Evaluation of Learning

Paulo Trigo Silva

Evaluation – “the key to making real progress”

How to evaluate?

1. Build a model from a **training** dataset with a **high number** of instances.
2. Apply the model to a **testing** dataset with a **high number** of instances.
3. **Analyze the error** from **applying the model to the testing** dataset.

So, what difficulties are there?

A. Get a training and testing dataset with a **high number** of instances.

B. Do a **cost/benefit analysis** regardless of the algorithm being used.

Paulo Trigo Silva

The error rate and the (training) data to build the model

- The "error rate" is generally used to measure the performance
  - of a classifier
- The classifier predicts the class value of an instance
  - if prediction is correct it is considered a ***success***; if not it is an ***error***
- The “error rate” is the proportion of “errors” in a set of instances
  - it is a measure of the overall performance of a classifier
- ... but we are interested in the likely future performance on new data
  - we are not interested in the error of the training dataset
  - ... since the model was built from that same dataset!
- So, the question that arises is, “**is the error rate on old data likely to be a good indicator of the error rate on new data?**”

… *resubstitution error* and the performance of a classifier

The error rate on the training data is called the *resubstitution error*, because it is calculated by resubstituting the training instances into a classifier that was constructed from them.

Although it is not a reliable predictor of the true error rate on new data, it is nevertheless often useful to know.

But, in order to properly predict the performance of a classifier on new data, we need to assess its error rate on a **dataset that played no part in the formation of the classifier**.

This independent dataset is called the **testing dataset** (or just, test data).

We assume that both the training dataset and the testing dataset are representative samples of the underlying problem.

The need for both the training and the testing datasets

“is the error rate on old data likely to be a good indicator of the error rate on new data?”

If the “old data” was used during the learning process for training the classifier, then the answer is NO.

Why? Because if the classifier was learned from the very same training data, any estimate of performance based on that data will be optimistic, and may be hopelessly optimistic!

So, **to estimate the performance of the learning process, we need**:

- to assess its error rate on a dataset that <u>played no part in the formation of the classifier</u>; such independent set is called the “**testing dataset**”,

- assume that both the training dataset and the testing dataset are representative samples of the underlying problem.

Paulo Trigo Silva

... datasets for training, validating and testing

- The dataset for testing can not participate in the training
  - some methods have two stages – *build* model and *tune* parameters
  - ... e.g., build a tree and perform the pruning
  - the testing dataset can not participate in any of those two stages
- If the method has 2 stages then we must have 3 datasets
  - training dataset to "build the model", e.g., decision tree
  - validating dataset to "tune parameters", e.g., perform pruning
  - testing dataset to "estimate error rate of future data"
- … also, the dataset used for training (and validating)
  - can not be used to estimate the error rate concerning future data
- The same dataset can be used by different methods
- ... should use different datasets for training, validating and testing

The process – using the largest possible sample of data

- The goal of the evaluation is to choose the learning method
  - with the lowest error rate, i.e., the best suited to the problem
- After choosing the method we can use the testing dataset
  - once the error rate has been determined, the testing data is bundled back into the training data to produce a new classifier for actual use
  - ... this enables to maximizes the amount of data used to generate the classifier that will actually be employed in practice
- In short, for each method to evaluate, we follow the process
  - [1] use **training** dataset to build the model
  - [2\option] use validation dataset to optimize the model's parameters
  - [3] use **testing** dataset to estimate error of the final, optimized, method
- ... compare the error rate and choose one (or more) methods
  - [4] **bundle back testing and validation** data into a single training dataset
  - [5] build a final model that will **actually be employed in practice**.

The difficulty – often “there is not a vast supply of data!”

- If there is a vast amount of data available, everything is very simple,
  - consider 2 (or 3) very large samples of independent data
  - ... **training**, validating and **testing**
- The difficulty occurs when there is not a vast supply of data available
  - often, the data, training and testing, is classified manually
  - ... and this limits the amount of data for training, validating and testing
- ... so, how to make the most of a limited dataset?
  - a certain amount is held over for testing; is called the *holdout* procedure
  - remaining is used for training; if necessary, part is set aside for validation
- There is a dilemma here; to find a “good classifier”
  - we want to use **as much of the data as possible** for <u>training</u>
  - … to get a good error estimate use **as much data as possible** for <u>testing</u>

… “overcome difficulty” via dataset-split formal techniques

- … the Python `sklearn.model_selection` provides techniques for,
  - applying **train-test-split** (dataset-split) techniques, and for
  - computing **model-evaluation** metrics

- also available are `sklearn.naive_bayes`, `sklearn.tree`,
  - ... and other classifiers

```
from sklearn.model_selection import

  # train-test-split techniques
  ShuffleSplit, StratifiedShuffleSplit, \
  KFold, StratifiedKFold, \
  RepeatedKFold, RepeatedStratifiedKFold, \
  LeaveOneOut, LeavePOut, \

  # model-evaluation metrics
  cross_val_score
```

`split_and_eval.py`

Paulo Trigo Silva

… “general” train-test-split recipe – build list of techniques

- **build** a list of train-test-split (`tt_split`) techniques
- **apply** each technique to get indexes to split the dataset

```
seed = 5 #None #used by random generator (value is integer or None)

list_func_tt_split = \
  [
    (holdout,                        (1.0/3.0, seed)),
    (stratified_holdout,             (1.0/3.0, seed)),
    (repeated_holdout,               (1.0/3.0, 2, seed)),
    (repeated_stratified_holdout,    (1.0/3.0, 2, seed)),
    (fold_split,                     (3, seed)),
    (stratified_fold_split,          (3, seed)),
    (repeated_fold_split,            (3, 2, seed)),
    (repeated_stratified_fold_split, (3, 2, seed)),
    (leave_one_out,                  ()),
    (leave_p_out,                    (2, )),
    (bootstrap_split_once,           (seed, )),
    (bootstrap_split_repeated,       (2, seed))
  ])
```

split_and_eval.py

… “general” train-test-split recipe – split the dataset

- **build** a list of train-test-split (`tt_split`) techniques
- **apply** each technique to get indexes to split the dataset

```
def train_test_split_recipe( file_feature_name,
                             func_tt_split, *args_tt_split ):
  (fileName, list_featureName) = file_feature_name
  D = load_dataset( fileName, list_featureName )
  (X, y) = split_dataset( D )

  # get train and test (tt) split indexes
  tt_split_indexes = func_tt_split( *args_tt_split )
  return ( X, y, tt_split_indexes )


___________
def main():
  for (f_tt_split, args_tt_split) in list_func_tt_split:
    ( X, y, tt_split_indexes ) = train_test_split_recipe(
                                   (fileName, list_featureName),
                                   f_tt_split, *args_tt_split )
    show_train_test_split( X, y, tt_split_indexes, numFirstRows=10 )
```

`split_and_eval.py`

… use split-indexes to show (summarize) train-test-split

the `split` method accepts the dataset features, `X`, and class, `y`, and returns the train and test indexes over `X` and `y`

```
def show_train_test_split( X, y, tt_split_indexes, numFirstRows=10 ):

  for ( train_index, test_index ) in tt_split_indexes.split( X, y ):
    print( "summarized data (max = {n:d} instances)".
                                       format( n=numFirstRows ) )
    train_index = train_index[0:numFirstRows]
    test_index = test_index[0:numFirstRows]

    print( "\n> train-indexes, test-indexes" )
    print("train-indexes:", train_index )
    print(" test-indexes:", test_index )

    X_train, y_train = X[train_index], y[train_index]
    X_test, y_test   = X[test_index],  y[test_index]

    print("\nX_train, y_train" )
    print( X_train, y_train, sep="\n" )
    print("\nX_test, y_test" )
    print( X_test, y_test, sep="\n" )
```

split_and_eval.py

Validation – “holdout” and “stratified holdout”

- holdout – reserves an amount for testing and remainder for training
  - practice is common to hold out 1/3 for testing (remaining 2/3 for training)
- ... but, how to choose the testing dataset?
  - e.g., a class value that only has testing examples can not be trained!
- stratified holdout method – safeguard against uneven representation
  - calculate the percentage of each class value in the original dataset
  - build the training dataset keeping the percentage of each class value
  - ... choose randomly each example from the training dataset
- e.g., dataset with 200 examples from A, 300 from B and 100 from C
  - the original dataset distribution is [A: 2/6=1/3, B: 3/6=1/2, C: 1/6]
  - ... reserve 1/3 of the examples for testing (= 200)
  - ... so, testing examples for each class value are: [A: 67, B: 100, C: 33]

… “holdout” – implemented & used

```
(holdout, (1.0/3.0, seed))
```

```
def holdout( test_size, seed=None ):
  # yields indices to random split once the data into:
  # - one train dataset and one test dataset
  tt_split_indexes = ShuffleSplit(
                       n_splits=1, test_size=test_size, random_state=seed )
  return tt_split_indexes
```

```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

split_and_eval.py

```
train_test_split: holdout

> train-indexes, test-indexes
train-indexes: [4 1 0 3]
 test-indexes: [5 2]

X_train, y_train
[[15 26]
 [12 23]
 [11 22]
 [14 25]]
[0 1 0 0]

X_test, y_test
[[16 27]
 [13 24]]
[1 1]
```

… “stratified holdout” – implemented & used

```
(stratified_holdout, (1.0/3.0, seed))
```

```
def stratified_holdout( test_size, seed=None ):
  # yields indices to random split of data into:
  # - one train dataset and one test dataset
  # - datasets preserve the percentage of samples for each class
  tt_split_indexes = StratifiedShuffleSplit(
                       n_splits=1, test_size=test_size, random_state=seed )
  return tt_split_indexes
```

```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

```
split_and_eval.py
```

```
train_test_split: stratified_holdout

> train-indexes, test-indexes
train-indexes: [1 3 0 5]
 test-indexes: [2 4]

X_train, y_train
[[12 23]
 [14 25]
 [11 22]
 [16 27]]
[1 0 0 1]

X_test, y_test
[[13 24]
 [15 26]]
[1 0]
```

... validation with “repeated (stratified) holdout”

- Mitigate the bias of a single holdout choice
  - adopting the idea of repeated holdout

- i.e., iteratively repeat the whole training and testing process
  - with different training and testing datasets

- ... in each iterations consider a new testing dataset
  - consider a certain proportion for testing e.g., 1/3 of original dataset
  - build each testing dataset using e.g., stratified holdout
  - estimate error as the average error rate from the several iterations

- In this approach each dataset plays a single role
  - i.e., the testing dataset is never used for training
  - ... alternative is to exchange roles, i.e., train with test data, test with train data and average the 2 results; only possible for a 50:50 partition
  - but, better use more than 1/2 for training even at expense of test data!

