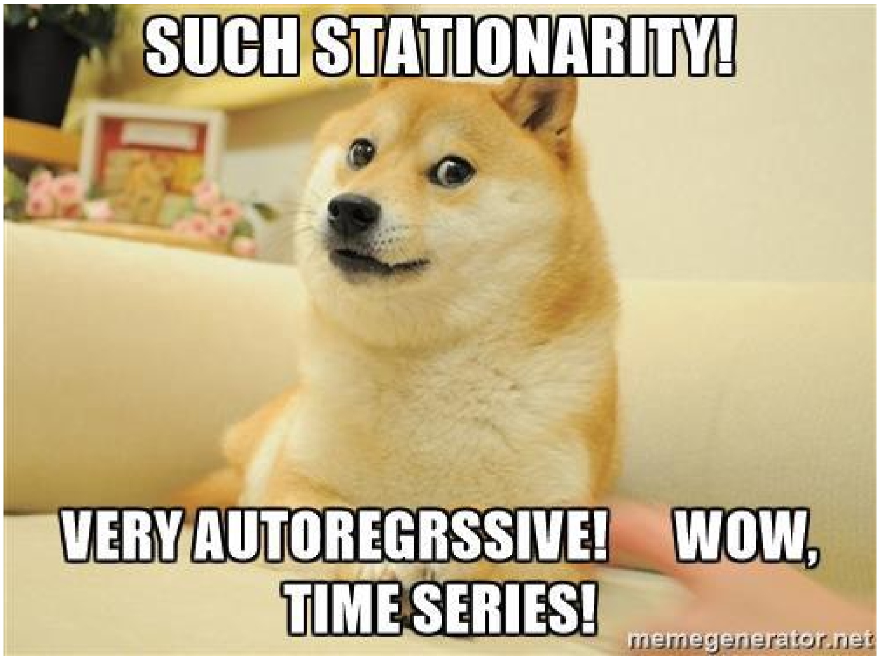
Dmitriy Sergeev Follow
Data Scientist at Zeptolab
Apr 10 · 23 min read

Open Machine Learning Course. Topic 9. Part 1. Time series analysis in Python



Hi there!

We continue our open machine learning course with a new article on time series.

Let’s take a look at how to work with time series in Python, what methods and models we can use for prediction; what’s double and triple exponential smoothing; what to do if stationarity is not you favorite game; how to build SARIMA and stay alive; how to make predictions using xgboost. All of this will be applied to (harsh) real world example.

Article outline

—Quality metrics

2. Move, smoothe, evaluate
—Rolling window estimations
—Exponential smoothing, Holt-Winters model
—Time-series cross validation, parameters selection

3. Econometric approach
—Stationarity, unit root
—Getting rid of non-stationarity
—SARIMA intuition and model building

4. Linear (and not quite) models on time series
—Feature extraction
—Linear models, feature importance
—Regularization, feature selection
—XGBoost

5. Assignment #9

The following content is better viewed and reproduced as a Jupyter-notebook

In my day to day job I encounter time series-connected tasks almost every day. The most frequent question is—what will happen with our metrics in the next day/week/month/etc.—how many players will install the app, how much time will they spend online, how many actions users will do, and so forth. We can approach prediction task using different methods, depending on the required quality of the prediction, length of the forecasted period, and, of course, time we have to choose features and tune parameters to achieve desired results.

Introduction

> *Small definition of time series:*
> Time series—is a series of data points indexed (or listed or graphed) in time order.

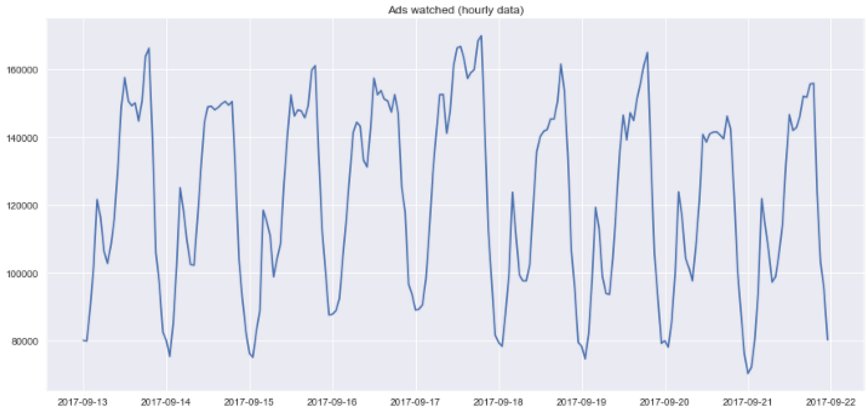
Therefore data is organized around relatively deterministic timestamps, and therefore, compared to random samples, may contain additional information that we will try to extract.

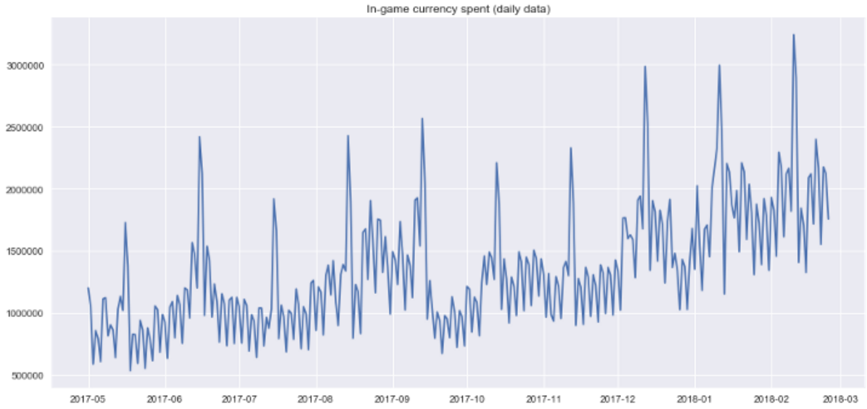
Let’s import some libraries. First and foremost we will need statsmodels library that has tons of statistical modeling functions, including time series. For R afficionados (that had to move to python) statsmodels will definitely look familiar as it supports model definitions like ‘Wage ~ Age + Education’.

```
import numpy as np                                    # vectors
import pandas as pd                                   # tables
import matplotlib.pyplot as plt                       # plots
import seaborn as sns                                 # more pl

from dateutil.relativedelta import relativedelta # working
from scipy.optimize import minimize                   # for fun

import statsmodels.formula.api as smf                 # statist
import statsmodels.tsa.api as smt
import statsmodels.api as sm
import scipy.stats as scs
```

As an example let’s use some real mobile game data on hourly ads watched by players and daily in-game currency spent:

```
ads = pd.read_csv('../../data/ads.csv', index_col=['Time']
currency = pd.read_csv('../../data/currency.csv', index_co

plt.figure(figsize=(15, 7))
plt.plot(ads.Ads)
plt.title('Ads watched (hourly data)')
plt.grid(True)
plt.show()

```





Forecast quality metrics

Before actually forecasting, let's understand how to measure the quality of predictions and have a look at the most common and widely used metrics

- **R squared**, coefficient of determination (in econometrics it can be interpreted as a percentage of variance explained by the model), (-*inf*, 1] `sklearn.metrics.r2_score`
- **Mean Absolute Error**, it is an interpretable metric because it has the same unit of measurement as the initial series, [0, +*inf*) `sklearn.metrics.mean_absolute_error`
- **Median Absolute Error**, again an interpretable metric, particularly interesting because it is robust to outliers, [0, +*inf*) `sklearn.metrics.median_absolute_error`
- **Mean Squared Error**, most commonly used, gives higher penalty to big mistakes and vise versa, [0, +*inf)*

`sklearn.metrics.mean_squared_error`

- **Mean Squared Logarithmic Error**, practically the same as MSE but we initially take logarithm of the series, as a result we give attention to small mistakes as well, usually is used when data has exponential trends, [0, +*inf*) `sklearn.metrics.mean_squared_log_error`
- **Mean Absolute Percentage Error**, same as MAE but percentage, —very convenient when you want to explain the quality of the model to your management, [0, +*inf*), not implemented in sklearn

```python
# Importing everything from above

from sklearn.metrics import r2_score, median_absolute_error
from sklearn.metrics import median_absolute_error, mean_squ

```

Excellent, now we know how to measure the quality of the forecasts, what metrics can we use and how to translate the results to the boss. Little thing is left—building the model.

Move, smoothe, evaluate

Let's start with a naive hypothesis—"tomorrow will be the same as today", but instead of a model like ŷ(t)=y(t−1) (which is actually a great baseline for any time series prediction problems and sometimes it's impossible to beat it with any model) we'll assume that the future value of the variable depends on the average ***n*** of its previous values and therefore we'll use **moving average**.

$$\hat{y}_t = \frac{1}{k} \sum_{n=0}^{k-1} y_{t-n}$$

```
def moving_average(series, n):
    """
        Calculate average of last n observations
    """
    return np.average(series[-n:])
```

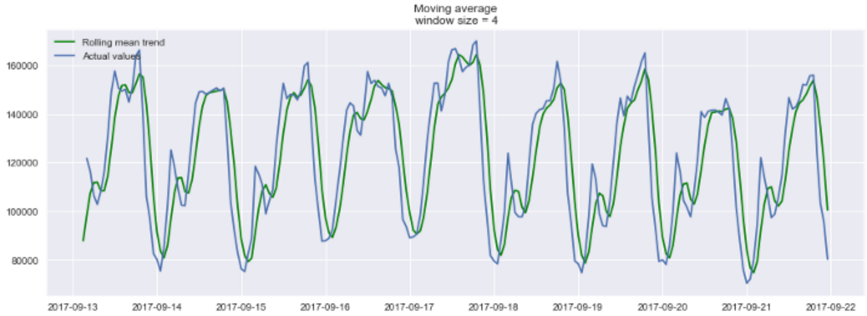
```
Out: 116805.0
```

Unfortunately we can’t make this prediction long-term—to get one for the next step we need the previous value to be actually observed. But moving average has another use case—smoothing of the original time series to indicate trends. Pandas has an implementation available `DataFrame.rolling(window).mean()` . The wider the window - the smoother will be the trend. In the case of the very noisy data, which can be very often encountered in finance, this procedure can help to detect common patterns.

```
def plotMovingAverage(series, window, plot_intervals=False

    """
        series - dataframe with timeseries
        window - rolling window size
        plot_intervals - show confidence intervals
        plot_anomalies - show anomalies

    """
    rolling_mean = series.rolling(window=window).mean()

    plt.figure(figsize=(15,5))
    plt.title("Moving average\n window size = {}".format(w
    plt.plot(rolling_mean, "g", label="Rolling mean trend"

    # Plot confidence intervals for smoothed values
    if plot_intervals:
        mae = mean_absolute_error(series[window:], rolling
        deviation = np.std(series[window:] - rolling_mean[
        lower_bond = rolling_mean - (mae + scale * deviati
```

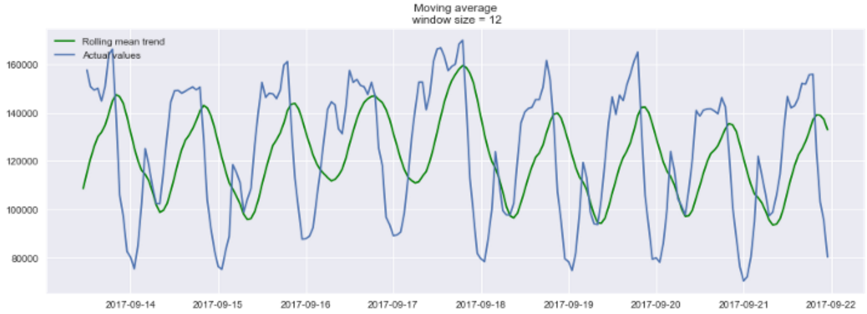
Smoothing by last 4 hours

```
plotMovingAverage(ads, 4)
```



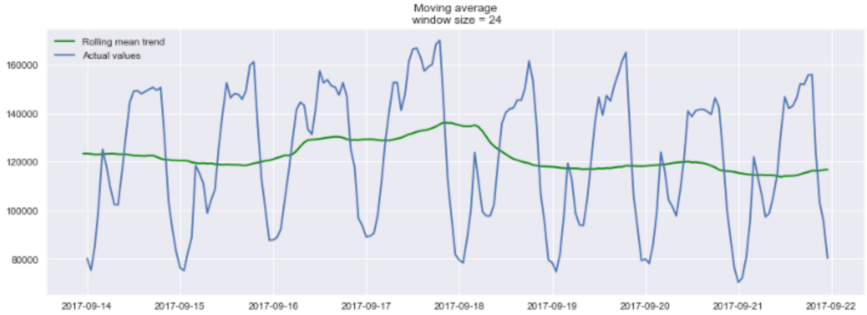
Smoothing by last 12 hours

```
plotMovingAverage(ads, 12)
```



Smoothing by 24 hours—we get daily trend

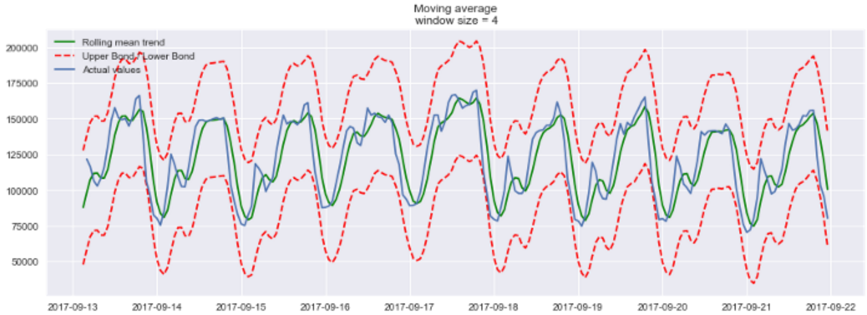
```
plotMovingAverage(ads, 24)
```



As you can see, applying daily smoothing on hour data allowed us to clearly see the dynamics of ads watched. During the weekends the values are higher (weekends—time to play) and weekdays are generally lower.

