Time Series Forecasting using Prophet

January 15, 2021

Contents

1 Time Series Forecasting using Prophet (`fbprophet`)

The following coursebook is produced by the team at **Algoritma** for **BRI Data Hackathon 2021 Workshop: Data Science Report for Time Series Analysis**. The coursebook is intended for a restricted audience only, i.e. the individuals having received this coursebook directly from the training organization. It may not be reproduced, distributed, translated or adapted in any form outside these individuals and organizations without permission.

Algoritma is a data science education center based in Jakarta. We organize workshops and training programs to help working professionals and students gain mastery in various data science sub-fields: data visualization, machine learning, data modeling, statistical inference, etc.

---

2 Preface

Before you go ahead and run the codes in this coursebook, it's often a good idea to go through some initial setup. Under the Training Objectives section we'll outline the syllabus, identify the key objectives and set up expectations for each module. Under the Libraries and Setup section you'll see some code to initialize our workspace and the libraries we'll be using for the projects. You may want to make sure that the libraries are installed beforehand by referring back to the packages listed here.

2.1 Introduction

Time series data is one of the most common form of data to be found in every industry. It is considered to be a significant area of interest for most industries: retail, telecommunication, logistic, engineering, finance, and socio-economic. Time series analysis aims to extract the underlying components of a time series to better understand the nature of the data. In this workshop we will tackle a common industry case of business sales.

2.2 Training Objectives

- Working with Time Series
  - Data Preprocessing
  - Visualization: Multiple vs Multivariate Time Series
- Modeling using `fbprophet`
  - Baseline Model
  - Trend Component
  - Seasonality Component
  - Holiday Effects

    - Adding Regressors
- Forecasting Evaluation
    - Train-Test Split
    - Evaluation Metrics: RMSLE
    - Expanding Window Cross Validation
- Hyperparameter Tuning

2.3 Library and Setup

In this section you'll see some code to initialize our workspace, and the packages we'll be using for this project.

Since all local variables and files in Google Colab will be refreshed on each runtime, we'll run the code below to download the dataset from provided Google Drive ID. The downloaded file can be seen on Files menu.

```
Downloading…
From: https://drive.google.com/uc?id=1KWrgTxR-Dq04NvPLr2YKNupG7wv9zmoR
To: /content/sales_train.csv
94.6MB [00:00, 142MB/s]
```

---

3 Data Loading

In this course, we will be using provided data from one of the largest Russian software firms - 1C Company and is made available through Kaggle platform. This is a good example case of data since it contains seasonality and a particular 'noise' in several data when special occurences happened.

[ ]:

	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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2935849 entries, 0 to 2935848
Data columns (total 6 columns):
 #   Column          Dtype
---  ------          -----
 0   date            object
 1   date_block_num  int64
 2   shop_id         int64
 3   item_id         int64
 4   item_price      float64
 5   item_cnt_day    float64
dtypes: float64(2), int64(3), object(1)
memory usage: 134.4+ MB
```

Some insights we can get from the output are:

- The data consist of 2,935,849 observations (or rows)
- It has 6 variables (or columns)
- The following are the glossary provided in the Kaggle platform:
  - `date` is the date format provided in **dd.mm.yyyy** format
  - `date_block_num` is a consecutive month number used for convenience (January 2013 is 0, February 2013 is 1, and so on)
  - `shop_id` is the unique identifier of the shop
  - `item_id` is the unique identifier of the product
  - `item_price` is the price of the item on the specified date
  - `item_cnt_day` is the number of products sold on the specified date

The variable of interest that we are trying to predict is the **`item_cnt_day` and `item_price`**, which we'll see how to analyze the business demand from the sales record.

4 Data Preprocessing

Time series data is defined as data observations that are collected at **regular time intervals**. In this case, we are talking about software sales **daily** data.

We have to make sure our data is ready to be fitted into models, such as:

- Convert `date` column data type from `object` to `datetime64`
- Sort the data ascending by `date` column
- Feature engineering of `total_revenue`, which will be forecasted

The legal and cultural expectations for datetime format may vary between countries. In Indonesia for example, most people are used to storing dates in DMY order. `pandas` will infer date as a **month first** order by default. Since the sales `date` is stored in **dd.mm.yyyy** format, we have to specify parameter **`dayfirst=True`** inside `pd.to_datetime()` method.

Take a look on the third observation below; `pandas` converts it to 2nd January while the actual data represents February 1st.

```
[ ]: 0    2021-01-30
     1    2021-01-31
     2    2021-01-02
     3    2021-02-02
     dtype: datetime64[ns]
```

Next, let's check the `date` range of `sales` data. Turns out it ranges from January 1st, 2013 to October 31st, 2015.

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: FutureWarning:

Treating datetime data as categorical rather than numeric in `.describe` is
deprecated and will be removed in a future version of pandas. Specify
`datetime_is_numeric=True` to silence this warning and adopt the future behavior
now.
```

```
[ ]: count                   2935849
     unique                     1034
     top         2013-12-28 00:00:00
     freq                       9434
     first       2013-01-01 00:00:00
     last        2015-10-31 00:00:00
     Name: date, dtype: object
```

We are interested in analyzing the most popular shop (`shop_id`) of our `sales` data. The popularity of a shop is defined by the number of transaction that occur. Let's create a frequency table using `.value_counts()` as follows:

```
[ ]: 31    235636
     25    186104
     54    143480
     Name: shop_id, dtype: int64
```

We have gain the information that **shop 31, 25, and 54** are the top three shops with the most record sales. Say we would like to analyze their time series attribute. To do that, we can apply **conditional subsetting** (filter) to `sales` data.

```
[ ]:             date  date_block_num  …  item_cnt_day  total_revenue
     84820 2013-01-01               0  …           1.0          149.0
     85870 2013-01-01               0  …           1.0        11489.7
     87843 2013-01-01               0  …           1.0          349.0
     76711 2013-01-01               0  …           1.0          249.0
     87341 2013-01-01               0  …           1.0          298.0

     [5 rows x 7 columns]
```

Now let's highlight again the most important definition of a time series: it is **an observation** that is recorded at a regular time interval. Notice that the records has a multiple samples of the same day. This must mean that our data frame violates the rules of a time series where the records is sampled multiple time a day. Based on the structure of our data, it is recording the sales of different items within the same day. An important aspect in preparing a time series is called a **data aggregation**, where we need to aggregate the sales from one day into one records. Now let's take a look at the codes:

```
[ ]:         date  shop_id  total_qty  total_revenue
     0 2013-01-01       54      415.0      316557.00
     1 2013-01-02       25      568.0      345174.13
     2 2013-01-02       31      568.0      396376.10
     3 2013-01-02       54      709.0      519336.00
     4 2013-01-03       25      375.0      249421.00
```

Note that in performing data aggregation, we can only transform a more frequent data sampling to a more sparse frequency, for example:

- Hourly to daily

- Daily to weekly
- Daily to monthly
- Monthly to quarterly, and so on

---

5 Visualization

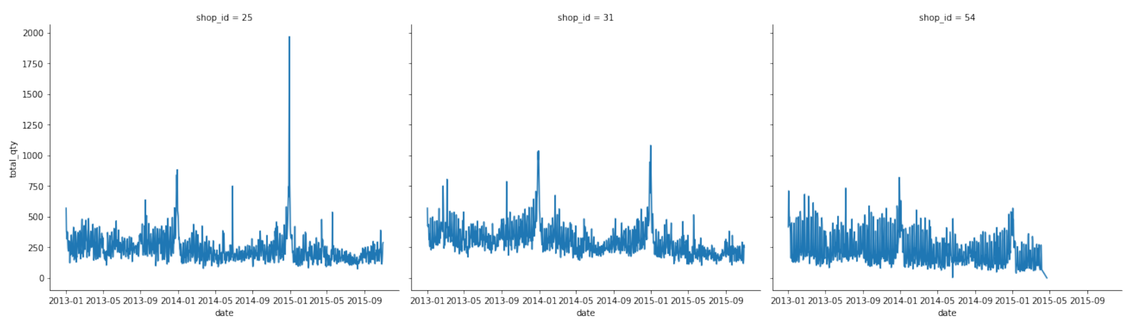
One of an important aspect in time series analysis is performing a visual exploratory analysis. Python is known for its graphical capabilities and has a very popular visualization package called **`matplotlib` and `seaborn`**. Let's take our `daily_sales` data frame we have created earlier and observe through the visualization.

There is a misconception between multiple and multivariate time series. Here are the definitions for each term:

- Multiple time series: There is **one variable** from **multiple objects** being observed from time to time.
- Multivariate Time series: There are **multiple variables** from only **one object** being observed from time to time. Typically for such series, the variables are closely interrelated.

5.1 Multiple Time Series

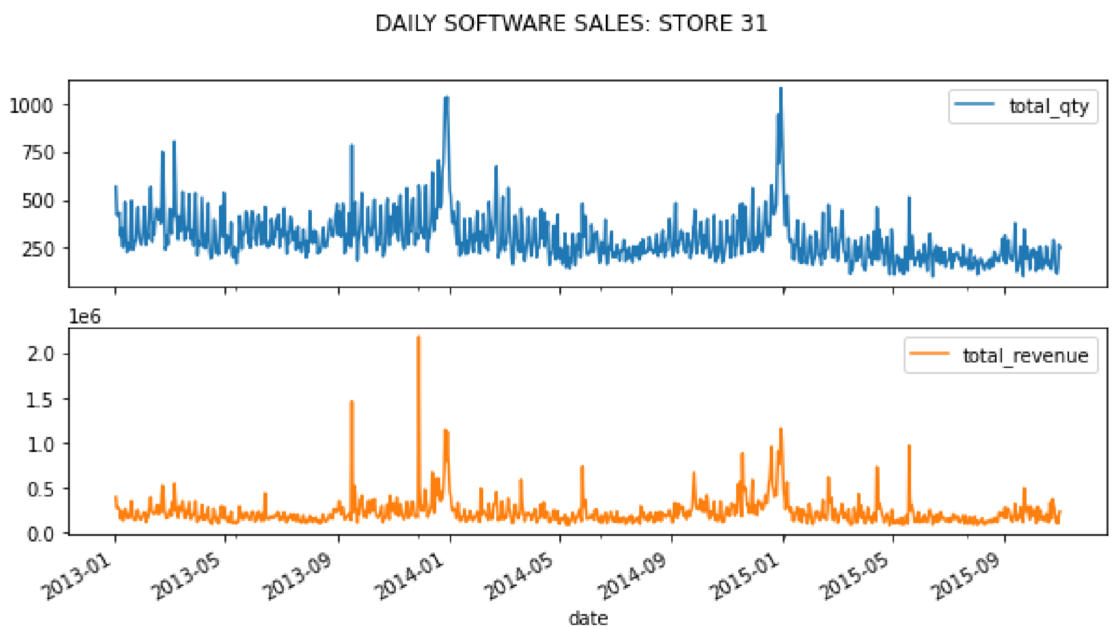
In our case, multiple time series is when we observed the fluctuation of `total_qty` over time, from the top three shops.



From the visualization we can conclude that the fluctuation of `total_qty` is very distinct for each shop. There are some extreme spikes on `shop_id` 25 and 31 at the end of each year, while `shop_id` 54 doesn't have any spike.

5.2 Multivariate Time Series

In our case, multivariate time series is when we observed the fluctuation of `total_qty` and `total_revenue` over time, from only `shop_id` 31. Notice that we perform conditional subsetting on `daily_sales` to produce `daily_sales_31`.



From the visualization we can conclude that the fluctuation of `total_qty` and `total_revenue` is quite similar for `shop_id` 31. In fact, from the business perspective, variable quantity and revenue are closely related to each other. When the `total_qty` sold increases, logically, the `total_revenue` will also increases.

6 Modeling using `fbprophet`

A very fundamental part in understanding time series is to be able to **decompose** its underlying components. A classic way in describing a time series is using **General Additive Model (GAM)**. This definition describes time series as a summation of its components. As a starter, we will define time series with 3 different components:

- Trend (T): Long term movement in its mean
- Seasonality (S): Repeated seasonal effects
- Residuals (E): Irregular components or random fluctuations not described by trend and seasonality

The idea of GAM is that each of them is added to describe our time series:

$Y(t) = T(t) + S(t) + E(t)$

When we are discussing time series forecasting there is one main assumption that needs to be remembered: **We assume correlation among successive observations**. Means that the idea of performing a forecasting for a time series is based on its past behavior. So in order to forecast the future values, we will take a look at any existing trend and seasonality of the time series and use it to generate future values.

Prophet enhanced the classical trend and seasonality components by adding a **holiday effect**. It will try to model the effects of holidays which occur on some dates and has been proven to be really useful in many cases. Take, for example: Lebaran Season. In Indonesia, it is really common to have an effect on Lebaran season. The effect, however, is a bit different from a classic seasonality effect because it shows the characteristics of an **irregular schedule**.

