

Empirical Corporate Finance Project:

Event Study of Lawsuits

Xinyu Liu

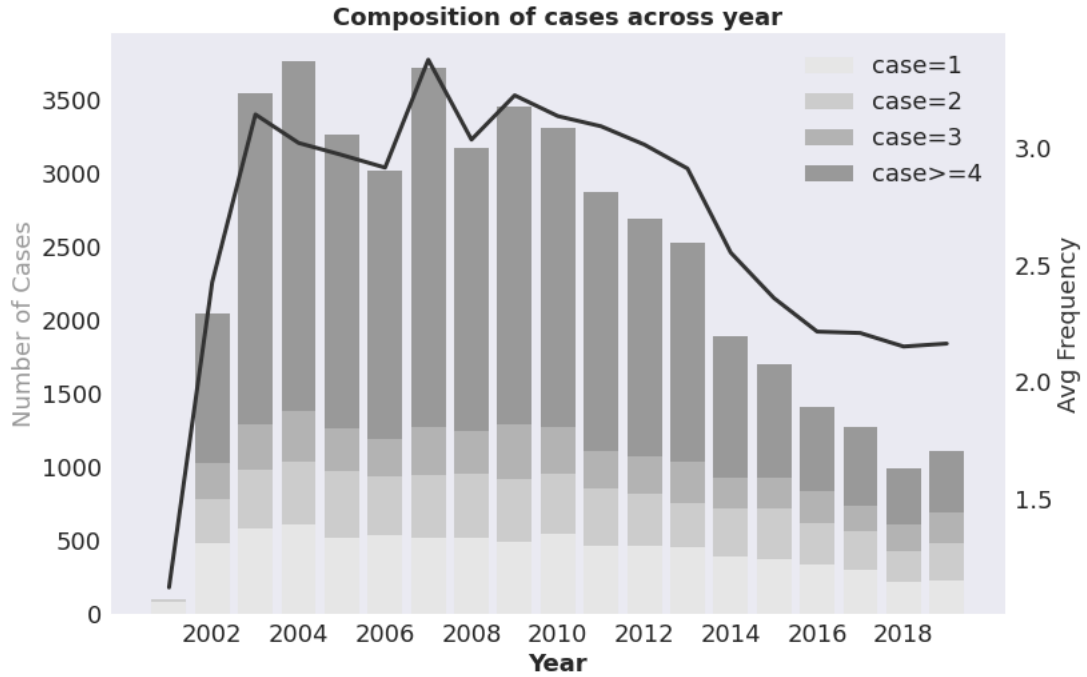
April 17, 2021

1. Data Processing

I use Python to analyze the data (see Appendix B the full code, which is provided by WRDs Python Replication). The Data range of this exercise is 01.2000 – 12.2019, a total span of 20 years. I download lawsuit event announcement date and company GVKEY records from Compustats-Key Development; Linktable of Compustats and CRSP from CCM; and daily return data from CRSP. I first explore the features of Lawsuits across time and across firms, and then calculate mean CAR for the default estimation and event window. Lastly, as a verification check, the automatically generated graph from WRDs Event Studies is given in the Appendix A. The complete output data can be accessed on Github.

Below is the histogram of lawsuit cases across year. Based on Figure 1 I hope to highlight the heterogeneity of number of lawsuits in each year, as well as the total number of lawsuits for each company. After merging with CRSP, there are 45845 event-permno pairs left in the sample. However, they are not evenly distributed across time, neither are they evenly distributed across firms. For example, the amount of cases keeps dropping after 2010, especially for the big companies which used to have many lawsuits going on. This is consistent with the trend of frequency, which indicates that case-involved firms have on average 2 – 3 cases every year, although this is highly heterogeneous. Note that the majority of cases come from companies that are involved in many cases, therefore the CAR I estimate later will inevitably mostly reflect these case-concentrated firms. In addition, the frequency of lawsuits for a typical company is a crucial feature when it comes to choose proper parameters for estimation and event window, here for simplicity I take the default window configuration from WRDs, which can be a potential problem if there are other events in the estimating window.

Figure 1

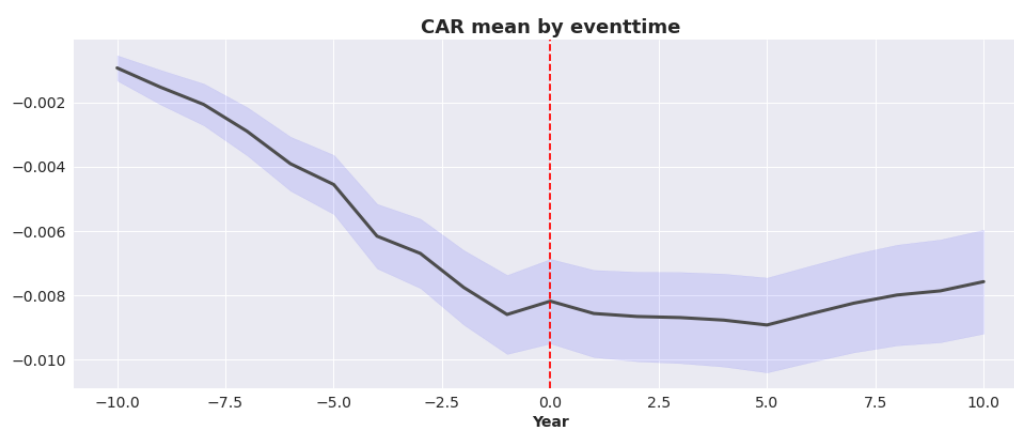


Note: Lawsuit-permno pairs are grouped by year to get the number of cases (left and bar). I calculate the number of unique permno in each year to get the number of companies sued in each year, which is used to calculate the average frequency (right and line) by dividing number of cases with it. In order to get a sense of how the number of cases are distributed across companies, I classify companies of different cases in each year, and display their corresponding part with different colors in the stacked bar.

