# Are We Confident This Stock is Mean Reverting?

In the last chapter, you saw that the autocorelation of MSFT's weekly stock returns was -0.16. That autocorrelation seems large, but is it statistically significant? In other words, can you say that there is less than a 5% chance that we would observe such a large negative autocorrelation if the true autocorrelation were really zero? And are there any autocorrelations at other lags that are significantly different from zero?

Even if the true autocorrelations were zero at all lags, in a finite sample of returns you won't see the estimate of the autocorrelations exactly zero. In fact, the standard deviation of the sample autocorrelation is 1/sqrt(N)

where *N* is the number of observations, so if *N*=100, for example, the standard deviation of the ACF is 0.1, and since 95% of a normal curve is between +1.96 and -1.96 standard deviations from the mean, the 95% confidence interval is ±1.96/sqrt(N)

. This approximation only holds when the true autocorrelations are all zero.

You will compute the actual and approximate confidence interval for the ACF, and compare it to the lag-one autocorrelation of -0.16 from the last chapter. The weekly returns of Microsoft is pre-loaded in a DataFrame called returns.

In [2]: # Import the plot\_acf module from statsmodels and sqrt from math

from statsmodels.graphics.tsaplots import plot\_acf

from math import sqrt

# Compute and print the autocorrelation of MSFT weekly returns

autocorrelation = returns['Adj Close'].autocorr()

print("The autocorrelation of weekly MSFT returns is %4.2f" %(autocorrelation))

# Find the number of observations by taking the length of the returns DataFrame

nobs = len(returns)

# Compute the approximate confidence interval

conf = 1.96/nobs

print("The approximate confidence interval is +/- %4.2f" %(conf))

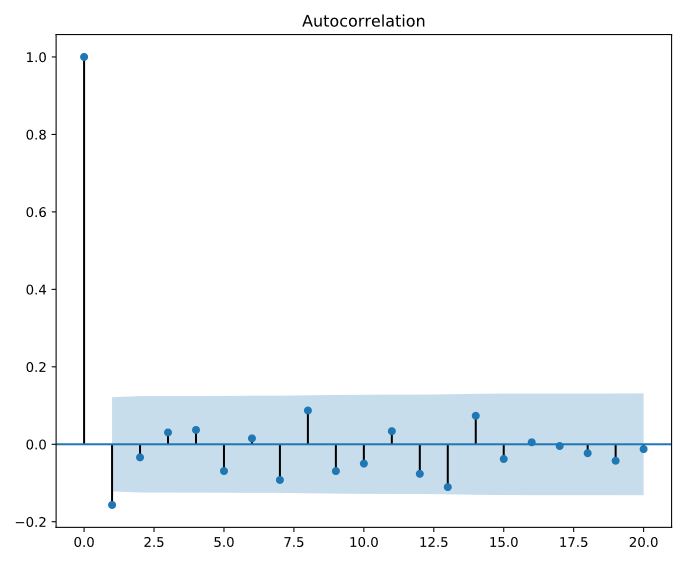
# Plot the autocorrelation function with 95% confidence intervals and 20 lags using plot\_acf

plot\_acf(returns, alpha=0.05, lags=20)

plt.show()

The autocorrelation of weekly MSFT returns is -0.16

The approximate confidence interval is +/- 0.01



# Can't Forecast White Noise

A white noise time series is simply a sequence of uncorrelated random variables that are identically distributed. Stock returns are often modelled as white noise. Unfortunately, for white noise, we cannot forecast future observations based on the past - autocorrelations at all lags are zero.

You will generate a white noise series and plot the autocorrelation function to show that it is zero for all lags. You can use np.random.normal() to generate random returns. For a Gaussian white noise process, the mean and standard deviation describe the entire process. Loc is the mean, scale is the std.

Plot this white noise series to see what it looks like, and then plot the autocorrelation function.

# Import the plot\_acf module from statsmodels

from statsmodels.graphics.tsaplots import plot\_acf

# Simulate wite noise returns

returns = np.random.normal(loc=0.02, scale=0.05, size=1000)

# Print out the mean and standard deviation of returns

mean = np.mean(returns)

std = np.std(returns)

print("The mean is %5.3f and the standard deviation is %5.3f" %(mean,std))

# Plot returns series

plt.plot(returns)

plt.show()

# Plot autocorrelation function of white noise returns

plot\_acf(returns, lags=20)

plt.show()

The mean is 0.020 and the standard deviation is 0.047

# Generate a Random Walk

Whereas stock returns are often modelled as white noise, stock prices closely follow a random walk. In other words, today's price is yesterday's price plus some random noise.

