Summary of Chapter 4: Classification

Introduction

Chapter 4 focuses on **Classification** methods, which aim to predict a *qualitative* response (also often called a *categorical* response). Unlike regression techniques that model quantitative outcomes, classification methods assign observations to one of several discrete classes. Classification techniques appear in a variety of real-world applications, such as determining whether a patient has a particular disease (yes/no), deciding if a credit card transaction is fraudulent (yes/no), or classifying the category of an image (e.g., dog, cat, bird).

This chapter presents multiple approaches to classification. It begins with a rationale for why simple linear regression is usually not suitable for a qualitative response. Then it introduces **Logistic Regression** as a foundational approach for binary classification. The chapter subsequently covers **Linear Discriminant Analysis (LDA)**, **Quadratic Discriminant Analysis (QDA)**, **Naive Bayes**, and **K-Nearest Neighbors (KNN)**. It also outlines how to evaluate and compare these methods, including the use of confusion matrices, sensitivity, specificity, and ROC curves. Towards the end, it discusses *Generalized Linear Models (GLMs)* more broadly, focusing on Poisson Regression as another instance of the GLM family that is suited to count data rather than strictly binary or continuous responses. Finally, the chapter includes labs demonstrating each of these methods in the `R` programming environment.

In this summary, each major section is reviewed in turn, emphasizing the core concepts, the assumptions behind the models, and the practical performance considerations.

1 Overview of Classification

Classification problems involve a response variable Y taking on qualitative values (classes). The fundamental goal is to assign an observation to the correct category based on a set of predictors $\mathbf{X} = (X_1, X_2, \ldots, X_p)$. Examples include:

- Determining a medical diagnosis (e.g., `Stroke`, `Overdose`, or `Seizure`) given a patient's vital signs and other clinical predictors.
- Predicting whether an individual will default on a credit payment (`Yes` vs. `No`) based on financial predictors such as credit balance and income.
- Detecting fraudulent online transactions (`Fraudulent` vs. `Legitimate`).

The data consist of *training samples* $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, where each observation has p predictors $\mathbf{x}_i$ and a class label y_i. We seek to build a model that can correctly classify new, previously unseen observations.

A concept repeatedly emphasized in the chapter is that classification methods aim either:

- to *directly* estimate the posterior probability $P(Y = k \mid \mathbf{X} = \mathbf{x})$ for each class k (e.g., logistic regression, naive Bayes), or
- to *indirectly* model the distribution of predictors $\mathbf{X}$ *within each class* and then apply *Bayes' theorem* (as in LDA and QDA).

1.1 Why Not Use Linear Regression?

One might attempt to fit a linear regression of the form $Y \approx \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$, even if Y is binary (coded 0 or 1). However, Chapter 4 points out several issues:

1. Predicted values might lie outside $[0, 1]$, making them invalid probabilities.
2. Linear regression treats the response as quantitative, implying equal spacing between classes when more than two classes exist.
3. Statistical properties (like normal residual assumptions) break down when modeling a categorical response using linear regression.

Hence, specialized classification methods are preferred.

2 Logistic Regression

2.1 Logistic Model Formulation

Logistic regression addresses the binary response setting (two classes, often labeled `Yes` vs. `No`, or `1` vs. `0`). Instead of modeling Y itself, we model

$$p(\mathbf{x}) = P(Y = 1 \mid \mathbf{X} = \mathbf{x}),$$

the probability of class `1` given predictors $\mathbf{x}$. Logistic regression posits that

$$\log\left(\frac{p(\mathbf{x})}{1 - p(\mathbf{x})}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p,$$

where $\log\left(\frac{p}{1-p}\right)$ is known as the *logit* or *log-odds* function. Hence, it ensures $0 \le p(\mathbf{x}) \le 1$ for any linear function of $\mathbf{x}$.

2.2 Coefficient Estimation & Interpretation

The parameters β_j are usually estimated via *maximum likelihood estimation (MLE)*:

$$\ell(\beta_0, \beta_1, \ldots, \beta_p) = \prod_{i=1}^{n} \left[p(\mathbf{x}_i)\right]^{y_i} \left[1 - p(\mathbf{x}_i)\right]^{(1-y_i)}.$$

Solving this optimization yields $\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_p$, which maximize the likelihood of the observed data. The estimates are often summarized in terms of z-statistics and p-values, analogous to linear regression but using logistic-specific assumptions.

Each β_j can be interpreted in terms of *odds ratio*: a one-unit increase in x_j multiplies the odds $p/(1-p)$ by e^{β_j}, holding other predictors fixed.

2.3 Prediction and Classification

After fitting the model, one obtains predicted probabilities

$$\hat{p}(\mathbf{x}) = \frac{e^{\hat{\beta}_0+\hat{\beta}_1 x_1+\cdots+\hat{\beta}_p x_p}}{1+e^{\hat{\beta}_0+\hat{\beta}_1 x_1+\cdots+\hat{\beta}_p x_p}}.$$

A default or typical rule is to predict `Yes` if $\hat{p}(\mathbf{x}) > 0.5$ and `No` otherwise, though thresholds can be tuned depending on the problem context (e.g., cost of false positives vs. false negatives).

2.4 Example: Default Data

The chapter uses `Default` data to predict whether an individual will default on a credit card. Two main predictors are credit `balance` and `income`. Logistic regression shows that `balance` is strongly associated with default probability. Adjusting the threshold can shift the balance between correctly capturing defaulters (high *sensitivity*) and avoiding false alarms.

3 Linear Discriminant Analysis (LDA)

3.1 Generative Perspective and Bayes' Theorem

LDA provides an alternative approach by first modeling the distribution of the predictors X *within each class*. Under Bayes' theorem, the posterior probability that $Y = k$ given $X = \mathbf{x}$ is

$$P(Y = k \mid \mathbf{x}) = \frac{\pi_k\, f_k(\mathbf{x})}{\sum_{\ell=1}^{K} \pi_\ell\, f_\ell(\mathbf{x})},$$

where $\pi_k = P(Y = k)$ is the prior probability of class k, and $f_k(\mathbf{x})$ is the class-conditional density. For LDA, we assume

$$\mathbf{x} \,\big|\, (Y = k) \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}),$$

that is, each class has a normal (Gaussian) distribution with class-specific mean vector $\boldsymbol{\mu}_k$ but *common* covariance matrix $\boldsymbol{\Sigma}$.

3.2 LDA Decision Boundaries

Plugging in the Gaussian densities into the Bayes formula shows that the decision rule is based on

$$\delta_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_k - \tfrac{1}{2} \boldsymbol{\mu}_k^T \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_k + \log \pi_k.$$

LDA classifies to the class k for which $\delta_k(\mathbf{x})$ is largest. Because $\mathbf{x}$ enters linearly ($\mathbf{x}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_k$), the boundaries are *linear* in x.

3.3 Example: Smarket Data

The chapter applies LDA to `Smarket`, which includes daily financial returns on the S&P 500. It demonstrates how one can estimate π_k, $\boldsymbol{\mu}_k$, and $\boldsymbol{\Sigma}$ from training data, then predict whether the market will go `Up` or `Down` on a future day. LDA often yields decision boundaries that are well-suited to roughly linear separation of classes in the predictor space.

4 Quadratic Discriminant Analysis (QDA)

QDA is similar to LDA but allows each class to have its own covariance matrix $\mathbf{\Sigma}_k$. That is, we still assume

$$\mathbf{x} \,\big|\, (Y = k) \sim \mathcal{N}(\boldsymbol{\mu}_k, \mathbf{\Sigma}_k),$$

but *no* longer require $\mathbf{\Sigma}_1 = \mathbf{\Sigma}_2 = \cdots = \mathbf{\Sigma}_K$. In that case, the decision boundary derived from Bayes' rule is *quadratic* in x, so QDA can capture more complex class boundaries.

However, QDA requires estimating many more parameters (each $\mathbf{\Sigma}_k$ has $p(p+1)/2$ parameters), and so is more prone to high variance if n (the sample size) is not large enough. When the class covariances truly differ, QDA can outperform LDA.

5 Naive Bayes

Another generative classification method is **Naive Bayes**, which also uses Bayes' theorem to compute posterior probabilities $P(Y = k \mid \mathbf{x})$. However, instead of assuming *multivariate* normal distributions for X, it makes a “naive” conditional independence assumption: within each class k, the p predictors $X_1, \ldots, X_p$ are treated as independent. Formally:

$$f_k(\mathbf{x}) = \prod_{j=1}^{p} f_{kj}(x_j).$$

Though naive in many real applications (since predictors often correlate), this assumption drastically reduces the number of parameters. Naive Bayes can work surprisingly well, especially in high-dimensional contexts or when n is small. It does well if the independence assumption is even *approximately* met, or if reducing variance outweighs the bias introduced by the assumption of independence.

6 K-Nearest Neighbors (KNN)

KNN is a fully non-parametric approach:

1. Choose K, the number of neighbors.
2. For a new observation $\mathbf{x}_0$, find the K training points closest to $\mathbf{x}_0$ (usually in Euclidean distance).
3. Predict the class that has the plurality among these K neighbors.

By avoiding assumptions on distributions or linearity, KNN can adapt to highly non-linear decision boundaries. However, it can suffer when p is large (the *curse of dimensionality*) or when n is small relative to p. Choosing K typically involves a bias-variance trade-off: small K yields flexible boundaries (low bias but high variance), while large K yields smoother boundaries (higher bias but lower variance).

7 Comparison of Classification Methods

7.1 Statistical Properties & Decision Boundaries

The chapter dedicates a thorough comparison to LDA, QDA, naive Bayes, logistic regression, and KNN. In short:

- *Logistic regression* and *LDA* share linear log-odds, but logistic regression arises from a direct modeling of $P(Y = k \mid X)$, while LDA is a generative model that also assumes Gaussian distributions with a shared covariance.

- *QDA* extends LDA by allowing separate covariance matrices, enabling more flexible, *quadratic* decision boundaries.

- *Naive Bayes* relies on an independence assumption for the features within each class but can drastically reduce complexity. It can be viewed as a special case of more general classifiers under certain conditions.

- *KNN* makes minimal distributional assumptions, can capture highly non-linear boundaries, but may struggle with large p or insufficient data.

7.2 Performance Metrics

Central to evaluating classifiers is the confusion matrix:

	Predicted Yes	Predicted No
Actual Yes	TP	FN
Actual No	FP	TN

From this table, measures such as:

- **Accuracy**: $(\text{TP} + \text{TN})/(\text{Total})$,

- **Sensitivity** (recall): $\text{TP}/(\text{TP} + \text{FN})$,

- **Specificity**: $\text{TN}/(\text{TN} + \text{FP})$,

- **Positive Predictive Value** (precision): $\text{TP}/(\text{TP} + \text{FP})$,

are used to compare how well each model identifies classes.

7.3 ROC Curves and AUC

Another important tool is the **ROC curve**, which plots *true positive rate* (sensitivity) versus *false positive rate* ($1 -$specificity) for different thresholds. The **Area Under the ROC Curve (AUC)** measures overall performance across all possible thresholds. An ideal classifier has an AUC near 1, while a random-guess classifier has AUC near 0.5.

7.4 Empirical Comparisons

In numerical experiments, the chapter demonstrates that no single method is universally the best:

- When class boundaries are truly linear, *LDA* or *logistic regression* often excel.
- When boundaries are moderately non-linear, *QDA* or *naive Bayes* might be preferable.
- For very complex boundaries, *KNN* can be superior if sufficient training data are available.

Ultimately, model choice depends on the actual data-generating process, the sample size, the dimensionality of predictors, and considerations of interpretability.

8 Generalized Linear Models (GLMs)

8.1 Beyond Binary Outcomes

The text then broadens to the concept of **generalized linear models**, in which linear regression (for continuous outcomes) and logistic regression (for binary outcomes) are special cases. A GLM is characterized by:

1. A *distribution* in the exponential family (e.g. normal, Bernoulli, Poisson).
2. A *link function* $\eta(\mu)$ relating the mean of the response Y to a linear combination of predictors:

$$\eta(\mathbb{E}[Y]) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p.$$

8.2 Poisson Regression

An important example is **Poisson regression**, used for *count data* where Y takes integer values in $\{0, 1, 2, \dots\}$. The Poisson distribution for Y with mean λ is:

$$P(Y = k) = \frac{\lambda^k e^{-\lambda}}{k!},$$

and Poisson regression sets

$$\log(\lambda(\mathbf{x})) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

This ensures the predicted mean $\lambda(\mathbf{x})$ is always positive, consistent with counts. For instance, the chapter uses `Bikeshare` data to model the number of hourly bike rentals. A linear model might give negative predictions or fail to reflect the mean-variance relationship in count data. By contrast, Poisson regression properly handles that the variance grows with the mean.

9 Lab: Classification Methods

In the final part of the chapter, a series of labs in `R` illustrate how to implement each classification technique:

- **Logistic Regression** on stock market (`Smarket`) data to predict up/down movements based on historical returns.

- **LDA** and **QDA** on the same data, highlighting that these two methods can yield different decision boundaries and performance.

- **Naive Bayes** using the `naiveBayes` function to demonstrate how the conditional independence assumption can simplify computations.

- **KNN** on both `Smarket` and `Caravan` data sets, showing how choosing K and standardizing features can significantly impact results.

- **Poisson Regression** on the `Bikeshare` data, illustrating how to handle count data effectively.

These hands-on examples reinforce the comparative strengths and weaknesses of each approach and underscore the necessity of careful data splitting (training vs. testing), performance measurement, and threshold tuning.

10 Conclusion

Chapter 4 provides a thorough treatment of classification, covering theory, practical considerations, and real-world examples. Central lessons include:

1. **Match the model to the data generation process when possible.** For linear boundaries and moderate data size, LDA/logistic can excel. If covariances differ by class, QDA is a candidate. If independence approximations hold or p is large, naive Bayes can be strong. If n is large or boundaries are highly non-linear, KNN is an option.

2. **Always assess performance on held-out (test) data.** Training error is usually overly optimistic.

3. **Consider interpretability.** Logistic regression yields probability estimates and interpretable coefficients. KNN is highly flexible but does not yield explicit coefficients or easily interpretable decision rules.

4. **Adjust thresholds to suit domain needs.** Costs of false positives and false negatives are often not symmetric.

5. **Generalized Linear Models are a unifying framework.** Logistic and Poisson regression are both examples within this broader class, revealing how a suitable link function and exponential family distribution can systematically address many outcome types.

Classification remains a cornerstone of supervised learning, and these techniques lay a foundation for more advanced methods like trees, random forests, boosting, and support vector machines (covered in later chapters). The chapter's empirical evidence and practical labs highlight that understanding your data, testing multiple methods, tuning hyperparameters (such as K in KNN), and carefully selecting the correct model assumptions are all crucial to success in classification tasks.

Summary of Chapter 5: Resampling Methods

Introduction

Chapter 5 focuses on *resampling methods*, a family of techniques that repeatedly draw samples or subsets from an available dataset and re-fit a model to each of these newly formed samples. The overarching goal of these methods is to gain additional insight about the variability or accuracy of a statistical learning procedure. Two of the most widely used resampling methods, **cross-validation** and **the bootstrap**, are emphasized:

- **Cross-validation** is frequently employed to estimate test error (and thus assess model performance) or to determine an appropriate level of model complexity.
- **The bootstrap** is used to quantify the uncertainty (e.g., the standard error) of an estimate and to provide interval estimates such as confidence intervals.

The chapter underscores the fact that, although computationally more intensive than fitting a model only once, resampling methods have become quite practical with modern computing power.

1 Cross-Validation

Cross-validation (CV) is introduced as a tool for obtaining an estimate of test error or predictive accuracy without requiring an explicitly held-out test set. It addresses a key problem in applied statistical learning: the discrepancy between *training error* (the error obtained by fitting a model to the training set) and *test error* (the error on new, unseen data).

1.1 Motivation and Definitions

- When no large external test set is available, cross-validation creates multiple *pseudo*-training and *pseudo*-validation subsets from the original dataset.
- The idea is to systematically hold out or "*validate on*" portions of the dataset in turn, fit the model on the remaining part, and then compute an estimate of how well the model predicts that held-out portion.

1.2 Validation Set Approach

- This is the simplest form of CV, sometimes called the *hold-out method.*
- The dataset is randomly split into two sets: a **training set** and a **validation set**.
- The model is fitted on the training set and then evaluated on the validation set to estimate the test error.
- Drawbacks:
  1. The estimate of test error can be highly variable, because results depend on how exactly the data are split.
  2. Only a fraction of the dataset is used to fit the model (the other portion being set aside), potentially causing higher bias in the fitted model.

1.3 Leave-One-Out Cross-Validation (LOOCV)

- LOOCV is an extreme version of the validation set approach:

  1. We repeatedly fit the model n times, each time leaving out exactly one observation from the training set of size $n - 1$.
  2. The omitted single observation acts as the validation set, and the prediction error is computed for that lone observation.
  3. We then average these n validation errors to get the LOOCV estimate of the test error.

- Advantages:

  – Much less bias than a single validation set split, because almost all data is used for training each time.

  – No randomness in how folds or splits are made (the single left-out observation shifts deterministically from one iteration to the next).

- Disadvantage:

  – Potentially higher variance of the error estimate, because the fits are trained on extremely similar subsets of size $n - 1$ and hence the individual predictive fits can be highly correlated.

  – Computational cost can be high if each model is expensive to fit, because n separate fits are required (though for linear models a convenient formula reduces the cost).

1.4 k-Fold Cross-Validation

- This generalizes the idea of LOOCV, splitting the data into k roughly equal-sized parts (*folds*) rather than leaving out exactly one observation per iteration.

- For each fold:

  1. One fold is designated as the validation set.
  2. The model is fitted on the other $k - 1$ folds combined.
  3. The resulting error on the held-out fold is recorded.

- The k errors are averaged to produce the overall k-fold CV error.

- A common choice is $k = 5$ or $k = 10$.

- Bias-variance perspective:

  – LOOCV ($k = n$) can have low bias but comparatively high variance in the estimate.

  – k-fold CV with moderate k reduces correlation among the training sets and hence can reduce variance, at the cost of a small increase in bias, relative to LOOCV.

1.5 Cross-Validation for Classification

- The same concepts can be applied when the response Y is qualitative, except that we measure misclassification error instead of mean squared error.
- LOOCV or k-fold CV is equally valid for classification error estimates.

2 The Bootstrap

Whereas cross-validation is mostly concerned with estimating (and tuning) the test error of a statistical learning method, the **bootstrap** is a flexible tool that can be used to assess the variability or uncertainty of an estimator.

2.1 Bootstrap Basics

- The bootstrap, in essence, simulates sampling from the original dataset *with replacement*.
- In practice:
  1. We treat our observed dataset of n points as a stand-in for the population.
  2. We sample n points *with replacement* from our original dataset to create a *bootstrap dataset*.
  3. We compute the quantity of interest (e.g., a regression coefficient, a correlation, etc.) on this bootstrap dataset.
  4. We repeat B times, collecting B estimates, from which we can compute a standard deviation (or standard error) or form confidence intervals.

2.2 Illustrations

Estimating the Standard Error of an Estimate. A running example in the chapter uses a toy problem of estimating α in a two-asset portfolio. The bootstrap is used to evaluate how much the estimate of α might vary if we slightly changed the dataset.

Bootstrap for Regression. Similarly, we can apply the bootstrap to linear regression to assess the variability of the estimated coefficients, especially when no formula-based standard errors are readily available (or when the standard formula assumptions may be violated).

3 Lab: Cross-Validation and the Bootstrap

The chapter concludes with practical labs (in `R`) demonstrating the usage of:

- `cv.glm()` from the `boot` library to compute k-fold and LOOCV estimates for generalized linear models.
- Validation set approaches manually coded or with specialized functions.
- The `boot()` function for implementing the bootstrap, illustrated through examples such as estimating the standard error of regression coefficients.

These labs highlight how to implement and compare the various resampling approaches discussed in the chapter. They use datasets such as the `Auto` data (for MPG regression) and the `Portfolio` data (for the two-asset α example). The code emphasizes the mechanics of splitting data, computing cross-validation errors, and bootstrapping estimates in practical settings.

Summary of Key Points

- **Resampling** is crucial in modern statistics for accurate estimation of model performance and parameter uncertainty.
- **Cross-validation**:
  - Provides a data-driven approach to estimate test error and select model complexity (e.g., for linear regression or classification).
  - Variants such as LOOCV and k-fold CV strike different balances between bias and variance in the error estimates.
- **Bootstrap**:
  - Provides a widely applicable way to quantify uncertainty (e.g., compute standard errors, confidence intervals) for virtually any estimator or predictive model.
  - Achieves this by sampling with replacement from the observed data, thus emulating what would happen if new datasets were repeatedly drawn from the population.
- By using these techniques, practitioners can better understand how a learning procedure might perform on unseen data and how reliable their estimates of model parameters truly are.

Summary of Chapter 6: Linear Model Selection and Regularization

1 Overview

Chapter 6 discusses methods for extending and improving upon the classical linear regression framework. In particular, it focuses on approaches that address two main issues:

1. **Prediction Accuracy:** When the number of predictors p is not small compared to the number of observations n, or when $p > n$, least squares estimates can suffer from high variance and can overfit the data. Methods that reduce variance (at the cost of a small increase in bias) often improve the predictive performance.
2. **Model Interpretability:** Traditional least squares does not naturally produce sparse models (i.e., models where some coefficients are exactly zero). Consequently, identifying the truly relevant predictors among a large set can be difficult. Certain regularization methods automatically set some coefficient estimates to zero, aiding interpretability.

The chapter covers three major classes of approaches:

- *Subset Selection:* Identifying a subset of predictors that best explain the response.
- *Shrinkage (Regularization):* Estimating coefficients by shrinking them toward zero; important examples are *ridge regression* and the *lasso*.
- *Dimension Reduction:* Transforming the original predictors into a smaller set of derived features (e.g., *principal components regression* and *partial least squares*).