Paulo Trigo Silva

… “repeated holdout” – implemented & used

```
(repeated_holdout, (1.0/3.0, 2, seed))
```

```
def repeated_holdout( test_size, n_repeat, seed=None ):
  # yields indices to random split of data into:
  # - a given number (n_repeat) of training and test datasets
  tt_split_indexes = ShuffleSplit(
                       n_splits=n_repeat, test_size=test_size, random_state=seed )
  return tt_split_indexes
```

```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

```
split_and_eval.py
```

```
train_test_split: repeated_holdout

> train-indexes, test-indexes
train-indexes: [4 1 0 3]
 test-indexes: [5 2]

X_train, y_train
[[15 26]
 [12 23]
 [11 22]
 [14 25]]
[0 1 0 0]

X_test, y_test
[[16 27]
 [13 24]]
[1 1]
```

```
> train-indexes, test-indexes
train-indexes: [4 2 0 3]
 test-indexes: [5 1]

X_train, y_train
[[15 26]
 [13 24]
 [11 22]
 [14 25]]
[0 1 0 0]

X_test, y_test
[[16 27]
 [12 23]]
[1 1]
```

… “repeated stratified holdout” – implemented & used

```
(repeated_stratified_holdout, (1.0/3.0, 2, seed))
```

```
def repeated_stratified_holdout( test_size, n_repeat, seed=None ):
  # yields indices to random split of data into:
  # - a given number (n_repeat) of training and test datasets
  # - datasets preserve the percentage of samples for each class
  tt_split_indexes = StratifiedShuffleSplit(
                      n_splits=n_repeat, test_size=test_size, random_state=seed )
  return tt_split_indexes
```

```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

```
split_and_eval.py
```

```
train_test_split: repeated_stratified_holdout

> train-indexes, test-indexes
train-indexes: [1 3 0 5]
 test-indexes: [2 4]

X_train, y_train
[[12 23]
 [14 25]
 [11 22]
 [16 27]]
[1 0 0 1]

X_test, y_test
[[13 24]
 [15 26]]
[1 0]
```

```
> train-indexes, test-indexes
train-indexes: [3 2 1 4]
 test-indexes: [0 5]

X_train, y_train
[[14 25]
 [13 24]
 [12 23]
 [15 26]]
[0 1 1 0]

X_test, y_test
[[11 22]
 [16 27]]
[0 1]
```

Paulo Trigo Silva

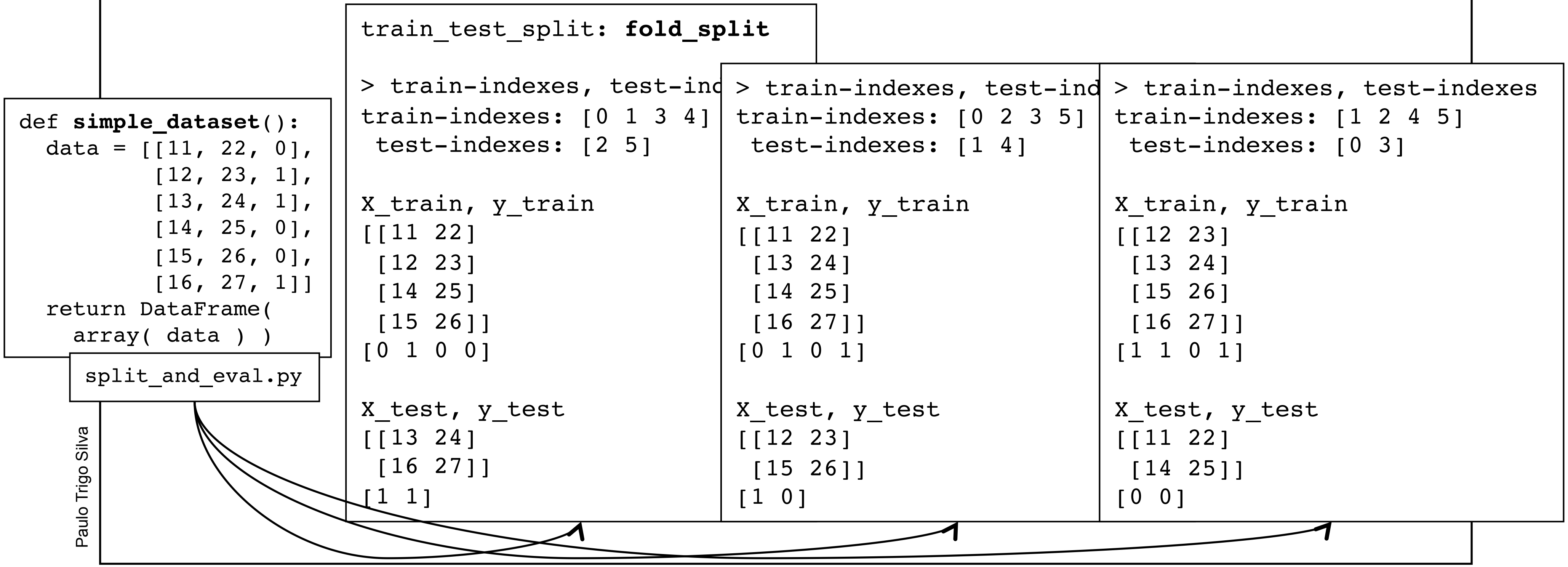
Method of “cross-validation”

- Variation of the role exchange method with more than 2 partitions
  - define a fixed number of folds or partitions of data
- If we consider K partitions of the original dataset
  - split data into K folds (partitions) with (approximately) same size
  - make K iterations of training and testing
  - choose, in each iteration, a testing fold (partition) not yet used
  - ... i.e., each fold is used for testing in 1 iteration and for training in k – 1
- e.g., three-fold (3-fold) cross-validation considers 3 folds (partitions)
  - i.e., executes 3 iterations of the training and testing process
  - ... in each iteration 2/3 of data is used for training and 1/3 for testing
- ... usually stratification is adopted – stratified 3-fold cross-validation
  - i.e., guarantee that the partitions keep the class values' proportionality

… "fold-split" – implemented & used

```
(fold_split, (3, seed))
```

```
def fold_split( k_folds, seed=None ):
  # yields indices to random split of data into:
  # - k consecutive folds (k_folds parameter)
  # - each fold used once as validation; the k-1 remaining folds as training dataset
  # - shuffle=True, so dataset is shuffled (random split) before building the folds
  tt_split_indexes = KFold( n_splits=k_folds, shuffle=True, random_state=seed )
  return tt_split_indexes
```



```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

```
split_and_eval.py
```

```
train_test_split: fold_split

> train-indexes, test-inc
train-indexes: [0 1 3 4]
 test-indexes: [2 5]

X_train, y_train
[[11 22]
 [12 23]
 [14 25]
 [15 26]]
[0 1 0 0]

X_test, y_test
[[13 24]
 [16 27]]
[1 1]
```

```
> train-indexes, test-ind
train-indexes: [0 2 3 5]
 test-indexes: [1 4]

X_train, y_train
[[11 22]
 [13 24]
 [14 25]
 [16 27]]
[0 1 0 1]

X_test, y_test
[[12 23]
 [15 26]]
[1 0]
```

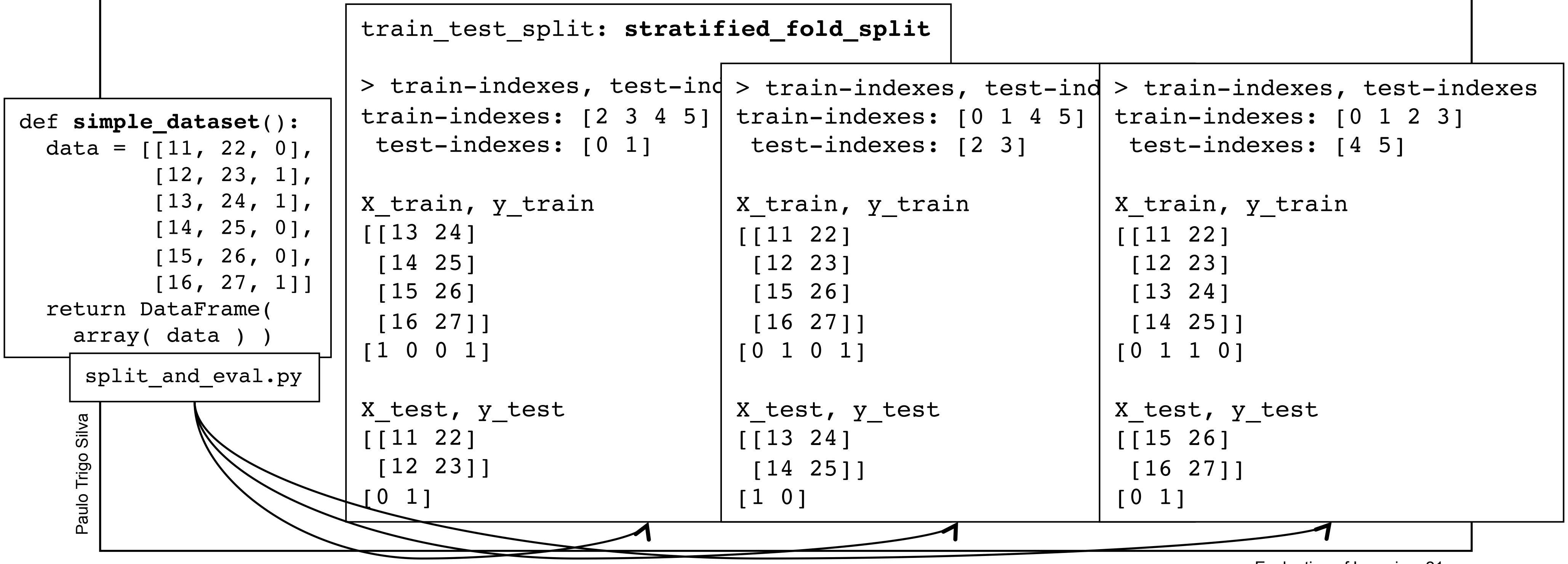
```
> train-indexes, test-indexes
train-indexes: [1 2 4 5]
 test-indexes: [0 3]

