Regularization Lab

Last modified: 2026-11-19

```
# The Python chunks on this page import numpy, pandas, scipy,
# matplotlib and plotly;
# reticulate provisions them.
library(reticulate) # nolint: undesirable_function_linter
py_packages <- c("numpy", "pandas", "scipy", "matplotlib", "plotly")
py_require(py_packages)
```

1 The book's Chapter 6 lab

This page holds the code lab for the Regularization chapter.

To work the lab without the solutions, download a stub with only the exercises:

- Quarto
- Jupyter notebook[1]
- Python[2]
- R[3]

Each solution shows the code in Python and in R. The exercises build on one another, so work them in order.

Section 6.5 of the book [1] is the source of these exercises. The Python code follows the book's Chapter 6 lab notebook[4] [2]. The book's lab fits ridge regression and the lasso with `glmnet` and fits principal components regression and partial least squares with the `pls` package [3, Sec. 6.5]. Here the code solves the penalized problems directly, with the closed form for ridge regression and the coordinate descent of the Regularization chapter for the lasso. The book's first two parts, best subset and forward selection, are the `Hitters` exercises in the Subset Selection chapter.

2 Ridge regression

Exercise 2.1 (Predicting baseball salaries with ridge regression)**.** The `Hitters` data give the 1987 opening-day salary, in thousands of dollars, of 263 Major League Baseball players, with 19 other variables: their performance statistics in 1986 and over their careers, and their league and division [1, Sec. 6.5]. We drop players whose salary is missing.
Split the players at random into a training half and a test half. Standardize each predictor using the training half. For a range of λ, fit ridge regression on the training half with the closed-form solution[a] of this chapter, and compute the sample mean squared error (sMSE) on the test half.

[a] https://morrison-lab.github.io/mcs/expressions-algorithms.html#def-closed-form-solution

[1] https://morrison-lab.github.io/lds/chapters/lab-stubs/regularization-lab-stub.ipynb
[2] https://morrison-lab.github.io/lds/chapters/lab-stubs/regularization-lab-stub.py
[3] https://morrison-lab.github.io/lds/chapters/lab-stubs/regularization-lab-stub.R
[4] https://github.com/intro-stat-learning/ISLP_labs/blob/main/Ch06-varselect-lab.ipynb

Solution

Solution 2.1.

Python

```
import numpy as np
import pandas as pd
from scipy import stats

hit_py = pd.read_csv("data/hitters.csv").dropna()
print(hit_py.head(3))
```

```
   AtBat  Hits  HmRun  Runs  ...  Assists  Errors  Salary NewLeague
1    315    81      7    24  ...       43      10   475.0         N
2    479   130     18    66  ...       82      14   480.0         A
3    496   141     20    65  ...       11       3   500.0         N

[3 rows x 20 columns]
```

```
y_py = hit_py["Salary"].to_numpy()
x_py = pd.get_dummies(hit_py.drop(columns="Salary"), drop_first=True).to_numpy(dtype=float)

rng = np.random.default_rng(1)
perm = rng.permutation(len(y_py))
train_py, test_py = perm[: len(y_py) // 2], perm[len(y_py) // 2 :]

mu, sd = x_py[train_py].mean(axis=0), x_py[train_py].std(axis=0, ddof=1)
X_py = np.column_stack([np.ones(len(y_py)), (x_py - mu) / sd])
I0_py = np.eye(X_py.shape[1])
I0_py[0, 0] = 0.0

def ridge_py(lam):
    Xt = X_py[train_py]
    return np.linalg.solve(Xt.T @ Xt + lam * I0_py, Xt.T @ y_py[train_py])

lams_py = 10.0 ** np.linspace(-2, 5, 50)
# The squared error of each test player at each penalty, one column per penalty.
sq_py = np.column_stack([(y_py[test_py] - X_py[test_py] @ ridge_py(l)) ** 2 for l in lams_py])
mse_py = sq_py.mean(axis=0)
```

R

```
hit <- na.omit(read.csv("data/hitters.csv"))
str(hit)
```

```
'data.frame':   263 obs. of  20 variables:
 $ AtBat    : int  315 479 496 321 594 185 298 323 401 574 ...
 $ Hits     : int  81 130 141 87 169 37 73 81 92 159 ...
 $ HmRun    : int  7 18 20 10 4 1 0 6 17 21 ...
 $ Runs     : int  24 66 65 39 74 23 24 26 49 107 ...
 $ RBI      : int  38 72 78 42 51 8 24 32 66 75 ...
 $ Walks    : int  39 76 37 30 35 21 7 8 65 59 ...
 $ Years    : int  14 3 11 2 11 2 3 2 13 10 ...
 $ CAtBat   : int  3449 1624 5628 396 4408 214 509 341 5206 4631 ...
 $ CHits    : int  835 457 1575 101 1133 42 108 86 1332 1300 ...
 $ CHmRun   : int  69 63 225 12 19 1 0 6 253 90 ...
 $ CRuns    : int  321 224 828 48 501 30 41 32 784 702 ...
```

```
 $ CRBI     : int  414 266 838 46 336 9 37 34 890 504 ...
 $ CWalks   : int  375 263 354 33 194 24 12 8 866 488 ...
 $ League   : chr  "N" "A" "N" "N" ...
 $ Division : chr  "W" "W" "E" "E" ...
 $ PutOuts  : int  632 880 200 805 282 76 121 143 0 238 ...
 $ Assists  : int  43 82 11 40 421 127 283 290 0 445 ...
 $ Errors   : int  10 14 3 4 25 7 9 19 0 22 ...
 $ Salary   : num  475 480 500 91.5 750 ...
 $ NewLeague: chr  "N" "A" "N" "N" ...
 - attr(*, "na.action")= 'omit' Named int [1:59] 1 16 19 23 31 33 37 39 40 42 ...
  ..- attr(*, "names")= chr [1:59] "1" "16" "19" "23" ...
```

```r
x_raw <- model.matrix(Salary ~ ., data = hit)[, -1]
y <- hit$Salary

set.seed(1)
train <- sample(nrow(hit), nrow(hit) / 2)
Xs <- scale(
  x_raw,
  center = colMeans(x_raw[train, ]),
  scale = apply(x_raw[train, ], 2, sd)
)
X <- cbind(1, Xs)
I0 <- diag(ncol(X))
I0[1, 1] <- 0

ridge <- function(lambda) {
  Xt <- X[train, ]
  drop(solve(t(Xt) %*% Xt + lambda * I0, t(Xt) %*% y[train]))
}

lambdas <- 10^seq(-2, 5, length.out = 50)
# The squared error of each test player at each penalty, one column per penalty.
sq_err <- sapply(
  lambdas,
  function(l) drop((y[-train] - X[-train, ] %*% ridge(l))^2)
)
test_mse <- colMeans(sq_err)
paths <- sapply(lambdas, ridge)
```

The two languages split the players at random in different ways, so their numbers differ. The numbers in the later parts of this solution are from the R split.

Exercise 2.2 (Predicting baseball salaries with ridge regression (part 2))**.** Use the setup of Exercise 2.1. Which λ has the lowest test error?

Solution

Solution 2.2.

Python

```python
def mean_ci_py(sq):
    # A 95% confidence interval for the sample mean squared error (sMSE) over test players:
    # the mean plus or minus t_crit standard errors.
    n = len(sq)
    half = stats.t.ppf(0.975, df=n - 1) * sq.std(ddof=1) / np.sqrt(n)
    return sq.mean() - half, sq.mean() + half

best_py = int(np.argmin(mse_py))
# The sRMSE is the square root of the sMSE, and so are the endpoints of its interval.
print("lowest test sRMSE:", round(float(np.sqrt(mse_py[best_py]))), "at lambda", round(lams_py[best_py
      np.round(np.sqrt(mean_ci_py(sq_py[:, best_py]))))
```

```
lowest test sRMSE: 322 at lambda 27 [252. 379.]
```

R

```r
# A 95% confidence interval for the sample mean squared error (sMSE) over test players:
# the mean plus or minus t_crit standard errors.
mean_ci <- function(sq) {
  half <- qt(0.975, df = length(sq) - 1) * sd(sq) / sqrt(length(sq))
  c(lower = mean(sq) - half, upper = mean(sq) + half)
}
ci_mse <- t(apply(sq_err, 2, mean_ci))
c(lowest = sqrt(min(test_mse)), at_lambda = lambdas[which.min(test_mse)])
```

```
   lowest at_lambda
 369.9421  268.2696
```

```r
best_lambda <- lambdas[which.min(test_mse)]
fmt <- function(x) format(round(x), big.mark = ",", scientific = FALSE)
rmse <- sqrt(test_mse)
ci_rmse <- sqrt(ci_mse)
```

The lowest test sample root mean squared error (sRMSE), 370 thousand dollars (95% confidence interval[a] [296, 431]), is at $\lambda \approx 268$. The sRMSE is the square root of the sample mean squared error, so it is in the units of `Salary` (why we report sRMSE). Each interval is the mean[b] of the test players' squared errors plus or minus 1.98 standard errors[c], the t interval for a mean, with its endpoints square-rooted. The lowest test error was also picked from 50 candidates by looking at the test half, so it is optimistic about the error of the chosen λ. An honest error for the chosen λ needs players that took no part in choosing it. The penalty trades a little bias[d] for a lower variance[e], and a model with 19 correlated predictors and 131 training players has variance to give up. Figure 1 shows the test error and the coefficients for every λ.

