k-fold cross-validation explained in plain English (For evaluating a model's performance and hyperparameter tuning)

Article · December 2020

CITATIONS
0

READS
525

1 author:

Rukshan Manorathna
University of Colombo
12 PUBLICATIONS **4** CITATIONS

SEE PROFILE

k-fold cross-validation explained in plain English

For evaluating a model's performance and hyperparameter tuning

Rukshan Manorathna **Dec 19, 2020**

Photo by Scott Webb on Unsplash

k-fold cross-validation is one of the most popular strategies widely used by data scientists. It is a **data partitioning strategy** so that you can effectively use your dataset to build **a more generalized model**. The main intention of doing any kind of machine learning is to develop a more generalized model which can perform well on **unseen data**. One can build a perfect model on the training data with 100% accuracy or 0 error, but it may fail to generalize for unseen data. So, it is not a good model. It overfits the training data. Machine Learning is all about **generalization** meaning that model's performance can only be measured with data points that have never been used during the training process. That is why we often split our data into a training set and a test set.

Data splitting process can be done more effectively with k-fold cross-validation. Here, we discuss two scenarios which involve k-fold cross-validation. Both involve splitting the dataset, but with different approaches.

- **Using k-fold cross-validation for evaluating a model's performance**
- **Using k-fold cross-validation for hyperparameter tuning**

Each scenario will be discussed by implementing the Python code with a real-world dataset. I will also use some graphical visualizations so that everyone can understand what is going on behind k-fold cross-validation in each scenario.

So, without further delay, you are ***all*** welcome to continue reading this article. I will also include the Python code hosted on my GitHub account and the dataset that I use here. So, you can use them to practise yourself.

We will begin with the first scenario.

Using k-fold cross-validation for evaluating a model's performance

Let's begin directly with an example. For this, I will use a very popular dataset called **bike-sharing dataset**.

	instant	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	cnt
0	1	1.0	0.0	1.0	0.0	6.0	0.0	2	0.344167	0.363625	0.805833	0.160446	985
1	2	1.0	0.0	1.0	0.0	0.0	0.0	2	0.363478	0.353739	0.696087	0.248539	801
2	3	1.0	0.0	1.0	0.0	1.0	1.0	1	0.196364	0.189405	0.437273	0.248309	1349
3	4	1.0	0.0	1.0	0.0	2.0	1.0	1	0.200000	0.212122	0.590435	0.160296	1562
4	5	1.0	0.0	1.0	0.0	3.0	1.0	1	0.226957	0.229270	0.436957	0.186900	1600

First 5 rows of the bike-sharing dataset

We will try to predict the correct number of bike rentals ('cnt') on a given day based on other variables in the dataset. So, this involves predicting a number, not a class. So, this is a regression task in machine learning. Here, we use linear regression (learn how linear regression works behind the scenes by reading **this article** written by me). We use the **Root Mean Squared Error (RMSE)** as the model performance metric to evaluate the regression model.

```python
import numpy as np
import pandas as pd

df = pd.read_csv("bike_rentals.csv")
X = df.iloc[:,:-1]
y = df.iloc[:,-1]

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True,
                                    test_size=0.20, random_state=42)

from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X_train, y_train)
y_pred = lr.predict(X_test)

from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
print("RMSE: ", np.round(rmse, 2))
```

cv.py hosted with ❤ by GitHub view raw

```
RMSE:  824.33
```

After running the above code block, we get RMSE of 824.33. But this value depends on how we split our data into ***X_train*** and ***y_train***. Here, we set **random_state=42**. According to that value, we obtained RMSE of 824.33. If we set **random_state=0**, we will obtain RMSE of 909.81. If we set **random_state=35**, we will get RMSE of 794.15. The reason behind getting different RMSE values is that each time we call the **train_test_split** function with different values of the **random_state** hyperparameter, it provides different sets of observations for ***X_train*** and ***y_train***. So, what value do we accept as the RMSE? Is it 824.33, 909.81, 794.15 or any other value? As a solution to that problem, one would suggest that we could train the model several times with different values of **random_state** and then get the average RMSE value. But instead of doing this manually, we can automate this process using the Scikitlearn **cross_val_score** function.

```python
from sklearn.utils import shuffle
X_shuffle, y_shuffle = shuffle(X, y, random_state=42)

from sklearn.model_selection import cross_val_score
scores = cross_val_score(lr, X_shuffle, y_shuffle,
                         scoring="neg_mean_squared_error",
                         cv=5, n_jobs=1)
rmse = np.sqrt(-scores)
print("RMSE values: ", np.round(rmse, 2))
print("RMSE average: ", np.mean(rmse))
```

cv2.py hosted with ❤ by GitHub view raw

```
RMSE values:  [824.33 955.56 841.34 932.75 863.13]
RMSE average:  883.4197051998128
```

