Pyramid: Enhancing Selectivity in Big Data Protection with Count Featurization

Mathias Lecuyer[*1], Riley Spahn[*1], Roxana Geambasu[1], Tzu-Kuo Huang[†2], and Siddhartha Sen[3]

[1]Columbia University, [2]Uber Advanced Technologies Group, and [3]Microsoft Research

Abstract—Protecting vast quantities of data poses a daunting challenge for the growing number of organizations that collect, stockpile, and monetize it. The ability to distinguish data that is actually needed from data collected "just in case" would help these organizations to limit the latter's exposure to attack. A natural approach might be to monitor data use and retain only the working-set of in-use data in accessible storage; unused data can be evicted to a highly protected store. However, many of today's big data applications rely on machine learning (ML) workloads that are periodically retrained by accessing, and thus exposing to attack, the entire data store. Training set minimization methods, such as *count featurization*, are often used to limit the data needed to train ML workloads to improve performance or scalability.

We present *Pyramid*, a limited-exposure data management system that builds upon count featurization to enhance data protection. As such, Pyramid uniquely introduces both the idea and proof-of-concept for leveraging training set minimization methods to instill rigor and selectivity into big data management. We integrated Pyramid into Spark Velox, a framework for ML-based targeting and personalization. We evaluate it on three applications and show that Pyramid approaches state-of-the-art models while training on less than *1% of the raw data*.

I. Introduction

Driven by cheap storage and the immense perceived potential of "big data," both public and private sectors are accumulating vast quantities of personal data: clicks, locations, visited websites, social interactions, and more. Data offers unique opportunities to improve personal and business effectiveness. It can boost applications' utility by personalizing their features; increase business revenues via targeted product placement; improve social processes such as healthcare, disaster response and crime prevention. Its commercialization potential, whether real or perceived, drives unprecedented efforts to grab and store raw data resources that can later be mined for profit.

Unfortunately, this "collect-everything" mentality poses serious risks for organizations by exposing extensive data stores to external and internal attacks. The hacking and exploiting of sensitive corporate and governmental information have become commonplace [1], [2]. Privacy-transgressing employees have been discovered snooping into data stores to spy on friends, family, and job candidates [3], [4]. Although organizations strive to restrict access to particularly sensitive data (such as passwords, SSNs, emails, banking data), properly managing access controls for diverse and potentially sensitive information remains an unanswered problem.

Compounding this challenge is a significant new thrust in the public and private spheres to integrate data collected from multiple sources into a single, giant repository (or "data lake") and make that available to any applications that might benefit from it [5]–[7]. This practice magnifies the data exposure problem, transforming big data into what some have called a "toxic asset" [8].

Our goal in this paper is to explore a more rigorous and selective approach to big data protection. We hypothesize that not all data that is collected and archived is, or may ever be, needed or used. The ability to distinguish data needed now or in the future from data collected "just in case" could enable organizations to restrict the latter's exposure to attacks. For example, one could ship unused data to a tightly controlled store, whose read accesses are carefully mediated and audited. Turning this hypothesis into a reality requires finding ways to: (1) minimize data kept in the company's widely-accessible data lakes, and (2) avoid the need to access the controlled store to meet current and evolving workload needs.

A natural approach might be to monitor data use and retain only the working set of in-use data in accessible storage; data unused for some time is evicted to the protected store [9]. However, many of today's big data applications involve machine learning (ML) workloads that are periodically retrained to incorporate new data, resulting in frequent accesses to all data. How can we determine and minimize the training set—the "working set" for emerging ML workloads—to adopt a more rigorous and selective approach to big data protection?

We observe that for ML workloads, significant research is devoted to limiting the amount of data required for training. The reasons are many but typically do not involve data protection. Rather, they include increasing performance, dealing with sparsity, and limiting labeling effort. Techniques such as dimensionality reduction [10], feature hashing [11], vector quantization [12], and count featurization [13] are routinely applied in practice to reduce data dimensionality so models can be trained on manageable training sets. Semi-supervised [14] and active learning [15] reduce the amount of labeled data needed for training when labeling requires manual effort.

Can such mechanisms also be used to limit exposure of the data being collected? How can an organization that already uses these methods develop a more robust data protection architecture around them? What kinds of protection guarantees can this architecture provide?

As a first step to answering these questions, we present *Pyramid*, a limited-exposure big-data management sys-

---

tem built around a specific training set minimization method called *count featurization* [13], [16]–[18]. Also called historical statistics, count featurization is a widely used technique for reducing training times by feeding ML algorithms with a limited subset of the collected data combined (or *featurized*) with historical aggregates from much larger amounts of data. The method is valuable when features with strong predictive power are highly dimensional, requiring large quantities of data (and large amounts of time and resources) to be properly modeled. Applications that use count featurization include targeted advertising, recommender systems, and content personalization systems. Such applications rely on user information to predict clicks, but since there can be hundreds of millions of users, training can be very expensive without some way to aggregate users, like count featurization. The advertising systems at Microsoft, Facebook, and Yahoo are all built upon this mechanism [19], and Microsoft Azure offers it as a service [20].

Pyramid builds on count featurization to construct a selective data protection architecture that minimizes exposure of individual observations (e.g., individual clicks). To highlight, Pyramid: keeps a small, rolling window of accessible raw data (the *hot window*); summarizes the history with privacy-preserving aggregates (called *counts*); trains application models with hot raw data featurized with counts; and rolls over the counts to forget all traces of observations past a specified retention period. Counts are infused with differentially private noise [21] to protect individual observations that are no longer in the hot window but still fall within the retention period. Counts can support modifications and additions of many (but not all) types of models; historical raw data, which may be needed for workloads not supported by count featurization, is kept in an encrypted store whose decryption requires special access.

While count featurization is not new, our paper is the first to retrofit it for data protection. Doing so raises significant challenges. We first need to define meaningful requirements and protection guarantees that can be achieved with this mechanism, such as the amount of exposed information or the granularity of protection. We then need to achieve these protection guarantees without affecting model accuracy and scalability, despite using much less raw data. Finally, to make the historical raw data store easier to protect, we need to access it as little as possible. This means supporting workload evolution, such as parameter tuning or trying new algorithms, without the need to go back to historical raw data store.

We overcome these challenges with three main techniques: (1) *weighted noise infusion*, which automatically shares the privacy budget to give noise-sensitive features less noise; (2) an *unbiased private count-median sketch*, a data structure akin to a count-min sketch that

resolves the large negative bias arising from applying differentially private noise to a count-min sketch; and (3) *automatic count selection*, which detects potentially useful groups of features to count together, to avoid accesses to the historical data. Together, these techniques reduce the impact of differentially private noise and count featurization.

We built Pyramid and integrated it into Spark Velox, a targeting and personalization framework, to add rigor and selectivity to its data management. We evaluated three applications: a targeted advertising system using the Criteo dataset, a movie recommender using the MovieLens dataset, and MSN's production news personalization system. Results show that: (1) Pyramid approaches state-of-the-art models while training on less than *1% of the raw data*. (2) Protecting historical counts with differential privacy has only *2% impact on accuracy*. (3) Pyramid adds just *5% performance overhead*.

Overall, we make the following contributions:

1) Formulating the *selective data protection problem* for emerging ML workloads as a training set minimization problem, for which many mechanisms already exist.
2) The design of Pyramid, the first selective data management system that minimizes data exposure in anticipation of attack. Built upon count featurization, Pyramid is particularly suited for targeting and personalization workloads.
3) A set of new techniques to balance solid protection guarantees with model accuracy and scalability, such as our unbiased private count-median sketches.
4) Pyramid's code, both integrated into Spark Velox and as a stand-alone library ready to integrate in other targeting/personalization frameworks. `https://columbia.github.io/selective-data-systems/`

II. Motivation and Goals

This paper argues for needs-based selectivity in big data protection: protecting data differently depending on whether or not it is actually needed to handle a company's day-to-day workloads. Intuitively, data that is needed day-to-day is less amenable to certain kinds of protection (e.g., auditing or case-by-case access control) than data needed only for exceptional situations. A key question is *whether a company's day-to-day needs can be captured with a limited and well-defined data subset*. While we do not claim to answer this question in full, we present with Pyramid the first evidence that selectivity can be achieved in one important big-data workload domain: *ML-based targeting and personalization*. The following scenario motivates selectivity and shows how and in what contexts Pyramid helps improve protection.

II.A. Example Use Case

MediaCo, a media conglomerate, collects *observations* of user behavior from its hundreds of affiliate news and entertainment sites. Observations include the articles users read and share, the ads they click, and how they respond to A/B testing. MediaCo uses this data to optimize various processes, including recommending articles to users, showing the most relevant articles first, and targeting ads. Initially, MediaCo collected observations from affiliate sites in separate, isolated repositories; different engineering teams used different repos to optimize these processes for each affiliate site. Recently, MediaCo has started to track users across sites using cookies and to integrate all data into a central data lake. Excited about the potential of the much richer information in the data lake, MediaCo plans to provide indiscriminate access to all engineers. However, aware of recent external hacking and insider attacks affecting other companies, it worries about the risks it assumes with such wide access.

MediaCo decides to use Pyramid to limit the exposure of historical observations in anticipation of such attacks. For MediaCo's *main workloads*, which consist of targeting and personalization, the company already uses count featurization to address sparsity challenges; hence, Pyramid is directly applicable for those workloads. They configure it by keeping Pyramid's hot window of raw observations, along with its noise-infused historical statistics, in the widely accessible data lake so all engineers can train their models, tune them, and explore new algorithms every day. Pyramid absorbs many workload needs—current and evolving—as long as the algorithms draw on the same user data to predict the same outcome (e.g., whether a user will click on an ad). MediaCo also configures a one-year retention period for all observations; after this period, Pyramid removes observations from the statistics and launches retraining of all application models to purge the old activity. Finally, MediaCo stores all raw observations in an encrypted store whose read accesses are disabled by default. Access to this store is granted temporarily and on a case-by-case basis to engineers who demonstrate the need for statistics beyond those that Pyramid maintains.

In addition to targeting/personalization workloads, MediaCo has *other, potentially non-ML workloads*, such as business analytics, trend studies, and forensics; for these, count featurization may not apply. Hence, MediaCo gives direct access to the raw-data store to engineers managing these workloads and isolates their computational resources from the targeting/personalization teams.

