Empirical Corporate Finance Project:

Event Study of Lawsuits

Xinyu Liu

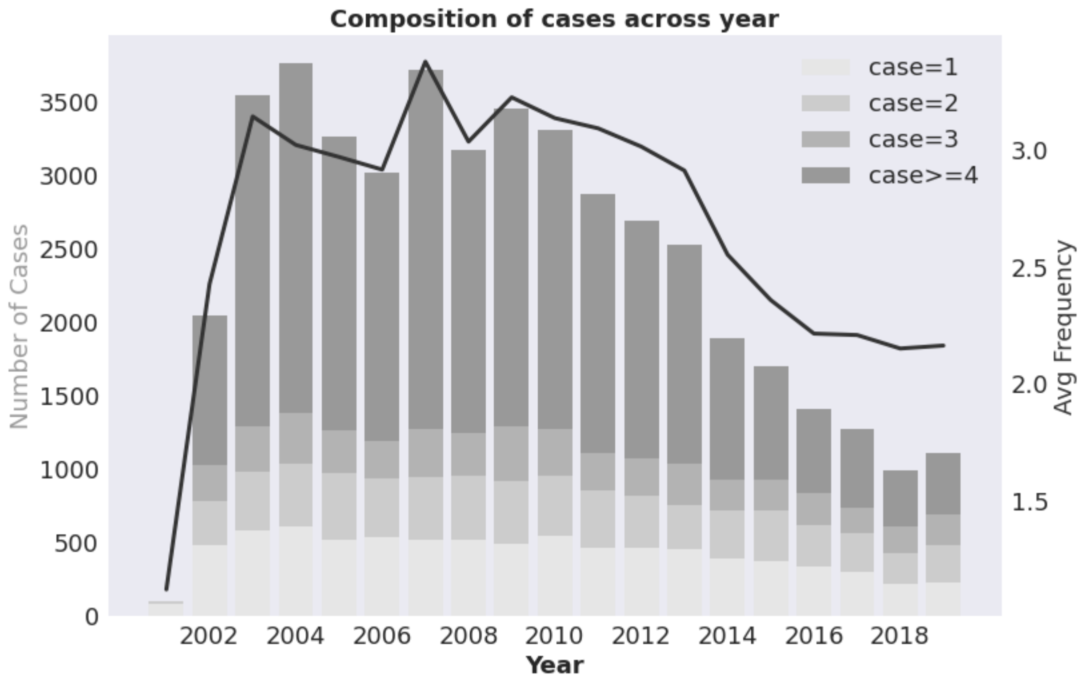
April 17, 2021

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 histgram 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

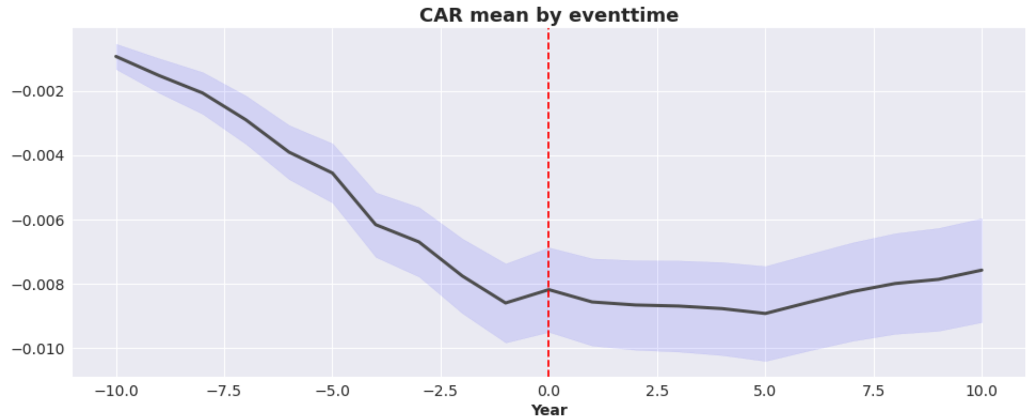


Note: Lawsuit-permno pairs are grouped by year to get the number of cases (left and bar). I calculate the number of unique permno in each year to get the number of companies sued in each year, which is used to calculate the average frequency (right and line) by dividing number of cases with it. In order 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



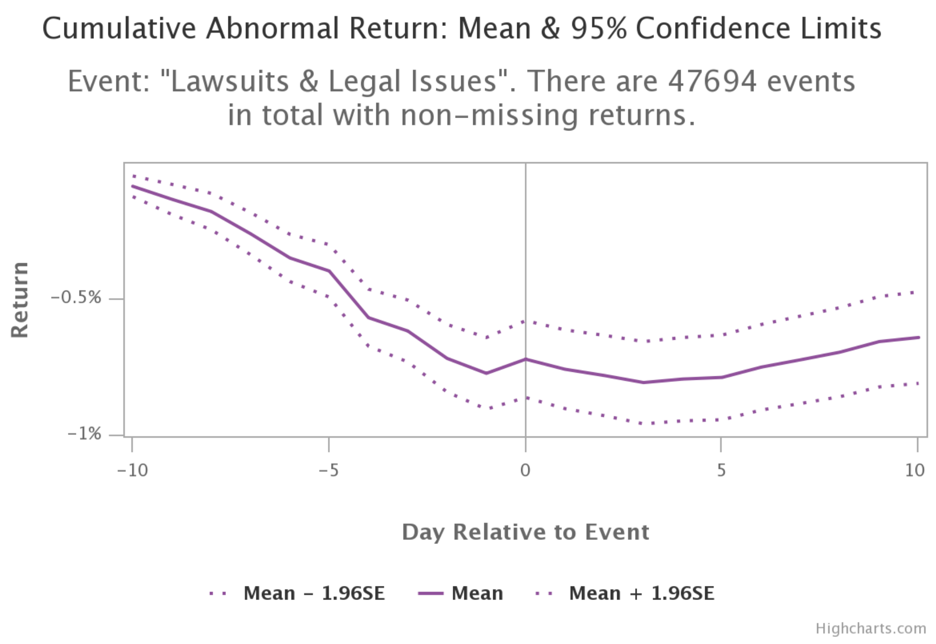
Note: Event window $[-10, 10]$, gap window $[-60, -10]$, estimation window$[-160, -60]$, normal model: adjusted market model.

Appendices

Appendix A

The following results are results generated directly by WRDs.

Figure 3



Appendix B

```python
from datetime import datetime, date
from io import StringIO as StringIO_StringIO
from json import (
    dumps as json_dumps,
    dump as json_dump,
    load as json_load,
    JSONEncoder as json_JSONEncoder,
)
import os

from pandas import (
    DataFrame as pd_DataFrame,
    ExcelWriter as pd_ExcelWriter,
)
from numpy import (
    abs as np_abs,
    nan as np_nan,
    mean as np_mean,
    std as np_std,
    sqrt as np_sqrt,
    ndarray as np_ndarray,
)
from statsmodels.api import (
    OLS as sm_OLS,
    add_constant as sm_add_constant
)
from tabulate import tabulate
import wrds


class EncoderJson(json_JSONEncoder):
    """
    Class used to encodes to JSON data format
    """

    def default(self, obj):
        if isinstance(obj, np_ndarray):
            return obj.tolist()
        elif isinstance(obj, datetime):
            return obj.__str__()
        elif isinstance(obj, date):
            return obj.__str__()

        return json_JSONEncoder.default(self, obj)


class EventStudy(object):
    """
    Main class that runs the event study.
    """

    ###################################################
    #  STEP 0 - AUTHENTICATE AND CONNECT TO POSTGRES  #
    ###################################################

    # parameters when the class is initialized.
    # pass an explicit output path for result file
    def __init__(self, output_path=''):
        if len(output_path) <= 0:
            self.output_path = os.path.expanduser('~')
        else:
            self.output_path = output_path

    # Connect to the Postgres database
    # Code assumes pgpass file has been created
    def connect(self):
