Una plataforma de Real Time Bidding escalable e inteligente

Cristián Antuña, Claudio Freire,
Juan Martín Pampliega, Carlos Pita

Jampp
{cristian,claudio,juan,carlos}@jampp.com

Resumen El trabajo busca describir las tecnologías, procesos y algoritmos aplicados para construir la plataforma de real-time bidding de Jampp y cómo se pretende evolucionar la misma.

1. Introducción

Jampp es una plataforma utilizada para la promoción y remarketing de aplicaciones móviles. Mediante la compra programática de publicidad, la plataforma ayuda a anunciantes a promover sus aplicaciones a nivel global, esto incluye tanto la adquisición de nuevos usuarios como la posibilidad de dirigirse a segmentos específicos de usuarios ya existentes, para poder atraerlos nuevamente a la aplicación y promover mayor participación.[1] De este modo, es posible recuperar usuarios que ya se instalaron la app, pero están inactivos. Como principal herramienta, Jampp cuenta con su plataforma propietaria de compra programática en tiempo real (Real Time Bidding, RTB), que está integrada a 18 exchanges de RTB y más de 150 ad networks móviles.

La compra programática en tiempo real es un proceso que se inicia cuando un usuario ingresa a una web o a una app que posee publicidad. Esta acción genera un pedido de publicidad que, a través, de un sistema intermedio (Ad Exchange) se hace llegar a todas las plataformas de demanda suscriptas a estos pedidos. En ese momento, las plataformas de demanda tienen 100ms para poder ofertar por el

[1] http://jampp.pr.co/96694-jampp-la-plataforma-de-mobile-app-marketing-recibio-una-inversion-de-7m-de-dolares-para-acelerar-su-crecimiento

espacio, en base a los datos que llegan del usuario que va a ver la publicidad (ver figura 1). Con las ofertas de todas las plataformas, se realiza una subasta y se muestra la publicidad de la ganadora.[2] Encarar este proceso a una escala rentable significa tratar con un gran volumen de datos, generados con una gran velocidad y con estructura poco definida.

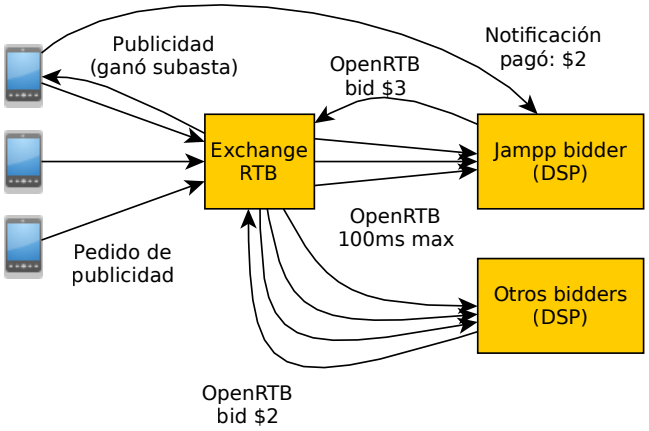

Figura 1. Ejemplo de subastas RTB, donde 3 oportunidades de impresiones generan 3 llamadas al exchange, y dónde sólo una llamada produce una oferta ganadora para ese slot publicitario. Éste es un caso de una *subasta Vickrey*. AGREGAR EXPLICACIÓN

Actualmente, Jampp recibe más de 180.000 pedidos de subastas RTB por segundo, en promedio, lo que genera un tráfico de aproximadamente 300 MB/s o 25 TB de datos por día. Además, Jampp trackea unos 50 millones de eventos por día: instalaciones de aplicaciones y acciones que los usuarios realizan dentro de la aplicación, como así también información contextual utilizada para la segmentación de usuarios. Para poder lidiar con este volumen, utilizamos tecnologías como ZMQ, PostgreSQL, SciPy, Cython y Memcache, y representaciones compactas de los datos, en particular eliminando redundancias.

La arquitectura de la plataforma Jampp consta de cuatro componentes principales que, actualmente, se encuentran corriendo en

[2] http://www.iab.net/guidelines/rtbproject

la nube de Amazon Web Services. En primer lugar, un *bidder* que implementa el protocolo OpenRTB y se ocupa de contestar los pedidos de subasta (por lo que su funcionamiento es muy sensible a la latencia). En segundo lugar, un sistema de trackeo de clicks y eventos dentro y fuera de un app que permite atribuir un click en una publicidad con la ocurrencia de un evento dentro de una aplicación y/o una instalación de la misma. Tercero, se implementó un sistema de aprendizaje supervisado que mejora el desempeño del *bidder* utilizando los datos generados por los dos sistemas anteriores de forma tal de tratar de estimar tasas de conversión y cómo explorar el mercado de manera óptima. Finalmente, un canal de publicación y suscripción de mensajes con muestreo consistente para interconectar los componentes entre sí de manera óptima (ver figura 2).

