05. Fractionally Differentiated Features

November 2, 2018

Table of Contents

6.3 (c) Form labels using the triple-barrier method, with symmetric horizontal barriers of twice the daily standard deviation, and a vertical barrier of 5 days

6.4 (d) Fit a bagging classifier of decision trees where:

6.4.1 (i) The observed features are bootstrapped using the sequential method from chapter 4.

6.4.2 (ii) On each bootstrapped sample, sample weights are determined using the techniques from Chapter 4

```
In [1]: %load_ext watermark
        %watermark

        %load_ext autoreload
        %autoreload 2
        # import standard libs
        import warnings
        warnings.filterwarnings("ignore")
        from IPython.display import display
        from IPython.core.debugger import set_trace as bp
        from pathlib import PurePath, Path
        import sys
        import time
        from collections import OrderedDict as od
        import re
        import os
        import json
        os.environ['THEANO_FLAGS'] = 'device=cpu,floatX=float32'

        # import python scientific stack
        import pandas as pd
        pd.set_option('display.max_rows', 100)
        from dask import dataframe as dd
        from dask.diagnostics import ProgressBar
        pbar = ProgressBar()
        pbar.register()
        import multiprocessing as mp
        from multiprocessing import cpu_count
        import numpy as np
        import scipy.stats as stats
        import statsmodels.api as sm
        import numba as nb
        import math
        import pymc3 as pm
        from theano import shared, theano as tt

        # import visual tools
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        import matplotlib.gridspec as gridspec
        %matplotlib inline
```

```
import seaborn as sns
import plotnine as pn

plt.style.use('seaborn-talk')
plt.style.use('bmh')
#plt.rcParams['font.family'] = 'DejaVu Sans Mono'
plt.rcParams['font.size'] = 9.5
plt.rcParams['font.weight'] = 'medium'
plt.rcParams['figure.figsize'] = 10,7
blue, green, red, purple, gold, teal = sns.color_palette('colorblind', 6)

# import util libs
import pyarrow as pa
import pyarrow.parquet as pq
from tqdm import tqdm, tqdm_notebook
import missingno as msno

from src.utils.utils import *
import src.features.bars as brs
import src.features.snippets as snp

import copyreg, types
copyreg.pickle(types.MethodType,snp._pickle_method,snp._unpickle_method)
RANDOM_STATE = 777

pdir = get_relative_project_dir('Adv_Fin_ML_Exercises')
data_dir = pdir/'data'/'processed'

print()
%watermark -p pandas,numpy,numba,pymc3,sklearn,statsmodels,scipy,matplotlib,seaborn
```

```
2018-10-18T16:55:10-06:00

CPython 3.6.6
IPython 6.5.0

compiler   : GCC 7.2.0
system     : Linux
release    : 4.15.0-36-generic
machine    : x86_64
processor  : x86_64
CPU cores  : 12
interpreter: 64bit

pandas 0.23.4
numpy 1.14.6
numba 0.41.0dev0+75.gdb0256a70
pymc3 3.5
```

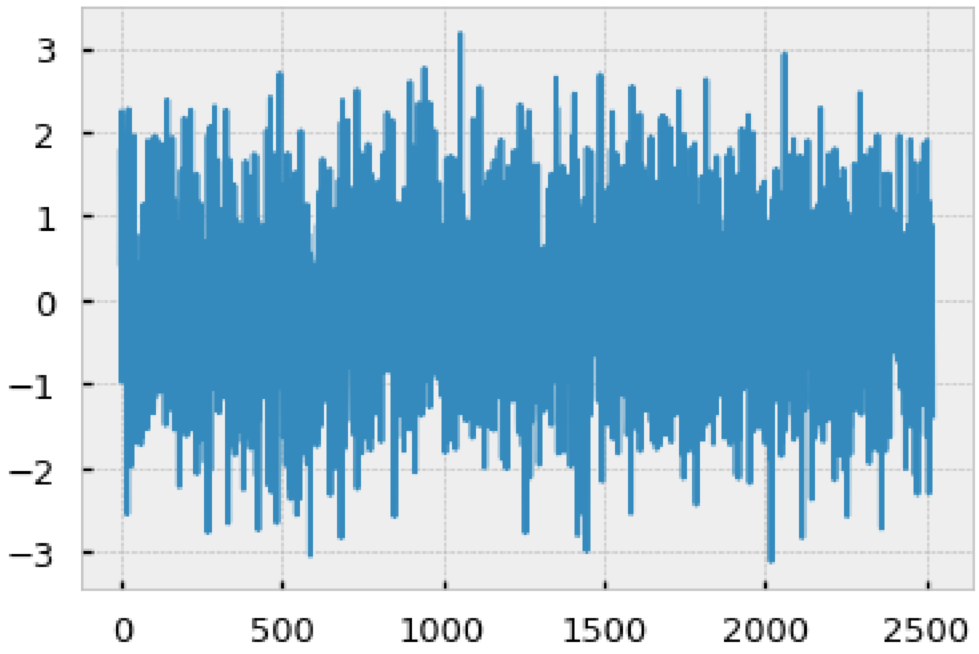
```
sklearn 0.20.0
statsmodels 0.9.0
scipy 1.1.0
matplotlib 3.0.0
seaborn 0.9.0
```

1 Chapter 5

1.1 [5.1] Generate a time series from an IID Gaussian random process. This is a memory-less, stationary series:

```
In [2]: np.random.seed(0)

        N = 252*10
        s = pd.Series(np.random.randn(N))
        s.plot()
```

```
Out[2]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4588a3cf28>
```



1.1.1 (a) Compute the ADF statistic on this series. What is the p-value?

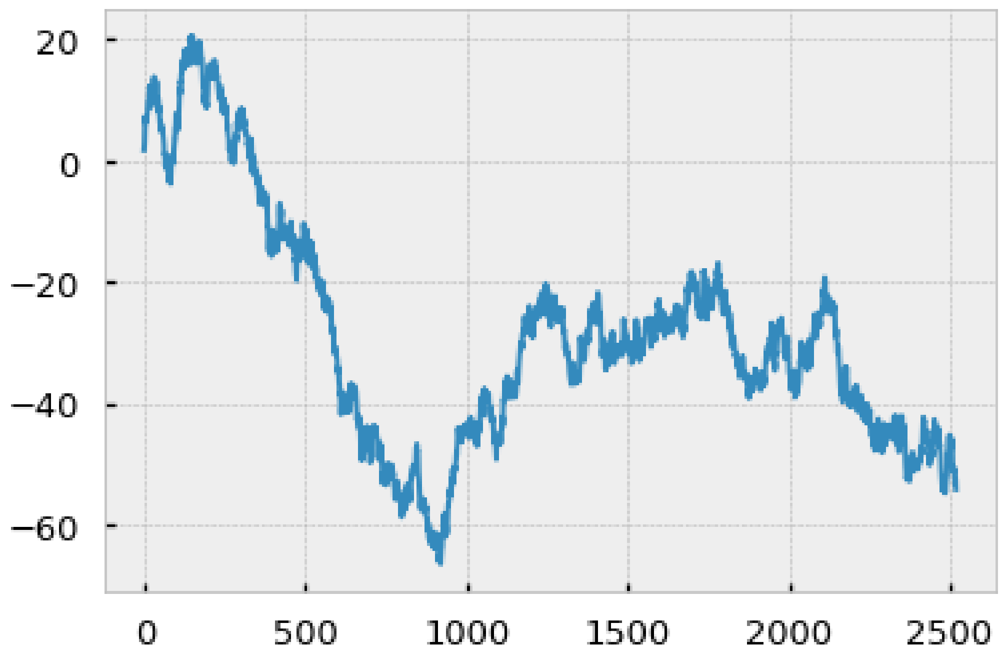
```
In [3]: adf = lambda s: sm.tsa.stattools.adfuller(s)
        p_val = lambda s: sm.tsa.stattools.adfuller(s)[1]
        res = adf(s); p = res[1]
        res, p
```

```
Out[3]: ((-50.80332180276013,
          0.0,
          0,
          2519,
          {'1%': -3.4329486408391174,
           '5%': -2.8626880695259413,
           '10%': -2.567381161224712},
          6950.968131407137),
         0.0)
```

1.1.2 (b) Compute the cumulative sum of the observations. This is a non-stationary series w/o memory.

```
In [4]: cmsm = pd.Series(s).cumsum()
        cmsm.plot()
```

```
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4531e30f60>
```



(i) What is the order of integration of this cumulative series?

```
In [5]: orders = [0, 1, 2, 3, 4]
        for o in orders:
            diff_ = np.diff(cmsm,o)
            print('='*27)
            print(f'order: {o}, pVal: {p_val(diff_)}')
```

```
===========================
order: 0, pVal: 0.5704444806659968
===========================
order: 1, pVal: 0.0
===========================
order: 2, pVal: 7.347529850653773e-30
===========================
order: 3, pVal: 0.0
===========================
order: 4, pVal: 0.0
```

(ii) Compute the ADF statistic on this series. What is the p-value?

```
In [6]: p_val(cmsm)
```

```
Out[6]: 0.5704444806659968
```

1.1.3 (c) Differentiate the series twice. What is the p-value of this over-differentiated series?