        """
        Connect to the Postgres via WRDS.
        """
        self.wrdsconn = wrds.Connection()
```

```python
        self.conn = self.wrdsconn.connect()
        return self.wrdsconn

    # This is the method that gets called to run the event study. The "heavy lifting" happens here.
    def eventstudy(self, data=None, model='m', estwin=100, gap=50, evtwins=-10, evtwine=10, minval=70, output='df'):
        """
            Paramaters passed to the event study method.

            data        =   event data (event date & permno combinations)
            model       =   madj (market-adjusted model)
                            m (market model)
                            ff (fama french)
                            ffm (fama french with momentum factor)
            estwin      =   estimation window
            gap         =   gap between estimation window and event window
            evtwins =   days preceding event date to begin event window
            evtwine =   days after event date to close the event window
            minval      =   minimum number of non-missing return observations (per event) to be regressed on
            output      =   output format of the event study results
                            xls (output an excel file to output path)
                            csv (output a csv file to output path)
                            json (output a json file to output path)
                            df (returns a dictionary of pandas dataframes)
                            print (outputs results to the console - not available via qsub)
        """

        # ##################################################################################
        #  STEP 1 - SET ESTIMATION, EVENT, AND GAP WINDOWS AND GRAB DATA FROM EVENTS FILE  #
        # ##################################################################################

        estwins = (estwin + gap + np_abs(evtwins))  # Estimation window start
        estwine = (gap + np_abs(evtwins) + 1)       # Estimation window end
        evtwinx = (estwins + 1)                     # evt time value (0=event date, -10=window start, 10=window end)
        evtwins = np_abs(evtwins)                   # convert the negative to positive as we will use lag function)
        evtrang = (evtwins + evtwine + 1)           # total event window days (lag + lead + the day itself)

        """
            With the event date as a fixed point, calculate the number of days needed to pass
            to sql lag and lead functions to identify estimation window, gap, and event window.

