Data Stream Event Prediction Based on Timing Knowledge and State Transitions

Yan Li
University of Massachusetts, Lowell, MA, USA
yli2@cs.uml.edu

Tingjian Ge
University of Massachusetts, Lowell, MA, USA
ge@cs.uml.edu

Cindy Chen
University of Massachusetts, Lowell, MA, USA
cchen@cs.uml.edu

ABSTRACT

We study a practical problem of predicting the upcoming events in data streams using a novel approach. Treating event time orders as relationship types between event entities, we build a dynamic knowledge graph and use it to predict future event timing. A unique aspect of this knowledge graph embedding approach for prediction is that we enhance conventional knowledge graphs with the notion of "states"—in what we call the ephemeral state nodes—to characterize the state of a data stream over time. We devise a complete set of methods for learning relevant events, for building the event-order graph stream from the original data stream, for embedding and prediction, and for theoretically bounding the complexity. We evaluate our approach with four real world stream datasets and find that our method results in high precision and recall values for event timing prediction, ranging between 0.7 and nearly 1, significantly outperforming baseline approaches. Moreover, due to our choice of efficient translation-based embedding, the overall throughput that the stream system can handle, including continuous graph building, training, and event predictions, is over one thousand to sixty thousand tuples per second even on a personal computer—which is especially important in resource constrained environments, including edge computing.

PVLDB Reference Format:
Yan Li, Tingjian Ge, and Cindy Chen. Data Stream Event Prediction Based on Timing Knowledge and State Transitions. *PVLDB*, 13(10): 1779-1792, 2020.
DOI: https://doi.org/10.14778/3401960.3401973

1. INTRODUCTION

Event matching, as part of complex event processing (CEP), is one of the most important topics in data streams [13]. Many commercial systems (e.g., [12, 1]) have implemented event matching and CEP. However, little work has been done on *predicting the timing* of an interesting event, which we call a *target event*—whether it will happen soon or long after the current time.

Example 1. *Outpatient monitoring and management of Type 1 diabetes (T1D) is a critical issue [23]. It relies principally on three interventions: diet, exercise, and exogenous insulin. Diabetes patient information is obtained from an automatic electronic recording device. The automatic device has an internal clock to timestamp events. We consider signals from a short period of time as a tuple, and there are multiple attributes in a tuple, such as regular insulin dose, NPH insulin dose, blood glucose measurement at a pre-meal or post-meal, hypoglycemic symptoms, typical or more/less-than-usual meal ingestion, and typical or more/less-than-usual exercise activity. The continuous monitoring data forms a multi-attribute data stream. Doctors may match interesting event patterns; they may also want to predict some events of interest, such as hypoglycemic conditions in the near future, or a blood glucose measurement that will increase significantly. Doctors and/or outpatients may be notified when such predictions occur, and critical interventions can be performed to prevent undesirable events.*

There are many other examples. Prediction of undesirable events in oil and gas wells can help prevent production losses, environmental accidents, and human casualties, and help reduce maintenance costs [43]. The stream tuples contain various measurement events in oil wells, as well as a number of undesirable events such as abrupt increase of Basic Sediment and Water (BSW), spurious closure of Downhole Safety Valve (DHSV), and severe slugging. The measurement events include pressure events at Permanent Downhole Gauge (PDG), temperature events at Temperature and Pressure Transducer (TPT), pressure events at TPT, among many others. This stream can be very fast, and we are managing many wells. The prediction must be quick and early for actions.

Yet another example, as used in our experiments (Section 6), is in the transportation domain [16]. In a metropolitan area, average *delay scores* of taxi trips, defined as the ratio between trip time and trip distance, are expensive to obtain as they require detailed information of all trips. The target events, such as delay score being very high in several locations of interest, can be predicted in advance based on a combination of basic events at a few locations that have cameras or sensors to only record simple statistics such as the frequency of incoming and outgoing taxis.

In addition to early actions to cope with a predicted event, event timing prediction is also useful in *predictive complex event processing* and *best-effort complex event matching under resource constraints* [22, 20, 31]. While there is much previous work on time series forecasting [18], little has been done on multi-attribute data stream discrete-event timing prediction—whether a target event will happen soon, or whether an event will happen much later, using limited training data to promptly build a model and predict. There is previous work on sequential association rules [40]. However, our experiments show that our proposed approach provides much better prediction accuracy in terms of precision and recall.

Proceedings of the VLDB Endowment, Vol. 13, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3401960.3401973

1.1 Requirements and Our Approach

Consistent with other data stream applications and methods, the training of the predictive model should be fast and possibly performed in real time. Along with this, the training should ideally only use a small amount of most recent data, so that the learning and training of models can be continuous, adaptive to any changes in data streams. For instance, deep layers of neural networks [24], which require a large amount of training data and heavy resources including GPUs and long training time, are unfit for fast real-time adaptive stream computing, which is sometimes even pushed to less powerful devices in edge computing [21]. Consider, for example, the oil wells application above. Each of the many wells produces a lot of sensory data very fast, and the wells may be geographically very dispersed. At each distant location near the well where the data is produced, it would be ideal to have a laptop or even a less powerful device to perform the undesirable-event model learning and prediction in place, rather than communicating all the data from each well to a central server for processing, which can be too slow and overwhelm the network. If needed, only detected critical events can be communicated to the central server for records and actions. Likewise, the same situation happens with the metropolitan traffic example above or self-driving cars and smart cities. Instead of communicating large amounts of streaming data to a central server, each light device can perform critical event prediction in place.

We propose an approach that is based on some state-of-the-art result in machine learning called *graph embedding* [10]. There are several novel aspects of our approach that are significantly different from prior work using graph embedding.

- First and foremost, we create a knowledge graph [25] that characterizes the *timing relationships* between two events, where the relationships include "happening soon after" and "happening long after".
- Unlike conventional knowledge graphs, we introduce the notion of "active state" that characterizes the events that hold true at the current timestamp, or at a particular timestamp of interest. From a different perspective, our model may also be considered as a *knowledge enhanced state machine*.
- For embedding, we introduce "ephemeral nodes" associated with each active set of the timing knowledge graph, whose embedding vectors are derived from the nodes in the active set through *ephemeral edges* and *attention* parameters (details in Section 3).

To build the timing knowledge graph, we study the problem of what set of *relevant events* to use as the nodes, in order to predict a given set of *target events*. Intuitively, a relevant event appears often in the vicinity (or context) of a target event, but appears significantly less often in the general data stream. For instance, in Example 1, for a target event "blood glucose measurement higher than the previous one", a relevant event may be "more-than-usual meal ingestion". We adopt the notion of *tf-idf* (term frequency – inverse document frequency) [34] from information retrieval as a metric to get the top-k relevant events with the highest tf-idf. All these events are the conjunction of one or more *basic events*, each of which is a primitive predicate over a tuple.

However, the search is very expensive and may involve multiple rounds of parsing the stream training data. We devise an efficient one-pass algorithm that uses fast bitmap operations and A* search (pruning and bounding). Finally, we provide a novel analysis of the error bound of prediction result using the Rademacher complexity theory.

Our experiments over four real-world datasets show that our proposed algorithms are very efficient. We can see the fast decrease and convergence of the loss function value for our attention-based ephemeral node embedding after around 50 epochs. Due to our choice of efficient translation-based embedding, the overall throughput that the stream system can handle, including continuous graph building, continuous training, and event predictions, is over one thousand to sixty thousand tuples per second even on a personal computer—which is especially important in resource constrained environments, including edge computing [21]. While the system throughput is high, the trained model also achieves high precision and recall values for event timing predictions, ranging from around 0.7 to nearly 1, much higher than the baseline approaches.

In summary, our contributions are as follows:

- We propose to build timing knowledge graphs for events in data streams to predict the timing of target events **(Sec. 3)**.
- We devise a novel graph embedding algorithm that incorporates the notions of active states, ephemeral nodes, and the attention mechanism **(Sec. 3)**.
- We design an efficient one-pass algorithm to learn the top relevant events as nodes of the graph, using tf-idf from information retrieval and an A* search **(Sec. 4)**.
- We analyze the *data-dependent* error bounds of the prediction using Rademacher complexity theory **(Sec. 5)**.
- We perform a systematic empirical study that demonstrates the high accuracy of our predictions, and the high efficiency of the graph-building and training algorithms **(Sec. 6)**.

2. PROBLEM STATEMENT

2.1 Problem Formulation

We are given a data stream $\mathcal{S}$ that consists of a sequence of records $(r_1, t_1), (r_2, t_2), ...$ where $t_1 < t_2 < \cdots$. An *event* at time t_i is a Boolean predicate over the record r_i, involving one or more attributes of r_i.

Let the time of record r^* be t^*. We say that event e will occur *soon* after r^* (or t^*) if it is true in record (r_i, t_i) and $0 < t_i - t^* < \delta_1$, for a (small) constant value δ_1. On the other hand, if e is false in each record (r_i, t_i) for $t_i \leq t^* + \delta_2$ (where $\delta_2 > \delta_1$ is a constant), we say that e happens *long* after r^* (or t^*). Note that the semantics on the event timing gaps δ_1 and δ_2 can either be count-based (i.e., number of records) or time-based. Without loss of generality, we assume the time-based semantics (the count-based one is essentially integer timestamps).

Given a set of interesting events $e_1, e_2, ..., e_k$, and the current record (r^*, t^*), and a window of most recent data before r^*, the problem is to predict whether each event e_j $(1 \leq j \leq k)$ will happen soon after r^*, or whether e_j will happen long after r^*.

Note that we focus on point-based events in this paper. Time relationship between interval-based events are more complicated [3], which could be a topic of future work. Nonetheless, even with interval events, it is often sufficient to be able to predict the critical time points within them, such as the start of an event, the end of an event, or other major time points.

2.2 Preliminaries

We first survey some background knowledge necessary for the rest of the paper. The terms *graph* and *network* are used interchangeably.

