CLK hash Documentation

Release 0.10.1

N1 Analytics

Apr 23, 2018

Contents

1 Table of Contents 3

2 External Links 21

3 Indices and tables 23

Bibliography 25

Python Module Index 27

`clkhash` is a python implementation of cryptographic linkage key hashing as described by Rainer Schnell, Tobias Bachteler, and Jörg Reiher in *A Novel Error-Tolerant Anonymous Linking Code [Schnell2011]*.

Clkhash is Apache 2.0 licensed, supports Python versions 2.7+, 3.4+, and runs on Windows, OSX and Linux.

Install with pip:

```
pip install clkhash
```

Hint: If you are interested in comparing CLKs (i.e carrying out record linkage) you might want to check out anonlink - our Python library for computing similarity scores, and best guess matches between two sets of cryptographic linkage keys.

CHAPTER 1

Table of Contents

1.1 Tutorial

For this tutorial we are going to process a data set for private linkage with clkhash using the Python API. Note you can also use the command line tool.

The Python package `recordlinkage` has a tutorial linking data sets in the clear, we will try duplicate that in a privacy preserving setting.

First install clkhash, recordlinkage and a few data science tools (pandas and numpy).

```
In [ ]: !pip install -U clkhash recordlinkage numpy pandas
```

```
In [24]: import io
         import numpy as np
         import pandas as pd
```

```
In [17]: import clkhash
         import recordlinkage
         from recordlinkage.datasets import load_febrl4
```

1.1.1 Data Exploration

First we have a look at the dataset.

```
In [4]: dfA, dfB = load_febrl4()

        dfA.head()
```

```
Out[4]: given_name   surname street_number            address_1  \
        rec_id
        rec-1070-org   michaela   neumann             8       stanley street
        rec-1016-org   courtney   painter            12    pinkerton circuit
        rec-4405-org    charles     green            38  salkauskas crescent
        rec-1288-org    vanessa      parr           905       macquoid place
        rec-3585-org    mikayla  malloney            37        randwick road
```

```
                        address_2            suburb postcode state  \
rec_id
rec-1070-org                miami     winston hills     4223   nsw
rec-1016-org          bega flats         richlands     4560   vic
rec-4405-org                kela             dapto     4566   nsw
rec-1288-org  broadbridge manor     south grafton     2135    sa
rec-3585-org             avalind  hoppers crossing     4552   vic

              date_of_birth soc_sec_id
rec_id
rec-1070-org       19151111    5304218
rec-1016-org       19161214    4066625
rec-4405-org       19480930    4365168
rec-1288-org       19951119    9239102
rec-3585-org       19860208    7207688
```

For this linkage we will **not** use the social security id column.

```
In [18]: dfA.columns
```

```
Out[18]: Index(['given_name', 'surname', 'street_number', 'address_1', 'address_2',
                'suburb', 'postcode', 'state', 'date_of_birth', 'soc_sec_id'],
               dtype='object')
```

```
In [41]: a_csv = io.StringIO()
         dfA.to_csv(a_csv)
         a_csv.seek(0)
```

```
Out[41]: 0
```

1.1.2 Linkage Schema Definition

A basic schema definition instructs clkhash how to treat each column. The identifiers are found in clkhash/identifier_types.py. The `INDEX` columns will not become part of the CLK.

Note: The schema specification is under heavy renovation for the next version.

```
In [42]: column_metadata = [
                     {"identifier": 'INDEX'},
                     {"identifier": 'NAME Surname'},
                     {"identifier": 'NAME First Name'},
                     {"identifier": 'ADDRESS House Number'},
                     {"identifier": 'ADDRESS Place Name'},
                     {"identifier": 'ADDRESS Place Name'},
                     {"identifier": 'ADDRESS Place Name'},
                     {"identifier": 'ADDRESS POSTCODE'},
                     {"identifier": 'ADDRESS Place Name'},
                     {"identifier": 'DOB YYYY/MM/DD'},
                     {"identifier": 'INDEX'}
                 ]
```

```
In [43]: schema = clkhash.schema.get_schema_types(column_metadata)
```

1.1.3 Hash the data

We can now hash our PII data from the CSV file using our defined schema. We must provide two *secret keys* to this command - these keys have to be used by both parties hashing data.

