Linear Model Selection and Regularization

Elsa Ferreira Gomes

2024/2025

Bias/Variance Tradeoff

Model's generalization error can be expressed as the sum of three very different errors:

Bias

- Wrong assumptions (e.g, assuming that the data is linear when it is actually quadratic).
- A high bias model is most likely to underfit the training data.

Variance

- Model's excessive sensitivity to small variations in the training data.
- A high variance model tends to overfit the training data.

Irreducible error

- Noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g.detect and remove outliers).

Increasing a model's complexity will typically **increase its variance and reduce its bias**. **Reducing a model's complexity increases its bias and reduces its variance**.

Regularization Linear Models

- There are two major problems related to training models: **overfitting and underfitting.**
  - Overfitting:
    - the model performs well on the training set but not so well on unseen (test) data.
  - Underfitting:
    - neither performs well on the train set nor on the test set.
- **Regularization** is implemented to avoid overfitting of the data, especially when there is a large variance between train and test set performances.
- With regularization, the number of features used in training is kept constant, yet the magnitude of the coefficients is reduced.
- There are different ways of reducing model complexity and preventing overfitting in linear models.
  - This includes **Ridge and Lasso Regression Models**.

Linear Regression

Let's recall the linear model:

- the model training essentially involves finding the appropriate values for coefficients.

$$Y = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$$

- We seek the values β_0, β_1,... β_p that minimize the Residual Sum of Squares (RSS):

$$RSS = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} \left(f(x_i) - \hat{f}(x_i)\right)^2 = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij}\right)^2$$

- RSS as Loss Function

- The **least squares** find the parameters β that **minimize** RSS given the data.

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

where $\bar{x} = \frac{1}{N} \sum_{i=1}^{n} (x_i)$ and $\bar{y} = \frac{1}{N} \sum_{i=1}^{n} (y_i)$

Least Squares - the need of alternatives

- Prediction Accuracy: especially when $p > n$, to control the variance.
- Model Interpretability: By removing irrelevant features (by setting the corresponding coefficient estimates to zero) we can obtain a model that is more easily interpreted.
- We will present some approaches for automatically performing feature selection

Classes of methods

- Subset Selection
  - Identify a subset of the p predictors (we believe to be related to the response).
  - Fit a model using least squares on the reduced set of variables.
- **Regularization**
  - Fit a model involving **all** p predictors, but the estimated coefficients are shrunken towards zero relative to the least squares estimates.
  - This shrinkage (or regularization) has the effect of reducing variance and can also perform variable selection.
- Dimension Reduction
  - Project the p predictors into a M-dimensional subspace, where $M < p$.
  - This is achieved by computing M different linear combinations, or projections, of the variables.
  - Then these M projections are used as predictors to fit a linear regression model by least squares.

Ridge Regression and the Lasso

Ridge regression and Lasso

• The subset selection methods use **least squares** to fit a linear model that contains a subset of the predictors.

• As an alternative, we can fit a model containing all p predictors using a technique that constrains or regularizes the coefficient estimates, or equivalently, that shrinks the coefficient estimates towards zero.

• It may not be immediately obvious why such a constraint should improve the fit, but it turns out that shrinking the coefficient estimates can significantly reduce their variance.

Ridge Regression — short introduction

(https://www.youtube.com/watch?v=lLlG5xkyqIA&t=15s)

- Is a variation of linear regression, specifically designed to address multicollinearity in the dataset.
- In linear regression, the goal is to find the best-fitting hyperplane that minimizes the sum of squared differences between the observed and predicted values.
  - However, when there are highly correlated variables, linear regression may become unstable and provide unreliable estimates.
  - Multicollinearity exists when two or more of the predictors in a regression model are moderately or highly correlated with one another.
- Ridge regression introduces a regularization term (L2 penalty) that penalizes large coefficients, helping to stabilize the model and prevent overfitting.

Ridge Regression

- The **Ridge Regression** estimates β^R are the values that minimize

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

where $\lambda >= 0$ is a **tuning parameter**, to be determined separately.