X_train, y_train
[[12 23]
 [13 24]
 [15 26]
 [16 27]]
[1 1 0 1]

X_test, y_test
[[11 22]
 [14 25]]
[0 0]
```

Paulo Trigo Silva

… “stratified-fold-split” – implemented & used

```
(stratified_fold_split, (3, seed))
```

```
def stratified_fold_split( k_folds, seed=None ):
  # yields indices to random split of data into:
  # - k consecutive folds (k_folds parameter)
  # - each fold used once as validation; the k-1 remaining folds as training dataset
  # - shuffle=True, so dataset is shuffled (random split) before building the folds
  # - datasets preserve the percentage of samples for each class
  tt_split_indexes = StratifiedKFold( n_splits=k_folds, shuffle=True, random_state=seed )
  return tt_split_indexes
```



```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

```
split_and_eval.py
```

```
train_test_split: stratified_fold_split

> train-indexes, test-inc
train-indexes: [2 3 4 5]
 test-indexes: [0 1]

X_train, y_train
[[13 24]
 [14 25]
 [15 26]
 [16 27]]
[1 0 0 1]

X_test, y_test
[[11 22]
 [12 23]]
[0 1]
```

```
> train-indexes, test-ind
train-indexes: [0 1 4 5]
 test-indexes: [2 3]

X_train, y_train
[[11 22]
 [12 23]
 [15 26]
 [16 27]]
[0 1 0 1]

X_test, y_test
[[13 24]
 [14 25]]
[1 0]
```

```
> train-indexes, test-indexes
train-indexes: [0 1 2 3]
 test-indexes: [4 5]

X_train, y_train
[[11 22]
 [12 23]
 [13 24]
 [14 25]]
[0 1 1 0]

X_test, y_test
[[15 26]
 [16 27]]
[0 1]
```

Paulo Trigo Silva

… "repeated-fold-split" – implemented & used

```
(repeated_fold_split, (3, 2, seed))
```

```
def repeated_fold_split( k_folds, n_repeat, seed=None
  # yields indices to random split of data into:
  # - k consecutive folds (k_folds parameter)
  # - each fold used once as validation; the k-1 remaining folds as training dataset
  # - repeats (n_repeat times) KFold with different randomization in each repetition
  tt_split_indexes = RepeatedKFold( n_splits=k_folds, n_repeats=n_repeat,
                                    random_state=seed )
  return tt_split_indexes
```

```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

```
split_and_eval.py
```

```
train_test_split: repeated_fold_split

> train-indexes, test-indexes
train-indexes: [0 1 3 4]
 test-indexes: [2 5]
...

> train-indexes, test-indexes
train-indexes: [0 2 3 5]
 test-indexes: [1 4]
...

> train-indexes, test-indexes
train-indexes: [1 2 4 5]
 test-indexes: [0 3]
...
```

```
> train-indexes, test-indexes
train-indexes: [0 2 3 4]
 test-indexes: [1 5]
...

> train-indexes, test-indexes
train-indexes: [0 1 3 5]
 test-indexes: [2 4]
...

> train-indexes, test-indexes
train-indexes: [1 2 4 5]
 test-indexes: [0 3]
...
```

Paulo Trigo Silva

… “repeated-stratified-fold-split” – implemented & used

```
(repeated_stratified_fold_split, (3, 2, seed))
```

```
def repeated_stratified_fold_split( k_folds, n_repeat, seed=None ):
  # yields indices to random split of data into:
  # - k consecutive folds (k_folds parameter)
  # - each fold used once as validation; the k-1 remaining folds as training dataset
  # - repeats (n_repeat times) StratifiedKFold with randomization in each repetition
  tt_split_indexes = RepeatedStratifiedKFold( n_splits=k_folds, n_repeats=n_repeat,
                                              random_state=seed )
  return tt_split_indexes
```

```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

```
split_and_eval.py
```

```
train_test_split: repeated_stratified_fold_split

> train-indexes, test-indexes
train-indexes: [1 2 3 4]
 test-indexes: [0 5]
...

> train-indexes, test-indexes
train-indexes: [0 2 4 5]
 test-indexes: [1 3]
...

> train-indexes, test-indexes
train-indexes: [0 1 3 5]
 test-indexes: [2 4]
...
```

```
> train-indexes, test-indexes
train-indexes: [0 1 2 4]
 test-indexes: [3 5]
...

> train-indexes, test-indexes
train-indexes: [2 3 4 5]
 test-indexes: [0 1]
...

> train-indexes, test-indexes
train-indexes: [0 1 3 5]
 test-indexes: [2 4]
...
```

... “standard” for error rate evaluation

- The “standard” is to apply the *stratified 10-fold cross-validation*
  - build, randomly, 10 partitions of the complete dataset
  - each partition keeps proportion of class values (relatively to dataset)
  - use 9 folds (i.e. 90% of dataset) as training data; use 1 partition as testing data and compute the error rate
  - execute the learning 10 times always changing the testing fold
  - compute the average of the 10 error rate to get a global error estimate
- Why using 10 folds (data partitions)?
  - experimental tests and some theoretical evidence supports this “10”
  - tests also show that the use of stratification improves results
- It is also “standard” to repeat 10 × stratified 10-fold cross-validation
  - … and average the results
  - i.e., execute 100 times the learning algorithm on datasets that are 9/10 the size of original; ... getting a good performance measure is intensive!

***implemented**, for example, as* `(repeated_stratified_fold_split, (10, 10, seed))`

Another evaluation method – "leave-one-out"

- The leave-one-out is simply a N-fold cross-validation
  - where N is the number of instances in the dataset
- i.e., in each iteration, just 1 (one) single instance is left out for testing
  - the learning method is trained on all the remaining instances
  - ... and it is evaluated as correct / incorrect on the remaining instance
  - the average of results from the N evaluations is the final error estimate
- Advantage: the greatest possible amount of data is used for training
  - gets the maximum from a small (or medium) size dataset
- Advantage: no random sampling involved; procedure is deterministic
  - there is no point in repeating the process; always the same result
- Disadvantage: high computational cost
  - execute N time the algorithm is not feasible with very large datasets

… “leave-one-out” – implemented & used

```
(leave_one_out, ())
```

```
def leave_one_out():
  # yields indices to split of data so that:
  # - each sample is used once as a test set (singleton), and
  # - the remaining samples form the training set
  # - same as KFold(n_splits=n), where n is number of dataset samples
  tt_split_indexes = LeaveOneOut()
  return tt_split_indexes
```

```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

```
split_and_eval.py
```

```
train_test_split: leave_one_out

> train-indexes, test-indexes
train-indexes: [1 2 3 4 5]
 test-indexes: [0]
...

> train-indexes, test-indexes
train-indexes: [0 2 3 4 5]
 test-indexes: [1]
...

> train-indexes, test-indexes
train-indexes: [0 1 3 4 5]
 test-indexes: [2]
...
```

```
> train-indexes, test-indexes
train-indexes: [0 1 2 4 5]
 test-indexes: [3]
...

> train-indexes, test-indexes
train-indexes: [0 1 2 3 5]
 test-indexes: [4]
...

> train-indexes, test-indexes
train-indexes: [0 1 2 3 4]
 test-indexes: [5]
...
```

… and “leave-p-out” – implemented & used

```
def leave_p_out( p ):
  # yields indices to split of data so that:
  # - testing on all distinct samples of size p, and
  # - the remaining n-p samples form the training set in each iteration
  # - NOT same as KFold(n_splits=n_samples//p), which creates non-overlapping test sets
  tt_split_indexes = LeavePOut( p=p )
  return tt_split_indexes
```

```
(leave_p_out, (2, ))
```

```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

```
split_and_eval.py
```

```
train_test_split: leave_p_out

> train-indexes, test-indexes
train-indexes: [2 3 4 5]
 test-indexes: [0 1]
...
> train-indexes, test-indexes
train-indexes: [1 3 4 5]
 test-indexes: [0 2]
...
> train-indexes, test-indexes
train-indexes: [1 2 4 5]
 test-indexes: [0 3]
...
> train-indexes, test-indexes
train-indexes: [1 2 3 5]
 test-indexes: [0 4]
...
> train-indexes, test-indexes
train-indexes: [1 2 3 4]
 test-indexes: [0 5]
```

```
> train-indexes, test-indexes
train-indexes: [0 3 4 5]
 test-indexes: [1 2]
...
> train-indexes, test-indexes
train-indexes: [0 2 4 5]
 test-indexes: [1 3]
...
> train-indexes, test-indexes
train-indexes: [0 2 3 5]
 test-indexes: [1 4]
...
> train-indexes, test-indexes
train-indexes: [0 2 3 4]
 test-indexes: [1 5]
...
> train-indexes, test-indexes
train-indexes: [0 1 4 5]
 test-indexes: [2 3]
...
> train-indexes, test-indexes
train-indexes: [0 1 3 5]
 test-indexes: [2 4]
...
> train-indexes, test-indexes
train-indexes: [0 1 3 4]
 test-indexes: [2 5]
...
```

```
> train-indexes, test-indexes
train-indexes: [0 1 2 5]
 test-indexes: [3 4]
...
> train-indexes, test-indexes
train-indexes: [0 1 2 4]
 test-indexes: [3 5]
...
> train-indexes, test-indexes
train-indexes: [0 1 2 3]
 test-indexes: [4 5]
...
```

... “leave-one-out” – advantages and disadvantages

- *recall*: a method is considered stratified if each training dataset
  - keeps the same proportion of class values as in the original dataset
- The method “leave-one-out” cannot be stratified
  - or worst than that, it guarantees a non-stratified sample!
- The worst scenario (although very artificial) of “leave-one-out” is
  - to imagine a completely random dataset
  - ... with two class values, both with the same number of examples
- ... with random data and 2 class values, the best a classifier can do
  - is to predict the majority class, giving a true error rate of 50%
- ... but, in each fold of “leave-one-out”
  - the opposite class to the test instance is in the majority, and therefore
  - ... prediction will always be incorrect, leading to an error rate of 100%!