You will simulate the price of a stock over time that has a starting price of 100 and every day goes up or down by a random amount. Then, plot the simulated stock price. If you hit the "Run Code" code button multiple times, you'll see several realizations.

# Generate 500 random steps with mean=0 and standard deviation=1

steps = np.random.normal(loc=0, scale=1, size=500)

# Set first element to 0 so that the first price will be the starting stock price

steps[0]=0

# Simulate stock prices, P with a starting price of 100

P = 100 + np.cumsum(steps)

# Plot the simulated stock prices

plt.plot(P)

plt.title("Simulated Random Walk")

plt.show()

# Get the Drift

In the last exercise, you simulated stock prices that follow a random walk. You will extend this in two ways in this exercise.

* You will look at a random walk with a drift. Many time series, like stock prices, are random walks but tend to drift up over time.
* In the last exercise, the noise in the random walk was additive: random, normal changes in price were added to the last price. However, when adding noise, you could theoretically get negative prices. Now you will make the noise multiplicative: you will add one to the random, normal changes to get a total return, and multiply that by the last price.

# Generate 500 random steps

steps = np.random.normal(loc=0.001, scale=.01, size=500) + 1

# Set first element to 1

steps[0]=1

# Simulate the stock price, P, by taking the cumulative product

P = 100 \* np.cumprod(steps)

# Plot the simulated stock prices

plt.plot(P)

plt.title("Simulated Random Walk with Drift")

plt.show()

# Are Stock Prices a Random Walk?

Most stock prices follow a random walk (perhaps with a drift). You will look at a time series of Amazon stock prices, pre-loaded in the DataFrame AMZN, and run the 'Augmented Dickey-Fuller Test' from the statsmodels library to show that it does indeed follow a random walk.

With the ADF test, the "null hypothesis" (the hypothesis that we either reject or fail to reject) is that the series follows a random walk. Therefore, a low p-value (say less than 5%) means we can reject the null hypothesis that the series is a random walk.

Print out just the p-value of the test (results[0] is the test statistic, and results[1] is the p-value).

In [1]: # Import the adfuller module from statsmodels

from statsmodels.tsa.stattools import adfuller

# Run the ADF test on the price series and print out the results

results = adfuller(AMZN['Adj Close'])

print(results)

# Just print out the p-value

print('The p-value of the test on prices is: ' + str(results[1]))

(4.0251685257707477, 1.0, 33, 5054, {'5%': -2.8621120497269161, '1%': -3.4316445438146865, '10%': -2.5670745025321304}, 30308.642164269811)

The p-value of the test on prices is: 1.0, can’t reject Random walk assumption

# How About Stock Returns?

In the last exercise, you showed that Amazon stock prices, contained in the DataFrame AMZN follow a random walk. In this exercise. you will do the same thing for Amazon returns (percent change in prices) and show that the returns do not follow a random walk.

In [2]: # Import the adfuller module from statsmodels

from statsmodels.tsa.stattools import adfuller

# Create a DataFrame of AMZN returns

AMZN\_ret = AMZN.pct\_change()

# Eliminate the NaN in the first row of returns

AMZN\_ret = AMZN\_ret.dropna()

# Run the ADF test on the return series and print out the p-value

results = adfuller(AMZN\_ret['Adj Close'])

print('The p-value of the test on returns is: ' + str(results[1]))

The p-value of the test on returns is: 2.56558980835e-22, Not a Random Walk, but a white noise

Dickey Fuller Test: P(t) = a + b\*P(t-1) + e(t), and b = 1 (random walk)

equivalent to P(t)-P(t-1) = a + b\*P(t-1) + e(t) and b = 0 (random walk)

if more lags on the right hand side, it’s ADF test (augmented DF test)

# Simulate AR(1) Time Series

You will simulate and plot a few AR(1) time series, each with a different parameter, *ϕ*

, using the arima\_process module in statsmodels. In this exercise, you will look at an AR(1) model with a large positive *ϕ* and a large negative *ϕ*

, but feel free to play around with your own parameters.

