Sarit Maitra

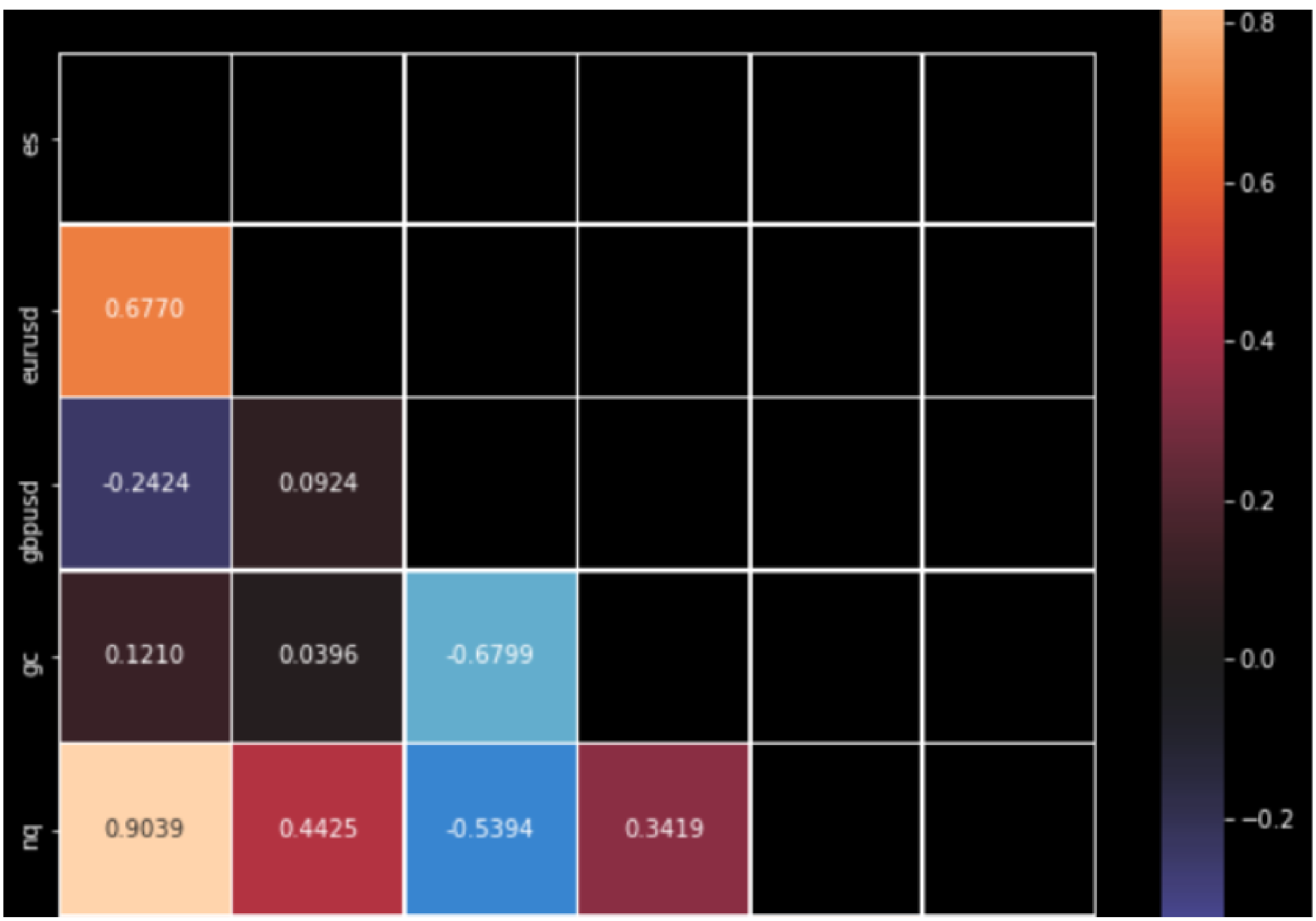
1.4K Followers About

SHORT TERM AND LONG TERM DYNAMICS MODEL

Vector Error Correction Model Configuration & Analysis

Relational econometric model for time-series data

Sarit Maitra Dec 15, 2020 · 13 min read ★



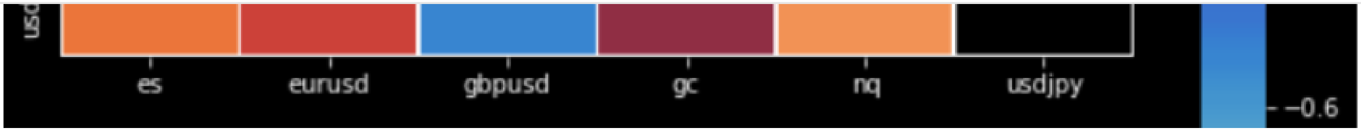


Image by author

Error correction model (ECM)is important in time-series analysis to better understand long-run dynamics. ECM can be derived from auto-regressive distributed lag model as long as there is a cointegration relationship between variables. In that context, each equation in the vector auto regressive (VAR) model is an autoregressive distributed lag model; therefore, it can be considered that the vector error correction model (VECM) is a VAR model with cointegration constraints.

Cointegration relations built into the specification so that it restricts the long-run behavior of the endogenous variables to converge to their cointegrating relationships while allowing for short-run adjustment dynamics. This is known as the error correction term since the deviation from long-run equilibrium is corrected gradually through a series of partial short-run adjustments.