With this configuration, MediaCo minimizes access to its collected data on a needs basis. Assuming no entity with full access to the historical raw data is malicious, Pyramid guarantees the following (detailed in §II-B). (1) Any observations preceding the hot window when
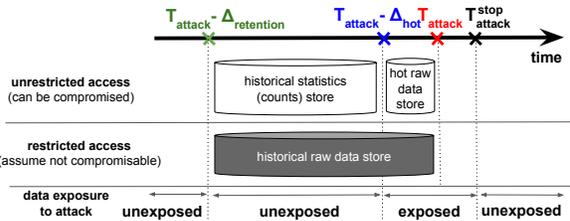

Fig. 1: **Threat model.** T_{attack}: time the attack starts; T_{attack}^{stop}: time the attack is eradicated; Δ_{hot}: hot window length; $\Delta_{retention}$: company's data retention period.

an attack begins will be hidden from the attacker. (2) Hiding is done at an individual observation level during the retention period and in bulk past the retention period. (3) Only in exceptional circumstances do engineers get access to the historical raw data. With these guarantees, MediaCo negotiates lower data loss insurance premiums and gains PR benefits for its efforts to protect user data.

II.B. Threat Model

Fig. 1 illustrates Pyramid's threat model and guarantees. Pyramid gives guarantees similar to those of forward secrecy: a one time compromise will not allow an adversary to access all past data. Attacks are assumed to have a well-defined start time, T_{attack}, when the adversary gains access to the machines charged with running Pyramid, and a well-defined end time, T_{attack}^{stop}, when administrators discover and stop the intrusion. Adversaries are assumed to not have had access to the system before T_{attack}, nor to have performed any action in anticipation of their attack (e.g., monitoring external predictions, the hot window, or the models' state), nor to have continued access after T_{attack}^{stop}. The attacker's goal is to exfiltrate individual observations of user activities (e.g., to know if a user clicked on a specific article/ad). Historical raw data is assumed to be protected through independent means and not compromised in this attack. Pyramid's goal is to limit the hot data in active use, which is widely accessible to the attacker.

Examples of adversaries that fit our threat model can be found among both the internal and external adversaries of a company. An external adversary may be a hacker who breaks into the company's computing infrastructure at time T_{attack} and starts looking for data that may prove of value (e.g., information about celebrities' specific activities, what they liked or disliked, where they were in the past, etc.). An internal adversary may be a privacy-transgressing employee who *spontaneously* decides at T_{attack} to look into some past action of a family member or friend (e.g., to check if the person has visited or liked a particular page).

After compromising Pyramid's internal state, the attacker will gain access to data in three different representations: the hot data store containing plaintext observations, the historical counts, and the trained models

themselves. The plaintext observations in the hot data store are not protected in any way. The historical statistics store contains differentially private count tables of the recent past. The attacker will learn some information from the count tables but individual records will be protected with a differentially private guarantee. Pyramid forces models to be retrained when observations are removed from the hot raw data store, so the attacker will not be able to learn anything from the models beyond what they have already learned above.

Pyramid provides three protection levels:

P1 *No protection for present or future observations.* Observations in the hot data store when the attack begins, plus observations added to the hot data store while the attack is ongoing, receive no protection; i.e., observations received between $(T_{attack} - \Delta_{hot})$ and T_{attack}^{stop} receive no protection.

P2 *Protection for individual observations for the length of the retention period.* Statistics about observations are retained in differentially private count tables for a predefined retention period $\Delta_{retention}$. The attacker may learn broad statistics about observations in the interval $[T_{attack} - \Delta_{retention}, T_{attack} - \Delta_{hot}]$ but will not be able to confidently determine if a specific observation is present in the table.

P3 *Protection in bulk past the retention period.* Observations past their retention period (i.e., older than $T_{attack} - \Delta_{retention}$) have been phased out of the historical statistics store and are protected separately by the historical raw data store.

Finally, we assume that no states created based on the hot raw data persist once the hot window is rolled over. While we explicitly launch retraining of models registered with Pyramid, we operate under the assumption that (1) the models' states are securely erased [22] and (2) no other state was created out of band based on the raw hot data (such as copies made by programmers).

II.C. Design Requirements

Given the threat model, our design requirements are:

R1 *Limit widely accessible data.* The hot data window is exposed to attackers; hence, Pyramid must limit its size subject to application-level requirements, such as the accuracy of models trained with it.

R2 *Avoid accesses to historical raw data even for evolving workloads.* Pyramid must absorb as many current and evolving workload needs as possible to limit access to the historical raw data.

R3 *Support retention policies.* Pyramid must enforce a company's retention policies. Although Pyramid provides a differential privacy guarantee, no protection is stronger than securely deleting data.

R4 *Limit impact on accuracy, performance, scalability.* We intend to preserve the functional properties of applications and models running on Pyramid.

III. The Pyramid Architecture

Pyramid, the first selective data management architecture, builds upon the ML technique of count-based featurization and augments it with new mechanisms to meet the preceding design requirements.

III.A. Background on Count-Based Featurization

Training predictive models can be challenging on data that contains categorical variables (features) with large numbers of possible values (e.g., an ID or an interest vector). Existing ML techniques that handle large feature spaces often make strong assumptions about the data, e.g., assuming a linear relationship between the features and the label (e.g., Lasso [23]). If the data does not meet these assumptions, results can be very poor.

Count-based featurization [13] is a popular approach to handling categorical variables of high cardinality. Rather than directly using the value of a categorical variable, this technique featurizes the data with the number of times a particular feature value (e.g., a user ID) was observed with each label and the conditional probability of the label given the feature value. This substantially reduces dimensionality. Suppose the raw data contains d categorical features with an average cardinality of K and a label of cardinality L, where $K \gg L$; e.g., in click prediction K can be millions (number of users), while L is 2 (click, non-click). Standard encoding of categorical variables [24] results in a feature space of dimension $O(dK)$, whereas with count featurization it is $O(dL)$. Count featurization can also be applied to continuous variables or continuous labels by first discretizing them; this increases dimensionality but only by a small factor.

The dramatic dimensionality reduction yields important benefits. It is known that fewer dimensions permit more efficient learning, both statistically and computationally, potentially at the cost of reducing predictive accuracy. However, count featurization makes it feasible to apply advanced, nonlinear models, such as neural networks, boosted trees, and random forests. This combination of succinct data representation and powerful learning models enables substantial reduction of the training data with little loss in predictive performance. Quantified in §V, this is the insight behind our use of count-based featurization to limit data exposure.

III.B. Architectural Components

Fig. 2 shows Pyramid's architecture. Pyramid manages collected data (observations) on behalf of application models hosted by a model management system. In our case, we use Velox [25], built on Spark. Velox facilitates ML-based targeting and personalization services by implementing three functions: (1) fast, but incomplete, incorporation of new observations into models that programmers register with Velox; (2) low-latency prediction serving from these models; and (3) periodic retraining of
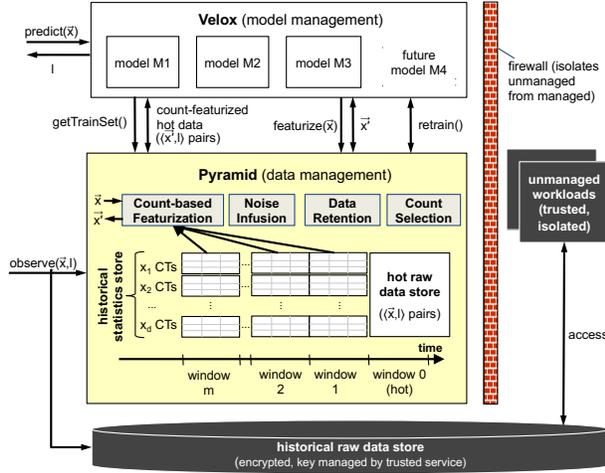
Fig. 2: **Pyramid's architecture.** Notation: $\vec{x}$: feature vector; l: label; $\vec{x'}$: count-featurized feature vector; CT: count table.

Fig. 3: **Count featurization example.**

the models to correct inconsistencies created by the incomplete incorporation of new observations. Velox saves observations in a separate data management component, Spark's Tachyon. Pyramid replaces this component to ensure rigorous and selective protection of observations.

Pyramid itself consists of four architectural components, shown across the top of the highlighted box in Fig. 2. The first is *count featurization*, which leverages the known ML mechanism to count featurize observations before feeding them to models for training and prediction. The second, third, and fourth are *noise infusion*, *data retention*, and *count selection*, which augment count featurization with differential privacy and a set of new mechanisms to meet Pyramid's design requirements. We discuss each component in turn.

III.B.1. Count Featurization

Pyramid hijacks the stream of observations collected by Velox (the `observe` method) and count-featurizes them. An observation is a pair $\langle \vec{x}, l \rangle$ with a feature vector $\vec{x} = \langle x_1, x_2, ..., x_d \rangle$ and a label l. Application models predict the label (or a probability for each possible label) for a given feature vector by training on count-featurized observations. When an observation arrives, Pyramid incorporates it into two data structures: (1) the *hot raw data store*, which retains observations from the recent past, and (2) the *historical statistics store*, which consists of multiple *count tables* that maintain the number of occurrences of each feature with each label. We maintain count tables for all features in $\vec{x}$ and for some feature combinations. A separate set of count tables is maintained for each time window.

Featurization transforms a feature vector $\vec{x}$ into a count-featurized feature vector $\vec{x'}$, by replacing each feature x_i with the conditional probabilities of each label value given x_i's value. The conditional probabilities are computed directly from the count tables as discussed
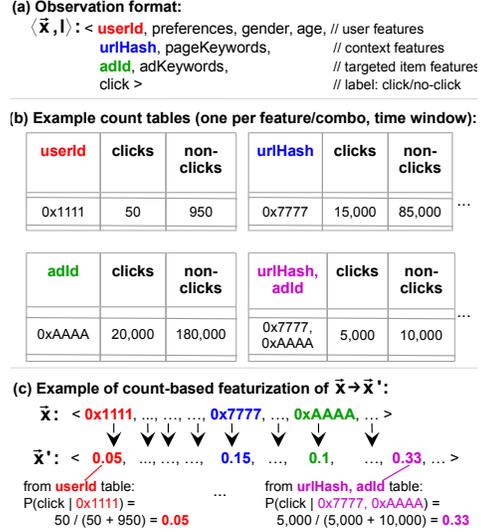
below. To train its models, an application requests a training set from Pyramid (`getTrainSet`). Pyramid featurizes the hot raw data with historical counts and returns it to the application. To predict the label for a feature vector $\vec{x}$, the application requests its featurization from Pyramid (`featurize`); Pyramid returns $\vec{x'}$.