2. CAR Estimation

For each pair of permno-time, I then estimate the market adjusted model to calculate the expected return, and accumulate the difference between actual return and expected return as CAR over the event window. Lastly, I take the average of all CARs to get the mean and inference. Below Figure 2 gives the main result of mean CAR. This graph aligns well with the one generated by WRDs, which is displayed in Figure 3. Basically there is a strong sign of information linkage for lawsuits, the CAR begins to drop steadily 10 days ahead of announcement, to the lowest of about 1%. Interestingly, during the announcement date there is no significant abnormal return. Longer period of event study shows that this CAR will eventually go back to 0, after about a month from the announcement. My interpretation is that lawsuits are more of a temporary shock and do not last long before the price is corrected based on fundamentals.

Figure 2



Note: Event window $[-10, 10]$, gap window $[-60, -10]$, estimation window $[-160, -60]$, normal model: adjusted market model.

Appendices

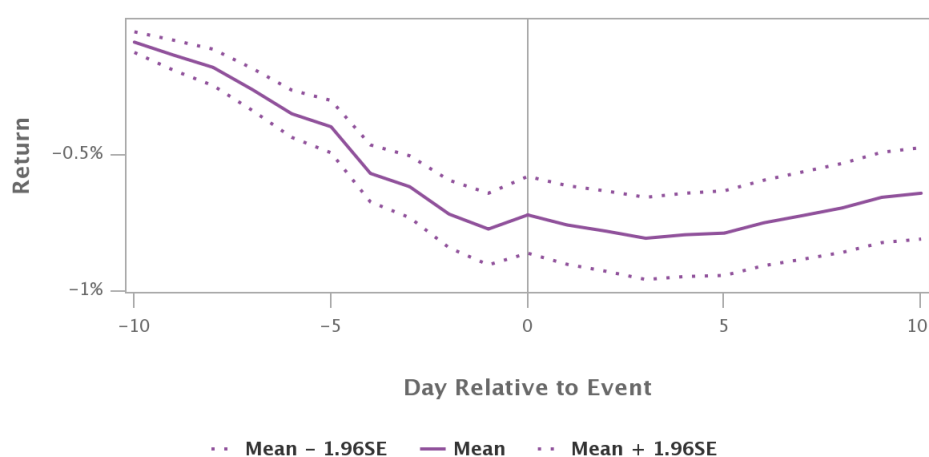
Appendix A

The following results are results generated directly by WRDs.

Figure 3

Cumulative Abnormal Return: Mean & 95% Confidence Limits

Event: "Lawsuits & Legal Issues". There are 47694 events in total with non-missing returns.



Highcharts.com

Appendix B

```
1 from datetime import datetime, date
2 from io import StringIO as StringIO_StringIO
3 from json import (
4     dumps as json_dumps,
5     dump as json_dump,
6     load as json_load,
7     JSONEncoder as json_JSONEncoder,
8 )
9 import os
10
11 from pandas import (
12     DataFrame as pd_DataFrame,
13     ExcelWriter as pd_ExcelWriter,
14 )
15 from numpy import (
16     abs as np_abs,
17     nan as np_nan,
18     mean as np_mean,
19     std as np_std,
20     sqrt as np_sqrt,
21     ndarray as np_ndarray,
22 )
23 from statsmodels.api import (
24     OLS as sm_OLS,
25     add_constant as sm_add_constant
26 )
27 from tabulate import tabulate
28 import wrds
29
30 class EncoderJson(json_JSONEncoder):
31     """
32     Class used to encodes to JSON data format
33     """
34
35     def default(self, obj):
36         if isinstance(obj, np_ndarray):
37             return obj.tolist()
38         elif isinstance(obj, datetime):
39             return obj.__str__()
40         elif isinstance(obj, date):
41             return obj.__str__()
42
43         return json_JSONEncoder.default(self, obj)
44
45 class EventStudy(object):
46     """
47     Main class that runs the event study.
48     """
49
50     #####
51     # STEP 0 - AUTHENTICATE AND CONNECT TO POSTGRES #
52     #####
53
54     # parameters when the class is initialized.
55     # pass an explicit output path for result file
56     def __init__(self, output_path=''):
57         if len(output_path) <= 0:
58             self.output_path = os.path.expanduser('~')
59         else:
60             self.output_path = output_path
61
62     # Connect to the Postgres database
63     # Code assumes pgpass file has been created
64     def connect(self):
65         """
66         Connect to the Postgres via WRDS.
67         """
68         self.wrdsconn = wrds.Connection()
```

```

71     self.conn = self.wrdsconn.connect()
72     return self.wrdsconn
73
74 # This is the method that gets called to run the event study. The "heavy lifting"
75 # happens here.
76 def eventstudy(self, data=None, model='m', estwin=100, gap=50, evtwins=-10, evtwine
77 =10, minval=70, output='df'):
78     """
79     Paramaters passed to the event study method.
80
81     data          =   event data (event date & permno combinations)
82     model          =   madj (market-adjusted model)
83                     m (market model)
84                     ff (fama french)
85                     ffm (fama french with momentum factor)
86     estwin         =   estimation window
87     gap            =   gap between estimation window and event window
88     evtwins        =   days preceding event date to begin event window
89     evtwine        =   days after event date to close the event window
90     minval         =   minimum number of non-missing return observations (per event
91 ) to be regressed on
92     output         =   output format of the event study results
93                     xls (output an excel file to output path)
94                     csv (output a csv file to output path)
95                     json (output a json file to output path)
96                     df (returns a dictionary of pandas dataframes)
97                     print (outputs results to the console - not available via
98 qsub)
99     """
100
101     #
102     #####
103     # STEP 1 - SET ESTIMATION, EVENT, AND GAP WINDOWS AND GRAB DATA FROM EVENTS
104     FILE #
105     #
106     #####
107
108     estwins = (estwin + gap + np_abs(evtwins)) # Estimation window start
109     estwine = (gap + np_abs(evtwins) + 1)      # Estimation window end
110     evtwinx = (estwins + 1)                    # evt time value (0=event date, -10=
111 window start, 10=window end)
112     evtwins = np_abs(evtwins)                  # convert the negative to positive
113 as we will use lag function)
114     evtrang = (evtwins + evtwine + 1)           # total event window days (lag +
115 lead + the day itself)
116
117     """
118     With the event date as a fixed point, calculate the number of days needed to
119     pass
120     to sql lag and lead functions to identify estimation window, gap, and event
121     window.
122
123     evtwins:      event date minus number of preceding days
124                   ("event date" - "number of days before event to start [evtwins
125 parameter]")
126
127     evtwine:      event date plus number of following days
128                   ("event date" + "number of days after event to end [evtwine
129 parameter]")
130
131     gap:          number of days between the end of the "estimation window"
132                   and the beginning of the "event window"
133
134     estwins:      start date of the estimation window
135                   ("event date" - "number of days before event to start [evtwins
136 parameter]"
137                   - "number of days in gap [gap parameter]"
138                   - "number of days in estimation window [estwin
139 parameter]")
140
141     evtrang:      entire time range of the event study even from estimate start,
142 through gap,
143                   until event window end

