CLK hash Documentation

Release 0.12.0

N1 Analytics

Sep 24, 2018

Contents

`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 Tutorials

The clkhash library can be used via the Python API or the command line tool *clkutil*.

1.1.1 Tutorial for Python API

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 [1]: import io
        import numpy as np
        import pandas as pd
```

```
In [2]: import clkhash
        from clkhash.field_formats import *
        import recordlinkage
        from recordlinkage.datasets import load_febrl4
```

Data Exploration

First we have a look at the dataset.

```
In [3]: dfA, dfB = load_febrl4()

        dfA.head()
```

```
Out[3]:              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 [4]: dfA.columns
```

```
Out[4]: Index(['given_name', 'surname', 'street_number', 'address_1', 'address_2',
               'suburb', 'postcode', 'state', 'date_of_birth', 'soc_sec_id'],
              dtype='object')
```

```
In [5]: a_csv = io.StringIO()
        dfA.to_csv(a_csv)
        a_csv.seek(0)
```

```
Out[5]: 0
```

Hashing Schema Definition

A hashing schema instructs clkhash how to treat each column for generating CLKs. A detailed description of the hashing schema can be found in the api docs. We will ignore the columns 'rec_id' and 'soc_sec_id' for CLK generation.

```
In [6]: schema = clkhash.randomnames.NameList.SCHEMA
        schema.fields = [
            Ignore('rec_id'),
            StringSpec('given_name', FieldHashingProperties(ngram=2, weight=1)),
            StringSpec('surname', FieldHashingProperties(ngram=2, weight=1)),
            IntegerSpec('street_number', FieldHashingProperties(ngram=1, positional=True, weight=1, r
            StringSpec('address_1', FieldHashingProperties(ngram=2, weight=1)),
            StringSpec('address_2', FieldHashingProperties(ngram=2, weight=1)),
            StringSpec('suburb', FieldHashingProperties(ngram=2, weight=1)),
            IntegerSpec('postcode', FieldHashingProperties(ngram=1, positional=True, weight=1)),
            StringSpec('state', FieldHashingProperties(ngram=2, weight=1)),
            IntegerSpec('date_of_birth', FieldHashingProperties(ngram=1, positional=True, weight=1, r
            Ignore('soc_sec_id')
            ]
```

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. For this toy example we will use the keys *'key1'* and *'key2'*, for real data, make sure that the keys contain enough entropy, as knowledge of these keys is sufficient to reconstruct the PII information from a CLK! Also, **do not share these keys with anyone, except the other participating party.**

```
In [7]: from clkhash import clk
        hashed_data_a = clk.generate_clk_from_csv(a_csv, ('key1', 'key2'), schema, validate=False)
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:05<00:00, 734clk/s, mean=885, std=33.4]
```

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 (885 out of 1024) which can effect accuracy.

There are two ways to control the popcount: - You can change the *'k'* value in the hashConfig section of the schema. It controls the number of entries in the CLK for each n-gram - or you can modify the individual *'weight'* values for the different fields. It allows to tune the contribution of a column to the CLK. This can be used to de-emphasise columns which are less suitable for linkage (e.g. information that changes frequently).

First, we will change the value of *k* from 30 to 15.

```
In [8]: schema.hashing_globals.k = 15
        a_csv.seek(0)
        hashed_data_a = clk.generate_clk_from_csv(a_csv, ('key1', 'key2'), schema, validate=False)
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:04<00:00, 930clk/s, mean=648, std=44.1]
```

And now we will modify the weights to de-emphasise the contribution of the address related columns.

```
In [10]: schema.hashing_globals.k = 20
         schema.fields = [
             Ignore('rec_id'),
             StringSpec('given_name', FieldHashingProperties(ngram=2, weight=1)),
             StringSpec('surname', FieldHashingProperties(ngram=2, weight=1)),
             IntegerSpec('street_number', FieldHashingProperties(ngram=1, positional=True, weight=0.5
             StringSpec('address_1', FieldHashingProperties(ngram=2, weight=0.5)),
             StringSpec('address_2', FieldHashingProperties(ngram=2, weight=0.5)),
             StringSpec('suburb', FieldHashingProperties(ngram=2, weight=0.5)),
             IntegerSpec('postcode', FieldHashingProperties(ngram=1, positional=True, weight=0.5)),
             StringSpec('state', FieldHashingProperties(ngram=2, weight=0.5)),
             IntegerSpec('date_of_birth', FieldHashingProperties(ngram=1, positional=True, weight=1,
             Ignore('soc_sec_id')
             ]
         a_csv.seek(0)
         hashed_data_a = clk.generate_clk_from_csv(a_csv, ('key1', 'key2'), schema)
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:04<00:00, 917clk/s, mean=602, std=39.8]
```

Each CLK is serialized in a JSON friendly base64 format:

```
In [11]: hashed_data_a[0]
```

```
Out[11]: 'BD8JWW7DzwP82PjV5/jbN40+bT3V4z7V+QBtHYcdF32WpPvDvHUdLXCX3tuV1/4rv+23v9R1fKmJcmoNi7OvoecRLMn
```

Hash data set B

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

```
In [12]: 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, validate=
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:04<00:00, 978clk/s, mean=592, std=45.5]
```

```
In [13]: len(hashed_data_b)
```

```
Out[13]: 5000
```

Find matches between the two sets of CLKs

We have generated two sets of CLKs which represent entity information in a privacy-preserving way. The more similar two CLKs are, the more likely it is that they represent the same entity.

For this task we will use anonlink, a Python (and optimised C++) implementation of anonymous linkage using CLKs.

