Ridge Regression

Jin Seo Jo

Adding a model to parsnip

The `parsnip` package is the member of `tidymodels` universe. It provides a *universal* interface for inference algorithm that allows them to be used interchangeably within the same data analysis workflow.

The abstraction that allows us to do this is the parsnip model and engine.

Consider the following code:

```
library(tidymodels)
library(tidyverse)
spec <- linear_reg(penalty = 0.1) %>%
  set_engine("glmnet")
```

The model here is `linear_reg()` which has the optional `penalty = 0.1`, where the *engine* (aka the thing that does the computation) is the `glmnet` package.

The problem is that no every model/engine combination that we might be interested in is going to be available as a parsnip model. So we will look at ways to add our own models.

The Algorithm

We are going to add our own algorithm for **ridge regression**. This is unnessesary: we could just use the one in `glmnet`, but it's useful to demonstrate exactly how to add a model to parsnip.

The first thing we need is a fitting algorithm:

```
ridge_fit <- function(x, y, lambda){
  p <- dim(x)[2]
  n <- dim(x)[1]

  ## We can centre x without changing beta (but it changes beta0)
  x_bar <- colMeans(x)
  x <- scale(x, center = TRUE, scale = FALSE)

  beta <- solve((t(x)%*%x + n*lambda *diag(p)), t(x)%*%y)
  beta0 <- mean(y) + sum(beta*x_bar) ## correct for non-centred variables

  names(beta) <- colnames(x)

  return(list(beta0 = beta0, beta = beta))
}
```

There are a couple things ere that we should all be aware of:
- The arguments `x` and `y` (lower case) are standardizedand it's convenient to use them

- I made sure that the entries of `beta` havce variable names associated. This is really just for cleanliness
- **Notice the scaling on `lambda`**: This helps keeps everything the same size and makes it easier to use the default values for `grid_regular(penalty())` later down the line. It's what we get if we minimized

$$\frac{1}{2n}\|y - X\beta\|_2^2 + \lambda\|\beta\|_2^2$$

I also need a prediction method:

```
ridge_pred <- function(fit, new_x){
  return((fit$beta0 + new_x %*% fit$beta) %>%
           as.numeric)
}
```

There are a few important things:
- The arguments of this function are arbitrary (aka they can be anything) but they need to work well with the output of `ridge_fit`
- The *input* needs to know about how the return from `ridge_fit` is organized. This will be a `list` that (in this case) only has one column (`beta`)
- The *output* from this function **must be a numeric vector**. This is why `as.numeric` is called: otherwise the result will be a matrix, which will break the code for computing metrics

Registering the model

The first thing we need to do is *register* our model so that we can make a model specification using something equivalent to `linear_reg()`. In this case, we need to write out own function, which we will call `ridge_reg`. This function needs to do two things:
- It needs to specify the *mode* that the model is working in (either "regression" or "classification")
- It needs to specify any arguments that we will need to specify to make the algorithm work

```
ridge_reg <- function(mode = "regression", penalty){
  args <- list(penalty = rlang::enquo(penalty))
  new_model_spec("ridge_reg",
                 args = args,
                 mode = mode,
                 eng_args = NULL,
                 method = NULL,
                 engine = NULL)
}
```

There are a few things here: - The first line contains all of the arguments that we will need to pass to the fit function
- The second line specifies the model using a helper function called `new_model_spec`. The extra arguments that are set a `NULL` needed for flexibility but aren't important for us

A word about `enquo`: It basically says "Please don't evaluate me yet. Just keep me and my associated data around until I'm called later." The technical explanation can be found in (Chapter 20 of Hadley Wickham's book Advanced R)[https://adv- r.hadley.nz/evaluation.html].

