04. Sample Weights

November 2, 2018

Table of Contents

4.5.3 (c) What is the distribution of the predicted labels, before and after applying balanced class weights?

4.6 Update the draw probabilities for the final draw in section 4.5.3.

4.7 In Section 4.5.3 suppose that number 2 is picked again in the second draw. Waht would be the updated probabilities for the third draw?

1 Sampling

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'
# 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 pymc3 as pm
import numba as nb
import math

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

```python
%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
```

```
2018-10-18T16:50:33-06:00

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
```

2 Code Snippets

2.1 Estimating uniqueness of a label [4.1]

```
In [2]: def mpNumCoEvents(closeIdx,t1,molecule):
            '''
            Compute the number of concurrent events per bar.
            +molecule[0] is the date of the first event on which the weight will be computed
            +molecule[-1] is the date of the last event on which the weight will be computed

            Any event that starts before t1[modelcule].max() impacts the count.
            '''
            #1) find events that span the period [molecule[0],molecule[-1]]
            t1=t1.fillna(closeIdx[-1]) # unclosed events still must impact other weights
            t1=t1[t1>=molecule[0]] # events that end at or after molecule[0]
            t1=t1.loc[:t1[molecule].max()] # events that start at or before t1[molecule].max()
            #2) count events spanning a bar
            iloc=closeIdx.searchsorted(np.array([t1.index[0],t1.max()]))
            count=pd.Series(0,index=closeIdx[iloc[0]:iloc[1]+1])
            for tIn,tOut in t1.iteritems():count.loc[tIn:tOut]+=1.
            return count.loc[molecule[0]:t1[molecule].max()]
```

2.2 Estimating the average uniqueness of a label [4.2]

```
In [3]: def mpSampleTW(t1,numCoEvents,molecule):
            # Derive avg. uniqueness over the events lifespan
            wght=pd.Series(index=molecule)
            for tIn,tOut in t1.loc[wght.index].iteritems():
                wght.loc[tIn]=(1./numCoEvents.loc[tIn:tOut]).mean()
            return wght
```

2.3 Sequential Bootstrap [4.5.2]

2.3.1 Build Indicator Matrix [4.3]

```
In [4]: def getIndMatrix(barIx,t1):
            # Get Indicator matrix
            indM=(pd.DataFrame(0,index=barIx,columns=range(t1.shape[0])))
```

```
        for i,(t0,t1) in enumerate(t1.iteritems()):indM.loc[t0:t1,i]=1.
        return indM
```

2.3.2 Compute average uniqueness [4.4]

```
In [5]: def getAvgUniqueness(indM):
            # Average uniqueness from indicator matrix
            c=indM.sum(axis=1) # concurrency
            u=indM.div(c,axis=0) # uniqueness
            avgU=u[u>0].mean() # avg. uniqueness
            return avgU
```

2.3.3 return sample from sequential bootstrap [4.5]

```
In [6]: def 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]=getAvgUniqueness(indM_).iloc[-1]
                prob=avgU/avgU.sum() # draw prob
                phi+=[np.random.choice(indM.columns,p=prob)]
            return phi
```

2.4 Determination of sample weight by absolute return attribution [4.10]

```
In [7]: def mpSampleW(t1,numCoEvents,close,molecule):
            # Derive sample weight by return attribution
            ret=np.log(close).diff() # log-returns, so that they are additive
            wght=pd.Series(index=molecule)
            for tIn,tOut in t1.loc[wght.index].iteritems():
                wght.loc[tIn]=(ret.loc[tIn:tOut]/numCoEvents.loc[tIn:tOut]).sum()
            return wght.abs()
```

2.5 Implementation of Time-Decay Factors [4.11]

```
In [8]: def getTimeDecay(tW,clfLastW=1.):
            # apply piecewise-linear decay to observed uniqueness (tW)
            # newest observation gets weight=1, oldest observation gets weight=clfLastW
            clfW=tW.sort_index().cumsum()
            if clfLastW>=0: slope=(1.-clfLastW)/clfW.iloc[-1]
            else: slope=1./((clfLastW+1)*clfW.iloc[-1])
            const=1.-slope*clfW.iloc[-1]
            clfW=const+slope*clfW
            clfW[clfW<0]=0
```

```
        print(const,slope)
        return clfW
```

