**Предварительный анализ прогнозирования предлагаемого времнного ряда**

* **План анализа:**

1. Движемся, сглаживаем и оцениваем
   * Rolling window estimations
   * Экспоненциальное сглаживание, модель Хольта-Винтерса
   * Кросс-валидация на временных рядах, подбор параметров
2. Эконометрический подход
   * Стационарность, единичные корни
   * Избавляемся от нестационарности и строим SARIMA
3. Линейные и не очень модели на временных рядах
   * Извлечение признаков (Feature extraction)
   * Линейная регрессия.

При аналиче временных рядов чаще всего возникает вопрос - а что у нас будет происходить с нашими показателями в ближайший день/неделю/месяц/пр. - как много действий совершат пользователи, и так далее. К задаче прогнозирования можно подходить по-разному, в зависимости от того, какого качества должен быть прогноз, на какой период мы хотим его строить, и, конечно, как долго нужно подбирать и настраивать параметры модели для его получения.

Начнем с простых методов анализа и прогнозирования - скользящих средних, сглаживаний и их вариаций.

* **Движемся, сглаживаем и оцениваем**

Небольшое [определение](https://ru.wikipedia.org/wiki/%D0%92%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D1%80%D1%8F%D0%B4) временного ряда:

Временной ряд – это последовательность значений, описывающих протекающий во времени процесс, измеренных в последовательные моменты времени, обычно через равные промежутки

Таким образом, данные оказываются упорядочены относительно неслучайных моментов времени, и, значит, в отличие от случайных выборок, могут содержать в себе дополнительную информацию, которую мы постараемся извлечь.

Импортируем нужные библиотеки. В основном нам понадобится модуль [statsmodels](http://statsmodels.sourceforge.net/stable/), в котором реализованы многочисленные методы статистического моделирования, в том числе для временных рядов. Для поклонников R, пересевших на питон, он может показаться очень родным, так как поддерживает написание формулировок моделей в стиле 'Wage ~ Age + Education'.

In [1]:

*#!pip install arch*

In [2]:

**import** **sys**

**import** **warnings**

warnings.filterwarnings('ignore')

**from** **tqdm** **import** tqdm

**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

**from** **sklearn.metrics** **import** mean\_absolute\_error, mean\_squared\_error

**import** **statsmodels.formula.api** **as** **smf**

**import** **statsmodels.tsa.api** **as** **smt**

**import** **statsmodels.api** **as** **sm**

**import** **scipy.stats** **as** **scs**

**from** **scipy.optimize** **import** minimize

**from** **arch** **import** arch\_model

**import** **matplotlib.pyplot** **as** **plt**

**import** **matplotlib** **as** **mpl**

**import** **matplotlib.dates** **as** **mdates**

%matplotlib inline

**from** **plotly** **import** \_\_version\_\_

**from** **plotly.offline** **import** download\_plotlyjs, init\_notebook\_mode, plot, iplot

**from** **plotly** **import** graph\_objs **as** go

init\_notebook\_mode(connected = **True**)

**def** plotly\_df(df, title = ''):

data = []

**for** column **in** df.columns:

trace = go.Scatter(

x = df.index,

y = df[column],

mode = 'lines',

name = column

)

data.append(trace)

layout = dict(title = title)

fig = dict(data = data, layout = layout)

iplot(fig, show\_link=**False**)

Возьмем реальные данные из предлагаемого временного ряда

In [42]:

dataset = pd.read\_csv('./cll.csv', index\_col=['date'], parse\_dates=['date'])

dataset = dataset.query('Bid < 5000')

dataset=dataset[-3000:]

plotly\_df(dataset, title = "Временной ряд торгов")

*#dataset*

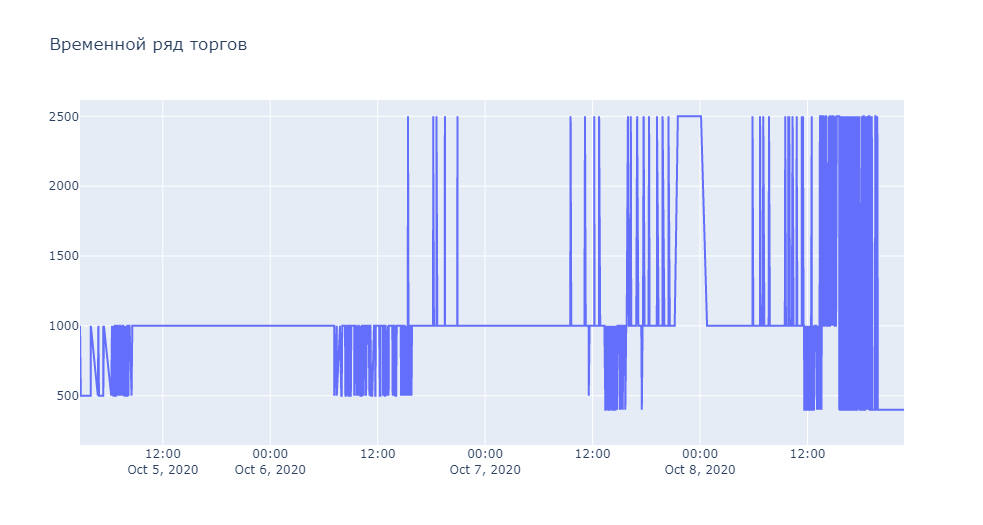


Рис. График анализируемого временного ряда

Начнем моделирование с наивного предположения - "завтра будет, как вчера", но вместо модели вида y^t=yt−1y^t=yt−1 будем считать, что будущее значение переменной зависит от среднего nn её предыдущих значений, а значит, воспользуемся скользящей средней.

y^t=1k∑n=0k−1yt−ny^t=1k∑n=0k−1yt−n

In [4]:

**def** moving\_average(series, n):

**return** np.average(series[-n:])

moving\_average(dataset.Bid, 24) *# посмотрим на прогноз, построенный по последнему наблюдаемому дню (24 часа)*

Out[4]:

400.0

К сожалению, такой прогноз долгосрочным сделать не удастся, для получения предсказания на шаг вперед предыдущее значение должно быть фактически наблюдаемой величиной. Зато у скользящей средней есть другое применение - сглаживание исходного ряда для выявления трендов, в пандасе есть готовая реализация - [DataFrame.rolling(window).mean()](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.rolling.html). Чем больше зададим ширину интервала - тем более сглаженным окажется тренд. В случае, если данные сильно зашумлены, что особенно часто встречается, например, в финансовых показателях, такая процедура может помочь увидеть общие паттерны.

In [5]:

**def** plotMovingAverage(series, n, plot\_bounds=**False**):

*"""*

*series - dataframe with timeseries*

*n - rolling window size*

*plot\_bounds: bool - whether to draw confidence interval*

*"""*

rolling\_mean = series.rolling(window=n).mean()

fig, ax = plt.subplots(figsize=(15,5))

plt.title("Moving average**\n** window size = **{}**".format(n))

plt.plot(rolling\_mean, "g", label="Rolling mean trend")

*# При желании, можно строить и доверительные интервалы для сглаженных значений*

**if** plot\_bounds:

rolling\_std = series.rolling(window=n).std()

upper\_bound = rolling\_mean+1.96\*rolling\_std

lower\_bound = rolling\_mean-1.96\*rolling\_std

plt.plot(upper\_bound, "r--", label="Upper Bound / Lower Bound")

plt.plot(lower\_bound, "r--")

plt.plot(series[n:], label="Actual values")

plt.legend(loc="upper left")

plt.grid(**True**)

*# Деления соответствуют понедельникам что дает представление о недельной цикличности графика*

ax.xaxis.set\_major\_locator(mdates.WeekdayLocator(byweekday=mdates.MONDAY))

*# Отображать значение дат в формате yy-mm-dd*

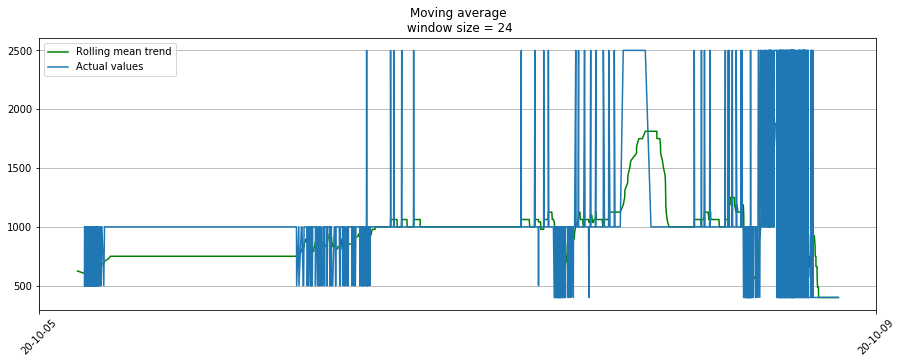
ax.xaxis.set\_major\_formatter(mdates.DateFormatter('%y-%m-**%d**'));

plt.xticks(rotation=45)

