cross_validation_train_test

May 29, 2021

1 The framework and why do we need it

In the previous notebooks, we introduce some concepts regarding the evaluation of predictive models. While this section could be slightly redundant, we intend to go into details into the cross-validation framework.

Before we dive in, let's linger on the reasons for always having training and testing sets. Let's first look at the limitation of using a dataset without keeping any samples out.

To illustrate the different concepts, we will use the California housing dataset.

[1]:
```
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing(as_frame=True)
data, target = housing.data, housing.target
```

In this dataset, the aim is to predict the median value of houses in an area in California. The features collected are based on general real-estate and geographical information.

Therefore, the task to solve is different from the one shown in the previous notebook. The target to be predicted is a continuous variable and not anymore discrete. This task is called regression.

This, we will use a predictive model specific to regression and not to classification.

[2]:
```
print(housing.DESCR)
```

```
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

    :Number of Instances: 20640

    :Number of Attributes: 8 numeric, predictive attributes and the target

    :Attribute Information:
        - MedInc        median income in block
        - HouseAge      median house age in block
        - AveRooms      average number of rooms
```

```
        - AveBedrms     average number of bedrooms
        - Population    block population
        - AveOccup      average house occupancy
        - Latitude      house block latitude
        - Longitude     house block longitude

    :Missing Attribute Values: None

This dataset was obtained from the StatLib repository.
http://lib.stat.cmu.edu/datasets/

The target variable is the median house value for California districts.

This dataset was derived from the 1990 U.S. census, using one row per census
block group. A block group is the smallest geographical unit for which the U.S.
Census Bureau publishes sample data (a block group typically has a population
of 600 to 3,000 people).

It can be downloaded/loaded using the
:func:`sklearn.datasets.fetch_california_housing` function.

.. topic:: References

    - Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
      Statistics and Probability Letters, 33 (1997) 291-297
```

[3]:
```
data.head()
```

[3]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	\
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	

	Longitude
0	-122.23
1	-122.22
2	-122.24
3	-122.25
4	-122.25

To simplify future visualization, let's transform the prices from the dollar (\\)*rangetothethousanddollars*(*k*
) range.

[4]:
```
target *= 100
target.head()
```

[4]:
```
0    452.6
1    358.5
2    352.1
3    341.3
4    342.2
Name: MedHouseVal, dtype: float64
```

Note

If you want a deeper overview regarding this dataset, you can refer to the Appendix - Datasets description section at the end of this MOOC.

1.1 Training error vs testing error

To solve this regression task, we will use a decision tree regressor.

[5]:
```
from sklearn.tree import DecisionTreeRegressor

regressor = DecisionTreeRegressor(random_state=0)
regressor.fit(data, target)
```

[5]:
```
DecisionTreeRegressor(random_state=0)
```

After training the regressor, we would like to know its potential statistical performance once deployed in production. For this purpose, we use the mean absolute error, which gives us an error in the native unit, i.e. k\$.

[6]:
```
from sklearn.metrics import mean_absolute_error

target_predicted = regressor.predict(data)
score = mean_absolute_error(target, target_predicted)
print(f"On average, our regressor makes an error of {score:.2f} k$")
```

```
On average, our regressor makes an error of 0.00 k$
```

We get perfect prediction with no error. It is too optimistic and almost always revealing a methodological problem when doing machine learning.

Indeed, we trained and predicted on the same dataset. Since our decision tree was fully grown, every sample in the dataset is stored in a leaf node. Therefore, our decision tree fully memorized the dataset given during `fit` and therefore made no error when predicting.

This error computed above is called the **empirical error** or **training error**.

Note

In this MOOC, we will consistently use the term “training error”.

We trained a predictive model to minimize the training error but our aim is to minimize the error on data that has not been seen during training.

This error is also called the **generalization error** or the "true" **testing error**.

Note

In this MOOC, we will consistently use the term "testing error".

Thus, the most basic evaluation involves:

- splitting our dataset into two subsets: a training set and a testing set;
- fitting the model on the training set;
- estimating the training error on the training set;
- estimating the testing error on the testing set.

So let's split our dataset.

[7]:
```
from sklearn.model_selection import train_test_split

data_train, data_test, target_train, target_test = train_test_split(
    data, target, random_state=0)
```

Then, let's train our model.

[8]:
```
regressor.fit(data_train, target_train)
```

[8]:
```
DecisionTreeRegressor(random_state=0)
```

Finally, we estimate the different types of errors. Let's start by computing the training error.

[9]:
```
target_predicted = regressor.predict(data_train)
score = mean_absolute_error(target_train, target_predicted)
print(f"The training error of our model is {score:.2f} k$")
```

```
The training error of our model is 0.00 k$
```

We observe the same phenomena as in the previous experiment: our model memorized the training set. However, we now compute the testing error.

