5QQMN534: Algorithmic Finance

Week8: Trading Strategies, Technical Analysis and Backtesting

Yves Hilpisch - Python for Finance 2nd Edition 2019: Chapter 15

Agenda Chapter15: Trading Strategies

- Introduction
  - Types of Strategy
  - Backtesting System Design Considerations
  - Types of Backtesting
  - Major Backtesting Considerations / Pitfalls
- Simple Moving Average (SMA)
  - Data Import
  - Trading Strategy
  - Vectorized Backtesting
  - Optimisation
- Random Walk Hypothesis
- Linear OLS Regression
  - The Data
  - Regression

Types of Strategies

Forecasting (Equity Valuation)

Predict the direction or value of an asset in future time periods based on certain historical factors / information

Mean Reversion (Pairs Trading)

Trades on spread deviating between two or more assets. Utilises cointegration tests to ascertain mean reverting behaviour.

Momentum / Trend following

Trades on the basis of information , direction, momentum (in direct contrast to Efficient Market Hypothesis). Including many technical analysis studies.

High Frequency Trading or HFT

Specifically referring to exploitation of sub-millisecond market microstructure. FPGAs, Infiniband networks (super computers (very low latency). Incredibly complex.

Backtesting System Design Considerations

Things to consider for system design include:

- Risk Management
- Front / Mid / Back Office Settlement
- Databases
- Signal Generation
- Execution Handler
- Optimisation
- Multi-Threading
- DATA Handler (input / output)
- Backtest to Live Transfer (code re-usability)
- Statistical Portfolio Performance Metrics

Backtesting and designing trading systems in C++, C#.NET, Matlab is much more difficult!

Python is "easier" to go from Backtesting to live production

Types of Backtesting

Research Vs Implementation

<u>Research (Identified Anomaly)</u>

- Prototyping Ideas "quickly"
- Testing / optimising "quickly"
- Statistical Analysis
- "Typically Unrealistic performance accuracy (see pitfalls)"

<u>Implementation (Our core focus)</u>

- Detailed development and testing
- Full Suite System Design
- Complex Event Processing (Example Text Translation News Signal Generation or Economic Data Signals)
- Re-usable code for Backtesting to live trading
- Model and pitfall considerations covered in detail = realistic performance.

Major Backtesting Considerations / Pitfalls

- **Market regime shift:** Regulatory change, macroeconomic events, “black swans”, markets change or shift regimes due to different market conditions (Markov models)
- **Transaction costs:** Unrealistic handling of slippage, market impact and fees. Imperative to model correctly. Slippage is the difference between price wanted from signal and price executed. Price moves rapidly in volatile conditions. Hard to model accuracy. However be conservative. Commissions and transaction costs will be disclosed by the broker so can model in effectively.

- **Liquidity constraints:** Ban on short selling (finance stocks in 2008 and Chinese stock market June 2015)

- **Optimisation Bias:** Over-fitting a model too closely to limited data. Common issue. An optimisation on parameters may work great in AAPL stock but poor in treasury bond futures. Robustness and cross market adaptiveness is hard to design strategies for. Market specific strategies / anomalies more common to exploit.

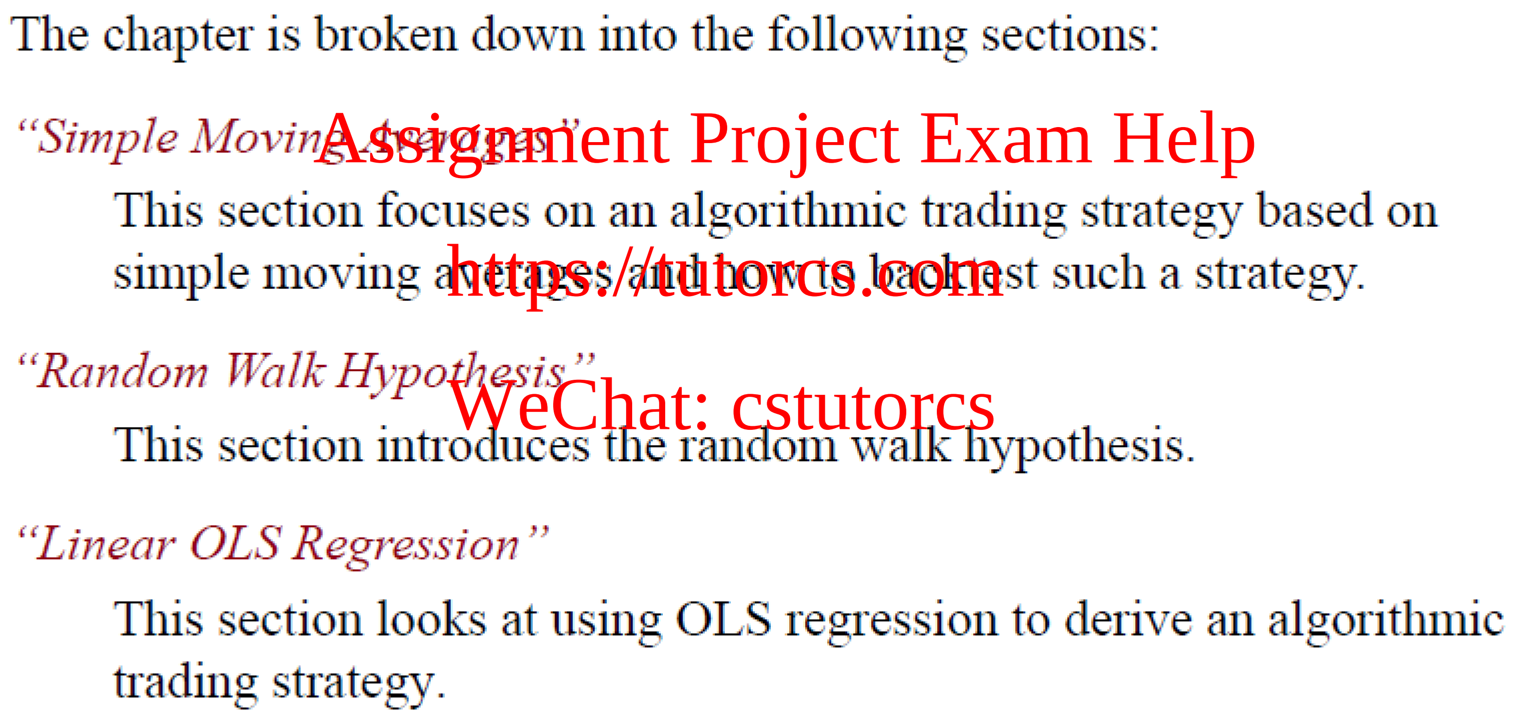
- **Survivorship Bias:** Only using instruments which still exist (incorrect sample). De-listed stocks included is an error. This is quite difficult to compensate for. Accuracy of data = very important.
- **Lookahead Bias:** Accidental introduction of future information into past data.
- **Interference:** Ignoring strategy rule, due to gut feeling, model is wrong. NEVER do this. If you have backtested a model, good results and go live trading and don’t have automated execution and only signal generation. If you don’t manual trade exactly as model says and miss a trade, that could be the one trade that makes your day / week / month. NEVER deviate from the rules ever. Manual intervention only in emergency situations (etc. exchange goes down). Called the kill switch!

Trading Strategies1

- This chapter is about the vectorized backtesting of algorithmic trading strategies.
- The term *algorithmic trading strategy* is used to describe any type of financial trading strategy that is based on an algorithm designed to take long, short, or neutral positions in financial instruments on its own without human interference.
- A simple algorithm, such as “altering every five minutes between a long and a neutral position in the stock of Apple, Inc.,” satisfies this definition.
- For the purposes of this chapter and a bit more technically, an algorithmic trading strategy is represented by some Python code that, given the availability of new data, decides whether to buy or sell a financial instrument in order to take long, short, or neutral positions in it.
- The chapter does not provide an overview of algorithmic trading strategies (see “Further Resources” for references that cover algorithmic trading strategies in more detail).
- It rather focuses on the technical aspects of the *vectorized backtesting* approach for a select few such strategies.
- With this approach the financial data on which the strategy is tested is manipulated in general as a whole, applying vectorized operations on `NumPy ndarray` and `pandas DataFrame` objects that store the financial data.**

** An alternative approach would be the *event-based backtesting* of trading strategies, during which the arrival of new data in markets is simulated by explicitly looping over every single new data point.

Trading Strategies2



The chapter is broken down into the following sections:

"Simple Moving Averages"

This section focuses on an algorithmic trading strategy based on simple moving averages and how to backtest such a strategy.

"Random Walk Hypothesis"

This section introduces the random walk hypothesis.

"Linear OLS Regression"

This section looks at using OLS regression to derive an algorithmic trading strategy.

Simple Moving Averages

- Trading based on simple moving averages (SMAs) is a decades-old trading approach (see, for example, the paper by Brock et al. (1992)). Although many traders use SMAs for their discretionary trading, they can also be used to formulate simple algorithmic trading strategies.
- This section uses SMAs to introduce vectorized backtesting of algorithmic trading strategies.
- It builds on the technical analysis example in Chapter 8.
- See AAPL SMA Example.xlsx for example file. **Sheet** AAPL SMA
- For the code outputs from code look at **sheet** Chapter15 Code Ex1 and Chapter15 Code Ex2

Data Import

First, some imports:

```
In [1]: import numpy as np
        import pandas as pd
        import datetime as dt
        from pylab import mpl, plt

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

- Second, the reading of the raw data and the selection of the financial time series for a single symbol, the stock of Apple, Inc. (`AAPL.O`).
- The analysis in this section is based on end-of-day data; intraday data is used in subsequent sections:

```
In [3]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                          index_col=0, parse_dates=True)

In [4]: raw.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 2216 entries, 2010-01-01 to 2018-06-29
        Data columns (total 12 columns):
        AAPL.O    2138 non-null float64
        MSFT.O    2138 non-null float64
        INTC.O    2138 non-null float64
        AMZN.O    2138 non-null float64
        GS.N      2138 non-null float64
        SPY       2138 non-null float64
        .SPX      2138 non-null float64
        .VIX      2138 non-null float64
        EUR=      2216 non-null float64
        XAU=      2211 non-null float64
        GDX       2138 non-null float64
        GLD       2138 non-null float64
        dtypes: float64(12)
        memory usage: 225.1 KB

In [5]: symbol = 'AAPL.O'

In [6]: data = (
            pd.DataFrame(raw[symbol])
            .dropna()
        )
```

Trading Strategy1

- Third, the calculation of the SMA values for two different rolling window sizes. Figure 15-1 shows the three time series visually:

```
In [7]: SMA1 = 42  ❶
        SMA2 = 252  ❷

In [8]: data['SMA1'] = data[symbol].rolling(SMA1).mean()  ❶
        data['SMA2'] = data[symbol].rolling(SMA2).mean()  ❷

In [9]: data.plot(figsize=(10, 6));
```

❶ Calculates the values for the *shorter* SMA.

❷ Calculates the values for the *longer* SMA.

Trading Strategy2

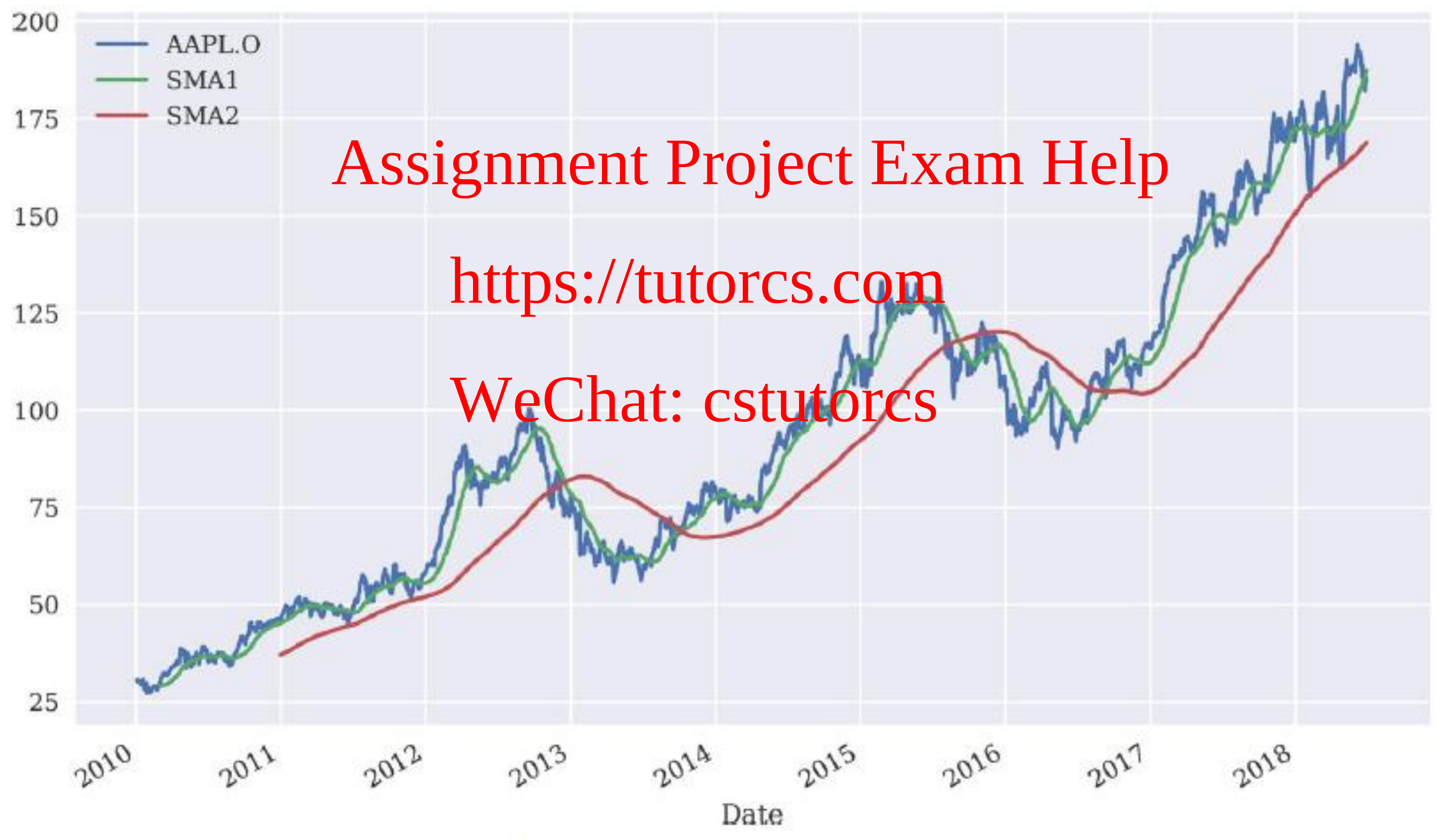


Figure 15-1. Apple stock price and two simple moving averages

Trading Strategy3

- Fourth, the derivation of the positions. The trading rules are:
- Go *long* (= `+1`) when the shorter SMA is **above** the longer SMA.
- Go *short* (= `-1`) when the shorter SMA is **below** the longer SMA.**

** Similarly, for a *long only* strategy one would use `+1` for a *long* position and `0` for a *neutral* position.

The positions are visualized in Figure 15-2:

```
In [10]: data.dropna(inplace=True)

In [11]: data['Position'] = np.where(data['SMA1'] > data['SMA2'], 1, -1)  ❶

In [12]: data.tail()
Out[12]:             AAPL.O        SMA1        SMA2  Position
         Date
         2018-06-25  182.17  185.606190  168.265556         1
         2018-06-26  184.43  186.087381  168.418770         1
         2018-06-27  184.16  186.607381  168.579206         1
         2018-06-28  185.50  187.089286  168.736627         1
         2018-06-29  185.11  187.470476  168.901032         1

In [13]: ax = data.plot(secondary_y='Position', figsize=(10, 6))
         ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```

❶ `np.where(cond, a, b)` evaluates the condition `cond` element-wise and places `a` when `True` and `b` otherwise.

Trading Strategy4

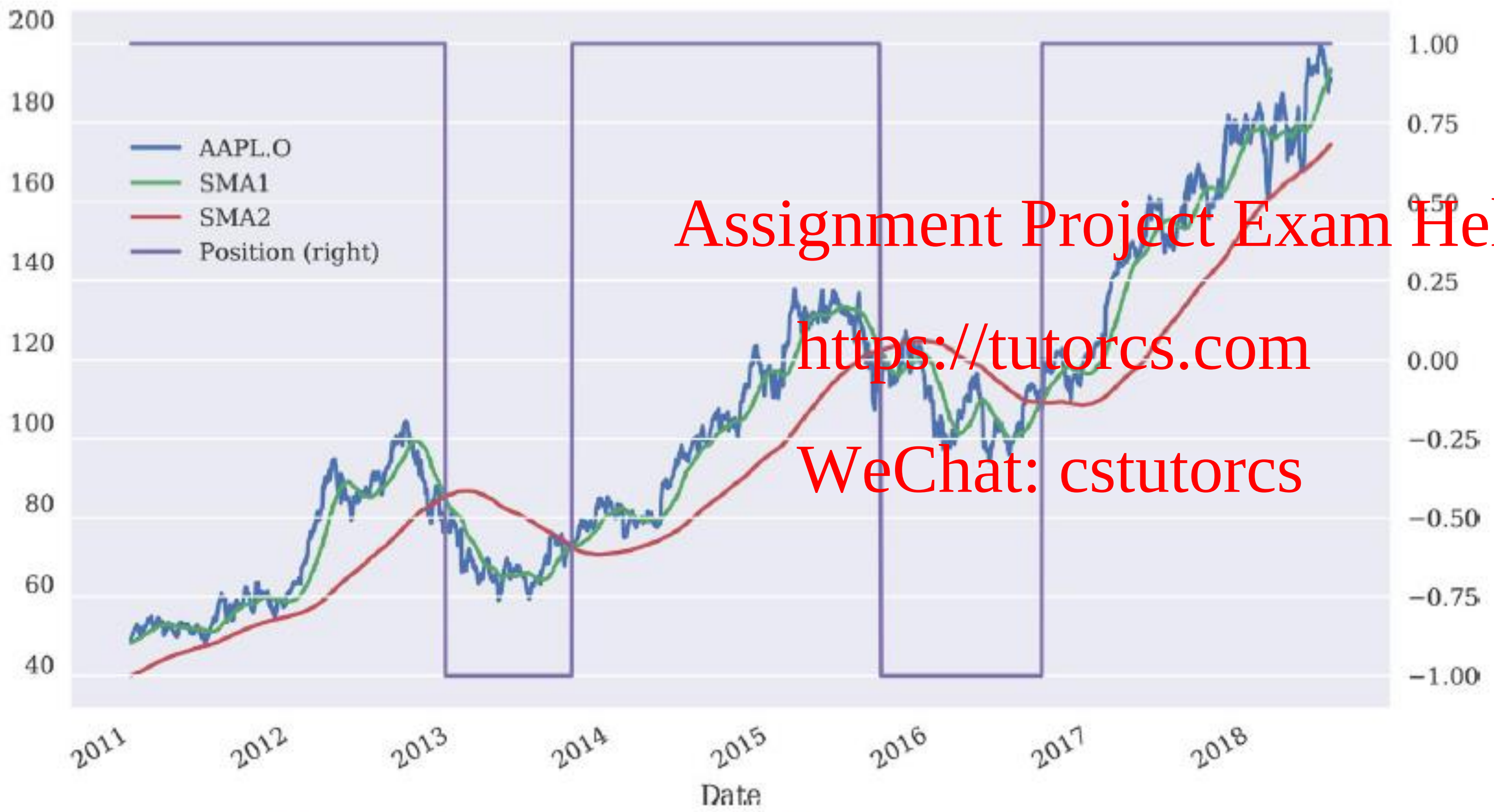


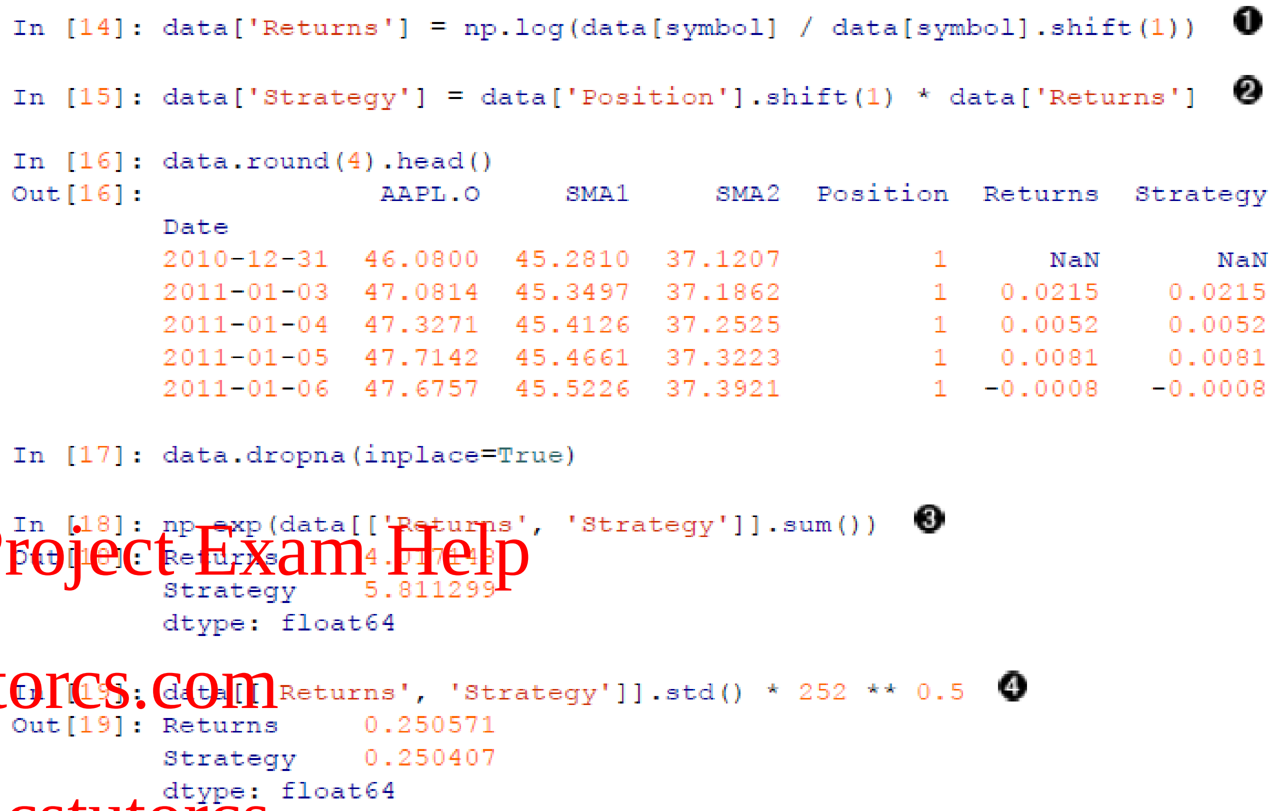
Figure 15-2. Apple stock price, two SMAs, and resulting positions

- This replicates the results derived in Chapter 8.
- What is not addressed there is if following the trading rules — i.e., implementing the algorithmic trading strategy — is superior compared to the benchmark case of simply going long on the Apple stock over the whole period.
- Given that the strategy leads to two periods only during which the Apple stock should be shorted, differences in the performance can only result from these two periods.

Vectorized Backtesting1

- The vectorized backtesting can now be implemented as follows. First, the log returns are calculated.
- Then the positionings, represented as `+1` or `−1`, are multiplied by the relevant log return.
- This simple calculation is possible since a long position earns the return of the Apple stock and a short position earns the negative return of the Apple stock.
- Finally, the log returns for the Apple stock and the algorithmic trading strategy based on SMAs need to be added up and the exponential function applied to arrive at the performance values:

The basic idea is that the algorithm can only set up a position in the Apple stock given *today's market data* (e.g., just before the close). The position then earns *tomorrow's return*.



```
In [14]: data['Returns'] = np.log(data[symbol] / data[symbol].shift(1))  ❶

In [15]: data['Strategy'] = data['Position'].shift(1) * data['Returns']  ❷

In [16]: data.round(4).head()
Out[16]:              AAPL.O     SMA1     SMA2  Position  Returns  Strategy
         Date
         2010-12-31  46.0800  45.2810  37.1207         1      NaN       NaN
         2011-01-03  47.0814  45.3497  37.1862         1   0.0215    0.0215
         2011-01-04  47.3271  45.4126  37.2525         1   0.0052    0.0052
         2011-01-05  47.7142  45.4661  37.3223         1   0.0081    0.0081
         2011-01-06  47.6757  45.5226  37.3921         1  -0.0008   -0.0008

In [17]: data.dropna(inplace=True)

In [18]: np.exp(data[['Returns', 'Strategy']].sum())  ❸
Out[18]: Returns     4.017148
         Strategy    5.811299
         dtype: float64

In [19]: data[['Returns', 'Strategy']].std() * 252 ** 0.5  ❹
Out[19]: Returns     0.250571
         Strategy    0.250407
         dtype: float64
```

❶ Calculates the log returns of the Apple stock (i.e., the benchmark investment).

❷ Multiplies the position values, shifted by one day, by the log returns of the Apple stock; the shift is required to avoid a foresight bias.[5]

❸ Sums up the log returns for the strategy and the benchmark investment and calculates the exponential value to arrive at the absolute performance.

❹ Calculates the annualized volatility for the strategy and the benchmark investment.

Annualised Volatility

To convert daily volatility to other time horizons so that you can make a fair comparison you multiply by the square root of the time scaling factor

$$Vol_A = Vol_D * \sqrt{252}$$

- For weekly returns, Annualized Standard Deviation = Standard Deviation of Weekly Returns * Sqrt(52).
- For monthly returns, Annualized Standard Deviation = Standard Deviation of Monthly Returns * Sqrt(12).
- For quarterly returns, Annualized Standard Deviation = Standard Deviation of Quarterly Returns * Sqrt(4).

