trading_strategies

July 8, 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), y
  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.

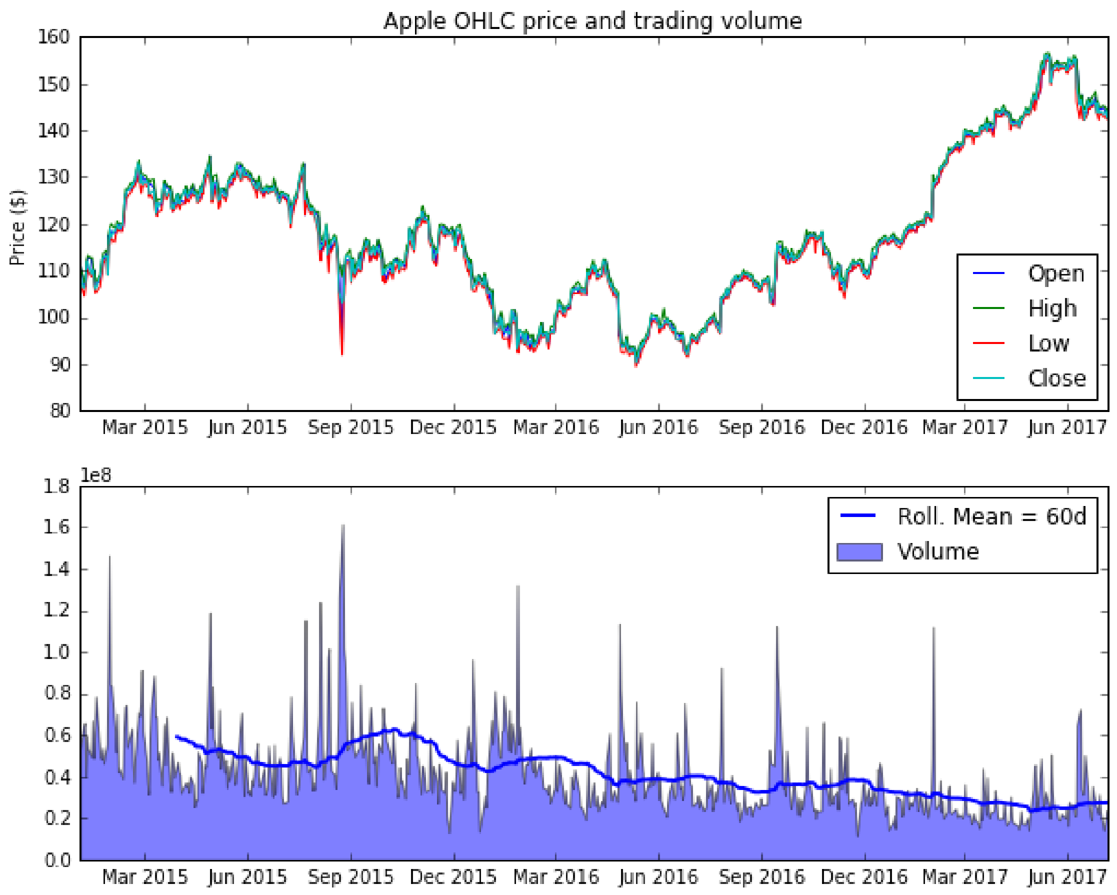
```
In [2]: def find_last_weekday(x):
            '''Finds the last weekday and returns it!
            x - datetime type date '''
            from datetime import datetime, timedelta   # import datetime libraries
            while datetime.weekday(x) in [5,6]:        # check if the date is a weekend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

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

```
In [3]: # 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 [4]: %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='Volum
        # plotting the volume in a shaded area
        plt.plot(pd.rolling_mean(bottom_plot, window=60), label='Roll. Mean = 60d', lw=2, colo
        # 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 [5]: 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