```

```

127         (evtwins + ewtwine + 1)
128     """
129
130     # default the event data in case it was not passed, otherwise read what was
131     # passed
132     evtdata = [{"edate": "05/29/2012", "permno": "10002"}]
133     if data is not None:
134         evtdata = json.dumps(data)
135
136     # init values wrapped up to be passed to sql statement
137     params = {'estwins': estwins, 'estwine': estwine, 'evtwins': ewtwins, 'ewtwine':
138         ewtwine, 'ewtwinx': ewtwinx, 'evtdata': evtdata}
139
140     #####
141     # STEP 2 - GET RETURNS DATA FROM POSTGRES #
142     #####
143
144     # Create a database connection
145     wconn = self.connect()
146
147     #####
148     # Get the initial data from the database and put it in a pandas dataframe #
149     #####
150
151     # create a pandas dataframe that will hold data
152     df = wconn.raw_sql("""
153     SELECT
154         a.*,
155         x.*,
156         c.date as rdate,
157         c.ret as ret1,
158         (f.mktrf+f.rf) as mkt,
159         f.mktrf,
160         f.rf,
161         f.smb,
162         f.hml,
163         f.umd,
164         (1+c.ret)*(coalesce(d.dlret,0.00)+1)-1-(f.mktrf+f.rf) as exret,
165         (1+c.ret)*(coalesce(d.dlret,0.00)+1)-1 as ret,
166         case when c.date between a.estwin1 and a.estwin2 then 1 else 0 end as
167         isest,
168         case when c.date between a.ewtwin1 and a.ewtwin2 then 1 else 0 end as
169         isewt,
170         case
171             when c.date between a.ewtwin1 and a.ewtwin2 then (rank() OVER (
172             PARTITION BY x.ewtid ORDER BY c.date)-%(ewtwinx)s)
173             else (rank() OVER (PARTITION BY x.ewtid ORDER BY c.date))
174             end as ewttime
175     FROM
176     (
177         SELECT
178             date,
179             lag(date, %(estwins)s ) over (order by date) as estwin1,
180             lag(date, %(estwine)s ) over (order by date) as estwin2,
181             lag(date, %(evtwins)s ) over (order by date) as ewtwin1,
182             lead(date, %(ewtwine)s ) over (order by date) as ewtwin2
183         FROM crsp_a_stock.dsi
184     ) as a
185     JOIN
186     (select
187         to_char(x.edate, 'ddMONYYYY') || trim(to_char(x.permno,'999999999')) as
188         ewtid,
189         x.permno,
190         x.edate
191     from
192         json_to_recordset('%(evtdata)s') as x(edate date, permno int)
193     ) as x
194     ON a.date=x.edate
195     JOIN crsp_a_stock.dsf c
196     ON x.permno=c.permno
197     AND c.date BETWEEN a.estwin1 and a.ewtwin2
198     JOIN ff_all.factors_daily f
199     ON c.date=f.date