```
In [ ]: !pip install -U anonlink
```

```
In [14]: from anonlink.entitymatch import calculate_mapping_greedy
         from bitarray import bitarray
         import base64

         def deserialize_bitarray(bytes_data):
             ba = bitarray(endian='big')
             data_as_bytes = base64.decodebytes(bytes_data.encode())
             ba.frombytes(data_as_bytes)
             return ba

         def deserialize_filters(filters):
             res = []
             for i, f in enumerate(filters):
                 ba = deserialize_bitarray(f)
                 res.append((ba, i, ba.count()))
             return res

         clks_a = deserialize_filters(hashed_data_a)
         clks_b = deserialize_filters(hashed_data_b)

         mapping = calculate_mapping_greedy(clks_a, clks_b, threshold=0.9, k=5000)
         print('found {} matches'.format(len(mapping)))
```

```
found 3636 matches
```

Let's investigate some of those matches and the overall matching quality

```
In [15]: a_csv.seek(0)
         b_csv.seek(0)
         a_raw = a_csv.readlines()
         b_raw = b_csv.readlines()

         num_entities = len(b_raw) - 1

         print('idx_a, idx_b, rec_id_a, rec_id_b')
         print('--------------------------------')
         for a_i in range(10):
             if a_i in mapping:
```

```
                a_data = a_raw[a_i + 1].split(',')
                b_data = b_raw[mapping[a_i] + 1].split(',')
                print('{}, {}, {}, {}'.format(a_i+1, mapping[a_i]+1, a_data[0], b_data[0]))

        TP = 0; FP = 0; TN = 0; FN = 0
        for a_i in range(num_entities):
            if a_i in mapping:
                if a_raw[a_i + 1].split(',')[0].split('-')[1] == b_raw[mapping[a_i] + 1].split(',')
                    TP += 1
                else:
                    FP += 1
                    FN += 1 # as we only report one mapping for each element in PII_a, then a wrong
            else:
                FN += 1 # every element in PII_a has a partner in PII_b

        print('--------------------------------')
        print('Precision: {}, Recall: {}, Accuracy: {}'.format(TP/(TP+FP), TP/(TP+FN), (TP+TN)/(TP+T
```

```
idx_a, idx_b, rec_id_a, rec_id_b
--------------------------------
2, 2751, rec-1016-org, rec-1016-dup-0
3, 4657, rec-4405-org, rec-4405-dup-0
4, 4120, rec-1288-org, rec-1288-dup-0
5, 3307, rec-3585-org, rec-3585-dup-0
7, 3945, rec-1985-org, rec-1985-dup-0
8, 993, rec-2404-org, rec-2404-dup-0
9, 4613, rec-1473-org, rec-1473-dup-0
10, 3630, rec-453-org, rec-453-dup-0
--------------------------------
Precision: 1.0, Recall: 0.7272, Accuracy: 0.7272
```

Precision tells us about how many of the found matches are actual matches. The score of 1.0 means that we did perfectly in this respect, however, recall, the measure of how many of the actual matches were correctly identified, is quite low with only 73%.

Let's go back to the mapping calculation (`calculate_mapping_greedy`) an reduce the value for `threshold` to `0.8`.

Great, for this threshold value we get a precision of 100% and a recall of 95.3%.

The explanation is that when the information about an entity differs slightly in the two datasets (e.g. spelling errors, abbrevations, missing values, ...) then the corresponding CLKs will differ in some number of bits as well. For the datasets in this tutorial the perturbations are such that only 72.7% of the derived CLK pairs overlap more than 90%. Whereas almost all matching pairs overlap more than 80%.

If we keep reducing the threshold value, then we will start to observe mistakes in the found matches – the precision decreases. But at the same time the recall value will keep increasing for a while, as a lower threshold allows for more of the actual matches to be found, e.g.: for threshold 0.72, we get precision: 0.997 and recall: 0.992. However, reducing the threshold further will eventually lead to a decrease in both precision and recall: for threshold 0.65 precision is 0.983 and recall is 0.980. Thus it is important to choose an appropriate threshold for the amount of perturbations present in the data.

This concludes the tutorial. Feel free to go back to the CLK generation and experiment on how different setting will affect the matching quality.

1.1.2 Tutorial for CLI tool `clkhash`

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

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 [1]: import io
        import json
        import numpy as np
        import pandas as pd
```

```
In [2]: import recordlinkage
        from recordlinkage.datasets import load_febrl4
```

Data Exploration

First we have a look at the dataset.

```
In [3]: dfA, dfB = load_febrl4()

        dfA.head()
```

```
Out[3]:              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
```

Note that for computing this linkage we will **not** use the social security id column or the `rec_id` index.

```
In [4]: dfA.columns
```

```
Out[4]: Index(['given_name', 'surname', 'street_number', 'address_1', 'address_2',
               'suburb', 'postcode', 'state', 'date_of_birth', 'soc_sec_id'],
              dtype='object')
```

```
In [5]: dfA.to_csv('PII_a.csv')
```

Hashing Schema Definition

A hashing schema instructs clkhash how to treat each column for generating CLKs. A detailed description of the hashing schema can be found in the api docs. We will ignore the columns 'rec_id' and 'soc_sec_id' for CLK generation.

```
In [6]: %%writefile schema.json
        {
          "version": 1,
          "clkConfig": {
            "l": 1024,
            "k": 30,
            "hash": {
              "type": "doubleHash"
            },
            "kdf": {
              "type": "HKDF",
              "hash": "SHA256",
                "info": "c2NoZW1hX2V4YW1wbGU=",
                "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4p
                "keySize": 64
            }
          },
          "features": [
            {
              "identifier": "rec_id",
              "ignored": true
            },
            {
              "identifier": "given_name",
              "format": { "type": "string", "encoding": "utf-8" },
              "hashing": { "ngram": 2, "weight": 1 }
            },
            {
              "identifier": "surname",
              "format": { "type": "string", "encoding": "utf-8" },
              "hashing": { "ngram": 2, "weight": 1 }
            },
            {
              "identifier": "street_number",
              "format": { "type": "integer" },
              "hashing": { "ngram": 1, "positional": true, "weight": 1, "missingValue": {"sentinel":
            },
            {
              "identifier": "address_1",
              "format": { "type": "string", "encoding": "utf-8" },
              "hashing": { "ngram": 2, "weight": 1 }
            },
            {
              "identifier": "address_2",
              "format": { "type": "string", "encoding": "utf-8" },
              "hashing": { "ngram": 2, "weight": 1 }
            },
            {
              "identifier": "suburb",
              "format": { "type": "string", "encoding": "utf-8" },
              "hashing": { "ngram": 2, "weight": 1 }
            },
            {
              "identifier": "postcode",
              "format": { "type": "integer", "minimum": 100, "maximum": 9999 },
              "hashing": { "ngram": 1, "positional": true, "weight": 1 }
            },
            {
              "identifier": "state",
```

```
            "format": { "type": "string", "encoding": "utf-8", "maxLength": 3 },
            "hashing": { "ngram": 2, "weight": 1 }
          },
          {
            "identifier": "date_of_birth",
            "format": { "type": "integer" },
            "hashing": { "ngram": 1, "positional": true, "weight": 1, "missingValue": {"sentinel":
          },
          {
            "identifier": "soc_sec_id",
            "ignored": true
          }
        ]
      }
```

```
Overwriting schema.json
```

