Subset Selection & Shrinkage

Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

Machine Learning for Economics and Finance

Bachelor in Economics

Marcel Weschke

27.05.2025

Learning Goals

At the end of this lecture, you should be able to:

- Understand how we can differentiate between different linear models
- Understand how subset model selection and shrinkage methods work
- Use shrinkage methods to prevent overfitting
- Understand the importance of normalization
- Apply ridge and lasso regressions in Python

Book Chapter: 6

Motivation

- Recall the linear model:

$$Y = \beta_0 + \beta_1 X_1 + ... + \beta_p X_p + \epsilon$$

- If the true relationship between the responses and the predictors is approximately linear, the model is a good choice
- However, we still face the question which features to include
- Especially if p is large (relative to the sample size n), including features that have low predictive power can lead to bad out-of-sample predictions
- By excluding features interpretability of the model increases
- **Key question**: How can we (automatically) select relevant features in our linear model?

Three classes of methods

- Subset Selection
  - We identify a subset of the p predictors that we believe to be related to the response.
  - We then fit a model using least squares on the reduced set of variables
- Shrinkage
  - We fit a model involving all p predictors, but some of the estimated coefficients are shrunken towards zero relative to the least squares estimates.
  - This shrinkage ("regularization") automatically selects relevant variables
- Dimension Reduction (not covered in this course)

Best subset selection

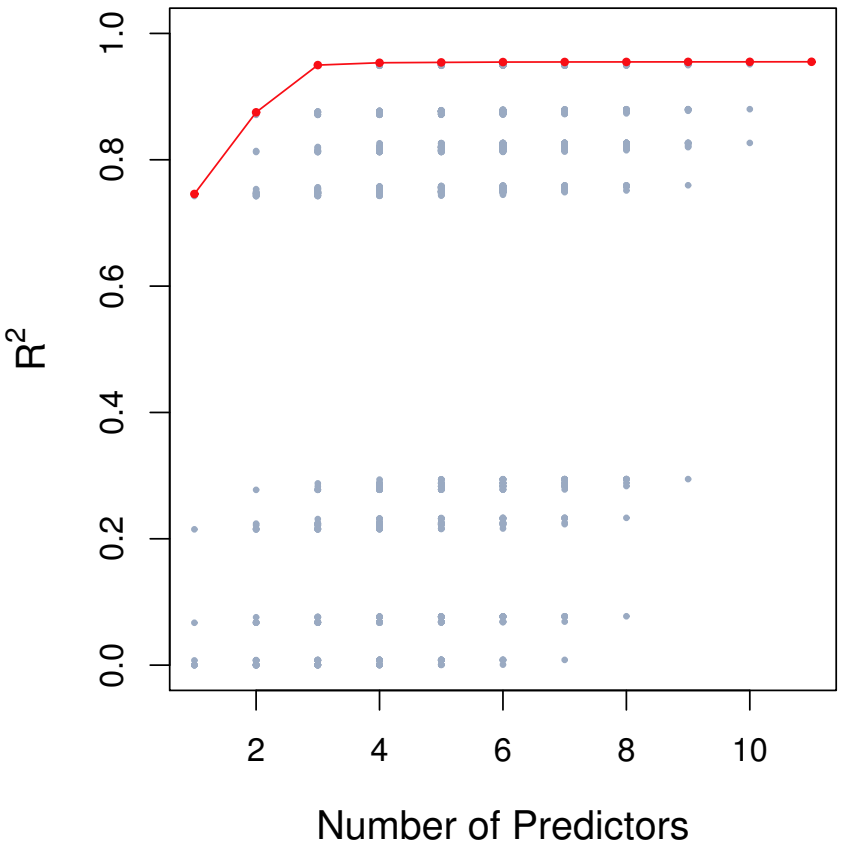
- Best subset means try all possible models
- Let $\mathcal{M}_0$ be the regression model with $p = 0$ predictors so that it predicts the sample mean for each observation

For $k = 1, 2, ..., p$:

- Fit all $\binom{p}{k} = \frac{p!}{k!(p-k)!}$ models that contain k predictors
- Pick the model among these $\binom{p}{k}$ models that has the largest R2, and call it $\mathcal{M}_k$
- Select a single best model from among $\mathcal{M}_0, \mathcal{M}_1, ..., \mathcal{M}_p$ using cross-validated prediction error

Example: Credit data set

- The R2 are displayed for each possible model containing a subset of the eleven predictors in the Credit data set



Remarks: Best subset selection

- Best subset selection cannot be applied with very large p as it becomes computationally very "expensive"
- Best subset selection often leads to overfitting
- $\Rightarrow$ Often poor predictive power when the final model is used for making out-of-sample predictions

Stepwise Selection

- Stepwise methods, which explore a more restricted set of models, are an alternative to best subset selection
- We have Forward Stepwise Selection, Backward Stepwise Selection and hybrid approaches
- We only cover Forward Stepwise Selection, for other methods see book

Forward Stepwise Selection

- Forward stepwise selection begins with a model containing no predictors, and then adds predictors to the model, one-at-a-time, until all of the predictors are in the model.
- At each step the variable that gives the greatest additional improvement to the fit is added to the model
- Computational advantage over best subset selection is clear
- It is not guaranteed to find the best possible model out of all 2^p models containing subsets of the p predictors

Forward Stepwise Selection: Algorithm

- Let $\mathcal{M}_0$ denote the modell with no predictors (only a constant)

- For $k = 0, 1, ..., p-1$:
  - Consider all the $p-k$ models that augment the predictor in $\mathcal{M}_k$ with one additional predictor
  - Choose the best among these models (highest in-sample R^2) and call it $\mathcal{M}_{k+1}$

- Select a single best model among $\mathcal{M}_0, \mathcal{M}_1, ..., \mathcal{M}_p$ using cross-validated prediction error

Python code: Forward stepwise selection - 1

Python-file: *04_ForwardSelection.ipynb*:

```
1  import pandas as pd
2  import numpy as np
3  from sklearn.linear_model import LinearRegression
4  from sklearn.feature_selection import SequentialFeatureSelector
5  from ISLP import load_data
6
7  # Load Hitters dataset from ISLP
8  Hitters = load_data('Hitters')
9
10 # Remove missing values
11 Hitters = Hitters.dropna()
12
13 # Create dummy variables for categorical columns
14 Hitters = pd.get_dummies(Hitters, drop_first=True)
15
16 # Separate response (target) and predictors
17 y = Hitters['Salary']
18 X = Hitters.drop(columns=['Salary'])
```

Python code: Forward stepwise selection - 2

```
20 # Define the linear regression model
21 model = LinearRegression()
22
23 # Perform forward stepwise selection using "SequentialFeatureSelector"
24 sfs = SequentialFeatureSelector(model, n_features_to_select=15,
   ↪  direction='forward')
25
26 # Fit the model to the data
27 sfs.fit(X, y)
28
29 # Get the selected features
30 selected_features = X.columns[sfs.get_support()]
31
32 # Fit the model with the selected features
33 model.fit(X[selected_features], y)
34
35 # Coefficients of the selected features
36 coefficients = pd.DataFrame({
37  'Feature': selected_features,
38  'Coefficient': model.coef_
39 })
```

Python code: Forward stepwise selection - 3

```
41 # Printing short summary - intercept, coefficients and R^2
42 print("\nIntercept:")
43 print(model.intercept_)
44
45 print("\nCoefficients:")
46 print(coefficients)
47
48 print("\nR-squared:")
49 print(model.score(X[selected_features], y))
```

Python code: Validation errors for FSS cont' - 1

```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
import statsmodels.api as sm

# Split the data into training and validation sets based on row indices
train_data = Hitters.iloc[:184]   # First 184 rows for training data
val_data = Hitters.iloc[184:263]  # Rows 185 to 263 for validation data

# Define X and y for both training and validation sets
X_train = train_data.drop(columns=['Salary'])
y_train = train_data['Salary']
X_val = val_data.drop(columns=['Salary'])
y_val = val_data['Salary']

# Ensure that all categorical variables are encoded as numeric
X_train = pd.get_dummies(X_train, drop_first=True).astype(float)
X_val = pd.get_dummies(X_val, drop_first=True).astype(float)

# Align columns of validation set to match training set
X_val = X_val.reindex(columns=X_train.columns,
↪  fill_value=0).astype(float)
```