This summary reviews these methods and addresses some unique considerations that arise in high-dimensional settings.

2 Subset Selection

In subset selection methods, the goal is to find a subset of predictors (from among the p available) that best models the response. The central challenge is deciding which subsets of predictors to include. The chapter describes:

2.1 Best Subset Selection

- **Idea:** Fit separate least squares regressions for all possible subsets of the p predictors. Then choose the "best" subset of each size $k = 0, 1, \ldots, p$.
- **Algorithm:**
  1. For $k = 0$ to p, consider all $\binom{p}{k}$ predictor subsets of size k.
  2. Fit a least squares model for each subset.
  3. Choose the best model (lowest RSS or highest R^2) among the models of size k.
  4. Select a single model from among these $p + 1$ candidates using criteria such as cross-validation, C_p, BIC, adjusted R^2, or validation-set error.

- **Issues:** The number of models grows exponentially (2^p). This becomes infeasible for even moderate p. The method also risks overfitting if p is large relative to n.

2.2 Stepwise Selection Methods

Because best subset selection quickly becomes computationally infeasible, simpler *stepwise* approaches are often used.

Forward Stepwise Selection

- Starts with the null model (no predictors).
- Adds predictors one at a time, at each step choosing the predictor that improves the model fit the most (e.g., yields the smallest RSS).
- Continues until all predictors are in the model (or until some stopping criterion).
- Far less computationally expensive than best subset selection.

Backward Stepwise Selection

- Starts with the full model (all p predictors).
- Iteratively removes the least useful predictor (e.g., the one whose removal increases the RSS the least).
- Requires $n > p$ so that the full model can be fit initially.

Hybrid Approaches

- Methods that add variables like forward selection but allow for occasional backward removal at each step.

2.3 Choosing the Optimal Model Size

Once all candidate models have been fit, we must select a single best model among the $p + 1$ (or fewer, in stepwise) possibilities. This can be done via:

- **Cross-Validation or Validation-Set Error:** Train each model on one portion of the data and assess its prediction error on the held-out portion.
- **C_p, AIC, BIC, Adjusted R^2:** Each is an estimate of test error (or test error-related quantity) that penalizes model complexity differently.

Often, one selects the model that yields the minimum cross-validation error or the smallest C_p, BIC, etc.

3 Shrinkage (Regularization) Methods

Rather than selecting a subset of predictors, *shrinkage* methods involve fitting a model containing all predictors but shrinking the coefficient estimates toward zero. The two most important approaches are **ridge regression** and the **lasso**.

3.1 Ridge Regression

- **Definition:** Estimates coefficients β_j by minimizing

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

- The second term $\lambda \sum \beta_j^2$ is a *shrinkage penalty* that penalizes large coefficient estimates.
- **Effect:** As λ grows, the coefficient estimates get shrunk toward zero, reducing variance but possibly increasing bias. This can lower test error if the bias-variance trade-off is improved.
- **No Variable Selection:** Ridge regression never sets coefficients exactly to zero (unless λ is infinite). Thus, it does not yield a sparse model.

3.2 The Lasso

- **Definition:** Estimates coefficients by minimizing

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

- The penalty is now an ℓ_1 norm ($\sum |\beta_j|$), rather than the ℓ_2 norm used by ridge regression.
- **Variable Selection:** The ℓ_1 penalty forces some coefficients to be *exactly* zero when λ is sufficiently large, producing a sparse model. This yields better interpretability (fewer predictors).

3.3 Selecting the Tuning Parameter

Both ridge regression and the lasso require choosing λ. Typically one uses cross-validation:

1. Fit the model for a grid of λ values.
2. For each λ, compute the cross-validation error.
3. Select the λ that yields the smallest cross-validation error.
4. Refit on the full data using this chosen λ.

3.4 Bayesian Interpretation

- Ridge regression corresponds to assuming a Gaussian prior for the coefficients (β_j), centered at zero.
- Lasso corresponds to a double-exponential (Laplace) prior centered at zero.
- In each case, the solution to the penalized least squares problem can be interpreted as the mode of the posterior distribution under those priors.

4 Dimension Reduction Methods

These techniques reduce the dimension of the problem by projecting the p-dimensional predictors onto a smaller M-dimensional subspace, and then regressing the response on these M derived predictors.

4.1 Principal Components Regression (PCR)

- **Principal Components Analysis (PCA):** Identify new orthogonal directions (principal components) that capture maximal variance in the predictor space.
- **PCR:** Select the first M principal components $Z_1, \ldots, Z_M$ of the predictors, and then fit a linear model to predict Y from these components.
- **Dimension Reduction:** Only $M < p$ directions are used, which can mitigate overfitting by dramatically reducing the number of fitted parameters.
- **Drawback:** The principal components are chosen *unsupervised*, i.e., without considering Y. Thus, it is not guaranteed that the components capturing largest variance in predictors are also most relevant for the response.

4.2 Partial Least Squares (PLS)

- **Similar to PCR**, but it is a *supervised* approach. The directions onto which we project the data are chosen to maximize variation in the predictors *and* correlation with the response.
- PLS can sometimes perform better than PCR because it uses Y in constructing the directions.
- Both PCR and PLS require selecting the number of components M, typically via cross-validation.

5 Considerations in High Dimensions

5.1 High-Dimensional Data

In modern applications, p (the number of predictors) can be much larger than n (the number of observations). This is called the *high-dimensional* setting. Traditional methods such as ordinary least squares can fail badly here, because:

- Perfect fits (zero residuals) on the training set are easy to achieve but do not generalize (huge test error).
- Multicollinearity is extreme: any predictor can be expressed approximately (or exactly) as a linear combination of other predictors.

5.2 Pitfalls in High Dimensions

- **Overfitting** becomes a severe threat as p grows.
- Traditional metrics like training set R^2 or residual sum of squares (RSS) can be *misleading*; they may be arbitrarily small due to overfitting.
- **Cross-validation or independent test sets** are essential to obtain honest estimates of predictive performance.
- Model interpretability is more challenging: if many different subsets of predictors yield similar fits, it may be hard to pinpoint a single "true" subset.

5.3 Methods for High Dimensions

- The methods covered in this chapter (subset selection, ridge, lasso, PCR, PLS) are often especially helpful in high-dimensional contexts.
- **Regularization** (ridge or lasso) often drastically reduces variance and can yield better prediction.
- **Sparse solutions** (from the lasso) can improve interpretability by identifying the most relevant predictors.

6 Lab: Linear Models and Regularization Methods

In the lab section of Chapter 6, various **R** techniques for implementing these methods are illustrated:

- **Subset Selection:** Using `regsubsets` (from the `leaps` package) for best subset, forward stepwise, and backward stepwise selection.
- **Shrinkage:** Using `glmnet` to fit ridge regression and the lasso over a grid of λ values and applying cross-validation with `cv.glmnet`.
- **PCR & PLS:** Using `pcr` and `plsr` (from the `pls` package) with cross-validation to choose the number of components M.
- **Comparison of Test Errors:** Demonstrating how to split data into training and testing subsets, compute test MSE, and compare different methods.

The lab emphasizes practical details in **R** for model fitting, hyperparameter selection (e.g., λ in ridge/lasso, or M in PCR/PLS), and error assessment via cross-validation.

7 Summary of Key Points

1. **Subset Selection Methods** reduce the model size by including only a subset of predictors. These can be computationally expensive or infeasible for large p.

2. **Ridge Regression** shrinks coefficients by adding an ℓ_2 penalty; it does not set them to zero, but typically lowers variance.

3. **The Lasso** uses an ℓ_1 penalty, performing automatic variable selection by setting some coefficients exactly to zero.

4. **PCR and PLS** reduce dimension by working with linear combinations of predictors. PCR is unsupervised (focuses on predictor variance), whereas PLS is supervised (also looks at the response).

5. In **high-dimensional settings**, careful use of regularization or dimension reduction is essential to avoid severe overfitting and to manage interpretability.

6. **Cross-validation** is crucial for tuning parameter selection and reliable estimation of test error.

In [1]:

```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
from statsmodels.api import OLS
import sklearn.model_selection as skm
import sklearn.linear_model as skl
from sklearn.preprocessing import StandardScaler
from ISLP import load_data
from ISLP.models import ModelSpec as MS
from functools import partial
```

In [2]:

```
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression
from ISLP.models import \
    (Stepwise,
     sklearn_selected,
     sklearn_selection_path)
from l0bnb import fit_path
```

In [3]:

```
from sklearn.exceptions import ConvergenceWarning
import warnings
warnings.filterwarnings("ignore", category=ConvergenceWarning)
warnings.filterwarnings("ignore", category=UserWarning)
```

In [4]:

```
Hitters = load_data('Hitters')
np.isnan(Hitters['Salary']).sum()
```

Out[4]:

```
59
```

In [5]:

```
Hitters = Hitters.dropna()
Hitters.shape
```

Out[5]:

```
(263, 20)
```

In [6]:

```
def nCp(sigma2, estimator, X, Y):
    "Negative Cp statistic"
    n, p = X.shape
    Yhat = estimator.predict(X)
    RSS = np.sum((Y - Yhat)**2)
    return -(RSS + 2 * p * sigma2) / n
```

In [7]:

```
design = MS(Hitters.columns.drop('Salary')).fit(Hitters)
Y = np.array(Hitters['Salary'])
X = design.transform(Hitters)
sigma2 = OLS(Y, X).fit().scale
```

In [8]:

```
neg_Cp = partial(nCp, sigma2)
```

In [9]:

```
strategy = Stepwise.first_peak(design,
                               direction='forward',
                               max_terms=len(design.terms))
```

In [10]:

```
hitters_MSE = sklearn_selected(OLS,
                               strategy)
hitters_MSE.fit(Hitters, Y)
hitters_MSE.selected_state_
```

Out[10]:

```
('Assists',
 'AtBat',
 'CAtBat',
 'CHits',
 'CHmRun',
 'CRBI',
 'CRuns',
 'CWalks',
 'Division',
 'Errors',
 'Hits',
 'HmRun',
 'League',
 'NewLeague',
 'PutOuts',
 'RBI',
 'Runs',
 'Walks',
 'Years')
```

In [11]:

```
hitters_Cp = sklearn_selected(OLS,
                              strategy,
                              scoring=neg_Cp)
hitters_Cp.fit(Hitters, Y)
hitters_Cp.selected_state_
```

Out[11]:

```
('Assists',
 'AtBat',
 'CAtBat',
 'CRBI',
 'CRuns',
 'CWalks',
 'Division',
 'Hits',
 'PutOuts',
 'Walks')
```

In [12]:

```
strategy = Stepwise.fixed_steps(design,
                                len(design.terms),
                                direction='forward')
full_path = sklearn_selection_path(OLS, strategy)
```

In [13]:

```
full_path.fit(Hitters, Y)
Yhat_in = full_path.predict(Hitters)
```

```
Yhat_in.shape
```

Out[13]:

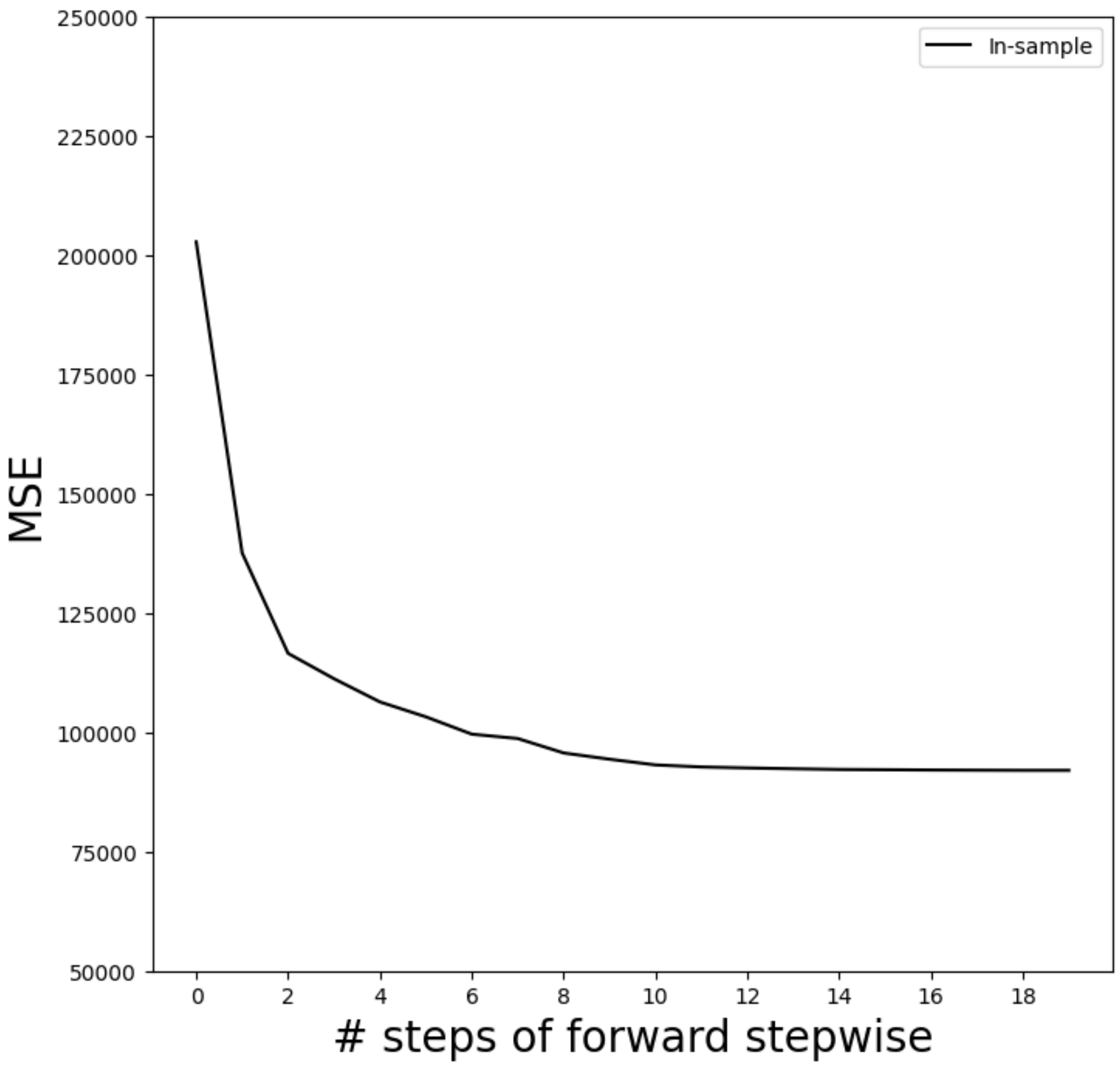
```
(263, 20)
```

In [14]:

```
mse_fig, ax = subplots(figsize=(8, 8))
insample_mse = ((Yhat_in - Y[:, None]) ** 2).mean(0)
n_steps = insample_mse.shape[0]
ax.plot(np.arange(n_steps),
        insample_mse,
        'k',  # color black
        label='In-sample')
ax.set_ylabel('MSE',
              fontsize=20)
ax.set_xlabel('# steps of forward stepwise',
              fontsize=20)
ax.set_xticks(np.arange(n_steps)[::2])
ax.legend()
ax.set_ylim([50000, 250000])
```

Out[14]:

```
(50000.0, 250000.0)
```



```
In [15]: K = 5
         kfold = skm.KFold(K,
                           random_state=0,
                           shuffle=True)
         Yhat_cv = skm.cross_val_predict(full_path,
                                         Hitters,
                                         Y,
                                         cv=kfold)
         Yhat_cv.shape
```

```
Out[15]: (263, 20)
```

```
In [16]: cv_mse = []
         for train_idx, test_idx in kfold.split(Y):
             errors = (Yhat_cv[test_idx] - Y[test_idx, None]) ** 2
             cv_mse.append(errors.mean(0))  # column means
         cv_mse = np.array(cv_mse).T
         cv_mse.shape
```

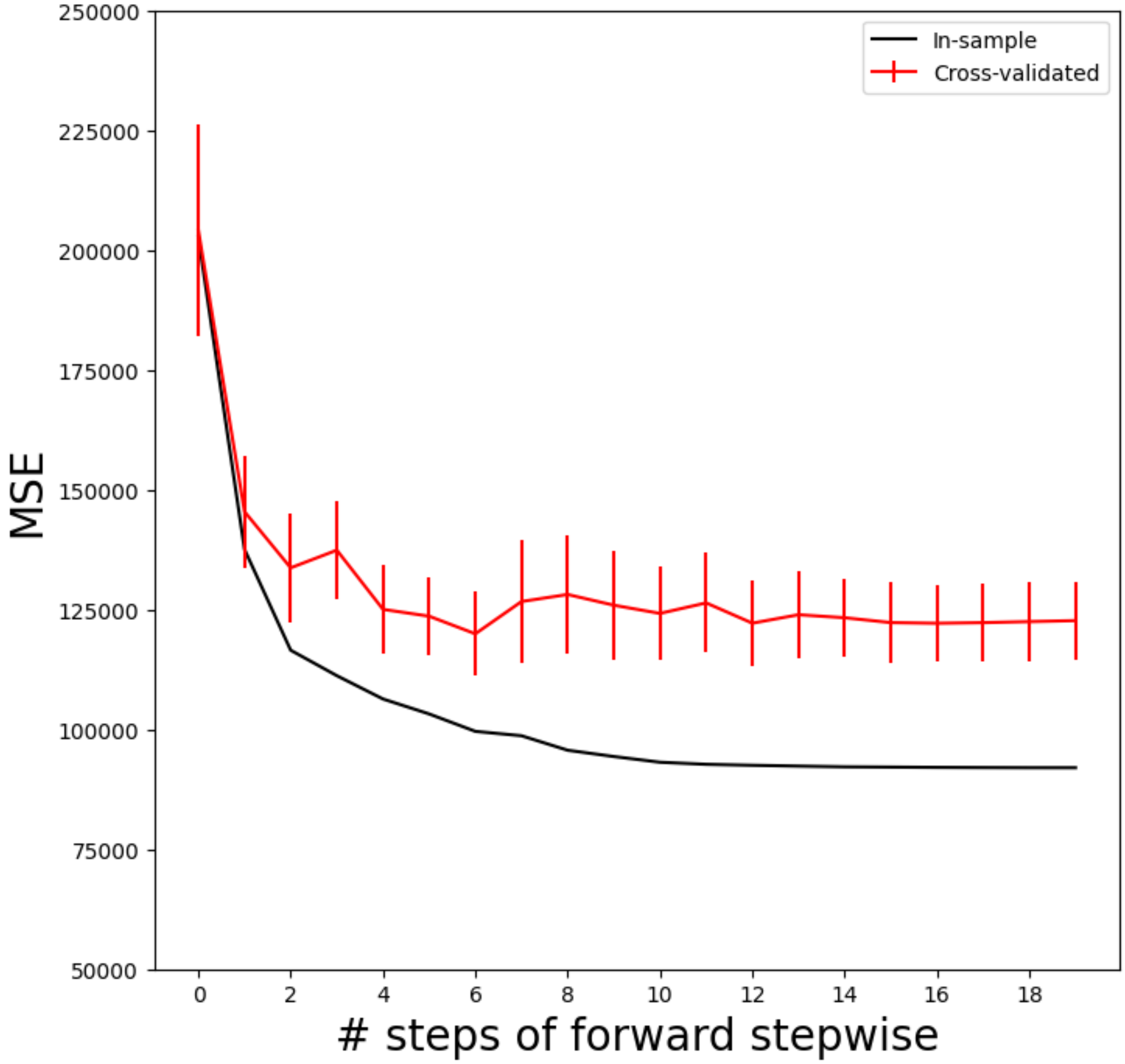
Out[16]:

```
(20, 5)
```

In [17]:

```
ax.errorbar(np.arange(n_steps),
            cv_mse.mean(1),
            cv_mse.std(1) / np.sqrt(K),
            label='Cross-validated',
            c='r')  # color red
ax.set_ylim([50000, 250000])
ax.legend()
mse_fig
```

Out[17]:



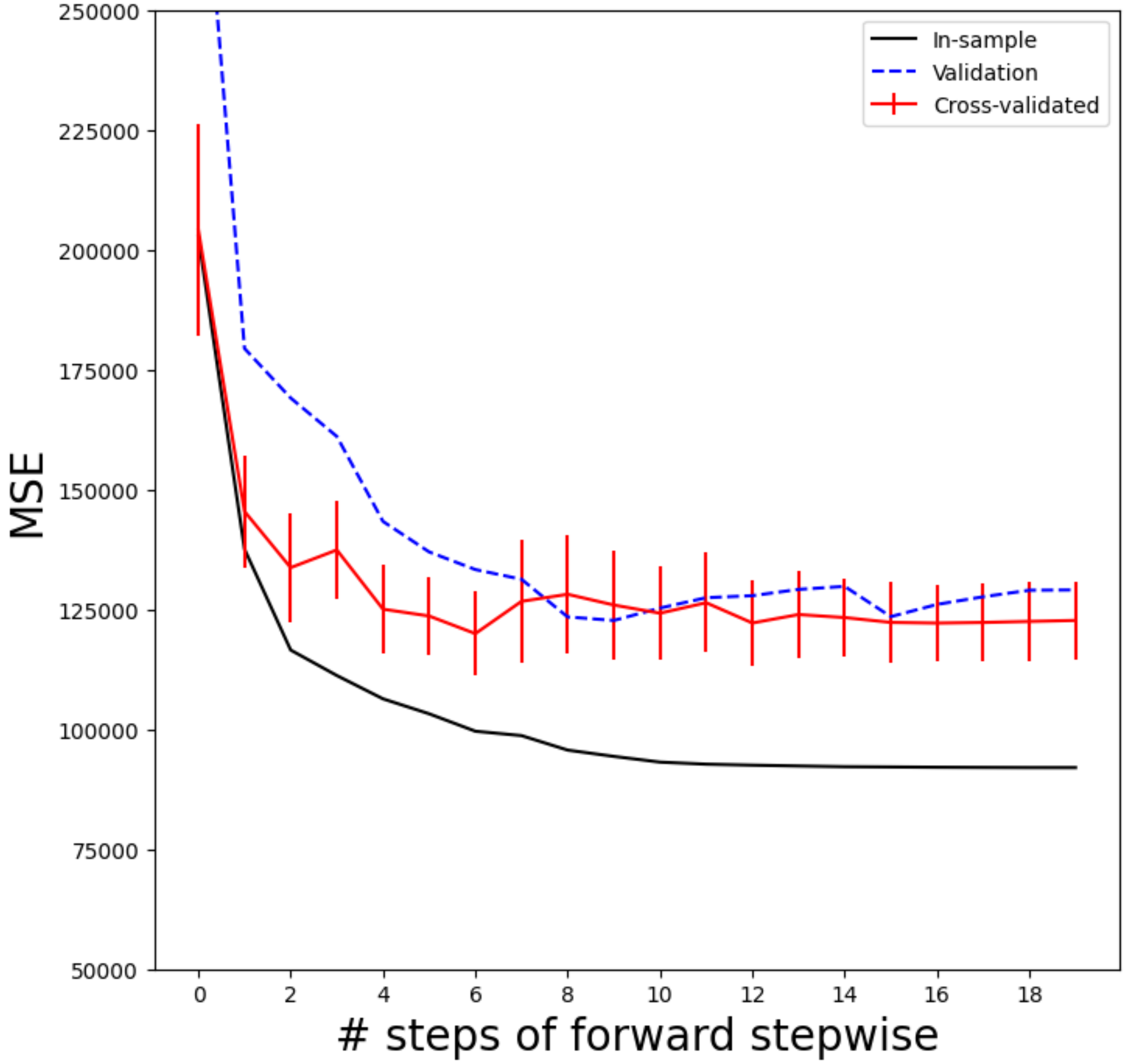
In [18]:

```
validation = skm.ShuffleSplit(n_splits=1,
                              test_size=0.2,
                              random_state=0)
for train_idx, test_idx in validation.split(Y):
    full_path.fit(Hitters.iloc[train_idx],
                  Y[train_idx])
    Yhat_val = full_path.predict(Hitters.iloc[test_idx])
    errors = (Yhat_val - Y[test_idx, None]) ** 2
    validation_mse = errors.mean(0)
```

In [19]:

```
ax.plot(np.arange(n_steps),
        validation_mse,
        'b--',  # color blue , broken line
        label='Validation')
ax.set_xticks(np.arange(n_steps)[::2])
ax.set_ylim([50000, 250000])
ax.legend()
mse_fig
```

Out[19]:



In [20]:

```
D = design.fit_transform(Hitters)
D = D.drop('intercept', axis=1)
X = np.asarray(D)
```

In [21]:

```
path = fit_path(X,
                Y,
                max_nonzeros=X.shape[1])
```

```
Preprocessing Data.
BnB Started.
Iteration: 1. Number of non-zeros:  1
Iteration: 2. Number of non-zeros:  2
Iteration: 3. Number of non-zeros:  2
Iteration: 4. Number of non-zeros:  2
Iteration: 5. Number of non-zeros:  3
Iteration: 6. Number of non-zeros:  3
Iteration: 7. Number of non-zeros:  4
Iteration: 8. Number of non-zeros:  9
Iteration: 9. Number of non-zeros:  9
Iteration: 10. Number of non-zeros:  9
Iteration: 11. Number of non-zeros:  9
Iteration: 12. Number of non-zeros:  9
Iteration: 13. Number of non-zeros:  9
Iteration: 14. Number of non-zeros:  9
Iteration: 15. Number of non-zeros:  9
Iteration: 16. Number of non-zeros:  9
Iteration: 17. Number of non-zeros:  9
Iteration: 18. Number of non-zeros:  17
Iteration: 19. Number of non-zeros:  19
```

In [22]:

```
path[3]
```

Out[22]:

```
{'B': array([0.        , 3.25484367, 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.67775265, 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        ]),
 'B0': -38.98216739555551,
 'lambda_0': 0.011416248027450178,
 'M': 0.5829861733382012,
 'Time_exceeded': False}
```

In [23]:

```
Xs = X - X.mean(0)[None, :]
X_scale = X.std(0)
Xs = Xs / X_scale[None, :]
lambdas = 10 ** np.linspace(8, -2, 100) / Y.std()
soln_array = skl.ElasticNet.path(Xs,
                                 Y,
                                 l1_ratio=0.,
                                 alphas=lambdas)[1]
soln_array.shape
```

Out[23]:

```
(19, 100)
```

In [24]:

```
soln_path = pd.DataFrame(soln_array.T,
                         columns=D.columns,
                         index=-np.log(lambdas))
soln_path.index.name = 'negative log(lambda)'
soln_path
```

Out[24]:

	AtBat	Hits	HmRun	Runs	RBI	Walk
negative log(lambda)						
-12.310855	0.000800	0.000889	0.000695	0.000851	0.000911	0.00090
-12.078271	0.001010	0.001122	0.000878	0.001074	0.001150	0.00113
-11.845686	0.001274	0.001416	0.001107	0.001355	0.001451	0.00143
-11.613102	0.001608	0.001787	0.001397	0.001710	0.001831	0.00180
-11.380518	0.002029	0.002255	0.001763	0.002158	0.002310	0.0022
...	...	...	...	...	...	.
9.784658	-290.823989	336.929968	37.322686	-59.748520	-26.507086	134.85591
10.017243	-290.879272	337.113713	37.431373	-59.916820	-26.606957	134.90054
10.249827	-290.923382	337.260446	37.518064	-60.051166	-26.686604	134.93613
10.482412	-290.958537	337.377455	37.587122	-60.158256	-26.750044	134.96447
10.714996	-290.986528	337.470648	37.642077	-60.243522	-26.800522	134.98702

100 rows × 19 columns

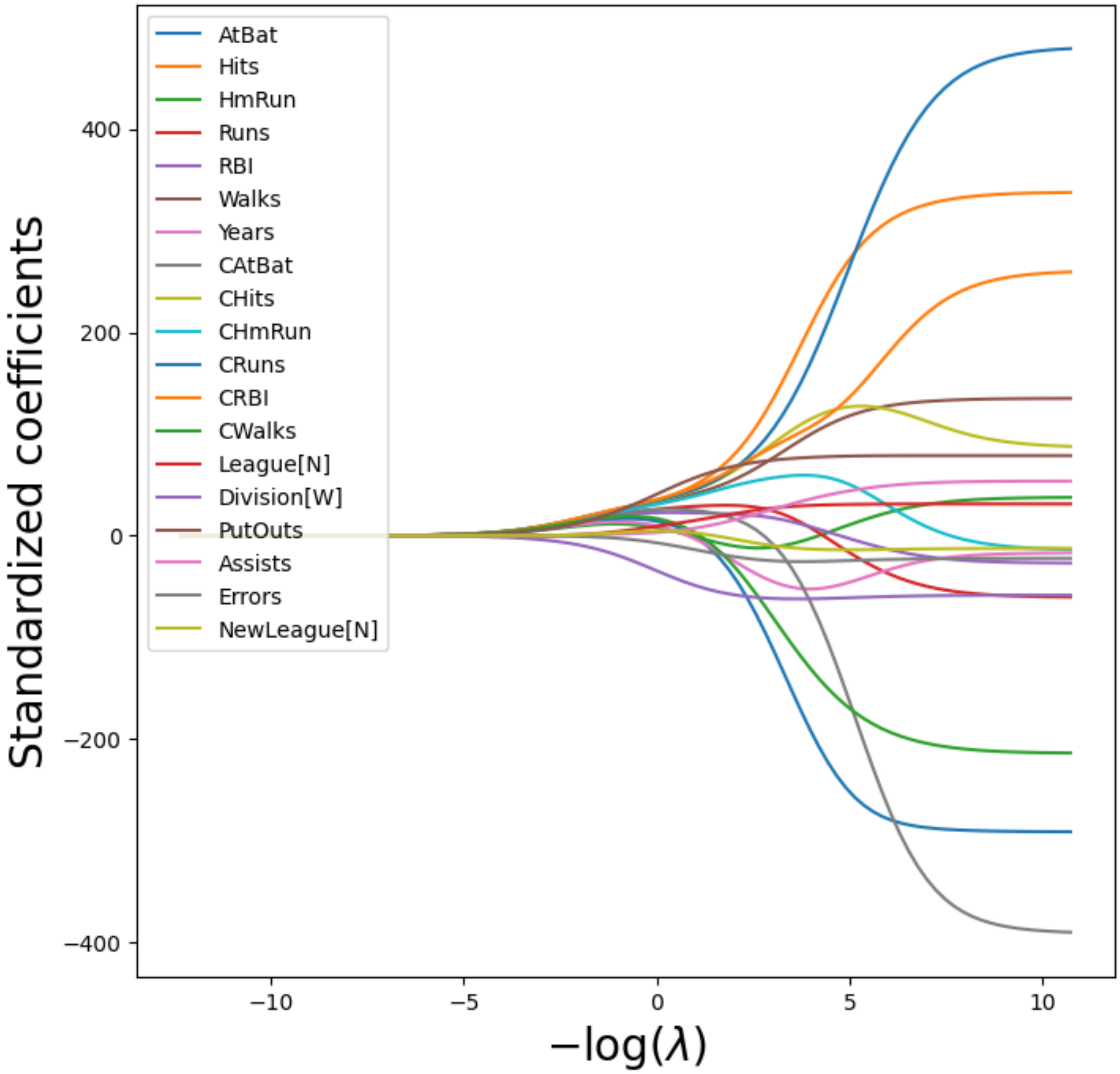
In [25]:

```python
path_fig, ax = subplots(figsize=(8, 8))
soln_path.plot(ax=ax, legend=False)
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Standardized coefficients', fontsize=20)
ax.legend(loc='upper left')
```

```
<>:3: SyntaxWarning: invalid escape sequence '\l'
<>:3: SyntaxWarning: invalid escape sequence '\l'
/var/folders/97/23ltc4v96g31pp78_gyv6dvm0000gn/T/ipykernel_10519/2938439178.
py:3: SyntaxWarning: invalid escape sequence '\l'
  ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
```

Out[25]:

```
<matplotlib.legend.Legend at 0x304386690>
```



In [26]:

```
beta_hat = soln_path.loc[soln_path.index[39]]
lambdas[39], beta_hat
```

Out[26]:

```
(25.53538897200662,
 AtBat           5.433750
 Hits            6.223582
 HmRun           4.585498
 Runs            5.880855
 RBI             6.195921
 Walks           6.277975
 Years           5.299767
 CAtBat          7.147501
 CHits           7.539495
 CHmRun          7.182344
 CRuns           7.728649
 CRBI            7.790702
 CWalks          6.592901
 League[N]       0.042445
 Division[W]    -3.107159
 PutOuts         4.605263
 Assists         0.378371
 Errors         -0.135196
 NewLeague[N]    0.150323
 Name: -3.240065292879872, dtype: float64)
```

In [27]:

```python
np.linalg.norm(beta_hat)
```

Out[27]:

```
24.17061720144378
```

In [28]:

```python
beta_hat = soln_path.loc[soln_path.index[59]]
lambdas[59], np.linalg.norm(beta_hat)
```

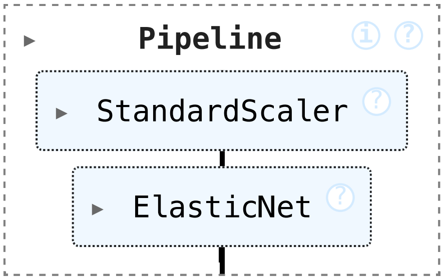
Out[28]:

```
(0.24374766133488554, 160.42371017725839)
```

In [29]:

```python
ridge = skl.ElasticNet(alpha=lambdas[59], l1_ratio=0)
scaler = StandardScaler(with_mean=True, with_std=True)
pipe = Pipeline(steps=[('scaler', scaler), ('ridge', ridge)])
pipe.fit(X, Y)
```

Out[29]:



In [30]:

```python
np.linalg.norm(ridge.coef_)
```

Out[30]:

```
160.4237101772591
```

In [31]:

```python
validation = skm.ShuffleSplit(n_splits=1,
                              test_size=0.5,
                              random_state=0)
```

```
ridge.alpha = 0.01
results = skm.cross_validate(ridge,
                             X,
                             Y,
                             scoring='neg_mean_squared_error',
                             cv=validation)
-results['test_score']
```

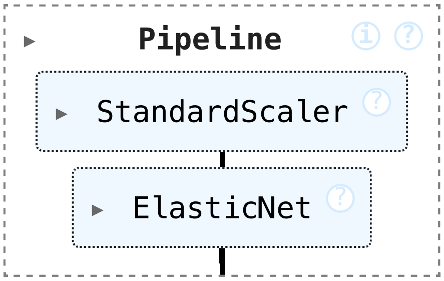
```
Out[31]:  array([134214.00419204])
```

```
In [32]: ridge.alpha = 1e10
         results = skm.cross_validate(ridge,
                                      X,
                                      Y,
                                      scoring='neg_mean_squared_error',
                                      cv=validation)
         -results['test_score']
```

```
Out[32]:  array([231788.32155285])
```

```
In [33]: param_grid = {'ridge__alpha': lambdas}
         grid = skm.GridSearchCV(pipe,
                                 param_grid,
                                 cv=validation,
                                 scoring='neg_mean_squared_error')
         grid.fit(X, Y)
         grid.best_params_['ridge__alpha']
         grid.best_estimator_
```

Out[33]:



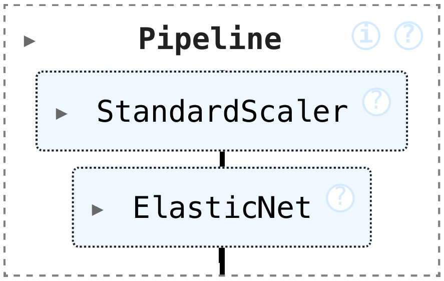
▸ Pipeline

▸ StandardScaler

▸ ElasticNet

```
In [34]: grid = skm.GridSearchCV(pipe,
                                 param_grid,
                                 cv=kfold,
                                 scoring='neg_mean_squared_error')
         grid.fit(X, Y)
         grid.best_params_['ridge__alpha']
         grid.best_estimator_
```

Out[34]:



▸ Pipeline

▸ StandardScaler

▸ ElasticNet

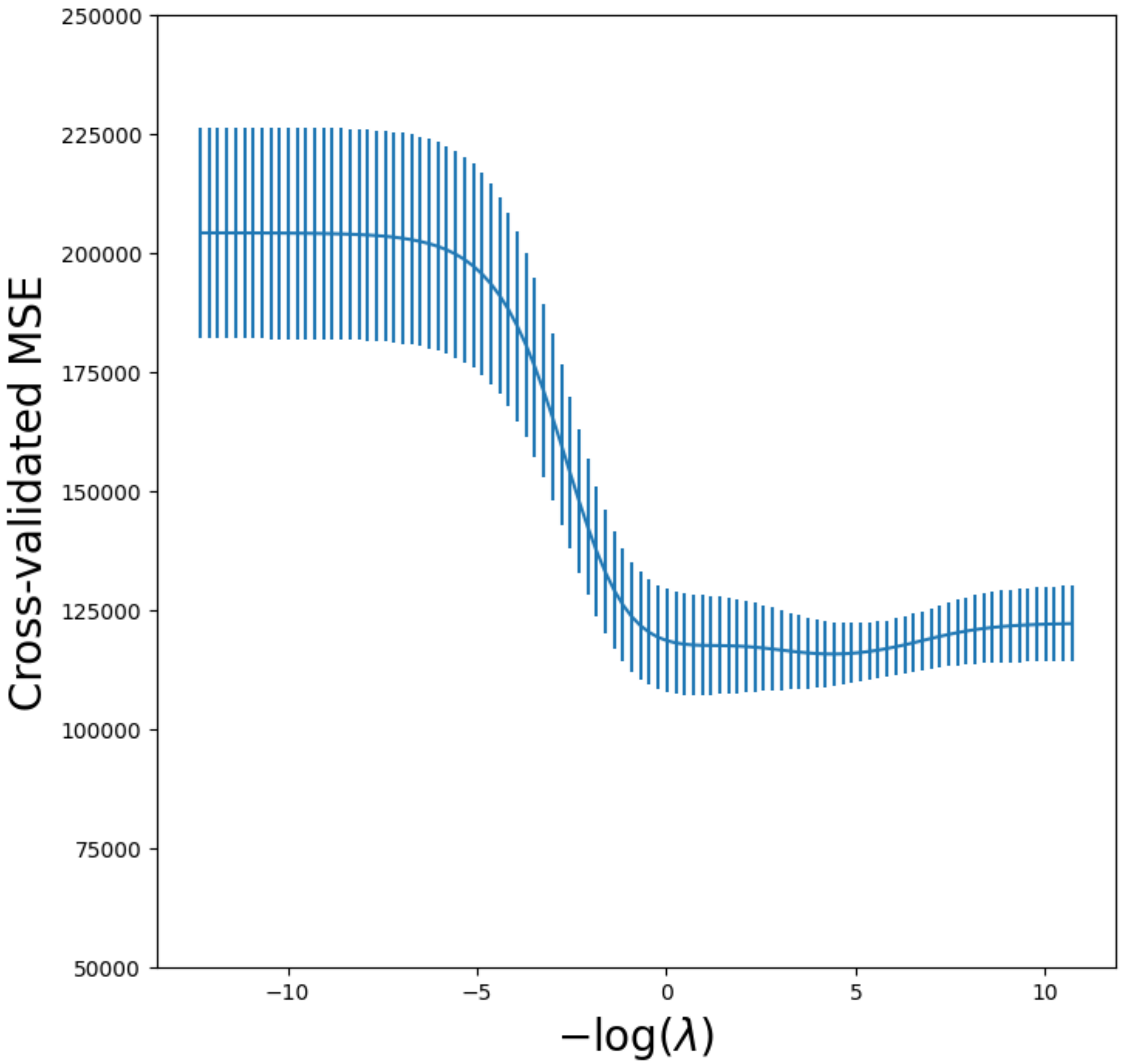
In [35]:

```
ridge_fig, ax = subplots(figsize=(8, 8))
ax.errorbar(-np.log(lambdas),
            -grid.cv_results_['mean_test_score'],
            yerr=grid.cv_results_['std_test_score'] / np.sqrt(K))
ax.set_ylim([50000, 250000])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Cross-validated MSE', fontsize=20)
```

```
<>:6: SyntaxWarning: invalid escape sequence '\l'
<>:6: SyntaxWarning: invalid escape sequence '\l'
/var/folders/97/23ltc4v96g31pp78_gyv6dvm0000gn/T/ipykernel_10519/252384649.p
y:6: SyntaxWarning: invalid escape sequence '\l'
  ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
```

Out[35]:

```
Text(0, 0.5, 'Cross-validated MSE')
```



In [36]:
```
grid_r2 = skm.GridSearchCV(pipe,
                           param_grid,
                           cv=kfold)
grid_r2.fit(X, Y)
```

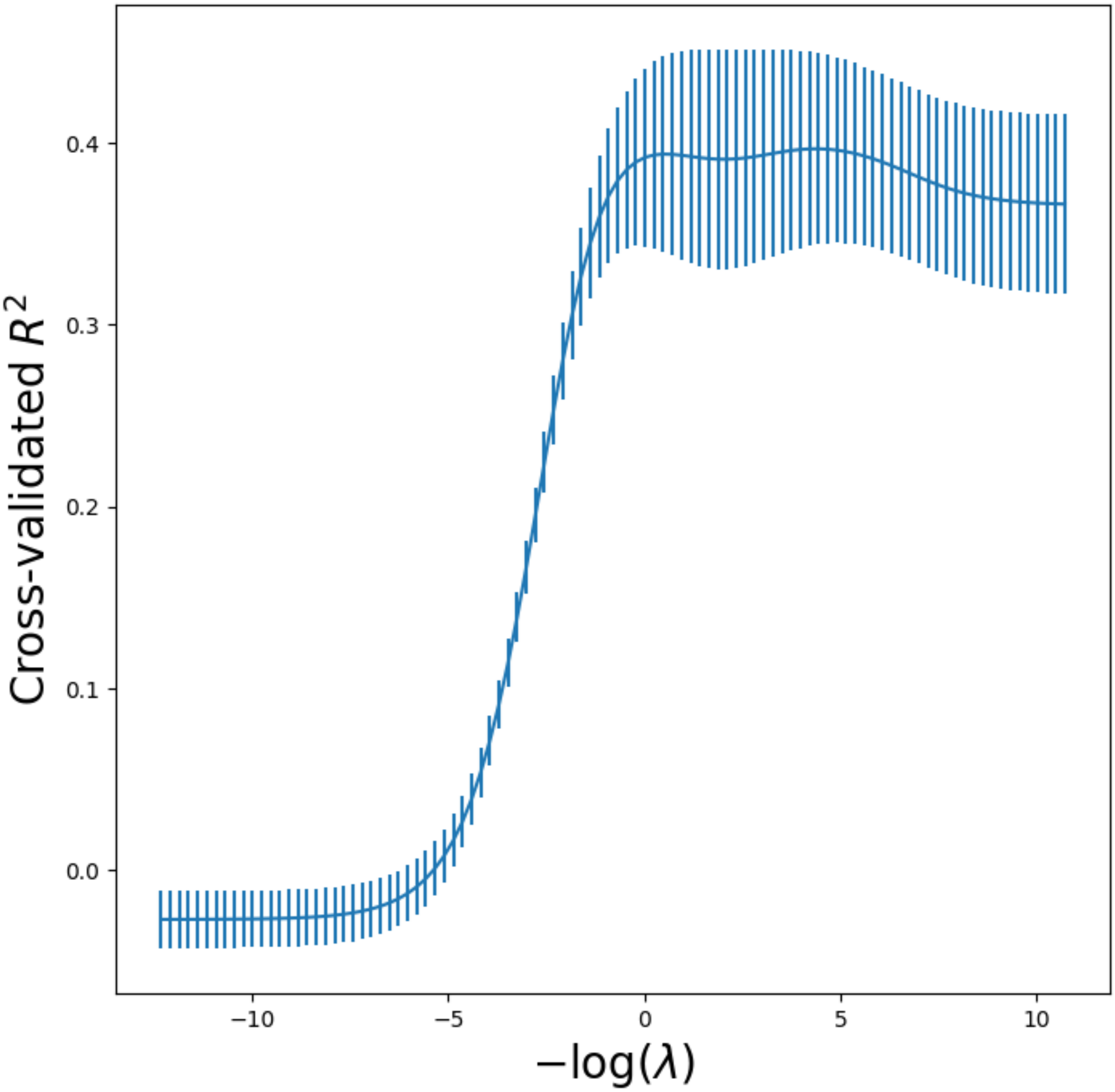
Out[36]:
```
▸       GridSearchCV
▸ best_estimator_: Pipeline
     ▸ StandardScaler
       ▸ ElasticNet
```