[a] https://morrison-lab.github.io/sds/inference.html#def-confidence-interval
[b] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean
[c] https://morrison-lab.github.io/sds/estimation.html#def-SE
[d] https://morrison-lab.github.io/sds/estimation.html#def-bias
[e] https://morrison-lab.github.io/pds/variance-covariance.html#def-variance

```r
library(ggplot2)
library(patchwork)
mse_df <- tibble::tibble(
  lambda = lambdas, test_rmse = sqrt(test_mse),
  lower = sqrt(ci_mse[, "lower"]), upper = sqrt(ci_mse[, "upper"])
)
p_mse <- ggplot(mse_df, aes(lambda, test_rmse)) +
  geom_ribbon(aes(ymin = lower, ymax = upper), alpha = 0.2) +
  geom_line() +
  geom_point(data = tibble::tibble(lambda = best_lambda, test_rmse = sqrt(min(test_mse))),
             shape = 1, size = 5) +
```

```
  scale_x_log10(
    labels = scales::label_number(big.mark = ",", drop0trailing = TRUE)
  ) +
  labs(x = "penalty (lambda)", y = "test sRMSE (thousand dollars)")

path_df <- tibble::tibble(
  lambda = rep(lambdas, each = nrow(paths) - 1),
  predictor = rep(seq_len(nrow(paths) - 1), times = length(lambdas)),
  coefficient = as.vector(paths[-1, ])
)
p_paths <- ggplot(path_df, aes(lambda, coefficient, group = predictor)) +
  geom_hline(yintercept = 0, linetype = "dashed") +
  geom_line(colour = "grey40") +
  scale_x_log10(
    labels = scales::label_number(big.mark = ",", drop0trailing = TRUE)
  ) +
  labs(x = "penalty (lambda)", y = "standardized coefficient")

p_mse + p_paths
```

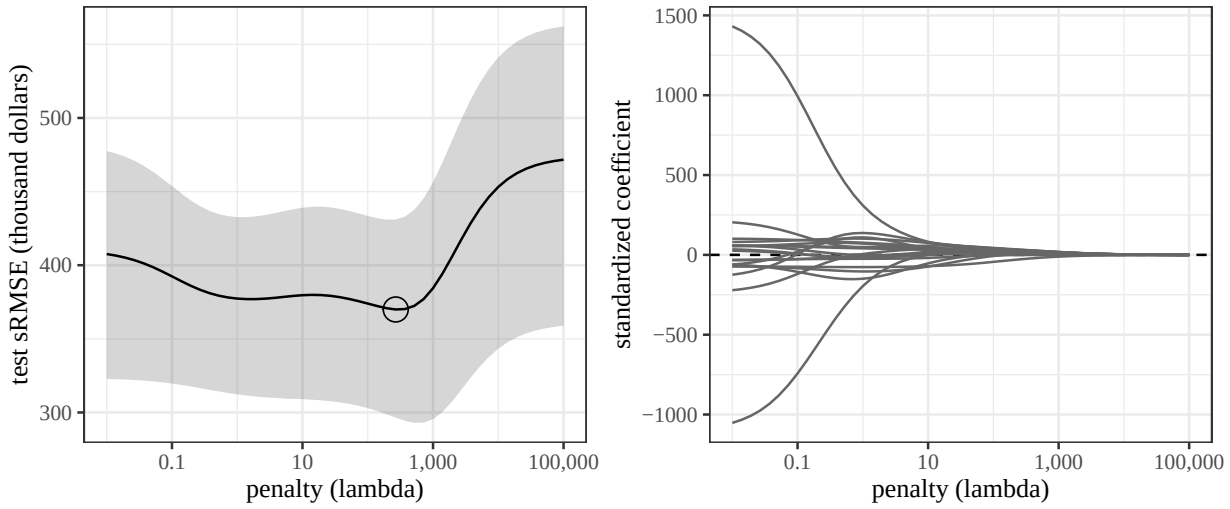


Figure 1: Ridge regression on the Hitters data. Left: test sample root mean squared error (sRMSE) against the penalty, with a shaded 95% confidence interval and the lowest value marked. Right: each standardized coefficient against the penalty, one line per predictor.

Exercise 2.3 (Predicting baseball salaries with ridge regression (part 3))**.** Use the setup of Exercise 2.1. How does the test error of that λ compare with that of $\lambda \approx 0$?

Solution

Solution 2.3.

```
# Every model predicts the same test players, so differences are taken player by player.
diff_small <- mean_ci(sq_err[, 1] - sq_err[, which.min(test_mse)])
small_covers_0 <- diff_small[["lower"]] <= 0 && diff_small[["upper"]] >= 0
```

Almost no penalty, $\lambda = 10^{-2}$, gives a test sRMSE of 408 ([323, 478]), which is 10% higher than the lowest. The intervals for different λ overlap because they use the same players, so compare errors

player by player, in squared units: almost no penalty has an sMSE larger by 29,259 on average than the lowest-error penalty, with a 95% confidence interval[a] of [-24,602, 83,119]. This interval includes 0, so on this split the improvement over almost no penalty is within what the choice of test players could produce, even though the penalty was picked because its test error was lowest.

[a] https://morrison-lab.github.io/sds/inference.html#def-confidence-interval

Exercise 2.4 (Predicting baseball salaries with ridge regression (part 4))**.** Use the setup of Exercise 2.1. How does the test error of that λ compare with that of predicting the training mean[a] salary for every player?

[a] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean

Solution

Solution 2.4.

```
sq_null <- (y[-train] - mean(y[train]))^2
null_mse <- mean(sq_null)
null_rmse <- sqrt(null_mse)
diff_null <- mean_ci(sq_null - sq_err[, which.min(test_mse)])
null_covers_0 <- diff_null[["lower"]] <= 0 && diff_null[["upper"]] >= 0
```

Predicting the training mean[a] salary for every player gives a test sRMSE of 474. Its sMSE is larger than that of the lowest-error penalty by 87,813 on average, with a 95% confidence interval[b] of [24,199, 151,427]. This interval excludes 0. Both intervals reflect which players were in the test half, not which were in the training half, and both differences are optimistic for the same reason as the lowest error: the penalty was picked by looking at the test half.

[a] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean
[b] https://morrison-lab.github.io/sds/inference.html#def-confidence-interval

Exercise 2.5 (Choose the ridge penalty by cross-validation on baseball salaries)**.** In Exercise 2.1 we chose λ by its error on the test half, so no player was left to estimate the error of the choice. Choose it using the training half alone instead.
Use the training half and the standardized predictors of Exercise 2.1. Randomly assign the training players to ten folds, and for each λ of that exercise's grid compute the 10-fold cross-validation estimate[a] and its fold-based standard error[b].

[a] https://morrison-lab.github.io/sds/model-validation.html#def-cv-estimate
[b] https://morrison-lab.github.io/sds/model-validation.html#def-cv-se

Solution

Solution 2.5. The code standardizes the predictors once on the whole training half, as in Solution 2.1, so the held-out fold's means[a] and standard deviations[b] take part in its scaling. That leaks a little information from each held-out fold into its own fit.

R

```r
x_tr <- X[train, ]
ytr <- y[train]
k_cv <- 10
set.seed(2)
fold <- sample(rep(seq_len(k_cv), length.out = length(train)))

# The squared error of each training player when its fold is held out,
# one column per penalty.
cv_sq <- sapply(lambdas, function(l) {
  sq <- numeric(length(ytr))
  for (f in seq_len(k_cv)) {
    held <- fold == f
    x_in <- x_tr[!held, ]
    b <- solve(t(x_in) %*% x_in + l * I0, t(x_in) %*% ytr[!held])
    sq[held] <- (ytr[held] - x_tr[held, ] %*% b)^2
  }
  sq
})

cv_est <- colMeans(cv_sq)
# The fold-based standard error: the standard deviation of the ten fold sMSEs over sqrt(10).
fold_mse <- apply(cv_sq, 2, function(sq) tapply(sq, fold, mean))
cv_se <- apply(fold_mse, 2, sd) / sqrt(k_cv)
```

Python

```python
import matplotlib.pyplot as plt

Xtr_py, ytr_py = X_py[train_py], y_py[train_py]
k_cv = 10
rng_cv = np.random.default_rng(2)
fold_py = rng_cv.permutation(np.arange(len(ytr_py)) % k_cv)

# The squared error of each training player when its fold is held out,
# one column per penalty.
cv_sq_py = np.empty((len(ytr_py), len(lams_py)))
for f in range(k_cv):
    held = fold_py == f
    x_in, ya = Xtr_py[~held], ytr_py[~held]
    for j, lam in enumerate(lams_py):
        b = np.linalg.solve(x_in.T @ x_in + lam * I0_py, x_in.T @ ya)
        cv_sq_py[held, j] = (ytr_py[held] - Xtr_py[held] @ b) ** 2

cv_py = cv_sq_py.mean(axis=0)
# The fold-based standard error: the standard deviation of the ten fold sMSEs over sqrt(10).
fold_mse_py = np.array([cv_sq_py[fold_py == f].mean(axis=0) for f in range(k_cv)])
se_py = fold_mse_py.std(axis=0, ddof=1) / np.sqrt(k_cv)
```

The numbers in the later parts of this solution are from the R split and its folds.