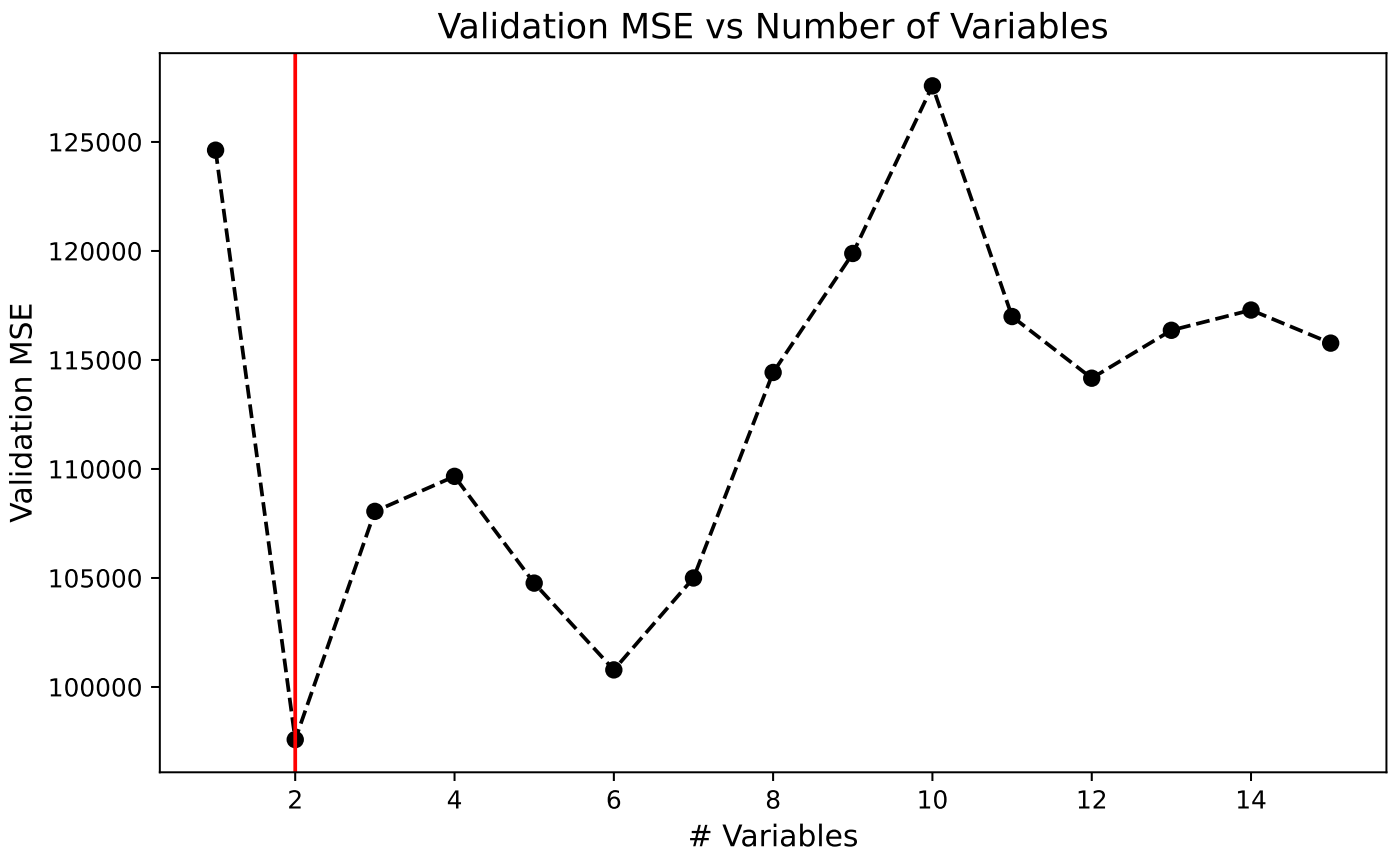
Python code: Validation errors for FSS - 2

```
23 # Convert validation data to matrix form (for statsmodels)
24 val_data = sm.add_constant(X_val)
25
26 # Ensure target variable is numeric
27 y_train_np = np.asarray(y_train).astype(float)
28 y_val_np = np.asarray(y_val).astype(float)
29
30 sfs2 = SFS(model,
31       k_features=15,
32       forward=True,
33       floating=False,
34       scoring='neg_mean_squared_error',
35       cv=0)  # No cross-validation
36
37 sfs2.fit(X_train, y_train)
38
39 # Extract selected features for each number of features (1 to 15)
40 selected_features = sfs2.subsets_
```