3 Example of Sequential Bootstrap [4.6]

```
In [9]: def main():
            np.random.seed(12121) # fix seed as results are unstable
            t1=pd.Series([2,3,5],index=[0,2,4]) # t0,t1 for each feature obs
            barIx=range(t1.max()+1) # index of bars
            indM=snp.getIndMatrix(barIx,t1)
            phi_random=np.random.choice(indM.columns,size=indM.shape[1])
            print(phi_random)
            print(f'Standard uniqueness: {snp.getAvgUniqueness(indM[phi_random]).mean():.4f}')
            phi_seq=snp.seqBootstrap(indM)
            print(phi_seq)
            print(f'Sequential uniqueness: {snp.getAvgUniqueness(indM[phi_seq]).mean():.4f}')

        main()
```

```
[2 2 2]
Standard uniqueness: 0.3333
[1, 2, 2]
Sequential uniqueness: 0.6667
```

4 Exercises

```
In [10]: 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=Path(data_dir/'clean_IVE_fut_prices.parquet')
         df = pd.read_parquet(infp)
         cprint(df)

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

```
-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                      price      bid      ask    size         v           dv
```

```
dates
2018-02-26 15:59:59  115.35  115.34  115.36     700     700     80745.0
2018-02-26 16:00:00  115.35  115.34  115.35    5362    5362    618506.7
2018-02-26 16:10:00  115.35  115.22  115.58       0       0         0.0
2018-02-26 16:16:14  115.30  114.72  115.62  778677  778677  89781458.1
2018-02-26 18:30:00  115.35  114.72  117.38       0       0         0.0
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1293578 entries, 2009-09-28 09:30:00 to 2018-02-26 18:30:00
Data columns (total 6 columns):
price    1293578 non-null float64
bid      1293578 non-null float64
ask      1293578 non-null float64
size     1293578 non-null int64
v        1293578 non-null int64
dv       1293578 non-null float64
dtypes: float64(4), int64(2)
memory usage: 69.1 MB
None
--------------------------------------------------------------------------------

[########################################] | 100% Completed | 26.0s
--------------------------------------------------------------------------------
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
--------------------------------------------------
<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
--------------------------------------------------------------------------------
```

4.1 [4.1]

4.1.1 (a) compute a t1 series using dollar bars derived from dataset

```
In [11]: dbars = brs.dollar_bar_df(dv_rs, 'dv', 1_000_000)
         cprint(dbars)

         close = dbars.price.copy()
         dailyVol = snp.getDailyVol(close)
         cprint(dailyVol)

         tEvents = snp.getTEvents(close,h=dailyVol.mean())
         print(tEvents)
         #cprint(tEvents)

         t1 = snp.addVerticalBarrier(tEvents, close)
         cprint(t1)

         # select profit taking stoploss factor
         ptsl = [1,1]
         # target is dailyVol computed earlier
         target=dailyVol
         # select minRet
         minRet = 0.005
         # get cpu count - 1
         cpus = cpu_count() - 1

         events = snp.getEvents(close,tEvents,ptsl,target,minRet,cpus,t1=t1)
         cprint(events)
```

```
100%|| 941297/941297 [00:00<00:00, 2962224.78it/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
------------------------------------------------------------------------------

  4%|         | 1243/30859 [00:00<00:02, 12427.49it/s]

------------------------------------------------------------------------------
dataframe information
------------------------------------------------------------------------------
                     dailyVol
dates
2018-02-26 15:31:06  0.006852
2018-02-26 15:40:15  0.006893
2018-02-26 15:49:42  0.006889
2018-02-26 15:59:04  0.006894
2018-02-26 16:16:14  0.006902
-------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 30844 entries, 2009-09-29 10:03:18 to 2018-02-26 16:16:14
Data columns (total 1 columns):
dailyVol    30843 non-null float64
dtypes: float64(1)
memory usage: 481.9 KB
None
------------------------------------------------------------------------------

100%|| 30859/30859 [00:01<00:00, 16113.68it/s]

DatetimeIndex(['2009-09-28 13:54:29', '2009-09-29 09:33:01',
               '2009-09-29 10:40:29', '2009-09-29 12:53:47',
               '2009-09-30 09:45:21', '2009-09-30 11:32:34',
```