[a] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean
[b] https://morrison-lab.github.io/pds/variance-covariance.html#def-sd

Exercise 2.6 (Choose the ridge penalty by cross-validation on baseball salaries (part 2))**.** Use the setup of Exercise 2.5.

Find the λ with the smallest estimate[a], and the largest λ whose estimate is within one fold-based standard error[b] of the smallest (the one-standard-error rule[c]).

[a] https://morrison-lab.github.io/sds/estimation.html#def-estimate
[b] https://morrison-lab.github.io/sds/model-validation.html#def-cv-se
[c] https://morrison-lab.github.io/sds/model-validation.html#def-one-se-rule

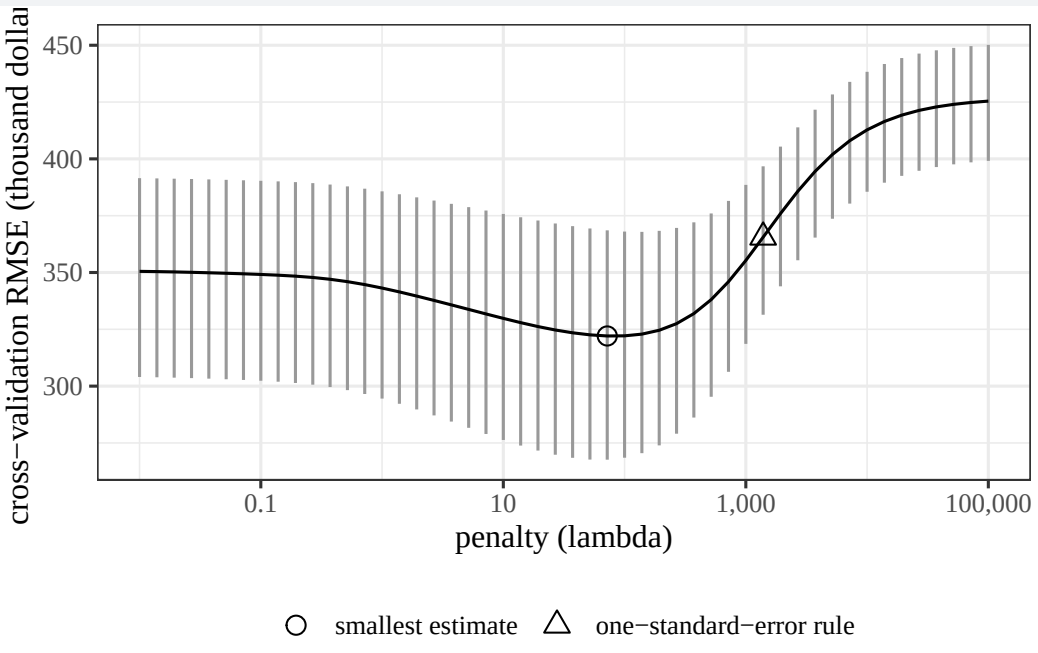
Solution

Solution 2.6. A ridge fit with a larger λ is the less flexible fit, so the one-standard-error rule[a] picks the largest λ in range.

R

```
i_min <- which.min(cv_est)
i_se <- max(which(cv_est <= cv_est[i_min] + cv_se[i_min]))
picks <- c(min = i_min, one_se = i_se)

library(ggplot2)
cv_df <- data.frame(lambda = lambdas, cv = cv_est, se = cv_se)
ggplot(cv_df, aes(lambda, sqrt(cv))) +
  geom_linerange(aes(ymin = sqrt(pmax(cv - se, 0)), ymax = sqrt(cv + se)), colour = "grey60") +
  geom_line() +
  geom_point(data = cv_df[picks, ], aes(shape = names(picks)), size = 3) +
  scale_x_log10(labels = scales::label_number(big.mark = ",", drop0trailing = TRUE)) +
  scale_shape_manual(
    values = c(min = 1, one_se = 2),
    labels = c(min = "smallest estimate", one_se = "one-standard-error rule")
  ) +
  labs(x = "penalty (lambda)", y = "cross-validation RMSE (thousand dollars)", shape = NULL) +
  theme(legend.position = "bottom")
```



Python

```
i_min_py = int(np.argmin(cv_py))
i_se_py = int(np.flatnonzero(cv_py <= cv_py[i_min_py] + se_py[i_min_py]).max())

fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
ax.errorbar(lams_py, np.sqrt(cv_py), color="black",
            yerr=[np.sqrt(cv_py) - np.sqrt(np.maximum(cv_py - se_py, 0)),
                  np.sqrt(cv_py + se_py) - np.sqrt(cv_py)],
            ecolor="grey");
ax.scatter(lams_py[i_min_py], np.sqrt(cv_py[i_min_py]), marker="o", s=80,
           facecolors="none", edgecolors="black", label="smallest estimate", zorder=3);
ax.scatter(lams_py[i_se_py], np.sqrt(cv_py[i_se_py]), marker="^", s=80,
           facecolors="none", edgecolors="black", label="one-standard-error rule", zorder=3);
ax.set_xscale("log");
ax.set_xlabel("penalty (lambda)");
ax.set_ylabel("cross-validation RMSE (thousand dollars)");
ax.legend();
plt.show()
```

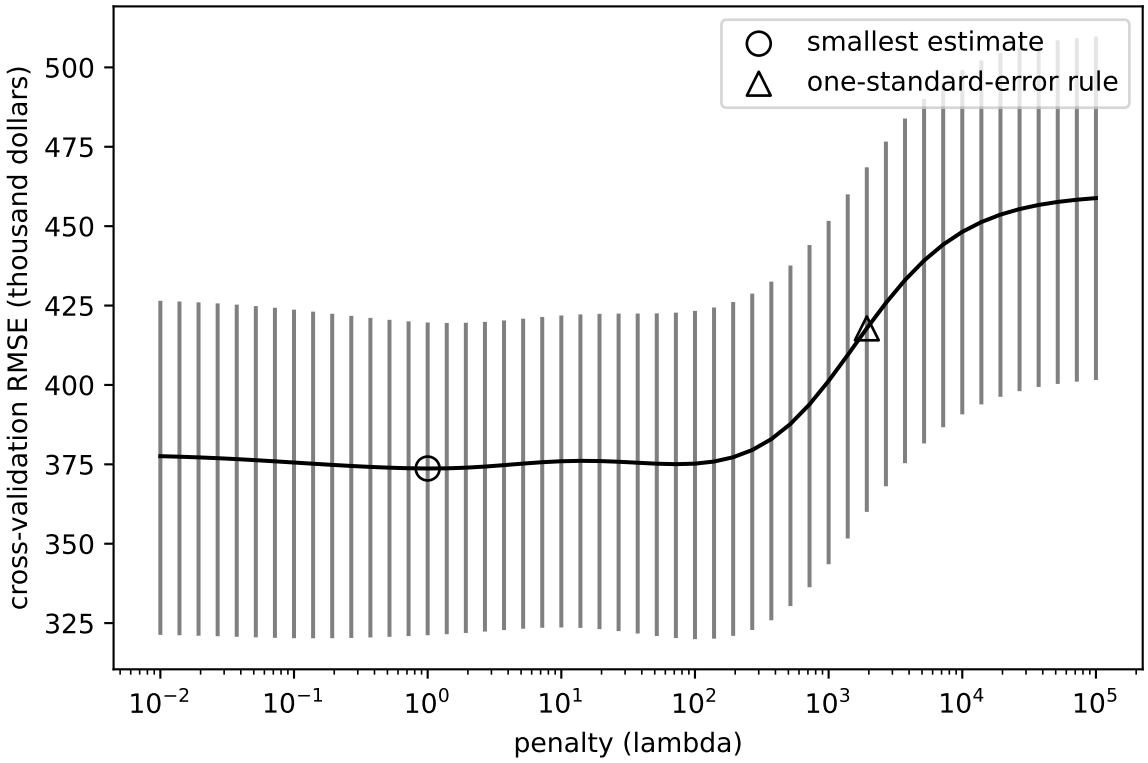


Figure 2: How the ridge fit scores against the penalty, with the best score and the one-standard-error choice marked.

Figure 2 plots the estimates[b] against the penalty. The smallest cross-validation estimate is at $\lambda \approx 72$, with a cross-validation root mean squared error (RMSE) of 322 thousand dollars. The one-standard-error rule[c] picks the larger $\lambda \approx 1389$, whose estimate[d] is 366.

[a] https://morrison-lab.github.io/sds/model-validation.html#def-one-se-rule
[b] https://morrison-lab.github.io/sds/estimation.html#def-estimate
[c] https://morrison-lab.github.io/sds/model-validation.html#def-one-se-rule
[d] https://morrison-lab.github.io/sds/estimation.html#def-estimate

Exercise 2.7 (Choose the ridge penalty by cross-validation on baseball salaries (part 3))**.** Use the setup of Exercise 2.5.
Refit on the whole training half at each of the two penalties, and compute their test errors.

Solution

Solution 2.7.

