
hts Documentation

Release 0.5.12

Carlo Mazzaferro

May 12, 2023

Contents:

1	scikit-hts	3
1.1	Overview	3
1.2	Features	3
1.3	Examples	4
1.4	Roadmap	4
1.5	Credits	4
2	Installation	5
2.1	From PyPi	5
2.2	With optional dependencies	5
2.3	From sources	6
3	Usage	7
3.1	Typical Usage	7
3.2	Ground Up Example	8
3.3	Reconcile Pre-Computed Forecasts	10
4	Hierarchical Representation	13
4.1	Hierarchical Structure	13
4.2	Grouped Structure	15
5	Supported Models	21
5.1	Models	21
6	Geo Handling Capabilities	23
7	API Index	25
7.1	hts	25
7.2	hts.convenience	28
7.3	hts.defaults	28
7.4	hts.functions	28
7.5	hts.revision	32
7.6	hts.transforms	32
7.7	hts._t	33
8	Parallelization	35
8.1	Parallelization of Model Fitting	35

8.2	Parallelization of Forecasting	35
9	How to deploy scikit-hts at scale	37
9.1	The distributor class	37
9.2	Using dask to distribute the calculations	38
9.3	Writing your own distributor	39
9.4	Acknowledgement	40
10	Contributing	41
10.1	Types of Contributions	41
10.2	Get Started!	42
10.3	Pull Request Guidelines	43
10.4	Tips	43
10.5	Deploying	43
11	History	45
11.1	0.1.0 (2020-01-02)	45
11.2	0.2.0 (2018-02-13)	45
11.3	0.2.3 (2020-03-28)	45
11.4	0.3.0 (2020-03-28)	45
11.5	0.4.0 (2020-03-28)	45
11.6	0.4.1 (2020-03-28)	46
11.7	0.5.2 (2020-03-28)	46
11.8	0.5.3 (2021-02-23)	46
11.9	0.5.4 (2021-04-20)	46
11.10	0.5.6 (2021-04-20)	46
11.11	0.5.7 (2021-05-30)	46
11.12	0.5.8 (2021-05-30)	46
11.13	0.5.9 (2021-05-30)	46
11.14	0.5.10 (2021-06-5)	47
11.15	0.5.11 (2021-06-5)	47
12	Indices and tables	49
	Python Module Index	51
	Index	53

NOTE: I unfortunately do not have time anymore to dedicate to this project, contributions are welcome.

CHAPTER 1

scikit-hts

Hierarchical Time Series with a familiar API. This is the result from not having found any good implementations of HTS on-line, and my work in the mobility space while working at Circ (acquired by Bird scooters).

My work on this is purely out of passion, so contributions are always welcomed. You can also buy me a coffee if you'd like:

ETH / BSC Address: 0xbF42b9c8F7B69D52b8b986AA4E0BAc6838Af6698

Documentation: <https://scikit-hts.readthedocs.io/en/latest/>

1.1 Overview

Building on the excellent work by Hyndman¹, we developed this package in order to provide a python implementation of general hierarchical time series modeling.

Note: STATUS: alpha. Active development, but breaking changes may come.

1.2 Features

- Supported and tested on python 3.6, python 3.7 and python 3.8

¹ Forecasting Principles and Practice. Rob J Hyndman and George Athanasopoulos. Monash University, Australia.

- Implementation of Bottom-Up, Top-Down, Middle-Out, Forecast Proportions, Average Historic Proportions, Proportions of Historic Averages and OLS revision methods
- Support for representations of hierarchical and grouped time series
- Support for a variety of underlying forecasting models, including: SARIMAX, ARIMA, Prophet, Holt-Winters
- Scikit-learn-like API
- Geo events handling functionality for geospatial data, including visualisation capabilities
- Static typing for a nice developer experience
- Distributed training & Dask integration: perform training and prediction in parallel or in a cluster with Dask

1.3 Examples

You can find code usages here: <https://github.com/carlomazzaferro/scikit-hts-examples>

1.4 Roadmap

- **More flexible underlying modeling support**
 - [P] AR, ARIMAX, VARMAX, etc
 - [P] Bring-Your-Own-Model
 - [P] Different parameters for each of the models
- **Decoupling reconciliation methods from forecast fitting**
 - [W] Enable to use the reconciliation methods with pre-fitted models

P: Planned

W: WIP

1.5 Credits

This package was created with [Cookiecutter](#) and the [audreyr/cookiecutter-pypackage](#) project template.

2.1 From PyPi

```
$ pip install scikit-hts
```

2.2 With optional dependencies

2.2.1 Geo Utilities

This allows the usage of `scikit-hts`'s geo handling capabilities. See more: [Geo Handling Capabilities](#).

See more at

```
$ pip install scikit-hts[geo]
```

2.2.2 Facebook's Prophet Support

This allows to train models using Facebook's [Prophet](#)

```
$ pip install scikit-hts[prophet]
```

2.2.3 Auto-Arima

This allows to train models using Alkaline-ml's excellent [auto arima implementation](#)

```
$ pip install scikit-hts[auto-arima]
```

2.2.4 Distributed Training

This allows to run distributed training with a local or remote Dask cluster

```
$ pip install scikit-hts[distributed]
```

2.2.5 Everything

Install's all optional dependencies

```
$ pip install scikit-hts[all]
```

2.3 From sources

The sources for scikit-hts can be downloaded from the [Github repo](#).

You can either clone the public repository:

```
$ git clone git://github.com/carlomazzaferro/scikit-hts
```

Or download the [tarball](#):

```
$ curl -OL https://github.com/carlomazzaferro/scikit-hts/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

3.1 Typical Usage

`scikit-hts` has one main class that provides the interface with your desired forecasting methodology and reconciliation strategy. Here you can find how to get started quickly with `scikit-hts`. We'll use some sample (fake) data.

```
>>> from datetime import datetime
>>> from hts import HTSRegressor
>>> from hts.utilities.load_data import load_hierarchical_sine_data

# load some data
>>> s, e = datetime(2019, 1, 15), datetime(2019, 10, 15)
>>> hsd = load_hierarchical_sine_data(s, e).resample('1H').apply(sum)
>>> hier = {'total': ['a', 'b', 'c'],
            'a': ['a_x', 'a_y'],
            'b': ['b_x', 'b_y'],
            'c': ['c_x', 'c_y'],
            'a_x': ['a_x_1', 'a_x_2'],
            'a_y': ['a_y_1', 'a_y_2'],
            'b_x': ['b_x_1', 'b_x_2'],
            'b_y': ['b_y_1', 'b_y_2'],
            'c_x': ['c_x_1', 'c_x_2'],
            'c_y': ['c_y_1', 'c_y_2']
            }

>>> hsd.head()
```

			total	a	b	c	d	aa	
	ab	...	ba	bb	bc	ca	cb	cc	cd
2019-01-15 00:00:00	11.934729	0.638735	3.436469	5.195530	2.663996	0.218140	0.		
↪ 420594	...	1.449734	1.727512	0.259222	0.593310	1.251554	2.217371	1.133295	
2019-01-15 01:00:00	8.698295	2.005391	2.687024	1.740504	2.265375	0.254958	1.		
↪ 750433	...	1.963620	0.390856	0.332549	0.566592	0.197838	0.547443	0.428632	

(continues on next page)

(continued from previous page)

```

2019-01-15 02:00:00 12.093040 3.802658 2.204833 2.933652 3.151896 3.185786 0.
↳616872 ... 0.110134 1.885216 0.209483 1.332533 0.301493 1.294185 0.005441
2019-01-15 03:00:00 14.365129 4.332290 3.234713 0.780173 6.017954 3.993601 0.
↳338689 ... 0.846830 0.777724 1.610158 0.091538 0.505417 0.079388 0.103830
2019-01-15 04:00:00 1.030305 2.073372 0.649284 -1.536231 -0.156119 -0.184177 2.
↳257549 ... 0.433048 -0.179693 0.395928 -0.667796 0.112877 -0.050382 -0.930930

>>> reg = HTSRegressor(model='prophet', revision_method='OLS')
>>> reg = reg.fit(df=hsd, nodes=hier)
>>> preds = reg.predict(steps_ahead=10)

```

More extensive usage, including a solution for Kaggle’s [M5 Competition](#), can be found in the [scikit-hts-examples](#) repo.

3.2 Ground Up Example

Here’s a ground up walk through of taking raw data, making custom forecasts, and reconciling them using the example from [FPP](#).

This small block creates the raw data. We assume a good number of users begin with tabular data coming from database.

