**1. Introduction to the Course**

00:00 - 00:51

Welcome to the first video of the "Introduction to Time Series Analysis Using Python" course. My name is Rob Reider. I'm an Adjunct Professor in the Math-Finance Master's program at NYU's Courant Institute, where I teach a course on Time Series Analysis. I'm also a consultant to a company called Quantopian, which has built a Python-based platform for analyzing and backtesting quantitative trading strategies. Authors of algorithms can enter into paper trading contests and be considered for an allocation of money. Authors receiving allocations are paid 10 percent of the strategy’s net profits, based on their strategy’s individual performance. Also, Quantopian hosts a community where members can ask for help, share ideas, and discuss and share code.

**2. Example of Time Series: Google Trends**

00:51 - 01:24

Time series analysis deals with data that is ordered in time. Of course, there are many other types of data that are not covered in this course - for example, cross-sectional data that are taken at one point in time. Time series come up in many contexts. Here is a time series of the frequency of Google searches for the word "diet" over a five year period. You can see an interesting pattern: it hits a low around the holidays, and then spikes up at the beginning of the year when people make New Year's resolutions to lose weight.

**3. Example of Time Series: Climate Data**

01:24 - 01:35

Here is another example of a time series: the average annual temperature in New York City since 1870. Notice that this time series is trending up. Many of the most interesting applications of time series analysis are financial time series. In this course, you will look at a variety of financial time series: stocks, bonds, commodities, even crytpocurrencies like Bitcoin.

**4. Example of Time Series: Quarterly Earnings Data**

01:35 - 02:11

Here is the time series of quarterly earnings for the company H&R Block. H&R Block is in the business of preparing tax returns for customers and selling tax software. The vast majority of their earnings occurs in the quarter that taxes are due. Notice the strong seasonality pattern in the earnings.

**5. Example of Multiple Series: Natural Gas and Heating Oil**

02:11 - 02:24

You will also look at two related series in the last chapter of this course. Here are the prices of two energy commodities, heating oil and natural gas, which move together.

**6. Goals of Course**

02:24 - 02:51

In this course, you will learn about various time series models, fit the data to these models, and use these models to make forecasts of the future. You will also learn how to use various statistical packages in Python to perform these tasks. Numerous examples will be provided, and I hope that these examples not only demonstrate how to apply these tools, but also address some interesting puzzles, mainly in the field of finance.

**7. Some Useful Pandas Tools**

02:51 - 03:23

In the course of analyzing time series data, you will use several convenient pandas tools for manipulating time series data. These methods will be used repeatedly throughout the course, so we will highlight a few of them now: to\_datetime() is used to convert an index, often read in as a string, into a datetime index. The plot method of pandas is a quick way to plot data, and if the index has been converted to a datetime object, you can slice the data by year, for example. You will sometimes need to merge or join two DataFrames.

**8. Some Useful Pandas Tools**

03:23 - 03:43

For example, one DataFrame may contain stock prices and another DataFrame may contain bond prices. Pandas makes it easy to resample data. For example, a DataFrame of daily data can be converted to weekly data with the resample method

**9. More pandas Functions**

03:43 - 04:05

Often, you will want to convert prices to returns, which you can do with the pct\_change method. Or if you want differences, you can use the diff method. You can compute the correlation of two series using the corr method, and the autocorrelation using the autocorr method. You'll learn more about these methods later in this chapter.

**10. Let's practice!**

04:05 - 04:12

Now let's practice using a few of these time series tools.

# A "Thin" Application of Time Series

[Google Trends](https://trends.google.com/trends/) allows users to see how often a term is searched for. We downloaded a file from Google Trends containing the frequency over time for the search word "diet", which is pre-loaded in a DataFrame called diet. A first step when analyzing a time series is to visualize the data with a plot. You should be able to clearly see a gradual decrease in searches for "diet" throughout the calendar year, hitting a low around the December holidays, followed by a spike in searches around the new year as people make New Year's resolutions to lose weight.

Like many time series datasets you will be working with, the index of dates are strings and should be converted to a datetime index before plotting.

This course touches on a lot of concepts you may have forgotten, so if you ever need a quick refresher, download the [*pandas basics Cheat Sheet*](https://datacamp-community-prod.s3.amazonaws.com/fbc502d0-46b2-4e1b-b6b0-5402ff273251) and keep it handy!

## Instructions 1/3

**Convert the date index to datetime using pandas's to\_datetime().**

**# Import pandas and plotting modules**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**# Convert the date index to datetime**

**diet.index = pd.to\_datetime(diet.index)**

**Plot the time series and set the argument grid to True to better see the year-ends.**

**# From previous step**

**diet.index = pd.to\_datetime(diet.index)**

**# Plot the entire time series diet and show gridlines**

**diet.plot(grid=True)**

**plt.show()**

**Slice the diet dataset to keep only values from 2012, assigning to diet2012.**

* **Plot the diet2012, again creating gridlines with the grid argument.**

**# From previous step**

**diet.index = pd.to\_datetime(diet.index)**

**# Slice the dataset to keep only 2012**

**diet2012 = diet['2012']**

**# Plot 2012 data diet2012.plot(grid=True)**

**plt.show()**

**Notice how searches for 'diet' spiked up after the holidays every year.**

## Exercise

# Merging Time Series With Different Dates

Stock and bond markets in the U.S. are closed on different days. For example, although the bond market is closed on Columbus Day (around Oct 12) and Veterans Day (around Nov 11), the stock market is open on those days. One way to see the dates that the stock market is open and the bond market is closed is to convert both indexes of dates into sets and take the difference in sets.

The pandas .join() method is a convenient tool to merge the stock and bond DataFrames on dates when both markets are open.

Stock prices and 10-year US Government bond yields, which were downloaded from [FRED](https://fred.stlouisfed.org/), are pre-loaded in DataFrames stocks and bonds.

## Instructions

* Convert the dates in the stocks.index and bonds.index into sets.
* Take the difference of the stock set minus the bond set to get those dates where the stock market has data but the bond market does not.
* Merge the two DataFrames into a new DataFrame, stocks\_and\_bonds using the .join() method, which has the syntax df1.join(df2).
  + To get the intersection of dates, use the argument how='inner'.

**# Import pandas**

**import pandas as pd**

**# Convert the stock index and bond index into sets**

**set\_stock\_dates = set(stocks.index)**

**set\_bond\_dates = set(bonds.index)**

**# Take the difference between the sets and print**

**print(set\_stock\_dates - set\_bond\_dates)**

**# Merge stocks and bonds DataFrames using join()**

**stocks\_and\_bonds = stocks.join(bonds, how='inner')**

**{'2010-10-11', '2015-10-12', '2014-11-11', '2007-10-08', '2013-11-11', '2014-10-13', '2017-06-09', '2016-10-10', '2009-10-12', '2007-11-12', '2011-11-11', '2015-11-11', '2012-11-12', '2010-11-11', '2012-10-08', '2008-10-13', '2008-11-11', '2011-10-10', '2016-11-11', '2013-10-14', '2009-11-11'}**

<script.py> output:

{'2010-10-11', '2015-10-12', '2014-11-11', '2007-10-08', '2013-11-11', '2014-10-13', '2017-06-09', '2016-10-10', '2009-10-12', '2007-11-12', '2011-11-11', '2015-11-11', '2012-11-12', '2010-11-11', '2012-10-08', '2008-10-13', '2008-11-11', '2011-10-10', '2016-11-11', '2013-10-14', '2009-11-11'

**Pandas helps make many time series tasks quick and efficient.**

**Correlation of 2 Time Series**

**1. Correlation of Two Time Series**

00:00 - 00:05

Often, two time series vary together.

**2. Correlation of Two Time Series**

00:05 - 00:19

Often, two time series vary together. Here is a plot of the stock prices of JP Morgan and the S&P500. You can see from this plot that, in general, when the market drops, JP Morgan drops as well, and when the market rises, JP Morgan also rises.

**3. Correlation of Two Time Series**

00:19 - 00:30

A scatter plot of the returns of JP Morgan and the returns of the market help to visualize the relationship between the two time series.

**4. More Scatter Plots**

00:30 - 01:05

The correlation coefficient is a measure of how much two series vary together. A correlation of one means that the two series have a perfect linear relationship with no deviations. High correlations mean that the two series strongly vary together. A low correlation means they vary together, but there is a weak association. And a high negative correlation means they vary in opposite directions, but still with a linear relationship.

**5. Common Mistake: Correlation of Two Trending Series**

01:05 - 01:46

Consider two time series that are both trending. Even if the two series are totally unrelated, you could still get a very high correlation. That's why, when you look at the correlation of say, two stocks, you should look at the correlation of their \*returns\*, not their levels. In this example, the two series, stock prices and UFO sightings, both trend up over time. Of course, there is no relationship between those two series, but the correlation is 0-point-94. If you compute the correlation of percent changes, the correlation goes down to approximately zero.

**6. Example: Correlation of Large Cap and Small Cap Stocks**

01:46 - 02:09

Now you'll see how to compute the correlation of two financial time series, the S&P500 index of large cap stocks and the Russell 2000 index of small cap stocks, using the pandas correlation method. First compute the percent changes using the pct\_change method. This gives the returns of these series instead of prices.

**7. Example: Correlation of Large Cap and Small Cap Stocks**

02:09 - 02:15

You can also visualize the correlation with a scatter plot.

**8. Example: Correlation of Large Cap and Small Cap Stocks**

02:15 - 02:25

Then, use the pandas correlation method for Series. The correlation between large cap and small cap stocks is very high: 0-point-868

**9. Let's practice!**

02:25 - 02:31

Time to put this into practice.

