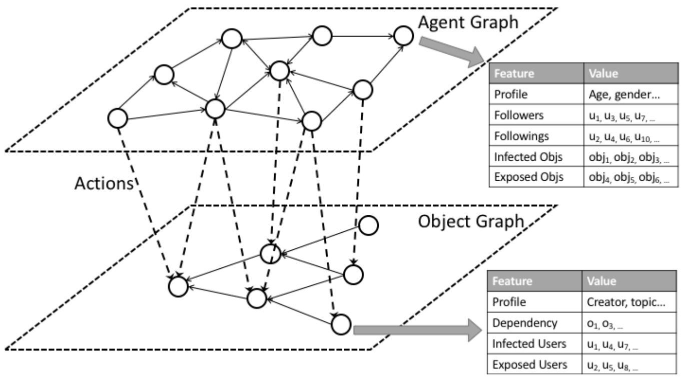
Dependency and Pattern Mining in Github Sequential Events

Yiran Zhao
University of Illinois at Urbana-Champaign
zhao97@illinois.edu

Shengzhong Liu
University of Illinois at Urbana-Champaign
sl29@illinois.edu

***Abstract*—Large volumes of event data are becoming increasingly available in a wide range of applications, such as healthcare analytics, smart cities, social sensing and crowdsourcing, as well as recommendation system construction. These events are usually causally dependent on each other, and present some frequent event patterns in the time series sequence, with respect to who perform the action and what they do. The dependencies and patterns usually reveals the user habits and interactions among different parties. Analyzing the trends, hidden rules, interactions and patterns behind social event data is both interesting and challenging. It possesses significant meaning in user behavior modeling and online marketing strategy design. This paper presents the event models and sequential patterns contained in the Github dataset, which contains millions of users, repositories and even more events generated by users. We analyze the interaction of events that are close in time by modeling them with point process models. Time separated interactions presented as sequential patterns are also discovered.**

***Index Terms*—Github, Point Process, Frequent Pattern Mining, Time Series Analysis.**



Feature	Value
Profile	Age, gender...
Followers	$u_1, u_3, u_5, u_7, \ldots$
Followings	$u_2, u_4, u_6, u_{10}, \ldots$
Infected Objs	$obj_1, obj_2, obj_3, \ldots$
Exposed Objs	$obj_4, obj_5, obj_6, \ldots$

Feature	Value
Profile	Creator, topic...
Dependency	$o_1, o_3, \ldots$
Infected Users	$u_1, u_4, u_7, \ldots$
Exposed Users	$u_2, u_5, u_8, \ldots$

Fig. 1. Unified Representation of Social Networks

I. Introduction

Many social and Internet platforms can be represented as a bipartite graph with different network structures within each side and connecting both sides, such as friend-event relationships from Facebook, colleague-job relationships from LinkedIn and user-repository relationships from Github. We can generally call the human side as agent, and denote the product side as object. The event data on these platforms integrates the agent-agent relationships, object-object dependencies, and agent-object interactions, which is shown in Figure 1. Mining the heterogeneous graph structure inside the networked data can reveal interesting clusters and help the process of prediction and recommendation [1].

In this work, we limit our attention to the agent-object interaction events, while ignore the agent-agent relationships and object-object dependencies. We model the graph in this dataset as a collection of agents and objects, with events as their interactions across the two sides. Agents are users on the Github, and objects are repositories that agents created. Many types of events are observed in this dataset, such as create, delete, push, pull, issue comments, etc. Each event link comes with a tuple of (*time, user, repo, type*), which encodes the information: who did what to which repo at when.

The agent-object-event model can also be applied to a variety of other aforementioned datasets. For example, different colleagues (agents) can join (an event) the same company (object), or users (agents) on YouTube can subscribe (an event) to different channels (objects). Even on Twitter, we can regard each extracted topic as a virtual object that users belonging to the same interest group to perform actions on. Therefore the techniques discussed in this paper is definitely not limited to Github analysis, but can be widely applied to analyze interactions in networked systems in general. We focus on the temporal aspect of analyzing the data, which may be helpful if we want to find the evolution of trends or simulate the whole social platform. We do not rely on any external static information such as user profiles and repo descriptions to identify the two sides of events, and analyze the internal causal relationships among events. We solely utilize the pure sequential event data with aforementioned format to mine the dependencies among different event types, and frequent patterns presented in the event data.

For the technical perspective, we combine two analysis tools in this project: the temporal point process model, and sequential pattern mining methods. Point processes provide the statistical language to describe the timing and properties of events. It can automatically learns the time-sensitive dependencies across different dimensions that are close with each other in time. Temporal point process model has been widely used in daily human behavior modeling, and online information propagation. On the other hand, sequential pattern mining has been widely used in database systems, phrase mining, and spatial-temporal mining applications. Two models are both similar and distinct to each other, with mutually enhancing performance:

- *Temporal point process*: It is a microscopic model that seeks to find causally related dimensions that are close in time, and quantify this causal correlation with a triggering function that decays with time. This model can capture the time varying property of the event dependencies, which means that different time distances represent different extent of dependencies, event if all other attributes(e.g., user, repo) are the same. The point process assumes that, the closer two events are in time, the stronger their correlation is. The limitation is that it can only model the pairwise relationships between two events, but may fail to dependencies with a chain of length three and more, and ignores dependencies that may lie across a long time period.
- *Frequent pattern mining*: Frequent pattern mining methods can find correlation with any length, as long as satisfying the given criteria on support and association rule. But the shortcoming is that it fails to distinguish correlations of event sequences with different time differences. This is also a macroscopic model to some extent, cause it can find the correlations of events cross different periods(manually set as 4 hour in this project). Since events that are too close in time may be concurrent to each other in fact, the frequent pattern mining methods can be good complement and assessment standard of correlations found by the microscopic point process model.

