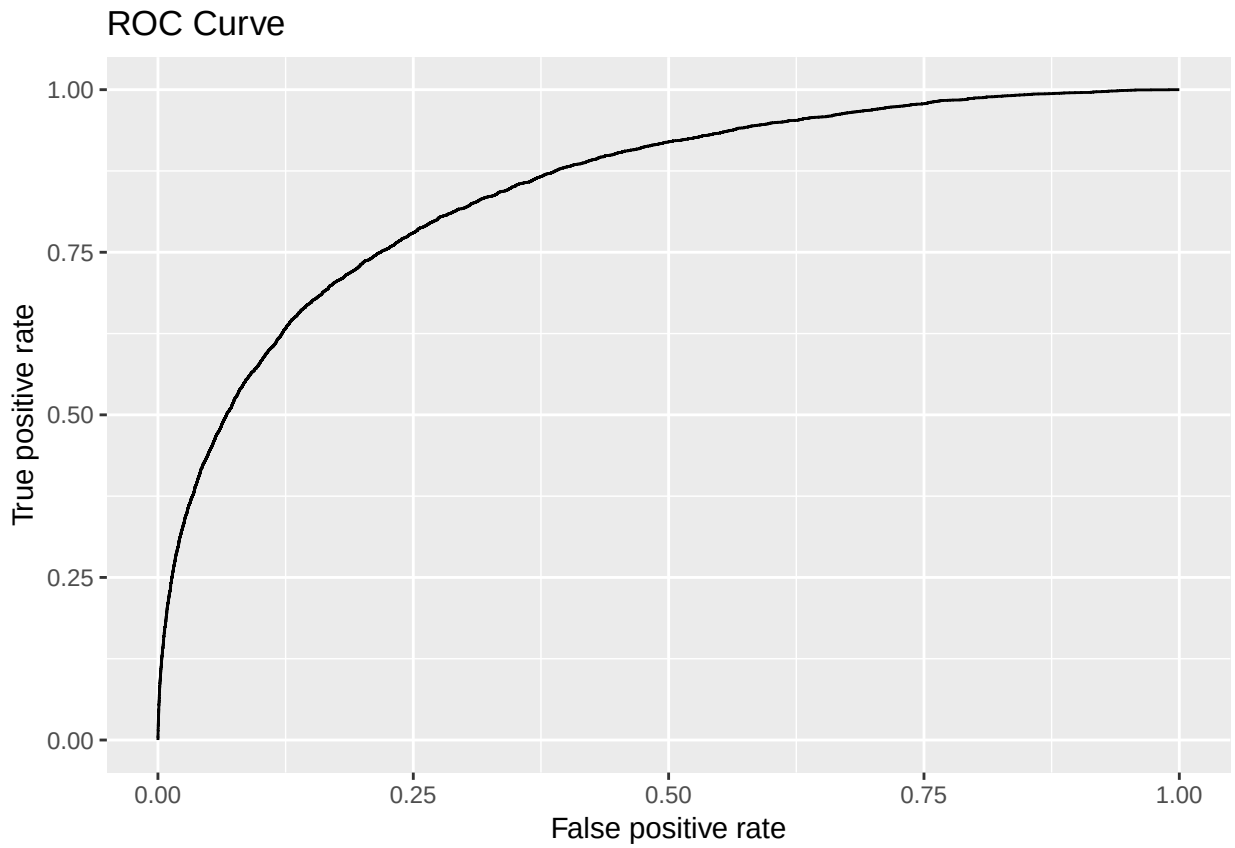
To get a sense of the **measure of success**, ROC AUC, defined in section 2, I calculate the value, as well as the corresponding curve:

```
rocr_pred_object <- ROCR::prediction(
  model %>% keras::predict_proba(x_train_validate) %>% as.vector(), # prediction prob
  y_train_validate # labels
)
ROC_AUC_metric(rocr_pred_object)
```

```
## [1] 0.8473313
```

The value is the area under the following curve:

```
ROC_AUC_plot(rocr_pred_object)
```



Finally, I save the model.

```
save_model_hdf5(model, "Model_01_Baseline.h5")
```

And clean up the environment:

```
keep <- c("x_test", "x_train", "x_train_train", "y_train_train",
          "x_train_validate", "y_train_validate")
rm(list = setdiff(ls(), keep))
source("functions.R") # reload functions
```

6. Scaling up: Develop a model that overfits

Motivation: We have statistical power, but there is still room for a more powerful model. In machine learning, the ideal model is found at on the border between under- and overfitting.

