Final Project

Tim Mastny

5/2/2018

Credit Card Defaults

This project will study credit card defaults. This report, the data, and all the code can be found at this project's Github repo. The following study was conducted on the dataset:

> Yeh, I. C., & Lien, C. H. (2009). The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. Expert Systems with Applications, 36(2), 2473-2480

The study found the following error rates based on a validation set. My goal is to see if we can replicate and improve on these results using modern machine learning techniques. Additionally, I would like to create a machine learning pipeline that allows reproducibility, while being flexible to changes.

Table 1
Classification accuracy

Method	Error rate		Area ratio	
	Training	Validation	Training	Validation
K-nearest neighbor	0.18	0.16	0.68	0.45
Logistic regression	0.20	0.18	0.41	0.44
Discriminant analysis	0.29	0.26	0.40	0.43
Naïve Bayesian	0.21	0.21	0.47	0.53
Neural networks	0.19	0.17	0.55	0.54
Classification trees	0.18	0.17	0.48	0.536

Cleaning

After we handle the excel format and some naming issues, the data itself is very clean.

```
library(here)
library(tidyverse)
d <- readxl::read_xls(here("data", "default of credit card clients.xls"),
                      skip = 1) %>%
  rename(default = `default payment next month`)

d %>%
  select_if(~any(is.na(.)))
```

```
## # A tibble: 30,000 x 0
```

Therefore, we will focus focus on data preprocessing and exploration.

Preprocessing

Note that the paper uses `knn` and neural networks, which are both very sensitive to initial conditions. Both perform better when working on standardized data, so we'll want two versions of the data to compare against.

Likewise, the model evaluation method in the paper is based on a hold-out set. We want to be sure to split the data before calculating means and standard deviations so our training data doesn't spill over into our test set.

We'll use `rsample::initial_split` to divide our data into 2/3 training and 1/3 testing.

Likewise, we'll use this recipes package to prepare the data for the models:

```
rec <- recipe(default ~ ., training(splits)) %>%
  step_num2factor(default, SEX, EDUCATION, MARRIAGE) %>%
  step_dummy(SEX, EDUCATION, MARRIAGE) %>%
  prep(training(splits))
```

This recipe can be easily modified to standardize the data as well.

Exploratory Analysis

Formatting as .csv

The first thing we need to do is to create a `.csv` out of the original `.xls` file type found on UCI Machine Learning Repository.

The file is formatted with two headers, so we'll be sure to skip the first one:

```
library(here)
library(tidyverse)

d <- readxl::read_xls(here("data", "default of credit card clients.xls"), skip = 1)
d %>%
  rename(default = `default payment next month`) %>%
  write_csv(here("data", "defaults.csv"))
```

This makes it more portable between systems, especially if we want to read it into Python to use Keras for deep learning.

Tidying

Let's take a look at our data:

```
library(here)
library(tidyverse)
library(recipes)

d <- read_csv(here("data", "defaults.csv"))
d <- d %>%
  select(-ID) %>%
  select(default, everything())
```

```
glimpse(d)
```

```
## Observations: 30,000
## Variables: 24
```

```
## $ default   <int> 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0...
## $ LIMIT_BAL <dbl> 20000, 120000, 90000, 50000, 50000, 50000, 500000, 1...
## $ SEX       <int> 2, 2, 2, 2, 1, 1, 1, 2, 2, 1, 2, 2, 2, 1, 1, 2, 1, 1...
## $ EDUCATION <int> 2, 2, 2, 2, 2, 1, 1, 2, 3, 3, 3, 1, 2, 2, 1, 3, 1, 1...
## $ MARRIAGE  <int> 1, 2, 2, 1, 1, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 3, 2, 1...
## $ AGE       <int> 24, 26, 34, 37, 57, 37, 29, 23, 28, 35, 34, 51, 41, ...
## $ PAY_0     <int> 2, -1, 0, 0, -1, 0, 0, 0, 0, -2, 0, -1, -1, 1, 0, 1,...
## $ PAY_2     <int> 2, 2, 0, 0, 0, 0, 0, -1, 0, -2, 0, -1, 0, 2, 0, 2, 0...
## $ PAY_3     <int> -1, 0, 0, 0, -1, 0, 0, -1, 2, -2, 2, -1, -1, 2, 0, 0...
## $ PAY_4     <int> -1, 0, 0, 0, 0, 0, 0, 0, 0, -2, 0, -1, -1, 0, 0, 0, ...
## $ PAY_5     <int> -2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, -1, -1, 0, 0, 0, ...
## $ PAY_6     <int> -2, 2, 0, 0, 0, 0, 0, -1, 0, -1, -1, 2, -1, 2, 0, 0,...
## $ BILL_AMT1 <dbl> 3913, 2682, 29239, 46990, 8617, 64400, 367965, 11876...
## $ BILL_AMT2 <dbl> 3102, 1725, 14027, 48233, 5670, 57069, 412023, 380, ...
## $ BILL_AMT3 <dbl> 689, 2682, 13559, 49291, 35835, 57608, 445007, 601, ...
## $ BILL_AMT4 <dbl> 0, 3272, 14331, 28314, 20940, 19394, 542653, 221, 12...
## $ BILL_AMT5 <dbl> 0, 3455, 14948, 28959, 19146, 19619, 483003, -159, 1...
## $ BILL_AMT6 <dbl> 0, 3261, 15549, 29547, 19131, 20024, 473944, 567, 37...
## $ PAY_AMT1  <dbl> 0, 0, 1518, 2000, 2000, 2500, 55000, 380, 3329, 0, 2...
## $ PAY_AMT2  <dbl> 689, 1000, 1500, 2019, 36681, 1815, 40000, 601, 0, 0...
## $ PAY_AMT3  <dbl> 0, 1000, 1000, 1200, 10000, 657, 38000, 0, 432, 0, 5...
## $ PAY_AMT4  <dbl> 0, 1000, 1000, 1100, 9000, 1000, 20239, 581, 1000, 1...
## $ PAY_AMT5  <dbl> 0, 0, 1000, 1069, 689, 1000, 13750, 1687, 1000, 1122...
## $ PAY_AMT6  <dbl> 0, 2000, 5000, 1000, 679, 800, 13770, 1542, 1000, 0,...
```

```
d %>%
  select_if(~any(is.na(.)))
```

```
## # A tibble: 30,000 x 0
```