Where

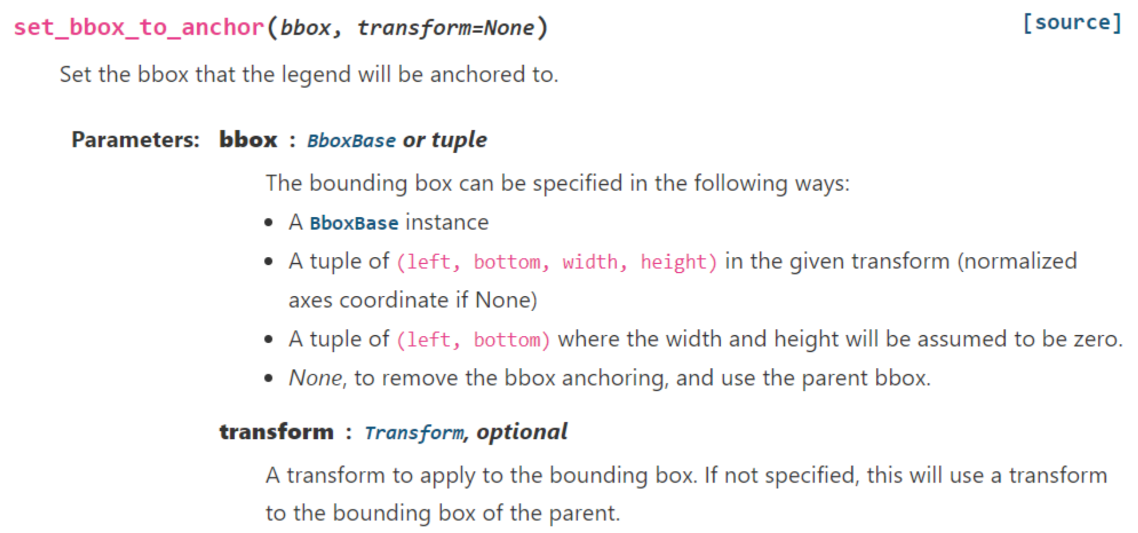
- VolA = Annualised Volatility
- VolD = Daily volatility
- 252 represents the typical number of trading days per year

https://campus.datacamp.com/courses/introduction-to-portfolio-analysis-in-python/risk-and-return?ex=1

Vectorized Backtesting2

- The numbers show that the algorithmic trading strategy indeed outperforms the benchmark investment of passively holding the Apple stock.
- Due to the type and characteristics of the strategy, the annualized volatility is the same, such that it also outperforms the benchmark investment on a risk-adjusted basis.
- To gain a better picture of the overall performance, Figure 15-3 shows the performance of the Apple stock and the algorithmic trading strategy over time:

```
In [20]: ax = data[['Returns', 'Strategy']].cumsum(
                 ).apply(np.exp).plot(figsize=(10, 6))
         data['Position'].plot(ax=ax, secondary_y='Position', style='--')
         ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```



`set_bbox_to_anchor(bbox, transform=None)` [source]

Set the bbox that the legend will be anchored to.

Parameters: **bbox** : *BboxBase or tuple*

The bounding box can be specified in the following ways:

- A `BboxBase` instance
- A tuple of `(left, bottom, width, height)` in the given transform (normalized axes coordinate if None)
- A tuple of `(left, bottom)` where the width and height will be assumed to be zero.
- *None*, to remove the bbox anchoring, and use the parent bbox.

transform : *Transform, optional*

A transform to apply to the bounding box. If not specified, this will use a transform to the bounding box of the parent.

https://matplotlib.org/stable/api/legend_api.html

Vectorized Backtesting3

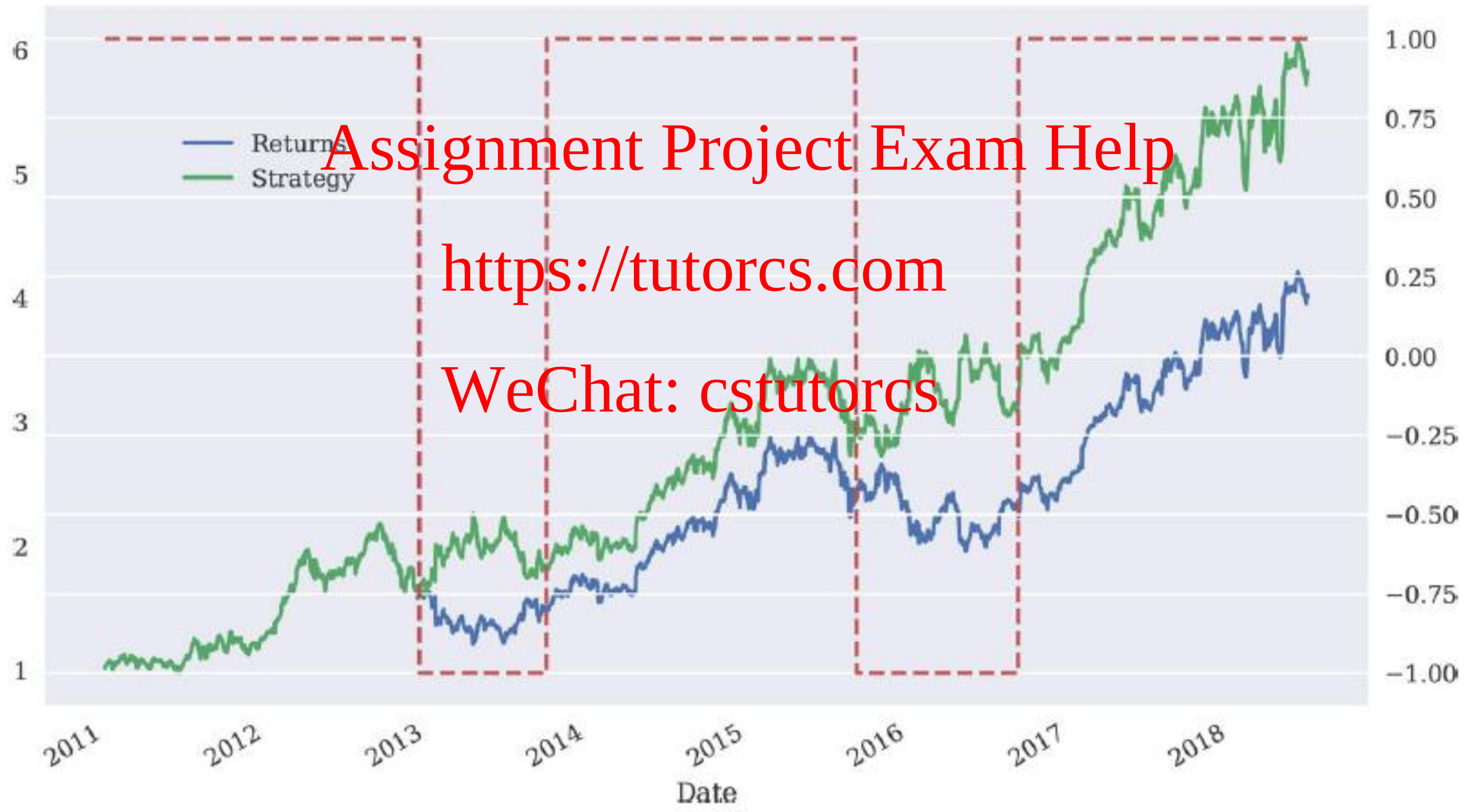
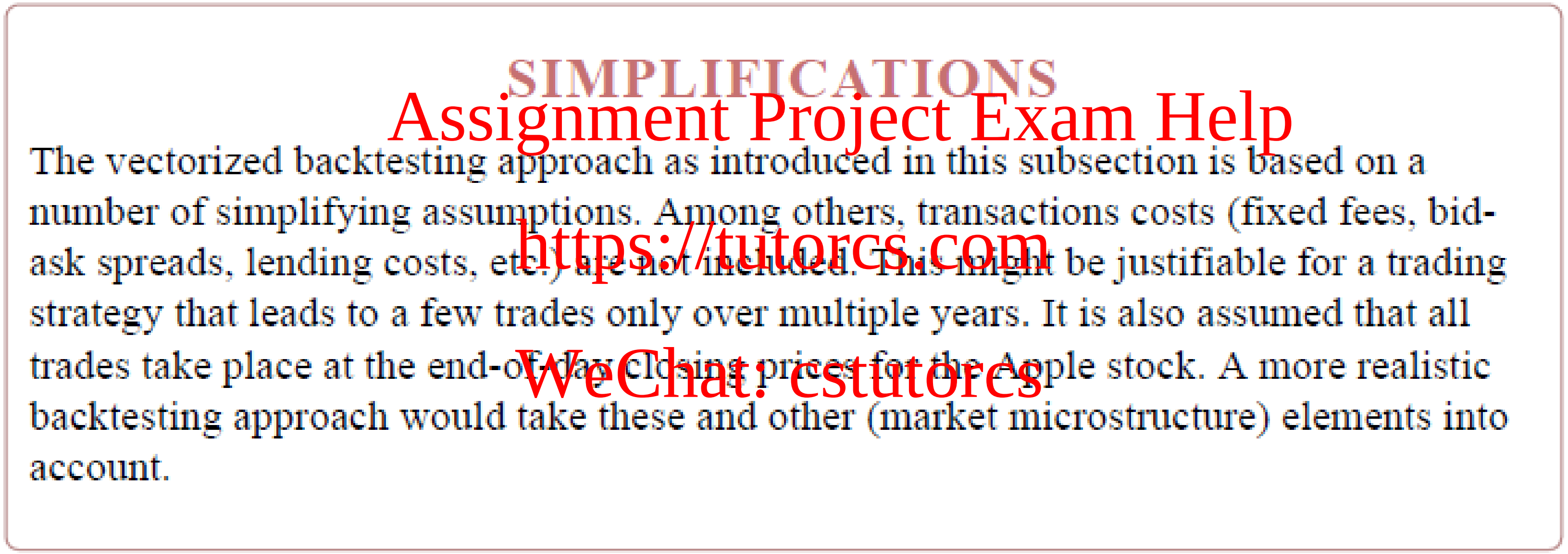


Figure 15-3. Performance of Apple stock and SMA-based trading strategy over time

Vectorized Backtesting4

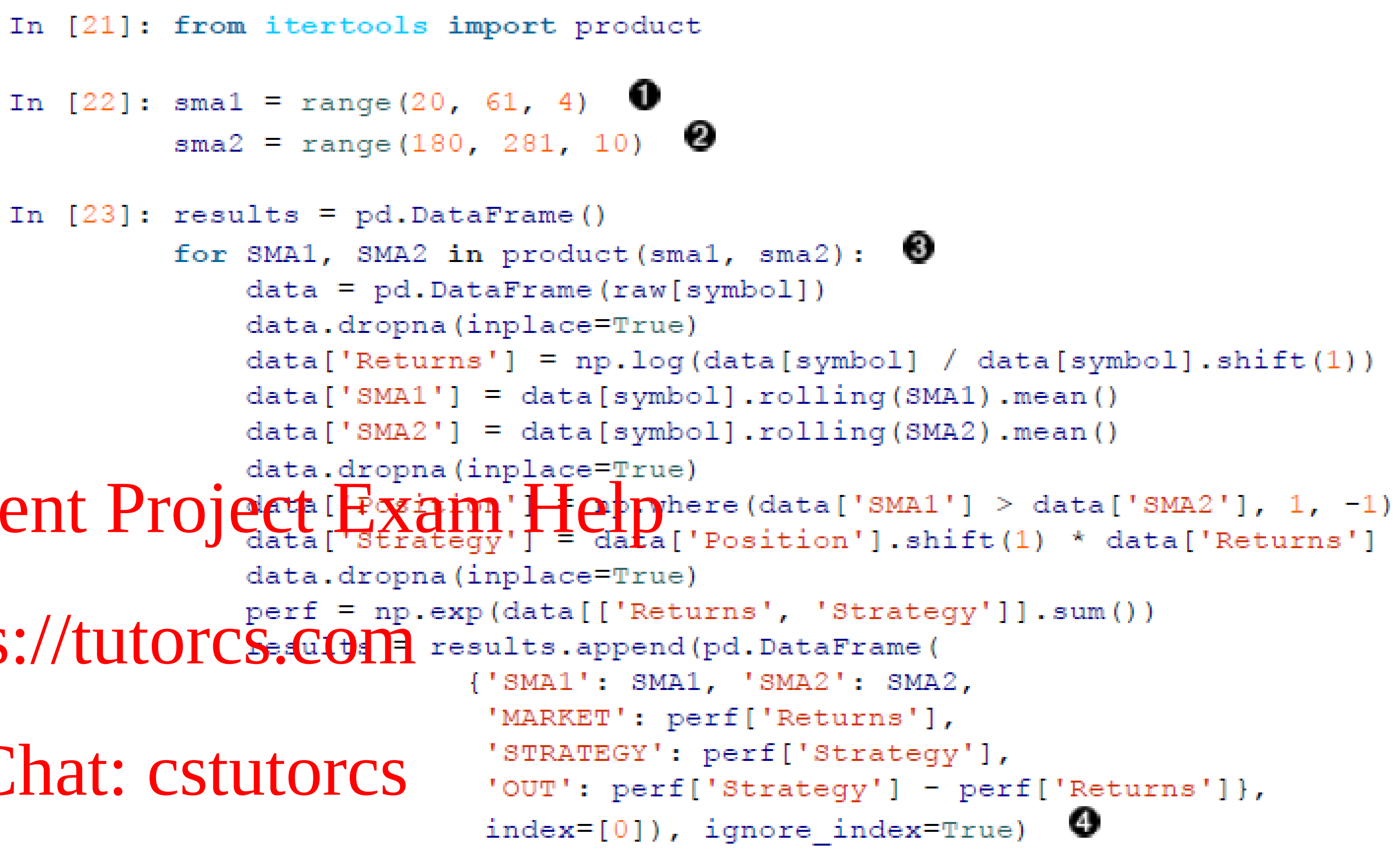


SIMPLIFICATIONS

The vectorized backtesting approach as introduced in this subsection is based on a number of simplifying assumptions. Among others, transactions costs (fixed fees, bid-ask spreads, lending costs, etc.) are not included. This might be justifiable for a trading strategy that leads to a few trades only over multiple years. It is also assumed that all trades take place at the end-of-day closing prices for the Apple stock. A more realistic backtesting approach would take these and other (market microstructure) elements into account.

Optimisation1

- A natural question that arises is if the chosen parameters `SMA1=42` and `SMA2=252` are the “right” ones.
- In general, investors prefer higher returns to lower returns *ceteris paribus*.
- Therefore, one might be inclined to search for those parameters that maximize the return over the relevant period.
- To this end, a brute force approach can be used that simply repeats the whole vectorized backtesting procedure for different parameter combinations, records the results, and does a ranking afterward.
- This is what the following code does:



```
In [21]: from itertools import product

In [22]: sma1 = range(20, 61, 4)  ❶
         sma2 = range(180, 281, 10)  ❷

In [23]: results = pd.DataFrame()
         for SMA1, SMA2 in product(sma1, sma2):  ❸
             data = pd.DataFrame(raw[symbol])
             data.dropna(inplace=True)
             data['Returns'] = np.log(data[symbol] / data[symbol].shift(1))
             data['SMA1'] = data[symbol].rolling(SMA1).mean()
             data['SMA2'] = data[symbol].rolling(SMA2).mean()
             data.dropna(inplace=True)
             data['Position'] = np.where(data['SMA1'] > data['SMA2'], 1, -1)
             data['Strategy'] = data['Position'].shift(1) * data['Returns']
             data.dropna(inplace=True)
             perf = np.exp(data[['Returns', 'Strategy']].sum())
             results = results.append(pd.DataFrame(
                         {'SMA1': SMA1, 'SMA2': SMA2,
                          'MARKET': perf['Returns'],
                          'STRATEGY': perf['Strategy'],
                          'OUT': perf['Strategy'] - perf['Returns']},
                          index=[0]), ignore_index=True)  ❹
```

❶ Specifies the parameter values for `SMA1`.

❷ Specifies the parameter values for `SMA2`.

❸ Combines all values for `SMA1` with those for `SMA2`.

❹ Records the vectorized backtesting results in a `DataFrame` object.

Fitting Overview

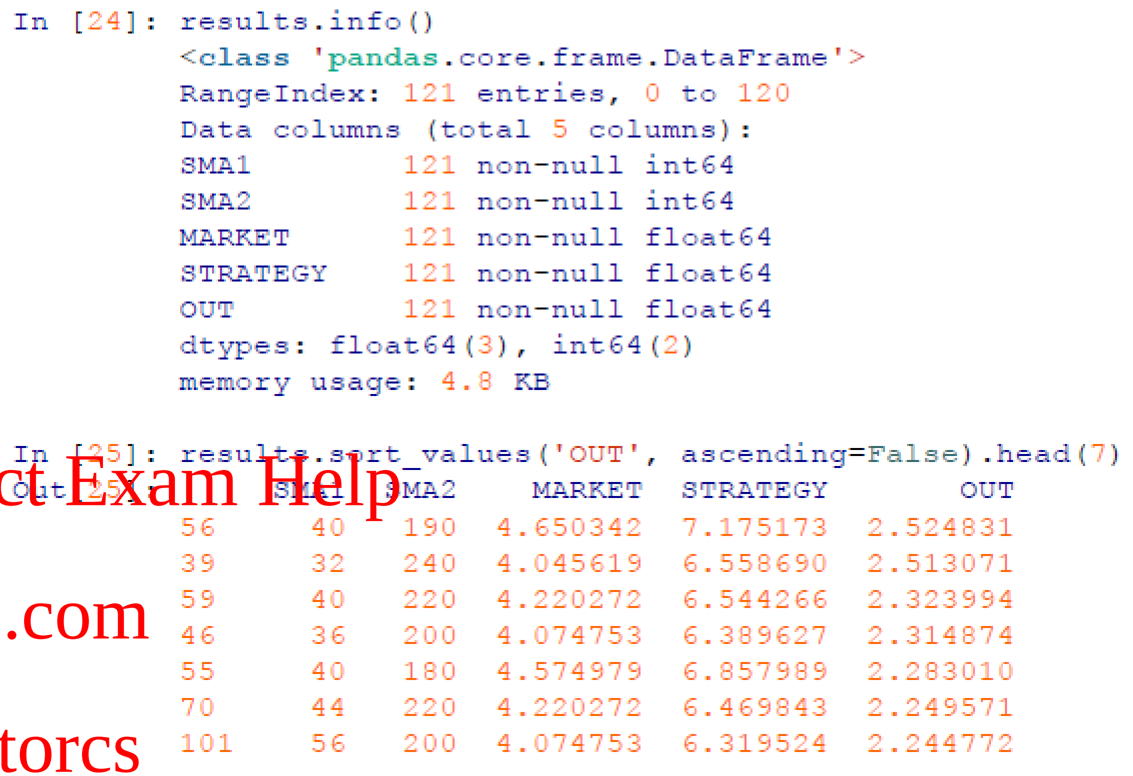
- The process of discovering what the optimal set of parameters is (normally highest return or highest Sharpe Ratio)
- Done within a **given parameter space** (parameters and ranges of possible values)
- We find the best possible parameters in a set of **in sample** data (or training data)
- We test the performance of the model on a different set of **out of sample** data (or test / evaluation data)

Example:

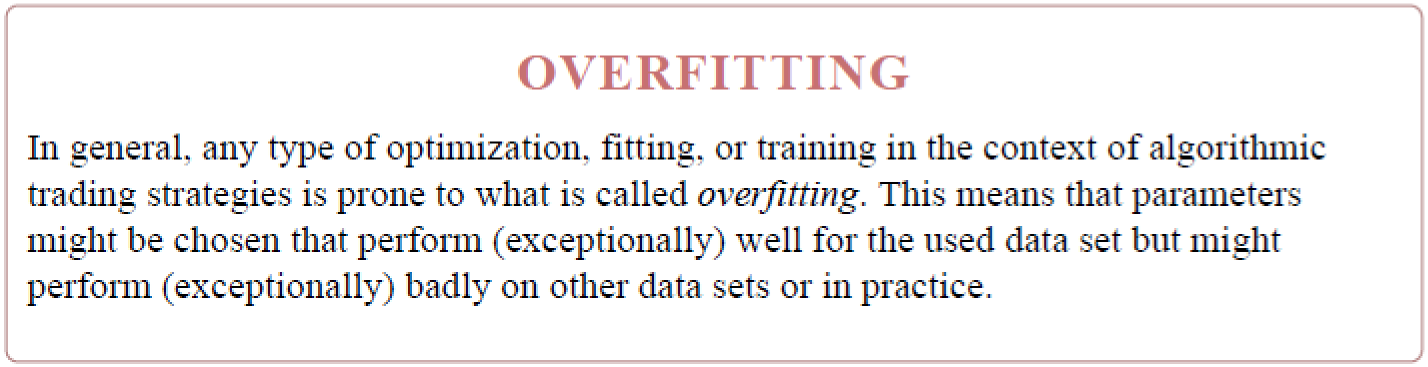
- In sample data: 1977 – 1986
- Out of sample test: 1987 - 2017

Optimisation2

- The following code gives an overview of the results and shows the seven best-performing parameter combinations of all those backtested. The ranking is implemented according to the outperformance of the algorithmic trading strategy compared to the benchmark investment.
- The performance of the benchmark investment varies since the choice of the SMA2 parameter influences the length of the time interval and data set on which the vectorized backtest is implemented:
- According to the brute force–based optimization, SMA1=40 and SMA2=190 are the optimal parameters, leading to an outperformance of some 230 percentage points.
- However, this result is heavily dependent on the data set used and is prone to overfitting.
- A more rigorous approach would be to implement the optimization on one data set, the in-sample or training data set, and test it on another one, the out-of-sample or testing data set.



```
In [24]: results.info()
         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 121 entries, 0 to 120
         Data columns (total 5 columns):
         SMA1        121 non-null int64
         SMA2        121 non-null int64
         MARKET      121 non-null float64
         STRATEGY    121 non-null float64
         OUT         121 non-null float64
         dtypes: float64(3), int64(2)
         memory usage: 4.8 KB

In [25]: results.sort_values('OUT', ascending=False).head(7)
Out[25]:      SMA1  SMA2    MARKET  STRATEGY       OUT
         56     40   190  4.650342  7.175173  2.524831
         39     32   240  4.045619  6.558690  2.513071
         59     40   220  4.220272  6.544266  2.323994
         46     36   200  4.074753  6.389627  2.314874
         55     40   180  4.574979  6.857989  2.283010
         70     44   220  4.220272  6.469843  2.249571
         101    56   200  4.074753  6.319524  2.244772
```



OVERFITTING

In general, any type of optimization, fitting, or training in the context of algorithmic trading strategies is prone to what is called *overfitting*. This means that parameters might be chosen that perform (exceptionally) well for the used data set but might perform (exceptionally) badly on other data sets or in practice.

Random Walk Hypothesis Introduction

- Random walk theory suggests that changes in stock prices have the same distribution and are independent of each other. Therefore, it assumes the past movement or trend of a stock price or market cannot be used to predict its future movement. In short, random walk theory proclaims that stocks take a random and unpredictable path that makes all methods of predicting stock prices futile in the long run.
- Random walk theory suggests that changes in stock prices have the same distribution and are independent of each other.
- Random walk theory infers that the past movement or trend of a stock price or market cannot be used to predict its future movement.
- Random walk theory believes it's impossible to outperform the market without assuming additional risk.
- Random walk theory considers technical analysis undependable because it results in chartists only buying or selling a security after a move has occurred.
- Random walk theory considers fundamental analysis undependable due to the often-poor quality of information collected and its ability to be misinterpreted.

Random Walk Hypothesis1

- The previous section introduces vectorized backtesting as an efficient tool to backtest algorithmic trading strategies. The single strategy backtested based on a single financial time series, namely historical end-of-day prices for the Apple stock, outperforms the benchmark investment of simply going long on the Apple stock over the same period.
- Although rather specific in nature, these results are in contrast to what the *random walk hypothesis* (RWH) predicts, namely that such predictive approaches should not yield any outperformance at all.
- The RWH postulates that prices in financial markets follow a random walk, or, in continuous time, an arithmetic Brownian motion without drift. The expected value of an arithmetic Brownian motion without drift at any point in the future equals its value today.**
- As a consequence, the best predictor for tomorrow's price, in a least-squares sense, is today's price if the RWH applies.
- The consequences are summarized in the following quote:
- "For many years, economists, statisticians, and teachers of finance have been interested in developing and testing models of stock price behaviour. One important model that has evolved from this research is the theory of random walks. This theory casts serious doubt on many other methods for describing and predicting stock price behavior — methods that have considerable popularity outside the academic world. For example, we shall see later that, if the random-walk theory is an accurate description of reality, then the various "technical" or "chartist" procedures for predicting stock prices are completely without value." Eugene F. Fama (1965)

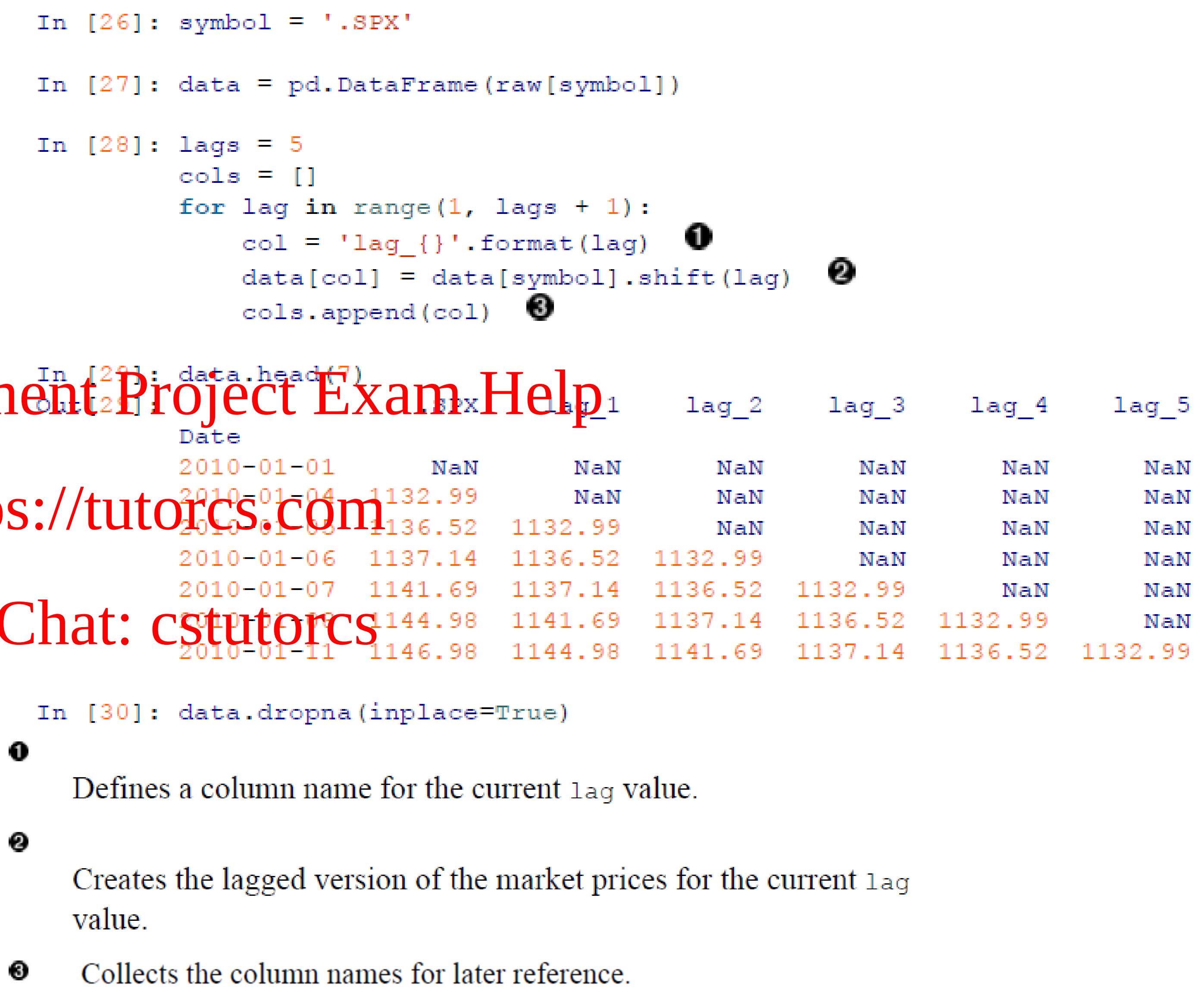
- ** For a formal definition and deeper discussion of random walks and Brownian motion–based processes, refer to Baxter and Rennie (1996).

Random Walk Hypothesis2

- The RWH is consistent with the *efficient markets hypothesis* (EMH), which, non-technically speaking, states that market prices reflect "all available information." Different degrees of efficiency are generally distinguished, such as *weak*, *semi-strong*, and *strong*, defining more specifically what "all available information" entails. Formally, such a definition can be based on the concept of an information set in theory and on a data set for programming purposes, as the following quote illustrates:
- A market is efficient with respect to an information set *S* if it is impossible to make economic profits by trading on the basis of information set *S*. Michael Jensen (1978)

Random Walk Hypothesis3

- Using Python, the RWH can be tested for a specific case as follows.
- A financial time series of historical market prices is used for which a number of *lagged* versions are created — say, five.
- OLS regression is then used to predict the market prices based on the lagged market prices created before.
- The basic idea is that the market prices from yesterday and four more days back can be used to predict today’s market price.
- The following Python code implements this idea and creates five lagged versions of the historical end-of-day closing levels of the S&P 500 stock index:



```
In [26]: symbol = '.SPX'

In [27]: data = pd.DataFrame(raw[symbol])

In [28]: lags = 5
         cols = []
         for lag in range(1, lags + 1):
             col = 'lag_{}'.format(lag)  ❶
             data[col] = data[symbol].shift(lag)  ❷
             cols.append(col)  ❸

In [29]: data.head(7)
Out[29]:              .SPX    lag_1    lag_2    lag_3    lag_4    lag_5
         Date
         2010-01-01       NaN      NaN      NaN      NaN      NaN      NaN
         2010-01-04   1132.99      NaN      NaN      NaN      NaN      NaN
         2010-01-05   1136.52  1132.99      NaN      NaN      NaN      NaN
         2010-01-06   1137.14  1136.52  1132.99      NaN      NaN      NaN
         2010-01-07   1141.69  1137.14  1136.52  1132.99      NaN      NaN
         2010-01-08   1144.98  1141.69  1137.14  1136.52  1132.99      NaN
         2010-01-11   1146.98  1144.98  1141.69  1137.14  1136.52  1132.99

In [30]: data.dropna(inplace=True)
```

❶ Defines a column name for the current `lag` value.

❷ Creates the lagged version of the market prices for the current `lag` value.

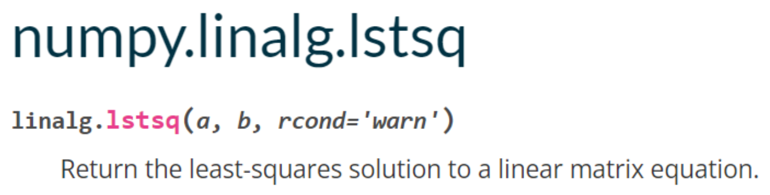
❸ Collects the column names for later reference.

OLS Regression

The **numpy linalg lstsq()** function returns the least-squares solution to a linear matrix equation. For example, it solves the equation **ax = b** by computing a vector x that minimizes the **Euclidean 2-norm || b – ax ||^2**.

```
reg = np.linalg.lstsq(data[cols], data[symbol], rcond=-1)[0]
```

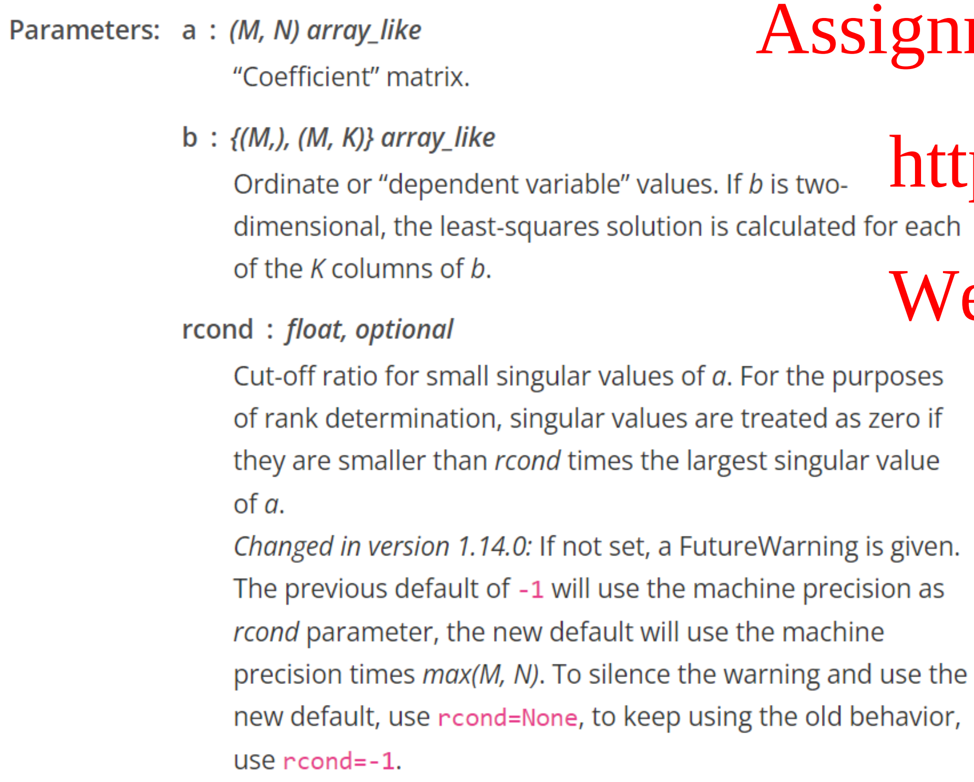
#[0] is X depicts the least-squares solution.



numpy.linalg.lstsq

`linalg.lstsq(a, b, rcond='warn')`

Return the least-squares solution to a linear matrix equation.



Parameters: **a** : *(M, N) array_like*

"Coefficient" matrix.

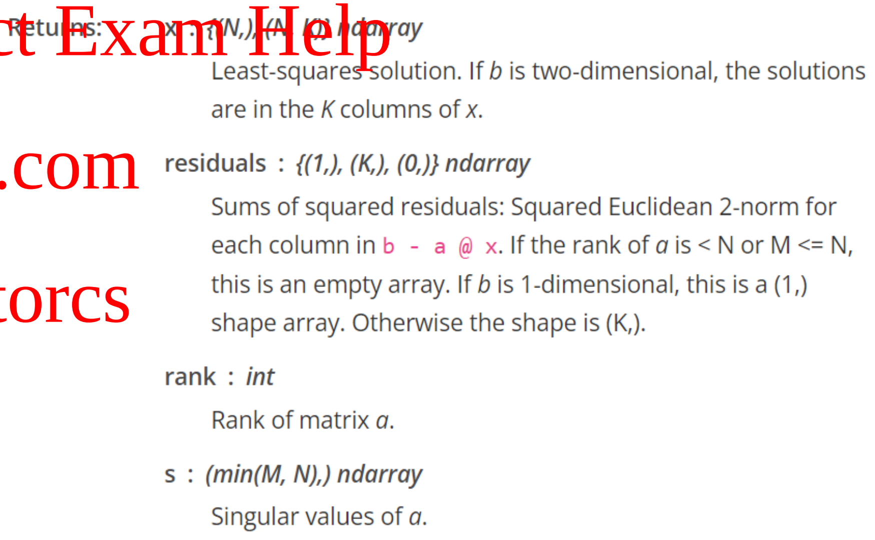
b : *{(M,), (M, K)} array_like*

Ordinate or "dependent variable" values. If *b* is two-dimensional, the least-squares solution is calculated for each of the *K* columns of *b*.

rcond : *float, optional*

Cut-off ratio for small singular values of *a*. For the purposes of rank determination, singular values are treated as zero if they are smaller than *rcond* times the largest singular value of *a*.

Changed in version 1.14.0: If not set, a FutureWarning is given. The previous default of `-1` will use the machine precision as *rcond* parameter, the new default will use the machine precision times *max(M, N)*. To silence the warning and use the new default, use `rcond=None`, to keep using the old behavior, use `rcond=-1`.



Returns: **x** : *{(N,), (N, K)} ndarray*

Least-squares solution. If *b* is two-dimensional, the solutions are in the *K* columns of *x*.

residuals : *{(1,), (K,), (0,)} ndarray*

Sums of squared residuals: Squared Euclidean 2-norm for each column in `b - a @ x`. If the rank of *a* is < N or M <= N, this is an empty array. If *b* is 1-dimensional, this is a (1,) shape array. Otherwise the shape is (K,).

rank : *int*

Rank of matrix *a*.

s : *(min(M, N),) ndarray*

Singular values of *a*.

https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html
Please read this additional explanation:
https://appdividend.com/2021/01/18/numpy-linalg-lstsq-function-in-python/

Random Walk Hypothesis4

- Using `NumPy`, the OLS regression is implemented.
- As the optimal regression parameters show, `lag_1` indeed is the most important one in predicting the market price based on OLS regression.
- Its value is close to 1. The other four values are rather close to 0.
- Figure 15-4 visualizes the optimal regression parameter values

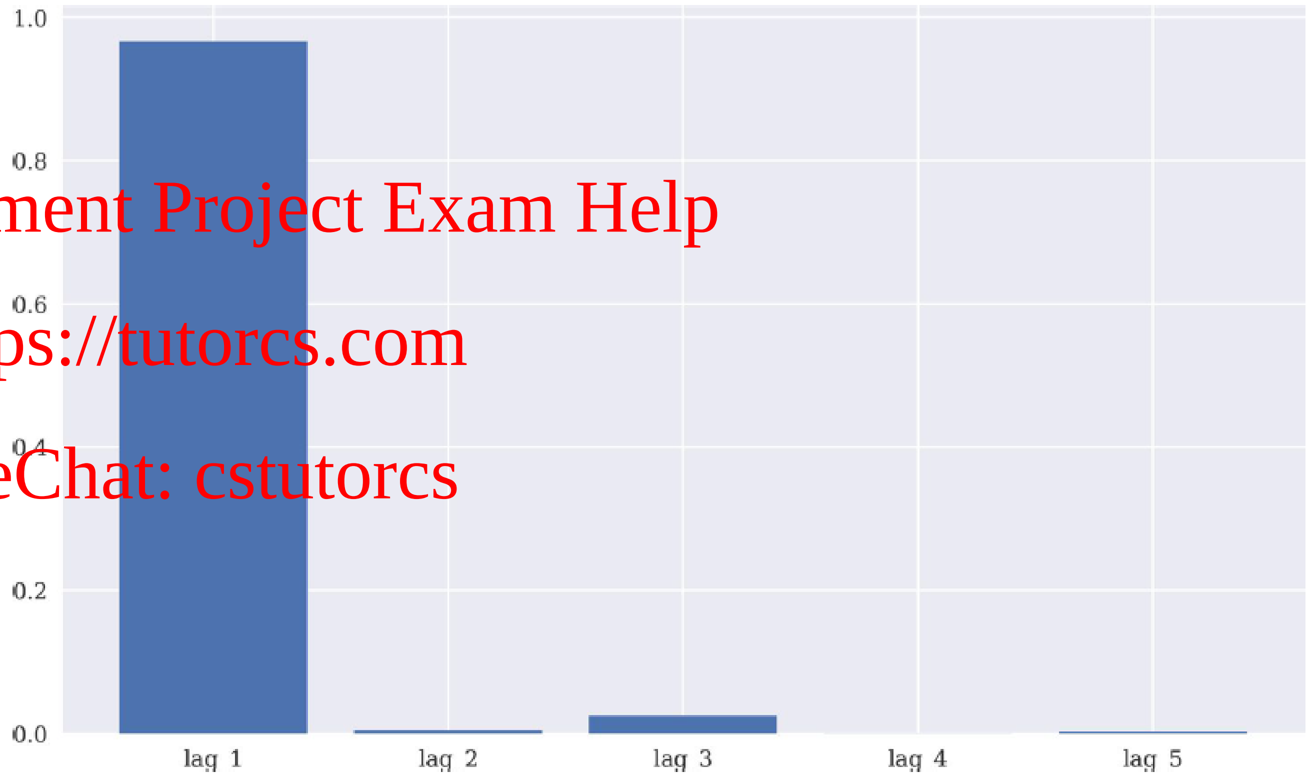
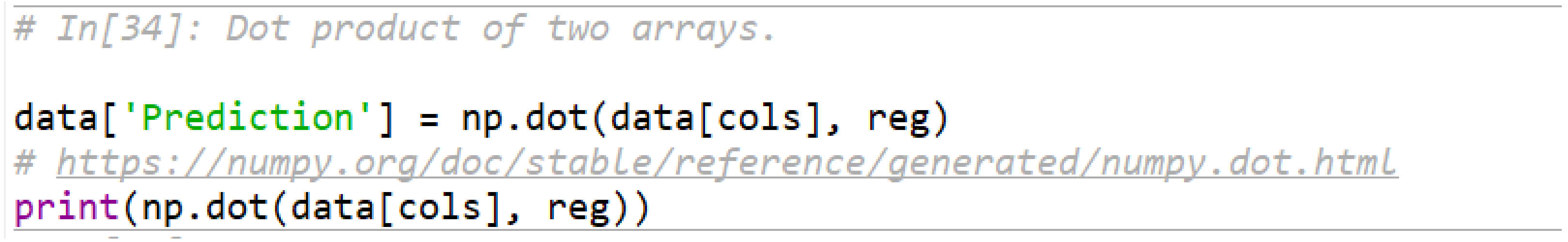
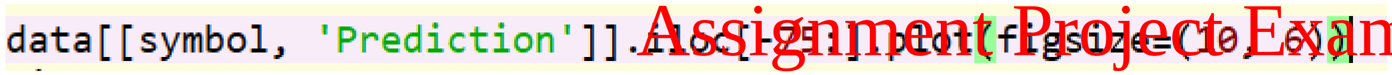


Figure 15-4. Optimal regression parameters from OLS regression for price prediction

Random Walk Hypothesis5



