TADA on univariate time series

March 17, 2021

E.Bachiorrini, C.Canavaggio, N.Ronzoni

The project have the following section:

- Context: What is TADA.
- Problem: Goal of the project, use Tada to forecast univariate time series data.
- Algorithms: After having select the best window size, create the features and fit them into the supported dataset format of Tada.
- Experiment: Compare Tada forecast with Arima model for different interval in the future.
- Results: Results of the experiment for each time series analyzed and conclusion.

1 Context

Machine Learning (ML) models can be generated with artificial neural network (ANN), deep learning (DL), but also – like in the case of MyDataModels – using Evolutionary Programming (EP) and Genetic Algorithms (GA). These lasts principles are implemented in TADA algorithm. A 'Genetic Algorithm' (GA) is a search-based optimization technique used to find optimal solutions to 'difficult' problems, otherwise long to solve. How (EP) and (GA) work:

1. A "pool" or a "population" of possible solutions to a given problem, is generated randomly.
2. A fitness function is established for each.
3. A parent selection mechanisms among the population is performed to allow recombination ("crossover") in order to produce new children ("the offspring").
4. Each individual (or candidate solution) is assigned a fitness function. The fitter individuals are given a higher chance to mate and generate more "fitter" individuals...("recombination or crossover").
5. Mutation is a random variation imposed to the individuals which contributes to additional 'variability'.
6. The process is repeated over various generations (often the stop criterion is a fixed number of "generations").

By defining an early convergence, the outcome is given with mathematical formulae which include the variables from the original dataset, thus, models become explainable.

An introductionary explanation of how TADA works could be find here https://vimeo.com/514264723

2 Problem

The algorithm in TADA was originally designed to perform Machine Learning predictions in Classification and Regression tasks – thus not forecasting. The aim of this project it is preliminarily tested TADA algorithm on real world uniformly spaced time series and then compare the forecasts against traditional techniques like ARIMA. However some pre-process steps are needed in order to:

- Compute the data transformation to fit the univariate time series in a supported Machine learning format.
- Define Useful features for the target.

In addition traditional train-test split, is not applicable 'as-is' in this instance, it ignores temporal component inherent to the problem. We must split data up and respect the temporal order in which values were observed.

3 Algorithms

Given a sequence of numbers for a time series dataset, we can restructure data to look like a supervised learning problem. We can do this by using previous time steps as input variables and use the next time step as the output variable. In order to pre-process time series and convert it to a form that can be fed into standard machine learning regression algorithm, such as TADA, preliminary step must be done. We must define a window size of data, in other words, how many time steps should be used as input variables to predict the output variable?

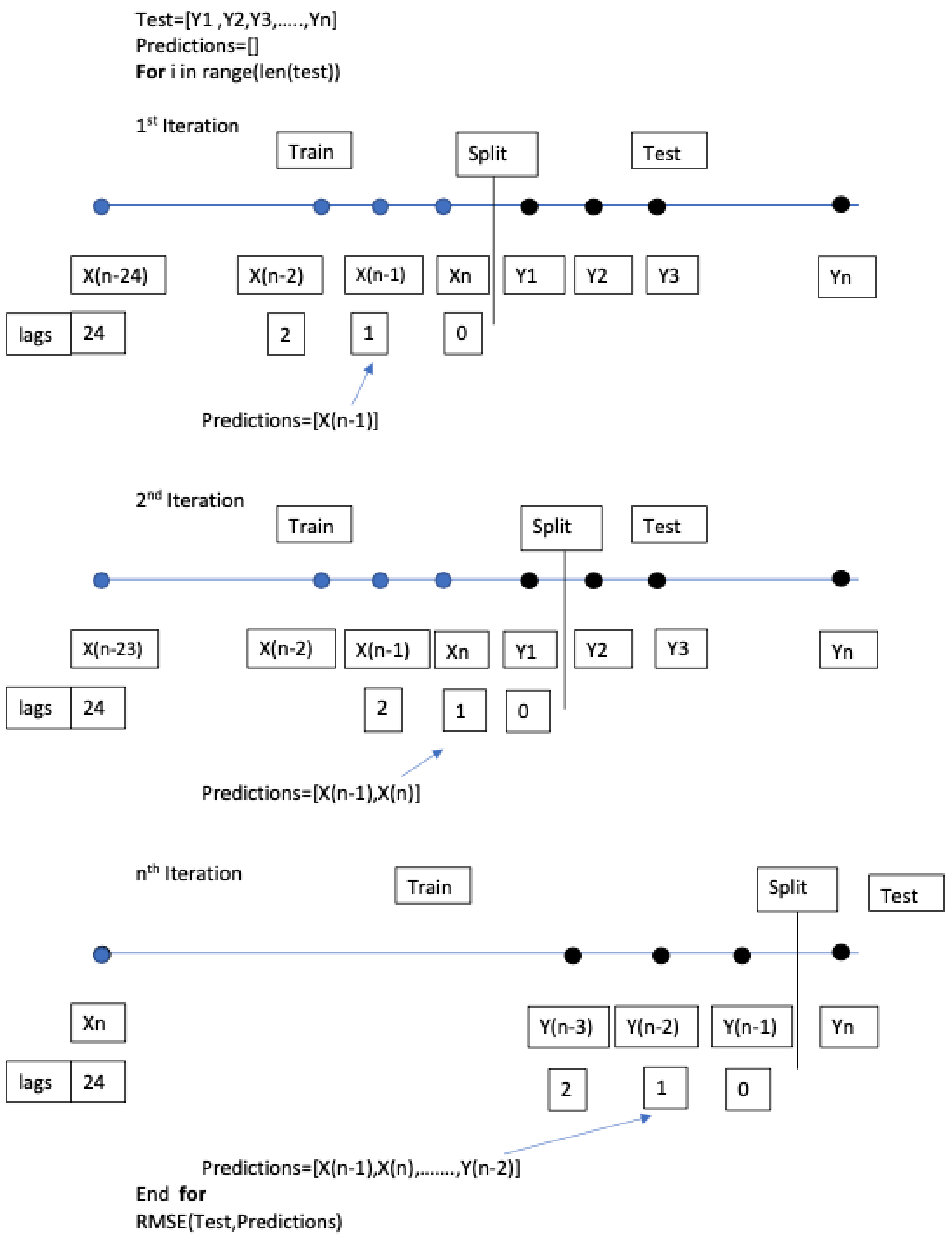
We have to implement a solution in order to decide how many lags must be considered in order to create significant features. These lags in a machine learning algorithms could be tune as hyper-parameters therefore we try to implement a grid search in order to decide the best size window among a set of choice that minimaize the error.

3.1 Grid search

In our grid search we consider a list of possible configuration (lags) that are evaluated in the test set via walk forward cross validation (by adding the first test observation to the train set in each iteration). For each possible configuration the root mean square error is computed.

Since we are dealing with monthly time series the possible lags for our window size are [1,2,3,4,5,6,7,8,9,10,11,12] that take in account lags in a range of one year.

A simple explanation of how the grid search works is proposed below for lag 1:



Below we will create the functions to implement the grid search.

```
[ ]: import math
     from math import sqrt
     from numpy import mean
     from pandas import read_csv
     from sklearn.metrics import mean_squared_error
```

The train_test_split() function below will split the series, taking the raw observations and the number of observations to use in the test set as arguments.

```
[ ]: # split a univariate dataset into train/test sets
     def train_test_split(data, n_test):
         return data[:-n_test], data[-n_test:]
```

A model will be fit once on the training dataset for a given configuration.

```
[ ]: # fit a model
     def model_fit(train, config):
         return None
```

Each time step of the test dataset is enumerated. A prediction is made using the fit model. Again, we will define a generic function named model_predict() that takes the fit model, the history, and the model configuration and makes a single one-step prediction.

```
[ ]: # forecast with a pre-fit model
     def model_predict(model, history, offset):
         return history[-offset]
```

We will calculate the root mean squared error, or RMSE, between predictions and the true values. This function will be used to evaluate different configurations (window-size).

```
[ ]: # root mean squared error or rmse
     def measure_rmse(actual, predicted):
         return sqrt(mean_squared_error(actual, predicted))
```