```

>>> import hts.functions
>>> import pandas
>>> import collections

>>> hier_df = pandas.DataFrame(
    data={
        'ds': ['2020-01', '2020-02'] * 5,
        'lev1': ['A', 'A',
                 'A', 'A',
                 'A', 'A',
                 'B', 'B',
                 'B', 'B'],
        'lev2': ['X', 'X',
                 'Y', 'Y',
                 'Z', 'Z',
                 'X', 'X',
                 'Y', 'Y'],
        'val': [1, 2,
                 3, 4,
                 5, 6,
                 7, 8,
                 9, 10]
    })
>>> hier_df

```

	ds	lev1	lev2	val
0	2020-01	A	X	1
1	2020-02	A	X	2
2	2020-01	A	Y	3
3	2020-02	A	Y	4
4	2020-01	A	Z	5

(continues on next page)

(continued from previous page)

5	2020-02	A	Z	6
6	2020-01	B	X	7
7	2020-02	B	X	8
8	2020-01	B	Y	9
9	2020-02	B	Y	10

Specify a hierarchy of your choosing. Where the `level_names` argument is a list of column names that represent levels in the hierarchy. The `hierarchy` argument consists of a list of lists, where you can specify what levels in your hierarchy to include in the hierarchy structure. You do not need to specify the bottom level of your hierarchy in the `hierarchy` argument. This is already included, since it is equivalent to `level_names` aggregation level.

Through the `hts.function.get_hierarchichal_df` function you will get a wide `pandas.DataFrame` with the individual time series for you to create forecasts.

```
>>> level_names = ['lev1', 'lev2']
>>> hierarchy = [['lev1'], ['lev2']]
>>> wide_df, sum_mat, sum_mat_labels = hts.functions.get_hierarchichal_df(hier_df,
                                                                           level_
↳names=level_names,
↳hierarchy=hierarchy,
↳colname='ds',
↳'val')
>>> wide_df
  lev1_lev2  A_X  A_Y  A_Z  B_X  B_Y  total  A  B  X  Y  Z
ds
2020-01      1   3   5   7   9    25   9  16  8  12  5
2020-02      2   4   6   8  10    30  12  18 10  14  6
```

Here's an example showing how to easily change your hierarchy, without changing your underlying data. We do not want to save these results for the sake of following parts of the example.

```
>>> hierarchy = [['lev1']]
>>> a, b, c = hts.functions.get_hierarchichal_df(hier_df,
                                                  level_names=level_names,
                                                  hierarchy=hierarchy,
                                                  date_colname='ds',
                                                  val_colname='val')
>>> a
  lev1_lev2  A_X  A_Y  A_Z  B_X  B_Y  total  A  B
ds
2020-01      1   3   5   7   9    25   9  16
2020-02      2   4   6   8  10    30  12  18
```

Create your forecasts and store them in a new `DataFrame` with the same format. Here we just do an average, but you can get as complex as you'd like.

```
# Create a DataFrame to store new forecasts in
>>> forecasts = pandas.DataFrame(index=['2020-03'], columns=wide_df.columns)

>>> import statistics
>>> for col in wide_df.columns:
    forecasts[col] = statistics.mean(wide_df[col])
```

(continues on next page)

(continued from previous page)

```
>>> forecasts
lev1_lev2  A_X  A_Y  A_Z  B_X  B_Y  total    A    B  X  Y  Z
2020-03    1.5  3.5  5.5  7.5  9.5   27.5  10.5  17  9  13  5.5
```

Store your forecasts in a dictionary to be passed to the reconciliation algorithm.

```
>>> pred_dict = collections.OrderedDict()

# Add predictions to dictionary in same order as summing matrix
>>> for label in sum_mat_labels:
    pred_dict[label] = pandas.DataFrame(data=forecasts[label].values, columns=['yhat
    ↪'])
```

Reconcile your forecasts. Here we use OLS optimal reconciliation. The, put reconciled forecasts in the same wide DataFrame format.

You'll notice the forecasts are the. Because we used an average to forecast, the forecasts were already coherent. Therefore, they remain the same/ coherent post-reconciliation. Demonstrating that the reconciliation is working.

```
>>> revised = hts.functions.optimal_combination(pred_dict, sum_mat, method='OLS', mse=
    ↪ {})

>>> revised_forecasts = pandas.DataFrame(data=revised[0:,0:],
                                         index=forecasts.index,
                                         columns=sum_mat_labels)

>>> revised_forecasts
      total    Z    Y    X    B    A  A_X  A_Y  A_Z  B_X  B_Y
2020-03  27.5  5.5  13.0  9.0  17.0  10.5  1.5  3.5  5.5  7.5  9.5
```

3.3 Reconcile Pre-Computed Forecasts

This is an example of creating forecasts outside of scikit-hts and then utilizing scikit-hts to do OLS optimal reconciliation on the forecasts.

```
>>> from datetime import datetime
>>> import hts
>>> from hts.utilities.load_data import load_hierarchical_sine_data
>>> import statsmodels
>>> import collections
>>> import pandas as pd

>>> s, e = datetime(2019, 1, 15), datetime(2019, 10, 15)
>>> hsd = load_hierarchical_sine_data(start=s, end=e, n=10000)
>>> hier = {'total': ['a', 'b', 'c'],
           'a': ['a_x', 'a_y'],
           'b': ['b_x', 'b_y'],
           'c': ['c_x', 'c_y'],
           'a_x': ['a_x_1', 'a_x_2'],
           'a_y': ['a_y_1', 'a_y_2'],
           'b_x': ['b_x_1', 'b_x_2'],
           'b_y': ['b_y_1', 'b_y_2'],
           'c_x': ['c_x_1', 'c_x_2'],
```

(continues on next page)

(continued from previous page)

```

        'c_y': ['c_y_1', 'c_y_2']
    }

>>> tree = hts.hierarchy.HierarchyTree.from_nodes(hier, hsd)
>>> sum_mat, sum_mat_labels = hts.functions.to_sum_mat(tree)

>>> forecasts = pd.DataFrame(columns=hsd.columns, index=['fake'])

    # Make forecasts made outside of package. Could be any modeling technique.
>>> for col in hsd.columns:
    model = statsmodels.tsa.holtwinters.SimpleExpSmoothing(hsd[col].values).fit()
    fcst = list(model.forecast(1))
    forecasts[col] = fcst

>>> pred_dict = collections.OrderedDict()

# Add predictions to dictionary is same order as summing matrix
>>> for label in sum_mat_labels:
    pred_dict[label] = pd.DataFrame(data=forecasts[label].values, columns=['yhat
↪'])

>>> revised = hts.functions.optimal_combination(pred_dict, sum_mat, method='OLS', mse=
↪{})

# Put reconciled forecasts in nice DataFrame form
>>> revised_forecasts = pd.DataFrame(data=revised[0:,0:],
                                     index=forecasts.index,
                                     columns=sum_mat_labels)

```

Hierarchical Representation

`scikit-hts`'s core data structure is the `HierarchyTree`. At its core, it is simply an **N-Ary Tree**, a recursive data structure where each node is specified by:

- A human readable key, such as 'germany', 'total', 'berlin', or '881f15ad61ffff'
- Keys should be unique and delimited by underscores. Therefore, using the example below there should not be duplicate values across level 1, 2 or 3. For example, a should not also a value in level 2.
- An item, represented by a `pandas.Series` (or `pandas.DataFrame` for multivariate inputs), which contains the actual data about that node

4.1 Hierarchical Structure

For instance, a tree with nodes and levels as follows:

- Level 1: a, b, c
- Level 2: x, y
- Level 3: 1, 2

```
nodes = {'total': ['a', 'b', 'c'],
         'a': ['a_x', 'a_y'],
         'b': ['b_x', 'b_y'],
         'c': ['c_x', 'c_y'],
         'a_x': ['a_x_1', 'a_x_2'],
         'a_y': ['a_y_1', 'a_y_2'],
         'b_x': ['b_x_1', 'b_x_2'],
         'b_y': ['b_y_1', 'b_y_2'],
         'c_x': ['c_x_1', 'c_x_2'],
         'c_y': ['c_y_1', 'c_y_2']}
}
```

Represents the following structure:

Level	# of nodes		Node Key		
↪					
1			t		
↪	1				
2	a		b		c
↪	3				
3	a_x	a_y	b_x	b_y	c_x c_
↪y	6				
4	a_x_1 a_x_2	a_y_1 a_y_2	b_x_1 b_x_2	b_y_1 b_y_2	c_x_1 c_x_2 c_
↪y_1 c_y_2	12				

To get a sense of how the hierarchy trees are implemented, some sample data can be loaded:

```
>>> from datetime import datetime
>>> from hts.hierarchy import HierarchyTree
>>> from hts.utilities.load_data import load_hierarchical_sine_data

