CS4619: Artificial Intelligence 2

Model Complexity

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [1]: %load_ext autoreload
   %autoreload 2
   %matplotlib inline

In [2]: import pandas as pd
   import numpy as np
   import matplotlib.pyplot as plt

Model Complexity

In the previous lecture, we discussed what to do when your estimator is under-performing:

- Add more training data
- Add more features
- Change the complexity of the model
- Change the class of model, e.g., to a more complex one

In this lecture, we discuss the third of these: changing the complexity of the model.

- Increasing the complexity may avoid underfitting at the risk of overfitting
- Decreasing the complexity may avoid overfitting at the risk of underfitting

Changing the Complexity of kNN

You change the complexity of kNN by changing the value of $k$.

Here's a graph of the MSE of distance-weighted kNN against different values of $k$ for the artificial dataset from the previous lecture. Note the horizontal axis.
```python
from sklearn.neighbors import KNeighborsRegressor
from sklearn.cross_validation import train_test_split
from sklearn.metrics import mean_squared_error

def inv_distances(dist):
    return 1 / (0.0001 + dist)

def make_dataset(m, func, error):
    X = np.random.random(m)
    y = func(X, error)
    return X.reshape(m, 1), y

def f(x, error = 1.0):
    y = 10 - 1 / (x + 0.1)
    if error > 0:
        y = np.random.normal(y, error)
    return y

X, y = make_dataset(50, f, 1.0)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = np.random)

kvals = range(10, 0, -1)
mses_train = []
mses_test = []

for k in kvals:
    estimator = KNeighborsRegressor(n_neighbors = k, weights = inv_distances)
    estimator.fit(X_train, y_train)
    mses_train.append(mean_squared_error(estimator.predict(X_train), y_train))
    mses_test.append(mean_squared_error(estimator.predict(X_test), y_test))

fig = plt.figure()
plt.xlabel("k")
plt.ylabel("MSE")
kvalsrev = [11 - kval for kval in kvals]
plt.xticks(kvalsrev, kvals)
plt.plot(kvalsrev, mses_train, label = 'training error', color = 'red')
plt.plot(kvalsrev, mses_test, label='test error', color = 'gold')
plt.legend()
plt.show()
```
Perhaps counter-intuitively,

- When $k = 1$ or close to 1, we have a more complex model and hence a risk of overfitting. When $k$ is 'too small' noisy examples have a greater impact.
- When $k$ is large, we have a less complex model. The effect of noisy examples is reduced: predictions are 'smoothed' across the $y$-values of several neighbours. But if $k$ is 'too large', there is a risk of underfitting: we are not exploiting the structure in the data; we are allowing distant examples to influence the predictions.

In the previous lecture, we had a plot with the degree of a polynomial on the horizontal axis: larger degree meant greater complexity. But for kNN, smaller $k$ means higher complexity. Hence, for consistency with the plot in the previous lecture (where complexity increased along the horizontal axis), in the plot above we show decreasing $k$ (so that again complexity increases along the horizontal axis).

Luckily, of course, we have a way of choosing a good value for $k$: see the lecture on model selection.

**Regularization: Changing the Complexity of Linear Regression**
Linear models are among the least complex models. Hence, we normally associate them with underfitting. But, even linear regression might overfit the training data. This can happen when you have ‘too many’ features. It can especially happen when you have a lot of features and too little training data. (The ‘curse of dimensionality’ again.) Problems also arise when two or more features are not independent of each other.

One solution to this problem is to discard some of the features — those we think are not predictive of the target outcome. We can do this manually, in conjunction with our domain expert. Or we can use a feature selection algorithm — we study these later in the module. But there is an alternative solution.

The alternative solution is regularization. In at least one of its versions, it keeps all the features — and so it's a good option when you think that each feature probably does contribute something to the prediction. Regularization avoids overfitting by adding more bias. It is done by penalizing the hypotheses that fit the data too well.

Recall that OLS linear regression finds coefficients $\beta$ that minimize

$$J_\beta(X, y) = \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^i) - y^{(i)})^2$$

Regularization imposes a penalty on the size of the coefficients.