6.1 Baseline Model

6.1.1 Prepare the data

To use the `fbprophet` package, we first need to prepare our time series data into a specific format data frame required by the package. The data frame requires 2 columns:

- `ds`: the time stamp column, stored in `datetime64` data type
- `y`: the value to be forecasted

In this example, we will be using the `total_qty` as the value to be forecasted.

```
[ ]:           ds      y
     0 2013-01-02  568.0
     1 2013-01-03  423.0
     2 2013-01-04  431.0
     3 2013-01-05  415.0
     4 2013-01-06  435.0
```

6.1.2 Fitting Model

Let's initiate a `fbprophet` object using `Prophet()` and fit the `daily_total_qty` data. The idea of fitting a time series model is to extract the pattern information of a time series in order to perform a forecasting over the specified period of time.

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```

```
[ ]: <fbprophet.forecaster.Prophet at 0x7f5572235e80>
```

6.1.3 Forecasting

Based on the existing data, we'd like to perform a forecasting for **1 years into the future**. To do that, we will need to first prepare a data frame that consist of the future time stamp range we'd like to forecast. Luckily, `fbprophet` has provided `.make_future_dataframe()` method that help us to prepare the data:

```
[ ]:              ds
     1391 2016-10-26
     1392 2016-10-27
     1393 2016-10-28
     1394 2016-10-29
     1395 2016-10-30
```

Now we have acquired a new `future_31` data frame that consist of a date span of **the beginning of a time series to 365 days into the future**. We will then use this data frame is to perform the forecasting by using `.predict()` method of our `model_31`:

```
[ ]:            ds       trend     weekly      yearly        yhat
     0    2013-01-02  376.014905 -32.833816  234.600919  577.782008
     1    2013-01-03  375.942465 -26.061642  215.077487  564.958311
     2    2013-01-04  375.870026  55.637544  193.971274  625.478844
     3    2013-01-05  375.797587  82.002893  171.625948  629.426428
     4    2013-01-06  375.725148  -2.450713  148.403261  521.677695
     …        …           …          …          …           …
     1391 2016-10-26  153.726469 -32.833816  -32.499417   88.393237
     1392 2016-10-27  153.545641 -26.061642  -30.534326   96.949673
     1393 2016-10-28  153.364812  55.637544  -27.645116  181.357241
     1394 2016-10-29  153.183984  82.002893  -23.842381  211.344496
     1395 2016-10-30  153.003156  -2.450713  -19.162378  131.390064

     [1396 rows x 5 columns]
```

Recall that in General Additive Model, we use time series components and perform a summation of all components. In this case, we can see that the model is extracting 3 types of components: `trend`, `weekly` seasonality, and `yearly` seasonality. Means, in forecasting future values it will use the following formula:

$yhat(t) = T(t) + S_{weekly}(t) + S_{yearly}(t)$

We can manually confirm from `forecast_31` that the column `yhat` = `trend` + `weekly` + `yearly`.

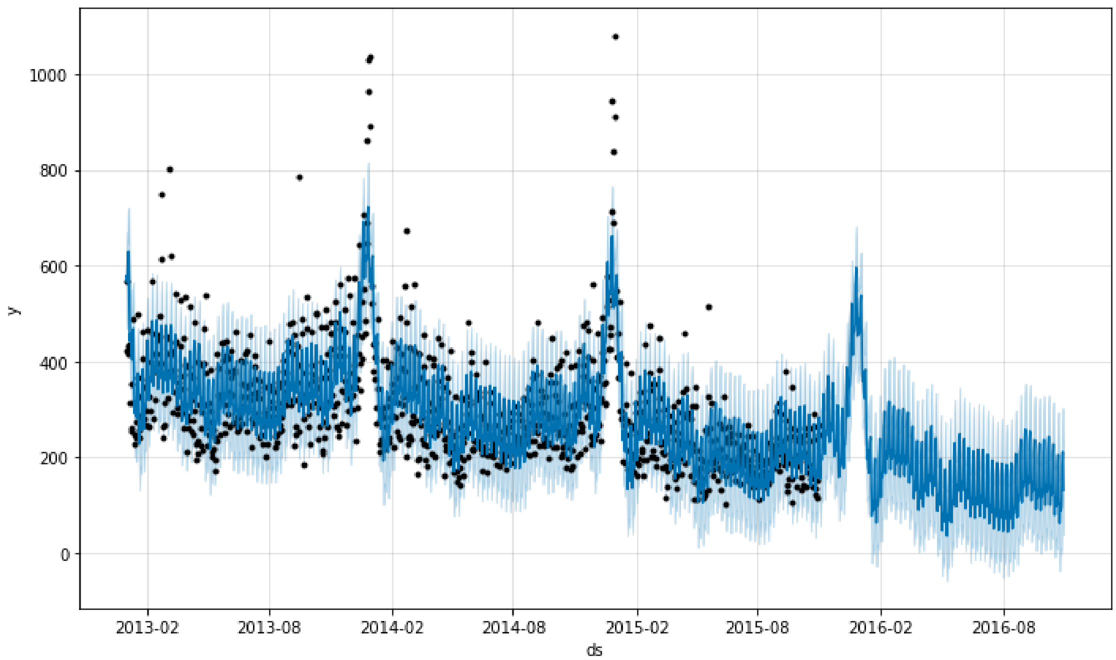
```
[ ]: 0       577.782008
     1       564.958311
     2       625.478844
     3       629.426428
     4       521.677695
                ...
     1391     88.393237
     1392     96.949673
     1393    181.357241
     1394    211.344496
     1395    131.390064
     Length: 1396, dtype: float64
```

```
[ ]: 0       577.782008
     1       564.958311
     2       625.478844
     3       629.426428
     4       521.677695
                ...
     1391     88.393237
     1392     96.949673
     1393    181.357241
     1394    211.344496
     1395    131.390064
```

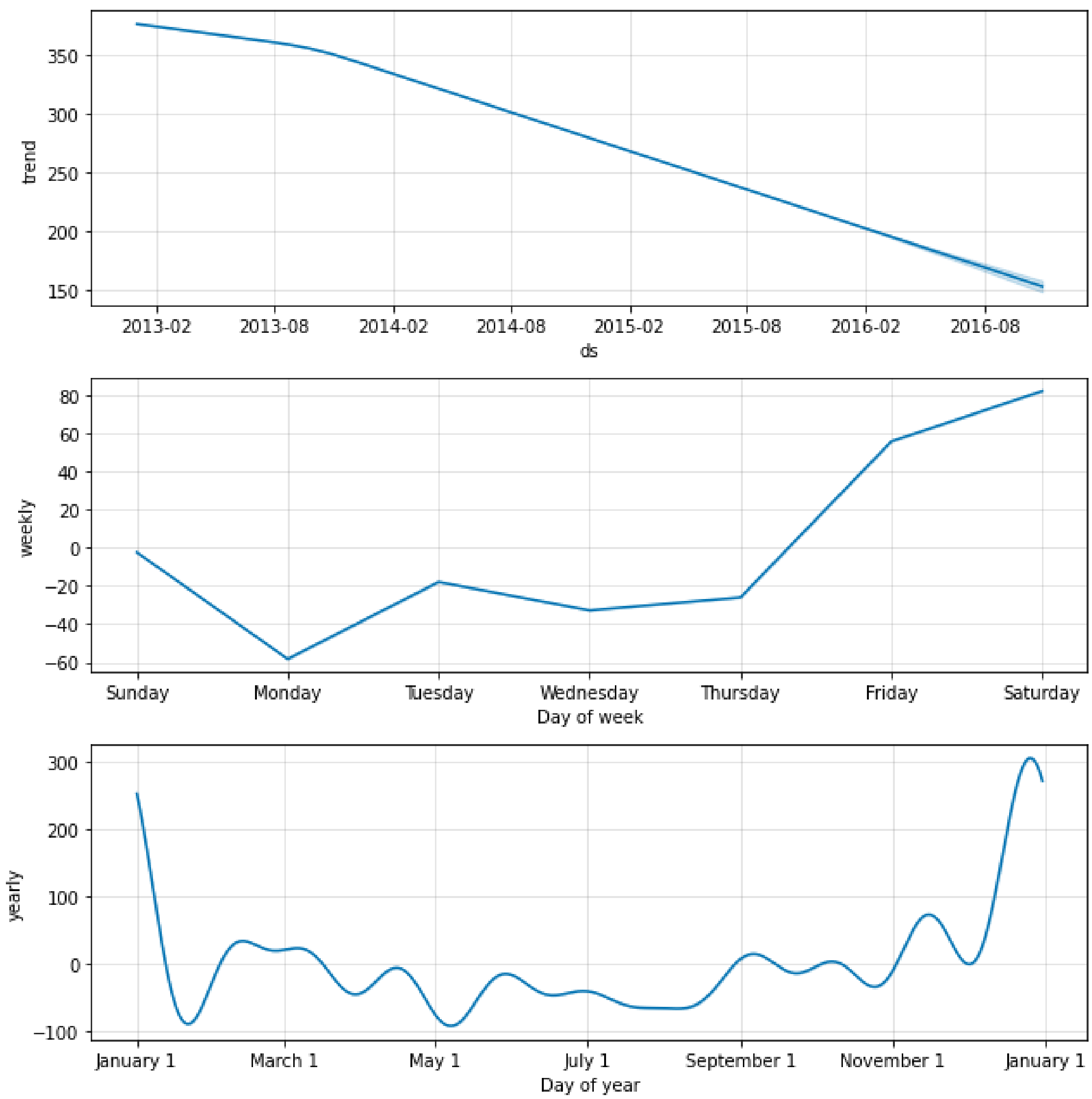
```
Name: yhat, Length: 1396, dtype: float64
```

6.1.4 Visualize

Now, observe how `.plot()` method take our model_31, and newly created forecast_31 object to create a `matplotlib` object that shows the forecasting result. The black points in the plot shows the **actual** time series, and the blue line shows the **fitted** time series along with its forecasted values 365 days into the future.



We can also visualize each of the trend and seasonality components using `.plot_components` method.



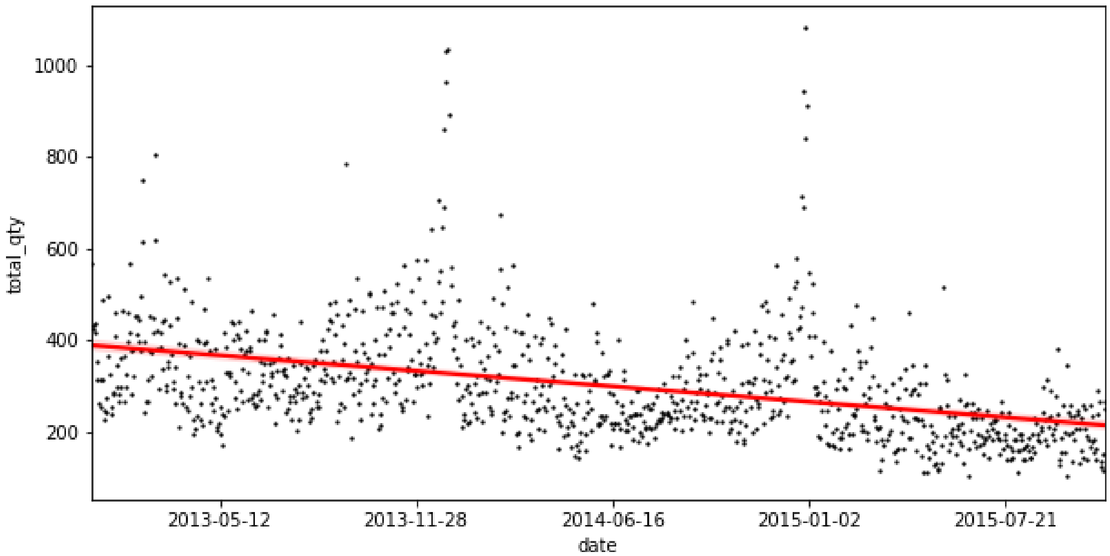
From the visualization above, we can get insights such as:

- The trend shows that the `total_qty` sold is decreasing from time to time.
- The weekly seasonality shows that sales on weekends are higher than weekdays.
- The yearly seasonality shows that sales peaked at the end of the year.

[Optional] Interactive Visualization An interactive figure of the forecast and components can be created with `plotly`. You will need to install `plotly` 4.0 or above separately, as it will not by default be installed with `fbprophet`. You will also need to install the `notebook` and `ipywidgets` packages.