Particularmente, en el sistema de aprendizaje, se tuvieron varias consideraciones debido a la naturaleza del problema a resolver, el volumen y lavelocidad de los datos. La llegada de cada observación es extremadamente frecuente y, además, en la etapa de estimación se expresa como un vector de respuestas acompañado de otro que indica características de esa operación. Este último puede llegar a tener un tamaño considerable, pues se compone de *dummies* que indican cientos de miles de categorías, definidas por alrededor de veinte variables relevantes y su producto cartesiano. Por este motivo, junto con un error de estimación bajo, se buscan soluciones *sparse* y métodos de estimación que se computen rápidamente. Para el primer aspecto, se aplica una regularización a la regresión que estima las tasas. Para el segundo, algoritmos *online*.

El objetivo de este paper es detallar las distintas tecnologías, procesos y algoritmos aplicados para la construcción de esta plataforma de *Real Time Bidding*, de manera que resulte escalable e inteligente. Además, se precisará hacia dónde se está pensando evolucionar la plataforma para aumentar y mejorar sus capacidades.

2. Bidder

El *bidder* es el componente que recibe los pedidos de subastas por *espacios* (o *slots*) publicitarios, y decide si comprar el *slot*, y cuánto pagar por él. Para esto, utiliza dos fuentes de datos: por un lado, una base de datos MySQL que contiene la configuración de las campañas,

expuesta a los usuarios a través de una aplicación web.Por otro lado, una base de datos PostgreSQL que contiene métricas agregadas, que se utilizan para regular el *pacing* [3] - es decir, la distribución a lo largo del tiempo del gasto en cada campaña.

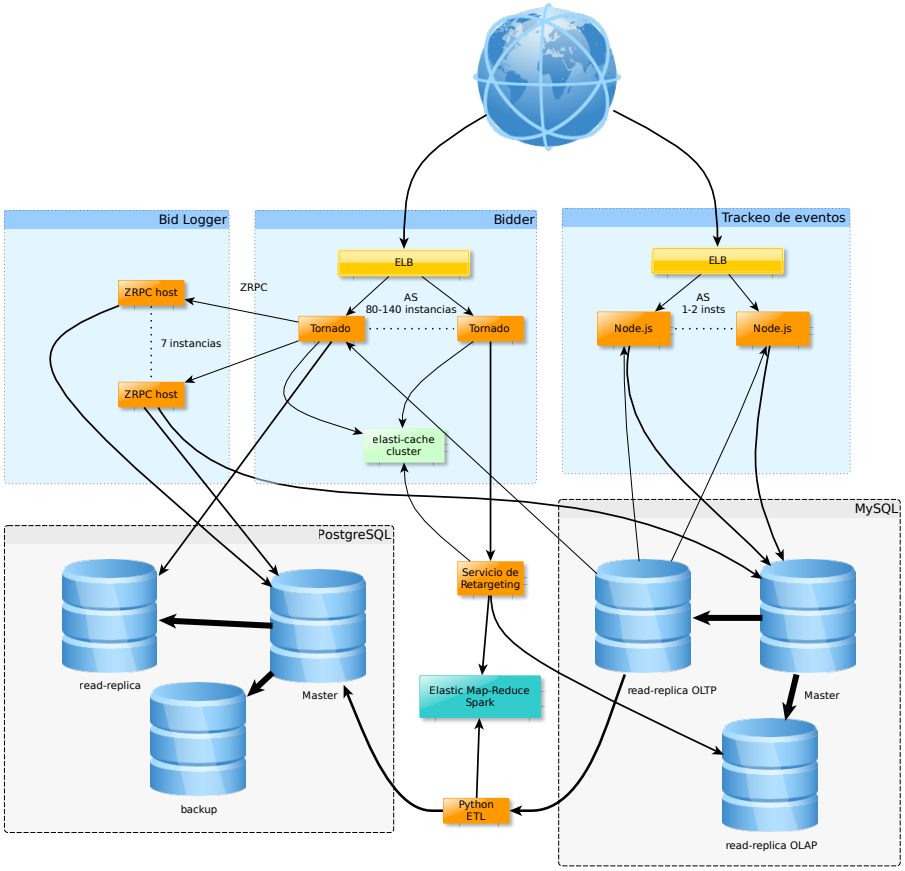

Figura 2. Esquema aproximado de la arquitectura del *bidder*

El *bidder* está compuesto por una cantidad variable de procesos implementados con tornado[3] (Python), detrás de un *load balancer* y un sistema de escalado automático, aprovechando la capacidad no utilizada de la nube (mercado de Spots) para mantener una operativa eficiente en términos de costos.

