Egor Polusmak Follow

Head of Data Analytics at Mail.Ru Search

Apr 14 · 16 min read

Open Machine Learning Course. Topic 9. Part 2. Predicting the future with Facebook Prophet

Time series forecasting finds wide application in data analytics. These are only some of the conceivable predictions of future trends that might be useful:

- The number of servers that an online service will need next year;

- The demand for a grocery product at a supermarket on a given day;

- The tomorrow closing price of a tradable financial asset.

For another example, we can make a prediction of some team's performance and then use it as a baseline: first to set goals for the team, and then to measure the actual team performance relative to the baseline.

There are quite a few different methods to predict future trends, for example, ARIMA, ARCH, regressive models, neural networks.

In this article, we will look at Prophet, a library for time series forecasting released by Facebook and open-sourced on February 23, 2017. We will also try it out in the problem of predicting the daily number of posts published on Medium.

Article outline

1. Introduction

2. The Prophet Forecasting Model

3. Practice with Prophet

- 3.1 Installation in Python

- 3.2 Dataset

- 3.3 Exploratory visual analysis

- 3.4 Making a forecast

- 3.5 Forecast quality evaluation

- 3.6 Visualization

4. Box-Cox Transformation

5. Summary

6. References

The following material is better viewed as a Jupyter notebook and can be reproduced locally with Jupyter if you clone the course repository.

1. Introduction

According to the article by Facebook Research, Prophet was initially developed for the purpose of creating high quality business forecasts. This library tries to address the following difficulties common to many business time series:

- Seasonal effects caused by human behavior: weekly, monthly and yearly cycles, dips and peaks on public holidays;

- Changes in trend due to new products and market events;

- Outliers.

The authors claim that, even with default settings, in many cases, their library produces forecasts as accurate as those delivered by experienced analysts.

Moreover, Prophet has a number of intuitive and easily interpretable customizations that allow gradually improving the quality of the forecasting model. What is especially important, these parameters are quite comprehensible even for non-experts in time series analysis, which is a field of data science requiring certain skill and experience.

By the way, the original article is called "Forecasting at Scale", but it is not about the scale in the "usual" sense, that it's addressing computational and infrastructure problems of a large number of working programs. According to the authors, Prophet should scale well in the following 3 areas:

- Accessibility to a wide audience of analysts, possibly without profound expertise in time series;

- Applicability to a wide range of distinct forecasting problems;

- Automated performance estimation of a large number of forecasts including flagging of potential problems for their subsequent inspection by the analyst.

2. The Prophet Forecasting Model

Now, let's take a closer look at how Prophet works. In its essence, this library utilizes the additive regression model $y(t)$ comprising the following components:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t,$$

where:

- Trend $g(t)$ models non-periodic changes;

- Seasonality $s(t)$ represents periodic changes;

- Holidays component *h(t)* contributes information about holidays and events.

Below, we will consider some important properties of these model components.

Trend

The Prophet library implements two possible trend models for *g(t)*.

The first one is called *Nonlinear, Saturating Growth*. It is represented in the form of the <u>logistic growth model</u>:

$$g(t) = \frac{C}{1 + e^{-k(t-m)}},$$

where:

- *C* is the carrying capacity (that is the curve's maximum value).

- *k* is the growth rate (which represents "the steepness" of the curve).

- *m* is an offset parameter.

This logistic equation allows modelling non-linear growth with saturation, that is when the growth rate of a value decreases with its growth. One of the typical examples would be representing the growth of the audience of an application or a website.

Actually, CC and kk are not necessarily constants and may vary over time. Prophet supports both automatic and manual tuning of their variability. The library can itself choose optimal points of trend changes by fitting the supplied historical data.

Also, Prophet allows analysts to manually set changepoints of the growth rate and capacity values at different points in time. For instance, analysts may have insights about dates of past releases that prominently influenced some key product indicators.

The second trend model is a simple *Piecewise Linear Model* with a constant rate of growth. It is best suited for problems without saturating growth.

Seasonality

The seasonal component $s(t)$ provides a flexible model of periodic changes due to weekly and yearly seasonality.

Weekly seasonal data is modeled with dummy variables. Six new variables are added: `monday` , `tuesday` , `wednesday` , `thursday` , `friday` , `saturday` , which take values 0 or 1 depending on the day of the week. The feature `sunday` is not added because it would be a linear combination of the other days of the week, and this fact would have an adverse effect on the model.

Yearly seasonality model in Prophet relies on Fourier series.

Since version 0.2 you can also use *sub-daily time series* and make *sub-daily forecasts* as well as employ the new *daily seasonality* feature.

