TSML: Time Series Machine Learning Toolbox

Paulito P. Palmes, Joern Ploennigs, Niall Brady

May 29, 2019

Contents

Part I

HOME

Chapter 1

TSML (Time-Series Machine Learning)

TSML (Time Series Machine Learning) is a package for Time Series data processing, classification, and prediction. It combines ML libraries from Python's ScikitLearn, R's Caret, and Julia ML using a common API and allows seamless ensembling and integration of heterogenous ML libraries to create complex models for robust time-series pre-processing and prediction/classification.

1.1 Motivations

Over the past years, the industrial sector has seen many innovations brought about by automation. Inherent in this automation is the installation of sensor networks for status monitoring and data collection. One of the major challenges in these data-rich environments is how to extract and exploit information from these large volume of data to detect anomalies, discover patterns to reduce downtimes and manufacturing errors, reduce energy usage, etc.

To address these issues, we developed TSML package. It leverages AI and ML libraries from ScikitLearn, Caret, and Julia as building blocks in processing huge amount of industrial time series data. It has the following characteristics described below.

1.2 Package Features

- TS data type clustering/classification for automatic data discovery
- TS aggregation based on date/time interval
- TS imputation based on Nearest Neighbors
- TS statistical metrics for data quality assessment
- TS ML wrapper more than 100+ libraries from caret, scikitlearn, and julia
- TS date/value matrix conversion of 1-D TS using sliding windows for ML input
- Common API wrappers for ML libs from JuliaML, PyCall, and RCall
- Pipeline API allows high-level description of the processing workflow
- Specific cleaning/normalization workflow based on data type
- Automatic selection of optimised ML model
- Automatic segmentation of time-series data into matrix form for ML training and prediction
- Easily extensible architecture by using just two main interfaces: fit and transform

- Meta-ensembles for robust prediction
- Support for distributed computation for scalability and speed

1.3 Installation

TSML is in the Julia Official package registry. The latest release can be installed at the Julia prompt using Julia's package management which is triggered by pressing] at the julia prompt:

```
julia> ]
(v1.0) pkg> add TSML
```

or

```
julia> using Pkg
julia> pkg"add TSML"
```

or

```
julia> using Pkg
julia> Pkg.add("TSML")
```

or

```
julia> pkg"add TSML"
```

Once TSML is installed, you can load the TSML package by:

```
julia> using TSML
```

or

```
julia> import TSML
```

Generally, you will need the different transformers and utils in TSML for time-series processing. To use them, it is standard in TSML code to have the following declared at the topmost part of your application:

```
using TSML
using TSML.TSMLTransformers
using TSML.TSMLTypes
using TSML.Utils
```

1.4 Tutorial Outline

- Aggregators and Imputers
  - DateValgator
  - DateValNNer
  - DateValizer
- Pipeline
  - Workflow of Pipeline
  - Extending TSML
- Statistical Metrics
  - Statifier for Both Non-Missing and Missing Values
  - Statifier for Non-Missing Values only
  - Statifier After Imputation
- TS Data Discovery
  - TSClassifier

1.5 Manual Outline

- Value Preprocessing
  - Matrifier
- Date Preprocessing
  - Dateifier
  - ML Features: Matrifier and Datefier
- Aggregation
  - DateValgator
- Imputation
  - DateValNNer
  - DateValizer

1.6 ML Library

- DecisionTreeLearners
  - Index
  - AutoDocs

- `TSML.DecisionTreeLearners.Adaboost`
- `TSML.DecisionTreeLearners.PrunedTree`

- `TSML.DecisionTreeLearners.RandomForest`
- `TSML.Outliernicers.Outliernicer`
- `TSML.Plotters.Plotter`
- `TSML.TSMLTypes.fit!`
- `TSML.TSMLTypes.fit!`
- `TSML.TSMLTypes.fit!`
- `TSML.TSMLTypes.transform!`
- `TSML.TSMLTypes.transform!`
- `TSML.TSMLTypes.transform!`
- `TSML.TSMLTypes.transform!`

Part II

Tutorial

Chapter 2

Aggregators and Imputers

The package assumes a two-column table composed of `Dates` and `Values`. The first part of the workflow aggregates values based on the specified date-time interval which minimizes occurence of missing values and noise. The aggregated data is then left-joined to the complete sequence of `DateTime` in a specified date-time interval. Remaining missing values are replaced by k nearest neighbors where k is the symmetric distance from the location of missing value. This replacement algo is called several times until there are no more missing values.

Let us create a Date, Value table with some missing values and output the first 15 rows. We will then apply some TSML functions to normalize/clean the data. Below is the code of the `generateDataWithMissing()` function:

```
using Random, Dates, DataFrames
function generateDataWithMissing()
   Random.seed!(123)
   gdate = DateTime(2014,1,1):Dates.Minute(15):DateTime(2016,1,1)
   gval = Array{Union{Missing,Float64}}(rand(length(gdate)))
   gmissing = 50000
   gndxmissing = Random.shuffle(1:length(gdate))[1:gmissing]
   df = DataFrame(Date=gdate,Value=gval)
   df[:Value][gndxmissing] .= missing
   return df
end
```

```
julia> X = generateDataWithMissing();

julia> first(X,15)
15×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  missing
 2    2014-01-01T00:15:00  missing
 3    2014-01-01T00:30:00  missing
 4    2014-01-01T00:45:00  missing
 5    2014-01-01T01:00:00  missing
 6    2014-01-01T01:15:00  missing
 7    2014-01-01T01:30:00  missing
 8    2014-01-01T01:45:00  0.0521332
 9    2014-01-01T02:00:00  0.26864
 10   2014-01-01T02:15:00  0.108871
 11   2014-01-01T02:30:00  0.163666
 12   2014-01-01T02:45:00  0.473017
```

```
 13   2014-01-01T03:00:00  0.865412
 14   2014-01-01T03:15:00  missing
 15   2014-01-01T03:30:00  missing
```

2.1 DateValgator

You'll notice several blocks of missing in the table above with reading frequency of every 15 minutes. To minimize noise and lessen the occurrence of missing values, let's aggregate our dataset by taking the hourly median using the `DateValgator` transformer.

```
using TSML
using TSML.TSMLTypes
using TSML.Utils
using TSML.TSMLTransformers
using TSML: DateValgator

dtvlgator = DateValgator(Dict(:dateinterval=>Dates.Hour(1)))
fit!(dtvlgator,X)
results = transform!(dtvlgator,X)
```

```
julia> first(results,10)
10×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  missing
 2    2014-01-01T01:00:00  missing
 3    2014-01-01T02:00:00  0.108871
 4    2014-01-01T03:00:00  0.473017
 5    2014-01-01T04:00:00  0.361194
 6    2014-01-01T05:00:00  0.582318
 7    2014-01-01T06:00:00  0.918165
 8    2014-01-01T07:00:00  0.614255
 9    2014-01-01T08:00:00  0.690462
 10   2014-01-01T09:00:00  0.92049
```

The occurrence of missing values is now reduced because of the hourly aggregation. While the default is hourly aggregation, you can easily change it by using a different interval in the argument during instance creation. Below indicates every 30 minutes interval.

```
julia> dtvlgator = DateValgator(Dict(:dateinterval=>Dates.Minute(30)))
TSML.TSMLTransformers.DateValgator(nothing, Dict{Symbol,Any}(:dateinterval=>30
↪   minutes,:aggregator=>:median))
```

`DateValgator` is one of the several TSML transformers to preprocess and clean the time series data. In order to create additional transformers to extend TSML, each transformer must overload the two `Transformer` functions:`fit!` and `transform!`. `DateValgator fit!` performs initial setups of necessary parameters and validation of arguments while its `transform!` function contains the algorithm for aggregation.

For machine learning prediction and classification transformer, `fit!` function is equivalent to ML training or parameter optimization, while the `transform!` function is for doing the actual prediction. The later part of the tutorial will provide an example how to add a `Transformer` to extend the functionality of TSML.

2.2 DateValNNer

Let's perform further processing to replace the remaining missing values with their nearest neighbors. We will use `DateValNNer` which is a TSML transformer to process the output of `DateValgator`. `DateValNNer` can also process non-aggregated data by first running similar workflow of `DateValgator` before performing its imputation routine.

```
using TSML: DateValNNer

datevalnner = DateValNNer(Dict(:dateinterval=>Dates.Hour(1)))
fit!(datevalnner, X)
results = transform!(datevalnner,X)
```

```
julia> first(results,10)
10×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.108871
 2    2014-01-01T01:00:00  0.108871
 3    2014-01-01T02:00:00  0.108871
 4    2014-01-01T03:00:00  0.473017
 5    2014-01-01T04:00:00  0.361194
 6    2014-01-01T05:00:00  0.582318
 7    2014-01-01T06:00:00  0.918165
 8    2014-01-01T07:00:00  0.614255
 9    2014-01-01T08:00:00  0.690462
 10   2014-01-01T09:00:00  0.92049
```

After running the `DateValNNer`, it's guaranteed that there will be no more missing data unless the input are all missing data.

