Lecture: Adding a regularizer to our model, and evaluating the regularized model
In this lecture we will...

• Extend our sentiment analysis codebase to incorporate a regularizer
• Demonstrate some of the model performance measures we covered previously
Incorporating a regularizer into our model

The first thing we want to do is to improve our previous model (for sentiment analysis) to include a regularizer:

$$\frac{1}{N} \sum_i (y_i - X_i \cdot \theta)^2 + \lambda \sum_k \theta_k^2$$

- MSE
- regularizer
This can be done using the "Ridge" model in sklearn.

```python
In [26]: from sklearn import linear_model

In [27]: help(linear_model.Ridge)
```

Help on class Ridge in module sklearn.linear_model.ridge:

```
class Ridge( BaseRidge, sklearn.base.RegressorMixin)
  _linear least squares with l2 regularization_  

  This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]).

  Read more in the :ref:`User Guide <ridge regression>`.

Parameters

- `alpha` : {float, array-like}, shape (n_targets)
  Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to `\`C^-1\` in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific
We can now fit the model much as before:

```
In [28]: model = linear_model.Ridge(1.0, fit_intercept=False)
    : model.fit(X, y)

Out[28]: Ridge(alpha=1.0, copy_X=True, fit_intercept=False, max_iter=None,
    : normalize=False, random_state=None, solver='auto', tol=0.001)
```

Regularization strength (i.e., lambda)
Again we can extract parameters etc. from the model, which may be slightly different than they were before:

```python
In [29]: theta = model.coef_

In [30]: wordWeights = list(zip(theta, words + ['offset']))
   ...: wordWeights.sort()

In [31]: wordWeights[:10]
```

```
Out[31]: [(-1.21086068960258, 'disappointing'),
       (-0.856640197404887, 'disappointed'),
       (-0.7889876776526171, 'unable'),
       (-0.6787442786286616, 'waste'),
       (-0.6621805938096973, 'charged'),
       (-0.5383376441665155, 'supposed'),
       (-0.5275057765332417, 'unfortunately'),
       (-0.49621911813910957, 'tried'),
       (-0.49620102935875465, 'australia'),
       (-0.47713054138625355, 'wont')]
```

```python
In [32]: wordWeights[-10:]
```

```
Out[32]: [(0.23572772781658063, 'whats'),
       (0.23820563310858286, 'problems'),
       (0.24343075578690235, 'particular'),
       (0.24673282991840637, 'worry'),
       (0.25082068960258, 'what')]
```
Model evaluation

Next, let's try to evaluate our model using some of the measures introduced previously
Calculating the MSE and R^2 statistic:

In [34]: predictions = model.predict(X)

In [35]: differences = [(x-y)**2 for (x,y) in zip(predictions,y)]

In [36]: MSE = sum(differences) / len(differences)
   print("MSE = " + str(MSE))
   MSE = 0.4260065431778631

In [37]: FVU = MSE / numpy.var(y)
   R2 = 1 - FVU
   print("R2 = " + str(R2))
   R2 = 0.38057450510836344

- FVU = Fraction of Variance Unexplained
- R^2 = 1 - FVU
Classifier evaluation

To look at some of the classifier evaluation measures we previously introduced, we can set the problem up as a classification problem.

To do so, rather than estimating the ratings (a regression problem), we'll estimate whether the rating is greater than 3 (a classification problem).
Code example: Setting up a classification problem

Convert the problem to a classification problem, and solve using logistic regression:

```python
y_class = [(rating > 3) for rating in y]

model = linear_model.LogisticRegression()
model.fit(X, y_class)
```

```
Out[40]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                        intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                        penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                        verbose=0, warm_start=False)
```
First we can calculate the accuracy of our classifier:

```python
In [41]: predictions = model.predict(X)
In [42]: correct = predictions == y_class
In [43]: accuracy = sum(correct) / len(correct)
print("Accuracy = " + str(accuracy))
```

```
Accuracy = 0.9627999946339697
```
Next, using our lists of predictions and labels, we can calculate true positives, true negatives, etc.

Code example: True Positives, True Negatives, etc.

In [44]:
TP = sum([(p and l) for (p,l) in zip(predictions, y_class)])
FP = sum([(p and not l) for (p,l) in zip(predictions, y_class)])
TN = sum([(not p and not l) for (p,l) in zip(predictions, y_class)])
FN = sum([(not p and l) for (p,l) in zip(predictions, y_class)])

In [45]:
print("TP = " + str(TP))
print("FP = " + str(FP))
print("TN = " + str(TN))
print("FN = " + str(FN))

TP = 138467
FP = 4445
TN = 5073
FN = 1101

Note: should add up to the total size of the dataset
Using these counts (TP/FP/TN/FN), we can now compute related statistics like the **accuracy**:

\[
\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}
\]

In [46]:
\[
\frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}
\]

Out[46]: 0.9627999946339697

The True Positive **Rate**, and True Negative **Rate** (etc.):

In [47]:
\[
\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \\
\text{TNR} = \frac{\text{TN}}{\text{TN} + \text{FP}}
\]

Or the **Balanced Error Rate**:

In [48]:
\[
\text{BER} = 1 - \frac{1}{2} \times (\text{TPR} + \text{TNR})
\]

print("Balanced error rate = " + str(BER))

Balanced error rate = 0.23755431598412025
Summary of concepts

- Showed how to adapt our regression code to incorporate a regularizer
- Computed simple statistics (such as the MSE and $R^2$) on regression data
- Computed several accuracy measures on classification data

On your own...

- Adapt the code to compute other evaluation measures, like the Mean Absolute Error, or the Precision and Recall