Python code: Validation errors for FSS - 3

```
42  # Compute validation mean squared errors for each model
43  val_err = np.zeros(15)
44  for i in range(1, 16):
45    # Get the selected feature names for this step
46    feature_names = selected_features[i]['feature_names']
47    # Select the corresponding features from the training set
48    X_train_selected = X_train[list(feature_names)]
49    # Add constant (intercept) term
50    X_train_selected = sm.add_constant(X_train_selected).astype(float)
51    # Ensure the selected features are numeric
52    X_train_selected_np = np.asarray(X_train_selected).astype(float)
53    # Fit OLS model
54    model = sm.OLS(y_train_np, X_train_selected_np).fit()
55    # Predict on validation set
56    X_val_selected = val_data[list(feature_names)]
57    X_val_selected_np = sm.add_constant(X_val_selected).astype(float)
58    y_pred_val = model.predict(X_val_selected_np)
59    # Compute MSE for validation set
60    val_err[i - 1] = MSE(y_val_np, y_pred_val)
```

Python code: Forward stepwise selection cont'



Shrinkage Methods

- Shrinkage methods offer an alternative to subset selection
- Can handle many potentially highly correlated features (multicollinearity)
- **Idea**: Shrink the coefficients of a linear regression towards zero to prevent overfitting so that the model doesn't rely too much on certain features
- Can achieve both an interpretable model and better out-of-sample predictions
- We consider two shrinkage methods: **Ridge Regression** and **Lasso**

Ridge regression

- Recall that the least squares fitting procedure estimates $\hat{\beta} = (\hat{\beta}_0, \hat{\beta}_1, \dots \hat{\beta}_p)$ using the values that minimize

$$RSS = \sum_{i=1}^{n} \left(y_i - \hat{\beta}_0 - \sum_{j=1}^{p} \hat{\beta}_j x_{i,j} \right)^2 \tag{1}$$

- The ridge regression coefficients $\hat{\beta} = (\hat{\beta}_0, \hat{\beta}_1, \dots \hat{\beta}_p)$ are the values that minimize

