Anonlink Client Documentation

Release 0.1.4

Confidential Computing

Aug 11, 2020

Contents

`anonlink-client` is a client-facing API to interact with anonlink system including command line tools and Rest API communication. It works with the following three components in Anonlink system:

- clkhash
- blocklib
- anonlink-entity-service

Anonlink-client is Apache 2.0 licensed, supports Python version 3.6+ and run on Windows, OSX and Linux.

Install with pip:

```
pip install anonlink-client
```

CHAPTER 1

Table of Contents

1.1 Tutorials

1.1.1 Multi-party Record Linkage with Blocking

In this tutorial, we will demonstrate the CLI tools for multiparty record linkage with blocking techniques.

[1]:
```
import io
import os
import math
import time
from IPython import display
import json
from collections import defaultdict
import pandas as pd

import anonlink
import clkhash
from clkhash import clk
from blocklib import assess_blocks_2party

from anonlinkclient.utils import combine_clks_blocks, deserialize_bitarray,␣
↪deserialize_filters

%run util.py
```

Suppose we are interested to find records that appear at least twice in 3 parties

Generate CLKs and Candidate Blocks

First we have a look at dataset

[2]:
```
# NBVAL_IGNORE_OUTPUT
corruption_rate = 20
file_template = 'data/ncvr_numrec_5000_modrec_2_ocp_' + str(corruption_rate) + '_myp_
↪{}_nump_10.csv'
df1 = pd.read_csv(file_template.format(0))
df1.head()
```

[2]:

	recid	givenname	surname	suburb	postcode
0	1503359	pauline	camkbell	lilescille	28091
1	1972058	deborah	galyen	ennike	286z3
2	889525	charle5	mitrhell	roaring river	28669
3	4371845	petehr	werts	swannanoa	28478
4	1187991	katpy	silbiger	duyham	27705

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

[3]:
```
# NBVAL_IGNORE_OUTPUT
with open("novt_schema.json") as f:
    print(f.read())
```

```
{
  "version": 3,
  "clkConfig": {
    "l": 1024,
    "kdf": {
      "type": "HKDF",
      "hash": "SHA256",
      "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
↪G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",
      "info": "c2NoZW1hX2V4YW1wbGU=",
      "keySize": 64
    }
  },
  "features": [
        {
      "identifier": "recid",
      "ignored": true
    },
    {
      "identifier": "givenname",
      "format": {
        "type": "string",
        "encoding": "utf-8",
        "maxLength": 30,
        "case": "lower"
      },
      "hashing": {
        "comparison": {"type":  "ngram", "n":  2},
        "strategy": {"bitsPerFeature":  100},
        "hash": {"type": "blakeHash"},
        "missingValue": {
          "sentinel": ".",
          "replaceWith": ""
        }
      }
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "identifier": "surname",
      "format": {
        "type": "string",
        "encoding": "utf-8",
        "maxLength": 30,
        "case": "lower"
      },
      "hashing": {
        "comparison": {"type":  "ngram", "n":  2},
        "strategy": {"bitsPerFeature":  100},
        "hash": {"type": "blakeHash"},
        "missingValue": {
          "sentinel": ".",
          "replaceWith": ""
        }
      }
    },
    {
      "identifier": "suburb",
      "format": {
        "type": "string",
        "encoding": "utf-8",
        "maxLength": 30,
        "case": "lower"
      },
      "hashing": {
        "comparison": {"type":  "ngram", "n":  2},
        "strategy": {"bitsPerFeature":  100},
        "hash": {"type": "blakeHash"},
        "missingValue": {
          "sentinel": ".",
          "replaceWith": ""
        }
      }
    },
    {
      "identifier": "postcode",
      "format": {
        "type": "string",
        "encoding": "utf-8",
        "maxLength": 30,
        "case": "lower"
      },
      "hashing": {
        "comparison": {"type":  "ngram", "n":  2},
        "strategy": {"bitsPerFeature":  100},
        "hash": {"type": "blakeHash"},
        "missingValue": {
          "sentinel": ".",
          "replaceWith": ""
        }
      }
    }
  ]
}
```

Validate the schema

The command line tool can check that the linkage schema is valid:

[4]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink validate-schema "novt_schema.json"
```

```
schema is valid
```

Hash data

We can now hash our Personally Identifiable Information (PII) data from the CSV file using our defined linkage schema. We must provide a secret key to this command - this key has to be used by both parties hashing data. For this toy example we will use the secret 'secret', for real data, make sure that the secret contains enough entropy, as knowledge of this secret is sufficient to reconstruct the PII information from a CLK!

[5]:
```
secret = 'secret'
```

[6]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink hash 'data/ncvr_numrec_5000_modrec_2_ocp_20_myp_0_nump_10.csv' secret 'novt_
↪schema.json' 'novt_clk_0.json'
```

```
CLK data written to novt_clk_0.json
```

Let's hash data for party B and C:

[7]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink hash 'data/ncvr_numrec_5000_modrec_2_ocp_20_myp_1_nump_10.csv' secret 'novt_
↪schema.json' 'novt_clk_1.json'
```

```
CLK data written to novt_clk_1.json
```

[8]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink hash 'data/ncvr_numrec_5000_modrec_2_ocp_20_myp_2_nump_10.csv' secret 'novt_
↪schema.json' 'novt_clk_2.json'
```

```
CLK data written to novt_clk_2.json
```

`anonlink` provides a command `describe` to inspect the hashing results i.e. the Cryptographic Longterm Key (CLK). Normally we will expect a relative symmetric shape popcount with a moderate mean comparing to bloom filter length.

[9]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink describe 'novt_clk_0.json'
```

```
    ------------------------------------------------------------------------------------
↪-----------------------------------------
    |                                                          popcounts                 ␣
↪                                        |
    ------------------------------------------------------------------------------------
↪-----------------------------------------

 298|                                      o
 282|                                    o ooooo
 267|                                    o ooooo
 251|                                    ooooooo
```

(continues on next page)

(continued from previous page)

```
 235|                                  oooooooooo o
 220|                                  oooooooooo o
 204|                                  oooooooooo o
 188|                                o oooooooooo oo
 173|                                ooooooooooo oo o
 157|                                ooooooooooo oooo
 141|                               oooooooooooo ooooo
 126|                               oooooooooooo ooooo
 110|                              ooooooooooooo ooooo
  94|                             oooooooooooooo ooooo
  79|                            ooooooooooooooo ooooooo
  63|                           ooooooooooooooooo ooooooo
  47|                         ooooooooooooooooooo ooooooo o
  32|                     o oooooooooooooooooooo ooooooooooo
  16|                  oooo oooooooooooooooooooo ooooooooooooo
   1| o   o  o ooooooooooooo oooooooooooooooooooo ooooooooooooooooo  o
     ------------------------------------------------------------
     2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
     9 9 9 9 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 5
     4 5 7 9 1 3 5 7 9 1 3 4 6 8 0 2 4 6 8 0 2 3 5 7 9 1 3 5 7 9 1
       . . . . . . . . .   . . . . . . . . .   . . . . . . . . .
       9 8 7 6 5 4 3 2 1   9 8 7 6 5 4 3 2 1   9 8 7 6 5 4 3 2 1


------------------------
|       Summary        |
------------------------
|  observations: 5000  |
|min value: 294.000000 |
|  mean : 328.055000   |
|max value: 351.000000 |
------------------------
```

In this case, the popcount mean is not very large compared to bloom filter length (1024). If the popcount mean is large, you can reduce it by modifying the schema. For more details, please have a look at this tutorial.

Block dataset

Blocking is a technique that makes record linkage scalable. It is achieved by partitioning datasets into groups, called blocks and only comparing records in corresponding blocks. This can reduce the number of comparisons that need to be conducted to find which pairs of records should be linked.

Similar to the hashing above, the blocking is configured with a schema. For this linkage we chose 'lambda-fold' as blocking technique. This blocking method is proposed in paper *An LSH-Based Blocking Approach with a Homomorphic Matching Technique for Privacy-Preserving Record Linkage*. We also provide a detailed explanation of how this blocking method works in this tutorial.

[10]:
```
# NBVAL_IGNORE_OUTPUT
with open("blocking_schema.json") as f:
    print(f.read())
```

```
{
    "type": "lambda-fold",
    "version": 1,
```

(continues on next page)

(continued from previous page)

```
    "config": {
        "blocking-features": [6],
        "Lambda": 3,
        "bf-len": 64,
        "num-hash-funcs": 3,
        "K": 5,
        "input-clks": true,
        "random_state": 0
    }
}
```

Party A Blocks its Data

[11]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink block 'novt_clk_0.json' 'blocking_schema.json' 'novt_blocks_0.json'
```

```
Statistics for the generated blocks:
        Number of Blocks:   96
        Minimum Block Size: 1
        Maximum Block Size: 1125
        Average Block Size: 156.25
        Median Block Size:  92
        Standard Deviation of Block Size:  188.02480239529433
```

Party B Blocks its Data

[12]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink block 'novt_clk_1.json' 'blocking_schema.json' 'novt_blocks_1.json'
```

```
Statistics for the generated blocks:
        Number of Blocks:   96
        Minimum Block Size: 1
        Maximum Block Size: 1125
        Average Block Size: 156.25
        Median Block Size:  85
        Standard Deviation of Block Size:  189.7736601988357
```

Party C Blocks its Data

[13]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink block 'novt_clk_2.json' 'blocking_schema.json' 'novt_blocks_2.json'
```

```
Statistics for the generated blocks:
        Number of Blocks:   96
        Minimum Block Size: 1
        Maximum Block Size: 1133
        Average Block Size: 156.25
        Median Block Size:  98
        Standard Deviation of Block Size:  188.99345574857736
```

Get Ground Truth

[14]:
```
#NBVAL_IGNORE_OUTPUT
truth = []
```

(continues on next page)

(continued from previous page)

```
for party in [0, 1, 2]:
    df = pd.read_csv('data/ncvr_numrec_5000_modrec_2_ocp_20_myp_{}_nump_10.csv'.
↪format(party))
    truth.append(pd.DataFrame({'id{}'.format(party): df.index, 'recid': df['recid']}))

dfj = truth[0].merge(truth[1], on='recid', how='outer')
for df in truth[2:]:
    dfj = dfj.merge(df, on='recid', how='outer')

dfj = dfj.drop(columns=['recid'])
true_matches = set()
for row in dfj.itertuples(index=False):
    cand = [(i, int(x)) for i, x in enumerate(row) if not math.isnan(x)]
    if len(cand) > 1:
        true_matches.add(tuple(cand))

print(f'we have {len(true_matches)} true matches')
e = iter(true_matches)
for i in range(10):
    print(next(e))
```

```
we have 1649 true matches
((0, 159), (1, 169), (2, 169))
((1, 2309), (2, 2137))
((0, 366), (1, 589), (2, 362))
((0, 3719), (1, 3701), (2, 1560))
((0, 182), (1, 758), (2, 760))
((0, 3886), (1, 3865), (2, 311))
((0, 2878), (2, 280))
((0, 498), (1, 2269), (2, 492))
((0, 3630), (2, 282))
((1, 692), (2, 374))
```

Solve with Anonlink

[15]:
```
# NBVAL_IGNORE_OUTPUT
clk_files = ['novt_clk_{}.json'.format(x) for x in range(3)]
block_files = ['novt_blocks_{}.json'.format(x) for x in range(3)]

clk_blocks = []

for i, (clk_f, block_f) in enumerate(zip(clk_files, block_files)):
    print('Combining CLKs and Blocks for Party {}'.format(i))
    clk_blocks.append(json.load(combine_clks_blocks(open(clk_f, 'rb'), open(block_f,
↪'rb')))['clknblocks'])


clk_groups = []
rec_to_blocks = {}

for i, clk_blk in enumerate(clk_blocks):
    clk_groups.append(deserialize_filters([r[0] for r in clk_blk]))
    rec_to_blocks[i] = {rind: clk_blk[rind][1:] for rind in range(len(clk_blk))}
```

```
Combining CLKs and Blocks for Party 0
Combining CLKs and Blocks for Party 1
Combining CLKs and Blocks for Party 2
```

Assess Linkage Quality

We can assess the linkage quality by precision and recall.

