Logistic LASSO Vignette

Dionysius Indraatmadja (1003024416)

02/12/2020

Preface

In this vignette we will outline the process for fitting data to a L1-regularized logistic regression model, and using the model to predict new data, in a tidymodels workflow. The following examples require the most up-to-date versions of the `tidymodels` and `tidyverse` packages.

Creating the data

We'll create some data to fit the logistic LASSO on, and split it into training and testing sets. Note that the columns `x` and `w` are numeric columns, `y` is a numeric factor, and `cat` is a character vector. The algorithm requires the dependent variable to be of type factor.

```
source("functions.R")
#> Warning: package 'tidymodels' was built under R version 4.0.3
#> -- Attaching packages --------------------------------------- tidymodels 0.1.2 --
#> v broom     0.7.2      v recipes   0.1.15
#> v dials     0.0.9      v rsample   0.0.8
#> v dplyr     1.0.2      v tibble    3.0.4
#> v ggplot2   3.3.2      v tidyr     1.1.2
#> v infer     0.5.3      v tune      0.1.2
#> v modeldata 0.1.0      v workflows 0.2.1
#> v parsnip   0.1.4      v yardstick 0.0.7
#> v purrr     0.3.4
#> Warning: package 'broom' was built under R version 4.0.3
#> Warning: package 'modeldata' was built under R version 4.0.3
#> Warning: package 'parsnip' was built under R version 4.0.3
#> Warning: package 'recipes' was built under R version 4.0.3
#> Warning: package 'tibble' was built under R version 4.0.3
#> Warning: package 'tune' was built under R version 4.0.3
#> Warning: package 'workflows' was built under R version 4.0.3
#> -- Conflicts ------------------------------------------ tidymodels_conflicts() --
#> x purrr::discard() masks scales::discard()
#> x dplyr::filter()  masks stats::filter()
#> x dplyr::lag()     masks stats::lag()
#> x recipes::step()  masks stats::step()
#> Warning: package 'tidyverse' was built under R version 4.0.3
#> -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
#> v readr   1.3.1     v forcats 0.5.0
#> v stringr 1.4.0
#> -- Conflicts ------------------------------------------ tidyverse_conflicts() --
#> x readr::col_factor() masks scales::col_factor()
#> x purrr::discard()    masks scales::discard()
```

```
#> x dplyr::filter()     masks stats::filter()
#> x stringr::fixed()    masks recipes::fixed()
#> x dplyr::lag()        masks stats::lag()
#> x readr::spec()       masks yardstick::spec()
source("make_tidy.R")
#> Information for `logistic_lasso`
#>  modes: unknown, classification
#>
#>  engines:
#>    classification: fit_logistic_lasso
#>
#>  arguments:
#>    fit_logistic_lasso:
#>       penalty --> lambda
#>
#>  fit modules:
#>                engine           mode
#>    fit_logistic_lasso classification
#>
#>  prediction modules:
#>              mode             engine     methods
#>    classification fit_logistic_lasso class, prob
set.seed(1)
n= 1000
dat <- tibble(x = seq(-3,3, length.out = n),
              w = 3*cos(3*seq(-pi,pi, length.out = n)),
              y = rbinom(n,size = 1, prob = 1/(1 + exp(-w+2*x)) )%>% as.numeric %>% factor,
              cat = sample(c("a","b","c"), n, replace = TRUE)
)
split <- initial_split(dat, strata = c("cat"))
train <- training(split)
test <- testing(split)
```

Let's take a peek at the data:

```
head(train)
#> # A tibble: 6 x 4
#>       x     w y     cat
#>   <dbl> <dbl> <fct> <chr>
#> 1 -3    -3    1     c
#> 2 -2.99 -3.00 1     c
#> 3 -2.99 -3.00 1     a
#> 4 -2.98 -3.00 1     c
#> 5 -2.98 -2.99 1     a
#> 6 -2.97 -2.99 1     b
```