We can also plot confidence intervals for our smoothed values

`plotMovingAverage(ads, 4, plot_intervals=True)`

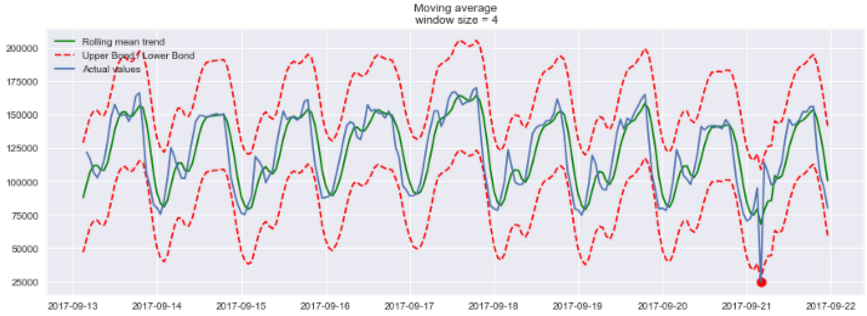


And now let’s create a simple anomaly detection system with the help of the moving average. Unfortunately, in this particular series everything is more or less normal, so we’ll intentionally make one of the values abnormal in the dataframe `ads_anomaly`

```
ads_anomaly = ads.copy()
ads_anomaly.iloc[-20] = ads_anomaly.iloc[-20] * 0.2 # say w
```

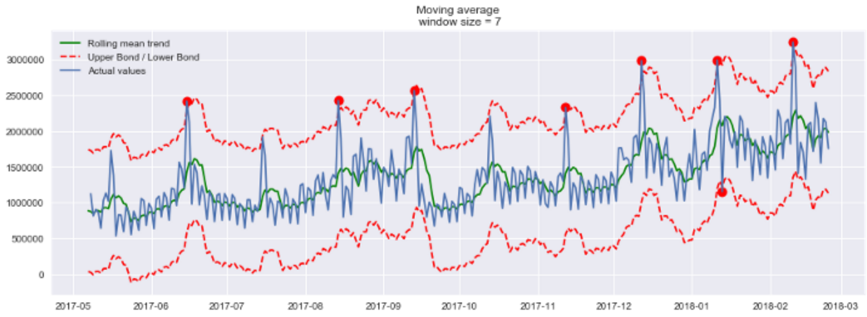
Let’s see, if this simple method can catch the anomaly

`plotMovingAverage(ads_anomaly, 4, plot_intervals=True, plot_anomalies=True)`



Neat! What about the second series (with weekly smoothing)?

`plotMovingAverage(currency, 7, plot_intervals=True, plot_anomalies=True)`



Oh no! Here is the downside of our simple approach—it did not catch monthly seasonality in our data and marked almost all 30-day peaks as an anomaly. If you don't want to have that many false alarms—it's best to consider more complex models.

Weighted average is a simple modification of the moving average, inside of which observations have different weights summing up to one, usually more recent observations have greater weight.

$$\hat{y}_t = \sum_{n=1}^{k} \omega_n y_{t+1-n}$$

```
def weighted_average(series, weights):
    """
        Calculate weighter average on series
    """
    result = 0.0
    weights.reverse()
    for n in range(len(weights)):
```

```
Out: 98423.0
```

Exponential smoothing

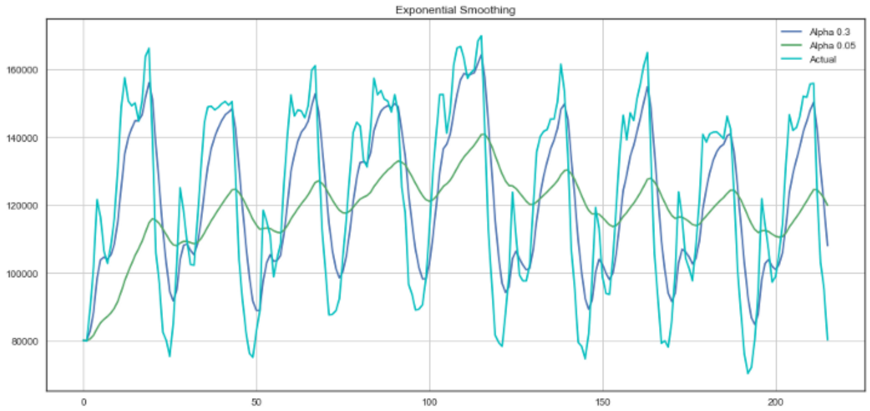
And now let’s take a look at what happens if instead of weighting the last nn values of the time series we start weighting all available observations while exponentially decreasing weights as we move further back in historical data. There’s a formula of the simple **exponential smoothing** that will help us in that:

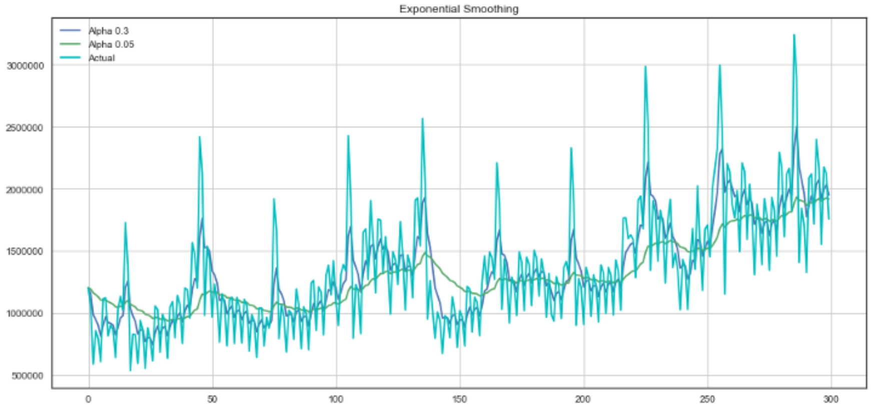
$$\hat{y}_t = \alpha \cdot y_t + (1 - \alpha) \cdot \hat{y}_{t-1}$$

Here the model value is a weighted average between the current true value and the previous model values. The α weight is called a smoothing factor. It defines how quickly we will “forget” the last available true observation. The less α is the more influence previous model values have, and the smoother the series is.

Exponentiality is hiding in the recursivity of the function—we multiply each time (1−α) by the previous model value which, in its turn, also containes (1−α) and so forth until the very beginning.

```
def exponential_smoothing(series, alpha):
    """
        series - dataset with timestamps
        alpha - float [0.0, 1.0], smoothing parameter
    """
    result = [series[0]] # first value is same as series
    for n in range(1, len(series)):
        result.append(alpha * series[n] + (1 - alpha) * re
    return result

def plotExponentialSmoothing(series, alphas):
    """
        Plots exponential smoothing with different alphas

        series - dataset with timestamps
        alphas - list of floats, smoothing parameters

    """
```





Double exponential smoothing

Until now all we could get from our methods in the best case was just a single future point prediction (and also some nice smoothing), that's cool but not enough, so let's extend exponential smoothing so that we can predict two future points (of course, we also get some smoothing).

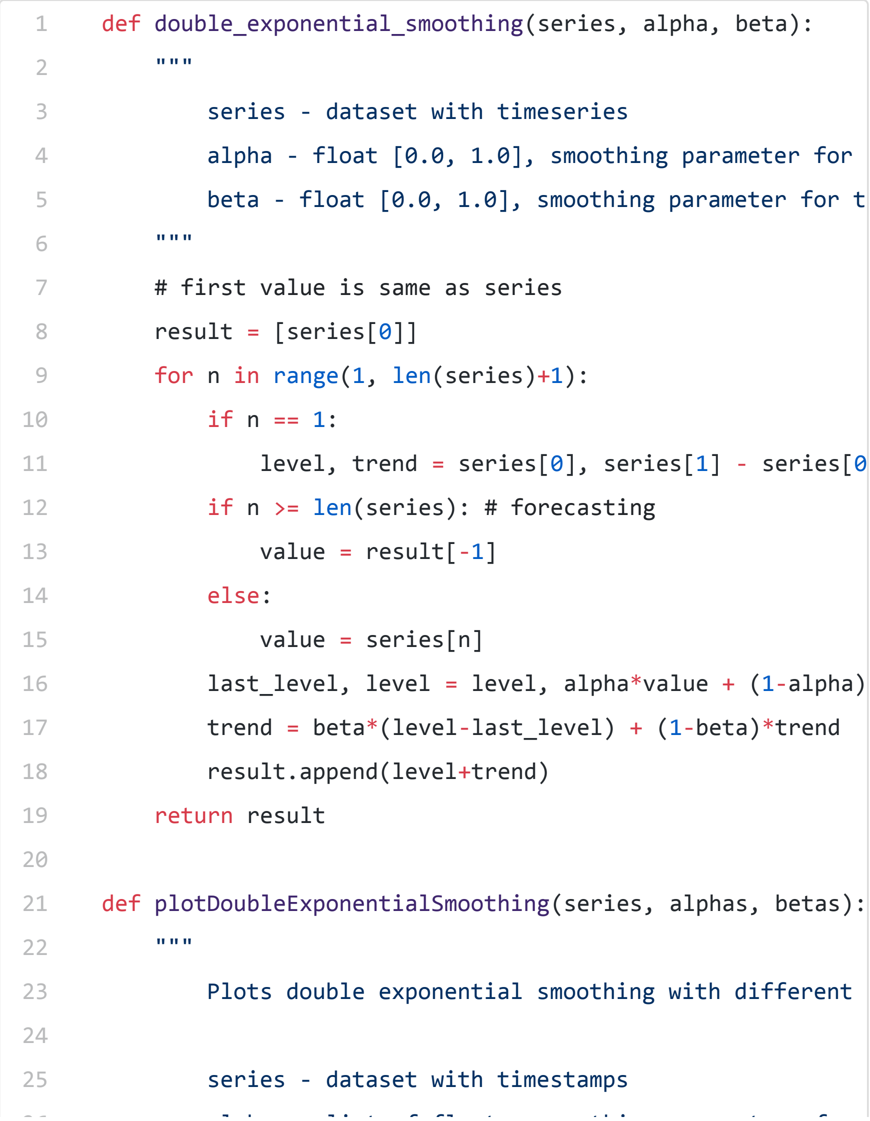
Series decomposition should help us—we obtain two components: intercept (also, level) ℓ and trend (also, slope) b. We've learnt to predict intercept (or expected series value) using previous methods, and now we will apply the same exponential smoothing to the trend, believing naively or perhaps not that the future direction of the time series changes depends on the previous weighted changes.

$$\ell_x = \alpha y_x + (1 - \alpha)(\ell_{x-1} + b_{x-1})$$

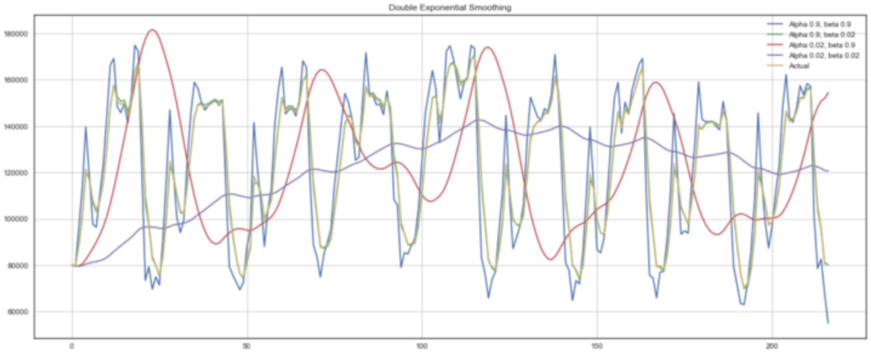
$$b_x = \beta(\ell_x - \ell_{x-1}) + (1 - \beta)b_{x-1}$$