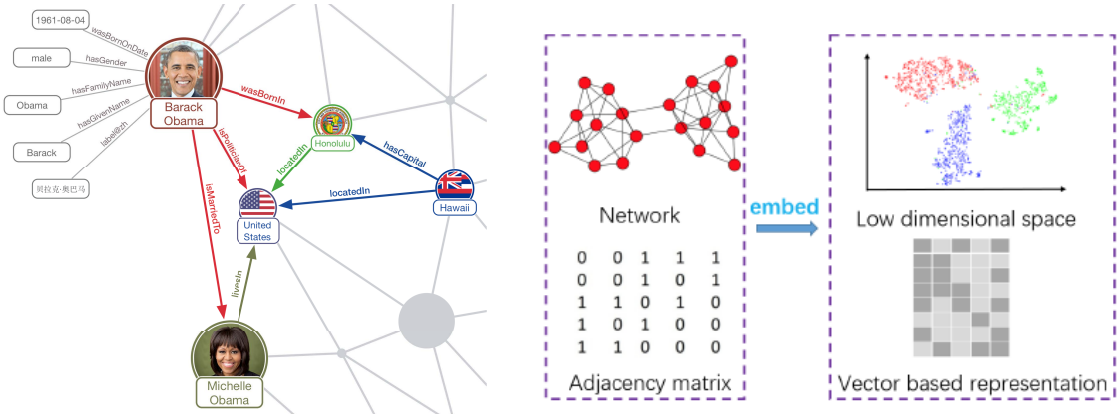

Figure 1: Illustrating the concepts of (a) a knowledge graph and (b) network embedding [14].

Knowledge Graphs. The Knowledge Graph is a knowledge base invented and first used by Google to enhance its search engine [39], as illustrated in Figure 1(a). It is basically a graph of interconnected entities. In Figure 1(a), various types of entities (e.g., people, countries, states/cities) as *vertices* are depicted, along with diverse types of relationships between entities shown as *edges*, such as "isMarriedTo" and "wasBornIn". Essentially each fact is represented as a triple (*subject, relationship, object*), where subject and object are also called *head* and *tail* nodes, respectively.

Knowledge Graph Embedding. Network/graph embedding in general [26] is used to map each node (and relationship type) in a network to a vector (point) in another multidimensional space. The basic goal is that if two nodes are "similar" or "close" in the graph, then the two points that they map to should also be close in the projected space, based on a certain distance metric. This is illustrated in Figure 1(b), where the adjacency matrix and the original graph on the left hand side have complicated structural correlations (hence high dimensionality), and the nodes are embedded as vectors in a low dimensional space (three dimensions for visualization) on the right hand side [14]. Within each of the three clusters in different colors, the nodes are "similar" or "close" in the original graph, however the notion of similarity is defined and used in the embedding training process.

A common type of *knowledge graph* embedding is *translational distance* models, which exploit distance-based scoring functions and measure the plausibility of a fact as the distance between the two entities, usually after a translation carried out by the relation. For example, a simple and efficient one is TransE [7], where the training tries to enforce the vectors satisfying $\mathbf{h}+\mathbf{r} \approx \mathbf{t}$, where $\mathbf{h}$, $\mathbf{t}$, and $\mathbf{r}$ are the embedding vectors of the head and tail entities, and the relationship type between them. The same soft constraint is used to predict the relationship between two entities.

In essence, graph embedding is *automatic latent feature extraction* and *dimensionality reduction*, as each value in the vectors associated with the nodes and relationship types is a latent/hidden feature, which is hard to be extracted by conventional feature engineering.

3. EVENT-TIMING GRAPHS AND STATE-BASED EMBEDDING

We first describe the core of our approach, which is to efficiently build a event-timing knowledge graph from one pass of the training data, and to efficiently perform embedding over this graph based on the new notions of active sets, ephemeral nodes, and attentions on ephemeral edges.

Note that, in this section, we assume that we are given the events to be used for building our data structure (the event graph). In Section 4, we still study the selection of relevant events to use for the algorithms in this section. Let us start with some intuitions in Section 3.1, followed by the details of the algorithms in Section 3.2.

3.1 Intuition and Basic Idea

The basic idea is that we represent a number of key relevant events (with respect to the given target events) as vertices V in a knowledge graph $\mathcal{G}$. The set of edges E in $\mathcal{G}$ indicates *timing* relationships. A directed edge (u, v) from event u to event v may have one of the two types of relationships: r_1 indicating that event v happens soon after event u, and r_2 indicating that event v happens long after event u.

On top of the basic graph elements described above, we define the notion of an activation graph formally below.

Definition 1. ***(Activation Graph, Active Set, and Ephemeral Node).*** *An* activation graph $\mathcal{G}_\mathcal{S}$ *of a data stream* $\mathcal{S}$ *is a dynamic graph where each vertex is an event.* $\mathcal{G}_\mathcal{S}$ *has* states*: at any moment, there is a subset of vertices* V_a *that are active, called the* active set*, which are the events that are true for the current tuple* (r_i, t_i) *in* $\mathcal{S}$*. The lifetime of* V_a *is* $[t_i, t_{i+1})$*, i.e., before the next tuple arrives.*

In addition, a pair (V_a, v_e) *is associated with an* ephemeral node v_{ae} *that has the same lifetime as* V_a*, where* v_e *is the target event after* V_a*, with either* r_1 *or* r_2 *relationship, indicating that the event* v_e *happens* soon after V_a *or* much later than V_a*, respectively. The edges of* $\mathcal{G}_\mathcal{S}$ *are of two categories, which are both ephemeral and incident to an ephemeral node at one end. The first category of edge is from each vertex* v_u *in* V_a *to* v_{ae}*, with a relationship type* r_{ue} *(corresponding to event nodes* v_u *and* v_e*), while the second category is from* v_{ae} *to an event vertex* v_e *with a relationship type either* r_1 *or* r_2*.*

Example 2. *Figure 2 shows an example, where each solid vertex is an event node while the two hollow vertices are ephemeral nodes. The four event vertices inside the green solid circle are the active set* V_a *at some time instant* t*;* V_a *is associated with an ephemeral node* v_{ae}*, for a target event* v_e *that follows with relationship* r_1 *(soon after). For instance, continuing Example 1, the four event nodes may be "less-than-usual meal ingestion", "blood glucose lower than previous one", "regular insulin dose", and "less-than-usual exercise activity", respectively, corresponding to the state at time instant* t*. Each of these*

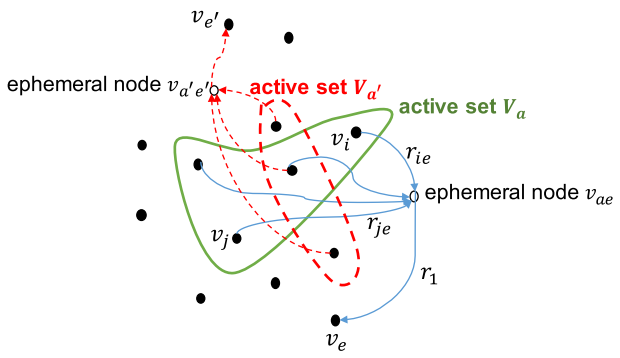

Figure 2: Illustrating an activation graph, where each solid vertex is an event, and each hollow vertex is an ephemeral node. A set of events in the current tuple is called an active set, such as V_a circled in green, which contains 4 events, "less-than-usual meal ingestion", "blood glucose lower than previous one", "regular insulin dose", and "less-than-usual exercise activity". For V_a and target event v_e ("hypoglycemic symptoms"), there is an ephemeral node v_{ae}. v_{ae} links to v_e with relationship r_1, as v_e happens soon after V_a. Moreover, an edge such as the one labeled with relationship r_{ie} indicates that event v_e follows v_i.

four nodes links to the ephemeral node v_{ae}. In turn, v_{ae} links to a target event node in the query, v_e, which is the event "hypoglycemic symptoms" (known to be an event soon after time t in the stream).

Similarly, the three vertices inside the red dashed circle are another active set $V_{a'}$ at a different time instant t', with the corresponding ephemeral node $v_{a'e'}$, for a target event $v_{e'}$ that follows. Note that the two active sets may have vertices in common (one vertex in common here). All edges are incident to ephemeral nodes. Take v_{ae} as an example. There is an edge of relationship type r_1 from v_{ae} to event vertex v_e (hypoglycemic symptoms) indicating that event v_e happens soon after the four events in V_a.

In addition, there are four edges from each event node of V_a to v_{ae}. The relationship type of such edges is based on the two events in the time precedence relationship. For instance, the relationship type of the edge from event v_i (less-than-usual meal ingestion) to ephemeral node v_{ae} is r_{ie} in Figure 2, since one end is event v_i and the other end is event v_e (i.e., event v_e follows event v_i). Likewise, the edge from v_j (blood glucose lower than previous one) to v_{ae} is of relationship type r_{je}.

For ease of reference, Table 1 summarizes the common notations and terms we will be using throughout the paper.

At time t when a tuple (r, t) of the stream arrives, it *activates* a set of event nodes V_t in $\mathcal{G}$. This is the major difference between our graph model and previous graphs or dynamic graphs. Our goal is to predict whether each of the target event nodes will be active soon or much later.

The basic idea of our solution is to perform customized knowledge graph embedding over $\mathcal{G}_\mathcal{S}$. We perform embedding over the nodes and dynamic edges with *attention* [44, 30, 4]. Informally, attentions are weights that can be learned through optimization in training, letting the optimization focus on the most relevant parts of the input to make decisions. For us, the intuition is that every event in an active set attends to a tuple's state behavior over a target event. Different events in an active set may have different *weights* (or importance) in predicting the subsequent target event. Such weights are the *attention* parameters, and can be learned during back-propagation, such as the *stochastic gradient descent* [28] of the training process. An ephemeral node serves as an *intermediate* node to aggregate the active events in the corresponding active set.

Table 1: Notations and terms used in the paper.

Notation	Meaning
$\mathcal{S}$	data stream
$\mathcal{G}_\mathcal{S}$	dynamic activation graph (events) of stream
$e_1, ..., e_k$	target events whose timing may be queried
$e_{k+1}, ..., e_n$	other events not in the target list
r_1	relationship for occurring soon after (gap $< \delta_1$)
r_2	relationship for occurring long after (gap $\geq \delta_2$)
active set	a set of events happening in the current tuple
ephemeral node	a temporary node created between an active set and a target event
r_{xe}	relationship from event v_x in an active set to the ephemeral node leading to target event v_e
a_{xe}	attention weight of relationship r_{xe}
$tarE$	set of triples (*active set, target event*, r_1 or r_2)
$ntarR1$	set of triples (*active set, non-target event*, r_1)
$ntarR2$	set of triples (*active set, non-target event*, r_2)

3.2 Graph Building Algorithm

We first introduce the graph building algorithm. Suppose we have identified n events $e_1, e_2, ..., e_n$ to follow, where the first k ($k \leq n$) events $e_1, ..., e_k$ are the target events that we predict. The main idea is to parse the training data window of stream $\mathcal{S}$ only once and construct $\mathcal{G}_\mathcal{S}$. We show that the algorithm only needs to keep a constant portion of $\mathcal{S}$ in memory while parsing it. The r_1 and r_2 relationship-type edges to target events $e_1, ..., e_k$ are added to $\mathcal{G}_\mathcal{S}$ first—while the active set may contain any events in $e_1, ..., e_n$. We say that these edges are *target edges*.

