CLK hash Documentation

Release 0.17.0

N1 Analytics

Feb 15, 2023

CONTENTS

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

Clkhash is Apache 2.0 licensed, supports Python versions 3.5+, and runs on Windows, OSX and Linux. Clkhash is part of the Anonlink project for Private Record Linkage from Data61.

Install `clkhash` with pip:

```
pip install clkhash
```

For a command line interface to clkhash see anonlink-client.

Hint: If you are interested in comparing CLK encodings (i.e carrying out record linkage) you might want to check out these related projects:

- anonlink
- anonlink-client
- anonlink-entity-service

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Tutorials

The *clkhash* library can be used via the Python API. For a command line interface to `clkhash` see anonlink-client.

The tutorial *tutorial_api.ipynb* shows an example linkage workflow.

With linkage schema version 3.0 *clkhash* introduced different comparison techniques for feature values. They are described in the tutorial *tutorial_comparisons.ipynb*.

1.1.1 running the tutorials

The notebooks can run online using binder.

You can download the tutorials from github. The dependencies are listed in *doc-requirements.txt*. Install and start Jupyter from the `docs` directory:

```
pip install -r doc-requirements.txt
python -m jupyter lab
```

Finally you can view a static version of the tutorials here.

Tutorial for Python API

For this tutorial we are going to process a data set for private linkage with `clkhash` using 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 the dependencies we will need:

[ ]:
```
# NBVAL_IGNORE_OUTPUT
!pip install -U clkhash anonlink recordlinkage pandas
```

[1]:
```
# NBVAL_IGNORE_OUTPUT
import io
import itertools
import pandas as pd
```

[2]:
```
import clkhash
from clkhash import clk
from clkhash.field_formats import *
from clkhash.schema import Schema
from clkhash.comparators import NgramComparison
from clkhash.serialization import serialize_bitarray
```

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

Data Exploration

First load the dataset, and preview the first few rows.

[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')
```

In this tutorial we will use `StringIO` buffers instead of files. Let's dump the data from the pandas dataframe into a csv:

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

Linkage Schema Definition

A hashing schema instructs `clkhash` how to treat each feature when encoding a CLK.

The linkage schema below details a 1024 bit encoding using equally weighted features. Most features are encoding using bigrams although the postcode and date of birth use unigrams. The schema specifies to ignore the columns `'rec_id'` and `'soc_sec_id'`.

A detailed description of the linkage schema can be found in the documentation.

[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='
↪'))),
    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)
```

Encode the data

We can now encode 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]:
```
a_csv.seek(0)
hashed_data_a = clk.generate_clk_from_csv(a_csv, secret, schema)
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:03<00:00, 1.39kclk/s, mean=944, std=14.4]
```

Inspect the output

clkhash has encoded 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, more than 900 out of 1024 bits are set on average which can affect accuracy.

We can control the popcount by adjusting the strategy. There are currently two different strategies implemented in the library:

- `BitsPerToken`: each token of a feature's value is inserted into the encoding `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 a feature's `FieldHashingProperties`.
- `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 a feature's `FieldHashingProperties`.

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

[10]:
```
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)
```

(continues on next page)

(continued from previous page)

```
a_csv.seek(0)
clks_a = clk.generate_clk_from_csv(a_csv, secret, schema)
```

```
generating CLKs: 100%|| 5.00k/5.00k [00:02<00:00, 2.20kclk/s, mean=696, std=22.7]
```

Each CLK is represented by a bitarray but can be serialized in a compact, JSON friendly base64 format:

[11]:
```
print("original:")
print(clks_a[0])
print("serialized:")
print(serialize_bitarray(clks_a[0]))
```

```
original:
bitarray(
↪'1111111100101100001100011011110111100111001111111000111110010100011101111111111110111000110111111110
↪')
serialized:
/ywxvec/j5R3/7jf71/l97u812e421MzNfNSrvyj+3uOfPbPFWt/t/WZX3+4/f1eXeb6TGLb29r/PSr/
↪d+bvwvx4Vfu97Yif/u+z79s+P76WkR6kKnb/n/9VnarWbcf78L8fPiX/vnxmjL7o/3S48vv9rNstV/t/
↪Xm9X93o3O70=
```

Hash data set B

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

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

```
generating CLKs: 100%|| 5.00k/5.00k [00:01<00:00, 2.58kclk/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.

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

[14]:
```
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,
```

(continues on next page)

(continued from previous page)

```
            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)
```

[15]:
```
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.