Knowledge of these keys is sufficient to reconstruct the PII information from a CLK! **Do not share these keys with anyone, except the other participating party.**

```
In [44]: hashed_data_a = clkhash.clk.generate_clk_from_csv(a_csv, ('key1', 'key2'), schema)
```

```
generating CLKs: 100%|| 5.00K/5.00K [00:02<00:00, 1.46Kclk/s, mean=916, std=27.1]
```

1.1.4 Inspect the output

clkhash has hashed the PII, creating a Cryptographic Longterm Key for each entity. The output of `generate_clk_from_csv` shows that the mean popcount is quite high (916 out of 1024) which can effect accuracy.

```
In [45]: len(hashed_data_a)
```

```
Out[45]: 5000
```

Each CLK is serialized in a JSON friendly base64 format:

```
In [46]: hashed_data_a[0]
```

```
Out[46]: '+79/+3//33/+O///fv//N67/7/+/u/vP////+3/+/7//d///////ft/////vM/7v/v//1/+///9/fv9f7/zu//5/+/
```

1.1.5 Hash data set B

Now we hash the second dataset using the same keys and same schema.

```
In [47]: b_csv = io.StringIO()
         dfB.to_csv(b_csv)
         b_csv.seek(0)
         hashed_data_b = clkhash.clk.generate_clk_from_csv(b_csv, ('key1', 'key2'), schema)
```

```
generating CLKs: 100%|| 5.00K/5.00K [00:02<00:00, 2.44Kclk/s, mean=909, std=32]
```

```
In [48]: len(hashed_data_b)
```

```
Out[48]: 5000
```

1.1.6 Wrapping Up

That is all for this tutorial, next you might want to look at comparing the CLKs with anonlink, or uploading them to an Entity Service.

Note the clkhash command line tool includes commands to upload to an entity service run by Data61.

1.2 Command Line Tool

This command line tool can be used to process PII data into Cryptographic Longterm Keys.

The command line tool can be accessed in two ways:

- Using the `clkutil` script which should have been added to your path during installation.
- directly running the python module `clkhash.cli` with `python -m clkhash.cli`.

1.2.1 Hashing

The command line tool `clkutil` can be used to hash a csv file of personally identifiable information. The tool needs to be provided with keys and a *Hashing Schema*; it will output a file containing json serialized hashes.

Example

Assume a csv (`fake-pii.csv`) contains rows like the following:

```
0,Libby Slemmer,1933/09/13,F
1,Garold Staten,1928/11/23,M
2,Yaritza Edman,1972/11/30,F
```

It can be hashed using `clkutil` with:

```
$ clkutil hash --schema simple-schema.json fake-pii.csv horse staple clk.json
```

Where:

- `horse staple` is the two part secret key that both participants will use to hash their data.
- `simple-schema.json` is a *Hashing Schema* describing how to hash the csv. E.g, ignore the first column, use bigram tokens of the name, use positional unigrams of the date of birth etc.
- `clk.json` is the output file.

1.2.2 Data Generation

The cli tool has an option for generating fake pii data.

```
$ clkutil generate 1000 fake-pii-out.csv
$ head -n 4  fake-pii-out.csv
INDEX,NAME freetext,DOB YYYY/MM/DD,GENDER M or F
0,Libby Slemmer,1933/09/13,F
1,Garold Staten,1928/11/23,M
2,Yaritza Edman,1972/11/30,F
```

The yaml schema used for the generated data is the following:

```
- identifier: "INDEX"
  notes: "Ignored"
- identifier: "NAME freetext"
- identifier: "DOB YYYY/MM/DD"
- identifier: "GENDER M or F"
```

1.2.3 Benchmark

A quick hashing benchmark can be carried out to determine the rate at which the current machine can generate 10000 clks from a simple schema (data as generated *above*):

```
python -m clkhash.cli benchmark
generating CLKs: 100%                 10.0K/10.0K [00:01<00:00, 7.72Kclk/s, mean=521,␣
↪std=34.7]
 10000 hashes in 1.350489 seconds. 7.40 KH/s
```

As a rule of thumb a single modern core will hash around 1M entities in about 20 minutes.

Note: Hashing speed is effected by the number of features and the corresponding schema. Thus these numbers will, in general, not be a good predictor for the performance of a specific use-case.