            evtwins:    event date minus number of preceding days
                        ("event date" - "number of days before event to start [evtwins parameter]")

            evtwine:    event date plus number of following days
                        ("event date" + "number of days after event to end [evtwine parameter]")

            gap:    number of days between the end of the "estimation window"
                    and the beginning of the "event window"

            estwins:     start date of the estimation window
                        ("event date" - "number of days before event to start [evtwins parameter]"
                                      - "number of days in gap [gap parameter]"
                                      - "number of days in estimation window [estwin parameter]")

            evtrang:    entire time range of the event study even from estimate start, through gap,
                        until event window end
```

```
                        (evtwins + evtwine + 1)
        """

        # default the event data in case it was not passed, otherwise read what was passed
        evtdata = [{"edate": "05/29/2012", "permno": "10002"}]
        if data is not None:
            evtdata = json_dumps(data)

        # init values wrapped up to be passed to sql statement
        params = {'estwins': estwins, 'estwine': estwine, 'evtwins': evtwins, 'evtwine': evtwine, 'evtwinx': evtwinx, 'evtdata': evtdata}

        #############################################
        #  STEP 2 - GET RETURNS DATA FROM POSTGRES  #
        #############################################

        # Create a database connection
        wconn = self.connect()

        ##############################################################################
        #  Get the initial data from the database and put it in a pandas dataframe   #
        ##############################################################################

        # create a pandas dataframe that will hold data
        df = wconn.raw_sql("""
        SELECT
                a.*,
                x.*,
                c.date as rdate,
                c.ret as ret1,
                (f.mktrf+f.rf) as mkt,
                f.mktrf,
                f.rf,
                f.smb,
                f.hml,
                f.umd,
                (1+c.ret)*(coalesce(d.dlret,0.00)+1)-1-(f.mktrf+f.rf) as exret,
                (1+c.ret)*(coalesce(d.dlret,0.00)+1)-1 as ret,
                case when c.date between a.estwin1 and a.estwin2 then 1 else 0 end as isest,
                case when c.date between a.evtwin1 and a.evtwin2 then 1 else 0 end as isevt,
                case
                  when c.date between a.evtwin1 and a.evtwin2 then (rank() OVER (PARTITION BY x.evtid ORDER BY c.date)-%(evtwinx)s)
                  else (rank() OVER (PARTITION BY x.evtid ORDER BY c.date))
                end as evttime
        FROM
          (
            SELECT
              date,
              lag(date, %(estwins)s ) over (order by date) as estwin1,
              lag(date, %(estwine)s )  over (order by date) as estwin2,
              lag(date, %(evtwins)s )  over (order by date) as evtwin1,
              lead(date, %(evtwine)s )  over (order by date) as evtwin2
            FROM crsp_a_stock.dsi
          ) as a
        JOIN
        (select
                to_char(x.edate, 'ddMONYYYY') || trim(to_char(x.permno,'999999999')) as evtid,
                x.permno,
                x.edate
        from
        json_to_recordset('%(evtdata)s') as x(edate date, permno int)
        ) as x
          ON a.date=x.edate
        JOIN crsp_a_stock.dsf c
            ON x.permno=c.permno
            AND c.date BETWEEN a.estwin1 and a.evtwin2
        JOIN ff_all.factors_daily f
            ON c.date=f.date
```

```
    LEFT JOIN crsp_a_stock.dsedelist d
        ON x.permno=d.permno
        AND c.date=d.dlstdt
    WHERE f.mktrf is not null
    AND c.ret is not null
    ORDER BY x.evtid, x.permno, a.date, c.date
    """ % params)

    # Columns coming from the database query
    df.columns = ['date', 'estwin1', 'estwin2', 'evtwin1', 'evtwin2',
                  'evtid', 'permno', 'edate', 'rdate', 'ret1', 'mkt',
                  'mktrf', 'rf', 'smb', 'hml', 'umd', 'exret', 'ret',
                  'isest', 'isevt', 'evttime']

    # Additional columns that will hold computed values (post-query)
    addcols = ['RMSE', 'INTERCEPT', 'var_estp', 'expret', 'abret',
               'alpha', '_nobs', '_p_', '_edf_', 'rsq', 'cret',
               'cexpret', 'car', 'scar', 'sar', 'pat_scale', 'bhar',