Holidays and Events

The component $h(t)$ represents predictable abnormal days of the year including those on irregular schedules, e.g., Black Fridays.

To utilize this feature, the analyst needs to provide a custom list of events.
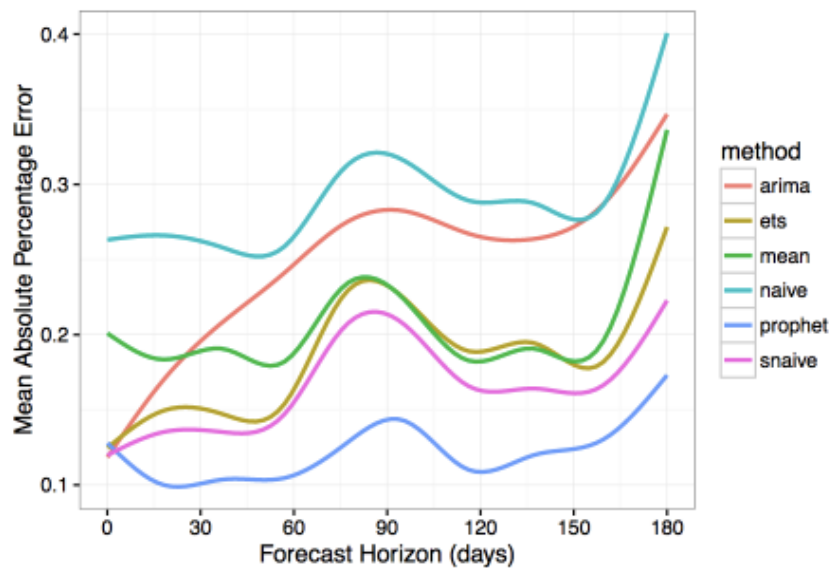
Error

The error term $\epsilon(t)$ represents information that was not reflected in the model. Usually it is modeled as normally distributed noise.

Prophet Benchmarking

For a detailed description of the model and algorithms behind Prophet refer to the paper "Forecasting at scale" by Sean J. Taylor and Benjamin Letham.

The authors also compared their library with several other methods for time series forecasting. They used Mean Absolute Percentage Error (MAPE) as a measure of prediction accuracy. In this research, Prophet has shown substantially lower forecasting error than the other models.

Let's look closer at how the forcasting quality was measured in the article. To do this, we will need the formula of Mean Absolute Percentage Error.

Let y_i be the *actual (historical) value* and $\hat{y}_i$ be the *forecast value* given by our model.

Then $e_i = y_i - \hat{y}_i$ is the *forecast error* and $p_i = e_i/y_i$ is the *relative forecast error*.

We define

$$MAPE = mean\left(\ |p_i|\ \right)$$

MAPE is widely used as a measure of prediction accuracy because it expresses error as a percentage and thus can be used in model evaluations on different datasets.

In addition, when evaluating a forecasting algorithm, it may prove useful to calculate MAE (Mean Absolute Error) in order to have a picture of errors in absolute numbers. Using previously defined components, its equation will be

$$MAE = mean\left(\,|e_i|\,\right)$$

A few words about the algorithms that Prophet was compared with. Most of them are quite simple and often are used as a baseline for other models:

- `naive` is a simplistic forecasting approach where we predict all future values relying solely on the observation at the last available point of time.

- `snaive` (seasonal naive) is a model that makes constant predictions taking into account information about seasonality. For instance, in the case of weekly seasonal data for each future Monday, we would predict the value from the last Monday, and for all future Tuesdays we would use the value from the last Tuesday and so on.

- `mean` uses the averaged value of data as a forecast.

- `arima` stands for *Autoregressive Integrated Moving Average*, see Wikipedia for details.

- `ets` stands for *Exponential Smoothing*, see Wikipedia for more.

3. Practice with Facebook Prophet

3.1 Installation in Python

First, you need to install the library. Prophet is available for Python and R. The choice will depend on your personal preferences and project requirements. Further in this article we will use Python.

In Python you can install Prophet using PyPI:

```
$ pip install fbprophet
```

In R you can find the corresponding CRAN package. Refer to the documentation for details.

Let's import the modules that we will need, and initialize our environment:

```
1    import warnings
2    warnings.filterwarnings('ignore')
3
4    import numpy as np
5    import pandas as pd
6    from scipy import stats
7    import statsmodels.api as sm
```

3.2 Dataset

We will predict the daily number of posts published on Medium.