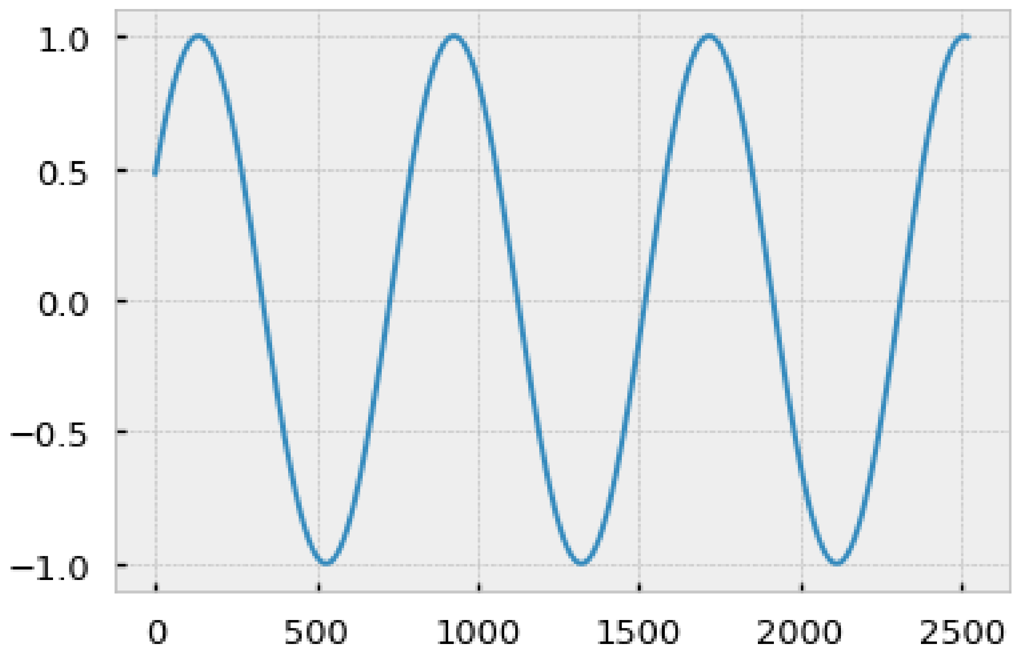
```
In [7]: diff_ = np.diff(cmsm,2)
        p_val(diff_)
```

```
Out[7]: 7.347529850653773e-30
```

1.2 [5.2] Generate a time series that follows a sinusoidal function. This is a stationary series with memory.

```
In [8]: np.random.seed(0)

        rand = np.random.random(N)

        idx = np.linspace(0,10, N)
        s = pd.Series(1*np.sin(2.*idx + .5))
        s.plot()
```

```
Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4531db3be0>
```

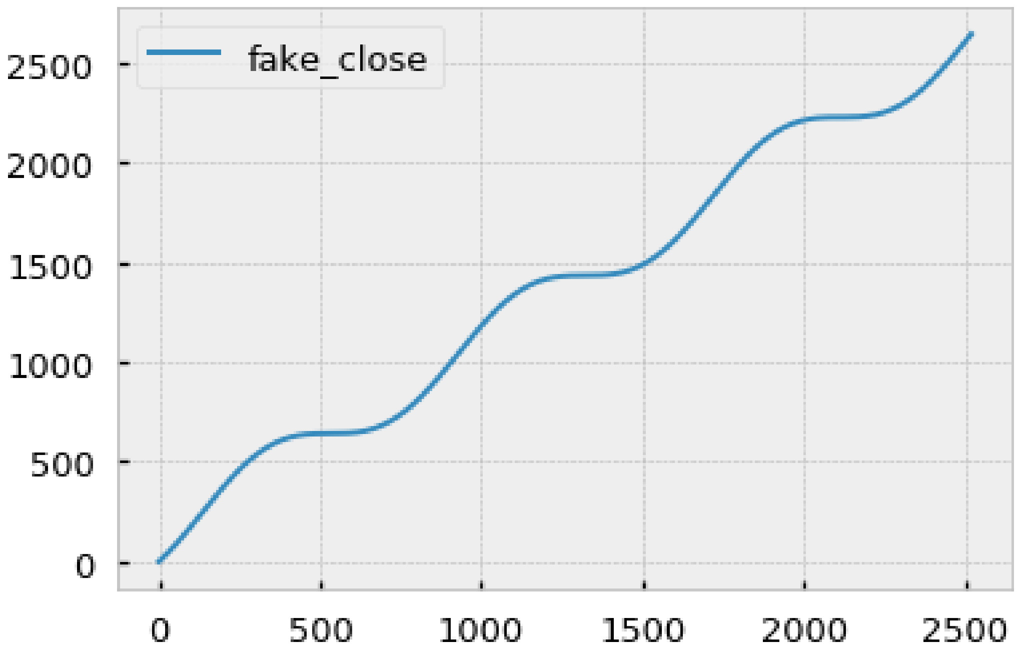


1.2.1 (a) Compute the ADF statistic on this series. What is the p-value?

```
In [9]: p_val(s)

Out[9]: 0.0
```

1.2.2 (b) Shift every observation by the same positive value. Compute the cumulative sum of the observations. This is a non-stationary series with memory.

```
In [10]: s_ = (s + 1).cumsum().rename('fake_close').to_frame()
         s_.plot()

Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4532cf7fd0>
```



(i) Compute the ADF statistic on this series. What is the p-value?

```
In [11]: adf(s_['fake_close'].dropna()), p_val(s_['fake_close'])
```

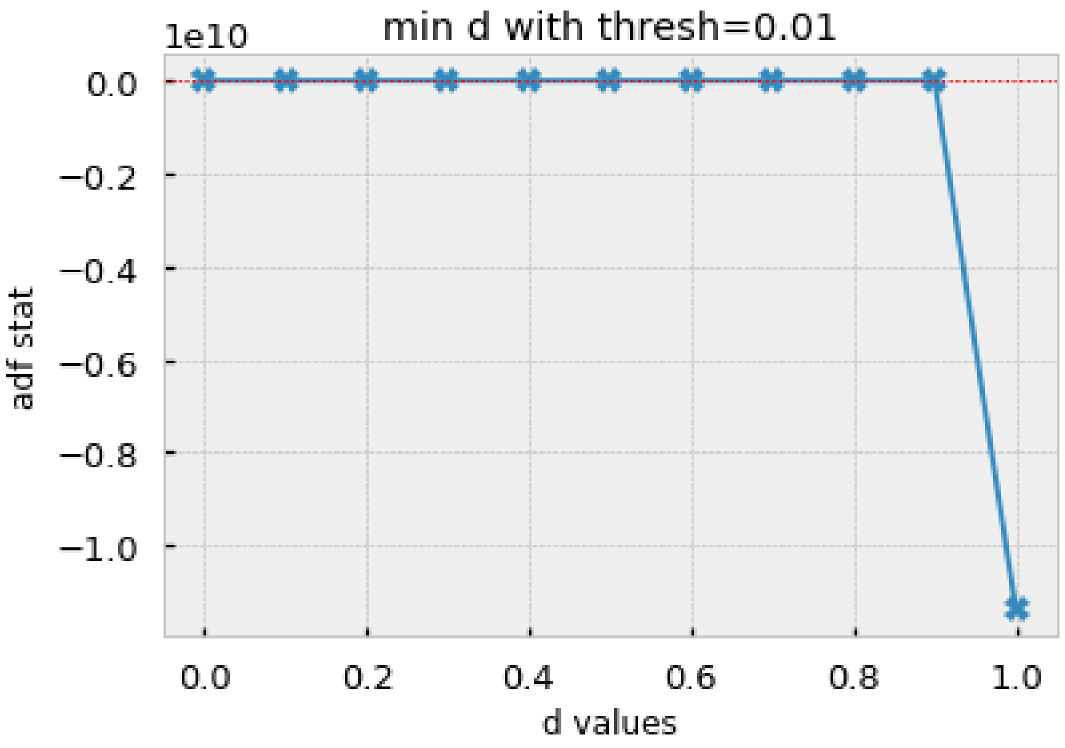
```
Out[11]: ((-0.19138785660460378,
           0.9395404066056363,
           27,
           2492,
           {'1%': -3.432976825339513,
            '5%': -2.862700515844509,
            '10%': -2.5673877878037974},
           -142670.50155880928),
          0.9395404066056363)
```