```

```

194     LEFT JOIN crsp_a_stock.dsedelist d
195         ON x.permno=d.permno
196         AND c.date=d.dlstdt
197     WHERE f.mktrf is not null
198     AND c.ret is not null
199     ORDER BY x.evtid, x.permno, a.date, c.date
200     """ % params)
201
202     # Columns coming from the database query
203     df.columns = ['date', 'estwin1', 'estwin2', 'evtwin1', 'evtwin2',
204                  'evtid', 'permno', 'edate', 'rdate', 'ret1', 'mkt',
205                  'mktrf', 'rf', 'smb', 'hml', 'umd', 'exret', 'ret',
206                  'isest', 'isevt', 'evttime']
207
208     # Additional columns that will hold computed values (post-query)
209     addcols = ['RMSE', 'INTERCEPT', 'var_estp', 'expret', 'abret',
210               'alpha', '_nobs', '_p_', '_edf_', 'rsq', 'cret',
211               'cexpret', 'car', 'scar', 'sar', 'pat_scale', 'bhar',
212               'lastevtwin', 'cret_edate', 'scar_edate', 'car_edate',
213               'bhar_edate', 'pat_scale_edate', 'xyz']
214
215     # Add them to the dataframe
216     for c in addcols:
217         if c == 'lastevtwin':
218             df[c] = 0
219         else:
220             df[c] = np_nan
221
222     #
223     #####
224     # STEP 3 - FOR EACH EVENT, CALCULATE ABNORMAL RETURN BASED ON CHOSEN RISK MODEL
225     #
226     #####
227
228     # Loop on every category
229     for evt in data:
230
231         permno = evt['permno']
232         xdate = evt['edate']
233         edate = datetime.strptime(xdate, "%m/%d/%Y").date()
234
235         est_mask = (df['permno'] == permno) & (df['edate'] == edate) & (df['isest']
236 == 1)
237         evt_mask = (df['permno'] == permno) & (df['edate'] == edate) & (df['isevt']
238 == 1)
239
240         #####
241         # Check to see it meets the min obs for est window #
242         #####
243         _nobs = df["ret"][est_mask].count()
244
245         # Only carry out the analysis if the number of obseervations meets the minimum
246         threshold
247         if _nobs >= minval:
248
249             #####
250             # Regression based on model choices=' ' #
251             #####
252
253             # Market-Adjusted Model
254             if model == 'madj':
255                 # Set y to the estimation window records
256                 y = df["exret"][est_mask]
257
258                 # Calculate mean and standard deviation of returns for the
259                 estimation period
260                 mean = np_mean(y)
261                 stdv = np_std(y, ddof=1)
262
263                 # Update the columns in the original dataframe (reusing the names
264                 from SAS code to help with continuity)
265                 df.loc[evt_mask, 'INTERCEPT'] = mean

```

```

259     df.loc[evt_mask, 'RMSE'] = stdv
260     df.loc[evt_mask, '_nobs'] = len(y)
261     df.loc[evt_mask, 'var_estp'] = stdv ** 2
262     df.loc[evt_mask, 'alpha'] = mean
263     df.loc[evt_mask, 'rsq'] = 0
264     df.loc[evt_mask, '_p_'] = 1
265     df.loc[evt_mask, '_edf_'] = (len(y) - 1)
266     df.loc[evt_mask, 'expret'] = df.loc[evt_mask, 'mkt']
267     df.loc[evt_mask, 'abret'] = df.loc[evt_mask, 'exret']
268     df_est = df[est_mask]
269     _nobs = len(df_est[df_est.ret.notnull()])
270
271     nloc = {'const': 0}
272
273     def f_cret(row):
274         tmp = ((row['ret'] * nloc['const']) + (row['ret'] + nloc['const',
275
276         nloc['const'] = tmp
277         return tmp
278     df.loc[evt_mask, 'cret'] = df[evt_mask].apply(f_cret, axis=1)
279     df.loc[evt_mask, 'cret_edate'] = nloc['const']
280
281     nloc = {'const': 0}
282
283     def f_cexpret(row):
284         tmp = ((row['expret'] * nloc['const']) + (row['expret'] + nloc['const',
285         nloc['const'] = tmp
286         return tmp
287     df.loc[evt_mask, 'cexpret'] = df[evt_mask].apply(f_cexpret, axis=1)
288     df.loc[evt_mask, 'cexpret_edate'] = nloc['const']
289
290     nloc = {'const': 0}
291
292     def f_car(row):
293         tmp = (row['abret'] + nloc['const'])
294         nloc['const'] = tmp
295         return tmp
296     df.loc[evt_mask, 'car'] = df[evt_mask].apply(f_car, axis=1)
297     df.loc[evt_mask, 'car_edate'] = nloc['const']
298
299     nloc = {'const': 0}
300
301     def f_sar(row):
302         tmp = (row['abret'] / np.sqrt(row['var_estp']))
303         nloc['const'] = tmp
304         return tmp
305     df.loc[evt_mask, 'sar'] = df[evt_mask].apply(f_sar, axis=1)
306     df.loc[evt_mask, 'sar_edate'] = nloc['const']
307
308     nloc = {'const': 0, 'evtrang': evtrang}
309
310     def f_scar(row):
311         tmp = (row['car'] / np.sqrt((evtrang * row['var_estp'])))
312         nloc['const'] = tmp
313         return tmp
314     df.loc[evt_mask, 'scar'] = df[evt_mask].apply(f_scar, axis=1)
315     df.loc[evt_mask, 'scar_edate'] = nloc['const']
316
317     nloc = {'const': 0}
318
319     def f_bhar(row):
320         tmp = (row['cret'] - row['cexpret'])
321         nloc['const'] = tmp
322         return tmp
323     df.loc[evt_mask, 'bhar'] = df[evt_mask].apply(f_bhar, axis=1)
324     df.loc[evt_mask, 'bhar_edate'] = nloc['const']
325
326     df.loc[evt_mask, 'pat_scale'] = (_nobs - 2.00) / (_nobs - 4.00)
327     df.loc[evt_mask, 'pat_scale_edate'] = (_nobs - 2.00) / (_nobs -
328
4.00)
329
330     # Market Model
331     elif model == 'm':

