Cross-validation: evaluating estimator performance

Evaluating estimator performance

- Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data.
- This situation is called overfitting.
- To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a test set X_test, y_test.
- Note that the word “experiment” is not intended to denote academic use only, because even in commercial settings machine learning usually starts out experimentally.

Evaluating estimator performance

- clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
- >>> clf.score(X_test, y_test)
- 0.96...
- When evaluating different settings (“hyperparameters”) for estimators, such as the c setting that must be manually set for an SVM, there is still a risk of overfitting *on the test set* because the parameters can be tweaked until the estimator performs optimally.
- This way, knowledge about the test set can “leak” into the model and evaluation metrics no longer report on generalization performance.
- To solve this problem, yet another part of the dataset can be held out as a so-called “validation set”: training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

Evaluating estimator performance

- A solution to this problem is a procedure called Cross Validation (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV.
- In the basic approach, called *k*-fold CV, the training set is split into *k* smaller sets (other approaches are described below, but generally follow the same principles). The following procedure is followed for each of the *k* “folds”:
- A model is trained using k-1 of the folds as training data;
- The resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).
- The performance measure reported by *k*-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set), which is a major advantage in problems such as inverse inference where the number of samples is very small.

Benefits/Drawbcks of cross-validation:

- More **reliable** estimate of out-of-sample performance than train/test split
  - Reduce the variance of a single trial of a train/test split
- Can be used for
  - Selecting **tuning parameters**
  - Choosing between **models**
  - Selecting **features**

Drawbacks of cross-validation:

- Can be computationally **expensive**
  - Especially when the data set is very large or the model is slow to train

Computing cross-validated metrics using Cross_val_score

- The simplest way to use cross-validation is to call the cross_val_score helper function on the estimator and the dataset.
- Sklearn.model_selection.cross_val_score (*estimator*, *X*, *y=None*, *scoring=None*, *cv=<int>*,…….)

  - estimator : estimator object implementing 'fit'
    The object to use to fit the data.

  - X : array-like
    The data to fit. Can be for example a list, or an array.

  - y : array-like, optional, default: None
    The target variable to try to predict in the case of supervised learning.

Computing cross-validated metrics

- scoring : string, callable or None, optional, default: None
- cv : int, cross-validation generator or an iterable, optional

Determines the cross-validation splitting strategy. Possible inputs for cv are:

None, to use the default 3-fold cross validation

integer, to specify the number of folds in a KFold,

CV splitter,

An iterable yielding (train, test) splits as arrays of indices.

Computing cross-validated metrics

- The function returns

  - scores : array of float, shape=(len(list(cv)),)

    Array of scores of the estimator for each run of the cross validation.

Computing cross-validated metrics

- The following example demonstrates how to estimate the accuracy of a linear kernel support vector machine on the iris dataset by splitting the data, fitting a model and computing the score 5 consecutive times (with different splits each time):
- clf = svm.SVC(kernel='linear', C=1)
- scores = cross_val_score(clf, iris.data, iris.target, cv=5)
- scores
- array([0.96..., 1. ..., 0.96..., 0.96..., 1.])

Computing cross-validated metrics

- The mean score and the 95% confidence interval of the score estimate are hence given by:
- print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
- Accuracy: 0.98 (+/- 0.03)
- By default, the score computed at each CV iteration is the score method of the estimator. It is possible to change this by using the scoring parameter:
- from sklearn import metrics
- scores = cross_val_score(
- ... clf, iris.data, iris.target, cv=5, scoring='f1_macro')
- scores
- array([0.96..., 1. ..., 0.96..., 0.96..., 1.])

parameter tuning using cross_val_score

- **Goal:** Select the best tuning parameters (ie "hyperparameters") for KNN on the iris dataset
- To select the best value of k for KNN model to predict species
- # imports
- from sklearn.datasets import load_iris
- from sklearn.neighbors import KNeighborsClassifier
- from sklearn.cross_validation import cross_val_score
- import matplotlib.pyplot as plt
- %matplotlib inline

Parameter tuning using cross_val_score

- # read in the iris data
- iris = load_iris()

- # create X (features) and y (response)
- X = iris.data
- y = iris.target

- print('X matrix dimensionality:', X.shape)
- print('Y vector dimensionality:', y.shape)

- Shows
- X matrix dimensionality: (150,4) Y vector dimensionality: (150,)

Parameter tuning using cross_val_score

- # 10-fold (cv=10) cross-validation with K=5 (n_neighbors=5) for KNN (the n_neighbors parameter)
- # instantiate model
- knn = KNeighborsClassifier(n_neighbors=5)
- # store scores in scores object
- # scoring metric used here is 'accuracy' because it's a classification problem
- # cross_val_score takes care of splitting X and y into the 10 folds that's why we pass X and y entirely instead of X_train and y_train
- scores = cross_val_score(knn, X, y, cv=10, scoring='accuracy')
- print(scores)

Shows

- [1. 0.93333333 1. 1. 0.86666667 0.93333333 0.93333333 1. 1. 1.]