There are a few conventions when using the arima\_process module that require some explanation. First, these routines were made very generally to handle both AR and MA models. We will cover MA models next, so for now, just ignore the MA part. Second, when inputting the coefficients, you must include the zero-lag coefficient of 1, and the sign of the other coefficients is opposite what we have been using (to be consistent with the time series literature in signal processing). For example, for an AR(1) process with *ϕ*=0.9

, the array representing the AR parameters would be ar = np.array([1, -0.9])

# import the module for simulating data

from statsmodels.tsa.arima\_process import ArmaProcess

# Plot 1: AR parameter = +0.9

plt.subplot(2,1,1)

ar1 = np.array([1, -0.9])

ma1 = np.array([1])

AR\_object1 = ArmaProcess(ar1, ma1)

simulated\_data\_1 = AR\_object1.generate\_sample(nsample=1000)

plt.plot(simulated\_data\_1)

# Plot 2: AR parameter = -0.9

plt.subplot(2,1,2)

ar2 = np.array([1, 0.9])

ma2 = np.array([1])

AR\_object2 = ArmaProcess(ar2, ma2)

simulated\_data\_2 = AR\_object2.generate\_sample(nsample=1000)

plt.plot(simulated\_data\_2)

plt.show()

# Compare the ACF for Several AR Time Series

The autocorrelation function decays exponentially for an AR time series at a rate of the AR parameter. For example, if the AR parameter, *ϕ*=+0.9

, the first-lag autocorrelation will be 0.9, the second-lag will be (0.9)2=0.81, the third-lag will be (0.9)3=0.729, etc. A smaller AR parameter will have a steeper decay, and for a negative AR parameter, say -0.9, the decay will flip signs, so the first-lag autocorrelation will be -0.9, the second-lag will be (−0.9)2=0.81, the third-lag will be (−0.9)3=−0.729

, etc.

The object simulated\_data\_1 is the simulated time series with an AR parameter of +0.9, simulated\_data\_2 is for an AR paramter of -0.9, and simulated\_data\_3 is for an AR parameter of 0.3

# Estimating an AR Model

You will estimate the AR(1) parameter, *ϕ*

, of one of the simulated series that you generated in the earlier exercise. Since the parameters are known for a simulated series, it is a good way to understand the estimation routines before applying it to real data.

For simulated\_data\_1 with a true *ϕ*

of 0.9, you will print out the estimate of *ϕ*. In addition, you will also print out the entire output that is produced when you fit a time series, so you can get an idea of what other tests and summary statistics are available in statsmodels.

# Import the ARMA module from statsmodels

from statsmodels.tsa.arima\_model import ARMA

# Fit an AR(1) model to the first simulated data

mod = ARMA(simulated\_data\_1, order=(1,0))

res = mod.fit()

# Print out summary information on the fit

print(res.summary())

# Print out the estimate for the constant and for phi

print("When the true phi=0.9, the estimate of phi (and the constant) are:")

print(res.params)

# Forecasting with an AR Model

In addition to estimating the parameters of a model that you did in the last exercise, you can also do forecasting, both in-sample and out-of-sample using statsmodels. The in-sample is a forecast of the next data point using the data up to that point, and the out-of-sample forecasts any number of data points in the future. These forecasts can be made using either the predict() method if you want the forecasts in the form of a series of data, or using the plot\_predict() method if you want a plot of the forecasted data. You supply the starting point for forecasting and the ending point, which can be any number of data points after the data set ends.

For the simulated series simulated\_data\_1 with *ϕ*=0.9

, you will plot in-sample and out-of-sample forecasts.