(ii) Apply an expanding window fracdiff, with $\tau = 1E - 2$. For what minimum d value do you get a p-value below 5%?

```
In [12]: def getWeights(d,size):
             # thres>0 drops insignificant weights
             w=[1.]
             for k in range(1,size):
                 w_ = -w[-1]/k*(d-k+1)
                 w.append(w_)
             w=np.array(w[::-1]).reshape(-1,1)
             return w

         #getWeights(0.1, s_.shape[0])
```

```
In [13]: def fracDiff(series, d, thres=0.01):
             '''
             Increasing width window, with treatment of NaNs
             Note 1: For thres=1, nothing is skipped
             Note 2: d can be any positive fractional, not necessarily
                 bounded between [0,1]
             '''
             #1) Compute weights for the longest series
             w=getWeights(d, series.shape[0])
             #bp()
             #2) Determine initial calcs to be skipped based on weight-loss threshold
             w_=np.cumsum(abs(w))
             w_ /= w_[-1]
             skip = w_[w_>thres].shape[0]
             #3) Apply weights to values
             df={}
             for name in series.columns:
                 seriesF, df_=series[[name]].fillna(method='ffill').dropna(), pd.Series()
                 for iloc in range(skip, seriesF.shape[0]):
                     loc=seriesF.index[iloc]
                     test_val = series.loc[loc,name] # must resample if duplicate index
                     if isinstance(test_val, (pd.Series, pd.DataFrame)):
                         test_val = test_val.resample('1m').mean()
                     if not np.isfinite(test_val).any(): continue # exclude NAs
                     try:
                         df_.loc[loc]=np.dot(w[-(iloc+1):,:].T, seriesF.loc[:loc])[0,0]
                     except:
                         continue
                 df[name]=df_.copy(deep=True)
             df=pd.concat(df,axis=1)
             return df

In [14]: cols = ['adfStat','pVal','lags','nObs','95% conf']#,'corr']
         out = pd.DataFrame(columns=cols)
         for d in np.linspace(0,1,11):
             try:
                 df0 = fracDiff(s_,d)
                 df0 = sm.tsa.stattools.adfuller(df0['fake_close'],maxlag=1,regression='c',autol
                 out.loc[d]=list(df0[:4])+[df0[4]['5%']]
             except:
                 break

         f,ax=plt.subplots()
         out['adfStat'].plot(ax=ax, marker='X')
         ax.axhline(out['95% conf'].mean(),lw=1,color='r',ls='dotted')
         ax.set_title('min d with thresh=0.01')
         ax.set_xlabel('d values')
         ax.set_ylabel('adf stat');
```

```
        display(out)

            adfStat          pVal  lags    nObs  95% conf
0.0  2.833609e+00  1.000000e+00   1.0  2517.0 -2.862689
0.1  8.870880e+00  1.000000e+00   1.0   761.0 -2.865345
0.2 -7.366367e+00  9.213847e-11   1.0   963.0 -2.864546
0.3 -2.267608e+01  0.000000e+00   1.0  1357.0 -2.863672
0.4 -2.259792e+01  0.000000e+00   1.0  1821.0 -2.863128
0.5 -3.781556e+01  0.000000e+00   1.0  2188.0 -2.862862
0.6 -4.388734e+01  0.000000e+00   1.0  2385.0 -2.862753
0.7 -6.322546e+01  0.000000e+00   1.0  2466.0 -2.862713
0.8 -7.371512e+01  0.000000e+00   1.0  2497.0 -2.862698
0.9 -4.877829e+01  0.000000e+00   1.0  2510.0 -2.862692
1.0 -1.135810e+10  0.000000e+00   1.0  2516.0 -2.862689
```



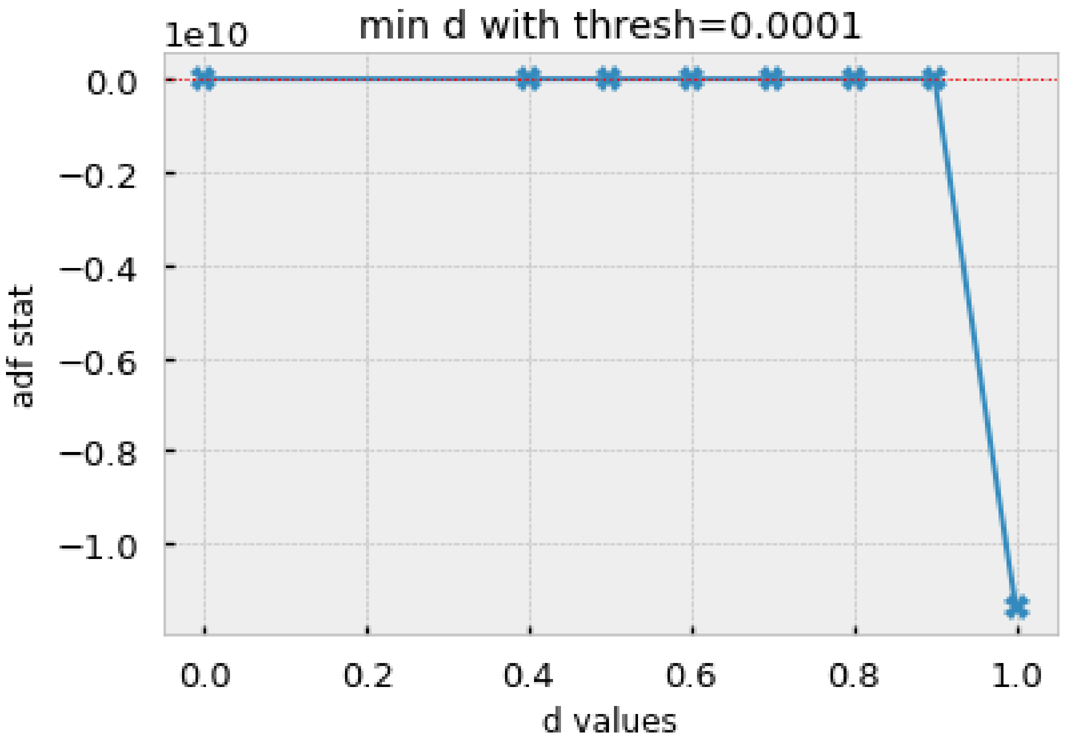
(iii) Apply FFD with $\tau = 1E-5$. For what minimum d value do you get a p-value below 5%

```
In [15]: cols = ['adfStat','pVal','lags','nObs','95% conf']#,'corr']
         out = pd.DataFrame(columns=cols)
         for d in np.linspace(0,1,11):
             try:
                 df0 = fracDiff(s_,d,thres=1e-5)
```

```
                df0 = sm.tsa.stattools.adfuller(df0['fake_close'],maxlag=1,regression='c',autol
                out.loc[d]=list(df0[:4])+[df0[4]['5%']]
            except Exception as e:
                print(f'd: {d}, error: {e}')
                continue

        f,ax=plt.subplots()
        out['adfStat'].plot(ax=ax, marker='X')
        ax.axhline(out['95% conf'].mean(),lw=1,color='r',ls='dotted')
        ax.set_title('min d with thresh=0.0001')
        ax.set_xlabel('d values')
        ax.set_ylabel('adf stat');
        display(out)
```

```
d: 0.1, error: maxlag should be < nobs
d: 0.2, error: maxlag should be < nobs
d: 0.30000000000000004, error: maxlag should be < nobs
```

```
          adfStat      pVal  lags    nObs   95% conf
0.0  2.833609e+00  1.000000   1.0  2517.0  -2.862689
0.4 -0.000000e+00  0.958532   1.0     2.0 -10.370190
0.5 -5.825592e+03  0.000000   1.0     6.0  -3.646238
0.6 -5.718052e+03  0.000000   1.0    18.0  -3.042046
0.7 -6.338667e+03  0.000000   1.0    48.0  -2.923954
0.8 -8.751121e+03  0.000000   1.0   142.0  -2.882118
0.9 -8.327682e+03  0.000000   1.0   495.0  -2.867397
1.0 -1.135810e+10  0.000000   1.0  2516.0  -2.862689
```



1.3 [5.3] Take the series from exercise 2.b:

1.3.1 (a) Fit the series to a sine function. What is the R-squared?

Note: Is there a simpler way to do this?

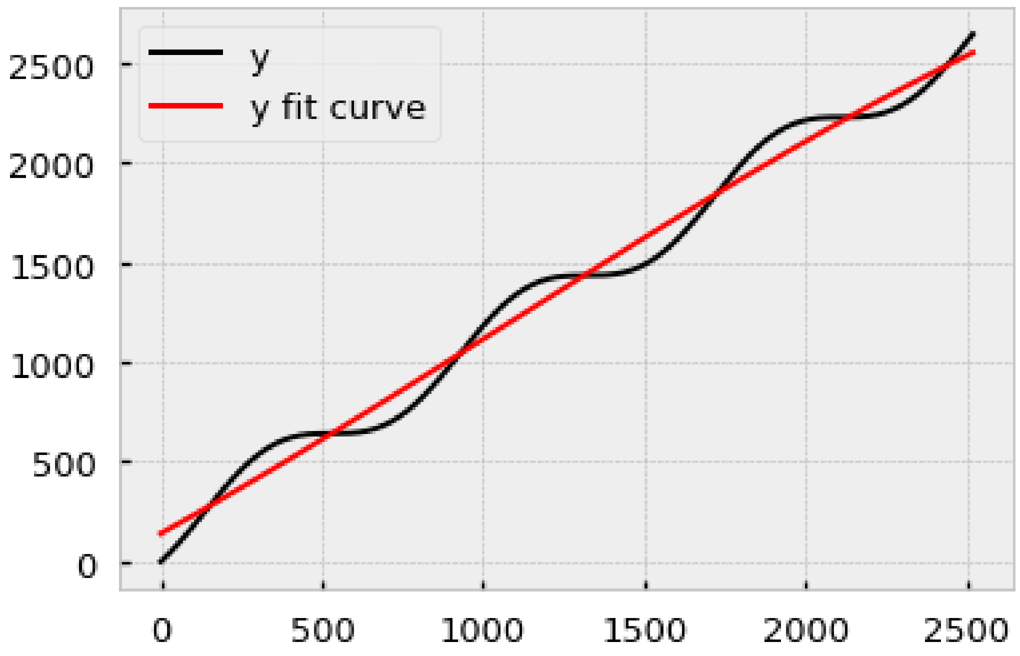
```
In [16]: ## fitting function taken from stackoverflow
         ##   https://stackoverflow.com/questions/16716302/how-do-i-fit-a-sine-curve-to-my-data-
         import numpy, scipy.optimize

         def fit_sin(tt, yy):
             '''Fit sin to the input time sequence, and return fitting parameters "amp", "omega"
             tt = numpy.array(tt)
             yy = numpy.array(yy)
             ff = numpy.fft.fftfreq(len(tt), (tt[1]-tt[0]))   # assume uniform spacing
             Fyy = abs(numpy.fft.fft(yy))
             guess_freq = abs(ff[numpy.argmax(Fyy[1:])+1])   # excluding the zero frequency "pea
             guess_amp = numpy.std(yy) * 2.**0.5
             guess_offset = numpy.mean(yy)
             guess = numpy.array([guess_amp, 2.*numpy.pi*guess_freq, 0., guess_offset])

             def sinfunc(t, A, w, p, c):  return A * numpy.sin(w*t + p) + c
             popt, pcov = scipy.optimize.curve_fit(sinfunc, tt, yy, p0=guess)
```

```
    A, w, p, c = popt
    f = w/(2.*numpy.pi)
    fitfunc = lambda t: A * numpy.sin(w*t + p) + c
    return {"amp": A, "omega": w, "phase": p, "offset": c, "freq": f, "period": 1./f, "
```

```
In [17]: res = fit_sin(s_.index.values, s_.values.ravel())
         res
```

```
Out[17]: {'amp': -2126.284918105075,
          'omega': 0.0004797999293735519,
          'phase': 2.6281709304906165,
          'offset': 1186.5758476117003,
          'freq': 7.636253045494306e-05,
          'period': 13095.42774502529,
          'fitfunc': <function __main__.fit_sin.<locals>.<lambda>(t)>,
          'maxcov': 20695.22970332341,
          'rawres': (array([1.01493055e+03, 2.49332750e-03, 0.00000000e+00, 1.36801775e+03]),
           array([-2.12628492e+03,  4.79799929e-04,  2.62817093e+00,  1.18657585e+03]),
           array([[ 2.06952297e+04,  4.98650324e-03, -7.28185170e+00,
                    3.99478720e+03],
                  [ 4.98650324e-03,  1.20345366e-09, -1.74718083e-06,
                    9.43037058e-04],
                  [-7.28185170e+00, -1.74718083e-06,  2.76112152e-03,
                   -1.81976202e+00],
                  [ 3.99478720e+03,  9.43037058e-04, -1.81976202e+00,
                    1.64645703e+03]]))}
```

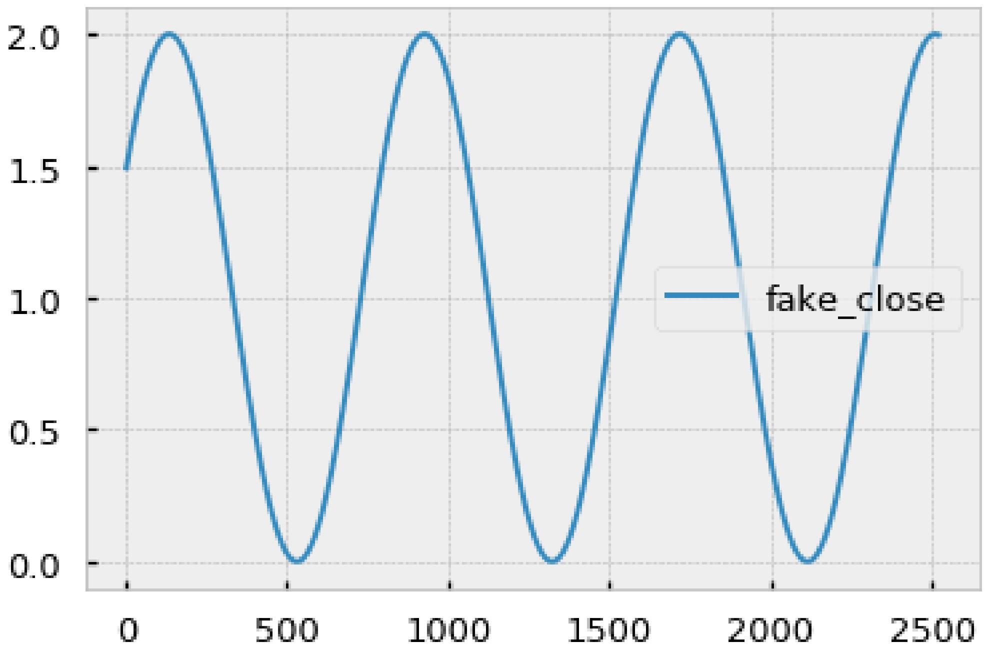
```
In [18]: xx = s_.index.values
         yy = s_.values.ravel()

         plt.plot(xx, yy, "-k", label="y", linewidth=2)
         #plt.plot(tt, yynoise, "ok", label="y with noise")
         plt.plot(xx, res["fitfunc"](xx), "r-", label="y fit curve", linewidth=2)
         plt.legend(loc="best")
         plt.show()
```



```
In [19]: slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(yy, res["fitfunc"]
         r_value**2

Out[19]: 0.9859147406461111
```

1.3.2 (b) Apply FFD$(d = 1)$. Fit the series to a sine function. What is the R-squared?

```
In [20]: #cols = ['adfStat','pVal','lags','nObs','95% conf']#,'corr']
         #out = pd.DataFrame(columns=cols)
         df1 = fracDiff(s_,d=1)
         #df1 = sm.tsa.stattools.adfuller(df0['fake_close'],maxlag=1,regression='c',autolag=None
         #out.loc[d]=list(df0[:4])+[df0[4]['5%']]
         df1.plot()

Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x7f452c31d160>
```



```
In [21]: xx = df1.index.values
         yy = df1.values.ravel()

         res = fit_sin(xx, yy)
         slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(yy, res["fitfunc"]
         r_value**2

Out[21]: 1.0
```

1.3.3 (c) What value of d maximizes the R-squared of a sinusoidal fit on FFD(d)? Why?

```
In [ ]:
```

1.4 5.4

Take dollar bar series on E-mini S&P 500 futures. Using the code in Snippet 5.3, for some `d` in `[0,2]`, compute `fracDiff_FFD(fracDiff_FFD(series,d)`. What do you get? Why?