- Precision is measured by the proportion of found record groups classified as true matches.
- Recall is measured by the proportion of true matching groups that are classified as found groups

[16]:
```
#NBVAL_IGNORE_OUTPUT
threshold = 0.87

# matching with blocking
found_groups = solve(clk_groups, rec_to_blocks, threshold)
print("Example found groups: ")
for i in range(10):
    print(found_groups[i])
precision, recall = evaluate(found_groups, true_matches)
print('\n\nWith blocking: ')
print(f'precision: {precision}, recall: {recall}')

# matching without blocking
found_groups = naive_solve(clk_groups, threshold)
precision, recall = evaluate(found_groups, true_matches)
print('Without blocking: ')
print(f'precision: {precision}, recall: {recall}')
```

```
Example found groups:
((0, 4963), (1, 4957), (2, 468))
((0, 3632), (1, 3611), (2, 368))
((0, 4502), (2, 1906), (1, 289))
((0, 4959), (1, 4949), (2, 49))
((0, 316), (2, 1251))
((0, 286), (1, 3418))
((0, 4228), (2, 4225))
((0, 1866), (1, 1842), (2, 1871))
((0, 4958), (2, 3852), (1, 233))
((0, 3269), (1, 3237), (2, 3964))


With blocking:
precision: 0.7802981205443941, recall: 0.730139478471801
Without blocking:
precision: 0.7808661926308985, recall: 0.7325651910248635
```

Assess Blocking

Reduction Ratio

Reduction ratio measures the proportion of number of comparisons reduced by using blocking technique. If we have two data providers each has N number of records, then

$$\text{reduction ratio} = 1 - \frac{\text{number of comparisons after blocking}}{N^3}$$

Set Completeness

Set completeness (aka pair completeness in two-party senario) measure how many true matches are maintained after blocking. It is evalauted as

$$\text{set completeness} = \frac{\text{number of true matches after blocking}}{\text{number of all true matches}}$$

[17]:
```
# NBVAL_IGNORE_OUTPUT
block_a = json.load(open('novt_blocks_0.json'))['blocks']
block_b = json.load(open('novt_blocks_1.json'))['blocks']
block_c = json.load(open('novt_blocks_2.json'))['blocks']

filtered_reverse_indices = [block_a, block_b, block_c]
# filtered_reverse_indices[0]
data = []
for party in [0, 1, 2]:
    dfa = pd.read_csv('data/ncvr_numrec_5000_modrec_2_ocp_0_myp_{}_nump_10.csv'.
↪format(party))
    recid = dfa['recid'].values
    data.append(recid)

rr, reduced_num_comparison, naive_num_comparison = reduction_ratio(filtered_reverse_
↪indices, data, K=2)
print('\nWith blocking, we reduced {:,} comparisons to {:,} comparisons i.e. the␣
↪reduction ratio={}'
      .format(naive_num_comparison, reduced_num_comparison, rr))
```

```
With blocking, we reduced 125,000,000,000 comparisons to 3,548,164,581 comparisons i.
↪e. the reduction ratio=0.971614683352
```

[18]:
```
# NBVAL_IGNORE_OUTPUT
sc = set_completeness(filtered_reverse_indices, true_matches, K=2)
print('Set completeness = {}'.format(sc))
```

```
Set completeness = 0.9727107337780473
```

1.1.2 Tutorial for CLI tool `anonlink-client`

For this tutorial we are going to process a data set for private linkage with clkhash using the command line tool `anonlink`.

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).

```
$ pip install -U anonlink-client recordlinkage numpy pandas
```

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

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

Data Exploration

First we have a look at the dataset.

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

dfA.head()
```

[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.

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

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

[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.

[6]:
```
with open("../_static/febrl_schema_v3_overweight.json") as f:
    print(f.read())
```

```
{
  "version": 3,
```

(continues on next page)

(continued from previous page)

```
  "clkConfig": {
    "l": 1024,
    "kdf": {
      "type": "HKDF",
      "hash": "SHA256",
        "info": "c2NoZW1hX2V4YW1wbGU=",
        "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
↪G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",
        "keySize": 64
    }
  },
  "features": [
    {
      "identifier": "rec_id",
      "ignored": true
    },
    {
      "identifier": "given_name",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 300}, "hash": {"type": "doubleHash"} }
    },
    {
      "identifier": "surname",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 300}, "hash": {"type": "doubleHash"} }
    },
    {
      "identifier": "street_number",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature": 300}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "address_1",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  300} }
    },
    {
      "identifier": "address_2",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  300} }
    },
    {
      "identifier": "suburb",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  300} }
    },
    {
      "identifier": "postcode",
      "format": { "type": "integer", "minimum": 100, "maximum": 9999 },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  300} }
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "identifier": "state",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 3 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 300} }
    },
    {
      "identifier": "date_of_birth",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  300}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "soc_sec_id",
      "ignored": true
    }
  ]
}
```

Validate the schema

The command line tool can check that the linkage schema is valid:

[7]:
```
!anonlink validate-schema "../_static/febrl_schema_v3_overweight.json"
```

```
schema is valid
```

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 secret 'secret', for real data, make sure that the secret contains enough entropy, as knowledge of this secret is sufficient to reconstruct the PII information from a CLK!

Also, **do not share these keys with anyone, except the other participating party.**

[8]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink hash "PII_a.csv" secret "../_static/febrl_schema_v3_overweight.json" "clks_
↪a.json"
```

```
CLK data written to clks_a.json
```

Inspect the output

clkhash has hashed the PII, creating a Cryptographic Longterm Key for each entity. The stats output shows that the mean popcount (number of bits set) is quite high (949 out of 1024) which can effect accuracy.

You can reduce the popcount by modify the *'strategy'* 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).

[9]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink describe "clks_a.json"
```

```
    ----------------------------------------------------------------------------------------
↪---------------------------------------
    |                                                          popcounts                    ␣
↪                                        |
    ----------------------------------------------------------------------------------------
↪---------------------------------------

 593|                                                          o
 562|                                                          o
 531|                                                          o
 500|                                                        o  o
 469|                                                        o  o
 437|                                                        o oo
 406|                                                        o oo
 375|                                                        oooo o
 344|                                                        oooooo
 313|                                                        oooooo
 281|                                                      o oooooo
 250|                                                      oooooooo
 219|                                                      oooooooo
 188|                                                      oooooooo
 157|                                                    o oooooooooo
 125|                                                  o  ooooooooooo
  94|                                               o oo oooooooooooooo
  63|                                             ooooooooooooooooooooo
  32|                                           ooooooooooooooooooooooooo
   1| o  o     o       oooooooooooooooooooooooooooooooooooooooooooooooo
     ------------------------------------------------------------------
     8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
     3 3 3 4 4 5 5 6 6 7 7 8 8 9 9 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7
     0 4 9 4 9 4 9 3 8 3 8 3 8 2 7 2 7 2 7 1 6 1 6 1 6 0 5 0 5 0
       . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       8 6 5 3 1 0 8 6 5 3 1 0 8 6 5 3 1 0 8 6 5 3 1 0 8 6 5 3 1

--------------------------
|        Summary         |
--------------------------
|   observations: 5000   |
| min value: 830.000000  |
|   mean : 944.245800    |
| max value: 975.000000  |
--------------------------
```

First, we will reduce the value of *bits_per_feature* for each feature.

[10]:
```
with open("../_static/febrl_schema_v3_reduced.json") as f:
    print(f.read())
```

```
{
  "version": 3,
  "clkConfig": {
    "l": 1024,
    "kdf": {
      "type": "HKDF",
      "hash": "SHA256",
```

(continues on next page)

(continued from previous page)

```
        "info": "c2NoZW1hX2V4YW1wbGU=",
        "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
↪G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",
        "keySize": 64
    }
  },
  "features": [
    {
      "identifier": "rec_id",
      "ignored": true
    },
    {
      "identifier": "given_name",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 200}, "hash": {"type": "doubleHash"} }
    },
    {
      "identifier": "surname",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 200}, "hash": {"type": "doubleHash"} }
    },
    {
      "identifier": "street_number",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature": 200}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "address_1",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  200} }
    },
    {
      "identifier": "address_2",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  200} }
    },
    {
      "identifier": "suburb",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  200} }
    },
    {
      "identifier": "postcode",
      "format": { "type": "integer", "minimum": 100, "maximum": 9999 },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  200} }
    },
    {
      "identifier": "state",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 3 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 200} }
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "identifier": "date_of_birth",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  200}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "soc_sec_id",
      "ignored": true
    }
  ]
}
```

[11]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink hash "PII_a.csv" secret "../_static/febrl_schema_v3_reduced.json" "clks_a.
↪json"
```

```
CLK data written to clks_a.json
```

And now we will modify the `bits_per_feature` values again, this time de-emphasising the contribution of the address related columns.

[12]:
```
with open("../_static/febrl_schema_v3_final.json") as f:
    print(f.read())
```

```
{
  "version": 3,
  "clkConfig": {
    "l": 1024,
    "kdf": {
      "type": "HKDF",
      "hash": "SHA256",
        "info": "c2NoZW1hX2V4YW1wbGU=",
        "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
↪G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",
        "keySize": 64
    }
  },
  "features": [
    {
      "identifier": "rec_id",
      "ignored": true
    },
    {
      "identifier": "given_name",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 200}, "hash": {"type": "doubleHash"} }
    },
    {
      "identifier": "surname",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 200}, "hash": {"type": "doubleHash"} }
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "identifier": "street_number",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature": 100}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "address_1",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  100} }
    },
    {
      "identifier": "address_2",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  100} }
    },
    {
      "identifier": "suburb",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  100} }
    },
    {
      "identifier": "postcode",
      "format": { "type": "integer", "minimum": 50, "maximum": 9999 },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  100} }
    },
    {
      "identifier": "state",
      "format": { "type": "string", "encoding":  "utf-8"},
      "hashing": {"comparison": {"type": "ngram", "n": 2, "positional": true},
↪"strategy": {"bitsPerFeature": 100}, "missingValue": {"sentinel":  ""}
      }
    },
    {
      "identifier": "date_of_birth",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  200}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "soc_sec_id",
      "ignored": true
    }
  ]
}
```

[13]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink hash "PII_a.csv" secret "../_static/febrl_schema_v3_final.json" "clks_a.json
↪"
```

```
CLK data written to clks_a.json
```

Great, now approximately half the bits are set in each CLK.

Each CLK is serialized in a JSON friendly base64 format:

[14]:
```
# If you have jq tool installed:
#!jq .clks[0] clks_a.json

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

[14]:
```
'eliv99lhdvGu27399h/5bV+NHSvr+Yf/EObeO/+32f9RsWvu/0Y1f3Jvyvj+12pp9De18P9dSA8/
↪3xztXqiTXvt/+pFVb3+vVeRiR3+Z//X3v9XzE/9/u/X//6P9qMumsbnl+f1y9U93ON+99f6Pf5WX13zR/nN/
↪0/9yo//v2Hk='
```

Hash data set B

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

[15]:
```
# NBVAL_IGNORE_OUTPUT
dfB.to_csv("PII_b.csv")

!anonlink hash "PII_b.csv" secret "../_static/febrl_schema_v3_final.json" "clks_b.json
↪"
```

```
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 'groups'. 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:

[16]:
```
# NBVAL_IGNORE_OUTPUT
SERVER = 'https://anonlink-client-ci.easd.data61.xyz/'
!anonlink status --server={SERVER}
```

```
{"project_count": 3, "rate": 2205592, "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.