Furthermore, if some event pairs/patterns appear in both microscopic point process model and macroscopic frequent patterns, it will be very likely they are actually correlated to each other. The patterns that appears in only one side and be either identified as long-term/short-term dependencies, or removed as unqualified patterns. This is exactly why we claimed that the two models are mutually enhancing each other. The combination of results from both sides can provide the high-quality patterns/dependencies in the Github event data.

The rest of this paper is organized as follows: In Section II, we briefly review the related literature of two parallel fields; and then we introduce the analysis framework of our sequential pattern mining module, and point process module in Section III. After that, the experiment results based on real world GitHub event data is presented in Section IV. Finally, we conclude this paper in Section V.

II. Related Work

A. Frequent Pattern Mining

Analyzing the occurrence of events can also be thought of as analyzing a time sequence of items. Researchers in data mining community have been developing many powerful algorithms to deal with the humongous human-generated data. [2]–[4] systematically study different approaches to ming frequently occurring patterns in various transaction databases and time-series databases. Since agent-object interactions we study is leaning more towards time aspect of event patterns, sequential pattern mining is more relevant. Sequential pattern mining [5] is an important data mining problem with broad applications, including the analysis of shopping purchase patterns, social interaction patterns, natural disasters, DNA sequences, etc. In [6] an efficient method called PSP is proposed, which offers ordered growth and reduced projected databases. PrefixSpan [5] further improves the efficiency of creating projected database with a pseudo-projection technique. Beside data that is already organized in series of item sets such as transactions, [7] transforms the bug finding problem in sensor networks into pattern mining problems. In the bug finding problem, the authors divide the event logs into good ones and bad ones, and then adopt discriminative frequent pattern mining algorithm to look for patterns that exist with very different frequencies in the two categories.

In our Github sequential pattern mining work, we look at the time series of user-repository interaction data and empirically divide the time space into chunks of “item-sets”. Then we run PrefixSpan [5] to find the frequent sequential “item-sets”. We consider two types of patterns, one with only events and the other with user-event tuples. We also modify the algorithm to account for highly skewed events in the Github dataset and discover patterns with items that are relatively rare.

B. Temporal Point Process

On the other hand, there are also a lot of works using temporal point process, especially Hawkes processes [8] to model discrete, inter-dependent events over continuous time. It ranges from finance field [9](e.g., buy/sell transaction in stock market), geophysics analysis [10](e.g., the triggering effect on the consecutive earthquakes), to the information propagation on the online social media(e.g., one user sharing a Tweet can lead to more retweets by his followers) [11]–[14]. In [11], Zhao et. al. model the propagation of tweet as a Hawkes point process with an exponential triggering kernel and decay function. Du et. al. [12] extend this work to use point process to model the event history of a specific user with multiple event types as a marked point process. They choose the event types as the marks of points. As for the kernels, they extend the parametric function to a recurrent neural network to capture the complex inter-dependent relationship among events. Following these two works, Kurashima et. al. [15] not only model the inter-dependency of user behavior events, but also include the time-varying and periodic properties in long-time range. They solve it by a EM algorithm based maximum likelihood estimation approach.

With temporal point process modeled event sequences, we can not only quantify the frequent event patterns in short term by a triggering matrix, but also touch the time-sensitive dependency. It means that event pairs with same meta information, but different differences will present different triggering intensity in this model. This exactly complement what we lack in the frequent pattern mining approaches, since all the event pairs with same support are regarded the same, while the microscopic time difference is ignored. On the other hand, the point process can not mine the event patterns with relatively long time range, while the frequent pattern mining techniques could. Therefore, in this project, we combine these two methods well to efficiently mine the high-quality event patterns in GitHub user-repository interaction events.

III. Analysis Framework

A. Sequential pattern mining

We develop our user-events sequential pattern mining method based on PrefixSpan [5]. Before even doing the mining part, we need to address the important preprocessing part, without which the mining process can take forever or yield uninteresting results.

1) Repository selection:

There are millions of repositories in just on month data. We use the Github data of January 2015. In order to find interesting patterns, we select repositories based on the following criteria.

1) The number of events occurred in the repository should be larger than a threshold, or otherwise the repository is not active enough to yield interesting patterns.
2) The number of users active in the repository should be larger than a threshold, or otherwise the repository does not contain interesting patterns between multiple users.
3) The distribution of contributions from each user should be as even as possible. If a repository is dominantly controlled by just one user, it may contain less interesting patterns.
4) The number of event types should be larger than a threshold, or otherwise the events in the pattern could just be frequent pushing events.
5) The number of times each event type occurs should be as even as possible. Similar to the previous reason, we want to see the interaction of different event types in the final results.

Based on such principles, we first describe the process of selecting good repositories. First we gather statistics from around 1 million repositories in the dataset, and look at the event number that occurred in that month.

From Figure 2 we can see that most repositories do not contain many events, we choose to filter out about 95% of them with log of events number less than 3.

Next we look at the number of users in each repository. Figure 3 shows that most repositories contains less than 4 users, which is around 1 if you take natural log as in the figure. Therefore, we filter out repositories with user number less than 4.

Then the total number of types of events in each repository is collected and we plot the distribution in Figure 4. Event types include PushEvent, CreateEvent, IssueComment, Pull-Request, ForkEvent, WatchEvent, IssuesEvent, DeleteEvent, ReleaseEvent, MemberEvent, PublicEvent, etc. If a repository has only a few types, it could just contain CreateEvent, PushEvent and nothing more interesting. Therefore we filter out again about 95% of the repositories by setting the threshold at 5.

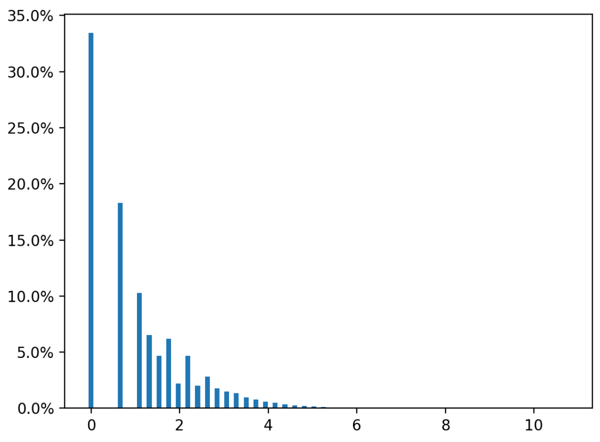


Fig. 2. Distribution of the number of events (log scale).