In [37]:
```
r2_fig, ax = subplots(figsize=(8, 8))
ax.errorbar(-np.log(lambdas),
             grid_r2.cv_results_['mean_test_score'],
             yerr=grid_r2.cv_results_['std_test_score'] / np.sqrt(K)
             )
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Cross-validated $R^2$', fontsize=20)
```

```
<>:6: SyntaxWarning: invalid escape sequence '\l'
<>:6: SyntaxWarning: invalid escape sequence '\l'
/var/folders/97/23ltc4v96g31pp78_gyv6dvm0000gn/T/ipykernel_10519/4088780906.
py:6: SyntaxWarning: invalid escape sequence '\l'
  ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
```

Out[37]:

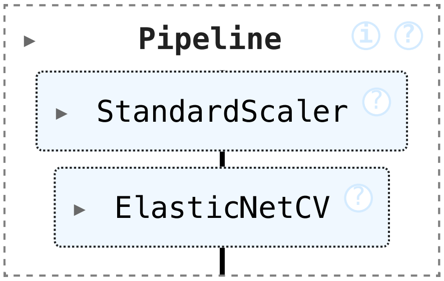
```
Text(0, 0.5, 'Cross-validated $R^2$')
```



In [38]:

```
ridgeCV = skl.ElasticNetCV(alphas=lambdas,
                           l1_ratio=0,
                           cv=kfold)
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('ridge', ridgeCV)])
pipeCV.fit(X, Y)
```

Out[38]:



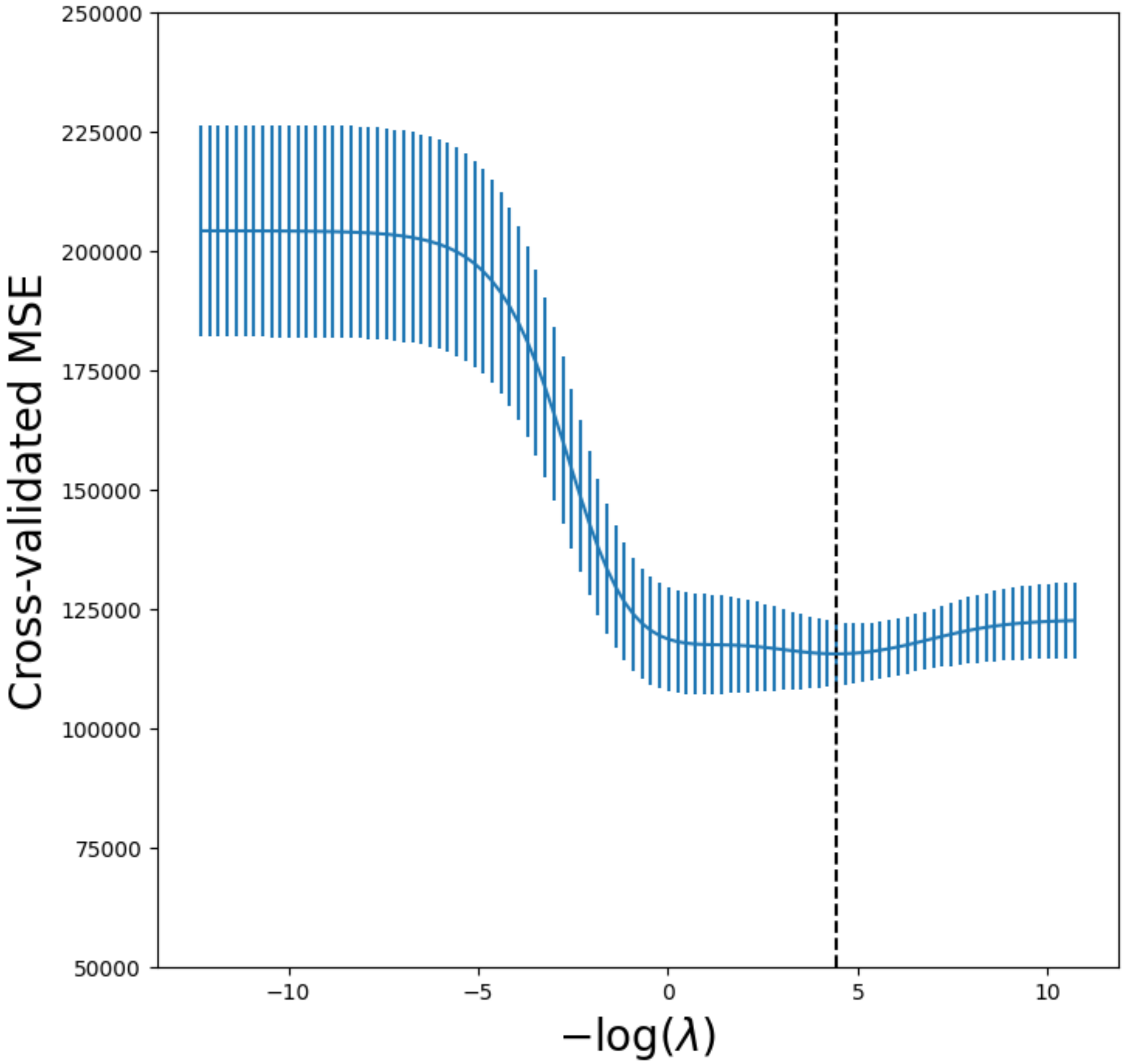
In [39]:

```
tuned_ridge = pipeCV.named_steps['ridge']
ridgeCV_fig, ax = subplots(figsize=(8, 8))
ax.errorbar(-np.log(lambdas),
             tuned_ridge.mse_path_.mean(1),
             yerr=tuned_ridge.mse_path_.std(1) / np.sqrt(K))
ax.axvline(-np.log(tuned_ridge.alpha_), c='k', ls='--')
ax.set_ylim([50000, 250000])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Cross-validated MSE', fontsize=20)
```

```
<>:8: SyntaxWarning: invalid escape sequence '\l'
<>:8: SyntaxWarning: invalid escape sequence '\l'
/var/folders/97/23ltc4v96g31pp78_gyv6dvm0000gn/T/ipykernel_10519/35476348.p
y:8: SyntaxWarning: invalid escape sequence '\l'
  ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
```

Out[39]:

```
Text(0, 0.5, 'Cross-validated MSE')
```



```
In [40]: np.min(tuned_ridge.mse_path_.mean(1))
```

```
Out[40]: 115526.70630987917
```

```
In [41]: tuned_ridge.coef_
```

```
Out[41]: array([-222.80877051,  238.77246614,    3.21103754,   -2.93050845,
                    3.64888723,  108.90953869,  -50.81896152, -105.15731984,
                  122.00714801,   57.1859509 ,  210.35170348,  118.05683748,
                 -150.21959435,   30.36634231,  -61.62459095,   77.73832472,
                   40.07350744,  -25.02151514,  -13.68429544])
```

```
In [42]: outer_valid = skm.ShuffleSplit(n_splits=1,
                                        test_size=0.25,
                                        random_state=1)
         inner_cv = skm.KFold(n_splits=5,
                              shuffle=True,
                              random_state=2)
```

```
ridgeCV = skl.ElasticNetCV(alphas=lambdas,
                           l1_ratio=0,
                           cv=inner_cv)
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('ridge', ridgeCV)])
```

In [43]:

```
results = skm.cross_validate(pipeCV,
                             X,
                             Y,
                             cv=outer_valid,
                             scoring='neg_mean_squared_error')
-results['test_score']
```

Out[43]:

```
array([132393.84003227])
```

In [44]:

```
lassoCV = skl.ElasticNetCV(n_alphas=100,
                           l1_ratio=1,
                           cv=kfold)
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('lasso', lassoCV)])
pipeCV.fit(X, Y)
tuned_lasso = pipeCV.named_steps['lasso']
tuned_lasso.alpha_
```

Out[44]:

```
3.1472370031649866
```

In [45]:

```
lambdas, soln_array = skl.Lasso.path(Xs,
                                     Y,
                                     l1_ratio=1,
                                     n_alphas=100)[:2]
soln_path = pd.DataFrame(soln_array.T,
                         columns=D.columns,
                         index=-np.log(lambdas))
```

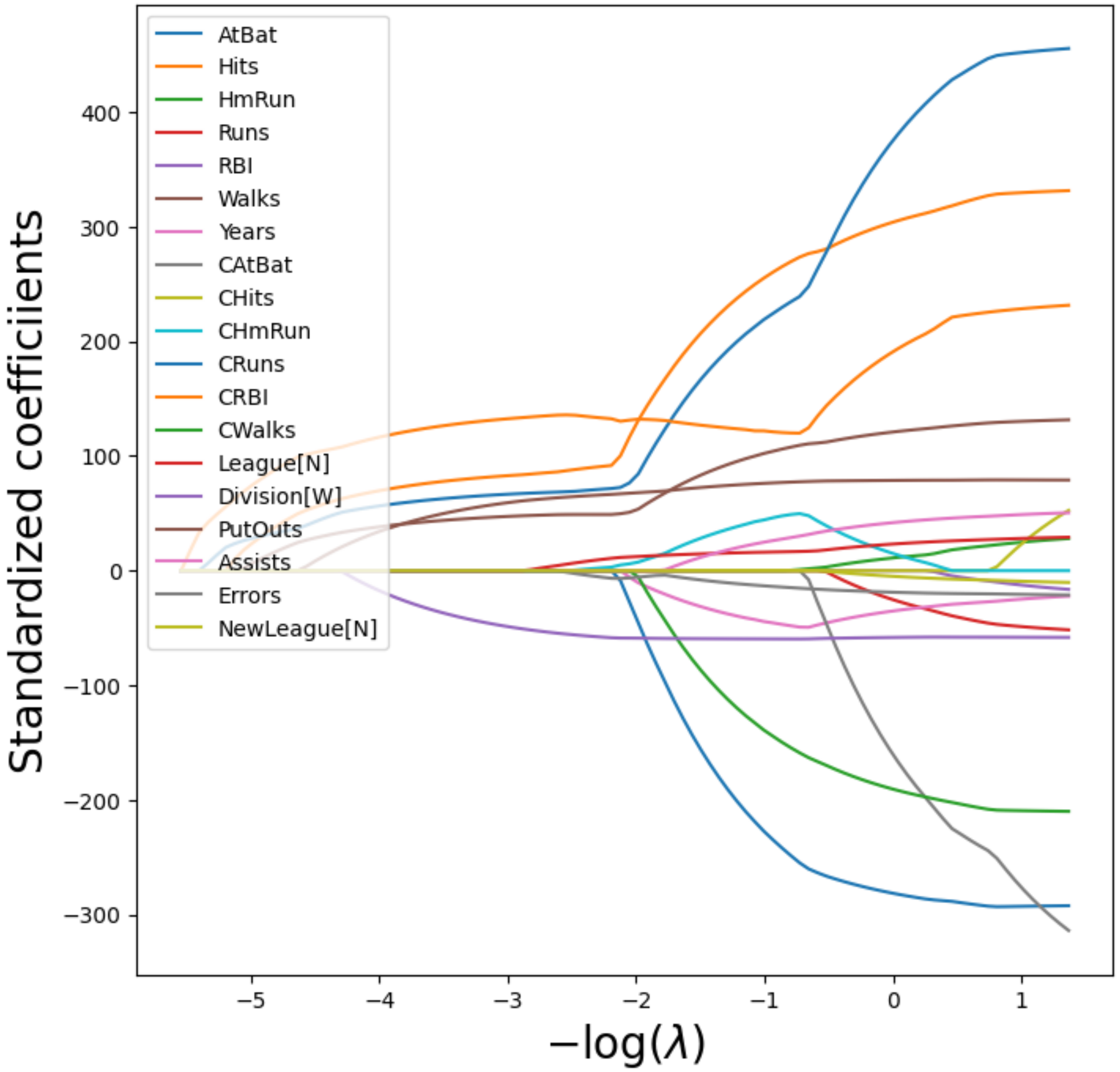
In [46]:

```
path_fig, ax = subplots(figsize=(8, 8))
soln_path.plot(ax=ax, legend=False)
ax.legend(loc='upper left')
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Standardized coefficiients', fontsize=20)
```

```
<>:4: SyntaxWarning: invalid escape sequence '\l'
<>:4: SyntaxWarning: invalid escape sequence '\l'
/var/folders/97/23ltc4v96g31pp78_gyv6dvm0000gn/T/ipykernel_10519/1325931816.
py:4: SyntaxWarning: invalid escape sequence '\l'
  ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
```

Out[46]:

```
Text(0, 0.5, 'Standardized coefficiients')
```



```
In [47]: np.min(tuned_lasso.mse_path_.mean(1))
```

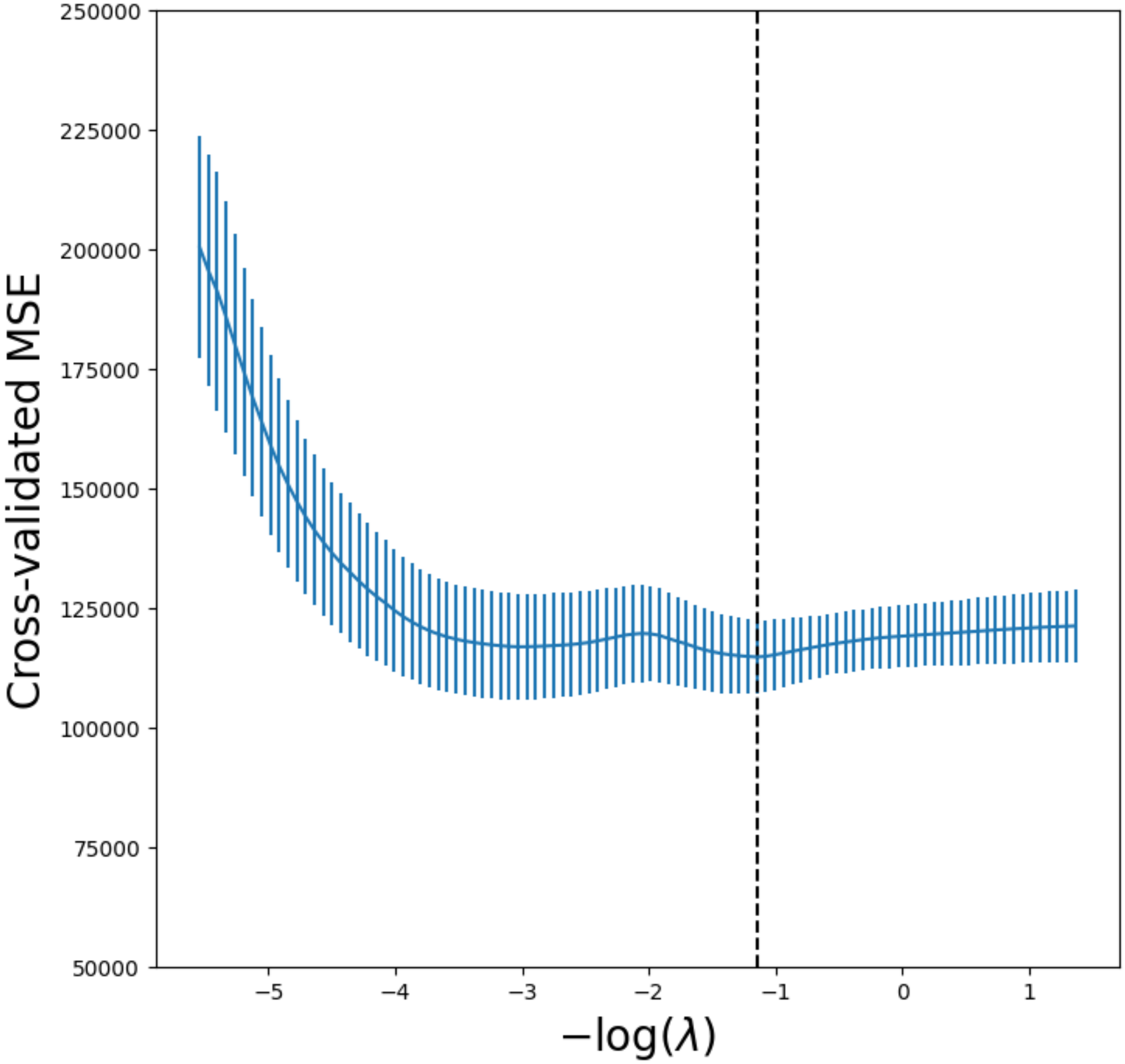
```
Out[47]: 114690.73118253727
```

```
In [48]: lassoCV_fig, ax = subplots(figsize=(8, 8))
         ax.errorbar(-np.log(tuned_lasso.alphas_),
                     tuned_lasso.mse_path_.mean(1),
                     yerr=tuned_lasso.mse_path_.std(1) / np.sqrt(K))
         ax.axvline(-np.log(tuned_lasso.alpha_), c='k', ls='--')
         ax.set_ylim([50000, 250000])
         ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
         ax.set_ylabel('Cross-validated MSE', fontsize=20)
```

```
<>:7: SyntaxWarning: invalid escape sequence '\l'
<>:7: SyntaxWarning: invalid escape sequence '\l'
/var/folders/97/23ltc4v96g31pp78_gyv6dvm0000gn/T/ipykernel_10519/3583132959.
py:7: SyntaxWarning: invalid escape sequence '\l'
  ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
```

Out[48]:

```
Text(0, 0.5, 'Cross-validated MSE')
```



In [49]:

```
tuned_lasso.coef_
```

Out[49]:

```
array([-210.01008773,  243.4550306 ,    0.        ,    0.        ,
          0.        ,   97.69397357,  -41.52283116,   -0.        ,
          0.        ,   39.62298193,  205.75273856,  124.55456561,
       -126.29986768,   15.70262427,  -59.50157967,   75.24590036,
         21.62698014,  -12.04423675,   -0.        ])
```

In [50]:

```
pca = PCA(n_components=2)
linreg = skl.LinearRegression()
pipe = Pipeline([('pca', pca),
                 ('linreg', linreg)])
pipe.fit(X, Y)
pipe.named_steps['linreg'].coef_
```

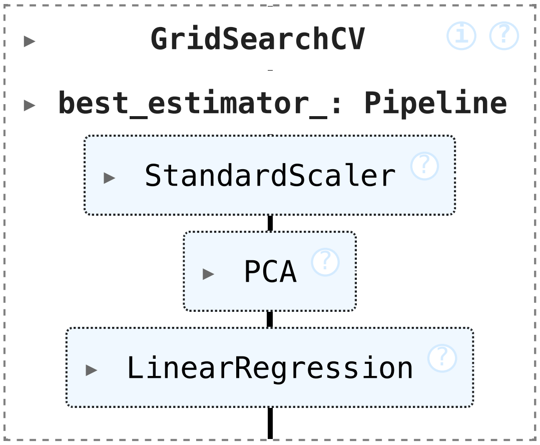
```
Out[50]:  array([0.09846131, 0.4758765 ])
```

```
In [51]: pipe = Pipeline([('scaler', scaler),
                          ('pca', pca),
                          ('linreg', linreg)])
         pipe.fit(X, Y)
         pipe.named_steps['linreg'].coef_
```

```
Out[51]:  array([106.36859204,  21.60350456])
```

