ESTIMATION OF FUTURE MODEL PERFORMANCE (MODEL EVALUATION)

ESTIMATION OF FUTURE MODEL PERFORMANCE (EVALUATION)

1. Performance: measure of how well your model does.
   - E.g. classification accuracy for classification tasks

$$Accuracy = \frac{1}{n}\sum_{k=1}^{n}(y_k == \hat{y}_k)$$

   - E.g. Root Mean Squared Error (RMSE) for regression

$$RMSE = \sqrt{\frac{1}{n}\sum_{k=1}^{n}(y_k - \hat{y}_k)^2}$$

   Performance metrics on training dataset are biased

2. Any performance evaluation of the model on the same data that was used for training, is going to be optimistically biased
   - In the same way than evaluating a student with an exam that contains the same problems the student used for learning: perhaps the student is just memorizing the problems s/he used for learning
3. The model must generalize beyond the training data and work well for new, unseen, instances (different to the ones in the training data)
4. The issue is then: how to estimate the future performance of the model (new/unseen data)?
5. Reasons for wanting to know this estimation:
   - Your company probably wants to know how well your model is going to perform in the future.
   - Or, if you are participating in a data science competition, you probably want to know how well your model will do.

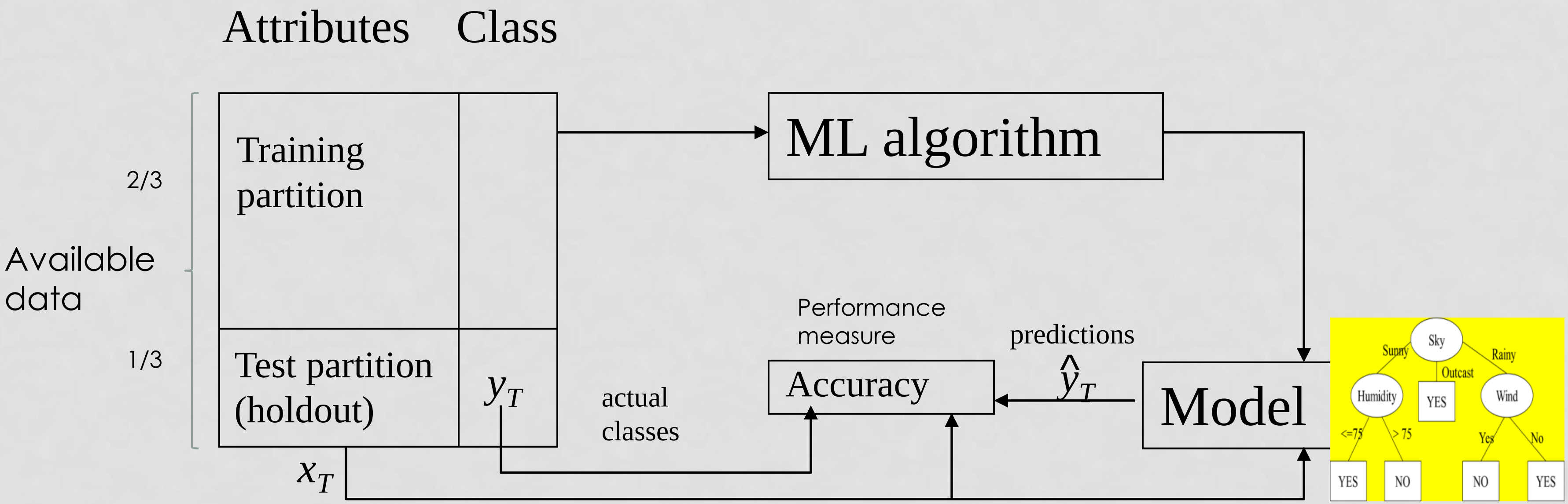
MODEL EVALUATION

- In summary, we now want two results/products out of machine learning:
  - A model
  - An estimation of its future performance: its performance on new (unseen) data
- There are several methods for estimating future performance (model evaluation), but the most widely used are:
  - Train / test (holdout) It is more appropiate when data size is big enough
  - Crossvalidation

- Bootstrap (Used for Confidence Intervals)

TRAIN / TEST EVALUATION METHOD (A.K.A. HOLDOUT METHOD)

Rule: don't evaluate a model with the same data used for training it

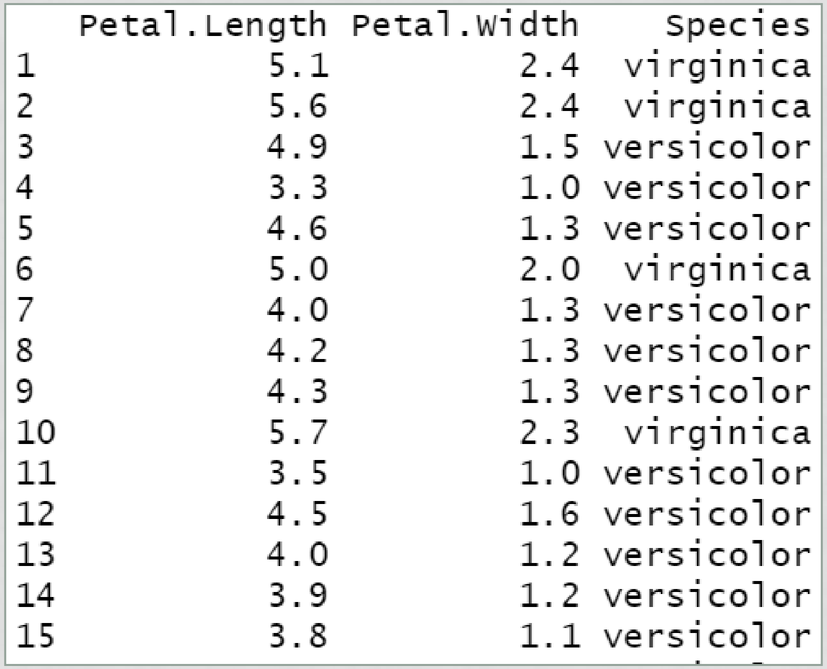


- Available data should be randomly shuffled before splitting! (**if** data are i.i.d.)
- The testing partition must be representative of the problem (also the training partition).

$$Accuracy = \frac{1}{n}\sum_{k=1}^{n} y_k == \hat{y}_k$$

TRAIN / TEST EVALUATION METHOD (A.K.A. HOLDOUT METHOD)

- Data is randomly shuffled **only if data i.i.d**. (independently identically distributed)
  - Example of i.i.d. data: Iris dataset. i.i.d. implies that:
    - Instances are not correlated (an instance appearing in the dataset does not make another instance more likely)
    - Any ordering of instances is equally likely (order does not matter: exchangeability)



```
   Petal.Length Petal.Width    Species
1           5.1         2.4  virginica
2           5.6         2.4  virginica
3           4.9         1.5 versicolor
4           3.3         1.0 versicolor
5           4.6         1.3 versicolor
6           5.0         2.0  virginica
7           4.0         1.3 versicolor
8           4.2         1.3 versicolor
9           4.3         1.3 versicolor
10          5.7         2.3  virginica
11          3.5         1.0 versicolor
12          4.5         1.6 versicolor
13          4.0         1.2 versicolor
14          3.9         1.2 versicolor
15          3.8         1.1 versicolor
```

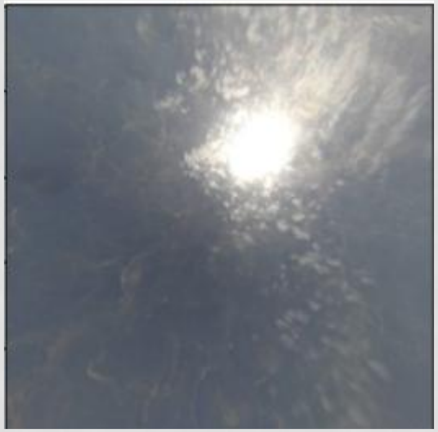
TRAIN / TEST EVALUATION METHOD (A.K.A. HOLDOUT METHOD)

- Data is randomly shuffled if i.i.d. (independently identically distributed)
  - Example of i.i.d. data: Iris dataset. i.i.d. implies that:
    - Instances are not correlated (an instance appearing in the dataset does not make another instance more likely)
    - Any ordering is equally likely
- However, not all data are i.i.d.

  We cant randomly shuffled our data because then our metrics would be too optimistic

  - For instance, if there is temporal ordering, data is not i.i.d.
    - E.g. A cloud classification problem with cloud pictures (instances) taken every minute
    - E.g. An energy forecasting problem, where energy production must be forecast every minute (training instances every minute)

E.G. CLOUD CLASSIFICATION PROBLEM



2015-08-11 10:15:00

2015-08-11 10:20:00

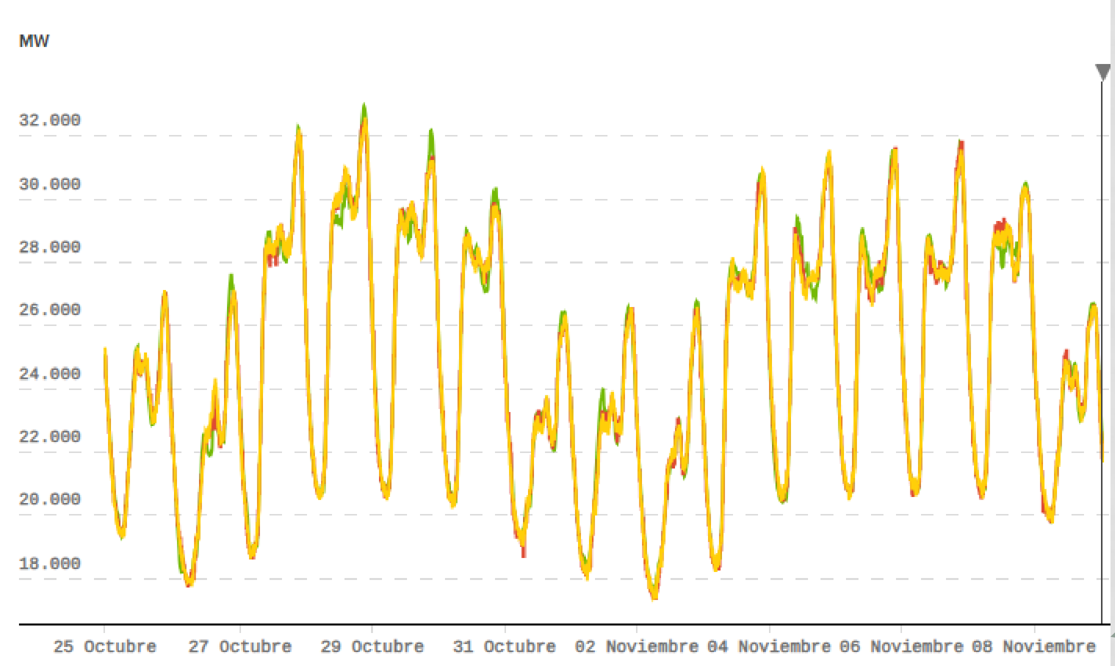
2015-08-11 10:25:00

2015-08-11 10:35:00

- Instances (images) close in time are correlated (similar)
- If data were randomly shuffled, *almost* the same image could (likely) be assigned to the training and test partitions, and therefore the evaluation would be overly optimistic, because of correlations of instances close in time
- Posible solutions:
  - Group split: for instance, groups of images during 2 hours are assigned either to the train partition or to the test partition
  - If we have large amounts of data (entire years): we could use for instance, 2 years for training and 1 year for test.

BLOCKS TESTING (All of the clouds from the same day go together to training or testing dataset)

E.G.: ENERGY FORECASTING PROBLEM



(every hour)

- Same problem as before (correlation between instances close in time)
- Additionally, when working with time series, models …
  - can be autoregressive
  - can use lagged features

$$y_t = f(y_{t-1}, y_{t-2}, \ldots, y_{t-p})$$

It depends on the past ouput

- Then again, instances cannot be randomly shuffled because all orderings of instances are not equally likely
- Solution: keep time ordering (e.g. use the "past" for training and the "future" for test)

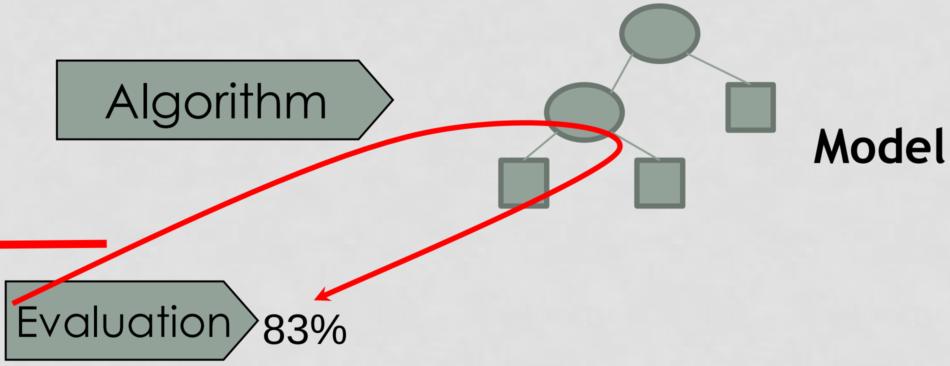
Train/test:
Now, in addition to training the model with the training partition, the model is evaluated with the test partition.

Available data

Training

Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No

Test



- Now, we get both the model **and** an estimation of its future performance
- Then, we can use the model.

Available data

Training

Test

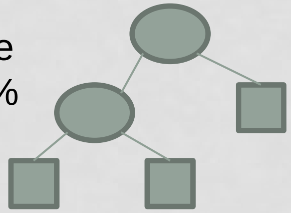
Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No

Sky	Tempe rature	Humidity	Wind	Tennis
Sun	60	65	No	?????

?

Make predictions

Estimation of future performance = 83%

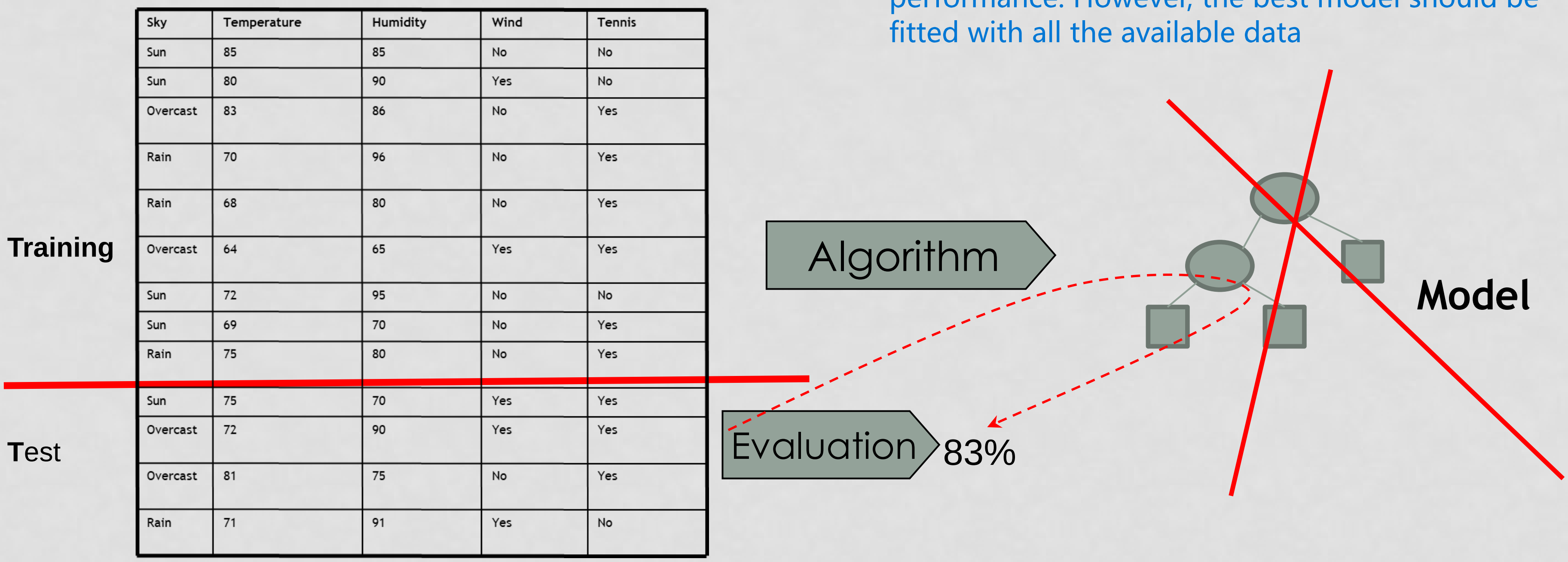


Model

Yes

- However, it is common practice that, after the estimation of future performance (83%) has been obtained …
- The model used to obtain it is discarded and …