The data looks pretty clean. All numerics without any NAs.

Next, we'll split the data into a training and test set, since that is how the paper evaluated their models. The paper doesn't specify the split percent, so we will use 33%. However, the outcomes are very unbalanced:

```
d %>%
  group_by(default) %>%
  summarise(count = n())
```

```
## # A tibble: 2 x 2
##   default count
##     <int> <int>
## 1       0 23364
## 2       1  6636
```

The paper explicitly says it created the validation set randomly, so we will avoid using stratified sampling.

```
library(rsample)
default_splits <- initial_split(d, prop = 2/3)
```

Next, we'll use the `recipes` package to process our data. Some of the columns in the dataset are actually categorical variables encoded as integers. To be consistent between Python and R methods which may or may not accept categorical features, I will use recipes to dummy encode as necessary.

```
default_rec <- recipe(default ~ ., training(default_splits)) %>%
  step_num2factor(SEX, EDUCATION, MARRIAGE) %>%
  step_dummy(SEX, EDUCATION, MARRIAGE) %>%
  prep(training(default_splits))
```

Lastly, we'll also standardize the data set, a standard technique used in some of the study's methods such as kNN and neural networks. The study does not say whether or not this sort of data pre-processing was used, so we will include it in our testing to see whether it impacts our results.

Also important: we'll use the training data to calculate the mean and standard deviation, so our testing data doesn't leak into our training.

```
default_std_rec <- recipe(default ~ ., training(default_splits)) %>%
  step_center(all_predictors(), -SEX, -EDUCATION, -MARRIAGE) %>%
  step_scale(all_predictors(), -SEX, -EDUCATION, -MARRIAGE) %>%
  step_num2factor(SEX, EDUCATION, MARRIAGE) %>%
  step_dummy(SEX, EDUCATION, MARRIAGE) %>%
  prep(training(default_splits))
```