R

```
test_rmse <- sapply(picks, function(i) {
  sqrt(mean((y[-train] - X[-train, ] %*% ridge(lambdas[i]))^2))
})
data.frame(
  rule = names(picks), lambda = lambdas[picks],
  cv_rmse = sqrt(cv_est[picks]), test_rmse = test_rmse
)
```

```
# A tibble: 2 x 4
  rule   lambda cv_rmse test_rmse
  <chr>   <dbl>   <dbl>     <dbl>
1 min      72.0    322.      376.
2 one_se 1389.     366.      394.
```

Python

```
for name, i in [("smallest estimate", i_min_py), ("one-standard-error rule", i_se_py)]:
    b = ridge_py(lams_py[i])
    test_sq = (y_py[test_py] - X_py[test_py] @ b) ** 2
    print(name, "- lambda", round(lams_py[i]), "- cv sRMSE", round(float(np.sqrt(cv_py[i]))),
          "- test sRMSE", round(float(np.sqrt(test_sq.mean()))))
```

```
smallest estimate - lambda 1 - cv sRMSE 374 - test sRMSE 327
one-standard-error rule - lambda 1931 - cv sRMSE 418 - test sRMSE 395
```

The training half alone chose both penalties, so the test half gives an error estimate[a] for each that the choice did not use. The test RMSEs are 376 at the smallest estimate and 394 at the larger penalty.

[a] https://morrison-lab.github.io/sds/estimation.html#def-estimate

Exercise 2.8 (Choose the ridge penalty by cross-validation on baseball salaries (part 4))**.** Use the setup of Exercise 2.5.
Do the two penalties differ much in cross-validation estimate and in test error?

Solution

Solution 2.8. The two penalties of Solution 2.6 differ by 43 in cross-validation RMSE and by 19 in test RMSE, so the larger penalty has the higher test error on this split. The rule trades a small increase in cross-validation error for a simpler, more heavily shrunken fit, and the cross-validation error alone cannot say which is better.

3 The lasso

Exercise 3.1 (The coordinate update for the lasso (part 5))**.** Use the same notation as the coordinate update exercise.
Set $\lambda = 0$. What does the update reduce to?

Solution

Solution 3.1. With $\lambda = 0$, $S_0(c) = c$, and the update is $\beta_j = \mathbf{D}_{\cdot j} \cdot r_j / \|\mathbf{D}_{\cdot j}\|^2$. That update is the least squares coefficient of the residual r_j on the single feature $\mathbf{D}_{\cdot j}$. Coordinate descent then amounts to fitting each feature to what the others leave unexplained, in turn.
Now apply it. We use the same training half and standardized predictors as Exercise 2.1. We center the salaries on the training mean[a], which takes care of the intercept[b].

Python

```
# Soft-thresholding shrinks z toward zero by tau,
# and returns exactly zero when |z| <= tau, which makes the lasso sparse.
def soft_py(z, tau):
  return np.sign(z) * max(abs(z) - tau, 0.0)

def lasso_cd_py(X, y, lam, tol=1e-8, max_sweeps=5000):
  beta = np.zeros(X.shape[1])
  col_ss = (X**2).sum(axis=0)  # ||X_j||^2 for each column, which does not change
  for _ in range(max_sweeps):
      biggest_change = 0.0
      for j in range(X.shape[1]):
          # Partial residual: y minus every feature's contribution except feature j's.
          r_j = y - X @ beta + X[:, j] * beta[j]
          new = soft_py(X[:, j] @ r_j, lam / 2) / col_ss[j]
          biggest_change = max(biggest_change, abs(new - beta[j]))
          beta[j] = new
      # Stop once a full sweep barely moves any coefficient.
      if biggest_change < tol:
          break
  return beta

# Drop the intercept column of X_py; centering y takes care of the intercept.
Xtr_py = X_py[train_py, 1:]
ytr_py = y_py[train_py] - y_py[train_py].mean()
lam_max_py = 2 * np.abs(Xtr_py.T @ ytr_py).max()  # the smallest lambda that zeroes every coefficient
lams_lasso_py = lam_max_py * 10.0 ** np.linspace(-4, 0, 30)

nonzero_py, mse_lasso_py = [], []
for lam in lams_lasso_py:
  b = lasso_cd_py(Xtr_py, ytr_py, lam)
  nonzero_py.append(int((b != 0).sum()))
  pred = y_py[train_py].mean() + X_py[test_py, 1:] @ b
  mse_lasso_py.append(np.mean((y_py[test_py] - pred) ** 2))
print("nonzero coefficients at the smallest and largest lambda:", nonzero_py[0], nonzero_py[-1])
```

```
nonzero coefficients at the smallest and largest lambda: 19 0
```

```
print("lowest test MSE:", round(min(mse_lasso_py)), "with", nonzero_py[int(np.argmin(mse_lasso_py))],
```

```
lowest test MSE: 106793 with 14 nonzero coefficients
```

R

```
# Soft-thresholding shrinks z toward zero by tau,
# and returns exactly zero when |z| <= tau, which makes the lasso sparse.
soft <- function(z, tau) sign(z) * pmax(abs(z) - tau, 0)

lasso_cd <- function(x_mat, y, lambda, tol = 1e-8, max_sweeps = 5000) {
beta <- rep(0, ncol(x_mat))
col_ss <- colSums(x_mat^2) # ||X_j||^2 for each column, which does not change
for (s in seq_len(max_sweeps)) {
  biggest_change <- 0
  for (j in seq_len(ncol(x_mat))) {
    # Partial residual: y minus every feature's contribution except j's.
    r_j <- y - x_mat %*% beta + x_mat[, j] * beta[j]
    new <- soft(sum(x_mat[, j] * r_j), lambda / 2) / col_ss[j]
    biggest_change <- max(biggest_change, abs(new - beta[j]))
    beta[j] <- new
  }
  # Stop once a full sweep barely moves any coefficient.
  if (biggest_change < tol) break
}
beta
}

# Drop the intercept column of x_mat; centering y takes care of the intercept.
x_train <- Xs[train, ]
ytr <- y[train] - mean(y[train])
# The smallest lambda that zeroes every coefficient.
lambda_max <- 2 * max(abs(crossprod(x_train, ytr)))
lambdas_lasso <- lambda_max * 10^seq(-4, 0, length.out = 30)

fits <- sapply(lambdas_lasso, function(l) lasso_cd(x_train, ytr, l))
nonzero <- colSums(fits != 0)
mse_lasso <- apply(
fits, 2,
function(b) mean((y[-train] - mean(y[train]) - Xs[-train, ] %*% b)^2)
)
print(nonzero[c(1, 30)])
```

```
[1] 19  0
```

```
best <- which.min(mse_lasso)
c(lowest_mse = min(mse_lasso), nonzero_at_lowest = unname(nonzero[best]))
```

```
       lowest_mse nonzero_at_lowest
         141389.5              9.0
```

```r
# At a solution, the gradient of the squared error balances the penalty's slope:
# 2 X_j'(y - Xb) = lambda * sign(b_j) when b_j is not zero,
# and |2 X_j'(y - Xb)| <= lambda when b_j is zero.
i_mid <- 15
b_mid <- fits[, i_mid]
score <- 2 * crossprod(x_train, ytr - x_train %*% b_mid)
kkt_gap <- max(
abs(score[b_mid != 0] - lambdas_lasso[i_mid] * sign(b_mid[b_mid != 0])),
pmax(abs(score[b_mid == 0]) - lambdas_lasso[i_mid], 0)
)
stopifnot(kkt_gap < 1e-5)
fmt <- function(x) format(round(x), big.mark = ",", scientific = FALSE)
```

The two languages split the players at random in different ways, so their numbers differ. The numbers below are from the R split.

With the smallest λ in the grid, all 19 coefficients are non-zero. With the largest, 0 are. The lowest test error, 141,389, comes with 9 non-zero coefficients. For comparison, ridge regression reached 136,857 in Solution 2.2, but it keeps every predictor.

As a check that coordinate descent has found the minimizer, the lasso's optimality conditions hold at the middle λ in the grid to within 10^{-5}. The three parts of the coordinate update solution encode these conditions: the penalty's slope[c] $\lambda \operatorname{sign}(\beta_j)$ balances the gradient of the squared error for each non-zero coefficient, and the gradient is no larger than λ in absolute value for each zero one.

[a] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean
[b] https://morrison-lab.github.io/mds/algebra-exponentials.html#def-intercept
[c] https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope

Exercise 3.2 (Count the zero coefficients of ridge and the lasso on baseball salaries)**.** Use the `Hitters` data and the training half of Exercise 2.1, with the predictors standardized and the salary centered by the training mean. For $\lambda = 10^2, 10^{2.5}, \dots, 10^5$, fit ridge regression with the closed-form solution[a] of this chapter, and fit the lasso by the coordinate descent of the lasso coordinate update.

[a] https://morrison-lab.github.io/mcs/expressions-algorithms.html#def-closed-form-solution

Solution

Solution 3.2.

