Rossmann Store Sales

Test challenge

Goal: Explore data and predict 6 weeks of daily sales for 1,115 stores located across Germany.

This notebook mainly focuses on the Time Series Analysis, a topic not covered at Rossmann Competition Kernels. We then disscuss advantages and drawbacks of modeling with Seasonal ARIMA and Prophet.

As it usually goes, we start with the Exploratory Data Analysis of the main metrics revealing present trends and patterns in the data, giving a solid foundation for the further causal analysis.

Also, alternatively to forecasting with Prophet, we use one of the most robust and sophisticated algorythm Extreme Gradient Boosting for regression.

In [428]:

```
import warnings
warnings.filterwarnings("ignore")

# loading packages
# basic + dates
import numpy as np
import pandas as pd
from pandas import datetime

# data visualization
import matplotlib.pyplot as plt
import seaborn as sns # advanced vizs
%matplotlib inline

# statistics
from statsmodels.distributions.empirical_distribution import ECDF

# time series analysis
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# prophet by Facebook
from fbprophet import Prophet

# machine learning: XGB
import xgboost as xgb
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV
from xgboost.sklearn import XGBRegressor # wrapper
```

In [429]:

```
# importing train data to learn
train = pd.read_csv("~/Documents/projects/rossmann/train.csv",
                    parse_dates = True, low_memory = False, index_col = 'Date')

# additional store data
store = pd.read_csv("~/Documents/projects/rossmann/store.csv",
                    low_memory = False)
# time series as indexes
train.index
```

Out[429]:

Out[429]:

```
DatetimeIndex(['2015-07-31', '2015-07-31', '2015-07-31', '2015-07-31',
               '2015-07-31', '2015-07-31', '2015-07-31', '2015-07-31',
               '2015-07-31', '2015-07-31',
               ...
               '2013-01-01', '2013-01-01', '2013-01-01', '2013-01-01',
               '2013-01-01', '2013-01-01', '2013-01-01', '2013-01-01',
               '2013-01-01', '2013-01-01'],
              dtype='datetime64[ns]', name='Date', length=1017209, freq=None)
```

Exploratory Data Analysis

In this first section we go through the train and store data, handle missing values and create new features for further analysis.

In [430]:

```
# first glance at the train set: head and tail
print("In total: ", train.shape)
train.head(5).append(train.tail(5))
```

```
In total:  (1017209, 8)
```

Out[430]:

Date	Store	DayOfWeek	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday
2015-07-31	1	5	5263	555	1	1	0	1
2015-07-31	2	5	6064	625	1	1	0	1
2015-07-31	3	5	8314	821	1	1	0	1
2015-07-31	4	5	13995	1498	1	1	0	1
2015-07-31	5	5	4822	559	1	1	0	1
2013-01-01	1111	2	0	0	0	0	a	1
2013-01-01	1112	2	0	0	0	0	a	1
2013-01-01	1113	2	0	0	0	0	a	1
2013-01-01	1114	2	0	0	0	0	a	1
2013-01-01	1115	2	0	0	0	0	a	1

Short description:

- **Sales: the turnover for any given day (target variable).**
- **Customers: the number of customers on a given day.**
- **Open: an indicator for whether the store was open: 0 = closed, 1 = open.**
- **Promo: indicates whether a store is running a promo on that day.**
- **StateHoliday: indicates a state holiday. Normally all stores, with few exceptions, are closed on state holidays.**
- **SchoolHoliday: indicates if the (Store, Date) was affected by the closure of public schools.**

We are dealing with time series data so it will probably serve us to extract dates for further analysis. We also have two likely correlated vaiables in the dataset, which can be combined into a new feature.

In [431]:

```
# data extraction
train['Year'] = train.index.year
train['Month'] = train.index.month
train['Day'] = train.index.day
train['WeekOfYear'] = train.index.weekofyear
```

```
# adding new variable
train['SalePerCustomer'] = train['Sales']/train['Customers']
train['SalePerCustomer'].describe()
```

Out[431]:

```
count    844340.000000
mean          9.493619
std           2.197494
min           0.000000
25%           7.895563
50%           9.250000
75%          10.899729
max          64.957854
Name: SalePerCustomer, dtype: float64
```

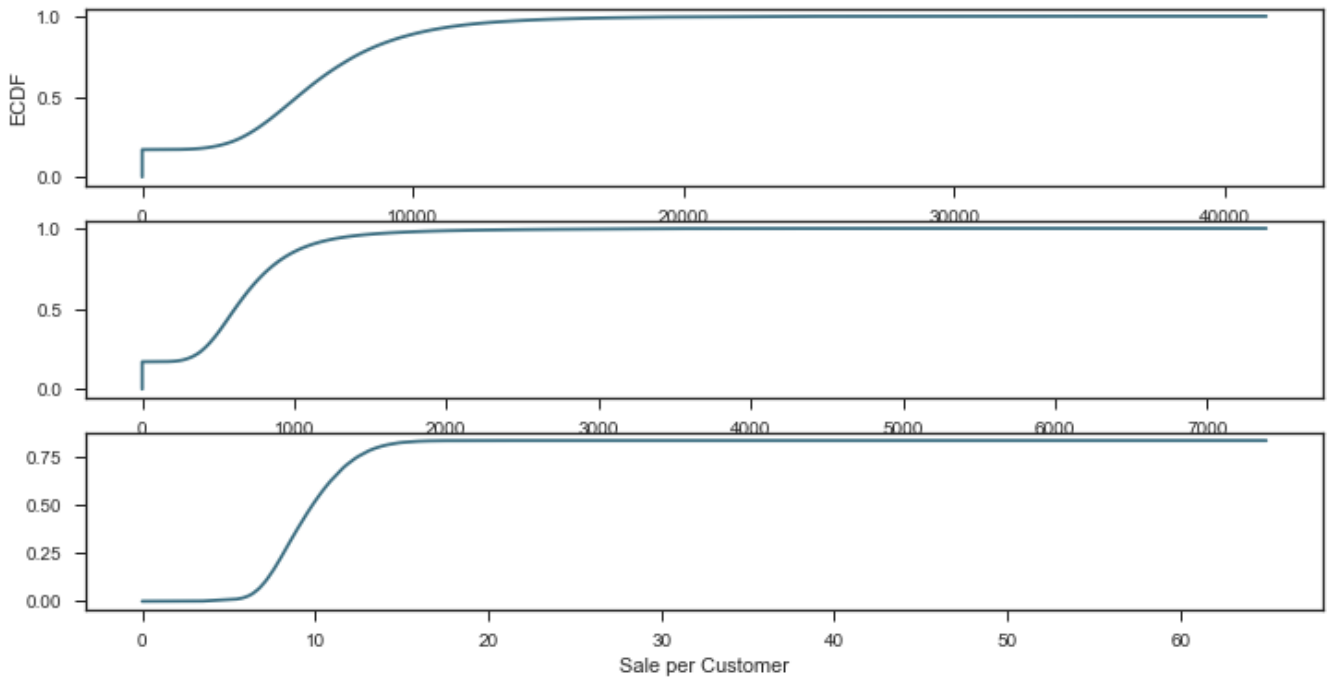
On average customers spend about 9.50$ per day. Though there are days with Sales equal to zero.

ECDF: empirical cumulative distribution function

To get the first impression about continious variables in the data we can plot ECDF.

In [11]:

```
sns.set(style = "ticks")# to format into seaborn
c = '#386B7F' # basic color for plots
plt.figure(figsize = (12, 6))

plt.subplot(311)
cdf = ECDF(train['Sales'])
plt.plot(cdf.x, cdf.y, label = "statmodels", color = c);
plt.xlabel('Sales'); plt.ylabel('ECDF');

# plot second ECDF
plt.subplot(312)
cdf = ECDF(train['Customers'])
plt.plot(cdf.x, cdf.y, label = "statmodels", color = c);
plt.xlabel('Customers');

# plot second ECDF
plt.subplot(313)
cdf = ECDF(train['SalePerCustomer'])
plt.plot(cdf.x, cdf.y, label = "statmodels", color = c);
plt.xlabel('Sale per Customer');
```



About 20% of data has zero amount of sales/customers that we need to deal with and almost 80% of time daily amount of sales was less than 1000. So what about zero sales, is it only due to the fact that the store is closed?

Missing values

Closed stores and zero sales stores

In [200]:

```
# closed stores
train[(train.Open == 0) & (train.Sales == 0)].head()
```

Out[200]:

Date	Store	DayOfWeek	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday	Year	Month	Day	WeekOfYear	Sa
2015-07-31	292	5	0	0	0	1	0	1	2015	7	31	31	
2015-07-31	876	5	0	0	0	1	0	1	2015	7	31	31	
2015-07-30	292	4	0	0	0	1	0	1	2015	7	30	31	
2015-07-30	876	4	0	0	0	1	0	1	2015	7	30	31	
2015-07-29	292	3	0	0	0	1	0	1	2015	7	29	31	

There're 172817 closed stores in the data. It is about 10% of the total amount of observations. To avoid any biased forecasts we will drop these values.

What about opened stores with zero sales?

In [13]:

```
# opened stores with zero sales
zero_sales = train[(train.Open != 0) & (train.Sales == 0)]
print("In total: ", zero_sales.shape)
zero_sales.head(5)
```

```
In total:  (54, 13)
```

Out[13]:

Date	Store	DayOfWeek	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday	Year	Month	Day	WeekOfYear	Sa
2015-05-15	971	5	0	0	1	0	0	1	2015	5	15	20	
2015-03-26	674	4	0	0	1	0	0	0	2015	3	26	13	
2015-02-05	699	4	0	0	1	1	0	0	2015	2	5	6	
2014-10-01	708	3	0	0	1	1	0	0	2014	10	1	40	
2014-09-22	357	1	0	0	1	0	0	0	2014	9	22	39	

Interestingly enough, there are opened store with **no sales on working days**. There're only 54 days in the data, so we can assume that there were external factors involved, for example manifestations

so we can assume that there were external factors involved, for example manifestations.

In [432]:

```
print("Closed stores and days which didn't have any sales won't be counted into the forec
asts.")
train = train[(train["Open"] != 0) & (train['Sales'] != 0)]

print("In total: ", train.shape)
```

```
Closed stores and days which didn't have any sales won't be counted into the forecasts.
In total:  (844338, 13)
```

What about store information:

In [15]:

```
# additional information about the stores
store.head()
```

Out[15]:

	Store	StoreType	Assortment	CompetitionDistance	CompetitionOpenSinceMonth	CompetitionOpenSinceYear	Promo2	Pro
0	1	c	a	1270.0	9.0	2008.0	0	
1	2	a	a	570.0	11.0	2007.0	1	
2	3	a	a	14130.0	12.0	2006.0	1	
3	4	c	c	620.0	9.0	2009.0	0	
4	5	a	a	29910.0	4.0	2015.0	0	

- **Store: a unique Id for each store**
- **StoreType: differentiates between 4 different store models: a, b, c, d**
- **Assortment: describes an assortment level: a = basic, b = extra, c = extended**
- **CompetitionDistance: distance in meters to the nearest competitor store**
- **CompetitionOpenSince[Month/Year]: gives the approximate year and month of the time the nearest competitor was opened**
- **Promo2: Promo2 is a continuing a promotion for some stores: 0 = store is not participating, 1 = store is participating**
- **Promo2Since[Year/Week]: describes the year and calendar week when the store started participating in Promo2**
- **PromoInterval: describes the consecutive intervals Promo2 is started, naming the months the promotion is started. E.g. "Feb,May,Aug,Nov" means each round starts in February, May, August, November of any given year for that store**

In [16]:

```
# missing values?
store.isnull().sum()
```

Out[16]:

```
Store                        0
StoreType                    0
Assortment                   0
CompetitionDistance          3
CompetitionOpenSinceMonth  354
CompetitionOpenSinceYear   354
Promo2                       0
Promo2SinceWeek            544
Promo2SinceYear            544
PromoInterval              544
dtype: int64
```

We have few variables with missing values that we need to deal with. Let's start with the

`CompetitionDistance`.

