anticipy 0.2.1

Release 0.2.1

unknown

Jan 18, 2023

CONTENTS

Contents:

CHAPTER

ONE

OVERVIEW

Anticipy is a tool to generate forecasts for time series. It takes a pandas Series or DataFrame as input, and returns a DataFrame with the forecasted values for a given period of time.

Features:

- **Simple interface**. Start forecasting with a single function call on a pandas DataFrame.
- **Model selection**. If you provide different models (e.g. linear, sigmoidal, exponential), the tool will compare them and choose the best fit for your data.
- **Trend and seasonality**. Support for weekly and monthly seasonality, among other types.
- **Calendar events**. Provide lists of special dates, such as holiday seasons or bank holidays, to improve model performance.
- **Data cleaning**. The library has tools to identify and remove outliers, and to detect and handle step changes in the data.

To get started, install the library with pip:

```
pip install anticipy
```

It is straightforward to generate a simple linear model with the tool - just call `forecast.run_forecast(my_dataframe)()`:

```
import pandas as pd, numpy as np
from anticipy import forecast, forecast_models

df = pd.DataFrame({'y': np.full(20,10.0)+np.random.normal(0.0, 0.1, 20),
                   'date':pd.date_range('2018-01-01', periods=20, freq='D')})
df_forecast = forecast.run_forecast(df, extrapolate_years=0.5)
print(df_forecast.tail(3))
```

Output:

```
.         date source  is_actuals   model         y        q5       q20        q80       ␣
↪ q95
219 2018-07-19    src       False  linear  9.490259  7.796581  8.339835  10.556202  11.
↪689470
220 2018-07-20    src       False  linear  9.487518  7.828049  8.362620  10.466285  11.
↪640854
221 2018-07-21    src       False  linear  9.484776  7.776001  8.343068  10.423964  11.
↪696145
```

For more advanced usage, check the *Tutorial*.

CHAPTER

TWO

INSTALLATION

To install the project with pip including all the required dependencies do:

```
pip install anticipy
```

To install the optional dependencies, do the following within your virtual environment:

```
pip install -e .[extras]
```

CHAPTER

THREE

TUTORIAL

Contents:

3.1 Getting Started

To get started, install the library with pip

```
pip install anticipy
```

It is straightforward to generate a forecast with the tool - just call `forecast.run_forecast(my_dataframe)()`:

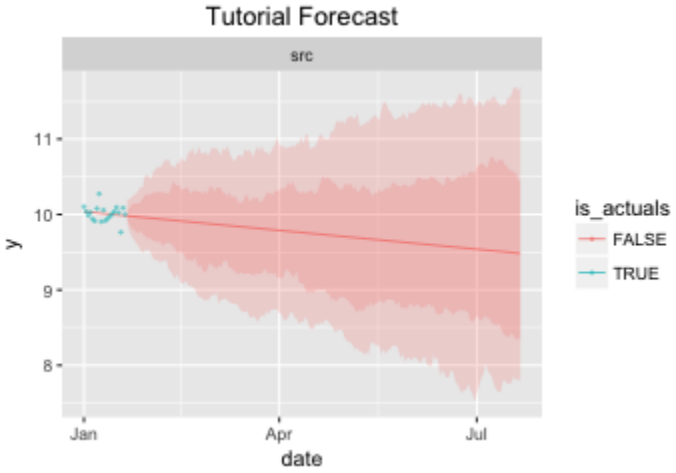
```
import pandas as pd, numpy as np
from anticipy import forecast, forecast_models, forecast_plot

df = pd.DataFrame({'y': np.full(20,10.0)+np.random.normal(0.0, 0.1, 20),
                   'date':pd.date_range('2018-01-01', periods=20, freq='D')})
df_forecast = forecast.run_forecast(df, extrapolate_years=0.5)
print(df_forecast.tail(3))
```

Output:

```
.         date source  is_actuals   model         y        q5       q20        q80       ␣
↪ q95
219 2018-07-19    src       False  linear  9.490259  7.796581  8.339835  10.556202  11.
↪689470
220 2018-07-20    src       False  linear  9.487518  7.828049  8.362620  10.466285  11.
↪640854
221 2018-07-21    src       False  linear  9.484776  7.776001  8.343068  10.423964  11.
↪696145
```

The tool automatically selects the best model from a list of defaults - in this case, a simple linear model. Advanced users can instead provide their preferred model or lists of candidate models, as explained in *Forecast models*.

You can plot the forecast output using the functions in `forecast_plot`. *`anticipy.forecast_plot.plot_forecast()`* saves the plot as a file or exports it to a jupyter notebook. The plot looks as follows:



The code to generate the plot is:

```
path_tutorial_plot = 'plot-tutorial'
# Save plot as a file
forecast_plot.plot_forecast(df_forecast, output='html',
path=path_tutorial_plot, width=350, height=240, title='Tutorial Forecast')
# Show plot in a jupyter notebook
forecast_plot.plot_forecast(df_forecast, output='jupyter', width=350,
height=240, title='Tutorial Forecast')
```

3.2 Input Format

The input time series needs to be formatted as a pandas series or dataframe. If a dataframe is passed, the following columns are used:

- **y:** float, values of the time series.
- **date:** (optional) timestamp, date and time of each sample. If this column is not present, and the dataframe index is a DatetimeIndex, the index will be used instead. This is an optional column, only required when using models that are aware of the date, such as weekly seasonality models.
- **x:** (optional) float or int, numeric index of each sample. If this column is not present, it will be inferred from the date column or, if that is not possible, the dataframe index will be used instead.
- **source:** (optional) string, the name of the data source for this time series. You can include multiple time series in your input dataframe, using different source values to identify them.

If a series is passed, 'y' will be the series values and, if the index is a DateTimeIndex, it will be used as the 'date' column. If the index is numeric, it will be used as the 'x' column instead.

An input dataframe should meet the following constraints:

- Minimum required samples depends on number of parameters in the chosen models. run_forecast() will fail to fit if n_samples < n_parameters+2

- y column may include null values.
- Multiple values per sample may be included, e.g. when the same metric is observed by multiple sources. In that case, it is possible to assign weights to each individual sample so that one source will be given higher priority. To do that, include a 'weight' column of floats in the input dataframe.
- A date column or date index is only required if the model is date-aware.
- Non-default column names may be used. In that case, you need to pass the new column names to run_forecast() using the col_name_ parameters.