[17]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink create-project --server={SERVER} --schema "../_static/febrl_schema_v3_final.
↪json" --output "credentials.json" --type "groups" --name "tutorial"
```

```
Project created
```

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

[18]:
```
# NBVAL_IGNORE_OUTPUT
credentials = json.load(open("credentials.json", 'rt'))
print(json.dumps(credentials, indent=4))
```

```
{
    "project_id": "e3603d3ff21199bc76e441f1fc21fa352b7b723a42566af2",
    "result_token": "aa5dd1bc498c24faff7108f0912192e7576cf38e20a07be4",
    "update_tokens": [
        "e3c98e45c3b50c7a23e2ba89e9d13071f9903bc0bb1bac09",
        "c82bca998aec6c2c5b04d5d39e3dde9d496dddbb5fed57d7"
    ]
}
```

Uploading the CLKs to the entity service

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

By default, data is uploaded to object store, if this feature is enabled in the entity service, since it has a more reliable uploading mechanism especially with large size datasets. You can also specify to upload to entity service directly by adding the flag `--to_entityservice` to the command `upload`. A message will be printed to tell users if the encodings and blocks are uploaded to object store:

```
Anonlink client: Uploading to the external object store - MINIO
```

or they are uploaded to entity service:

```
Anonlink client: Uploading to entity service
```

Note that the progress bar is only available when uploading to object store. Uploading to entity service is performed with requests which does not support streaming upload yet.

[19]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink upload \
       --project="{credentials['project_id']}" \
        --apikey="{credentials['update_tokens'][0]}" \
        --output "upload_a.json" \
        --server="{SERVER}" \
        "clks_a.json"
```

```
uploads
Anonlink client: Uploading to the external object store - MINIO
Upload clks_a.json: |####################| 0.84 MB/0.84 MB 100% [elapsed: 00:00 left:␣
↪00:00,  1.22 MB/sec]
```

[20]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink upload \
       --project="{credentials['project_id']}" \
        --apikey="{credentials['update_tokens'][1]}" \
        --output "upload_b.json" \
        --server="{SERVER}" \
       "clks_b.json"
```

```
uploads
Anonlink client: Uploading to the external object store - MINIO
Upload clks_b.json: |####################| 0.84 MB/0.84 MB 100% [elapsed: 00:00 left:␣
↪00:00,  1.59 MB/sec]
```

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`:

[21]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink create --verbose  \
    --server="{SERVER}" \
    --output "run_info.json" \
    --threshold=0.9 \
    --project="{credentials['project_id']}" \
    --apikey="{credentials['result_token']}" \
    --name="CLI tutorial run A"
```

```
Connecting to Entity Matching Server: https://anonlink-client-ci.easd.data61.xyz/
```

[22]:
```
# NBVAL_IGNORE_OUTPUT
run_info = json.load(open("run_info.json", 'rt'))
run_info
```

[22]:
```
{'name': 'CLI tutorial run A',
 'notes': 'Run created by anonlink-client 0.1.2',
 'run_id': '7f0e4570d6d36286cd78f3f5b325adbf1d5e024f82df3803',
 'threshold': 0.9}
```

Results

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

[23]:
```
!anonlink results --watch \
        --project="{credentials['project_id']}" \
        --apikey="{credentials['result_token']}" \
        --run="{run_info['run_id']}" \
        --server="{SERVER}" \
        --output results.txt
```

```
State: running
Stage (2/3): compute similarity scores
Progress: 100.00%
State: running
Stage (2/3): compute similarity scores
Progress: 100.00%
State: running
Stage (3/3): compute output
State: completed
Stage (3/3): compute output
Downloading result
Received result
```

[24]:
```
def extract_matches(file):
    with open(file, 'rt') as f:
        results = json.load(f)['groups']
    # each entry in `results` looks like this: '((0, 4039), (1, 2689))'.
```

(continues on next page)

(continued from previous page)

```
    # The format is ((dataset_id, row_id), (dataset_id, row_id))
    # As we only have two parties in this example, we can remove the dataset_ids.
    # Also, turning the solution into a set will make it easier to assess the
    # quality of the matching.
    found_matches = set((a, b) for ((_, a), (_, b)) in results)
    print('The service linked {} entities.'.format(len(found_matches)))
    return found_matches

found_matches = extract_matches('results.txt')
```

```
The service linked 4051 entities.
```

Let's investigate some of those matches and the overall matching quality. In this case we have the ground truth so we can compute the precision and recall.

Fortunately, the febrl4 datasets contain record ids which tell us the correct linkages. Using this information we are able to create a set of the true matches.

[25]:
```
# rec_id in dfA has the form 'rec-1070-org'. We only want the number. Additionally,␣
↪as we are
# interested in the position of the records, we create a new index which contains the␣
↪row numbers.
dfA_ = dfA.rename(lambda x: x[4:-4], axis='index').reset_index()
dfB_ = dfB.rename(lambda x: x[4:-6], axis='index').reset_index()
# now we can merge dfA_ and dfB_ on the record_id.
a = pd.DataFrame({'ida': dfA_.index, 'rec_id': dfA_['rec_id']})
b = pd.DataFrame({'idb': dfB_.index, 'rec_id': dfB_['rec_id']})
dfj = a.merge(b, on='rec_id', how='inner').drop(columns=['rec_id'])
# and build a set of the corresponding row numbers.
true_matches = set((row[0], row[1]) for row in dfj.itertuples(index=False))
```

[26]:
```
def describe_matching_quality(found_matches, show_examples=False):
    if show_examples:
        print('idx_a, idx_b,     rec_id_a,       rec_id_b')
        print('-----------------------------------------')
        for a_i, b_i in itertools.islice(found_matches, 10):
            print('{:3}, {:6}, {:>15}, {:>15}'.format(a_i+1, b_i+1, a.iloc[a_i]['rec_
↪id'], b.iloc[b_i]['rec_id']))
        print('-----------------------------------------')

    tp = len(found_matches & true_matches)
    fp = len(found_matches - true_matches)
    fn = len(true_matches - found_matches)

    precision = tp / (tp + fp)
    recall = tp / (tp + fn)


    print('Precision: {:.2f}, Recall: {:.2f}'.format(precision, recall))
```

[27]:
```
# NBVAL_IGNORE_OUTPUT
describe_matching_quality(found_matches, True)
```

```
idx_a, idx_b,     rec_id_a,       rec_id_b
-----------------------------------------
3170,    259,            3730,            3730
733,   2003,           4239,           4239
```

(continues on next page)

(continued from previous page)

```
1685,    3323,            2888,            2888
4550,    3627,            4216,            4216
1875,    2991,            4391,            4391
3928,    2377,            3493,            3493
4928,    4656,             276,             276
334,     945,            4848,            4848
2288,    4331,            3491,            3491
4088,    2454,            1850,            1850
-----------------------------------------------
Precision: 1.00, Recall: 0.81
```

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 81%.

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

[28]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink create --verbose  \
    --server="{SERVER}" \
    --output "run_info.json" \
    --threshold=0.8 \
    --project="{credentials['project_id']}" \
    --apikey="{credentials['result_token']}" \
    --name="CLI tutorial run B"

run_info = json.load(open('run_info.json', 'rt'))
```

```
Connecting to Entity Matching Server: https://anonlink-client-ci.easd.data61.xyz/
```

[29]:
```
!anonlink results --watch \
        --project="{credentials['project_id']}" \
        --apikey="{credentials['result_token']}" \
        --run="{run_info['run_id']}" \
        --server="{SERVER}" \
        --output results.txt
```

```
State: running
Stage (2/3): compute similarity scores
Progress: 100.00%
State: running
Stage (2/3): compute similarity scores
Progress: 100.00%
State: running
Stage (3/3): compute output
State: completed
Stage (3/3): compute output
Downloading result
Received result
```

[30]:
```
found_matches = extract_matches('results.txt')

describe_matching_quality(found_matches)
```

```
The service linked 4962 entities.
Precision: 1.00, Recall: 0.99
```

Great, for this threshold value we get a precision of 100% and a recall of 99%.

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 80% of the derived CLK pairs overlap more than 90% (the first threshold). Whereas 99% of 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 (if an entry in dataset A has no match in dataset B, but we keep reducing the threshold, eventually a comparison with an entry in B will be above the threshold leading to a false match). 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. However, as our example dataset only contains matches (every entry in A has a match in B), this phenomenon cannot be observered. With the threshold `0.72` we identify all matches but one correctly (at the cost of a longer execution time).

[31]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink create --verbose  \
    --server="{SERVER}" \
    --output "run_info.json" \
    --threshold=0.72 \
    --project="{credentials['project_id']}" \
    --apikey="{credentials['result_token']}" \
    --name="CLI tutorial run B"

run_info = json.load(open("run_info.json", 'rt'))
```

```
Connecting to Entity Matching Server: https://anonlink-client-ci.easd.data61.xyz/
```

[32]:
```
!anonlink results --watch \
        --project="{credentials['project_id']}" \
        --apikey="{credentials['result_token']}" \
        --run="{run_info['run_id']}" \
        --server="{SERVER}" \
        --output results.txt
```

```
State: running
Stage (2/3): compute similarity scores
State: running
Stage (2/3): compute similarity scores
State: running
Stage (2/3): compute similarity scores
Progress: 100.00%
State: running
Stage (3/3): compute output
State: completed
Stage (3/3): compute output
Downloading result
Received result
```

[33]:
```
found_matches = extract_matches('results.txt')

describe_matching_quality(found_matches)
```

```
The service linked 4998 entities.
Precision: 1.00, Recall: 1.00
```

It is important to choose an appropriate threshold for the amount of perturbations present in the data.

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

Cleanup

Finally to remove the results from the service delete the individual runs, or remove the uploaded data and all runs by deleting the entire project.

[34]:
```
# NBVAL_IGNORE_OUTPUT
# Deleting a run
!anonlink delete --project="{credentials['project_id']}" \
        --apikey="{credentials['result_token']}" \
        --run="{run_info['run_id']}" \
        --server="{SERVER}"
```

```
Run deleted
```

[35]:
```
# NBVAL_IGNORE_OUTPUT
# Deleting a project
!anonlink delete-project --project="{credentials['project_id']}" \
        --apikey="{credentials['result_token']}" \
        --server="{SERVER}"
```

```
Project deleted
```

[ ]:
```
```

1.1.3 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 anonlink-client, clkhash, recordlinkage and a few data science tools (pandas and numpy):

```
$ pip install -U anonlink-client clkhash anonlink recordlinkage numpy pandas
```

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

[2]:
```
# import modules necessary for schema definition
import clkhash
from clkhash.field_formats import *
from clkhash.schema import Schema
from clkhash.comparators import NgramComparison

from anonlinkclient.utils import generate_clk_from_csv, generate_candidate_blocks_
↪from_csv, combine_clks_blocks
```

[3]:
```
import recordlinkage
from recordlinkage.datasets import load_febrl4
```

Data Exploration

First we have a look at the dataset.