In [17]:

```
# missing values in CompetitionDistance
store[pd.isnull(store.CompetitionDistance)]
```

Out[17]:

	Store	StoreType	Assortment	CompetitionDistance	CompetitionOpenSinceMonth	CompetitionOpenSinceYear	Promo2	P
290	**291**	**d**	**a**	**NaN**	**NaN**	**NaN**	**0**	
621	**622**	**a**	**c**	**NaN**	**NaN**	**NaN**	**0**	
878	**879**	**d**	**a**	**NaN**	**NaN**	**NaN**	**1**	

Apperently this information is simply missing from the data. No particular pattern observed. In this case, it makes a complete sense to replace NaN with the median values (which is twice less that the average).

In [433]:

```
# fill NaN with a median value (skewed distribuion)
store['CompetitionDistance'].fillna(store['CompetitionDistance'].median(), inplace = Tru
e)
```

Continuing further with missing data. What about `Promo2SinceWeek`? May it be that we observe unsusual data points?

In [20]:

```
# no promo = no information about the promo?
_ = store[pd.isnull(store.Promo2SinceWeek)]
_[_.Promo2 != 0].shape
```

Out[20]:

```
(0, 10)
```

No, if there's no `Promo2` then there's no information about it. We can replace these values by zeros. The same goes for tha variables deducted from the competition, `CompetitionOpenSinceMonth` and `CompetitionOpenSinceYear`.

In [434]:

```
# replace NA's by 0
store.fillna(0, inplace = True)
```

In [435]:

```
print("Joining train set with an additional store information.")

# by specifying inner join we make sure that only those observations
# that are present in both train and store sets are merged together
train_store = pd.merge(train, store, how = 'inner', on = 'Store')

print("In total: ", train_store.shape)
train_store.head()
```

```
Joining train set with an additional store information.
In total:  (844338, 22)
```

Out[435]:

	Store	DayOfWeek	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday	Year	Month	...	SalePerCustomer	Stor
0	**1**	**5**	**5263**	**555**	**1**	**1**	**0**	**1**	**2015**	**7**	**...**	**9.482883**	

	Store	DayOfWeek	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday	Year	Month	...	SalePerCustomer	Stor
1	1	4	5020	546	1	1	0	1	2015	7	...	9.194139	
2	1	3	4782	523	1	1	0	1	2015	7	...	9.143403	
3	1	2	5011	560	1	1	0	1	2015	7	...	8.948214	
4	1	1	6102	612	1	1	0	1	2015	7	...	9.970588	

5 rows × 22 columns

Store types

In this section we will closely look at different levels of `StoreType` and how the main metric `Sales` is distributed among them.

In [23]:

```
train_store.groupby('StoreType')['Sales'].describe()
```

Out[23]:

StoreType	count	mean	std	min	25%	50%	75%	max
a	457042.0	6925.697986	3277.351589	46.0	4695.25	6285.0	8406.00	41551.0
b	15560.0	10233.380141	5155.729868	1252.0	6345.75	9130.0	13184.25	38722.0
c	112968.0	6933.126425	2896.958579	133.0	4916.00	6408.0	8349.25	31448.0
d	258768.0	6822.300064	2556.401455	538.0	5050.00	6395.0	8123.25	38037.0

`StoreType` B has the highest average of Sales among all others, however we have much less data for it. So let's print an overall sum of `Sales` and `Customers` to see which `StoreType` is the most selling and crowded one:

In [24]:

```
train_store.groupby('StoreType')['Customers', 'Sales'].sum()
```

Out[24]:

StoreType	Customers	Sales
a	363541431	3165334859
b	31465616	159231395
c	92129705	783221426
d	156904995	1765392943

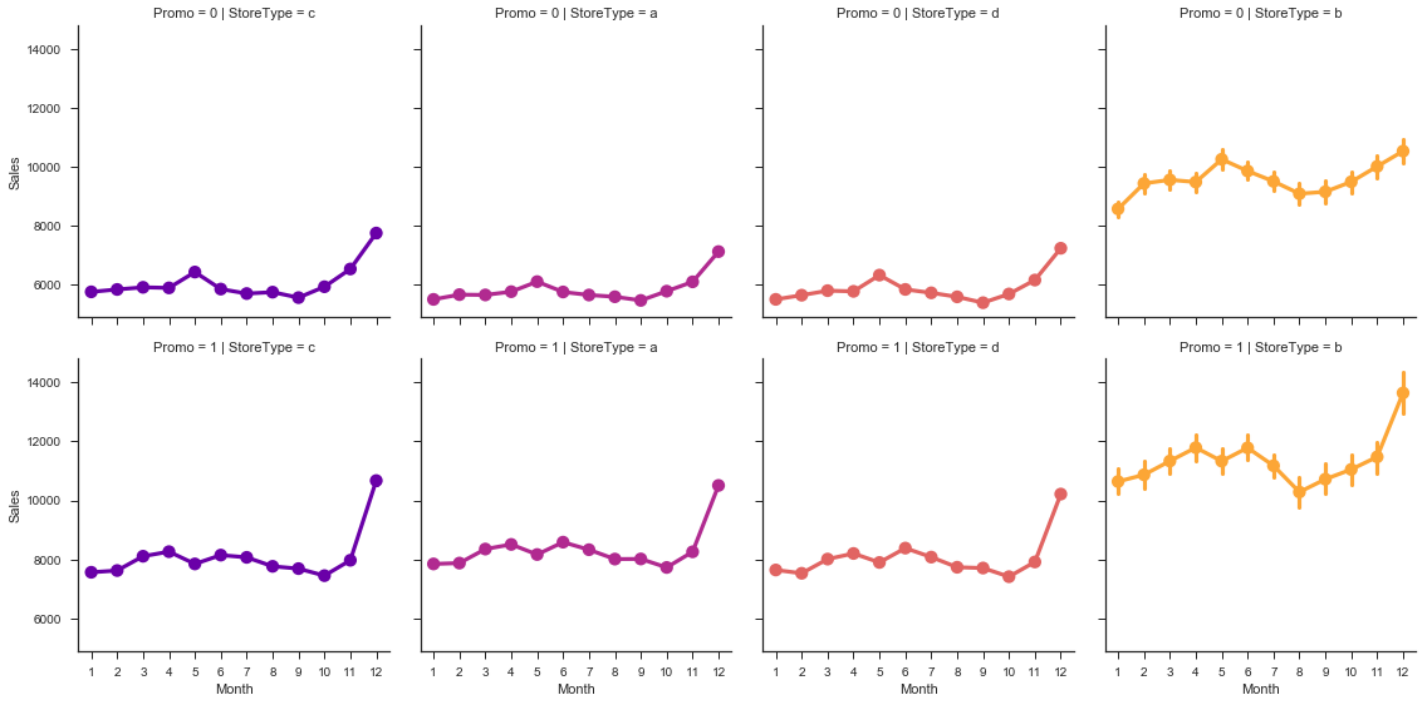
Clearly stores of type A. `StoreType` D goes on the second place in both `Sales` and `Customers`. What about date periods? Seaborn's facet grid is the best tool for this task:

In [25]:

```
# sales trends
sns.factorplot(data = train_store, x = 'Month', y = "Sales",
               col = 'StoreType', # per store type in cols
               palette = 'plasma',
               hue = 'StoreType',
               row = 'Promo', # per promo in the store in rows
               color = c)
```

Out[25]:

```
<seaborn.axisgrid.FacetGrid at 0x11b46f588>
```

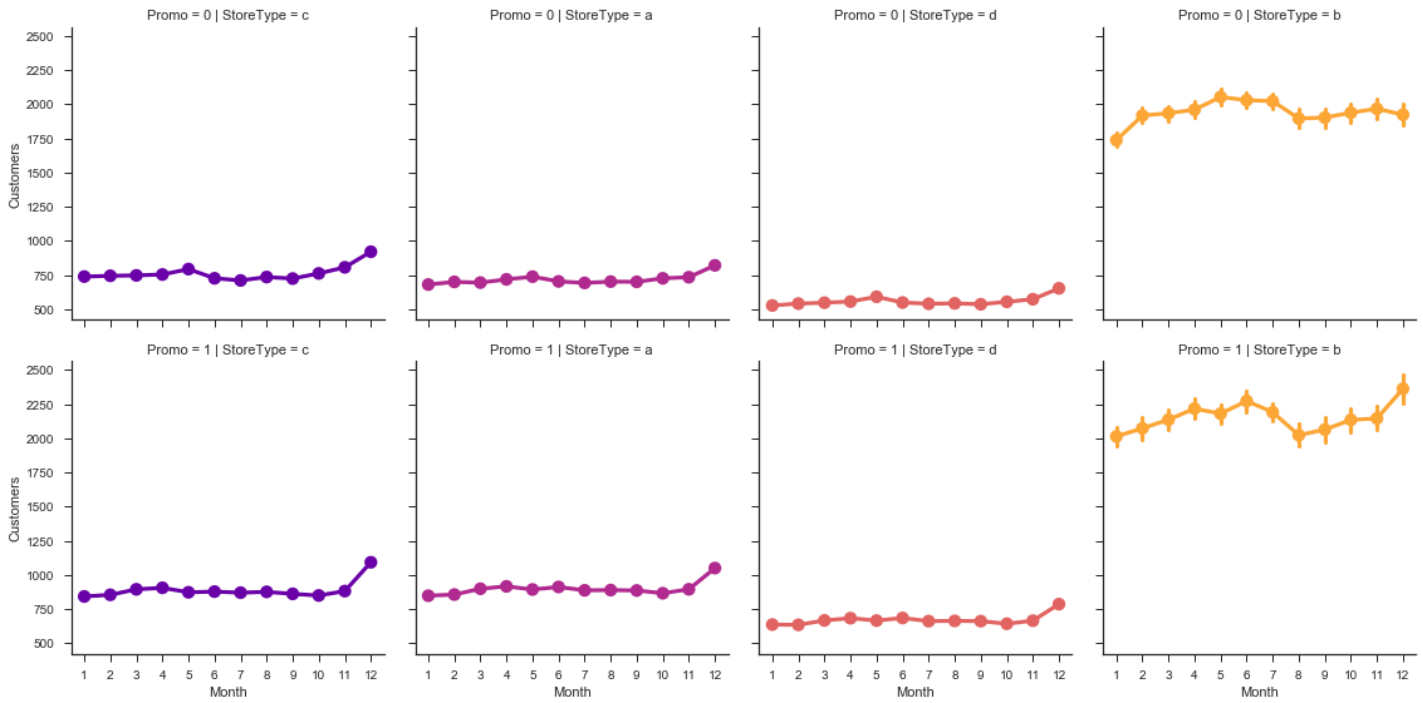


In [26]:

```
# sales trends
sns.factorplot(data = train_store, x = 'Month', y = "Customers",
               col = 'StoreType', # per store type in cols
               palette = 'plasma',
               hue = 'StoreType',
               row = 'Promo', # per promo in the store in rows
               color = c)
```

Out[26]:

```
<seaborn.axisgrid.FacetGrid at 0x105eae5f8>
```



All store types follow the same trend but at different scales depending on the presence of the (first) promotion `Promo` and `StoreType` itself (case for B).

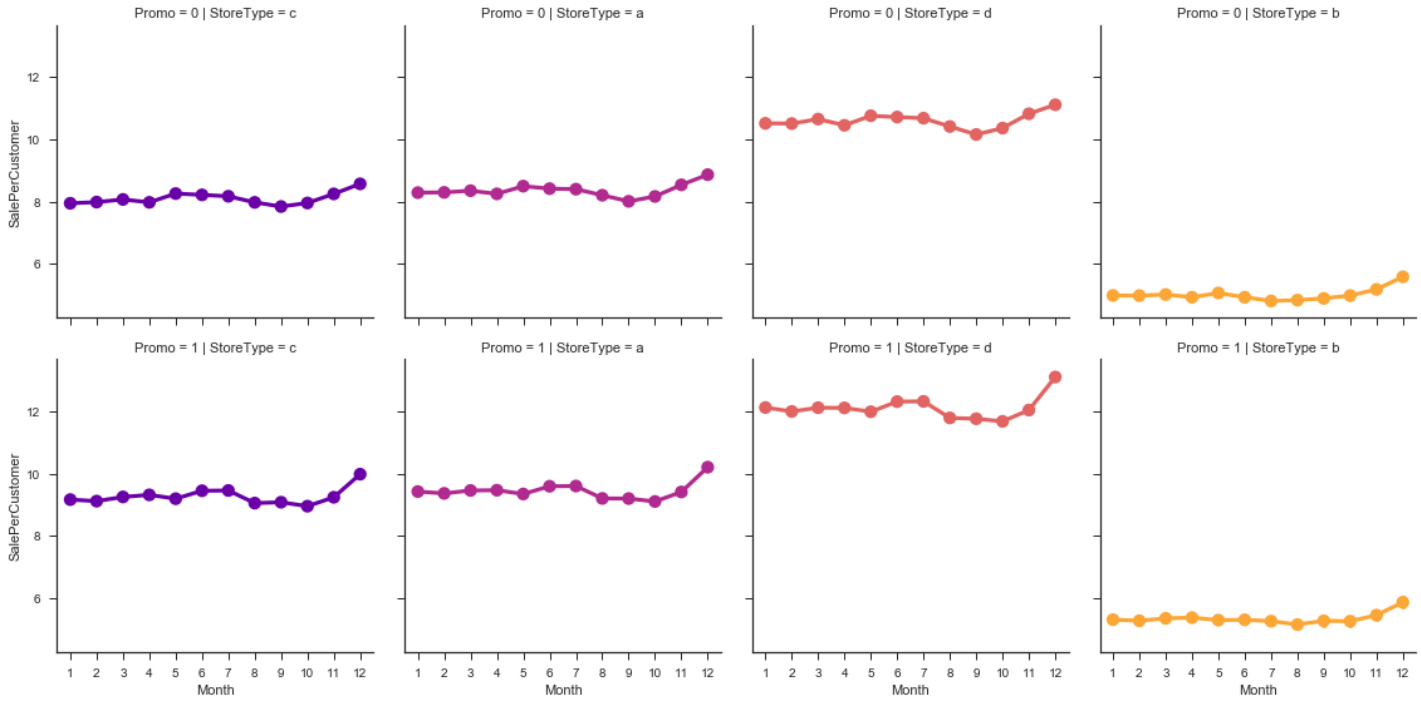
Already at this point, we can see that Sales escalate towards Christmas holidays. But we'll talk about seasonalities and trends later in the Time Series Analysis section.

In [400]:

```
# sale per customer trends
sns.factorplot(data = train_store, x = 'Month', y = "SalePerCustomer",
               col = 'StoreType', # per store type in cols
               palette = 'plasma',
               hue = 'StoreType',
               row = 'Promo', # per promo in the store in rows
               color = c)
```

Out[400]:

```
<seaborn.axisgrid.FacetGrid at 0x1cab84cc0>
```



Aha! Eventhough the plots above showed `StoreType` B as the most selling and performant one, in reality it is not true. The highest `SalePerCustomer` amount is observed at the `StoreType` D, about 12€ with `Promo` and 10€ without. As for `StoreType` A and C it is about 9€.

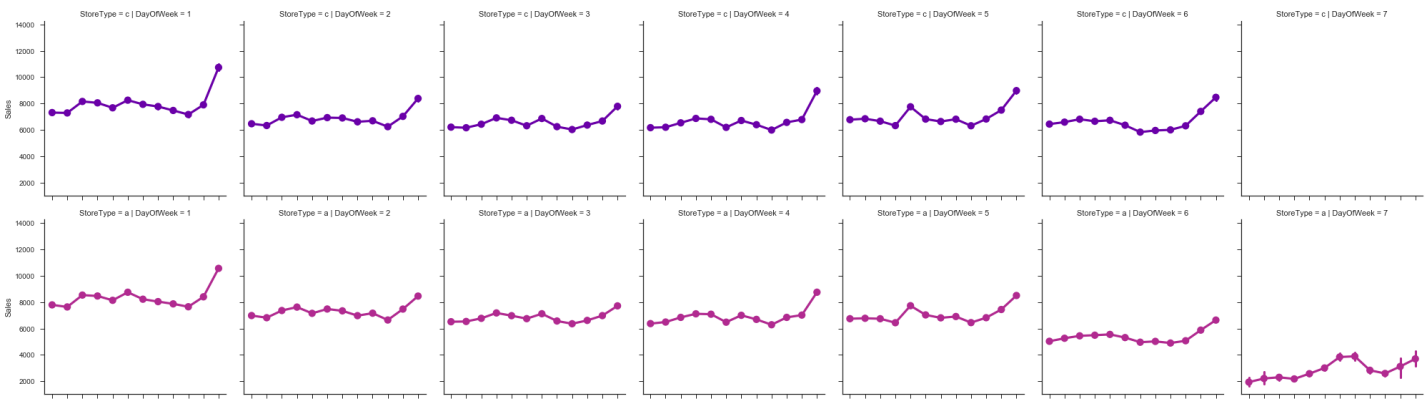
Low `SalePerCustomer` amount for `StoreType` B describes its Buyer Cart: there are a lot of people who shop essentially for "small" things (or in a little quantity). Plus we saw that overall this `StoreType` generated the least amount of sales and customers over the period.

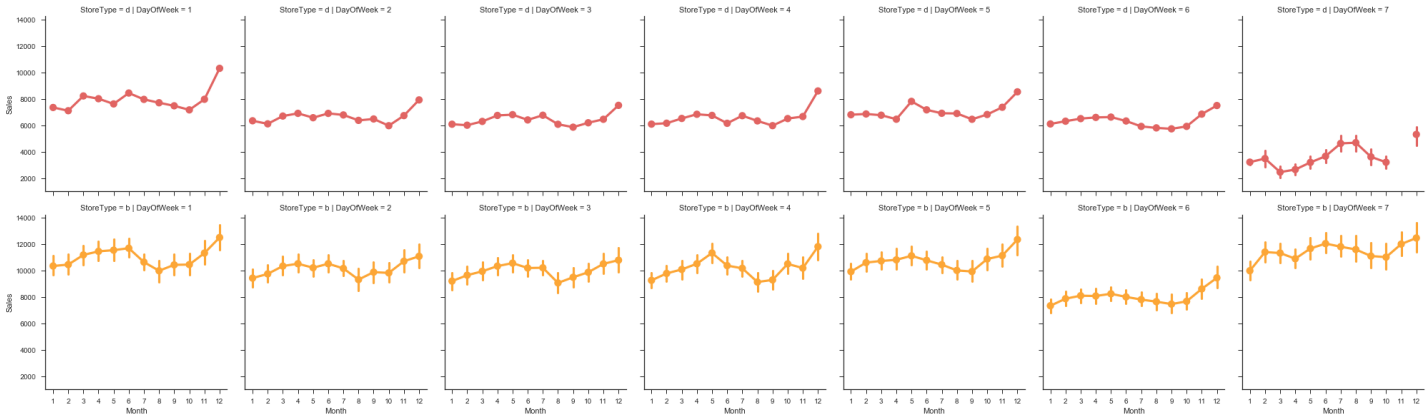
In [465]:

```
# customers
sns.factorplot(data = train_store, x = 'Month', y = "Sales",
               col = 'DayOfWeek', # per store type in cols
               palette = 'plasma',
               hue = 'StoreType',
               row = 'StoreType', # per store type in rows
               color = c)
```

Out[465]:

```
<seaborn.axisgrid.FacetGrid at 0x1d7380eb8>
```





We see that stores of `StoreType` C are all closed on Sundays, whereas others are most of the time opened. Interestingly enough, stores of `StoreType` D are closed on Sundays only from October to December.

Bt the way what are the stores which are opened on Sundays?

In [432]:

```
# stores which are opened on Sundays
train_store[(train_store.Open == 1) & (train_store.DayOfWeek == 7)]['Store'].unique()
```

Out[432]:

```
array([  85,  122,  209,  259,  262,  274,  299,  310,  335,  353,  423,
        433,  453,  494,  512,  524,  530,  562,  578,  676,  682,  732,
        733,  769,  863,  867,  877,  931,  948, 1045, 1081, 1097, 1099])
```

To complete our preliminary data analysis, we can add variables describing the period of time during which competition and promotion were opened:

In [436]:

```
# competition open time (in months)
train_store['CompetitionOpen'] = 12 * (train_store.Year - train_store.CompetitionOpenSin
ceYear) + \
        (train_store.Month - train_store.CompetitionOpenSinceMonth)

# Promo open time
train_store['PromoOpen'] = 12 * (train_store.Year - train_store.Promo2SinceYear) + \
        (train_store.WeekOfYear - train_store.Promo2SinceWeek) / 4.0

# replace NA's by 0
train_store.fillna(0, inplace = True)

# average PromoOpen time and CompetitionOpen time per store type
train_store.loc[:, ['StoreType', 'Sales', 'Customers', 'PromoOpen', 'CompetitionOpen']].
groupby('StoreType').mean()
```

Out[436]:

StoreType	Sales	Customers	PromoOpen	CompetitionOpen
a	6925.697986	795.422370	12918.492198	7115.514452
b	10233.380141	2022.211825	17199.328069	11364.495244
c	6933.126425	815.538073	12158.636107	6745.418694
d	6822.300064	606.353935	10421.916846	9028.526526

The most selling and crowded `StoreType` A doesn't appear to be the one the most exposed to competitors. Instead it's a `StoreType` B, which also has the longest running period of promotion.

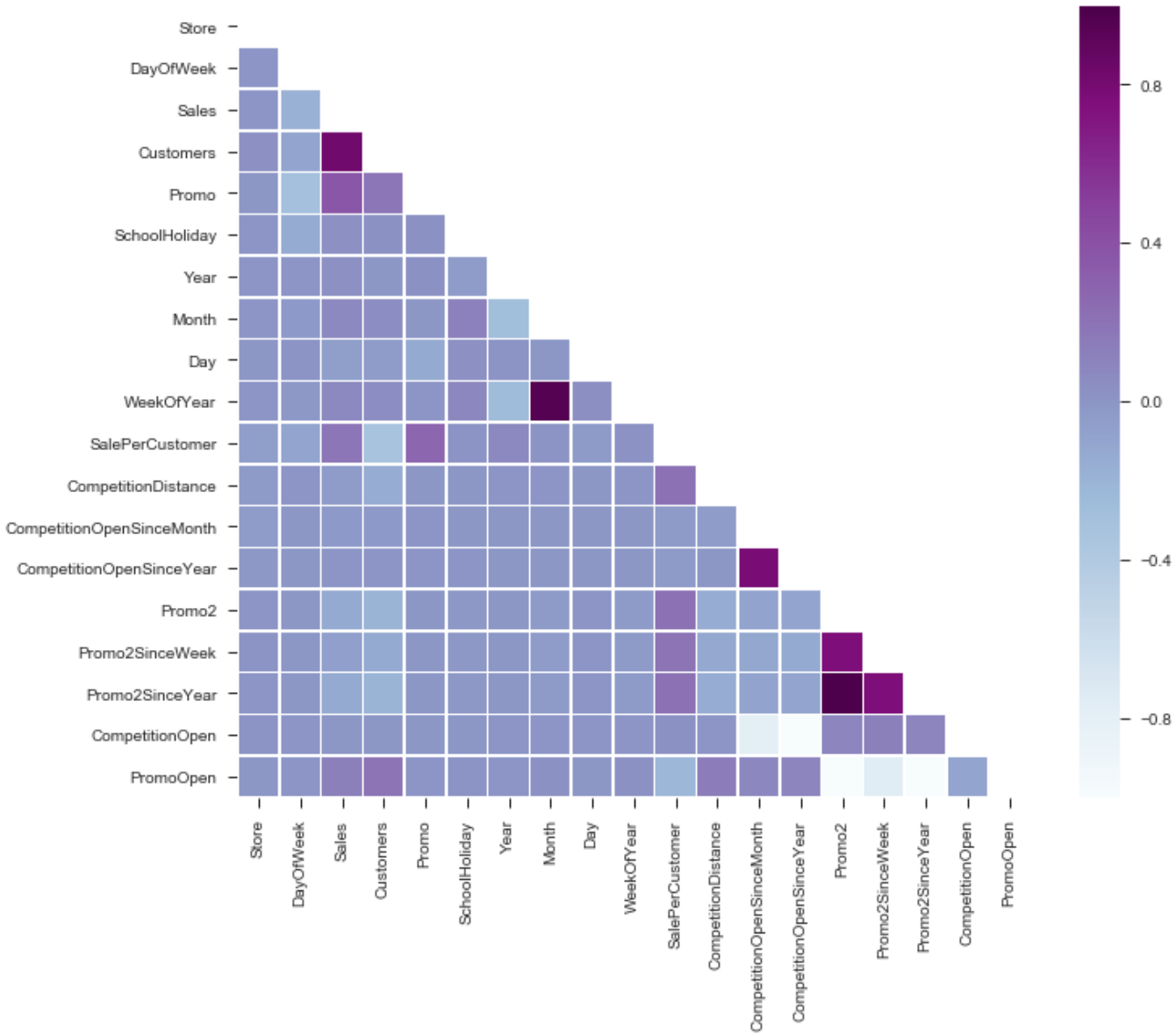
Correlational Analysis

Correlational Analysis

We are finished with adding new variables to the data, so now we can check the overall correlations by plotting the `seaborn` heatmap:

In [35]:

```
# Compute the correlation matrix
# exclude 'Open' variable
corr_all = train_store.drop('Open', axis = 1).corr()

# Generate a mask for the upper triangle
mask = np.zeros_like(corr_all, dtype = np.bool)
mask[np.triu_indices_from(mask)] = True

# Set up the matplotlib figure
f, ax = plt.subplots(figsize = (11, 9))

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr_all, mask = mask,
            square = True, linewidths = .5, ax = ax, cmap = "BuPu")
plt.show()
```



As mentioned before, we have a strong positive correlation between the amount of Sales and Customers of a store. We can also observe a positive correlation between the fact that the store had a running promotion (`Promo` equal to 1) and amount of `Customers`.

However, as soon as the store continues a consecutive promotion (`Promo2` equal to 1) the number of `Customers` and `Sales` seems to stay the same or even decrease, which is described by the pale negative correlation on the heatmap. The same negative correlation is observed between the presence of the promotion

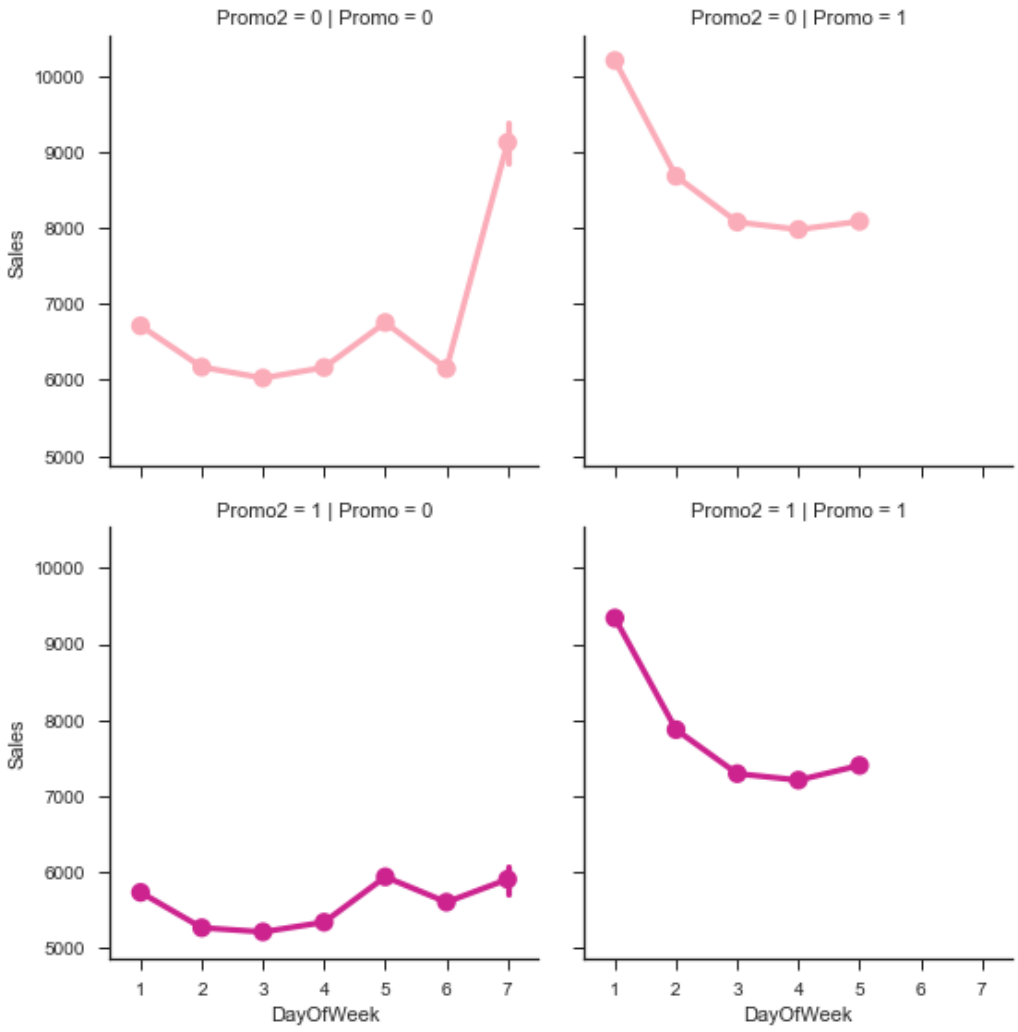
correlation on the heatmap. The same negative correlation is observed between the presence of the promotion in the store and the day of a week.

In [406]:

```
# sale per customer trends
sns.factorplot(data = train_store, x = 'DayOfWeek', y = "Sales",
               col = 'Promo',
               row = 'Promo2',
               hue = 'Promo2',
               palette = 'RdPu')
```

Out[406]:

```
<seaborn.axisgrid.FacetGrid at 0x1e2ff6d68>
```



There are several things here:

- **In case of no promotion, both `Promo` and `Promo2` are equal to 0, `Sales` tend to peak on Sunday (!). Though we should note that `StoreType` C doesn't work on Sundays. So it is mainly data from `StoreType` A, B and D.**
- **On the contrary, stores that run the promotion tend to make most of the `Sales` on Monday. This fact could be a good indicator for Rossmann marketing campaigns. The same trend follow the stores which have both promotion at the same time (`Promo` and `Promo2` are equal to 1).**
- **`Promo2` alone doesn't seem to be correlated to any significant change in the `Sales` amount. This can be also prooved by the blue pale area on the heatmap above.**

Conclusion of EDA

- **The most selling and crowded `StoreType` is A.**
- **The best "Sale per Customer" `StoreType` D indicates to the higher Buyer Cart. We could also assume that the stores of this types are situated in the rural areas, so that customers prefer buying more but less often.**
- **Low `SalePerCustomer` amount for `StoreType` B indicates to the possible fact that people shop there essentially for small things. Which can also indicate to the label of this store type - "urban" - as it's more accessible for public, and customers don't mind shopping there from time to time during a week.**
- **Customers tends to buy more on Mondays when there's one promotion running (`Promo`) and on Sundays when there is no promotion at all (both `Promo` and `Promo1` are equal to 0).**
- **Promo2 alone doesn't seem to be correlated to any significant change in the `Sales` amount.**

Time-Series Analysis per Store Type

What makes a time series different from a regular regression problem?

- **It is time dependent. The basic assumption of a linear regression that the observations are independent doesn't hold in this case.**
- **Along with an increasing or decreasing trend, most time series have some form of seasonality trends, i.e. variations specific to a particular time frame. For example, for Christmas holidays, which we will see in this dataset.**

We build a time series analysis on store types instead of individual stores. The main advantage of this approach is its simplicity of presentation and overall account for different trends and seasonaltities in the dataset.

In this section, we will analyse time series data: its trends, sesonalities and autocorrelation. Usually at the end of the analysis, we are able to develop a seasonal ARIMA (Autoregression Integrated Moving Average) model but it won't be our main focus today. Instead, we try to understand the data, and only later come up with the forecasts using Prophet methodology.

