**Time Series Forecasting — A Getting Started Guide**

**An intro to forecasting with multivariate time series**

**Intro**

When I started writing this post I thought of just explaining how to do predictions with a “simple” time series (aka univariate time series). The challenging part of the project I was in, however, was the fact that the prediction needed to be made in conjunction with multiple variables. For this reason, I decided to bring this guide a little bit closer to reality and use a multivariate time series.

**Let’s get some concepts straight first…**

A [**Multivariate TS**](https://www.analyticsvidhya.com/blog/2018/09/multivariate-time-series-guide-forecasting-modeling-python-codes/) is a time series with more than one time-dependent variable. Each variable depends on its past values but also has some dependency on other variables. This dependency is taken into account when predicting values. These variables can be endogenous or exogenous. I will be focusing on exogenous variables here.

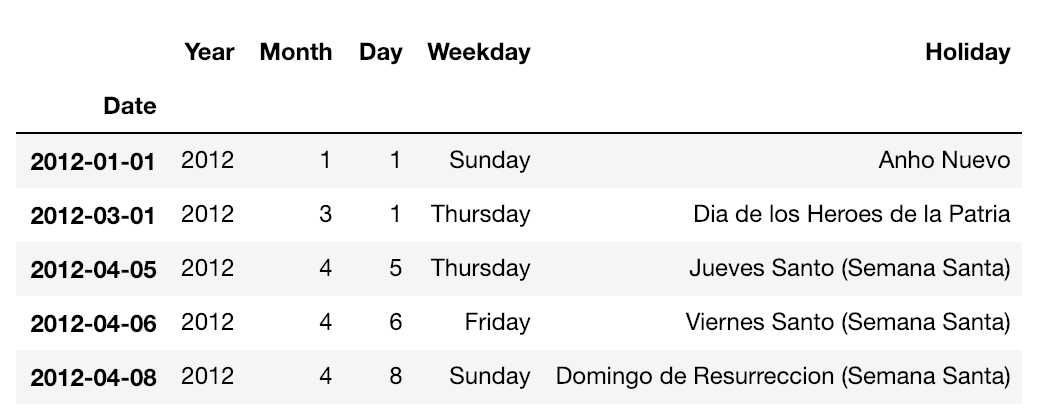
An **exogenous variable** is one whose value is determined outside the model and is imposed on the model. In other words, variables that affect a model without being affected by it. Read more about exogenous variables [here](http://www.businessdictionary.com/definition/exogenous-variable.html).

Many models can be used to solve a task like this, but **SARIMAX** is the one we’ll be working with. SARIMAX stands for [Seasonal AutoRegressive Integrated Moving Average with eXogenous](https://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html) regressors.

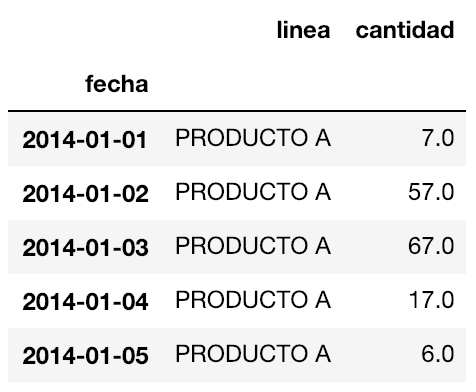
All good! Now we will go through the steps one can follow to build a sales forecaster.

As [I have explained before](https://medium.com/@oscarzamendia/an-introduction-to-time-series-analysis-using-python-and-pandas-222fe72b191a?source=friends_link&sk=ad5965b01d205cacee81aef407d5cda3), dealing with time series poses some challenges such as making it stationary. If you want more details on why I perform some transformations on the dataframe go and check out my previous post. The focus of this article is the forecasting method.