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

dfA.head()
```

[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.

[5]:
```
dfA.columns
```

[5]:
```
Index(['given_name', 'surname', 'street_number', 'address_1', 'address_2',
       'suburb', 'postcode', 'state', 'date_of_birth', 'soc_sec_id'],
      dtype='object')
```

[6]:
```
a_csv = io.StringIO()
dfA.to_csv(a_csv)
```

Hashing Schema Definition

A hashing schema instructs anonlink-client 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.

[7]:
```
fields = [
    Ignore('rec_id'),
    StringSpec('given_name', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(300))),
    StringSpec('surname', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(300))),
    IntegerSpec('street_number', FieldHashingProperties(comparator=NgramComparison(1,␣
↪True), strategy=BitsPerFeatureStrategy(300), missing_
↪value=MissingValueSpec(sentinel=''))),
```

(continues on next page)

(continued from previous page)

```
    StringSpec('address_1', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(300))),
    StringSpec('address_2', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(300))),
    StringSpec('suburb', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(300))),
    IntegerSpec('postcode', FieldHashingProperties(comparator=NgramComparison(1,␣
↪True), strategy=BitsPerFeatureStrategy(300))),
    StringSpec('state', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(300))),
    IntegerSpec('date_of_birth', FieldHashingProperties(comparator=NgramComparison(1,␣
↪True), strategy=BitsPerFeatureStrategy(300), missing_
↪value=MissingValueSpec(sentinel=''))),
    Ignore('soc_sec_id')
]

schema = Schema(fields, 1024)
```

Hash the data

We can now hash our PII data from the CSV file using our defined schema. We must provide a *secret* to this command - this secret has to be used by both parties hashing data. For this toy example we will use the secret *'secret'*, for real data, make sure that the key contains enough entropy, as knowledge of this secret is sufficient to reconstruct the PII information from a CLK!

Also, **do not share this secret with anyone, except the other participating party.**

[8]:
```
secret = 'secret'
```

[9]:
```
# NBVAL_IGNORE_OUTPUT
a_csv.seek(0)
clks_a = generate_clk_from_csv(a_csv, secret, schema)
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:03<00:00, 1.58kclk/s, mean=944, std=14.4]
```

Inspect the output

anonlink-client 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 (950 out of 1024) which can affect accuracy.

We can control the popcount by adjusting the hashing strategy. There are currently two different strategies implemented in the library. - *BitsPerToken*: each token of a feature's value is inserted into the CLK *bits_per_token* times. Increasing *bits_per_token* will give the corresponding feature more importance in comparisons, decreasing *bits_per_token* will de-emphasise columns which are less suitable for linkage (e.g. information that changes frequently). The *BitsPerToken* strategy is set with the 'strategy=BitsPerTokenStrategy(bits_per_token=30)' argument for each feature's FieldHashingProperties. (for a total of numberOfTokens * 30 insertions) - *BitsPerFeature*: In this strategy we always insert a fixed number of bits into the CLK for a feature, irrespective of the number of tokens. This strategy is set with the 'strategy=BitsPerFeatureStrategy(bits_per_feature=100)' argument for each feature's FieldHashingProperties.

In this example, we will reduce the value of `bits_per_feature` for address related columns.

[10]:
```
# NBVAL_IGNORE_OUTPUT
fields = [
    Ignore('rec_id'),
    StringSpec('given_name', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(200))),
    StringSpec('surname', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(200))),
    IntegerSpec('street_number', FieldHashingProperties(comparator=NgramComparison(1,␣
↪True), strategy=BitsPerFeatureStrategy(100), missing_
↪value=MissingValueSpec(sentinel=''))),
    StringSpec('address_1', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(100))),
    StringSpec('address_2', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(100))),
    StringSpec('suburb', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(100))),
    IntegerSpec('postcode', FieldHashingProperties(comparator=NgramComparison(1,␣
↪True), strategy=BitsPerFeatureStrategy(100))),
    StringSpec('state', FieldHashingProperties(comparator=NgramComparison(2),␣
↪strategy=BitsPerFeatureStrategy(100))),
    IntegerSpec('date_of_birth', FieldHashingProperties(comparator=NgramComparison(1,␣
↪True), strategy=BitsPerFeatureStrategy(200), missing_
↪value=MissingValueSpec(sentinel=''))),
    Ignore('soc_sec_id')
]

schema = Schema(fields, 1024)
a_csv.seek(0)
clks_a = generate_clk_from_csv(a_csv, secret, schema)
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:01<00:00, 2.87kclk/s, mean=696, std=22.7]
```

Each CLK is respresented as a bitarray.

[11]:
```
clks_a[0]
```

[11]:
```
bitarray(
↪'1111111100101100001100011011110111100111001111111000111110010100011101111111111110111000110111111
↪')
```

Hash data set B

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

[12]:
```
# NBVAL_IGNORE_OUTPUT
b_csv = io.StringIO()
dfB.to_csv(b_csv)
b_csv.seek(0)
clks_b = generate_clk_from_csv(b_csv, secret, schema)
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:01<00:00, 2.92kclk/s, mean=687, std=30.4]
```

[13]:
```
len(clks_b)
```

[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.

[14]:
```
from anonlinkclient.utils import deserialize_filters
from bitarray import bitarray
import base64
```

Using `anonlink` we find the candidate pairs - which is all possible pairs above the given `threshold`. Then we solve for the most likely mapping.

[15]:
```
import anonlink

def mapping_from_clks(clks_a, clks_b, threshold):
    results_candidate_pairs = anonlink.candidate_generation.find_candidate_pairs(
            [clks_a, clks_b],
            anonlink.similarities.dice_coefficient,
            threshold
    )
    solution = anonlink.solving.greedy_solve(results_candidate_pairs)
    print('Found {} matches'.format(len(solution)))
    # each entry in `solution` looks like this: '((0, 4039), (1, 2689))'.
    # The format is ((dataset_id, row_id), (dataset_id, row_id))
    # As we only have two parties in this example, we can remove the dataset_ids.
    # Also, turning the solution into a set will make it easier to assess the
    # quality of the matching.
    return set((a, b) for ((_, a), (_, b)) in solution)
```

[16]:
```
found_matches = mapping_from_clks(clks_a, clks_b, 0.9)
```

```
Found 4049 matches
```

Evaluate matching quality

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

Fortunately, the febrl4 datasets contain record ids which tell us the correct linkages. Using this information we are able to create a set of the true matches.

[17]:
```
# rec_id in dfA has the form 'rec-1070-org'. We only want the number. Additionally,␣
↪as we are
# interested in the position of the records, we create a new index which contains the␣
↪row numbers.
dfA_ = dfA.rename(lambda x: x[4:-4], axis='index').reset_index()
dfB_ = dfB.rename(lambda x: x[4:-6], axis='index').reset_index()
# now we can merge dfA_ and dfB_ on the record_id.
a = pd.DataFrame({'ida': dfA_.index, 'rec_id': dfA_['rec_id']})
b = pd.DataFrame({'idb': dfB_.index, 'rec_id': dfB_['rec_id']})
dfj = a.merge(b, on='rec_id', how='inner').drop(columns=['rec_id'])
# and build a set of the corresponding row numbers.
true_matches = set((row[0], row[1]) for row in dfj.itertuples(index=False))
```

[18]:
```
def describe_matching_quality(found_matches, show_examples=False):
    if show_examples:
        print('idx_a, idx_b,     rec_id_a,       rec_id_b')
        print('-----------------------------------------------')
        for a_i, b_i in itertools.islice(found_matches, 10):
            print('{:4d}, {:5d}, {:>11}, {:>14}'.format(a_i+1, b_i+1, a.iloc[a_i][
↪'rec_id'], b.iloc[b_i]['rec_id']))
        print('-----------------------------------------------')

    tp = len(found_matches & true_matches)
    fp = len(found_matches - true_matches)
    fn = len(true_matches - found_matches)

    precision = tp / (tp + fp)
    recall = tp / (tp + fn)

    print('Precision: {:.3f}, Recall: {:.3f}'.format(precision, recall))
```

[19]:
```
# NBVAL_IGNORE_OUTPUT
describe_matching_quality(found_matches, show_examples=True)
```

```
idx_a, idx_b,     rec_id_a,       rec_id_b
-----------------------------------------------
3170,   259,        3730,           3730
 733,  2003,        4239,           4239
1685,  3323,        2888,           2888
4550,  3627,        4216,           4216
1875,  2991,        4391,           4391
3928,  2377,        3493,           3493
4928,  4656,         276,            276
2288,  4331,        3491,           3491
 334,   945,        4848,           4848
4088,  2454,        1850,           1850
-----------------------------------------------
Precision: 1.000, Recall: 0.810
```

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 81%.

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

[20]:
```
found_matches = mapping_from_clks(clks_a, clks_b, 0.8)
describe_matching_quality(found_matches)
```

```
Found 4962 matches
Precision: 1.000, Recall: 0.992
```

Great, for this threshold value we get a precision of 100% and a recall of 99.2%.

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. It is important to choose an appropriate threshold for the amount of perturbations present in the data (a threshold of 0.72 and below generates an almost perfect mapping with little mistakes).

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

[ ]:

1.1.4 Client side encoding with blocking and server side linking

For this tutorial we are going to encode a data set for private record linkage using the command line tool `anonlink`.

Note you can also directly 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).

[ ]:
```
# NBVAL_IGNORE_OUTPUT
!pip install -U anonlink-client recordlinkage numpy pandas metaphone matplotlib
```

[11]:
```
%matplotlib inline
import pandas as pd
import itertools
from clkhash.describe import get_encoding_popcounts
from clkhash.serialization import deserialize_bitarray
import json
import matplotlib.pyplot as plt
from recordlinkage.datasets import load_febrl4
```

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

dfA.head()
```

[4]:

rec_id	given_name	surname	street_number	address_1	address_2	suburb	postcode	state	date_of_birth	soc_sec_id
rec-1070-org	michaela	neumann	8	stanley street	miami	winston hills	4223	nsw	19151111	5304218
rec-1016-org	courtney	painter	12	pinkerton circuit	bega flats	richlands	4560	vic	19161214	4066625
rec-4405-org	charles	green	38	salkauskas crescent	kela	dapto	4566	nsw	19480930	4365168
rec-1288-org	vanessa	parr	905	macquoid place	broadbridge manor	south grafton	2135	sa	19951119	9239102
rec-3585-org	mikayla	malloney	37	randwick road	avalind	hoppers crossing	4552	vic	19860208	7207688

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

[5]:
```
dfA.columns
```

[5]:
```
Index(['given_name', 'surname', 'street_number', 'address_1', 'address_2',
       'suburb', 'postcode', 'state', 'date_of_birth', 'soc_sec_id'],
      dtype='object')
```

[6]:
```
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.

[7]:
```
with open("../_static/febrl_schema_v3_overweight.json") as f:
    print(f.read())
```

```
{
  "version": 3,
  "clkConfig": {
    "l": 1024,
    "kdf": {
      "type": "HKDF",
      "hash": "SHA256",
        "info": "c2NoZW1hX2V4YW1wbGU=",
        "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
↪G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",
        "keySize": 64
    }
  },
  "features": [
    {
      "identifier": "rec_id",
      "ignored": true
    },
    {
      "identifier": "given_name",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 300}, "hash": {"type": "doubleHash"} }
    },
    {
      "identifier": "surname",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 300}, "hash": {"type": "doubleHash"} }
    },
    {
      "identifier": "street_number",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature": 300}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "address_1",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  300} }
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "identifier": "address_2",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  300} }
    },
    {
      "identifier": "suburb",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  300} }
    },
    {
      "identifier": "postcode",
      "format": { "type": "integer", "minimum": 100, "maximum": 9999 },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  300} }
    },
    {
      "identifier": "state",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 3 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 300} }
    },
    {
      "identifier": "date_of_birth",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  300}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "soc_sec_id",
      "ignored": true
    }
  ]
}
```

Validate the schema

The command line tool can check that the linkage schema is valid:

[8]:
```
!anonlink validate-schema "../_static/febrl_schema_v3_overweight.json"
```

```
schema is valid
```

Encode the data

We can now encode our Personally Identifiable Information (PII) data from the CSV file using our defined linkage schema. We must provide a *secret key* to this command - this secret has to be used by all parties contributing data. For this toy example we will use the secret `horse staple battery`, for real data, make sure that the secret contains enough entropy, as knowledge of this secret is sufficient to reconstruct the PII information from a CLK!

Also, **do not share these keys with anyone, except the other participating party.**

[9]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink hash "PII_a.csv" "horse staple battery" "../_static/febrl_schema_v3_
↪overweight.json" "clks_a.json" -v
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:03<00:00, 1.45kclk/s, mean=944, std=14.
CLK data written to clks_a.json
```

Inspect the output

The PII has now been encoded, anonlink-client has used the `clkhash` library to create a *Cryptographic Longterm Key* for each entity. The stats output shows that the mean number of bits set is quite high (more than 900 out of 1024) which can effect accuracy.

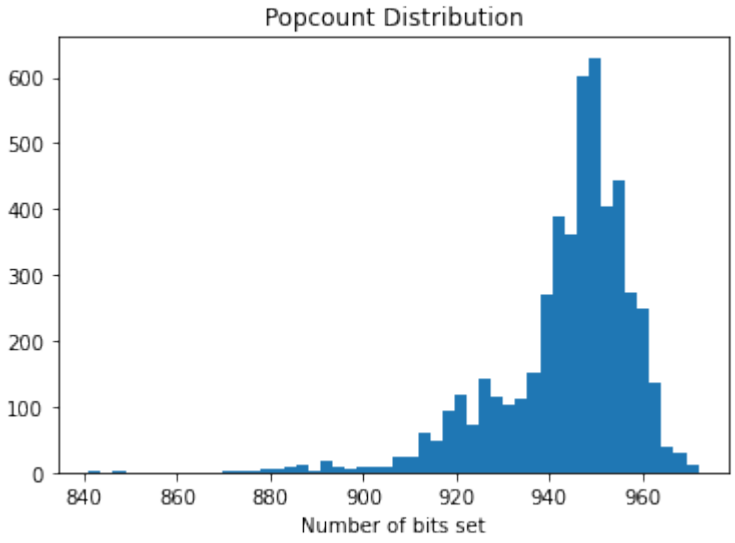
To fix this, we can reduce the popcount by modifying the *'strategy'* for different fields.

