Zap: Making Predictions Based on Online User Behavior

Yuri Chervonyi, Dragos Harabor, Brian Zhang, Josh Sacks

{yuri*, dragos, brian, jjs}@bytegain.com

Abstract

This paper introduces Zap, a generic machine learning pipeline for making predictions based on online user behavior. Zap combines well known techniques for processing sequential data with more obscure techniques such as Bloom filters, bucketing, and model calibration into an end-to-end solution. The pipeline creates website- and task-specific models *without knowing anything about the structure of the website*. It is designed to minimize the amount of website-specific code, which is realized by factoring all website-specific logic into example generators. New example generators can typically be written up in a few lines of code.

Keywords: Customer intent, Audience segmentation, Online data, Sequential data, Deep learning

1 Introduction

As more people spend their days on the Internet, businesses have followed by exchanging their physical stores for websites and apps. However, in the process of moving online, many businesses have lost the personal touch they had with their customers. Instead, they serve generic experiences and random pop-ups. The irony is that the increase in online activity means that user web data[2] has evolved into a high-resolution reflection of a person's daily life. Businesses could use this highly textured online data to predict user behavior, provide recommendations and increase engagement. A recent study commissioned by the Digital Advertising Alliance (DAA) finds that over 40% of respondents would prefer personalized marketing based on their online behavior [1]. Many consumers have also experienced, and now expect, the magic of Google Search autocomplete, Netflix recommendations and more. Every business, not just big tech companies, could benefit from personalized experiences. Moving forward, every business, not just big tech companies, could benefit from personalized experiences.

Analyzing web data is a complicated problem and machine learning has become a standard method of solving such problems when heuristics become too complex. Machine learning becomes especially effective when a large amount of data is available in which case deep learning (for introduction see [5]) is almost a canonical approach producing state-of-the-art results.

Most modern deep learning models are based on artificial neural networks[3]. A neural network is a collection of simple elements called neurons. Neurons receive input, change their internal state (activation) and produce output based on the input and activation. The network is formed by connecting the output of certain neurons to the inputs of other neurons. The strength of connections between neurons is encoded via weights that can be modified by a process called learning.

*To whom correspondence should be addressed: yuri@bytegain.com.

[2]By web data we mean user action on websites or mobile applications.

[3]We will drop "artificial" from now on.

Preprint. Work in progress.

We believe that neural networks represent the beginning of a fundamental shift in software development [8]. In "classic" software development (Software 1.0), the programmer identifies a specific point in program space with some desirable behavior and expresses that by writing lines of code. In contrast, neural networks (Software 2.0) are written in a more abstract language, such as the weights of a neural network. The weights are not explicitly specified but learned via backpropagation and stochastic gradient descent.

Deep learning has made advances in solving problems that have previously resisted the best attempts of the machine learning community [10]. It also has turned out to be excellent in the discovery of intricate structures in high-dimensional data and therefore a natural candidate for analyzing web data. Finally, another advantage of deep learning is that it requires very little feature engineering and can easily take advantage of increases in the amount of available data and computation.

Making predictions based on web data is important for multi-billion dollar online industries, but there is little published research in this area. Tech giants such as Google, Facebook and Yahoo published their research some time ago [3, 7, 11] but it is now outdated. With Zap we had to re-think many aspects of data generation, data collection, feature engineering, applying machine learning models and serving predictions.

2 Data Processing

We start by reviewing the data collection procedure.

Analytics events such as page views, clicks, scrolls, positive outcomes (e.g. purchase), etc. are collected by a JavaScript library. Similar events from mobile applications are collected by a Mobile SDK. These libraries are comparable to Google Analytics[4] or Segment[5].

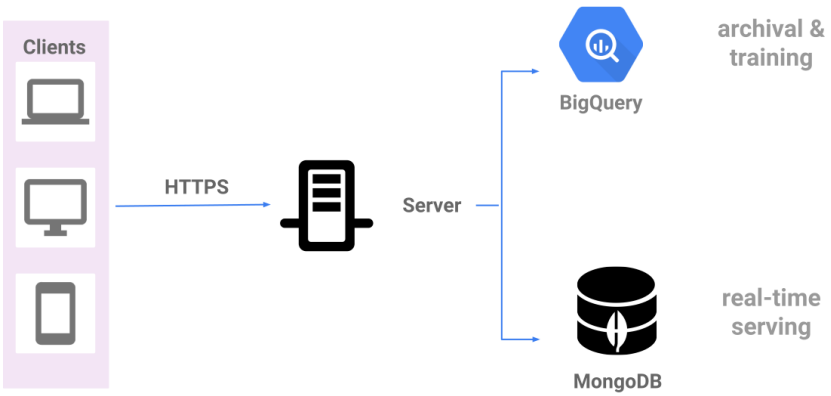


Figure 1: Data collection.

Collected data is posted in real-time to our server end-point, either event-by-event or in batches. Then the data is cloned into two streams: one that goes to our long-term archival solution (Google BigQuery), and another that goes to a fast database (such as MongoDB or Redis). The fast database is used for real-time predictions (see Fig. 1). Only a limited number of the most recent events and a subset of relevant event types are stored in the real-time database, just enough to be able to compute the predictions when the client queries for them. Long-term archival databases are used for training, but are not suited to real-time serving. Data from the real-time database is typically aged out after a few days, while long-term data is archived in a data warehouse.

Three key concepts used in our approach are *user sessions*, *instances* and *examples*. They will be described in details in the next two subsections.