6.2 Trend Component

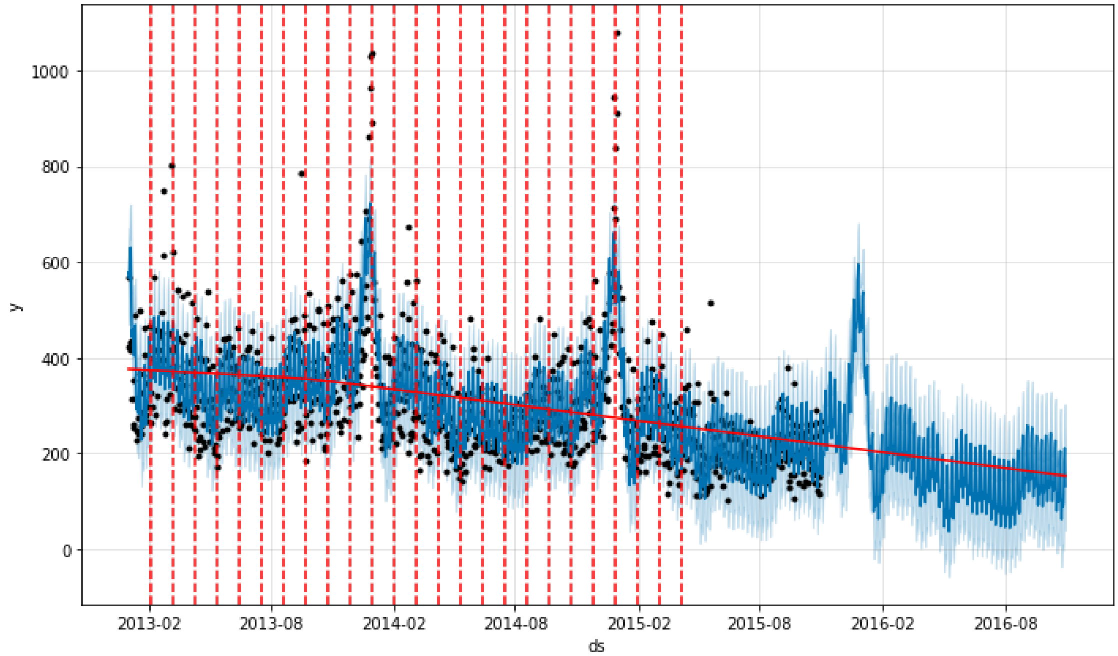
The trend components of our model, as plotted using `.plot_components()` method is producing a decreasing trend over the year. Trend is defined as a long term movement of average over the year. The methods that is implemented by Prophet is by default a **linear model** as shown below:



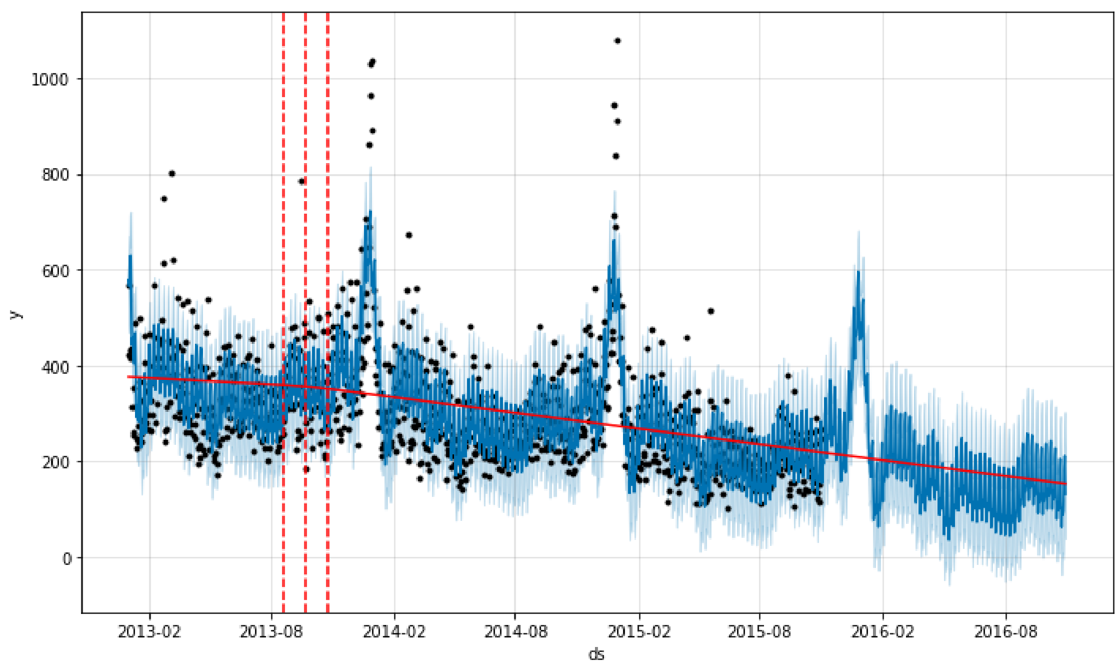
6.2.1 Automatic Changepoint Detection

Prophet however implements a changepoint detection which tries to automatically detect a point where the slope has a significant change rate. It will tries to split the series using several points where the trend slope is calculated for each range.

By default, `prophet` specifies 25 potential changepoints (`n_changepoints=25`) which are placed uniformly on the first 80% of the time series (`changepoint_range=0.8`).



From the 25 potential changepoints, it will then calculate the magnitude of the slope change rate and decided the **significant** change rate. The model detected **3 significant changepoints** and separate the series into **4 different trend slopes**.



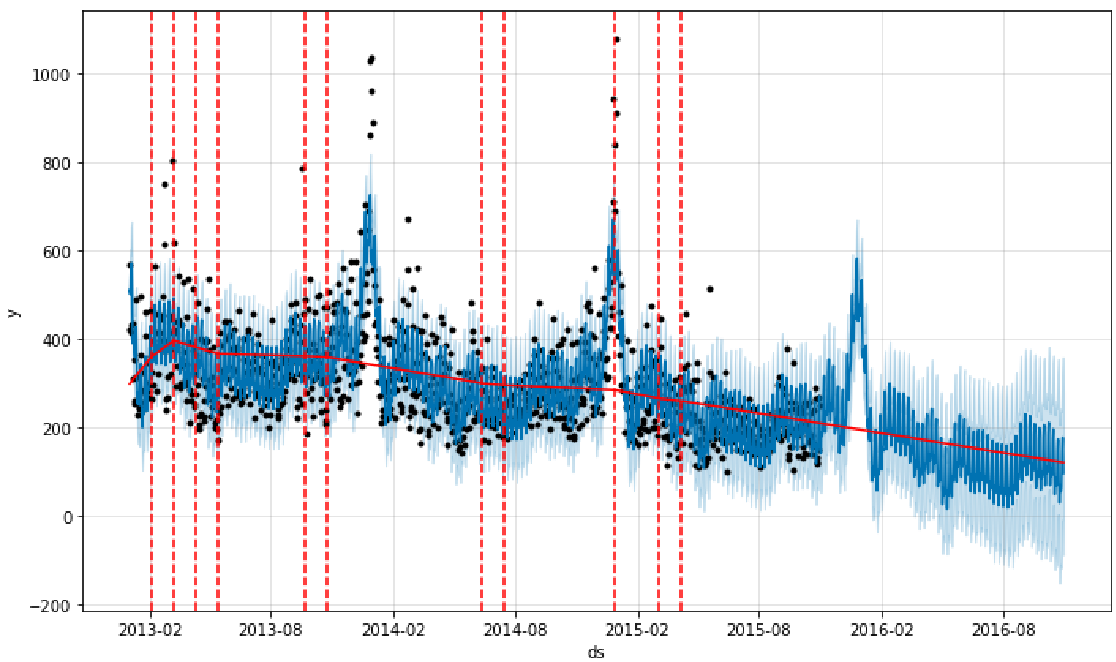
6.2.2 Adjusting Trend Flexibility

Prophet provided us a tuning parameter to adjust the detection flexibility:

- `n_changepoints` (default = 25): The number of potential changepoints, **not recommended** to be tuned, this is better tuned by adjusting the regularization (`changepoint_prior_scale`)
- `changepoint_range` (default = 0.8): Proportion of the history in which the trend is allowed to change. Recommended range: [0.8, 0.95]
- `changepoint_prior_scale` (default = 0.05): The flexibility of the trend, and in particular how much the trend changes at the trend changepoints. Recommended range: [0.001, 0.5]

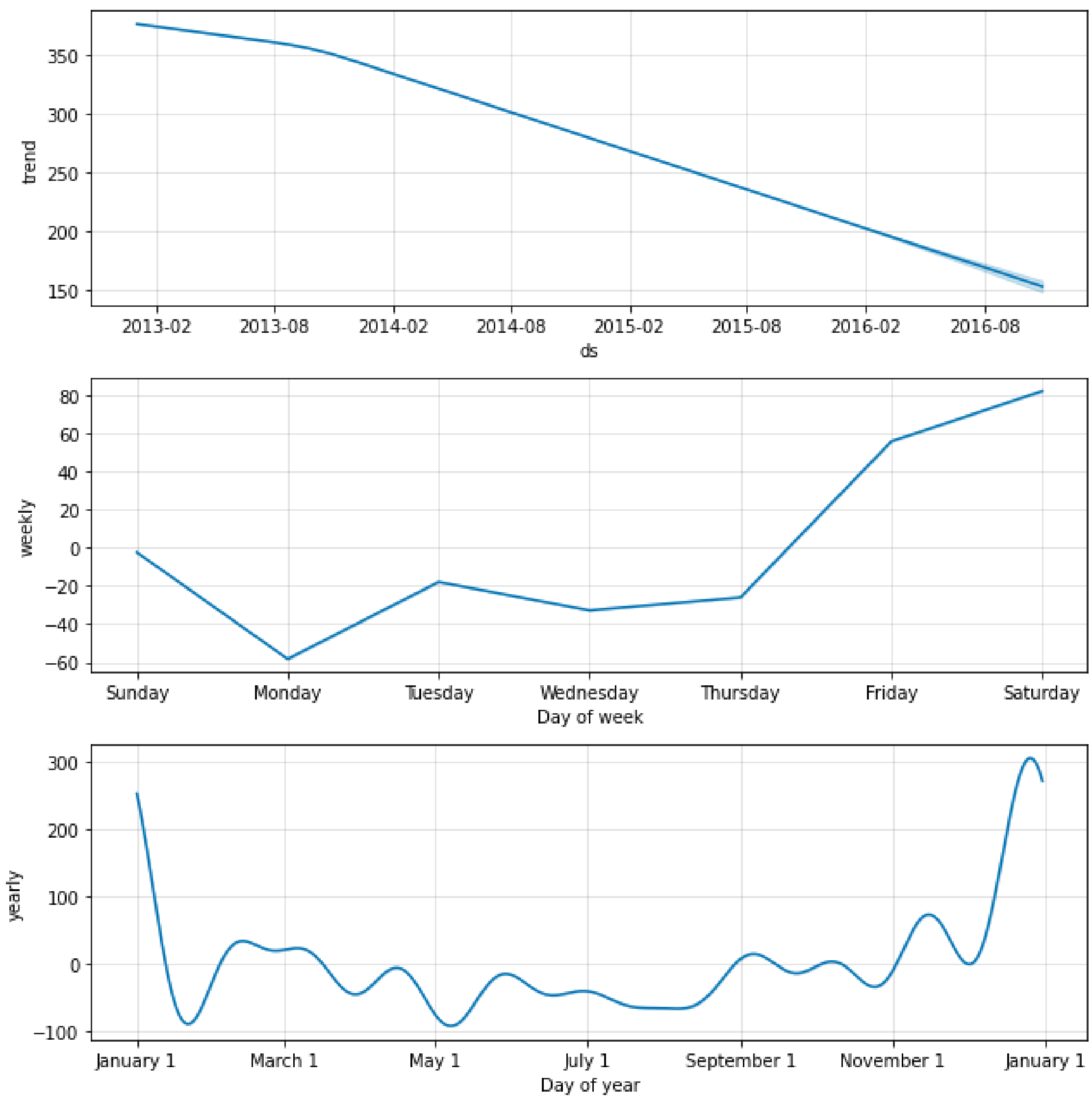
Increasing the default value of the parameter above will give extra flexibility to the trend line (overfitting the training data). On the other hand, decreasing the value will cause the trend to be less flexible (underfitting).

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```



6.3 Seasonality Component

Let's talk about other time series component, seasonality. We will review the following plot components.



By default, Prophet will try to determine existing seasonality based on existing data provided. In our case, the data provided is a **daily** data from early 2013 to end 2015.

- Any daily sampled data by default will be detected to have a **weekly seasonality**.
- While **yearly seasonality**, by default will be set as `True` if the provided data has more than 2 years of daily sample.
- The other regular seasonality is a **daily seasonality** which tries to model an hourly pattern of a time series. Since our data does not accommodate hourly data, by default the daily seasonality will be set as `False`.