```
# In[34]: Dot product of two arrays.

data['Prediction'] = np.dot(data[cols], reg)
# https://numpy.org/doc/stable/reference/generated/numpy.dot.html
print(np.dot(data[cols], reg))
```



```
data[[symbol, 'Prediction']].iloc[-75:].plot(figsize=(10, 6));
```

- When using the optimal results to visualize the prediction values as compared to the original index values for the S&P 500, it becomes obvious from Figure 15-5 that indeed `lag_1` is basically what is used to come up with the prediction value.
- Graphically speaking, the prediction line in Figure 15-5 is the original time series shifted by one day to the right (with some minor adjustments).

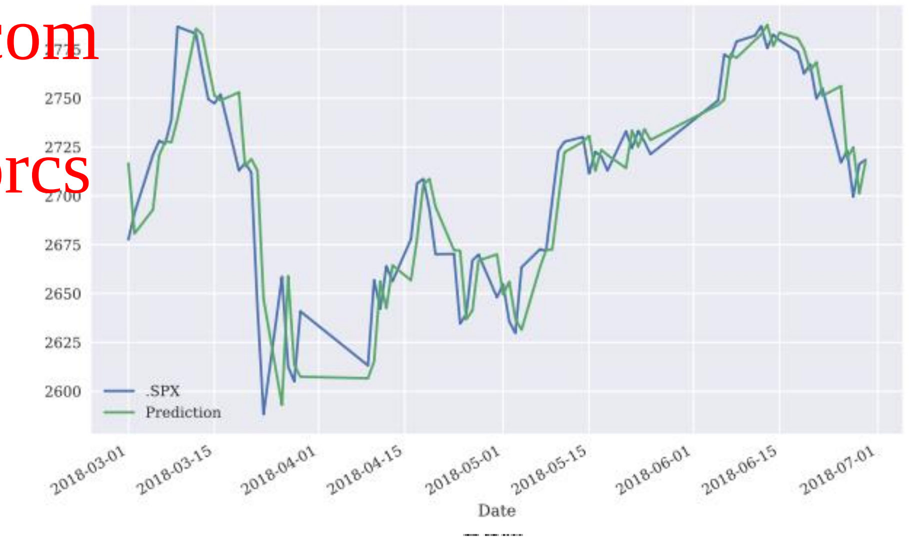


Figure 15-5. S&P 500 levels compared to prediction values from OLS regression

https://numpy.org/doc/stable/reference/generated/numpy.dot.html

Random Walk Hypothesis

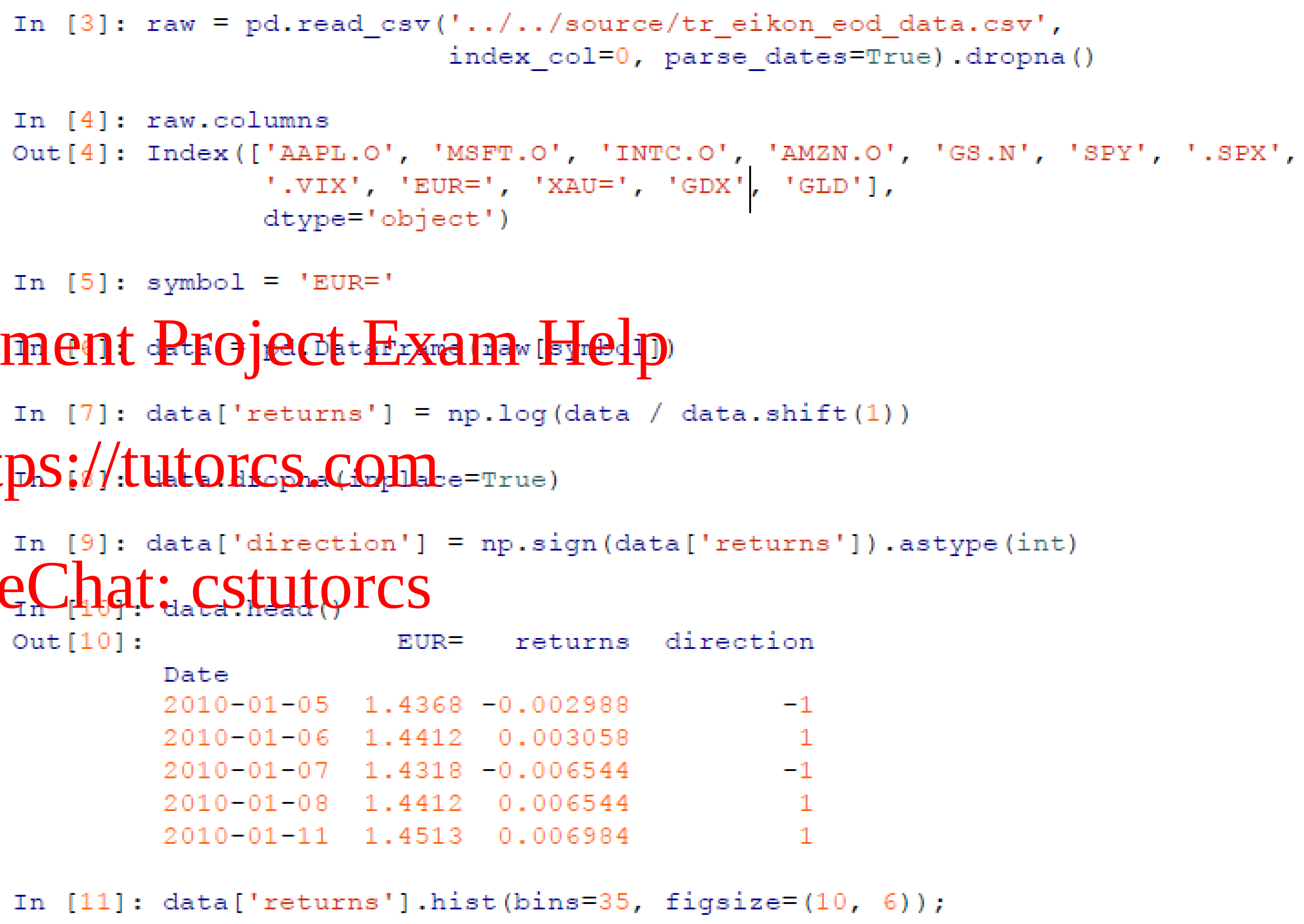
- All in all, the brief analysis in this section reveals some support for both the RWH and the EMH.
- For sure, the analysis is done for a single stock index only and uses a rather specific parameterization — but this can easily be widened to incorporate multiple financial instruments across multiple asset classes, different values for the number of lags, etc.
- In general, one will find out that the results are qualitatively more or less the same.
- After all, the RWH and EMH are among the financial theories that have broad empirical support.
- In that sense, any algorithmic trading strategy must prove its worth by proving that the RWH does not apply in general. This for sure is a tough hurdle.

Linear OLS Regression

- This section applies *linear OLS regression* to predict the direction of market movements based on historical log returns.
- To keep things simple, only two features are used. The first feature (`lag_1`) represents the log returns of the financial time series lagged by *one day*.
- The second feature (`lag_2`) lags the log returns by *two days*. Log returns — in contrast to prices — are *stationary* in general, which often is a necessary condition for the application of statistical algorithms.
- The basic idea behind the usage of lagged log returns as features is that they might be informative in predicting future returns.
- For example, one might hypothesize that after two downward movements an upward movement is more likely (“mean reversion”), or, to the contrary, that another downward movement is more likely (“momentum” or “trend”).
- The application of regression techniques allows the formalization of such informal reasonings.

The Data1

- First, the importing and preparation of the data set. Figure 15-6 shows the
- frequency distribution of the daily historical log returns for the EUR/USD
- exchange rate. They are the basis for the features as well as the labels to be
- used in what follows:



```
In [3]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                          index_col=0, parse_dates=True).dropna()

In [4]: raw.columns
Out[4]: Index(['AAPL.O', 'MSFT.O', 'INTC.O', 'AMZN.O', 'GS.N', 'SPY', '.SPX',
               '.VIX', 'EUR=', 'XAU=', 'GDX', 'GLD'],
              dtype='object')

In [5]: symbol = 'EUR='

In [6]: data = pd.DataFrame(raw[symbol])

In [7]: data['returns'] = np.log(data / data.shift(1))

In [8]: data.dropna(inplace=True)

In [9]: data['direction'] = np.sign(data['returns']).astype(int)

In [10]: data.head()
Out[10]:               EUR=   returns  direction
         Date
         2010-01-05  1.4368 -0.002988         -1
         2010-01-06  1.4412  0.003058          1
         2010-01-07  1.4318 -0.006544         -1
         2010-01-08  1.4412  0.006544          1
         2010-01-11  1.4513  0.006984          1

