Forecasting Volatility

Denis Ovchinnikov

June 15, 2021

Problem formulation. *You are given data on prices of* 6 *stocks sampled at* 1*-minute intervals over the period of one year. Write a* 2-4 *page report answering the following questions for each of the stock:*

1. *Your best estimation for the volatility (in annualized percent return) over the next month following the end of samples.*
2. *Your confidence in that estimation.*

Include the code and explain any models used in the prediction.

Report

This report forecasts and estimates the volatility of several stocks over the next month. We are given a data set that records prices for 6 stocks over a period of one year at 1 minute intervals. I will denote stocks as ′a′, ′b′, ′c′, ′d′, ′e′, ′f′, as given in the data set.

1 Cleaning and preparing data

First, I entered all data in pandas DataFrame ′df′. For convenience, I re-indexed the data frame according with a pandas DateTime index. For that, I chose an arbitrary Monday as a starting date of the records (controlled by a variable ′starting_day′), and then adjusted date/time according to the data provided. I also assume all prices are given in dollars (but specific units don't affect the analysis). See inputs 2,3.

I then visualized and cleaned the data. After a quick inspection (inputs/outputs 6,7), we see a few things about data that need to be addressed before any further analysis:

1. There are missing values. For most of the stocks this shouldn't be a problem since missing values represent a small fraction of all records (under 100 observations out of $98,000$ total observations). One exception is stock ′f′ that has 1371 missing observations.
2. There are some obviously incorrectly entered values. More specifically, stock ′a′ has 93 observations that indicate price \$0, and stock ′d′ has 93 observations that indicate price \$1. Note that outside of those observations, minimal prices for ′a′ and ′d′ are over \$300 and \$35, so clearly the observations mentioned are incorrect. Interestingly those errors for ′a′, ′d′ were recorded at exactly same intervals (see input/output 8).

I suggest interpolating missing and incorrect values for all the stocks, based on linear interpolation. I believe this is a reasonable suggestion, and due to low amount of corrected data, exact method of dealing with it shouldn't significantly alter the results. We can make a case of using a different method for stock ′f′, due to a relatively large amount of missing data, however as we will see later, there are some specifics about it that make this method acceptable. See input 9 for interpolation code and output 9 for the graphs of interpolated results). From here on, DataFrame ′df′ has interpolated data, while DataFrame ′df_raw′ is a copy of the original data.

The final part of preparing data is re-sampling the data. It seems like there is no consensus of ′the best′ sampling frequency. While theoretically, more frequent samples should provide more accurate measures of volatility, and therefore provide better forecasts, it seems like over-sampling might be affected by 'noise'. One of the more common sampling windows seems to be 5 minutes (see e.g. [1]), therefore I will re-sample all data with 5-minute intervals by using mean price within that 5-minute interval. See inputs/outputs 10,11 for the implementation. Variable 'df_5_min' represents the data re-sampled as discussed above.

Since volatility is usually estimated using absolute log-returns, let's compute log-returns (see input 12) and visualize absolute log-returns (see output 12). The formula for R_i– the log-returns at moment i, is $R_i = \ln(p_i) - \ln(p_{i-1})$, where p_i is the price at moment i. It is similar (but not quite equal) to the rate of return, which is measured in % increase. By drawing histograms of log-returns (input/output 13), we see that most of them look reasonable, with the exception of 'f'.

In stock 'f' we observe that 90% of 5-minute intervals (input/output 14), log-returns are equal to 0, indicating no change in price. We also note that when the price does change, it changes by a significant amount. In my opinion, this indicates that **the intervals where the price of 'f' *doesn't change* in actuality correspond to intervals where no trading occurs**. It is an extra assumption, that ideally I would double-check with the source of data, but seems reasonable from given data. To deal with this, I suggest we remove all intervals for 'f' with constant price, and only estimate volatility when a price change occurs. This is done in input 18. We immediately see in output 18 that the histogram of stock 'f' log-returns looks much more reasonable. **We will use described DataFrame 'log_returns' to estimate volatility.**

2 Estimating and inspecting daily volatility

We are now ready to proceed with estimating historic volatility. There are a few ways to estimate volatility from log-returns, with the precise method depending on applications and the source of data. Probably the most simple and interpretable estimation of the volatility is square root of realized variance. Perhaps a more common estimation for high-frequency data is realized volatility estimate (sum of squares of returns, see [1], [2]). Due to any lack of information of the nature of stocks or desired applications, I will use realized standard error, but changing to realized volatility is simple (see comment in input 19) and doesn't affect results much.

I will estimate daily volatility from available data and then use it to forecast average daily volatility σ_d over the next month. The monthly volatility σ_m then can be estimated by the formula $\sigma_m = \sqrt{N}\sigma_d$ where N is the number of trading days in the next month, and annualized volatility in percents can be estimated by the formula $\sigma_y = \sqrt{M} \cdot \sigma_d \cdot 100$, where M is the number of trading days in a year

The input 19 establishes DataFrames 'daily_var' and 'daily_vol' that represent daily variation and square root of the variation of log-returns. See output 20 for daily volatility graphs. As expected, in stock 'c' due to a one-time drop in price, the picture is not very informative. We deal with that later. I also introduce DataFrames 'daily_avg' and 'daily_return' (input 22) that represent average daily price and total daily log-return correspondingly. None of my final models utilized that data, but it might be useful in some other models (e.g. ARCH).

I next inspect correlations of daily volatility with daily returns and average prices, correlations of volatilises between different stocks, as well as auto-correlations of each volatility series (see inputs/outputs 23-34). Importantly for us, stocks 'a', 'd', 'e', 'f' do not show any significant auto-correlation, suggesting simple models like a historic average might be the best. On the contrary, stock 'b' shows relatively high auto-correlation, suggesting a model that values recent observations more than more distant in time. Finally, stock 'c', as expected, shows completely unstructured information unless the extreme volatility (corresponding to the price drop) is removed. To deal with it, we introduce one more column in daily volatility DataFrame, named 'c_no_extremes' that removes that extreme value (input 29). I replace it with the second biggest value of volatility of 'c', but several other approaches seem valid (replace it with nan, with twice the second largest value, etc.). After doing that, we see that volatility of 'c' seems to be better behaved and in fact show high auto-correlation, suggesting similar class of models as for 'b'.

3 Evaluating performance of models

First of all, I want to re-iterate that I will forecast **square mean daily volatility** σ_{future} over the next month (30 days). I chose to forecast this instead of monthly volatility for two reasons:

1. Due to not having a specific starting date, we do not know exactly how many trading days the next month has. If this number is N, then the estimation of monthly volatility σ_m is $\sigma_m = \sqrt{N}\sigma_{\text{future}}$, so clearly any estimate that doesn't assume N would not be very precise.
2. Monthly volatility does not necessarily represent volatility over the duration of the next month, and rather estimates volatility within this month. This is in particular very evident for stock 'd': due to large changes within each day, monthly volatility would be large, while the total price change after a month is not that significant.

Similarly to estimating volatility, there is no single universal way to compare performances of different models (or to evaluate performance of a single model), see e.g. [2]. The choice of an error function is especially influenced by the desired application of forecasted volatility. I would use two estimators: **mean percent error** and **mean absolute error**. Assume we are given daily volatilities $\sigma_1, \ldots, \sigma_t$ over t days, and we use a model to estimate σ_{future}– the square mean daily volatility over the next month, and the estimation is $\hat{\sigma}_{\text{future}}$. Assume furthermore, that out-of-set data provides an actual realized value of σ_{future}. Then the mean percent error is $\text{pe}_t = \frac{\hat{\sigma}_{\text{future}} - \sigma_{\text{future}}}{\sigma_{\text{future}}}$, and the mean absolute error is $\text{ae}_t = |\hat{\sigma}_{\text{future}} - \sigma_{\text{future}}|$. If furthermore we do this experiment for several dates $t_1, t_2, \ldots, t_k$, I will implement total absolute error:

$$\text{tae} = \text{tae}(\text{model}, t_1, \ldots, t_k) = \frac{(\sum \text{ae}_{t_i})/k}{\text{Mean}(\sigma_i)}$$

that represents average absolute error relative to the average volatility, and total percent error:

$$\text{tpe} = \text{tpe}(\text{model}, t_1, \ldots, t_k) = \frac{\sum |\text{pe}_{t_i}|}{k}$$

that represents mean absolute percent error. In all of the experiments, **the final decision/analysis is based on tae (implemented as 'total_error(model, error_method='Raw error',...) function)**. It is easy to change to different evaluation techniques as well, using total_error and test_model functions with correct parameters.

There are several other evaluation techniques used in literature (see 4.1 of [2] for an overview), but without having specifics of the desired applications, it is impossible to choose a perfect technique, so I just use the simplest ones (in my opinion).

The final point of discussion is choosing test points $t_1, \ldots, t_k$ over the given year of data. Intuitively, picking as many points as possible should provide better estimates of performance of the model. However, if 30-days intervals of t_i and t_j overlap, then the error for t_i an t_j would not be independent. For this reason, I sample days t_i every 30 days, that is, if t is the last day of observations, $t_k = t - 30$, $t_{i-1} = t_i - 30$ for other i's. Furthermore I start when the initial date t_0 is less than 60 days. This ensures sufficient amount of data for the first error estimation.

This leads to different computed errors for different t_i to be (at least somewhat) independent, but notably significantly limits our testing set size (each method is only tested on 10 dates). Again, in my opinion, simply testing over more densely sampled t_i's wouldn't produce better estimates and would add a false sense of a chosen model performing well.

4 Suggested models

Denote computed daily volatility $\sigma_1, \ldots, \sigma_t$ over days $1, \ldots, t$. To forecast σ_{future}– the average daily volatility over the next month, I consider the following models:

1. Historic mean. That is, $\hat{\sigma}^2_{\text{future}} = \frac{1}{t}\sum_{i=1}^{t}(\sigma_i^2)$. The advantages of this model is that it captures the fact that volatility is stable over time, and that this model is easy to interpret. The main disadvantage is that it doesn't weight recent observations more than older ones. Due to lack of auto-correlation, I think that this simple model would be recommended for stocks 'a', 'd', 'e', 'f'.
2. Moving mean. That is, for some fixed τ (lookback_interval in the code), $\hat{\sigma}^2_{\text{future}} = \frac{1}{\tau}\sum_{i=t-\tau+1}^{t}(\sigma_i^2)$. This model has the advantages in that it captures the impact of more recent observations, and again is easy to interpret. It however fails to capture long-lasting impact of volatility, so I don't think it is recommended for given stocks.
3. Exponential decay. This model in a sense 'combines' historic mean and moving mean by giving bigger weigth to recent observations. More precisely, for a given γ, $\hat{\sigma}^2_{\text{future}} = \frac{1}{\sum_{i=0}^{t-1}\gamma^i}\sum_{i=1}^{t}\sigma_i^2\gamma^{t-i}$. For uniformity of functions in implementation, we pick γ in such a way that $\gamma^{t-\text{lookback\_interval}} = 1/2$. I think this model is recommended for stocks 'b', 'c' due to them having significant auto-correlation.

There are numerous other models implemented and considered in literature, notably ARCH/GARCH family of models, and stochastic volatility models. I chose to stay away from them for this exercise due to extremely limited test set size (only 10 observations for each model evaluation). This, in conjunction with complexity of the above-mentioned models, risks over-fitting training data and failure to generalize properly on out-of sample data. If, in contrast to forecasting one month ahead, we would need to forecast one day (or even one week) ahead, I would strongly consider using ARCH/GARCH models, in particular for 'b' and 'c' as they show significant auto-correlation and correlation with average daily price.

One other consideration would be to consider models use of other stocks data to predict volatility for each stock. This seems promising for the triple of stocks 'a', 'b' ,'f' due to their volatility having relatively strong correlation. I did not consider this for the same reason as above: adding extra parameters when we only have 10 test cases increases risk of over-fitting.

