Multiworld Testing Decision Service: A System for Experimentation, Learning, And Decision-Making

Alekh Agarwal Sarah Bird Markus Cozowicz Miro Dudík Luong Hoang
John Langford Lihong Li Dan Melamed Gal Oshri Siddhartha Sen
Alex Slivkins*

First version: June 2015
This version: July 4, 2016

Abstract

Many applications are continuously faced with decisions to make, often based on some contextual information. We create a machine learning system for making such decisions, which supports the full cycle from exploration to logging to training policies (decision rules) to deploying them in production. The system is general: it works for any discrete choices, with respect to any immediate reward metric, and with many machine learning algorithms and feature representations. Implemented as a cloud service, the system has a simple API, and is designed to be modular and reproducible to ease deployment and debugging, respectively.

At the core of the system is a machine learning methodology for principled and efficient experimentation, based on the research on contextual bandits. In particular, we implement *Multiworld Testing* (`MWT`): the capability to test and optimize over K policies using an amount of data and computation that scales *logarithmically* in K, without necessarily knowing these policies before or during data collection. With this capability, one can answer exponentially more detailed questions compared to traditional A/B testing.

The provided evaluation shows that the system makes decisions in real time and incorporates new data quickly into learned policies. A large-scale deployment for news site `MSN.com` has been handling all traffic since January 2016, resulting in a 25% relative lift in clicks.

About this document. This document is a white paper for a broad technical audience, both in product groups and in research. The primary purpose is to provide an 'entry path' and background for potential users of the system. The document covers the machine learning methodology and systems design, as well as deployments and experimental evaluation. It does *not* cover precise instructions and API specs; those can be found in the documentation [28].

This version is a substantial revision and expansion compared to the initial version (which has been available online since June 2015). It heavily overlaps with a recent research manuscript [1] in terms of material, but covers more ground (esp. in terms of machine learning methodology), and is designed with more emphasis on potential users.

*All: Microsoft Research. Author emails: {alekha, slbird, marcozo, mdudik, jcl, lihongli, lhoang, melamed, gal.oshri, sidsen, slivkins}@microsoft.com.
Contact email for MWT: mwtservice@microsoft.com.
Project webpage: `http://research.microsoft.com/mwt/`.
MWT Decision Service: `http://aka.ms/mwt`.

Contents

1 Introduction

We create a "complete" machine learning system, in the sense that it supports the full cycle from exploration to logging to training policies (decision rules) to deploying them in production. The system can plug into an application that interacts with an environment, such as a website that interacts with users, a gaming service that interacts with gamers, or a data center that interacts with job requests. The system helps optimize such application so as to make these interactions more productive according to some observable, application-specific reward metric such as the number of clicks on a website or the waiting time of a job in a data center.

Our running example is optimization of a news website. While it is based on a successful product deployment, we ignore some real-life complications for the sake of clarity. Thus, let us consider a very simple news website, henceforth called `SimpleNews`, which displays exactly one headline to each user. When a user arrives, some information is available pertaining to the interaction with this particular user, *e.g.,* age, gender, geolocation, time of day, and possibly much more; such information is summarily called the *context.* The website chooses a news topic (*e.g.,* politics, sports, tech, etc.), possibly depending on the context, and the top headline for the chosen topic is displayed. The website's goal is to maximize the number of clicks on the headlines that it displays. For this example, we are only concerned with the choice of news topic: that is, we want to pick a news topic whose top headline is most likely to be clicked on for a given context.

We target applications such as `SimpleNews` in which interactions follow a simple template:

- observe a *context*: *e.g.,* the user-specific information,
- take an *action*: *e.g.,* choose a news topic,
- observe an *outcome*: *e.g.,* a click or no click and the time spent on the page.

The outcome should meaningfully depend on both context and action, and determine the numerical reward. The outcome can be, and typically is, very *noisy*: *e.g.,* in `SimpleNews` we only observe whether there was a click, rather than the underlying click probability.

An application such as `SimpleNews` is governed by a *policy*: a function that inputs a context and outputs an action. Many such policies exist: to wit, the number of possible policies is exponential in the number of possible contexts (which is often a very large number in itself).[1] Some policies may perform better than others, in terms of the average number of clicks that they accumulate. One may be able to hand-pick a reasonable "default policy": *e.g.,* among sports news, show hockey news to users with Canadian IPs, and cricket news to users with Indian IPs. However, much better policies may exist, and one typically needs experimentation to discover them. Indeed, if one only uses the default policy, without experimentation, then logging and data analysis alone does not suffice, as it cannot answer the counterfactual questions: what would have happened if the website chose a different action than the one prescribed by the default policy? For example, a Canada-based user who visits Indian websites may be an Indian ex-pat who loves cricket.

A standard methodology for experimentation, called *A/B testing* [41, 42], randomly assigns users into one of the several buckets, one bucket per each policy being tested, and records the average reward for each bucket. However, this methodology has several important drawbacks. It cannot handle many policies since the required amount of data scales linearly with the number of policies. It also cannot adjust over time to avoid wasting resources on low-performing alternatives. Furthermore, all policies need to be known in advance to test on live users.

[1] The number of feasible policies tends to be huge even though in practice the policies are usually restricted to have some "nice" shape such as a decision tree or a neural net.

We use a different machine learning methodology which remedies all these drawbacks: it can test and optimize over exponentially more policies with the same amount of resources, and these policies do not even need to be known in advance. In particular, the policies being tested do not need to be approved and implemented in production, thus saving much administrative and engineering effort. We refer to this dramatic improvement in capability as *Multiworld Testing* (MWT). Correspondingly, our methodology is called "the MWT methodology". The key insight which enables MWT is that each data point can be used to evaluate all policies picking the same action for the same context. (Whereas with A/B testing, each data point is only used for a single "bucket".)

Another key insight in the methodology is an explicit reduction to conventional machine learning. While we deal with experimentation and noisy data, deep down we need to tackle the problem of training a policy given a fixed, noiseless dataset. In other words, imagine a simplified problem when all experimentation is over, each data point includes rewards of all possible actions, and there is no noise in the rewards; for example, if one knows the click probability of every news topic for every user. Even then, training a policy is a deep and non-trivial problem. Fortunately, it is a well-studied problem in machine learning, known as *cost-sensitive classification* (see Section 3.3 for more details). Rather than re-invent solutions for this problem in more complicated scenarios, the MWT methodology internally uses an algorithm for cost-sensitive classification. In fact, it can be an arbitrary algorithm for this problem, allowing us to benefit from prior art and to use algorithm libraries such as VW [68]. The choice of an algorithm for cost-sensitive classification implicitly determines the class of policies considered by MWT: essentially, these are all policies that can be trained by this algorithm. A typical algorithm focuses on policies that follow a specific template, such as a linear classifier, a decision tree, or a neural net.

The methodology consists of four components:

Exploration Choose an action for each arriving context so as to collect useful information about the policies being tested. Typically one randomizes among the available actions. The randomization need not be uniform: it can give preference to the action prescribed by the default policy (*e.g.,* the policy currently used in production), and/or to actions that are likely to perform well based on the observations so far.

Logging Record the data points collected during exploration. In SimpleNews, a data point describes the interaction with a single user, including the context, the action (news topic) chosen for this context, and the outcome of this action, i.e., whether there was a click. The data point also includes the probability assigned to the chosen action by exploration.

Policy learning Train an approximately optimal policy based on the data collected so far. Also, *evaluate* a given policy, i.e. to approximately predict user response if this policy were deployed. In SimpleNews, the quantity to be predicted is the average number of clicks. The policy can be suggested by the system's designers or by MWT itself.

Policy deployment Deploy an improved policy found by the policy learning stage.

Taken together, these four components form a virtuous cycle (Figure 1). Indeed, exploration can continue after an improved policy is deployed, so as to collect more information and find an even better policy. Moreover, exploration can take into account the improved policy and become more efficient, thereby limiting the potential harm to user experience.

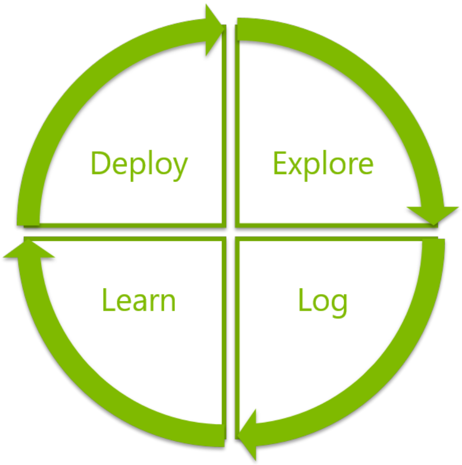

Figure 1: The virtuous circle: from exploration to logging to policy learning to policy deployment.

Implementing MWT runs into a number of technical issues. How do you record the correct action and context for learning? When rewards arrive with a considerable delay after the actions are chosen, how do you *join* with the corresponding context and action? And how do you do it with high throughput? How do you learn from the data and deploy a model quickly enough to exploit the solution? And how do you fit all these pieces together into a single easy-to-use system? Resolving these issues for a particular application requires a substantial amount of new programming *and*, typically, substantial new infrastructure. In our experience with several product teams at Microsoft, this is a complex and subtle task, well beyond the capability of most teams.

Our experience suggests that the best (and perhaps the only feasible) way to implement MWT is via a unified system suitable for many applications. In particular, many of the the pertinent issues are better addressed once and for all, rather than re-developed for each application. We present a general modular design for the system, incorporating all four components of the cycle in Figure 1. We define the functionality of the components and the ways they interact with one another. Modularity allows implementation details to be customized to the infrastructure available in a given organization. Using this design, we build a cloud service called the *MWT Decision Service*, which includes both the machine learning algorithms and the supporting infrastructure. The system consists of several modules which can be used jointly or separately depending on application-specific needs. The system is fully functional and publicly available.

We deployed the Decision Service in production to recommend personalized news articles on a heavily-trafficked major news site at MSN.com. All previous machine learning approaches failed in this setting, yet the system provided a 25% relative lift in click-through rate.

Relation to research. The machine learning problem in the SimpleNews example is known as *contextual bandits* (among other names). The MWT methodology synthesizes ideas from a particular line of work on contextual bandits (*e.g.,* [3, 5, 29, 43, 44]) and policy evaluation therein (*e.g.,* [30, 31, 45, 46]). Reducing policy optimization to a supervised learning problem, a key idea in this line of work, is one of several useful "learning reductions" [20], whereby a more complicated machine learning problem is solved using an algorithm for a simpler one. The system design for the Decision Service has not yet appeared in a peer-reviewed academic publication.

2 Getting Started: How to Deploy MWT for Your Application

Before we describe how the MWT methodology and the Decision Service work under the hood, let us discuss how to start using them: how to model your machine learning problem so as to make the MWT methodology suitable for your application, how to assess the scale and feasibility of this problem, and (on a high level) how to use a simple version of the Decision Service. The application system — the system to which the Decision Service is applied — is denoted by APP.

Modeling the problem. On a conceptual level, the interactions between APP and its users should be interpreted as a sequence of small interactions with individual users, possibly overlapping in time. Each small interaction should follow a simple template: observe a *context*, make a *decision*, choosing from the available alternatives, and observe the *outcome* of this decision. The meaning of contexts, decisions and outcomes should be consistent throughout. As a matter of terminology, an interaction fitting this template is called an *experimental unit*. APP's objective is to optimize the decision for a given context so as to bring about the most desirable outcome.

One needs to decide what contexts, decisions, and outcomes mean in your APP:

- The *context* should encompass the properties of the current user and/or task to be accomplished, and must be known to APP. It can describe complicated objects such as a webpage (*e.g.,* to place an ad on). The context is typically represented as a feature vector.
- The *decision* must be controlled by APP. Typically the decision is to choose an *action* among a set of feasible actions. (However, in some applications the decision can have a more complicated semantics, such as choosing a *slate* of actions.) The set of feasible actions should be known: either fixed in advance or specified by the context. It is often useful to describe actions with features of their own, a.k.a. *action-specific features.*
- The outcome consists of one or several events, all of which must be observable by APP not too long after the action is chosen. The outcome (perhaps jointly with the context) should define a *reward*: the short-term objective to be optimized. The total reward collected over time should provide a good proxy for the "true" long-term objective of APP, such as long-term user satisfaction.

Once the model is chosen, one should be able to fill in Table 1.

	SimpleNews	Your APP
Context	(gender, location, time-of-day)	
Decision	a news topic to display	
Feasible actions	{politics, sports, tech, arts}	
Action-specific features	none	
Outcome	click or no click within 20 seconds	
Reward	1 if clicked, 0 otherwise	

Table 1: Modeling the problem (and using SimpleNews as an example)

There can be multiple possible ways to define these notions for a given APP: to wit, which features to include in the context and (if applicable) in each action's feature vector, and how to represent these features numerically; what exactly is the decision that we want to optimize; which observations to include in the "outcome" and how to synthesize the "reward" from these observations. Such issues arise in many applications of machine learning; they are usually application-specific, and do not have generic, easy-to-describe solutions. We describe how these issues are resolved in a specific application in Section 6.

Consider SimpleNews to illustrate how the semantics of the decision, outcome and reward can be non-trivial. For example, SimpleNews can be modified so that actions correspond to news articles, rather than news topics, and the decision in each experimental unit consists of choosing a slate of news articles. The outcome might distinguish between clicks on different articles in the slate, and can include other information such as dwell time. The reward might be a complicated function of all pieces of information included in the outcome.

Scale and feasibility. To estimate the scale and feasibility of the learning problem, one needs to estimate a few parameters: the number of features (#features) and the number of feasible actions (#actions) in a typical experimental unit, a typical delay between making a decision and observing the corresponding outcome, and *the data rate*: the number of experimental units per second (say). If the outcome may include a rare event whose frequency is crucial to estimate — e.g., clicks are typically rare compared to non-clicks, — then we also need a rough estimate of this frequency.

Finally, we need the *stationarity interval*: the time interval during which a typical experimental unit does not change too much. We are interested in several properties: which contexts are likely to arrive, what are the feasible actions, and what are the expected rewards. More specifically, two things should not change significantly during the timescale: the distribution of arriving contexts (where a context includes the set of feasible actions and, if applicable, their features), and the expected rewards for each context-action pair. In the SimpleNews example, this corresponds to the distribution of arriving user profiles and the click probability for the top headline (for a given news topic, when presented to a typical user with a given user profile).

To summarize, one should be able to fill in the last column in Table 2.

	SimpleNews	Your APP
#features	3	
#actions	4 news topics	
Typical delay	5 sec	
Data rate	100 users/sec	
Rare event frequency	typical click prob. 2-5%	
Stationarity interval	one week	

Table 2: The scalability parameters (using SimpleNews as an example)

A good rule-of-thumb is that for a successful application of MWT the stationarity interval should be much larger than the typical delay, and moreover we should have

$$\texttt{Timescale} \times \texttt{DataRate} \times \texttt{RareEventFreq} \gg \#\texttt{actions} \times \#\texttt{features} \tag{1}$$

The left-hand side is simply the number of rare events in the timescale, and the right-hand side characterizes the complexity of the learning problem. This rule is for the default linear representation of policies, with other representations potentially requiring more or less events while yielding more or less benefit.

Web API. A simple usage mode for the Decision Service is provided by the web API.[2] The APP communicates with the Decision Service via HTTP, with message body formatted in JSON. Each experimental unit is handled by three HTTP messages: APP sends the context to the Decision Service, the Decision Service returns the action to be taken, and finally APP reports the reward. For a simple (and somewhat stylized) example:

1. APP sends context to the Decision Service:
   `https://{hostname}/api/decision?defaultAction=5`
   It is an HTTP POST request whose body contains the context:
   `{"FeatureName1":"FeatureValue1", "FeatureName2":"FeatureValue2"}`

2. The Decision Service responds with an action (*e.g.,* which article to show):
   `{"Action":2, "EventId":"XXX", "TimeStamp":"YYY"}`
   The action and the auxiliary information are contained in the body of the HTTP response.

3. APP returns the reward (*e.g.,* 1=click), using provided event ID and time stamp:
   `https://{hostname}/api/reward?reward=1&eventId=XXX&timestamp=YYY`

APP should take the action returned by the Decision Service, and faithfully report back the reward. Once a reward is reported, the experimental unit is complete and the corresponding data point tuple can be used by the Decision Service for policy learning, resulting in recommendations improving over time. The "default action" in the first HTTP request represents APP's default choice in the absence of exploration.

APP should be "registered" with the Decision Service, a painless process which should only take a few minutes. In terms of access rights and billing, the only thing needed is an account with Azure [12]. This is where APP's data is stored, and where the storage/CPU usage is charged to. One also needs to specify a few configurable options; choosing defaults, where applicable, is usually OK. Billing is handled automatically, according to the pricing scheme associated with the Azure account. Free trial and academic accounts are currently also available.

3 Machine Learning Methodology for MWT

Multiworld Testing (MWT) refers to the capability to test and optimize over K policies, using an amount of data and computation that scales *logarithmically* in K, without necessarily knowing these policies before or during data collection. This section overviews the machine learning methodology for implementing MWT (henceforth, *MWT methodology*). The methodology emphasizes modularity: exploration is separate from policy evaluation/training, and policy training is reduced to cost-sensitive classification. Modularity of the methodology maps to modularity in the system design.

In what follows, we formalize the learning problem and go over all components of the loop in Figure 1: exploration, logging, policy learning, and policy deployment. We use SimpleNews, the news website described in the Introduction, as a running example. We make some simplifying assumptions for clarity and intuition, and revisit them in Section 3.7.

[2] A more flexible API is provided by the client library, see Section 5.3. The precise and current specification for the web API and the client library can be found in documentation [28].

3.1 Framing and the basic notions

Let us quickly recap the basics: MWT is applied to a system, henceforth called APP, that interacts with environment as follows: a *context* arrives and is observed by APP, then APP chooses an *action*, then an *outcome* (from choosing this action given this context) is realized and observed by APP; the outcome includes a numerical reward. Such interaction is the smallest possible unit of experimentation in MWT; henceforth we call it an *experimental unit.* Many such interactions happen, possibly overlapping in time. The context is typically represented as a feature vector. Actions may have features of their own, a.k.a. *action-specific features.*

In each experimental unit, APP chooses an action by applying a *policy*: a function that inputs a context and returns an action.[3] Informally, the goal is to optimize the policy so as to maximize the total reward over all experimental units.

In SimpleNews each experimental unit corresponds to a user interacting to the site, a context consists of known properties of the user, and actions correspond to the news topics. Accordingly, a policy is a mapping from user properties to the news topics. Rewards correspond to clicks: 1 if the user clicked on the headline, and 0 otherwise. The outcome can include other information such as the *dwell time*, the time spent reading the news article. The goal is to choose a policy that maximizes the total number of clicks over time.

Throughout, we denote contexts with x, actions with a, and policies with π. In particular, we write $a = \pi(x)$ if policy π chooses action a given context x. The set of all possible contexts is denoted with X, and the set of all possible actions is denoted with A.

Example 3.1. In SimpleNews contexts and actions (*i.e.,* news topics) could be represented as known feature vectors $x = (x_{(1)}\,,\;\dots\;, x_{(d)})$ and $a = (a_{(1)}\,,\;\dots\;, a_{(d)})$ in the same feature space $\mathbb{R}^d$. The features may correspond to possible subjects so that $x_{(j)}$ represents the user's interest in this subject, and $a_{(j)}$ represents a given news topic's relevance to this subject. A policy could pick an action a that, given context x, maximizes the weighted product $\sum_{j=1}^{d} w_{(j)}\, a_{(j)}\, x_{(j)}$, according to some weight vector $w = (w_{(1)}\,,\;\dots\;, w_{(d)}) \in \mathbb{R}^d$. The weighted product can, for example, represent an estimate for the reward of action a given context x. Each such policy is specified by vector w.

Example 3.2. Alternatively, a policy can be based on a *decision tree*, a flowchart in which each internal node corresponds to a "test" on some attribute(s) of the context (*i.e.,* is the user male of female), branches correspond to the possible outcomes of that test, and each terminal node is associated with a particular action. An action is computed starting from the "root node" of the flowchart and following the branches until a terminal node is reached.

Policy comparison and assumptions. How do we compare policies to one another? This may be problematic because one policy may be better than another today and worse tomorrow, and because one policy may perform better than another on some contexts and worse on some others. We get around these issues by assuming that APP faces a stationary environment (but see Section 3.7 for how to cope with a limited amount of non-stationarity). In precise terms, each experimental unit is characterized by three things: the distribution from which the context is sampled, the set of feasible actions, and the reward distribution for each context and each feasible action. We assume that all three things are the same across all experimental units. In the SimpleNews example, users

[3] In the absence of exploration, policies deployed in production are typically *deterministic*, *i.e.,* always returning the same action for the same context. In our terminology, both deterministic and randomized policies are allowed.

are drawn independently from a large, fixed pool of users, the set of news topics is always the same, and the click probability is determined by the user context and the news topic.

Under these assumptions, the performance of each policy π can be summarized with a single number: its expected reward per experimental unit, denoted $\mu(\pi)$. In other words, $\mu(\pi)$ is the expected reward from applying policy π to a randomly drawn context x. Denoting with $\mu(a|x)$ the expected reward of action a given context x, we can write

$$\mu(\pi) = \mathbb{E}\left[\mu(\pi(x)|x)\right], \tag{2}$$

where the expectation is over the random choice of x, and possibly also over the randomness in policy π. Thus, one policy is better than another if it has a larger expected reward.

3.2 Exploration and logging

Exploration collects data for all policies at once. It is governed by an *exploration policy*: a randomized function from contexts to actions. Thus, for each context x the exploration policy specifies a distribution over actions, denoted $p_{\texttt{exp}}(\cdot|x)$. The probability of choosing action a given context x is denoted $p_{\texttt{exp}}(a|x)$.

Conceptually, each experimental unit i consists of the following steps:

1. context $x_i \in X$ is observed.
2. action $a_i \in A$ is drawn independently at random from distribution $p_{\texttt{exp}}(\cdot|x_i)$.
3. outcome o_i is realized and observed, including reward r_i.
4. the i-th data point $(x_i, a_i, p_i, r_i, o_i, \texttt{aux}_i)$ is recorded, where $p_i = p_{\texttt{exp}}(a_i|x_i)$ is the probability assigned by the exploration policy to a_i, and $\texttt{aux}_i$ is (optional) auxiliary information.

These data points, one for each experimental unit i, comprise the exploration dataset.

The outcome o_i may consists of multiple fragments, possibly observable at different times. For example, in `SimpleNews` the outcome can consist of click and dwell time. The reward r_i may be affected by more than one of these fragments.

The auxiliary information $\texttt{aux}_i$ can help with debugging and/or learning. Debugging information can include a time stamp, application ID, and exploration policy configuration. For the sake of learning one may wish to record additional features and exploration probabilities of other actions.

Exploration policies. While there are many possible ways to define an exploration policy, one needs to select one that is most suitable to the application scenario at hand.

- `Uniform` is a simple exploration policy that picks an action uniformly at random.

One would like to simultaneously maximize the quality of the dataset and minimize the performance degradation due to exploration. These objectives, traditionally called *exploration* and *exploitation* in Machine Learning, are at odds with one another; this tension is known as the *exploration-exploitation tradeoff*. One way to control the performance degradation is to take into account the *default policy*: a safe-choice policy such as the policy currently used in production, and an *exploration parameter* which controls the amount of exploration.

- `TauFirst` is an exploration policy that explores for the first τ experimental units, by applying `Uniform`, and uses the default policy afterwards.

- EpsilonGreedy is an exploration policy that explores with probability ϵ in each experimental unit, by applying Uniform, and uses the default policy with the remaining probability.

Here τ and ϵ are exploration parameters. One can view TauFirst as a version of EpsilonGreedy that front-loads all exploration. This is more efficient (because the exploration data become available faster), but less robust to change over time. For example, if SimpleNews deploys TauFirst throughout the day, then all exploration may happen in the morning and not be indicative of the afternoon, whereas EpsilonGreedy would not have this problem.

An exploration policy that assigns a substantial positive probability to every action is appealing because the resulting dataset can be used to analyze every possible policy, and find good policies even if they are very different from the default policy. An obvious downside is that low-performing actions get explored, too.

To reduce performance degradation even further, uniform exploration may be replaced by exploration that does not stray too far from the default policy. For example, if an "ensemble" of several reasonable policies is known, the default policy may randomize among them. Then simply following such default policy already provides some exploration if the policies in the "ensemble" are sufficiently different from one another. We may further add low-probability uniform exploration, as in EpsilonGreedy, to ensure that exploration happens no matter what.

- EpsilonEnsemble is a version of EpsilonGreedy in which the default policy is based on a given "ensemble" of policies. In each experimental unit, the default policy selects a policy from the "ensemble" independently and uniformly at random, and applies it to the context.

Alternatively, the default policy may be based on a *score predictor* that assigns a numerical score $\hat{\nu}(a|x)$ to each action a for a given context x, *e.g.,* an estimated expected reward. Then the default policy simply picks an action with the highest predicted score, *e.g.,* like a linear policy from Example 3.1. However, other actions need to be included for exploration, giving preference to actions with higher predicted scores:

- SoftMax is an exploration policy parameterized by a default policy with score predictor $\hat{\nu}$. Given context x, each action a is sampled with probability

$$\texttt{SoftMax}(a|x) = \frac{e^{\tau \cdot \hat{\nu}(a|x)}}{\sum_{a \in A} e^{\tau \cdot \hat{\nu}(a|x)}},$$

  where τ is the exploration parameter. Here $\tau = 0$ corresponds to Uniform, and increasing τ biases SoftMax in favor of the higher-scoring actions.

In practice, SoftMax and EpsilonEnsemble tend to provide a better exploration-exploitation tradeoff than Uniform, TauFirst and EpsilonGreedy, as long as one has, resp., a good score predictor or an ensemble of reasonably good policies that are sufficiently different from one another. EpsilonEnsemble may be computationally expensive if the "ensemble" is too large.[4]

[4] This is because one needs to execute each of these policies in order to compute the probabilities p_i for EpsilonEnsemble, even though only one of these policies is actually chosen. Indeed, the probability of choosing a particular action in experimental unit i is equal to the fraction of policies that choose this action given context x_i.

3.3 Policy learning

Policy learning applies machine learning to the collected dataset. It happens *offline* — on a fixed dataset, without receiving any new data. Policy learning comes in two flavors: *evaluate* a given a policy, *i.e.,* estimate its expected reward as defined in Equation 2, and *train* a policy, *i.e.,* find a policy among the allowed policies that (approximately) maximizes the estimated expected reward. The same data point i can be used to evaluate many different policies π, namely all policies π that choose the same action $a_i = \pi(x_i)$ for a given context x_i. This allows exponential improvement over A/B testing, as explained in Section 3.6.

Policy evaluation. The basic method to evaluate a given policy π is *Inverse Propensity Scoring* (`IPS`). For simplicity, assume π is deterministic, *i.e.,* it always returns the same action for the same policy. Then the expected reward $\mu(\pi)$ is estimated with

$$\hat{\mu}_{\texttt{ips}}(\pi) = \frac{1}{N} \sum_{\text{data points } i \text{ such that } a_i=\pi(x_i)} \frac{r_i}{p_i}, \tag{3}$$

where N is the number of data points. This estimator has three important properties. First, it is *data-efficient.* Each interaction on which policy π matches the exploration data can be used in evaluating π, regardless of the policy collecting the data. In contrast, A/B testing uses only data collected using policy π to evaluate π. Second, the division by p_t makes it statistically *unbiased*: it converges to the true reward as $N \to \infty$. This is a standard desirable property in statistics. Third, the estimator can be recomputed incrementally when new data arrives.

The `IPS` estimator can be easily extended to a randomized policy π. Let $q(a|x)$ be the probability assigned by π to an action a given context x. Then the expected reward $\mu(\pi)$ is estimated with

$$\hat{\mu}_{\texttt{ips}}(\pi) = \frac{1}{N} \sum_{i=1}^{N} q(a_i|x_i)\, \frac{r_i}{p_i}. \tag{4}$$

Policy training finds a policy that approximately maximizes the estimated expected reward (usually given by the `IPS` estimator). A simple but computationally inefficient solution is to evaluate every allowed policy and output the one with the highest estimated reward. A better approach is to use a reduction to *cost-sensitive classification*, a well-studied problem in machine learning for which many efficient algorithms have been designed and implemented. The choice of a particular algorithm for cost-sensitive classification implicitly defines the class of allowed policies: it is simply the class of all policies that can possibly be trained using this algorithm. In particular, there are algorithms for policy classes specified by linear weights (as in Example 3.1), decision trees, and neural nets. All such policies tend to be very fast to execute. Many cost-sensitive classification algorithms internally compute a “score predictor” needed for an exploration policy like `SoftMax`.

For intuition, let us describe a basic reduction to cost-sensitive classification. An algorithm `ALG` for cost-sensitive classification inputs a sequence of N data points, where each data point i consists of context $x_i \in X$ and costs $c_i(a)$ for all actions $a \in A$. Here x_i is interpreted as an example to be labeled, and $c_i(a)$ is the cost of assigning label a to this example. One can view each policy π as a classifier that assigns label $\pi(x)$ to each example $x \in X$. Thus, policy π incurs cost $\sum_{i=1}^{N} c_i(\pi(x_i))$. `ALG` outputs a policy π whose cost is approximately minimal among all policies considered by `ALG`.

Policy training executes a single call to ALG with costs

$$c_i(a) = \begin{cases} -\frac{1}{N}\, r_i/p_i, & a = a_i \\ 0 & \text{otherwise,} \end{cases}$$

and returns the policy computed by ALG. The costs are defined so as to match IPS, in the sense that the cost of each policy π equals $-\hat{\mu}_{\texttt{ips}}(\pi)$. Thus, minimizing the cost of a policy is equivalent to maximizing the IPS estimator.

Choosing a particular policy class and a particular algorithm for cost-sensitive classification is a delicate matter which we leave beyond the scope of this document. We refer the reader to the documentation of ML software packages such as VW [68] and TLC [66], and recommend using default/recommended algorithms and settings unless they have a good reason to do otherwise.

3.4 Policy deployment: completing the loop

According to the loop in Figure 1, the trained policies should eventually be deployed in APP. Further, the loop should be iterated: from exploration to logging to policy learning to policy deployment and back to data collection. If possible, the exploration policy should be updated so that the policy trained in the previous iteration becomes the new "default policy" in the next iteration. Then the system can learn more efficiently, gradually zooming in on better and better policies and thus reducing performance degradation associated with exploration.

The minimal feasible frequency of iterations of the loop depends on several factors: how long it takes to collect a data point and prepare it for the policy training phase, how long it takes to retrain a policy given the new data, and whether deploying a new default policy requires human oversight. Also, in some applications deploying a new default model makes sense only if it improves over the old one by a statistically significant amount; so one may need to collect more data points for the difference to become statistically significant.

Note that one does not need to *trust* a policy training algorithm to produce a good policy. Instead, one can treat the trained policy as a suggestion, and use policy evaluation to compare it with other suggested policies before deploying it. A standard machine learning consideration applies: one cannot evaluate a trained policy on the data it has been trained on, and should instead set aside a separate dataset for evaluation.

3.5 Discussion: exploration policies and policy training

In principle, the MWT methodology allows an arbitrary exploration policy to be used in conjunction with any policy training algorithm,[5] so that one can choose a combination that works well for a particular application. Besides, the choice of exploration policies and policy training algorithms may be restricted by business constraints or available implementations.

The approaches known to work well in research, either in theory or in experiments, tend to combine a particular exploration policy with a specific policy training algorithm (but allow to plug in an arbitrary algorithm ALG for cost-sensitive classification). The basic approach combines EpsilonGreedy exploration policy and the reduction to ALG described in Section 3.3. It is very fast computationally, taking a single call to ALG, and achieves a reasonable learning performance (more

[5] As long as the two agree on the format of the default policy as it is updated. For example, SoftMax exploration policy requires a default policy based on a score predictor.

on this in Section 3.6). However, all actions must be assigned the same probability in exploration, even the low-performing ones, which is suboptimal.

More advanced approaches [29, 5] jointly implement exploration policy and policy training, and the loop from one to another, so as to achieve better learning performance. The policy training algorithm calls ALG on multiple carefully constructed instances of cost-sensitive classification, and uses the solutions to update the default policy in a way that ensures good exploration-exploitation tradeoff. The downside is larger (but still reasonable) computational cost due to the extra calls to ALG. Two such approaches are implemented in the Decision Service, under the names OnlineCover and Bagging; both use EpsilonEnsemble exploration policy.

3.6 Performance characterization and comparison to A/B testing

We characterize the performance of policy evaluation and policy training, explain how it depends on the pertinent parameters, and highlight the improvement over A/B testing. This subsection can be skipped if one wishes to move to system design and implementation.

Let us characterize the statistical efficiency of policy evaluation, as given by the IPS estimator in Equation 3. Specifically, let us consider the width of the confidence interval around the IPS estimator (here low width means that the estimator has high precision). In general, the width is small unless the exploration policy tends to assign very small probability to actions chosen by π. To take a concrete example, suppose there are m actions, and N data points are collected using an exploration policy such as EpsilonGreedy which places probability at least ϵ/m on each action, for some constant $\epsilon \leq 1$. Further, suppose we wish to evaluate K different policies at once with error probability $\delta > 0$: *i.e.,* with probability at least $1-\delta$ it must be the case that the expected reward of each policy lies in the respective confidence interval. Then

$$\texttt{width} < \sqrt{\frac{Cm}{\epsilon N} \log \frac{K}{\delta}} \quad \text{for any } \delta > 0, \tag{5}$$

where C is a small constant.[6] Note the square-root dependence on m, the inverse-root dependence on N and ϵ, and the logarithmic dependence on K and $1/\delta$. The dependence on ϵ summarizes the influence of the exploration policy on the confidence interval, and in particular shows the importance of exploration: with $\epsilon \to 0$, Equation 5 becomes uninformative.

To directly compare against A/B testing, note that if one A/B tests K policies and collects N data points total, then the width of the confidence interval is as large as $C'\sqrt{\frac{K}{N}\log\frac{K}{\delta}}$, for some constant C'. Therefore, when $\frac{m}{\epsilon}$ is small, Equation 5 achieves ***exponential improvement over A/B testing*** in terms of the dependence on K: we can evaluate exponentially more policies to the same precision with the same amount of data.

Policy training allows for strong provable guarantees, similar to those for policy evaluation. Continuing with the same example as above, the basic reduction described in Section 3.3 computes an allowed policy that is optimal up to an additive error of

$$\texttt{error} < \gamma + \sqrt{\frac{Cm}{\epsilon N} \log K}, \tag{6}$$

where γ is the additive error in the call to ALG, and C is a small constant. The error scales *logarithmically* in the number of policies. Whereas if one uses A/B testing to choose an approximately

[6] The proof is a standard application of *Bernstein's Inequality*, a well-known tool from statistics.

optimal policy among the K allowed policies and collects N data points total, this gives additive error as large as $C'\sqrt{\frac{K \log K}{N}}$, for some constant C'. Thus, we again achieve an ***exponential improvement over A/B testing*** when $\frac{m}{\epsilon}$ is small: we can train over exponentially more policies to the same precision with the same amount of data.

3.7 Essential extensions

We touch upon several extensions of the basic `MWT` methodology that tend to be essential in applications. In particular, we revisit some of the assumptions made earlier in the section.

Reward metrics. There may be several reasonable ways to define rewards, especially when the outcome of an experimental unit consists of multiple fragments. For example, in `SimpleNews` the reward can include a bonus that depends on the dwell time, and there may be several reasonable choices for defining such a bonus. We support any reward metric that can be computed for each experimental unit as a function of the recorded outcome o_i. Further, the reward may depend on the context x_i, *e.g.,* in `SimpleNews` some clicks may be more important than others, depending on the user's demographics.

The reward metric used by `APP` may change over time, either because of a change in priorities or when another reward metric is discovered to be a better proxy for the long-term objective. Further, application owners may wish to keep track of some auxiliary reward metrics. To switch to a new reward metric, one can just post-process the dataset and recompute the rewards r_i. Alternatively, one can replace every reference to r_i with a function call that computes r_i from o_i and x_i.

Variable action sets. In some applications the set of feasible actions may change over time, *i.e.,* from one experimental unit to another. Then the feasible actions are included in the context, along with the corresponding action-specific features (if applicable). Actions are typically identified by their feature vectors, as far as the `MWT` methodology is concerned.

Any policy based on a score predictor easily extends to this scenario, choosing an action with maximal predicted score among all actions that are feasible in the current experimental unit (rather than among all actions). Likewise, a policy based on a decision tree can be adapted, so that in a given experimental unit the non-feasible actions are removed.[7] All exploration policies in Section 3.2 extend, too: *e.g.,* `Uniform` would choose uniformly among the feasible actions.

Non-stationarity. While predictions about future performance rely on the stationarity assumption, applications exhibit only periods of near-stationarity in practice. To cope with a changing environment, we use a continuous loop in the vein of Figure 1 in which the policies are re-trained and re-deployed as quickly as possible as new data becomes available. This has three implications for policy training. First, policy training should be implemented using an online learning algorithm, *i.e.,* an algorithm that incorporates new data points efficiently without restarting the training. Second, policy training should be adapted to a changing environment. For example, while most online learning algorithms gradually become less sensitive to new data, we adjust the sensitivity so that learning stabilizes during a period of near-stationarity, but can adapt to new trends over longer periods of time. This can be achieved by occasionally resetting the step-sizes present in

[7] Assume for simplicity that each decision node has only two outgoing branches. The decision tree can be preprocessed so that each branch is annotated with the set of all actions reachable from this branch. In a given experimental unit, a branch with no feasible actions is ignored, and the "twin" branch is chosen instead.

most online learning algorithms. Third, we need enough data within a period of near-stationarity (see Equation 1 in Section 2 for a rule-of-thumb).

Non-stationarity can often be partially mitigated if the context captures some of the variable properties of the "environment". For example, users of `SimpleNews` may become more interested in sports during a major sports event such as the Olympics. Rather than just treat such events as a major source of non-stationarity, one can include their presence or absence as a feature in the context. Then the response to a particular news topic given a particular context may become much more consistent over time.

In a non-stationary environment, the goal of policy evaluation is no longer to estimate the expected reward per experimental unit (simply because this quantity changes over time). Instead, the goal is *counterfactual*: estimate the policy's performance if it was used in the experimental units from which the exploration data was collected. This is a mathematically precise goal that is achievable (say) by the `IPS` estimator regardless of how the environment changes in the meantime. When algorithms for exploration and/or policy training are evaluated on the exploration data, the goal is counterfactual in a similar sense. Such counterfactual evaluation of policies and algorithms, while not necessarily a good predictor of the future, is very useful for comparing them with the alternative approaches.

Slates of actions. In some applications, a ranked list of actions (a *slate*) is chosen in each experimental unit, *e.g.,* list of search results or news articles. Treating the entire slate as a single action is inefficient, because too many slates are possible.[8] One practical approach, adopted in the `MSN.com` deployment, is to train policies to optimize the top slot of the slate, and use the same policies to pick actions for all other slots. More precisely, the approach works as follows. Each trained policy should be based on a *score predictor*, as described in Section 3.2. Given the scores predicted for a given context, an action for the top slot is chosen according to an exploration policy such as `EpsilonGreedy` or `SoftMax`. The remaining feasible actions are ranked in the order of predicted scores, and assigned to the remaining slots in this order. Note that this approach allows for policy evaluation for the top slot of the slate, but not for the other slots. We are currently developing principled approaches to explore, train and evaluate for all slots at once [43, 62].

Evaluating trained policies. One can evaluate the entire sequence of policies trained over time, producing a single number for the entire sequence. This estimate may be more informative than evaluation of specific trained policies if the environment keeps changing as new policies are being trained. Moreover, this estimate can be computed incrementally, and does not require a separate "testing dataset". This is in contrast to evaluating a particular trained policy, which should be done on a separate dataset, and with a full pass over the said dataset. In practice, this estimate can be used as a safeguard to verify that the learning loop is behaving as expected.

Let us state more precisely what we wish to evaluate. Let π_i be the default policy for experimental unit i. That is, π_i is the default policy used by the exploration policy when making a decision on this experimental unit; it is trained using some or all of the previous data points. With the benefit of hindsight, consider an algorithm which knows all π_i's in advance, and just uses policy π_i in each experimental unit i. We can think of this algorithm as a *dynamic* trained policy. We are interested in its total expected reward on the first t experimental units, denoted $\mu^{(t)}$.

The quantity $\mu^{(t)}$ can be estimated as follows. Since the i-th data point was not used for training π_i, it can be safely used for evaluating it. We can use a one-step `IPS` estimate $\pi_i(a_i|x_i)\, r_i/p_i$, where

[8] Recall that the statistical guarantees in Equation 5 and Equation 6 depend on $\sqrt{\#\text{actions}}$.

$\pi_i(a|x)$ is the probability assigned by π_i to action a given context x. Thus, $\mu^{(t)}$ can be estimated as the sum of such one-step estimates: $\sum_{i=1}^{t} \pi_i(a_i|x_i)\, r_i/pi_i$. Call it the `ProgressiveIPS` estimate. Just like `IPS`, it is statistically unbiased and can be recomputed incrementally for the next t.

Offline experimentation. Policy evaluation, as defined above, uses exploration data to simulate an experiment on live users (namely, repeatedly executing a particular policy). Exploration data can be used to simulate live experiments in a much broader sense. Training and evaluating policies via alternative algorithms (such as those that require more time to compute, use different hyper-parameters, are based on estimators other than `IPS`, or lead to different policy classes), trying out and tuning new algorithms for exploration and policy learning, experimenting with unused logged features, and switching to different reward metrics are all possible. Such experimentation using exploration data is summarily called *offline experimentation*. Extending the `IPS` technique, research ensures that offline experimentation is counterfactually accurate [30, 31, 45, 46]. Just like policy evaluation, offline experimentation does not require additional live experiments and does not require approving and implementing the algorithms being tested in production.

4 System Design for MWT: the MWT Decision Service

We present the MWT Decision Service: a system designed to provide the complete learning loop in Figure 1 and to harness the power of `MWT` for a wide range of potential applications, while preventing many of the errors we have encountered deploying `MWT` in practice. This section outlines the design requirements and the system architecture, suppressing implementation- and application-specific details in favor of general principles. While the implementation (described in Section 5) builds on `Azure` [12], the architecture could easily be migrated to another cloud provider.

4.1 Challenges and design requirements

Building the Decision Service runs into many challenges in terms of data collection, latency, scalability, usability, robustness, debugging and machine learning. Let us discuss these challenges and the associated design requirements in more detail.

Data collection. `MWT` relies crucially on accurate logging of the $(x_i, a_i, p_i, r_i, o_i)$ tuples. The system must record (x_i, a_i, p_i) at the time of decision and eventually match it with the corresponding outcome and reward (r_i, o_i). There are many ways we have seen data collection go wrong in practice. For example, the probabilities p_i may be incorrectly recorded or accidentally included as action features. Second, features may be stored as a reference to a database which is updated over time. Then the feature values available at the time of policy evaluation might differ from the ones at the time the decision was recorded. Third, when optimizing an intermediate step in a complex system, the action chosen initially might be overridden by downstream business logic, and the recorded action might incorrectly be this final outcome rather than the initially chosen action. Finally, the rewards and other outcome fragments, which are often delayed and may arrive from entirely different paths within a given application, may be lost or incorrectly joined to the decision.

Latency. For interactive high-value applications serving latency tends to be directly linked to user experience and revenue [58]. Many applications now seek to keep their total response times within 100 ms, so the Decision Service should make decisions in 10 ms or less.

Some applications must quickly incorporate new data into the trained policy (e.g. for breaking news stories). Then the Decision Service should be able to update the policy every few minutes. The delay to update the trained policy is called *learning latency.*

Scalability. The Decision Service needs to efficiently support applications with both high and low volumes of data. The volume of in-memory data depends on the size of the interaction tuple, the interaction arrival rate, and the typical delay until the reward is observed. The online learning algorithm should be able to handle the data arrival rate. Finally, reproducibility may demand additional buffering due to reordering from scale-out components.

Usability and flexibility. The Decision Service should be *modular* so as to integrate easily with an application's existing infrastructure. That is, it should consist of components which are usable independently or in combination with well-defined interfaces that admit multiple consistent implementations. This avoids costly re-implementation of existing functionality, and allows the service to improve seamlessly as better implementations become available. Supporting multiple programming languages and avoiding mandatory dependencies on particular external libraries reduces the barrier to adoption, as applications use different programming languages and environments, and some hesitate to take additional dependencies. Finally, the service should be easy to try out, since application developers may be wary of automatic learning systems. Overall, the design should not enforce a ones-size-fits-all approach, but provide sensible defaults for every component to reduce setup complexity for common cases.

Fault-tolerance. The application must continue making decisions despite any failures in the components of the Decision Service. Conversely, the Decision Service must be able to recover from the failures, restarts or reconfigurations in the application itself. Even in the business-as-usual mode, data collection may be skewed in time, and some of the data may not reach the service. Either way, the Decision Service should not lose data that was received, and should recover the previously learned policies and other valuable state of the learning algorithm.

Debuggability. Systems which span many components and interact with users are often notoriously difficult to debug. In particular, events may be delayed, re-ordered or dropped and affect the system in complex ways, making it difficult to reproduce bugs offline. This generic difficulty is magnified when the system is continuously learning, because the system is no longer stationary, and it's difficult to disentangle issues in the learning algorithms from systems issues. Therefore, it is essential to be able to fully reproduce an online run of the service offline.

Machine learning. The service should provide adequate systems support for machine learning, which means supporting a variety of algorithms and processes used by data scientists in practice to explore, tune, and adapt policies. As much as possible, we would like to not be tied to a particular ML library or policy structure, so that applications can add the Decision Service on to their existing ML workflow. Also, in order to adapt to a changing environment and workloads the Decision Service should be able to revise the learned policies continuously and support real-time resets/reconfigurations of the learning algorithms. Furthermore, there is plenty of prior art in machine learning (*e.g.,* supervised learning) that should be leveraged when appropriate. Finally, the service should support offline experimentation to tune and try out exploration/learning algorithms in realistic conditions.

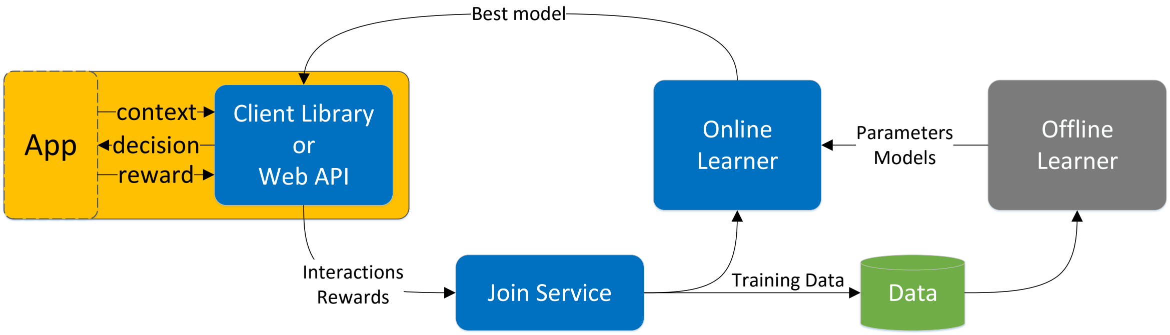

Figure 2: Decision Service architecture

4.2 Architecture and Semantics

The high-level architecture of the Decision Service depicted in Figure 2 roughly maps to the four steps of the loop in Figure 1. The Client Library interfaces with the application system (`APP`), makes decisions, performs exploration and issues logging requests. The Join Service is responsible for joining and logging exploration data. The Online/Offline Learner components perform policy learning, and trained policies are deployed back to the Client Library.

Each module has a well-defined interface; in particular, the modules agree on the format of the data being transmitted. As a consequence, each component can be used in isolation or replaced with customized implementations to suit the application's environment. For example: the Client Library supports custom logging classes, which can send exploration data to an external system for logging and learning; the Join Service can be implemented by any key-value store that supports expiration timers; the Online Learner can take data points generated by an external source, and can be implemented using any ML package that understands this data and satisfies the latency and throughput requirements of `APP`. We leverage this flexibility in the deployment with `MSN.com`.

Client Library: This module correctly implements various exploration policies discussed in Section 3.2. Concretely, it takes as input context features x and a key k from `APP`, and outputs an action a. The key k uniquely identifies the experimental unit, and is also called *UnitID*. Separately, a keyed tuple $\langle k, (x, a, p, \texttt{aux})\rangle$, called the *decision tuple*, is transmitted to the Join Service; here p is the probability of the chosen action a according to the exploration policy, and `aux` is auxiliary information to be used for policy learning and/or debugging. Later, a key k and an outcome fragment o (*e.g.,* a reward), is input from `APP`, triggering a separate transmission of tuple $\langle k, (o)\rangle$ to the Join Service. Note that multiple such tuples, called *outcome tuples*, can be transmitted for a given experimental unit.

Returning the action a is time-critical, as it delays `APP`'s response to a user, and should be optimized for latency. Whereas communication with Join Service can be delayed (as long as the delay is much smaller than the desirable learning latency), and therefore should mainly be optimized for throughput. The Client Library should allow one to specify the default policy, and to revise it as needed. A recommended mode of revising the default policy is to periodically pull it from the Online Learner at a configurable rate.

Randomization occurs by seeding a pseudorandom number generator (PRG) using the UnitID and also the application ID. The PRG is invoked exactly once per experimental unit and is never reused across experimental units. Including the application ID in the seed ensures that the randomization from multiple uses of the Decision Service in the same stack are not correlated.

Join Service: This module collects exploration data: each decision tuple is joined with the outcome tuple(s) from the same experimental unit, often arriving after a substantial delay or from an entirely different component of `APP`. The Join Service is parameterized by a *join window*: how long to wait for the decision/outcome tuples with a given UnitID after the first tuple has arrived.

Logging and joining is not subject to the stringent response latency requirements of `APP`. Therefore Join Service can be delegated to back-end infrastructure, and should mainly be optimized for throughput. The Join Service should scale to the volume and frequency of the arriving decision/outcome tuples; the said frequency may be huge if `APP` provides little or no batching of the tuples. The only requirement in terms of latency is that the additional delay on top of the join window remains small compared to the desired learning latency.

The Join Service is oblivious to the content of the tuples being joined. Its interface is very simple: the Join Service takes a stream of keyed tuples (k, o) and emits a stream of joined tuples $(o_1, o_2, ...)$ where observations are joined if they share the same key and occur within the join window from the first appearance of this key. The joined tuples are then output to the Online Learner (via a queue), and also sent to Data Storage for offline learning. While the implementation of Join Service may be complex (e.g., it could span many machines in a data center, and be heavily optimized to a specific data center), by design it should be completely transparent to the rest of the Decision Service.

Online Learner: This component performs policy training *online*, *i.e.,* quickly updates the trained policy to incorporate a stream of data points output by the Join Service. It can run continuously and adapt to changing workloads or application environments. Policies are re-trained and checkpointed at a configurable rate, and pulled by the Client Library at (another) configurable rate.

As discussed in Section 3.2, some exploration policies may require an "inside view" into the default policy, *e.g.,* `SoftMax` requires access to the score predictor. Accordingly, policies trained by the Online Learner should be encapsulated in a policy class which provides an "inside view" compatible with the needs of the Client Library.

The Online Learner can also be used to evaluate arbitrary policies in real time. First, performance of a “default” or “safe” policy can be tracked (*e.g.,* via the IPS estimator, as discussed in Section 3.3). Second, the trained policies can be evaluated via `ProgressiveIPS` or a similar estimator (see Section 3.7), and/or via policy evaluation on the randomly chosen test data. These statistics can be used to implement performance safeguards (cf. Section 4.5) and to display performance on a dashboard, *e.g.,* using tools such as `PowerBI` [56] or `D3.js` [27].

The Online Learner consists of a Reader Module to process the incoming data stream, and an ML Module to actually perform policy learning. The Reader Module translates from the output format of the Join Service to the input format of the ML module, and is therefore specific to both.

Offline Learner: The Offline Learner provides *offline experimentation* capability, as discussed in Section 3.7. Improvements generated through offline experimentation can be integrated into the online loop by simply restarting the Online Learner with the newly optimized policy or algorithm. Similarly to its online counterpart, the Offline Learner consists of a Reader Module which ingests exploration data from Data Storage, an ML module which actually performs the learning, and the interface module. The ML module can be the same as the one used by the Online Learner.

ML module(s). An ML module comprises algorithms for policy learning and a system for running these algorithms at a sufficient scale. In principle, any ML system can be used, as long as it provides machine learning from exploration data and (for the Online Learner) supports sufficiently fast incremental updates. An ML module can be based on a third-party ML software such as `VW` [68], `SuperAB` [47], and `TLC` [66], and be built on top of an existing cloud service such as `AzureML` [17] or Azure Data Factory [15]. The same ML module can potentially be used across multiple applications.

The current status of the above-mentioned projects, as related to the Decision Service, is as follows. `VW` provides algorithms for cost-sensitive classification, and policy training via a reduction to cost-sensitive classification; both are suitably fast to be used in the Online Learner. `SuperAB` provides reduces policy training to cost-sensitive classification and calls `TLC`, which in turn offers algorithms for cost-sensitive classification.[9] `VW` and `TLC` are also available via AzureML. Availability outside of Microsoft is as follows: `VW` is open-source, `AzureML` and `Azure Data Factory` are publicly available cloud services, whereas `TLC` and `SuperAB` are only available inside Microsoft (but `TLC` can be used indirectly via `AzureML`).

Reward computation. The outcome fragment(s) pertaining to a given experimental unit do not necessarily specify the numerical reward as an explicit field. For example, the absence of a click does not typically get logged explicitly as a specific no-click outcome with a specific reward. So there must be a default reward value to be inserted in post-processing in the absence of an explicit reward. Moreover, the reward may be defined as a function of multiple outcome fragments, in which case having rewards logged by the Client Library may be impractical most of the time. Thus, the data pipeline typically includes a reward computation step, implemented either as a post-processing step in the Join Service or as a pre-processing step in the Reader module of the Online Learner, whichever is more convenient. Ideally, the "reward function" should be customizable, rather than hard-coded into the implementation.

4.3 A typical front-end data flow

Let us describe the front-end data flow in more detail. We focus on a typical scenario, depicted in Figure 3, in which the outcomes (such as clicks) are generated directly by human users. A user client such as an app or a web browser communicates with front-end servers (FES) of the application system. Each FES runs an instance of the Client Library, and logs exploration data to the Join Service. A single experimental unit involves communication between the user client, one or more FES of the application system, and the back-end of the Decision Service. The typical time flow of an experimental unit i follows the following steps:

(S1) At a front-end server (FES) of the application system:

- (S1a) context x_i is received from a user.
- (S1b) unique key k_i (UnitID) is generated, and action a_i is drawn independently at random from distribution $p_{\texttt{exp}}(\cdot|x_i)$, where $p_{\texttt{exp}}(a|x_i)$ is the probability assigned by the exploration policy to action a.
- (S1c) pair (k_i, a_i) is sent to the user.
- (S1d) decision tuple $(k_i, x_i, a_i, p_i, \texttt{aux}_i)$ is sent to the Join Service, where $p_i = p_{\texttt{exp}}(a_i|x_i)$ and $\texttt{aux}_i$ is (possibly empty) auxiliary information.

(S2) At the user client:

[9] More precisely, `TLC` handles a special case of cost-sensitive classification that is sufficient for our purposes. The algorithms in `TLC` are different from those offered by `VW`.

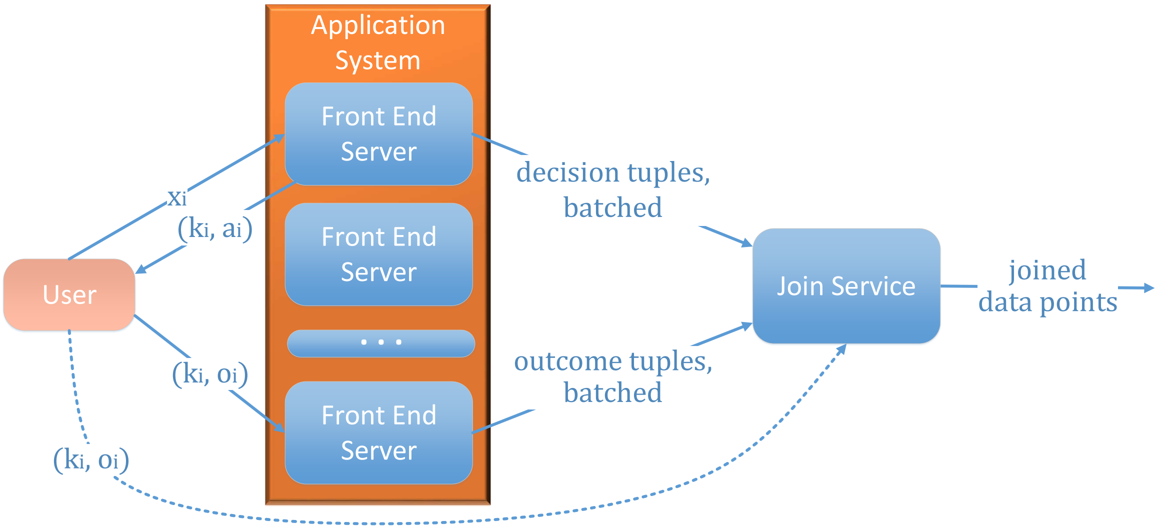

Figure 3: A typical front-end scenario

(S2a) pair (k_i, a_i) is received from the FES in Step (S1).

(S2b) for each observed outcome fragment o_i, the outcome tuple (k_i, o_i) is sent to an FES (possibly not the same FES as in Step (S1)).

(S3) Decision tuples (k_i, o_i) are transmitted to the Join Service by the receiving FES in Step (S2b)

While we previously treated context x_i as an immutable object, it may be altered as it is transmitted from one component to another. First, context x_i received from the user in step (S1a) may contain “pointers” that are subsequently expanded by the FES to the “full context” as it is used in step (S1b) and logged in step (S1c). For instance, if the context includes the user ID or a URL, the FES can look up the corresponding user profile or webpage (and features thereof), as stored in the application system. Second, the Client Library would typically summarize the “full context” as a feature vector, and only use that feature vector as far as the Decision Service is concerned. The choice of features may impact the size of a context as well as the performance of policy training. Third, *feature caching* may be used to save bandwidth when the outcome tuples are transmitted from the Client Library to the Join Service: if an action is feasible for many experimental units, its features can be replaced with references most of the time.

The action sent to the user in step (S1c) may be a “direct consequence” of action a_i chosen in step (S1b) and logged in step (S1d). For instance, in `SimpleNews` action a_i is a news topic, whereas the user receives a webpage rendered according to the chosen a_i.

The user client may be instrumented to transmit the decision tuples to the Join Service directly (with or without sending them to an FES), *e.g.,* via web API.

4.4 Full reproducibility

As pointed out in Section 4.1, capability to fully reproduce an online run of the Decision Service offline, using logged data, is essential for debuggability. Instrumenting this capability requires additional specifications for each component of the Decision Service.

Policies trained by the Online Learner should be identified by unique policy IDs and published to the Data Store. The Client Library should record the policy ID of the default policy used in a given experimental unit, and log it as auxiliary information. Pseudo-random generator (PRG) should also be published to the Data Store. Since the Client Library seeds PRG with UnitID and application ID, all pseudo-random are reproducible.

Data points may be re-ordered inside the Join Service, especially when the Join Service runs on multiple machines, and the Online Learner should records the order in which data points are processed (and log this order along with the exploration data). Since the Online Learner periodically resets the hyperparameters of the policy training algorithm so as to favor recent experimental units, these resets should also be recorded. If the Online Learner uses parallelization, one needs to record a few additional values such as the parallel learning configuration.[10] Together, this ensures each trained policy is reproducible.

4.5 Safeguards

The Decision Service enables a fully automated online machine learning loop: *automated* and *machine learning* are key selling points of the service. However, they are also liabilities, because it can be difficult to verify the behavior statically. As a result, applications engineers may not trust the behavior of such a system. The Decision Service has two key properties that make it possible to safeguard a deployed system and alleviate these concerns.

The first is compatibility with business logic and environmental constraints. The Decision Service allows arbitrary business logic to exist downstream that may alter, bound, or reverse any of its choices. Such logic can be used to implement fail-safes against violations of business policy or human safety. As long as this logic remains stationary, it will not affect `MWT`'s ability to provide unbiased counterfactual estimates of policy performance.

The second property is the ability to track its own performance in real time, as provided by the Online Learner. In particular, one can compare a new policy against a pre-defined "default" before it is deployed. This feedback can be incorporated into a control loop that changes the exploration policy in a reactive fashion, *e.g.,* tunes down exploration and/or switches to the "safe" default policy when substantial performance degradation is detected.

5 Implementation: MWT Decision Service v2

This section describes the current implementation of the Decision Service, released in June 2016. The back-end builds on `Azure` [12] and several services thereof.

5.1 Components

Client Library. Versions of the Client Library are implemented in C#, C++, and Java. The code is open-source, available at Client Library GitHub repo [53]. The C# library is about 5K lines of code: 1.5K for the various exploration policies and another 1.5K which handle batched uploads to the Join Service. The Client Library may link to `VW` invoke policies (*i.e.,* to compute the action for a given context), but it does not call any subroutines for policy training or evaluation. The

[10] However, we have not required parallelization so far in our deployments.

default policy can be invoked in a multi-threaded manner without duplicating its state, resulting in sub-millisecond parallel decisions with low memory overhead.

Web API. The web API described in Section 2 is implemented by running instances of the Client Library on a web server. The web server is built and hosted on Azure Web Apps [19]. In particular, it is auto-scalable in a configurable way, so as to meet the applications' requirements. Technical support is provided by Azure. Compared to the Client Library, Web API is easier to deploy, (more) cross-platform and has no code dependencies. On the other hand, Client Library is more customizable and provides better decision latency.

Join Service. The Join Service is implemented using Azure Stream Analytics [9] (`ASA`), which allows one to specify a delayed join with 4 lines of query language (where the delay is set to the join window). `ASA` can be scaled up/down, and handles failures/restarts, by storing incoming data in a fault-tolerant queue (an Azure Event Hub [16]), and replaying data as needed. The Join Service is also *fast*, adding a negligible delay on top of the join window. The Join Service works with both Client Library and Web API: decision tuples and outcome tuples can come from either source, even within the same application and the same experimental unit.

Reward computation is implemented as a post-processing step, which is extremely convenient in `ASA`. Reward is either identified as a particular JSON field or computed from multiple fields.[11] When the required field(s) are not present, the "default reward" is substituted, *e.g.,* one corresponding to the absence of a click.

Online Learner. The Online Learner is implemented in 1.8K lines of C# as a stand-alone Azure worker role. For the ML Module, we use `VW` [68], an open-source online learning library. The Reader Module (implemented as a C# wrapper for `VW`) translates the exploration data from the on-the-wire JSON format used by the Client Library and Client Library to a `VW` format.

`VW` solves the policy training problem by reducing it to cost-sensitive classification. By default, we train a policy with a linear representation (a vector of weights, as in Example 3.1), using algorithm "Cost-Sensitive-One-Against All" [32]. This algorithm performs incremental updates extremely fast, adding only milliseconds of latency to the learning process. Many other representations and algorithms are available, too. `VW` also supports parallel online learning using the `AllReduce` communication primitive [2], which provides scaling across cores and machines. Thus far we have not needed parallel learning for our deployments.

Offline Learner. The Offline Learner is implemented using Azure Data Factory [15] to orchestrate the data flow, and Azure Batch [13] to schedule parallel batch jobs. Further, we provide a "wrapper" with the same interface as the Online Learner; we call it the *low-frequency trainer.* It may be useful for applications in which the default policy can be updated with low frequency, but retraining requires much computation.

Data Storage is implemented via Azure Blobs [14].

5.2 Features and optimizations

Actions and features. The implementation supports variable number of actions and action-specific features. More precisely, there are two regimes: fixed number of actions without action-specific

[11] However, the supported reward functions are restricted to those expressible as a Transact-SQL queries.

features and variable number of actions with action-specific features. They are handled differently by the ML module and the Client Library. One or the other must be chosen during setup.

The Online Learner supports adding/removing features without recompile or restart. This is because feature representation is specified in the data, rather than baked into the code.

Exploration policies. All exploration policies described in Section 3.2 are supported, and available from the Client Library API as well as from the web API.[12] `SoftMax` and `EpsilonEnsemble` are implemented in `VW`, and linked from the Client Library. A custom exploration policy can be specified and used via the Client Library.

Advanced logging. In addition to the chosen action probability, the Client Library logs probabilities for all feasible actions. These probabilities can be used for debugging as well as for some of the advanced approaches for policy learning.

Push-button deployment. The Decision Service can be deployed with the push of a button after registering the application at the Decision Service Portal [54]. Currently deployment takes around 6 minutes due to the time required to deploy various Azure components. Billing is taken care of automatically via an Azure account. Registration form also asks to configure a few options such as the join window and the exploration policy (most of these options can later be reset via the Management Center). Registration form provides an authorization token which should be included in the API calls to identify the application to the Decision Service.

Management Center. The *Management Center*, built and hosted on Azure Web Apps [19], allows to (re)configure various settings in a running instance of the Decision Service without having to recompile or redeploy. The configurable settings include the choice of a built-in exploration policy and a built-in policy training algorithm (and parameters thereof). Settings for exploration apply to the Client Library and the web API in a uniform way. Also, the Online Learner can be forced to reset or create a check-point; the latter includes outputting the trained policy to be consumed by the Client Library. The Management Center additionally has the a dashboard for policy evaluation (implemented via `D3.js` [27]), which shows a graph comparing different policies. It also has links to all Azure resources used by the Decision Service, so that users can view and configure them in more detail via the Azure portal.

Robust logging and randomization. The Client Library systematically prevents bugs by consolidating all logging and randomization logic in the application, and having the corresponding code developed and verified by experts in systems, machine learning, and software development. In particular, the decision tuple is set aside for logging at the point of decision, so that the feature values used for policy evaluation are the same as those used for decision, the correct probabilities are recorded, and the action chosen by the randomization is correctly logged even if downstream logic overrides it. All randomization logic exists in one place: a two-layer design is used, whereby various exploration policies sitting in the lower layer take as input a context and output a distribution over actions, and then the top layer samples randomly from this distribution.

Batching. All components support configurable batching to improve throughput. Specifically, batching can be used to transfer decision/outcome tuples from the Client Library to the Join Service, and the joined data points from the Join Service to the Online Learner and the Data Storage.

[12] Currently `EpsilonEnsemble` is only available jointly with a specific policy training algorithm (their combination is called `Bagging`). We plan to remove this limitation in the near future.

Context processing. Context processing is optimized in three ways. First, since the same context class (*e.g.,* `MyContext` in Section 5.3) is sent to the Join Service repeatedly, we construct and reuse an abstract syntax tree to speed up the serialization of the class. Second, we use a simple caching scheme to substantially reduce data transfer: if the same action is feasible for many experimental units, its features are sent explicitly only periodically, and are replaced with references otherwise. The Reader Module in the Online Learner buffers data to deal with references that arrive before the actual fragment they refer to. The latter may happen as the data points may be re-ordered in the Join Service, particularly if the Join Service runs on multiple machines. Third, a similar feature caching scheme is implemented in the decision-making code inside the Client Library, so as to substantially reduce the decision latency.

5.3 API with Client Library

Client Library provides a flexible API, which we explain below through an example (full specification can be found in documentation [28]). Recall the web API interaction from Section 2. Using the Client Library, this interaction can be customized as follows:

```
var cfg = new DecisionServiceConfiguration(".. settings URL ..");
using (var cl = DecisionService.Create<MyContext>(cfg)
    .ExploitUntilModelReady(new MyPolicy())
{
    var key =  new guid ;
    // context arrives, with Age=25
    var x = new MyContext { Age = 25, ... }
    var action = cl.ChooseAction(key , x);
        // wait for reward  to arrive
    cl.ReportReward(reward , key );
}
```

First, a configuration object cfg is created and passed an authorization token, and a Decision Service service object `cl` is created with this configuration. The code uses a custom class for contexts (`MyContext`) and a default class for policies (`MyPolicy`). The latter satisfies a simple interface, mapping contexts of type `MyContext` to an action. `ExploitUntilModelReady(policy)` specifies the default policy to use until the first trained policy is available. The exploration policy is typically set through Management Center. Finally, the `ChooseAction` and `ReportReward` calls correspond, resp., the second and third HTTP message in the Web API. Alternatively, `ChooseAction` can mimic the web API and input the default action and a JSON string with context.

The Client Library provides a flexible interface for specifying context classes that serialize to JSON. In particular, annotations can specify which members of a class should be treated as features. For example:

```
public class MyContext {
    // Feature: Age:25
    public int Age { get; set; }
    // Feature: l:New_York
    [JsonProperty("l")]
    public string Location { get; set; }
    // Logged but not used due to _
    [JsonProperty("_SessionId")]
    public string SessionId { get; set; }
    // Not logged, not used as feature
```

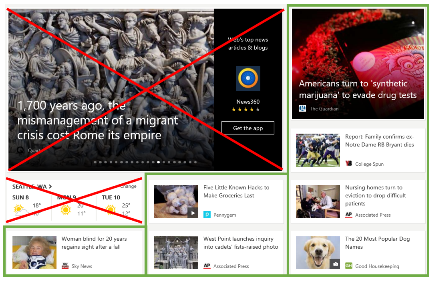
1,700 years ago, the mismanagement of a migrant crisis cost Rome its empire
Quartz
Web's top news articles & blogs
News360
Get the app
Americans turn to 'synthetic marijuana' to evade drug tests
The Guardian
Report: Family confirms ex-Notre Dame RB Bryant dies
College Spun
SEATTLE, WA
Change
SUN 8
MON 9
TUE 10
18°
10°
20°
11°
25°
12°
Five Little Known Hacks to Make Groceries Last
Pennygem
Nursing homes turn to eviction to drop difficult patients
Associated Press
Woman blind for 20 years regains sight after a fall
Sky News
West Point launches inquiry into cadets' fists-raised photo
Associated Press
The 20 Most Popular Dog Names
Good Housekeeping

Figure 4: `MSN.com` homepage showing areas optimized by the Decision Service. The Slate (boxed in green) now uses the Decision Service in production. Tests on other areas show strong positive results as well.

```
    [JsonIgnore]
    public bool SomeField { get; set; }
}
```

This context includes a regular feature (`Age`), a renamed feature (`Location`), a field that is logged but not used for policy training (`SessionId`), and a field that is completely ignored (`SomeField`). Logging unused features can be useful for future evaluation of policies using them.

The Client Library permits actions represented as feature vectors. The features are specified in a separate class using similar annotations to the above with an array embedded in the context class allowing the set of actions to depend on the context and change over time.

6 Success story: deployment with MSN

We have deployed the Decision Service to personalize the news stories displayed on the `MSN.com` homepage shown in Figure 4. This deployment is the production default since January 2016, handling thousands of requests per second. We describe the deployment and lessons learned.

Conceptually, we face a problem similar to `SimpleNews`. A user requests the homepage and `MSN.com` must decide which news articles to show. The homepage is divided into fixed slots, so that one news article must be shown in each slot, and the slots are ranked in the order of prominence. If a user is logged in, there is context: demographics (*e.g.,* age, location) and the kinds of news stories they have clicked on in the past. Otherwise only location is available. The available actions are the news articles selected and ranked by the editors; typically, tens of articles, updated at least daily. Each article has features that describe its topic. The reward is the total number of clicks. Compared to `SimpleNews`, we have two important differences: the actions correspond to news articles rather than news topics, and the decision in each experimental unit consists of choosing multiple actions.

The Decision Service is deployed at a particular segment of the homepage, consisting of multiple slots. (Multiple deployments correspond to different, non-overlapping segments.) The goal is to optimize the rate at which the clicks are received in this segment, a.k.a. the *click-through rate* (CTR). While the basic `MWT` methodology described in Section 3 focuses on choosing one action in each experimental unit, here we need to choose several actions at once. To this end, we use a simple solution described in Section 3.7, whereby the Decision Service trains policies to optimize the CTR at the most prominent slot of the segment, and uses the same policies to pick articles for

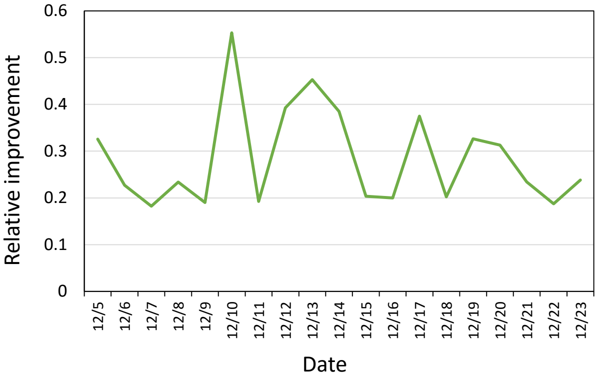

Figure 5: Daily CTR lift for Slate (vs the editorial policy). Live experiment in December 2015.

all other slots. The default exploration policy `EpsilonGreedy` is used with $\epsilon = 33\%$.[13] The join window is set to 10 minutes, and a new model is deployed to the Client Library every 5 minutes.

Live experiments and impact. `MSN.com` did several experiments before moving to production use of the Decision Service. In each experiment, the Decision Service was compared to the editorial ordering using standard A/B testing procedures. Editorial ordering was the production default which beat several previous attempts to use machine learning.

Experiments on the Slate proportion of the page (shown in Figure 4) were very successful, revealing a $> 25\%$ CTR improvement over a two-week period. Figure 5 shows the per-day CTR lift of this run. There is day-to-day variation due to the availability of articles, but the Decision Service consistently delivered at least 18% lift and a maximum of 55% lift in the period. Based on offline experimentation, we believe roughly half of the CTR lift comes from the policy training algorithm optimizing with the users' demographics and the other half from the users' reading history.

The above gains were achieved while maintaining or improving longer-term engagement metrics such as sessions per unique user and average session length, showing that the easily-optimized CTR metric is aligned with longer-term goals in this case. We expect a more sophisticated exploration policy to further improve CTR.

The success of the experiments led `MSN.com` to make it the default in production for all logged-in users. The `MSN.com` team has since begun experimenting with using the Decision Service on more users and areas of the site[14], revealing further high-impact applications. These later experiments have been run by the `MSN.com` team on their own, showing the system is usable by non-experts.

In general, having the Decision Service learn to fine-tune the ranking of articles frees the editors to focus their attention on their essential task of curating the articles.

Deployment characteristics. `MSN.com` deployed a custom version of the Decision Service using some of the default components and re-implementing others as required. They used the C# Client Library in front-end servers for low-latency.

`MSN.com` implemented their own version of the Join Service based on Redis Cluster [57]. It is configured to accommodate a request rate in the low thousands per second, interaction size in the thousands of bytes, and a 10 minute join window. Scale is primarily driven by the request rate,

[13] This choice of ϵ was driven by the stationarity interval and the observed CTRs.

[14] One nice property of the Decision Service is that extending to different areas of the site can be implemented easily as different applications.

since memory pressure is low for short join window. Clicks are reported directly from user web browsers via web requests.

Further, MSN.com implemented a custom version of the Online Learner based on VW [68], in part because the available infrastructure did not naturally support sufficiently fast online updates. Each instance of the Online Learner works on a single core. MSN.com used the provided Client Library, but customized the classes responsible for logging the exploration data and polling for new trained policies. The offline experimentation has been done as a one-off, but a generic Offline Learner is in the works (built on top of Cosmos, a cloud system internal to Microsoft, and based on VW).

The deployment supports multiple applications, corresponding to different page segments. The Join Service is shared across the applications (distinguished by application ID), whereas a separate instance of the Online Learner is used for each application to isolate the trained models.

The deployment used the default algorithms for exploration and policy training. Offline experimentation, with no additional live experiments, was used to tune these algorithms. In terms of implementation, the MSN.com developers directly used code written by our developers, either by copying and modifying as needed or as black-box libraries delivered via Nuget packages.