In this opportunity, we have two files: one with data about past sales, and the other containing information about local public holidays. As you can imagine, the task will be to predict the amount of sales by combining these two datasets. After loading the files, the dataframes end up looking like this:



feriados\_df — Holidays dataframe



ventas\_df — past sales dataframe

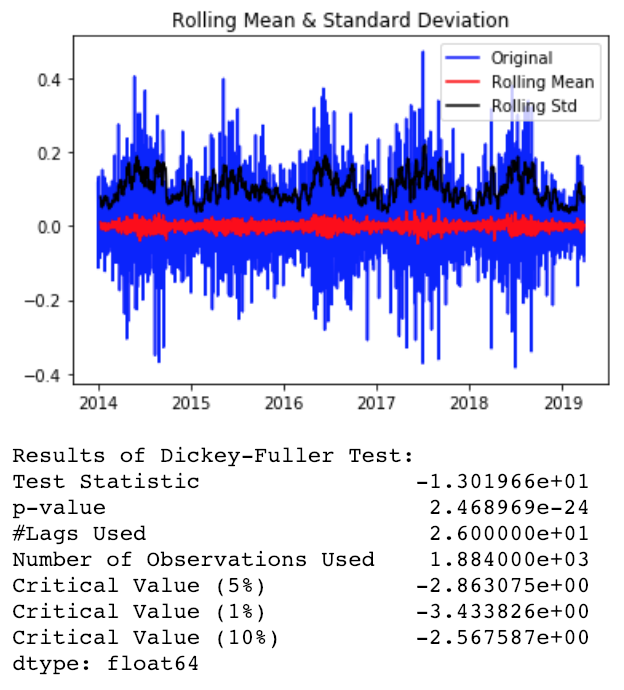
The granularity of both of our datasets is at day level, that is, both columns ‘Date’ and ‘fecha’ are indices with a daily frequency. If we want to set the frequency of a dataset we can run the following line:

ventas\_df = ventas\_df.resample(‘D’).mean() # 'D' for daily frequency

We will need to join these two datasets in order to fit our model with all the data we have. ‘ventas\_df’ has the variable we want to predict. ‘feriados\_df’ contains our exogenous variables.

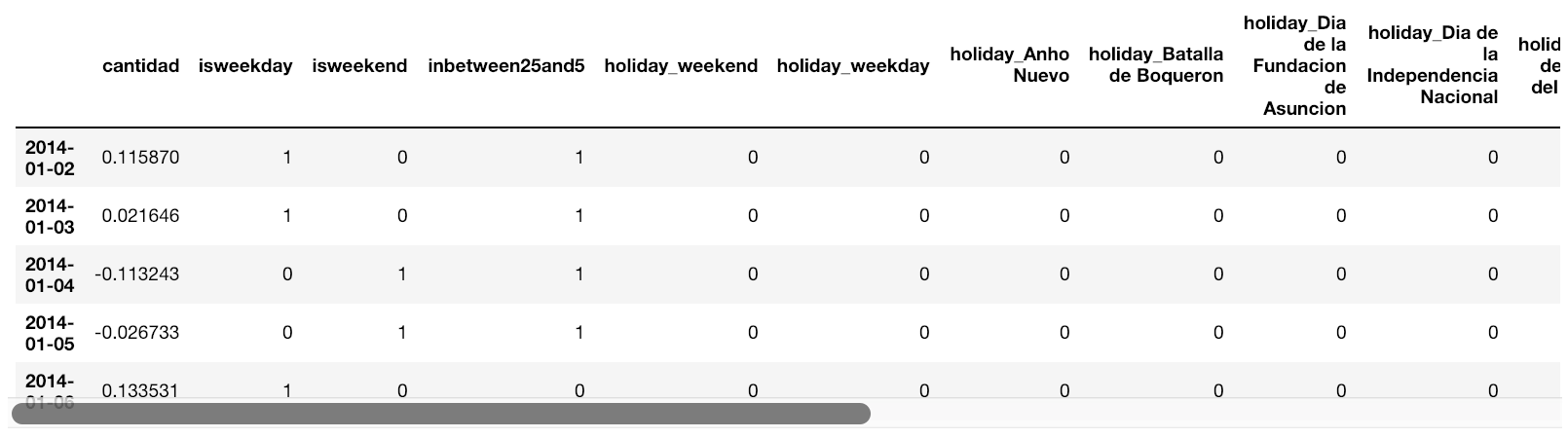
To make our lives easier, it is better to stationarise ventas\_df before joining it with feriados\_df. The method I used to make the series more stationary consisted in applying a log transformation and [differencing](https://otexts.com/fpp2/stationarity.html). The stationarised series was stored in ‘ts\_log\_diff’ dataframe.

test\_stationarity(ts\_log\_diff)



