trading_strategies

July 1, 2017

1 Trading strategy building and statistical testing

Written by: Zsolt Diveki, PhD

2 Content

3 1) Introduction

The intention of this document is to describe a few simple trading strategies and their application in python in order to give ideas to others how they could build and test strategies. It is not our intention to present a solid framework of trading machine that includes every aspect of algorithmic trading. Also we do not force to present profitable setups although we careful design probably they can be turned into profitable ones. Rather we would like to put the emphasis on the importance of statistical testing of the ideas.

We will present three trading strategies, moving average cross over trend, momentum trend and relative value pair trading. In each example we present trading rules, position sizing and optimization of some parameters with different types of statistical tests. Although each of these steps could be analysed and optimized separately, we only select one parameter to optimize in each strategy according to a predifined goodness of fit measure.

4 2) Getting data

Before starting building trading strategies we have to get data to work with. We will use inbuilt pandas methods to obtain data from finance.google.com.

4.1 2.1) Download prices with pandas

We import first some python libraries that will help our work:

```
In [1]: import matplotlib.pyplot as plt
        import pandas as pd
        import pandas.io.data as web
        import datetime as dt
        from datetime import timedelta
```

```
/usr/lib/python3/dist-packages/pandas/io/data.py:33: FutureWarning:
The pandas.io.data module is moved to a separate package (pandas-datareader) and will be remove
After installing the pandas-datareader package (https://github.com/pydata/pandas-datareader), 
  FutureWarning)
```

We define a helping function first. Based on an input date this function makes sure that the output is a week day. It requires a *datetime* type date as an input.

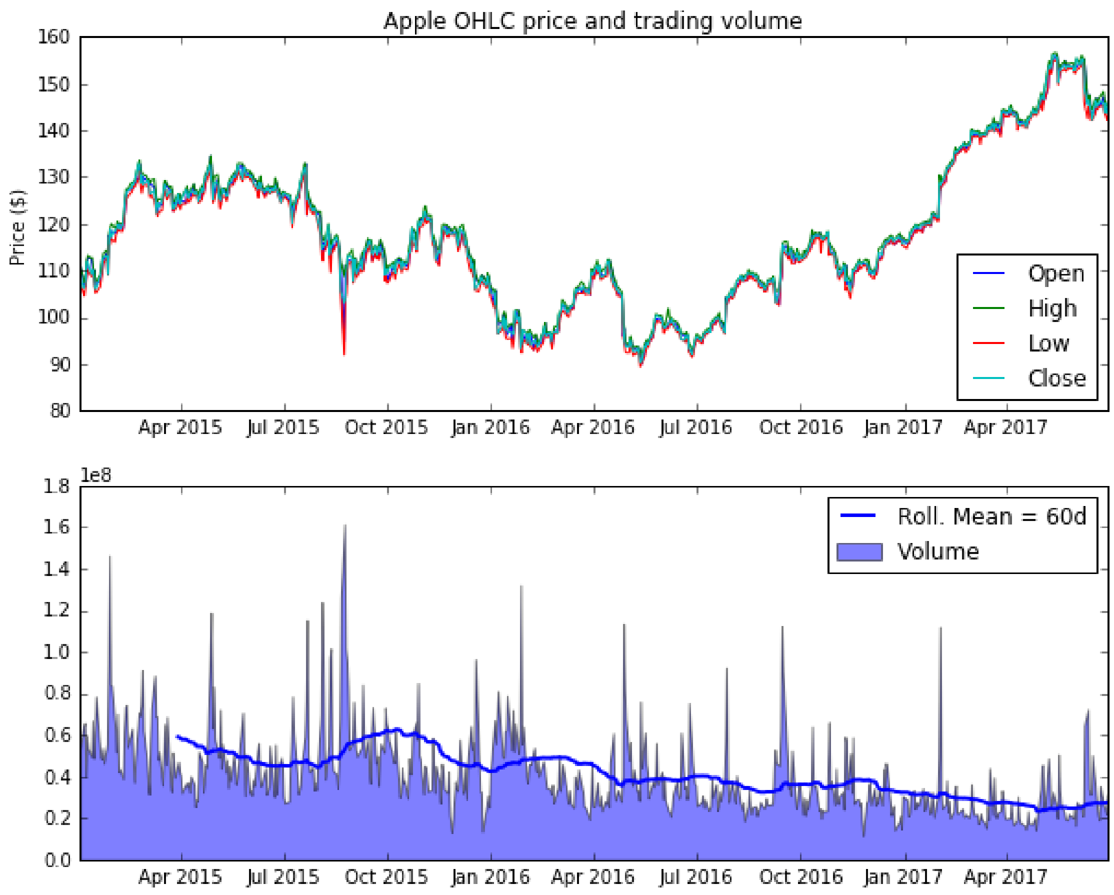
```
In [2]: def find_last_weekday(x):
            '''Finds the last weekday and returns it!
            x - datetime type date '''
            from datetime import datetime, timedelta   # import datetime libraries
            while datetime.weekday(x) in [5,6]:        # check if the date is a weekend
                x = x - timedelta(1)
            return(x)
```