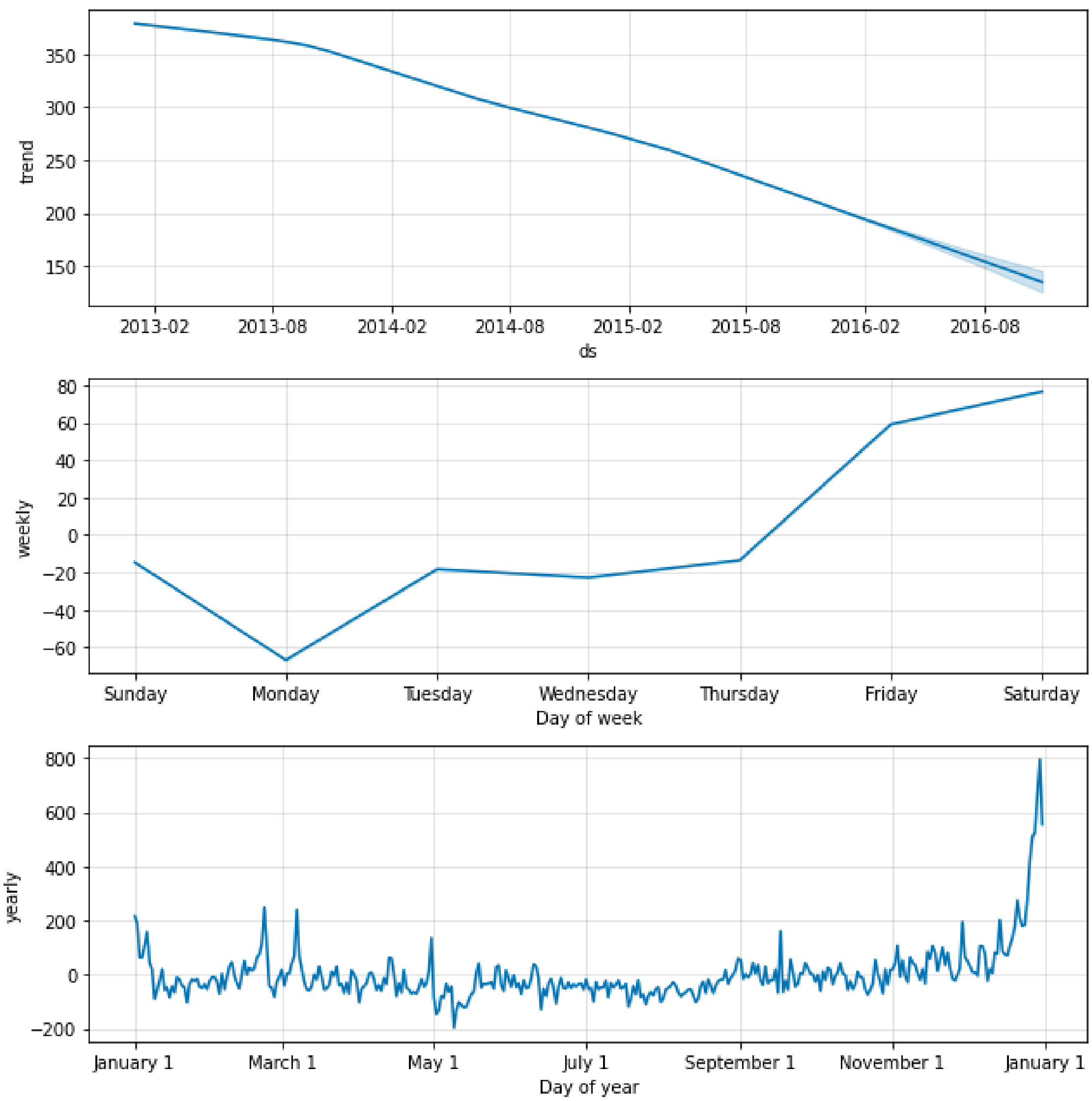
6.3.1 Fourier Order

Prophet uses a Fourier series to approximate the seasonality effect. It is a way of approximating a periodic function as a (possibly infinite) **sum of sine and cosine** functions.

The number of terms in the partial sum (the order) is a parameter that determines how quickly the seasonality can change. Increasing the fourier order will give extra flexibility to the seasonality (overfitting the training data), and vice versa.

Here is an interactive introduction to Fourier: http://www.jezzamon.com/fourier/

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```



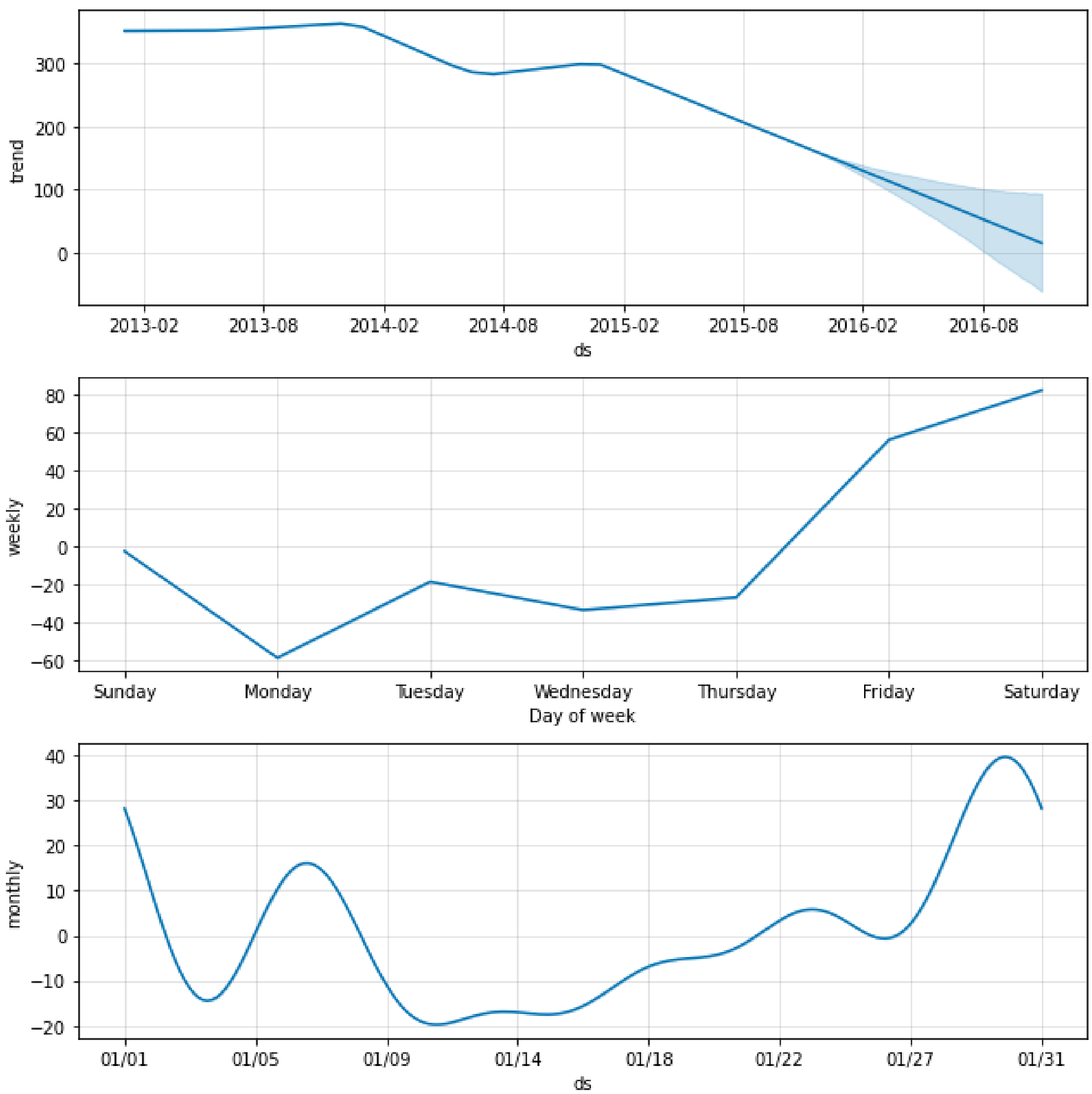
6.3.2 Custom Seasonalities

The default provided seasonality modelled by Prophet for a daily sampled data is: weekly and yearly.

Consider this case: a sales in your business is heavily affected by payday. Most customers tends to buy your product based on the day of the month. Since it did not follow the default seasonality of yearly and weekly, we will need to define a non-regular seasonality. There are two steps we have to do: 1. Remove default seasonality (eg: remove yearly seasonality) by setting `False` 2. Add seasonality (eg: add monthly seasonality) by using `.add_seasonality()` method before fitting the model

We ended up with formula: $yhat(t) = T(t) + S_{weekly}(t) + \mathbf{S_{monthly}(t)}$

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```



For monthly seasonality, we provided period = 30.5 indicating that there will be non-regular 30.5 frequency in one season of the data. The 30.5 is a common frequency quantifier for monthly seasonality, since there are some months with a total of 30 and 31 (some are 28 or 29).

Recommended Fourier order according to the seasonality: - weekly seasonality = 3 - monthly seasonality = 5 - yearly seasonality = 10

6.4 Holiday Effects

One of the advantage in using Prophet is the ability to model a holiday effect. This holiday effect is defined as a non-regular effect that needs to be **manually** specified by the user.

6.4.1 Modeling Holidays and Special Events

Now let's take a better look for our data. We could see that **every end of a year**, there is a significant increase of sales which exceeds 800 sales a day.

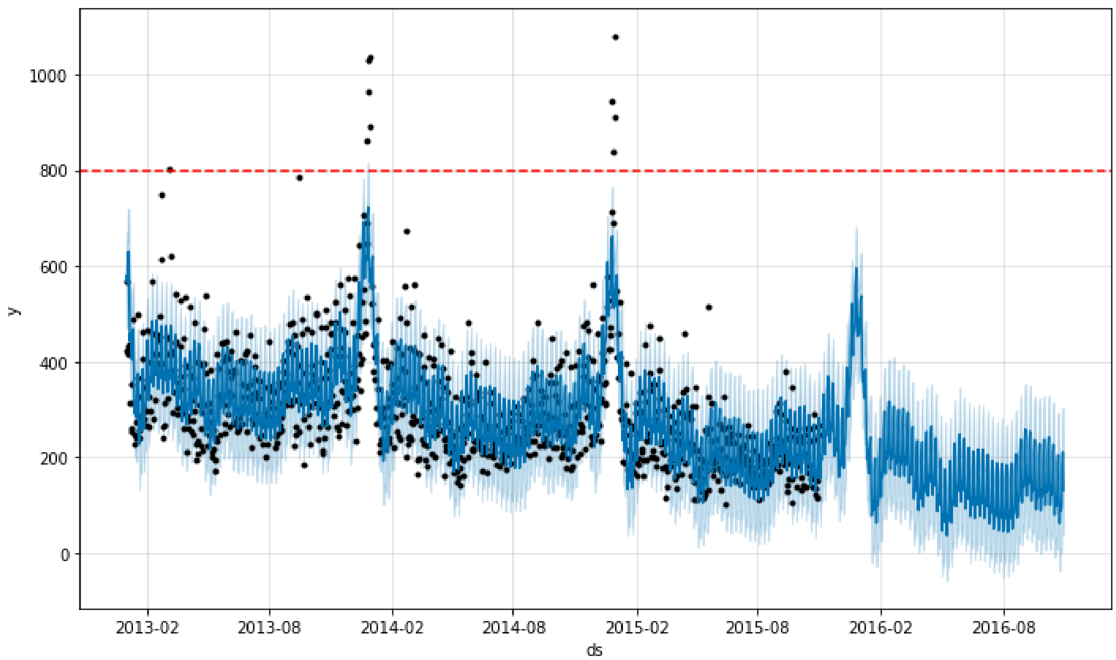


Table below shows that the relatively large sales mostly happened at the very end of a year between **27th to 31st** December. Now let's assume that this phenomenon is the result of the **new year eve** where most people spent the remaining budget of their Christmas or End year bonus to buy our goods.

