# Project 2: A new version of the event study

## Contents

# 1 Overview

## 1.1 Project Goals

In this project, we will explore an alternative implementation of the event study discussed in class. This version supports multiple tickers and assumes stock and market return data come from "dirty" data files provided by various sources.

In the first part of this project, you will create functions to compute stock and market returns for this new version of the event study. In the second part, you will answer questions regarding the implementation of the code required to compute CARs and t-statistics.

## 1.2 Overview of Part 1: Calculating stock and market returns

Our implementation of the event study involves five steps:

- Step 1: Download the data
- Step 2: Obtain/calculate stock and market returns (to compute CARs)
- Step 3: Select events of interest
- Step 4: Calculate CARs for each event
- Step 5: Calculate t-stats for downgrades and upgrades using the CARs from Step 4.

In the first part of this project, we will discuss how to implement "Step 2" in this updated version of the event study. This version differs from the one discussed in class in two key ways: first, it allows for multiple tickers; second, it utilizes data from different providers.

### 1.2.1 The output data frame

The original version of the event study discussed in class focused on a single ticker, `TSLA`. The result of "Step 2" was a data frame with the following structure:

| date | ret | mkt |
|---|---|---|

Here, `ret` and `mkt` represent returns on the `TSLA` stock and the market, respectively. That data frame was constructed from two CSV files, denoted as `<prc_csv>` and `<mkt_csv>`.

To account for multiple tickers, this version of "Step 2" will produce a data frame with the following structure:

| date | `<tic0>` | ... | `<ticN>` | mkt |
|---|---|---|---|---|

Where the columns `<tic0>`, ..., `<ticN>` include stock returns for the tickers `<TIC0>`, ..., `<TICN>`, respectively. For instance, if we include `AAPL` and `MSFT` in our sample, the output data frame would look like:

| date | aapl | msft | mkt |
|---|---|---|---|

### 1.2.2 Data sources

For the purposes of this project, the relevant data comes from two different providers. We assume that "Step 1: Downloading Data" has already been modified to accommodate these new providers. In the revised Step 1, data from these providers is stored in `.dat` files.

There are two types of `.dat` files, referred to as `<PRICE_DAT>` and `<RET_DAT>`:

- `<PRICE_DAT>` files contain historical **price** and volume data for various tickers, obtained from the first data provider.
- `<RET_DAT>` files contain historical **return** and volume data for various tickers, as well as market returns, obtained from the second data provider.

Unfortunately, the data in these files is often unreliable and improperly formatted, requiring data cleaning before use. For example, column headers in `.dat` files lack a standardized format. We will discuss additional known issues with this data later in this document.

We can represent the structure of these two `.dat` files as follows (the column order may vary):

- `<PRICE_DAT>`:

| <date> | <ticker> | <volume> | ... | <adj_close> |
|---|---|---|---|---|

- `<RET_DAT>`:

| <date> | <ticker> | <volume> | ... | <return> |
|---|---|---|---|---|

Above, `<date>`, `<ticker>`, `<volume>`, `<adj_close>`, and `<return>` represent columns with dates, tickers, volume, adjusted prices, and returns, respectively. The actual headers in both files may vary.

### 1.2.3 Obtaining stock returns

We will assume that data from the first provider is more reliable. Therefore, we will prioritize data from `<PRICE_DAT>` whenever available, using data from `<RET_DAT>` only when necessary.

Specifically, we will calculate returns as follows:

- **If a ticker is present in `<PRICE_DAT>`**: Compute returns using `<adj_close>`, ignoring any data for that ticker in `<RET_DAT>`.
- **If a ticker is absent in `<PRICE_DAT>`**: Use the `<return>` column from `<RET_DAT>`.

This approach excludes **all** information from `<RET_DAT>` for any ticker found in the `<ticker>` column of `<PRICE_DAT>`, regardless of values in `<adj_close>`. Data from `<RET_DAT>` will only be used if a ticker is missing from the `<ticker>` column in `<PRICE_DAT>`.

### 1.2.4 Obtaining market returns

Market returns are only available from the second provider. Assume there is a special ticker, `MKT`, which never appears in any `<PRICE_DAT>` file but is always included in `<RET_DAT>` files. This ensures that market returns can consistently be obtained from `<RET_DAT>`.