Another evaluation method – “bootstrap”

- Based on the statistical procedure of *sampling with replacement*
  - the same instance can be selected (sampled) several times
  - … idea is to sample the dataset with replacement to form a training set
- Given a dataset of size *N*
  - select, from the original dataset, *N* instances
  - ... but, each selected instance remains in the original dataset
  - ... and therefore the same instance can be selected several times
  - the *N* selected instances constitute the (new) training dataset
  - the instances that don’t occur in the training dataset belong to testing
- Then, what is the expected percentage of testing examples?
  - or, what is the chance of an instance not being selected for training?

... “bootstrap” – what percentage of testing examples?

- Or, what probability for an example not being selected for training?
  - each instance has $1/N$ probability of being picked each time
  - ... so, $1 - 1/N$ is the probability of not being picked for training

- ... the picking events are independent (because of the replacement)
  - so, for N picking opportunities the probability of not belonging to training
  - is $(1 - 1/N)^N \approx e^{-1} \approx 0.368$,
  - … where e is the base of natural logarithms (2.7183); not the error rate!

- i.e., given an original dataset the bootstrap should originate
  - a testing dataset with 36.8% of the size of the original dataset
  - so, about 63.2% of different examples in the training dataset
  - … some instances will be repeated in the training dataset, bringing it up to a total size of N instances, the same as in the original dataset

... "bootstrap" – compensate pessimistic error estimate

- The bootstrap error estimation is very pessimistic, because
  - the training set (although of size *N*) contains only 63% of instances
  - ... compare with the 90% used in the 10-fold cross-validation!
- To compensate the pessimistic perspective combine with optimism
  - the most optimistic is to consider the error of the training dataset
  - … the error of the training dataset, gives a very optimistic estimate of the true error and should not be used as an error figure on its own
  - … but bootstrap combines it with test error rate to give a final estimate
  - i.e., combine the error of the testing set with error of the training set
- ... so, to compensate the pessimistic perspective with optimism,
  - calculate: $\text{error} = 0.632 \times \text{error}_{\text{test}} + 0.368 \times \text{error}_{\text{train}}$
- The whole bootstrap procedure is repeated several times, with
  - different replacement samples for training set; the results are averaged

… “bootstrap” – `SplitOnce` – implemented & used

```
(bootstrap_split_once, (seed, ))
```

```
# different implementation because:
# - it must use "resample" which does not return indexes
def bootstrap_split_once( seed=None ):
  tt_split_indexes = MyBootstrapSplitOnce( seed )
  return tt_split_indexes
```

```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

```
split_and_eval.py
```

```
train_test_split: bootstrap_split_once

> train-indexes, test-indexes
train-indexes: [4, 1, 3, 3, 4, 1]
 test-indexes: [0, 2, 5]

X_train, y_train
[[15 26]
 [12 23]
 [14 25]
 [14 25]
 [15 26]
 [12 23]]
[0 1 0 0 0 1]

X_test, y_test
[[11 22]
 [13 24]
 [16 27]]
[0 1 1]
```

Paulo Trigo Silva

… “bootstrap” – SplitRepeated – implemented & used

```
(bootstrap_split_repeated, (2, seed))
```

```
# different implementation because:
# - it must use "resample" which does not return indexes
def bootstrap_split_repeated( n_repeat, seed=None ):
  tt_split_indexes = MyBootstrapSplitRepeated( n_repeat, seed )
  return tt_split_indexes
```

```
def simple_dataset():
  data = [[11, 22, 0],
          [12, 23, 1],
          [13, 24, 1],
          [14, 25, 0],
          [15, 26, 0],
          [16, 27, 1]]
  return DataFrame(
    array( data ) )
```

split_and_eval.py

```
train_test_split: bootstrap_split_repeated

> train-indexes, test-indexes
train-indexes: [3, 5, 0, 1, 0, 4]
 test-indexes: [2]

X_train, y_train
[[14 25]
 [16 27]
 [11 22]
 [12 23]
 [11 22]
 [15 26]]
[0 1 0 1 0 0]

X_test, y_test
[[13 24]]
[1]
```

```
train-indexes: [2, 1, 3, 4, 2, 5]
 test-indexes: [0]

X_train, y_train
[[13 24]
 [12 23]
 [14 25]
 [15 26]
 [13 24]
 [16 27]]
[1 1 0 0 1 1]

X_test, y_test
[[11 22]]
[0]
```

Paulo Trigo Silva

… “bootstrap” – but, ... how to implement?

the Python `sklearn` does not directly implements the bootstrap method!

but, `sklearn` provides a function (`resample`) that we can use to resample a dataset and (with that support) implement bootstrap

```
from sklearn.utils import resample
# we also need to “score” the model
from sklearn.metrics import accuracy_score, precision_score, recall_score
# abc = Abstract Base Classes
from abc import ABC, abstractmethod


# abstract class that defines general behavior:
# - split and score
class MyBootstrap( ABC ):
  @abstractmethod
  def get_seed(): pass

 def __init__( self, seed=None ):
    self.seed = seed
    self.reset_tt_split_indexes()

  def reset_tt_split_indexes( self ):
    self.tt_split_indexes = None
```

… “bootstrap” – build “my” train-test-split indexes

the `resample` default strategy implements one step of bootstrap method,
but, returns new datasets and not indexes!

for compatibility with previous code we use `resample` and also return indexes

```
...

  def split( self, X, y=None ):
    # if train|test split already exists, then return
    if self.tt_split_indexes != None: return self.tt_split_indexes

    #_____________________________
    # build a new train|test split
    dim_dataset = len( X )
    indexes = list( range( dim_dataset ) )

    # training set is created from resamples (samples with reposition)
    # n_samples = number of samples to generate; None means “the size of dataset”
    seed = self.get_seed()
    train_indexes = resample( indexes, n_samples=None, random_state=seed )

    #testing set is created from individuals, i, not in training set
    test_indexes = [i for i in indexes if i not in train_indexes]

    self.tt_split_indexes = [ ( train_indexes, test_indexes ) ]
    return self.tt_split_indexes
```

… “bootstrap” – “SplitOnce” and “SplitRepeated”

```
class MyBootstrapSplitOnce( MyBootstrap ):
  def get_seed( self ):
    return self.seed


class MyBootstrapSplitRepeated( MyBootstrap ):
 def __init__( self, n_repeat, seed=None ):
    super().__init__( seed )
    self.n_repeat = n_repeat
    self.tt_split_repeated_indexes = None

 def get_seed( self ):
    seed_current = self.seed
    if self.seed != None: self.seed = self.seed  + 1
    return seed_current

 def split( self, X, y=None ):
    # if train|test split-repeated already exists, then return
    if self.tt_split_repeated_indexes != None:
      return self.tt_split_repeated_indexes