# Correlation of Stocks and Bonds

Investors are often interested in the correlation between the returns of two different assets for asset allocation and hedging purposes. In this exercise, you'll try to answer the question of whether stocks are positively or negatively correlated with bonds. Scatter plots are also useful for visualizing the correlation between the two variables.

**Keep in mind that you should compute the correlations on the percentage changes rather than the levels.**

Stock prices and 10-year bond yields are combined in a DataFrame called stocks\_and\_bonds under columns SP500 and US10Y

The pandas and plotting modules have already been imported for you. For the remainder of the course, pandas is imported as pd and matplotlib.pyplot is imported as plt.

## Instructions

100 XP

* Compute percent changes on the stocks\_and\_bonds DataFrame using the .pct\_change() method and call the new DataFrame returns.
* Compute the correlation of the columns SP500 and US10Y in the returns DataFrame using the .corr() method for Series which has the syntax series1.corr(series2).
* Show a scatter plot of the percentage change in stock and bond yields.
* **# Compute percent change using pct\_change()**
* **returns = stocks\_and\_bonds.pct\_change()**
* **# Compute correlation using corr()**
* **correlation = returns['SP500'].corr(returns['US10Y'])**
* **print("Correlation of stocks and interest rates: ", correlation)**
* **# Make scatter plot**
* **plt.scatter(returns['SP500'], returns['US10Y'])**
* **plt.show()**
* **Correlation of stocks and interest rates: 0.4119448886249272**
* **<script.py> output:**
* **Correlation of stocks and interest rates: 0.4119448886249272**

**The positive correlation means that when interest rates go down, stock prices go down. For example, during crises like 9/11, investors sold stocks and moved their money to less risky bonds (this is sometimes referred to as a 'flight to quality'). During these periods, stocks drop and interest rates drop as well. Of course, there are times when the opposite relationship holds too.**

# Flying Saucers Aren't Correlated to Flying Markets

Two trending series may show a strong correlation even if they are completely unrelated. This is referred to as "spurious correlation". That's why when you look at the correlation of say, two stocks, you should look at the correlation of their returns and not their levels.

To illustrate this point, calculate the correlation between the levels of the stock market and the annual sightings of UFOs. Both of those time series have trended up over the last several decades, and the correlation of their levels is very high. Then calculate the correlation of their percent changes. This will be close to zero, since there is no relationship between those two series.

The DataFrame levels contains the levels of DJI and UFO. UFO data was downloaded from [www.nuforc.org](http://www.nuforc.org).

## Instructions

100 XP

* Calculate the correlation of the columns DJI and UFO.
* Create a new DataFrame of changes using the .pct\_change() method.
* Re-calculate the correlation of the columns DJI and UFO on the changes.

**Great work! Notice that the correlation on levels is high but the correlation on changes is close to zero.**

**# Compute correlation of levels**

**correlation1 = levels['DJI'].corr(levels['UFO'])**

**print("Correlation of levels: ", correlation1)**

**# Compute correlation of percent changes**

**changes = levels.pct\_change()**

**correlation2 = changes['DJI'].corr(changes['UFO'])**

**print("Correlation of changes: ", correlation2)**

**Correlation of levels: 0.9399762210726432**

**Correlation of changes: 0.06026935462405376**

**<script.py> output:**

**Correlation of levels: 0.9399762210726432**

**Correlation of changes: 0.06026935462405376**

**1. Simple Linear Regressions**

00:00 - 00:07

In this video you'll learn about simple linear regressions of time series.

**2. What is a Regression?**

00:07 - 00:25

A simple linear regression finds the slope, beta, and intercept, alpha, of a line that's the best fit between a dependent variable, y, and an independent variable, x. The x's and y's can be two time series.

**3. What is a Regression?**

00:25 - 00:40

A linear regression is also known as Ordinary Least Squares, or OLS, because it minimizes the sum of the squared distances between the data points and the regression line.

**4. Python Packages to Perform Regressions**

00:40 - 01:19

Regression techniques are very common, and therefore there are many packages in Python that can be used. In statsmodels, there is OLS. In numpy, there is polyfit, and if you set degree equals 1, it fits the data to a line, which is a linear regression. Pandas has an ols method, and scipy has a linear regression function. Beware that the order of x and y is not consistent across packages. All these packages are very similar, and in this course, you will use the statsmodels OLS.

**5. Example: Regression of Small Cap Returns on Large Cap**

01:19 - 02:02

Now you'll regress the returns of the small cap stocks on the returns of large cap stocks. Compute returns from prices using the "pct\_change" method in pandas. You need to add a column of ones as a dependent, right hand side variable. The reason you have to do this is because the regression function assumes that if there is no constant column, then you want to run the regression without an intercept. By adding a column of ones, statsmodels will compute the regression coefficient of that column as well, which can be interpreted as the intercept of the line. The statsmodels method "add constant" is a simple way to add a constant.

**6. Regression Example (continued)**

02:02 - 02:52

Notice that the first row of the return series is NaN. Each return is computed from two prices, so there is one less return than price. To delete the first row of NaN's, use the pandas method "dropna". You're finally ready to run the regression. The first argument of the statsmodel regression is the series that represents the dependent variable, y, and the next argument contains the independent variable or variables. In this case, the dependent variable is the R2000 returns and the independent variables are the constant and SPX returns. The method "fit" runs the regression and results are saved in a class instance called results.

**7. Regression Example (continued)**

02:52 - 03:28

The summary method of results shows the entire regression output. We will only focus on a few items of the regression results. In the red box, the coefficent 1-point-1412 is the slope of the regression, which is also referred to as beta. The coefficient above that is the intercept, which is very close to zero. You can also pull out individual items from results, like the intercept, in results-dot-params zero, and the slope, in results-dot-params one.

**8. Regression Example (continued)**

03:28 - 03:36

Another statistic to take note of is the R-Squared of 0-point-753. That will be discussed next.

**9. Relationship Between R-Squared and Correlation**

03:36 - 04:33

From the scatter diagrams, you saw that the correlation measures how closely the data are clustered along a line. The R-squared also measures how well the linear regression line fits the data. So as you would expect, there is a relationship between correlation and R-squared. The magnitude of the correlation is the square root of the R-squared. And the sign of the correlation is the sign of the slope of the regression line. If the regression line is positively sloped, the correlation is positive and if the the regression line is negatively sloped, the correlation is negative. In the example you just analyzed, of large cap and small cap stocks, the R-Squared was 0-point-753, the slope of the regression was positive, so the correlation is then positive the square root of 0-point-753, or 0-point-868, which can be verified by computing the correlation directly.

**10. Let's practice!**

04:33 - 04:38

Now it's your turn.

# Looking at a Regression's R-Squared

**R-squared measures how closely the data fit the regression line,** so the R-squared in a simple regression is related to the correlation between the two variables. In particular, the magnitude of the correlation is the square root of the R-squared and the sign of the correlation is the sign of the regression coefficient.

In this exercise, you will start using the statistical package statsmodels, which performs much of the statistical modeling and testing that is found in R and software packages like SAS and MATLAB.

You will take two series, x and y, compute their correlation, and then regress y on x using the function OLS(y,x) in the statsmodels.api library (note that the dependent, or right-hand side variable y is the first argument). Most linear regressions contain a constant term which is the intercept (the

in the regression

). To include a constant using the function OLS(), you need to add a column of 1's to the right hand side of the regression.

The module statsmodels.api has been imported for you as sm.

## Instructions

100 XP

* Compute the correlation between x and y using the .corr() method.
* Run a regression:
  + First convert the Series x to a DataFrame dfx.
  + Add a constant using sm.add\_constant(), assigning it to dfx1
  + Regress y on dfx1 using sm.OLS().fit().
* Print out the results of the regression and compare the R-squared with the correlation.

**# Import the statsmodels module**

**import statsmodels.api as sm**

**# Compute correlation of x and y**

**correlation = x.corr(y)**

**print("The correlation between x and y is %4.2f" %(correlation))**

**# Convert the Series x to a DataFrame and name the column x**

**dfx = pd.DataFrame(x, columns=['x'])**

**# Add a constant to the DataFrame dfx**

**dfx1 = sm.add\_constant(dfx)**

**# Regress y on dfx1**

**result = sm.OLS(y, dfx1).fit()**

**# Print out the results and look at the relationship between R-squared and the correlation above**

**print(result.summary())**

**The correlation between x and y is -0.90**

**OLS Regression Results**

**==============================================================================**

**Dep. Variable: y R-squared: 0.818**

**Model: OLS Adj. R-squared: 0.817**

**Method: Least Squares F-statistic: 4471.**

**Date: Fri, 12 May 2023 Prob (F-statistic): 0.00**

**Time: 01:06:58 Log-Likelihood: -560.10**

**No. Observations: 1000 AIC: 1124.**

**Df Residuals: 998 BIC: 1134.**

**Df Model: 1**

**Covariance Type: nonrobust**

**==============================================================================**

**coef std err t P>|t| [0.025 0.975]**

**------------------------------------------------------------------------------**

**const -0.0052 0.013 -0.391 0.696 -0.032 0.021**

**x -0.9080 0.014 -66.869 0.000 -0.935 -0.881**

**==============================================================================**

**Omnibus: 0.048 Durbin-Watson: 2.066**

**Prob(Omnibus): 0.976 Jarque-Bera (JB): 0.103**

**Skew: -0.003 Prob(JB): 0.950**

**Kurtosis: 2.951 Cond. No. 1.03**

**==============================================================================**

**Notes:**

**[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.**

**<script.py> output:**

**The correlation between x and y is -0.90**

**OLS Regression Results**

**==============================================================================**

**Dep. Variable: y R-squared: 0.818**

**Model: OLS Adj. R-squared: 0.817**

**Method: Least Squares F-statistic: 4471.**

**Date: Fri, 12 May 2023 Prob (F-statistic): 0.00**

**Time: 01:08:46 Log-Likelihood: -560.10**

**No. Observations: 1000 AIC: 1124.**

**Df Residuals: 998 BIC: 1134.**

**Df Model: 1**

**Covariance Type: nonrobust**

**==============================================================================**

**coef std err t P>|t| [0.025 0.975]**

**------------------------------------------------------------------------------**

**const -0.0052 0.013 -0.391 0.696 -0.032 0.021**

**x -0.9080 0.014 -66.869 0.000 -0.935 -0.881**

**==============================================================================**

**Omnibus: 0.048 Durbin-Watson: 2.066**

**Prob(Omnibus): 0.976 Jarque-Bera (JB): 0.103**

**Skew: -0.003 Prob(JB): 0.950**

**Kurtosis: 2.951 Cond. No. 1.03**

**==============================================================================**

**Notes:**

**[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.**

**Notice that the two different methods of computing correlation give the same result. The correlation is about -0.9 and the R-squared is about 0.81.**