Knowledge graph embedding is performed over such edges first. If the algorithm is performed in real time and there is still time remaining, the real-time embedding is performed over non-target edges. The intuition is that target edges are directly related to predicting target events, and are hence more relevant for learning the embedding.

Example 3. *Let us go back to the example in Figure 2. Suppose v_e is one of the k target events, then all the five edges incident to the ephemeral node v_{ae} are target edges (e.g., the ones shown of relationship types r_{ie}, r_{je}, and r_1 in Figure 2). Our embedding will be performed over such edges first under time constraints. Now suppose v'_e is not a target event. Then the four red dashed edges incident to the ephemeral node $v_{a'e'}$ in Figure 2 are non-target edges, which have lower priority in the online embedding.*

In order to get non-target edges on demand for embedding after the one pass over the stream, we perform *reservoir sampling* [45] over the r_1 and r_2 non-target edges, respectively, to get a uniformly random pool of such edges for selection as needed.

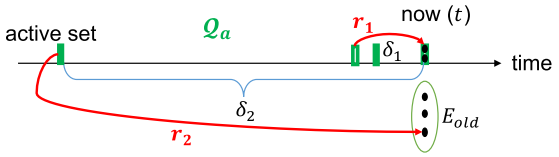

Figure 3: Illustrating the flow of BuildEventOrderGraph.

Figure 3 illustrates the flow of our algorithm that builds the graph. Our goal is to create r_1 and r_2 edges from an active set (tuple) to an event, shown as the red edges in the figure. We maintain a queue of active sets $\mathcal{Q}_a$ from "now" (time t, the current tuple)

back in time up to length δ_2. In addition, we also maintain a set of events E_{old} that has not appeared for at least δ_2 time, shown as the green oval in Figure 3. Then, as soon as an active set in $\mathcal{Q}_a$ is δ_2 old, we remove it from $\mathcal{Q}_a$ and create r_2 edges, one to each event in E_{old}. Furthermore, when the current tuple at time t (now) joins $\mathcal{Q}_a$, we also create r_1 edges from each active set within δ_1 back in time to each event in the current tuple. We present the graph building algorithm in BuildEventOrderGraph.

Algorithm 1:
BuildEventOrderGraph ($\mathcal{S}, e_1, ..., e_k, ..., e_n$)

Input: $\mathcal{S}$: data stream;
$e_1, ..., e_k, ..., e_n$: relevant events, first k are target events
Output: event order dynamic graph $\mathcal{G}_\mathcal{S}$

1 initialize $t_1, ..., t_n$ as last occurrence times of $e_1, ..., e_n$
2 $E_{old} \leftarrow \emptyset$ //set of events that are at least δ_2 old
3 **for** *each tuple* $s[t] \in$ *current window of* $\mathcal{S}$ **do**
4 $E_a[t] \leftarrow s[t] \cap \{e_1, ..., e_n\}$ //get active events
5 **for** *each* $e_i \in E_a[t]$ **do**
6 $t_i \leftarrow t$ //update last-occurrence time
7 $E_{old} \leftarrow E_{old} \setminus \{e_i\}$//remove from old tuple set
8 $\mathcal{Q}_a \leftarrow \mathcal{Q}_a \cup \{E_a[t]\}$ //add $E_a[t]$ into $\mathcal{Q}_a$
9 **for** *each* $e_i \in \{e_1, ..., e_n\} \setminus E_a[t]$ **do**
//check if it is now old
10 **if** $e_i \notin E_{old}$ *and* $t - t_i > \delta_2$ **then**
11 $E_{old} \leftarrow E_{old} \cup \{e_i\}$
12 **for** *each* $E_a[t'] \in \mathcal{Q}_a$ *s.t.* $t - t' > \delta_2$ **do**
13 $\mathcal{Q}_a \leftarrow \mathcal{Q}_a \setminus \{E_a[t']\}$
14 AddR2Edges($\mathcal{G}_\mathcal{S}, E_a[t'], E_{old}$)
15 **for** *each* $E_a[t'] \in \mathcal{Q}_a$ *s.t.* $0 < t - t' < \delta_1$ **do**
16 AddR1Edges($\mathcal{G}_\mathcal{S}, E_a[t'], E_a[t]$)
17 $\mathcal{G}_\mathcal{S}$ is continuously used for embedding

Line 1 of the algorithm initializes, for each of the n events, the last time that it occurs. We will use this information to determine when an event is too old (i.e., it has not appeared in the past δ_2 window), and we will add an r_2 edge from an old active set (δ_2 earlier) to this old event. Line 2 initializes a set that stores such old events. The main loop in lines 3-16 does continuous and incremental parsing of the stream over sliding windows and deposits r_1 and r_2 edge information to the edge pool as illustrated in Figure 3. Line 17 indicates that such a graph (edge pool) is continuously used by the embedding algorithm in the next stage of the pipeline.

In line 4, we get the set of active events that are true in the current tuple. Then lines 5-7 update each of these active events' last occurrence time and remove it from the old set if it is there. In line 8, we add the active set to a queue $\mathcal{Q}_a$. We trim the queue when an event is over δ_2 old. $\mathcal{Q}_a$ is needed for adding an r_2 edge from an old active set (δ_2 earlier) to an old event (that has not appeared for long). Lines 12-14 trim an active set from $\mathcal{Q}_a$ if it is over δ_2 old, and add the corresponding r_2 edges to each old event in E_{old}, as discussed earlier. On the other hand, lines 15-16 add the r_1 edges from each recent active set (within δ_1) to each event in the current tuple.

We next look at the AddR1Edges algorithm (AddR2Edges is similar). Each r_1 edge essentially records a relationship from an active set E_a to an event node e_i. Note that the embedding algorithm presented later will temporally add the ephemeral node v_{ae_i} between E_a and e_i, as well as the ephemeral edges r_{xe_i} from each event node x in E_a to v_{ae_i} (as shown in Figure 2). But for now, if the to-event is a target event (lines 2-3), we only add the triple to the set $tarE$ (target edges); others in lines 5-10 we perform the reservoir sampling [45] so the triple will be put in a fixed-size (*cap*) buffer $ntarR1$ (non-target r_1 edges) uniformly at random. Thus, essentially the graph building algorithm only creates *hyperedges* from an active set (multiple nodes) to a single event node.

Algorithm 2: AddR1Edges ($\mathcal{G}_\mathcal{S}, E_a, E_{to}$)

Input: $\mathcal{G}_\mathcal{S}$: dynamic event order graph stream;
E_a: the active set at "from" end of r_1 edge;
E_{to}: set of event nodes at "to" end of r_1 edge
Output: updated $\mathcal{G}_\mathcal{S}$

1 **for** *each* $e_i \in E_{to}$ **do**
2 **if** $i \leq k$ **then**
//to a target event
3 $\mathcal{G}_\mathcal{S}.tarE \leftarrow \mathcal{G}_\mathcal{S}.tarE \cup \{(E_a, e_i, r_1)\}$ //target edges
4 **else**
5 **if** $count \leq cap$ **then**
//count is total non-target r_1
6 $\mathcal{G}_\mathcal{S}.ntarR1[count] \leftarrow (E_a, e_i, r_1)$
7 **else**
8 $j \leftarrow random(1, count)$
9 **if** $j \leq cap$ **then**
10 $\mathcal{G}_\mathcal{S}.ntarR1[j] \leftarrow (E_a, e_i, r_1)$
11 **return** $\mathcal{G}_\mathcal{S}$

It is not hard to see that the BuildEventOrderGraph algorithm has a per-tuple time complexity of $O(\lambda\delta_1 c + n)$, where λ is the average stream rate (tuples/second), c is the average number of events per tuple, and n is the number of relevant events as input. In particular, the utility algorithm AddR1Edges has a complexity of $O(|E_{to}|)$. Moreover, the space complexity of BuildEventOrderGraph is $O(\lambda\delta_2)$. The time complexity of BuildEventOrderGraph is because we need to add an r_1 edge from each tuple's active set to each of the c events in the $\lambda\delta_1$ tuples that follow, and because for each tuple we need to add one r_2 edge to each of the old events (and there are no more than n of them). The space complexity is due to the fact that we need to maintain a window of size $O(\lambda\delta_2)$ to be able to determine the r_2 edges.

3.3 Training Embedding Vectors and Attention Parameters

3.3.1 The Embedding Algorithm

Having built the event-timing knowledge graph, we now use the data stream data to obtain the graph embedding vectors. Due to the requirement of data stream algorithms (Section 1.1), we extend the efficient TransE [7] knowledge graph embedding algorithm and add ephemeral nodes (with derived embedding vectors), as well as the attention parameters.

The basic idea of this algorithm is quite simple. Recall that the r_1 and r_2 edges we build in the graph are triples (E_a, e, r), from an active set E_a to an event e with relationship r (r_1 or r_2). We iteratively sample such a triple from our triple pool. Recall from Figure 2 that a triple actually consists of two sets of edges—from

each event in E_a to the ephemeral node and from the ephemeral node to e. Our objective (loss) function used for stochastic gradient descent will combine the constraint relationship from these two sets of edges, as well as a negative triple by corrupting either end of the positive triple. We present the algorithm in StateBasedEmbedding.