Here is an example of using col_name_ parameters to run a forecast witn non-default column names:

```
df2 = pd.DataFrame({'my_y': np.full(20,10.0)+np.random.normal(0.0, 0.1, 20),
                    'my_date':pd.date_range('2018-01-01', periods=20, freq='D')})
df_forecast2 = forecast.run_forecast(df2, extrapolate_years=0.5,
                                     col_name_y='my_y',
                                     col_name_date='my_date')
```

3.3 Detailed Output

The library uses scipy.optimize to fit model functions to the input data. You can examine the model parameters, quality metrics and other useful information with the argument simplify_output=False:

```
dict_result = forecast.run_forecast(df, extrapolate_years=0.5,
                                    simplify_output=False, include_all_fits=True)
# Table with actuals and forecast for best-fitting model, including prediction intervals
print dict_result['forecast'].groupby('model').tail(1)
# Table including time series actuals and forecast
print dict_result['data'].groupby('model').tail(1)
# Metadata table: model parameters and fitting output
print dict_result['metadata']
# Table with output data from scipy.optimize, for debugging purposes
print dict_result['optimize_info']
```

Output - forecast table, same as output from run_forecast(simplify_output=True):

```
.         date source  is_actuals          model          y        q5        q20       ␣
↪q80        q95
19  2018-01-20    src        True              y   9.928176       NaN        NaN       ␣
↪NaN        NaN
221 2018-07-21    src       False  (linear+ramp)  10.838865  9.597208  10.121438  11.
↪717812  12.551523
```

Output - data table. Has actuals and forecasts, including forecasts from non-optimal models if include_all_fits=True

```
.         date                          model          y source                      ␣
↪source_long  is_actuals  is_weight  is_filtered  is_best_fit
19  2018-01-20                         weight   1.000000    src  src:1-1:D:2018-01-
↪01::2018-01-20        True       True        False        False
39  2018-01-20                              y   9.928176    src  src:1-1:D:2018-01-
↪01::2018-01-20        True      False        False        False
241 2018-07-21                         linear   9.230972    src  src:1-1:D:2018-01-
↪01::2018-01-20       False      False        False        False
```

(continues on next page)

(continued from previous page)

```
443 2018-07-21          (linear+season_wday)   9.283372    src  src:1-1:D:2018-01-
↪01::2018-01-20       False      False       False       False
645 2018-07-21                 (linear+ramp)  10.838865    src  src:1-1:D:2018-01-
↪01::2018-01-20       False      False       False        True
847 2018-07-21  ((linear+ramp)+season_wday)  10.989835    src  src:1-1:D:2018-01-
↪01::2018-01-20       False      False       False       False
```

Output - metadata table. Includes model parameters and model quality metrics such as cost and AICC:

```
.  source                       model weights         actuals_x_range freq  is_fit     ␣
↪cost       aic_c                                        params_str status           ␣
↪          source_long                                              params  is_best_fit
0    src                      linear     1-1  2018-01-01::2018-01-20    D    True  0.
↪063076 -111.182993                                  [-3.9e-03  1.0e+01]    FIT  src:1-
↪1:D:2018-01-01::2018-01-20       [-0.0038931365581278176, 10.013491979601325]       ␣
↪False
1    src         (linear+season_wday)     1-1  2018-01-01::2018-01-20    D    True  0.
↪039519  -95.533948  [-3.3e-03  1.0e+01  1.0e-01 -1.4e-02  8.4e-02 ...    FIT  src:1-
↪1:D:2018-01-01::2018-01-20  [-0.0032764198059819344, 9.975993454168774, 0....       ␣
↪False
2    src                (linear+ramp)     1-1  2018-01-01::2018-01-20    D    True  0.
↪045997 -111.498115                                    [-0.  10.1  6.   0. ]    FIT  src:1-
↪1:D:2018-01-01::2018-01-20  [-0.030005422538483477, 10.103677325164737, 6....       ␣
↪True
3    src  ((linear+ramp)+season_wday)     1-1  2018-01-01::2018-01-20    D    True  0.
↪020303  -93.853970  [-3.2e-02  1.0e+01  6.0e+00  3.8e-02  1.1e-01 ...    FIT  src:1-
↪1:D:2018-01-01::2018-01-20  [-0.0318590092714045, 10.062880686158646, 6.00...       ␣
↪False
```

Output - optimize information table. Includes detailed data generated by scipy.optimize, useful for debugging:

```
.  source                       model  success                                          ␣
↪params_str      cost    optimality  iterations  status  jac_evals                      ␣
↪              message                      source_long                                ␣
↪             params
0    src                      linear     True                                  [-3.9e-03 ␣
↪1.0e+01]  0.063076  1.213028e-09           4       1          4  `gtol` termination␣
↪condition is satisfied.  src:1-1:D:2018-01-01::2018-01-20       [-0.
↪0038931365581278176, 10.013491979601325]
1    src         (linear+season_wday)     True  [-3.3e-03  1.0e+01  1.0e-01 -1.4e-02  8.
↪4e-02 ...  0.039519  8.348877e-14           4       1          4  `gtol` termination␣
↪condition is satisfied.  src:1-1:D:2018-01-01::2018-01-20  [-0.0032764198059819344, 9.
↪975993454168774, 0....
2    src                (linear+ramp)     True                                [-0.  10.1 ␣
↪6.   0. ]  0.045997  1.765921e-03          34       2         22  `ftol` termination␣
↪condition is satisfied.  src:1-1:D:2018-01-01::2018-01-20  [-0.030005422538483477, 10.
↪103677325164737, 6....
3    src  ((linear+ramp)+season_wday)     True  [-3.2e-02  1.0e+01  6.0e+00  3.8e-02  1.
↪1e-01 ...  0.020303  2.777755e-02          45       2         28  `ftol` termination␣
↪condition is satisfied.  src:1-1:D:2018-01-01::2018-01-20  [-0.0318590092714045, 10.
↪062880686158646, 6.00...
```

3.4 Forecast models

By default, run_forecast() automatically generates a list of candidate models. However, you can specify a list of models in the argument l_model_trend, so that the tool fits each model and chooses the best. Only the best fitting model will be included in the output, unless you use the argument include_all_fits=True. The following example runs a forecast with two models: linear and constant:

```
dict_result = forecast.run_forecast(df, extrapolate_years=1, simplify_output=False,
                                    l_model_trend = [forecast_models.model_linear,
                                                     forecast_models.model_constant],
                                    include_all_fits=True)
# Table including time series actuals and forecast
print dict_result['data'].tail(6)
# Metadata table: model parameters and fitting output
print dict_result['metadata']
```

Output:

```
.           date     model    y source                    source_long  is_actuals  is_
↪weight  is_filtered  is_best_fit
739 2018-12-31  constant  2.0    src  src:1:D:2018-01-01::2018-01-05       False     ␣
↪False        False        False
740 2019-01-01  constant  2.0    src  src:1:D:2018-01-01::2018-01-05       False     ␣
↪False        False        False
741 2019-01-02  constant  2.0    src  src:1:D:2018-01-01::2018-01-05       False     ␣
↪False        False        False
742 2019-01-03  constant  2.0    src  src:1:D:2018-01-01::2018-01-05       False     ␣
↪False        False        False
743 2019-01-04  constant  2.0    src  src:1:D:2018-01-01::2018-01-05       False     ␣
↪False        False        False
744 2019-01-05  constant  2.0    src  src:1:D:2018-01-01::2018-01-05       False     ␣
↪False        False        False
```

```
.  source     model weights         actuals_x_range freq  is_fit          cost       aic_
↪c         params_str status                     source_long                          ␣
↪params  is_best_fit
0    src    linear       1  2018-01-01::2018-01-05    D    True  6.162976e-33 -368.
↪880931  [1.0e+00 1.1e-16]    FIT  src:1:D:2018-01-01::2018-01-05  [1.0, 1.
↪1102230246251565e-16]         True
1    src  constant       1  2018-01-01::2018-01-05    D    True  5.000000e+00    3.
↪000000               [2.]    FIT  src:1:D:2018-01-01::2018-01-05                    ␣
↪   [2.0]        False
```

