Protection and the Control of Information Sharing in Machine Learning Systems

Anonymous Submission

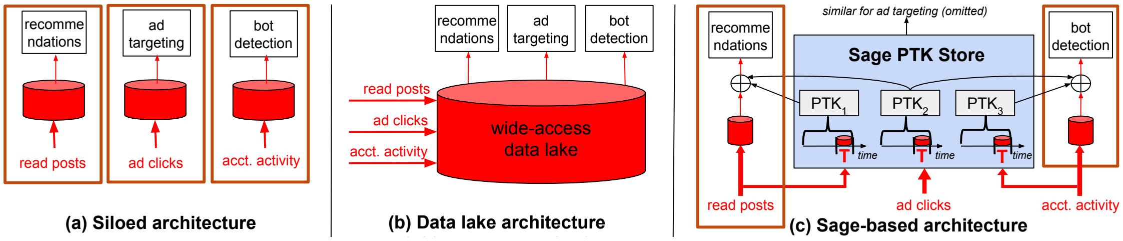
Abstract

1 Introduction

Data-rich, machine learning (ML) ecosystems arising in today's companies, governments, and organizations forgo important principles for rigorous data protection. Consider the *least privilege principle* introduced by Saltzer in the 1974 Multics paper [**?**]: *"[e]very program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job."* This principle has influenced the design of not only Multics' protection system, but also of UNIX and through that of many modern operating systems, including Linux, OSX, and Windows. Unfortunately, in today's data-driven world, this principle appears all but forgotten. Take for instance the "data lake" architecture that is emerging as an ideal data management approach in ML ecosystems [**?**, **?**]: user data from the company's multiple products is collected and integrated into a single, giant repository, which archives it for indefinite time periods and makes it accessible to every data engineer and service within the company who might have some use for it. While this architecture fosters data-driven innovation, it also flaunts the least privilege principle and exposes extensive data stores to "unintentional, unwanted, or improper uses of privilege" by internal parties or external hackers [**?**]. Other principles introduced in the Multics paper, including fail-safe defaults and separation of privilege, are equally shattered by this practice.

We believe that the main reason companies are forsaking good protection principles is the lack of appropriate data protection, sharing, and retention abstractions for ML-driven workloads. Traditional protection and sharing abstractions, such as files, directories, database tables, and views, were all designed for "traditional" software, where it was very clear what data was necessary to implement certain functionality. For example, a "traditional" social networking application might consist of several services, each requiring a different subset of user data to do its job: the authentication service needs access to the user account database; the social networking service needs access to the users' friends lists and posts, but not to the account database or credit card information; and an in-app product purchase service needs access to credit card information but not to the social graph or account information. To meet the principle of least privilege for this application, the company could store the different types of data in different databases or tables, and enable access to the various services and their engineering teams strictly on a needs basis.

In contrast, in emerging ML-driven applications, the question of whether a particular type of data is "necessary" for some functionality is much more blurry. For example, data collected to improve a post recommendation service



(a) Siloed architecture

(b) Data lake architecture

(c) Sage-based architecture

Fig. 1: **Data architectures.** (a) Silo: limited data access. (b) Data lake: wide data access. (c) Sage: limited data access, wide PTK access.

– including the posts previously read or liked by each user – appear relevant not only for that service but also for a friend recommendation service, an ad targeting service, and potentially even for a bot detection service. Similarly, the data collected for ad targeting – such as ad clicks or pages visited on the Internet – may be relevant both for the post and friend recommendation services and for the bot detection service. In some cases, the data may not prove useful in the end, but knowing whether it is useful requires its use – and therefore access to its file, table, or stream – for experimentation and potentially even for evaluation in production.

These blurry lines between what data is necessary vs. not for the various services, processes, and teams within a data-driven company create a tension between an interest in enabling wider access to data within the company, and the privacy, ethical, and exposure concerns raised by cross-product data access and use. Different companies resolve the tension differently, each with its caveat. Fig. 1(a),(b) shows some options. Some companies prefer to silo sensitive data streams to the detriment of functionality (a). Others embrace wider-access, data-lake architectures to the detriment of data protection (b). And yet others choose to share some data streams and silo other streams. Regardless, it is unclear what the protection benefits and risks are of siloing some data streams but not others, and what opportunity costs exist for not leveraging data where it is in fact useful.

We believe that this tension can be addressed with new data protection abstractions that are more suitable for emerging ML workloads than traditional files, tables, or data streams. We propose *private transferrable knowledge* (PTK), a new abstraction for protected data sharing and retention in an ML ecosystem. PTKs are *feature models*, trained over historical data and made *differentially private* (DP) to protect the confidentiality of their training data. Two considerations motivate our abstraction. First, we observe that in certain sharing scenarios, it is not the raw data whose access is actually needed (or even desirable), but historical statistics and well-engineered features extracted from that data. The friend recommendation service may benefit more from incorporating user embeddings already trained and curated over the post service's data, rather than the users' raw posting activity. These embeddings, which map each user to a low-dimension space that captures the similarity among users in terms of the posts they read, write, and like, are usually easier to incorporate in new ML tasks compared to the raw data, which is high-dimensional and messy. User embeddings are already shared across teams at Twitter [**?**]; and Uber and Instacart have developed Feature Stores where teams share with each other tens of thousands of well-engineered features from their data streams [**?**, **?**, **?**]. In none of these cases are the features computed rigorously to protect the confidentiality of their training data. The PTK abstraction formalizes the notion of a DP feature model as a unit for protected data sharing in cases where access to the raw data is not needed. Second, with good features, model training and optimization become (comparably) easy, and typically require less data, because the features already capture important historical characteristics. This provides opportunities for reducing the exposure timeframe of raw data in services that *require* access to it, where PTKs can serve as protected units for long-term retention.

To realize the PTK abstraction, we developed the *Sage PTK Store*, a minimal-exposure system that trains, optimizes, and makes available PTKs on a company's data streams. Fig. 1(c) shows how Sage facilitates a more rigorous raw-data access policy – such as a siloed architecture – by giving wide access to the PTKs instead.

Sage's design takes after existing feature stores, with notable differences. Like in existing feature stores, the developers in charge of some data stream define feature learning procedures and register them with Sage for continuous computation on that data stream. Sage ingests data from these streams to train these feature models and makes the trained models available to others, who incorporate them into their predictive models. Unlike in existing feature stores, Sage: (1) accepts only feature learning procedures that preserve DP guarantees and (2) strives to minimize the exposure of the raw data it ingests so *it* does not violate the least privilege principle or introduce new risks for the company. For (1), we leverage directly the enormous body of existing work on DP learning algorithms [**?**] to develop an initial PTK library, which we envision will be extended in the future to cover most popular feature learning techniques. (2) raises substantial challenges not addressed in DP ML literature.

Specifically, a tension exists between the amount of data Sage needs to retain, and therefore expose, to train multiple PTKs with acceptable performance, versus the strength of the global privacy semantic it enforces across PTKs. That tension increases at scale, when the number of PTKs grows. We develop two new mechanisms to relieve the tension: (1) the first generic method to estimate rigorous sample complexity bounds for DP models that fit the general empirical risk minimization framework []; and (2) the first resource allocation algorithm that assigns training samples and privacy budgets to PTKs that meet a global privacy guarantee while minimizing the amount of data Sage retains for training. With these methods, Sage becomes the *first minimal-exposure DP ML system*.

Using **[ten]** ML tasks from previous Kaggle competitions, we demonstrate two aspects. First, PTKs can be *useful* xxx
units for data protection, enabling transfer learning sharing scenarios without the need to share the raw data, and allowing ML tasks to gain the benefits of historical features without the need to retain the raw data long term. Second, Sage can train effective PTKs while enforcing meaningful privacy guarantees. In all scenarios, using DP features instead of unprotected features (akin to the ones in a vanilla feature store) affects the performance of predictive models by *at most* **[1%]**. Thus, PTKs are not only a useful abstraction but also a practical one that can be implemented in a xxx
data-rigorous way.

2 Goals and Background

Our goal is to develop a *strong-semantic data sharing and retention abstraction* that will enable a more principled approach to tuning the level of access to user data in ML ecosystems. The abstraction should enable companies that currently over-expose user data – through wide-access policies or long retention periods – to limit that exposure without losing much accuracy in their workloads. It should also enable companies that currently over-constrain access to data – through siloed architectures or short retention periods – to increase accuracy in their workloads without increasing exposure.

More specifically, we have three requirements for the abstraction and the system that implements it:

R1: *Suitable abstraction for ML workloads.* The protection abstraction must naturally fit into common patterns in ML ecosystems, and support many data sharing/retention use cases and predictive tasks.

R2: *Limited impact on accuracy and performance.* When used to reduce exposure, the abstraction should result in minimal loss of accuracy, under say 1%, for predictive tasks compared to the best alternative without using it. When used to improve performance, we aim for substantial improvement, above say 5%, to justify any added computational overheads. In all cases, performance overheads for predictive models should be minimal.

R3: *Minimal-exposure system.* The principle of least privilege should apply to the system that implements this abstraction (Sage). Sage should in fact be a model of rigorous data management and enforce a well-defined data exposure semantic under strong threat models relevant to ML ecosystems.

2.1 Threat Model

We consider ecosystems where streams of user data are being collected for use in various predictive ML tasks. We assume that each stream has a set of *primary tasks* which require access to that stream's raw data to function. We additionally assume that there exist other tasks that do not actually need access to the raw data in that stream, but that would benefit from statistics or features learned from the stream in a transfer learning setting. We call these *transfer tasks*, and they are assumed to have available some raw data of their own.

We are concerned with two classes of adversaries who have or gain access to the company's internal state and data stores. First are *abusive employees*, who leverage their privilege within the company to learn about friends' or family members' interactions with the company's products. These adversaries cannot intrude past traditional access controls to escalate their privilege, but can continuously monitor for nefarious purposes any data or state to which they have legitimate access. With the notable exception of the engineers running Sage, all employees constitute a risk. Second are *intruders*, who have no legitimate access to the company's internal state, but who manage to break into its compute resources and gain some access. For example, they might break into an employee's laptop and gain access to any state to which the employee has access. Alternatively, they might identify a vulnerability into a running service (such as a buffer overflow or a heartbleed-like vulnerability) and retrieve the service's state, in memory or on disk.

Intruders can, at worst, access arbitrary state within one or more compromised services, can appear at any time and without prior notice to the company, and can appear multiple times within the lifetime of the company. However, we assume that intruders are not continuously present within the company, nor will they be able to continuously monitor the company's external predictions to its users. Both classes of attacks are highly relevant, as evidenced by numerous media reports exemplifying each: [**?**], [**?**].

The goal of our abstraction is to enable companies to reduce exposure of their user data streams against both types of attackers. First, we wish to enable the protected transfer of features from a source data stream to its transfer tasks without increasing the data's exposure to the potentially abusive employees in charge of these tasks, or to the compromisable services running them. Sharing the raw data constitutes exposure, but so does sharing *any state* computed from the data and not protected with a differential privacy or other cryptographic guarantee. Prior research has shown that even the most innocuous-looking aggregate statistic and the parameters of ML models can reveal a lot of information about individual examples in their training sets []. Second, we wish to enable reduction in the amount of raw user data that is exposed at any time to intruders through its primary task(s). A good approach to such reduction is to limit the data's retention period in its primary tasks.

2.2 Candidate Approaches

A protection abstraction has two parts: a protection mechanism (e.g., access control lists in traditional systems) and a unit for protection (e.g., files in traditional systems). For ML systems, we base our protection abstraction on differential privacy applied to the level of feature models. Other mechanisms and units exist and deserve consideration, however we argue that they are either do not appear as suitable to ML workloads or will likely require combination with our abstraction for sufficient protection against the our adversaries.

As alternative mechanisms, consider leveraging cryptographic mechanisms, such as homomorphic encryption (HE) or secure multi-party computation (MPC), applied to traditional protection units, such as files or streams. Imagine an encrypted data lake architecture, where the raw data is shared widely but in homomorphic-encrypted form; teams and services design and train models "blindly" using homomorphic training algorithms [**?**, **?**]. Alternatively, imagine a siloed architecture, where the raw data is only accessible to the team that collects it; predictive models are trained (again, blindly) on one or more datasets using MPC. Aside from the non-trivial performance concerns of running even state-of-the-art HE and MPC mechanisms, a key limitation of using encryption alone as the protection mechanism in ML systems is that it does not provide adequate protection: the trained models, whose decryption is usually assumed for prediction, can leak information about the training data [**?**, **?**]. The problem can be fixed either by an additional MPC protocol with the end users [**?**] (which add to the performance concern) or by combining HE/MPC with DP.

As an alternative unit of protection, consider applying DP at the level of *SQL queries*. For example, imagine a company requiring that all access to the raw data be done through a DP SQL interface, such as PINQ [**?**] and FLEX [**?**].

For workloads easily written on SQL, such as simple aggregates or learning algorithms that can be efficiently converted to the statistical query model [**?**], this is a good approach. However, for broad classes of learning algorithms, including stochastic gradient descent (SGD), implementing them on SQL would be challenging and inefficient. More generally, we believe that for ML workloads, the SQL interface is a too low-level abstraction at which to enforce protection, and that is why we elevate the level of abstraction to *machine learning models*, and more specifically, to *feature models*. While DP versions exist for virtually all important learning *algorithms* (e.g., [**?**]), the literature is limited on *systems* that manage many DP models with minimal exposure properties.

2.3 Background

Differential Privacy. DP provides a well-defined semantic for preventing leakage of individual records in a dataset through the output of a computation over that dataset. It works by adding randomness into the computation so that details of individual records are "hidden" by the randomness. A (randomized) algorithm A that takes as input a dataset D and outputs a value in a space B is said to satisfy *ϵ-DP at record level* if, for any datasets D and D' differing in at most one record, and for any subset of possible outputs $S \subseteq B$, we have: $P(A(d) \in S) \leq e^{\epsilon} P(A(d') \in S)$. Here, $\epsilon > 0$, called the *privacy budget*, is a parameter that quantifies the strength of the privacy guarantee (lower is better). $\epsilon \leq 1$ is generally considered good protection.

Three important properties of DP are: (1) *Post-processing resilience*: any computation applied on the output of an ϵ-DP algorithm remains ϵ-DP [**?**]. (2) *Composability theorem*: if the same dataset is used as input for two DP algorithms, ϵ_1-DP A_1 and ϵ_2-DP A_2, then the combined privacy semantic is $\epsilon_1 + \epsilon_2$-DP [3]. (3) *Resistance to auxiliary information*: regardless of external knowledge, an adversary with access to the outputs of a DP algorithm draws the same conclusions about the presence/absence of a record in the input dataset [**?**]. A corollary of the last property is that if two DP algorithms, ϵ_1-DP A_1 and ϵ_2-DP A_2, use as input two non-overlapping datasets (perhaps iid partitions of a bigger dataset), then the combined privacy semantic is $max(\epsilon_1, \epsilon_2)$-DP.

Transfer Learning. [Background here on transfer learning and when it's supposed to work and why.] xxx

3 PTK Abstraction

We propose *private transferrable knowledge* (PTK), a new protection abstraction specifically designed for ML systems. Its unit of protection is the *feature model*; its protection mechanism is *differential privacy* (DP). We believe that this abstraction, together with the system that implements it, meets the three requirements we defined in §2. First, feature models are already being used as units for data sharing in today's ML ecosystems []. This encourages us to believe that they will be also serve as natural and suitable units for protected data sharing in these systems (requirement **R1**). Second, the literature is rife with DP implementations of most popular ML algorithms [**?**], and there is increasing experimental evidence that DP is amenable to them [**?**, **?**]. This encourages us to believe that DP can be

```
struct ptk: id, model_state, config, performance_target,
            oldest_window, newest_window, parent, child

// PTK developer API:
ptk.train_dp(trainset_iter, ε_train, previous_model_state,
    previous_config) → overwrite
ptk.eval(testset_iter) → PTK-specific evaluation metric
ptk.profile_dp(profileset_iter, ε_profile, τ) → map{ε_i->n_i}

// PTK user function (not part of API):
featurize(targetset_iter, ptk[]) → featurizedset_iter

// Sage API:
register_ptk(ptk_train_dp_fn, ptk_eval_fn,
    ptk_profile_dp_fn, train_frequency) → ptk_id/reject
deregister_ptk(ptk_id)
update_ptk_config(ptk_id, new_config)
update_ptk_performance_target(ptk_id, new_target)
subscribe_to_ptk(ptk_id, rpc_endpoint)
notify_new_ptk_state(ptk_id)
```

Fig. 2: **PTK API.**

applied to feature models with limited accuracy overheads (requirement **R2**). Finally, §4 discusses how PTKs can be implemented and managed by a scalable system with rigorous guarantees of minimal exposure (requirement **R3**).

The PTK abstraction is used to reenact the principle of least privilege and reduce exposure of the sensitive user data to abusive employees and intruders. Imagine an ML ecosystem where access to raw data is given sparingly to primary tasks that truly need it, and where access is retained by these tasks for bounded time periods. PTKs are used to (1) improve performance in transfer tasks that can leverage their features without requiring access to the raw data (*unit for data sharing*), and (2) improve performance in the primary tasks by letting them use knowledge from historical data without retaining long-term access to those streams (*unit for long-term retention*).

3.1 API

Fig. 2 shows the PTK API, which is used by two entities: the *PTK developer* and one or more *PTK users*.

PTK Developer. The developer starts by defining a procedure to learn useful features from a data stream to which she has access. She turns the feature learning procedure into a PTK by implementing the PTK developer API (described shortly). The developer then identifies a reasonable performance target for the PTK, which Sage will try to honor as it assigns privacy budgets and data samples to train the PTKs on overlapping streams. splits the global privacy parameter

	PTK	**S/U**	**W/C**	`ptk_train_dp`	`ptk_eval`	`ptk_profile_dp`
1	**Count featurization**	S	W	DP contingency tables [**?**]	error from true value	ERM profiling
2	**Tree featurization**	S	W	XXX [**?**]	loss on a holdout set	ERM profiling
3	**User embeddings**	S	C	DP Poisson factorization []	loss on holdout set	ERM profiling
4	**Marginal regression**	S	W	XXX [**?**]	loss on a holdout set	ERM profiling
5	**Covariance matrix**	U	W	XXX []	error from true value	XXX
6	**Historical statistics**	U	W	keyed aggregates w/ Laplace noise []	error from true value	XXX
7	**Clustering**	U	C	DP Lloyd algorithm []	XX	experimental

Tab. 1: **Implemented PTKs.** S/U: supervised/unsupervised. W/C: windowed/continuous training. Core methods used to implement the PTK API.

$\mathcal{E}$ and the data it has available to train multiple PTKs on overlapping streams. Finally, she registers the PTK with Sage and gives Sage access to the source raw data stream on which to train it. Sage assigns a unique ID to the PTK.

PTK User. The user of a PTK can be either the same entity that developed it (primary task) or a separate entity that will use the PTK in a transfer learning setting (transfer task). To use a previously registered PTK, the user subscribes to it by calling `subscribe_to_ptk`, specifying the PTK's ID and an RPC endpoint. Sage trains the PTK periodically, stores them in the widely accessible PTK lake, and notifies the user of new model states available for the PTK. The user retrieves those states from the PTK lake and them incorporates into her predictive models. Our abstraction imposes no API for that, but usually, the PTK user will implement a `featurize` function that transforms her dataset based on one or more PTKs.

PTK Developer API. `ptk.train_dp` is a DP version of the developer's training procedure. It takes a *training set* and a privacy budget, ϵ_{train}, and must satisfy ϵ_{train}-DP. We train PTKs periodically on $\mathcal{T}$-sized time windows (where $\mathcal{T}$ is the exposure window for the PTK's input data stream). We support continuous PTK training across windows by supplying `train_dp` with the previous state of the model trained from the previous $\mathcal{T}$-sized window. `ptk.eval` is a regular evaluation procedure for the PTK that takes in a testing set and returns some PTK-specific evaluation metric, such as a loss metric for a supervised PTK. The `eval` function need not be DP. `ptk.profile_dp` profiles the amount of training data the PTK needs to likely reach a performance target, τ, as a function of the privacy budget it is assigned. The profiling function is supplied with a held-out *profiling set*, separate from the training set, plus a privacy budget, $\epsilon_{profile}$, and the performance target, τ. The function must satisfy $\epsilon_{profile}$-DP.

3.2 Example PTKs

We are implementing *ptklib*, a library of PTKs for popular feature learning algorithms. We use a recent feature engineering textbook [**?**] to prioritize algorithms for implementation. Tab. 1 shows the PTKs implemented so far, the type of algorithm (supervised/unsupervised), whether we train it on a per-window basis or continuously, and the methods we used to implement the developer API for each PTK. For most PTKs, we used known DP versions of training algorithms for `ptk.train_dp`, sensible performance metrics for `ptk.eval`, and a generic profiling method we invented (§4.3) for `ptk.profile_dp`. We describe three PTKs.

Count Featurization PTK. Count featurization is a popular technique for featurizing high cardinality categorical variables, such as user identifiers and IPs, when training ML classification models [**?**, **?**]. The technique replaces each value of the feature vector with the number of times that feature value has been observed with each label and the conditional probability of each label given that feature value. This leads to dramatic dimensionality reduction over standard one-hot encoding, which (intuitively) should allow for more efficient learning and require fewer samples for training. In our system, this means that the PTK user, by featurizing his data with counts and conditional probabilities trained on historical data streams, may be able to learn related labels using less data. §3.3 shows how one can leverage this PTK to reduce raw data retention in primary workloads.

To implement the count featurization PTK on top of a data stream, we compute and publish periodically DP contingency tables of each feature separately with all features that a PTK developer specifies as interesting labels, either for his/her workload or potentially for others in the company. The tables are made DP by initializing their cells with random draws from a Laplace distribution [**?**].

Tree Featurization PTK. The tree featurization PTK leverages a forest of randomized decision trees to learn a set of non-linear features that can be used to increase the performance of predictive models. Decision trees are a set of nested if-else statements where each root to leaf path in a tree corresponds to a non-linear predicate of the input. The tree featurization PTK can featurize data in a number of different fashions. Each leaf node in the tree can be used as a categorical feature so a forest of n decision trees will generate n new categorical features that can be processed with count featurization, hash featurization [4], or count featurization. Users can also use the predictions returned by each tree or by the forest as a whole as features for an application level model. Decision trees have previously been used at Facebook to improve the performance of ad click prediction [2].

[TODO(riley): Describe DP Tree implementation.] xxx

User Embeddings PTK. [TODO(mathias).] xxx

3.3 USAGE

We illustrate the usefulness of PTKs as units for protected sharing and long-term retention using corporate-inspired scenarios that we instantiate and evaluate on public datasets. §5 details the datasets, methodology, and results. We show highlights here to explain usage.

USAGE 1: PTK as Unit of Protected Sharing. *Scenario:* A transportation company is already operating a large fleet of taxis in a city. A different division of that company would like to launch a bike sharing system in the same city. The bike share division would like to use the ride information from the taxi operations to help bootstrap forecasting bike demand in different areas of the city, because they expect that the taxi and bike share data will be used for similar trips. They will use the forecasting to provision bike stations throughout the day. They wish to share PTKs instead of raw data.

We instantiate this scenario on two public datasets: NYC Taxi dataset (33 million taxi rides over three months) and NYC Citibike dataset (3 million bike rides over three months). We use these as the datasets available to the taxi and bike divisions in our scenario, resp. The Citibike task (target task) predicts the destination bike station of each ride out of 465 possible stations.

PTK Usage: The two datasets have several common features: ride origin, destination, date, time of day, and weather information. The Taxi dataset is amenable to tree modeling, so we define a tree PTK on the dataset that uses the common features (except destination), to predict the closest destination bike station for each taxi ride. **[Update.]** We use the tree PTK to generate 465 new features for each CitiBike ride that correspond to the predicted probability of each the 465 destination stations. We use the featurized CitiBike data to train a logistic regression. We find that using the taxi PTKs, the bike task gains significant performance (6%) compared to not using the PTKs (§5.2). Thus, in this scenario, the PTK is a useful unit for sharing and obviates the need to share – and therefore expose – the raw data with the target task. xxx

USAGE 2: PTK as Unit of Long-term Retention. *Scenario:* An online advertising company collects ad clicking activity from its users. The primary task for this data stream predicts the likelihood that an ad will be clicked by a user if shown on a particular page. The team responsible for this task worries about the potential of its members to get hacked. They wish to minimize the amount of ad click data they retain at any time in anticipation of attack. They observe that with good features trained over long periods of time, their models require much less data to converge. They want to leverage PTKs to capture good historical features and safely retain them for long periods of time, while reducing the amount of raw data used to train their predictive models at any time.

We instantiate this scenario on the Criteo dataset [1] (39 million ad impressions over seven days). Each entry has 26 categorical features, 13 integer features, and a binary label indicating if the ad impression resulted in a click. We use the same task as a Kaggle competition organized around this dataset: predicting whether a user will click on an ad. We adopt the winning predictive model: a neural network binary classifier.

PTK Usage: The Criteo dataset contains a number of high-dimensional categorical features; 8 of the 26 categorical features have 100K or more observed values. We therefore hypothesize that count featurization [**?**], which reduces exposure, will help improve the rate of learning for this predictive task and therefore will require models less raw data to converge. We register a count featurization PTK, and train it over XX-hour windows for the entire 7 days of the dataset. We sum the counts across windows for featurization. We train the classifier using the count features in addition to the 18 lower-dimensional raw features. We find that **[XXX]** (§5.2). Thus, in this scenario, the PTK is a useful unit for long-term retention and obviates the need for the primary task to retain access to – and therefore expose – the raw data long-term. xxx

4 Sage Design

To realize the PTK abstraction, we developed the *Sage PTK store*, a system, a minimal-exposure system that periodically trains and makes available PTKs on a company's data streams. Fig. 3 shows its architecture, which consists

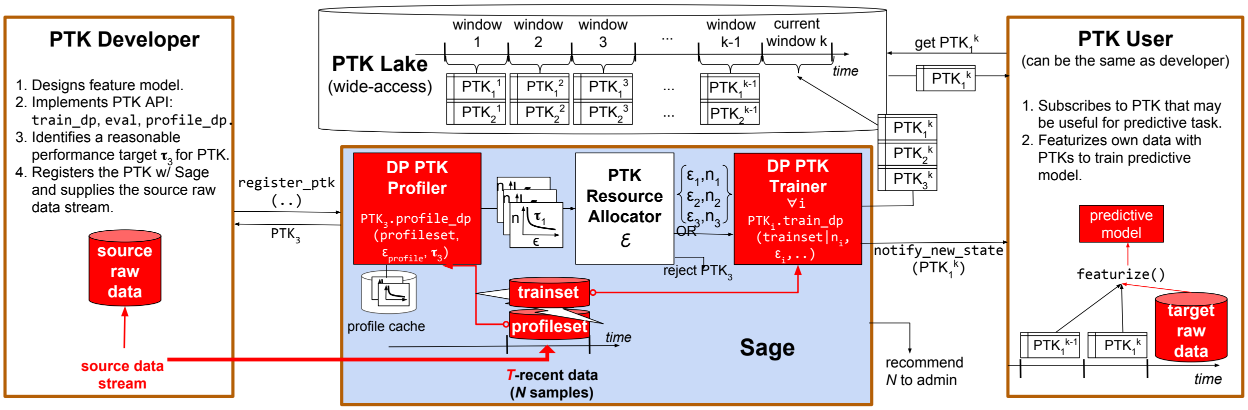

PTK Developer
1. Designs feature model.
2. Implements PTK API: train_dp, eval, profile_dp.
3. Identifies a reasonable performance target τ3 for PTK.
4. Registers the PTK w/ Sage and supplies the source raw data stream.
source raw data
source data stream
register_ptk (..)
PTK3
PTK Lake
(wide-access)
window 1
window 2
window 3
...
window k-1
current window k
time
PTK1 1
PTK2 1
PTK1 2
PTK2 2
PTK1 3
PTK2 3
PTK1 k-1
PTK2 k-1
DP PTK Profiler
PTK3.profile_dp (profileset, εprofile, τ3)
n
τ1
ϵ
profile cache
PTK Resource Allocator
ℰ
ε1,n1
ε2,n2
ε3,n3
OR
reject PTK3
DP PTK Trainer
∀i
PTKi.train_dp (trainset|ni, εi,..)
trainset
profileset
time
T-recent data (N samples)
Sage
PTK1 k
PTK2 k
PTK3 k
notify_new_state (PTK1 k)
recommend N to admin
get PTK1 k
PTK1 k
PTK User
(can be the same as developer)
1. Subscribes to PTK that may be useful for predictive task.
2. Featurizes own data with PTKs to train predictive model.
predictive model
featurize()
target raw data
PTK1 k-1
PTK1 k
time

Fig. 3: **Sage Architecture.** PTK_i: PTK object; PTK_i^j: the state of PTK_i at time window j; $trainset|_{n_i}$: n_i iid samples from trainset.

of two top-level components: *the Sage service*, which ingests several of the company's user streams and trains PTKs registered with it on top of these streams; and a *PTK lake*, a regular database that stores the trained PTKs (including the models' parameters and configurations) and makes them widely accessible to any engineer, team, or service who might have some use for them. A key property of To train the PTKs, Sage retains in

Sage trains the PTKs periodically using data from a recent window of the source data stream, whose timespan we denote $\mathcal{T}$.

4.1 Protection Semantic

The Sage architecture creates two vulnerability points in the face of attackers in our threat model (§2.1), which the Sage protection semantic seeks to eliminate. First, because the PTKs computed by Sage are made widely accessible in the PTK lake, any employee with access to the lake can monitor any and all of these PTKs in an attempt to glean insights about user data streams to which they lack access. To prevent exposure of user data through the PTK lake, Sage enforces across all PTKs a global privacy semantic, $\mathcal{E}$-DP, for $\mathcal{E}$ configured by the Sage administrator (default $\mathcal{E} = 1$).

Second, because the Sage service ingests many streams of user information, it becomes a point of vulnerability for the company. While the Sage administrator is trusted to not abuse the wealth of information, the service itself can be compromised by intruders. To limit the exposure of user information through the Sage service, Sage minimizes the amount of raw data that is exposed at any time through its internal state, such as the $\mathcal{T}$-recent window of raw data it keeps for training PTKs and any other state it maintains. The specific guarantee it provides is as follows: if an intrusion occurs at time T_start and ends at time T_{end} (both times unknown to the company), then the intruder will be able to glean information about raw user data streamed into the Sage service in the interval $[T_{start} - \mathcal{T}, T_{end}]$, but any information from before or after that interval will be protected with an $\mathcal{E}$-DP guarantee. $\mathcal{T}$ is called the *window of exposure* to intrusion attacks and is XX *automatically tuned* by Sage on a per-stream basis to be the timeframe necessary to collect the minimal amount of data that is necessary to achieve particular performance targets for the PTKs registered for that stream.

4.2 Overview

Each PTK is responsible for ensuring that individual model states are ϵ_{train}-DP, where ϵ_{train} is a parameter of the `PTK.train_dp` function. Sage is responsible for enforcing the global $\mathcal{E}$-DP semantic *across* PTKs. Given the DP properties (2) and (3) from §2.3, two approaches arise. One approach is to split the global privacy parameter $\mathcal{E}$ equally across the p PTKs, assigning PTK_i a privacy budget of $\mathcal{E}/p$ and the full $\mathcal{T}$-window of recent data for training. This means calling PTK_i`.train_dp(`$trainset|_{\mathcal{N}}$`,` $\mathcal{E}/p$`, ...)` for each $i = 1..p$. This works if the PTKs can tolerate small privacy budgets, which is not always true. For example, while our count featurization PTK works well even for privacy budgets ≥ 0.001, our tree and user embedding PTKs become worthless at privacy budgets < 1. Regardless, this is not a scalable option for large p. An alternative is to assign all PTKs full privacy budget $\mathcal{E}$ and split the $\mathcal{T}$-recent dataset that Sage keeps for training. This means calling PTK_i`.train_dp(`$trainset|\mathcal{N}/p$`,` $\mathcal{E}$`, ...)` for each $i = 1..p$, where $trainset|_{\mathcal{N}/p}$ takes $\mathcal{N}/p$ iid samples from the available training set. This lets PTKs achieve their best performance under the configured $\mathcal{E}$ privacy guarantee, but will pressure Sage to increase the size of the window of recent data it needs to maintain, thereby increasing the exposure of its ingested data to intruders.

Our approach is to treat the challenge as a *resource allocation problem*, a common problem in systems, and to leverage the canonical systems approach to address it. The resources here are the privacy budget ($\mathcal{E}$) and the $\mathcal{T}$-window of recent data (of size $\mathcal{N}$). The former is a fixed, unscalable resource (cannot be increased as the workload increases); the latter is more scalable, although for minimal-exposure semantics, we wish to minimize $\mathcal{N}$. We need to allocate these resources across p PTKs by assigning each PTK_i privacy budget ϵ_i and a training set consisting of n_i iid samples from the available window of recent data, and invoking PTK_i`.train_dp(`$trainset|_{n_i}$`,` ϵ_i`, ...)` to train it. Our approach, taken from systems, first *profiles* each PTK in isolation, as it is registered with Sage, in terms of how its performance is impacted by various values of ϵ_i, n_i; it then finds an allocation $(\epsilon_i, n_i)\forall i$ that preserves the global privacy guarantee ($\mathcal{E}$) and minimizes the total amount of data needed for training ($\mathcal{N}$) while meeting certain performance targets for the PTKs.

Fig. 3 reflects Sage's key architectural components that implement this procedure: the *PTK Profiler*, *PTK Resource Allocator*, and *PTK Trainer*. At every window k, we partition the $\mathcal{T}$-window of recent data in two: a *profileset* and a *trainset*. The *PTK Profiler* uses the profileset to profile a small set of new or not recently profiled PTKs (PTK_3 in the figure). The profiles, which are made DP, are saved in a profile cache for future reuse. The *PTK Resource Allocator* uses the profiles of all PTKs registered in the system to determine an allocation $(ei, ni)\forall i$ that satisfies the conditions in the preceding paragraph. It is possible that the allocator cannot find such an allocation, in which case it will REJECT the new PTK (PTK_3) or otherwise announce the Sage administrator that a sound allocation does not exist under the current PTK mix, and revert to its previous allocation plan. The *PTK Trainer* uses the allocation to train the PTKs on the *trainset*. Finally, Sage automatically determines a proper value for $\mathcal{N}$, to support both PTK profiling and training.

It recommends the value to the administrator, who can decide to update the $\mathcal{T}$-window of recent data and the exposure allowed through the $\mathcal{T}, \mathcal{E}$-pan privacy semantic.

4.3 PTK Profiler

The PTK Profiler is responsible for profiling one PTK (or a small number of PTKs), on a heldout dataset (profileset), in isolation from one another. Its goal is to estimate the number of samples needed by the PTK to reach a particular performance target as a function of the privacy resource being allocated for training.

Formal Problem.

Optimal Risk Test.

Non-DP Profiling Algorithm.

DP Profiling.

Analysis.

4.4 PTK Resource Allocator

[TODO(roxana): will write next.] xxx

Describe the problem briefly and in intuitive terms. Cast the problem as a resource allocation problem, but with a very non-scalable resource, privacy budget, and a more scalable resource, data.

Problem Formulation. Formalize the problem statement.

Solution. Say how we solve it: we throw it in an LP solver. Say what we do with various answers from it.

5 Evaluation

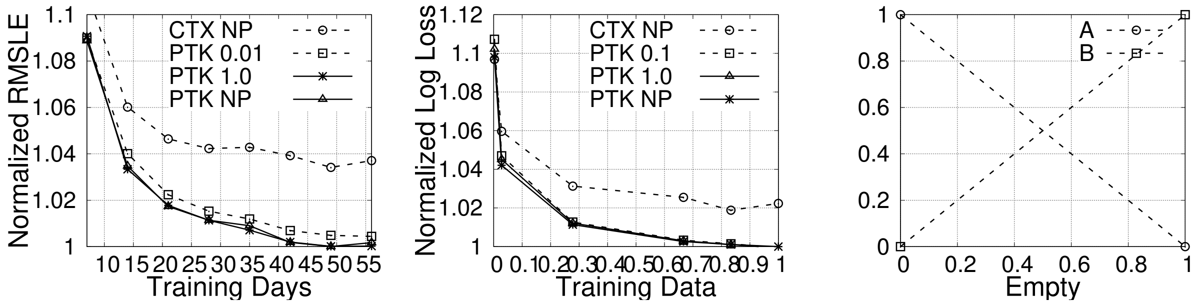
5.1 Methodology

Datasets. We evaluate Sage on five publicly available datasets plus a copy of the Netflix 2005 dataset. Most of the datasets have been used as part of Kaggle competitions; the tasks we define on these datasets for evaluation (described in the next header) are inspired by the tasks formulated in these competitions. We describe the datasets here and give statistics about them in Table **??**.

- *Criteo [**?**]:* See §3.3.
- *Movielens [**?**]:* Ratings 1..5 from XXX users on XXX movies over XXX period. This is an academic dataset. Users were recruited to provide ratings. Time has limited relevance in this dataset. Consists of XXX total examples, each with XXX features, including user ID, movie ID, genre, and integer 1..5 rating. The dataset was the subject of a Kaggle competition to predict the rating of movies (either regression or classification 1..5 task) [**?**]. We use the model that won this competition as baseline: XXX characterize the model.
- *Netflix:* Ratings 1..5 from XXX users on XXX movies over XXX period. The dataset is no longer publicly available; we are using a copy we have from when it was first distributed. Consists of XXX total examples, each

Task	Description	Problem Type	Q	PTKs Used	Predictive Model	Model parameters
T1	Predict ad click on Criteo	binary classif.	Q1	count featurization. 39 features	neural net.	VW. One 35 nodes hidden layer with tanh activation. LR: 0.15. BP: 25. Passes: 20. Early Terminate: 1.
T2	Predict movie rating on Movielens	regression	Q1	XXX	singular value decomposition (svd)	VW. Rank 10. L2 penalty: 0.001. LR: 0.015. BP: 18. Passes: 20. LR Decay: 0.97. PowerT: 0.
T3	Predict movie rating on Netflix with PTKs from Movielens	regression	Q2	1 covariance matrix	ridge regression	TODO
T4	Predict ride duration on Taxi	regression	Q1	9 historical statistics, 2 clustering, 2 PCA PTKs	gradient boosting	XGBoost. LR: 0.05. columns sample: 0.5. min child weight: 75.0. reg. lambda 3.0. 500 estimators. early stopping: 30
T6	Predict ride price on Taxi	regression	Q1	9 historical statistics, 2 clustering, 2 PCA PTKs	gradient boosting	XGBoost. LR: 0.05. columns sample: 0.5. min child weight: 75.0. reg. lambda 3.0. 500 estimators. early stopping: 30
T7	Predict ride destination on Bikes with PTKs from Taxi	465-class classif.	Q2	1 tree PTK	logistic regression	VW. BP: 26. LR: 0.0742. PowerT: 0. Passes: 5.
T8	Predict movie rating on Douban-Movies with PTKs from Douban-Music	classif. 1..5	Q2	1 user embedding PTK	XXX	TODO
T9	Optimize PTKs for Taxi tasks; use them for Taxi tasks	XXX	Q3(a)	classif.	XXX	TODO
T10	Optimize PTKs for Taxi tasks; use them for Bikes task	XXX classif.	Q3(b)	XXX	XXX	TODO

Tab. 2: **Evaluation tasks.** XXX



(a) **Taxi duration prediction**

(b) **Criteo click prediction**

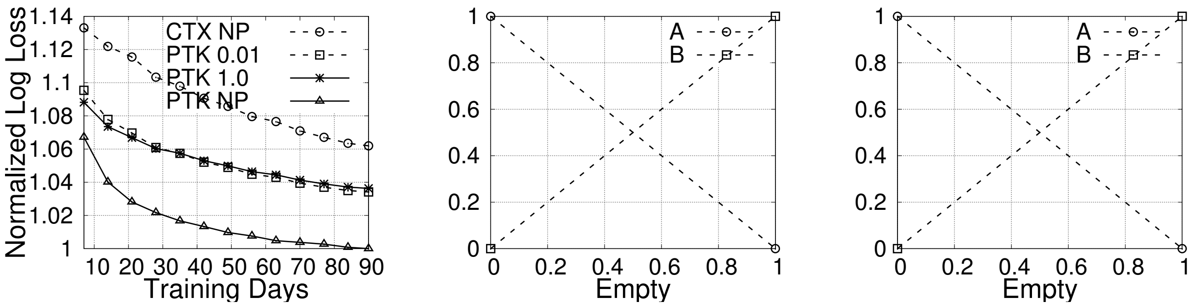
(c) **MovieLens classification**

Fig. 4: **PTKs as units of retention.** XXX PLACEHOLDER IMAGE XXX

with XXX features, including userID, movieID, genre, and an integer 1..5 rating. The dataset was not part of any Kaggle competition. We use it to demonstrate transfer of PTKs from Netflix to Movielens. XXX characterize the model.

- *Taxi [**?**]:* See §3.3.
- *Bikes [**?**]:* See §3.3.
- *Douban [**?**]:* Three datasets with ratings of movies, books, songs, respectively, by 36673 users of the Douban Chinese media company, over between 2005 and 2011. Consists of XXX, XXX, and XXX, resp. examples. Each example has XXX features, including userID, movie/book/songID, genre. XXX Kaggle and model description.

Methodology. XXX Describe methodology briefly.



(a) **Taxi to CitiBike** (b) **Netflix To MovieLens** (c) **Douban to Douban**

Fig. 5: **PTKs as units of protected sharing.** XXX PLACEHOLDER IMAGE XXX

5.2 PTK Usefulness

USAGE 2: PTKs as Protected Unit of Long-term Retention. Figure 4 will have a model for taxi 4(a), criteo 4(b), and MovieLnes 4(c). We may want to consider replacing Movielens with another of the taxi models or something because it may be confusing to have the Movielens classification because we mostly used that before so that it would be useful with count featurization.

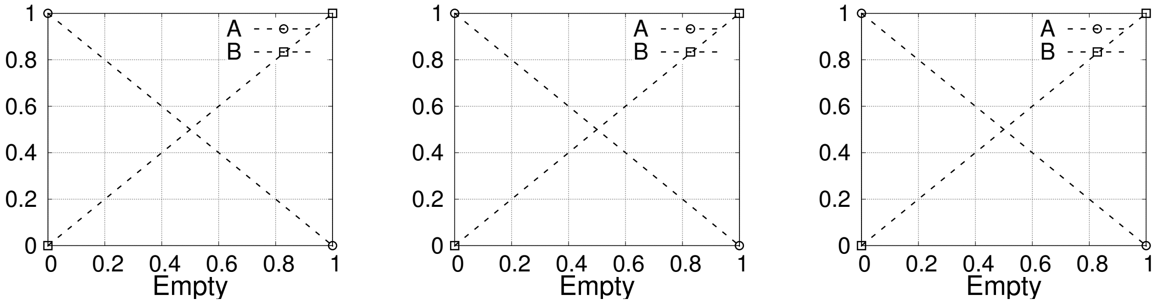
X axis will be amount of training data. Y axis will be the normalized loss normalized to the non private context only model. This is different than what we had before but will match up better with the next section. 4 lines per figure: context only, non private ptk model, private ptk model $\epsilon = 1$, private ptk model $\epsilon = 0.1$.

We have two conclusions that we may be able to draw about PTK's usefulness for retention. First in a scenario like criteo we can leverage ptks to reduce the amount of data required to achieve a decent model (the pyramid conclusion). This is true even if the model is not improved by training the raw+ptk features on the entire dataset. Second in a scenario like the taxi dataset we can say something about PTKs being useful to just improve performance. We may also be able to comment about the dataset being very time dependent and PTKs making data useful if if theadding more data to the model does not improve performance. We can also point out that the performance is improved when adding differential privacy and that we have few PTKs here.

USAGE 1: PTKs as Unit of Protected Sharing.

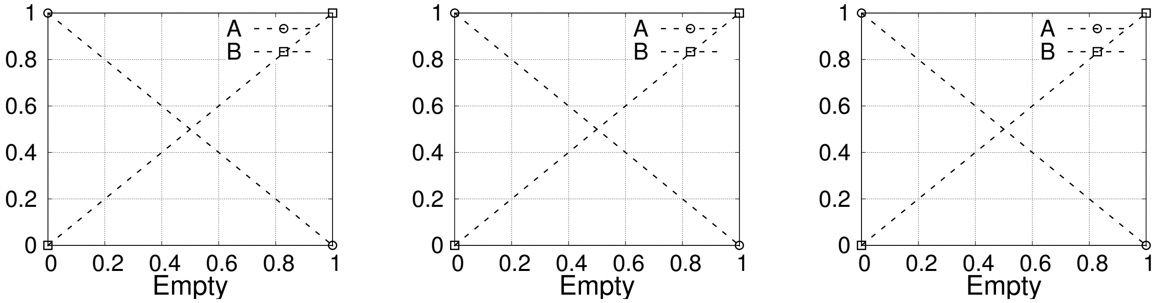
Figure 5 will have a model for taxi to citibike 5(a), netflix to movielens 5(b), and MovieLnes 4(c). This will be the MovieLens as a regression problem.

X axis will be amount of training data. Y axis is the loss normalized to the loss of the model trained on only the target dataset. The baseline model may vary and be a different type of model than what we train using the transferred PTKs. Baselines: CitiBike - VW Logistic regression. This will likely be the same with and without PTKs. MovieLens - VW SVD - The PTK models will be either KNN or ridge regression Douban - VW SVD - not sure what the PTK models will be



(a) **Tree Profile** (b) **Clustering Profile** (c) **Count Featurization Profile**

Fig. 6: **PTK Profiles.** XXX PLACEHOLDER IMAGE XXX This will have privacy budgets on the X axis and the amount of data required to achieve τ_1, τ_2, and τ_3 on the y axis.



(a) **Allocation Model Performance** (b) **Allocation Solving Time** (c) **PTK Prediction Latency**

Fig. 7: **PTK Performance.** XXX PLACEHOLDER IMAGE XXX

For PTKs to be useful as a unit of datasharing the target model must be improved by leveraging the source PTK. It can be improved in 1 of two ways by improving the performance when trained on the entire dataset or by helping the model converge faster which will assist with bootstrapping. We can also comment about DP without contention.

5.3 PTK Resource Allocation

Budget/Data Allocation under Contention. Figure 7(a) will show the performance of the Taxi duration prediction model with 3 lines under the three different allocation models: evenly divide budget, evenly divide data, and the profiling based allocation. The Y axis will be model performance and the x axis will either be the total number of PTKs or the number of PTKs in addition to those used by the model. We can x values of 50, 100, 1000, 10000. I don't think that we should talk about TFX because we never mentioned it before and it'd be confusing to introduce it here.

We exepct to see the performance degrade as we increase the number of PTKs but the profile based allocation should degrade less badly.

PTK Profiles. Figure 6 will show the performance profile of the tree PTK **??** and the clustering PTK **??** on the taxi dataset, and the count featurization ptk **??** on the criteo dataset. It may be better to include three from the same dataset but we can decide later exactly which figures we include here based on which are interesting. [**Instead of clustering, it would be better to use PCA or something for which we have a profiling approach.** —RG] xxx

The X axis will be privacy budget and the Y axis will be the amount of data required to meet the performance goal. There will be three performance goals: τ_1, τ_2, and τ_3. There will be two lines for each performance goal. The predicted amount of data to reach τ and the actual amount.

What the profiles look like for various PTKs and at various τ values. Compare to the experimental version of determining the profile: i.e., determine experimentally (e.g., using a binary search algorithm) how much data you need to reach τ.

Amounts of Data Used by Allocation. Plot or report the amount of data required to allocate all of the PTKs using our allocation procedure (sum over all partitions). You can place these numbers on top of the corresponding bars. In this little paragraph, refer to these numbers and comment that they are "reasonable."

5.4 Performance

Figure 7(c) show the prediction latency of two models Taxi duration prediction and citibike in the context only and the ptk configurations. The X axis will be percentile and the Y axis will show the latency. We expect that both of the context only models will lowest tail latency. I expect both of the PTK models will have a much higher tail latency. The CitiBike model because random forests often have a long tail latency and the taxi models because it will require two rounds of featurization. Both will still probably be dominated by network time.

Figure 7(b) will show the time required to solve the budget allocation problem after we have generated the profiles for each of the PTKs. The X axis will be the number of PTKs and the Y axis will be the time required to solve the MILP problem. We expect the time to polynomialy in the number of PTKs for which we're solving. Since this is under something like a day it should be fine since this will be run infrequently.

6 Analysis

Need a security analysis here to clarify the threats that PTKs leave open and re-enforce why we believe we still add benefits.

Return to Multics principles and see how we redeem not only least privilege but also others.

Threats:

1. Ecosystem-wide threats: - A malicious employee will be able to access every data stream, state and service to which he is granted access. They can monitor the raw data, the internal state of servers, and the service's predictions to the users. However, if these resources are correctly siloed from employees who do not need access to them, then he won't be able to access them. The PTK abstraction aims to constrain the reach of any individual employee to the minimum they need, and assumes that access controls and firewalls correctly implement minimal access policies.

- An intruder into the ML ecosystem can, in the limit, retrieve arbitrary state in all of the predictive tasks/teams. This includes the state that Sage maintains, plus The PTK abstraction aims to constrain the reach of an intruder by allowing individual primary tasks to reduce their access retention periods by leveraging historical PTKs. Hence, the intruder will be able to retrieve data in the maximum retention period for all primary tasks for a particular data stream.

- Many attacks against integrity/availability: - Malicious employee registers a lot of PTKs with high eps. Won't get Sage to increase privacy budget beyond $\mathcal{E}$, so confidentiality wise we are OK. But it may get Sage to increase $\mathcal{T}$, at least to some point. We envision that the Sage admin will bound the max T they wish to enforce. The attacker may also cause extreme contention and hence new PTKs may be rejected by Sage. - Train bogus PTK, change a good PTK into worse. That sabotages the workloads that rely on the PTK.

2. Sage threats: Fewer, because Sage enforces a strict semantic. But there are a few caveats. - An intruder who breaks into the Sage service may change $\mathcal{E}$ or the bound on T. This will get Sage to weaken its guarantees. We assume that such changes are not possible without the (trusted) administrator's knowledge.

- When PTKs are registered, they are assumed to already have been designed. That design is usually done through a lot of experimentation and optimization on a dataset. The PTK's training procedure, can (in theory) reveal information about the dataset the developer used to design the PTK. We believe this is reasonable, but we invite care into how that is done and awareness XXX.

- A bad (or malicious) programmer may break the assumptions we make about the PTK functions, e.g., that `ptk.train_dp` and `ptk.profile_dp` satisfy the specified DP semantic for all of their outputs and side-effects. This can lead to leakage of user data the through the PTK. A natural solution would be to require that PTK functions must be certified through code review to be accepted by Sage.

The difference between Sage and the rest of the ecosystem is that Sage*enforces* the minimal-exposure policy while the rest of the ecosystem is merely encouraged to do so by leveraging Sage's abstractions. However, we acknowledge that enforcing minimal-exposure is non-trivial even with the PTK abstractions: when data is expired, all of its traces must be securely erased, all non-DP models based on them must be retrained without it, and worst of all; to determine, one needs to profile his model's dependency on the amount of data, akin to how we do it, and tune that with changes in the data.

Sage provides a few of these services, and could in theory be used to secure predictive models, too. They could develop DP versions of their predictive models and register them with Sage for periodic profiling, training, and releasing. This is useful to gain DP under continual output observation for the predictive tasks/teams. To get pan privacy for these tasks/teams, that's much harder, because it most likely means that once they register the predictive PTK, the engineers should no longer have access to the data. This means they won't be able to design and bootstrap new, improved versions of the predictive models, which is an important part of engineers' duties in an ML ecosystem. Designing an ecosystem-wide, $\mathcal{T},\mathcal{E}$-pan private system is very much an open problem.

7 Related Work

Data exposure in ML systems is a well recognized problem, for which multiple approaches have been proposed in the past. Neither meets our goals and requirements. First, one might use homomorphic encryption to train all predictive models in an ML ecosystem on encrypted data. Recent advances in such techniques have led to rather practical data mining algorithms []. However, this approach is insufficient for our needs: the trained models, whose decryption is ultimately needed to make predictions, can reveal a lot about the data on which they were trained []. Addressing this challenge means ensuring that the computation to be performed homomorphically also be made DP. Under our threat model, this appears to require functional encryption [], which in general is very expensive.

Second, one might use secure multi-party computation (MPC) to compute models across teams without exposing the raw data []. As with homomorphic encryption, one would need to combine MPC with DP. Even so, efficient, general-purpose MPC is an unresolved challenge.

Third, a company might require that all access to the raw data be done through a DP SQL interface, such as PINQ [] and FLEX []. For workloads easily written on SQL, that is a good approach; in fact, our PTK library could leverage such interfaces to support general historical statistics. However, for complex ML training algorithms, such as stochastic gradient descent (SGD), it is difficult to think about how one would implement them efficiently on SQL. More generally, we believe that for ML workloads, the SQL interface is a too low-level abstraction at which to enforce protection, and that is why we elevate the level of abstraction to *feature models*.

8 Conclusion

References

[1] Kaggle display advertising challenge dataset. `http://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/`, 2014.

[2] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9. ACM, 2014.

[3] 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*, SIGMOD '09, pages 19–30, New York, NY, USA, 2009. ACM.

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

A PTK Profiling Proofs

B Sage Semantic Proofs