First, we load our dataset:

```
df = pd.read_csv('../../data/medium_posts.csv', sep='\t')
```

Next, we leave out all columns except `published` and `url` . The former corresonds to the time dimension while the latter uniquely identifies a post by its URL. Along the way we get rid of possible duplicates and missing values in the data:

```
df = df[['published', 'url']].dropna().drop_duplicates()
```

Next, we need to convert `published` to the datetime format because by default `pandas` treats this field as string-valued.

```
df['published'] = pd.to_datetime(df['published'])
```

Let's sort the dataframe by time and take a look at what we've got:

```
df.sort_values(by=['published']).head(n=3)
```

	published	url
50931	1970-01-01 00:00:00.001	https://medium.com/iiot
40243	1970-01-01 00:00:00.001	https://medium.com/@ikaella/melon-rebranding-b...
37395	1970-01-18 05:11:46.500	http://www.novosti.rs/%D0%B2%D0%B5%D1%81%D1%82...

Medium's public release date was August 15, 2012. But, as you can see from the data above, there are at least several rows with much earlier publication dates. They have somehow turned up in our dataset, but they are hardly legitimate ones. We will just trim our time series to keep only those rows that fall onto the period from August 15, 2012 to June 25, 2017:

```
df = df[(df['published'] > '2012-08-15') &
        (df['published'] < '2017-06-26')].
          sort_values(by=['published'])
df.head(n=3)
```

	published	url
24630	2012-08-15 00:25:03.373	https://medium.com/launch-day/jean-attempts-to...
24631	2012-08-15 00:25:29.419	https://medium.com/launch-day/dan-and-kristin-...
17811	2012-08-15 00:34:59.502	https://medium.com/i-m-h-o/the-world-is-social...

```
df.tail(n=3)
```

	published	url
62122	2017-06-25 23:36:01.171	https://medium.com/push-the-pace/the-official-...
72471	2017-06-25 23:41:48.295	https://medium.com/parti-xyz-developers/%EA%B4...
83283	2017-06-25 23:51:43.000	http://www.johanr.com/blog/people-support-dreams

As we are going to predict the number of published posts, we will aggregate and count unique posts at each given point in time. We will name the corresponding new column `posts` :

```
aggr_df = df.groupby('published')[['url']].count()
aggr_df.columns = ['posts']
```

In this practice, we are interested in the number of posts **a day**. But at this moment all our data is divided into irregular time intervals that are less than a day. This is called a *sub-daily time series*. To see it, let's print out the first 3 rows:

```
aggr_df.head(n=3)
```

published	posts
2012-08-15 00:25:03.373	1
2012-08-15 00:25:29.419	1
2012-08-15 00:34:59.502	1

To fix this, we need to aggregate the post counts by "bins" of a date size. In time series analysis, this process is referred to as *resampling*. And if we *reduce* the sampling rate of data it is often called *downsampling*.

Luckily, `pandas` has a built-in functionality for this task. We will resample our time index down to 1-day bins:

```
daily_df = aggr_df.resample('D').apply(sum)
daily_df.head(n=3)
```

	posts
published	
2012-08-15	16.0
2012-08-16	11.0
2012-08-17	4.0

3.3 Exploratory visual analysis

As always, it may be helpful and instructive to look at a graphical representation of your data.

We will create a time series plot for the whole time range. Displaying data over such a long period of time can give clues about seasonality and conspicuous abnormal deviations.

First, we import and initialize the `Plotly` library, which allows creating beautiful interactive plots:

```
1   from plotly.offline import init_notebook_mode, iplot
2   from plotly import graph_objs as go
3
4   # Initialize plotly
```

We also define a helper function, which will plot our dataframes throughout the article:

```
1   def plotly_df(df, title=''):
2       """Visualize all the dataframe columns as line plot
3       common_kw = dict(x=df.index, mode='lines')
4       data = [go.Scatter(y=df[c], name=c, **common_kw) fo
5       layout = dict(title=title)
6       fig = dict(data=data, layout=layout)
```

Let's try and plot our dataset *as is*:

```
plotly_df(daily_df, title='Posts on Medium (daily)')
```