Python

```python
import numpy as np
import pandas as pd

hz_py = pd.read_csv("data/hitters.csv").dropna()
hz_x = pd.get_dummies(hz_py.drop(columns="Salary"), drop_first=True)
hz_names = list(hz_x.columns)
hz_x = hz_x.to_numpy(dtype=float)
hz_y = hz_py["Salary"].to_numpy()

rng = np.random.default_rng(1)
hz_train = rng.permutation(len(hz_y))[: len(hz_y) // 2]

# Standardize with the training half's mean and standard deviation, and center
# the salary, so that no intercept needs to be fit or penalized.
xt = hz_x[hz_train]
xt = (xt - xt.mean(axis=0)) / xt.std(axis=0, ddof=1)
yt = hz_y[hz_train] - hz_y[hz_train].mean()

def ridge_zero_py(lam):
    return np.linalg.solve(xt.T @ xt + lam * np.eye(xt.shape[1]), xt.T @ yt)

def lasso_zero_py(lam, sweeps=200):
    # Coordinate descent: each coefficient in turn is set to the soft-thresholded
    # correlation of its feature with the partial residual, over the feature's
    # sum of squares.
    b = np.zeros(xt.shape[1])
    for _ in range(sweeps):
        for j in range(xt.shape[1]):
            r = yt - xt @ b + xt[:, j] * b[j]
            z = xt[:, j] @ r
            b[j] = np.sign(z) * max(abs(z) - lam / 2, 0.0) / (xt[:, j] @ xt[:, j])
    return b

lams_py = 10.0 ** np.arange(2, 5.5, 0.5)
```

R

```r
hz <- na.omit(read.csv("data/hitters.csv"))
hz_x <- model.matrix(Salary ~ ., data = hz)[, -1]

set.seed(1)
hz_train <- sample(nrow(hz), nrow(hz) / 2)

# Standardize with the training half's mean and standard deviation, and center
# the salary, so that no intercept needs to be fit or penalized.
xt <- scale(
  hz_x[hz_train, ],
  center = colMeans(hz_x[hz_train, ]),
  scale = apply(hz_x[hz_train, ], 2, sd)
)
yt <- hz$Salary[hz_train] - mean(hz$Salary[hz_train])

ridge_zero <- function(lambda) {
  drop(solve(crossprod(xt) + lambda * diag(ncol(xt)), crossprod(xt, yt)))
}

lasso_zero <- function(lambda, sweeps = 200) {
  # Coordinate descent: each coefficient in turn is set to the soft-thresholded
  # correlation of its feature with the partial residual, over the feature's
  # sum of squares.
  b <- rep(0, ncol(xt))
  for (s in seq_len(sweeps)) {
    for (j in seq_len(ncol(xt))) {
      r <- yt - xt[, -j, drop = FALSE] %*% b[-j]
      z <- sum(xt[, j] * r)
      b[j] <- sign(z) * max(abs(z) - lambda / 2, 0) / sum(xt[, j]^2)
    }
  }
  b
}

lams <- 10^seq(2, 5, by = 0.5)
```

The two languages split the players at random in different ways, so their counts differ. The numbers in the later parts of this solution are from the R split.

Exercise 3.3 (Count the zero coefficients of ridge and the lasso on baseball salaries (part 2))**.** Use the setup of Exercise 3.2.
For each λ, count the coefficients that are exactly zero in each fit.

Solution

Solution 3.3.

Python

```python
pd.DataFrame({
    "lambda": lams_py.round(),
    "ridge_zeros": [int(np.sum(ridge_zero_py(l) == 0)) for l in lams_py],
    "lasso_zeros": [int(np.sum(lasso_zero_py(l) == 0)) for l in lams_py],
})
```

```
     lambda  ridge_zeros  lasso_zeros
0     100.0            0            0
1     316.0            0            1
2    1000.0            0            5
3    3162.0            0           12
4   10000.0            0           14
5   31623.0            0           16
6  100000.0            0           19
```

R

```
zero_counts <- data.frame(
  lambda = round(lams),
  ridge_zeros = sapply(lams, function(l) sum(ridge_zero(l) == 0)),
  lasso_zeros = sapply(lams, function(l) sum(lasso_zero(l) == 0))
)
zero_counts
```

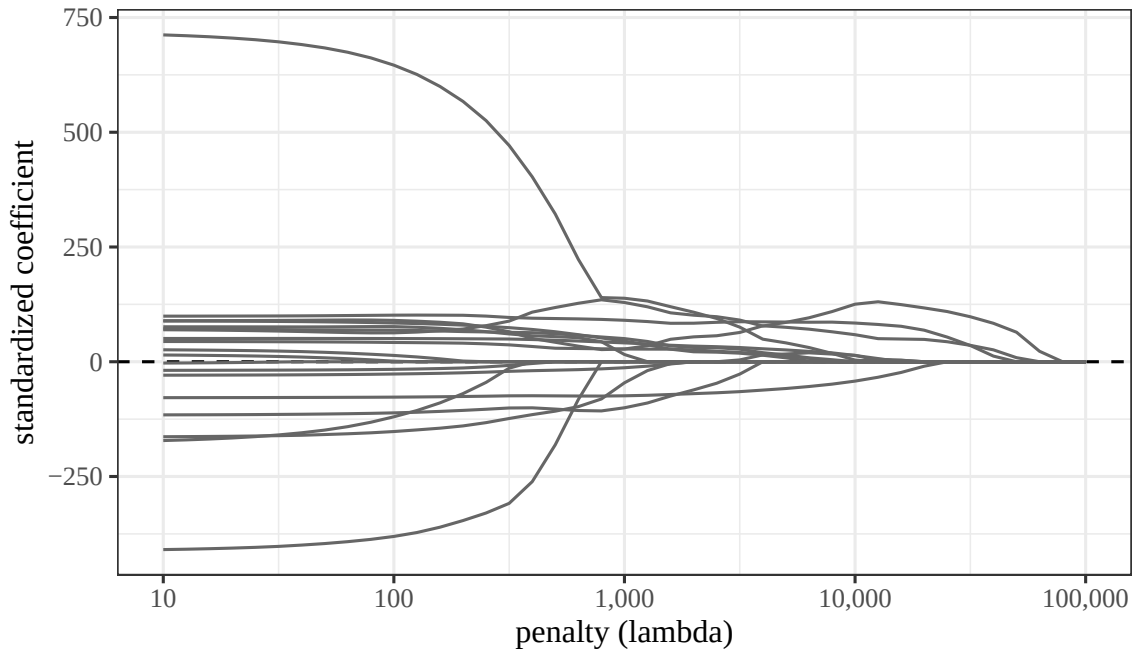
```
# A tibble: 7 x 3
  lambda ridge_zeros lasso_zeros
   <dbl>       <int>       <int>
1    100           0           1
2    316           0           3
3   1000           0           6
4   3162           0           8
5  10000           0          11
6  31623           0          16
7 100000           0          19
```

Ridge leaves every coefficient nonzero at every λ in the grid, so the count of zeros stays at 0. The lasso's count climbs from 1 of the 19 coefficients at $\lambda = 10^2$ to all 19 at $\lambda = 10^5$. Figure 3 plots each lasso coefficient against λ on a finer grid, the counterpart of the ridge paths in Figure 1: the ridge curves shrink smoothly toward zero, while the lasso curves reach zero exactly, one after another.

3.0.1 R

```
# Fit the lasso of the R block at 41 penalties, evenly spaced on a log scale
# from 10 to 100,000, so the paths are smooth between the grid of the exercise.
path_lams <- 10^seq(1, 5, by = 0.1)
n_coef <- ncol(xt)
one_fit <- numeric(n_coef)
lasso_path <- vapply(path_lams, lasso_zero, one_fit)
n_lams <- length(path_lams)
first_fit <- lasso_path[, 1]
last_fit <- lasso_path[, n_lams]
nonzero_first <- sum(first_fit != 0)
nonzero_last <- sum(last_fit != 0)
lambda_column <- rep(path_lams, each = n_coef)
coef_names <- colnames(xt)
predictor_column <- rep(coef_names, times = n_lams)
coefficient_column <- as.vector(lasso_path)
lasso_path_df <- tibble::tibble(
  lambda = lambda_column,
  predictor = predictor_column,
  coefficient = coefficient_column
)
lambda_labels <- scales::label_number(big.mark = ",", drop0trailing = TRUE)

lasso_path_df |>
  ggplot2::ggplot() +
  ggplot2::aes(x = lambda, y = coefficient, group = predictor) +
  ggplot2::geom_hline(yintercept = 0, linetype = "dashed") +
  ggplot2::geom_line(colour = "grey40") +
  ggplot2::scale_x_log10(labels = lambda_labels) +
  ggplot2::labs(x = "penalty (lambda)", y = "standardized coefficient")
```



3.0.2 Python

```
import matplotlib.pyplot as plt

# Fit the lasso of the Python block at 41 penalties, evenly spaced on a log
# scale from 10 to 100,000, so the paths are smooth between the grid of the exercise.
path_lams_py = 10.0 ** np.arange(1, 5.05, 0.1)
lasso_path_py = np.array([lasso_zero_py(l) for l in path_lams_py])
nonzero_first_py = int(np.sum(lasso_path_py[0] != 0))
nonzero_last_py = int(np.sum(lasso_path_py[-1] != 0))
fig, ax = plt.subplots(figsize=(6, 3.5), layout="constrained")
ax.axhline(0, linestyle="--", color="black");
ax.plot(path_lams_py, lasso_path_py, color="grey");
ax.set_xscale("log");
ax.set_xlabel("penalty (lambda)");
ax.set_ylabel("standardized coefficient");
plt.show()
```