Algorithm 3: StateBasedEmbedding ($\mathcal{G}_S$)

Input: $\mathcal{G}_S$: dynamic event order graph stream
Output: embedding vectors for nodes and relationship types of $\mathcal{G}_S$

1 **loop**
2 $pool \leftarrow \mathcal{G}_S.tarE$ //first only use target edges
3 $S_{batch} \leftarrow sample(pool, b)$ //draw a mini-batch of size b
4 **for** $(E_a, e, r) \in S_{batch}$ **do**
5 //loop over the original S_{batch}
$(E'_a, e', r) \leftarrow sample(S'_{(E_a,e,r)})$ //sample a corrupted triple
6 $S_{batch} \leftarrow S_{batch} \cup \{(E'_a, e', r)\}$
7 let $f = \sum_{(E_a,e,r)\in S_{batch}} \sigma((E_a, e, r)) \cdot \sum_{i=1}^{d} [\mathbf{e}_i - \mathbf{r}_i - \frac{1}{n} \sum_{x\in E_a} \mathbf{a}_{xe}(\mathbf{x}_i + \mathbf{r}_{\mathbf{xe}_i})]^2$
8 update embeddings and attention **a** w.r.t. gradient of f
9 **while** *time remains* **do**
10 $pool \leftarrow \mathcal{G}_S.tarE \cup \mathcal{G}_S.ntarR1 \cup \mathcal{G}_S.ntarR2$
11 do lines 3-8

In the loop of lines 1-8, we first perform the iterative training over target edges, which, as discussed earlier, have a higher priority as they directly lead to the target nodes to be predicted. Lines 9-11 are essentially the same, but work on all edges. Line 3 samples a mini-batch of triples (for mini-batch stochastic gradient descent). Similar to TransE (and most other translational distance based embedding), line 5 does negative sampling [32] by corrupting either the head or tail of a positive triple in the sample set, and line 6 includes the negative sample in the batch too.

Line 7 has the key loss function of the embedding, where the $\sigma(\cdot)$ function is the sign function that is $+1$ for a positive sample and -1 for a negative sample, and d is the dimensionality of the embedding vectors. This is similar to the L_2-distance version of TransE [7] minimizing $\|\mathbf{t} - \mathbf{r} - \mathbf{h}\|_2$, except that **t** is the event node **e**, and the head **h** is replaced by the embedding of the ephemeral node v_{ae} in Figure 2. In turn, v_{ae} is the tails of the triples from the ephemeral edges such as r_{ie} in Figure 2. Again using the constraint **t = h + r** we derive the embedding of v_{ae} from $\mathbf{x} + \mathbf{r}_{\mathbf{xe}}$, and does a weighted sum of them from each event node of the active set, where the weight is the attention parameter $\mathbf{a}_{xe}$.

In line 8, the algorithm does stochastic gradient descent optimization over each parameter value in each embedding vectors, including those of the event nodes, r_1, r_2, and all ephemeral edges' relationship types r_{ie} (as in Figure 2), as well as all the attention parameters $\mathbf{a}_{xe}$. Like TransE, we normalize each embedding vector to length 1, and normalize the attention parameters such that $\sum_{x\in E_a} \mathbf{a}_{xe} = n$ (where n is the total number of events). We organize the training algorithm iterations into *epochs*, where each epoch has the number of random samples equal to the total number of training records used. Our experiments show that the convergence of the loss function value is quite fast, the details of which are in Section 6.

It is easy to see that the complexity of StateBasedEmbedding is $O(Idc)$, where I is the number of iterations to reach convergence, d is the dimensionality of embedding vectors, and c is the average number of events per tuple. Note that, as we continuously train embedding vectors from one sliding window to the next, the number of iterations I to reach converge of course depends on data—data that changes significantly over time tends to require a greater I, while a more stable stream needs a smaller I for incremental embedding.

3.3.2 Making Prediction

Once we have all the embedding vectors and attention parameter values, making predictions is again based on the same loss function as in line 7, except that we do not need to use a negative sample, and there is only one triple in the sample set S_{batch}. That is, we use the loss function $f = \sum_{i=1}^{d} [\mathbf{e}_i - \mathbf{r}_i - \frac{1}{n} \sum_{x\in E_a} \mathbf{a}_{xe}(\mathbf{x}_i + \mathbf{r}_{\mathbf{xe}_i})]^2$. For example, if the current record has active set E_a, and we want to predict whether a target event e will be more likely to occur soon or to occur much later, we use the loss function above, and the relationship that results in a smaller loss function value wins.

4. LEARNING RELEVANT EVENTS

In the previous section, we assume that we are given the events to use for building the activation graph and for event predictive queries. In this section, we will study how to select the relevant events to use with respect to a set of possible target events to be queried (predicted). These top relevant events are used for continuous dynamic graph building, as well as continuous embedding learning and predictive queries as presented in the previous section.

4.1 Preliminaries on tf-idf

In information retrieval, tf-idf is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus [34]. The tf-idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general. The tf-idf is the *product* of two statistics, term frequency and inverse document frequency. A simple way to define tf is: $\text{tf}(t, d) = 1$ if t occurs in d and 0 otherwise. A common way to define idf is: $\text{idf}(t, D) = \log \frac{|D|}{|\{d\in D|t\in d\}|}$, where $|D|$ is the number of documents in the corpus D, and $|d \in D|t \in d|$ is the number of documents that contain the term t. Thus, a more common term has a smaller idf.

4.2 Learning Algorithm

Basic Ideas and Intuitions. In this subsection, we discuss how to identify "significant" events that help predict target events. The basic idea is that we aim to find characteristic events that tend to precede target events (i.e., within time δ_1, the threshold used to determine relationship r_1), but not so much for other events. We resort to a metric in information retrieval called *tf-idf* (short for term frequency-inverse document frequency) [34]. For our problem, each event candidate is analogous to the "term", while context (tuples prior to the target event) before a target event occurs is analogous to the document we are interested in, and the general context of the whole stream is analogous to the text corpus.

Then there is the computation issue—how do we efficiently find the top events with the highest tf-idf with respect to each target? We first partition the set of values or value range of each attribute

into *basic events*. Intuitively, they are the elementary events in a general event. For instance, a basic event in Example 1 may be "less-than-usual meal ingestion". However, just the basic events themselves may not be discriminative enough, as each basic event alone may be very common in the general stream context. Thus, we also explore the combination of two or more basic events together as a *composite event*. A possible composite event from Example 1 is "less-than-usual meal ingestion" and "blood glucose lower than previous one". A composite event will have lower (or the same) tf (term frequency) in the context of target events than the individual basic events within it, but it will also have higher (or the same) idf (inverse document frequency).

Therefore, we aim to find the top-n events (either basic or composite) that have the highest tf-idf for each target event. While the number of basic events is manageable (typically a constant number of partitions times the number of attributes), there are an exponential number of composite events—it is computationally challenging to compute the tf-idf of all of them by checking the tuples prior to each target event and those prior to each tuple in the stream.

We devise a novel algorithm by using efficient bitmap operations, sampling, and A*-style pruning and bounding [37]. The main ideas are as follows. We build a bitmap $\mathcal{B}_1$ for each basic event e where each bit of $\mathcal{B}_1$ corresponds to one occurrence of the target event (say, at time t), and the bit is 1 if e occurs within time $[t-\delta_1, t)$, i.e., within δ_1 interval prior to the target event, and is 0 otherwise. This will be used to compute the tf part of tf-idf. In the same vein, we build a bitmap $\mathcal{B}_2$ for each basic event e for its occurrence in the general stream context (i.e., every tuple). However, the problem is that there might be too many tuples in the stream, which makes $\mathcal{B}_2$ too large. Thus, we use random sampling which provides a provably accurate estimate of the idf part of tf-idf. We sample the stream tuples and each bit of $\mathcal{B}_2$ corresponds to a tuple that is chosen in the sample. Like $\mathcal{B}_1$, if the chosen tuple arrives at time t, the bit of $\mathcal{B}_2$ is 1 if e occurs within time $[t-\delta_1, t)$, and is 0 otherwise.

The next idea is that we use A*-style aggressive pruning and bounding to efficiently search the space for top-n events in tf-idf. We maintain a priority queue $\mathcal{Q}$, where each element in $\mathcal{Q}$ is an event (basic or composite) along with a weight, which is an optimistic upper bound of its tf-idf value (proven in Theorem 1 below). Each time, we pop out an event with the highest weight from $\mathcal{Q}$, and expand it with another basic event unless it is marked *finalized*. A finalized event e has its weight exactly the same as its tf-idf, and since this value is higher than the upper bounds of the tf-idf of all other events in $\mathcal{Q}$, it should be returned as a top event.

The Algorithm. We now show the algorithm GetTopRelevantEvents. In line 1, we assign an arbitrary but *fixed* order to all basic events. Lines 4-5 build the two bitmaps as discussed above. Lines 6-29 perform the A* search of top-n events with highest tf-idf. Specifically, the loop in lines 7-12 first add each basic event into the priority queue $\mathcal{Q}$. The weights set in lines 9 or 11 are used to order the items in $\mathcal{Q}$, with the root of $\mathcal{Q}$ (to be popped next) having the greatest w. Theorem 1 below shows the reason for the w value, which is an upper bound of the tf-idf value of all events that can be derived from e_j.

Line 15 pops the root of $\mathcal{Q}$ each time for the event with the greatest w. In general, w is an upper bound of tf-idf. The *finalized* flag in line 16 marks that w is actually the exact tf-idf of the corresponding event e_i. Thus, if the condition in line 16 is true, event e_i's tf-idf is higher than every other event's tf-idf upper bound—which means that e_i must have the highest tf-idf among all the events not already in E_r. Then line 17 adds e_i into the set to be returned.

Lines 19-27 expand the current event e_i by one more basic event e_j, where e_j is after all the basic events in e_i according to the order in line 1. Lines 23-26 set the upper bound value w in the same way as lines 8-11. Line 28 updates the w of the already popped out e_i to its exact tf-idf value, and marks it as *finalized* before putting it back into $\mathcal{Q}$. Note that the worse case complexity of GetTopRelevantEvents is still $O(2^b n)$, where b is the number of basic events and n is the number of top events to be extracted. Of course, as any A* algorithms, it has aggressive pruning and is much faster in practice. Theorem 1 below shows the correctness of the algorithm.