```



```

329         # Set y to the estimation window records
330         X = df["mktrf"][est_mask]
331         y = df["ret"][est_mask]
332
333         # Fit an OLS model with intercept on mktrf
334         X = sm_add_constant(X)
335         est = sm_OLS(y, X).fit()
336
337         # Set the variables from the output
338         df_est = df[(df['permno'] == permno) & (df['edate'] == edate) & (df[
'isest'] == 1)]
339         _nobs = len(df_est[df_est.ret.notnull()]) # not null observations
340
341         # aggregate variables
342         # cret_edate = np_nan
343         # scar_edate = np_nan
344         # car_edate = np_nan
345         # bhar_edate = np_nan
346         # pat_scale_edate = np_nan
347         alpha = est.params._getitem_('const')
348         beta1 = est.params._getitem_('mktrf')
349
350         df.loc[evt_mask, 'INTERCEPT'] = alpha
351         df.loc[evt_mask, 'alpha'] = alpha
352         df.loc[evt_mask, 'RMSE'] = np_sqrt(est.mse_resid)
353         df.loc[evt_mask, '_nobs'] = _nobs
354         df.loc[evt_mask, 'var_estp'] = est.mse_resid
355         df.loc[evt_mask, 'rsq'] = est.rsquared
356         df.loc[evt_mask, '_p_'] = 2
357         df.loc[evt_mask, '_edf_'] = (len(y) - 2)
358
359         nloc = {'alpha': alpha, 'beta1': beta1, 'const': 0}
360
361         def f_expret(row):
362             return (nloc['alpha'] + (nloc['beta1'] * row['mktrf']))
363         df.loc[evt_mask, 'expret'] = df[evt_mask].apply(f_expret, axis=1)
364
365         nloc = {'alpha': alpha, 'beta1': beta1, 'const': 0}
366
367         def f_abret(row):
368             return (row['ret'] - (nloc['alpha'] + (nloc['beta1'] * row[
mktrf']))))
369
370         df.loc[evt_mask, 'abret'] = df[evt_mask].apply(f_abret, axis=1)
371
372         nloc = {'const': 0}
373
374         def f_cret(row):
375             tmp = ((row['ret'] * nloc['const']) + (row['ret'] + nloc['const'
]))
376             nloc['const'] = tmp
377             return tmp
378         df.loc[evt_mask, 'cret'] = df[evt_mask].apply(f_cret, axis=1)
379         df.loc[evt_mask, 'cret_edate'] = nloc['const']
380
381         nloc = {'const': 0}
382
383         def f_cexpret(row):
384             tmp = ((row['expret'] * nloc['const']) + (row['expret'] + nloc[
const'])))
385             nloc['const'] = tmp
386             return tmp
387         df.loc[evt_mask, 'cexpret'] = df[evt_mask].apply(f_cexpret, axis=1)
388
389         nloc = {'const': 0}
390
391         def f_car(row):
392             # nonlocal const
393             tmp = (row['abret'] + nloc['const'])
394             nloc['const'] = tmp
395             return tmp
396         df.loc[evt_mask, 'car'] = df[evt_mask].apply(f_car, axis=1)
397         df.loc[evt_mask, 'car_edate'] = nloc['const']

```

```

398     nloc = {'const': 0}
399
400     def f_sar(row):
401         tmp = (row['abret'] / np_sqrt(row['var_estp']))
402         nloc['const'] = tmp
403         return tmp
404     df.loc[evt_mask, 'sar'] = df[evt_mask].apply(f_sar, axis=1)
405     df.loc[evt_mask, 'sar_edate'] = nloc['const']
406
407     nloc = {'const': 0, 'evtrang': evtrang}
408
409     def f_scar(row):
410         tmp = (row['car'] / np_sqrt((evtrang * row['var_estp'])))
411         nloc['const'] = tmp
412         return tmp
413     df.loc[evt_mask, 'scar'] = df[evt_mask].apply(f_scar, axis=1)
414     df.loc[evt_mask, 'scar_edate'] = nloc['const']
415
416     nloc = {'const': 0}
417
418     def f_bhar(row):
419         tmp = (row['cret'] - row['cexpret'])
420         nloc['const'] = tmp
421         return tmp
422     df.loc[evt_mask, 'bhar'] = df[evt_mask].apply(f_bhar, axis=1)
423     df.loc[evt_mask, 'bhar_edate'] = nloc['const']
424
425     df.loc[evt_mask, 'pat_scale'] = (_nobs - 2.00) / (_nobs - 4.00)
426     df.loc[evt_mask, 'pat_scale_edate'] = (_nobs - 2.00) / (_nobs -
4.00)
427
428     # Fama-French Three Factor Model
429     elif model == 'ff':
430         # Set y to the estimation window records
431         df_est = df[(df['permno'] == permno) & (df['edate'] == edate) & (df[
'isest'] == 1)]
432         X = df_est[['smb', 'hml', 'mktrf']]
433         y = df_est['ret']
434
435         # Fit an OLS model with intercept on mktrf, smb, hml
436         X = sm_add_constant(X)
437         est = sm_OLS(y, X).fit()
438         # est = smf.ols(formula='ret ~ smb + hml + mktrf', data=df_est).fit
439         ()
440
441         alpha = est.params.__getitem__('const')
442         beta1 = est.params.__getitem__('mktrf')
443         beta2 = est.params.__getitem__('smb')
444         beta3 = est.params.__getitem__('hml')
445
446         df.loc[evt_mask, 'INTERCEPT'] = alpha
447         df.loc[evt_mask, 'alpha'] = alpha
448         df.loc[evt_mask, 'RMSE'] = np_sqrt(est.mse_resid)
449         df.loc[evt_mask, '_nobs'] = _nobs
450         df.loc[evt_mask, 'var_estp'] = est.mse_resid
451         df.loc[evt_mask, 'rsq'] = est.rsquared
452         df.loc[evt_mask, '_p_'] = 2
453         df.loc[evt_mask, '_edf_'] = (len(y) - 2)
454
455         nloc = {'alpha': alpha, 'beta1': beta1, 'beta2': beta2, 'beta3':
beta3, 'const': 0}
456
457         def f_expret(row):
458             return ((nloc['alpha'] + (nloc['beta1'] * row['mktrf']) + (nloc[
'beta2'] * row['smb']) + (nloc['beta3'] * row['hml'])))
459         df.loc[evt_mask, 'expret'] = df[evt_mask].apply(f_expret, axis=1)
460
461         nloc = {'alpha': alpha, 'beta1': beta1, 'beta2': beta2, 'beta3':
beta3, 'const': 0}
462
463         def f_abret(row):
464             return (row['ret'] - ((nloc['alpha'] + (nloc['beta1'] * row[
mktrf']) + (nloc['beta2'] * row['smb']) + (nloc['beta3'] * row['hml']))))