### 1.2.5 Summary

Once the data in <PRICE_DAT> and <RET_DAT> have been cleaned, stock and market returns are computed as follows:

1. For each ticker in <PRICE_DAT>, stock returns are calculated using the <adj_close> column.
2. For each ticker in <RET_DAT> that is not found in the <ticker> column of <PRICE_DAT>, stock returns are obtained from the <return> column in <RET_DAT>.
3. Market returns are derived from the <return> column in <RET_DAT> using the special ticker MKT.

### 1.2.6 Example

Let <prc_dat> and <ret_dat> represent data frames created after cleaning and processing all data in the <PRICE_DAT> and <RET_DAT> files, respectively:

- <prc_dat>:

| <date> | <ticker> | ... | <adj_close> |
|---|---|---|---|
| Date(0) | A | ... | P(A, 0) |
| Date(1) | A | ... | P(A, 1) |

- <ret_dat>:

| <date> | <ticker> | ... | <return> |
|---|---|---|---|
| Date(1) | B | ... | Ret(B, 1) |
| Date(1) | MKT | ... | Ret(MKT, 1) |

In this case, the output data frame with stock and market returns will be:

| <date> | <A> | <B> | <MKT> |
|---|---|---|---|
| Date(1) | Ret(A, 1) | Ret(B,1) | Ret(MKT, 1) |

Where:

- Ret(A, 1) = P(A, 1)/P(A, 0) - 1 is computed from <prc_dat>.
- Ret(B, 1), Ret(MKT, 1) represent returns from <ret_dat>.
- <date>, <A>, <B>, and <MKT> are formatted column labels.

## 1.3 Overview of Part 2: Short Answers

In the first part of this project, you implemented the new version of "Step 2". In this part of the project, you will answer questions about the new versions of steps 4 and 5. See the section entitled Part 2: Short Answers for more information.

# 2 Part 1: Completing and submitting your codes

## 2.1 Preparing PyCharm

You should develop your code within PyCharm. Submission, however, will be through Ed. You will need to copy the main.py file from your project into Ed. Unlike the code challenges, Ed will not provide feedback on your code. You can still submit multiple times before the deadline – only your final submission will be marked.

### 2.1.1 The Source Files

All required files are included in a zip archive with the following structure. Please unzip these into your toolkit project folder so it looks like this:

```
toolkit/                        <- PyCharm project folder
|__ project2/                   <- Project 2 files
|   |__ data/
|   |   |__ prc0.dat            <- The`<PRICE_DAT>` example file
|   |   |__ ret0.dat            <- The `<RET_DAT>`  example file
|   |__ __init__.py
|   |__ project2_main.py        <- File to submit
|   |__ verify.py               <- Run before submitting
| ...
|__ toolkit_config.py
```

## 2.2 Completing the `project2_main.py` module

The `project2_main.py` module is the only file you need to complete and submit. Detailed instructions for completing the required functions are provided later in this document. For now, please keep the following in mind:

- Do not import additional modules or create any extra functions (at the module level) in `project2_main.py`; marks will be deducted if you do.
- Before submitting, run the `verify.py` module. This module helps ensure that no unnecessary modules or functions are imported or created.
- Ensure your code is runnable. Marks will be deducted if we cannot import your module for any reason. Please follow instructions carefully and run `verify.py` before submission.
- There are no test functions provided. See the next section for further details.
- The files `prc0.dat` and `ret0.dat` are examples of the `<PRICE_DAT>` and `<RET_DAT>` files referenced in this project. You can use them in your test functions. Keep in mind that your code will be tested with a variety of `.dat` files.
- Modify only the sections marked with the `"<COMPLETE THIS PART>"` tag.

### 2.2.1 Example files and test functions

This project does not include any test functions. If you wish to create test functions, please do so in a *separate module*. You can import the functions defined in `project2_main` into this new module and test them as desired. The files `prc0.dat` and `ret0.dat` can be used as inputs for your test functions.

After completing your code, check the resulting data frame for consistency using your custom test functions. Creating summary statistics may help identify data issues or coding errors.

Remember, do not alter any import statements, function names, or parameter names in the `project2_main.py` module. Crete and use separate modules for testing.

