Documentation for the Scikit-Splearn toolbox

Spectral learning compatible with scikit-learn

Dominique Benielli & Rémi Eyraud

Labex Archimède

November 26, 2016

Contents

1 Introduction

The goal of this toolbox is to implement a python version of the spectral learning algorithm for weighted automata, compatible with the well-known toolbox for statistical machine learning called Scikit-learn[1]. In particular, this allows the use of the tuning functions of Scikit-learn, such as the cross-validation ones and the grid search. Giving the large public using Scikit-learn, we hope that will convince statistical machine learner to get interested into spectral learning.

For a general introduction about the spectral learning of weighted automata, we refer the Reader to this tutorial.

[1]http://scikit-learn.org/

A python toolbox called Sp2Learning for spectral learning of weighted automata is already in production: it corresponds to algorithms developed in the context of the Sequence PredIction Challenge (SPiCe)[2]. However, no compatibility with Scikit-learn is allowed in Sp2Learn: Scikit-SpLearn is the adaptation to the scikit-learn requirements of Sp2Learn (note that Sp2Learn is not maintained anymore).

This toolbox thus provides an implementation of an estimator with 3 main methods: fit, predict, and loss. The data format has also been a important source of modifications. The rest of the toolbox, mainly the implementation of weighted automata and useful methods that come with, is the same than in Sp2Learn.

1.1 Authors

This project has been developed by :

- Denis Arrivault (LabEx Archimède, Aix-Marseille University)
- Dominique Benielli (LabEx Archimède, Aix-Marseille University)
- François Denis (QARMA team, LIF, Aix-Marseille University)
- Rémi Eyraud (QARMA team, LIF, Aix-Marseille University)

2 Installation

2.1 Introduction

The original scikit-splearn Toolbox is developed in Python at *LabEx Archimède* (http://labex-archimede.univ-amu.fr), as a *Laboratoire d'Informatique Fondamentale (LIF)* (http://www.lif.univ-mrs.fr) project at the Aix-Marseille University.

This package, as well as the Sp2Learn toolbox, is a free and open source software, released under the Free BSD License.

The latest version of scikit-splearn can be downloaded from the following PyPI page https://pypi.python.org/pypi/scikit-splearn.

The technical documentation is available at a pythonhosted site: http://pythonhosted.org/scikit-splearn/

The development is done in this gitlab project https://gitlab.lif.univ-mrs.fr/dev/scikit-splearn which provides the git repository managing the source code and where issues can be reported. This is not publicly available for now (it will soon!), but you can contact any of the authors if you want to join the project.

[2]http://spice.lif.univ-mrs.fr/

2.2 Installation

The package can be installed directly by:

```
pip install scikit-splearn
```

or after downloading the sources by:

```
pip install -e .
```

2.3 Package splearn

- splearn
  - splearn/datasets
    - splearn/datasets/base.py
    - splearn/datasets/data_sample.py
  - gilearn/automaton.py
  - gilearn/hankel.py
  - gilearn/learning.py

2.4 Unit and coverage tests

```
(env3_scikit)dominique@ARCHIMEDE:~/projets/scikit-splearn$ nosetests tests
...............................................
Name                                Stmts   Miss  Cover   Missing
-----------------------------------------------------------------
splearn.py                              5      0   100%
splearn/automaton.py                  289      8    97%   136, 142, 147-149, 154, 247, 597
splearn/datasets.py                     2      0   100%
splearn/datasets/base.py               53      5    91%   85, 106, 154-156
splearn/datasets/data_sample.py        52      1    98%   299
splearn/hankel.py                     108     10    91%   81, 88, 95, 176-181, 194-195, 200-201
splearn/spectral.py                   328     33    90%   189, 213-214, 219-220, 259-273, 276-284
                                                          332, 340, 343-344, 352, 356-357, 359, 361,
                                                          363, 381-383, 427, 430, 453, 458-461, 627
-----------------------------------------------------------------
TOTAL                                 837     57    93%
----------------------------------------------------------------------
Ran 49 tests in 118.206s

OK
```

3 Data format and load

The main difficulty for the adaptation to the scikit-learn requirements is related to the input data format.

