# Time Series Forecasting: ARIMA vs LSTM vs PROPHET

## Time Series Forecasting with Machine Learning and Python

Mauro Di Pietro Follow
Mar 9 · 12 min read ★

### Summary

The purpose of this article is to find the best algorithm for forecasting, the competitors are ARIMA processes, LSTM neural network, Facebook Prophet model. I will walk through every line of code with comments, so that you can easily replicate this example (link to the full code below).

We will use a dataset from the Kaggle competition "**Predict Future Sales**" (linked below) in which you are provided with daily historical sales data and the task is to forecast the total amount of products sold. The dataset presents an interesting time series as it is very similar to use cases that can be found in real world, as we know daily sales of any product are never stationary and are always heavily affected by seasonality.

**Full Code** (Github):

mdipietro09/DataScience_ArtificialIntelligence_Utils

Permalink Dismiss GitHub is home to over 50 million developers working together to host and review code, manage…

github.com

**Dataset** (Kaggle):

Predict Future Sales

Final project for "How to win a data science competition" Coursera course

www.kaggle.com

## Setup

First of all, we will import the following libraries

```
## For data
import pandas as pd
import numpy as np

## For plotting
import matplotlib.pyplot as plt

## For Arima
import pmdarima
import statsmodels.tsa.api as smt

## For Lstm
from tensorflow.keras import models, layers, preprocessing as
kprocessing

## For Prophet
from fbprophet import Prophet
```

Then we will read the data into a pandas Dataframe

```
dtf = pd.read_csv('data.csv')

dtf.head()
```

| | date | date_block_num | shop_id | item_id | item_price | item_cnt_day |
|---|---|---|---|---|---|---|
| 0 | 02.01.2013 | 0 | 59 | 22154 | 999.00 | 1.0 |
| 1 | 03.01.2013 | 0 | 25 | 2552 | 899.00 | 1.0 |
| 2 | 05.01.2013 | 0 | 25 | 2552 | 899.00 | -1.0 |
| 3 | 06.01.2013 | 0 | 25 | 2554 | 1709.05 | 1.0 |
| 4 | 15.01.2013 | 0 | 25 | 2555 | 1099.00 | 1.0 |

The original dataset has different columns, however for the purpose of this tutorial we only need the following column: date and the number of products sold (item_cnt_day). In other words, we'll be creating a pandas Series (named "sales") with a daily frequency datetime index using only the daily amount of sales

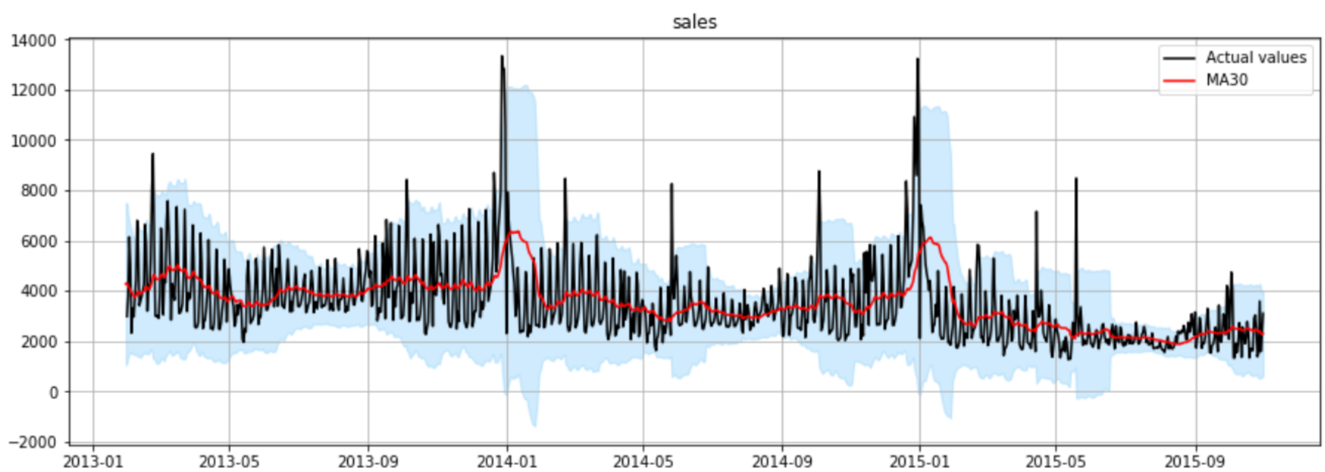
```
## format datetime column
dtf["date"] = pd.to_datetime(dtf['date'], format='%d.%m.%Y')

## create time series
ts = dtf.groupby("date")["item_cnt_day"].sum().rename("sales")

ts.head()
```

```
date
2013-01-01    1951.0
2013-01-02    8198.0
2013-01-03    7422.0
2013-01-04    6617.0
2013-01-05    6346.0
Name: sales, dtype: float64
```

```
ts.tail()
```

```
date
2015-10-27    1551.0
2015-10-28    3593.0
2015-10-29    1589.0
2015-10-30    2274.0
2015-10-31    3104.0
Name: sales, dtype: float64
```

So the time series ranges from **2013–01–01** until **2015–10–31**, it has **1034 observations**, a **mean of 3528** and a **standard deviation of 1585**. It looks like this:




code to plot this is in the first tutorial, link on top

Let's get started now, shall we?