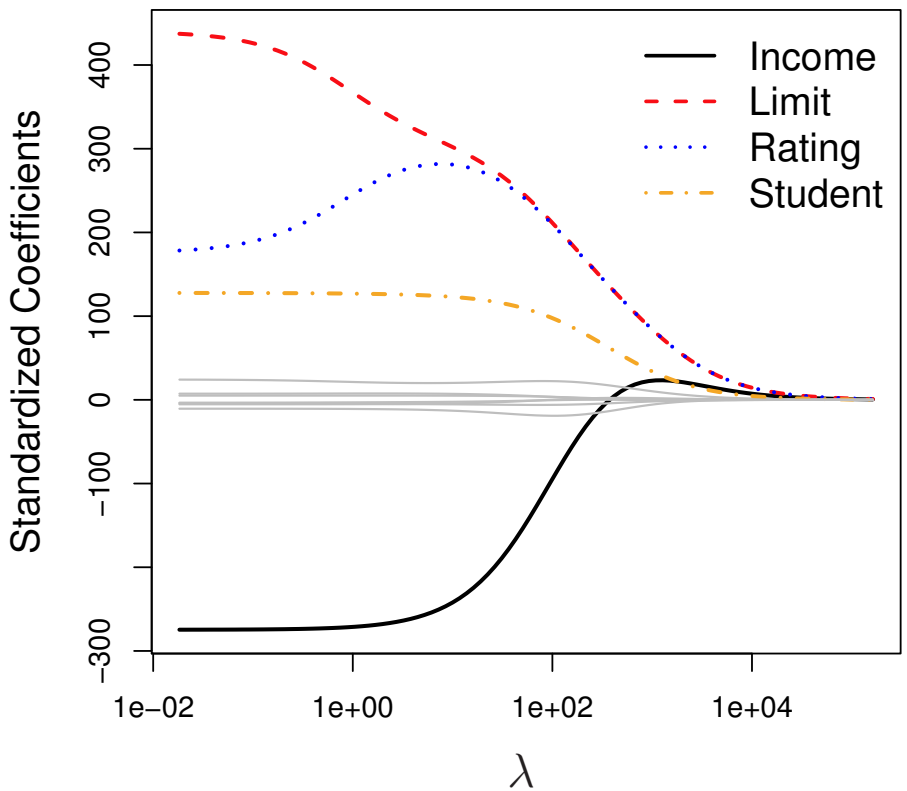
$$\sum_{i=1}^{n} \left(y_i - \hat{\beta}_0 - \sum_{j=1}^{p} \hat{\beta}_j x_{i,j} \right)^2 + \lambda \sum_{j=1}^{p} \hat{\beta}_j^2 \tag{2}$$

```
1 ridge.mod = skl.ElasticNet(alpha=100, l1_ratio=0)
```

Ridge regression cont'

- As with least squares, ridge regression seeks coefficient estimates that fit the data well, by making the RSS small
- The second term $\lambda \sum_{j=1}^{p} \beta_j^2$ is called a shrinkage penalty
- The tuning parameter $\lambda \geq 0$ controls the "shrinkage intensity"
- For $\lambda > 0$, large coefficients β_j are "penalized" (*pushed* towards zero) so that the model does not rely too much on a given variable
- Q: What happens in the two extreme cases when $\lambda \to 0$ and $\lambda \to \infty$?

Ridge Regression: Credit data



Coefficients $\hat{\beta}_j$ from ridge regressions using different values for the tuning parameter λ

How to choose the tuning parameter λ?

- There is no general rule on how to choose λ
- $\Rightarrow$ Use cross validation:
  - Split the sample into training and validation data
  - Set up a grid of values for λ
  - For each λ use the training data to fit the ridge regression
  - Use the ridge regression estimates for each λ to compute the corresponding prediction errors in the validation set
  - Choose the model (tuning parameter) for which the validation error is smallest
  - Fit the final model again using both, the training and validation data
- Even better: Instead of using a simple validation set approach to compute validation errors, evaluate each ridge regression using k-fold cross validation

```
1 lambdas = 10**np.linspace(8, -2, 100) / y.std()
2 cv.out = skl.ElasticNetCV(alphas=lambdas, l1_ratio=0, cv=10)
```

Ridge regression: scaling of predictors

- In linear regressions, the standard least squares estimates are scale invariant
- That is, if we multiply a feature X_j by some constant K, the corresponding OLS coefficient $\hat{\beta}_j$ will simply be given by $\hat{\beta}_j/K$ so that $\hat{\beta}_j X_j$ remains the same

  $\Rightarrow$ Scaling of a variable does not change the outputs of a linear regression

Ridge regression: scaling of predictors

- For ridge this does not hold
- Suppose we multiply feature X_j by $K = 1/1000$
- The OLS estimate will be given by $1000\hat{\beta}_j$
- Recall the ridge estimator:

$$\min_{\hat{\beta}} \sum_{i=1}^{n} \left(y_i - \hat{\beta}_0 - \sum_{j=1}^{p} \hat{\beta}_j x_{i,j} \right)^2 + \lambda \sum_{j=1}^{p} \hat{\beta}_j^2 \tag{3}$$

- Now the penalty $\lambda\hat{\beta}_j^2$ will be significantly larger (even though we didn't change the information content of the feature)