[ ]:

```
             ds       y
64   2013-03-07   803.0
359  2013-12-27   861.0
360  2013-12-28  1028.0
361  2013-12-29   962.0
362  2013-12-30  1035.0
363  2013-12-31   891.0
723  2014-12-27   942.0
725  2014-12-29   839.0
```

```
726 2014-12-30  1080.0
727 2014-12-31   912.0
```

We'll need to prepare a `holiday` data frame with the following column:

- `holiday`: the holiday unique name identifier
- `ds`: timestamp
- `lower_window`: how many time unit **behind** the holiday that is assumed to to be affected (smaller or equal than zero)
- `upper_window`: how many time unit **after** the holiday that is assumed to be affected (larger or equal to zero)

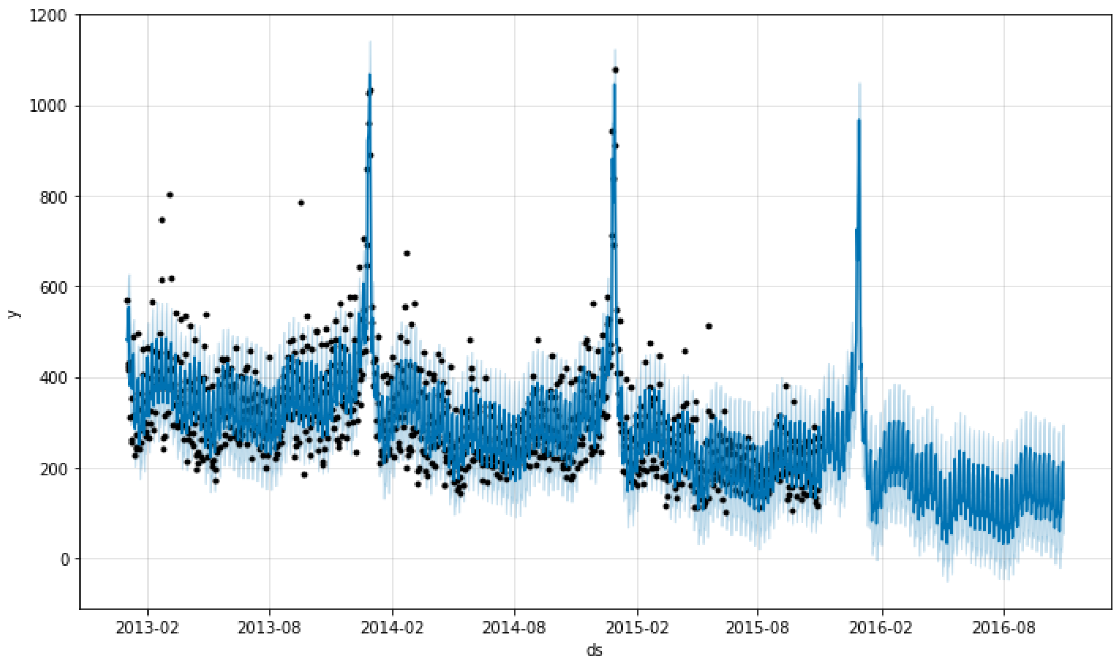
It must include all occurrences of the holiday, both in the **past** (back as far as the historical data go) and in the **future** (out as far as the forecast is being made).

[ ]:

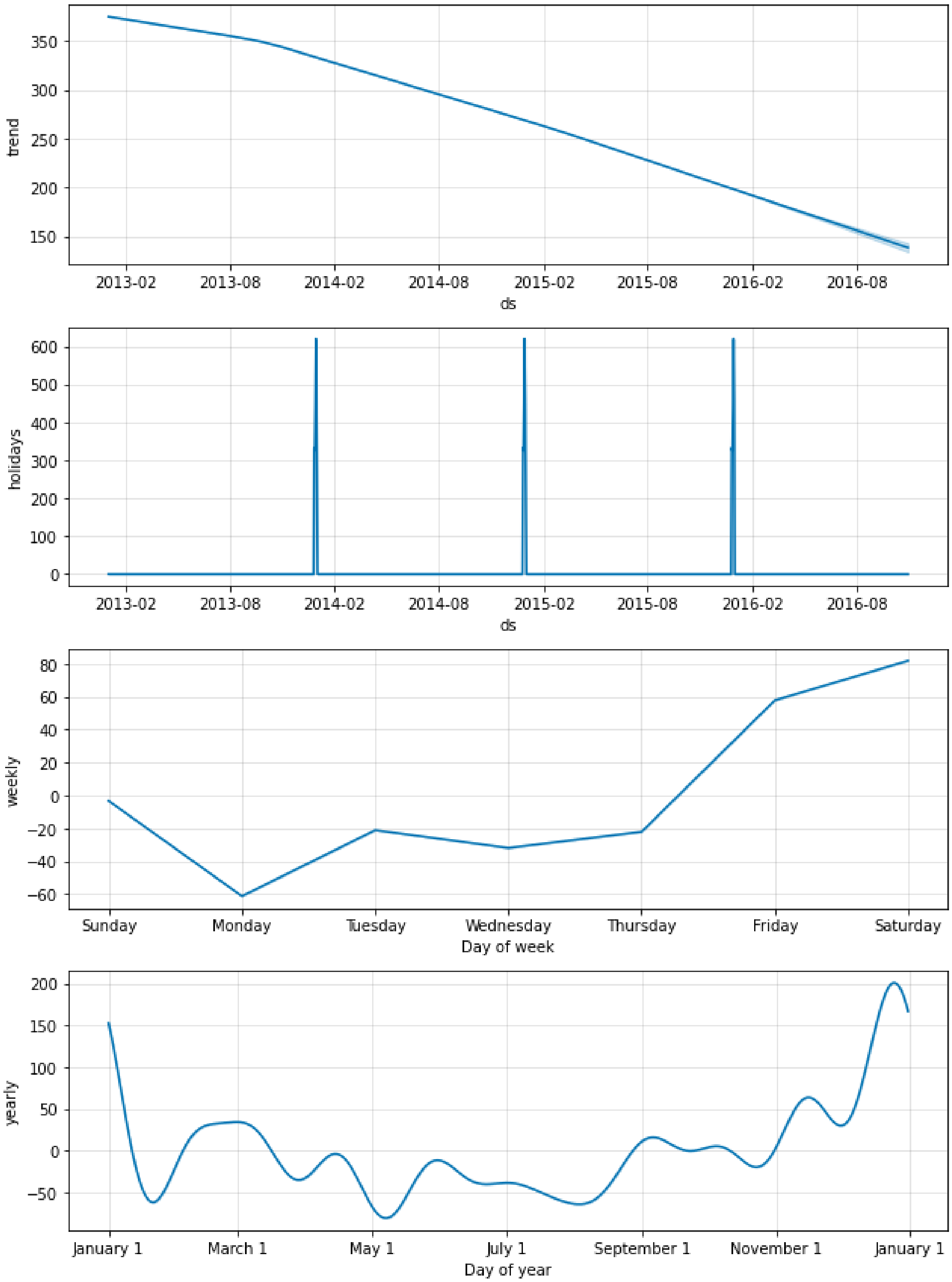
```
       holiday         ds  lower_window  upper_window
0  new_year_eve 2013-12-31            -4             0
1  new_year_eve 2014-12-31            -4             0
2  new_year_eve 2015-12-31            -4             0
```

Once we have prepared our `holiday` data frame, we can pass that into the `Prophet()` class:

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```



Observe how now it has more confidence in capturing the holiday effect on the end of the year instead of relying on the yearly seasonality effect. If we plot the components, we could also get the holiday components listed as one of the time series components:


trend
ds
holidays
ds
weekly
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Day of week
yearly
January 1
March 1
May 1
July 1
September 1
November 1
January 1
Day of year

6.4.2 Built-in Country Holidays

We can use a built-in collection of country-specific holidays using the `.add_country_holidays()` method before fitting model. For Indonesia, we can specify parameter `country_name='ID'`.

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
/usr/local/lib/python3.6/dist-packages/fbprophet/hdays.py:105: Warning:

We only support Nyepi holiday from 2009 to 2019
```

[ ]:

```
0                New Year's Day
1              Chinese New Year
2          Day of Silence/ Nyepi
3        Ascension of the Prophet
4                     Labor Day
5             Ascension of Jesus
6              Buddha's Birthday
7                   Eid al-Fitr
8              Independence Day
9          Feast of the Sacrifice
10             Islamic New Year
11                    Christmas
12         Birth of the Prophet
dtype: object
```

We can also manually populate Indonesia holiday by using `hdays` module. This is useful if we want to take a look on the holiday dates and then manually include only certain holidays.

```
/usr/local/lib/python3.6/dist-packages/fbprophet/hdays.py:105: Warning:

We only support Nyepi holiday from 2009 to 2019
```

[ ]:

	ds	holiday
0	2020-01-01	New Year's Day
1	2020-01-25	Chinese New Year
2	2020-03-22	Ascension of the Prophet
3	2020-05-01	Labor Day
4	2020-05-21	Ascension of Jesus
5	2020-05-07	Buddha's Birthday
6	2020-06-01	Pancasila Day
7	2020-05-25	Eid al-Fitr
8	2020-08-17	Independence Day
9	2020-08-20	Islamic New Year
10	2020-10-29	Birth of the Prophet
11	2020-12-25	Christmas
12	2021-01-01	New Year's Day

```
13  2021-02-12           Chinese New Year
14  2021-03-11  Ascension of the Prophet
15  2021-05-01                 Labor Day
16  2021-05-13        Ascension of Jesus
17  2021-05-26         Buddha's Birthday
18  2021-06-01             Pancasila Day
19  2021-05-14               Eid al-Fitr
20  2021-08-17          Independence Day
21  2021-07-20    Feast of the Sacrifice
22  2021-08-10          Islamic New Year
23  2021-10-19      Birth of the Prophet
24  2021-12-25                 Christmas
```

6.5 Adding Regressors

Additional regressors can be added to the linear part of the model using the `.add_regressor()` method, before fitting model. In this case, we want to forecast `total_revenue` based on its previous revenue components (trend, seasonality, holiday) and also `total_qty` sold as the regressor:

$$revenue(t) = T_{revenue}(t) + S_{revenue}(t) + H_{revenue}(t) + \mathbf{qty(t)}$$

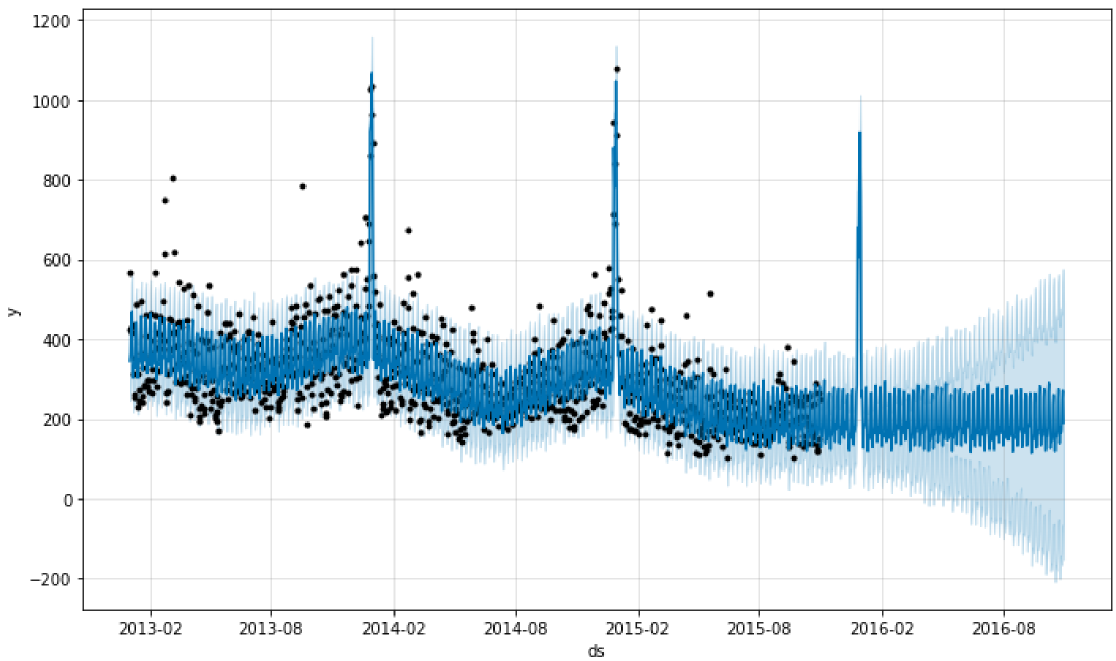
The extra regressor must be known for both the **history** and for **future** dates. It thus must either be something that has known future values or something that has separately been forecasted with a time series model, such as Prophet. A note of **caution** around this approach: error in the forecast of regressor will produce error in the forecast of target value.

[ ]:
```
        date  shop_id  total_qty  total_revenue
0 2013-01-02       31      568.0      396376.10
1 2013-01-03       31      423.0      276933.11
2 2013-01-04       31      431.0      286408.00
3 2013-01-05       31      415.0      273245.00
4 2013-01-06       31      435.0      260775.00
```

6.5.1 Forecast the Regressor (`total_qty`)

In this section, we separately create a Prophet model to forecast `total_qty`, before we forecast `total_revenue`.

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```



The table below shows the forecasted total quantity for the last 365 days (from November 1st, 2015 until October 30th, 2016).

[ ]:

	ds	total_qty
1031	2015-11-01	193.255929
1032	2015-11-02	128.734150
1033	2015-11-03	164.798473
1034	2015-11-04	158.194736
1035	2015-11-05	178.743638
…	…	…
1391	2016-10-26	165.831597
1392	2016-10-27	171.156393
1393	2016-10-28	247.616489
1394	2016-10-29	271.470601
1395	2016-10-30	189.416426

[365 rows x 2 columns]

On the other hand, the table below shows the actual total quantity which we used for training model. We have to rename the column exactly like the previous table.

[ ]:

	ds	total_qty
0	2013-01-02	568.0
1	2013-01-03	423.0
2	2013-01-04	431.0
3	2013-01-05	415.0

```
4    2013-01-06      435.0
…         …        …
1026 2015-10-27      123.0
1027 2015-10-28      117.0
1028 2015-10-29      152.0
1029 2015-10-30      267.0
1030 2015-10-31      249.0