Algorithm 4: GetTopRelevantEvents $(\mathcal{S}, e_x, n)$

Input: $\mathcal{S}$: data stream;
e_x: a target event;
n: number of events to retrieve
Output: top n events with the highest tf-idf

1 $E_b \leftarrow$ *a fixed order of basic events in a tuple of* $\mathcal{S}$
2 **while** *one pass of* $\mathcal{S}$ **do**
3 **for** $e_j \in E_b$ **do**
4 build bitmap $\mathcal{B}_1(e_j)$, bit i indicates if e_j occurs in $[t-\delta_1, t)$, where t is the i-th occurrence time of e_x
5 build bitmap $\mathcal{B}_2(e_j)$, bit i indicates if e_j occurs in $[t-\delta_1, t)$, where t is occurrence time of i-th tuple in sample of $\mathcal{S}$
6 initialize priority queue $\mathcal{Q}$ of events
7 **for** $e_j \in E_b$ **do**
8 **if** $|\mathcal{B}_1(e_j)| < \frac{|\mathcal{S}|}{e}$ **then**
9 $w \leftarrow |\mathcal{B}_1(e_j)| \cdot \log \frac{|\mathcal{S}|}{|\mathcal{B}_1(e_j)|}$
10 **else**
11 $w \leftarrow \frac{|\mathcal{S}|}{e} \cdot \log e$
12 add e_j into $\mathcal{Q}$ with weight w
13 $E_r \leftarrow \emptyset$ `//result to be returned`
14 **while** $|E_r| < n$ **do**
15 pop (e_i, w) from $\mathcal{Q}$
16 **if** e_i *is marked finalized* **then**
17 $E_r \leftarrow E_r \cup e_i$
18 **continue**
19 **for** *each* $e_j \in E_b$ *after all basic events in* e_i **do**
20 $e_i' \leftarrow e_i \cap e_j$
21 **if** $|\mathcal{B}_1(e_i')| = 0$ **then**
22 **continue**
23 **if** $|\mathcal{B}_1(e_i')| < \frac{|\mathcal{S}|}{e}$ **then**
24 $w \leftarrow |\mathcal{B}_1(e_i')| \cdot \log \frac{|\mathcal{S}|}{|\mathcal{B}_1(e_i')|}$
25 **else**
26 $w \leftarrow \frac{|\mathcal{S}|}{e} \cdot \log e$
27 add e_i' into $\mathcal{Q}$ with weight w
28 $w \leftarrow |\mathcal{B}_1(e_i)| \cdot \log \frac{N}{|\mathcal{B}_2(e_i)|}$ `//N is number of bits in` $\mathcal{B}_2(e_i)$
29 mark e_i *finalized* and add e_i into $\mathcal{Q}$ with weight w
30 **return** E_r

Theorem 1. *The GetTopRelevantEvents algorithm returns the correct top-n events with the highest tf-idf. In particular, the w bound calculated in lines 23-26 of the algorithm for event e_i' is an upper bound of the tf-idf of all the events that can be derived from e_i' by adding basic events into it.*

Proof. Let the tf value of an event e_i be c, i.e., e_i appears in c places where the target event appears. To ensure the correctness of the A* pruning in the algorithm, its w value must be an upper bound of the tf-idf of all events that can be obtained by extending e_i—we call such events the *descendants* of e_i. Let $1 \leq x \leq c$ be the tf of such a descendant (whose tf can only be smaller than or equal to e_i's). Then $f = x \log \frac{N}{x}$, where N is the total size of the stream, is an upper bound of the tf-idf of this descendant. To get an upper bound among all such descendants, we need to find the maximum value of $f = x \log \frac{N}{x}$ for an integer $1 \leq x \leq c$. By taking the derivative of f over x, we get that if $c < \frac{N}{e}$, the upper bound is just $c \log \frac{N}{c}$; otherwise the upper bound is $\frac{N}{e} \cdot \log e$, where e is the the base of the natural logarithm. This exactly corresponds to the w assignment in lines 23-26. □

Discussions and Remarks. Our model training (Section 3) uses the top relevant events. A natural question is what happens if new events become more relevant after training. This concern is addressed from several aspects. First of all, our learning-events, building graph, and embedding pipeline is continuous and incremental. The top events, graph, and embedding vectors are continuously updated. Secondly, previously unseen events have relations with our identified events. Thus, we do not need to explicitly list every event. In other words, the relational machine learning approach [33] is robust to new events, as a new event has relationships with the old ones (e.g., co-occurrence) and the learned latent features in embedding vectors (of other events) intuitively capture the essence of all events. Finally, there are more expensive deep learning approaches such as graph neural networks [44] that have more generalizability by estimating the embedding of new nodes based on its neighborhood. However, typically such an approach is more computationally expensive and requires significantly larger amounts of training data unsuitable for the requirements of real-time streams (Section 1.1). Further exploration of this problem is beyond the scope of this paper, and we leave it to future work.

5. ANALYSIS OF COMPLEXITY AND BOUNDS

5.1 Preliminaries on Rademacher Complexity

Rademacher complexity [29] is a fundamental concept to study the rate of convergence of a set of sample averages to their expectations. It is at the core of statistical learning theory [41], but its usefulness extends way beyond the learning framework. The Rademacher bounds depend on the training set distribution (unlike VC-dimension based bounds [42] which are data independent), and hence can often give better bounds for specific input distributions. Moreover, it is estimated from the training set, allowing for strong bounds derived from a sample itself.

We consider a finite domain $\mathcal{D}$. Let $\mathcal{F}$ be a family of functions from $\mathcal{D}$ to $[0, 1]$, and let $S = \{s_1, ..., s_n\}$ be a set of n independent samples from $\mathcal{D}$. For each $f \in \mathcal{F}$, define $m_{\mathcal{D}}(f) = \frac{1}{|\mathcal{D}|} \sum_{c \in \mathcal{D}} f(c)$ and $m_S(f) = \frac{1}{n} \sum_{i=1}^{n} f(s_i)$. Some results of Rademacher complexity theory are bounding the maximum deviation of $m_S(f)$ from $m_{\mathcal{D}}(f)$, i.e., $\sup_{f \in \mathcal{F}} |m_S(f) - m_{\mathcal{D}}(f)|$. Specifically, *Rademacher variables* are defined as n independent random variables $\sigma = (\sigma_1, ..., \sigma_n)$ with $\Pr(\sigma_i = -1) = \Pr(\sigma_i = 1) = 1/2$. Then the (empirical) *Rademacher complexity* is defined as $R_{\mathcal{F}}(S) = \mathbf{E}_\sigma[\sup_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \sigma_i f(s_i)]$. A key property of the Rademacher complexity of a set of functions $\mathcal{F}$ is that it bounds the expected maximum error in estimating the mean of any function $f \in \mathcal{F}$ using a sample, as we use in Theorem 2 below.

5.2 Analysis

In this section, we perform some analysis on our prediction algorithm. Our embedding training algorithm and timing prediction use L2 distance as the loss function. Since we always normalize the vectors that we obtain to unit length, it is easy to see that using L2 distance as the objective function is equivalent to using cosine distance/similarity. This is because $\sum_{i=1}^{n}(x_i - y_i)^2 = \sum_{i=1}^{n}(x_i^2 + y_i^2 - 2x_i y_i) = \sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} y_i^2 - 2\sum_{i=1}^{n} x_i \cdot y_i = 1 + 1 - 2\mathbf{x} \cdot \mathbf{y}$. Therefore, the predicted event probability of target event e can be written as:

$$\begin{aligned} p(e) &= \frac{[\frac{1}{n} \sum_{x \in E_a} \mathbf{a}_{xe}(\mathbf{x} + \mathbf{r}_{\mathbf{xe}}) + \mathbf{r}] \cdot \mathbf{e}}{Z} \\ &= \frac{\sum_{i=1}^{d} [\frac{1}{n} \sum_{x \in E_a} \mathbf{a}_{xe}(\mathbf{x}_i + \mathbf{r}_{\mathbf{xe}_i}) + \mathbf{r}_i] \cdot \mathbf{e}_i}{Z} \\ &= \frac{1}{n} \sum_{x \in E_a} \frac{\sum_{i=1}^{d} [\mathbf{a}_{xe}(\mathbf{x}_i + \mathbf{r}_{\mathbf{xe}_i}) + \mathbf{r}_i] \cdot \mathbf{e}_i}{Z} \end{aligned} \quad (1)$$

where Z is a normalization constant (e.g., so that the probabilities to all target events add up to 1). The second equality is to expand the dot product, while the third equality is to swap the two summations.

The basic idea of using Rademacher complexity for our approach is as follows. The *ideal* scenario is if we had used *all the events* $\mathcal{D}$ to build the event order knowledge graph $\mathcal{G}_{\mathcal{S}}$, and then use the embedding vectors to predict event probability following our algorithm. But clearly that would be infeasible. Instead, our BuildEventOrderGraph algorithm uses $n \ll |\mathcal{D}|$ events. It is reasonable to assume that the top relevant events that we pick in Section 4 based on tf-idf are at least as good as uniform random samples. We use Rademacher complexity theory [29] to analyze the accuracy guarantee provided by the n event sample. Let

$$f(x) = \frac{\sum_{i=1}^{d} [\mathbf{a}_{xe}(\mathbf{x}_i + \mathbf{r}_{\mathbf{xe}_i}) + \mathbf{r}_i] \cdot \mathbf{e}_i}{Z} \quad (2)$$

From Equations (1) and (2), $p(e) = \frac{1}{n} \sum_{x \in E_a} f(x)$, and we consider this as the result from a sample S of n points, and name it $m_S(f)$, while the ideal value is $m_{\mathcal{D}}(f) = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} f(x)$, where $\mathcal{D}$ is the set of all events as discussed earlier. Let $\mathcal{F}$ be the family of f functions over all target events. Then we use the Rademacher complexity to bound the error of $m_S(f)$ from $m_{\mathcal{D}}(f)$. Therefore, we have the following main result, where the bounds themselves are from [36].

Theorem 2. *With probability at least* $1 - \epsilon$, $\sup_{f \in \mathcal{F}} |m_S(f) - m_{\mathcal{D}}(f)| \leq 2R_{\mathcal{F}}(S) + \frac{\ln \frac{3}{\epsilon} + \sqrt{(\ln \frac{3}{\epsilon} + 4nR_{\mathcal{F}}(S)) \ln \frac{3}{\epsilon}}}{n} + \sqrt{\frac{\ln \frac{3}{\epsilon}}{2n}}$, *where* $\epsilon \in (0, 1)$ *and* $R_{\mathcal{F}}(S)$ *is the Rademacher complexity of* $\mathcal{F}$ *on* S *satisfying* $R_{\mathcal{F}}(S) \leq \min_{r \in R^+} w(r)$, $w(r) = \frac{1}{r} \ln(\sum_{\boldsymbol{v} \in V_S} \exp[\frac{r^2 \|\boldsymbol{v}\|_2^2}{2n^2}])$, $V_S = \{\boldsymbol{v}_{fS}, f \in \mathcal{F}\}$, *and* $\boldsymbol{v}_{fS} = (f(e_1), ..., f(e_n))$.