In k-fold cross-validation, we make an assumption that all observations in the dataset are nicely distributed in a way that the data are not biased. That is why we first shuffle the dataset using the **shuffle** function. Then we call the **cross_val_score** function. Note that we provide the whole dataset to this function. Here, we do not need to provide the split data as ***X_train***, ***y_train*** because the splitting is done internally during the process. The **cv** hyperparameter represents the number of folds (here it is 5). By providing appropriate values to **n_jobs**, we can do parallel computations. We can distribute the evaluation of the different folds across multiple CPUs on our computer. If we set the **n_jobs** to 1, only one CPU will be used to evaluate the performances. However, by setting ***n_jobs=2***, we could distribute the process of cross-validation to two CPUs. Finally, by setting **n_jobs=-1**, we can use all available CPUs on our computer to do the computation in parallel. This is a very effective way, especially when we have a large dataset and when we set cv to a large number, for example, 10.

Let’s see what is going on behind k-fold cross-validation when evaluating a model’s performance.

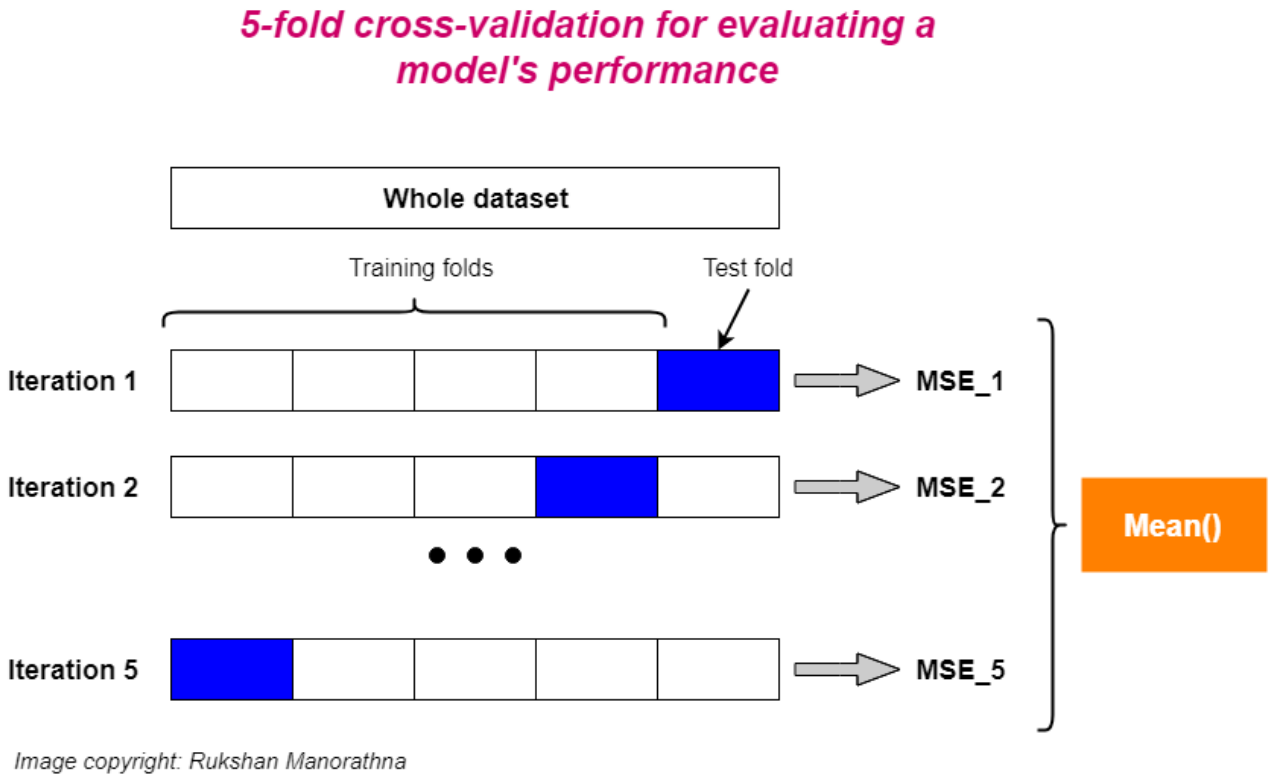

Image by Author

The general process of k-fold cross-validation for evaluating a model's performance is:

- The whole dataset is *randomly* split into *independent* **k-folds** *without replacement*.
- **k-1 folds** are used for the model training and one fold is used for performance evaluation.
- This procedure is repeated **k times** (iterations) so that we obtain **k number** of performance estimates (e.g. MSE) for each iteration.
- Then we get the mean of **k number** of performance estimates (e.g. MSE).

Remark 1: The splitting process is done without replacement. So, each observation will be used for training and validation exactly once.