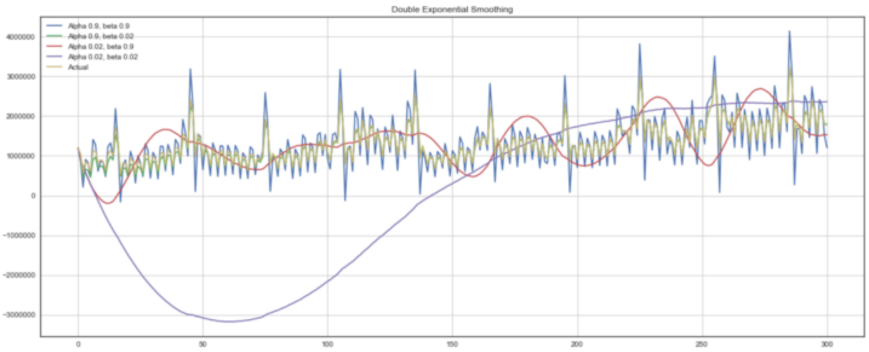
$$\hat{y}_{x+1} = \ell_x + b_x$$

As a result we get a set of functions. The first one describes intercept, as before it depends on the current value of the series, and the second term is now split into previous values of the level and of the trend. The second function describes trend—it depends on the level changes at the current step and on the previous value of the trend. In this case β coefficient is a weight in the exponential smoothing. The final prediction is the sum of the model values of the intercept and trend.



```
def double_exponential_smoothing(series, alpha, beta):
    """
        series - dataset with timeseries
        alpha - float [0.0, 1.0], smoothing parameter for
        beta - float [0.0, 1.0], smoothing parameter for t
    """
    # first value is same as series
    result = [series[0]]
    for n in range(1, len(series)+1):
        if n == 1:
            level, trend = series[0], series[1] - series[0
        if n >= len(series): # forecasting
            value = result[-1]
        else:
            value = series[n]
        last_level, level = level, alpha*value + (1-alpha)
        trend = beta*(level-last_level) + (1-beta)*trend
        result.append(level+trend)
    return result

def plotDoubleExponentialSmoothing(series, alphas, betas):
    """
        Plots double exponential smoothing with different

        series - dataset with timestamps
```





Now we have to tune two parameters—α and β. The former is responsible for the series smoothing around trend, and the latter for the smoothing of the trend itself. The bigger the values, the more weight the latest observations will have and the less smoothed the model series will be. Combinations of the parameters may produce really weird results, especially if set manually. We'll look into choosing parameters automatically in a bit, immediately after triple exponential smoothing.

Triple exponential smoothing a.k.a. Holt-Winters

Hooray! We've successfully reached our next variant of exponential smoothing, this time triple.

The idea of this method is that we add another, third component—seasonality. This means we should't use the method if our time series do not have seasonality, which is not the case in our example. Seasonal component in the model will explain repeated variations around intercept and trend, and it will be described by the length of the season, in other words by the period after which variations repeat. For each observation in the season there's a separate component, for example, if the length of the season is 7 (weekly seasonality), we will have 7 seasonal components, one for each day of the week.

Now we get a new system:

$$\ell_x = \alpha(y_x - s_{x-L}) + (1-\alpha)(\ell_{x-1} + b_{x-1})$$

$$b_x = \beta(\ell_x - \ell_{x-1}) + (1-\beta)b_{x-1}$$

$$s_x = \gamma(y_x - \ell_x) + (1-\gamma)s_{x-L}$$

$$\hat{y}_{x+m} = \ell_x + mb_x + s_{x-L+1+(m-1)modL}$$

Intercept now depends on the current value of the series minus corresponding seasonal component, trend stays unchanged, and the seasonal component depends on the current value of the series minus intercept and on the previous value of the component. Please take into account that the component is smoothed through all the available seasons, for example, if we have a Monday component then it will only be averaged with other Mondays. You can read more on how averaging works and how initial approximation of the trend and seasonal components is done here. Now that we have seasonal component we can predict not one and not even two but arbitrary mm future steps which is very encouraging.

Below is the code for a triple exponential smoothing model, also known by the last names of its creators—Charles Holt and his student Peter Winters. Additionally Brutlag method was included into the model to build confidence intervals:

$$\hat{y}_{max_x} = \ell_{x-1} + b_{x-1} + s_{x-T} + m \cdot d_{t-T}$$

$$\hat{y}_{min_x} = \ell_{x-1} + b_{x-1} + s_{x-T} - m \cdot d_{t-T}$$

$$d_t = \gamma \mid y_t - \hat{y}_t \mid + (1-\gamma)d_{t-T},$$

where T is the length of the season, d is the predicted deviation, and the other parameters were taken from the triple exponential smoothing. You can read more about the method and its applicability to anomalies detection in time series here.

```
class HoltWinters:

    """
    Holt-Winters model with the anomalies detection using

    # series - initial time series
    # slen - length of a season
    # alpha, beta, gamma - Holt-Winters model coefficient
    # n_preds - predictions horizon
    # scaling_factor - sets the width of the confidence i

    """


    def __init__(self, series, slen, alpha, beta, gamma,
        self.series = series
        self.slen = slen
        self.alpha = alpha
        self.beta = beta
        self.gamma = gamma
        self.n_preds = n_preds
        self.scaling_factor = scaling_factor


    def initial_trend(self):
        sum = 0.0
        for i in range(self.slen):
            sum += float(self.series[i+self.slen] - self.
        return sum / self.slen

    def initial_seasonal_components(self):
        seasonals = {}
        season_averages = []
        n_seasons = int(len(self.series)/self.slen)
        # let's calculate season averages
        for j in range(n_seasons):
            season_averages.append(sum(self.series[self.s
        # let's calculate initial values
        for i in range(self.slen):
            sum_of_vals_over_avg = 0.0
            for j in range(n_seasons):
                sum_of_vals_over_avg += self.series[self.
            seasonals[i] = sum_of_vals_over_avg/n_seasons
        return seasonals


    def triple_exponential_smoothing(self):
        self.result = []
        self.Smooth = []
        self.Season = []
        self.Trend = []
        self.PredictedDeviation = []
        self.UpperBond = []
        self.LowerBond = []

        seasonals = self.initial_seasonal_components()

        for i in range(len(self.series)+self.n_preds):
            if i == 0: # components initialization
                smooth = self.series[0]
                trend = self.initial_trend()
                self.result.append(self.series[0])
```

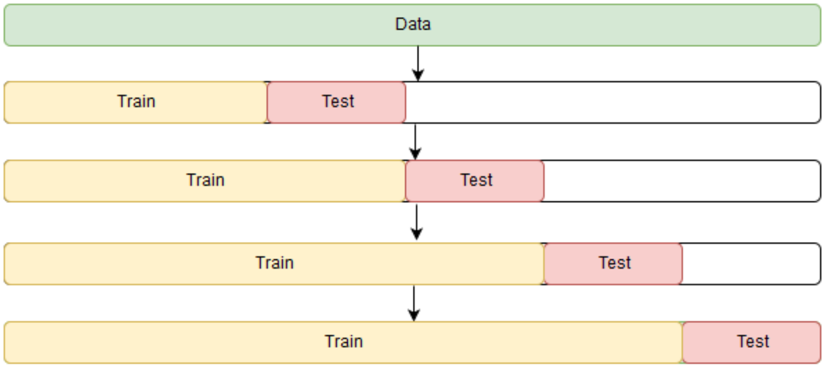
Time series cross validation

Before we start building model let's talk first about how to estimate model parameters automatically.

There's nothing unusual here, as always we have to choose a loss function suitable for the task, that will tell us how close the model approximates data. Then using cross-validation we will evaluate our chosen loss function for given model parameters, calculate gradient, adjust model parameters and so forth, bravely descending to the global minimum of error.

The question is how to do cross-validation on time series, because, you know, time series do have time structure and one just can't randomly mix values in a fold without preserving this structure, otherwise all time dependencies between observations will be lost. That's why we will have to use a bit more tricky approach to optimization of the model parameters, I don't know if there's an official name to it but on CrossValidated, where one can find all the answers but the Answer to the Ultimate Question of Life, the Universe, and Everything, “cross-validation on a rolling basis” was proposed as a name.

The idea is rather simple—we train our model on a small segment of the time series, from the beginning until some t, make predictions for the next $t+n$ steps and calculate an error. Then we expand our training sample until $t+n$ value and make predictions from $t+n$ until $t+2*n$, and we continue moving our test segment of the time series until we hit the last available observation. As a result we have as many folds as many n will fit between the initial training sample and the last observation.



Now, knowing how to set cross-validation, we will find optimal parameters for the Holt-Winters model, recall that we have daily seasonality in ads, hence the `slen=24` parameter

```
from sklearn.model_selection import TimeSeriesSplit # you

def timeseriesCVscore(params, series, loss_function=mean_s
    """
        Returns error on CV

        params - vector of parameters for optimization
        series - dataset with timeseries
        slen - season length for Holt-Winters model
    """
    # errors array
    errors = []

    values = series.values
    alpha, beta, gamma = params

    # set the number of folds for cross-validation
    tscv = TimeSeriesSplit(n_splits=3)

```

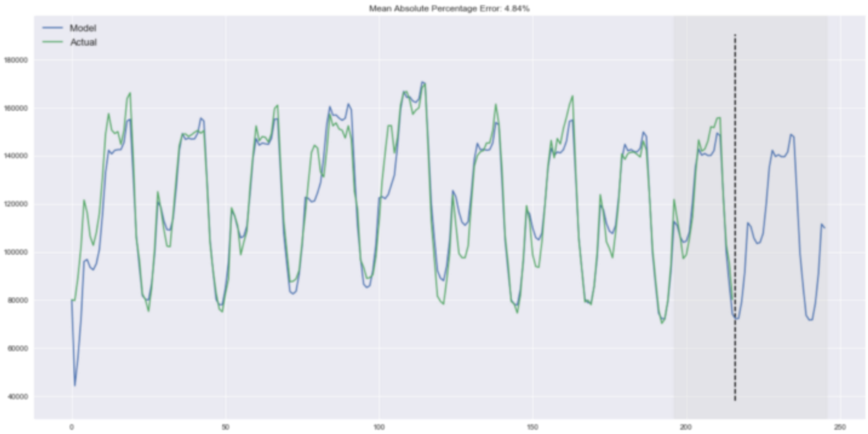
In the Holt-Winters model, as well as in the other models of exponential smoothing, there’s a constraint on how big smoothing parameters could be, each of them is in the range from 0 to 1, therefore to minimize loss function we have to choose an algorithm that supports constraints on model parameters, in our case—Truncated Newton conjugate gradient.

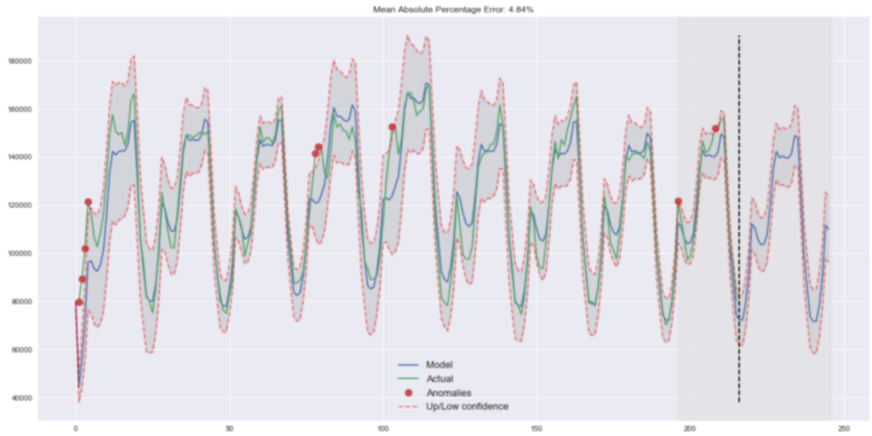
```
%%time
data = ads.Ads[:-20] # leave some data for testing

# initializing model parameters alpha, beta and gamma
x = [0, 0, 0]

# Minimizing the loss function
opt = minimize(timeseriesCVscore, x0=x,
               args=(data, mean_squared_log_error),
               method="TNC", bounds = ((0, 1), (0, 1), (0,
              )

# Take optimal values...
alpha_final, beta_final, gamma_final = opt.x
```

```
0.11652680227350454 0.002677697431105852 0.05820973606789237
CPU times: user 1.96 s, sys: 17.3 ms, total: 1.98 s
Wall time: 2 s
```

Chart rendering code

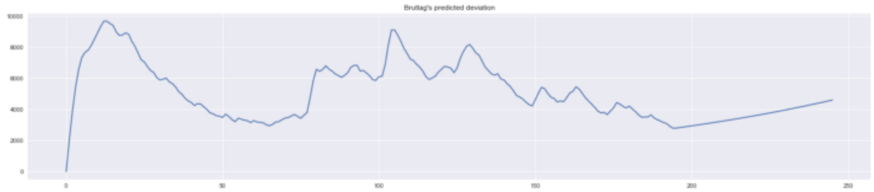
```python
def plotHoltWinters(series, plot_intervals=False, plot_ano
    """
        series - dataset with timeseries
        plot_intervals - show confidence intervals
        plot_anomalies - show anomalies
    """

    plt.figure(figsize=(20, 10))
    plt.plot(model.result, label = "Model")
    plt.plot(series.values, label = "Actual")
    error = mean_absolute_percentage_error(series.values,
    plt.title("Mean Absolute Percentage Error: {0:.2f}%".f

    if plot_anomalies:
        anomalies = np.array([np.NaN]*len(series))
        anomalies[series.values<model.LowerBond[:len(serie
            series.values[series.values<model.LowerBond[:l
        anomalies[series.values>model.UpperBond[:len(serie
            series.values[series.values>model.UpperBond[:l
        plt.plot(anomalies, "o", markersize=10, label = "A