$\Rightarrow$ Scaling of features is very important

Normalization

- Before fitting a model, normalize the data:
  - For each feature compute its mean and standard deviation
  - Scale each observation of the feature by subtracting the corresponding mean and dividing by the corresponding standard deviation:

$$\tilde{x}_{i,j} = \frac{x_{i,j} - mean(x_j)}{sd(x_j)}$$

- Sometimes Machine Learning Libraries do this for you, or come with the "pre-process" option
- Make sure to use the same normalization procedure in the training, validation and test set
- Normalization is in general a good idea not only for ridge (and for many methods necessary)

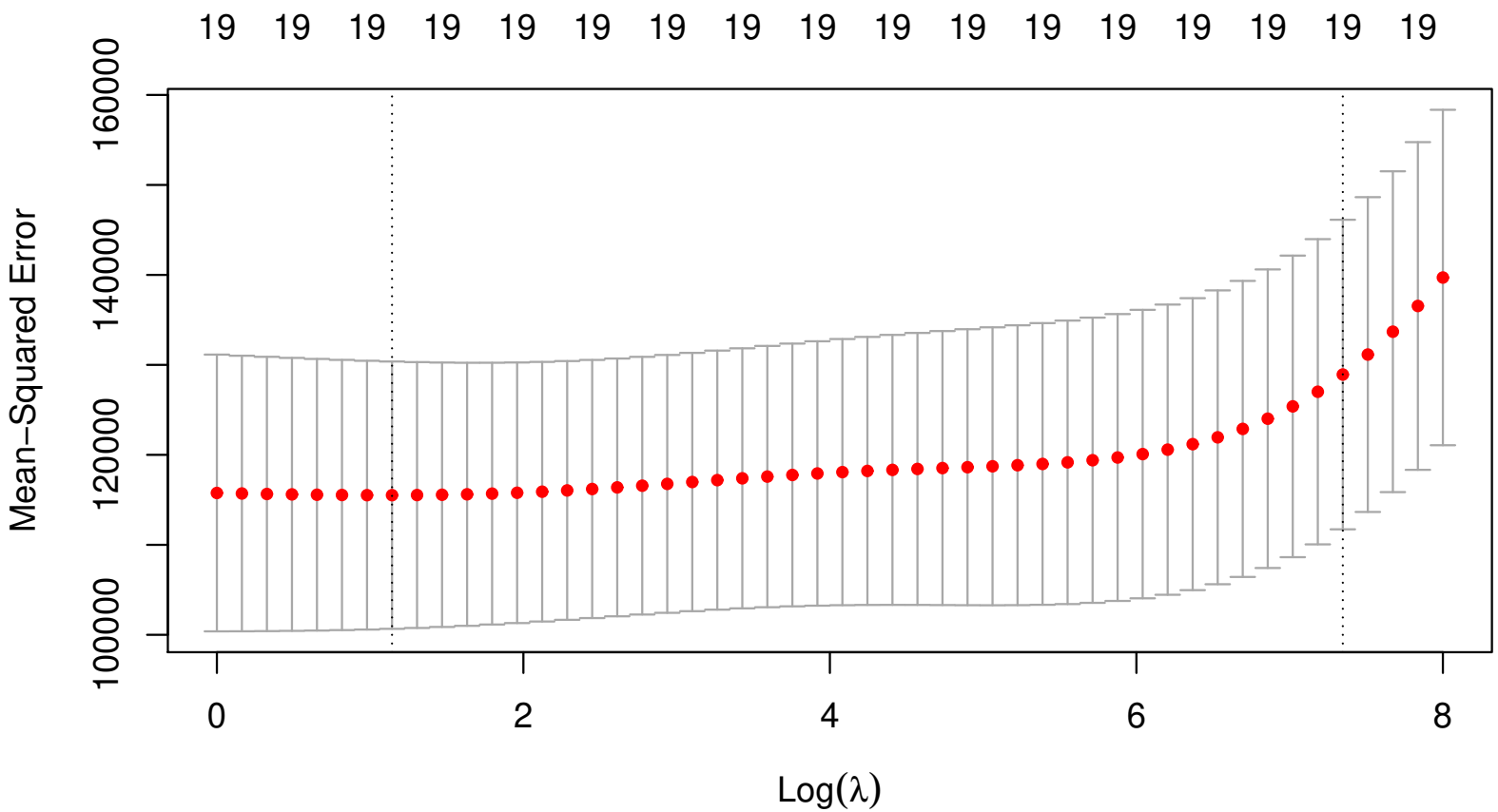
Python code: Ridge regression 1 (File: *04_RidgeRegression.ipynb*)

```python
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge

# Standardize predictors
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Fit Ridge regression model with fixed regularization parameter
↪  lambda=100
ridge_fixed = Ridge(alpha=100)
ridge_fixed.fit(X_scaled, y)

# Extract model coefficients
ridge_fixed_coeffs = ridge_fixed.coef_

# Make predictions on the first 5 observations
ridge_fixed_preds = ridge_fixed.predict(X_scaled[:5])
```