```

```

464         df.loc[evt_mask, 'abret'] = df[evt_mask].apply(f_abret, axis=1)
465
466         nloc = {'const': 0}
467
468         def f_cret(row):
469             tmp = ((row['ret'] * nloc['const']) + (row['ret'] + nloc['const',
470 ]))
471             nloc['const'] = tmp
472             return tmp
473         df.loc[evt_mask, 'cret'] = df[evt_mask].apply(f_cret, axis=1)
474         df.loc[evt_mask, 'cret_edate'] = nloc['const']
475
476         nloc = {'const': 0}
477
478         def f_cexpret(row):
479             tmp = ((row['expret'] * nloc['const']) + (row['expret'] + nloc['
const'])))
480             nloc['const'] = tmp
481             return tmp
482         df.loc[evt_mask, 'cexpret'] = df[evt_mask].apply(f_cexpret, axis=1)
483         nloc = {'const': 0}
484
485         def f_car(row):
486             tmp = (row['abret'] + nloc['const'])
487             nloc['const'] = tmp
488             return tmp
489         df.loc[evt_mask, 'car'] = df[evt_mask].apply(f_car, axis=1)
490         df.loc[evt_mask, 'car_edate'] = nloc['const']
491
492         nloc = {'const': 0}
493
494         def f_sar(row):
495             tmp = (row['abret'] / np_sqrt(row['var_estp']))
496             nloc['const'] = tmp
497             return tmp
498         df.loc[evt_mask, 'sar'] = df[evt_mask].apply(f_sar, axis=1)
499         df.loc[evt_mask, 'sar_edate'] = nloc['const']
500
501         nloc = {'const': 0, 'evtrang': evtrang}
502
503         def f_scar(row):
504             tmp = (row['car'] / np_sqrt((evtrang * row['var_estp'])))
505             nloc['const'] = tmp
506             return tmp
507         df.loc[evt_mask, 'scar'] = df[evt_mask].apply(f_scar, axis=1)
508         df.loc[evt_mask, 'scar_edate'] = nloc['const']
509
510         nloc = {'const': 0}
511
512         def f_bhar(row):
513             tmp = (row['cret'] - row['cexpret'])
514             nloc['const'] = tmp
515             return tmp
516         df.loc[evt_mask, 'bhar'] = df[evt_mask].apply(f_bhar, axis=1)
517         df.loc[evt_mask, 'bhar_edate'] = nloc['const']
518
519         df.loc[evt_mask, 'pat_scale'] = (_nobs - 2.00) / (_nobs - 4.00)
520         df.loc[evt_mask, 'pat_scale_edate'] = (_nobs - 2.00) / (_nobs -
4.00)
521
522         # Fama-French Plus Momentum
523         elif model == 'ffm':
524             # Set y to the estimation window records
525             df_est = df[(df['permno'] == permno) & (df['edate'] == edate) & (df[
'isest'] == 1)]
526
527             X = df_est[['mktrf', 'smb', 'hml', 'umd']] # indicator variables
528             y = df_est['ret'] # response variables
529
530             # Fit an OLS (ordinary least squares) model with intercept on mktrf,
531             smb, hml, and umd
532             X = sm_add_constant(X)
533             est = sm_OLS(y, X).fit()