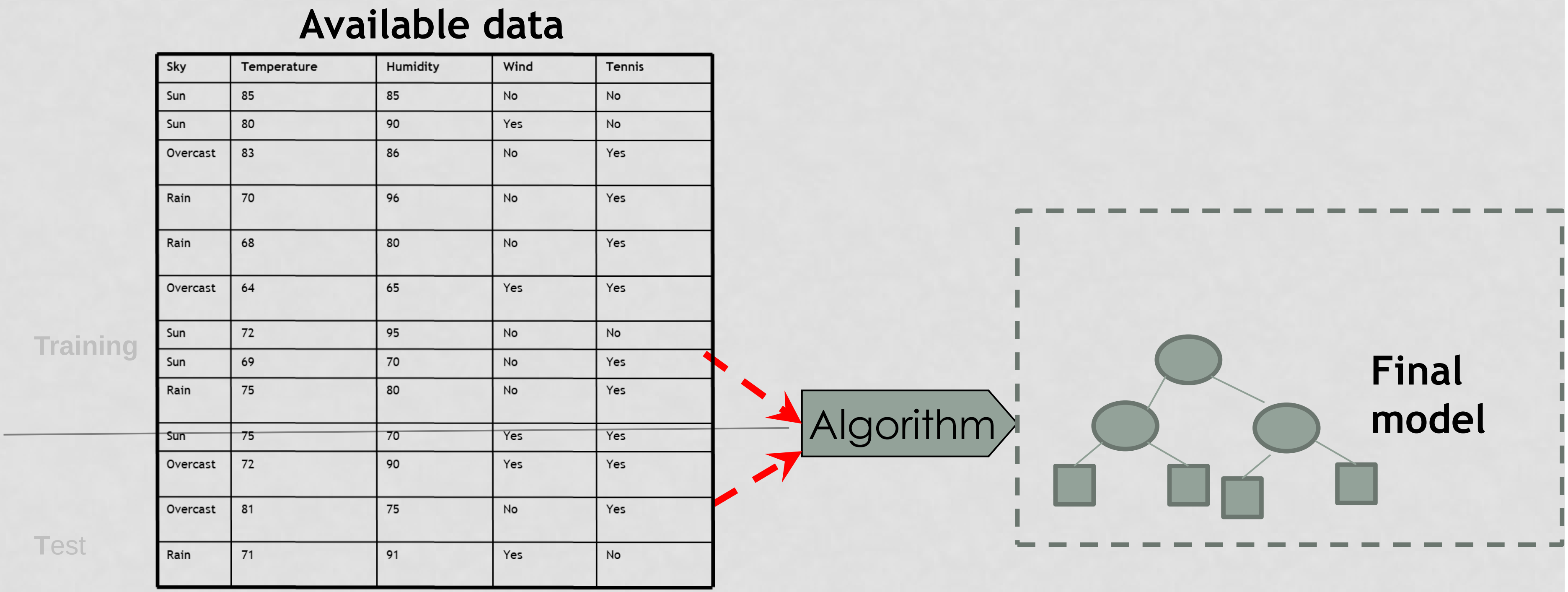
Available data



Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No

We just use this model for obtaining the performance. However, the best model should be fitted with all the available data

- … and a **final model** is trained with the **complete dataset** (available data = training and test partitions).
- The reason is that the more data is used to train the model, the better the model should be (at least, it shouldn't be worse).



Available data

Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No

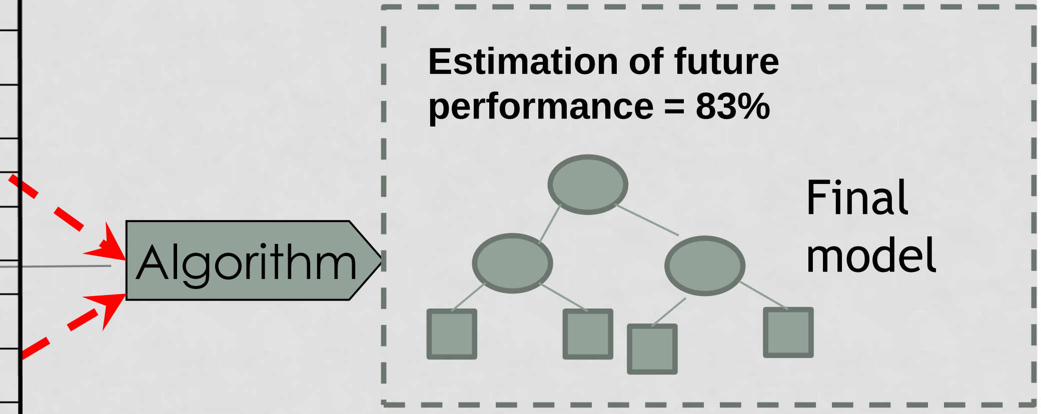
- … and a **final model** is trained with the **complete dataset** (available data = training and test partitions).
- The reason is that the more data is used to train the model, the better the model should be (at least, it shouldn’t be worse).
- It is considered that the evaluation computed previously (83%), is also a good estimation of this new final model, and it is kept.
- In fact, it is considered that 83% is a **pessimistic evaluation**, because the final model is trained with a larger dataset than the one used previously for evaluation

Available data

Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No

Training

Test



Estimation of future performance = 83%

- Then, we can use this final model for making predictions

Available data

Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No

Training

Test

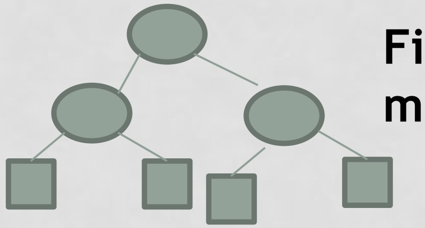
Algorithm

Sky	Tempe rature	Humidity	Wind	Tennis
Sun	60	65	No	?????

?

Make predictions

Estimation of future performance = 83%

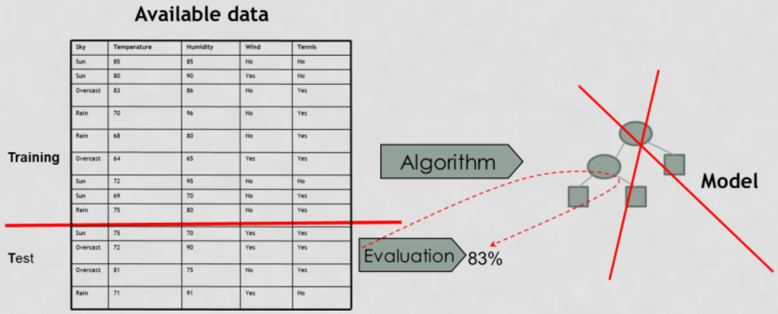


Final model

Yes

Summary

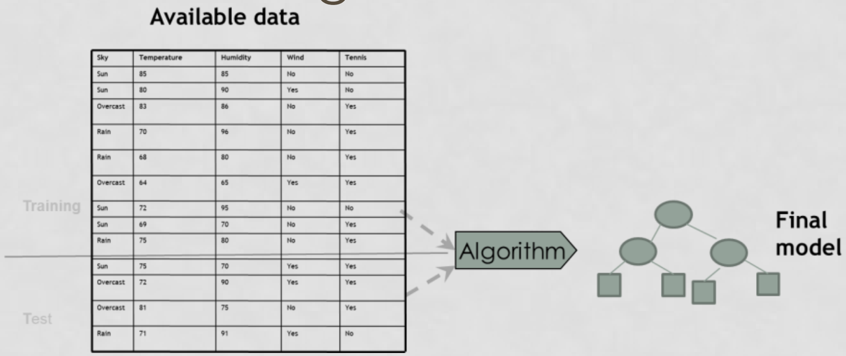
1. The goal of estimation of future performance / estimation of performance on new data / model evaluation is not getting a model, but getting a performance measure.



Available data

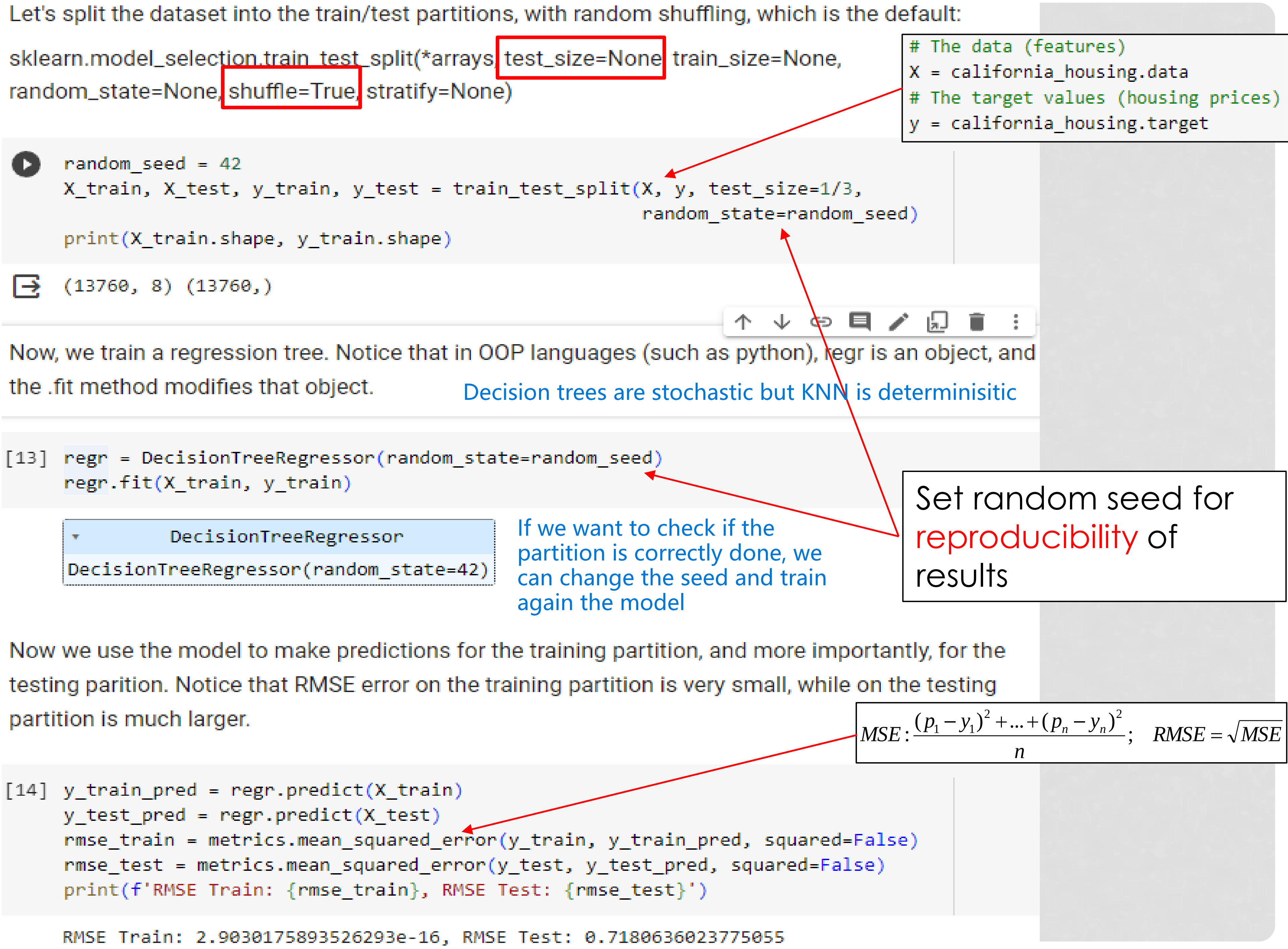
Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No

2. The final model is always trained using all available data.
   - In fact, if you are not interested in model evaluation, this is exactly what you would do: use all data for training the model.



Available data

Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No



Let's split the dataset into the train/test partitions, with random shuffling, which is the default:

sklearn.model_selection.train_test_split(*arrays, test_size=None, train_size=None, random_state=None, shuffle=True, stratify=None)

```
# The data (features)
X = california_housing.data
# The target values (housing prices)
y = california_housing.target
```

```
random_seed = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/3,
                                                    random_state=random_seed)

print(X_train.shape, y_train.shape)
```

```
(13760, 8) (13760,)
```

Now, we train a regression tree. Notice that in OOP languages (such as python), regr is an object, and the .fit method modifies that object.

Decision trees are stochastic but KNN is determinisitic

```
[13] regr = DecisionTreeRegressor(random_state=random_seed)
     regr.fit(X_train, y_train)
```

```
DecisionTreeRegressor
DecisionTreeRegressor(random_state=42)
```

If we want to check if the partition is correctly done, we can change the seed and train again the model

Set random seed for reproducibility of results

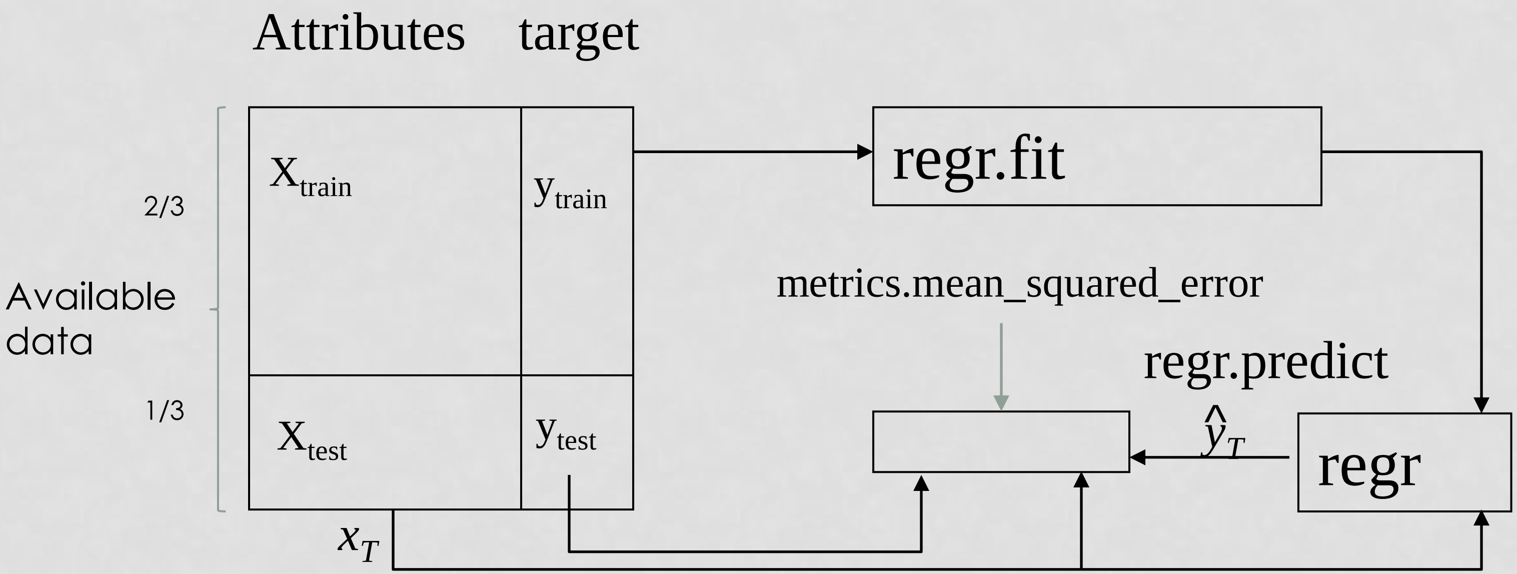
Now we use the model to make predictions for the training partition, and more importantly, for the testing parition. Notice that RMSE error on the training partition is very small, while on the testing partition is much larger.

$$MSE:\frac{(p_1-y_1)^2+...+(p_n-y_n)^2}{n};\quad RMSE=\sqrt{MSE}$$

```
[14] y_train_pred = regr.predict(X_train)
     y_test_pred = regr.predict(X_test)
     rmse_train = metrics.mean_squared_error(y_train, y_train_pred, squared=False)
     rmse_test = metrics.mean_squared_error(y_test, y_test_pred, squared=False)
     print(f'RMSE Train: {rmse_train}, RMSE Test: {rmse_test}')
```

```
RMSE Train: 2.9030175893526293e-16, RMSE Test: 0.7180636023775055
```

TRAIN / TEST EVALUATION METHOD (A.K.A. HOLDOUT METHOD)



Attributes target

X_{train} y_{train}

2/3

Available data

1/3

X_{test} y_{test}

x_T

regr.fit

metrics.mean_squared_error

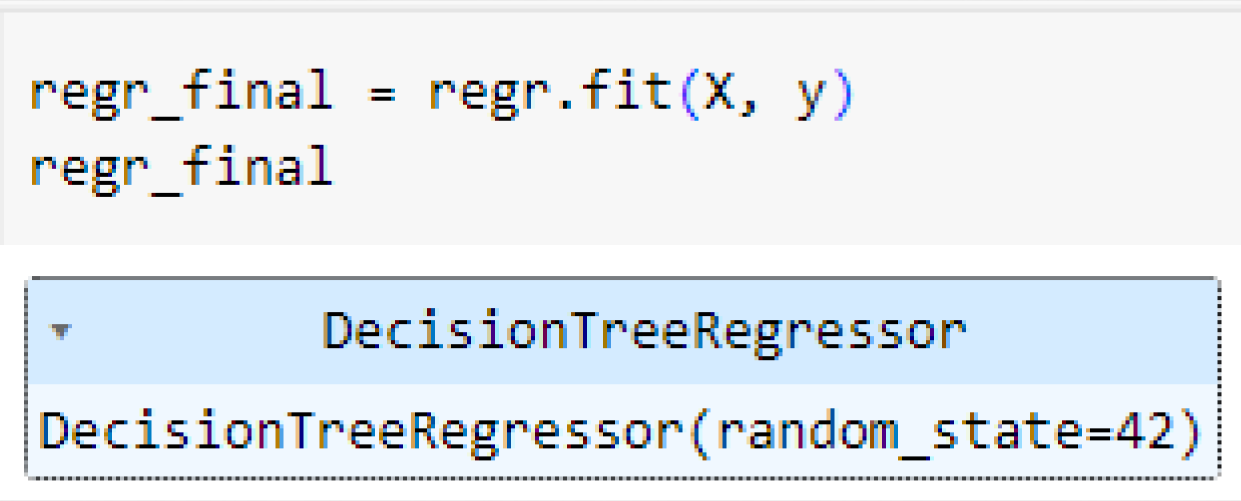
regr.predict

$\hat{y}_T$

regr

TRAIN-TEST IN SCIKITLEARN

Training the **final model** with the complete dataset: fitting regr again with (X, y)



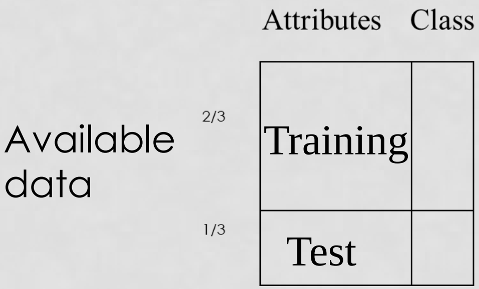
```
regr_final = regr.fit(X, y)
regr_final
```

```
▾        DecisionTreeRegressor
DecisionTreeRegressor(random_state=42)
```

Its estimation of future performance is pessimistic

On the size of the test partition: Estimating confidence intervals for accuracy

- 2/3 vs. 1/3 for train and test is arbitrary but commonly used.