Since the model from #5 already displays overfitting, I skip this step. Normal operations would be to (1) add layers (2) make the layers bigger, and (3) train network for more epochs.

7. Modify and Tune model

This is the main step, and a process where the model is repeatedly modified, trained, evaluated, to achieve the best possible model. The goal is to find a model that *generalize*, that is, does not overfit.

We need to *regularize* the model from #5 (or, normally #6). The simplest way to do this is to reduce the size of the model, that is, reduce the number of layers, or the number of units per layer. In this way, the network can memorize fewer patterns, and will focus on the more important ones, that generalize better.

Other approaches is to add weight regularization (add cost of having large weights in the network) *or* add dropout (zero'ing values from each layer output). I go with the former:

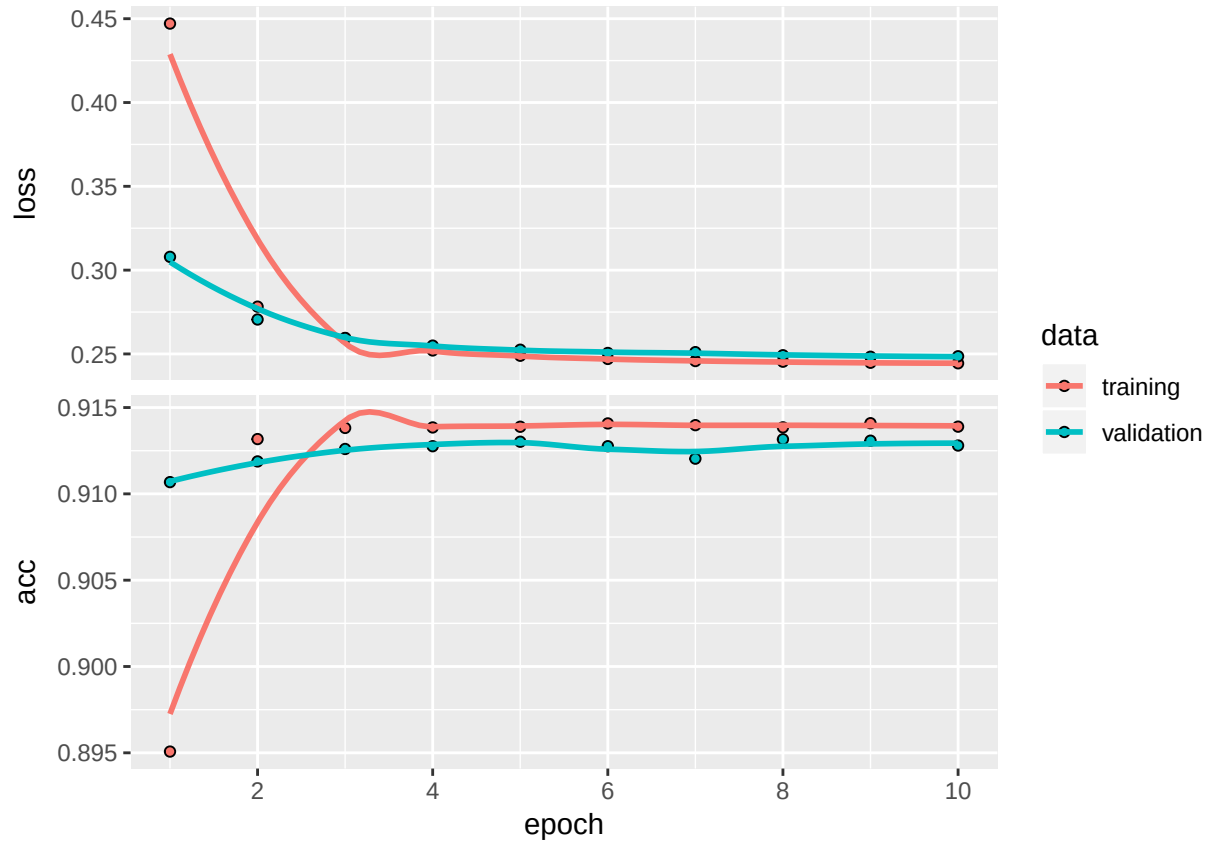
```
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu",
              kernel_regularizer = regularizer_l1_l2(l1 = 0.001, l2 = 0.001),
              input_shape = dim(x_train)[2]) %>%
  layer_dense(units = 16, activation = "relu",
              kernel_regularizer = regularizer_l1_l2(l1 = 0.001, l2 = 0.001)) %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history <- model %>% fit(
  x_train_train,
  y_train_train,
  epocs = 25, # iterate 25 times through data
  batch_size = 2^8,
  validation_data = list(x_train_validate, y_train_validate)
)
```