Python code: Ridge regression 2 (File: *04_RidgeRegression.ipynb*)

```
17 from sklearn.linear_model import RidgeCV
18
19 # Ridge regression with cross-validation to find best lambda
20 # Define grid of lambda values
21 alphas = 10**np.linspace(10, -2, 100) * 0.5  # Equivalent to R's lambda
   ↪  grid
22
23 # Perform 10-fold cross-validation to find the optimal lambda
24 ridge_cv = RidgeCV(alphas=alphas, scoring='neg_mean_squared_error',
   ↪  cv=10)
25 ridge_cv.fit(X_scaled, y)
26
27 # Extract the best lambda and corresponding coefficients
28 best_lambda_ridge = ridge_cv.alpha_
29 ridge_cv_coeffs = ridge_cv.coef_
30
31 # Make predictions on the first 5 observations using the best model
32 ridge_cv_preds = ridge_cv.predict(X_scaled[:5])
```

Output: plot(cv.out)



The Lasso

- Ridge regression does have one obvious disadvantage; it will include all p predictors in the final model (even though some $\hat{\beta}_j$ might be close to 0)
- The Lasso is a good alternative to ridge regression that overcomes this disadvantage
- It implicitly forces some $\hat{\beta}_j$ to be equal to zero
- The lasso coefficients minimize

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{i,j}\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j|$$

Python code: Lasso regression (File: *04_RidgeRegression.ipynb*)

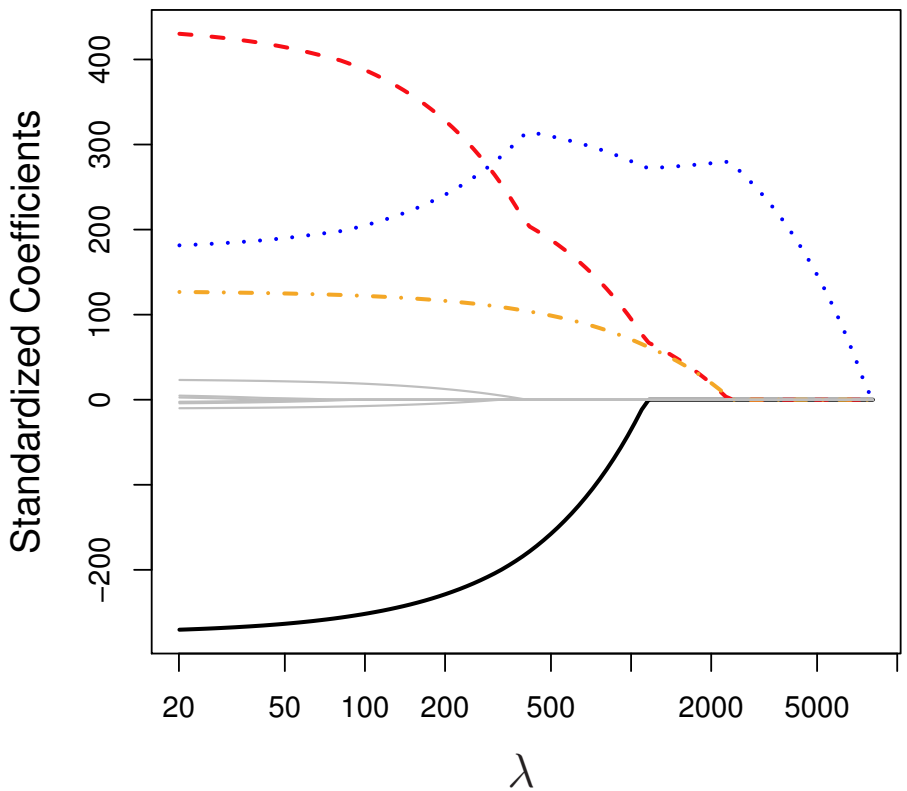
- As ridge regressions, the lasso shrinks the coefficient estimates towards zero
- In contrast to ridge regression, the "Lasso penalty" can force some of the coefficient estimates to be exactly equal to zero
- We say that the lasso yields "sparse models" (few variables)

```
33 from sklearn.linear_model import LassoCV
34
35 # Fit Lasso regression with automatic lambda tuning via cross-validation
36 lasso_cv = LassoCV(cv=10, max_iter=10000)
37 lasso_cv.fit(X_scaled, y)
38
39 # Get the best lambda (regularization strength) chosen by CV
40 best_lambda_lasso = lasso_cv.alpha_
41
```