[16]:
```
# 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))
```

[17]:
```
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)
```

(continues on next page)

(continued from previous page)

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

[18]:
```
describe_matching_quality(found_matches, show_examples=True)
```

```
idx_a, idx_b,     rec_id_a,       rec_id_b
---------------------------------------------
3170,   259,        3730,           3730
1685,  3323,        2888,           2888
 733,  2003,        4239,           4239
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: 1.000, 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`.

[19]:
```
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]:
```
# NBVAL_IGNORE_OUTPUT
import random
import io
import csv
import numpy as np
import matplotlib.pyplot as plt

from clkhash.field_formats import *
from clkhash.schema import Schema
from clkhash.comparators import NgramComparison, ExactComparison, NumericComparison
from clkhash.clk import generate_clk_from_csv
```

Explanantion of the different comparison techniques

The clkhash library is based on the concept of a CLK. This is a special type of Bloom filter, and a Bloom filter is a probabilistic data structure that allow space-efficient testing of set membership. By first tokenising a record and then inserting those tokens into a CLK, the comparison of CLKs approximates the comparisons of the sets of tokens of the CLKs.

The challenge lies in finding good tokenisation strategies, as they define what is considered similiar and what is not. We call these tokenisation strategies *comparison techniques*.

With Schema v3, we currently support three different comparison techniques:

- ngram comparison
- exact comparison
- numeric comparison

In this notebook we describe how these techniques can be used and what type of data they are best suited.

n-gram Comparison

n-grams are a popular technique for approximate string matching.

An *n-gram* is a n-tuple of characters which follow one another in a given string. For example, the 2-grams of the string 'clkhash' are ' c', 'cl', 'lk', 'kh', 'ha', 'as', 'sh', 'h '. Note the white- space in the first and last token. They serve the purpose to a) indicate the beginning and end of a word, and b) gives every character in the input text a representation in two tokens.

The number of *n-grams* in common defines a similiarity measure for comparing strings. The strings 'clkhash' and 'clkhush' have 6 out of 8 2-grams in common, whereas 'clkhash' and 'anonlink' have none out of 9 in common.

A positional n-gram also encodes the position of the n-gram within the word. The positional 2-grams of 'clkhash' are '1 c', '2 cl', '3 lk', '4 kh', '5 ha', '6 as', '7 sh', '8 h '. Positional n-grams can be useful for comparing words where the position of the characters are important, e.g., postcodes or phone numbers.

n-gram comparison of strings is tolerant to spelling mistakes, as one wrong character will only affect *n n-grams*. Thus, the larger you choose 'n', the more the error propagates.

Exact Comparison

The exact comparison technique creates high similarity scores if inputs are identical, and low otherwise. This can be useful when comparing data like credit card numbers or email addresses. It is a good choice whenever data is either an exact match or has no similarity at all. The main advantage of the *Exact Comparison* technique is that it better separates the similarity scores of the matches from the non-matches (but cannot acount for errors).

We will show this with the following experiment. First, we create a dataset consisting of random 6-digit numbers. Then we compare the dataset with itself, once encoded with the *Exact Comparison*, and twice encoded with the *Ngram Comparison* (uni- and bi-grams) technique.

[2]:
```
data = [[i, x] for i, x in enumerate(random.sample(range(1000000), k=1000))]
a_csv = io.StringIO()
csv.writer(a_csv).writerows(data)
```

We define three different schemas, one for each comparison technique.

[3]:
```
unigram_fields = [
    Ignore('rec_id'),
    IntegerSpec('random', FieldHashingProperties(comparator=NgramComparison(1, True),␣
↪strategy=BitsPerFeatureStrategy(300))),
]
unigram_schema = Schema(unigram_fields, 512)


bigram_fields = [
    Ignore('rec_id'),
    IntegerSpec('random', FieldHashingProperties(comparator=NgramComparison(2, True),␣
↪strategy=BitsPerFeatureStrategy(300))),
]
bigram_schema = Schema(bigram_fields, 512)


exact_fields = [
    Ignore('rec_id'),
    IntegerSpec('random', FieldHashingProperties(comparator=ExactComparison(),␣
↪strategy=BitsPerFeatureStrategy(300))),
]

exact_schema = Schema(exact_fields, 512)


secret_key = 'password1234'
```