Now we can join feriados\_df and ts\_log\_diff, which is our transformed ventas\_df.

data\_df = ts\_log\_diff.join(feriados\_df, how='left')  
data\_df.head()



data\_df — joined dataframe

Sometimes after performing some operations with Pandas, our resulting dataframe loses its frequency. To fix that we can do:

data\_df = data\_df.asfreq('D')

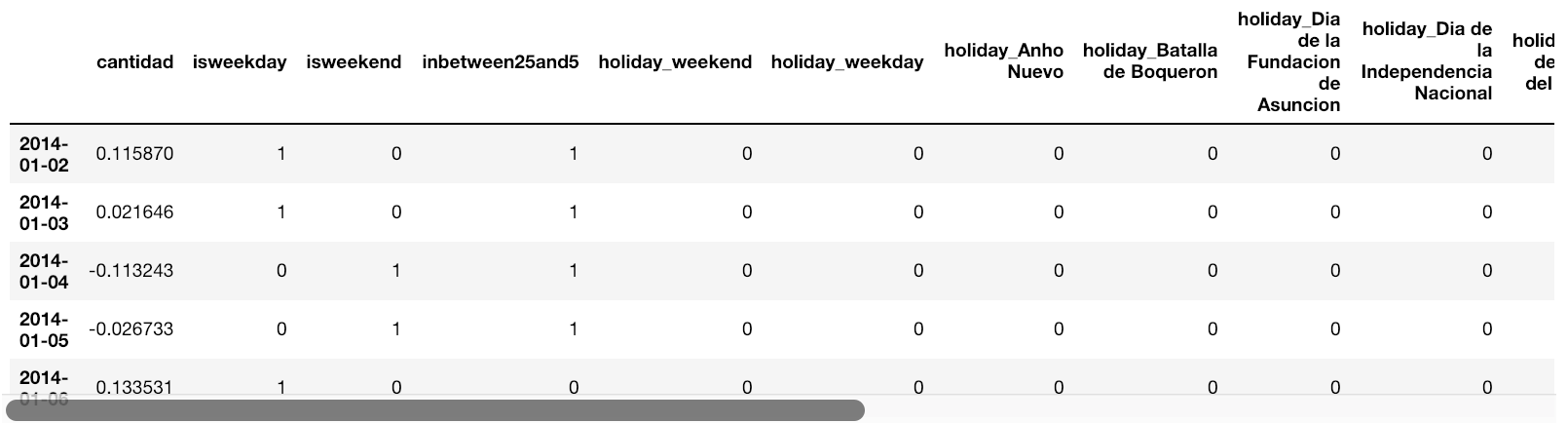
**It’s time for some feature engineering!**

One can come up with multiple ideas for creating new features out of the existing ones. For simplicity, let us compute the columns below.  
- holiday\_weekday: whether the public holiday fell on a weekday  
- holiday\_weekend: whether the public holiday fell on a Saturday or Sunday  
- isweekday: whether the date is a weekday  
- isweekend: if it is weekend  
- inbetween25and5: salaries are often paid during these days

**data\_df['isweekday']** = [1 if d >= 0 and d <= 4 else 0 for d in data\_df.index.dayofweek]  
**data\_df['isweekend']** = [0 if d >= 0 and d <= 4 else 1 for d in data\_df.index.dayofweek]  
**data\_df['inbetween25and5']** = [1 if d >= 25 or d <= 5 else 0 for d in data\_df.index.day]  
**data\_df['holiday\_weekend']** = [1 if (we == 1 and h not in [np.nan]) else 0 for we,h in data\_df[['isweekend','Holiday']].values]  
**data\_df['holiday\_weekday']** = [1 if (wd == 1 and h not in [np.nan]) else 0 for wd,h in data\_df[['isweekday','Holiday']].values]

Let’s apply one-hot-encoding on column ‘Holiday’ as well.

data\_df = pd.get\_dummies(data\_df, columns=['Holiday'], prefix=['holiday'], dummy\_na=True)



feature-engineered data\_df

**Can we predict already please!?**

Yass! Sort of… First, we have to split our data into train and test data. You know, for good practices and avoiding overfitting stuff ;)

We can’t just use k-folding methods to split our dataset up into training and test. This is because for TS we must take into account the time factor. There are some techniques that we can apply, among them:

1. **Train-Test split** that respect temporal order of observations.
2. **Multiple Train-Test splits** that respect temporal order of observations.
3. **Walk-Forward Validation** where a model may be updated each time step new data is received.