The above theory would be clear once we run through an example and a use case. Let us collect some data sample.

```
df = pd.read_csv("April_data_6series.csv")
df.sample(5)
```

	Unnamed: 0	timestamp	es	eurusd	gbpusd	gc	nq	usdjpy
16162	16162	2020-05-18 14:19:00	2932.625	1085275.0	1217650.0	1745.00	9287.750	107496500.0
14594	14594	2020-05-15 09:12:00	2857.125	1081545.0	1220935.0	1744.40	9130.625	107073000.0
2457	2457	2020-05-04 19:56:00	2832.125	1089745.0	1244615.0	1711.60	8815.750	106705000.0
29228	29228	2020-05-29 19:04:00	3019.250	1110050.0	1233080.0	1731.05	9522.875	107787000.0
217	217	2020-05-01 03:37:00	2863.625	1094560.0	1256330.0	1694.30	8818.125	107160500.0

```
X = df[:15000] # subset of data
X
```

	es	eur_usd	gbp_usd	gc	nq	usd_jpy

2020-05-01 00:01:00	2875.38	1094215.00	1257935.00	1695.60	8841.25	107368000.00
2020-05-01 00:02:00	2874.62	1094165.00	1257765.00	1695.65	8837.75	107376500.00
2020-05-01 00:03:00	2874.12	1094115.00	1257565.00	1695.40	8836.75	107381500.00
2020-05-01 00:04:00	2875.25	1094190.00	1257535.00	1694.85	8841.12	107376500.00

Visualization:

```
plt.style.use('dark_background')
def plot_vars(train, levels, color, leveltype):
    """Displays historical trends of variables and see if it's
sensible to just select levels instead of differences"""
    fig, ax = plt.subplots(1, 6, figsize=(16,2.5), sharex=True)
    for col, i in dict(zip(levels, list(range(6)))).items():
        X[col].plot(ax=ax[i], legend=True, linewidth=1.0,
color=color, sharex=True)
    fig.suptitle(f"Historical trends of {leveltype} variables",
                 fontsize=12, fontweight="bold")
plot_vars(X.values, levels = X.columns, color="red", leveltype =
"levels")
plt.tight_layout()
```

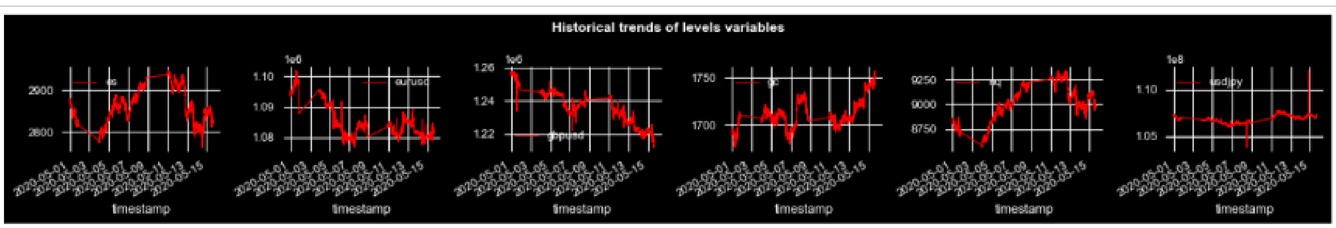


Image by author

Stationarity check:

The outcome of unit root testing matters for the empirical model to be estimated. Stationarity means here that the mean, variance and intertemporal correlation structure remains constant over time. Non-stationarities here from the stochastic properties of the process.

ADF Test:

```
    p_value = output['pvalue']
    def adjust(val, length= 6): return str(val).ljust(length)

    # Print Summary
    print(f'    Augmented Dickey-Fuller Test on "{name}"', "\n   ", '-'*47)
    print(f' Null Hypothesis: Data has unit root. Non-Stationary.')
    print(f' Significance Level    = {signif}')
    print(f' Test Statistic        = {output["test_statistic"]}')
    print(f' No. Lags Chosen       = {output["n_lags"]}')

    for key,val in r[4].items():
        print(f' Critical value {adjust(key)} = {round(val, 3)}')

    if p_value <= signif:
        print(f" => P-Value = {p_value}. Rejecting Null Hypothesis.")
        print(f" => Series is Stationary.")
    else:
        print(f" => P-Value = {p_value}. Weak evidence to reject the Null Hypothesis.")
        print(f" => Series is Non-Stationary.")
```

With a p-value of > 0.05 , we have no reason to reject the null hypothesis, meaning that we can conclude that the series is not stationary.

```
def kpss_test(x, h0_type='c'):
    indices = ['Test Statistic', 'p-value', '# of Lags']
    kpss_test = kpss(x, regression=h0_type, nlags ='auto')
    results = pd.Series(kpss_test[0:3], index=indices)
    for key, value in kpss_test[3].items():
        results[f'Critical Value ({key})'] = value
        return results

print('KPSS-EURUSD:')
print(kpss_test(X.eurusd))
print('_______________________')
print('KPSS-GBPUSD:')
print(kpss_test(X.gbpusd))
print('_______________________')
print('KPSS-USDJPY:')
print(kpss_test(X.usdjpy))
print('_______________________')
print('KPSS-GC:')
print(kpss_test(X.gc))
print('_______________________')
print('KPSS-NQ:')
print(kpss_test(X.nq))
print('_______________________')
print('KPSS-ES:')
print(kpss_test(X.es))
```

favor of the alternative one, meaning that the series is not stationary.

To extract maximum information from our data, it is important to have a normal or Gaussian distribution of the data. To check for that, we have done a normality test based on the Null and Alternate Hypothesis intuition.