# Import the ARMA module from statsmodels

from statsmodels.tsa.arima\_model import ARMA

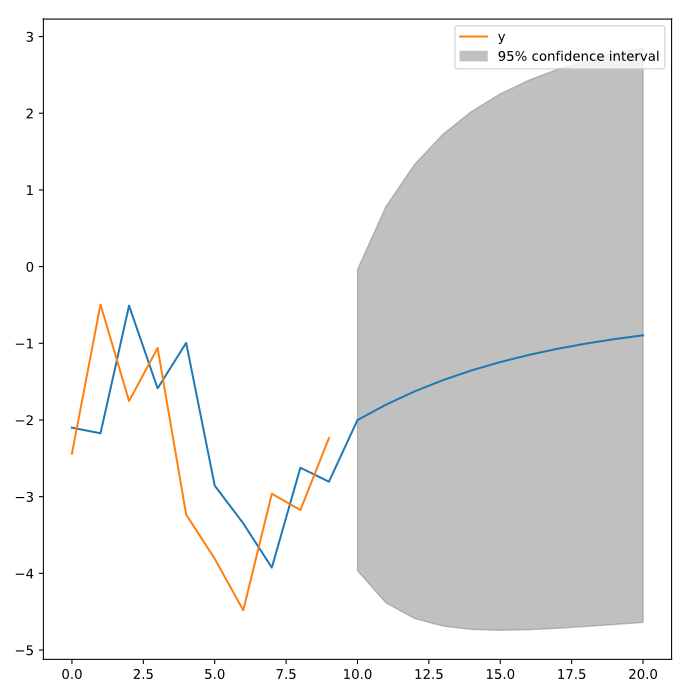
# Forecast the first AR(1) model

mod = ARMA(simulated\_data\_1, order=(1,0))

res = mod.fit()

res.plot\_predict(start=990, end=1010)

plt.show()



# Import the ARMA module from statsmodels

from statsmodels.tsa.arima\_model import ARMA

# Forecast interest rates using an AR(1) model

mod = ARMA(interest\_rate\_data, order=(1,0))

res = mod.fit()

# Plot the original series and the forecasted series

res.plot\_predict(start=0, end='2022')

plt.legend(fontsize=8)

plt.show()

# Compare AR Model with Random Walk

Sometimes it is difficult to distinguish between a time series that is slightly mean reverting and a time series that does not mean revert at all, like a random walk. You will compare the ACF for the slightly mean-reverting interest rate series of the last exercise with a simulated random walk with the same number of observations.

You should notice when plotting the autocorrelation of these two series side-by-side that they look very similar.

# Import the plot\_acf module from statsmodels

from statsmodels.graphics.tsaplots import plot\_acf

# Plot the interest rate series and the simulated random walk series side-by-side

fig, axes = plt.subplots(2,1)

# Plot the autocorrelation of the interest rate series in the top plot

fig = plot\_acf(interest\_rate\_data, alpha=1, lags=12, ax=axes[0])

# Plot the autocorrelation of the simulated random walk series in the bottom plot

fig = plot\_acf(simulated\_data, alpha=1, lags=12, ax=axes[1])

# Label axes

axes[0].set\_title("Interest Rate Data")

axes[1].set\_title("Simulated Random Walk Data")

plt.show()

# Estimate Order of Model: PACF

One useful tool to identify the order of an AR model is to look at the Partial Autocorrelation Function (PACF). In this exercise, you will simulate two time series, an AR(1) and an AR(2), and calculate the sample PACF for each. You will notice that for an AR(1), the PACF should have a significant lag-1 value, and roughly zeros after that. And for an AR(2), the sample PACF should have significant lag-1 and lag-2 values, and zeros after that.

Just like you used the plot\_acf function in earlier exercises, here you will use a function called plot\_pacf in the statsmodels module.

# Import the modules for simulating data and for plotting the PACF

from statsmodels.tsa.arima\_process import ArmaProcess

from statsmodels.graphics.tsaplots import plot\_pacf

# Simulate AR(1) with phi=+0.6

ma = np.array([1])

ar = np.array([1, -0.6])

AR\_object = ArmaProcess(ar, ma)

simulated\_data\_1 = AR\_object.generate\_sample(nsample=5000)

# Plot PACF for AR(1)

plot\_pacf(simulated\_data\_1, lags=20)

plt.show()

# Simulate AR(2) with phi1=+0.6, phi2=+0.3

ma = np.array([1])

ar = np.array([1, -.6, -.3])

AR\_object = ArmaProcess(ar, ma)

simulated\_data\_2 = AR\_object.generate\_sample(nsample=5000)

# Plot PACF for AR(2)

plot\_pacf(simulated\_data\_2, lags=20)

plt.show()

# Estimate Order of Model: Information Criteria

Another tool to identify the order of a model is to look at the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC). These measures compute the goodness of fit with the estimated parameters, but apply a penalty function on the number of parameters in the model. You will take the AR(2) simulated data from the last exercise, saved as simulated\_data\_2, and compute the BIC as you vary the order, p, in an AR(p) from 0 to 6.