Daily XP400

# Match Correlation with Regression Output

Here are four scatter plots, each showing a linear regression line and an R-squared:

A picture containing text, line, plot, diagram

Description automatically generated

Which correlation is correct

#### Possible Answers

* Fig 1: correlation = -0.6
* Fig 2: correlation = -0.9
* **Fig 3: correlation = -0.9**
* Fig 4: correlation = -0.32
* **1. Autocorrelation**
* 00:00 - 00:06
* So far, you have looked at the correlation of two time series.
* **2. What is Autocorrelation?**
* 00:06 - 00:30
* Autocorrelation is the correlation of a single time series with a lagged copy of itself. It's also called "serial correlation". Often, when we refer to a series's autocorrelation, we mean the "lag-one" autocorrelation. So when using daily data, for example, the autocorrelation would be the correlation of the series with the same series lagged by one day.
* **3. Interpretation of Autocorrelation**
* 00:30 - 00:41
* What does it mean when a series has a positive or negative autocorrelation? With financial time series, when returns have a negative autocorrelation, we say it is "mean reverting".
* **4. Interpretation of Autocorrelation**
* 00:41 - 00:48
* Alternatively, if a series has positive autocorrelation, we say it is "trend-following".
* **5. Traders Use Autocorrelation to Make Money**
* 00:48 - 01:33
* Lest you think these concepts of autocorrelation are purely theoretical, they are actually used on Wall Street to make money. Many hedge fund strategies are only slightly more complex versions of mean reversion and momentum strategies. Since stocks have historically had negative autocorrelation over horizons of about a week, one popular strategy is to buy stocks that have dropped over the last week and sell stocks that have gone up. For other assets like commodities and currencies, they have historically had positive autocorrelation over horizons of several months, so the typical hedge fund strategy there is to buy commodities that have gone up in the last several months and sell those commodities that have gone down.
* **6. Example of Positive Autocorrelation: Exchange Rates**
* 01:33 - 02:37
* Here is an example of how you would compute the monthly autocorrelation for the Japanese Yen-US Dollar exchange rate. The data was downloaded from the FRED website, which stands for Federal Reserve Economic Data. The date column was read in as a string, so before you can compute autocorrelations, you will have to convert the index of dates from a string to a datetime object using the pandas method "to\_datetime". Now that it's a datetime object, downsample the data using the resample method. The "rule" argument indicates the desired frequency. 'M' stands for monthly. You can use any number of functions after resample. Here we used last for the last date of the period. But you could use first date of the period, or even an average over the period. Finally, compute the autocorrelation using the pandas method "autocorr". Notice in this example that the autocorrelation is positive, .0567, so this series exhibits some momentum.
* **7. Let's practice!**
* 02:37 - 02:47
* Now it's your turn. You'll look at a few financial time series that have negative autocorrelation.

# A Popular Strategy Using Autocorrelation

One puzzling anomaly with stocks is that investors tend to overreact to news. Following large jumps, either up or down, stock prices tend to reverse. This is described as mean reversion in stock prices: prices tend to bounce back, or revert, towards previous levels after large moves, which are observed over time horizons of about a week. A more mathematical way to describe mean reversion is to say that stock returns are negatively autocorrelated.

This simple idea is actually the basis for a popular hedge fund strategy. If you're curious to learn more about this hedge fund strategy (although it's not necessary reading for anything else later in the course), see [here](https://web.mit.edu/Alo/www/Papers/august07.pdf).

You'll look at the autocorrelation of weekly returns of MSFT stock from 2012 to 2017. You'll start with a DataFrame MSFT of daily prices. You should use the .resample() method to get weekly prices and then compute returns from prices. Use the pandas method .autocorr() to get the autocorrelation and show that the autocorrelation is negative. Note that the .autocorr() method only works on Series, not DataFrames (even DataFrames with one column), so you will have to select the column in the DataFrame.

## Instructions

100 XP

* Use the .resample() method with rule='W' followed by the function .last() to convert daily data to weekly data.
* Create a new DataFrame, returns, of percent changes in weekly prices using the .pct\_change() method.
* Compute the autocorrelation using the .autocorr() method on the series of closing stock prices, which is the column 'Adj Close' in the DataFrame returns.

**Notice how the autocorrelation of returns for MSFT is negative, so the stock is 'mean reverting'.**

**# Convert the daily data to weekly data**

**MSFT = MSFT.resample(rule='W').last()**

**# Compute the percentage change of prices**

**returns = MSFT.pct\_change()**

**# Compute and print the autocorrelation of returns**

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

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

**The autocorrelation of weekly returns is -0.16**

**<script.py> output:**

**The autocorrelation of weekly returns is -0.16**

# Are Interest Rates Autocorrelated?

When you look at daily changes in interest rates, the autocorrelation is close to zero. However, if you resample the data and look at annual changes, the autocorrelation is negative. This implies that while short term changes in interest rates may be uncorrelated, long term changes in interest rates are negatively autocorrelated. A daily move up or down in interest rates is unlikely to tell you anything about interest rates tomorrow, but a move in interest rates over a year can tell you something about where interest rates are going over the next year. And this makes some economic sense: over long horizons, when interest rates go up, the economy tends to slow down, which consequently causes interest rates to fall, and vice versa.

The DataFrame daily\_rates contains daily data of 10-year interest rates from 1962 to 2017.

## Instructions

100 XP

* Create a new DataFrame, daily\_diff, of changes in daily rates using the .diff() method.
* Compute the autocorrelation of the column 'US10Y' in daily\_diff using the .autocorr() method.
* Use the .resample() method with the argument rule='A' followed by the function .last() to convert to annual frequency.
* Create a new DataFrame, yearly\_diff of changes in annual rates and compute the autocorrelation, as above.

**# Compute the daily change in interest rates**

daily\_diff = daily\_rates.diff()

**# Compute and print the autocorrelation of daily changes**

autocorrelation\_daily = daily\_diff['US10Y'].autocorr()

print("The autocorrelation of daily interest rate changes is %4.2f" %(autocorrelation\_daily))

**# Convert the daily data to annual data**

yearly\_rates = daily\_rates.resample(rule='A').last()

**# Repeat above for annual data**

**yearly\_diff = yearly\_rates.diff()**

**autocorrelation\_yearly = yearly\_diff['US10Y'].autocorr()**

**print("The autocorrelation of annual interest rate changes is %4.2f" %(autocorrelation\_yearly))**

**The autocorrelation of daily interest rate changes is 0.07**

**The autocorrelation of annual interest rate changes is -0.22**

**<script.py> output:**

**The autocorrelation of daily interest rate changes is 0.07**

**The autocorrelation of annual interest rate changes is -0.22**

**Notice how the daily autocorrelation is small but the annual autocorrelation is large and negative.**

**1. Autocorrelation Function**

00:00 - 00:05

The sample autocorrelation function, or ACF,

**2. Autocorrelation Function**

00:05 - 00:22

shows not only the lag-one autocorrelation from the last chapter, but the entire autocorrelation function for different lags. Any significant non-zero autocorrelations implies that the series can be forecast from the past.

**3. ACF Example 1: Simple Autocorrelation Function**

00:22 - 00:34

This autocorrelation function implies that you can forecast the next value of the series from the last two values, since the lag-one and lag-two autocorrelations differ from zero.

**4. ACF Example 2: Seasonal Earnings**

00:34 - 00:59

Consider the time series of quarterly earnings of the company H&R Block. As we mentioned last chapter, a vast majority of their earnings occurs in the quarter that taxes are due. In this case, you can see clearly a seasonal pattern in the quarterly data on the left, and the autocorrelation function on the right shows strong autocorrelation at lags 4, 8, 12, 16, and 20

**5. ACF Example 3: Useful for Model Selection**

00:59 - 01:13

The ACF can also be useful for selecting a parsimonious model for fitting the data. In this example, the pattern of the autocorrelation suggests a model for the series that will be discussed in the next chapter.

**6. Plot ACF in Python**

01:13 - 01:35

plot\_acf is the statsmodels function for plotting the autocorrelation function. The input x is a series or array. The argument lags indicates how many lags of the autocorrelation function will be plotted. The alpha argument sets the width of the confidence interval, which is discussed on the next slide.

**7. Confidence Interval of ACF**

01:35 - 01:42

Here is an ACF plot that contains confidence intervals for each lag, which is the blue region in the figure.