               'lastevtwin', 'cret_edate', 'scar_edate', 'car_edate',
               'bhar_edate', 'pat_scale_edate', 'xyz']

    # Add them to the dataframe
    for c in addcols:
        if c == 'lastevtwin':
            df[c] = 0
        else:
            df[c] = np_nan

    # ##################################################################################
    #  STEP 3 - FOR EACH EVENT, CALCULATE ABNORMAL RETURN BASED ON CHOSEN RISK MODEL  #
    # ##################################################################################

    # Loop on every category
    for evt in data:

        permno = evt['permno']
        xdate = evt['edate']
        edate = datetime.strptime(xdate, "%m/%d/%Y").date()

        est_mask = (df['permno'] == permno) & (df['edate'] == edate) & (df['isest'] == 1)
        evt_mask = (df['permno'] == permno) & (df['edate'] == edate) & (df['isevt'] == 1)

        #######################################################
        #  Check to see it meets the min obs for est window   #
        #######################################################
        _nobs = df["ret"][est_mask].count()

        # Only carry out the analysis if the number of obsevations meets the minimum threshold
        if _nobs >= minval:

            #######################################################
            #  Regression based on model choices=''               #
            #######################################################

            # Market-Adjusted Model
            if model == 'madj':
                # Set y to the estimation window records
                y = df["exret"][est_mask]

                # Calculate mean and standard deviation of returns for the estimation period
                mean = np_mean(y)
                stdv = np_std(y, ddof=1)

                # Update the columns in the original dataframe (reusing the names from SAS code to help with continuity)
                df.loc[evt_mask, 'INTERCEPT'] = mean
```

```
                df.loc[evt_mask, 'RMSE'] = stdv
                df.loc[evt_mask, '_nobs'] = len(y)
                df.loc[evt_mask, 'var_estp'] = stdv ** 2
                df.loc[evt_mask, 'alpha'] = mean
                df.loc[evt_mask, 'rsq'] = 0
                df.loc[evt_mask, '_p_'] = 1
                df.loc[evt_mask, '_edf_'] = (len(y) - 1)
                df.loc[evt_mask, 'expret'] = df.loc[evt_mask, 'mkt']
                df.loc[evt_mask, 'abret'] = df.loc[evt_mask, 'exret']
                df_est = df[est_mask]
                _nobs = len(df_est[df_est.ret.notnull()])

                nloc = {'const': 0}

                def f_cret(row):
                    tmp = ((row['ret'] * nloc['const']) + (row['ret'] + nloc['const']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'cret'] = df[evt_mask].apply(f_cret, axis=1)
                df.loc[evt_mask, 'cret_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_cexpret(row):
                    tmp = ((row['expret'] * nloc['const']) + (row['expret'] + nloc['const']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'cexpret'] = df[evt_mask].apply(f_cexpret, axis=1)

                nloc = {'const': 0}

                def f_car(row):
                    tmp = (row['abret'] + nloc['const'])
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'car'] = df[evt_mask].apply(f_car, axis=1)
                df.loc[evt_mask, 'car_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_sar(row):
                    tmp = (row['abret'] / np_sqrt(row['var_estp']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'sar'] = df[evt_mask].apply(f_sar, axis=1)
                df.loc[evt_mask, 'sar_edate'] = nloc['const']

                nloc = {'const': 0, 'evtrang': evtrang}

                def f_scar(row):
                    tmp = (row['car'] / np_sqrt((evtrang * row['var_estp'])))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'scar'] = df[evt_mask].apply(f_scar, axis=1)
                df.loc[evt_mask, 'scar_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_bhar(row):
                    tmp = (row['cret'] - row['cexpret'])
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'bhar'] = df[evt_mask].apply(f_bhar, axis=1)
                df.loc[evt_mask, 'bhar_edate'] = nloc['const']

                df.loc[evt_mask, 'pat_scale'] = (_nobs - 2.00) / (_nobs - 4.00)
                df.loc[evt_mask, 'pat_scale_edate'] = (_nobs - 2.00) / (_nobs - 4.00)

            # Market Model
            elif model == 'm':
```

```
                # Set y to the estimation window records
                X = df["mktrf"][est_mask]