With this all in place, we can *register* the model. This is done with a sequence of function calles that sets everything up to make a parsnip model.

```
set_new_model("ridge_reg")
set_model_mode(model = "ridge_reg", mode = "regression")
set_model_engine("ridge_reg", mode = "regression", eng = "ridge_fit")
set_dependency("ridge_reg", eng = "ridge_fit", pkg = "base")
```

That `set_dependency` call is a bit of a hack here. Because the function `ridge_fit` isn't in a package, we just declare it as a `base` function (i.e. not an interesting function)

We also need to tell `parsnip` what the parameters are in the model. In this case we have one, which is called `lambda` in the fit function. It's nicer, however, to follow the guidelines and use the generic name `penalty`. This makes it easier to swap between similar engines and models.

```
set_model_arg(
  model = "ridge_reg",
  eng = "ridge_fit",
  parsnip = "penalty", ## what parsnip will call it
  original = "lambda", ## what we call it
  func = list(pkg = "dials", fun = "penalty"), ## use dials::penalty() to set
  has_submodel = FALSE
)
```

The `func` argument here is required to let `parsnip` know what "type" of parameter it is. In this case, it's a penalty parameter so we can just use the function `dials::penalty()` to provide this information. If we run the function we can see what it does:

```
dials::penalty()
```

```
## Amount of Regularization (quantitative)
## Transformer:  log-10
## Range (transformed scale): [-10, 0]
```

This tells us that the penalty parameter should be manipulated internally on the log10-scale and it's values should be constrained to be in $[10^{-10}, 1]$. **This is sensible range if the covariates are scaled and the penalty is scaled (as it is in our function)**. If the penalty hasn't been scaled, we may need much larger penalties than 1. This is because the diagonal elements of $X^T X$ will be $\approx n$. So for an unscaled penalty, we might want to use `penalty(range = log10(n) + c(-10, 0))` everywhere below where I use `penalty()`.

Finally, we need to tell `parsnip` how to go from a formula to the `x` matrix. This is particularty important when some of our variables are factors or nominal variables.

```
set_encoding(
  model = "ridge_reg",
  eng = "ridge_fit",
  mode = "regression",
  options = list(
    predictor_indicators = "traditional",
    compute_intercept = TRUE,
    remove_intercept = TRUE,
    allow_sparse_x = FALSE
  )
)
```

The key features here are:
- `predictor_indicators = "traditional"`: This tells parsnip to do everything with factors that `lm` would do. This is the option we usually want.
- `compute_intercept = TRUE`: This is the default behaviour for ordinary R funcitons like `lm` and we frequently want this. It also changes exactly how the factors are encoded *see the help for details)
- `remove_intercept = TRUE`: This asks if we want to remove the intercept column for x. This is useful if we fit function treats the intercept differently. For penalized methods, this makes sense: we don't want β_0 to be there with the rest of the β, so we don't want that column in x. But we do have to make sure our fit function does the right thing
- `allow_sparse_x = FALSE`: If we don't know what this means, it should be set to `FALSE`

With all of this in place, we can take a look at how the model is registered:

```
show_model_info("ridge_reg")
```

```
## Information for `ridge_reg`
##  modes: unknown, regression
##
##  engines:
##    regression: ridge_fit
##
##  arguments:
##    ridge_fit:
##       penalty --> lambda
##
##  no registered fit modules.
##
##  no registered prediction modules.
```

This says that we still need to register fit and prediction modules.

Fit and predictio modules

So now we need to finish this off by telling `parsnip` how to fit the model and do prediction. This is the **important** part of the code, as it's going to link us to our bespoke fitting and prediction functions.

The first thing we will do is register the fit function:

```
set_fit(
  model = "ridge_reg",
  eng = "ridge_fit",
  mode = "regression",
  value = list(
    interface = "matrix",
    protect = c("x", "y"),
    func = c(fun = "ridge_fit"),
    defaults = list()
  )
)
```

The first three entries here are standard (what's the model, which engine are we using, and what mode are we using it in), but the final one is important. Value is a `list` that contains all of the information `parsnip` needs to know about how to fit the model.