Posts on Medium (daily)
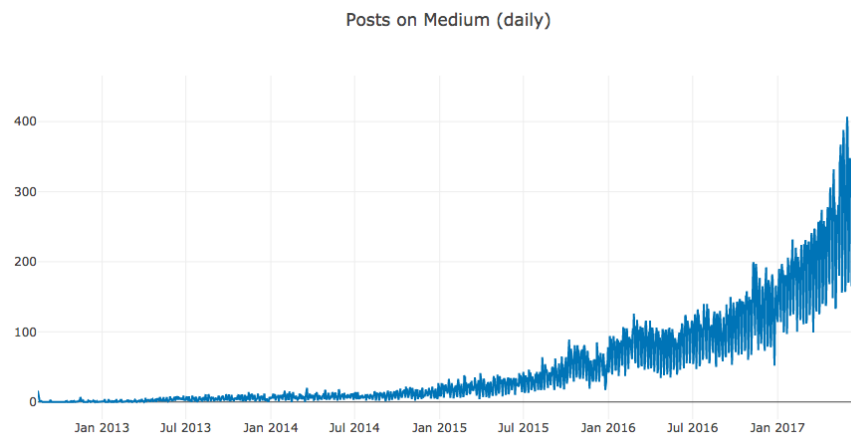
High-frequency data can be rather difficult to analyze. Even with the ability to zoom in provided by `Plotly` , it is hard to infer anything meaningful from this chart apart from the prominent upward and accelerating trend.

To reduce the noise, we will resample the post counts down to weekly bins. Besides *binning*, other possible techniques of noise reduction include Moving-Average Smoothing and Exponential Smoothing, among others.

We save our downsampled dataframe in a separate variable because further in this practice we will work only with daily series:

```
weekly_df = daily_df.resample('W').apply(sum)
```

Finally, we plot the result:

```
plotly_df(weekly_df, title='Posts on Medium (weekly)')
```

Posts on Medium (weekly)
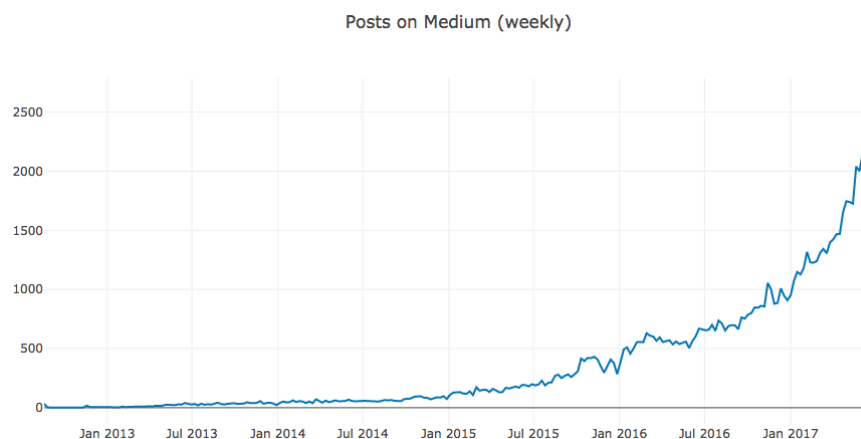
This downsampled chart proves to be somewhat better for an analyst's perception.

One of the most useful functions that `Plotly` provides is the ability to quickly dive into different periods of timeline in order to better understand the data and find visual clues about possbile trends, periodic and irregular effects.

For example, zooming-in on a couple of consecutive years shows us time points corresponding to Christmas holidays, which greatly influence human behaviors.

Now, we're going to omit the first few years of observations, up to 2015. First, they won't contribute much into the forecast quality in 2017. Second, these first years, having very low number of posts per day, are likely to increase noise in our predictions, as the model would be forced to fit this abnormal historical data along with more relevant and indicative data from the recent years.

```
daily_df = daily_df.loc[daily_df.index >= '2015-01-01']
daily_df.head(n=3)
```

	posts
published	
2015-01-01	8.0
2015-01-02	11.0
2015-01-03	11.0

To sum up, from visual analysis we can see that our dataset is non-stationary with a prominent growing trend. It also demonstrates weekly and yearly seasonality and a number of abnormal days in each year.

3.4 Making a forecast

Prophet's API is very similar to the one you can find in `sklearn`. First we create a model, then call the method `fit`, and, finally, make a forecast. The input to the method `fit` is a `DataFrame` with two columns:

- `ds` (datestamp) must be of type `date` or `datetime`.

- `y` is a numeric value we want to predict.

To get started, we'll import the library and mute unimportant diagnostic messages:

```
from fbprophet import Prophet
import logging

logging.getLogger().setLevel(logging.ERROR)
```

Let's convert our dataframe to the format required by Prophet:

```
df = daily_df.reset_index()
df.columns = ['ds', 'y']
df.tail(n=3)
```

	ds	y
904	2017-06-23	421.0
905	2017-06-24	277.0
906	2017-06-25	253.0

The authors of the library generally advise to make predictions based on at least several months, ideally, more than a year of historical data. Luckily, in our case we have more than a couple of years of data to fit the model.