Note: for some reason this never finishes computing in my notebook

```
In [22]: def getWeights_FFD(d,thres):
             w,k=[1.],1
             while True:
                 w_=-w[-1]/k*(d-k+1)
                 if abs(w_)<thres:break
                 w.append(w_);k+=1
             return np.array(w[::-1]).reshape(-1,1)
```

```
#---------------------------------------------------------------------------

def fracDiff_FFD(series,d,thres=1e-5):
    # Constant width window (new solution)
    w = getWeights_FFD(d,thres)
    width = len(w)-1
    df={}
    for name in series.columns:
        seriesF, df_=series[[name]].fillna(method='ffill').dropna(), pd.Series()
        for iloc1 in range(width,seriesF.shape[0]):
            loc0,loc1=seriesF.index[iloc1-width], seriesF.index[iloc1]
            test_val = series.loc[loc1,name] # must resample if duplicate index
            if isinstance(test_val, (pd.Series, pd.DataFrame)):
                test_val = test_val.resample('1m').mean()
            if not np.isfinite(test_val).any(): continue # exclude NAs
            #print(f'd: {d}, iloc1:{iloc1} shapes: w:{w.T.shape}, series: {seriesF.loc[
            try:
                df_.loc[loc1]=np.dot(w.T, seriesF.loc[loc0:loc1])[0,0]
            except:
                continue
        df[name]=df_.copy(deep=True)
    df=pd.concat(df,axis=1)
    return df
```

```
In [23]: def dask_resample(ser, freq='L'):
             dds = dd.from_pandas(ser, chunksize=len(ser)//100)
             tdf = (dds
                    .resample(freq)
                    .mean()
                    .dropna()
                   ).compute()
             return tdf

         infp=PurePath(data_dir/'clean_IVE_fut_prices.parquet')
         df = pd.read_parquet(infp)

         dv_rs = dask_resample(df, '1s')
         cprint(dv_rs)

         dbars = brs.dollar_bar_df(dv_rs, 'dv', 1_000_000)
         cprint(dbars)
```

```
[########################################] | 100% Completed | 31.4s
-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                      price     bid     ask      size          v             dv
```

```
dates
2018-02-26 15:59:59  115.35  115.34  115.36     412.5     412.5  4.758188e+04
2018-02-26 16:00:00  115.35  115.34  115.35    5362.0    5362.0  6.185067e+05
2018-02-26 16:10:00  115.35  115.22  115.58       0.0       0.0  0.000000e+00
2018-02-26 16:16:14  115.30  114.72  115.62  778677.0  778677.0  8.978146e+07
2018-02-26 18:30:00  115.35  114.72  117.38       0.0       0.0  0.000000e+00
--------------------------------------------------


  0%|          | 0/941297 [00:00<?, ?it/s]

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 941297 entries, 2009-09-28 09:30:00 to 2018-02-26 18:30:00
Data columns (total 6 columns):
price    941297 non-null float64
bid      941297 non-null float64
ask      941297 non-null float64
size     941297 non-null float64
v        941297 non-null float64
dv       941297 non-null float64
dtypes: float64(6)
memory usage: 50.3 MB
None
-------------------------------------------------------------------------------



100%|| 941297/941297 [00:00<00:00, 2822315.09it/s]


-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                      price         bid         ask           size  \
dates
2018-02-26 15:31:06  115.29  115.280000  115.290000    2022.000000
2018-02-26 15:40:15  115.41  115.400000  115.410000     723.000000
2018-02-26 15:49:42  115.20  115.176667  115.186667    4487.166667
2018-02-26 15:59:04  115.27  115.260000  115.270000     300.000000
2018-02-26 16:16:14  115.30  114.720000  115.620000  778677.000000

                                v            dv
dates
2018-02-26 15:31:06    2022.000000  2.331164e+05
2018-02-26 15:40:15     723.000000  8.344143e+04
2018-02-26 15:49:42    4487.166667  5.171190e+05
2018-02-26 15:59:04     300.000000  3.458100e+04
2018-02-26 16:16:14  778677.000000  8.978146e+07
--------------------------------------------------
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 30861 entries, 2009-09-28 09:53:49 to 2018-02-26 16:16:14
Data columns (total 6 columns):
price    30861 non-null float64
bid      30861 non-null float64
ask      30861 non-null float64
size     30861 non-null float64
v        30861 non-null float64
dv       30861 non-null float64
dtypes: float64(6)
memory usage: 1.6 MB
None
--------------------------------------------------------------------------------
```

```
In [24]: d = 0.5
         sel = dbars[['price']].iloc[:100]
         #val = fracDiff_FFD(fracDiff_FFD(sel, d), -d) # Never finishes don't run
         #val
```

1.5 5.5

Take the dollar bar series on E-mini S&P 500 futures.

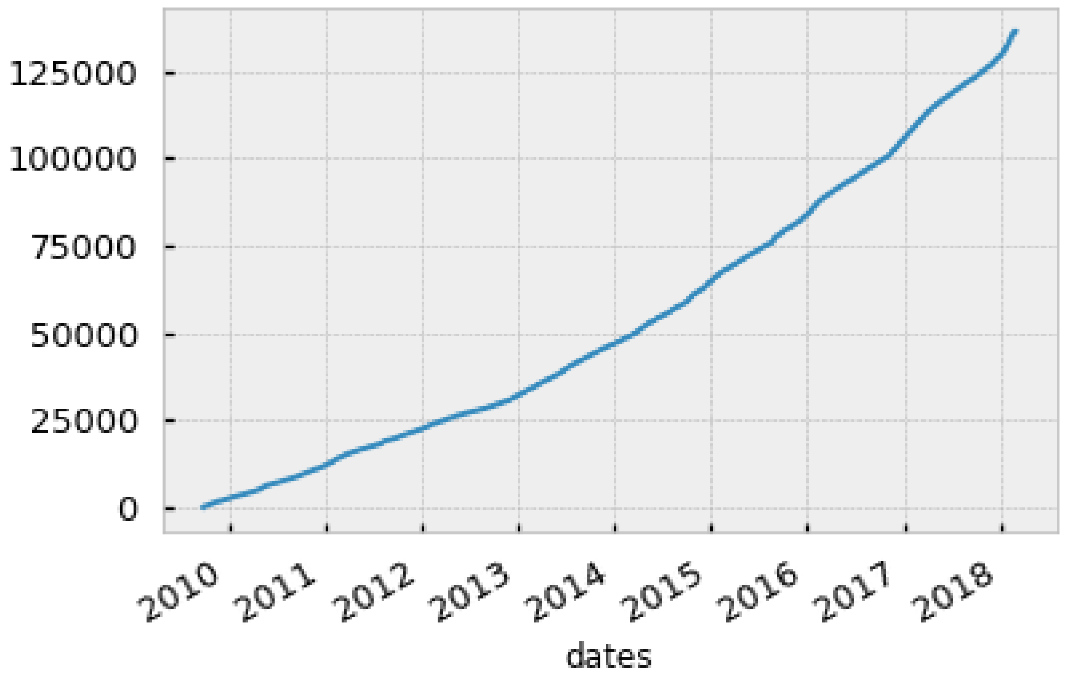
1.5.1 (a) Form a new series as a cumulative sum of log-prices

```
In [25]: x = np.log(dbars.price).cumsum()
         cprint(x)

         x.plot()
```

```
--------------------------------------------------------------------------------
dataframe information
--------------------------------------------------------------------------------
                             price
dates
2018-02-26 15:31:06  136219.267867
2018-02-26 15:40:15  136224.016358
2018-02-26 15:49:42  136228.763027
2018-02-26 15:59:04  136233.510305
2018-02-26 16:16:14  136238.257842
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 30861 entries, 2009-09-28 09:53:49 to 2018-02-26 16:16:14
Data columns (total 1 columns):
price    30861 non-null float64
dtypes: float64(1)
memory usage: 482.2 KB
```

```
None
--------------------------------------------------------------------------------
```

```
Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x7f452c294c18>
```