[1031 rows x 2 columns]
```

Now, we have to prepare concatenated data of `total_qty` as the regressor values of `total_revenue`:

- First 1031 observations: actual values of `total_qty`
- Last 365 observations: forecasted values of `total_qty`

[ ]:
```
             ds   total_qty
0    2013-01-02  568.000000
1    2013-01-03  423.000000
2    2013-01-04  431.000000
3    2013-01-05  415.000000
4    2013-01-06  435.000000
…         …         …
1391 2016-10-26  165.831597
1392 2016-10-27  171.156393
1393 2016-10-28  247.616489
1394 2016-10-29  271.470601
1395 2016-10-30  189.416426

[1396 rows x 2 columns]
```

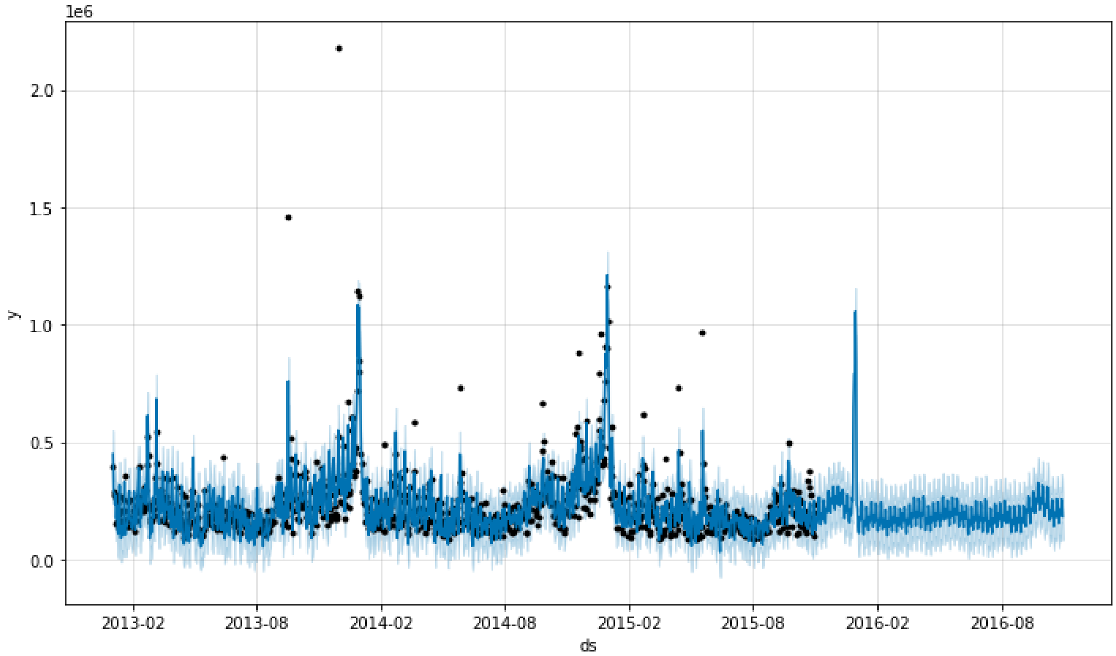
6.5.2 Forecast the Target Variable (`total_revenue`)

Next, we create a Prophet model to forecast `total_revenue`, using `total_qty` as the regressor. Make sure to rename the date as `ds` and the value to be forecasted as `y`.

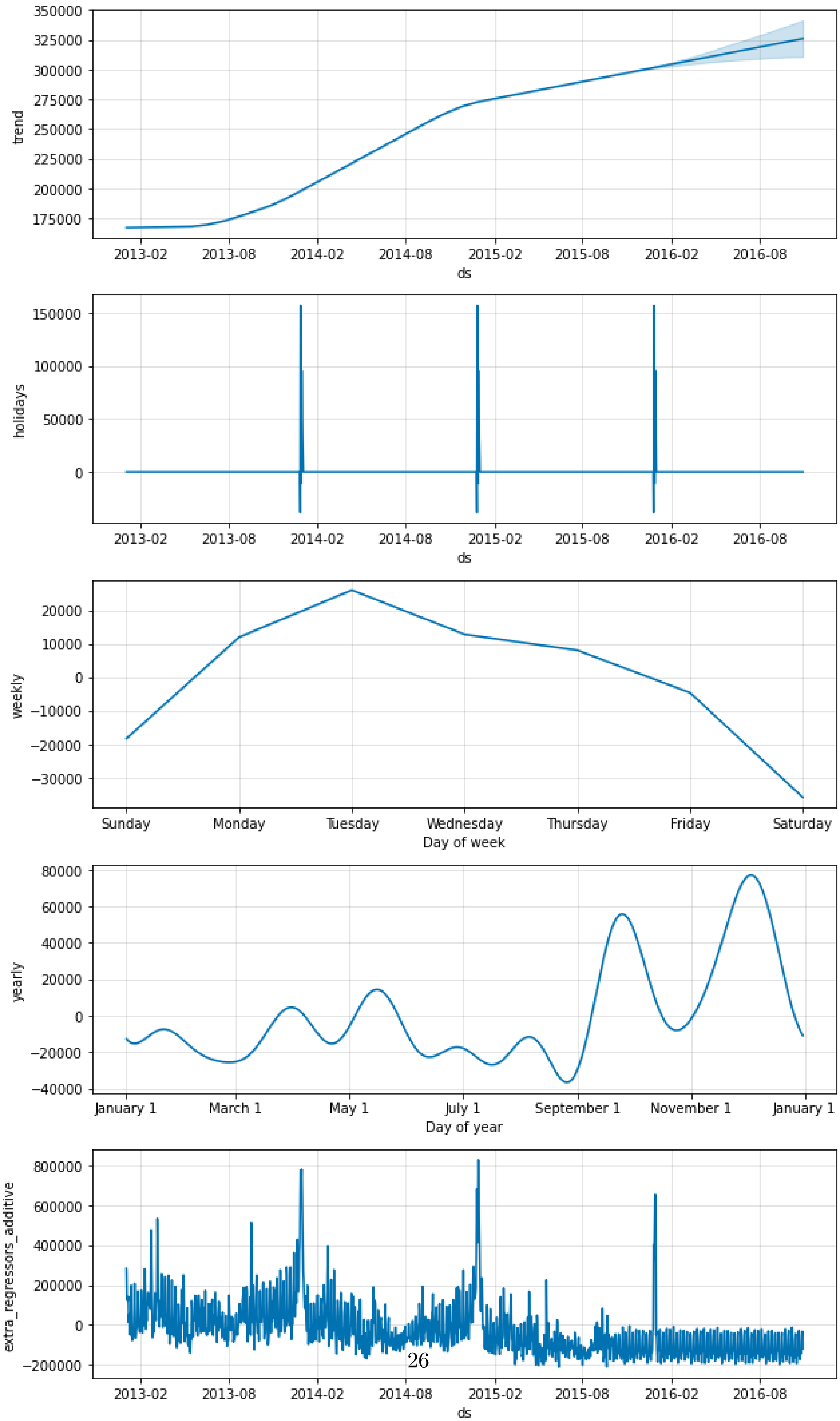
[ ]:
```
          ds          y  total_qty
0 2013-01-02  396376.10      568.0
1 2013-01-03  276933.11      423.0
2 2013-01-04  286408.00      431.0
3 2013-01-05  273245.00      415.0
4 2013-01-06  260775.00      435.0
```

During fitting a model with regressor, make sure: - Apply `.add_regressor()` method before fitting - Forecast the value using `future_with_regressor` data frame that we have prepared before, containing `ds` and the regressor values

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```



If we plot the components, we could also get the extra regressors components listed as one of the time series components:


trend
ds
holidays
ds
weekly
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Day of week
yearly
January 1
March 1
May 1
July 1
September 1
November 1
January 1
Day of year
extra_regressors_additive
ds

By adding regressors, we lose the ability to interpret the other components (trend, seasonality, holiday) due to the fluctuation of the extra regressor value.

7 Forecasting Evaluation

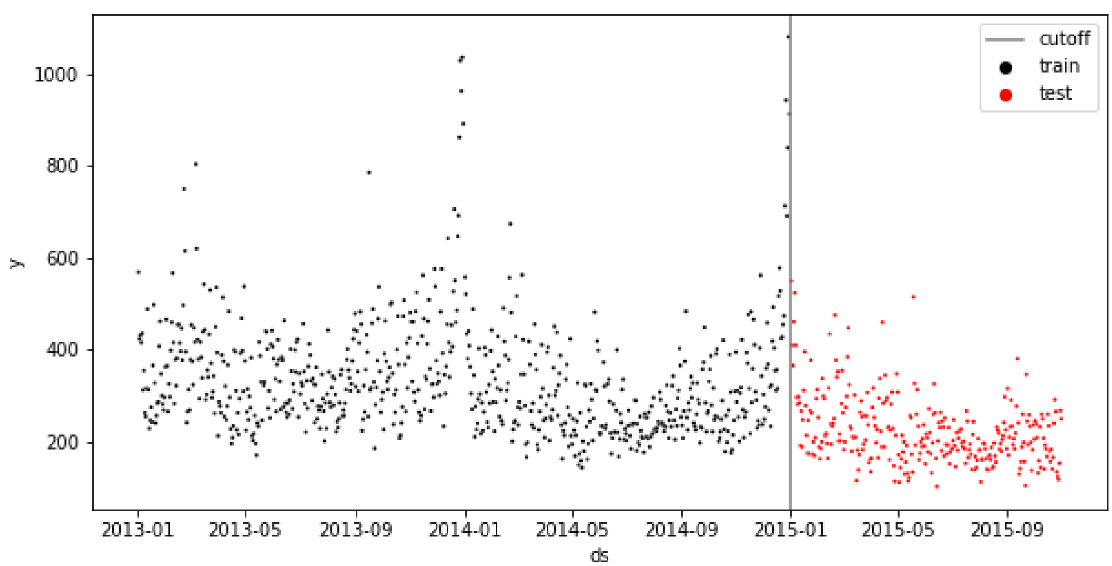
Recall how we performed a visual analysis on how the performance of our forecasting model earlier. The technique was in fact, a widely used technique for model cross-validation. It involves splitting our data into two parts:

- Train data is used to train our time series model in order to acquire the underlying patterns such as trend and seasonality.
- Test data is purposely being kept for us to perform a cross-validation and see how our model perform on an **unseen data**.

The objective is quite clear, is that we are able to acquire a glimpse of what kind of error are we going to expect for the model.

7.1 Train-Test Split

Recall that our data has the range of early 2013 to end 2015. Say, we are going to save the records of 2015 as a test data and use the rest for model training. The points in red will now be treated as unseen data and will not be passed in to our Prophet model.

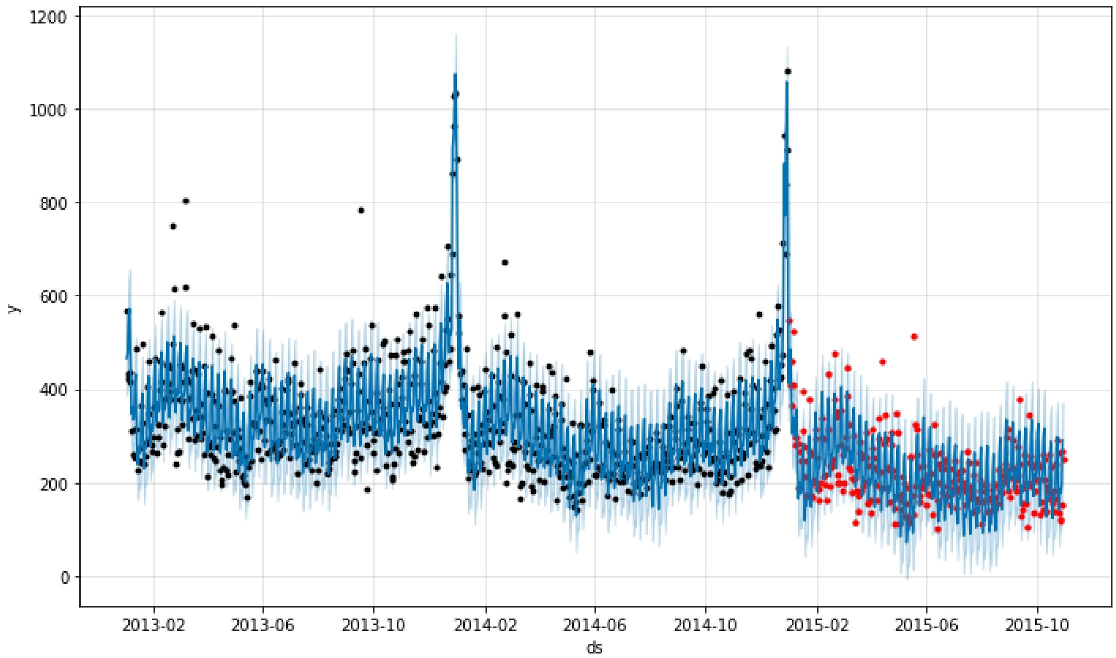


We can split at a cutoff using conditional subsetting as below:

```
Train size: (728, 3)
Test size: (303, 3)
```

Now let's train the model using data from 2013-2014 only, and forecast **303 days** into the future (until October 31st, 2015).

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```



7.2 Evaluation Metrics

Based on the plot above, we can see that the model is capable in forecasting the actual future value. But for most part, we will need to quantify the error to be able to have a conclusive result. To quantify an error, we need to calculate the **difference between actual demand and the forecasted demand**. However, there are several metrics we can use to express the value. Some of them are:

- Root Mean Squared Error

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(p_i - a_i)^2}$$

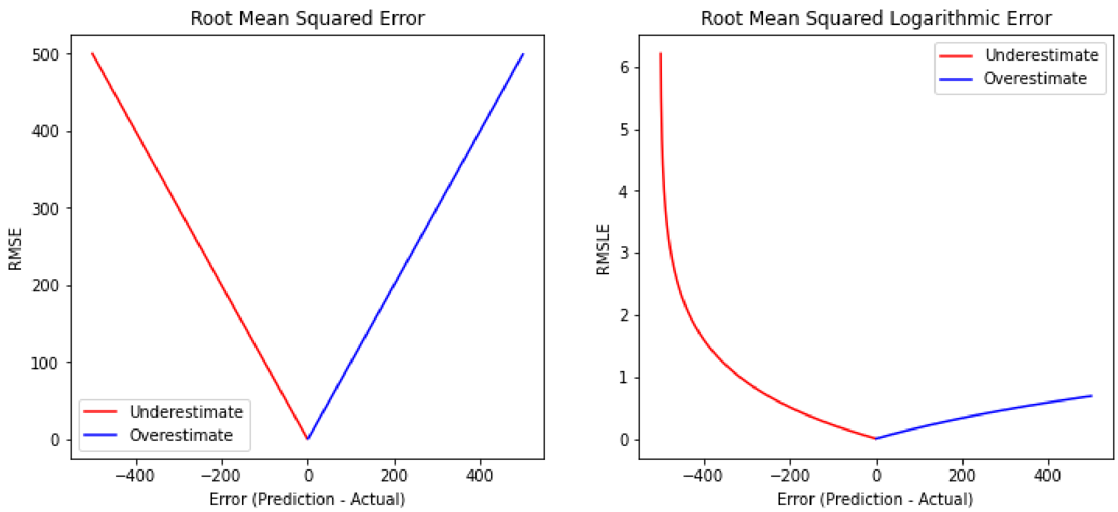
- Root Mean Squared Logarithmic Error

$$RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(log(p_i + 1) - log(a_i + 1))^2}$$

Notation:

- n: length of time series
- p_i: predicted value at time i

- a_i: actual value at time i



The main reason RMSLE is prefered over RMSE: It incurs a larger penalty for the underestimation of the actual value than the overestimation. This is useful for business cases where the underestimation of the target variable is not acceptable but overestimation can be tolerated.

```
[ ]: 0.18125248249390458
```

```
[ ]: 0.36602550027367353
```

Any of the metrics can be used to benchmark a forecasting model, as long as we are consistent in using it. Other regression metrics can be seen on Scikit Learn Documentation

7.3 Expanding Window Cross Validation

Instead of only doing one time train-test split, we can do cross validation as shown below:

This cross validation procedure is called as **expanding window** and can be done automatically by using the `cross_validation()` method. There are three parameters to be specified:

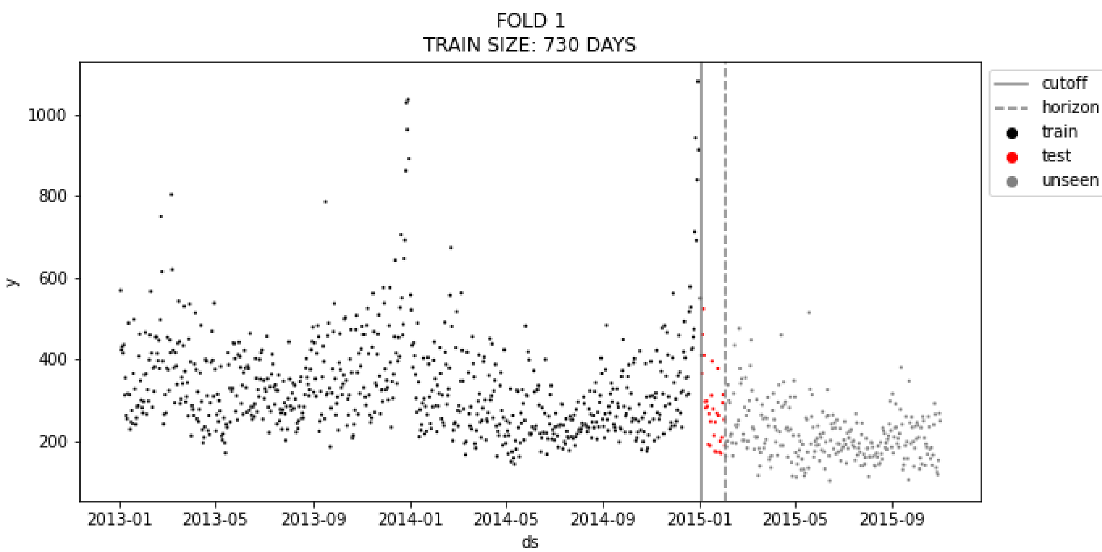
- `initial`: the length of the initial training period
- `horizon`: forecast length
- `period`: spacing between cutoff dates

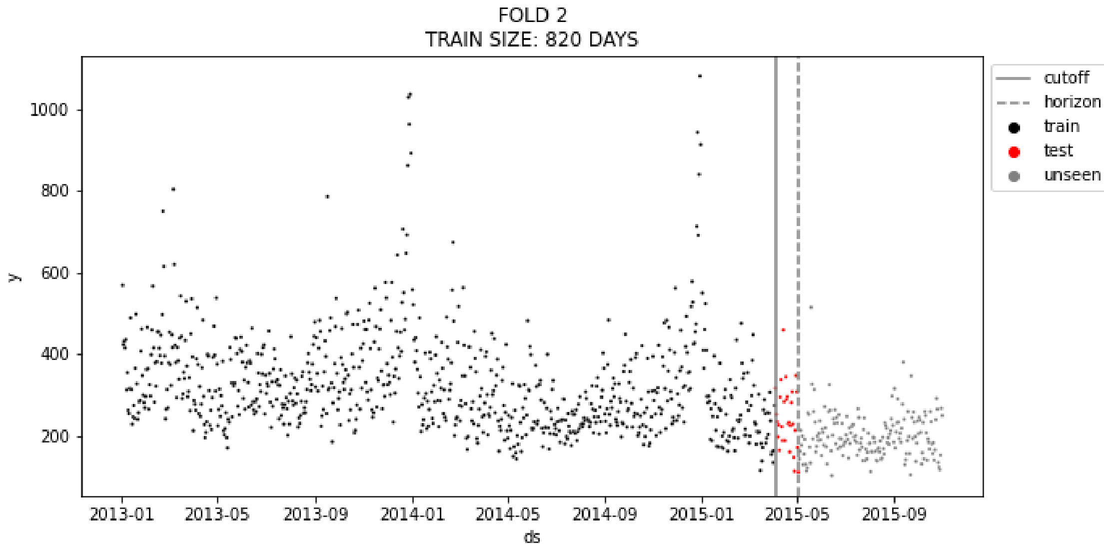
```
INFO:fbprophet:Making 4 forecasts with cutoffs between 2015-01-04 00:00:00 and
2015-10-01 00:00:00
```

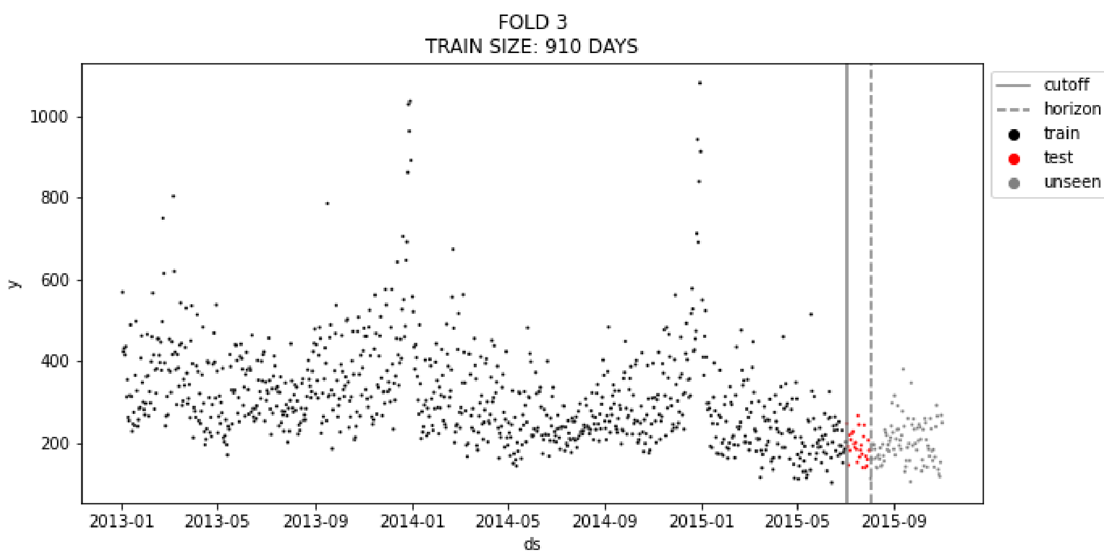
```
HBox(children=(FloatProgress(value=0.0, max=4.0), HTML(value='')))
```

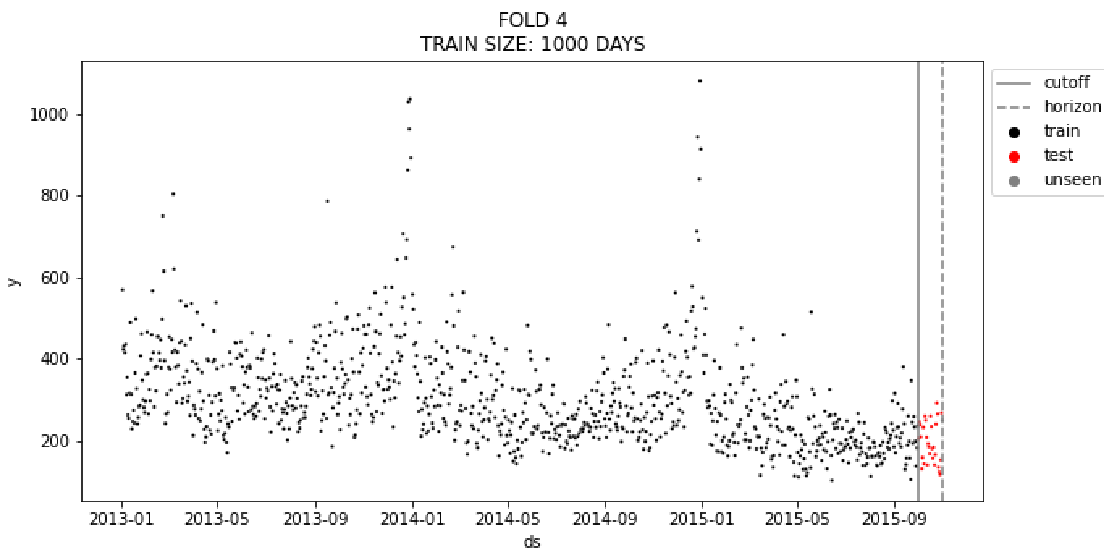
```
[ ]:            ds           yhat  …         y     cutoff
0    2015-01-05  453243.686249  …  357996.0 2015-01-04
1    2015-01-06  528839.864534  …  562368.0 2015-01-04
2    2015-01-07  400103.526338  …  290563.0 2015-01-04
3    2015-01-08  282273.692327  …  285423.0 2015-01-04
4    2015-01-09  255507.073608  …  232971.0 2015-01-04
..          …              …  …         …          …
115  2015-10-27  111576.408050  …  111851.0 2015-10-01
116  2015-10-28   91895.680477  …  180557.0 2015-10-01
117  2015-10-29  126869.389853  …  103456.0 2015-10-01
118  2015-10-30  237405.154426  …  204317.0 2015-10-01
119  2015-10-31  187605.724288  …  237587.0 2015-10-01

[120 rows x 6 columns]
```

The cross validation process above will be carried out for 4 folds, where at each fold a forecast will be made for the next 30 days (`horizon`) from the cutoff dates. Below is the illustration for each fold:




FOLD 2
TRAIN SIZE: 820 DAYS
cutoff
horizon
train
test
unseen
y
ds
1000
800
600
400
200
2013-01
2013-05
2013-09
2014-01
2014-05
2014-09
2015-01
2015-05
2015-09


FOLD 3
TRAIN SIZE: 910 DAYS
cutoff
horizon
train
test
unseen
y
ds
1000
800
600
400
200
2013-01
2013-05
2013-09
2014-01
2014-05
2014-09
2015-01
2015-05
2015-09



Cross validation error metrics can be evaluated for each folds, here shown for RMSLE.

[ ]:
```
cutoff
2015-01-04    0.235920
2015-04-04    0.304658
2015-07-03    0.236829
2015-10-01    0.308821
dtype: float64
```

We can aggregate the metrics by using its mean. In other words, we are calculating the **mean of RMSLE** to represent the overall model performance.

[ ]:
```
0.27155703183152
```

8 Hyperparameter Tuning

In this section, we implement a **Grid search algorithm** for model tuning by using for-loop. It builds model for every combination from specified hyperparameters and then evaluate it. The goal is to choose a set of optimal hyperparameters which minimize the forecast error (in this case, smallest RMSLE).

You can use the code template below, please change it as needed in the section marked by `TO DO`.

Click here for a list of recommended hyperparameters to be tuned.