[10]:
```
target_predicted = regressor.predict(data_test)
score = mean_absolute_error(target_test, target_predicted)
print(f"The testing error of our model is {score:.2f} k$")
```

```
The testing error of our model is 47.28 k$
```

This testing error is actually about what we would expect from our model if it was used in a production environment.

1.2 Stability of the cross-validation estimates

When doing a single train-test split we don't give any indication regarding the robustness of the evaluation of our predictive model: in particular, if the test set is small, this estimate of the testing

error will be unstable and wouldn't reflect the "true error rate" we would have observed with the same model on an unlimited amount of test data.

For instance, we could have been lucky when we did our random split of our limited dataset and isolated some of the easiest cases to predict in the testing set just by chance: the estimation of the testing error would be overly optimistic, in this case.

Cross-validation allows estimating the robustness of a predictive model by repeating the splitting procedure. It will give several training and testing errors and thus some **estimate of the variability of the model statistical performance**.

There are different cross-validation strategies, for now we are going to focus on one called "shuffle-split". At each iteration of this strategy we:

- randomly shuffle the order of the samples of a copy of the full dataset;
- split the shuffled dataset into a train and a test set;
- train a new model on the train set;
- evaluate the testing error on the test set.

We repeat this procedure `n_splits` times. Using `n_splits=40` means that we will train 40 models in total and all of them will be discarded: we just record their statistical performance on each variant of the test set.

To evaluate the statistical performance of our regressor, we can use `sklearn.model_selection.cross_validate` with a `sklearn.model_selection.ShuffleSplit` object:

[11]:
```
from sklearn.model_selection import cross_validate
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=40, test_size=0.3, random_state=0)
cv_results = cross_validate(
    regressor, data, target, cv=cv, scoring="neg_mean_absolute_error")
```

The results `cv_results` are stored into a Python dictionary. We will convert it into a pandas dataframe to ease visualization and manipulation.

[12]:
```
import pandas as pd

cv_results = pd.DataFrame(cv_results)
cv_results.head()
```

[12]:

	fit_time	score_time	test_score
0	0.133409	0.002485	-46.909797
1	0.133467	0.002443	-46.421170
2	0.132288	0.002619	-47.411089
3	0.133599	0.002537	-44.319824
4	0.130700	0.002498	-47.607875

Tip

A score is a metric for which higher values mean better results. On the contrary, an error is a metric for which lower values mean better results. The parameter scoring in cross_validate always expect a function that is a score.

To make it easy, all error metrics in scikit-learn, like mean_absolute_error, can be transformed into a score to be used in cross_validate. To do so, you need to pass a string of the error metric with an additional neg_ string at the front to the parameter scoring; for instance scoring="neg_mean_absolute_error". In this case, the negative of the mean absolute error will be computed which would be equivalent to a score.

Let us revert the negation to get the actual error:

[13]:
```
cv_results["test_error"] = -cv_results["test_score"]
```

Let's check the results reported by the cross-validation.

[14]:
```
cv_results.head(10)
```

[14]:

	fit_time	score_time	test_score	test_error
0	0.133409	0.002485	-46.909797	46.909797
1	0.133467	0.002443	-46.421170	46.421170
2	0.132288	0.002619	-47.411089	47.411089
3	0.133599	0.002537	-44.319824	44.319824
4	0.130700	0.002498	-47.607875	47.607875
5	0.132588	0.002569	-45.901300	45.901300
6	0.133336	0.002481	-46.572767	46.572767
7	0.133233	0.002484	-46.194585	46.194585
8	0.133414	0.002421	-45.590236	45.590236
9	0.134270	0.002467	-45.727998	45.727998

We get timing information to fit and predict at each cross-validation iteration. Also, we get the test score, which corresponds to the testing error on each of the splits.