[4] `https://www.google.com/analytics/`
[5] `https://segment.com/`

2.1 User Sessions

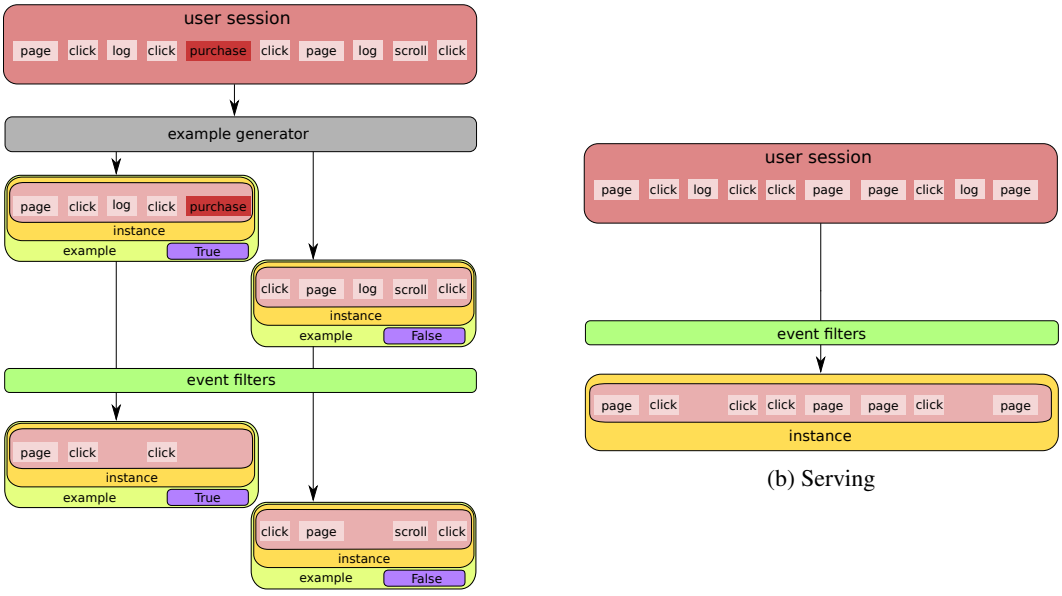
A *user session* is a stream of chronological events tracking the actions of a user that occur on a client device such as a web browser or mobile application. Each event has a `timestamp`, an `anonymousId`, a `userId` (if the user is logged in), and a payload describing the event. Some events are generated in response to a user action (e.g. page visit, click, scroll) while others may be synthetic events (e.g. a prediction point) automatically inserted by the client libraries or by the server.

Data preprocessing performed during serving and training are slightly different (see Fig. 2). First, we will describe the process used in training, then discuss how serving is different.

The first stage of our data preprocessing pipeline is transforming events from BigQuery (or another database) into user sessions. We run an Apache Beam job that maps rows from BigQuery to `anonymousId` and then performs a `GroupByKey` operation. Next we filter out events not generated by a real user, this includes events generated by scrapers, bots, our models, testing events, etc., and ignore long sessions[6]. Finally, forming user sessions is completed by ordering events by timestamps.

Next, we perform what can be thought as preliminary feature engineering: we remove duplicate information from web events, parse URLs and extract user agents. The importance of this step will be clarified in Sec. 2.3.

2.2 Instances and Examples



(a) Training

(b) Serving

Figure 2: Constructing instances and examples from user sessions in (a) training and (b) serving. User session consists of events such as page, click, log, scroll and purchase. Example generator picks several events (according to the desired logic), then event filters remove unwanted events such as log events or clicks/scrolls. For training, the example generator also generates a label (according to the desired logic) to form an example. In serving, the entire session at a given point in time is an instance, so the example generator is not used.

After user sessions are constructed, they are sliced into what we call prediction instances (see Fig. 2), which are basically fragments of user sessions generated by an *example generator*. An example generator defines points within the user session where predictions would have been made. For training, each instance is combined with a label to form an *example*, which is also done by an example generator (see Fig. 2a). In serving, the example generator is not used (see Fig. 2b).

[6] A large session is an arbitrary and a website specific term, we typically drop user sessions with more than 1000 events.

A procedure of constructing examples from a session is demonstrated in Fig. 2a. Depicted user session contains a purchase event, which we call a *positive event*. The example generator goes through the session looking for a positive event. As soon as it finds one (or the end of the session is reached[7]), the example generator creates a label. Then, *event filters* remove all irrelevant information or information which could lead to bogus predictions to form an instance. For example, predictive models could just map a purchase event to the actual purchase, but since the goal is to predict this event based on the data available before it happens, using this event is clearly a bogus behavior.

Finally, by attaching a label to the instance we get an example. Examples constructed in this way are then fed into machine learning models[8]. It should be emphasized that instance generators are the *only* website-specific part of Zap.

Along with instances, some of our machine learning models use *metadata*, which is a characteristic of the user or the session in general. This data is not of a sequential nature, so, as will be explained in Sec. 3, it is fed into the machine learning models separately (see Fig. 5a).

2.3 Feature Scaling and Instance Coding

The last stage of our data preprocessing pipeline is transforming instances and examples into representation suitable for our deep learning models: the input data should be a sequence of real valued feature vectors. Moreover, it is a good practice in general and particularly important for our models to perform feature scaling. We further restrict feature vectors to have values in $[0, 1]$-range. This is done because, as will be explained later in this section, we combine several different types of data such as strings and numbers into one feature vector (see Fig. 5b). If, however, some elements of this vector are not of the same scale, i.e. features with smaller numerical values, they might be suppressed by other features from a larger scale. That could lead to models converging to a worse local minima and, as a consequence, to less accurate model predictions[9].