2.3 DateValizer

One more imputer to replace missing data is `DateValizer`. It computes the hourly median over 24 hours and use the `hour => median` hashmap learned to replace missing data using `hour` as the key. In this implementation, `fit!` function is doing the training of parameters by computing the medians and save it for the `transform!` function to use for imputation. It is possible that the hashmap can contain missing values in cases where the pooled hourly median in a particular hour have all missing data. Below is a sample workflow to replace missing data in X with the hourly medians.

```
using TSML: DateValizer

datevalizer = DateValizer(Dict(:dateinterval=>Dates.Hour(1)))
fit!(datevalizer, X)
results = transform!(datevalizer,X)
```

```
julia> first(results,10)
10×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.498827
 2    2014-01-01T01:00:00  0.500748
```

```
3    2014-01-01T02:00:00  0.108871
4    2014-01-01T03:00:00  0.473017
5    2014-01-01T04:00:00  0.361194
6    2014-01-01T05:00:00  0.582318
7    2014-01-01T06:00:00  0.918165
8    2014-01-01T07:00:00  0.614255
9    2014-01-01T08:00:00  0.690462
10   2014-01-01T09:00:00  0.92049
```

Chapter 3

Pipeline

Instead of calling `fit!` and `transform!` for each transformer to process time series data, we can use the `Pipeline` transformer which does this automatically by iterating through the transformers and calling `fit!` and `transform!` repeatedly for each transformer in its argument.

Let's start again by using a function to generate a time series dataframe with some missing data.

```
julia> X = generateDataWithMissing();

julia> first(X,15)
15×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  missing
 2    2014-01-01T00:15:00  missing
 3    2014-01-01T00:30:00  missing
 4    2014-01-01T00:45:00  missing
 5    2014-01-01T01:00:00  missing
 6    2014-01-01T01:15:00  missing
 7    2014-01-01T01:30:00  missing
 8    2014-01-01T01:45:00  0.0521332
 9    2014-01-01T02:00:00  0.26864
 10   2014-01-01T02:15:00  0.108871
 11   2014-01-01T02:30:00  0.163666
 12   2014-01-01T02:45:00  0.473017
 13   2014-01-01T03:00:00  0.865412
 14   2014-01-01T03:15:00  missing
 15   2014-01-01T03:30:00  missing
```

3.1 Workflow of Pipeline

Let's use the pipeline transformer to aggregate and impute:

```
using Dates
using TSML
using TSML.TSMLTypes
using TSML.TSMLTransformers
using TSML: Pipeline
using TSML: DateValgator
```

```
using TSML: DateValNNer

dtvalgator = DateValgator(Dict(:dateinterval => Dates.Hour(1)))
dtvalnner = DateValNNer(Dict(:dateinterval => Dates.Hour(1)))

mypipeline = Pipeline(
  Dict( :transformers => [
            dtvalgator,
            dtvalnner
         ]
  )
)

fit!(mypipeline,X)
results = transform!(mypipeline,X)
```

```
julia> first(results,10)
10×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.108871
 2    2014-01-01T01:00:00  0.108871
 3    2014-01-01T02:00:00  0.108871
 4    2014-01-01T03:00:00  0.473017
 5    2014-01-01T04:00:00  0.361194
 6    2014-01-01T05:00:00  0.582318
 7    2014-01-01T06:00:00  0.918165
 8    2014-01-01T07:00:00  0.614255
 9    2014-01-01T08:00:00  0.690462
 10   2014-01-01T09:00:00  0.92049
```

Using the `Pipeline` transformer, it becomes straightforward to process the time series data. It also becomes trivial to extend TSML functionality by adding more transformers and making sure each support the `fit!` and `transform!` interfaces. Any new transformer can then be easily added to the `Pipeline` workflow without invasively changing the existing codes.

3.2 Extending TSML

To illustrate how simple it is to add a new transformer, below extends TSML by adding `CSVReader` transformer and added in the pipeline to process CSV data:

```
using TSML.TSMLTypes
using TSML.Utils
import TSML.TSMLTypes.fit!
import TSML.TSMLTypes.transform!

using CSV

mutable struct CSVReader <: Transformer
    model
    args
    function CSVReader(args=Dict())
        default_args = Dict(
```

```
            :filename => "",
            :dateformat => ""
        )
        new(nothing,mergedict(default_args,args))
    end
end

function fit!(csvrdr::CSVReader,x::T=[],y::Vector=[]) where {T<:Union{DataFrame,Vector,Matrix}}
    fname = csvrdr.args[:filename]
    fmt = csvrdr.args[:dateformat]
    (fname != "" && fmt != "") || error("missing filename or date format")
    model = csvrdr.args
end

function transform!(csvrdr::CSVReader,x::T=[]) where {T<:Union{DataFrame,Vector,Matrix}}
    fname = csvrdr.args[:filename]
    fmt = csvrdr.args[:dateformat]
    df = CSV.read(fname)
    ncol(df) == 2 || error("dataframe should have only two columns: Date,Value")
    rename!(df,names(df)[1]=>:Date,names(df)[2]=>:Value)
    df[:Date] = DateTime.(df[:Date],fmt)
    df
end
```

Instead of passing table X that contains the time series, we will add an instance of the`CSVReader` at the start of the array of transformers in the pipeline to read the csv data. CSVReader `transform!` function converts the csv time series table into a dataframe, which will be consumed by the next transformer in the pipeline for processing.

```
fname = joinpath(dirname(pathof(TSML)),"../data/testdata.csv")
csvreader = CSVDateValReader(Dict(:filename=>fname,:dateformat=>"d/m/y H:M"))
fit!(csvreader)
csvdata = transform!(csvreader)
```

```
julia> first(csvdata,10)
10×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:06:00  10.0
 2    2014-01-01T00:18:00  10.0
 3    2014-01-01T00:29:00  10.0
 4    2014-01-01T00:40:00  9.9
 5    2014-01-01T00:51:00  9.9
 6    2014-01-01T01:02:00  10.0
 7    2014-01-01T01:13:00  9.8
 8    2014-01-01T01:24:00  10.0
 9    2014-01-01T01:35:00  9.8
 10   2014-01-01T01:46:00  10.0
```

Let us now include the newly created `CSVReader` in the pipeline to read the csv data and process it by aggregation and imputation.

```
mypipeline = Pipeline(
  Dict( :transformers => [
```

```
            csvreader,
            dtvalgator,
            dtvalnner
         ]
  )
)

fit!(mypipeline)
results = transform!(mypipeline)
```

```
julia> first(results,10)
10×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  10.0
 2    2014-01-01T01:00:00  9.9
 3    2014-01-01T02:00:00  10.0
 4    2014-01-01T03:00:00  10.0
 5    2014-01-01T04:00:00  10.0
 6    2014-01-01T05:00:00  10.0
 7    2014-01-01T06:00:00  10.0
 8    2014-01-01T07:00:00  9.8
 9    2014-01-01T08:00:00  9.85
 10   2014-01-01T09:00:00  9.9
```

Notice that there is no more the need to pass X in the arguments of `fit!` and `transform` because the data is now transmitted by the `CSVReader` instance to the other transformers in the pipeline.

Chapter 4

Statistical Metrics

Each TS can be evaluated to extract its statistical features which can be used for data quality assessment, data discovery by clustering and classification, and anomaly characterization among others.

TSML relies on `Statifier` to perform statistical metrics on the TS which can be configured to extract the statistics of missing blocks aside from the non-missing elements. Some of the scalar statistics it uses include: pacf, acf, autocor, quartiles, mean, median, max, min, kurtosis, skewness, variation, standard error, entropy, etc. It has only one argument `:processmissing => true` which indicates whether to include the statistics of missing data.

Let us again start generating an artificial data with missing values using the `generateDataWithMissing()` described in the beginning of tutorial.

```
julia> X = generateDataWithMissing();

julia> first(X,15)
15×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  missing
 2    2014-01-01T00:15:00  missing
 3    2014-01-01T00:30:00  missing
 4    2014-01-01T00:45:00  missing
 5    2014-01-01T01:00:00  missing
 6    2014-01-01T01:15:00  missing
 7    2014-01-01T01:30:00  missing
 8    2014-01-01T01:45:00  0.0521332
 9    2014-01-01T02:00:00  0.26864
 10   2014-01-01T02:15:00  0.108871
 11   2014-01-01T02:30:00  0.163666
 12   2014-01-01T02:45:00  0.473017
 13   2014-01-01T03:00:00  0.865412
 14   2014-01-01T03:15:00  missing
 15   2014-01-01T03:30:00  missing
```

4.1 Statifier for Both Non-Missing and Missing Values

TSML includes `Statifier` transformer that computes scalar statistics to characterize the time series data. By default, it also computes statistics of missing blocks of data. To disable this feature, one can pass `:processmissing => false` to the argument during its instance creation. Below illustrates this workflow.

```julia
using Dates
using TSML
using TSML.TSMLTypes
using TSML.TSMLTransformers
using TSML: Pipeline
using TSML: DateValgator
using TSML: DateValNNer
using TSML: Statifier

dtvalgator = DateValgator(Dict(:dateinterval => Dates.Hour(1)))
dtvalnner = DateValNNer(Dict(:dateinterval => Dates.Hour(1)))
dtvalizer = DateValizer(Dict(:dateinterval => Dates.Hour(1)))
stfier = Statifier(Dict(:processmissing => true))

mypipeline = Pipeline(
  Dict( :transformers => [
            dtvalgator,
            stfier
         ]
  )
)

fit!(mypipeline,X)
results = transform!(mypipeline,X)
```

```
julia> show(results,allcols=true)
1×26 DataFrames.DataFrame
 Row  tstart               tend                 sfreq     count
      Dates.DateTime       Dates.DateTime       Float64   Int64

 1    2014-01-01T00:00:00  2016-01-01T00:00:00  0.999943  13055

 Row  max       min          median    mean      q1        q2
      Float64   Float64      Float64   Float64   Float64   Float64

 1    0.999751  0.000456433  0.500944  0.502196  0.146624  0.251897

 Row  q25       q75       q8        q9        kurtosis   skewness
      Float64   Float64   Float64   Float64   Float64    Float64

 1    0.304154  0.701455  0.750592  0.860293  -0.928492  0.00290679

 Row  variation  entropy  autocor    pacf       bmedian  bmean
      Float64    Float64  Float64    Float64    Float64  Float64

 1    0.510627   3544.12  0.0475025  0.0477302  1.0      1.3268

 Row  bq25     bq75     bmin     bmax
      Float64  Float64  Float64  Float64

 1    1.0      1.0      1.0      6.0
```