The prediction is added to a list of predictions and the true observation from the test set is added to a list of observations that was seeded with all observations from the training dataset. This list is built up during each step in the walk-forward validation, allowing the model to make a one-step prediction using the most recent history. A further explanation of walk forward validation is proposed later for the ARIMA/SARIMA model. All of the predictions can then be compared to the true values in the test set and an error measure is calculated. We will calculate the root mean squared error between predictions and the true values.

```
[ ]: # walk-forward validation for univariate data
     #takes the dataset, the number of observations to use as the test set, and the␣
      ↪configuration for the model,
     #and returns the RMSE for the model performance on the test set.
     def walk_forward_validation(data, n_test, cfg):
         predictions = list()
         # split dataset
         train, test = train_test_split(data, n_test)
         # fit model
         model = model_fit(train, cfg)
         # seed history with training dataset
         history = [x for x in train]
         # step over each time-step in the test set
         for i in range(len(test)):
             # fit model and make forecast for history
```

```
        yhat = model_predict(model, history, cfg)
        # store forecast in list of predictions
        predictions.append(yhat)
        # add actual observation to history for the next loop
        history.append(test[i])
    # estimate prediction error
    error = measure_rmse(test, predictions)
    print(' > %.3f' % error)
    return error
```

Repeat evaluation multiple times: Genetic Algorithm such as TADA are adaptive. Given the same model configuration and the same training dataset, a different set of weights will result each time, this mean that we could have different models with different performances. This is a benefit because the model find high performing configurations to complex problem. However it is also a problem when evaluating the performance of a model and choosing a final model to predict. To address model evaluation, we will evaluate a model configuration multiple times via walk-forward validation and report the error as the average error across each evaluation. Indeed this is not always possible especially when we deal with computational intense algorithms where a lot of weights must be set. The repeat_evaluate() function below implements this and allows the number of repetitions to be specified as an optional parameter, by default is set to 1 and returns the mean RMSE score from all repetitions.

```
[ ]: #repeat evaluation
     def repeat_evaluate(data, config, n_test, n_repeats=1):
         # convert config to a key
         key = str(config)
         # fit and evaluate the model n times
         scores = [walk_forward_validation(data, n_test, config) for _ in␣
      ↪range(n_repeats)]
         # summarize score
         result = mean(scores)
         print('> Model[%s] %.3f' % (key, result))
         return (key, result)
```

All that is left is to create a function to drive the search. We can define a grid_search() function that takes the dataset, a list of configurations to search, and the number of observations to use as the test set and perform the search.

```
[ ]: # grid search configs
     def grid_search(data, cfg_list, n_test):
         # evaluate configs
         scores = [repeat_evaluate(data, cfg, n_test) for cfg in cfg_list]
         # sort configs by error, asc
         scores.sort(key=lambda tup: tup[1])
         return scores
```

After having create the grid_search function, we apply it to our data. We decide to split the train and test set considering as a test just the last 12 observations (more recent observations). We take into account the best 6 models (lower RMSE) and select from here the best window size. We are going to create for each observation a number of columns equal to the best lag selected. We should keep in mind in the selection of the best window size that,for example, features created for lag 12 includes all features that are going to be created picking up lag 6.

```
[ ]: data = X
     # data split
     n_test = 12
     # model configs
     cfg_list = [1,2,3,4,5,6,7,8,9,10,11,12]
     # grid search
     scores = grid_search(data, cfg_list, n_test)
     print('done')
     # list top 10 configs
     for cfg, error in scores[:6]:
         print(cfg, error)
```

3.2 tsExtract: Time Series Preprocessing Library

To restructure time series data to look like a supervised learning problem we will use tsExtract library. tsExtract is a time series preprocessing library. Using sliding windows, tsExtract allows for the conversion of time series data to a form that can be fed into standard machine learning as well as Tada Algorithm.

Main Feature:

- Take sliding window of data and with that create additional columns representing the window, we use the grid search previously computed to decide the best window size.
- Calculate statistics on windowed and differenced data. These include temporal and spectral statistics functions.
- Include_tzero (optional): this gives the option on whether to include the column t+0 identified below by X_t. We are going to include these feature since the lags consider on the previous grid search start from 1.
- target_lag - Sets lag value: the Default value is 3. If we put target_lag=12 the target for the value on Date 2018-12-31 is the value on 2019-12-31. In general the target at the time t is Xt+lag_score. We are going to consider target_lag=1, target_lag=2, target_lag=3 identified below by X_{t+i} where i=1,2,3.

Feature: - window: takes sliding window of the data. A single value will take a sliding window corresponding to that value. A parameter of [10] will take window from 1 to 10. if [5,10] passed then a window of 5 to 10 steps will be taken instead.

- window_statistic: this perform windowing like above, but then applies specified statistic operation to reduce a matrix to a vector of 1d.
- difference/momentum/force: performs differencing by subtracting from the value in the present time step, the value in the previous time step. The parameter expected is a list of size 2 or 3. Like in the windowing the first 2 values refer to the window size. As in

the window just one parameter could be taken. The final value is the lag, this refer to the differencing lag for subtraction.

Supported features for summary statistic:

- Mean
- Absolute energy
- Median
- Range
- Mean Absolute difference
- Standard Deviation
- Moment
- Minimum
- Maximum

Another advantage of this library is that it deletes automatically the observations when sliding windows are performed, without creating problem with Nan values on TADA.

Starting from the window size k selected with the grid search the following features are created for the selected target X_{t+i} where i =(1,2,3):

- $X_t, X_{t-1}, X_{t-2}, ..., X_{t-k}$
- $(X_t - X_{t-12})$
- $std(X_{t-1}, X_{t-2}, ..., X_{t-12})$
- $(X_t - X_{t-1})^2 + (X_{t-1} + X_{t-2})^2$
- $median(X_{t-1}, X_{t-2}, ..., X_{t-12})$

[ ]:
```
import tsextract
from tsextract.feature_extraction.extract import build_features
from tsextract.domain.statistics import median, mean, skew, kurtosis, std
from tsextract.domain.temporal import abs_energy

features_request = {
    "window":[k], # Xt-1, Xt-2, Xt-3, Xt-4,Xt-5,Xt-6,....,Xt-k  features
    "window_statistic":[12, std], #standard deviation of(Xt-1,Xt-2,Xt-3,....
 ↪,Xt-12)
    "difference":[13,12], #(Xt-Xt-12) feature
    "difference_statistic":[3,1, abs_energy], #(Xt-Xt-1)^2+(Xt-1-Xt-2)^2␣
 ↪feature
     "window_statistic":[12,median],#median of (Xt-1,Xt-2,Xt-3,....,Xt-12)␣
 ↪feature
}
```

```
features = build_features(series, features_request, include_tzero=True,␣
↪target_lag=i)
# our goal is to predict i step forward target_lag=(1,2,3)
```

Save the features created in a csv file and use them on TADA.

TADA algorithm perform an automatic train and test split to evaluate the model and a features selection. Only the most important features for the target are going to be included in the model. Despite this TADA maybe get confused with a window size too big that reduce the number of observations and increase the probability to overfit. A way to control this is taking a window size not to big, by taking in account lags within a year in the grid search, and check if the feature X_t (tzero) is included in the model.