Remark 2: Good standard values for ***k*** in k-fold cross-validation are 5 and 10. However, the value of ***k*** depends on the size of the dataset. For small datasets, we can

use higher values for ***k***. However, larger values of ***k*** will also increase the runtime of the cross-validation algorithm and the computational cost.

Remark 3: When **k=5**, 20% of the test set is held back each time. When **k=10**, 10% of the test set is held back each time and so on…

Remark 4: A special case of k-fold cross-validation is the **Leave-one-out crossvalidation (LOOCV)** method in which we set **k=n** (number of observations in the dataset). Only one training sample is used for testing during each iteration. This method is very useful when working with very small datasets.

Using k-fold cross-validation for hyperparameter tuning

A note on model parameters vs hyperparameters

Model parameters are the parameters which learn during the training process. We do not manually set values for the parameters and they learn from the data that we provide. In contrast, the ***model hyperparameters*** are the parameters that do not learn from data. So, we have to set values for them manually. We always set values for the model hyperparameters at the creation of a particular model and before we start the training process.

Grid search is a popular hyperparameter optimization technique. It is looking for an optimal combination of hyperparameter values in a way that it can further improve the performance of a model.

Using k-fold cross-validation in combination with grid search is a very useful strategy to improve the performance of a machine learning model by tuning the model hyperparameters.

In grid search, we can set values for multiple hyperparameters. Let’s say we have two hyperparameters each having three different values. So, we have 9 (3 x 3) different combinations of hyperparameter values. The space in which all those combinations contain is called the **hyperparameter space**. When there are two hyperparameters, space is two dimensional.

In grid search, the algorithm takes one combination of hyperparameter values at a time from the hyperparameter space that we have specified. Then it trains the model using those hyperparameter values and evaluates it through k-fold cross-validation. It stores the performance estimate (e.g. MSE). Then the algorithm takes another combination of hyperparameter values and does the same. After taking all the combinations, the algorithm stores all the performance estimates. Out of those estimates, it selects the best one. The combination of hyperparameter values that yields the best performance estimate is the optimal combination of hyperparameter values. The best model includes those hyperparameter values.

Let's see what is going on behind k-fold cross-validation with grid search.