In [11]: data['returns'].hist(bins=35, figsize=(10, 6));
```

https://numpy.org/doc/stable/reference/generated/numpy.sign.html

The **`sign`** function returns `-1 if x < 0, 0 if x==0, 1 if x > 0`. nan is returned for nan inputs.

The Data2

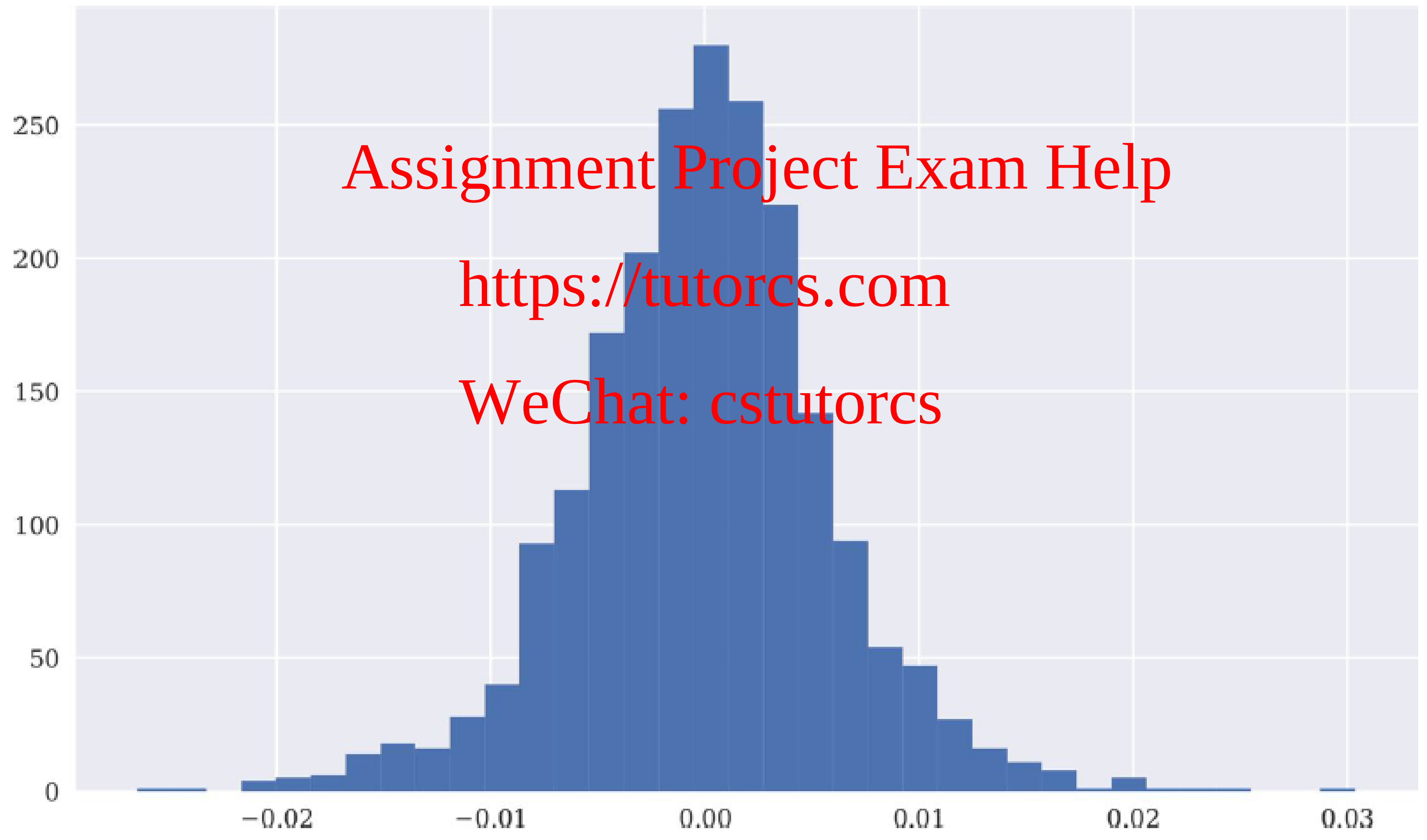
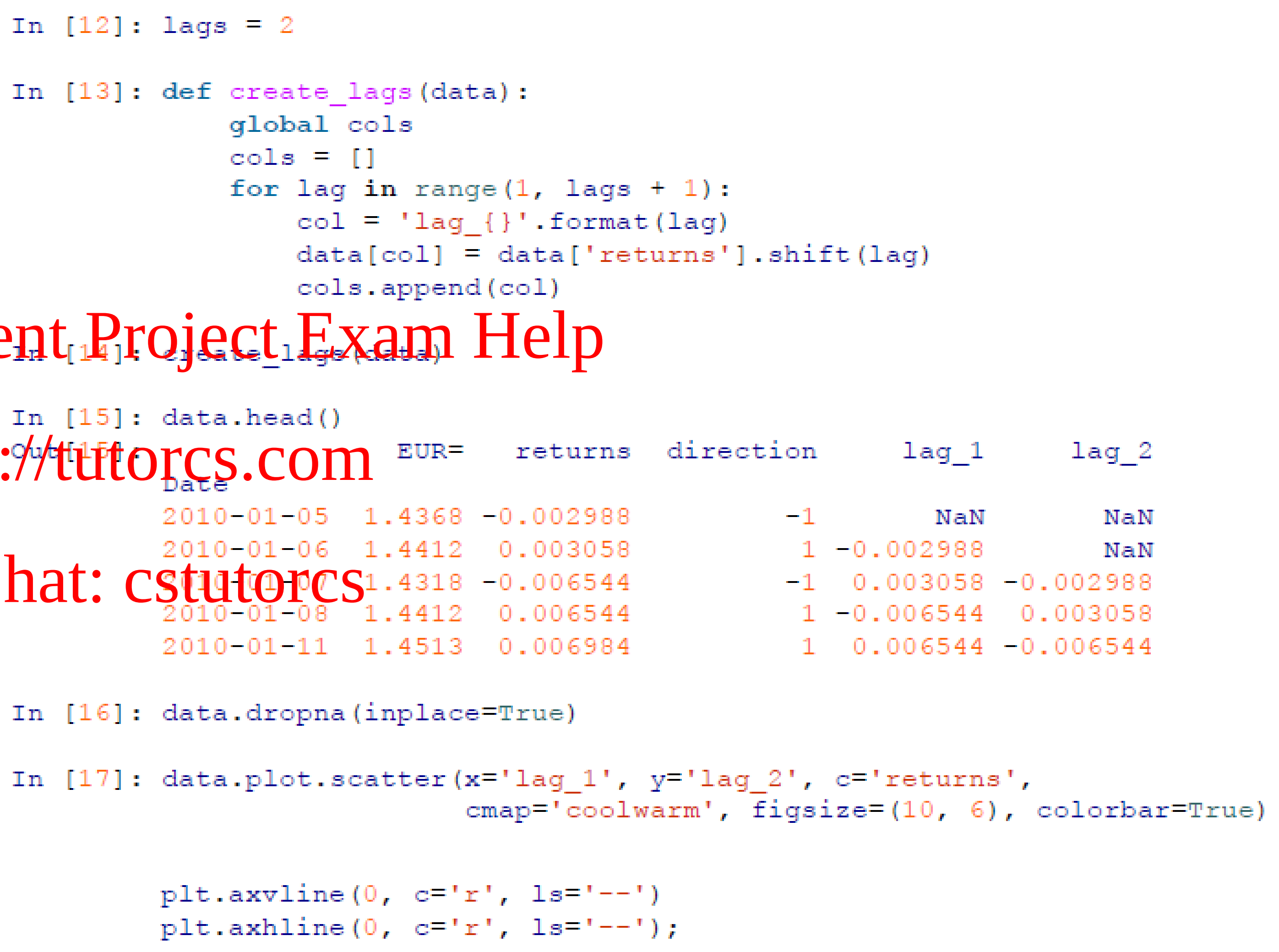


Figure 15-6. Histogram of log returns for EUR/USD exchange rate

The Data3

- Second, the code that creates the features data by lagging the log returns and visualizes it in combination with the returns data (see Figure 15-7):



```
In [12]: lags = 2

In [13]: def create_lags(data):
             global cols
             cols = []
             for lag in range(1, lags + 1):
                 col = 'lag_{}'.format(lag)
                 data[col] = data['returns'].shift(lag)
                 cols.append(col)

In [14]: create_lags(data)

In [15]: data.head()
Out[15]:         EUR=   returns  direction     lag_1     lag_2
         Date
         2010-01-05  1.4368 -0.002988        -1       NaN       NaN
         2010-01-06  1.4412  0.003058         1 -0.002988       NaN
         2010-01-07  1.4318 -0.006544        -1  0.003058 -0.002988
         2010-01-08  1.4412  0.006544         1 -0.006544  0.003058
         2010-01-11  1.4513  0.006984         1  0.006544 -0.006544

In [16]: data.dropna(inplace=True)

In [17]: data.plot.scatter(x='lag_1', y='lag_2', c='returns',
                           cmap='coolwarm', figsize=(10, 6), colorbar=True)

         plt.axvline(0, c='r', ls='--')
         plt.axhline(0, c='r', ls='--');
```

The Data4

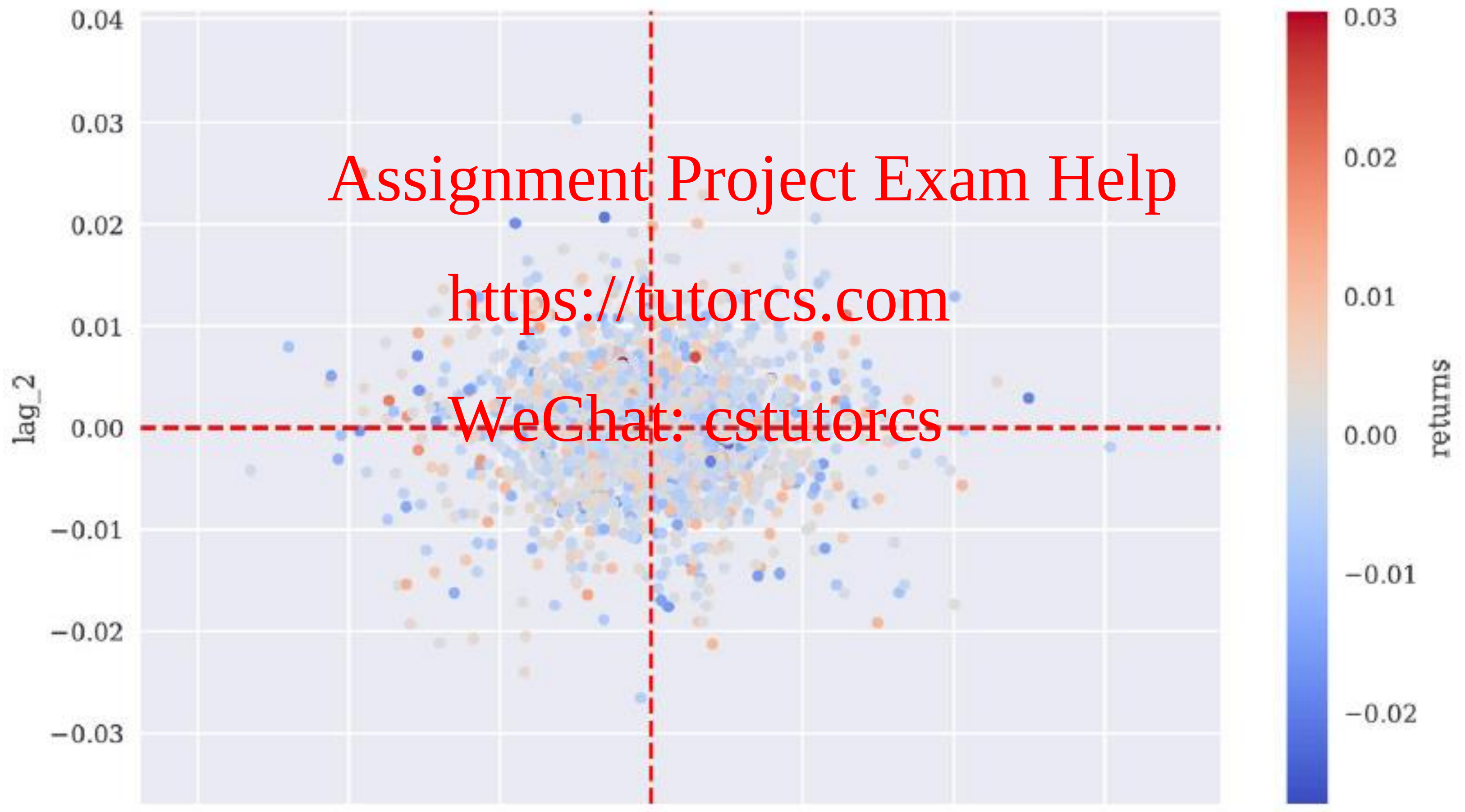


Figure 15-7. Scatter plot based on features and labels data

Regression1

- With the data set completed, linear OLS regression can be applied to learn about any potential (linear) relationships, to predict market movement based on the features, and to backtest a trading strategy based on the predictions.
- Two basic approaches are available: using the *log returns* or only the *direction data* as the dependent variable during the regression. In any case, predictions are real-valued and therefore transformed to either `+1` or `-1` to only work with the direction of the prediction:

Regression2

❶ The linear OLS regression implementation from scikit-learn is used.

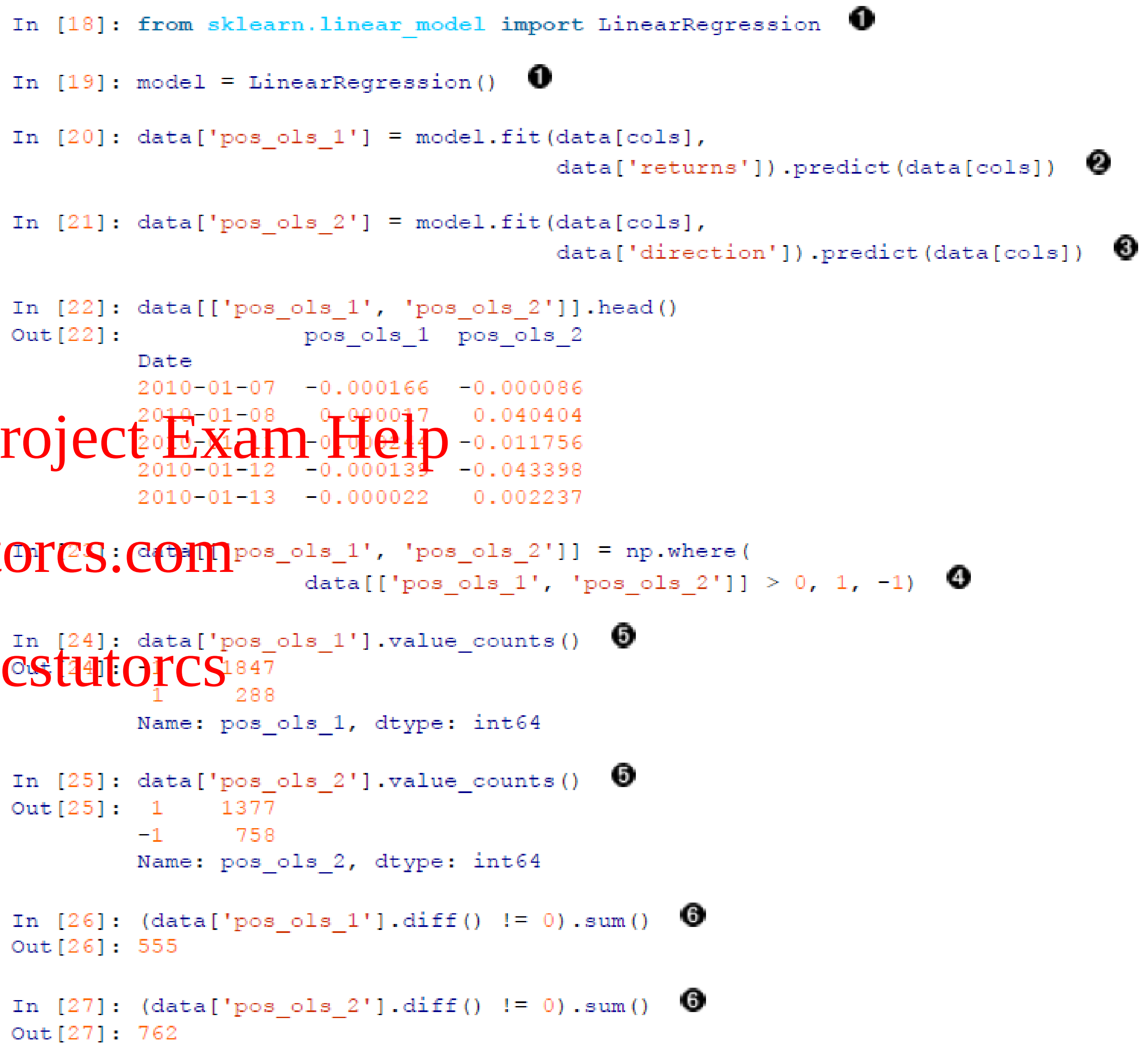
❷ The regression is implemented on the *log returns* directly …

❸ … and on the *direction data* which is of primary interest.

❹ The real-valued predictions are transformed to directional values (+1, -1).

❺ The two approaches yield different directional predictions in general.

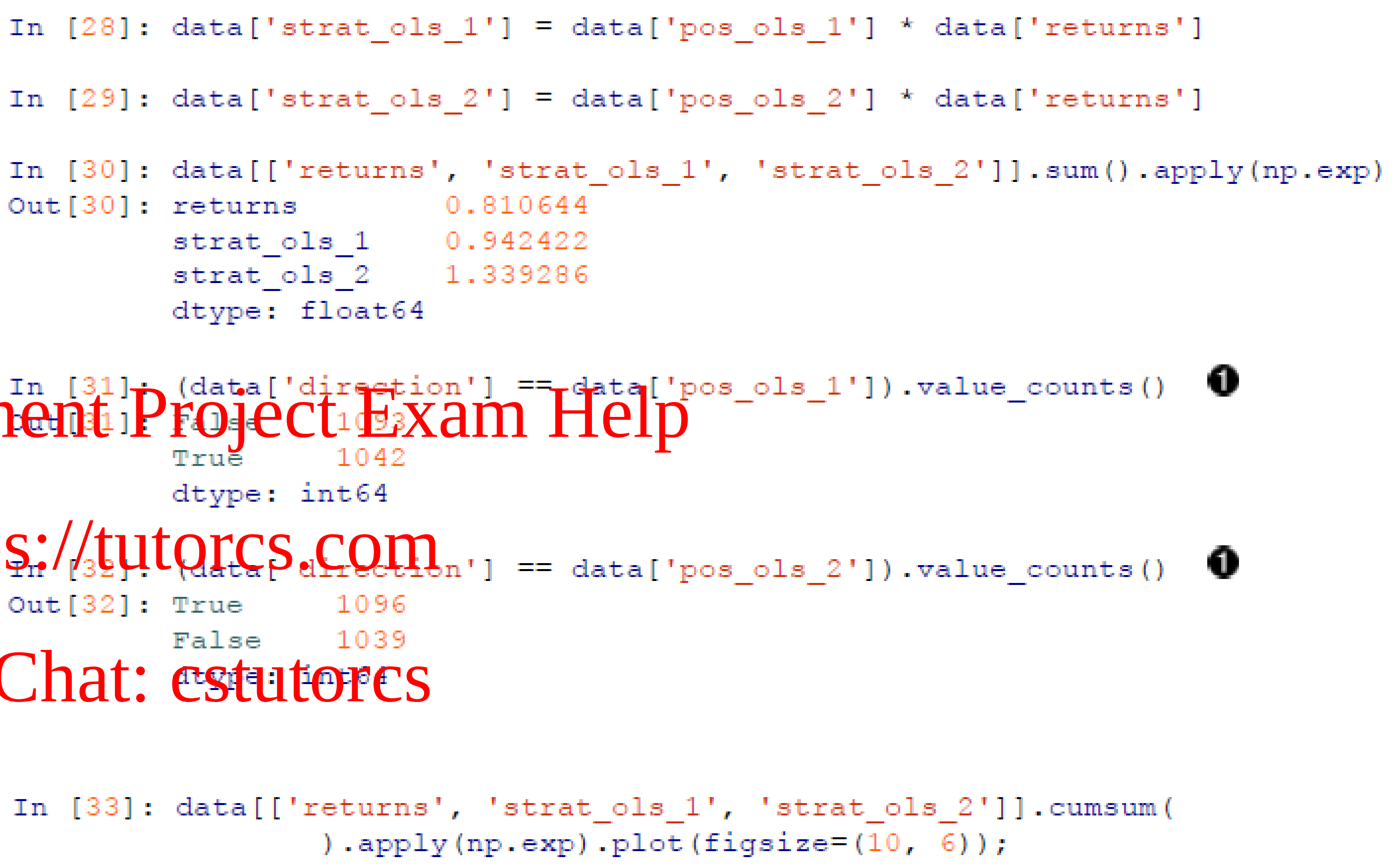
❻ However, both lead to a relatively large number of trades over time.