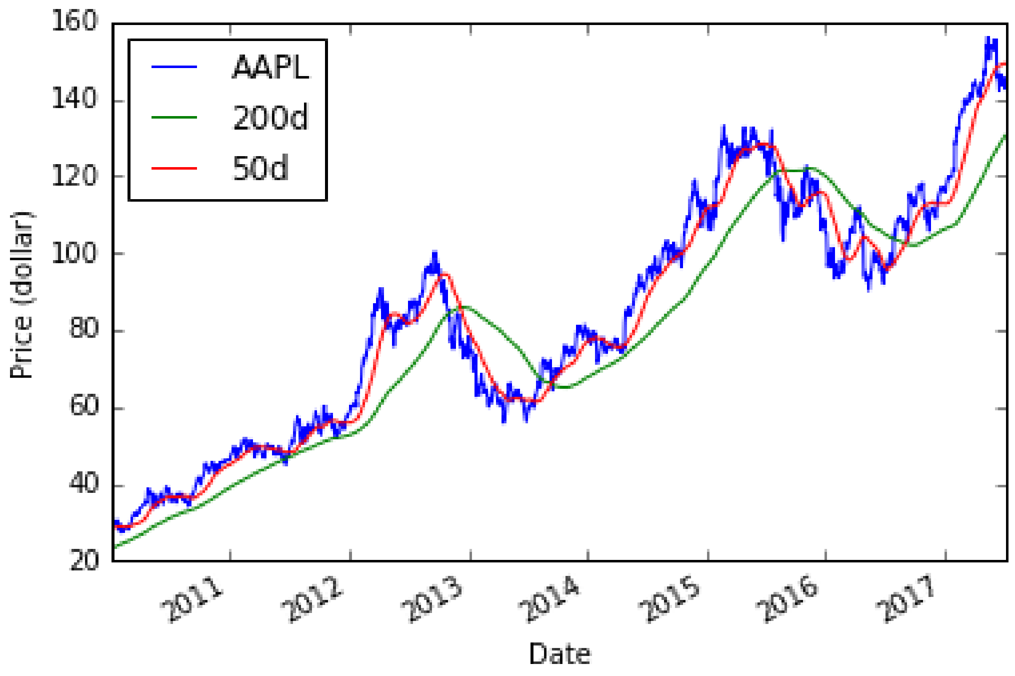
This technique is based on the calculation of a fast and a slow simple moving average (MA) price series that will be used to predict the direction of the price movement. Lets look at how does two different MA look like for the Apple close prices obtained from Google. One must be careful with prices series 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 [6]: 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[6]: <matplotlib.text.Text at 0x7fa0dfa454e0>
```

```
<matplotlib.figure.Figure at 0x7fa0f2f9b9e8>
```



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.

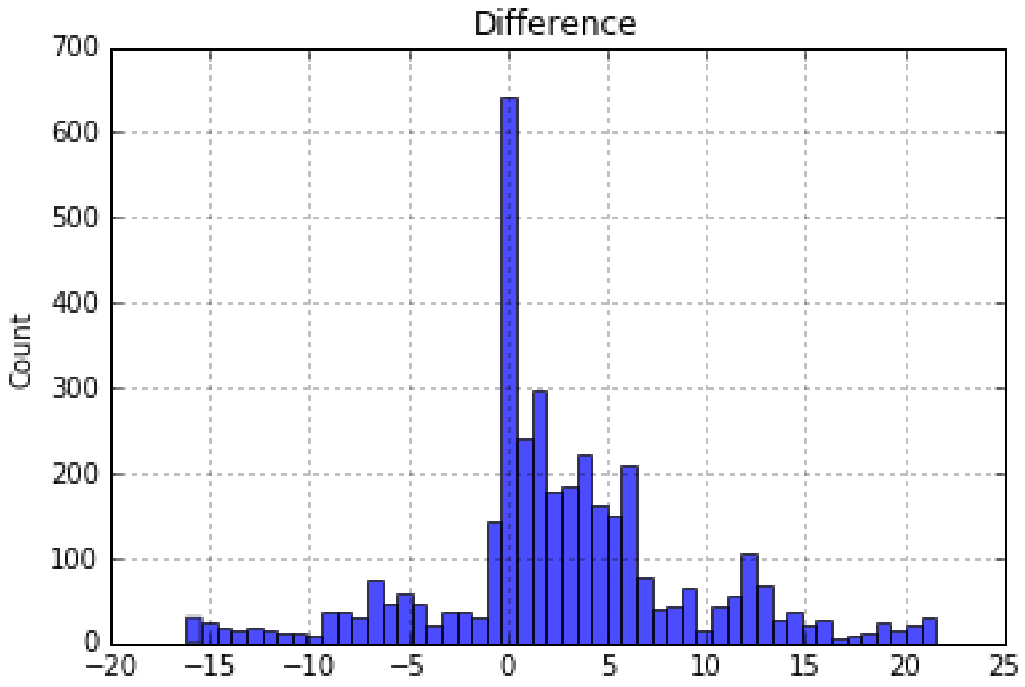
5.3 3.3) Optimization of delta

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 0x7fa0dfbe6f28>
```



```
In [9]: 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 [10]: 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:
                 tmp     - data frame with daily pnls
             '''
             tmp = pd.DataFrame()
             for col in df.columns:
                 tmp[col] = df[col] * ret
             tmp.dropna(inplace = True)
             return(tmp)

         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 [11]: pos = calculate_positions(apple, delta)
         pos = pos.shift(1).dropna()
         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 [12]: 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()])
             pos_test = pos_test.shift(1).dropna()
             # 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 [13]: 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()])
             pos_test = pos_test.shift(1).dropna()
             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 [14]: 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): -0.049, IR(b): -0.045
Test range: 2008-2009 IR(a): -0.781, IR(b): -1.105
Test range: 2009-2010 IR(a): 1.222, IR(b): 1.003
Test range: 2010-2011 IR(a): 1.341, IR(b): 1.056
Test range: 2011-2012 IR(a): 1.009, IR(b): 1.009
Test range: 2012-2013 IR(a): 1.067, IR(b): 0.981
Test range: 2013-2014 IR(a): 1.335, IR(b): 1.270
Test range: 2014-2015 IR(a): 0.790, IR(b): 0.831
Test range: 2015-2016 IR(a): 0.177, IR(b): 0.177
Test range: 2016-2017 IR(a): 0.807, IR(b): 0.807
Mean(IR(a)): 0.692, Mean(IR(b)): 0.598
Std(IR(a)): 0.696, Std(IR(a)): 0.725
```