- Good for thousands of instances on the testing partition. Probably, for millions of instances, 5% for test should be enough.
- Dilemma:
  - The larger the test, the more accurate model evaluation
  - But then, fewer instances are available for training the model

On the size of the test partition: Estimating confidence intervals for accuracy

- In order to estimate a reasonable size for the test partition, let's compute a **confidence interval** that contains the **true accuracy** with some high probability. Ej: [80% 85%]
  - The true accuracy of the model is the one we would compute on a (possibly) infinite dataset

We cant compute the true accuracy

- If the confidence interval is too wide, that means that our knowledge of the true accuracy is very uncertain, and therefore the testing partition should be larger
- If the confidence interval is narrow, then we are quite certain about the true accuracy, and therefore the size of the testing partition is right.
- If the size of the test set cannot be changed, at least we have an estimation of the reliability of accuracy (the confidence interval)

On the size of the test partition: Estimating confidence intervals for accuracy

- Let's suppose that the test partition contains N instances, and that the **empirical accuracy** of the model on the testing partition is $\hat{f}$

- If the true accuracy is *f* ... ($N \rightarrow \infty \Rightarrow \hat{f} \rightarrow f$)
- ... then the number of correct predictions of the model on the test partition ($\mathrm{N} * \hat{f}$) follows a binomial distribution with probability = *f*
  - Similar to: if the true probability of heads for a coin is *f=0.5*, then the number of heads (in N trials) follows a binomial distribution with probability *f*.

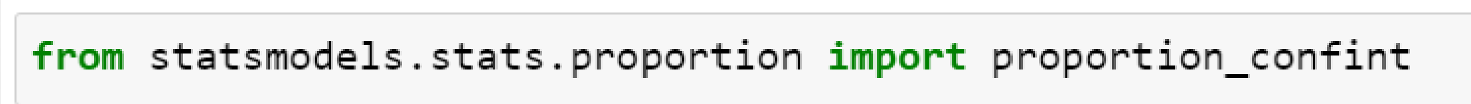
On the size of the test partition: Estimating confidence intervals for accuracy

- Let's suppose that the test partition contains N instances, and that the **empirical accuracy** of the model on the testing partition is $\hat{f}$

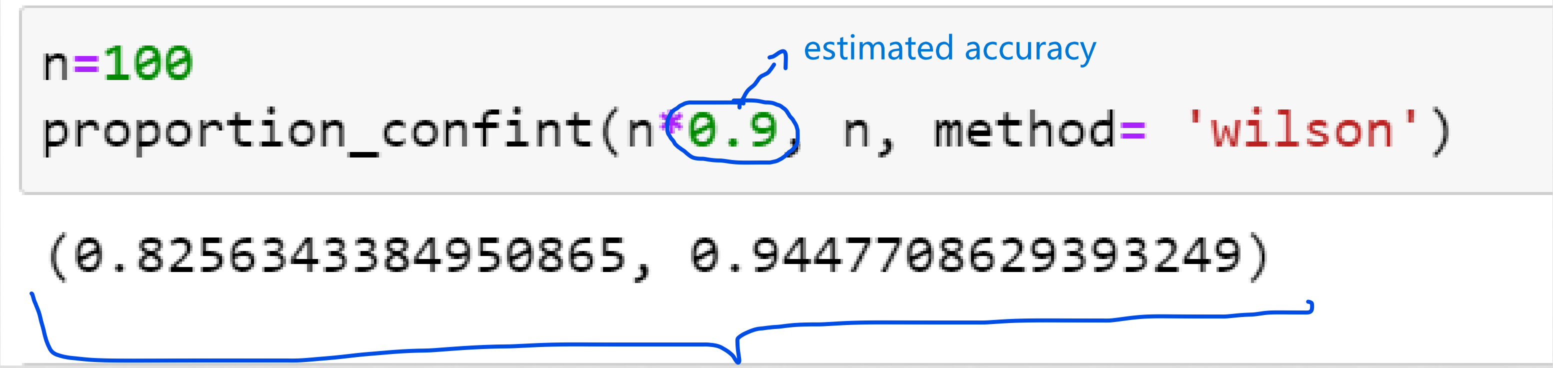
- If the true accuracy is *f* ... ($N \to \infty \Rightarrow \hat{f} \to f$)
- ... then the number of correct predictions of the model on the test partition ($\mathrm{N} * \hat{f}$) follows a binomial distribution with probability = *f*
- and confidence intervals around $\hat{f}$ can be estimated like this:
  - f ∊ [$\hat{f}$-l, $\hat{f}$+u] with 95% probability

On the size of the test partition: Estimating confidence intervals for accuracy

- For example, if $\hat{f} = 0.9$, estimated on some test set T, and T contains 100 instances, then the 95% confidence interval is [0.83 0.95]



```
from statsmodels.stats.proportion import proportion_confint
```



```
n=100
proportion_confint(n*0.9, n, method= 'wilson')
```

estimated accuracy

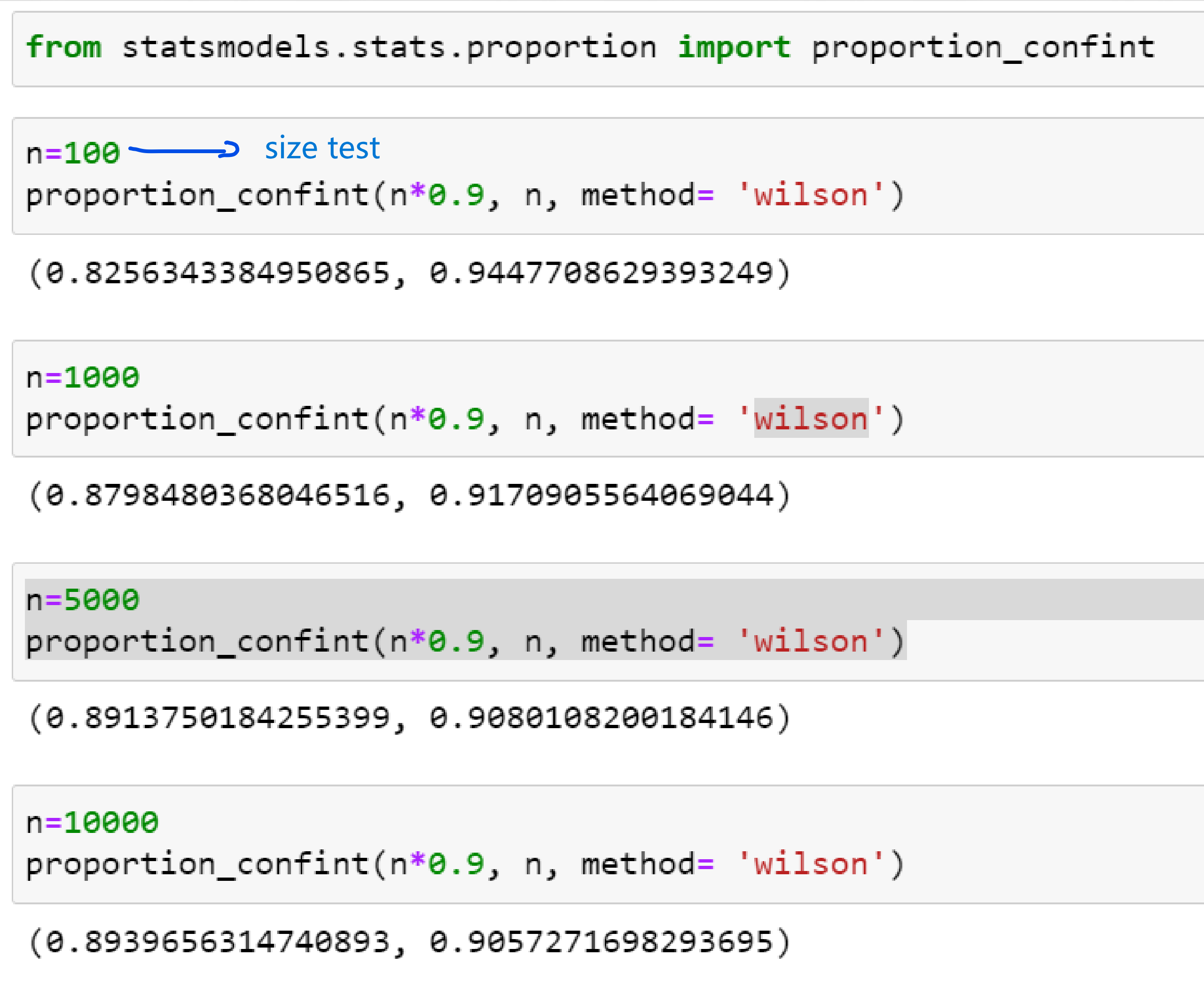
```
(0.8256343384950865, 0.9447708629393249)
```

Confidence interval for the accuracy

For includimng the true accuracy

method="beta", very slightly more accurate

RELATION BETWEEN THE WIDTH OF THE CONFIDENCE INTERVAL AND THE SIZE OF THE TEST PARTITION



```
from statsmodels.stats.proportion import proportion_confint
```

```
n=100
proportion_confint(n*0.9, n, method= 'wilson')
```

size test

```
(0.8256343384950865, 0.9447708629393249)
```

```
n=1000
proportion_confint(n*0.9, n, method= 'wilson')
```

```
(0.8798480368046516, 0.9170905564069044)
```

```
n=5000
proportion_confint(n*0.9, n, method= 'wilson')
```

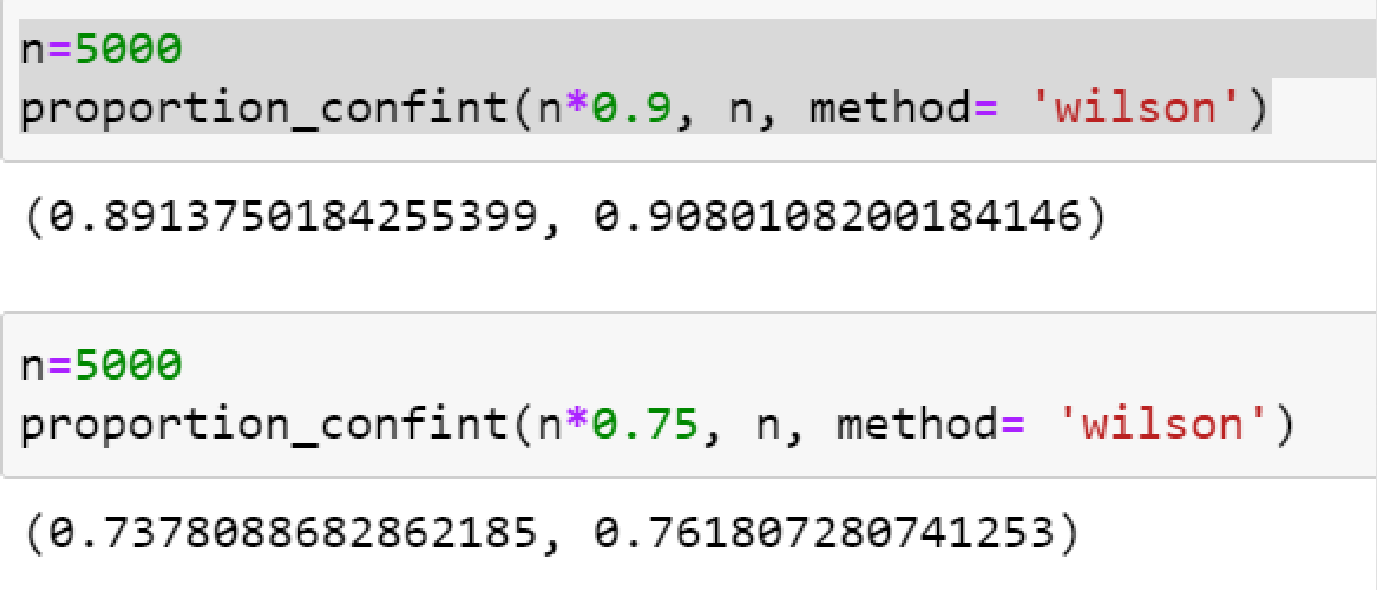
```
(0.8913750184255399, 0.9080108200184146)
```

```
n=10000
proportion_confint(n*0.9, n, method= 'wilson')
```

```
(0.8939656314740893, 0.9057271698293695)
```

RELATION BETWEEN THE WIDTH OF THE CONFIDENCE INTERVAL AND THE EMPIRICAL ACCURACY

The size of the confidence interval also depends on $\hat{f}$ (the larger $\hat{f}$, the narrower the interval)



```
n=5000
proportion_confint(n*0.9, n, method= 'wilson')
```

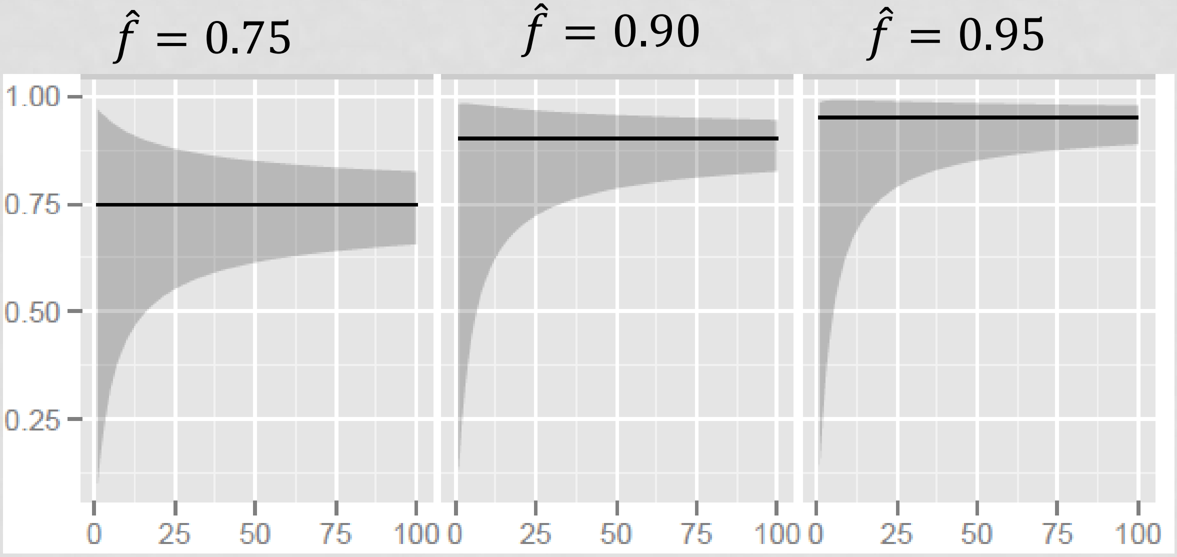
```
(0.8913750184255399, 0.9080108200184146)
```

```
n=5000
proportion_confint(n*0.75, n, method= 'wilson')
```

```
(0.7378088682862185, 0.761807280741253)
```

RELATION BETWEEN THE WIDTH OF THE CONFIDENCE INTERVAL AND THE EMPIRICAL ACCURACY

- The confidence interval size depends on $\hat{f}$ and N



$\hat{f} = 0.75$ $\hat{f} = 0.90$ $\hat{f} = 0.95$

- More information: Beleites, C., Neugebauer, U., Bocklitz, T., Krafft, C., & Popp, J. (2013). Sample size planning for classification models. *Analytica chimica acta*, *760*, 25-33.