```





Judging by the chart, our model was able to successfully approximate the initial time series, catching daily seasonality, overall downwards trend and even some anomalies. If you take a look at the modeled deviation, you can clearly see that the model reacts quite sharply to the changes in the structure of the series but then quickly returns deviation to the normal values, “forgetting” the past. This feature of the model allows us to quickly build anomaly detection systems even for quite noisy series without spending too much time and money on preparing data and training the model.

```python
plt.figure(figsize=(25, 5))
plt.plot(model.PredictedDeviation)
plt.grid(True)
plt.axis('tight')
```

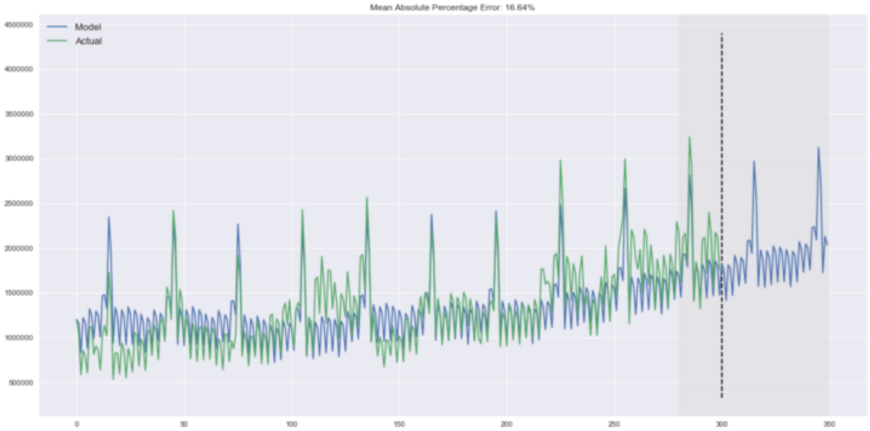


Deviation increases when we have anomalies

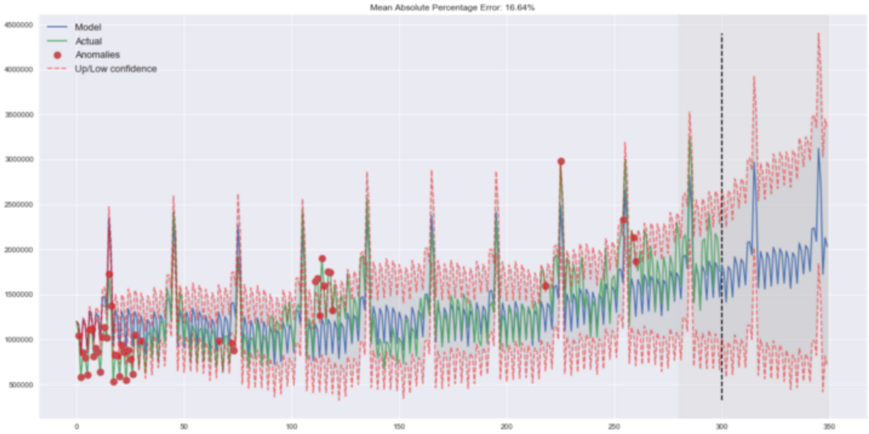
We’ll apply the same algorithm for the second series which, as we know, has trend and 30-day seasonality

```
%%time
data = currency.GEMS_GEMS_SPENT[:-50]
slen = 30 # 30-day seasonality

x = [0, 0, 0]

opt = minimize(timeseriesCVscore, x0=x,
               args=(data, mean_absolute_percentage_error,
               method="TNC", bounds = ((0, 1), (0, 1), (0,
              )

alpha_final, beta_final, gamma_final = opt.x
print(alpha_final, beta_final, gamma_final)
```

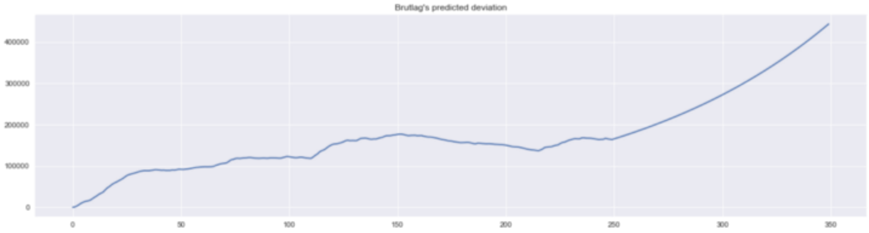
```
0.012841445048055122 0.04883371471892228 0.00943678056045777
CPU times: user 3.03 s, sys: 24.8 ms, total: 3.05 s
Wall time: 3.11 s
```



Looks quite adequate, model has caught both upwards trend and seasonal spikes and overall fits our values nicely



Caught some anomalies as well



Deviations increases as forecast period moves further

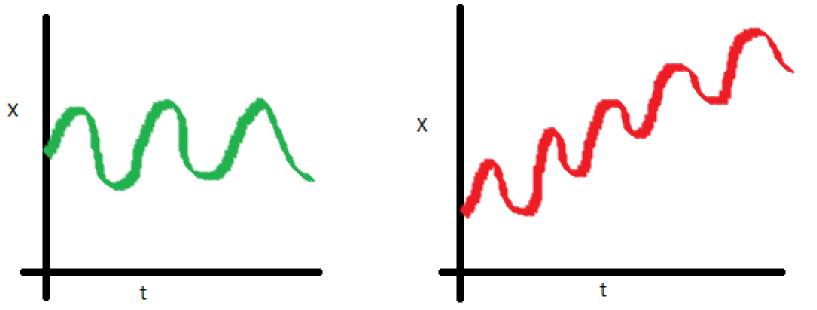
Econometric approach

Stationarity

Before we start modeling we should mention such an important property of time series as **stationarity**.

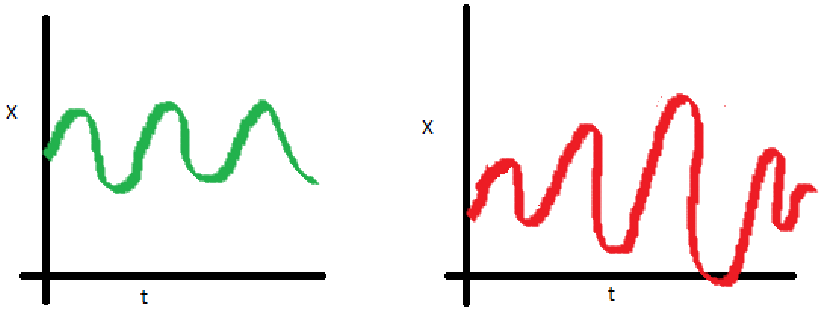
If the process is stationary that means it doesn't change its statistical properties over time, namely mean and variance do not change over time (constancy of variance is also called homoscedasticity), also covariance function does not depend on the time (should only depend on the distance between observations). You can see this visually on the pictures from the post of Sean Abu:

- The red graph below is not stationary because the mean increases over time.



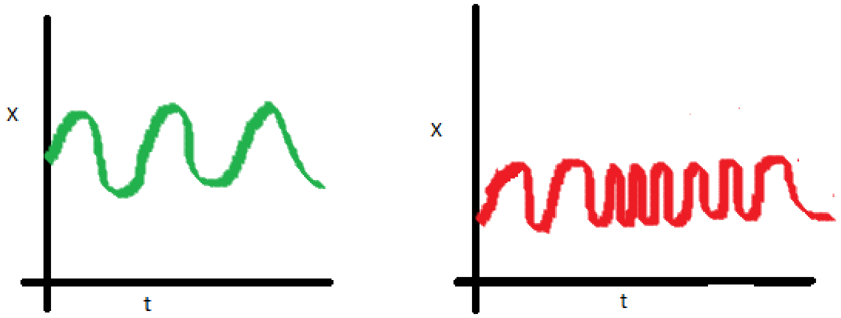
Stationary series Non-Stationary series

- We were unlucky with the variance, see the varying spread of values over time



Stationary series Non-Stationary series

- Finally, the covariance of the i th term and the (i + m) th term should not be a function of time. In the following graph, you will notice the spread becomes closer as the time increases. Hence, the covariance is not constant with time for the right chart.



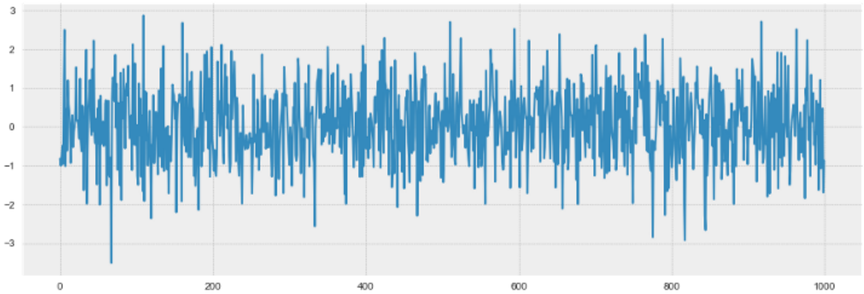
Stationary series Non-Stationary series

So why stationarity is so important? Because it's easy to make predictions on the stationary series as we assume that the future statistical properties will not be different from the currently observed. Most of the time series models in one way or the other model and predict those properties (mean or variance, for example), that's why predictions would be wrong if the original series were not stationary. Unfortunately most of the time series we see outside of textbooks are non-stationary but we can (and should) change this.

So, to fight non-stationarity we have to know our enemy so to say. Let's see how to detect it. To do that we will now take a look at the white noise and random walks and we will learn how to get from one to another for free, without registration and SMS.

White noise chart:

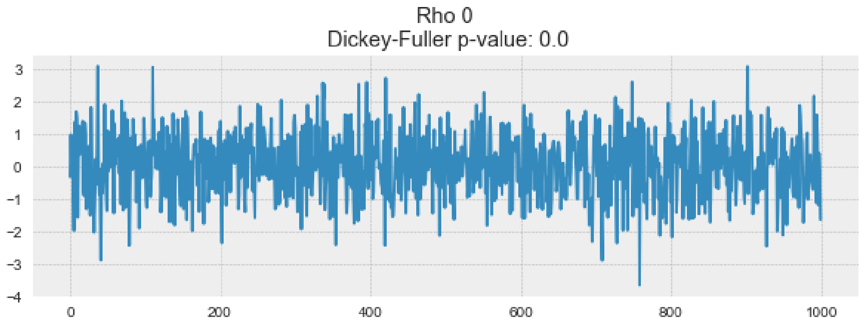
```
white_noise = np.random.normal(size=1000)
with plt.style.context('bmh'):
    plt.figure(figsize=(15, 5))
```

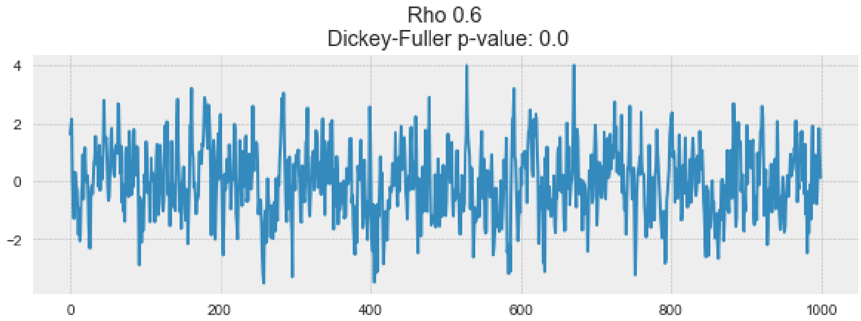


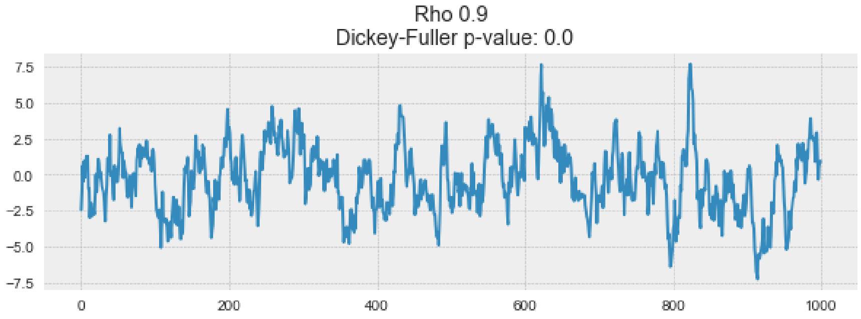
So the process generated by standard normal distribution is stationary and oscillates around 0 with with deviation of 1. Now based on this process we will generate a new one where each next value will depend on the previous one: x(t)=ρ*x(t−1)+e(t)

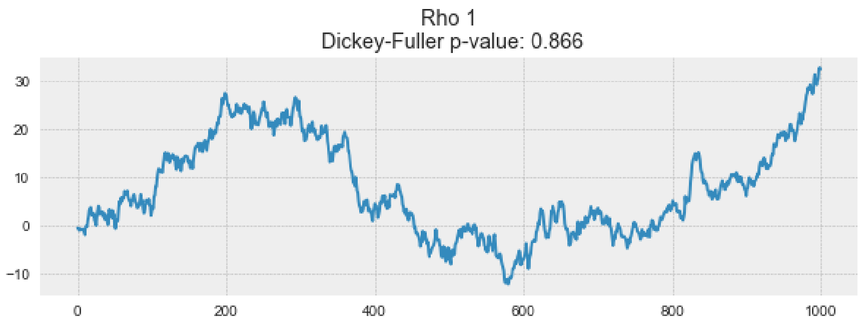
Chart rendering code

```
def plotProcess(n_samples=1000, rho=0):
    x = w = np.random.normal(size=n_samples)
    for t in range(n_samples):
        x[t] = rho * x[t-1] + w[t]

    with plt.style.context('bmh'):
        plt.figure(figsize=(10, 3))
        plt.plot(x)
```


Rho 0
Dickey-Fuller p-value: 0.0


Rho 0.6
Dickey-Fuller p-value: 0.0


Rho 0.9
Dickey-Fuller p-value: 0.0



On the first chart you can see the same stationary white noise you've seen before. On the second one the value of ρρ increased to 0.6, as a result wider cycles appeared on the chart but overall it is still stationary. The third chart deviates even more from the 0 mean but still oscillates around it. Finally, the value of ρ equal to 1 gives us a random walk process—non-stationary time series.

