Time Series Forecaster Documentation

Release 0.1

Miguel Díaz Lozano

Jul 09, 2018

Contents

TSF is a library that extend Scikit-learn software composed by several time series preprocessing algorithms developed at the University of Cordoba in the Learning and Artificial Neural Networks (AYRNA) research group. This library is able to preprocess any time serie, either univariate or multivariate, and create a inputs matrix for every sample of the time serie(s) so any model of Scikit-learn can be trained.

TSF code is open source and available at the Github repository.

CHAPTER 1

Contents

1.1 Quickstart

1.1.1 Installation

You can get TSF Library directly from PyPI like this:

```
pip install tsf
```

Otherwise, you can clone the project directly from the Github repository using git clone:

```
git clone https://github.com/migueldl96/TSF-library.git
cd TSF-library
[sudo] python setup.py install
```

1.1.2 First transformation: Look *n_prev* backward!

The simplest preprocess algorithm is just to take some previous samples for every element of time serie. In TSF this is call SimpleAR transformation and works with a fixed window size (we call window to the previous samples used for forecast a sample of the serie). Let's jump right in with a synthetic time serie!

```
from tsf.windows import SimpleAR

time_serie = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

ar = SimpleAR(n_prev=5)
X = ar.transform(X=[], y=time_serie)

print X
```

The code below import SimpleAR class from windows module, create an instance of it and call transform method. This method will return the input matrix resulting from the transformation

Running this code we'll obtain:

```
> python simplear.py
[[1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]
 [4 5 6 7 8]
 [5 6 7 8 9]]
```

That's out inputs matrix! Now we only need some the corresponding outputs to train any model. Luckily, all the TSF transformers have a `offset_y` method that returns our precious outputs. Let's do some modifications to our previous script:

```
from tsf.windows import SimpleAR

time_serie = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

ar = SimpleAR(n_prev=5)
X = ar.transform(X=[], y=time_serie)
y = ar.offset_y(X, time_serie)

print X
print y
```

If we run it now:

```
> python simplear.py
[[1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]
 [4 5 6 7 8]
 [5 6 7 8 9]]
[ 6  7  8  9 10]
```

That's it! We have our inputs matrix (X) and the output for every pattern (y).

Note: From a time serie of 10 samples we have obtain 5 patterns. This is because we set `n_prev` to 5: the algorithm needs to take the first 5 samples to build the first pattern. All the autorregresive models always need to build patterns from previous samples, and always will return less patterns than the time serie length depending on the transformers parameters.

1.2 Complement a database

TSF algorithms not only serves to create inputs matrixs from a time serie; they can also complement input data to obtain better performance when forescasting. In previous chapter we called `transform` method passing an empty array to X parameter. Let's suppose we have not only our time serie but also some features (e.g. some climatological indices) that we want to keep and complement with autorregresive information from our serie:

```
from tsf.windows import SimpleAR

features = [[-10, -20, -30, -40, -50, -60, -70, -80, -90],
            [-11, -21, -31, -41, -51, -61, -71, -81, -91],
            [-12, -22, -32, -42, -52, -62, -72, -82, -92],
```

(continues on next page)

(continued from previous page)

```
              [-13, -23, -33, -43, -53, -63, -73, -83, -93],
              [-14, -24, -34, -44, -54, -64, -74, -84, -94],
              [-15, -25, -35, -45, -55, -65, -75, -85, -95],
              [-16, -26, -36, -46, -56, -66, -76, -86, -96],
              [-17, -27, -37, -47, -57, -67, -77, -87, -97],
              [-18, -28, -38, -48, -58, -68, -78, -88, -98],
              [-19, -29, -39, -49, -59, -69, -79, -89, -99]]
time_serie = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

ar = SimpleAR(n_prev=5)
X = ar.transform(X=features, y=time_serie)
y = ar.offset_y(X, time_serie)

print X
print y
```

In the example below, the negative integers simulate some extra features that we want to keep in our model. If we call transform method passing this `features` matrix to X argument, the SimpleAR information will be appended to the `features` matrix:

```
> python complementing.py
[[-15 -25 -35 -45 -55 -65 -75 -85 -95   0   1   2   3   4]
 [-16 -26 -36 -46 -56 -66 -76 -86 -96   1   2   3   4   5]
 [-17 -27 -37 -47 -57 -67 -77 -87 -97   2   3   4   5   6]
 [-18 -28 -38 -48 -58 -68 -78 -88 -98   3   4   5   6   7]
 [-19 -29 -39 -49 -59 -69 -79 -89 -99   4   5   6   7   8]]
[5 6 7 8 9]
```

1.3 Multivariate problems

Sometimes it is useful to combine several strongly linked time series to obtain better results when forecasting. For example, SimpleAR transformer can be applied to a rains time serie to predict when you will have to take the umbrella, but you would obtain better performance on your predictions if you could combine this information with temperature as they are strongly linked climatic conditions.