The output shows a running mean and std deviation of the generated clks' popcounts. This can be used as a basic sanity check - ensure the CLK's popcount is not around 0 or 1024.

1.2.4 Interaction with Entity Service

There are several commands that interact with a REST api for carrying out privacy preserving linking. These commands are:

- status
- create
- upload
- results

1.3 Hashing Schema

Caution: This document and the referenced `schema.json` file are still in a draft status.

As CLKs are usually used for privacy preserving linkage, it is important, that participating organisations agree on how raw personally identifiable information is hashed to create the CLKs.

We call the configuration of how to create CLKs a *hashing schema*. The organisations agree on one hashing schema as configuration to ensure that the have been created in the same way.

This aims to be an open standard such that different client implementations could take the schema and create identical CLKS given the same data.

The hashing-schema is a detailed description of exactly what is fed to the hashing operation, along with any configuration for the hashing itself.

The format of the hashing schema is defined in a separate `JSON Schema` document hashing-schema.json.

1.3.1 Basic Structure

A hashing schema consists of three parts:

- *version*, contains the version number of the hashing schema
- *clkConfig*, CLK wide configuration, independent of features
- *features*, configuration that is specific to the individual features

1.3.2 Example Schema

```
{
  "version": 1,
  "clkConfig": {
    "l": 1024,
    "k": 20,
    "hash": {
      "type": "doubleHash"
    },
    "kdf": {
      "type": "HKDF"
    }
  },
  "features": [
    {
      "identifier": "full name",
      "format": {
        "type": "string",
        "maxLength": 30,
        "encoding": "utf-8"
      },
      "hashing": { "ngram": 2 }
    },
    {
      "identifier": "gender",
      "format": {
        "type": "enum",
        "values": ["M", "F", "O"]
      },
      "hashing": { "ngram": 1 }
    },
    {
      "identifier": "postcode",
      "format": {
        "type": "integer",
        "minimum": 1000,
        "maximum": 9999
      },
      "hashing": { "ngram": 1, "positional": true }
    }
  ]
}
```

A more advanced example can be found here.

1.3.3 Schema Components

Version

Integer value which describes the version of the hashing schema.

clkConfig

Describes the general construction of the CLK.

name	type	optional	description
l	integer	no	the length of the CLK in bits
k	integer	no	max number of indices per n-gram
xorFolds	integer	yes	number of XOR folds (as proposed in *[Schnell2016]*).
kdf	*KDF*	no	defines the key derivation function used to generate individual secrets for each feature derived from the master secret
hash	*Hash*	no	defines the hashing scheme to encode the n-grams

KDF

We currently only support HKDF (for a basic description, see https://en.wikipedia.org/wiki/HKDF).

name	type	optional	description
type	string	no	must be set to "HKDF"
hash	enum	yes	hash function used by HKDF, either "SHA256" or "SHA512"
salt	string	yes	base64 encoded bytes
info	string	yes	base64 encoded bytes
keySize	integer	yes	size of the generated keys in bytes

Hash

Describes and cofigures the hash that is used to encode the n-grams.

Choose one of:

- *double hash*, as described in *[Schnell2011]*.

name	type	optional	description
type	string	no	must be set to "doubleHash"
prevent_singularity	boolean	yes	see discussion in https://github.com/n1analytics/clkhash/issues/33

- *blake hash*

name	type	optional	description
type	string	no	must be set to "blakeHash"

featureConfig

A feature is configured in three parts:

- identifier, the name of the feature
- format, describes the expected format of the values of this feature
- hashing, configures the hashing

name	type	optional	description
identifier	string	no	the name of the feature
description	string	yes	free text, ignored by clkhash
hashing	*hashingConfig*	no	configures feature specific hashing parameters
format	one of: *textFormat*, *textPatternFormat*, *numberFormat*, *dateFormat*, *enumFormat*	no	describes the expected format of the feature values

hashingConfig

name	type	optional	description
ngram	integer	no	specifies the n in n-gram (the tokenization of the input values).
positional	boolean	yes	adds the position to the n-grams. String "222" would be tokenized (as uni-grams) to "1 2", "2 2", "3 2"
weight	float	yes	positive number, which adjusts the number of hash functions (k) used for encoding. Thus giving this feature more or less importance compared to others.