```
  0%|          | 0/6 [00:00<?, ?it/s]INFO:fbprophet:Disabling daily seasonality.
Run prophet with daily_seasonality=True to override this.
INFO:fbprophet:Making 3 forecasts with cutoffs between 2015-03-05 00:00:00 and
2015-09-01 00:00:00
INFO:fbprophet:Applying in parallel with
```

```
<concurrent.futures.process.ProcessPoolExecutor object at 0x7faa211b2080>
 17%|         | 1/6 [00:05<00:29,  5.92s/it]INFO:fbprophet:Disabling daily
seasonality. Run prophet with daily_seasonality=True to override this.
INFO:fbprophet:Making 3 forecasts with cutoffs between 2015-03-05 00:00:00 and
2015-09-01 00:00:00
INFO:fbprophet:Applying in parallel with
<concurrent.futures.process.ProcessPoolExecutor object at 0x7faa211dc2b0>
 33%|        | 2/6 [00:11<00:23,  5.91s/it]INFO:fbprophet:Disabling daily
seasonality. Run prophet with daily_seasonality=True to override this.
INFO:fbprophet:Making 3 forecasts with cutoffs between 2015-03-05 00:00:00 and
2015-09-01 00:00:00
INFO:fbprophet:Applying in parallel with
<concurrent.futures.process.ProcessPoolExecutor object at 0x7faa24b2e668>
 50%|       | 3/6 [00:18<00:18,  6.08s/it]INFO:fbprophet:Disabling daily
seasonality. Run prophet with daily_seasonality=True to override this.
INFO:fbprophet:Making 3 forecasts with cutoffs between 2015-03-05 00:00:00 and
2015-09-01 00:00:00
INFO:fbprophet:Applying in parallel with
<concurrent.futures.process.ProcessPoolExecutor object at 0x7faa232bd0b8>
 67%|      | 4/6 [00:24<00:12,  6.15s/it]INFO:fbprophet:Disabling daily
seasonality. Run prophet with daily_seasonality=True to override this.
INFO:fbprophet:Making 3 forecasts with cutoffs between 2015-03-05 00:00:00 and
2015-09-01 00:00:00
INFO:fbprophet:Applying in parallel with
<concurrent.futures.process.ProcessPoolExecutor object at 0x7faa22e01518>
 83%|     | 5/6 [00:30<00:06,  6.16s/it]INFO:fbprophet:Disabling daily
seasonality. Run prophet with daily_seasonality=True to override this.
INFO:fbprophet:Making 3 forecasts with cutoffs between 2015-03-05 00:00:00 and
2015-09-01 00:00:00
INFO:fbprophet:Applying in parallel with
<concurrent.futures.process.ProcessPoolExecutor object at 0x7faa2387c5f8>
100%|     | 6/6 [00:37<00:00,  6.19s/it]
```

We can observe the error metrics for each hyperparameter combination, and sort by ascending:

[ ]:

	changepoint_prior_scale	changepoint_range	rmsle
5	0.100	0.95	0.264595
4	0.100	0.80	0.264672
3	0.010	0.95	0.265335
2	0.010	0.80	0.266934
0	0.001	0.80	0.268179
1	0.001	0.95	0.275176

Best hyperparameter combination can be extracted as follows:

[ ]:
```
{'changepoint_prior_scale': 0.1, 'changepoint_range': 0.95}
```

Lastly, re-fit the model and use it for forecasting.

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```

```
[ ]: <fbprophet.forecaster.Prophet at 0x7faa232473c8>
```

Note: `**` is an operator for dictionary unpacking. It delivers key-value pairs in a dictionary into a function's arguments.

9 [Optional] Error Diagnostics

Prophet has provide us several frequently used evaluation metrics by using `performance_metrics()`:

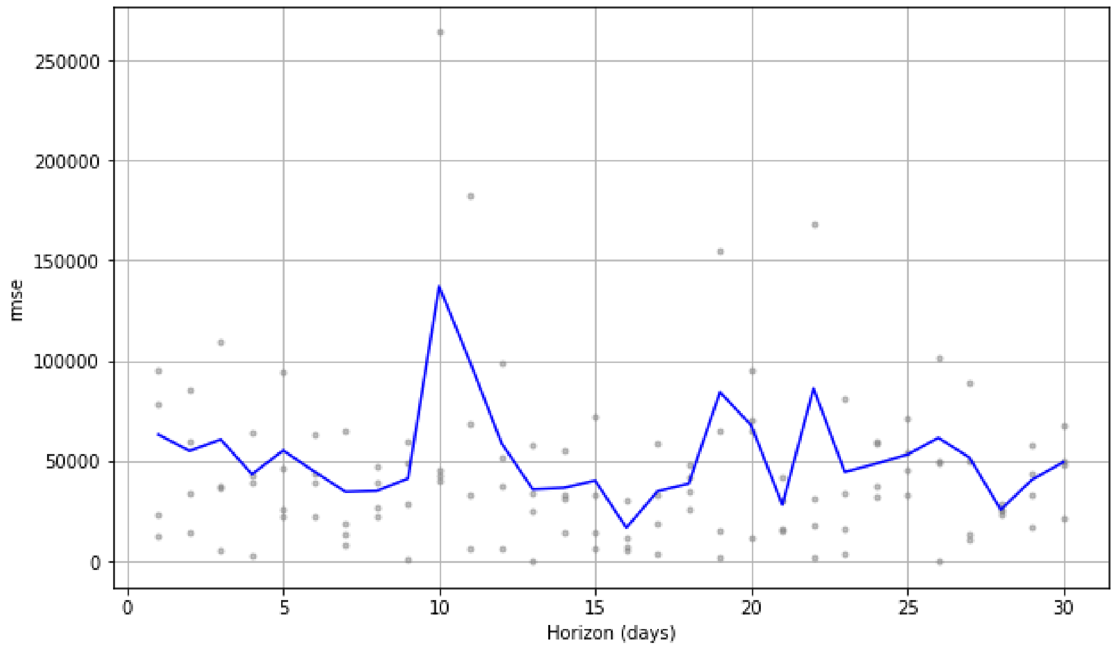
- Mean Squared Error (MSE)
- Root Mean Squared Error (RMSE)
- Mean Absolute Error (MAE)
- Mean Absolute Percentage Error (MAPE)
- Median Absolute Percentage Error (MDAPE)
- Coverage: Percentage of actual data that falls on the forecasted uncertainty (confidence) interval

```
[ ]:    horizon           mse           rmse  …      mape     mdape  coverage
    0    1 days  3.994022e+09   63198.275661  …  0.219069  0.193591      1.00
    1    2 days  3.029427e+09   55040.228983  …  0.278188  0.238097      1.00
    2    3 days  3.688972e+09   60736.911716  …  0.206292  0.210830      0.75
    3    4 days  1.881936e+09   43381.285624  …  0.264149  0.215639      1.00
    4    5 days  3.054242e+09   55265.193585  …  0.383068  0.302298      1.00
    5    6 days  2.000743e+09   44729.665907  …  0.197795  0.207520      1.00
    6    7 days  1.199535e+09   34634.305123  …  0.133289  0.092500      1.00
    7    8 days  1.232793e+09   35111.156633  …  0.203406  0.175338      1.00
    8    9 days  1.699657e+09   41226.902300  …  0.198048  0.199360      1.00
    9   10 days  1.878038e+10  137041.527486  …  0.330543  0.329390      0.75
    10  11 days  9.801675e+09   99003.409264  …  0.299810  0.282302      0.75
    11  12 days  3.463804e+09   58854.091268  …  0.208275  0.212405      1.00
    12  13 days  1.287332e+09   35879.414831  …  0.146204  0.179784      1.00
    13  14 days  1.342285e+09   36637.201070  …  0.168220  0.178241      1.00
    14  15 days  1.617733e+09   40221.049884  …  0.195245  0.107827      1.00
    15  16 days  2.784726e+08   16687.496976  …  0.077429  0.056516      1.00
    16  17 days  1.219819e+09   34925.908299  …  0.151663  0.122949      1.00
    17  18 days  1.498776e+09   38714.027600  …  0.310942  0.304227      1.00
    18  19 days  7.113056e+09   84338.934975  …  0.254228  0.274203      0.75
    19  20 days  4.598471e+09   67812.029787  …  0.328830  0.314520      1.00
    20  21 days  7.914749e+08   28133.163420  …  0.144335  0.149402      1.00
    21  22 days  7.431953e+09   86208.774809  …  0.188154  0.145252      0.75
    22  23 days  1.982702e+09   44527.538187  …  0.163817  0.174394      1.00
    23  24 days  2.368130e+09   48663.432588  …  0.378250  0.356780      1.00
    24  25 days  2.822479e+09   53127.012300  …  0.379622  0.411288      1.00
    25  26 days  3.791062e+09   61571.598733  …  0.269882  0.274819      1.00
```

```
26 27 days  2.667743e+09   51650.199465  …  0.198813  0.127268      1.00
27 28 days  6.598579e+08   25687.698689  …  0.207142  0.215036      1.00
28 29 days  1.654698e+09   40677.980863  …  0.276104  0.250000      1.00
29 30 days  2.441526e+09   49411.803495  …  0.405376  0.345830      1.00

[30 rows x 7 columns]
```

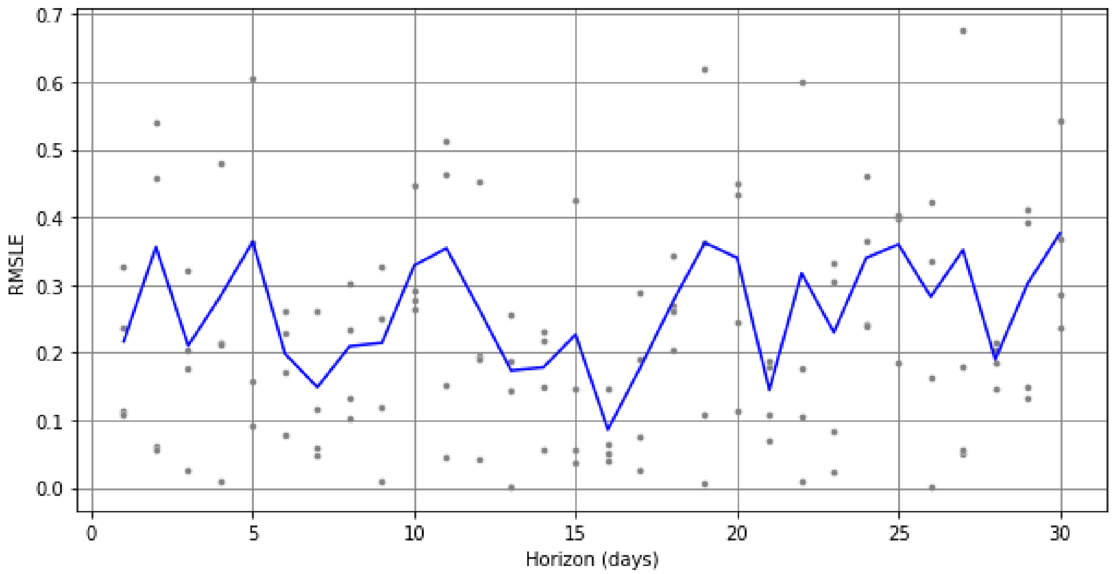
Cross validation performance metrics can be visualized with `plot_cross_validation_metric`, here shown for RMSE.

- Dots show the root squared error (RSE) for each prediction in `df_cv`.
- The blue line shows the RMSE for each horizon.



Unfortunately, Prophet has not implement RMSLE metric in their library. Therefore, we have to calculate it manually according to its mathematical formula, or simply use `sklearn.metrics` module.

- Dots show the root squared logarithmic error (RSLE) for each prediction in `df_cv`.
- The blue line shows the RMSLE for each horizon.



10 References

Prophet related:

- Prophet Documentation
- Paper: Forecasting at Scale
- Algoritma: Time Series Forecasting using `prophet` in R

Further reading (for R):

- Textbook Forecasting: Principles and Practice
- Algotech: Multiple Seasonality Time Series
- Algotech: Time Series LSTM (Neural Network)
- Algotech: Multiple Time Series Model