                y = df["ret"][est_mask]

                # Fit an OLS model with intercept on mktrf
                X = sm_add_constant(X)
                est = sm_OLS(y, X).fit()

                # Set the variables from the output
                df_est = df[(df['permno'] == permno) & (df['edate'] == edate) & (df['isest'] == 1)]
                _nobs = len(df_est[df_est.ret.notnull()])   # not null observations

                # aggregate variables
                # cret_edate = np_nan
                # scar_edate = np_nan
                # car_edate = np_nan
                # bhar_edate = np_nan
                # pat_scale_edate = np_nan
                alpha = est.params.__getitem__('const')
                beta1 = est.params.__getitem__('mktrf')

                df.loc[evt_mask, 'INTERCEPT'] = alpha
                df.loc[evt_mask, 'alpha'] = alpha
                df.loc[evt_mask, 'RMSE'] = np_sqrt(est.mse_resid)
                df.loc[evt_mask, '_nobs'] = _nobs
                df.loc[evt_mask, 'var_estp'] = est.mse_resid
                df.loc[evt_mask, 'rsq'] = est.rsquared
                df.loc[evt_mask, '_p_'] = 2
                df.loc[evt_mask, '_edf_'] = (len(y) - 2)

                nloc = {'alpha': alpha, 'beta1': beta1, 'const': 0}

                def f_expret(row):
                    return (nloc['alpha'] + (nloc['beta1'] * row['mktrf']))
                df.loc[evt_mask, 'expret'] = df[evt_mask].apply(f_expret, axis=1)

                nloc = {'alpha': alpha, 'beta1': beta1, 'const': 0}

                def f_abret(row):
                    return (row['ret'] - (nloc['alpha'] + (nloc['beta1'] * row['mktrf'])))
                df.loc[evt_mask, 'abret'] = df[evt_mask].apply(f_abret, axis=1)

                nloc = {'const': 0}

                def f_cret(row):
                    tmp = ((row['ret'] * nloc['const']) + (row['ret'] + nloc['const']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'cret'] = df[evt_mask].apply(f_cret, axis=1)
                df.loc[evt_mask, 'cret_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_cexpret(row):
                    tmp = ((row['expret'] * nloc['const']) + (row['expret'] + nloc['const']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'cexpret'] = df[evt_mask].apply(f_cexpret, axis=1)

                nloc = {'const': 0}

                def f_car(row):
                    # nonlocal const
                    tmp = (row['abret'] + nloc['const'])
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'car'] = df[evt_mask].apply(f_car, axis=1)
                df.loc[evt_mask, 'car_edate'] = nloc['const']

```

```
                nloc = {'const': 0}

                def f_sar(row):
                    tmp = (row['abret'] / np_sqrt(row['var_estp']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'sar'] = df[evt_mask].apply(f_sar, axis=1)
                df.loc[evt_mask, 'sar_edate'] = nloc['const']

                nloc = {'const': 0, 'evtrang': evtrang}

                def f_scar(row):
                    tmp = (row['car'] / np_sqrt((evtrang * row['var_estp'])))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'scar'] = df[evt_mask].apply(f_scar, axis=1)
                df.loc[evt_mask, 'scar_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_bhar(row):
                    tmp = (row['cret'] - row['cexpret'])
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'bhar'] = df[evt_mask].apply(f_bhar, axis=1)
                df.loc[evt_mask, 'bhar_edate'] = nloc['const']

                df.loc[evt_mask, 'pat_scale'] = (_nobs - 2.00) / (_nobs - 4.00)
                df.loc[evt_mask, 'pat_scale_edate'] = (_nobs - 2.00) / (_nobs - 4.00)

            # Fama-French Three Factor Model
            elif model == 'ff':
                # Set y to the estimation window records
                df_est = df[(df['permno'] == permno) & (df['edate'] == edate) & (df['isest'] == 1)]
                X = df_est[['smb', 'hml', 'mktrf']]
                y = df_est['ret']

                # Fit an OLS model with intercept on mktrf, smb, hml
                X = sm_add_constant(X)
                est = sm_OLS(y, X).fit()
                # est = smf.ols(formula='ret ~ smb + hml + mktrf', data=df_est).fit()

                alpha = est.params.__getitem__('const')
                beta1 = est.params.__getitem__('mktrf')
                beta2 = est.params.__getitem__('smb')
                beta3 = est.params.__getitem__('hml')

                df.loc[evt_mask, 'INTERCEPT'] = alpha
                df.loc[evt_mask, 'alpha'] = alpha