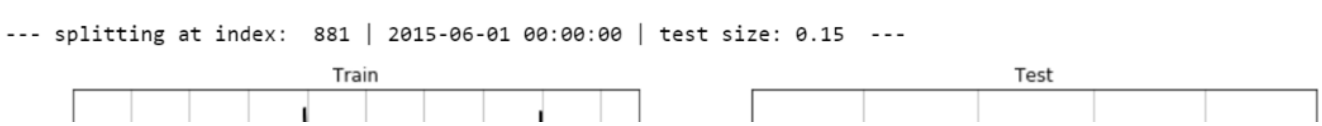
## Partitioning

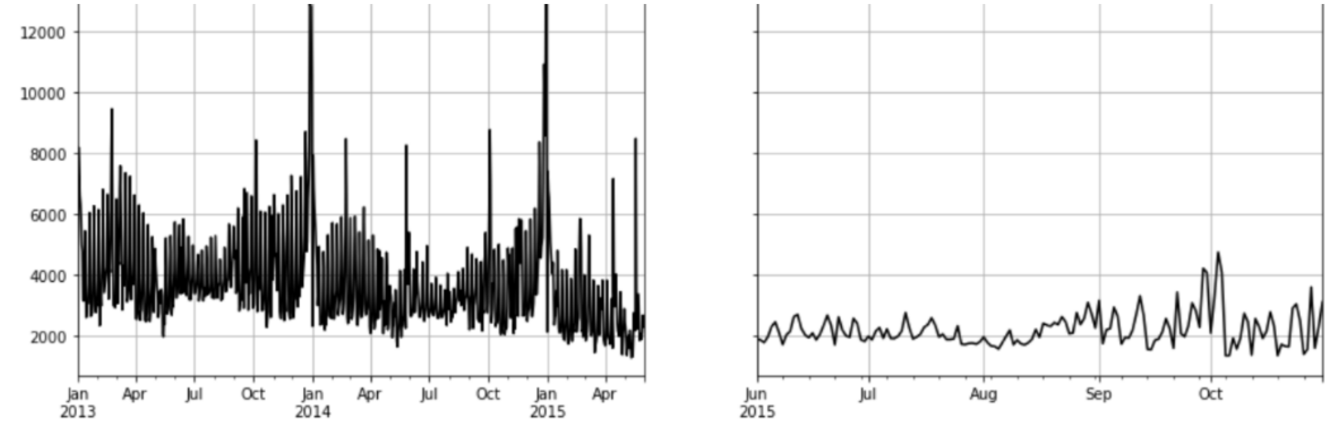
First things first, we need to split train / test set and we are going to write some useful functions to evaluate the performance of each algorithm. Just like we did in previous tutorials, we'll write a flexible function useful for any kind of time series data (date-time index, numeric index, …)

```
'''
Split train/test from any given data point.
:parameter
    :param ts: pandas Series
    :param test: num or str - test size (ex. 0.20) or index position
                 (ex. "yyyy-mm-dd", 1000)
:return
    ts_train, ts_test
'''
def split_train_test(ts, test=0.20, plot=True, figsize=(15,5)):
    ## define splitting point
    if type(test) is float:
        split = int(len(ts)*(1-test))
        perc = test
    elif type(test) is str:
        split = ts.reset_index()[
                      ts.reset_index().iloc[:,0]==test].index[0]
        perc = round(len(ts[split:])/len(ts), 2)
    else:
        split = test
        perc = round(len(ts[split:])/len(ts), 2)
    print("--- splitting at index: ", split, "|",
          ts.index[split], "| test size:", perc, " ---")

    ## split ts
    ts_train = ts.head(split)
    ts_test = ts.tail(len(ts)-split)
    if plot is True:
        fig, ax = plt.subplots(nrows=1, ncols=2, sharex=False,
                               sharey=True, figsize=figsize)
        ts_train.plot(ax=ax[0], grid=True, title="Train",
                      color="black")
        ts_test.plot(ax=ax[1], grid=True, title="Test",
                     color="black")
        ax[0].set(xlabel=None)
        ax[1].set(xlabel=None)
        plt.show()

    return ts_train, ts_test
```

Let's split the data:

```
ts_train, ts_test = split_train_test(ts, test="2015-06-01")
```






Now, the function to evaluate the models: it's a function that expects a dataframe as input with input data (column "ts"), fitted values (column "models"), predictions (column "forecast")

```
'''
Evaluation metrics for predictions.
:parameter
    :param dtf: DataFrame with columns raw values, fitted training
                 values, predicted test values
:return
    dataframe with raw ts and forecast
'''
def utils_evaluate_forecast(dtf, title, plot=True, figsize=(20,13)):
    try:
        ## residuals
        dtf["residuals"] = dtf["ts"] - dtf["model"]
        dtf["error"] = dtf["ts"] - dtf["forecast"]
        dtf["error_pct"] = dtf["error"] / dtf["ts"]

        ## kpi
        residuals_mean = dtf["residuals"].mean()
        residuals_std = dtf["residuals"].std()
        error_mean = dtf["error"].mean()
        error_std = dtf["error"].std()
        mae = dtf["error"].apply(lambda x: np.abs(x)).mean()
        mape = dtf["error_pct"].apply(lambda x: np.abs(x)).mean()
        mse = dtf["error"].apply(lambda x: x**2).mean()
        rmse = np.sqrt(mse)  #root mean squared error

        ## intervals
        dtf["conf_int_low"] = dtf["forecast"] - 1.96*residuals_std
        dtf["conf_int_up"] = dtf["forecast"] + 1.96*residuals_std
        dtf["pred_int_low"] = dtf["forecast"] - 1.96*error_std
        dtf["pred_int_up"] = dtf["forecast"] + 1.96*error_std

        ## plot
        if plot==True:
```

```
            fig = plt.figure(figsize=figsize)
            fig.suptitle(title, fontsize=20)
            ax1 = fig.add_subplot(2,2, 1)
            ax2 = fig.add_subplot(2,2, 2, sharey=ax1)
            ax3 = fig.add_subplot(2,2, 3)
            ax4 = fig.add_subplot(2,2, 4)
            ### training
            dtf[pd.notnull(dtf["model"])]
[["ts","model"]].plot(color=["black","green"], title="Model", 
grid=True, ax=ax1)
            ax1.set(xlabel=None)
            ### test
            dtf[pd.isnull(dtf["model"])]
[["ts","forecast"]].plot(color=["black","red"], title="Forecast", 
grid=True, ax=ax2)
            ax2.fill_between(x=dtf.index, y1=dtf['pred_int_low'], 
y2=dtf['pred_int_up'], color='b', alpha=0.2)
            ax2.fill_between(x=dtf.index, y1=dtf['conf_int_low'], 
y2=dtf['conf_int_up'], color='b', alpha=0.3)
            ax2.set(xlabel=None)
            ### residuals
            dtf[["residuals","error"]].plot(ax=ax3, color=
["green","red"], title="Residuals", grid=True)
            ax3.set(xlabel=None)
            ### residuals distribution
            dtf[["residuals","error"]].plot(ax=ax4, color=
["green","red"], kind='kde', title="Residuals Distribution", 
grid=True)
            ax4.set(ylabel=None)
            plt.show()
            print("Training --> Residuals mean:", 
np.round(residuals_mean), " | std:", np.round(residuals_std))
            print("Test --> Error mean:", np.round(error_mean), " | 
std:", np.round(error_std),
                  " | mae:",np.round(mae), " | 
mape:",np.round(mape*100), "%  | mse:",np.round(mse), " | 
rmse:",np.round(rmse))

        return 
dtf[["ts","model","residuals","conf_int_low","conf_int_up", 

"forecast","error","pred_int_low","pred_int_up"]]

    except Exception as e:
        print("--- got error ---")
        print(e)
```

We will use this later.

## ARIMA

$$y[t+1] = (c + a0^* y[t] + a1^* y[t-1] + ... + ap^* y[t-p]) + (e[t] + b1^* e[t-1] + b2^* e[t-2] + ... + bq^* e[t-q])$$

The hard part of modeling Arima is to find the right parameters combination. Luckily there is a package that does that job for us: pmdarima

```
best_model = pmdarima.auto_arima(ts, exogenous=exog,
                                 seasonal=True, stationary=False,
                                 m=7, information_criterion='aic',
                                 max_order=20,
                                 max_p=10, max_d=3, max_q=10,
                                 max_P=10, max_D=3, max_Q=10,
                                 error_action='ignore')
print("best model --> (p, d, q):", best_model.order, " and  (P, D,
      Q, s):", best_model.seasonal_order)
```

```
best model --> (p, d, q): (1, 1, 1)  and  (P, D, Q, s): (1, 0, 1, 7)
```

| | | | |
|---|---|---|---|
| **Dep. Variable:** | y | **No. Observations:** | 881 |
| **Model:** | SARIMAX(1, 1, 1)x(1, 0, 1, 7) | **Log Likelihood** | -7268.748 |
| **Date:** | Wed, 26 Feb 2020 | **AIC** | 14549.497 |
| **Time:** | 17:44:07 | **BIC** | 14578.176 |
| **Sample:** | 0 | **HQIC** | 14560.464 |
| | - 881 | | |
| **Covariance Type:** | opg | | |