This happens because after reaching the critical value the series x(t)=ρ*x(t−1)+e(t) does not return to its mean value. If we subtract x(t−1) from the left and the right side we will get x(t)−x(t−1)=(ρ−1)*x(t−1)+e(t), where the expression on the left is called the first difference. If ρ=1 then the first difference gives us stationary white noise e(t). This fact is the main idea of the Dickey-Fuller test for the stationarity of time series (presence of a unit root). If we can get stationary series from non-stationary using the first difference we call those series integrated of order 1. Null hypothesis of the test—time series is non-stationary, was rejected on the first three charts and was accepted on the last one. We've got to say that the first difference is not always enough to get stationary series as the process might be integrated of order d, d > 1 (and have multiple unit roots), in such cases the augmented Dickey-Fuller test is used that checks multiple lags at once.

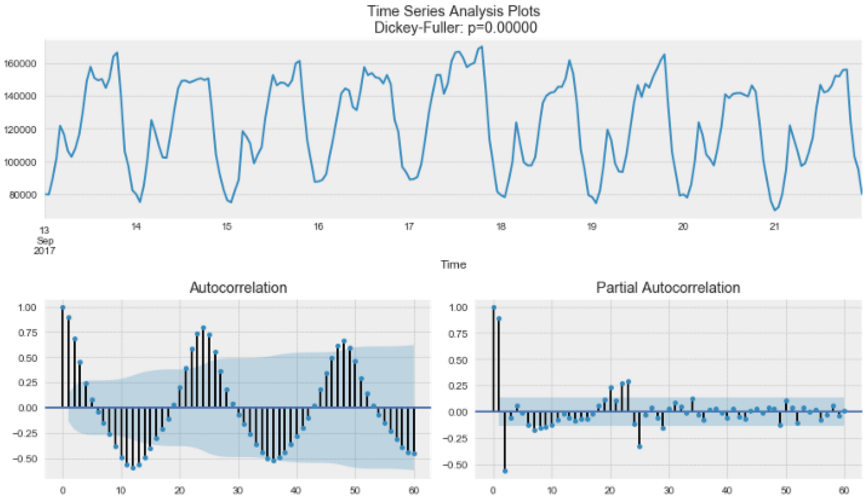
We can fight non-stationarity using different approaches—various order differences, trend and seasonality removal, smoothing, also using transformations like Box-Cox or logarithmic.

Getting rid of non-stationarity and building SARIMA

Now let's build an ARIMA model by walking through all the circles of hell stages of making series stationary.

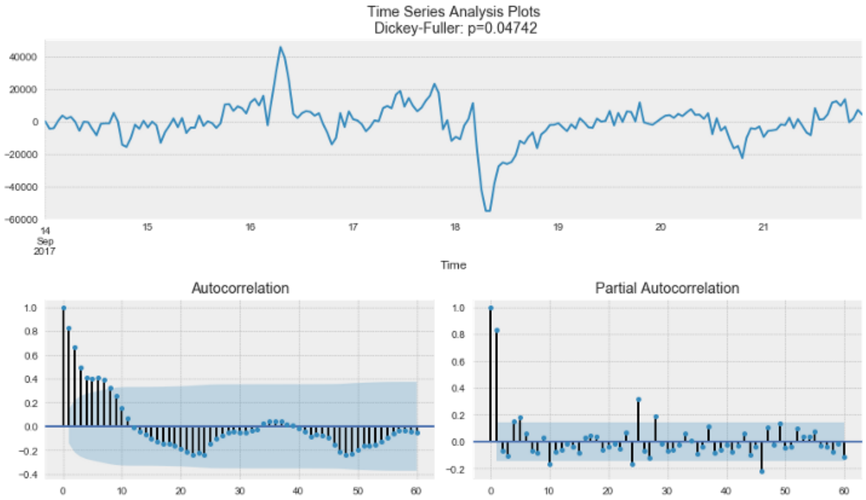
Chart rendering code

```
def tsplot(y, lags=None, figsize=(12, 7), style='bmh'):
    """
        Plot time series, its ACF and PACF, calculate Dick

        y - timeseries
        lags - how many lags to include in ACF, PACF calcu
    """
    if not isinstance(y, pd.Series):
        y = pd.Series(y)

    with plt.style.context(style):
        fig = plt.figure(figsize=figsize)
        layout = (2, 2)
        ts_ax = plt.subplot2grid(layout, (0, 0), colspan=2
        acf_ax = plt.subplot2grid(layout, (1, 0))
```



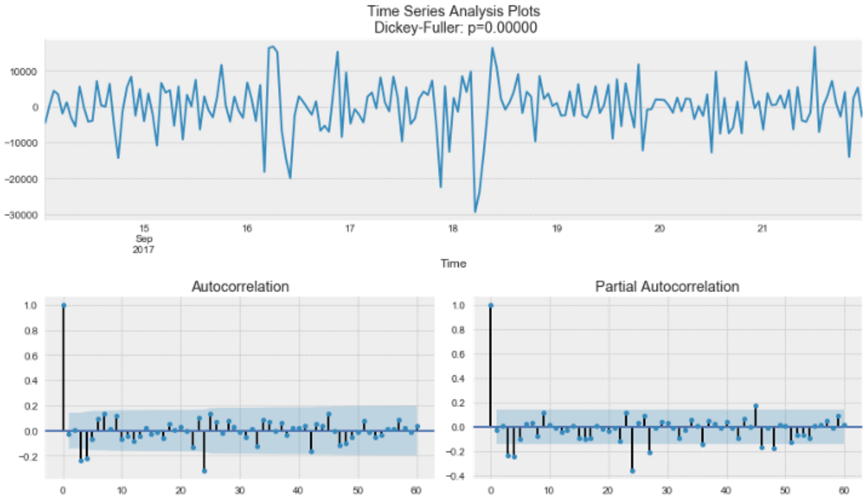
Surprisingly, initial series are stationary, Dickey-Fuller test rejected null hypothesis that a unit root is present. Actually, it can be seen on the plot itself—we don't have a visible trend, so mean is constant, variance is pretty much stable throughout the series. The only thing left is seasonality which we have to deal with before modelling. To do so let's take "seasonal difference" which means a simple subtraction of series from itself with a lag that equals the seasonal period.

```
ads_diff = ads.Ads - ads.Ads.shift(24)
tsplot(ads_diff[24:], lags=60)
```



That’s better, visible seasonality is gone, however autocorrelation function still has too many significant lags. To remove them we’ll take first differences—subtraction of series from itself with lag 1

```
ads_diff = ads_diff - ads_diff.shift(1)
tsplot(ads_diff[24+1:], lags=60)
```



Perfect! Our series now look like something undescribable, oscillating around zero, Dickey-Fuller indicates that it’s stationary and the number of significant peaks in ACF has dropped. We can finally start modelling!

ARIMA-family Crash-Course

A few words about the model. Letter by letter we'll build the full name —**SARIMA(*p,d,q*)(*P,D,Q,s*)**, Seasonal Autoregression Moving Average model:

- **AR(*p*)**—autoregression model, i.e., regression of the time series onto itself. Basic assumption—current series values depend on its previous values with some lag (or several lags). The maximum lag in the model is referred to as ***p***. To determine the initial ***p*** you need to have a look at PACF plot—find the biggest significant lag, after which **most** other lags are becoming not significant.

- **MA(*q*)**—moving average model. Without going into detail it models the error of the time series, again the assumption is—current error depends on the previous with some lag, which is referred to as ***q***. Initial value can be found on ACF plot with the same logic.

Let's have a small break and combine the first 4 letters:

AR(*p*) + MA(*q*) = ARMA(*p,q*)

What we have here is the Autoregressive–moving-average model! If the series is stationary, it can be approximated with those 4 letters. Shall we continue?

- **I(*d*)**— order of integration. It is simply the number of nonseasonal differences needed for making the series stationary. In our case it's just 1, because we used first differences.

Adding this letter to four previous gives us **ARIMA** model which knows how to handle non-stationary data with the help of nonseasonal differences. Awesome, last letter left!

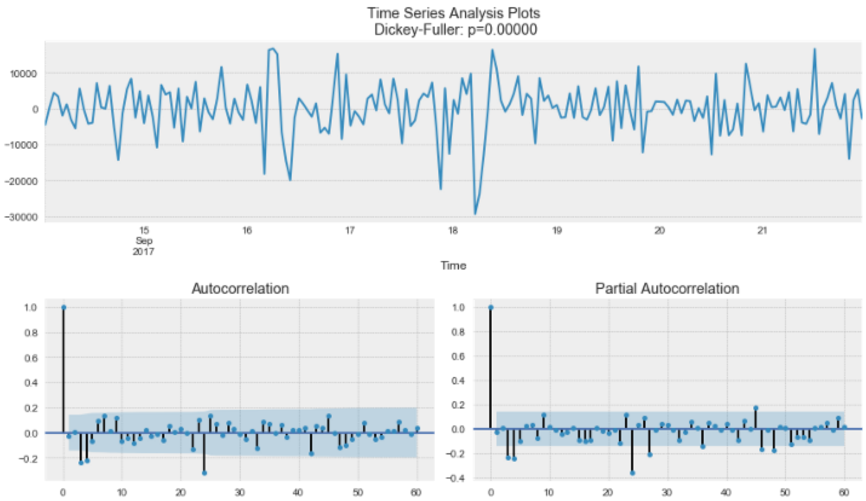
- **S(*s*)**—this letter is responsible for seasonality and equals the season period length of the series

After attaching the last letter we find out that instead of one additional parameter we get three in a row—***(P,D,Q)***

- ***P***—order of autoregression for seasonal component of the model, again can be derived from PACF, but this time you need to look at the number of significant lags, which are the multiples of the season period length, for example, if the period equals 24 and looking at PACF we see 24-th and 48-th lags are significant, that means initial ***P*** should be 2.
- ***Q***—same logic, but for the moving average model of the seasonal component, use ACF plot
- ***D***—order of seasonal integration. Can be equal to 1 or 0, depending on whether seasonal differences were applied or not

Now, knowing how to set initial parameters, let's have a look at the final plot once again and set the parameters:

```
tsplot(ads_diff[24+1:], lags=60)
```



- ***p*** is most probably 4, since it's the last significant lag on PACF after which most others are becoming not significant.
- ***d*** just equals 1, because we had first differences
- ***q*** should be somewhere around 4 as well as seen on ACF
- ***P*** might be 2, since 24-th and 48-th lags are somewhat significant on PACF
- ***D*** again equals 1—we performed seasonal differentiation
- ***Q*** is probably 1, 24-th lag on ACF is significant, while 48-th is not