>>> s, e = datetime(2019, 1, 15), datetime(2019, 10, 15)
>>> hsd = load_hierarchical_sine_data(start=s, end=e, n=10000)
>>> print(hsd.head())
```

	b_x	b_y	total	a	b	c	a_x	a_
↪y			c_x ...	a_y_2	b_x_1	b_x_2	b_y_1	b_
↪y_2	c_x_1	c_x_2	c_y_1 c_y_2					
2019-01-15 01:11:09.255573	2.695133	0.150805	0.031629	2.512698	0.037016	0.		
↪113789	0.028399	0.003231	0.268406 ...	0.080803	0.013131	0.015268	0.000952	
↪0.002279	0.175671	0.092734	0.282259	1.962034				
2019-01-15 01:18:30.753096	-3.274595	-0.199276	-1.624369	-1.450950	-0.117717	-0.		
↪081559	-0.300076	-1.324294	-1.340172 ...	-0.077289	-0.177000	-0.123075	-0.178258	-
↪1.146035	-0.266198	-1.073975	-0.083517	-0.027260				
2019-01-15 01:57:48.607109	-1.898038	-0.226974	-0.662317	-1.008747	-0.221508	-0.		
↪005466	-0.587826	-0.074492	-0.929464 ...	-0.003297	-0.218128	-0.369698	-0.021156	-
↪0.053335	-0.225994	-0.703470	-0.077021	-0.002262				
2019-01-15 02:06:57.994575	13.904908	6.025506	5.414178	2.465225	5.012228	1.		
↪013278	4.189432	1.224746	1.546544 ...	0.467630	1.297829	2.891602	0.671085	
↪0.553661	0.066278	1.480266	0.769954	0.148728				
2019-01-15 02:14:22.367818	11.028013	3.537919	6.504104	0.985990	2.935614	0.		
↪602305	4.503611	2.000493	0.179114 ...	0.091993	4.350293	0.153318	1.349629	
↪0.650864	0.066946	0.112168	0.473987	0.332889				

```
>>> hier = {'total': ['a', 'b', 'c'],
            'a': ['a_x', 'a_y'],
            'b': ['b_x', 'b_y'],
            'c': ['c_x', 'c_y'],
            'a_x': ['a_x_1', 'a_x_2'],
            'a_y': ['a_y_1', 'a_y_2'],
            'b_x': ['b_x_1', 'b_x_2'],
            'b_y': ['b_y_1', 'b_y_2'],
            'c_x': ['c_x_1', 'c_x_2'],
            'c_y': ['c_y_1', 'c_y_2']}

>>> tree = HierarchyTree.from_nodes(hier, hsd, root='total')
>>> print(tree)
```

(continues on next page)

(continued from previous page)

```

- total
  |- a
  |   |- a_x
  |   |   |- a_x_1
  |   |   - a_x_2
  |   - a_y
  |       |- a_y_1
  |       - a_y_2
  |- b
  |   |- b_x
  |   |   |- b_x_1
  |   |   - b_x_2
  |   - b_y
  |       |- b_y_1
  |       - b_y_2
- c
  |- c_x
  |   |- c_x_1
  |   - c_x_2
  - c_y
    |- c_y_1
    - c_y_2

```

4.2 Grouped Structure

In order to create a grouped structure, instead of a strictly hierarchichal structure you must specify all levels within the grouping strucure dictionary and dataframe as seen below.

Levels in example:

- Level 1: A, B
- Level 2: X, Y

```

import hts
import pandas as pd

>>> hierarchy = {
    "total": ["A", "B", "X", "Y"],
    "A": ["A_X", "A_Y"],
    "B": ["B_X", "B_Y"],
}

>>> grouped_df = pd.DataFrame(
    data={
        "total": [],
        "A": [],
        "B": [],
        "X": [],
        "Y": [],
        "A_X": [],
        "A_Y": [],
        "B_X": [],
        "B_Y": [],
    }

```

(continues on next page)

(continued from previous page)

```

)

>>> tree = hts.hierarchy.HierarchyTree.from_nodes(hierarchy, grouped_df)
>>> sum_mat, sum_mat_labels = hts.functions.to_sum_mat(tree)
>>> print(sum_mat)  # Commented labels will not appear in the printout, they are here,
↳as an example.
[[1. 1. 1. 1.] # totals
 [0. 1. 0. 1.] # Y
 [1. 0. 1. 0.] # X
 [0. 0. 1. 1.] # B
 [1. 1. 0. 0.] # A
 [1. 0. 0. 0.] # A_X
 [0. 1. 0. 0.] # A_Y
 [0. 0. 1. 0.] # B_X
 [0. 0. 0. 1.]] # B_Y

>>> print(sum_mat_labels)  # Use this if you need to match summing matrix rows with
↳labels.
['total', 'Y', 'X', 'B', 'A', 'A_X', 'A_Y', 'B_X', 'B_Y']

```

class `hts.hierarchy.HierarchyTree` (*key: str = None, item: Union[pandas.core.series.Series, pandas.core.frame.DataFrame] = None, exogenous: List[str] = None, children: List[hts.t.NAryTreeT] = None, parent: hts.t.NAryTreeT = None*)

A generic N-ary tree implementations, that uses a list to store it's children.

classmethod `from_geo_events` (*df: pandas.core.frame.DataFrame, lat_col: str, lon_col: str, nodes: Tuple, levels: Tuple[int, int] = (6, 7), resample_freq: str = '1H', min_count: Union[float, int] = 0.2, root_name: str = 'total', fillna: bool = False*)

Parameters

- **df** (*pandas.DataFrame*) –
- **lat_col** (*str*) – Column where the latitude coordinates can be found
- **lon_col** (*str*) – Column where the longitude coordinates can be found
- **nodes** (*str*) –
- **levels** –
- **resample_freq** –
- **min_count** –
- **root_name** –
- **fillna** –

Returns

Return type *HierarchyTree*

classmethod `from_nodes` (*nodes: Dict[str, List[str]], df: pandas.core.frame.DataFrame, exogenous: Dict[str, List[str]] = None, root: Union[str, HierarchyTree] = 'total', top: Optional[hts.hierarchy.HierarchyTree] = None, stack: List[T] = None*)

Standard method for creating a hierarchy from nodes and a dataframe containing as columns those nodes. The nodes are represented as a dictionary containing as keys the nodes, and as values list of edges. See the examples for usage. The total column must be named total and not something else.

Parameters

- **nodes** (*NodesT*) – Nodes definition. See Examples.
- **df** (*pandas.DataFrame*) – The actual data containing the nodes
- **exogenous** (*ExogT*) – The nodes representing the exogenous variables
- **root** (*Union[str, HierarchyTree]*) – The name of the root node
- **top** (*HierarchyTree*) – Not to be used for initialisation, only in recursive calls
- **stack** (*list*) – Not to be used for initialisation, only in recursive calls

Returns *hierarchy* – The hierarchy tree representation of your data

Return type *HierarchyTree*

Examples

In this example we will create a tree from some multivariate data

```
>>> from hts.utilities.load_data import load_mobility_data
>>> from hts.hierarchy import HierarchyTree
```

```
>>> hmv = load_mobility_data()
>>> hmv.head()
      WF-01  CH-07  BT-01  CBD-13  SLU-15  CH-02  CH-08  SLU-01  BT-03
→ CH-05  SLU-19  SLU-07  SLU-02  CH-01  total  CH  SLU  BT  OTHER  temp
→precipitation
starttime
2014-10-13      16      14      20      16      20      42      24      24      12
→      22      14      2      8      6      240  108      68      32      32  62.0
→      0.00
2014-10-14      22      28      28      38      36      36      42      40      14
→      26      18      32      16      18      394  150     142     42      60  59.0
→      0.11
2014-10-15      10      14      8      20      18      38      16      28      18
→      10      0      24      10      16      230   94      80     26      30  58.0
→      0.45
2014-10-16      22      18      24      44      44      40      24      20      22
→      18      8      26      14      14      338  114     112     46      66  61.0
→      0.00
2014-10-17      8      12      16      20      18      22      32      12      8
→      28      10      30      8      10      234  104      78     24      28  60.0
→      0.14
```

```
>>> hier = {
    'total': ['CH', 'SLU', 'BT', 'OTHER'],
    'CH': ['CH-07', 'CH-02', 'CH-08', 'CH-05', 'CH-01'],
    'SLU': ['SLU-15', 'SLU-01', 'SLU-19', 'SLU-07', 'SLU-02'],
    'BT': ['BT-01', 'BT-03'],
    'OTHER': ['WF-01', 'CBD-13']
}
>>> exogenous = {k: ['precipitation', 'temp'] for k in hmv.columns if k not
→in ['precipitation', 'temp']}
>>> ht = HierarchyTree.from_nodes(hier, hmv, exogenous=exogenous)
>>> print(ht)
- total
  |- CH
```

(continues on next page)

(continued from previous page)