Example. Fig. 3 shows (a) a sample observation format, (b) some count tables used by Pyramid to count-featurize it, and (c) a sample count-featurized observation.

• *Observation format.* In targeting and personalization, an observation's feature vector $\vec{x}$ typically consists of *user features* (e.g., id, gender, age, and previously compiled preferences) and *contextual information* for the observation (e.g., the URL of the article or the ad shown to the user, plus any features of these). The label l might indicate whether the user clicked on the article/ad.

• *Count tables.* Once an observation stream of the preceding type is registered with Pyramid, the *userId* table maintains for each user the number of clicks the user has made on any ad shown and the number of non-clicks; it therefore encodes each user's propensity to click on ads. The *urlHash* table maintains for each URL the number of clicks that each user made on any ad shown on that page; it therefore encodes the page's inherent "ad-clickability." Pyramid maintains count tables for every feature in $\vec{x}$ and for some feature combinations with predictive potential, such as the $\langle urlHash, adId \rangle$ table, which encodes the joint probability of a particular ad being clicked when it is shown on a particular page.

• *Count featurization.* To count-featurize a feature vector $\vec{x} = \langle x_1, x_2, \ldots, x_d \rangle$, Pyramid first replaces each of its features with the conditional probabilities computed from the count tables, e.g., $\vec{x'} = \langle P(click|x_1), P(click|x_2), \ldots, P(click|x_d) \rangle$, where $P(click|x_i) = \frac{clicks}{clicks+non\text{-}clicks}$ from the row matching the

value of x_i in the table corresponding to x_i. Pyramid also appends to $\vec{x'}$ the conditional probabilities for any feature combinations it maintains. Fig. 3(c) shows an example of feature vector $\vec{x}$ and its count-featurized version $\vec{x'}$. This is a simplified version of the count featurization function. We can also include the raw counts in $\vec{x'}$, and support non-binary categorical labels by including conditional probabilities for each label. To avoid featurizing with an effectively random probability when a given feature value has very few counts, we estimate the variance of our probability estimate and, if it is too high, featurize with a default probability $P(click)$.

• *Training and prediction.* Suppose a boosted-tree model is trained on a count-featurized dataset ($\langle \vec{x'}, l \rangle$ pairs). It might find that for users with a click propensity over 0.04, the chances of a click are high for ads whose clickability exceeds 0.05 placed on websites with ad-clickability over 0.1. In this case, the model would predict a "click" label for the feature vector in Fig. 3(c).

Process. Pyramid count-featurizes all features x_i for each observation type. For categorical features, we featurize them as described above. For low-cardinality features, we can additionally include the raw feature values in $\vec{x'}$ alongside the conditional probabilities. Continuous features are first mapped to a discrete space, binning them by percentiles, and then count-featurized as categorical. We do the same with continuous labels.

Pyramid maintains hot windows and count tables as follows. There is one hot window for each observation stream. There is one count table per feature or feature group; it has a column for each label and a row for each value the feature can take. To support granular retention times, each count table is composed of multiple windowed count tables holding data for observations collected during disjoint windows of time. The complete count table is the sum of the associated windowed count tables. When a new observation arrives, it is added to the hot window and made immediately available to the models for (re)training. The hot window is a sliding window that may be sized differently from the count table window. It is also added to the current windowed count table; this count table is withheld when computing the complete count table until it is finished populating. At this point, Pyramid begins using it as part of the featurization process, phases out the oldest count table if it is past its retention period, and begins populating a new count table that has been initialized with differentially private noise. Once count tables are incorporated into the featurization process, they are never updated again.

Count-min sketches (CMSes). A key challenge with count featurization is its storage requirement. For a categorical variable of cardinality K and a label of cardinality L, the count table is of size $O(LK)$. A common solution, used in Azure [20], is to store each

table in a Count-Min Sketch (CMS) [26], a data structure that approximates counts in sub-linear space. A CMS consists of a 2D array with an independent hash function for each row. When a new feature arrives, the CMS uses the hash function for each row to assign the feature to a column and increment the value in that cell.

We query the CMS for a feature count by hashing the feature into a column of each row and taking the minimum value. Despite overcounting from collisions, CMS provides sufficiently accurate count estimates to train ML models. With a CMS, we can maintain more and/or larger count tables with bounded storage overheads. This gives developers flexibility in the types of modeling they can do atop in-use data without tapping into the historical data store. The CMS poses challenges to our noise infusion process, as described next.

III.B.2. Noise Infusion

Pyramid's key contribution is to retrofit count featurization, a technique developed for performance and scalability, to protect past observations against exposure to attack. Pyramid infuses noise into the count tables to protect these observations. While we leverage differential privacy methods [21], correctly applying these methods in our context poses scaling challenges. For example, each observation contributes to multiple count tables, increasing the noise required to guarantee differential privacy, and a naïve application degrades accuracy when there are many count tables. We present two techniques to address this challenge. First, we use a *weighted noise infusion* technique to mitigate the impact of noise, allowing us to navigate the privacy/utility trade-off. Second, for high noise levels, we replace the CMS by a count-median sketch [27], a data structure with weaker accuracy guarantees than CMS but that provides an unbiased frequency estimate, making it more robust to negative noise values. To our knowledge, we are the first to observe that the count-median sketch structure is better suited to differential privacy. After a brief overview of differential privacy, we describe these techniques.

Differential privacy properties. Pyramid's noise infusion component uses four differential privacy properties:

1. Privacy guarantees: Let D_1 be the database of past observations, D_2 be a database that differs from D_1 by exactly one observation (i.e., D_2 adds or removes 1 observation), and S the range of all possible count tables that can result from a randomized query $Q()$ that builds a count table from a window of observations. The count table query $Q()$ is ϵ-*differentially private* if $P[Q(D_1) \in S] \leq e^{\epsilon} \times P[Q(D_2) \in S]$. In other words, adding or removing an observation in D_1 does not significantly change the probability distribution of possible count tables; therefore, the count table does not leak significant information about any specific observation [21]. ϵ is called the query's *privacy budget*.

2. Laplace distribution: Let a query's *sensitivity* be the magnitude of the change in the query result triggered by adding or removing a single observation. If the query has sensitivity Δ, then adding noise drawn from a Laplace distribution with scale parameter $\frac{\Delta}{\epsilon}$ guarantees that the result is ϵ-differentially private [21]. Increasing $\frac{\Delta}{\epsilon}$ increases the standard deviation of the distribution (stdev of a Laplace distribution with parameter b is $b\sqrt{2}$).

3. Composability: Differentially private queries are composable: the sum of n ϵ_n-differentially private queries is $(\Sigma\epsilon_n)$-differentially private [28]. This lets us maintain multiple count tables, possibly with different budgets, and combine them without breaking guarantees. (Advanced composition theorems allow sublinear loss in the privacy budget by relaxing the guarantees to (ϵ, δ)-differential privacy [29], but we do not explore that here.)

4. Post-processing resilience: Any computation on a differentially private data release remains differentially private [29]. This is a crucial point for Pyramid's protection guarantees: it ensures that guarantee **P2**, the protection of individual past observations during their lifetime, holds for each model's internal state and outputs. As long as models comply with retrain calls and erase all internal state when they do, their output is differentially private with regard to observations outside the hot window.

Basic noise infusion process. We apply these known properties when creating count tables for the hot window. Upon creating a count table, we initialize each cell of the CMS storing that table with a random draw from a Laplace distribution. This noise is added only once: the count tables are updated as observations arrive and are sealed when the hot window rolls over. To determine the correct parameter for the Laplace distribution, b, we must account for three factors: (1) the internal structure of the CMS, (2) the number of observations we want to hide simultaneously, and (3) the number of count tables (features or feature combinations) we are maintaining.

First, an exact count table has sensitivity 1 since adding or removing an observation can only change one count by 1. For a CMS, each observation is counted once per hash function; hence, the sensitivity is h, the number of hash functions. Second, if we aim to hide any group of k observations with a privacy budget of ϵ, then we make a count table ϵ-differentially private by adding noise from a Laplace distribution of parameter $b = \frac{hk}{\epsilon}$ in every cell of the CMS. Third, we must maintain multiple count tables for the different features and feature groups. Since each observation affects every count table, we need to split the privacy budget ϵ among them, e.g., splitting it evenly by adding noise with $b = \frac{nhk}{\epsilon}$ to each table.

The third consideration poses a significant challenge for Pyramid: the amount of noise we apply grows linearly with the number of count tables we keep. Since the amount of noise directly affects application accuracy, this yields a protection/accuracy tradeoff, which we address with weighted noise infusion.

Weighted noise infusion process. We note that count tables are not all equally susceptible to noise. For example in our movie recommender, the $user$ table most likely contains low values, since each user rates only a few movies (29 for the median user). Moreover, we do not expect this count to change significantly when adding more data, since single users will not rate significantly more movies. Each $genre$ table however contains higher values (1M or more), since each genre characterizes multiple movies, each rated by many users. Sharing noise equally between tables would pollute all counts by a standard deviation of 145 ($\epsilon = 1$, $h = 5$, and $k = 1$), a reasonable amount for $genre$s, but devastating for the $user$ feature, which essentially becomes random.

Pyramid's weighted noise infusion distributes the privacy budget unevenly across count tables, adding less noise to low-count features. This way, we retain more utility from those tables, and the composability property of differential privacy preserves our protection guarantees. Each table's share of noise is determined automatically, based on the count values observed in the hot window. Specifically, the user specifies a quantile, and the privacy budget is shared between each feature proportionally to this quantile of its counts. For instance we use the first percentile, so that 99% of the counts for a feature will be less affected by the noise. Sharing the privacy budget proportionally to the counts is a heuristic that makes the noise's standard-deviation proportional to the typical counts of each feature. This scheme is also independent of the learning algorithm.

Finally, the weight selection process should be made differentially private so the weights computed on a previous hot window do not reveal anything about that window's data at a later time. While our implementation currently does not do this, a design might use a small portion of one window's privacy budget and leverage smooth sensitivity [30] to compute differentially private count percentiles that can be used as feature weights. One could compute each weight as a separate differentially private query, or use the sample-aggregate framework and the center of attention aggregation [30] to compute all the weights in one query.