In [6]:

plotMovingAverage(dataset, 24) *# сглаживаем по дням*

plotMovingAverage(dataset, 24\*7)*#, plot\_bounds=True) # сглаживаем по неделям*



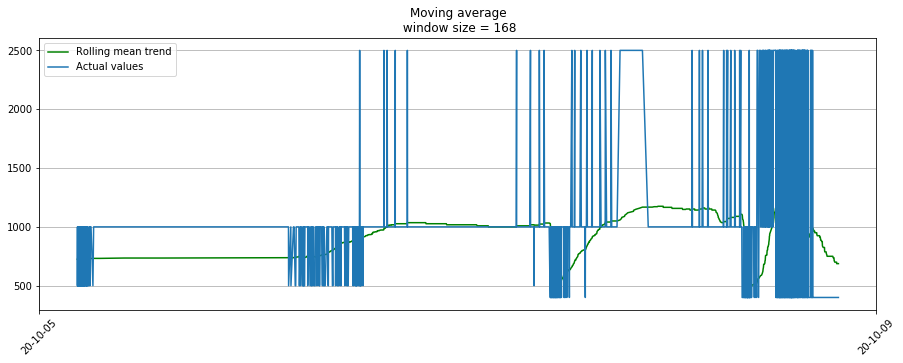


Рис. Графики сглаживания по суткам и неделям

Для нашего ряда тренды и так вполне очевидны, но если сгладить по дням, становится лучше видна динамика онлайна по будням и выходным, а недельное сглаживание хорошо отражает общие изменения, связанные с резким ростом числа кликов.

Модификацией простой скользящей средней является взвешенная средняя, внутри которой наблюдениям придаются различные веса, в суме дающие единицу, при этом обычно последним наблюдениям присваивается больший вес.

y^t=∑n=1kωnyt+1−ny^t=∑n=1kωnyt+1−n

In [7]:

**def** weighted\_average(series, weights):

result = 0.0

weights.reverse()

**for** n **in** range(len(weights)):

result += series[-n-1] \* weights[n]

**return** result

In [8]:

dataset.Users=dataset.Bid

In [9]:

weighted\_average(dataset.Bid, [0.6, 0.2, 0.1, 0.07, 0.03])

Out[9]:

400.0

**Экспоненциальное сглаживание**

А теперь посмотрим, что произойдёт, если вместо взвешивания последних nn значений ряда мы начнем взвешивать все доступные наблюдения, при этом экспоненциально уменьшая веса по мере углубления в исторические данные. В этом нам поможет формула простого [экспоненциального сглаживания](http://www.machinelearning.ru/wiki/index.php?title=%D0%AD%D0%BA%D1%81%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%86%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%81%D0%B3%D0%BB%D0%B0%D0%B6%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5):

y^t=α⋅yt+(1−α)⋅y^t−1y^t=α⋅yt+(1−α)⋅y^t−1

Здесь модельное значение представляет собой средневзвешенную между текущим истинным и предыдущим модельным значениями. Вес αα называется сглаживающим фактором. Он определяет, как быстро мы будем "забывать" последнее доступное истинное наблюдение. Чем меньше αα, тем больше влияния оказывают предыдущие модельные значения, и тем сильнее сглаживается ряд.

Экспоненциальность скрывается в рекурсивности функции - каждый раз мы умножаем (1−α)(1−α) на предыдущее модельное значение, которое, в свою очередь, также содержало в себе (1−α)(1−α), и так до самого начала.

In [10]:

**def** exponential\_smoothing(series, alpha):

result = [series[0]] *# first value is same as series*

**for** n **in** range(1, len(series)):

result.append(alpha \* series[n] + (1 - alpha) \* result[n-1])

**return** result

In [11]:

**with** plt.style.context('seaborn-white'):

plt.figure(figsize=(20, 8))

**for** alpha **in** [0.3, 0.05]:

plt.plot(exponential\_smoothing(dataset.Bid, alpha), label="Alpha **{}**".format(alpha))

plt.plot(dataset.Bid.values, "c", label = "Actual")

plt.legend(loc="best")

plt.axis('tight')

plt.title("Exponential Smoothing")

plt.grid(**True**)

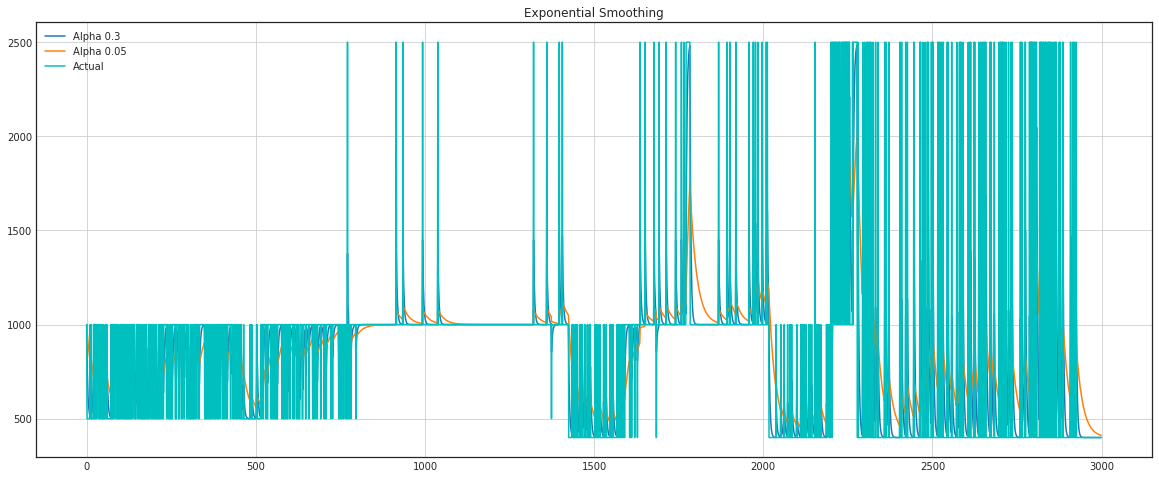


Рис. График экспоненциального сглаживания временного ряда.

Как видим экспоненциальное сглаживание на наших данных тожне не очень хорошо работает.

**Двойное экспоненциальное сглаживание**

Переходим к расширению экспоненциального сглаживания, которое позволит строить прогноз сразу на две точки вперед (и тоже красиво сглаживать ряд).

В этом нам поможет разбиение ряда на две составляющие - уровень (level, intercept) ℓℓ и тренд bb (trend, slope). Уровень, или ожидаемое значение ряда, мы предсказывали при помощи предыдущих методов, а теперь такое же экспоненциальное сглаживание применим к тренду, наивно или не очень полагая, что будущее направление изменения ряда зависит от взвешенных предыдущих изменений.

ℓx=αyx+(1−α)(ℓx−1+bx−1)ℓx=αyx+(1−α)(ℓx−1+bx−1)

bx=β(ℓx−ℓx−1)+(1−β)bx−1bx=β(ℓx−ℓx−1)+(1−β)bx−1

y^x+1=ℓx+bxy^x+1=ℓx+bx

В результате получаем набор функций. Первая описывает уровень - он, как и прежде, зависит от текущего значения ряда, а второе слагаемое теперь разбивается на предыдущее значение уровня и тренда. Вторая отвечает за тренд - он зависит от изменения уровня на текущем шаге, и от предыдущего значения тренда. Здесь в роли веса в экспоненциальном сглаживании выступает коэффициент ββ. Наконец, итоговое предсказание представляет собой сумму модельных значений уровня и тренда.