Before we look at the details, one more observation. It is conventional to scale all the features in advance so that they have the same range or the same mean and standard deviation: you may find different coefficients on scaled and unscaled data and the error in the case of scaled data is supposed to be lower. (In my experience, it often isn't, and I don't know why!) We'll discuss how to do scaling in detail in later lectures, so we won't do any scaling below..

**Lasso Regression: Using the $l_1$-norm**
Lasso stands for 'least absolute shrinkage and selection operator' — but this doesn't matter to us at all. What does matter is the following.

In Lasso Regression, we penalize by the $l_1$-norm, which is simply the sum of the absolute value of the coefficients $\beta$, i.e. $\sum_{j=1}^{n} |\beta_j|$. So Lasso Regression finds the $\beta$ that minimizes

$$J_\beta(X, y) = \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} |\beta_j|$$

(Minor point: we don’t penalize $\beta_0$, which is why $j$ starts at 1.)

$\lambda$ is called the 'regularization parameter'. It controls how much penalization we want and this determines the balance between the two parts of the modified loss function: fitting the data versus shrinking the parameters.

- As $\lambda \to 0$, Lasso Regression gets closer to being OLS Linear Regression. When $\lambda = 0$, Lasso Regression is the same as OLS Linear Regression.
- When $\lambda \to \infty$, penalties are so great that all the coefficients will tend to zero: the only way to minimize the loss function will be to make the coefficients as small as possible. It’s likely that in this case we will underfit the data.

So, for regularization to work well, we must choose the value of $\lambda$ carefully.

- So what kind of thing is $\lambda$?
- How do we choose its value?

An important observation about Lasso Regression is that: as $\lambda$ grows, some of the $\beta_j$ will be driven to zero. This means that the model that it learns treats some features as irrelevant. Hence, it performs some feature selection too. Compare this with Ridge Regression below.

**Ridge Regression: Using the $l_2$-norm**

In Ridge Regression, we penalize by the $l_2$-norm, which is simply the sum of the squares of the coefficients, i.e. $\sum_{j=1}^{n} \beta_j^2$. (Strictly speaking, the $l_2$-norm is the square root of the sum of squares.) So Ridge Regression finds the $\beta$ that minimizes

$$J_\beta(X, y) = \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \beta_j^2$$

Both Lasso and Ridge Regression shrink the values of the coefficients but, as we mentioned, Lasso Regression may additionally result in coefficients being set to zero. This does not happen with Ridge Regression. Optionally, consult section 3.4.3 of *The Elements of Statistical Learning* by Hastie, Friedman & Tibshirani (available online) for an explanation. One observation from the book is that, roughly speaking, Lasso Regression shrinks the coefficients by approximately the same constant amount (unless they are so small that they get shrunk to zero), whereas, again roughly speaking, Ridge Regression shrinks the coefficients by approximately the same proportion.

**Fitting Regularized Linear Regression Models**
We looked at two ways of fitting an OLS Linear Regression model to a training set. One was to set the partial derivative of the loss function to zero and solve for $\beta$, giving the normal equations. The other was to use gradient descent to search for the $\beta$ that minimizes the loss function, where we changed the values of $\beta$ on each iteration by an amount related to the partial derivative of the loss function.

**Lasso Regression:**
- Neither of these methods is available to use for Lasso Regression. Why?
- scikit-learn's main implementation uses an approach called 'coordinate descent'.

**Ridge Regression:**
Both approaches are available to us. The details don't matter. But in case you're interested, here are some of them:
- If we take the direct approach, we get
  \[
  \beta = (X^TX + \lambda I)^{-1}X^Ty
  \]
  $I$ is the $(n + 1)$ identity matrix, i.e. all zeros except for the main diagonal which is all ones. (In fact, for consistency with what we were doing above, where we chose not to penalize $\beta_0$, you want a zero in the top left, so this is not really the identity matrix.)