[3] http://www.tornadoweb.org/en/stable/

2.1. Pacing y cacheo

Dada la arquitectura distribuida del *bidder*, se hace necesaria cierta comunicación entre los nodos para poder llevar el algoritmo de *pacing* descripto en [3]. Se puede modelar como un problema aleatorio, pues depende de factores externos fuera de nuestro control y relativamente impredecibles, como son los otros *bidders*:

$$E(S) = E(P_v|P_b, R) \cdot P(W|P_b, R)$$

Donde S es la variable aleatoria que representa el gasto efectivo por cada 1000 impresiones, P_v el precio de vaciado, P_b el precio ofertado, W ganar la subasta, y R representa la información de contexto provista en el pedido de subasta.

Con este gasto esperado, pues, se puede actualizar de forma especulativa el estado del algoritmo de *pacing*, ajustando el funcionamiento del *bidder* según el resultado esperado de cada oferta, y periódicamente se ajusta el estado al costo efectivamente trackeado según la base de datos. De esta forma se consigue una operación estable a la vez minimizando la comunicación necesaria entre nodos.

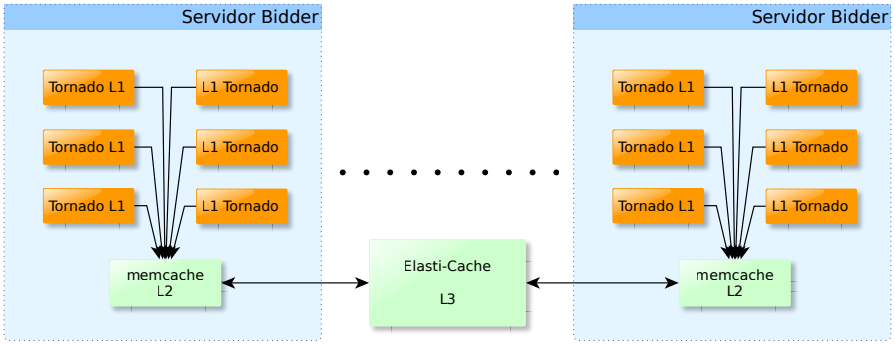

Figura 3. Cacheo en el *bidder*. Un LRU *in-process* inclusivo (L1) alimentado por un memcache local (L2), que a su vez es alimentado por un memcache remoto (L3). Los resultados se guardan en el L3, y se promueven a medida que se hace necesario.

Para realizar todo esto, incluyendo la extracción de información de contexto del *request*, es necesario consultar ambas bases de datos,

calcular métricas y planes —proyecciones de gasto según los parámetros del pacing—, lo cual es relativamente costoso, y si se realizara de forma sincrónica sería prohibitivo dados los estrictos requisitos de latencia. Para sobrellevar esto, se implementó una arquitectura de cacheo en varios niveles, ilustrado en la figura 3.

La ubicación del primer nivel de caché en una estructura de datos *in-process* garantiza la inmediata disponibilidad del conjunto más relevante de datos. En la mayoría de los accesos al caché, un faltante en el primer nivel dispara una promoción o cálculo de forma asincrónica, mientras el *bidder* procede sin el dato, modificando el algoritmo de forma acorde, o simplemente no ofertando en esta subasta. Se asume que, dada la recurrencia del tráfico, cualquier oportunidad así ignorada podrá ser aprovechada en un momento futuro, cuando el dato necesario se encuentre ya en el caché.

La coherencia entre los niveles de caché se consigue comunicando los procesos en un canal PUB/SUB implementado con ZMQ. Así, si un proceso necesita un valor, puede coordinar con los otros procesos para no duplicar trabajo. A su vez, si por alguna razón se invalida o reemplaza una entrada, los niveles inferiores de los otros procesos son notificados a través de este canal para que también sean invalidados.

