PyDP

Release 1.1.1

OpenMined

Oct 15, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION TO DIFFERENTIAL PRIVACY

CHAPTER

TWO

INTRODUCTION

The era where we are living in is data driven, tons and tons of data are being generated in every second. A lot of this data is being used to improve our own lifestyle - be it recommending the best series to watch after a tiring day at work, suggesting the best gifts to buy when it's our best friend's birthday or keeping our birthday party photos sorted so that we can cherish them years later. All big companies are using data to gain insights of their progress which drives their business. Machine Learning has made our life from easy to easier but is it just about improving our lifestyle? This raises a question can machine learning change the way we live ? Can it improve our healthcare? Can ML be friends to those who are lonely and have no one to talk with? The answer is "Yes" and also "No".

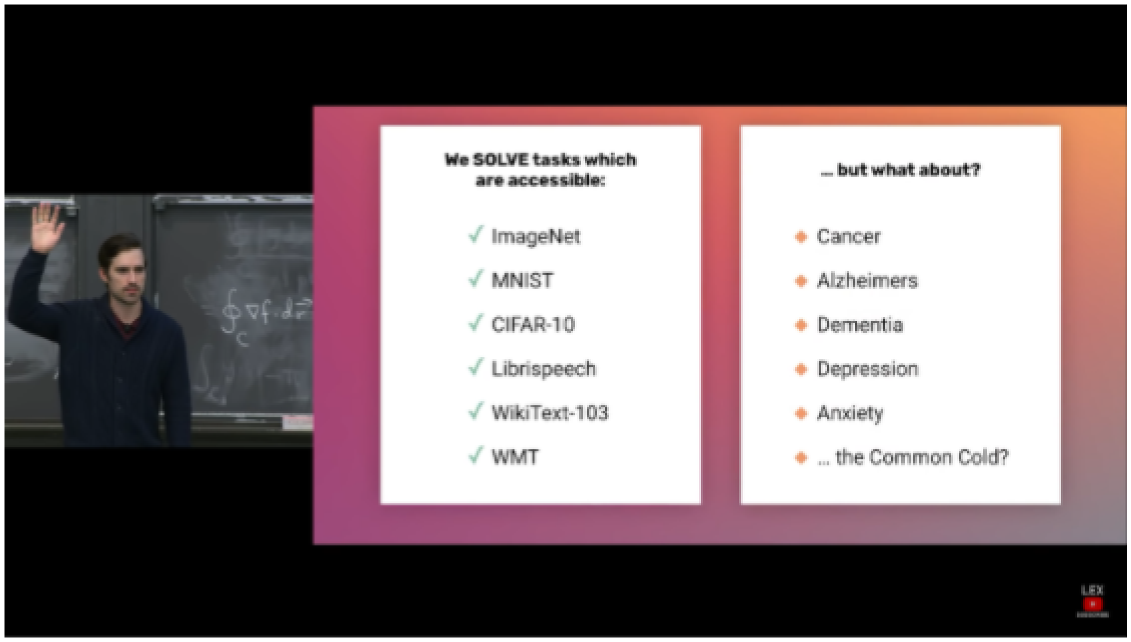
CHAPTER

THREE

MACHINE LEARNING AND DATA

Machine Learning is extensively both data and research driven. The more the data is, better will be the research on that particular topic. Now, all data cannot be released for research, there is a lot of private information which once leaked can be misused. Take for example, to tackle a particular medical problem we need a lot of medical health records. These records are considered as private information as no person would love the fact that her/his medical records are identifiable by anyone on the internet. Hence, these are some real world issues that need immediate solutions but the hands of the researchers are tied due to the unavailability of data. So, is there a solution ?

This is where "Differential Privacy" comes into the picture, a smarter way to a more secure and private AI. According to Andrew Trask, Founder at OpenMined - "Differential Privacy is the process to answer questions or solve problems using the data that we cannot see." In this way researchers from all over the world can use private data in their research work without identifying the individual.



We SOLVE tasks which are accessible:

- ✓ ImageNet
- ✓ MNIST
- ✓ CIFAR-10
- ✓ Librispeech
- ✓ WikiText-103
- ✓ WMT

... but what about?

- Cancer
- Alzheimers
- Dementia
- Depression
- Anxiety
- ... the Common Cold?

Fig. 1: (Privacy Preserving AI (Andrew Trask) | MIT Deep Learning Series)

CHAPTER

FOUR

WHY IS DIFFERENTIAL PRIVACY SO IMPORTANT ?

The aim of any privacy algorithm is to keep one's private information safe and secured from external attacks. Differential privacy aims to keep an individual's identity secured even if their data is being used in research. An easy approach to maintain this kind of privacy is "Data Anonymization" which is a process of removing personally identifiable information from a dataset. It is seen that there are cons in following this approach:

- Anonymizing certain fields may make the entire dataset useless and not fit for any analysis.
- There are related sources or datasets available on the web and by statistically studying both the datasets, an individual can easily be re-identified.
- If the dataset is large, the type of queries that can be drawn from the dataset cannot be predicted. This makes any dataset prone to external attacks.

Hence, this process is prone to risk and is considered as fundamentally wrong. Netflix once released a challenge for everyone to build up the best recommendation engine. For this they released an anonymized dataset of 100 million movie ratings from half a million users. So, they did not publicly release any data that could lead to the identification of the users.

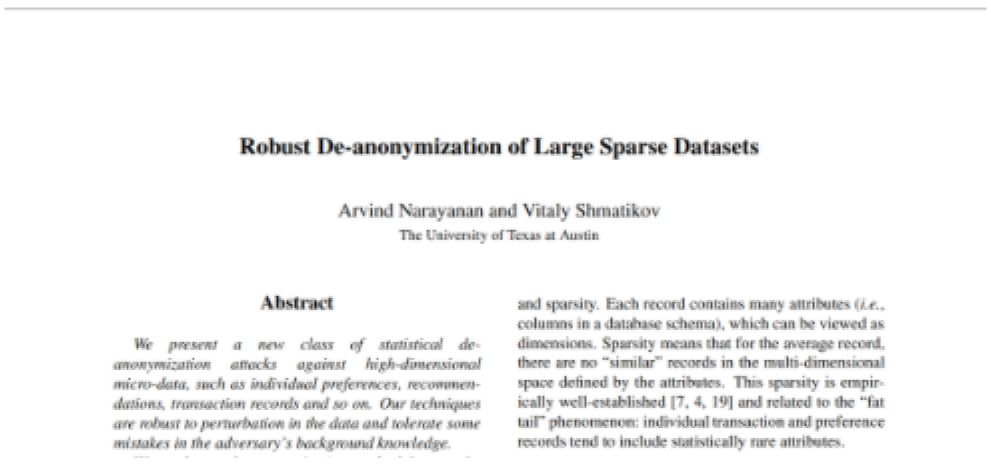
The Netflix Prize sought to substantially improve the accuracy of predictions about how much someone is going to enjoy a movie based on their movie preferences.

On September 21, 2009 we awarded the $1M Grand Prize to team "BellKor's Pragmatic Chaos". Read about their algorithm, checkout team scores on the Leaderboard, and join the discussions on the Forum.

We applaud all the contributors to this quest, which improves our ability to connect people to the movies they love.

Fig. 1: Image Credits: Secure and Private AI (Udacity)

Despite the fact that the dataset was anonymized (no username or movie name was released) yet two Researchers at University of Texas released a paper where they showed how they have de-anonymized a maximum chunk of the dataset.



Robust De-anonymization of Large Sparse Datasets

Arvind Narayanan and Vitaly Shmatikov
The University of Texas at Austin

Abstract

We present a new class of statistical de-anonymization attacks against high-dimensional micro-data, such as individual preferences, recommendations, transaction records and so on. Our techniques are robust to perturbation in the data and tolerate some mistakes in the adversary's background knowledge.

and sparsity. Each record contains many attributes (*i.e.*, columns in a database schema), which can be viewed as dimensions. Sparsity means that for the average record, there are no "similar" records in the multi-dimensional space defined by the attributes. This sparsity is empirically well-established [7, 4, 19] and related to the "fat tail" phenomenon: individual transaction and preference records tend to include statistically rare attributes.

They scraped the IMDB Website and by statistical analysis on these two datasets, they were able to identify the movie names and also the individual names. Ten years down the line they have published yet another research paper where they have reviewed de-anonymization of datasets in the present world. There are other instances too where such attacks have been made which led to the leakage of private information.

Now, that we have learnt how important is "Differential Privacy", let see how is the Differential Privacy actually implemented.

CHAPTER

FIVE

HOW IS DIFFERENTIAL PRIVACY IMPLEMENTED ?

According to Cynthia Dwork- *"Differential privacy" describes a promise, made by a data holder, or curator, to a data subject: "You will not be affected, adversely or otherwise, by allowing your data to be used in any study or analysis, no matter what other studies, data sets, or information sources, are available."*

Thus this new area of research addresses the paradox of learning nothing about an individual while learning useful information about the population. This is done by sending queries (a function applied to a database) to the data curator (a protocol run by the set of individuals, using the various techniques for secure multiparty protocols). The goal of the curator is to answer all the queries with highest possible accuracy without leaking any individual information using various Differential-Privacy algorithms.

These algorithms add random noise to the queries and to the database. This is done in two ways:

- Local Differential Privacy
- Global Differential Privacy

5.1 Local Differential Privacy

In local differential privacy the random noise is applied at the start of the process(local) level i.e when the data is sent to the data curator/aggregator. If the data is too confidential, generally the data generators do not want to trust the curator and hence add noise to the dataset beforehand. This is adopted when the Data Curator cannot be completely trusted.

5.2 Global Differential Privacy

In Global differential privacy the random noise is applied at the global level i.e when the answer to a query is returned to the User. This type of differential privacy is adopted when the Data generators trusts the data curator completely and leaves it to the curator the amount of noise to be added to the results. This type of privacy results is more accurate as it involves lesser noise.

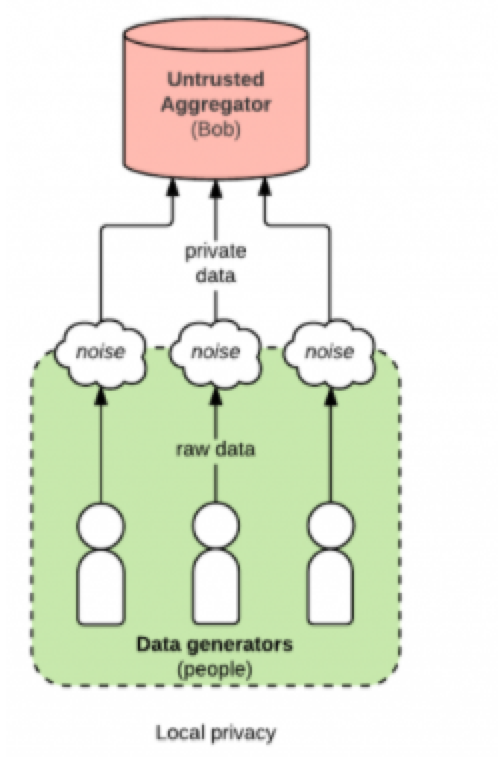


Fig. 1: Image Credit: Google Images

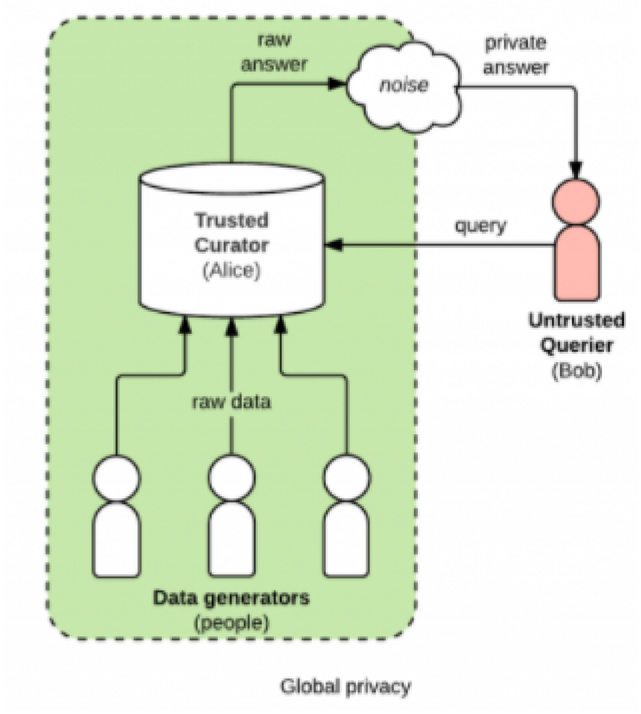


Fig. 2: Image Credits: Google Images

CHAPTER

SIX

FORMAL DEFINITION OF DIFFERENTIAL PRIVACY

In the book, "The Algorithmic Foundations of Differential Privacy" by Cynthia Dwork and Aaron Roth. Differential Privacy is formally defined as: .. glossary:: *A randomized algorithm M with domain N |X| is (,)-differentially private if for all S Range(M) and for all x, y N |X| such that x y1 1:*

> *Pr[M(x) S] exp() Pr[M(y) S]* +

The Epsilon *()* and *Delta()* parameters measure the threshold for leakage.

- The Epsilon defines how different the actual actual data is from the queried data. If *=0*, exp()=1 which means both the data are equal.
- The Delta is the probability that an information will accidentally be leaked as compared to the value of Epsilon. If *=0*, that means no data is being leaked.

This when both Epsilon and Delta is 0, it is called Perfect-Privacy. The values are set in such a way so that the privacy is maintained. This set of values is known as Privacy-Budget.

CHAPTER

SEVEN

DIFFERENTIAL - PRIVACY IN REAL WORLD

Differential Privacy ensures privacy of all sorts of data which can be used by anyone to draw insights which can help them run their business. In the present world, Differentially Private Data Analysis is widely used and these are implemented by using various libraries.

PyDP by OpenMined is a Python Wrapper for Differential Privacy which allows all sorts of users to use Differential Privacy in their Projects. Apart from this there are various other real-world cases of Differential Privacy from Medical Imaging to Geolocation search. These have been covered in this blogpost by OpenMined.

SOME OTHER LIBRARIES FOR DP

- OpenDp by Harvard University and Microsoft
- Diffprivlib by IBM
- Google's Differential Privacy Library .

DIFFERENTIAL PRIVACY IN USE

Top tech companies are using "Differential Privacy" in their day to day business for the privacy of data. Some of the use cases are here as follows:

- Uber

Uber, a popular ride-sharing company uses Differential Privacy in its practices. The company uses a method of Differential Privacy called "elastic sensitivity", developed in the University of California at Berkeley. It uses mathematics to set limits on the number of statistical queries the staff can conduct on traffic patterns and driver's revenue. This method also ensures addition of noise in case the potential of a privacy breach is more severe.

- Apple

Apple also makes use of differential privacy to analyse user behaviour and improve user experience. Accessing private data such as browsing history, apps that we browse, words that we type etc can compromise user privacy. But these data are extremely useful when it comes to improving user experience. Apple makes use of "Local Differential Privacy" algorithms which ensures that the raw data is randomized before sending it to the servers. This approach is implemented at scale across on millions of users and by harnessing this data various business decisions are taken.

- Google

Google also uses this novel approach to keep user data private to themselves and perform data analysis with that data to drive some of their core products. One such product is the Gboard (Google Keyboard), where it uses private data of the user to generate word suggestions. The method used is "Federated Learning" which decreases the reliance on the cloud and puts a strong focus on a user's privacy. Rather than sending encrypted data to the servers, it downloads the current model on device and improves it by learning from the data on device. The updated model with the changes is sent to the cloud using encrypted communication. This is done at scale across all users and the updates from each user is immediately averaged with other updates to improve the shared model. In the year 2019, Google open sourced the Differential Privacy library for others to use.

Differential Privacy is playing an important role in building Privacy-protected Machine Learning solutions. PyDP is an effort to democratize this field. To know more about Differential Privacy and PyDP head over to our amazing blog series at OpenMined Blog.

CHAPTER EIGHT

FURTHER READING

- Secure and Private AI Course on Udacity by Andrew Trask
- “The Algorithmic Foundations of Differential Privacy” by Cynthia Dwork and Aaron Roth
- OpenMined Blogs on Differential Privacy

CHAPTER

NINE

INTRODUCTION TO PYDP

In today's data-driven world, more and more researchers and data scientists use machine learning to create better models or more innovative solutions for a better future.

These models often tend to handle sensitive or personal data, which can cause privacy issues. For example, some AI models can memorize details about the data they've been trained on and could potentially leak these details later on.

To help measure sensitive data leakage and reduce the possibility of it happening, there is a mathematical framework called differential privacy.

In 2020, OpenMined created a Python wrapper for Google's Differential Privacy project called PyDP. The library provides a set of -differentially private algorithms, which can be used to produce aggregate statistics over numeric data sets containing private or sensitive information. Therefore, with PyDP you can control the privacy guarantee and accuracy of your model written in Python.

Things to remember about PyDP:

- ::rocket: Features differentially private algorithms including: BoundedMean, BoundedSum, Max, Count Above, Percentile, Min, Median, etc.
- All the computation methods mentioned above use Laplace noise only (other noise mechanisms will be added soon! :smiley:).
- ::fire: Currently supports Linux and macOS (Windows support coming soon :smiley:)
- ::star: Use Python 3.6+. Support for Python 3.5 and below is deprecated.

9.1 Installation

To install PyDP, use the PiPy package manager:

```
pip install python-dp
```

(If you have `pip3` separately for Python 3.x, use `pip3 install python-dp`.)

9.2 Examples

Refer to the curated list of tutorials and sample code to learn more about the PyDP library.

You can also get started with an introduction to PyDP (a Jupyter notebook) and the carrots demo (a Python file).

Example: calculate the Bounded Mean

```
# Import PyDP
import pydp as dp
# Import the Bounded Mean algorithm
from pydp.algorithms.laplacian import BoundedMean

# Calculate the Bounded Mean
# Basic Structure: `BoundedMean(epsilon: float, lower_bound: Union[int, float, None],␣
↪upper_bound: Union[int, float, None])`
# `epsilon`: a Double, between 0 and 1, denoting the privacy threshold,
#            measures the acceptable loss of privacy (with 0 meaning no loss is␣
↪acceptable)
x = BoundedMean(epsilon=0.6, lower_bound=1, upper_bound=10)

# If the lower and upper bounds are not specified,
# PyDP automatically calculates these bounds
# x = BoundedMean(epsilon: float)
x = BoundedMean(0.6)

# Calculate the result
# Currently supported data types are integers and floats
# Future versions will support additional data types
# (Refer to https://github.com/OpenMined/PyDP/blob/dev/examples/carrots.py)
x.quick_result(input_data: list)
```

9.3 Learning Resources

Go to resources to learn more about differential privacy.

9.4 Support and Community on Slack

If you have questions about the PyDP library, join OpenMined's Slack and check the **#lib_pydp** channel. To follow the code source changes, join **#code_dp_python**.

9.5 Contributing

To contribute to the PyDP project, read the guidelines.

Pull requests are welcome. If you want to introduce major changes, please open an issue first to discuss what you would like to change.

Please make sure to update tests as appropriate.

<!– ## Contributors –>

9.6 License

Apache License 2.0

CHAPTER

TEN

PYDP

10.1 Algorithms

class pydp.algorithms.laplacian.**BoundedMean**(*epsilon: float = 1.0*, *delta: float = 0*, *lower_bound: Optional[Union[int, float]] = None*, *upper_bound: Optional[Union[int, float]] = None*, *l0_sensitivity: int = 1*, *linf_sensitivity: int = 1*, *dtype: str = 'int'*)

BoundedMean computes the average of values in a dataset, in a differentially private manner.

Incrementally provides a differentially private average. All input vales are normalized to be their difference from the middle of the input range. That allows us to calculate the sum of all input values with half the sensitivity it would otherwise take for better accuracy (as compared to doing noisy sum / noisy count). This algorithm is taken from section 2.5.5 of the following book (algorithm 2.4): https://books.google.com/books?id=WFttDQAAQBAJ&pg=PA24#v=onepage&q&f=false

add_entries(*data: List[Union[int, float]]*) → None

Adds multiple inputs to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current list passed is not added.

add_entry(*value: Union[int, float]*) → None

Adds one input to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current data passed is not added.

property delta: float

Returns the epsilon set at initialization.

property epsilon: float

Returns the epsilon set at initialization.

property l0_sensitivity: float

Returns the l0_sensitivity set at initialization.

property linf_sensitivity: float

Returns the linf_sensitivity set at initialization.

memory_used() → float

Returns the memory currently used by the algorithm in bytes.

merge(*summary*)

Merges serialized summary data into this algorithm. The summary proto must represent data from the same algorithm type with identical parameters. The data field must contain the algorithm summary type of the corresponding algorithm used. The summary proto cannot be empty.

noise_confidence_interval(*confidence_level: float*, *privacy_budget: float*) → float

Returns the confidence_level confidence interval of noise added within the algorithm with specified privacy budget, using epsilon and other relevant, algorithm-specific parameters (e.g. bounds) provided by the constructor.

This metric may be used to gauge the error rate introduced by the noise.

If the returned value is <x,y>, then the noise added has a confidence_level chance of being in the domain [x,y].

By default, NoiseConfidenceInterval() returns an error. Algorithms for which a confidence interval can feasibly be calculated override this and output the relevant value.

Conservatively, we do not release the error rate for algorithms whose confidence intervals rely on input size.

privacy_budget_left() → float

Returns the remaining privacy budget.

quick_result(*data: List[Union[int, float]]*) → Union[int, float]

Runs the algorithm on the input using the epsilon parameter provided in the constructor and returns output.

Consumes 100% of the privacy budget.

Note: It resets the privacy budget first.

reset() → None

Resets the algorithm to a state in which it has received no input. After Reset is called, the algorithm should only consider input added after the last Reset call when providing output.

result(*privacy_budget: Optional[float] = None*, *noise_interval_level: Optional[float] = None*) → Union[int, float]

Gets the algorithm result.

The default call consumes the remaining privacy budget.

When *privacy_budget* (defined on [0,1]) is set, it consumes only the *privacy_budget* amount of budget.

noise_interval_level provides the confidence level of the noise confidence interval, which may be included in the algorithm output.

serialize()

Serializes summary data of current entries into Summary proto. This allows results from distributed aggregation to be recorded and later merged.

Returns empty summary for algorithms for which serialize is unimplemented.

class pydp.algorithms.laplacian.**BoundedSum**(*epsilon: float = 1.0*, *delta: float = 0*, *lower_bound: Optional[Union[int, float]] = None*, *upper_bound: Optional[Union[int, float]] = None*, *l0_sensitivity: int = 1*, *linf_sensitivity: int = 1*, *dtype: str = 'int'*)

BoundedSum computes the sum of values in a dataset, in a differentially private manner.

Incrementally provides a differentially private sum, clamped between upper and lower values. Bounds can be manually set or privately inferred.

add_entries(*data: List[Union[int, float]]*) → None

Adds multiple inputs to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current list passed is not added.

add_entry(*value: Union[int, float]*) → None

Adds one input to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current data passed is not added.

property delta: float

Returns the epsilon set at initialization.

property epsilon: float

Returns the epsilon set at initialization.

property l0_sensitivity: float

Returns the l0_sensitivity set at initialization.

property linf_sensitivity: float

Returns the linf_sensitivity set at initialization.

memory_used() → float

Returns the memory currently used by the algorithm in bytes.

merge(*summary*)

Merges serialized summary data into this algorithm. The summary proto must represent data from the same algorithm type with identical parameters. The data field must contain the algorithm summary type of the corresponding algorithm used. The summary proto cannot be empty.

noise_confidence_interval(*confidence_level: float*, *privacy_budget: float*) → float

Returns the confidence_level confidence interval of noise added within the algorithm with specified privacy budget, using epsilon and other relevant, algorithm-specific parameters (e.g. bounds) provided by the constructor.

This metric may be used to gauge the error rate introduced by the noise.

If the returned value is <x,y>, then the noise added has a confidence_level chance of being in the domain [x,y].

By default, NoiseConfidenceInterval() returns an error. Algorithms for which a confidence interval can feasibly be calculated override this and output the relevant value.

Conservatively, we do not release the error rate for algorithms whose confidence intervals rely on input size.

privacy_budget_left() → float

Returns the remaining privacy budget.

quick_result(*data: List[Union[int, float]]*) → Union[int, float]

Runs the algorithm on the input using the epsilon parameter provided in the constructor and returns output.

Consumes 100% of the privacy budget.

Note: It resets the privacy budget first.

reset() → None

Resets the algorithm to a state in which it has received no input. After Reset is called, the algorithm should only consider input added after the last Reset call when providing output.

result(*privacy_budget: Optional[float] = None*, *noise_interval_level: Optional[float] = None*) → Union[int, float]

Gets the algorithm result.

The default call consumes the remaining privacy budget.

When *privacy_budget* (defined on [0,1]) is set, it consumes only the *privacy_budget* amount of budget.

noise_interval_level provides the confidence level of the noise confidence interval, which may be included in the algorithm output.

serialize()

Serializes summary data of current entries into Summary proto. This allows results from distributed aggregation to be recorded and later merged.

Returns empty summary for algorithms for which serialize is unimplemented.

class pydp.algorithms.laplacian.**BoundedStandardDeviation**(*epsilon: float = 1.0*, *delta: float = 0*, *lower_bound: Optional[Union[int, float]] = None*, *upper_bound: Optional[Union[int, float]] = None*, *l0_sensitivity: int = 1*, *linf_sensitivity: int = 1*, *dtype: str = 'int'*)

BoundedStandardDeviation computes the standard deviation of values in a dataset, in a differentially private manner.

Incrementally provides a differentially private standard deviation for values in the range [lower..upper]. Values outside of this range will be clamped so they lie in the range. The output will also be clamped between 0 and (upper - lower).

The implementation simply computes the bounded variance and takes the square root, which is differentially private by the post-processing theorem. It relies on the fact that the bounded variance algorithm guarantees that the output is non-negative.

add_entries(*data: List[Union[int, float]]*) → None

Adds multiple inputs to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current list passed is not added.

add_entry(*value: Union[int, float]*) → None

Adds one input to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current data passed is not added.

property delta: float

Returns the epsilon set at initialization.

property epsilon: float

Returns the epsilon set at initialization.

property l0_sensitivity: float

Returns the l0_sensitivity set at initialization.

property linf_sensitivity: float

Returns the linf_sensitivity set at initialization.

memory_used() → float

Returns the memory currently used by the algorithm in bytes.

merge(*summary*)

Merges serialized summary data into this algorithm. The summary proto must represent data from the same algorithm type with identical parameters. The data field must contain the algorithm summary type of the corresponding algorithm used. The summary proto cannot be empty.

noise_confidence_interval(*confidence_level: float*, *privacy_budget: float*) → float

Returns the confidence_level confidence interval of noise added within the algorithm with specified privacy budget, using epsilon and other relevant, algorithm-specific parameters (e.g. bounds) provided by the constructor.

This metric may be used to gauge the error rate introduced by the noise.

If the returned value is <x,y>, then the noise added has a confidence_level chance of being in the domain [x,y].

By default, NoiseConfidenceInterval() returns an error. Algorithms for which a confidence interval can feasibly be calculated override this and output the relevant value.

Conservatively, we do not release the error rate for algorithms whose confidence intervals rely on input size.

privacy_budget_left() → float

Returns the remaining privacy budget.

quick_result(*data: List[Union[int, float]]*) → Union[int, float]

Runs the algorithm on the input using the epsilon parameter provided in the constructor and returns output.

Consumes 100% of the privacy budget.

Note: It resets the privacy budget first.

reset() → None

Resets the algorithm to a state in which it has received no input. After Reset is called, the algorithm should only consider input added after the last Reset call when providing output.

result(*privacy_budget: Optional[float] = None*, *noise_interval_level: Optional[float] = None*) → Union[int, float]

Gets the algorithm result.

The default call consumes the remaining privacy budget.

When *privacy_budget* (defined on [0,1]) is set, it consumes only the *privacy_budget* amount of budget.

noise_interval_level provides the confidence level of the noise confidence interval, which may be included in the algorithm output.

serialize()

Serializes summary data of current entries into Summary proto. This allows results from distributed aggregation to be recorded and later merged.

Returns empty summary for algorithms for which serialize is unimplemented.

class pydp.algorithms.laplacian.**BoundedVariance**(*epsilon: float = 1.0*, *delta: float = 0*, *lower_bound: Optional[Union[int, float]] = None*, *upper_bound: Optional[Union[int, float]] = None*, *l0_sensitivity: int = 1*, *linf_sensitivity: int = 1*, *dtype: str = 'int'*)

BoundedVariance computes the variance of values in a dataset, in a differentially private manner.

Incrementally provides a differentially private variance for values in the range [lower..upper]. Values outside of this range will be clamped so they lie in the range. The output will also be clamped between 0 and (upper - lower)^2. Since the result is guaranteed to be positive, this algorithm can be used to compute a differentially private standard deviation.

The algorithm uses O(1) memory and runs in O(n) time where n is the size of the dataset, making it a fast and efficient. The amount of noise added grows quadratically in (upper - lower) and decreases linearly in n, so it might not produce good results unless n >> (upper - lower)^2.

The algorithm is a variation of the algorithm for differentially private mean from "Differential Privacy: From Theory to Practice", section 2.5.5: https://books.google.com/books?id=WFttDQAAQBAJ&pg=PA24#v=onepage&q&f=false

`add_entries`(*data: List[Union[int, float]]*) → None

Adds multiple inputs to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current list passed is not added.

`add_entry`(*value: Union[int, float]*) → None

Adds one input to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current data passed is not added.

property delta: float

Returns the epsilon set at initialization.

property epsilon: float

Returns the epsilon set at initialization.

property l0_sensitivity: float

Returns the l0_sensitivity set at initialization.

property linf_sensitivity: float

Returns the linf_sensitivity set at initialization.

`memory_used`() → float

Returns the memory currently used by the algorithm in bytes.

`merge`(*summary*)

Merges serialized summary data into this algorithm. The summary proto must represent data from the same algorithm type with identical parameters. The data field must contain the algorithm summary type of the corresponding algorithm used. The summary proto cannot be empty.

`noise_confidence_interval`(*confidence_level: float*, *privacy_budget: float*) → float

Returns the confidence_level confidence interval of noise added within the algorithm with specified privacy budget, using epsilon and other relevant, algorithm-specific parameters (e.g. bounds) provided by the constructor.

This metric may be used to gauge the error rate introduced by the noise.

If the returned value is <x,y>, then the noise added has a confidence_level chance of being in the domain [x,y].

By default, NoiseConfidenceInterval() returns an error. Algorithms for which a confidence interval can feasibly be calculated override this and output the relevant value.

Conservatively, we do not release the error rate for algorithms whose confidence intervals rely on input size.

`privacy_budget_left`() → float

Returns the remaining privacy budget.

`quick_result`(*data: List[Union[int, float]]*) → Union[int, float]

Runs the algorithm on the input using the epsilon parameter provided in the constructor and returns output.

Consumes 100% of the privacy budget.

Note: It resets the privacy budget first.

reset() → None

Resets the algorithm to a state in which it has received no input. After Reset is called, the algorithm should only consider input added after the last Reset call when providing output.

result(*privacy_budget: Optional[float] = None*, *noise_interval_level: Optional[float] = None*) → Union[int, float]

Gets the algorithm result.

The default call consumes the remaining privacy budget.

When *privacy_budget* (defined on [0,1]) is set, it consumes only the *privacy_budget* amount of budget.

noise_interval_level provides the confidence level of the noise confidence interval, which may be included in the algorithm output.

serialize()

Serializes summary data of current entries into Summary proto. This allows results from distributed aggregation to be recorded and later merged.

Returns empty summary for algorithms for which serialize is unimplemented.

`class pydp.algorithms.laplacian.`**`Max`**(*epsilon: float = 1.0*, *delta: float = 0*, *lower_bound: Optional[Union[int, float]] = None*, *upper_bound: Optional[Union[int, float]] = None*, *l0_sensitivity: int = 1*, *linf_sensitivity: int = 1*, *dtype: str = 'int'*)

Max computes the Max value in the dataset, in a differentially private manner.

add_entries(*data: List[Union[int, float]]*) → None

Adds multiple inputs to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current list passed is not added.

add_entry(*value: Union[int, float]*) → None

Adds one input to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current data passed is not added.

property delta: float

Returns the epsilon set at initialization.

property epsilon: float

Returns the epsilon set at initialization.

property l0_sensitivity: float

Returns the l0_sensitivity set at initialization.

property linf_sensitivity: float

Returns the linf_sensitivity set at initialization.

memory_used() → float

Returns the memory currently used by the algorithm in bytes.

merge(*summary*)

Merges serialized summary data into this algorithm. The summary proto must represent data from the same algorithm type with identical parameters. The data field must contain the algorithm summary type of the corresponding algorithm used. The summary proto cannot be empty.

`noise_confidence_interval`(*confidence_level: float*, *privacy_budget: float*) → float

Returns the confidence_level confidence interval of noise added within the algorithm with specified privacy budget, using epsilon and other relevant, algorithm-specific parameters (e.g. bounds) provided by the constructor.

This metric may be used to gauge the error rate introduced by the noise.

If the returned value is <x,y>, then the noise added has a confidence_level chance of being in the domain [x,y].

By default, NoiseConfidenceInterval() returns an error. Algorithms for which a confidence interval can feasibly be calculated override this and output the relevant value.

Conservatively, we do not release the error rate for algorithms whose confidence intervals rely on input size.

`privacy_budget_left`() → float

Returns the remaining privacy budget.

`quick_result`(*data: List[Union[int, float]]*) → Union[int, float]

Runs the algorithm on the input using the epsilon parameter provided in the constructor and returns output.

Consumes 100% of the privacy budget.

Note: It resets the privacy budget first.

`reset`() → None

Resets the algorithm to a state in which it has received no input. After Reset is called, the algorithm should only consider input added after the last Reset call when providing output.

`result`(*privacy_budget: Optional[float] = None*, *noise_interval_level: Optional[float] = None*) → Union[int, float]

Gets the algorithm result.

The default call consumes the remaining privacy budget.

When *privacy_budget* (defined on [0,1]) is set, it consumes only the *privacy_budget* amount of budget.

noise_interval_level provides the confidence level of the noise confidence interval, which may be included in the algorithm output.

`serialize`()

Serializes summary data of current entries into Summary proto. This allows results from distributed aggregation to be recorded and later merged.

Returns empty summary for algorithms for which serialize is unimplemented.

class `pydp.algorithms.laplacian.Min`(*epsilon: float = 1.0*, *delta: float = 0*, *lower_bound: Optional[Union[int, float]] = None*, *upper_bound: Optional[Union[int, float]] = None*, *l0_sensitivity: int = 1*, *linf_sensitivity: int = 1*, *dtype: str = 'int'*)

Min computes the minium value in the dataset, in a differentially private manner.

`add_entries`(*data: List[Union[int, float]]*) → None

Adds multiple inputs to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current list passed is not added.

`add_entry`(*value: Union[int, float]*) → None

Adds one input to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current data passed is not added.

property delta: float

Returns the epsilon set at initialization.

property epsilon: float

Returns the epsilon set at initialization.

property l0_sensitivity: float

Returns the l0_sensitivity set at initialization.

property linf_sensitivity: float

Returns the linf_sensitivity set at initialization.

memory_used() → float

Returns the memory currently used by the algorithm in bytes.

merge(*summary*)

Merges serialized summary data into this algorithm. The summary proto must represent data from the same algorithm type with identical parameters. The data field must contain the algorithm summary type of the corresponding algorithm used. The summary proto cannot be empty.

noise_confidence_interval(*confidence_level: float*, *privacy_budget: float*) → float

Returns the confidence_level confidence interval of noise added within the algorithm with specified privacy budget, using epsilon and other relevant, algorithm-specific parameters (e.g. bounds) provided by the constructor.

This metric may be used to gauge the error rate introduced by the noise.

If the returned value is <x,y>, then the noise added has a confidence_level chance of being in the domain [x,y].

By default, NoiseConfidenceInterval() returns an error. Algorithms for which a confidence interval can feasibly be calculated override this and output the relevant value.

Conservatively, we do not release the error rate for algorithms whose confidence intervals rely on input size.

privacy_budget_left() → float

Returns the remaining privacy budget.

quick_result(*data: List[Union[int, float]]*) → Union[int, float]

Runs the algorithm on the input using the epsilon parameter provided in the constructor and returns output.

Consumes 100% of the privacy budget.

Note: It resets the privacy budget first.

reset() → None

Resets the algorithm to a state in which it has received no input. After Reset is called, the algorithm should only consider input added after the last Reset call when providing output.

result(*privacy_budget: Optional[float] = None*, *noise_interval_level: Optional[float] = None*) → Union[int, float]

Gets the algorithm result.

The default call consumes the remaining privacy budget.

When *privacy_budget* (defined on [0,1]) is set, it consumes only the *privacy_budget* amount of budget.

noise_interval_level provides the confidence level of the noise confidence interval, which may be included in the algorithm output.

serialize()

Serializes summary data of current entries into Summary proto. This allows results from distributed aggregation to be recorded and later merged.

Returns empty summary for algorithms for which serialize is unimplemented.

class pydp.algorithms.laplacian.**Median**(*epsilon: float = 1.0*, *delta: float = 0*, *lower_bound: Optional[Union[int, float]] = None*, *upper_bound: Optional[Union[int, float]] = None*, *l0_sensitivity: int = 1*, *linf_sensitivity: int = 1*, *dtype: str = 'int'*)

Median computes the Median value in the dataset, in a differentially private manner.

add_entries(*data: List[Union[int, float]]*) → None

Adds multiple inputs to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current list passed is not added.

add_entry(*value: Union[int, float]*) → None

Adds one input to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current data passed is not added.

property delta: float

Returns the epsilon set at initialization.

property epsilon: float

Returns the epsilon set at initialization.

property l0_sensitivity: float

Returns the l0_sensitivity set at initialization.

property linf_sensitivity: float

Returns the linf_sensitivity set at initialization.

memory_used() → float

Returns the memory currently used by the algorithm in bytes.

merge(*summary*)

Merges serialized summary data into this algorithm. The summary proto must represent data from the same algorithm type with identical parameters. The data field must contain the algorithm summary type of the corresponding algorithm used. The summary proto cannot be empty.

noise_confidence_interval(*confidence_level: float*, *privacy_budget: float*) → float

Returns the confidence_level confidence interval of noise added within the algorithm with specified privacy budget, using epsilon and other relevant, algorithm-specific parameters (e.g. bounds) provided by the constructor.

This metric may be used to gauge the error rate introduced by the noise.

If the returned value is <x,y>, then the noise added has a confidence_level chance of being in the domain [x,y].

By default, NoiseConfidenceInterval() returns an error. Algorithms for which a confidence interval can feasibly be calculated override this and output the relevant value.

Conservatively, we do not release the error rate for algorithms whose confidence intervals rely on input size.

privacy_budget_left() → float

Returns the remaining privacy budget.

quick_result(*data: List[Union[int, float]]*) → Union[int, float]

Runs the algorithm on the input using the epsilon parameter provided in the constructor and returns output.

Consumes 100% of the privacy budget.

Note: It resets the privacy budget first.

reset() → None

Resets the algorithm to a state in which it has received no input. After Reset is called, the algorithm should only consider input added after the last Reset call when providing output.

result(*privacy_budget: Optional[float] = None*, *noise_interval_level: Optional[float] = None*) → Union[int, float]

Gets the algorithm result.

The default call consumes the remaining privacy budget.

When *privacy_budget* (defined on [0,1]) is set, it consumes only the *privacy_budget* amount of budget.

noise_interval_level provides the confidence level of the noise confidence interval, which may be included in the algorithm output.

serialize()

Serializes summary data of current entries into Summary proto. This allows results from distributed aggregation to be recorded and later merged.

Returns empty summary for algorithms for which serialize is unimplemented.

class pydp.algorithms.laplacian.**Count**(*epsilon: float = 1.0*, *l0_sensitivity: int = 1*, *linf_sensitivity: int = 1*, *dtype: str = 'int'*)

Count computes the Count of number of items in the dataset, in a differentially private manner.

add_entries(*data: List[Union[int, float]]*) → None

Adds multiple inputs to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current list passed is not added.

add_entry(*value: Union[int, float]*) → None

Adds one input to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current data passed is not added.

property delta: float

Returns the epsilon set at initialization.

property epsilon: float

Returns the epsilon set at initialization.

property l0_sensitivity: float

Returns the l0_sensitivity set at initialization.

property linf_sensitivity: float

Returns the linf_sensitivity set at initialization.

memory_used() → float

Returns the memory currently used by the algorithm in bytes.

merge(*summary*)

Merges serialized summary data into this algorithm. The summary proto must represent data from the same algorithm type with identical parameters. The data field must contain the algorithm summary type of the corresponding algorithm used. The summary proto cannot be empty.

noise_confidence_interval(*confidence_level: float*, *privacy_budget: float*) → float

Returns the confidence_level confidence interval of noise added within the algorithm with specified privacy budget, using epsilon and other relevant, algorithm-specific parameters (e.g. bounds) provided by the constructor.

This metric may be used to gauge the error rate introduced by the noise.

If the returned value is <x,y>, then the noise added has a confidence_level chance of being in the domain [x,y].

By default, NoiseConfidenceInterval() returns an error. Algorithms for which a confidence interval can feasibly be calculated override this and output the relevant value.

Conservatively, we do not release the error rate for algorithms whose confidence intervals rely on input size.

privacy_budget_left() → float

Returns the remaining privacy budget.

quick_result(*data: List[Union[int, float]]*) → Union[int, float]

Runs the algorithm on the input using the epsilon parameter provided in the constructor and returns output.

Consumes 100% of the privacy budget.

Note: It resets the privacy budget first.

reset() → None

Resets the algorithm to a state in which it has received no input. After Reset is called, the algorithm should only consider input added after the last Reset call when providing output.

result(*privacy_budget: Optional[float] = None*, *noise_interval_level: Optional[float] = None*) → Union[int, float]

Gets the algorithm result.

The default call consumes the remaining privacy budget.

When *privacy_budget* (defined on [0,1]) is set, it consumes only the *privacy_budget* amount of budget.

noise_interval_level provides the confidence level of the noise confidence interval, which may be included in the algorithm output.

serialize()

Serializes summary data of current entries into Summary proto. This allows results from distributed aggregation to be recorded and later merged.

Returns empty summary for algorithms for which serialize is unimplemented.

class pydp.algorithms.laplacian.**Percentile**(*epsilon: float = 1.0*, *percentile: float = 0.0*, *lower_bound: Optional[Union[int, float]] = None*, *upper_bound: Optional[Union[int, float]] = None*, *dtype: str = 'int'*)

Perencetile finds the value in the dataset with that percentile, in a differentially private manner.

add_entries(*data: List[Union[int, float]]*) → None

Adds multiple inputs to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current list passed is not added.

add_entry(*value: Union[int, float]*) → None

Adds one input to the algorithm.

Note: If the data exceeds the overflow limit of storage, the current data passed is not added.

property delta: float

Returns the epsilon set at initialization.

property epsilon: float

Returns the epsilon set at initialization.

property l0_sensitivity: float

Returns the l0_sensitivity set at initialization.

property linf_sensitivity: float

Returns the linf_sensitivity set at initialization.

memory_used() → float

Returns the memory currently used by the algorithm in bytes.

merge(*summary*)

Merges serialized summary data into this algorithm. The summary proto must represent data from the same algorithm type with identical parameters. The data field must contain the algorithm summary type of the corresponding algorithm used. The summary proto cannot be empty.

noise_confidence_interval(*confidence_level: float*, *privacy_budget: float*) → float

Returns the confidence_level confidence interval of noise added within the algorithm with specified privacy budget, using epsilon and other relevant, algorithm-specific parameters (e.g. bounds) provided by the constructor.

This metric may be used to gauge the error rate introduced by the noise.

If the returned value is <x,y>, then the noise added has a confidence_level chance of being in the domain [x,y].

By default, NoiseConfidenceInterval() returns an error. Algorithms for which a confidence interval can feasibly be calculated override this and output the relevant value.

Conservatively, we do not release the error rate for algorithms whose confidence intervals rely on input size.

property percentile: float

percentile Gets the value that was set in the constructor.

privacy_budget_left() → float

Returns the remaining privacy budget.

quick_result(*data: List[Union[int, float]]*) → Union[int, float]

Runs the algorithm on the input using the epsilon parameter provided in the constructor and returns output.

Consumes 100% of the privacy budget.

Note: It resets the privacy budget first.

reset() → None

Resets the algorithm to a state in which it has received no input. After Reset is called, the algorithm should only consider input added after the last Reset call when providing output.

result(*privacy_budget: Optional[float] = None*, *noise_interval_level: Optional[float] = None*) → Union[int, float]

Gets the algorithm result.

The default call consumes the remaining privacy budget.

When *privacy_budget* (defined on [0,1]) is set, it consumes only the *privacy_budget* amount of budget.

noise_interval_level provides the confidence level of the noise confidence interval, which may be included in the algorithm output.

serialize()

Serializes summary data of current entries into Summary proto. This allows results from distributed aggregation to be recorded and later merged.

Returns empty summary for algorithms for which serialize is unimplemented.

10.2 Numerical Mechanisms

class pydp.algorithms.numerical_mechanisms.**NumericalMechanism**

Base class for all (,)-differenially private additive noise numerical mechanisms.

add_noise(**args*, ***kwargs*)

Overloaded function.

1. add_noise(self: pydp.NumericalMechanism, result: int, privacy_budget: float) -> int
2. add_noise(self: pydp.NumericalMechanism, result: int, privacy_budget: float) -> int
3. add_noise(self: pydp.NumericalMechanism, result: float, privacy_budget: float) -> float
4. add_noise(self: pydp.NumericalMechanism, result: int) -> int
5. add_noise(self: pydp.NumericalMechanism, result: int) -> int
6. add_noise(self: pydp.NumericalMechanism, result: float) -> float

property epsilon

The of the numerical mechanism

memory_used(*self:* pydp.NumericalMechanism) → int

noise_confidence_interval(*self:* pydp.NumericalMechanism, *confidence_level:* *float*, *privacy_budget:* *float*, *noised_result:* *float*) → pydp.ConfidenceInterval

Returns the confidence interval of the specified confidence level of the noise that AddNoise() would add with the specified privacy budget. If the returned value is <x,y>, then the noise added has a confidence_level chance of being in the domain [x,y]

noised_value_above_threshold(*self:* pydp.NumericalMechanism, *arg0:* *float*, *arg1:* *float*) → bool

Quickly determines if *result* with added noise is above certain *threshold*.

class pydp.algorithms.numerical_mechanisms.**LaplaceMechanism**

Bases: `NumericalMechanism`

property diversity

The diversity of the Laplace mechanism.

get_uniform_double(*self:* pydp.LaplaceMechanism) → float

property sensitivity

The L1 sensitivity of the query.

class pydp.algorithms.numerical_mechanisms.**GaussianMechanism**

Bases: `NumericalMechanism`

property delta

The of the Gaussian mechanism.

property l2_sensitivity

The L2 sensitivity of the query.

property std

The standard deviation parameter of the Gaussian mechanism underlying distribution.

10.3 Distributions

`class pydp.distributions.`**`GaussianDistribution`**

`sample`(*self:* pydp.GaussianDistribution, *scale: float = 1.0*) → float

Samples the Gaussian with distribution Gauss(scale*stddev).

scale
A factor to scale stddev.

property stddev

Returns stddev

`class pydp.distributions.`**`LaplaceDistribution`**

Draws samples from the Laplacian distribution.

`get_diversity`(*self:* pydp.LaplaceDistribution) → float

Returns the parameter defining this distribution, often labeled b.

`get_uniform_double`(*self:* pydp.LaplaceDistribution) → float

Returns a uniform random integer of in range [0, 2^53).

`sample`(*self:* pydp.LaplaceDistribution, *scale: float = 1.0*) → float

Samples the Laplacian distribution Laplace(u, scale*b).

Parameters
`scale` – A factor to scale b.

10.4 Util

`pydp.util.`**`Geometric`**`()` → int

`pydp.util.`**`UniformDouble`**`()` → float

`pydp.util.`**`correlation`**(*arg0: List[float]*, *arg1: List[float]*) → float

Returns linear correlation coefficient.

`pydp.util.`**`get_next_power_of_two`**(*arg0: float*) → float

Outputs value of a power of two that is greater than and closest to the given numerical input.

`pydp.util.`**`mean`**(**args*, ***kwargs*)

Overloaded function.

1. mean(arg0: List[float]) -> float

Calculation of the mean of given set of numbers for a double int data type.

2. mean(arg0: List[int]) -> float

Calculation of the mean of given set of numbers for an int data type.

`pydp.util.`**`order_statistics`**(*arg0: float*, *arg1: List[float]*) → float

Sample values placed in ascending order.

`pydp.util.`**`qnorm`**(*arg0: float*, *arg1: float*, *arg2: float*) → pydp._pydp.StatusOrD

Quantile function of normal distribution, inverse of the cumulative distribution function.

`pydp.util.`**`standard_deviation`**(*arg0: List[float]*) → float

Standard Deviation, the square root of variance.

`pydp.util.`**`variance`**(*arg0: List[float]*) → float

Calculate variance for a set of values.

`pydp.util.`**`vector_filter`**(*arg0: List[float]*, *arg1: List[bool]*) → List[float]

Filtering a vector using a logical operatio with only values selected using true output in their positions.

`pydp.util.`**`vector_to_string`**(*arg0: List[float]*) → str

Conversion of a vector to a string data type.

10.5 ML

10.6 Partition Selection

`class` `pydp.algorithms.partition_selection.`**`PartitionSelectionStrategy`**

Base class for all (,)-differenially private partition selection strategies.

`should_keep`(*num_users: int*) → bool

Decides whether or not to keep a partition with *num_users* based on differential privacy parameters and strategy.

`pydp.algorithms.partition_selection.`**`create_partition_strategy`**(*strategy: str*, *epsilon: float*, *delta: float*, *max_partitions_contributed: int*) → *PartitionSelectionStrategy*

Creates a `PartitionSelectionStrategy` instance.

Parameters

- **`strategy`** –

  One of:

  - **’truncated_geomteric’**: creates a Truncated Geometric Partition Strategy.
  - **’laplace’**: creates a private partition strategy with Laplace mechanism.
  - **’gaussian’**: creates a private partition strategy with Gaussian mechanism.

- **`epsilon`** – The ε of the partition mechanism
- **`delta`** – The δ of the partition mechanism
- **`max_partitions_contributed`** – The maximum amount of partitions contributed by the strategy.

PYTHON MODULE INDEX

INDEX