Error Estimation

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

Mean Squared Error

So, you've trained an estimator on a training set. You want to know how well it will do in practice, once you start to use it to make predictions. Easy right? We have the training set, so we measure how well the estimator performs on the training set. For each example in the training set, we ask the estimator to predict the value of the dependent variable and compare with the actual value, which is also in the training set.

For regression, we will compute the mean squared error:

$$\frac{1}{m} \sum_{i=0}^{m} (\hat{y}_i - y_i)^2$$

where $\hat{y}_i$ is the predicted value for example $i$ and $y_i$ is the actual value.

Example of Mean Squared Error on the Training Set

Let's compute the mean squared error for Linear Regression trained on the Cork property dataset:
Training Error and Test Error

Out[3]: 55456.55323798101
We will refer to the error on the training set as the **training error**. (Some people call it the ‘resubstitution error’ and sometimes the ‘in-sample error’.) But, remember, we’re not much interested in how well we have done on this data; we want to know how well we will perform in the future, on unseen data. Is the training error a good indicator of performance on unseen data? The answer is, in general: no.

The estimator’s training error (its performance on the very data on which it was trained) is likely to give an optimistic, even very optimistic, view of its future performance.

- One intuition of why this is wrong is that it’s a bit like a teacher who sets exams whose questions test the very same examples s/he used when teaching the material.
- Another intuition of why this is wrong: which estimator that we have studied can have zero training error but would be likely to perform much less well in practice?

(By the way, although the training error is not a good predictor of future performance, it can still be useful, as we will see in the lecture on Underfitting and Overfitting.)

To predict future performance, we need to measure error on an independent dataset — one that played no part in creating the estimator. This second dataset is called the **test set**, and our error on the test set we will call the **test error**. (In some circumstances people might call it the ‘out-of-sample error’ or ‘extra-sample error’.)

- If you have a ready supply of quality data, then collect one very large dataset to be the training set, and collect another very large dataset to be the test set. But large datasets are not always available, and large high-quality datasets are even harder to come by. (And, remember, it must also have the actual values of the dependent variable as well as the values of the features.)
- If the supply of data is more limited, then collect one dataset (as large as you can) and partition it into training set and test set. This is called the **holdout** method, because the test set is withheld during training. It is essential that the test set is not used in any way to create the estimator. We look at holdout, and variations of it, in more detail in this lecture.

## Holdout

We split the dataset randomly but ensuring the two sets are **disjoint**. There is a tension here. To learn a good estimator, we want the training set to be as big as possible. But for a good prediction of future performance, we want the test set to be as big as possible. Commonly, the training set will be between 50% and 80% of the dataset.

Splitting the dataset in this way is very easy in scikit-learn:
In [4]: `from sklearn.cross_validation import train_test_split`

    `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = np.random)`

    # Create linear regression object
    estimator = LinearRegression()

    # Train the model using the training set
    estimator.fit(X_train, y_train)

    # The training error
    # Predict on the training set and measure the difference between the predictions and the actual values in the training set:
    `y_predicted = estimator.predict(X_train)`
    `mse_train = mean_squared_error(y_train, y_predicted)`

    # The test error
    # Predict on the test set and measure the difference between the predictions and the actual values in the test set:
    `y_predicted = estimator.predict(X_test)`
    `mse_test = mean_squared_error(y_test, y_predicted)`

    # Display
    `mse_train, mse_test`

Out[4]: `(72457.840398695451, 18432.326882260084)`

You will find it instructive to run the above again and again to see the effect of different random splits.

**Pros and Cons of Holdout**

The advantage of this method is that the test error is independent of the training set.

The disadvantages of the holdout method are:

- You will observe that results can vary quite a lot. Informally, you might get lucky — or unlucky. Maybe you get a very 'helpful' training set, or a very 'unhelpful' training set; a very 'easy-to-predict' test set, or a very 'hard-to-predict' test set. In other words, in any one split, the data used for training or testing might not be representative.
- We are training on only a subset of the available dataset, perhaps as little as 50% of it. From so little data, we may learn a worse model and so our error measurement may be pessimistic.

In practice, you would not use the holdout method — unless you had a very large dataset that would mitigate the above problems. Instead, you would use one of its variants that we describe below. Each of these variants uses **resampling**, meaning that the examples get re-used for training and testing.

**Repeated Holdout**
One solution to the problem of biased holdout sets is to *repeat* the whole process:

- split the dataset into training and test sets
- train on the training set
- make predictions for the test set
- measure error (e.g. MSE)
- report the mean of the errors

### Illustrating scikit-learn's ShuffleSplit Class

scikit-learn provides a ShuffleSplit class, which is an iterator that gives Boolean indexes that split the dataset. Here's a simple use:

```python
In [15]: from sklearn.cross_validation import ShuffleSplit

# The dataset has 10 elements. Split it 70%/30%. Do so 3 times. Return an iterator over the indexes
ss = ShuffleSplit(n=10, n_iter=3, test_size=0.3, random_state=np.random)

# Display the indexes
for train_indexes, test_indexes in ss:
    print train_indexes, test_indexes

[8 4 0 6 9 1 5] [7 2 3]
[2 6 1 0 4 3 9] [8 5 7]
[1 4 2 5 9 0 6] [3 7 8]
```