**8. Confidence Interval of ACF**

01:42 - 02:31

In plot\_acf, the argument alpha determines the width of the confidence intervals. For example, if alpha equals 0-point-05, that means that if the true autocorrelation at that lag is zero, there is only a 5% chance the sample autocorrelation will fall outside that window. You will get a wider confidence interval if you set alpha lower, or if you have fewer observations. An approximation to the width of the 95% confidence intervals, if you make some simplifying assumptions, is plus or minus 2 over the square root of the number of observations in your series. Incidentally, if you don't want to see confidence intervals in your plot, set alpha equal to one.

**9. ACF Values Instead of Plot**

02:31 - 02:40

Besides plotting the ACF, you can also extract its numerical values using a similar Python function, acf, instead of plot\_acf.

**10. Let's practice!**

02:40 - 02:45

Now it's your turn.

# Taxing Exercise: Compute the ACF

In the last chapter, you computed autocorrelations with one lag. Often we are interested in seeing the autocorrelation over many lags. The quarterly earnings for H&R Block (ticker symbol HRB) is plotted on the right, and you can see the extreme cyclicality of its earnings. A vast majority of its earnings occurs in the quarter that taxes are due.

You will compute the array of autocorrelations for the H&R Block quarterly earnings that is pre-loaded in the DataFrame HRB. Then, plot the autocorrelation function using the plot\_acf module. This plot shows what the autocorrelation function looks like for cyclical earnings data. The ACF at lag=0 is always one, of course. In the next exercise, you will learn about the confidence interval for the ACF, but for now, suppress the confidence interval by setting alpha=1.

## Instructions

100 XP

* Import the acf module and plot\_acf module from statsmodels.
* Compute the array of autocorrelations of the quarterly earnings data in DataFrame HRB.
* Plot the autocorrelation function of the quarterly earnings data in HRB, and pass the argument alpha=1 to suppress the confidence interval.

**# Import the acf module and the plot\_acf module from statsmodelsfrom statsmodels.tsa.stattools import acf**

**from statsmodels.graphics.tsaplots import plot\_acf**

**# Compute the acf array of HRB**

**acf\_array = acf(HRB)**

**print(acf\_array)**

**# Plot the acf function**

**plot\_acf(HRB, alpha=1)**

**plt.show()**

**[ 1. -0.22122696 -0.39856504 -0.26615093 0.83479804 -0.1901038**

**-0.3475634 -0.23140368 0.71995993 -0.15661007 -0.29766783 -0.22097189**

**0.61656933 -0.15022869 -0.27922022 -0.22465946 0.5725259 ]**

**<script.py> output:**

**[ 1. -0.22122696 -0.39856504 -0.26615093 0.83479804 -0.1901038**

**-0.3475634 -0.23140368 0.71995993 -0.15661007 -0.29766783 -0.22097189**

**0.61656933 -0.15022869 -0.27922022 -0.22465946 0.5725259 ]**

**Notice the strong positive autocorrelation at lags 4, 8, 12, 16, 20, ...**

# Are We Confident This Stock is Mean Reverting?

In the last chapter, you saw that the autocorrelation 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

where is the number of observations, so if , 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

. 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.

## Instructions

100 XP

* Recompute the autocorrelation of weekly returns in the Series 'Adj Close' in the returns DataFrame.
* Find the number of observations in the returns DataFrame using the len() function.
* Approximate the 95% confidence interval of the estimated autocorrelation. The math function sqrt() has been imported and can be used.
* Plot the autocorrelation function of returns using plot\_acf that was imported from statsmodels. Set alpha=0.05 for the confidence intervals (that's the default) and lags=20

**# 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/sqrt(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 +/- 0.12**

**<script.py> output:**

**The autocorrelation of weekly MSFT returns is -0.16**

**The approximate confidence interval is +/- 0.12**

**Notice that the autocorrelation with lag 1 is significantly negative, but none of the other lags are significantly different from zero.**

**1. White Noise**

00:00 - 00:07

Although people define white noise slightly differently, a general definition

**2. What is White Noise?**

00:07 - 00:29

is that it is a series with mean that is constant with time, a variance that is also constant with time, and zero autocorrelation at all lags. There are several special cases of White Noise. For example, if the data is white noise but also has a normal, or Gaussian, distribution, then it is called Gaussian White Noise.

**3. Simulating White Noise**

00:29 - 00:43

numpy random normal creates an array of normally distributed random numbers. The loc argument is the mean and the scale argument is the standard deviation. This is one way to generate a white noise series.

**4. What Does White Noise Look Like?**

00:43 - 00:48

And here is a plot of the white noise series.

**5. Autocorrelation of White Noise**

00:48 - 00:54

And all the autocorrelations of a white noise series are zero. The returns on the stock market are pretty close to a white noise process.

**6. Stock Market Returns: Close to White Noise**

00:54 - 01:07

Here is the autocorrelation function for the S&P500. Notice that there are pretty much no lags where the autocorrelation is significantly different from zero.

**7. Let's practice!**

01:07 - 01:11

Time to put this into practice.

# 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 modeled 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.

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

## Instructions

100 XP

* Generate 1000 random normal returns using np.random.normal() with mean 2% (0.02) and standard deviation 5% (0.05), where the argument for the mean is loc and the argument for the standard deviation is scale.
* Verify the mean and standard deviation of returns using np.mean() and np.std().
* Plot the time series.
* Plot the autocorrelation function using plot\_acf with lags=20

**# Import the plot\_acf module from statsmodels**

**from statsmodels.graphics.tsaplots import plot\_acf**

**# Simulate white 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 0.018 and the standard deviation is 0.050**

**<script.py> output:**

**The mean is 0.018 and the standard deviation is 0.050**

**Notice that for a white noise time series, all the autocorrelations are close to zero, so the past will not help you forecast the future.**

**1. Random Walk**

00:00 - 00:03

In a random walk,

**2. What is a Random Walk?**

00:03 - 00:12

today's price is equal to yesterday's price plus some noise. Here is a plot of a simulated random walk.

**3. What is a Random Walk?**

00:12 - 00:37

The change in price of a random walk is just White Noise. Incidentally, if prices are in logs, then the difference in log prices is one way to measure returns. The bottom line is that if stock \*prices\* follow a random walk, then stock \*returns\* are White Noise. You can't forecast a random walk. The best guess for tomorrow's price is simply today's price.

**4. What is a Random Walk?**

00:37 - 01:02

In a random walk with drift, prices on average drift by mu every period. And the change in price for a random walk with drift is still white noise but with a mean of mu. So if we now think of stock prices as a random walk with drift, then the returns are still white noise, but with an average return of mu instead of zero.

**5. Statistical Test for Random Walk**

01:02 - 01:29

To test whether a series like stock prices follows a random walk, you can regress current prices on lagged prices. If the slope coefficient, beta, is not significantly different from one, then we cannot reject the null hypothesis that the series is a random walk. However, if the slope coefficient is significantly less than one, then we can reject the null hypothesis that the series is a random walk.

**6. Statistical Test for Random Walk**

01:29 - 01:42

An identical way to do that test is to regress the difference in prices on the lagged price, and instead of testing whether the slope coefficient is 1, now we test whether it is zero.

**7. Statistical Test for Random Walk**

01:42 - 01:53

This is called the "Dickey-Fuller" test. If you add more lagged prices on the right hand side, then it's called the Augmented Dickey-Fuller test.

**8. ADF Test in Python**

01:53 - 02:01

statsmodels has a function, adfuller, for performing the Augmented Dickey-Fuller test.

**9. Example: Is the S&P500 a Random Walk?**

02:01 - 02:57

As an example, let's run the Augmented Dickey-Fuller test on a time series of S&P500 prices using the adfuller function. The results are stored in results. The main output we're interested in is the p-value of the test. If the p-value is less than 5%, we can reject the null hypothesis that the series is a random walk with 95% confidence. In this case, the p-value is much higher than point-05 - it's 0-point-78. Therefore, we cannot reject the null hypothesis that the S&P500 is a random walk. You can also print out the full output of the test, which gives other information, like the number of observations (1257), the test statistic (-point-917) and the critical values of the test statistic for various alphas - 1%, 10%, and 5%.

**10. Let's practice!**

02:57 - 03:01

Now it's your turn.

# Generate a Random Walk

Whereas stock returns are often modeled 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.

## Instructions

100 XP

* Generate 500 random normal "steps" with mean=0 and standard deviation=1 using np.random.normal(), where the argument for the mean is loc and the argument for the standard deviation is scale.
* Simulate stock prices P:
  + Cumulate the random steps using the numpy .cumsum() method
  + Add 100 to P to get a starting stock price of 100.
* Plot the simulated random walk

**The simulated price series you plotted should closely resemble a random walk.**

**# 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.

## Instructions

100 XP

* Generate 500 random normal multiplicative "steps" with mean 0.1% and standard deviation 1% using np.random.normal(), which are now returns, and add one for total return.
* Simulate stock prices P:
  + Cumulate the product of the steps using the numpy .cumprod() method.
  + Multiply the cumulative product of total returns by 100 to get a starting value of 100.
* Plot the simulated random walk with drift.

**# Generate 500 random steps**

**steps = np.random.normal(loc=0.001, scale=0.01, 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()**

**This simulated price series you plotted should closely resemble a random walk for a high flying stock.**

# 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.

## Instructions

100 XP

* Import the adfuller module from statsmodels.
* Run the Augmented Dickey-Fuller test on the series of closing stock prices, which is the column 'Adj Close' in the AMZN DataFrame.
* Print out the entire output, which includes the test statistic, the p-values, and the critical values for tests with 1%, 10%, and 5% levels.
* Print out just the p-value of the test (results[0] is the test statistic, and results[1] is the p-value).