https://www.sciencedirect.com/science/article/pii/S0003267012016479?via%3Dihub

- And R code:

https://ars.els-cdn.com/content/image/1-s2.0-S0003267012016479-mmc3.pdf

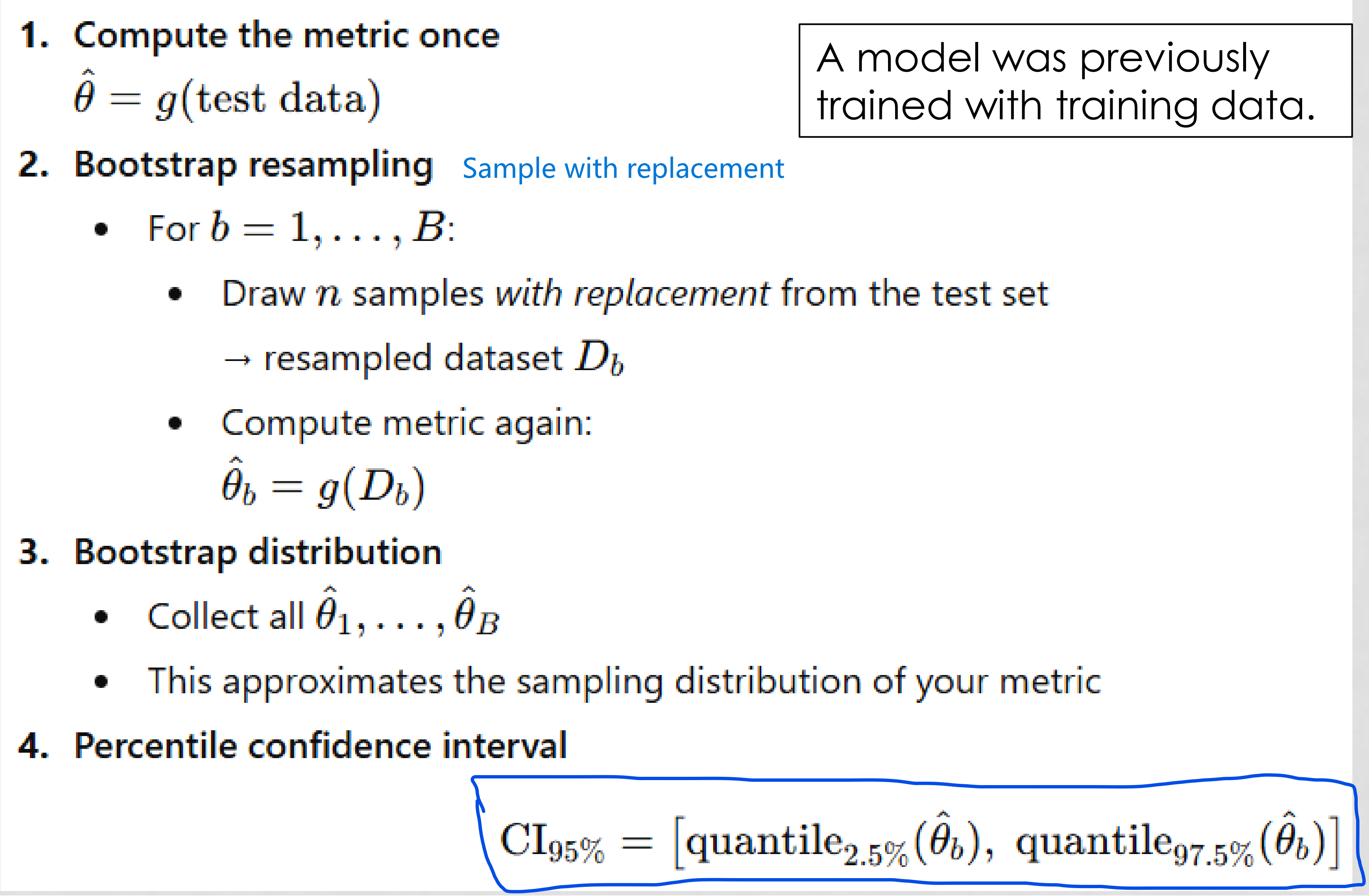
Estimating confidence intervals for other metrics

- Limitations:
  - Method valid for 2-classes. For 3 classes, Cis for each of the classes can be computed.
  - Method valid for the accuracy metric
  - Other classification metrics? (balanced accuracy, recall, F1, ...)
  - Regression metrics? (RMSE, MAE, ...)

You can compute a global accuracy. You need to compute 3 different accuracies for each class

ANOTHER WAY OF PERFORMING CONFIDENCE INTERVALS

Estimating confidence intervals for any metric *g*: bootstrap method (test resample)



A model was previously trained with training data.

1. **Compute the metric once**

   $\hat{\theta} = g(\text{test data})$

2. **Bootstrap resampling** Sample with replacement
   - For $b = 1, \dots, B$:
     - Draw n samples *with replacement* from the test set

       → resampled dataset D_b
     - Compute metric again:

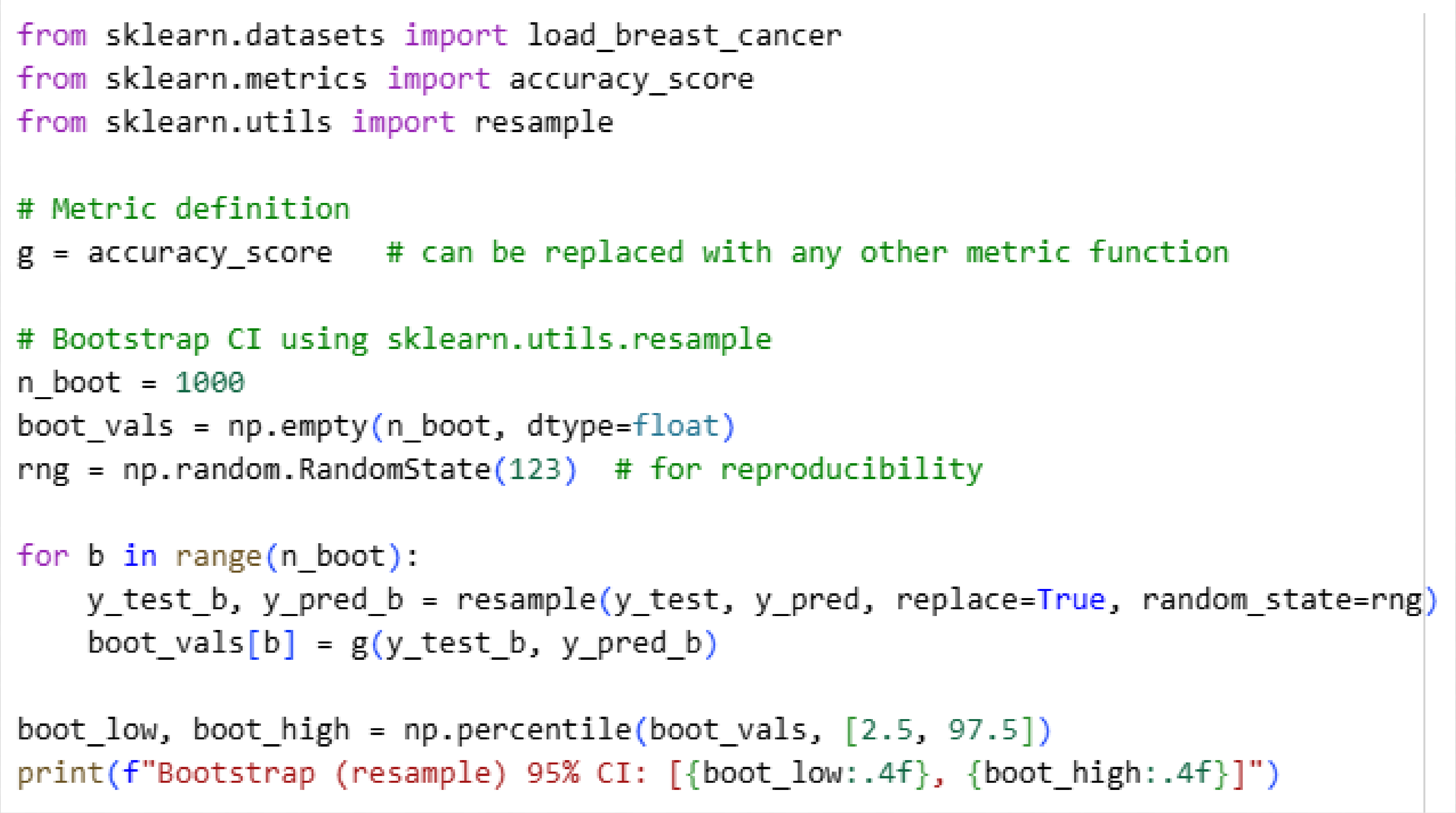
       $\hat{\theta}_b = g(D_b)$

3. **Bootstrap distribution**
   - Collect all $\hat{\theta}_1, \dots, \hat{\theta}_B$
   - This approximates the sampling distribution of your metric

4. **Percentile confidence interval**

$$\boxed{\mathrm{CI}_{95\%} = \left[\text{quantile}_{2.5\%}(\hat{\theta}_b),\ \text{quantile}_{97.5\%}(\hat{\theta}_b)\right]}$$

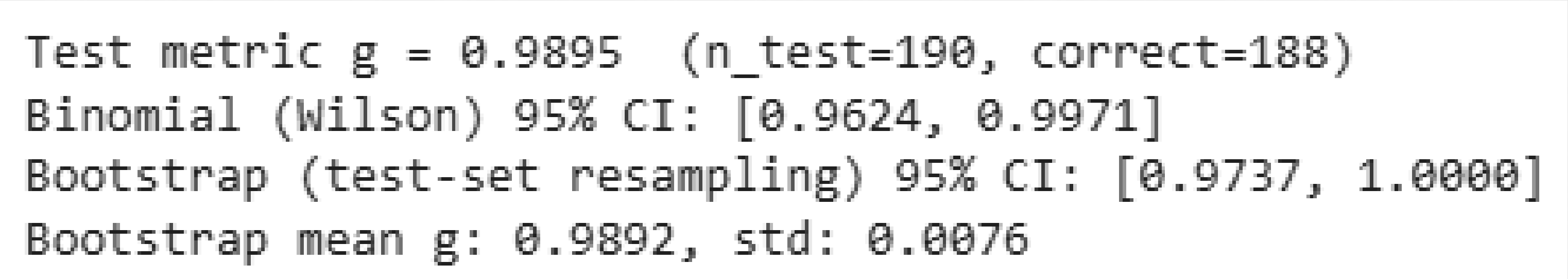
Estimating confidence intervals for any metric *g*: bootstrap method (test resample)



```python
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import accuracy_score
from sklearn.utils import resample

# Metric definition
g = accuracy_score   # can be replaced with any other metric function

# Bootstrap CI using sklearn.utils.resample
n_boot = 1000
boot_vals = np.empty(n_boot, dtype=float)
rng = np.random.RandomState(123)  # for reproducibility

for b in range(n_boot):
    y_test_b, y_pred_b = resample(y_test, y_pred, replace=True, random_state=rng)
    boot_vals[b] = g(y_test_b, y_pred_b)

boot_low, boot_high = np.percentile(boot_vals, [2.5, 97.5])
print(f"Bootstrap (resample) 95% CI: [{boot_low:.4f}, {boot_high:.4f}]")
```

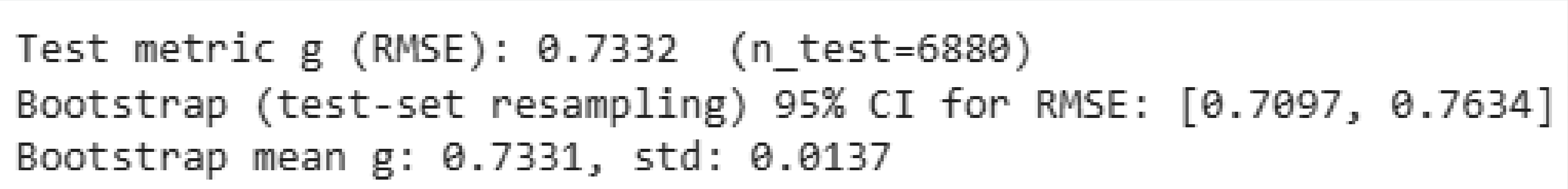
Estimating confidence intervals for any metric *g*: bootstrap method (test resample)

- For the breast-cancer dataset (a two-class classification problem with metric = accuracy)



```
Test metric g = 0.9895  (n_test=190, correct=188)
Binomial (Wilson) 95% CI: [0.9624, 0.9971]
Bootstrap (test-set resampling) 95% CI: [0.9737, 1.0000]
Bootstrap mean g: 0.9892, std: 0.0076
```

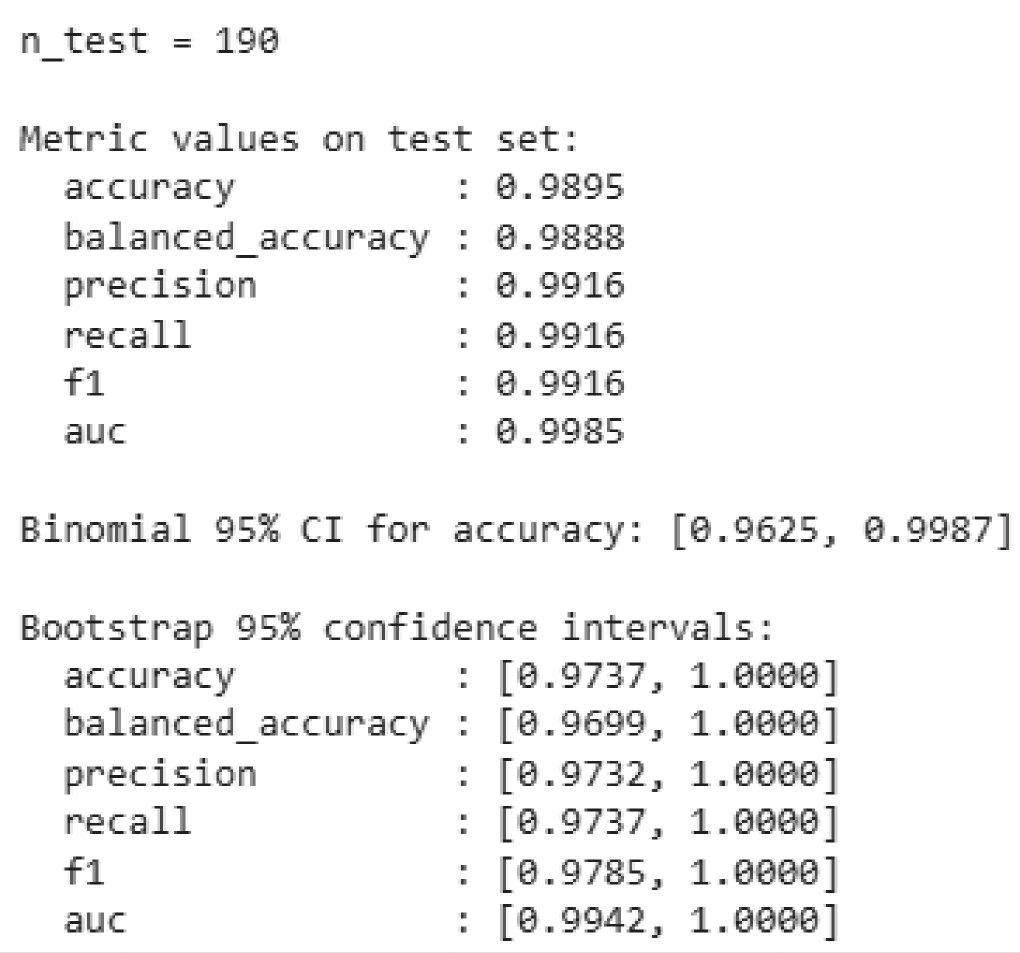
- For the california-housing dataset (a regression problem with metric = RMSE)



```
Test metric g (RMSE): 0.7332  (n_test=6880)
Bootstrap (test-set resampling) 95% CI for RMSE: [0.7097, 0.7634]
Bootstrap mean g: 0.7331, std: 0.0137
```