# Import the module for estimating an ARMA model

from statsmodels.tsa.arima\_model import ARMA

# Fit the data to an AR(p) for p = 0,...,6 , and save the BIC

BIC = np.zeros(7)

for p in range(7):

mod = ARMA(simulated\_data\_2, order=(p,0))

res = mod.fit()

# Save BIC for AR(p)

BIC[p] = res.bic

# Plot the BIC as a function of p

plt.plot(range(1,7), BIC[1:7], marker='o')

plt.xlabel('Order of AR Model')

plt.ylabel('Baysian Information Criterion')

plt.show()

# Simulate MA(1) Time Series

You will simulate and plot a few MA(1) time series, each with a different parameter, *θ*

, using the arima\_process module in statsmodels, just as you did in the last chapter for AR(1) models. You will look at an MA(1) model with a large positive *θ* and a large negative *θ*

.

As in the last chapter, when inputting the coefficients, you must include the zero-lag coefficient of 1, but unlike the last chapter on AR models, the sign of the MA coefficients is what we would expect. For example, for an MA(1) process with *θ*=−0.9

, the array representing the MA parameters would be ma = np.array([1, -0.9])

# import the module for simulating data

from statsmodels.tsa.arima\_process import ArmaProcess

# Plot 1: MA parameter = -0.9

plt.subplot(2,1,1)

ar1 = np.array([1])

ma1 = np.array([1, -0.9])

MA\_object1 = ArmaProcess(ar1, ma1)

simulated\_data\_1 = MA\_object1.generate\_sample(nsample=1000)

plt.plot(simulated\_data\_1)

# Plot 2: MA parameter = +0.9

plt.subplot(2,1,2)

ar2 = np.array([1])

ma2 = np.array([1, .9])

MA\_object2 = ArmaProcess(ar2, ma2)

simulated\_data\_2 = MA\_object2.generate\_sample(nsample=1000)

plt.plot(simulated\_data\_2)

# Compute the ACF for Several MA Time Series

Unlike an AR(1), an MA(1) model has no autocorrelation beyond lag 1, an MA(2) model has no autocorrelation beyond lag 2, etc. The lag-1 autocorrelation for an MA(1) model is not *θ*

, but rather *θ*/(1+(*θ)2*). For example, if the MA parameter, *θ*, is = +0.9, the first-lag autocorrelation will be 0.9/(1+(0.9)2)=0.497, and the autocorrelation at all other lags will be zero. If the MA parameter, *θ*, is -0.9, the first-lag autocorrelation will be −0.9/(1+(−0.9)2)=−0.497

.

You will verify these autocorrelation functions for the three time series you generated in the last exercise.

# Estimating an MA Model

You will estimate the MA(1) parameter, *θ*

, of one of the simulated series that you generated in the earlier exercise. Since the parameters are known for a simulated series, it is a good way to understand the estimation routines before applying it to real data.

For simulated\_data\_1 with a true *θ*

of -0.9, you will print out the estimate of *θ*. In addition, you will also print out the entire output that is produced when you fit a time series, so you can get an idea of what other tests and summary statistics are available in statsmodels.

# Import the ARMA module from statsmodels

from statsmodels.tsa.arima\_model import ARMA

# Fit an MA(1) model to the first simulated data

mod = ARMA(simulated\_data\_1, order=(0,1))

res = mod.fit()

# Print out summary information on the fit

print(res.summary())

# Print out the estimate for the constant and for theta

print("When the true theta=-0.9, the estimate of theta (and the consant) are:")

print(res.params)

ARMA Model Results

==============================================================================

Dep. Variable: y No. Observations: 1000

Model: ARMA(0, 1) Log Likelihood -1422.950

Method: css-mle S.D. of innovations 1.003

Date: Thu, 05 Apr 2018 AIC 2851.899

Time: 05:39:45 BIC 2866.623

Sample: 0 HQIC 2857.495

==============================================================================

coef std err z P>|z| [0.025 0.975]

------------------------------------------------------------------------------

const -0.0060 0.002 -2.572 0.010 -0.011 -0.001

ma.L1.y -0.9277 0.011 -87.374 0.000 -0.949 -0.907

Roots

=============================================================================

Real Imaginary Modulus Frequency

-----------------------------------------------------------------------------

MA.1 1.0779 +0.0000j 1.0779 0.0000

-----------------------------------------------------------------------------

When the true theta=-0.9, the estimate of theta (and the consant) are:

[-0.00597352 -0.92773541]

# Forecasting with MA Model

As you did with AR models, you will use MA models to forecast in-sample and out-of-sample data using statsmodels.