Section V shows that weighted noise infusion is vital for providing protection while preserving accuracy at scale: without it, the cost of hiding single observations is a 15% accuracy loss; with it, the loss is less than 5%. We leave the evaluation of incorporating differential privacy into the weight selection method for future work.

Unbiased private count-median sketch. Another factor that degrades performance when adding differentially private noise is the interaction between the noise and the CMS. In the CMS, the final estimate for a count is

$\min(h_i(key))$ for each row i. The minimum makes sense here since collisions can only increase the counts. The Laplace distribution however is symmetric around zero, so we may add negative noise to the counts. Taking the minimum of multiple draws—each cell is initiated with a random draw from the distribution—thus selects the most extreme negative values, creating a downward bias that can be very large for a small ϵ.

We observe that because the mean of the Laplace distribution is 0, an unbiased estimator would not suffer from this drawback. For tables with large noise, we thus use a count-median sketch [27], which differs in two ways: 1) each row i has another hash function s_i that maps the key to a random sign $s_i(key) \in \{+1, -1\}$, with each cell updated with $s_i(key)h_i(key)$; 2) the estimator is the median of all counts multiplied by their sign, instead of the minimum. The signed update means that collisions have an expected impact of zero, since they have an equal chance of being negative or positive, making the cell an unbiased estimate of the true count. The median is a robust estimate that preserves the unbiased property.

Using this count-median sketch reduces the impact of noise, since values from the Laplace distribution are exponentially concentrated around the mean of zero. §V shows that for small ϵ, or a large number of features, it is worth trading the CMS's better guarantees for reduced noise impact with the count-median sketch.

III.B.3. Data Retention

While differential privacy provides a reasonable level of protection for past observations, complete removal of information remains the cleanest, strongest form of protection (design **R3** in §II-C). Pyramid supports data expiration with *windowed count tables*. When an observation arrives, Pyramid updates the count tables for the current count window only. To featurize $\vec{x}$, Pyramid sums the relevant counts across windows. Periodically, it drops the oldest window and invokes retraining of all models in Velox (`retrain` method). Our use of count-based featurization supports such behaviors because retraining is cheap (§V-E), so we can afford to do it frequently.

III.B.4. Count Selection

Pyramid seeks to support workload evolution (model changes/additions, such as future model M4 in Fig. 2) using only the widely accessible stores without tapping into the historical raw data store. To do so, it uses two approaches. First, it stores the count tables in a very compact representation—the count-median sketches—so it can afford to keep plenty of count tables. Second, it includes an automatic process of *count table selection* that inspects the data to identify *feature combinations* worth counting, whether they are used in the current workloads or not. This technique is useful because count featurization tends to obscure correlations between features. For example, different users may have different opinions about specific ads. Although that information could be inferred by a learning algorithm from the raw data points, it is not accessible in the count-featurized data unless we explicitly count the joint occurrences of specific users with specific ads, i.e., maintain a table for the $\langle userId, adId \rangle$ group.

We adapted several feature selection techniques [31] to select feature groups and describe one here. *Mutual Information* (MI) is a measure of dependence between two random variables. A common feature selection technique keeps features of high MI with the label. We extend this mechanism for group count selection. Our goal is to identify feature groups that provide more information about the label than individual features. For each feature x_i, we find all other features x_j such that x_i and x_j together exhibit higher MI with the label than x_i alone. From these groups, we select a configurable number with highest MIs. To find promising groups of larger sizes, we apply this process greedily, trying out new features with existing groups. For each selected group, Pyramid creates and maintains a count table.

This exploration of promising groups operates on the *hot window of raw data*. Because the hot raw data is limited, the selection may not be entirely reliable. Therefore, count tables for new groups are added on a "trial basis." As more data accumulates in the counts, Pyramid re-evaluates them by computing the MI metric *on the count tables*. With the increased amount of data, Pyramid can make a more reliable decision regarding which count tables to keep and which to drop. Because count selection—like feature selection—is never perfect, we give engineers an API to specify groups that they know are worth counting from domain knowledge. Finally, like the weight selection process, count selection should be made differentially private so the groups selected in a particular hot window, which are preserved over time, do not leak information about the window's data in the future. We leave this for future work.

III.C. Supported Workload Evolution

Count featurization is a model-independent preprocessing step, allowing Pyramid to absorb some common evolutions during an ML application's life cycle without tapping the historical raw data store. §V-G gives anecdotal evidence of this claim from a production workload. This section reviews the types of workload changes Pyramid currently absorbs.

A developer may want to change four aspects of the model: (1) the algorithm used to train the model (2) hyperparameters for the model or for the underlying optimization algorithm, (3) features used by the model, and (4) the predicted label. Pyramid supports (1) and (2), partially supports (3), and usually does not support (4).

• *Algorithm changes: Supported.* Pyramid allows developers to move between types of models and libraries used to train those models as long as they are using features and labels that are already counted. In our evaluation we experimented with linear models and neural networks in Vowpal Wabbit [32] and gradient boosted trees in scikit-learn [33] using the same count tables.

• *Hyperparameter tuning: Supported.* By far the most common type of model change we encountered, both in our own evaluation and in reports from a production setting, was hyperparameter tuning. For example, a developer may want to change model hyperparameters, such as the number of hidden units in a neural network, or tune parameters of the underlying optimization algorithm, such as the learning rate or an L1/L2 regularization penalty. Changing hyperparameters is independent from the underlying features so is supported by Pyramid.

• *Feature changes: Partially supported.* Pyramid supports making minimal feature changes. A developer may want to perform one of three types of feature changes: adding new features, removing existing features, or adding interactions between existing features. Pyramid trivially supports removing existing features, and lets developers add new features if they are based on existing ones. For example, the developer could not create an $\langle Age, Location \rangle$ feature interaction if the individual features were not already counted together. Introducing new feature combinations or interactions requires creating new count tables. This highlights the importance of count selection to support workload evolution.

• *Label changes: Mostly unsupported.* Changes in predicted labels are not supported except if a new label is a subset of an existing label. For example, a news recommender could not start predicting retention time instead of clicks unless retention time was previously declared as a label. As with features, Pyramid can support label changes when the new label is a subset of an existing one. For example, if a label exists that tracks retention time in time buckets, Pyramid can support new, coarser labels, such as the three classes "0 seconds," "less than a minute," and "more than a minute."

III.D. Summary

With these components, Pyramid meets the design requirements noted in §II-C, as follows. **R1:** By enhancing the training set with historical statistics gathered over a longer period of time, we minimize the hot data. **R2:** By automatically identifying combinations of features worth maintaining, we avoid having to access the historical raw data for workloads that use the same observation streams to predict the same label. **R3:** By rolling the count windows and retraining the application models, we support data retention policies, albeit at a coarse level. §V evaluates **R4**: accuracy and performance impact.

IV. Prototype

Pyramid is implemented in 2600 lines of Scala, as a modular library. It integrates into the feature engineering stage of an ML pipeline, before the actual learning algorithms are invoked. The modular backend allows count tables to be stored locally in memory or in a remote datastore such as Redis or Cassandra.

We integrated Pyramid into the Velox model management system [25] with minimal effort, by adding/modifying around 500 lines of code. The changes we made to Velox involve interposing on all of Velox's interfaces that interact with raw data (e.g., adding observations, making predictions, and retraining). Now prediction requests are passed through the Pyramid featurization layer, which performs count featurization.

One of Velox's key contributions is performing low latency predictions by pushing models to application servers. To enable low-latency predictions, Pyramid periodically replicates snapshots of the central count tables to the application servers, allowing them to perform featurization locally. §V-E evaluates prediction performance in Velox/Pyramid with and without this optimization.

V. Evaluation

We evaluate Pyramid using different versions of three data-driven applications: two ad targeting applications, two movie recommendation applications, and MSN's production news personalization system. We compare models on count-featurized data to state-of-the-art models trained on raw data, and answer these questions:

Q1. Can we accurately learn on less data using counts?
Q2. How does past-data protection impact utility?
Q3. Does counting feature groups improve accuracy?
Q4. How efficient is Pyramid?
Q5. To what problems does Pyramid apply?

Our evaluation yields four findings: (1) On classification problems, count featurization lets models perform within 4% of state-of-the-art models while training on less than *1% of the data*. (2) Count featurization enables powerful nonlinear algorithms, such as neural networks and boosted trees, that would be infeasible due to high-cardinality features. (3) Protecting individual past observations with differential privacy adds 1% penalty to the accuracy, which remains within 5% of state-of-the-art models. (4) Pyramid's performance overheads are small.

V.A. Methodology

Workloads. Table II shows our apps, datasets, and baselines. We defer discussion of MSN to §V-G.

• *Criteo ad targeting.* Using two versions of the well-known Criteo ads dataset, we build a binary click/no-click classifier. We use seven days of the Criteo ad click dataset amounting to 1.2 billion total observations. This dataset is very imbalanced with an approximate click rate of 3.34%. The second version of the Criteo dataset

App	Dataset	Obs.	Feat.	Baseline
Ad targeting (classification)	Criteo Kaggle [34]	45M	39	neural net in Kaggle [35]
Ad targeting (classification)	Criteo Full [36]	1.2B	39	regularized linear model
Movie recommendation (classification)	MovieLens [37]	22M	21	matrix factorization [32]
Movie recommendation (regression)	MovieLens [37]	22M	21	matrix factorization [32]
News personalization (regression)	MSN.com production	24M	507	contextual bandits [38], [39]

TABLE I: **Workloads.** Apps and datasets; number of observations and features in each dataset; and baselines used for comparison. All baselines are trained using VW [32].