We hash web data strings, e.g. URLs, user agents, into a real-valued feature vector that we call *hash buckets*[10]. Consider a set of strings S, which needs to be transformed into a feature vector v (which then will be fed into a neural network). We start by initializing v with zeros. Next, for $s_i \in S$ we apply a hash function h to s_i as follows $h_i = h(s_i), \bar{h}_i = h(s_i + \text{“}some\_fixed\_string\text{”})$, finally we set $v[h_i] = 1, v[\bar{h}_i] = 1$. Note that v should have enough capacity to avoid collisions. We empirically determined that for a typical website setting $\dim(v) = 100$ results in the collision rate of 10%. The sparsity in this case is approximately 80%[11]. This procedure leads to creation of a bit vector v. See Algorithm 1.

Algorithm 1 Hash Buckets Generation

Input: A set of strings S, a hash function h and an integer n.
Output: Vector v, such that $\dim(v) = n$ and $v_i \in [0, 1]$.
  Let $v = [0]^n$.
  for all $s_i \in S$ **do**
    Compute two hash values:
    $h_i = h(s_i), \bar{h}_i = h(s_i + \text{“}some\_fixed\_string\text{”})$.
    Set bits in v using h_i and $\bar{h}_i$ as indices:
    $v[h_i] = v[\bar{h}_i] = 1$.
  end for

For categorical data we use standard *one hot encoding*: if the number of possible values of a particular variable is limited to a fixed set of length m this variable can be represented as a bit vector, w (of length m). Categorical values should be mapped to integer values. Then each integer value is

[7] Slicing rules could be different and are specified by a programmer.

[8] Note that because we use neural networks as our machine learning models, raw web data has to be transformed into a useful representation before being fed into neural networks. This transformation is discussed in the next section Sec. 2.3

[9] Some advanced optimization techniques work better with non-scaled features. For example, Adam [9] maintains different learning rates for each network parameter and separately adapts them as learning unfolds.

[10] This approach is inspired by the Bloom filter [2].

[11] The detailed analysis of sparsity will be performed in the follow-up paper.

represented as a bit vector that is all zero values, except the index of the integer which is marked as one.

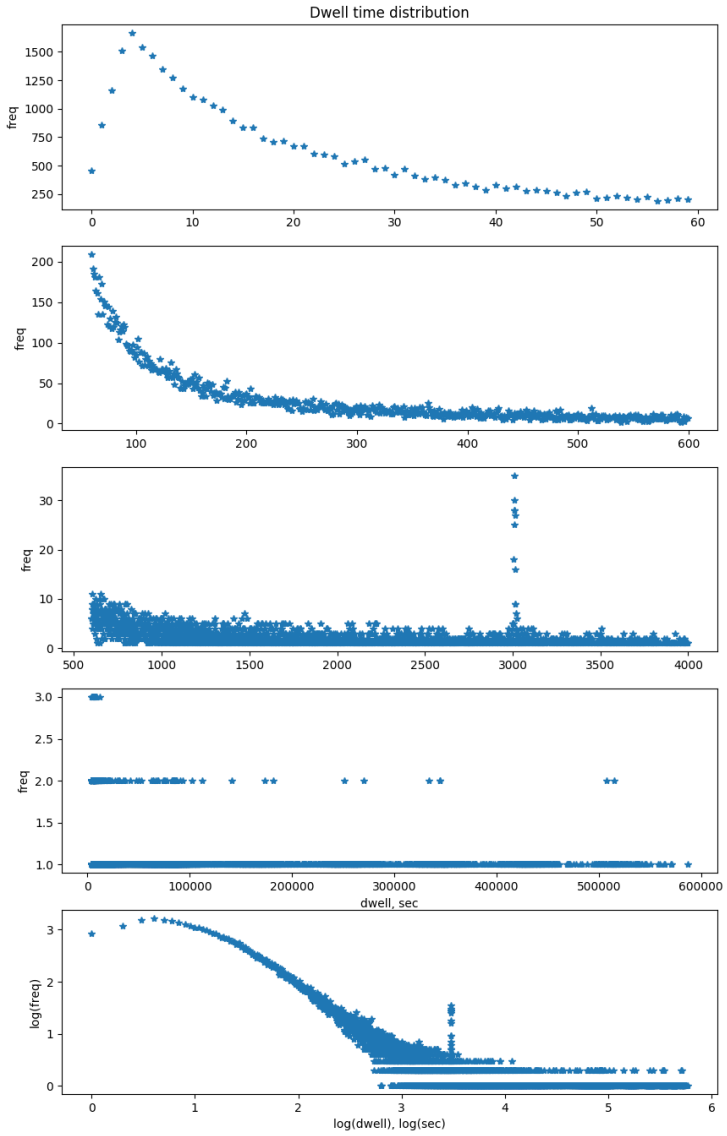


Figure 3: A typical dwell time distribution (distribution of time intervals between page loads measured in seconds). The peak at the 1 hour mark is an artifact due to the website refreshing pages after 1 hour of inactivity.

Along with text data we compute numerical features (such as dwell time between events), map them into $[0, 1]$-range and then concatenate to other features such as hash buckets, v, computed at the previous stage. To achieve it we use two different techniques: *normalization* and *data bucketing*

(binning)[12]. If the range of a numerical value, X is known, $X \in [X_{min}, X_{max}]$, one can simply apply normalization as

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}, \tag{1}$$

which ensures that $X' \in [0, 1]$. If the range is unknown or feature values are distributed non-uniformly, then data binning is preferred. A relevant example of a non-uniform distribution is showed in Fig. 3, where we plot a typical dwell time distribution (distribution of time intervals between page loads measured in seconds)[13].

Data binning is a form of quantization, in which original values that fall in a given small interval, a bin, are replaced by a value representative of that interval. In our pipeline numerical values are converted into bit vectors. For example, rounding a floating point number to an integer and then turning it into a one-hot vector is an example of *linear bucketing*.

Let's say our feature's current value is 42 and it is distributed in the range $[0, 100]$. If we want to bucket it into 11 bins its current value would be transformed into $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$, where the first bin corresponds to values in $[0, 10)$-range, the second one in $[10, 20)$ and so on. In this example all bins are of the same size (that is why they are called linear bins). A simple Python implementation is shown on Listing 1.

Listing 1: Linear bucketing with `numpy`.

```
1  def get_one_hot_bucket_vector(value, start, end, n_buckets):
2      step = (end - start)/(n_buckets - 1)
3      feature = numpy.zeros(n_buckets)
4      buckets = numpy.arange(start, end + 1, step)
5      index = numpy.digitize(value, buckets)
6      feature[index - 1] = 1
7      return feature
8
9  n_buckets = 11
10 start, end = 0, 100
11 value = 42
12 bucketed_feature = get_one_hot_bucket_vector(value, start, end,
       n_buckets)
```

This technique works well for features with uniformly distributed values as shown in Fig. 4a, but would lead to a big information loss when applied to variables with non-uniform distributions. A better solution is to have buckets whose sizes depend on the distribution density: it is desirable to have a higher resolution (which corresponds to having more buckets) near the distribution peak, and lower resolution away from the peak (see Fig. 4b).

Creating perfect buckets is not a simple task because one needs to know a functional form of the given distribution, which is almost never known a-priori, so for our tasks we assume a simple power law distribution to create more buckets near the peak and less at the tail. This approach is a good approximation for data with distributions such as depicted in Fig. 3. Essentially we combine linear and non-linear buckets. To do so we specify three more parameters: linear buckets step, s_l, linear buckets cutoff, c_l, and non-linear buckets cutoff, c_n. Then linear, n_l, and non-linear, n_n buckets are computed as follows:

$$n_l^i = s_l i, \qquad 0 \leq i \leq c_l, \tag{2}$$

$$n_n^i = s_n p^i, \qquad 1 \leq i \leq N - c_l, \tag{3}$$

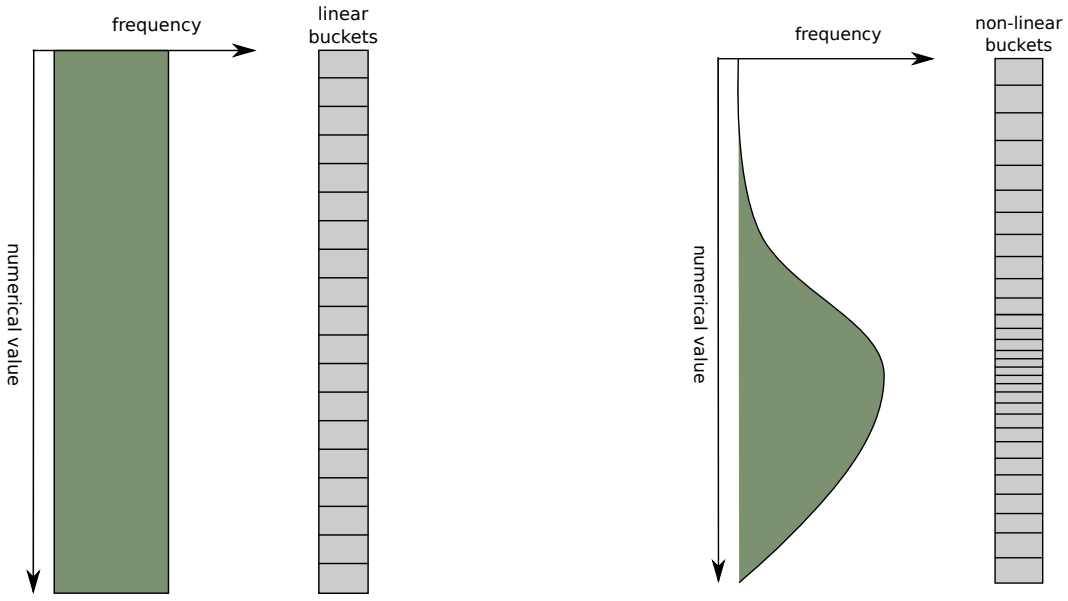
where s_n and p are determined from the following boundary conditions:

$$n_n^i|_{i=0} = c_l, \quad n_n^i|_{i=N-c_l} = c_n. \tag{4}$$

The first one is a continuity of linear and non-linear buckets, and the second one is the upper bound on non-linear buckets. Once buckets are generated, the procedure outlined in Listing 1 is applied (except line 3 is replaced with buckets constructed beforehand). We also use a more sophisticated technique that allows choosing the desired resolution for non-linear buckets.

[12] Terms "bucketing" and "data binning" will be used interchangeably in this article.

[13] Another example of dwell time distribution can be found in [4].

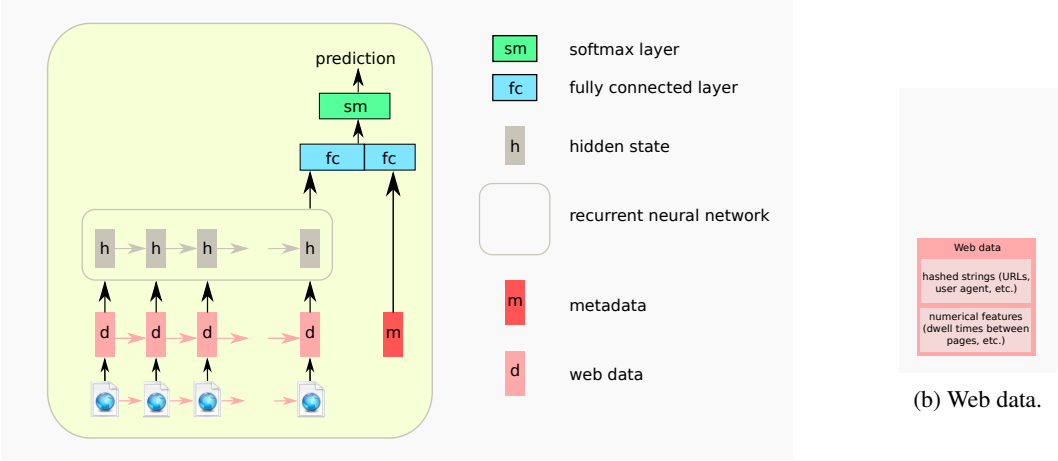


(a) Linear buckets

(b) Non-linear buckets

Figure 4: Pictorial representation of linear and non-linear buckets. (a) For uniformly distributed data, using buckets of the same size is a good quantization method without much information loss. (b) If data distribution is non-linear, linear buckets would result in heavy information loss because most values (which are close to the distribution peak) would be assigned the same value. In this case a better approach is to use buckets of different size, larger buckets near the tails and smaller buckets near the peak.

3 Predictive Models



(a) RNN model architecture.

(b) Web data.

Figure 5: RNN model architecture and data. (a) Our model uses an RNN module to process sequential web data (e.g. page visits, clicks, scrolls, etc.) and several fully connected layers to process metadata, which characterizes a user session as a whole. (b) Each element of a user session is a combination of hashed strings and numerical features.

After preprocessing, the data is fed into predictive models. Our models consist of two parts: one processing sequential data and another one processing metadata. The sequential data is processed

via a recurrent neural network (RNN) or a convolutional neural network (CNN). In this paper we focus on the RNN architecture. A CNN approach will be described in a follow-up paper.

The RNN architecture[14] is depicted in Fig. 5a. The metadata is processed via several fully-connected layers, which is then merged with the output of RNN and passed to the softmax layer. Note that each element of sequential data is obtained by concatenating hash buckets with buckets for numerical features such as dwell time (see Fig. 5b), and metadata is fed separately (see Fig. 5a).

We train our models with the Adam optimizer [9] and use L2 and dropout [14] for regularization. Another important component of our training approach is using cross entropy weighting [12]. The datasets we work with are highly unbalanced due to the nature of problems we deal with: positive events could be very rare (e.g. $< 1\%$), which makes training difficult. Weighting cross entropy allows one to trade off recall and precision by up- or down-weighting the cost of a positive error relative to a negative error. Another way to deal with highly unbalanced datasets is to use negative down-sampling. However since we also want our predictions to reflect the actual probabilities (as will be explained in Sec. 7.1) we do not use negative down-sampling.

Finally, to find the best model we do a hyperparameter search over L2, dropout and the number of RNN units.

4 Real-time Serving of Trained Models

As mentioned in Section 2, incoming client data is cloned into two streams: one for long term archival and training, another one for real-time serving, stored in MongoDB (see Fig. 1).

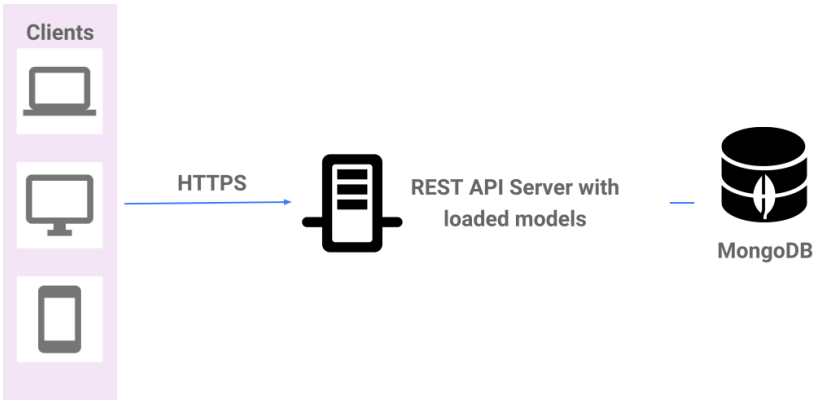


Figure 6: Client API call to retrieve a prediction. The server queries MongoDB for events and inferences using frozen models loaded in memory.

Model predictions are served via a REST API. The client making the API call can be a web application (JavaScript) or a mobile application. When a request comes in, the list of most recent events for the given user ID is pulled from MongoDB, ordered by event timestamp. The size of the list depends on the model fixed input size, e.g. 40 most recent events (if not enough events are available, the list is padded with blank events up to the fixed size). The events are turned into an instance and run through the currently active model version to generate a prediction. The prediction value is returned to the client application which can then take an action based on the value. This procedure is demonstrated in Fig. 6.

For each application, there is a current version of the model loaded in the server memory. The models are stored and loaded in memory in a TensorFlow frozen form which speeds up loading and reduces memory footprint.

[14] As a recurrent unit we use Gated Recurrent Unit (GRU), which showed the best performance in our tests.

5 Re-training Models

Since our models make predictions for a particular website or app, it is important to keep the models up-to-date with the current version of that website or app.

New model versions are re-trained automatically on a periodic basis, e.g. weekly. This ensures that the latest collected data is used and models take into account the most recent changes to application structure and user behavior. There is an automated process that goes through the following steps every week:

- Re-trains a new model for each application as described in the previous sections. A rolling window of most recent data is used, e.g. 7-60 days, depending on how much data each application produces.
- Validates that the new model is acceptable by checking certain performance metrics such as AUC (see more about metrics used in Section 8) vs. previous model versions in the same family.
- Archives the new model version together with stats about it and the examples generated during training (see Section 2.2). This allows reverting back to an older version of a model if issues are discovered in production, provides a detailed history of how the model evolved and allows comparing different versions and their stats.
- Verifies that the new model will be served correctly. During training, a random subset of examples (e.g. 1000) is saved together with the computed prediction value for each example. The verification consists of running these examples through the exact code path used for real-time serving with the expectation that the served prediction value exactly matches the value computed during training. This ensures that the same common code is used both in training and serving (see Fig. 2).
- If validation and verification pass, the new model version is auto-deployed to production and new predictions are served using the new version.

6 Post Serving Analysis

During serving, each prediction made by a model is logged, together with sufficient metadata that can identify the model version and the events used to construct the instance. This allows for later analysis of actual model performance by looking at what the model predicted for a user over time vs the actual action the user performed.

7 Confidence and Calibrating Predictions

In our applications, it is important to classify user behavior, but it is also important to quantify the expected accuracy of the prediction. This problem of predicting probability as a true correctness likelihood is known as *confidence calibration* (for recent developments see [6]). The importance of calibration is also discussed in [11, 7]. A rough estimate of how well a model is calibrated is given by *expected calibration error* (ECE). In order to compute ECE[15], we group predictions into M interval bins (each of size $1/M$) and calculate accuracy/confidence in each bin. Let B_m be the set of indices of samples whose prediction falls into the interval $I_m = (\frac{m-1}{M}, \frac{m}{M}]$, the accuracy is then

$$\mathrm{acc}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} \mathbf{1}(\hat{y}_i = y_i), \tag{5}$$

where $|B_m|$ is the number of elements in B_m, and $\hat{y}_i$ and y_i are the predicted and true class labels for sample i. The confidence is defined as:

$$\mathrm{conf}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} \hat{p}_i, \tag{6}$$

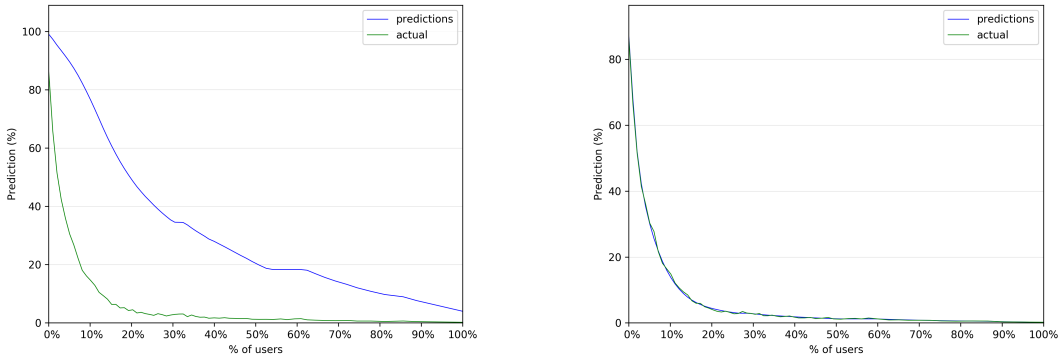
[15]Here we follow [6].

where $\hat{p}_i$ is the predicted probability. Finally, the expected calibration error is

$$ECE = \sum_{m=1}^{M} \frac{|B_m|}{n} \left|\mathrm{acc}(B_m) - \mathrm{conf}(B_m)\right|, \tag{7}$$

where n is the number of samples. A perfectly calibrated model will have $\mathrm{acc}(B_m) = \mathrm{conf}(B_m)$ for all $m \in M$, so $ECE_{perfect} = 0$.

ECE gives only a rough estimate of how well a model is calibrated. To get a better understanding, we visualize predictions and actual outcomes by plotting a *calibration curve*. We start by sorting training examples by probability from high to low, then we put them into buckets of different size such that each bucket has at least 100 positive outcomes (to minimize noise and maximize resolution). Next, we compute confidence (6) and the ratio of actual positive outcomes to the total number of examples in each bucket. Confidence forms a "predictions" curve, and the ratio of positive outcomes forms an "actual" curve. An example of a calibration curve is depicted in Fig. 7a. We use this graph as a performance metric as will be explained in Sec. 8. Note that for a perfectly calibrated model the two curves should coincide (see Fig. 7b).



(a) Uncalibrated model.

(b) Calibrated model.

Figure 7: Examples of calibration curves.

We observe that our models turn out to be miscalibrated, a typical calibration curve looks like in Fig. 7a. Possible causes of miscalibration include big model capacity and lack of regularization[6]. An investigation into our applications shows that cross entropy weighting is the main source of miscalibration. This is because we usually deal with highly unbalanced datasets.

7.1 Recalibration

Since our models turn out to be miscalibrated we perform a *post-processing calibration*. There are several methods of doing so (as reviewed in [6]), but we found that the best results were obtained by using the *matrix scaling* method. The idea here is to apply a linear transformation to the logit vector, z_i, produced before the softmax layer for input x_i. Then the output of a new softmax layer, $\hat{q}_i$, is:

$$\hat{q}_i = \max_k \sigma_{SM}(\mathbf{W}\mathbf{z_i} + \mathbf{b})^{(k)}. \tag{8}$$

The parameters $\mathbf{W}$ and $\mathbf{b}$ are optimized with respect to the error function on a half of the validation set used for training, another half is used for the final validation. We use standard gradient descent with a learning rate decay. Fig. 7a shows a calibration curve for an uncalibrated model, the result after applying the calibration procedure it depicted in Fig. 7b.

8 Evaluation of Model Performance

Most of our models are binary classifiers, so we use AUC (area under the ROC curve) as our main performance metric. We also found useful two more metrics, distribution of predictions and calibration curve.

Since our models are used to categorize users into several segments for a subsequent action (such as showing users different context, ad retargeting, etc.), we can use the relative distribution of users between different segments as another performance metric. To construct segments we first compute the average positive action rate, then we sort predictions from high to low, split them evenly into several[16] buckets and compute a positive action rate in each bucket, finally we plot buckets and print relative odds of a positive action in each bucket. The odds ratio plot reflects how likely the average user in each bucket is to perform a positive action with respect to the base rate. A typical odds segments plot is depicted in Fig. 8. It shows that this model puts users who are 3.39 times more likely to perform a positive event than an average user into a "very high" bucket, the next bucket has users who are 0.98 times more likely to perform a positive events, and so on. The ideal model would put all users who generated a positive event into the first bucket[17] and the worst model would distribute users into the buckets evenly.

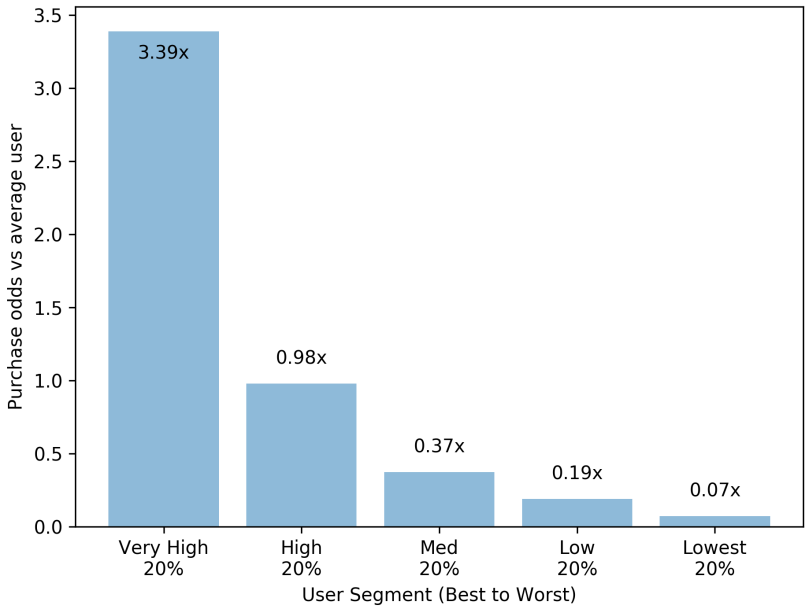


Figure 8: Our predictive models split users into segments with respect to how likely they perform a "positive" action such a purchase. This chart shows that among all users who performed a "positive" action our model was able to put users who are 3.39 times are more likely to purchase than an average user into the first bucket.

Finally, as another performance metric we use calibration curve described in Sec. 7. Examples of an uncalibrated and calibrated models are given in Fig. 7a and Fig. 7b respectively.

9 Interpretability of Predictions

Interpretability of machine learning models is critical for many practical applications. First of all, it ensures that the model is aligned with the problem one wants to solve. Second, it helps indicate patterns the model actually found in the data so one can use the model's predictions more effectively.

One important example comes from using machine learning in health care: if a model makes a prediction that a patient is at a high risk of an illness, it is very important to help the doctor understand why before ordering expensive or risky treatment. Consider another example, if a model identified that a user is going to end her subscription for an online service, it would be very helpful to know why and be able to improve the service quality without having to explicitly ask for feedback.

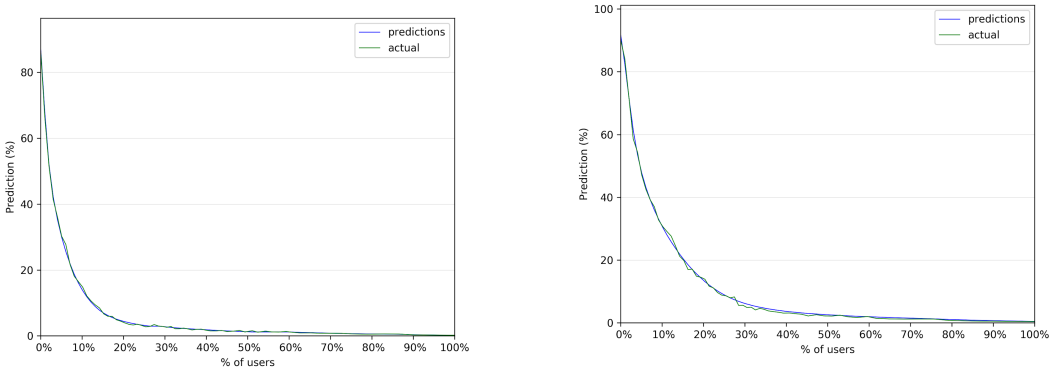
[16] For our purposes, we split users into 5 segments.

[17] Unless more than 20% of users generated positive outcomes, in which case the second bucket will be also used.