Normality test:

```
stat,p = stats.normaltest(X.eurusd)
print('Statistics=%.3f, p=%.3f' % (stat,p))
alpha = 0.05
if p > alpha:
    print('EURUSD Data looks Gaussian (fail to reject H0)')
else:
    print('EURUSD Data do not look Gaussian (reject H0)')
print('______________')
stat,p = stats.normaltest(X.gbpusd)
print('Statistics=%.3f, p=%.3f' % (stat,p))
alpha = 0.05
if p > alpha:
    print('GBPUSD Data looks Gaussian (fail to reject H0)')
else:
    print('GBPUSD Data do not look Gaussian (reject H0)')
print('______________')
stat,p = stats.normaltest(X.usdjpy)
print('Statistics=%.3f, p=%.3f' % (stat,p))
alpha = 0.05
if p > alpha:
    print('USDJPY Data looks Gaussian (fail to reject H0)')
else:
    print('USDJPY Data do not look Gaussian (reject H0)')
print('______________')
stat,p = stats.normaltest(X.es)
print('Statistics=%.3f, p=%.3f' % (stat,p))
alpha = 0.05
if p > alpha:
    print('ES Data looks Gaussian (fail to reject H0)')
else:
    print('ES Data do not look Gaussian (reject H0)')
print('______________')
stat,p = stats.normaltest(X.nq)
print('Statistics=%.3f, p=%.3f' % (stat,p))
alpha = 0.05
if p > alpha:
    print('NQ Data looks Gaussian (fail to reject H0)')
else:
```

```
print('Statistics=%.3f, p=%.3f' % (stat,p))
alpha = 0.05
if p > alpha:
    print('GC Data looks Gaussian (fail to reject H0)')
else:
    print('GC Data do not look Gaussian (reject H0)')
print('______________')

print('EURUSD: Kurtosis of normal distribution: {}'.
format(stats.kurtosis(X.eurusd)))
print('EURUSD: Skewness of normal distribution: {}'.
format(stats.skew(X.eurusd)))
print('************')
print('GBPUSD: Kurtosis of normal distribution: {}'.
format(stats.kurtosis(X.gbpusd)))
print('GBPUSD: Skewness of normal distribution: {}'.
format(stats.skew(X.gbpusd)))
print('************')
print('USDJPY: Kurtosis of normal distribution: {}'.
format(stats.kurtosis(X.usdjpy)))
print('USDJPY: Skewness of normal distribution: {}'.
format(stats.skew(X.usdjpy)))
print('************')
print('ES: Kurtosis of normal distribution: {}'.
format(stats.kurtosis(X.es)))
print('ES: Skewness of normal distribution: {}'.
format(stats.skew(df.es)))
print('************')
print('NQ: Kurtosis of normal distribution: {}'.
format(stats.kurtosis(X.nq)))
print('NQ: Skewness of normal distribution: {}'.
format(stats.skew(X.nq)))
print('************')
print('GC: Kurtosis of normal distribution: {}'.
format(stats.kurtosis(X.gc)))
print('GC: Skewness of normal distribution: {}'.
format(stats.skew(X.gc)))
```

```
Statistics=1730.135, p=0.000
EURUSD Data do not look Gaussian (reject H0)
______________
Statistics=1213.219, p=0.000
GBPUSD Data do not look Gaussian (reject H0)
______________
Statistics=1714.579, p=0.000
USDJPY Data do not look Gaussian (reject H0)
______________
```

```
Statistics=2540.823, p=0.000
NQ Data do not look Gaussian (reject H0)
_______________
Statistics=822.580, p=0.000
GC Data do not look Gaussian (reject H0)
_______________
EURUSD: Kurtosis of normal distribution: -0.24795142380816637
EURUSD: Skewness of normal distribution: 0.6268226365879681
************
GBPUSD: Kurtosis of normal distribution: -0.3789097254009337
GBPUSD: Skewness of normal distribution: 0.4601503086221758
************
USDJPY: Kurtosis of normal distribution: 1.9488984116380061
USDJPY: Skewness of normal distribution: -0.28981427530311443
************
ES: Kurtosis of normal distribution: -0.990571444317438
ES: Skewness of normal distribution: 0.10577134642346937
************
NQ: Kurtosis of normal distribution: -0.6626025675168852
NQ: Skewness of normal distribution: -0.5375467712666768
************
GC: Kurtosis of normal distribution: -0.31649760066630384
GC: Skewness of normal distribution: 0.3761381862370276
```

These distribution gives us some intuition about the normal distribution of our data. Value close to 0 for Kurtosis indicates a Normal Distribution where asymmetrical nature is signified by a value between -0.5 and +0.5 for skewness. The tails are heavier for kurtosis greater than 0 and vice versa. Moderate skewness refers to the value between -1 and -0.5 or 0.5 and 1.

Correlation & Causation:

Though correlation helps us determine the degree of relationship between the variables, it does not tell us about the cause & effect of the relationship. A high degree of correlation does not always necessarily mean a relationship of cause & effect exists between variables. Here, in this context, it can be noted that, correlation does not imply causation, although the existence of causation always implies correlation.

```
# Generate a mask for the upper triangle
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(10, 10))
# Heatmap with the mask and correct aspect ratio
sns.heatmap(corr, annot=True, fmt = '.4f', mask=mask, center=0,
square=True, linewidths=.5)
print("value > 0.5 is considerred correlated, > 0.8 is highly
correlated")
plt.show()

print('Correlation matrix:')
corr = X.corr()
corr.style.background_gradient(cmap='coolwarm').set_precision(2)
```

```
value > 0.5 is considerred correlated, > 0.8 is highly correlated
```