Creating the tidymodels workflow

We define the recipe making sure *not* to use `step_intercept()` in this algorithm since the intercept is treated differently due to the penalization. Since `cat` is nominal data, we use `step_dummy(all_nominal(), -y)` to convert the character values into placeholder numeric values; we exclude `y` because it is the dependent variable.

```
rec <- recipe(y ~ . , data = train) %>%
  step_dummy(all_nominal(), -y) %>% step_zv(all_outcomes()) %>%
  step_normalize(all_numeric(), -y)
```

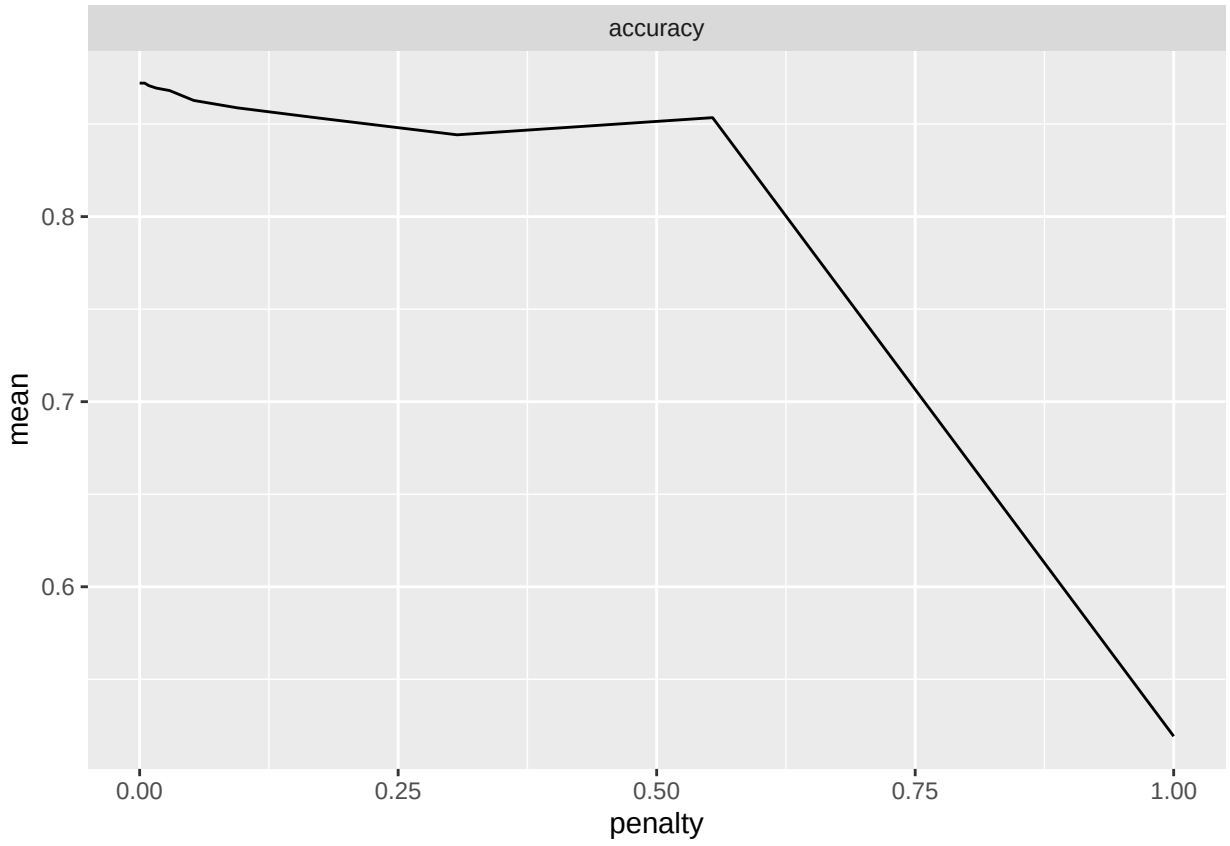
Since we need to specify the penalty value `lambda` to the algorithm, we will set up the tuning of lambda. "Tuning" the penalty means numerically finding the optimal penalty value (and automatically via tidymodels).

```
grid <- grid_regular(penalty(), levels = 40)
spec_tune <- logistic_lasso(penalty = tune()) %>% set_engine("fit_logistic_lasso")
```