4.2 Statifier for Non-Missing Values only

If you are not intested with the statistics of the missing blocks, you can disable missing blocks stat summary by indicating `:processmissing => false` in the instance argument:

```
stfier = Statifier(Dict(:processmissing=>false))
mypipeline = Pipeline(
  Dict( :transformers => [
            dtvalgator,
            stfier
         ]
  )
)
fit!(mypipeline,X)
results = transform!(mypipeline,X)
```

```
julia> show(results,allcols=true)
1×20 DataFrames.DataFrame
 Row  tstart               tend                 sfreq     count
      Dates.DateTime       Dates.DateTime       Float64   Int64

 1    2014-01-01T00:00:00  2016-01-01T00:00:00  0.999943  13055

 Row  max       min          median    mean      q1        q2
      Float64   Float64      Float64   Float64   Float64   Float64

 1    0.999751  0.000456433  0.500944  0.502196  0.146624  0.251897

 Row  q25       q75       q8        q9        kurtosis   skewness
      Float64   Float64   Float64   Float64   Float64    Float64

 1    0.304154  0.701455  0.750592  0.860293  -0.928492  0.00290679

 Row  variation  entropy  autocor    pacf
      Float64    Float64  Float64    Float64

 1    0.510627   3544.12  0.0475025  0.0477302
```

4.3 Statifier After Imputation

Let us check the statistics after the imputation by adding `DateValNNer` instance in the pipeline. We expect that if the imputation is successful, the stats for missing blocks will all be NaN because stats of empty set is an NaN.

```
stfier = Statifier(Dict(:processmissing=>true))
mypipeline = Pipeline(
  Dict( :transformers => [
            dtvalgator,
            dtvalnner,
            stfier
         ]
  )
)
fit!(mypipeline,X)
results = transform!(mypipeline,X)
```

```
julia> show(results,allcols=true)
1×26 DataFrames.DataFrame
 Row  tstart               tend                 sfreq     count
      Dates.DateTime       Dates.DateTime       Float64   Int64

 1    2014-01-01T00:00:00  2016-01-01T00:00:00  0.999943  17521

 Row  max       min          median   mean      q1        q2
      Float64   Float64      Float64  Float64   Float64   Float64

 1    0.999751  0.000456433  0.50022  0.501232  0.167654  0.274743

 Row  q25       q75       q8       q9        kurtosis   skewness
      Float64   Float64   Float64  Float64   Float64    Float64

 1    0.320764  0.680764  0.7263   0.838924  -0.789838  0.00479609

 Row  variation  entropy  autocor   pacf      bmedian  bmean
      Float64    Float64  Float64   Float64   Float64  Float64

 1    0.485412   4896.49  0.279811  0.273143  NaN      NaN

 Row  bq25     bq75     bmin     bmax
      Float64  Float64  Float64  Float64

 1    NaN      NaN      NaN      NaN
```

As we expected, the imputation is successful and there are no more missing values in the processed time series dataset.

Let's try with the other imputation using `DateValizer` and validate that there are no more missing values based on the stats.

```
stfier = Statifier(Dict(:processmissing=>true))
mypipeline = Pipeline(
  Dict( :transformers => [
            dtvalgator,
            dtvalizer,
            stfier
         ]
  )
)
fit!(mypipeline,X)
results = transform!(mypipeline,X)
```

```
julia> show(results,allcols=true)
1×26 DataFrames.DataFrame
 Row  tstart               tend                 sfreq     count
      Dates.DateTime       Dates.DateTime       Float64   Int64

 1    2014-01-01T00:00:00  2016-01-01T00:00:00  0.999943  17521

 Row  max       min          median    mean      q1        q2
      Float64   Float64      Float64   Float64   Float64   Float64

 1    0.999751  0.000456433  0.500748  0.502152  0.185242  0.319607
```

```
Row  q25       q75       q8        q9        kurtosis   skewness
     Float64   Float64   Float64   Float64   Float64    Float64

1    0.377796  0.625371  0.685253  0.820227  -0.225831  0.00396169

Row  variation  entropy  autocor    pacf       bmedian  bmean
     Float64    Float64  Float64    Float64    Float64  Float64

1    0.441044   5088.2   0.0284397  0.0284876  NaN      NaN

Row  bq25     bq75     bmin     bmax
     Float64  Float64  Float64  Float64

1    NaN      NaN      NaN      NaN
```

Indeed, the imputation got rid of the missing values.

Chapter 5

Monotonic Detection and Plotting

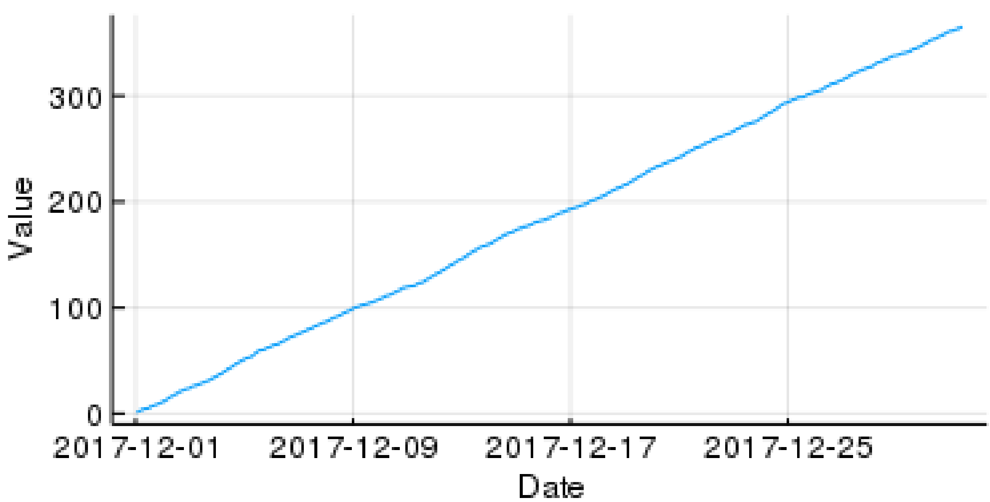
One important preprocessing step for time series data processing is the detection of monotonic data and transform it to non-monotonic type by using the finite difference operator.

5.1 Artificial Data Example

Let's create an artificial monotonic data and apply our monotonic transformer to normalize it. We can use the `Plotter` filter to visualize the generated data.

```
using Dates, DataFrames, Random
using TSML, TSML.Utils, TSML.TSMLTypes
using TSML: Plotter

Random.seed!(123)
pltr = Plotter(Dict(:interactive => false, :pdfoutput => true))
mdates = DateTime(2017,12,1,1):Dates.Hour(1):DateTime(2017,12,31,10) |> collect
mvals = rand(length(mdates)) |> cumsum
df =  DataFrame(Date=mdates ,Value = mvals)
fit!(pltr,df)
transform!(pltr,df)
```



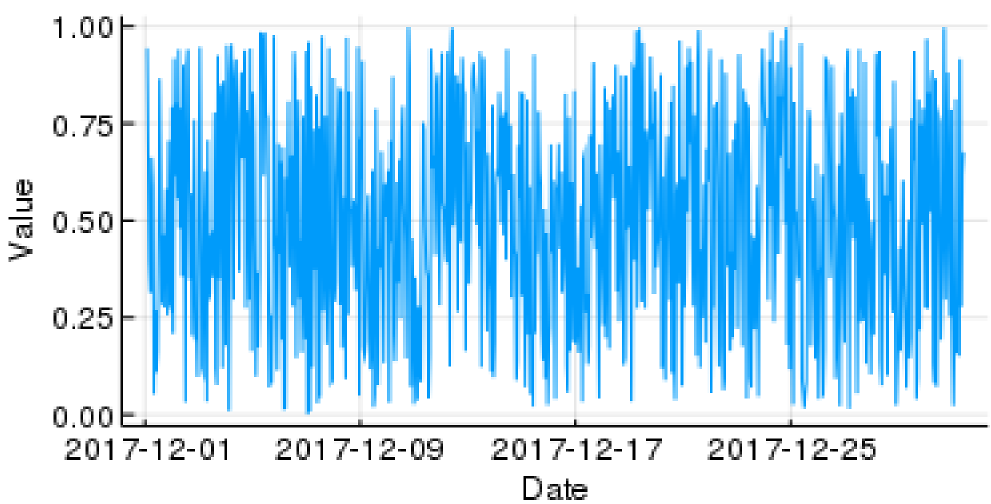
Now that we have a monotonic data, let's use the `Monotonicer` to normalize and plot the result:

```
using TSML, TSML.Utils, TSML.TSMLTypes
using TSML.TSMLTransformers
using TSML: Monotonicer

mono = Monotonicer(Dict())

pipeline = Pipeline(Dict(
   :transformers => [mono,pltr]
   )
)

fit!(pipeline,df)
res=transform!(pipeline,df)
```