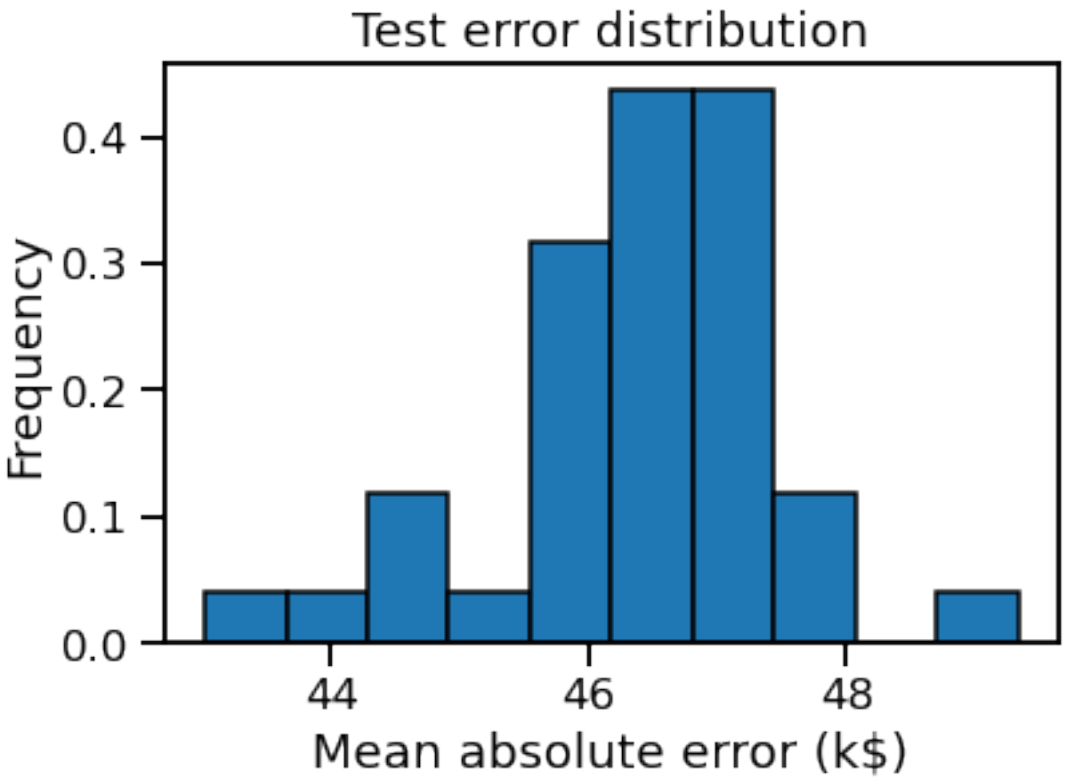
[15]:
```
len(cv_results)
```

[15]: 40

We get 40 entries in our resulting dataframe because we performed 40 splits. Therefore, we can show the testing error distribution and thus, have an estimate of its variability.

[16]:
```
import matplotlib.pyplot as plt

cv_results["test_error"].plot.hist(bins=10, edgecolor="black", density=True)
plt.xlabel("Mean absolute error (k$)")
_ = plt.title("Test error distribution")
```



We observe that the testing error is clustered around 47 k\$ and ranges from 43 k\$ to 50 k\$.

[17]:
```
print(f"The mean cross-validated testing error is: "
      f"{cv_results['test_error'].mean():.2f} k$")
```

```
The mean cross-validated testing error is: 46.36 k$
```

[18]:
```
print(f"The standard deviation of the testing error is: "
      f"{cv_results['test_error'].std():.2f} k$")
```

```
The standard deviation of the testing error is: 1.17 k$
```

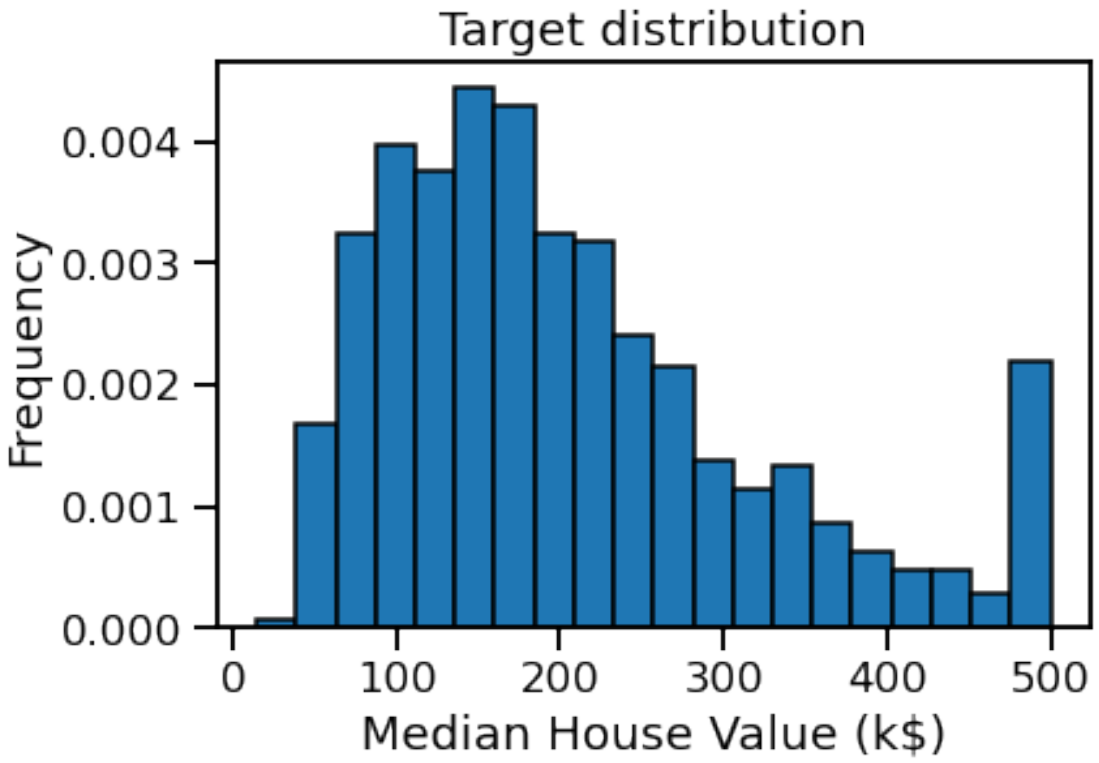
Note that the standard deviation is much smaller than the mean: we could summarize that our cross-validation estimate of the testing error is 46.36 +/- 1.17 k\$.