The deploy of a TADA model is showed below

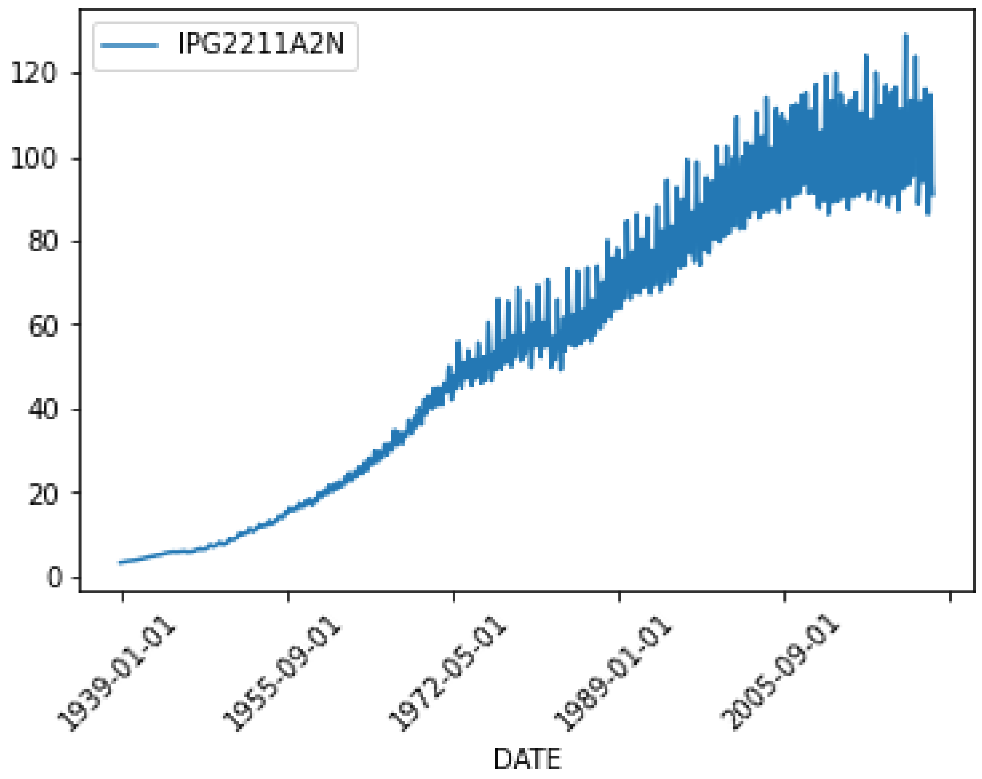
```
[ ]: abs = math.fabs
ceiling = math.ceil
cos = math.cos
exp = math.exp
floor = math.floor
log = math.log
sin = math.sin
sqrt = math.sqrt
truncate = math.trunc

variables = ["T-4", "difference_13_12-1", "T-6", "T-9", "tzero",␣
↪"window_statistic_12_median"]


def check_sample(sample: pd.Series):
    for variable in variables:
        if variable not in sample.index:
            raise Exception("Variable error: {} not present in input".
↪format(variable))


def predict(sample: pd.Series, preprocessed: bool) -> float:
    check_sample(sample)

    column_14 = float(sample["T-4"])
    column_15 = float(sample["difference_13_12-1"])
    column_4 = float(sample["T-6"])
    column_7 = float(sample["T-9"])
    tzero = float(sample["tzero"])
    window_statistic_12_median = float(sample["window_statistic_12_median"])
    return 0.000283602467268655*column_14 + 0.340961672402632*column_15 + 1.
↪07043829206052*column_7 + 0.00343208879541341*column_14/(0.
↪262807174911542*sin(tzero)*window_statistic_12_median + 0.
↪154145598415688*truncate(tzero)*window_statistic_12_median + 0.
↪58304722667277*window_statistic_12_median**2) - 0.
↪00343208879541341*column_15/(0.
↪262807174911542*sin(tzero)*window_statistic_12_median + 0.
↪154145598415688*truncate(tzero)*window_statistic_12_median + 0.
↪58304722667277*window_statistic_12_median**2) - 0.
↪0171012268159996*column_15**2/(0.
↪262807174911542*sin(tzero)*window_statistic_12_median + 0.
↪154145598415688*truncate(tzero)*window_statistic_12_median + 0.
↪58304722667277*window_statistic_12_median**2) - 0.
↪000842712957285145*floor(column_7)/(0.
↪139132745139051*cos(column_4)*abs(column_4) + 0.
↪160096225100308*cos(column_4)**2 + 0.0302285715322137*abs(column_4)**2 + 0.
↪670542458228428*sin(0.697093156328679*cos(column_4) + 0.
↪302906843671321*abs(column_4))) - 0.00452568613129497*sin(tzero)*column_7 +␣
↪0.00121614068378524*sin(tzero)*window_statistic_12_median - 0.
↪000359464212885213*cos(column_4)*abs(column_4) - 0.
↪00265447317861414*truncate(tzero)*column_7 + 0.
↪000713309039309294*truncate(tzero)*window_statistic_12_median + 0.
↪00118416406984618*truncate((0.630304417486839*sin(tzero) + 0.
↪369695582513161*truncate(tzero))/column_7)*column_7 - 0.
↪0013835616287463*column_4*column_7/(0.
↪139132745139051*cos(column_4)*abs(column_4) + 0.
↪160096225100308*cos(column_4)**2 + 0.0302285715322137*abs(column_4)**2 + 0.
↪670542458228428*sin(0.697093156328679*cos(column_4) + 0.
↪302906843671321*abs(column_4))) - 0.00141311906755442*column_15**2 + 0.
↪00269805211050108*window_statistic_12_median**2 - 0.
↪000413625588168054*cos(column_4)**2 - 7.80987226379516e-05*abs(column_4)**2␣
↪- 0.00215231264110972*floor(0.621469443808652*column_4*column_7 + 0.
↪378530556191348*floor(column_7)) - 0.000746383844253243*sin(tzero) - 0.
↪00173241760386675*sin(0.697093156328679*cos(column_4) + 0.
↪302906843671321*abs(column_4)) - 0.00043778022559294*truncate(tzero)

# Association table between variable and column names:
# "column_14" = "T-4"
# "column_15" = "difference_13_12-1"
# "column_4" = "T-6"
# "column_7" = "T-9"
```

