Anexo I. Script propio con funciones útiles

July 4, 2019

In [ ]:
```
import datetime
from dateutil.relativedelta import relativedelta

import pandas as pd
import numpy as np
from statsmodels.graphics import tsaplots
from statsmodels.api import ProbPlot
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns

from statsmodels.tsa.stattools import adfuller, kpss
from scipy.stats import gaussian_kde, norm


def tsplot (serie, alpha=1):
    plt.plot(serie, alpha=alpha)
    plt.xlabel("Tiempo en meses")
    plt.xlim([serie.index[0], serie.index[-1]])

def ts_split (serie, train_size=0.8):
    train_len  = int(len(serie) * train_size)
    split_date_train = serie.index[0] + relativedelta(months = train_len)
    split_date_test = serie.index[0] + relativedelta(months = train_len+1)
    train = serie[:split_date_train]
    test  = serie[split_date_test:]
    return train, test

def train_test_plot (train, test, leg_position=0):
    plt.plot(train)
    plt.plot(test, alpha=0.6)
    plt.axvspan(test.index[0], test.index[-1], facecolor='grey', alpha=0.1)
    plt.xlabel("Tiempo (meses)")
    plt.xlim([train.index[0], test.index[-1]])
    plt.legend(["Período de modelización", "Período de predicción"],
               loc=leg_position)
```

```
def forecast_plot (observed, predicted, leg_position=0):
    plt.plot(observed, alpha=0.6)
    plt.plot(predicted, "-.")
    plt.xlabel("Tiempo (meses)")
    plt.xlim([observed.index[0], predicted.index[-1]])
    plt.legend(["Observado", "Predicho"], loc=leg_position)

def naive (train, test):
    naive_list = []
    last = train.value[-1]

    for i in range(len(test)):
        naive_list.append(last)

    naive_forecast = pd.Series(naive_list)
    naive_forecast.index = test.index
    return naive_forecast

def snaive (train, test):
    snaive_list = []
    last_seas = list(train.value[-12:])

    for i in range(len(test)):
        last_seas.append(last_seas[-12])
        snaive_list.append(last_seas[-12])

    snaive_forecast = pd.Series(snaive_list)
    snaive_forecast.index = test.index
    return snaive_forecast

def stationarity_test (serie):
    adf_test  = adfuller(np.reshape(serie.values, len(serie)))
    kpss_test = kpss(np.reshape(serie.values, len(serie)))
    return adf_test, kpss_test

def resid_diag (serie):
    fig, ax = plt.subplots(2, 2, figsize=(12, 8))

    ax[0,0].plot(serie)
    ax[0,0].set_xlim(serie.index[0], serie.index[-1])
    ax[0,0].set_title('Residuales estandarizados')

    ax[0,1].hist(serie, normed=True, alpha=0.5)
    kde = gaussian_kde(serie)
    xlim = (-1.96*2, 1.96*2)
    x = np.linspace(xlim[0], xlim[1])
    ax[0,1].plot(x, kde(x), label='KDE')
    ax[0,1].plot(x, norm.pdf(x), label='N(0,1)')
```

```
    ax[0,1].set_title('Densidad teórica N(0, 1) y estimada')

    pp = ProbPlot(serie, dist=norm, fit=True)
    qq = pp.qqplot(ax=ax[1,0], markerfacecolor='cornflowerblue',
                   line="45")
    ax[1,0].set_title('Gráfico QQ')

    tsaplots.plot_acf(serie, ax=ax[1, 1], lags=24)
    ax[1,1].set_title('ACF')

    plt.subplots_adjust(hspace=0.3, wspace=0.2)

    return fig, ax

def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
    """
    Frame a time series as a supervised learning dataset.
    Arguments:
    data: Sequence of observations as a list or NumPy array.
    n_in: Number of lag observations as input (X).
    n_out: Number of observations as output (y).
    dropnan: Boolean whether or not to drop rows with NaN values.
    Returns:
    Pandas DataFrame of series framed for supervised learning.
    """
    n_vars = 1 if type(data) is list else data.shape[1]
    df = pd.DataFrame(data)
    cols, names = list(), list()
    # input sequence (t-n, ... t-1)
    for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
    # put it all together
    agg = pd.concat(cols, axis=1)
    agg.columns = names
    # drop rows with NaN values
    if dropnan:
        agg.dropna(inplace=True)
    return agg

def iter_forecast(model, input_x, n_input, test):
```

```
    yhat_sequence = []
    input_data = [x for x in input_x]
    for j in range(len(test)):
        # prepare the input data
        X = np.array(input_data[-n_input:]).reshape(1, n_input)
        # make a one-step forecast
        yhat = model.predict(X)[0]
        # add to the result
        yhat_sequence.append(yhat)
        # add the prediction to the input
        input_data.append(yhat)
    yhat_ts = pd.Series(yhat_sequence)
    yhat_ts.index = test.index
    return yhat_ts

def tscv_plot (train, test, end):

    train_start = train.index[0]
    test_start  = test.index[0]
    test_end    = test.index[-1]

    plt.plot(train)
    plt.plot(test)
    plt.axvspan(test_start, test_end, facecolor='grey', alpha=0.1)
    plt.xlabel("Tiempo (meses)")
    plt.xlim([train_start, end])
    plt.legend(["Train", "Validation"], loc="upper center",
               bbox_to_anchor=(1.08, 0.6), ncol=1, frameon = False)

def smape (observed, predicted):
    residual = predicted - observed
    pe_sum = np.abs(residual) / (np.abs(observed) + np.abs(predicted))
    return (200/len(observed))*np.sum(pe_sum)
```