```

```

532
533     alpha = est.params.__getitem__('const')
534     beta1 = est.params.__getitem__('mktrf')
535     beta2 = est.params.__getitem__('smb')
536     beta3 = est.params.__getitem__('hml')
537     beta4 = est.params.__getitem__('umd')
538
539     df.loc[evt_mask, 'INTERCEPT'] = alpha
540     df.loc[evt_mask, 'alpha'] = alpha
541     df.loc[evt_mask, 'RMSE'] = np.sqrt(est.mse_resid)
542     df.loc[evt_mask, '_nobs'] = _nobs
543     df.loc[evt_mask, 'var_estp'] = est.mse_resid
544     df.loc[evt_mask, 'rsq'] = est.rsquared
545     df.loc[evt_mask, '_p_'] = 2
546     df.loc[evt_mask, '_edf_'] = (len(y) - 2)
547
548     nloc = {'alpha': alpha, 'beta1': beta1, 'beta2': beta2, 'beta3':
beta3, 'beta4': beta4, 'const': 0}
549
550     def f_expret(row):
551         return ((nloc['alpha'] + (nloc['beta1'] * row['mktrf']) + (nloc[
'beta2'] * row['smb']) + (nloc['beta3'] * row['hml']) + (nloc['beta4'] * row['umd']
))
552
553         df.loc[evt_mask, 'expret'] = df[evt_mask].apply(f_expret, axis=1)
554
555         nloc = {'alpha': alpha, 'beta1': beta1, 'beta2': beta2, 'beta3':
beta3, 'beta4': beta4, 'const': 0}
556
557         def f_abret(row):
558             return (row['ret'] - ((nloc['alpha'] + (nloc['beta1'] * row['
mktrf']) + (nloc['beta2'] * row['smb']) + (nloc['beta3'] * row['hml']) + (nloc['
beta4'] * row['umd']))))
559
560         df.loc[evt_mask, 'abret'] = df[evt_mask].apply(f_abret, axis=1)
561
562         nloc = {'const': 0}
563
564         def f_cret(row):
565             tmp = ((row['ret'] * nloc['const']) + (row['ret'] + nloc['const'
]))
566
567             nloc['const'] = tmp
568             return tmp
569
570         df.loc[evt_mask, 'cret'] = df[evt_mask].apply(f_cret, axis=1)
571         df.loc[evt_mask, 'cret_edate'] = nloc['const']
572
573         nloc = {'const': 0}
574
575         def f_cexpret(row):
576             tmp = ((row['expret'] * nloc['const']) + (row['expret'] + nloc['
const'])))
577
578             nloc['const'] = tmp
579             return tmp
580
581         df.loc[evt_mask, 'cexpret'] = df[evt_mask].apply(f_cexpret, axis=1)
582         nloc = {'const': 0}
583
584         def f_car(row):
585             tmp = (row['abret'] + nloc['const'])
586             nloc['const'] = tmp
587             return tmp
588
589         df.loc[evt_mask, 'car'] = df[evt_mask].apply(f_car, axis=1)
590         df.loc[evt_mask, 'car_edate'] = nloc['const']
591
592         nloc = {'const': 0}
593
594         def f_sar(row):
595             tmp = (row['abret'] / np.sqrt(row['var_estp']))
596             nloc['const'] = tmp
597             return tmp
598
599         df.loc[evt_mask, 'sar'] = df[evt_mask].apply(f_sar, axis=1)
600         df.loc[evt_mask, 'sar_edate'] = nloc['const']
601
602         nloc = {'const': 0, 'evtrang': evtrang}
603
604         def f_scar(row):

```

```

597         tmp = (row['car'] / np_sqrt((evtrang * row['var_estp'])))
598         nloc['const'] = tmp
599         return tmp
600     df.loc[evt_mask, 'scar'] = df[evt_mask].apply(f_scar, axis=1)
601     df.loc[evt_mask, 'scar_edate'] = nloc['const']
602
603     nloc = {'const': 0}
604
605     def f_bhar(row):
606         tmp = (row['cret'] - row['cexpret'])
607         nloc['const'] = tmp
608         return tmp
609     df.loc[evt_mask, 'bhar'] = df[evt_mask].apply(f_bhar, axis=1)
610     df.loc[evt_mask, 'bhar_edate'] = nloc['const']
611
612     df.loc[evt_mask, 'pat_scale'] = (_nobs - 2.00) / (_nobs - 4.00)
613     df.loc[evt_mask, 'pat_scale_edate'] = (_nobs - 2.00) / (_nobs -
4.00)
614
615     # Something erroneous was passed
616     else:
617         df['isest'][evt_mask] = -2
618
619     #####
620     # STEP 4 - OUTPUT THE RESULTS #
621     #####
622     df_sta = df[df['isevt'] == 1]
623     levt = df_sta['evtttime'].unique()
624
625     columns = ['evtttime',
626               'car_m',
627               'ret_m',
628               'abret_m',
629               'abret_t',
630               'sar_t',
631               'pat_ar',
632               'cret_edate_m',
633               'car_edate_m',
634               'pat_car_edate_m',
635               'car_edate_t',
636               'scar_edate_t',
637               'bhar_edate_m']
638
639     idxlist = list(levt)
640     df_stats = pd.DataFrame(index=idxlist, columns=columns)
641     df_stats = df_stats.fillna(0.00000000) # with 0s rather than NaNs
642
643     # Event
644     df_stats['evtttime'] = df_sta.groupby(['evtttime'])['evtttime'].unique()
645     # Means
646     df_stats['abret_m'] = df_sta.groupby(['evtttime'])['abret'].mean()
647     df_stats['bhar_edate_m'] = df_sta.groupby(['evtttime'])['bhar_edate'].mean()
648     df_stats['car_edate_m'] = df_sta.groupby(['evtttime'])['car_edate'].mean()
649     df_stats['car_m'] = df_sta.groupby(['evtttime'])['car'].mean()
650     df_stats['cret_edate_m'] = df_sta.groupby(['evtttime'])['cret_edate'].mean()
651     df_stats['pat_scale_m'] = df_sta.groupby(['evtttime'])['pat_scale'].mean()
652     df_stats['pat_car_edate_m'] = 0
653     df_stats['ret_m'] = df_sta.groupby(['evtttime'])['ret'].mean()
654     df_stats['sar_m'] = df_sta.groupby(['evtttime'])['sar'].mean()
655     df_stats['scar_edate_m'] = df_sta.groupby(['evtttime'])['scar_edate'].mean()
656     df_stats['scar_m'] = df_sta.groupby(['evtttime'])['scar'].mean()
657     # Standard deviations
658     df_stats['car_v'] = df_sta.groupby(['evtttime'])['car'].std()
659     df_stats['abret_v'] = df_sta.groupby(['evtttime'])['abret'].std()
660     df_stats['sar_v'] = df_sta.groupby(['evtttime'])['sar'].std()
661     df_stats['pat_scale_v'] = df_sta.groupby(['evtttime'])['pat_scale'].std()
662     df_stats['car_edate_v'] = df_sta.groupby(['evtttime'])['car_edate'].std()
663     df_stats['scar_edate_v'] = df_sta.groupby(['evtttime'])['scar_edate'].std()
664     df_stats['scar_v'] = df_sta.groupby(['evtttime'])['scar'].std()
665     # Counts
666     df_stats['scar_n'] = df_sta.groupby(['evtttime'])['scar'].count()
667     df_stats['scar_edate_n'] = df_sta.groupby(['evtttime'])['scar_edate'].count()
668     df_stats['sar_n'] = df_sta.groupby(['evtttime'])['sar'].count()
669     df_stats['car_n'] = df_sta.groupby(['evtttime'])['car'].count()