You can configure run_forecast to fit a seasonality model in addition to the trend model. To do so, include the argument l_model_season with a list of one or more seasonality models. If the list includes model_null, a non-seasonal model will also be fit and compared with the seasonal models. The function tries all combinations of trend models and seasonality models and selects the best:

```
df=pd.DataFrame({'y': np.full(21, 10.)+np.tile(np.arange(0., 7),3)},
           index=pd.date_range('2018-01-01', periods=21, freq='D'))
dict_result = forecast.run_forecast(df, extrapolate_years=0.5, simplify_output=False,
                                    l_model_trend = [forecast_models.model_linear,
                                                     forecast_models.model_constant],
```

(continues on next page)

(continued from previous page)

```
                                    l_model_season = [forecast_models.model_null,
↪# no seasonality model
                                                     forecast_models.model_season_wday],
↪# weekday seasonality model
                                    include_all_fits=True)

print dict_result['data'].tail(6)
print dict_result['metadata'][['source','model','is_fit','cost','aic_c','params_str','is_
↪best_fit']]
```

Output:

```
.          date                          model     y source                    source_
↪long  is_actuals  is_weight  is_filtered  is_best_fit
2331 2019-01-16  (constant_mult_season_wday)  12.0    src  src:1:D:2018-01-01::2018-01-
↪21       False      False        False        False
2332 2019-01-17  (constant_mult_season_wday)  13.0    src  src:1:D:2018-01-01::2018-01-
↪21       False      False        False        False
2333 2019-01-18  (constant_mult_season_wday)  14.0    src  src:1:D:2018-01-01::2018-01-
↪21       False      False        False        False
2334 2019-01-19  (constant_mult_season_wday)  15.0    src  src:1:D:2018-01-01::2018-01-
↪21       False      False        False        False
2335 2019-01-20  (constant_mult_season_wday)  16.0    src  src:1:D:2018-01-01::2018-01-
↪21       False      False        False        False
2336 2019-01-21  (constant_mult_season_wday)  10.0    src  src:1:D:2018-01-01::2018-01-
↪21       False      False        False        False
```

```
. source                         model  is_fit          cost       aic_c                ␣
↪                      params_str  is_best_fit
0    src                        linear    True  3.741818e+01   16.130320                ␣
↪                      [ 0.1 11.9]       False
1    src     (linear_add_season_wday)    True  1.840127e-15 -742.442745  [3.5e-10 6.
↪3e+00 3.7e+00 4.7e+00 5.7e+00 6.7e+...       False
2    src                      constant    True  4.200000e+01   16.556091                ␣
↪                        [13.]       False
3    src   (constant_add_season_wday)    True  1.686607e-15 -750.272181          [12.7 -
↪2.7 -1.7 -0.7  0.3  1.3  2.3  3.3]       False
4    src    (linear_mult_season_wday)    True  2.761458e-16 -782.272627  [-2.2e-10  1.
↪9e+01  5.3e-01  5.9e-01  6.4e-01 ...        True
5    src  (constant_mult_season_wday)    True  6.833738e-13 -624.181393          [15.9 ␣
↪0.6  0.7  0.8  0.8  0.9  0.9  1. ]       False
```

The following trend and seasonality models are currently supported. They are available as attributes from *anticipy.forecast_models*:

Table 1: Default forecast models

name	params	formula	notes
model_null	0	y=0	Does nothing. Used to disable components (e.g. seasonality)
model_linear	2	y=Ax + B	Linear model
model_ramp	2	y = (x-A)*B if x>A	Ramp model
model_season_wday	6	see desc.	Weekday seasonality model. Assigns a constant value to each weekday
model_season_fourier_yearly	20	see desc	Fourier yearly seasonality model

Table 2: Other forecast models

name	params	formula	notes
model_constant	1	y=A	Constant model
model_linear_nondec	2	y=Ax + B	Non decreasing linear model. With boundaries to ensure model slope >=0
model_quasilinear	3	y=A*(x^B) + C	Quasilinear model
model_exp	2	y=A * B^x	Exponential model
model_decay	4	Y = A * e^(B*(x-C)) + D	Exponential decay model
model_step	2	y=0 if x<A, y=B if x>=A	Step model
model_two_steps	4	see model_step	2 step models. Parameter initialization is aware of # of steps.
model_sigmoid_step	3	y = A + (B - A) / (1 + np.exp(- D * (x - C)))	Sigmoid step model
model_sigmoid	3	y = A + (B - A) / (1 + np.exp(- D * (x - C)))	Sigmoid model
model_season_wday_2	2	see desc.	Weekend seasonality model. Assigns a constant to each of weekday/weekend
model_season_month	11	see desc.	Month seasonality model. Assigns a constant value to each month

If the available range of models isn't a good match for your data, it is also possible to define new models using *anticipy.forecast_models.ForecastModel*

3.5 Models with dummy variables

You can use *anticipy.forecast_models.get_model_dummy()* to get a model based on a dummy variable. This model returns a constant value when the dummy variable is 1, and 0 otherwise:

```
# Example dummy model - check if date matches specific dates in list
model_dummy_l_date = forecast_models.get_model_dummy('dummy_l_date', ['2017-12-22',
↪'2017-12-27'])

# Example dummy model - checks if it is Christmas
model_dummy_christmas = forecast_models.get_model_dummy('dummy_christmas',
                                        lambda a_x, a_date: ((a_date.month == 12) & (a_
↪date.day == 25)).astype(float))
```

(continues on next page)

(continued from previous page)

```
a_x = np.arange(0,10)
a_date = pd.date_range('2017-12-21','2017-12-30')
params = np.array([10.]) # A=10

print model_dummy_1_date(a_x, a_date, params)
print model_dummy_christmas(a_x, a_date, params)
```

Output:

```
[ 0. 10.  0.  0.  0.  0. 10.  0.  0.  0.]
[ 0.  0.  0.  0. 10.  0.  0.  0.  0.  0.]
```

Dummy variables can be very useful when used in composition with simpler models. A common application is to check for bank holidays or other special dates. The following example uses a dummy variable to improve fit in a linear time series with a spike on Christmas:

```
df=pd.DataFrame({'y': 100+np.arange(0,6)+np.array([0.,0.,0.,0.,50.,0.,])},
            index=pd.date_range('2017-12-21','2017-12-26'))

# Example dummy model - checks if it is Christmas
model_dummy_christmas = forecast_models.get_model_dummy('dummy_christmas',
                                        lambda a_x, a_date: ((a_date.month == 12) & (a_
↪date.day == 25)).astype(float))

dict_result = forecast.run_forecast(df, extrapolate_years=1, simplify_output=False,
                                    l_model_trend = [forecast_models.model_linear,
                                                     forecast_models.model_linear+model_
↪dummy_christmas],
                                    include_all_fits=True)

print dict_result['metadata'][['source','model','is_fit','cost','aic_c','params_str','is_
↪best_fit']]
```

Output:

```
. source                          model  is_fit          cost       aic_c        params_
↪str  is_best_fit
0    src                         linear    True  8.809524e+02   37.935465       [ 5.3 97.
↪6]        False
1    src  (linear_add_dummy_christmas)    True  9.980807e-20 -255.256784  [  1. 100.  50.
↪]         True
```

3.6 Outlier Detection

If you call *anticipy.forecast.run_forecast()* and specify as input *find_outliers=True*, it will try to automatically identify any outliers exist in the input Series. The weight for these samples is set to 0, so that they are ignored by the forecast logic.