5 Final errors and model selection

Using total absolute and total relative errors for each of the methods, we get the following estimations of errors:

```
[49]:        tae HM    tre HM  tae MM (d=10)  tre MM (d=10)    tae ED (d=30)  tre ED (d=30)
      a    0.272356  0.254990       0.210867       0.199707       0.275881       0.265818
      b    0.323691  0.228082       0.238068       0.172459       0.167944       0.117074
      c    1.707013  1.173616       1.987098       1.844812       1.863170       1.369053
      d    0.050792  0.048688       0.071880       0.069600       0.054210       0.052536
      e    0.046338  0.045083       0.061724       0.060141       0.052215       0.050902
      f    0.305863  0.234798       0.363184       0.299144       0.311063       0.231917
      c_e  0.341306  0.265150       0.386297       0.355020       0.270812       0.231115
```

Here HM stands for Historic Mean, MM for Moving Mean, ED for Exponential decay, and c_e for stock 'c' with eliminated extreme. We see that, as expected, for stocks 'b' and 'c_e', exponential decay method seems to give the best results, by far. For the other stocks different methods perform mostly the same. For this reason, I will use exponential decay to forecast stocks 'b' and 'c' (and use 'c_e' to forecast 'c'), and historic mean for other stocks (on can argue that moving mean is better for stock 'a', but improvement is insignificant enough that it can be caused by the small amount of data). One can also try to optimize lookback_days value, e.g. by cross-validation, which again I didn't do due to risk of overfitting.

We refer to inputs/outputs 39-49 for more detailed look at errors of various methods.

6 Final predictions

Here we use the models chosen above to make predictions for corresponding stocks, we get the following predictions for square mean daily volatility over the next month (inputs/outputs 50,51). **Assuming 253 trading days in a year, and the formula $\sigma_y = \sigma_d \cdot \sqrt{253}$, the estimated annualized volatility (in %) are $\hat{\sigma}_{y,a} = 2.638384$, $\hat{\sigma}_{y,b} = 10.485158$, $\hat{\sigma}_{y,c} = 4.167963$, $\hat{\sigma}_{y,d} = 13.246544$, $\hat{\sigma}_{y,e} = 3.151951$, $\hat{\sigma}_{y,f} = 9.992446$.**

I expect the actual volatility to be within 95% confidence intervals constructed by sampling standard deviations of corresponding errors:

$$\hat{\sigma}_{y,a} \in [1.052503, 4.224265], \hat{\sigma}_{y,b} \in [5.087983, 15.882333], \hat{\sigma}_{y,c} \in [1.807707, 6.528220],$$

$$\hat{\sigma}_{y,d} \in [11.660826, 14.832262], \hat{\sigma}_{y,e} \in [2.777973, 3.525930], \hat{\sigma}_{y,f} \in [2.192945, 17.791946]$$

Here I simply use standard distribution, but one can argue that t-distribution is better due to small sample size. I have to emphasize that this forecast doesn't account for unexplained jumps in price (e.g. stock 'c' price drop). It is also worth mentioning that due to only having one year worth of data, and forecasting month in advance, some factors might be unobserved (e.g. some stocks might have seasonal behaviour).

Appendix: code

Due to the exercise being mostly about data exploring, I did all the coding in Jupyter notebook. If I am to actually implement this, I would include the essential code in a Python script, and leave graphs and data exploration in a notebook.

I decided to leave the comments/thoughts I had in the process as Jupyter markdowns, they do not add anyhing essential to the summary on the first 4 pages.

I also kept bibliography short, and cited only to two of the most established and frequently cited papers on volatility research. I never done volatility forecasting before, so I read through several other articles.

References

[1] Torben G. Andersen, Tim Bollerslev, Francis X. Diebold, and Heiko Ebens. The distribution of realized stock return volatility. *Journal of Financial Economics*, 61(1):43–76, 2001.

[2] Ser-Huang Poon and Clive W.J. Granger. Forecasting volatility in financial markets: A review. *Journal of Economic Literature*, 41(2):478–539, June 2003.

[1]:
```
import pandas as pd
import numpy as np
import sklearn
import math
import matplotlib.pyplot as plt
import sklearn
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge, RidgeCV
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error, make_scorer
from statsmodels.graphics.tsaplots import plot_pacf
import statsmodels.api as sm
from scipy import stats
import seaborn as sns
```

[2]:
```
df=pd.read_csv("stockdata3.csv")
starting_day=pd.to_datetime('2020-05-04T00:00')
df['dtime']=starting_day
df['day']=df['day'].apply(lambda day: pd.Timedelta(days=day-1))
df['timestr']=df['timestr'].apply(lambda time: pd.Timedelta(time))
df['dtime']=df['dtime']+df['day']+df['timestr']
df.set_index('dtime',inplace=True)
```

[3]:
```
stocks=['a','b','c','d','e','f']
```

let's replace day+time with a single datetime column. To do so, fix a day that corresponds to day 1 (from examining the data, it should be Monday), then add a day and time to that timestamp. I chose first Monday of May 2020 (you can probably figure out exact time period by holidays, but for the purpose of the exercise doesn't seem to be important).

Sampling over as much as possible should give us more robust estimates, but might be sensitive to various noise sources. The common compromise in literature seems to use 5 minute intervals, but it probably highly depends on particular stock/reporting system.

We should also think about whether our model performance is evaluated on in-sample or out-of-sample data. While out-of-sample forecasts are much more reflective of model's performances, with only 1 year worth of data and the task of estimating monthly volatility, it seems hard to have a reasonable amount of out-of-sample testing (unless done daily).

I'm going to predict average daily volatility over the next 30 days. The reason being that the number of trading days in the next month depends on the month/holidays, etc. For a particular month the estimate for monthly volatility would be my estimate times the sqaure root of the number of trading days that month.

[4]:
```
df.head()
```

[4]:
```
                        day          timestr        a       b       c       d  \
dtime
2020-05-04 09:30:00 0 days 0 days 09:30:00  325.450  13.795  94.500  49.985
2020-05-04 09:31:00 0 days 0 days 09:31:00  325.245  13.890  94.515  49.990
2020-05-04 09:32:00 0 days 0 days 09:32:00  325.580  13.905  94.565  49.995
2020-05-04 09:33:00 0 days 0 days 09:33:00  325.470  13.955  94.645  50.065
2020-05-04 09:34:00 0 days 0 days 09:34:00  325.295  13.975  94.580  50.030

                         e       f
dtime
2020-05-04 09:30:00  49.93  17.025
2020-05-04 09:31:00  49.96  17.025
2020-05-04 09:32:00  49.96  17.025
2020-05-04 09:33:00  49.92  17.025
2020-05-04 09:34:00  49.90  17.025
```

[5]:
```
df.dtypes
```

[5]:
```
day        timedelta64[ns]
timestr    timedelta64[ns]
a                  float64
b                  float64
c                  float64
d                  float64
e                  float64
f                  float64
dtype: object
```

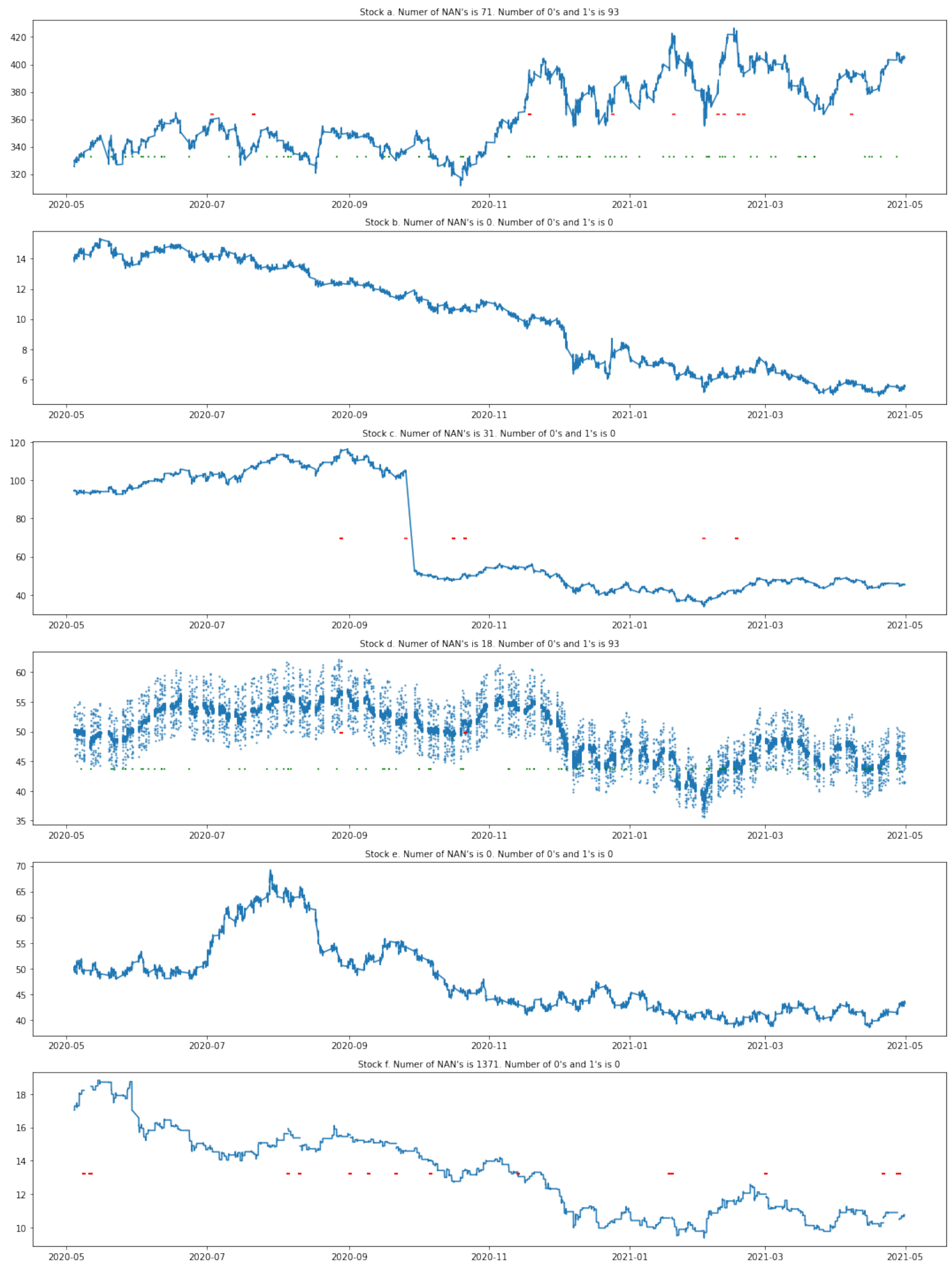
[6]:
```
df.loc[:,'a':'f'].describe()
```

[6]:
```
                  a             b             c             d             e  \
count  98281.000000  98352.000000  98321.000000  98334.000000  98352.000000
mean     363.767737     10.048836     69.445570     49.803747     47.727738
std       28.281333      3.332643     28.675762      4.539923      7.053875
min        0.000000      4.885000     33.807000      1.000000     38.550000
25%      341.785000      6.605000     45.330000     46.175000     42.040000
50%      357.915000     10.595000     50.960000     50.105000     45.410000
75%      387.890000     13.465000    101.905000     53.735000     51.350000
```

```
max      426.560000     15.335000    116.385000     62.137000     69.210000

                  f
count  96981.000000
mean      13.204838
std        2.404705
min        9.365000
25%       10.945000
50%       13.325000
75%       15.035000
max       18.815000
```

We see that 'a' has values of 0 (93 of them to be precise, see below), 'd' has values of 1, and all columns have some missing values. Stocks 'a' to 'e' have relatively little missing values (under 100 for each), but stock 'f' is missing for a sizable part (about 2.5%) of the data.