Then we will save the data as a `.csv` for future training.

```
bake(default_rec, training(default_splits)) %>%
  write_csv(here("data", "train.csv"))

bake(default_rec, testing(default_splits)) %>%
  write_csv(here("data", "test.csv"))

bake(default_std_rec, training(default_splits)) %>%
  write_csv(here("data", "s-train.csv"))

bake(default_std_rec, testing(default_splits)) %>%
  write_csv(here("data", "s-test.csv"))
```

```
glimpse(read_csv(here("data", "train.csv")))
```

```
## Observations: 20,000
## Variables: 31
## $ default      <int> 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0...
## $ LIMIT_BAL    <dbl> 20000, 120000, 90000, 50000, 50000, 500000, 20000...
## $ AGE          <int> 24, 26, 34, 37, 37, 29, 34, 41, 30, 23, 24, 49, 4...
## $ PAY_0        <int> 2, -1, 0, 0, 0, 0, 0, -1, 1, 1, 0, 0, 1, 1, 0, -2...
## $ PAY_2        <int> 2, 2, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, -2, -2, 0, -2...
## $ PAY_3        <int> -1, 0, 0, 0, 0, 0, 2, -1, 2, 0, 2, 0, -2, -2, 0, ...
## $ PAY_4        <int> -1, 0, 0, 0, 0, 0, 0, -1, 0, 0, 2, -1, -2, -2, 0,...
## $ PAY_5        <int> -2, 0, 0, 0, 0, 0, 0, -1, 0, 0, 2, -1, -2, -2, 0,...
## $ PAY_6        <int> -2, 2, 0, 0, 0, 0, -1, -1, 2, 0, 2, -1, -2, -2, -...
## $ BILL_AMT1    <dbl> 3913, 2682, 29239, 46990, 64400, 367965, 11073, 1...
## $ BILL_AMT2    <dbl> 3102, 1725, 14027, 48233, 57069, 412023, 9787, 65...
## $ BILL_AMT3    <dbl> 689, 2682, 13559, 49291, 57608, 445007, 5535, 650...
## $ BILL_AMT4    <dbl> 0, 3272, 14331, 28314, 19394, 542653, 2513, 6500,...
## $ BILL_AMT5    <dbl> 0, 3455, 14948, 28959, 19619, 483003, 1828, 6500,...
## $ BILL_AMT6    <dbl> 0, 3261, 15549, 29547, 20024, 473944, 3731, 2870,...
## $ PAY_AMT1     <dbl> 0, 0, 1518, 2000, 2500, 55000, 2306, 1000, 3200, ...
## $ PAY_AMT2     <dbl> 689, 1000, 1500, 2019, 1815, 40000, 12, 6500, 0, ...
## $ PAY_AMT3     <dbl> 0, 1000, 1000, 1200, 657, 38000, 50, 6500, 3000, ...
## $ PAY_AMT4     <dbl> 0, 1000, 1000, 1100, 1000, 20239, 300, 6500, 3000...
## $ PAY_AMT5     <dbl> 0, 0, 1000, 1069, 1000, 13750, 3738, 2870, 1500, ...
## $ PAY_AMT6     <dbl> 0, 2000, 5000, 1000, 800, 13770, 66, 0, 0, 1100, ...
## $ SEX_X2       <int> 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0...
## $ EDUCATION_X1 <int> 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1...
## $ EDUCATION_X2 <int> 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0...
```

```
## $ EDUCATION_X3 <int> 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0...
## $ EDUCATION_X4 <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ EDUCATION_X5 <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ EDUCATION_X6 <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ MARRIAGE_X1  <int> 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0...
## $ MARRIAGE_X2  <int> 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1...
## $ MARRIAGE_X3  <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0...
```

```
glimpse(read_csv(here("data", "s-train.csv")))
```

```
## Observations: 20,000
## Variables: 31
## $ default      <int> 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0...
## $ LIMIT_BAL    <dbl> -1.13711060, -0.36645924, -0.59765465, -0.9059151...
## $ AGE          <dbl> -1.2417420, -1.0244586, -0.1553250, 0.1706001, 0....
## $ PAY_0        <dbl> 1.81021178, -0.87509756, 0.02000555, 0.02000555, ...
## $ PAY_2        <dbl> 1.7990745, 1.7990745, 0.1173821, 0.1173821, 0.117...
## $ PAY_3        <dbl> -0.7013392, 0.1409079, 0.1409079, 0.1409079, 0.14...
## $ PAY_4        <dbl> -0.6692800, 0.1906455, 0.1906455, 0.1906455, 0.19...
## $ PAY_5        <dbl> -1.5322418, 0.2354066, 0.2354066, 0.2354066, 0.23...
## $ PAY_6        <dbl> -1.4866861, 1.9957279, 0.2545209, 0.2545209, 0.25...
## $ BILL_AMT1    <dbl> -0.643205266, -0.660056573, -0.296514593, -0.0535...
## $ BILL_AMT2    <dbl> -0.64727769, -0.66671019, -0.49310188, -0.0103798...
## $ BILL_AMT3    <dbl> -0.66799779, -0.63916247, -0.48179080, 0.03519038...
## $ BILL_AMT4    <dbl> -0.6722309, -0.6212272, -0.4488403, -0.2308742, -...
## $ BILL_AMT5    <dbl> -0.66222121, -0.60516530, -0.41536974, -0.1839919...
## $ BILL_AMT6    <dbl> -0.65027072, -0.59554630, -0.38933544, -0.1544282...
## $ PAY_AMT1     <dbl> -0.347205137, -0.347205137, -0.254099287, -0.2245...
## $ PAY_AMT2     <dbl> -0.22817582, -0.21469837, -0.19303044, -0.1705391...
## $ PAY_AMT3     <dbl> -0.29938977, -0.24192551, -0.24192551, -0.2304326...
## $ PAY_AMT4     <dbl> -0.3279612, -0.2580674, -0.2580674, -0.2510780, -...
## $ PAY_AMT5     <dbl> -0.3078426, -0.3078426, -0.2444465, -0.2400722, -...
## $ PAY_AMT6     <dbl> -0.293466154, -0.180003332, -0.009809099, -0.2367...
## $ SEX_X2       <int> 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0...
## $ EDUCATION_X1 <int> 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1...
## $ EDUCATION_X2 <int> 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0...
## $ EDUCATION_X3 <int> 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0...
## $ EDUCATION_X4 <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ EDUCATION_X5 <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ EDUCATION_X6 <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ MARRIAGE_X1  <int> 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0...
## $ MARRIAGE_X2  <int> 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1...
## $ MARRIAGE_X3  <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0...
```