2.2. Cómputo de la oferta

Para decidir **si** ofertar, y **cuánto** ofertar, se utilizan similares métricas, en este caso derivadas de algoritmos de *machine learning* (ver sección 5). Con métricas como la tasas de click y la tasa de conversión, respectivamente

$$CTR = P(\texttt{click}|\texttt{vista})$$
$$CVR = P(\texttt{apertura}|\texttt{click})$$

Se elige un precio que, en promedio, consiga ser rentable:

$$P_b = P_c \cdot R_g$$
$$CTR = \frac{N_c}{N_i}$$
$$CVR = \frac{N_a}{N_c}$$

$$R_g = CTR \circ CTR \cdot CVR$$

Donde P_c es el precio por objetivo que acepta el cliente, N_c la cantidad de clicks en la muestra, N_i la cantidad de impresiones, y N_a la cantidad de aperturas.

Luego se ejecuta una subasta interna entre todas las campañas cuyas reglas de negocio permiten ofertar en el pedido actual, eligiendo una creatividad al azar para cada una, con una distribución que premia mejor desempeño para maximizarlo.

Con el fin de no penalizar campañas o contextos que tienen todavía una muestra demasiado pequeña, la fórmula anterior se la torna en una variante optimista, tomando un extremo optimista de un intervalo de confianza de las tasas estimadas respectivas. Ésto se hace sólo para la subasta interna, no para elegir el precio ofertado.

El ganador de la subasta interna también se elige al azar, con una distribución basada en la probabilidad de ganar y ser rentable que se estima para cada bid interno. De este modo, se evita inanición de las campañas de menor desempeño, que ocurriría de elegir el bid con mayor desempeño.

3. Trackeo y Stream de Eventos

El sistema de trackeo y atribución de eventos está dividido en dos partes. Por un lado, una aplicación escrita en Node.js horizontalmente escalable que guarda los datos en una base MySQL. Por otro lado, adelante de la aplicación de Node.js se encuentra el producto Elastic Load Balancer de AWS, que reparte la carga entre las distintas instancias (servidores virtuales) donde se encuentra corriendo la aplicación de Node.js.

Primero, el sistema recibe clicks hechos en publicidades de Jampp y registra datos del dispositivo que nos permiten identificarlo unívocamente. Luego, si el usuario sigue adelante recibe un *install*, que dependiendo del tipo de campaña de promoción a la cual pertenece el ad puede significar una conversión (evento al que se desea llegar y por el cual nos paga el cliente). Finalmente, registra eventos que ocurren dentro de la aplicación como la compra (en una aplicación de e-commerce) o la reserva (en una aplicación de reserva de taxis). Estos eventos pueden pertenecer a dispositivos que instalaron

la aplicación debido a una publicidad de Jampp (se denominan eventos atribuídos) o los que la instalaron por otras razones (orgánicos).

Utilizamos dos tipos de campañas: las de instalación de apps (denominadas *User Acquisition*) y las de *retargeting* (denominadas *User Engagement*). En el primer tipo de campañas, el cliente paga por cada instalación. Por lo tanto, se desea optimizar para obtener la mayor cantidad de instalaciones con el menor gasto posible en publicidad. En el segundo tipo, el cliente paga por click en la publicidad, y se intenta que el usuario vuelva a utilizar la aplicación o que realice una acción determinada dentro de la misma, en base a su comportamiento anterior.

El *bidder*, por otro lado, también genera otro tipo de eventos, como subastas, ofertas, subastas ganadas, perdidas e impresiones. Éstos se envían por un mecanismo de RPC sobre ZMQ a un set de procesos llamados Bid Loggers, los cuales agregan y muestrean los datos para incorporarlos al Postgres y así poder alimentar al sistema de *pacing*.

Estos otros eventos se suman a los eventos de la plataforma de trackeo y se envían por un *stream* constante de datos hecho en ZMQ. El *stream* fue diseñado como una cola de mensajes sin persistencia. Es decir, solamente se envía un mensaje si hay algún sistema suscripto a ese tópico y partición en sí. Esto se debe, como mencionamos previamente, a que el *bidder* genera un volumen de datos muy grande (más de 25 TB por día) y almacenar todos estos datos sería prohibitivo en cuanto a costo. Por el momento, elegimos hacer muestreo de los eventos que genera el *bidder* y almacenar el 100 % de los eventos post-impresión que recibe el sistema de trackeo.

En el *stream* los eventos se envían particionados por un *id de transacción* que se mantiene durante toda la cadena de eventos: subasta, *bid*, impresión, click, apertura, compra, etc. Esto permite realizar un muestreo consistente, al suscribirse a una única partición (o a un subconjunto de particiones) para ciertos eventos, y aún así obtener todos los eventos asociados, que podrían perderse en un muestreo simple. Esta capacidad es central al momento de entrenar los modelos de *machine learning*, pues evita el sesgo sistemático que se desprendería de descartar eventos post-click en un muestreo simple.