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 [15]: 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: 0.29376, p-value = 0.77231
```

From the p-value we know that there is 77% 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

In the followings we will describe briefly what is a momentum trend strategy, present an example of creating buy and sell signals from it and finally use bootstrap to optimize the lookback period to obtain the best performance from a given input.

An n-day momentum is usually defined as the difference between the price at time *t* and the price at time *t-n*. The momentum obtained this way will be used as a forecaster of tomorrow's return. We define several lookback periods and try to find the most optimal one to achieve a relatively high and stable performance. We choose the *edge* as our goodness of fit, which is not directly a performance measure like the information ratio.

Our very first task is to get some data. We will work with stock prices of Amazon. We reuse the methods introduced in Chapter 2) to retrieve the data:

```
In [16]: # define the name of the selected stock
         names=['AMZN']
         # start date of the price series
         start    =     dt.datetime(1990, 1, 4)
         # end date
         end = find_last_weekday(dt.datetime.today()-timedelta(1))
         # get the close price quotes of the selected stock
         stock = get_prices(names, start, end)
         # calculate the daily percentage return
         stock['Ret'] = stock.pct_change()
```

6.1 4.1) Trading rules and position sizing

Let us define 5, 20, 65, 130 and 260 days lookback period. Actually one can choose any positive integer she wants and test it. Our choice represents the one week, one month, one quarter, half and one year periods. These are in general important time ranges in finance that many trading participant may consider. This common behaviour can be one of the reasons behind the existence of trend. If people believe that in 20 days the prices are going up, they will buy and naturally fuel the rise of the prices.

Next we calculate the n-day momenta and collect the results in a dataframe over the whole time range. These numbers will serve as the predictor of tomorrows daily return.

```
In [17]: # definition of lookbacks
         lookback = np.array([ 5, 20, 65, 130, 260])
         # copying the original data for future dataframe extension
         train = stock[names]
         # calculating the momentum for each lookback
```

```
        for lkb in lookback:
            # momentum
            tmp = stock[names].diff(lkb)
            # renaming the column to the lookback period
            tmp.columns = [str(lkb)]
            # extending the price and returns with momenta
            train=pd.concat([train, tmp], axis=1)
```

Next we create a function that defines the trading rules. Our trading strategy for a given momentum is the following: * If momentum > 0 -> go long * If momentum < 0 -> go short * If momentum = 0 -> go/stay neutral

At this stage one could also think about position sizing strategies. For simplicity, we stick to go one share long or one share short.