Example:

```
a_y = [19.8, 19.9, 20.0, 20.1, 20.2, 20.3, 20.4, 20.5,
       20.6, 10., 20.7, 20.8, 20.9, 21.0,
       21.1, 21.2, 21.3, 21.4, 21.5]
a_date = pd.date_range(start='2018-01-01', periods=len(a_y), freq='D')
df_spike = pd.DataFrame({'y': a_y})

dict_result = forecast.run_forecast(df_spike, find_outliers=True,
                                    simplify_output=False, include_all_fits=True,
                                    season_add_mult='add')
df_data = dict_result['data']
# Subset of output - shows that the sample with a spike now has weight=0, and is ignored␣
↪by forecast
df_weighted_actuals = df_data.loc[df_data.model=='actuals'][['y','weight']]
```

Output:

```
.       y  weight
0   19.8     1.0
1   19.9     1.0
2   20.0     1.0
3   20.1     1.0
4   20.2     1.0
5   20.3     1.0
6   20.4     1.0
7   20.5     1.0
8   20.6     1.0
9   10.0     0.0
10  20.7     1.0
11  20.8     1.0
12  20.9     1.0
13  21.0     1.0
14  21.1     1.0
15  21.2     1.0
16  21.3     1.0
17  21.4     1.0
18  21.5     1.0
```

CHAPTER

FOUR

LIBRARY REFERENCE

4.1 forecast

Functions to run forecast

anticipy.forecast.**get_residuals**(*params*, *model*, *a_x*, *a_y*, *a_date*, *a_weights=None*, *df_actuals=None*, ***kwargs*)

Given a time series, a model function and a set of parameters, get the residuals

Parameters

- **params** (*numpy array of floats*) – parameters for model function
- **model** (*function or ForecastModel instance*) – model function. Usage: model(a_x, a_date, params)
- **a_x** (*numpy array of floats*) – X axis for model function.
- **a_y** (*numpy array of floats*) – Input time series values, to compare to the model function
- **a_date** (*numpy array of datetimes*) – Dates for the input time series
- **a_weights** (*numpy array of floats*) – weights for each individual sample
- **df_actuals** (*pandas DataFrame*) – The original dataframe with actuals data. Not required for regression but used by naive models

Returns
array with residuals, same length as a_x, a_y

Return type
numpy array of floats

anticipy.forecast.**optimize_least_squares**(*model*, *a_x*, *a_y*, *a_date*, *a_weights=None*, *df_actuals=None*, *use_cache=True*)

Given a time series and a model function, find the set of parameters that minimises residuals

Parameters

- **model** (*function*) – model function, to be fitted against the actuals
- **a_x** (*numpy array of floats*) – X axis for model function.
- **a_y** (*numpy array of floats*) – Input time series values, to compare to the model function
- **a_date** (*numpy array of datetimes*) – Dates for the input time series

- **a_weights** (*numpy array of floats*) – weights for each individual sample
- **df_actuals** (*pandas DataFrame*) – The original dataframe with actuals data. Not required for regression but used by naive models
- **use_cache** (*bool*) – If true, save some model variables to cache when fitting

Returns

table(success, params, cost, optimality,
iterations, status, jac_evals, message):

- success (bool): True if successful fit
- params (list): Parameters of fitted model
- cost (float): Value of cost function
- optimality(float)
- iterations (int) : Number of function evaluations
- status (int) : Status code
- jac_evals(int) : Number of Jacobian evaluations
- message (str) : Output message

Return type
pandas.DataFrame

anticipy.forecast.**normalize_df**(*df_y*, *col_name_y='y'*, *col_name_weight='weight'*, *col_name_x='x'*, *col_name_date='date'*, *col_name_source='source'*)

Converts an input dataframe for run_forecast() into a normalized format suitable for fit_model()

Parameters

- **df_y** (*pandas.DataFrame*) – unformatted input dataframe, for use by run_forecast()
- **col_name_y** (*basestring*) – name for column with time series values
- **col_name_weight** (*basestring*) – name for column with time series weights
- **col_name_x** (*basestring*) – name for column with time series indices
- **col_name_date** (*basestring*) – name for column with time series dates
- **col_name_source** (*basestring*) – name for column with time series source identifiers

Returns
formatted input dataframe, for use by run_forecast()

Return type
pandas.DataFrame

anticipy.forecast.**fit_model**(*model*, *df_y*, *freq='W'*, *source='test'*, *df_actuals=None*, *use_cache=True*)

Given a time series and a model, optimize model parameters and return

Parameters

- **model** (*function or ForecastModel instance*) – model function. Usage: model(a_x, a_date, params)
- **df_y** (*pandas.DataFrame*) –

  Dataframe with the following columns:

- y:
- date: (optional)
- weight: (optional)
- x: (optional)

- **source** (*basestring*) – source identifier for this time series
- **freq** (*basestring*) – 'W' or 'D' . Used only for metadata
- **df_actuals** (*pandas DataFrame*) – The original dataframe with actuals data. Not required for regression but used by naive models
- **use_cache** (*bool*) – If true, save some model variables to cache when fitting

Returns
table (source, model_name, y_weights , freq, is_fit, aic_c, params)

Return type
pandas.DataFrame

This function calls optimize_least_squares() to perform the optimization loop. It performs some cleaning up of input and output parameters.

anticipy.forecast.**extrapolate_model**(*model*, *params*, *date_start_actuals*, *date_end_actuals*, *freq='W'*, *extrapolate_years=2.0*, *x_start_actuals=0.0*, *df_actuals=None*)

Given a model and a set of parameters, generate model output for a date range plus a number of additional years.