| | **coef** | **std err** | **z** | **P>\|z\|** | **[0.025** | **0.975]** |
|---|---|---|---|---|---|---|
| **intercept** | -0.0045 | 0.005 | -0.926 | 0.354 | -0.014 | 0.005 |
| **ar.L1** | 0.6900 | 0.012 | 56.023 | 0.000 | 0.666 | 0.714 |
| **ma.L1** | -1.0000 | 0.019 | -51.446 | 0.000 | -1.038 | -0.962 |
| **ar.S.L7** | 0.9927 | 0.003 | 373.008 | 0.000 | 0.988 | 0.998 |
| **ma.S.L7** | -0.8856 | 0.018 | -48.105 | 0.000 | -0.922 | -0.850 |
| **sigma2** | 8.639e+05 | 2.28e-08 | 3.78e+13 | 0.000 | 8.64e+05 | 8.64e+05 |

| | | | | | | |
|---|---|---|---|---|---|---|
| sigma2 | 8.639e+05 | 2.28e-08 | 3.78e+13 | 0.000 | 8.64e+05 | 8.64e+05 |

| | | | |
|---|---|---|---|
| **Ljung-Box (Q):** | 52.97 | **Jarque-Bera (JB):** | 9730.31 |
| **Prob(Q):** | 0.08 | **Prob(JB):** | 0.00 |
| **Heteroskedasticity (H):** | 2.02 | **Skew:** | 1.18 |
| **Prob(H) (two-sided):** | 0.00 | **Kurtosis:** | 19.12 |

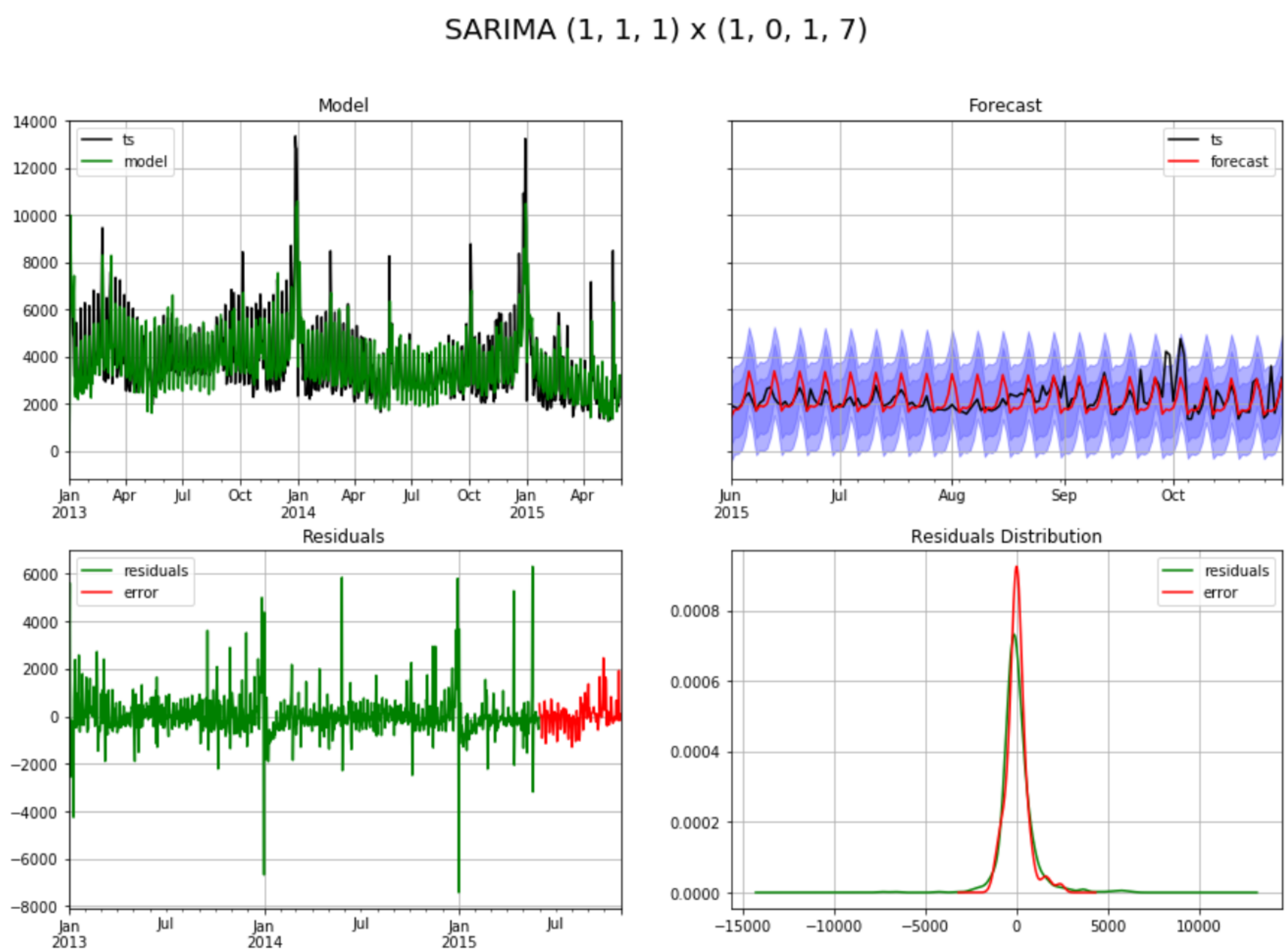
It’s time to build and train the model and evualuate the predictions on the test set:

```
'''
Fit SARIMAX (Seasonal ARIMA with External Regressors):
    y[t+1] = (c + a0*y[t] + a1*y[t-1] +...+ ap*y[t-p]) + (e[t] +
              b1*e[t-1] + b2*e[t-2] +...+ bq*e[t-q]) + (B*X[t])
:parameter
    :param ts_train: pandas timeseries
    :param ts_test: pandas timeseries
    :param order: tuple - ARIMA(p,d,q) --> p: lag order (AR), d:
                  degree of differencing (to remove trend), q: order
                  of moving average (MA)
    :param seasonal_order: tuple - (P,D,Q,s) --> s: number of
                  observations per seasonal (ex. 7 for weekly
                  seasonality with daily data, 12 for yearly
                  seasonality with monthly data)
    :param exog_train: pandas dataframe or numpy array
    :param exog_test: pandas dataframe or numpy array
:return
    dtf with predictons and the model
'''
def fit_sarimax(ts_train, ts_test, order=(1,0,1),
                seasonal_order=(0,0,0,0), exog_train=None,
                exog_test=None, figsize=(15,10)):
    ## train
    model = smt.SARIMAX(ts_train, order=order,
                        seasonal_order=seasonal_order,
                        exog=exog_train, enforce_stationarity=False,
                        enforce_invertibility=False).fit()
    dtf_train = ts_train.to_frame(name="ts")
    dtf_train["model"] = model.fittedvalues

    ## test
    dtf_test = ts_test.to_frame(name="ts")
    dtf_test["forecast"] = model.predict(start=len(ts_train),
                            end=len(ts_train)+len(ts_test)-1,
                            exog=exog_test)

    ## evaluate
```

```
    dtf = dtf_train.append(dtf_test)
    title = "ARIMA "+str(order) if exog_train is None else
            "ARIMAX "+str(order)
    title = "S"+title+" x "+str(seasonal_order) if
            np.sum(seasonal_order) > 0 else title
    dtf = utils_evaluate_forecast(dtf, figsize=figsize, title=title)
    return dtf, model
```

Let's fit the model on the train set and forecast the same period of the test set:

```
dtf, model = fit_sarimax(ts_train, ts_test, order=(1,1,1),
                         seasonal_order=(1,0,1,7))
```