Download Apple Inc. stock prices using *pandas DataReader* function:

```
In [41]: # setting a variable to the name of stocks we want to download: Apple
         stocks    =     'AAPL'
         # set the start date of the time series
         start     =     dt.datetime(1990, 1, 4)
         # set the end date as the last working day as of yesterday
         end = find_last_weekday(dt.datetime.today()-timedelta(1))
         f = web.DataReader(stocks, 'google', start, end)
```

Plot the OHLC prices in one plot and the volume below it:

```
In [42]: %matplotlib inline
         # make the figures appear in the notebook
         # select OHLC data starting from 2015
         top_plot = f[['Open', 'High', 'Low', 'Close']]['2015-01-01':]
         # storing volumes in a different variable
         bottom_plot = f[['Volume']]['2015-01-01':]
         # creating a figure with predifined size
         plt.figure(figsize=(10,8))
         # creating the upper plot for OHLC data
         plt.subplot(211)
         # plotting OHLC data one by one
         for name in top_plot.columns:
             plt.plot(top_plot[name], label = name)
         plt.legend(loc='best')                                                  # making legend
         plt.ylabel('Price ($)')                                                 # adding y-axis
         plt.title('Apple OHLC price and trading volume')                        # adding title
         plt.subplot(212)                                                        # preparing the
         plt.fill_between(bottom_plot.index, bottom_plot.Volume, \
                          where=bottom_plot.Volume<=bottom_plot.Volume, alpha=0.5, label='Volu
         # plotting the volume in a shaded area
         plt.plot(pd.rolling_mean(bottom_plot, window=60), label='Roll. Mean = 60d', lw=2, col
         # plotting rolling mean of the volume with a window size of 60 days
         plt.legend(loc='best')                                                  # adding legend
         plt.show()
```



One of the performance tests will be cross asset strategy testing, therefore we define a function that is able to return a dataframe with several columns containing the historical close prices of different stocks.

```
In [44]: def get_prices(names, start, end):
             '''Function for getting historical prices of several stocks
             Input:
             names - list of stock name strings
             start - datetime/date/string type of date of first price tick
             end   - datetime/date/string type of date of last price tick
             Output:
             pandas dataframe of historical stock prices
             '''
             # set the end date as the last working day as of yesterday
             end = find_last_weekday(end)
             # rename the close column to its stock name and create a
             # dataframe with columns of different stocks
             f = pd.DataFrame({name: web.DataReader(name, 'google', \
                                        start, end)['Close'] for name in names})
             # fills in missing values with previous days value
```

```
    f.fillna(method = 'ffill', inplace = True)
    return(f)
```

5 3) Building and testing trend strategies

In this chapter we create, optimize and test two trend strategies. The emphasize is rather on the optimization and statistical tests than on creating profitable strategies. The two strategies that will be presented here are:

1. Moving average cross over signal generation.
2. Momentum based trend strategy.

For simplicity we neglect slippage and trading costs. We will use kfold and bootstrap for performance optimization and testing.

5.1 3.1) Moving average cross over trend

This technique is based on the calculation of a fast and a slow simple moving average (MA) price series that will be used to predict the direction of the price movement. Lets look at how does two different MA look like for the Apple close prices obtained from Google. One must be careful with prices series obtained from Google because they might not well back adjusted for dividends and share splits therefore any strategy would create false signals. Since our objective is to show examples of how to create, optimize and test trading strategies and not to create trading ready algorithms, we do not go through data back adjustment.

Let us take the close values of Apple and calculate the 200 days and 50 days moving average of it and visualize it.

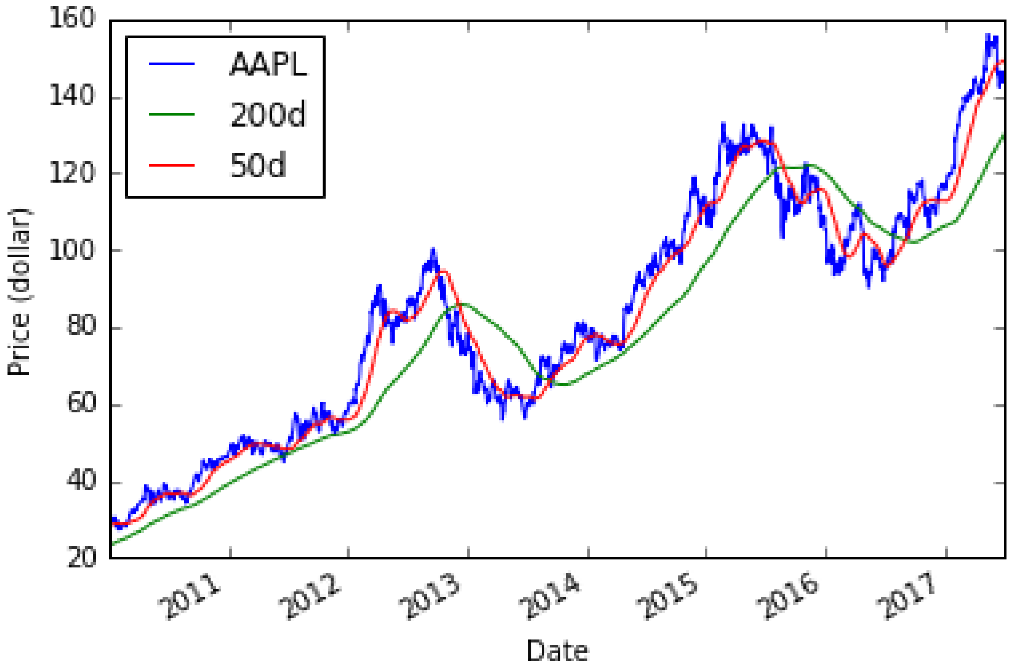
```
In [45]: names=['AAPL']
         start     =     dt.datetime(1990, 1, 4)
         end = find_last_weekday(dt.datetime.today()-timedelta(1))

         apple = get_prices(names, start, end)
         fast_ma = 50                           # lookback of the fast MA
         slow_ma = 200                          # lookback of the slow MA
         # calculation of the slow MA
         apple[str(slow_ma)+'d'] = pd.rolling_mean(apple['AAPL'], window = slow_ma)
         # calculation of the fast MA
         apple[str(fast_ma)+'d'] = pd.rolling_mean(apple['AAPL'], window = fast_ma)

         plt.figure(figsize = (10,7))
         apple['2010-01-01':].plot()
         plt.ylabel('Price (dollar)')
```

```
Out[45]: <matplotlib.text.Text at 0x7f9b6d0fff98>
```

```
<matplotlib.figure.Figure at 0x7f9b740decc0>
```



To determine the final performance of the strategies we have to calculate the returns as well. We determine first the daily logarithmic returns and then combine it with the positions to obtain the strategy profit and loss.

```
In [7]: apple['Ret'] = (apple['AAPL'] - apple['AAPL'].shift(1)) / apple['AAPL'].shift(1)
```

5.2 3.2) Trading rules

It is time to set up rules for trading based on the MA indicators:

- BUY: when fast MA > slow MA + delta
- SELL: when fast MA < slow MA - delta
- NEUTRAL: when (slow MA - delta) < fast MA < (slow MA + delta).