Parameters

- **model** (*function or ForecastModel instance*) – model function. Usage: model(a_x, a_date, params)
- **params** (*numpy array of floats*) – parameters for model function
- **date_start_actuals** (*str, datetime, int or float*) – date or numeric index for first actuals sample
- **date_end_actuals** (*str, datetime, int or float*) – date or numeric index for last actuals sample
- **freq** (*basestring*) – Time unit between samples. Supported units are 'W' for weekly samples, or 'D' for daily samples. (untested) Any date unit or time unit accepted by numpy should also work, see https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.datetime.html#arrays-dtypes-dateunits # noqa
- **extrapolate_years** (*float*) – Number of years (or fraction of year) covered by the generated time series, after the end of the actuals
- **x_start_actuals** –
- **df_actuals** (*pandas DataFrame*) – The original dataframe with actuals data. Not required for regression but used by naive models

Returns
dataframe with a time series extrapolated from the model function

Return type
pandas.DataFrame, with an 'y' column of floats

anticipy.forecast.**get_list_model**(*l_model_trend*, *l_model_season*, *season_add_mult='both'*)

Generate a list of composite models from lists of trend and seasonality models

Parameters

- **l_model_trend** (*list of ForecastModel*) – list of trend models
- **l_model_season** (*list of ForecastModel*) – list of seasonality models
- **season_add_mult** (*basestring*) – 'mult', 'add' or 'both', for multiplicative/additive composition (or both types)

Returns

Return type
list of *ForecastModel*

All combinations of possible composite models are included

anticipy.forecast.**get_df_actuals_clean**(*df_actuals*, *source*, *source_long*)

Convert an actuals dataframe to a clean format

Parameters

- **df_actuals** (*pandas.DataFrame*) – dataframe in normalized format, with columns y and optionally x, date, weight
- **source** (*basestring*) – source identifier for this time series
- **source_long** (*basestring*) – long-format source identifier for this time series

Returns
clean actuals dataframe

Return type
pandas.DataFrame

anticipy.forecast.**run_forecast**(*df_y*, *l_model_trend=None*, *l_model_season=None*, *date_start_actuals=None*, *source_id='src'*, *col_name_y='y'*, *col_name_weight='weight'*, *col_name_x='x'*, *col_name_date='date'*, *col_name_source='source'*, *extrapolate_years=0*, *season_add_mult='add'*, *include_all_fits=False*, *simplify_output=True*, *find_outliers=False*, *l_season_yearly=None*, *l_season_weekly=None*, *verbose=None*, *l_model_naive=None*, *l_model_calendar=None*, *n_cum=None*, *pi_q1=5*, *pi_q2=20*, *pi_widening_freq='Y'*, *use_cache=True*)

Generate forecast for one or more input time series

Parameters

- **df_y** (*pandas.DataFrame*) –

  input dataframe with the following columns:
  - Mandatory: a value column, with the time series values
  - Optional: weight column, source ID column, index column, date column

- **l_model_trend** (*list of ForecastModel*) – list of trend models
- **l_model_season** (*list of ForecastModel*) – list of seasonality models
- **date_start_actuals** (*str, datetime, int or float*) – date or numeric index for first actuals sample to be used for forecast. Previous samples are ignored
- **source_id** (*basestring*) – source identifier for time series, if source column is missing
- **col_name_y** (*basestring*) – name for column with time series values
- **col_name_weight** (*basestring*) – name for column with time series weights

- **col_name_x** (*basestring*) – name for column with time series indices
- **col_name_date** (*basestring*) – name for column with time series dates
- **col_name_source** (*basestring*) – name for column with time series source identifiers
- **extrapolate_years** (*float*) – Number of years (or fraction of year) covered by the forecast, after the end of the actuals
- **season_add_mult** (*str*) – 'add', 'mult', or 'both'. Whether forecast seasonality will be additive, multiplicative, or the best fit of the two.
- **find_outliers** (*bool*) – If True, find outliers in input data, ignore outlier samples in forecast
- **include_all_fits** (*bool*) – If True, also include non-optimal models in output
- **simplify_output** (*bool*) – If False, return dict with forecast and metadata. Otherwise, return only forecast.
- **l_season_yearly** (*list of* ForecastModel) – yearly seasonality models to consider in automatic seasonality detection
- **l_season_weekly** (*list of* ForecastModel) – yearly seasonality models to consider in automatic seasonality detection
- **verbose** (*bool*) – If True, enable verbose logging
- **l_model_naive** (*list of* ForecastModel) – list of naive models to consider for forecast. Naive models are not fitted with regression, they are based on the last actuals samples
- **l_model_calendar** (*list of* ForecastModel) – list of calendar models to consider for forecast, to handle holidays and calendar-based events
- **n_cum** (*int*) – Used for widening prediction interval. Interval widens every n_sims samples.
- **pi_q1** (*int*) – Percentile for outer prediction interval (defaults to 5%-95%)
- **pi_q2** (*int*) – Percentile for inner prediction interval (defaults to 20%-80%)
- **pi_widening_freq** (*str*) – Specifies the frequency with which the prediction interval widens: Y (yearly), M (monthly), W (weekly), D (daily). This parameters is deprecated - use pi_widening_freq instead
- **use_cache** (*bool*) – If true, save some model variables to cache when fitting

Returns

With simplify_output=False, returns a dictionary with 4 dataframes:
- forecast: output time series with prediction interval
- data: output time series. If include_all_fits, includes all fitting
  models
- metadata: forecast metadata table
- optimize_info: debugging metadata from scipy.optimize

With simplify_output=True, returns the 'forecast' dataframe,
as described above

Return type
pandas.DataFrame or dict of pandas.DataFrames

anticipy.forecast.**aggregate_forecast_dict_results**(*l_dict_result*)

Aggregates a list of dictionaries with forecast outputs into a single dictionary

Parameters
l_dict_result (*list of dictionaries*) – list with outputs dictionaries from run_forecast_single

Returns
aggregated dictionary

Return type
dict

anticipy.forecast.**run_forecast_single**(*df_y*, *l_model_trend=None*, *l_model_season=None*, *date_start_actuals=None*, *source_id='src'*, *extrapolate_years=0*, *season_add_mult='add'*, *include_all_fits=False*, *simplify_output=True*, *find_outliers=False*, *l_season_yearly=None*, *l_season_weekly=None*, *l_model_naive=None*, *l_model_calendar=None*, *n_cum=1*, *pi_q1=5*, *pi_q2=20*, *pi_widening_freq=None*, *use_cache=True*)

Generate forecast for one input time series

Parameters

- **df_y** (*pandas.DataFrame*) –

  input dataframe with the following columns:
  - y: time series values
  - x: time series indices
  - weight: time series weights (optional)
  - date: time series dates (optional)

- **l_model_trend** (*list of* ForecastModel) – list of trend models
- **l_model_season** (*list of* ForecastModel) – list of seasonality models
- **date_start_actuals** (*str, datetime, int or float*) – date or numeric index for first actuals sample to be used for forecast. Previous samples are ignored
- **source_id** (*basestring*) – source identifier for time series
- **extrapolate_years** (*float*) –
- **season_add_mult** (*str*) – 'add', 'mult', or 'both'. Whether forecast seasonality will be additive, multiplicative, or the best fit of the two.
- **include_all_fits** (*bool*) – If True, also include non-optimal models in output
- **simplify_output** (*bool*) – If False, return dict with forecast and metadata. Otherwise, return only forecast.
- **find_outliers** (*bool*) – If True, find outliers in input data, ignore outlier samples in forecast
- **l_season_yearly** (*list of* ForecastModel) – yearly seasonality models to consider in automatic seasonality detection
- **l_season_weekly** (*list of* ForecastModel) – yearly seasonality models to consider in automatic seasonality detection
- **l_model_naive** (*list of* ForecastModel) – list of naive models to consider for forecast. Naive models are not fitted with regression, they are based on the last actuals samples

