Iteratively Reweighted Least Squares

Jin Seo Jo

We will use two standard metapackages

```r
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.1     v dplyr   1.0.5
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```r
library(tidymodels)
```

```
## -- Attaching packages -------------------------------------- tidymodels 0.1.3 --

## v broom        0.7.6      v rsample      0.0.9
## v dials        0.0.9      v tune         0.1.5
## v infer        0.5.4      v workflows    0.2.2
## v modeldata    0.1.0      v workflowsets 0.0.2
## v parsnip      0.1.5      v yardstick    0.0.8
## v recipes      0.1.16

## -- Conflicts ----------------------------------------- tidymodels_conflicts() --
## x scales::discard() masks purrr::discard()
## x dplyr::filter()   masks stats::filter()
## x recipes::fixed()  masks stringr::fixed()
## x dplyr::lag()      masks stats::lag()
## x yardstick::spec() masks readr::spec()
## x recipes::step()   masks stats::step()
## * Use tidymodels_prefer() to resolve common conflicts.
```

Implement the fit

A feature of tidymodels is that it requires the response for a classification problem be a `factor`. That means instead of `y` begin a vector of zeros and ones, it is a factor with 2 levels. So we need to make sure we don't lose the factor information. In this case the easiest thing is to just save the names of the factor levels and store them with the output.

```r
fit_IRLS <- function(x, y, beta0 = NULL, tol = 1e-6, max_iter = 1000){
  n <- dim(x)[1]
  p <- dim(x)[2]

  if (is.null(beta0)) {
    beta0 <- rep(0,p)
  }

  ## Process the factor to be 0/1
  ## Make sure you save the names of the factor levels so we can
  ## use them in the predictions
  fct_levels <- levels(y)
  y <- as.numeric(y) - 1

  beta <- beta0
  x_beta0 <- (x %*% beta0) %>% as.numeric
  p <- 1/(1 + exp(-x_beta0))

  for (iter in 1:max_iter) {
    w <- p * (1 - p)
    z <- x_beta0 + (y - p)/w

  ## We could solve the weighted least squares problem directly (and add the
  # names)
  # beta <- solve(t(x) %*% diag(w) %*% x, t(x) %*% (w * z)) %>% as.numeric
  # or we could use lm (remembering to remove the automatic intercept!)
  beta <- lm(z ~ -1 + x, weights = w) %>% coef
  names(beta) <- colnames(x)

  ## now to stop (see below)
  ## Compute gradient (which also computes p for the next step of the algorithm)
  x_beta0 <- (x %*% beta) %>% as.numeric
  p <- 1/(1 + exp(-x_beta0))

  grad <- t(x) %*% (y - p)

  if ( sqrt(sum(grad^2)) < tol) {
    return(
      list(beta = beta, fct_levels = fct_levels,
           iter = iter, converged = TRUE, error = sqrt(sum(grad^2)))
      )
    }

  ## Othewise we go again
  ## We don't need to do anything here - we've already computed x_beta0 and p
  ## for the next iteration
  }
  warning(paste("Method did not converge in", max_iter, "iterations", sep = " "))
  return(
    list(beta = beta, fct_levels = fct_levels,
         iter = iter, converged = FALSE, error = sqrt(sum(grad^2)))
  )
}
```

There are two interesting things here:
1. There are two ways to solve the weighted least squares equations: either solve the normal equations or use `lm` with `weights` option. Both are equivalent 2. To stop the IRLS algorithm, we check to see if the gradient is sufficiently small. We get

$$\frac{\partial L}{\partial \beta_j} = \sum_{i=1}^{n} (y_i - p(x_i)) x_{ij}$$

Note that the $p(x_i)$ are computed using the *most recent* value of β. This is also the vector `p` that we will need in the next iteration (when the most recent value of β is now the old value). We can write out the entire vector by remembering that $\sum_{i=1}^{n} x_{ij} y_i = X^T y$, which gives

$$\nabla L = X^T (y - p)$$

.