This kind of problems are call **multivariate time series** and are very present in real world problems in which we have one target serie that we call *endogenous* and others called *exogenous* relevant for our problem. TSF library is prepared to deal with these kind of problems.

1.3.1 Dealing with several time series

The `y` parameter on `transform` methods should receive the time serie. However, it can be a 1 dimension array (*vector*) representing one time serie or a 2 dimensions array (*matrix*) in which every row will represent a single time serie. In this case, the endogenous serie will be always the first row of the matrix, and the rest the exogenous ones.

When working with several time series, the window transformers will be applied to all of them. This time we'll use another preprocessing algorithm included in TSF library: **DinamicWindow**.

```
from tsf.windows import DinamicWindow

time_series =   [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],                # Endogenous serie
                 [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],       # Exogenous serie
↪#1
```

(continues on next page)

(continued from previous page)

```
                  [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]]          # Exogenous serie
↪#2

dw = DinamicWindow(stat='variance', ratio=0.1, metrics=['variance', 'mean'])
X = dw.transform(X=[], y=time_series)
y = dw.offset_y(X, time_series)

print X
print y
```

The length for every window in DinamicWindow algorithm is determined by a limit depending on `stat` and `ratio` parameters. In this case, the windows will grow while the window variance is less than 10% global variance. Once the limit is reached, the window samples will be summarize in *variance* and *mean* (`metrics` parameter).

Running this block of code, we'll get this output:

```
> python multivariate.py
[[ 0.          0.          0.          10.          0.          20.         ]
 [ 0.25        0.5         0.25        10.5         0.25        20.5        ]
 [ 0.66666667  1.          0.66666667 11.          0.66666667 21.         ]
 [ 0.66666667  2.          0.66666667 12.          0.66666667 22.         ]
 [ 0.66666667  3.          0.66666667 13.          0.66666667 23.         ]
 [ 0.66666667  4.          0.66666667 14.          0.66666667 24.         ]
 [ 0.66666667  5.          0.66666667 15.          0.66666667 25.         ]
 [ 0.66666667  6.          0.66666667 16.          0.66666667 26.         ]
 [ 0.66666667  7.          0.66666667 17.          0.66666667 27.         ]]
[1 2 3 4 5 6 7 8 9]
```

As you can see, transforming these time series using DinamicWindow algorithm returns an input matrix with 6 features. By default, all the time series are involved in the transformation, so the first two columns correspond to *variance* and *mean* for the first serie, the next two columns for the second serie and the last two columns for the thirst one. The outputs are the elements of our *endogenous* serie as it is our target.

1.3.2 Skip some time series

Maybe you don't want an algorithm to be applied in some series. It is useful when concatenating several transformers with pipelines and working with ordinal time series where DinamicWindow doesn't make much sense. To avoid applying an algorithm to a serie, every transformer in TSF library has a `indexs` parameter that allows you to indicate which series you want to include in the preprocessing task. By default, this parameters is `None`, meaning that all series are considered. Let's do a little modification to our previous script:

```
from tsf.windows import DinamicWindow

time_series =   [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],                     # Endogenous serie
                  [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],          # Exogenous serie
↪#1
                  [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]]          # Exogenous serie
↪#2

dw = DinamicWindow(stat='variance', ratio=0.1, metrics=['variance', 'mean'],␣
↪indexs=[0, 2])
X = dw.transform(X=[], y=time_series)
y = dw.offset_y(X, time_series)
```

(continues on next page)

(continued from previous page)

```
print X
print y
```

The code below suppose that we are not interested in getting DinamicWindow information for the second serie. Running it, we'll get the following output:

```
> python multivariate.py
[[ 0.          0.          0.         20.        ]
 [ 0.25        0.5         0.25       20.5       ]
 [ 0.66666667  1.          0.66666667 21.        ]
 [ 0.66666667  2.          0.66666667 22.        ]
 [ 0.66666667  3.          0.66666667 23.        ]
 [ 0.66666667  4.          0.66666667 24.        ]
 [ 0.66666667  5.          0.66666667 25.        ]
 [ 0.66666667  6.          0.66666667 26.        ]
 [ 0.66666667  7.          0.66666667 27.        ]]
[1 2 3 4 5 6 7 8 9]
```

We have ignored the second time serie in this example, so the third and forth column have disappeared.

Note: `indexs` parameters should receive an array of ints indicating the indices of the rows from the time series matrix. If an index is out of bounds, you will get a `UserWarning`, but program will continue its execution.

1.4 TSFPipeline

Pipelines are a great tool included in Scikit-learn that allows to concatenate several transformers and estimators in a single object. TSF library include its own Pipeline class that extend the original and allows concatenating several TSF transformers.