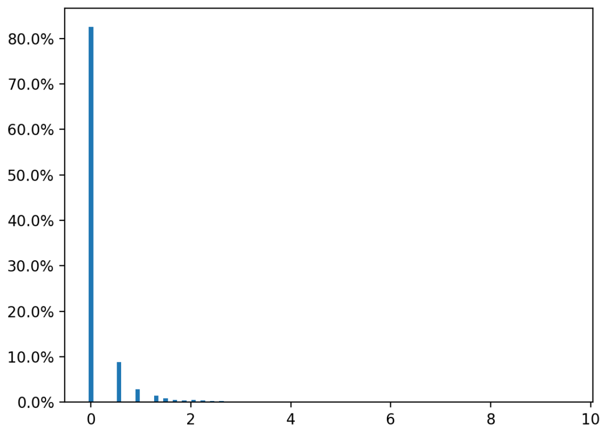


Fig. 3. Distribution of the number of users (log scale).

Next we want to find good repositories that each type of events occur as evenly as possible. We compare the distribution of event types in each repository to the uniform distribution and find the KL divergence between the two distributions. The smaller the KL divergence, the more even the types are.

Figure 5 shows the distribution of KL divergence. We empirically set the threshold for this criteria to be 0.3 which is about where the peak in the histogram is.

Finally we want to find repositories that each user contribute as evenly as possible. If the repository is only controlled by just one or two dominant contributor, the patterns may be not so diverse. We compare the distribution of user events in each repository to the uniform distribution and find the KL divergence between the two distributions. The smaller the KL divergence the more interactions among users there will be.

Figure 6 shows the distribution of KL divergence. Note that we limited the KL divergence to be less than 2 in the plot so

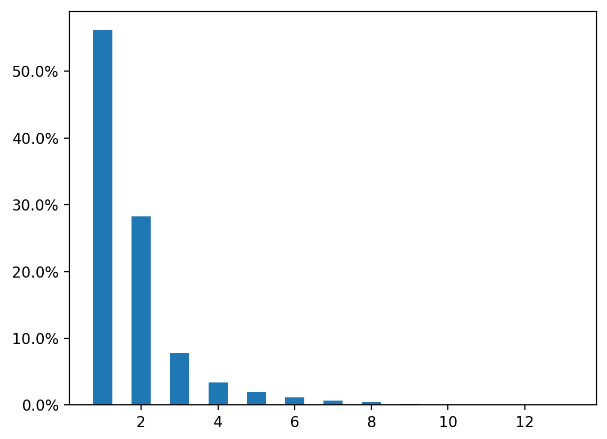


Fig. 4. Distribution of the number of event types.

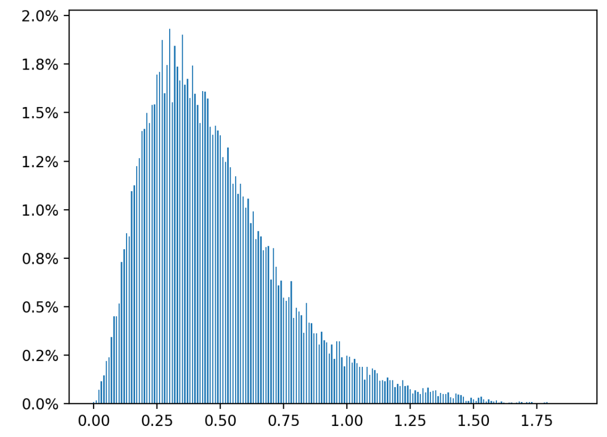


Fig. 5. Distribution of KL divergence of event types.

there is a high vertical line at the right side of the figure. We empirically set the threshold for this criteria to be 0.4 which is about where the peak in the histogram is.

After the initial filtering, we have 1439 repositories to mine, which is significantly smaller than the original number of repositories.

2) Sequential pattern mining:

To find interesting patterns, we need to derive "item-sets" as in purchase records, as well as items that are relatively rare but contribute just as much. We define a group of events that happen close in time space without big time gaps between them as a "transaction", which is also called an "item-set". To find such groups in the time series of events, we set a threshold on the time gap so that we cut the time axis if the time gap is greater than the threshold.

We first look at the proper value of time gap to cut the time axis into segments of item-sets. We plot the distribution of adjacent events and their time between and plot the histogram in bins of 2 hours in Figure 7. If we set the time threshold too small, we end up with a lot of sequential item sets and relatively few items in each item set. If we set the threshold too large, then we cut too infrequently and group too many events in one item set. Therefore, we choose the time gap to be 4 hours (240 min) since it is right larger than most time gaps while not that large to cause insufficient grouping.

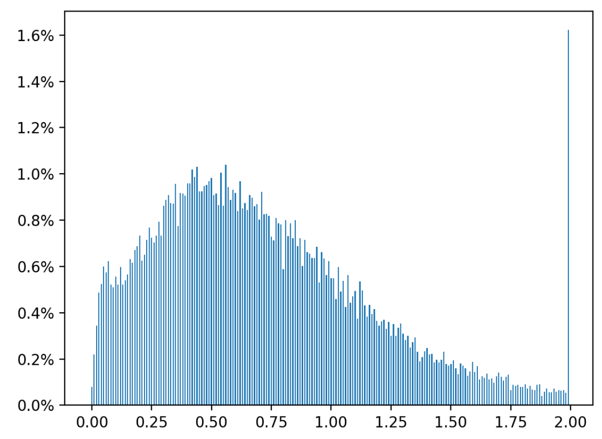


Fig. 6. Distribution of KL divergence of user contributions.