Hash the data

We can now hash our Personally Identifiable Information (PII) data from the CSV file using our defined linkage schema. We must provide two *secret keys* to this command - these keys have to be used by both parties hashing data. For this toy example we will use the keys *'key1'* and *'key2'*, for real data, make sure that the keys contain enough entropy, as knowledge of these keys is sufficient to reconstruct the PII information from a CLK! Also, **do not share these keys with anyone, except the other participating party.**

```
In [7]: !clkutil hash PII_a.csv key1 key2 schema.json clks_a.json
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:05<00:00, 927clk/s, mean=885, std=33.4]
CLK data written to clks_a.json
```

Inspect the output

clkhash has hashed the PII, creating a Cryptographic Longterm Key for each entity. The progress bar output shows that the mean popcount is quite high (885 out of 1024) which can effect accuracy.

There are two ways to control the popcount: - You can change the *'k'* value in the `clkConfig` section of the linkage schema. This controls the number of entries in the CLK for each n-gram - or you can modify the individual *'weight'* values for the different fields. It allows to tune the contribution of a column to the CLK. This can be used to de-emphasise columns which are less suitable for linkage (e.g. information that changes frequently).

First, we will change the value of *k* from 30 to 15.

```
In [8]: schema = json.load(open('schema.json', 'rt'))
        schema['clkConfig']['k'] = 15
        json.dump(schema, open('schema.json', 'wt'))

        !clkutil hash PII_a.csv key1 key2 schema.json clks_a.json
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:04<00:00, 867clk/s, mean=648, std=44.1]
CLK data written to clks_a.json
```

And now we will modify the weights to de-emphasise the contribution of the address related columns.

```
In [9]: schema = json.load(open('schema.json', 'rt'))
        schema['clkConfig']['k'] = 20
        address_features = ['street_number', 'address_1', 'address_2', 'suburb', 'postcode', 'state'
        for feature in schema['features']:
            if feature['identifier'] in address_features:
                feature['hashing']['weight'] = 0.5
        json.dump(schema, open('schema.json', 'wt'))
```

```
!clkutil hash PII_a.csv key1 key2 schema.json clks_a.json
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:04<00:00, 924clk/s, mean=602, std=39.8]
CLK data written to clks_a.json
```

Each CLK is serialized in a JSON friendly base64 format:

```
In [10]: # If you have jq tool installed:
         #!jq .clks[0] clks_a.json

         import json
         json.load(open('clks_a.json'))['clks'][0]
```

```
Out[10]: 'BD8JWW7DzwP82PjV5/jbN40+bT3V4z7V+QBtHYcdF32WpPvDvHUdLXCX3tuV1/4rv+23v9R1fKmJcmoNi7OvoecRLMr
```

Hash data set B

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

```
In [11]: dfB.to_csv('PII_b.csv')

         !clkutil hash PII_b.csv key1 key2 schema.json clks_b.json
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:04<00:00, 964clk/s, mean=592, std=45.5]
CLK data written to clks_b.json
```

Find matches between the two sets of CLKs

We have generated two sets of CLKs which represent entity information in a privacy-preserving way. The more similar two CLKs are, the more likely it is that they represent the same entity.

For this task we will use the entity service, which is provided by Data61. The necessary steps are as follows: - The analyst creates a new project with the output type 'mapping'. They will receive a set of credentials from the server. - The analyst then distributes the `update_tokens` to the participating data providers. - The data providers then individually upload their respective CLKs. - The analyst can create *runs* with various thresholds (and other settings) - After the entity service successfully computed the mapping, it can be accessed by providing the `result_token`

First we check the status of an entity service:

```
In [13]: SERVER = 'https://testing.es.data61.xyz'
         !clkutil status --server={SERVER}
```

```
{"project_count": 223, "rate": 52027343, "status": "ok"}
```

The analyst creates a new project on the entity service by providing the hashing schema and result type. The server returns a set of credentials which provide access to the further steps for project.

```
In [15]: !clkutil create-project --server={SERVER} --schema schema.json --output credentials.json --t
```

```
Entity Matching Server: https://testing.es.data61.xyz
Checking server status
Server Status: ok
```

The returned credentials contain a - `project_id`, which identifies the project - `result_token`, which gives access to the mapping result, once computed - `upload_tokens`, one for each provider, allows uploading CLKs.

```
In [16]: credentials = json.load(open('credentials.json', 'rt'))
         !python -m json.tool credentials.json
```

```
{
    "project_id": "5c9a47049161bcb3f32dd1fef4c71c1df9cc7658f5e2cd55",
    "result_token": "2886b2faf85ad994339059f192a1b8f32206ec32d878b160",
```

```
    "update_tokens": [
        "7d08294eed16bbe8b3189d193358258b3b5045e67f44306f",
        "04da88e3a5e90aa55049c5a2e8a7085a8bc691653d895447"
    ]
}
```

Uploading the CLKs to the entity service

Each party individually uploads its respective CLKs to the entity service. They need to provide the `resource_id`, which identifies the correct mapping, and an `update_token`.

```
In [17]: !clkutil upload \
                --project="{credentials['project_id']}" \
                 --apikey="{credentials['update_tokens'][0]}" \
                 --output "upload_a.json" \
                 --server="{SERVER}" \
                "clks_a.json"

         !clkutil upload \
                --project="{credentials['project_id']}" \
                 --apikey="{credentials['update_tokens'][1]}" \
                 --output "upload_b.json" \
                 --server="{SERVER}" \
                "clks_b.json"
```

```
Uploading CLK data from clks_a.json
To Entity Matching Server: https://testing.es.data61.xyz
Project ID: 5c9a47049161bcb3f32dd1fef4c71c1df9cc7658f5e2cd55
Checking server status
Status: ok
Uploading CLK data to the server
Uploading CLK data from clks_b.json
To Entity Matching Server: https://testing.es.data61.xyz
Project ID: 5c9a47049161bcb3f32dd1fef4c71c1df9cc7658f5e2cd55
Checking server status
Status: ok
Uploading CLK data to the server
```