**# 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]))**

**(4.025168525770738, 1.0, 33, 5054, {'1%': -3.4316445438146865, '5%': -2.862112049726916, '10%': -2.5670745025321304}, 30308.64216426981)**

**The p-value of the test on prices is: 1.0**

**<script.py> output:**

**(4.025168525770738, 1.0, 33, 5054, {'1%': -3.4316445438146865, '5%': -2.862112049726916, '10%': -2.5670745025321304}, 30308.64216426981)**

**The p-value of the test on prices is: 1.0**

**According to this test, we cannot reject the hypothesis that Amazon prices follow a random walk. In the next exercise, you'll look at Amazon returns.**

# 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.

## Instructions

100 XP

* Import the adfuller module from statsmodels.
* Create a new DataFrame of AMZN returns by taking the percent change of prices using the method .pct\_change().
* Eliminate the NaN in the first row of returns using the .dropna() method on the DataFrame.
* Run the Augmented Dickey-Fuller test on the 'Adj Close' column of AMZN\_ret, and print out the p-value in results[1].

**The p-value is extremely small, so we can easily reject the hypothesis that returns are a random walk at all levels of significance.**

**# 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: 2.565589808348563e-22**

**<script.py> output:**

**The p-value of the test on returns is: 2.565589808348563e-22**

**1. Stationarity**

00:00 - 00:04

There are different ways to define stationarity,

**2. What is Stationarity?**

00:04 - 00:38

but in its strictest sense, it means that the joint distribution of the observations do not depend on time. A less restrictive version of stationarity, and one that is easier to test, is weak stationarity, which just means that the mean, variance, and autocorrelations of the observations do not depend on time. In other words, for the autocorrelation, the correlation between X-t and X-(t-tau) is only a function of the lag tau, and not a function of time.

**3. Why Do We Care?**

00:38 - 01:03

If a process is not stationary, then it becomes difficult to model. Modeling involves estimating a set of parameters, and if a process is not stationary, and the parameters are different at each point in time, then there are too many parameters to estimate. You may end up having more parameters than actual data! So stationarity is necessary for a parsimonious model, one with a smaller set of parameters to estimate.

**4. Examples of Nonstationary Series**

01:03 - 01:20

A random walk is a common type of non-stationary series. The variance grows with time. For example, if stock prices are a random walk, then the uncertainty about prices tomorrow is much less than the uncertainty 10 years from now.

**5. Examples of Nonstationary Series**

01:20 - 01:33

Seasonal series are also non-stationary. Here is the dataset you saw earlier on the frequency of Google searches for the word 'diet'. The mean varies with the time of the year.

**6. Examples of Nonstationary Series**

01:33 - 01:44

Here is White Noise, which would ordinarily be a stationary process, but here the mean increases over time, which makes it non-stationary.

**7. Transforming Nonstationary Series Into Stationary Series**

01:44 - 02:08

Many non-stationary series can be made stationary through a simple transformation. A Random Walk is a non-stationary series, but if you take the first differences, the new series is White Noise, which is stationary. On the left are S&P500 prices, which is a non-stationary random walk, but if you compute first differences on the right, it becomes a stationary white noise process.

**8. Transforming Nonstationary Series Into Stationary Series**

02:08 - 02:25

On the left, we have the quarterly earnings for H&R Block, which has a large seasonal component and is therefore not stationary. If we take the seasonal difference, by taking the difference with lag of 4, the transformed series looks stationary.

**9. Transforming Nonstationary Series Into Stationary Series**

02:25 - 02:51

Sometimes, you may need to make two transformations. Here is a time series of Amazon's quarterly revenue. It is growing exponentially as well as exhibiting a strong seasonal pattern. First, if you take only the log of the series, in the upper right, you eliminate the exponential growth. But if you take both the log of the series and then the seasonal difference, in the lower right, the transformed series looks stationary.

**10. Let's practice!**

02:51 - 02:56

Now let's try some examples.

# Is it Stationary?

Here are four time series plots:

A picture containing text, line, plot, diagram

Description automatically generated

Which one is stationary?

##### Answer the question

* A
* B
* C
* D

**Well done! This is white noise, which is stationary.**

# Seasonal Adjustment During Tax Season

Many time series exhibit strong seasonal behavior. The procedure for removing the seasonal component of a time series is called seasonal adjustment. For example, most economic data published by the government is seasonally adjusted.

You saw earlier that by taking first differences of a random walk, you get a stationary white noise process. For seasonal adjustments, instead of taking first differences, you will take differences with a lag corresponding to the periodicity.

Look again at the ACF of H&R Block's quarterly earnings, pre-loaded in the DataFrame HRB, and there is a clear seasonal component. The autocorrelation is high for lags 4,8,12,16,… because of the spike in earnings every four quarters during tax season. Apply a seasonal adjustment by taking the fourth difference (four represents the periodicity of the series). Then compute the autocorrelation of the transformed series.

## Instructions

100 XP

* Create a new DataFrame of seasonally adjusted earnings by taking the lag-4 difference of quarterly earnings using the .diff() method.
* Examine the first 10 rows of the seasonally adjusted DataFrame and notice that the first four rows are NaN.
* Drop the NaN rows using the .dropna() method.
* Plot the autocorrelation function of the seasonally adjusted DataFrame.

###### Incorrect submission

Check your call of HRB.diff(). Did you specify the first argument?

**By seasonally adjusting the series, we eliminated the seasonal pattern in the autocorrelation function.**

**# Import the plot\_acf module from statsmodels**

**from statsmodels.graphics.tsaplots import plot\_acf**

**# Seasonally adjust quarterly earnings**

**HRBsa = HRB.diff(4)**

**# Print the first 10 rows of the seasonally adjusted series**

**print(HRBsa.head(10))**

**# Drop the NaN data in the first four rows**

**HRBsa = HRBsa.dropna()**

**# Plot the autocorrelation function of the seasonally adjusted series**

**plot\_acf(HRBsa)**

**plt.show()**

**Earnings**

**Quarter**

**2007Q1 NaN**

**2007Q2 NaN**

**2007Q3 NaN**

**2007Q4 NaN**

**2008Q1 0.02**

**2008Q2 -0.04**

**2008Q3 -0.05**

**2008Q4 0.26**

**2009Q1 -0.05**

**2009Q2 0.02**

**1. Introducing an AR Model**

00:00 - 00:03

In an Autoregressive model,

**2. Mathematical Description of AR(1) Model**

00:03 - 00:37

or AR model, today's value equals a mean plus a fraction phi of yesterday's value, plus noise. Since there is only one lagged value on the right hand side, this is called an AR model of order 1, or simply an AR(1) model. If the AR parameter, phi, is one, then the process is a random walk. If phi is zero, then the process is white noise. In order for the process to be stable and stationary, phi has to be between -1 and +1.

**3. Interpretation of AR(1) Parameter**

00:37 - 01:08

As an example, suppose R\_t is a time series of stock returns. If phi is negative, then a positive return last period, at time t-1, implies that this period's return is more likely to be negative. This was referred to as "mean reversion" in Chapter 1. On the other hand, if phi is positive, then a positive return last period implies that this period's return is expected to be positive. This was referred to as "momentum" in Chapter 1.

**4. Comparison of AR(1) Time Series**

01:08 - 01:31

Here are four simulated time series with different AR parameters. When phi equals 0-point-9, it looks close to a random walk. When phi equals minus 0-point-9, the process looks more erratic - a large positive value is usually followed by a large negative one. The bottom two are similar, but are less exaggerated and closer to white noise.

**5. Comparison of AR(1) Autocorrelation Functions**

01:31 - 02:01

Here are four autocorrelation functions for different AR parameters. The autocorrelation decays exponentially at a rate of phi. Therefore if phi is 0-point-9, the lag-1 autocorrelation is 0-point-9, the lag-2 autocorrelation is 0-point-9 squared, the lag-3 autocorrelation is 0-point-9 cubed, etc. When phi is negative, the autocorrelation function still decays exponentially, but the signs of the autocorrelation function reverse at each lag.

**6. Higher Order AR Models**

02:01 - 02:14

So far, we've been only looking at AR(1) models. The model can be extended to include more lagged values and more phi parameters. Here we show an AR(1), an AR(2), and an AR(3).

**7. Simulating an AR Process**

02:14 - 03:17

Often, if you want to study and understand a pure AR process, it is useful to work with simulated data. Statsmodels provides modules for simulating AR processes. First, import the class, ArmaProcess. Then define the order and parameters of the AR process. The convention is a little counterintuitive: You must include the zero-lag coefficient of 1, and the sign of the other coefficient is the opposite of what we have been using. For example, for an AR(1) process with phi equal to plus 0-point-9, the second element of the ar array should be the opposite sign, \*minus 0-point-9\*. This is consistent with the time series literature in the field of signal processing. You also have to input the MA parameters. You will learn about MA models in the next chapter, so for now, just ignore the MA part. Then, you create an instance of the class ArmaProcess. To simulate data, use the method generate\_sample, with the number of simulated samples as an argument.

**8. Let's practice!**

03:17 - 03:22

Now let's try some examples.

# Simulate AR(1) Time Series

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

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

, 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

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

## Instructions

100 XP

* Import the class ArmaProcess in the arima\_process module.
* Plot the simulated AR processes:
  + Let ar1 represent an array of the AR parameters [1,
  + ] as explained above. For now, the MA parameter array, ma1, will contain just the lag-zero coefficient of one.
  + With parameters ar1 and ma1, create an instance of the class ArmaProcess(ar,ma) called AR\_object1.
  + Simulate 1000 data points from the object you just created, AR\_object1, using the method .generate\_sample(). Plot the simulated data in a subplot.
* Repeat for the other AR parameter.