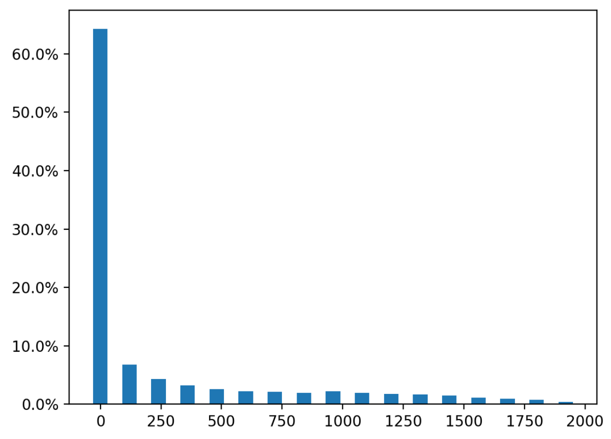


Fig. 7. Distribution of time gaps (minutes).

After grouping events into time-ordered groups (item sets), we adopt the PrefixSpan sequential pattern mining algorithm to discover interesting interactions between users in the same repository. However, specifying the same minimum support for all types of events may be suboptimal since many events happen very few times. We then modify the algorithm to make the min-support smaller for less frequent event types such as delete and release events.

In order to make discount on minimum support for rare event types, we first find the number of times each event happened in the dataset. Table I summarizes the times each event occurred. We can clearly see that in order to discover patterns involving rare events such as "MemberEvent" or "ReleaseEvent", we need to lower the support threshold for sequences that contains these events.

The number of item set sequence to mind is about 1400, and we set the min-support percentage for event patterns to be 50%, and set the min-support for patterns that contain rare events (< 2000 times) to be 2%. To mine the user-event tuple patterns, we set the min-support percentage for event patterns to be 12.5%, and set the min-support for rare events to be 0.5%. We find the number and quality of results to be good under these settings.

TABLE I
EVENT TYPES AND OCCURRED TIMES.

Type	Number
PushEvent	8288
CreateEvent	3039
IssueCommentEvent	10610
PullRequestReviewCommentEvent	2224
PullRequestEvent	7730
ForkEvent	5380
WatchEvent	7935
IssuesEvent	4317
DeleteEvent	1615
CommitCommentEvent	671
GollumEvent	103
ReleaseEvent	76
MemberEvent	237
PublicEvent	7

To modify the PrefixSpan algorithm, we dynamically change the minimum support threshold in the code upon detecting a candidate sequence that contains a rare event. In this way, interesting patterns with few appearance can also be discovered.

To find good patterns, we choose to mine maximal frequent sequence, which means if we add one more item to the sequence, it will not become frequent any more (support falls below the threshold). Maximal patterns can deliver more meaningful information than just regular results.

B. Temporal Point Process

1) Model:

Before presenting the analysis part of temporal point process, we first briefly introduce the model of temporal point process. Point processes are collections of random points falling in some space, such as time and location. Point processes provide the statistical language to describe the timing and properties of events [16]. Point-process models are useful for answering a range of different questions. These include explaining the nature of the underlying process, simulating future events, and predicting the likelihood and volume of future events.

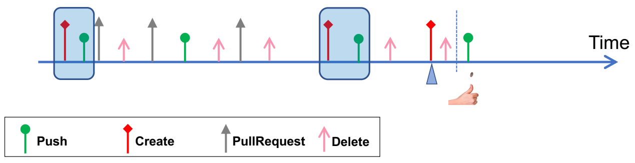


Fig. 8. An example of point process.

Mathematically, point process is a natural extension of the well-known Poisson process. Poisson process assumes that the points(events) on the timeline are independent to each other, and the probability of a new event has no relation to the event history. This independent assumption gives the Poisson process some graceful properties. For example, any inter-event time is an exponential distribution, and given any period with length l, the distribution of event number is a Poisson distribution with parameter λl, where λ is the intensity parameter of Poisson process.

However, in point process, we seek to describe the inter-dependent relationship between different events. Thus, the independent assumption does not hold anymore, as described in Figure 8. The inter-event times now are dependent on the previous events. A temporal point process can be completely specified by the distribution of its inter-event times:

$$f(t_1, \ldots, t_n) = \prod_{i=1}^{n} f(t_i|t1, \ldots, t_{i-1}) = \prod_{i=1}^{n} f^*(t_i)$$

Denote the event history as $\mathcal{H}_t$, to define a temporal point process, it is sufficient to define $f(t|\mathcal{H}_t)$, or $f^*(t)$. More directly, we can define it by using a conditional intensity function $\lambda(t)$:

$$\lambda^*(t) = \frac{f^*(t)}{1 - F^*(t)}$$

It represents the probability for the new event to happen in t, whose definition is given by:

$$\begin{aligned}\lambda &= \lim_{\Delta t \to 0} \frac{\mathbb{E}(N(t + \Delta t) - N(t)|\mathcal{H}_t)}{\Delta t} \\ &= \lim_{\Delta t \to 0} \frac{\mathbb{P}(N(t + \Delta t) - N(t) = 1)|\mathcal{H}_t)}{\Delta t}\end{aligned}$$

We will introduce the formula we choose in this project then. We model the event sequence of each repository in previous introduced repo set as a separate point process, with each event type as a dimension within it. Note here we use the multi-dimensional Hawkes point process, instead of the homogeneous process in Figure 9. We only model the popular repos selected by the previous subsection.