4. Retargeting

Para implementar los sistemas de *retargeting* (remarketing), la mayoría de las empresas utilizan bases de datos *key-value store* con latencia muy baja que les permite consultar de manera muy rápida el historial de eventos de una persona para poder determinar en cada caso si pertenece a un segmento al cual se quiera apuntar con publicidad con el objetivo de que realicen alguna acción dentro de la aplicación. A menudo, estos sistemas resultan muy costosos ya que utilizan grandes clústeres para mantener toda esta información en memoria con bases de datos como AeroSpike y también necesitan un enorme *throughtput* que encarece otras soluciones como DynamoDB.

Al investigar distintas alternativas, encontramos que el uso de estructuras probabilísticas - en este caso Bloom Filters[4] encaja mejor con nuestras necesidades.

Inicialmente, decidimos relajar la necesidad de mantener los segmentos constantemente actualizados. El sistema utiliza Python y MySQL para calcular el conjunto de identificaciones de dispositivos que conforman el segmento y posteriormente transformarlo a un Bloom Filter suficientemente compacto como para mantenerlo en memoria.

Debido a las necesidades de escalar para manejar rangos de tiempo más grandes, que involucran un volumen de datos mucho mayor (cientos de GB), estamos empezando a desarrollar una plataforma que utiliza Apache Spark para el cálculo de los segmentos. Además de poder escalar de forma horizontal para poder manejar más volumen, Apache Spark nos permite utilizar procesamiento de *streams* para mantener los segmentos constantemente actualizados de manera eficiente, dado que, agregar un elemento a un Bloom Filter tien complejidad de $O(1)$.

5. Aprendizaje

Nuestra plataforma cuenta con un sistema de aprendizaje para alimentar modelos de predicción de tasas de conversión para distin-

[4] Una estructura de datos probabilística que permite probar si un elemento es miembro de un set con 100 % de certeza en el caso de una respuesta negativa y, en el caso de que sea positiva, con un porcentaje de certeza que depende del tamaño utilizado para armar el Bloom Filter.

tos eventos de interés (*clicks, installs, app-opens*, etc.). Las tasas de conversión estimadas reflejan la probabilidad de alcanzar el evento objetivo, dado que, se consiguió imprimir (*i.e.*, la audiencia ha sido efectivamente expuesta al *banner* publicitario). Esta probabilidad condicional —junto con parámetros económicos previamente ingresados al sistema— permite optimizar nuestras ofertas (*bids*) para cada espacio publicitario subastado, en base a estimaciones que toman como predictores un conjunto de características (*features*) de la transacción en curso.

Dados los requerimientos que enfrentamos, decidimos utilizar algoritmos *online* de tipo SGD (*Stochastic Gradient Descent*). Específicamente, implementamos una versión del algoritmo FTRL-P (
emphFollow the Regularized Leader Proximal[6]) con tasa de aprendizaje adaptativa [2]. Esta es una técnica utilizada actualmente en el campo de la publicidad *online*[7] para predecir tasas de conversión, que explota convenientemente el *trade-off* entre error de optimización y error de estimación en escenarios para los cuales la capacidad computacional —y no la cantidad de datos— aparece como el cuello de botella[1]. Nuestra implementación es capaz de procesar grandes cantidades de observaciones que ingresan continuamente al sistema de aprendizaje desde un *stream* de eventos de mercado publicado por el *bidder*.

El primer paso, luego de recibir un evento del *stream*, es convertirlo en un vector numérico capaz de ser asimilado por el estimador SGD. La mayor parte de los predictores son *features* categóricas —*i.e.* medidas en una escala puramente nominal— que deben ser transformadas en *features* binarias, categoría por categoría. Además de las variables individuales, consideramos interacciones de hasta k variables[5]. Llamamos *diseño* a este paso inicial, ya que, esencialmente transforma observaciones del dominio en filas de una "matriz de diseño" (la cual nunca se materializa, dada la naturaleza *online* del procedimiento).

Evidentemente, las transformaciones y combinaciones efectuadas durante el paso de diseño aumentan la dimensión del espacio de

[5] k es un parámetro del sistema. Otros parámetros del sistema, que iremos introduciendo en el transcurso de la presente sección, son: p, λ, η_0 y ϕ.

predictores potenciales de manera exponencial. Por fortuna, las observaciones presentan una estructura muy *sparse*: solo unas pocas categorías y combinaciones de categorías estarán efectivamente presentes en cada una, por lo que, el vector que representa la observación transformada consiste casi íntegramente de ceros. Formalmente, el vector es enorme; a fines prácticos, solo contiene unas pocas decenas o centenas de valores no nulos. Por otra parte, la mayor parte de las combinaciones de categorías no llegan a ocurrir en la práctica, u ocurren muy raramente. Por lo tanto, filtrar las combinaciones inusuales reduce la complejidad del sistema sin sacrificar demasiado poder predictivo. A este fin, implementamos una heurística sencilla según la cual una combinación de categorías es aceptada con probabilidad p; una vez aceptada por primera vez, dicha combinación será sucesivamente aceptada de forma automática.