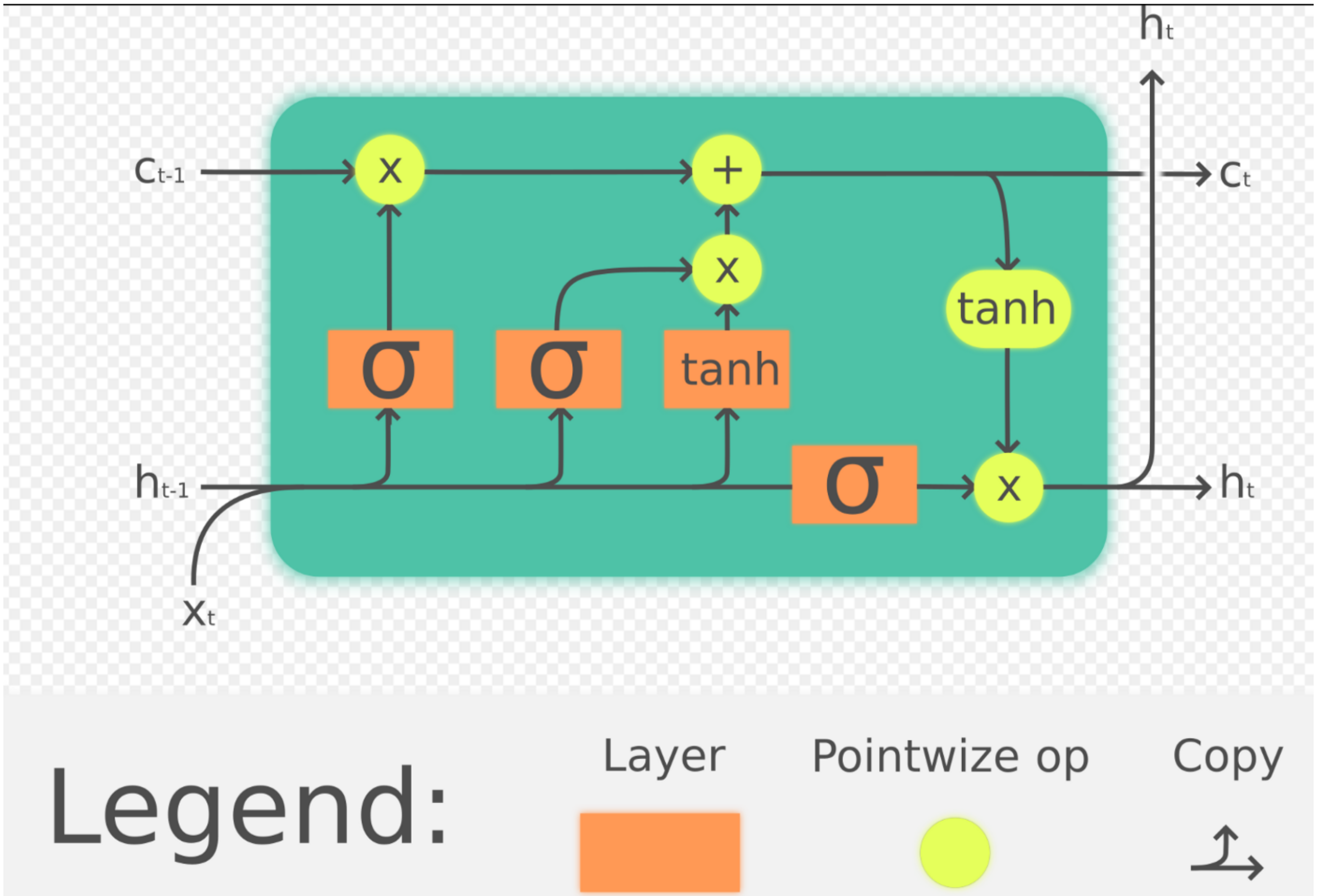



```
Training --> Residuals mean: 25.0  | std: 955.0
Test --> Error mean: 29.0  | std: 592.0  | mae: 394.0  | mape: 17.0 %  | mse: 348871.0  | rmse: 591.0
```

Not bad: when forecasting, the average error of prediction in 394 unit of sales (17% of the predicted value).

## LSTM

The long-short term memory is a type of recurrent neural netwok that stores past observations into its memory and during training it learns when to use this memory. A lstm layer has this structure:




Source: Wikipedia

I will use a really simple neural network: a lstm layer + a fully connected output layer with an input of dimension 365, meaning that it will have a whole year as memory (losing 365 days of training).

```
## Keras models
model = models.Sequential()
model.add( layers.LSTM(input_shape=(1,365), units=50,
           activation='relu', return_sequences=False) )
model.add( layers.Dense(1) )
model.compile(optimizer='adam', loss='mean_absolute_error')
```

In order to fit the model, a bit of preprocessing is necessary:

```
'''
Preprocess a ts partitioning into X and y.
:parameter
    :param ts: pandas timeseries
    :param s: num - number of observations per seasonal (ex. 7 for
weekly seasonality with daily data, 12 for yearly seasonality with
monthly data)
    :param scaler: sklearn scaler object - if None is fitted
    :param exog: pandas dataframe or numpy array
:return
    X, y, scaler
'''
def utils_preprocess_ts(ts, s, scaler=None, exog=None):
    ## scale
    if scaler is None:
        scaler = preprocessing.MinMaxScaler(feature_range=(0,1))
    ts_preprocessed =
scaler.fit_transform(ts.values.reshape(-1,1)).reshape(-1)

    ## create X,y for train
    ts_preprocessed =
kprocessing.sequence.TimeseriesGenerator(data=ts_preprocessed,

targets=ts_preprocessed,

length=s, batch_size=1)
    lst_X, lst_y = [], []
    for i in range(len(ts_preprocessed)):
        xi, yi = ts_preprocessed[i]
        lst_X.append(xi)
        lst_y.append(yi)
    X = np.array(lst_X)
    y = np.array(lst_y)
    return X, y, scaler

'''
Get fitted values.
'''
def utils_fitted_lstm(ts, model, scaler, exog=None):
    ## scale
    ts_preprocessed =
scaler.fit_transform(ts.values.reshape(-1,1)).reshape(-1)

    ## create Xy, predict = fitted
    s = model.input_shape[-1]
    lst_fitted = [np.nan]*s
    for i in range(len(ts_preprocessed)):
        end_ix = i + s
        if end_ix > len(ts_preprocessed)-1:
```

```
                break
            X = ts_preprocessed[i:end_ix]
            X = np.array(X)
            X = np.reshape(X, (1,1,X.shape[0]))
            fit = model.predict(X)
            fit = scaler.inverse_transform(fit)[0][0]
            lst_fitted.append(fit)
        return np.array(lst_fitted)

'''
Predict ts using previous predictions.
'''
def utils_predict_lstm(ts, model, scaler, pred_ahead, exog=None):
    ## scale
    s = model.input_shape[-1]
    ts_preprocessed = list(scaler.fit_transform(ts[-
s:].values.reshape(-1,1)))

    ## predict, append, re-predict
    lst_preds = []
    for i in range(pred_ahead):
        X = np.array(ts_preprocessed[len(ts_preprocessed)-s:])
        X = np.reshape(X, (1,1,X.shape[0]))
        pred = model.predict(X)
        ts_preprocessed.append(pred)
        pred = scaler.inverse_transform(pred)[0][0]
        lst_preds.append(pred)
    return np.array(lst_preds)
```