```
               '2009-09-30 14:41:36', '2009-10-01 09:43:58',
               '2009-10-01 10:36:11', '2009-10-01 13:33:25',
               ...
               '2018-02-22 10:39:58', '2018-02-22 12:50:26',
               '2018-02-22 14:44:33', '2018-02-22 15:44:31',
               '2018-02-23 09:30:00', '2018-02-23 12:09:26',
               '2018-02-23 15:02:21', '2018-02-26 09:30:00',
               '2018-02-26 11:12:44', '2018-02-26 15:12:24'],
              dtype='datetime64[ns]', length=4610, freq=None)
-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                                  dates
2018-02-22 14:44:33 2018-02-23 15:02:21
2018-02-22 15:44:31 2018-02-23 15:51:23
2018-02-23 09:30:00 2018-02-26 09:30:00
2018-02-23 12:09:26 2018-02-26 09:30:00
2018-02-23 15:02:21 2018-02-26 09:30:00
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4607 entries, 2009-09-28 13:54:29 to 2018-02-23 15:02:21
Data columns (total 1 columns):
dates    4607 non-null datetime64[ns]
dtypes: datetime64[ns](1)
memory usage: 72.0 KB
None
-------------------------------------------------------------------------------



2018-10-18 16:52:19.124984 100.0% applyPtSlOnT1 done after 0.01 minutes. Remaining 0.0 minutes.


-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                                     t1      trgt
2018-02-23 12:09:26 2018-02-26 09:30:00  0.006306
2018-02-23 15:02:21 2018-02-26 09:30:00  0.006102
2018-02-26 09:30:00 2018-02-26 15:40:15  0.006995
2018-02-26 11:12:44 2018-02-26 15:20:10  0.006694
2018-02-26 15:12:24                 NaT  0.006757
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3757 entries, 2009-09-30 09:45:21 to 2018-02-26 15:12:24
Data columns (total 2 columns):
t1      3756 non-null datetime64[ns]
trgt    3757 non-null float64
```

```
dtypes: datetime64[ns](1), float64(1)
memory usage: 88.1 KB
None
-------------------------------------------------------------------------------
```

4.1.2 (b) Apply the function `mpNumCoEvents` to compute the number of overlapping outcomes at each point in time.

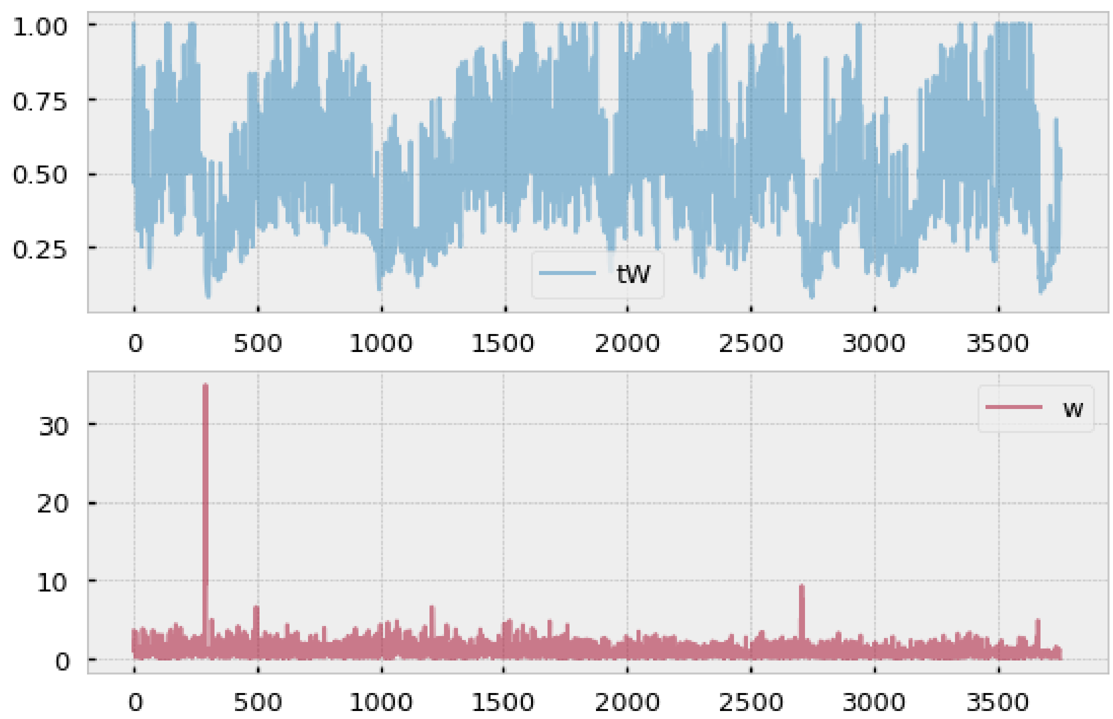
```
In [12]: ## Example
         numCoEvents = snp.mpPandasObj(snp.mpNumCoEvents,('molecule',events.index),
                                       cpus,closeIdx=close.index,t1=events['t1'])
         numCoEvents = numCoEvents.loc[~numCoEvents.index.duplicated(keep='last')]
         numCoEvents = numCoEvents.reindex(close.index).fillna(0)
         out=pd.DataFrame()
         out['tW'] = snp.mpPandasObj(snp.mpSampleTW,('molecule',events.index),
                                     cpus,t1=events['t1'],numCoEvents=numCoEvents)
         ## 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 16:52:20.212208 100.0% mpNumCoEvents done after 0.0 minutes. Remaining 0.0 minutes..
2018-10-18 16:52:20.923674 100.0% mpSampleTW done after 0.0 minutes. Remaining 0.0 minutes..
2018-10-18 16:52:21.688889 100.0% mpSampleW done after 0.0 minutes. Remaining 0.0 minutes..