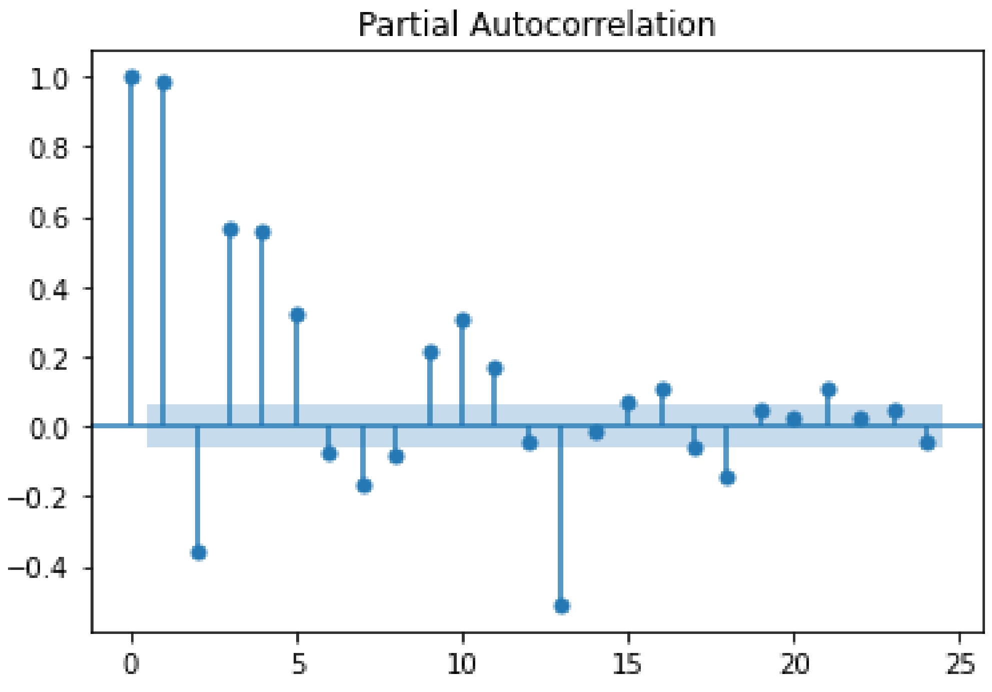
4 Experiment

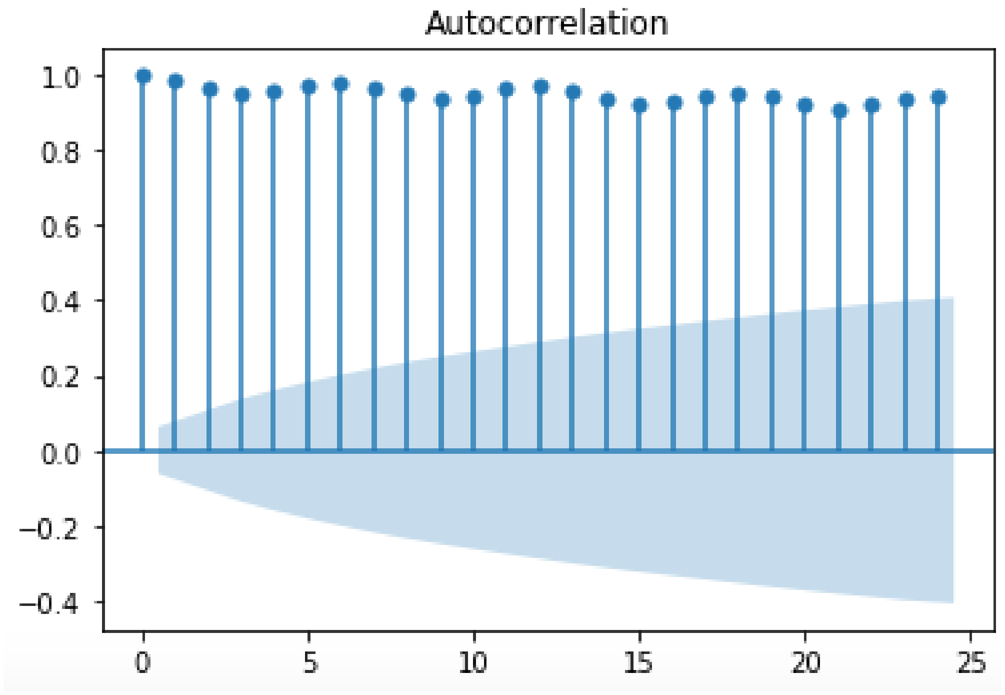
The aim of this section is try to verify the procedure proposed above on three monthly univariate time series. Then compare the result with an Arima model in which parameter are tuned based on a grid search. Before proceeding with the experiment a descriptive analysis for every dataset is proposed.

4.1 Monthly Eletric Gas Utilities Sales

The dataset contains 982 observations from 1939-01 to 2020-10. The series present non stationarity in mean and stationarity in variance in the first part, then both non stationarity in mean and in variance in the second part.

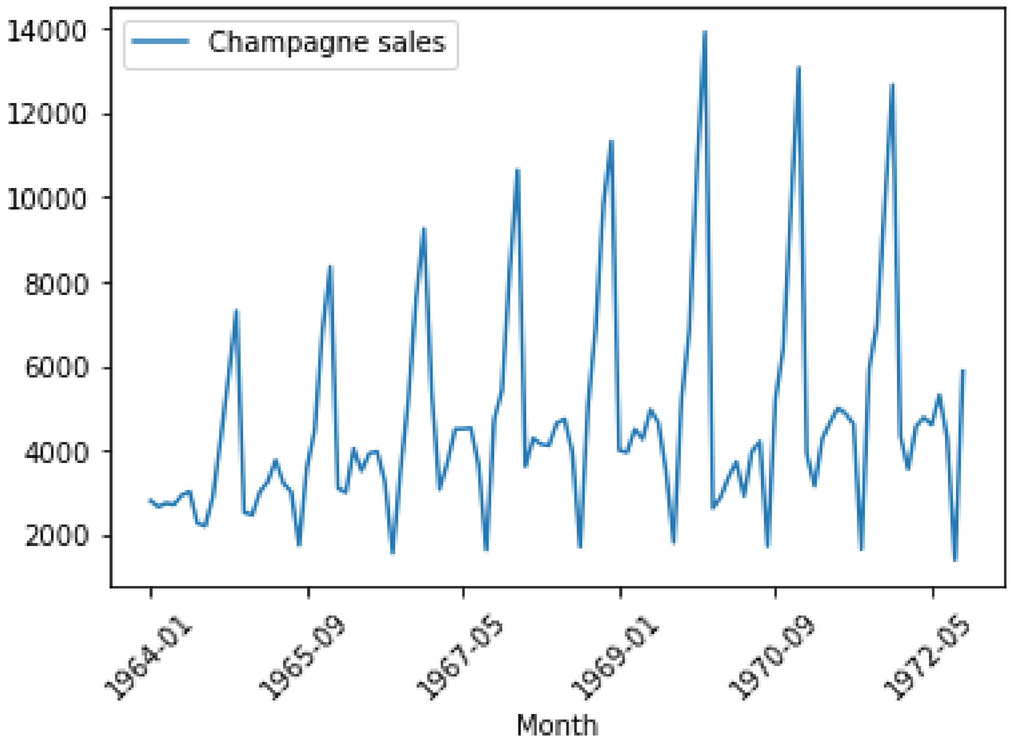


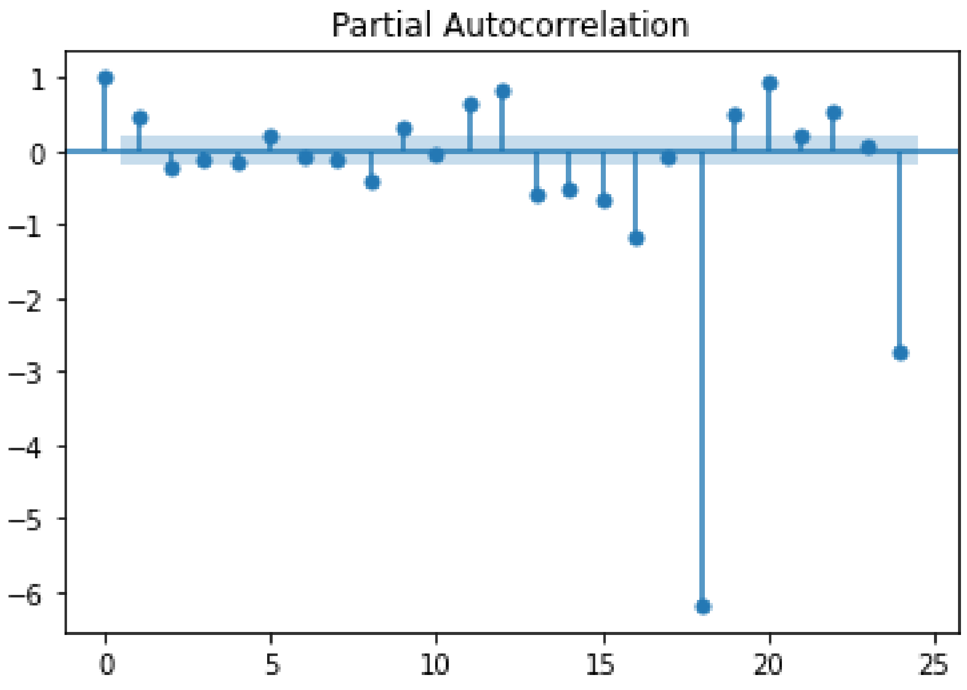


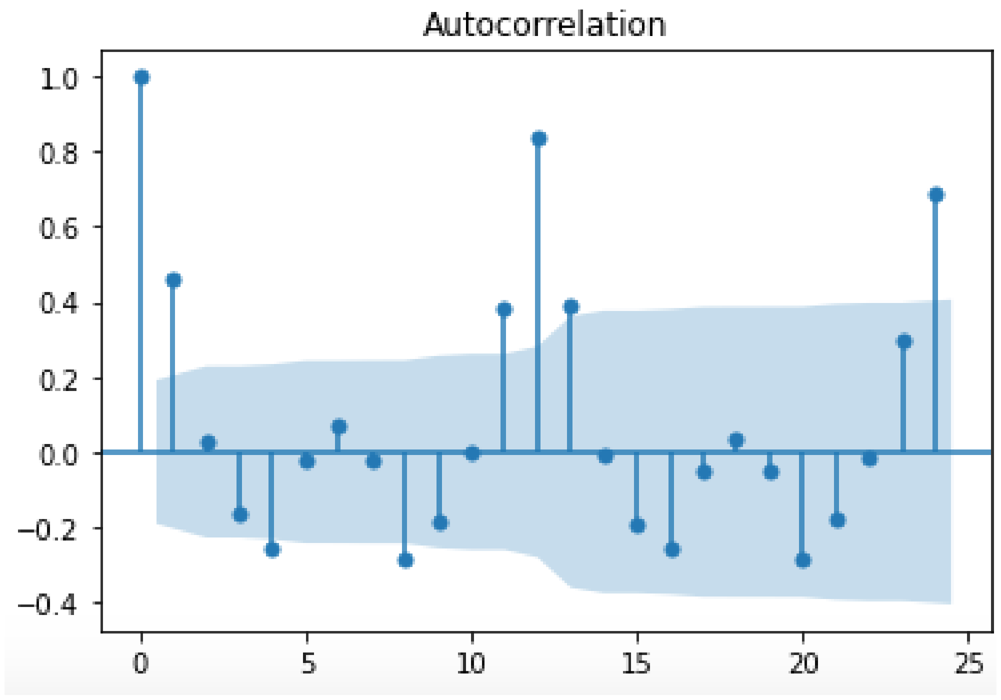


4.2 Monthly Champagne Sales

The dataset contains 105 observations from 1964-01 to 1972-09. The series shows seasonality and a non stationarity in mean.