We can finally write the function to fit the model

```
'''
Fit Long short-term memory neural network.
:parameter
    :param ts: pandas timeseries
    :param exog: pandas dataframe or numpy array
    :param s: num - number of observations per seasonal (ex. 7 for
weekly seasonality with daily data, 12 for yearly seasonality with
monthly data)
:return
    generator, scaler
'''
def fit_lstm(ts_train, ts_test, model, exog=None, s=20,
             figsize=(15,5)):
    ## check
    print("Seasonality: using the last", s, "observations to predict
           the next 1")

    ## preprocess train
    X_train, y_train, scaler = utils_preprocess_ts(ts_train,
```

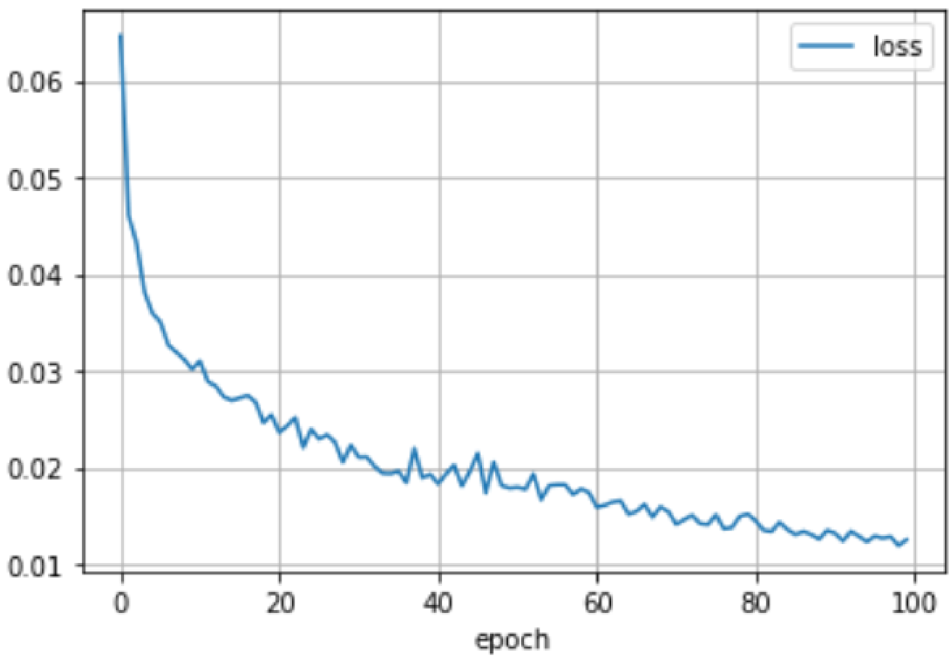
```
                                         scaler=None, exog=exog, s=s)

    ## lstm
    if model is None:
        model = models.Sequential()
        model.add( layers.LSTM(input_shape=X_train.shape[1:], 
units=50, activation='relu', return_sequences=False) )
        model.add( layers.Dense(1) )
        model.compile(optimizer='adam', loss='mean_absolute_error')

    ## train
    print(model.summary())
    training = model.fit(x=X_train, y=y_train, batch_size=1, 
                         epochs=100, shuffle=True, verbose=0, 
                         validation_split=0.3)

    dtf_train = ts_train.to_frame(name="ts")
    dtf_train["model"] = utils_fitted_lstm(ts_train, training.model, 
                                           scaler, exog)
    dtf_train["model"] = dtf_train["model"].fillna(method='bfill')

    ## test
    preds = utils_predict_lstm(ts_train[-s:], training.model, 
                               scaler, pred_ahead=len(ts_test), 
                               exog=None)
    dtf_test = ts_test.to_frame(name="ts").merge(
                       pd.DataFrame(data=preds, index=ts_test.index, 
                                    columns=["forecast"]), 
               how='left', left_index=True, right_index=True)

    ## evaluate
    dtf = dtf_train.append(dtf_test)
    dtf = utils_evaluate_forecast(dtf, figsize=figsize, 
                                  title="LSTM (memory:"+str(s)+")")
    return dtf, training.model
```

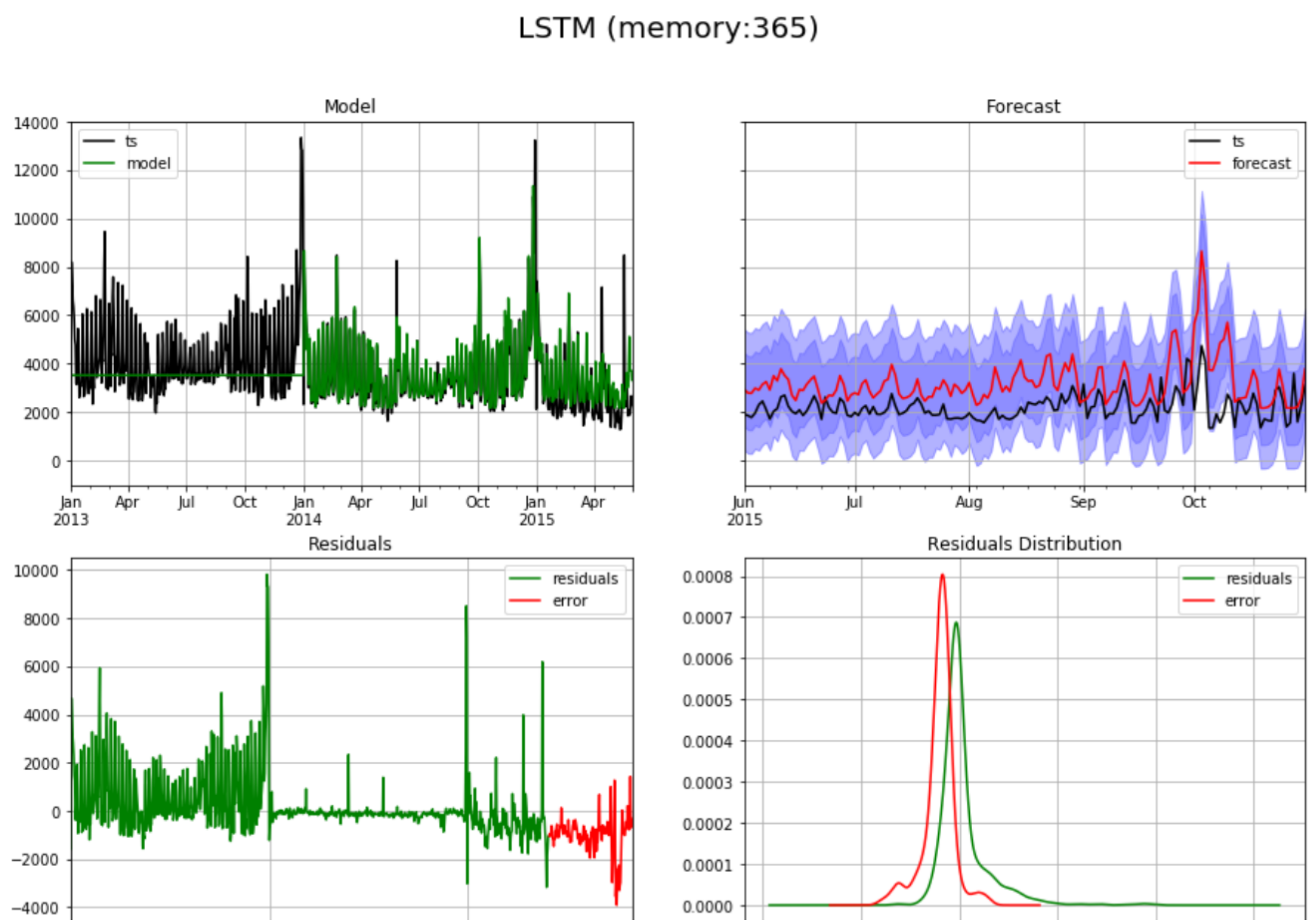
And run it

```
dtf, model = fit_lstm(ts_train, ts_test, model, s=365)
```

```
Seasonality: using the last 365 observations to predict the next 1
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 50)                83200
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 51
```

```
=================================================================
Total params: 83,251
Trainable params: 83,251
Non-trainable params: 0
_________________________________________________________________
None
```




```
Training --> Residuals mean: 210.0  | std: 1274.0
Test --> Error mean: -1004.0  | std: 778.0  | mae: 1077.0  | mape: 51.0 %  | mse: 1608403.0  | rmse: 1268.0
```

