Model Selection

As discussed previously, the learning algorithm finds the values of parameters. But there are other parameters of learning algorithms — ones that the learning algorithm does not set — which are called hyperparameters. Our best example so far is the value of $k$ in kNN.

In this lecture, we look at ways of setting the values of the hyperparameters. This is called model selection. Despite what it may sound like, this is not about choosing between different learning algorithms (e.g. between OLS linear regression and kNN) — that, after all, can be done by error estimation (which we discussed in a previous lecture). Model selection is about optimizing the settings for each learning algorithm.

For some learning algorithms there might be special methods for setting some of the hyperparameters. If not, in essence, you just try lots of values for your hyperparameters and see which is best — an approach known as ‘grid search’. But how do we estimate the error of each hyperparameter value that we try? This is what we will discuss first.

Validation Sets
As in the holdout method, we randomly partition the dataset into disjoint sets. Here, these partitions are called the training set and the **validation set**.

For each value of the hyperparameter that you wish to try, you train the learning algorithm on the training set. So, assuming one hyperparameter with \( \text{numvals} \) values that you wish to try, you will learn \( \text{numvals} \) models. Let's designate them by \( h_1, \ldots, h_{\text{numvals}} \). You test each of them on the validation set, e.g. measuring mean squared error, as before. From this, you can see which of the \( \text{numvals} \) models (corresponding to \( \text{numvals} \) different values of the hyperparameter) appears to be the best — the one with the lowest error on the validation set. This gives you the value of the hyperparameter. That's model selection.

The weakness of this (as before) is that the training and validation sets may not be representative: we may get 'lucky' or 'unlucky'. So we need to use resampling again: either repeated holdout, \( k \)-fold cross-validation, repeated \( k \)-fold cross-validation, leave-one-out cross-validation, or one of the other methods that we did not study. For concreteness, let's focus on one of them: \( k \)-fold cross-validation.

### \( k \)-Fold Cross-Validation for Model Selection

To use \( k \)-fold cross-validation for model selection, you divide into \( k \) folds. You take all but fold 1 as the training set and you learn \( \text{numvals} \) models (one per hyperparameter value), and you test them on fold 1, recording the mean squared error of each model. You repeat, taking a different fold to be the test set each time. For each hyperparameter value, you take the mean of its \( k \) mse's. The hyperparameter value with lowest mean mse is the value that you should 'go live' with. (At this point, you can train on all the data, using the best hyperparameter value that you have just discovered.) Here it is in pseudocode:

\[
\text{partition the dataset } D \text{ into } k \text{ disjoint equal-sized subsets, } T_1, T_2, \ldots, T_k \\
\text{for } u = 1 \text{ to } k \\
\quad \text{for } v = 1 \text{ to } \text{numvals} \\
\quad \quad \text{train on } D \setminus T_u \text{ with the } v\text{th hyperparameter value} \\
\quad \quad \text{make predictions for } T_u \\
\quad \quad \text{measure error (e.g. MSE)} \\
\text{get the mean of the errors for each hyperparameter value} \\
\text{report the hyperparameter value with lowest meanmse}
\]

Let's now combine this validation set idea with grid search — and let's do so in scikit-learn.

### Grid Search

**Grid search** is an 'exhaustive' approach — it tries every value for the hyperparameter (or, at least, all values in a certain range). If there is more than one hyperparameter, it tries every combination of their values.

- If one hyperparameter has \( \text{numvals}_1 \) values and another has \( \text{numvals}_2 \) values, how many models will grid search train?
Grid Search and Holdout in scikit-learn