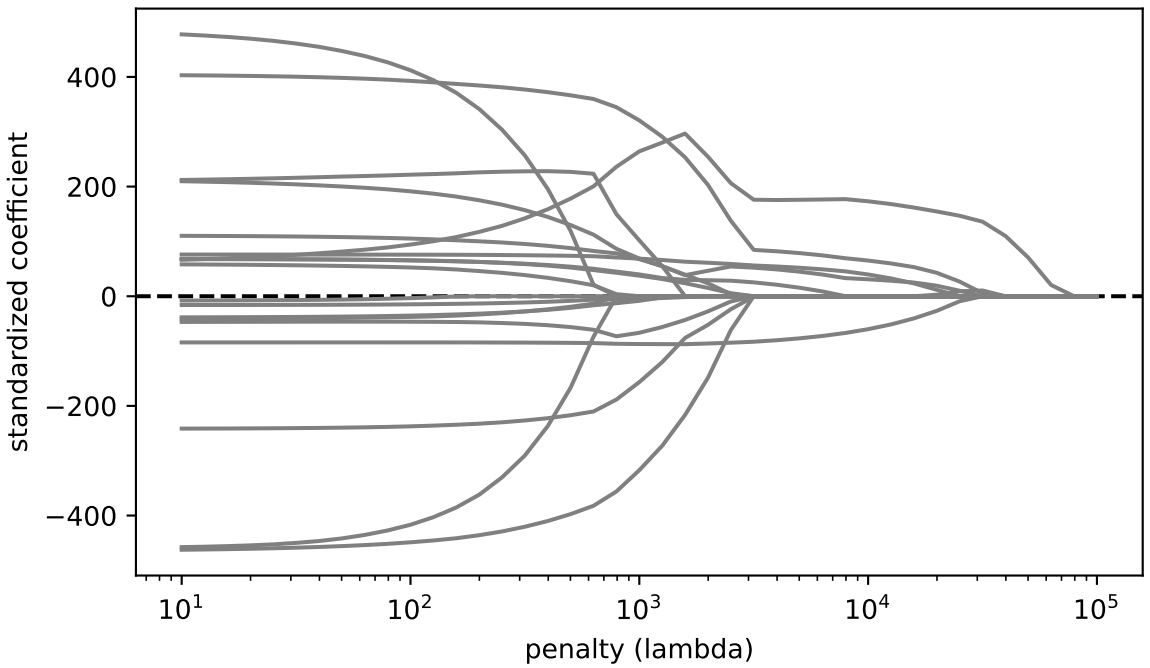


Figure 3: Each curve shows the lasso coefficient of one of the 19 standardized predictors of `Salary` against λ on a logarithmic axis. Each tab fits the lasso to its own language's training half of the `Hitters` data, and the two languages split the players differently, so the tabs' paths differ. At $\lambda = 10$, 19 of the coefficients are non-zero on the R tab and 19 on the Python tab; at $\lambda = 10^5$, 0 are non-zero on the R tab and 0 on the Python tab.

Exercise 3.4 (Count the zero coefficients of ridge and the lasso on baseball salaries (part 3))**.** Use the setup of Exercise 3.2.
Which predictors still have non-zero lasso coefficients at $\lambda = 10^4$?

Solution

Solution 3.4.

Python

```
b_1e4_py = pd.Series(lasso_zero_py(1e4), index=hz_names)
b_1e4_py[b_1e4_py != 0].round()
```

```
Hits           65.0
CHmRun         31.0
CRBI          173.0
PutOuts        40.0
Division_W    -60.0
dtype: float64
```

R

```
# The predictors that survive at lambda = 10^4, with their coefficients.
b_1e4 <- setNames(lasso_zero(1e4), colnames(xt))
round(b_1e4[b_1e4 != 0])
```

```
     Hits       RBI     Walks     CHits    CHmRun      CRBI DivisionW   PutOuts
       14         2        84         4        59       125       -42        14
```

At $\lambda = 10^4$, 8 predictors remain: Hits, RBI, Walks, CHits, CHmRun, CRBI, DivisionW, PutOuts.

Exercise 3.5 (Count the zero coefficients of ridge and the lasso on baseball salaries (part 4))**.** Use the setup of Exercise 3.2.
How do the counts of exactly zero coefficients compare with your prediction in the prediction exercise?

Solution

Solution 3.5. The counts match the prediction of the prediction exercise: ridge only shrinks, and the lasso selects.

Exercise 3.6 (Choose the lasso penalty by cross-validation on baseball salaries)**.** Use the `Hitters` data and the standardized training half of Exercise 3.2. Randomly assign the training players to ten folds. For $\lambda = 10^{1.5}, 10^2, \ldots, 10^{4.5}$, compute the 10-fold cross-validation estimate of the test error of the lasso and of ridge regression, with the lasso computed by the coordinate descent of the lasso coordinate update.

Solution

Solution 3.6.

R

```r
# The standardized training half xt and centered salary yt are from the previous solution.
ridge_fit <- function(x, y, lambda) {
  drop(solve(crossprod(x) + lambda * diag(ncol(x)), crossprod(x, y)))
}

lasso_fit <- function(x, y, lambda, sweeps = 100) {
  b <- numeric(ncol(x))
  r <- y
  ss <- colSums(x^2)
  for (s in seq_len(sweeps)) {
    for (j in seq_len(ncol(x))) {
      z <- sum(x[, j] * r) + ss[j] * b[j]
      b_new <- sign(z) * max(abs(z) - lambda / 2, 0) / ss[j]
      r <- r - x[, j] * (b_new - b[j])
      b[j] <- b_new
    }
  }
  b
}

set.seed(2)
folds <- sample(rep(1:10, length.out = nrow(xt)))
lams_cv <- 10^seq(1.5, 4.5, by = 0.5)

cv_estimate <- function(fit) {
  sapply(lams_cv, function(lambda) {
    sq <- numeric(nrow(xt))
    for (f in 1:10) {
      held <- folds == f
      b <- fit(xt[!held, ], yt[!held], lambda)
      sq[held] <- (yt[held] - xt[held, ] %*% b)^2
    }
    mean(sq)
  })
}

cv_ridge <- cv_estimate(ridge_fit)
cv_lasso <- cv_estimate(lasso_fit)

data.frame(
  lambda = round(lams_cv),
  ridge_cv_rmse = round(sqrt(cv_ridge)),
  lasso_cv_rmse = round(sqrt(cv_lasso))
)
```

```
# A tibble: 7 x 3
  lambda ridge_cv_rmse lasso_cv_rmse
   <dbl>         <dbl>         <dbl>
1     32           322           346
2    100           320           344
3    316           327           340
4   1000           353           331
5   3162           388           328
6  10000           411           331
7  31623           420           367
```

Python

```
# The standardized training half xt and centered salary yt are from the previous solution.
def ridge_fit_py(x, y, lam):
    return np.linalg.solve(x.T @ x + lam * np.eye(x.shape[1]), x.T @ y)

def lasso_fit_py(x, y, lam, sweeps=100):
    b = np.zeros(x.shape[1])
    r = y.copy()
    ss = (x ** 2).sum(axis=0)
    for _ in range(sweeps):
        for j in range(x.shape[1]):
            z = x[:, j] @ r + ss[j] * b[j]
            b_new = np.sign(z) * max(abs(z) - lam / 2, 0.0) / ss[j]
            r -= x[:, j] * (b_new - b[j])
            b[j] = b_new
    return b

folds_py = np.random.default_rng(2).permutation(np.arange(len(yt)) % 10)
lams_cv_py = 10.0 ** np.arange(1.5, 4.6, 0.5)

def cv_estimate_py(fit):
    out = []
    for lam in lams_cv_py:
        sq = np.empty(len(yt))
        for f in range(10):
            held = folds_py == f
            b = fit(xt[~held], yt[~held], lam)
            sq[held] = (yt[held] - xt[held] @ b) ** 2
        out.append(sq.mean())
    return np.array(out)

cv_ridge_py = cv_estimate_py(ridge_fit_py)
cv_lasso_py = cv_estimate_py(lasso_fit_py)

pd.DataFrame({
    "lambda": lams_cv_py.round(),
    "ridge_cv_rmse": np.sqrt(cv_ridge_py).round(),
    "lasso_cv_rmse": np.sqrt(cv_lasso_py).round(),
})
```

```
    lambda  ridge_cv_rmse  lasso_cv_rmse
0     32.0          371.0          376.0
1    100.0          371.0          375.0
2    316.0          377.0          373.0
3   1000.0          397.0          373.0
4   3162.0          426.0          378.0
5  10000.0          444.0          378.0
6  31623.0          452.0          410.0
```

The two languages split the players at random in different ways, so their numbers differ. The numbers in the later parts of this solution are from the R split.

Exercise 3.7 (Choose the lasso penalty by cross-validation on baseball salaries (part 2))**.** Use the setup of Exercise 3.6.

For each method, find the λ with the smallest estimate[a], refit on the whole training half, and compute the root mean squared error on the test half.

[a] https://morrison-lab.github.io/sds/estimation.html#def-estimate

Solution

Solution 3.7.