The average error of prediction in 1077 unit of sales (51% of the predicted value).

## PROPHET

The Facebook Prophet model is composed of 3 componentes:

$$y = \text{trend} + \text{seasonality} + \text{holidays}$$

Don't forget: the package takes a data frame as input (not a pandas Series) with 2 columns (ds with dates and y with values)

```
dtf_train = ts_train.reset_index().rename(columns={"date":"ds",
                                                    "sales":"y"})
dtf_test = ts_test.reset_index().rename(columns={"date":"ds",
                                                  "sales":"y"})
dtf_train.tail()
```

| | ds | y |
|---|---|---|
| 876 | 2015-05-27 | 1953.0 |
| 877 | 2015-05-28 | 1885.0 |
| 878 | 2015-05-29 | 2146.0 |
| 879 | 2015-05-30 | 2665.0 |
| 880 | 2015-05-31 | 2283.0 |

The package has a lot of parameters , so I suggest to go look them up on the official website or github repo. Here I will use the basic and standard configuration:

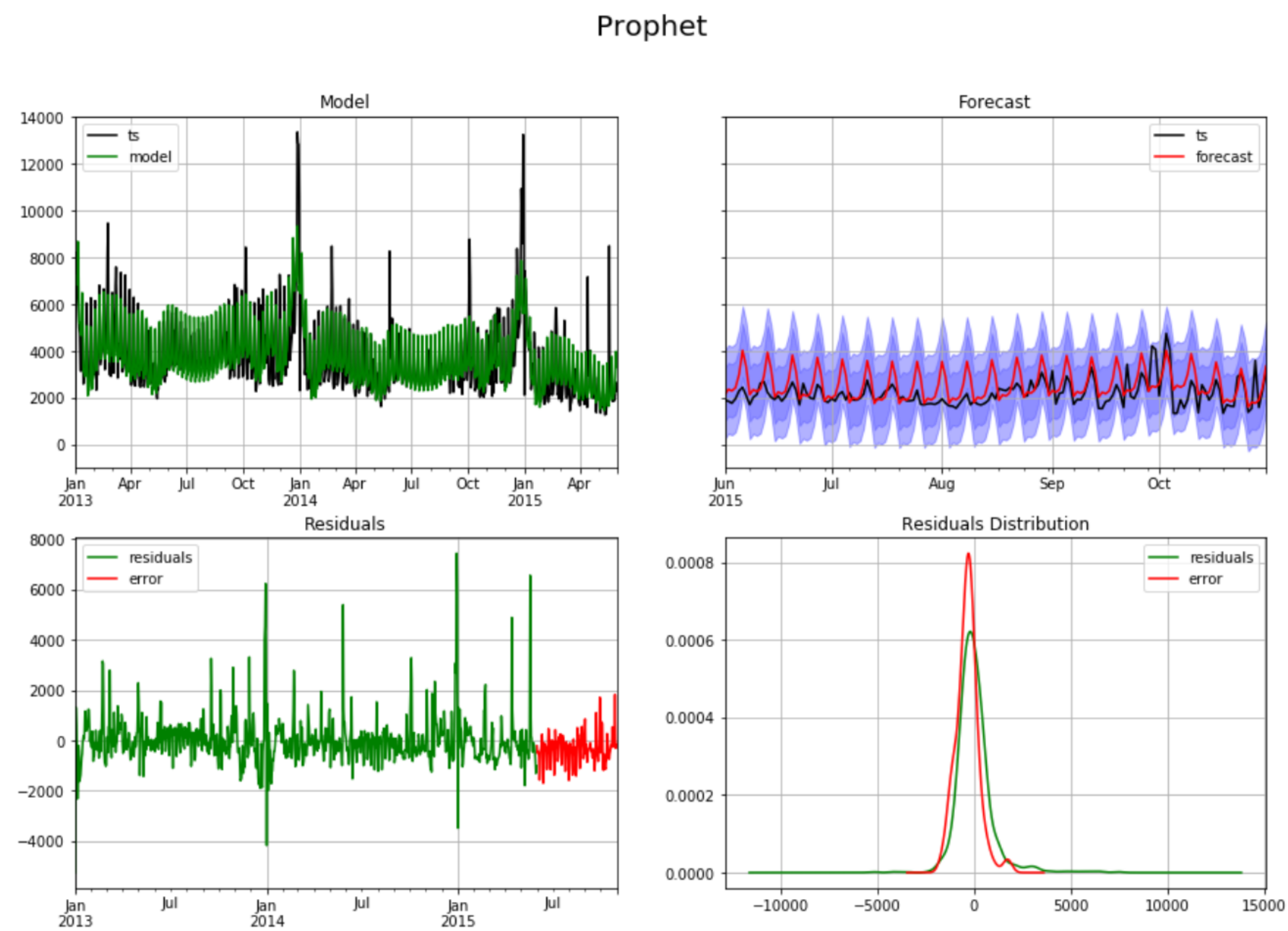
```
model = Prophet(growth="linear", changepoints=None,
                n_changepoints=25,
                seasonality_mode="multiplicative",
                yearly_seasonality="auto",
                weekly_seasonality="auto",
                daily_seasonality=False,
                holidays=None)
```

Let’s wrtie the function to fit and test the model

```
'''
Fits prophet on Business Data:
    y = trend + seasonality + holidays
:parameter
    :param dtf_train: pandas Dataframe with columns 'ds' (dates),
             'y' (values), 'cap' (capacity if growth="logistic"),
             other additional regressor
    :param dtf_test: pandas Dataframe with columns 'ds' (dates), 'y'
                     (values), 'cap' (capacity if
                      growth="logistic"),
                      other additional regressor
    :param lst_exog: list - names of variables
    :param freq: str - "D" daily, "M" monthly, "Y" annual, "MS"
                       monthly start ...
:return
    dtf with predictons and the model
'''
def fit_prophet(dtf_train, dtf_test, lst_exog=None, model=None,
                freq="D", figsize=(15,10)):
    ## train
    model.fit(dtf_train)

    ## test
    dtf_prophet = model.make_future_dataframe(periods=len(dtf_test),
                  freq=freq, include_history=True)
    dtf_prophet = model.predict(dtf_prophet)
    dtf_train = dtf_train.merge(dtf_prophet[["ds","yhat"]],
                how="left").rename(columns={'yhat':'model',
                'y':'ts'}).set_index("ds")
    dtf_test = dtf_test.merge(dtf_prophet[["ds","yhat"]],
                how="left").rename(columns={'yhat':'forecast',
                'y':'ts'}).set_index("ds")

    ## evaluate
    dtf = dtf_train.append(dtf_test)
    dtf = utils_evaluate_forecast(dtf, figsize=figsize,
                                  title="Prophet")
    return dtf, model
```

Let’s run it:

```
dtf, model = fit_prophet(dtf_train, dtf_test, model=model, freq="D")
```




```
Training --> Residuals mean: -2.0  | std: 963.0
Test --> Error mean: -356.0  | std: 581.0  | mae: 534.0  | mape: 25.0 %  | mse: 461781.0  | rmse: 680.0
```

Good, the average error of prediction in 534 unit of sales (25% of the predicted value).