[7]:
```
fig,axs=plt.subplots(6,1)
i=0
for stock in stocks:
    nan=df.index[df[stock].isnull()]
    value_0_or_1=df.index[(df[stock]==0) | (df[stock]==1)]
    if stock!='d':
        axs[i].plot(df.loc[(df[stock]!=0) & (df[stock]!=1),stock])
    else:
        axs[i].scatter(df.index[(df[stock]!=0) & (df[stock]!=1)],df.loc[(df[stock]!=0)␣
 ↪& (df[stock]!=1),stock], s=1)
    axs[i].scatter(nan,[df[stock].mean()]*len(nan), s=10, c='red', marker="_")
    axs[i].scatter(value_0_or_1,[(-df[stock].max()+3*df[stock].mean())/
 ↪2]*len(value_0_or_1), s=7, c='green', marker="|")
    axs[i].set_title("Stock "+stock+". Numer of NAN's is "+str(len(nan))+". Number of␣
 ↪0's and 1's is "+str(df.loc[((df[stock]==0) | (df[stock]==1)),stock].count())␣
 ↪,fontsize=10)
    i+=1
fig.set_size_inches(15,20)
fig.tight_layout()
```



Visually it looks like ‘b’, ‘c’, ‘e’, ‘f’ are pretty stable (besides a really big drop in ‘c’ and a couple of smaller dropps/raises in other places).

Stock ‘a’ seems to be somewhat volitile, and ‘d’ looks extremely volitile within small intervals of time, while having a pretty consistent moving average. Also decided to scatterplot ‘d’ to have a better idea of how prices are distributed, it looks like most points are closer to the moving average than it seemed from a normal graph, with still a sizable part varying strongly.

Maybe we should drop weekends for better graphs?

Curious that 'a' and 'd' has the same number of (obviously) incorrectly recorded points. Might be worth looking whether those occured at the same time.

```
[8]: (1-(df.loc[df['a']==0].index==df.loc[df['d']==1].index)).sum()
```

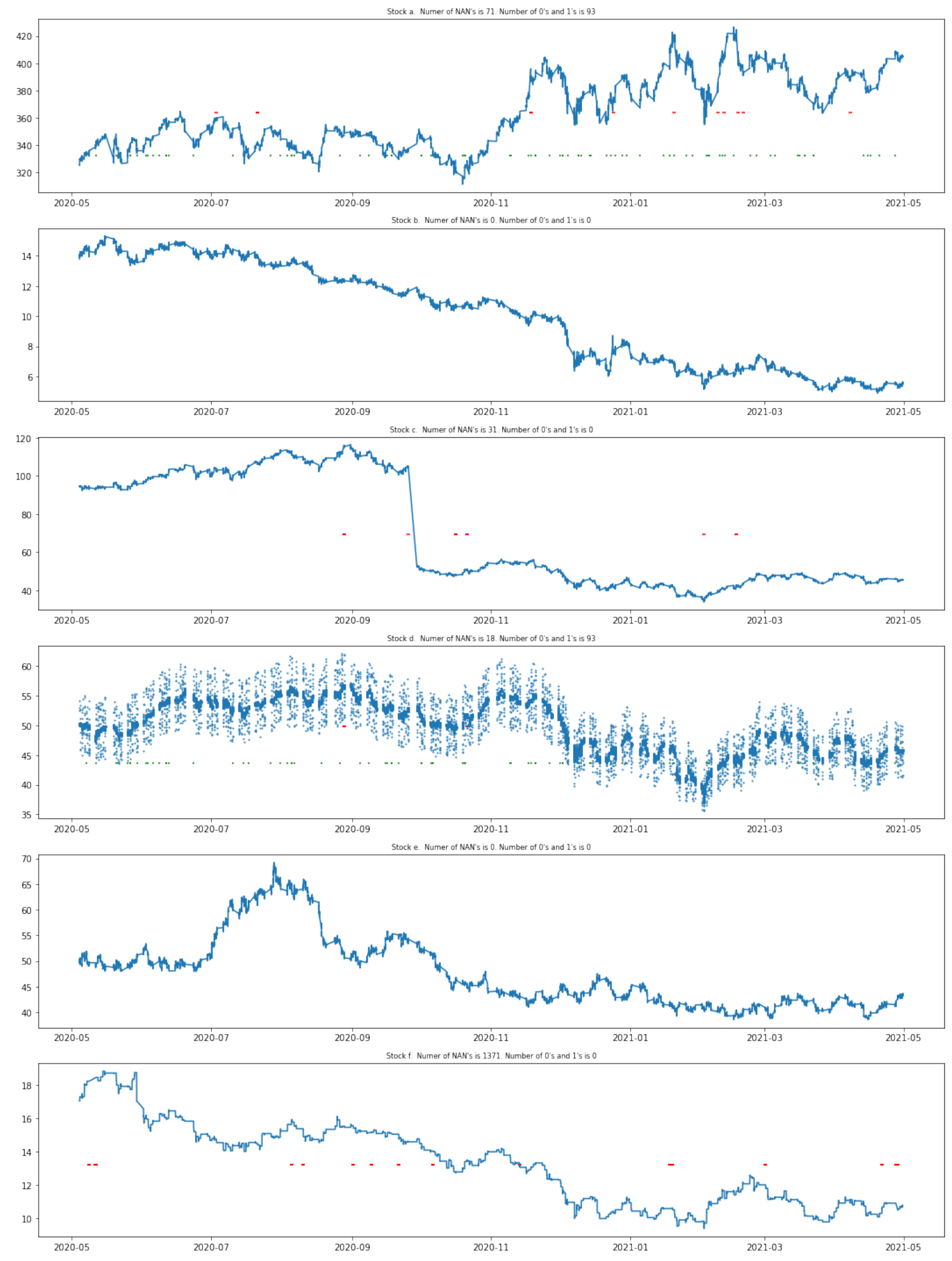
```
[8]: 0
```

Now, to deal with missing and incorrectly reported prices, I suggest just doing linear interpolation. It seems reasonable for most of the data, since the ammount of data interpolated is small, and doesn't clamp up a lot. One exception is 'f' that has sizable parts of data missing for continuous intervals of time, so it might be better to just not analyze our approach near those missing intervals.

Maybe a better approach would be to add noise to interpolations with variations computed from data.

Let's interpolate data:

```
[9]: fig,axs=plt.subplots(6,1)
i=0
df_raw=df.copy() ### just creating a copy of raw data in case we need to access nan's␣
 ↪etc later.
for x in stocks:

    #### plot nan's in red and 0/1s in green (around the mean of each graph)
    nan=df_raw.index[df_raw[x].isnull()]
    value_0_or_1=df.index[(df_raw[x]==0) | (df_raw[x]==1)]
    axs[i].scatter(nan,[df_raw[x].mean()]*len(nan), s=10, c='red', marker="_")
    axs[i].scatter(value_0_or_1,[(-df_raw[x].max()+3*df_raw[x].mean())/
 ↪2]*len(value_0_or_1), s=7, c='green', marker="|")
    axs[i].set_title("Stock "+x+".  Numer of NAN's is "+str(len(nan))+". Number of 0's␣
 ↪and 1's is "+str(len(value_0_or_1)) ,fontsize=8)

    #### Interpolate nan's and 0/1s. Doing it inplance, which might not be ideal if we␣
 ↪need raw data again later.
    df.loc[value_0_or_1,x]=float('nan')
    df[x].interpolate(method='time',inplace=True)

    #### Plot graphs: normal graphs for all but 'd'
    if x!='d':
        axs[i].plot(df.loc[(df[x]!=0) & (df[x]!=1),x])
    else:
        axs[i].scatter(df.index[(df[x]!=0) & (df[x]!=1)],df.loc[(df[x]!=0) & (df[x]!
 ↪=1),x], s=1)

    i+=1
fig.set_size_inches(15,20)
fig.tight_layout()
```



Now let's resample to 5-minute intervals (with mean over those 5 minute intervals as the resampling function).

[10]:
```
df_5_min=df.resample('5T').mean()
```

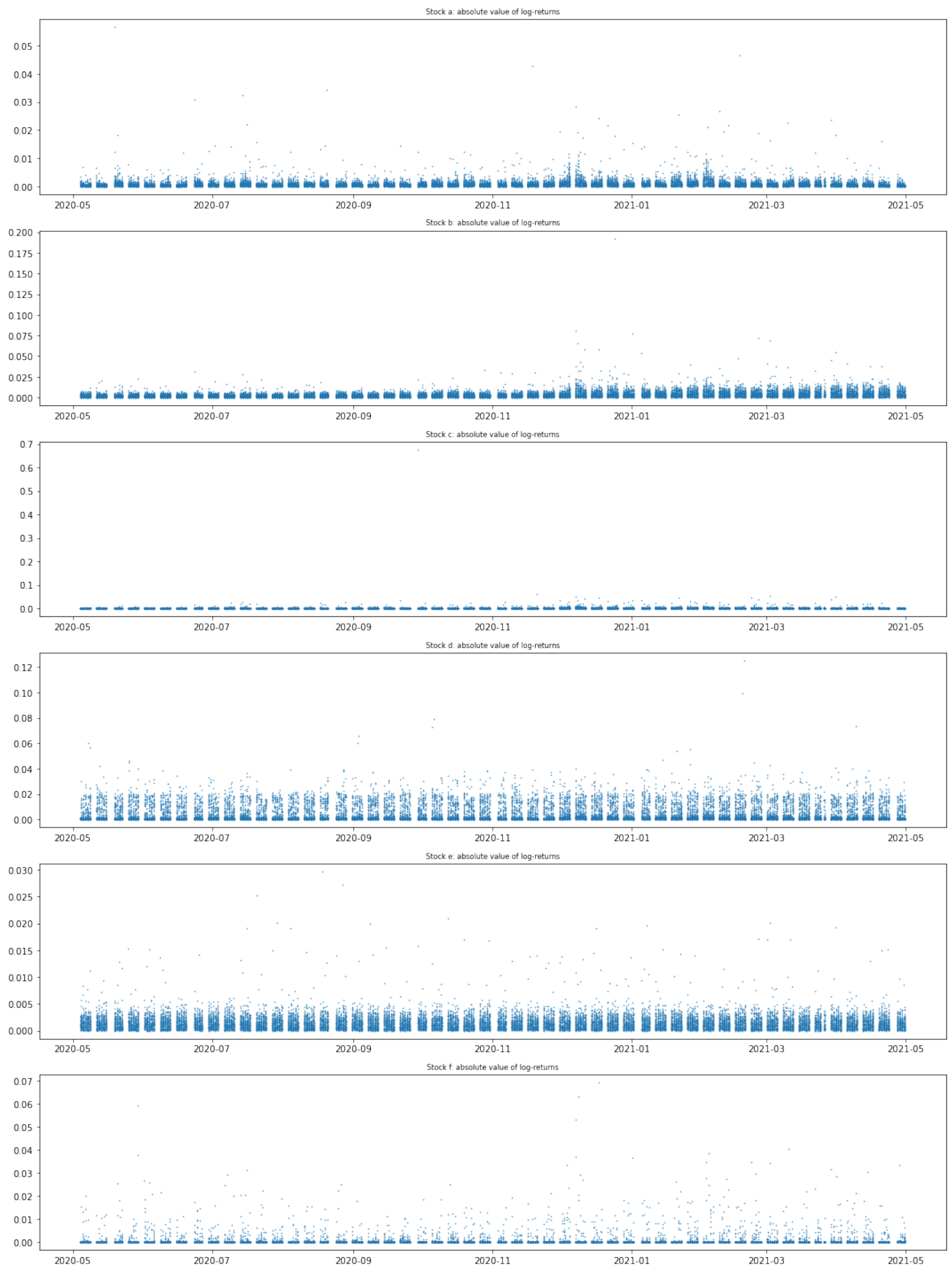
```
df_5_min=df_5_min.join(df,  how='inner', rsuffix='_old').loc[:
 ↪,['day','timestr','a','b','c','d','e','f']]
```

[11]:
```
df_5_min.head()
```

[11]:
```
                              day         timestr        a       b       c       d  \
dtime
2020-05-04 09:30:00 0 days 0 days 09:30:00  325.408  13.904  94.561  50.013
2020-05-04 09:35:00 0 days 0 days 09:35:00  325.745  13.963  94.574  50.061
2020-05-04 09:40:00 0 days 0 days 09:40:00  326.890  14.023  94.610  50.139
2020-05-04 09:45:00 0 days 0 days 09:45:00  327.540  13.945  94.653  50.226
2020-05-04 09:50:00 0 days 0 days 09:50:00  327.492  13.909  94.644  50.259

                          e       f
dtime
2020-05-04 09:30:00  49.934  17.025
2020-05-04 09:35:00  49.976  17.025
2020-05-04 09:40:00  49.884  17.025
2020-05-04 09:45:00  49.820  17.025
2020-05-04 09:50:00  49.948  17.025
```

Let's compute the log returns. We can either use percent change (p(t+1)-p(t))/p(t). They are pretty close (for small changes), but it is convinient to have additive property.