                df.loc[evt_mask, 'RMSE'] = np_sqrt(est.mse_resid)
                df.loc[evt_mask, '_nobs'] = _nobs
                df.loc[evt_mask, 'var_estp'] = est.mse_resid
                df.loc[evt_mask, 'rsq'] = est.rsquared
                df.loc[evt_mask, '_p_'] = 2
                df.loc[evt_mask, '_edf_'] = (len(y) - 2)

                nloc = {'alpha': alpha, 'beta1': beta1, 'beta2': beta2, 'beta3': beta3, 'const': 0}

                def f_expret(row):
                    return ((nloc['alpha'] + (nloc['beta1'] * row['mktrf']) + (nloc['beta2'] * row['smb']) + (nloc['beta3'] * row['hml'])))
                df.loc[evt_mask, 'expret'] = df[evt_mask].apply(f_expret, axis=1)

                nloc = {'alpha': alpha, 'beta1': beta1, 'beta2': beta2, 'beta3': beta3, 'const': 0}

                def f_abret(row):
                    return (row['ret'] - ((nloc['alpha'] + (nloc['beta1'] * row['mktrf']) + (nloc['beta2'] * row['smb']) + (nloc['beta3'] * row['hml']))))
```

```
                df.loc[evt_mask, 'abret'] = df[evt_mask].apply(f_abret, axis=1)

                nloc = {'const': 0}

                def f_cret(row):
                    tmp = ((row['ret'] * nloc['const']) + (row['ret'] + nloc['const']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'cret'] = df[evt_mask].apply(f_cret, axis=1)
                df.loc[evt_mask, 'cret_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_cexpret(row):
                    tmp = ((row['expret'] * nloc['const']) + (row['expret'] + nloc['const']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'cexpret'] = df[evt_mask].apply(f_cexpret, axis=1)
                nloc = {'const': 0}

                def f_car(row):
                    tmp = (row['abret'] + nloc['const'])
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'car'] = df[evt_mask].apply(f_car, axis=1)
                df.loc[evt_mask, 'car_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_sar(row):
                    tmp = (row['abret'] / np_sqrt(row['var_estp']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'sar'] = df[evt_mask].apply(f_sar, axis=1)
                df.loc[evt_mask, 'sar_edate'] = nloc['const']

                nloc = {'const': 0, 'evtrang': evtrang}

                def f_scar(row):
                    tmp = (row['car'] / np_sqrt((evtrang * row['var_estp'])))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'scar'] = df[evt_mask].apply(f_scar, axis=1)
                df.loc[evt_mask, 'scar_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_bhar(row):
                    tmp = (row['cret'] - row['cexpret'])
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'bhar'] = df[evt_mask].apply(f_bhar, axis=1)
                df.loc[evt_mask, 'bhar_edate'] = nloc['const']

                df.loc[evt_mask, 'pat_scale'] = (_nobs - 2.00) / (_nobs - 4.00)
                df.loc[evt_mask, 'pat_scale_edate'] = (_nobs - 2.00) / (_nobs - 4.00)

            # Fama-French Plus Momentum
            elif model == 'ffm':
                # Set y to the estimation window records
                df_est = df[(df['permno'] == permno) & (df['edate'] == edate) & (df['isest'] == 1)]

                X = df_est[['mktrf', 'smb', 'hml', 'umd']]  # indicator variables
                y = df_est['ret']                          # response variables

                # Fit an OLS (ordinary least squares) model with intercept on mktrf, smb, hml, and umd
                X = sm_add_constant(X)
                est = sm_OLS(y, X).fit()
```

```

                alpha = est.params.__getitem__('const')
                beta1 = est.params.__getitem__('mktrf')
                beta2 = est.params.__getitem__('smb')
                beta3 = est.params.__getitem__('hml')
                beta4 = est.params.__getitem__('umd')

                df.loc[evt_mask, 'INTERCEPT'] = alpha
                df.loc[evt_mask, 'alpha'] = alpha
                df.loc[evt_mask, 'RMSE'] = np_sqrt(est.mse_resid)
                df.loc[evt_mask, '_nobs'] = _nobs
                df.loc[evt_mask, 'var_estp'] = est.mse_resid
                df.loc[evt_mask, 'rsq'] = est.rsquared
                df.loc[evt_mask, '_p_'] = 2
                df.loc[evt_mask, '_edf_'] = (len(y) - 2)

                nloc = {'alpha': alpha, 'beta1': beta1, 'beta2': beta2, 'beta3': beta3, 'beta4': beta4, 'const': 0}

                def f_expret(row):
                    return ((nloc['alpha'] + (nloc['beta1'] * row['mktrf']) + (nloc['beta2'] * row['smb']) + (nloc['beta3'] * row['hml']) + (nloc['beta4'] * row['umd'])))