**The two AR parameters produce very different looking time series plots, but in the next exercise you'll really be able to distinguish the time series.**

**# 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,

, the first-lag autocorrelation will be 0.9, the second-lag will be , the third-lag will be , 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 , the third-lag will be

, 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 parameter of -0.9, and simulated\_data\_3 is for an AR parameter of 0.3

## Instructions

100 XP

* Compute the autocorrelation function for each of the three simulated datasets using the plot\_acf function with 20 lags (and suppress the confidence intervals by setting alpha=1).

**Well Done! The ACF plots match what we predicted.**

**# Import the plot\_acf module from statsmodels**

**from statsmodels.graphics.tsaplots import plot\_acf**

**# Plot 1: AR parameter = +0.9**

**plot\_acf(simulated\_data\_1, alpha=1, lags=20)**

**plt.show()**

**# Plot 2: AR parameter = -0.9**

**plot\_acf(simulated\_data\_2, alpha=1, lags=20)**

**plt.show()**

**# Plot 3: AR parameter = +0.3**

**plot\_acf(simulated\_data\_3, alpha=1, lags=20)**

**plt.show()**

# Match AR Model with ACF

Here are four Autocorrelation plots:

A picture containing text, diagram, line, plot

Description automatically generated

Which figure corresponds to an AR(1) model with an AR parameter of -0.5?

##### Answer the question

**50XP**

#### Possible Answers

* A
* B
* C
* D
* **1. Estimating and Forecasting an AR Model**
* 00:00 - 00:07
* Statsmodels has another module for estimating the parameters of a given AR model.
* **2. Estimating an AR Model**
* 00:07 - 01:06
* The statsmodels class ARMA has been deprecated and replaced with the slightly more general ARIMA class. After importing ARIMA, create an instance of that class called mod, with the arguments being the data that you're trying to fit, and the order of the model. The order (1,0,0) means you're fitting the data to an AR(1) model. An order (2,0,0) would mean you're fitting the data to an AR(2) model. The middle number, d, relates to whether you take first differences of the data to make the time series stationary, like you would do with a random walk. For now, we'll assume there are no first differences so the middle number will be zero. In the last chapter, we will give an example where we do take first differences. The third number, q, is the MA part, which will be discussed in the next chapter. Once you instantiate the class, you can use the method fit to estimate the model, and store the results in result.
* **3. Estimating an AR Model**
* 01:06 - 01:26
* To see the full output, use the summary method on result. The coefficients for the mean mu and AR(1) parameter phi are highlighted in red. In the simulated data, mu was zero and phi was 0-point-9, and you can see that the estimated parameters are very close to the true parameters.
* **4. Estimating an AR Model**
* 01:26 - 01:39
* If you just want to see the coefficients rather than the entire regression output, you can use the params property, which returns an array of the fitted coefficients, mu and phi in this case.
* **5. Forecasting With an AR Model**
* 01:39 - 02:26
* To plot forecasts, both in-sample and out-of-sample, you have to import a new statsmodels function called plot\_predict. The first argument in the plot\_predict function is the result from the fitted model that we just talked about. You also give plot\_predict the starting and ending data points for forecasting. If the index of the data is a DatetimeIndex object as it is here, you can pick dates for the start and end date. The alpha argument sets the darkness of the shaded confidence interval region, and if you don't want a confidence interval at all, set alpha equal to None. You also need to set ax equal to ax so that the data and the predictions are on the same axes. In this plot, notice how the confidence interval gets wider the farther out the forecast is.
* **6. Let's practice!**
* 02:26 - 02:33
* Time to put this into practice.

# Estimating an AR Model

You will estimate the AR(1) parameter,

, 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

of 0.9, you will print out the estimate of

. 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.

## Instructions

100 XP

* Import the class ARIMA in the module statsmodels.tsa.arima.model.
* Create an instance of the ARIMA class called mod using the simulated data simulated\_data\_1 and the order (p,d,q) of the model (in this case, for an AR(1)), is order=(1,0,0).
* Fit the model mod using the method .fit() and save it in a results object called res.
* Print out the entire summary of results using the .summary() method.
* Just print out an estimate of

using the .params[1] attribute (no parentheses).

**Notice how close the estimated parameter is to the true parameter.**

**# Import the ARIMA module from statsmodels**

**from statsmodels.tsa.arima.model import ARIMA**

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

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

**res = mod.fit()**

**# Print out summary information on the fit**

**print(res.summary)**

**# Print out the estimate for phi**

**print("When the true phi=0.9, the estimate of phi is:")**

**print(res.params[1])**

**<bound method SARIMAXResults.summary of <statsmodels.tsa.arima.model.ARIMAResults object at 0x7ffab5a1bc40>>**

**When the true phi=0.9, the estimate of phi is:**

**0.9011059395968692**

**<script.py> output:**

**<bound method SARIMAXResults.summary of <statsmodels.tsa.arima.model.ARIMAResults object at 0x7ffab86f0250>>**

**When the true phi=0.9, the estimate of phi is:**

**0.9011059395968692**

**<script.py> output:**

**SARIMAX Results**

**==============================================================================**

**Dep. Variable: y No. Observations: 1000**

**Model: ARIMA(1, 0, 0) Log Likelihood -1420.051**

**Date: Sun, 14 May 2023 AIC 2846.103**

**Time: 09:11:25 BIC 2860.826**

**Sample: 0 HQIC 2851.699**

**- 1000**

**Covariance Type: opg**

**==============================================================================**

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

**------------------------------------------------------------------------------**

**const -0.3986 0.320 -1.247 0.212 -1.025 0.228**

**ar.L1 0.9011 0.013 67.552 0.000 0.875 0.927**

**sigma2 1.0005 0.045 22.239 0.000 0.912 1.089**

**===================================================================================**

**Ljung-Box (L1) (Q): 0.58 Jarque-Bera (JB): 0.17**

**Prob(Q): 0.45 Prob(JB): 0.92**

**Heteroskedasticity (H): 1.00 Skew: -0.03**

**Prob(H) (two-sided): 0.99 Kurtosis: 2.98**

**===================================================================================**

**Warnings:**

**[1] Covariance matrix calculated using the outer product of gradients (complex-step).**

**When the true phi=0.9, the estimate of phi is:**

**0.9011059395968692**

# 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. You can plot the forecasted data using the function plot\_predict(). 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 data in DataFrame simulated\_data\_1, with

, you will plot out-of-sample forecasts and confidence intervals around those forecasts.

## Instructions

100 XP

* Import the class ARIMA and also import the function plot\_predict
* Create an instance of the ARIMA class called mod using the simulated data in DataFrame simulated\_data\_1 and the order (p,d,q) of the model (in this case, for an AR(1)), order=(1,0,0)
* Fit the model mod using the method .fit() and save it in a results object called res
* Plot the in-sample data starting with data point 950
* Plot out-of-sample forecasts of the data and confidence intervals using the plot\_predict() function, starting where the data ends at the 1000th point, and ending the forecast at point 1010

**Notice how, when phi is high like here, the forecast gradually moves to the long term mean of zero, but if phi were low, it would move much quicker to the long term mean. Try it out and see for yourself!**

**# Import the ARIMA and plot\_predict**

**from statsmodels from statsmodels.tsa.arima.model import ARIMA**

**from statsmodels.graphics.tsaplots import plot\_predict**

**# Forecast the first AR(1) model**

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

**res = mod.fit()**

**# Plot the data and the forecast**

**fig, ax = plt.subplots()**

**simulated\_data\_1.loc[950:].plot(ax=ax)**

**plot\_predict(res, start=1000, end=1010, ax=ax)**

**plt.show()**

Daily XP650

## Exercise

## Exercise

# Let's Forecast Interest Rates

You will now use the forecasting techniques you learned in the last exercise and apply it to real data rather than simulated data. You will revisit a dataset from the first chapter: the annual data of 10-year interest rates going back 56 years, which is in a Series called interest\_rate\_data. Being able to forecast interest rates is of enormous importance, not only for bond investors but also for individuals like new homeowners who must decide between fixed and floating rate mortgages.

You saw in the first chapter that there is some mean reversion in interest rates over long horizons. In other words, when interest rates are high, they tend to drop and when they are low, they tend to rise over time. Currently they are below long-term rates, so they are expected to rise, but an AR model attempts to quantify how much they are expected to rise.

The class ARIMA and the function plot\_predict have already been imported.

## Instructions

100 XP

* Create an instance of the ARIMA class called mod using the annual interest rate data and choosing the order for an AR(1) model.
* Fit the model mod using the method .fit() and save it in a results object called res.
* Plot the data and the in-sample and out-of-sample forecasts of the data using the .plot\_predict() function.
  + The first argument of plot\_predict() should be the fitted model.
  + Pass the arguments start=0 to start the in-sample forecast from the beginning, and choose end to be '2027' to forecast several years in the future.
  + Note that the end argument 2027 must be in quotes here since it represents a date and not an integer position.

**# Forecast interst rates using an AR(1) model**

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

**res = mod.fit()**

**# Plot the data and the forecast**

**fig, ax = plt.subplots()**

**interest\_rate\_data.plot(ax=ax)**

**plot\_predict(res, start=0, end='2027', alpha=None, ax=ax)**

**plt.show()**

**According to an AR(1) model, 10-year interest rates are forecasted to rise from 2.16%, towards the end of 2017 to 3.35% in five years.**

# 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.

## Instructions

100 XP

* Import plot\_acf function from the statsmodels module
* Create two axes for the two subplots
* Plot the autocorrelation function for 12 lags of the interest rate series interest\_rate\_data in the top plot
* Plot the autocorrelation function for 12 lags of the interest rate series simulated\_data in the bottom plot

**# 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()**

**Notice the Autocorrelation functions look very similar for the two series.**

**1. Choosing the Right Model**

00:00 - 00:07

In practice, you will ordinarily not be told the order of the model that you're trying to estimate.