[4]:
```
from bitarray import bitarray
import base64
import anonlink

def grouped_sim_scores_from_clks(clks_a, clks_b, threshold):
    """returns the pairwise similarity scores for the provided clks, grouped into␣
↪matches and non-matches"""
    results_candidate_pairs = anonlink.candidate_generation.find_candidate_pairs(
            [clks_a, clks_b],
            anonlink.similarities.dice_coefficient,
            threshold
    )
    matches = []
    non_matches = []
    sims, ds_is, (rec_id0, rec_id1) = results_candidate_pairs
    for sim, rec_i0, rec_i1 in zip(sims, rec_id0, rec_id1):
        if rec_i0 == rec_i1:
            matches.append(sim)
        else:
            non_matches.append(sim)
    return matches, non_matches
```

generate the CLKs according to the three different schemas.

[5]:
```
a_csv.seek(0)
clks_a_unigram = generate_clk_from_csv(a_csv, secret_key, unigram_schema, header=False)
```

(continues on next page)

(continued from previous page)

```
a_csv.seek(0)
clks_a_bigram = generate_clk_from_csv(a_csv, secret_key, bigram_schema, header=False)
a_csv.seek(0)
clks_a_exact = generate_clk_from_csv(a_csv, secret_key, exact_schema, header=False)
```

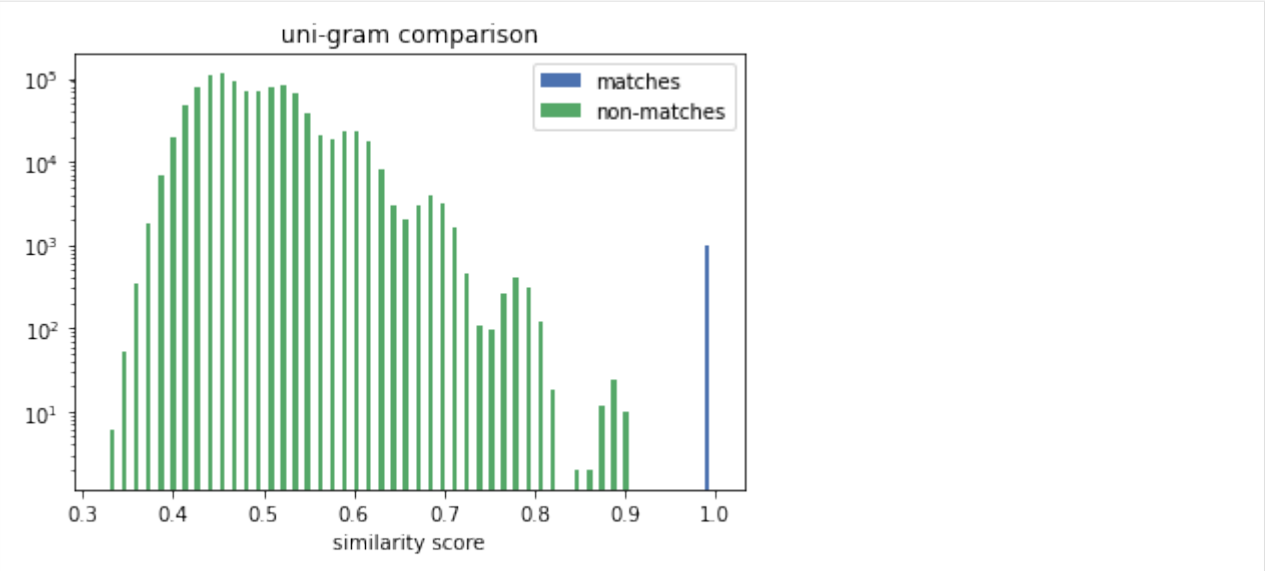
```
generating CLKs: 100%|| 1.00k/1.00k [00:00<00:00, 9.90kclk/s, mean=229, std=5.8]
generating CLKs: 100%|| 1.00k/1.00k [00:00<00:00, 25.5kclk/s, mean=228, std=5.89]
generating CLKs: 100%|| 1.00k/1.00k [00:00<00:00, 24.8kclk/s, mean=227, std=5.73]
```