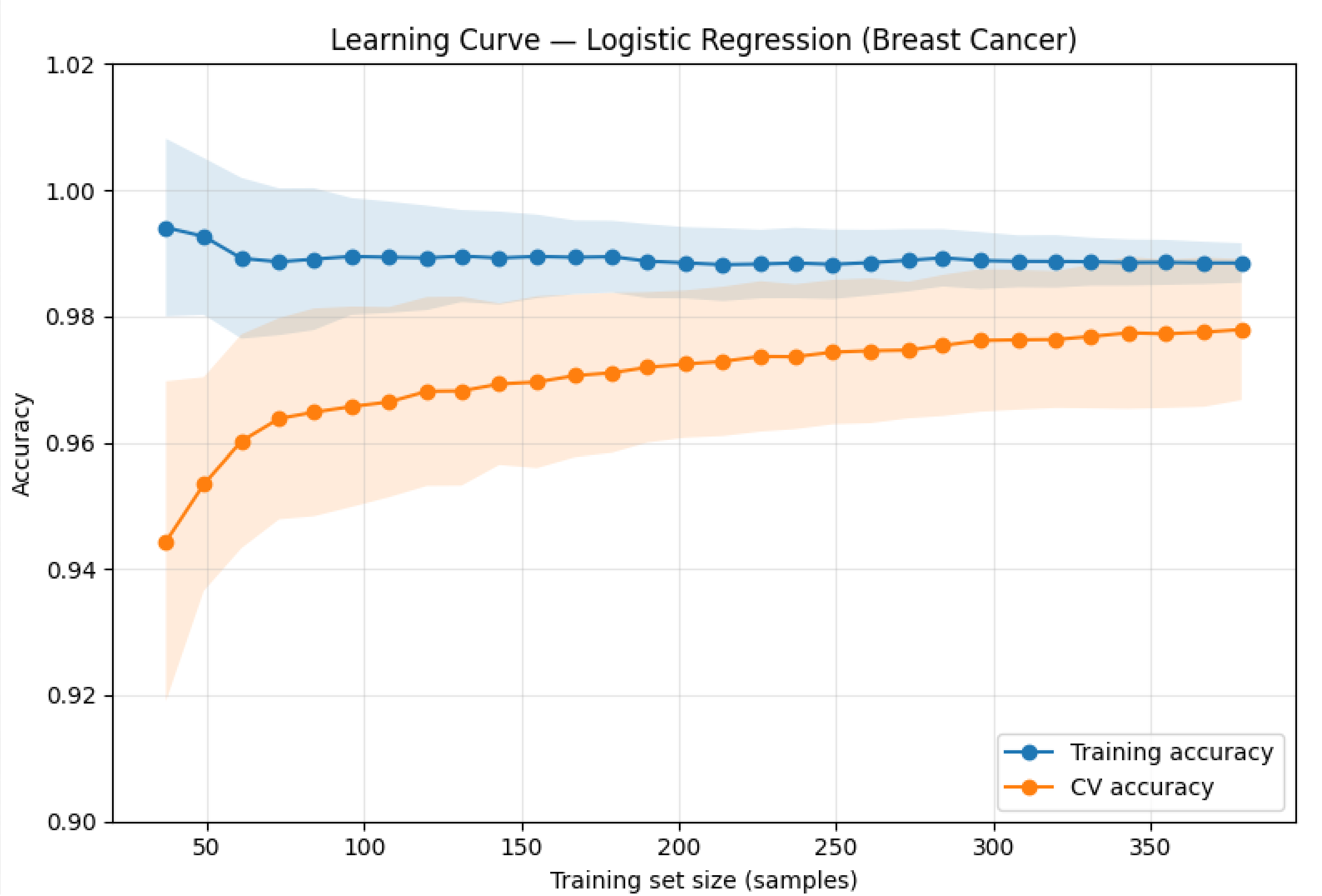
Estimating confidence intervals for any metric *g*: bootstrap method (test resample)

- For the breast-cancer dataset with several metrics



```
n_test = 190

Metric values on test set:
  accuracy          : 0.9895
  balanced_accuracy : 0.9888
  precision         : 0.9916
  recall            : 0.9916
  f1                : 0.9916
  auc               : 0.9985

Binomial 95% CI for accuracy: [0.9625, 0.9987]

Bootstrap 95% confidence intervals:
  accuracy          : [0.9737, 1.0000]
  balanced_accuracy : [0.9699, 1.0000]
  precision         : [0.9732, 1.0000]
  recall            : [0.9737, 1.0000]
  f1                : [0.9785, 1.0000]
  auc               : [0.9942, 1.0000]
```

On the size of the training partition: Learning Curves



SYLLABUS

1. Introduction to Machine Learning: tasks, algorithms & models

2. Basic methods for training classification and regression models:

3. Methodology (the Machine Learning workflow): **model evaluation**, hyper-parameter tuning, preprocessing, …

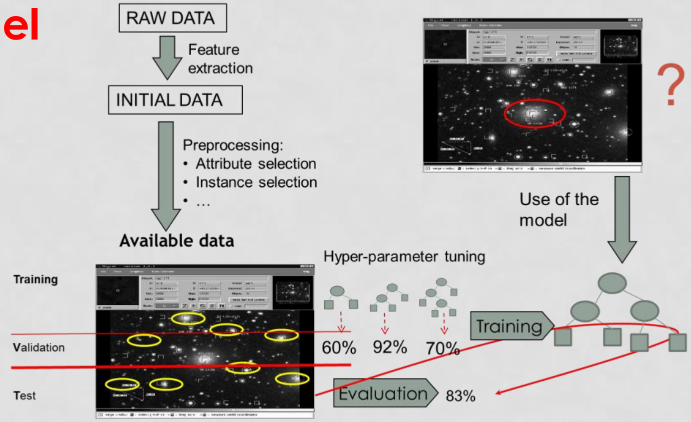
4. Methods for preprocessing

5. Advanced training methods based on ensembles of models

6. Large Scale Machine Learning. Big Data

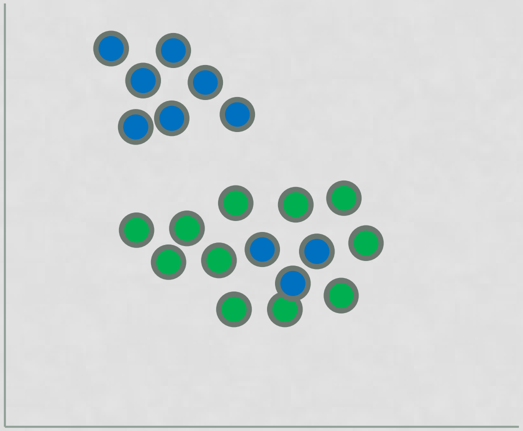
7. Advanced topics

8. Software tools

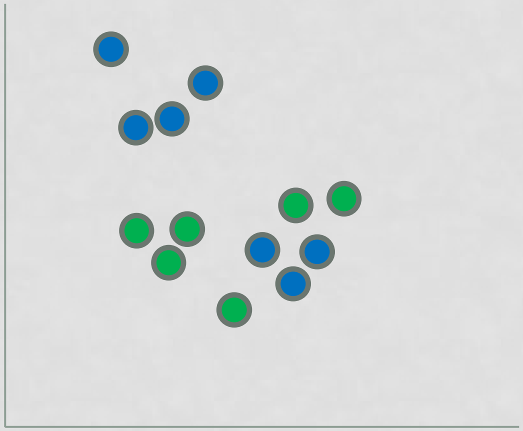


train/ test (holdout) drawback

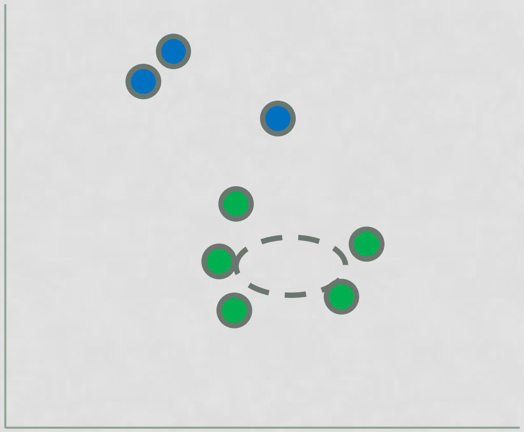
- It is possible that the test partition does not represent well the problem (by chance), mainly **if dataset is small**.



Available data



Train partition



Biased test partition

CROSSVALIDATION

- A possible solution is to repeat the train and test procedure several times (splitting into train and test in different ways) and then compute the average.
- Given that biases in train and/or test are random, computing the average may cancel them.
- Crossvalidation is such a solution, with the advantage that the different test partitions are independent (they do not overlap)

CROSSVALIDATION

- The available data is divided into k folds (k partitions).
- With k=3, three folds X, Y, and Z.
- The process has k steps (3 in this case):
  - Train model with X, Y, and test it with Z (T1 = success rate on Z)
  - Train model with X, Z, and test it with Y (T2 = success rate on Y)
  - Train model with Y, Z and test it with X (T3 = success rate on X)
  - Accuracy TX = (T1+T2+T3)/3
- k=10 is recommended. K between 5 and 10 can also be used.

3-fold cross-validation evaluation

Train with X and Y, evaluate with Z

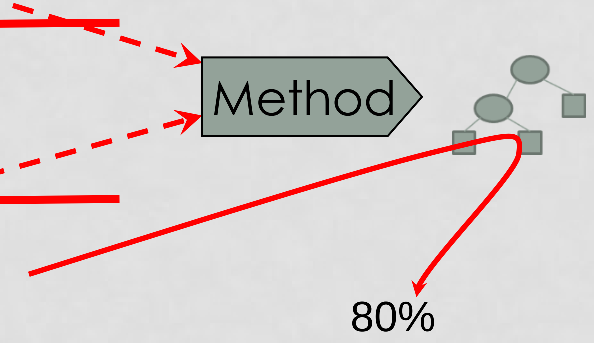
Available data

Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No

Fold X

Fold Y

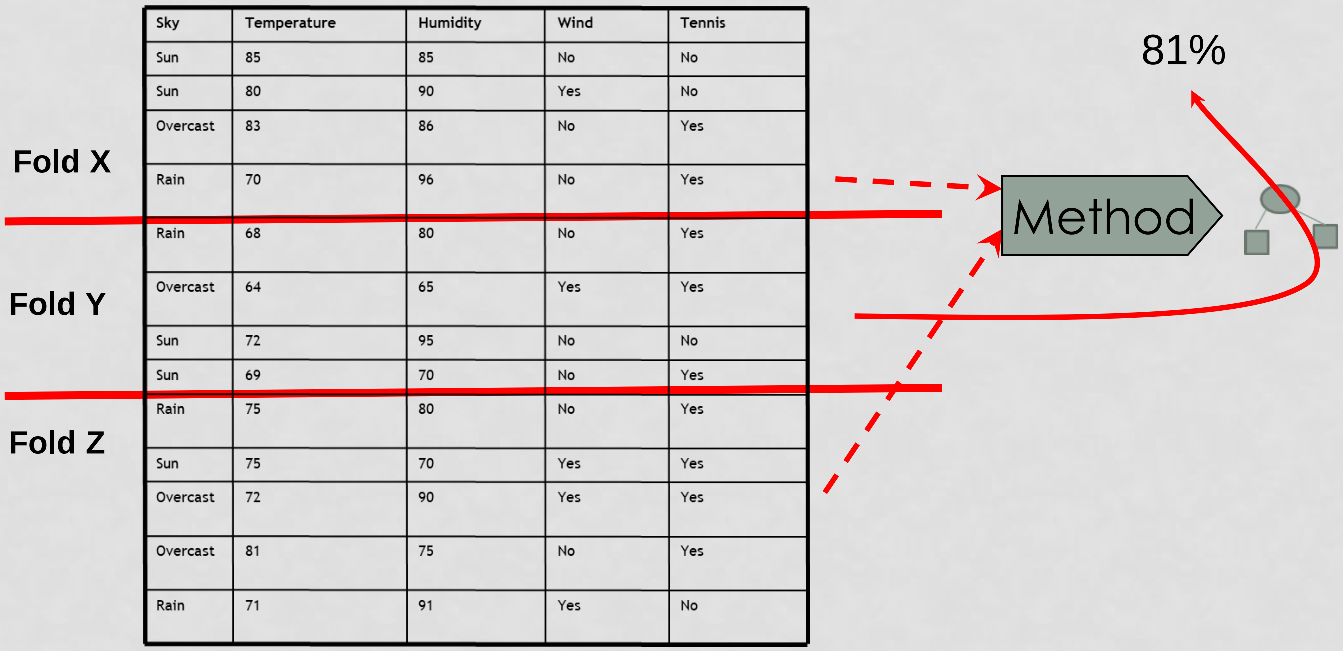
Fold Z



3-fold cross-validation evaluation

Train with X, Z; evaluate with Y

Available data



Fold X

Fold Y

Fold Z

Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No

Method

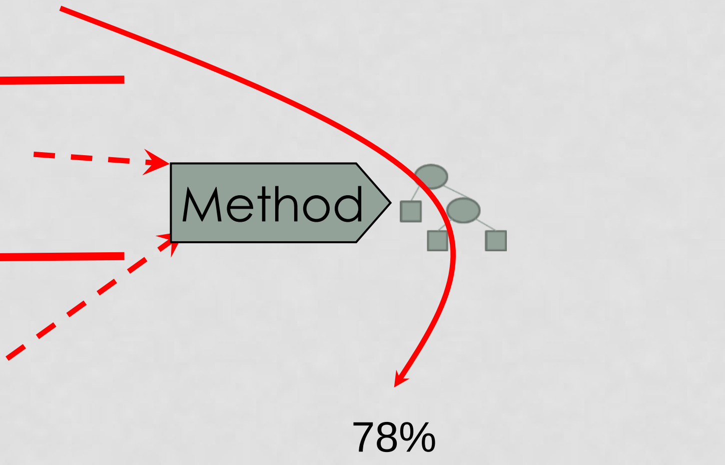
81%

3-fold cross-validation evaluation

Train with Y, Z; evaluate with X

Available data

	Sky	Temperature	Humidity	Wind	Tennis
Fold X	Sun	85	85	No	No
	Sun	80	90	Yes	No
	Overcast	83	86	No	Yes
	Rain	70	96	No	Yes
	Rain	68	80	No	Yes
Fold Y	Overcast	64	65	Yes	Yes
	Sun	72	95	No	No
	Sun	69	70	No	Yes
	Rain	75	80	No	Yes
Fold Z	Sun	75	70	Yes	Yes
	Overcast	72	90	Yes	Yes
	Overcast	81	75	No	Yes
	Rain	71	91	Yes	No



3-fold cross-validation evaluation

The estimation of future performance T is the average of the three folds.

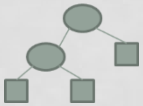
Available data

Fold X

Fold Y

Fold Z

Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No



80%

81%

78%

Evaluation

T=
(80%+81%+78%)/3
= 79.7%

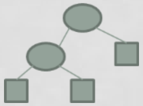
3-fold cross-validation evaluation

Crossvalidation is repeated train/test, with the advantage that the different test partitions are independent (they do not overlap)

Available data

The estimation of future performance T is the average of the three folds.

	Sky	Temperature	Humidity	Wind	Tennis
Fold X	Sun	85	85	No	No
	Sun	80	90	Yes	No
	Overcast	83	86	No	Yes
	Rain	70	96	No	Yes
	Rain	68	80	No	Yes
Fold Y	Overcast	64	65	Yes	Yes
	Sun	72	95	No	No
	Sun	69	70	No	Yes
	Rain	75	80	No	Yes
Fold Z	Sun	75	70	Yes	Yes
	Overcast	72	90	Yes	Yes
	Overcast	81	75	No	Yes
	Rain	71	91	Yes	No



80%

81%

78%

Evaluation

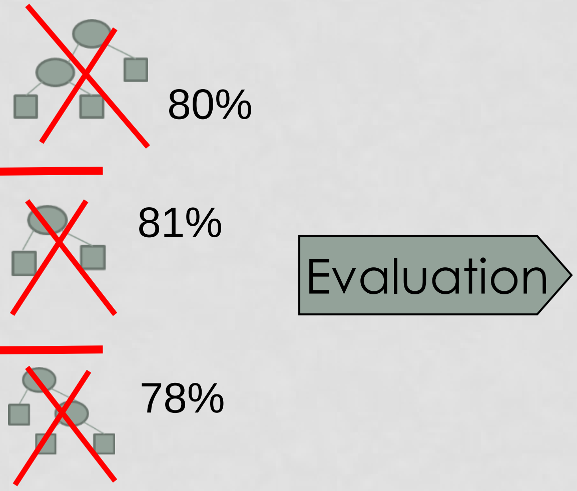
T=
(80%+81%+78%)/3
= 79.7%

3-fold cross-validation evaluation

Once T has been computed, the three models used to compute it are discarded and …

Available data

	Sky	Temperature	Humidity	Wind	Tennis
Fold X	Sun	85	85	No	No
	Sun	80	90	Yes	No
	Overcast	83	86	No	Yes
	Rain	70	96	No	Yes
	Rain	68	80	No	Yes
Fold Y	Overcast	64	65	Yes	Yes
	Sun	72	95	No	No
	Sun	69	70	No	Yes
	Rain	75	80	No	Yes
Fold Z	Sun	75	70	Yes	Yes
	Overcast	72	90	Yes	Yes
	Overcast	81	75	No	Yes
	Rain	71	91	Yes	No