3.1 File format

The learning algorithms for finite state machines such as Probabilistic Automata (PA), Hidden Markov Models (HMM), and Weighted Automata (WA) usually need as input sequences of different length. The toolbox focuses on a widely used format for data file, used for instance during the SPiCe and PAutomaC competitions:

```
20000 4
7 3 0 3 1 3 1 3
2 3 3
5 3 2 0 3 0
4 3 0 1 0
7 3 3 0 0 1 3 0
2 3 1
4 3 0 3 3
5 3 0 3 1 3
23 3 1 3 0 2 0 3 1 2 0 3 0 2 0 1 0 3 0 1 0 3 3 3
....
```

where the first line contains first the number of samples, and then the number of different symbols (also called letters of the alphabet), each of the others lines contains a single sample, with in first position the length of the sequence (to allow the empty sequence of 0 symbol). This format is not compatible with the 2D array input format expected by scikit-learn.

3.2 Data format

The loaded files are formatted and reshaped to be included in a 2D array, shape (n_samples, n_features) where n_features is the length of the longest sequence in the sample. If a sequence is smaller than the longest, the useless cells contain -1, and so the data follows the following format:

```
3  0  3  1  3  1  3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3  3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3  2  0  3  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3  0  1  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3  3  0  0  1  3  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3  0  3  3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3  0  3  1  3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3  1  3  0  2  0  3  1  2  0  3  0  2  0  1  0  3  0  1  0  3  3  3
....
```

During the learning process (method `fit`) the array is analyzed by the algorithm and transformed into dictionaries, a more useful storage format.

3.3 Loading a data set

Scikit-Splearn contains a module that inherits form scikit-learn datasets.base that creates the needed array from a file in SPiCe/PAutomaC format.

```
>>> from splearn.datasets.base import load_data_sample
>>> train_file = '3.pautomac.train' #path to the training file
>>> data = load_data_sample(train_file)
# or simply: data = load_data_sample('4.spice.train')
>>> data.nbL
4
>>> data.nbEx
5000
>>> data.data
Splearn_array([[ 3.,  0.,  3., ..., -1., -1., -1.],
       [ 3.,  3., -1., ..., -1., -1., -1.],
       [ 3.,  2.,  0., ..., -1., -1., -1.],
       ...,
       [ 3.,  1.,  3., ..., -1., -1., -1.],
       [ 3.,  0.,  3., ..., -1., -1., -1.],
       [ 3.,  3.,  1., ..., -1., -1., -1.]])
>>>
```

where >>> denotes the Python interpreter prompt. The data structure is a dictionary-like object that contains all the data and some metadata, with different fields such as nbL the number of letters, nbEx the number of sequences, and the sequences themselves in a plain 2D array of type Splearn_array.

3.4 Class Splearn_array data format

The loaded data are formatted in the field `data` as a Splearn_array: this format inherits from numpy ndarray (a scikit-learn requirement), and contains as a main element a 2D array [n_samples, n_features] as described in Section 3.2.

Splearn_array also encapsulates other variables: numbers nbL (numbers of letters) and nbEx (numbers of samples) defined above, and 4 dictionaries named sample, pref, suff and fact and corresponding respectively to sequences, prefixes, suffixes, and factors inside the data set. These useful dictionaries are populated only by the `fit` method (scikit-learn requirement).

4 The estimator class: Spectral

4.1 Spectral init and parameters setting

The estimator model is named Spectral. It can be found in the splearn package and it works as scikit estimator. The spectral estimator accepts as input different parameters:

- *partial* boolean
  - *False*: the computation of the Hankel matrix is performed with all possible elements from the learning sample
  - *True*: the computation is performed with a given limited length for elements or with given sets of elements (see parameter *lrows, lcolumns*)
- *lrows* number or list of rows (int or tuple). A list of strings or an integer indicating the max length of elements to consider if partial=True. If not instantiated, based on all prefixes if version='classic' or 'prefix', all factors otherwise
- *lcolumns* number or list of columns (int or tuple) a list of strings or an interger indicating the max length of elements to consider if partial=True If not instantiated, based on all suffixes if version='classic' or 'suffix', all factors otherwise
- *sparse* (boolean) *True* for sparse computation with a sparse Hankel matrix, *False* for a non-sparse matrix
- *smooth_ method* (string, default value = 'none') indicate the method of smoothing
  - *'trigram'*: the 3-Gram trigram dictionary is computed and used by the predict function, in this case the trigram probability is used instead of Spectral probability when a negative weight is given by the learned WA
  - *'none'* or anything else: no smooth method is used in `predict` function.
- *rank* (int): the value for the rank factorization of the Hankel matrix
- *version* (string): (default value = "classic") variant of the Hankel matrix to use, version value can be 'classic', 'prefix', 'suffix', or 'factor'
- *mode_ quiet* (boolean): (default value = False) if True no output information is printed while running the `fit` method

Here is a use case:

```
>>> from splearn.spectral import Spectral
>>> sp = Spectral(rank=5, lcolumns=6, lrows=6)
>>> sp.get_params()
{'lcolumns': 6, 'partial': True, 'rank': 5,
'sparse': True, 'version': 'classic', 'mode_quiet': False,
'smooth_method': 'none', 'lrows': 6}
>>> sp.set_params(smooth_method='trigram')
Spectral(lcolumns=6, lrows=6, partial=True, rank=5, sparse=True,
   smooth_method='trigram', mode_quiet=False, version='classic')
```

4.2 Spectral: the Fit method

```
>>> sp.fit(data.data)
Start Hankel matrix computation
End of Hankel matrix computation
Start Building Automaton from Hankel matrix
End of Automaton computation
Spectral(lcolumns=6, lrows=6, partial=True, rank=5,
    smooth_method='none', sparse=True, version='classic')
>>> sp.set_params(mode_quiet=True)
Spectral(lcolumns=6, lrows=6, mode_quiet=True, partial=True,
  rank=5, smooth_method='none', sparse=True, version='classic')
>>>
```

The `fit` method computes and instantiates a weighted automaton (see Section 6 for more details about the automaton class). It is accessible by the following commands:

```
>>> sp.Automaton.initial
array([-0.00049249,  0.00304676, -0.04405996, -0.10765322, -0.08660063])
>>> sp.Automaton.final
array([ 0.07498848, -0.02407125, -0.44675369, 0.62799252, -0.55444987])
>>> sp.Automaton.transitions
[array([[ 0.04323057, -0.24063466, 0.35033337, -0.2795733 , -0.21986786],
        [ 0.06896795, -0.30065877, 0.20745919, -0.15087461, -0.56014468],
        [ 0.02885238, -0.13799954, 0.18365036, -0.2099106 , -0.14498752],
        [ 0.00612321, -0.02332236, -0.06608033, 0.10764515, -0.15109704],
        [-0.02015367,  0.08988955, -0.00553278, -0.03131386, 0.24333355]]),
array([[ 0.07994532,  0.09135396, -0.30878986, 0.28080686,  0.20376293],
        [-0.09865201, -0.08073288, 0.2593956 , -0.12114366, -0.11119892],
        [-0.06136638, -0.06247307, 0.12020222,  0.0025085 , -0.15696747],
        [-0.00246864, -0.00898858, -0.0005062 , -0.00855869, -0.05379994],
        [ 0.03049939,  0.03972289, -0.04998491, 0.00356841,  0.14193265]]),
array([[-0.06691411, -0.11421271, 0.37924968, -0.21186808, -0.24443885],
        [ 0.11687672,  0.15004827, -0.13393244, -0.00919187, 0.35004331],
        [ 0.01205358,  0.01938594,  0.04148993, -0.03541727, 0.02317928],
        [ 0.00729786,  0.00555287, -0.02248987, 0.0361802 , -0.03854873],
        [-0.01069779, -0.01068881, -0.00056362, -0.025575 , 0.0498595 ]]),
array([[ 0.07153358, -0.01601894, 0.07362502, -0.10483001, 0.02441764],
        [-0.05174562, -0.08994   ,  0.27684684, -0.23754248, 0.07353445],
        [-0.00746697, -0.04863056, -0.62933143, 0.46853876, 0.09339057],
        [-0.00695291, -0.05607908, -0.36582793, -0.01322074, 0.649387 ],
        [ 0.00240461, -0.02149357, 0.0911477 , -0.38472518,  0.66182296]])]
```