In our model, for each dimension(event type), its conditional intensity function is given by:

$$\lambda_i(t) = \mu_i + \sum_{j=1}^{D} \sum_{t_k^j < t} \phi_{ij}(t - t_k^j)$$

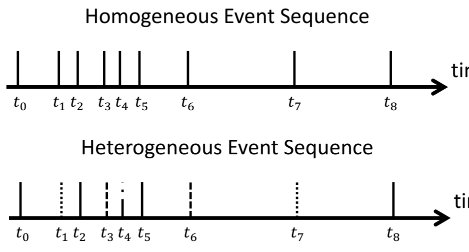

Homogeneous Event Sequence
time
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8
Heterogeneous Event Sequence
time
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8

Fig. 9. Homogeneous and heterogeneous event sequences.

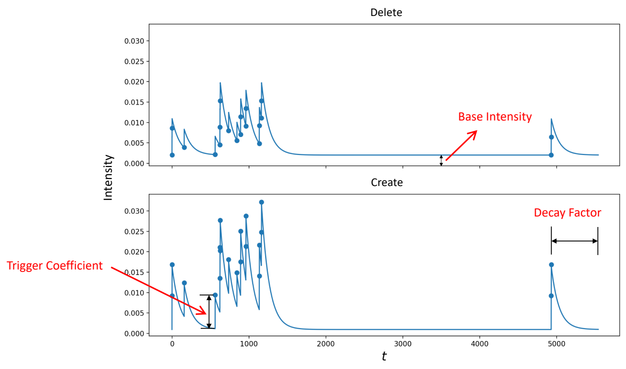


Fig. 10. Example of Point process intensity.

where the triggering kernel $\phi_{ij}(t)$ is:

$$\phi_{ij}(t) = \alpha^{ij}\beta^{ij}\exp(-\beta^{ij}t)1_{t>0}$$

Figure 10 presents an example of the meaning of intensity function we introduced above. We can see that this model focus on the short time dependency. If two event types often comes together with very short time interval, they will probably have a high triggering effect between these two event types. But on the other hand, this is not guaranteed, cause they may be independent to each other, but it is just coincident that they happen together.

Actually, for the condition we mentioned above, the point process model can automatically distinguish it from the dependent event types. Note that we have a base intensity and a triggered intensity for each dimension. The triggered intensity indeed describe the burst property of events in this dimension. If two event types constantly happen together, but they do not affect the arrival rate of each other, the inferred result will present a high base intensity on both dimensions, but a low triggering coefficient, because they do not lead to a change in intensity to each other. Next we will show the analysis part of this model.

2) Analysis:

As mentioned before, we model the repositories in the selected active repo set as independent point processes, and independently estimate the parameters for each repository. We first derive the log-likelihood function of a given repo event sequence, based on the intensity function above:

$$\log L = \sum_{i=1}^{D}\left(\int_0^T \lambda_i(T)dt - \int_0^T \log\lambda_i(t)dN_i(t)\right)$$

For each repo event sequence with k event types, we have three parts of parameters to estimate:

- μ: a vector of k, it represents the base intensity of each dimension. It determines the arrival rate of events triggered by the extraneous reasons.
- α: a matrix of shape $k \times k$, it denotes the triggering coefficients within and across each dimension. In Figure 10, it corresponds to the extent of intensity jump when a new event happens. And this part of parameters is exactly what we use to quantify the dependencies across different event types.
- β: a vector of k, it represents the decay factor of the triggering effect. We have to set the triggering effect between dimensions as a time decay function, otherwise it will result in an infinite loop of event triggering.

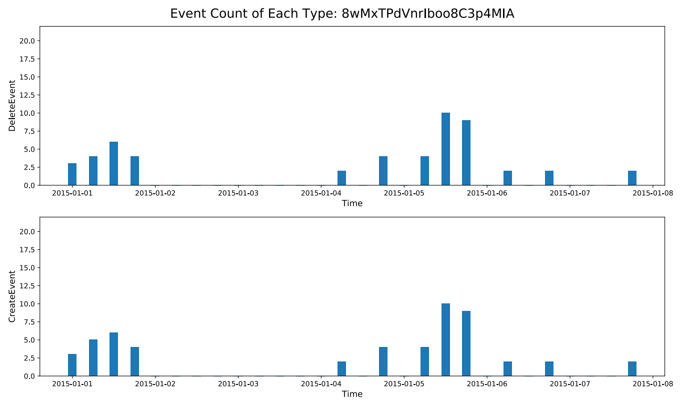
Due to the existence of decay factors in the parameters, this maximum likelihood estimation problem is no longer a convex optimization problem. Therefore, we have to utilize some iterative algorithms or sampling methods, such as expectation-maximization, MCMC, or moment matching algorithms. Here, we choose the Broyden-Fletcher-Glodfarb-Shanno(BFGS) algorithm [17], which is a widely used quasi-Newtons method for unconstrained optimization problems without requiring the computation of second-order partial derivatives.

Figure 11 shows a real repo event time distribution in Jan. 2015, and its estimated parameters. The event data on this repo always has create event following a delete event within a few seconds. Therefore, we have a very high triggering coefficient from delete event to create event, but relatively low triggering coefficient from create event to delete event. This is what we previously called time-varying triggering effect. In frequent pattern mining, they are definitely regarded the same, cause delete and create always accompany each other. Only from the time information can we tell that it is indeed the delete event triggers the create event, but not in the inverse direction.

After individually modeling and estimating the event sequence for each popular repository, we combine their triggering matrix by a weighted sum step, with the weight proportional to the total number of events on this repo. The heuristic behind this step is that we believe the triggering factor derived from a very popular repo with a large number of events is more convincing than a repo with less events.

C. Mutual Enhancement between Two Models

Although the two models seem to be redundant to each other, where they are both used to mine the frequent event patterns and dependencies across event types. However, they are actually distinct to each other, and can enhance the result of each other very well. We next explain how does it work in the mutual enhancement.



Base Intensity

	Delete	Create
Base	0.00263	0.000204

Triggering Matrix

Trigger	Delete	Create
Delete	0	0.999
Create	0.508	0

Decay Factor

	Delete	Create
Decay	0.454	23.552

Feature: High value in some non-diagonal elements in triggering matrix.

Fig. 11. A real repo example with its estimated parameters.