Python code: Lasso regression (File: *04_RidgeRegression.ipynb*)

```
42 # Extract coefficients at best lambda (some may be zero due to variable
   ↪ selection)
43 lasso_cv_coeffs = lasso_cv.coef_
44
45 # Predict on the first 5 observations
46 lasso_cv_preds = lasso_cv.predict(X_scaled[:5])
```

Lasso Regressions: Credit data



Coefficients $\hat{\beta}_j$ from Lasso regressions using different values for the tuning parameter λ

Comparing the Lasso and Ridge Regression

- Neither ridge regression nor the lasso will universally dominate the other
- Might expect the lasso to perform better when the response is a function of only a relatively small number of predictors
- But the number of predictors that is related to the response is never known a priori for real data sets
- Use cross-validation to determine which approach is better in a particular setting

Python code: Ridge/Lasso regression Summary (File: *04_RidgeRegression.ipynb*)

```
# Create summary table showing key results across models
summary = pd.DataFrame({
 'Model': ['Ridge (lambda=100)', 'RidgeCV (best lambda)', 'LassoCV (best
 ↪  lambda)'],
 'Best Lambda': [100, best_lambda_ridge, best_lambda_lasso],
 'Non-zero Coefficients': [
 np.sum(ridge_fixed_coeffs != 0),
 np.sum(ridge_cv_coeffs != 0),
 np.sum(lasso_cv_coeffs != 0)
 ]
})

# Display the result summary
print(summary)
```

```
                    Model      Best Lambda      Non-zero Coefficients
0    Ridge  (lambda=100)        100.000000                         19
1  RidgeCV (best lambda)          3.067954                         19
2  LassoCV (best lambda)          2.552821                         13
```

Task:

Start with the Python-File *04_RidgeRegression.ipynb*

1. Use the final model (tuning parameter) obtained from 10-fold CV and fit the model again using the full dataset and display the corresponding coefficients.
2. Multiply the feature *Errors* by 1/1000 and again fit the model from Task 1. Display the coefficients and interpret.
3. Now look at the *glmnet* documentation and set an option such that *glmnet* does NOT normalize (standardize) the data. Refit the same model again and display the coefficients. Interpret.
4. Split the dataset into a training set using 80% of the observations and validation set using all other observations.
5. Set up a grid for the tuning parameter λ and fit Lasso regressions for all tuning parameters using the training data. Make sure that you choose the mininmum and maximum values of λ so that it allows you to determine the optimal λ parameter in the next task (you might need to play with the grid size a bit).
6. For each model (tuning parameter), compute the mean squared prediction error in the validation dataset. Plot the validation error as a function of λ and find the best model which minimizes the validation error. Display the estimated coefficients for the best model and check whether some features are not selected in the final regression.
7. Finally compare the best Lasso model obtained from the validation set approach from Task 6 to the best Lasso model obtained by 5-fold cross-validation.
8. Compare the best model from Task 7 to the best ridge regression obtained from 5-fold cross validation. How do the coefficients of the two models differ?