textFormat

name	type	optional	description
type	string	no	has to be "string"
encoding	enum	yes	one of "ascii", "utf-8", "utf-16", "utf-32". Default is "utf-8".
case	enum	yes	one of "upper", "lower", "mixed".
minLength	integer	yes	positive integer describing the minimum length of the input string.
maxLength	integer	yes	positive integer describing the maximum length of the input string.
description	string	yes	free text, ignored by clkhash.

textPatternFormat

name	type	optional	description
type	string	no	has to be "string"
encoding	enum	yes	one of "ascii", "utf-8", "utf-16", "utf-32". Default is "utf-8".
pattern	string	no	a regular expression describing the input format.
description	string	yes	free text, ignored by clkhash.

numberFormat

name	type	optional	description
type	string	no	has to be “integer”
minimum	integer	yes	positive integer describing the lower bound of the input values.
maximum	integer	yes	positive integer describing the upper bound of the input values.
description	string	yes	free text, ignored by clkhash.

dateFormat

name	type	optional	description
type	string	no	has to be “date”
format	enum	no	one of [“rfc3339”]. That’s the standard internet format: yyyy-mm-dd.
description	string	yes	free text, ignored by clkhash.

enumFormat

name	type	optional	description
type	string	no	has to be “enum”
values	array	no	an array of items of type “string”
description	string	yes	free text, ignored by clkhash.

1.4 Development

1.4.1 API Documentation

Bloom filter

Generate a Bloom filter

class clkhash.bloomfilter.**NgramEncodings**

Bases: enum.Enum

Lists the available schemes for encoding n-grams.

Note: the slightly awkward looking construction with the calls to partial and the overwrite of __call__ are due to compatibility issues with Python 2.7.

BLAKE_HASH = functools.partial(<function blake_encode_ngrams>)

uses the BLAKE2 hash function, which is one of the fastest modern hash functions, and does less hash function calls compared to the DOUBLE_HASH based schemes. It avoids one of the exploitable weaknesses of the DOUBLE_HASH scheme. Also see *blake_encode_ngrams()*

DOUBLE_HASH = functools.partial(<function double_hash_encode_ngrams>)

the initial encoding scheme as described in Schnell, R., Bachteler, T., & Reiher, J. (2011). A Novel Error-Tolerant Anonymous Linking Code. Also see *double_hash_encode_ngrams()*

DOUBLE_HASH_NON_SINGULAR = functools.partial(<function double_hash_encode_ngrams_non_s
very similar to DOUBLE_HASH, but avoids singularities in the encoding. Also see *double_hash_encode_ngrams_non_singular()*

clkhash.bloomfilter.**blake_encode_ngrams**(*ngrams*, *key*, *k*, *l*)
Computes the encoding of the provided ngrams using the BLAKE2 hash function.

We deliberately do not use the double hashing scheme as proposed in *[Schnell2011]*, because this would introduce an exploitable structure into the Bloom filter. For more details on the weakness, see *[Kroll2015]*.

In short, the double hashing scheme only allows for l^2 different encodings for any possible n-gram, whereas the use of k different independent hash functions gives you $\sum_{j=1}^{k} \binom{l}{j}$ combinations.

Our construction

It is advantageous to construct Bloom filters using a family of hash functions with the property of k-independence to compute the indices for an entry. This approach minimises the change of collisions.

An informal definition of *k-independence* of a family of hash functions is, that if selecting a function at random from the family, it guarantees that the hash codes of any designated k keys are independent random variables.

Our construction utilises the fact that the output bits of a cryptographic hash function are uniformly distributed, independent, binary random variables (well, at least as close to as possible. See *[Kaminsky2011]* for an analysis). Thus, slicing the output of a cryptographic hash function into k different slices gives you k independent random variables.

We chose Blake2 as the cryptographic hash function mainly for two reasons:

- it is fast.
- in keyed hashing mode, Blake2 provides MACs with just one hash function call instead of the two calls in the HMAC construction used in the double hashing scheme.

Warning: Please be aware that, although this construction makes the attack of *[Kroll2015]* infeasible, it is most likely not enough to ensure security. Or in their own words:

> However, we think that using independent hash functions alone will not be sufficient to ensure security, since in this case other approaches (maybe related to or at least inspired through work from the area of Frequent Itemset Mining) are promising to detect at least the most frequent atoms automatically.