    # build a new train|test split-repeated
    self.tt_split_repeated_indexes = list()
    for i in range( self.n_repeat ):
      self.reset_tt_split_indexes()
      self.tt_split_repeated_indexes = self.tt_split_repeated_indexes + \
                                       super().split( X )
    return self.tt_split_repeated_indexes
```

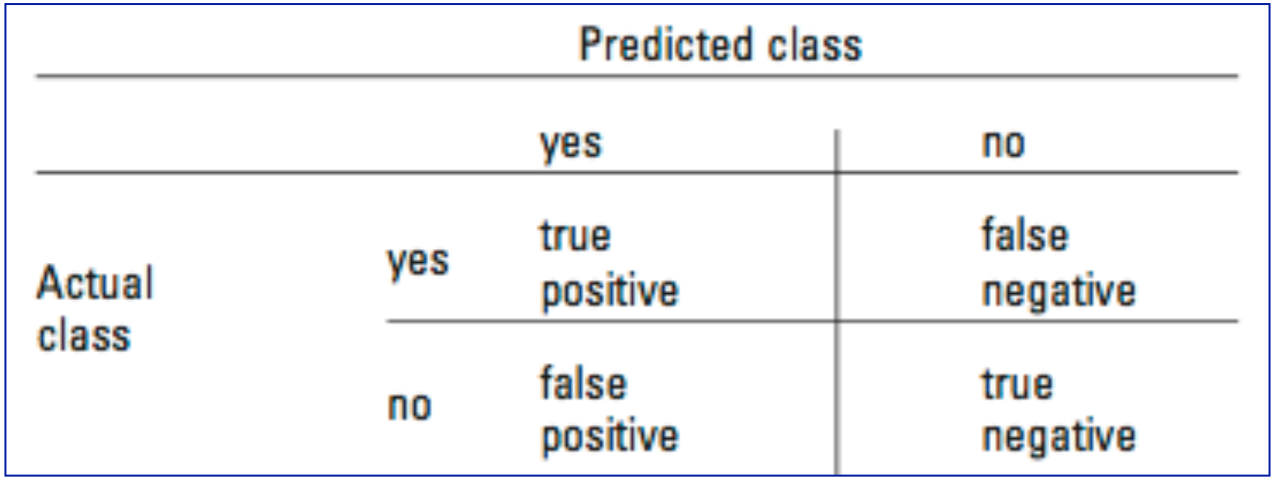
... “bootstrap” – advantages and disadvantages

- Given a small size dataset
  - the bootstrap is, usually, the best method to estimate error
  - ... bootstrap is less computationally demanding than leave-one-out
- The worst scenario (although very artificial) of bootstrap is
  - just like in the worst scenario of leave-one-out
  - a completely random dataset
  - ... with two class values, both with the same number of examples
- The true error rate is 50% for any prediction rule
  - but, a method that memorizes (overfitting) the training dataset
  - ... would give $\text{error}_{\text{train}} = 0$
  - i.e., $\text{error} = 0.632 \times \text{error}_{\text{test}} + 0.368 \times 0 \Leftrightarrow \text{error} = 0.632 \times 0.5 + 0 = 0.316$
  - i.e., error of 31.6% which is misleadingly optimistic (regarding the 50%)

Type of error and associated cost

- Different types of error can originate different costs
  - e.g., cost of sending mail to someone that doesn't respond is less than the lost-business cost of not sending it to someone that would respond
  - e.g., the cost of lending to a defaulter is far greater than the lost-business cost of refusing a loan to a non-defaulter
- A problem of classification with 2 classes (e.g., *yes*, *no*)
  - has 4 possibilities: 2 true and 2 false
- ... the 2 true correspond to correct classifications
  - true positive, e.g., predicted class is *yes* and actual class is *yes*
  - true negative, e.g., predicted class is *no* and actual class is *no*
- ... the 2 false correspond to incorrect classifications
  - false positive, e.g., predicted class is *yes* and actual class is *no*
  - false negative, e.g., predicted class is *no* and actual class is *yes*

Different outcomes of a two-class prediction



		Predicted class	
		yes	no
Actual class	yes	true positive	false negative
	no	false positive	true negative

The **true positives** (TP) and **true negatives** (TN) are correct classifications.

A **false positive** (FP) occurs when the outcome is incorrectly predicted as yes (or positive) when it is actually no (negative).

A **false negative** (FN) occurs when the outcome is incorrectly predicted as negative when it is actually positive

... metrics for the types of error (with 2 classes)

Consider the set of class values:
{ *yes*, *no* }

		predicted class (algorithm)	
		yes	*no*
actual class (human)	*yes*	true positive (TP)	falso negativo (FN)
	no	falso positivo (FP)	true negative (TN)

accuracy | ** or, **success rate ≡ (TP + TN) / (TP + TN + FP + FN)
proportion of correct classifications out of all classified

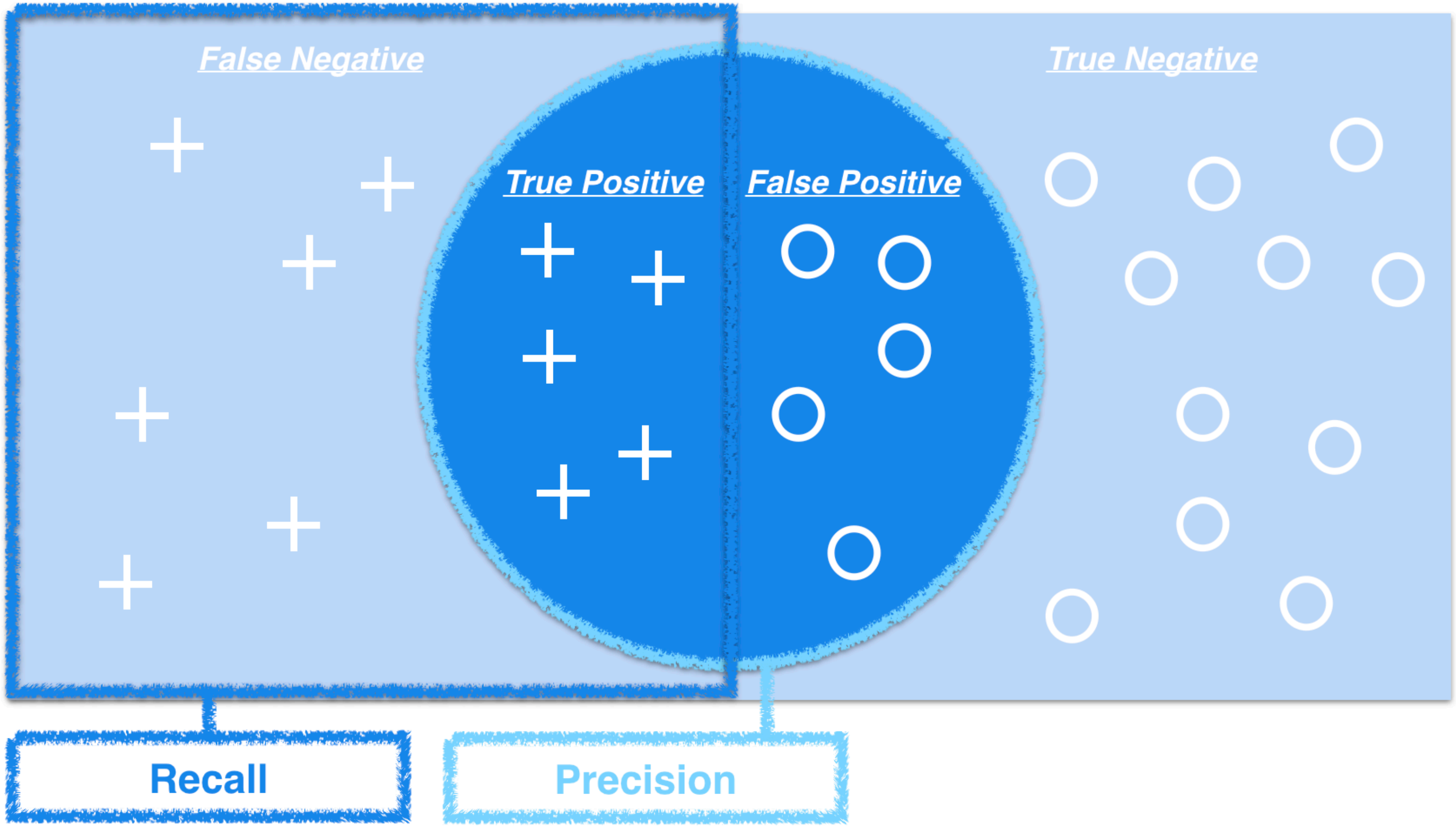
precision (*p*) | or, **positive predictive value** ≡ TP / (TP + FP)
proportion of positives correctly predicted out of all the positive predictions

recall (*r*) | or, **sensitivity** or, **true positive rate** ≡ TP / (TP + FN)
proportion of positives correctly predicted out of all the positives

F1-score | or, **harmonic average of *p* and *r*** ≡ $[\frac{1}{2} (\boldsymbol{p}^{-1} + \boldsymbol{r}^{-1})]^{-1}$ **=** $2 \cdot \boldsymbol{p} \cdot \boldsymbol{r} / (\boldsymbol{p} + \boldsymbol{r})$
conveys the balance between ***p*** and ***r***; the range is [0..1], the best value is 1
(… *a remark: the **harmonic average** is often used to get the average of rates*)

Paulo Trigo Silva

... precision & recall – a visual perspective

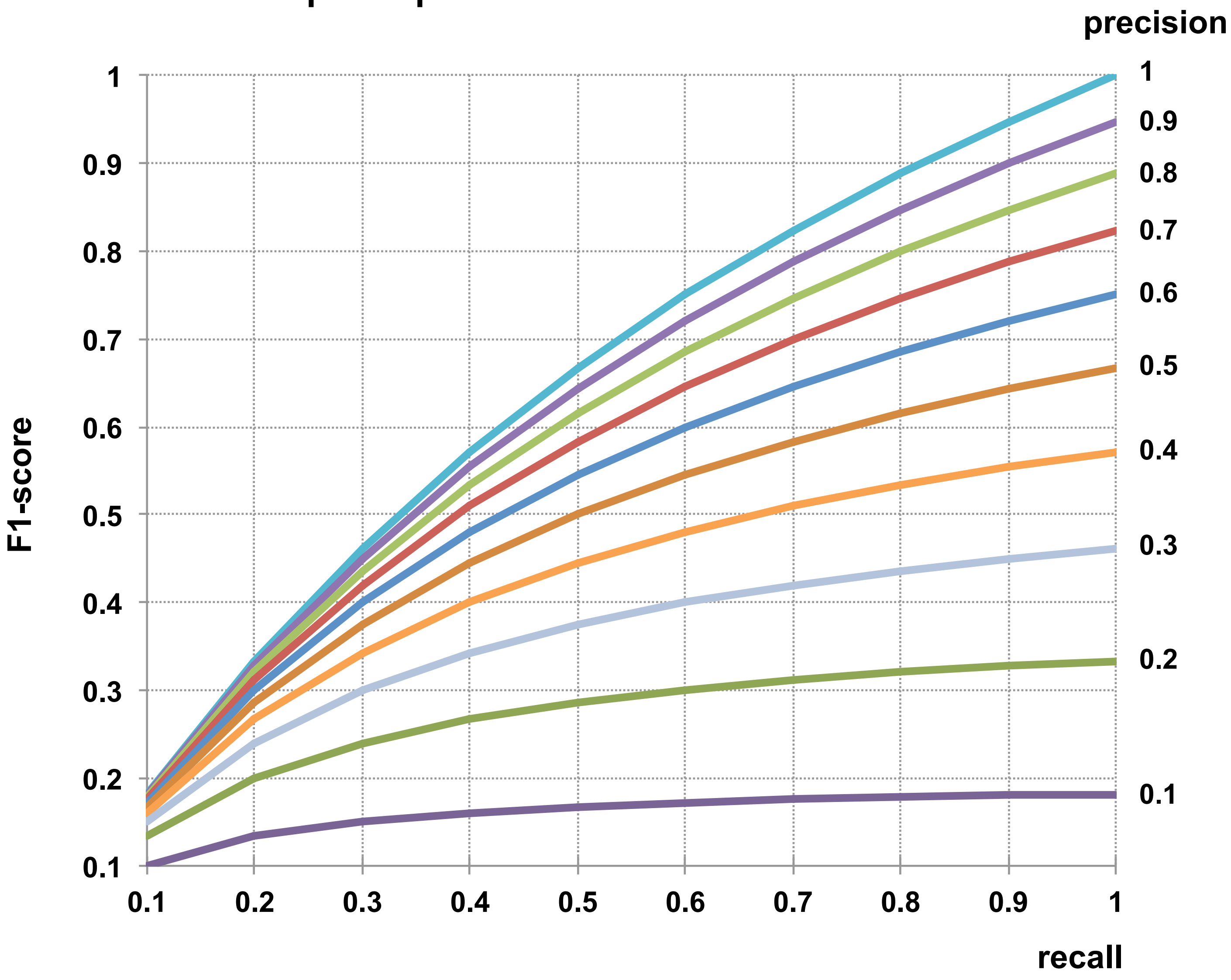


actual class of:
+ is ***positive***
O is ***negative***

***classified* as:**
positive ***negative***

Paulo Trigo Silva

... F1-score – a visual perspective



as **F1-score** gets higher values (i.e., closer to 1),

also, both **precision** and **recall** are getting higher values

Paulo Trigo Silva

... and now we implement a “general” score recipe

- **build** a list of classifiers (`list_func_classifier`)
- **build** a list of score metrics (`list_core_metric`)
- **apply** to each classifier the list of metrics