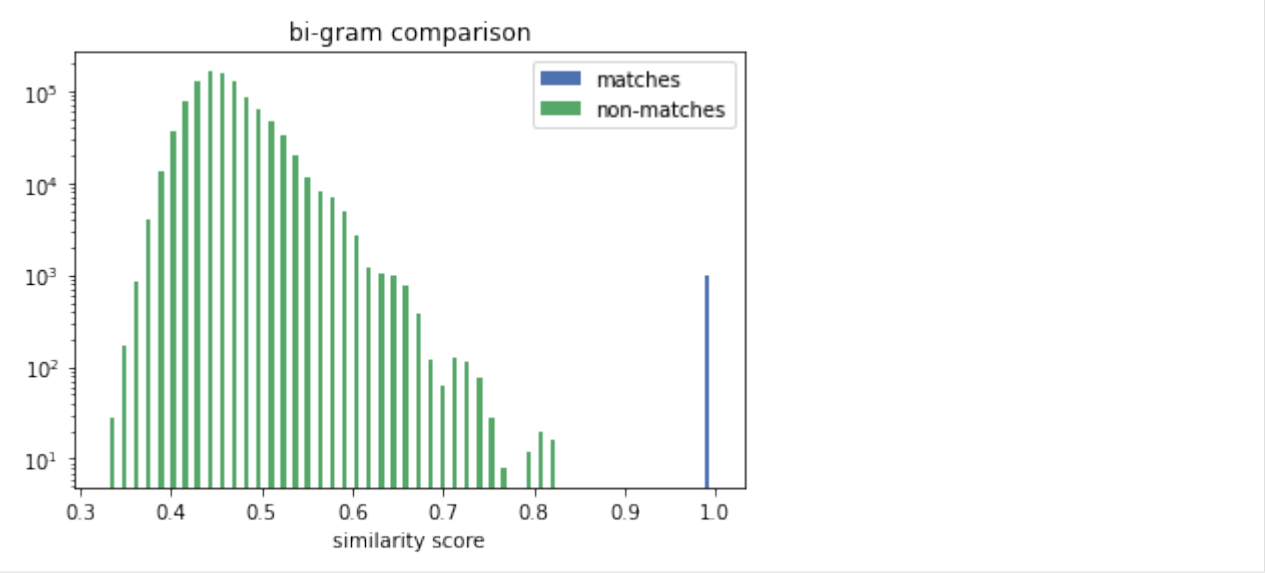
We do an exhaustive pairwise comparison for the CLKs and group the similarity scores into 'matches' - the similarity scores for the correct linkage - and non-matches.

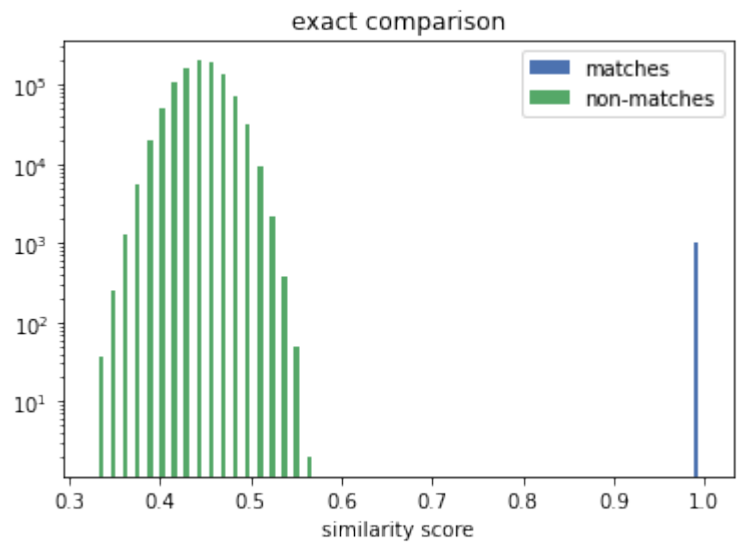
[6]:
```
sims_matches_unigram, sims_non_matches_unigram = grouped_sim_scores_from_clks(clks_a_
↪unigram, clks_a_unigram, 0.0)
sims_matches_bigram, sims_non_matches_bigram = grouped_sim_scores_from_clks(clks_a_
↪bigram, clks_a_bigram, 0.0)
sims_matches_exact, sims_non_matches_exact = grouped_sim_scores_from_clks(clks_a_exact,␣
↪clks_a_exact, 0.0)
```

We will plot the similarity scores as histograms. Note the log scale of the y-axis.