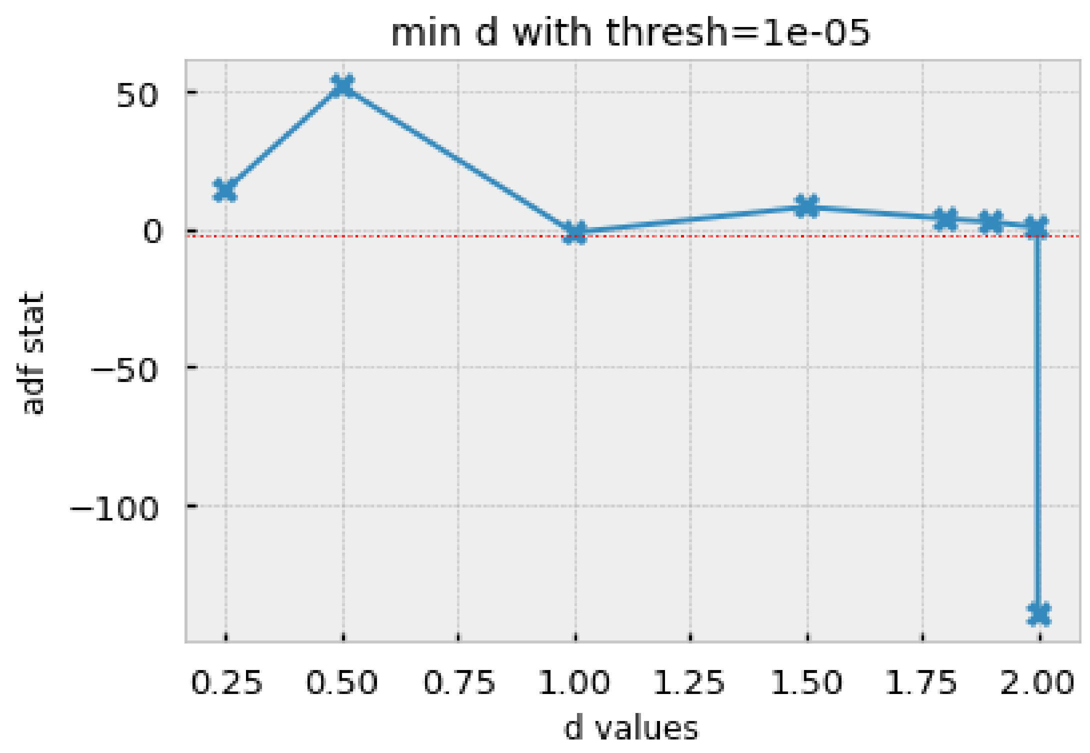


1.5.2 (b) Apply FFD, with $\tau = 1E - 5$. Determine for what minimum $d \in [0, 2]$ the new series is stationary

```
In [26]: def get_optimal_ffd(ds, t=1e-5):

             cols = ['adfStat','pVal','lags','nObs','95% conf']#,'corr']
             out = pd.DataFrame(columns=cols)

             for d in tqdm_notebook(ds):
                 try:
                     #dfx = fracDiff(x.to_frame(),d,thres=1e-5)
                     dfx = fracDiff_FFD(x.to_frame(),d,thres=t)
                     dfx = sm.tsa.stattools.adfuller(dfx['price'], maxlag=1,regression='c',autol
                     out.loc[d]=list(dfx[:4])+[dfx[4]['5%']]
                 except Exception as e:
                     print(f'{d} error: {e}')
                     break
             return out
```

```
         #==========================================
         ds = [0.25,0.5,1,1.5,1.8,1.9,1.999,2]
         thres = 1e-5
         out = get_optimal_ffd(ds, thres) # takes 15 minutes to run on ~44k points

HBox(children=(IntProgress(value=0, max=8), HTML(value='')))
```

```
In [27]: f,ax=plt.subplots()
         out['adfStat'].plot(ax=ax, marker="X", markersize=10)
         ax.axhline(out['95% conf'].mean(),lw=1,color='r',ls='dotted')
         ax.set_title(f'min d with thresh={thres}')
         ax.set_xlabel('d values')
         ax.set_ylabel('adf stat');
         display(out)

          adfStat      pVal  lags     nObs  95% conf
0.250   13.729830  1.000000   1.0  28056.0 -2.861643
0.500   51.436628  1.000000   1.0  29933.0 -2.861637
1.000   -1.552027  0.507636   1.0  30858.0 -2.861634
1.500    7.740002  1.000000   1.0  30788.0 -2.861634
1.800    3.516861  1.000000   1.0  30818.0 -2.861634
1.900    2.326653  0.998971   1.0  30829.0 -2.861634
1.999    0.342193  0.979200   1.0  30853.0 -2.861634
2.000 -139.096911  0.000000   1.0  30857.0 -2.861634
```



```
In [28]: min_ffd = out[out.pVal < 0.05].iloc[0].name
         min_ffd
```

```
Out[28]: 2.0
```

1.5.3 (c) Compute the correlation of the fracdiff series to the original (untransformed) series

```
In [29]: dfx2 = fracDiff_FFD(x.to_frame(),min_ffd,thres=thres)
         cprint(dfx2)

         joined = dfx2.join(x.rename('original'), how='left')
         joined.corr()
```

```
-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                        price
2018-02-26 15:31:06 -0.000575
2018-02-26 15:40:15  0.001040
2018-02-26 15:49:42 -0.001821
2018-02-26 15:59:04  0.000607
2018-02-26 16:16:14  0.000260
--------------------------------------------------
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 30859 entries, 2009-09-28 10:19:50 to 2018-02-26 16:16:14
Data columns (total 1 columns):
price    30859 non-null float64
dtypes: float64(1)
memory usage: 482.2 KB
None
--------------------------------------------------------------------------------
```

```
Out[29]:              price  original
         price     1.000000 -0.002704
         original -0.002704  1.000000
```

1.5.4 (d) Apply Engel-Granger cointegration test on the original and fracdiff series. Are they cointegrated? Why?

```
In [30]: sm.tsa.stattools.coint(joined.price, joined.original)
```

```
Out[30]: (-27.177877591790516, 0.0, array([-3.89679495, -3.33632801, -3.04458745]))
```

1.5.5 (e) Apply a Jarque-Bera normality test on the fracdiff series.

```
In [31]: np.random.seed(0)
         stats.jarque_bera(dfx2)
```

```
Out[31]: (4067589070.053132, 0.0)
```

1.6 5.6 Take the fracdiff series from exercise 5

1.6.1 (a) Apply a CUSUM filter (Chapter 2), where h is twice the standard deviation of the series.

```
In [32]: tEvents = snp.getTEvents(dfx2,h=dfx2.std().iat[0]*2)
         display(tEvents)
```

```
100%|| 8417/8417 [00:03<00:00, 2430.45it/s]


DatetimeIndex(['2009-09-28 11:34:21', '2009-09-28 11:53:11',
               '2009-09-28 12:30:58', '2009-09-28 14:44:10',
               '2009-09-28 15:39:46', '2009-09-29 12:53:47',
               '2009-09-29 13:59:56', '2009-09-30 12:40:13',
               '2009-09-30 13:31:12', '2009-10-01 09:29:51',
               ...
               '2018-02-26 12:09:16', '2018-02-26 12:48:18',
               '2018-02-26 13:06:34', '2018-02-26 13:21:31',
               '2018-02-26 13:48:20', '2018-02-26 13:58:17',
               '2018-02-26 14:09:45', '2018-02-26 15:20:10',
              '2018-02-26 15:24:32', '2018-02-26 16:16:14'],
             dtype='datetime64[ns]', length=8341, freq=None)
```

1.6.2 (b) Use the filtered timestamps to sample a features' matrix. Use as one of the features the fracDiff value.

```
In [33]: dbars_feat = dbars.price.loc[tEvents]
         frac_diff_feat = dfx2.loc[tEvents]
         ftMtx = (pd.DataFrame()
                  .assign(dbars=dbars_feat,
                          frac_diff_feat=frac_diff_feat)
                  .drop_duplicates().dropna())
         cprint(ftMtx)
```

```
-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                        dbars  frac_diff_feat
2018-02-26 13:58:17  115.1200        0.000391
2018-02-26 14:09:45  115.1501        0.000261
2018-02-26 15:20:10  115.3200        0.000694
2018-02-26 15:24:32  115.3563        0.000315
2018-02-26 16:16:14  115.3000        0.000260
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 8329 entries, 2009-09-28 11:34:21 to 2018-02-26 16:16:14
Data columns (total 2 columns):
dbars             8329 non-null float64
frac_diff_feat    8329 non-null float64
dtypes: float64(2)
memory usage: 195.2 KB
None
-------------------------------------------------------------------------------
```

1.6.3 (c) Form labels using the triple-barrier method, with symmetric horizontal barriers of `twice` the daily standard deviation, and a vertical barrier of `5` days

```
In [34]: dailyVol = snp.getDailyVol(ftMtx.dbars)
         t1 = snp.addVerticalBarrier(tEvents, ftMtx.dbars, numDays=5)

         ptsl = [1,1]
         #ptsl = [daily]
         target=dailyVol*2
         # select minRet
         minRet = 0.01
         # get cpu count - 1
         cpus = cpu_count() - 1
         events = snp.getEvents(ftMtx.dbars,tEvents,ptsl,target,minRet,cpus,t1=t1)
         cprint(events)
```

```
2018-10-18 17:06:56.495423 100.0% applyPtSlOnT1 done after 0.02 minutes. Remaining 0.0 minutes..


-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                     t1      trgt
2018-02-26 13:58:17 NaT  0.023527
2018-02-26 14:09:45 NaT  0.023438
2018-02-26 15:20:10 NaT  0.023394
2018-02-26 15:24:32 NaT  0.023354
2018-02-26 16:16:14 NaT  0.023291
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6948 entries, 2009-09-30 12:40:13 to 2018-02-26 16:16:14
Data columns (total 2 columns):
t1      6914 non-null datetime64[ns]
trgt    6948 non-null float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 162.8 KB
None
-------------------------------------------------------------------------------
```

```
In [35]: ## Example
         close=ftMtx.dbars
         numCoEvents = snp.mpPandasObj(snp.mpNumCoEvents,('molecule',events.index),
                                  cpus,closeIdx=close.index,t1=events['t1'])
         numCoEvents = numCoEvents.loc[~numCoEvents.index.duplicated(keep='last')]
         numCoEvents = numCoEvents.reindex(close.index).fillna(0)
         out=pd.DataFrame()
         out['tW'] = snp.mpPandasObj(snp.mpSampleTW,('molecule',events.index),
                                  cpus,t1=events['t1'],numCoEvents=numCoEvents)
         cprint(out)
```

```
2018-10-18 17:06:58.956595 100.0% mpNumCoEvents done after 0.01 minutes. Remaining 0.0 minutes.


-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                     tW
2018-02-26 13:58:17 NaN
```

```
2018-02-26 14:09:45 NaN
2018-02-26 15:20:10 NaN
2018-02-26 15:24:32 NaN
2018-02-26 16:16:14 NaN
-------------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6948 entries, 2009-09-30 12:40:13 to 2018-02-26 16:16:14
Data columns (total 1 columns):
tW    6914 non-null float64
dtypes: float64(1)
memory usage: 108.6 KB
None
-------------------------------------------------------------------------------



2018-10-18 17:06:59.900933 100.0% mpSampleTW done after 0.01 minutes. Remaining 0.0 minutes.
```

```
In [36]: ## example ##
         out['w']=snp.mpPandasObj(snp.mpSampleW,('molecule',events.index),cpus,
                                  t1=events['t1'],numCoEvents=numCoEvents,close=close)
         out['w']*=out.shape[0]/out['w'].sum()
         cprint(out)
```

```
2018-10-18 17:07:01.247596 100.0% mpSampleW done after 0.01 minutes. Remaining 0.0 minutes.


-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                     tW    w
2018-02-26 13:58:17 NaN  0.0
2018-02-26 14:09:45 NaN  0.0
2018-02-26 15:20:10 NaN  0.0
2018-02-26 15:24:32 NaN  0.0
2018-02-26 16:16:14 NaN  0.0
-------------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6948 entries, 2009-09-30 12:40:13 to 2018-02-26 16:16:14
Data columns (total 2 columns):
tW    6914 non-null float64
w     6948 non-null float64
dtypes: float64(2)
memory usage: 162.8 KB
None
-------------------------------------------------------------------------------
```

```
In [37]: # get labels
         labels = snp.getBins(events, ftMtx.dbars)
         #cprint(labels)

         clean_labels = snp.dropLabels(labels)
         cprint(clean_labels)
```

```
dropped label:  0.0 0.0011570726063060456
-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                          ret  bin
2018-02-21 09:47:52  0.009430  1.0
2018-02-21 11:10:00  0.008097  1.0
2018-02-21 12:53:48  0.012764  1.0
2018-02-21 13:19:05  0.012619  1.0
2018-02-21 14:12:30  0.011240  1.0
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6906 entries, 2009-09-30 12:40:13 to 2018-02-21 14:12:30
Data columns (total 2 columns):
ret    6906 non-null float64
bin    6906 non-null float64
dtypes: float64(2)
memory usage: 161.9 KB
None
-------------------------------------------------------------------------------
```