5.2 Real Data Example

We will now apply the entire pipeline starting from reading csv data, aggregate, impute, normalize if it's monotonic, and plot. We will consider three different data types: a regular time series data, a monotonic data, and a daily monotonic data. The difference between monotonic and daily monotonic is that the values in daily monotonic resets to zero or some baseline and cumulatively increases in a day until the next day where it resets to zero or some baseline value. `Monotonicer` automatically detects these three different types and apply the corresponding normalization accordingly.

```
using TSML: DateValgator, DateValNNer, Statifier, Monotonicer
regularfile = joinpath(dirname(pathof(TSML)),"../data/typedetection/regular.csv")
monofile = joinpath(dirname(pathof(TSML)),"../data/typedetection/monotonic.csv")
dailymonofile = joinpath(dirname(pathof(TSML)),"../data/typedetection/dailymonotonic.csv")

regularfilecsv = CSVDateValReader(Dict(:filename=>regularfile,:dateformat=>"dd/mm/yyyy HH:MM"))
monofilecsv = CSVDateValReader(Dict(:filename=>monofile,:dateformat=>"dd/mm/yyyy HH:MM"))
dailymonofilecsv = CSVDateValReader(Dict(:filename=>dailymonofile,:dateformat=>"dd/mm/yyyy
↪  HH:MM"))
```

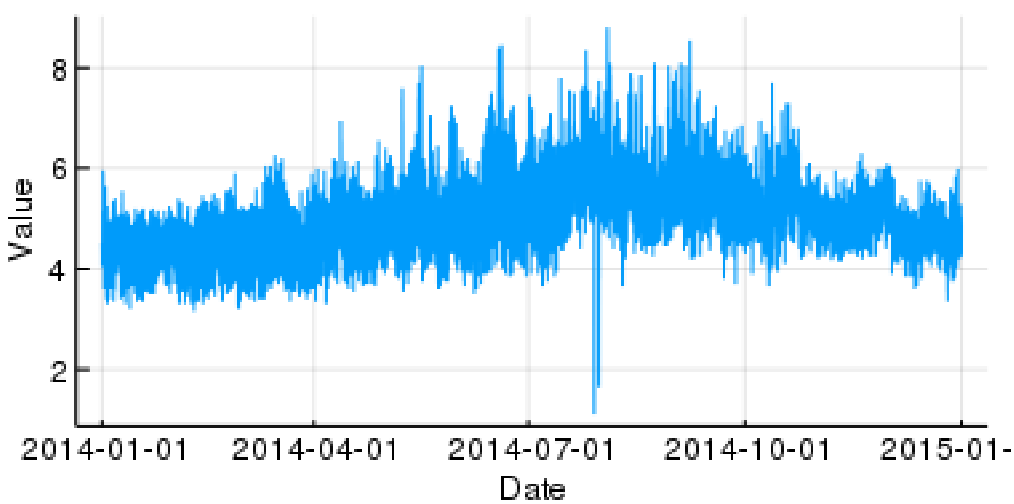
```
valgator = DateValgator(Dict(:dateinterval=>Dates.Hour(1)))
valnner = DateValNNer(Dict(:dateinterval=>Dates.Hour(1)))
stfier = Statifier(Dict(:processmissing=>true))
mono = Monotonicer(Dict())
```

5.3 Regular TS Processing

Let's test by feeding the regular time series type to the pipeline. We expect that for this type, `Monotonicer` will not perform further processing:

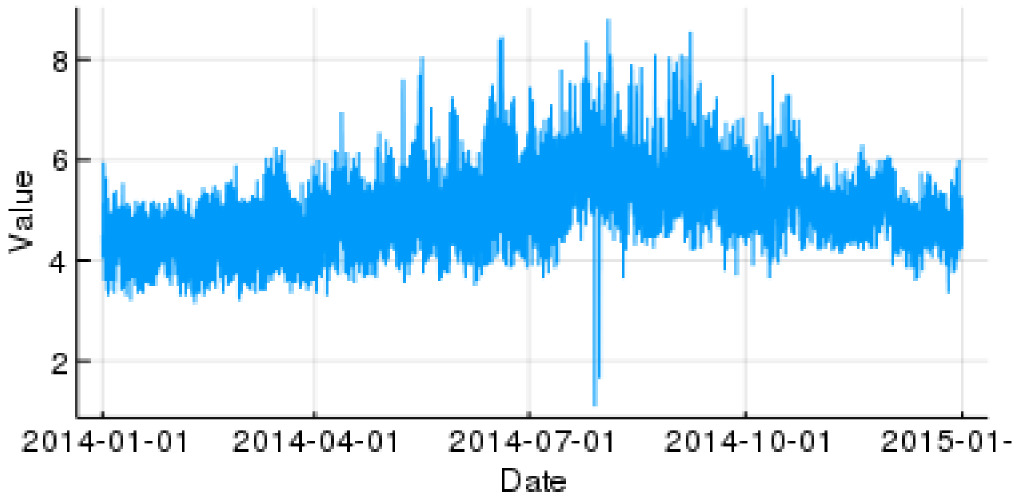
- Pipeline with `Monotonicer`: regular time series

```
pipeline = Pipeline(Dict(
    :transformers => [regularfilecsv,valgator,valnner,mono,pltr]
   )
)
fit!(pipeline)
transform!(pipeline)
```



- Pipeline without `Monotonicer`: regular time series

```
pipeline = Pipeline(Dict(
    :transformers => [regularfilecsv,valgator,valnner,pltr]
   )
)
fit!(pipeline)
transform!(pipeline)
```



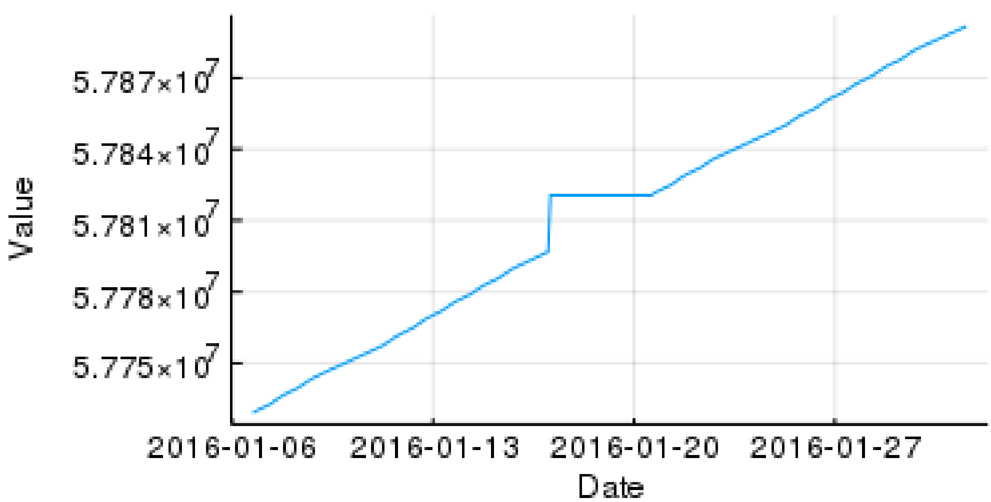
Notice that the plots are the same with or without the `Monotonicer` instance.

5.4 Monotonic TS Processing

Let's now feed the same pipeline with a monotonic csv data.

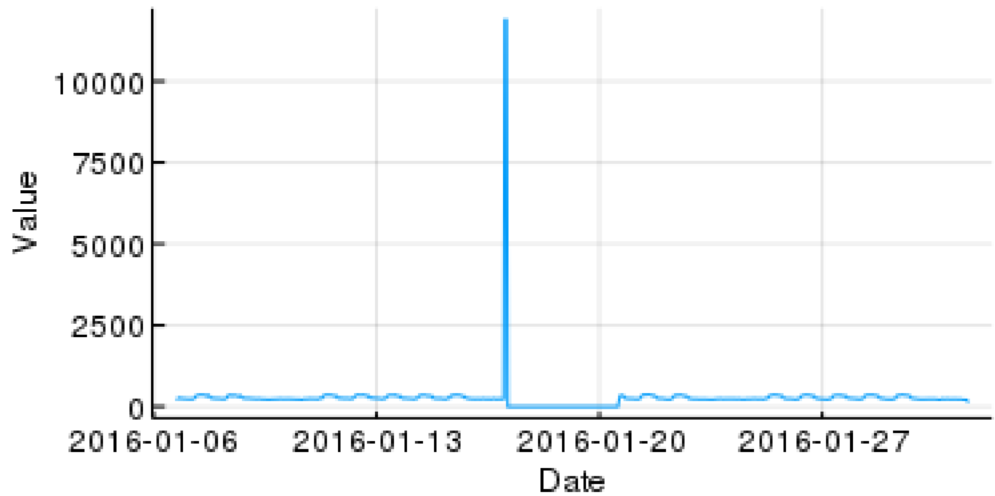
- Pipeline without `Monotonicer`: monotonic time series

```
pipeline = Pipeline(Dict(
     :transformers => [monofilecsv,valgator,valnner,pltr]
    )
)
fit!(pipeline)
transform!(pipeline)
```



- Pipeline with `Monotonicer`: monotonic time series

```
pipeline = Pipeline(Dict(
    :transformers => [monofilecsv,valgator,valnner,mono,pltr]
   )
)
fit!(pipeline)
transform!(pipeline)
```