To measure the quality of our forecast, we need to split our dataset into the *historical part,* which is the first and biggest slice of our data, and the *prediction part,* which will be located at the end of the timeline. We will remove the last month from the dataset in order to use it later as a prediction target:

```
prediction_size = 30
train_df = df[:-prediction_size]
train_df.tail(n=3)
```

	ds	y
874	2017-05-24	375.0
875	2017-05-25	298.0
876	2017-05-26	269.0

Now we need to create a new `Prophet` object. Here we can pass the parameters of the model into the constructor. But in this article we will use the defaults. Then we train our model by invoking its `fit` method on our training dataset:

```
m = Prophet()
m.fit(train_df);
```

Using the helper method `Prophet.make_future_dataframe`, we create a dataframe which will contain all dates from the history and also extend into the future for those 30 days that we left out before.

```
future = m.make_future_dataframe(periods=prediction_size)
future.tail(n=3)
```

	ds
904	2017-06-23
905	2017-06-24
906	2017-06-25

We predict values with `Prophet` by passing in the dates for which we want to create a forecast. If we also supply the historical dates (as in our case), then in addition to the prediction we will get an in-sample fit for the history. Let's call the model's `predict` method with our `future` dataframe as an input:

```
forecast = m.predict(future)
forecast.tail(n=3)
```

|	ds	trend	trend_lower	trend_upper	yhat_lower	yhat_upper	...	weekly_lower	weekly_upper	yearly	yearly_lower	yearly_upper	yhat
904	2017-06-23	276.350830	275.863328	276.704959	252.052719	303.125970	...	3.480611	3.480611	-1.264644	-1.264644	-1.264644	278.566797
905	2017-06-24	277.053781	276.517877	277.428172	216.483803	269.291540	...	-34.024898	-34.024898	-1.067821	-1.067821	-1.067821	241.961063
906	2017-06-25	277.756733	277.181380	278.167349	222.118253	269.361504	...	-30.581790	-30.581790	-0.871454	-0.871454	-0.871454	246.303489

In the resulting dataframe you can see many columns characterizing the prediction, including trend and seasonality components as well as their confidence intervals. The forecast itself is stored in the `yhat` column.

The Prophet library has its own built-in tools for visualization that enable us to quickly evaluate the result.

First, there is a method called `Prophet.plot` that plots all the points from the forecast:

```
m.plot(forecast);
```
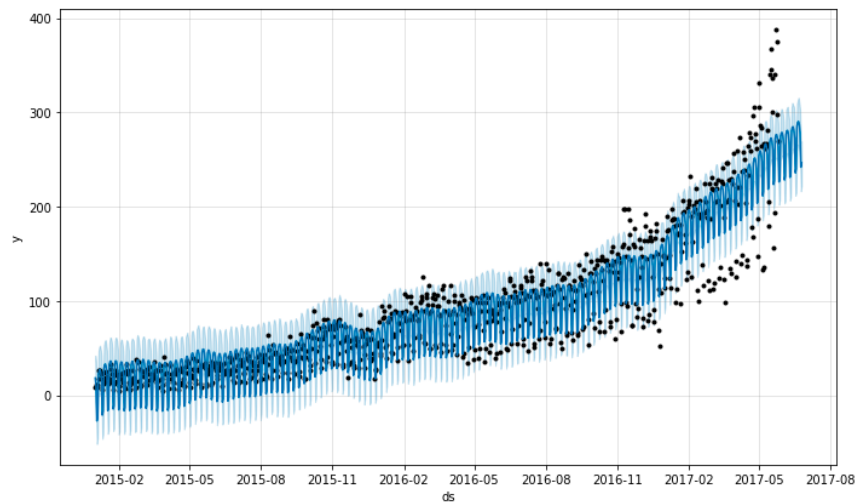

This chart doesn't look very informative. The only definitive conclusion that we can draw here is that the model treated many of the data points as outliers.

The second function `Prophet.plot_components` might be much more useful in our case. It allows us to observe different components of the model separately: trend, yearly and weekly seasonality. In addition, if you supply information about holidays and events to your model, they will also be shown in this plot.