Our first step will be to calculate the difference between the fast and slow MAs. For simplicity we assume that we buy (or sell) only one share and keep them until the signal says so.

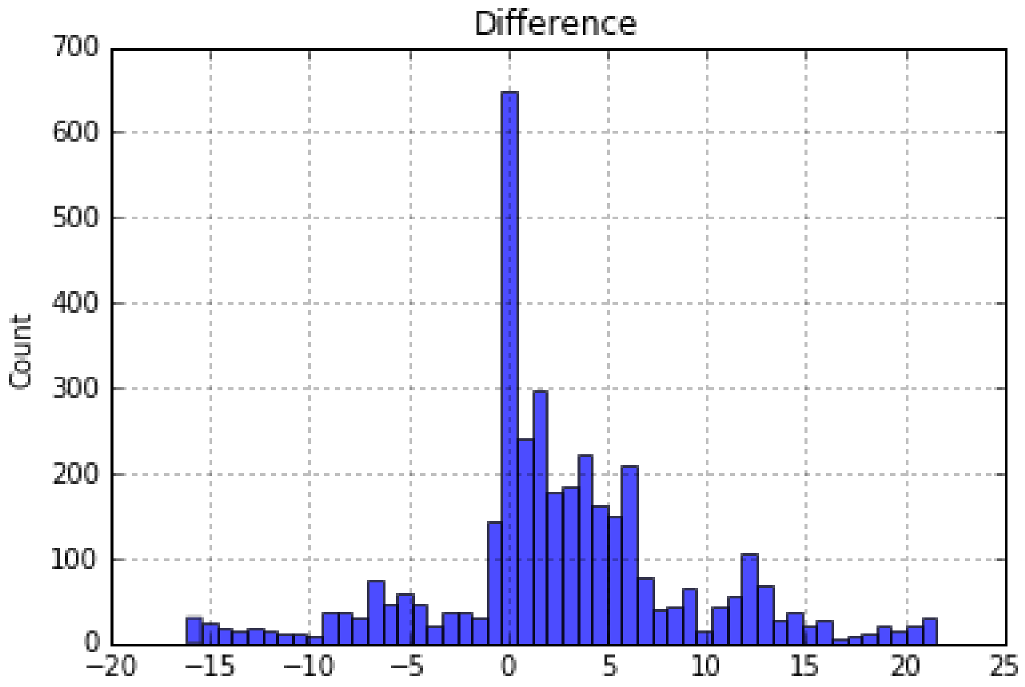
5.3 3.3) Optimization of delta

One of the crucial elements of this simplistic model is the value of *delta* (however we could optimize our trading for quite some other factors like the lookback period or the position sizing). We will apply an expending window cross validation technique to find the optimized *delta* value. We will use an initial 4 year period for the first train set and the consecutive 2 years for test set. Then we expand the training window by 1 year and test the second round again on the next 2 years and so on. We will run two different approaches to calculate *delta* and try to make a decision which one of them is performs better. We will use the information ratio as the goodness of fit measure of this optimization process (one could use many other things like mean return, volatility, skewness, hit ratio etc.).

To investigate the right range of delta we plot the histogram of the difference of the fast and slow MAs.

```
In [8]: apple['Difference'] = apple['50d']-apple['200d']
        apple.hist(column='Difference', bins=50, alpha=0.7)
        plt.ylabel('Count')

Out[8]: <matplotlib.text.Text at 0x7f9b7cd82e10>
```



```
In [33]: delta = np.arange(-15, 20, 1)                 # sampling range for delta
         apple.dropna(inplace=True)                     # remove NaNs from the dataframe
         # initialize time ranges that we will scan through for the training set
         train_dates = np.arange(2006,2016)
         # initialize time ranges that we will scan through for the test set
         test_dates = train_dates + 2
```

We can see that the difference ranges from -15 to 20. This will be our sampling range for *delta*. Also we have initialized time ranges that we scan for the training and testing set.