4.3 Spectral: Predict and predict_proba methods

The `predict` method returns the weights given by the learned model to a set of data (SpLearn_array format) as an array of dimension 1: each element is the weight given by the automaton to the sequence of same index in the data structure given as parameter. The method `predict_proba` (scikit-learn requirement) gives the same results as the `predict` method.

```
>>> sp.predict(data.data)
array([  4.38961058e−04,  1.10616861e−01,  1.35569353e−03, ...,
         4.66041996e−06,  4.68177275e−02,  5.24287604e−20])
>>> sp.predict_proba(data.data)
array([  4.38961058e−04,  1.10616861e−01,  1.35569353e−03, ...,
         4.66041996e−06,  4.68177275e−02,  5.24287604e−20])
```

4.3.1 The smooth_method option 'trigram'

If the estimator is set with *smooth_method='trigram'* option, a trigram dictionary is computed: it is accessible as an attribute 'trigram' of the Spectral object. In this case, while predicting a weight, if the automaton returns a non-positive value, the probability given by the trigram is then used. This is not a smoothing method per se: it becomes one only when used together with a normalize scoring function (like the perplexity one implemented in the toolbox, see Section 4.4.1).

The attribute *trigram_index* gives the positions where the trigram dictionary is used instead of the weighted automata prediction. After a prediction or the computation of a score, the `nb_trigram` method gives the numbers of items affected by the smoothing, i.e. the number of times the trigram has been used.

```
>>> sp.set_params(smooth_method='trigram')
Spectral(lcolumns=6, lrows=6, partial=True, rank=5,
         smooth_method='trigram', sparse=True,
         version='classic', mode_quiet=True)
>>> sp.fit(data.data)
Spectral(lcolumns=6, lrows=6, partial=True, rank=5,
         smooth_method='trigram', sparse=True,
         version='classic', mode_quiet=True)
>>> sp.predict(data.data)
array([  4.38961058e−04,   1.10616861e−01,
1.35569353e−03, ...,
         4.66041996e−06,   4.68177275e−02,   5.24287604e−20])
>>> sp.trigram
{(0, 1): {0: 233, 1: 309, 2: 357, 3: 742, −2: 141, −1: 1782},
(1, 2): {0: 366, 1: 309, 2: 126, 3: 291, −2: 120, −1: 1212},
(3, 2): {0: 292, 1: 88, 2: 45, 3: 107, −1: 667, −2: 135},
(0, 0): {0: 177, 1: 258, 2: 80, 3: 225, −1: 984, −2: 244},
(3, 3): {0: 1450, 1: 895, 2: 306, 3: 664, -2: 1000, -1: 4315},
(2, 0): {0: 126, 1: 179, 2: 120, 3: 626, -2: 105, -1: 1156},
(3, 0): {0: 571, 1: 1083, 2: 1065, 3: 2043, -1: 5517, -2: 755},
(3, 1): {0: 359, 1: 457, 2: 567, 3: 1465, -2: 135, -1: 2983},
(2, 2): {0: 148, 1: 21, 2: 22, 3: 73, -1: 292, -2: 28},
(2, 1): {0: 88, 1: 232, 2: 96, 3: 202, -2: 59, -1: 677},
(1, 1): {0: 124, 1: 605, 2: 192, 3: 562, -1: 1603, -2: 120},
(-1, 3): {0: 1901, 1: 834, 2: 123, 3: 1872, -2: 270, -1: 5000},
(1, 3): {0: 735, 1: 339, 2: 136, 3: 1167, -1: 2971, -2: 594},
(2, 3): {0: 313, 1: 202, 2: 56, 3: 82, -1: 917, -2: 264},
(-1, -1): {3: 5000, -1: 5000},
(1, 0): {0: 110, 1: 262, 2: 127, 3: 150, -1: 804, -2: 155},
(0, 3): {0: 1118, 1: 713, 2: 46, 3: 530, -2: 637, -1: 3044},
(0, 2): {0: 350, 1: 259, 2: 99, 3: 446, -2: 238, -1: 1392}}
>>> sp.trigram_index
array([False, False, False, ..., False, False], dtype=bool)
>>> sp.nb_trigram()
220
```