Let's try it out:

```
m.plot_components(forecast);
```
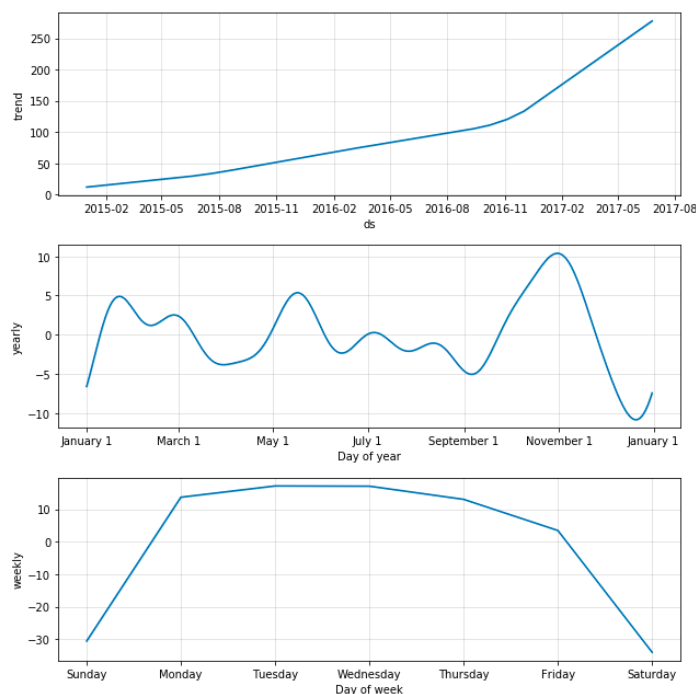
As you can see from the trend graph, Prophet did a good job by fitting the accelerated growth of new posts at the end of 2016. The graph of weekly seasonality leads to the conclusion that usually there are less new posts on Saturdays and Sundays than on the other days of the week. In the yearly seasonality graph there is a prominent dip on Christmas Day.

3.5 Forecast quality evaluation

Let's evaluate the quality of the algorithm by calculating the error metrics for the last 30 days that we predicted. For this, we will need the observations y_i and the corresponding predicted values $\hat{y}_i$.

Let's look into the object `forecast` that the library created for us:

```
print(', '.join(forecast.columns))
```

ds, trend, trend_lower, trend_upper, yhat_lower, yhat_upper, seasonal, seasonal_lower, seasonal_upper, seasonalities, seasonalities_lower, seasonalities_upper, weekly, weekly_lower, weekly_upper, yearly, yearly_lower, yearly_upper, yhat

We can see that this dataframe contains all the information we need except for the historical values. We need to join the `forecast` object with the actual values `y` from the original dataset `df` . For this we will define a helper function that we will reuse later:

```python
1   def make_comparison_dataframe(historical, forecast):
2       """Join the history with the forecast.
3
4           The resulting dataset will contain columns 'yhat
5       """
```

Let's apply this function to our last forecast:

```python
cmp_df = make_comparison_dataframe(df, forecast)
cmp_df.tail(n=3)
```

ds	yhat	yhat_lower	yhat_upper	y
2017-06-23	278.566797	254.997728	303.699502	421.0
2017-06-24	241.961063	216.599532	267.446210	277.0
2017-06-25	246.303489	222.081290	270.730153	253.0

We are also going to define a helper function that we will use to gauge the quality of our forecasting with MAPE and MAE error measures:

```python
1   def calculate_forecast_errors(df, prediction_size):
2       """Calculate MAPE and MAE of the forecast.
3
4           Args:
5               df: joined dataset with 'y' and 'yhat' colu
6               prediction_size: number of days at the end
7       """
8
9       # Make a copy
10      df = df.copy()
11
12      # Now we calculate the values of e_i and p_i accor
13      df['e'] = df['y'] - df['yhat']
14      df['p'] = 100 * df['e'] / df['y']
15
16      # Recall that we held out the values of the last `
17      # in order to predict them and measure the quality
```

Let's use our function:

```python
for err_name, err_value in
 calculate_forecast_errors(cmp_df, prediction_size).items():
    print(err_name, err_value)


MAPE 22.7243579814
MAE 70.452857085
```

As a result, the relative error of our forecast (MAPE) is about 22.72%, and on average our model is wrong by 70.45 posts (MAE).

3.6 Visualization

Let's create our own visualization of the model built by Prophet. It will comprise the actual values, forecast and confidence intervals.

First, we will plot the data for a shorter period of time to make the data points easier to distinguish. Second, we will show the model performance only for the period that we predicted, that is the last 30 days. It seems that these two measures should give us a more legible plot.

Third, we will use `Plotly` to make our chart interactive, which is great for exploring.