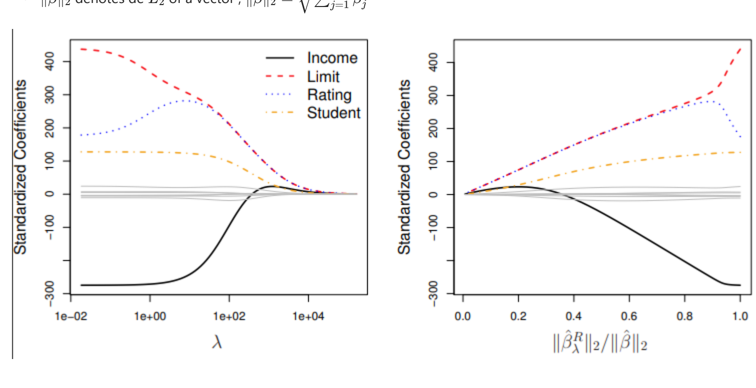
Ridge Regression

The objective of Ridge is to minimize the RSS & Square of cofficient

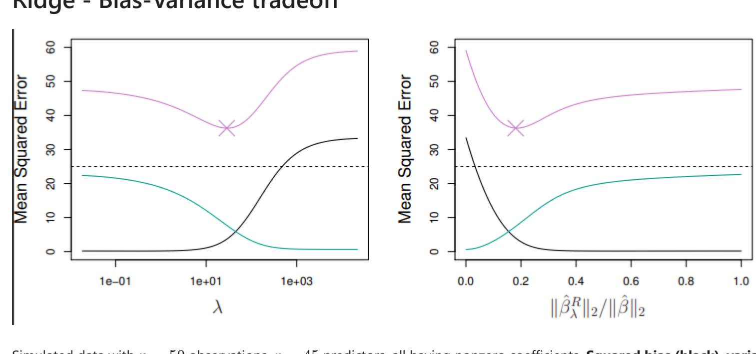
- The second term, $\lambda \sum_{j=1}^{p} \beta^2$ (shrinkage penalty), is small when $\beta_1, \ldots, \beta_p$ are close to zero,
  - it has the effect of shrinking the estimates of β_j towards zero.
- The tuning parameter λ serves to control the relative impact of these two terms on the regression coefficient estimates.
- When λ=0, the penalty has no effect, and ridge regression reduces to the ordinary least squares method.
- However, as λ→∞ the impact of the penalty grows, and the estimates of the coefficients β_j in ridge regression shrink towards zero.
- Selecting a good value for λ is very important
  - Cross-validation is used for this

Ridge Regression

- Left panel: each curve corresponds to the ridge regression coefficient estimate for one of the ten variables, plotted as a function of λ.
- Right panel: for the same ridge coefficient estimates, we now display $\frac{\|\beta^R_\lambda\|_2}{\|\beta\|_2}$, where β is the least squares coefficient estimates.
- $\|\beta\|_2$ denotes de L_2 of a vector , $\|\beta\|_2 = \sqrt{\sum_{j=1}^{p} \beta_j^2}$



Ridge - Bias-Variance tradeoff



Simulated data with $n = 50$ observations, $p = 45$ predictors, all having nonzero coefficients. **Squared bias (black)**, **variance (green)**, and **test mean squared error (purple)** for the ridge regression predictions on a simulated data set, as a function of λ and $\frac{\|\beta^R_\lambda\|_2}{\|\beta\|_2}$. The **horizontal dashed lines indicate the minimum possible MSE**. The **purple crosses** indicate the ridge regression models for which the **MSE is smallest**.