[7]:
```
# NBVAL_IGNORE_OUTPUT
import matplotlib.pyplot as plt
plt.style.use('seaborn-deep')

plt.hist([sims_matches_unigram, sims_non_matches_unigram], bins=50, label=['matches',
↪'non-matches'])
plt.legend(loc='upper right')
plt.yscale('log')
plt.xlabel('similarity score')
plt.title('uni-gram comparison')
plt.show()
plt.hist([sims_matches_bigram, sims_non_matches_bigram], bins=50, label=['matches', 'non-
↪matches'])
plt.legend(loc='upper right')
plt.yscale('log')
plt.xlabel('similarity score')
plt.title('bi-gram comparison')
plt.show()
plt.hist([sims_matches_exact, sims_non_matches_exact], bins=50, label=['matches', 'non-
↪matches'])
plt.legend(loc='upper right')
plt.yscale('log')
plt.xlabel('similarity score')
plt.title('exact comparison')
plt.show()
```


uni-gram comparison
matches
non-matches
10^5
10^4
10^3
10^2
10^1
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
similarity score


bi-gram comparison
matches
non-matches
10^5
10^4
10^3
10^2
10^1
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
similarity score



The true matches all lie on the vertical line above the 1.0. We can see that the *Exact Comparison* technique significantly widens the gap between matches and non-matches. Thus increases the range of available solving thresholds (only similarity scores above are considered a potential match) which provide the correct linkage result.

Numeric Comparison

This technique enables numerical comparisons of integers and floating point numbers.

Comparing numbers creates an interesting challenge. The comparison of 1000 with 1001 should lead to the same result as the comparison of 1000 and 999. They are both exactly 1 apart. However, string-based techniques like n-gram comparison will produce very different results, as the first pair has three digits in common, compared to none in the last pair.

We have implemented a technique, where the numerical distance between two numbers relates to the similarity of the produced tokens.

We generate a dataset with one column of random 6-digit integers, and a second dataset where we alter the integers of the first dataset by +/- 100.

[8]:
```
data_A = [[i, random.randrange(1000000)] for i in range(1000)]
data_B = [[i, x + random.randint(-100,100)] for i,x in data_A]
```

[9]:
```
a_csv = io.StringIO()
b_csv = io.StringIO()
csv.writer(a_csv).writerows(data_A)
csv.writer(b_csv).writerows(data_B)
```

We define two linkage schemas, one for postitional uni-gram comparison and one for numeric comparison.

The parameter *resolution* controls how many different token are generated. Clkhash will produce *2 * resolution + 1* tokens (*resolution* tokens on either side of the input value plus the input value iteself).

And *threshold_distance* controls the sensitivity of the comparison. Only numbers that are not more than *threshold_distance* apart will produce overlapping tokens.

[10]:
```
unigram_fields = [
    Ignore('rec_id'),
    IntegerSpec('random',
                FieldHashingProperties(comparator=NgramComparison(1, True),
                                       strategy=BitsPerFeatureStrategy(301))),
]
unigram_schema = Schema(unigram_fields, 512)

bigram_fields = [
    Ignore('rec_id'),
    IntegerSpec('random',
                FieldHashingProperties(comparator=NgramComparison(2, True),
                                       strategy=BitsPerFeatureStrategy(301))),
]
bigram_schema = Schema(unigram_fields, 512)

numeric_fields = [
    Ignore('rec_id'),
    IntegerSpec('random',
                FieldHashingProperties(comparator=NumericComparison(threshold_
↪distance=500, resolution=150),
                                       strategy=BitsPerFeatureStrategy(301))),
]
numeric_schema = Schema(numeric_fields, 512)

secret_key = 'password1234'
```