Parameters

- **ngrams** – list of n-grams to be encoded
- **key** – secret key for blake2 as bytes
- **k** – number of hash functions to use per element of the ngrams
- **l** – length of the output bitarray (has to be a power of 2)

Returns bitarray of length l with the bits set which correspond to the encoding of the ngrams

clkhash.bloomfilter.**calculate_bloom_filters**(*dataset*, *schema*, *keys*, *xor_folds=0*)

Parameters

- **dataset** – A list of indexable records.
- **schema** – An iterable of identifier types.
- **keys** – A tuple of two lists of secret keys used in the HMAC.

- **xor_folds** – number of XOR folds to perform

Returns List of bloom filters as 3-tuples, each containing bloom filter (bitarray), record first element - usually index, bitcount (int)

clkhash.bloomfilter.**crypto_bloom_filter**(*record*, *tokenizers*, *keys*, *xor_folds=0*, *l=1024*, *k=30*, *ngram_encoding=<NgramEncodings.DOUBLE_HASH: functools.partial(<function double_hash_encode_ngrams>)>*)

Makes a Bloom filter from a record with given tokenizers and lists of keys.

Using the method from http://www.record-linkage.de/-download=wp-grlc-2011-02.pdf

Parameters

- **record** – plaintext record tuple. E.g. (index, name, dob, gender)
- **tokenizers** – A list of IdentifierType tokenizers (one for each record element)
- **keys** – tuple of tuple of keys for the hash functions as bytes
- **xor_folds** – number of XOR folds to perform
- **l** – length of the Bloom filter in number of bits
- **k** – number of hash functions to use per element
- **ngram_encoding** –

Returns 3-tuple: - bloom filter for record as a bitarray - first element of record (usually an index) - number of bits set in the bloomfilter

clkhash.bloomfilter.**double_hash_encode_ngrams**(*ngrams*, *keys*, *k*, *l*)

Computes the double hash encoding of the provided ngrams with the given keys.

Using the method from http://www.record-linkage.de/-download=wp-grlc-2011-02.pdf

Parameters

- **ngrams** – list of n-grams to be encoded
- **keys** – hmac secret keys for md5 and sha1 as bytes
- **k** – number of hash functions to use per element of the ngrams
- **l** – length of the output bitarray

Returns bitarray of length l with the bits set which correspond to the encoding of the ngrams

clkhash.bloomfilter.**double_hash_encode_ngrams_non_singular**(*ngrams*, *keys*, *k*, *l*)

computes the double hash encoding of the provided n-grams with the given keys.

The original construction of *[Schnell2011]* displays an abnormality for certain inputs: An n-gram can be encoded into just one bit irrespective of the number of k.

Their construction goes as follows: the k different indices g_i of the Bloom filter for an n-gram x are defined as:

$$g_i(x) = (h_1(x) + ih_2(x)) \mod l$$

with $0 \leq i < k$ and l is the length of the Bloom filter. If the value of the hash of x of the second hash function is a multiple of l, then

$$h_2(x) = 0 \mod l$$

and thus

$$g_i(x) = h_1(x) \mod l,$$

irrespective of the value i. A discussion of this potential flaw can be found here.

Parameters

- **ngrams** – list of n-grams to be encoded
- **key_sha1** – hmac secret keys for sha1 as bytes
- **key_md5** – hmac secret keys for md5 as bytes
- **k** – number of hash functions to use per element of the ngrams
- **l** – length of the output bitarray

Returns bitarray of length l with the bits set which correspond to the encoding of the ngrams

clkhash.bloomfilter.**fold_xor**(*bloomfilter*, *folds*)
Performs XOR folding on a Bloom filter.

If the length of the original Bloom filter is n and we perform r folds, then the length of the resulting filter is n / 2 ** r.

