Penalized Regression

Ridge, Lasso, Elastic net

Joe Nese

Week 4, Class 1

Agenda

- Introduce penalized regression
- Specify a model
- Fit a model
- Tune a model
  - regular grids

Penalized Regression

(AKA Regularized Regression)

Let’s revisit linear regression

- What’s good
  - Parsimonious
  - Interpretable results
  - Coefficients are unbiased (given standard assumptions)
    - Because they minimize the sum-of-squared errors (SSE)
  - Lowest variance (of all unbiased linear techniques)

Let's revisit linear regression

- What's not so good
  - Sensitive to highly correlated predictors – multicollinearity
  - Including irrelevant predictors may hurt model performance
  - Model fit is influenced by "outliers" because it wants to minimize SSE
  - Although we can model nonlinearity by adding terms to the model (x^2 or log(x))
    - this may not capture the relationship between predictors and outcome
    - adds predictors to the model (problematic with many predictors fewer observations)

Penalized Regression

- OLS regression coefficients are unbiased because the model minimizes SSE
- But it turns out that adding a little bias to the coefficients can substantially decrease variance, resulting in a smaller MSE and better prediction of unseen data
- How to add bias to the coefficients?
- Add a ***penalty*** to the SSE if the coefficients become too large
  - Basically: penalize the model for coefficients as they move away from zero
  - As a regression coefficient grows large, the penalty must also increase to enforce the minimization of SSE
  - In order to have a large coefficient, a predictor will need to have a large impact on the model fit

Penalized Regression

- How does a penalty help?
  - Shrinking our coefficients toward zero reduces the model's variance (think of model where all coefficients are equal to zero – no variance)
  - The optimal penalty will balance reduced variance with increased bias
  - Particularly useful for dealing with multicollinearity
    - As multicollinearity increases, the estimated regression coefficients are inflated and become unstable

Penalized Regression Models

1) ridge regression (Hoerl, 1970)
2) lasso (Tibshirani, 1996)
3) elastic net (Aou & Hastie, 2005)

- AKA
  - Regularized Regression
  - Shrinkage methods

Ridge Regression

$$SSE_{\underline{L_2}} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{P}\beta_j^2$$

squared coefficients

penalty

- Penalize the model for coefficients as they move away from zero **unless** there is a proportional reduction in the SSE
- L_2 penalty = second-order penalty (squared coefficients)
- λ = 0 = linear regression
- As the penalty (λ) increases, the coefficients shrink toward 0 (at different rates)
- A new set of coefficients is produced for each value of λ

Penalized Regression

- Scale matters
- The units of the predictors can substantially affect results
- The scale of predictors doesn't affect SSE, but does affect the coefficients
  - Think of coefficient interpretation for *meters* vs. *kilometers*
  - Ridge regression will pay a larger penalty for *meters*
- So we need to put all predictors on the same scale prior to analysis
- Center and scale (standardize) all predictors

  (x - mean(x)) / sd(x)

Ridge Regression

- Ridge penalty is mostly associated with addressing collinearity between predictors
- Shrinks the coefficients of correlated predictors toward each other
    - rather than allowing one to be wildly positive and the other wildly negative
- Many less-important predictors get pushed toward zero which helps identify the important predictors in our data
- Shrinks coefficients toward 0, but will never equal 0, no matter how large the penalty
- A coefficient equal to 0 would, of course, be dropped from the model
- That would be automatic feature selection!
- That would be nice!
- lasso models do this!
    - Least Absolute Shrinkage and Selection Operator

lasso - Least Absolute Shrinkage and Selection Operator

$$SSE_{L_1} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{P}|\beta_j|$$

penalty

absolute coefficients

- Penalize the model for coefficients as they move away from zero **unless** there is a proportional reduction in the SSE
- L_1 penalty = absolute coefficients
- As the penalty (λ) increases, the coefficients shrink toward 0 (at different rates)
- Allows coefficients equal to 0

Ridge and lasso

- Both equally penalize overestimating and underestimating a coefficient
- No free lunch

Ridge

$$\lambda \sum_{j=1}^{P} \beta_j^2$$

L_2 penalty

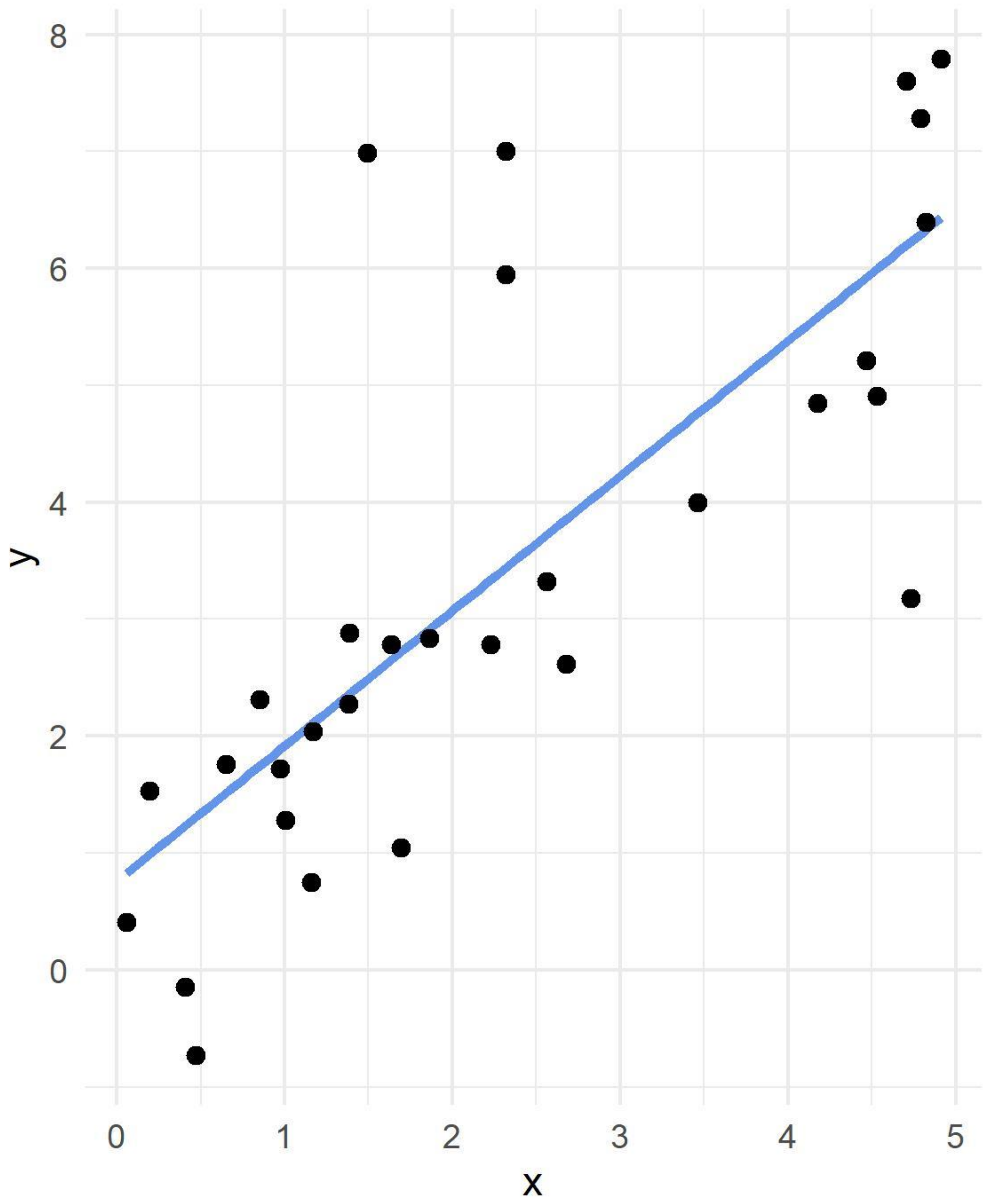
- Larger errors are worse
- Tends to shrinks coefficients of correlated predictors toward each other
  - Extreme example: for *P* identical predictors, each has a coefficient of 1/*P* the size as one modeled by itself
- Helps if you want to keep all predictors in your model and reduce the noise of less influential variables (e.g., smaller data sets with severe multicollinearity)