**2. Identifying the Order of an AR Model**

00:07 - 00:18

There are two techniques that can help determine the order of the AR model: The Partial Autocorrelation Function, and the Information Criteria

**3. Partial Autocorrelation Function (PACF)**

00:18 - 00:53

The Partial Autocorrelation Function measures the incremental benefit of adding another lag. Imagine running several regressions, where you regress returns on more and more lagged values. The coefficients in the red boxes represent the values of the partial autocorrelation function for different lags. For example, in the bottom row, the coefficient in the red box, phi 4-4, is the lag-4 value of the Partial Autocorrelation Function, and it represents how significant adding a fourth lag is when you already have three lags.

**4. Plot PACF in Python**

00:53 - 01:20

plot\_pacf is the statsmodels function for plotting the partial autocorrelation function. The arguments are the same as that of the plot\_acf module that you saw earlier. The input x is a series or array. The argument lags indicates how many lags of the partial autocorrelation function will be plotted. And the alpha argument sets the width of the confidence interval.

**5. Comparison of PACF for Different AR Models**

01:20 - 01:48

These plots show the Partial Autocorrelation Function for AR models of different orders. In the upper left, for an AR(1) model, only the lag-\*1\* PACF is significantly different from zero. Similarly, for an AR(2) model, two lags are different from zero, and for and AR(3), three lags are different from zero. Finally, for White Noise, there are no lags that are significantly different from zero.

**6. Information Criteria**

01:48 - 02:12

The more parameters in a model, the better the model will fit the data. But this can lead to overfitting of the data. The information criteria adjusts the goodness-of-fit of a model by imposing a penalty based on the number of parameters used. Two common adjusted goodness-of-fit measures are called the Akaike Information Criterion and the Bayesian Information Criterion.

**7. Information Criteria**

02:12 - 02:23

This is the full output from estimating an AR(2) model. The AIC and BIC are highlighted in the red box. To get the AIC and BIC statistics,

**8. Getting Information Criteria From statsmodels**

02:23 - 02:52

you follow the same procedure from the last section to fit the data to a model. In the last section, you learned how to get the full output using summary or just the AR parameters using the params attribute. You can also get the AIC or BIC using those attributes. In practice, the way to use the Bayesian information criterion is to fit several models, each with a different number of parameters, and choose the one with the lowest information criterion.

**9. Information Criteria**

02:52 - 03:11

Suppose we are given a time series of data, and unknown to us, it was simulated from an AR(3) model. Here is a plot of the BIC when we fit the data to an AR(1) up to an AR(6) model. You can see that the lowest BIC occurs for an AR(3).

**10. Let's practice!**

03:11 - 03:17

Now it's your turn.

Daily XP900

## Exercise

## Exercise

# 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.

## Instructions

100 XP

* Import the modules for simulating data and for plotting the PACF
* Simulate an AR(1) with

 (remember that the sign for the AR parameter is reversed)

 Plot the PACF for simulated\_data\_1 using the plot\_pacf function

 Simulate an AR(2) with

 (again, reverse the signs)

 Plot the PACF for simulated\_data\_2 using the plot\_pacf function

**Notice that the number of significant lags for the PACF indicate the order of the AR model.**

**# 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, -0.6, -0.3])**

**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()**

Daily XP1000

## Exercise

## Exercise

# 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.

## Instructions

100 XP

* Import the ARIMA module for estimating the parameters and computing BIC.
* Initialize a numpy array BIC, which we will use to store the BIC for each AR(p) model.
* Loop through order p for p = 0,…,6.
  + For each p, fit the data to an AR model of order p.
  + For each p, save the value of BIC using the .bic attribute (no parentheses) of res.
* Plot BIC as a function of p (for the plot, skip p=0 and plot for p=1,…6).
* **For an AR(2), the BIC achieves its minimum at p=2, which is what we expect.**

**# Import the module for estimating an ARIMA model**

**from statsmodels.tsa.arima.model import ARIMA**

**# 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 = ARIMA(simulated\_data\_2, order=(p,0,0))**

**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('Bayesian Information Criterion')**

**plt.show()**

**Moving Average(MA) and ARMA Models**

**1. Describe Model**

00:00 - 00:04

In a Moving Average, or MA model,

**2. Mathematical Description of MA(1) Model**

00:04 - 00:29

today's value equals a mean plus noise, plus a fraction theta of yesterday's noise. Since there is only one lagged error on the right hand side, this is called an MA model of order 1, or simply an MA(1) model. If the MA parameter, theta, is zero, then the process is white noise. MA models are stationary for all values of theta.

**3. Interpretation of MA(1) Parameter**

00:29 - 01:02

Suppose R t is a time series of stock returns. If theta is negative, then a positive shock last period, represented by epsilon t-1, would have caused last period's return to be positive, but this period's return is more likely to be negative. A shock two periods ago would have no effect on today's return - only the shock now and last period. Also, note that the lag-1 autocorrelation turns out not to be theta, but theta over 1 plus theta squared.

**4. Comparison of MA(1) Autocorrelation Functions**

01:02 - 01:21

Here are four autocorrelation functions for different MA parameters. In each case, there is zero autocorrelation for an MA(1) beyond lag-1. When theta is positive, the lag-1 autocorrelation is positive and when theta is negative, the lag-1 autocorrelation is negative.

**5. Example of MA(1) Process: Intraday Stock Returns**

01:21 - 01:49

Higher frequency stock returns are a nice example of an MA(1) process. Here is an intraday plot for the stock price of Sprint Corporation for one day. The frequency of the data is one minute. Stocks trade at discrete one-cent increments rather than at continuous prices, and you can see that the stock can bounce back and forth over a one-cent range for long periods of time. This is sometimes referred to as the "bid/ask bounce".

**6. Autocorrelation Function of Intraday Stock Returns**

01:49 - 02:10

The bid/ask bounce induces a negative lag-1 autocorrelation, but no autocorrelation beyond lag-1. You can see this with the autocorrelation function plot of intraday returns. The lag-1 autocorrelation is significantly negative, and the other autocorrelations are not significantly different from zero.

**7. Higher Order MA Models**

02:10 - 02:23

So far, we've been only looking at MA(1) models. The model can be extended to include more lagged errors and more theta parameters. Here we show an MA(1), an MA(2), and an MA(3) model.

**8. Simulating an MA Process**

02:23 - 03:01

Just like in the last chapter with AR models, you may want to simulate a pure MA process. You can use the same statsmodels module, ArmaProcess. This time, for an MA(1), the AR order is just an array containing 1. The MA order is an array containing 1 and the MA(1) parameter theta. Unlike with the AR simulation, you don't need to reverse the sign of theta. As before, you create an instance of the class ArmaProcess. To simulate data, use the method generate\_sample, with the number of simulated samples as an argument.

**9. Let's practice!**

03:01 - 03:05

Now let's try some examples.

## Exercise

# Simulate MA(1) Time Series

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

, 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 and a large negative

.

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

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

## Instructions

100 XP

* Import the class ArmaProcess in the arima\_process module.
* Plot the simulated MA(1) processes
  + Let ma1 represent an array of the MA parameters [1,
*  ] as explained above. The AR parameter array will contain just the lag-zero coefficient of one.
* With parameters ar1 and ma1, create an instance of the class ArmaProcess(ar,ma) called MA\_object1.
* Simulate 1000 data points from the object you just created, MA\_object1, using the method .generate\_sample(). Plot the simulated data in a subplot.

 Repeat for the other MA parameter.

**The two MA parameters produce different time series plots, but in the next exercise you'll really be able to distinguish the time series.**

**# 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, 0.9])**

**MA\_object2 = ArmaProcess(ar2, ma2)**

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

**plt.plot(simulated\_data\_2)**

**plt.show()**

## Exercise

## Exercise

# 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

, but rather . For example, if the MA parameter, , is = +0.9, the first-lag autocorrelation will be , and the autocorrelation at all other lags will be zero. If the MA parameter, , is -0.9, the first-lag autocorrelation will be

.

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

## Instructions 1/3

simulated\_data\_1 is the first simulated time series with an MA parameter of

Compute the autocorrelation function of simulated\_data\_1 using the plot\_acf function with 20 lags.

**# Import the plot\_acf module from statsmodels**

**from statsmodels.graphics.tsaplots import plot\_acf**

**# Plot 1: MA parameter = -0.9**

**plot\_acf(simulated\_data\_1, lags=20)**

**plt.show()**

simulated\_data\_2 is the second simulated time series with an MA parameter of

Compute the autocorrelation function using the plot\_acf function with lags=20.

**# Plot 2: MA parameter = 0.9**

**plot\_acf(simulated\_data\_2, lags=20)**

**plt.show()**

simulated\_data\_3 is the third simulated time series with an MA parameter of

Compute the autocorrelation function using the plot\_acf() function with 20 lags.