### Using ShuffleSplit to Compute Training Error and Test Error
from sklearn.cross_validation import ShuffleSplit

def repeated_holdout_for_regression(estimator, X, y, num_iterations = 10, test_size = 0.3):
    mses_train = np.zeros(num_iterations)
    mses_test = np.zeros(num_iterations)
    ss = ShuffleSplit(n = len(y), n_iter = num_iterations, test_size = test_size,
                      random_state = np.random)
    for i, (train_indexes, test_indexes) in zip(range(num_iterations), ss):
        X_train = X[train_indexes]
        y_train = y[train_indexes]
        X_test = X[test_indexes]
        y_test = y[test_indexes]
        estimator.fit(X_train, y_train)
        y_predicted = estimator.predict(X_train)
        mses_train[i] = mean_squared_error(y_train, y_predicted)
        y_predicted = estimator.predict(X_test)
        mses_test[i] = mean_squared_error(y_test, y_predicted)
    return np.mean(mses_train), np.mean(mses_test)

# Here's an example of calling the function:
estimator = LinearRegression()
mean_mse_train, mean_mse_test = repeated_holdout_for_regression(estimator, X, y)
mean_mse_train, mean_mse_test

Out[16]: (51732.57780736428, 80806.160479259182)

Using ShuffleSplit to Compute Test Error

scikit-learn does provide a more convenient way of doing this, but it only computes the test error and it negates the values so that ‘higher is better’:

In [17]: from sklearn.cross_validation import ShuffleSplit
   : from sklearn.cross_validation import cross_val_score

ss = ShuffleSplit(n = len(y), n_iter = 10, test_size = 0.3, random_state = np.random)
estimator = LinearRegression()
mses_test = np.abs(cross_val_score(estimator, X, y, scoring = 'mean_squared_error', cv = ss))
mean_mse_test = np.mean(mses_test)
mean_mse_test

Out[17]: 59578.364218363036

Pros and Cons of Repeated Holdout
The advantage here is we can repeat indefinitely to improve our confidence. The disadvantage is training sets may overlap with each other and test sets may overlap with each other, although the effect of this is reduced if the dataset is large.

Let's look at another method.

\textbf{\(k\)-Fold Cross-Validation}

In this approach, we randomly partition the data into \(k\) disjoint subsets of equal size. (This is a different use of \(k\) from the \(k\) in \(k\NN).) Each of the partitions is called a \textit{fold}. Typically, \(k = 10\), so you have 10 folds. But, for conventional statistical significance testing to be applicable, you should probably ensure that the number of examples in each fold does not fall below 30. If this isn't possible, then either use a smaller value for \(k\), or do not use \(k\)-fold cross validation.

You take each fold in turn and use it as the test set, training the learner on the remaining folds. Clearly, you can do this \(k\) times, so that each fold gets 'a turn' at being the test set.

\begin{verbatim}
partition the dataset \(D\) into \(k\) disjoint equal-sized subsets, \(T_1, T_2, \ldots, T_k\)
for \(i = 1\) to \(k\)
  - train on \(D \setminus T_i\)
  - make predictions for \(T_i\)
  - measure error (e.g. MSE)
report the mean of the errors
\end{verbatim}

By this method, each example is used exactly once for testing, and \(k - 1\) times for training.

\textbf{Pros and Cons of \(k\)-Fold Cross-Validation}

Compared with repeated holdout, the advantages of this method are:

- The test errors of the folds are independent — because examples are included in only one test set.
- Better use is made of the dataset: for \(k = 10\), for example, we train using 9/10 of the dataset.

The disadvantages are:

- While the test sets are independent of each other, the training sets are not: they will overlap with each other to some degree. (This effect of this will be less, of course, for larger datasets.)
- The number of folds is constrained by the size of the dataset and the desire to have folds of at least 30 examples.
- It can be costly to train the learning algorithm \(k\) times.
- There may still be some variability in the results due to 'lucky'/'unlucky' splits — which motivates Repeated \(k\)-Fold Cross-Validation, below.

\textbf{Illustrating scikit-learn's KFold Class}
scikit-learn provides the KFold class, which is an iterator, similar to the ShuffleSplit class. Here’s a simple use:

```python
In [18]: from sklearn.cross_validation import KFold

    # The dataset has 10 elements and we want 5 folds
    kf = KFold(n=10, n_folds=5, shuffle=True, random_state=np.random)

    # Display the indexes
    for train_indexes, test_indexes in kf:
        print(train_indexes, test_indexes)

[0 1 2 3 5 7 8 9] [4 6]
[1 2 3 4 5 6 7 8] [0 9]
[0 2 4 5 6 7 8 9] [1 3]
[0 1 2 3 4 5 6 9] [7 8]
[0 1 3 4 6 7 8 9] [2 5]
```