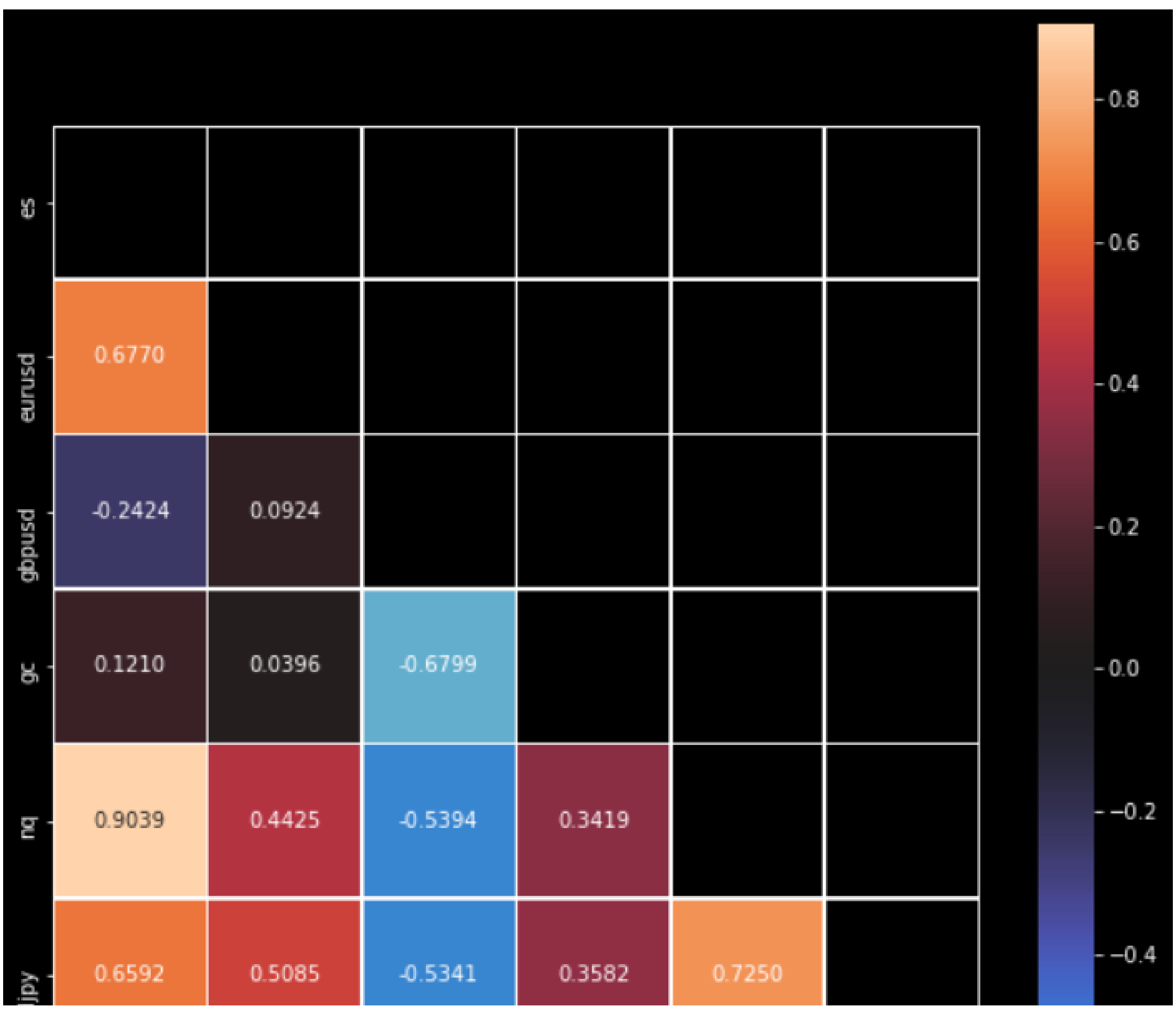


Image by author

	es	eurusd	gbpusd	gc	nq	usdjpy
es	1.00	0.68	-0.24	0.12	0.90	0.66
eurusd	0.68	1.00	0.09	0.04	0.44	0.51
gbpusd	-0.24	0.09	1.00	-0.68	-0.54	-0.53
gc	0.12	0.04	-0.68	1.00	0.34	0.36
nq	0.90	0.44	-0.54	0.34	1.00	0.72
usdjpy	0.66	0.51	-0.53	0.36	0.72	1.00

Granger Casuality test

The basis behind Vector Auto-Regression is that each of the time series in the system influences each other. This way, we can predict the series with past values of itself along with other series in the system. We will use Granger's Causality Test to test this relationship before building the model.

- Null hypothesis (H0) = coefficients of past values in the regression equation is zero.

Below, we are checking Granger Causality of all possible combinations of the series. The rows are the response variable, columns are predictors. The values in the table are the P-Values.

```
max_lag = 6
test = 'ssr_chi2test'
def causation_matrix(data, variables, test='ssr_chi2test',
verbose=False):
  X = DataFrame(np.zeros((len(variables), len(variables))),
columns=variables, index=variables)
  for c in X.columns:
    for r in X.index:
    test_result = grangercausalitytests(data[[r, c]], maxlag =
max_lag, verbose = False)
    p_values = [round(test_result[i+1][0][test][1],4) for i in
range(max_lag)]
```

Open in app

```
X.columns = [var + '-x axis' for var in variables]
X.index = [var + '-y axis' for var in variables]
return X
causation_matrix(X, variables = X.columns)
```

	es-x axis	eurusd-x axis	gbpusd-x axis	gc-x axis	nq-x axis	usdjpy-x axis
es-y axis	1.00	0.00	0.05	0.70	0.01	0.00
eurusd-y axis	0.00	1.00	0.00	0.00	0.00	0.02
gbpusd-y axis	0.00	0.00	1.00	0.00	0.00	0.03
gc-y axis	0.03	0.03	0.00	1.00	0.06	0.00
nq-y axis	0.00	0.00	0.03	0.69	1.00	0.00
usdjpy-y axis	0.06	0.17	0.00	0.10	0.00	1.00

P value is less than the significant level of 5%, which indicates the need to accept the null hypothesis, namely the existence of Granger cause.

We have seen earlier hat all the series are unit root non-stationary, they may be co-integrated. This extension of unit root concept to multiple time series means that a liner combination of two or more series is stationary and hence, mean reverting. VAR model is not equipped to handle this case without differencing. So, we will use here Vector Error Correction Model (VECM). We will explore here cointegration because it can be leveraged for trading strategy.

However, the concept of an integrated multivariate series is complicated by the fact that, all the component series of the process may be individually integrated but the process is not jointly integrated in the sense that one or more linear combinations of the series exist that produce a new stationary series. To simplify, a combination of two co-integrated series has a stable mean to which this linear combination reverts. A multivariate series with this characteristics is said to be cointegrated.

Test for co-integration:

two or more series is stationary and hence, mean reverting. The first hypothesis, tests for the presence of cointegration.

When we are dealing with models based on nonstationary variables, we normally difference I(1) data and using OLS we create a dynamic mode. But, in this process long-term relationship is lost from the data. Dependencies between non-stationary variables which are sometimes stable in time are called co-integration relationships. There is a mechanism that brings the system back to equilibrium every time it is shocked away from it (Granger theorem).