                df.loc[evt_mask, 'expret'] = df[evt_mask].apply(f_expret, axis=1)

                nloc = {'alpha': alpha, 'beta1': beta1, 'beta2': beta2, 'beta3': beta3, 'beta4': beta4, 'const': 0}

                def f_abret(row):
                    return (row['ret'] - ((nloc['alpha'] + (nloc['beta1'] * row['mktrf']) + (nloc['beta2'] * row['smb']) + (nloc['beta3'] * row['hml']) + (nloc['beta4'] * row['umd']))))
                df.loc[evt_mask, 'abret'] = df[evt_mask].apply(f_abret, axis=1)

                nloc = {'const': 0}

                def f_cret(row):
                    tmp = ((row['ret'] * nloc['const']) + (row['ret'] + nloc['const']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'cret'] = df[evt_mask].apply(f_cret, axis=1)
                df.loc[evt_mask, 'cret_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_cexpret(row):
                    tmp = ((row['expret'] * nloc['const']) + (row['expret'] + nloc['const']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'cexpret'] = df[evt_mask].apply(f_cexpret, axis=1)
                nloc = {'const': 0}

                def f_car(row):
                    tmp = (row['abret'] + nloc['const'])
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'car'] = df[evt_mask].apply(f_car, axis=1)
                df.loc[evt_mask, 'car_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_sar(row):
                    tmp = (row['abret'] / np_sqrt(row['var_estp']))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'sar'] = df[evt_mask].apply(f_sar, axis=1)
                df.loc[evt_mask, 'sar_edate'] = nloc['const']

                nloc = {'const': 0, 'evtrang': evtrang}

                def f_scar(row):
```

```
                    tmp = (row['car'] / np_sqrt((evtrang * row['var_estp'])))
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'scar'] = df[evt_mask].apply(f_scar, axis=1)
                df.loc[evt_mask, 'scar_edate'] = nloc['const']

                nloc = {'const': 0}

                def f_bhar(row):
                    tmp = (row['cret'] - row['cexpret'])
                    nloc['const'] = tmp
                    return tmp
                df.loc[evt_mask, 'bhar'] = df[evt_mask].apply(f_bhar, axis=1)
                df.loc[evt_mask, 'bhar_edate'] = nloc['const']

                df.loc[evt_mask, 'pat_scale'] = (_nobs - 2.00) / (_nobs - 4.00)
                df.loc[evt_mask, 'pat_scale_edate'] = (_nobs - 2.00) / (_nobs -
4.00)
            # Something erroneous was passed
            else:
                df['isest'][evt_mask] = -2

    #################################
    #  STEP 4 - OUTPUT THE RESULTS  #
    #################################
    df_sta = df[df['isevt'] == 1]
    levt = df_sta['evttime'].unique()

    columns = ['evttime',
               'car_m',
               'ret_m',
               'abret_m',
               'abret_t',
               'sar_t',
               'pat_ar',
               'cret_edate_m',
               'car_edate_m',
               'pat_car_edate_m',
               'car_edate_t',
               'scar_edate_t',
               'bhar_edate_m']

    idxlist = list(levt)
    df_stats = pd_DataFrame(index=idxlist, columns=columns)
    df_stats = df_stats.fillna(0.00000000)  # with 0s rather than NaNs

    # Event
    df_stats['evttime'] = df_sta.groupby(['evttime'])['evttime'].unique()
    # Means
    df_stats['abret_m'] = df_sta.groupby(['evttime'])['abret'].mean()
    df_stats['bhar_edate_m'] = df_sta.groupby(['evttime'])['bhar_edate'].mean()
    df_stats['car_edate_m'] = df_sta.groupby(['evttime'])['car_edate'].mean()
    df_stats['car_m'] = df_sta.groupby(['evttime'])['car'].mean()
    df_stats['cret_edate_m'] = df_sta.groupby(['evttime'])['cret_edate'].mean()
    df_stats['pat_scale_m'] = df_sta.groupby(['evttime'])['pat_scale'].mean()
    df_stats['pat_car_edate_mean'] = 0
    df_stats['ret_m'] = df_sta.groupby(['evttime'])['ret'].mean()
    df_stats['sar_m'] = df_sta.groupby(['evttime'])['sar'].mean()
    df_stats['scar_edate_m'] = df_sta.groupby(['evttime'])['scar_edate'].mean()
    df_stats['scar_m'] = df_sta.groupby(['evttime'])['scar'].mean()
    # Standard deviations
    df_stats['car_v'] = df_sta.groupby(['evttime'])['car'].std()
    df_stats['abret_v'] = df_sta.groupby(['evttime'])['abret'].std()
    df_stats['sar_v'] = df_sta.groupby(['evttime'])['sar'].std()
    df_stats['pat_scale_v'] = df_sta.groupby(['evttime'])['pat_scale'].std()
    df_stats['car_edate_v'] = df_sta.groupby(['evttime'])['car_edate'].std()
    df_stats['scar_edate_v'] = df_sta.groupby(['evttime'])['scar_edate'].std()