The Decision Service was subject to various performance tests before an experiment was allowed to go live. In particular, MSN.com carefully monitors the CPU-per-request metric, as their front-end servers are CPU-bounded. The Decision Service increased CPU/request by 4.9%, which was deemed acceptable. The majority of this overhead is due to invoking the trained policy.

Feature caching was used, and has been tremendously useful, resulting in 20X reduction in the volume of data transferred to the Decision Service.

Lessons learned. We learned several lessons from the MSN.com deployment.

Reward encodings matter. In our initial experiment, we found poor performance, which was traced down to a combination of low CTR and an click/no-click reward encoding as $\{-1, 0\}$ instead of $\{0, 1\}$. This seemingly minor difference had a huge impact on the variance of estimators. Since then, we have modified the policy training algorithms in VW to be less dependent on a particular reward encoding.

Large Risk for Large Reward. It is a natural urge to want to shield the most critical and high-value portions of an application from exploration. This results, for example, in editors locking important slots away from the recommendations of the Decision Service. While this limits the negative impact, it also places a low ceiling on possible improvements as we can only optimize less impactful portions of the page. In our deployment, we found each time the editors unlocked a more important slot, the CTR and engagement overall improved significantly.

Reproducibility is key. The ability to deterministically reproduce offline any failure mode or performance degradation observed in the logs was instrumental in fixing numerous issues we encountered over the deployment.

Deployment flexibility is essential. The modular design of the Decision Service made it easy integrate with the existing infrastructure: take advantage of that which is suitable, avoid dependency on that which is not suitable, and be in the position to ponder which is which. In particular, the Join Service was built on top of Redis Cluster [57], and that happened after MSN.com concluded that an alternative based on Azure Tables [18] was not suitable. The Client Library easily incorporated custom logging for Redis Cluster [57], and would have just as easily supported the same for Azure Tables or many other systems. The Online Learner was custom-made so as to avoid a dependency on some of the existing infrastructure that was not optimized for online learning; again, the provided Client Library easily supported polling for the trained policies.

Communication and trust help. To take an example, MSN.com was using multiple policy training modules for fault-tolerance, which affected reproducibility. Communicating this earlier would have sped up debugging. Another example: a clear communication about the additional business logic preceding and succeeding the Decision Service in the pipeline was instrumental in explaining and debugging some of the behaviours. Finally, having MSN.com developers trust and directly use code written by our developers helped speed up the deployment.

7 Experimental Evaluation

The previous section reported results from a live deployment with MSN.com on real user traffic. In this section, we use the data collected from this deployment to evaluate various aspects of the Decision Service design. We answer the following questions:

- How quickly are decisions made and policies learned?
- Can high data rates be handled?
- Are policy evaluation estimates reliable?
- How quickly does a policy achieve good performance when starting from scratch?
- How important is it to continuously train the policy?
- What effect does bad logging have on the policy?

Experimental setup and methodology. The Decision Service was deployed in our own Azure subscription. The sample code distributed with the Client Library was deployed on several VM instances with 8 cores, 14GB memory, and 1Gbps NIC each ("A4" instances in Azure terms). The Join Service uses Azure Stream Analytics [9], which provides scalability through a configurable number of *streaming units*. The Event Hub queues feeding the Join Service and storing its output can also be scaled as needed. The Online Learner is as implemented as a stand-alone worker role on a single VM instance with 16 cores and 56GB memory ("D5" instances in Azure terms).

All our experiments use real exploration data from the MSN.com deployment for a three-day period in April 2016. There are, approximately, 20 actions per experimental unit (on average), 500 features per action, and 500+ "global" (non-action-specific) features. By joining this data with browser click logs, we can determine both the time of decision and the time of click, for a realistic replay of the data. To evaluate the performance of policies, we use the policy evaluation capability of the Online Learner (via the IPS estimator in Equation 3).

Latency of learning and decisions. We are interested in two quantities: the *decision latency* and the *learning latency*. Decision latency is the time to make a decision in the Client Library (*i.e.,* the ChooseAction call). Learning latency is the time from when ReportReward has been called to when it affects a deployed policy in the Client Library.

We measured the decision latency by training a policy on one hour of MSN.com data, deploying this policy in the Client Library, and then repeatedly calling ChooseAction. The average decision latency measured is 0.2ms.

To measure the learning latency, we first removed any configurable sources of delay. We disabled batching and caching in the Client Library (because it tends to re-order the decision/outcome tuples as they arrive to the JS, which in turn leads to increased delays). Also, we configured the Client Library to poll for new models every 100ms; we set the Join Service join window to 1 second; and

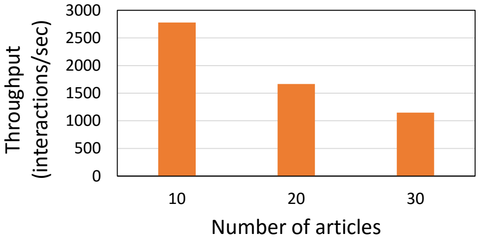

Figure 6: Online Learner throughput for different article counts per experimental unit.

configured the Online Learner to publish an updated policy after processing each data point. We then replayed one experimental unit of the `MSN.com` data and waited for a policy to appear (and repeated it many times). The average learning latency observed is 7.3 seconds. We attribute most of this latency (on top of the join window and the polling delay) to moving data to/from various queues inside Azure.

Scalability to high data rates. The default Join Service implementation and the one used by `MSN.com` (based on Azure Stream Analytics [9] and Redis Cluster [57], resp.) are inherently scalable services, so we focus on the Online Learner.

The data rates seen in production so far have been adequately handled by learning on a single core. To saturate the Online Learner, we preloaded `MSN.com` data into the Join Service output queue and processed it at full speed. We used the Client Library compression scheme to mimic what `MSN.com` does. The throughput achieved by the Online Learner was stable at 2000 experimental units/sec, implying that applications with 100 million experimental units/day are viable. Buffering for reordered events whose compressed features were not yet available was minimal, remaining at 0 most of the time with occasional spikes up to 2250 buffered experimental units.

In the case of `MSN.com`, the throughput of policy training is directly affected by the number of articles (*i.e.,* actions) in each experimental unit. We measured this effect for different representative article counts; the corresponding throughputs are shown in Figure 6. The average number of articles in a typical day is close to 20.

Reliable policy evaluation. While policy estimation using `IPS` as per Equation 3 has theoretical guarantees, it is prudent to verify this empirically. We took 3 simple policies, which always choose, resp., the first, second and third article from the editorial ranking for the top slot. The first policy corresponds to the editorial baseline, while others are reasonable alternatives. For each policy, we computed an `IPS` estimate of its value from a day's worth of data for two different days in April. Additionally, we ran a control experiment on both days: an A/B test in which all three policies were deployed live, which allowed us to we estimated the performance of each policy with a very high precision. Across all policies and all days, the relative difference between the `IPS` and control values was no more than 2.5%, and all `IPS` estimates were within a 95% confidence interval around the control values.

Speed of learning. We next use similar `IPS` estimates to capture the speed of learning. The policies currently in production in `MSN.com` have been continuously trained for months (with daily resets of the learning rate). How quickly does a policy achieve good performance when starting from scratch? To investigate this, we played a full day of `MSN.com` data and compared the trained policy to the editorial policy. We leveraged the Online Learner's support for evaluating arbitrary policies

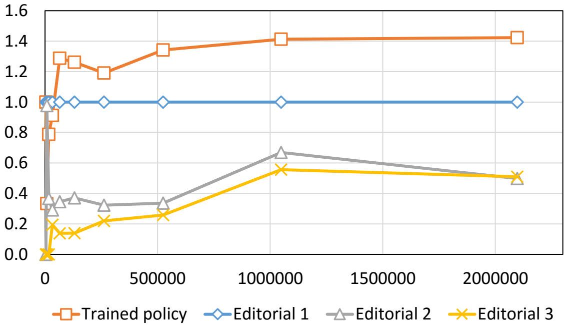

Figure 7: Real-time comparison of trained policy against editorial policies.

in real-time. Figure 7 shows the results, where each datapoint was captured within an average of 10 seconds of the corresponding decision/outcome tuple. Both the trained and editorial policies exhibit high variance and become statistically significant with more data. The trained policy starts outperforming editorial after just 65K experimental units—for a request rate of 1000/sec, this is about 1 minute—and eventually achieves a 42% improvement by end of day.

Continuous policy training. To demonstrate the importance of continuous policy training, we used the policy from the previous experiment (trained on day 1) and tested it on day 2 and day 3. We compared this to a policy trained on the corresponding day. We found that day 1's policy achieves 73% of the CTR of day 2's policy when evaluated on day 2 data, and 46% of day 3's policy when evaluated on day 3 data. This suggests that the environment and articles have changed, and day 1's policy has become stale. Continuous policy training solves this problem.

Data collection failures. Our motivation for building the Decision Service came from witnessing past failures to deploy `MWT`, many of which were due to incorrect logging of exploration data. To illustrate the effects of incorrect logging, we simulated two common failure modes using the exploration data from `MSN.com`. We amplified the "severity" of the failures to make their effects easier to observe.

For each of the two experiments, we restricted our attention to the most prominent slot on the page. We took a day's data and randomly allocated 80% of it for training and 20% for testing while preserving time order, thus approximating the standard train/test split methodology of supervised learning. Running the Decision Service on the training set simulates having the Decision Service train a policy in production, while the test set allows us to evaluate the trained policy (*i.e.,* simulate the performance of this policy when deployed online throughout). This simulation is imperfect because training occurs on fewer events than in the actual deployment (*i.e.,* 80% of events vs. 100% of events), and because what is being evaluated on the test set is the final trained policy rather than the entire Decision Service.

We also evaluated the entire sequence of policies trained by the Decision Service over time, using the `ProgressiveIPS` estimate from Section 3.7 on the training set. Recall that the `ProgressiveIPS` estimate can be used by the Decision Service as a safeguard to verify that the learning loop is working well. Interestingly, when evaluating the final trained policy π on the test set, only a 90% of CTR is observed compared to the `ProgressiveIPS` estimate. We attribute this discrepancy to intra-day non-stationarity: since the environment may have been changing throughout the day, the policies trained earlier in the day may be better than π when run on the earlier experimental units.

In the first experiment, we simulated incorrect logging of actions caused by downstream logic, in this case: editors overriding some of the actions chosen by the exploration policy. Specifically, after exploration policy chooses an action a, with probability 10% this action is overridden with some other action, and this other action is logged instead of a. We trained a policy π on the training set and evaluated it on the test set. We found that the CTR of π, evaluated on the test set, was 3 times lower than the `ProgressiveIPS` CTR. Such huge discrepancy indicates that policy training on the corrupted data may be almost useless, and moreover this may be difficult to ascertain (via a reasonable safeguard provided by `ProgressiveIPS`) with the corrupted training data alone.

The second experiment focuses on another common failure mode whereby the identity or probability of the random action chosen by exploration is used as a feature for policy training later on. We simulated this failure mode by adding a new feature to each action. In the training set, the logged value of this feature was 1 if the action was sampled by the Decision Service, and 0 otherwise. The test data was not augmented with this feature since this random choice is not available to the policy at decision time. We trained a policy π on the training set and evaluated it on the test set. We found that the CTR of π, as evaluated on the test set, was 8.7 times lower than that estimated by `ProgressiveIPS`. We make the same conclusion: corrupted data hurts policy training in a way that may be difficult to catch via a reasonable safeguard.

References

[1] Alekh Agarwal, Sarah Bird, Markus Cozowicz, John Langford, Stephen Lee, Jiaji Li, Luong Hoang, Dan Melamed, Gal Oshri, Oswaldo Ribas, Siddhartha Sen, and Alex Slivkins. Multiworld testing decision service. Technical report `http://arxiv.org/abs/1606.03966`, under submission, 2016.

[2] Alekh Agarwal, Oliveier Chapelle, Miroslav Dudíik, and John Langford. A reliable effective terascale linear learning system. *J. of Machine Learning Research (JMLR)*, 15:1111−−1133, 2014.

[3] Alekh Agarwal, Miroslav Dudíik, Satyen Kale, and John Langford. Contextual bandit learning under the realizability assumption. In *15th Intl. Conf. on Artificial Intelligence and Statistics (AISTATS)*, 2012.

[4] Alekh Agarwal, Miroslav Dudík, Satyen Kale, John Langford, and Robert E. Schapire. Contextual bandit learning with predictable rewards. In *15th Intl. Conf. on Artificial Intelligence and Statistics (AISTATS)*, pages 19–26, 2012.

[5] Alekh Agarwal, Daniel Hsu, Satyen Kale, John Langford, Lihong Li, and Robert Schapire. Taming the monster: A fast and simple algorithm for contextual bandits. In *31st Intl. Conf. on Machine Learning (ICML)*, 2014.

[6] Deepak Agarwal, Bee-Chung Chen, Qi He, Zhenhao Hua, Guy Lebanon, Yiming Ma, Pannagadatta Shivaswamy, Hsiao-Ping Tseng, Jaewon Yang, and Liang Zhang. Personalizing LinkedIn Feed. In *21th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 1651–1660, 2015.

[7] Shipra Agrawal and Navin Goyal. Thompson sampling for contextual bandits with linear payoffs. In *30th Intl. Conf. on Machine Learning (ICML)*, pages 127–135, 2013.

[8] Amazon Machine Learning - Predictive Analytics with AWS. `https://aws.amazon.com/machine-learning/`.

[9] Azure Stream Analytics. `https://azure.microsoft.com/en-us/services/stream-analytics/`.

[10] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *J. of Machine Learning Research (JMLR)*, 3:397–422, 2002. Preliminary version in *41st IEEE FOCS*, 2000.

[11] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, 2002. Preliminary version in *36th IEEE FOCS*, 1995.

[12] Microsoft Azure: Cloud Computing Platform & Services. `https://azure.microsoft.com/en-us/`.

[13] Azure Batch: Cloud-scale job scheduling and compute management. `https://azure.microsoft.com/en-us/services/batch/`.

[14] Azure Blob Storage: REST-based object storage for unstructured data in the cloud. `https://azure.microsoft.com/en-us/services/storage/blobs/`.

[15] Azure Data Factory: compose and orchestrate data services at scale. `https://azure.microsoft.com/en-us/services/data-factory/`.

[16] Azure Event Hubs: cloud-scale telemetry ingestion from websites, apps, and devices. `https://azure.microsoft.com/en-us/services/event-hubs/`.

[17] Azure Machine Learning. `https://azure.microsoft.com/en-us/services/machine-learning`.

[18] Azure Table Storage: A NoSQL key-value store for rapid development using massive semi-structured datasets. `https://azure.microsoft.com/en-us/services/storage/tables/`.

[19] Azure Web Apps: Create and deploy mission-critical web apps that scale with your business. `https://azure.microsoft.com/en-us/services/app-service/web/`.

[20] Alina Beygelzimer, Hal Daumé III, John Langford, and Paul Mineiro. Learning reductions that really work. *Proceedings of the IEEE*, 104(1):136–147, 2016.

[21] Alina Beygelzimer, John Langford, Zhang Tong, and Daniel J Hsu. Agnostic active learning without constraints. In *Advances in Neural Information Processing Systems (NIPS)*, pages 199–207, 2010.

[22] Leon Bottou, Jonas Peters, Joaquin Quinonero-Candela, Denis X. Charles, D. Max Chickering, Elon Portugaly, Dipankar Ray, Patrice Simard, and Ed Snelson. Counterfactual reasoning and learning systems: The example of computational advertising. *J. of Machine Learning Research (JMLR)*, 14(1):3207–3260, 2013.

[23] Sébastien Bubeck and Nicolo Cesa-Bianchi. Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. *Foundations and Trends in Machine Learning*, 5(1), 2012.

[24] Wei Chu, Lihong Li, Lev Reyzin, and Robert E. Schapire. Contextual Bandits with Linear Payoff Functions. In *14th Intl. Conf. on Artificial Intelligence and Statistics (AISTATS)*, 2011.

[25] Computational Network Toolkit. `http://www.cntk.ai/`.

[26] Daniel Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li, Zhao Zhang, Michael J. Franklin, Ali Ghodsi, and Michael I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research*, 2015.

[27] D3.js: Data Driven Documents. `https://d3js.org/`.

[28] Decision Service Documentation. `https://github.com/Microsoft/mwt-ds/wiki/Decision-Service-Documentation`, 2016.

[29] Miroslav Dudíik, Daniel Hsu, Satyen Kale, Nikos Karampatziakis, John Langford, Lev Reyzin, and Tong Zhang. Efficient optimal leanring for contextual bandits. In *27th Conf. on Uncertainty in Artificial Intelligence (UAI)*, 2011.

[30] Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. Sample-efficient nonstationary policy evaluation for contextual bandits. In *28th Conf. on Uncertainty in Artificial Intelligence (UAI)*, pages 247–254, 2012.

[31] Miroslav Dudík, John Langford, and Lihong Li. Doubly robust policy evaluation and learning. In *28th Intl. Conf. on Machine Learning (ICML)*, pages 1097–1104, 2011.

[32] Ariel Faigon. Cost Sensitive One Against All (csoaa) multi class example. `https://github.com/JohnLangford/vowpal_wabbit/wiki/Cost-Sensitive-One-Against-All-(csoaa)-multi-class-example`.

[33] Alan S. Gerber and Donald P. Green. *Field Experiments: Design, Analysis, and Interpretation.* W.W. Norton&Co, Inc., 2012.

[34] John Gittins, Kevin Glazebrook, and Richard Weber. *Multi-Armed Bandit Allocation Indices.* John Wiley & Sons, 2011.

[35] Google Analytics. `http://www.google.com/analytics`. See `http://services.google.com/websiteoptimizer` for documentation on bandits.

[36] Google Cloud Machine Learning. `https://cloud.google.com/ml/`.

[37] GraphLab Create. `http://graphlab.org` and `https://dato.com/products/create`.

[38] Steve Hanneke. Theory of disagreement-based active learning. *Foundations and Trends® in Machine Learning*, 7(2-3):131–309, 2014.

[39] Katja Hofmann, Lihong Li, and Filip Radlinski. Online evaluation for information retrieval. *Foundations and Trends® in Information Retrieval*, 10(1):1–117, 2016.

[40] Kevin G Jamieson, Lalit Jain, Chris Fernandez, Nicholas J Glattard, and Rob Nowak. Next: A system for real-world development, evaluation, and application of active learning. In *Advances in Neural Information Processing Systems*, pages 2638–2646, 2015.

[41] Ron Kohavi and Roger Longbotham. Online controlled experiments and a/b tests. In Claude Sammut and Geoff Webb, editor, *Encyclopedia of Machine Learning and Data Mining.* Springer, 2015. To appear.

[42] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. Controlled experiments on the web: survey and practical guide. *Data Min. Knowl. Discov.*, 18(1):140–181, 2009.

[43] Akshay Krishnamurthy, Alekh Agarwal, and Miroslav Dudík. Efficient contextual semi-bandit learning. *arxiv.org*, abs/1502.05890, 2015.

[44] John Langford and Tong Zhang. The Epoch-Greedy Algorithm for Contextual Multi-armed Bandits. In *21st Advances in Neural Information Processing Systems (NIPS)*, 2007.

[45] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *19th Intl. World Wide Web Conf. (WWW)*, 2010.

[46] Lihong Li, Wei Chu, John Langford, and Xuanhui Wang. Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In *4th ACM Intl. Conf. on Web Search and Data Mining (WSDM)*, 2011.

[47] Lihong Li and Andrzej Pastusiak. SuperAB: a machine-learning toolkit for learning-to-interact problems. `http://superAB/`. Only available inside Microsoft.

[48] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 583–598, 2014.

[49] Tyler Lu, Dávid Pál, and Martin Pál. Showing Relevant Ads via Lipschitz Context Multi-Armed Bandits. In *14th Intl. Conf. on Artificial Intelligence and Statistics (AISTATS)*, 2010.

[50] Language Understanding Intelligent Service (LUIS). `https://www.luis.ai`.

[51] Mixpanel: Mobile Analytics. `https://mixpanel.com/`.

[52] SPARK MLlib. `http://spark.apache.org/mllib`.

[53] MWT Decision Service Client Library: GitHub repository. `https://github.com/multiworldtesting/decision`.

[54] Multiworld Testing Decision Service. `http://aka.ms/mwt`.

[55] Optimizely: A/B Testing & Personalization Platform. `https://www.optimizely.com/`.

[56] Microsoft Power BI: Interactive Data Visualization BI Tools. `https://powerbi.microsoft.com/`.

[57] Redis Cluster Specification. `http://redis.io/topics/cluster-spec`.

[58] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity*, 2009.

[59] Patrice Simard, David Chickering, Aparna Lakshmiratan, Denis Charles, Léon Bottou, Carlos Garcia Jurado Suarez, David Grangier, Saleema Amershi, Johan Verwey, and Jina Suh. Ice: enabling non-experts to build models interactively for large-scale lopsided problems. *arXiv preprint arXiv:1409.4814*, 2014.

[60] Aleksandrs Slivkins. Contextual bandits with similarity information. *J. of Machine Learning Research (JMLR)*, 15(1):2533–2568, 2014. Preliminary version in *COLT 2011*.

[61] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[62] Adith Swaminathan, Akshay Krishnamurthy, Alekh Agarwal, Miroslav Dudík, John Langford, Damien Jose, and Imed Zitouni. Off-policy evaluation for slate recommendation. arXiv:1605.04812, 2016.

[63] Csaba Szepesvári. *Algorithms for Reinforcement Learning.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

[64] TensorFlow – an Open Source Software Library for Machine Learning. `https://www.tensorflow.org/`.

[65] William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285294, 1933.

[66] TLC Machine Learning Toolkit. `http://tlc/`. Only available inside Microsoft.

[67] Torch: A scientific computing framework for LuaJIT. `http://torch.ch/`.

[68] Vowpal Wabbit (Fast Learning). `http://hunch.net/~vw/`.

[69] Minjie Wang, Tianjun Xiao, Jianpeng Li, Jiaxing Zhang, Chuntao Hong, and Zheng Zhang. Minerva: A scalable and highly efficient training platform for deep learning. In *NIPS Workshop, Distributed Machine Learning and Matrix Computations*, 2014.

[70] Yelp MOE (Metrics Optimization Engine). `http://yelp.github.io/MOE`.

[71] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Conf. on Hot Topics in Cloud Computing (HotCloud)*, 2010.

A Related work: machine learning, exploration and systems

Here we discuss other machine learning approaches related to `MWT` and experimentation, and machine learning systems related to the Decision Service.

Machine learning with exploration. There are hundreds of papers related to exploration and machine learning which broadly fall into 3 categories. The simplest of these is *active learning* [21, 38, 40, 59] where the algorithm helps select examples to label in partnership with a user in order to improve model performance. A maximally general setting is *reinforcement learning* [61, 63] where an algorithm repeatedly chooses among actions and receives rewards and other feedback depending on the chosen actions. A more minimal setting is multi-armed bandits (MAB) where only a single action affects observed reward [23, 34]. The `MWT` methodology builds on contextual bandits with policy sets [4, 5, 11, 29, 44] which reduces policy optimization on exploration data to a supervised learning problem, allowing incorporation of tools developed for supervised learning (see [20] for background on learning reductions). The `MWT` methodology also incorporates offline policy evaluation in contextual bandits [30, 31, 45, 46] (see survey [39] for additional background). Alternative approaches for contextual bandits typically impose substantial modeling assumptions such as linearity [10, 24, 45], Lipschitz rewards [49, 60], or availability of (correct) Bayesian priors [7], which often limits applicability.

A/B testing. A/B testing refers to randomized experiments with subjects randomly partitioned amongst treatments. It is routinely used in medicine and social science, and has become standard in many Internet services [42, 41], as well supported by statistical theory [33]. A more advanced version, "multi-variate testing", runs many A/B tests in parallel. Several commercialized systems provide A/B testing in web services (Google Analytics [35], Optimizely [55], MixPanel [51], etc.). The Decision Service instead builds on MWT, a paradigm exponentially more efficient in data usage than A/B testing.

Bandit learning systems. Several platforms support bandit learning for web services. Google Analytics [35] supports Thompson Sampling [65], a well-known algorithm for the basic Multi-Arm Bandit problem. Yelp MOE [70] is an open-source software package which implements optimization over a large parameter space via sequential A/B tests. Bayesian optimization and Gaussian Processes are used to compute parameters for the "next" A/B test. SigOpt.com is a commercial platform which builds on Yelp MOE. However, these systems do not support *contextual* bandit learning, and they do not instrument automatic deployment of learned policies (and hence do not "close the loop" in Figure 1).

Contextual bandits deployments. There have been several applications of contextual bandit learning in web services that we are aware of (e.g., news recommendation [6, 45, 46] and Advertising [22]); however, they all have been completely custom rather than a general-purpose system like the Decision Service.

Systems for supervised machine learning. There are many systems designed for supervised machine learning such as CNTK [25], GraphLab [37], Parameter Server [48], MLlib [52], TensorFlow [64], Torch [67], Minerva [69] and Vowpal Wabbit [68] to name a few. These principally support Machine Learning model development. A few more, such as Google Cloud ML [36], Amazon ML [8], and AzureML [17] are designed to support development and *deployment*. However, these systems do not support data gathering or exploration.

Velox [26] is an open-source system that supports model serving and batch training. Velox can adapt models online to users by adjusting preference weights in the user profile. It collects data that can be used to retrain models via Spark [71]. Velox does not perform exploration.

We know of two other systems designed to fully support data collection with exploration, model development, and deployment: LUIS [50] (based on ICE [59]), and Next [40]. These systems support *active learning*, and hence make exploration decisions for labeling in the back-end (unlike the Decision Service which makes decisions to guide customer-facing application behavior), and do not provide MWT capability.