[12]:
```
log_returns=pd.DataFrame(index=df_5_min.index)
for stock in stocks:
    log_returns[stock]=np.log(df_5_min[stock]).diff()
log_returns['day']=df_5_min['day']

fig_log, axs=plt.subplots(6,1)
i=0
for stock in stocks:
    axs[i].set_title("Stock "+stock+": absolute value of log-returns", fontsize=8)
    #axs[i].plot(log_returns[stock]*log_returns[stock])
    axs[i].scatter(log_returns.index,np.abs(log_returns[stock]),s=0.2)
    i+=1
fig_log.set_size_inches(15,20)
fig_log.tight_layout()
```



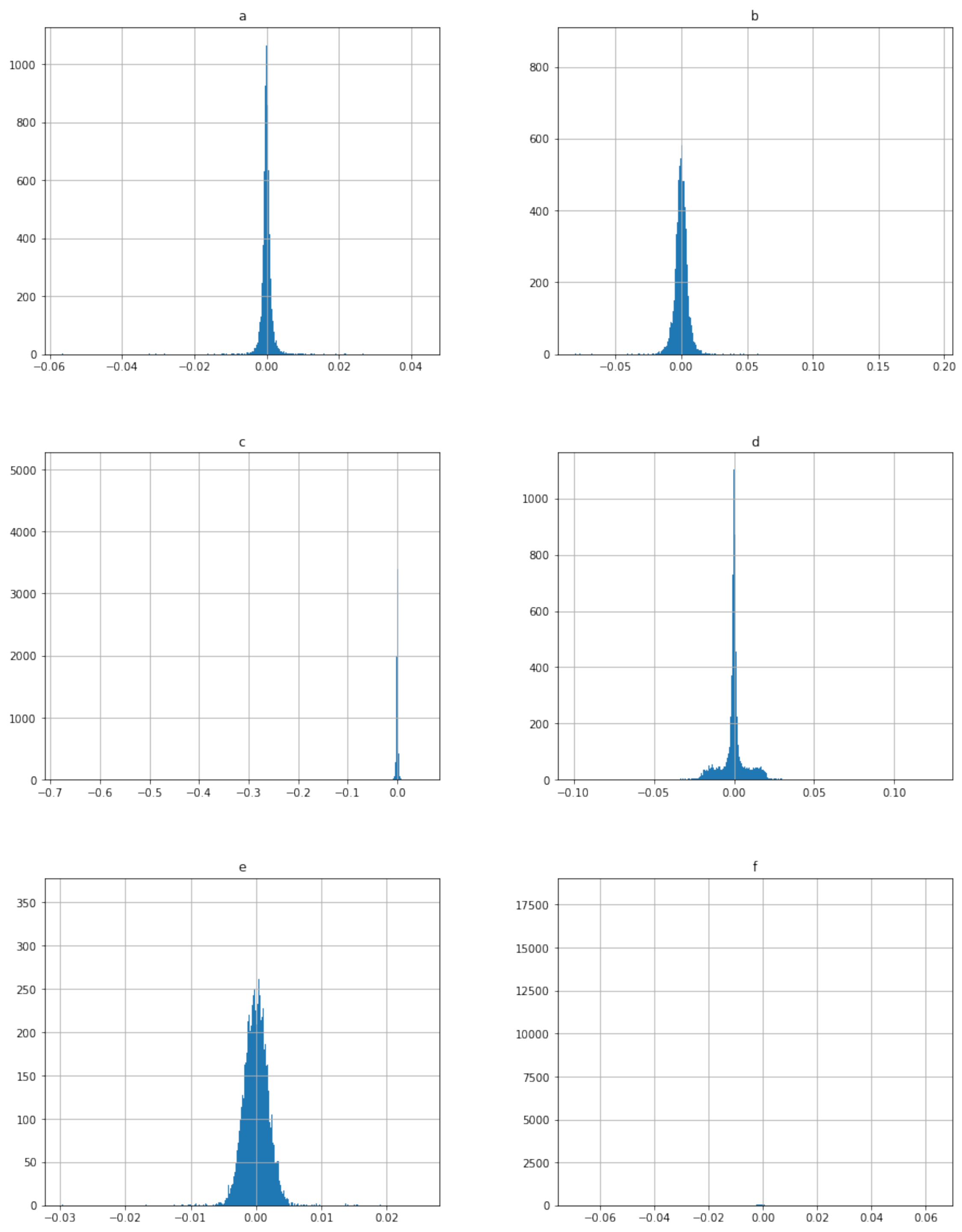
One extra thing we notice from stock ‘f’ is the amount of time intervals where the price is constant. This might be due to low trading volume (e.g. if stock is only traded a few times a day). We should account for that when we compute volatility (as if that is the case, constant time intervals represent no trading, not a constant trading price). This can also be observed fromthe figure below.

Similar thing can be seen for stock ‘b’: there is a clear duality between 0 and non-0 log-returns (and 0 log-returns amount to almost 30% of observations which is still more than one can expect). This again suggests that the periods with 0 returns correspond to no trading events periods, not actually to periods with constant price. ## This is resolved by resampling.

If I am getting this data from an outside source, I would ask for the amount of each stock traded during each time interval to incorporate into volatility model. fig_hist, axs = plt.subplots(3,2,tight_layout=True, sharex=False,sharey=False) i,j=0,0 n_bins=1000 for stock in stocks: axs[i][j].set_title("Stock "+stock+" histogram of log-returns (number of bins is "+str(n_bins)+")", fontsize=8) axs[i][j].hist(log_returns[stock],bins=n_bins) if j==1: j=0 i+=1 else: j+=1 fig_hist.set_size_inches(15,20)

[13]:
```
log_returns.loc[:,'a':'f'].hist(layout=(3,2),bins=1000,figsize=(15,20))
```

[13]:
```
array([[<AxesSubplot:title={'center':'a'}>,
        <AxesSubplot:title={'center':'b'}>],
       [<AxesSubplot:title={'center':'c'}>,
        <AxesSubplot:title={'center':'d'}>],
       [<AxesSubplot:title={'center':'e'}>,
        <AxesSubplot:title={'center':'f'}>]], dtype=object)
```



```
[14]: amount_of_0=[0]*len(stocks)
      i=0
      for stock in stocks:
          amount_of_0[i]=round(log_returns.loc[log_returns[stock]==0,stock].size/log_returns.
       ↪index.size,4)
          i+=1
      amount_of_0
```

```
[14]: [0.0023, 0.0258, 0.0021, 0.0054, 0.0087, 0.9047]
```

we see that indeed 90% of 'f' log_returns is 0

In raw files, stock 'b' also had way too many 0's, but after resampling, this problem seemed to be fixed

I propose to get rid of 0's in stock 'f' completely. Alternatively, we can only get rid of most of 0's to get the number of no change to be on par with the number of observations of non 0 entries.

Again, one thing to care for here is to double check the sourse of data to see that those no change periods are actually no trading periods.

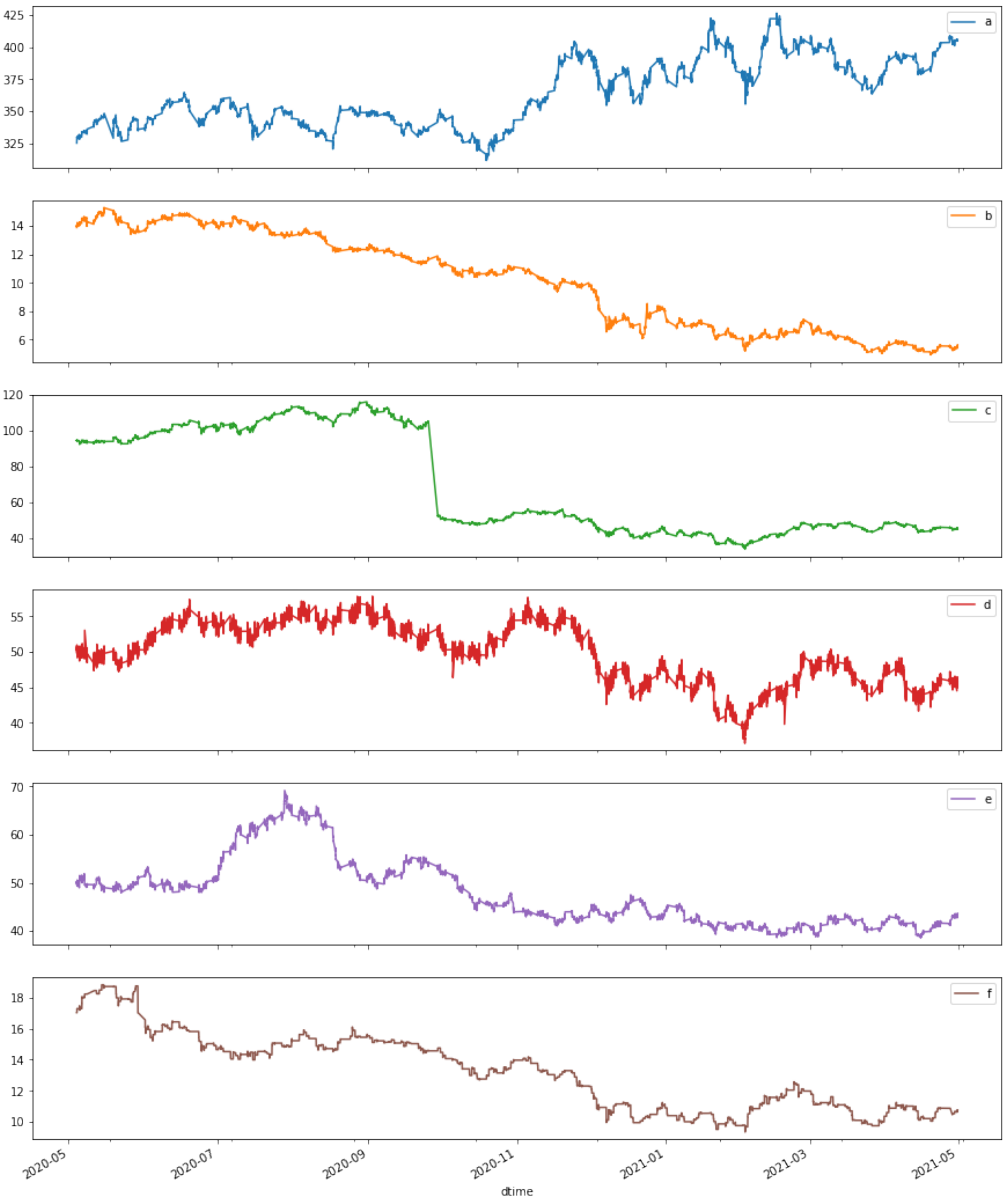
```
[15]: log_returns.loc[:,'a':'f'].mean()/log_returns.loc[:,'a':'f'].var()
```

```
[15]: a    3.938703
      b   -1.707653
      c   -1.277662
      d   -0.064443
      e   -1.690904
      f   -4.923666
      dtype: float64
```