```
list_func_classifier = \
  [
    (GaussianNB,             ()),
    (DecisionTreeClassifier, ())
  ]


list_score_metric = \
  [
    (accuracy_score,    {}),
    (precision_score,   {"average":"weighted"}),
    (recall_score,      {"average":"weighted"}),
    (f1_score,          {"average":"weighted"}),
    (cohen_kappa_score, {})
  ]
```

split_and_eval.py

… so, first extend “main” to include classification & score

```
def main():
  for (f_tt_split, args_tt_split) in list_func_tt_split:
    ( X, y, tt_split_indexes ) = train_test_split_recipe(
                                   (fileName, list_featureName),
                                   f_tt_split, *args_tt_split )
    show_train_test_split( X, y, tt_split_indexes, numFirstRows=10 )

    for (f_classifier, args_classifier) in list_func_classifier:
      classifier = f_classifier( *args_classifier )

      for (f_score, keyword_args_score) in list_score_metric:
        score_all = score_recipe( classifier, X, y, tt_split_indexes,
                                 f_score, **keyword_args_score )

        show_score( score_all )
```

split_and_eval.py

… then, the “general” score recipe – overall description

- **build** a list of score metrics (`list_core_metric`)
- **build** a list of classifiers (`list_func_classifier`)
- **apply** to each classifier the list of metrics

the overall “general” score recipe

1. use **train** data (`X` and `y`) and **`fit`** function to build the classifier’s model
2. classify (predict) the **test** data (only `X`) using **`predict`** function
3. use **test** data (only `y`) and previous prediction (`y_predict`) to score

.. and we must be aware that:

- **precision** is ill-defined (and is set to zero) if there are no positive predictions
- **recall** and **F1-score** are ill-defined (and set to zero) if there are no true samples

Paulo Trigo Silva

… then, the “general” score recipe – implementation

```
def score_recipe( classifier, X, y, tt_split_indexes,
                  f_score, **keyword_args_score ):
    score_all_list = list()

    for ( train_index, test_index ) in tt_split_indexes.split( X, y ):
      X_train, y_train = X[train_index], y[train_index]
      X_test,  y_test  = X[test_index],  y[test_index]

      # fit (build) model using classifier, X_train and y_train
      classifier.fit( X_train, y_train )

      # predict using the model and X_test (testing dataset)
      y_predict = classifier.predict( X_test )

      # score the model using y_test (expected) and y_predict (predicted)
      score = f_score( y_test, y_predict, **keyword_args_score )

      score_all_list.append( score )

    return score_all_list
```

split_and_eval.py

Paulo Trigo Silva

… and, show the score mean and standard deviation

each evaluation method usually follows a “repetition approach” to attenuate the bias of (random) sampling; so, we compute its mean and std

```
def show_score( score_all ):
  if not score_all: return
  print( "::all-evaluated-datasets::" )

  all_evaluated_datasets = [i*100.0 for i in score_all]
  for v in all_evaluated_datasets: print( " %.2f%% " %(v), end="|" )

  if isinstance( score_all, list ): score_all = array( score_all )
  print( "%.2f%% (+/- %.2f%%)" % \
         (score_all.mean()*100.0, score_all.std()*100.0) )
```

```
train_test_split: stratified_fold_split
classifier: DecisionTreeClassifier


______________
score_method: accuracy_score
::all-evaluated-datasets::
 50.00% | 100.00% | 50.00% |
66.67% (+/- 23.57%)

______________
score_method: precision_score
::all-evaluated-datasets::
 25.00% | 100.00% | 25.00% |
50.00% (+/- 35.36%)
```

```
______________
score_method: recall_score
::all-evaluated-datasets::
 50.00% | 100.00% | 50.00% |
66.67% (+/- 23.57%)

______________
score_method: f1_score
::all-evaluated-datasets::
 33.33% | 100.00% | 33.33% |
55.56% (+/- 31.43%)
```

Multi-class prediction – show result with “confusion matrix”

- In a multi-class prediction the result on a test set is often displayed
  - as a square matrix with a row and a column for each class value
  - ... such two dimension representation is named as the *confusion matrix*
- Each element of the confusion matrix shows the number of which,
  - the actual class is the row and the predicted class is the column
- A good result on the test corresponds to
  - large numbers down the main diagonal, and
  - small numbers (ideally zero) in the remaining (off-diagonal) elements

confusion matrix example

		Predicted Class			
		A	B	C	TOTAL
Actual Class	A	88	10	2	100
	B	14	40	6	60
	C	18	10	12	40
	TOTAL	120	60	20	*200*

*What is the **accuracy** of the test represented in this confusion matrix?*

Paulo Trigo Silva

The confusion matrix and the random classifier

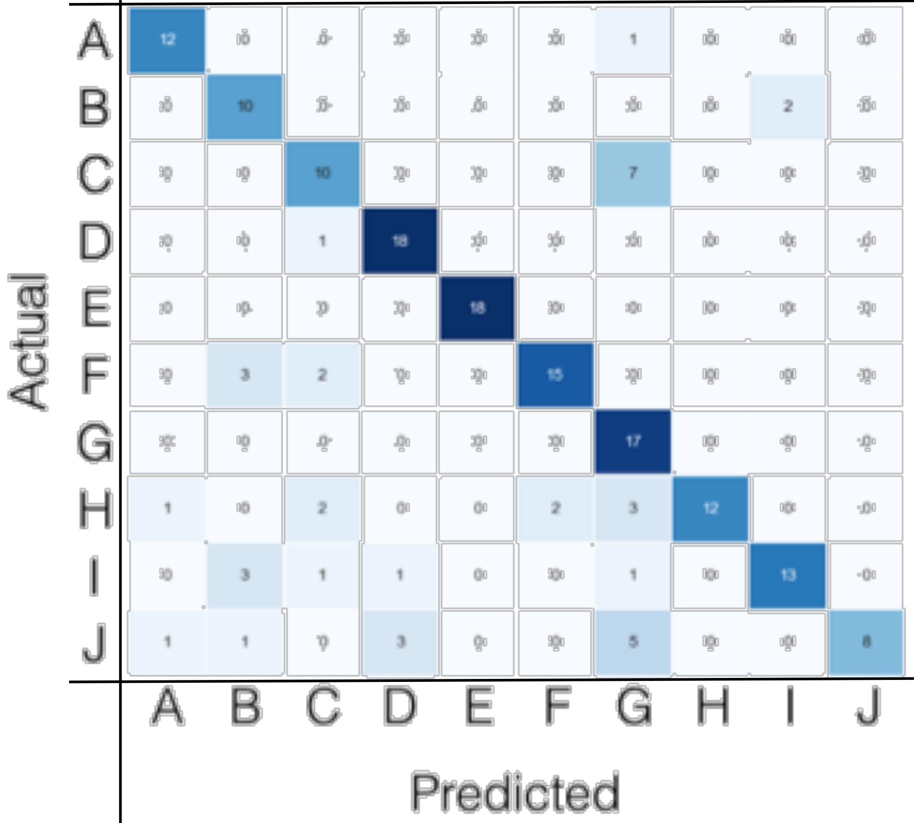
		Predicted Class			
		A	B	C	TOTAL
Actual Class	A	88	10	2	100
	B	14	40	6	60
	C	18	10	12	40
	TOTAL	120	60	20	*200*

accuracy =
(88 + 40 + 12) / 200 =
140 / 200 =
70%

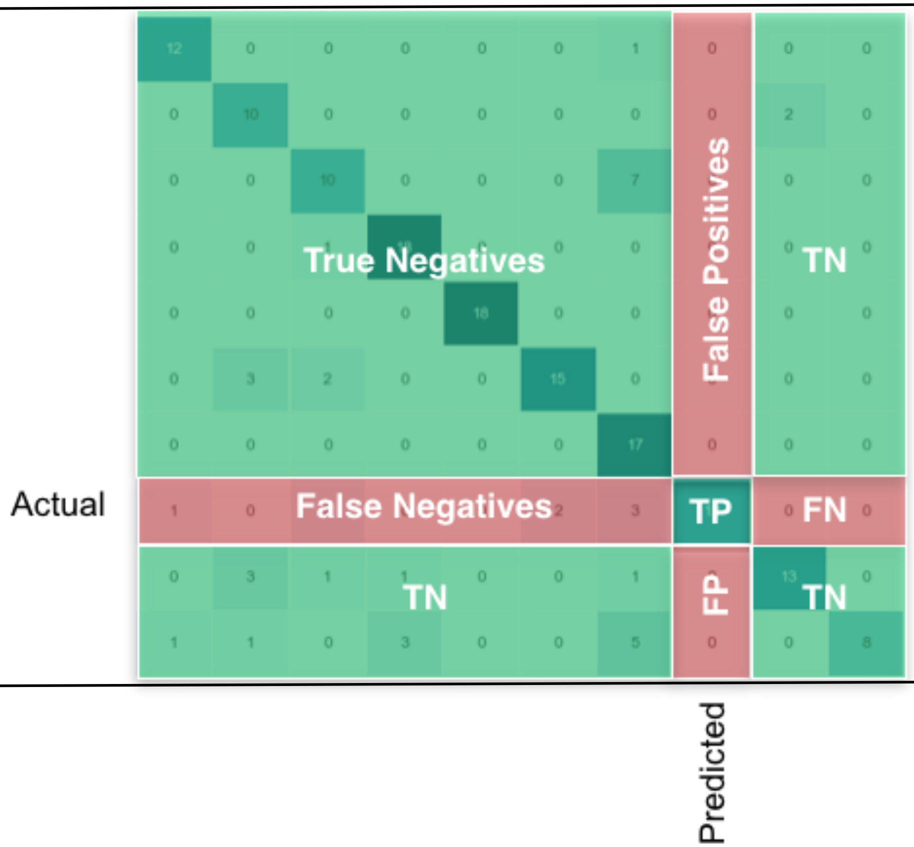
so, **multi-class** evaluation is an extension of binary case

- a collection of actual *vs* predicted binary outcomes, **one per class**
- overall evaluation metrics are averages across classes
- … and, there are <u>several ways to average</u> multi-class results

… the multi-class evaluation – how to “visualize” it?



	A	B	C	D	E	F	G	H	I	J
A	12	0	0	0	0	0	1	0	0	0
B	0	10	0	0	0	0	0	0	2	0
C	0	0	10	0	0	0	7	0	0	0
D	0	0	1	18	0	0	0	0	0	0
E	0	0	0	0	18	0	0	0	0	0
F	0	3	2	0	0	15	0	0	0	0
G	0	0	0	0	0	0	17	0	0	0
H	1	0	2	0	0	2	3	12	0	0
I	0	3	1	1	0	0	1	0	13	0
J	1	1	0	3	0	0	5	0	0	8



general idea – for **each class value**:
compute the metric (e.g., precision, recall, f1) as for binary classification

then, compute the **average** of each metric (over all class values)

Paulo Trigo Silva

(some) “average” alternatives – **macro**, **micro**, **weighted**

macro-average: each **class** has equal weight
(*so, classes with smaller number of instances will be favored*)

1. compute metric within each class
2. average resulting metrics across classes

micro-average: each **instance** has equal weight
(so, *classes with higher number of instances will be favored*)

1. aggregate outcomes across all classes
2. compute metric with aggregated outcomes

weighted-average: each **class is weighted** according to its proportion
(*so, the score is sensitive to the distribution of class values)*

1. compute metric within each class (as with macro-average)
2. compute the average weight by each (actual) class proportion

Paulo Trigo Silva

(some) “average” alternatives – **macro**-average

macro-average: each **class** has equal weight
(*so, classes with smaller number of instances will be favored*)

1. compute metric within each class
2. average resulting metrics across classes

		Predicted Class			
		A	B	C	TOTAL
Actual Class	A	88	10	2	100
	B	14	40	6	60
	C	18	10	12	40
	TOTAL	120	60	20	*200*

?
given this confusion matrix

what are the **macro**-average:
- precision
- recall
- F1-score

?

... **macro**-average (precision, recall, F1) – an example

		Predicted Class				
		A	B	C	TOTAL	**FN**
Actual Class	A	88	10	2	100	12
	B	14	40	6	60	20
	C	18	10	12	40	28
	TOTAL	120	60	20	*200*	

macro-average				
	TP	88	40	12
	FP	32	20	8

precision = TP / (TP + FP)

precision	0,73	0,67	0,60

macro-average precision (p)
0,67

recall = TP / (TP + FN)

recall	0,88	0,67	0,30

macro-average recall (r)
0,62

F1-score = 2*p*r / (p + r)

F1-score	0,80	0,67	0,40

macro-average F1-score
0,62

… the details,

