eventstudy

python package

Documentation

version 0.1a1

Jean-Baptiste Lemaire

March 9, 2020

Contents

1. Introduction

Event Study package is an open-source python project created to facilitate the computation of financial event study analyses.

> **Warning:** This package is in alpha. The API will certainly be modified frequently before the release candidate. However, the logic and the structure of the package is mature enough to be published here.
>
> If you have suggestions to improve and extend this package, contact me through GitHub.

1.1 Install

To install the package run the following command-line in your terminal:

```
$ pip install eventstudy
```

2. Get Started

Through two examples, we will discover how to perform an event study analysis on a single event or on an aggregate of events.

Note: You can use the interactive version of this tutorial to play with the functions yourself.

2.1 Preliminary work

1. Load the `eventstudy` module and its dependencies: `numpy` and `matplotlib`:

```
import eventstudy as es
import numpy as np
import matplotlib.pyplot as plt
```

2. Set the parameters needed for your events: the returns and Fama-French factors (using `es.Single.import_returns()`and `es.Single.import_FamaFrench()`):

```
es.Single.import_returns('returns.csv')
es.Single.import_FamaFrench('famafrench.csv')
```

2.2 Example 1: A single event

As an introductory example, we will compute the event study analysis of the announcement of the first iphone, made by Steve Jobs during MacWorld exhibition, on January 7, 2007.

1. Run the event study, here using the Fama-French 3-factor model:

```
event = es.Single.FamaFrench_3factor(
    security_ticker = 'AAPL',
    event_date = np.datetime64('2013-03-04'),
    event_window = (-2,+10),
    estimation_size = 300,
    buffer_size = 30
)
```

Note: You can easily play with the parameters and adjust the event study analysis to your needs.

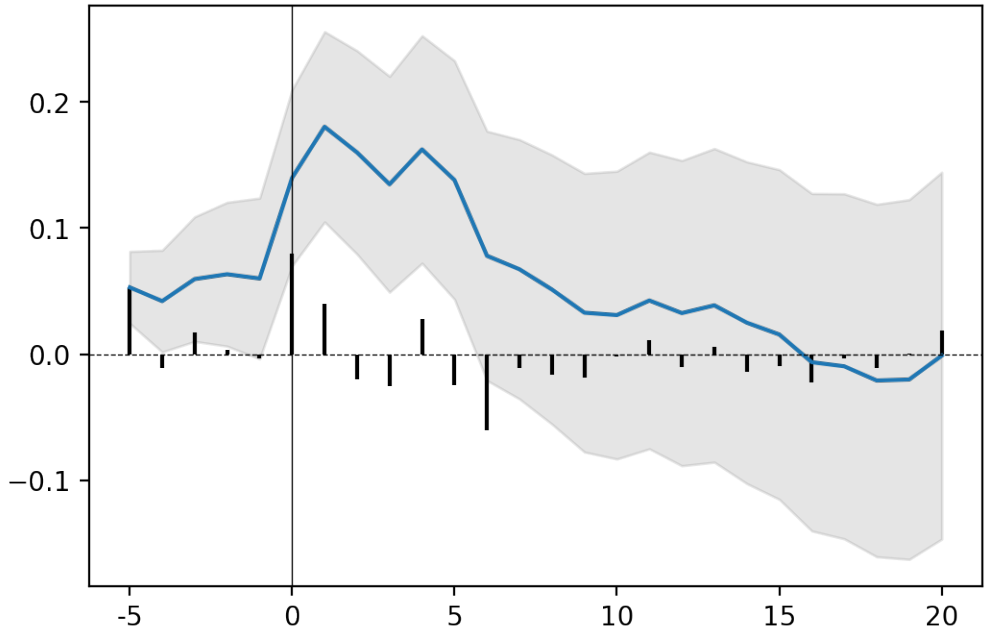
For more details, see the documentation on FamaFrench_3factor function.

See also other models' function. You can even set your own modelisation functions

2. Display results:

   - In a plot:

```
event.plot(AR=True)
plt.show() # use standard matplotlib function to display the plot
```



grey area: confidence interval (here at 90%); blue line: CAAR; black bars: AR

Note: You can remove the confidence interval (set *CI = False*) or change its level of confidence (set *confidence = .95* for a confidence interval at 95%).

By default AR are note displayed (set *AR = True* to add them to the plot).

For more details, see the documentation on plot function .

   - Or in a table:

```
event.results(decimals=[3,5,3,5,2,2])
```

Note: Asterisks are added automatically to highlight the level of significance (Significance level: *** at 99%, ** at 95%, * at 90%). You can remove asterisks by setting *asterisks* parameter at *False*.

decimals is a list of integer setting for each column (except index) the rounding decimal. You can also set one integer (e.g. *decimals = 3*) if you want all columns to be rounded the same.

See the documentation on this results function for more details.

2.3 Example 2: An aggregate of events

The eventstudy package offer three different entry points. The list of events can be provided using a csv file, a plain python text or a python list.

In this second example, based on the same data and preliminary work, we will compute an event study on GAFA annual report (10-K form) releases.

2.3.1 Data preparation

The first step is to gather the data about the events and to format it. For the purpose of this example we will use the csv file methods. The csv file must contains all information needed to compute the model.

The event study will be computed using the market model. According to the (/api/eventstudy.Single.market_model.html#eventstudy.Single.market_model)[API documentation] aside from event study parameters which will be set globally for all events (event_window, estimation_size and buffer_size), the market model needs the security_ticker, the market_ticker and the event_date parameters.

Thus, the csv file should contain three columns with the exact same names as the parameter names.

Note: The csv file must use comma (‘,’) as separator.

2.3.2 Event study computation

To create an event study analysis of an aggregate of events, we will use the `Multiple` class. This class is designed following the same spirit and structure of the `Single` class.

Run the event study using the csv importer method:

```
release_10K = es.Multiple.from_csv(
    path = '10K.csv', # the path to the csv file created
    event_study_model = es.Single.market_model,
    event_window = (-5,+10),
    estimation_size = 200,
    buffer_size = 30,
    date_format = '%d/%m/%Y',
    ignore_errors = True
)
```

Note: The event_study_model parameter must point to a function. This function can either be a model function provided by the *Single* class or a user-defined function.

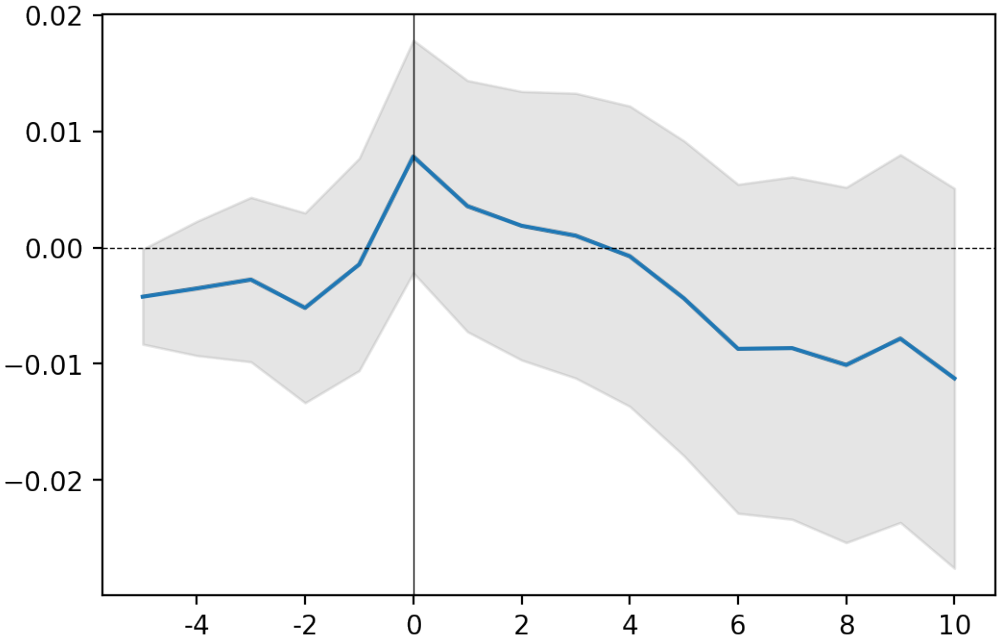
When the ignore_errors parameter is set to *True* (which is its default value), the event study analysis will be computed regardless of any error encounter. Event triggering an error will be remove of the sample. Use error_report method to get the full list of error and their explanation. Errors are often coming from data unavailability.