Now that the CLK data has been uploaded the analyst can create one or more *runs*. Here we will start by calculating a mapping with a threshold of `0.9`:

```
In [18]: !clkutil create --verbose  \
             --server="{SERVER}" \
             --output "run_info.json" \
             --threshold=0.9 \
             --project="{credentials['project_id']}" \
             --apikey="{credentials['result_token']}" \
             --name="tutorial_run"
```

```
Entity Matching Server: https://testing.es.data61.xyz
Checking server status
Server Status: ok
```

```
In [23]: run_info = json.load(open('run_info.json', 'rt'))
         run_info
```

```
Out[23]: {'name': 'tutorial_run',
          'notes': 'Run created by clkhash command line tool',
          'run_id': 'b700b16393eb5eb704322497226078c36ad9e16724797239',
          'threshold': 0.9}
```

Results

Now after some delay (depending on the size) we can fetch the results. This can be done with clkutil:

```
In [26]: !clkutil results \
                --project="{credentials['project_id']}" \
                --apikey="{credentials['result_token']}" \
                --run="{run_info['run_id']}" \
                --server="{SERVER}" \
                --output results.txt

         with open('results.txt') as f:
             str_mapping = json.load(f)['mapping']
             mapping = {int(k): int(v) for k,v in str_mapping.items()}
         print('The service linked {} entities.'.format(len(mapping)))
```

```
The service linked 3636 entities.
```

```
Checking server status
Status: ok
Response code: 200
Received result
```

Let's investigate some of those matches and the overall matching quality

```
In [27]: with open('PII_a.csv', 'rt') as f:
             a_raw = f.readlines()
         with open('PII_b.csv', 'rt') as f:
             b_raw = f.readlines()

         num_entities = len(b_raw) - 1

         print('idx_a, idx_b, rec_id_a, rec_id_b')
         print('--------------------------------')
         for a_i in range(10):
             if a_i in mapping:
                 a_data = a_raw[a_i + 1].split(',')
                 b_data = b_raw[mapping[a_i] + 1].split(',')
                 print('{}, {}, {}, {}'.format(a_i+1, mapping[a_i]+1, a_data[0], b_data[0]))

         TP = 0; FP = 0; TN = 0; FN = 0
         for a_i in range(num_entities):
             if a_i in mapping:
                 if a_raw[a_i + 1].split(',')[0].split('-')[1] == b_raw[mapping[a_i] + 1].split(',')
                     TP += 1
                 else:
                     FP += 1
                     FN += 1 # as we only report one mapping for each element in PII_a, then a wrong
             else:
                 FN += 1 # every element in PII_a has a partner in PII_b

         print('--------------------------------')
         print('Precision: {}, Recall: {}, Accuracy: {}'.format(TP/(TP+FP), TP/(TP+FN), (TP+TN)/(TP+T
```

```
idx_a, idx_b, rec_id_a, rec_id_b
--------------------------------
2, 2751, rec-1016-org, rec-1016-dup-0
3, 4657, rec-4405-org, rec-4405-dup-0
4, 4120, rec-1288-org, rec-1288-dup-0
5, 3307, rec-3585-org, rec-3585-dup-0
7, 3945, rec-1985-org, rec-1985-dup-0
```

```
8, 993, rec-2404-org, rec-2404-dup-0
9, 4613, rec-1473-org, rec-1473-dup-0
10, 3630, rec-453-org, rec-453-dup-0
--------------------------------
Precision: 1.0, Recall: 0.7272, Accuracy: 0.7272
```

Precision tells us about how many of the found matches are actual matches. The score of 1.0 means that we did perfectly in this respect, however, recall, the measure of how many of the actual matches were correctly identified, is quite low with only 73%.

Let's go back and create another mapping with a `threshold` value of `0.8`.

Great, for this threshold value we get a precision of 100% and a recall of 95.3%.

The explanation is that when the information about an entity differs slightly in the two datasets (e.g. spelling errors, abbrevations, missing values, ...) then the corresponding CLKs will differ in some number of bits as well. For the datasets in this tutorial the perturbations are such that only 72.7% of the derived CLK pairs overlap more than 90%. Whereas almost all matching pairs overlap more than 80%.

If we keep reducing the threshold value, then we will start to observe mistakes in the found matches – the precision decreases. But at the same time the recall value will keep increasing for a while, as a lower threshold allows for more of the actual matches to be found, e.g.: for threshold 0.72, we get precision: 0.997 and recall: 0.992. However, reducing the threshold further will eventually lead to a decrease in both precision and recall: for threshold 0.65 precision is 0.983 and recall is 0.980. Thus it is important to choose an appropriate threshold for the amount of perturbations present in the data.

This concludes the tutorial. Feel free to go back to the CLK generation and experiment on how different setting will affect the matching quality.

```
In [ ]:
```

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 Help

The `clkutil` tool has help pages for all commands built in.:

```
$ clkutil hash --help
Usage: clkutil hash [OPTIONS] INPUT KEYS... SCHEMA OUTPUT

  Process data to create CLKs

  Given a file containing csv data as INPUT, and a json document defining
  the expected schema, verify the schema, then hash the data to create CLKs
  writing to OUTPUT. Note the CSV file should contain a header row - however
  this row is not used by this tool.

  It is important that the keys are only known by the two data providers.
  Two words should be provided. For example:
```

(continues on next page)

(continued from previous page)

```
  $clkutil hash input.txt horse staple output.txt

  Use "-" to output to stdout.

Options:
  -q, --quiet             Quiet any progress messaging
  --no-header             Don't skip the first row
  --check-header BOOLEAN  If true, check the header against the schema
  --validate BOOLEAN      If true, validate the entries against the schema
  --help                  Show this message and exit.
```

1.2.2 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.

