Financial Data Statistics

October 6, 2021

[4]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

[5]:
```
data = pd.read_csv('all_stocks_5yr.csv', parse_dates=True)
```

[6]:
```
data.head()
```

[6]:

	date	open	high	low	close	volume	Name
0	2013-02-08	15.07	15.12	14.63	14.75	8407500	AAL
1	2013-02-11	14.89	15.01	14.26	14.46	8882000	AAL
2	2013-02-12	14.45	14.51	14.10	14.27	8126000	AAL
3	2013-02-13	14.30	14.94	14.25	14.66	10259500	AAL
4	2013-02-14	14.94	14.96	13.16	13.99	31879900	AAL

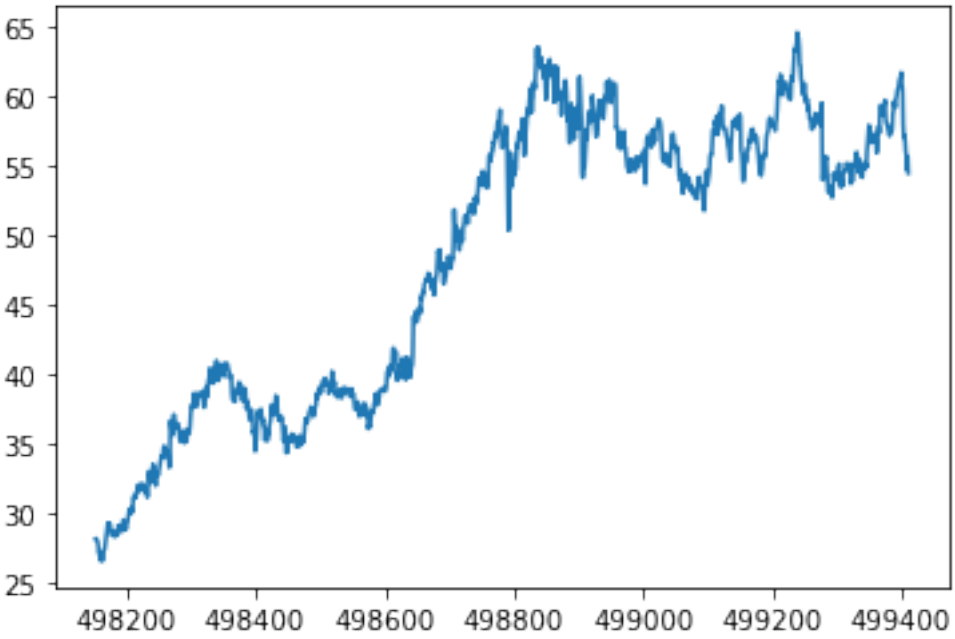
[10]:
```
'''
Next, we're going to extract a new data frame by filtering all the rows where␣
↪the name is SBUX.
'''

sbux=data[data['Name']=='SBUX'].copy()
sbux.head()
```

[10]:

	date	open	high	low	close	volume	Name
498152	2013-02-08	27.920	28.325	27.920	28.185	7146296	SBUX
498153	2013-02-11	28.260	28.260	27.930	28.070	5457354	SBUX
498154	2013-02-12	28.000	28.275	27.975	28.130	8665592	SBUX
498155	2013-02-13	28.230	28.230	27.750	27.915	7022056	SBUX
498156	2013-02-14	27.765	27.905	27.675	27.775	8899188	SBUX

[12]:
```
'''
Next, we can call the plot function on the clothes column to look at the stock␣
↪price as a time series.
'''
sbux['close'].plot()
```