```
precision(A) = 88/(88+32) = 0.73
precision(B) = 40/(40+20) = 0.67
precision(C) = 12/(12+8)  = 0.60


recall(A) = 88/(88+12) = 0.88
recall(B) = 40/(40+20) = 0.67
recall(C) = 12/(12+28) = 0.30
```

Paulo Trigo Silva

(some) “average” alternatives – **micro**-average

micro-average: each **instance** has equal weight
(so, *classes with higher number of instances will be favored*)

1. aggregate outcomes across all classes
2. compute metric with aggregated outcomes

		Predicted Class			
		A	B	C	TOTAL
Actual Class	A	88	10	2	100
	B	14	40	6	60
	C	18	10	12	40
	TOTAL	120	60	20	*200*

?
given this confusion matrix

what are the **micro**-average:
- precision
- recall
- F1-score

?

... **micro**-average (precision, recall, F1) – an example

		Predicted Class				
		A	B	C	TOTAL	**FN**
Actual Class	A	88	10	2	100	12
	B	14	40	6	60	20
	C	18	10	12	40	28
	TOTAL	120	60	20	*200*	**60**

micro-average	**TP**	88	40	12	**140**
	FP	32	20	8	**60**

precision = TP / (TP + FP)	
micro-average precision (p)	**0,70**

recall = TP / (TP + FN)	
micro-average recall (r)	**0,70**

F1-score = 2*p*r / (p + r)	
micro-average F1-score	**0,70**

… the details,

```
precision = 140/(140+60) = 0.70

recall    = !40/(140+60) = 0.70
```

accuracy =
(88 + 40 + 12) / 200 =
140 / 200 =
0.70

Note: if all class values are included, then “micro”-averaging in a multiclass setting will produce precision, recall and F1 that are all identical to accuracy.

Paulo Trigo Silva

(some) “average” alternatives – **weighted**-average

weighted-average: each **class is weighted** according to its proportion (*so, the score is sensitive to the distribution of class values)*

1. compute metric within each class (as with macro-average)
2. compute the average weight by each class proportion

		Predicted Class			
		A	B	C	TOTAL
Actual Class	A	88	10	2	100
	B	14	40	6	60
	C	18	10	12	40
	TOTAL	120	60	20	*200*

?

given this confusion matrix

what are the **weighted**-average:

- precision
- recall
- F1-score

?

Paulo Trigo Silva

... **weighted**-average (precision, recall, F1) – an example

		Predicted Class					
		A	B	C	TOTAL	**FN**	**weight**
Actual Class	A	88	10	2	100	12	0,50
	B	14	40	6	60	20	0,30
	C	18	10	12	40	28	0,20
	TOTAL	120	60	20	*200*		

weighted-average				
	TP	88	40	12
	FP	32	20	8

precision = TP / (TP + FP)

precision	0,73	0,67	0,60

weighted-average precision (p)

0,69

recall = TP / (TP + FN)

recall	0,88	0,67	0,30

weighted-average recall (r)

0,70

F1-score = 2*p*r / (p + r)

F1-score	0,80	0,67	0,40

weighted-average F1-score

0,68

… the details,

```
weight(A) = 100/200 = 0.50
weight(B) =  60/200 = 0.30
weight(C) =  40/200 = 0.20
```

… the details,

```
precision(A) = 88/(88+32) = 0.73
precision(B) = 40/(40+20) = 0.67
precision(C) = 12/(12+8)  = 0.60
weighted-precision =
0.50*0.73 + 0.30*0.67 + 0.20*0.60
= 0.69