### 2.3 Implementation strategy

The goal of this modified version of "Step 2" is to produce a data frame with stock and market returns obtained from `<PRICE_DAT>` and `<RET_DAT>` files:

| date | <tic0> | ... | <ticN> | mkt |
|---|---|---|---|---|

We will follow the following strategy to implement this task:

1. Create a function called `read_prc_dat` to read a `<PRICE_DAT>` file and produce a properly formatted data frame with the parsed data:

   | <date> | <ticker> | <return> | <volume> |
   |---|---|---|---|

2. Create a function called `read_ret_dat` to read a `<RET_DAT>` file and produce a properly formatted data frame with the parsed data:

   | <date> | <ticker> | <return> | <volume> |
   |---|---|---|---|

   This data frame includes a special ticker, `MKT`, with market returns.

3. Create a function called `mk_ret_df` to produce the output data frame:

   | date | <tic0> | ... | <ticN> | mkt |
   |---|---|---|---|---|

   This function will call the `read_prc_dat` and `read_ret_dat` to create two data frames with the format described above. It will then construct the output data frame by combining information from those two data frames.

The functions `read_prc_dat` and `read_ret_dat` are responsible for reading, cleaning/parsing/formatting the original data from `<PRICE_DAT>` and `<RET_DAT>` files. In both cases, the general strategy is to:

1. Create a data frame with the "raw", unparsed, data from a `.dat` file.
   - Column labels will be as they appear in the `.dat` file.
   - Every element in this data frame will be a `str` instance, with values as they appear in the `.dat` file.
   - We will refer to this data frame as `raw`.
2. Format the column labels in `raw` according to some criteria. The same criteria will be applied to all data frames in this project.

3. Format the elements in `raw` according to some criteria. This includes converting the raw `str` instances to their most appropriate type.

Since many of these tasks apply to both `read_prc_dat` and `read_ret_dat`, we can delegate them to auxiliary functions.

In the next section, we describe the functions we will use to implement the strategy described above.

## 2.4 Completing the `project2_main.py` functions

### 2.4.1 Auxiliary functions

There are two functions which have already been written for you:

- `read_dat(pth: str) -> pd.DataFrame`: This function loads data from a `.dat` file into a data frame. It does not parse or clean the data, nor does it assign specific data types. All entries in the resulting data frame are stored as `str` instances, and all columns have an object `dtype`. This function can be used to load any `.dat` file.
- `str_to_float(value: str) -> float`: This function attempts to convert a string into a float. It returns a float if the conversion is successful and None otherwise.

You will need to complete the body of the following functions (see the docstrings for more information):

- `fmt_col_name(label: str) -> str`: This function formats a column label according to the rules specified in the section Formatting column labels below.
- `fmt_ticker(value: str) -> str`: This function formats a ticker according to the rules specified in the section Formatting tickers below.

### 2.4.2 The `read_prc_dat` and `read_ret_dat` functions

- `read_prc_dat(pth: str) -> pd.DataFrame`: This function produces a data frame with volume and returns from a single `<PRICE_DAT>` file. It takes the location of this file as a single parameter and produces a data frame with the following columns (in any order):

  ```
  Column        dtype
  ------        -----
  <date>        datetime64[ns]
  <ticker>      object
  <return>      float64
  <volume>      float64
  ```

  Where `<date>`, `<ticker>`, `<return>` and `<volume>` are formatted column names representing dates, tickers, stock returns, and volume.

  The original data is unreliable and should be cleaned. See the Cleaning the data section for more information. Column labels and tickers should conform to the format specified by the functions `fmt_col_name` and `fmt_ticker` above.

  Returns should be computed using adjusted closing prices from the original `<PRICE_DAT>` file.

  Assume that there are no gaps in the time series of adjusted closing prices for each ticker.

- `read_ret_dat(pth: str) -> pd.DataFrame`: This function produces a data frame with volume and returns from a single `<RET_DAT>` file. It takes the location of this file as a single parameter and produces a data frame with the following columns (in any order):

  ```
  Column        dtype
  ------        -----
  <date>        datetime64[ns]
  <ticker>      object
  <return>      float64
  <volume>      float64
  ```

  Where `<date>`, `<ticker>`, `<return>` and `<volume>` are formatted column names representing dates, tickers, stock returns, and volume.

  The original data is unreliable and should be cleaned. See the *Cleaning the data* section for more information. Column labels and tickers should conform to the format specified by the functions `fmt_col_name` and `fmt_ticker` above.