Here, we are testing the order of integration using Johansen's procedurce. Let us determine the lag value by fitting a VECM model and passing a maximum lag as 8.

```
nobs = 15
train_ecm, test_ecm = X[0:-nobs], X[-nobs:]

# Check size
print(train_ecm.shape)
print(test_ecm.shape)
```

```
(14985, 6)
(15, 6)
```

Order selection:

Rule-of-thumb formula for maximum lag length:

$$\left(4 * \frac{T}{100}\right)^{1/4}$$

where T sample size (Schwert, 2002)

```
# VECM model fitting
from statsmodels.tsa.vector_ar import vecm
# pass "1min" frequency
train_ecm.index = pd.DatetimeIndex(train_ecm.index).to_period('1min')
model = vecm.select_order(train_ecm, maxlags=8)
```

```
==================================================
        AIC         BIC         FPE         HQIC
--------------------------------------------------
0       45.14       45.16   4.006e+19       45.14
1       44.94       44.98   3.298e+19       44.96
2       44.87      44.93*   3.071e+19       44.89
3       44.86       44.94   3.041e+19       44.89
4       44.85       44.95   3.019e+19      44.89*
5       44.85       44.97   3.018e+19       44.89
6       44.85       44.98   3.015e+19       44.90
7       44.85       45.00   3.013e+19       44.90
8      44.85*       45.02  3.012e+19*       44.91
--------------------------------------------------
```

Above AIC and BIC are both penalized-likelihood criteria. BIC penalizes model complexity more heavily. At large number of instances, AIC tends to pick somewhat larger models than BIC. Comparatively, BIC penalizes the number of parameters in the model to a greater extent than AIC. We will consider BIC (3: 44,93) for our use case.

Johansen co-integration on level data:

Johansen test assesses the validity of a cointegrating relationship, using a maximum likelihood estimates (MLE) approach.

Two types of Johansen's test:

- one uses trace (from linear algebra),
- the other a maximum eigenvalue approach (an eigenvalue is a special scalar; when we multiply a matrix by a vector and get the same vector as an answer, along with a new scalar, the scalar is called an eigenvalue).
- Both forms of the test will determine if cointegration is present. The hypothesis is stated as:

Johansen Cointegration Test releases two statistics — Trace Statistic (from linear algebra) and Max-Eigen Statistic (an eigenvalue is a special scalar; when we multiply a matrix by a vector and get the same vector as an answer, along with a new scalar, the scalar is called an eigenvalue).

The difference is in the alternate hypothesis (H1): the trace test alternate hypothesis is simply that the number of cointegrating relationships is at least one (shown by the number of linear combinations).

- Rejecting the H0 is basically stating there is only one combination of the non-stationary variables that gives a stationary process.

```
pd.options.display.float_format = "{:.2f}".format
"""definition of det_orderint:
-1 - no deterministic terms; 0 - constant term; 1 - linear trend"""
pd.options.display.float_format = "{:.2f}".format
model = coint_johansen(endog = train_ecm, det_order = 1, k_ar_diff = 3)
print('Eigen statistic:')
print(model.eig)
print()
print('Critical values:')
d = DataFrame(model.cvt)
d.rename(columns = {0:'90%', 1: '95%', 2:'99%'}, inplace=True)
print(d); print()
print('Trace statistic:')
print(DataFrame(model.lr1))
```

```
Eigen statistic:
[1.04952026e-02 2.94113957e-03 1.52293361e-03 8.61024360e-04
 6.18626590e-04 8.31512622e-05]

Critical values:
     90%    95%    99%
0 102.47 107.34 116.98
1  75.10  79.34  87.77
2  51.65  55.25  62.52
3  32.06  35.01  41.08
4  16.16  18.40  23.15
5   2.71   3.84   6.63

Trace statistic:
       0
0 248.44
1  90.38
2  46.25
3  23.42
4  10.52
5   1.25
```

1. trace statistic (46,23) < critical value (55,25),
2. trace statistics (23,42) < critical value (35, 01),
3. trace statistics (10,52) < critical value (18,40),
4. trace statistics (1,25) < critical value (3,84)at 95% confidence level
5. However, 1st (248,44) and 2nd (90,38) > all the critical values at 90%, 95% and 99% confidence level.

So, we have a strong evidence to reject the H0 of no cointegration and H1 of cointegration exists are accepted. This makes it a good candidate for error correction model.

We can safely assume that, the series in question are related and therefore can be combined in a linear fashion. If there are shocks in the short run, which may affect movement in the individual series, they would converge with time (in the long run). We can estimate both long-run and short-run models here. The series are moving together in such a way that their linear combination results in a stationary time series and sharing an underlying common stochastic trend.

So, we see here that, cointegration analysis demonstrates that, the series is question do have long-run equilibrium relationships, but, in the short term, the series are in disequilibrium. The short-term imbalance and dynamic structure can be expressed as VECM.

Error correction model (ECM):

ECM shows the long-run equilibrium relationships of variables by inducing a short-run dynamic adjustment mechanism that describes how variables adjust when they are out of equilibrium.

Let us identify the cointegration rank.

```
from statsmodels.tsa.vector_ar.vecm import select_coint_rank
rank1 = select_coint_rank(train_ecm, det_order = 1, k_ar_diff = 3,
                                   method = 'trace', signif=0.01)
```

```
r_0 r_1 test statistic critical value
-------------------------------------
  0   6          248.4          117.0
  1   6          90.38          87.77
  2   6          46.25          62.52
-------------------------------------
```

- 1st column in the table shows the rank which is the number of cointegrating relationships for the dataset, while the 2nd reports the number of equations in total.
- λ trace statistics in the 3rd column, together with the corresponding critical values.
- In 1st 2 rows, we see that test statistic > critical values, so the null of at most one cointegrating vector is rejected.
- However, test statistic (46,25) at 3rd row do not exceeds the critical value (62,52), so the null of at most three cointegrating vectors cannot be rejected.