Set the ignore_errors to False if you prefer the analysis to stop at the first error.

2.3.3 Display results:

- In a plot:

```
release_10K.plot(confidence=.95)
plt.show() # use standard matplotlib function to display the plot
```



grey area: confidence interval (here defined by the user at 95%); blue line: CAAR

- Or in a table:

```
release_10K.results(decimals=[3,5,3,5,2,2])
```

Note: Asterisks are added automatically to highlight the level of significance (Significance level: *** at 99%, ** at 95%, * at 90%). You can remove asterisks by setting *asterisks* parameter at *False*.

decimals is a list of integer setting for each column (except index) the rounding decimal. You can also set one integer (e.g. *decimals = 3*) if you want all columns to be rounded the same.

See the documentation on this results function for more details.

- Get CAR distribution:

```
release_10K.get_CAR_dist(4)
```

2.4 Bonus: Loop in event

The true strengh of this package is to use python. You can create complex algorithms to compute programatically event stuy analyses. From the same data as above, we will compute an event study of 10-K form releases for each company.

2.4.1 Data preparation

This time, we will use the `eventstudy.Multiple.from_list` method which set event list using a python list. This let the user manipulate the list of event before computing event studies.

This list must contains all parameters needed to compute the selected model (here we will use the Fama-French 3-factor model). The Fama-French factors' data has to be set using the `eventstudy.Single.import_FamaFrench` method (see preliminary step).

```
events_db = [
    {'security_ticker': 'AAPL', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2019-10-31')},
    {'security_ticker': 'AAPL', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2018-11-05')},
    {'security_ticker': 'AAPL', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2017-11-03')},
    {'security_ticker': 'AAPL', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2016-10-26')},
    {'security_ticker': 'AAPL', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2015-10-28')},
    {'security_ticker': 'AAPL', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2014-10-27')},
    {'security_ticker': 'AAPL', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2013-10-30')},
    {'security_ticker': 'AAPL', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2012-10-31')},
    {'security_ticker': 'AAPL', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2011-10-26')},
    {'security_ticker': 'AAPL', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2010-10-27')},
    {'security_ticker': 'GOOG', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2020-02-04')},
    {'security_ticker': 'GOOG', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2019-02-05')},
    {'security_ticker': 'GOOG', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2018-02-06')},
    {'security_ticker': 'GOOG', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2017-02-03')},
    {'security_ticker': 'GOOG', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2016-02-11')},
    {'security_ticker': 'FB', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2020-01-30')},
    {'security_ticker': 'FB', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2019-01-31')},
    {'security_ticker': 'FB', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2018-02-01')},
    {'security_ticker': 'FB', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2016-01-28')},
    {'security_ticker': 'FB', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2015-01-29')},
```

(continues on next page)

(continued from previous page)

```
    {'security_ticker': 'FB', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2014-01-31')},
    {'security_ticker': 'AMZN', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2020-01-31')},
    {'security_ticker': 'AMZN', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2019-02-01')},
    {'security_ticker': 'AMZN', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2018-02-02')},
    {'security_ticker': 'AMZN', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2017-02-10')},
    {'security_ticker': 'AMZN', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2016-01-29')},
    {'security_ticker': 'AMZN', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2015-01-30')},
    {'security_ticker': 'AMZN', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2013-01-30')},
    {'security_ticker': 'AMZN', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2012-02-01')},
    {'security_ticker': 'AMZN', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2011-01-28')},
    {'security_ticker': 'AMZN', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2010-01-29')},
    {'security_ticker': 'MSFT', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2019-08-01')},
    {'security_ticker': 'MSFT', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2018-08-03')},
    {'security_ticker': 'MSFT', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2017-08-02')},
    {'security_ticker': 'MSFT', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2016-07-28')},
    {'security_ticker': 'MSFT', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2015-07-31')},
    {'security_ticker': 'MSFT', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2014-07-31')},
    {'security_ticker': 'MSFT', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2013-07-30')},
    {'security_ticker': 'MSFT', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2012-07-26')},
    {'security_ticker': 'MSFT', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2011-07-28')},
    {'security_ticker': 'MSFT', 'market_ticker': 'SPY', 'event_date': np.datetime64(
↪'2010-07-30')}
]
```

2.4.2 Event studies computation

Using a list comprehension, the full list of events can be filtered to create one list per company's ticker. Then we can compute for each company, the event study using the Fama-French 3 factor model and store each event study in a `releases` dictionary.

```
tickers = ['GOOG', 'AAPL', 'FB', 'AMZN', 'MSFT']
releases = dict()

for ticker in tickers:
```

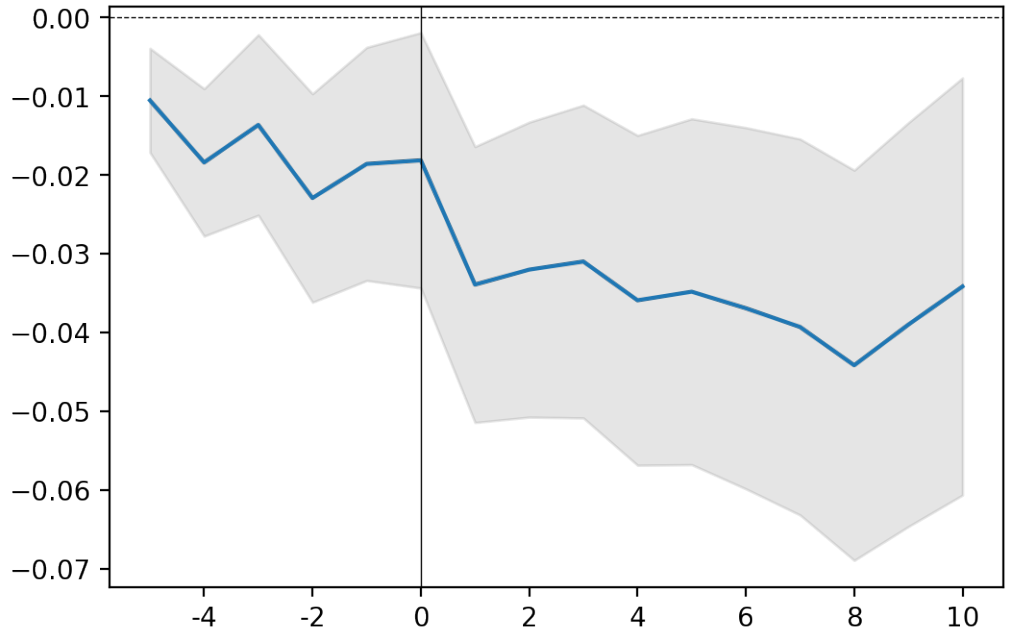
(continues on next page)