1.6.4 (d) Fit a bagging classifier of decision trees where:

(i) The observed features are bootstrapped using the sequential method from chapter 4. Note: must use multiprocessing of some kind as seqBootstrap is very slow

```
In [38]: @nb.njit
         def func(arr,i):
             col = arr[i]
             mask = np.where(col>0)
             return np.mean(col[mask])

         @nb.njit
         def njit_getAvgUniqueness(indM):
             # Average uniqueness from indicator matrix
             c=indM.sum(axis=1).reshape(-1,1) # concurrency
             u=np.divide(indM,c) # uniqueness
             avgU = np.zeros(len(u.T)) # avg. uniqueness
             i = 0
```

```
        for i in range(len(u.T)):
            avgU[i] = func(u.T,i)
            i+=1
        return avgU

    @nb.jit
    def jit_seqBootstrap(indM,sLength=None):
        # Generate a sample via sequential bootstrap
        if sLength is None:sLength=indM.shape[1]
        phi=[]
        while len(phi)<sLength:
            avgU=pd.Series()
            for i in indM:
                indM_=indM[phi+[i]] # reduce indM
                avgU.loc[i]=njit_getAvgUniqueness(indM_.values)[-1]
            prob=avgU/avgU.sum() # draw prob
            phi+=[np.random.choice(indM.columns,p=prob)]
        return phi
    #------------------------

    def split_t1(t1, partitions):
        return np.array_split(t1, partitions)

    def mp_func(indM):
        # jit funcs about 2x as fast
        phi = jit_seqBootstrap(indM)
        seqU = njit_getAvgUniqueness(indM[phi].values).mean()
        #phi = snp.seqBootstrap(indM)
        #seqU= snp.getAvgUniqueness(indM[phi])
        return seqU

    def main_mp(t1, partitions=100, cpus=8):
        jobs = []
        splits = split_t1(t1,partitions=100)
        for part_t1 in splits:
            indM = snp.getIndMatrix(part_t1.index, part_t1)
            job = {'func':mp_func,'indM':indM}
            jobs.append(job)
        if cpus==1: out=snp.processJobs_(jobs)
        else: out=snp.processJobs(jobs,numThreads=cpus)
        return pd.DataFrame(out)
```

```
In [39]: seqUs = main_mp(t1)
```

```
2018-10-18 17:09:25.339624 100.0% mp_func done after 2.29 minutes. Remaining 0.0 minutes.
```

```
In [40]: seqUs.describe()
```

```
Out[40]:                  0
         count  100.000000
         mean     0.074860
         std      0.022629
         min      0.029593
         25%      0.062652
         50%      0.071346
         75%      0.089933
         max      0.133672
```

```
In [41]: # get avg uniqueness for bootstrapping
         avgU = seqUs.mean()[0]
         avgU
```

```
Out[41]: 0.07485970931182089
```

(ii) On each bootstrapped sample, sample weights are determined using the techniques from Chapter 4 Note: alternative implementations are welcome.

```
In [42]: from sklearn.model_selection import train_test_split
         from sklearn.tree import DecisionTreeClassifier
         from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor, BaggingClas
```

```
In [43]: def evaluate(X,y,clf):
             from sklearn import metrics
             # The random forest model by itself
             y_pred_rf = clf.predict_proba(X)[:, 1]
             y_pred = clf.predict(X)
             fpr_rf, tpr_rf, _ = metrics.roc_curve(y, y_pred_rf)
             print(metrics.classification_report(y, y_pred))

             plt.figure(figsize=(9,6))
             plt.plot([0, 1], [0, 1], 'k--')
             plt.plot(fpr_rf, tpr_rf, label='clf')
             plt.xlabel('False positive rate')
             plt.ylabel('True positive rate')
             plt.title('ROC curve')
             plt.legend(loc='best')
             plt.show()
```

```
In [44]: trgt = clean_labels.bin
         trgt
```

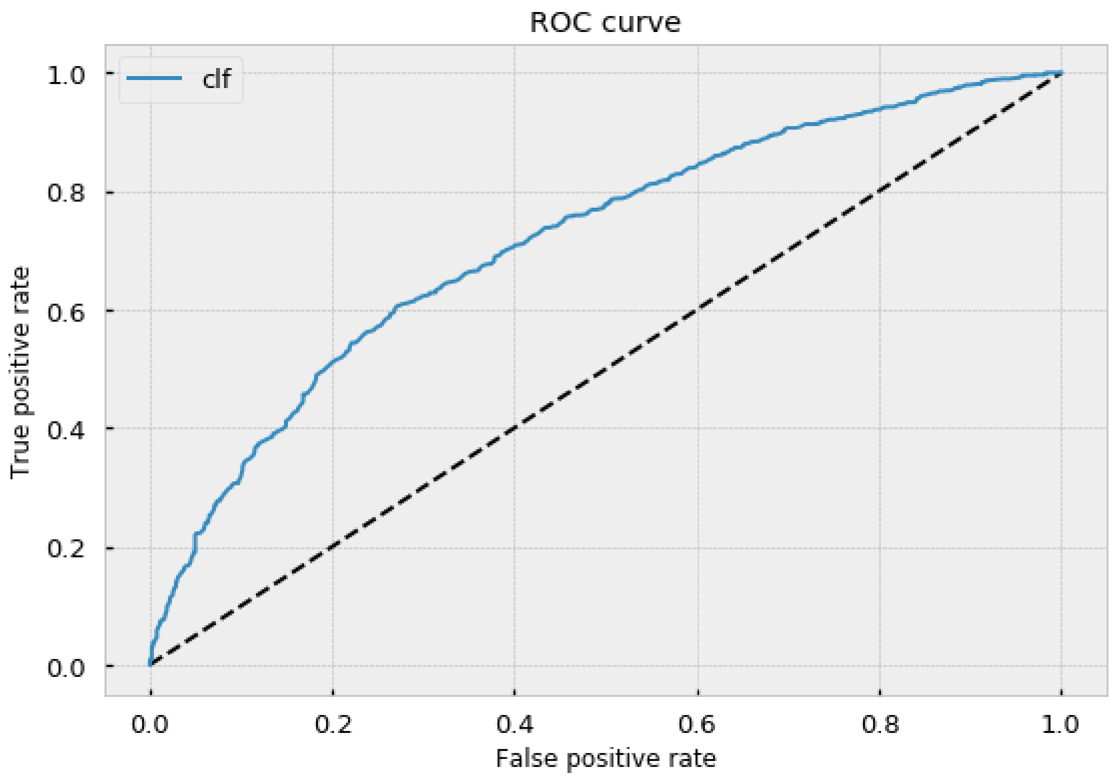
```
Out[44]: 2009-09-30 12:40:13   -1.0
         2009-09-30 13:31:12   -1.0
         2009-10-01 09:29:51   -1.0
         2009-10-02 10:18:42    1.0
         2009-10-02 10:35:05    1.0
         2009-10-02 10:37:29    1.0
```

```
2009-10-02 13:20:04     1.0
2009-10-05 10:41:50     1.0
2009-10-05 11:50:49     1.0
2009-10-05 12:25:18     1.0
2009-10-05 13:11:27     1.0
2009-10-06 09:29:52     1.0
2009-10-06 10:16:02     1.0
2009-10-06 11:32:02     1.0
2009-10-06 15:35:49     1.0
2009-10-07 15:34:00     1.0
2009-10-07 15:53:56     1.0
2009-10-08 09:29:51     1.0
2009-10-08 12:07:39     1.0
2009-10-08 12:52:50     1.0
2009-10-09 11:21:48     1.0
2009-10-09 14:58:12     1.0
2009-10-09 15:28:04     1.0
2009-10-12 09:31:02     1.0
2009-10-12 10:14:38     1.0
2009-10-12 11:12:35     1.0
2009-10-12 14:08:45     1.0
2009-10-14 09:29:52     1.0
2009-10-14 11:59:09     1.0
2009-10-14 12:25:52     1.0
2009-10-14 14:12:01     1.0
2009-10-15 11:02:54    -1.0
2009-10-15 14:07:36    -1.0
2009-10-15 14:52:32    -1.0
2009-10-15 15:18:10    -1.0
2009-10-15 15:57:25    -1.0
2009-10-16 13:37:21    -1.0
2009-10-16 15:40:15    -1.0
2009-10-19 10:46:14    -1.0
2009-10-19 11:39:38    -1.0
2009-10-19 12:34:14    -1.0
2009-10-19 13:35:25    -1.0
2009-10-19 14:32:08    -1.0
2009-10-21 09:41:06    -1.0
2009-10-21 10:11:57    -1.0
2009-10-21 10:57:04    -1.0
2009-10-22 11:36:37    -1.0
2009-10-22 12:20:25    -1.0
2009-10-22 13:04:37    -1.0
2009-10-22 15:17:46    -1.0
                       ...
2018-02-12 11:17:12     1.0
2018-02-12 11:22:33     1.0
2018-02-12 11:55:37     1.0
```

```
2018-02-12 12:12:51    1.0
2018-02-12 12:23:34    1.0
2018-02-12 13:57:53    1.0
2018-02-12 15:14:17    1.0
2018-02-13 11:48:45    1.0
2018-02-13 12:01:57    1.0
2018-02-13 13:43:37    1.0
2018-02-14 10:09:34    1.0
2018-02-14 10:19:18    1.0
2018-02-14 10:30:48    1.0
2018-02-14 11:41:53    1.0
2018-02-14 12:28:36    1.0
2018-02-14 12:58:13    1.0
2018-02-14 13:14:57    1.0
2018-02-14 13:36:02    1.0
2018-02-14 13:53:59    1.0
2018-02-14 14:08:28    1.0
2018-02-14 14:48:01    1.0
2018-02-14 15:00:14    1.0
2018-02-14 15:08:14    1.0
2018-02-15 09:31:56    1.0
2018-02-15 09:56:14    1.0
2018-02-15 11:22:26    1.0
2018-02-15 11:29:54    1.0
2018-02-15 13:21:31   -1.0
2018-02-15 14:51:30   -1.0
2018-02-15 15:58:36   -1.0
2018-02-15 15:59:47   -1.0
2018-02-15 16:00:00   -1.0
2018-02-16 10:42:42   -1.0
2018-02-16 10:59:17   -1.0
2018-02-16 11:11:50   -1.0
2018-02-16 12:05:18   -1.0
2018-02-16 12:32:45   -1.0
2018-02-16 13:46:07   -1.0
2018-02-16 14:56:08   -1.0
2018-02-16 15:03:29   -1.0
2018-02-16 15:10:45   -1.0
2018-02-20 09:50:13    1.0
2018-02-20 11:36:52    1.0
2018-02-20 12:35:02    1.0
2018-02-20 15:50:16    1.0
2018-02-21 09:47:52    1.0
2018-02-21 11:10:00    1.0
2018-02-21 12:53:48    1.0
2018-02-21 13:19:05    1.0
2018-02-21 14:12:30    1.0
Name: bin, Length: 6906, dtype: float64
```