1.2.3 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
```

A corresponding hashing schema can be generated as well:

```
$ clkutil generate-default-schema schema.json
$ cat schema.json
{
  "version": 1,
  "clkConfig": {
    "l": 1024,
    "k": 30,
    "hash": {
      "type": "doubleHash"
    },
    "kdf": {
      "type": "HKDF",
      "hash": "SHA256",
      "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
↪G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",
      "info": "c2NoZW1hX2V4YW1wbGU=",
      "keySize": 64
    }
  },
  "features": [
    {
      "identifier": "INDEX",
      "format": {
        "type": "integer"
      },
      "hashing": {
        "ngram": 1,
        "weight": 0
      }
    },
    {
      "identifier": "NAME freetext",
      "format": {
        "type": "string",
        "encoding": "utf-8",
        "case": "mixed",
        "minLength": 3
      },
      "hashing": {
        "ngram": 2,
        "weight": 0.5
      }
    },
    {
      "identifier": "DOB YYYY/MM/DD",
      "format": {
        "type": "string",
        "encoding": "ascii",
        "description": "Numbers separated by slashes, in the year, month, day order",
        "pattern": "(?:\\d\\d\\d\\d/\\d\\d/\\d\\d)\\Z"
      },
      "hashing": {
        "ngram": 1,
        "positional": true
      }
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "identifier": "GENDER M or F",
      "format": {
        "type": "enum",
        "values": ["M", "F"]
      },
      "hashing": {
        "ngram": 1,
        "weight": 2
      }
    }
  ]
}
```

1.2.4 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.

1.2.5 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-project
- create
- upload
- results

See also the *Tutorial for CLI*.

1.3 Hashing Schema

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 their respective CLKs 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 master-schemas/v1.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": "index",
      "ignored": true
    },
    {
      "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 }
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "identifier": "postcode",
      "format": {
        "type": "integer",
        "minimum": 1000,
        "maximum": 9999
      },
      "hashing":{
        "ngram": 1,
        "positional": true,
        "missingValue": {
          "sentinel": "N/A",
          "replaceWith": ""
        }
      }
    }
  ]
}
```

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"

features

A feature is either described by a *featureConfig*, or alternatively, it can be ignored by the clkhash library by defining a *ignoreFeature* section.

ignoreFeature

If defined, then clkhash will ignore this feature.

name	type	optional	description
identifier	string	no	the name of the feature
ignored	boolean	no	has to be set to "True"
description	string	yes	free text, ignored by clkhash

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.
missingValue	*missingValue*	yes	allows to define how missing values are handled

missingValue

Data sets are not always complete – they can contain missing values. If specified, then clkhash will not check the format for these missing values, and will optionally replace them with the 'replaceWith' value. This can be useful if the data

name	type	optional	description
sentinel	string	no	the sentinel value indicates missing data, e.g. 'Null', 'N/A', '', ...
replaceWith	string	yes	specifies the value clkhash should use instead of the sentinel value.

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	integer describing the lower bound of the input values.
maximum	integer	yes	integer describing the upper bound of the input values.
description	string	yes	free text, ignored by clkhash.

dateFormat

A date is described by an ISO C89 compatible strftime() format string. For example, the format string for the internet date format as described in rfc3339, would be '%Y-%m-%d'. The clkhash library will convert the given date to the '%Y%m%d' representation for hashing, as any fill character like '-' or '/' do not add to the uniqueness of an entity.

name	type	optional	description
type	string	no	has to be "date"
format	string	no	ISO C89 compatible format string, eg: for 1989-11-09 the format is '%Y-%m-%d'
description	string	yes	free text, ignored by clkhash.

The following subset contains the most useful format codes:

directive	meaning	example
%Y	Year with century as a decimal number	1984, 3210, 0001
%y	Year without century, zero-padded	00, 09, 99
%m	Month as a zero-padded decimal number	01, 12
%d	Day of the month, zero-padded	01, 25, 31

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

The available schemes for encoding n-grams.

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*, *keys*, *k*, *l*, *encoding*)

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
- **keys** – 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)
- **encoding** – the encoding to use when turning the ngrams to bytes

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

`clkhash.bloomfilter.`**`crypto_bloom_filter`**(*record*, *tokenizers*, *fields*, *keys*, *hash_properties*)

Computes the composite Bloom filter encoding of a record.

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 tokenizers. A tokenizer is a function that returns tokens from a string.
- **fields** – A list of FieldSpec. One for each field.
- **keys** – Keys for the hash functions as a tuple of lists of bytes.
- **hash_properties** – Global hashing properties.

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*, *encoding*)

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
- **encoding** – the encoding to use when turning the ngrams to bytes

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*, *encoding*)

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
- **keys** – tuple with (key_sha1, key_md5). That is, (hmac secret keys for sha1 as bytes, 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
- **encoding** – the encoding to use when turning the ngrams to bytes

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.`**`int_from_bytes`**()

int.from_bytes(bytes, byteorder, *, signed=False) -> int

Return the integer represented by the given array of bytes.

The bytes argument must be a bytes-like object (e.g. bytes or bytearray).

The byteorder argument determines the byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use 'sys.byteorder' as the byte order value.

The signed keyword-only argument indicates whether two's complement is used to represent the integer.

`clkhash.bloomfilter.`**`stream_bloom_filters`**(*dataset*, *keys*, *schema*)

Compute composite Bloom filters (CLKs) for every record in an iterable dataset.

Parameters

- **dataset** – An iterable of indexable records.
- **schema** – An instantiated Schema instance

- **keys** – A tuple of two lists of secret keys used in the HMAC.

Returns Generator yielding bloom filters as 3-tuples

CLK

Generate CLK from data.

`clkhash.clk.`**`chunks`**(*seq*, *chunk_size*)

Split seq into chunk_size-sized chunks.

Parameters

- **seq** – A sequence to chunk.
- **chunk_size** – The size of chunk.

`clkhash.clk.`**`generate_clk_from_csv`**(*input_f*, *keys*, *schema*, *validate=True*, *header=True*, *progress_bar=True*)

Generate Bloom filters from CSV file, then serialise them.