Seasonality

We take four stores from store types to represent their group:

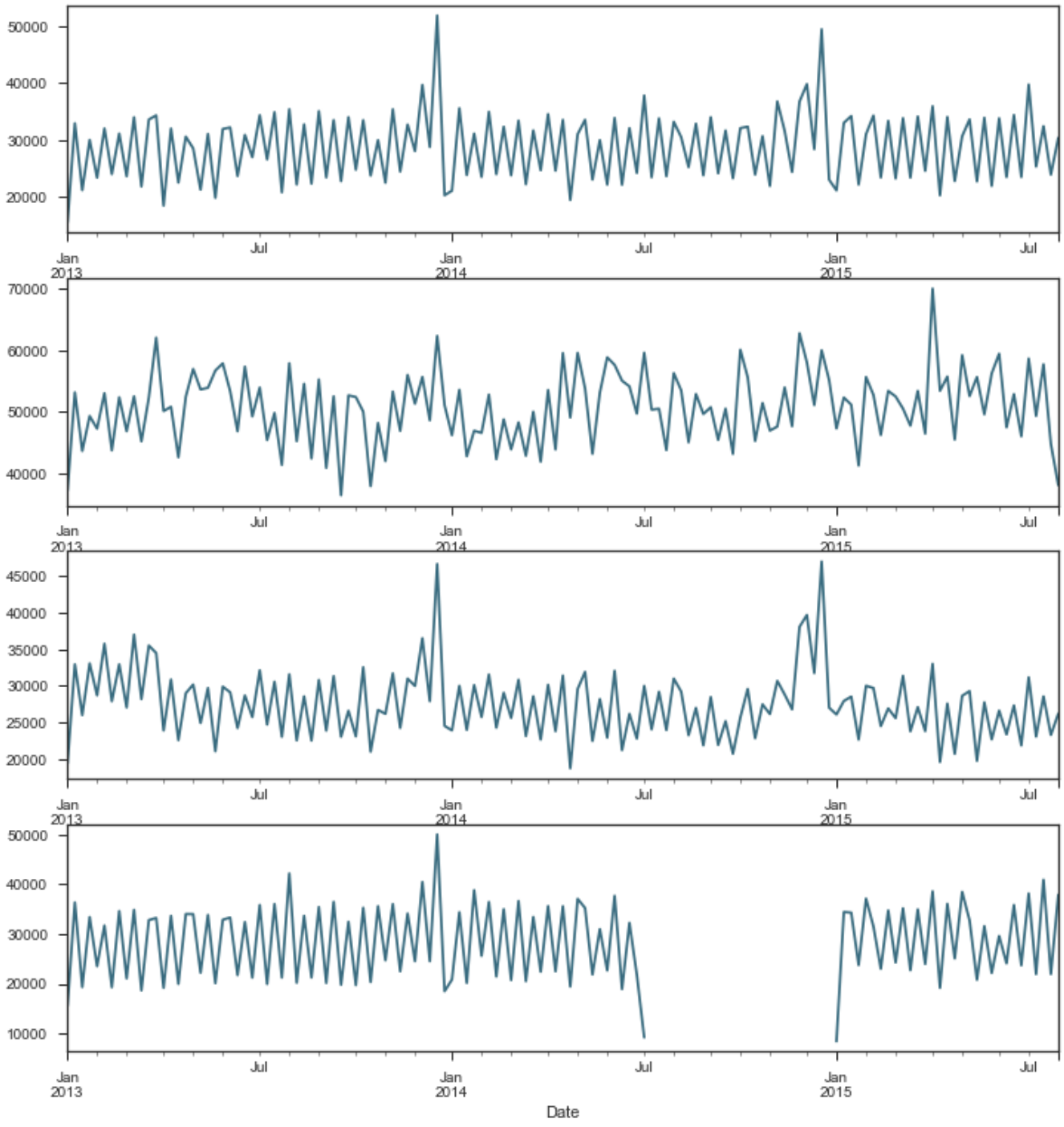
- **Store number 2 for `StoreType` A**
- **Store number 85 for `StoreType` B,**
- **Store number 1 for `StoreType` C**
- **Store number 13 for `StoreType` D.**

It also makes sense to downsample the data from days to weeks using the `resample` method to see the present trends more clearly.

In [546]:

```
# preparation: input should be float type
train['Sales'] = train['Sales'] * 1.0

# store types
sales_a = train[train.Store == 2]['Sales']
sales_b = train[train.Store == 85]['Sales'].sort_index(ascending = True) # solve the rev
erse order
sales_c = train[train.Store == 1]['Sales']
```

```
sales_d = train[train.Store == 13]['Sales']

f, (ax1, ax2, ax3, ax4) = plt.subplots(4, figsize = (12, 13))

# store types
sales_a.resample('W').sum().plot(color = c, ax = ax1)
sales_b.resample('W').sum().plot(color = c, ax = ax2)
sales_c.resample('W').sum().plot(color = c, ax = ax3)
sales_d.resample('W').sum().plot(color = c, ax = ax4)
```



Retail sales for `StoreType` A and C tend to peak for the Christmas season and then decline after the holidays. We might have seen the same trend for `StoreType` D (at the bottom) but there is no information from July 2014 to January 2015 about these stores as they were closed.

Yearly trend

The next thing to check the presence of a trend in series.

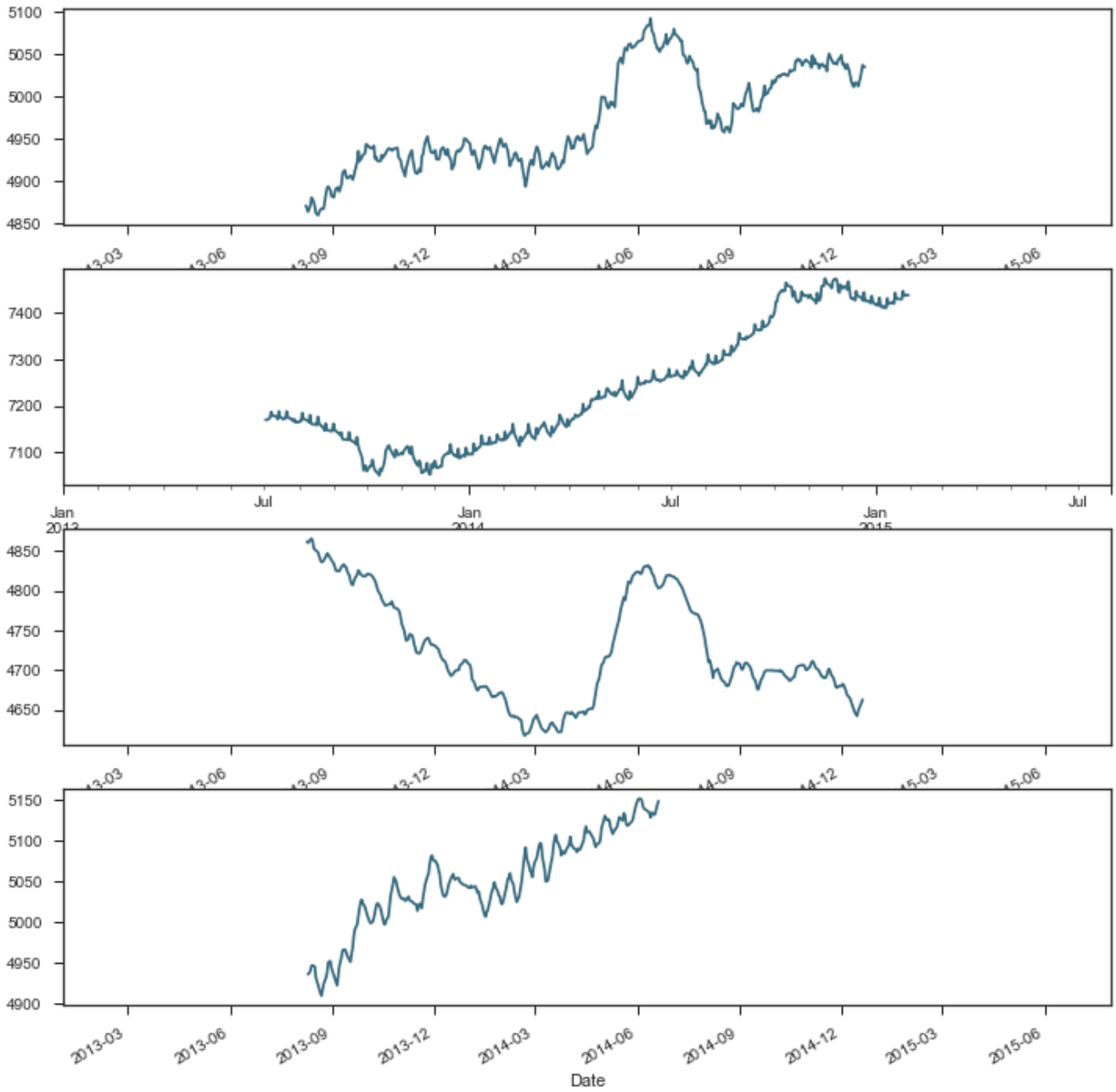
In [545]:

```
f, (ax1, ax2, ax3, ax4) = plt.subplots(4, figsize = (12, 13))
```

```
# monthly
decomposition_a = seasonal_decompose(sales_a, model = 'additive', freq = 365)
decomposition_a.trend.plot(color = c, ax = ax1)

decomposition_b = seasonal_decompose(sales_b, model = 'additive', freq = 365)
decomposition_b.trend.plot(color = c, ax = ax2)

decomposition_c = seasonal_decompose(sales_c, model = 'additive', freq = 365)
decomposition_c.trend.plot(color = c, ax = ax3)

decomposition_d = seasonal_decompose(sales_d, model = 'additive', freq = 365)
decomposition_d.trend.plot(color = c, ax = ax4)
```

Out[545]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x22c160978>
```



Overall sales seems to increase, however not for the `StoreType` C (a third from the top). Eventhough the `StoreType` A is the most selling store type in the dataset, it seems that it cab follow the same decresing trajectory as `StoreType` C did.

Autocorrelaion

The next step in ourtime series analysis is to review Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots.

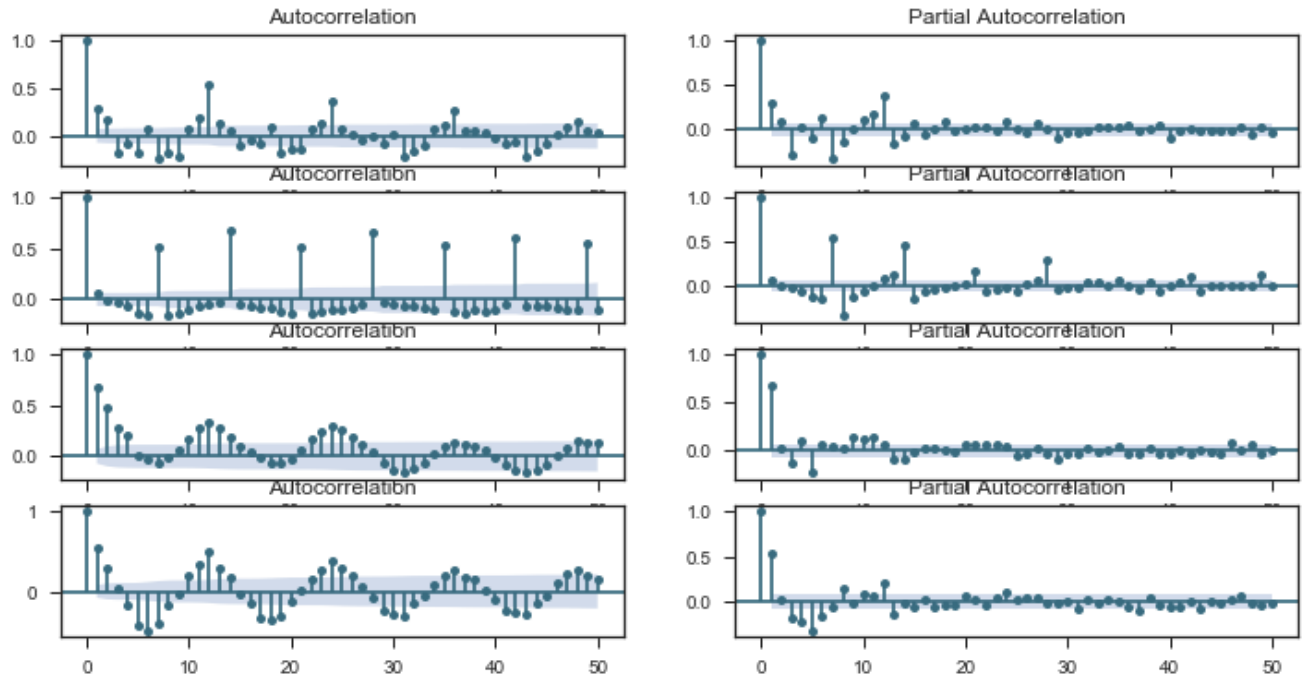
Function (PACF) plots:

ACF is a measure of the correlation between the timeseries with a lagged version of itself. For instance at lag 5, ACF would compare series at time instant 't1'…'tn' with series at instant 't1-5'…'tn-5' (t1-5 and tn being end points).

PACF, on the other hand, measures the correlation between the timeseries with a lagged version of itself but after eliminating the variations explained by the intervening comparisons. Eg. at lag 5, it will check the correlation but remove the effects already explained by lags 1 to 4.

In [551]:

```
# figure for subplots
plt.figure(figsize = (12, 8))

# acf and pacf for A
plt.subplot(421); plot_acf(sales_a, lags = 50, ax = plt.gca(), color = c)
plt.subplot(422); plot_pacf(sales_a, lags = 50, ax = plt.gca(), color = c)

# acf and pacf for B
plt.subplot(423); plot_acf(sales_b, lags = 50, ax = plt.gca(), color = c)
plt.subplot(424); plot_pacf(sales_b, lags = 50, ax = plt.gca(), color = c)

# acf and pacf for C
plt.subplot(425); plot_acf(sales_c, lags = 50, ax = plt.gca(), color = c)
plt.subplot(426); plot_pacf(sales_c, lags = 50, ax = plt.gca(), color = c)

# acf and pacf for D
plt.subplot(427); plot_acf(sales_d, lags = 50, ax = plt.gca(), color = c)
plt.subplot(428); plot_pacf(sales_d, lags = 50, ax = plt.gca(), color = c)

plt.show()
```



We can read these plots horizontally. Each horizontal pair is for one 'StoreType', from A to D. In general, those plots are showing the correlation of the series with itself, lagged by x time units correlation of the series with itself, lagged by x time units.

There is at two things common for each pair of plots: non randomnes of the time series and high lag-1 (which will probably need a higher order of differencing d/D).

- **Type A and type B: Both types show seasonalities at certain lags. For type A, it is each 12th observation with positives spikes at the 12 (s) and 24(2s) lags and so on. For type B it's a weekly trend with positives spikes at the 7(s), 14(2s), 21(3s) and 28(4s) lags.**
- **Type C and type D: Plots of these two types are more complex. It seems like each observation is coorrelated to its adjacent observations.**

Time Series Analysis and Forecasting with Prophet

Forecasting for the next 6 weeks for the first store

The Core Data Science team at Facebook recently published a new procedure for forecasting time series data called Prophet. It is based on an additive model where non-linear trends are fit with yearly and weekly seasonality, plus holidays. It enables performing automated forecasting which are already implemented in R at scale in Python 3.

In [69]:

```
# importing data
df = pd.read_csv("~/Documents/projects/rossmann/train.csv",
                 low_memory = False)

# remove closed stores and those with no sales
df = df[(df["Open"] != 0) & (df['Sales'] != 0)]

# sales for the store number 1 (StoreType C)
sales = df[df.Store == 1].loc[:, ['Date', 'Sales']]

# reverse to the order: from 2013 to 2015
sales = sales.sort_index(ascending = False)

# to datetime64
sales['Date'] = pd.DatetimeIndex(sales['Date'])
sales.dtypes
```

Out[69]:

```
Date     datetime64[ns]
Sales             int64
dtype: object
```

In [70]:

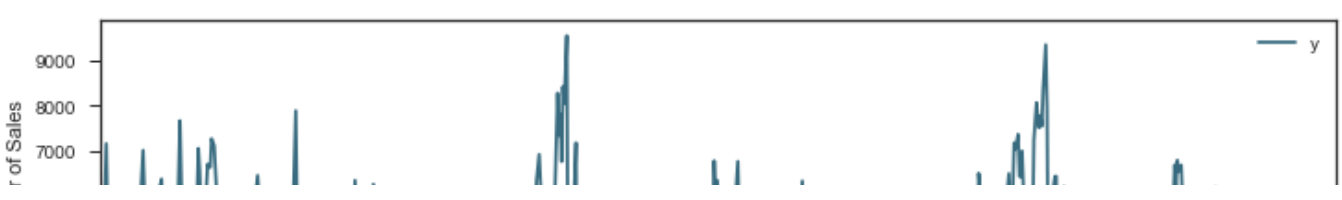
```
# from the prophet documentation every variables should have specific names
sales = sales.rename(columns = {'Date': 'ds',
                                'Sales': 'y'})
sales.head()
```

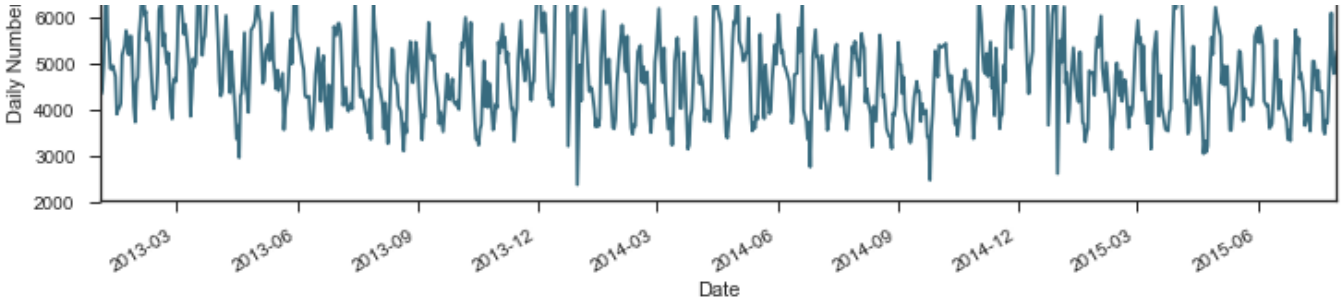
Out[70]:

	ds	y
1014980	2013-01-02	5530
1013865	2013-01-03	4327
1012750	2013-01-04	4486
1011635	2013-01-05	4997
1009405	2013-01-07	7176

In [71]:

```
# plot daily sales
ax = sales.set_index('ds').plot(figsize = (12, 4), color = c)
ax.set_ylabel('Daily Number of Sales')
ax.set_xlabel('Date')
plt.show()
```





Modeling Holidays

Prophet also allows to model for holidays, and that's what we do here.

The StateHoliday variable in the dataset indicates a state holiday, at which all stores are normally closed. There are also school holidays in the dataset at which ceratin stores are also closing their doors.

In [72]:

```
# create holidays dataframe
state_dates = df[(df.StateHoliday == 'a') | (df.StateHoliday == 'b') & (df.StateHoliday
== 'c')].loc[:, 'Date'].values
school_dates = df[df.SchoolHoliday == 1].loc[:, 'Date'].values

state = pd.DataFrame({'holiday': 'state_holiday',
                      'ds': pd.to_datetime(state_dates)})
school = pd.DataFrame({'holiday': 'school_holiday',
                       'ds': pd.to_datetime(school_dates)})

holidays = pd.concat((state, school))
holidays.head()
```

Out[72]:

	ds	holiday
0	2015-06-04	state_holiday
1	2015-06-04	state_holiday
2	2015-06-04	state_holiday
3	2015-06-04	state_holiday
4	2015-06-04	state_holiday

In [73]:

```
# set the uncertainty interval to 95% (the Prophet default is 80%)
my_model = Prophet(interval_width = 0.95,
                   holidays = holidays)
my_model.fit(sales)

# dataframe that extends into future 6 weeks
future_dates = my_model.make_future_dataframe(periods = 6*7)

print("First week to forecast.")
future_dates.tail(7)
```

```
First week to forecast.
```

Out[73]:

	ds
816	2015-09-05
817	2015-09-06
818	2015-09-07

	ds
819	2015-09-08
820	2015-09-09
821	2015-09-10
822	2015-09-11

In [75]:

```
# predictions
forecast = my_model.predict(future_dates)

# preditions for last week
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail(7)
```

Out[75]:

	ds	yhat	yhat_lower	yhat_upper
816	2015-09-05	4101.946814	2523.250390	5793.274809
817	2015-09-06	4084.055656	2318.985242	5720.277386
818	2015-09-07	4172.307411	2566.179681	5866.116669
819	2015-09-08	3674.492449	2138.620847	5236.075479
820	2015-09-09	3561.143037	1964.304227	5057.196170
821	2015-09-10	3471.802885	1786.679574	5192.860091
822	2015-09-11	3726.622495	2006.832681	5380.954021

The forecast object here is a new dataframe that includes a column yhat with the forecast, as well as columns for components and uncertainty intervals.

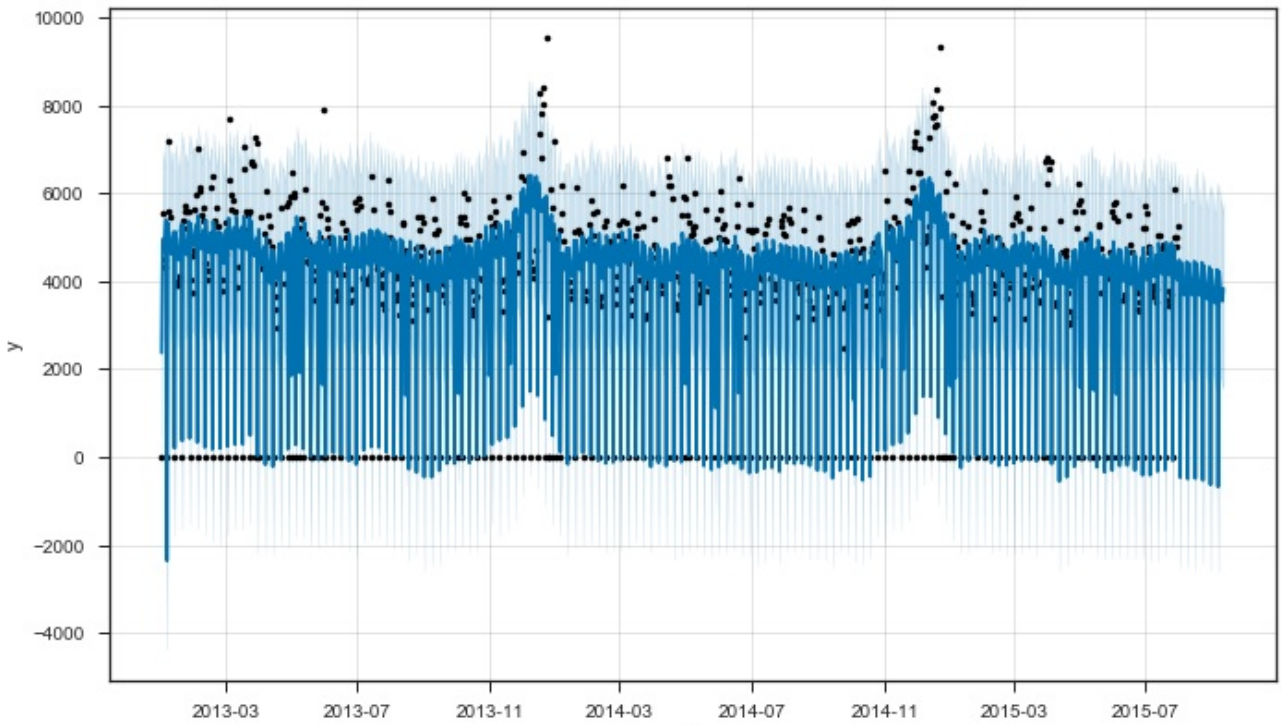
In [ ]:

```
fc = forecast[['ds', 'yhat']].rename(columns = {'Date': 'ds', 'Forecast': 'yhat'})
```

Prophet plots the observed values of our time series (the black dots), the forecasted values (blue line) and the uncertainty intervals of our forecasts (the blue shaded regions).

In [52]:

```
# visualizing predicions
my_model.plot(forecast);
```



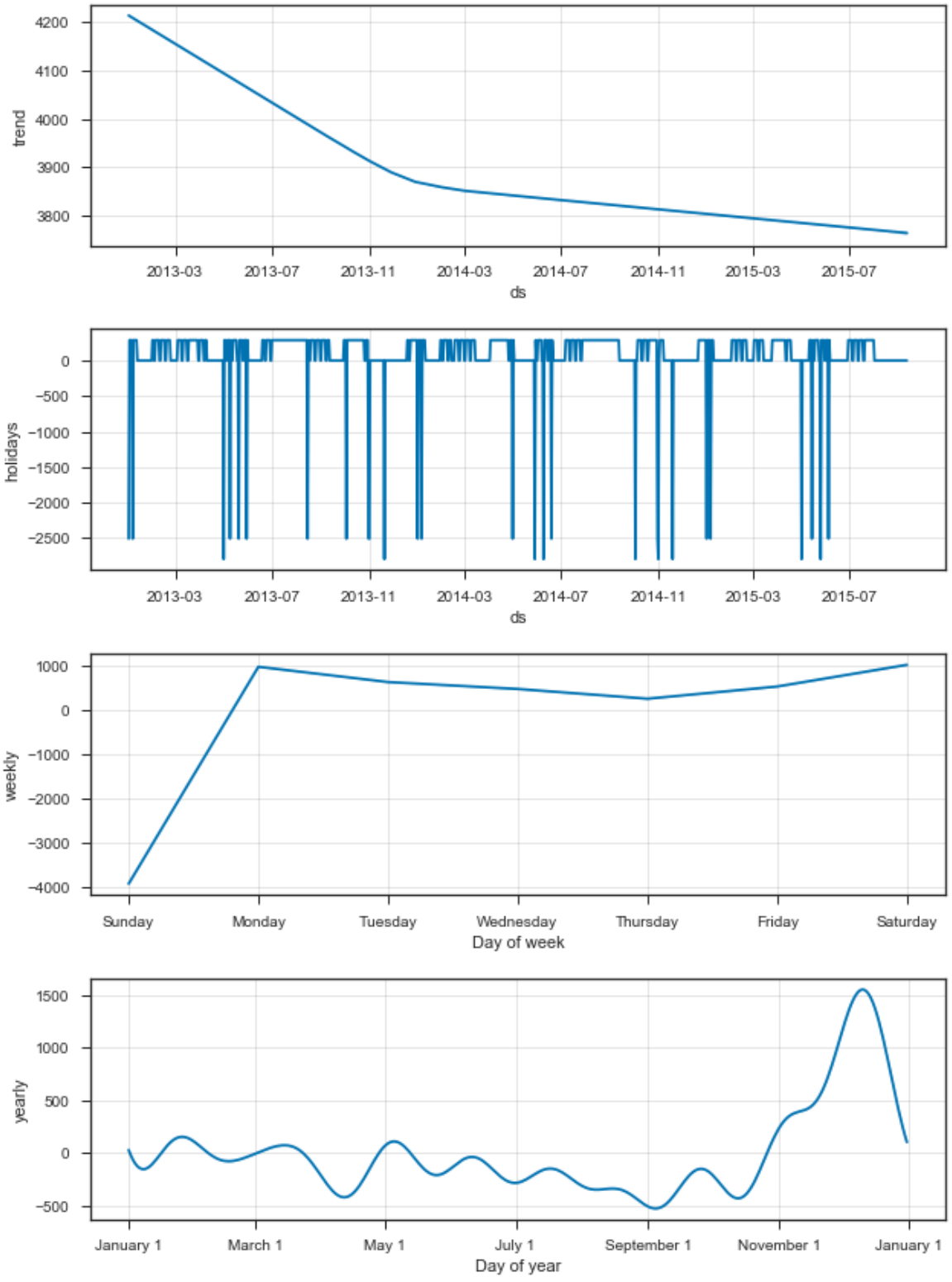
ds

As we see Prophet catches the trends and most of the time gets future values right.

One other particularly strong feature of Prophet is its ability to return the components of our forecasts. This can help reveal how daily, weekly and yearly patterns of the time series plus manyally included holidayes contribute to the overall forecasted values:

In [47]:

```
my_model.plot_components(forecast);
```



The first plot shows that the monthly sales of store number 1 has been linearly decreasing over time and the second shows the holiays gaps included in the model. The third plot highlights the fact that the weekly volume of last week sales peaks towards the Monday of the next week, while the forth plot shows that the most buzy season occurs during the Christmas holidays.

Conclusion of Time Series forecasting

During this part we discussed time series analysis with `.seasonal_decompose()`, `ACF` and `PCF` plots and fited forecasting model using a new procedure by Facebook `Prophet`.

We can now present main advantages and drawbacks of time series forecasting:

Advantages

- **Powerfull tool for the time series forecasting as it accounts for time dependencies, seasonalities and holidays (Prophet: manualy).**
- **Easily implemented with R `auto.arima()` from `forecast` package, which runs a complex grid search and sophisticated algorythm behind the scene.**

Drawbacks

- **Doesn't catch interactions between external features, which could improve the forecasting power of a model. In our case, these variables are `Promo` and `CompetitionOpen`.**
- **Eventhough Prophet offers an automated solution for ARIMA, this methodology is under development and not completely stable.**
- **Fitting seasonal ARIMA model needs 4 to 5 whole seasons in the dataset, which can be the the biggest drawback for new companies.**
- **Seasonal ARIMA in Python has 7 hyperparameters which can be tuned only manually affecting significantly the speed of the forecasting process.**

Alternative Approach: Regression XGBoost

XGBoost is an implementation of Gradient Boosted Decision trees designed for speed and performance. Its more suitable name is a as regularized Gradient Boosting, as it uses a more regularized model formalization to control over-fitting.

Additional advantages of this algorythm are:

- **Automated missing values handling: XGB uses a "learned" default direction for the missing values. "Learned" means learned in the tree construction process by choosing the best direction that optimizes the training loss.**
- **Interactive feature analysis (yet implemented only in R): plots the structure of decision trees with splits and leaves.**
- **Feature importance analysis: a sorted barplot of the most significant variables.**

As we already saw in the previos section our data is higly seasonal and not random (dependent). Therefore, before fitting any models we need to "smooth" target variable Sales. The typical preprocessing step is to log transform the data in question. Once we perform the forecasting we will unwind log transformations in reverse order.

Quick Run through

In [437]:

```
# to predict to
test = pd.read_csv("~/Documents/projects/rossmann/test.csv",
                   parse_dates = True, low_memory = False, index_col = 'Date')
test.head()
```

Out[437]:

Date	Id	Store	DayOfWeek	Open	Promo	StateHoliday	SchoolHoliday
2015-09-17	1	1	4	1.0	1	0	0
2015-09-17	2	3	4	1.0	1	0	0
2015-09-17	3	7	4	1.0	1	0	0
2015-09-17	4	8	4	1.0	1	0	0
2015-09-17	5	9	4	1.0	1	0	0

The Id variable represents a (Store, Date) duple within the test set.

In [138]:

```
# test: missing values?
test.isnull().sum()
```

Out[138]:

```
Id                0
Store             0
DayOfWeek         0
Open             11
Promo             0
StateHoliday      0
SchoolHoliday     0
dtype: int64
```

In [139]:

```
test[pd.isnull(test.Open)]
```

Out[139]:

Date	Id	Store	DayOfWeek	Open	Promo	StateHoliday	SchoolHoliday
2015-09-17	480	622	4	NaN	1	0	0
2015-09-16	1336	622	3	NaN	1	0	0
2015-09-15	2192	622	2	NaN	1	0	0
2015-09-14	3048	622	1	NaN	1	0	0
2015-09-12	4760	622	6	NaN	0	0	0
2015-09-11	5616	622	5	NaN	0	0	0
2015-09-10	6472	622	4	NaN	0	0	0
2015-09-09	7328	622	3	NaN	0	0	0
2015-09-08	8184	622	2	NaN	0	0	0
2015-09-07	9040	622	1	NaN	0	0	0
2015-09-05	10752	622	6	NaN	0	0	0

We see that these stores should be normally opened. Let's assume that they are then.

In [438]:

```
# replace NA's in Open variable by 1
test.fillna(1, inplace = True)
```

Data Encoding

XGBoost doesn't support anything else than numbers. So prior to modeling we need to encode certain factor variables into numerical plus extract dates as we did before for the train set.

In [439]:

```
# data extraction
test['Year'] = test.index.year
test['Month'] = test.index.month
test['Day'] = test.index.day
test['WeekOfYear'] = test.index.weekofyear

# to numerical
mappings = {'0':0, 'a':1, 'b':2, 'c':3, 'd':4}
test.StateHoliday.replace(mappings, inplace = True)

train_store.Assortment.replace(mappings, inplace = True)
train_store.StoreType.replace(mappings, inplace = True)
train_store.StateHoliday.replace(mappings, inplace = True)
train_store.drop('PromoInterval', axis = 1, inplace = True)

store.StoreType.replace(mappings, inplace = True)
store.Assortment.replace(mappings, inplace = True)
store.drop('PromoInterval', axis = 1, inplace = True)
```

Returning back to the `train_store` data:

In [227]:

```
# take a look on the train and store again
train_store.head()
```

Out[227]:

	Store	DayOfWeek	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday	Year	Month	...	StoreType	Assortment
0	1	5	5263	555	1	1	0	1	2015	7	...	3	1
1	1	4	5020	546	1	1	0	1	2015	7	...	3	1
2	1	3	4782	523	1	1	0	1	2015	7	...	3	1
3	1	2	5011	560	1	1	0	1	2015	7	...	3	1
4	1	1	6102	612	1	1	0	1	2015	7	...	3	1

5 rows × 23 columns

Let's merge `test` and `store` data too:

In [441]:

```
print("Joining test set with an additional store information.")
test_store = pd.merge(test, store, how = 'inner', on = 'Store')

test_store['CompetitionOpen'] = 12 * (test_store.Year - test_store.CompetitionOpenSinceY
ear) + (test_store.Month - test_store.CompetitionOpenSinceMonth)
test_store['PromoOpen'] = 12 * (test_store.Year - test_store.Promo2SinceYear) + (test_st
ore.WeekOfYear - test_store.Promo2SinceWeek) / 4.0

print("In total: ", test_store.shape)
test_store.head()
```

```
Joining test set with an additional store information.
In total:  (41088, 21)
```

Out[441]:

	Id	Store	DayOfWeek	Open	Promo	StateHoliday	SchoolHoliday	Year	Month	Day	...	StoreType	Assortment	Comp
0	1	1	4	1.0	1	0	0	2015	9	17	...	3	1	
1	857	1	3	1.0	1	0	0	2015	9	16	...	3	1	

	Id	Store	DayOfWeek	Open	Promo	StateHoliday	SchoolHoliday	Year	Month	Day	...	StoreType	Assortment	Comp
2	1713	1	2	1.0	1	0	0	2015	9	15	...	3	1	
3	2569	1	1	1.0	1	0	0	2015	9	14	...	3	1	
4	3425	1	7	0.0	0	0	0	2015	9	13	...	3	1	

5 rows × 21 columns

Model Training

Approach

1. Split train data to train and test set to evaluate the model.
2. Set `eta` to a relatively high value (e.g. 0.05 ~ 0.1), num_round to 300 ~ 500
3. Use grid search to find the best combination of additional parameters.
4. Lower `eta` until we reach the optimum.
5. Use the validation set as watchlist to retrain the model with the best parameters.

In [443]:

```
# split into training and evaluation sets
# excluding Sales and Id columns
predictors = [x for x in train_store.columns if x not in ['Customers', 'Sales', 'SalePer
Customer']]
y = np.log(train_store.Sales) # log transformation of Sales
X = train_store

# split the data into train/test set
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size = 0.3, # 30% for the eval
uation set
                                                    random_state = 42)
```

In [444]:

```
# predictors
X.columns
```

Out[444]:

```
Index(['Store', 'DayOfWeek', 'Sales', 'Customers', 'Open', 'Promo',
       'StateHoliday', 'SchoolHoliday', 'Year', 'Month', 'Day', 'WeekOfYear',
       'SalePerCustomer', 'StoreType', 'Assortment', 'CompetitionDistance',
       'CompetitionOpenSinceMonth', 'CompetitionOpenSinceYear', 'Promo2',
       'Promo2SinceWeek', 'Promo2SinceYear', 'CompetitionOpen', 'PromoOpen'],
      dtype='object')
```

In [445]:

```
# evaluation metric: rmspe
# Root Mean Square Percentage Error
# code chunk shared at Kaggle

def rmspe(y, yhat):
    return np.sqrt(np.mean((yhat / y-1) ** 2))

def rmspe_xg(yhat, y):
    y = np.expm1(y.get_label())
    yhat = np.expm1(yhat)
    return "rmspe", rmspe(y, yhat)
```

Tuning Parameters

- `eta`: Step size used in updating weights. Lower value means slower training but better convergence.
- `num_round`: Total number of iterations

- `num_round`: **Total number of iterations.**
- `subsample`: **The ratio of training data used in each iteration; combat overfitting. Should be configured in the range of 30% to 80% of the training dataset, and compared to a value of 100% for no sampling.**
- `colsample_bytree`: **The ratio of features used in each iteration, default 1.**
- `max_depth`: **The maximum depth of each tree. If we do not limit max depth, gradient boosting would eventually overfit.**
- `early_stopping_rounds`: **If there's no increase in validation score for a given number of iterations, the algorithm will stop early, also combats overfitting.**

In [446]:

```
# base parameters
params = {
    'booster': 'gbtree',
    'objective': 'reg:linear', # regression task
    'subsample': 0.8, # 80% of data to grow trees and prevent overfitting
    'colsample_bytree': 0.85, # 85% of features used
    'eta': 0.1,
    'max_depth': 10,
    'seed': 42} # for reproducible results
```

In [ ]:

```
# XGB with xgboost library
dtrain = xgb.DMatrix(X_train[predictors], y_train)
dtest = xgb.DMatrix(X_test[predictors], y_test)

watchlist = [(dtrain, 'train'), (dtest, 'test')]

xgb_model = xgb.train(params, dtrain, 300, evals = watchlist,
                      early_stopping_rounds = 50, feval = rmspe_xg, verbose_eval = True)
```

Last five rows:

```
[295]   train-rmspe:0.106959    test-rmspe:0.111575
[296]   train-rmspe:0.106855    test-rmspe:0.111498
[297]   train-rmspe:0.106467    test-rmspe:0.111439
[298]   train-rmspe:0.106348    test-rmspe:0.111331
[299]   train-rmspe:0.105759    test-rmspe:0.111298
```

Essentially, we want the least value. The model with base hyperparameters gives out better resukt on the train set, indicating to the overfitting issue.

Grid Search from sklearn

Scikit learn wrapper is famous for the `GridSearchCV` and `RandomizedSearchCV`. Between these two, most of the time the preference leans towards `RandomnizedSearchCV`, faster version of `GridSearchCV`.

As an input, `RandomnizedSearchCV` takes only sklearn wrapper of XGboost, so instead of using the first version of a model, we build the analogous model in sklearn with `XGBRegressor`.

In [449]:

```
# XGB with sklearn wrapper
# the same parameters as for xgboost model
params_sk = {'max_depth': 10,
             'n_estimators': 300, # the same as num_rounds in xgboost
             'objective': 'reg:linear',
             'subsample': 0.8,
             'colsample_bytree': 0.85,
             'learning_rate': 0.1,
             'seed': 42}
```

```
skrg = XGBRegressor(**params_sk)

skrg.fit(X_train, y_train)
```

Out[449]:

```
XGBRegressor(base_score=0.5, colsample_bylevel=1, colsample_bytree=0.85,
       gamma=0, learning_rate=0.1, max_delta_step=0, max_depth=10,
       min_child_weight=1, missing=None, n_estimators=300, nthread=-1,
       objective='reg:linear', reg_alpha=0, reg_lambda=1,
       scale_pos_weight=1, seed=42, silent=True, subsample=0.8)
```

For parameters we will specify the regularization parameter `reg_alpha` which reduce model complexity and enhance performance, as well as `gamma` parameter which represents the minimum loss reduction required to make a split and also `max_depth` used to control over-fitting.

In [450]:

```
import scipy.stats as st

params_grid = {
    'learning_rate': st.uniform(0.01, 0.3),
    'max_depth': list(range(10, 20, 2)),
    'gamma': st.uniform(0, 10),
    'reg_alpha': st.expon(0, 50)}

search_sk = RandomizedSearchCV(skrg, params_grid, cv = 5) # 5 fold cross validation
search_sk.fit(X_train, y_train)

# best parameters
print(search_sk.best_params_); print(search_sk.best_score_)
```

```
{'gamma': 0.80198330585415034, 'learning_rate': 0.044338624448041611, 'max_depth': 16, 'r
eg_alpha': 23.008226565535971}
0.999596090945
```

In [ ]:

```
# with new parameters
params_new = {
    'booster': 'gbtree',
    'objective': 'reg:linear',
    'subsample': 0.8,
    'colsample_bytree': 0.85,
    'eta': 0.044338624448041611,
    'max_depth': 16,
    'gamma': 0.80198330585415034,
    'reg_alpha': 23.008226565535971,
    'seed': 42}

model_final = xgb.train(params_new, dtrain, 300, evals = watchlist,
                        early_stopping_rounds = 50, feval = rmspe_xg, verbose_eval = Tru
e)
```

Last five rows:

```
[295]   train-rmspe:0.213295    test-rmspe:0.147425
[296]   train-rmspe:0.213214    test-rmspe:0.147367
[297]   train-rmspe:0.213061    test-rmspe:0.147199
[298]   train-rmspe:0.213084    test-rmspe:0.14701
[299]   train-rmspe:0.212913    test-rmspe:0.146808
```

We resolved an issue with overfitting, but due to the decrease of learning rate (`eta`) we got a bit worse overall score on the test set (~0.11 to ~0.14).

In [453]:

```
yhat = model_final.predict(xgb.DMatrix(X_test[predictors]))
```

```
yhat = model_final.predict(xgb.DMatrix(X_test[predictors]))
error = rmspe(X_test.Sales.values, np.exp(yhat))

print('First validation yelds RMSPE: {:.6f}'.format(error))
```

```
First validation yelds RMSPE: 0.146761
```

Even though we observed a bit higher RMSPE value, we need to remember that the corresponding `eta` for the first version was 0.1, which is usually considered as high. I think that it's remarkable that we got more or less the same result but with 2x lower `eta` (0.1 to 0.04).

Next steps towards model's improvement will involve further decrease of `eta`, tuning of correpsonding `gamma` and `max_depth`.

Model understanding

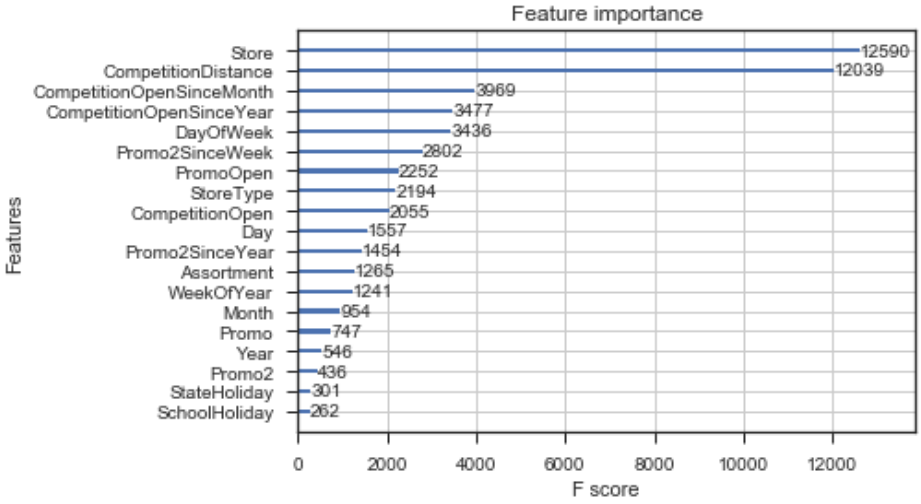
Feature importance scores help us see which variables contributed the most to the score.

In [454]:

```
xgb.plot_importance(model_final)
```

Out[454]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x117a35748>
```



Variables `Store` and `CompetitionDistance` are both leading. Then go features `CompetitionOpenSinceMonth`, `CompetitionOpenSinceYear`, `DayOfWeek`, `PromoSinceWeek` and deducted features `PromoOpen`.

Prediction to unseen data

In [455]:

```
# predictions to unseen data
unseen = xgb.DMatrix(test_store[predictors])
test_p = model_final.predict(unseen)

forecasts = pd.DataFrame({'Id': test['Id'],
                          'Sales': np.exp(test_p)})
# forecasts
forecasts.head()
```

Out[455]:

	Id	Sales
Date		

Date	Id	Sales
2015-09-17	1	4428.240723
2015-09-17	2	4577.308105
2015-09-17	3	4895.477051
2015-09-17	4	5571.973145
2015-09-17	5	4961.958496

Final thoughts

Time Series Analysis is a must for time series data. It goes much deeper than ad-hoc Exploratory Data Analysis, revealing trends, non randomness of the data and seasonalities.

I was particularly excited to use a new forecasting procedure `Prophet`. Eventhough this tool is still under development, it has everything set for the advanced modeling as it can account for change points in trends and holidays in the data. In the meantime, the most sophisticated tool for the Time Series Analysis stays `auto.arima` from R `forecast` package.

A significant jump in the forecasting performance of the model fitted above, XGboost with xgboost library, can be achieved by increasing the number and range of hyperparameters. Due to the number of observations (800k) and with a laptop like mine, the "more developped" grid search would take about 2-3 days to fit. So I left this room for the improvement for later notebooks.

Another method that I didn't cover here is a regression model Stacking, which works great for small or medium size data sets. We would basically combine XGboost, RandomForest, NN and SVM for regression. And then stack them together by building the final model.

Concerning the XGboost model, recently Microsoft open sourced LightGBM, a potentially better library than XGboost. For the next project, I will try it out.

Thank you for reading!

P.S. Submission

As it is a Kaggle competition, I made two submissions to leaderboard: forecast from the base and tuned model.

In [456]:

```
# first
# 0.66419
test_base = xgb_model.predict(unseen)

forecasts_base = pd.DataFrame({'Id': test['Id'],
                               'Sales': np.exp(test_base)})
forecasts_base.to_csv("xgboost_2_submission.csv", index = False)
```

In [ ]:

```
# final
# 0.60553
forecasts.to_csv("xgboost_submission.csv", index = False)
```

The score with four tuned parameters improved from 0.66 to 0.60 on the Public Board and from 0.62 to 0.68 on the Provate Board (test set reserved by Kaggle itself) .

These scores are still quite low, but it's a good start for further improvement.