We will define a custom helper function `show_forecast` and call it (for more on how it works please refer to the comments in the code and the documentation):

```python
1   def show_forecast(cmp_df, num_predictions, num_values,
2       """Visualize the forecast."""
3
4       def create_go(name, column, num, **kwargs):
5           points = cmp_df.tail(num)
6           args = dict(name=name, x=points.index, y=point
7           args.update(kwargs)
8           return go.Scatter(**args)
9
10      lower_bound = create_go('Lower Bound', 'yhat_lower
11                              line=dict(width=0),
12                              marker=dict(color="444"))
13      upper_bound = create_go('Upper Bound', 'yhat_upper
14                              line=dict(width=0),
15                              marker=dict(color="444"),
16                              fillcolor='rgba(68, 68, 68
17                              fill='tonexty')
18      forecast = create_go('Forecast', 'yhat', num_predi
19                              line=dict(color='rgb(31, 119,
20      actual = create_go('Actual', 'y', num_values
```

New posts on Medium
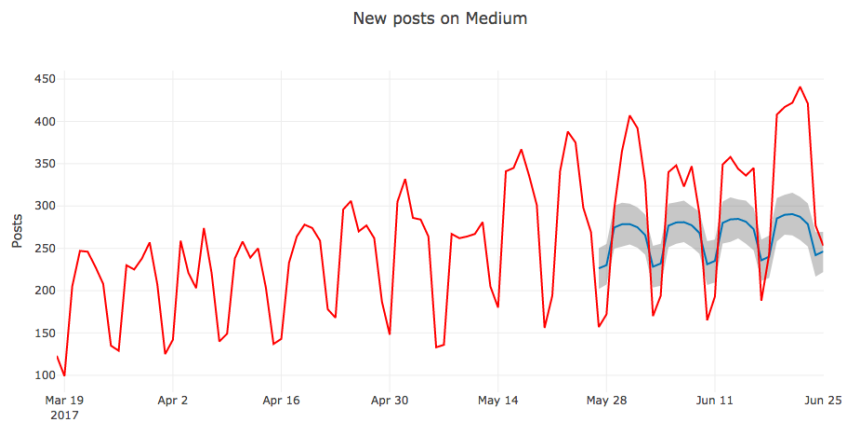
At first glance, the prediction of the mean values by our model seems to be sensible. The high value of MAPE that we got above may be explained by the fact that the model failed to catch on to increasing peak-to-peak amplitude of weakly seasonality.

Also, we can conclude from the graph above that many of the actual values lie outside the confidence interval. Prophet may not be suitable for time series with unstable variance, at least when the default settings are used. We will try to fix this by applying a transform to our data.

4. Box-Cox Transformation

So far we have used Prophet with the default settings and the original data. We will leave the parameters of the model alone. But despite this we still have some room for improvement. In this section, we will apply the Box–Cox transformation to our original series. Let's see where it will lead us.

A few words about this transformation. This is a monotonic data transformation that can be used to stabilize variance. We will use the one-parameter Box–Cox transformation, which is defined by the following expression:

$$
boxcox^{(\lambda)}(y_i) = \begin{cases} \dfrac{y_i^{\lambda} - 1}{\lambda} & \text{, if } \lambda \neq 0. \\ ln(y_i) & \text{, if } \lambda = 0. \end{cases}
$$

We will need to implement the inverse of this function in order to be able to restore the original data scale. It is easy to see that the inverse is defined as:

$$
invboxcox^{(\lambda)}(y_i) = \begin{cases} e^{\left(\dfrac{ln(\lambda y_i + 1)}{\lambda}\right)} & \text{, if } \lambda \neq 0. \\ e^{y_i} & \text{, if } \lambda = 0. \end{cases}
$$

The corresponding function in Python is implemented as follows:

```
1    def inverse_boxcox(y, lambda_):
2        return np.exp(y) if lambda_ == 0 else np.exp(np.log
```

First, we prepare our dataset by setting its index:

```
train_df2 = train_df.copy().set_index('ds')
```

Then, we apply the function `stats.boxcox` from `Scipy`, which applies the Box–Cox transformation. In our case it will return two values. The first one is the transformed series and the second one is the found value of λ that is optimal in terms of the maximum log-likelihood:

```
train_df2['y'], lambda_prophet =
stats.boxcox(train_df2['y'])
train_df2.reset_index(inplace=True)
```

We create a new `Prophet` model and repeat the fit-predict cycle that we have already done above:

```
m2 = Prophet()
m2.fit(train_df2)
future2 = m2.make_future_dataframe(periods=prediction_size)
forecast2 = m2.predict(future2)
```

At this point, we need to revert the Box–Cox transformation with our inverse function and the known value of λ:

```
for column in ['yhat', 'yhat_lower', 'yhat_upper']:
    forecast2[column] = inverse_boxcox(forecast2[column],
                                        lambda_prophet)
```

Here we will reuse our tools for making the comparison dataframe and calculating the errors:

```
cmp_df2 = make_comparison_dataframe(df, forecast2)
for err_name, err_value in
calculate_forecast_errors(cmp_df2, prediction_size).items():
    print(err_name, err_value)
```

```
MAPE 11.5921879552
MAE 39.072031256
```

So, we can definitely state an increase in the quality of the model.