4.4 Spectral: loss and score methods

The `loss` method returns the opposite of the mean (if parameter *normalize* is True) or the sum (if *normalize* is False) of the logarithm of the probability in the case of non-supervised learning (i.e. while parameter y values 'none'), and the least squares in the supervised case.

The `score` method returns the opposite of `loss` except in the case of supervised learning with the scoring option valued to 'perplexity'.

```
>>> sp.loss(data.data)
10.530029936056017
>>> sp.loss(data.data, normalize=False)
52650.149680280083
>>> test_sample = load_data_sample("3.pautomac.test") #get test
>>> target = open("3.pautomac_solution.txt", "r") #target proba
>>> targets.readline() #get rid of first (useless) line
'1000\n'
>>> target_probas = [float(line[:-1]) for line in targets]
#target_probas is a vector of the real probas of each element
of the test sample
>>> est.loss(test_sample.data, y=target_probas)
2.6569772687614514e-05
```

4.4.1 Score and scoring = 'perplexity'

The `score` method is always normalized and returns the inverse of the loss method (scitkit-learn requirement: scores have to work in a 'the greater the better' way). In the case of supervised learning the normalized perplexity[3] is computed if the scoring parameter is set to 'perplexity' (default value). Note that it can only be used if a smoothing method is set.

```
>>> sp.score(data.data)
-10.530029936056017
>>> sp.score(data.data, scoring='perplexity')
-10.530029936056017
>>> sp.score(data.data, scoring='none')
-10.530029936056017
>>> sp.score(test_sample.data, target_probas)
56.1778597742712
>>> sp.score(test_sample.data, target_probas, scoring='perplexity')
56.1778597742712
>>> sp.score(test_sample.data, target_probas, scoring='none')
-2.6569772687614514e-05
```

5 Scikit-learn compatibility

5.1 check_estimator

The compatibility of splearn is validated by the check_estimator.

```
>>> from sklearn.utils.estimator_checks import check_estimator
>>> check_estimator(Spectral)
Process finished with exit code 0
```

This returned code expresses that some errors occurred but this is due to the fact that not all verification are doable (some algorithms at the core of scikit-learn, like the SVM implementation, obtain similar results). When a module is not compliant with the crucial parts of scikit-learn, explicit error messages are obtained.

5.2 cross_validation

```
>>> sp = Spectral()
>>> sp.set_params(partial=True, version='classic', lcolumns= 6,
        lrows= 6, rank=5, smooth_method='trigram', mode_quiet= True)
>>> from sklearn import cross_validation
>>> Xtrain, Xtest = cross_validation.train_test_split(data.data,
test_size=0.4, random_state=0)
```

[3] see PAutomaC papers and website for a definition

```
>>> sp.fit(Xtrain)
>>> scores= cross_validation.cross_val_score(sp,data.data,cv=4)
>>> print(scores)
[-10.61271014  -10.71261867  -10.36931802  -11.01046119]
>>> from sklearn.cross_validation import cross_val_predict
>>> predicted = cross_val_predict(sp, data.data, cv=10)
>>> print(predicted)
[  3.63092273e-04   1.01803674e-01   1.27485779e-03 ...,
   1.21643536e-06   4.06223489e-02   2.93321679e-20]
```