Dataset	Model	Parameters
Criteo-Kaggle	**B:** neural net (nn)	VW. One 35 nodes hidden layer with tanh activation. LR: 0.15. BP: 25. Passes: 20. Early Terminate: 1.
	logistic regression (log. reg.)	VW. LR: 0.5. BP: 26.
	gradient boosted trees (gbt)	Sklearn. 100 trees with 8 leaves. Subsample: 0.5. LR: 0.1. BP: 8.
Criteo-Full	**B:** ridge regression (rdg. reg.)	VW. L2 penalty: $1.5e^{-8}$. LR: 0.5. BP: 26.
MovieLens Regression	**B:** singular value decomposition (svd)	VW. Rank 10. L2 penalty: 0.001. LR: 0.015. BP: 18. Passes: 20. LR Decay: 0.97. PowerT: 0.
	linear regression (lin. reg.)	VW. LR: 0.5. BP: 22. Passes: 5. Early Terminate: 1.
	gradient boosted trees (gbt)	Sklearn. 100 trees with 8 leaves. Subsample: 0.5. LR: 0.1. BP: 8.
MovieLens Classification	**B:** singular value decomposition (svd)	VW. Rank 10. L2 penalty: 0.001. LR: 0.015. BP: 18. Passes: 20. LR decay: 0.97. PowerT: 0.
	logistic regression (log. reg.)	VW. LR: 0.5. BP: 22. Passes: 5. Early Terminate: 1.
	gradient boosted trees (gbt)	Sklearn. 100 trees with 8 leaves. Subsample: 0.5. LR: 0.1. BP: 8.
MSN.com	contextual bandit	VW. IPS context. bandit. LR: 0.02. BP: 18.

TABLE II: **Model parameters.** The libraries and parameters used to train each model. The parameters not noted use library defaults. "LR" indicates the learning rate. "BP" indicates the hash featurization's bit precision (only applicable to raw models). "PowerT" exponent controls learning learning rate decay per step. "**B:**" indicates that the model will be used as a baseline. VW and Sklearn denote that the model was trained with Vowpal Wabbit [32] and scikit-learn [33], respectively.

has 45 million observations, and was released as part of a Kaggle competition. In the Criteo Kaggle dataset, the click and non-click points were sampled at different rates to create a more balanced class split with a 25% click rate. Each observation has 39 features (13 numeric, 26 categorical), and 8 of the categorical features are high dimensional ($> 100K$ values). The numeric features were binned into 4 equal size bins for each dataset. As a baseline, we use a feed-forward neural network that performed well for the competition dataset [35], and we use ridge regression for the full dataset.

• *MovieLens movie recommendation.* Using the well-known MovieLens dataset, which consists of $22M$ ratings on $34K$ movies from $240K$ users, we build two predictors: (1) a regression model that predicts the user's rating as a continuous value in $[0, 5]$, (2) a binary classifier that predicts if a user will give a rating of 4 or more. As a baseline, we use the matrix factorization algorithm in Vowpal Wabbit (VW) [32]; algorithms in this class are

state-of-the-art for recommender systems [40], although this specific implementation is not the most advanced.

Method. For each application, we try a variety of count models, including linear or logistic regression, neural networks, and boosted trees. We split each dataset by time into a training set (80%) and testing set (20%), except for the full Criteo dataset for which we use the first six days for training and the seventh for testing. On the training set, we compute the counts and train our models on windows of growing sizes, where all windows contain the most recent training data and grow backwards to include older data. This ensures that training occurs on the most recent data (closest to the testing set), and that count tables only include observations from the hot window or the past. We use the testing set to compare the performance of our count algorithms to their raw data counterparts and to the baseline algorithms. For all baselines, we apply any dimensionality reduction mechanisms (e.g., hash featurization [41]) that those models typically apply to strengthen them.

Metrics. We use two model accuracy metrics.

(1) The *average logistic loss* for classification problems with categorical labels (e.g. click/no-click). Algorithms predict a probability for each class and are penalized by the logarithm of the probability predicted for the true class: $-\log(p_{true\_class})$. Models are penalized less for incorrect, low-confidence predictions and more for incorrect, high-confidence predictions. Logistic loss is better suited than accuracy for classification problems with imbalanced classes because a model cannot perform well simply by returning the most common class.

(2) The *average squared loss* for regression problems with continuous labels. Algorithms make real-valued predictions that are penalized by the square of the difference with the label: $||prediction - label||^2$.

We conclude our evaluation with our experience with a production setting, in which we can directly estimate click-through rate, a more intuitive metric.

Result interpretation. All graphs report loss normalized by the baseline model trained on the *entire training data*. Lower values are better in all graphs: a value of 1 or less means that we beat the baseline's best performance; and a value > 1 means that we do worse than the baseline.

For completeness, we specify our baselines' performance: MovieLens classification matrix factorization has a logistic loss of 0.537; MovieLens regression matrix factorization has a squared loss of 0.697; Criteo-Kaggle neural network has a logistic loss of 0.467; and Criteo-Full ridge regression has a logistic loss of 0.136.

V.B. Training Set Reduction (Q1)

Pyramid's design is predicated on count featurization's ability to substantially reduce training sets. While this method has long been known, we are unaware of scientific studies of its effectiveness for training set reduction.
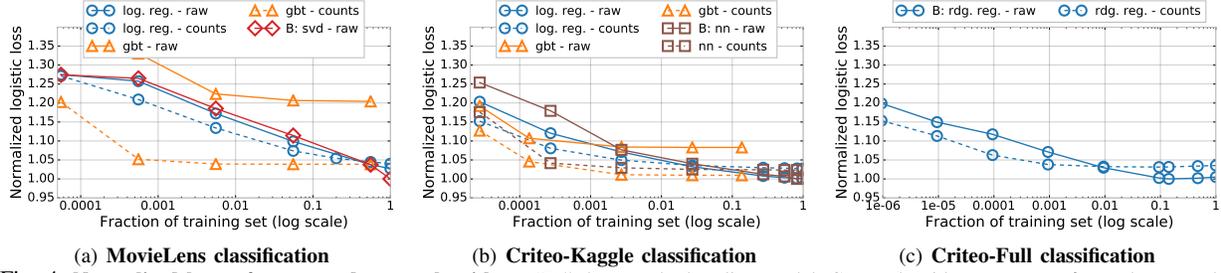
(a) **MovieLens classification** (b) **Criteo-Kaggle classification** (c) **Criteo-Full classification**

Fig. 4: **Normalized losses for raw and count algorithms.** "B:" denotes the baseline model. Count algorithms converge faster than raw data algorithms, to results that are within 4% on MovieLens, and within 2% and 4% on Criteo Kaggle and full respectively.
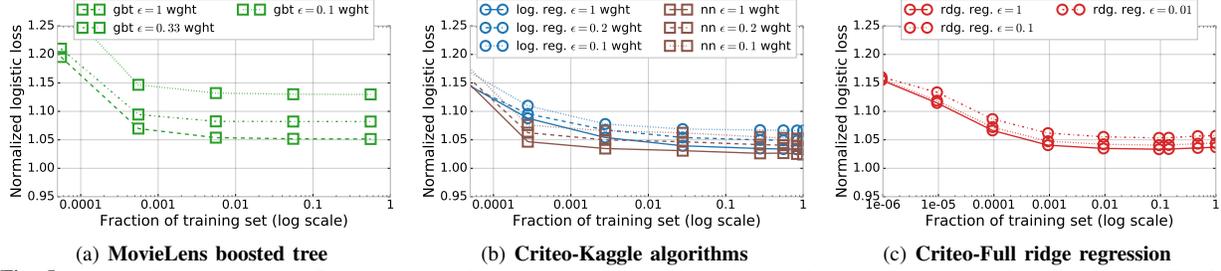

(a) **MovieLens boosted tree** (b) **Criteo-Kaggle algorithms** (c) **Criteo-Full ridge regression**

Fig. 5: **Impact of data protection.** Results are normalized by the baselines. We fix $k = 1$ and vary ϵ, the privacy budget. Fig. 5(a) and Fig. 5(b) show results using the weighted noise (denoted wght). On MovieLens our weighting scheme is crucial to hide 1 observation. On Criteo we can easily hide 1 observation with little performance degradation and can hide up to 100 observations while remaining within 5% of the baseline.

We hence perform a study here. The count models must converge faster than raw-data models (reach their best performance with less data), and perform on par with state-of-the-art baselines. Fig. 4 shows the performance of several linear and nonlinear models, on raw and count-featurized data. We make two observations.

First, **training with counts requires less data**. On both Criteo and MovieLens the best count-featurized algorithm approaches the best raw-data algorithm by training on *1% of the data or less*. On Criteo-Kaggle (Fig. 4(b)), the count-featurized neural network comes within 3% of the baseline when trained on 0.4% of the data and performs within 1.7% of the baseline with 28% of the training data. On Criteo-Full (Fig. 4(c)), the count-featurized ridge regression model comes within 3.3% of the baseline with only 0.1% of the data, and within 2.5% when trained on 15% of the data. These results show that models trained on count-featurized data can perform close to raw models in both balanced and very imbalanced datasets (Criteo Full and Kaggle's respective click rates are 3% and 25%). On MovieLens (Fig. 4(a)), the count-featurized boosted tree needs only 0.8% of the data to get within 4% of the baseline, or match the raw data logistic regression. Because counts summarize history and reduce dimensionality, they allow algorithms to perform well with very little data. We say that they *converge faster* than raw data algorithms.

Second, **counts enable new models**. In Fig. 4, the boosted tree performs poorly on raw data but very well on the count-featurized data. This reveals an interesting insight. The raw-data boosted tree uses a dimensionality reduction technique known as feature hashing [41],

which hashes all categorical values to a limited-size space. This technique exhibits a trade-off: increasing the hash space reduces collisions at the cost of introducing more features, leading to overfitting. Count featurization does not have this problem: a categorical feature is mapped to a few new features (roughly one per label value). This lets us train boosted trees very effectively.

V.C. Past-Data Protection Evaluation (Q2)

We have shown that count-featurized algorithms converge faster than models trained on raw data. This allows Pyramid to keep, and thus expose, only a small amount of raw data to train ML models. However the count tables, while only aggregates of past data, can still leak information about past observations. To prevent such leaks, Pyramid adds differentially private noise to the tables. The amount of noise to add depends on the desired privacy guarantee, parameterized by ϵ (smaller is more private), but also on the number of features (see Table II) and CMS hash functions (five here), through the formula from §III-B2. In this section we evaluate the noise's impact on performance, as well as Pyramid's two mechanisms that increase data utility: automatic weighted noise infusion and the use of private count-median sketches. We also show the impact of the number of windows used, which defines the granularity at which past observations can be entirely dropped.

Impact of noise. Fig. 5 shows the performance of different algorithms and datasets when protecting an observation, $k = 1$, with different privacy budgets ϵ (note the direct tradeoff between the two parameters: the noise is proportional to $\frac{k}{\epsilon}$). We find that Pyramid can protect observations with minimal performance loss.

When $\epsilon = 1$, the boosted tree model on the MovieLens dataset remains within 5% of the baseline with only 1% of the training data. The logistic regression and neural network models on the Criteo-Kaggle dataset perform within 2.7% and 1.8% of the baseline respectively, and the Criteo-Full ridge regression is within 3%. All Criteo models also come within 5% of their respective baseline with a privacy budget as small as $\epsilon = 0.2$.

The Criteo-Full ridge regression performance degrades less than models on other datasets when the noise increases. For instance, it degrades by less than 1% with ϵ going from 1 to 0.1, while the Criteo-Kaggle neural network loses 6.5%. This is explained by the fact that the amount of noise required to make a query differentially private is not related to the size of the dataset. The Criteo-Full dataset is much larger, so the additional noise is much smaller relative to the counts.

Weighted noise infusion. Weighted noise infusion is integral to the protection of past observations with minimal performance cost. Fig. 6(a) shows the impact of noise on the boosted tree for the MovieLens dataset. Without weighting the privacy budget of different features, the model performs 15% worse than the baseline even for $\epsilon = 1$. With weighting, the MovieLens model performs at 5% of the baseline. The weighted noise infusion technique is thus critical to maintaining performance on the MovieLens dataset. Intuitively, this is because the users making the rating and the movie being rated are the most important features when predicting ratings. Most users rate relatively few movies, and a long tail of movies are rarely rated, so their respective counts are quickly overwhelmed by the noise when the privacy budget is equally distributed among all features.

The Criteo models do not depend as much on the weighting trick, since they do not rely on a few features with small counts. Noise weighting is still beneficial, though: e.g., the Criteo-Kaggle neural network gains about 0.5% of performance, as shown in Fig. 6(b).

Private count-median sketch. Another technique that Pyramid uses to reduce the impact of noise is to switch to a private count-median sketch. As noted in §III-B2, the count-min sketch will exhibit a strong downward bias when initialized with differentially private noise, because taking the minimum of multiple observations will select the most extreme negative noise values. The count-median sketch uses the median instead of the minimum and does not suffer from this effect. Fig. 6(c) shows that when noise is added, the count-median sketch improves performance over the count-min sketch by around 0.5%, on MovieLens and Criteo-Kaggle.

When combined with weighted noise infusion, the private count-median sketch is less useful at first, since the noise is small on features with small counts. However, it provides an improvement for lower ϵ. For instance, the

MovieLens boosted tree improves by 0.5% even after noise weighting for $\epsilon = 0.10$.

Number of windows. Another factor impacting accuracy is the number of count windows kept to support granular retention policies. Fig. 7 shows Criteo-Full's ridge regression for $k = 1$ and $\epsilon = 1$ while varying the number of windows. We observe that it is possible to support a large number of windows. On Criteo, we can support 1000 windows with little degradation, enough to support a daily granularity for a multi-year retention period. While we believe this granularity for retention policies should be enough in practice, we also simulated a binary tree scheme [42] that supports huge numbers of windows. We can see that on Criteo, this allows using $100K$ windows with a penalty similar to 10 windows using the basic scheme.

V.D. Count Selection Evaluation (Q3)

Without noise. We measure the performance of our algorithms when the featurization is augmented by MI-selected groups. We evaluate on MovieLens, as groups provided little additional benefit on Criteo. A total of 35 groups were selected by MI and given 10% of the privacy budget to share. When using these groups, the accuracy of the count boosted tree gets within 3% of the baseline with the same 0.8% of the data, 1% better than without feature groups. Logistic regression does not improve asymptotically but converges faster, getting within 5% of the baseline with 15% of the data instead of 22%. Thus, count selection selects relevant groups.

With noise. We also evaluate the impact of group selection on MovieLens with noise $k = 1$, $\epsilon = 1$. Logistic regression is not improved by the grouped features, but the boosted tree is still 1% closer to the baseline. Thus, the algorithm can still extract useful information from the groups despite the increased noise.

While these results are encouraging, we leave for future work the full investigation of how the improvement in accuracy gained from maintaining and using relevant groups is affected by the higher noise levels necessary to maintain a large number of count tables for fixed ϵ.

V.E. Performance Evaluation (Q4)

We evaluate Pyramid's overhead on Velox by measuring the median latency of a prediction request to Velox. We perform this evaluation using the 39-feature Criteo dataset. Fig. 9 shows the median latencies and a breakdown of the time into four components: computing the prediction, unmarshalling the message into a usable form, performing count featurization, and other functions like the network and traversing the web stack. We show the results with and without count table caching in the application servers (§IV). Without caching, prediction latency is around 200ms. Caching reduces it to 1.6ms, a 5% overhead with the total time dominated by the network and traversing the web framework used to
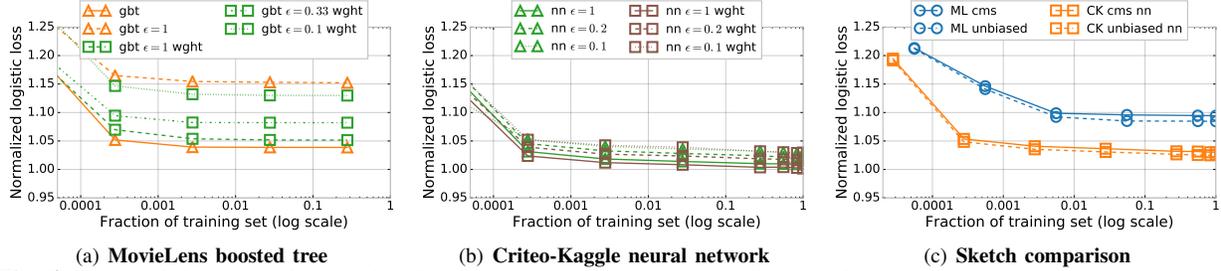
(a) **MovieLens boosted tree**   (b) **Criteo-Kaggle neural network**   (c) **Sketch comparison**

Fig. 6: **Impact of data protection (continued).** Results are normalized to the baselines. We fix $k = 1$ and vary ϵ, the privacy budget. (a) Without the feature weighting trick the gradient boosted trees perform unacceptably poorly. (b) The weighting trick marginally improves the performance of Criteo-Kaggle models over equally distributing the privacy budget. (c) Private count-median sketch improves performance in both MovieLens (ML) and Criteo-Kaggle (CK) models with $\epsilon = 1$.
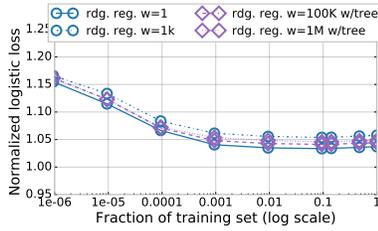

Fig. 7: **Criteo-Full windows.** The Criteo datasets can support 1K windows with reasonable penalty. Supporting more windows requires a scheme based on binary trees.
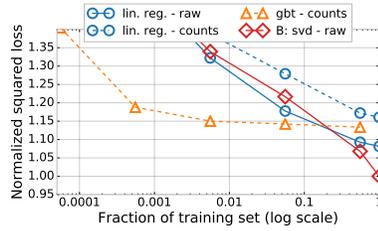

Fig. 8: **MovieLens regression.** Linear regression algorithms are not amenable. Boosted tree converges quickly but does not match the baseline.

Action	P. w/o cache	P. w/ cache	Velox
Featurization	99.22%	4.37%	N/A
Marshalling	0.04%	6.44%	7.06%
Prediction	0.01%	0.51%	0.63%
Network/Framework	0.73%	88.68%	92.31%
Total Latency	283.69 ms	1.65 ms	1.58 ms

Fig. 9: **Prediction Latency.** Median time to serve a model prediction. Caching is crucial for Pyramid to achieve low overhead compared to Velox.

implement Velox. Pushing count tables to the application servers is crucial for performance and does not significantly increase the attack surface.

V.F. Applicability Evaluation (Q5)

Pyramid works well for classification problems. We now consider another broad class of supervised learning problems: regression problems. In regression, the algorithm guesses a label on a continuous scale, and the goal is for the prediction to be as close to the true label as possible. Intuitively, count featurization should be less effective for regression problems, because it needs to bin the continuous label into discrete buckets.

Fig. 8 shows the performance of linear and boosted tree (nonlinear) regressions on the MovieLens dataset. We first observe that linear regression does worse on count-featurized data than on raw data. This is not surprising: count featurization gives the probability of each label conditioned on a feature. The algorithm cannot find a linear relationship between, say, $P(rating = 3|user)$ and the rating. Indeed, the rating does not keep growing with this probability, it keeps getting closer to 3.

Nonlinear algorithms do not have this limitation. The boosted tree converges quickly and outperforms raw models trained on similar amounts of data until we reach 55% of the data. At that point, the boosted tree plateaus and never comes close to the baseline. Although we did not find good algorithms for this dataset, we suspect that some nonlinear algorithms may perform well on counts.

Count featurization is most reminiscent of the counts used by Naive Bayes classifiers [43], and there are
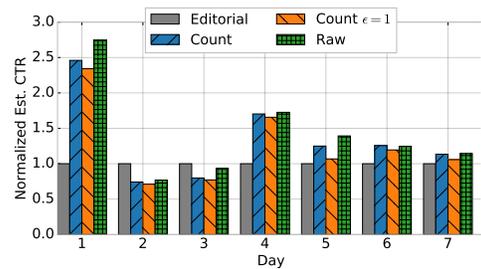

Fig. 10: **Estimated article CTR for MSN.** The raw model, count model, and private count model are normalized against the estimated performance of human editors. The count models perform slightly worse than the raw models; all models outperform human editors on five out of seven days.

workloads for which it is not suitable. For instance, count featurization requires a label and is thus not applicable to unsupervised learning. Other feature representations may be better suited to such types of models. Our choice of count featurization reflects its suitability to data protection in a practical system architecture.

Even in settings that are less amenable to Pyramid, such as online learning applications that avoid retraining, we found that Pyramid can perform well and help protect past observations, as we describe in the next section.

V.G. Experience with a Production Setting

In addition to public datasets, we also evaluated Pyramid on a production workload. One of the authors helped build MSN's news personalization service, which we used to evaluate three aspects: (1) How to adapt count featurization to a different type of learning, (2)

how Pyramid applies to this application, and (3) how Pyramid supports the application's workload evolution. **Adapting count featurization.** MSN uses contextual bandit learning [44], [45] (via the Decision Service [46]) to personalize the order of articles shown to each user so as to maximize clicks, based on 507 features of user demographics and past browsing history. This is a challenging scenario due to the large number of features and low click signal. Contextual bandit algorithms use randomization to explore different action choices, e.g., picking the top article at random. This produces a dataset that assigns a probability (importance weight) to each datapoint. The probabilities are used to optimize models offline and obtain unbiased estimates of their performance had they been run online [38], [39], [47].

Importance-weighted data have interesting implications for Pyramid. When updating the count tables with a given data point, Pyramid must increment the counts by $1/p$, rather than 1, to ensure they remain unbiased. This weighting also increases the noise required for differential privacy, because the sensitivity of a single observation can now be as high as $1/p_{min}$, where p_{min} is the minimum probability of any data point.

With these changes, we built a linear model on count-featurized data and compare it to the (linear) raw-data model used in production. Both models were trained using VW's online contextual bandit learner; in the production system, a snapshot of the model is deployed to application servers every five minutes.

Applicability. Our results suggest that in this application, selectivity is achieved naturally by retaining only the last day of data in the hot window and without the need for Pyramid's training set minimization. This is because news is highly non-stationary: new content appears every hour and breaking news influences people's short-term interests. As a result, even without Pyramid, training models on the last day of raw data is sufficient, and in fact better than training on more days. This is in contrast to the MovieLens and Criteo datasets, which are much more stationary and hence can benefit from Pyramid's training set reduction.

That said, even in non-stationary settings, Pyramid can still enhance data protection through its privacy-preserving counts. We compared the estimated click-through rate (CTR) of the count model (with and without noise) to the raw model across a seven-day period in April 2016. Fig. 10 shows the results relative to the default article ranking by editors. Despite day-to-day variations, on average count models perform within 7% and 13.5% (with noise) of the raw model performance. **Support for workload evolution.** We also assessed how Pyramid would support changes in MSN over time, without accessing the raw data store. MSN developers have spent hundreds (thousands) of human (compute)

hours optimizing the production models. The changes include: tuning hyperparameters and learning rates, adding L1/L2 regularization, testing different exploration rates or model deployment intervals, and adding/interacting/removing features. For example, in some regions regulatory policies prevent certain user data from being collected, so they are removed and models are retrained. Pyramid supports all of the listed changes (§III-C) except adding new features/feature interactions.

VI. Analysis and Limitations

We analyze Pyramid's security properties in the context of our threat model (§II-B), pointing out its limitations. A Pyramid deployment has three components: (1) A central repository of raw data in cold storage that is infrequently accessed and is assumed to be secure. Protecting this data store is outside of Pyramid's scope. (2) A compute/storage cluster used to train models, store the plaintext hot window, and to store and update count tables. (3) Numerous model servers storing trained models and cached versions of count tables.

We first examine the effects of compromising the cluster responsible for training models, maintaining the hot window, and storing the count tables. This will reveal the state of the count tables at time T_{attack}-Δ_{hot} by subtracting all observations residing in the hot window at T_{attack}. Property **P1** in §II-B captures this exposure. However, the observations from the range $[T_{attack}$-$\Delta_{retention}, T_{attack}$-$\Delta_{hot}]$ are protected through differential privacy (property **P2** in §II-B). We expect that the hot window (Δ_{hot}) will be small enough that only a small fraction of an organization's data will be exposed. Observations whose retention period ended before T_{attack} will have been erased, and the models will have been retrained to forget this information (property **P3** in §II-B).

In addition to the hot data, the adversary can siphon observations arriving in the interval $[T_{attack}, T_{attack}^{end}]$. Hence, the amount of data exposed depends on the time to discover and respond to an attack. The sliding nature of Pyramid's hot window gives the organization an advantage when investigating breaches. If an organization knows T_{attack} and T_{attack}^{stop}, it will be able to determine exactly which observations were exposed to the attacker and take the appropriate steps. Knowing these times is only required for post-attack auditing, not for protection of past data during the attack.

Under our current threat model, Pyramid does not protect data from multiple intrusions happening during the same time window. If an attacker accesses Pyramid's internal count tables, that attack is eradicated, and then gains access again at $T_{attack2}$ where $T_{attack2}$ follows T_{attack}^{stop}, the attacker will be able to compute the full fidelity count tables for updates that occurred during the time range $[T_{attack}^{stop}, min(T_{attack2}, T_{win\_end})]$ by subtracting the

state of the count table at T_{attack} from the state of the same count table at $T_{attack2}$. T_{win_end} is the time when Pyramid finishes populating the count table it was populating at T_{attack}^{stop}. One approach to mitigate this attack is to require that Pyramid recomputes count tables after T_{attack}^{stop}, including reinitializing them with new draws from the Laplacian distribution. This will require an increased privacy budget but will still provide a privacy guarantee.

§V demonstrates the need to cache count tables on the application model servers. Attackers that compromise an application server will gain access to the existing cached count table, trained models, and a stream of plaintext prediction requests (unlabeled observations). With access only to the application server the adversary will be able to calculate the difference between the existing count table and new count tables as they are replicated. The adversary will learn little because the difference between the cached count table and the newly replicated count table will be differentially private.

A key limitation of our system stems from our design choice to expose data for a period of time, while it is hot. Data is exposed through the hot data store, trained models, external predictions, and other states that may persist after the data is phased out into the differentially private count tables. There are three implications of this design choice. First, an adversary may monitor these states *before* actually mounting the full-system break-in that Pyramid is designed to protect against (so before T_{start}). §II-B explicitly leaves this attack out of scope. Second, exposing the hot data in raw form to programmers and applications may produce data residues that persist after the data is phased out, potentially revealing past information when an attacker breaks in at T_{start}. For example, a programmer may create a local copy of the hot window at time T for experimentation purposes. While we cannot ensure that state created out-of-band is securely managed, the Pyramid design strives to eliminate any residues for state that Pyramid manages. This is why we enforce model retraining whenever the hot window is rolled over. And this is why we clarify in §III-B2 that the count and weight selection mechanisms should incorporate differential privacy. Third, while the exposed hot data may be small (e.g., 1% of all the data), it may still reveal sufficient sensitive information to satisfy the attacker's goal. Despite these caveats, we believe that our design decision to expose a little hot data affords important practical benefits that would be difficult to achieve with a fully protected design. For example, unlike fully differentially private designs [48], our scheme allows training of *unchanged* ML algorithms with limited impact on their accuracy. Unlike encrypted databases [49], [50], our scheme provides performance and scalability close to—or even better than—running on the raw, fully exposed data.

VII. Related Work

Closest works. Closest to our work are the building blocks we leverage for Pyramid's selective data protection architecture: count featurization and differential privacy. *Count featurization* has been developed and adopted to improve performance and scalability of certain learning systems. We are the first to retrofit it to improve data protection, defining the protection guarantees that can be achieved and implementing them without sacrificing accuracy.

To implement these guarantees, we leverage *differential privacy theory* [51]. The typical threat model for differentially private systems [28], [48], [52] is different from ours: they protect user privacy in the results of a publicly released computation, whereas Pyramid aims to protect data inside the system, by minimizing access to historical data so its accesses can be controlled and monitored more tightly. For example, differential privacy frameworks (e.g., PINQ [28] and Airavat [52], adding privacy to LINQ and MapReduce respectively) ensure that the result of a query will be differentially private. However, these systems require full and permanent access to the data. The same holds for privacy-preserving recommender systems [48]. Pan-privacy [42], [53], [54] is a variant of differential privacy that holds even when an adversary can observe the system's internal state, a threat model close to ours.

Pyramid is the first to combine count featurization with differential privacy for protection.[1] This raises significant challenges at scale, including rampant noise with large numbers of count tables and damaging interference of differential privacy noise with count-min sketches. To address these challenges, our design includes two techniques: noise weighting and private count-median sketches. Prior art, such as iReduct [55] or GUPT [56], included a noise weighting scheme to allocate less of the privacy budget to queries with larger results. To our knowledge, we are the first to point out the limitations of CMS integration with differential privacy and propose private count-median sketches as a solution.

Alternative protection approaches. Many alternative protection models exist. First, many companies enforce a *data retention period*. However, because of the data's perceived benefit, most companies configure long periods. Google maintains data for 9-18 months [57]. Pyramid limits the data's exposure for as long as the company decides to retain it. Second, some companies *anonymize data*: Google erases the last byte of IP addresses in search logs after 6 months [58]. Anonymization provides very weak protection [59]. Pyramid leverages differential privacy to provide rigorous protection guarantees. Third,

---

[1]Azure applies tiny levels of Laplacian noise to count featurization to avoid overfitting, but such low levels neither provide protection nor raise the challenges we encountered.

some companies enforce *access controls* on the data. Google's Sawmill strips out sensitive data before returning results to processes lacking certain permissions [60]. Given the push toward increased developer access to data [5], [6], Pyramid provides additional benefit by protecting data on a needs basis.

Data minimization. Compact data representation is an important topic in big data systems, and many techniques exist for different scenarios. *Sketching techniques* compute compact representations of the data that support queries of summary statistics [26], large-scale regression analysis [61], privacy preserving aggregation [62]; *streaming/online algorithms* [63], [64] process the data using bounded memory, retaining only the information relevant for the problem at hand; *dimensionality reduction techniques* [10] find a low-dimensional, faithful representation of the raw data, according to different measures of faithfulness; *hash featurization* [11] compacts high-cardinality categorical variables; *coresets* [65], [66] are data subsets giving a good approximation for a given computation; *autoencoders* attempt to learn a compressed identity function [67].

We believe that this rich literature should be inspected for candidates for selective data protection. Not all mechanisms will be suitable. For example, according to our evaluation (Fig. 4), hash featurization [11] does not yield sufficient training set reduction. And none of the mechanisms listed above appear to support workload evolution. The next section presents a few promising techniques we have identified.

VIII. Closing: A Vision for Selectivity

We close with our vision for selectivity in big data systems. Today's indiscriminate data collection, long-term archival, and wide-access practices are risky and unsustainable. It is time for a more rigorous and selective approach to big data collection, access, and protection so that its benefits can be reaped without undue risks.

Our vision (illustrated on the right) involves architecting data-driven systems to permit clean separation of data needed by current and evolving workloads, from data collected and archived for possible future needs. The former should be minimized in size and time span (hence the pyramid shape). The latter should be protected vigorously and only tapped under exceptional circumstances. These requirements should be met without disrupting functional properties of the workloads.