Let's try all values of kNN's hyperparameter $k$ between 1 and 10 for unweighted kNN on the Cork property dataset using holdout (where we simply split the dataset once into training set and validation set):

```python
In [3]: from sklearn.neighbors import KNeighborsRegressor
from sklearn.cross_validation import ShuffleSplit
from sklearn.grid_search import GridSearchCV

def read_csv(file):
    # Use pandas to read the CSV file
    df = pd.read_csv("dataset-corkA.csv")

    # Get the feature-values and the target values into separate numpy arrays of numbers
    X = df[['flarea', 'bdrms', 'bthrms']].values
    y = df['price'].values

    # Create kNN Regressor object
    estimator = KNeighborsRegressor()

    # Dictionary containing each hyperparameter and the hyperparameter#'s possible values
    hyperparameters = {'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}

    # We want to split the dataset just once into training and validation
    ss = ShuffleSplit(n=len(y), n_iter=1, test_size=.3, random_state=np.random)

    # And here we create the Grid Search object and run its fit method
    gs = GridSearchCV(estimator, hyperparameters, scoring='mean_squared_error', cv=ss)
    gs.fit(X, y)

    # Display some of its instance variables afterwards
    print(gs.grid_scores_)
    print(gs.best_estimator_)
    print(gs.best_score_)
    print(gs.best_params_)

KNeighborsRegressor(algorithm=auto, leaf_size=30, metric=minkowski, n_neighbors=5, p=2, weights=uniform)
-5904.72941176
{'n_neighbors': 5}
```
This suggests a particular value for $k$. But, remember, we are doing only one split of the data. Re-run it and you will see that the best value for the hyperparameter differs from run to run.

### Grid Search and $k$-Fold Cross-Validation in scikit-learn

If we want to do $k$-fold cross-validation instead (which we hope will give a more robust answer), we simply feed a different cross-validation method into the grid search. As we saw before, scikit-learn makes it very easy to choose $k$-fold cross-validation: simply supply the number of folds (but, remember, you may need to do your own shuffling):

```python
In [4]: from sklearn.cross_validation import KFold

# And here's we create the Grid Search object and run its fit method
gs = GridSearchCV(estimator, hyperparameters, scoring = 'mean_squared_error', cv = 10)
gs.fit(X, y)

# Display some of its instance variables afterwards
print(gs.grid_scores_)
print(gs.best_estimator_)
print(gs.best_score_)
print(gs.best_params_)
```

```
[mean: -120836.04911, std: 229132.75725, params: {'n_neighbors': 1}],
mean: -137319.21429, std: 242303.31122, params: {'n_neighbors': 2}],
mean: -136642.22569, std: 241851.18113, params: {'n_neighbors': 3}],
mean: -114827.41462, std: 228408.08430, params: {'n_neighbors': 4}],
mean: -109692.46893, std: 233010.31668, params: {'n_neighbors': 5}],
mean: -104796.04501, std: 242880.24167, params: {'n_neighbors': 6}],
mean: -103158.51576, std: 255682.18331, params: {'n_neighbors': 7}],
mean: -99171.72112, std: 256983.59966, params: {'n_neighbors': 8}],
mean: -96955.54249, std: 258614.41602, params: {'n_neighbors': 9}],
mean: -97213.61929, std: 263218.88941, params: {'n_neighbors': 10}]
KNeighborsRegressor(algorithm=auto, leaf_size=30, metric=minkowski, n_neighbors=9, p=2, weights=uniform)

-96955.5424934
{'n_neighbors': 9}
```

It selects $k = 9$ (although if we were feeding in a suitably-initialized KFold object, you may still get different answers each time you run it, but with a lot less variability than before).

Assuming we now want to 'go live', we would train 9NN on all the data.

### Error Estimation, again
Suppose you want to do error estimation, as before, i.e. you want to know how well a model is likely to perform on unseen data. But suppose there is a hyperparameter and so you want your estimate of the error on unseen data to use a good value for the hyperparameter. In this case, we need to combine ideas from the earlier part of this lecture with ideas from the error estimation lecture.

The simplest case is to partition your data into three sets (assuming you have enough data!):

- training set,
- validation set, and
- test set.

You use the training set and the validation set to choose the best value for the hyperparameter, i.e. train `numvals` models (one for each value of the hyperparameter), test them on the validation set, using grid search. Choose the value that gave the lowest mse. Then, test the model you have just selected on the test set: the mse that this produces is your estimate of the error on unseen examples.