```
In [18]: from sklearn.linear_model import LinearRegression  ❶

In [19]: model = LinearRegression()  ❶

In [20]: data['pos_ols_1'] = model.fit(data[cols],
                                 data['returns']).predict(data[cols])  ❷

In [21]: data['pos_ols_2'] = model.fit(data[cols],
                                 data['direction']).predict(data[cols])  ❸

In [22]: data[['pos_ols_1', 'pos_ols_2']].head()
Out[22]:             pos_ols_1  pos_ols_2
         Date
         2010-01-07  -0.000166  -0.000086
         2010-01-08   0.000017   0.040404
         2010-01-11  -0.000044  -0.011756
         2010-01-12  -0.000139  -0.043398
         2010-01-13  -0.000022   0.002237

In [23]: data[['pos_ols_1', 'pos_ols_2']] = np.where(
                     data[['pos_ols_1', 'pos_ols_2']] > 0, 1, -1)  ❹

In [24]: data['pos_ols_1'].value_counts()  ❺
Out[24]: -1    1847
          1     288
         Name: pos_ols_1, dtype: int64

In [25]: data['pos_ols_2'].value_counts()  ❺
Out[25]:  1    1377
         -1     758
         Name: pos_ols_2, dtype: int64

In [26]: (data['pos_ols_1'].diff() != 0).sum()  ❻
Out[26]: 555

In [27]: (data['pos_ols_2'].diff() != 0).sum()  ❻
Out[27]: 762
```

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

Regression3

- Equipped with the directional prediction, vectorized backtesting can be applied to judge the performance of the resulting trading strategies.
- At this stage, the analysis is based on a number of simplifying assumptions, such as “zero transaction costs” and the usage of the same data set for both training and testing.
- Under these assumptions, however, both regression-based strategies outperform the benchmark passive investment, while only the strategy trained on the direction of the market shows a positive overall performance (Figure 15-8):



```
In [28]: data['strat_ols_1'] = data['pos_ols_1'] * data['returns']

In [29]: data['strat_ols_2'] = data['pos_ols_2'] * data['returns']

In [30]: data[['returns', 'strat_ols_1', 'strat_ols_2']].sum().apply(np.exp)
Out[30]: returns        0.810644
         strat_ols_1    0.942422
         strat_ols_2    1.339286
         dtype: float64

In [31]: (data['direction'] == data['pos_ols_1']).value_counts()  ❶
Out[31]: False    1093
         True     1042
         dtype: int64

In [32]: (data['direction'] == data['pos_ols_2']).value_counts()  ❶
Out[32]: True     1096
         False    1039
         dtype: int64

In [33]: data[['returns', 'strat_ols_1', 'strat_ols_2']].cumsum(
                 ).apply(np.exp).plot(figsize=(10, 6));
```

❶ Shows the number of correct and false predictions by the strategies.

Regression4

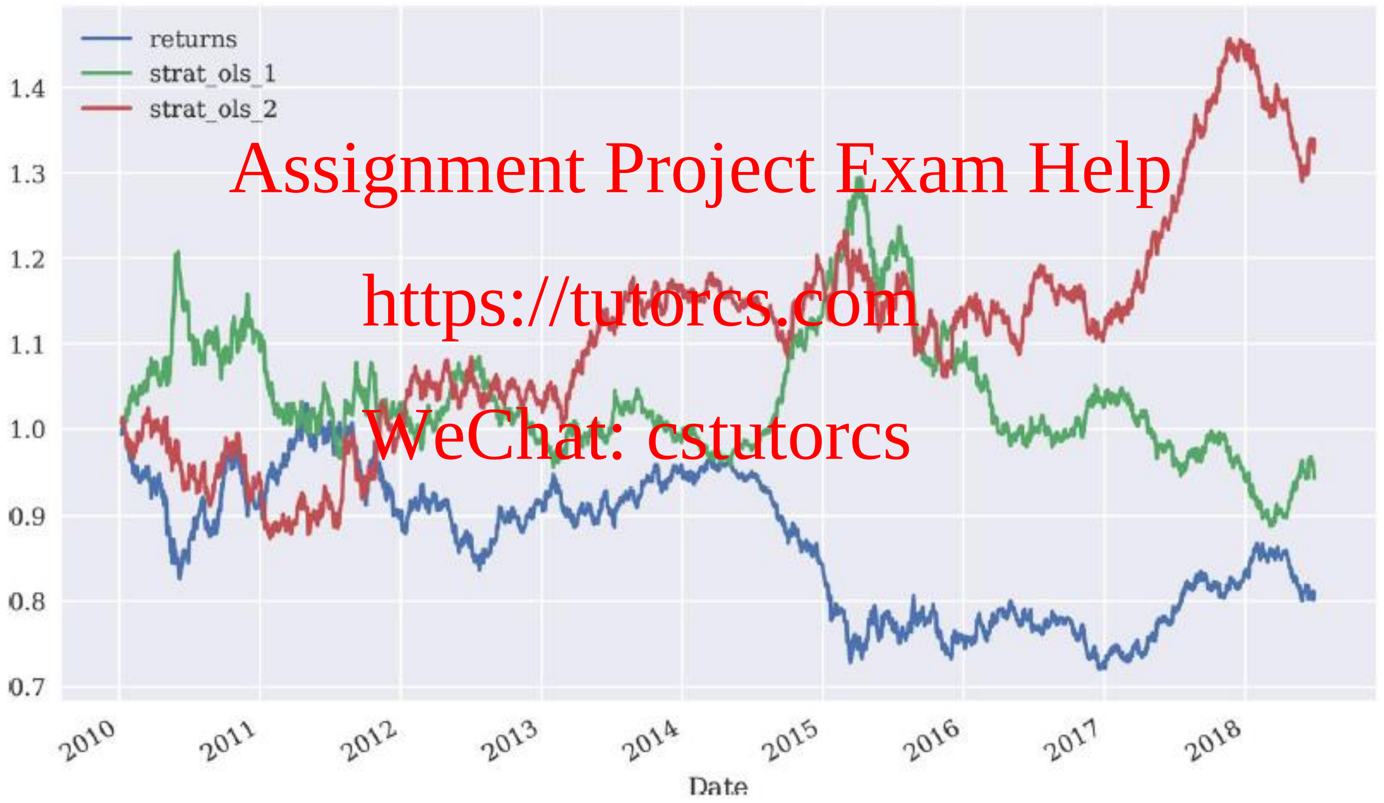


Figure 15-8. Performance of EUR/USD and regression-based strategies over time

Conclusion

- This chapter is about algorithmic trading strategies and judging their performance based on vectorized backtesting.
- It starts with a rather simple algorithmic trading strategy based on two simple moving averages, a type of strategy known and used in practice for decades.
- This strategy is used to illustrate vectorized backtesting, making heavy use of the vectorization capabilities of `NumPy` and `pandas` for data analysis.
- Using OLS regression, the chapter also illustrates the random walk hypothesis on the basis of a real financial time series.
- This is the benchmark against which any algorithmic trading strategy must prove its worth.

Further Resources

- The papers referenced in this chapter are:
- Brock, William, Josef Lakonishok, and Blake LeBaron (1992). “Simple Technical Trading Rules and the Stochastic Properties of Stock Returns.” *Journal of Finance*, Vol. 47, No. 5, pp. 1731–1764.
- Fama, Eugene (1965). “Random Walks in Stock Market Prices.” Selected Papers, No. 16, Graduate School of Business, University of Chicago.
- Jensen, Michael (1978). “Some Anomalous Evidence Regarding Market Efficiency.” *Journal of Financial Economics*, Vol. 6, No. 2/3, pp. 95–101.