recall(A) = 88/(88+12) = 0.88
recall(B) = 40/(40+20) = 0.67
recall(C) = 12/(12+28) = 0.30
weighted-recall =
0.50*0.88 + 0.30*0.67 + 0.20*0.30
= 0.70
```

Paulo Trigo Silva

The “average” – macro & micro & weighted

- If the class values have **about the same** number of instances, then
  - macro, micro, weighted will also be **about the same**
- If some classes have much more instances than others, and you want to **weight the metric toward the smallest ones**, then
  - **macro** average is more appropriate
- If some classes have much more instances than others, and you want to **weight the metric toward the largest ones**, then
  - **micro** average is more appropriate
- If **macro-average is much lower than micro-average**, then
  - **examine the smaller classes** whenever you have “**poor**” score results
  - … or, the **larger classes** if **micro is much lower than macro**
- If the score must be sensitive to distribution of class instances, then
  - **weighted** average is more appropriate

The **confusion matrix** and the “**random** classifier”

		Predicted Class A	Predicted Class B	Predicted Class C	TOTAL
Actual Class	A	88	10	2	100
	B	14	40	6	60
	C	18	10	12	40
	TOTAL	120	60	20	*200*

accuracy =
(88 + 40 + 12) / 200 =
140 / 200 =
70%

- … but the 70% are a fair measure of the overall success?
  - how many agreements would you expect by “chance” (random classifier)
  - this classifier predicts [A: 120 (60%), B: 60 (30%), C: 20 (10%)]
- ... what if you had a random classifier that predicted the same total numbers of the three classes, i.e., [A: 60%, B: 30%, C: 10%]?
  - ... it classifies the 100 As (line 1) keeping the proportion [60%, 30%, 10%]
  - ... and the same for the 60 Bs (line 2) and for the 40 Cs (line 3)

What would be the confusion matrix of such a random classifier?

Paulo Trigo Silva

The confusion matrix and the “Kappa statistics”

		Predicted Class			
		A	B	C	TOTAL
Actual Class	A	88	10	2	100
	B	14	40	6	60
	C	18	10	12	40
	TOTAL	120	60	20	*200*
		60%	***30%***	***10%***	

Random Classifier

	Predicted Class			
	A	B	C	TOTAL
A	**60**	**30**	**10**	100
B	**36**	**18**	**6**	60
C	**24**	**12**	**4**	40
%	120	60	20	

number of true positives from random classifier = 60 + 18 + 4 = **82**

88 + 40 + 12

number of extra successes (against the random classifier) = 140 – 82 = **58**

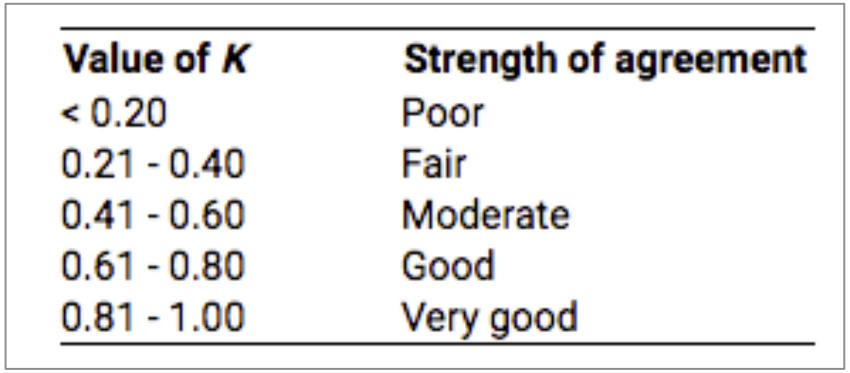
maximum extra successes (against the random classifier) = 200 – 82 = **118**

the value of the ***Kappa statistic*** *is* = 58 / 118 = **49.2%**
i.e., proportion of the total for a perfect predictor (without randomness)

... the Kappa statistics

- The Kappa statistics is used to measure the “level of agreement”
  - between predicted and observed categorizations of a dataset
  - … while “correcting” for agreement that occurs by chance (by random)
- The maximum Kappa value (100%) occurs when the classifier
  - agrees in all predictions with the class values of the testing dataset
  - i.e., only the main diagonal of the confusion matrix is different from zero
  - ... all the remaining cells (off-diagonal) are zero
  - and the predictions that result from chance don’t change this agreement
- The Kappa value of 0% occurs when the classifier
  - fails all the predictions, i.e., the main diagonal is all zeros
  - or, all the predictions result from chance
  - ... i.e., main diagonal sums the same as the one from random classifier

... interpretation of Kappa statistics – and, an example



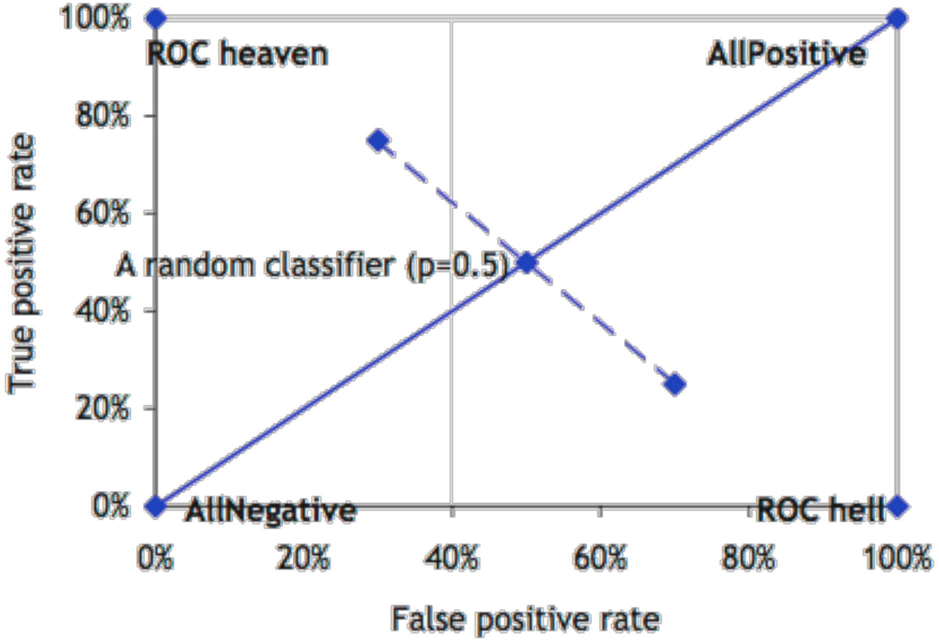
Value of *K*	Strength of agreement
< 0.20	Poor
0.21 - 0.40	Fair
0.41 - 0.60	Moderate
0.61 - 0.80	Good
0.81 - 1.00	Very good

```
train_test_split: stratified_fold_split
classifier: DecisionTreeClassifier


_____________
score_method: accuracy_score
::all-evaluated-datasets::
 50.00% | 100.00% | 50.00% |
66.67% (+/- 23.57%)

_____________
score_method: precision_score
::all-evaluated-datasets::
 25.00% | 100.00% | 25.00% |
50.00% (+/- 35.36%)


_____________
score_method: cohen_kappa_score
::all-evaluated-datasets::
 0.00% | 100.00% | 0.00% |
33.33% (+/- 47.14%)
```

The operating characteristics of a model – ROC analysis

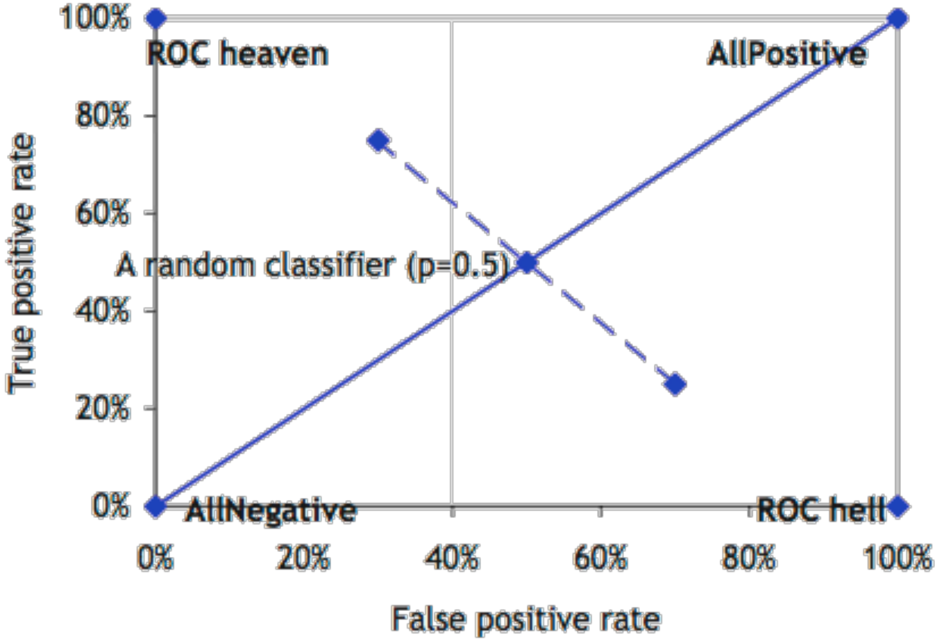
- The operating characteristics describe, for instance,
  - whether a model is more accurate on the positives or on the negatives
- Once the characteristics of the deployment context are known
  - i.e., class distribution and misclassification costs for each class value
  - the user decides which model is is optimal for that deployment context
- Support for such decision making is provided by ROC analysis
  - where ROC stands for “receiver operating characteristics”
- The most common ROC space is a 2-dimensional coordinate system
  - with false positive rate on the X-axis, and
  - with true positive rate on the Y-axis

… ROC space (a global view)



- ROC analysis does not evaluate the learning algorithms
  - … but the classifiers that they produce
- Any diagonal point represents equal true and false positive values
  - … such behavior can be achieved by a random classifier
  - a classifier is no good if it is not above the positive diagonal

… ROC analysis – inverting predictions



- A classifier below the diagonal can easily be transformed into
  - … one above the diagonal by inverting all the predictions
  - e.g., the (70%, 25%) point can be transformed into the (30%, 75%)
- Inverting predictions corresponds to
  - point-mirroring the original point through (50%, 50%) in ROC space

Paulo Trigo Silva

… ROC analysis – inverting predictions (point-mirroring)

to invert prediction point-mirror original point through (50%, 50%) in ROC space

given two points p0 and p1 we have:

$$(x - x1) / (x1 - x0) = (y - y0) / (y1 - y0)$$

and, from there, the equation line of the line p0-p1:

$$\mathbf{y = m (x - x0) + y0}$$

where, the slope, $\mathbf{m = (y1 - y0) / (x1 - x0)}$

given a point p0 all points (x,y) at a distance k from p0 are given by:

$$\mathrm{sqrt}((x - x0)^2 + (y - y0)^2) = k$$

and, from there, we have:

$$\mathbf{y = y0 \pm \mathrm{sqrt}(k^2 - (x - x0)^2)}$$

now, get the point at distance k from p0 that goes through line p0-p1:

$$m (x - x0) + y0 = y0 \pm \mathrm{sqrt}(k^2 - (x - x0)^2)$$

Paulo Trigo Silva

… ROC analysis – point-mirroring example

now, get the point at distance k from p0 that goes through line p0-p1:

$$m (x - x0) + y0 = y0 \pm \mathrm{sqrt}(k^2 - (x - x0)^2)$$

which gives:

$$m^2 (x - x0)^2 = k^2 - (x - x0)^2$$

$$(m^2 + 1) (x - x0)^2 = k^2$$

$$\mathbf{x = x0 \pm (k / \mathrm{sqrt}(m^2 + 1))}$$

p0	x0	0.50
	y0	0.50

p0 is the (50%, 50% point in ROC space

p1	x1	0.70
	y1	0.25

p1 is a point below diagonal line in ROC

m	-1.25
k	0.32

m = slope of line p0-p1
k = distance between p0 and p1

p2	+	x2	0.70
		y2	0.25
	-	x2	0.30
		y2	0.75

$x = x0 \mathbf{+} (k / \mathrm{sqrt}(m^2 + 1))$
$y = m (x - x0) + y0$

$x = x0 \mathbf{-} (k / \mathrm{sqrt}(m^2 + 1))$
$y = m (x - x0) + y0$

first calculate the x-coordinate of the line the goes through p0 and p1 and has length k, and then calculate the y-coordinate of that point

Paulo Trigo Silva

Multiclass ROC analysis

- Much of ROC analysis literature maintains the two-class assumption
  - tradeoff errors (FP - false positive) and benefits (TP - true positive)
- With N classes instead of tradeoffs between TP and FP
  - we have tradeoffs between N benefits and N^2 – N errors
  - … only with 3 classes space becomes a 3^2 – 3 = 6-dimensional polytope
- A way to deal with N classes is to produce N different ROC graphs
  - one for each class; which is called the “class reference” formulation
- If *C* is the set of all class values, ROC graph *g* plots the performance
  - using class value $c_g \in C$ as the positive class and all other as negative
  - … although in this method the augment of the prevalence of a class value may influence the performance evaluation, in practice the method works well and provides reasonable flexibility in evaluation