We see that most stocks have substential avg. price changes.

```
[16]: df_5_min.loc[:,'a':'f'].plot(subplots=True, layout=(6,1),figsize=(15,20))
```

```
[16]: array([[<AxesSubplot:xlabel='dtime'>],
             [<AxesSubplot:xlabel='dtime'>],
             [<AxesSubplot:xlabel='dtime'>],
             [<AxesSubplot:xlabel='dtime'>],
             [<AxesSubplot:xlabel='dtime'>],
             [<AxesSubplot:xlabel='dtime'>]], dtype=object)
```



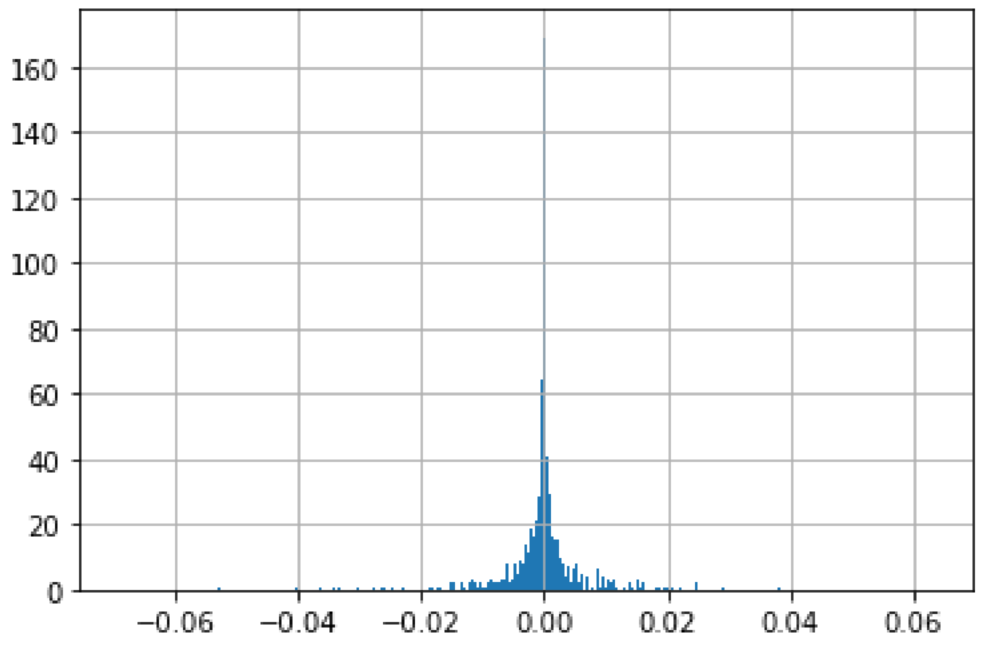
```
[17]: log_returns.describe()
```

```
[17]:                   a             b             c             d             e  \
      count  19871.000000  19871.000000  19871.000000  19871.000000  19871.000000
      mean       0.000011     -0.000046     -0.000037     -0.000004     -0.000007
      std        0.001672      0.005174      0.005384      0.008336      0.002001
      min       -0.056551     -0.080619     -0.676460     -0.098887     -0.029644
      25%       -0.000538     -0.002563     -0.000834     -0.001535     -0.001191
      50%        0.000003      0.000000     -0.000004     -0.000019      0.000000
      75%        0.000548      0.002498      0.000804      0.001460      0.001192
```

```
max        0.042774      0.192560      0.048259      0.124918      0.025258

                  f                         day
count  19871.000000                       19872
mean      -0.000023  180 days 07:12:01.739130434
std        0.002167  104 days 14:52:53.334428426
min       -0.069025             0 days 00:00:00
25%        0.000000            88 days 00:00:00
50%        0.000000           179 days 00:00:00
75%        0.000000           270 days 00:00:00
max        0.063149           361 days 00:00:00
```

[18]:
```
log_returns.loc[log_returns['f']==0,'f']=np.nan
log_returns['f'].hist(bins=1000)
```

[18]:
```
<AxesSubplot:>
```



This seems like a much more likely distribution of log-returns.

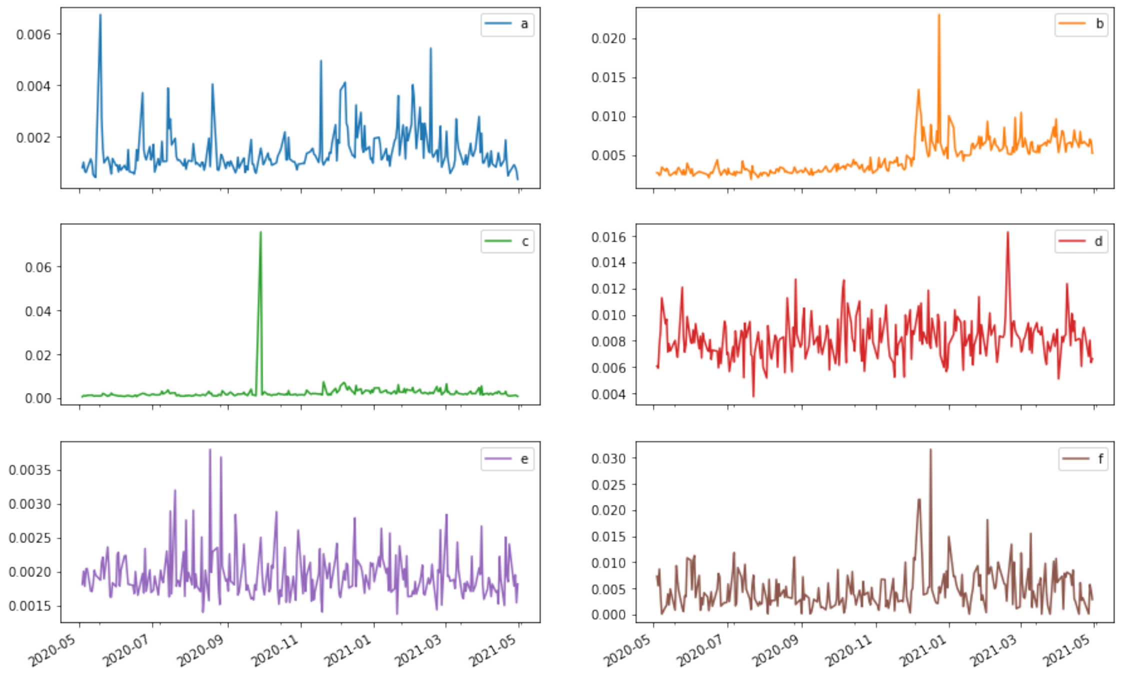
Let's compute daily variance squared:

[19]:
```
daily_var=log_returns.groupby(by='day', as_index=False).var(ddof=0)   #### degrees of␣
 ↪freedom 0 for an easier formula/better interpretation. Using standard ddof=1␣
 ↪wouldn't change the results much.

daily_vol=daily_var.copy()
daily_vol.loc[:,'a':'f']=daily_vol.loc[:,'a':'f'].pow(0.5)

## If we want to use realized volatility. Results seem to be similar.
##log_returns_sq=log_returns.copy()
##log_returns_sq.loc[:,'a':'f']=log_returns_sq.loc[:,'a':'f'].pow(2)
```

```
##daily_vol=log_returns_sq.groupby(by='day', as_index=False).sum()

daily_vol.set_index(np.unique(df.index.date),inplace=True)
daily_var.set_index(np.unique(df.index.date),inplace=True)
```

[20]:
```
daily_vol.loc[:,'a':'f'].plot(subplots=True, layout=(3,2),figsize=(15,10))
```

[20]:
```
array([[<AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>]], dtype=object)
```



I would have expected more persistent behaviour. With the way it looks, I would say that simple moving mean or historic mean might be a good model in most cases, due to lack of aucorrelation (reflected in the lack of consistent intervals of high/low volatility on the graph above).

[21]:
```
daily_vol.head()
```

[21]:
```
               day         a         b         c         d         e         f
2020-05-04  0 days  0.000805  0.002709  0.000563  0.006077  0.001809  0.007281
2020-05-05  1 days  0.001010  0.002678  0.001036  0.005927  0.002000  0.005445
2020-05-06  2 days  0.000673  0.002359  0.001020  0.007701  0.001780  0.008649
2020-05-07  3 days  0.000615  0.002532  0.000923  0.008828  0.002037  0.003588
2020-05-08  4 days  0.000756  0.003424  0.001186  0.011283  0.002041  0.000002
```

[22]:
```
daily_avg=df_5_min.groupby(by='day', as_index=False).mean()
daily_return=log_returns.groupby(by='day', as_index=False).sum()
daily_avg.set_index(np.unique(df.index.date),inplace=True)
daily_return.set_index(np.unique(df.index.date),inplace=True)
```

[23]:
```
daily_vol.corrwith(daily_avg)
```

[23]:
```
a    0.167120
b   -0.744042
c   -0.153837
d   -0.155671
e    0.100551
f   -0.246041
dtype: float64
```

[24]:
```
daily_vol.corrwith(daily_return)
```

[24]:
```
a   -0.129511
b    0.017381
c   -0.862087
d    0.000999
e   -0.152175
f   -0.154090
dtype: float64
```

[25]:
```
daily_vol.corr()
```

[25]:
```
          a         b         c         d         e         f
a  1.000000  0.381033  0.136875  0.141410  0.006151  0.345988
b  0.381033  1.000000  0.126608  0.153634 -0.021272  0.389761
c  0.136875  0.126608  1.000000 -0.052782  0.111674  0.070125
d  0.141410  0.153634 -0.052782  1.000000  0.141014  0.016787
e  0.006151 -0.021272  0.111674  0.141014  1.000000 -0.053884
f  0.345988  0.389761  0.070125  0.016787 -0.053884  1.000000
```

we see that maybe a,b,f are somewhat correlated, but the rest seem to not be.