- `interface = "matrix"`: This matches what we are expecting to pass to our fit function (it takes a matrix of features and a vector of data). Other options are `"formula"`, for when your function uses the formula interface and `"data.frame"` for when it expects a `data.frame`.
- `protect = c("x", "y")`: These are the arguments. We do not want the user to be able to set these: they should be set directly by `parsnip/tidymodels`.
- `func = c(fun = "ridge_fit")`: The name of the fit function. (If it is in a package names `package` the command should be `c(pkg = "package", fun = "ridge_fit")`).
- `defaults = list()`: If any of the parameteres have defaults, they can go here.

Finally we can add the predcition module. This is a little bit more tedious becuase we have to make sure everyhting is organized correctly. The key thing to know is that the `parsnip` structure will give you two things:
- `object`: The outcome of the ift procedure organized nicely. It is a `list` and the important element is `object$fit`, which contains the output of the fit function.
- `new_data`: The data that needs to be predicted. It's a `data.frame`.

```
set_pred(
  model = "ridge_reg",
  eng = "ridge_fit",
  mode = "regression",
  type = "numeric",
  value = list(
    pre = NULL,
    post = NULL,
    func = c(fun = "ridge_pred"),
    args = list(
      fit = expr(object$fit),
      new_x = expr(as.matrix(new_data[, names(object$fit$beta)]))
    )
  )
)
```

There are several things happening here after our friendly "which model is this" arguments:
- `type = "numeric"`: This is what sort of prediction we are doing. If this was a classification problem, we would use `type = "class"`.
- `pre = NULL`: A function that does things to the data before the prediction function is run. We won't need this.
- `post = NULL`: A function that does things to the prediction function output before it can be used by the rest of `tidymodels`. We won't need this.
- `func = c(fun = "ridge_pred")`: what's the function that does the prediction.
- `arg = list(...)`: These are the arguments that are passed to the prediction functino. In this case there are two (`fit` and `new_x`). The right hand side are what's passed to the function.

What does `expr` do? This is a lot like `enquo` with one key difference: we use enquo to pass an argument *through* a function. For instance, where we used it above we sent the argument `penalty` from the list of function arguments into a different function without ever doing anything to it. On the other hand, `expr` is used when we are making a new expression that we don't want to evaluate yet. So the lines

```
func = c(fun = "ridge_pred"),
args = list(
  fit = expr(object$fit),
  new_x = expr(as.matrix(new_data[, names(object$fit$beta)]))
)
```

are telling us that down the road somewhere, `parsnip` should call the function

```
ridge_pred(object$fit, as.matrix(new_data[, names(object$fit$beta)]))
```