In the next step we define a few useful functions that will help us to keep our code concise: * *calculate_positions* - based on the difference of fastMA and slowMA and the delta we determine the signals to go short or long one share. * *calculate_pnl* - based on the previously obtained signals we calculate what pnl would we take home each day. * *information_ratio* - definition of the goodness of fit measure, the information ratio.

```
In [34]: def calculate_positions(df, delta):
             '''
             Calculate unit positions.
             Inputs:
                 df      - data frame of the difference of fastMA and slowMA
                 delta   - list of deviations from the differences.
             Output:
                 a       - data frame with unit positions for different deltas
             '''
             a=pd.DataFrame()
             for dlt in delta:
                 a[str(dlt)] = np.where(df['Difference'] > dlt, 1, 0)
                 a[str(dlt)] = np.where(df['Difference'] < -dlt, -1, a[str(dlt)])
             a.set_index(df.index, inplace=True)
             return(a)

         def calculate_pnl(df, ret):
             '''
             Calculate daily profit and loss based on the given positions.
             Inputs:
                 df      - data frame of the unit positions
                 ret     - series of daily returns calculated from daily prices
             Output:
                 df      - data frame with daily pnls
             '''
             for col in df.columns:
                 df[col] = df[col] * ret
             df.dropna(inplace = True)
             return(df)

         def information_ratio(pnl):
             '''
             Calculate information ratio.
             Inputs:
                 pnl     - series, np.arrays of daily pnls
             Output:
                 tmp     - annualized information ratio
             '''
             tmp = np.mean(pnl) * np.sqrt(252) / np.std(pnl)
             return(tmp)
```

When trading we can pocket the return of the nth day by having the position of the (n-1)th day. Therefore, we have to shift forward the signalled positions by one day to combine them with the right returns. This is done in the *calculate_pnl* function.

In the following lines we calculate the positions and pnls in advance for the whole history (with all possible deltas), so that we do not have to do it each time when rolling. This is acceptable when dealing with small data, but with high frequency multi year data we could not do it.

```
In [35]: pos = calculate_positions(apple, delta)
```

```
        pnl = calculate_pnl(pos, apple['Ret'])
```

Next we implement the rolling cross over optimization. We train over a 5 year window. We create variables to save the information ratios for the training and test sets and a variable to store the optimal deltas from the process. The optimal delta is calculated from the training set and immediately applied on the test set.

We will apply two different optimization processes: a) take the *delta* that provides the largest information ratio on the training set b) take the positive information ratios from the training set and average the corresponding *delta*s to obtain the optimized one.

a)

```
In [36]: ir_train = pd.DataFrame()             # variable to store information ratios from tra
         ir_test = pd.Series()                 # variable to store information ratios from tes
         dlts = pd.Series()                    # variable to store optimized deltas
         lookback = 5                          # number of years to look back in time
         for i in np.arange(len(train_dates)):
             # subset the training set
             train = pnl[str(train_dates[i]-lookback):str(train_dates[i])]
             # calculate the information ratio for each delta in the training set
             ir_train[str(train_dates[i]-lookback) + '-' + str(train_dates[i])] = \
                 train.apply(information_ratio, axis = 0)
             # choose the delta that belongs the largest information ratio
             dlts[str(train_dates[i]+1)+'-'+str(test_dates[i])] = \
                 float(ir_train.where(ir_train.ix[:,i]==ir_train.ix[:,i].max()).dropna().index
             # subset the test set
             test  = apple[str(train_dates[i]+1):str(test_dates[i])]
             # calculate the unit positions with test set and optimized delta
             pos_test = calculate_positions(test, [dlts.ix[i].tolist()])
             # calculate the corresponding pnl
             pnl_test = calculate_pnl(pos_test, apple['Ret'])
             # determine the information ratio on the test set
             ir_test[str(train_dates[i]+1) + '-' + str(test_dates[i])] = \
                 information_ratio(pnl_test).values[0]
```