Now we want to test various models and see which one is better

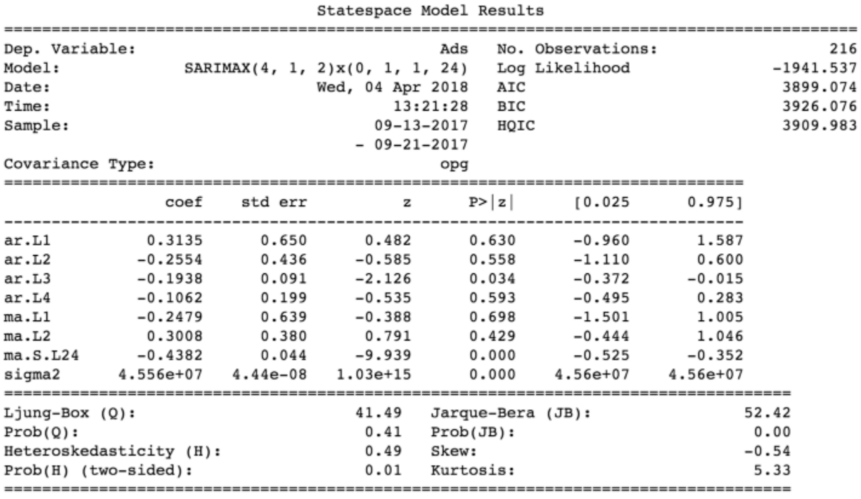
```
# setting initial values and some bounds for them
ps = range(2, 5)
d=1
qs = range(2, 5)
Ps = range(0, 3)
D=1
Qs = range(0, 2)
s = 24 # season length is still 24
```

```
Out: 54
```

```
def optimizeSARIMA(parameters_list, d, D, s):
    """
        Return dataframe with parameters and corresponding

        parameters_list - list with (p, q, P, Q) tuples
        d - integration order in ARIMA model
        D - seasonal integration order
        s - length of season
    """

    results = []
    best_aic = float("inf")

    for param in tqdm_notebook(parameters_list):
        # we need try-except because on some combinations
        try:
            model=sm.tsa.statespace.SARIMAX(ads.Ads, order
                                            seasonal_order
        except:
            continue
        aic = model.aic
        # saving best model, AIC and parameters
        if aic < best_aic:
```

```
CPU times: user 41.3 s, sys: 2.76 s, total: 44.1 s
Wall time: 37.4 s
```

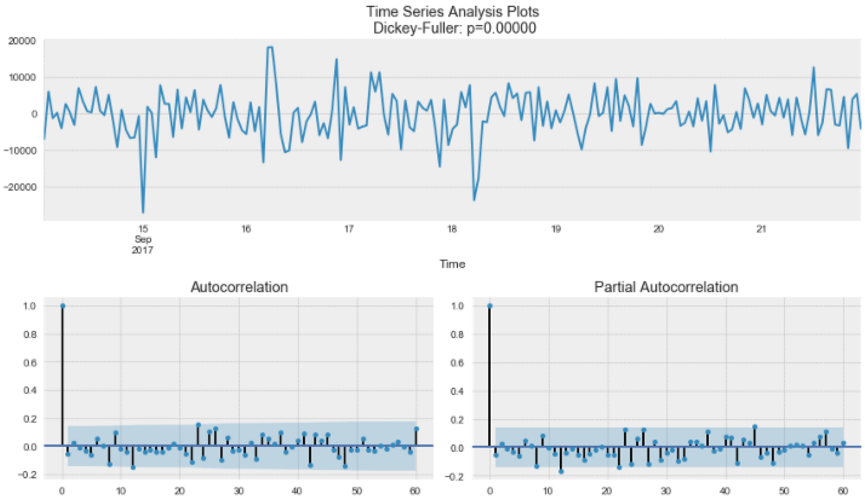
```
# set the parameters that give the lowest AIC
p, q, P, Q = result_table.parameters[0]

best_model=sm.tsa.statespace.SARIMAX(ads.Ads, order=(p, d,
```



```
                                 Statespace Model Results
==========================================================================================
Dep. Variable:                                Ads   No. Observations:                  216
Model:             SARIMAX(4, 1, 2)x(0, 1, 1, 24)   Log Likelihood               -1941.537
Date:                            Wed, 04 Apr 2018   AIC                           3899.074
Time:                                    13:21:28   BIC                           3926.076
Sample:                                09-13-2017   HQIC                          3909.983
                                     - 09-21-2017
Covariance Type:                              opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.3135      0.650      0.482      0.630      -0.960       1.587
ar.L2         -0.2554      0.436     -0.585      0.558      -1.110       0.600
ar.L3         -0.1938      0.091     -2.126      0.034      -0.372      -0.015
ar.L4         -0.1062      0.199     -0.535      0.593      -0.495       0.283
ma.L1         -0.2479      0.639     -0.388      0.698      -1.501       1.005
ma.L2          0.3008      0.380      0.791      0.429      -0.444       1.046
ma.S.L24      -0.4382      0.044     -9.939      0.000      -0.525      -0.352
sigma2      4.556e+07   4.44e-08   1.03e+15      0.000    4.56e+07    4.56e+07
===================================================================================
Ljung-Box (Q):                       41.49   Jarque-Bera (JB):                52.42
Prob(Q):                              0.41   Prob(JB):                         0.00
Heteroskedasticity (H):               0.49   Skew:                            -0.54
Prob(H) (two-sided):                  0.01   Kurtosis:                         5.33
===================================================================================
```

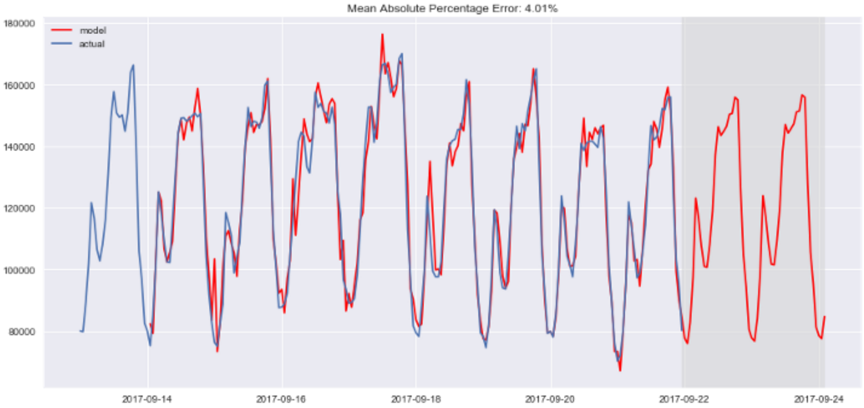
Let’s inspect the residuals of the model

`tsplot(best_model.resid[24+1:], lags=60)`



Well, it’s clear that the residuals are stationary, there are no apparent autocorrelations, let’s make predictions using our model

```
def plotSARIMA(series, model, n_steps):
    """
        Plots model vs predicted values

        series - dataset with timeseries
        model - fitted SARIMA model
        n_steps - number of steps to predict in the future

    """
    # adding model values
    data = series.copy()
    data.columns = ['actual']
    data['arima_model'] = model.fittedvalues
    # making a shift on s+d steps, because these values we
    # due to the differentiating
    data['arima_model'][:s+d] = np.NaN

    # forecasting on n_steps forward
    forecast = model.predict(start = data.shape[0], end =
```



In the end we got quite adequate predictions, our model on average was wrong by 4.01%, which is very very good, but overall costs of preparing data, making series stationary and brute-force parameters selecting might not be worth this accuracy.

Linear (and not quite) models on time series

Small lyrical digression again. Often in my job I have to build models with the only principle guiding me known as *fast, good, cheap*. That means some of the models will never be "production ready" as they demand too much time for the data preparation (for example, SARIMA), or require frequent re-training on new data (again, SARIMA), or are difficult to tune (good example—SARIMA), so it's very often much easier to select a couple of features from the existing time series and build a simple linear regression or, say, a random forest. Good and cheap.

Maybe this approach is not backed up by theory, breaks different assumptions (like, Gauss-Markov theorem, especially about the errors being uncorrelated), but it's very useful in practice and quite frequently used in machine learning competitions.

Feature exctraction

Alright, model needs features and all we have is a 1-dimentional time series to work with. What features can we exctract?

Lags of time series, of course

Window statistics:

- Max/min value of series in a window
- Average/median value in a window
- Window variance
- etc.

Date and time features:

- Minute of an hour, hour of a day, day of the week, you get it
- Is this day a holiday? Maybe something special happened? Make it a boolean feature

Target encoding

Forecasts from other models (though we can lose the speed of prediction this way)

Let's run through some of the methods and see what we can extract from our ads series

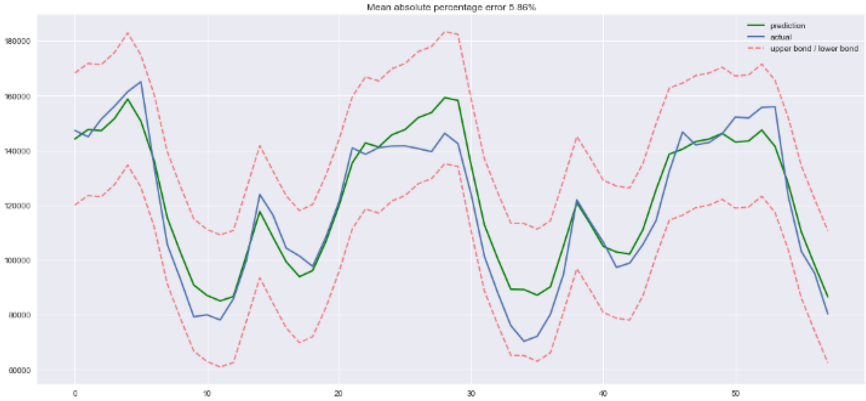
Lags of time series

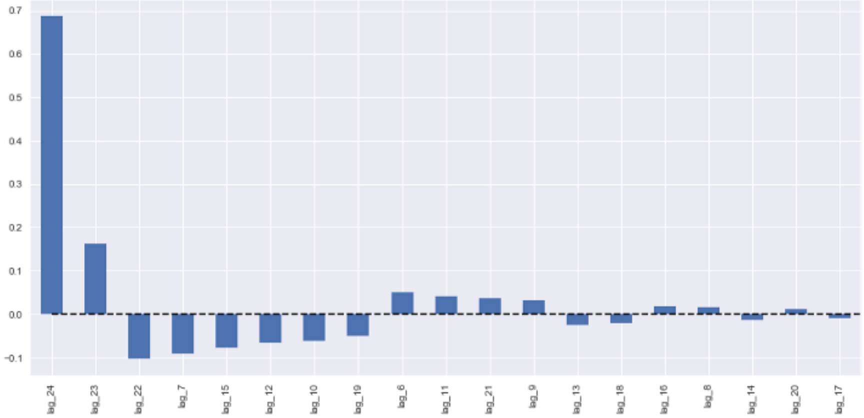
Shifting the series ***n*** steps back we get a feature column where the current value of time series is aligned with its value at the time ***t−n***. If we make a 1 lag shift and train a model on that feature, the model will be able to forecast 1 step ahead having observed current state of the series. Increasing the lag, say, up to 6 will allow the model to make predictions 6 steps ahead, however it will use data, observed 6 steps back. If something fundamentally changes the series during that unobserved period, the model will not catch the changes and will return forecasts with big error. So, during the initial lag selection one has to find a balance between the optimal prediction quality and the length of forecasting horizon.

```
# Creating a copy of the initial datagrame to make various
data = pd.DataFrame(ads.Ads.copy())
data.columns = ["y"]

# Adding the lag of the target variable from 6 steps back u
```

Wonderful, we got ourselves a dataset here, why don't we train a model?

```python
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

# for time-series cross-validation set 5 folds
tscv = TimeSeriesSplit(n_splits=5)

def timeseries_train_test_split(X, y, test_size):
    """
        Perform train-test split with respect to time seri
    """

    # get the index after which test set starts
    test_index = int(len(X)*(1-test_size))

    X_train = X.iloc[:test_index]
    y_train = y.iloc[:test_index]
    X_test = X.iloc[test_index:]
    y_test = y.iloc[test_index:]

    return X_train, X_test, y_train, y_test

def plotModelResults(model, X_train=X_train, X_test=X_test
    """
        Plots modelled vs fact values, prediction interval

    """

    prediction = model.predict(X_test)

    plt.figure(figsize=(15, 7))
    plt.plot(prediction, "g", label="prediction", linewidt
    plt.plot(y_test.values, label="actual", linewidth=2.0)

    if plot_intervals:
        cv = cross_val_score(model, X_train, y_train,
                                    cv=tscv,
                                    scoring="neg_mean_abso
        mae = cv.mean() * (-1)
        deviation = cv.std()

        scale = 1.96
        lower = prediction - (mae + scale * deviation)
        lower = prediction - (mae + scale * deviation)
        upper = prediction + (mae + scale * deviation)

        plt.plot(lower, "r--", label="upper bond / lower b
        plt.plot(upper, "r--", alpha=0.5)

        if plot_anomalies:
            anomalies = np.array([np.NaN]*len(y_test))
            anomalies[y_test<lower] = y_test[y_test<lower]
            anomalies[y_test>upper] = y_test[y_test>upper]
```





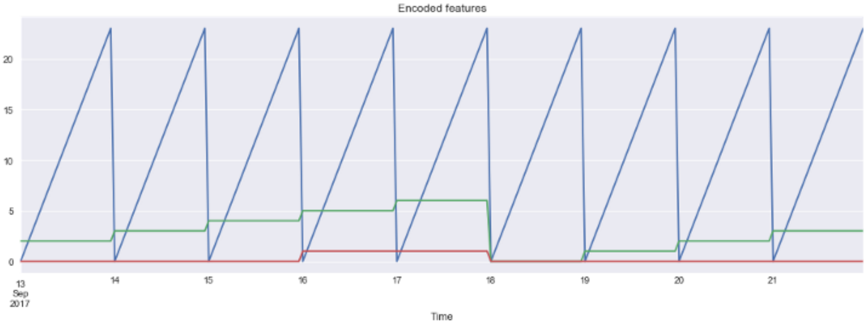
Well, simple lags and linear regression gave us predictions that are not that far from SARIMA in quality. There are lot’s of unnecessary features, but we’ll do feature selection a bit later. Now let’s continue engineering!