- **l_model_calendar** (*list of* ForecastModel) – list of calendar models to consider for forecast, to handle holidays and calendar-based events
- **n_cum** (*int*) – Used for widening prediction interval. Interval widens every n_sims samples. This parameters is deprecated - use pi_widening_freq instead
- **pi_q1** (*int*) – Percentile for outer prediction interval (defaults to 5%-95%)
- **pi_q2** (*int*) – Percentile for inner prediction interval (defaults to 20%-80%)
- **widening_freq** – Specifies the frequency with which the prediction interval widens: Y (yearly), M (monthly), W (weekly), D (daily).
- **use_cache** (*bool*) – If true, save some model variables to cache when fitting

Returns

With simplify_output=False, returns a dictionary with 4 dataframes:
- forecast: output time series with prediction interval
- data: output time series. If include_all_fits, includes all fitting models
- metadata: forecast metadata table
- optimize_info: debugging metadata from scipy.optimize

With simplify_output=True, returns the 'forecast' dataframe, as described above

Return type
pandas.DataFrame or dict of pandas.DataFrames

anticipy.forecast.**run_l_forecast**(*l_fcast_input*, *col_name_y='y'*, *col_name_weight='weight'*, *col_name_x='x'*, *col_name_date='date'*, *col_name_source='source'*, *extrapolate_years=0*, *season_add_mult='add'*, *include_all_fits=False*, *find_outliers=False*, *use_cache=True*)

Generate forecasts for a list of SolverConfig objects, each including a time series, model functions, and other configuration parameters.

Parameters

- **l_fcast_input** (*list of* ForecastInput) – List of forecast input configurations. Each element includes a time series, candidate forecast models for trend and seasonality, and other configuration parameters. For each input configuration, a forecast time series will be generated.
- **return_all_models** (*bool*) –

  If True, result includes non-fitting models, with null AIC and an empty forecast df. Otherwise, result includes only fitting models, and for time series where no fitting model is available, a 'no-best-model' entry with null AIC and an empty forecast df is added.
- **return_all_fits** (*bool*) – If True, result includes all models for each nput time series. Otherwise, only the best model is included.
- **extrapolate_years** (*float*) –

- **season_add_mult** (*str*) – 'add', 'mult', or 'both'. Whether forecast seasonality will be additive, multiplicative, or the best fit of the two.
- **fill_gaps_y_values** (*bool*) – If True, gaps in time series will be filled with NaN values
- **freq** (*str*) – 'W' or 'D' . Sampling frequency of the output forecast: weekly or daily.
- **use_cache** (*bool*) – If true, save some model variables to cache when fitting

Returns

dict(data,metadata)
data: dataframe(date, source, model, y)
metadata: dataframe('source', 'model', 'res_weights', 'freq',
'is_fit', 'cost', 'aic_c', 'params', 'status')

Return type
dict

class anticipy.forecast.**ForecastInput**(*source_id*, *df_y*, *l_model_trend=None*, *l_model_season=None*, *weights_y_values=1.0*, *date_start_actuals=None*)

Class that encapsulates input variables for forecast.run_forecast()

anticipy.forecast.**get_pi**(*df_forecast*, *n_sims=100*, *n_cum=None*, *pi_q1=5*, *pi_q2=20*, *widening_freq='Y'*)

Generate prediction intervals for a table with multiple forecasts, using bootstrapped residuals.

Parameters

- **df_forecast** (*pandas.DataFrame*) – forecasted time series
- **n_sims** (*int*) – Number of bootstrapped samples for prediction interval
- **n_cum** (*int*) – Used for widening prediction interval. Interval widens every n_sims samples. This parameters is deprecated - use widening_freq instead
- **pi_q1** (*int*) – Percentile for outer prediction interval (defaults to 5%-95%)
- **pi_q2** (*int*) – Percentile for inner prediction interval (defaults to 20%-80%)
- **widening_freq** (*str*) – Specifies the frequency with which the prediction interval widens: Y (yearly), M (monthly), W (weekly), D (daily).

Returns

Forecast time series table with added columns:
- q5: 5% percentile of prediction interval
- q5: 20% percentile of prediction interval
- q5: 80 percentile of prediction interval
- q5: 95% percentile of prediction interval

Return type
pandas.DataFrame

Based on https://otexts.org/fpp2/prediction-intervals.html

4.2 forecast_models

Defines the ForecastModel class, which encapsulates model functions used in forecast model fitting, as well as their number of parameters and initialisation parameters.

class anticipy.forecast_models.**ForecastModel**(*name*, *n_params*, *f_model*, *f_init_params=None*, *f_bounds=None*, *l_f_validate_input=None*, *l_cache_vars=None*, *dict_f_cache=None*)

Class that encapsulates model functions for use in forecasting, as well as their number of parameters and functions for parameter initialisation.

A ForecastModel instance is initialized with a model name, a number of model parameters, and a model function. Class instances are callable - when called as a function, their internal model function is used. The main purpose of ForecastModel objects is to generate predicted values for a time series, given a set of parameters. These values can be compared to the original series to get an array of residuals:

```
y_predicted = model(a_x, a_date, params)
residuals = (a_y - y_predicted)
```

This is used in an optimization loop to obtain the optimal parameters for the model.

The reason for using this class instead of raw model functions is that ForecastModel supports function composition:

```
model_sum = fcast_model1 + fcast_model2
# fcast_model 1 and 2 are ForecastModel instances, and so is model_sum
a_y1 = fcast_model1(
    a_x, a_date, params1) + fcast_model2(a_x, a_date, params2)
params = np.concatenate([params1, params2])
a_y2 = model_sum(a_x, a_date, params)
a_y1 == a_y2  # True
```

Forecast models can be added or multiplied, with the + and * operators. Multiple levels of composition are supported:

```
model = (model1 + model2) * model3
```

Model composition is used to aggregate trend and seasonality model components, among other uses.

Model functions have the following signature:

- f(a_x, a_date, params, is_mult)
- a_x : array of floats
- a_date: array of dates, same length as a_x. Only required for date-aware models, e.g. for weekly seasonality.
- params: array of floats - model parameters - the optimisation loop updates this to fit our actual values. Each model function uses a fixed number of parameters.
- is_mult: boolean. True if the model is being used with multiplicative composition. Required because some model functions (e.g. steps) have different behaviour when added to other models than when multiplying them.
- returns an array of floats - with same length as a_x - output of the model defined by this object's modelling function f_model and the current set of parameters

By default, model parameters are initialized as random values between 0 and 1. It is possible to define a parameter initialization function that picks initial values based on the original time series. This is passed during

ForecastModel creation with the argument f_init_params. Parameter initialization is compatible with model composition: the initialization function of each component will be used for that component's parameters.