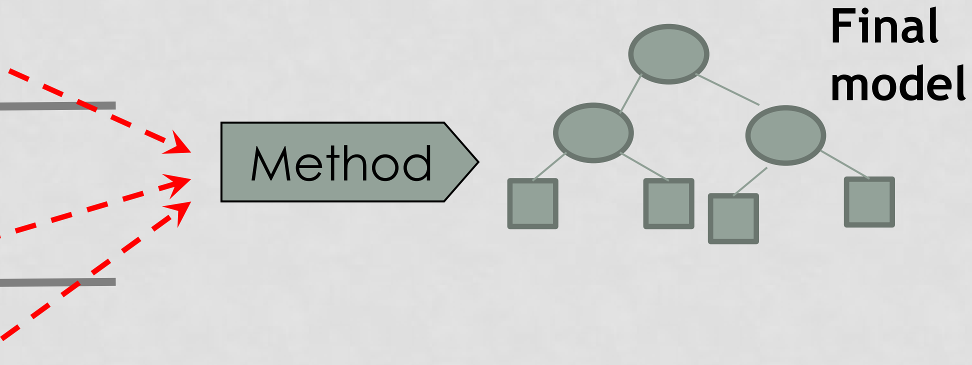
T= (80%+81%+78%)/3 = 79.7%

3-fold cross-validation evaluation

- <u>A final model is trained with the complete dataset</u>

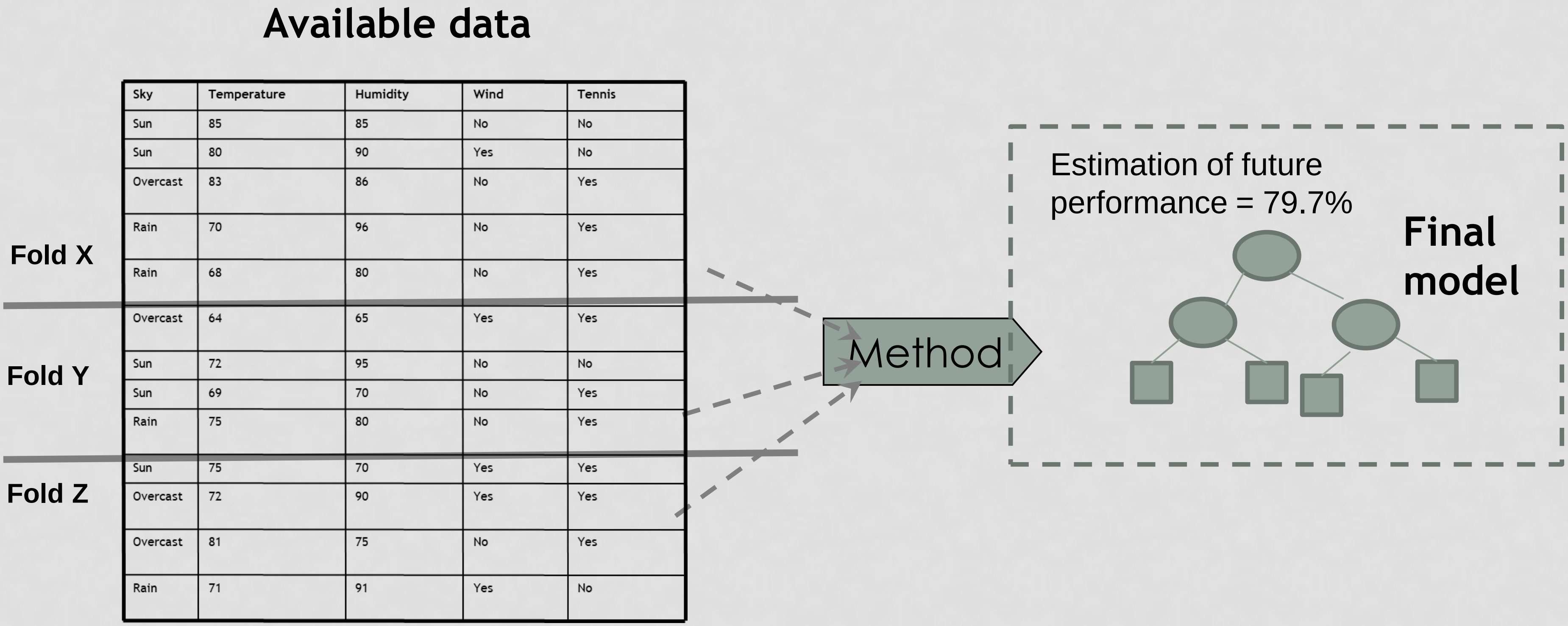
Available data

	Sky	Temperature	Humidity	Wind	Tennis
Fold X	Sun	85	85	No	No
	Sun	80	90	Yes	No
	Overcast	83	86	No	Yes
	Rain	70	96	No	Yes
	Rain	68	80	No	Yes
Fold Y	Overcast	64	65	Yes	Yes
	Sun	72	95	No	No
	Sun	69	70	No	Yes
	Rain	75	80	No	Yes
Fold Z	Sun	75	70	Yes	Yes
	Overcast	72	90	Yes	Yes
	Overcast	81	75	No	Yes
	Rain	71	91	Yes	No



3-fold cross-validation evaluation

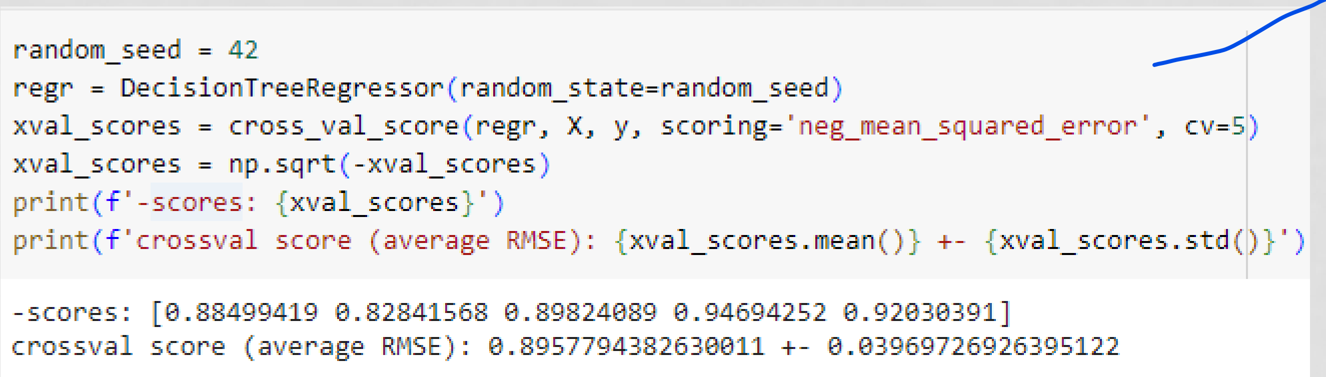
- A final model is trained with the entire dataset
- The estimation of future performance computed previously is kept (79.7%)
- Again, this is considered a **pesimistic estimation**, because the data partitions used to compute it were smaller (2/3) than the dataset used to train the final model.



Sky	Temperature	Humidity	Wind	Tennis
Sun	85	85	No	No
Sun	80	90	Yes	No
Overcast	83	86	No	Yes
Rain	70	96	No	Yes
Rain	68	80	No	Yes
Overcast	64	65	Yes	Yes
Sun	72	95	No	No
Sun	69	70	No	Yes
Rain	75	80	No	Yes
Sun	75	70	Yes	Yes
Overcast	72	90	Yes	Yes
Overcast	81	75	No	Yes
Rain	71	91	Yes	No

CROSSVALIDATION IN SCIKIT-LEARN

Dont use this method



```
random_seed = 42
regr = DecisionTreeRegressor(random_state=random_seed)
xval_scores = cross_val_score(regr, X, y, scoring='neg_mean_squared_error', cv=5)
xval_scores = np.sqrt(-xval_scores)
print(f'-scores: {xval_scores}')
print(f'crossval score (average RMSE): {xval_scores.mean()} +- {xval_scores.std()}')
```

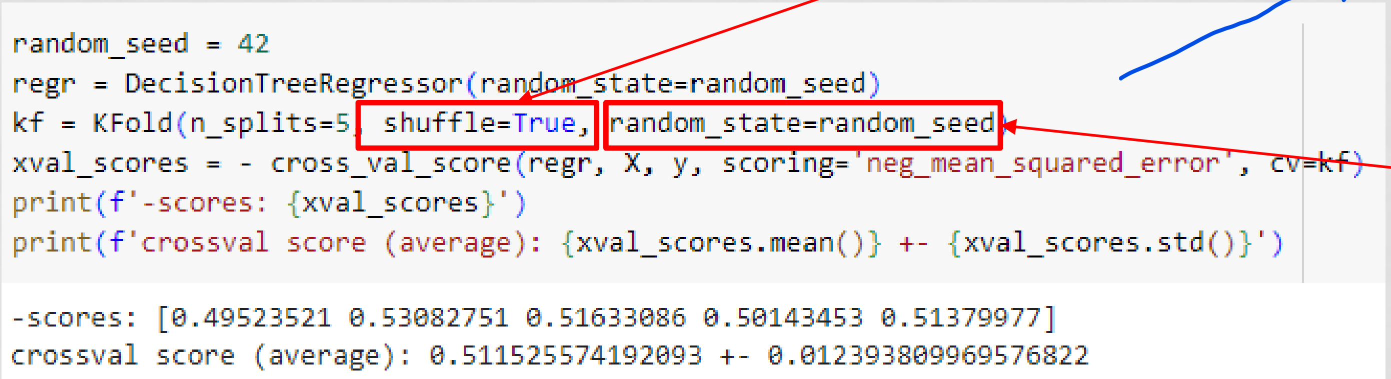
```
-scores: [0.88499419 0.82841568 0.89824089 0.94694252 0.92030391]
crossval score (average RMSE): 0.8957794382630011 +- 0.03969726926395122
```

Note: for sklearn, a **score** means "higher is better", hence the "neg"

IMPORTANT

- *cross_val_score* does not randomly shuffle the dataset before splitting into folds. If shuffling is required:

Use this method for cv



```
random_seed = 42
regr = DecisionTreeRegressor(random_state=random_seed)
kf = KFold(n_splits=5, shuffle=True, random_state=random_seed)
xval_scores = - cross_val_score(regr, X, y, scoring='neg_mean_squared_error', cv=kf)
print(f'-scores: {xval_scores}')
print(f'crossval score (average): {xval_scores.mean()} +- {xval_scores.std()}')
```

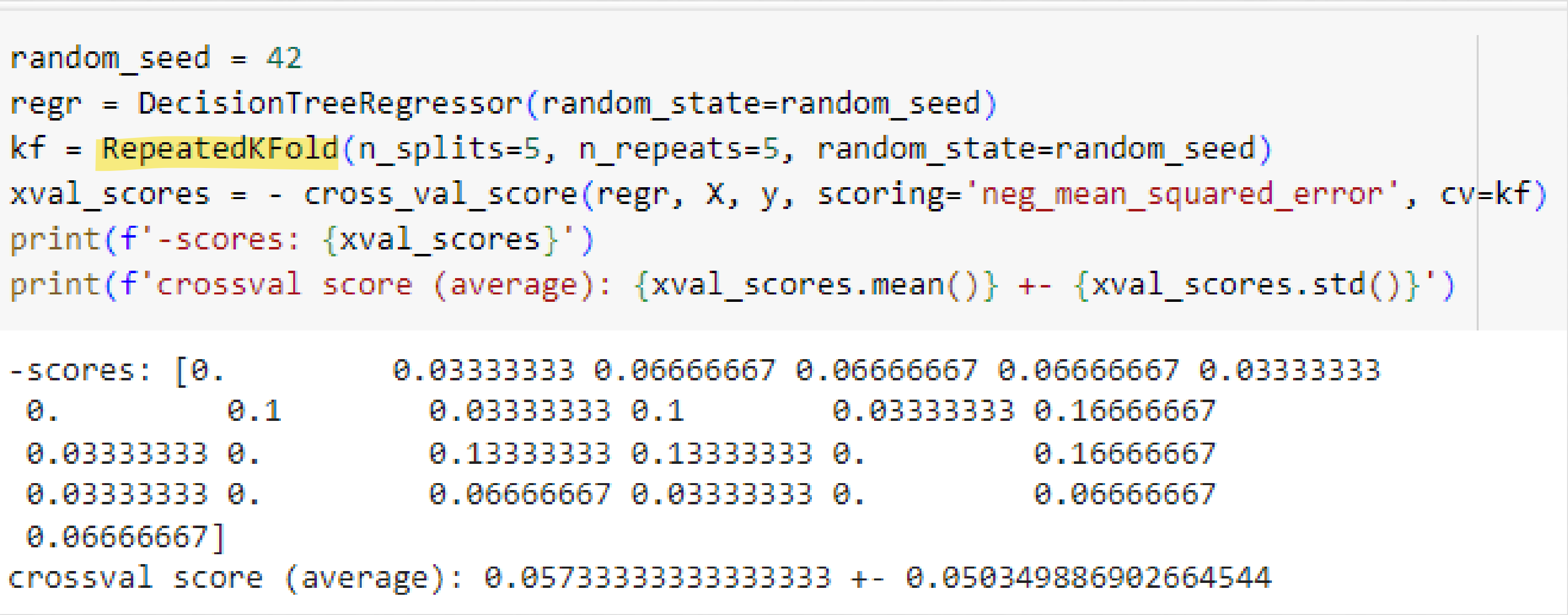
```
-scores: [0.49523521 0.53082751 0.51633086 0.50143453 0.51379977]
crossval score (average): 0.511525574192093 +- 0.012393809969576822
```

For **reproducibility** of results

CROSSVALIDATION IN SCIKIT-LEARN

- Different random states will result in different data shuffling which in turn will return different crossvalidation scores
- If **stability** of crossvalidation results is required, do repeated crossvalidation

You get 25 values and the T is the average

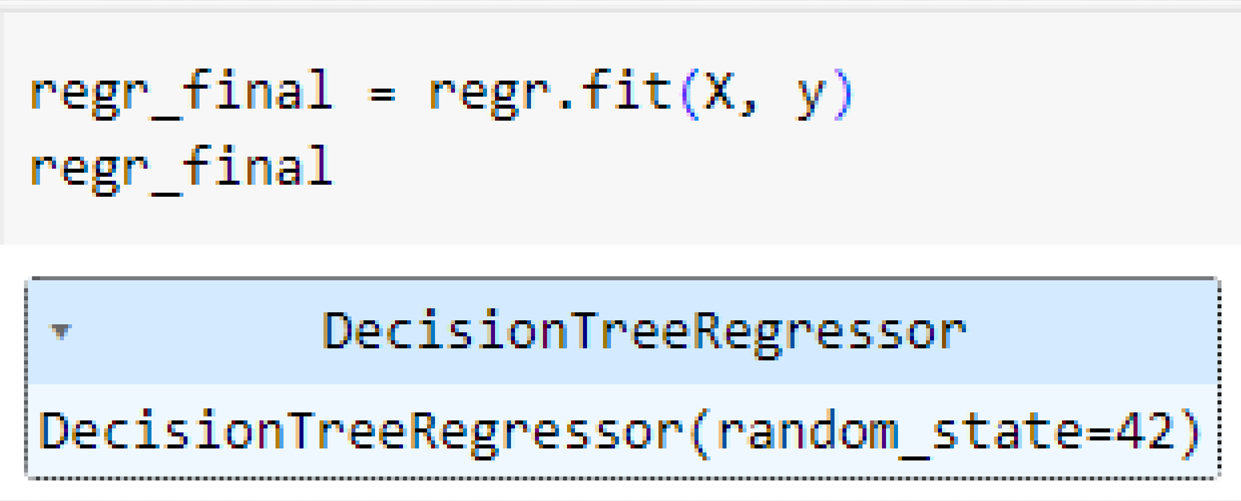


```python
random_seed = 42
regr = DecisionTreeRegressor(random_state=random_seed)
kf = RepeatedKFold(n_splits=5, n_repeats=5, random_state=random_seed)
xval_scores = - cross_val_score(regr, X, y, scoring='neg_mean_squared_error', cv=kf)
print(f'-scores: {xval_scores}')
print(f'crossval score (average): {xval_scores.mean()} +- {xval_scores.std()}')
```

```
-scores: [0.         0.03333333 0.06666667 0.06666667 0.06666667 0.03333333
 0.         0.1        0.03333333 0.1        0.03333333 0.16666667
 0.03333333 0.         0.13333333 0.13333333 0.         0.16666667
 0.03333333 0.         0.06666667 0.03333333 0.         0.06666667
 0.06666667]
crossval score (average): 0.05733333333333333 +- 0.05034988690266454
```

TRAIN-TEST IN SCIKITLEARN

Training the **final model** with the complete dataset: fitting regr again with (X, y)



```
regr_final = regr.fit(X, y)
regr_final
```

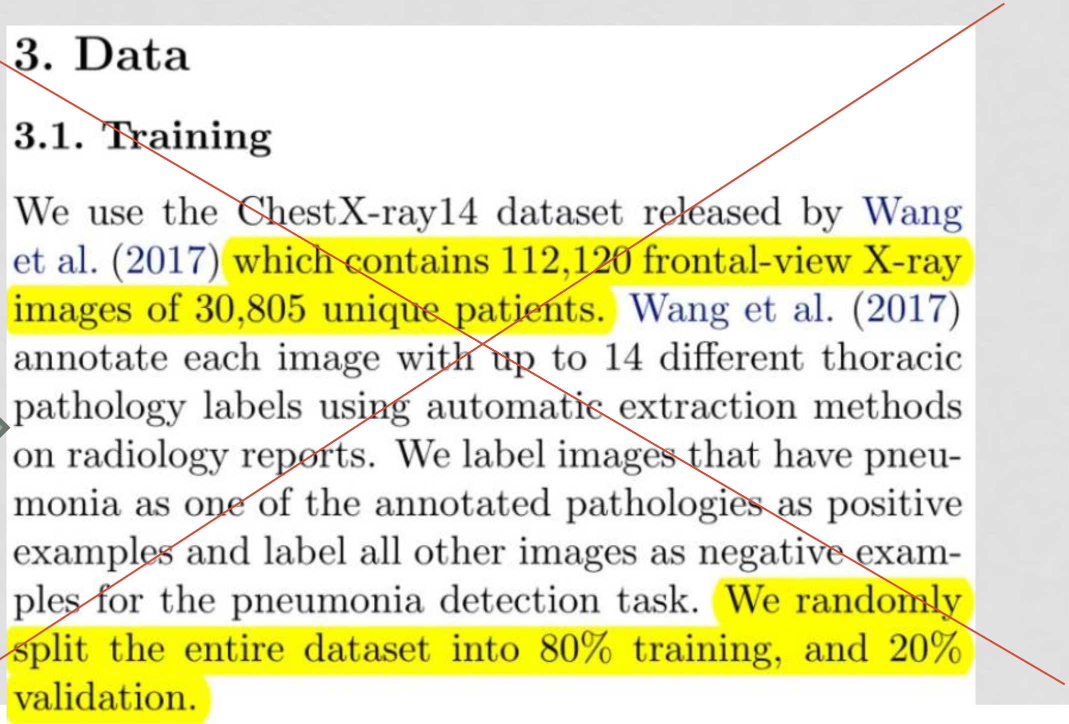
```
▾ DecisionTreeRegressor
DecisionTreeRegressor(random_state=42)
```

OTHER FORMS OF CROSSVALIDATION

- Standard k-fold assumes instances are i.i.d. (that is why data can be randomly shuffled)
- Group k-fold crossvalidation:
  - But sometimes, they may come in groups (e.g. records about hospital patients)
  - Same patient instances should go all together (i.e. grouped) to either train or test

Data is correlated because 1 user can have more than 1 picture

Wrong!