Lasso

$$\lambda \sum_{j=1}^{P} |\beta_j|$$

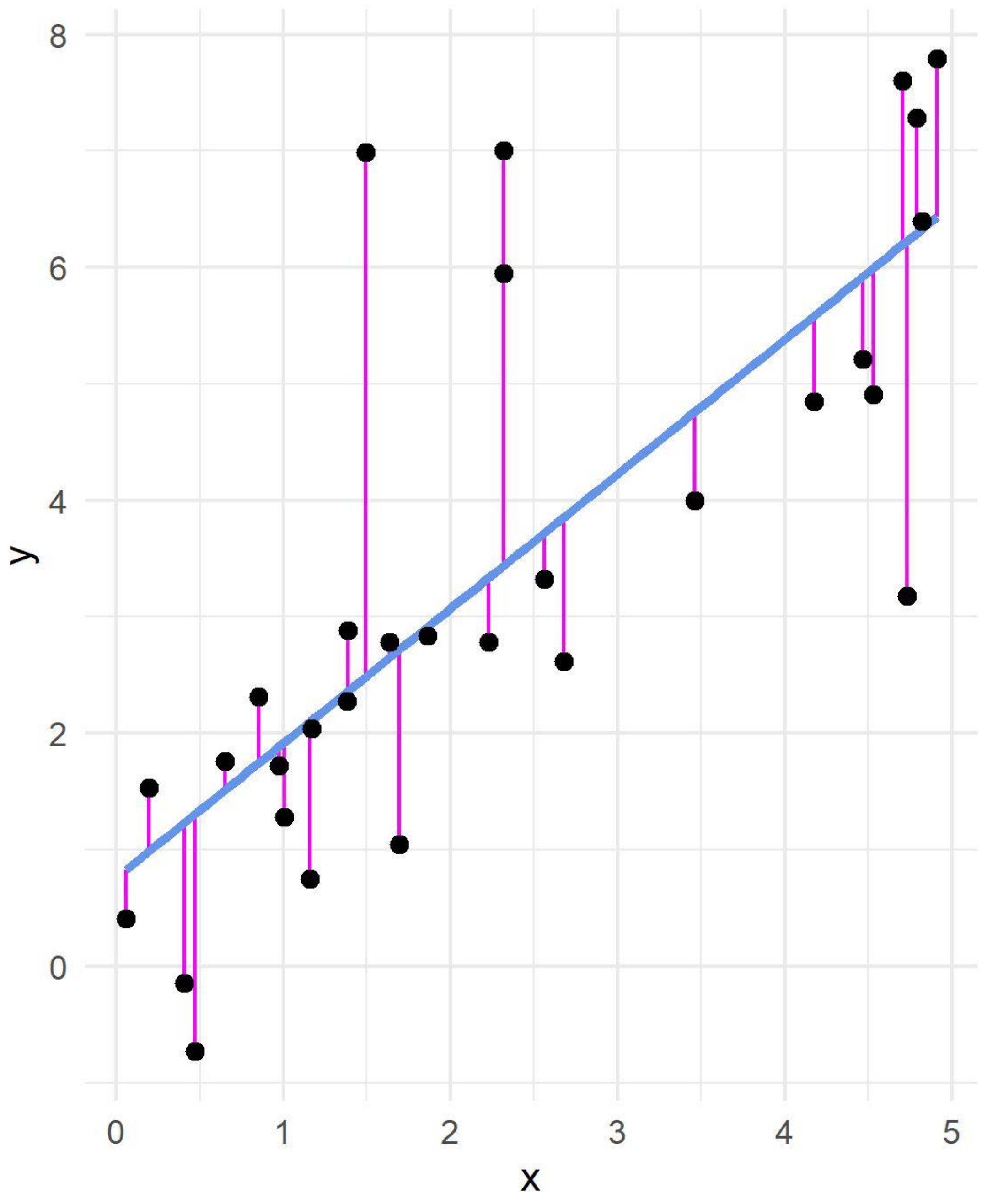
L_1 penalty

- Additional error is equally bad everywhere
- Tends to just choose one predictor and not model the others
  - Extreme example: for *P* identical predictors, will model one predictor and allow coefficient of zero for the rest
- Helps find the predictors with the largest (and most consistent) coefficients in data with many predictors



Code credit to [Ed Rubin]

loss



L_2 loss function

$$\Sigma_i (y_i - \hat{y}_i)^2$$

$(4.5)^2$

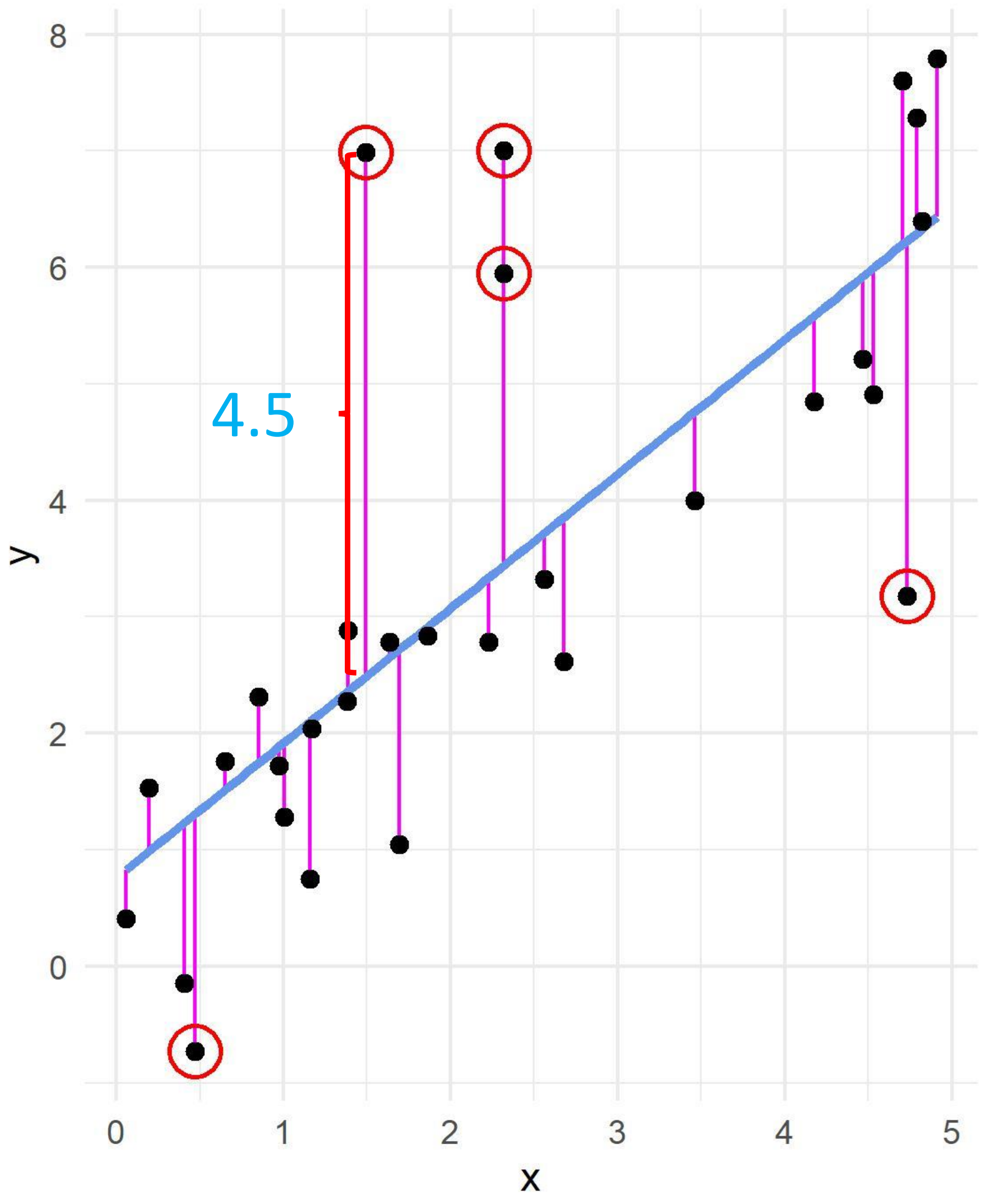
20.25

L_1 loss function

$$\Sigma_i |y_i - \hat{y}_i|$$

$|4.5|$

4.5



Elastic net

$$SSE_{Enet} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{P} \beta_j^2 + \lambda \sum_{j=1}^{P} |\beta_j|$$