(continued from previous page)

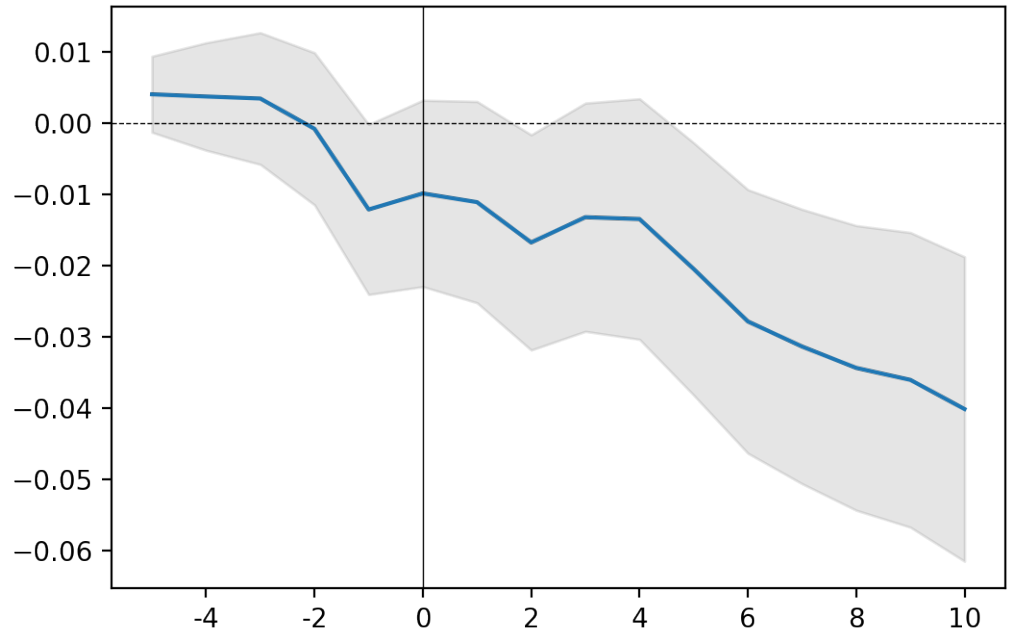
```
    events = [event for event in events_db if event['security_ticker']==ticker]

    releases[ticker] = es.Multiple.from_list(
        events,
        es.Single.FamaFrench_3factor,
        event_window= (-5,+10)
    )
```

2.4.3 Display results

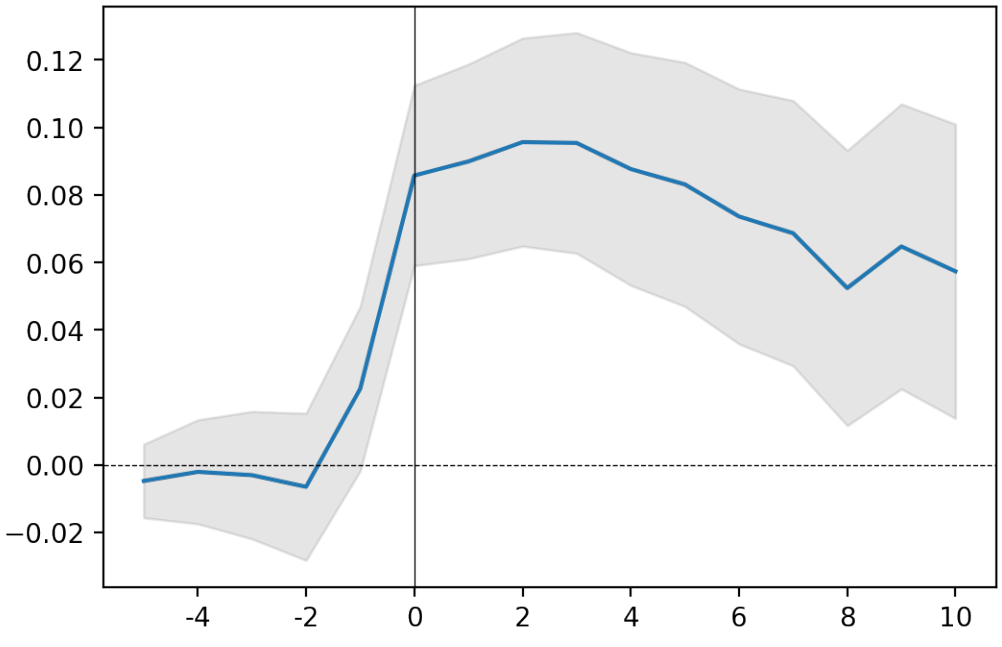
```
for ticker, event in releases.items():
    print(ticker)
    event.plot()
    plt.show()
```