R

```
lam_ridge <- lams_cv[which.min(cv_ridge)]
lam_lasso <- lams_cv[which.min(cv_lasso)]
c(ridge = lam_ridge, lasso = lam_lasso)
```

```
   ridge    lasso
 100.000 3162.278
```

```
hc_test <- setdiff(seq_len(nrow(hz)), hz_train)
xte <- scale(
  hz_x[hc_test, ],
  center = colMeans(hz_x[hz_train, ]),
  scale = apply(hz_x[hz_train, ], 2, sd)
)
yte <- hz$Salary[hc_test] - mean(hz$Salary[hz_train])

b_ridge <- ridge_fit(xt, yt, lam_ridge)
b_lasso <- lasso_fit(xt, yt, lam_lasso)
rmse_test <- function(b) sqrt(mean((yte - xte %*% b)^2))
c(ridge = rmse_test(b_ridge), lasso = rmse_test(b_lasso))
```

```
   ridge    lasso
373.9591 378.3654
```

Python

```
lam_ridge_py = lams_cv_py[np.argmin(cv_ridge_py)]
lam_lasso_py = lams_cv_py[np.argmin(cv_lasso_py)]
print("chosen lambda:", lam_ridge_py.round(), lam_lasso_py.round())
```

```
chosen lambda: 100.0 316.0
```

```
hc_test = np.setdiff1d(np.arange(len(hz_y)), hz_train)
xte = (hz_x[hc_test] - hz_x[hz_train].mean(axis=0)) / hz_x[hz_train].std(axis=0, ddof=1)
yte = hz_y[hc_test] - hz_y[hz_train].mean()

b_ridge_py = ridge_fit_py(xt, yt, lam_ridge_py)
b_lasso_py = lasso_fit_py(xt, yt, lam_lasso_py)
for name, b in [("ridge", b_ridge_py), ("lasso", b_lasso_py)]:
    print(name, round(float(np.sqrt(np.mean((yte - xte @ b) ** 2)))))
```

```
ridge 325
lasso 331
```

Cross-validation chooses $\lambda = 100$ for ridge regression and $\lambda = 3162$ for the lasso. The test root mean squared error is 374 thousand dollars for ridge regression and 378 for the lasso.

Exercise 3.8 (Choose the lasso penalty by cross-validation on baseball salaries (part 3))**.** Use the setup of Exercise 3.6.
How many lasso coefficients are exactly zero at the chosen λ?

Solution

Solution 3.8.

R

```
c(zeros = sum(b_lasso == 0), coefficients = length(b_lasso))
```

```
       zeros coefficients
           8           19
```

Python

```
print(int(np.sum(b_lasso_py == 0)), "of", len(b_lasso_py), "coefficients are zero")
```

```
1 of 19 coefficients are zero
```

At its choice the lasso sets 8 of the 19 coefficients to exactly zero.

Exercise 3.9 (Choose the lasso penalty by cross-validation on baseball salaries (part 4))**.** Use the setup of Exercise 3.6.
Which method predicts the test half better?

Solution

Solution 3.9. Ridge regression has the lower test error, 374 thousand dollars against 378. The test half played no part in choosing either penalty, so these two errors compare the methods fairly.

4 Principal components regression and partial least squares

Exercise 4.1 (Predict baseball salaries from a few combinations of the predictors)**.** Ridge regression and the lasso keep all 19 predictors of the `Hitters` data and penalize their coefficients. Here we instead replace the predictors by M linear combinations[a] of them, and fit least squares on those combinations alone. We build the combinations in two ways.

- *Greatest variance of the predictors.* Take the first M directions, in order, along which the standardized training predictors vary most. These are the first M right singular vectors of the standardized training predictor matrix (see principal component analysis). Each combination weights the predictors by the entries of one singular vector.
- *Relation to the salary.* Weight each standardized predictor by its simple regression coefficient on the centered salary, and add the weighted predictors to make the first combination. Subtract from every predictor the part of it that the first combination explains, by regressing each predictor on the combination and keeping the residuals. Build the second combination from the residuals in the same way, and repeat.

Use the training half and the standardized predictors of Exercise 2.1, and the ten folds of Exercise 2.5. For each way and each $M = 0, 1, \dots, 19$, compute the 10-fold cross-validation estimate. Take $M = 0$ to predict the training mean[b] salary for every player.

[a] https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-linear-combination
[b] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean

Solution

Solution 4.1. The first way is principal components regression and the second is partial least squares, both defined in this section. Each is a linear combination[a] of the standardized predictors, so each fit has one coefficient per predictor.
For the second way, the residual of a predictor after regressing on a combination $\underset{\sim}{z}$ is $x_j - \underset{\sim}{z}\,\underset{\sim}{z}\cdot x_j / \underset{\sim}{z}\cdot\underset{\sim}{z}$, so

after $m-1$ combinations the residual predictors are $\mathbf{D}\prod_{k<m}(I-\underset{\sim}{\phi}_k\underset{\sim}{p}_k{}^\top)$, where $\underset{\sim}{\phi}_k$ holds the weights of combination k and $\underset{\sim}{p}_k$ the regression coefficients of the predictors on it. That lets the code express every combination in the original standardized predictors.

R

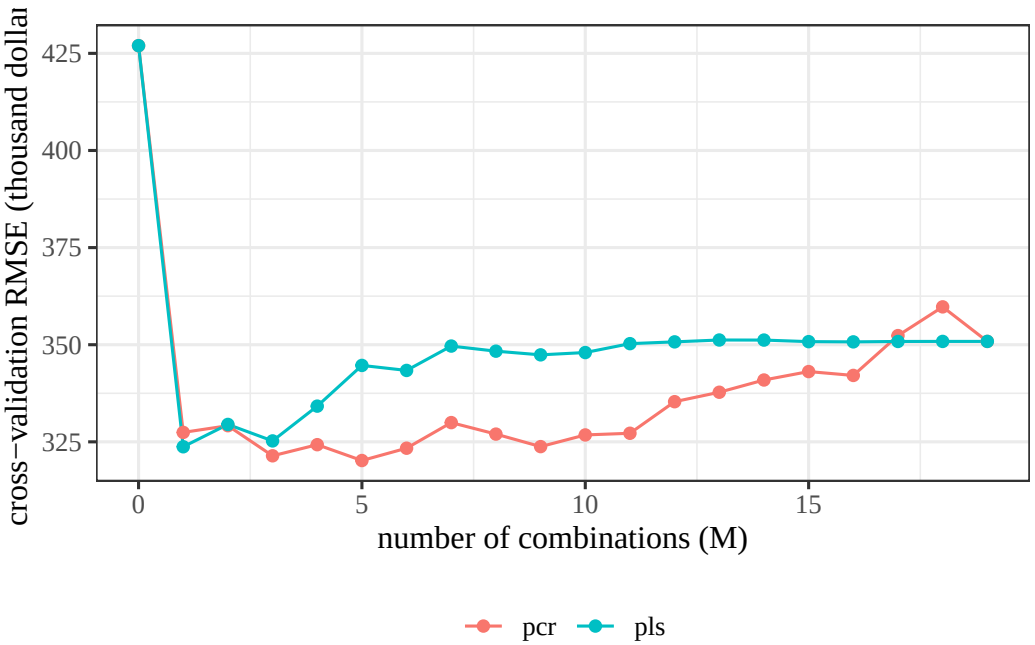
```r
xs_tr <- Xs[train, ]
y_tr <- y[train]
d <- ncol(xs_tr)

pls_dirs <- function(x, y, m) {
  # Columns are the weights of each combination on the original predictors.
  proj <- diag(ncol(x))
  xk <- x
  out <- matrix(0, ncol(x), m)
  for (k in seq_len(m)) {
    phi <- drop(crossprod(xk, y)) / colSums(xk^2)
    z <- drop(xk %*% phi)
    p <- drop(crossprod(xk, z)) / sum(z^2)
    out[, k] <- proj %*% phi
    proj <- proj %*% (diag(ncol(x)) - outer(phi, p))
    xk <- xk - outer(z, p)
  }
  out
}

fit_predict <- function(x, y, m, method, x_new) {
  x_bar <- colMeans(x)
  x_c <- sweep(x, 2, x_bar)
  y_c <- y - mean(y)
  if (m == 0) {
    return(rep(mean(y), nrow(x_new)))
  }
  w <- if (method == "pcr") svd(x_c, nu = 0, nv = m)$v else pls_dirs(x_c, y_c, m)
  z <- x_c %*% w
  beta <- drop(w %*% solve(crossprod(z), crossprod(z, y_c)))
  mean(y) + drop(sweep(x_new, 2, x_bar) %*% beta)
}

methods <- c("pcr", "pls")
cv_sq_dr <- lapply(setNames(methods, methods), function(meth) {
  sq <- matrix(0, length(y_tr), d + 1)
  for (f in seq_len(k_cv)) {
    held <- fold == f
    for (m in 0:d) {
      pred <- fit_predict(xs_tr[!held, ], y_tr[!held], m, meth, xs_tr[held, , drop = FALSE])
      sq[held, m + 1] <- (y_tr[held] - pred)^2
    }
  }
  sq
})

library(ggplot2)
cv_curves <- do.call(rbind, lapply(methods, function(meth) {
  data.frame(method = meth, m = 0:d, cv_rmse = sqrt(colMeans(cv_sq_dr[[meth]])))
}))
ggplot(cv_curves, aes(m, cv_rmse, colour = method)) +
  geom_line() +
  geom_point() +
  labs(x = "number of combinations (M)", y = "cross-validation RMSE (thousand dollars)", colour = NULL
  theme(legend.position = "bottom")
```


cross−validation RMSE (thousand dollar
425
400
375
350
325
0
5
10
15
number of combinations (M)
pcr
pls