Con la observación ya convertida en un vector numérico $[x, y]$ —con x el vector resultante del paso de diseño e y una variable binaria de respuesta que indica la conversión o no del evento de interés— que el estimador SGD puede procesar de forma directa, alimentamos varios estimadores parametrizados de acuerdo a una grilla de combinaciones paramétricas para los parámetros k, p, λ y η_0, algunos de los cuales ya describimos y otros describiremos más adelante. Actualmente, la grilla está prefijada y contiene alrededor de 100 combinaciones paramétricas, pero consideramos la posibilidad de introducir —a mediano plazo— un algoritmo genético que explore el espacio paramétrico de manera dinámica y evolutiva. Para cada estimador se calcula un puntaje empleando la técnica de validación progresiva: antes de asimilar una nueva observación se calcula la predicción $\hat{y}$ del estimador condicional a x y se obtiene la pérdida (cuadrática) $L(y, \hat{y})$; luego el estimador aprende de la nueva observación[6].

El paso final del *pipeline* consiste en seleccionar, periódicamente, el estimador de mejor desempeño (*i.e.* aquel con el menor error de predicción esperado estimado) para proveer de estimaciones al sistema optimizador del *bidder*. Como el *bidder* opera sujeto a una restricción temporal muy ajustada, dentro de la cual debe llevar a cabo numerosas estimaciones (una para cada uno de cientos de *ban-*

[6] Notar cierta similitud con la técnica LOOCV: la pérdida es calculada sobre un estimador que aprendió de todas las observaciones disponibles excepto una.

ners potenciales) a fin de calcular el *bid* para la subasta en curso, el estimador original se transforma en una estructura de árbol informacionalmente equivalente, que garantiza una evaluación en $O(m)$, con m el número de *features* originales. A diferencia de las decenas o cientos de *features* transformadas y combinadas en el paso de diseño, el número m es usualmente pequeño (menor a 20). Adicionalmente, el árbol se puede evaluar de manera parcial sobre un subconjunto de *features* comunes a toda la transacción, y luego subevaluar a fin de obtener predicciones específicas a cada *banner* potencial.

Pasamos ahora a describir en mayor detalle el algoritmo de aprendizaje. Como ya se mencionó, se trata de una implementación $in-house$ de FTRL-P con tasas de aprendizaje adaptativas. Sobre la base de FTRL-P, y a fin de reducir el número de predictores, introducimos un *update* compuesto (*composite update*[6]) consistente en una regularización de tipo Lasso, es decir, de norma L_1. La implementación introduce un número de optimizaciones, algunas de ellas basadas en la implementación de Vowpal Wabbit[4], algunas de ellas de cuño propio.

Sean θ el vector de coeficientes estimado e I los índices de los elementos no nulos de x . Nuestra predicción de tasa de conversión $\hat{y}$ es lineal en θ[7]:

$$\hat{y_T} = \sum_{i \in I} \theta_{i,T}.x_{i,T}$$

Presentamos a continuación el pseudo-código del algoritmo de aprendizaje, que muestra esquemáticamente los pasos ejecutados frente a la T-ésima observación $[x, y]$ recibida para actualizar el vector de coeficientes estimados θ:

Se puede probar que el paso T del algoritmo equivale a calcular θ_{T+1} según:

$$\theta_{T+1} = \underset{\theta}{argmin} \left[\sum_{t=1}^{T} g_t.\theta + \frac{1}{2\eta_0} \sum_{t=1}^{T} \|\theta - \theta_T\|^2_{\left[\sqrt{diag(\sum_{t=1}^{T} g_t g_t')}\right]} + \lambda \|\theta\|_1 \right]$$

Resaltamos algunas características salientes del algoritmo presentado:

[7] Estamos comenzando a experimentar también con *links* Logit.