  Interestingly, you also don't need to implement this with the pseudo-inverse. It's possible to prove that, provided $\lambda > 0$, then $X^TX + \lambda I$ will be invertible.
- If we use gradient descent, then the update rule for $\beta_j$ for all $j$ except $j = 0$ becomes:
  \[
  \beta_j \leftarrow \beta_j - \alpha \left(\frac{1}{m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)}) \times x_j^{(i)} + \frac{\lambda}{m} \beta_j\right)
  \]
  We can re-arrange this to:
  \[
  \beta_j \leftarrow \beta_j (1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)}) \times x_j^{(i)}
  \]
  which helps to show why this shrinks $\beta_j$.

**Comparing Lasso and Ridge Regression**

We plot the artificial dataset and the linear model found by OLS Linear Regression, Lasso Regression and Ridge Regression. I have shown the MSE, the intercept and the coefficient (only one coefficient because, in this artificial dataset, there was only one feature) above each plot.

A couple of notes about the code:
- Everyone uses $\lambda$ for the hyperparameter. Regrettably, scikit-learn calls it alpha, presumably because lambda is a Python reserved word.
- Also, as discussed above, there is a direct method for finding the coefficients for Ridge Regression; but for Lasso Regression a search is used. The code below simply accepts scikit-learn's defaults for the search, such as the maximum number of iterations.

You can play with the hyperparameters to see how they affect the fit.
```python
import matplotlib.gridspec as gridspec
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge

lasso_lambda = 1.0
ridge_lambda = 1.0

ols = LinearRegression()
ols.fit(X_train, y_train)
y_predicted_ols = ols.predict(X_test)
 mse_ols = mean_squared_error(y_predicted_ols, y_test)

lasso = Lasso(alpha = lasso_lambda)
lasso.fit(X_train, y_train)
y_predicted_lasso = lasso.predict(X_test)
mse_lasso = mean_squared_error(y_predicted_lasso, y_test)

ridge = Ridge(alpha = ridge_lambda)
ridge.fit(X_train, y_train)
y_predicted_ridge = ridge.predict(X_test)
mse_ridge = mean_squared_error(y_predicted_ridge, y_test)

# Set up the two subplots
fig = plt.figure(figsize=(14, 4.5))
gs = gridspec.GridSpec(1, 3)
# Leftmost diagram: OLS
ax0 = plt.subplot(gs[0])
plt.title("OLS Linear Regression
MSE: %.3f
Intercept: %.3f
Coefficient: %.3f" % (mse_ols, ols.intercept_, ols.coef_[0]))
plt.xlabel("Feature")
plt.ylabel("MSE")
plt.ylim(-4, 14)
ax0.scatter(X_train.ravel(), y_train, color = 'green')
ax0.plot(X_test.ravel(), y_predicted_ols, color = 'blue')
# Middle diagram: Lasso
ax1 = plt.subplot(gs[1])
plt.title("Lasso Regression
MSE: %.3f
Intercept: %.3f
Coefficient: %.3f" % (mse_lasso, lasso.intercept_, lasso.coef_[0]))
plt.xlabel("Feature")
plt.ylabel("MSE")
plt.ylim(-4, 14)
ax1.scatter(X_train.ravel(), y_train, color = 'green')
ax1.plot(X_test.ravel(), y_predicted_lasso, color = 'blue')
# Rightmost diagram: Ridge
ax2 = plt.subplot(gs[2])
plt.title("Ridge Regression
MSE: %.3f
Intercept: %.3f
Coefficient: %.3f" % (mse_ridge, ridge.intercept_, ridge.coef_[0]))
plt.xlabel("Feature")
plt.ylabel("MSE")
plt.ylim(-4, 14)
ax2.scatter(X_train.ravel(), y_train, color = 'green')
ax2.plot(X_test.ravel(), y_predicted_ridge, color = 'blue')

fig.tight_layout()
plt.show()
```
You can see the way the regularization shrinks $\beta$, making the line flatter. This isn't having a good effect on MSE in this example — but that's because it's supposed to help with overfitting caused by 'too many' features, whereas linear regression in fact underfits this training set and we have only one feature!

