Sampling RTB transactions in an online machine learning setting

Carlos Pita (carlos@jampp.com)

Jampp

Abstract. We (the machine learning team at Jampp) strive to predict click-through rates (CTR) and conversion rates (CVR) for the real-time bidding (RTB) online advertising market by means of an in-house online machine learning platform based on a state-of-the-art stochastic gradient descent estimator. Our estimation framework has already been covered in a previous paper, so here we want to focus on some peripheral aspects of our platform that, in spite of being of a somewhat ancillary nature, nevertheless tend to dominate development efforts and overall system complexity; namely, in order to feed the learning system we first need to sample a very high-volume stream of out-of-order and scattered-in-time events and consolidate them into a sequence of observations representing the underlying market transactions, each observation composed of a set of features and a response, from which the estimator is ultimately able to learn. This paper is written in a down-to-earth fashion: we describe a number of particular difficulties the general problem of sampling in an online high-volume setting poses and then we present our concrete answers to those difficulties based on real, hands-on, experience.

Introduction

Jampp is a demand-side platform (DSP) that allows companies to promote their mobile applications by leveraging real-time bidding (RTB) technologies in order to buy digital advertising from multiple inventory sources and exchanges. During an RTB transaction, an auction is announced and any interested bidder has to answer with a bid price within a time constraint of about 100 milliseconds. The bidder that wins the auction pays the second-highest price and obtains the right to display an ad (the act of displaying the ad or banner is called an impression) on a publisher site. If the banner is clicked by the user further goal events might be tracked, *v.g.* the application may be installed or opened. Obtaining a goal is called a conversion; our main goal is that the user installs, so the typical conversion rate we care about is the probability that the user installs given that we got the impression.

More generally, as the machine learning team in a DSP we attempt to estimate the probability of occurrence of some kind of response event (`click`, `install`, ...) conditional to information codified as a set of features extracted from earlier events (`auction`, `bid`, `impression`) belonging to the same market transaction. For example, given that a transaction got an `impression`, we

want to build an observation including rich information taken mainly from the `auction` event of that transaction and having a 0–1 response showing whether the `install` event occurred or not; this observation will then be fed to the learning system[1]. Simply put, an RTB transaction can be seen as a timeline along which some prefix of the following sequence of events takes place in order: `auction`, `bid`, `impression`, `click`, `install`, ... For each kind of response event, we want to be able to sample and build an observation (a set of features plus a binary response) representing a market transaction, in order to feed the learning system. From (i) our estimate of the probability of occurrence of some goal event, (ii) the value of this goal event to our customer, and (iii) a desired expected profit margin, our bidding system is then able to compute a bid price.

Unfortunately, what we actually see is a raw, high-volume, stream of scattered and disordered market events coming from disparate sources (hundreds of bidding processes, a number of tracking platforms, internal enrichment subsystems, etc.). Even at this early stage of the exposition a number of difficulties are already evident:

1. How do we connect our sampler to such heterogeneous sources of events?
2. How do we consolidate different events arriving at different moments in time into what is, conceptually, the same market transaction?
3. Given the sheer amount of streamed market events:
   (a) How do we random sample a fraction of the total market transactions, in case we were not able to process the total amount?
   (b) How do we shard our samplers in order to distribute the load among some convenient partition of the total market transactions?
   (c) How do we avoid relatively rare but informative events (`click`, `install`, ...) from starving, while a flood of high frequency but less informative events (`auction`, `bid`, `impression`, ...) monopolizes sampler resources?

In what follows we will be addressing these questions, first from the perspective of organizing and consuming the raw event stream, then from the perspective of waiting for the sequence of events belonging to the same market transaction. Please keep in mind that both aspects of the problem are closely related, so the distinction is not as clear-cut as we would like for exposition purposes.

Streaming events

An important part of our answers to the above questions relates to the specific use we make of the ZeroMQ (zmq[8]) distributed messaging library, in particular

[1] In our case, this learning system is implemented as a Stochastic gradient descent (SGD[2]) optimizer which continuously updates a massive logistic model running-estimate. Our SGD estimator leverages state-of-the-art techniques like Adaptive learning rates (Adagrad[3]) and Elastic net regularization, implemented within the Follow the proximally regularized leader (FTRL-P[10]) framework. We have already focused on the structure of the RTB market and the details of our estimation framework in a previous work[1].

of its publisher-subscriber (pubsub[7]) pattern implementation. For starters, zmq pubsub allows many subscribers to listen to a set of prefixes of topics with which a publisher "tags" the messages it posts. Each publisher and each subscriber has its own message queue. Topics with prefixes that are not being listened by any subscriber are filtered out at the publisher side, thus sparing precious network resources. Having said that about zmq pubsub, we are ready to present our approach:

1. Each sampler subscribes to a number of sources. After discovering these sources, all the sampler needs to know is that it can subscribe to some set of desired topics. Which sources publish which topics is completely irrelevant for the subscriber, since it just sees a unified stream of messages tagged with topics for prefixes it has subscribed to.
2. Messages for events belonging to the same market transaction are tagged with a transaction identifier topic. The transaction identifier is represented as a hexadecimal number, so that a subscriber listening to a prefix of this number (say `ab`) will receive all events in every transaction which identifier starts with that prefix[2].
3. (a) The assignment of transaction ids to transactions is done in a random way. Thus, subscribing to the prefix `a` will have the effect of random sampling a fraction of about 1/16 of the total market transactions. Similarly, subscribing to prefixes `a` and `01` implies a sampling rate of about $1/16 + 1/256$. It's easy to generalize this pattern to implement simple random sampling with arbitrary sampling rates up to some precision.[3][4]
   (b) Moreover, the random assignment of transaction ids makes the above mechanism well suited for sharding: just subscribe each shard to mutually exclusive transaction id prefixes. For example, we may have four shards respectively subscribed to prefixes $\{0, 1, 2, 3\}$, $\{4, 5, 6, 7\}$, $\{8, 9, a, b\}$ and $\{c, d, e, f\}$.
   (c) Up to this point we have glossed over the fact that our topics also include an event type identifier besides the transaction id. The full topic format is `<event-type>-<transaction-id>`, where `<event-type>` may be one of `auction`, `bid`, `impression`, `click`, `install`, ... Remembering that each subscriber gets its own message queue, a simple strategy to keep high-frequency events (`auction`, `bid`, `impression`) from displacing low-frequency but very informative events (`click`, `install`, ...), consists of connecting two different subscribers, one subscribed to prefixes `auction`, `bid`, `impression` and the other to prefixes `click`, `install`, ..., and then

[2] As we have advanced in the introduction there is another aspect to this problem, namely to determine how and how much to wait for each kind of event. We defer the detailed treatment of this diachronous aspect until the next section.

[3] Moreover, the topic pattern described in answer (3c) allows for the implementation of stratified sampling, with event types being the strata.

[4] We also point out that zmq implementation of prefix matching is based on a trie that is able to cope with a big number of subscription prefixes without imposing considerable overhead.

reading from them in a round-robin fashion (for which we take advantage of zmq fair-queueing inbuilt mechanism).

To sum up, we might say that zmq pubsub protocol implementation provides us with a building block that (at least partially) addresses several non-trivial problems in a pretty straightforward way. That's not to say that it all amounts to a more or less out-of-the-box usage of the library: since it is very low-level and bare-bones tool, it still puts considerable burden on the application side. Nevertheless it offers a very general building block that can be flexibly accommodated to demanding scenarios, as our use-case testifies.

Waiting for events

We return now to question 2 above to address its temporal dimension. Since the learning system must be fed with observations containing both a set of features and a response, the need arises to wait for this response once we have collected the features. The features are extracted from earlier events (`auction`, `bid`, `impression`) in the transaction, while the response comes from events arriving hours or even days later (`click`, `install`, ...). This implies that we have to establish attribution or waiting windows for the different types of response events; this also implies that we have to keep a buffer of ongoing transactions so that, when a waiting window expires for some event in some transaction, we may emit an observation with the appropriate response to be consumed by the learning system. Notice that this buffer contains ongoing transactions currently waiting for one or more kinds of events, and that to each of these event kinds corresponds a different attribution window. Thus, at a given moment and for any kind of event, we must be able to retrieve from the buffer all those transactions for which the attribution window just expired, then construct and emit the corresponding observation and, finally, somehow mark the retrieved transactions as "done" for that kind of event (and possibly remove them from the buffer altogether, in case they are not longer waiting for any kind of event). All this poses a number of difficulties that we glossed over in the previous section but we now address in earnest:

2. (a) The buffer may grow to a considerable extent if the attribution windows are very large. This may seriously rise the computational cost of some buffer operations.
   (b) Also, large attribution windows imply large learning latencies, which is particularly undesirable in an online setting, since we wish to learn as soon as possible.
   (c) Potentially, many concurrent read and write operations will hit the buffer. We want to minimize them and, if possible, make them access disjoint parts of the buffer.