Here, the function w is convex and continuous in R^+, and has first and second derivatives everywhere in its domain. Hence it is possible to minimize it efficiently using standard convex optimization methods [8]. In our experiments, we simply implement gradient descent to get $\min_{r \in R^+} w(r)$. The $f(e_i)$, for $1 \leq i \leq n$, in Theorem 2 is from Equation (2), where $e_1, ..., e_n$ are the n event nodes. The function family $\mathcal{F}$ is for predicting each of the k target entities. A final remark is that this is a data dependent bound—the $f(e_i)$ values are based on the actual data and prediction results. We will further examine the bounds, as well as the time taken to obtain them, using real-world datasets in the experiment section next.

6. EXPERIMENTAL EVALUATION

6.1 Datasets and Setup

We use the following real world datasets in four different domains, namely biomedicine, mobile systems, the petroleum industry, and transportation: **(1) Diabetes data.** This is the AIM-94 dataset provided by Michael Kahn, MD, PhD, Washington University in St. Louis [15]. It records diabetes patients' event logs (multi-attribute streams) for about 5 to 9 months including insulin dose (regular, NPH, and UltraLente), blood glucose at various times, hypoglycemic symptoms, meal ingestion amount, and exercise activity amount. **(2) Mobile system data.** This dataset is collected and used by Banerjee et al. in an ACM UbiComp paper [5]. The data contains traces of battery usage data for laptops. In addition to battery usage, the multi-attribute streams also contain data on CPU utilization, disk space, on-AC status, Internet connectivity, and idle time (based on keyboard events) for the laptop users. **(3) Oil well data.** This dataset from Brazil [17, 43] is as introduced in Section 1. Prediction of undesirable events in oil wells is critical. It records various measurement events in oil wells, as well as a number of undesirable real events. The measurement events include pressure events at Permanent Downhole Gauge (PDG), temperature events at Temperature and Pressure Transducer (TPT), pressure events at TPT, among many others. The dataset is over 5 GB and has about 1 tuple per second. **(4) NY taxi data.** The trip data of this dataset is about 30 GB, containing the information of all taxi trips in the New York City in 2013 [16]. It has 14 attributes, including medallion, hack license, vendor ID, pick-up date/time, drop-off date/time, pick-up longitude/latitude, drop-off longitude/latitude, trip time, and trip distance.

We implement all the algorithms presented in this paper in Java. In particular, we extend the code of TransE [7] to handle active states, ephemeral nodes, and the attention mechanism. In addition, we have also implemented three most relevant baseline methods for comparisons: (1) sequential association rule mining, the Generalized Sequential Pattern (GSP) algorithm [40], (2) event prediction with Bayesian and Bloom filters in ICPE'13 [46], and (3) kernel-SVM [11], marked as SAR, ICPE, and KSVM in our upcoming figures, respectively. The experiments are performed on a MacBook Pro machine with OS X version 10.11.4, a 2.5 GHz Intel Core i7 processor, a 16 GB 1600 MHz DDR3 memory, and a Macintosh hard disk.

6.2 Experimental Results

6.2.1 Learning Top Relevant Events

Based on the attributes of each of the two datasets, we define a set of *basic events*, as well as a set of *target events*. For the diabetes data, there are 16 basic events, each of which is out of a single attribute, including the second (or more) insulin injection of the day, a significantly increased blood glucose measurement, hypoglycemic symptoms, more-than-usual meal ingestion, and less-than-usual exercise activity. Out of the basic events, we define the target events that a user may be interested in, such as a blood glucose measurement significantly higher (or lower) than the most recent measurements, and the hypoglycemic symptoms.

Likewise, for the mobile system dataset, there are 16 basic events on individual attributes, including battery being near-empty, being connected to the Internet, and so on. We also define target events such as CPU usage of 95% or more, and idle time of at least 20 seconds. For the diabetes data, we set δ_1 to be 6 hours and δ_2 to be 80 hours, while for the mobile system application, these two time intervals should be much shorter to be useful, and we set δ_1 to be 10 minutes and δ_2 to be 1 hour.

For the oil well data, we define the increases and decreases of each measurement attribute as basic events, and each type of undesirable events as target events. For the NY taxi data, we partition the latitude and longitude ranges of NYC into 8-by-8 grids. As mentioned earlier, we define the ratio between trip time and trip distance as the trip's *delay score*. We then define the target events as the average delay score of all trips within 5 minutes at a grid area that we are interested in is above (or below) a threshold—top (or bottom) 1/4 of the whole history at that area. However, delays scores are expensive to obtain, as they require the statistics of all trips going into or coming out of an area. The idea is to use easy-to-observe simple statistics of some grid areas (other than the target areas), such as the incoming/outgoing taxi counts within 5 minutes. These events will help us predict he target events. For both oil well and NY taxi datasets, we set the δ_1 of 1/3 of the target events to be 5 minutes, 1/3 to be 10 minutes, and 1/3 to be 15 minutes. Furthermore, we set the δ_2 of the oil well data to be 3 hours, and the δ_2 of the NY taxi data to be 1 hour. In this section, unless otherwise specified, we set the default number of relevant events n to be 150 for the diabetes and mobile system data, 300 for the oil well data, and 400 for the NY taxi data.

In the first set of experiments, we evaluate the GetTopRelevantEvents algorithm that retrieves the most relevant events (w.r.t. the target events) which are any possible combinations of the basic events. Since our algorithm is a one-pass stream algorithm, to evaluate the processing speed, we adopt the conventional approach of measuring the *throughput* of the algorithm, i.e., how many stream tuples it can handle per second.

We first run the GetTopRelevantEvents algorithm using the diabetes dataset and varying the number of target events. The results are shown in Figure 4 for the mobile system dataset, and in Figure 5 for the oil well dataset (the results of other two datasets show similar trends and are omitted). In order to understand the impact of the A* search to performance, we also run a version of the algorithm skipping the A* search part only. We can see that A* search slightly decreases the throughput by a small percentage, for both datasets. Moreover, the throughput slightly decreases as the number of target events increases. This is because the algorithm needs to proportionally handle more events and candidate intermediate events.

Then we examine the distribution of the discovered top relevant events, in what we call the *cardinality*, which is the number of basic events that a discovered relevant event comprises. The cardinality distribution of the top events is shown in Figure 6 for the diabetes dataset, and in Figure 7 for the mobile system dataset. We can see that, for the diabetes data, the top relevant events have the highest fraction of cardinality 2 (the next one is 3), while cardinality 4 has the highest fraction for the mobile system data.

The above result reflects a tradeoff in the cardinality of a relevant event. Having too few basic events gives a higher "tf" part of the tf-idf, since individual basic events are more likely to appear in the context of target events; however, that would also result in a lower idf as it also appears frequently in the general stream context. In the other extreme, a very-high-cardinality event will have a greater idf but a very low tf.

6.2.2 Continuous Building of Event-Order Graphs

In the next set of experiments, we examine the performance of building event-order graphs. Our algorithm BuildEventOrderGraph is again a one-pass stream algorithm, and we use throughput to uniformly measure the performance. The results are shown

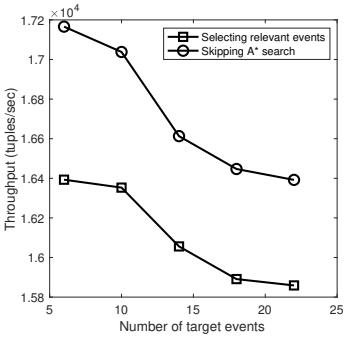

Fig 4 Learning events (mobile)

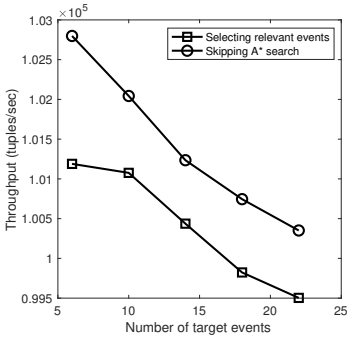

Fig 5 Learning events (oil well)

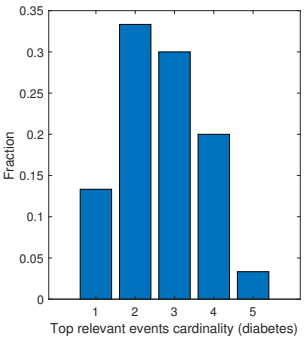

Fig 6 Event cardinality (diabetes)

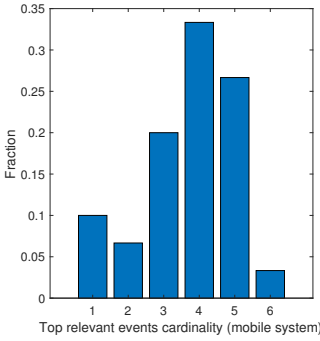

Fig 7 Event cardinality (mobile)

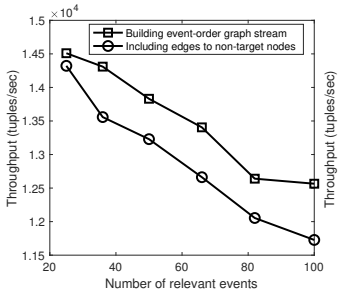

Fig 8 Building graph (diabetes)

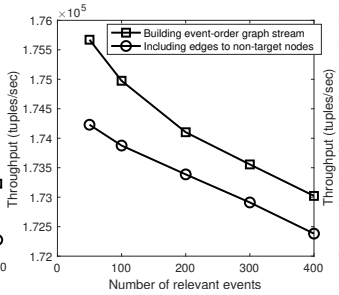

Fig 9 Building graph (NY taxi)

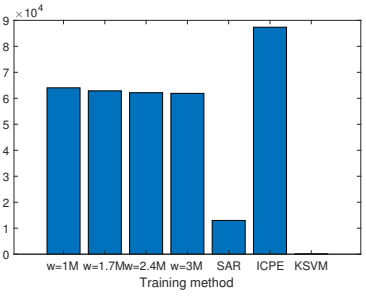

Fig 10 Training model (oil well)

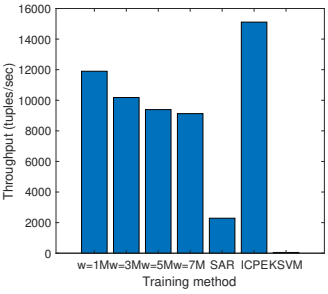

Fig 11 Training model (NY taxi)

in Figure 8 for the diabetes data and Figure 9 for the NY taxi data. Recall that BuildEventOrderGraph does not include all edges that lead to *non-target* nodes, but keeps a reservoir sample of them. We compare the performance with a variant of the algorithm that includes in the graph all edges to the non-target nodes as well.