-------------------------------------------------------------------------------
dataframe information
-------------------------------------------------------------------------------
                           tW         w
2018-02-23 12:09:26  0.377778  1.230338
2018-02-23 15:02:21  0.440476  1.181118
2018-02-26 09:30:00  0.580000  1.084218
2018-02-26 11:12:44  0.479167  1.123437
2018-02-26 15:12:24       NaN  0.000000
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3757 entries, 2009-09-30 09:45:21 to 2018-02-26 15:12:24
Data columns (total 2 columns):
tW    3756 non-null float64
w     3757 non-null float64
dtypes: float64(2)
memory usage: 88.1 KB
None
```

```
-------------------------------------------------------------------------------
```

```
In [13]: fig, ax = plt.subplots(figsize=(9,6))
         out.reset_index(drop=True).plot(subplots=True, alpha=0.5, ax=ax);
```

```
/media/bcr/HDD/anaconda3/envs/bayes_dash/lib/python3.6/site-packages/pandas/plotting/_core.py:18
  plot_obj.generate()
```



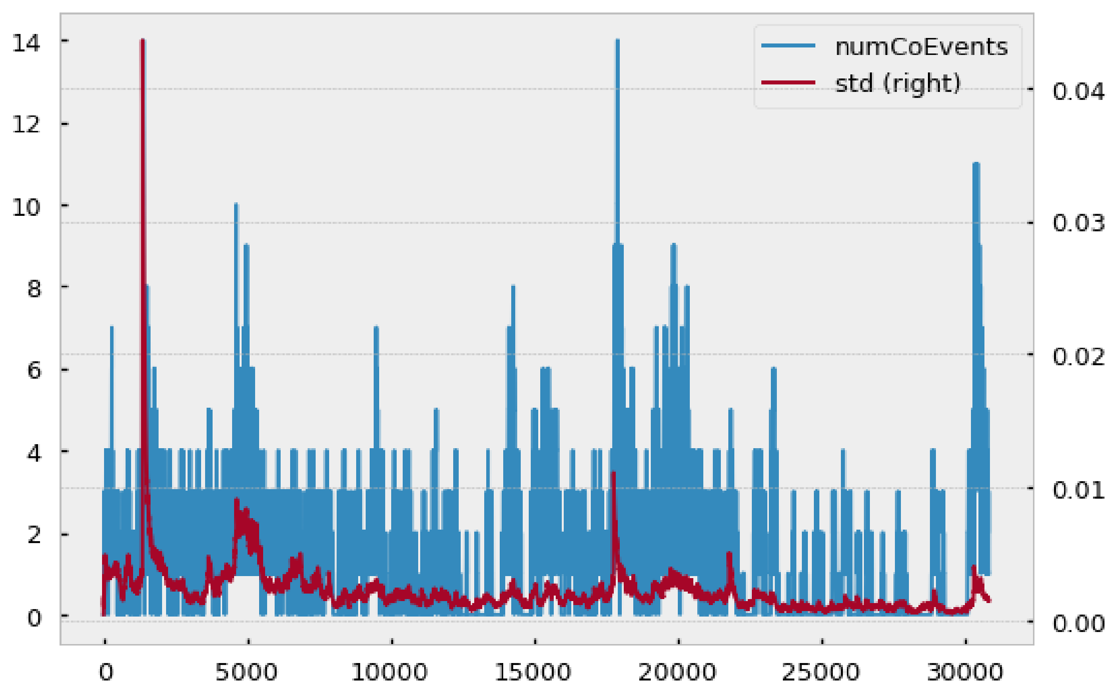
4.1.3 (c) Plot the time series of number of concurrent labels on primary axis and time series of exponentially weighted moving standard deviation of returns on secondary axis