For the simulated series simulated\_data\_1 with *θ*=−0.9

, you will plot in-sample and out-of-sample forecasts. One big difference you will see between out-of-sample forecasts with an MA(1) model and an AR(1) model is that the MA(1) forecasts more than one period in the future are simply the mean of the sample.

# High Frequency Stock Prices

Higher frequency stock data is well modelled by an MA(1) process, so it's a nice application of the models in this chapter.

The DataFrame intraday contains one day's prices (on September 1, 2017) for Sprint stock (ticker symbol "S") sampled at a frequency of one minute. The stock market is open for 6.5 hours (390 minutes), from 9:30am to 4:00pm.

Before you can analyze the time series data, you will have to clean it up a little, which you will do in this and the next two exercises. When you look at the first few rows (see the Ipython Shell), you'll notice several things. First, there are no column headers.The data is not time stamped from 9:30 to 4:00, but rather goes from 0 to 390. And you will notice that the first date is the odd-looking "a1504272600". The number after the "a" is Unix time which is the number of seconds since January 1, 1970. This is how this dataset separates each day of intraday data.

If you look at the data types, you'll notice that the DATE column is an object, which here means a string. You will need to change that to numeric before you can clean up some missing data.

The source of the minute data is Google Finance (see [here](https://www.quantshare.com/sa-426-6-ways-to-download-free-intraday-and-tick-data-for-the-us-stock-market) on how the data was downloaded).

The datetime module has already been imported for you.

# import datetime module

import datetime

# Change the first date to zero

intraday.iloc[0,0] = 0

# Change the column headers to 'DATE' and 'CLOSE'

intraday.columns = ['DATE','CLOSE']

# Examine the data types for each column

print(intraday.dtypes)

# Convert DATE column to numeric

intraday['DATE'] = pd.to\_numeric(intraday['DATE'])

# Make the `DATE` column the new index

intraday = intraday.set\_index('DATE')

# More Data Cleaning: Missing Data

When you print out the length of the DataFrame intraday, you will notice that a few rows are missing. There will be missing data if there are no trades in a particular one-minute interval. One way to see which rows are missing is to take the difference of two sets: the full set of every minute and the set of the DataFrame index which contains missing rows. You can fill in the missing rows with the .reindex() method, convert the index to time of day, and then plot the data.

Stocks trade at discrete one-cent increments (although a small percentage of trades occur in between the one-cent increments) rather than at continuous prices, and when you plot the data you should observe that there are long periods when the stock bounces back and forth over a one cent range. This is sometimes referred to as "bid/ask bounce".

In [1]: # Notice that some rows are missing

print("The length of the DataFrame is: ",len(intraday))

# Find the missing rows

print("Missing rows: ", set(range(391)) - set(intraday.index))

# Fill in the missing rows

intraday = intraday.reindex(range(391), method='ffill')

# Change the index to the intraday times

intraday.index = pd.date\_range(start='2017-08-28 9:30', end='2017-08-28 16:00', freq='1min')

# Plot the intraday time series

intraday.plot(grid=True)

plt.show()

The length of the DataFrame is: 389

Missing rows: {182, 14}

# Applying an MA Model

The bouncing of the stock price between bid and ask induces a negative first order autocorrelation, but no autocorrelations at lags higher than 1. You get the same ACF pattern with an MA(1) model. Therefore, you will fit an MA(1) model to the intraday stock data from the last exercise.

The first step is to compute minute-by-minute returns from the prices in intraday, and plot the autocorrelation function. You should observe that the ACF looks like that for an MA(1) process. Then, fit the data to an MA(1), the same way you did for simulated data.

In [1]: # Import plot\_acf and ARMA modules from statsmodels

from statsmodels.graphics.tsaplots import plot\_acf

from statsmodels.tsa.arima\_model import ARMA

# Compute returns from prices and drop the NaN

returns = intraday.pct\_change()

returns = returns.dropna()

# Plot ACF of returns with lags up to 60 minutes

plot\_acf(returns, lags=60)

plt.show()

# Fit the data to an MA(1) model

mod = ARMA(returns, order=(0,1))

res = mod.fit()

print(res.params)

const -0.000002

ma.L1.CLOSE -0.179272

dtype: float64

# Equivalence of AR(1) and MA(infinity)

To better understand the relationship between MA models and AR models, you will demonstrate that an AR(1) model is equivalent to an MA(∞

) model with the appropriate parameters.