If we were to train a single model on the full dataset (without cross-validation) and then later had access to an unlimited amount of test data, we would expect its true testing error to fall close to that region.

While this information is interesting in itself, it should be contrasted to the scale of the natural variability of the vector `target` in our dataset.

Let us plot the distribution of the target variable:

[19]:
```
target.plot.hist(bins=20, edgecolor="black", density=True)
plt.xlabel("Median House Value (k$)")
_ = plt.title("Target distribution")
```



[20]:
```
print(f"The standard deviation of the target is: {target.std():.2f} k$")
```

```
The standard deviation of the target is: 115.40 k$
```

The target variable ranges from close to 0 k\$ up to 500 k\$ and, with a standard deviation around 115 k\$.

We notice that the mean estimate of the testing error obtained by cross-validation is a bit smaller than the natural scale of variation of the target variable. Furthermore, the standard deviation of the cross validation estimate of the testing error is even smaller.

This is a good start, but not necessarily enough to decide whether the generalization performance is good enough to make our prediction useful in practice.

We recall that our model makes, on average, an error around 47 k\$. *With this information and looking at the target distribution, such an error might be acceptable when predicting hous* . However, it would be an issue with a house with a value of 50 k\$. Thus, this indicates that our metric (Mean Absolute Error) is not ideal.

We might instead choose a metric relative to the target value to predict: the mean absolute percentage error would have been a much better choice.

But in all cases, an error of 47 k\$ might be too large to automatically use our model to tag house values without expert supervision.

1.3 More detail regarding `cross_validate`

During cross-validation, many models are trained and evaluated. Indeed, the number of elements in each array of the output of `cross_validate` is a result from one of these `fit`/`score` procedures. To make it explicit, it is possible to retrieve theses fitted models for each of the splits/folds by passing the option `return_estimator=True` in `cross_validate`.

[21]:
```
cv_results = cross_validate(regressor, data, target, return_estimator=True)
cv_results
```

[21]:
```
{'fit_time': array([0.1583066 , 0.15358043, 0.15300274, 0.15377641,
0.14847398]),
 'score_time': array([0.00181603, 0.00184298, 0.00179005, 0.00170827,
0.00186396]),
 'estimator': [DecisionTreeRegressor(random_state=0),
  DecisionTreeRegressor(random_state=0),
  DecisionTreeRegressor(random_state=0),
  DecisionTreeRegressor(random_state=0),
  DecisionTreeRegressor(random_state=0)],
 'test_score': array([0.26291527, 0.41947109, 0.44492564, 0.23357874,
0.40788361])}
```

[22]:
```
cv_results["estimator"]
```

[22]:
```
[DecisionTreeRegressor(random_state=0),
 DecisionTreeRegressor(random_state=0),
 DecisionTreeRegressor(random_state=0),
 DecisionTreeRegressor(random_state=0),
 DecisionTreeRegressor(random_state=0)]
```

The five decision tree regressors corresponds to the five fitted decision trees on the different folds. Having access to these regressors is handy because it allows to inspect the internal fitted parameters of these regressors.

In the case where you only are interested in the test score, scikit-learn provide a `cross_val_score` function. It is identical to calling the `cross_validate` function and to select the `test_score` only (as we extensively did in the previous notebooks).

[23]:
```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(regressor, data, target)
scores
```

[23]:
```
array([0.26291527, 0.41947109, 0.44492564, 0.23357874, 0.40788361])
```

1.4 Summary

In this notebook, we saw:

- the necessity of splitting the data into a train and test set;
- the meaning of the training and testing errors;
- the overall cross-validation framework with the possibility to study statistical performance variations;