Python

```
xs_tr_py = X_py[train_py, 1:]
y_tr_py = y_py[train_py]
d_py = xs_tr_py.shape[1]


def pls_dirs_py(x, y, m):
    # Columns are the weights of each combination on the original predictors.
    proj = np.eye(x.shape[1])
    xk = x
    out = np.zeros((x.shape[1], m))
    for k in range(m):
        phi = (xk.T @ y) / (xk ** 2).sum(axis=0)
        z = xk @ phi
        p = (xk.T @ z) / (z @ z)
        out[:, k] = proj @ phi
        proj = proj @ (np.eye(x.shape[1]) - np.outer(phi, p))
        xk = xk - np.outer(z, p)
    return out


def fit_predict_py(x, y, m, method, x_new):
    x_bar = x.mean(axis=0)
    x_c, y_c = x - x_bar, y - y.mean()
    if m == 0:
        return np.full(len(x_new), y.mean())
    w = (np.linalg.svd(x_c, full_matrices=False)[2][:m].T
         if method == "pcr" else pls_dirs_py(x_c, y_c, m))
    z = x_c @ w
    beta = w @ np.linalg.solve(z.T @ z, z.T @ y_c)
    return y.mean() + (x_new - x_bar) @ beta


methods_py = ["pcr", "pls"]
cv_sq_dr_py = {meth: np.empty((len(y_tr_py), d_py + 1)) for meth in methods_py}
for meth in methods_py:
    for f in range(k_cv):
        held = fold_py == f
        for m in range(d_py + 1):
            pred = fit_predict_py(xs_tr_py[~held], y_tr_py[~held], m, meth, xs_tr_py[held])
            cv_sq_dr_py[meth][held, m] = (y_tr_py[held] - pred) ** 2

fig, ax = plt.subplots(figsize=(6, 4), layout="constrained")
for meth in methods_py:
    ax.plot(range(d_py + 1), np.sqrt(cv_sq_dr_py[meth].mean(axis=0)), marker="o", label=meth);
ax.set_xlabel("number of combinations (M)");
ax.set_ylabel("cross-validation RMSE (thousand dollars)");
ax.legend();
plt.show()
```

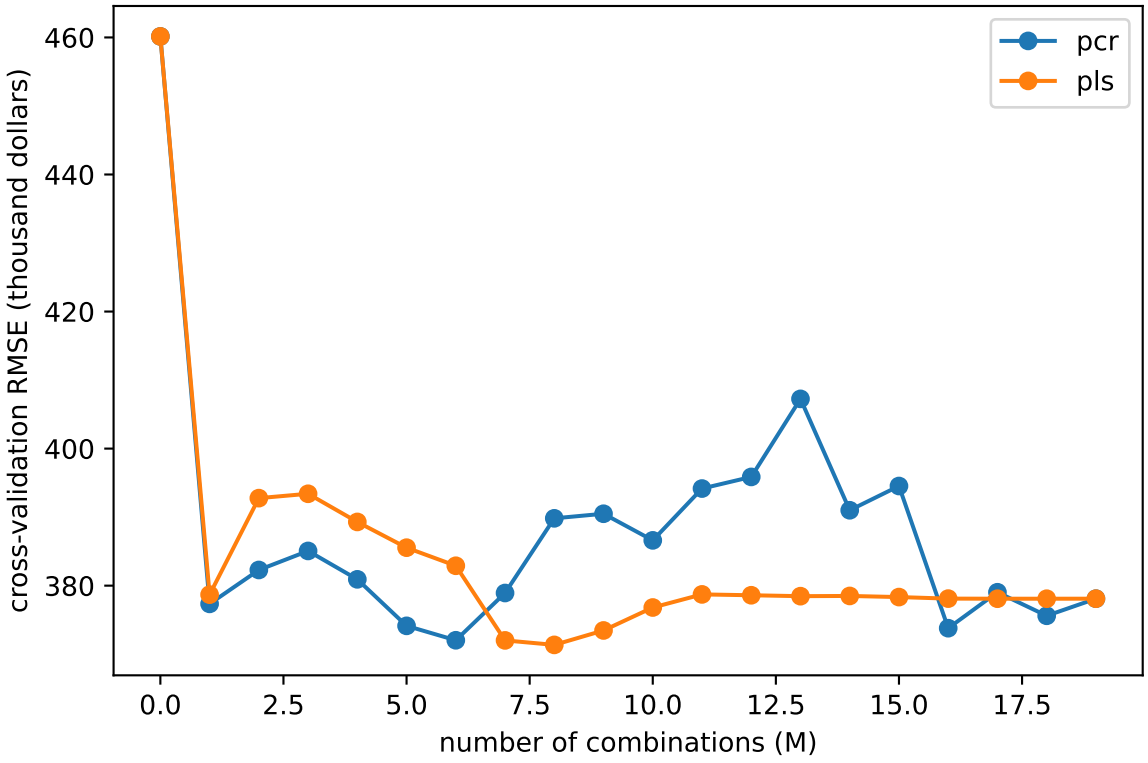


Figure 4: How each of the two ways scores against the number of combinations used.

Figure 4 plots the score of each way against the number of combinations.
The numbers in the later parts of this solution are from the R split and its folds. With $M = 19$ both ways give the least squares fit on all the predictors.

[a] https://morrison-lab.github.io/mds/linear-algebra-vectors.html#def-linear-combination

Exercise 4.2 (Predict baseball salaries from a few combinations of the predictors (part 2))**.** Use the setup of Exercise 4.1.
For each way, which M has the smallest estimate[a]?

[a] https://morrison-lab.github.io/sds/estimation.html#def-estimate

Solution

Solution 4.2.

R

```
dr_cv <- sapply(cv_sq_dr, function(sq) sqrt(colMeans(sq)))
dr_m <- apply(dr_cv, 2, which.min) - 1
rbind(M = dr_m, cv_rmse = apply(dr_cv, 2, min))
```

```
            pcr      pls
M         5.000   1.0000
cv_rmse 320.188 323.7307
```

Python

```python
for meth in methods_py:
    cv_m = cv_sq_dr_py[meth].mean(axis=0)
    m_best = int(np.argmin(cv_m))
    print(meth, "- M", m_best, "- cv sRMSE", round(float(np.sqrt(cv_m[m_best]))))
```

```
pcr - M 6 - cv sRMSE 372
pls - M 8 - cv sRMSE 371
```

Principal components regression chooses $M = 5$ combinations, and partial least squares chooses $M = 1$.

Exercise 4.3 (Predict baseball salaries from a few combinations of the predictors (part 3))**.** Use the setup of Exercise 4.1.
Compute the test errors of the two choices.

Solution

Solution 4.3.

R

```r
dr_test <- sapply(methods, function(meth) {
  pred <- fit_predict(xs_tr, y_tr, dr_m[[meth]], meth, Xs[-train, ])
  sqrt(mean((y[-train] - pred)^2))
})
dr_summary <- data.frame(
  method = methods, m = unname(dr_m),
  cv_rmse = unname(apply(dr_cv, 2, min)), test_rmse = unname(dr_test)
)
dr_summary
```

```
# A tibble: 2 x 4
  method     m cv_rmse test_rmse
  <chr>  <dbl>   <dbl>     <dbl>
1 pcr        5    320.      378.
2 pls        1    324.      390.
```

Python

```python
for meth in methods_py:
    m_best = int(np.argmin(cv_sq_dr_py[meth].mean(axis=0)))
    pred = fit_predict_py(xs_tr_py, y_tr_py, m_best, meth, X_py[test_py, 1:])
    test_rmse_dr = np.sqrt(np.mean((y_py[test_py] - pred) ** 2))
    print(meth, "- test sRMSE", round(float(test_rmse_dr)))
```

```
pcr - test sRMSE 323
pls - test sRMSE 334
```

The test root mean squared error (RMSE) is 378 thousand dollars for principal components regression and 390 for partial least squares.

Exercise 4.4 (Predict baseball salaries from a few combinations of the predictors (part 4))**.** Use the setup of Exercise 4.1.
How do the test errors of the two choices compare with that of ridge regression at the penalty chosen in Exercise 2.6?

Solution

Solution 4.4. Ridge regression at the penalty chosen in Solution 2.6 has a test RMSE of 376, against 378 for principal components regression and 390 for partial least squares.

[1] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, *An introduction to statistical learning: With applications in Python*. Springer, 2023. doi: 10.1007/978-3-031-38747-0[5].

[2] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, "ISLP labs: Lab notebooks for an introduction to statistical learning with applications in python." GitHub repository, BSD 2-Clause license, 2023. Available: https://github.com/intro-stat-learning/ISLP_labs

[3] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning with applications in R*, 2nd ed. Springer, 2021. Available: https://www.statlearning.com/

[5] https://doi.org/10.1007/978-3-031-38747-0