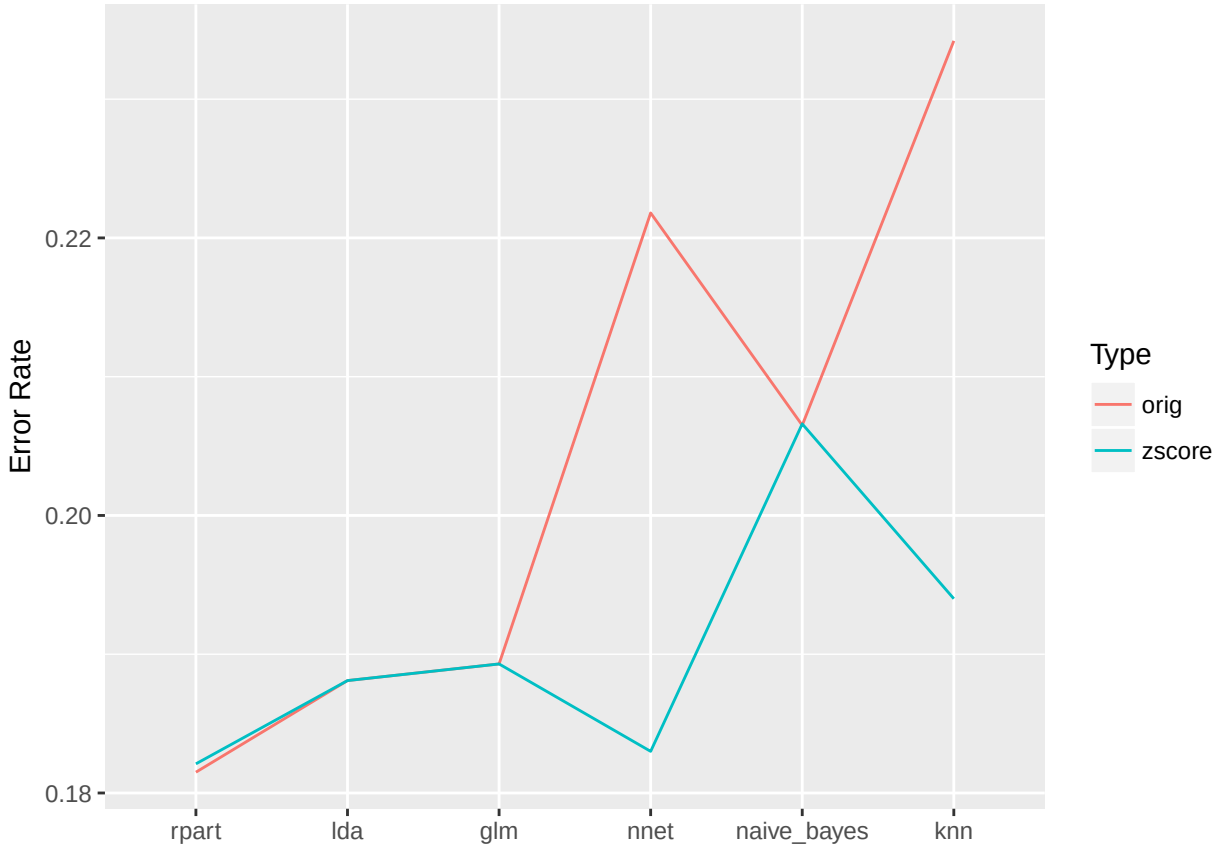
Results

```
library(tidyverse)
library(leadr)
```

```
paper_methods <- c("knn", "glm", "lda", "naive_bayes", "nnet", "rpart")
b <- board() %>%
  filter(model %in% paper_methods) %>%
  mutate(error = 1 - public) %>%
```

```
  select(model, error, dir)

ggplot(b, aes(fct_reorder(model, error), error, group = dir, color = dir)) +
  geom_line() +
  labs(x = NULL, y = "Error Rate", color = "Type")
```