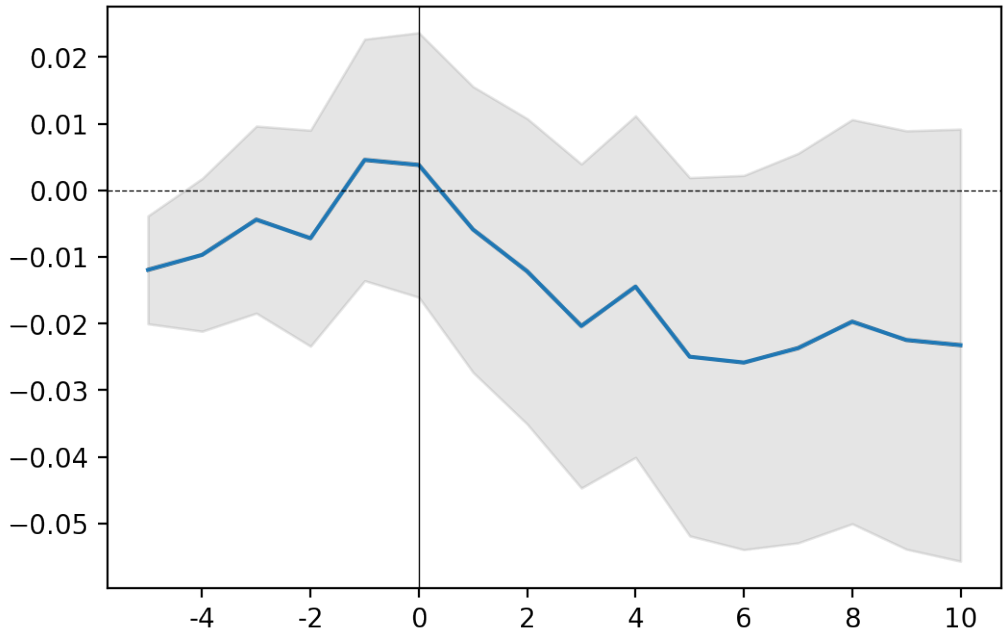
GOOG


0.01
0.00
−0.01
−0.02
−0.03
−0.04
−0.05
−0.06
-4
-2
0
2
4
6
8
10

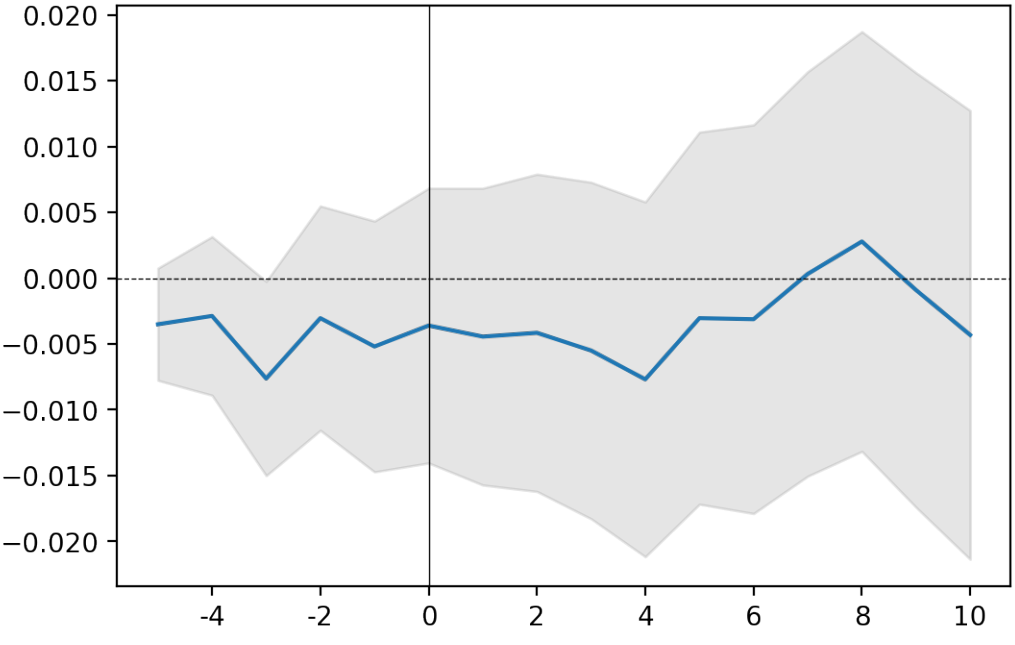
AAPL


0.12
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
-4
-2
0
2
4
6
8
10

FB


0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.04
−0.05
-4
-2
0
2
4
6
8
10

AMZN


0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020
-4
-2
0
2
4
6
8
10

MSFT

3. API

The package implements two classes *Single* and *Multiple* to compute event studies respectively on single events (with measures such as Abnormal Returns (AR) and Cumulative Abnormal Returns (CAR)) and on aggregates of events (with measures such as Average Abnormal Returns (AAR) and Cumulative Abnormal Returns (CAAR)).

The second class (*Multiple*) relies on the first one (*Single*) as it basically performs a loop of single event studies and then aggregates them.

3.1 Single Class

Event Study package's core object. Implement the classical event study methodology[1] for a single event. This implementation heavily relies on the work of MacKinlay[2].

References

- *Run the event study*
- *Import data*
- *Retrieve results*

3.1.1 Run the event study

eventstudy.Single.__init__	Low-level (complex) way of runing an event study.
eventstudy.Single.market_model	Modelise returns with the market model.
eventstudy.Single.constant_mean	Modelise returns with the constant mean model.
eventstudy.Single.FamaFrench_3factor	Modelise returns with the Fama-French 3-factor model.

[1] Fama, E. F., L. Fisher, M. C. Jensen, and R. Roll (1969). "The Adjustment of Stock Prices to New Information". In: International Economic Review 10.1, pp. 1–21.

[2] Mackinlay, A. (1997). "Event Studies in Economics and Finance". In: Journal of Economic Literature 35.1, p. 13.

eventstudy.Single.__init__

Single.__init__(*model_func*, *model_data: dict*, *event_window: tuple = (-10, 10)*, *estimation_size: int = 300*, *buffer_size: int = 30*, *keep_model: bool = False*, *description: str = None*, *event_date: numpy.datetime64 = None*)

Low-level (complex) way of runing an event study. Prefer the simpler use of model methods.

Parameters

- **model_func** – Function computing the modelisation of returns.
- **model_data** (*dict*) – Dictionary containing all parameters needed by *model_func*.
- **event_window** (*tuple, optional*) – Event window specification (T2,T3), by default (-10, +10). A tuple of two integers, representing the start and the end of the event window. Classically, the event-window starts before the event and ends after the event. For example, *event_window = (-2,+20)* means that the event-period starts 2 periods before the event and ends 20 periods after.
- **estimation_size** (*int, optional*) – Size of the estimation for the modelisation of returns [T0,T1], by default 300
- **buffer_size** (*int, optional*) – Size of the buffer window [T1,T2], by default 30
- **keep_model** (*bool, optional*) – If true the model used to compute the event study will be stored in memory. It will be accessible through the class attributes eventstudy.Single.model, by default False

See also:

market_model(), *FamaFrench_3factor()*, *constant_mean()*

Example

Run an event study based on : .. the *market_model* function defined in the *models* submodule, .. given values for security and market returns, .. and default parameters

```
>>> from eventstudy import Single, models
>>> event = Single(
...     models.market_model,
...     {'security_returns':[0.032,-0.043,...], 'market_returns':[0.012,-0.04,...
↪]}
... )
```

eventstudy.Single.market_model

classmethod Single.**market_model**(*security_ticker: str*, *market_ticker: str*, *event_date: numpy.datetime64*, *event_window: tuple = (-10, 10)*, *estimation_size: int = 300*, *buffer_size: int = 30*, *keep_model: bool = False*, ***kwargs*)

Modelise returns with the market model.

Parameters

- **security_ticker** (*str*) – Ticker of the security (e.g. company stock) as given in the returns imported.

- **market_ticker** (*str*) – Ticker of the market (e.g. market index) as given in the returns imported.
- **event_date** (*np.datetime64*) – Date of the event in numpy.datetime64 format.
- **event_window** (*tuple, optional*) – Event window specification (T2,T3), by default (-10, +10). A tuple of two integers, representing the start and the end of the event window. Classically, the event-window starts before the event and ends after the event. For example, *event_window = (-2,+20)* means that the event-period starts 2 periods before the event and ends 20 periods after.
- **estimation_size** (*int, optional*) – Size of the estimation for the modelisation of returns [T0,T1], by default 300
- **buffer_size** (*int, optional*) – Size of the buffer window [T1,T2], by default 30
- **keep_model** (*bool, optional*) – If true the model used to compute the event study will be stored in memory. It will be accessible through the class attributes eventstudy.Single.model, by default False
- ****kwargs** – Additional keywords have no effect but might be accepted to avoid freezing if there are not needed parameters specified.

See also:

FamaFrench_3factor(), *constant_mean()*

Example

Run an event study for the Apple company for the announcement of the first iphone, based on the market model with the S&P500 index as a market proxy.

```
>>> event = EventStudy.market_model(
...     security_ticker = 'AAPL',
...     market_security = 'SPY',
...     event_date = np.datetime64('2007-01-09'),
...     event_window = (-5,+20)
... )
```

eventstudy.Single.constant_mean

classmethod Single.**constant_mean**(*security_ticker*, *event_date: numpy.datetime64*, *event_window: tuple = (-10, 10)*, *estimation_size: int = 300*, *buffer_size: int = 30*, *keep_model: bool = False*, ***kwargs*)

Modelise returns with the constant mean model.