In [12]:

**def** double\_exponential\_smoothing(series, alpha, beta):

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): *# прогнозируем*

value = result[-1]

**else**:

value = series[n]

last\_level, level = level, alpha\*value + (1-alpha)\*(level+trend)

trend = beta\*(level-last\_level) + (1-beta)\*trend

result.append(level+trend)

**return** result

**with** plt.style.context('seaborn-white'):

plt.figure(figsize=(20, 8))

**for** alpha **in** [0.9, 0.02]:

**for** beta **in** [0.9, 0.02]:

plt.plot(double\_exponential\_smoothing(dataset.Users, alpha, beta), label="Alpha **{}**, beta **{}**".format(alpha, beta))

plt.plot(dataset.Users.values, label = "Actual")

plt.legend(loc="best")

plt.axis('tight')

plt.title("Double Exponential Smoothing")

plt.grid(**True**)

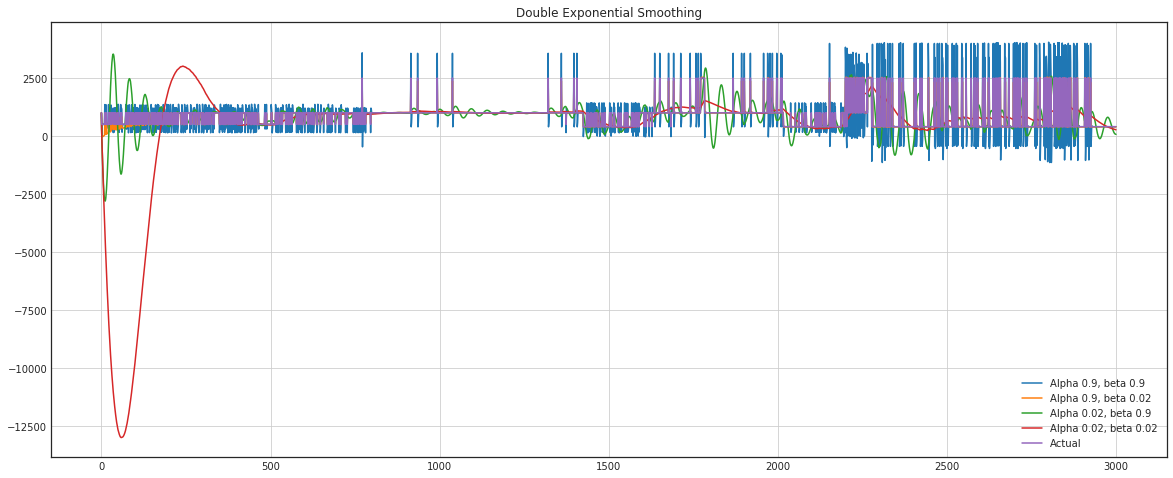


Рис. График двойного экспоненциального сглаживания.

Теперь настраивать пришлось уже два параметра - αα и ββ. Первый отвечает за сглаживание ряда вокруг тренда, второй - за сглаживание самого тренда. Чем выше значения, тем больший вес будет отдаваться последним наблюдениям и тем менее сглаженным окажется модельный ряд. Комбинации параметров могут выдавать достаточно причудливые результаты, особенно если задавать их руками. А о не ручном подборе параметров расскажу чуть ниже, сразу после тройного экспоненциального сглаживания.

**Тройное экспоненциальное сглаживание a.k.a. Holt-Winters**

Идея этого метода заключается в добавлении еще одной, третьей, компоненты - сезонности. Соответственно, метод применим только в случае, если ряд этой сезонностью не обделён, что в нашем случае верно. Сезонная компонента в модели будет объяснять повторяющиеся колебания вокруг уровня и тренда, а характеризоваться она будет длиной сезона - периодом, после которого начинаются повторения колебаний. Для каждого наблюдения в сезоне формируется своя компонента, например, если длина сезона составляет 7 (например, недельная сезонность), то получим 7 сезонных компонент, по штуке на каждый из дней недели.

Получаем новую систему:

ℓx=α(yx−sx−L)+(1−α)(ℓx−1+bx−1)ℓx=α(yx−sx−L)+(1−α)(ℓx−1+bx−1)

bx=β(ℓx−ℓx−1)+(1−β)bx−1bx=β(ℓx−ℓx−1)+(1−β)bx−1

sx=γ(yx−ℓx)+(1−γ)sx−Lsx=γ(yx−ℓx)+(1−γ)sx−L

y^x+m=ℓx+mbx+sx−L+1+(m−1)modLy^x+m=ℓx+mbx+sx−L+1+(m−1)modL

Уровень теперь зависит от текущего значения ряда за вычетом соответствующей сезонной компоненты, тренд остаётся без изменений, а сезонная компонента зависит от текущего значения ряда за вычетом уровня и от предыдущего значения компоненты. При этом компоненты сглаживаются через все доступные сезоны, например, если это компонента, отвечающая за понедельник, от и усредняться она будет только с другими понедельниками. Подробнее про работу усреднений и оценку начальных значений тренда и сезонных компонент можно почитать [здесь](http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc435.htm). Теперь, имея сезонную компоненту, мы можем предсказывать уже не на один, и даже не на два, а на произвольные mm шагов вперёд, что не может не радовать.

Ниже приведен код для построения модели тройного экспоненциального сглаживания, также известного по фамилиям её создателей - Чарльза Хольта и его студента Питера Винтерса. Дополнительно в модель включен метод Брутлага для построения доверительных интервалов:

y^maxx=ℓx−1+bx−1+sx−T+m⋅dt−Ty^maxx=ℓx−1+bx−1+sx−T+m⋅dt−T

y^minx=ℓx−1+bx−1+sx−T−m⋅dt−Ty^minx=ℓx−1+bx−1+sx−T−m⋅dt−T

dt=γ∣yt−y^t∣+(1−γ)dt−Tdt=γ∣yt−y^t∣+(1−γ)dt−T,

где TT - длина сезона, dd - предсказанное отклонение, а остальные параметры берутся из тройного сглаживани. Подробнее о методе и о его применении к поиску аномалий во временных рядах можно прочесть [здесь](https://fedcsis.org/proceedings/2012/pliks/118.pdf)

In [13]:

**class** **HoltWinters**:

*"""*

*Модель Хольта-Винтерса с методом Брутлага для детектирования аномалий*

*https://fedcsis.org/proceedings/2012/pliks/118.pdf*

*# series - исходный временной ряд*

*# slen - длина сезона*

*# alpha, beta, gamma - коэффициенты модели Хольта-Винтерса*

*# n\_preds - горизонт предсказаний*

*# scaling\_factor - задаёт ширину доверительного интервала по Брутлагу (обычно принимает значения от 2 до 3)*

*"""*

**def** \_\_init\_\_(self, series, slen, alpha, beta, gamma, n\_preds, scaling\_factor=1.96):

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.series[i]) / self.slen

**return** sum / self.slen

**def** initial\_seasonal\_components(self):

seasonals = {}

season\_averages = []

n\_seasons = int(len(self.series)/self.slen)

*# вычисляем сезонные средние*

**for** j **in** range(n\_seasons):

season\_averages.append(sum(self.series[self.slen\*j:self.slen\*j+self.slen])/float(self.slen))

*# вычисляем начальные значения*

**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.slen\*j+i]-season\_averages[j]

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: *# инициализируем значения компонент*

smooth = self.series[0]

trend = self.initial\_trend()

self.result.append(self.series[0])

self.Smooth.append(smooth)

self.Trend.append(trend)

self.Season.append(seasonals[i%self.slen])

self.PredictedDeviation.append(0)

self.UpperBond.append(self.result[0] +

self.scaling\_factor \*

self.PredictedDeviation[0])

self.LowerBond.append(self.result[0] -

self.scaling\_factor \*

self.PredictedDeviation[0])

**continue**

**if** i >= len(self.series): *# прогнозируем*

m = i - len(self.series) + 1

self.result.append((smooth + m\*trend) + seasonals[i%self.slen])

*# во время прогноза с каждым шагом увеличиваем неопределенность*

self.PredictedDeviation.append(self.PredictedDeviation[-1]\*1.01)

**else**:

val = self.series[i]

last\_smooth, smooth = smooth, self.alpha\*(val-seasonals[i%self.slen]) + (1-self.alpha)\*(smooth+trend)

trend = self.beta \* (smooth-last\_smooth) + (1-self.beta)\*trend

seasonals[i%self.slen] = self.gamma\*(val-smooth) + (1-self.gamma)\*seasonals[i%self.slen]

self.result.append(smooth+trend+seasonals[i%self.slen])

*# Отклонение рассчитывается в соответствии с алгоритмом Брутлага*

self.PredictedDeviation.append(self.gamma \* np.abs(self.series[i] - self.result[i])

+ (1-self.gamma)\*self.PredictedDeviation[-1])

self.UpperBond.append(self.result[-1] +

self.scaling\_factor \*

self.PredictedDeviation[-1])

self.LowerBond.append(self.result[-1] -

self.scaling\_factor \*

self.PredictedDeviation[-1])

self.Smooth.append(smooth)

self.Trend.append(trend)

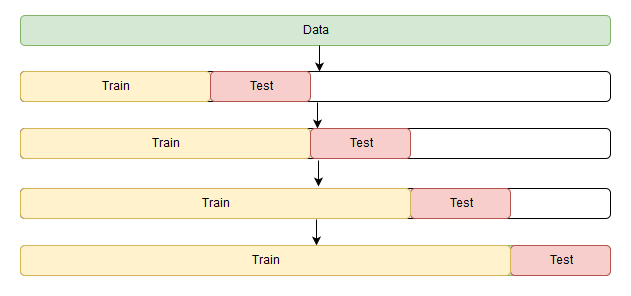
self.Season.append(seasonals[i%self.slen])

**Кросс-валидация на временных рядах**

Сначала необходимо выбрать подходящуюю для данной задачи функцию потерь: [RMSE](https://en.wikipedia.org/wiki/Root-mean-square_deviation), [MAE](https://en.wikipedia.org/wiki/Mean_absolute_error), [MAPE](https://en.wikipedia.org/wiki/Mean_absolute_percentage_error) и др., которая будет следить за качеством подгонки модели под исходные данные. Затем будем оценивать на кросс-валидации значение функции потерь при данных параметрах модели, искать градиент, менять в соответствии с ним параметры и бодро опускаться в сторону глобального минимума ошибки.