```
In [18]: def calculate_positions_momentum(df, delta):
             '''
             Calculate unit positions.
             Inputs:
                 df      - data frame of the momenta
                 delta   - list of lookback periods
             Output:
                 a       - data frame with unit positions for different deltas
             '''
             # create container dataframe
             a=pd.DataFrame()
             # loop over the lookback periods
             for dlt in delta:
                 # create long positions
                 a[str(dlt)] = np.where(df[str(dlt)] > 0, 1, 0)
                 # create short positions
                 a[str(dlt)] = np.where(df[str(dlt)] < 0, -1, a[str(dlt)])
                 # create neutral positions (this line is not necessary)
                 a[str(dlt)] = np.where(df[str(dlt)] == 0, 0, a[str(dlt)])
             # fill the dataframe with date index
             a.set_index(df.index, inplace=True)
             return(a)
```

6.2 4.2) Optimization process

After determining the trading rules we divide our data into a training and a testing set. The training set will cover the data up to and including 2012. With the training set prices we calculate the momenta, the daily unit positions and finally the daily pnl according to our trading strategy.

```
In [19]: # calculate daily unit positions
         pos_train = calculate_positions_momentum(train[:'2012'].dropna(), lookback)
         # lag the dates to use those positions with tomorrows returns
         pos_train = pos_train.shift(1).dropna()
         # calculate daily pnl
         pnl_train = calculate_pnl(pos_train, stock['Ret'])
```

To determine what is the ideal lookback period to use with the Amazon data we will introduce a new goodness of fit, the *edge*. The edge can be derived from the formula of the expected return with some arrangment of terms. The outcome is:

$edge = \frac{1}{2}(hr * (1 + wl))$, where:
hr - hit ratio, days with positive returns divided with the number of total days
$wl = \frac{E[pnl>0]}{|E[pnl<0]|}$, the ratio between the expected positive and absolute value of negative daily returns.

This measure helps to decide whether the strategy does better or worse than a random position taker: * edge > 0.5 -> the strategy is expected to make money on the longterm * edge < 0.5 -> the strategy is expected to loose money on the longterm * edge = 0.5 -> corresponds to the random position taker. Normally we would expect to loose money on the longterm at least because of the cost of trading and slippages.

We partially test the validity of these statments by determining what hr and wl corresponds to the edge value of 0.5 (random outcome). There can be several combinations for that scenario but one of them is when $hr = 0.5$ and $wl = 1$. The former means that this strategy has 50% chance to get the sign of tomorrow's return right, while the latter means that the expected positive return is equal to the expected loss in absolute sense. This scenario basically corresponds to a strategy where our postion is determined by the flip of a coin.

In order to decide which lookback period is supposed to give the optimal edge we use bootstrap. In bootstrap we assume that the observed pnl sample is a good approximation of the population of pnls and by resampling it with replacement we can calculate different statistics of the edge. That will help us to observe the distribution of the possible edges and come up with a procedure that will show us which edge distribution of the lookback periods has the largest mean and smallest dispersion. This choice would mean that the reproduction of the same edge is very frequent therefore the strategy is stable in the performance.

In the followings we define the hit ratio, win loss, the edge and the bootstrap function:

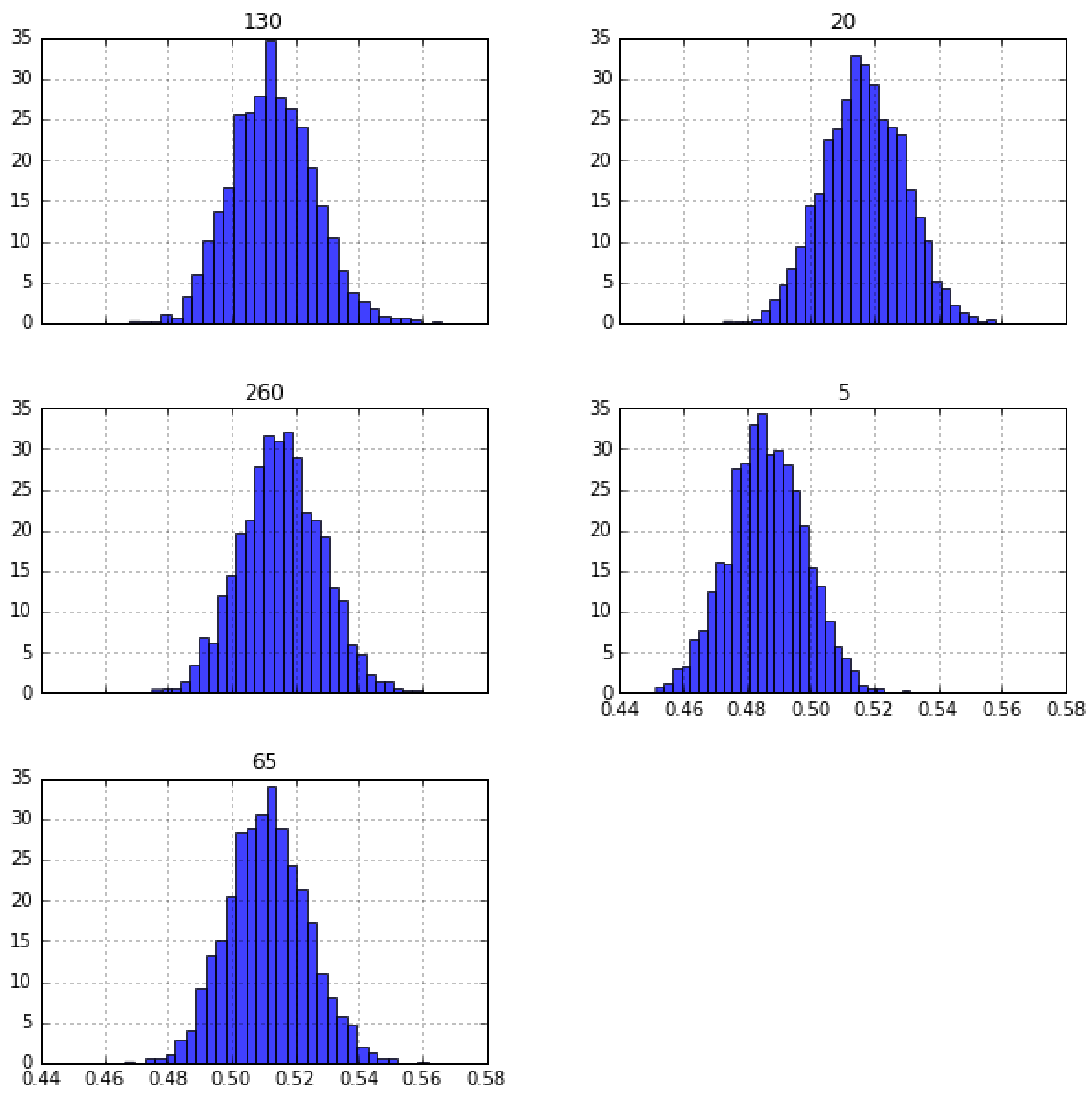
```
In [20]: def hit_ratio(pnl):
             tmp = pnl[pnl > 0].shape[0] / pnl.shape[0]
             return(tmp)