- Combines the two types of penalties
- Enables effective regularization with ridge penalty (L_2)
- Offers feature selection with lasso penalty (L_1)
- Better able to handle multicollinearity

Specify a model

Specify the model

- select a model
  - https://www.tidymodels.org/find/parsnip/
  - we will be discussing many different modeling options
- select the engine
  - the package (software) that will be used to fit the model
- select the mode
  - regression or classification

- We're just setting up the framework, we’re not estimating anything yet

select a model

- Welcome to `{parsnip}`!
- List of at least 30 models
  - https://www.tidymodels.org/find/parsnip/
- We will be using the model for linear regression
  - which also allows for penalized regression

```
linear_reg()
```

`set_engine()`

- Used to specify which package will be used to fit the model
- and any arguments specific to that software

- We'll be using `glmnet` (default) for our penalized [regression models]
  - can also use `stan`, `spark`, `keras`

```
set_engine("glmnet")
```

set_mode()

- specify whether the outcome is
  - `set_mode("regression")`
  - `set_mode("classification")`

Specify the model

```
linear_reg() %>%
  set_engine("glmnet") %>%
  set_mode("regression") %>% # redundant; just getting in the habit
```

-or-

```
linear_reg(mode = “regression”) %>% # only option available
  set_engine("glmnet")
```

`linear_reg()`

```
linear_reg(mode = "regression", penalty = NULL, mixture = NULL)
```

`mode` = can only be “regression,” not “classification”

(`logistic_reg` is used for classification)

`penalty` = An non-negative number representing the total amount of regularization. This can be a combination of L1 and L2 (depending on the value of mixture)

`mixture` = A number between zero and one (inclusive) that represents the proportion of L1 regularization (the lasso)

- ridge = `mixture = 0;` no L1
- lasso = `mixture = 1;` completely L1 (and no ridge)
- enet = `0 < mixture < 1;` mixture of L1 (lasso) and ridge (L2)

```r
math <- math <- read_csv(here::here("data", "train.csv"))

# 1 - Initial Split

set.seed(3000)
math_split <- initial_split(math)

math_train <- training(math_split)
math_test  <- testing(math_split)

# 2 - Resample

set.seed(3000)
cv_splits <- vfold_cv(math_train)
```

Before we continue…

- Penalized regression cannot handle missing data
  - Can either delete or impute
  - For simplicity here, we are just going to delete
- We need to center and scale our continuous predictors
- This is part of data preprocessing, or feature engineering
  - "the process of creating representations of data that increase the effectiveness of a model" (Kuhn & Johnson, 2019)
- Very quick preview of next week's topic and the `{recipes}` package
- Center: average is subtracted from the predictor's individual values
  - All predictors will have a mean of zero
- Scale: divide a variable by the standard deviation
  - All predictors have a standard deviation of one

{recipes}

```
penreg_rec <-
  recipe(
    formula = score ~ enrl_grd + econ_dsvntg + lat + lon,
    data = math_train
  ) %>%
  step_naomit(all_predictors(), skip = TRUE) %>%
  step_string2factor(econ_dsvntg) %>%
  step_dummy(econ_dsvntg) %>%
  step_normalize(lat, lon, enrl_grd)
```

{recipes}

```
penreg_rec <-
  recipe(
    formula = score ~ enrl_grd + econ_dsvntg + lat + lon,
    data = math_train
  ) %>%
  step_naomit(all_predictors(), skip = TRUE) %>%
  step_string2factor(econ_dsvntg) %>%
  step_dummy(econ_dsvntg) %>%
  step_normalize(lat, lon, enrl_grd)
```

defines outcome and predictors

{recipes}

```
penreg_rec <-
  recipe(
    formula = score ~ enrl_grd + econ_dsvntg + lat + lon,
    data = math_train
  ) %>%
  step_naomit(all_predictors(), skip = TRUE) %>%
  step_string2factor(econ_dsvntg) %>%
  step_dummy(econ_dsvntg) %>%
  step_normalize(lat, lon, enrl_grd)
```

Catalogs the names and types of each variable
Informs `recipe()` what is numeric and what is nominal

{recipes}

```
penreg_rec <-
  recipe(
    formula = score ~ enrl_grd + econ_dsvntg + lat + lon,
    data = math_train
  ) %>%
  step_naomit(all_predictors(), skip = TRUE) %>%
  step_string2factor(econ_dsvntg) %>%
  step_dummy(econ_dsvntg) %>%
  step_normalize(lat, lon, enrl_grd)
```

drops missing values from all predictors

{recipes}

```
penreg_rec <-
  recipe(
    formula = score ~ enrl_grd + econ_dsvntg + lat + lon,
    data = math_train
  ) %>%
  step_naomit(all_predictors(), skip = TRUE) %>%
  step_string2factor(econ_dsvntg) %>%
  step_dummy(econ_dsvntg) %>%
  step_normalize(lat, lon, enrl_grd)
```

converts strings (“Y”, “N”) to factors

{recipes}

```
penreg_rec <-
  recipe(
    formula = score ~ enrl_grd + econ_dsvntg + lat + lon,
    data = math_train
  ) %>%
  step_naomit(all_predictors(), skip = TRUE) %>%
  step_string2factor(econ_dsvntg) %>%
  step_dummy(econ_dsvntg) %>%
  step_normalize(lat, lon, enrl_grd)
```

Converts nominal data into dummy variables

{recipes}

```
penreg_rec <-
  recipe(
    formula = score ~ enrl_grd + econ_dsvntg + lat + lon,
    data = math_train
  ) %>%
  step_naomit(all_predictors(), skip = TRUE) %>%
  step_string2factor(econ_dsvntg) %>%
  step_dummy(econ_dsvntg) %>%
  step_normalize(lat, lon, enrl_grd)
```

Normalizes (centers and scales); necessary for penalized regression

Could also use:

```
step_center(lat, lon, enrld_grd)
step_scale(lat, lon, enrld_grd)
step_normalize(all_numeric(), -all_outcomes())
```

```r
# 3 - Set Model

## Ridge

mod_ridge <- linear_reg() %>%
  set_engine("glmnet") %>%
  set_mode("regression") %>% # redundant; just setting a habit
  set_args(penalty = .1,       # arbitrarily set the penalty = .1
           mixture = 0)        # specifies a ridge regression model
```

```r
## lasso

mod_lasso <- linear_reg() %>%
  set_engine("glmnet") %>%
  set_mode("regression") %>% # redundant; just setting a habit
  set_args(penalty = .1,     # arbitrarily set the penalty = .1
           mixture = 1)      # specifies a lasso model
```

```r
## Elastic net

mod_enet <- linear_reg() %>%
  set_engine("glmnet") %>%
  set_mode("regression") %>% # redundant; just setting a habit
  set_args(penalty = .1,     # arbitrarily set the penalty = .1
           mixture = .7)     # specifies 70% L1 penalty (lasso)
                             # and 30% L2 penalty (ridge)
```

Fit a model

fit_resamples()