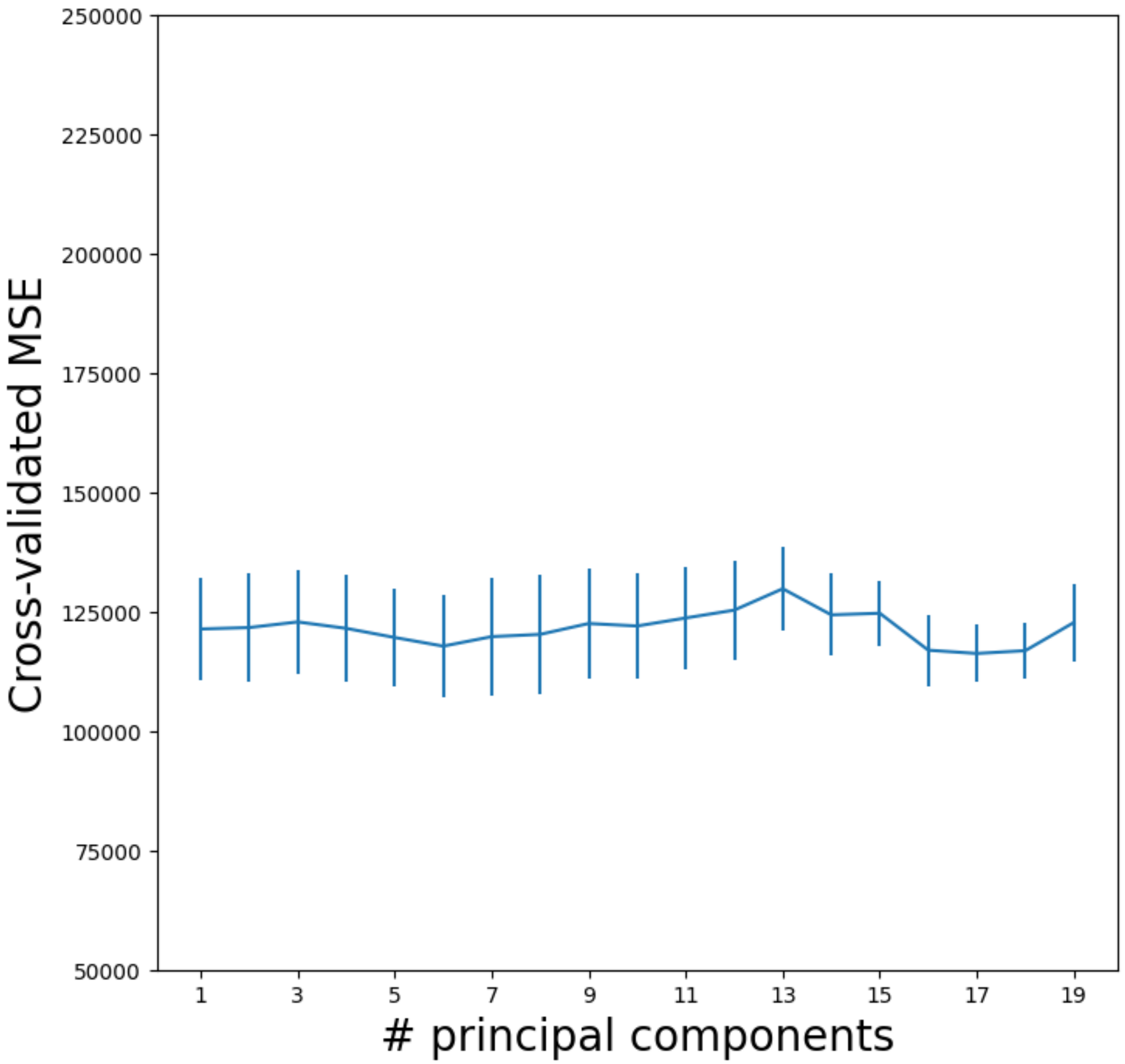
```
In [52]: param_grid = {'pca__n_components': range(1, 20)}
         grid = skm.GridSearchCV(pipe,
                                 param_grid,
                                 cv=kfold,
                                 scoring='neg_mean_squared_error')
         grid.fit(X, Y)
```

Out[52]:



```
In [53]: pcr_fig, ax = subplots(figsize=(8, 8))
         n_comp = param_grid['pca__n_components']
         ax.errorbar(n_comp,
                     -grid.cv_results_['mean_test_score'],
                     grid.cv_results_['std_test_score'] / np.sqrt(K))
         ax.set_ylabel('Cross-validated MSE', fontsize=20)
         ax.set_xlabel('# principal components', fontsize=20)
         ax.set_xticks(n_comp[::2])
         ax.set_ylim([50000, 250000])
```

```
Out[53]:  (50000.0, 250000.0)
```



In [54]:

```
Xn = np.zeros((X.shape[0], 1))
cv_null = skm.cross_validate(linreg,
                             Xn,
                             Y,
                             cv=kfold,
                             scoring='neg_mean_squared_error')
-cv_null['test_score'].mean()
```

Out[54]:

```
204139.30692994667
```

In [55]:

```
pipe.named_steps['pca'].explained_variance_ratio_
```

Out[55]:

```
array([0.3831424 , 0.21841076])
```

In [56]:

```
pls = PLSRegression(n_components=2,
                    scale=True)
pls.fit(X, Y)
```

```
Out[56]:
▾ PLSRegression
PLSRegression()
```

```
In [57]:
```

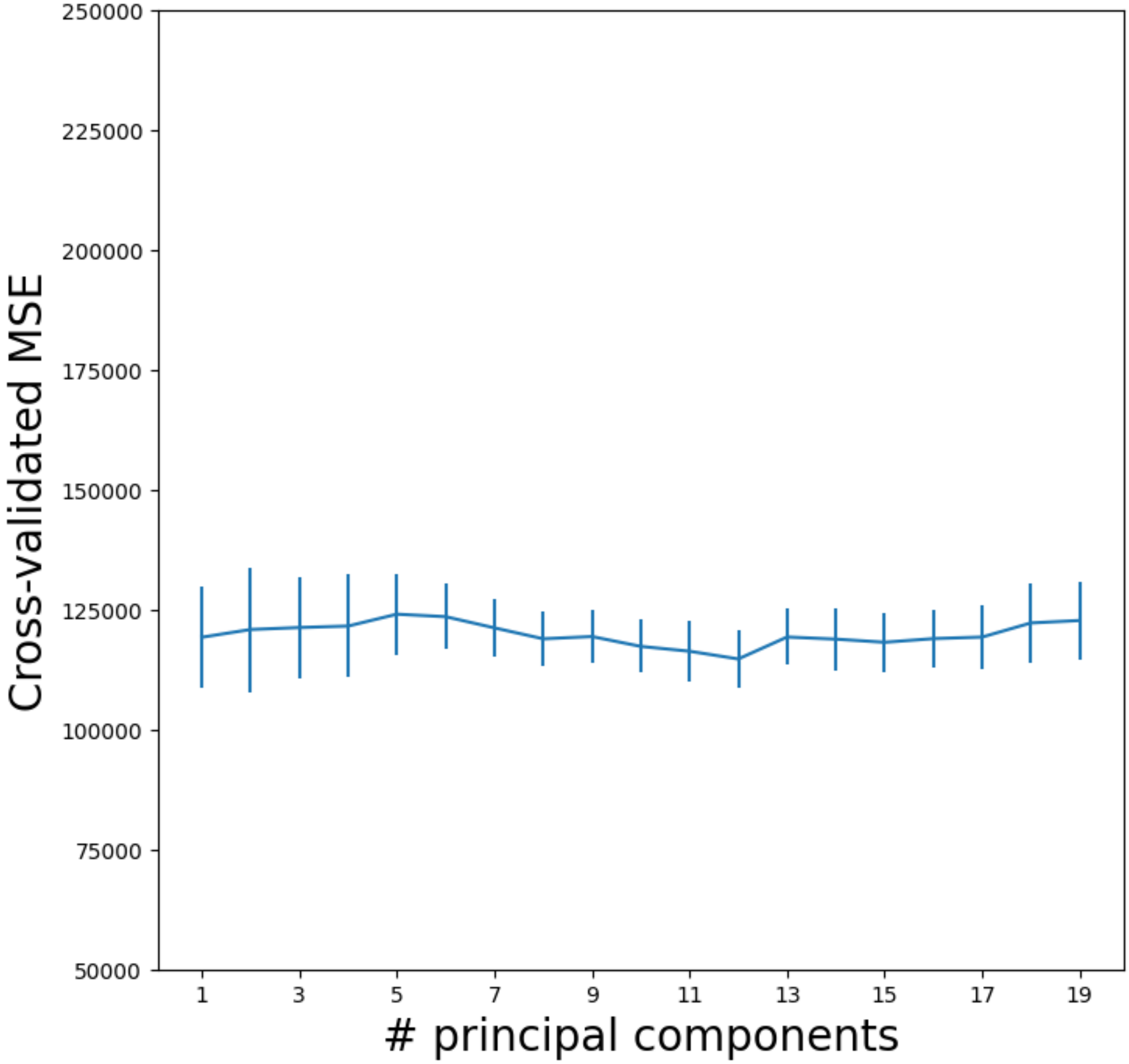
```python
param_grid = {'n_components': range(1, 20)}
grid = skm.GridSearchCV(pls,
                        param_grid,
                        cv=kfold,
                        scoring='neg_mean_squared_error')
grid.fit(X, Y)
```

```
Out[57]:
▸ GridSearchCV
▸ best_estimator_: PLSRegression
    ▸ PLSRegression
```

```
In [58]:
```

```python
pls_fig, ax = subplots(figsize=(8, 8))
n_comp = param_grid['n_components']
ax.errorbar(n_comp,
            -grid.cv_results_['mean_test_score'],
            grid.cv_results_['std_test_score'] / np.sqrt(K))
ax.set_ylabel('Cross-validated MSE', fontsize=20)
ax.set_xlabel('# principal components', fontsize=20)
ax.set_xticks(n_comp[::2])
ax.set_ylim([50000, 250000])
```

```
Out[58]: (50000.0, 250000.0)
```


250000
225000
200000
175000
150000
125000
100000
75000
50000
Cross-validated MSE
1
3
5
7
9
11
13
15
17
19
# principal components

STAT702 Assignment 3, Problem 4-12

Daniel Lee

March 10, 2025

4-12.

a. First:
odds ratio $= \frac{exp(\hat{\beta}_0+\hat{\beta}_1 x)}{1}$
Then:
log(odds ratio) $= \hat{\beta}_0 + \hat{\beta}_1 x$

b. First:
odds ratio $= \frac{exp(\hat{\alpha}_{orange0}+\hat{\alpha}_{orange1}x)}{exp(\hat{\alpha}_{apple0}+\hat{\alpha}_{apple1}x)}$
Then:
log(odds ratio) $= \hat{\alpha}_{orange0} + \hat{\alpha}_{orange1}x - (\hat{\alpha}_{apple0} + \hat{\alpha}_{apple1}x)$
$= (\hat{\alpha}_{orange0} - \hat{\alpha}_{apple0}) + (\hat{\alpha}_{orange1} - \hat{\alpha}_{apple1})x$

c. Since my friend's equation is also written as $\hat{\beta}_0 + \hat{\beta}_1 x$, we can set:
$2 = \hat{\alpha}_{orange0} - \hat{\alpha}_{apple0}$, $-1 = \hat{\alpha}_{orange1} - \hat{\alpha}_{apple1}$
Then we could solve this system of equations
(or something else, maybe set $\hat{\alpha}_{orange0} =$ (a function of $\hat{\alpha}_{apple0}$),
and the same for $\hat{\alpha}_{orange1}$?)

d. Based on the same formats of the two equations, we can do:
$\hat{\beta}_0 = 1.2 - 3 = -1.8, \hat{\beta}_1 = (-2) - 0.6 = -2.6$

e. Because we made the two models the same in d), I will expect the predictions for the two models to be the same 100%.

In [1]:

```
#pip install ISLP
```

In [2]:

```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
import statsmodels.api as sm
from ISLP import load_data
from ISLP.models import (ModelSpec as MS,
summarize)
```

In [3]:

```
from ISLP import confusion_table
from ISLP.models import contrast
from sklearn.discriminant_analysis import \
(LinearDiscriminantAnalysis as LDA, QuadraticDiscriminantAnalysis as QDA)
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

In [4]:

```
from ISLP import load_data
from ISLP.models import (ModelSpec as MS,
                         summarize,
                         poly)
```

In [5]:

```
Auto = load_data("Auto")
```

14a.

In [6]:

```
mpg01=(Auto['mpg']>Auto['mpg'].median()).astype(int)
mpg01
```

Out[6]:

	mpg
name	
chevrolet chevelle malibu	0
buick skylark 320	0
plymouth satellite	0
amc rebel sst	0
ford torino	0
...	...
ford mustang gl	1
vw pickup	1
dodge rampage	1
ford ranger	1
chevy s-10	1

392 rows × 1 columns

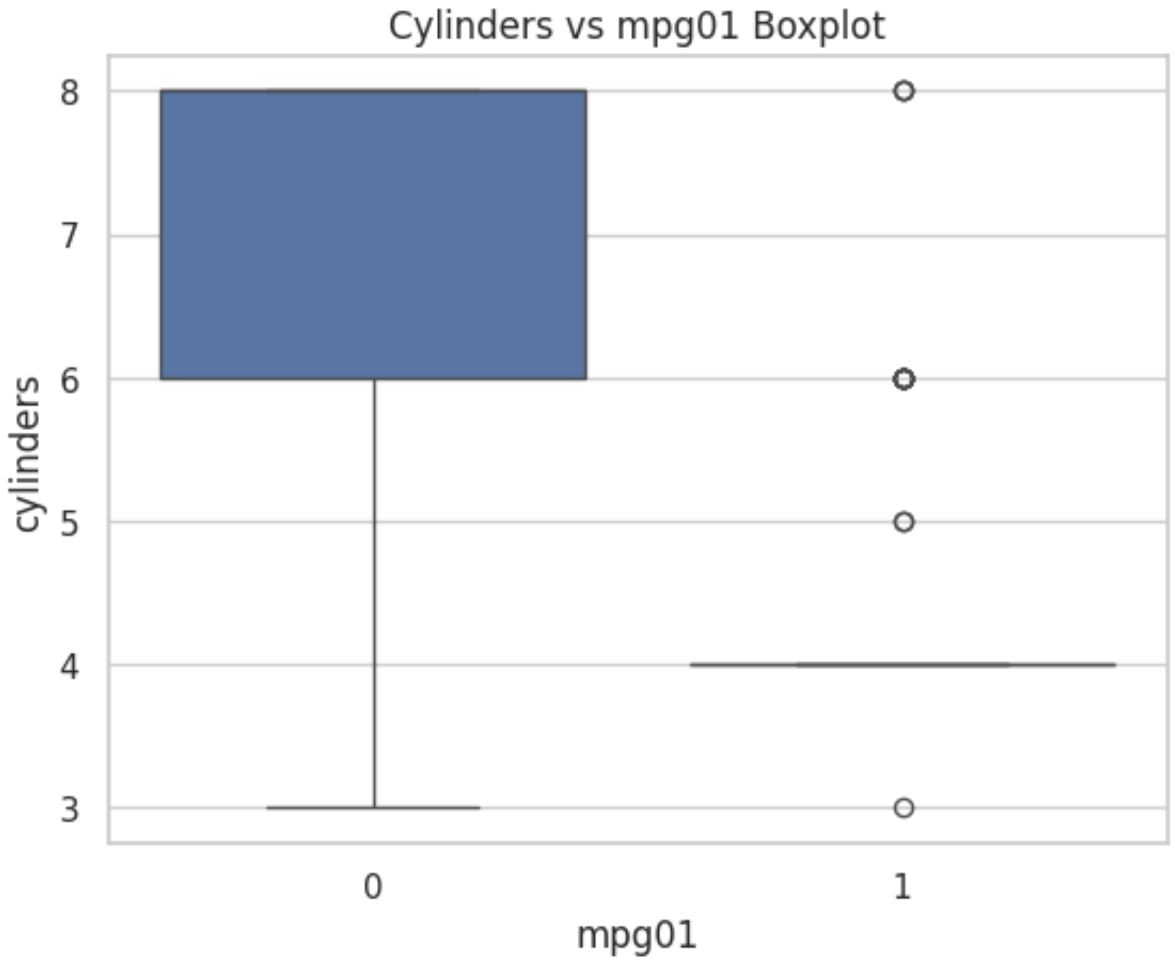
dtype: int64

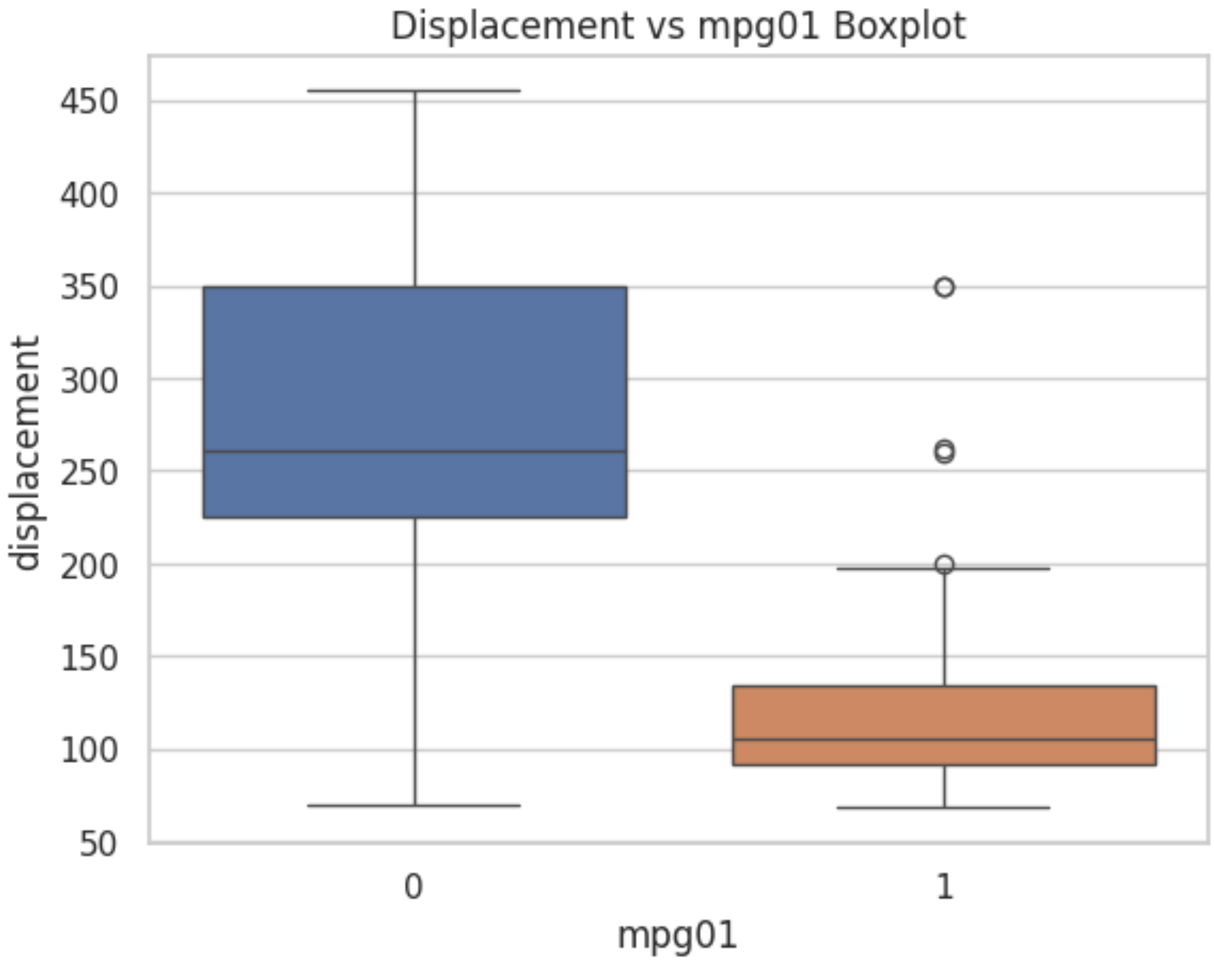
In [7]:
```
Auto['mpg01'] = mpg01
```

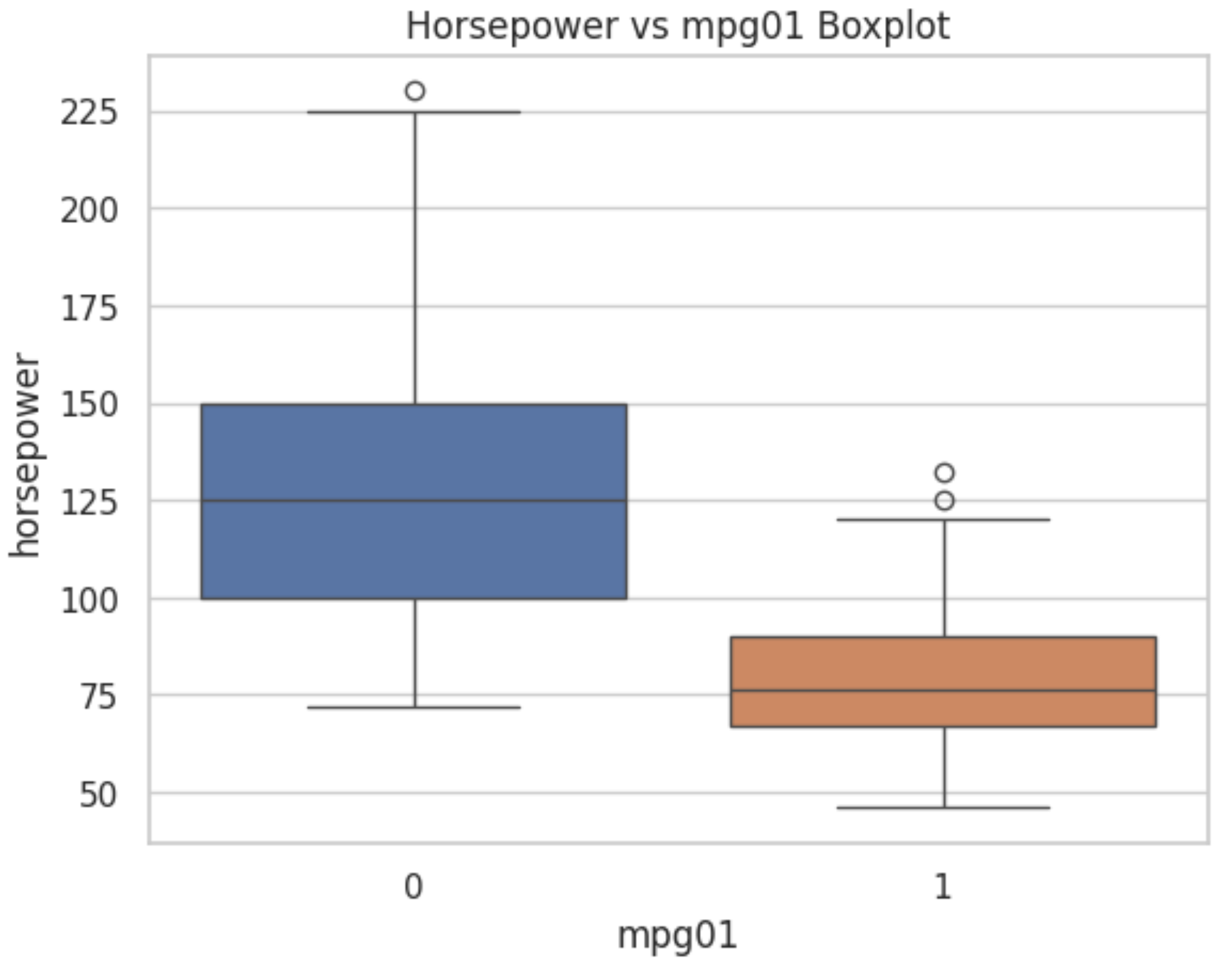
14b.

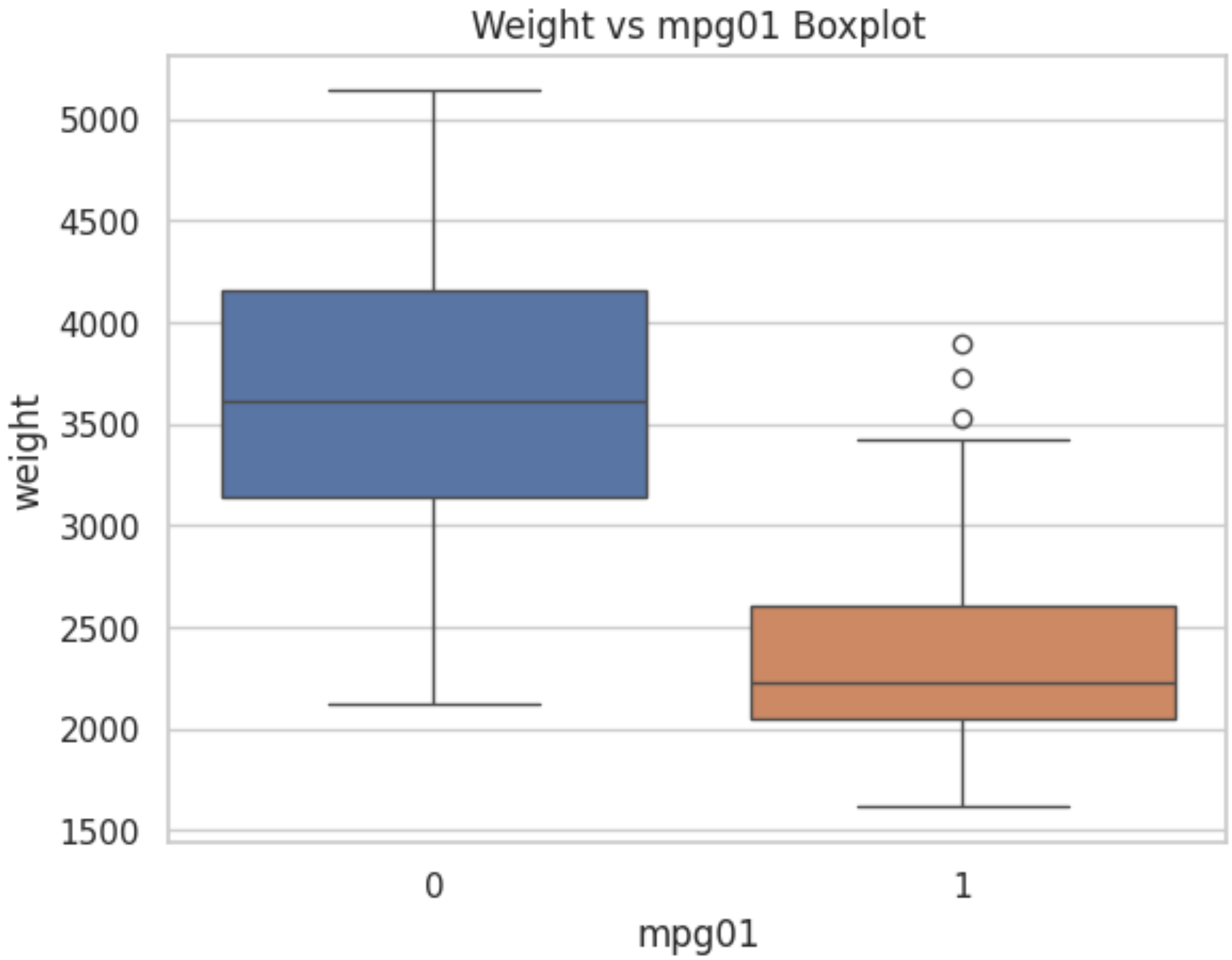
In [8]:
```
#Originally generated by CHATgpt
import matplotlib.pyplot as plt
import seaborn as sns
# Set plot style
sns.set(style="whitegrid")