Below test statistic on maximum eigen value:

Maximum-eigenvalue statistic assumes a given number of r cointegrating relations under the null hypothesis and tests this against the alternative that there are r + 1 cointegrating equations.

```
rank2 = select_coint_rank(train_ecm, det_order = 1, k_ar_diff = 3,
                          method = 'maxeig', signif=0.01)

print(rank2.summary())
```

```
Johansen cointegration test using maximum eigenvalue test statistic with 1% significance level
=====================================
r_0 r_1 test statistic critical value
-------------------------------------
  0   1          158.1          49.41
  1   2          44.13          42.86
  2   3          22.83          36.19
-------------------------------------
```

Here too, we see the 3rd row test statistic (23,83) do not exceeds the critical value (36,19).

Model fitting:

```
from statsmodels.tsa.vector_ar.vecm import VECM
# VECM
vecm = VECM(train_ecm, k_ar_diff=3, coint_rank = 3, deterministic='ci')
"""estimates the VECM on the prices with 3 lags, 3 cointegrating relationship, and
a constant within the cointegration relationship"""
vecm_fit = vecm.fit()
print(vecm_fit.summary())
```

Residual auto-correlation:

```
from statsmodels.stats.stattools import durbin_watson
out = durbin_watson(vecm_fit.resid)
for col, val in zip(train_ecm.columns, out):
    print((col), ':', round(val, 2))
```

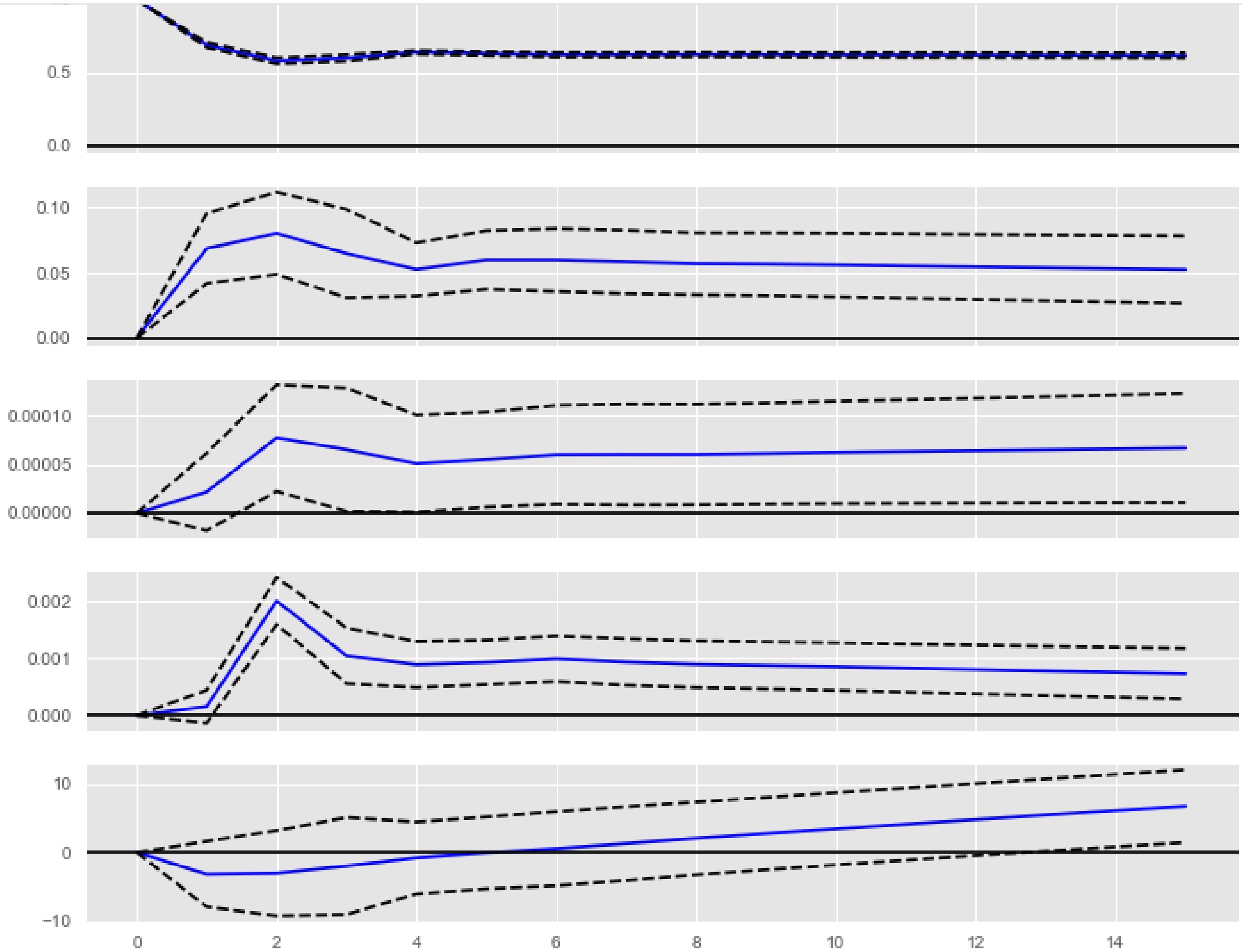
```
es : 2.0
eurusd : 2.01
gbpusd : 2.0
gc : 2.0
nq : 2.0
usdjpy : 2.0
```

Impulse-response function (IRF):

In order to analyze dynamic effects of the model responding to certain shocks as well as how the effects are among the 6 variables, further analysis can be made through IRF, and the results for 15 periods can be obtained. IRF is adopted to reflect shock effect of a system on an internal variable.

I have shown and explained one as below:

```
plt.style.use('ggplot')
irf.plot(impulse='eurusd')
plt.show()
```



As shown the 2nd plot from top, after analysis of the effects of *eurusd* price shock, it is found that positive shock have some impact. *eurusd* prices decline after a positive shock, reach the lowest point in the 2nd period, then rise slowly, reach the peak in the 4th period, and then remain at a stable level. This suggests that positive shock of *eurusd* prices has considerable influence on its own increasing, and the considerable influence has relatively long sustained effectiveness. Here our period is in minute frequency.

