DATA DRIVEN GROWTH WITH PYTHON

Predicting Sales

Forecasting the monthly sales with LSTM

Barış Karaman Follow
Jun 9, 2019 · 8 min read ★

This series of articles was designed to explain how to use Python in a simplistic way to fuel your company's growth by applying the predictive approach to all your actions. It will be a combination of programming, data analysis, and machine learning.

I will cover all the topics in the following nine articles:

1- Know Your Metrics

2- Customer Segmentation

3- Customer Lifetime Value Prediction

4- Churn Prediction

5- Predicting Next Purchase Day

6- **Predicting Sales**

7- Market Response Models

8- Uplift Modeling

9- A/B Testing Design and Execution

Articles will have their own code snippets to make you easily apply them. If you are super new to programming, you can have a good introduction for Python and Pandas (a famous library that we will use on everything) here. But still without a coding introduction, you can learn the concepts, how to use your data and start generating value out of it:

> Sometimes you gotta run before you can walk — Tony Stark

As a pre-requisite, be sure Jupyter Notebook and Python are installed on your computer. The code snippets will run on Jupyter Notebook only.

Alright, let's start.

Part 6: Predicting Sales

Before this section, almost all our prediction models were on customer level (e.g. churn prediction, next purchase day, etc.). It is useful to zoom out and look at the broader picture as well. By considering all our efforts on the customer side, how do we affect the sales?

Time series forecasting is one of the major building blocks of Machine Learning. There are many methods in the literature to achieve this like Autoregressive Integrated Moving Average (ARIMA), Seasonal Autoregressive Integrated Moving-Average (SARIMA), Vector Autoregression (VAR), and so on.

In this article, we will focus on Long Short-term Memory (LSTM) method, which is a quite popular one if you want to use Deep Learning. We will use Keras in our project to implement LSTM.

Lastly, how does knowing the future sales helps our business?

First of all, it is a benchmark. We can use it as the business as usual level we are going to achieve if nothing changes in our strategy. Moreover, we can calculate the incremental value of our new actions on top of this benchmark.

Second, it can be utilized for planning. We can plan our demand and supply actions by looking at the forecasts. It helps to see where to invest more.

Last but not least, it is an excellent guide for planning budgets and targets.

Now it is time to jump into coding and build our first deep learning model. The implementation of our model will have 3 steps:

- Data Wrangling
- Data Transformation to make it stationary and supervised
- Building the LSTM model & evaluation

Data Wrangling

In this example, we use the dataset from a Kaggle competition. It represents the daily sales for each store and item.

Like always we start with importing the required libraries and importing our data from CSV:

```
from datetime import datetime, timedelta,date
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from __future__ import division

import warnings
warnings.filterwarnings("ignore")

import plotly.plotly as py
import plotly.offline as pyoff
import plotly.graph_objs as go

#import Keras
import keras
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
from keras.utils import np_utils
from keras.layers import LSTM
from sklearn.model_selection import KFold, cross_val_score, train_test_split

#initiate plotly
pyoff.init_notebook_mode()

#read the data in csv
df_sales = pd.read_csv('sales_data.csv')

#convert date field from string to datetime
df_sales['date'] = pd.to_datetime(df_sales['date'])

#show first 10 rows
df_sales.head(10)
```

```
df_sales.head(10)
```

q6 intro.py hosted with ♡ by GitHub view raw

Our data looks like below:

```
df_sales.head(10)
```

	date	store	item	sales
0	2013-01-01	1	1	13
1	2013-01-01	1	1	11
2	2013-01-01	1	1	14
3	2013-01-01	1	1	13
4	2013-01-01	1	1	10
5	2013-01-01	1	1	12
6	2013-01-01	1	1	10
7	2013-01-01	1	1	9
8	2013-01-01	1	1	12
9	2013-01-01	1	1	9

Our task is to forecast monthly total sales. We need to aggregate our data at the monthly level and sum up the *sales* column.

```
#represent month in date field as its first day
df_sales['date'] = df_sales['date'].dt.year.astype('str') + '-' +
df_sales['date'].dt.month.astype('str') + '-01'
df_sales['date'] = pd.to_datetime(df_sales['date'])

#groupby date and sum the sales
df_sales = df_sales.groupby('date').sales.sum().reset_index()
```