```
In [14]: coEvents_std = (
             pd.DataFrame()
             .assign(
                 numCoEvents = numCoEvents.reset_index(drop=True),
                 std = brs.returns(dbars.price).ewm(50).std().reset_index(drop=True))
         )
         cprint(coEvents_std)
```

```
-------------------------------------------------------------------------------
dataframe information
```

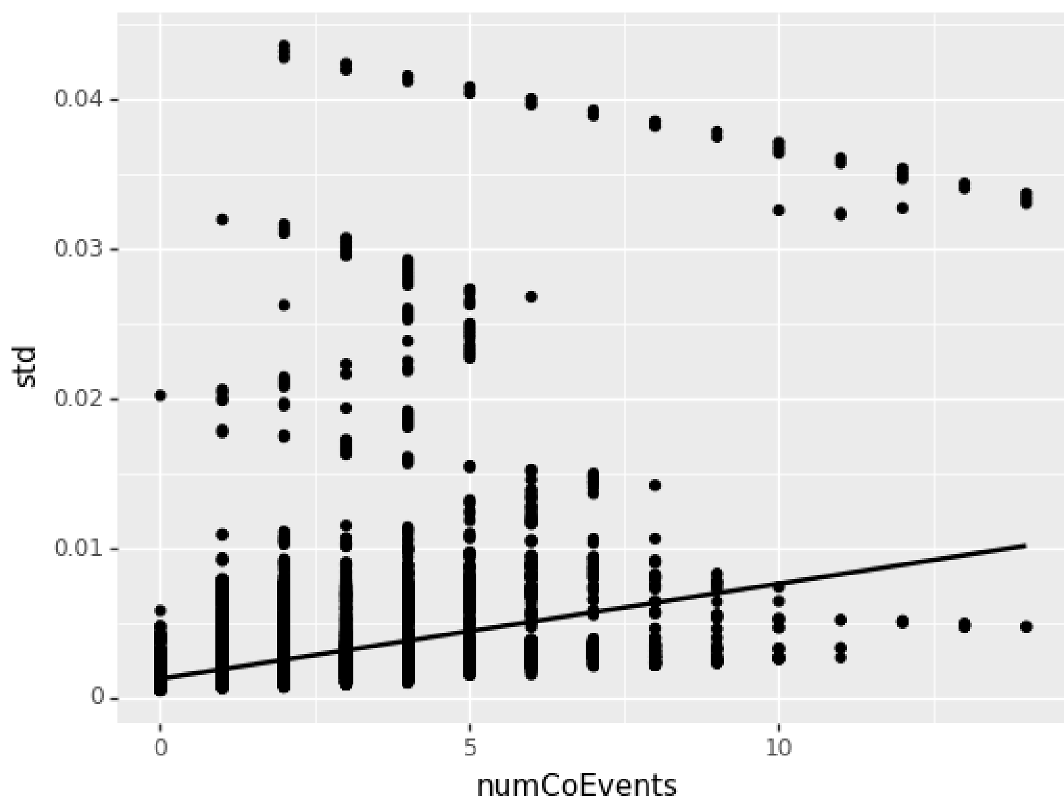
```
--------------------------------------------------------------------------------
       numCoEvents       std
30856          2.0  0.001482
30857          2.0  0.001496
30858          1.0  0.001483
30859          1.0  0.001468
30860          1.0       NaN
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30861 entries, 0 to 30860
Data columns (total 2 columns):
numCoEvents    30861 non-null float64
std            30859 non-null float64
dtypes: float64(2)
memory usage: 482.3 KB
None
--------------------------------------------------------------------------------
```

```
In [15]: fig, ax = plt.subplots(figsize=(9,6))

         coEvents_std.numCoEvents.plot(legend=True, ax=ax)
         coEvents_std['std'].plot(secondary_y=True, legend=True, ax=ax)

Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x7f742f5d88d0>
```



4.1.4 (d) Produce a scatterplot of the number of concurrent labels (x-axis) and the exponentially weighted moving std dev of returns (y-axis).

In [16]:
```
import warnings
warnings.filterwarnings("ignore")

(pn.ggplot(coEvents_std, pn.aes('numCoEvents', 'std'))
 +pn.geom_point()
 +pn.stat_smooth())
```



Out[16]: <ggplot: (8758561724480)>

4.2 [4.2] Using the function `mpSampleTW` compute the avg uniqueness of each label. What is the first-order serial correlation, `AR(1)` of this time series? Is it statistically significant? Why?

In [17]:
```
lag = 1
lag_col = f'tW_lag_{lag}'
out[lag_col] = out['tW'].shift(lag)
cprint(out.dropna())
```

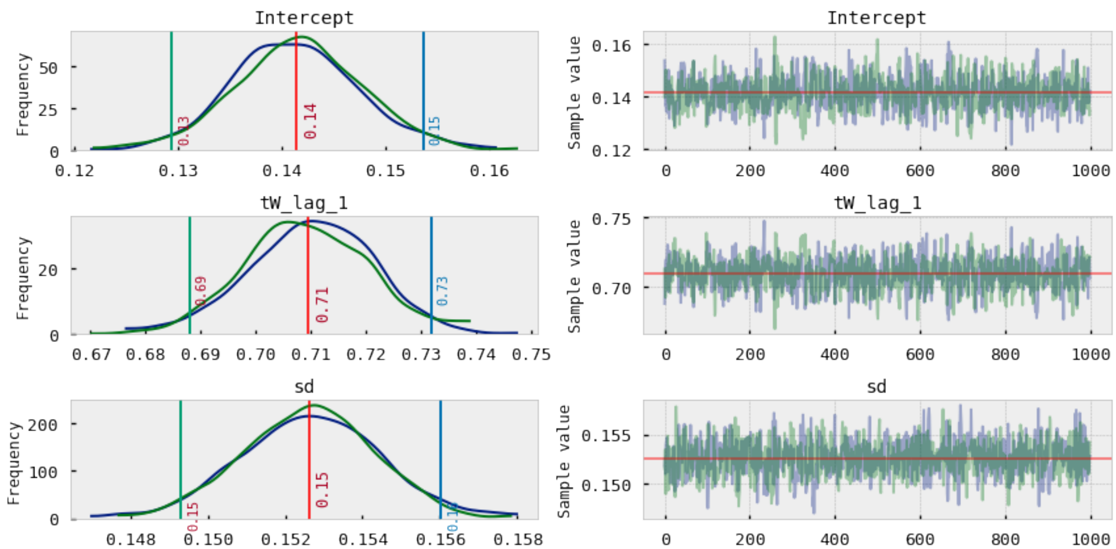
```
--------------------------------------------------------------------------------
dataframe information
--------------------------------------------------------------------------------
                           tW         w  tW_lag_1
2018-02-23 09:30:00  0.296296  0.895596  0.272619
2018-02-23 12:09:26  0.377778  1.230338  0.296296
2018-02-23 15:02:21  0.440476  1.181118  0.377778
2018-02-26 09:30:00  0.580000  1.084218  0.440476
2018-02-26 11:12:44  0.479167  1.123437  0.580000
--------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3755 entries, 2009-09-30 11:32:34 to 2018-02-26 11:12:44
Data columns (total 3 columns):
tW          3755 non-null float64
w           3755 non-null float64
tW_lag_1    3755 non-null float64
dtypes: float64(3)
memory usage: 117.3 KB
None
--------------------------------------------------------------------------------
```

```
In [18]: with pm.Model() as mdl:
             pm.GLM.from_formula(f'tW ~ {lag_col}', out.dropna())
             trace = pm.sample(3000, cores=1, nuts_kwargs={'target_accept':0.95})
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Sequential sampling (2 chains in 1 job)
NUTS: [sd, tW_lag_1, Intercept]
100%|| 3500/3500 [00:11<00:00, 307.60it/s]
100%|| 3500/3500 [00:16<00:00, 214.81it/s]
```

```
In [19]: plt.figure(figsize=(9, 6))
         plot_traces(trace, retain=1_000)
         plt.tight_layout();
```