Небольшая загвоздка возникает только в кросс-валидации. Проблема состоит в том, что временной ряд имеет, как ни парадоксально, временную структуру, и случайно перемешивать в фолдах значения всего ряда без сохранения этой структуры нельзя, иначе в процессе потеряются все взаимосвязи наблюдений друг с другом. Поэтому придется использовать чуть более хитрый способ для оптимизации параметров, официального названия которому я так и не нашел, но на сайте [CrossValidated](https://stats.stackexchange.com/questions/14099/using-k-fold-cross-validation-for-time-series-model-selection), где можно найти ответы на всё, кроме главного вопроса Жизни, Вселенной и Всего Остального, предлагают название "cross-validation on a rolling basis", что не дословно можно перевести как кросс-валидация на скользящем окне.

Суть достаточно проста - начинаем обучать модель на небольшом отрезке временного ряда, от начала до некоторого tt, делаем прогноз на t+nt+n шагов вперед и считаем ошибку. Далее расширяем обучающую выборку до t+nt+n значения и прогнозируем с t+nt+n до t+2∗nt+2∗n, так продолжаем двигать тестовый отрезок ряда до тех пор, пока не упрёмся в последнее доступное наблюдение. В итоге получим столько фолдов, сколько nn уместится в промежуток между изначальным обучающим отрезком и всей длиной ряда.



Код для кросс-валидации на временном ряду. Значение длины сезона 24\*7 возникло не случайно - в исходном ряде отчетливо видна дневная сезонность, (отсюда 24), и недельная - по будням ниже, на выходных - выше, (отсюда 7), суммарно сезонных компонент получится 24\*7.

In [14]:

**from** **sklearn.model\_selection** **import** TimeSeriesSplit

**def** timeseriesCVscore(x):

*# вектор ошибок*

errors = []

values = data.values

alpha, beta, gamma = x

*# задаём число фолдов для кросс-валидации*

tscv = TimeSeriesSplit(n\_splits=3)

*# идем по фолдам, на каждом обучаем модель, строим прогноз на отложенной выборке и считаем ошибку*

**for** train, test **in** tscv.split(values):

model = HoltWinters(series=values[train], slen = 24\*7, alpha=alpha, beta=beta, gamma=gamma, n\_preds=len(test))

model.triple\_exponential\_smoothing()

predictions = model.result[-len(test):]

actual = values[test]

error = mean\_squared\_error(predictions, actual)

errors.append(error)

**return** np.mean(np.array(errors))

В модели Хольта-Винтерса, как и в остальных моделях экспоненциального сглаживания, есть ограничение на величину сглаживающих параметров - каждый из них может принимать значения от 0 до 1, поэтому для минимизации функции потерь нужно выбирать алгоритм, поддерживающий ограничения на параметры, в данном случае - Truncated Newton conjugate gradient.

In [15]:

%%time

data = dataset.Users[:-500] *# отложим часть данных для тестирования*

*# инициализируем значения параметров*

x = [0, 0, 0]

*# Минимизируем функцию потерь с ограничениями на параметры*

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

*# Из оптимизатора берем оптимальное значение параметров*

alpha\_final, beta\_final, gamma\_final = opt.x

print(alpha\_final, beta\_final, gamma\_final)

0.0005210650469877898 0.00537973933346414 0.023529336915308785

Wall time: 25.8 s

Передадим полученные оптимальные значения коэффициентов αα, ββ и γγ и построим прогноз на 5 дней вперёд (128 часов)

In [16]:

*# Передаем оптимальные значения модели,*

data = dataset.Users

model = HoltWinters(data[:-128], slen = 24\*7, alpha = alpha\_final, beta = beta\_final, gamma = gamma\_final, n\_preds = 128, scaling\_factor = 2.56)

model.triple\_exponential\_smoothing()

Код для отрисовки графика

In [17]:

**def** plotHoltWinters():

Anomalies = np.array([np.NaN]\*len(data))

Anomalies[data.values<model.LowerBond] = data.values[data.values<model.LowerBond]

plt.figure(figsize=(25, 10))

plt.plot(model.result, label = "Model")

plt.plot(model.UpperBond, "r--", alpha=0.5, label = "Up/Low confidence")

plt.plot(model.LowerBond, "r--", alpha=0.5)

plt.fill\_between(x=range(0,len(model.result)), y1=model.UpperBond, y2=model.LowerBond, alpha=0.5, color = "grey")

plt.plot(data.values, label = "Actual")

plt.plot(Anomalies, "o", markersize=10, label = "Anomalies")

plt.axvspan(len(data)-128, len(data), alpha=0.5, color='lightgrey')

plt.grid(**True**)

plt.axis('tight')

plt.legend(loc="best", fontsize=13);

In [18]:

plotHoltWinters()

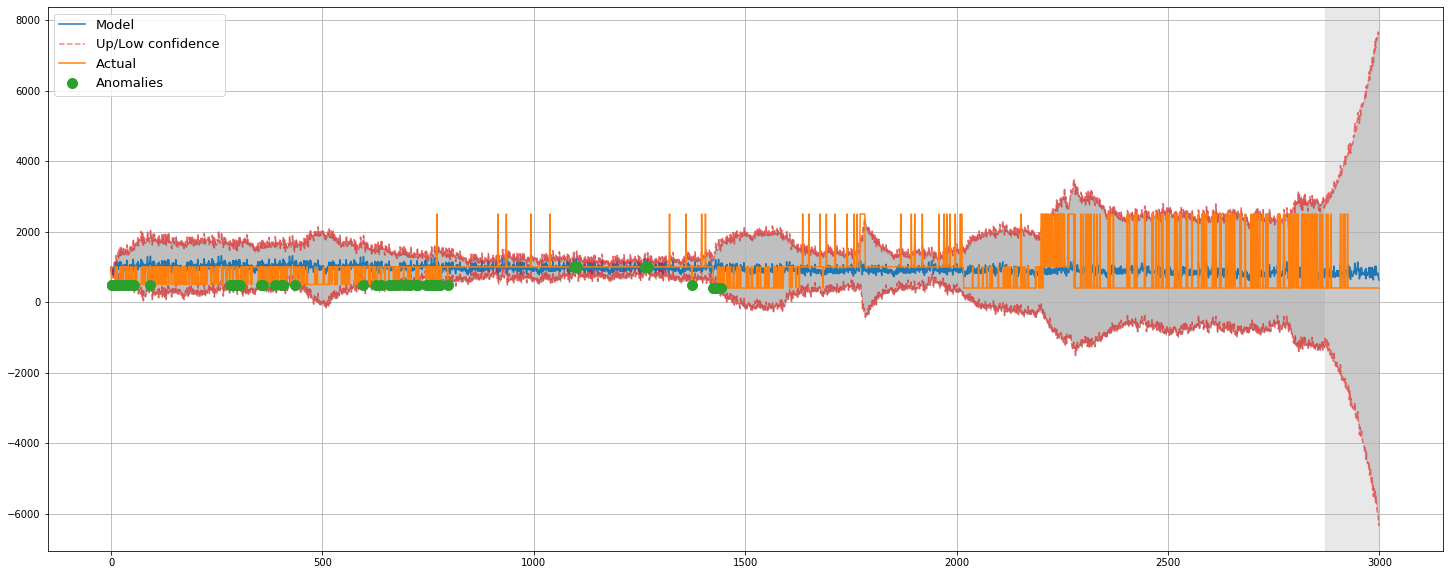


Рис. График модели Хольта-Винтерса

Судя по графику, модель неплохо описала исходный временной ряд, и даже смогла поймать аномальные выбросы, вышедшие за пределы доверительных интервалов. Если посмотреть на смоделированное отклонение, хорошо видно, что модель достаточно резко регирует на значительные изменения в структуре ряда, но при этом быстро возвращает дисперсию к обычным значениям, "забывая" прошлое. Такая особенность позволяет неплохо и без значительных затрат на подготовку-обучение модели настроить систему по детектированию аномалий даже в достаточно шумных рядах.