After applying the code above, **df_sales** is now showing the aggregated sales we need:

```
df_sales.head()
```

	date	sales
0	2013-01-01	454904
1	2013-02-01	459417
2	2013-03-01	617382
3	2013-04-01	682274
4	2013-05-01	763242

Data Transformation

To model our forecast easier and more accurate, we will do the transformations below:

- We will convert the data to stationary if it is not
- Converting from time series to supervised for having the feature set of our LSTM model

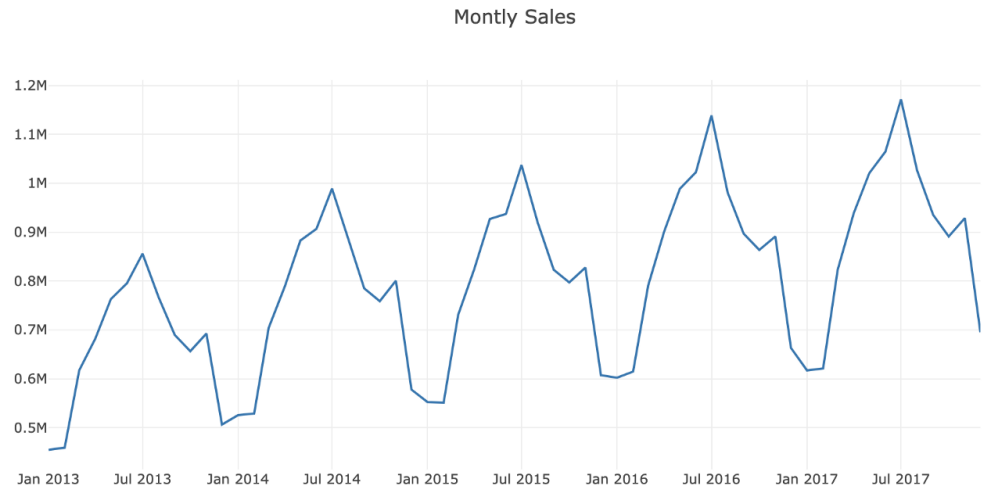
- Scale the data

First off, how do we check if the data is not stationary? Let's plot it and see:

```
#plot monthly sales
plot_data = [
    go.Scatter(
        x=df_sales['date'],
        y=df_sales['sales'],
    )
]

plot_layout = go.Layout(
        title='Montly Sales'
    )
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
```

Monthly sales chart:



Monthly Sales — not stationary

Obviously, it is not stationary and has an increasing trend over the months. One method is to get the difference in sales compared to the previous month and build the model on it:

```
#create a new dataframe to model the difference
df_diff = df_sales.copy()

#add previous sales to the next row
df_diff['prev_sales'] = df_diff['sales'].shift(1)

#drop the null values and calculate the difference
df_diff = df_diff.dropna()
df_diff['diff'] = (df_diff['sales'] - df_diff['prev_sales'])

df_diff.head(10)
```

Now we have the required dataframe for modeling the difference:

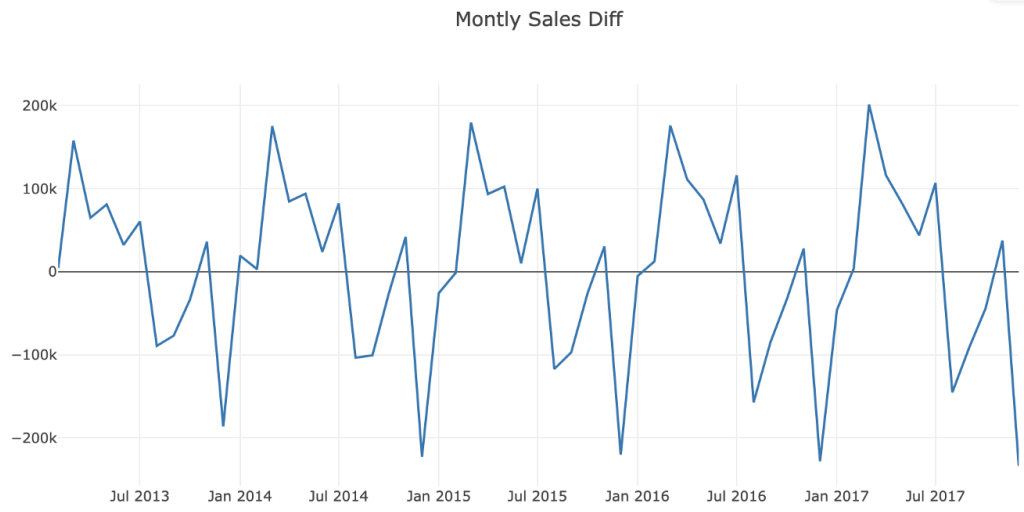
```
df_diff.head(10)
```

	date	sales	prev_sales	diff
1	2013-02-01	459417	454904.0	4513.0
2	2013-03-01	617382	459417.0	157965.0
3	2013-04-01	682274	617382.0	64892.0
4	2013-05-01	763242	682274.0	80968.0
5	2013-06-01	795597	763242.0	32355.0
6	2013-07-01	855922	795597.0	60325.0
7	2013-08-01	766761	855922.0	-89161.0
8	2013-09-01	689907	766761.0	-76854.0
9	2013-10-01	656587	689907.0	-33320.0
10	2013-11-01	692643	656587.0	36056.0

Let's plot it and check if it is stationary now:

```
#plot sales diff
plot_data = [
    go.Scatter(
        x=df_diff['date'],
        y=df_diff['diff'],
    )
]

plot_layout = go.Layout(
        title='Montly Sales Diff'
    )
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
```



Monthly Sales Difference — stationary

Perfect! Now we can start building our feature set. We need to use previous monthly sales data to forecast the next ones. The look-back period may vary for every model. Ours will be 12 for this example.

So what we need to do is to create columns from lag_1 to lag_12 and assign values by using **shift()** method:

```
#create dataframe for transformation from time series to supervised
df_supervised = df_diff.drop(['prev_sales'],axis=1)

#adding lags
for inc in range(1,13):
    field_name = 'lag_' + str(inc)
    df_supervised[field_name] = df_supervised['diff'].shift(inc)

#drop null values
df_supervised = df_supervised.dropna().reset_index(drop=True)
```

Check out our new dataframe called df_supervised:

```
df_supervised.head(10)
```

	date	sales	diff	lag_1	lag_2	lag_3	lag_4	lag_5	lag_6	lag_7	lag_8	lag_9	lag_10	lag_11	lag_12
0	2014-02-01	529117	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0	32355.0	80968.0	64892.0	157965.0	4513.0
1	2014-03-01	704301	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0	32355.0	80968.0	64892.0	157965.0
2	2014-04-01	788914	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0	32355.0	80968.0	64892.0
3	2014-05-01	882877	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0	32355.0	80968.0
4	2014-06-01	906842	23965.0	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0	32355.0
5	2014-07-01	989010	82168.0	23965.0	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0
6	2014-08-01	885596	-103414.0	82168.0	23965.0	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0
7	2014-09-01	785124	-100472.0	-103414.0	82168.0	23965.0	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0
8	2014-10-01	758883	-26241.0	-100472.0	-103414.0	82168.0	23965.0	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0
9	2014-11-01	800783	41900.0	-26241.0	-100472.0	-103414.0	82168.0	23965.0	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0

We have our feature set now. Let's be a bit more curious and ask this question:

How useful are our features for prediction?

Adjusted R-squared is the answer. It tells us how good our features explain the variation in our label (lag_1 to lag_12 for diff, in our example).

Let's see it in an example:

```
# Import statsmodels.formula.api
import statsmodels.formula.api as smf

# Define the regression formula
model = smf.ols(formula='diff ~ lag_1', data=df_supervised)
```

```
# Fit the regression
model_fit = model.fit()

# Extract the adjusted r-squared
regression_adj_rsq = model_fit.rsquared_adj
print(regression_adj_rsq)
```

So what happened above?

Basically, we fit a linear regression model (OLS — Ordinary Least Squares) and calculate the Adjusted R-squared. For the example above, we just used **lag_1** to see how much it explains the variation in column **diff**. The output of this code block is:

lag_1 explains 3% of the variation. Let's check out others:

Adding four more features increased the score from 3% to 44%.