Repeat the same procedure as above for the b) approach.

```
In [37]: ir_train_mean = pd.DataFrame()
         ir_test_mean = pd.Series()
         dlts_mean = pd.Series()
         for i in np.arange(len(train_dates)):
             train = pnl[str(train_dates[i]-lookback):str(train_dates[i])]
             ir_train_mean[str(train_dates[i]-lookback) + '-' + str(train_dates[i])] = \
                     train.apply(information_ratio, axis = 0)
             tmp = ir_train_mean.where(ir_train_mean.ix[:,i]>0).dropna().index
             dlts_mean[str(train_dates[i]+1)+'-'+str(test_dates[i])] = \
                     np.array([float(xx) for xx in tmp.tolist()]).mean()
             test  = apple[str(train_dates[i]+1):str(test_dates[i])]
             pos_test = calculate_positions(test, [dlts_mean.ix[i].tolist()])
```

```
            pnl_test = calculate_pnl(pos_test, apple['Ret'])
            ir_test_mean[str(train_dates[i]+1) + '-' + str(test_dates[i])] = \
                    information_ratio(pnl_test).values[0]
```

Let us print out the information ratios for a) and b) for all the test ranges. We also print out their mean and standard deviation.

```
In [38]: for i in range(ir_test_mean.shape[0]):
             print('Test range: %s IR(a): %.3f, IR(b): %.3f' % \
                   (ir_test.index[i], ir_test.ix[i], ir_test_mean.ix[i]))
         print('Mean(IR(a)): %.3f, Mean(IR(b)): %.3f' % (ir_test.mean(), ir_test_mean.mean()))
         print('Std(IR(a)): %.3f, Std(IR(a)): %.3f' % (ir_test.std(), ir_test_mean.std()))
```

```
Test range: 2007-2008 IR(a): 0.038, IR(b): 0.082
Test range: 2008-2009 IR(a): -0.777, IR(b): -1.037
Test range: 2009-2010 IR(a): 0.976, IR(b): 0.875
Test range: 2010-2011 IR(a): 1.368, IR(b): 1.020
Test range: 2011-2012 IR(a): 1.144, IR(b): 1.144
Test range: 2012-2013 IR(a): 1.388, IR(b): 1.141
Test range: 2013-2014 IR(a): 1.347, IR(b): 1.243
Test range: 2014-2015 IR(a): 0.642, IR(b): 0.564
Test range: 2015-2016 IR(a): -0.019, IR(b): -0.034
Test range: 2016-2017 IR(a): 0.769, IR(b): 0.807
Mean(IR(a)): 0.688, Mean(IR(b)): 0.581
Std(IR(a)): 0.727, Std(IR(a)): 0.718
```

Our job is now to make a decision which approach, a) or b), is more profitable to us. From the previously printed statistics we can see that the results are very close to each other. With regard to the fact that we do not have many samples (less than 30) we will have difficulty to make a confident decision. Nevertheless we will present a approach to do that.

We will assume that the samples of both a) and b) are coming from a normal distribution. This is a very weak statement since we do not have many samples, but we rely on the central limit theory, which states that given a sufficiently large sample size the distribution of the samples will follow a normal distribution.

We can use the t-test to decide if the two samples are coming from the same normal population, based on their mean value. If they are coming from the same population, in ideal case they should have the same mean and same variance. In python the *scipy.stats* package has a function to calculate t-statistics and p-value. Our null hypothesis is that the two samples are coming from the same population.

```
In [40]: import scipy.stats as stats
         t_stat, p_val = stats.ttest_ind(ir_test, ir_test_mean, equal_var=True)
         print('t-statistic: %.5f, p-value = %.5f' % (t_stat, p_val))
```

```
t-statistic: 0.33104, p-value = 0.74444
```

From the p-value we know that there is 74% chance to observe these mean values in two samples assuming they are from the same population. This is a high probabilty therefore we cannot confidentely say (with 95% or more) that one of the approaches would perform better than the other.

6 4) Building and testing momentum strategies

7 5) Building and testing pair traiding strategies

```
In [ ]:
```