Parameters

- **bloomfilter** – Bloom filter to fold
- **folds** – number of folds

Returns folded bloom filter

clkhash.bloomfilter.**serialize_bitarray**(*ba*)
Serialize a bitarray (bloomfilter)

clkhash.bloomfilter.**stream_bloom_filters**(*dataset*, *schema_types*, *keys*, *xor_folds=0*)
Yield bloom filters

Parameters

- **dataset** – An iterable of indexable records.
- **schema_types** – An iterable of identifier type names.
- **keys** – A tuple of two lists of secret keys used in the HMAC.
- **xor_folds** – number of XOR folds to perform

Returns Yields bloom filters as 3-tuples

CLK

Generate CLK from CSV file

clkhash.clk.**chunks**(*l*, *n*)
Yield successive n-sized chunks from l.

clkhash.clk.**generate_clk_from_csv**(*input*, *keys*, *schema_types*, *no_header=False*, *progress_bar=True*, *xor_folds=0*)

clkhash.clk.**generate_clks**(*pii_data*, *schema_types*, *key_lists*, *xor_folds*, *callback=None*)

clkhash.clk.**hash_and_serialize_chunk**(*chunk_pii_data*, *schema_types*, *keys*, *xor_folds*)
Generate Bloom filters (ie hash) from chunks of PII then serialize the generated Bloom filters. It also computes and outputs the Hamming weight (or popcount) – the number of bits set to one – of the generated Bloom filters.

Parameters

- **chunk_pii_data** – An iterable of indexable records.
- **schema_types** – An iterable of identifier type names.
- **keys** – A tuple of two lists of secret keys used in the HMAC.
- **xor_folds** – Number of XOR folds to perform. Each fold halves the hash length.

Returns A list of serialized Bloom filters and a list of corresponding popcounts

identifier types

Convert PII to tokens

clkhash.identifier_types.**identifier_type_from_description**(*schema_object*)
Convert a dictionary describing a feature into an IdentifierType

Parameters **schema_object** –

Returns An IdentifierType

IdentifierType

class clkhash.identifier_types.**IdentifierType**(*unigram=False*, *weight=1*, ***kwargs*)
Bases: object

Base class used for all identifier types.

Required to provide a mapping of schema to hash type uni-gram or bi-gram.

__call__(*entry*)
Call self as a function.

__init__(*unigram=False*, *weight=1*, ***kwargs*)

Parameters

- **unigram** (*bool*) – Use uni-gram instead of using bi-grams
- **weight** (*float*) – adjusts the "importance" of this identifier in the Bloom filter. Can be set to zero to skip
- **kwargs** – Extra keyword arguments passed to the tokenizer

---

Note: For each n-gram of an identifier, we compute k different indices in the Bloom filter which will be set to true. There is a global $k_{default}$ value, and the k value for each identifier is computed as

$$k = weight * k_{default},$$

rounded to the nearest integer.

Reasons why you might want to set weights:

- Long identifiers like street name will produce a lot more n-grams than small identifiers like zip code. Thus street name will flip more bits in the Bloom filter and will have a bigger influence in the overall matching score.

- The matching might produce better results if identifiers that are stable and / or have low error rates are given higher prominence in the Bloom filter.

__weakref__
list of weak references to the object (if defined)

key derivation

class clkhash.key_derivation.**HKDFconfig**(*master_secret*, *salt=None*, *info=None*, *hash_algo='SHA256'*)

Bases: object

static check_is_bytes(*value*)

static check_is_bytes_or_none(*value*)

supported_hash_algos = ('SHA256', 'SHA512')

clkhash.key_derivation.**generate_key_lists**(*master_secrets*, *num_identifier*, *key_size=64*, *salt=None*, *info=None*, *kdf='HKDF'*)

Generates a derived key for each identifier for each master secret using a key derivation function (KDF).

The only supported key derivation function for now is 'HKDF'.

The previous key usage can be reproduced by setting kdf to 'legacy'. This is highly discouraged, as this strategy will map the same n-grams in different identifier to the same bits in the Bloom filter and thus does not lead to good results.

Parameters

- **master_secrets** – a list of master secrets (either as bytes or strings)
- **num_identifier** – the number of identifiers
- **key_size** – the size of the derived keys
- **salt** – salt for the KDF as bytes
- **info** – optional context and application specific information as bytes
- **kdf** – the key derivation function algorithm to use

Returns The derived keys. First dimension is of size num_identifier, second dimension is the same as master_secrets. A key is represented as bytes.

clkhash.key_derivation.**hkdf**(*hkdf_config*, *num_keys*, *key_size=64*)

Executes the HKDF key derivation function as described in rfc5869 to derive *num_keys* keys of size *key_size* from the master_secret.