- Fit multiple models via resampling

```
fit_resamples(
  object,
  preprocessor,
  resamples,
  ...,
  metrics = NULL,
  control = control_resamples()
)
```

fit_resamples()

- Fit multiple models via resampling

```
fit_resamples(
  object,
  preprocessor,
  resamples,
  ...,
  metrics = NULL,
  control = control_resamples()
)
```

`parsnip` model specification or a `workflows::workflow()` we'll get to this later

`mod_ridge`

`mod_lasso`

`mod_enet`

fit_resamples()

- Fit multiple models via resampling

```
fit_resamples(
  object,
  preprocessor,
  resamples,
  ...,
  metrics = NULL,
  control = control_resamples()
)
```

`score ~ enrl_grd + econ_dsvntg + lat + lon`

a traditional model formula or a `recipes::recipe()`

`penreg_rec`

fit_resamples()

- Fit multiple models via resampling

```
fit_resamples(
  object,
  preprocessor,
  resamples,
  ...,
  metrics = NULL,
  control = control_resamples()
)
```

A resample rset created from an `rsample` function

`cv_splits`

fit_resamples()

- Fit multiple models via resampling

```
fit_resamples(
  object,
  preprocessor,
  resamples,
  ...,
  metrics = NULL,
  control = control_resamples()
)
```

A `yardstick::metric_set()` or NULL to compute a standard set of metrics

fit_resamples()

- Fit multiple models via resampling

```
fit_resamples(
  object,
  preprocessor,
  resamples,
  ...,
  metrics = NULL,
  control = control_resamples()
)
```

```r
# 4 - Fit the models

## Ridge

fit_ridge <- tune::fit_resamples(
  mod_ridge,
  preprocessor = penreg_rec,
  resamples = cv_splits,
  metrics = yardstick::metric_set(rmse), # default is rmse & rsq
  control = tune::control_resamples(verbose = TRUE,
                                    save_pred = TRUE))
```

This will print to your console the model fitting process by Fold, so you can get an idea of progress and time

```
# 4 - Fit the models

## Ridge

fit_ridge <- tune::fit_resamples(
  mod_ridge,
  preprocessor = penreg_rec,
  resamples = cv_splits,
  metrics = yardstick::metric_set(rmse), # default is rmse & rsq
  control = tune::control_resamples(verbose = TRUE,
                                    save_pred = TRUE))
```

This will print to your console the model fitting process by Fold, so you can get an idea of progress and time

```r
# 4 - Fit the models

## Ridge

fit_ridge <- tune::fit_resamples(
  mod_ridge,
  preprocessor = penreg_rec,
  resamples = cv_splits,
  metrics = yardstick::metric_set(rmse), # default is rmse & rsq
  control = tune::control_resamples(verbose = TRUE,
                                    save_pred = TRUE))
```

This will save the out-of-sample (analysis) predictions for each model evaluated

```
## Ridge

fit_ridge %>%
 tune::collect_metrics()
```

```
# A tibble: 1 x 5
  .metric .estimator  mean     n std_err
  <chr>   <chr>      <dbl> <int>   <dbl>
1 rmse    standard    102.    10   0.351
```

```r
## Ridge

fit_ridge %>%
 tune::collect_metrics(summarize = FALSE)
```

```
# A tibble: 10 x 4
   id     .metric .estimator .estimate
   <chr>  <chr>   <chr>          <dbl>
 1 Fold01 rmse    standard       101.
 2 Fold02 rmse    standard       101.
 3 Fold03 rmse    standard       101.
 4 Fold04 rmse    standard        99.3
 5 Fold05 rmse    standard       103.
 6 Fold06 rmse    standard       103.
 7 Fold07 rmse    standard       102.
 8 Fold08 rmse    standard       103.
 9 Fold09 rmse    standard       101.
10 Fold10 rmse    standard       102.
```

```r
## lasso

fit_lasso <- tune::fit_resamples(
  mod_lasso,
  preprocessor = penreg_rec,
  resamples = cv_splits,
  metrics = yardstick::metric_set(rmse),
  control = tune::control_resamples(verbose = TRUE,
                                    save_pred = TRUE))
fit_lasso %>%
  collect_metrics()
```

```
# A tibble: 1 x 5
  .metric .estimator  mean     n std_err
  <chr>   <chr>      <dbl> <int>   <dbl>
1 rmse    standard    101.    10   0.356
```

```r
## Elastic net

fit_enet <- tune::fit_resamples(
  mod_enet,
  preprocessor = penreg_rec,
  resamples = cv_splits,
  metrics = yardstick::metric_set(rmse),
  control = tune::control_resamples(verbose = TRUE,
                                    save_pred = TRUE))
fit_enet %>%
  collect_metrics()
```

```
# A tibble: 1 x 5
  .metric .estimator  mean     n std_err
  <chr>   <chr>      <dbl> <int>   <dbl>
1 rmse    standard    101.    10   0.356
```

```r
collect_metrics(fit_ridge)
```

```
# A tibble: 1 x 5
  .metric .estimator  mean     n std_err
  <chr>   <chr>      <dbl> <int>   <dbl>
1 rmse    standard    102.    10   0.351
```

```r
collect_metrics(fit_lasso)
```

```
# A tibble: 1 x 5
  .metric .estimator  mean     n std_err
  <chr>   <chr>      <dbl> <int>   <dbl>
1 rmse    standard    101.    10   0.356
```

```r
collect_metrics(fit_enet)
```

```
# A tibble: 1 x 5
  .metric .estimator  mean     n std_err
  <chr>   <chr>      <dbl> <int>   <dbl>
1 rmse    standard    101.    10   0.356
```

Penalized regression

- Thus far we have used `penalty = .1` (λ)
- Choosing a good value for the penalty is very important
  - Too small a penalty and our model is essentially OLS
  - Too large a penalty and we shrink all our coefficients too close to zero
- So how can we find an optimal value?
- Model tuning

Tune a model

regular grids

{tune}

- Facilitates the tuning of hyper-parameters in `tidymodels` packages
- Hyperparameters (tuning parameters) control the complexity of some ML models (and the bias-variance trade-off)
- Hyperparameters cannot be directly estimated from the data
- Some models have **many** tuning parameters (e.g., boosted trees)
- We use cross-validation to find the optimal tuning parameter values with either:
  - grid search - predefined values
  - iterative search - where each iteration finds novel tuning parameter values to evaluate

tune()

- A placeholder for hyper-parameters to be "tuned“

```
ridge_tune_mod <- linear_reg() %>%
  set_engine("glmnet") %>%
  set_args(penalty = tune(),
           mixture = 0)  # specifies a ridge regression model
```