```
|   |- CH-07
|   |- CH-02
|   |- CH-08
|   |- CH-05
|   - CH-01
|- SLU
|   |- SLU-15
|   |- SLU-01
|   |- SLU-19
|   |- SLU-07
|   - SLU-02
|- BT
|   |- BT-01
|   - BT-03
- OTHER
  |- WF-01
  - CBD-13
```

get_level_order_labels () → List[List[str]]

Get the associated node labels from the NAryTreeT level_order_traversal().

Parameters **self** (NAryTreeT) – Tree being searched.

Returns Node labels corresponding to level order traversal.

Return type List[List[str]]

get_node (key: str) → Optional[hts._t.NAryTreeT]

Get a node given its key

Parameters **key** (str) – The key of the node of interest

Returns **node** – The node of interest

Return type *HierarchyTree*

is_leaf ()

Check if node is a leaf Node

Returns True or False

Return type bool

level_order_traversal () → List[List[int]]

Iterate through the tree in level order, getting the number of children for each node

Returns

Return type list[list[int]]

num_nodes () → int

Return the of nodes in the tree

Returns num nodes

Return type int

to_pandas () → pandas.core.frame.DataFrame

Transforms the hierarchy into a pandas.DataFrame :returns: **df** – Dataframe representation of the tree
:rtype: pandas.DataFrame

traversal_level () → List[hts._t.NAryTreeT]
Level order traversal of the tree

Returns

Return type list of nodes

Supported Models

Scikit-hts extends the work done by Hyndman in a few ways. One of the most important ones is the ability to use a variety of different underlying modeling techniques to predict the base forecasts.

We have implemented so far 4 kinds of underlying models:

1. [Auto-Arima](#), thanks to the excellent implementation provided by the folks at [alkaline-ml](#)
2. [SARIMAX](#), implemented by the [statsmodels](#) package
3. [Holt-Winters](#) exponential smoothing, also implemented in [statsmodels](#)
4. [Facebook's Prophet](#)

The full feature set of the underlying models is supported, including exogenous variables handling. Upon instantiation, use keyword arguments to pass the the arguments you need to the underlying model instantiation, fitting, and prediction.

Note: The main development focus is adding more support underlying models. Stay tuned, or feel free to check out the [Contributing](#) guide.

5.1 Models

```
class hts.model.AutoArimaModel (node: hts.hierarchy.HierarchyTree, **kwargs)
    Wrapper class around pmdarima.AutoARIMA
```

Variables

- **model** (*pmdarima.AutoARIMA*) – The instance of the model
- **mse** (*float*) – MSE for in-sample predictions
- **residual** (*numpy.ndarray*) – Residuals for the in-sample predictions
- **forecast** (*pandas.DataFrame*) – The forecast for the trained model

fit (*self*, ***fit_args*)
 Fits underlying models to the data, passes kwargs to `AutoARIMA`

predict (*self*, *node*, *steps_ahead*: *int* = 10, *alpha*: *float* = 0.05)
 Predicts the n-step ahead forecast. Exogenous variables are required if models were fit using them

class `hts.model.SarimaxModel` (*node*: `hts.hierarchy.HierarchyTree`, ***kwargs*)
 Wrapper class around `statsmodels.tsa.statespace.sarimax.SARIMAX`

Variables

- **model** (*SARIMAX*) – The instance of the model
- **mse** (*float*) – MSE for in-sample predictions
- **residual** (*numpy.ndarray*) – Residuals for the in-sample predictions
- **forecast** (*pandas.DataFrame*) – The forecast for the trained model

fit (*self*, ***fit_args*)
 Fits underlying models to the data, passes kwargs to `SARIMAX`

predict (*self*, *node*, *steps_ahead*: *int* = 10, *alpha*: *float* = 0.05)
 Predicts the n-step ahead forecast. Exogenous variables are required if models were fit using them

class `hts.model.HoltWintersModel` (*node*: `hts.hierarchy.HierarchyTree`, ***kwargs*)
 Wrapper class around `statsmodels.tsa.holtwinters.ExponentialSmoothing`

Variables

- **model** (*ExponentialSmoothing*) – The instance of the model
- **_model** (*HoltWintersResults*) – The result of model fitting. See `statsmodels.tsa.holtwinters.HoltWintersResults`
- **mse** (*float*) – MSE for in-sample predictions
- **residual** (*numpy.ndarray*) – Residuals for the in-sample predictions
- **forecast** (*pandas.DataFrame*) – The forecast for the trained model

fit (*self*, ***fit_args*)
 Fits underlying models to the data, passes kwargs to `SARIMAX`

predict (*self*, *node*, *steps_ahead*: *int* = 10)
 Predicts the n-step ahead forecast

class `hts.model.FBProphetModel` (*node*: `hts.hierarchy.HierarchyTree`, ***kwargs*)
 Wrapper class around `fbprophet.Prophet`

Variables

- **model** (*Prophet*) – The instance of the model
- **mse** (*float*) – MSE for in-sample predictions
- **residual** (*numpy.ndarray*) – Residuals for the in-sample predictions
- **forecast** (*pandas.DataFrame*) – The forecast for the trained model

fit (*self*, ***fit_args*)
 Fits underlying models to the data, passes kwargs to `fbprophet.Prophet`

predict (*self*, *node*, *steps_ahead*: *int* = 10, *freq*: *str* = 'D', ***predict_args*)
 Predicts the n-step ahead forecast. Exogenous variables are required if models were fit using them, frequency should be passed as well

CHAPTER 6

Geo Handling Capabilities

For a complete treatment, please visit the [geo notebook](#).

7.1 hts

```
class hts.HTSRegressor(model: str = 'prophet', revision_method: str = 'OLS', transform:  

Union[hts.t.Transform, bool, None] = False, n_jobs: int = 1, low_memory:  

bool = False, **kwargs)  

Bases: sklearn.base.BaseEstimator, sklearn.base.RegressorMixin
```

Main regressor class for scikit-hts. Likely the only import you'll need for using this project. It takes a pandas dataframe, the nodes specifying the hierarchies, model kind, revision method, and a few other parameters. See Examples to get an idea of how to use it.

Variables

- **transform** (*Union[NamedTuple[str, Callable], bool]*) – Function transform to be applied to input and outputs. If True, it will use `scipy.stats.boxcox` and `scipy.special._ufuncs.inv_boxcox` on input and output data
- **sum_mat** (*array_like*) – The summing matrix, explained in depth in [Forecasting](#)
- **nodes** (*Dict[str, List[str]]*) – Nodes representing node, edges of the hierarchy. Keys are nodes, values are list of edges.
- **df** (*pandas.DataFrame*) – The dataframe containing the nodes and edges specified above
- **revision_method** (*str*) – One of: "OLS", "WLSS", "WLSV", "FP", "PHA", "AHP", "BU", "NONE"
- **models** (*dict*) – Dictionary that holds the trained models
- **mse** (*dict*) – Dictionary that holds the mse scores for the trained models
- **residuals** (*dict*) – Dictionary that holds the mse residual for the trained models
- **forecasts** (*dict*) – Dictionary that holds the forecasts for the trained models

- **model_instance** (*TimeSeriesModel*) – Reference to the class implementing the actual time series model

__init__ (*model: str = 'prophet', revision_method: str = 'OLS', transform: Union[hts._t.Transform, bool, None] = False, n_jobs: int = 1, low_memory: bool = False, **kwargs*)

Parameters

- **model** (*str*) – One of the models supported by hts. These can be found
- **revision_method** (*str*) – The revision method to be used. One of: "OLS", "WLSS", "WLSV", "FP", "PHA", "AHP", "BU", "NONE"
- **transform** (*Boolean or NamedTuple*) – If True, `scipy.stats.boxcox` and `scipy.special._ufuncs.inv_boxcox` will be applied prior and after fitting. If False (default), no transform is applied. If you desired to use custom functions, use a NamedTuple like:

```
from collections import namedtuple

Transform = namedtuple('Transform', ['func', 'inv_func'])
transform = Transform(func=np.exp, inv_func=np.log)

ht = HTSRegressor(transform=transform, ...)
```

The signatures for the `func` as well as `inv_func` parameters must both be `Callable[[numpy.ndarray], numpy.ndarray]`, i.e. they must take an array and return an array, both of equal dimensions

- **n_jobs** (*int*) – Number of parallel jobs to run the forecasting on
- **low_memory** (*Bool*) – If True, models will be fit, serialized, and released from memory. Usually a good idea if you are dealing with a large amount of nodes
- **kwargs** – Keyword arguments to be passed to the underlying model to be instantiated

fit (*df: Optional[pandas.core.frame.DataFrame] = None, nodes: Optional[Dict[str, List[str]]] = None, tree: Optional[hts.hierarchy.HierarchyTree] = None, exogenous: Optional[Dict[str, List[str]]] = None, root: str = 'total', distributor: Optional[hts.utilities.distribution.DistributorBaseClass] = None, disable_progressbar=False, show_warnings=False, **fit_kwargs*) → `hts.core.regressor.HTSRegressor`
Fit hierarchical model to dataframe containing hierarchical data as specified in the `nodes` parameter.

Exogenous can also be passed as a dict of (string, list), where string is the specific node key and the list contains the names of the columns to be used as exogenous variables for that node.

Alternatively, a pre-built `HierarchyTree` can be passed without specifying the node and `df`. See more at [`hts.hierarchy.HierarchyTree`](#)

Parameters

- **df** (*pandas.DataFrame*) – A Dataframe of time series with a `DateTimeIndex`. Each column represents a node in the hierarchy. Ignored if `tree` argument is passed
- **nodes** (*Dict[str, List[str]]*) –

The hierarchy defined as a dict of (string, list), as specified in [`HierarchyTree.from_nodes`](#)

- **tree** (*HierarchyTree*) – A pre-built `HierarchyTree`. Ignored if `df` and `nodes` are passed, as the tree will be built from these

- **distributor** (*Optional[DistributorBaseClass]*) – A distributor, for parallel/distributed processing
- **exogenous** (*Dict[str, List[str]] or None*) – Node key mapping to columns that contain the exogenous variable for that node
- **root** (*str*) – The name of the root node
- **disable_progressbar** (*Bool*) – Disable or enable progressbar
- **show_warnings** (*Bool*) – Disable warnings
- **fit_kwargs** (*Any*) – Any arguments to be passed to the underlying forecasting model's fit function

Returns The fitted HTSRegressor instance

Return type *HTSRegressor*

predict (*exogenous_df: pandas.core.frame.DataFrame = None, steps_ahead: int = None, distributor: Optional[hts.utilities.distribution.DistributorBaseClass] = None, disable_progressbar: bool = False, show_warnings: bool = False, **predict_kwargs*) → *pandas.core.frame.DataFrame*

Parameters

- **distributor** (*Optional[DistributorBaseClass]*) – A distributor, for parallel/distributed processing
- **disable_progressbar** (*Bool*) – Disable or enable progressbar
- **show_warnings** (*Bool*) – Disable warnings
- **predict_kwargs** (*Any*) – Any arguments to be passed to the underlying forecasting model's predict function
- **exogenous_df** (*pandas.DataFrame*) – A dataframe of length == *steps_ahead* containing the exogenous data for each of the nodes. Only required when using prophet or auto_arima models. See [fbprophet's additional regression docs](#) and [AutoARIMA's exogenous handling docs](#) for more information.

Other models do not require additional regressors at predict time.

- **steps_ahead** (*int*) – The number of forecasting steps for which to produce a forecast

Returns

- Revised Forecasts, as a *pandas.DataFrame* in the same format as the one passed for fitting, extended by *steps_ahead*
- time steps'

class *hts.RevisionMethod* (*name: str, sum_mat: numpy.ndarray, transformer*)

Bases: *object*

revise (*forecasts=None, mse=None, nodes=None*) → *numpy.ndarray*

Parameters

- **forecasts** –
- **mse** –
- **nodes** –

7.2 hts.convenience

`hts.convenience.revise_forecasts` (*method*: str, *forecasts*: Dict[str, Union[numpy.ndarray, pandas.core.series.Series, pandas.core.frame.DataFrame]], *errors*: Optional[Dict[str, float]] = None, *residuals*: Optional[Dict[str, Union[numpy.ndarray, pandas.core.series.Series, pandas.core.frame.DataFrame]]] = None, *summing_matrix*: numpy.ndarray = None, *nodes*: hts.t.NAryTreeT = None, *transformer*: Union[hts.t.Transform, bool] = None)

Convenience function to get revised forecast for pre-computed base forecasts

Parameters

- **method** (str) – The reconciliation method to use
- **forecasts** (Dict[str, ArrayLike]) – A dict mapping key name to its forecasts (including in-sample forecasts). Required, can be of type `numpy.ndarray` of `ndim == 1`, `pandas.Series`, or single columned `pandas.DataFrame`
- **errors** (Dict[str, float]) – A dict mapping key name to the in-sample errors. Required for methods: OLS, WLSS, WLSV if `residuals` is not passed
- **residuals** (Dict[str, ArrayLike]) – A dict mapping key name to the residuals of in-sample forecasts. Required for methods: OLS, WLSS, WLSV, can be of type `numpy.ndarray` of `ndim == 1`, `pandas.Series`, or single columned `pandas.DataFrame`. If passing residuals, `errors` dict is not required and will instead be calculated using MSE metric: `numpy.mean(numpy.array(residual) ** 2)`
- **summing_matrix** (numpy.ndarray) – Not required if `nodes` argument is passed, or if using BU approach
- **nodes** (NAryTreeT) – The tree of nodes as specified in [HierarchyTree](#). Required if not if using AHP, PHA, FP methods, or if using passing the OLS, WLSS, WLSV methods and not passing the `summing_matrix` parameter
- **transformer** (TransformT) – A transform with the method: `inv_func` that will be applied to the forecasts

Returns revised forecasts – The revised forecasts

Return type pandas.DataFrame

7.3 hts.defaults

7.4 hts.functions

`hts.functions._create_bl_str_col` (*df*: pandas.core.frame.DataFrame, *level_names*: List[str])
→ List[str]

Concatenate the column values of all the specified `level_names` by row into a single column.

Parameters

- **df** (pandas.DataFrame) – Tabular data.
- **level_names** (List[str]) – Levels in the hierarchy.

Returns Concatenated column values by row.

Return type List[str]

`hts.functions._get_bl (grouped_levels: List[str], bottom_levels: List[str]) → List[List[str]]`

Get bottom level columns required to sum to create grouped columns.

Parameters

- **grouped_levels** (*List[str]*) – Grouped level, underscore delimited, column names.
- **bottom_levels** (*List[str]*) – Bottom level, underscore delimited, column names.

Returns Bottom level column names that make up each individual aggregated node in the hierarchy.

Return type List[List[str]]

`hts.functions.add_agg_series_to_df (df: pandas.core.frame.DataFrame, grouped_levels: List[str], bottom_levels: List[str]) → pandas.core.frame.DataFrame`

Add aggregate series columns to wide dataframe.

Parameters

- **df** (*pandas.DataFrame*) – Wide dataframe containing bottom level series.
- **grouped_levels** (*List[str]*) – Grouped level, underscore delimited, column names.
- **bottom_levels** (*List[str]*) – Bottom level, underscore delimited, column names.

Returns Wide dataframe with all series in hierarchy.

Return type pandas.DataFrame

`hts.functions.forecast_proportions (forecasts, nodes)`

Cons: Produces biased revised forecasts even if base forecasts are unbiased

`hts.functions.get_agg_series (df: pandas.core.frame.DataFrame, levels: List[List[str]]) → List[str]`

Get aggregate level series names.

Parameters

- **df** (*pandas.DataFrame*) – Tabular data.
- **levels** (*List[List[str]]*) – List of lists containing the desired level of aggregation.

Returns Aggregate series names.

Return type List[str]

`hts.functions.get_hierarchical_df (df: pandas.core.frame.DataFrame, level_names: List[str], hierarchy: List[List[str]], date_colname: str, val_colname: str) → Tuple[pandas.core.frame.DataFrame, numpy.array, List[str]]`

Transform your tabular dataframe to a wide dataframe with desired levels a hierarchy.

Parameters

- **df** (*pd.DataFrame*) – Tabular dataframe
- **level_names** (*List[str]*) – Levels in the hierarchy.
- **hierarchy** (*List[List[str]]*) – Desired levels in your hierarchy.

- `date_colname` (*str*) – Date column name
- `val_colname` (*str*) – Name of column containing series values.

Returns

- *pd.DataFrame* – Wide dataframe with levels of specified aggregation.
- *np.array* – Summing matrix.
- *List[str]* – Summing matrix labels.

Examples