### Training Set, Validation Set and Test Set in scikit-learn

In this scikit-learn example, we will split the data 60-20-20:

```python
In [5]:
from sklearn.cross_validation import train_test_split
from sklearn.metrics import mean_squared_error

# Split the data 80/20: the 80 is the combined training and validation sets; the 20 is the final test set
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size = 0.2, random_state = np.random)

# Split the combined set just once into training and validation: 75/25 gives 60/20 of the original data
ss = ShuffleSplit(n = len(y_train_val), n_iter = 1, test_size = 0.25, random_state = np.random)

# And here we create the Grid Search object and run its fit method
gs = GridSearchCV(estimator, hyperparameters, scoring = 'mean_squared_error', cv = ss)
gs.fit(X, y)

# Take the best estimator found by the grid search
# Test it on the test set
y_predicted = gs.predict(X_test)
mse_test = mean_squared_error(y_test, y_predicted)

# Display
mse_test
```

```
Out[5]: 51752.965277777781
```

This estimate has the weakness (again) that it is based on just one way of partitioning the data: different partitions will have very different error estimates — as you can confirm by running the code several times. We need to use resampling.
Nested $k$-Fold Cross-Validation

What we will do is called nested $k$-fold cross validation. As usual, it partitions the dataset into $k$ folds. One fold is kept as the test set. The remaining data is then partitioned again into $k$ folds. One of these folds is kept as the validation set. The remaining is used for training. We learn numvals models from the training data (one for each value of the hyperparameter). We test each of them on the validation set. We repeat this, taking each inner fold in turn as the validation set. When we have done, we can select the best value of the hyperparameter. We test this model on the outer fold that we kept as test set. Then the whole thing repeats: a different outer fold becomes the test set; $k$-fold cross-validation is run on the remaining data to find another best value for the hyperparameter; and this model is tested on the outer fold that was kept as test set. And so on.

In pseudocode:

partition the dataset $D$ into $k$ disjoint equal-sized subsets, $T_1, T_2, \ldots, T_k$
for $u_1 = 1$ to $k$
  • let $D'$ be $D \setminus T_{u_1}$
  • partition $D'$ into $k$ disjoint equal-sized subsets, $S_1, S_2, \ldots, S_k$
  • for $u_2 = 1$ to numvals
    • train on $D' \setminus S_{u_2}$ with the $v$th hyperparameter value
    • make predictions for $S_{u_2}$
    • measure validation error (e.g. MSE)
  • get the mean of the errors for each hyperparameter value
  • select the model (hyperparameter value) with lowest mean mse
  • use the selected model to make predictions for $T_{u_1}$
  • measure test error (e.g. MSE)
report the means of the test errors

Nested $k$-Fold Cross-Validation in scikit-learn

In [6]: from sklearn.cross_validation import cross_val_score

outer_kf = KFold(n = len(y), n_folds = 10, shuffle = True, random_state = np.random)
ininner_kf = KFold(n = int(len(y) * 9 / 10.0), n_folds = 10, shuffle = True, random_state = np.random)
gs = GridSearchCV(estimator, hyperparameters, scoring = 'mean_squared_error', cv = inner_kf)
mse_train = np.abs(cross_val_score(gs, X, y, scoring = 'mean_squared_error', cv = outer_kf))
mean_mse_train = np.mean(mse_train)
mean_mse_test

Out[6]: 152073.62807795344

Happily, there’s a shorter version again (but again probably requiring you to do the shuffling):
In [7]:
   gs = GridSearchCV(estimator, hyperparameters, scoring = 'mean_squared_error', cv = 10)
   mses_test = np.abs(cross_val_score(gs, X, y, scoring = 'mean_squared_error', cv = 10))
   mean_mse_test = np.mean(mses_test)
   mean_mse_test

Out[7]: 101001.56507719708

Of course, our little dataset really isn't big enough for this kind of treatment.

**Confusion**

You might be thinking to yourself: "But the best value for the hyperparameter can be different for each outer fold." You are correct. And so then you may be thinking: "So which value for the hyperparameter do I choose?" To which the reply is: "None of them. This is not model selection. This is just error estimation." If you want to do model selection, see earlier.