In [59]:
```python
# Import all libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

We will use the sklearn package to perform Ridge Regression and the Lasso.

The main functions in this package that we care about are Ridge(), which can be used to fit ridge regression models, and Lasso() which will fit lasso models. They also have cross-validated counterparts: RidgeCV() and LassoCV().

Before proceeding, let's prepare our data.

In [60]:
```python
df = pd.read_csv('Hitters.csv').dropna() ##.drop('Player', axis = 1)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 263 entries, 1 to 321
Data columns (total 20 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   AtBat      263 non-null    int64
 1   Hits       263 non-null    int64
 2   HmRun      263 non-null    int64
 3   Runs       263 non-null    int64
 4   RBI        263 non-null    int64
 5   Walks      263 non-null    int64
 6   Years      263 non-null    int64
 7   CAtBat     263 non-null    int64
 8   CHits      263 non-null    int64
 9   CHmRun     263 non-null    int64
 10  CRuns      263 non-null    int64
 11  CRBI       263 non-null    int64
 12  CWalks     263 non-null    int64
 13  League     263 non-null    object
 14  Division   263 non-null    object
 15  PutOuts    263 non-null    int64
 16  Assists    263 non-null    int64
 17  Errors     263 non-null    int64
 18  Salary     263 non-null    float64
 19  NewLeague  263 non-null    object
dtypes: float64(1), int64(16), object(3)
memory usage: 43.1+ KB
```

In [61]:
```python
df.head(5)
```

Out[61]:

	AtBat	Hits	HmRun	Runs	RBI	Walks	Years	CAtBat	CHits	CHmRun	CRuns	CRBI	CWalks	League	Division	PutOuts	Assists	Errors
1	315	81	7	24	38	39	14	3449	835	69	321	414	375	N	W	632	43	10
2	479	130	18	66	72	76	3	1624	457	63	224	266	263	A	W	880	82	14
3	496	141	20	65	78	37	11	5628	1575	225	828	838	354	N	E	200	11	3
4	321	87	10	39	42	30	2	396	101	12	48	46	33	N	E	805	40	4
5	594	169	4	74	51	35	11	4408	1133	19	501	336	194	A	W	282	421	25

In [62]:
```python
# Null values
df.isnull().sum()*100/df.shape[0]
```

Out[62]:
```
AtBat        0.0
Hits         0.0
HmRun        0.0
Runs         0.0
RBI          0.0
Walks        0.0
Years        0.0
CAtBat       0.0
CHits        0.0
CHmRun       0.0
CRuns        0.0
CRBI         0.0
CWalks       0.0
League       0.0
Division     0.0
PutOuts      0.0
Assists      0.0
Errors       0.0
Salary       0.0
NewLeague    0.0
dtype: float64
```

In [63]:
```python
y = df.Salary

# Drop the column with the independent variable (Salary), and columns for which we created dummy variables
X_ = df.drop(['Salary', 'League', 'Division', 'NewLeague'], axis = 1).astype('float64')

# Define the feature set X.
X = pd.concat([X_, dummies[['League_N', 'Division_W', 'NewLeague_N']]], axis = 1)

X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 263 entries, 1 to 321
Data columns (total 19 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   AtBat        263 non-null    float64
 1   Hits         263 non-null    float64
 2   HmRun        263 non-null    float64
 3   Runs         263 non-null    float64
 4   RBI          263 non-null    float64
 5   Walks        263 non-null    float64
 6   Years        263 non-null    float64
 7   CAtBat       263 non-null    float64
 8   CHits        263 non-null    float64
 9   CHmRun       263 non-null    float64
 10  CRuns        263 non-null    float64
 11  CRBI         263 non-null    float64
 12  CWalks       263 non-null    float64
 13  PutOuts      263 non-null    float64
 14  Assists      263 non-null    float64
 15  Errors       263 non-null    float64
 16  League_N     263 non-null    bool
 17  Division_W   263 non-null    bool
 18  NewLeague_N  263 non-null    bool
dtypes: bool(3), float64(16)
memory usage: 35.7 KB
```

In [64]:
```python
from sklearn.preprocessing import LabelEncoder,StandardScaler
labelencode = LabelEncoder()

categ = ['Salary', 'League', 'Division', 'NewLeague']

# Encode Categorical Columns
le = LabelEncoder()
df[categ] = df[categ].apply(le.fit_transform)

y=labelencode.fit_transform(y)
```

In [65]:
```python
df.head(5)
```

Out[65]:

	AtBat	Hits	HmRun	Runs	RBI	Walks	Years	CAtBat	CHits	CHmRun	CRuns	CRBI	CWalks	League	Division	PutOuts	Assists	Errors
1	315	81	7	24	38	39	14	3449	835	69	321	414	375	1	1	632	43	10
2	479	130	18	66	72	76	3	1624	457	63	224	266	263	0	1	880	82	14
3	496	141	20	65	78	37	11	5628	1575	225	828	838	354	1	0	200	11	3
4	321	87	10	39	42	30	2	396	101	12	48	46	33	1	0	805	40	4
5	594	169	4	74	51	35	11	4408	1133	19	501	336	194	0	1	282	421	25

In [ ]:

In [66]:
```python
#applying standard scaler

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [67]:
```python
###MINMAX
#from sklearn.preprocessing import minmax_scale
#X_train=minmax_scale(X_train, axis=0)
#X_test=sc.transform(X_test)
```

In [68]:
```python
# Split data into training and test sets
X_train, X_test , y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)
```

The Ridge() function has an alpha argument (λ, with a different name) that is used to tune the model. We'll generate an array of alpha values ranging from very big to very small, essentially covering the full range of scenarios from the null model containing only the intercept, to the least squares fit:

In [69]:
```python
alphas = 10**np.linspace(10,-2,100)*0.5
alphas
```

Out[69]:
```
array([5.00000000e+09, 3.78231664e+09, 2.86118383e+09, 2.16438064e+09,
       1.63727458e+09, 1.23853818e+09, 9.36908711e+08, 7.08737081e+08,
       5.36133611e+08, 4.05565415e+08, 3.06795364e+08, 2.32079442e+08,
       1.75559587e+08, 1.32804389e+08, 1.00461650e+08, 7.59955541e+07,
       5.74878498e+07, 4.34874501e+07, 3.28966612e+07, 2.48851178e+07,
       1.88246790e+07, 1.42401793e+07, 1.07721735e+07, 8.14875417e+06,
       6.16423370e+06, 4.66301673e+06, 3.52740116e+06, 2.66834962e+06,
       2.01850863e+06, 1.52692775e+06, 1.15506485e+06, 8.73764200e+05,
       6.60970574e+05, 5.00000000e+05, 3.78231664e+05, 2.86118383e+05,
       2.16438064e+05, 1.63727458e+05, 1.23853818e+05, 9.36908711e+04,
       7.08737081e+04, 5.36133611e+04, 4.05565415e+04, 3.06795364e+04,
       2.32079442e+04, 1.75559587e+04, 1.32804389e+04, 1.00461650e+04,
       7.59955541e+03, 5.74878498e+03, 4.34874501e+03, 3.28966612e+03,
       2.48851178e+03, 1.88246790e+03, 1.42401793e+03, 1.07721735e+03,
       8.14875417e+02, 6.16423370e+02, 4.66301673e+02, 3.52740116e+02,
       2.66834962e+02, 2.01850863e+02, 1.52692775e+02, 1.15506485e+02,
       8.73764200e+01, 6.60970574e+01, 5.00000000e+01, 3.78231664e+01,
       2.86118383e+01, 2.16438064e+01, 1.63727458e+01, 1.23853818e+01,
       9.36908711e+00, 7.08737081e+00, 5.36133611e+00, 4.05565415e+00,
       3.06795364e+00, 2.32079442e+00, 1.75559587e+00, 1.32804389e+00,
       1.00461650e+00, 7.59955541e-01, 5.74878498e-01, 4.34874501e-01,
       3.28966612e-01, 2.48851178e-01, 1.88246790e-01, 1.42401793e-01,
       1.07721735e-01, 8.14875417e-02, 6.16423370e-02, 4.66301673e-02,
       3.52740116e-02, 2.66834962e-02, 2.01850863e-02, 1.52692775e-02,
       1.15506485e-02, 8.73764200e-03, 6.60970574e-03, 5.00000000e-03])
```

Associated with each alpha value is a vector of ridge regression coefficients, which we'll store in a matrix coefs. Remember that we'll want to standardize the variables so that they are on the same scale.

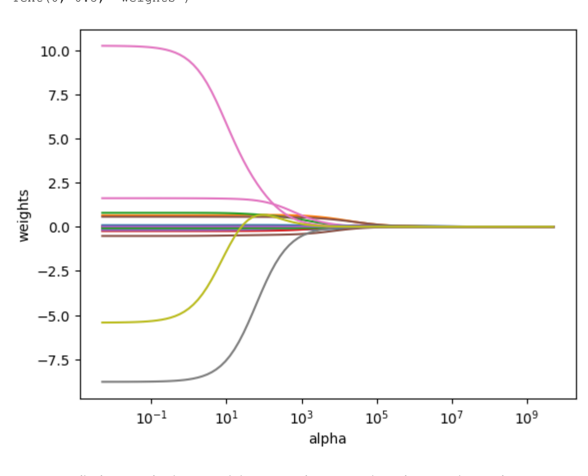
In [70]:
```python
from sklearn import linear_model

ridge = Ridge()
coefs = []

for a in alphas:
    ridge.set_params(alpha = a)
    ridge.fit(X, y)
    coefs.append(ridge.coef_)

np.shape(coefs)
```

Out[70]:
```
(100, 19)
```

We expect the coefficient estimates to be much smaller, in terms of l2 norm, when a large value of alpha is used, as compared to when a small value of alpha is used. Let's plot and find out:

In [71]:
```python
ax = plt.gca()
ax.plot(alphas, coefs)
ax.set_xscale('log')
plt.axis('tight')
plt.xlabel('alpha')
plt.ylabel('weights')
```

Out[71]:
```
Text(0, 0.5, 'weights')
```



We now split the samples into a training set and a test set in order to estimate the test error of ridge regression and the lasso:

Next we fit a ridge regression model on the training set, and evaluate its MSE on the test set, using λ=4:

In [72]:
```python
ridge2 = Ridge(alpha = 4)
ridge2.fit(X_train, y_train)             # Fit a ridge regression on the training data
pred2 = ridge2.predict(X_test)           # Use this model to predict the test data
print(pd.Series(ridge2.coef_)) ##, index = X.columns)) # Print coefficients
print(mean_squared_error(y_test, pred2))          # Calculate the test MSE
```

```
0    -0.032993
1     0.272655
2     0.118852
3    -0.077358
4     0.018518
5     0.506066
6     3.894261
7    -0.016972
8     0.060893
9     0.087340
10    0.026635
11   -0.024989
12   -0.019892
13    0.022592
14    0.003583
15   -0.371551
16    6.032131
17   -7.646746
18    0.742740
dtype: float64
971.3283255686893
```

The test MSE when alpha = 4 is 115325 Now let´s see what happens if we use a huge value of alpha, say 10^10:

In [73]:
```python
ridge3 = Ridge(alpha = 10**10)
ridge3.fit(X_train, y_train)             # Fit a ridge regression on the training data
pred3 = ridge3.predict(X_test)           # Use this model to predict the test data
print(pd.Series(ridge3.coef_)) # Print coefficients
print(mean_squared_error(y_test, pred3))          # Calculate the test MSE
```

```
0     3.598662e-05
1     1.177923e-05
2     1.539706e-06
3     6.319652e-06
4     7.135167e-06
5     6.203672e-06
6     1.271598e-06
7     7.077145e-04
8     2.073445e-04
9     2.536652e-05
10    1.044246e-04
11    1.110556e-04
12    7.314638e-05
13    6.099547e-05
14   -7.630394e-07
15   -9.655496e-09
16    1.243447e-09
17   -2.183971e-08
18   -1.139353e-09
dtype: float64
1711.151911660474
```

This big penalty shrinks the coefficient. This over-shrinking makes the model more biased, resulting in a higher MSE.

Okay, so fitting a ridge regression model with alpha = 4 leads to a much lower test MSE than fitting a model with just an intercept. We now check whether there is any benefit to performing ridge regression with alpha = 4 instead of just performing least squares regression. Recall that least squares is simply ridge regression with alpha = 0.

In [74]:
```python
ridge2 = Ridge(alpha = 0)
ridge2.fit(X_train, y_train)             # Fit a ridge regression on the training data
pred = ridge2.predict(X_test)            # Use this model to predict the test data
print(pd.Series(ridge2.coef_, index = X.columns)) # Print coefficients
print(mean_squared_error(y_test, pred))           # Calculate the test MSE
```

```
AtBat         -0.030098
Hits           0.265894
HmRun          0.158683
Runs          -0.068112
RBI           -0.004406
Walks          0.504743
Years          3.983040
CAtBat        -0.017938
CHits          0.063943
CHmRun         0.092800
CRuns          0.025747
CRBI          -0.025564
CWalks        -0.019768
PutOuts        0.022587
Assists        0.004459
Errors        -0.403721
League_N      10.132821
Division_W    -8.710705
NewLeague_N   -2.680650
dtype: float64
972.84444613292
```

Instead of arbitrarily choosing alpha's, we use cross-validation to choose the tuning parameter alpha. We can do this using the cross-validated ridge regression function, RidgeCV(). By default, the function performs generalized cross-validation (an efficient form of LOOCV), though this can be changed using the argument cv.