This function also computes and outputs the Hamming weight (a.k.a popcount – the number of bits set to high) of the generated Bloom filters.

Parameters

- **input_f** – A file-like object of csv data to hash.
- **keys** – A tuple of two lists of secret keys.
- **schema** – Schema specifying the record formats and hashing settings.
- **validate** – Set to *False* to disable validation of data against the schema. Note that this will silence warnings whose aim is to keep the hashes consistent between data sources; this may affect linkage accuracy.
- **header** – Set to *False* if the CSV file does not have a header. Set to *'ignore'* if the CSV file does have a header but it should not be checked against the schema.
- **progress_bar** (*bool*) – Set to *False* to disable the progress bar.

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

`clkhash.clk.`**`generate_clks`**(*pii_data*, *schema*, *keys*, *validate=True*, *callback=None*)

`clkhash.clk.`**`hash_and_serialize_chunk`**(*chunk_pii_data*, *keys*, *schema*)

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.
- **keys** – A tuple of two lists of secret keys used in the HMAC.
- **schema** (*Schema*) – Schema specifying the entry formats and hashing settings.

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

key derivation

`clkhash.key_derivation.`**`generate_key_lists`**(*master_secrets*, *num_identifier*, *key_size=64*, *salt=None*, *info=None*, *kdf='HKDF'*, *hash_algo='SHA256'*)

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
- **`hash_algo`** – the hashing algorithm to use (ignored if *kdf* is not 'HKDF')

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`**(*master_secret*, *num_keys*, *hash_algo='SHA256'*, *salt=None*, *info=None*, *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

- **`master_secret`** – input keying material
- **`num_keys`** – the number of keys the kdf should produce
- **`hash_algo`** – The hash function used by HKDF for the internal HMAC calls. The choice of hash function defines the maximum length of the output key material. Output bytes <= 255 * hash digest size (in bytes).
- **`salt`** – HKDF is defined to operate with and without random salt. This is done to accommodate applications where a salt value is not available. We stress, however, that the use of salt adds significantly to themstrength of HKDF, ensuring independence between different uses of the hash function, supporting "source-independent" extraction, and strengthening the analytical results that back the HKDF design.

  Random salt differs fundamentally from the initial keying

  material in two ways: it is non-secret and can be re-used. Ideally, the salt value is a random (or pseudorandom) string

  of the length HashLen. Yet, even a salt value of less quality (shorter in size or with limited entropy) may still make a significant contribution to the security of the output keying material.

- **info** – While the 'info' value is optional in the definition of HKDF, it is often of great importance in applications. Its main objective is to bind the derived key material to application- and context-specific information. For example, 'info' may contain a protocol number, algorithm identifiers, user identities, etc. In particular, it may prevent the derivation of the same keying material for different contexts (when the same input key material (IKM) is used in such different contexts). It may also accommodate additional inputs to the key expansion part, if so desired (e.g., an application may want to bind the key material to its length L, thus making L part of the 'info' field). There is one technical requirement from 'info': it should be independent of the input key material value IKM.
- **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

Randomly generated PII records.

SCHEMA = <Schema (v1): 4 fields>

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*, *subsets=2*)

Return random subsets with nonempty intersection.

The random subsets are of specified size. If an element is common to two subsets, then it is common to all subsets. This overlap is controlled by a parameter.

Parameters

- **sz** – size of subsets to generate
- **overlap** – size of the intersection, as fraction of the subset length
- **subsets** – number of subsets to generate

Raises **ValueError** – if there aren't sufficiently many names in the list to satisfy the request; more precisely, raises if (1 - subsets) * floor(overlap * sz)

- subsets * sz > len(self.names).

Returns tuple of subsets

load_names()

Loads a name database from package data

Uses data files sourced from http://www.quietaffiliate.com/free-first-name-and-last-name-databases-csv-and-sql/

schema_types

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

Loads first column of specified CSV file from package data.

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

Generate 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*, *headers*, *file*)

Output generated data to file as CSV with header.

Parameters

- **data** – An iterable of tuples containing raw data.
- **headers** – Iterable of feature names
- **file** – A writeable stream in which to write the CSV

schema

Schema loading and validation.

class clkhash.schema.**GlobalHashingProperties**(*k*, *l*, *hash_type*, *kdf_type*, *xor_folds=0*, *hash_prevent_singularity=None*, *kdf_hash='SHA256'*, *kdf_info=None*, *kdf_salt=None*, *kdf_key_size=64*)

Bases: object

Stores global hashing properties.

Parameters

- **k** – The number of bits of the hash to set per ngram.
- **l** – The length of the resulting hash in bits. This is the length after XOR folding.
- **xor_folds** – The number of XOR folds to perform on the hash.
- **hash_type** – The hashing function to use. Choices are 'doubleHash' and 'blakeHash'.
- **hash_prevent_singularity** – Ignored unless hash_type is 'doubleHash'. Prevents bloom filter collisions in certain cases when True.
- **kdf_type** – The key derivation function to use. Currently, the only permitted value is 'HKDF'.
- **kdf_hash** – The hash function to use in key derivation. The options are 'SHA256' and 'SHA512'.
- **kdf_info** – The info for key derivation. See documentation of hkdf for details.
- **kdf_salt** – The salt for key derivation. See documentation of hkdf for details.

- **kdf_key_size** – The size of the derived keys in bytes.

classmethod from_json_dict(*properties_dict*)
Make a GlobalHashingProperties object from a dictionary.

Parameters **properties_dict** – The dictionary must have a *'type'* key and a *'config'* key. The *'config'* key must map to a dictionary containing a *'kdf'* key, which itself maps to a dictionary. That dictionary must have *'type'*, *'hash'*, *'keySize'*, *'salt'*, and *'type'* keys.

Returns The resulting `GlobalHashingProperties` object.

exception clkhash.schema.**MasterSchemaError**
Bases: `Exception`

Master schema missing? Corrupted? Otherwise surprising? This is the exception for you!

class clkhash.schema.**Schema**(*version*, *hashing_globals*, *fields*)
Bases: `object`

Linkage Schema which describes how to encode plaintext identifiers.