The notion of selectivity applies to many big data workloads, including ML and non-ML, and there are perhaps multiple ways to conceptualize the data selectivity problem. For ML workloads, we find that a productive
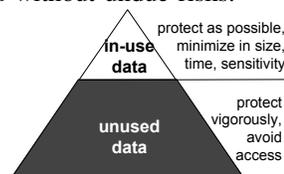
way of identifying potential mechanisms is to model the problem as a *training set minimization problem*. This reveals a rich set of mechanisms that might be leveraged to achieve data selectivity. We have identified several promising mechanisms, which we hope to incorporate into Pyramid for wider workload coverage:

• *Vector quantization (VQ).* VQ [12] is a family of techniques used to compactly represent high dimensional, real-valued feature vectors. At a high level, VQ computes a small subset of vectors, known as the codebook or the centroids, that are representative of the entire set of input vectors (e.g., historical data).

• *Sampling.* Uniform random sampling and more advanced techniques like herding [68] can be used to maintain a representative sample of the historical data. This sample can be combined with in-use data to form a training set. Compared to VQ, which often makes certain assumptions about the underlying data (*e.g.,* that it forms clusters), sampling techniques are more general.

• *Active learning.* Active learning algorithms [15] tell users what specific data points they need for improved accuracy. Originally built to decrease manual labeling, they may be valuable to selective data collection.

We leave investigation of such mechanisms for future work. The key challenge will be to identify the kinds of protection and privacy guarantees achievable with these mechanisms, and how to effectively implement them. This paper provides a first blueprint for this process.

IX. Acknowledgements

References

[1] J. Eng, "OPM hack: Government finally starts notifying 21.5 million victims," http://www.nbcnews.com/tech/security/opm-hack-government-finally-starts-notifying-21-5-million-victims-n437126, 2015.

[2] T. Gryta, "T-Mobile customers information compromised by data breach at credit agency," http://www.wsj.com/articles/experian-data-breach-may-have-compromised-roughly-15-million-consumers-1443732359, 2015.

[3] S. Gorman, "NSA officers spy on love interests," http://blogs.wsj.com/washwire/2013/08/23/nsa-officers-sometimes-spy-on-love-interests/, 2013.

[4] C. Ornstein, "Celebrities medical records tempt hospital workers to snoop," https://www.propublica.org/article/clooney-to-kardashian-celebrities-medical-records-hospital-workers-snoop, 2015.

[5] D. Wilson, "Hearst's VP of data on connecting the data dots," http://www.pubexec.com/article/hearsts-vp-data-connecting-data-dots/, 2014.

[6] L. Rao, "Google consolidates privacy policy; will combine user data across services," http://techcrunch.com/2012/01/24/google-consolidates-privacy-policy-will-combine-user-data-across-services/, 2012.

[7] O. Chiu, "Introducing Azure Data Lake," https://azure.microsoft.com/en-us/blog/introducing-azure-data-lake/, 2015.

[8] B. Schneier, "Data is a toxic asset," https://www.schneier.com/blog/archives/2016/03/data_is_a_toxic.html, 2015.

[9] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "CleanOS: Mobile OS abstractions for managing sensitive data," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[10] C. J. Burges, *Dimension reduction: A guided tour*. Now Publishers Inc, 2010.

[11] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan, "Hash kernels for structured data," *The Journal of Machine Learning Research*, vol. 10, pp. 2615–2637, 2009.

[12] A. Gersho and R. M. Gray, *Vector quantization and signal compression*. Springer Science & Business Media, 2012, vol. 159.

[13] A. Srivastava, A. C. König, and M. Bilenko, "Time adaptive sketches (ada-sketches) for summarizing data streams," in *ACM SIGMOD Conference*. ACM, June 2016.

[14] X. Zhu, "Semi-supervised learning literature survey," 2006.

[15] B. Settles, "Active learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 6, no. 1, pp. 1–114, 2012.

[16] O. Chapelle, E. Manavoglu, and R. Rosales, "Simple and scalable response prediction for display advertising," *ACM Trans. Intell. Syst. Technol.*, vol. 5, no. 4, pp. 61:1–61:34, Dec. 2014.

[17] Y. Chen, D. Pavlov, and J. F. Canny, "Large-scale behavioral targeting," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 209–218.

[18] W. Li, X. Wang, R. Zhang, Y. Cui, J. Mao, and R. Jin, "Exploitation and exploration in a performance based contextual advertising system." in *KDD*, B. Rao, B. Krishnapuram, A. Tomkins, and Q. Yang, Eds. ACM, 2010, pp. 27–36.

[19] M. Bilenko, "Learning with counts," In preparation, 2016.

[20] AzureML, "Build counting transform," https://msdn.microsoft.com/en-us/library/azure/mt243845.aspx, 2016.

[21] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Proceedings of the Third Conference on Theory of Cryptography*, ser. TCC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 265–284.

[22] P. Gutmann, "Secure deletion of data from magnetic and solid-state memory," in *Proc. of USENIX Security*, 1996.

[23] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[24] A. Agresti, *Categorical Data Analysis*, ser. Wiley Series in Probability and Statistics. Wiley, 2013.

[25] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan, "The missing piece in complex analytics: Low latency, scalable model management and serving with Velox," *CoRR*, vol. abs/1409.3809, 2014.

[26] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[27] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ser. ICALP '02. London, UK, UK: Springer-Verlag, 2002, pp. 693–703.

[28] F. D. McSherry, "Privacy integrated queries: An extensible platform for privacy-preserving data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 19–30.

[29] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014. [Online]. Available: http://dx.doi.org/10.1561/0400000042

[30] K. Nissim, S. Raskhodnikova, and A. Smith, "Smooth sensitivity and sampling in private data analysis," in *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '07. New York, NY, USA: ACM, 2007, pp. 75–84. [Online]. Available: http://doi.acm.org/10.1145/1250790.1250803

[31] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *The Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[32] J. Langford, L. Li, and A. Strehl, "Vowpal Wabbit online learning project," 2007.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[34] "Criteo display advertising challenge," https://www.kaggle.com/c/criteo-display-ad-challenge, 2014.

[35] https://www.kaggle.com/c/criteo-display-ad-challenge/discussion/10429#54591, 2014.

[36] "Criteo releases its new dataset," http://labs.criteo.com/2015/03/criteo-releases-its-new-dataset/, 2015.

[37] F. M. Harper and J. A. Konstan, "The MovieLens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, Dec. 2015.

[38] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Intl. World Wide Web Conf. (WWW)*, 2010.

[39] M. Dudík, J. Langford, and L. Li, "Doubly robust policy evaluation and learning," in *Intl. Conf. on Machine Learning (ICML)*, 2011, pp. 1097–1104.

[40] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, no. 8, pp. 30–37, 2009.

[41] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, 2009, pp. 1113–1120.

[42] T.-H. H. Chan, E. Shi, and D. Song, "Private and continual release of statistics," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 3, pp. 26:1–26:24, Nov. 2011.

[43] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.

[44] J. Langford and T. Zhang, "The Epoch-Greedy Algorithm for Contextual Multi-armed Bandits," in *Advances in Neural Information Processing Systems (NIPS)*, 2007.

[45] A. Agarwal, D. Hsu, S. Kale, J. Langford, L. Li, and R. Schapire, "Taming the monster: A fast and simple algorithm for contextual bandits," in *Intl. Conf. on Machine Learning (ICML)*, 2014.

[46] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, S. Sen, and A. Slivkins, "A multiworld testing decision service," *CoRR*, vol. abs/1606.03966, 2016.

[47] L. Li, W. Chu, J. Langford, and X. Wang, "Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms," in *Intl. Conf. on Web Search and Data Mining (WSDM)*, 2011.

[48] F. McSherry and I. Mironov, "Differentially private recommender systems: Building privacy into the Netflix prize contenders," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 627–636.

[49] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 85–100.

[50] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," in *Proceedings of the VLDB Endowment*. VLDB Endowment, 2013.

[51] C. Dwork, "Differential privacy," in *Automata, languages and programming*. Springer, 2006, pp. 1–12.

[52] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for MapReduce." in *NSDI*, vol. 10, 2010, pp. 297–312.

[53] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum, "Differential privacy under continual observation," in *Proceedings of the forty-second ACM symposium on Theory of computing*. ACM, 2010, pp. 715–724.

[54] D. Mir, S. Muthukrishnan, A. Nikolov, and R. N. Wright, "Pan-private algorithms via statistics on sketches," in *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2011, pp. 37–48.

[55] X. Xiao, G. Bender, M. Hay, and J. Gehrke, "iReduct: Differential privacy with reduced relative errors," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 229–240.

[56] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler, "GUPT: Privacy preserving data analysis made easy," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 349–360.

[57] N. Anderson, "Why Google keeps your data forever, tracks you with ads," http://arstechnica.com/tech-policy/2010/03/google-keeps-your-data-to-learn-from-good-guys-fight-off-bad-guys/, 2010.

[58] P. Fleischer, "The European Commision's data protection findings," http://googlepublicpolicy.blogspot.com/2008/04/european-commissions-data-protection.html, 2008.

[59] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 111–125.

[60] A. Becker, "Replacing Sawzall a case study in domain-specific language migration," http://www.unofficialgoogledatascience.com/2015/12/replacing-sawzall-case-study-in-domain.html, 2015.

[61] M. W. Mahoney, "Randomized algorithms for matrices and data," *Foundations and Trends® in Machine Learning*, vol. 3, no. 2, pp. 123–224, 2011.

[62] L. Melis, G. Danezis, and E. De Cristofaro, "Efficient private statistics with succinct sketches," in *Network and Distributed System Security Symposium–NDSS 2016*, 2016.

[63] S. Muthukrishnan, *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

[64] S. Shalev-Shwartz, "Online learning and online convex optimization," *Foundations and Trends in Machine Learning*, vol. 4, no. 2, pp. 107–194, 2011.

[65] D. Feldman, A. Fiat, H. Kaplan, and K. Nissim, "Private core-sets," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*. ACM, 2009, pp. 361–370.

[66] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan, "Geometric approximation via coresets," *Combinatorial and computational geometry*, vol. 52, pp. 1–30, 2005.

[67] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning," 2016, book in preparation for MIT Press. [Online]. Available: http://www.deeplearningbook.org

[68] Y. Chen, M. Welling, and A. Smola, "Super-samples from kernel herding," in *Proceedings of the Twenty-Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI)*. Corvallis, Oregon: AUAI Press, 2010, pp. 109–116.