Regarding (2a), remember that we are already sharding based on the pubsub mechanism along the lines described in the previous section. Now, each shard

deals with a subset of the total market transactions, so its ongoing transactions buffer only has to store transactions in this subset. In more concrete terms, each shard stores its buffer in a table residing at a database in a (local or remote) postgresql server; in the local server case we implement these tables on top of the inheritance-based postgresql table partitioning feature[4].

Another strategy to keep the buffer size in check is what we call subsetting. Most predictions in an RTB setting are conditional to a bid and an impression having occurred, since there is not much to say otherwise[5]. Our bid events just piggyback the auction information, so there is no need to receive and store separate auction events: we only care about the relatively small subset of auctions for which we bid, usually less than one tenth of the number of auctions. Similarly, since impressions occur very early in the event sequence, we quickly remove from the buffer those transactions for which the impression attribution window has expired without an impression event having arrived. Thus, after a short wait for impression events, we only keep the subset of bids that effectively bought an impression, which is usually about one fifth of the number of bids. Analogously, we decompose the estimate of the probability of occurrence of an install event given that an impression occurred in: (i) the estimate of the probability of occurrence of a click given that an impression occurred (CTR $= p(\texttt{click}|\texttt{impression} = 1)$) and (ii) the estimate of the probability of occurrence of an install given that a click occurred (CVR $= p(\texttt{install}|\texttt{click} = 1)$). As CTRs are typically less than 5% and clicks arrive much earlier than installs, this enables us to discard impressions without clicks immediately after the click attribution window expires, keeping only the subset of transactions with clicks for the purpose of CVR estimation.

Regarding (2b), for events with very large attribution windows we split the waiting window in two: a short window and a long window. When the short window expires, we emit an observation for the "short event", in order to estimate the probability of the event occurring inside the short window as soon as possible. This short window usually contains about 90% of event occurrences, because the distribution of event occurrence times is typically very skewed and long-tailed, and thus the short window is good enough for most practical concerns. Much later, when the long window expires, we also emit the "long event", in order to estimate the probability of occurrence of the event after the short window has expired but before the long window expires. As the short and long events are deliberately defined to be disjoint, both probabilities may be combined by a simple addition. But the bottom line here is that one of the terms in the addition is much more informative and much more frequently and quickly updated than the other. Notice that the separate estimate of CTR and CVR described in the previous paragraph also gives us a quick estimate of the CTR, without the need to wait for an install which may arrive days later. Putting everything together, we may compute the probability of occurrence of an install given that an impression occurred as

[5] Further development of an RTB transaction which auction your bidder has not won is unknown to you.

$p(\texttt{click}|\texttt{impression}{=}1)\cdot[p(\texttt{install}_{short}|\texttt{click}{=}1)+p(\texttt{install}_{long}|\texttt{click}{=}1)]$, that is $\mathrm{CTR}\cdot(\mathrm{CVR}_{short}+\mathrm{CVR}_{long})$, learning faster and more frequently from earlier events but still attempting a more precise estimation by means of including later events. The price to pay obviously is that more models have to be estimated and evaluated, but usually this must be done anyway because relevant parameters and hyper-parameters vary across models for the different event rates.

Regarding (2c), we avoid by all means explicitly marking the transactions in the buffer as "waiting for these events" or, alternatively, as "done for those events". Instead, we keep a sequence of increasing timestamps $w_{1t} < w_{2t} < \cdots < w_{kt}$ meaning that, at moment t, each transaction in the buffer that was created between w_{it} and $w_{i+1,t}$ (we call this interval phase i)[6], is currently waiting for events of types $\{i, i+1, \ldots, k\}$ but is already done waiting for events of types $\{1, 2, \ldots, i-1\}$. Periodically, the sequence $w_{1t} < w_{2t} < \cdots < w_{kt}$ will be updated to $w_{1t'} < w_{2t'} < \cdots < w_{kt'}$, with $t' > t$, implying that the transactions in the buffer that were created between w_{it} and $w_{it'}$ are now done waiting for event i, so the corresponding observation can be emitted for the learning system to consume (we say that those transactions moved from phase i to phase $i+1$). This way, only the timestamp sequence must be updated, while the buffer itself just needs to implement efficient updating by transaction id (to consolidate incoming events for the same transactions) and efficient querying by date range. Many storage systems comply with these requirements and, as we have already mentioned, we are currently using postgresql with pretty satisfactory results. Notice also that $t' - t$ defines a period that we can conveniently configure in order to process and emit batches of transactions of length at least $t' - t$, so as to avoid flooding the database server with a multitude of tiny queries. In our experience, this reduces overhead and largely improves performance.