This can be used to de-emphasise columns which are less suitable for linkage (e.g. information that changes frequently).

[13]:
```
def show_clk_popcounts(filename):
    with open(filename) as f:
        serialized_clks = json.load(f)['clks']
    popcounts = get_encoding_popcounts([deserialize_bitarray(clk) for clk in␣
↪serialized_clks])

    plt.hist(popcounts, bins=50)
    plt.title("Popcount Distribution")
    plt.xlabel("Number of bits set")

show_clk_popcounts("clks_a.json")
```



First, we will reduce the value of *bits_per_feature* for each feature.

[14]:
```
with open("../_static/febrl_schema_v3_reduced.json") as f:
    print(f.read())
```

```
{
  "version": 3,
  "clkConfig": {
```

(continues on next page)

(continued from previous page)

```
    "l": 1024,
    "kdf": {
      "type": "HKDF",
      "hash": "SHA256",
        "info": "c2NoZW1hX2V4YW1wbGU=",
        "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
↪G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",
        "keySize": 64
    }
  },
  "features": [
    {
      "identifier": "rec_id",
      "ignored": true
    },
    {
      "identifier": "given_name",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 200}, "hash": {"type": "doubleHash"} }
    },
    {
      "identifier": "surname",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 200}, "hash": {"type": "doubleHash"} }
    },
    {
      "identifier": "street_number",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature": 200}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "address_1",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  200} }
    },
    {
      "identifier": "address_2",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  200} }
    },
    {
      "identifier": "suburb",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  200} }
    },
    {
      "identifier": "postcode",
      "format": { "type": "integer", "minimum": 100, "maximum": 9999 },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  200} }
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "identifier": "state",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 3 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 200} }
    },
    {
      "identifier": "date_of_birth",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  200}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "soc_sec_id",
      "ignored": true
    }
  ]
}
```

[15]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink hash "PII_a.csv" secret "../_static/febrl_schema_v3_reduced.json" "clks_a.
↪json" -v
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:02<00:00, 1.67kclk/s, mean=839, std=20.
CLK data written to clks_a.json
```

And now we will modify the `bits_per_feature` values again, this time de-emphasising the contribution of the address related columns.

[16]:
```
with open("../_static/febrl_schema_v3_final.json") as f:
    print(f.read())
```

```
{
  "version": 3,
  "clkConfig": {
    "l": 1024,
    "kdf": {
      "type": "HKDF",
      "hash": "SHA256",
        "info": "c2NoZW1hX2V4YW1wbGU=",
        "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
↪G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",
        "keySize": 64
    }
  },
  "features": [
    {
      "identifier": "rec_id",
      "ignored": true
    },
    {
      "identifier": "given_name",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 200}, "hash": {"type": "doubleHash"} }
    },
```

(continues on next page)

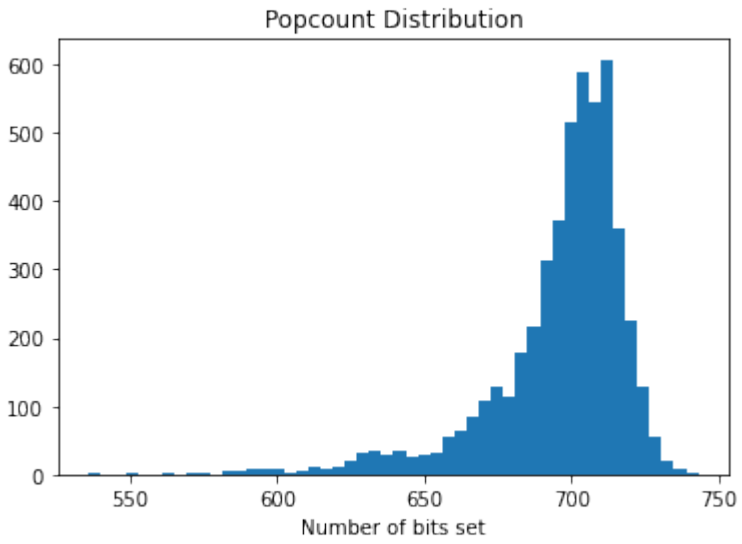
(continued from previous page)

```
    {
      "identifier": "surname",
      "format": { "type": "string", "encoding": "utf-8", "maxLength": 64 },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature": 200}, "hash": {"type": "doubleHash"} }
    },
    {
      "identifier": "street_number",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature": 100}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "address_1",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  100} }
    },
    {
      "identifier": "address_2",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  100} }
    },
    {
      "identifier": "suburb",
      "format": { "type": "string", "encoding": "utf-8" },
      "hashing": { "comparison": {"type": "ngram", "n": 2}, "strategy": {
↪"bitsPerFeature":  100} }
    },
    {
      "identifier": "postcode",
      "format": { "type": "integer", "minimum": 50, "maximum": 9999 },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  100} }
    },
    {
      "identifier": "state",
      "format": { "type": "string", "encoding":  "utf-8"},
      "hashing": {"comparison": {"type": "ngram", "n": 2, "positional": true},
↪"strategy": {"bitsPerFeature": 100}, "missingValue": {"sentinel":  ""}
      }
    },
    {
      "identifier": "date_of_birth",
      "format": { "type": "integer" },
      "hashing": { "comparison": {"type": "ngram", "n": 1, "positional": true},
↪"strategy": {"bitsPerFeature":  200}, "missingValue": {"sentinel": ""} }
    },
    {
      "identifier": "soc_sec_id",
      "ignored": true
    }
  ]
}
```

[17]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink hash "PII_a.csv" secret "../_static/febrl_schema_v3_final.json" "clks_a.json
↪" -v
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:02<00:00, 2.12kclk/s, mean=696, std=23.
CLK data written to clks_a.json
```

[18]:
```
show_clk_popcounts("clks_a.json")
```



Great, now approximately half the bits are set in each CLK.

Each CLK is serialized in a JSON friendly base64 format:

Hash data set B

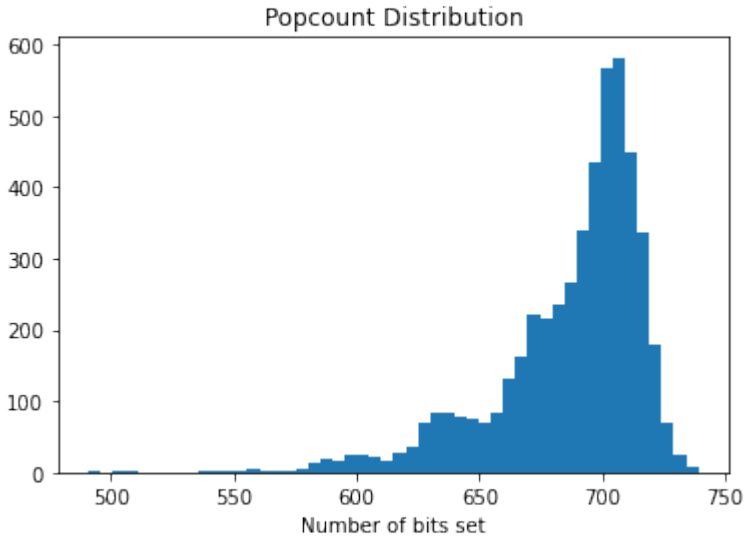
Now we hash the second dataset using the same keys and same schema.

[19]:
```
# NBVAL_IGNORE_OUTPUT
dfB.to_csv("PII_b.csv")

!anonlink hash "PII_b.csv" secret "../_static/febrl_schema_v3_final.json" "clks_b.json
↪" -v
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:02<00:00, 1.88kclk/s, mean=687, std=30.
CLK data written to clks_b.json
```

[20]:
```
show_clk_popcounts("clks_b.json")
```



1.1.5 Blocking

We create a blocking schema to group the encodings into similar blocks on the client side. The central linkage service will then compare all encodings in each block, efficient blocking can drastically reduce the total number of encoding comparisons the central linkage service must carry out.

[21]:
```
# NBVAL_IGNORE_OUTPUT
with open("blocking_schema.json", 'wt') as f:
    f.write("""
{
    "type": "lambda-fold",
    "version": 1,
    "config": {
        "blocking-features": [6],
        "Lambda": 3,
        "bf-len": 64,
        "num-hash-funcs": 3,
        "K": 5,
        "input-clks": true,
        "random_state": 0
    }
}
    """)
```

[22]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink block 'clks_a.json' 'blocking_schema.json' 'blocks_a.json'
```

```
Statistics for the generated blocks:
        Number of Blocks:   96
        Minimum Block Size: 2
        Maximum Block Size: 1152
        Average Block Size: 156.25
        Median Block Size:  85
        Standard Deviation of Block Size:  188.9700616778391
```

[23]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink block 'clks_b.json' 'blocking_schema.json' 'blocks_b.json'
```

```
Statistics for the generated blocks:
        Number of Blocks:   96
        Minimum Block Size: 3
        Maximum Block Size: 1108
        Average Block Size: 156.25
        Median Block Size:  86
        Standard Deviation of Block Size:  181.9370625706651
```

Various blocking statistics are printed out to help with refining the blocking schema. The record linkage run time will be largely dominated by the maximum block size, and the number of blocks. In general the smaller the average block size, the better.

Find matches between the two sets of CLKs

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

To compare the similarity between the bits we will use the anonlink entity service, which is provided by Data61.

The necessary steps are as follows:

- The analyst creates a new project. 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 encodings.
- 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:

[25]:
```
# NBVAL_IGNORE_OUTPUT
SERVER = 'https://anonlink.easd.data61.xyz'
!anonlink status --server={SERVER}
```

```
{"project_count": 1554, "rate": 521702, "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.

[26]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink create-project --server={SERVER} --schema "../_static/febrl_schema_v3_final.
↪json" --output "credentials.json" --type "groups" --name "Blocking Test" --blocked
```

```
Project created
```

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

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

[27]:
```
# NBVAL_IGNORE_OUTPUT
credentials = json.load(open("credentials.json", 'rt'))
print(json.dumps(credentials, indent=4))
```

```
{
    "project_id": "c23150a3d79b8abc7f6af32e50fa3f99a78cb87bc306063e",
    "result_token": "3e93c3bcb29c48c70fafbd4f8b6f50826baab3cb9b01a6e9",
    "update_tokens": [
        "bbadf1cf836438c013fda6c320fa1d394bd0d54ba8c7478b",
        "a36fa5b5f2ec5d92bd0676adf3022ab255914dc126df42e9"
    ]
}
```

Uploading the CLKs to the entity service

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

By default, data is uploaded to object store since it has more reliable uploading mechanism especially with large size datasets. You can also specify to upload to entity service directly by adding a flag `--to_entityservice` to command `upload`. A message will be printed to tell users if the encodings and blocks are uploaded to object store:

```
Anonlink client: Uploading to the external object store - MINIO
```

or they are uploaded to entity service:

```
Anonlink client: Uploading to entity service
```

Note that the progress bar is only available when uploading to object store. Uploading to entity service is performed with `requests` which does not support streaming upload yet.