In [19]:

plt.figure(figsize=(25, 5))

plt.plot(model.PredictedDeviation)

plt.grid(**True**)

plt.axis('tight')

plt.title("Brutlag's predicted deviation");

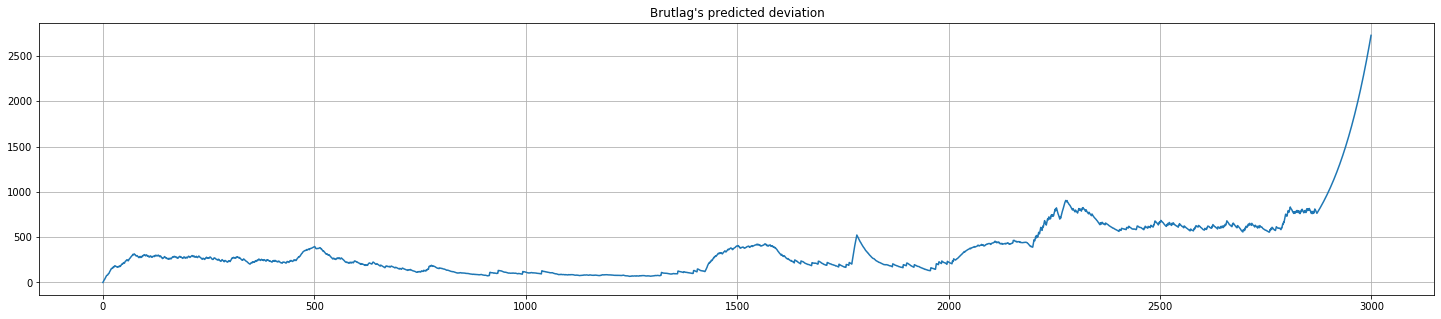


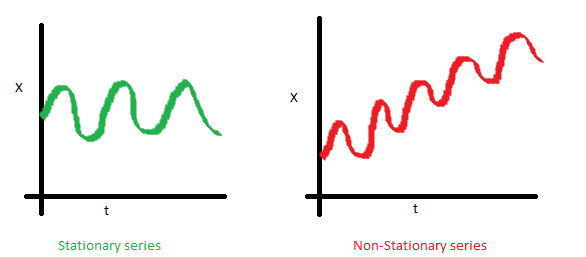
Рис. График предсказания аномалий.

* **Эконометрический подход**

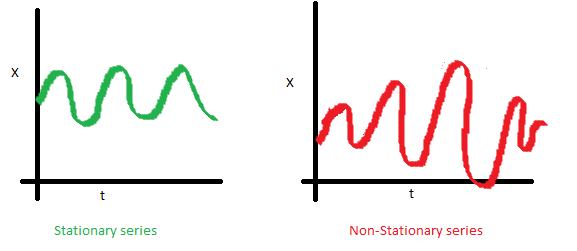
**Стационарность**

Перед тем, как перейти к моделированию, стоит сказать о таком важном свойстве временного ряда, как [**стационарность**](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C). Под стационарностью понимают свойство процесса не менять своих статистических характеристик с течением времени, а именно постоянство матожидания, постоянство дисперсии (она же [гомоскедастичность](https://ru.wikipedia.org/wiki/%D0%93%D0%BE%D0%BC%D0%BE%D1%81%D0%BA%D0%B5%D0%B4%D0%B0%D1%81%D1%82%D0%B8%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C)) и независимость ковариационной функции от времени (должна зависеть только от расстояния между наблюдениями). Наглядно можно посмотреть на эти свойства на картинках, взятых из поста [Sean Abu](http://www.seanabu.com/2016/03/22/time-series-seasonal-ARIMA-model-in-python/):

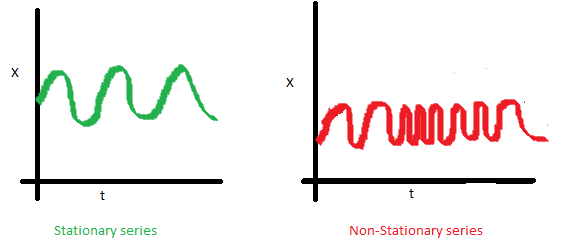
* Временной ряд справа не является стационарным, так как его матожидание со временем растёт



* Здесь не повезло с дисперсией - разброс значений ряда существенно варьируется в зависимости от периода



* Наконец, на последнем графике видно, что значения ряда внезапно становятся ближе друг ко другу, образуя некоторый кластер, а в результате получаем непостоянство ковариаций



Почему стационарность так важна? По стационарному ряду просто строить прогноз, так как мы полагаем, что его будущие статистические характеристики не будут отличаться от наблюдаемых текущих. Большинство моделей временных рядов так или иначе моделируют и предсказывают эти характеристики (например, матожидание или дисперсию), поэтому в случае нестационарности исходного ряда предсказания окажутся неверными. К сожалению, большинство временных рядов, с которыми приходится сталкиваться за пределами учебных материалов, стационарными не являются, но с этим можно (и нужно) бороться.

Чтобы бороться с нестационарностью, нужно узнать её в лицо, потому посмотрим, как её детектировать. Для этого обратимся к белому шуму и случайному блужданию, чтобы выяснить как попасть из одного в другое бесплатно и без смс.

**Избавляемся от нестационарности и строим SARIMA**

Попробуем теперь построить ARIMA модель для наших данных, пройдя все стадии приведения ряда к стационарному виду. Про саму модель можно почитать здесь- [Построение модели SARIMA с помощью Python+R](https://habrahabr.ru/post/210530/), [Анализ временных рядов с помощью python](https://habrahabr.ru/post/207160/), поэтому подробно останавливаться на ней не буду.

Код для отрисовки графиков

In [20]:

**def** tsplot(y, lags=**None**, figsize=(12, 7), style='bmh'):

**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))

pacf\_ax = plt.subplot2grid(layout, (1, 1))

y.plot(ax=ts\_ax)

ts\_ax.set\_title('Time Series Analysis Plots')

smt.graphics.plot\_acf(y, lags=lags, ax=acf\_ax, alpha=0.5)

smt.graphics.plot\_pacf(y, lags=lags, ax=pacf\_ax, alpha=0.5)

print("Критерий Дики-Фуллера: p=**%f**" % sm.tsa.stattools.adfuller(y)[1])

plt.tight\_layout()

**return**

In [21]:

tsplot(dataset.Users, lags=30)

Критерий Дики-Фуллера: p=0.000046

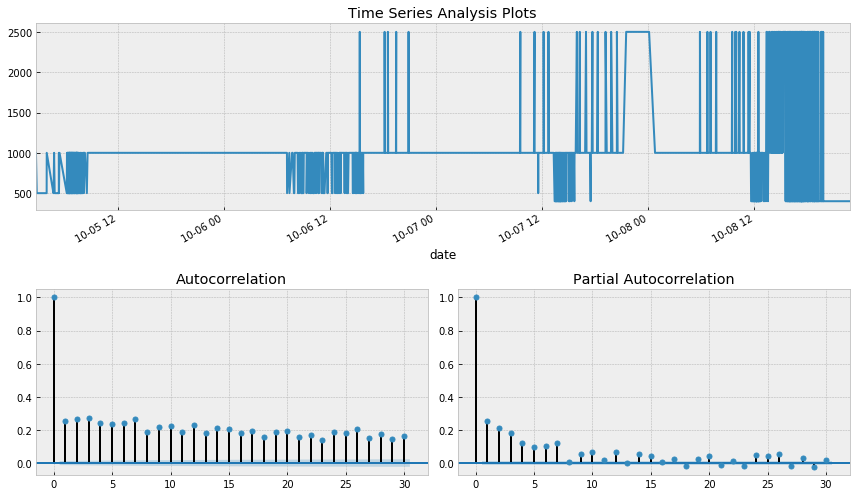


Рис. Графики прогноза и автокореляций

Как и следовало ожидать, исходный ряд стационарным не является, критерий Дики-Фуллера не отверг нулевую гипотезу о наличии единичного корня. Попробуем стабилизировать дисперсию преоразованием Бокса-Кокса

In [22]:

**def** invboxcox(y,lmbda):

*# обратное преобразование Бокса-Кокса*

**if** lmbda == 0:

**return**(np.exp(y))

**else**:

**return**(np.exp(np.log(lmbda\*y+1)/lmbda))

data = dataset.copy()

data['Users\_box'], lmbda = scs.boxcox(data.Bid+1) *# прибавляем единицу, так как в исходном ряде есть нули*

tsplot(data.Users\_box, lags=30)

print("Оптимальный параметр преобразования Бокса-Кокса: **%f**" % lmbda)