Figures 8 and 9 show that the performance slightly decreases as the number of relevant events increases. However, since we are dealing with the top events that are relevant to the target events, the number of such events does not need to be high to achieve an equivalent prediction accuracy, as found in our subsequent experiments. Including all edges to non-target nodes also slightly decreases the performance for building the graph, compared to discarding many such edges but only keep a uniformly random sample as in the reservoir sampling. Another interesting fact is that the throughput with the NY taxi data is in general higher than the diabetes data. This is because we aggregate the tuples of every five minutes to obtain the events of the NY taxi data, and hence the overall throughput is higher.

6.2.3 *Embedding Training and Overall System Throughput*

Our next set of experiments is concerned with a key step, which is to train the embedding vectors and the attention parameters of our event-order graph. We first show the throughput performance of the training, as shown in Figure 10 for the oil well data and Figure 11 for the NY taxi data. Recall that our pipeline of building dynamic graph and learning embedding is dynamic and incremental over each sliding window of size w. For training, this means that we keep sampling *(active set, r_1 or r_2, target event)* triples (and the associated negative samples) from the current window and performing stochastic gradient descent (SGD) until convergence. Thus, we measure the throughput of training with varying window sizes, ranging from 1M tuples to 3M tuples per window for the oil well dataset and 1M to 7M tuples for the NY taxi dataset. Later on we will also examine the impact of window size on prediction accuracy.

In addition, we also examine the training speed of the three baseline methods, SAR, ICPE, and KSVM. Note that, unlike our method which incrementally updates the embedding over sliding windows, the training methods of baseline ones are not continuous but work in batch—but we still report them in the form of throughputs for comparison.

Figures 10 and 11 show that the training throughput slightly decreases as we increase window size w (but still remains high). This is because, as mentioned above, our incremental embedding samples the triples in the current window and performs SGD until convergence. Increasing w may slightly decrease convergence speed because more tuples will likely exhibit more variable latent features; however, this variability tends to be less as we further increase the window size as it approaches more global stability. In fact, as shown in the experiments later, a larger window size does not necessarily translate to better prediction accuracy after some point.

Among the three baseline methods, sequential association rule mining (SAR) is slower than our method, while ICPE is faster and KSVM is the slowest in training. ICPE is faster due to its simplistic data structures and algorithms; however, as shown later, its prediction accuracy is the worst. Furthermore, off-the-shelf classification methods such as ICPE and KSVM are not designed for predicting the timing of *future* events in a future tuple—instead, they are designed for predicting the unknown class attribute in the *current* tuple. To use ICPE or KSVM, we have to couple the current tuple with the r_1 or r_2 relationship to a target event in a *future* tuple.

Note that, as shown in Figure 18 later, the actual prediction using the learned embedding vectors only incurs simple calculation and has a negligible cost—less than 10 microseconds per prediction. Since continuous training is the bottleneck of the stream

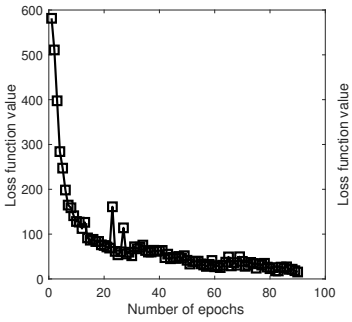

Fig 12 Loss function (diabetes)

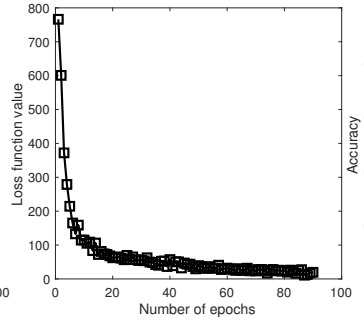

Fig 13 Loss function (mobile)

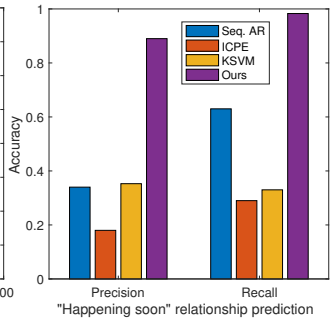

Fig 14 Accuracy (mobile, r_1)

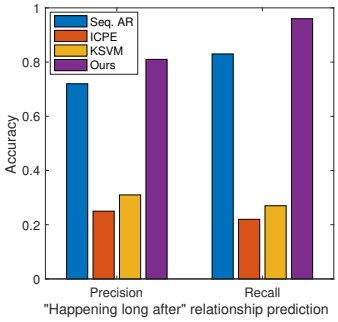

Fig 15 Accuracy (mobile, r_2)

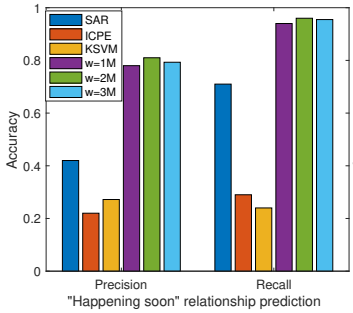

Fig 16 Accuracy (oil well, r_1)

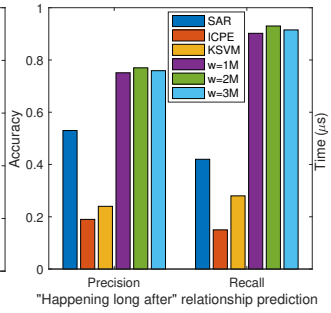

Fig 17 Accuracy (oil well, r_2)

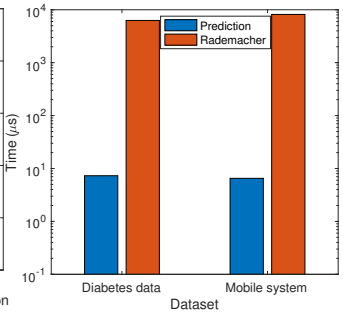

Fig 18 Prediction and bounds

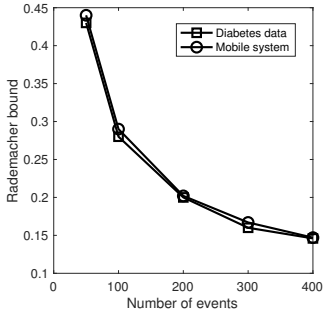

Fig 19 Rademacher bounds

processing pipeline, we find that the overall system throughput for graph building, embedding training, and event prediction is about the same as that from embedding training, which ranges from over one thousand to sixty thousand tuples per second for the four datasets. Such high efficiency makes possible preventive and predictive interventions, as well as predictive complex event processing (e.g., under resource constraints) [22, 20, 31], as discussed in Section 1.

We then look into the *loss function* values (i.e., the f function values in line 7 of StateBasedEmbedding) after each *epoch*, which is defined as a uniform random sample of edges of the same size as the total number of edges in the graph, combined with an equal number of negative sample edges, following the terminology in the TransE algorithm [7] that we adopt and extend. We show the results in Figure 12 for the diabetes data and Figure 13 for the mobile system data (the other two datasets show a similar convergence).

From Figures 12 and 13, we can see that, interestingly, the loss function value sharply decreases after each of the initial epochs of the embedding, where we perform stochastic gradient descent optimization over each value in the embedding vectors and the attention values for each ephemeral edge between an event node in the active set and a target event node. After 40 to 50 epochs, the loss function values in both figures level off, and converge at the lowest by around 80 to 90 epochs. The convergence is slightly faster with the mobile system dataset.

6.2.4 *Predictive Query Accuracy*

After observing the training process, we next examine the event prediction accuracy. The prediction tests are run over the period after the current window. We predict both the r_1 relationship ("happening soon") and the r_2 relationship ("happening long after"). We show the results in Figures 14 and 15 (for r_1 and r_2, respectively) for the mobile dataset, and in Figures 16 and 17 for the oil well dataset.

As our work is to predict discrete events over multiple-attribute data streams in real time, the closest previous work is mining sequential association rules [40] and using the rules to predict. We first mine all the rules that have the target events (to be predicted) on the right hand side, for both r_1 and r_2. Then for a target event e to be predicted using an active set A of events, we identify all the rules that have e on the right hand side, and compute the similarity between its left hand side and the active set A—combining this and the confidence of the rules gives us the prediction of r_1 or r_2. As discussed earlier, ICPE and KSVM are not designed for predicting the timing of *future* events (in a future tuple), but for predicting the unknown class attribute in the *current* tuple. To use ICPE and KSVM, we have to extend the current stream tuple with its r_1 or r_2 relationship with a target event in a *future* tuple, treating it as the class attribute. We need to add one class attribute for each target-event and r_1 or r_2 combination.

For accuracy, we measure both *precision* and *recall* [40]. We first parse the test data (a window of stream after the training period) and for each distinct active set of events, we record the set of target events that have r_1 or r_2 relationships with it. Using this as the ground truth, we calculate the precision and recall values when using our trained embedding vectors and attention values for prediction. First, Figures 14 and 15 over the mobile dataset show that our method is much more accurate than the three baseline methods for this problem, achieving good precision and recall values ranging from around 0.8 to nearly 1. ICPE and KSVM are not designed for this future event timing prediction problem, and they do not capture very well the timing relationships of event co-occurrence, following by a short interval, and following by a large interval. The accuracy of ICPE is the worst due to its simplistic data structures and algorithms.

In Figures 16 and 17 with the oil well dataset, we further show the impact of sliding window size on prediction accuracy. We find

that, while initial increase of window size can improve precision and recall accuracy, further increase beyond a certain point actually decreases the accuracy. This is because the more recent data is more accurate for training the *current* model; data more ancient in the history may distract the training process in learning the current latent features.