Parameters

- **security_ticker** (*str*) – Ticker of the security (e.g. company stock) as given in the returns imported.
- **event_date** (*np.datetime64*) – Date of the event in numpy.datetime64 format.
- **event_window** (*tuple, optional*) – Event window specification (T2,T3), by default (-10, +10). A tuple of two integers, representing the start and the end of the event window. Classically, the event-window starts before the event and ends after

the event. For example, *event_window = (-2,+20)* means that the event-period starts 2 periods before the event and ends 20 periods after.

- **estimation_size** (*int, optional*) – Size of the estimation for the modelisation of returns [T0,T1], by default 300
- **buffer_size** (*int, optional*) – Size of the buffer window [T1,T2], by default 30
- **keep_model** (*bool, optional*) – If true the model used to compute the event study will be stored in memory. It will be accessible through the class attributes eventstudy.Single.model, by default False
- ****kwargs** – Additional keywords have no effect but might be accepted to avoid freezing if there are not needed parameters specified. For example, if market_ticker is specified.

See also:

market_model(), *Single.FamaFrench_3factor()*

Example

Run an event study for the Apple company for the announcement of the first iphone, based on the constant mean model.

```
>>> event = EventStudy.constant_mean(
...     security_ticker = 'AAPL',
...     event_date = np.datetime64('2007-01-09'),
...     event_window = (-5,+20)
... )
```

eventstudy.Single.FamaFrench_3factor

classmethod Single.**FamaFrench_3factor**(*security_ticker*, *event_date: numpy.datetime64*, *event_window: tuple = (-10, 10)*, *estimation_size: int = 300*, *buffer_size: int = 30*, *keep_model: bool = False*, ***kwargs*)

Modelise returns with the Fama-French 3-factor model. The model used is the one developped in Fama and French (1992)[1].

Parameters

- **security_ticker** (*str*) – Ticker of the security (e.g. company stock) as given in the returns imported.
- **event_date** (*np.datetime64*) – Date of the event in numpy.datetime64 format.
- **event_window** (*tuple, optional*) – Event window specification (T2,T3), by default (-10, +10). A tuple of two integers, representing the start and the end of the event window. Classically, the event-window starts before the event and ends after the event. For example, *event_window = (-2,+20)* means that the event-period starts 2 periods before the event and ends 20 periods after.
- **estimation_size** (*int, optional*) – Size of the estimation for the modelisation of returns [T0,T1], by default 300

[1] Fama, E. F. and K. R. French (1992). "The Cross-Section of Expected Stock Returns". In: The Journal of Finance 47.2, pp. 427–465.

- **buffer_size** (*int, optional*) – Size of the buffer window [T1,T2], by default 30
- **keep_model** (*bool, optional*) – If true the model used to compute the event study will be stored in memory. It will be accessible through the class attributes eventstudy.Single.model, by default False
- ****kwargs** – Additional keywords have no effect but might be accepted to avoid freezing if there are not needed parameters specified. For example, if market_ticker is specified.

See also:

market_model(), *constant_mean()*

Example

Run an event study for the Apple company for the announcement of the first iphone, based on the Fama-French 3-factor model.

```
>>> event = EventStudy.FamaFrench_3factor(
...     security_ticker = 'AAPL',
...     event_date = np.datetime64('2007-01-09'),
...     event_window = (-5,+20)
... )
```

References

3.1.2 Import data

eventstudy.Single.import_FamaFrench	Import Fama-French factors from a csv file to the *Single* Class parameters.
eventstudy.Single.import_returns	Import returns from a csv file to the *Single* Class parameters.
eventstudy.Single.import_returns_from_API	

eventstudy.Single.import_FamaFrench

classmethod Single.**import_FamaFrench**(*path: str*, ***, *rescale_factor: bool = True*, *date_format: str = '%Y%m%d'*)

Import Fama-French factors from a csv file to the *Single* Class parameters. Once imported, the factors are shared among all *Single* instances.

Parameters

- **path** (*str*) – Path to the factors' csv file
- **rescale_factor** (*bool, optional*) – Divide by 100 the factor provided, by default True, Fama-French factors are given in percent on Kenneth R. French website.
- **date_format** (*str, optional*) – Format of the date provided in the csv file, by default "%Y-%m-%d". Refer to datetime standard library for more details date_format: https://docs.python.org/2/library/datetime.html#strftime-strptime-behavior

eventstudy.Single.import_returns

classmethod Single.**import_returns**(*path: str*, ***, *is_price: bool = False*, *log_return: bool = True*, *date_format: str = '%Y-%m-%d'*)

Import returns from a csv file to the *Single* Class parameters. Once imported, the returns are shared among all *Single* instances.

Parameters

- **path** (*str*) – Path to the returns' csv file
- **is_price** (*bool, optional*) – Specify if the file contains price (True) or returns (False), by default False. If set at True, the function will convert prices to returns.
- **log_return** (*bool, optional*) – Specify if returns must be computed as log returns (True) or percentage change (False), by default True. Only used if 'is_price'is set to True.
- **date_format** (*str, optional*) – Format of the date provided in the csv file, by default "%Y-%m-%d". Refer to datetime standard library for more details date_format: https://docs.python.org/2/library/datetime.html#strftime-strptime-behavior

eventstudy.Single.import_returns_from_API

classmethod Single.**import_returns_from_API**()

3.1.3 Retrieve results

eventstudy.Single.plot	Plot the event study result.
eventstudy.Single.results	Return event study's results in a table format.

eventstudy.Single.plot

Single.**plot**(***, *AR=False*, *CI=True*, *confidence=0.9*)

Plot the event study result.

Parameters

- **AR** (*bool, optional*) – Add to the figure a bar plot of AR, by default False
- **CI** (*bool, optional*) – Display the confidence interval, by default True
- **confidence** (*float, optional*) – Set the confidence level, by default 0.90

Returns Plot of CAR and AR (if specified).

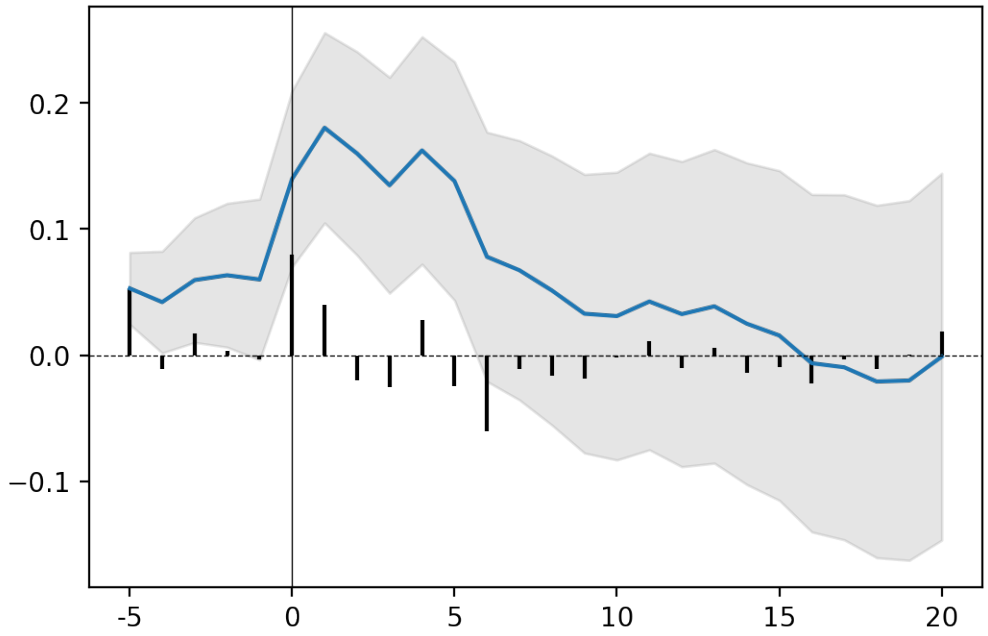
Return type matplotlib.figure

Note: The function return a fully working matplotlib function. You can extend the figure and apply new set-up with matplolib's method (e.g. savefig).