Finally, let's plot our previous performance with the latest results side-by-side. Note that we use `prediction_size` for the third parameter in order to zoom in on the interval being predicted:
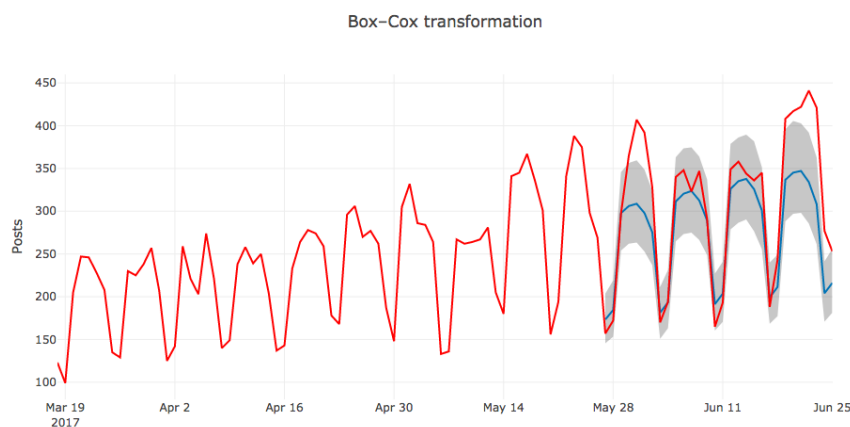
```
show_forecast(cmp_df, prediction_size, 100, 'No
transformations')
show_forecast(cmp_df2, prediction_size, 100, 'Box—Cox
transformation')
```


No transformations

Box–Cox transformation
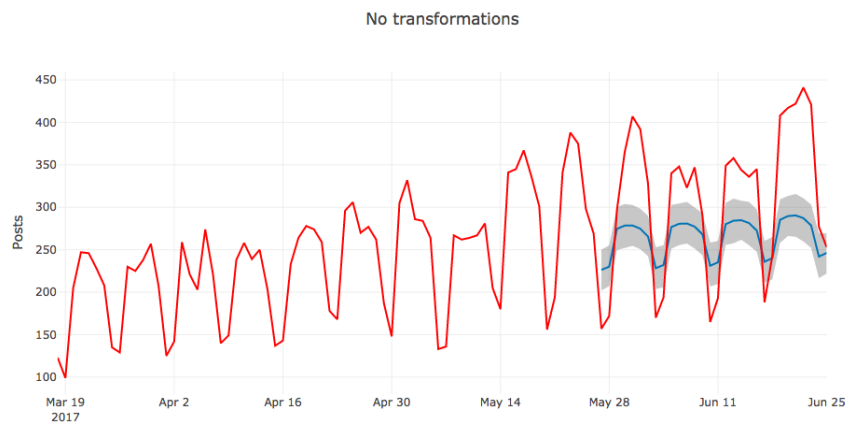
We see that the forecast of weekly changes in the second graph is much closer to the real values now.

5. Summary

We have taken a look at *Prophet*, an open-source forecasting library that is specifically targeted at business time series. We have also done some hands-on practice in time series prediction.

As we have seen, the Prophet library does not make wonders, and its predictions out-of-box are not <u>ideal</u>. It is still up to the data scientist to explore the forecast results, tune model parameters and transform data when necessary.

However, this library is user-friendly and easily customizable. The sole ability to take into account abnormal days that are known to the analyst beforehand might make a difference in some cases.

All in all, the Prophet library is worth being a part of your analytical toolbox.

6. References

- Official <u>Prophet repository</u> on GitHub.

- Official <u>Prophet documentation</u>.

- Sean J. Taylor, Benjamin Letham <u>"Forecasting at scale"</u>—scientific paper explaining the algorithm which lays the foundation of `Prophet`.

- Forecasting Website Traffic Using Facebook's Prophet Library—
  `Prophet` overview with an example of website traffic forecasting.

- Rob J. Hyndman, George Athanasopoulos "Forecasting: principles and practice"—a very good online book about time series forecasting.

.　　.　　.

Author: Egor Polusmak, Head of Data Analytics at Mail.Ru Search.