I then plot output:

```
plot(history)
```



Now, the overfitting has disappeared: The model generalizes better.

Further, accuracy is *higher* than the previous model:

```
metrics <- model %>% evaluate(x_train_validate, y_train_validate)
metrics$acc
```

```
## [1] 0.9128
```

And so is ROC AUC:

```
rocr_pred_object <- ROCR::prediction(
  model %>% keras::predict_proba(x_train_validate) %>% as.vector(), # prediction probs
  y_train_validate # labels
)
ROC_AUC_metric(rocr_pred_object)
```

```
## [1] 0.8545907
```

More fine-tuning could be done to optimise model results. Some suggestions are already mentioned in the beginning of #7.

Other suggestions:

- Switch evaluation protocol, for example to *Iterated K-fold validation*. This would give more training data, and allow me to evaluate my model more precisely.

- Other models could be tried, e.g., gradient boosting trees. Deep learning may not be necessary for this specific problem!

However, to keep the example simple, I decide to finish here, and save the model.

```
save_model_hdf5(model, "Model_02_Final.h5"); rm(model);
```

And that's it: The 7 steps just demonstrated are natural parts of all machine learning problems.

The final step demonstrates the *upload* step to participate in Kaggle competitions.

8. The Kaggle Step: Upload results

First, the chosen model must be applied to the *test data*. This was already prepared in the above steps, and looks like this:

```
x_test[1:10, 1:5]
```

```
##              [,1]        [,2]       [,3]       [,4]          [,5]
##  [1,]  0.12686819  2.32279483  0.8475948  1.2884287  0.2183204641
##  [2,] -0.70706522  0.71157784  0.2232229 -0.7882906 -1.1588169797
##  [3,] -1.70958134 -2.15565014 -0.2175369  0.1230209 -0.5024387608
##  [4,] -0.70476263  0.07541194  0.4948355 -0.1084653 -1.3754328815
##  [5,]  0.33745677  0.36911244  1.2928606  0.4669221 -1.2166674722
##  [6,] -1.54395912 -0.16386940 -0.7987415  0.1331025  1.9518637644
##  [7,] -0.72943327 -1.10588381 -1.2703621  0.7175438  0.5742334517
##  [8,]  2.17877464 -0.19594312  1.0162119  0.7846405 -0.0003901307
##  [9,] -1.21521462  0.60933204  1.1363227 -0.9893850 -1.4785657084
## [10,] -0.09829252 -1.31040010  1.4971475  1.1244308  0.5097908370
```

I then load and run the model:

```
model <- load_model_hdf5("Model_02_Final.h5")
results <- model %>% predict_classes(x_test)
```

The results are now in a binary vector:

```
results[1:10]
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

The distribution is as follows:

```
table(results)
```

```
## results
##      0      1
## 194566   5434
```

I now reload the original tibble of test_data. And combine it with the results just optained.

```
test_data <- get_test_data("feather")

to_csv <- tibble(
  ID_Code = test_data$ID_code,
  target = results
)
```

I can now export to csv:

```
readr::write_csv(to_csv, path = "sample_submission.csv")
```

And, finally, upload to: https://www.kaggle.com/c/santander-customer-transaction-prediction/submit.

Another option is to submit via the command line: kaggle competitions submit -c santander-customer-transaction-prediction -f submission.csv -m "Message"