**# Plot 3: MA parameter = -0.3**

**plot\_acf(simulated\_data\_3, lags=20)**

**plt.show()**

**Well Done! The ACF plots match what we predicted.**

# Match ACF with MA Model

Here are four Autocorrelation plots:

A picture containing text, line, diagram, number

Description automatically generated

Which figure corresponds to an MA(1) model with an MA parameter of -0.5

##### Answer the question

**50XP**

#### Possible Answers

* D
* 
* 
* **1. Estimation and Forecasting an MA Model**
* 00:00 - 00:05
* The same module that you used to estimate the parameters of
* **2. Estimating an MA Model**
* 00:05 - 00:37
* The same module that you used to estimate the parameters of an AR model can be used to estimate the parameters of an MA model. Import the class ARIMA as before, and create an instance of that class called mod, with the arguments being the data that you're trying to fit, and the order of the model. However, now the order is (0,0,1), for an MA(1), not (1,0,0) as it was for an AR(1). And as before with an AR model, once you instantiate the class, you can then use the method fit to estimate the model, and store the results in result.
* **3. Forecasting an MA Model**
* 00:37 - 01:09
* The procedure for forecasting an MA model is the same as that for an AR model: you have to import the statsmodels function plot\_predict. The first argument in the plot\_predict function is the result from the fitted model. You also give plot\_predict the starting and ending data points for forecasting. And you set ax equal to ax so that the data and the predictions are on the same axes. One thing to note, is that with an MA(1) model, unlike an AR model, all forecasts beyond the one-step ahead forecast will be the same.
* **4. Let's practice!**
* 01:09 - 01:11
* Time to put this into practice.

# Estimating an MA Model

You will estimate the MA(1) parameter,

, 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

of -0.9, you will print out the estimate of

. 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.

## Instructions

100 XP

* Import the class ARIMA in the module statsmodels.tsa.arima.model.
* Create an instance of the ARIMA class called mod using the simulated data simulated\_data\_1 and the order (p,d,q) of the model (in this case, for an MA(1)), is order=(0,0,1).
* Fit the model mod using the method .fit() and save it in a results object called res.
* Print out the entire summary of results using the .summary() method.
* Just print out an estimate of the theta parameter using the .params[1] attribute.

**# Import the ARIMA module from statsmodels**

**from statsmodels.tsa.arima.model import ARIMA**

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

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

**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 is:")**

**print(res.params[1])**

**SARIMAX Results**

**==============================================================================**

**Dep. Variable: y No. Observations: 1000**

**Model: ARIMA(0, 0, 1) Log Likelihood -1420.500**

**Date: Mon, 15 May 2023 AIC 2846.999**

**Time: 22:15:28 BIC 2861.723**

**Sample: 0 HQIC 2852.595**

**- 1000**

**Covariance Type: opg**

**==============================================================================**

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

**------------------------------------------------------------------------------**

**const -0.0038 0.003 -1.162 0.245 -0.010 0.003**

**ma.L1 -0.8967 0.014 -65.162 0.000 -0.924 -0.870**

**sigma2 1.0015 0.045 22.223 0.000 0.913 1.090**

**===================================================================================**

**Ljung-Box (L1) (Q): 0.46 Jarque-Bera (JB): 0.14**

**Prob(Q): 0.50 Prob(JB): 0.93**

**Heteroskedasticity (H): 1.00 Skew: -0.03**

**Prob(H) (two-sided): 0.99 Kurtosis: 2.98**

**===================================================================================**

**Warnings:**

**[1] Covariance matrix calculated using the outer product of gradients (complex-step).**

**When the true theta=-0.9, the estimate of theta is:**

**-0.8967129858089047**

**Notice how close the estimated parameter is to the true parameter.**

Daily XP150

## Exercise

## Exercise

# 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 the plot\_predict() function in statsmodels.

For the simulated series simulated\_data\_1 with

, 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.

## Instructions

100 XP

* Import the class ARIMA and also import the function plot\_predict
* Create an instance of the ARIMA class called mod using the simulated data simulated\_data\_1 and the (p,d,q) order of the model (in this case, for an MA(1)), order=(0,0,1)
* Fit the model mod using the method .fit() and save it in a results object called res
* Plot the in-sample data starting with data point 950
* Plot out-of-sample forecasts of the data and confidence intervals using the plot\_predict() function, starting with data point 950 and ending the forecast at point 1010

**# Import the ARIMA and plot\_predict from statsmodels**

**from statsmodels.tsa.arima.model import ARIMA**

**from statsmodels.graphics.tsaplots import plot\_predict**

**# Forecast the first MA(1) model**

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

**res = mod.fit()**

**# Plot the data and the forecast**

**fig, ax = plt.subplots()**

**simulated\_data\_1.loc[950:].plot(ax=ax)**

**plot\_predict(res, start=1000, end=1010, ax=ax)**

**1. ARMA models**

00:00 - 00:06

An ARMA model is a combination of an AR and MA model.

**2. ARMA Model**

00:06 - 00:14

Here is the formula for an ARMA(1,1) model, which has the familiar AR(1) and MA(1) components.

**3. Converting Between ARMA, AR, and MA Models**

00:14 - 00:46

ARMA models can be converted to pure AR or pure MA models. Here is an example of converting an AR(1) model into and MA(infinity) model. The first line is an AR(1) model. Then, in the second line, the AR(1) equation is substituted for R t-1. You do the same for R t-2 in the third line, etc., and you eventually end up with an MA(infinity) model with coefficients phi, phi squared, phi cubed, etc.

**4. Let's practice!**

00:46 - 00:52

Now it's your turn.

**plt.show()**

**Notice that the out-of-sample forecasts are flat into the future after the first data point.**

Daily XP300

## Exercise

## Exercise

# High Frequency Stock Prices

Higher frequency stock data is well modeled 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.

## Instructions

100 XP

* Manually change the first date to zero using .iloc[0,0].
* Change the two column headers to 'DATE' and 'CLOSE' by setting intraday.columns equal to a list containing those two strings.
* Use the pandas attribute .dtypes (no parentheses) to see what type of data are in each column.
* Convert the 'DATE' column to numeric using the pandas function to\_numeric().
* Make the 'DATE' column the new index of intraday by using the pandas method .set\_index(), which will take the string 'DATE' as its argument (not the entire column, just the name of the column).
* **# 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')**
* **DATE int64**
* **CLOSE float64**
* **dtype: object**

**Good job getting the data in a format that we can work with.**

Daily XP400

## Exercise

## Exercise

# 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. After filling in the missing rows, you can 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".

## Instructions 1/4

Print out the length of intraday using len().

**# Notice that some rows are missing**

**print("If there were no missing rows, there would be 391 rows of minute data")**

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

**If there were no missing rows, there would be 391 rows of minute data**

**The actual length of the DataFrame is: 389**

Daily XP425

## Exercise

## Exercise

# 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. After filling in the missing rows, you can 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".

## Instructions 2/4

Find the missing rows by making range(391) into a set and then subtracting the set of the intraday index, intraday.index.

**# Notice that some rows are missing**

**print("If there were no missing rows, there would be 391 rows of minute data")**

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

**# Everything**

**set\_everything = set(range(391))**

**# The intraday index as a set**

**set\_intraday = set(intraday.index)**

**# Calculate the difference**

**set\_missing = set\_everything - set\_intraday**

**# Print the difference**

**print("Missing rows: ", set\_missing)**

**# Notice that some rows are missing**

**print("If there were no missing rows, there would be 391 rows of minute data")**

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

**If there were no missing rows, there would be 391 rows of minute data**

**The actual length of the DataFrame is: 389**

**Instructions**

Fill in the missing rows using the .reindex() method, setting the index equal to the full range(391) and forward filling the missing data by setting the method argument to 'ffill'.

# Fill in the missing rows

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

* Change the index to times using pandas function date\_range(), starting with '2017-09-01 9:30' and ending with '2017-09-01 16:00' and passing the argument freq='1min'.
* Plot the data and include gridlines.

**# From previous step**

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

**# Change the index to the intraday times**

**intraday.index = pd.date\_range(start='2017-09-01 9:30', end='2017-09-01 16:00', freq='1min') # Plot the intraday time series**

**intraday.plot(grid=True)**

**plt.show()**

**Missing data is common with high frequency financial time series, so good job fixing that.**

Daily XP500

## Exercise

## Exercise

# 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.

## Instructions

100 XP

* Import plot\_acf and ARIMA modules from statsmodels
* Compute minute-to-minute returns from prices:
  + Compute returns with the .pct\_change() method
  + Use the pandas method .dropna() to drop the first row of returns, which is NaN
* Plot the ACF function with lags up to 60 minutes
* Fit the returns data to an MA(1) model and print out the MA(1) parameter
* **# Import plot\_acf and ARIMA modules from statsmodels**
* **from statsmodels.graphics.tsaplots import plot\_acf**
* **from statsmodels.tsa.arima.model import ARIMA**
* **# 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 = ARIMA(returns, order=(0,0,1))**
* **res = mod.fit()**
* **print(res.params[1])**
* **-0.17171619394800597**
* **<script.py> output:**
* **-0.17171619394800597**

**Notice the significant negative lag-1 autocorrelation, just like for an MA(1) model.**

## Exercise

## Exercise

# 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

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

.

Note, to raise a number *x* to the power of an exponent *n*, use the format *x\*\*n*.

## Instructions

100 XP

* Import the modules for simulating data and plotting the ACF from statsmodels
* Use a list comprehension to build a list with **exponentially** decaying MA parameters:

 Simulate 5000 observations of the MA(30) model

 Plot the ACF of the simulated series

**'''**

**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,…**

**0.8**

**,**

**0.8**

**2**

**,**

**0.8**

**3**

**,**

**…**

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

**ϕ**

**=**

**0.8**

**.**

**INSTRUCTIONS**

**100XP**

**Import the modules for simulating data and plotting the ACF from statsmodels**

**Use a list comprehension to build a list with exponentially decaying MA parameters: 1,0.8,0.82,0.83,…**

**1**

**,**

**0.8**

**,**

**0.8**

**2**

**,**

**0.8**

**3**

**,**

**…**

**Simulate 5000 observations of the MA(30) model**

**Plot the ACF of the simulated series**

**'''**

**# 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()**

**# 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()**

**Notice that the ACF looks the same as an AR(1) with parameter 0.8.**