[12]:
```
<AxesSubplot:>
```



[13]:
```
'''
Now, to get down to the real work, we know that in order to calculate the␣
 ↪return, we need the current
close price as well as the previous close price.

What we want to have is the previous close price in the same row as the close␣
 ↪price to do this, we
can call the shift function, we pass on the value one.

To say that we want to shift the close call by one will assign this new column␣
 ↪to be priv close.

If we use the head, come in again, we see that our new column of clothes has␣
 ↪been created.

Notice how all the items in the clothes column are just the items from the␣
 ↪clothes column shifted up
by one.
Since there is nothing to shift up, buy one for the very first value. This␣
 ↪value must necessarily be NaN.
'''

sbux['prev_close']=sbux['close'].shift(1)
sbux.head()
```

[13]:

	date	open	high	low	close	volume	Name	prev_close
498152	2013-02-08	27.920	28.325	27.920	28.185	7146296	SBUX	NaN
498153	2013-02-11	28.260	28.260	27.930	28.070	5457354	SBUX	28.185
498154	2013-02-12	28.000	28.275	27.975	28.130	8665592	SBUX	28.070
498155	2013-02-13	28.230	28.230	27.750	27.915	7022056	SBUX	28.130
498156	2013-02-14	27.765	27.905	27.675	27.775	8899188	SBUX	27.915

[29]:
```
'''
In the next block of code, we calculate the return as discussed previously.

That's the close column divided by the previous column and then minus one will␣
 ↪assign this value to
a column called Return.
'''
from IPython.display import display, Math, Latex
display(Math(r'R = {P_t \over P_{t-1}}-1'))
```

$$R = \frac{P_t}{P_{t-1}} - 1$$

[30]:
```
%%latex
\begin{align}
\newline
R= {{P_{t}\over P_{t-1}} -1}
\end{align}
```

$$R = \frac{P_t}{P_{t-1}} - 1 \tag{1}$$

[35]:
```
from IPython.display import Latex
Latex(r"""\begin{eqnarray} R= {{P_{t}\over P_{t-1}} -1}
\end{eqnarray}""")
```

[35]:

$$R = \frac{P_t}{P_{t-1}} - 1 \tag{2}$$

[25]:
```
'''
Note the use of vectorized operations here.
There's no need to do something like a for loop through each row, calculating␣
 ↪the return one by one.

If you're trained in programming, that's probably your first thought for how to␣
 ↪calculate the return,
given a two dimensional array of items.
```

```
But luckily, pandas make this operation very easy by allowing us to calculate␣
 ↪all of the returns at once.

If we do a sbux.head() again to see what our new data frame looks like, we see␣
 ↪our return.

Collum note again how the first row is NaN This must be the case since we've␣
 ↪prev_close is NaN and therefore it's not possible to calculate
the return on this date.

Also, notice how small the returns are, as mentioned, financial engineers are␣
 ↪pretty accustomed to
working with very small numbers like this, and that's why we use units such as␣
 ↪basis points.
'''
sbux['return']=sbux['close']/sbux['prev_close'] -1
sbux.head()
```

[25]:

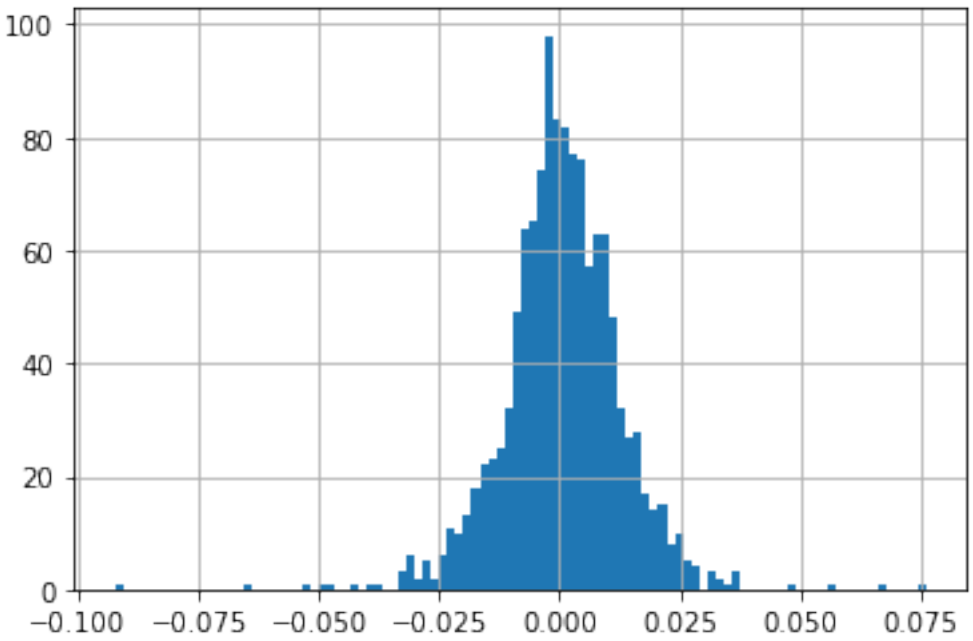
	date	open	high	low	close	volume	Name	prev_close	return
498152	2013-02-08	27.920	28.325	27.920	28.185	7146296	SBUX	NaN	NaN
498153	2013-02-11	28.260	28.260	27.930	28.070	5457354	SBUX	28.185	-0.004080
498154	2013-02-12	28.000	28.275	27.975	28.130	8665592	SBUX	28.070	0.002138
498155	2013-02-13	28.230	28.230	27.750	27.915	7022056	SBUX	28.130	-0.007643
498156	2013-02-14	27.765	27.905	27.675	27.775	8899188	SBUX	27.915	-0.005015

[36]:
```
'''
The way to do this is we call the pct_change function.
We pass in the argument one to mean that we want to calculate the percent␣
 ↪change over one timestep.

Upon inspection, we see that both the return and return2 to columns are the␣
 ↪same, verifying that our
calculation of the return is correct.
'''
sbux['return2']=sbux['close'].pct_change(1)
sbux.head()
```

[36]:
```
             date    open    high     low   close   volume  Name  prev_close  \
498152  2013-02-08  27.920  28.325  27.920  28.185  7146296  SBUX         NaN
498153  2013-02-11  28.260  28.260  27.930  28.070  5457354  SBUX      28.185
498154  2013-02-12  28.000  28.275  27.975  28.130  8665592  SBUX      28.070
498155  2013-02-13  28.230  28.230  27.750  27.915  7022056  SBUX      28.130
498156  2013-02-14  27.765  27.905  27.675  27.775  8899188  SBUX      27.915

          return   return2
498152       NaN       NaN
498153 -0.004080 -0.004080
498154  0.002138  0.002138
498155 -0.007643 -0.007643
498156 -0.005015 -0.005015
```

[38]:
```
'''
Although up until now, we've been plotting Time series, what we are often␣
 ↪interested in when we look
at returns is the distribution of returns.

One quick visualization we can do to get a feel for the distribution of a set␣
 ↪of numbers is the histogram in pendas.

All we need to do is call the highest hist on our data frame or series object.
We pass in the bins argument to specify how fine grained we want the histogram␣
 ↪to be.

So as you can see, what we get is this typical bell shaped curve.
'''

sbux['return'].hist(bins=100)
```

[38]:
```
<AxesSubplot:>
```



[40]:
```
'''
Another thing we can do with our series of returns is calculate statistics such␣
 ↪as the sample mean and
the sample variance.

The STD function actually gives us the standard deviation, which is the square␣
 ↪root of the variance.

As you can see, the return is very small, very close to zero, and the standard␣
 ↪deviation is also
quite small, about zero point zero one.

'''

sbux['return'].mean(),sbux['return'].std()
```