- Point process $\rightarrow$ pattern mining: The complement in this direction is reflected in two parts: 1) Short-term time varying dependency provided by point process. As described in previous parts, the pattern mining techniques do not distinguish the events that belong to the same transaction, and the time information is only used to divide the transactions. In contrary, the point process nicely handle these two aspect in its design. Neighbor events with different orders and inter-event times are assigned different parameters. 2) Distinguish coincidence and real dependency. Not every frequent pattern is truly existed dependency. It can be just coincidence. However, in point process, we not only rely on the frequency/support, but also the change in intensity caused by the preceding event, which is more reliable in identifying the dependency.
- Pattern mining $\rightarrow$ point process: The point process usually ignore the dependencies that may range a relatively long time(i.e., across transactions), where exactly pattern mining is good at. The frequent pattern mining patterns can skip the events in the same transaction, and capture the frequent pairs/tuples across consecutive transactions. This helps improve the quality and completeness of dependencies of point process results.

IV. Results

A. Sequential pattern mining

After running sequential pattern mining algorithm on January 2015 GitHub dataset, we find both events patterns and user-event tuple patterns. The event patterns are sequential collection of event types, which tell us what types of events often happen together in the same group. Events that separately occur in different item sets usually mean that one type of event has some influence on the other type in consecutive time periods. Table II presents the event sequential patterns.

In Table II, we find many reasonable patterns. For example, from $ID = 1$ we see that PushEvent, IssueCommentEvent, and PullRequestEvent often occur together. From $ID = 2$ we see that PushEvent, DeleteEvent often appear close in time. Maybe some users delete a repository and set up a new one and push the updated contents. From $ID = 3$ we see that $\{$PushEvent, ReleaseEvent$\}$ often leads to $\{$IssueCommentEvent$\}$. From $ID = 6$ we see that PushEvent is often followed by ReleaseEvent. Other patterns may tell us that for example, WatchEvent and ForkEvent and IssueCommentEvent often interact with each other.

Next we look at the user-event tuple sequential patterns presented in Table III. The format of the pattern is $\{$`user1:event1, user2:event2`$\} \rightarrow \{$`user3:event3`$\}$. User plus event tuple sequences may be more informative as we can know users' interactions as well. For example, from $ID = 10$ we see that one user pushes are often followed by another user's comment. From $ID = 11, 12$ we see that if one user pushes something to their repository, some other users may later issue comments, or have pull requests. From $ID = 3$ we observe that if create events and delete events happen together, they usually come from the same user.

In summary, these maximal sequential patterns are reasonable and can help us model user behavior in GitHub community and even other platforms in general.

B. Temporal point process

For each qualified repository we selected from January 2015 GitHub data, we first individually model its event sequence by point process and estimate the three parts of parameters jointly by maximum likelihood estimation. After that, to summarize the general event patterns from all repositories, we perform weighted sum in the derived triggering matrix by each repo, with weight proportional to the total number of events on this

TABLE II

EVENT SEQUENTIAL PATTERNS.

ID	Pattern	Support
1	{PushEvent, IssueCommentEvent, PullRequestEvent}	0.5094
2	{PushEvent, DeleteEvent}	0.2300
3	{PushEvent, ReleaseEvent} → {IssueCommentEvent}	0.020
4	{PushEvent, MemberEvent} → {PushEvent}	0.0243
5	{PushEvent} → {PullRequestEvent}	0.5782
6	{PushEvent} → {ReleaseEvent}	0.0229
7	{IssueCommentEvent, PullRequestEvent} → {IssueCommentEvent}	0.5288
8	{IssueCommentEvent, IssuesEvent}	0.5546
9	{IssueCommentEvent} → {PushEvent, PullRequestEvent}	0.5010
10	{IssueCommentEvent} → {IssueCommentEvent, PullRequestEvent}	0.5261
11	{IssueCommentEvent} → {IssueCommentEvent} → {ForkEvent}	0.5205
12	{IssueCommentEvent} → {ReleaseEvent}	0.0215
13	{PullRequestEvent, ForkEvent}	0.5511
14	{PullRequestEvent, DeleteEvent}	0.2196
15	{PullRequestEvent, MemberEvent}	0.0278
16	{PullRequestEvent} → {PushEvent, PullRequestEvent}	0.5455
17	{PullRequestEvent} → {IssueCommentEvent, PullRequestEvent}	0.5337
18	{PullRequestEvent} → {ForkEvent}	0.5573
19	{PullRequestEvent} → {WatchEvent}	0.5024
20	{PullRequestEvent} → {ReleaseEvent}	0.0236
21	{ForkEvent} → {PushEvent}	0.5184
22	{ForkEvent} → {IssueCommentEvent} → {IssueCommentEvent}	0.5754
23	{ForkEvent} → {IssueCommentEvent} → {WatchEvent}	0.5108
24	{ForkEvent} → {PullRequestEvent}	0.6074
25	{ForkEvent} → {WatchEvent} → {ForkEvent}	0.5059
26	{ForkEvent} → {IssuesEvent}	0.5101
27	{WatchEvent} → {IssueCommentEvent} → {IssueCommentEvent}	0.5101
28	{WatchEvent} → {IssueCommentEvent} → {WatchEvent}	0.5170
29	{WatchEvent} → {PullRequestEvent}	0.5115
30	{WatchEvent} → {ForkEvent} → {IssueCommentEvent}	0.5073
31	{IssuesEvent} → {IssueCommentEvent}	0.5962
32	{IssuesEvent} → {ForkEvent}	0.5017
33	{ReleaseEvent} → {PushEvent}	0.0264
34	{ReleaseEvent} → {CreateEvent}	0.0202
35	{ReleaseEvent} → {PullRequestEvent}	0.0236
36	{MemberEvent} → {PushEvent} → {PushEvent}	0.0243

repo. We present the final triggering coefficients returned by the temporal point process in Table IV.