Parameters

- **hkdf_config** – an HKDFconfig object containing the configuration for the HKDF.
- **num_keys** – the number of keys the kdf should produce
- **key_size** – the size of the produced keys

Returns Derived keys

random names

Module to produce a dataset of names, genders and dates of birth and manipulate that list

Currently very simple and not realistic. Additional functions for manipulating the list of names - producing reordered and subset lists with a specific overlap

ClassList class - generate a list of length n of [id, name, dob, gender] lists

TODO: Get age distribution right by using a mortality table TODO: Get first name distributions right by using distributions TODO: Generate realistic errors TODO: Add RESTfull api to generate reasonable name data as requested

class clkhash.randomnames.**NameList**(*n*)

Bases: object

List of randomly generated names

generate_random_person(*n*)

Generator that yields details on a person with plausible name, sex and age.

Yields Generated data for one person tuple - (id: int, name: str('First Last'), birthdate: str('DD/MM/YYYY'), sex: str('M' | 'F'))

generate_subsets(*sz*, *overlap=0.8*)

Return a pair of random subsets of the name list with a specified proportion of elements in common.

Parameters

- **sz** – length of subsets to generate
- **overlap** – fraction of the subsets that should have the same names in them

Raises **ValueError** – if there aren't sufficiently many names in the list to satisfy the request; more precisely, raises if sz + (1 - overlap)*sz > n = len(self.names)

Returns pair of subsets

load_names()

This function loads a name database into globals firstNames and lastNames

initial version uses data files from http://www.quietaffiliate.com/free-first-name-and-last-name-databases-csv-and-sql/

schema = [{'identifier': 'INDEX'}, {'identifier': 'NAME freetext'}, {'identifier':

schema_types

clkhash.randomnames.**load_csv_data**(*resource_name*)

Loads a specified CSV data file and returns the first column as a Python list

clkhash.randomnames.**random_date**(*start*, *end*)

This function will return a random datetime between two datetime objects.

Parameters

- **start** – datetime of start
- **end** – datetime of end

Returns random datetime between start and end

clkhash.randomnames.**save_csv**(*data*, *schema*, *file*)

Output generated data to file as CSV with header.

Parameters

- **data** – An iterable of tuples containing raw data.
- **schema** – Iterable of schema definition dicts
- **file** – A writeable stream in which to write the CSV

schema

clkhash.schema.**get_schema_types**(*schema*)

clkhash.schema.**load_schema**(*schema_file*)

tokenizer

Functions to tokenize words (PII)

clkhash.tokenizer.**bigramlist**(*word*, *toremove=None*)
Make bigrams from word with pre- and ap-pended spaces

s -> [‘ ‘ + s0, s0 + s1, s1 + s2, .. sN + ‘ ‘]

Parameters

- **word** – string to make bigrams from
- **toremove** – List of strings to remove before construction

Returns list of bigrams as strings

clkhash.tokenizer.**positional_unigrams**(*instr*)
Make positional unigrams from a word.

E.g. 1987 -> [“1 1”, “2 9”, “3 8”, “4 7”]

Parameters **instr** – input string

Returns list of strings with unigrams

clkhash.tokenizer.**unigramlist**(*instr*, *toremove=None*, *positional=False*)
Make 1-grams (unigrams) from a word, possibly excluding particular substrings

Parameters

- **instr** – input string
- **toremove** – Iterable of strings to remove

Returns list of strings with unigrams

1.4.2 Testing

Make sure you have all the required modules before running the tests (modules that are only needed for tests are not included during installation):

```
$ pip install -r requirements.txt
```

Now run the unit tests and print out code coverage with *py.test*:

```
$ python -m pytest --cov=clkhash
```

Note several tests will be skipped by default. To enable the command line tests set the *INCLUDE_CLI* environment variable. To enable the tests which interact with an entity service set the *TEST_ENTITY_SERVICE* environment variable to the target service's address:

```
$ TEST_ENTITY_SERVICE= INCLUDE_CLI= python -m pytest --cov=clkhash
```

1.4.3 Type Checking

`clkhash` uses static typechecking with `mypy`. To run the type checker (in Python 3.5 or later):

```
$ pip install mypy
$ mypy clkhash --ignore-missing-imports --strict-optional --no-implicit-optional --
↪disallow-untyped-calls
```

1.5 References

CHAPTER 2

External Links

- clkhash on Github
- clkhash on PyPi

CHAPTER 3

Indices and tables

- genindex
- modindex

Bibliography

[Schnell2011] Schnell, R., Bachteler, T., & Reiher, J. (2011). A Novel Error-Tolerant Anonymous Linking Code.