[40]:
```
(0.0006002332205830914, 0.012360934026133882)
```

[41]:
```
'''
Now, since we learned about log returns as well, let's try doing the same␣
 ↪process, but on the log
returns, luckily numpy operations, broadcast over pandas data structures.

So all we need to do is take the return coloumn at once and called log on that.
```

```
you recall that when X is very small, it's approximately equal to log(X+1), and␣
 ↪we can see that
kind of behavior here.

Notice how the log returns are actually very close to the non log returns.
They only differ in about the last two decimal places.
'''

sbux['log_return']=np.log(sbux['return']+1)
sbux.head()
```

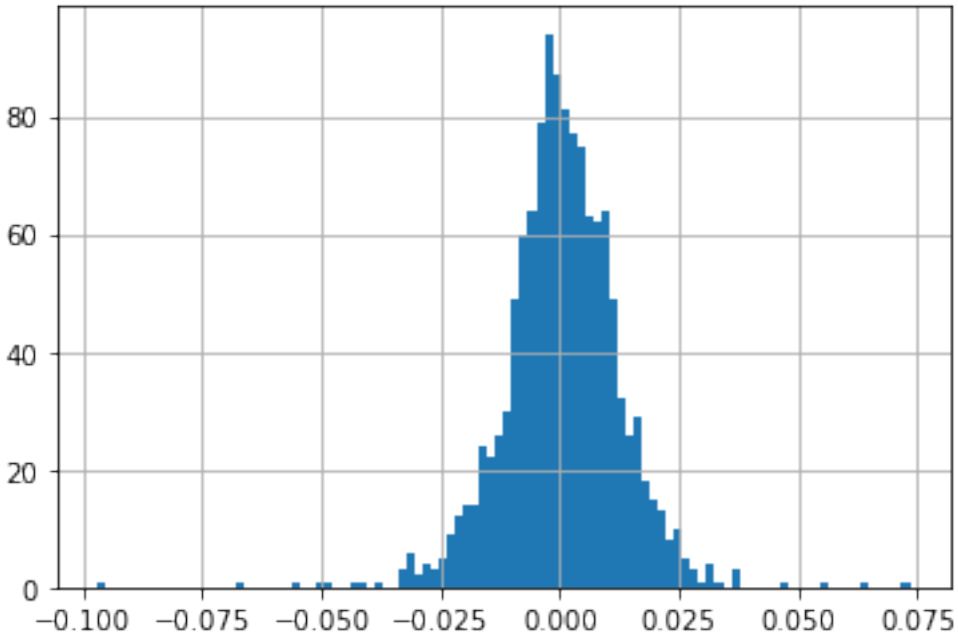
[41]:

	date	open	high	low	close	volume	Name	prev_close	\
498152	2013-02-08	27.920	28.325	27.920	28.185	7146296	SBUX	NaN	
498153	2013-02-11	28.260	28.260	27.930	28.070	5457354	SBUX	28.185	
498154	2013-02-12	28.000	28.275	27.975	28.130	8665592	SBUX	28.070	
498155	2013-02-13	28.230	28.230	27.750	27.915	7022056	SBUX	28.130	
498156	2013-02-14	27.765	27.905	27.675	27.775	8899188	SBUX	27.915	

	return	return2	log_return
498152	NaN	NaN	NaN
498153	-0.004080	-0.004080	-0.004089
498154	0.002138	0.002138	0.002135
498155	-0.007643	-0.007643	-0.007672
498156	-0.005015	-0.005015	-0.005028

[42]:
```
'''
Next, we're going to plot a histogram of our log returns using the highest␣
 ↪function, as you can see,
we get pretty much the exact same distribution that we got for the non log␣
 ↪returns.
'''

sbux['log_return'].hist(bins=100)
```

[42]: <AxesSubplot:>



[43]:
```
'''
Finally, we can calculate the sample mean and the sample standard deviation of␣
 ↪the log return again

as before, we see that the mean is very close to zero
and the standard deviation is about zero point zero one.
'''
sbux['log_return'].mean(),sbux['log_return'].std()
```

[43]: (0.000523590274810868, 0.012381234216101253)

[ ]: