DDUO: General-Purpose Dynamic Analysis for Differential Privacy

Chike Abuah¹, Alex Silence¹, David Darais², and Joe Near¹

¹Computer Science Department, University of Vermont
²Galois, Inc.

Abstract—Differential privacy enables general statistical analysis of data with formal guarantees of privacy protection at the individual level. Tools that assist data analysts with utilizing differential privacy have frequently taken the form of programming languages and libraries. However, many existing programming languages designed for compositional verification of differential privacy impose significant burden on the programmer (in the form of complex type annotations). Supplementary library support for privacy analysis built on top of existing general-purpose languages has been more usable, but incapable of pervasive end-to-end enforcement of sensitivity analysis and privacy composition.

We introduce DDUO, a dynamic analysis for enforcing differential privacy. DDUO is usable by non-experts: its analysis is automatic and it requires no additional type annotations. DDUO can be implemented as a library for existing programming languages; we present a reference implementation in Python which features moderate runtime overheads on realistic workloads. We include support for several data types, distance metrics and operations which are commonly used in modern machine learning programs. We also provide initial support for tracking the sensitivity of data transformations in popular Python libraries for data analysis.

We formalize the novel core of the DDUO system and prove it sound for sensitivity analysis via a logical relation for metric preservation. We also illustrate DDUO’s usability and flexibility through various case studies which implement state-of-the-art machine learning algorithms.

I. INTRODUCTION

Differential privacy has achieved prominence over the past decade as a rigorous formal foundation upon which diverse tools and mechanisms for performing private data analysis can be built. The guarantee of differential privacy is that it protects privacy at the individual level: if the result of a differentially private query or operation on a dataset is publicly released, any individual present in that dataset can claim plausible deniability. This means that any participating individual can deny the presence of their information in the dataset based on the query result, because differentially private queries introduce enough random noise to make the result indistinguishable from that of the same query run on a dataset which actually does not contain the individual’s information. Additionally, differential privacy guarantees are resilient against any form of linking attack in the presence of auxiliary information about individuals.

High profile tech companies such as Google have shown a commitment to differential privacy by developing projects such as RAPPOR [53] as well as several open-source privacy-preserving technologies [27], [28], [47]. Facebook recently released an unprecedented social dataset, protected by differential privacy guarantees, which contains information regarding people who publicly shared and engaged with about 38 million unique URLs, as an effort to help researchers study social media’s impact on democracy and the 2020 United States presidential election [38], [32], [33], [22], [23]. The US Census Bureau has also adopted differential privacy to safeguard the 2020 census results [2].

Both static and dynamic tools have been developed to help non-experts write differentially private programs. Many of the static tools take the form of statically-typed programming languages, where correct privacy analysis is built into the soundness of the type system. However, existing language-oriented tools for compositional verification of differential privacy impose significant burden on the programmer (in the form of additional type annotations) [40], [25], [39], [17], [52], [49], [8], [10], [11], [9], [43], [4], [51], [46], [13], [18], [45] (see Section IX for a longer discussion).

The best-known dynamic tool is PINQ [35], a dynamic analysis for sensitivity and privacy. It features an extensible system which allows non-experts in differential privacy to execute SQL-like queries against relational databases. However, PINQ comes with several restrictions that limit its applicability. For example, PINQ’s expressiveness is limited to a subset of the SQL language for relational databases. Methods in PINQ are assumed to be side-effect free, which is necessary to preserve their privacy guarantee.

We introduce DDUO, a dynamic analysis for enforcing differential privacy. DDUO is usable by non-experts: its analysis is automatic and it requires no additional type annotations. DDUO can be implemented as a library for existing programming languages; we present a reference implementation in Python. Our goal in this work is to answer the following four questions, based on the limitations of PINQ:

- Can a PINQ-style dynamic analysis extend to base types in the programming language, to allow its use pervasively?
- Is the analysis sound in the presence of side effects?
- Can we use this style of analysis for complex algorithms like differentially private gradient descent?
- Can we extend the privacy analysis beyond pure $\epsilon$-differential privacy?

We answer all four questions in the affirmative, building on...
PINQ in the following ways:

- **DDUO** provides a dynamic analysis for base types in a general purpose language (Python). **DDUO** supports general language operations, such as mapping arbitrary functions over lists, and tracks the sensitivity (stability) and privacy throughout.
- Methods in **DDUO** are not required to be side-effect free and allow programmers to mutate references inside functions which manipulate sensitive values.
- **DDUO** supports for various notions of sensitivity and arbitrary distance metrics (including \( L_1 \) and \( L_2 \) distance).
- **DDUO** is capable of leveraging advanced privacy variants such as \((\epsilon, \delta)\) and Rényi differential privacy.

Privacy analysis is reliant on sensitivity analysis, which determines the scale of noise an analyst must add to values in order to achieve any level of privacy. Dynamic analysis for differential privacy is thus a dual challenge:

**Dynamic sensitivity analysis.** Program sensitivity is a (hyper)property quantified over two runs of a program with related inputs (sources). A major challenge for dynamic sensitivity analysis is the ability to bound sensitivity, ensuring that the metric preservation property is satisfied, by only observing a single run of the program. In addition, an analysis which is performed on a specific input to the program must generalize to future possible arbitrary inputs.

The key insight to our solution is attaching sensitivity environments and distance metric information to values rather than variables. Our approach provides a sound upper bound on global sensitivity even in the presence of side effects, conditionals, and higher-order functions. We present a proof using a step-indexed logical relation which shows that our sensitivity analysis is sound.

**Dynamic privacy analysis.** To implement a dynamic privacy analysis, we leverage prior work on privacy filters and odometers [41]. This work, originally designed for the adaptive choice of privacy parameters, can also be used as part of a dynamic analysis for privacy analysis. We view each application of a privacy mechanism (e.g. the Laplace mechanism) as a global privacy effect on total privacy cost, and use privacy filters and odometers to track total privacy cost.

We implemented these features in a Python prototype of **DDUO** via object proxies and other pythonic idioms. We implement several case studies to showcase these features and demonstrate the usage of **DDUO** in practice. We also provide integrations with several popular Python libraries for data and privacy analysis.

**Contributions.** In summary, this paper makes the following contributions:

- We formalize a subset of **DDUO** in a core language model, and prove the soundness of **DDUO**’s dynamic sensitivity analysis (as encoded in the model) using a step-indexed logical relation.
- We present several case studies demonstrating the use of **DDUO** to build practical, verified Python implementations of complex differentially private algorithms.

The rest of the paper is organized as follows. First we provide some background knowledge regarding the field of differential privacy (Section II). We provide an overview of our work and the necessary tradeoffs (Section III). We illustrate the usefulness and power of **DDUO** through some worked examples (Section IV). We then discuss some of the nuances of dynamic sensitivity (Section V) and dynamic privacy tracking (Section VI). We present the formalization of **DDUO** and prove the soundness of our sensitivity analysis (Section VII). We provide several case studies demonstrating the usefulness of **DDUO** in practice (Section VIII). Finally we outline related work (Section IX) and conclude (Section X).

**II. BACKGROUND**

**Differential Privacy.** Differential privacy is a formal notion of privacy; certain algorithms (called mechanisms) can be said to satisfy differential privacy. Intuitively, the idea behind a differential privacy mechanism is that: given inputs which differ in the data of a single individual, the mechanism should return statistically indistinguishable answers. This means that the data of any one individual should not have any significant effect on the outcome of the mechanism, effectively protecting privacy on the individual level. Formally, differential privacy is parameterized by the privacy parameters \( \epsilon, \delta \) which control the strength of the guarantee.

When we say “neighboring” inputs, this implies two inputs that differ in the information of a single individual. However, formally we can defer to some general distance metric which may take several forms. We then say that according to the distance metric, the distance between two databases must have an upper bound of 1. Variation of the distance metric has led to several other useful, non-standard forms of differential privacy in the literature.

**Definition II.1** (Differential privacy). Given a distance metric \( d_A \in A \times A \rightarrow \mathbb{R} \), a randomized algorithm (or mechanism) \( M \in A \rightarrow B \) satisfies \((\epsilon, \delta)-\)differential privacy if for all \( x, x' \in A \) such that \( d_A(x, x') \leq 1 \) and all possible sets \( S \subseteq B \) of outcomes, \( \Pr[M(x) \in S] \leq e^\epsilon \Pr[M(x') \in S] + \delta \).

Differential privacy is compositional: running two mechanisms \( M_1 \) and \( M_2 \) with privacy costs of \((\epsilon_1, \delta_1)\) and \((\epsilon_2, \delta_2)\) respectively has a total privacy cost of \((\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)\). Advanced composition [19] improves on this composition bound for iterative algorithms; several variants of differential privacy (e.g. Rényi differential privacy [36] and zero-concentrated differential privacy [15]) have been developed that improve the bound even further. Importantly, sequential composition theorems for differential privacy do not necessarily allow the
privacy parameters to be chosen adaptively, which presents a special challenge in our setting—we discuss this issue in Section VI.

III. OVERVIEW OF DDUO

DDUO is a dynamic analysis for enforcing differential privacy. Our approach does not require static analysis of programs, and allows DDUO to be implemented as a library for programming languages like Python. DDUO’s dynamic analysis has complete access to run-time information, so it does not require the programmer to write any additional type annotations—in many cases, DDUO can verify differential privacy for essentially unmodified Python programs (see the case studies in Section VIII). As a Python library, DDUO is easily integrated with popular libraries like Pandas and NumPy.

Threat model. We assume an “honest but fallible” programmer—that is, the programmer intends to produce a differentially private program, but may unintentionally introduce bugs. We assume that the programmer is not intentionally attempting to subvert DDUO’s enforcement approach. Our reference implementation is embedded in Python, an inherently dynamic language with run-time features like reflection. In this setting, a malicious programmer or privacy-violating third-party libraries can bypass our dynamic monitor and extract sensitive information directly. Data-independent exceptions can be safely used in our system, however our model must explicitly avoid data-dependent exceptions such as division-by-zero errors. Terminated programs can be rerun safely (while consuming the privacy budget) because our analysis is independent of any sensitive information (our metatheory implies that sensitivity of a value is itself not sensitive). We also do not address side-channels, including execution time. Like existing enforcement approaches (PINQ, OpenDP, Diffprivlib), DDUO is intended as a tool to help well-intentioned programmers produce correct differentially private algorithms.

Soundness of the analysis. We formalize our dynamic sensitivity analysis and prove its soundness in Section VII. Our formalization includes the most challenging features of the dynamic setting—conditionals and side effects—and provides evidence that our Python implementation will be effective in catching privacy bugs in real programs. DDUO relies on existing work on privacy filters and odometers (discussed in Section VI), whose soundness has been previously established, for tracking privacy cost.

IV. DDUO BY EXAMPLE

This section introduces the DDUO system via examples written using our reference Python implementation.

Data Sources. Data sources are wrappers around sensitive data that enable tracking of privacy information in the DDUO python library. Each data source is associated with an identifying string, such as the name of the input file the data was read from. Data sources can be created manually by attaching an identifying string (such as a filename) to a raw value (such as a vector). Or, data sources be created automatically upon loading data through DDUO’s custom-wrapped third party APIs, such as pandas. Note that our API can be easily modified to account for initial sensitivities greater than 1 when users have multiple datapoints in the input data.

```python
from duo import pandas as pd
df = pd.read_csv("data.csv")
```

```plaintext
Sensitive(<'DataFrame'>, {data.csv -> 1}, \infty)
```

A Sensitive value is returned. Sensitive values represent sensitive information that cannot be viewed by the analyst. When a Sensitive value is printed out, the analyst sees (1) the type of the value, (2) its sensitivity environment, and (3) its distance metric. The latter two components are described next. The analyst is prevented from viewing the value itself.

Sensitivity & distance metrics. Function sensitivity is a scalar value which represents how much a change in a function’s input will change the function’s output. For example, the binary addition function \( f(x, y) = x + y \) is 1-sensitive in both \( x \) and \( y \), because changing either input by \( n \) will change the sum by \( n \). The function \( f(x) = x + x \), on the other hand, is 2-sensitive in its argument \( x \), because changing \( x \) by \( n \) changes the function’s output by \( 2n \). Sensitivity is key to differential privacy because it is directly proportional to the amount of noise we must add to the output of a function to make it private.

DDUO tracks the sensitivity of a value to changes in the program’s inputs using a sensitivity environment mapping input data sources to sensitivities. Our example program returned a Sensitive value with a sensitivity environment of \( \{\text{data.csv} \rightarrow 1\} \), indicating that the underlying value is 1-sensitive in the data contained in data.csv. The DDUO library tracks and updates the sensitivity environments of Sensitive objects as operations are applied to them. For example, adding a constant value to the elements of the DataFrame results in no change to the sensitivity environment.

```plaintext
df + 5 # no change to sensitivity environment
```

Adding the DataFrame to itself doubles the sensitivity, in the same way as the function \( f(x) = x + x \).

```plaintext
df + df # doubles the sensitivity
```

```plaintext
Sensitive(<'DataFrame'>, {data.csv -> 2}, \infty)
```

Finally, multiplying the DataFrame by a constant scales the sensitivity, and multiplying the DataFrame by itself results in infinite sensitivity.

```plaintext
( df + 5, df + df)
```
The distance metric component of a Sensitive value describes how to measure sensitivity. For simple numeric functions like \( f(x) = x + x \), the distance between two possible inputs \( x \) and \( x' \) is simply \( |x - x'| \) (this is called the cartesian metric). For more complicated data structures (e.g. DataFrames), calculating the distance between two values is more involved. The \( L_\infty \) metric used in our example calculates the distance between two DataFrames by measuring how many rows are different (this is one standard way of defining “neighboring databases” in differential privacy). DD\textsubscript{DUO}’s handling of distance metrics is detailed in Section V-B.

Privacy. DD\textsubscript{DUO} also tracks the privacy of computations. To achieve differential privacy, programs add noise to sensitive values. The Laplace mechanism described earlier is one basic mechanism for achieving differential privacy by adding noise drawn from the Laplace distribution (DD\textsubscript{DUO} provides a number of basic mechanisms, including the Gaussian mechanism). The following expression counts the number of rows in our example DataFrame and uses the Laplace mechanism to achieve \( \epsilon \)-differential privacy, for \( \epsilon = 1.0 \).

```python
dduo.laplace(df.shape[0], \epsilon=1.0)
```

The result is a regular Python value—the analyst is free to view it, write it to a file, or do further computation on it. Once the correct amount of noise has been added, the principle of post-processing applies, and so DD\textsubscript{DUO} no longer needs to track the sensitivity or privacy cost of operations on the value.

When the Laplace mechanism is used multiple times, their privacy costs compose (i.e. the \( \epsilon \) “add up” as described earlier). DD\textsubscript{DUO} tracks total privacy cost using objects called privacy odometers [41]. The analyst can interact with a privacy odometer object to learn the total privacy cost of a complex computation.

```python
with dduo.EpsOdometer() as odo:
    _ = dduo.laplace(df.shape[0], \epsilon=1.0)
    _ = dduo.laplace(df.shape[0], \epsilon=1.0)
    print(odo)
```

Printing the odometer’s value allows the analyst to view the privacy cost of the program with respect to each of the data sources used in the computation. In this example, two differentially private approximations of the number of rows in the dataframe \( df \) are computed, each with a privacy cost of \( \epsilon = 1.0 \). The total privacy cost of running the program is therefore \( 2 \times \epsilon = 2.0 \).

DD\textsubscript{DUO} also allows the analyst to place upper bounds on total privacy cost (i.e. a privacy budget) using privacy filters [41]. Privacy odometers and filters are discussed in detail in Section VI.

V. Dynamic Sensitivity Tracking

DD\textsubscript{DUO} implements a dynamic sensitivity analysis by wrapping values in Sensitive objects and calculating sensitivities as operations are performed on these objects. Type systems for sensitivity [40], [25] construct a sensitivity environment for each program expression; in the static analysis setting, a sensitivity environment records the expression’s sensitivity with respect to each of the variables currently in scope.

DD\textsubscript{DUO} attaches sensitivity environments to values at runtime: each Sensitive object holds both a value and its sensitivity environment. As described earlier, DD\textsubscript{DUO}’s sensitivity environments record a value’s sensitivity with respect to each of the program’s data sources. Formally, the sensitivity of a single-argument function \( f \) in its input is defined as:

\[
sens(f) \triangleq \arg\max_{x,y} \left( \frac{d(f(x), f(y))}{d(x, y)} \right)
\]

Where \( d \) is a distance metric over the values \( x \) and \( y \) could take (distance metrics are discussed in Section V-B). Thus, a sensitivity environment \( \{ a \mapsto f \} \) means that if the value of the program input \( a \) changes by \( n \), then the value of \( f(a) \) will change by at most \( n \).

A. Bounding the Sensitivity of Operations

Operations on Sensitive objects are defined to perform the same operation on the underlying values, and also construct a new sensitivity environment for the operation’s result. For example, DD\textsubscript{DUO}’s \( \_\_add\_\_ \) operation sums both the underlying values and their sensitivity environments:

```python
def \_\_add\_\_(self, other):
    assert self.metric == other.metric
    return dduoSensitive(self.value + other.value,
                        self.senv + other.senv,
                        self.metric)
```

The sum of two sensitivity environments is defined as the element-wise sum of their items. For example:

\[
\{ a \mapsto 2, b \mapsto 1 \} + \{ b \mapsto 3, c \mapsto 5 \} = \{ a \mapsto 2, b \mapsto 4, c \mapsto 5 \}
\]

The DD\textsubscript{DUO} library provides sensitivity-aware versions of Python’s basic numeric operations (formalized in Section VII). We have also defined sensitivity-aware versions of commonly-used library functions, including the Pandas functions used in Section IV, and subsets of NumPy and Scikit-learn.

B. Distance Metrics

At the core of the concept of sensitivity is the notion of distance: how far apart we consider two information sources to be from each other. For scalar values, the following two distance metrics are often used:

- Cartesian (absolute difference) metric: \( d(x, y) = |x - y| \)
- Discrete metric: \( d(x, y) = 0 \) if \( x = y \); 1 otherwise

For more complex structures—like lists and dataframes—we can use distance metrics on vectors. Two commonly-used metrics for vectors \( x \) and \( y \) of equal length are:

- \( L_1(d_i) \) metric: \( d(x, y) = \sum_{x, y \in x, y} d_i(x_i, y_i) \)
• $L_2(d_i)$ metric: $d(x, y) = \sqrt{\sum_{x_i, y_i \in x, y} d_i(x_i, y_i)^2}$

Both metrics are parameterized by $d_i$, a metric for the vector’s elements. In addition to these two, we use the shorthand $L_\infty$ to mean $L_1(d)$, where $d$ is the cartesian metric defined above. The $L_\infty$ metric works for any space with equality (e.g. strings), and measures the number of elements where $x$ and $y$ differ.

The definition of differential privacy is parameterized by a distance metric that is intended to capture the idea of two inputs that differ in one individual’s data. Database-oriented algorithms typically assume that each individual contributes exactly one row to the database, and use the $L_\infty$ metric to define neighboring databases (as we did in Section IV).

Distance metrics can be manipulated manually through operations such as clipping, a technique commonly employed in differentially private machine learning. DDUO tracks distance metrics for Sensitive information, which can allow for automatic conservation of the privacy budget while providing more accurate query analysis.

Lists and arrays are compared by one of the $L_1$, $L_2$, or $L_\infty$ distance metrics. The choice of distance metric is important when defining sensitivity and thus privacy. For example, the Laplace mechanism can only be used with the $L_1$ metric, while the Gaussian mechanism can be used with either $L_1$ or $L_2$.

C. Conditionals & Side Effects

Conditionals and other branching structures are challenging for any sensitivity analysis, but they present a particular challenge for our dynamic analysis. Consider the following conditional:

```python
if df.shape[0] == 10:
    return df.shape[0]
el:
    return df.shape[0] * 10000
```

Here, the two branches have different sensitivities (the else branch is 10,000 times more sensitive in its data sources than the then branch). Static sensitivity analyses handle this situation by taking the maximum of the two branches’ sensitivities (i.e. they assume the worst-case branch is executed), but this approach is not possible in our dynamic analysis.

In addition, special care must be taken when a sensitive value appears in the guard position (as in our example). Static analyses typically scale the branches’ sensitivity by the sensitivity of the guard; in practice, this approach results in infinite sensitivity for conditionals with a sensitive guard.

To retain soundness in our dynamic analysis, DDUO requires that conditional guards contain no sensitive values. A run-time error is thrown if DDUO finds a sensitive value in the guard position (as in our example above). Disallowing sensitive guards makes it possible to ignore branches that are not executed: the guard’s value remains the same under neighboring program inputs, so the program follows the same branch for neighboring executions. This approach does not limit the set of useful programs we can write, since conditionals with sensitive guards yield infinite sensitivities even under a precise static analysis.

Since DDUO attaches sensitivity environments to values (instead of variables), the use of side effects does not affect the soundness of the analysis. When a program variable is updated to reference a new value, that value’s sensitivity environment remains attached. DDUO handles many common side-effect-based patterns used in Python this way; for example, DDUO correctly infers that the following program results in the variable total holding a value that is 20 times more sensitive than df.shape[0].

```python
total = 0
for i in range(20):
    total = total + df.shape[0]
```

For side effects, our dynamic analysis is more capable than type-based static analysis, due to the additional challenges arising in the static setting (e.g. aliasing). We have formalized the way DDUO handles side effects and conditionals, and proved the soundness of our sensitivity analysis; our formalization appears in Section VII.

VI. DYNAMIC PRIVACY TRACKING

DDUO tracks privacy cost dynamically, at runtime. Dynamic privacy tracking is challenging because the dynamic analysis has no visibility into code that is not executed. For example, consider the following conditional:

```python
if dduo.gauss(ε=1.0, δ=1e-5, x) > 5:
    print(dduo.gauss(ε=1.0, δ=1e-5, y))
else:
    print(dduo.gauss(ε=100000000000.0, δ=1e-5, y))
```

The executed branch of this conditional depends on the result of the first call to dduo.gauss, which is non-deterministic. The two branches use different privacy parameters for the remaining calls to dduo.gauss; in other words, the privacy parameter for the second use of the Gaussian mechanism is chosen adaptively, based on the results of the first use. Sequential composition theorems for differential privacy [19] are typically stated in terms of fixed (i.e. non-adaptive) privacy parameters, and do not apply if the privacy parameters are chosen adaptively.

A static analysis of this program will consider both branches, and most analyses will produce an upper bound on the program’s privacy cost by combining the two (i.e. taking the maximum of the two $\epsilon$ values). This approach avoids the issue of adaptively-chosen privacy parameters.

A dynamic analysis, by contrast, cannot consider both branches, and must bound privacy cost by analyzing only the branch that is executed. Sequential composition does not apply directly when privacy parameters are chosen adaptively, so ignoring the non-executed branch in a dynamic analysis of privacy would be unsound.

A. Privacy Filters & Odometers

Privacy filters and odometers were originally developed by Rogers et al. [41] specifically to address the setting in which privacy parameters are selected adaptively. Winograd-Cort et al. [49] used privacy filters and odometers as part of the
Adaptive Fuzz framework, which integrates both dynamic analysis (for composing privacy mechanisms) and static analysis (for bounding the cost of individual mechanisms). Recently, Feldman and Zrnic [24] developed filters and odometers for Rényi differential privacy [36].

**Privacy odometers** can be used to obtain a running upper bound on total privacy cost at any point in the sequence of adaptive mechanisms, and to obtain an overall total at the end of the sequence. A function \( \text{COMP}_{\delta} : \mathbb{R}^{\geq 0} \rightarrow \mathbb{R} \cup \{\infty\} \) is called a valid privacy odometer [41] for a sequence of mechanisms \( \mathcal{M}_1, \ldots, \mathcal{M}_k \) if for all (adaptively-chosen) settings of \( (\epsilon_1, \delta_1), \ldots, (\epsilon_k, \delta_k) \) for the individual mechanisms in the sequence, their composition satisfies \( \text{COMP}_{\delta}(\cdot), \delta_g \)-differential privacy. In other words, \( \text{COMP}_{\delta}(\cdot) \) returns a value for \( \epsilon \) that upper-bounds the privacy cost of the adaptive sequence of mechanisms. A valid privacy odometer for sequential composition in \( (\epsilon, \delta) \)-differential privacy can be defined as follows (Rogers et al. [41], Theorem 3.6):

\[
\text{COMP}_{\delta}(\epsilon_1, \delta_1, \ldots, \epsilon_k, \delta_k) = \begin{cases} 
\infty & \text{if } \sum_{i=1}^{k} \delta_i > \delta_g \\
\sum_{i=1}^{k} \epsilon_i & \text{otherwise}
\end{cases}
\]

**Privacy filters** allow the analyst to place an upper bound \( (\epsilon_g, \delta_g) \) on the desired privacy cost, and halt the computation immediately if the bound is violated. A function \( \text{COMP}_{\epsilon, \delta} : \mathbb{R}^{\geq 0} \rightarrow \{\text{HALT, CONT}\} \) is called a valid privacy filter [41] for a sequence of mechanisms \( \mathcal{M}_1, \ldots, \mathcal{M}_k \) if for all (adaptively-chosen) settings of \( (\epsilon_1, \delta_1), \ldots, (\epsilon_k, \delta_k) \) for the individual mechanisms in the sequence, \( \text{COMP}_{\epsilon, \delta}(\epsilon_1, \delta_1, \ldots, \epsilon_k, \delta_k) \) outputs \text{CONT} only if the sequence satisfies \( (\epsilon_g, \delta_g) \)-differential privacy (otherwise, it outputs \text{HALT} for the first mechanism in the sequence that violates the privacy cost bound). A valid privacy filter for sequential composition in \( (\epsilon, \delta) \)-differential privacy can be defined as follows (Rogers et al. [41], Theorem 3.6):

\[
\text{COMP}_{\epsilon, \delta}(\epsilon_1, \delta_1, \ldots, \epsilon_k, \delta_k) = \\
\begin{cases} 
\text{HALT} & \text{if } \sum_{i=1}^{k} \delta_i > \delta_g \text{ or } \sum_{i=1}^{k} \epsilon_i > \epsilon_g \\
\text{CONT} & \text{otherwise}
\end{cases}
\]

It is clear from these definitions that the odometer and filter for sequential composition under \( (\epsilon, \delta) \)-differential privacy yield the same bounds on privacy loss as the standard theorem for sequential composition [19] (i.e. there is no “cost” to picking the privacy parameters adaptively).

Rogers et al. [41] also define filters and odometers for advanced composition under \( (\epsilon, \delta) \)-differential privacy (§5 and §6); in this case, there is a cost. In exchange for the ability to set privacy parameters adaptively, filters and odometers for advanced composition have slightly worse constants than the standard advanced composition theorem [19] (but are asymptotically the same).

### B. Filters & Odometers in DDuo

DDuo’s API allows the programmer to explicitly create privacy odometers and filters, and make them active for a specific part of the program (using Python’s `with` syntax). When an odometer is active, it records a running total of the total privacy cost, and it can be queried to return this information to the programmer.

```python
with dduo.EdOdometer(delta = 10e-6) as odo:
    _ = dduo.gauss(df.shape[0], epsilon = 1.0, delta = 10e-6)
print(odo)
```

When a filter is active, it tracks the privacy cost for individual mechanisms, and halts the program if the filter’s upper bound on privacy cost is violated.

```python
with dduo.EdFilter(epsilon = 1.0, delta = 10e-6) as odo:
    print('1:', dduo.gauss(df.shape[0], epsilon = 1.0, delta = 10e-6))
print('2:', dduo.gauss(df.shape[0], epsilon = 1.0, delta = 10e-6))
```

In addition to odometers and filters for sequential composition under \( (\epsilon, \delta) \)-differential privacy (such as EdFilter and EdOdometer), DDuo provides odometers and filters for advanced composition (AdvEdFilter and AdvEdOdometer) and Rényi differential privacy (RenyiFilter and RenyiOdometer, which follow the results of Feldman and Zrnic [24]).

### C. Loops and Composition

Iterative algorithms can be built in DDuo using Python’s standard looping constructs, and DDuo’s privacy odometers and filters take care of ensuring the correct form of composition. Parallel composition is also available—via functional mapping. Advanced composition can be achieved via special advanced composition filters and odometers exposed in the DDuo API. For example, the following simple loop runs the Laplace mechanism 20 times, and its total privacy cost is reflected by the odometer:

```python
with dduo.EpsOdometer() as odo:
    for i in range(20):
        dduo.laplace(df.shape[0], epsilon = 1.0)
print(odo)
```

To use advanced composition instead of sequential composition, we simply replace the odometer with a different one:

```python
with dduo.AdvEdOdometer() as odo:
    for i in range(20):
        dduo.gauss(df.shape[0], epsilon = 0.01, delta = 0.001)
```
D. Mixing Variants of Differential Privacy

The DDUO library includes support for pure $\epsilon$-differential privacy, $(\epsilon, \delta)$-differential privacy, and Rényi differential privacy (RDP). Programs may use all three variants together, convert between them, and compose mechanisms from each.

We demonstrate execution of a query while switched to the Rényi differential privacy variant using pythonic "with" syntax blocks. For programs that make extensive use of composition, this approach yields significant improvements in privacy cost. For example, the following program uses the Gaussian mechanism 200 times, using Rényi differential privacy for sequential composition; the total privacy cost is automatically converted into an $(\epsilon, \delta)$-differential privacy cost after the loop finishes.

```python
with dduo.RenyiDP(1e-5):
    for x in range(200):
        noisy_count = dduo.renyi_gauss(\alpha - 10, \epsilon=0.2, df.shape[0])
duo.print_privacy_cost()
```

VII. Formal Description of Sensitivity Analysis

In DDUO we implement a novel dynamic analysis for function sensitivity, which is a relational (hyper)property quantified over two runs of the program with arbitrary but related inputs. In particular, our analysis computes function sensitivity—a two-run property—after only observing one execution of the program. Only observing one execution poses challenges to the design of the analysis, and significant challenges to the proof, all of which we overcome. To overcome this challenge in the design of the analysis, we first disallow branching control flow which depends on any sensitive inputs; this ensures that any two runs of the program being considered for the purposes of privacy will take the same branch observed by the dynamic analysis. Second, we disallow sensitive input-dependent arguments to the "scalar" side of multiplication; this ensures that the dynamic analysis’ use of that argument in analysis results is identical for any two runs of the program being considered for the purposes of privacy. Our dynamic analysis for function sensitivity is sound—meaning that the true sensitivity of a program is guaranteed to be equal or less than the sensitivity reported by DDUO’s dynamic monitor—and we support this claim with a detailed proof.

Formal Approach. We formalize the correctness of our dynamic analysis for function sensitivity using a step-indexed big-step semantics to describe the dynamic analysis, a step-indexed logical relation to describe the meaning of function sensitivity, and a proof by induction and case analysis on program syntax to show that dynamic analysis results soundly predict function sensitivity. A step-indexed relation is a relation $R \in A \rightarrow B \rightarrow \text{prop}$ whose definition is stratified by a natural number index $n$, so for each level $n$ there is a new relation $R_n$. Typically, the relation $R_0$ is defined $R_0(x, y) \triangleq \text{true}$, and the final relation of interest is $R \triangleq \bigcap_n R_n$, i.e., $R(x, y) \iff \forall n. R_n(x, y)$. Step-indexing is typically used—as we do in our formalism—when the definition of a relation would be not well founded in its absence. The most common reason a relation definition might be not well-founded is the use of self-reference without any decreasing measure. When a decreasing measure exists, self-reference leads to well-founded recursion, however when a decreasing measure does not exist, self-reference is not well-founded. When using step-indexing, self-reference is allowed in the definition of $R_n$, but only for the relation at strictly lower levels, so $R_n'$ when $n' < n$; this is well-founded because the index $n$ becomes a decreasing measure for the self-reference. In this way, step-indexing enables self-reference without any existing decreasing measure by introducing a new decreasing measure, and maintains well-foundedness of the relation definition.

A logical relation is one where the definition of relation on function values (or types) is extensional, essentially saying "when given related inputs, the function produces related outputs". This definition is self-referential and not well-founded, and among common reasons to introduce step-indexing in programming language proofs. As the relation $R$ is stratified with a step-index to $R_n$, so must the definition of the semantics, so for a big-step relation $e \triangleright v$ (relating an expression $e$ to its final value $v$ after evaluation) we stratify as $e \triangleright_{\sigma} v$. Also, because the definition of a logical relation decrements the step-index for the case of function values, we increment the step-index in the semantic case for function application. These techniques are standard from prior work [3], and we merely summarize the key ideas here to give background to our reader.

Formal Definition of Dynamic Analysis. We model language features for arithmetic operations ($e \odot e$), conditionals ($\text{if} \{e\}\{e\}\{e\}$), pairs ($\langle e, e \rangle$ and $\pi_i(e)$), functions ($\lambda x. e$ and $e(e)$) and references ($e \mathrel{\inref} e$ and $e \leftrightarrow e$); the full language is shown in Figure 1. There is one base value: $r \in \mathbb{R}$ for a real number result $r$ tagged with dynamic analysis information $\Sigma$—the sensitivity analysis for the expression which evaluated to $r$—and $m$—the metric associated with the resulting value $r$. The sensitivity analysis $\Sigma$—also called a sensitivity environment—is a map from sensitive sources $o \in \text{source}$ to how sensitive the result is w.r.t. that source. Our formalism includes two base metrics $m \in \text{metric}$: $\text{diff}$ and $\text{disc}$ for absolute difference $|x - y|$ and discrete distance (0 if $x = y$ and 1 otherwise) respectively—and two derived metrics: $\cap$ and $\cup$ for the smallest metric larger than each base metric and largest metric smaller than each base metric, respectively. Each metric is commonly used when implementing differentially private algorithms. Pair values $\langle v, v \rangle$, closure values $\langle \lambda x. e \rangle$, and reference values ($v$) do not contain dynamic analysis information.

Our dynamic analysis is described formally as a big-step relation $\rho \vdash_\Sigma \sigma, e, \uplus_n \sigma, v$ where $\rho \in \text{var} \rightarrow \text{value}$ is the lexical environment mapping lexical variables to values, $\sigma \in \text{loc} \rightarrow \text{value}$ is the dynamic environment (i.e., the heap, or
store) mapping dynamically allocated references to values, \( e \) is the expression being executed, and \( v \) is the resulting runtime value which also includes dynamic analysis information. We write gray box corners around the “input” configuration \( \sigma, e \) and the “output” configuration \( \sigma, v \) to aid readability. The index \( n \) is for step-indexing, and tracks the number of function applications which occurred in the process of evaluation. We show the full definition of the dynamic analysis in Figure 2.

Consider the following example:

\[
\{ x \rightarrow 21 @ \{ \text{diff} \} \} \vdash \forall, (x + x) \Downarrow_0 42 @ \{ \text{diff} \} \]

This relation corresponds to a scenario where the program to evaluate and analyze is \( x + x \), the variable \( x \) represents a sensitive source value \( o \), we want to track sensitivity w.r.t. the absolute difference metric, and the initial value for \( x \) is \( 21 \). This information is encoded in an initial environment \( \rho = \{ x \rightarrow 21 @ \{ \text{diff} \} \} \). The result value is \( 42 \), and the analysis reports that \( e \) is \( 2 \)-sensitive in the source \( o \) w.r.t. the absolute difference metric. This analysis information is encoded in the return value \( 42 @ \{ \text{diff} \} \). Because no function applications occur during evaluation, the step index \( n \) is 0.

**Formal Definition of Function Sensitivity.** Function sensitivity is encoded through multiple relation definitions:

1. \( \rho_1, \sigma_1, e_1 \sim_n \rho_2, \sigma_2, e_2 \) holds when the input triples \( \rho_1, \sigma_1, e_1 \) and \( \rho_2, \sigma_2, e_2 \) evaluate to output stores and values which are related by \( \Sigma \). Note this definition decrements the step-index \( n \), and is the constant relation when \( n = 0 \).
2. \( r_1 \sim_m r_2 \) holds when the difference between real numbers \( r_1 \) and \( r_2 \) w.r.t. the metric \( m \) is less than \( r \).
3. \( v_1 \sim_{\Sigma} v_2 \) holds when values \( v_1 \) and \( v_2 \) are related for initial distance \( \Sigma \) and step-index \( n \). The definition is by case analysis on the syntactic category for values, such as:

a) The relation on base values \( r_1 \#_{m} \sim_n r_2 \#_{m} \#_{\Sigma} \) holds when \( \Sigma_1, \Sigma_2, m_1 \) and \( m_2 \) are pairwise equal, and when \( r_1 \) and \( r_2 \) are related by \( \Sigma : \Sigma_1 \), where \( \Sigma \) is the initial distances between each input source \( o \), and \( \Sigma_1 \) is the step-index \( n \). These functions are applied via vector dot product.

b) The relation on pair values \( \langle v_{11}, v_{12} \rangle \sim_{\Sigma} \langle v_{21}, v_{22} \rangle \) holds when each element of the pair are pairwise related.

c) The relation on function values \( \langle \lambda x. e_1 | \rho_1 \rangle \sim_{\Sigma} \langle \lambda x. e_2 | \rho_2 \rangle \) holds when each closure returns related output configurations when evaluated with related inputs.

d) The relation on locations \( \ell_1 \sim_{\Sigma} \ell_2 \) holds when the two locations are equal.

4. \( \rho_1 \sim_n \rho_2 \) holds when lexical environments \( \rho_1 \) and \( \rho_2 \) map all variables to related values.
5. \( \sigma_1 \sim_n \sigma_2 \) holds when dynamic environments \( \sigma_1 \) and \( \sigma_2 \) map all locations to related values.

Note that the definitions of \( \rho_1, \sigma_1, e_1 \sim_n \rho_2, \sigma_2, e_2 \) and \( v \sim_n v \) are mutually recursive, but are well founded due to the decrement of the step index in the former relation. We show the full definition of these relations in Figure 1.

The function sensitivity of an expression is encoded first as a statement about expressions respecting relatedness, that is, returning related outputs when given related inputs, i.e., assuming no use of the store if \( \rho_1 \sim_n \rho_2 \) and \( \rho_1 \vdash \forall, e \Downarrow_{n_1} \forall, e_1 \) and \( \rho_2 \vdash \forall, e \Downarrow_{n_2} \forall, e_2 \) then \( n_1 = n_2 \) and \( v_1 \sim_{\Sigma_{n_1}} v_2 \).

When instantiated to base types, we have: if \( \rho_1 \sim_n \rho_2 \) and \( \rho_1 \vdash \forall, e \Downarrow_{n_1} \forall, e_1 \) then \( n_1 = n_2 \), \( \Sigma_1 = \Sigma_2 \), \( m_1 = m_2 \) and \( r_1 \sim_{\Sigma_{m_1}} r_2 \). The fully general form of this property is called metric preservation, which is the main property we prove in our formal development.

**Metric Preservation.** Metric preservation states that when given related initial configurations and evaluation outputs, then those outputs are related. Outputs include result values, as well as dynamic analysis results, and the relationship that holds demonstrates the soundness of the analysis results.

**Theorem VII.1 (Metric Preservation).**

If: \( \rho_1 \sim_n \rho_2 \)
And: \( \sigma_1 \sim_n \sigma_2 \)
Then: \( \rho_1, \sigma_1, e \sim_n \rho_2, \sigma_2, e \)

That is, either \( n = 0 \) or \( n = n' + 1 \) and...

If: \( n_1 \leq n' \)
And: \( \rho_1 \vdash \forall, e \Downarrow_{n_1} \forall, \sigma_1, e_1 \)
And: \( \rho_2 \vdash \forall, e \Downarrow_{n_2} \forall, \sigma_2, e_2 \)

Then: \( n_1 = n_2 \)
And: \( \sigma_1 \sim_{\Sigma_{n_1}} \sigma_2 \)
And: \( v_1 \sim_{\Sigma_{n_1}} v_2 \)

Proof. See detailed proof in the appendix. □

**Instantiating Metric Preservation.** Metric preservation is not enough on its own to demonstrate sound dynamic analysis of function sensitivity. Suppose we execute the dynamic analysis on program \( e \) with initial environment \( \rho \) yielding a final store \( \sigma \), base value \( r \), sensitivity environment \( \Sigma \), metric \( m \) and step-index \( n \) as a result:

\( \rho \vdash \forall, e \Downarrow_{\Sigma} \forall, \sigma, r \Downarrow_{\Sigma} \)

To know the sensitivity of \( e \) is to know a bound on two arbitrary runs of \( e \), that is, using two arbitrary environments \( \rho_1 \) and \( \rho_2 \). Does \( \Sigma \) tell us this? Remarkably, it does, with one small condition: \( \rho_1 \) and \( \rho_2 \) must agree with \( \rho \) on all non-sensitive values. This is not actually limiting: a non-sensitive value is essentially auxiliary information; they are constants and fixed for the purposes of sensitivity and privacy.

We can encode the relationship that environments \( \rho \) and \( \rho_1 \) agree on all non-sensitive values as \( \rho \sim_{\Sigma'} \rho_1 \) for any \( \Sigma' \), and we allow for environments \( \rho \) and \( \rho_1 \) to differ on
\[ b \in \mathbb{B} \quad n \in \mathbb{N} \quad i \in \mathbb{Z} \quad r \in \mathbb{R} \quad x \in \text{var} \]

- \( o \in \text{source} \): sensitive sources
- \( \ell \in \text{loc} \): reference locations
- \( e \in \text{expr} \): variables
- \( \sigma \in \text{env} \): value environment
- \( \rho \in \text{store} \): mutable store
- \( \Sigma \in \text{envv} \): sens. environment
- \( q \in \hat{\mathbb{R}} \): ext. reals
- \( \odot \in \text{binop} \): operations
- \( m \in \text{metric} \): absolute difference
- \( \text{disc} \): discrete
- \( \top \): bot metric
- \( \text{top metric} \)
- \( v \in \text{value} \): tagged base value
- \( \rho \in \text{env} \): value environment
- \( \sigma \in \text{store} \): value (pointer)
- \( \Sigma \in \text{envv} \): sens. environment

### Fig. 1. Formal Syntax & Step-indexed Logical Relation.

\[
\begin{align*}
\rho_1, \sigma_1, e_1 &\sim^0_n \rho_2, \sigma_2, e_2 \iff \text{true} \\
\rho_1, \sigma_1, e_1 &\sim^\Sigma_{n+1} \rho_2, \sigma_2, e_2 \iff \forall n_1 \leq n, n_2, \sigma'_1, \sigma'_2, v_1, v_2.
\end{align*}
\]

any sensitive value while agreeing on non-sensitive values as \( \rho \sim^{(0, \infty)} \rho_1 \). Under such an assumption, \( \Sigma \) and \( m \) are sound dynamic analysis results for two arbitrary runs of \( e \), i.e., under environments \( \rho_1 \) and \( \rho_2 \), so long as one of those environments agrees with \( \rho \)—the environment used to compute the dynamic analysis. We encode this property formally as the following corollary to metric preservation:

#### Corollary VII.1.1 (Sound Dynamic Analysis for Sensitivity).

**If:** \( n_1 < n, n_2 < n \) and \( n_3 < n \)

**And:** \( \rho \sim^{(0, \infty)} \rho_1 \)

**And:** \( \rho \vdash e, e \downarrow \rho_1, r @^\Sigma_m \rho_2 \)

**And:** \( \rho_1 \sim^\Sigma_n \rho_2 \)

**And:** \( \rho_1 \vdash e, e \downarrow \rho_1, r @^\Sigma_{m_1j} \rho_2 \)

**And:** \( \rho_2 \vdash e, e \downarrow \rho_3, r @^\Sigma_{m_2j} \rho_2 \)

**Then:** \( r_1 \sim^\Sigma_m r_2 \)

**Proof.**

By Metric Preservation, \((H2), (H1), (H3) \) and \((H5) \) we have \( \Sigma_1 = \Sigma \) and \( m_1 = m \). By Metric Preservation, \((H4), (H1), (H5) \) and \((H6) \) we have proved the goal \((C1) \). \( \square \)
Note that the final results are related using $\Sigma$—the analysis result derived from an execution under $\rho$—while $r_1$ and $r_2$ are derived from executions under unrelated (modulo auxiliary information) environments $\rho_1$ and $\rho_2$.

In simpler terms, this corollary shows that even though the dynamic analysis only sees one particular execution of the program, it is accurate in describing the sensitivity of the program—even though the notion of sensitivity considers two arbitrary runs of the program, including those whose inputs differ entirely from those used in the dynamic analysis.

VIII. IMPLEMENTATION & CASE STUDIES

We have developed a reference implementation of DDUO as a Python library, using the approaches described in Sections IV, V, and VI.

A major goal in the design of DDUO is seamless integration with other libraries. Our reference implementation provides initial support for NumPy, Pandas, and Sklearn. DDUO provides hooks for tracking both sensitivity and privacy, to simplify integrating with additional libraries.

We present the case studies which focus on demonstrating DDUO’s (1) similarity to regular Python code, (2) applicability
to complex algorithms, (3) easy integration with existing libraries. In this section we focus on the Noisy Gradient Descent case study, other case studies have been moved to the appendix due to space requirements. We introduce new adaptive variants of algorithms that stop early when possible to conserve privacy budget. These variants cannot be verified by prior work using purely static analyses, because their privacy parameters are chosen adaptively.

**Run-time overhead.** Run-time overhead is a key concern in DDUO’s instrumentation for dynamic analysis. Fortunately, experiments on our case studies suggest that the overhead of DDUO’s analysis is generally low. Table I presents the run-time performance overhead of DDUO’s analysis as a percentage increase of total runtime. The worst overhead time observed in our case studies was less than 60%.

In certain rare cases, DDUO’s overhead can be much higher. For example, mapping the function \( \lambda x: x + 1 \) over a list of 1 million numbers takes 160x longer under DDUO than in standard Python. The overhead in this case comes from a combination of factors: first, DDUO’s map function, itself implemented in Python, is much slower than Python’s built-in map operator; second, DDUO’s map function requires the creation of a new Sensitive object for each element of the list—a slow operation in Python.

Fortunately, the same strategies for producing high-performance Python code without privacy also help reduce DDUO’s overhead. Python’s performance characteristics have prompted the development of higher-performance libraries like NumPy and Pandas, which essentially provide data-processing combinators that programmers compose. By providing sensitivity annotations for these libraries, we can re-use these high-performance implementations and avoid creating extra objects. As a result, none of our case studies demonstrates the worst-case performance overhead described above.

**Case study: gradient descent with NumPy.** Our first case study (Figure 4) is a simple machine learning algorithm based on [12] implemented directly with DDUO-instrumented NumPy primitives. The remaining case studies appear in the Appendix.

Given a dataset \( X \) which is a list of feature vectors representing training examples, and a vector \( y \) which classifies each element of \( X \) in a finite set, gradient descent is the process of computing a model (a linear set of weights) which most accurately classifies a new, never seen before training example, based on our pre-existing evidence represented by the model.

Gradient descent works by first specifying a loss function that computes the effectiveness of a model in classifying a given dataset according to its known labels. The algorithm then iteratively computes a model that minimizes the loss function, by calculating the gradient of the loss function and moving the model in the opposite direction of the gradient.

One method of ensuring privacy is in gradient descent involves adding noise to the gradient calculation, which is the only part of the process that is exposed to the private training data. In order to add noise to the gradient, it is convenient to bound its sensitivity via clipping to some L2 norm. In this example, clipping occurs in the gradient\_sum function before summation.

The original implementation of this algorithm [12] was

| Name     | Type                                      | Conditions                                                                 |
|----------|-------------------------------------------|-----------------------------------------------------------------------------|
| laplace  | \( (\epsilon : \mathbb{R}, \text{value} : \mathbb{R}) \rightarrow \mathbb{R} \) | where \( \text{priv(laplace}(\epsilon, \text{value})) \triangleq \epsilon \) |
| gauss    | \( (\epsilon : \mathbb{R}, \delta : \mathbb{R}, \text{value} : \mathbb{R}) \rightarrow \mathbb{R} \) | where \( \text{priv(gauss}(\epsilon, \delta, \text{value})) \triangleq (\epsilon, \delta) \) |
| ed\_odo  | \( (f : A \rightarrow B, \text{in} : A) \rightarrow (\text{out} : B, (\epsilon, \delta) : (\mathbb{R}, \mathbb{R})) \) | where \( \text{priv(ed\_odo}(f, \text{in})) \triangleq \text{priv}(f(\text{in})) \) |
| renyi\_odo| \( (f : A \rightarrow B, \text{in} : A) \rightarrow (\text{out} : B, (\epsilon, \delta) : (\mathbb{R}, \mathbb{R})) \) | where \( \text{priv(renyi\_odo}(f, \text{in})) \triangleq \text{priv}(f(\text{in})) \) |
| ed\_filter | \( (f : A \rightarrow B, \text{in} : A, (\epsilon, \delta) : (\mathbb{R}, \mathbb{R})) \rightarrow (\text{out} : B) \) | where \( \epsilon, \delta \geq \text{priv}(f(\text{in})) \) |
| renyi\_filter | \( (f : A \rightarrow B, \text{in} : A, (\epsilon, \delta) : (\mathbb{R}, \mathbb{R})) \rightarrow (\text{out} : B) \) | where \( \epsilon, \delta \geq \text{priv}(f(\text{in})) \) |
| conv\_renyi | \( (f : A \rightarrow B, \text{in} : A, (\epsilon, \delta) : (\mathbb{R}, \mathbb{R})) \rightarrow (\text{out} : B) \) | where \( \epsilon, \delta \geq \text{priv}(f(\text{in})) \) and \( \text{con}(\epsilon, \delta) = (\epsilon, \delta) \) |

\( \text{priv} \): denotes the privacy leakage of a program given by dynamic analysis; \( \text{sens} \): denotes the sensitivity of a program given by dynamic analysis; \( \text{conv} \): represents the conversion equation from renyi to approximate differential privacy.

Types are written as follows: the \( \rightarrow \) symbol is used to separate the domain and range of a function, either of which may be given as an atomic type such as a natural number (\( \mathbb{N} \)), or as a tuple which is a comma-separated list of types surrounded by parentheses, or as a symbol \( \lambda \) indicating parametric polymorphism (generics). In some cases, types may also be accompanied with a placeholder name \( (\epsilon : \mathbb{R}) \) for further qualification in the where clause.

Fig. 3. Core API Methods

| Technique                          | Ref. | Libraries Used | Overhead |
|------------------------------------|------|----------------|---------|
| Noisy Gradient Descent             | [12] | NumPy          | 6.42%   |
| Multiplicative Weights (MWEM)      | [29] | Pandas         | 14.90%  |
| Private Naive Bayes Classification | [44] | DiffPrivLib    | 12.44%  |
| Private Logistic Regression        | [16] | DiffPrivLib    | 56.33%  |

**TABLE I**

List of case studies included with the DDUO implementation.
based on the advanced composition theorem. Advanced composition improves on sequential composition by providing much tighter privacy bounds over several iterations, but requires the analyst to fix the number of iterations up front, regardless of how many iterations the gradient descent algorithm actually takes to converge to minimal error.

We present a modified version based on adaptive Renyi differential privacy which provides not only a tighter analysis of the privacy leakage over several iterations, but also allows the analyst to halt computation adaptively (conserving the remaining privacy budget) once a certain level of model accuracy has been reached, or loss has been minimized. We obscure the accuracy calculation because it is a computation on the sensitive input training dataset in this case.

IX. RELATED WORK

Dynamic Enforcement of Differential Privacy. The first approach for dynamic enforcement of differential privacy was PINQ [35]. Since then several works have been based on PINQ, such as Featherweight PINQ [20] which models PINQ formally and proves that any programs which use its simplified PINQ API are differentially private. ProPer [21] is a system (based on PINQ) designed to maintain a privacy budget for each individual in a database system, and operates by silently dropping records from queries when their privacy budget is exceeded. UniTrax [37] follows up on ProPer: this system allows per-user budgets but gets around the issue of silently dropping records by tracking queries against an abstract database as opposed to the actual database records. These approaches are limited to an embedded DSL for expressing relational database queries, and do not support general purpose programming.

A number of programming frameworks for differential privacy have been developed as libraries for existing programming languages. DP-ELLA [34] is a Haskell library that provides static bounds on the accuracy of differentially private programs. Diffprivlib [31] (for Python) and Google’s library [48] (for several languages) provide differentially private algorithms, but do not track sensitivity or privacy as these algorithms are composed. ekteIo [50] executes programmer-specified plans that encode differentially private algorithms using framework-supplied building blocks.

Dynamic Information Flow Control. Our approach to dynamic enforcement of differential privacy can be seen as similar to work on dynamic information flow control (IFC) and taint analysis [6]. The sensitivities that we attach to values are comparable to IFC labels. However, dynamic IFC typically allows the programmer to branch on sensitive information and handles implicit flows dynamically. DDUO prevents branching on sensitive information, similar to an approach taken for preventing side-channels in the “constant time” programming discipline for cryptographic code [7]. Another connection with this line of work is that our use of the logical relations proof technique for a differential privacy theorem is similar to the usage of this technique for noninterference theorems [42], [30], [1], [14], [5], [26].

Static Verification of Differential Privacy. The first approach for static verification of differential privacy was FUUZ [40], which used linear types to analyze sensitivity. DFUZZ [25] adds dependent types, and later work [17], [39] extends the approach to \((\epsilon, \delta)-\)differential privacy. FUZZI [52] integrates a FUZZ-like type system with an expressive program logic. Adaptive FUZZ [49] combines a FUZZ-style static type system for sensitivity analysis with a dynamic privacy analysis using privacy filters and odometers; Adaptive FUZZ is most similar to our approach, but uses a static sensitivity analysis. All of these approaches require additional type annotations.

A second category is based on approximate couplings [8]. The aprhRL [10], [11], aprhRL+ [9], and SPAN-aprhl [43] relational logics are extremely expressive, but less amenable to automation. Albarghouthi and Hsu [4] use an alternative approach based on constraint solving to synthesize approximate couplings. A third approach is based on randomness alignments; LightDP [51] and ShadowDP [46] take this approach. Randomness alignments are effective for verifying low-level mechanisms like the sparse vector technique. The latter two categories are generally restricted to first order imperative programs.

Dynamic Testing for Differential Privacy. A recent line of work [13], [18], [45], [48] has resulted in approaches for testing differentially private programs. These approaches generate a series of neighboring inputs, run the program many times on the neighboring inputs, and raise an alarm if a counterexample is found. These approaches do not require
type annotations, but do require running the program many times.

X. Conclusion

We have presented DDUO, a dynamic analysis that supports general-purpose differentially private programming with an emphasis on machine learning. We have formalized the sensitivity analysis of DDUO and proven its soundness using a step-indexed logical relation. Our case studies demonstrate the utility of DDUO by enforcing adaptive variants of several differentially private state-of-the-art machine learning algorithms from the ground up, while integrating with some of Python’s most popular libraries for data analysis.
Proof. By mutual induction on $n$ for all three properties, and additionally case analysis on $e_1$ and $e_2$ for property (3).

**Theorem A.6 (Metric Preservation).**

If $\rho_1 \sim^\Sigma_{\Sigma_i} \rho_2$ (H1)\n
And $\sigma_1 \sim^\Sigma_{n} \sigma_2$ (H2)\n
Then $\rho_1, \sigma_1, e \sim^\Sigma_{n} \rho_2, \sigma_2, e$.

That is, either $n = 0$ or $n = n' + 1$ and...

If $n_1 \leq n'$ (H3)\n
And $\rho_1 \vdash_1 \sigma_1, e \downarrow n_1, \sigma_1', v_1$ (H4)\n
And $\rho_2 \vdash_2 \sigma_2, e \downarrow n_2, \sigma_2', v_2$ (H5)\n
Then $n_1 = n_2$ (C1)\n
And $\sigma_1' \sim^{\Sigma_{\Sigma_i}^{-1}}_{n_1} \sigma_2'$ (C2)\n
And $v_1 \sim^{\Sigma_{n_1}}_{n_2} v_2$ (C3)

**Proof.** See detailed proof later in this section.

**Corollary A.6.1 (Sound Dynamic Analysis for Sensitivity).**

If $n_1 < n_2 < n$ and $n_3 < n$ (H1)\n
And $\rho \sim^\rho_{[0,\infty)} \rho_1$ (H2)\n
And $\rho \vdash_1 \sigma, e \downarrow n_1, \sigma, r \circ_{n_3} \rho_2$ (H3)\n
And $\rho_1 \sim^{\Sigma'}_{n} \rho_2$ (H4)\n
And $\rho_1 \vdash_1 \sigma, e \downarrow n_2, \sigma, r \circ_{n_1} \rho_1$ (H5)\n
And $\rho_2 \vdash_2 \sigma, e \downarrow n_3, \sigma_2, r_2 \circ_{n_2} \rho_2$ (H6)\n
Then $r_1 \sim^{\Sigma_{\Sigma_i}}_{m_1} r_2$ (C1)

**Proof.**

By Metric Preservation, (H2), (H1), (H3) and (H5) we have $\Sigma_i = \Sigma$ and $m_1 = m$.

By Metric Preservation, (H4), (H1), (H5) and (H6) we have proved the goal (C1).

**Proof of Metric Preservation.**

By strong induction on $n$ and case analysis on $e$:

- **Case $n = 0$:** Trivial by definition.
- **Case $n = n + 1$ and $e = x$:**

By inversion on (H4) and (H5) we have: $n_1 = n_2 = 0$, $\sigma'_1 = \sigma_1$, $\sigma'_2 = \sigma_2$, $v_1 = \rho_1(x)$ and $v_2 = \rho_2(x)$. To show: (C1): $0 = 0$; (C2): $\sigma_1 \sim^{\Sigma}_{n} \sigma_2$; and (C3): $\rho_1(x) \sim^{\rho}_{n} \rho_2(x)$. (C1) is trivial. (C2) is immediate by (H2). (C3) is immediate by (H1).

- **Case $n = n + 1$ and $e = r$:**

By inversion on (H4) and (H5) we have $n_1 = n_2 = 0$, $v_1 = v_2 = r \circ_{\text{disc}} \sigma'_1 = \sigma_1$ and $\sigma'_2 = \sigma_2$. To show: (C1): $0 = 0$; (C2): $\sigma_1 \sim^{\Sigma}_{n} \sigma_2$; and (C3): $r \circ^{\rho}_{\text{disc}} r$. (C1) is trivial. (C2) is immediate by (H2). (C3) is immediate by definition of relation $\sim^{\rho}$.
- Case $n = n + 1$ and $e = e_1 + e_2$:

By inversion on (H4) and (H5) we have:

\[
\rho_1 \vdash \sigma_1, e_1 \downarrow_{n_1} \sigma'_1, r_1 @ \Sigma_{m_1} (H4.1)
\]

\[
\rho_1 \vdash \sigma'_1, e_2 \downarrow_{n_2} \sigma''_1, r_2 @ \Sigma_{m_2} (H4.2)
\]

\[
\rho_2 \vdash \sigma_2, e_1 \downarrow_{n_1} \sigma'_2, r_1 @ \Sigma_{m_1} (H5.1)
\]

\[
\rho_2 \vdash \sigma'_2, e_2 \downarrow_{n_2} \sigma''_2, r_2 @ \Sigma_{m_2} (H5.2)
\]

and we also have:

\[
\begin{align*}
\sigma'_1 &= \rho_1(n_1) + r_1 \Delta_{m_1} \\
\sigma''_1 &= \rho_2(n_2) + r_2 \\
\sigma'_2 &= \rho_2(n_1) + r_1 \\
\sigma''_2 &= \rho_1(n_2) + r_2
\end{align*}
\]

By IH $(n = n - n_1)$ decreasing, (F2), (H1), (H2), (H3), (H4.1) and (H5.1) we have:

\[
\begin{align*}
\Sigma_{m_1} &= n_1 = n_2 (IH.1.C1) \\
\Sigma_{m_1} &= n_1 - n_1 = n_2 (IH.1.C2)
\end{align*}
\]

By unfolding the definition in (IH.1.C3), we have:

\[
\begin{align*}
\Sigma_{m_1} &= \Sigma_{m_1} (IH.1.C3.1) \\
\Sigma_{m_1} &= m_{21} (IH.1.C3.2)
\end{align*}
\]

and $r_1 \sim r_2$. (IH.2.C1) and (IH.2.C2).

Case $n = n + 1$ and $e = e_1 + e_2$: Analogous to previous case.

- Case $n = n + 1$ and $e = 0$:

By inversion on (H4) and (H5) we have 4 subcases, each which induce:

\[
\begin{align*}
\rho_1 &\vdash \sigma_1, e_1 \downarrow_{n_1} \sigma'_1, r_1 @ \Sigma_{m_1} (H4.1) \\
\rho_2 &\vdash \sigma_2, e_2 \downarrow_{n_2} \sigma''_2, r_2 @ \Sigma_{m_2} (H5.1)
\end{align*}
\]

By IH $(n = n - n_1)$ decreasing, (H1), (H2), (H3), (H4.1) and (H5.1) we have:

\[
\begin{align*}
\Sigma_{m_1} &= n_1 = n_2 (IH.1.C1) \\
\Sigma_{m_1} &= n_1 - n_1 = n_2 (IH.1.C2)
\end{align*}
\]

and $r_1 \sim r_2$. (IH.2.C1) and (IH.2.C2).

As a consequence of (IH.1.C3), we have: $r_1 = r_2$. Note the following facts: $n_1 \leq n - n_1 (F1)$ and $r_1 \sim m_{n_1} (F2)$. (F1) follows from (H3) and $n_1 = n_1 + n_2$. (F2) follows from (H1) and Step-index Weakening.1. By IH $(n = n - n_1)$ decreasing, (F2), (H1), (H2), (H3), (H4.1) and (H5.1) we have:

\[
\begin{align*}
\Sigma_{m_1} &= n_1 = n_2 (IH.1.C1) \\
\Sigma_{m_1} &= m_{21} (IH.1.C3.2)
\end{align*}
\]

and $r_1 \sim r_2$. (IH.2.C1) and (IH.2.C2). (C2) is immediate from (IH.2.C2). (C2) holds as follows: By Times Respects, $r_1 = r_2$ and (IH.2.C3): $r_1 \sim r_2$. (C1) is immediate from (IH.1.C1) and (IH.2.C1). (C2) is immediate from (IH.2.C2).

To show (C3): $n_1 + n_1 = n_1 + n_2$ and (C3):

\[
\begin{align*}
\Sigma_{m_1} &= \Sigma_{m_1} (IH.1.C3.1) \\
\Sigma_{m_1} &= \Sigma_{m_1} (IH.1.C3.2)
\end{align*}
\]

and $r_1 \sim r_2$. (IH.2.C3) and (IH.2.C2). (C3) is immediate from (IH.2.C2). (C3) holds as follows: By Times Respects, $r_1 = r_2$ and (IH.2.C3):

\[
\begin{align*}
\Sigma_{m_1} &= \Sigma_{m_1} (IH.1.C1) \\
\Sigma_{m_1} &= \Sigma_{m_1} (IH.2.C1)
\end{align*}
\]

We have shown the goal.
and we also have: $n_1 = n_{11} + n_{12}$, $n_2 = n_{21} + n_{22}$, $\sigma_1' = \sigma_1''$, $\sigma_2' = \sigma_2''$, $v_1 = v_1$, $v_2 = v_2$. We continue reasoning in a generic way outside this subcase.

Note the following facts: $n_{12} \leq n - n_{11}$ (F1); and $n_1 \sim^{\Sigma}_{n,n_{11}} (F2)$, (F1) follows from (H3) and $n_1 = n_{11} + n_{12}$ (F2) follows from (H1) and Step-index Weakening.1. By IH ($n = n - n_{11}$ decreasing), (F2), (IH.1.C2), (F1), (H4.2) and (H5.2) we have: $n_{21} = n_{22}$ (IH.2.C1); $\sigma_1' \sim^{\Sigma}_{n-n_{11} - n_{22}} \sigma_2'$ (IH.2.C2); and $v_1 \sim^{\Sigma}_{n-n_{11} - n_{22}} v_2$ (IH.2.C3). To show: (C1): $n_{11} + n_{12} = n_{21} + n_{22}$; (C2): $\sigma_1' \sim^{\Sigma}_{n-n_{11} - n_{22}} \sigma_2''$; and (C3): $v_1 \sim^{\Sigma}_{n-n_{11} - n_{22}} v_2$. (C1) is immediate from (IH.1.C1) and (IH.2.C2). (C3) is immediate from (IH.2.C2). (C2) is immediate from (IH.2.C3).

- **Case** $n = n + 1$ and $e = \langle e_1, e_2 \rangle$:

By inversion on (H4) and (H5) we have:

- $\rho_1 \vdash \sigma_1, e_1 \downarrow v_{n_{11}} \sigma_1', v_1_{11}$ (H4.1)
- $\rho_1 \vdash \sigma_1', e_2 \downarrow v_{n_{12}} \sigma_1''$, $v_{12}$ (H4.2)
- $\rho_2 \vdash \sigma_2, e_1 \downarrow v_{n_{21}} \sigma_1'$ (H5.1)
- $\rho_2 \vdash \sigma_2', e_2 \downarrow v_{n_{22}} \sigma_1''$, $v_{22}$ (H5.2)

and we also have: $n_1 = n_{11} + n_{12}$, $n_2 = n_{21} + n_{22}$, $\sigma_1' = \sigma_1'$, $\sigma_2' = \sigma_2''$, $v_1 = v_{11}$, $v_2 = v_{12}$. By IH ($n = n$ decreasing), (H1), (H2), (H3), (H4.1) and (H5.1) we have: $n_{11} = n_{11}$ (IH.1.C1); $\sigma_1' \sim_{n-n_{11} - n_{22}} \sigma_2'$ (IH.1.C2); and $v_1 \sim_{n-n_{11} - n_{22}} v_{11}$ (IH.1.C3). Note the following facts: $n_{12} \leq n - n_{11}$ (F1); and $\rho_1 \sim_{n-n_{11}} (F2)$. (F1) follows from (H3) and $n_1 = n_{11} + n_{12}$ (F2) follows from (H1) and Step-index Weakening.1. By IH ($n = n - n_{11}$ decreasing), (F2), (IH.1.C2), (F1), (H4.2) and (H5.2) we have: $n_{21} = n_{22}$ (IH.1.C2); $\sigma_1' \sim_{n-n_{11} - n_{22}} \sigma_2'$ (IH.2.C2); and $v_1 \sim_{n-n_{11} - n_{22}} v_{22}$ (IH.2.C3). To show: (C1): $n_{11} + n_{12} = n_{21} + n_{22}$; (C2): $\sigma_1' \sim_{n-n_{11} - n_{22}} \sigma_2''$; and (C3): $v_1 \sim_{n-n_{11} - n_{22}} v_{22}$. (C1) is immediate from (IH.1.C1) and (IH.1.C2). (C3) is immediate from (IH.2.C2). (C3) is immediate from (IH.2.C3).

- **Case** $n = n + 1$ and $e = \sigma_1(e)$:

By inversion on (H4) and (H5) we have:

- $\rho_1 \vdash \sigma_1, v_{n_{11}} \sigma_1', v_{11}$ (H4.1)
- $\rho_1 \vdash \sigma_1', v_{n_{12}} \sigma_1''$, $v_{12}$ (H4.2)
- $\rho_2 \vdash \sigma_2, v_{n_{21}} \sigma_1'$ (H5.1)
- $\rho_2 \vdash \sigma_2', v_{n_{22}} \sigma_1''$, $v_{22}$ (H5.2)

and we also have: $n_1 = n_{11} + n_{12} + n_{13} + 1$, $n_2 = n_{21} + n_{22} + n_{23} + 1$, $\sigma_1' = \sigma_1'$, $\sigma_2' = \sigma_2''$, $v_1 = v_1$, and $v_2 = v_2$. By IH ($n = n$ decreasing), (H1), (H2), (H3), (H4.1) and (H5.1) we have: $n_{11} = n_{21}$ (IH.1.C1); $\sigma_1' \sim_{n-n_{11} - n_{22}} \sigma_2'$ (IH.1.C2); and $\langle \lambda x. e_1' || \rho_1 \rangle \sim_{n-n_{11}} v_{21}$ (IH.1.C3). Note the following facts: $n_{12} \leq n - n_{11}$ (F1); and $\rho_1 \sim_{n-n_{11}} (F2)$. (F1) follows from (H3) and $n_1 = n_{11} + n_{12}$ (F2) follows from (H1) and Step-index Weakening.1. By IH ($n = n - n_{11}$ decreasing), (H2), (IH.1.C2), (F1), (H4.2) and (H5.2) we have: $n_{21} = n_{22}$ (IH.2.C2); $\sigma_1' \sim_{n-n_{11} - n_{22}} \sigma_2'$ (IH.2.C2); and $v_1 \sim_{n-n_{11} - n_{12}} v_{22}$ (IH.2.C3). Note the following facts, each of which follow from (H3) and $n_1 = n_{11} + n_{12} + n_{13} + 1$: $n_{13} \leq n - n_{11} - n_{12} - n_{13}$ (F3); and $n_{11} - n_{12} - n_{13} > 0$ (F4). Also note the following facts which follow from (IH.1.C3), (IH.2.C2), (IH.2.C3), (F3) and (F4): $n_{13} = n_{23}$ (F4.C1); $\sigma_1'' \sim_{n-n_{11} - n_{13} - n_{12} - n_{13} - 1} \sigma_2''$ (F4.C2); and $v_1' \sim_{n-n_{11} - n_{13} - n_{12} - n_{13} - 1} v'$ (F4.C3). To show: (C1): $n_{11} + n_{12} + n_{13} + 1 = n_{21} + n_{22} + n_{23} + 1$; (C2): $\sigma_1' \sim_{n-n_{11} - n_{13} - n_{12} - 1} \sigma_2'$; and (C3): $v_1' \sim_{n-n_{11} - n_{13} - n_{12} - 1} v'$. (C1) is immediate from (IH.1.C1), (IH.1.C2) and (IH.1.C4). (C2) is immediate from (F4.C2) and Step-index Weakening.2. (C3) is immediate from (F4.C3) and Step-index Weakening.3.

- **Case** $n = n + 1$ and $e = \text{return}(e)$:

By inversion on (H4) and (H5) we have:
\( \rho_1 \vdash \sigma_1, e \downarrow n_1, \sigma'_1, v_1 \) \hspace{1cm} (H4.1) \\
\( \ell_1 = \text{alloc}(\text{dom}(\sigma'_1)) \) \hspace{1cm} (H4.2) \\
\( \rho_2 \vdash \sigma_2, e \downarrow n_2, \sigma'_2, v_2 \) \hspace{1cm} (H5.1) \\
\( \ell_2 = \text{alloc}(\text{dom}(\sigma'_2)) \) \hspace{1cm} (H4.2)

and we also have: 
\( n_1 = n_1, \ n_2 = n_2, \) 
\( \sigma'_1 = \{ \ell \mapsto v_1 \} \uplus \sigma'_1, \) 
\( \sigma'_2 = \{ \ell \mapsto v_2 \} \uplus \sigma'_2, \) 
\( v_1 = \ell_1 \) and 
\( v_2 = \ell_2. \) By IH \((n = n \text{ decreasing}), \) (H1), (H2), (H3), (H4.1) and (H5.1) we have: 
\( n_1 = n_2 \) \hspace{1cm} (IH.1.C1); 
\( \sigma'_1 \sim_{n-n_1} \sigma'_2 \) \hspace{1cm} (IH.1.C2); and 
\( v_1 \sim_{n-n_1} v_2 \) \hspace{1cm} (IH.1.C3).

Because \( \sigma_1 \sim_{n-n_1} \sigma_2, \) we know \( \text{dom}(\sigma_1) = \text{dom}(\sigma_2) \) and therefore \( \ell_1 = \ell_2. \) To show: (C1): \( n_1 = n_2; \) 
\( C2): \{ \ell \mapsto v_1 \} \uplus \sigma'_1 \sim_{n-n_1} \{ \ell \mapsto v_2 \} \uplus \sigma'_2, \) and 
(C3): \( \ell_1 \sim_{n-n_1} \ell_2. \) (C1) is immediate from (IH.1.C1). (C2) is immediate from (IH.1.C2), (IH.1.C3) and the definition of \( \sigma \sim_{n} \sigma. \) (C3) is immediate by definition of \( v \sim_{n} v \) and \( \ell_1 = \ell_2. \)

- **Case** \( n = n+1 \) and \( e = e_1 \)

By inversion on (H4) and (H5) we have:

\( \rho_1 \vdash \sigma_1, e_1 \downarrow n_1, \sigma'_1, \ell_1 \) \hspace{1cm} (H4.1) \\
\( \rho_2 \vdash \sigma_2, e_2 \downarrow n_2, \sigma'_2, \ell_2 \) \hspace{1cm} (H5.1)

and we also have: 
\( n_1 = n_1, \ n_2 = n_2, \) 
\( \sigma'_1 = \sigma'_1, \) 
\( \sigma'_2 = \sigma'_2, \) 
\( v_1 = \sigma'_1(\ell_1) \) and 
\( v_2 = \sigma'_2(\ell_2). \) By IH \((n = n \text{ decreasing}), \) (H1), (H2), (H3), (H4.1) and (H5.1) we have: 
\( n_1 = n_2 \) \hspace{1cm} (IH.1.C1); 
\( \sigma'_1 \sim_{n-n_1} \sigma'_2 \) \hspace{1cm} (IH.1.C2); and 
\( \ell_1 \sim_{n-n_1} \ell_2 \) \hspace{1cm} (IH.1.C3).

Because \( \ell_1 \sim_{n-n_1} \ell_2, \) we know \( \ell_1 = \ell_2 \) by definition of \( v \sim_{n} v. \) To show: (C1): \( n_1 = n_2; \) 
\( C2): \{ \ell \mapsto v_1 \} \uplus \sigma'_1 \sim_{n-n_1} \{ \ell \mapsto v_2 \} \uplus \sigma'_2, \) and 
(C3): \( \sigma'_1(\ell_1) \sim_{n-n_1} \sigma'_2(\ell_2). \) (C1) is immediate from (IH.1.C1). (C2) is immediate from (IH.1.C2). (C3) is immediate from (IH.1.C2) and (IH.2.C1).

\( \square \)

### Appendix B

#### Additional Case Studies

**A. MWEM with Pandas**

The MWEM algorithm [29] constructs a differentially private synthetic dataset that approximates a real dataset. MWEM produces competitive privacy bounds by utilizing a combination of the exponential mechanism, Laplacian/Gaussian noise, and the multiplicative weights update rule. The algorithm uses these mechanisms iteratively, providing a tight analysis of privacy leakage via composition.

The inputs to the MWEM are as follows: some uniform or random distribution over a domain \((\text{syn}_{\text{data}}),\) some sensitive dataset \((\text{age}\_\text{counts}),\) a query workload, a number of iterations \(i\), and a privacy budget \(\epsilon.\)

The algorithm works by, at each iteration:

- privately selecting a query from the query workload (using the exponential mechanism) whose result on the synthetic dataset greatly differs from the real dataset

\[
\text{for } t \text{ in range}(i): \\
q = \text{exponential}(\text{q\_workload, score\_fn, eps}\{(2+i)\}) \\
\ldots
\]

- then privately using the query result on the real dataset to adjust the synthetic dataset towards the truth using the multiplicative weights update rule

\[
\text{for } t \text{ in range}(i): \\
\ldots \\
\text{syn}_{\text{data}} = \text{mwem\_step}(q, \text{age}\_\text{counts}, \text{syn}_{\text{data}})
\]

We present a modified, adaptive MWEM algorithm (Figure 5) which privately halts execution if the error of the synthetic dataset reaches an acceptably low level before the entire privacy budget is exceeded, conserving the remainder of the budget for other private analyses.

**B. DiffPrivLib**

DiffPrivLib is library for experimenting with analytics and machine learning with differential privacy in Python by IBM. It provides a comprehensive suite of differentially private mechanisms, tools, and machine learning models.

While DiffPrivLib provides several mechanisms, models and tools for developing private applications, as well as a basic privacy accountant, it lacks the ability to perform a tight privacy analysis in the context of more sophisticated forms of composition with dynamic and adaptive privacy tracking. Via integration with DDUO we are able to gain these abilities with minimal changes to library and program code.

Figure 6 shows an example of a modified DiffPrivLib program: a private naive Bayes classifier run on the standard iris dataset. The original program has been modified with DDUO hooks to detect sensitivity violations and track privacy cost.
```python
def mwem_step(query, real_data, syn_data):
    lower, upper = query
    sm = [v for k, v in syn_data.items()]
    total = np.sum(sm)
    q_ans = range_query(real_data, lower, upper)
    real = dduo.renyi_gauss(α=alpha, ϵ=eps, q_ans, sens)
    syn = range_query(syn_data, lower, upper)
    l = [(k, mwem_update(k, x, lower, upper, real, syn, total))
        for k, x in syn_data.items()]
    return dict(l)
```