Variables

- **version** – Version for the schema. Needed to keep behaviour consistent between clkhash versions for the same schema.
- **hashing_globals** – Configuration affecting hashing of all fields. For example cryptographic salt material, bloom filter length.
- **fields** – Information and configuration specific to each field. For example how to validate and tokenize a phone number.

classmethod from_json_dict(*schema_dict*, *validate=True*)
Make a Schema object from a dictionary.

Parameters

- **schema_dict** – This dictionary must have a *'features'* key specifying the columns of the dataset. It must have a *'version'* key containing the master schema version that this schema conforms to. It must have a *'hash'* key with all the globals.
- **validate** – (default True) Raise an exception if the schema does not conform to the master schema.

Returns The resulting `Schema` object.

classmethod from_json_file(*schema_file*, *validate=True*)
Load a Schema object from a json file.

Parameters

- **schema_file** – A JSON file containing the schema.
- **validate** – (default True) Raise an exception if the schema does not conform to the master schema.

Raises ***SchemaError*** – When the schema is invalid.

Returns The resulting `Schema` object.

exception clkhash.schema.**SchemaError**
Bases: `Exception`

The user-defined schema is invalid.

`clkhash.schema.get_master_schema(version)`

Loads the master schema of given version as bytes.

Parameters **version** – The version of the master schema whose path we wish to retrieve.

Raises *SchemaError* – When the schema version is unknown. This usually means that either (a) clkhash is out of date, or (b) the schema version listed is incorrect.

Returns Bytes of the schema.

`clkhash.schema.validate_schema_dict(schema)`

Validate the schema.

This raises iff either the schema or the master schema are invalid. If it's successful, it returns nothing.

Parameters **schema** – The schema to validate, as parsed by *json*.

Raises

- *SchemaError* – When the schema is invalid.
- *MasterSchemaError* – When the master schema is invalid.

field_formats

Classes that specify the requirements for each column in a dataset. They take care of validation, and produce the settings required to perform the hashing.

class `clkhash.field_formats.DateSpec(identifier, hashing_properties, format, description=None)`

Bases: *clkhash.field_formats.FieldSpec*

Represents a field that holds dates.

Dates are specified as full-dates in a format that can be described as a *strptime()* (C89 standard) compatible format string. E.g.: the format for the standard internet format RFC3339 (e.g. 1996-12-19) is '%Y-%m-%d'.

ivar str format The format of the date.

OUTPUT_FORMAT = '%Y%m%d'

classmethod from_json_dict(*json_dict*)

Make a DateSpec object from a dictionary containing its properties.

Parameters

- **json_dict** (*dict*) – This dictionary must contain a *'format'* key. In addition, it must contain a *'hashing'* key, whose contents are passed to *FieldHashingProperties*.
- **json_dict** – The properties dictionary.

validate(*str_in*)

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

An entry is invalid iff (1) the string does not represent a date in the correct format; or (2) the date it represents is invalid (such as 30 February).

Parameters **str_in** (*str*) – String to validate.

Raises

- *InvalidEntryError* – Iff entry is invalid.
- *ValueError* – When self.format is unrecognised.

class clkhash.field_formats.**EnumSpec**(*identifier*, *hashing_properties*, *values*, *description=None*)

Bases: *clkhash.field_formats.FieldSpec*

Represents a field that holds an enum.

The finite collection of permitted values must be specified.

Variables **values** – The set of permitted values.

classmethod from_json_dict(*json_dict*)

Make a EnumSpec object from a dictionary containing its properties.

Parameters **json_dict** (*dict*) – This dictionary must contain an *'enum'* key specifying the permitted values. In addition, it must contain a *'hashing'* key, whose contents are passed to *FieldHashingProperties*.

validate(*str_in*)

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

An entry is invalid iff it is not one of the permitted values.

Parameters **str_in** (*str*) – String to validate.

Raises ***InvalidEntryError*** – When entry is invalid.

class clkhash.field_formats.**FieldHashingProperties**(*ngram*, *encoding='utf-8'*, *weight=1*, *positional=False*, *missing_value=None*)

Bases: object

Stores the settings used to hash a field. This includes the encoding and tokenisation parameters.

Variables

- **encoding** (*str*) – The encoding to use when converting the string to bytes. Refer to Python's documentation for possible values.
- **ngram** (*int*) – The n in n-gram. Possible values are 0, 1, and 2.
- **positional** (*bool*) – Controls whether the n-grams are positional.
- **weight** (*float*) – Controls the weight of the field in the Bloom filter.

classmethod from_json_dict(*json_dict*)

Make a *FieldHashingProperties* object from a dictionary.

Parameters **json_dict** (*dict*) – The dictionary must have have an 'ngram' key. It may have 'positional' and 'weight' keys; if these are missing, then they are filled with the default values. The encoding is always set to the default value.

Returns A *FieldHashingProperties* instance.

replace_missing_value(*str_in*)

returns 'str_in' if it is not equals to the 'sentinel' as defined in the missingValue section of the schema. Else it will return the 'replaceWith' value.

Parameters **str_in** –

Returns str_in or the missingValue replacement value

class clkhash.field_formats.**FieldSpec**(*identifier*, *hashing_properties*, *description=None*)

Bases: object

Abstract base class representing the specification of a column in the dataset. Subclasses validate entries, and modify the *hashing_properties* ivar to customise hashing procedures.

Variables

- **identifier** (*str*) – The name of the field.
- **description** (*str*) – Description of the field format.
- **hashing_properties** (FieldHashingProperties) – The properties for hashing.

format_value(*str_in*)

formats the value 'str_in' for hashing according to this field's spec.

There are several reasons why this might be necessary:

1. This field contains missing values which have to be replaced by some other string
2. There are several different ways to describe a specific value for this field, e.g.: all of '+65', ' 65', '65' are valid representations of the integer 65.
3. Entries of this field might contain elements with no entropy, e.g. dates might be formatted as yyyy-mm-dd, thus all dates will have '-' at the same place. These artifacts have no value for entity resolution and should be removed.

Parameters **str_in** (*str*) – the string to format

Returns a string representation of 'str_in' which is ready to be hashed

classmethod from_json_dict(*field_dict*)

Initialise a FieldSpec object from a dictionary of properties.