         def win_loss_ratio(pnl):
             tmp = - pnl[pnl > 0].mean() / pnl[pnl <= 0].mean()
             return(tmp)

         def edge(pnl):
             hr = hit_ratio(pnl)
             wlr = win_loss_ratio(pnl)
             tmp = 0.5 * (hr * (1 + wlr))
             return(tmp)

         def bootstrap_with_replacement(pnl, experiments = 10, samples=len(pnl)):
             '''
             Returns experiment number of resamples with replacement.
             Inputs:
             pnl         - np.array/pd.series of pnls
             experiments - number of times to resample the pnl array
             samples     - number of draws in a sample
```

```
    Output:
    tmp         - np.ndarray (matrix) of size experiments * samples
    '''
    # generate random integer matrix of experiments * samples as the index of pnl
    inds = np.random.randint(0, len(pnl), (experiments, samples))
    # for each column of pnl resampled with inds, calculate edge
    tmp = np.apply_along_axis(edge, 1, pnl[inds])
    return(tmp)
```

Now we are ready to apply bootstrap to the observed pnl samples and create a dataframe with the distribution of probable edges. To have an idea which lookback period may result in the most stable (and hopefully profitable) performance we visually inspect the histogram of edges.

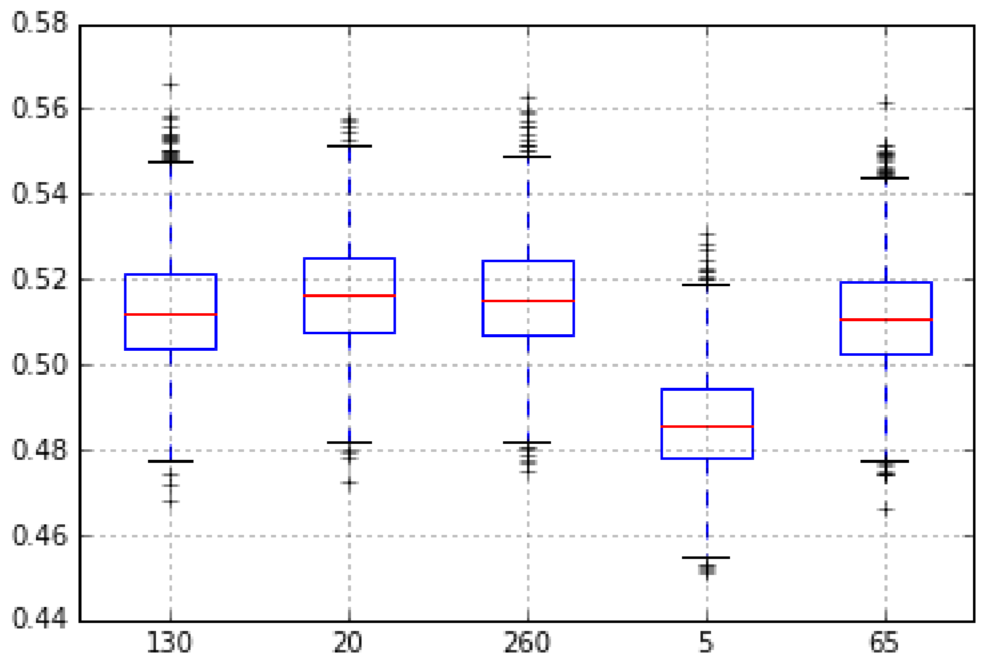
```
In [21]: # create a dataframe of edges of 3000 experiments over each lookback period
         train_bts=pd.DataFrame({col: bootstrap_with_replacement(pnl_train[col], experiments=3(
         # plot a histogram of the probable edges
         train_bts.hist(bins=30, figsize=(10,10), normed=1, alpha=0.75, sharex=True)
```

```
Out[21]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fa0d44566d8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fa0d40a1588>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x7fa0d406f080>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fa0d402ab38>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x7fa0cffdb518>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fa0cff95470>]], dtype=obj(
```



Even with visual inspection we can tell that the 5 days lookback strategy is most likely to loose money, since the most likely edge is smaller than 0.5. The rest of them are pretty similar, therefore we might need a different way of visualisation. Therefore we plot them on a boxplot that helps us see where is the median (horizontal red line) what is the interquartile range (top and bottom edge of the boxes), a constant multiple of interquartile range (horizontal black lines) and the outliers:

```
In [22]: lines = train_bts.boxplot(return_type='dict')
```



This representation helps us to see that the mean of 20 days lookback is the highest and the observed interquartile ranges are roughly the same. There are several ways to measure how centralized a distribution is (for example with calculating the kurtosis) but we implement a measure that is higher if the mean is high, the standard deviation and the interquartile range is small. Then we sort the value of this measure and see which lookback gives the highest number to decide which lookback might give the most stable performance:

```
In [38]: av = train_bts.mean() - train_bts.std() - (train_bts.quantile(0.75) - train_bts.quant
         av.sort_values()