We’ll add into our dataset hour, day of the week and boolean for the weekend. To do so we need to transform current dataframe index into `datetime` format and exctract `hour` and `weekday` out of it.

```
data.index = data.index.to_datetime()
data["hour"] = data.index.hour
data["weekday"] = data.index.weekday
```

We can visualize the resulting features

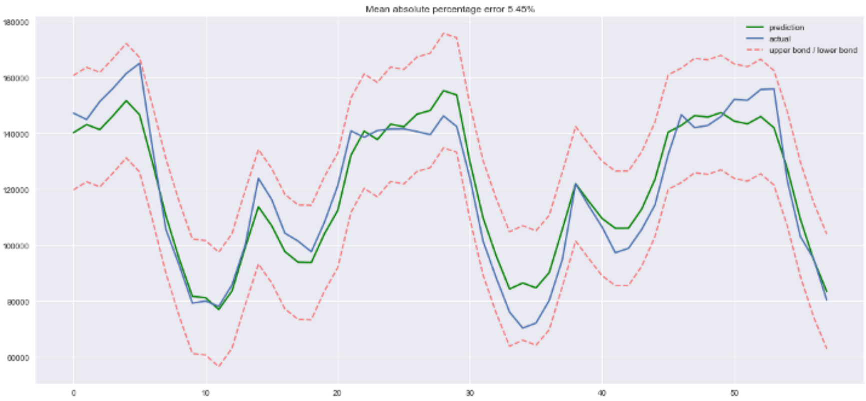
```
plt.figure(figsize=(16, 5))
plt.title("Encoded features")
data.hour.plot()
data.weekday.plot()
```

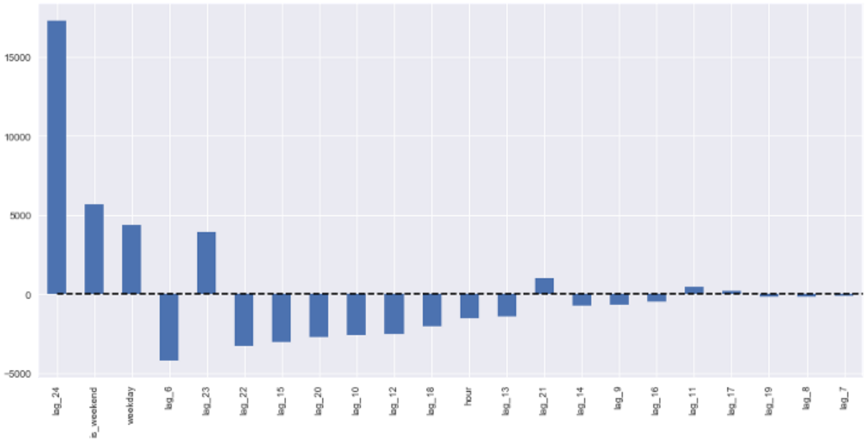


Blue spiky line — hour feature, green ladder — weekday, red bump — weekends!

Since now we have different scales of variables — thousands for lag features and tens for categorical, it's reasonable to transform them into same scale to continue exploring feature importances and later — regularization.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

y = data.dropna().y
X = data.dropna().drop(['y'], axis=1)

X_train, X_test, y_train, y_test = timeseries_train_test_s

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```





Test error goes down a little bit and judging by the coefficients plot we can say that `weekday` and `is_weekend` are rather useful features

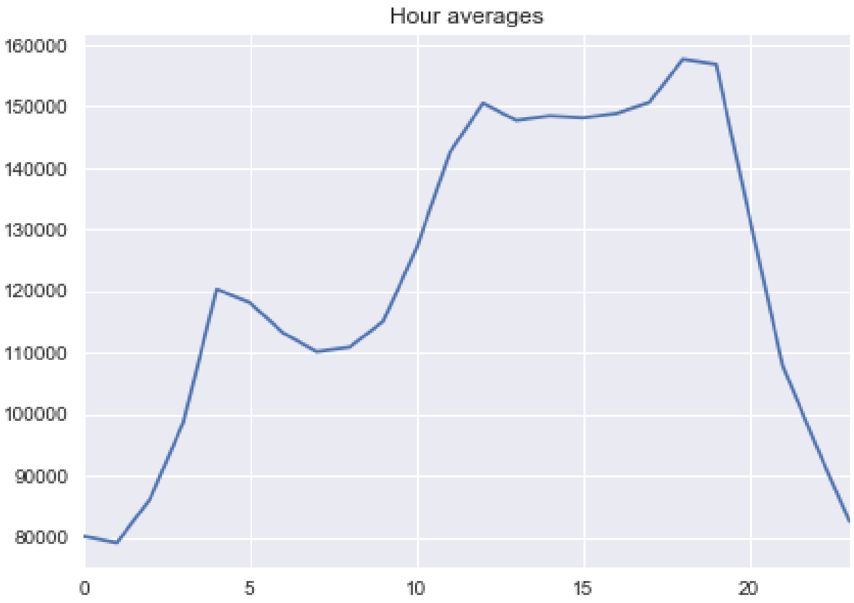
Target encoding

I’d like to add another variant of encoding categorical variables—by mean value. If it’s undesirable to explode dataset by using tons of dummy variables that can lead to the loss of information about the distance, and if they can’t be used as real values because of the conflicts like “0 hours < 23 hours”, then it’s possible to encode a variable with a little bit more interpretable values. Natural idea is to encode with the mean value of the target variable. In our example every day of the week and every hour of the day can be encoded by the corresponding average number of ads watched during that day or hour. It’s very important to make sure that the mean value is calculated over train set only (or over current cross-validation fold only), so that the model is not aware of the future.

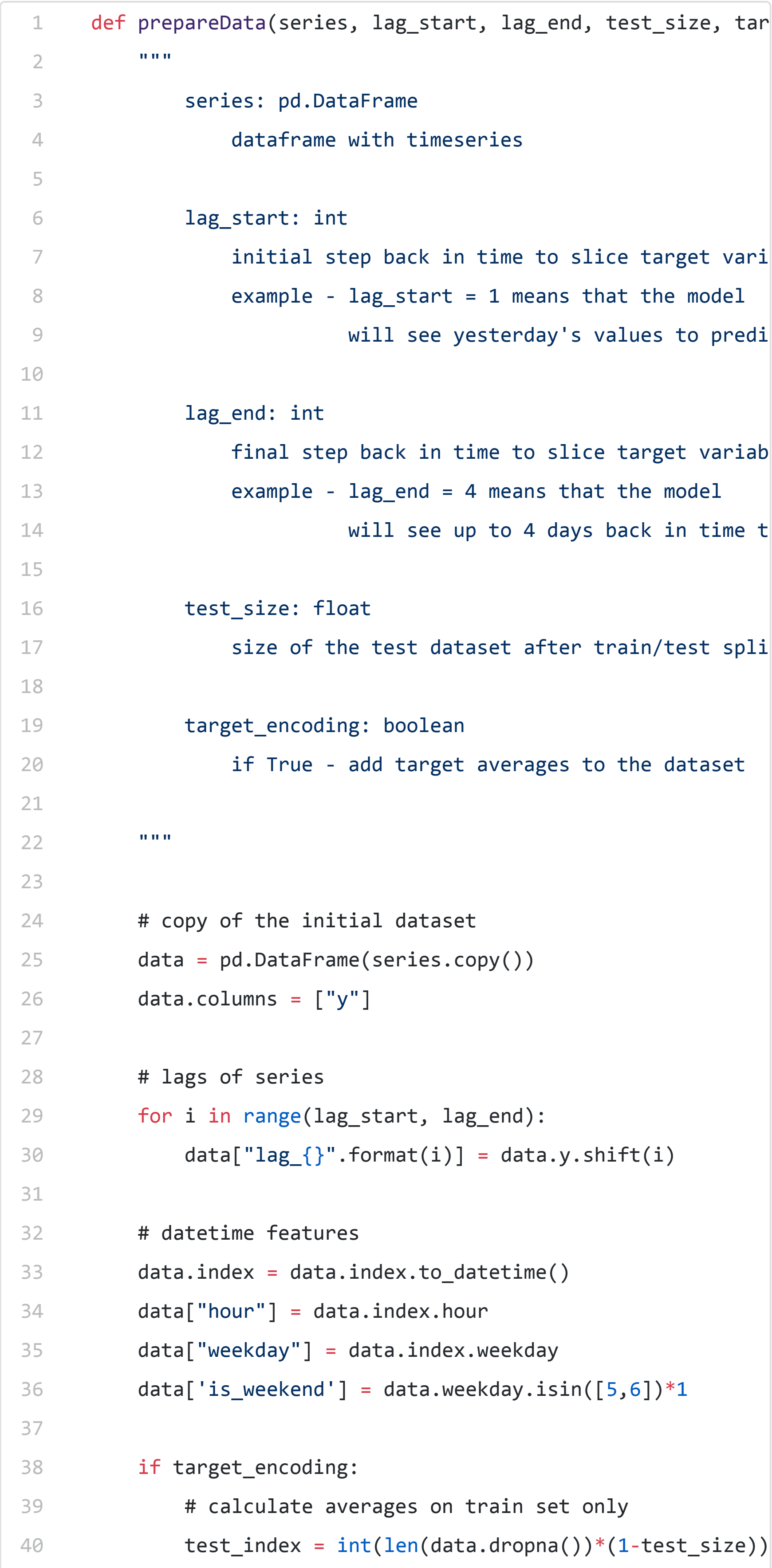
```
def code_mean(data, cat_feature, real_feature):
    """
    Returns a dictionary where keys are unique categories o
    and values are means over real_feature
    """
```

Let’s have a look at hour averages

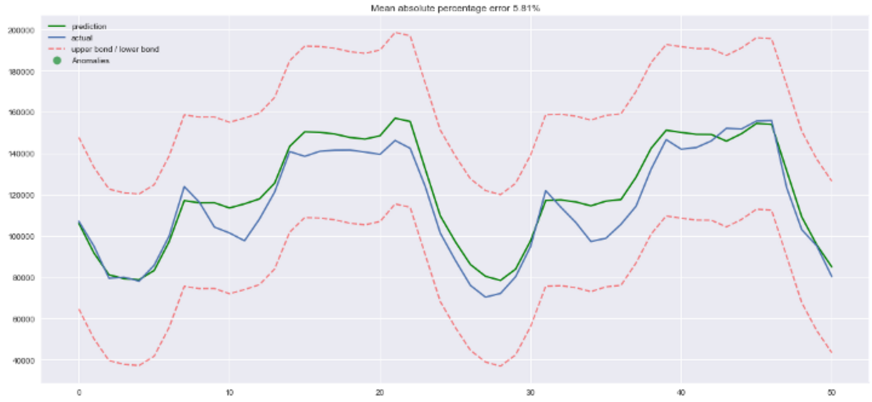
```
average_hour = code_mean(data, 'hour', "y")
plt.figure(figsize=(7, 5))
plt.title("Hour averages")
pd.DataFrame.from_dict(average_hour, orient='index')[0].plo
```

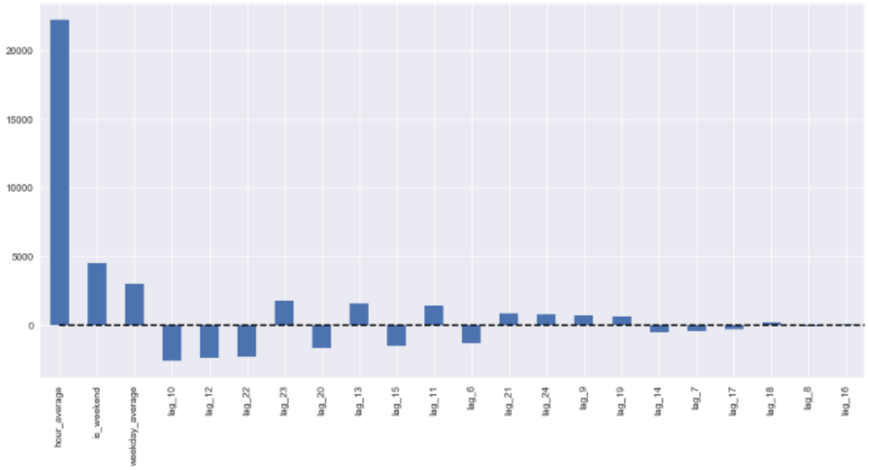


Finally, put all the transformations together in a single function



```
def prepareData(series, lag_start, lag_end, test_size, tar
    """
        series: pd.DataFrame
            dataframe with timeseries

        lag_start: int
            initial step back in time to slice target vari
            example - lag_start = 1 means that the model
                      will see yesterday's values to predi

        lag_end: int
            final step back in time to slice target variab
            example - lag_end = 4 means that the model
                      will see up to 4 days back in time t

        test_size: float
            size of the test dataset after train/test spli

        target_encoding: boolean
            if True - add target averages to the dataset

    """

    # copy of the initial dataset
    data = pd.DataFrame(series.copy())
    data.columns = ["y"]

    # lags of series
    for i in range(lag_start, lag_end):
        data["lag_{}".format(i)] = data.y.shift(i)

    # datetime features
    data.index = data.index.to_datetime()
    data["hour"] = data.index.hour
    data["weekday"] = data.index.weekday
    data['is_weekend'] = data.weekday.isin([5,6])*1

    if target_encoding:
        # calculate averages on train set only
        test_index = int(len(data.dropna())*(1-test_size))
```





Here comes **overfitting**! `Hour_average` variable was so great on train dataset that the model decided to concentrate all its forces on it - as a result the quality of prediction dropped. This problem can be approached in a variety of ways, for example, we can calculate target encoding not for the whole train set, but for some window instead, that way encodings from the last observed window will probably describe current series state better. Or we can just drop it manually, since we're sure here it makes things only worse.

```
X_train, X_test, y_train, y_test =\
prepareData(ads.Ads, lag_start=6, lag_end=25, test_size=0.3

X_train_scaled = scaler.fit_transform(X_train)
```

Regularization and feature selection

As we already know, not all features are equally healthy, some may lead to overfitting and should be removed. Besides manual inspecting we can apply regularization. Two most popular regression models with regularization are Ridge and Lasso regressions. They both add some more constrains to our loss function.