1.4.1 Creating a TSFPipeline

Concatenating algorithms is a great idea when preprocessing time series as it allows you to get a database from autorregresive techniques. `TSFPipeline` works in the same way as original Pipeline does and it is compatible with all Scikit-learn environment.

```
from tsf.windows import SimpleAR, DinamicWindow
from tsf.pipeline import TSFPipeline

time_series =   [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],                    # Endogenous serie
                 [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]]          # Exogenous serie
↪#1

pipe = TSFPipeline([('ar', SimpleAR(n_prev=2)),
                    ('dw', DinamicWindow(stat='variance', ratio=0.1, metrics=[
↪'variance', 'mean'])),
                    ('', None)])

X, y = pipe.transform(X=[], y=time_series)

print X
print y
```

In the example below, TSFPipeline is applied to make the transformations without a final estimator. The algorithms are applied sequentially and results appended to the array passed to `X` parameter:

```
> python pipeline.py
[[ 0.          1.          10.         11.          0.25        0.5
   0.25       10.5        ]
 [ 1.          2.          11.         12.          0.66666667  1.
   0.66666667 11.         ]
 [ 2.          3.          12.         13.          0.66666667  2.
   0.66666667 12.         ]
 [ 3.          4.          13.         14.          0.66666667  3.
   0.66666667 13.         ]
 [ 4.          5.          14.         15.          0.66666667  4.
   0.66666667 14.         ]
 [ 5.          6.          15.         16.          0.66666667  5.
   0.66666667 15.         ]
 [ 6.          7.          16.         17.          0.66666667  6.
   0.66666667 16.         ]
 [ 7.          8.          17.         18.          0.66666667  7.
   0.66666667 17.         ]]
[2 3 4 5 6 7 8 9]
```

Note: TSFPipeline `transform` method returns `X` and `y`.

TSFPipeline is useful to apply the same transformation to several time series.

1.4.2 Adding a final estimator to TSFPipeline

As genuine Pipeline do, you can append an estimator to the steps of the Pipeline so you can use methods like `predict` and `fit` directly from TSFPipeline object.

```
from tsf.windows import SimpleAR, DinamicWindow
from tsf.pipeline import TSFPipeline
from sklearn.linear_model import LassoCV

train_series =      [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
                 [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]]


pipe = TSFPipeline([('ar', SimpleAR(n_prev=2)),
                    ('dw', DinamicWindow(stat='variance', ratio=0.1, metrics=[
↪'variance', 'mean'])),
                    ('lasso', LassoCV())])

pipe.fit(X=[], y=time_series)
```

1.5 TSFGridSearch

When dealing with several transformers and estimators there is thousands of possible parameter combinations. Built-in GridSearchCV Scikit-learn class helps to optimize these parameters choosing the ones that returns better performance.

TSF Library include a similar mechanism to optimize its transformers parameters (such as `n_prev` in *SimpleAR* or `ratio` in *DinamicWindow*): **TSFGridsearch**. It decorates original GridSearchCV `fit` method and adapt it to TSF

Library needs, therefore the use is identical as the genuine class.

1.5.1 Optimizing hiperparameters

The full potential of TSFGridSearch is when combining it with *TSFPipeline*. You can create a sequential list of step transformations and optimize the parameters. Is this example, we'll use a combination of *SimpleAR* and *DinamicWindow* with a MLPRegressor:

```
from tsf.windows import SimpleAR, DinamicWindow
from tsf.pipeline import TSFPipeline
from tsf.grid_search import TSFGridSearch
from sklearn.neural_network import MLPRegressor

# Random continous time series
time_series =   [[0.2, 0.5, 0.4, 0.32, 0.7, 0.8, 0.91, 0.53, 0.12, -0.26],
                 [1.5, 1.54, 1.2, 1.96, 1.43, 1.32, 1.68, 1.23, 1.85, 1.01]]

# Pipeline
pipe = TSFPipeline([('ar', SimpleAR()),
                    ('dw', DinamicWindow()),
                    ('MLP', MLPRegressor())])

# Params grid
params =    [
                {
                    'ar__n_prev': [1, 2, 3]
                },
                {
                    'dw__ratio': [0.1, 0.2]
                },
                {
                    'hidden_layer_sizes': [80, 90, 100, 110]
                }
            ]

# Grid search
grid = TSFGridSearch(pipe, params)

# Fit and best params
grid.fit(X=[], y=time_series)
print grid.best_params_
```

`best_params_` attribute returns the dictionary with the best parameters combinations. Is this example, this dictionary is:

```
> python gridsearch.py
{'MLP__hidden_layer_sizes': 110, 'ar__n_prev': 2, 'dw__ratio': 0.1}
```

Note: As randomness is not contemplated, best parameters dictionary may differ from the obtained in this example.

1.6 References

Energy Flux Range Classification by Using a Dynamic Window Autoregressive Model