# Convert 'origin' to categorical with labels
Auto['origin'] = Auto['origin'].replace({1: 'American', 2: 'European', 3: 'J

# Plot 1: Cylinders vs mpg01 Boxplot
g1 = sns.boxplot(data=Auto, x='mpg01', y='cylinders', hue='mpg01')
g1.legend_.remove()
plt.title('Cylinders vs mpg01 Boxplot')
plt.show()

# Plot 2: Displacement vs mpg01 Boxplot
g2 = sns.boxplot(data=Auto, x='mpg01', y='displacement', hue='mpg01')
g2.legend_.remove()
plt.title('Displacement vs mpg01 Boxplot')
plt.show()
```

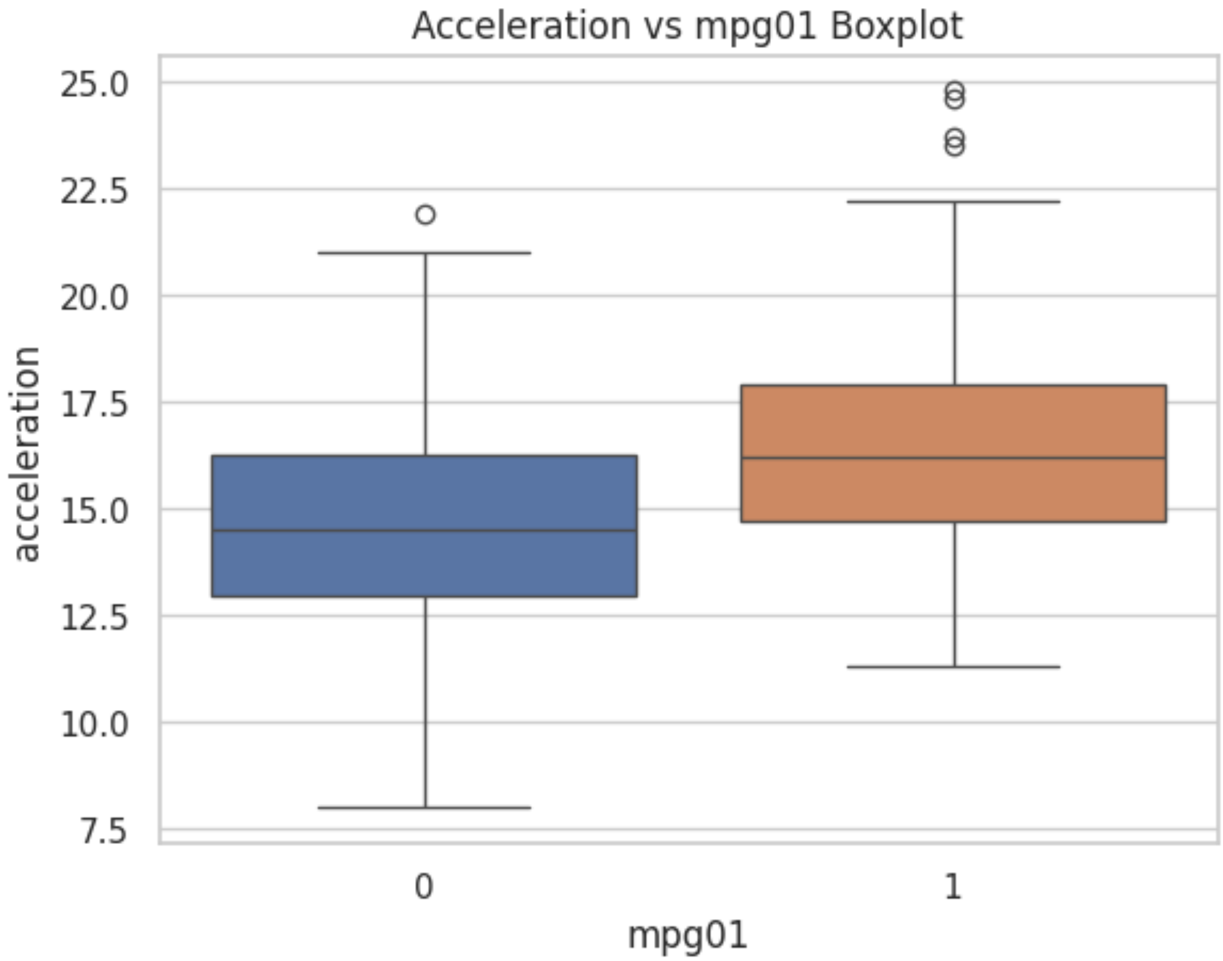
```python
# Plot 3: Horsepower vs mpg01 Boxplot
g3 = sns.boxplot(data=Auto, x='mpg01', y='horsepower', hue='mpg01')
g3.legend_.remove()
plt.title('Horsepower vs mpg01 Boxplot')
plt.show()

# Plot 4: Weight vs mpg01 Boxplot
g4 = sns.boxplot(data=Auto, x='mpg01', y='weight', hue='mpg01')
g4.legend_.remove()
plt.title('Weight vs mpg01 Boxplot')
plt.show()

# Plot 5: Acceleration vs mpg01 Boxplot
g5 = sns.boxplot(data=Auto, x='mpg01', y='acceleration', hue='mpg01')
g5.legend_.remove()
plt.title('Acceleration vs mpg01 Boxplot')
plt.show()

# Plot 6: Year vs mpg01 - Boxplot
g6 = sns.boxplot(data=Auto, x='mpg01', y='year', hue='mpg01')
g6.legend_.remove()
plt.title('Year vs mpg01 Boxplot')
plt.show()

# Plot 7: Origin vs mpg01 Bar plot
g7 = sns.histplot(data=Auto, x='origin', hue='mpg01', multiple='fill', shrin
plt.gca().yaxis.set_major_formatter(plt.FuncFormatter(lambda y, _: '{:.0%}'.
plt.ylabel('')
plt.title('Origin vs mpg01 Bar plot')
plt.show()
```

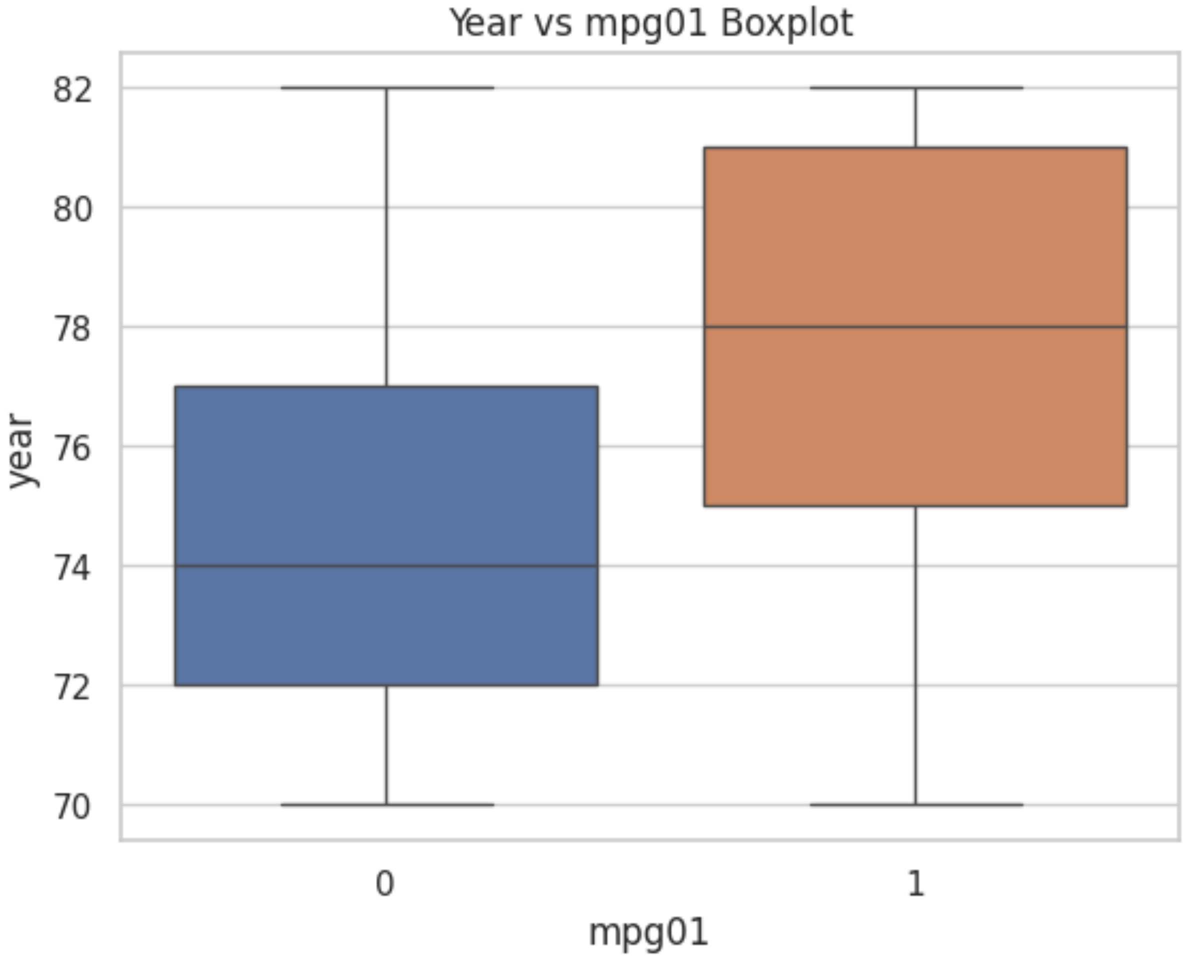

Cylinders vs mpg01 Boxplot
cylinders
8
7
6
5
4
3
0
1
mpg01

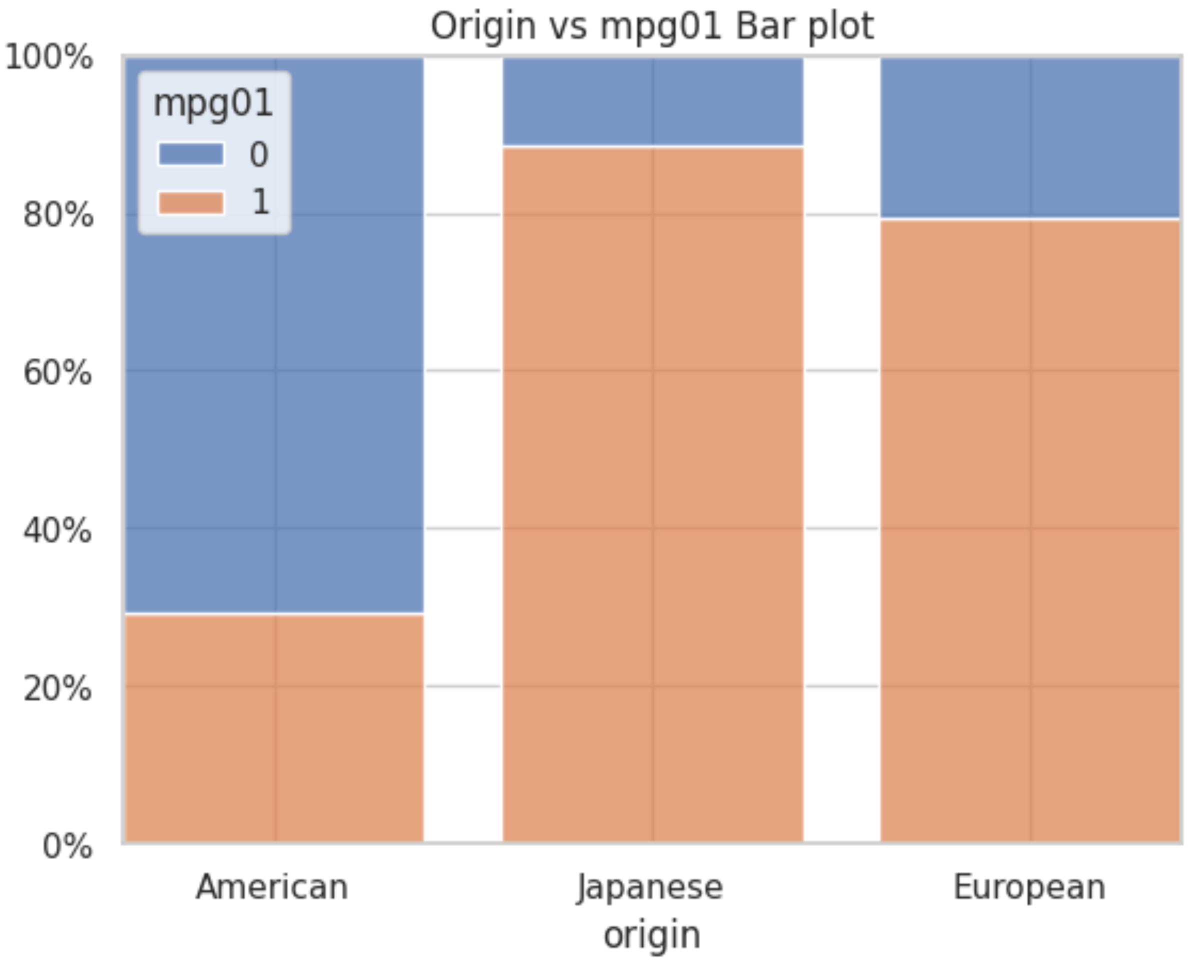

Displacement vs mpg01 Boxplot
displacement
450
400
350
300
250
200
150
100
50
0
1
mpg01


Horsepower vs mpg01 Boxplot
horsepower
225
200
175
150
125
100
75
50
0
1
mpg01


Weight vs mpg01 Boxplot
weight
5000
4500
4000
3500
3000
2500
2000
1500
0
1
mpg01


Acceleration vs mpg01 Boxplot
acceleration
25.0
22.5
20.0
17.5
15.0
12.5
10.0
7.5
0
1
mpg01


Year vs mpg01 Boxplot
82
80
78
76
74
72
70
year
0
1
mpg01



Explanation: It looks like all of the other features are useful in predicting mpg01, but I would say cylinders, displacement, weight, and horsepower are more powerful than the others.

14c.

```
In [9]: np.random.seed(123)
        train, test = train_test_split(Auto, test_size=0.5, random_state=123)
```

14d.

```
In [10]: # Prepare features and mpg01
         X_train = train[['cylinders', 'displacement', 'weight', 'horsepower']]
         y_train = train['mpg01']
         X_test = test[['cylinders', 'displacement', 'weight', 'horsepower']]
```

```
In [11]: lda = LDA(store_covariance=True)
         lda.fit(X_train, y_train)
         predicted_lda = lda.predict(X_test)

         # Calculate test error
         print(np.mean(predicted_lda != test['mpg01']))
```

```
0.10204081632653061
```

14e.

In [12]:

```
qda = QDA(store_covariance=True)
qda.fit(train[['cylinders', 'displacement', 'weight', 'horsepower']], train[
predicted_qda = qda.predict(X_test)
print(np.mean(predicted_qda != test['mpg01']))
```

```
0.09693877551020408
```

14f.

In [13]:

```
logit = LogisticRegression(C=1e10, solver='liblinear')
logit.fit(X_train, y_train)
logit_pred = logit.predict(X_test)
print(np.mean(logit_pred != test['mpg01']))
```

```
0.12755102040816327
```

14g.

In [14]:

```
NB = GaussianNB()
NB.fit(X_train, y_train)
NB_pred=NB.predict(X_test)
print(np.mean(NB_pred != test['mpg01']))
```

```
0.10204081632653061
```

14h.

In [15]:

```
from sklearn.model_selection import GridSearchCV
knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train, y_train)
knn_pred = knn.predict(X_test)
print(np.mean(knn_pred != test['mpg01']))
```

```
0.11734693877551021
```

The best value of k I could find was 7 since the error starts inccreasing after k=7.

Test error for k=1: 0.16326530612244897

Test error for k=8: 0.12244897959183673

16.

In [16]:

```
Boston = load_data("Boston")
crim01=(Boston['crim']>Boston['crim'].median()).astype(int)
crim01
```

Out[16]:

	crim
0	0
1	0
2	0
3	0
4	0
...	...
501	0
502	0
503	0
504	0
505	0

506 rows × 1 columns

dtype: int64

In [17]:

```
Boston['crim01'] = crim01
```

In [18]:

```
np.random.seed(234)
train, test = train_test_split(Boston, test_size=0.5, random_state=234)
```

In [19]:

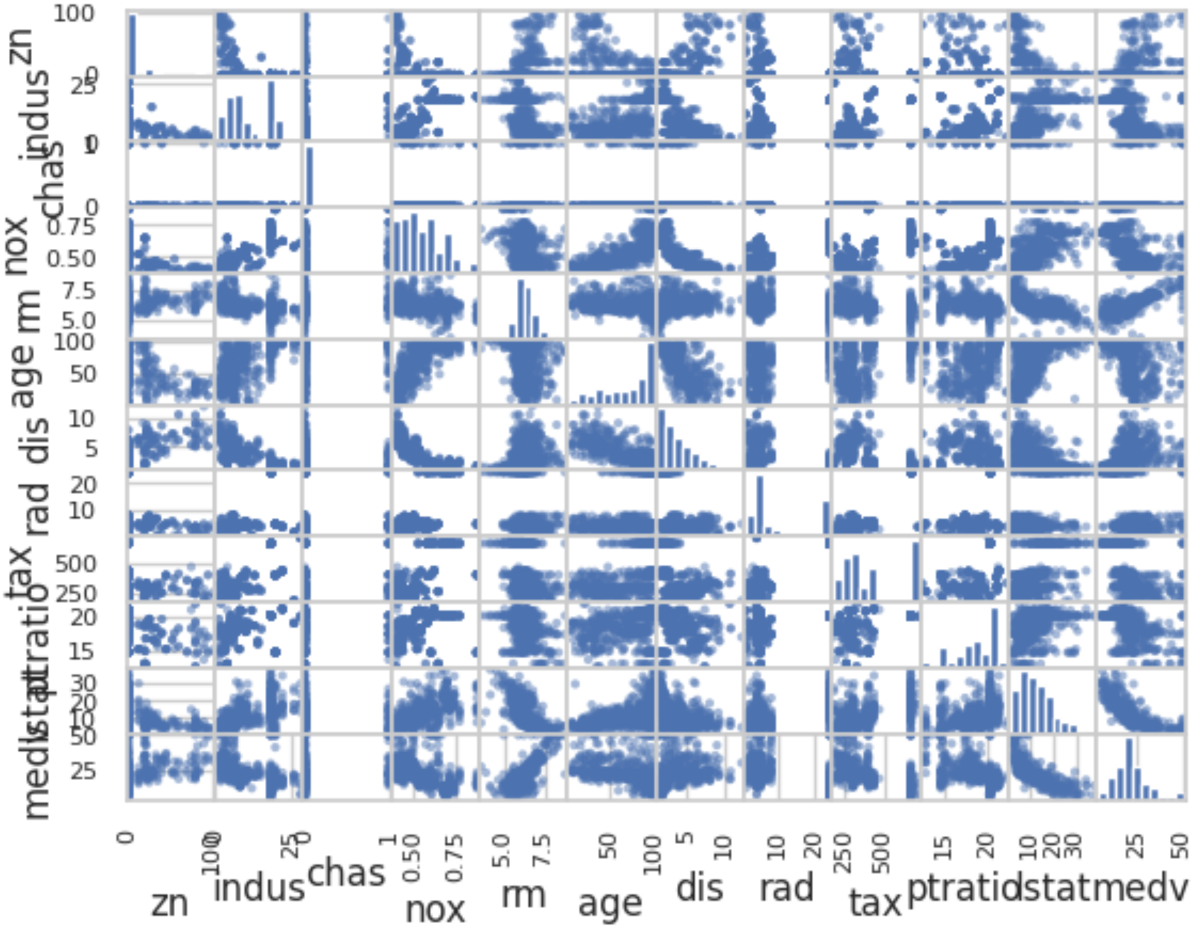
```
Boston.columns
```

Out[19]:

```
Index(['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'ta
x',
       'ptratio', 'lstat', 'medv', 'crim01'],
      dtype='object')
```

In [20]:

```
pd.plotting.scatter_matrix(Boston[['zn', 'indus', 'chas', 'nox', 'rm', 'age'
       'ptratio', 'lstat', 'medv']]);
```



In [21]:

```
# Prepare features and crim01
BosAll=['zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax', 'ptr
X_train = train[BosAll]
y_train = train['crim01']
X_test = test[BosAll]
#logit
logit = LogisticRegression(C=1e10, solver='liblinear')
logit.fit(X_train, y_train)
logit_pred = logit.predict(X_test)
print("Logit:"+ str(np.mean(logit_pred != test['crim01'])))
#lda
lda = LDA(store_covariance=True)
lda.fit(X_train, y_train)
predicted_lda = lda.predict(X_test)
print("LDA:"+str(np.mean(predicted_lda != test['crim01'])))
#NB
NB = GaussianNB()
NB.fit(X_train, y_train)
NB_pred=NB.predict(X_test)
print("NB:"+str(np.mean(NB_pred != test['crim01'])))
#KNN
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train, y_train)
knn_pred = knn.predict(X_test)
print("KNN:"+str(np.mean(knn_pred != test['crim01'])))
```

```
Logit:0.11462450592885376
LDA:0.18181818181818182
NB:0.2015810276679842
KNN:0.07114624505928854
```

In [22]:

```python
# Prepare features and crim01
BosSub=['zn', 'indus', 'rm', 'dis', 'rad', 'tax','nox']
X_train = train[BosSub]
y_train = train['crim01']
X_test = test[BosSub]
#logit
logit = LogisticRegression(C=1e10, solver='liblinear')
logit.fit(X_train, y_train)
logit_pred = logit.predict(X_test)
print("Logit:"+ str(np.mean(logit_pred != test['crim01'])))
#lda
lda = LDA(store_covariance=True)
lda.fit(X_train, y_train)
predicted_lda = lda.predict(X_test)
print("LDA:"+str(np.mean(predicted_lda != test['crim01'])))
#NB
NB = GaussianNB()
NB.fit(X_train, y_train)
NB_pred=NB.predict(X_test)
print("NB:"+str(np.mean(NB_pred != test['crim01'])))
#KNN
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train, y_train)
knn_pred = knn.predict(X_test)
print("KNN:"+str(np.mean(knn_pred != test['crim01'])))
```

```
Logit:0.1857707509881423
LDA:0.16205533596837945
NB:0.22134387351778656
KNN:0.05533596837944664
```

Findings:
For all predictors, I noticed that the best value of k for KNN is k=6.
KNN also gave the lowest mean error out of the 4 methods for this case.

But since some of the predictors appeared to be correlated with each other, I removed some of them (and created BosSub), which made the logistic and NB test errors to increase, while LDA and KNN test errors decreased (KNN is still the lowest).

STAT 702 - Homework 3

Noah Javadi

2025-03-09

##Setup

```
#install.packages('ISLR2','boot','glmnet','pls')
library(ISLR2)
```

```
## Warning: package 'ISLR2' was built under R version 4.3.3
```

```
library(boot)
library(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 4.3.3
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 4.1-8
```

```
library(pls)
```

```
## Warning: package 'pls' was built under R version 4.3.3
```

```
##
## Attaching package: 'pls'
```

```
## The following object is masked from 'package:stats':
##
##     loadings
```

##Problem 5-9

```
#Problem 9a
u <- mean(Boston$medv)
u
```

```
## [1] 22.53281
```

```
#Problem 9b
se1 <- sd(Boston$medv)/sqrt(nrow(Boston))
se1
```

```
## [1] 0.4088611
```

```
#Problem 9c - The bootstrap estimation is very close to the manual calculation from b
se.fn <- function(Boston, index) {
  x <- Boston$medv[index]
  y <- length(index)

  se <- sd(x)/sqrt(y)
  se
}
se.fn(Boston,1:500)
```

```
## [1] 0.4130334
```

```
boot(Boston,se.fn,R=1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Boston, statistic = se.fn, R = 1000)
##
##
## Bootstrap Statistics :
##      original       bias    std. error
## t1* 0.4088611 -0.001018028  0.01657769
```

```
#Problem 9d - The results from the manual calculation and t.test are very close as well
CI.u <- c(u - 1.96*boot(Boston,se.fn,R=1000)[[1]],u + 1.96*boot(Boston,se.fn,R=1000)[[1]])
CI.u
```

```
## [1] 21.73144 23.33417
```

```
t.test(Boston$medv)
```

```
##
##  One Sample t-test
##
## data:  Boston$medv
## t = 55.111, df = 505, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  21.72953 23.33608
## sample estimates:
## mean of x
##  22.53281
```

```r
#Problem 9e
median(Boston$medv)
```

```
## [1] 21.2
```

```r
#Problem 9f -
se.fn(Boston,1:quantile(Boston$medv,probs = c(0.5)))
```

```
## [1] 1.375661
```

```r
#Problem 9g
u0.1 <- mean(quantile(Boston$medv),probs = c(0.1))
u0.1
```

```
## [1] 23.645
```

```r
#Problem 9h -
se.fn(Boston,1:quantile(Boston$medv,probs = c(0.1)))
```

```
## [1] 2.080688
```

##Problem 6-9

```r
#Problem 9a
College.Train <- College[1:388,]
College.Test <- College[389:777,]

College.Trainy <- sample(1:nrow(College),nrow(College)/2)
College.Testy <- (-College.Trainy)
Apps.test <- College$Apps[College.Testy]

#Problem 9b
model.train <- lm(Apps ~ .,data = College.Train)
summary(model.train)
```

```
##
## Call:
## lm(formula = Apps ~ ., data = College.Train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2721.8  -337.1   -30.6   253.5  6341.8
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -7.681e+02  4.895e+02  -1.569  0.11748
## PrivateYes  -4.834e+02  1.748e+02  -2.766  0.00596 **
## Accept       1.202e+00  7.875e-02  15.267  < 2e-16 ***
## Enroll       8.369e-02  2.780e-01   0.301  0.76358
## Top10perc    4.062e+01  6.950e+00   5.845 1.11e-08 ***
## Top25perc   -1.353e+01  5.692e+00  -2.377  0.01795 *
## F.Undergrad  3.144e-02  4.338e-02   0.725  0.46902
## P.Undergrad  6.651e-03  5.587e-02   0.119  0.90530
## Outstate    -2.514e-02  2.306e-02  -1.090  0.27638
## Room.Board   1.872e-01  5.813e-02   3.220  0.00139 **
## Books       -1.958e-01  2.599e-01  -0.753  0.45166
## Personal     1.062e-01  8.226e-02   1.291  0.19756
## PhD          7.152e-01  6.115e+00   0.117  0.90695
## Terminal    -1.088e+01  6.780e+00  -1.604  0.10956
## S.F.Ratio    1.168e+01  1.508e+01   0.774  0.43915
## perc.alumni -6.553e+00  5.098e+00  -1.285  0.19944
## Expend       8.440e-02  1.415e-02   5.963 5.78e-09 ***
## Grad.Rate    7.445e+00  3.515e+00   2.118  0.03483 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 884.6 on 370 degrees of freedom
## Multiple R-squared:  0.9072, Adjusted R-squared:  0.9029
## F-statistic: 212.7 on 17 and 370 DF,  p-value: < 2.2e-16
```

```r
#Problem 9c
grid <- 10^seq(10, -2, length = 388)
ridge.train <- glmnet(College.Train[,c(3:18)],College.Train$Apps,alpha=0,thresh = 1e-12)

ridge.mod <- glmnet(model.matrix(Apps ~ .,College.Train),College.Trainy,alpha = 0,lambda = grid)

cv.out <- cv.glmnet(model.matrix(Apps ~ .,College.Train),College.Trainy,alpha=0)

bestlam <- cv.out$lambda.min

ridge.pred <- predict(ridge.mod, s=bestlam, newx = model.matrix(Apps ~ .,College.Train))
#ridge.pred

mean((ridge.pred - College.Trainy)^2)
```

```
## [1] 50486.42
```

```r
#Problem 9d
lasso.train <- glmnet(College.Train[,c(1,3:18)],College.Train$Apps,alpha=1)

lasso.mod <- glmnet(model.matrix(Apps ~ .,College.Train),College.Trainy,alpha = 1,lambda = grid)

cv.out <- cv.glmnet(model.matrix(Apps ~ .,College.Train),College.Trainy,alpha=1)

bestlam <- cv.out$lambda.min

lasso.pred <- predict(lasso.mod, s=bestlam, newx = model.matrix(Apps ~ .,College.Train))
#lasso.pred

lasso.pred <- predict(lasso.mod, s=bestlam, type = 'coefficients')[1:19,]
#lasso.pred

mean((lasso.pred - College.Trainy)^2)
```

```
## Warning in lasso.pred - College.Trainy: longer object length is not a multiple
## of shorter object length
```

```
## [1] 200053.7
```

```
#Problem 9e - M = 10
pcr.fit <- pcr(Apps ~ .,data = College,scale = TRUE, validation = "CV")
summary(pcr.fit)
```

```
## Data:    X dimension: 777 17 
##  Y dimension: 777 1
## Fit method: svdpc
## Number of components considered: 17
## 
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV            3873     3836     2024     2030     1748     1587     1579
## adjCV         3873     3837     2022     2030     1651     1579     1577
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## CV        1570     1540     1498      1491      1495      1497      1502
## adjCV     1571     1534     1495      1489      1492      1494      1499
##        14 comps  15 comps  16 comps  17 comps
## CV         1503      1443      1172      1131
## adjCV      1500      1424      1165      1125
## 
## TRAINING: % variance explained
##       1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps
## X      31.670    57.30    64.30    69.90    75.39    80.38    83.99    87.40
## Apps    2.316    73.06    73.07    82.08    84.08    84.11    84.32    85.18
##       9 comps  10 comps  11 comps  12 comps  13 comps  14 comps  15 comps
## X       90.50     92.91     95.01     96.81      97.9     98.75     99.36
## Apps    85.88     86.06     86.06     86.10      86.1     86.13     90.32
##       16 comps  17 comps
## X        99.84    100.00
## Apps     92.52     92.92
```

```
pcr.pred <- predict(pcr.fit,College.Test,ncomp = 10)
#pcr.pred

mean((pcr.pred - Apps.test)^2)
```

```
## [1] 31777007
```

```
#Problem 9f - M = 10
pls.fit <- plsr(Apps ~ ., data = College, subset = College.Trainy, scale = TRUE, validation = "CV")
summary(pls.fit)
```

```
## Data:    X dimension: 388 17 
##  Y dimension: 388 1
## Fit method: kernelpls
## Number of components considered: 17
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV            3631     1607     1473     1279     1234     1154     1103
## adjCV         3631     1604     1470     1275     1222     1133     1094
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## CV        1082     1080     1080      1076      1075      1075      1074
## adjCV     1077     1076     1075      1071      1070      1070      1069
##        14 comps  15 comps  16 comps  17 comps
## CV         1073      1073      1073      1073
## adjCV      1068      1068      1068      1068
##
## TRAINING: % variance explained
##       1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps
## X       26.75    53.38    62.95    65.66    67.76    72.06    76.33    80.62
## Apps    81.18    84.72    88.85    90.58    92.08    92.36    92.46    92.50
##       9 comps  10 comps  11 comps  12 comps  13 comps  14 comps  15 comps
## X       82.68     84.69     87.67     90.74     92.67     95.26     96.89
## Apps    92.58     92.63     92.64     92.64     92.65     92.65     92.65
##       16 comps  17 comps
## X        99.13    100.00
## Apps     92.65     92.65
```