Need to take care because 'c' correlation with daily return is likely due to the price drop

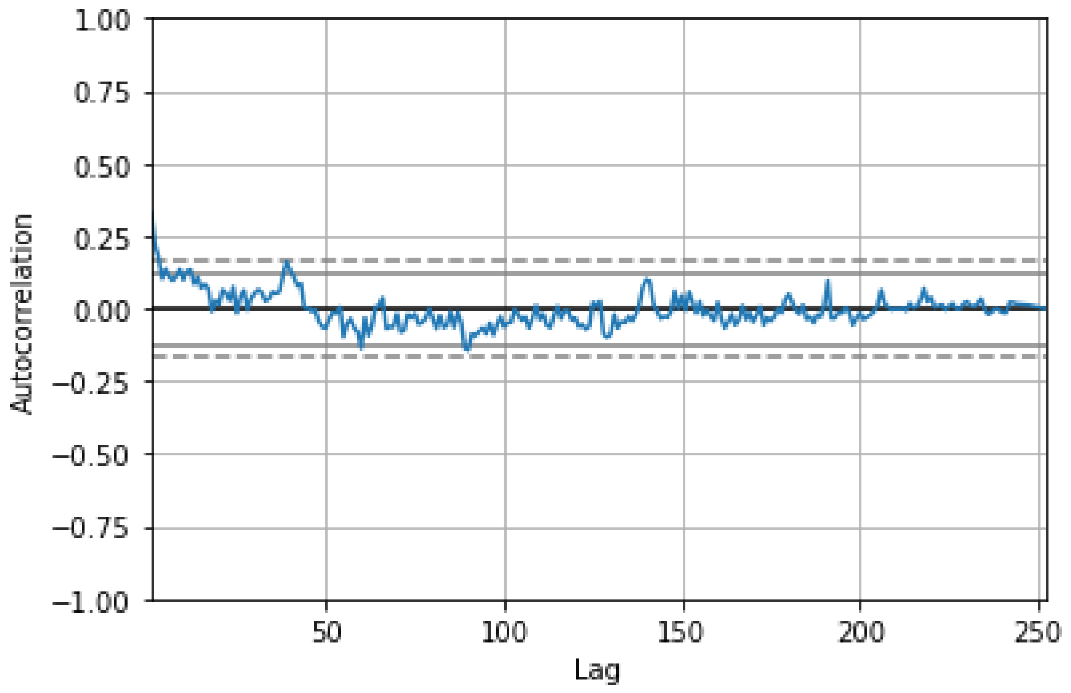
looking at the autocorrelation plots, only stock's ''b' daily variation seems to have some significant autocorrelation. #### Need to clean 'c' data

fig_acorr, axs = plt.subplots(3,2,tight_layout=True, sharex=False,sharey=False,figsize=(15,20)) i=0 j=0 for stock in stocks: axs[i][j].set_title("Stock"+stock+": daily variance autocorrelation", fontsize=8) axs[i][j].acorr(daily_var[stock], normed=False,maxlags=250) if j==1: j=0 i+=1 else: j+=1

```
###### aparently matplotlib acorr doesn't work..
```

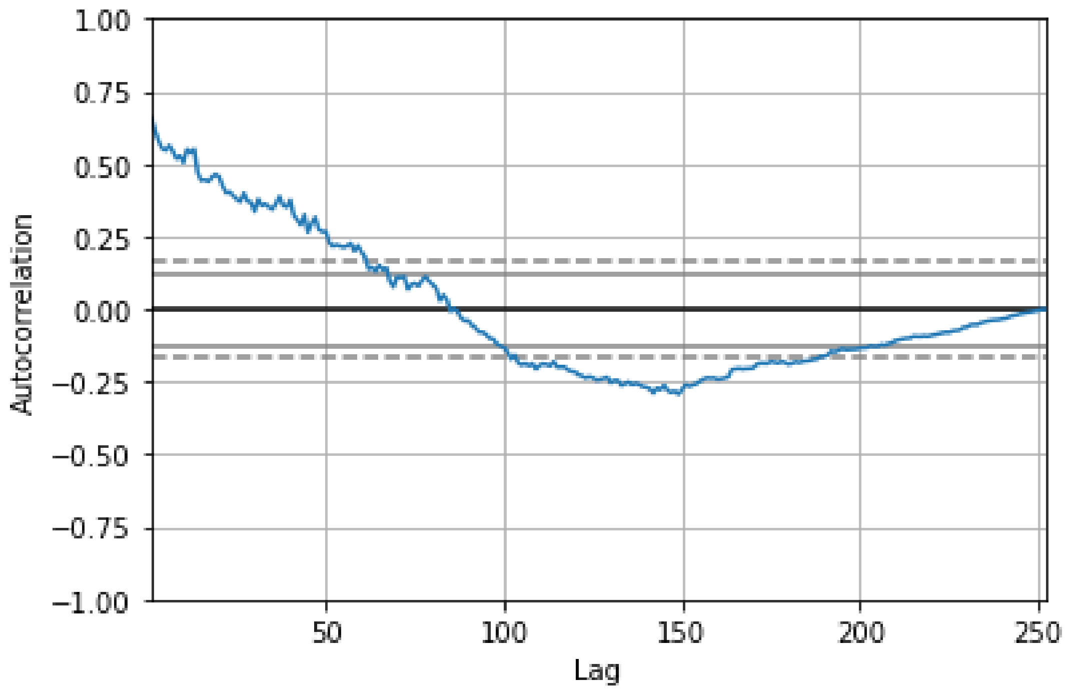
[26]:
```
pd.plotting.autocorrelation_plot(daily_vol['a'])
```

[26]:
```
<AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>
```



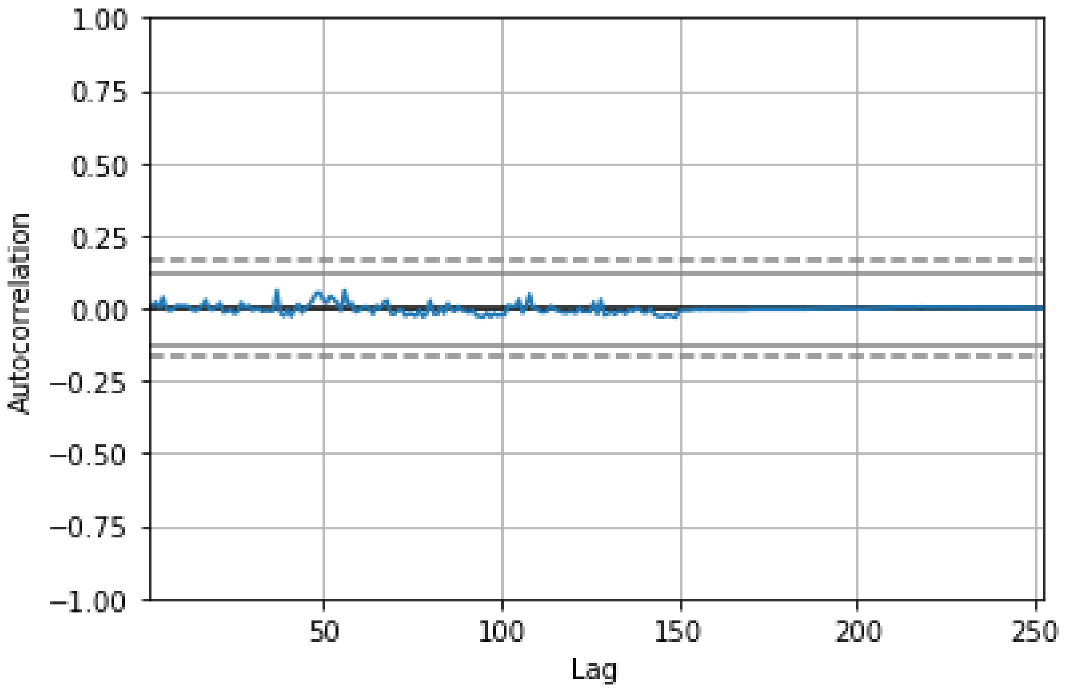
[27]:
```
pd.plotting.autocorrelation_plot(daily_vol['b'])
```

[27]:
```
<AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>
```



[28]:
```
pd.plotting.autocorrelation_plot(daily_vol['c'])
```

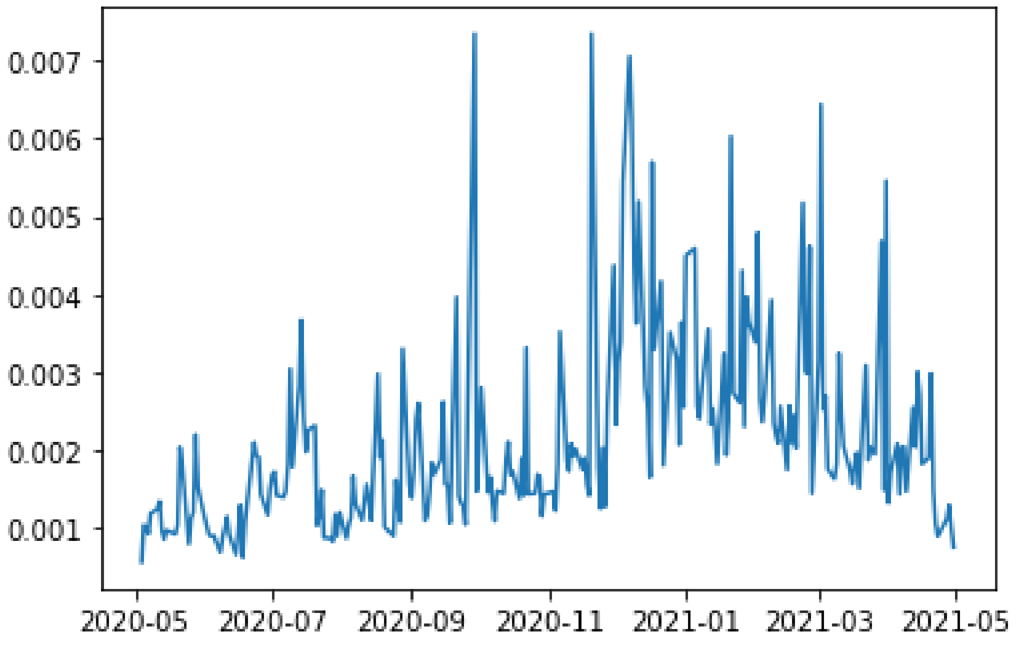
[28]: <AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>



[29]:
```
daily_vol['c_no_extremes']=daily_vol['c']
daily_vol.loc[daily_vol['c_no_extremes'].idxmax(),'c_no_extremes']=␣
 ↪daily_vol['c_no_extremes'].nlargest(2).iloc[-1] ### or =np.nan if we want to remove␣
 ↪it.
stocks_and_c_no_extremes=stocks.copy()
stocks_and_c_no_extremes.append('c_no_extremes')
```

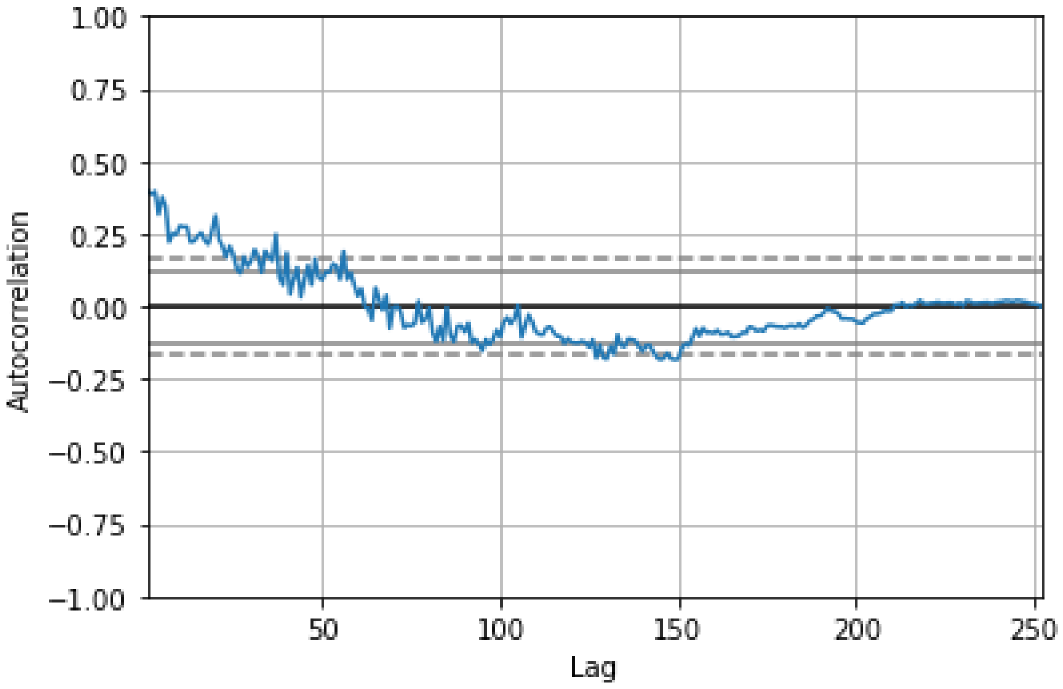
[30]:
```
daily_vol['c_no_extremes'].plot()
```