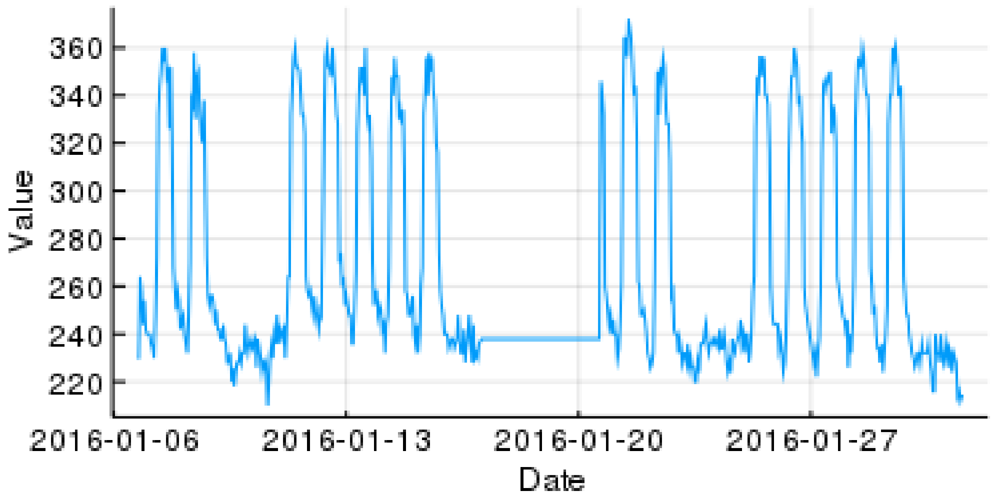
Notice that without the `Monotonicer` instance, the data is monotonic. Applying the `Monotonicer` instance in the pipeline converts the data into a regular time series but with outliers.

We can use the `Outliernicer` filter to remove outliers. Let's apply this filter after the `Monotonicer` and plot the result.

- Pipeline with `Monotonicer` and `Outliernicer`: monotonic time series

```
using TSML: Outliernicer
outliernicer = Outliernicer(Dict(:dateinterval=>Dates.Hour(1)));

pipeline = Pipeline(Dict(
    :transformers => [monofilecsv,valgator,valnner,mono, outliernicer,pltr]
   )
)
fit!(pipeline)
transform!(pipeline)
```

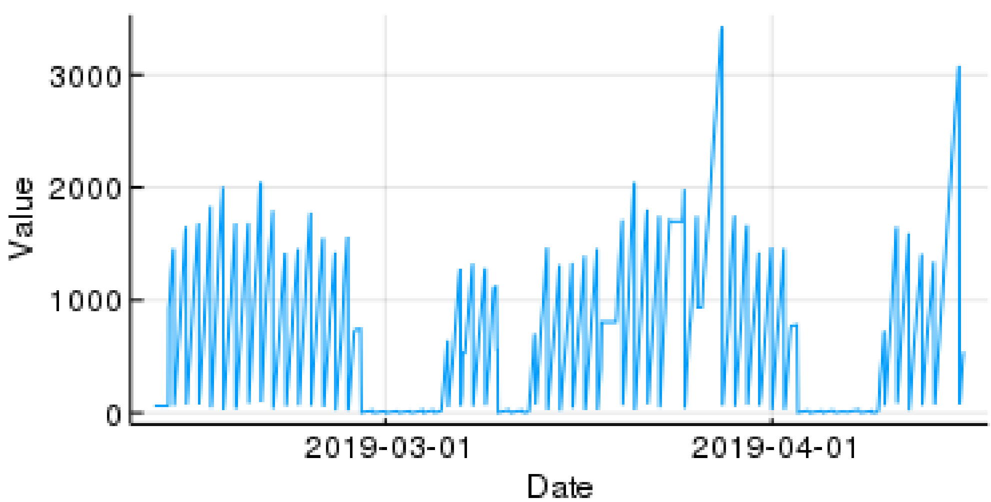


5.5 Daily Monotonic TS Processing

Lastly, let's feed the daily monotonic data using similar pipeline and examine its plot.

- Pipeline without `Monotonicer`: daily monotonic time series

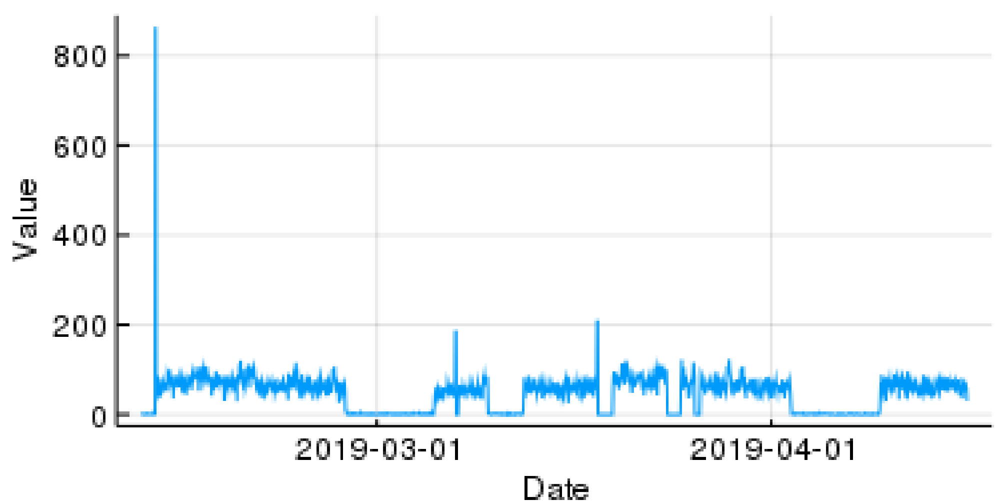
```
pipeline = Pipeline(Dict(
     :transformers => [dailymonofilecsv,valgator,valnner,pltr]
    )
)
fit!(pipeline)
transform!(pipeline)
```



This plot is characterized by monotonically increasing trend but resets to certain baseline value at the end of the day and repeat similar trend daily. The challenge for the monotonic normalizer is to differentiate between daily monotonic from the typical monotonic function to apply the correct normalization.

- Pipeline with `Monotonicer`: daily monotonic time series

```
pipeline = Pipeline(Dict(
     :transformers => [dailymonofilecsv,valgator,valnner,mono,pltr]
    )
)
fit!(pipeline)
transform!(pipeline)
```

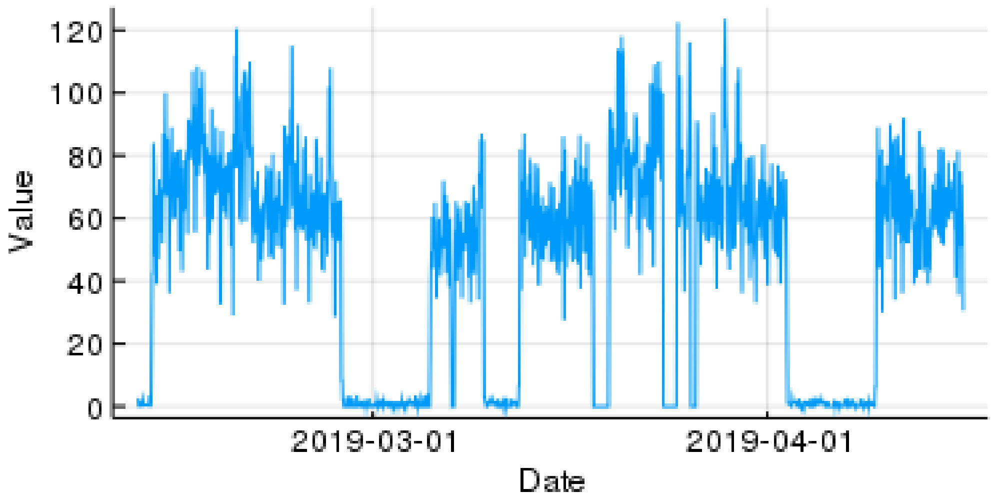


While the `Monotonicer` filter is able to transform the data into a regular time series, there are significant outliers due to noise and the nature of this kind of data or sensor.

Let's remove the outliers by applying the `Outliernicer` filter and examine the result.

- Pipeline with `Monotonicer` and `Outliernicer`: daily monotonic time series

```
pipeline = Pipeline(Dict(
     :transformers => [dailymonofilecsv,valgator,valnner,mono,outliernicer,pltr]
    )
)
fit!(pipeline)
transform!(pipeline)
```



The `Outliernicer` filter effectively removed the outliers as shown in the plot.

Chapter 6

TS Data Discovery

We have enough building blocks to perform data discovery given a bunch of time series data generated by sensors. Processing hundreds or thousands of time series data is becoming a common occurrence and typical challenge nowadays with the rapid adoption of IoT technology in buildings, manufacturing industries, etc.

In this section, we will use those transformers discussed in the previous sections to normalize and extract the statistical features of TS. These extracted stat features will be used as input to a Machine learning model. We will train this model to learn the signatures of different TS types so that we can use it to classify unknown or unlabeled sensor data.

In this tutorial, we will use `TSClassifier` which works in the following context: Given a bunch of time-series with specific types. Get the statistical features of each, use these as inputs to a classifier with output as the TS type, train, and test. Another option is to use these stat features for clustering and check cluster quality. If accuracy is poor, add more stat features and repeat same process as outlined for training and testing. Assume that each time series during training is named based on their type which will be used as the target output. For example, temperature time series will be named as temperature?.csv where ? is any positive integer. Using this setup, the `TSClassifier` loops over each file in the `training` directory, get the stats and record these accumulated stat features into a dataframe and train the model to learn the input->output mapping during `fit!` operation. Apply the learned models in the `transform!` operation loading files in the `testing` directory.