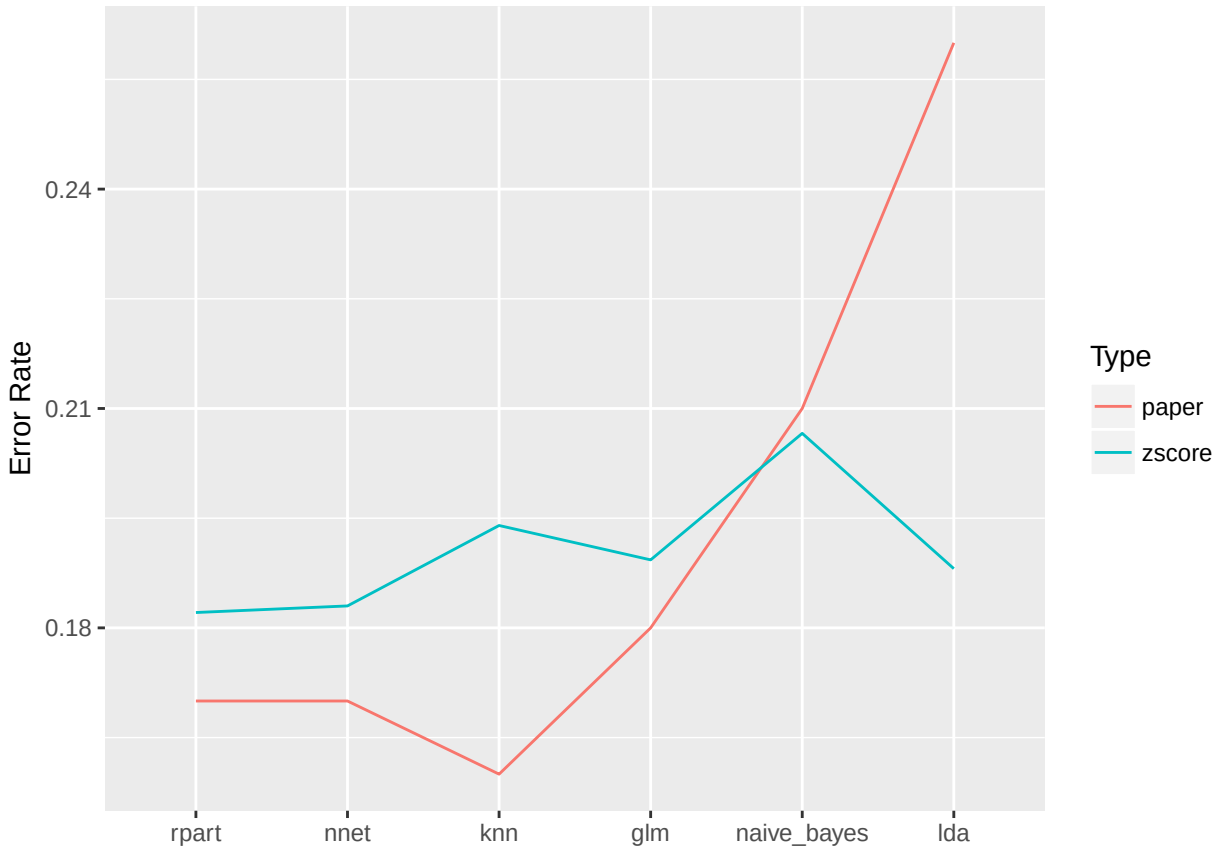


This graph also demonstrates the necessity to standardize for `knn` and neural newtorks.

Keeping that in mind, let's compare `zscore` scores to the error rates from the paper:

```
d <- tibble(
  model = c("knn", "glm", "lda", "naive_bayes", "nnet", "rpart"),
  error = c(.16, .18, .26, .21, .17, .17),
  dir = rep("paper", 6)
)

bind_rows(b, d) %>%
  filter(dir != "orig") %>%
  ggplot(aes(fct_reorder(model, error), error, group = dir, color = dir)) +
  geom_line() +
  labs(x = NULL, y = "Error Rate", color = "Type")
```



We see substantially lower error rate in the original paper for all models except linear discriminate analysis. However, I would argue that the paper makes replication difficult for a few reasons:

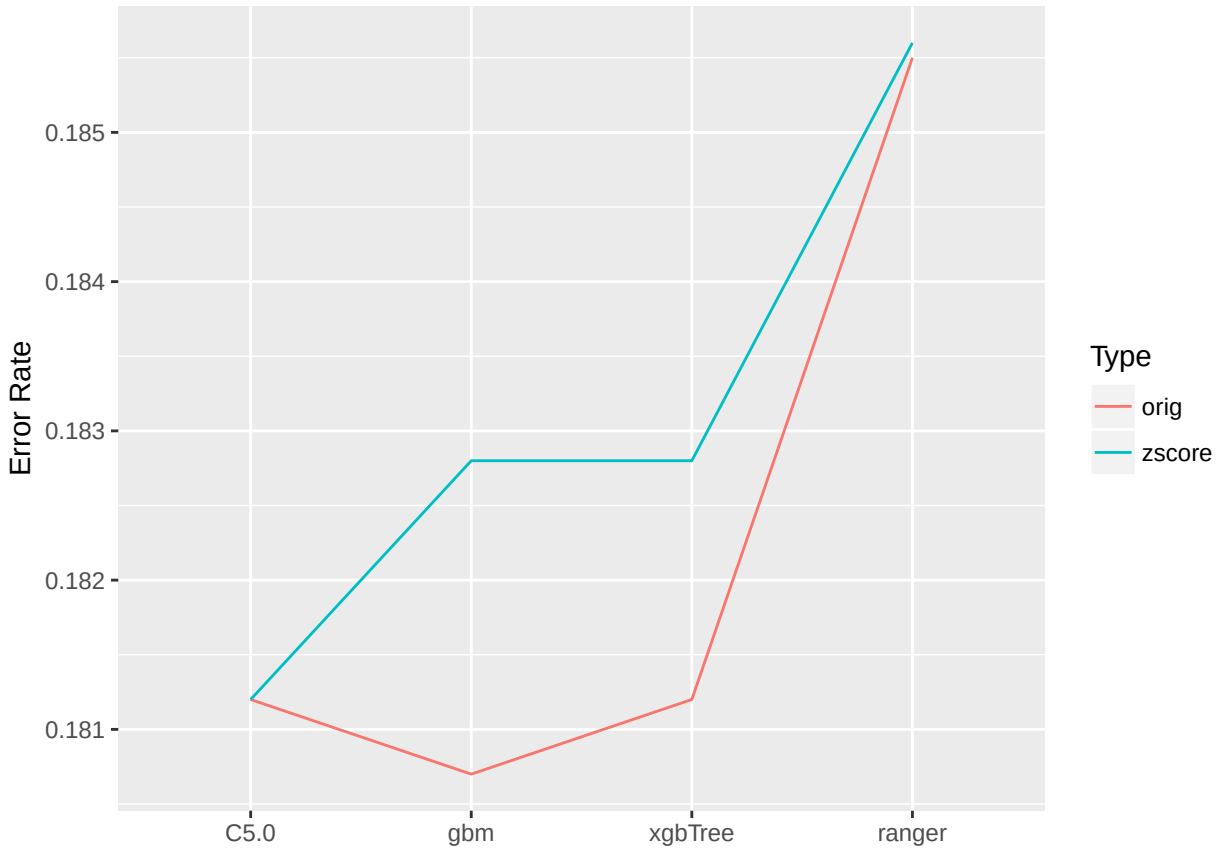
1. They don't list the model implementations used or tuning parameters.
2. They don't discuss the validation set proportion, or whether they used stratified sampling.
3. They opted to use a hold set. It can be shown that the average error rated measured by repeated k-fold cross-validation has less bias and variance[1].

Improvements

One missing group of models is tree ensembles, such as random forests and gradient boosting. Let's see how those results compare:

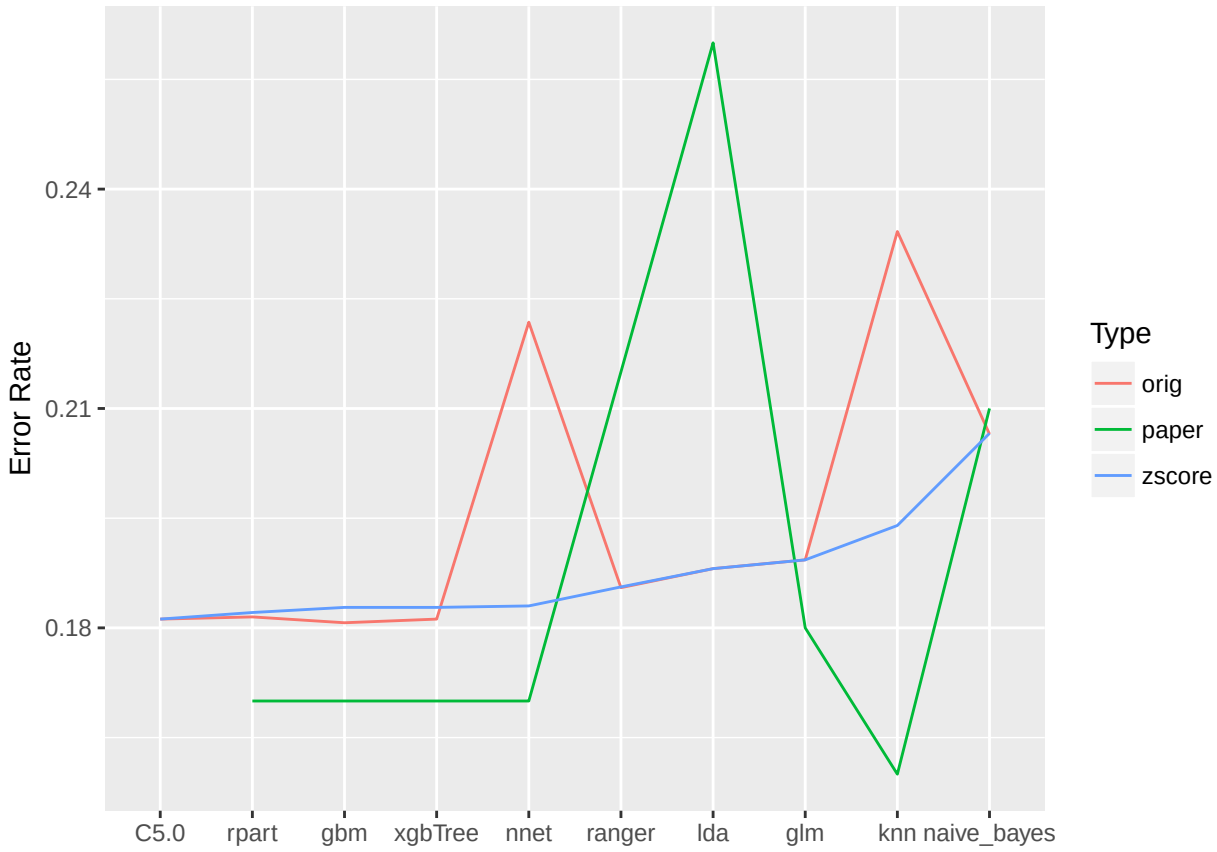
```
board() %>%
  filter(!model %in% paper_methods) %>%
  mutate(error = 1 - public) %>%
  ggplot(aes(fct_reorder(model, error), error, group = dir, color = dir)) +
  geom_line() +
  labs(x = NULL, y = "Error Rate", color = "Type")
```