Критерий Дики-Фуллера: p=0.010689

Оптимальный параметр преобразования Бокса-Кокса: -0.285677

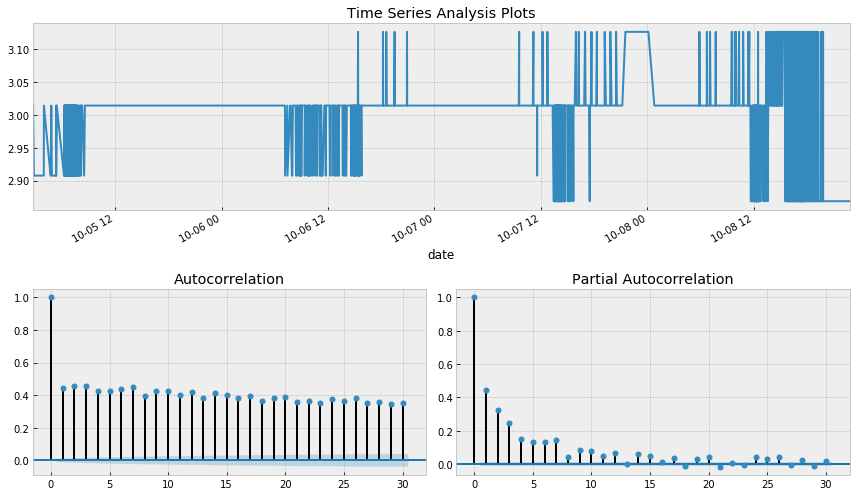


Рис. Графики прогноза и автокореляций

Уже лучше, однако критерий Дики-Фуллера по-прежнему не отвергает гипотезу о нестационарности ряда. А автокорреляционная функция явно намекает на сезонность в получившемся ряде. Возьмём сезонные разности:

In [23]:

data['Users\_box\_season'] = data.Users\_box - data.Users\_box.shift(24\*7)

tsplot(data.Users\_box\_season[24\*7:], lags=30)

Критерий Дики-Фуллера: p=0.001532

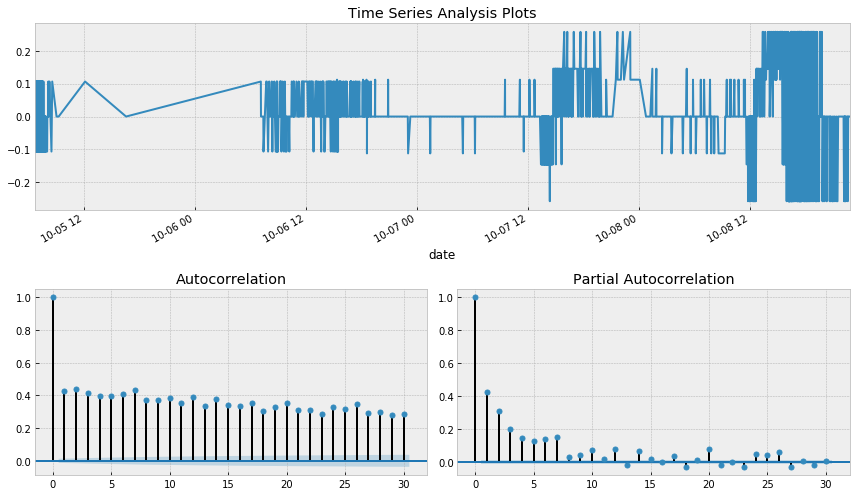


Рис.Рис. Графики прогноза и автокореляций

Критерий Дики-Фуллера теперь отвергает нулевую гипотезу о нестационарности, но автокорреляционная функция всё ещё выглядит нехорошо из-за большого числа значимых лагов. Так как на графике частной автокорреляционной функции значим лишь один лаг, стоит взять еще первые разности, чтобы привести, наконец, ряд к стационарному виду.

In [24]:

data['Users\_box\_season\_diff'] = data.Users\_box\_season - data.Users\_box\_season.shift(1)

tsplot(data.Users\_box\_season\_diff[24\*7+1:], lags=30)

Критерий Дики-Фуллера: p=0.000000

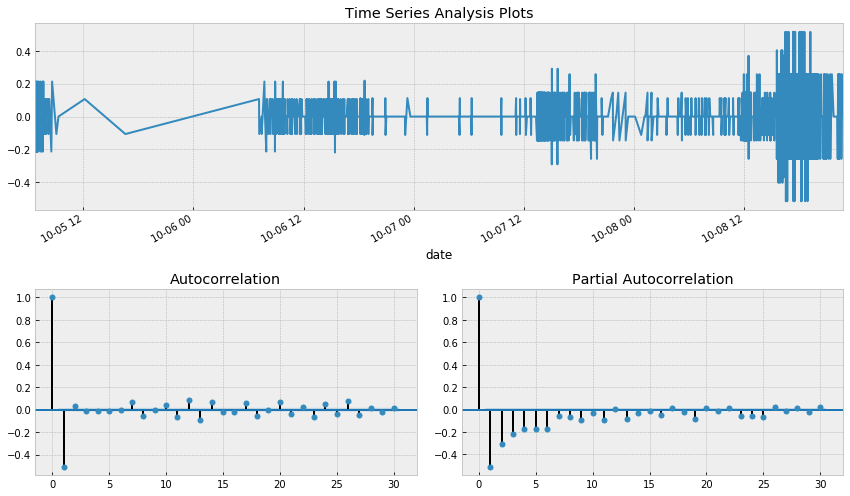


Рис. Графики прогноза и автокорреляций

Наконец, получили стационарный ряд, по автокорреляционной и частной автокорреляционной функции прикинем параметры для SARIMA модели, на забыв, что предварительно уже сделали первые и сезонные разности.

Начальные приближения Q = 1, P = 4, q = 3, p = 4

In [25]:

ps = range(0, 5)

d=1

qs = range(0, 4)

Ps = range(0, 5)

D=1

Qs = range(0, 1)

**from** **itertools** **import** product

parameters = product(ps, qs, Ps, Qs)

parameters\_list = list(parameters)

len(parameters\_list)

Out[25]:

100

In [26]:

%%time

results = []

best\_aic = float("inf")

warnings.filterwarnings('ignore')

**for** param **in** parameters\_list:

*#try except нужен, потому что на некоторых наборах параметров модель не обучается*

**try**:

model=sm.tsa.statespace.SARIMAX(data.Users\_box, order=(param[0], d, param[1]),

seasonal\_order=(param[3], D, param[3], 24)).fit(disp=-1)

*#выводим параметры, на которых модель не обучается и переходим к следующему набору*

**except** **ValueError**:

print('wrong parameters:', param)

**continue**

aic = model.aic

*#сохраняем лучшую модель, aic, параметры*

**if** aic < best\_aic:

best\_model = model

best\_aic = aic

best\_param = param

results.append([param, model.aic])

warnings.filterwarnings('default')

result\_table = pd.DataFrame(results)

result\_table.columns = ['parameters', 'aic']

print(result\_table.sort\_values(by = 'aic', ascending=**True**).head())

parameters aic

90 (4, 2, 0, 0) -6234.927454

92 (4, 2, 2, 0) -6234.927454

93 (4, 2, 3, 0) -6234.927454

91 (4, 2, 1, 0) -6234.927454

94 (4, 2, 4, 0) -6234.927454

Wall time: 9min 7s

Лучшие параметры подставляем в модель:

In [28]:

*#Q*

In [29]:

%%time

best\_model=sm.tsa.statespace.SARIMAX(data.Users\_box, order=(4, d, 3),

seasonal\_order=(4, D, 1, 24)).fit(disp=-1)

C:\Users\nnig9\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa\_model.py:218: ValueWarning:

A date index has been provided, but it has no associated frequency information and so will be ignored when e.g. forecasting.

C:\Users\nnig9\anaconda3\lib\site-packages\statsmodels\base\model.py:568: ConvergenceWarning:

Maximum Likelihood optimization failed to converge. Check mle\_retvals

Wall time: 6min 53s

In [30]:

print(best\_model.summary())

SARIMAX Results

============================================================================================

Dep. Variable: Users\_box No. Observations: 3000

Model: SARIMAX(4, 1, 3)x(4, 1, [1], 24) Log Likelihood 4035.880

Date: Sat, 14 Nov 2020 AIC -8045.760

Time: 17:46:32 BIC -7967.786

Sample: 0 HQIC -8017.701

- 3000

Covariance Type: opg

==============================================================================

coef std err z P>|z| [0.025 0.975]

------------------------------------------------------------------------------

ar.L1 -0.6890 0.423 -1.630 0.103 -1.518 0.140

ar.L2 -0.5627 0.307 -1.832 0.067 -1.165 0.039

ar.L3 0.0349 0.028 1.227 0.220 -0.021 0.091

ar.L4 0.0335 0.021 1.616 0.106 -0.007 0.074

ma.L1 -0.2180 0.423 -0.515 0.607 -1.048 0.612

ma.L2 -0.0090 0.436 -0.021 0.983 -0.864 0.846

ma.L3 -0.5177 0.265 -1.952 0.051 -1.038 0.002

ar.S.L24 0.0404 0.014 2.866 0.004 0.013 0.068

ar.S.L48 -0.0685 0.015 -4.439 0.000 -0.099 -0.038

ar.S.L72 0.0155 0.014 1.078 0.281 -0.013 0.044

ar.S.L96 -0.0569 0.016 -3.495 0.000 -0.089 -0.025

ma.S.L24 -0.9662 0.008 -126.211 0.000 -0.981 -0.951

sigma2 0.0038 6.01e-05 63.096 0.000 0.004 0.004

===================================================================================

Ljung-Box (Q): 66.81 Jarque-Bera (JB): 2927.49

Prob(Q): 0.00 Prob(JB): 0.00

Heteroskedasticity (H): 3.78 Skew: 1.42

Prob(H) (two-sided): 0.00 Kurtosis: 6.94

===================================================================================

Warnings:

[1] Covariance matrix calculated using the outer product of gradients (complex-step).

Проверим остатки модели:

In [31]:

tsplot(best\_model.resid[24:], lags=30)

Критерий Дики-Фуллера: p=0.000000

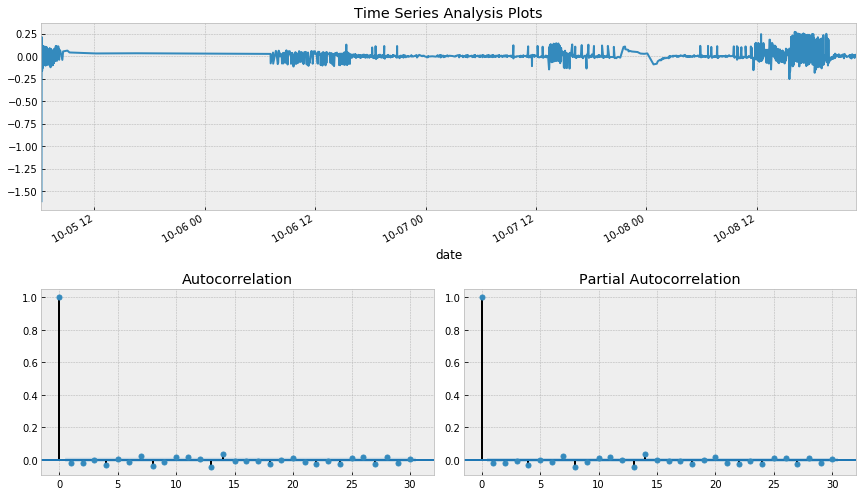


Рис. График остатков временного ряда.

Что ж, остатки стационарны, явных автокорреляций нет, построим прогноз по получившейся модели

In [32]:

data["arima\_model"] = invboxcox(best\_model.fittedvalues, lmbda)

forecast = invboxcox(best\_model.predict(start = data.shape[0], end = data.shape[0]+100), lmbda)

forecast = data.arima\_model.append(forecast).values[-500:]

actual = data.Bid.values[-400:]

plt.figure(figsize=(15, 7))

plt.plot(forecast, color='r', label="model")

plt.title("SARIMA model**\n** Mean absolute error **{}** ".format(round(mean\_absolute\_error(data.dropna().Bid, data.dropna().arima\_model))))

plt.plot(actual, label="actual")

plt.legend()

plt.axvspan(len(actual), len(forecast), alpha=0.5, color='lightgrey')

plt.grid(**True**)

C:\Users\nnig9\anaconda3\lib\site-packages\pandas\core\series.py:856: RuntimeWarning:

invalid value encountered in log

C:\Users\nnig9\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa\_model.py:583: ValueWarning:

No supported index is available. Prediction results will be given with an integer index beginning at `start`.

C:\Users\nnig9\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa\_model.py:587: DeprecationWarning:

No supported index is available. In the next version, calling this method in a model without a supported index will result in an exception.

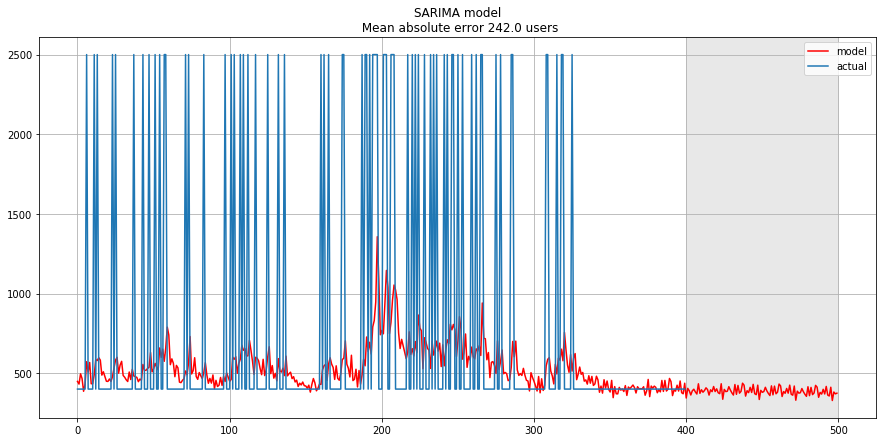


Рис. SARIMA модель

В финале получаем достаточно адекватный прогноз, который слабо отличается от простого сглаживания. Данный тип модели не подходит для обработки нашего временного ряда.

* **Линейные и нелинейные модели на временных рядах**

Снова небольшое лирическое отступление. Часто на работе приходится строить модели, руководствуясь одним основополагающим принципом – [*быстро, качественно, недорого*](http://lurkmore.to/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%BE,_%D0%BA%D0%B0%D1%87%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE,_%D0%BD%D0%B5%D0%B4%D0%BE%D1%80%D0%BE%D0%B3%D0%BE). Поэтому часть моделей могут банально не подойти для "продакшн-решений", так как либо требуют слишком больших затрат по подготовке данных (например, SARIMA), либо сложно настраиваются (хороший пример – SARIMA), либо требуют частого переобучения на новых данных (опять SARIMA), поэтому зачастую гораздо проще бывает выделить несколько признаков из имеющегося временного ряда и построить по ним обычную линейную регрессию или навесить решаюший лес. Дешево и сердито.

Возможно, этот подход не является значительно подкрепленным теорией, нарушает различные предпосылки, например, условия Гаусса-Маркова, особенно пункт про некоррелированность ошибок, однако на практике нередко выручает и достаточно активно используется в соревнованиях по машинному обучению.

* **Извлечение признаков (Feature exctraction)**

Помимо стандартных признаков вроде лагов целевой переменной, много информации содержат в себе дата и время. Про извлечение признаков из них уже здорово описано в одной из [предыдущих статей](https://habrahabr.ru/company/ods/blog/325422/#vremennye-ryady-veb-i-prochee) курса.

Добавлю только про еще один вариант кодирования категориальных признаков – кодирование средним. Если не хочется раздувать датасет множеством дамми-переменных, которые могут привести к потере информации о расстоянии, а в вещественном виде возникают противоречивые результаты а-ля "0 часов < 23 часа", то можно закодировать переменную чуть более интерпретируемыми значениями. Естественный вариант – закодировать средним значением целевой переменной. В нашем случае каждый день недели или час дня можно закодировать сооветствующим средним числом игроков, находившихся в этот день недели или час онлайн. При этом важно следить за тем, чтобы расчет среднего значения производился только в рамках тестового датасета (или в рамках текущего наблюдаемого фолда при кросс-валидации), иначе можно ненароком привнести в модель информацию о будущем.

In [33]:

**def** code\_mean(data, cat\_feature, real\_feature):

*"""*

*Возвращает словарь, где ключами являются уникальные категории признака cat\_feature,*

*а значениями - средние по real\_feature*

*"""*

**return** dict(data.groupby(cat\_feature)[real\_feature].mean())

Создадим новый датафрейм и добавим в него час, день недели и выходной в качестве категориальных переменных. Для этого переводим имеющийся в датафрейме индекс в формат datetime, и извлекаем из него hour и weekday.