You will simulate an MA model with parameters 0.8,0.82,0.83,…

for a large number (30) lags and show that it has the same Autocorrelation Function as an AR(1) model with *ϕ*=0.8.

# import the modules for simulating data and plotting the ACF

from statsmodels.tsa.arima\_process import ArmaProcess

from statsmodels.graphics.tsaplots import plot\_acf

# Build a list MA parameters

ma = [0.8\*\*i for i in range(30)]

# Simulate the MA(30) model

ar = np.array([1])

AR\_object = ArmaProcess(ar, ma)

simulated\_data = AR\_object.generate\_sample(nsample=5000)

# Plot the ACF

plot\_acf(simulated\_data, lags=30)

plt.show()

# A Dog on a Leash? (Part 1)

The Heating Oil and Natural Gas prices are pre-loaded in DataFrames HO and NG. First, plot both price series, which look like random walks. Then plot the difference between the two series, which should look more like a mean reverting series (to put the two series in the same units, we multiply the heating oil prices, in $/gallon, by 7.25, which converts it to $/millionBTU, which is the same units as Natural Gas).

The data for continuous futures (each contract has to be spliced together in a continuous series as contracts expire) was obtained from [Quandl](https://blog.quandl.com/api-for-futures-data).

# Plot the prices separately

plt.subplot(2,1,1)

plt.plot(7.25\*HO, label='Heating Oil')

plt.plot(NG, label='Natural Gas')

plt.legend(loc='best', fontsize='small')

# Plot the spread

plt.subplot(2,1,2)

plt.plot(7.25\*HO-NG, label='Spread')

plt.legend(loc='best', fontsize='small')

plt.axhline(y=0, linestyle='--', color='k')

plt.show()

# A Dog on a Leash? (Part 2)

To verify that HO and NG are cointegrated, First apply the Dickey-Fuller test to HO and NG separately to show they are random walks. Then apply the test to the difference, which should strongly reject the random walk hypothesis. The Heating Oil and Natural Gas prices are pre-loaded in DataFrames HO and NG.

In [1]: # Import the adfuller module from statsmodels

from statsmodels.tsa.stattools import adfuller

# Compute the ADF for HO and NG

result\_HO = adfuller(HO['Close'])

print("The p-value for the ADF test on HO is ", result\_HO[1])

result\_NG = adfuller(NG['Close'])

print("The p-value for the ADF test on NG is ", result\_NG[1])

# Compute the ADF of the spread

result\_spread = adfuller(7.25 \* HO['Close'] - NG['Close'])

print("The p-value for the ADF test on the spread is ", result\_spread[1])

The p-value for the ADF test on HO is 0.956710878502

The p-value for the ADF test on NG is 0.900874744468

The p-value for the ADF test on the spread is 7.01943930214e-05

# Are Bitcoin and Ethereum Cointegrated?

Cointegration involves two steps: regressing one time series on the other to get the cointegration vector, and then perform an ADF test on the residuals of the regression. In the last example, there was no need to perform the first step since we implicitly assumed the cointegration vector was (1,−1)

. In other words, we took the difference between the two series (after doing a units conversion). Here, you will do both steps.

You will regress the value of one crytocurrency, bitcoin (BTC), on another cryptocurrency, ethereum (ETH). If we call the regression coeffiecient *b*

, then the cointegration vector is simply (1,−*b*). Then perform the ADF test on BTC −*b*

ETH. Bitcoin and Ethereum prices are pre-loaded in DataFrames BTC and ETH.

Bitcoin data are in DataFrame BTC and Ethereum data are in ETH.