5.3 GridSearch

```
>>> from sklearn.grid_search import GridSearchCV
>>> tuned_parameters = [{'version' : ['classic'],
 'lcolumns': [5, 6, 7], 'lrows': [5, 6, 7]},
 {'version' :['factor'], 'lcolumns': [4, 5, 6],
 'lrows': [4, 5, 6]}]
>>> clf = GridSearchCV(sp, tuned_parameters, cv=5)
>>> clf.fit(X_train)
>>> print("Best␣parameters␣found␣on␣development␣sample:\n",
          clf.best_params_)
Best parameters found on development sample:
 {'version': 'factor', 'lcolumns': 5, 'lrows': 4}
>>> print("Grid␣scores␣on␣development␣set:")
Grid scores on development set:
>>> for params, mean_score, scores in clf.grid_scores_:
        print("%0.4f␣(+/-%0.02f)␣for␣%r"\
        %(mean_score, scores.std() * 2, params))
-11.0912 (+/-0.26) for {'version':'classic','lcolumns':5,'lrows':5}
-11.0892 (+/-0.27) for {'version':'classic','lcolumns':5,'lrows':6}
-11.0901 (+/-0.27) for {'version':'classic','lcolumns':5,'lrows':7}
-11.0976 (+/-0.25) for {'version':'classic','lcolumns':6,'lrows':5}
-11.0990 (+/-0.26) for {'version':'classic','lcolumns':6,'lrows':6}
-11.0978 (+/-0.27) for {'version':'classic','lcolumns':6,'lrows':7}
-11.1060 (+/-0.24) for {'version':'classic','lcolumns':7,'lrows':5}
-11.1010 (+/-0.27) for {'version':'classic','lcolumns':7,'lrows':6}
-11.1018 (+/-0.27) for {'version':'classic','lcolumns':7,'lrows':7}
-10.3290 (+/-0.22) for {'version':'factor','lcolumns':4,'lrows':4}
-10.3416 (+/-0.22) for {'version':'factor','lcolumns':4,'lrows':5}
-10.3437 (+/-0.22) for {'version':'factor','lcolumns':4,'lrows':6}
-10.3207 (+/-0.23) for {'version':'factor','lcolumns':5,'lrows':4}
-10.3343 (+/-0.22) for {'version':'factor','lcolumns':5,'lrows':5}
-10.3374 (+/-0.23) for {'version':'factor','lcolumns':5,'lrows':6}
-10.3208 (+/-0.23) for {'version':'factor','lcolumns':6,'lrows':4}
```

```
−10.3310 (+/−0.22) for {'version':'factor','lcolumns':6,'lrows':5}
−10.3351 (+/−0.22) for {'version':'factor','lcolumns':6,'lrows':6}
```

6 Automaton class

The models learned by the toolbox are weighted automata whose expressiveness in term of probabilistic distribution is strictly higher that Probabilistic Automaton (PA) and Hidden Markov Model (HMM).

These automata are stored as linear representations: there are defined by a initial vector (available via `self.initial`, a final vector (`self.final`) and an array of transitions matrices, one for each symbol (`self.transition`).

A bunch of methods are defined to work on an automaton, among which:

- `transformation(self, source="classic", target="prefix")`: from an automaton computing a probability distribution in the 'source' format, it returns an automaton computing the distribution defined by the parameter 'target'. For instance, if source='classic' the input automaton is considered to computes weights for whole sequences, and if target='prefix' then the outputted automaton will compute the weight of a sequence a a prefix in the former automaton. Parameters values could be 'classic', 'prefix', 'suffix', 'factor'.
- `BuildHankels(self, lrows=[], lcolumns=[])`: from an automaton, build the corresponding Hankel matrix with the rows and columns specified in corresponding parameters.
- `mirror(self)`: computes and returns the mirror automaton
- `def val(self, word)`: computes the weight of a given sequence given as a string
- `minimisation(self, tau)`: computes an equivalent minimal automaton, to the precision tau. This algorithm is proven to be numerically stable.
- `_calcAbsConv(self)`: test a sufficient condition for the automaton to be absolutely convergent.
- `calc_prefix_completion_weights(self, prefix)`: for each possible letter, computes the weight of the prefix concatenated with the letter. The automaton has to be in prefix form (see methods `transformation`).