[30]: <AxesSubplot:>



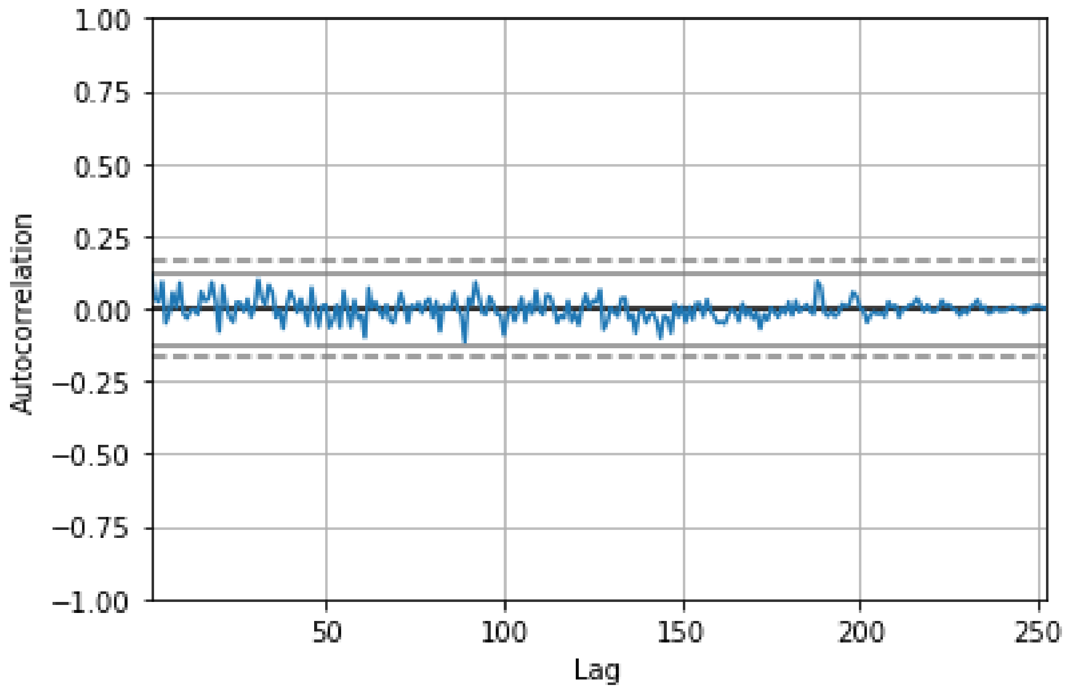
```
[31]: pd.plotting.autocorrelation_plot(daily_vol['c_no_extremes'])
```

```
[31]: <AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>
```



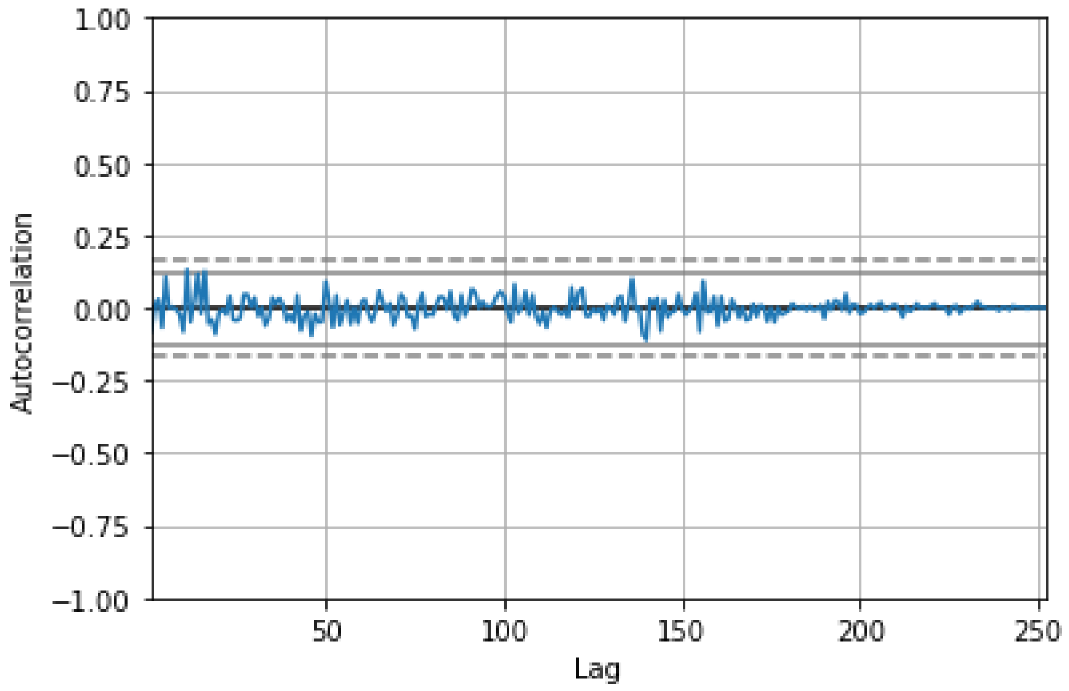
```
[32]: pd.plotting.autocorrelation_plot(daily_vol['d'])
```

```
[32]: <AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>
```



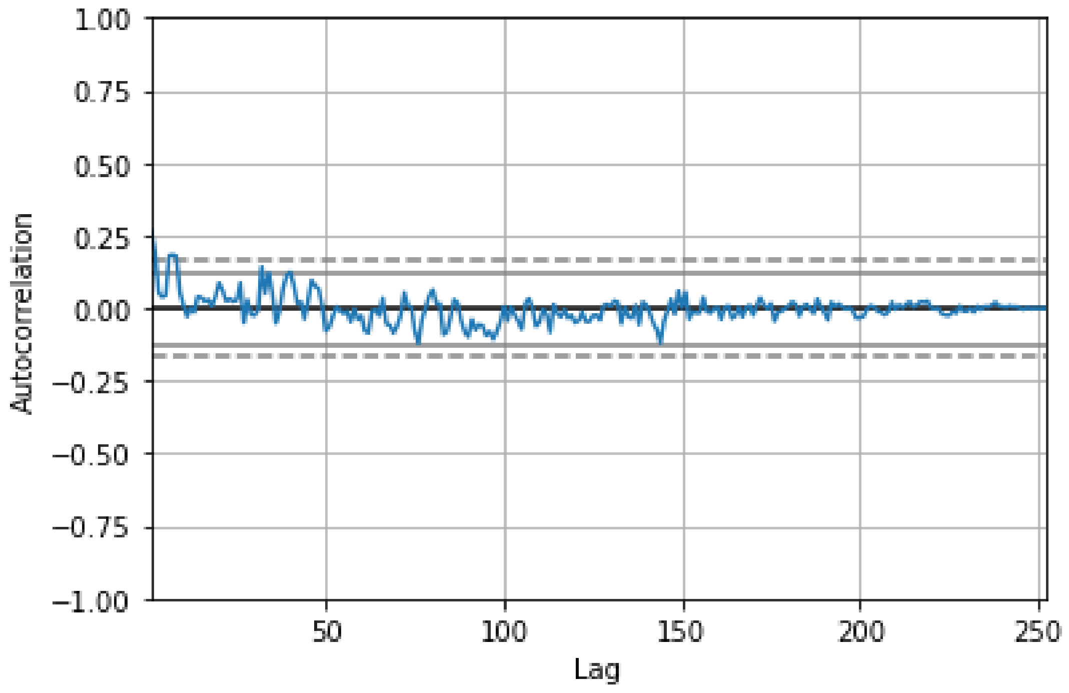
```
[33]: pd.plotting.autocorrelation_plot(daily_vol['e'])
```

```
[33]: <AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>
```



```
[34]: pd.plotting.autocorrelation_plot(daily_vol['f'])
```

```
[34]: <AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>
```



```
[35]: for stock in stocks:
          daily_vol[stock+'_prev']=daily_vol[stock].shift(periods=-1)
      daily_vol.head()
```

```
[35]:               day         a         b         c         d         e         f  \
      2020-05-04 0 days  0.000805  0.002709  0.000563  0.006077  0.001809  0.007281
      2020-05-05 1 days  0.001010  0.002678  0.001036  0.005927  0.002000  0.005445
      2020-05-06 2 days  0.000673  0.002359  0.001020  0.007701  0.001780  0.008649
      2020-05-07 3 days  0.000615  0.002532  0.000923  0.008828  0.002037  0.003588
      2020-05-08 4 days  0.000756  0.003424  0.001186  0.011283  0.002041  0.000002

                  c_no_extremes    a_prev    b_prev    c_prev    d_prev    e_prev  \
      2020-05-04       0.000563  0.001010  0.002678  0.001036  0.005927  0.002000
      2020-05-05       0.001036  0.000673  0.002359  0.001020  0.007701  0.001780
      2020-05-06       0.001020  0.000615  0.002532  0.000923  0.008828  0.002037
      2020-05-07       0.000923  0.000756  0.003424  0.001186  0.011283  0.002041
      2020-05-08       0.001186  0.001136  0.002921  0.001242  0.009302  0.001709

                    f_prev
      2020-05-04  0.005445
      2020-05-05  0.008649
      2020-05-06  0.003588
      2020-05-07  0.000002
      2020-05-08  0.001352
```

```
[36]: def model_historic_mean(df, stock, lookback_interval=10, prediction_interval=30):
          return ((df[stock]**2).mean())**(0.5)
```

```
#def model_historic_mean_exclude_extreme(df, stock, lookback_interval=10,␣
↪prediction_interval=30):
    #return df.loc[df[stock]<=df.nlargest(1,stock).iloc[-1,:][stock],stock].mean()
def model_moving_avg(df, stock, lookback_interval=10, prediction_interval=30):
    return (((df.loc[df.index+pd.Timedelta(days=lookback_interval)>df.
↪index[-1],stock])**2).mean())**(0.5)

def model_exponential_decay(df, stock, lookback_interval=30, prediction_interval=30):
                            #### lookback interval=number of days after which the␣
↪weight is halfed (i.e. gamma=1/2^(1/lookback_interval))
    gamma=2**(-1/lookback_interval)
    gamma_ar=gamma**(len(df.index)-np.arange(len(df.index))-1)
    return (((df[stock]**2)*gamma_ar).sum()/gamma_ar.sum())**0.5

##def model_exponential_decay_exclude_extreme(df, stock, lookback_interval=30,␣
↪prediction_interval=30):
                            #### lookback interval=number of days after which the␣
↪weight is halfed (i.e. gamma=1/2^(1/lookback_interval))
    ##df_exclude_extremes=df[stock]
```

[37]:
```
def test_model(model, df, prediction_interval=30,stocks_f=df.columns, min_days=60,␣
↪error_method='Raw error', lookback_interval=10):
                           #### I'm mainly planning to use meam squared error to␣
↪compare models.
                           #### Other error function: 'Relative error' (error/observed␣
↪value)
                           #### prediction_interval is the number of days to predict␣
↪ahead. We can do it in months (using numpy datetime instead of pandas),
                           #### but it doesn't really make sense, since we set up an␣
↪arbitrary starting date anyway.
                           #### stocks_f is the stocks we want to predict
                           #### min_days is the minimal number of days required to␣
↪give an estimate

    number_of_months=int((df.index[-1]-df.index[0]-pd.Timedelta(days=min_days))/pd.
↪Timedelta(days=prediction_interval))
    prediction_error=pd.DataFrame(columns=stocks_f, index=np.arange(number_of_months))
    for stock in stocks_f:
        day=df.index[-1]
        i=0
        while day>df.index[0]+pd.Timedelta(days=min_days):
            first_day=df.loc[df.index+pd.Timedelta(days=prediction_interval)>day].
↪index[0]
            pred=model(df=df.loc[df.index<first_day], stock=stock,␣
↪prediction_interval=prediction_interval, lookback_interval=lookback_interval)
            obs=(df.loc[((df.index>=first_day) & (df.index<=day)),stock].pow(2).
↪mean())**(1/2)
            if error_method=='Raw error':
                prediction_error.loc[number_of_months-i-1,stock]=pred-obs
            if error_method=='Relative error':
                prediction_error.loc[number_of_months-i-1,stock]=(pred-obs)/obs
            i+=1
            day=df.loc[df.index<first_day].index[-1]

    return prediction_error
```