In [75]:
```python
from sklearn.linear_model import RidgeCV

ridgecv = RidgeCV(alphas = alphas, scoring = 'neg_mean_squared_error')
ridgecv.fit(X_train, y_train)
ridgecv.alpha_
```

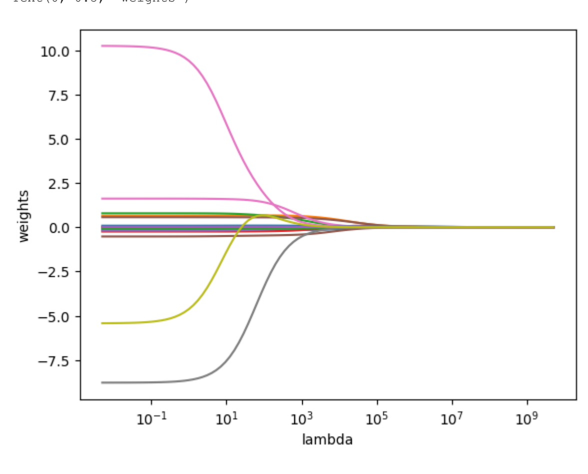
Out[75]:
```
2018508.6292982749
```

In [76]:
```python
ridge4 = Ridge(alpha = ridgecv.alpha_)
ridge4.fit(X_train, y_train)
mean_squared_error(y_test, ridge4.predict(X_test))
```

Out[76]:
```
1037.929992612727
```

In [78]:
```python
ax = plt.gca()
ax.plot(alphas, coefs)
ax.set_xscale('log')
plt.axis('tight')
#plt.xlabel('alpha')
plt.xlabel('lambda')
plt.ylabel('weights')
```

Out[78]:
```
Text(0, 0.5, 'weights')
```



Lasso

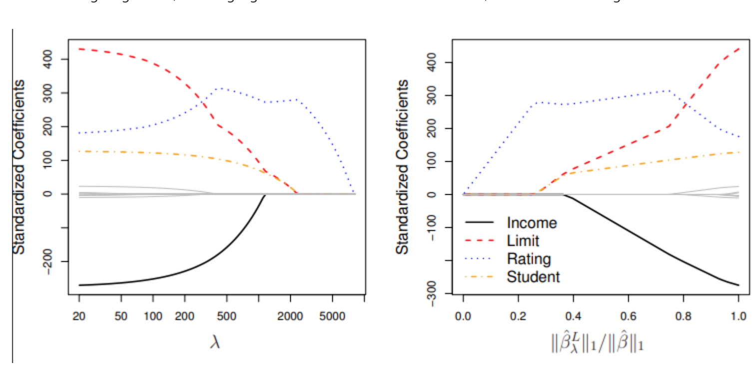
(https://www.youtube.com/watch?v=0tfPuddPhEY)

- Ridge regression disadvantage:unlike subset selection, which will generally select models that involve just a subset of the variables, ridge regression will include all p predictors in the final mode
  - The **Lasso** overcomes this disadvantage
- The **Lasso** estimates β^R_λ are the values that minimize

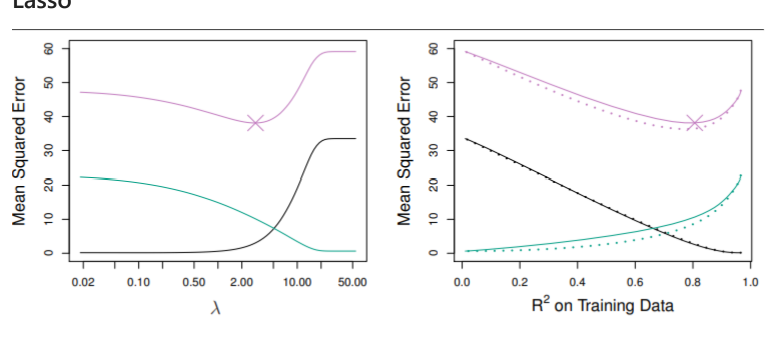
$$\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij}\right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| = RSS + \lambda \sum_{j=1}^{p} |\beta_j|$$

where $\lambda >= 0$ is a **tuning parameter**, to be determined separately.

- Uses L1 norm of a coefficient vector β is given by $\|\beta\|_1 = \sum |\beta_j|$
  - forcing some of the coefficient estimates to be exactly equal to zero when the tuning parameter λ is sufficiently large
- Shrinks the coefficient estimates towards zero
  - Hence, much like best subset selection, the lasso performs variable selection
- Yields sparse models (models that involve only a subset of the variables)
- As in ridge regression, selecting a good value of λ for the lasso is critical; cross-validation is again the method of choice



Lasso

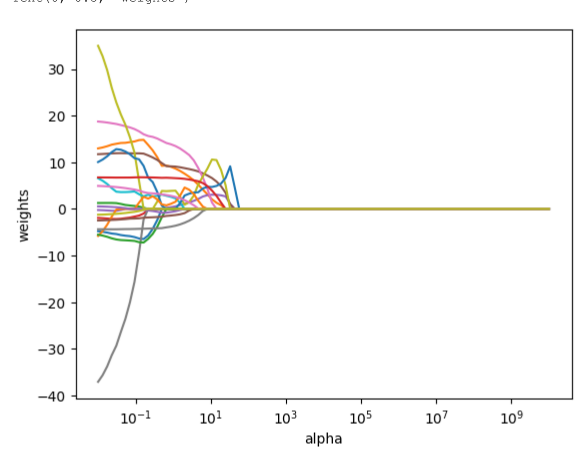


Left: Plots of squared bias (black), variance (green), and test MSE (purple) for the lasso on simulated (dataset of Bias-covarience tradeoff). Right: Comparison of squared bias, variance and test MSE between lasso (solid) and ridge (dashed). Both are plotted against their R^2 on the training data, as a common form of indexing. The crosses in both plots indicate the lasso model for which the MSE is smallest.

We saw an example that ridge regression with a wise choice of alpha can outperform least squares as well as the null model on the Hitters data set. We now ask whether the lasso can yield either a more accurate or a more interpretable model than ridge regression. In order to fit a lasso model, we'll use the Lasso() function; however, this time we'll need to include the argument max_iter = 10000. Other than that change, we proceed just as we did in fitting a ridge model:

In [79]:
```python
from sklearn.linear_model import Lasso, LassoCV
from sklearn.preprocessing import scale

lasso = Lasso(max_iter = 10000)
coefs = []

for a in alphas:
    lasso.set_params(alpha=a)
    lasso.fit(scale(X_train), y_train)
    coefs.append(lasso.coef_)

ax = plt.gca()
ax.plot(alphas*2, coefs)
ax.set_xscale('log')
plt.axis('tight')
plt.xlabel('alpha')
plt.ylabel('weights')
```

Out[79]:
```
Text(0, 0.5, 'weights')
```



Notice that in the coefficient plot that depending on the choice of tuning parameter, some of the coefficients are exactly equal to zero. We now perform 10-fold cross-validation to choose the best alpha, refit the model, and compute the associated test error:

In [80]:
```python
lassocv = LassoCV(alphas = None, cv = 10, max_iter = 100000)
lassocv.fit(X_train, y_train)

lasso.set_params(alpha=lassocv.alpha_)
lasso.fit(X_train, y_train)
mean_squared_error(y_test, lasso.predict(X_test))
```

Out[80]:
```
1028.2334655561267
```

In [81]:
```python
from sklearn.linear_model import LassoCV

alphas1 = np.random.randint(0,1000,100)
alphas2 = 10**np.linspace(10,-2,100)*0.5
alphas3 = np.linspace(0,1,1000)
lasso_cv_model = LassoCV(alphas = alphas1, cv = 10).fit(scale(X_train), y_train)
```

This is substantially lower than the test set MSE of the null model and of least squares, and only a little worse than the test MSE of ridge regression with alpha chosen by cross-validation.

However, the lasso has a substantial advantage over ridge regression in that the resulting coefficient estimates are sparse. Here we see that 13 of the 19 coefficient estimates are exactly zero:

In [82]:
```python
# Some of the coefficients are now reduced to exactly zero.
pd.Series(lasso.coef_, index=X.columns)
```

Out[82]:
```
AtBat          0.043116
Hits           0.000000
HmRun          0.000000
Runs           0.000000
RBI            0.000000
Walks          0.000000
Years          0.000000
CAtBat         0.009709
CHits          0.000000
CHmRun         0.000000
CRuns          0.000000
CRBI           0.000000
CWalks         0.000000
PutOuts        0.028185
Assists       -0.000000
Errors        -0.000000
League_N       0.000000
Division_W    -0.000000
NewLeague_N    0.000000
dtype: float64
```

Elastic net regularization

The elastic net is a regularized regression method that linearly combines the L1 and L2 penalties of the Lasso and Ridge methods.

(https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html)

Ridge, lasso, and elastic net can be efciently fit along a sequence of λ values, creating what is known as a solution path or regularization path.

- Hence there is specialized code to ft such paths, and to choose a suitable value of λ using cross-validation.
- Even with identical splits the results will not agree exactly with our grid above because the standardization of each feature in grid is carried out on each fold, while in pipeCV below it is carried out only once.
- Nevertheless, the results are similar as the normalization is relatively stable across folds

Ref: https://hastie.su.domains/ISLP/ISLP_website.pdf - Chp6