### 2.4.3 The `mk_ret_dat` function.

This function has the following signature:

```
def mk_ret_df(
    pth_prc_dat: str,
    pth_ret_dat: str,
    tickers: list[str]) -> pd.DataFrame:
```

- Parameters:
  - `pth_prc_dat`, `pth_ret_dat`: The location of the `<PRICE_DAT>` and `<RET_DAT>` files, respectively.
  - `tickers`: A list with tickers to be included in the output data frame.
- Output: A data frame with a `DatetimeIndex` and the following columns (in any order):

  ```
  Column      dtype
  ------      -----
  <tic0>      float64
  <tic1>      float64
  ...
  <ticN>      float64
  <mkt>       float64
  ```

  Where `<tic0>`, ..., `<ticN>` are formatted column labels with tickers in the list `tickers`, and `<mkt>` is the formatted column label representing market returns.

  Only observations with non-missing market returns should be included.

## 2.5 Cleaning the data

Below are some known issues with the `<PRICE_DAT>` and `<RET_DAT>` files.

- Column headers lack a standardised format. For example, the column with adjusted closing prices in `<PRICE_DAT>` files may be labeled inconsistently as “adj_close,” “Adj close,” or “Adj_close.”

- Numerical data requires cleaning. For example, the number `0.1234` might appear in `.dat` files as either `0.1234` or `"0.1234"` (with quotes). Additionally, typos are common: the number `0` may be mistakenly recorded as the (uppercase) letter `O`, and some price columns in `<PRICE_DAT>` files contain negative numbers that should be interpreted as errors.
- Null values (NaN) are inconsistently represented. For example, the integer `-99` or the float `-99.9` is used instead of an empty string.

There may be other issues with the two files provided to you. Your code should deal with any other data issue you encounter.

### 2.6 Formatting column labels

Assume that original column headers in the `.dat` files meet the following criteria:

- Column names include only alphanumeric characters and underscores.
- White spaces and underscores could be used to separate words in the original column header. Words can be separated by any number of spaces and underscores. For example, both 'Adj Close' or 'Adj Close' could be used to separate the words "Adj" and "Close".
- Column names may include leading or trailing white spaces.

Column names should be formatted according to the following rules:

- Formatted column names should disregard any leading or trailing spaces found in the original column name.
- Words in the formatted column name should be separated by a single underscore, regardless of how they are separated in the original column name.
- Formatted column names should not include uppercase characters.

### 2.7 Formatting tickers

Ticker values should be formatted according to the following rules:

- Formatted ticker values should disregard any leading/trailing spaces or quotes found in the original ticker.
- Formatted tickers consist of uppercase letters only.

NOTE: Tickers also appear as column labels in the data frame produced by `mk_ret_df`. In this case, tickers should be converted to lowercase (i.e., column label formatting rules take precedence).

### 2.8 Example files and test functions

NOTE: The files in the `data` folder provide examples of the types of `<PRICE_DAT>` and `<RET_DAT>` your code needs to handle. Specifically, The `data` folder includes two example files, `prc0.da` and `ret0.dat`. Keep in mind that these files may or may not include the data issues described above.

# 3 Part 2: Short Answers

## 3.1 Overview

The goal of this project is to explore a modified version of the simplified event study. This updated version incorporates data from multiple tickers and different providers.

In the first part of the project, you implemented the new version of "Step 2." We will assume that the updated versions of Steps 1 and 3 are already complete. In this part of the project, you will answer questions about the new versions of Steps 4 and 5.

**Note**: Please refer to the Appendix for a review of the steps required to implement the original version of the event study discussed in class. The Appendix also includes an overview of how the new event study was set up. Below is a summary of the new version of steps 1 to 3.

### 3.1.1 New Step 1: Downloading the data

Output:

- The `.dat` files `<PRICE_DAT>` and `<RET_DAT>`
- A CSV file with recommendations:
  - `<rec csv>`
    - `date_time`
    - `<ticker>` (new column)
    - `firm`
    - `...`
    - `action`

  The only difference between this and the original version of `<rec_csv>` is the addition of the column `<ticker>`.

### 3.1.2 New Step 2: Calculating stock and market returns

This is the step you implemented in the first part of this project.