So, which should we use in general? As we've discussed, you think about using Lasso and Ridge Regression when you have lots of features causing OLS Linear Regression to overfit. If a small number of the features are relevant (likely to have large coefficients) and the rest are irrelevant (likely to have coefficients that are zero or close to zero), then you might prefer Lasso Regression. If most of the features are predictive to some degree, especially to roughly the same degree, you might use Ridge Regression. Of course, you're unlikely to know this in advance. So you must error estimation to choose between them.

The final question now is how to set the hyperparameters.

**Model Selection for Lasso and Ridge Regression**

There is a variety of methods for selecting the best value for $\lambda$. The obvious one is cross-validation. scikit-learn offers some handy classes that can make the code for this quite compact. Here it is for Lasso Regression. For Ridge Regression, it is similar.

We illustrate for the CorkA dataset. Bear in mind that the code should be improved by scaling the data first — see the later lecture where we learn how to do this in a pipeline.
In [6]:
from sklearn.linear_model import LassoCV

df = pd.read_csv("dataset-corkA.csv")
X = df[['flarea', 'bdrms', 'bthrms']].values
y = df['price'].values

lassocv = LassoCV(cv=10)
lassocv.fit(X, y)

print lassocv.alphas_  # The alphas (lambdas) it tried (you can pass
                      # your own into the constructor)
print lassocv.alpha_  # The alpha (lambda) that was best

[ 2.3842478e+04  2.08756244e+04  1.94686772e+04  1.81565535e+04
  1.69328626e+04  1.57916443e+04  1.47273404e+04  1.37347669e+04
  1.28090896e+04  1.19457998e+04  1.1106929e+04  1.03898475e+04
  9.68960653e+03  9.03655948e+03  8.42752562e+03  7.85953860e+03
  7.32983200e+03  6.83582586e+03  6.37511047e+03  5.94545272e+03
  5.5447911e+03  5.17105157e+03  4.82254000e+03  4.49751695e+03
  4.19439936e+03  3.91171088e+03  3.64807466e+03  3.4020664e+03
  3.17290930e+03  2.95906583e+03  2.75963470e+03  2.57364455e+03
  2.40018952e+03  2.23842478e+03  2.08756244e+03  1.94686772e+03
  1.81565535e+03  1.69328626e+03  1.57916443e+03  1.47273404e+03
  1.37347669e+03  1.28090896e+03  1.19457998e+04  1.1106929e+04
  1.03898475e+03  9.68960653e+03  9.03655948e+03  8.42752562e+03
  7.85953860e+03  7.32983200e+03  6.83582586e+03  6.37511047e+03
  5.94545272e+03  5.5447911e+03  5.17105157e+03  4.82254000e+03
  4.49751695e+03  4.19439936e+03  3.91171088e+03  3.64807466e+03
  3.4020664e+03  3.17290930e+03  2.95906583e+03  2.75963470e+03
  2.57364455e+03  2.40018952e+03  2.23842478e+03  2.08756244e+03
  1.94686772e+03  1.81565535e+03  1.69328626e+03  1.57916443e+03
  1.47273404e+03  1.37347669e+03  1.28090896e+03  1.19457998e+03
  1.1106929e+04  1.03898475e+03  9.68960653e+03  9.03655948e+03
  8.42752562e+03  7.85953860e+03  7.32983200e+03  6.83582586e+03
  6.37511047e+03  5.94545272e+03  5.5447911e+03  5.17105157e+03
  4.82254000e+03  4.49751695e+03  4.19439936e+03  3.91171088e+03
  3.64807466e+03  3.4020664e+03  3.17290930e+03  2.95906583e+03
  2.75963470e+03  2.57364455e+03  2.40018952e+03  2.23842478e+03]

90.365594817

For error estimation, you want nested cross-validation:

In [7]:
from sklearn.cross_validation import cross_val_score

lassocv = LassoCV(cv=10)
mses_test = np.abs(cross_val_score(lassocv, X, y,
                      scoring = 'mean_squared_error', cv=10))
mean_mse_test = np.mean(mses_test)
mean_mse_test

Out[7]: 92299.83116105033

For comparison, here is error estimate for OLS Linear Regression:
Lasso Regression gave us some improvement, and there might have been more if we had troubled to scale the data.