In [3]: # Import the statsmodels module for regression and the adfuller function

import statsmodels.api as sm

from statsmodels.tsa.stattools import adfuller

# Regress BTC on ETH

ETH = sm.add\_constant(ETH)

result = sm.OLS(BTC,ETH).fit()

# Compute ADF

b = result.params[1]

adf\_stats = adfuller(BTC['Price'] - b\*ETH['Price'])

print("The p-value for the ADF test is ", adf\_stats[1])

The p-value for the ADF test is 0.0233690023235

# Import the modules for plotting the sample ACF and PACF

from statsmodels.graphics.tsaplots import plot\_acf, plot\_pacf

# Take first difference of the temperature Series

chg\_temp = temp\_NY.diff()

chg\_temp = chg\_temp.dropna()

# Plot the ACF and PACF on the same page

fig, axes = plt.subplots(2,1)

# Plot the ACF

plot\_acf(chg\_temp, lags=20, ax=axes[0])

# Plot the PACF

plot\_pacf(chg\_temp, lags=20, ax=axes[1])

plt.show()

# Import the module for estimating an ARMA model

from statsmodels.tsa.arima\_model import ARMA

# Fit the data to an AR(1) model and print AIC:

mod = ARMA(chg\_temp, order=(1,0))

res = mod.fit()

print("The AIC for an AR(1) is: ", res.aic)

# Fit the data to an AR(2) model and print AIC:

mod = ARMA(chg\_temp, order=(2,0))

res = mod.fit()

print("The AIC for an AR(2) is: ", res.aic)

# Fit the data to an MA(1) model and print AIC:

mod = ARMA(chg\_temp, order=(0,1))

res = mod.fit()

print("The AIC for an MA(1) is: ", res.aic)

# Fit the data to an ARMA(1,1) model and print AIC:

mod = ARMA(chg\_temp, order=(1,1))

res = mod.fit()

print("The AIC for an ARMA(1,1) is: ", res.aic)

# Manipulate Time Series

You have learned in the video how to create a sequence of dates using pd.date\_range(). You have also seen that each date in the resulting pd.DatetimeIndex is a pd.Timestamp with various attributes that you can access to obtain information about the date.

Now, you'll create a week of data, iterate over the result, and obtain the dayofweek and weekday\_name for each date.

# Create the range of dates here

seven\_days = pd.date\_range('2017-1-1', periods=7)

# Iterate over the dates and print the number and name of the weekday

for day in seven\_days:

print(day.dayofweek, day.weekday\_name)

**How to plot multiple curves in the same plot (if diff plots, then use subplot=True)**

# Create dataframe prices here

prices = pd.DataFrame()

# Select data for each year and concatenate with prices here

for year in ['2013', '2014', '2015']:

price\_per\_year = yahoo.loc[year, ['price']].reset\_index(drop=True)

price\_per\_year.rename(columns={'price': year}, inplace=True)

prices = pd.concat([prices, price\_per\_year], axis=1)

# Plot prices

prices.plot()

plt.show();

# Plot performance difference vs benchmark index

In the video, you learned how to calculate and plot the performance difference of a stock in percentage points relative to a benchmark index.

Let's compare the performance of Microsoft (MSFT) and Apple (AAPL) to the S&P 500 over the last 10 years.

# Create tickers

tickers = ['MSFT', 'AAPL']

# Import stock data here

stocks = pd.read\_csv('msft\_aapl.csv', parse\_dates=['date'], index\_col='date')

# Import index here

sp500 = pd.read\_csv('sp500.csv', parse\_dates=['date'], index\_col='date')

# Concatenate stocks and index here

data = pd.concat([stocks, sp500], axis=1).dropna()

# Normalize data

normalized = data.div(data.iloc[0]).mul(100)

# Subtract the normalized index from the normalized stock prices, and plot the result

normalized[tickers].sub(normalized['SP500'], axis=0).plot()

plt.show()

# Cumulative return on $1,000 invested in google vs apple II

Apple outperformed Google over the entire period, but this may have been different over various 1-year sub periods, so that switching between the two stocks might have yielded an even better result.

To analyze this, calculate that cumulative return for rolling 1-year periods, and then plot the returns to see when each stock was superior.

# Import numpy

import numpy as np

# Define a multi\_period\_return function

def multi\_period\_return(period\_returns):

return np.prod(period\_returns + 1) - 1

# Calculate daily returns

daily\_returns = data.pct\_change()

# Calculate rolling\_annual\_returns

rolling\_annual\_returns = daily\_returns.rolling('360D').apply(multi\_period\_return)

# Plot rolling\_annual\_returns

rolling\_annual\_returns.mul(100).plot();

plt.show()