## Forecast unknown future

The winner is …. Arima !!! The Arima algorithm is the one who performed better on the test set. But let’s do a final test: forecasting the unknown future. In particular I want to see if these models will predict a peak in January, like how it was on Jan 2014 and Jan 2015.

Let’s write the functions to forecast the unknown. First of all we need a function to create future date index

```
'''
Generate dates to index predictions.
:parameter
    :param start: str - "yyyy-mm-dd"
    :param end: str - "yyyy-mm-dd"
    :param n: num - length of index
    :param freq: None or str - 'B' business day, 'D' daily, 'W'
                 weekly, 'M' monthly, 'A' annual, 'Q' quarterly
'''
def utils_generate_indexdate(start, end=None, n=None, freq="D"):
    if end is not None:
        index = pd.date_range(start=start, end=end, freq=freq)
    else:
        index = pd.date_range(start=start, periods=n, freq=freq)
    index = index[1:]
    print("--- generating index date --> start:", index[0],
          "| end:", index[-1], "| len:", len(index), "---")
    return index
```

Let’s write a function to plot the predictions with the date index from the function above

```
'''
Plot unknown future forecast.
'''
def utils_plot_forecast(dtf, zoom=30, figsize=(15,5)):
    ## interval
    dtf["residuals"] = dtf["ts"] - dtf["model"]
    dtf["conf_int_low"] = dtf["forecast"] -
1.96*dtf["residuals"].std()
    dtf["conf_int_up"] = dtf["forecast"] +
1.96*dtf["residuals"].std()
    fig, ax = plt.subplots(nrows=1, ncols=2, figsize=figsize)

    ## entire series
    dtf[["ts","forecast"]].plot(color=["black","red"], grid=True,
ax=ax[0], title="History + Future")
    ax[0].fill_between(x=dtf.index, y1=dtf['conf_int_low'],
y2=dtf['conf_int_up'], color='b', alpha=0.3)

    ## focus on last
    first_idx = dtf[pd.notnull(dtf["forecast"])].index[0]
    first_loc = dtf.index.tolist().index(first_idx)
    zoom_idx = dtf.index[first_loc-zoom]
    dtf.loc[zoom_idx:][["ts","forecast"]].plot(color=
["black","red"], grid=True, ax=ax[1], title="Zoom on the last
"+str(zoom)+" observations")
    ax[1].fill_between(x=dtf.loc[zoom_idx:].index,
y1=dtf.loc[zoom_idx:]['conf_int_low'],
                       y2=dtf.loc[zoom_idx:]['conf_int_up'],
```

```
color='b', alpha=0.3)
    plt.show()
    return
dtf[["ts","model","residuals","conf_int_low","forecast","conf_int_up
"]]
```

One last step: we need the functions to do the actual forecast

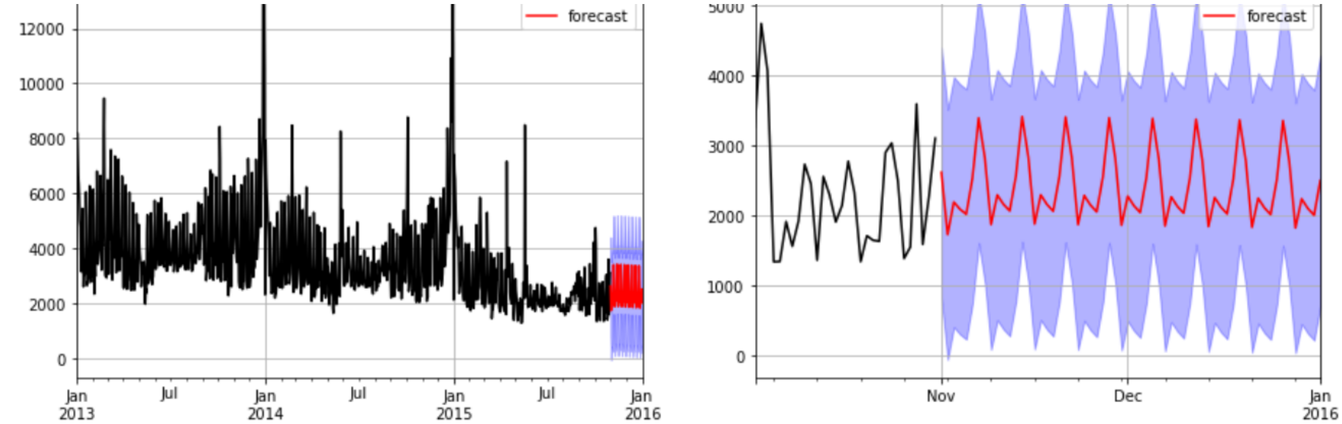
```
'''
Forecast unknown future.
'''
def forecast_arima(ts, model, pred_ahead=None, end=None, freq="D",
zoom=30, figsize=(15,5)):
    ## fit
    model = model.fit()
    dtf = ts.to_frame(name="ts")
    dtf["model"] = model.fittedvalues
    dtf["residuals"] = dtf["ts"] - dtf["model"]

    ## index
    index = utils_generate_indexdate(start=ts.index[-1], end=end,
n=pred_ahead, freq=freq)

    ## forecast
    preds = model.forecast(len(index))
    dtf = dtf.append(preds.to_frame(name="forecast"))

    ## plot
    dtf = utils_plot_forecast(dtf, zoom=zoom)
    return dtf

'''
Forecast unknown future.
'''
def forecast_lstm(ts, model, pred_ahead=None, end=None, freq="D",
zoom=30, figsize=(15,5)):
    ## fit
    s = model.input_shape[-1]
    X, y, scaler = utils_preprocess_ts(ts, scaler=None, exog=None,
s=s)
    training = model.fit(x=X, y=y, batch_size=1, epochs=100,
shuffle=True, verbose=0, validation_split=0.3)
    dtf = ts.to_frame(name="ts")
    dtf["model"] = utils_fitted_lstm(ts, training.model, scaler,
None)
    dtf["model"] = dtf["model"].fillna(method='bfill')

    ## index
    index = utils_generate_indexdate(start=ts.index[-1], end=end,
n=pred_ahead, freq=freq)
```

```
    ## forecast
    preds = utils_predict_lstm(ts[-s:], training.model, scaler, 
pred_ahead=len(index), exog=None)
    dtf = dtf.append(pd.DataFrame(data=preds, index=index, columns=
["forecast"]))
    
    ## plot
    dtf = utils_plot_forecast(dtf, zoom=zoom)
    return dtf

'''
Forecast unknown future.
'''
def forecast_prophet(dtf, model, pred_ahead=None, end=None, 
freq="D", zoom=30, figsize=(15,5)):
    ## fit
    model.fit(dtf)
    
    ## index
    index = utils_generate_indexdate(start=dtf["ds"].values[-1], 
end=end, n=pred_ahead, freq=freq)
    
    ## forecast
    dtf_prophet = model.make_future_dataframe(periods=len(index), 
freq=freq, include_history=True)
    dtf_prophet = model.predict(dtf_prophet)
    dtf = dtf.merge(dtf_prophet[["ds","yhat"]], 
how="left").rename(columns={'yhat':'model', 
'y':'ts'}).set_index("ds")
    preds = pd.DataFrame(data=index, columns=["ds"])
    preds = preds.merge(dtf_prophet[["ds","yhat"]], 
how="left").rename(columns={'yhat':'forecast'}).set_index("ds")
    dtf = dtf.append(preds)
    
    ## plot
    dtf = utils_plot_forecast(dtf, zoom=zoom)
    return dtf
```