[1]See Max Kuhn: cross-validation http://appliedpredictivemodeling.com/blog/2014/11/27/vpuig01pqbklmi72b8lcl3ij5hj2qm



These are actually very comparable to the models tested in the paper:

```
board() %>%
  mutate(error = 1 - public) %>%
  bind_rows(d) %>%
  ggplot(aes(fct_reorder(model, error), error, group = dir, color = dir)) +
  geom_line() +
  labs(x = NULL, y = "Error Rate", color = "Type")
```

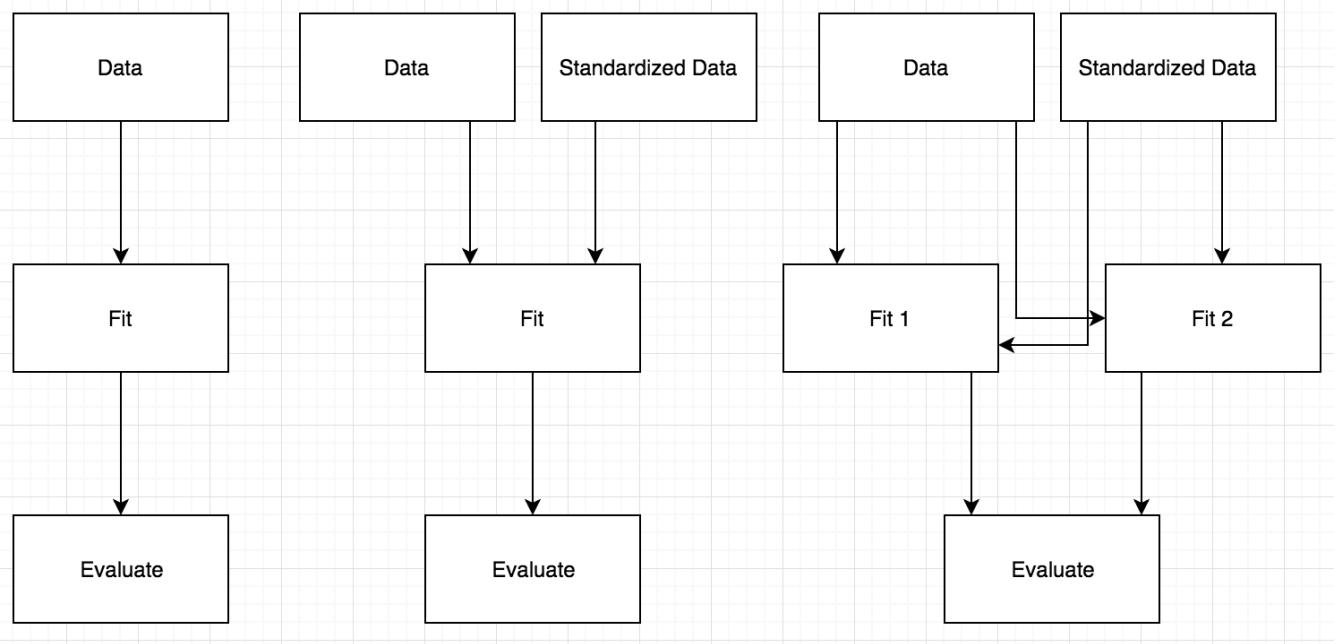


Model Pipeline

I have a personal interest in machine learning pipelines and organization. In our first class contest, I created the package `leadr` to record and manage models fit on the data in response to my total lack of organization..

Contest 2 was better. I used the `leadr` package, so I saved all my models and accurately tracked my progress. However, my project directory was still a mess. I was trying to compare models across data preprocessing techniques and had no system to organize modeling scripts.

For any given model, the pipeline is straightfoward: `data -> fit -> evaluate`. But in the real world, you have different versions of the data, multiple models to fit, and you need a way to evaluate and compare multiple models.