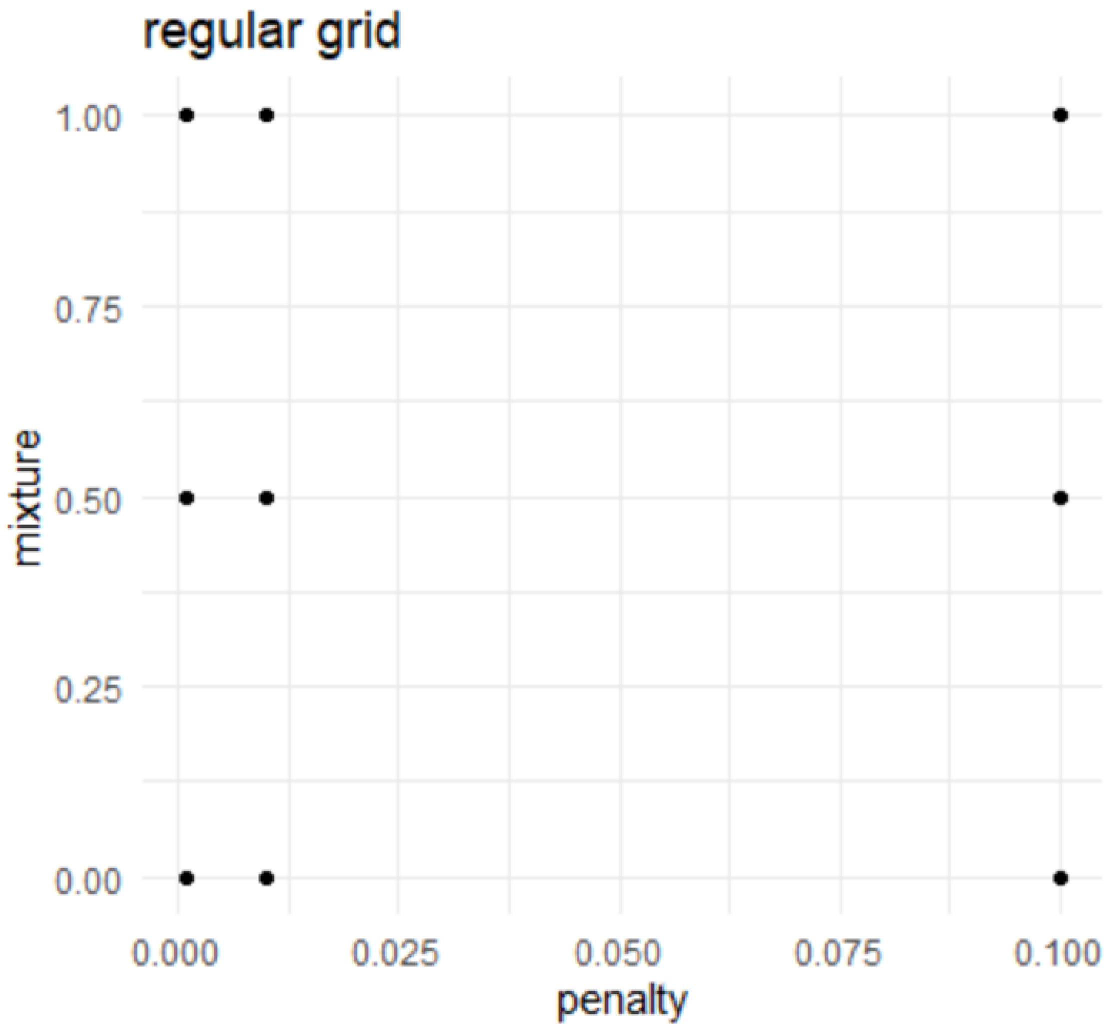
grid search

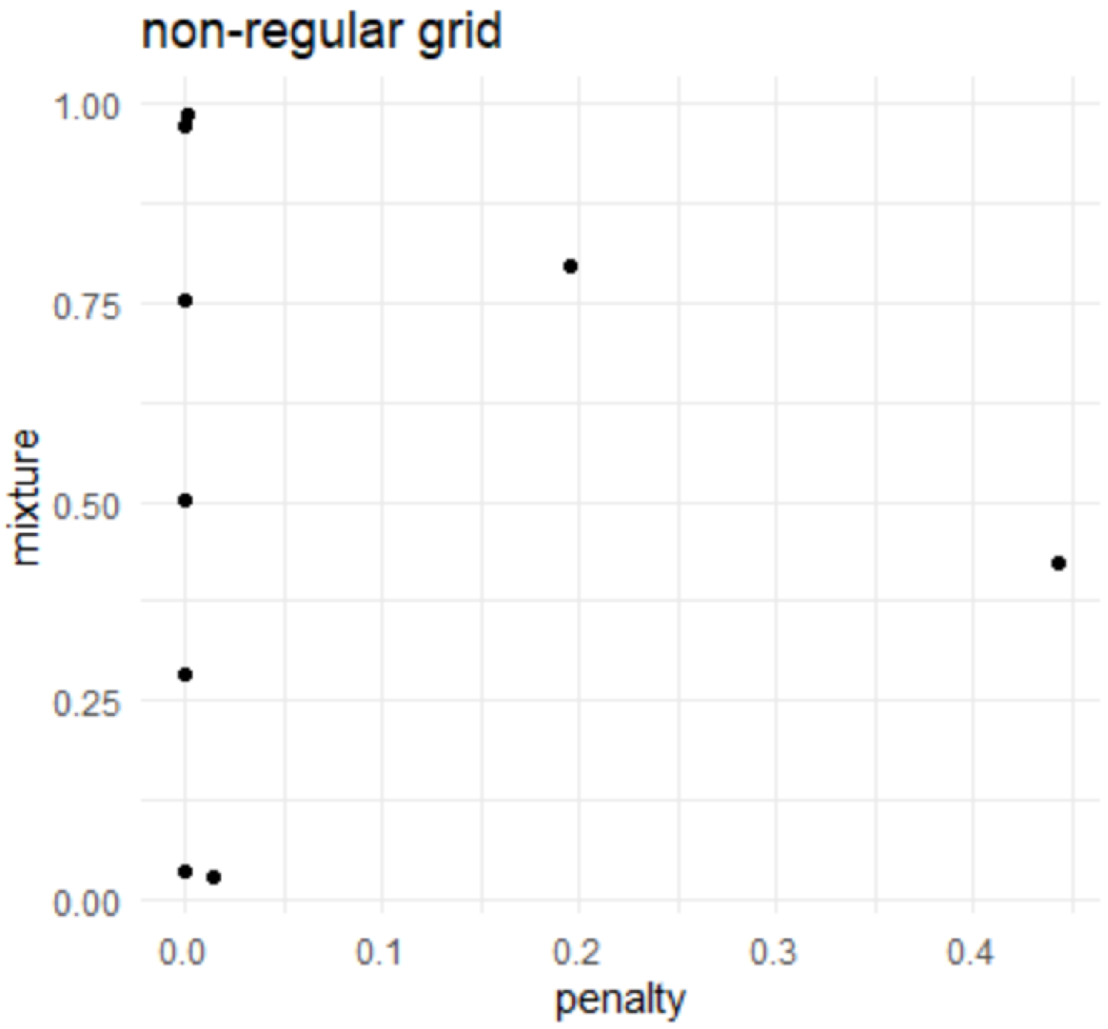
- Set a **pre-defined** set of tuning parameter values and evaluate their performance so that the best values can be used in the final model
    - For models with more than one tuning parameter, the grid is multidimensional
- Using resampling to evaluate each distinct parameter value combination to get estimates of how well each performs
- Calculate results and model performance, and use the “best” tuning parameter combination to fit to the entire training set

Regular grids

- Usually a combination of vectors of tuning parameter values
- The number of values don't have to be the same per parameter
- The values can be regular on a transformed scale (e.g. log-10 for `penalty`)
- Quantitative and qualitative parameters can be combined
- As the number of parameters increases, so does the burden of dimensionality
- Thought to be inefficient but not in all cases
- Bad when performance plateaus over a range of one or more parameters

Kuhn (2019)