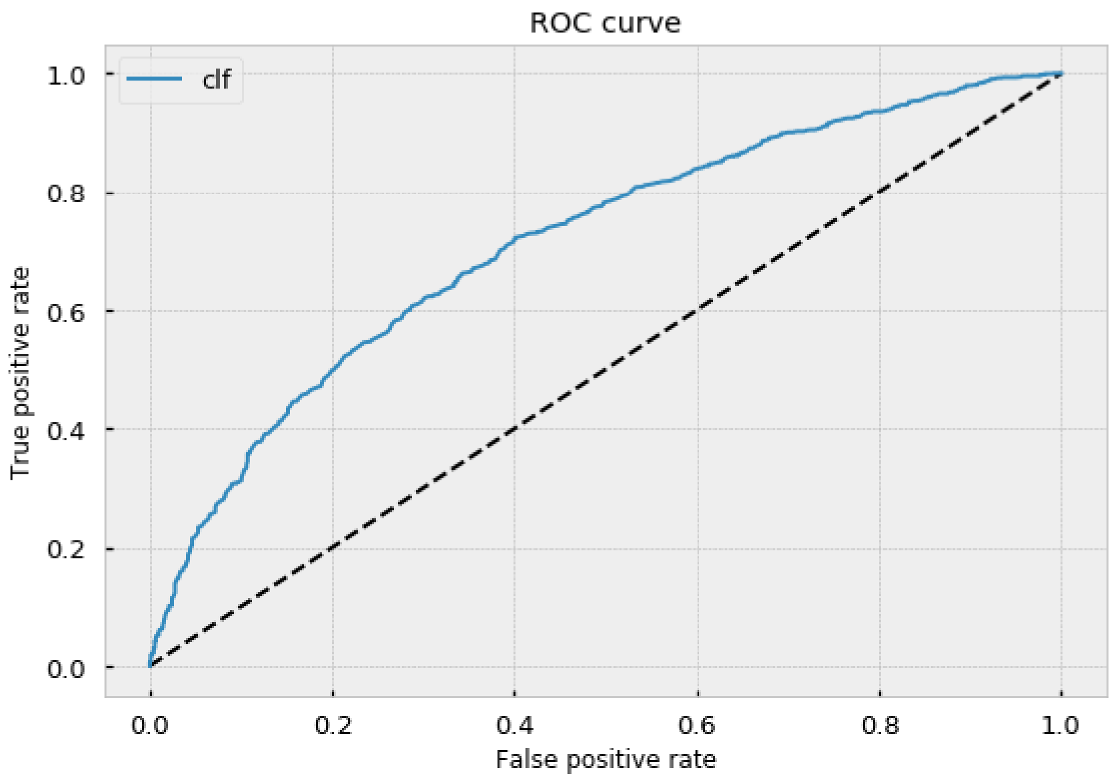
```
In [45]: # model data
         #data = ftMtx.join(out,how='left').join(trgt,how='left').iloc[phi].dropna()
         data = ftMtx.join(out,how='left').join(trgt,how='left').dropna()
         X = data.iloc[:,:-1].values
         y = data.iloc[:,-1].values.reshape(-1,1)

In [46]: cprint(data)

-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                        dbars  frac_diff_feat        tW         w  bin
2018-02-21 09:47:52  113.4700        0.002338  0.056174  0.355414  1.0
2018-02-21 11:10:00  113.6200        0.000304  0.056076  0.196611  1.0
2018-02-21 12:53:48  113.6000        0.000969  0.052932  0.258432  1.0
2018-02-21 13:19:05  113.6163        0.000143  0.051662  0.270087  1.0
2018-02-21 14:12:30  114.0382        0.001476  0.048293  0.326394  1.0
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6906 entries, 2009-09-30 12:40:13 to 2018-02-21 14:12:30
Data columns (total 5 columns):
dbars             6906 non-null float64
frac_diff_feat    6906 non-null float64
tW                6906 non-null float64
w                 6906 non-null float64
bin               6906 non-null float64
dtypes: float64(5)
memory usage: 323.7 KB
None
-------------------------------------------------------------------------------



In [47]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=R

In [48]: base_clf = DecisionTreeClassifier(criterion='entropy',max_features='auto',
                                           class_weight='balanced')
         bc = BaggingClassifier(base_estimator=base_clf,n_estimators=1000,
                                max_samples=avgU,max_features=1.,random_state=RANDOM_STATE)

In [49]: fit = bc.fit(X_train,y_train)

/media/bcr/HDD/anaconda3/envs/bayes_dash/lib/python3.6/site-packages/sklearn/ensemble/bagging.py
  y = column_or_1d(y, warn=True)


In [50]: evaluate(X_test,y_test,fit)

              precision    recall  f1-score   support
```

```
        -1.0       0.65      0.44      0.53       905
         1.0       0.65      0.81      0.72      1167

   micro avg       0.65      0.65      0.65      2072
   macro avg       0.65      0.63      0.63      2072
weighted avg       0.65      0.65      0.64      2072
```



```
In [51]: rf_clf = RandomForestClassifier(n_estimators=1,
                                         class_weight='balanced_subsample',
                                         criterion='entropy',
                                         bootstrap=False)
         bc_rf = BaggingClassifier(base_estimator=rf_clf, n_estimators=1000,
                                   max_samples=avgU, max_features=1.,
                                   random_state=RANDOM_STATE)

         fit = bc_rf.fit(X_train,y_train)
         evaluate(X_test,y_test,fit)

/media/bcr/HDD/anaconda3/envs/bayes_dash/lib/python3.6/site-packages/sklearn/ensemble/bagging.py
  y = column_or_1d(y, warn=True)
```

```
              precision    recall  f1-score   support

        -1.0       0.65      0.45      0.53       905
         1.0       0.66      0.81      0.73      1167

   micro avg       0.65      0.65      0.65      2072
   macro avg       0.65      0.63      0.63      2072
weighted avg       0.65      0.65      0.64      2072
```



```
In [ ]:
```