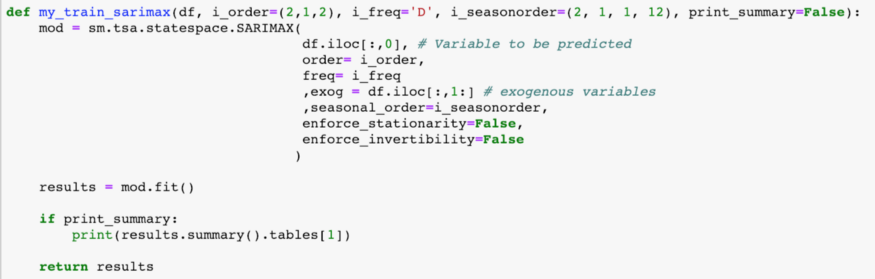
In this case, number one will be used. Data points from the beginning of the series up to February 2019 will serve as training data. The rest of the data points will be used for testing.

**Producing and Visualizing Forecasts**

result\_daily = my\_train\_sarimax(data\_df[:'2019-02-28'], i\_order=(2,1,2), i\_freq='D', i\_seasonorder=(2, 1, 1, 12))

In the line above, the training data points ‘data\_df[:’2019–02–28’] are passed to the function. It is worth noticing that the first column in the dataframe must contain the values to predict. The rest of the columns are our exogenous variables (i.e., holidays and engineered features). The frequency of the dataframe is given in the ‘i\_freq’ argument. Arguments ‘i\_order’ and ‘i\_seasonorder’ specify the parameters required to train the model, check documentation for SARIMAX to know more about these parameters.

Definition of my\_train\_sarimax() function as follow.