Input:

- The `.dat` files `<PRICE_DAT>` and `<RET_DAT>`

Output:

- `<ret_df>`: A data frame with structure
  - `date` (`DatetimeIndex`)
  - `<tic0>`
  - `...`
  - `<ticN>`
  - `mkt`

### 3.1.3 New Step 3: Obtain the events of interest

Input:

- The CSV file with recommendations (`<rec_csv>`)

Output:

- `<event_df>`: A data frame with structure:
  - `event_id` (`Index`)
  - `ticker`: Tickers formatted according to the rules above
  - `event_date` (`datetime64[ns]`)
  - `firm`: Firm names (fully parsed)
  - `event_type`: Values are either "upgrade" or "downgrade"

Below is an example of `<event_df>`:

| event_id* | ticker | event_date | firm | event_type |
|---|---|---|---|---|
| 1 | TSLA | 2020-09-23 | DEUTSCHE BANK | upgrade |
| 2 | TSLA | 2020-09-23 | WUNDERLICH | downgrade |
| 3 | TSLA | 2020-11-18 | MORGAN STANLEY | upgrade |
| ... | ... | ... | ... | ... |
| $N-2$ | AAPL | 2023-03-03 | JP MORGAN | downgrade |
| $N-1$ | AAPL | 2023-09-03 | GOLDMAN SACHS | downgrade |
| $N$ | AAPL | 2023-11-08 | MORGAN STANLEY | upgrade |

Implementation:

- Similar to the original Step 2 but applied to multiple tickers. See the Appendix for more details on the original Step 2.

#### 3.1.4 Short answers: Conventions

For the second part of the project, you need to answer two questions. Please use the following convention in your answers (as required):

- `<PRICE_DAT>` and `<RET_DAT>` represent the two types of `.dat` files.
- `<rec_csv>` represents the CSV with recommendations, as described in the "New Step 1" section above.
- `<ret_df>` represents the output of the new version of Step 2.
- `<event_df >` represents the output of the new version of Step 3.
- Use the convention above to describe column labels and indexes. For example:
  - `date` (`DatetimeIndex`)
  - `event_id` (`Index`)
  - `ticker`
  - `event_date`
  - `firm`
  - `event_type`
  - `<tic0>`
  - `...`
  - `<ticN>`
  - `mkt`

## 3.2 Questions

### 3.2.1 Question 1: Sketch the new implementation of "Step 4: Calculate CARs"

Your answer should follow the same format as outlined in the section "Original Step 4" of the Appendix. Feel free to use bullet points to organise your answer. There is no need to provide details on how these tasks should be implemented in Python.

Your answer should include the following information:

- **Input**:
  - Describe the input for this step.
- **Output**:
  - Describe the output for this step.
  - Specify what the columns represent and their data types.
- **Implementation**:
  - List the functions created. For each function, include its name, parameters, and output type.
  - Describe the tasks handled by each function:
    - Use bullet points as in the "Original Step 4" description in the Appendix.
    - Describe the purpose of each task without detailing Python implementation.
  - If a task involves calling or applying another function created in Step 4, specify the function name and how it will be applied. For example, in the "Original Step 4" section in the Appendix, the `calc_car` function is applied to each row of the input DataFrame `event_df`.

### 3.2.2 Question 2: Sketch the new implementation of the new "Step 5: Compute t-stats for upgrades/downgrades"

Please follow the same guidelines as discussed in Question 1.

Your answer should include the following information:

- **Input**:
  - Describe the input for this step.
- **Output**:
  - Describe the output for this step.
  - Specify what the columns represent and their data types.
- **Implementation**:
  - Same as provided in Question 1 above.

# 4 Marking guidelines and checklist

## 4.1 Marking guidelines

The project will be scored out of a total of 100 marks. Partial marks will be given. For the functions, you will be assessed based on whether your function conforms with its docstring **and** the guidelines described in this document.

- Part 1: 80 marks
  - `fmt_col_name`: 15 marks
  - `fmt_ticker`: 5 marks
  - `read_prc_dat`: 20 marks
  - `read_ret_dat`: 20 marks
  - `mk_ret_dat`: 20 marks
- Part 2: 20 marks
  - Question 1: 15 marks
  - Question 2: 5 marks

## 4.2 Checklist

☐ You completed the body of all functions in `project2_main.py`

☐ You ran the module `verify.py`

☐ Your group representative submitted the complete version of your `project2_main.py` module (See the *Submit your code here* slide in ED)

☐ Your group representative answered the short questions in the *Submit your short questions here* slide in ED.