Parameters **field_dict** (*dict*) – The properties dictionary to use. Must contain a *'hashing'* key that meets the requirements of *FieldHashingProperties*. Subclasses may require

Raises ***InvalidSchemaError*** – When the *properties* dictionary contains invalid values. Exactly what that means is decided by the subclasses.

is_missing_value(*str_in*)

tests if 'str_in' is the sentinel value for this field

Parameters **str_in** (*str*) – String to test if it stands for missing value

Returns True if a missing value is defined for this field and str_in matches this value

validate(*str_in*)

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

Subclasses must override this method with their own validation. They should call the parent's *validate* method via *super*.

Parameters **str_in** (*str*) – String to validate.

Raises ***InvalidEntryError*** – When entry is invalid.

class clkhash.field_formats.**Ignore**(*identifier=None*)

Bases: *clkhash.field_formats.FieldSpec*

represent a field which will be ignored throughout the clk processing.

validate(*str_in*)

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

Subclasses must override this method with their own validation. They should call the parent's *validate* method via *super*.

Parameters **str_in** (*str*) – String to validate.

Raises ***InvalidEntryError*** – When entry is invalid.

class clkhash.field_formats.**IntegerSpec**(*identifier*, *hashing_properties*, *description=None*, *minimum=None*, *maximum=None*, ***kwargs*)

Bases: *clkhash.field_formats.FieldSpec*

Represents a field that holds integers.

Minimum and maximum values may be specified.

Variables

- **minimum** (*int*) – The minimum permitted value.
- **maximum** (*int*) – The maximum permitted value or None.

classmethod from_json_dict(*json_dict*)

Make a IntegerSpec object from a dictionary containing its properties.

Parameters

- **json_dict** (*dict*) – This dictionary may contain *'minimum'* and *'maximum'* keys. In addition, it must contain a *'hashing'* key, whose contents are passed to *FieldHashingProperties*.
- **json_dict** – The properties dictionary.

validate(*str_in*)

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

An entry is invalid iff (1) the string does not represent a base-10 integer; (2) the integer is not between *self.minimum* and *self.maximum*, if those exist; or (3) the integer is negative.

Parameters **str_in** (*str*) – String to validate.

Raises ***InvalidEntryError*** – When entry is invalid.

exception clkhash.field_formats.**InvalidEntryError**

Bases: ValueError

An entry in the data file does not conform to the schema.

field_spec = None

exception clkhash.field_formats.**InvalidSchemaError**

Bases: ValueError

The schema is not valid.

This exception is raised if, for example, a regular expression included in the schema is not syntactically correct.

class clkhash.field_formats.**MissingValueSpec**(*sentinel*, *replace_with=None*)

Bases: object

Stores the information about how to find and treat missing values.

Variables

- **sentinel** (*str*) – sentinel is the string that identifies a missing value e.g.: 'N/A', ''. the sentinel will not be validated against the feature format definition

- **replaceWith** (*str*) – defines the string which replaces the sentinel whenever present, can be 'None', then sentinel will not be replaced.

classmethod from_json_dict(*json_dict*)

class clkhash.field_formats.**StringSpec**(*identifier*, *hashing_properties*, *description=None*, *regex=None*, *case='mixed'*, *min_length=0*, *max_length=None*)

Bases: *clkhash.field_formats.FieldSpec*

Represents a field that holds strings.

One way to specify the format of the entries is to provide a regular expression that they must conform to. Another is to provide zero or more of: minimum length, maximum length, casing (lower, upper, mixed).

Each string field also specifies an encoding used when turning characters into bytes. This is stored in *hashing_properties* since it is needed for hashing.

Variables

- **regex** – Compiled regular expression that entries must conform to. Present only if the specification is regex- -based.
- **case** (*str*) – The casing of the entries. One of *'lower'*, *'upper'*, or *'mixed'*. Default is *'mixed'*. Present only if the specification is not regex-based.
- **min_length** (*int*) – The minimum length of the string. *None* if there is no minimum length. Present only if the specification is not regex-based.
- **max_length** (*int*) – The maximum length of the string. *None* if there is no maximum length. Present only if the specification is not regex-based.

classmethod from_json_dict(*json_dict*)

Make a StringSpec object from a dictionary containing its properties.

Parameters **json_dict** (*dict*) – This dictionary must contain an *'encoding'* key associated with a Python-conformant encoding. It must also contain a *'hashing'* key, whose contents are passed to *FieldHashingProperties*. Permitted keys also include *'pattern'*, *'case'*, *'minLength'*, and *'maxLength'*.

Raises ***InvalidSchemaError*** – When a regular expression is provided but is not a valid pattern.

validate(*str_in*)

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

An entry is invalid iff (1) a pattern is part of the specification of the field and the string does not match it; (2) the string does not match the provided casing, minimum length, or maximum length; or (3) the specified encoding cannot represent the string.

Parameters **str_in** (*str*) – String to validate.

Raises

- ***InvalidEntryError*** – When entry is invalid.
- **ValueError** – When self.case is not one of the permitted values (*'lower'*, *'upper'*, or *'mixed'*).

clkhash.field_formats.**spec_from_json_dict**(*json_dict*)

Turns a dictionary into the appropriate object.

Parameters **json_dict** (*dict*) – A dictionary with properties.

Returns An initialised instance of the appropriate FieldSpec subclass.

tokenizer

Functions to tokenize words (PII)

`clkhash.tokenizer.`**`get_tokenizer`**(*hash_settings*)

Get tokeniser function from the hash settings.

This function takes a FieldHashingProperties object. It returns a function that takes a string and tokenises based on those properties.

`clkhash.tokenizer.`**`tokenize`**(*n*, *positional*, *word*, *ignore=None*)

Produce *n*-grams of *word*.

Parameters

- **`n`** – Length of *n*-grams.
- **`positional`** – If *True*, then include the index of the substring with the *n*-gram.
- **`word`** – The string to tokenize.
- **`ignore`** – The substring whose occurrences we remove from *word* before tokenization.

Raises **`ValueError`** – When *n* is negative.

Returns Tuple of n-gram strings.

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

Index

I

L

M

N

O

R

S

T

V