Parameter initialisation functions have the following signature:

- f_init_params(a_x, a_y, is_mult)
- a_x: array of floats - same length as time series
- a_y: array of floats - time series values
- returns an array of floats - with length equal to this object's n_params value

By default, model parameters have no boundaries. However, it is possible to define a boundary function for a model, that sets boundaries for each model parameter, based on the input time series. This is passed during ForecastModel creation with the argument f_bounds. Boundary definition is compatible with model composition: the boundary function of each component will be used for that component's parameters.

Boundary functions have the following signature:

- f_bounds(a_x, a_y, a_date)
- a_x: array of floats - same length as time series
- a_y: array of floats - time series values
- a_date: array of dates, same length as a_x. Only required for date-aware models, e.g. for weekly seasonality.
- returns a tuple of 2 arrays of floats. The first defines minimum parameter boundaries, and the second the maximum parameter boundaries.

As an option, we can assign a list of input validation functions to a model. These functions analyse the inputs that will be used for fitting a model, returning True if valid, and False otherwise. The forecast logic will skip a model from fitting if any of the validation functions for that model returns False.

Input validation functions have the following signature:

- f_validate_input(a_x, a_y, a_date)
- See the description of model functions above for more details on these parameters.

Our input time series should meet the following constraints:

- Minimum required samples depends on number of model parameters
- May include null values
- May include multiple values per sample
- A date array is only required if the model is date-aware

Class Usage:

```
model_x = ForecastModel(name, n_params, f_model, f_init_params,
l_f_validate_input)
# Get model name
model_name = model_x.name
# Get number of model parameters
n_params = model_x.n_params
# Get parameter initialisation function
f_init_params = model_x.f_init_params
# Get initial parameters
init_params = f_init_params(t_values, y_values)
# Get model fitting function
```

(continues on next page)

(continued from previous page)

```
f_model = model_x.f_model
# Get model output
y = f_model(a_x, a_date, parameters)
```

The following pre-generated models are available. They are available as attributes from this module: # noqa

Table 1: Forecast models

name	params	formula	notes
model_null	0	y=0	Does nothing. Used to disable components (e.g. seasonality)
model_constant	1	y=A	Constant model
model_linear	2	y=Ax + B	Linear model
model_linear_nondec	2	y=Ax + B	Non decreasing linear model. With boundaries to ensure model slope >=0
model_quasilinear	3	y=A*(x^B) + C	Quasilinear model
model_exp	2	y=A * B^x	Exponential model
model_decay	4	Y = A * e^(B*(x-C)) + D	Exponential decay model
model_step	2	y=0 if x<A, y=B if x>=A	Step model
model_two_steps	4	see model_step	2 step models. Parameter initialization is aware of # of steps.
model_sigmoid_step	3	y = A + (B - A) / (1 + np.exp(- D * (x - C)))	Sigmoid step model
model_sigmoid	3	y = A + (B - A) / (1 + np.exp(- D * (x - C)))	Sigmoid model
model_season_wday	7	see desc.	Weekday seasonality model. Assigns a constant value to each weekday
model_season_wday	6	see desc.	6-param weekday seasonality model. As above, with one constant set to 0.
model_season_wday_2	2	see desc.	Weekend seasonality model. Assigns a constant to each of weekday/weekend
model_season_month	12	see desc.	Month seasonality model. Assigns a constant value to each month
model_season_fourier_yearly	10	see desc	Fourier yearly seasonality model

anticipy.forecast_models.**get_model_outliers**(*df*, *window=3*)

Identify outlier samples in a time series

Parameters

- **df** (*pandas.DataFrame*) – Input time series
- **window** (*int*) – The x-axis window to aggregate multiple steps/spikes

Returns

tuple (mask_step, mask_spike)
mask_step: True if sample contains a step
mask_spike: True if sample contains a spike

Return type
tuple of 2 numpy arrays of booleans

TODO: require minimum number of samples to find an outlier

anticipy.forecast_models.**get_model_dummy**(*name*, *dummy*, ***kwargs*)

Generate a model based on a dummy variable.

Parameters

- **name** (*basestring*) – Name of the model
- **dummy** (*function, or list-like of numerics or datetime-likes*) –

  Can be a function or a list-like.
  If a function, it must be of the form f_dummy(a_x, a_date),
  and return a numpy array of floats
  with the same length as a_x and values that are either 0 or 1.
  If a list-like of numerics, it will be converted to a f_dummy function
  as described above, which will have values of 1 when a_x has one of
  the values in the list, and 0 otherwise. If a list-like of date-likes,
  it will be converted to a f_dummy function as described above, which
  will have values of 1 when a_date has one of the values in the list,
  and 0 otherwise.

- **kwargs** –

Returns

A model that returns A when dummy is 1, and 0 (or 1 if is_mult==True)
otherwise.

Return type
ForecastModel

class anticipy.forecast_models.**CalendarBankHolUK**(*name=None*, *rules=None*)

class anticipy.forecast_models.**CalendarChristmasUK**(*name=None*, *rules=None*)

class anticipy.forecast_models.**CalendarBankHolIta**(*name=None*, *rules=None*)

class anticipy.forecast_models.**CalendarChristmasIta**(*name=None*, *rules=None*)

anticipy.forecast_models.**get_model_from_calendars**(*l_calendar*, *name=None*)

Create a ForecastModel based on a list of pandas Calendars.

Parameters
calendar (*pandas.tseries.AbstractHolidayCalendar*) –

Returns
model based on the input calendar

Return type
ForecastModel

In pandas, Holidays and calendars provide a simple way to define holiday rules, to be used in any analysis that requires a predefined set of holidays. This function converts a Calendar object into a ForecastModel that assigns a parameter to each calendar rule.

As an example, a Calendar with 1 rule defining Christmas dates generates a model with a single parameter, which determines the amount added/multiplied to samples falling on Christmas. A calendar with 2 rules for Christmas and New Year will have two parameters - the first one applying to samples in Christmas, and the second one applying to samples in New Year.

Usage:

```
from pandas.tseries.holiday import USFederalHolidayCalendar
model_calendar = get_model_from_calendar(USFederalHolidayCalendar())
```

anticipy.forecast_models.**get_model_from_datelist**(*name=None*, **args*)

Create a ForecastModel based on one or more lists of dates.

Parameters

- **name** (*str*) – Model name
- **args** – Each element in args is a list of dates.

Returns

model based on the input lists of dates

Return type

ForecastModel

Usage:

```
model_datelist1=get_model_from_date_list('datelist1',
                                         [date1, date2, date3])
model_datelists23 = get_model_from_date_list('datelists23',
                                        [date1, date2], [date3, date4])
```

In the example above, model_datelist1 will have one parameter, which determines the amount added/multiplied to samples with dates matching either date1, date2 or date3. model_datelists23 will have two parameters - the first one applying to samples in date1 and date2, and the second one applying to samples in date 3 and date4

anticipy.forecast_models.**fix_params_fmodel**(*forecast_model*, *l_params_fixed*)

Given a forecast model and a list of floats, modify the model so that some of its parameters become fixed

Parameters

- **forecast_model** (ForecastModel) – Input model
- **l_params_fixed** (*list*) – List of floats with same length as number of parameters in model. For each element, a non-null value means that the parameter in that position is fixed to that value. A null value means that the parameter in that position is not fixed.