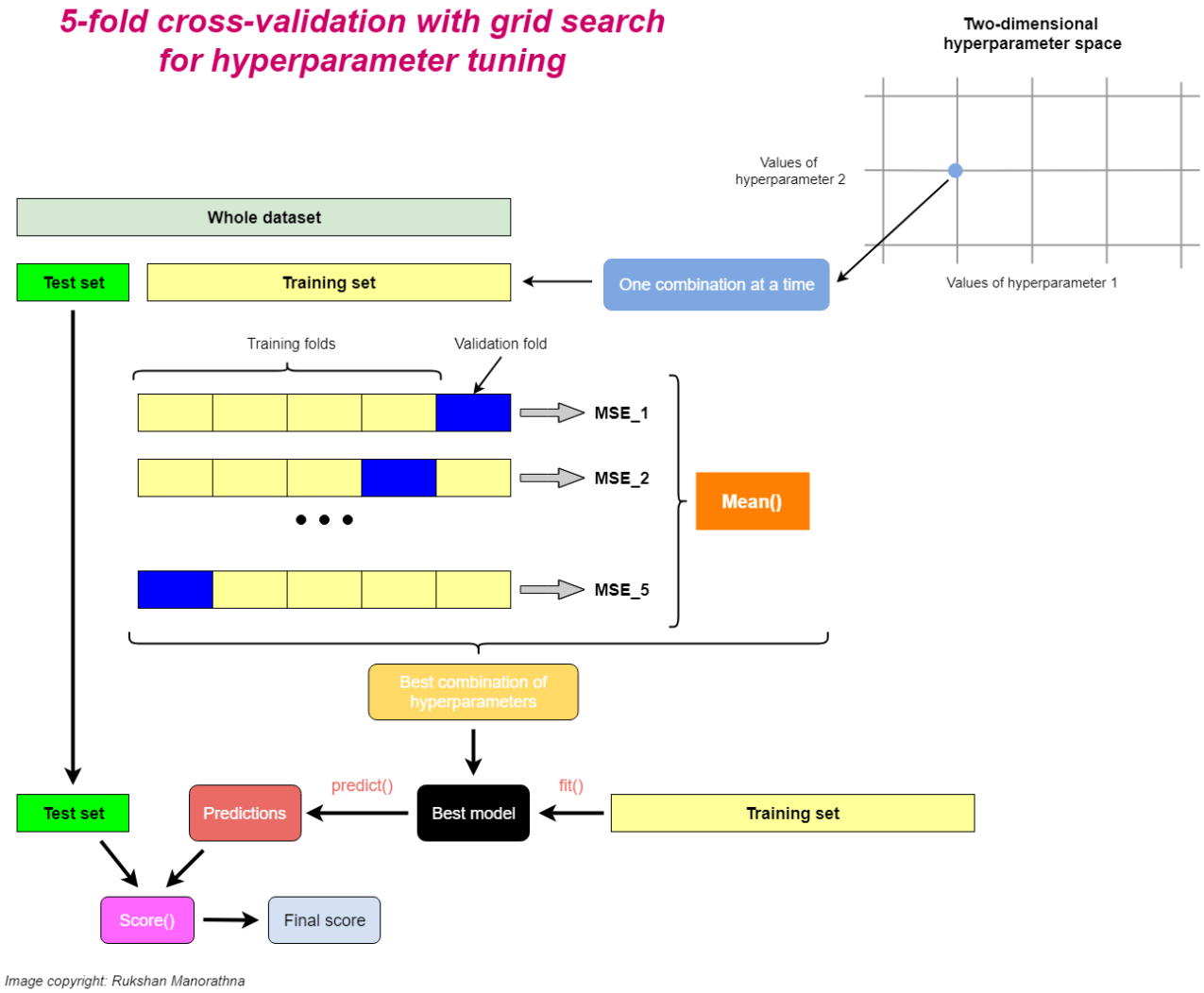

Image by Author

The reason behind fitting the best model to the whole training set after k-fold cross-validation is to provide more training samples to the learning algorithm of the best model. This usually results in a more accurate and robust model.

Lets' get our hands dirty by writing some code to perform k-fold cross-validation for hyperparameter tuning. This time, I will use the same **bike-sharing dataset**, but with a different learning algorithm. We will try to predict the correct number of bike rentals using a random forest regressor (learn how random forest works behind the scenes by reading **this article** written by me). This is still a regression task.

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=100, criterion="mse",
                           bootstrap=True, oob_score=True, n_jobs=2,
                           random_state=42)

# Defining 3-dimensional hyperparameter space as a Python dictionary
hyperparameter_space = {'max_depth':[None,4,6,8,10,12,15,20],
                        'min_samples_leaf':[1,2,4,6,8,10,20,30],
                        'max_features':['auto','sqrt','log2']}

from sklearn.model_selection import GridSearchCV
gs = GridSearchCV(rf, param_grid=hyperparameter_space ,
                  scoring="neg_mean_squared_error",
                  n_jobs=2, cv=5, return_train_score=True)

gs.fit(X_train, y_train)
print("Optimal hyperparameter combination: ", gs.best_params_)
print("Mean cross-validated MSE or training score of the best_estimator: ",
       np.sqrt(-gs.best_score_))
gs.best_estimator_.fit(X_train, y_train)
y_pred = gs.best_estimator_.predict(X_test)

from sklearn.metrics import mean_squared_error as MSE
rmse_test = np.sqrt(MSE(y_test, y_pred))
print("Test score: ", np.round(rmse_test, 2))
```

cv3.py hosted with ❤ by GitHub view raw

```
Optimal hyperparameter combination:  {'max_depth': 15, 'max_features': 'sqrt', 'min_samples_leaf': 1}
Mean cross-validated MSE or training score of the best_estimator:  659.4814164563437
Test score:  661.72
```

Here, we tune 3 hyperparameters called ***'max_depth'***, ***'min_samples_leaf'*** and ***'max_features'*** in **RandomForestRegressor**. So, the hyperparameter space is 3 dimensional so that each combination contains 3 values. The number of combinations is 192 (8 x 8 x 3). This is because ***max_depth*** contains **8** values, ***min_samples_leaf*** contains **8** values and ***max_features*** contains **3** values. This means we train 192 different models! Each combination is repeated 5 times in the 5-fold cross-validation process. So, the total number of iterations is 960 (192 x 5). But also note that each **RandomForestRegressor** has 100 decision trees. So, the total computation is 96,000 (960 x 100)! So, running the above code definitely takes about 5 minutes with 2 CPUs in the computer. If your computer has more CPUs (e.g. 8), you can set ***n_jobs=4*** to get faster results.

Here, the test score (RMSE) is 661.72 in y units. This number is much smaller than one (883.42) we previously obtained. Is RMSE of 661.72 still good? To find out this, let's see some information on bike rentals.

```
df['cnt'].describe()
```

```
count     731.000000
mean     4504.348837
std      1937.211452
min        22.000000
25%      3152.000000
50%      4548.000000
75%      5956.000000
max      8714.000000
Name: cnt, dtype: float64
```

With a range of 22 to 8714, a mean of 4504.35 and a standard deviation of 1937.21, an RMSE of 661.72 is very good. But it is not the best!

Thanks for reading!

This content was designed and created by **Rukshan Pramoditha**, the Author of **Data Science 365 Blog**.

2020–12–19