```
<Figure size 648x432 with 0 Axes>
```




```
In [20]: df_smry = pm.summary(trace[1000:])
         df_smry
```

Out[20]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5	n_eff
Intercept	0.141484	0.006083	0.000136	0.130403	0.154605	1423.400118
tW_lag_1	0.709362	0.011346	0.000256	0.687189	0.731663	1431.785680
sd	0.152635	0.001739	0.000038	0.149174	0.155986	2091.837602

	Rhat
Intercept	0.999822
tW_lag_1	1.000334
sd	1.000220

The first order correlation between `tW` and `tW lag 1` appears statistically significant as all the mass of the distribution is nonzero.

4.3 [4.3] Fit a random forest to a financial dataset where $I^{-1}\sum_{i=1}^{I}\bar{u} \ll 1$

4.3.1 (a) What is the mean out of bag accuracy?

```
In [21]: from sklearn.model_selection import train_test_split
         from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor

         Xy = (pd.DataFrame()
               .assign(close=close,
                       close_lag=close.shift(1))
              ).dropna()

         y = Xy.loc[:,'close'].values
```

```
X = Xy.loc[:,'close_lag'].values.reshape(-1,1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
                                                    shuffle=False)

n_estimator = 50
rf = RandomForestRegressor(max_depth=1, n_estimators=n_estimator,
                           criterion='mse', oob_score=True,
                           random_state=RANDOM_STATE)
rf.fit(X_train, y_train)
```

Out[21]:
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=1,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=50, n_jobs=None,
           oob_score=True, random_state=777, verbose=0, warm_start=False)
```

In [22]:
```
rf.oob_score_
```

Out[22]:
```
0.846752960649153
```

4.3.2 (b) What is the mean accuracy of k-fold cross-validation (without shuffling) on the same dataset?

In [23]:
```
from sklearn.model_selection import cross_validate

n_estimator = 50
rf = RandomForestRegressor(max_depth=1, n_estimators=n_estimator,
                           criterion='mse', oob_score=True,
                           random_state=RANDOM_STATE)

scores = cross_validate(rf, X, y, cv=5, return_estimator=True)
```

In [24]:
```
oob_scores = [est.oob_score_ for est in scores['estimator']]
oob_scores, np.mean(oob_scores)
```

Out[24]:
```
([0.7252605826044242,
  0.8075575721618704,
  0.7617962610176714,
  0.7575686086929522,
  0.8310269271466907],
 0.7766419903247217)
```

4.3.3 Why is out-of-bag accuracy so much higher than cross-validation accuracy? Which one is more correct / less biased? What is the source of this bias?

Out of bag accuracy is higher than cross-validation b/c the incorrect assumption of IID draws leads to oversampling of redudant samples.

For random forests this means that the trees too similar. The random sampling also means that in-bag and out-of-bag samples will be similar inflating the `oob_score_`. In this example the cross-validation is less-biased.