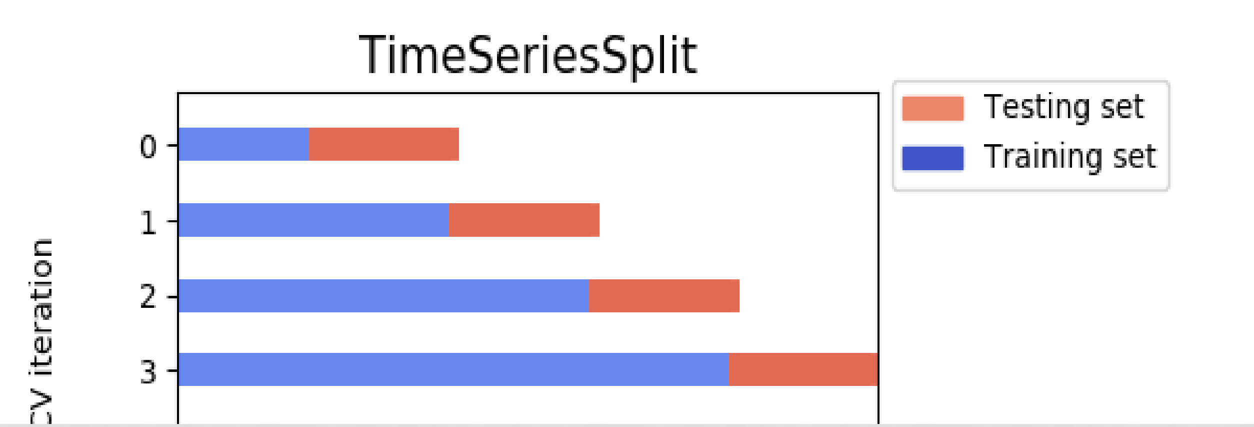
3. Data

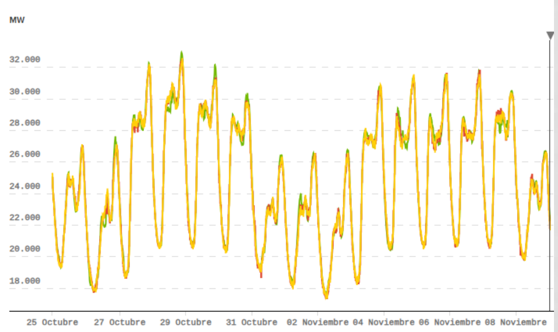
3.1. Training

We use the ChestX-ray14 dataset released by Wang et al. (2017) which contains 112,120 frontal-view X-ray images of 30,805 unique patients. Wang et al. (2017) annotate each image with up to 14 different thoracic pathology labels using automatic extraction methods on radiology reports. We label images that have pneumonia as one of the annotated pathologies as positive examples and label all other images as negative examples for the pneumonia detection task. We randomly split the entire dataset into 80% training, and 20% validation.

OTHER FORMS OF CROSSVALIDATION

- Standard k-fold assumes instances are i.i.d.
- Group k-fold crossvalidation:
  - But sometimes, they may come in groups (e.g. instances (records) about hospital patients)
  - Same patient instances should go all to either train or test (but not to both)
- Time series crossvalidation:
  - Typically, the past is chosen for training and the future for testing





$$y_t = f(y_{t-1}, y_{t-2}, \ldots, y_{t-p})$$

BASIC CRITERIA FOR EVALUATING CLASSIFICATION MODELS

- The standard performance measure for classification is

$$Accuracy = \frac{1}{n}\sum_{k=1}^{n}(y_k == \hat{y}_k)$$

Ground truth: $\{y_1, \ldots, y_n\}$
Model predictions: $\{\hat{y}_1, \ldots, \hat{y}_n\}$

  - Or equivalently the missclassification error = 1-accuracy
- In order to know whether a model is not useless, compare it with a trivial / naive / dummy model.
- E.g. with 2 classes, our model must be better than chance (tossing a head/tails coin): accuracy > 0.5
- For m classes, accuracy > 1/m
- However …

BASIC CRITERIA FOR EVALUATING CLASSIFICATION MODELS

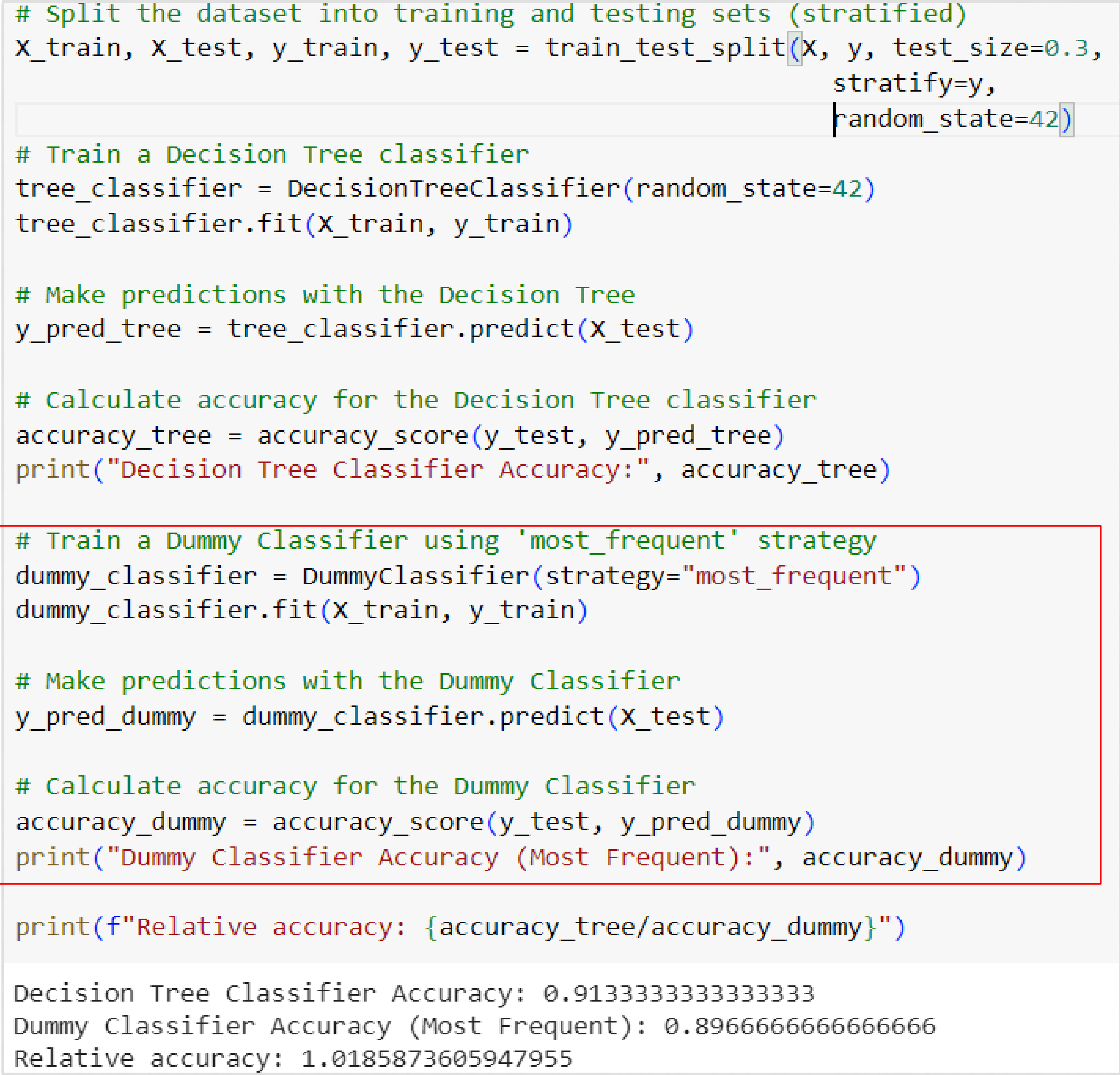
- Imbalanced datasets contain much more data for one of the classes (the majority class) than the other. E.g. most people do not have cancer.
- Example of imbalanced dataset:
  - 990 negative instances (99%)
  - 10 positive instances (1%)
- What is the minimum accuracy that our model should have in order to be considered "not useless"? 99%

The majority proportion class

BASIC CRITERIA FOR EVALUATING CLASSIFICATION MODELS

- Let be a problem with an imbalanced dataset:
  - 990 negative instances (99%)
  - 10 positive instances (1%)
- What is the minimum success rate that our model should have in order to be considered useful?

- A **trivial/naive (dummy) classifier** that always predicts "Negative" would already obtain 99% accuracy!!
- In imbalanced classification problems, a successful model has to obtain an accuracy larger than that of the majority class (most frequent) classifier.
- Solution: in order to know whether your model is "not useless", compare it with a dummy model, like the majority class (most-frequent) classifier.



```python
# Split the dataset into training and testing sets (stratified)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                     stratify=y,
                                                     random_state=42)
# Train a Decision Tree classifier
tree_classifier = DecisionTreeClassifier(random_state=42)
tree_classifier.fit(X_train, y_train)

# Make predictions with the Decision Tree
y_pred_tree = tree_classifier.predict(X_test)

# Calculate accuracy for the Decision Tree classifier
accuracy_tree = accuracy_score(y_test, y_pred_tree)
print("Decision Tree Classifier Accuracy:", accuracy_tree)

# Train a Dummy Classifier using 'most_frequent' strategy
dummy_classifier = DummyClassifier(strategy="most_frequent")
dummy_classifier.fit(X_train, y_train)

# Make predictions with the Dummy Classifier
y_pred_dummy = dummy_classifier.predict(X_test)

# Calculate accuracy for the Dummy Classifier
accuracy_dummy = accuracy_score(y_test, y_pred_dummy)
print("Dummy Classifier Accuracy (Most Frequent):", accuracy_dummy)

print(f"Relative accuracy: {accuracy_tree/accuracy_dummy}")
```

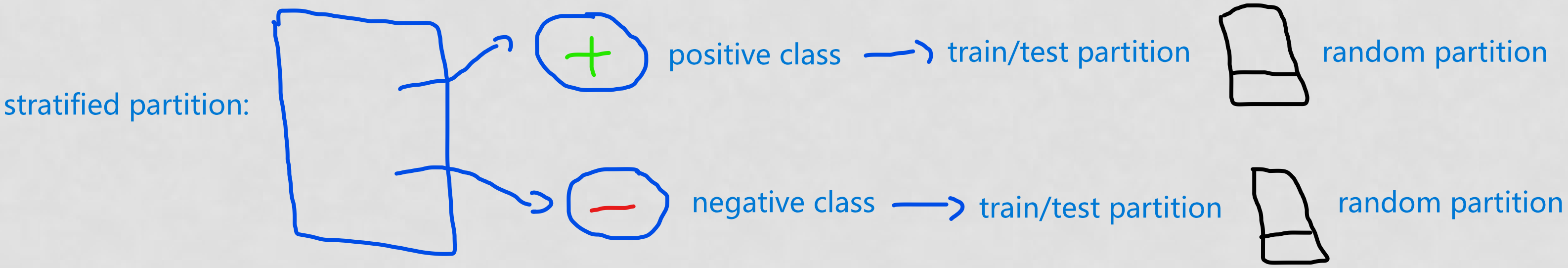
```
Decision Tree Classifier Accuracy: 0.9133333333333333
Dummy Classifier Accuracy (Most Frequent): 0.8966666666666666
Relative accuracy: 1.0185873605947955
```

in order to know whether your model is useful enough, compare it with a dummy model

EVALUATING IMBALANCED CLASSIFICATION PROBLEMS

- Also, **Stratified** partitions must be used: we have to make sure that data partitions are representative (the class distribution in the partitions should be the same as in the original data)
  - Example: if in the available data the distribution is 99%(-) / 1%(+), train and test should have the same distribution.

  - Stratified partitions keep the same positive / negative class distribution in train and test sets.
  - This kind of partition is difficult to achieve in imbalanced datasets by just splitting the data randomly. It has to be enforced:

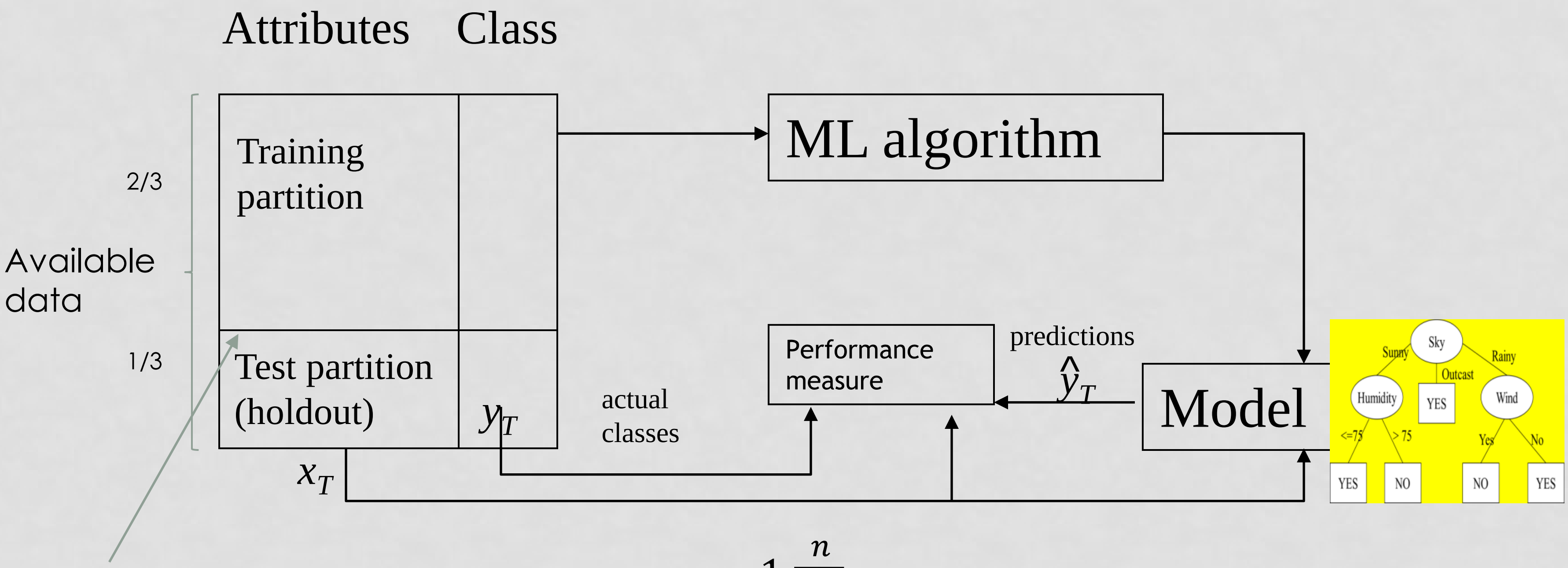


stratified partition:

+ positive class → train/test partition → random partition

− negative class → train/test partition → random partition

STANDARD HOLDOUT (TRAIN/TEST) FOR MODEL EVALUATION



Attributes Class

Available data

2/3

1/3

Training partition

Test partition (holdout)

y_T

x_T

ML algorithm

Model

Performance measure

predictions

$\hat{y}_T$

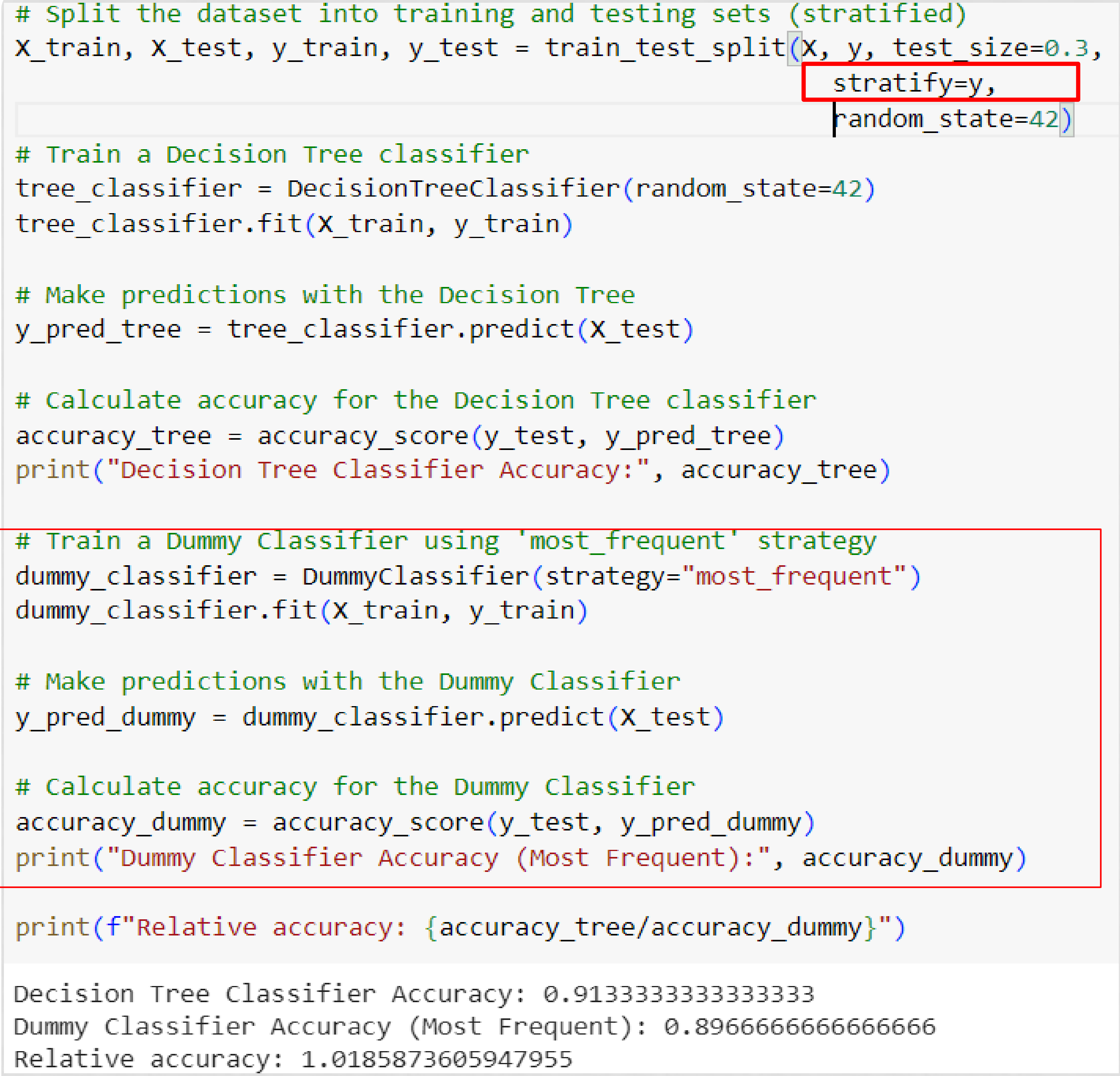
actual classes

if we split randomly, it is likely that the test partition will not be representative of the original dataset.

$$Accuracy = \frac{1}{n}\sum_{k=1}^{n} y_k == \hat{y}_k$$

EVALUATING IMBALANCED CLASSIFICATION PROBLEMS

- Stratification is available for both for train/test and crossvalidation:
  - X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42, **stratify=y**)
  - cv = **StratifiedKFold**(n_splits=5, random_state=42, shuffle=True)
    - sklearn.model_selection.StratifiedKFold



```
# Split the dataset into training and testing sets (stratified)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                      stratify=y,
                                                      random_state=42)
# Train a Decision Tree classifier
tree_classifier = DecisionTreeClassifier(random_state=42)
tree_classifier.fit(X_train, y_train)

# Make predictions with the Decision Tree
y_pred_tree = tree_classifier.predict(X_test)

# Calculate accuracy for the Decision Tree classifier
accuracy_tree = accuracy_score(y_test, y_pred_tree)
print("Decision Tree Classifier Accuracy:", accuracy_tree)

# Train a Dummy Classifier using 'most_frequent' strategy
dummy_classifier = DummyClassifier(strategy="most_frequent")
dummy_classifier.fit(X_train, y_train)

# Make predictions with the Dummy Classifier
y_pred_dummy = dummy_classifier.predict(X_test)

# Calculate accuracy for the Dummy Classifier
accuracy_dummy = accuracy_score(y_test, y_pred_dummy)
print("Dummy Classifier Accuracy (Most Frequent):", accuracy_dummy)

print(f"Relative accuracy: {accuracy_tree/accuracy_dummy}")
```

```
Decision Tree Classifier Accuracy: 0.9133333333333333
Dummy Classifier Accuracy (Most Frequent): 0.8966666666666666
Relative accuracy: 1.0185873605947955
```

Using stratified partitions **and** comparing with the dummy model

OTHER METRICS FOR IMBALANCED CLASSIFICATION PROBLEMS

- Accuracy is the main metric for classification problems.
- But accuracy is less meaningful for imbalanced problems because even naive (dummy/trivial) models get high accuracy.
- There are other metrics, which might be more meaningful than accuracy for imbalanced datasets:
  - Confusion matrix: True Positive Rate (TPR), True Negative Rate (TNR), …
  - Balanced accuracy
  - Area under the ROC curve (AUROC)
  - F1
  - Kappa
  - …

PERFORMANCE MEASURES FOR REGRESSION

- Actual values of the response variable (ground truth): $\{y_1, \dots, y_n\}$
- Model predictions: $\{\hat{y}_1, \dots, \hat{y}_n\}$

MSE = Mean Squared Error
RMSE = Root-Mean Squared Error

$$MSE = \frac{(\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \cdots + (\hat{y}_n - y_n)^2}{n}$$

$$RMSE = \sqrt{MSE}$$

- RMSE is very widely used in regression problems.
- But instances with large errors have too much weight on the average (because we square residuals which are already large).

PERFORMANCE MEASURES FOR REGRESSION

MAE = Mean Absolute Error

$$\text{MAE} = \frac{|\hat{y}_1 - y_1| + |\hat{y}_2 - y_2| + \cdots + |\hat{y}_n - y_n|}{n}$$

- With MAE, instances with large errors (outliers) do not have so much weight on the average (compared to RMSE).
- There is also the Median Absolute Error, which is even more robust to outliers.

PERFORMANCE MEASURES FOR REGRESSION

MAE = Mean Absolute Error

$$\mathrm{MAE} = \frac{|\hat{y}_1 - y_1| + |\hat{y}_2 - y_2| + \cdots + |\hat{y}_n - y_n|}{n}$$

- With MAE, instances with large errors do not have so much weight on the average (compared to RMSE).
- But both RMSE and MAE are scale-dependent: their magnitude depend on the scale of the output variable (y_i)
  - E.g.: if the output variable unit is meters, the RMSE and MAE will be 1000 larger than if the unit is km. That does not mean that the model is 1000 times worse. Just the scale is different.
- Therefore, in order to know whether the error of my model is “too large”, compare it with the error of a trivial / naive / dummy model.

PERFORMANCE MEASURES FOR REGRESSION

- Dummy models for regression:
  - If the metric is MSE, the dummy method should be a model that always predicts the average of the output variable.
    - MSE_{model} should be smaller than $\mathrm{MSE}_{\bar{y}}$
  - If the metric is MAE, the dummy method should be a model that always predicts the median of the output variable
    - MAE_{model} should be smaller than $MAE_{median(y)}$

PERFORMANCE MEASURES FOR REGRESSION

NOOOOOOO

- An example of naive model for regression (similar to classification): predict with a constant, disregarding the input attributes.
- What is the constant *c* that minimizes MSE?
- $MSE_c = \frac{1}{n}\sum_{i=1}^{i=n}(c - y_i)^2$
- $0 = \frac{dMSE}{dc} = \frac{1}{n}\sum_{i=1}^{i=n} 2(c - y_i) \quad = 2\left(c - \frac{1}{n}\sum_{i=1}^{i=n} y_i\right) = 2(c - \bar{y}) = 0$
  - $c = \bar{y}$
- Therefore, in the case of MSE, the MSE of the naive model is:
- $\mathrm{MSE}_{\bar{y}} = \frac{1}{n}\sum_{i=1}^{i=n}(\bar{y} - y_i)^2$
- Which happens to be the variance of the response variable
- MSE_{model} should be smaller than $\mathrm{MSE}_{\bar{y}}$

PERFORMANCE MEASURES FOR REGRESSION

NOOOOOO

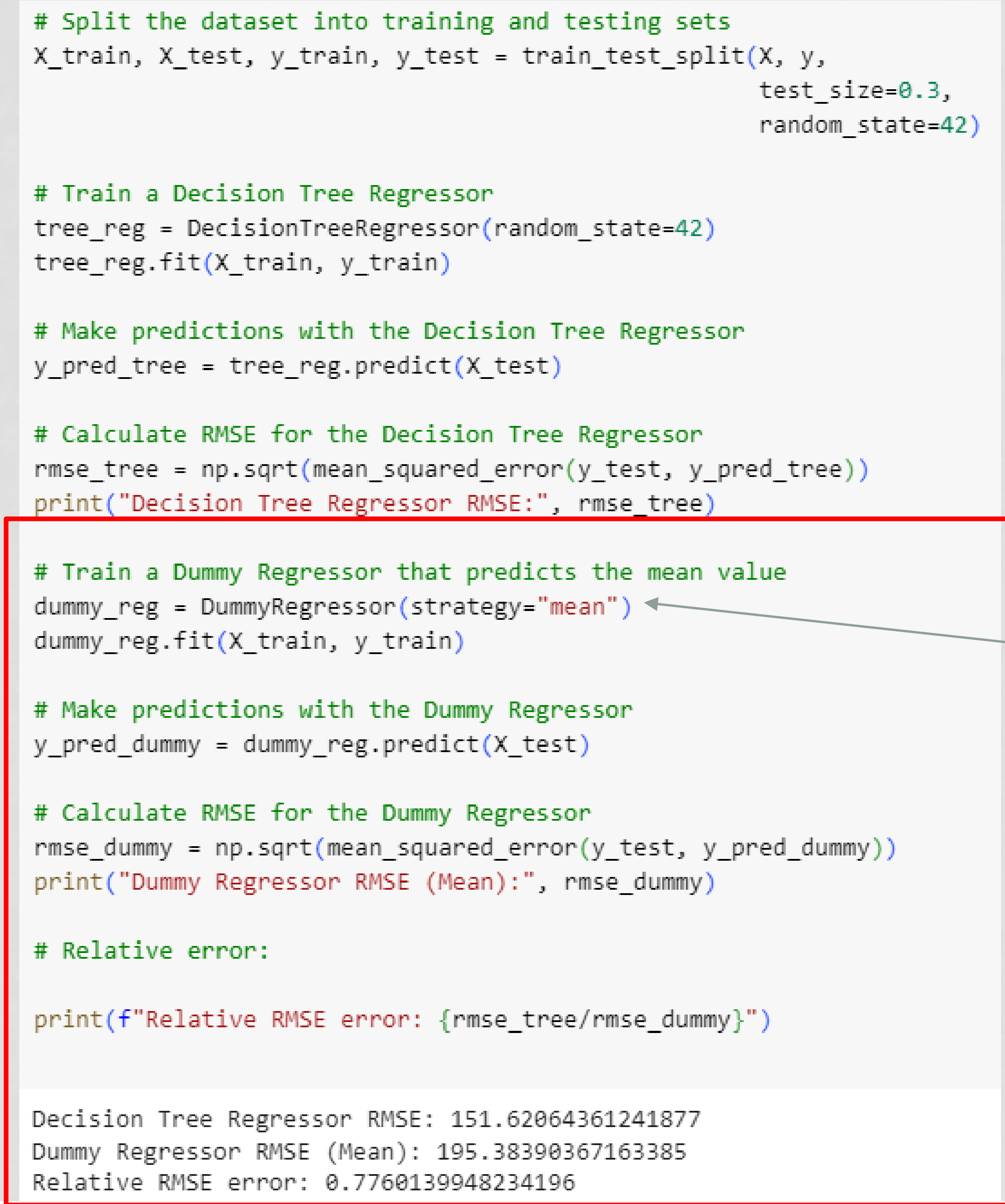
- An example of naive model for regression: predict with a constant.
- What is the constant *c* that minimizes MAE?
- $MAE_c = \frac{1}{n}\sum_{i=1}^{i=n}|c - y_i| \quad = \frac{1}{n}\sum_{i=1}^{i=a}(c - y_i) \quad + \frac{1}{n}\sum_{i=1}^{i=b}(y_i - c)$

  $a+b=n$
- $0 = \frac{dMAE}{dc} = \frac{1}{n}\sum_{i=1}^{i=a}(+1) \quad + \frac{1}{n}\sum_{i=1}^{i=b}(-1) \quad = 0$ when a=b
  - $c = median(y)$
- Therefore, in the case of MAE, the MAE of the naive model is:
- $MAE_{median(y)} = \frac{1}{n}\sum_{i=1}^{i=n}|median(y) - y_i|$
- MAE_{model} should be smaller than $MAE_{median(y)}$

TRIVIAL / NAIVE MODELS IN SCIKITLEARN: DUMMY ESTIMATORS

- In practice, in sklearn we can compute the error of our model and then compare it with the error of the appropriate dummy model

- DummyRegressor:
  - Mean (for MSE): always predicts the mean of the training targets.
  - Median (for MAE): always predicts the median of the training targets.



```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    random_state=42)


# Train a Decision Tree Regressor
tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(X_train, y_train)

# Make predictions with the Decision Tree Regressor
y_pred_tree = tree_reg.predict(X_test)

# Calculate RMSE for the Decision Tree Regressor
rmse_tree = np.sqrt(mean_squared_error(y_test, y_pred_tree))
print("Decision Tree Regressor RMSE:", rmse_tree)

# Train a Dummy Regressor that predicts the mean value
dummy_reg = DummyRegressor(strategy="mean")
dummy_reg.fit(X_train, y_train)

# Make predictions with the Dummy Regressor
y_pred_dummy = dummy_reg.predict(X_test)

# Calculate RMSE for the Dummy Regressor
rmse_dummy = np.sqrt(mean_squared_error(y_test, y_pred_dummy))
print("Dummy Regressor RMSE (Mean):", rmse_dummy)

# Relative error:

print(f"Relative RMSE error: {rmse_tree/rmse_dummy}")
```

```
Decision Tree Regressor RMSE: 151.62064361241877
Dummy Regressor RMSE (Mean): 195.38390367163385
Relative RMSE error: 0.7760139948234196
```

Could be “median” if MAE

RELATIVE MEASURES

- Relative Squared Error (RSE) and Relative Absolute Error (RAE)
  - They range from 0 (best value) to 1 (worst value), although it can be larger than 1 if the model is very bad.
- Skill scores: comparison between the error of a model and the error of a simpler/reference model (for example, a trivial/dummy model)

$$RSE = \frac{\frac{1}{n}\sum_{i=1}^{i=n}(\hat{y}_i - y_i)^2}{\frac{1}{n}\sum_{i=1}^{i=n}(\bar{y} - y_i)^2} = \frac{MSE_{model}}{MSE_{dummy(mean)}}$$

$$SS_{MSE} = 1 - \frac{MSE_{model}}{MSE_{dummy(mean)}}$$

0 is the worst and 1 is the best

$$RAE = \frac{\frac{1}{n}\sum_{i=1}^{i=n}|\hat{y}_i - y_i|}{\frac{1}{n}\sum_{i=1}^{i=n}|\bar{y} - y_i|} = \frac{MAE_{model}}{MAE_{dummy(median)}}$$

$$SS_{MAE} = 1 - \frac{MAE_{model}}{MAE_{dummy(median)}}$$