```
>>> import hts.functions
>>> hier_df = pandas.DataFrame(
    data={
        'ds': ['2020-01', '2020-02'] * 5,
        'lev1': ['A', 'A',
                 'A', 'A',
                 'A', 'A',
                 'B', 'B',
                 'B', 'B'],
        'lev2': ['X', 'X',
                 'Y', 'Y',
                 'Z', 'Z',
                 'X', 'X',
                 'Y', 'Y'],
        'val': [1, 2,
                 3, 4,
                 5, 6,
                 7, 8,
                 9, 10]
    }
)
>>> hier_df
   ds  lev1 lev2  val
0 2020-01   A   X    1
1 2020-02   A   X    2
2 2020-01   A   Y    3
3 2020-02   A   Y    4
4 2020-01   A   Z    5
5 2020-02   A   Z    6
6 2020-01   B   X    7
7 2020-02   B   X    8
8 2020-01   B   Y    9
9 2020-02   B   Y   10
>>> level_names = ['lev1', 'lev2']
>>> hierarchy = [['lev1'], ['lev2']]
>>> wide_df, sum_mat, sum_mat_labels = hts.functions.get_hierarchical_df(hier_df,
                                                                              level_
↳names=level_names,
                                                                              ↳
↳hierarchy=hierarchy,
                                                                              ↳
↳colname='ds',
                                                                              ↳
                                                                              ↳
↳colname='val')
>>> wide_df
```

(continues on next page)

(continued from previous page)

lev1_lev2	A_X	A_Y	A_Z	B_X	B_Y	total	A	B	X	Y	Z
ds											
2020-01	1	3	5	7	9	25	9	16	8	12	5
2020-02	2	4	6	8	10	30	12	18	10	14	6

`hts.functions.optimal_combination` (*forecasts*: *Dict[str, pandas.core.frame.DataFrame]*,
sum_mat: *numpy.ndarray*, *method*: *str*, *mse*: *Dict[str, float]*)

Produces the optimal combination of forecasts by trace minimization (as described by Wickramasuriya, Athanasopoulos, Hyndman in “Optimal Forecast Reconciliation for Hierarchical and Grouped Time Series Through Trace Minimization”)

Parameters

- **forecasts** (*dict*) – Dictionary of pandas.DataFrames containing the future predictions
- **sum_mat** (*np.ndarray*) – The summing matrix
- **method** (*str*) –

One of:

- OLS (ordinary least squares)
- WLSS (structurally weighted least squares)
- WLSV (variance weighted least squares)

- **mse** –

`hts.functions.project` (*hat_mat*: *numpy.ndarray*, *sum_mat*: *numpy.ndarray*, *optimal_mat*: *numpy.ndarray*) → *numpy.ndarray*

`hts.functions.proportions` (*nodes*, *forecasts*, *sum_mat*, *method*='PHA')

`hts.functions.to_sum_mat` (*ntree*: *hts.t.NAryTreeT* = *None*, *node_labels*: *List[str]* = *None*) → *Tuple[numpy.ndarray, List[str]]*

This function creates a summing matrix for the bottom up and optimal combination approaches All the inputs are the same as above The output is a summing matrix, see Rob Hyndman’s “Forecasting: principles and practice” Section 9.4

Parameters

- **ntree** (*NAryTreeT*) –
- **node_labels** (*List[str]*) – Labels corresponding to node names/ summing matrix. Get from `hts.functions.get_hierarchical_df(...)`

Returns

- *numpy.ndarray* – Summing matrix.
- *List[str]* – Row order list of the level in the hierarchy represented by each row in the summing matrix.

`hts.functions.y_hat_matrix` (*forecasts*, *keys*=*None*)

7.5 hts.revision

class hts.revision.RevisionMethod(*name: str, sum_mat: numpy.ndarray, transformer*)

Bases: object

revise (*forecasts=None, mse=None, nodes=None*) → numpy.ndarray

Parameters

- **forecasts** –
- **mse** –
- **nodes** –

7.6 hts.transforms

class hts.transforms.BoxCoxTransformer

Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

fit (*x: pandas.core.series.Series, y=None, **fit_params*)

fit_transform (*x: pandas.core.series.Series, y=None, **fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs), default=None*) – Target values (None for unsupervised transformations).
- ****fit_params** (*dict*) – Additional fit parameters.

Returns **X_new** – Transformed array.

Return type ndarray array of shape (n_samples, n_features_new)

inverse_transform (*x: Union[pandas.core.series.Series, numpy.ndarray]*)

transform (*x: pandas.core.series.Series*)

class hts.transforms.FunctionTransformer(*func: callable = None, inv_func: callable = None*)

Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

fit (*x: pandas.core.series.Series, y=None, **fit_params*)

fit_transform (*x: pandas.core.series.Series, y=None, **fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs), default=None*) – Target values (None for unsupervised transformations).

- ****fit_params** (*dict*) – Additional fit parameters.

Returns **X_new** – Transformed array.

Return type ndarray array of shape (n_samples, n_features_new)

inverse_transform (*x: Union[pandas.core.series.Series, numpy.ndarray]*)

transform (*x: pandas.core.series.Series*)

7.7 hts._t

class hts._t.ExtendedEnum

Bases: enum.Enum

An enumeration.

list = <bound method ExtendedEnum.list of <enum 'ExtendedEnum'>>

names = <bound method ExtendedEnum.names of <enum 'ExtendedEnum'>>

class hts._t.HierarchyVisualizerT

Bases: object

create_map ()

class hts._t.MethodT

Bases: hts._t.ExtendedEnum

An enumeration.

AHP = 'AHP'

BU = 'BU'

FP = 'FP'

NONE = 'NONE'

OLS = 'OLS'

PHA = 'PHA'

WLSS = 'WLSS'

WLSV = 'WLSV'

class hts._t.ModelT

Bases: str, hts._t.ExtendedEnum

An enumeration.

auto_arima = 'auto_arima'

holt_winters = 'holt_winters'

prophet = 'prophet'

sarimax = 'sarimax'

class hts._t.NAryTreeT

Bases: object

Type definition of an NAryTree

add_child (*key=None, item=None, exogenous=None*) → hts._t.NAryTreeT

```
exogenous = None
get_height() → int
get_level_order_labels() → List[List[str]]
get_node_height(key: str) → int
get_series() → pandas.core.series.Series
is_leaf() → bool
leaf_sum() → int
level_order_traversal() → List[List[int]]
num_nodes() → int
parent
string_repr(prefix=" ", _last=True)
sum_at_height(level) → int
to_pandas() → pandas.core.frame.DataFrame
traversal_level() → List[hts._t.NAryTreeT]
value_at_height(level: int) → List[T]

class hts._t.TimeSeriesModelT
    Bases: sklearn.base.BaseEstimator, sklearn.base.RegressorMixin
    Type definition of an TimeSeriesModel
    create_model(**kwargs)
    fit(*fit_args) → hts._t.TimeSeriesModelT
    predict(node: hts._t.NAryTreeT, **predict_args)

class hts._t.Transform(func, inv_func)
    Bases: tuple
    func
        Alias for field number 0
    inv_func
        Alias for field number 1

class hts._t.UnivariateModelT
    Bases: str, hts._t.ExtendedEnum
    An enumeration.
    arima = 'arima'
    auto_arima = 'auto_arima'
    holt_winters = 'holt_winters'
    prophet = 'prophet'
    sarimax = 'sarimax'
```

The model fitting as well as the forecasting offer the possibility of parallelization. Out of the box both tasks are parallelized by scikit-hts. However, the overhead introduced with the parallelization should not be underestimated. Here we discuss the different settings to control the parallelization. To achieve best results for your use-case you should experiment with the parameters.

8.1 Parallelization of Model Fitting

We use a `multiprocessing.Pool` to parallelize the fitting of each model to a node's data. On instantiation we set the Pool's number of worker processes to `n_jobs`. This field defaults to the number of processors on the current system. We recommend setting it to the maximum number of available (and otherwise idle) processors.

The chunksize of the Pool's map function is another important parameter to consider. It can be set via the `chunksize` field. By default it is up to `multiprocessing.Pool` is parallelisation parameter. One data chunk is defined as a singular time series for one node. The chunksize is the number of chunks that are submitted as one task to one worker process. If you set the chunksize to 10, then it means that one worker task corresponds to calculate all forecasts for 10 node time series. If it is set it to None, depending on distributor, heuristics are used to find the optimal chunksize. The chunksize can have an crucial influence on the optimal cluster performance and should be optimised in benchmarks for the problem at hand.

8.2 Parallelization of Forecasting

For the feature extraction scikit-hts exposes the parameters `n_jobs` and `chunksize`. Both behave analogue to the parameters for the feature selection.

To do performance studies and profiling, it sometimes quite useful to turn off parallelization at all. This can be setting the parameter `n_jobs` to 0.

8.2.1 Acknowledgement

This documentation, as well as the underlying implementation, exists only thanks to the folks at [blue-yonder](#). The This page was pretty much copy and pasted from their [tsfresh](#) package. Many thanks for their excellent package.

How to deploy scikit-hts at scale

The high volume of time series data can demand an analysis at scale. So, time series need to be processed on a group of computational units instead of a singular machine.

Accordingly, it may be necessary to distribute the extraction of time series features to a cluster. Indeed, it is possible to extract features with *hts* in a distributed fashion. This page will explain how to setup a distributed *hts*.

9.1 The distributor class

To distribute the calculation of features, we use a certain object, the Distributor class (contained in the `hts.utilities.distribution` module).

Essentially, a Distributor organizes the application of feature calculators to data chunks. It maps the feature calculators to the data chunks and then reduces them, meaning that it combines the results of the individual mapping into one object, the feature matrix.

So, Distributor will, in the following order,

1. calculates an optimal `chunk_size`, based on the characteristics of the time series data at hand (by `calculate_best_chunk_size()`)
2. split the time series data into chunks (by `partition()`)
3. distribute the applying of the feature calculators to the data chunks (by `distribute()`)
4. combine the results into the feature matrix (by `map_reduce()`)
5. close all connections, shutdown all resources and clean everything (by `close()`)

So, how can you use such a Distributor to extract features with *hts*? You will have to pass it into as the `distributor` argument to the `extract_features()` method.

The following example shows how to define the `MultiprocessingDistributor`, which will distribute the calculations to a local pool of threads:

```

from hts import HTSRegressor
from hts.utilities.load_data import load_mobility_data
from hts.utilities.distribution import MultiprocessingDistributor

df = load_mobility_data()

# Define hierarchy
hier = {
    'total': ['CH', 'SLU', 'BT', 'OTHER'],
    'CH': ['CH-07', 'CH-02', 'CH-08', 'CH-05', 'CH-01'],
    'SLU': ['SLU-15', 'SLU-01', 'SLU-19', 'SLU-07', 'SLU-02'],
    'BT': ['BT-01', 'BT-03'],
    'OTHER': ['WF-01', 'CBD-13']
}

distributor = MultiprocessingDistributor(n_workers=4,
                                         disable_progressbar=False,
                                         progressbar_title="Feature Extraction")
hts.fit(df=df, nodes=hier, n_jobs=4, distributor=distributor)

```

This example actually corresponds to the existing multiprocessing API, where you just specify the number of jobs, without the need to construct the Distributor:

```

from hts import HTSRegressor
from hts.utilities.load_data import load_mobility_data

df = load_mobility_data()

# Define hierarchy
hier = {
    'total': ['CH', 'SLU', 'BT', 'OTHER'],
    'CH': ['CH-07', 'CH-02', 'CH-08', 'CH-05', 'CH-01'],
    'SLU': ['SLU-15', 'SLU-01', 'SLU-19', 'SLU-07', 'SLU-02'],
    'BT': ['BT-01', 'BT-03'],
    'OTHER': ['WF-01', 'CBD-13']
}

hts.fit(df=df, nodes=hier, n_jobs=4)

```

9.2 Using dask to distribute the calculations

We provide distributor for the [dask framework](#), where “*Dask is a flexible parallel computing library for analytic computing.*”

Dask is a great framework to distribute analytic calculations to a cluster. It scales up and down, meaning that you can even use it on a singular machine. The only thing that you will need to run *hts* on a Dask cluster is the ip address and port number of the [dask-scheduler](#).

Lets say that your dask scheduler is running at 192.168.0.1:8786, then we can easily construct a `ClusterDaskDistributor` that connects to the scheduler and distributes the time series data and the calculation to a cluster:

```

from hts import HTSRegressor
from hts.utilities.load_data import load_mobility_data
from hts.utilities.distribution import ClusterDaskDistributor

```

(continues on next page)

(continued from previous page)

```

df = load_mobility_data()

# Define hierarchy
hier = {
    'total': ['CH', 'SLU', 'BT', 'OTHER'],
    'CH': ['CH-07', 'CH-02', 'CH-08', 'CH-05', 'CH-01'],
    'SLU': ['SLU-15', 'SLU-01', 'SLU-19', 'SLU-07', 'SLU-02'],
    'BT': ['BT-01', 'BT-03'],
    'OTHER': ['WF-01', 'CBD-13']
}

distributor = ClusterDaskDistributor(address="192.168.0.1:8786")
hts.fit(df=df, nodes=hier)
...

# Prediction also runs in a distributed fashion
preds = hts.predict(steps_ahead=10)

```

Compared to the `MultiprocessingDistributor` example from above, we only had to change one line to switch from one machine to a whole cluster. It is as easy as that. By changing the `Distributor` you can easily deploy your application to run to a cluster instead of your workstation.

You can also use a local `DaskCluster` on your local machine to emulate a `Dask` network. The following example shows how to setup a `LocalDaskDistributor` on a local cluster of 3 workers:

```

from hts import HTSRegressor
from hts.utilities.load_data import load_mobility_data
from hts.utilities.distribution import LocalDaskDistributor

df = load_mobility_data()

# Define hierarchy
hier = {
    'total': ['CH', 'SLU', 'BT', 'OTHER'],
    'CH': ['CH-07', 'CH-02', 'CH-08', 'CH-05', 'CH-01'],
    'SLU': ['SLU-15', 'SLU-01', 'SLU-19', 'SLU-07', 'SLU-02'],
    'BT': ['BT-01', 'BT-03'],
    'OTHER': ['WF-01', 'CBD-13']
}

distributor = LocalDaskDistributor(n_workers=3)
hts.fit(df=df, nodes=hier)
...

# Prediction also runs in a distributed fashion
preds = hts.predict(steps_ahead=10)

```

9.3 Writing your own distributor

If you want to use another framework than `Dask`, you will have to write your own `Distributor`. To construct your custom `Distributor`, you will have to define an object that inherits from the abstract base class `hts.utilities.distribution.DistributorBaseClass`. The `hts.utilities.distribution` module contains more

information about what you will need to implement.

9.4 Acknowledgement

This documentation, as well as the underlying implementation, exists only thanks to the folks at [blue-yonder](#). The This page was pretty much copy and pasted from their [tsfresh](#) package. Many thanks for their excellent package.

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways:

10.1 Types of Contributions

10.1.1 Report Bugs

Report bugs at <https://github.com/carlomazzaferro/scikit-hts/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

10.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

10.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

10.1.4 Write Documentation

scikit-hts could always use more documentation, whether as part of the official scikit-hts docs, in docstrings, or even on the web in blog posts, articles, and such.

10.1.5 Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/carlomazzaferro/scikit-hts/issues>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

10.2 Get Started!

Ready to contribute? Here's how to set up *scikit-hts* for local development.

1. Fork the *scikit-hts* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/scikit-hts.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv scikit-hts
$ cd scikit-hts/
$ pip install -e ."[all]"
$ pip install -e ."[dev]"
$ pip install -e ."[test]"
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass black, flake8 and isort and the tests with *Make*:

```
$ REPORT=False make test
```

To get the linting done, run:

```
$ black .
$ isort --profile black .
$ flake8 hts
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

10.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6, unless it is a python compatibility request that targets a specific python release. Check <https://github.com/carlomazzaferro/scikit-hts/actions> and make sure that the tests pass for all supported Python versions.