```

```

669         df_stats['n'] = df_sta.groupby(['evtttime'])['evtttime'].count()
670         # Sums
671         df_stats['pat_scale_edate_s'] = df_sta.groupby(['evtttime'])['pat_scale_edate'].
sum()
672         df_stats['pat_scale_s'] = df_sta.groupby(['evtttime'])['pat_scale'].sum()
673
674         # T statistics 1
675         def tststat(row, m, v, n):
676             return row[m] / (row[v] / np_sqrt(row[n]))
677
678         df_stats['abret_t'] = df_stats.apply(tststat, axis=1, args=('abret_m', 'abret_v',
'n'))
679         df_stats['sar_t'] = df_stats.apply(tststat, axis=1, args=('sar_m', 'sar_v', 'n'))
680         df_stats['car_edate_t'] = df_stats.apply(tststat, axis=1, args=('car_edate_m', '
car_edate_v', 'n'))
681         df_stats['scar_edate_t'] = df_stats.apply(tststat, axis=1, args=('scar_edate_m', '
scar_edate_v', 'scar_edate_n'))
682
683         # T statistics 2
684         def tststat2(row, m, s, n):
685             return row[m] / (np_sqrt(row[s]) / row[n])
686
687         df_stats['pat_car'] = df_stats.apply(tststat2, axis=1, args=('scar_m', '
pat_scale_s', 'scar_n'))
688         df_stats['pat_car_edate_m'] = df_stats.apply(tststat2, axis=1, args=('scar_edate_m
', 'pat_scale_edate_s', 'scar_edate_n'))
689         df_stats['pat_ar'] = df_stats.apply(tststat2, axis=1, args=('sar_m', 'pat_scale_s'
, 'sar_n'))
690
691         # FILE 2
692         # EVENT WINDOW
693         df_evtw = df.ix[(df['isevt'] == 1), ['permno', 'edate', 'rdate', 'evtttime', 'ret
', 'abret']]
694         df_evtw.sort_values(['permno', 'evtttime'], ascending=[True, True])
695
696         # FILE 1
697         # EVENT DATE
698         maxv = max(levt)
699         df_evtd = df.ix[(df['isevt'] == 1) & (df['evtttime'] == maxv), ['permno', 'edate'
, 'cret', 'car', 'bhar']]
700         df_evtd.sort_values(['permno', 'edate'], ascending=[True, True])
701
702         if output == 'df':
703             retval = {}
704             retval['event_stats'] = df_stats
705             retval['event_window'] = df_evtw
706             retval['event_date'] = df_evtd
707             return retval
708         elif output == 'print':
709             retval = {}
710             print(tabulate(df_evtd.sort_values(['permno', 'edate'], ascending=[True,
True]), headers='keys', tablefmt='psql'))
711             print(tabulate(df_evtw, headers='keys', tablefmt='psql'))
712             print(tabulate(df_stats, headers='keys', tablefmt='psql'))
713             return retval
714         elif output == 'json':
715             retval = {}
716             retval['event_stats'] = df_stats.to_dict(orient='split')
717             retval['event_window'] = df_evtw.to_dict(orient='split')
718             retval['event_date'] = df_evtd.to_dict(orient='split')
719             # Write this to a file
720             with open(os.path.join(self.output_path, 'EventStudy.json'), 'w') as outfile
:
721                 json_dump(retval, outfile, cls=EncoderJson)
722             # Return the output in case they are doing something programmatically
723             return json_dumps(retval, cls=EncoderJson)
724         elif output == 'csv':
725             retval = ''
726             es = StringIO_StringIO()
727             df_stats.to_csv(es)
728             retval += es.getvalue()
729             ew = StringIO_StringIO()
730             df_evtw.to_csv(ew)

```

```

731         retval += "\r"
732         retval += ew.getvalue()
733         ed = StringIO_StringIO()
734         df_evtd.to_csv(ed)
735         retval += ed.getvalue()
736
737         # write this to a file
738         with open(os.path.join(self.output_path, 'EventStudy.csv'), 'w') as outfile:
739             outfile.write(retval)
740
741         # return the output in case they are doing something programmatically
742         return retval
743     elif output == 'xls':
744         retval = {}
745         xlswriter = pd.ExcelWriter(os.path.join(self.output_path, 'EventStudy.xls'))
746         df_stats.to_excel(xlswriter, 'Stats')
747         df_evtw.to_excel(xlswriter, 'Event Window')
748         df_evtd.to_excel(xlswriter, 'Event Date')
749         xlswriter.save()
750         return retval
751     else:
752         pass
753
754
755 #####
756 # Instantiate the class and call the function #
757 #####
758 # Use absolute path: /home/[institution]/[username]/ (e.g. /home/wharton/jwharton/)
759 eventstudy = EventStudy(output_path='/home/[institution]/[username]/wrds-eventstudy/')
760 with open('/home/[institution]/[username]/wrds-eventstudy/evtstudy-sample.json') as
761     data_file:
762     events = json_load(data_file)
763 result = eventstudy.eventstudy(data=events, model='madj', output='xls')

```