Parameter tuning using cross_val_score

- # use average accuracy as an estimate of out-of-sample accuracy

- # scores is a numpy array so we can use the mean method
- print(scores.mean())

Shows: 0.9666666666666668

Example for opimizing an algorithm

- # search for an optimal value of K for KNN
- k_range = range(1, 31)
- # list of scores from k_range
- k_scores = []
- # 1. we will loop through reasonable values of k
- for k in k_range:
- # 2. run KNeighborsClassifier with k neighbours
- knn = KNeighborsClassifier(n_neighbors=k)
- # 3. obtain cross_val_score for KNeighborsClassifier with k neighbours
- scores = cross_val_score(knn, X, y, cv=10, scoring='accuracy')
- # 4. append mean of scores for k neighbors to k_scores list
- k_scores.append(scores.mean())
- print(k_scores)

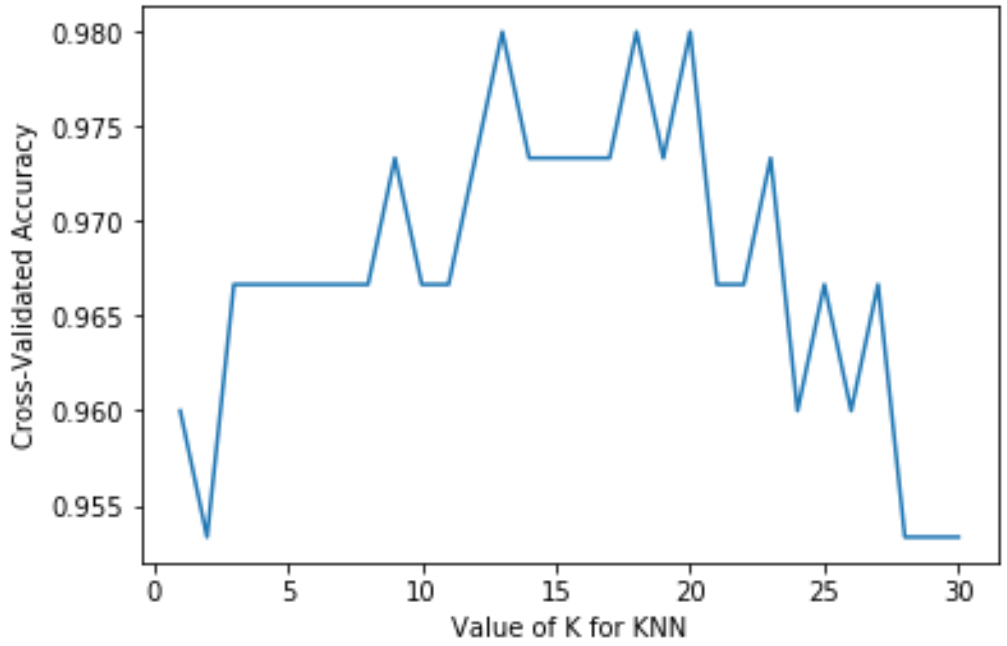
Example for opimizing an algorithm

Shows:

[0.96, 0.9533333333333334, 0.9666666666666666, 0.9666666666666666, 0.9666666666666668, 0.9666666666666668, 0.9666666666666668, 0.9666666666666668, 0.9733333333333334, 0.9666666666666668, 0.9666666666666668, 0.9733333333333334, 0.9800000000000001, 0.9733333333333334, 0.9733333333333334, 0.9733333333333334, 0.9733333333333334, 0.9800000000000001, 0.9733333333333334, 0.9800000000000001, 0.9666666666666666, 0.9666666666666666, 0.9733333333333334, 0.96, 0.9666666666666666, 0.96, 0.9666666666666666, 0.9533333333333334, 0.9533333333333334, 0.9533333333333334]

Plotting different values of K

- # plot the value of K for KNN (x-axis) versus the cross-validated accuracy (y-axis)
- plt.plot(k_range, k_scores)
- plt.xlabel('Value of K for KNN')
- plt.ylabel('Cross-Validated Accuracy')



Computing cross-validated metrics using Cross_val_predict

- The function cross_val_predict has a similar interface to cross_val_score but returns, for each element in the input, the prediction that was obtained for that element when it was in the test set. Only cross-validation strategies that assign all elements to a test set exactly once can be used (otherwise, an exception is raised).
- The result of cross_val_predict may be different from those obtained using cross_val_score as the elements are grouped in different ways. The function cross_val_score takes an average over cross-validation folds, whereas cross_val_predict simply returns the labels (or probabilities) from several distinct models undistinguished.
- The function cross_val_predict is appropriate for :
  - Visualization of predictions obtained from different models.
  - Model blending: When predictions of one supervised estimator are used to train another estimator in ensemble methods.
- sklearn.model_selection.cross_val_predict(estimator, X, y=None, fit_params=None, method='predict')