Plot 1 (topmost) is the IRF diagram of *es* (*E-miniS&PFutures)* changes caused by *eurusd* price shocks. As seen in the figure, the first positive shock in the first period causes *es* fluctuation and *es* reaches the peak at the 2nd period. Then *es* quickly declines to the lowest point in the third period, and after that returns to a stable condition. This shows that *eurusd* price shock can be shortly transferred to *es*, and has relatively large impacts on *es* in the short term, but *es* becomes stable around 3rd or 4th period. *eurusd* price

Likewise, we can plot and analyze all the variables against each one.

Prediction:

```
pd.options.display.float_format = "{:.2f}".format
forecast, lower, upper = vecm_fit.predict(nobs, 0.05)
print("lower bounds of confidence intervals:")
print(DataFrame(lower.round(2)))
print("\npoint forecasts:")
print(DataFrame(forecast.round(2)))
print("\nupper bounds of confidence intervals:")
print(DataFrame(upper.round(2)))
```

Below we get the prediction as in point forecast with lower bound and upper bound confidence intervals.

```
lower bounds of confidence intervals:
         0          1          2       3       4            5
0  2817.22 1081006.74 1211561.53 1754.12 8959.26 107147036.20
1  2816.13 1080905.78 1211440.30 1753.73 8955.87 107107152.97
2  2815.31 1080842.24 1211342.66 1753.46 8953.31 107077619.20
3  2814.59 1080781.27 1211252.25 1753.24 8951.13 107052471.30
4  2813.96 1080720.92 1211172.35 1753.05 8949.22 107030968.88
5  2813.38 1080666.47 1211100.55 1752.88 8947.49 107012107.55
6  2812.84 1080616.94 1211034.70 1752.72 8945.89 106995296.70
7  2812.34 1080570.33 1210973.82 1752.58 8944.41 106980125.01
8  2811.86 1080526.16 1210917.20 1752.45 8943.02 106966301.13
9  2811.41 1080484.32 1210864.18 1752.33 8941.70 106953616.84
10 2810.97 1080444.52 1210814.29 1752.22 8940.44 106941910.30
11 2810.56 1080406.47 1210767.15 1752.11 8939.24 106931053.24
12 2810.16 1080369.97 1210722.45 1752.01 8938.09 106920942.73
13 2809.77 1080334.89 1210679.94 1751.92 8936.98 106911494.52
14 2809.39 1080301.09 1210639.39 1751.83 8935.91 106902638.28
```

```
point forecasts:
         0          1          2       3       4            5
0  2819.58 1081467.48 1212277.02 1755.19 8967.01 107273990.95
1  2819.53 1081469.18 1212294.51 1755.22 8967.05 107275652.98
2  2819.50 1081471.41 1212308.00 1755.26 8967.12 107278381.53
```

```
6  2819.23 1081480.37 1212372.87 1755.42 8966.95 107287269.05
7  2819.16 1081482.84 1212389.28 1755.46 8966.91 107289334.35
8  2819.10 1081485.36 1212405.73 1755.50 8966.87 107291340.80
9  2819.03 1081487.90 1212422.04 1755.54 8966.84 107293292.49
10 2818.97 1081490.44 1212438.26 1755.58 8966.80 107295190.62
11 2818.91 1081492.97 1212454.40 1755.62 8966.76 107297035.67
12 2818.84 1081495.51 1212470.45 1755.66 8966.73 107298828.62
13 2818.78 1081498.05 1212486.43 1755.70 8966.69 107300570.56
14 2818.72 1081500.59 1212502.32 1755.74 8966.65 107302262.43
```

```
upper bounds of confidence intervals:
         0          1          2       3       4            5
0  2821.94 1081928.23 1212992.51 1756.25 8974.75 107400945.71
1  2822.93 1082032.57 1213148.73 1756.70 8978.22 107444153.00
2  2823.69 1082100.59 1213273.33 1757.05 8980.92 107479143.87
3  2824.26 1082163.17 1213389.48 1757.36 8983.02 107509045.50
4  2824.76 1082229.42 1213507.11 1757.63 8984.85 107534982.14
5  2825.21 1082289.19 1213612.54 1757.88 8986.49 107558188.56
6  2825.62 1082343.81 1213711.04 1758.12 8988.01 107579241.39
7  2825.99 1082395.34 1213804.74 1758.34 8989.41 107598543.69
8  2826.33 1082444.56 1213894.26 1758.55 8990.73 107616380.47
9  2826.66 1082491.48 1213979.90 1758.75 8991.97 107632968.15
10 2826.96 1082536.36 1214062.23 1758.94 8993.16 107648470.95
11 2827.25 1082579.48 1214141.65 1759.13 8994.28 107663018.11
12 2827.53 1082621.04 1214218.46 1759.31 8995.36 107676714.52
13 2827.79 1082661.21 1214292.92 1759.48 8996.40 107689646.61
14 2828.04 1082700.09 1214365.24 1759.65 8997.40 107701886.58
```

Below we are renaming the predicted columns as _pred

```
pd.options.display.float_format = "{:.2f}".format
forecast = DataFrame(forecast, index= test_ecm.index, columns= test_ecm.columns)
forecast.rename(columns = {'eurusd':'eurusd_pred', 'gbpusd':'gbpusd_pred', 'usdjpy':'usdjpy_pred',
                    'gc':'gc_pred', 'nq':'nq_pred', 'es':'es_pred'}, inplace = True)
forecast
```

	es_pred	eurusd_pred	gbpusd_pred	gc_pred	nq_pred	usdjpy_pred
timestamp						
2020-05-15 15:43:00	2819.58	1081467.48	1212277.02	1755.19	8967.01	107273990.95
2020-05-15 15:44:00	2819.53	1081469.18	1212294.51	1755.22	8967.05	107275652.98
2020-05-15 15:45:00	2819.50	1081471.41	1212308.00	1755.26	8967.12	107278381.53