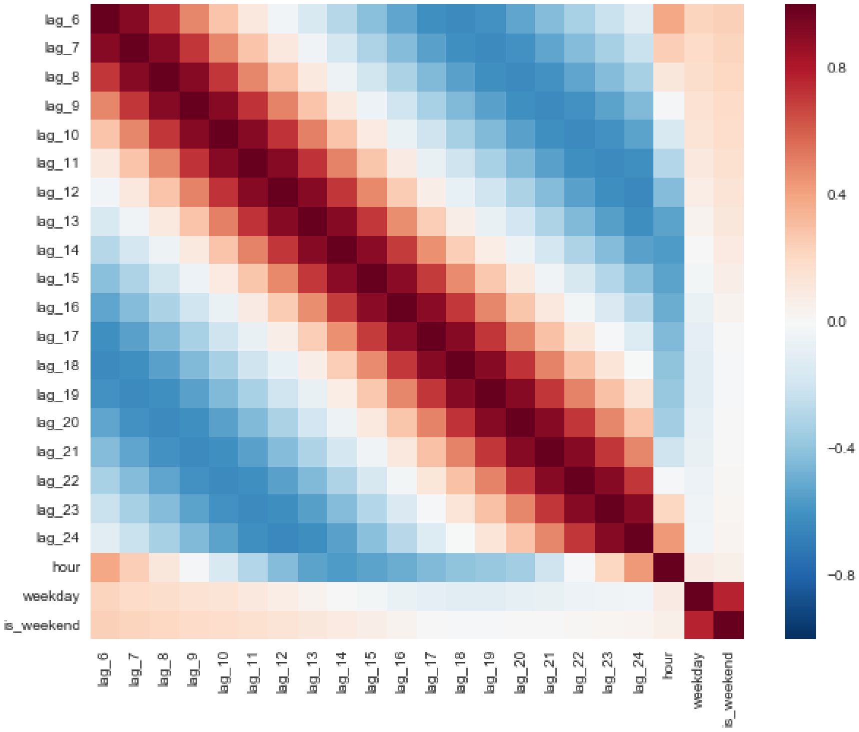
In case of ***Ridge regression***—those constrains are the sum of squares of coefficients, multiplied by the regularization coefficient. I.e. the bigger coefficient feature has—the bigger our loss will be, hence we will try to optimize the model while keeping coefficients fairly low.

As a result of such regularization which has a proud name ***L2*** we'll have higher bias and lower variance, so the model will generalize better (at least that's what we hope will happen).

Second model—***Lasso regression***, here we add to the loss function not squares but absolute values of the coefficients, as a result during the optimization process coefficients of unimportant features may become zeroes, so Lasso regression allows for automated feature selection. This regularization type is called ***L1***.

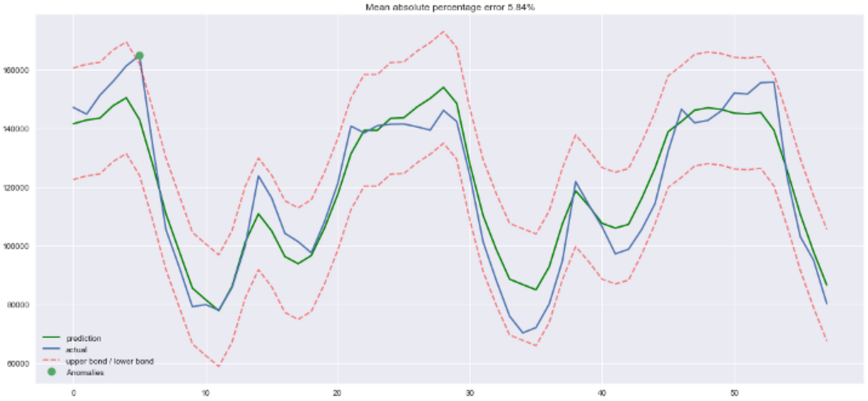
First, make sure we have things to drop and data truly has highly correlated features

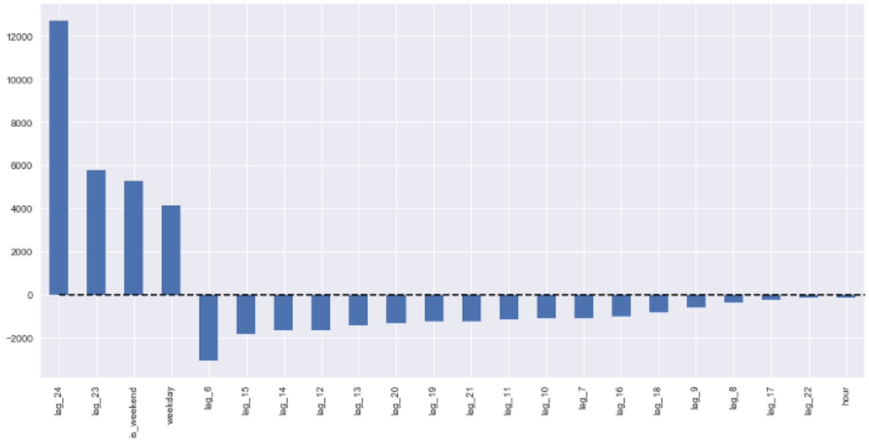
```
plt.figure(figsize=(10, 8))
sns.heatmap(X_train.corr());
```



Prettier than some modern art

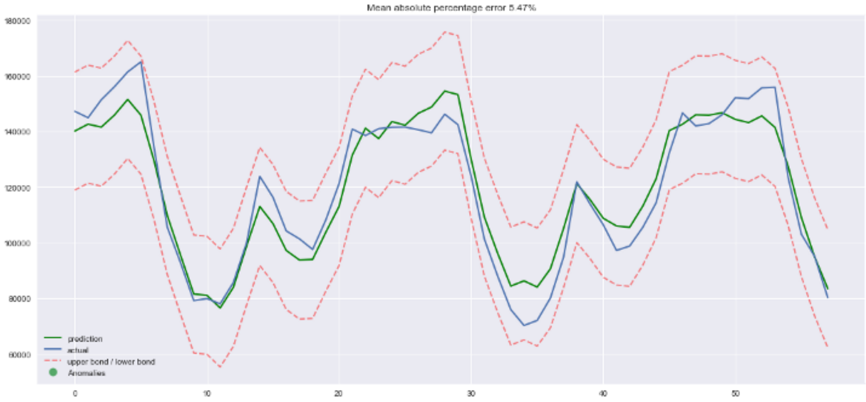
```
from sklearn.linear_model import LassoCV, RidgeCV

ridge = RidgeCV(cv=tscv)
ridge.fit(X_train_scaled, y_train)

plotModelResults(ridge,
                 X_train=X_train_scaled,
```

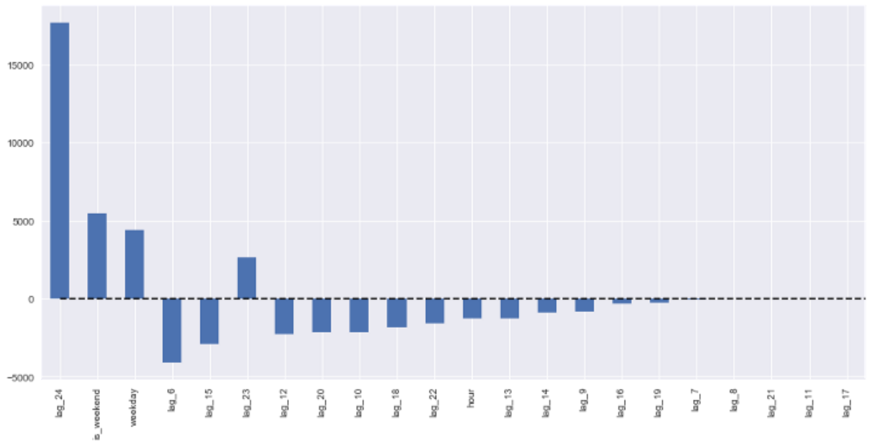




We can clearly see how coefficients are getting closer and closer to zero (thought never actually reach it) as their importance in the model drops

```
lasso = LassoCV(cv=tscv)
lasso.fit(X_train_scaled, y_train)

plotModelResults(lasso,
                 X_train=X_train_scaled,
                 X_test=X_test_scaled
```



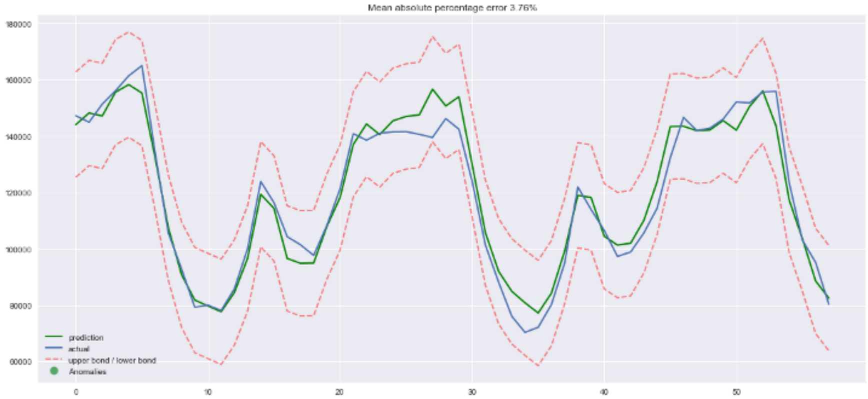


Lasso regression turned out to be more conservative and removed 23-rd lag from most important features (and also dropped 5 features completely) which only made the quality of prediction better.

Boosting

Why not try XGBoost now?

```
from xgboost import XGBRegressor

xgb = XGBRegressor()
xgb.fit(X_train_scaled, y_train)

plotModelResults(xgb,
```



Here is the winner! The smallest error on the test set among all the models we've tried so far.

Yet this victory is decieving and it might not be the brightest idea to fit xgboost as soon as you get your hands over time series data. Generally tree-based models poorly handle trends in data, compared to linear models, so you have to detrend your series first or use some tricks to make the magic happen. Ideally—make the series stationary and then use XGBoost, for example, you can forecast trend separately with a linear model and then add predictions from xgboost to get final forecast.

Conclusion

We got acquainted with different time series analysis and prediction methods and approaches. Unfortunately, or maybe luckily, there's no silver bullet to solve this kind of problems. Methods developed in the 60s of the last century (and some even in the beginning of the XIX century) are still popular along with the LSTM and RNN (not covered in this article). Partially this is related to the fact that the prediction task as any other data related task is creative in so many aspects and definitely requires research. In spite of the large number of formal quality metrics and approaches to parameters estimation, it's often required to seek and try something different for each time series. Last but not least the balance between quality and cost is important. As a good example SARIMA model mentioned here not once or twice can produce spectacular results after due tuning but might require many hours of tambourine dancing time series manipulation, as in the same time simple linear regression model can be build in 10 minutes giving more or less comparable results.

Assignment #9

Full versions of assignments are announced each week in a new run of the course (October 1, 2018). Meanwhile, you can practice with a demo version: Kaggle Kernel, nbviewer.

Useful resources

- Online textbook of the advanced statistical forecasting course of the Duke University—covers in details various smoothing techniques, linear and ARIMA models
- Comparison of ARIMA and Random Forest time series models for prediction of avian influenza H5N1 outbreaks—one of a few where random forest applicability to the tasks of time series forecasting is actively defended
- Time Series Analysis (TSA) in Python—Linear Models to GARCH ARIMA models family and their applicability to the task of modeling financial indicators (Brian Christopher)

. . .

Author: Dmitry Sergeyev. Translated and edited by Borys Zibrov, and Yuanyuan Pao.