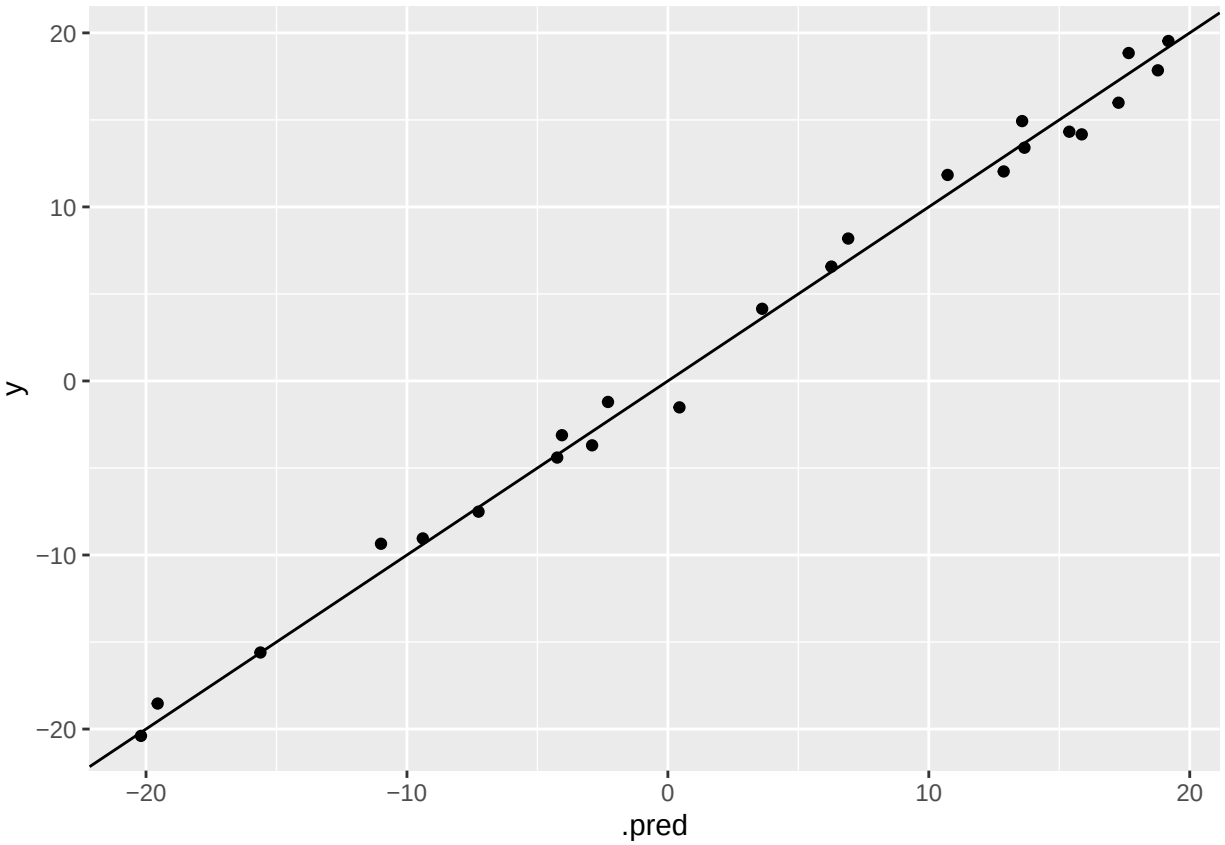
If we didn't wrap it in `expr()` R would try to evaluate the `object$fit` and `as.matrix(new_data[, names(object$fit$beta)])` immediately and *this would throw an error* because the variables object or `new_data` don't exist yet.

What's with new_x?? This is some bookkeeping code that makes sure that we only get the parts of x that we want. In particular, if `new_data` is training data it will have the response in it and we want to make sure that's gone! So the code subsets `new_data` to make sure it's only the columns that correspond to a `beta` that are passed to the function. If we didn't do this we might end up with a dimension error in the matrix-vector mulitiplcation.

So now we are done.