Open in app

2020-05-15 15:49:00	2819.23	1081480.37	1212372.87	1755.42	8966.95	107287269.05
2020-05-15 15:50:00	2819.16	1081482.84	1212389.28	1755.46	8966.91	107289334.35
2020-05-15 15:51:00	2819.10	1081485.36	1212405.73	1755.50	8966.87	107291340.80
2020-05-15 15:52:00	2819.03	1081487.90	1212422.04	1755.54	8966.84	107293292.49
2020-05-15 15:53:00	2818.97	1081490.44	1212438.26	1755.58	8966.80	107295190.62
2020-05-15 15:54:00	2818.91	1081492.97	1212454.40	1755.62	8966.76	107297035.67
2020-05-15 15:55:00	2818.84	1081495.51	1212470.45	1755.66	8966.73	107298828.62
2020-05-15 15:56:00	2818.78	1081498.05	1212486.43	1755.70	8966.69	107300570.56
2020-05-15 15:57:00	2818.72	1081500.59	1212502.32	1755.74	8966.65	107302262.43

Accuracy metrics:

Below metrics for reporting purpose.

```
# score eur_usd
mae = mean_absolute_error(pred.eurusd, pred['eurusd_pred'])
mse = mean_squared_error(pred.eurusd, pred.eurusd_pred)
rmse = np.sqrt(mse)
sum = DataFrame(index = ['Mean Absolute Error', 'Mean squared error', 'Root mean squared error'])
sum['Accuracy metrics :    EURUSD'] = [mae, mse, rmse]

# score gbp_usd
mae = mean_absolute_error(pred.gbpusd, pred['gbpusd_pred'])
mse = mean_squared_error(pred.gbpusd, pred.gbpusd_pred)
rmse = np.sqrt(mse)
sum['GBPUSD'] = [mae, mse, rmse]

# score usd_jpy
mae = mean_absolute_error(pred.usdjpy, pred['usdjpy_pred'])
mse = mean_squared_error(pred.usdjpy, pred.usdjpy_pred)
rmse = np.sqrt(mse)
sum['USDJPY'] = [mae, mse, rmse]

# score nq
mae = mean_absolute_error(pred.nq, pred['nq_pred'])
mse = mean_squared_error(pred.nq, pred.nq_pred)
rmse = np.sqrt(mse)
sum['NQ'] = [mae, mse, rmse]

# score usd_jpy
mae = mean_absolute_error(pred.es, pred['es_pred'])
mse = mean_squared_error(pred.es, pred.es_pred)
rmse = np.sqrt(mse)
sum['ES'] = [mae, mse, rmse]

# score usd_jpy
mae = mean_absolute_error(pred.gc, pred['gc_pred'])
mse = mean_squared_error(pred.gc, pred.gc_pred)
rmse = np.sqrt(mse)
sum['GC'] = [mae, mse, rmse]
sum
```

Mean squared error	60524.42	161320.02	247956749.74	182.65	20.76	0.30
Root mean squared error	246.02	401.65	15746.64	13.51	4.56	0.55

Here, point to be noted that metrics and accuracy is useful but still we need to know what happens in practice. Because everything changes in real-life case scenario. To put the above metrics into business context, let us see where does predicted output stands against the actuals (separated test samples).

```
combine = concat([test_ecm, forecast], axis=1)
pred = combine[['eurusd', 'eurusd_pred', 'gbpusd', 'gbpusd_pred', 'usdjpy',
                'usdjpy_pred', 'gc', 'gc_pred', 'nq', 'nq_pred', 'es', 'es_pred']]
def highlight_cols(s):
    color = 'yellow'
    return 'background-color: %s' % color

pred.style.applymap(highlight_cols, subset=pd.IndexSlice[:, ['eurusd_pred', 'gbpusd_pred', 'usdjpy_pred',
                                                              'gc_pred', 'nq_pred', 'es_pred']])
```

timestamp	eurusd	eurusd_pred	gbpusd	gbpusd_pred	usdjpy	usdjpy_pred	gc	gc_pred	nq	
2020-05-15 15:43:00	1081535.000000	1081467.484800	1212550.000000	1212277.019402	107262500.000000	107273990.954676	1755.150000	1755.186882	8971.750000	896
2020-05-15 15:44:00	1081330.000000	1081469.177896	1212365.000000	1212294.513649	107273500.000000	107275652.980769	1754.950000	1755.218273	8979.500000	896
2020-05-15 15:45:00	1081385.000000	1081471.413731	1212360.000000	1212307.995259	107280000.000000	107278381.534239	1754.600000	1755.255400	8975.750000	896
2020-05-15 15:46:00	1081410.000000	1081472.219924	1212405.000000	1212320.866256	107286500.000000	107280758.402493	1755.350000	1755.298074	8972.750000	896
2020-05-15 15:47:00	1081430.000000	1081475.167970	1212050.000000	1212339.729484	107287000.000000	107282975.512584	1755.350000	1755.339047	8976.875000	896
2020-05-15 15:48:00	1081435.000000	1081477.829332	1212060.000000	1212356.542135	107284000.000000	107285148.058757	1755.750000	1755.379744	8974.500000	896
2020-05-15 15:49:00	1081435.000000	1081480.374206	1212060.000000	1212372.873424	107291500.000000	107287269.047323	1755.650000	1755.420356	8973.000000	896
2020-05-15	1081605.000000	1081482.835624	1212005.000000	1212389.279867	107291000.000000	107289334.346802	1755.750000	1755.460885	8982.250000	896

Key takeaways:

Error correction model is a dynamic model in which the change of the variable in the current time period is related to the distance between its value in the previous period and its value in the long-run equilibrium. Cointegration relations built into the specification of ECM which is kind of a long-term relation between time-series and variables and their rate of adjustment to the long-run equilibrium relationship.

VECM enables to use non stationary data (but cointegrated) for interpretation. This helps retain the relevant information in the data which would otherwise get missed on differencing of the same.

I can be reached *here*.

Complete code and data can be found here.

Econometric Modeling   Vector Autoregression   Statistical Analysis