The entire process of training to learn the appropriate parameters and classification to identify unlabeled data exploits the idea of the pipeline workflow discussed in the previous sections.

Let's illustrate the process by loading some sample data:

```
using Random
using TSML

Random.seed!(12345)

trdirname = joinpath(dirname(pathof(TSML)),"../data/realdatatsclassification/training")
tstdirname = joinpath(dirname(pathof(TSML)),"../data/realdatatsclassification/testing")
modeldirname = joinpath(dirname(pathof(TSML)),"../data/realdatatsclassification/model")
```

```
"/Users/ppalmes/.julia/packages/TSML/Ozjv9/src/../data/realdatatsclassification/model"
```

Here's the list of files for training:

```
show(readdir(trdirname) |> x->filter(y->match(r".csv",y) != nothing,x))
```

```
["AirOffTemp1.csv", "AirOffTemp2.csv", "AirOffTemp3.csv", "Energy1.csv", "Energy2.csv", "Energy3.
    csv", "Energy4.csv", "Energy6.csv", "Energy7.csv", "Energy8.csv", "Energy9.csv", "Pressure1.
    csv", "Pressure2.csv", "Pressure3.csv", "Pressure4.csv", "Pressure6.csv", "RetTemp11.csv", "
    RetTemp21.csv", "RetTemp41.csv", "RetTemp51.csv"]
```

and here are the files in testing directory:

```
show(readdir(tstdirname) |> x->filter(y->match(r".csv",y) != nothing,x))
```

```
["AirOffTemp4.csv", "AirOffTemp5.csv", "Energy10.csv", "Energy5.csv", "Pressure2.csv", "Pressure5
    .csv", "RetTemp31.csv"]
```

The files in testing directory doesn't need to be labeled but we use the labeling as a way to validate the effectiveness of the classifier. The labels will be used as the groundtruth during prediction/classification.

6.1 TSClassifier

Let us now setup an instance of the `TSClassifier` and pass the arguments containing the directory locations of files for training, testing, and modeling.

```
using TSML: TSClassifier
using TSML: fit!, transform!

tscl = TSClassifier(Dict(:trdirectory=>trdirname,
          :tstdirectory=>tstdirname,
          :modeldirectory=>modeldirname,
          :feature_range => 6:20,
          :num_trees=>20)
       )
```

Time to train our `TSClassifier` to learn the mapping between extracted stats features with the TS type.

```
julia> fit!(tscl)
getting stats of AirOffTemp1.csv
getting stats of AirOffTemp2.csv
getting stats of AirOffTemp3.csv
getting stats of Energy1.csv
getting stats of Energy2.csv
getting stats of Energy3.csv
getting stats of Energy4.csv
getting stats of Energy6.csv
getting stats of Energy7.csv
getting stats of Energy8.csv
getting stats of Energy9.csv
getting stats of Pressure1.csv
skipping due to error Pressure2.csv
getting stats of Pressure3.csv
getting stats of Pressure4.csv
getting stats of Pressure6.csv
getting stats of RetTemp11.csv
getting stats of RetTemp21.csv
getting stats of RetTemp41.csv
getting stats of RetTemp51.csv
```

```
RandomForest(Ensemble of Decision Trees
Trees:      10
Avg Leaves: 4.0
Avg Depth:  2.5,
↪  Dict{Symbol,Any}(:tstdirectory=>"/Users/ppalmes/.julia/packages/TSML/Ozjv9/src/../data/realdatatsclassificatio
↪  1,:purity_threshold=>1.0,:num_subfeatures=>0,:min_samples_split=>2,:min_samples_leaf=>1,:num_trees=>10,:min_pu
```

We can examine the extracted features saved by the model that is used for its training.

```
using CSV, DataFrames

mdirname = tscl.args[:modeldirectory]
modelfname=tscl.args[:juliarfmodelname]

trstatfname = joinpath(mdirname,modelfname*".csv")
res = CSV.read(trstatfname) |> DataFrame
```

```
julia> first(res,5)
5×22 DataFrames.DataFrame. Omitted printing of 18 columns
 Row  tstart               tend                 sfreq     count
      Dates.DateTime       Dates.DateTime       Float64   Int64

 1    2012-12-01T00:00:00  2013-01-01T00:00:00  0.998658  745
 2    2012-12-01T00:00:00  2013-01-01T00:00:00  0.998658  745
 3    2012-12-01T00:00:00  2013-01-01T00:00:00  0.998658  745
 4    2017-10-01T00:00:00  2018-10-30T23:00:00  0.999895  9480
 5    2017-10-01T00:00:00  2018-10-30T23:00:00  0.999895  9480
```

Let's check the accuracy of prediction with the test data using the `transform!` function.

```
julia> dfresults = transform!(tscl)
getting stats of AirOffTemp4.csv
getting stats of AirOffTemp5.csv
getting stats of Energy10.csv
getting stats of Energy5.csv
skipping due to error Pressure2.csv
getting stats of Pressure5.csv
getting stats of RetTemp31.csv
loading model from file:
↪  /Users/ppalmes/.julia/packages/TSML/Ozjv9/src/../data/realdatatsclassification/model/juliarfmodel.serialized
6×2 DataFrames.DataFrame
 Row  fname            predtype
      String           SubStrin…

 1    AirOffTemp4.csv  AirOffTemp
 2    AirOffTemp5.csv  AirOffTemp
 3    Energy10.csv     AirOffTemp
 4    Energy5.csv      Energy
 5    Pressure5.csv    Pressure
 6    RetTemp31.csv    Energy
```

The table above shows the prediction corresponding to each filename which is the groundtruth. We can compute the accuracy by extracting from the filename the TS type and compare it with the corresponding prediction. Below computes the prediction accuracy:

```
prediction = dfresults[:predtype]
fnames = dfresults[:fname]
myregex = r"(?<dtype>[A-Z _ - a-z]+)(?<number>\d*).(?<ext>\w+)"
groundtruth=map(fnames) do fname
  mymatch=match(myregex,fname)
  mymatch[:dtype]
end
```

```
julia> sum(groundtruth .== prediction) / length(groundtruth) * 100
66.66666666666666
```

Of course we need more data to split between training and testing to improve accuracy and get a more stable measurement of performance.

Part III

Manual

Chapter 7

Date Processing

7.1 Date Preprocessing

Extracting the Date features in a `Date,Value` table follows similar workflow with the value preprocessing of the previous section. The main difference is we are only interested on the date corresponding to the last column of the values generated by the `Matrifier`. This last column contains the values before the prediction happens and the dates corresponding to these values carry significant information based on recency compared to the other dates.

Let us start by creating a Date,Value dataframe similar to the previous section.

```
using Dates
using TSML, TSML.Utils, TSML.TSMLTypes
using TSML.TSMLTransformers
using DataFrames

lower = DateTime(2017,1,1)
upper = DateTime(2018,1,31)
dat=lower:Dates.Day(1):upper |> collect
vals = rand(length(dat))
x = DataFrame(Date=dat,Value=vals)
```

```
julia> first(x,5)
5×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2017-01-01T00:00:00  0.184567
 2    2017-01-02T00:00:00  0.797453
 3    2017-01-03T00:00:00  0.385266
 4    2017-01-04T00:00:00  0.992998
 5    2017-01-05T00:00:00  0.543396
```

Dateifier

Let us create an instance of `Dateifier` passing the size of row, stride, and steps ahead to predict:

```
mtr = Dateifier(Dict(:ahead=>24,:size=>24,:stride=>5))
fit!(mtr,x)
res = transform!(mtr,x)
```

```
julia> first(res,5)
5×8 DataFrames.DataFrame
 Row  year   month  day    hour   week   dow    doq    qoy
      Int64  Int64  Int64  Int64  Int64  Int64  Int64  Int64

 1    2018   1      7      0      1      7      7      1
 2    2018   1      2      0      1      2      2      1
 3    2017   12     28     0      52     4      89     4
 4    2017   12     23     0      51     6      84     4
 5    2017   12     18     0      51     1      79     4
```

The model `transform!` output extracts automatically several date features such as year, month, day, hour, week, day of the week, day of quarter, quarter of year.

ML Features: Matrifier and Datefier

You can then combine the outputs in both the `Matrifier` and `Datefier` as input features to a machine learning model. Below is an example of the workflow where the code extracts the Date and Value features combining them to form a matrix of features as input to a machine learning model.

```
commonargs = Dict(:ahead=>3,:size=>5,:stride=>2)
dtr = Dateifier(commonargs)
mtr = Matrifier(commonargs)

lower = DateTime(2017,1,1)
upper = DateTime(2018,1,31)
dat=lower:Dates.Day(1):upper |> collect
vals = rand(length(dat))
X = DataFrame(Date=dat,Value=vals)

fit!(mtr,X)
valuematrix = transform!(mtr,X)
fit!(dtr,X)
datematrix = transform!(dtr,X)
mlfeatures = hcat(datematrix,valuematrix)
```

```
julia> first(mlfeatures,5)
5×14 DataFrames.DataFrame. Omitted printing of 6 columns
 Row  year   month  day    hour   week   dow    doq    qoy
      Int64  Int64  Int64  Int64  Int64  Int64  Int64  Int64

 1    2018   1      28     0      4      7      28     1
 2    2018   1      26     0      4      5      26     1
 3    2018   1      24     0      4      3      24     1
 4    2018   1      22     0      4      1      22     1
 5    2018   1      20     0      3      6      20     1
```

Chapter 8

Value Processing

8.1 Value Preprocessing

In order to process 1-D TS as input for ML model, it has to be converted into Matrix form where each row represents a slice of 1-D TS representing daily/hourly/weekly pattern depending on the size of the chunk, stride, and number of steps ahead for prediction. Below illustrates the processing workflow to `Matrify` a 1-D TS.