    df_stats['scar_v'] = df_sta.groupby(['evttime'])['scar'].std()
    # Counts
    df_stats['scar_n'] = df_sta.groupby(['evttime'])['scar'].count()
    df_stats['scar_edate_n'] = df_sta.groupby(['evttime'])['scar_edate'].count()
    df_stats['sar_n'] = df_sta.groupby(['evttime'])['sar'].count()
    df_stats['car_n'] = df_sta.groupby(['evttime'])['car'].count()
```

```
    df_stats['n'] = df_sta.groupby(['evttime'])['evttime'].count()
    # Sums
    df_stats['pat_scale_edate_s'] = df_sta.groupby(['evttime'])['pat_scale_edate'].sum()
    df_stats['pat_scale_s'] = df_sta.groupby(['evttime'])['pat_scale'].sum()

    # T statistics 1
    def tstat(row, m, v, n):
        return row[m] / (row[v] / np_sqrt(row[n]))

    df_stats['abret_t'] = df_stats.apply(tstat, axis=1, args=('abret_m', 'abret_v', 'n'))
    df_stats['sar_t'] = df_stats.apply(tstat, axis=1, args=('sar_m', 'sar_v', 'n'))
    df_stats['car_edate_t'] = df_stats.apply(tstat, axis=1, args=('car_edate_m', 'car_edate_v', 'n'))
    df_stats['scar_edate_t'] = df_stats.apply(tstat, axis=1, args=('scar_edate_m', 'scar_edate_v', 'scar_edate_n'))

    # T statistics 2
    def tstat2(row, m, s, n):
        return row[m] / (np_sqrt(row[s]) / row[n])

    df_stats['pat_car'] = df_stats.apply(tstat2, axis=1, args=('scar_m', 'pat_scale_s', 'scar_n'))
    df_stats['pat_car_edate_m'] = df_stats.apply(tstat2, axis=1, args=('scar_edate_m', 'pat_scale_edate_s', 'scar_edate_n'))
    df_stats['pat_ar'] = df_stats.apply(tstat2, axis=1, args=('sar_m', 'pat_scale_s', 'sar_n'))

    # FILE 2
    # EVENT WINDOW
    df_evtw = df.ix[(df['isevt'] == 1), ['permno', 'edate', 'rdate', 'evttime', 'ret', 'abret']]
    df_evtw.sort_values(['permno', 'evttime'], ascending=[True, True])

    # FILE 1
    # EVENT DATE
    maxv = max(levt)
    df_evtd = df.ix[(df['isevt'] == 1) & (df['evttime'] == maxv), ['permno', 'edate', 'cret', 'car', 'bhar']]
    df_evtd.sort_values(['permno', 'edate'], ascending=[True, True])

    if output == 'df':
        retval = {}
        retval['event_stats'] = df_stats
        retval['event_window'] = df_evtw
        retval['event_date'] = df_evtd
        return retval
    elif output == 'print':
        retval = {}
        print(tabulate(df_evtd.sort_values(['permno', 'edate'], ascending=[True, True]), headers='keys', tablefmt='psql'))
        print(tabulate(df_evtw, headers='keys', tablefmt='psql'))
        print(tabulate(df_stats, headers='keys', tablefmt='psql'))
        return retval
    elif output == 'json':
        retval = {}
        retval['event_stats'] = df_stats.to_dict(orient='split')
        retval['event_window'] = df_evtw.to_dict(orient='split')
        retval['event_date'] = df_evtd.to_dict(orient='split')
        # Write this to a file
        with open(os.path.join(self.output_path, 'EventStudy.json'), 'w') as outfile:
            json_dump(retval, outfile, cls=EncoderJson)
        # Return the output in case they are doing something programmatically
        return json_dumps(retval, cls=EncoderJson)
    elif output == 'csv':
        retval = ''
        es = StringIO_StringIO()
        df_stats.to_csv(es)
        retval += es.getvalue()
        ew = StringIO_StringIO()
        df_evtw.to_csv(ew)
```

```
            retval += "\r"
            retval += ew.getvalue()
            ed = StringIO_StringIO()
            df_evtd.to_csv(ed)
            retval += ed.getvalue()

            # write this to a file
            with open(os.path.join(self.output_path, 'EventStudy.csv'), 'w') as outfile:
                outfile.write(retval)

            # return the output in case they are doing something programmatically
            return retval
        elif output == 'xls':
            retval = {}
            xlswriter = pd_ExcelWriter(os.path.join(self.output_path, 'EventStudy.xls'))
            df_stats.to_excel(xlswriter, 'Stats')
            df_evtw.to_excel(xlswriter, 'Event Window')
            df_evtd.to_excel(xlswriter, 'Event Date')
            xlswriter.save()
            return retval
        else:
            pass


#################################################
#  Instantiate the class and call the function  #
#################################################
# Use absolute path: /home/[institution]/[username]/ (e.g. /home/wharton/jwharton/)
eventstudy = EventStudy(output_path='/home/[institution]/[username]/wrds-eventstudy/')
with open('/home/[institution]/[username]/wrds-eventstudy/evtstudy-sample.json') as
    data_file:
    events = json_load(data_file)
result = eventstudy.eventstudy(data=events, model='madj', output='xls')
```