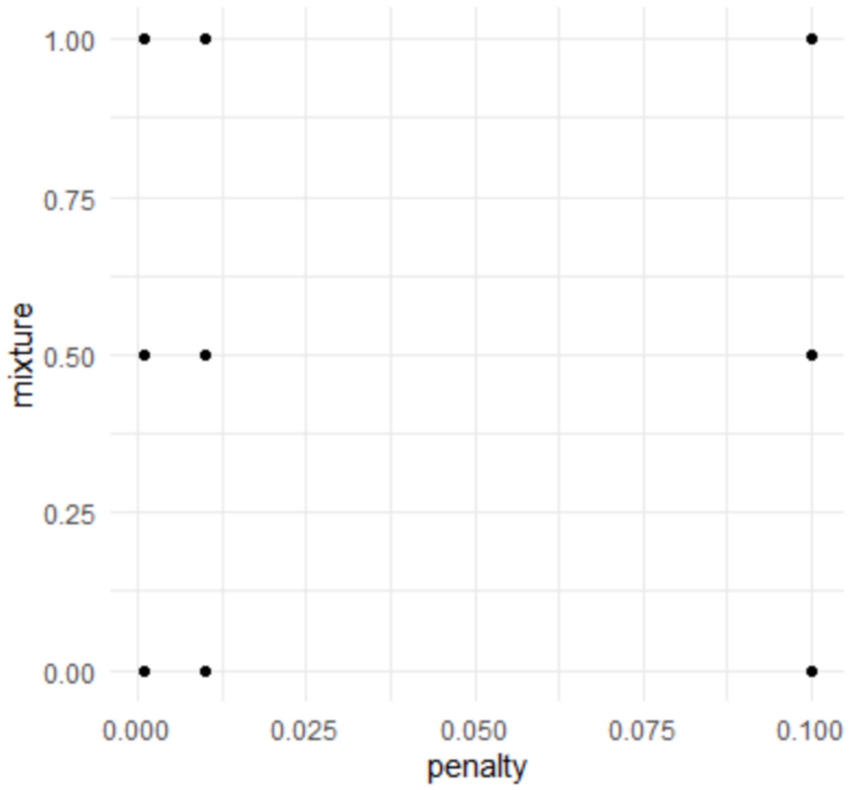

regular grid
1.00
0.75
0.50
0.25
0.00
mixture
0.000
0.025
0.050
0.075
0.100
penalty


non-regular grid
1.00
0.75
0.50
0.25
0.00
mixture
0.0
0.1
0.2
0.3
0.4
penalty

regular grid example

```
base::expand.grid(
  penalty = c(.001, .01, .1),
  mixture = c(0, .5, 1))
```

```
  penalty mixture
1   0.001     0.0
2   0.010     0.0
3   0.100     0.0
4   0.001     0.5
5   0.010     0.5
6   0.100     0.5
7   0.001     1.0
8   0.010     1.0
9   0.100     1.0
```



grid_regular()

```
penalty() # from the {dials} package
Amount of Regularization  (quantitative)
Transformer:  log-10
Range (transformed scale): [-10, 0]
```

```
grid_regular(penalty())
# A tibble: 3 x 1
       penalty
         <dbl>
1 0.0000000001
2 0.00001
3 1
```

```
grid_regular(penalty(), levels = 10)

# A tibble: 10 x 1
          penalty
            <dbl>
 1 0.0000000001
 2 0.00000000129
 3 0.0000000167
 4 0.000000215
 5 0.00000278
 6 0.0000359
 7 0.000464
 8 0.00599
 9 0.0774
10 1
```

tune_grid()

- A version of `fit_resamples()` that performs a grid search for the best combination of tuned hyperparameters

```
tune_grid(
  object,
  preprocessor,
  resamples,
  ...,
  param_info = NULL,
  grid = 10,
  metrics = NULL,
  control = control_grid()
)
```

`tune_grid()`

- A version of `fit_resamples()` that performs a grid search for the best combination of tuned hyperparameters

```
tune_grid(
  object,
  preprocessor,
  resamples,
  ...,
  param_info = NULL,
  grid = 10,
  metrics = NULL,
  control = control_grid()
)
```

a `{parsnip}` model or `workflow()`

`tune_grid()`

- A version of `fit_resamples()` that performs a grid search for the best combination of tuned hyperparameters

```
tune_grid(
  object,
  preprocessor,
  resamples,
  ...,
  param_info = NULL,
  grid = 10,
  metrics = NULL,
  control = control_grid()
)
```

A traditional model formula or a `recipe()`

tune_grid()

- A version of `fit_resamples()` that performs a grid search for the best combination of tuned hyperparameters

```
tune_grid(
  object,
  preprocessor,
  resamples,
  ...,
  param_info = NULL
  grid = 10,
  metrics = NULL,
  control = control_
)
```

Either:
- a data frame of tuning combinations (have columns for each parameter being tuned and rows for tuning parameter candidates)
- a positive integer (number of candidate parameter sets to be created automatically)

```r
ridge_tune_mod <- linear_reg() %>%
  set_engine("glmnet") %>%
  set_args(penalty = tune(),
           mixture = 0)

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

ridge_tune_mod_results <- tune::tune_grid(
  ridge_tune_mod,
  preprocessor = penreg_rec,
  resamples = cv_splits,
  grid = penreg_grid,
  metrics = yardstick::metric_set(rmse),
  control = tune::control_resamples(verbose = TRUE,
                                    save_pred = TRUE)
)
```

Results: Tuned ridge regression

```
ridge_tune_mod_results %>%
   collect_metrics()
```

```
# A tibble: 10 x 6
         penalty .metric .estimator  mean     n std_err
           <dbl> <chr>   <chr>      <dbl> <int>   <dbl>
 1 0.0000000001  rmse    standard    102.    10   0.351
 2 0.00000000129 rmse    standard    102.    10   0.351
 3 0.0000000167  rmse    standard    102.    10   0.351
 4 0.000000215   rmse    standard    102.    10   0.351
 5 0.00000278    rmse    standard    102.    10   0.351
 6 0.0000359     rmse    standard    102.    10   0.351
 7 0.000464      rmse    standard    102.    10   0.351
 8 0.00599       rmse    standard    102.    10   0.351
 9 0.0774        rmse    standard    102.    10   0.351
10 1             rmse    standard    102.    10   0.351
```

Results: Tuned ridge regression

```
ridge_tune_mod_results %>%
   collect_metrics(summarize = FALSE)
```

```
# A tibble: 100 x 6
   id           penalty .metric .estimator .estimate .config
   <chr>          <dbl> <chr>   <chr>          <dbl> <chr>
 1 Fold01 0.0000000001  rmse    standard        101. Model01
 2 Fold01 0.00000000129 rmse    standard        101. Model02
 3 Fold01 0.0000000167  rmse    standard        101. Model03
 4 Fold01 0.000000215   rmse    standard        101. Model04
 5 Fold01 0.00000278    rmse    standard        101. Model05
 6 Fold01 0.0000359     rmse    standard        101. Model06
 7 Fold01 0.000464      rmse    standard        101. Model07
 8 Fold01 0.00599       rmse    standard        101. Model08
 9 Fold01 0.0774        rmse    standard        101. Model09
10 Fold01 1             rmse    standard        101. Model10
# ... with 90 more rows
```