It is well-known that deep learning models are hard to interpret due to the large number of parameters and the complex approach to extracting and combining features. As this class of models is able to obtain state-of-the-art performance on a wide variety of tasks, more research is focused on linking model predictions to the inputs. Some progress has been done in computer vision where the gradients of the target concept calculated in a backward pass are used to produce a map that highlights the important regions in the input for predicting the target concept (e.g. see [13]). This way, one can actually see what a model is focused on when making predictions. In our case, it is not applicable, so we follow another path.

Our method is more manual and requires some intuition about the problem at hand and the data in general. The idea is the following: while generating examples, we compute statistics that we think are important and store them in each example. Then we make hypotheses and use our performance metrics to check them. This method is inspired by feature importance analysis, where one only uses a subset of features to see which one is the most important. In the case of deep learning, it is not practical since it would require expensive re-training after each choice of features.

Let's demonstrate the concept via a simple example. We have a website with some defined positive event, such as a purchase, and we want to confirm that our model makes reasonable predictions. Among all the features fed into our model, let's say we assume that the number of clicks and the time spent on the website are most important. To explore this conjecture, we plot calibration curves for users with different number of clicks and different time spent on the website, and compute the positive rate for top 10% predictions. It turns out that the positive rate in top 10% predictions for *all* users is 0.4, while for users with more than 10 clicks who spent more than 1 minute of the website the positive rate turns out to be 0.6. We conclude that the model works according to our intuition. The calibration curves are depicted in Fig. 9. Of course, the model's logic is more complicated than that, but its interpretability is still an active area of research.



(a) All users.

(b) Users with more than 10 clicks who spent more than 1 minute on the website.

Figure 9: The meaning of predictions is extracted from calibration curves. Calibration curves for all users (a) and for users with more than 10 clicks who spent more than 1 minute on the website (b) demonstrate that the model is more confident about the latter segment of users, which means that the number of clicks and the time spent on the website are important features as expected.

10 Use Cases

Zap is a general pipeline for predicting online user behavior. As a predictive analytics tool, we believe it has applicability to a wide range of problems. Zap can answer questions about customer intent and audience segmentation. How likely is this user to purchase? How likely is this user to return to the site? Which users are the most valuable long-term user?

Here are a few examples of Zap running in production to solve real-world problems:

1. Improve ads retargeting by segmenting customers based on likelihood to purchase.

2. Identify customers most likely to join an email list. A model is being used to target only high propensity user to avoid spamming all website visitors. This model also increases sign-up rates by identifying the best time to ask the user.
3. Identify the most "in-market" segment of customers. This is used to target sales effort on the most promising leads.

We have found that these production models often exhibit a 20/80 type of performance - identifying the 20% of users who account for 80% of a desired behavior. This performance drives big improvements in the task, whether it is ads ROI or email sign up rates.

11 Conclusions

In this paper we described Zap, a machine learning pipeline that makes predictions based on online user data. We designed Zap to process enterprise-level data in order to produce individual predictions that are 1) consumable through an API and 2) understandable to humans. The Zap platform goes beyond well-defined academic examples such as image recognition: it turns a marketing or sales question into an actionable decision. Furthermore, it provides an automation tool that collects the data and creates the decision algorithm based on that data. This is just the first step in building commercial-grade machine learning applications.

Zap is heavily based on Google Cloud Platform technologies such as BigQuery, Cloud Storage, Cloud Dataflow, Cloud Machine Learning, as well as other tools such as MongoDB. Big data is processed via Apache Beam and the machine learning component is written in TensorFlow. The key feature of Zap is that one can create and serve machine learning models for different websites by specifying example generators which typically consist of a few lines of code. The data is generated by our JavaScript library and then cloned into two streams: BigQuery for a long-term archival and MongoDB, which is used to serve real-time predictions. To train a new model one uses data from BigQuery, forms user sessions and then examples (which are generated by the only website-specific code in our pipeline called example generators). The examples are then fed into a deep learning model, which is trained on Cloud Machine Learning Engine. Once a model is trained it can be used to serve real-time prediction using the data coming from MongoDB.

Acknowledgments

The original Zap pipeline was implemented by Josh Sacks and Chris Atenasio. We also thank Mark Vandevoorde and Mike Chu for valuable discussions and feedback on this work.

References

[1] Zogby Analytics. Consumers say they prefer targeted to random online ads. `https://www.marketingcharts.com/digital-28825`, 2013.

[2] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] Haibin Cheng, Roelof van Zwol, Javad Azimi, Eren Manavoglu, Ruofei Zhang, Yang Zhou, and Vidhya Navalpakkam. Multimedia features for click prediction of new ads in display advertising. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 777–785. ACM, 2012.

[4] Alexander Dallmann, Alexander Grimm, Christian Pölitz, Daniel Zoller, and Andreas Hotho. Improving session recommendation with recurrent neural networks by exploiting dwell time. *arXiv preprint arXiv:1706.10231*, 2017.

[5] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[6] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. *arXiv preprint arXiv:1706.04599*, 2017.

[7] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9. ACM, 2014.

[8] Andrej Karpathy. Software 2.0. `https://medium.com/@karpathy/software-2-0-a64152b37c35`, 2017.

[9] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[10] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436, 2015.

[11] 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.

[12] Sankaran Panchapagesan, Ming Sun, Aparna Khare, Spyros Matsoukas, Arindam Mandal, Björn Hoffmeister, and Shiv Vitaladevuni. Multi-task learning and weighted cross-entropy for dnn-based keyword spotting. In *INTERSPEECH*, pages 760–764, 2016.

[13] Ramprasaath R Selvaraju, Abhishek Das, Ramakrishna Vedantam, Michael Cogswell, Devi Parikh, and Dhruv Batra. Grad-cam: Why did you say that? *arXiv preprint arXiv:1611.07450*, 2016.

[14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.