[Schnell2016] Schnell, R., & Borgs, C. (2016). XOR-Folding for hardening Bloom Filter-based Encryptions for Privacy-preserving Record Linkage.

[Kroll2015] Kroll, M., & Steinmetzer, S. (2015). Who is 1011011111…1110110010? automated cryptanalysis of bloom filter encryptions of databases with several personal identifiers. In Communications in Computer and Information Science. https://doi.org/10.1007/978-3-319-27707-3_21

[Kaminsky2011] Kaminsky, A. (2011). GPU Parallel Statistical and Cube Test Analysis of the SHA-3 Finalist Candidate Hash Functions.

Python Module Index

C

clkhash.bloomfilter, 11
clkhash.clk, 14
clkhash.identifier_types, 15
clkhash.key_derivation, 16
clkhash.randomnames, 17
clkhash.schema, 18
clkhash.tokenizer, 18

Index

Symbols

__call__() (clkhash.identifier_types.IdentifierType method), 15
__init__() (clkhash.identifier_types.IdentifierType method), 15
__weakref__ (clkhash.identifier_types.IdentifierType attribute), 16

B

bigramlist() (in module clkhash.tokenizer), 18
blake_encode_ngrams() (in module clkhash.bloomfilter), 12
BLAKE_HASH (clkhash.bloomfilter.NgramEncodings attribute), 11

C

calculate_bloom_filters() (in module clkhash.bloomfilter), 12
check_is_bytes() (clkhash.key_derivation.HKDFconfig static method), 16
check_is_bytes_or_none() (clkhash.key_derivation.HKDFconfig static method), 16
chunks() (in module clkhash.clk), 14
clkhash.bloomfilter (module), 11
clkhash.clk (module), 14
clkhash.identifier_types (module), 15
clkhash.key_derivation (module), 16
clkhash.randomnames (module), 17
clkhash.schema (module), 18
clkhash.tokenizer (module), 18
crypto_bloom_filter() (in module clkhash.bloomfilter), 13

D

DOUBLE_HASH (clkhash.bloomfilter.NgramEncodings attribute), 11
double_hash_encode_ngrams() (in module clkhash.bloomfilter), 13
double_hash_encode_ngrams_non_singular() (in module clkhash.bloomfilter), 13
DOUBLE_HASH_NON_SINGULAR (clkhash.bloomfilter.NgramEncodings attribute), 11

F

fold_xor() (in module clkhash.bloomfilter), 14

G

generate_clk_from_csv() (in module clkhash.clk), 14
generate_clks() (in module clkhash.clk), 14
generate_key_lists() (in module clkhash.key_derivation), 16
generate_random_person() (clkhash.randomnames.NameList method), 17
generate_subsets() (clkhash.randomnames.NameList method), 17
get_schema_types() (in module clkhash.schema), 18

H

hash_and_serialize_chunk() (in module clkhash.clk), 14
hkdf() (in module clkhash.key_derivation), 16
HKDFconfig (class in clkhash.key_derivation), 16

I

identifier_type_from_description() (in module clkhash.identifier_types), 15
IdentifierType (class in clkhash.identifier_types), 15

L

load_csv_data() (in module clkhash.randomnames), 17
load_names() (clkhash.randomnames.NameList method), 17
load_schema() (in module clkhash.schema), 18

N

NameList (class in clkhash.randomnames), 17

NgramEncodings (class in clkhash.bloomfilter), 11

P

positional_unigrams() (in module clkhash.tokenizer), 18

R

random_date() (in module clkhash.randomnames), 17

S

save_csv() (in module clkhash.randomnames), 17
schema (clkhash.randomnames.NameList attribute), 17
schema_types (clkhash.randomnames.NameList attribute), 17
serialize_bitarray() (in module clkhash.bloomfilter), 14
stream_bloom_filters() (in module clkhash.bloomfilter), 14
supported_hash_algos (clkhash.key_derivation.HKDFconfig attribute), 16

U

unigramlist() (in module clkhash.tokenizer), 18