Algoritmo 1 Update del estimador ante observación $[x, y]$

$\eta_0, \lambda \leftarrow$ inputs de cada estimador
$u \leftarrow U[0, 1]$
for $i \in I$ **do**
  $g_i \leftarrow (y - \hat{y}).x_i$
  $\sigma \leftarrow \frac{\sqrt{\sum_{t=1}^{T} {g_{i,t}}^2} - \sqrt{\sum_{t=1}^{T-1} {g_{i,t}}^2}}{\eta_0}$
  $\eta \leftarrow \frac{\eta_0}{\sum_{t=1}^{T} g_{i,t}}$
  $w_{i,T} \leftarrow w_{i,T-1} + \sigma\theta_i$
  $z_i \leftarrow \sum_{t=1}^{T} g_{i,t} - w_{i,T}$
  if $z_i \in [-\lambda, \lambda]$ **then**
    $\theta_i \leftarrow 0$
  else
    $\theta_i \leftarrow -\eta(z_i - sgn(z_i)\lambda)$
  end if
end for

- Iterar solamente sobre I permite aprovechar la representación *sparse* de las observaciones. Aprender de una observación implica una única pasada sobre sus valores no nulos, que se puede realizar muy eficientemente.

- Las actualizaciones son del estilo FTL, empleando una aproximación lineal (el gradiente) de la función de pérdida. En consecuencia, se consideran los gradientes desde el momento $t = 1$ hasta el momento $t = T$. Esto se refleja en el primer término de la ecuación anterior.

- El segundo término de la ecuación es un término de proximalidad dado por la sumatoria de las distancias de Mahalanobis (con respecto a $\sqrt{diag(\sum_{t=1}^{T} g_t g_t')}$) entre el nuevo vector θ y los vectores θ_t desde el momento $t = 1$ hasta el momento $t = T$. Este término es un regularizador que apunta a mantener la estabilidad del estimador, evitando variaciones bruscas en la estimación de θ entre observaciones sucesivas y, por lo tanto, limitando la influencia que cada una puede tener por sí sola sobre la estimación.

- El tercer término de la ecuación corresponde a una regularización de tipo Lasso para el *update*. El objetivo de este término es obtener un modelo más parsimonioso (con menos coeficientes

no nulos); a parámetros λ mayores corresponderán modelos más parsimoniosos[8].

- Las tasas de aprendizaje son adaptativas[2]. Esto implica que cada *feature* posee su propia tasa, que decae según el comportamiento observado para la *feature*. A grandes rasgos, la tasa decae más rápidamente cuanto más "ruidosa" es la *feature*, *i.e.* cuanto más oscilante es el componente del gradiente correspondiente a la *feature* en cuestión. Al mantener tasas de aprendizaje por *feature* evitamos el problema de que *features* informativas pero infrecuentes paguen el costo de llegar tarde, cuando una tasa de aprendizaje global ya hubiera decaído considerablemente. El parámetro η_0 indica la magnitud del primer movimiento de la *feature*, que tiene lugar la primera vez que es observada con un gradiente no nulo.

Además de la calidad predictiva, una de los objetivos principales es mantener un modelo lo más parsimonioso posible, a fin de que el árbol de predicción exportado al *bidder* no exceda un tamaño tratable. Los objetivos de predicción y parsimonia no siempre están en conflicto; a fin de cuentas, un exceso de predictores conduce al sobreajuste y, por ende, a predicciones excesivamente variables. Pero, sin duda, existe un subespacio paramétrico que sí presenta un *trade-off* que debe calibrarse cuidadosamente. Para cerrar, y dada la importancia de este aspecto, listamos —a riesgo de repetir algunas— las distintas técnicas que empleamos para eliminar predictores sin sacrificar excesivamente la calidad predictiva, cada una de las cuales regulada por un parámetro diferente:

- Para las combinaciones de categorías extremadamente raras, es difícil ingresar al modelo dada la probabilidad de admisión p. En promedio, una combinación de categorías debe aparecer $1/p$ veces para ser admitida.
- El término de Lasso, con ponderación λ, que tiende a descartar coeficientes muy cercanos a 0.

[8] Si este término se introdujera directamente en la función de pérdida acabaríamos optimizando sobre su gradiente, con la consecuencia de que la probabilidad de que un *update* resultara en un 0 exacto sería virtualmente nula. Al introducirlo explícitamente como un *composite update* evitamos este problema. Existen otras soluciones en la literatura, como [5], que permiten aproximaciones al Lasso dentro de la estructura formal de descenso de gradientes —por contraposición a la estructura FTL—.

- Un factor de olvido ϕ que relaja exponencialmente la restricción de proximalidad respecto a θ_t's alejados en el tiempo, conservando solo una fracción $(1-\phi)^{T-t}$ de la restricción total [9]. El relajamiento de la restricción de proximalidad permite al vector de coeficientes θ distanciarse cada vez más de estimaciones viejas y ajustarse mejor a un escenario cambiante con potenciales *breaks* estructurales. Si una *feature* no reaparece durante un período prolongado de tiempo es posible que el término de Lasso prime sobre el término de proximalidad y la *feature* sea eliminada del modelo.