Similar as before, we can also find many interesting and meaningful patterns from this table. For example, from $ID = 4$, after creating a new branch for a repo, you can do some side update on this branch and then request master branch to merge your update. From $ID = 7$, after you pull request is accepted, your local branch will be useless, thus you will delete this branch. From $ID = 12$, the push events usually happens in burst. From $ID = 13$, after creating a repo or creating a new branch for this repo, you will often do some updates and push the change to this repo. From $ID = 16$, when you reply to an opened issue, you will be probably get the response in short time. We will not iterate over all patterns. But we just want to express that many patterns returned by the temporal point process are interesting and contains actual meanings, which verifies the capability of this model.

C. Combined Results

In this part, we combined the event-event patterns in Table II and Table IV to summarize the high-quality patterns that both appear in short-term dependencies and cross-transaction association rules. We define the high-quality event-event pattern as those appear in both short term dependencies in Table IV and long term patterns in Table II. The high-quality patterns we find in this dataset is shown in Table V.

TABLE V

HIGH QUALITY EVENT-EVENT PATTERNS.

ID	Pattern
1	{PushEvent} → {DeleteEvent}
2	{PushEvent} → {PullRequestEvent}
3	{PushEvent} → {ReleaseEvent}
4	{PushEvent} → {PushEvent}
5	{PullRequestEvent} → {DeleteEvent}
6	{PullRequestEvent} → {PushEvent}
7	{PullRequestEvent} → {PullRequestEvent}
8	{ForkEvent} → {PullRequestEvent}
9	{WatchEvent} → {ForkEvent}
10	{IssuesEvent} → {IssueCommentEvent}
11	{ReleaseEvent} → {CreateEvent}
12	{IssueCommentEvent} → {IssueCommentEvent}

As we can see from Table V, after the combination and mutual filtering fromGitHub two tables, the final distilled patterns are much more meaningful and useful than the intermediate

TABLE III
USER-EVENT SEQUENTIAL PATTERNS.

ID	Pattern	Support
1	{1:CreateEvent, 1:PushEvent}	0.1557
2	{1:CreateEvent, 1:PullRequestEvent}	0.1371
3	{1:CreateEvent, 1:DeleteEvent}	0.0599
4	{1:PushEvent, 1:IssueCommentEvent, 1:PullRequestEvent}	0.1477
5	{1:PushEvent, 1:PullRequestEvent} $\rightarrow$ {1:PushEvent}	0.1300
6	{1:PushEvent, 1:DeleteEvent}	0.1004
7	{1:PushEvent} $\rightarrow$ {1:PullRequestEvent}	0.1404
8	{1:PushEvent} $\rightarrow$ {1:DeleteEvent}	0.0554
9	{1:PushEvent} $\rightarrow$ {2:PushEvent}	0.1259
10	{1:PushEvent} $\rightarrow$ {2:IssueCommentEvent}	0.1346
11	{1:PushEvent} $\rightarrow$ {2:PullRequestEvent}	0.1468
12	{1:PushEvent} $\rightarrow$ {2:DeleteEvent}	0.0550
13	{1:IssueCommentEvent, 1:IssuesEvent}	0.1736
14	{1:IssueCommentEvent, 2:IssueCommentEvent}	0.1457
15	{1:IssueCommentEvent} $\rightarrow$ {1:PushEvent}	0.1418
16	{1:IssueCommentEvent} $\rightarrow$ {1:PullRequestEvent}	0.1449
17	{1:IssueCommentEvent} $\rightarrow$ {2:PushEvent}	0.1382
18	{1:IssueCommentEvent} $\rightarrow$ {2:PullRequestEvent}	0.1807
19	{1:PullRequestEvent, 1:ForkEvent}	0.1628
20	{1:PullRequestEvent, 1:DeleteEvent}	0.1004
21	{1:PullRequestEvent} $\rightarrow$ {1:IssueCommentEvent}	0.1732
22	{1:PullRequestEvent} $\rightarrow$ {1:DeleteEvent}	0.0537
23	{1:PullRequestEvent} $\rightarrow$ {2:PushEvent}	0.1569
24	{1:PullRequestEvent} $\rightarrow$ {2:IssueCommentEvent}	0.1869
25	{1:PullRequestEvent} $\rightarrow$ {2:DeleteEvent}	0.0533
26	{1:DeleteEvent} $\rightarrow$ {1:PushEvent}	0.0535
27	{1:DeleteEvent} $\rightarrow$ {2:PushEvent}	0.0527
28	{1:DeleteEvent} $\rightarrow$ {2:PullRequestEvent}	0.0522
29	{1:IssuesEvent} $\rightarrow$ {1:IssueCommentEvent}	0.1294

result from two sources. We can basically find a actual scenario for all the high quality patterns appearing in this table. For example, from $ID = 1$, after pushing and merging the result in the master branch, the programmer will often delete the side branch. And from $ID = 8$, after forking a repository and fix some bugs, the programmer probably submit a pull request to the owner of that repo to merge his update. And from $ID = 10$, an issue event may lead to the responses from the contributors from the community and $ID = 12$ indicates the continuous discussions in the issue page. Therefore, we can claim that the combination of sequential pattern mining technique and temporal point process model could mutually enhance each other to distill real high quality event patterns in GitHub, which is both short-term and long-term dependent to each other.

V. Conclusion

This paper presents the dependency and frequent pattern mining combining the frequent pattern mining and temporal point process model using Github event dataset. We characterize the repositories by showing their statistics and characterize events by finding their triggering effects. We also treat time series events as sequential patterns and adopt a well known method to mine sequential patterns. The profiling of user and event interactions help tremendously in analyzing social networks and simulating human behaviors. Through comparison of result returned by each method, and combined results from two parts, we conclude that the combination of these two methods could obtain a set of much more high-quality and meaningful event patterns and dependencies, integrating the time-seperated patterns and short-term time-varying dependencies.