Returns

A forecast model with a number of parameters equal to the number of null values in l_params_fixed, with f_model modified so that some of its parameters gain fixed values equal to the non-null values in l_params

Return type

ForecastModel

anticipy.forecast_models.**simplify_model**(*f_model*, *a_x=None*, *a_y=None*, *a_date=None*)

Check a model's bounds, and update model to make parameters fixed if their min and max bounds are equal

Parameters

- **f_model** (ForecastModel) – Input model

- **a_x** (*numpy array of floats*) – X axis for model function.
- **a_y** (*numpy array of floats*) – Input time series values, to compare to the model function
- **a_date** (*numpy array of datetimes*) – Dates for the input time series

Returns
Model with simplified parameters based on bounds

Return type
ForecastModel

anticipy.forecast_models.**get_l_model_auto_season**(*a_date*, *min_periods=1.5*, *season_add_mult='add'*, *l_season_yearly=None*, *l_season_weekly=None*)

Generates a list of candidate seasonality models for an series of timestamps

Parameters

- **a_date** (*numpy array of timestamps*) – date array of a time series
- **min_periods** (*float*) – Minimum number of periods required to apply seasonality
- **season_add_mult** – 'add' or 'mult'

Returns
list of candidate seasonality models

Return type
list of *ForecastModel*

4.3 model_utils

Utility functions for model generation

anticipy.model_utils.**array_transpose**(*a*)

Transpose a 1-D numpy array

Parameters
a (*numpy.Array*) – An array with shape (n,)

Returns
The original array, with shape (n,1)

Return type
numpy.Array

anticipy.model_utils.**model_requires_scaling**(*model*)

Given a *anticipy.forecast_models.ForecastModel*
return True if the function requires scaling a_x

Parameters
model (*function*) – A get_model_<modeltype> function from `anticipy.model.periodic_models` or `anticipy.model.aperiodic_models`

Returns
True if function is logistic or sigmoidal

Return type
bool

anticipy.model_utils.**apply_a_x_scaling**(*a_x*, *model=None*, *scaling_factor=100.0*)

Modify a_x for forecast_models that require it

Parameters

- **a_x** (*numpy array*) – x axis of time series
- **model** (*function or None*) – a *anticipy.forecast_models.ForecastModel*
- **scaling_factor** (*float*) – Value used for scaling t_values for logistic models

Returns
a_x with scaling applied, if required

Return type
numpy array

anticipy.model_utils.**get_normalized_x_from_date**(*s_date*)

Get column of days since Monday of first date

anticipy.model_utils.**get_s_x_extrapolate**(*date_start_actuals*, *date_end_actuals*, *model=None*, *freq=None*, *extrapolate_years=2.5*, *scaling_factor=100.0*, *x_start_actuals=0.0*)

Return a_x series with DateTimeIndex, covering the date range for the
actuals, plus a forecast period.

Parameters

- **date_start_actuals** (*str, datetime, int or float*) – date or numeric index for first actuals sample
- **date_end_actuals** (*str, datetime, int or float*) – date or numeric index for last actuals sample
- **extrapolate_years** (*float*) –
- **model** (*function*) –
- **freq** (*basestring*) – Time unit between samples. Supported units are 'W' for weekly samples, or 'D' for daily samples. (untested) Any date unit or time unit accepted by numpy should also work, see https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.datetime.html#arrays-dtypes-dateunits # noqa
- **shifted_origin** (*int*) – Offset to apply to a_x
- **scaling_factor** (*float*) – Value used for scaling a_x for certain model functions
- **x_start_actuals** (*int*) – numeric index for the first actuals sample

Returns
Series of floats with DateTimeIndex. To be used as (a_date, a_x) input for a model function.

Return type
pandas.Series

The returned series covers the actuals time domain plus a forecast period lasting extrapolate_years, in years. The number of additional samples for the forecast period is time_resolution * extrapolate_years, rounded down

anticipy.model_utils.**get_aic_c**(*fit_error*, *n*, *n_params*)

This function implements the corrected Akaike Information Criterion (AICc) taking as input a given fit error and data/model degrees of freedom. We assume that the residuals of the candidate model are distributed according to independent identical normal distributions with zero mean. Hence, we can use define the AICc as

$$AICc = AIC + \frac{2k(k+1)}{n-k-1} = 2k + n\log\left(\frac{E}{n}\right) + \frac{2k(k+1)}{n-k-1},$$

where k and n denotes the model and data degrees of freedom respectively, and E denotes the residual error of the fit.

Parameters

- **fit_error** (*float*) – Residual error of the fit
- **n** (*int*) – Data degrees of freedom
- **n_params** (*int*) – Model degrees of freedom

Returns
Corrected Akaike Information Criterion (AICc)

Return type
float

Note:

- see AIC in Wikipedia article on the AIC.

anticipy.model_utils.**is_multiplicative**(*df*, *freq='M'*)

For an input time series, check if model composition should be multiplicative.

Return True if multiplicative is best - otherwise, use additive composition.

We assume multiplicative composition is best if variance correlates heavily (>0.8) with mean. We aggregate data on a monthly basis by default for this analysis. Use

The following exceptions apply:

- If any time series value is <=0, use additive
- If date information is unavailable (only x column), use additive
- If less than 2 periods worth of data are available, use additive

4.4 forecast_plot

Functions to plot forecast outputs

anticipy.forecast_plot.**plot_forecast**(*df_fcast*, *output='html'*, *path=None*, *width=None*, *height=None*, *title=None*, *dpi=70*, *show_legend=True*, *auto_open=False*, *include_interval=False*, *pi_q1=5*, *pi_q2=20*)

Generates matplotlib or plotly plot and saves it respectively as png or html

Parameters

- **df_fcast** (*pandas.DataFrame*) –

  Forecast Dataframe with the following columns:
  - date (timestamp)
  - model (str) : ID for the forecast model

- y (float) : Value of the time series in that sample
- is_actuals (bool) : True for actuals samples, False for forecast

- **output** (*basestring*) – Indicates the output type (html=Default, png or jupyter)
- **path** (*basestring*) – File path for output
- **width** (*int*) – Image width, in pixels
- **height** (*int*) – Image height, in pixels
- **title** (*basestring*) – Plot title
- **dpi** (*int*) – Image dpi
- **show_legend** (*bool*) – Indicates whether legends will be displayed
- **auto_open** (*bool*) – Indicates whether the output will be displayed automatically
- **pi_q1** (*int*) – Percentile for outer prediction interval (defaults to 5%-95%)
- **pi_q2** (*int*) – Percentile for inner prediction interval (defaults to 20%-80%)

Returns
Success or failure code.

Return type
int

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

R

S