**Using KFold to Compute Test Error**

Assuming that we are happy to get just the test error, we can use the cross_val_score method again:

```python
In [19]: from sklearn.cross_validation import KFold
    from sklearn.cross_validation import cross_val_score

    kf = KFold(n=len(y), n_folds=10, shuffle=True, random_state=np.random)
    estimator = LinearRegression()
    mses_test = np.abs(cross_val_score(estimator, X, y, scoring='mean_squared_error', cv=kf))
    mean_mse_test = np.mean(mses_test)
    mean_mse_test

Out[19]: 68955.237792380722
```

But, $k$-fold cross-validation is so commonplace, that there is a shorter way to write the code above, as follows:

```python
In [20]: estimator = LinearRegression()
    mses_test = np.abs(cross_val_score(estimator, X, y, scoring='mean_squared_error', cv=10))
    mean_mse_test = np.mean(mses_test)
    mean_mse_test

Out[20]: 100047.69646341192
```
Be warned, however, this almost certainly does not shuffle the dataset before splitting it into folds. Q: Why might that be a problem?

You should probably shuffle the DataFrame just after reading it in from the CSV file using, e.g.:

df = df.take(np.random.permutation(len(df)))

Final observation: In the above, we ran the 10-fold cross validation on the Cork property dataset. That dataset has only 224 examples — not enough examples to give at least 30 examples in each of the 10 folds. So this isn't an ideal use of the method.

Repeated $k$-Fold Cross-Validation

It's not uncommon to find people repeating the $k$-fold cross validation to reduce variability in the results. For example, you might run 10 times 10-fold cross-validation and average the results. This means running the learning algorithm 100 times, each time on a training set that is nine tenths of the full dataset — quite computationally expensive.

We won't look at the code. Straightforwardly, you wrap an extra loop around the code we gave above.

Leave-One-Out Cross-Validation (LOOCV)

Leave-one-out cross-validation is $k$-fold cross-validation in which $k = m$, the number of examples in the dataset: each example is in its own fold. In other words, you train the learner on all examples but one, and that one remaining example is used for testing. And you do this in turn for each example in the dataset. You'll get $m$ error values, which you can average.

\[
\text{for } i = 1 \text{ to } m \\
\quad \text{train on } D \setminus \{x^{(i)}\} \\
\quad \text{make prediction for } x^{(i)} \\
\quad \text{measure error (e.g. MSE)} \\
\quad \text{report the mean of the errors}
\]

As with $k$-fold cross-validation, each example is used exactly once in a test set. But each example is used in $m - 1$ different training sets.

Pros and Cons of LOOCV
There are advantages:

- One advantage of LOOCV is that the maximum amount of data is used for training, which makes an accurate estimator more likely. (But, see the disadvantage below.)
- Another advantage is that there is no randomness: we can't get lucky or unlucky. And there's no point in repeating the process, we'll get the same result each time.

But there are disadvantages:

- The obvious disadvantage is the cost: the learner must be trained \( m \) times, and each time it will be trained on almost all the data. This method is therefore infeasible in some cases.

**Question:**
- For which estimator do you think LOOCV is fairly common? Why?
- More subtly, LOOCV's \( m \) models are trained on almost identical data; in \( k \)-Fold Cross-Validation, the \( k \) models are trained on data with less overlap.

(Advanced note, which you can ignore: We said that LOOCV must train the learning algorithm \( m \) times. In fact, for some learners, including OLS linear regression, you can learn just the first model and then, with a bit of maths, work out the final average error without learning any of the other models. So this makes LOOCV practical for this class of learning algorithms.)

You can see that there is a trade-off here. Empirically, \( k \)-Fold Cross-Validation with \( k = 5 \) or \( k = 10 \) tends to report the most reliable error figures.

### Using LeaveOneOut to Compute Test Error

Here is LOOCV in scikit-learn:

```python
In [22]: from sklearn.cross_validation import LeaveOneOut
   from sklearn.cross_validation import cross_val_score

   loocv = LeaveOneOut(n = len(y))
estimator = LinearRegression()
mses_test = np.abs(cross_val_score(estimator, X, y, scoring = 'mean_squared_error', cv = loocv))
mean_mse_test = np.mean(mses_test)
mean_mse_test
```

```
Out[22]: 72376.115691746352
```

### Final Remarks
1. There are methods other than those covered including Bootstrapping and Permutation Tests.
2. So you've used one of the above methods and found the test error of your estimator. The dirty secret of Machine Learning is this: at this point, if dissatisfied with the test error, many Machine Learning researchers, start tweaking their learning algorithms to try to bring down the test error. This is wrong! It is called **leakage**: knowledge of the test set is being used to develop the estimator. It's like the teacher letting the students take the same exam again. It will result in the test error giving an optimistic view of the ultimate performance of the estimator on unseen data.

   If you must do something like this, then somewhat less problematic is if you ensure that you are using different random splits when evaluating your tweaks.

3. Finally, suppose you have used one of the above methods to estimate the error of your regressor. You are ready to release your regressor on the world. At this point, you can train it on *all* the examples in your dataset, so as to maximize the use of the data.