References

[1] M. A. Russell, *Mining the Social Web: Data Mining Facebook, Twitter, LinkedIn, Google+, GitHub, and More*. " O'Reilly Media, Inc.", 2013.

[2] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *ACM sigmod record*, vol. 29, no. 2. ACM, 2000, pp. 1–12.

[3] F. Zhu, X. Yan, J. Han, S. Y. Philip, and H. Cheng, "Mining colossal frequent patterns by core pattern fusion," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 2007, pp. 706–715.

[4] C. C. Aggarwal and J. Han, *Frequent pattern mining*. Springer, 2014.

[5] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "Mining sequential patterns by pattern-growth: The prefixspan approach," *IEEE Transactions on knowledge and data engineering*, vol. 16, no. 11, pp. 1424–1440, 2004.

[6] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, "Freespan: frequent pattern-projected sequential pattern mining," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2000, pp. 355–359.

[7] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han, "Dustminer: troubleshooting interactive complexity bugs in sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, 2008, pp. 99–112.

[8] A. G. Hawkes, "Spectra of some self-exciting and mutually exciting point processes," *Biometrika*, vol. 58, no. 1, pp. 83–90, 1971.

[9] P. Embrechts, T. Liniger, and L. Lin, "Multivariate hawkes processes: an application to financial data," *Journal of Applied Probability*, vol. 48, no. A, pp. 367–378, 2011.

TABLE IV
EVENT TYPES DEPENDENCIES.

ID	Pattern	Triggering Coefficient
1	ReleaseEvent → MemberEvent	0.827849
2	PublicEvent → ForkEvent	0.363926
3	PullRequestReviewCommentEvent → PullRequestReviewCommentEvent	0.262126
4	CreateEvent → PullRequestEvent	0.242272
5	MemberEvent → MemberEvent	0.191892
6	PullRequestEvent → PushEvent	0.191744
7	PullRequestEvent → DeleteEvent	0.186207
8	GollumEvent → GollumEvent	0.157783
9	CommitCommentEvent → CommitCommentEvent	0.144221
10	MemberEvent → GollumEvent	0.142544
11	PushEvent → DeleteEvent	0.126622
12	PushEvent → PushEvent	0.125056
13	CreateEvent → PushEvent	0.119058
14	PushEvent → ReleaseEvent	0.117570
15	PullRequestEvent → IssueCommentEvent	0.117371
16	IssueCommentEvent → IssueCommentEvent	0.114982
17	PullRequestReviewCommentEvent → IssueCommentEvent	0.114367
18	ForkEvent → PullRequestEvent	0.111687
19	CreateEvent → MemberEvent	0.106749
20	IssueCommentEvent → PullRequestEvent	0.105892
21	PublicEvent → PullRequestEvent	0.101896
22	PushEvent → CreateEvent	0.100707
23	IssueCommentEvent → PushEvent	0.098150
24	PushEvent → PullRequestEvent	0.094478
25	PullRequestEvent → PullRequestReviewCommentEvent	0.093788
26	PushEvent → IssueCommentEvent	0.093015
27	PullRequestEvent → PullRequestEvent	0.087576
28	ReleaseEvent → CreateEvent	0.083736
29	PullRequestReviewCommentEvent → PushEvent	0.083532
30	IssuesEvent → IssueCommentEvent	0.079877
31	IssuesEvent → IssuesEvent	0.079727
32	DeleteEvent → CreateEvent	0.077415
33	WatchEvent → ForkEvent	0.077222
34	MemberEvent → CreateEvent	0.075559
35	DeleteEvent → DeleteEvent	0.073422
36	CommitCommentEvent → PushEvent	0.071292
37	IssueCommentEvent → IssuesEvent	0.063706
38	MemberEvent → IssuesEvent	0.062428
39	ReleaseEvent → DeleteEvent	0.062424
40	GollumEvent → IssuesEvent	0.062234
41	PushEvent → PullRequestReviewCommentEvent	0.061823
42	MemberEvent → PushEvent	0.059514
43	ForkEvent → MemberEvent	0.059135
44	GollumEvent → PullRequestEvent	0.057699
45	CreateEvent → IssueCommentEvent	0.056191
46	IssueCommentEvent → PullRequestReviewCommentEvent	0.054704
47	IssueCommentEvent → DeleteEvent	0.054695
48	PushEvent → CommitCommentEvent	0.054116
49	CreateEvent → CreateEvent	0.050834
50	PullRequestEvent → CreateEvent	0.050071

[10] Y. Ogata, "Statistical models for earthquake occurrences and residual analysis for point processes," *Journal of the American Statistical association*, vol. 83, no. 401, pp. 9–27, 1988.

[11] Q. Zhao, M. A. Erdogdu, H. Y. He, A. Rajaraman, and J. Leskovec, "Seismic: A self-exciting point process model for predicting tweet popularity," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 1513–1522.

[12] N. Du, H. Dai, R. Trivedi, U. Upadhyay, M. Gomez-Rodriguez, and L. Song, "Recurrent marked temporal point processes: Embedding event history to vector," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 1555–1564.

[13] S. Mishra, M.-A. Rizoiu, and L. Xie, "Feature driven and point process approaches for popularity prediction," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 1069–1078.

[14] Y. Wang, X. Ye, H. Zhou, H. Zha, and L. Song, "Linking micro event history to macro prediction in point process models," in *Artificial Intelligence and Statistics*, 2017, pp. 1375–1384.

[15] T. Kurashima, T. Althoff, and J. Leskovec, "Modeling interdependent and periodic real-world action sequences," in *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2018, pp. 803–812.

[16] M.-A. Rizoiu, Y. Lee, S. Mishra, and L. Xie, "A tutorial on hawkes processes for events in social media," *arXiv preprint arXiv:1708.06401*, 2017.

[17] D. C. Liu and J. Nocedal, "On the limited memory bfgs method for large scale optimization," *Mathematical programming*, vol. 45, no. 1-3, pp. 503–528, 1989.