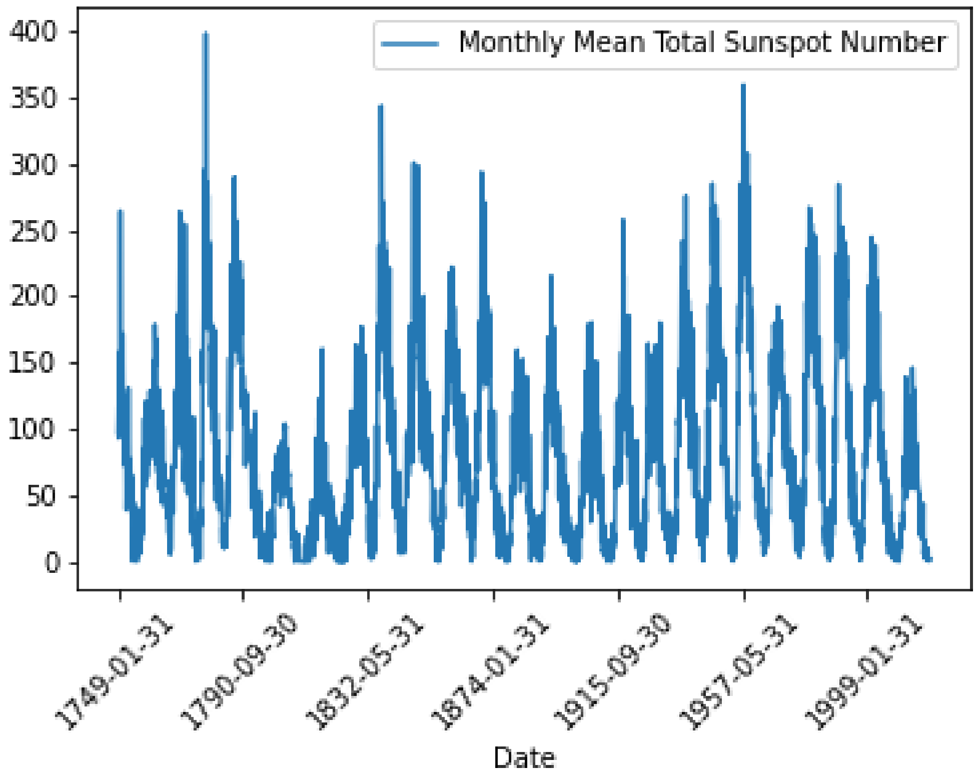

Champagne sales
14000
12000
10000
8000
6000
4000
2000
1964-01
1965-09
1967-05
1969-01
1970-09
1972-05
Month



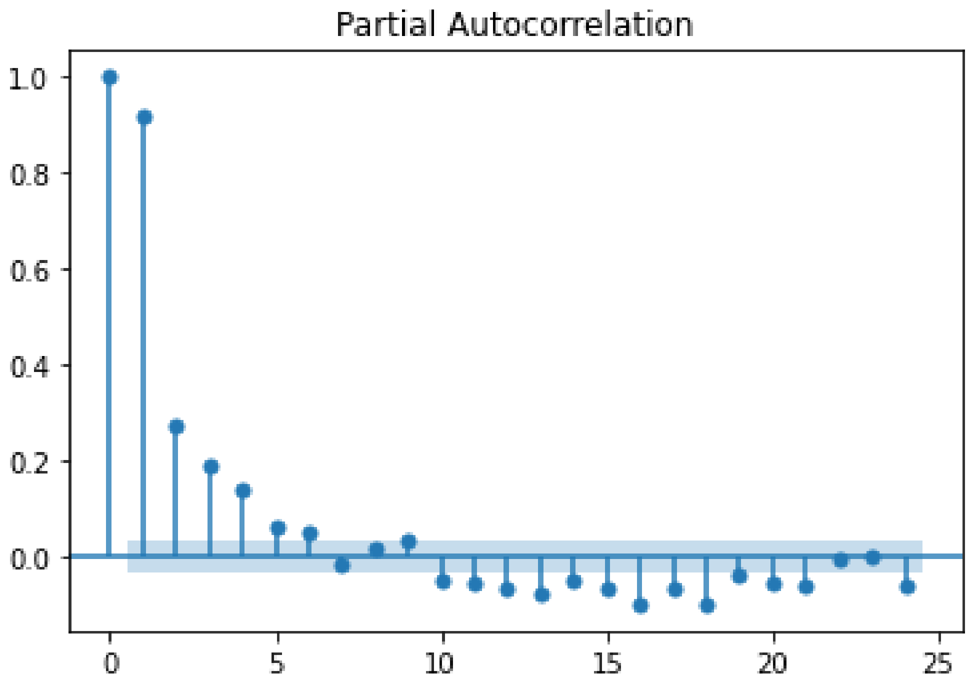


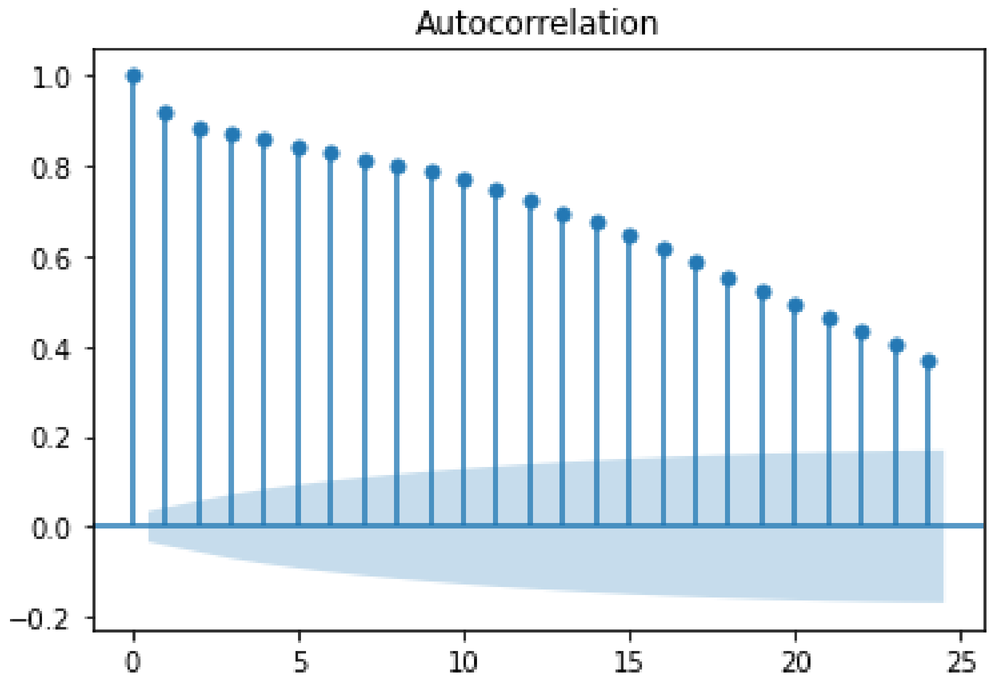
4.3 Monthly Mean Sunspot Number

The dataset contains 3252 observations from 1749-01 to 2019-12. The series shows seasonality with large differences between seasons and stationarity in mean.



Monthly Mean Total Sunspot Number
400
350
300
250
200
150
100
50
0
1749-01-31
1790-09-30
1832-05-31
1874-01-31
1915-09-30
1957-05-31
1999-01-31
Date





4.4 Grid search ARIMA/SARIMA model

To compare a traditional model like Arima with TADA model we defined another grid search in order to tune the parameters of the model and make the comparison. The grid search is implemented for a SARIMA model that includes also parameters to capture the seasonal trend. However this require high computational resources, the number of possible configurations increase a lot, therefore we set equal to zero the parameters for the seasonal trend and create an Arima model. Indeed the procedure can be improve by consider also SARIMA model in the grid search. As before we state the function used.

```
[ ]: from warnings import catch_warnings
     from warnings import filterwarnings
     from statsmodels.tsa.statespace.sarimax import SARIMAX
```

The function takes an array of observations and a list of configuration parameters used to configure the model: specifically two tuples and a string for the trend order, seasonal order trend and parameters.

```
[ ]: # one-step sarima forecast
     def sarima_forecast(history, config):
         order, sorder, trend = config
         # define model
         model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,␣
      ↪enforce_stationarity=False, enforce_invertibility=False)
         # fit model
         model_fit = model.fit(disp=False)
         # make one step forecast
         yhat = model_fit.predict(len(history), len(history))
         return yhat[0]
```

we need to build up some functions for fitting and evaluating a model repeatedly via walk-forward validation,including splitting a dataset into train and test sets and evaluating one-step forecasts. we have already created a train_test_split and measure_rmse functions for the window size grid search, we will reuse them here.

We can now implement the walk-forward validation scheme. This is a standard approach to evaluating a time series forecasting. The function sarima_forecast() has the opportunity to make good forecasts at each time step, as new data become available.

1. A provided univariate time series dataset is split into train and test sets using the train_test_split() function.
2. The number of observations in the test set are enumerated. For each we fit a model on all of the history and make a one step forecast.
3. The true observation for the time step is then added to the history and the process is repeated.

The sarima_forecast() function is called in order to fit a model and make a prediction.

4. An error score is calculated by comparing all one-step forecasts to the actual test set by calling the measure_rmse() function.

```
[ ]: # walk-forward validation for univariate data
     def walk_forward_validation(data, n_test, cfg):
         predictions = list()
         # split dataset
         train, test = train_test_split(data, n_test)
         # seed history with training dataset
         history = [x for x in train]
```

```
    # step over each time-step in the test set
    for i in range(len(test)):
        # fit model and make forecast for history
        yhat = sarima_forecast(history, cfg)
        # store forecast in list of predictions
        predictions.append(yhat)
        # add actual observation to history for the next loop
        history.append(test[i])
    # estimate prediction error
    error = measure_rmse(test, predictions)
    return error
```

Possible problems: - Some combinations of model configurations may not be called for the model and will throw an exception.

- Some models may also raise warnings on some data.

Solutions: - Ignore warnings during the grid search.

- Return some information about the skill of each model evaluated. This is helpful when a large number of models are evaluated.

The score_model() function below implements this and returns a tuple of (key and result), where the key is a string version of the tested model configuration. Score a model, return None on failure.

[ ]:
```
def score_model(data, n_test, cfg, debug=False):
    result = None
    # convert config to a key
    key = str(cfg)
    # show all warnings and fail on exception if debugging
    if debug:
        result = walk_forward_validation(data, n_test, cfg)
    else:
        # one failure during model validation suggests an unstable config
        try:
            # never show warnings when grid searching, too noisy
            with catch_warnings():
                filterwarnings("ignore")
                result = walk_forward_validation(data, n_test, cfg)
        except:
            error = None
    # check for an interesting result
    if result is not None:
        print(' > Model[%s] %.3f' % (key, result))
    return (key, result)
```

Now we need a loop to test a list of different model configurations. This is the main function that drives the grid search process, we will call the score_model() function for each model configuration. The grid_search() function below implements this behavior given:

- a univariate time series dataset.
- a list of model configurations (list of lists).
- the number of time steps to use in the test set.

```
[ ]: # grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
    scores = None
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
    # remove empty results
    scores = [r for r in scores if r[1] != None]
    # sort configs by error, asc
    scores.sort(key=lambda tup: tup[1])
    return scores
```

The only thing left to do is to define a list of model configurations to try for a dataset. We can define this generically. The only parameter we may want to specify is the periodicity of the seasonal component in the series, if one exists. By default, we will assume no seasonal component but of course it depends on the time series that we are analyzing.

```
[ ]: # create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
    models = list()
    # define config lists
    #autoregressive parameter for modeling the trend
    p_params = [0, 1, 2]
    #integrated parameter for modeling the trend
    d_params = [0, 1]
    #moving average for modeling the trend
    q_params = [0, 1, 2]
    #parameters for deterministic trend
    #no trend, constant trend, linear trend
    t_params = ['n','c','t']
    #autoregressive parameter for modeling the seasonality
    P_params = [0, 1]
    #integrated parameter for modeling the seasonality
    D_params = [0, 1]
    #moving average for modeling the seasonality
    Q_params = [0, 1]
    #seasonal component defined by the user
    m_params = seasonal
    # create config instances
    for p in p_params:
        for d in d_params:
            for q in q_params:
                for t in t_params:
                    for P in P_params:
                        for D in D_params:
```

```
                                    for Q in Q_params:
                                        for m in m_params:
                                            cfg = [(p,d,q), (P,D,Q,m), t]
                                            models.append(cfg)
    return models
```

For time computing reason, as explained before, we set seasonal trend parameters equal to 0, so at the end we will have a ARIMA model to compare our TADA result.

After having create the grid_search function, we apply it to our data. We decide to split the train and test set considering as a test just the last 12 observations (more recent observations). To select the best ARIMA model the best three models with lowest RMSE are printed.

[ ]:
```
# define dataset
data = X

# data split
n_test = 12
# model configs
#manually state the parameter for the seasonality that we would like to try.
cfg_list = sarima_configs(seasonal=[0])
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 3 configs
for cfg, error in scores[:3]:
    print(cfg, error)
```

4.5 Comparison TADA VS ARIMA

The comparison between best Arima(, ,) model previously selected and Tada algorithm is showed below. Here some guidelines:

- Since we selected the last 12 (more recent) observations in the train_test_split in both grid search we are going to consider has test set these observations.
- To compare a Time series model with a machine learning algorithms such as TADA we used two different approach:
  - Walk forward validation in Arima model. After every prediction the true value is added at the train set and a new prediction is computed.Then the RMSE between predictions and true value is computed.
  - On TADA for every observations in the last 12 rows (more recent ones) we applied the coefficients given by the algorithm at the features and compute the error with respect to the target stored in the same row. Then the RMSE between predictions and true value is computed
- The step forward considered are 1,2,3
- A summary plot is computed.

```
[ ]: predictions_ARIMA = list()
     # define model configuration
     my_order = (, , )
     #seasonal parameters set to zero for computing resources
     my_seasonal_order = (0, 0, 0, 0)
     # split dataset
     train, test = train_test_split(X, 12)
     history = [x for x in train]

     # walk-forward validation
     for t in range(len(test)):
         model = SARIMAX(history,order=my_order, seasonal_order=my_seasonal_order)
         model_fit = model.fit()
         output = model_fit.forecast()
         yhat = output[0]
         predictions_ARIMA.append(yhat)
         obs = test[t]
         history.append(obs)
         print('predicted ARIMA=%f, expected=%f' % (yhat, obs))
     # evaluate forecasts ARIMA
     rmse_ARIMA = sqrt(mean_squared_error(test, predictions_ARIMA))
     print('Test RMSE ARIMA: %.3f' % rmse_ARIMA)


     predictions_TADA=list()

     #from features dataframe, we select only the last 12 rows
     ft=features.iloc[-12:]
     for i in range(len(ft)):
         i=int(i)
         current=ft.iloc[i, ]
         target=float(current["Target_Tplus3"])
         column_14 = float(current["T-4"])
         column_15 = float(current["difference_13_12-1"])
         column_4 = float(current["T-6"])
         column_7 = float(current["T-9"])
         tzero = float(current["tzero"])
         window_statistic_12_median = float(current["window_statistic_12_median"])
```

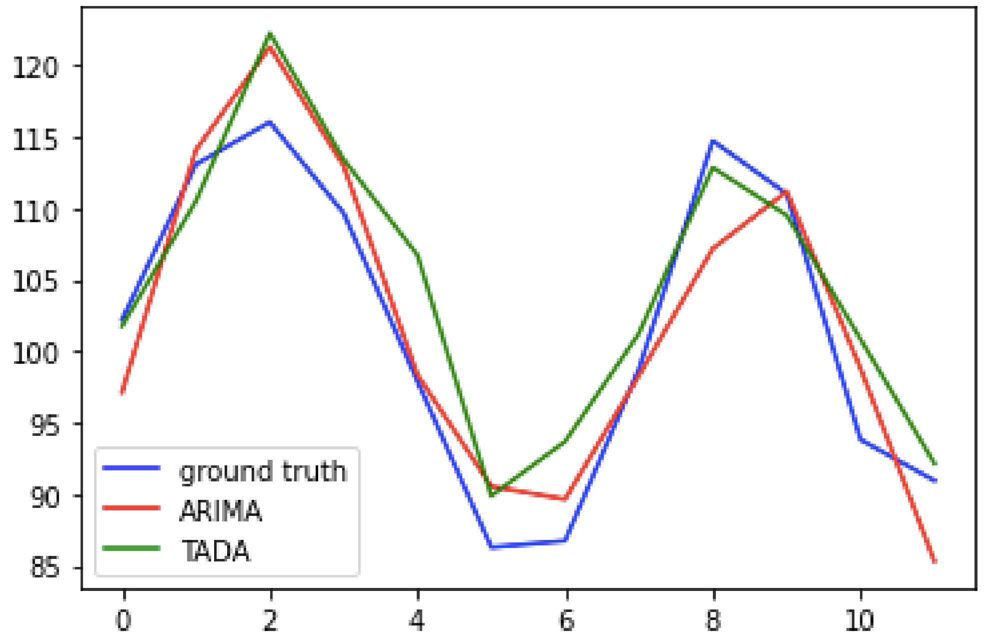
```
    yhat_TADA=0.000283602467268655*column_14 + 0.340961672402632*column_15 + 1.
↪07043829206052*column_7 + 0.00343208879541341*column_14/(0.
↪262807174911542*sin(tzero)*window_statistic_12_median + 0.
↪154145598415688*truncate(tzero)*window_statistic_12_median + 0.
↪58304722667277*window_statistic_12_median**2) - 0.
↪00343208879541341*column_15/(0.
↪262807174911542*sin(tzero)*window_statistic_12_median + 0.
↪154145598415688*truncate(tzero)*window_statistic_12_median + 0.
↪58304722667277*window_statistic_12_median**2) - 0.
↪0171012268159996*column_15**2/(0.
↪262807174911542*sin(tzero)*window_statistic_12_median + 0.
↪154145598415688*truncate(tzero)*window_statistic_12_median + 0.
↪58304722667277*window_statistic_12_median**2) - 0.
↪000842712957285145*floor(column_7)/(0.
↪139132745139051*cos(column_4)*abs(column_4) + 0.
↪160096225100308*cos(column_4)**2 + 0.0302285715322137*abs(column_4)**2 + 0.
↪670542458228428*sin(0.697093156328679*cos(column_4) + 0.
↪302906843671321*abs(column_4))) - 0.00452568613129497*sin(tzero)*column_7 +␣
↪0.00121614068378524*sin(tzero)*window_statistic_12_median - 0.
↪000359464212885213*cos(column_4)*abs(column_4) - 0.
↪00265447317861414*truncate(tzero)*column_7 + 0.
↪00071330903930929*truncate(tzero)*window_statistic_12_median + 0.
↪00118416406984618*truncate((0.630304417486839*sin(tzero) + 0.
↪369695582513161*truncate(tzero))/column_7)*column_7 - 0.
↪0013835616287463*column_4*column_7/(0.
↪139132745139051*cos(column_4)*abs(column_4) + 0.
↪160096225100308*cos(column_4)**2 + 0.0302285715322137*abs(column_4)**2 + 0.
↪670542458228428*sin(0.697093156328679*cos(column_4) + 0.
↪302906843671321*abs(column_4))) - 0.00141311906755442*column_15**2 + 0.
↪00269805211050108*window_statistic_12_median**2 - 0.
↪000413625588168054*cos(column_4)**2 - 7.80987226379516e-05*abs(column_4)**2␣
↪- 0.00215231264110972*floor(0.621469443808652*column_4*column_7 + 0.
↪378530556191348*floor(column_7)) - 0.000746383844253243*sin(tzero) - 0.
↪00173241760386675*sin(0.697093156328679*cos(column_4) + 0.
↪302906843671321*abs(column_4)) - 0.00043778022559294*truncate(tzero)
    predictions_TADA.append(yhat_TADA)
    print('predicted TADA=%f, expected=%f' % (yhat_TADA, target))
#evaluate forecast TADA
rmse_TADA = sqrt(mean_squared_error(test, predictions_TADA))
print('Test RMSE TADA: %.3f' % rmse_TADA)
# plot forecasts against actual outcomes
plt.plot(test,color='blue', label="ground truth")
plt.plot(predictions_ARIMA, color='red', label="ARIMA")
plt.plot(predictions_TADA, color='green',label="TADA")
plt.legend()
plt.show()
```

5 Results

5.1 Monthly Eletric Gas Utilities Sales result

With respect to the test observations selected (12 more recents ones) and a forecast 3 steps (3 months) in the future we obtain:

- RMSE ARIMA: 4.143
- RMSE TADA: 4.671

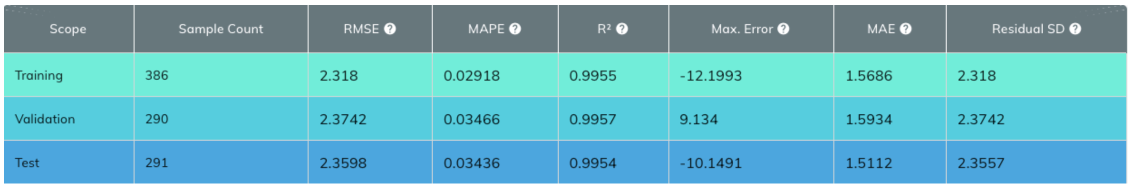


TADA to predict X_{t+3} used only:

- X_t
- X_{t-12}
- X_{t-6}
- X_{t-9}
- $median(X_{t-1}, X_{t-2}, ..., X_{t-12})$

after having selected a window size of 12 from the grid search.

The main metric used (root mean squared error) is really close in the two approaches proposed. The TADA model performance in the train, validation and test set, shows a similar value of the RMSE with respect to the RMSE computed only for the 12 observations (more recent ones) selected for the comparison. Despite a R^2 really close to 1 in the three sets, we think that TADA do not overfit the problem, also the performance of the ARIMA model is good. Further experiment must be done, by increase the test observations selected and split the series in two subset (the number of observation is 982) since it has two different behaviour identified in the time.

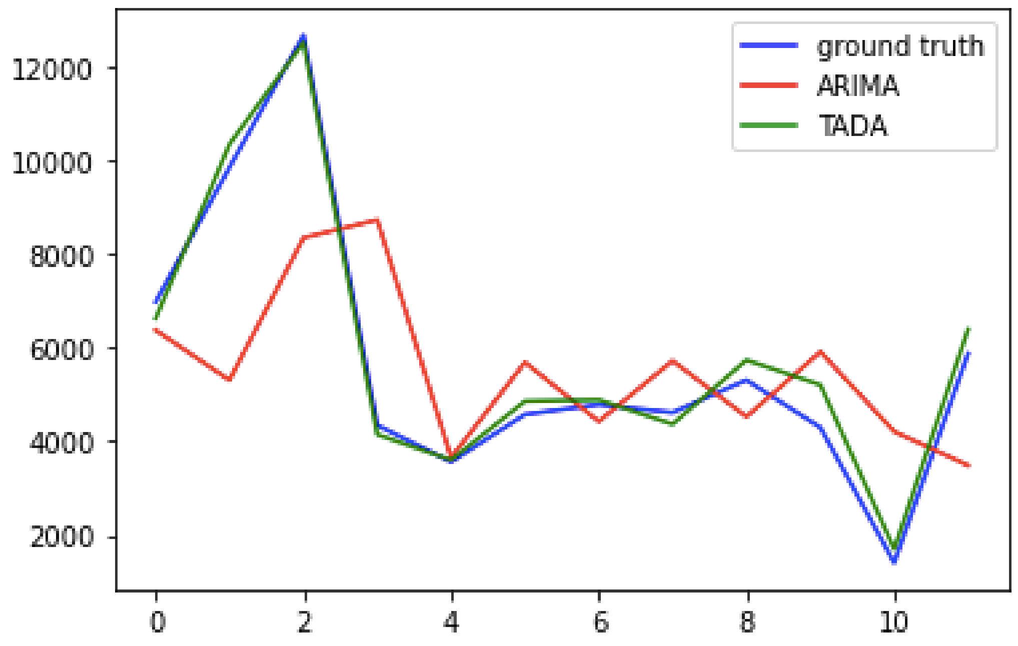


Scope	Sample Count	RMSE ❓	MAPE ❓	R² ❓	Max. Error ❓	MAE ❓	Residual SD ❓
Training	386	2.318	0.02918	0.9955	-12.1993	1.5686	2.318
Validation	290	2.3742	0.03466	0.9957	9.134	1.5934	2.3742
Test	291	2.3598	0.03436	0.9954	-10.1491	1.5112	2.3557

5.2 Monthly Champagne Sales result

With respect to the test observations selected (12 more recents ones) and a forecast 2 steps (2 months) in the future we obtain:

- RMSE ARIMA: 2553.124
- RMSE TADA: 403.167

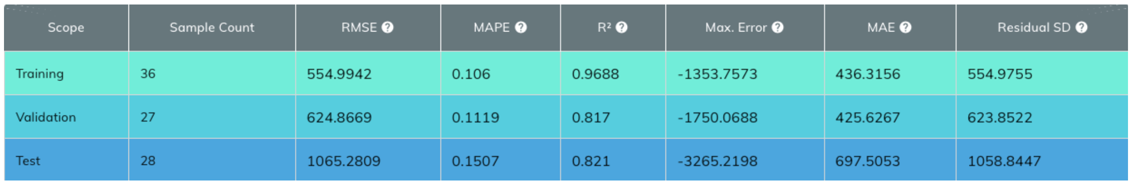


TADA to predict X_{t+2} used only:

- X_t
- X_{t-1}
- X_{t-3}
- X_{t-8}
- X_{t-10}

after having selected a window size of 12 from the grid search.

The main metric used (root mean squared error) is lower in the TADA approach than in the ARIMA model. However the TADA model performance in the train, validation and test set, shows greater value of the RMSE with respect to the RMSE compute only for the 12 observations (more recent ones) selected for the comparison. The Champagne series has only 105 observations, then by the creation of the features only 91 observations (rows) are preserved. It is possible that the algorithm get confused by a low number of input data and overfit the problem.

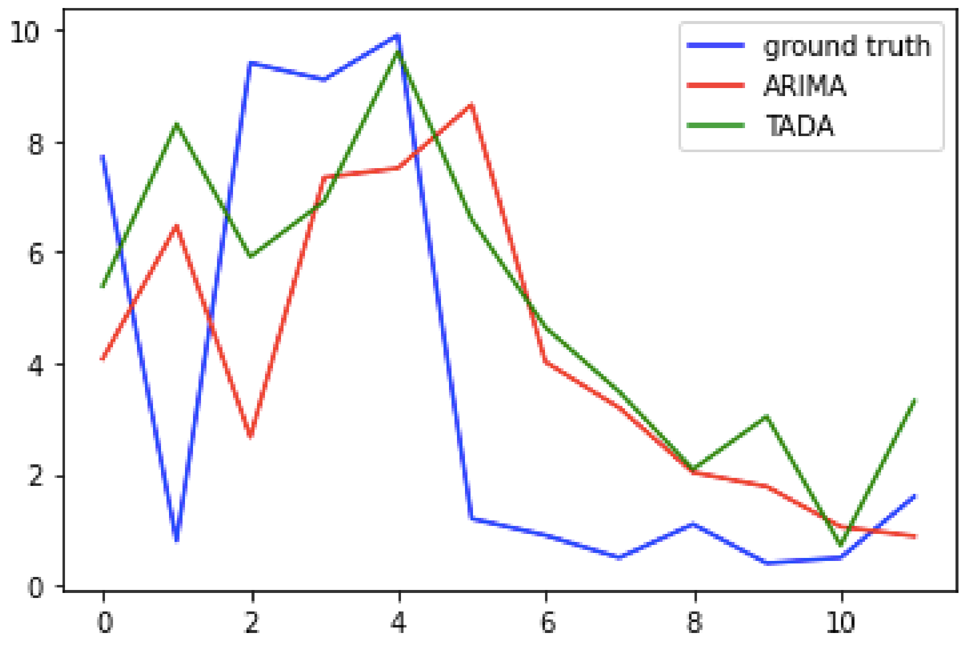


Scope	Sample Count	RMSE	MAPE	R²	Max. Error	MAE	Residual SD
Training	36	554.9942	0.106	0.9688	-1353.7573	436.3156	554.9755
Validation	27	624.8669	0.1119	0.817	-1750.0688	425.6267	623.8522
Test	28	1065.2809	0.1507	0.821	-3265.2198	697.5053	1058.8447

5.3 Monthly Mean Sunspot Number result

With respect to the test observations selected (12 more recents ones) and a forecast 1 steps (1 months) in the future we obtain:

- RMSE ARIMA: 3.825
- RMSE TADA: 3.435

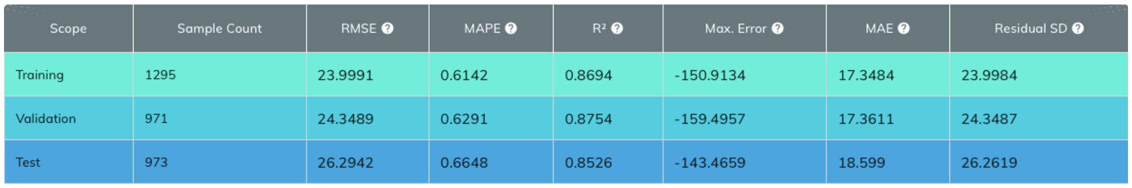


TADA to predict X_{t+1} used only:

- X_t
- X_{t-1}
- X_{t-3}
- X_{t-8}
- $(X_t - X_{t-12})$

after having selected a window size of 11 from the grid search.

The main metric used (root mean squared error) is lower in the TADA approach than in the ARIMA model. However the TADA model performance in the train, validation and test set, shows greater value of the RMSE with respect to the RMSE compute only for the 12 observations (more recent ones) selected for the comparison. Again we obtain a discrepancy between the metrics. In this case the Monthly Mean Sunspot Number series has 3252 observations then by creation of the features 3239 observations are preserved, while the comparison with the ARIMA model is take in account only 12 observations. This differences in the number of observations can cause variance in the metrics computed.



Scope	Sample Count	RMSE	MAPE	R²	Max. Error	MAE	Residual SD
Training	1295	23.9991	0.6142	0.8694	-150.9134	17.3484	23.9984
Validation	971	24.3489	0.6291	0.8754	-159.4957	17.3611	24.3487
Test	973	26.2942	0.6648	0.8526	-143.4659	18.599	26.2619

5.4 Conclusion and possible future works:

- Tada on monthly univariate time series works for short steps forecast (1,2,3 months) in the future. The performances in the series are in the same order of magnitude of the Arima model except for the Champagne sales series where probably the algorithm get confused by lack of information.
- The performance of the Algorithm varies with respect to the amount of data available. When the number of observations exceed a certain value a creation of a subset is needed. Instead when the number of observation is low the algorithm tend to overfit without generalize well.
- For the comparison with the Arima model we only consider the 12 most recent observations without taking in account the number of observations available. Perhaps this bunch of data is too small, known train_test split could be used (0.5, 0.33, 0.1) to evaluate the models.
- Since we focus on short term forecast we used ARIMA, model for long term forecast like LSTM could be used to evaluate the performance of TADA on long term predictions.

The code is available at the following link: https://github.com/NicolaRonzoni/MyDataModels-.git