Out[38]: 5      0.457545
         65     0.481895
         130    0.482261
         260    0.485666
         20     0.486340
         dtype: float64
```

20 days seems to be the right choice. However the values of these measure are not too different, so just for sanity check we make sure that these distributions are coming from different (normal) populations. We do this comparison with the t-test:

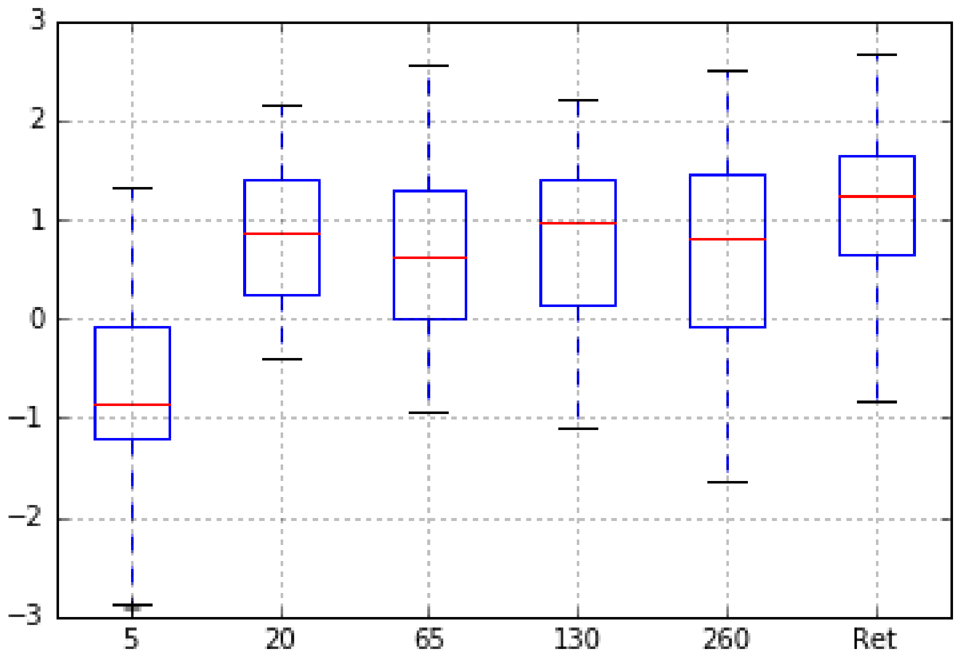
```
In [29]: for col in train_bts.columns:
             t_stat, p_val = stats.ttest_ind(train_bts[col], train_bts['20'], equal_var=True)
             print('%s lookback: t-statistic: %.5f, p-value = %.5f' % (col, t_stat, p_val))

130 lookback: t-statistic: -11.55142, p-value = 0.00000
20 lookback: t-statistic: 0.00000, p-value = 1.00000
```

```
260 lookback: t-statistic: -2.95668, p-value = 0.00312
5 lookback: t-statistic: -95.89676, p-value = 0.00000
65 lookback: t-statistic: -16.52688, p-value = 0.00000
```

Since the p-value is really small, we can state that we are more than 95% confident that these are different distribution when the difference of their mean values are equal to the observed ones.

Now that we know that the 20 day lookback should give a stable and good performance we can test this idea out of sample. We left out from these calculation the data after 2012. In the following block we compare what information ratios would the different lookbacks give. We also include a simple buy and hold strategy (with column nem *Ret*).