Example

Plot CAR (in blue) and AR (in black), with a confidence interval of 95% (in grey).

```
>>> event = EventStudy.market_model(
...     security_ticker = 'AAPL',
...     market_ticker = 'SPY',
...     event_date = np.datetime64('2007-01-09'),
...     event_window = (-5,+20)
... )
>>> event.plot(AR = True, confidence = .95)
```



eventstudy.Single.results

`Single.`**`results`**`(`*asterisks: bool = True, decimals=3*`)`

Return event study's results in a table format.

Parameters

- **asterisks** (*bool, optional*) – Add asterisks to CAR value based on significance of p-value, by default True
- **decimals** (*int or list, optional*) – Round the value with the number of decimal specified, by default 3. *decimals* can either be an integer, in this case all value will be round at the same decimals, or a list of 6 decimals, in this case each columns will be round based on its respective number of decimal.

Note: When *asterisks* is set as True, CAR's are converted to string type. To make further computation on CARs possible set *asterisks* to False.

Returns AR and AR's variance, CAR and CAR's variance, T-stat and P-value, for each T in the event window.

Return type pandas.DataFrame

Note: The function return a fully working pandas DataFrame. All pandas method can be used on it, especially exporting method (to_csv, to_excel,...)

Example

Get results of a market model event study, with specific number of decimal for each column:

```
>>> event = EventStudy.market_model(
...     security_ticker = 'AAPL',
...     market_ticker = 'SPY',
...     event_date = np.datetime64('2007-01-09'),
...     event_window = (-5,+5)
... )
>>> event.results(decimals = [3,5,3,5,2,2])
```

	AR	Variance AR	CAR	Variance CAR	T-stat	P-value
-5	-0.053	0.00048	-0.053 **	0.00048	-2.42	0.01
-4	0.012	0.00048	-0.041 *	0.00096	-1.33	0.09
-3	-0.013	0.00048	-0.055 *	0.00144	-1.43	0.08
-2	0.004	0.00048	-0.051	0.00192	-1.15	0.13
-1	0	0.00048	-0.051	0.00241	-1.03	0.15
0	-0.077	0.00048	-0.128 **	0.00289	-2.37	0.01
1	-0.039	0.00048	-0.167 ***	0.00337	-2.88	0
2	0.027	0.00048	-0.14 **	0.00385	-2.26	0.01
3	0.024	0.00048	-0.116 **	0.00433	-1.77	0.04
4	-0.024	0.00048	-0.14 **	0.00481	-2.02	0.02
5	0.023	0.00048	-0.117 *	0.00529	-1.61	0.05

Note: Significance level: *** at 99%, ** at 95%, * at 90%

3.2 Multiple Class

Implement computations on an aggregate of event studies. Among which cumulative average abnormal returns (CAAR) and its significance tests. This implementation heavily relies on the work of MacKinlay[1].

Basically, this class takes in input a list of single event studies (*eventstudy.Single*), aggregate them and gives access to aggregate statistics and tests.

Note: All single event studies must have the same specifications (event, estimation and buffer windows). However, the model used for each event study can be different (if needed).

[1] Mackinlay, A. (1997). "Event Studies in Economics and Finance". In: Journal of Economic Literature 35.1, p. 13.

References

- *Run the event study*
- *Import data*
- *Retrieve results*

3.2.1 Run the event study

eventstudy.Multiple.__init__	Low-level (complex) way of runing an aggregate of event studies.
eventstudy.Multiple.from_csv	Compute an aggregate of event studies from a csv file containing each event's parameters.
eventstudy.Multiple.from_list	Compute an aggregate of event studies from a list containing each event's parameters.
eventstudy.Multiple.from_text	Compute an aggregate of event studies from a multi-line string containing each event's parameters.
eventstudy.Multiple.error_report	Return a report of errors faced during the computation of event studies.

eventstudy.Multiple.__init__

Multiple.**__init__**(*sample: list*, *errors=None*)

Low-level (complex) way of runing an aggregate of event studies.

Parameters

- **sample** (*list*) – List containing *eventstudy.Single* objects. You can run independently each eventstudy, aggregate them in a dictionary and compute their aggregate statistics.
- **errors** (*list, optional*) – A list containing errors encountered during the computation of single event studies, by default None.

See also:

from_csv(), *from_list()*, *from_text()*

Example

Run an aggregate of event studies for Apple Inc. 10-K form releases. We loop into a list of dates (in string format). We first convert dates to a numpy.datetie64 format, then run each event study, store them in an *events* list. Finally, we run the aggregate event study.

1. Import packages: »> import numpy as np »> import datetime »> import eventstudy as es

2. import datas and initialize an empty list to store events: »> es.Single.import_returns('returns.csv') »> dates = ['05/11/2018', '03/11/2017', '26/10/2016', ... '28/10/2015', '27/10/2014', '30/10/2013', ... '31/10/2012', '26/10/2011', '27/10/2010'] »> events = list()

3. Run each single event: »> for date in dates: ... formated_date = np.datetime64(... datetime.datetime.strptime(date, '%d/%m/%Y') ...) ... event = es.Single.market_model(... security_ticker = 'AAPL', ... market_ticker = 'SPY', ... event_date = formated_date ...) ... events.append(event)

4. Run the aggregate event study »> agg = es.Multiple(events)

eventstudy.Multiple.from_csv

classmethod Multiple.**from_csv**(*path*, *event_study_model*, *event_window: tuple = (-10, 10)*, *estimation_size: int = 300*, *buffer_size: int = 30*, ***, *date_format: str = '%Y%m%d'*, *keep_model: bool = False*, *ignore_errors: bool = True*)

Compute an aggregate of event studies from a csv file containing each event's parameters.

Parameters

- **path** (*str*) – Path to the csv file containing events' parameters. The first line must contains the name of each parameter needed to compute the event_study_model. All value must be separated by a comma.
- **event_study_model** – Function returning an eventstudy.Single class instance. For example, eventstudy.Single.market_model() (a custom functions can be created).
- **event_window** (*tuple, optional*) – Event window specification (T2,T3), by default (-10, +10). A tuple of two integers, representing the start and the end of the event window. Classically, the event-window starts before the event and ends after the event. For example, *event_window = (-2,+20)* means that the event-period starts 2 periods before the event and ends 20 periods after.
- **estimation_size** (*int, optional*) – Size of the estimation for the modelisation of returns [T0,T1], by default 300
- **buffer_size** (*int, optional*) – Size of the buffer window [T1,T2], by default 30
- **date_format** (*str, optional*) – Format of the date provided in the event_date column, by default "%Y-%m-%d". Refer to datetime standard library for more details date_format: https://docs.python.org/2/library/datetime.html#strftime-strptime-behavior
- **keep_model** (*bool, optional*) – If true the model used to compute each single event study will be stored in memory. They will be accessible through the class attributes eventStudy.Multiple.singles[n].model, by default False
- **ignore_errors** (*bool, optional*) – If true, errors during the computation of single event studies will be ignored. In this case, these events will be removed from the computation. However, a warning message will be displayed after the computation to warn for errors. Errors can also be accessed using *print(eventstudy.Multiple.error_report())*. If false, the computation will be stopped by any error encounter during the computation of single event studies, by default True

See also:

from_text(), *from_list()*

Example

```
>>> agg = eventstudy.Multiple.from_csv(
...     path = 'events.csv',
...     event_study_model = eventstudy.Single.market_model,
...     event_window = (-5,+10),
...     date_format = "%d/%m/%Y"
... )
```

eventstudy.Multiple.from_list

classmethod Multiple.**from_list**(*event_list: list*, *event_study_model*, *event_window: tuple = (-10, 10)*, *estimation_size: int = 300*, *buffer_size: int = 30*, ***, *keep_model: bool = False*, *ignore_errors: bool = True*)

Compute an aggregate of event studies from a list containing each event's parameters.