6. Evolución futura

En cuanto al aspecto tecnológico tenemos tres herramientas cuya adopción puede ayudarnos a extender las capacidades de la plataforma en varios aspectos. Por un lado, Apache Kafka[10] nos permitiría estandarizar la comunicación de eventos entre los distintos subsistemas que requieren estos datos y poder disponibilizar la información en tiempo real.

Por otro lado, Elasticsearch nos permitiría mejorar nuestra capacidad de analizar *logs* de aplicaciones para poder monitorear de manera más simple el estado de las mismas y determinar de manera más rápida las causas de errores. Además, Elasticsearch tiene herramientas complementarias como Kibana para armar *dashboards* y Watcher para el armado de alarmas en base a reglas que nos permitirían incorporar aún más funcionalidades sin mucho trabajo extra.

Finalmente, a pesar de que ya actualmente estamos comenzando a utilizar Apache Spark, esta herramienta puede permitirnos estandarizar una forma de resolver muchos casos en los cuales necesitamos procesamiento escalable de datos y además interactuar con simpleza con datos en tiempo real y utlizar algoritmos de la librería de *machine learning*.

[9] Por motivos de eficiencia, este cálculo no se realiza cada vez que se observa un nuevo mensaje, sino que al final de cada período de duración τ se aplica el ajuste en *batch* sobre todos los coeficientes.

[10] una cola de mensajes distribuída, particionable y altamente durable que permite almacenar toda la historia de eventos de forma eficiente

7. Conclusión y Lecciones Aprendidas

Entre las numerosas lecciones aprendidas construyendo nuestra plataforma, podemos resaltar algunas particularmente importantes para cualquiera intentando resolver un problema similar.

En primer lugar, cuando se trabaja en problemas que implican respuestas en tiempo real con muy baja latencia, resulta crucial el uso correcto de *caches* para garantizar el *throughput* y latencia, balanceando la frescura de los resultados contra la latencia máxima de acceso (cómputo asincrónico). Además, es fundamental implementar mecanismos para mantener coherencia entre los valores en los *caches*, especialmente, en ciertos casos como el gasto de las campañas. ZMQ resultó ser una herramienta muy versátil para resolver este tipo de problemas.

En segundo lugar, no es necesario renunciar al *feature set* de una base relacional para poder tener baja latencia cuando se utiliza y configura correctamente una base madura como PostgreSQL. En general, en casos como el nuestro, las falencias de este tipo de bases de datos vienen de la mano del bajo *throughput* que pueden ofrecer para lecturas y escrituras, debido al *overhead* de las transacciones y la imposibilidad de escalar horizontalmente. El bajo *throughput* de escritura lo mejoramos realizando *buffering* de escrituras y agregaciones tempranas de los datos. En el caso de las lecturas, las mejoramos trabajando para optmizar al máximo nuestra estrategia de *caching*.

En tercer lugar, fue fundamental la implementación de un *stream* de datos sobre el cual se pueda hacer muestreo consistente. Ya que el volumen de eventos generado por el *bidder* es muy grande, a menudo para poder hacer análisis se necesita hacer muestreo de los datos y es fundamental contar con un canal donde los eventos se publiquen de forma constante y consistente[11] a medida que llegan.

Al final del día, la calidad de los algoritmos de optimización del *bidder* es lo que define si el proyecto es rentable o no. Para poder mejorar la calidad de los mismos, tuvimos que implementar todos los otros componentes para permitir un análisis efectivo de la información y poder entrenar modelos para aumentar la calidad.

[11] Es decir, que todos los eventos correspondientes a una misma secuencia (bid, impresión, install, etc.) se publiquen en la misma partición

Referencias

1. Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.
2. 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.
3. Ali Dasdan Kuang-Chih Lee and, Ali Jalali and. Real time bid optimization with smooth budget delivery in online advertising. *CoRR*, abs/1305.3011, 2013.
4. John Langford, L Li, and A Strehl. Vowpal wabbit. *URL https://github.com/JohnLangford/vowpal_wabbit/wiki*, 2011.
5. John Langford, Lihong Li, and Tong Zhang. Sparse online learning via truncated gradient. In *Advances in neural information processing systems*, pages 905–912, 2009.
6. H Brendan McMahan. A unified view of regularized dual averaging and mirror descent with implicit updates. *arXiv preprint arXiv:1009.3240*, 2010.
7. H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2013.