How is the score if we use the entire feature set:

The result is impressive as the score is 98%. Now we can confidently build our model after scaling our data. But there is one more step before scaling. We should split our data into train and test sets. As the test set, we have selected the last 6 months' sales.

```
#import MinMaxScaler and create a new dataframe for LSTM model
from sklearn.preprocessing import MinMaxScaler
df_model = df_supervised.drop(['sales','date'],axis=1)

#split train and test set
train_set, test_set = df_model[0:-6].values, df_model[-6:].values
```

As the scaler, we are going to use MinMaxScaler, which will scale each future between -1 and 1:

```
#apply Min Max Scaler
scaler = MinMaxScaler(feature_range=(-1, 1))
scaler = scaler.fit(train_set)
# reshape training set
train_set = train_set.reshape(train_set.shape[0], train_set.shape[1])
train_set_scaled = scaler.transform(train_set)

# reshape test set
test_set = test_set.reshape(test_set.shape[0], test_set.shape[1])
test_set_scaled = scaler.transform(test_set)
```

Building the LSTM model

Everything is ready to build our first deep learning model. Let's create feature and label sets from scaled datasets:

```
X_train, y_train = train_set_scaled[:, 1:], train_set_scaled[:, 0:1]
X_train = X_train.reshape(X_train.shape[0], 1, X_train.shape[1])

X_test, y_test = test_set_scaled[:, 1:], test_set_scaled[:, 0:1]
X_test = X_test.reshape(X_test.shape[0], 1, X_test.shape[1])
```

Let's fit our LSTM model:

```
model = Sequential()
model.add(LSTM(4, batch_input_shape=(1, X_train.shape[1],
X_train.shape[2]), stateful=True))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(X_train, y_train, nb_epoch=100, batch_size=1, verbose=1,
shuffle=False)
```

The code block above prints how the model improves itself and reduce the error in each epoch:

Let's do the prediction and see how the results look like:

```
y_pred = model.predict(X_test,batch_size=1)

#for multistep prediction, you need to replace X_test values with the
predictions coming from t-1
```


y_pred vs y_test

Results look similar but it doesn't tell us much because these are scaled data that shows the difference. How we can see the actual sales prediction?

First, we need to do the inverse transformation for scaling:

```
#reshape y_pred
y_pred = y_pred.reshape(y_pred.shape[0], 1, y_pred.shape[1])

#rebuild test set for inverse transform
pred_test_set = []
for index in range(0,len(y_pred)):
    print np.concatenate([y_pred[index],X_test[index]],axis=1)

pred_test_set.append(np.concatenate([y_pred[index],X_test[index]],axi
s=1))

#reshape pred_test_set
pred_test_set = np.array(pred_test_set)
pred_test_set = pred_test_set.reshape(pred_test_set.shape[0],
pred_test_set.shape[2])

#inverse transform
pred_test_set_inverted = scaler.inverse_transform(pred_test_set)
```

Second, we need to build the dataframe has the dates and the predictions. Transformed predictions are showing the difference. We should calculate the predicted sales numbers:

```
#create dataframe that shows the predicted sales
result_list = []
sales_dates = list(df_sales[-7:].date)
act_sales = list(df_sales[-7:].sales)
for index in range(0,len(pred_test_set_inverted)):
    result_dict = {}
    result_dict['pred_value'] = int(pred_test_set_inverted[index][0]
+ act_sales[index])
    result_dict['date'] = sales_dates[index+1]
    result_list.append(result_dict)
df_result = pd.DataFrame(result_list)

#for multistep prediction, replace act_sales with the predicted sales
```

Output:

Great! We've predicted the next six months' sales numbers. Let's check them in the plot to see how good is our model:

```
#merge with actual sales dataframe
df_sales_pred = pd.merge(df_sales,df_result,on='date',how='left')

#plot actual and predicted
plot_data = [
    go.Scatter(
        x=df_sales_pred['date'],
        y=df_sales_pred['sales'],
        name='actual'
    ),
        go.Scatter(
        x=df_sales_pred['date'],
        y=df_sales_pred['pred_value'],
        name='predicted'
    )

]
```

```
plot_layout = go.Layout(
        title='Sales Prediction'
    )
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
```

Actual vs predicted:

Looks pretty good for a simple model.

One improvement we can do for this model is to add holidays, breaks, and other seasonal effects. They can be simply added as a new feature.

By using this model, we have our baseline sales predictions. But how we can predict the effect of a promotion on sales? We will look into it in Part 7.

You can find the Jupyter Notebook for this article here.

. . .

To discuss growth marketing & data science, go ahead and book a free session with me here.

Machine Learning Data Driven Growth Growth Hacking Data Science Programming

About Help Legal