Let’s make a regular grid for our enet model

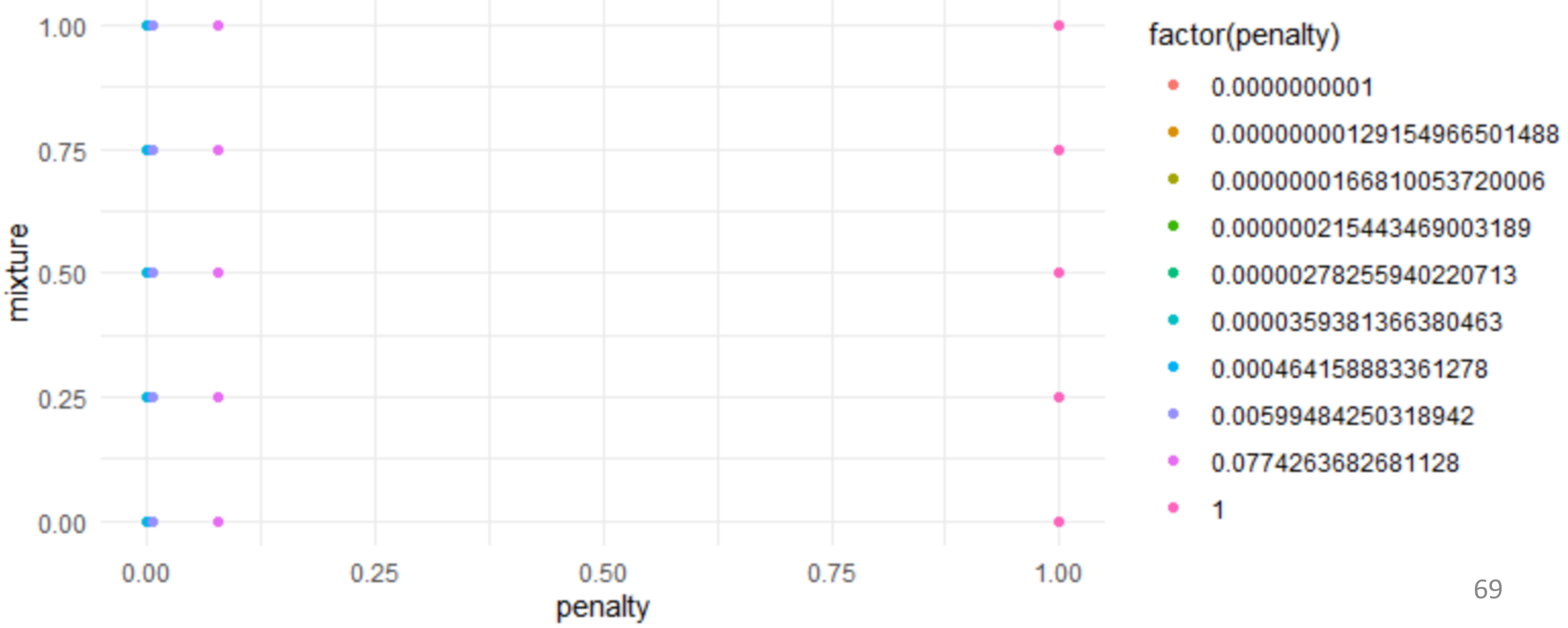
```
(enet_params <- parameters(penalty(), mixture()))
Collection of 2 parameters for tuning
      id parameter type object class
 penalty        penalty    nparam[+]
 mixture        mixture    nparam[+]

enet_grid <- grid_regular(enet_params, levels = c(10, 5))
# A tibble: 50 x 2
         penalty mixture
           <dbl>   <dbl>
 1 0.0000000001       0
 2 0.00000000129      0
 3 0.0000000167       0
 4 0.000000215        0
 5 0.00000278         0
 6 0.0000359          0
 7 0.000464           0
 8 0.00599            0
 9 0.0774             0
10 1                  0
# ... with 40 more rows
```

This is 50 models per fold = 500 models!

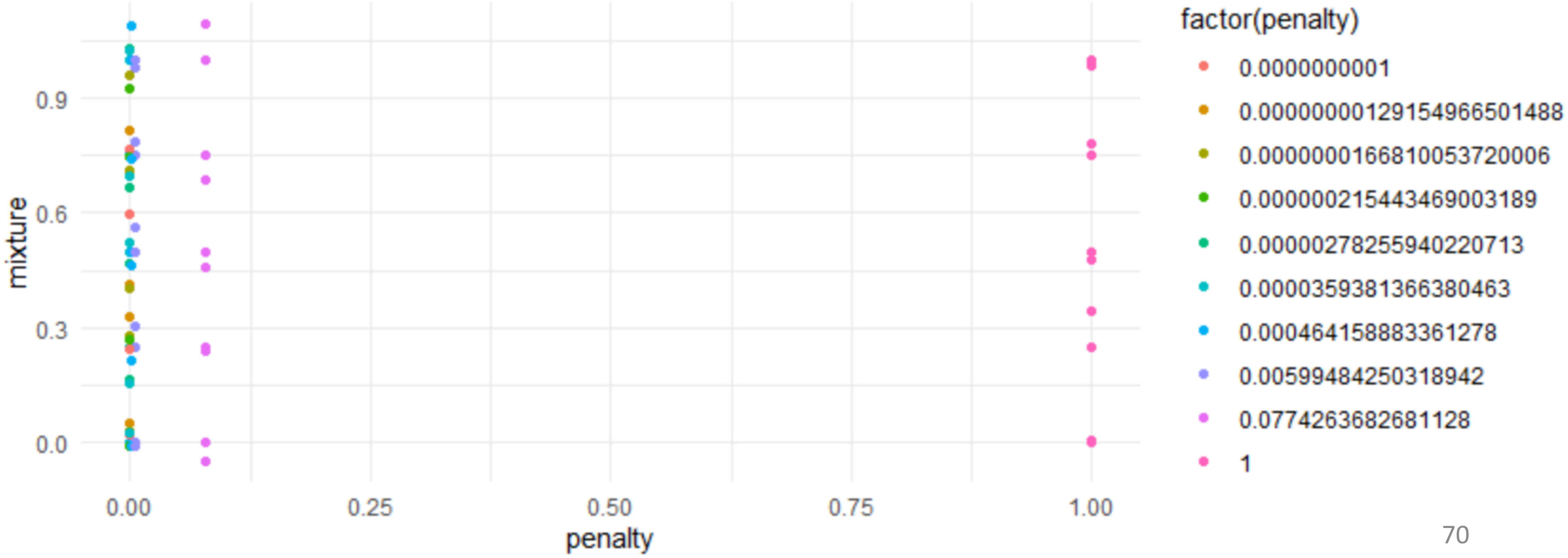
```
options(scipen = 999)
unique(enet_grid$penalty)
[1] 0.0000000010000 0.0000000129155 0.0000001668101 0.0000021544347
[5] 0.0000278255940 0.0003593813664 0.0046415888336 0.0599484250319
[9] 0.07742636826811 1.0000000000000
```

```
unique(enet_grid$mixture)
0.00 0.25 0.50 0.75 1.00
```

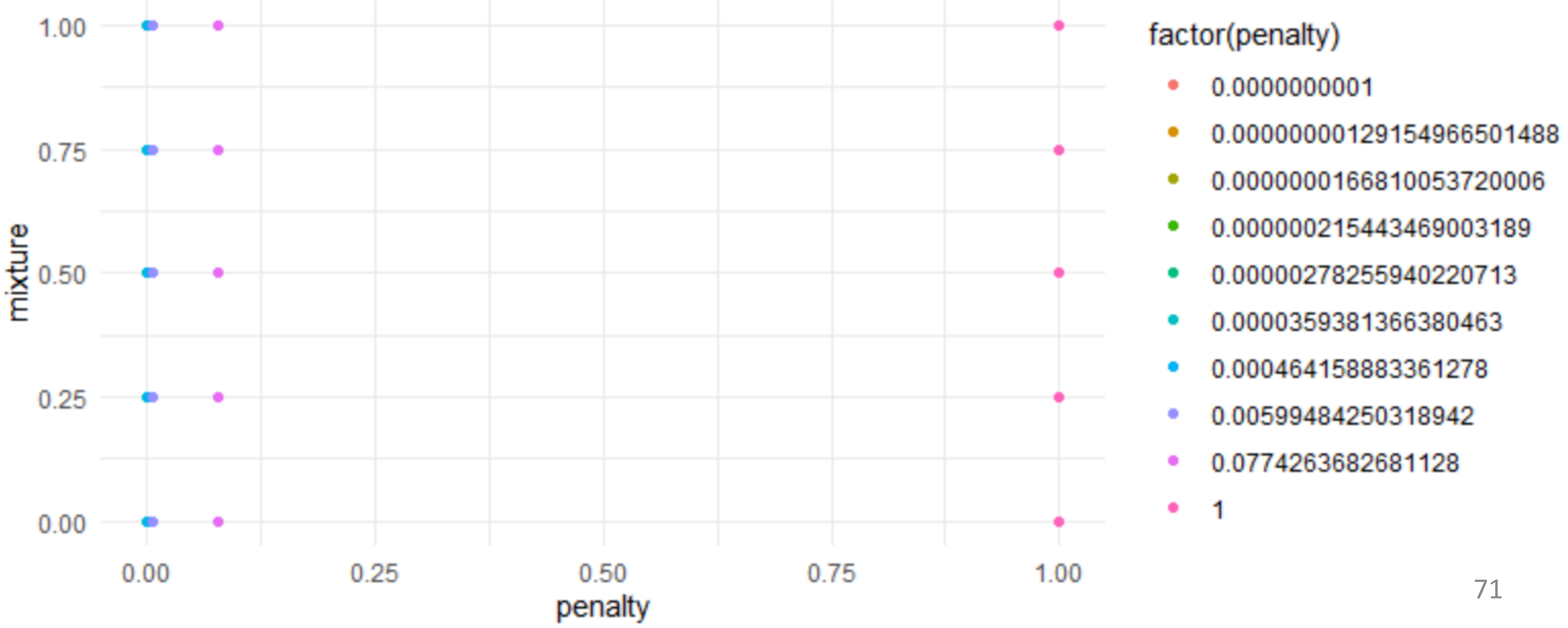
```
enet_grid %>%
  ggplot(aes(penalty, mixture, color = factor(penalty))) +
  geom_point()
```



```
options(scipen = 999)
unique(enet_grid$penalty)
[1] 0.0000000010000 0.0000000129155 0.0000001668101 0.0000021544347
[5] 0.0000278255940 0.0003593813664 0.0046415888336 0.0599484250319
[9] 0.0774263682681 1.0000000000000

unique(enet_grid$mixture)
0.00 0.25 0.50 0.75 1.00

enet_grid %>%
  ggplot(aes(penalty, mixture, color = factor(penalty))) +
  geom_point()  +
  geom_jitter()
```



```r
options(scipen = 999)
unique(enet_grid$penalty)
[1] 0.0000000010000 0.0000000129155 0.0000001668101 0.0000021544347
[5] 0.0000278255940 0.0003593813664 0.0046415888336 0.0599484250319
[9] 0.0774263682681 1.0000000000000

unique(enet_grid$mixture)
0.00 0.25 0.50 0.75 1.00

enet_grid %>%
  ggplot(aes(penalty, mixture, color = factor(penalty))) +
  geom_point()
```