```
In [42]: # calculate the unit positions for the out of sample data (starting from 2013)
         pos_test = calculate_positions_momentum(train['2013':].dropna(), lookback)
         # shift the positions by one day to calculate pnl
         pos_test = pos_test.shift(1).dropna()
         # calculate out of sample daily pnl
         pnl_test = calculate_pnl(pos_test, stock['Ret'])
         # add the buy and hold strategy daily returns
         pnl_test=pd.concat([pnl_test, stock[['Ret']]], axis=1).dropna()
         # calculate a one year information ratio on a daily rolling basis
         irs=pd.rolling_apply(pnl_test, 260, information_ratio)
         # boxplot of the information ratios
         ll=irs.boxplot(return_type='dict')
```

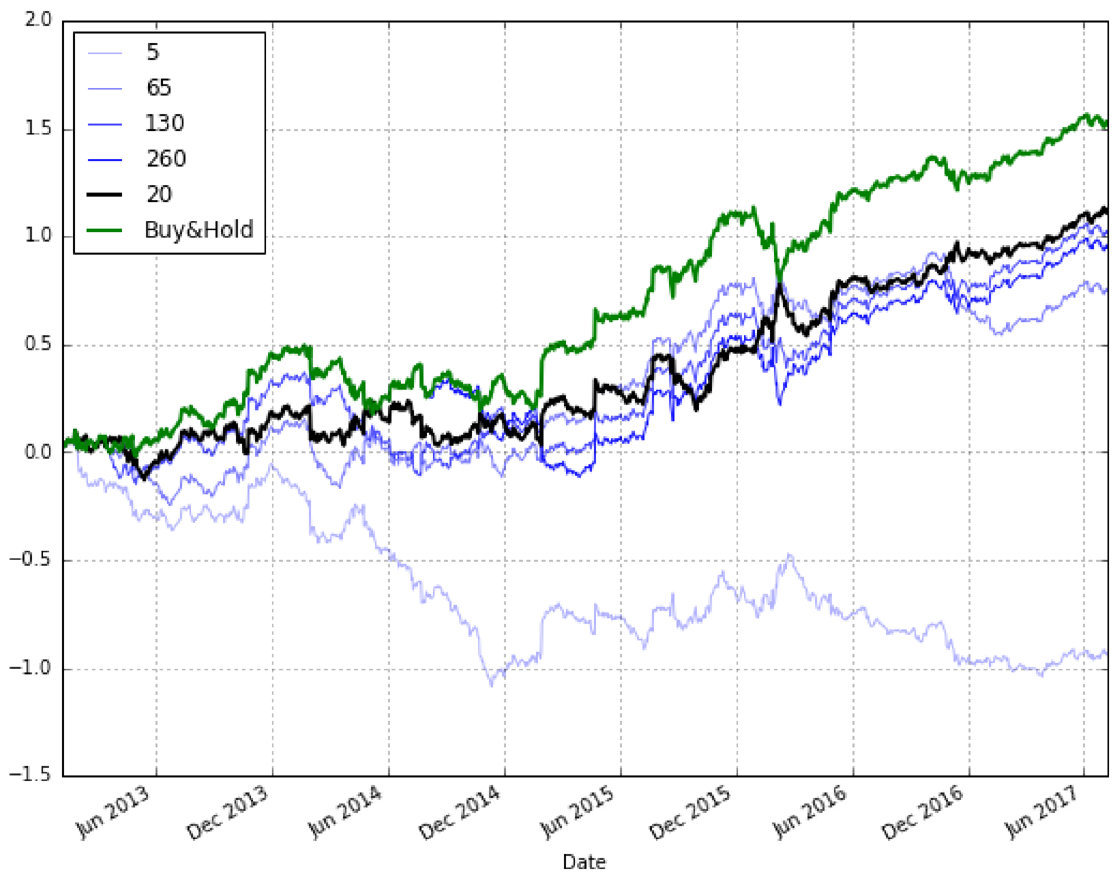


As we were expecting the 20 day lookback outperformes the other strategies (except for the buy and hold). Important to notice that 20 days lookback has the smallest range of whiskers too, meaning that this approach indeed gives a stable performance. Interestingly 130 day lookback looks to be more stable than the others (except for 20 days) and its median outperforms the median of 260 days.

The most interesting thing is that the buy and hold (Ret) approach seems to work the best between these strategies. Its median is above 1, its interquartile range is smaller than for the others, meaning that the performance is peaked around the center.

Finally let us plot the cumulative return applying these 6 strategies. Buy and hold largely outperforms all the others.

```
In [43]: # definition of transparency
         alphas = np.array([0.3, 0.5, 0.7, 0.9])
         # initialize variable for color storage
         rgba_colors = np.zeros((4,4))
         # set color to blue
         rgba_colors[:,2] = 1
         # set the transparency
         rgba_colors[:,3] = alphas
         # plot all strategies except buy and hold and 20 days
         pnl_test[['5','65', '130', '260']].cumsum().plot(color = rgba_colors, figsize=(10,8))
         # plot 20 days lookback cumulative pnl
         plt.plot(pnl_test[['20']].cumsum(), color = 'black', linewidth = 2, label='20')
         # plot the buy and hold cumulative pnl
         plt.plot(pnl_test[['Ret']].cumsum(), color = 'green', linewidth = 2, label='Buy&Hold'
         # create a guiding grid
         plt.grid()
         # add labels to legend
         plt.legend(loc='best')

Out[43]: <matplotlib.legend.Legend at 0x7fa0cf5e1f28>
```



7 5) Building and testing pair traiding strategies

```
In [58]:
```