For illustration purposes, the code below generates a Date,Value dataframe where the values are just a sequece of integer from 1 to the length of the date sequence. We use this simple sequence to have a better understanding how the slicing of rows, steps ahead, and the stride to create the `Matrified` output is generated.

```
using Dates
using TSML, TSML.Utils, TSML.TSMLTypes
using TSML.TSMLTransformers
using DataFrames

lower = DateTime(2017,1,1)
upper = DateTime(2017,1,5)
dat=lower:Dates.Hour(1):upper |> collect
vals = 1:length(dat)
x = DataFrame(Date=dat,Value=vals)
```

```
julia> last(x,5)
5×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Int64

 1    2017-01-04T20:00:00  93
 2    2017-01-04T21:00:00  94
 3    2017-01-04T22:00:00  95
 4    2017-01-04T23:00:00  96
 5    2017-01-05T00:00:00  97
```

Matrifier

Let us create an instance of Matrifier passing the size of row, stride, and steps ahead to predict:

```
mtr = Matrifier(Dict(:ahead=>6,:size=>6,:stride=>3))
fit!(mtr,x)
res = transform!(mtr,x)
```

```
julia> first(res,5)
5×7 DataFrames.DataFrame
 Row  x1     x2     x3     x4     x5     x6     output
      Int64  Int64  Int64  Int64  Int64  Int64  Int64

 1    86     87     88     89     90     91     97
 2    83     84     85     86     87     88     94
 3    80     81     82     83     84     85     91
 4    77     78     79     80     81     82     88
 5    74     75     76     77     78     79     85
```

In this example, we have hourly values. We indicated in the `Matrifier` to generate a matrix where the size of each row is 6 hours, steps ahead for prediction is 6 hours and the stride of 3 hours. There are 7 columns because the last column indicates the value indicated by the steps `ahead` argument.

Let us try to make a matrix with the size of 6 hours, steps ahead of 2 hours, and a stride of 3 hours:

```
mtr = Matrifier(Dict(:ahead=>2,:size=>6,:stride=>3))
fit!(mtr,x)
res = transform!(mtr,x)
```

```
julia> first(res,5)
5×7 DataFrames.DataFrame
 Row  x1     x2     x3     x4     x5     x6     output
      Int64  Int64  Int64  Int64  Int64  Int64  Int64

 1    90     91     92     93     94     95     97
 2    87     88     89     90     91     92     94
 3    84     85     86     87     88     89     91
 4    81     82     83     84     85     86     88
 5    78     79     80     81     82     83     85
```

Chapter 9

Aggregation

9.1 Aggregation

`DateValgator` is a data type that supports operation for aggregation to minimize noise and lessen the occurrence of missing data. It expects to receive one argument which is the date-time interval for grouping values by taking their median. For example, hourly median as the basis of aggregation can be carried out by passing this argument: `:dateinterval => Dates.Hour(1)`

To illustrate `DateValgator` usage, let's start by generating an artificial data with sample frequencey every 5 minutes and print the first 10 rows.

```
using Dates, DataFrames
gdate = DateTime(2014,1,1):Dates.Minute(5):DateTime(2014,5,1)
gval = rand(length(gdate))

df = DataFrame(Date=gdate,Value=gval)
```

```
julia> first(df,10)
10×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.102505
 2    2014-01-01T00:05:00  0.67414
 3    2014-01-01T00:10:00  0.595703
 4    2014-01-01T00:15:00  0.337895
 5    2014-01-01T00:20:00  0.127011
 6    2014-01-01T00:25:00  0.230526
 7    2014-01-01T00:30:00  0.662658
 8    2014-01-01T00:35:00  0.59878
 9    2014-01-01T00:40:00  0.0888085
 10   2014-01-01T00:45:00  0.244319
```

DateValgator

Let's apply the aggregator and try diffent groupings: hourly vs half hourly vs daily aggregates of the data.

```
using TSML, TSML.TSMLTransformers, TSML.Utils, TSML.TSMLTypes

hourlyagg = DateValgator(Dict(:dateinterval => Dates.Hour(1)))
```

```
halfhourlyagg = DateValgator(Dict(:dateinterval => Dates.Minute(30)))
dailyagg = DateValgator(Dict(:dateinterval => Dates.Day(1)))

fit!(halfhourlyagg,df)
halfhourlyres = transform!(halfhourlyagg,df)

fit!(hourlyagg,df)
hourlyres = transform!(hourlyagg,df)

fit!(dailyagg,df)
dailyres = transform!(dailyagg,df)
```

The first 5 rows of half-hourly, hourly, and daily aggregates:

```
julia> first(halfhourlyres,5)
5×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.102505
 2    2014-01-01T00:30:00  0.662658
 3    2014-01-01T01:00:00  0.172371
 4    2014-01-01T01:30:00  0.200161
 5    2014-01-01T02:00:00  0.272878

julia> first(hourlyres,5)
5×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.284211
 2    2014-01-01T01:00:00  0.269655
 3    2014-01-01T02:00:00  0.507651
 4    2014-01-01T03:00:00  0.542122
 5    2014-01-01T04:00:00  0.296328

julia> first(dailyres,5)
5×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.464811
 2    2014-01-02T00:00:00  0.563538
 3    2014-01-03T00:00:00  0.47755
 4    2014-01-04T00:00:00  0.460352
 5    2014-01-05T00:00:00  0.510602
```

Chapter 10

Imputation

10.1 Imputation

There are two ways to impute the `date,value` TS data. One uses `DateValNNer` which uses nearest neighbor and `DateValizer` which uses the dictionary of medians mapped to certain date-time interval grouping.

DateValNNer

`DateValNNer` expects the following arguments with their default values during instantation:

- `:dateinterval => Dates.Hour(1)`
  - grouping interval
- `:nnsize => 1`
  - size of neighborhood
- `:missdirection => :symmetric`
  - `:forward` vs `:backward` vs `:symmetric`
- `:strict => true`
  - whether or not to repeatedly iterate until no more missing data

The `:missdirection` indicates the imputation direction and the extent of neighborhood. Symmetric implies getting info from both sides of the missing data. `:forward` direction starts imputing from the top while the `:reverse` starts from the bottom. Please refer to Aggregators and Imputers for other examples.

Let's use the same dataset we have used in the tutorial and print the first few rows.

```
julia> first(X,10)
10×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  missing
 2    2014-01-01T00:15:00  missing
 3    2014-01-01T00:30:00  missing
```

```
 4    2014-01-01T00:45:00  missing
 5    2014-01-01T01:00:00  missing
 6    2014-01-01T01:15:00  missing
 7    2014-01-01T01:30:00  missing
 8    2014-01-01T01:45:00  0.0521332
 9    2014-01-01T02:00:00  0.26864
 10   2014-01-01T02:15:00  0.108871
```

Let's try the following setup grouping daily with `forward` imputation and 10 neighbors:

```
dnnr = DateValNNer(Dict(:dateinterval=>Dates.Hour(2),
             :nnsize=>10,:missdirection => :forward,
             :strict=>false))
fit!(dnnr,X)
forwardres=transform!(dnnr,X)
```

```
julia> first(forwardres,5)
5×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.491286
 2    2014-01-01T02:00:00  0.108871
 3    2014-01-01T04:00:00  0.361194
 4    2014-01-01T06:00:00  0.918165
 5    2014-01-01T08:00:00  0.690462
```

Same parameters as above but uses `reverse` instead of `forward` direction:

```
dnnr = DateValNNer(Dict(:dateinterval=>Dates.Hour(2),
             :nnsize=>10,:missdirection => :reverse,
             :strict=>false))
fit!(dnnr,X)
reverseres=transform!(dnnr,X)
```

```
julia> first(reverseres,5)
5×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.491286
 2    2014-01-01T02:00:00  0.108871
 3    2014-01-01T04:00:00  0.361194
 4    2014-01-01T06:00:00  0.918165
 5    2014-01-01T08:00:00  0.690462
```

Using `symmetric` imputation:

```
dnnr = DateValNNer(Dict(:dateinterval=>Dates.Hour(2),
             :nnsize=>10,:missdirection => :symmetric,
             :strict=>false))
fit!(dnnr,X)
symmetricres=transform!(dnnr,X)
```

```
julia> first(symmetricres,5)
5×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.491286
 2    2014-01-01T02:00:00  0.108871
 3    2014-01-01T04:00:00  0.361194
 4    2014-01-01T06:00:00  0.918165
 5    2014-01-01T08:00:00  0.690462
```

Unlike `symmetric` imputation that guarantees 100% imputation of missing data as long as the input has non-missing elements, `forward` and `reverse` cannot guarantee that the imputation replaces all missing data because of the boundary issues. If the top or bottom of the input is missing, the assymetric imputation will not be able to replace the endpoints that are missing. It is advised that to have successful imputation, `symmetric` imputation shall be used.