And when you go back and find a mistake or make a change, you are forced to rerun all the steps.

For this project, I was determined to find a better way. I decided to use the R package drake. Drake is a tool that allows you to specify dependencies on R objects. Drake tracks the latest modified date on all dependencies, and rebuilds whenever that changes.

Drake

Let's look at the plan I used for this project.

```
library(drake)
default_plan <- drake_plan(
  data = tidy_data(file_in("data/default of credit card clients.xls")),
  splits = split_data(data),
  prepped = prep_data(splits, "type__"),
  baked = bake_data(splits, prepped_type__),
  model = run(def_model, default ~ ., baked_type__$train,
              list(model = "method__")),
  save_results(model_type___method__, "type__", baked_type__$test)
)
```

The `drake_plan` simply returns a dataframe that lists all the dependencies between the R functions. Each `command` will be ran and saved into the target. That target is then available for future commands.

```
default_plan
```

```
## # A tibble: 6 x 2
##   target         command
##   <chr>          <chr>
## 1 data           tidy_data(file_in('data/default of credit card clients.x~
## 2 splits         split_data(data)
## 3 prepped        prep_data(splits, 'type__')
## 4 baked          bake_data(splits, prepped_type__)
## 5 model          run(def_model, default ~ ., baked_type__$train, list(mod~
```

```
## 6 drake_target_1 "save_results(model_type___method__, \"type__\", baked_t~
```

The best part about drake is the flexibility. Once you've defined the workflow, you can iterate and expand to include any model types or proprocessing. In my example, I use the wildcard variables `type__` and `model__` to automatically adjust the plan:

```
rules = list(type__ = "orig", method__ = "rf")
evaluate_plan(default_plan, rules = rules)
```

```
## # A tibble: 6 x 2
##   target                 command
##   <chr>                  <chr>
## 1 data                   tidy_data(file_in('data/default of credit card c~
## 2 splits                 split_data(data)
## 3 prepped_orig           prep_data(splits, 'orig')
## 4 baked_orig             bake_data(splits, prepped_orig)
## 5 model_orig_rf          run(def_model, default ~ ., baked_orig$train, li~
## 6 drake_target_1_orig_rf "save_results(model_orig_rf, \"orig\", baked_ori~
```

Here, we explicitly tell the drake plan we want to run a random forest model on the original, unprocessed data.

But the real power comes from expanding. Let's say we also want to run a `knn` model. We could replace `knn` with `rf`, but we want to keep the random forest model for comparison. Wildcards automatically expand the plan to accommodate these additons:

```
rules = list(type__ = "orig", method__ = c("rf", "knn"))
evaluate_plan(default_plan, rules = rules)
```

```
## # A tibble: 8 x 2
##   target                  command
##   <chr>                   <chr>
## 1 data                    tidy_data(file_in('data/default of credit card ~
## 2 splits                  split_data(data)
## 3 prepped_orig            prep_data(splits, 'orig')
## 4 baked_orig              bake_data(splits, prepped_orig)
## 5 model_orig_rf           run(def_model, default ~ ., baked_orig$train, l~
## 6 model_orig_knn          run(def_model, default ~ ., baked_orig$train, l~
## 7 drake_target_1_orig_rf  "save_results(model_orig_rf, \"orig\", baked_or~
## 8 drake_target_1_orig_knn "save_results(model_orig_knn, \"orig\", baked_o~
```

`knn` should really be ran on standardized data. Again using the wildcards, we can use the `type__` wildcard to specify that the data should also be standardized:

```
rules = list(type__ = c("orig", "zscore"), method__ = c("rf", "knn"))
evaluate_plan(default_plan, rules = rules)
```

```
## # A tibble: 14 x 2
##    target                   command
##    <chr>                    <chr>
##  1 data                     tidy_data(file_in('data/default of credit ca~
##  2 splits                   split_data(data)
##  3 prepped_orig             prep_data(splits, 'orig')
##  4 prepped_zscore           prep_data(splits, 'zscore')
##  5 baked_orig               bake_data(splits, prepped_orig)
##  6 baked_zscore             bake_data(splits, prepped_zscore)
##  7 model_orig_rf            run(def_model, default ~ ., baked_orig$train~
##  8 model_orig_knn           run(def_model, default ~ ., baked_orig$train~
```

```
##  9 model_zscore_rf           run(def_model, default ~ ., baked_zscore$tra~
## 10 model_zscore_knn          run(def_model, default ~ ., baked_zscore$tra~
## 11 drake_target_1_orig_rf    "save_results(model_orig_rf, \"orig\", baked~
## 12 drake_target_1_orig_knn   "save_results(model_orig_knn, \"orig\", bake~
## 13 drake_target_1_zscore_rf  "save_results(model_zscore_rf, \"zscore\", b~
## 14 drake_target_1_zscore_knn "save_results(model_zscore_knn, \"zscore\", ~
```