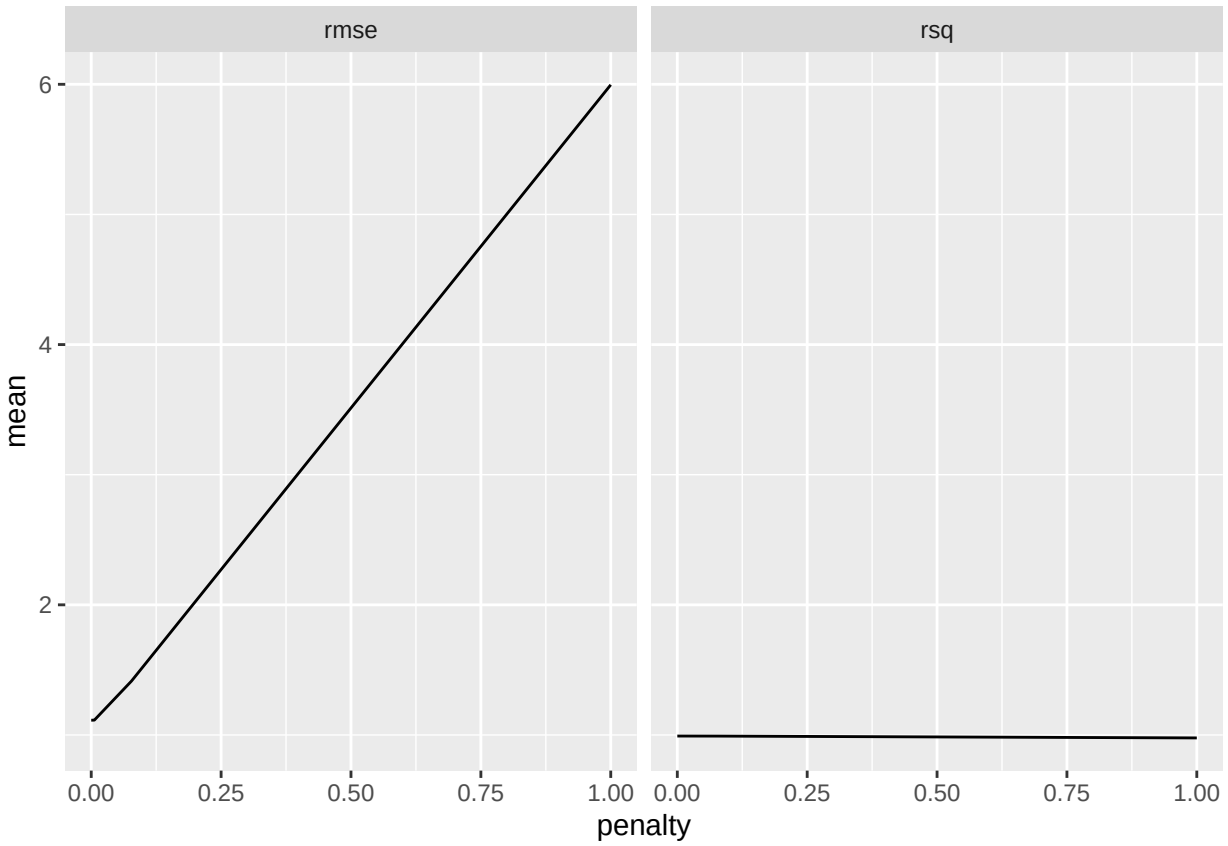
Let's fit some data:

```
n = 100

dat <- tibble(x = seq(-20,20, length.out = n),
              w = cos(seq(-pi,pi, length.out = n)),
              y = rnorm(n, x + w),
              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()) %>%
  step_zv(all_outcomes()) %>%
  step_normalize(all_numeric(), -y) # don't normalize y!

spec <- ridge_reg(penalty = 0.001) %>% set_engine("ridge_fit")

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

predict(fit, new_data = test) %>%
  bind_cols(test %>% select(y)) %>%
  ggplot(aes(.pred,y)) + geom_point() + geom_abline()
```



Now let's try to tune the model! (Something small will go wrong) Because this is a very small model, it most likely won't use very much penalization.

```
## Try it with some tuning

grid <- grid_regular(penalty(), levels = 10)

spec_tune <- ridge_reg(penalty = tune()) %>%
  set_engine("ridge_fit")
wf <- workflow() %>% add_recipe(rec) %>% add_model(spec_tune)

folds <- vfold_cv(train)

fit_tune <- wf %>%
  tune_grid(resamples = folds, grid = grid)

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




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

wf_final <- wf %>%
  finalize_workflow(penalty_final)
```

This doesn't work: Well basically `finalize_workflow` calls the function `update` and we don't have a version of `update` that works for a `ridge_reg` model.

We have two options: either update the parameter manually, or write an update function. Here is the update function. It basically creates a new spec that has the final parameter.

Pro R tip: If you need a version of a generic function (like `update`) that works for a specific class of input (like `ridge_reg`) just make a function called generic.class (like `update.ridge_reg`) and that will be called.

```
update.ridge_reg <- function(object, penalty = NULL, ...) {
  if(! is.null(penalty)) {
    object$args <- list(penalty = enquo(penalty))
    }
  new_model_spec("ridge_reg", args = object$args, eng_args = NULL,
                 mode = "regression", method = NULL, engine = object$engine)
}
```

So let's try that again:

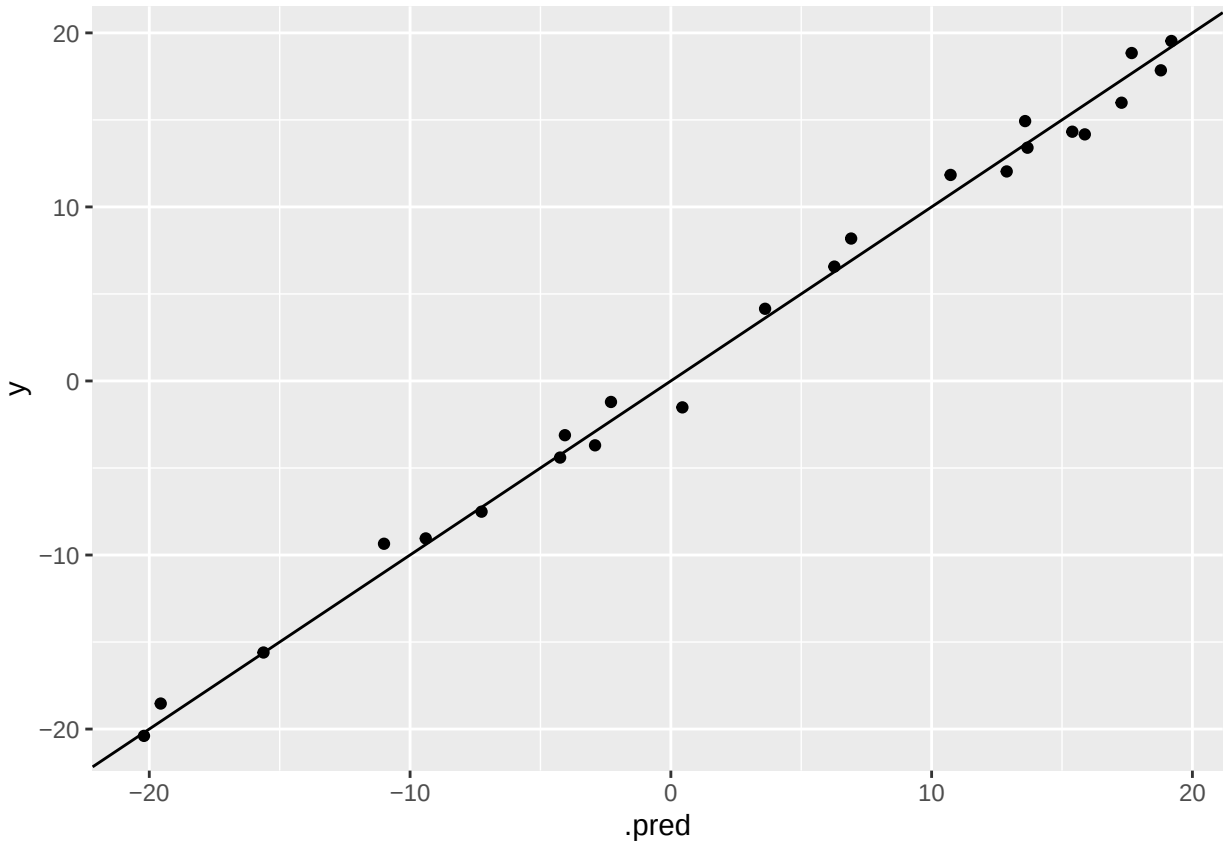
```
penalty_final <- fit_tune %>% select_best(metric = "rmse")

wf_final <- wf %>%
```

```
  finalize_workflow(penalty_final)

wf_final
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: ridge_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
##
## * step_dummy()
## * step_zv()
## * step_normalize()
##
## -- Model -----------------------------------------------------------------------
## Model Specification (regression)
##
## Main Arguments:
##   penalty = 0.000464158883361278
##
## Computational engine: ridge_fit
```

Now let's look at the training error:

```
final_fit <- wf_final %>% fit(train)
predict(final_fit, new_data = test) %>%
  bind_cols(test %>% select(y)) %>%
  ggplot(aes(.pred,y)) + geom_point() + geom_abline()
```


20
10
0
−10
−20
y
−20
−10
0
10
20
.pred