In the example above, the number of remaining missing data not imputed for `forward`, `reverse`, and `symmetric` is:

```
julia> sum(ismissing.(forwardres[:Value]))
3

julia> sum(ismissing.(reverseres[:Value]))
3

julia> sum(ismissing.(symmetricres[:Value]))
0
```

DateValizer

`DateValizer` operates on the principle that there is a reqularity of patterns in a specific time period such that replacing values is just a matter of extracting which time period it belongs and used the pooled median in that time period to replace the missing data. The default time period for `DateValizer` is hourly. In a more advanced implementation, we can add daily, hourly, and weekly periods but it will require much larger hash table. Additional grouping criteria can result into smaller subgroups which may contain 100% missing in some of these subgroups resulting to imputation failure. `DateValizer` only depends on the `:dateinterval => Dates.Hour(1)` argument with default value of hourly. Please refer to Aggregators and Imputers for more examples.

Let's try hourly, daily, and monthly median as the basis of imputation:

```
julia> hourlyzer = DateValizer(Dict(:dateinterval => Dates.Hour(1)))
TSML.TSMLTransformers.DateValizer(nothing, Dict{Symbol,Any}(:medians=>0×0 DataFrames.DataFrame
,:dateinterval=>1 hour))

julia> monthlyzer = DateValizer(Dict(:dateinterval => Dates.Month(1)))
TSML.TSMLTransformers.DateValizer(nothing, Dict{Symbol,Any}(:medians=>0×0 DataFrames.DataFrame
,:dateinterval=>1 month))

julia> dailyzer = DateValizer(Dict(:dateinterval => Dates.Day(1)))
TSML.TSMLTransformers.DateValizer(nothing, Dict{Symbol,Any}(:medians=>0×0 DataFrames.DataFrame
,:dateinterval=>1 day))

julia> fit!(hourlyzer,X)
Dict{Symbol,Any} with 2 entries:
  :medians      => 24×2 DataFrames.DataFrame…
  :dateinterval => 1 hour
```

```
julia> hourlyres = transform!(hourlyzer,X)
17521×2 DataFrames.DataFrame
 Row    Date                 Value
        Dates.DateTime       Float64

 1      2014-01-01T00:00:00  0.498827
 2      2014-01-01T01:00:00  0.500748
 3      2014-01-01T02:00:00  0.108871
 4      2014-01-01T03:00:00  0.473017
 5      2014-01-01T04:00:00  0.361194
 6      2014-01-01T05:00:00  0.582318
 7      2014-01-01T06:00:00  0.918165

 17514  2015-12-31T17:00:00  0.549606
 17515  2015-12-31T18:00:00  0.680491
 17516  2015-12-31T19:00:00  0.500731
 17517  2015-12-31T20:00:00  0.468921
 17518  2015-12-31T21:00:00  0.28438
 17519  2015-12-31T22:00:00  0.533108
 17520  2015-12-31T23:00:00  0.308998
 17521  2016-01-01T00:00:00  0.498827

julia> fit!(dailyzer,X)
Dict{Symbol,Any} with 2 entries:
  :medians      => 31×2 DataFrames.DataFrame…
  :dateinterval => 1 day

julia> dailyres = transform!(dailyzer,X)
731×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.48
 2    2014-01-02T00:00:00  0.628368
 3    2014-01-03T00:00:00  0.509263
 4    2014-01-04T00:00:00  0.559623
 5    2014-01-05T00:00:00  0.539073
 6    2014-01-06T00:00:00  0.387866
 7    2014-01-07T00:00:00  0.464466

 724  2015-12-25T00:00:00  0.44458
 725  2015-12-26T00:00:00  0.625784
 726  2015-12-27T00:00:00  0.659934
 727  2015-12-28T00:00:00  0.368161
 728  2015-12-29T00:00:00  0.506546
 729  2015-12-30T00:00:00  0.516895
 730  2015-12-31T00:00:00  0.299126
 731  2016-01-01T00:00:00  0.434787

julia> fit!(monthlyzer,X)
Dict{Symbol,Any} with 2 entries:
  :medians      => 12×2 DataFrames.DataFrame…
  :dateinterval => 1 month

julia> monthlyres = transform!(monthlyzer,X)
```

```
25×2 DataFrames.DataFrame
 Row  Date                 Value
      Dates.DateTime       Float64

 1    2014-01-01T00:00:00  0.525587
 2    2014-02-01T00:00:00  0.501297
 3    2014-03-01T00:00:00  0.540474
 4    2014-04-01T00:00:00  0.492871
 5    2014-05-01T00:00:00  0.514414
 6    2014-06-01T00:00:00  0.515317
 7    2014-07-01T00:00:00  0.501932

 18   2015-06-01T00:00:00  0.499711
 19   2015-07-01T00:00:00  0.509305
 20   2015-08-01T00:00:00  0.505218
 21   2015-09-01T00:00:00  0.511359
 22   2015-10-01T00:00:00  0.504835
 23   2015-11-01T00:00:00  0.487876
 24   2015-12-01T00:00:00  0.512668
 25   2016-01-01T00:00:00  0.482073
```

Part IV

ML Library

Chapter 11

Decision Tree

11.1 DecisionTreeLearners

Creates an API wrapper for DecisionTrees for pipeline workflow.

Index

- `TSML.DecisionTreeLearners.Adaboost`
- `TSML.DecisionTreeLearners.PrunedTree`
- `TSML.DecisionTreeLearners.RandomForest`

AutoDocs

`TSML.DecisionTreeLearners.Adaboost` – Type.

```
Adaboost(
  Dict(
    :output => :class,
    :num_iterations => 7
  )
)
```

Adaboosted decision tree stumps. See DecisionTree.jl's documentation

Hyperparameters:

- `:num_iterations` => 7 (number of iterations of AdaBoost)

Implements `fit!`, `transform!`

source

`TSML.DecisionTreeLearners.PrunedTree` – Type.

```
PrunedTree(
  Dict(
    :purity_threshold => 1.0,
    :max_depth => -1,
    :min_samples_leaf => 1,
    :min_samples_split => 2,
```

```
    :min_purity_increase => 0.0
  )
)
```

Decision tree classifier. See DecisionTree.jl's documentation

Hyperparameters:

- `:purity_threshold` => 1.0 (merge leaves having >=thresh combined purity)
- `:max_depth` => -1 (maximum depth of the decision tree)
- `:min_samples_leaf` => 1 (the minimum number of samples each leaf needs to have)
- `:min_samples_split` => 2 (the minimum number of samples in needed for a split)
- `:min_purity_increase` => 0.0 (minimum purity needed for a split)

Implements `fit!`, `transform!`

source

`TSML.DecisionTreeLearners.RandomForest` – Type.

```
RandomForest(
  Dict(
    :output => :class,
    :num_subfeatures => 0,
    :num_trees => 10,
    :partial_sampling => 0.7,
    :max_depth => -1
  )
)
```

Random forest classification. See DecisionTree.jl's documentation

Hyperparameters:

- `:num_subfeatures` => 0 (number of features to consider at random per split)
- `:num_trees` => 10 (number of trees to train)
- `:partial_sampling` => 0.7 (fraction of samples to train each tree on)
- `:max_depth` => -1 (maximum depth of the decision trees)
- `:min_samples_leaf` => 1 (the minimum number of samples each leaf needs to have)
- `:min_samples_split` => 2 (the minimum number of samples in needed for a split)
- `:min_purity_increase` => 0.0 (minimum purity needed for a split)

Implements `fit!`, `transform!`

source

`TSML.TSMLTypes.fit!` – Method.

```
fit!(adaboost::Adaboost, features::T, labels::Vector) where
↪  {T<:Union{Vector,Matrix,DataFrame}}
```

Function to optimize the hyperparameters of `Adaboost` instance.

source

`TSML.TSMLTypes.fit!` – Method.

```
fit!(tree::PrunedTree, features::T, labels::Vector) where {T<:Union{Vector,Matrix,DataFrame}}
```

Function to optimize the hyperparameters of `PrunedTree` instance.

source

`TSML.TSMLTypes.fit!` – Method.

```
fit!(forest::RandomForest, features::T, labels::Vector) where
↪  {T<:Union{Vector,Matrix,DataFrame}}
```

Function to optimize the parameters of the `RandomForest` instance.

source

`TSML.TSMLTypes.transform!` – Method.

```
transform!(adaboost::Adaboost, features::T) where {T<:Union{Vector,Matrix,DataFrame}}
```

Function to predict using the optimized hyperparameters of the trained `Adaboost` instance.

source

`TSML.TSMLTypes.transform!` – Method.

```
transform!(tree::PrundTree, features::T) where {T<:Union{Vector,Matrix,DataFrame}}
```

Function to predict using the optimized hyperparameters of the trained `PrunedTree` instance.

source

`TSML.TSMLTypes.transform!` – Method.

```
transform!(forest::RandomForest, features::T) where {T<:Union{Vector,Matrix,DataFrame}}
```

Function to predict using the optimized hyperparameters of the trained `RandomForest` instance.

source

Chapter 12

Types and Functions

12.1 Types and Functions

Creates an API wrapper for DecisionTrees for pipeline workflow.

Index

AutoDocs

`TSML.Outliernicers.Outliernicer` – Type.

```
Outliernicer(Dict())
```

Detects outliers below or above (q25-iqr,q75+iqr) and replace them with missing so that ValNNer can use nearest neighbors to replace the missings.

source

`TSML.Plotters.Plotter` – Type.

```
Plotter()
```

Plots a TS by default but performs interactive plotting if specified during instance creation.

source

`TSML.TSMLTypes.transform!` – Method.

Convert missing into NaN for plotting discontinuity

source