Cross Validation Techniques

Definition:

There is always a need to validate the stability of your machine learning model. I mean you just can't fit the model to your training data and hope it would accurately work for the real data it has never seen before. You need some kind of assurance that your model has got most of the patterns from the data correct, and it's not picking up too much on the noise, or in other words it's low on bias and variance.

Generally, an error estimation for the model is made after training, better known as evaluation of residuals. In this process, a numerical estimate of the difference in predicted and original responses is done, also called the training error. However, this only gives us an idea about how well our model does on data used to train it. Now it's possible that the model is underfitting or overfitting the data. So, the problem with this evaluation technique is that it does not give an indication of how well the learner will generalize to an independent/ unseen data set. Getting this idea about our model is known as **Cross Validation**.

Some of the cross validation techniques are:

1) Holdout Method
2) K-Fold Cross Validation
3) Stratified K-Fold Cross Validation
4) Leave-P-Out Cross Validation

Holdout Method:

Now a basic remedy for this involves removing a part of the training data and using it to get predictions from the model trained on rest of the data. The error estimation then tells how our model is doing on unseen data or the validation set. This is a simple kind of cross validation technique, also known as the holdout method. Although this method doesn't take any overhead to compute and is better than traditional validation, it still suffers from issues of high variance. This is because it is not certain which data points will end up in the validation set and the result might be entirely different for different sets.

K-Fold Cross Validation:

As there is never enough data to train your model, removing a part of it for validation poses a problem of underfitting. By reducing the training data, we risk losing important patterns/ trends in data set, which in turn increases error induced by bias. So, what we require is a method that provides ample data for training the model and also leaves ample data for validation. K Fold cross validation does exactly that.

In K Fold cross validation, the data is divided into k subsets. Now the holdout method is repeated k times, such that each time, one of the k subsets is used as the test set/ validation set and the other k-1 subsets are put together to form a training set. The error estimation is averaged over all k trials to get total effectiveness of our model. As can be seen, every data point gets to be in a validation set exactly once, and gets to be in a training set k-1 times. This significantly reduces bias as we are using most of the data for fitting, and also significantly reduces variance as most of the data is also being used in validation set. Interchanging the training and test sets also adds to the effectiveness of this method. As a general rule and empirical evidence, K = 5 or 10 is generally preferred, but nothing's fixed and it can take any value.

Parameters:

- **n_splits :** int, default=3 - Number of folds. Must be at least 2.
- **shuffle** : boolean, optional - Whether to shuffle the data before splitting into batches.
- **random_state** : int, RandomState instance or None, optional, default=None - If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Used when shuffle == True.

Methods:

`get_n_splits`(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
`split`(self, X[, y, groups])	Generate indices to split data into training and test set.

Examples:

import numpy as np

from sklearn.model_selection import KFold

X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])

y = np.array([1, 2, 3, 4])

kf = KFold(n_splits=2)

kf.get_n_splits(X) - 2

print(kf)

KFold(n_splits=2, random_state=None, shuffle=False)

for train_index, test_index in kf.split(X):

```
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

```
TRAIN: [2 3] TEST: [0 1]
TRAIN: [0 1] TEST: [2 3]
```

Stratified K-Fold Cross Validation:

In some cases, there may be a large imbalance in the response variables. For example, in dataset concerning price of houses, there might be large number of houses having high price. Or in case of classification, there might be several times more negative samples than positive samples. For such problems, a slight variation in the K Fold cross validation technique is made, such that each fold contains approximately the same percentage of samples of each target class as the complete set, or in case of prediction problems, the mean response value is approximately equal in all the folds. This variation is also known as **Stratified K Fold.**

Parameters:

- **n_splits :** int, default=3 - Number of folds. Must be at least 2.
- **shuffle :** boolean, optional - Whether to shuffle each class's samples before splitting into batches.
- **random_state :** int, RandomState instance or None, optional, default=None - If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Used when shuffle == True.

Methods:

`get_n_splits`(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
`split`(self, X, y[, groups])	Generate indices to split data into training and test set.

Examples:

```
import numpy as np
from sklearn.model_selection import StratifiedKFold
X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
y = np.array([0, 0, 1, 1])
skf = StratifiedKFold(n_splits=2)
skf.get_n_splits(X, y)
2
print(skf)
StratifiedKFold(n_splits=2, random_state=None, shuffle=False)
for train_index, test_index in skf.split(X, y):
        print("TRAIN:", train_index, "TEST:", test_index)
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]
TRAIN: [1 3] TEST: [0 2]
TRAIN: [0 2] TEST: [1 3]
```

Leave-P-Out Cross Validation:

This approach leaves p data points out of training data, i.e. if there are n data points in the original sample then, n-p samples are used to train the model and p points are used as the validation set. This is repeated for all combinations in which original sample can be separated this way, and then the error is averaged for all trials, to give overall effectiveness. This method is exhaustive in the sense that it needs to train and validate the model for all possible combinations, and for moderately large p, it can become computationally infeasible.

A particular case of this method is when p = 1. This is known as **leave one out cross validation**. This method is generally preferred over the previous one because it does not suffer from the intensive computation, as number of possible combinations is equal to number of data points in original sample or n.

Parameters:

- **X :** array-like, shape (n_samples, n_features) - Training data, where n_samples is the number of samples and n_features is the number of features.
- **y :** array-like, of length n_samples - The target variable for supervised learning problems.
- **groups :** array-like, with shape (n_samples,), optional - Group labels for the samples used while splitting the dataset into train/test set.

Methods:

`get_n_splits`(self, X[, y, groups])	Returns the number of splitting iterations in the cross-validator
`split`(self, X[, y, groups])	Generate indices to split data into training and test set.

Examples:

```
import numpy as np
from sklearn.model_selection import LeaveOneOut
X = np.array([[1, 2], [3, 4]])
y = np.array([1, 2])
loo = LeaveOneOut()
loo.get_n_splits(X)
2
print(loo)
LeaveOneOut()
for train_index, test_index in loo.split(X):
        print("TRAIN:", train_index, "TEST:", test_index)
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]
        print(X_train, X_test, y_train, y_test)
TRAIN: [1] TEST: [0]
[[3 4]] [[1 2]] [2] [1]
TRAIN: [0] TEST: [1]
[[1 2]] [[3 4]] [1] [2]
```