[28]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink upload \
       --project="{credentials['project_id']}" \
        --apikey="{credentials['update_tokens'][0]}" \
        --output "upload_a.json" \
        --server="{SERVER}" \
        --blocks="blocks_a.json" \
        "clks_a.json"
```

```
uploads
Anonlink client: Uploading to the external object store - MINIO
Upload blocks_a.json: |####################| 0.45 MB/0.45 MB 100% [elapsed: 00:00␣
↪left: 00:00,  1.56 MB/sec]
```

[29]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink upload \
       --project="{credentials['project_id']}" \
        --apikey="{credentials['update_tokens'][1]}" \
        --output "upload_b.json" \
        --server="{SERVER}" \
        --blocks="blocks_b.json" \
        "clks_b.json"
```

```
uploads
Anonlink client: Uploading to the external object store - MINIO
Upload blocks_b.json: |####################| 0.45 MB/0.45 MB 100% [elapsed: 00:00␣
↪left: 00:00,  1.76 MB/sec]
```

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`:

[30]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink create --verbose  \
    --server="{SERVER}" \
    --output "run_info.json" \
    --threshold=0.9 \
    --project="{credentials['project_id']}" \
    --apikey="{credentials['result_token']}" \
    --name="Blocked run"
```

```
Connecting to Entity Matching Server: https://anonlink.easd.data61.xyz
```

[31]:
```
# NBVAL_IGNORE_OUTPUT
run_info = json.load(open("run_info.json", 'rt'))
run_info
```

[31]:
```
{'name': 'Blocked run',
 'notes': 'Run created by anonlink-client 0.1.4b0',
 'run_id': '3a294af61ba6123734eefae00781706158edcd58e08883fd',
 'threshold': 0.9}
```

Results

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

[32]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink results --watch \
        --project="{credentials['project_id']}" \
        --apikey="{credentials['result_token']}" \
        --run="{run_info['run_id']}" \
        --server="{SERVER}" \
        --output results.txt
```

```
State: completed
Stage (3/3): compute output
State: completed
Stage (3/3): compute output
State: completed
Stage (3/3): compute output
Downloading result
Received result
```

[33]:
```
# NBVAL_IGNORE_OUTPUT
def extract_matches(file):
    with open(file, 'rt') as f:
        results = json.load(f)['groups']
    # each entry in `results` looks like this: '((0, 4039), (1, 2689))'.
    # The format is ((dataset_id, row_id), (dataset_id, row_id))
    # As we only have two parties in this example, we can remove the dataset_ids.
    # Also, turning the solution into a set will make it easier to assess the
    # quality of the matching.
    found_matches = set((a, b) for ((_, a), (_, b)) in results)
    print('The service linked {} entities.'.format(len(found_matches)))
    return found_matches

found_matches = extract_matches('results.txt')
```

```
The service linked 3941 entities.
```

Let's investigate some of those matches and the overall matching quality. In this case we have the ground truth so we can compute the precision and recall.

Fortunately, the febrl4 datasets contain record ids which tell us the correct linkages. Using this information we are able to create a set of the true matches.

[34]:
```
# rec_id in dfA has the form 'rec-1070-org'. We only want the number. Additionally,␣
↪as we are
# interested in the position of the records, we create a new index which contains the␣
↪row numbers.
dfA_ = dfA.rename(lambda x: x[4:-4], axis='index').reset_index()
dfB_ = dfB.rename(lambda x: x[4:-6], axis='index').reset_index()
# now we can merge dfA_ and dfB_ on the record_id.
a = pd.DataFrame({'ida': dfA_.index, 'rec_id': dfA_['rec_id']})
b = pd.DataFrame({'idb': dfB_.index, 'rec_id': dfB_['rec_id']})
dfj = a.merge(b, on='rec_id', how='inner').drop(columns=['rec_id'])
# and build a set of the corresponding row numbers.
true_matches = set((row[0], row[1]) for row in dfj.itertuples(index=False))
```

[35]:
```
def describe_matching_quality(found_matches, show_examples=False):
    if show_examples:
        print('idx_a, idx_b,     rec_id_a,       rec_id_b')
        print('-----------------------------------------')
        for a_i, b_i in itertools.islice(found_matches, 10):
            print('{:3}, {:6}, {:>15}, {:>15}'.format(a_i+1, b_i+1, a.iloc[a_i]['rec_
↪id'], b.iloc[b_i]['rec_id']))
        print('-----------------------------------------')

    tp = len(found_matches & true_matches)
    fp = len(found_matches - true_matches)
    fn = len(true_matches - found_matches)

    precision = tp / (tp + fp)
    recall = tp / (tp + fn)


    print('Precision: {:.2f}, Recall: {:.2f}'.format(precision, recall))
```

[36]:
```
# NBVAL_IGNORE_OUTPUT
describe_matching_quality(found_matches, True)
```

```
idx_a, idx_b,     rec_id_a,       rec_id_b
-----------------------------------------
3170,    259,            3730,            3730
733,   2003,            4239,            4239
1685,   3323,            2888,            2888
4550,   3627,            4216,            4216
1875,   2991,            4391,            4391
3928,   2377,            3493,            3493
4928,   4656,             276,             276
2288,   4331,            3491,            3491
334,    945,            4848,            4848
4088,   2454,            1850,            1850
-----------------------------------------
Precision: 1.00, Recall: 0.79
```

Precision tells us about how many of the found matches are actual matches, and recall tells us how many of the actual matches we found.

We may have to change how we encode or block the data, but first let's create another run with a lower `threshold` value.

[37]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink create --verbose  \
    --server="{SERVER}" \
    --output "run_info.json" \
    --threshold=0.8 \
    --project="{credentials['project_id']}" \
    --apikey="{credentials['result_token']}" \
    --name="Another blocked run"

run_info = json.load(open('run_info.json', 'rt'))
```

```
Connecting to Entity Matching Server: https://anonlink.easd.data61.xyz
```

[38]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink results --watch \
        --project="{credentials['project_id']}" \
        --apikey="{credentials['result_token']}" \
        --run="{run_info['run_id']}" \
        --server="{SERVER}" \
        --output results.txt
```

```
State: running
Stage (3/3): compute output
State: running
Stage (3/3): compute output
State: completed
Stage (3/3): compute output
Downloading result
Received result
```

[39]:
```
with open("results.txt", 'rt') as f:
    results = json.load(f)['groups']

found_matches = extract_matches('results.txt')
describe_matching_quality(found_matches)
```

```
The service linked 4586 entities.
Precision: 1.00, Recall: 0.92
```

Great, for this lower threshold value we get a higher precision.

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.

It is important to choose an appropriate threshold for the amount of perturbations present in the data.

Feel free to go back to the encoding and blocking and experiment on how different setting will affect the matching quality.

P-Sig Blocking

Here we can try a different blocking technique - probability signature blocking.

[43]:
```
with open("psig_blocking_schema.json") as f:
    print(f.read())
```

```
{
        "type": "p-sig",
        "version": 1,
        "config": {
            "blocking-features": [1, 2, 3, 4, 5, 6, 7],
            "filter": {
                "type": "count",
                "max": 100,
                "min": 0.00
            },
            "blocking-filter": {
                "type": "bloom filter",
                "number-hash-functions": 4,
                "bf-len": 2048
            },
            "signatureSpecs": [
                [
                    {"type": "characters-at", "config": {"pos": ["0"]}, "feature-idx":
↪ 1},
                    {"type": "characters-at", "config": {"pos": ["0"]}, "feature-idx":
↪ 2}
                ],
                [
                    {"type": "characters-at", "config": {"pos": ["0:"]}, "feature-idx
↪": 7}
                ],
                [
                    {"type": "characters-at", "config": {"pos": ["0:"]}, "feature-idx
↪": 4}
                ],
                [
                    {"type": "characters-at", "config": {"pos": ["0:3"]}, "feature-idx
↪": 1},
                    {"type": "characters-at", "config": {"pos": ["0"]}, "feature-idx":
↪ 2}
                ],
                [
                    {"type": "characters-at", "config": {"pos": ["1:4"]}, "feature-idx
↪": 1},
                    {"type": "characters-at", "config": {"pos": ["0"]}, "feature-idx":
↪ 2}
                ],
                [
                    {"type": "characters-at", "config": {"pos": ["0:"]}, "feature-idx
↪": 2},
                    {"type": "characters-at", "config": {"pos": ["0:"]}, "feature-idx
↪": 3},
                    {"type": "characters-at", "config": {"pos": ["0:"]}, "feature-idx
↪": 4}
                ],
                [
                    {"type": "metaphone", "feature-idx": 1},
                    {"type": "metaphone", "feature-idx": 2},
                    {"type": "characters-at", "config": {"pos": ["0:"]}, "feature-idx
↪": 7}
```

(continues on next page)

(continued from previous page)

```
                ]
            ]
        }
}
```

[44]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink block 'PII_a.csv' 'psig_blocking_schema.json' 'psig_blocks_a.json'
```

```
P-Sig: 100.0% records are covered in blocks
Statistics for the generated blocks:
        Number of Blocks:   24640
        Minimum Block Size: 1
        Maximum Block Size: 15
        Average Block Size: 1.2175324675324675
        Median Block Size:  1
        Standard Deviation of Block Size:  0.9897405036787084
```

[45]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink block 'PII_b.csv' 'psig_blocking_schema.json' 'psig_blocks_b.json'
```

```
P-Sig: 100.0% records are covered in blocks
Statistics for the generated blocks:
        Number of Blocks:   24650
        Minimum Block Size: 1
        Maximum Block Size: 15
        Average Block Size: 1.2170385395537526
        Median Block Size:  1
        Standard Deviation of Block Size:  0.9891874327374874
```

[46]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink create-project --server={SERVER} --schema "../_static/febrl_schema_v3_final.
↪json" --output "credentials.json" --type "groups" --name "Blocking Test" --blocked
```

```
Project created
```

[47]:
```
# NBVAL_IGNORE_OUTPUT
credentials = json.load(open("credentials.json", 'rt'))
print(json.dumps(credentials, indent=4))
```

```
{
    "project_id": "2c50410532dca33b9415a98078302b8b04b37024b6186bac",
    "result_token": "c2a53a2442ea40dd0863bf1a45213803ca58e70422817ff7",
    "update_tokens": [
        "2c0b3cf1275b085fdf0991430934e58cd92e61b389ee2a2a",
        "c405aa7cb58cc2461e500e188da43bfbd2bd6a58e532a754"
    ]
}
```

[48]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink upload \
       --project="{credentials['project_id']}" \
        --apikey="{credentials['update_tokens'][0]}" \
        --output "upload_a.json" \
        --server="{SERVER}" \
        --blocks="psig_blocks_a.json" \
        "clks_a.json"
```

```
uploads
Anonlink client: Uploading to the external object store - MINIO
Upload psig_blocks_a.json: |####################| 1.24 MB/1.24 MB 100% [elapsed: 00:
↪00 left: 00:00,  1.93 MB/sec]
```

[49]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink upload \
       --project="{credentials['project_id']}" \
        --apikey="{credentials['update_tokens'][1]}" \
        --output "upload_b.json" \
        --server="{SERVER}" \
        --blocks="psig_blocks_b.json" \
        "clks_b.json"
```

```
uploads
Anonlink client: Uploading to the external object store - MINIO
Upload psig_blocks_b.json: |####################| 1.24 MB/1.24 MB 100% [elapsed: 00:
↪00 left: 00:00,  2.00 MB/sec]
```

[50]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink create --verbose  \
    --server="{SERVER}" \
    --output "run_info.json" \
    --threshold=0.8 \
    --project="{credentials['project_id']}" \
    --apikey="{credentials['result_token']}" \
    --name="Psig Blocked run"
```

```
Connecting to Entity Matching Server: https://anonlink.easd.data61.xyz
```

[51]:
```
# NBVAL_IGNORE_OUTPUT
run_info = json.load(open("run_info.json", 'rt'))
run_info
```

[51]:
```
{'name': 'Psig Blocked run',
 'notes': 'Run created by anonlink-client 0.1.4b0',
 'run_id': '7e090fc37a00dbb54d47babb65921af8d0053d189ada1d23',
 'threshold': 0.8}
```

[52]:
```
# NBVAL_IGNORE_OUTPUT
!anonlink results --watch \
        --project="{credentials['project_id']}" \
        --apikey="{credentials['result_token']}" \
        --run="{run_info['run_id']}" \
        --server="{SERVER}" \
        --output psig_results.txt
```

```
State: created
Stage (1/3): waiting for CLKs
Progress: 0.00%
State: created
Stage (1/3): waiting for CLKs
Progress: 0.00%
State: created
Stage (1/3): waiting for CLKs
Progress: 50.00%
State: running
Stage (2/3): compute similarity scores
```

(continues on next page)

(continued from previous page)

```
State: running
Stage (2/3): compute similarity scores
Progress: 100.00%
State: running
Stage (3/3): compute output
State: completed
Stage (3/3): compute output
Downloading result
Received result
```

[53]:
```
found_matches = extract_matches('psig_results.txt')
describe_matching_quality(found_matches)
```

```
The service linked 4752 entities.
Precision: 1.00, Recall: 0.95
```

Cleanup

Finally to remove the results from the service delete the individual runs, or remove the uploaded data and all runs by deleting the entire project.