Some final words regarding the use of postgresql in this particular scenario. The buffer tables are obviously subject to many insert, update and delete operations, since incoming events may create a new row or update an existent one, and the end of attribution windows may trigger the deletion of waiting rows. It's well known that postgresql implements multiversion concurrency control (MVCC) in order to handle multiple concurrent accesses to the same table, so an update operation doesn't really modify an existent row but a new one is created instead and, similarly, a delete operation doesn't remove a row at all but just flags it as removed. This makes possible for currently running transactions to see their own "snapshots" of the database without locking, which is a good thing. Periodically, a vacuum operation marks all obsolete rows enabling newer transactions

[6] In practice, attribution window lengths show wild dynamic variations because of ever changing market structure and internal processing lags. So phases shrink and enlarge, pulling and pushing adjacent phases. We decided to leave this topic out of this paper because its description would have required a significant amount of space and also because our solution, while not completely uninteresting by itself, is conflated with some specifics of our platform that are not really of general interest.

to reuse their space[7]. Therefore, long transactions may impede the vacuum process to effectively flag space as reusable[9], and this is particularly problematic in our use case since the buildup of buffer tables can become significant in a matter of minutes when no proper vacuum is taking place. We dealt with this issue in two ways: first, we implemented a fine-tuned aggressive vacuum schedule for critical tables; second, we isolated the buffer tables into their own database[8], thus preventing long, unrelated, transactions to interfere with the vacuuming of buffers. Another problem related to the high frequency of buffer updates is the dismaying growth of the write-ahead log (WAL). There exists a full range of solutions to reduce the overhead that durability of transactions imposes[5], each one offering its own durability/performance trade-off, but we simply got radical here: we made the buffer tables unlogged[6], which means they won't be written to the WAL at all. The gains in speed and space are overwhelming but, of course, the entire table may be lost after a crash; this is not such a big deal as it might sound, since buffers are transient, crashes are rare, and we always have plenty of new data to learn from.

Conclusions

It is often assumed that the most demanding and daunting task in building a machine learning system is the implementation of the learning algorithm itself. But, specially in an online massive-scale learning scenario, some peripheral tasks —like those of collecting and preprocessing the information so that it can be readily consumed by the learning algorithm— may pose problems that dominate development times and overall system complexity.

We would be satisfied if this work has conveyed some useful heads-ups and nudges for practical developers working in this exciting field. We also expect to have shown that sometimes to combine some simple but general building blocks and strategies could be a more effective trade-off than to immediately resort to a heap of fashionable tools.

Another ancillary task that is often overlooked by the literature is that of evaluating models. In a real-time bidding setting, in which prices for a large decision set of candidate banners must be computed in a few milliseconds based on event rate predictions, model evaluation cannot be an afterthought. We plan to cover our approach to model evaluation next year, in a follow-up to this work that keeps the same hands-on spirit.

[7] Moreover, a full vacuum operation that actually deletes those obsolete rows and compresses the table may be scheduled to run from time to time, but it's an expensive process that acquires an exclusive lock on the table and, thus, should be used sparingly at most.

[8] A postgresql database is just a named collections of SQL objects, not a separate server or instance, so it's quite easy and cheap to create a new one.

References

1. Cristián Antuña, Claudio Freire, Juan Pampliega, and Carlos Pita. Una plataforma de real time bidding escalable e inteligente. In *AGRANDA, Simposio Argentino de GRANdes DAtos*. 2015.
2. Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.
3. John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.
4. The PostgreSQL Global Development Group. *PostgreSQL 9.6 Documentation*, chapter 5.10. Available at `http://www.postgresql.org/docs/9.6/static/ddl-partitioning.html`.
5. The PostgreSQL Global Development Group. *PostgreSQL 9.6 Documentation*, chapter 14.5. Available at `https://www.postgresql.org/docs/9.6/static/non-durability.html`.
6. The PostgreSQL Global Development Group. *PostgreSQL 9.6 Documentation*. Available at `https://www.postgresql.org/docs/9.6/static/sql-createtable.html#SQL-CREATETABLE-UNLOGGED`.
7. Pieter Hintjens. 29/pubsub - zeromq publish-subscribe pattern. Available at `http://rfc.zeromq.org/spec:29`.
8. Pieter Hintjens and Martin Sustrik. Zeromq distributed messaging library. Available at `http://zeromq.org`.
9. Reuven Lerner. *In PostgreSQL, as in life, don't wait too long to commit*. 2015. Available at `http://blog.lerner.co.il/in-postgresql-as-in-life-dont-wait-too-long-to-commit/`.
10. H Brendan McMahan. A unified view of regularized dual averaging and mirror descent with implicit updates. *arXiv preprint arXiv:1009.3240*, 2010.