[38]:
```
def total_error(model, df, prediction_interval=30,stocks_f=df.columns, min_days=60,␣
 ↪error_method='Raw error', lookback_interval=10):
        if error_method=='Raw error':
            return test_model(model, df, prediction_interval,stocks_f, min_days,␣
 ↪error_method, lookback_interval).abs().mean()/df[stocks_f].mean()
        elif error_method=='Relative error':
            return test_model(model, df, prediction_interval,stocks_f, min_days,␣
 ↪error_method, lookback_interval).abs().mean()
```

[39]:
```
test_model(model_historic_mean, daily_vol, stocks_f=stocks_and_c_no_extremes,␣
 ↪error_method='Relative error')
```

[39]:
```
            a           b          c           d            e          f  \
0 -0.00293818 -0.00517095  -0.329313   0.0789854   -0.0626344 -0.0731401
1    0.107339  -0.0213722 -0.0856494  -0.0364676    -0.106523   0.156894
2    0.529411  -0.0337394  -0.907664  -0.0346531    0.0252889   0.725525
3    0.186096   -0.191431    3.27546   -0.107355     0.045883   0.137331
4   -0.175504   -0.218571    1.62861 -0.00426324    0.0541135   0.133336
5   -0.356989   -0.635396   0.659223  -0.0519551  -0.00896103  -0.585716
6   -0.114819   -0.304174   0.907513  -0.0253584   -0.0170886 -0.0246435
7   -0.324286   -0.321854   0.972449   -0.109443    0.0573655  -0.229762
8    0.104559   -0.295417   0.992486   0.0335496 -0.000240755 -0.0967581
9    0.647961   -0.253693     1.9778  0.00485093      0.07273   0.184878

  c_no_extremes
0     -0.329313
1    -0.0856494
2     -0.366788
3    -0.0141376
4     -0.334999
5     -0.516197
6     -0.305647
7     -0.207785
8     -0.143921
9      0.347059
```

we see that expectedly, simple historic mean assigns too much value to hitoric behaviour (e.g. last prediction of 'a' is)

We also see (again expectedly), that 'c' predictions are extremely unnacurate after big drop.

[40]:
```
error_df=pd.DataFrame(index=stocks_and_c_no_extremes)
```

[41]:
```
error_df['tae HM']=total_error(model_historic_mean, daily_vol,␣
 ↪stocks_f=stocks_and_c_no_extremes, error_method='Raw error')
total_error(model_historic_mean, daily_vol, stocks_f=stocks_and_c_no_extremes,␣
 ↪error_method='Raw error')
```

```
[41]: a                 0.272356
      b                 0.323691
      c                 1.707013
      d                 0.050792
      e                 0.046338
      f                 0.305863
      c_no_extremes     0.341306
      dtype: float64
```

```
[42]: error_df['tre HM']=total_error(model_historic_mean, daily_vol,␣
       ↪stocks_f=stocks_and_c_no_extremes, error_method='Relative error')
      total_error(model_historic_mean, daily_vol, stocks_f=stocks_and_c_no_extremes,␣
       ↪error_method='Relative error')
```

```
[42]: a                 0.254990
      b                 0.228082
      c                 1.173616
      d                 0.048688
      e                 0.045083
      f                 0.234798
      c_no_extremes     0.265150
      dtype: float64
```

```
[43]: test_model(model_moving_avg, daily_vol, stocks_f=stocks_and_c_no_extremes,␣
       ↪error_method='Relative error',lookback_interval=20)
```

```
[43]:            a            b            c            d          e          f  \
      0 -0.0608257  0.000982442    -0.248546   0.00851132  -0.105114  -0.301841
      1    0.20586  -0.00226074     0.137298   -0.0710961 -0.0270785  0.0727083
      2   0.541749    0.0148056    -0.890304    0.0240511   0.142981   0.648776
      3  -0.130605    -0.148864      10.8678    -0.129786 -0.0102092  -0.319946
      4  -0.244163   -0.0719929    -0.325594    0.0471435  0.0288645 -0.0218167
      5  -0.168977    -0.549971    -0.275533   -0.0826153  -0.049499  -0.626522
      6  0.0250287     0.427036    0.0389395   0.00256096 -0.0022142   0.549448
      7   -0.20668    -0.122522    0.0655597   -0.0981421  0.0388196  -0.346253
      8   0.382027   -0.0528898   0.00484412     0.191211 -0.0185071  0.0289082
      9   0.485746    0.0086495     0.443038   -0.0184267  0.0349126   0.139633

        c_no_extremes
      0     -0.248546
      1      0.137298
      2      -0.24774
      3      0.543859
      4     -0.325594
      5     -0.275533
      6     0.0389395
      7     0.0655597
      8    0.00484412
      9      0.443038
```

```
[44]: error_df['tae MM (d=10)']=total_error(model_moving_avg, daily_vol,␣
       ↪stocks_f=stocks_and_c_no_extremes, error_method='Raw error',lookback_interval=10)
```

```
total_error(model_moving_avg, daily_vol, stocks_f=stocks_and_c_no_extremes,␣
 ↪error_method='Raw error',lookback_interval=10)
```

[44]:
```
a                0.210867
b                0.238068
c                1.987098
d                0.071880
e                0.061724
f                0.363184
c_no_extremes    0.386297
dtype: float64
```

[45]:
```
error_df['tre MM (d=10)']=total_error(model_moving_avg, daily_vol,␣
 ↪stocks_f=stocks_and_c_no_extremes, error_method='Relative␣
 ↪error',lookback_interval=10)
total_error(model_moving_avg, daily_vol, stocks_f=stocks_and_c_no_extremes,␣
 ↪error_method='Relative error',lookback_interval=10)
```

[45]:
```
a                0.199707
b                0.172459
c                1.844812
d                0.069600
e                0.060141
f                0.299144
c_no_extremes    0.355020
dtype: float64
```

[46]:
```
test_model(model_exponential_decay, daily_vol, stocks_f=stocks_and_c_no_extremes,␣
 ↪error_method='Relative error',lookback_interval=30)
```

[46]:
```
            a          b          c           d          e          f  \
0   -0.026053 -0.0038559   -0.31125   0.0692829 -0.0662487 -0.0938957
1   0.0861399 -0.0230591 -0.0317467  -0.0540205 -0.0953491    0.14033
2    0.488896 -0.0287065  -0.902123  -0.0326056  0.0591802   0.676662
3   0.0795092  -0.180879    5.74592   -0.102791  0.0589844  0.0101115
4   -0.253212  -0.169486    2.47975    0.025019  0.0518915  0.0158783
5    -0.34988  -0.596792   0.884708  -0.0334712 -0.0215591  -0.622937
6  0.00853906  0.0047255   0.897073 -0.00216325 -0.0223655    0.26245
7   -0.238454 -0.0763639   0.735849  -0.0893363  0.0566421 -0.0869846
8    0.313218 -0.0637746   0.569231   0.0884321 -0.0124247  0.0633646
9    0.814276 -0.0231018    1.13288   0.0282352  0.0643718   0.346558

  c_no_extremes
0      -0.31125
1    -0.0317467
2      -0.32879
3      0.157334
4     -0.265495
5     -0.424244
6    -0.0810338
7     0.0282009
8     0.0685374
9      0.614515
```

[47]:
```
error_df['tae ED (d=30)']=total_error(model_exponential_decay, daily_vol,␣
  ↪stocks_f=stocks_and_c_no_extremes, error_method='Raw error',lookback_interval=30)
total_error(model_exponential_decay, daily_vol, stocks_f=stocks_and_c_no_extremes,␣
  ↪error_method='Raw error',lookback_interval=30)
```

[47]:
```
a                0.275881
b                0.167944
c                1.863170
d                0.054210
e                0.052215
f                0.311063
c_no_extremes    0.270812
dtype: float64
```

[48]:
```
error_df['tre ED (d=30)']=total_error(model_exponential_decay, daily_vol,␣
  ↪stocks_f=stocks_and_c_no_extremes, error_method='Relative␣
  ↪error',lookback_interval=30)
total_error(model_exponential_decay, daily_vol, stocks_f=stocks_and_c_no_extremes,␣
  ↪error_method='Relative error',lookback_interval=30)
```

[48]:
```
a                0.265818
b                0.117074
c                1.369053
d                0.052536
e                0.050902
f                0.231917
c_no_extremes    0.231115
dtype: float64
```

[49]:
```
error_df
```

[49]:
```
                 tae HM    tre HM  tae MM (d=10)  tre MM (d=10)  \
a              0.272356  0.254990       0.210867       0.199707
b              0.323691  0.228082       0.238068       0.172459
c              1.707013  1.173616       1.987098       1.844812
d              0.050792  0.048688       0.071880       0.069600
e              0.046338  0.045083       0.061724       0.060141
f              0.305863  0.234798       0.363184       0.299144
c_no_extremes  0.341306  0.265150       0.386297       0.355020

               tae ED (d=30)  tre ED (d=30)
a                   0.275881       0.265818
b                   0.167944       0.117074
c                   1.863170       1.369053
d                   0.054210       0.052536
e                   0.052215       0.050902
f                   0.311063       0.231917
c_no_extremes       0.270812       0.231115
```

[50]:
```
predictions=pd.DataFrame(index=stocks)
predictions['prediction']=np.nan
```

```
[51]: for stock in ['a','d','e','f']:
          predictions.loc[stock,'prediction']=model_historic_mean(daily_vol, stock=stock)
      predictions.
       ↪loc['b','prediction']=model_exponential_decay(daily_vol,stock='b',lookback_interval=30)
      predictions.
       ↪loc['c','prediction']=model_exponential_decay(daily_vol,stock='c_no_extremes',lookback_interval=30)
      predictions=predictions*(253**(1/2))*100
      predictions
```

```
[51]:    prediction
      a    2.638384
      b   10.485158
      c    4.167963
      d   13.246544
      e    3.151951
      f    9.992446
```

```
[52]: std_dev=pd.DataFrame(index=stocks)
      std_dev['std']=np.nan
      std_dev.loc[['a','d','e','f'],'std']=(test_model(model_historic_mean, daily_vol,␣
       ↪stocks_f=stocks_and_c_no_extremes, error_method='Raw error').var()).
       ↪loc[['a','d','e','f']]
      std_dev.loc['b','std']=(test_model(model_exponential_decay, daily_vol,␣
       ↪stocks_f=stocks_and_c_no_extremes, error_method='Raw error').var()).loc['b']
      std_dev.loc['c','std']=(test_model(model_exponential_decay, daily_vol,␣
       ↪stocks_f=stocks_and_c_no_extremes, error_method='Raw error').var()).
       ↪loc['c_no_extremes']
      std_dev=(std_dev**(0.5))*(253**(1/2))*100
      std_dev
```

```
[52]:        std
      a  0.792941
      b  2.698588
      c  1.180128
      d  0.792859
      e  0.186989
      f  3.899750
```

```
[53]: predictions['prediction']-2*std_dev['std']
```

```
[53]: a     1.052503
      b     5.087983
      c     1.807707
      d    11.660826
      e     2.777973
      f     2.192945
      dtype: float64
```

```
[54]: predictions['prediction']+2*std_dev['std']
```

```
[54]: a     4.224265
      b    15.882333
```

```
c     6.528220
d    14.832262
e     3.525930
f    17.791946
dtype: float64
```