Let’s try them:

```
model = smt.SARIMAX(ts, order=(1,1,1), seasonal_order=(1,0,1,7))

future = forecast_arima(ts, model, end="2016-01-01")
```

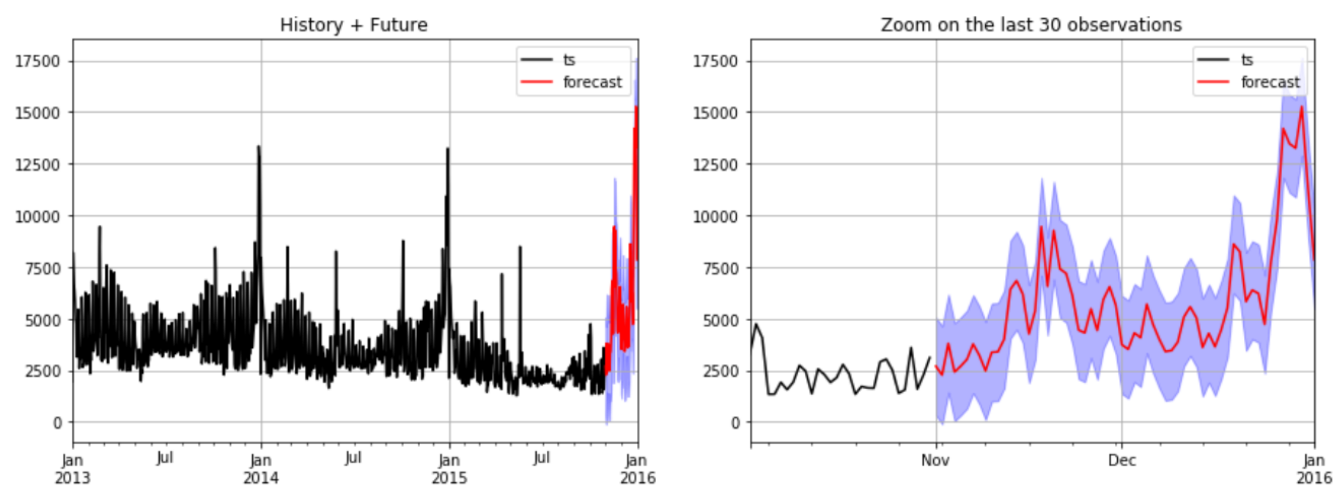
```
--- generating index date --> start: 2015-11-01 00:00:00 | end: 2016-01-01 00:00:00 | len: 62 ---
```




Arima predicts that the series will keep the downtrend and there won't be any peaks next January.

```
model = models.Sequential()
model.add( layers.LSTM(input_shape=(1,365), units=50,
                       activation='relu', return_sequences=False) )
model.add( layers.Dense(1) )
model.compile(optimizer='adam', loss='mean_absolute_error')

future = forecast_lstm(ts, model, end="2016-01-01")
```

```
--- generating index date --> start: 2015-11-01 00:00:00 | end: 2016-01-01 00:00:00 | len: 62 ---
```

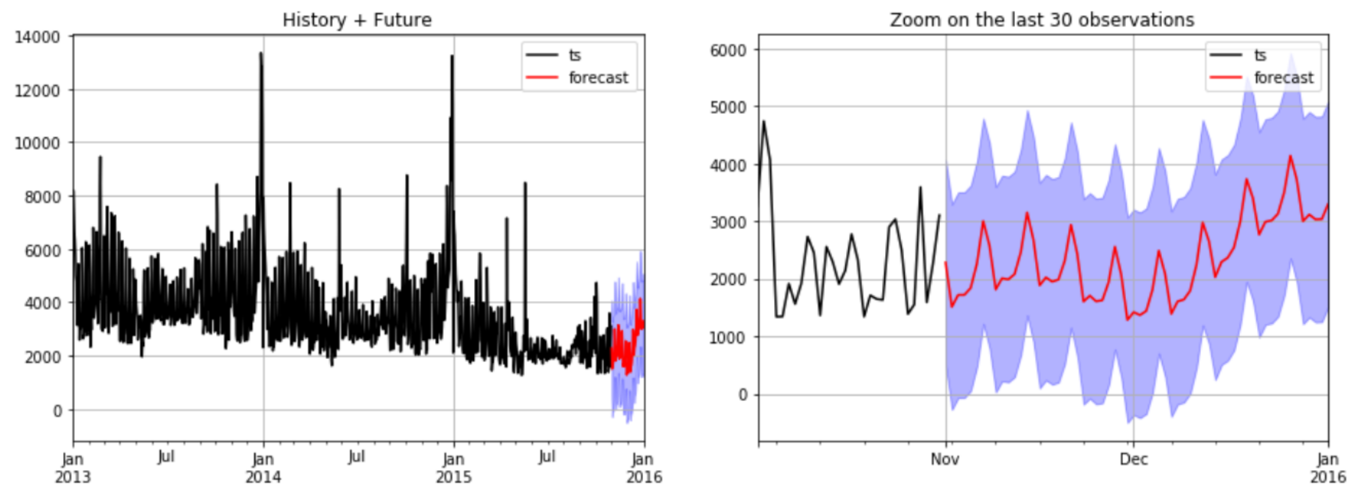



Lstm with 1 year of memory predicts that the series will replicate yearly seasonality, with peaks in January.

```
model = Prophet(growth="linear", changepoints=None,
                n_changepoints=25,
```

```
                      seasonality_mode="multiplicative",
                      yearly_seasonality="auto",
                      weekly_seasonality="auto", daily_seasonality=False,
                      holidays=None)

future = forecast_prophet(dtf, model, end="2016-01-01")
```

```
--- generating index date --> start: 2015-11-01 00:00:00 | end: 2016-01-01 00:00:00 | len: 62 ---
```




Prophet predicts that the series will keep the downtrend but includes yearly seasonality, with smaller peaks in January. This one looks more realistic.

> *This article is part of the series* ***Time Series Forecasting with Python****, see also:*

## Time Series Analysis for Machine Learning

Time Series Forecasting with Machine Learning and Python

towardsdatascience.com

## Time Series Forecasting with Random Walk

Time Series Forecasting with Machine Learning and Python

medium.com

Time series forecasting with simple Parametric Curve Fitting

Predict when the COVID-19 pandemic will stop in your country

medium.com

Contacts: LinkedIn | Twitter



Data Science Machine Learning Programming Deep Learning Data Visualization