ISEL

MERCM – ADEETC

AMD (MLDM – Machine Learning and Data Mining) – Module of Practice

Paulo Trigo Silva

---

1. Extend the programmatic support of "Orange Data Mining"

The "Orange Data Mining Framework" provides an excellent "visual programming" environment where we can easily (and quickly) explore and combine several techniques. Additionally, the "Orange" also provides an API to programmatically implement data mining solutions. This combination of "visual-and-textual" programming is very powerful because the visual (and quick) approach helps to better understand the problem before "diving" into textual-programming.

But, there are some more important libraries in this field of data mining (with Python); namely the "pandas" and "scikit-learn" libraries. The "pandas" supports dataset manipulation in a way that has some similarities with SQL (Structured Query Language); for details on such similarities take a look at: "`https://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html`". The "scikit-learn" provides implementations of many data mining algorithms and techniques and is also used by the "Orange".

So, "scikit-learn" is already available in the "Orange virtual-environment".

a) Now we want to extend "Orange virtual-environment" with the "pandas" library. So, follow the instructions in "`_virtualEnv_extend-Orange.txt`".

b) Then, within "`IDLE`", check your settings:

```
import pandas, numpy, sklearn
print('pandas: {}'.format(pandas.__version__))
print('numpy: {}'.format(numpy.__version__))
print('sklearn: {}'.format(sklearn.__version__))
```

"

2. Load a dataset

Consider the code in "`my_split_and_eval.py`".

a) Open the file (e.g., using IDLE) and execute the code (as it is).

b) Look for the `<your-code-here>` tag near the `func_datasetLoader` variable and change its value so that you get the "`pima-indians-diabetes.data.csv`" loaded.

c) Explore the "`load_dataset`" function.

d) Change (the code) in order to load (again) the dataset using the "`simple_dataset`" function.

3. Apply "train-test" split methods

a) Look for the `<your-code-here>` tag and apply the "**holdout**" method to "`simple_dataset`".

b) Change the code to apply "**holdout**" method but now using 50% for training and 50% for testing.

c) Comment the "holdout" and apply the "**stratified_holdout**". Also analyze the 50/50 split and its "impact" on the "stratified" perspective.

d) Repeat the previous analyzes now using the "**repeated_stratified_holdout**" technique.

Paulo Trigo Silva

4. Complete code for "train-test" split methods and analyze results

a) Look for the `<your-code-here>` tag and implement all the remaining train-test split methods (from the "**fold_split**" to the "**leave_p_out**").

b) Analyze all your implemented train-split methods using the "`simple_dataset`".

c) Analyze all your implemented train-split methods using the "`pima-indians-diabetes`" dataset.

5. Analyze and complete the "bootstrap" implementation

Consider the code in "`my_split_bootstrap.py`".

a) Open the file (e.g., using IDLE) and execute the code (as it is). Notice that the train and test datasets are exactly the same! This is not correct!

b) Look for the `<your-code-here>` tag (in "`split`" method) and correct the problem! Test you code and be sure to get (with seed variable set to 5):

```
tt_split_indexes | once
 [([3, 5, 0, 1, 0, 4], [2])]


tt_split_indexes | repeated
 [([3, 5, 0, 1, 0, 4], [2]),
  ([2, 1, 3, 4, 2, 5], [0]), ([4, 1, 3, 3, 4, 1], [0, 2, 5])]
```

c) Look for the `<your-code-here>` tag and try a different number of train-test split repetitions.

d) Now, consider the code in "`my_split_and_eval.py`.

e) Look for the `<your-code-here>` tag and complete both bootstrap train-split implementations.

6. Build a classifier and use a train-test split method

Consider the code in "`my_split_and_eval.py`".

a) Comment all the train-test split method in the "`list_func_tt_split`".

b) Look for the `<your-code-here>` tag and apply the "`DecisionTreeClassifier`". Execute as is and notice that "nothing happened"!

c) Apply the "`holdout`" split (using "`DecisionTreeClassifier`"). Execute and notice that we now need to apply the "`score_recipe`" using some score method(s)!

7. Build classifier, use train-test split method and get overall score

Consider the code in "`my_split_and_eval.py`".

d) Look for the <your-code-here> tag and complete the "`score_recipe`" function.

e) Test you code; be sure to get using "`(repeated_holdout, (1.0/3.0, 2, seed))`" and "`DecisionTreeClassifier`" (with seed variable set to 5):

```
score_method: accuracy_score
::all-evaluated-datasets::
 50.00% | 0.00% |
25.00% (+/- 25.00%)

score_method: precision_score
::all-evaluated-datasets::
 100.00% | 0.00% |
50.00% (+/- 50.00%)

score_method: recall_score
::all-evaluated-datasets::
 50.00% | 0.00% |
25.00% (+/- 25.00%)


score_method: f1_score
::all-evaluated-datasets::
 66.67% | 0.00% |
33.33% (+/- 33.33%)


score_method: cohen_kappa_score
::all-evaluated-datasets::
 0.00% | 0.00% |
0.00% (+/- 0.00%))
```

f) Now, for the "Warnings" that you may already have noticed! Read them carefully and identify the "`no` **`true`** `samples`" and "`no` **`predicted`** `samples`" text.

g) Explain the difference between "`no true samples`" and "`no predicted samples`".

h) Explain why "`no true samples`" warning is thrown by "`recall`" and "`f1_score`".

i) Explain why "`no predicted samples`" warning is thrown by "`precision`" and "`f1_score` ".

j) Look for the <your-code-here> tag and use the "`ignore`" value to turn-off the presentation of warnings under the category "`UndefinedMetricWarning`". Be aware that turning-off these warnings you are loosing the information. Do this only if you really know the corresponding impact!