[54]:
```
# NBVAL_IGNORE_OUTPUT
# Deleting a run
!anonlink delete --project="{credentials['project_id']}" \
        --apikey="{credentials['result_token']}" \
        --run="{run_info['run_id']}" \
        --server="{SERVER}"
```

```
Run deleted
```

[55]:
```
# NBVAL_IGNORE_OUTPUT
# Deleting a project
!anonlink delete-project --project="{credentials['project_id']}" \
        --apikey="{credentials['result_token']}" \
        --server="{SERVER}"
```

```
Project deleted
```

[ ]:
```

```

1.2 Command Line Tool

`anonlink-client` includes a command line tool which can be used to interact without writing Python code. The primary use case is to encode personally identifiable data from a csv into Cryptographic Longterm Keys.

The command line tool can be accessed in two equivalent ways:

- Using the `anonlink` script which gets added to your path during installation.
- directly running the python module with `python -m anonlinkclient`.

A list of valid commands can be listed with the `--help` argument:

```
$ anonlink --help
Usage: anonlink [OPTIONS] COMMAND [ARGS]...

  This command line application allows a user to hash their data into
  cryptographic longterm keys for use in private comparison.

  This tool can also interact with a entity matching service; creating new
  mappings, uploading locally hashed data, watching progress, and retrieving
  results.

  Example:

      anonlink hash private_data.csv secret schema.json output-clks.json

  All rights reserved Confidential Computing 2016.

Options:
  --version      Show the version and exit.
  -v, --verbose  Script is more talkative
  --help         Show this message and exit.

Commands:
  benchmark                carry out a local benchmark
  block                    generate candidate blocks from local PII data
  compare                  compare two schemas
  convert-schema           converts schema file to latest version
  create                   create a run on the entity service
  create-project           create a linkage project on the entity service
  delete                   delete a run on the anonlink entity service
  delete-project           delete a project on the anonlink entity service
  describe                 show distribution of clk popcounts
  generate                 generate random pii data for testing
  generate-default-schema  get the default schema used in generated random PII
  hash                     generate hashes from local PII data
  results                  fetch results from entity service
  status                   get status of entity service
  upload                   upload hashes to entity service
  validate-schema          validate linkage schema
```

1.2.1 Command specific help

The `anonlink` tool has help pages for all commands built in - simply append `--help` to the command.

1.2.2 Hashing

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

```
$ anonlink hash --help
Usage: anonlink hash [OPTIONS] PII_CSV SECRET SCHEMA CLK_JSON

  Process data to create CLKs

  Given a file containing CSV data as PII_CSV, and a JSON document defining
```

(continues on next page)

(continued from previous page)

```
  the expected schema, verify the schema, then hash the data to create CLKs
  writing them as JSON to CLK_JSON.

  It is important that the secret is only known by the two data providers.
  One word must be provided. For example:

  $anonlink hash pii.csv horse_stable pii-schema.json clks.json

  Use "-" for CLK_JSON to write JSON to stdout.

Options:
  --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
  -v, --verbose           Script is more talkative
  --help                  Show this message and exit.
```

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 `anonlink` with:

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

Where:

- `horse` is the secret that both participants will use to hash their data.
- `simple-schema.json` is a *Linkage 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 Blocking

The command line tool `anonlink` can be used to generate blocks given a csv file of personally identifiable information. The tool needs to be provided with keys and a *Blocking Schema*; it will output a file containing json serialized candidate blocks.

```
$ anonlink block --help
Usage: anonlink block [OPTIONS] PII_CSV SCHEMA BLOCK_JSON

  Process data to create blocking information

  Given a file containing CSV data as PII_CSV, and a JSON document defining
  the blocking configuration, then generate candidate blocks writing to JSON
  output. Note the CSV file should contain a header row - however this row
  is not used by this tool. Setting the verbose flag outputs more detailed
  blocking statistics.
```

(continues on next page)

(continued from previous page)

```
  For example:

  $anonlink block pii.csv blocking-schema.json blocks.json

  Use "-" for BLOCKS_JSON to write JSON to stdout.

Options:
  --no-header    Don't skip the first row
  -v, --verbose  Script is more talkative
  --help         Show this message and exit.
```

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 `anonlink` with:

```
$ anonlink block --schema blocking-schema.json fake-pii.csv horse candidate_blocks.
↪json
```

1.2.4 Describing

Users can inspect the distribution of the number of bits set in `CLKs` by using the `describe` command. Note that this *describe* only works on the CLKs produced by *anonlink hash*.

```
$ anonlink describe --help
Usage: anonlink describe [OPTIONS] CLK_JSON

  show distribution of clk's popcounts using a ascii plot.

Options:
  --help  Show this message and exit.
```

Example

```
$ anonlink describe example_clks_a.json


 339|                                    oo
 321|                                   ooo
 303|                                   ooo
 285|                                   ooo o
 268|                                   oooooo
 250|                                 oooooooo
 232|                                 oooooooo
 214|                               ooooooooo
```

(continues on next page)

(continued from previous page)

```
 196|                              o oooooooooo o
 179|                              o ooooooooooo
 161|                              ooooooooooooo
 143|                             ooooooooooooooo
 125|                            ooooooooooooooooo
 107|                            ooooooooooooooooo
  90|                          ooooooooooooooooooooo
  72|                          ooooooooooooooooooooo
  54|                         ooooooooooooooooooooooo
  36|                       ooooooooooooooooooooooooooo
  18|                     ooooooooooooooooooooooooooooooo
   1| o  o  oooooooooooooooooooooooooooooooooooooooooooooooooo oo
     ------------------------------------------------------------
     4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 7 7 7 7
     1 2 3 4 5 6 7 9 0 1 2 3 4 5 7 8 9 0 1 2 3 5 6 7 8 9 0 1 3 4
     0 1 2 4 5 7 8 0 1 2 4 5 7 8 0 1 2 4 5 7 8 0 1 2 4 5 7 8 0 1
       . . . . . . . . . . . . . . . . . . . . . . . . . . . .
       4 8 3 7 1 6 0 4 9 3 7 2 6 0 5 9 3 8 2 6 1 5 9 4 8 2 7 1 5

-------------------------
|        Summary        |
-------------------------
|   observations: 5000  |
| min value: 410.000000 |
|   mean : 601.571600   |
| max value: 753.000000 |
-------------------------
```

Note: It is an indication of problems in the hashing if the distribution is skewed towards no bits set or all bits set. Consult the *Tutorial for CLI tool anonlink-client* for further details.

1.2.5 Schema Handling

A schema file can be tested for validity against the schema specification with the `validate-schema` command. Note that currently `validate-schema` only works for linkage schema.

```
$ anonlink validate-schema --help
Usage: anonlink validate-schema [OPTIONS] SCHEMA

  Validate a linkage schema

  Given a file containing a linkage schema, verify the schema is valid
  otherwise print detailed errors.

Options:
  --help  Show this message and exit.
```

Example

```
$ anonlink validate-schema clkhash/data/randomnames-schema.json
schema is valid
```

Schema files of older versions can be converted to the latest version with the `convert-schema` command.

```
$ anonlink convert-schema --help
Usage: anonlink convert-schema [OPTIONS] SCHEMA_JSON OUTPUT

  convert the given schema file to the latest version.

Options:
  --help  Show this message and exit.
```

1.2.6 Data Generation

The command line tool has a `generate` command for generating fake pii data.

```
$ anonlink generate --help
Usage: anonlink generate [OPTIONS] [SIZE] OUTPUT

  Generate fake PII data for testing

Options:
  -s, --schema FILENAME
  --help                  Show this message and exit.
```

```
$ anonlink 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:

```
$ anonlink 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"
```

(continues on next page)

(continued from previous page)

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

1.2.7 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*):

```
anonlink 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.

1.2.8 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 Linkage Schema

As CLKs are usually used for privacy preserving linkage, it is important that participating organisations agree on how raw personally identifiable information is encoded to create the CLKs. The linkage schema allows putting more emphasis on particular features and provides a basic level of data validation.

We call the configuration of how to create CLKs a *linkage schema*. The organisations agree on a linkage schema 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 (and secret keys).

The linkage schema is a detailed description of exactly how to carry out the encoding operation, along with any configuration for the low level hashing itself.

The format of the linkage schema is defined in a separate JSON Schema specification document - schemas/v3.json.

Earlier versions of the linkage schema will continue to work, internally they are converted to the latest version (currently `v3`).

1.3.1 Basic Structure

A linkage schema consists of three parts:

- *version*, contains the version number of the hashing schema.
- *clkConfig*, CLK wide configuration, independent of features.
- *features*, an array of configuration specific to individual features.

1.3.2 Example Schema

```
{
  "version": 3,
  "clkConfig": {
    "l": 1024,
    "kdf": {
      "type": "HKDF",
      "hash": "SHA256",
      "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
↪G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",
      "info": "",
      "keySize": 64
    }
  },
  "features": [
    {
      "identifier": "INDEX",
      "ignored": true
    },
    {
      "identifier": "NAME freetext",
      "format": {
        "type": "string",
        "encoding": "utf-8",
        "case": "mixed",
        "minLength": 3
      },
      "hashing": {
        "comparison": {
          "type": "ngram",
          "n": 2
        },
        "strategy": {
            "bitsPerFeature": 100
        },
        "hash": {"type": "doubleHash"}
      }
    },
    {
      "identifier": "DOB YYYY/MM/DD",
      "format": {
        "type": "date",
        "description": "Numbers separated by slashes, in the year, month, day order",
        "format": "%Y/%m/%d"
      },
      "hashing": {
        "comparison": {
          "type": "ngram",
          "n": 1,
          "positional": true
        },
        "strategy": {
          "bitsPerFeature": 200
        },
        "hash": {"type": "doubleHash"}
      }
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "identifier": "GENDER M or F",
      "format": {
        "type": "enum",
        "values": ["M", "F"]
      },
      "hashing": {
        "comparison": {
          "type": "ngram",
          "n": 1
        },
        "strategy": {
          "bitsPerFeature": 400
        },
        "hash": {"type": "doubleHash"}
      }
    }
  ]
}
```

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
kdf	*KDF*	no	defines the key derivation function used to generate individual secrets for each feature derived from the master secret
xorFolds	integer	yes	number of XOR folds (as proposed in [Schnell2016]).

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

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

Each feature is configured by:

- identifier, the human readable name. E.g. `"First Name"`.
- description, a human readable description of this 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
comparison	one of: *n-gram comparison*, *exact comparison*, *numeric comparison*	no	specifies the comparison technique for this feature.
strategy	one of: *BitsPerTokenStrategy*, *BitsPerFeatureStrategy*	no	the strategy for assigning bits to the encoding.
hash	one of: *DoubleHash BlakeHash*	yes	specifies the hash function for inserting bits into the Bloom filter, defaults to bake hash
missingValue	*missingValue*	yes	allows to define how missing values are handled

Strategies

A strategy defines how often a token is inserted into the Bloom filter.

BitsPerTokenStrategy

Insert every token `bitsPerToken` number of times.

name	type	optional	description
bitsPerToken	integer	no	max number of indices per token

BitsPerFeatureStrategy

Same number of insertions for each value of this feature, irrespective of the actual number of tokens. The number of filter insertions for a token is computed by dividing `bitsPerFeature` equally amongst the tokens.

name	type	optional	description
bitsPerFeature	integer	no	max number of indices per feature

Hash

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

Choose one of:

DoubleHash

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/data61/clkhash/issues/33

BlakeHash

the (default) option

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

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 the `sentinel` with the `replaceWith` value.