```r
pls.pred <- predict(pls.fit,College.Test,ncomp = 10)
#pcr.pred

mean((pls.pred - Apps.test)^2)
```

```
## [1] 33543289
```

#Problem 9g - It does not seem like we are able to accurately predict the number of college applications received based on the data provided. The MSE squared is quite large for the prediction vs test data. The test errors do seem similar between the 5 approaches.

##Problem 6-11

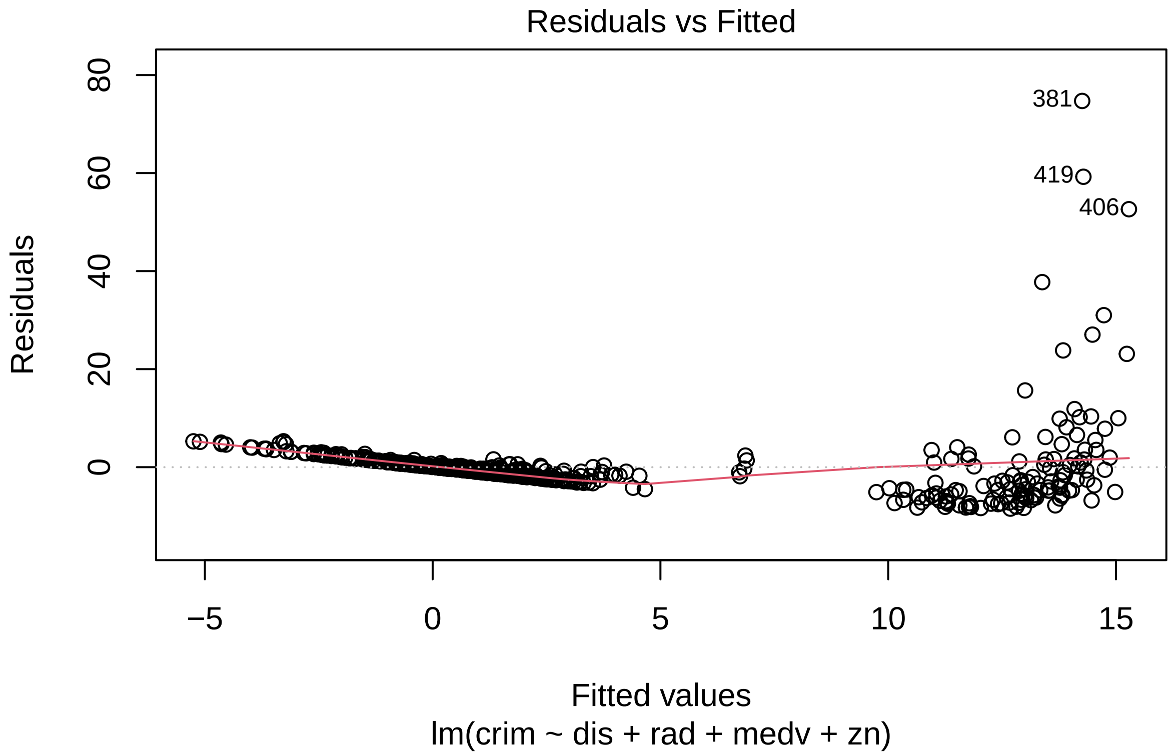
```r
#Problem 11a
#Linear
Boston.lm <- lm(crim ~ .,data = Boston)
summary(Boston.lm)
```

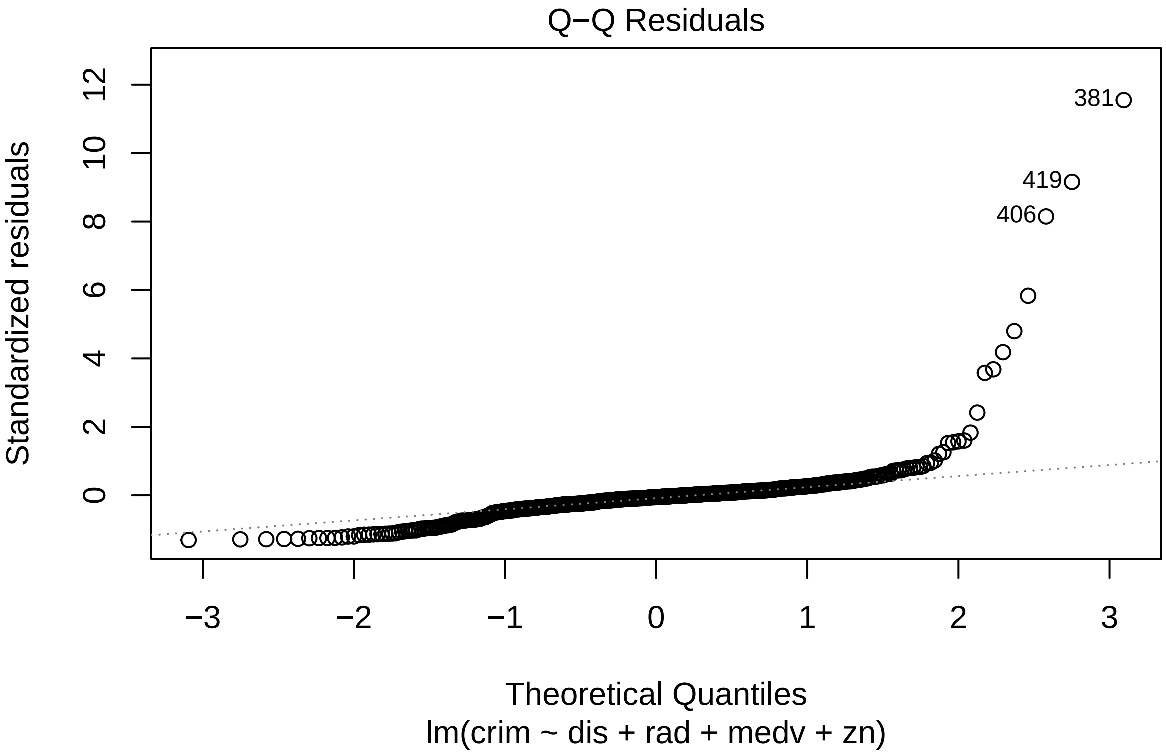
```
##
## Call:
## lm(formula = crim ~ ., data = Boston)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -8.534 -2.248 -0.348  1.087 73.923
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) 13.7783938  7.0818258   1.946 0.052271 .
## zn           0.0457100  0.0187903   2.433 0.015344 *
## indus       -0.0583501  0.0836351  -0.698 0.485709
## chas        -0.8253776  1.1833963  -0.697 0.485841
## nox         -9.9575865  5.2898242  -1.882 0.060370 .
## rm           0.6289107  0.6070924   1.036 0.300738
## age         -0.0008483  0.0179482  -0.047 0.962323
## dis         -1.0122467  0.2824676  -3.584 0.000373 ***
## rad          0.6124653  0.0875358   6.997 8.59e-12 ***
## tax         -0.0037756  0.0051723  -0.730 0.465757
## ptratio     -0.3040728  0.1863598  -1.632 0.103393
## lstat        0.1388006  0.0757213   1.833 0.067398 .
## medv        -0.2200564  0.0598240  -3.678 0.000261 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.46 on 493 degrees of freedom
## Multiple R-squared:  0.4493, Adjusted R-squared:  0.4359
## F-statistic: 33.52 on 12 and 493 DF,  p-value: < 2.2e-16
```

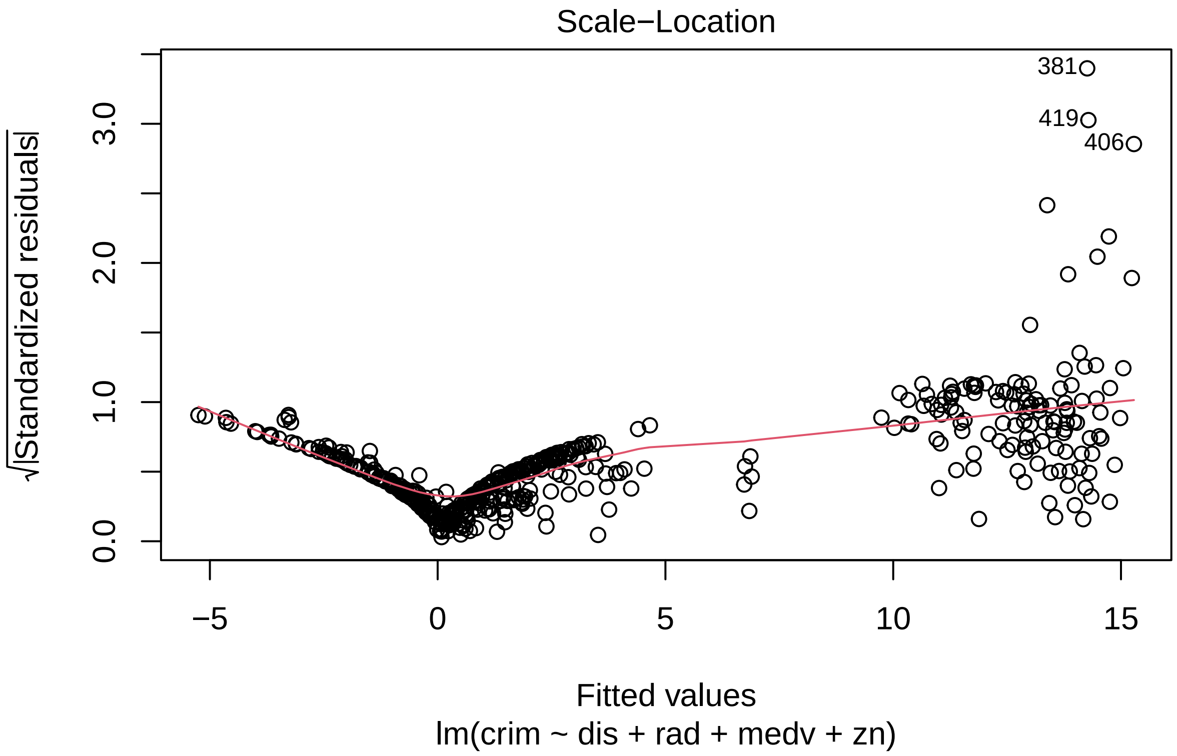
```
Boston.red.lm <- lm(crim ~ dis + rad + medv + zn, data = Boston)
summary(Boston.red.lm)
```

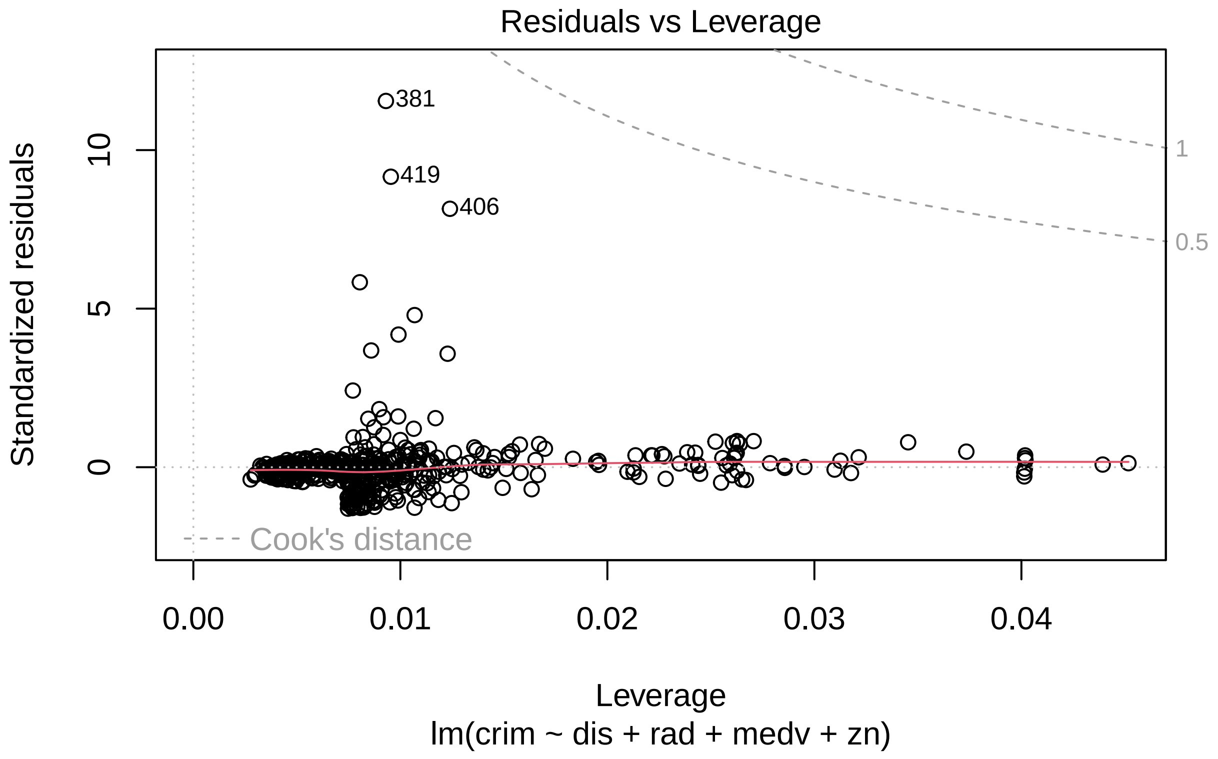
```
##
## Call:
## lm(formula = crim ~ dis + rad + medv + zn, data = Boston)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -8.459 -1.960 -0.331  0.857 74.718
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  5.26548    1.34674   3.910 0.000105 ***
## dis         -0.72291    0.20254  -3.569 0.000393 ***
## rad          0.50021    0.04044  12.370  < 2e-16 ***
## medv        -0.19122    0.03566  -5.362 1.26e-07 ***
## zn           0.05487    0.01735   3.163 0.001658 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.5 on 501 degrees of freedom
## Multiple R-squared:  0.4335, Adjusted R-squared:  0.429
## F-statistic: 95.84 on 4 and 501 DF,  p-value: < 2.2e-16
```

```
plot(Boston.red.lm)
```


Residuals vs Fitted
Residuals
80
60
40
20
0
−5
0
5
10
15
381
419
406
Fitted values
lm(crim ~ dis + rad + medv + zn)


Q−Q Residuals
Standardized residuals
0
2
4
6
8
10
12
−3
−2
−1
0
1
2
3
381
419
406
Theoretical Quantiles
lm(crim ~ dis + rad + medv + zn)


Scale−Location
√|Standardized residuals|
0.0
1.0
2.0
3.0
−5
0
5
10
15
381
419
406
Fitted values
lm(crim ~ dis + rad + medv + zn)



```r
#Ridge
Boston.x <- model.matrix(crim ~ .,Boston)[,-1]
Boston.y <- Boston$crim
Boston.lambda <- 10^seq(10, -2, length = 100)

Boston.train <- sample(1:nrow(Boston),nrow(Boston)/2)
Boston.test <- (-Boston.train)
Boston.ytest <- Boston.y[Boston.test]

Boston.ridge.mod <- glmnet(Boston.x,Boston.y,alpha = 0,lambda = Boston.lambda)
predict(Boston.ridge.mod,s=0, type = 'coefficients')
```

```
## 13 x 1 sparse Matrix of class "dgCMatrix"
##                         s1
## (Intercept) 13.4473681349
## zn           0.0452630596
## indus       -0.0605142483
## chas        -0.8223648009
## nox         -9.7788562633
## rm           0.6281672462
## age         -0.0008916473
## dis         -1.0038844850
## rad          0.6051810433
## tax         -0.0034139067
## ptratio     -0.2990248708
## lstat        0.1401134551
## medv        -0.2178870802
```

```
Boston.ridge.mod <- glmnet(Boston.x[Boston.train,], Boston.y[Boston.train], alpha = 0, lambda = Boston.l
Boston1.cv.out <- cv.glmnet(Boston.x[Boston.train,],Boston.y[Boston.train],alpha=0)
summary(Boston.ridge.mod)
```

```
##            Length Class     Mode
## a0          100   -none-    numeric
## beta       1200   dgCMatrix S4
## df          100   -none-    numeric
## dim           2   -none-    numeric
## lambda      100   -none-    numeric
## dev.ratio   100   -none-    numeric
## nulldev       1   -none-    numeric
## npasses       1   -none-    numeric
## jerr          1   -none-    numeric
## offset        1   -none-    logical
## call          5   -none-    call
## nobs          1   -none-    numeric
```

```
Boston1.bestlam <- Boston1.cv.out$lambda.min

Boston.ridge.pred <- predict(Boston.ridge.mod,s=Boston1.bestlam,newx = Boston.x[Boston.test,])
Boston.s.pred <- predict(Boston.lm,newdata = Boston[Boston.test,])
summary(Boston.ridge.pred)
```

```
##        s1
##  Min.   :-3.14605
##  1st Qu.: 0.08169
##  Median : 1.21295
##  Mean   : 3.36367
##  3rd Qu.: 3.82836
##  Max.   :14.84202
```

```
summary(Boston$crim)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##  0.00632  0.08204  0.25651  3.61352  3.67708 88.97620
```

```
mean((Boston.s.pred - Boston.ytest)^2)
```

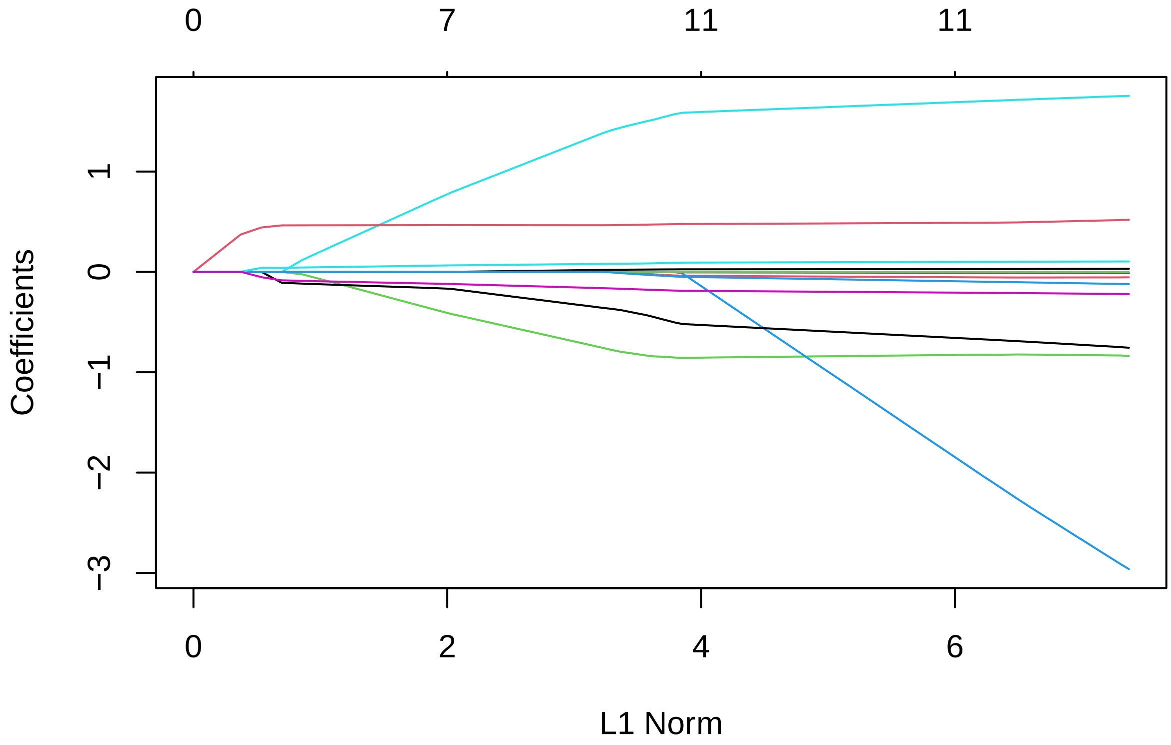
```
## [1] 47.13302
```

```
mean((Boston.ridge.pred - Boston.ytest)^2)
```

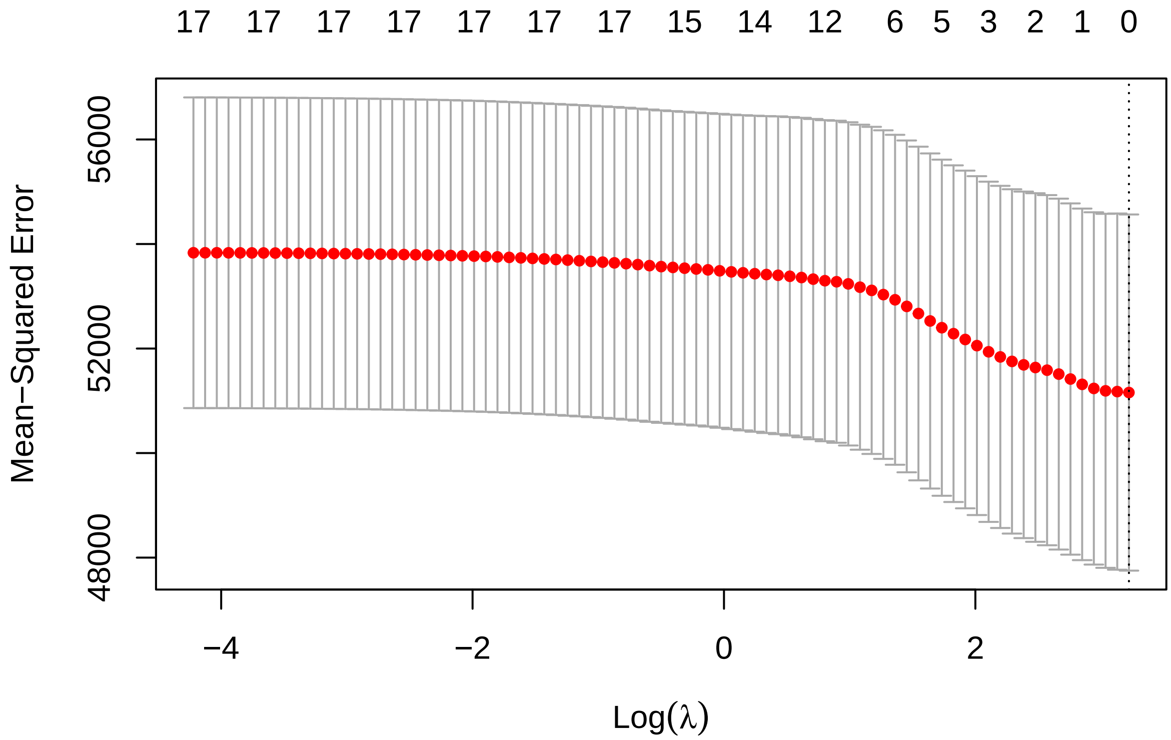
```
## [1] 50.61527
```

```
#Lasso
Boston.lasso.mod <- glmnet(Boston.x[Boston.train,],Boston.y[Boston.train],alpha = 1,lambda = grid)
plot(Boston.lasso.mod)
```

```
## Warning in regularize.values(x, y, ties, missing(ties), na.rm = na.rm):
## collapsing to unique 'x' values
```



```
Boston2.cv.out <- cv.glmnet(Boston.x[Boston.train,],Boston.y[Boston.train],alpha=1)
plot(cv.out)
```



```r
Boston2.bestlam <- Boston2.cv.out$lambda.min
bestlam
```

```
## [1] 25.07386
```

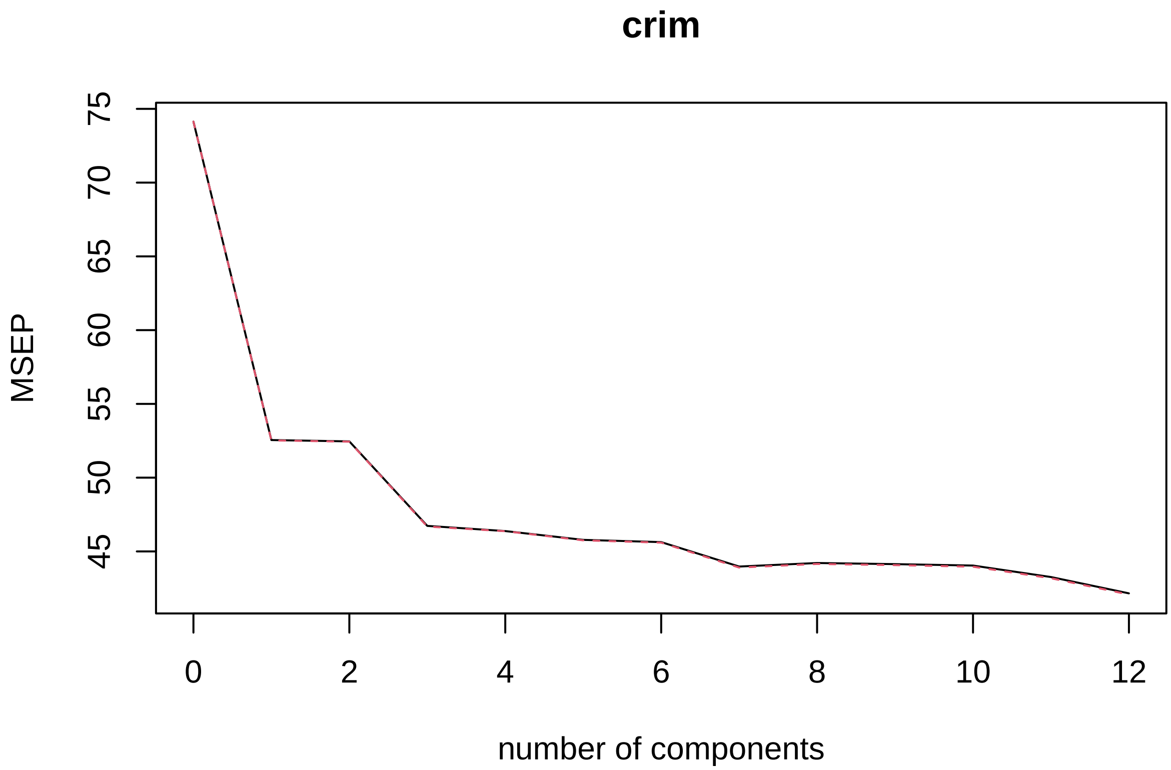
```r
Boston.lasso.pred <- predict(Boston.lasso.mod, s=Boston2.bestlam, newx = Boston.x[Boston.test,])
#Boston.lasso.pred

mean((Boston.lasso.pred - Boston.ytest)^2)
```

```
## [1] 50.00845
```

```r
#PCR
Boston.pcr.fit <- pcr(crim ~ .,data = Boston,scale = TRUE, validation = "CV")
summary(Boston.pcr.fit)
```

```
## Data:    X dimension: 506 12
##  Y dimension: 506 1
## Fit method: svdpc
## Number of components considered: 12
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV             8.61    7.249    7.243    6.836     6.81    6.766    6.755
## adjCV          8.61    7.247    7.241    6.833     6.81    6.764    6.752
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps
## CV       6.631    6.649    6.644     6.636     6.578     6.493
## adjCV    6.626    6.645    6.639     6.631     6.571     6.487
##
## TRAINING: % variance explained
##       1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps
## X       49.93    63.64    72.94    80.21    86.83    90.26    92.79    94.99
## crim    29.39    29.55    37.39    37.85    38.85    39.23    41.73    41.82
##       9 comps  10 comps  11 comps  12 comps
## X       96.78     98.33     99.48    100.00
## crim    42.12     42.43     43.58     44.93
```

```
validationplot(Boston.pcr.fit,val.type = "MSEP")
```



```
Boston.pcr.pred <- predict(Boston.pcr.fit,Boston.x[Boston.test,],ncomp = 12)
#Boston.pcr.pred

Boston.pcr.fit1 <- pcr(Boston.y ~ Boston.x,scale = TRUE,ncomp =12)
summary(Boston.pcr.fit1$fitted.values)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -8.0696 -0.2981  1.4915  3.6135  8.4130 17.7592
```

```
mean((Boston.pcr.pred - Boston.ytest)^2)
```

```
## [1] 47.13302
```

#Problem 11b - The PCR model seems to more accurately fit the data and is slightly improved from the linear model. The PCR model also better describes the data based on plots of the Boston crime rate per capita.

#Problem 11c - The PCR model does contain all components of the data set because that is where we achieve the most variability accounted for in Boston crime rate per capita.