Quick recap

```r
enet_params <- parameters(penalty(), mixture())
enet_grid <- grid_regular(enet_params, levels = c(10, 5))
```

Make new *tuned model*

```r
enet_tune_mod <- linear_reg() %>%
  set_engine("glmnet") %>%
  set_args(penalty = tune(),
           mixture = tune())
```

Fit tuned model with `tune_grid()`

```r
enet_tune_mod_results <- tune_grid(
  enet_tune_mod,
  preprocessor = penreg_rec,
  resamples = cv_splits,
  grid = enet_grid,
# metrics = yardstick::metric_set(rmse),
  control = tune::control_resamples(verbose = TRUE,
                                    save_pred = TRUE)
)
```

[Run the previous slide to show the verbose output]

Quick note

- It turns out that evaluating values of penalty are cheaper than values of mixture
- This is because the model simultaneously computes parameter estimates for **all possible** penalty values (for a fixed mixture)
- So we evaluate 50 models pe fold, but only fit 5 per fold
- Somehow it is able to derive all penalty values given just one (the largest). I believe it uses `predict()` somehow to do this. But I am unsure how, or why it works for some hyperparemeters and not others.
  - For example, I believe it will work with some models/packages (`C5.0, earth, enet, glmboost, glmnet, lasso, rpart`) and some parameters (e.g., `n_trees`).
  - `Tidymodels` will do this automatically (obviously I did not do this)

Results: Tuned elastic net regression

```
collect_metrics(enet_tune_mod_results)
```

50 models x 2 metrics (rmse, rsq) = 100

```
# A tibble: 100 x 7
      penalty mixture .metric .estimator    mean     n std_err
        <dbl>   <dbl> <chr>   <chr>        <dbl> <int>   <dbl>
 1 0.0000000001    0    rmse    standard   102.       10 0.351
 2 0.0000000001    0    rsq     standard     0.229    10 0.00217
 3 0.0000000001    0.25 rmse    standard   101.       10 0.357
 4 0.0000000001    0.25 rsq     standard     0.230    10 0.00220
 5 0.0000000001    0.5  rmse    standard   101.       10 0.357
 6 0.0000000001    0.5  rsq     standard     0.230    10 0.00220
 7 0.0000000001    0.75 rmse    standard   101.       10 0.357
 8 0.0000000001    0.75 rsq     standard     0.230    10 0.00220
 9 0.0000000001    1    rmse    standard   101.       10 0.357
10 0.0000000001    1    rsq     standard     0.230    10 0.00220
# ... with 90 more rows
```

show_best()

```r
enet_tune_mod_results  %>%
  show_best(metric = "rmse", n = 5)
```

```
# A tibble: 5 x 7
        penalty mixture .metric .estimator  mean     n std_err
          <dbl>   <dbl> <chr>   <chr>      <dbl> <int>   <dbl>
1 0.0000000001     0.25 rmse    standard    101.    10   0.357
2 0.00000000129    0.25 rmse    standard    101.    10   0.357
3 0.0000000167     0.25 rmse    standard    101.    10   0.357
4 0.000000215      0.25 rmse    standard    101.    10   0.357
5 0.00000278       0.25 rmse    standard    101.    10   0.357
```

select_best()

```
tnr_enet_results %>%
    select_best(metric = "rmse")
```

```
# A tibble: 1 x 2
       penalty mixture
         <dbl>   <dbl>
1 0.0000000001    0.25
```

Final fit!

```
# Select best tuning parameters
enet_best <- enet_tune_mod_results %>%
  select_best(metric = "rmse")

# Finalize your model using the best tuning parameters
enet_mod_final <- enet_tune_mod %>%
  finalize_model(enet_best)

# Finalize your recipe using the best turning parameters
enet_rec_final <- penreg_rec %>%
  finalize_recipe(enet_best)

# Run your last fit on your initial data split
enet_test_results <- last_fit(
  enet_mod_final,
  enet_rec_final,
  split = math_split)

#Collect metrics
enet_test_results %>%
  collect_metrics()
# A tibble: 2 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard     101.
2 rsq     standard       0.235
```

This will spend your test set…
SO DON'T DO THIS UNLESS YOU ARE CERTAIN OF YOUR MODELLING PROCESS

ABSOLUTELY CERTAIN!

These are the prediction measures you can reasonably expect

Quick comparison

- Resampled fit

```
show_best(enet_tune_mod_results, metric = "rmse", n = 1) %>%
  bind_rows(show_best(enet_tune_mod_results, metric = "rsq", n = 1)) %>%
  select(`.metric`, `.estimator`, mean)
```

```
# A tibble: 2 x 3
  .metric .estimator    mean
  <chr>   <chr>        <dbl>
1 rmse    standard   101.
2 rsq     standard     0.230
```

- Final fit

```
enet_test_results %>%
   collect_metrics()
```

```
# A tibble: 2 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard     101.
2 rsq     standard       0.235
```

Name the package

- `initial_split()`

```
set.seed(210)
math_split <- initial_split(math)
```

Name the package

- `training()`
- `testing()`

```
math_train <- training(math_split)
math_test  <- testing(math_split)
```

Name the package

- `vfold_cv()`

```
set.seed(210)
cv_splits <- vfold_cv(math_train)
```

Name the package

- `recipe()`
- `step_*()`

```
penreg_rec <-
  recipe(
    score ~ enrl_grd + econ_dsvntg + lat + lon,
    data = math_train
  ) %>%
  step_dummy(all_nominal()) %>%
  step_normalize(lat, lon)
```

Name the package

- linear_reg()
- set_engine()
- set_mode()
- set_args()

```
mod_ridge <- linear_reg() %>%
  set_engine("glmnet") %>%
  set_mode("regression") %>%
  set_args(penalty = .1,
           mixture = 0)
```

Name the package

- `fit_resamples()`

```
fit_resamples(
  penreg_rec,
  model = mod_ridge,
  resamples = cv_splits,
  metrics = yardstick::metric_set(rmse),
  control = tune::control_resamples(verbose = TRUE,
                                    save_pred = TRUE)
)
```

Name the package

- `tune_grid()`

Lab 2