Implement the predict function

Recall that, for logistic regression we predict

$$\hat{y_i} = \begin{cases} 1, & \text{if } {x_i}^T \beta \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

The easiest way to do this is to remember that `TRUE/FALSE` converts to `1/0`.
For example

```r
as.numeric(c(TRUE, FALSE, TRUE))
```

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

This leads to a particularly simple prediction function. The only thing that stops this from being a one line function is we need to convert back to the correct factor levels.

```r
pred_IRLS <- function(fit, new_x) {
  numeric_pred <- (new_x %*% fit$beta >= 0) %>%  as.numeric
  return(fit$fct_levels[numeric_pred + 1] %>% factor)
}
```

Make it work with tidymodels

Remember that `mode = "classification"` now.

```r
IRLS <- function(mode = "classification") {

  new_model_spec("IRLS",
                 args = NULL,
                 mode = mode,
                 eng_args = NULL,
                 method = NULL,
                 engine = NULL)
}

set_new_model("IRLS")
set_model_mode(model = "IRLS", mode = "classification")
set_model_engine("IRLS",
  mode = "classification",
  eng = "fit_IRLS"
)
set_dependency("IRLS", eng = "fit_IRLS", pkg = "base")

set_encoding(
  model = "IRLS",
  eng = "fit_IRLS",
  mode = "classification",
  options = list(
    predictor_indicators = "traditional",
    compute_intercept = TRUE,
    remove_intercept = FALSE,
    allow_sparse_x = FALSE
  )
)
show_model_info("IRLS")
```

```
## Information for `IRLS`
##  modes: unknown, classification
##
##  engines:
##    classification: fit_IRLS
##
##  no registered arguments.
##
##  no registered fit modules.
##
##  no registered prediction modules.
```

There are two differences here:
- Because there is no penalty, there's no reason to treat the intercept differently, so we no longer need to remove it from `x`. This means we set `remove_intercept = FALSE`.
- There are no extra parameters in this model so we don't need to do `set_model_arg`.
Now we need to add the fit and the predict methods:

```r
set_fit(
  model = "IRLS",
  eng = "fit_IRLS",
  mode = "classification",
  value = list(
    interface = "matrix",
    protect = c("x", "y"),
    func = c(fun = "fit_IRLS"),
    defaults = list()
  )
)
```

```r
set_pred(
  model = "IRLS",
  eng = "fit_IRLS",
  mode = "classification",
  type = "class",
  value = list(
    pre = NULL,
    post = NULL,
    func = c(fun = "pred_IRLS"),
    args = list(
      fit = expr(object$fit),
      new_x = expr(as.matrix(new_data[, names(object$fit$beta)]))
    )
  )
)
```

The only difference here is that the `type` argument for `set_pred` is set to `"class"` to indicate that we are doing classification.

Check on data

```r
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)

rec <- recipe(y ~ . , data = train) %>%
  step_dummy(all_nominal(), -y) %>%
  step_zv(all_outcomes()) %>%
  step_normalize(all_numeric(), -y) %>% # don't normalize !
  step_intercept() ## This is always last

spec <- IRLS() %>% set_engine("fit_IRLS")

fit <- workflow() %>% add_recipe(rec) %>% add_model(spec) %>% fit(train)

predict(fit, new_data = test) %>%
  bind_cols(test %>% select(y)) %>%
  conf_mat(truth = y, estimate = .pred_class)
```

```
##             Truth
## Prediction   0   1
##           0 112  25
##           1   8 104
```

Not bad at all. But we should check that we got the answers correct by comparing with `glm`:

```r
# make the data
ddat<- rec %>% prep(train) %>% juice

ff<-glm(y ~ -1 + ., family = "binomial", data =ddat)
ff %>% tidy %>% select(term, estimate) %>%
  mutate(IRLS_estimate = fit$fit$fit$fit$beta, err = estimate - IRLS_estimate)
```

```
## # A tibble: 5 x 4
##   term      estimate IRLS_estimate       err
##   <chr>        <dbl>         <dbl>     <dbl>
## 1 intercept   -0.329        -0.329  6.20e-12
## 2 x           -3.96         -3.96  -1.39e-10
## 3 w            2.23          2.23   7.41e-11
## 4 cat_b        0.113         0.113 -6.42e-13
## 5 cat_c       -0.156        -0.156 -1.37e-11
```

And we should check that predict works too:

```r
## Make the test data
test_dat <- rec %>% prep(train) %>% bake(test)
glm_pred <- (predict(ff, test_dat, type = "response") > 0.5) %>%
  as.numeric
preds <- predict(fit, new_data = test) %>%
  bind_cols(glm_pred = glm_pred)
any(preds$.pred_class != preds$glm_pred)
```

```
## [1] FALSE
```

And with that we are done.