# 5 Appendix

## 5.1 The new version of the event study: Setup

In this new version, the event of interest can be described as follows: On a given date (`event_date`), an analyst working for a given bank (`firm`) downgraded/upgraded a stock (`ticker`). This was the last recommendation on that `event_date` by any analyst working for that `firm` covering that `ticker`.

The setup of this new version of the event study is:

1. Event of interest: The last recommendation by an analyst working for a given `firm` on a given `event_date` which resulted in either the upgrade or downgrade of a given `ticker`.
2. Null hypothesis: Upgrades/Downgrades have no impact on stock prices.
3. Outcome variable: Cumulative abnormal returns (CAR)

- Computed as $CAR = \sum_{\tau=-2}^{\tau=2}(AR_\tau)$
- Under the null, $E[CAR|Downgrade] = 0$ and $E[CAR|Upgrade] = 0$

4. Test the null hypothesis using simple t-tests (separately for upgrades and downgrades)

## 5.2 The original (simplified) version of the event study

In what follows, we summarise the implementation of the simplified version of the event study discussed during Week 10.

### 5.2.1 Original Step 1: Downloading the data

Output: CSV files with structure

- `<prc csv>`:
  - `date`
  - `...`
  - `adj_close`
- `<rec csv>`
  - `date_time`
  - `firm`
  - `...`
  - `action`
- `<mkt csv>`:
  - `date`
  - `...`
  - `mkt`

### 5.2.2 Original Step 2: Calculating stock and market returns

Input:

- The CSV files `<prc_csv>` and `<mkt_csv>`

Output:

- `<ret_df>`: A data frame with structure
  - `date` (`DatetimeIndex`)
  - `ret`
  - `mkt`

  This DF only includes observations with both `ret` and `mkt` information.

### 5.2.3 Original Step 3: Obtain the events of interest

Input:

- The CSV file with recommendations (`<rec_csv>`)

Output:

- `<event_df>`: A data frame with structure:
  - `event_id` (`Index`)
  - `event_date`

  - `firm`
  - `event_type`

Implementation:

1. Read `<rec_csv>`
2. Convert values of `firm` to uppercase
3. Create a new column called `event_date`
4. Keep only the last recommendation for each value of *(firm, event_date)*
5. Create a column called `event_type` with values "upgrade" and "downgrade"
6. Create an index called `event_id`, starting at 1

#### 5.2.4 Original Step 4: Calculate CARs

Input:

- The data frames `<ret_df>` and `<event_df>`

Output:

- `<car_df>`: A data frame with structure:
  - `event_id` (`Index`)
  - `firm`
  - `event_date`
  - `event_type`
  - `car`

Implementation:

1. Create a function to compute CAR for a given event

```
def calc_car(
    ser: pd.Series,
    ret_df: pd.DataFrame) -> float | None:
```

   - Expand the dates in `ser` and create a DF with columns:
     - `firm`
     - `event_date`
     - `event_type`
     - `ret_date` (calendar dates for the event window $[-2, 2]$)
   - Set the index of this DF to `ret_date`
   - Inner join with `ret_df`. Since the indexes of this DF and `ret_df` are calendar dates, only obs with available `ret` and `mkt` are included in the result.
   - If this data frame is empty, return None. Otherwise proceed.
   - Create a series with abnormal returns (the difference between `ret` and `mkt`).
   - Sum the elements of this series and return the resulting float.
2. Create a function called to implement step 4:

```
def step4(
    ret_df: pd.DataFrame,
    event_df: pd.DataFrame,
    ) -> pd.DataFrame:
```

- Apply the `calc_car` function above to each row of `event_df`.
- Add the resulting series to the `event_df` and return a DF with the following structure:
  - `event_id` (`Index`)
  - `firm`
  - `event_date`
  - `event_type`
  - `car`

#### 5.2.5 Original Step 5: Compute t-stats for upgrades/downgrades

Input:

- `<car_df>`: A data frame with structure
  - `event_id` (`Index`)
  - `firm`
  - `event_date`
  - `event_type`
  - `car`

Output:

- Two t-stats

Implementation:

1. Split the `<car_df>` into two groups based on the value of `<event_type>`
2. For each group, compute a t-stat.