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.

n-gram comparison

Approximate string matching with n-gram tokenization. Also see the API docs for NgramComparison

name	type	optional	description
type	string	no	has to be 'ngram'
n	integer	no	The 'n' in n-gram
positional	boolean	yes	positional n-grams also contains the position of the n-gram within the string

exact comparison

Exact string matching. Also see the API docs for ExactComparison

name	type	optional	description
type	string	no	has to be 'exact'

numeric comparison

Numerical comparisons of integers or floating point numbers such that the distance between two numbers relate to the similarity of the produced tokens. Also see the API docs for NumericComparison

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 Blocking Schema

Each blocking method has its own configuration and parameters to tune with. To make our API as generic as possible, we designed the blocking schema to specify the configuration of the blocking method including features to use in generating blocks and hyperparameters etc.

Currently we support two blocking methods:

- "*p-sig*": Probability signature
- "*lambda-fold*": LSH based λ-fold

which are proposed by the following publications:

- Scalable Entity Resolution Using Probabilistic Signatures on Parallel Databases
- An LSH-Based Blocking Approach with a Homomorphic Matching Technique for Privacy-Preserving Record Linkage

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

1.4.1 Basic Structure

A blocking schema consists of three parts:

- *type*, the blocking method to be used
- *version*, the version number of the hashing schema.
- *config*, an json configuration of that blocking method that varies with different blocking methods

1.4.2 Example Schema

```
{
  "type": "lambda-fold",
  "version": 1,
  "config": {
    "blocking-features": [1, 2],
    "Lambda": 30,
    "bf-len": 2048,
    "num-hash-funcs": 5,
    "K": 20,
    "input-clks": true,
    "random_state": 0
  }
}
```

1.4.3 Schema Components

type

String value which describes the blocking method.

name	detailed description
"*p-sig*"	Probability Signature blocking method from Scalable Entity Resolution Using Probabilistic Signatures on Parallel Databases
"*lambda-fold*"	LSH based Lambda Fold Redundant blocking method from Scalable Entity Resolution Using Probabilistic Signatures on Parallel Databases

version

Integer value that indicates the version of blocking schema. Currently the only supported version is *1*.

config

Configuration specific to each blocking method. Next we will detail the specific configuration for supported blocking methods.

Specific configuration of supported blocking methods can be found here:

- *config of p-sig <blocking-schema/p-sig>*
- *config of lambda-fold <blocking-schema/lambda-fold>*

Probabilistic Signature Configuration

attribute	type	description
blocking-features	list[integer]	specify which features u
filter	dictionary	filtering threshold
blocking-filter	dictionary	type of filter to generate blocks
signatureSpecs	list of lists	signature strategies where each list is a combination of signature strategies

Filter Configuration

attribute	type	description
type	string	either "ratio" or "count" that represents proportional or absolute filtering
max	numeric	for ratio, it should be within 0 and 1; for count, it should not exceed the number of records

Blocking-filter Configuration

attribute	type	description
type	string	currently we only support "bloom filter"
number-hash-functions	integer	this specifies how many bits will be flipped for each signature
bf-len	integer	defines the length of blocking filter, for bloom filter usually this is 1024 or 2048

SignatureSpecs Configurations

It is better to illustrate this one with an example:

```
{
  "signatureSpecs": [
    [
     {"type": "characters-at", "config": {"pos": [0]}, "feature-idx": 1},
     {"type": "characters-at", "config": {"pos": [0]}, "feature-idx": 2},
    ],
    [
     {"type": "metaphone", "feature-idx": 1},
     {"type": "metaphone", "feature-idx": 2},
    ]
  ]
}
```

here we generate two signatures for each record where each signature is a combination of signatures: - first signature is the first character of feature at index 1, concatenating with first character of feature at index 2 - second signature is the metaphone transformation of feature at index 1, concatenating with metaphone transformation of feature at index 2

The following specifies the current supported signature strategies:

strategies	description
feature-value	exact feature at specified index
characters-at	substring of feature
metaphone	phonetic encoding of feature

Finally a full example of p-sig blocking schema:

```
{
 "type": "p-sig",
 "version": 1,
 "config": {
     "blocking_features": [1],
     "filter": {
         "type": "ratio",
         "max": 0.02,
         "min": 0.00,
     },
     "blocking-filter": {
         "type": "bloom filter",
         "number-hash-functions": 4,
         "bf-len": 2048,
     },
     "signatureSpecs": [
         [
              {"type": "characters-at", "config": {"pos": [0]}, "feature-idx": 1},
              {"type": "characters-at", "config": {"pos": [0]}, "feature-idx": 2},
         ],
         [
             {"type": "metaphone", "feature-idx": 1},
             {"type": "metaphone", "feature-idx": 2},
         ]
     ]
```

(continues on next page)

(continued from previous page)

```
    }
  }
```

LSH based λ-fold Configuration

attribute	type	description
blocking-features	list[integer]	specify which features to used in blocks generation
Lambda	integer	denotes the degree of redundancy - H^i, $i = 1, 2, ..., \Lambda$ where each H^i represents one independent blocking group
bf-len	integer	length of bloom filter
num-hash-funcs	integer	number of hash functions used to map record to Bloom filter
K	integer	number of bits we will select from Bloom filter for each reocrd
random_state	integer	control random seed
input-clks	boolean	if true, the input data is CLK i.e. the output of *anonlink hash*; else the input data is raw data

Here is a full example of lambda-fold blocking schema:

```
{
  "type": "lambda-fold",
  "version": 1,
  "config": {
     "blocking-features": [1, 2],
     "Lambda": 5,
     "bf-len": 2048,
     "num-hash-funcs": 10,
     "K": 40,
     "random_state": 0,
     "input-clks": False
  }
}
```

1.5 Development

1.5.1 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 *pytest*:

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

Note several tests will be skipped by default. 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=anonlinkclient
```

1.5.2 Type Checking

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

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

1.5.3 Packaging

The `anonlink` command line tool can be frozen into an exe using PyInstaller:

```
pyinstaller cli.spec
```

Look for *anonlink.exe* in the *dist* directory.

1.6 Devops

1.6.1 Azure Pipeline

`anonlink-client` is automatically built and tested using Azure Pipeline for Windows environment, in the project Anonlink

Two pipelines are available: - Build pipeline, - Release pipeline.

Build Pipeline

The build pipeline is described by the script *azurePipeline.yml*.

There are 3 top level stages in the build pipeline:

- *Static Checks* - runs *mypy* typechecking over the codebase. Also adds a Azure DevOps tag *"Automated"* if the build was triggered by a Git tag.
- *Unit tests* - A template expands out into a number of builds and tests for different version of python and system architecture.
- *Packaging* - Pulls together the created files into a single release artifact.

The *Build & Test* job does:

- install the requirements,
- package `anonlink-client`,
- run tests as described in the following table,
- publish the test results,
- publish the code coverage (on Azure and codecov),
- publish the artifacts from the build using `Python 3.7` (i.e. the wheel, the sdist *tar.gz* and an exe for x86 and x64).

The build pipeline requires one environment variable provided by Azure environment:

- *CODECOV_TOKEN* which is used to publish the coverage to codecov.

Description of what is tested:

Python Version	Operating System	Standard pytest	INLCUDE_CLI	TEST_ENTITY_SERVICE	Notebooks
3.6	ubuntu-18.04	Yes	No	No	No
3.6	macos-10.13	Yes	No	No	No
3.6	vs2017-win2016 (x64)	Yes	No	No	No
3.6	vs2017-win2016 (x86)	Yes	No	No	No
3.7	ubuntu-18.04	Yes	Yes	Yes	Yes
3.7	macos-10.13	Yes	Yes	Yes	Yes
3.7	vs2017-win2016 (x64)	Yes	Yes	Yes	No
3.7	vs2017-win2016 (x86)	Yes	No	No	No
3.8	ubuntu-18.04	Yes	Yes	Yes	Yes
3.8	macos-10.13	Yes	No	No	No

The tests using the environment variable *TEST_ENTITY_SERVICE* will use the URL provided by the Azure pipeline variable *ENTITY_SERVICE_URL* (which is by default set to *https://anonlink.easd.data61.xyz*), which enables to run manually the pipeline with a different deployed service. However, we note that the pipeline will send github updates to the corresponding commit for the chosen deployment, not the default one if the variable has been overwritten.

Build Artifacts

A pipeline artifact named **Release** is created by the build pipeline which contains the universal wheel, source distribution and Windows executables for x86 and x64 architectures. Other artifacts are created from each build, including code coverage.

Release Pipeline

The release pipeline can either be triggered manually, or automatically from a successful build on master where the build is tagged *Automated* (i.e. if the commit is tagged, cf previous paragraph).

The release pipeline consists of two steps: - asking for a manual confirmation that the artifacts from the triggering build should be released, - uses `twine` to publish the artifacts.

The release pipeline requires two environment variables provided by Azure environment: - *PYPI_LOGIN*: login to push an artifact to `anonlink-client Pypi` repository, - *PYPI_PASSWORD*: password to push an artifact to `anonlink-client Pypi` repository for the user *PYPI_LOGIN*.

1.7 Rest Client API Documentation

`anonlink-client` includes a module for interacting with the Anonlink Entity Service.

class anonlinkclient.rest_client.**ClientWaitingConfiguration**(*wait_exponential_multiplier_ms=10000*, *wait_exponential_max_ms=10000*, *stop_max_delay_ms=20000*)

Bases: object

DEFAULT_STOP_MAX_DELAY_MS = 20000

DEFAULT_WAIT_EXPONENTIAL_MAX_MS = 10000

DEFAULT_WAIT_EXPONENTIAL_MULTIPLIER_MS = 100

exception anonlinkclient.rest_client.**RateLimitedClient**(*msg*, *response*)

Bases: *anonlinkclient.rest_client.ServiceError*

Exception indicating client is asking for updates too frequently.

class anonlinkclient.rest_client.**RestClient**(*server*, *client_waiting_configuration=None*)

Bases: object

get_temporary_objectstore_credentials(*project*, *apikey*)

Retrieve temporary object store credentials to upload

Parameters

- **project** – A project ID
- **apikey** – A dataprovider's upload token

Returns The credentials and upload information from the Anonlink Entity Service API.

project_create(*schema*, *result_type*, *name*, *notes=None*, *parties=2*, *uses_blocking=False*)

project_delete(*project*, *apikey*)

project_get_description(*project*, *apikey*)

project_upload_clks(*project*, *apikey*, *clk_data*)

run_create(*project_id*, *apikey*, *threshold*, *name*, *notes=None*)

run_delete(*project*, *run*, *apikey*)

run_get_result_text(*project*, *run*, *apikey*)

run_get_status(*project*, *run*, *apikey*)

server_get_status()

wait_for_run(*project*, *run*, *apikey*, *timeout=None*, *update_period=1*)

Monitor a linkage run and return the final status updates. If a timeout is provided and the run hasn't entered a terminal state (error or completed) when the timeout is reached a TimeoutError will be raised.

Parameters

- **project** –
- **run** –
- **apikey** –
- **timeout** – Stop waiting after this many seconds. The default (None) is to never give you up.
- **update_period** – Time in seconds between queries to the run's status.

Raises **TimeoutError** – if timeout is reached

watch_run_status(*project*, *run*, *apikey*, *timeout=None*, *update_period=1*)
Monitor a linkage run and yield status updates. Will immediately yield an update and then only yield further updates when the status object changes. If a timeout is provided and the run hasn't entered a terminal state (error or completed) when the timeout is reached, updates will cease and a TimeoutError will be raised.

Parameters

- **project** –
- **run** –
- **apikey** –
- **timeout** – Stop waiting after this many seconds. The default (None) is to never give you up.
- **update_period** – Time in seconds between queries to the run's status.

Raises **TimeoutError** – if timeout is reached

exception anonlinkclient.rest_client.**ServiceError**(*msg*, *response*)
Bases: Exception

Problem with the upstream API

anonlinkclient.rest_client.**format_run_status**(*status*)

1.8 References

CHAPTER 2

External Links

- anonlink-client on Github
- anonlink-client on Pypi

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.

Python Module Index

a

Index