In [34]:

data = pd.DataFrame(dataset.copy())

*#data*

In [35]:

data = pd.DataFrame(dataset.copy())

data.columns = ["y"]

data.index = data.index

data["hour"] = data.index.hour

data["weekday"] = data.index.weekday

data['is\_weekend'] = data.weekday.isin([5,6])\*1

data.head()[:3000]

Out[35]:

|  | **y** | **hour** | **weekday** | **is\_weekend** |
| --- | --- | --- | --- | --- |
| **date** |  |  |  |  |
| **2020-10-05 02:45:13** | 1000 | 2 | 0 | 0 |
| **2020-10-05 02:50:42** | 500 | 2 | 0 | 0 |
| **2020-10-05 02:57:33** | 500 | 2 | 0 | 0 |
| **2020-10-05 03:08:50** | 500 | 3 | 0 | 0 |
| **2020-10-05 03:09:12** | 500 | 3 | 0 | 0 |

Посмотрим на средние по дням недели

In [36]:

code\_mean(data, 'weekday', "y")

Out[36]:

{0: 755.6818181818181,

1: 958.3333333333334,

2: 933.7192474674385,

3: 843.2210353327855}

Помимо перечисленных преобразований для увеличения числа признаков используют и множество других метрик, например, максимальное/минимальное значение, наблюдавшееся в скользящем по ряду окне, медианы, число пиков, взвешенные дисперсии и многое другое. Автоматически этим занимается уже упоминавшаяся в курсе [библиотека tsfresh](https://github.com/blue-yonder/tsfresh).

Для удобства все преобразования можно записать в одну функцию, которая сразу же будет возвращать разбитые на трейн и тест датасеты и целевые переменные.

In [37]:

**def** prepareData(data, lag\_start=5, lag\_end=20, test\_size=0.15):

data = pd.DataFrame(data.copy())

data.columns = ["y"]

*# считаем индекс в датафрейме, после которого начинается тестовый отрезок*

test\_index = int(len(data)\*(1-test\_size))

*# добавляем лаги исходного ряда в качестве признаков*

**for** i **in** range(lag\_start, lag\_end):

data["lag\_**{}**".format(i)] = data.y.shift(i)

data.index = data.index

data["hour"] = data.index.hour

data["weekday"] = data.index.weekday

data['is\_weekend'] = data.weekday.isin([5,6])\*1

*# считаем средние только по тренировочной части, чтобы избежать лика*

data['weekday\_average'] = list(map(code\_mean(data[:test\_index], 'weekday', "y").get, data.weekday))

data["hour\_average"] = list(map(code\_mean(data[:test\_index], 'hour', "y").get, data.hour))

*# выкидываем закодированные средними признаки*

data.drop(["hour", "weekday"], axis=1, inplace=**True**)

data = data.dropna()

data = data.reset\_index(drop=**True**)

*# разбиваем весь датасет на тренировочную и тестовую выборку*

X\_train = data.loc[:test\_index].drop(["y"], axis=1)

y\_train = data.loc[:test\_index]["y"]

X\_test = data.loc[test\_index:].drop(["y"], axis=1)

y\_test = data.loc[test\_index:]["y"]

**return** X\_train, X\_test, y\_train, y\_test

Обучим на получившихся данных простую линейную регрессию. При этом лаги будем брать начиная с двенадцатого, таким образом модель будет способна строить предсказания на 12 часов вперёд, имея фактические наблюдения за предыдущие пол дня

In [41]:

**from** **sklearn.linear\_model** **import** LinearRegression

X\_train, X\_test, y\_train, y\_test = prepareData(dataset.Bid, test\_size=0.3, lag\_start=12, lag\_end=48)

lr = LinearRegression()

lr.fit(X\_train, y\_train)

prediction = lr.predict(X\_test)

plt.figure(figsize=(15, 7))

plt.plot(prediction, "r", label="prediction")

plt.plot(y\_test.values, label="actual")

plt.legend(loc="best")

plt.title("Linear regression**\n** Mean absolute error **{}**".format(round(mean\_absolute\_error(prediction, y\_test))))

plt.grid(**True**);

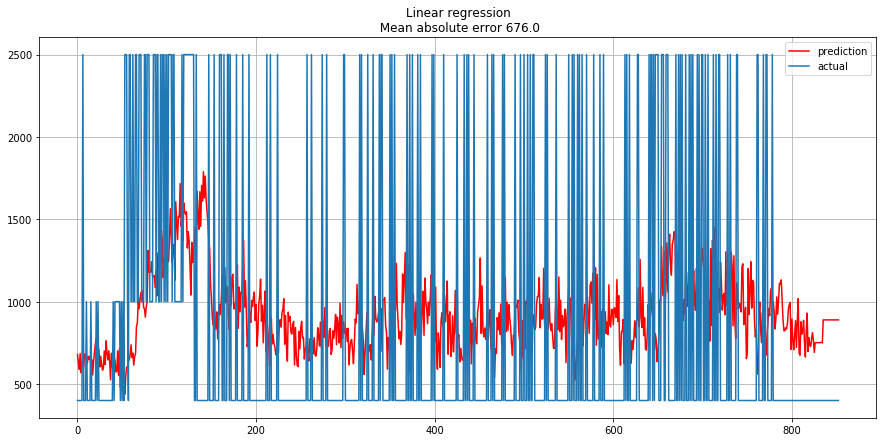


Рис. График линейной регресии

Результат получен, однако даже при отборе признаков модель ошибается довольно существенно. Также можно провести оценку модели на кросс-валидации, тому же принципу, что был использован ранее. Для этого воспользуемся функцией (с небольшими модификациями), предложенной в посте Pythonic Cross Validation on Time Series](<http://francescopochetti.com/pythonic-cross-validation-time-series-pandas-scikit-learn/>)

In [39]:

**def** performTimeSeriesCV(X\_train, y\_train, number\_folds, model, metrics):

print('Size train set: **{}**'.format(X\_train.shape))

k = int(np.floor(float(X\_train.shape[0]) / number\_folds))

print('Size of each fold: **{}**'.format(k))

errors = np.zeros(number\_folds-1)

*# loop from the first 2 folds to the total number of folds*

**for** i **in** range(2, number\_folds + 1):

print()

split = float(i-1)/i

print('Splitting the first ' + str(i) + ' chunks at ' + str(i-1) + '/' + str(i) )

X = X\_train[:(k\*i)]

y = y\_train[:(k\*i)]

print('Size of train + test: **{}**'.format(X.shape)) *# the size of the dataframe is going to be k\*i*

index = int(np.floor(X.shape[0] \* split))

*# folds used to train the model*

X\_trainFolds = X[:index]

y\_trainFolds = y[:index]

*# fold used to test the model*

X\_testFold = X[(index + 1):]

y\_testFold = y[(index + 1):]

model.fit(X\_trainFolds, y\_trainFolds)

errors[i-2] = metrics(model.predict(X\_testFold), y\_testFold)

*# the function returns the mean of the errors on the n-1 folds*

**return** errors.mean()

In [40]:

performTimeSeriesCV(X\_train, y\_train, 5, lr, mean\_absolute\_error)

Size train set: (2101, 39)

Size of each fold: 420

Splitting the first 2 chunks at 1/2

Size of train + test: (840, 39)

Splitting the first 3 chunks at 2/3

Size of train + test: (1260, 39)

Splitting the first 4 chunks at 3/4

Size of train + test: (1680, 39)

Splitting the first 5 chunks at 4/5

Size of train + test: (2100, 39)

Out[40]:

9086770555.698288

На 5 фолдах получили очень большую среднюю абсолютную ошибку, что говорит о неэффективности применеия этого метода прогнозирования.

* **Заключение**

Специфика предоставленных данных не позволяет достаточно эффективно применить Классические методы прогнозирования временных рядов, рассматриваемые в курсах статистического анализа. Для эффективного решения задачи требуется рассмотреть методы, вытекающие из возможности анализа и прогнозирования числовых последовательностей, возможно с применением нейронных сетей.

* **Используемая информация**
* [Онлайн учебник](https://people.duke.edu/~rnau/411home.htm) курса по продвинутому статистическому прогнозированию университета Duke - разобраны всевозможные сглаживания, линейные модели и ARIMA модели
* Статья [Comparison of ARIMA and Random Forest time series models for prediction of avian influenza H5N1 outbreaks](https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-15-276) - одна из немногих, где активно защищается позиция случайного леса в задачах по прогнозированию временных рядов
* Статья [Time Series Analysis (TSA) in Python - Linear Models to GARCH](http://www.blackarbs.com/blog/time-series-analysis-in-python-linear-models-to-garch/11/1/2016) семействе моделей ARIMA и их применении для моделирования финансовых показателей (Brian Christopher)