Now we're ready to create the workflow and tune the model. Once tuned, we can see how the mean accuracy changes with an increase in the penalty.

```
wf <- workflow() %>% add_recipe(rec) %>% add_model(spec_tune)
folds <- vfold_cv(train)
fit_tune <- wf %>%
  tune_grid(resamples = folds, grid = grid, metrics = metric_set(accuracy))
#> Warning: package 'rlang' was built under R version 4.0.3
#>
#> Attaching package: 'rlang'
#> The following objects are masked from 'package:purrr':
#>
#>     %@%, as_function, flatten, flatten_chr, flatten_dbl, flatten_int,
#>     flatten_lgl, flatten_raw, invoke, list_along, modify, prepend,
#>     splice
#>
#> Attaching package: 'vctrs'
#> The following object is masked from 'package:tibble':
#>
#>     data_frame
#> The following object is masked from 'package:dplyr':
#>
#>     data_frame

fit_tune %>% collect_metrics() %>% ggplot(aes(penalty, mean)) + geom_line() +
  facet_wrap(~.metric)
```



We can see how the penalty affects the accuracy of the fit in the graph above and we have the details of the fits corresponding to each lambda value we tested in the tuning. However, there's a convenient way to find the best penalty value: we'll automatically select the best lambda based on the "accuracy" metric and then save the fit that corresponds to this penalty as `final_fit`. We'll be able to use the final fit to classify/predict new data.

```
penalty_final <- fit_tune %>%
  select_best(metric = "accuracy")

wf_final <- wf %>%
  finalize_workflow(penalty_final)
final_fit <- wf_final %>% fit(train)
```

Using the testing data, we can see if the model predicts the classification data vector accurately.

```
logistic_lasso_pred<-predict(final_fit, new_data = test)
logistic_lasso_pred %>% bind_cols(test %>% select(y)) %>% conf_mat(truth = y, estimate = .pred_class)
#>            Truth
#> Prediction   0   1
#>          0 121  12
#>          1  14 102
```

We found that 26 out of 249 observations were incorrectly classified, which is reasonable!

Conclusion

Logistic regression with L1-regularization of the logistic LASSO helps to exclude predictors with a weak relationship with the dependent variable in classification problems where we have more features (most of them being not meaningful) than observations. The tidymodels workflow standardizes the data preparation and formatting which makes it easier to fit data using different models; or to fit a model to different data sets.

Looking at our final fit:

```
final_fit
#> == Workflow [trained] ==========================================================
#> Preprocessor: Recipe
#> Model: logistic_lasso()
#>
#> -- Preprocessor ----------------------------------------------------------------
#> 3 Recipe Steps
#>
#> * step_dummy()
#> * step_zv()
#> * step_normalize()
#>
#> -- Model ----------------------------------------------------------------------
#> $success
#> [1] TRUE
#>
#> $passed
#> intercept         x         w     cat_b     cat_c
#>      TRUE      TRUE      TRUE      TRUE      TRUE
#>
#> $iter
#> [1] 18
#>
#> $intercept
#> intercept
#> 0.2114557
#>
#> $beta
#>          x          w      cat_b      cat_c
#> -3.3211355  1.8528749  0.2433500  0.1680604
#>
#> $lambda
#> [1] 1e-10
#>
#> $fct_levels
#> [1] "0" "1"
```

We see that we can access the factor level names, coefficients, or penalty from the final fit. For example, we can find the original factor level names before they were converted to numeric values in the algorithm:

```
fct_names <- final_fit$fit$fit$fit$fct_levels
fct_names
#> [1] "0" "1"
```

This can be used to assign the original class names to the newly predicted data.