10.4 Tips

To run a subset of tests:

```
$ pytest tests.test_yor_test_file
```

10.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version --new-version 0.5.X patch # X = current + 1
$ git push
$ git push --tags
```

Github Actions will then deploy to PyPI if tests pass.

11.1 0.1.0 (2020-01-02)

- First release on PyPI.

11.2 0.2.0 (2018-02-13)

- Major feature implementation and documentation
- Static typing
- Testing - 44% coverage

11.3 0.2.3 (2020-03-28)

- Testing up to 75%
- Exogenous variable support
- Extensive docs

11.4 0.3.0 (2020-03-28)

- Parallel and distributed training

11.5 0.4.0 (2020-03-28)

- Testing for all reconciliation methods, line coverage > 80%

11.6 0.4.1 (2020-03-28)

- Python 3.6 support

11.7 0.5.2 (2020-03-28)

- Added support for no revision, thanks @ryanvolpi
- Added multiple example at <https://github.com/carlomazzaferro/scikit-hts-examples>, thanks @vtoliveira
- Logging fixes and usability improvements

11.8 0.5.3 (2021-02-23)

- Support for grouped time series, thanks to @noahsa! See: <https://github.com/carlomazzaferro/scikit-hts/pull/51>

11.9 0.5.4 (2021-04-20)

- Fixed long-standing BU forecasting bug, thanks to @javierhuertay! See: <https://github.com/carlomazzaferro/scikit-hts/issues/35>

11.10 0.5.6 (2021-04-20)

- Fixed input sanitization for convenience methods. See: <https://github.com/carlomazzaferro/scikit-hts/issues/65>

11.11 0.5.7 (2021-05-30)

- Ability to build hierarchies from tabular data. Thanks @noahsa! See: <https://github.com/carlomazzaferro/scikit-hts/pull/70>

11.12 0.5.8 (2021-05-30)

- Fix long-standing bugs related to transformers implementation. See: <https://github.com/carlomazzaferro/scikit-hts/issues/66>, <https://github.com/carlomazzaferro/scikit-hts/issues/33>, <https://github.com/carlomazzaferro/scikit-hts/issues/38>

11.13 0.5.9 (2021-05-30)

- Fix long-standing bugs related to handling exogenous variables. See: <https://github.com/carlomazzaferro/scikit-hts/issues/55>

11.14 0.5.10 (2021-06-5)

- Minor bug fix for transforms fixed: <https://github.com/carlomazzaferro/scikit-hts/issues/66#issuecomment-855223892>

11.15 0.5.11 (2021-06-5)

- Further fix to exogenous variable handling, thanks to @wilfredesert! See: <https://github.com/carlomazzaferro/scikit-hts/issues/75>

CHAPTER 12

Indices and tables

- `genindex`
- `modindex`
- `search`

h

- `hts`, 25
- `hts._t`, 33
- `hts.convenience`, 28
- `hts.defaults`, 28
- `hts.functions`, 28
- `hts.hierarchy`, 16
- `hts.model`, 21
- `hts.revision`, 32
- `hts.transforms`, 32

Symbols

`__init__()` (*hts.HTSRegressor* method), 26
`_create_bl_str_col()` (in module *hts.functions*), 28
`_get_bl()` (in module *hts.functions*), 29

A

`add_agg_series_to_df()` (in module *hts.functions*), 29
`add_child()` (*hts._t.NAryTreeT* method), 33
 AHP (*hts._t.MethodT* attribute), 33
 arima (*hts._t.UnivariateModelT* attribute), 34
 auto_arima (*hts._t.ModelT* attribute), 33
 auto_arima (*hts._t.UnivariateModelT* attribute), 34
 AutoArimaModel (class in *hts.model*), 21

B

BoxCoxTransformer (class in *hts.transforms*), 32
 BU (*hts._t.MethodT* attribute), 33

C

`create_map()` (*hts._t.HierarchyVisualizerT* method), 33
`create_model()` (*hts._t.TimeSeriesModelT* method), 34

E

exogenous (*hts._t.NAryTreeT* attribute), 33
 ExtendedEnum (class in *hts._t*), 33

F

FBProphetModel (class in *hts.model*), 22
`fit()` (*hts._t.TimeSeriesModelT* method), 34
`fit()` (*hts.HTSRegressor* method), 26
`fit()` (*hts.model.AutoArimaModel* method), 21
`fit()` (*hts.model.FBProphetModel* method), 22
`fit()` (*hts.model.HoltWintersModel* method), 22
`fit()` (*hts.model.SarimaxModel* method), 22
`fit()` (*hts.transforms.BoxCoxTransformer* method), 32

`fit()` (*hts.transforms.FunctionTransformer* method), 32
`fit_transform()` (*hts.transforms.BoxCoxTransformer* method), 32
`fit_transform()` (*hts.transforms.FunctionTransformer* method), 32
`forecast_proportions()` (in module *hts.functions*), 29
 FP (*hts._t.MethodT* attribute), 33
`from_geo_events()` (*hts.hierarchy.HierarchyTree* class method), 16
`from_nodes()` (*hts.hierarchy.HierarchyTree* class method), 16
 func (*hts._t.Transform* attribute), 34
 FunctionTransformer (class in *hts.transforms*), 32

G

`get_agg_series()` (in module *hts.functions*), 29
`get_height()` (*hts._t.NAryTreeT* method), 34
`get_hierarchichal_df()` (in module *hts.functions*), 29
`get_level_order_labels()` (*hts._t.NAryTreeT* method), 34
`get_level_order_labels()` (*hts.hierarchy.HierarchyTree* method), 18
`get_node()` (*hts.hierarchy.HierarchyTree* method), 18
`get_node_height()` (*hts._t.NAryTreeT* method), 34
`get_series()` (*hts._t.NAryTreeT* method), 34

H

HierarchyTree (class in *hts.hierarchy*), 16
 HierarchyVisualizerT (class in *hts._t*), 33
 holt_winters (*hts._t.ModelT* attribute), 33
 holt_winters (*hts._t.UnivariateModelT* attribute), 34
 HoltWintersModel (class in *hts.model*), 22
 hts (module), 25
 hts._t (module), 33
 hts.convenience (module), 28

`hts.defaults` (*module*), 28
`hts.functions` (*module*), 28
`hts.hierarchy` (*module*), 16
`hts.model` (*module*), 21
`hts.revision` (*module*), 32
`hts.transforms` (*module*), 32
`HTSRegressor` (*class in hts*), 25

I

`inv_func` (*hts._t.Transform attribute*), 34
`inverse_transform()`
 (*hts.transforms.BoxCoxTransformer method*), 32
`inverse_transform()`
 (*hts.transforms.FunctionTransformer method*), 33
`is_leaf()` (*hts._t.NAryTreeT method*), 34
`is_leaf()` (*hts.hierarchy.HierarchyTree method*), 18

L

`leaf_sum()` (*hts._t.NAryTreeT method*), 34
`level_order_traversal()` (*hts._t.NAryTreeT method*), 34
`level_order_traversal()`
 (*hts.hierarchy.HierarchyTree method*), 18
`list` (*hts._t.ExtendedEnum attribute*), 33

M

`MethodT` (*class in hts._t*), 33
`ModelT` (*class in hts._t*), 33

N

`names` (*hts._t.ExtendedEnum attribute*), 33
`NAryTreeT` (*class in hts._t*), 33
`NONE` (*hts._t.MethodT attribute*), 33
`num_nodes()` (*hts._t.NAryTreeT method*), 34
`num_nodes()` (*hts.hierarchy.HierarchyTree method*), 18

O

`OLS` (*hts._t.MethodT attribute*), 33
`optimal_combination()` (*in module hts.functions*), 31

P

`parent` (*hts._t.NAryTreeT attribute*), 34
`PHA` (*hts._t.MethodT attribute*), 33
`predict()` (*hts._t.TimeSeriesModelT method*), 34
`predict()` (*hts.HTSRegressor method*), 27
`predict()` (*hts.model.AutoArimaModel method*), 22
`predict()` (*hts.model.FBProphetModel method*), 22
`predict()` (*hts.model.HoltWintersModel method*), 22
`predict()` (*hts.model.SarimaxModel method*), 22

`project()` (*in module hts.functions*), 31
`prophet` (*hts._t.ModelT attribute*), 33
`prophet` (*hts._t.UnivariateModelT attribute*), 34
`proportions()` (*in module hts.functions*), 31

R

`revise()` (*hts.revision.RevisionMethod method*), 32
`revise()` (*hts.RevisionMethod method*), 27
`revise_forecasts()` (*in module hts.convenience*), 28
`RevisionMethod` (*class in hts*), 27
`RevisionMethod` (*class in hts.revision*), 32

S

`sarimax` (*hts._t.ModelT attribute*), 33
`sarimax` (*hts._t.UnivariateModelT attribute*), 34
`SarimaxModel` (*class in hts.model*), 22
`string_repr()` (*hts._t.NAryTreeT method*), 34
`sum_at_height()` (*hts._t.NAryTreeT method*), 34

T

`TimeSeriesModelT` (*class in hts._t*), 34
`to_pandas()` (*hts._t.NAryTreeT method*), 34
`to_pandas()` (*hts.hierarchy.HierarchyTree method*), 18
`to_sum_mat()` (*in module hts.functions*), 31
`Transform` (*class in hts._t*), 34
`transform()` (*hts.transforms.BoxCoxTransformer method*), 32
`transform()` (*hts.transforms.FunctionTransformer method*), 33
`traversal_level()` (*hts._t.NAryTreeT method*), 34
`traversal_level()` (*hts.hierarchy.HierarchyTree method*), 18

U

`UnivariateModelT` (*class in hts._t*), 34

V

`value_at_height()` (*hts._t.NAryTreeT method*), 34

W

`WLSS` (*hts._t.MethodT attribute*), 33
`WLSV` (*hts._t.MethodT attribute*), 33

Y

`y_hat_matrix()` (*in module hts.functions*), 31