Parameters

- **event_list** (*list*) – List containing dictionaries specifing each event's parameters (see example for more details).
- **event_study_model** – Function returning an eventstudy.Single class instance. For example, eventstudy.Single.market_model() (a custom functions can be created).
- **event_window** (*tuple, optional*) – Event window specification (T2,T3), by default (-10, +10). A tuple of two integers, representing the start and the end of the event window. Classically, the event-window starts before the event and ends after the event. For example, *event_window = (-2,+20)* means that the event-period starts 2 periods before the event and ends 20 periods after.
- **estimation_size** (*int, optional*) – Size of the estimation for the modelisation of returns [T0,T1], by default 300
- **buffer_size** (*int, optional*) – Size of the buffer window [T1,T2], by default 30
- **keep_model** (*bool, optional*) – If true the model used to compute each single event study will be stored in memory. They will be accessible through the class attributes eventStudy.Multiple.singles[n].model, by default False
- **ignore_errors** (*bool, optional*) – If true, errors during the computation of single event studies will be ignored. In this case, these events will be removed from the computation. However, a warning message will be displayed after the computation to warn for errors. Errors can also be accessed using *print(eventstudy.Multiple.error_report())*. If false, the computation will be stopped by any error encounter during the computation of single event studies, by default True

See also:

from_text(), *from_csv()*

Example

```
>>> list = [
...     {'event_date': np.datetime64("2018-11-05"), 'security_ticker': 'AAPL'},
...     {'event_date': np.datetime64("2017-11-03"), 'security_ticker': 'AAPL'},
...     {'event_date': np.datetime64("2016-10-26"), 'security_ticker': 'AAPL'},
...     {'event_date': np.datetime64("2015-10-28"), 'security_ticker': 'AAPL'},
... ]
>>> agg = eventstudy.Multiple.from_list(
...     text = list,
...     event_study_model = eventstudy.Single.FamaFrench_3factor,
...     event_window = (-5,+10),
... )
```

eventstudy.Multiple.from_text

classmethod Multiple.**from_text**(*text: str*, *event_study_model*, *event_window: tuple = (-10, 10)*, *estimation_size: int = 300*, *buffer_size: int = 30*, ***, *date_format: str = '%Y-%m-%d'*, *keep_model: bool = False*, *ignore_errors: bool = True*)

Compute an aggregate of event studies from a multi-line string containing each event's parameters.

Parameters

- **text** (*str*) – List of events in a multi-line string format. The first line must contains the name of each parameter needed to compute the event_study_model. All value must be separated by a comma (see example for more details).
- **event_study_model** – Function returning an eventstudy.Single class instance. For example, eventstudy.Single.market_model() (a custom functions can be created).
- **event_window** (*tuple, optional*) – Event window specification (T2,T3), by default (-10, +10). A tuple of two integers, representing the start and the end of the event window. Classically, the event-window starts before the event and ends after the event. For example, *event_window = (-2,+20)* means that the event-period starts 2 periods before the event and ends 20 periods after.
- **estimation_size** (*int, optional*) – Size of the estimation for the modelisation of returns [T0,T1], by default 300
- **buffer_size** (*int, optional*) – Size of the buffer window [T1,T2], by default 30
- **date_format** (*str, optional*) – Format of the date provided in the event_date column, by default "%Y-%m-%d". Refer to datetime standard library for more details date_format: https://docs.python.org/2/library/datetime.html#strftime-strptime-behavior
- **keep_model** (*bool, optional*) – If true the model used to compute each single event study will be stored in memory. They will be accessible through the class attributes eventStudy.Multiple.singles[n].model, by default False
- **ignore_errors** (*bool, optional*) – If true, errors during the computation of single event studies will be ignored. In this case, these events will be removed from the computation. However, a warning message will be displayed after the computation to warn for errors. Errors can also be accessed using *print(eventstudy.Multiple.error_report())*. If false, the computation will be stopped by any error encounter during the computation of single event studies, by default True

See also:

from_list(), *from_csv()*

Example

```
>>> text = """security_ticker, market_ticker, event_date
...     AAPL, SPY, 05/11/2018
...     AAPL, SPY, 03/11/2017
...     AAPL, SPY, 26/10/2016
...     AAPL, SPY, 28/10/2015
... """
>>> agg = eventstudy.Multiple.from_text(
...     text = text,
...     event_study_model = eventstudy.Single.market_model,
...     event_window = (-5,+10),
...     date_format = "%d/%m/%Y"
... )
```

eventstudy.Multiple.error_report

Multiple.**error_report**()
Return a report of errors faced during the computation of event studies.

Example

```
>>> agg = eventstudy.Multiple.from_csv(
...     path = 'events.csv',
...     event_study_model = eventstudy.Single.market_model
... )
>>> print(agg.error_report())
```

3.2.2 Import data

Note: Returns and factor data are directly imported at the single event study level.

eventstudy.Single.import_FamaFrench	Import Fama-French factors from a csv file to the *Single* Class parameters.
eventstudy.Single.import_returns	Import returns from a csv file to the *Single* Class parameters.
eventstudy.Single.import_returns_from_API	

3.2.3 Retrieve results

eventstudy.Multiple.plot	Plot the event study result.
eventstudy.Multiple.results	Give event study result in a table format.
eventstudy.Multiple.get_CAR_dist	Give CARs' distribution descriptive statistics in a table format.
eventstudy.Multiple.sign_test	Not implemented yet
eventstudy.Multiple.rank_test	Not implemented yet

eventstudy.Multiple.plot

Multiple.**plot**(*, *AAR=False*, *CI=True*, *confidence=0.9*)

Plot the event study result.

Parameters

- **AAR** (*bool, optional*) – Add to the figure a bar plot of AAR, by default False
- **CI** (*bool, optional*) – Display the confidence interval, by default True
- **confidence** (*float, optional*) – Set the confidence level, by default 0.90

Returns Plot of CAAR and AAR (if specified).