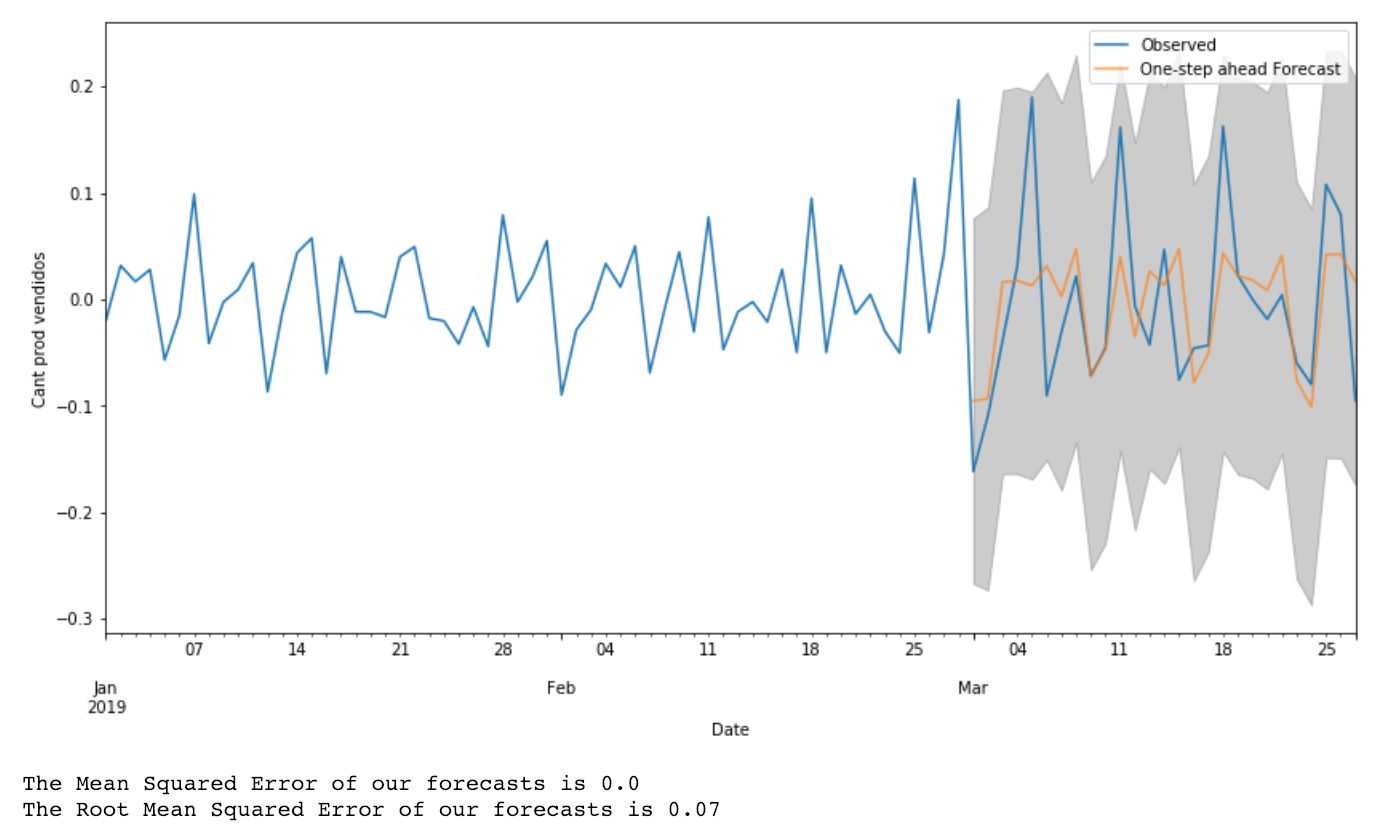


Now it’s time to validate our predictions. To do that we are going to use a function to retrieve the predicted values and then compare them against the true values in our test data points.

ypred, ytruth = compare\_pred\_vs\_real(result\_daily, data\_df, ‘2019–03–01’, exog\_validation=data\_df[‘2019–03–01’:].iloc[:,1:])

It is worth mentioning that the exogenous variables for the time frame to be predicted must be provided to the model. Remember that these are variables external to the model and it needs them to make the predictions.

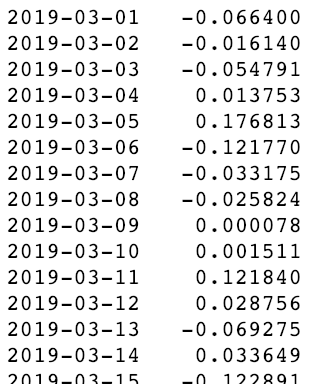
If we look at the ‘compare\_pred\_vs\_real()’ definition, we can see that the predictions are made with the ‘get\_prediction()’ function. The values can be extracted by using the ‘predicted\_mean’ method.



Performance of our model

We can say our model has a pretty decent performance in terms of MSE and RMSE. Let’s see how far appart are our predictions from the actual *number of items sold*.

ypred - ytruth



predictions minus true values

But, wait… Why do we see decimal values? The number of items to be sold should always be integers!

**(De-)transforming predictions**

Remember we log-transformed and then applied differencing to our dataset. In order to see the actual numbers our model estimates will be sold we must revert those transformations.

Because of the differencing operation the original first date in the TS was lost. We need to fill that missing value from ‘data\_df’. Next, we need to append to ‘y\_pred’ all the dates *before the prediction*. These dates also come from ‘data\_df’. Once we are done with all that, we can revert the differencing with cumsum() and then apply exp() to revert the log transformation.

#create a series with the dates that were dropped with differencing  
restore\_first\_values = pd.Series([6.008813], index=[pd.to\_datetime(‘2014–01–01’)])#get the values that the prediction does not have  
missing\_part = data\_df[‘cantidad’][:’2019–02–28']  
rebuilt = restore\_first\_values.append(missing\_part).append(ypred)#revert differencing:  
rebuilt = rebuilt.cumsum()#revert log transformation:  
rebuilt = np.exp(rebuilt).round() # apply round() to have integers

We can finally see our predicted values and compare them with the actual ones. Yay!

# Check how far were the predictions from the actual values  
rebuilt['2019-03-01':] - ventas\_df['cantidad']['2019-03-01':]

https://miro.medium.com/max/46/1*NQ-_qFAH5l5-NP6X2LbJFw.png?q=20

Looks like we got it right most of the time :)

**Final comments**

Please keep in mind that many methods can be used to accomplish stationarity in a TS. Also, SARIMAX is not the only model that exists to make predictions on time series, and further parameter tuning can help improve the accuracy of the model.