4.4 Modify the code in Section 4.7 to apply an exponential time-decay factor

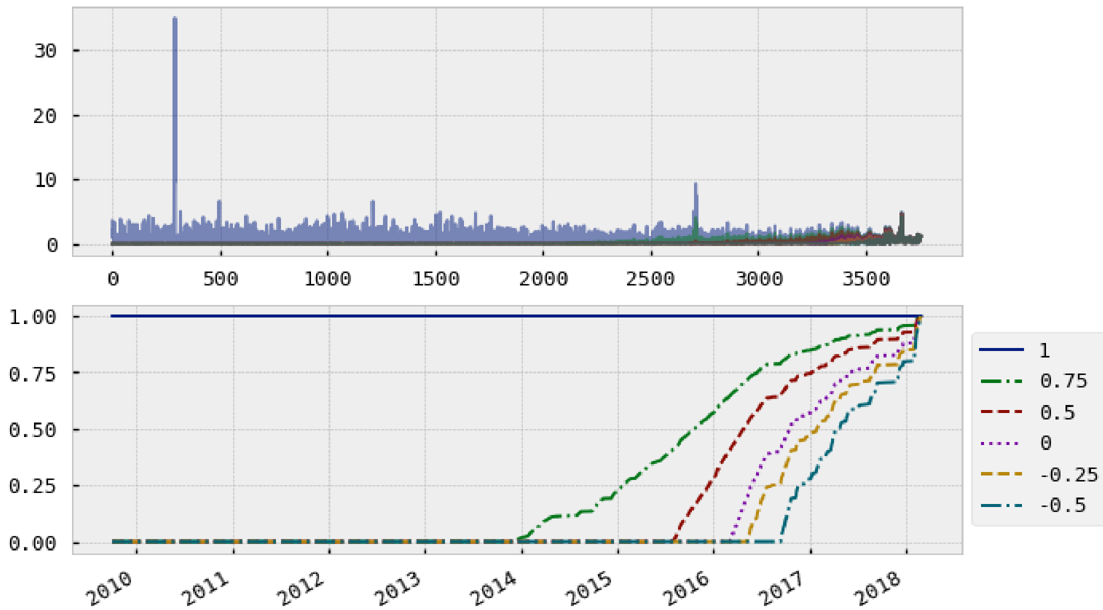
Note: would like to see alternative implementations here.

```
In [25]: def getExTimeDecay(tW,clfLastW=1.,exponent=1):
             # apply exponential decay to observed uniqueness (tW)
             # newest observation gets weight=1, oldest observation gets weight=clfLastW
             clfW=tW.sort_index().cumsum()
             if clfLastW>=0: slope=((1.-clfLastW)/clfW.iloc[-1])**exponent
             else: slope=(1./((clfLastW+1)*clfW.iloc[-1]))**exponent
             const=1.-slope*clfW.iloc[-1]
             clfW=const+slope*clfW
             clfW[clfW<0]=0
             print(round(const,4), round(slope,4))
             return clfW
```

```
In [26]: f,ax=plt.subplots(2,figsize=(10,7))
         fs = [1,.75,.5,0,-.25,-.5]
         ls = ['-','-.','--',':','--','-.']
         for lstW, l in zip(fs,ls):
             decayFactor = getExTimeDecay(out['tW'].dropna(),
                                          clfLastW=lstW,
                                          exponent=0.75) # experiment by changing exponent
             ((out['w'].dropna()*decayFactor).reset_index(drop=True)
              .plot(ax=ax[0],alpha=0.5))
             s = (pd.Series(1,index=out['w'].dropna().index)*decayFactor)
             s.plot(ax=ax[1], ls=l, label=str(lstW))
         ax[1].legend(loc='center left', bbox_to_anchor=(1, 0.5))
```

```
1.0 0.0
-1.3121 0.0013
-2.8885 0.0021
-5.5396 0.0036
-7.1144 0.0044
-9.9983 0.006
```

```
Out[26]: <matplotlib.legend.Legend at 0x7f73906abeb8>
```



4.5 Consider you have applied meta-labels to events determined by a trend-following model. Suppose `2/3` of labels are `0` and `1/3` are `1`.

4.5.1 (a) What happens if you fit a classifier without balancing class weights?

The classifier will maximize accuracy by over predicting the dominant class

4.5.2 (b) A label `1` means true positive and a label `0` means a false positive. By applying balanced class weights, we are forcing the classifier to pay more attention to the true positives, and less attention to the false positives. Why does that make sense?

Tying the output to real-life purpose means that too many false positives result in bad trades/investments which means lost capital. From a ML perspective without balanced class weights we will maximize accuracy by simply predicting the dominant class. We need to improve `precision: TP/(TP+FP)` relative to `recall: TP/(TP+FN)` not just overall accuracy `(TP+TN)/(TP+FP+TN+FN)`.

4.5.3 (c) What is the distribution of the predicted labels, before and after applying balanced class weights?

Before balanced class weights is an unbalanced or skewed distribution. After balanced class weights predicted labels would be more evenly distributed depending on the predictive power of the feature set.

4.6 Update the draw probabilities for the final draw in section `4.5.3`.

Note: Could use assistance understanding and breaking down how to compute this intuitively

```
In [27]: t1=pd.Series([2,3,5],index=[0,2,4]) # t0,t1 for each feature obs
         barIx=range(t1.max()+1) # index of bars
         indM=snp.getIndMatrix(barIx,t1)
         print(indM)
         phi=np.random.choice(indM.columns,size=indM.shape[1])
```

```
     0    1    2
0  1.0  0.0  0.0
1  1.0  0.0  0.0
2  1.0  1.0  0.0
3  0.0  1.0  0.0
4  0.0  0.0  1.0
5  0.0  0.0  1.0
```

4.7 In Section 4.5.3 suppose that number 2 is picked again in the second draw. Waht would be the updated probabilities for the third draw?

Note: Could use assistance understanding and breaking down how to compute this intuitively

```
In [ ]:
```