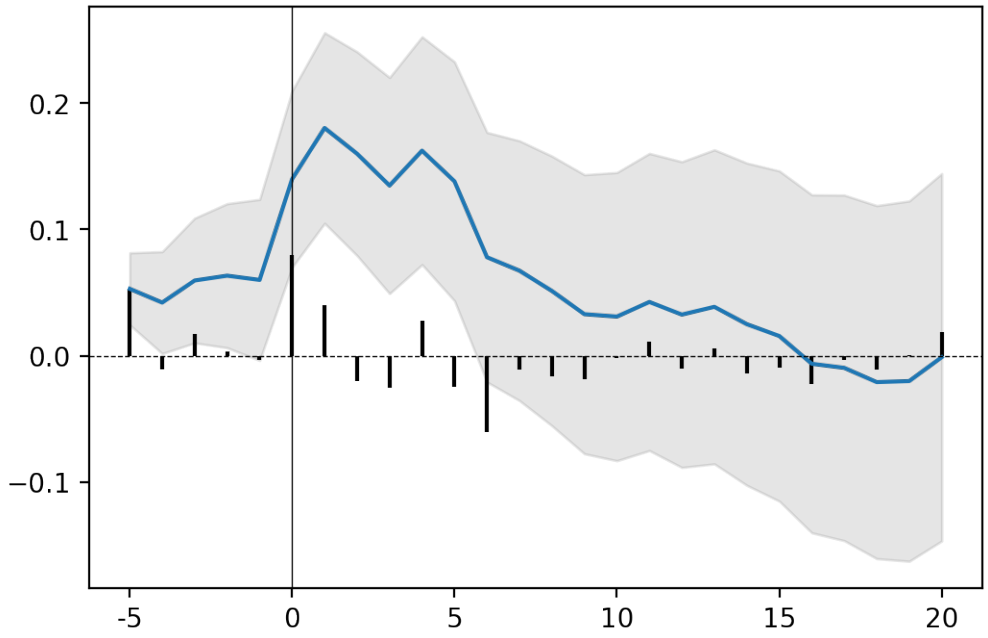
Return type matplotlib.figure

Note: The function return a fully working matplotlib function. You can extend the figure and apply new set-up with matplolib's method (e.g. savefig).

Example

Plot CAR (in blue) and AR (in black), with a confidence interval of 95% (in grey).

```
>>> events = es.Multiple.from_csv(
...     'AAPL_10K.csv',
...     es.Single.FamaFrench_3factor,
...     event_window = (-5,+5),
...     date_format = '%d/%m/%Y'
... )
>>> events.plot(AR = True, confidence = .95)
```



eventstudy.Multiple.results

`Multiple.`**`results`**`(`*asterisks: bool = True*, *decimals=3*`)`

Give event study result in a table format.

Parameters

- **asterisks** (*bool, optional*) – Add asterisks to CAR value based on significance of p-value, by default True
- **decimals** (*int or list, optional*) – Round the value with the number of decimal specified, by default 3. *decimals* can either be an integer, in this case all value will be round at the same decimals, or a list of 6 decimals, in this case each columns will be round based on its respective number of decimal.

Note: When *asterisks* is set as True, CAR's are converted to string type. To make further computation on CARs possible set *asterisks* to False.

Returns AAR and AAR's variance, CAAR and CAAR's variance, T-stat and P-value, for each T in the event window.

Return type pandas.DataFrame

Note: The function return a fully working pandas DataFrame. All pandas method can be used on it, especially exporting method (to_csv, to_excel,...)

Example

Get results of a market model event study on an aggregate of events (Apple Inc. 10-K form releases) imported from a csv, with specific number of decimal for each column:

```
>>> events = es.Multiple.from_csv(
...     'AAPL_10K.csv',
...     es.Single.FamaFrench_3factor,
...     event_window = (-5,+5),
...     date_format = '%d/%m/%Y'
... )
>>> events.results(decimals = [3,5,3,5,2,2])
```

	AAR	Variance AAR	CAAR	Variance CAAR	T-stat	P-value
-5	-0	3e-05	-0.0	3e-05	-0.09	0.47
-4	-0.002	3e-05	-0.003	5e-05	-0.35	0.36
-3	0.009	3e-05	0.007	8e-05	0.79	0.22
-2	0.003	3e-05	0.01	0.0001	1.03	0.15
-1	0.008	3e-05	0.018 *	0.00013	1.61	0.05
0	-0	3e-05	0.018 *	0.00015	1.46	0.07
1	-0.006	3e-05	0.012	0.00018	0.88	0.19
2	0.006	3e-05	0.017	0.0002	1.22	0.11
3	0	3e-05	0.018	0.00023	1.17	0.12
4	-0.007	3e-05	0.011	0.00025	0.69	0.24
5	0.001	3e-05	0.012	0.00028	0.72	0.24

Note: Significance level: *** at 99%, ** at 95%, * at 90%

eventstudy.Multiple.get_CAR_dist

`Multiple.`**`get_CAR_dist`**`(`*decimals=3*`)`

Give CARs' distribution descriptive statistics in a table format.

Parameters **`decimals`** (*`int or list, optional`*) – Round the value with the number of decimal specified, by default 3. *decimals* can either be an integer, in this case all value will be round at the same decimals, or a list of 6 decimals, in this case each columns will be round based on its respective number of decimal.

Returns CARs' descriptive statistics

Return type pandas.DataFrame

Note: The function return a fully working pandas DataFrame. All pandas method can be used on it, especially exporting method (to_csv, to_excel,...)

Example

Get CARs' descriptive statistics of a market model event study on an aggregate of events (Apple Inc. 10-K release) imported from a csv, with specific number of decimal for each column:

```
>>> events = es.Multiple.from_csv(
...     'AAPL_10K.csv',
...     es.Single.FamaFrench_3factor,
...     event_window = (-5,+5),
...     date_format = '%d/%m/%Y'
... )
>>> events.get_CAR_dist(decimals = 4)
```

	Mean	Vari-ance	Kurto-sis	Min	Quantile 25%	Quantile 50%	Quantile 75%	Max
-5	-0	0.001	0.061	-0.052	-0.014	0.001	0.015	0.047
-4	-0.003	0.001	0.247	-0.091	-0.022	0.003	0.015	0.081
-3	0.007	0.002	0.532	-0.082	-0.026	0.006	0.027	0.139
-2	0.01	0.002	-0.025	-0.088	-0.021	0.002	0.033	0.115
-1	0.018	0.003	-0.065	-0.091	-0.012	0.02	0.041	0.138
0	0.018	0.003	-0.724	-0.084	-0.012	0.012	0.057	0.128
1	0.012	0.004	-0.613	-0.076	-0.024	0.003	0.059	0.143
2	0.017	0.005	-0.55	-0.117	-0.026	0.024	0.057	0.156
3	0.018	0.005	0.289	-0.162	-0.032	0.027	0.057	0.17
4	0.011	0.007	2.996	-0.282	-0.039	0.035	0.052	0.178
5	0.012	0.008	1.629	-0.266	-0.05	0.035	0.064	0.174

Note: Significance level: *** at 99%, ** at 95%, * at 90%

eventstudy.Multiple.sign_test

`Multiple.`**`sign_test`**`(`*sign='positive'*, *confidence=0.9*`)`
Not implemented yet

eventstudy.Multiple.rank_test

`Multiple.`**`rank_test`**`(`*confidence*`)`
Not implemented yet

3.3 Excel export

If needed, you can export your results to excel directly using the `excelExporter` module.

3.3.1 Prerequisite

To use the excel export functionalities, you must first install `XlsxWriter` with the following command-line.

```
$ pip install XlsxWriter
```

Then add the following import statement at the beginning of your python script:

```
import eventstudy as es
from eventstudy import excelExporter
```

The last line will attach to the `eventstudy.Single` and `eventstudy.Multiple` classes a new function: `.to_excel()`.

3.3.2 Usage

lorem...

Python Module Index

e

Index