Last but not least, we observe that, interestingly, the recall accuracy values are in general slightly higher than the precision values, for both r_1 and r_2 relationships. The reason is as follows. Just as knowledge graphs are generally *incomplete* [33], what we observe in the test window as "ground truth" data is generally incomplete (i.e., the "fact" just has not happened yet). Therefore, our method's *recall* value is relatively higher, since we (almost always) correctly tell that those relationships in the ground truth should be there. But since the "ground truth" (test data window) may miss some real targets for a relationship, while those targets may be correctly returned by our model, the *precision* of our method using the "ground truth" slightly suffers.

6.2.5 Training Data Dependent Complexity Bounds

Finally, we implement the computation of the data-dependent accuracy bounds using Rademacher complexity as in Theorem 2 (which we call Rademacher bounds here), where we set the ϵ parameter to be 0.05. This involves solving a convex optimization problem to get $\min_{r \in R^+} w(r)$. We implement gradient descent for that purpose. In Figure 18, we show the overhead of making a prediction (as done for Figures 14–17) and that of computing the Rademacher bound side by side for diabetes and mobile system data (the other two datasets have very similar results). We can see that making a prediction is very fast (as the model of embedding is already trained), in a few *microseconds*, while computing the Rademacher bound is longer (mostly due to the gradient descent optimization)—but it is still only a few *milliseconds*. In Figure 19, we show the computed average Rademacher bounds as a function of the number of event nodes used. The error bound decreases as the sample size increases, and the average bounds from the two datasets are close. Note that these bounds are theoretical guarantees and tend to be more conservative. As shown earlier, in practice, we often get better accuracy. Moreover, the bound here is the error in predicting the exact probability. In practice, it often suffices to make relative judgements, e.g., which target event between the two is more likely to happen soon, or will this event more likely happen soon or long after now, which can be based on exact probabilities but are more robust to exact-probability errors.

6.3 Summary of Results

The experimental results in this section show that learning the top relevant events for a set of target events using A* search is quite efficient, and the cardinality (number of basic events) of the top events typically ranges from 2 to 5. We have also evaluated the efficiency of building the event-order graphs and of training the embedding vectors and attention values. We have clearly observed the fast convergence of the loss function value after around 50 epochs during the training. The overall system throughput for graph building, embedding training, and event prediction ranges from over one thousand to sixty thousand tuples per second for the four datasets. Using our trained model for event timing relationship (r_1 or r_2) prediction is generally accurate, with precision and recall values ranging from around 0.7 to nearly 1, much higher than those of the three baseline methods, some of which are off-the-shelf classification methods not specifically designed for our future event timing prediction problem. We have also computed the Rademacher bounds with real data.

7. OTHER RELATED WORK

Most closely related work has been discussed inline above. We survey other related work here.

Time series forecasting. Our work bears some similarity with time series forecasting. [18] is a comprehensive review of this research over the past years. [9] and [19] present additional work. We, however, are not dealing with single numerical attribute time series forecasting. We focus on discrete events over multiple attributes in data streams, and their timing relationship modeling and predictions.

Data streams and event processing. Event matching and complex event processing have been well studied in both research and the industry, and is used by practitioners (e.g., [1, 2, 47]). They typically extend the regular expression syntax to define a complex event that is a sequence of simple events. However, this line of work does not deal with predicting future events or predicting the timing relationships among events.

Classification methods. There are a number of off-the-shelf classification methods, such as kernel machines and SVM [11, 27, 38], succinct data structures for stream classification [46], and deep learning [24]. We have compared with some of them in detail in the experiments. Off-the-shelf classification methods are not designed for our problem. Their model is to predict the class attribute of a stream tuple given its other attributes that are observed. As discussed in Section 6, we could use it to solve our problem by extending each tuple with one class attribute for each target event–r_1/r_2 combination. However, this does not perform as well as our relational machine learning approach (knowledge graph embedding) [33]. This is because relational machine learning creates and models new relationships r_1 and r_2 that weave the intricate connections among the selected events, as well as their co-occurrence relationship in tuples. The embedding training learns latent features based on this network of entities and relationships that are not only efficient but also powerful. As discussed in Section 1.1, deep layers of neural networks [24] require a large amount of training data and heavy resources including GPUs and long training time, unfit for fast real-time adaptive stream computing.

Complexity analysis for learning models. VC dimensions and Rademacher complexity have been applied to study the complexity and accuracy guarantees of machine learning algorithms [6]. For instance, Riondato and Upfal [35] apply Rademacher complexity to analyze a sampling based algorithm to compute and approximate node centralities in a very large graph.

8. CONCLUSION

In this paper, we propose a novel approach to predict event timing information in multi-attribute data streams. We represent the event timing knowledge in training data using a knowledge graph where nodes are events, and edges encode timing relationships such as "happening soon after" and "happening long after". On top of this knowledge graph, we need the notion of "active state" to characterize the state information, as well as ephemeral nodes and edges along with attention parameters for learning the embedding vectors containing latent features. With this novel embedding, we are able to achieve high precision and recall values in predicting event timing, ranging from about 0.7 to nearly 1, significantly outperforming baseline approaches.

Acknowledgments. This work is supported in part by NSF grant IIS-1633271.

9. REFERENCES

[1] Oracle complex event processing: Lightweight modular application event stream processing in the real world. *An Oracle White Paper*, 2009.

[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM.

[3] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015.

[5] N. Banerjee, A. Rahmati, and M. D. Corner1. Users and batteries: Interactions and adaptive energy management in mobile systems. In *ACM UbiComp*, 2007.

[6] P. L. Bartlett and S. Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *J. Mach. Learn. Res.*, 3:463–482, Mar. 2003.

[7] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems 26*. Curran Associates, Inc., 2013.

[8] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[9] P. Brockwell and R. Davis. *Introduction to Time Series and Forecasting*. Springer, 2002.

[10] H. Cai, V. W. Zheng, and K. C. Chang. A comprehensive survey of graph embedding: Problems, techniques and applications. *CoRR*, abs/1709.07604, 2017.

[11] T. N. Chan, M. L. Yiu, and L. H. U. KARL: fast kernel aggregation queries. *ICDE*, 2019.

[12] Streambase systems, 2019. Available at `https://www.crunchbase.com/organization/streambase-systems`.

[13] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[14] P. Cui, X. Wang, J. Pei, and W. Zhu. A survey on network embedding. *IEEE Transactions on Knowledge and Data Engineering*, 2017.

[15] Diabetes data, 2019. Available at `http://archive.ics.uci.edu/ml/datasets/Diabetes`.

[16] New york taxi data, 2013. Available at `http://chriswhong.com/open-data/foil_nyc_taxi/`.

[17] Oil wells dataset, 2020. Available at `https://github.com/ricardovvargas/3w_dataset`.

[18] J. G. De Gooijer and R. Hyndman. 25 years of IIF time series forecasting: A selective review. *SSRN Electronic Journal*, 2005.

[19] R. Eubank. *A Kalman filter primer*. Chapman and Hall, 2005.

[20] L. Fulop, A. Beszedes, G. Toth, H. Demeter, L. Vidacs, and L. Farkas. Predictive complex event processing: a conceptual framework for combining complex event processing and predictive analytics. *Proceedings of the Fifth Balkan Conference in Informatics*, 2012.

[21] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, 2015.

[22] S. Gillani, A. Kammoun, K. Singh, J. Subercaze, C. Gravier, J. Fayolle, and F. Lafores. Pi-cep: Predictive complex event processing using range queries over historical pattern space. *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2017.

[23] A. E. Goebel-Fabbri, N. Uplinger, S. Gerken, D. Mangham, A. Criego, and C. Parkin. Outpatient management of eating disorders in type 1 diabetes. *Diabetes Spectrum*, 22(3):147–152, 2009.

[24] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[25] Google. Google inside search. `https://www.google.com/intl/en_us/insidesearch/features/search/knowledge.html`, 2018.

[26] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *CoRR*, abs/1709.05584, 2017.

[27] A. Kampouraki, G. Manis, and C. Nikou. Heartbeat time series classification with support vector machines. *IEEE Transactions on Information Technology in Biomedicine*, 13:512–518, 2009.

[28] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[29] V. Koltchinskii. Rademacher penalties and structural risk minimization. *IEEE Transactions on Information Theory*, 47, 2001.

[30] J. B. Lee, R. A. Rossi, S. Kim, N. K. Ahmed, and E. Koh. Attention models in graphs: A survey. *CoRR*, 2018.

[31] Z. Li and T. Ge. History is a mirror to the future: Best-effort approximate complex event matching with insufficient resources. *PVLDB*, 10(4):397–408, 2016.

[32] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, 2013.

[33] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104, 2016.

[34] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[35] M. Riondato and E. Upfal. VC-dimension and Rademacher Averages: From Statistical Learning Theory to Sampling Algorithms. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*, 2015.

[36] M. Riondato and E. Upfal. ABRA: Approximating betweenness centrality in static and dynamic graphs with Rademacher averages. *ACM Transactions on Knowledge Discovery from Data*, 2018.

[37] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (3rd Ed.)*. Prentice Hall, 2009.

[38] B. Scholkopf and A. J. Smola. *Learning with Kernels: support vector machines, regularization, optimization, and beyond*. MIT Press, 2002.

[39] A. Singhal. Introducing the knowledge graph: Things, not strings. *Google Official Blog.*, 2012.

[40] P.-N. Tan, M. Steinbach, A. Karpatne, and V. Kumar. *Introduction to Data Mining*. Pearson, 2nd edition, 2018.

[41] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, 1999.

[42] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and Its Applications*, 16(2):264, 1971.

[43] R. E. V. Vargas, C. J. Munaro, P. M. Ciarelli, A. G. Medeiros, B. G. do Amaral, D. C. Barrionuevo, J. C. D. de Arajo, J. L. Ribeiro, and L. P. MagalhQues. A realistic and public dataset with rare undesirable real events in oil wells. *Journal of Petroleum Science and Engineering*, 181:106223, 2019.

[44] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *ICLR*, 2018.

[45] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, pages 37–57, Mar. 1985.

[46] M. Wang, V. Holub, J. Murphy, and P. OSullivan. Stream based event prediction using Bayesian and Bloom filters. *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 13)*, 2013.

[47] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with fpgas. *PVLDB*, 3(1-2):660–669, 2010.