[11]:
```
a_csv.seek(0)
clks_a_unigram = generate_clk_from_csv(a_csv, secret_key, unigram_schema, header=False)
b_csv.seek(0)
clks_b_unigram = generate_clk_from_csv(b_csv, secret_key, unigram_schema, header=False)
a_csv.seek(0)
clks_a_bigram = generate_clk_from_csv(a_csv, secret_key, bigram_schema, header=False)
b_csv.seek(0)
clks_b_bigram = generate_clk_from_csv(b_csv, secret_key, bigram_schema, header=False)
a_csv.seek(0)
clks_a_numeric = generate_clk_from_csv(a_csv, secret_key, numeric_schema, header=False)
b_csv.seek(0)
clks_b_numeric = generate_clk_from_csv(b_csv, secret_key, numeric_schema, header=False)
```

```
generating CLKs: 100%|| 1.00k/1.00k [00:00<00:00, 17.4kclk/s, mean=229, std=5.98]
generating CLKs: 100%|| 1.00k/1.00k [00:00<00:00, 18.7kclk/s, mean=229, std=5.85]
generating CLKs: 100%|| 1.00k/1.00k [00:00<00:00, 16.5kclk/s, mean=229, std=5.98]
generating CLKs: 100%|| 1.00k/1.00k [00:00<00:00, 18.1kclk/s, mean=229, std=5.85]
generating CLKs: 100%|| 1.00k/1.00k [00:00<00:00, 3.26kclk/s, mean=228, std=5.86]
generating CLKs: 100%|| 1.00k/1.00k [00:00<00:00, 3.23kclk/s, mean=228, std=5.75]
```

First, we will look at the similarity score distributions. We will group the similiarity scores into *matches* - the similarity scores for the correct linkage - and *non-matches*.

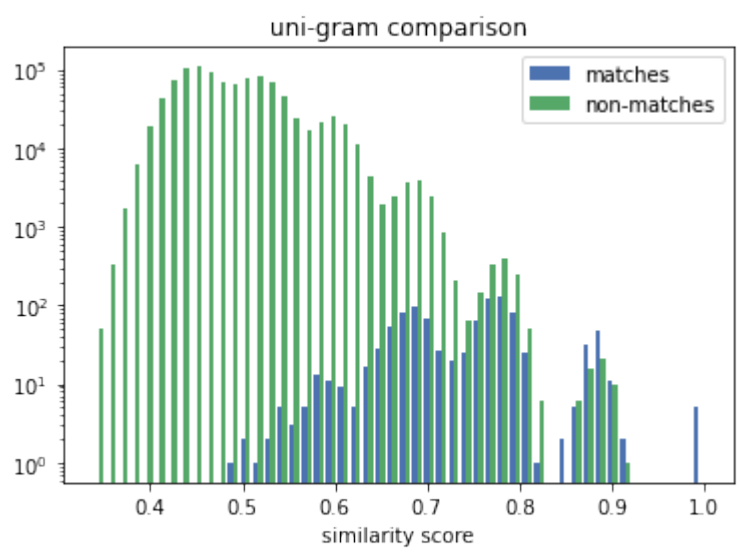
[12]:
```
sims_matches_unigram, sims_non_matches_unigram = grouped_sim_scores_from_clks(clks_a_
↪unigram, clks_b_unigram, 0.0)
sims_matches_bigram, sims_non_matches_bigram = grouped_sim_scores_from_clks(clks_a_
↪bigram, clks_b_bigram, 0.0)
```

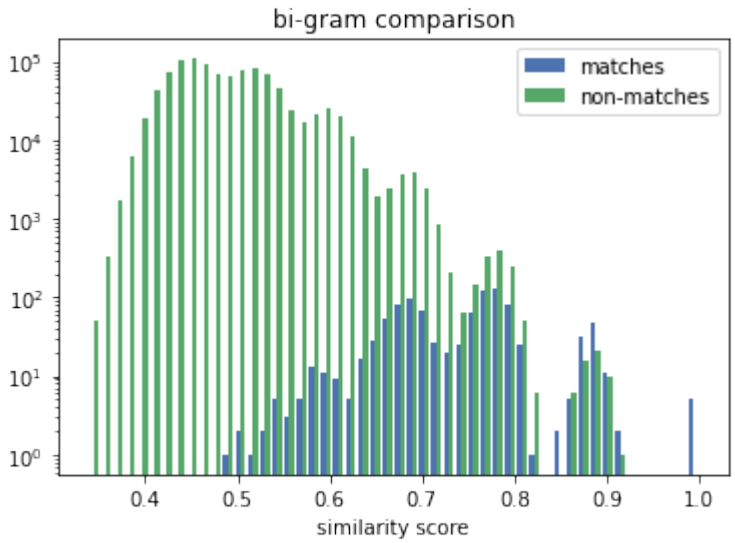
(continues on next page)

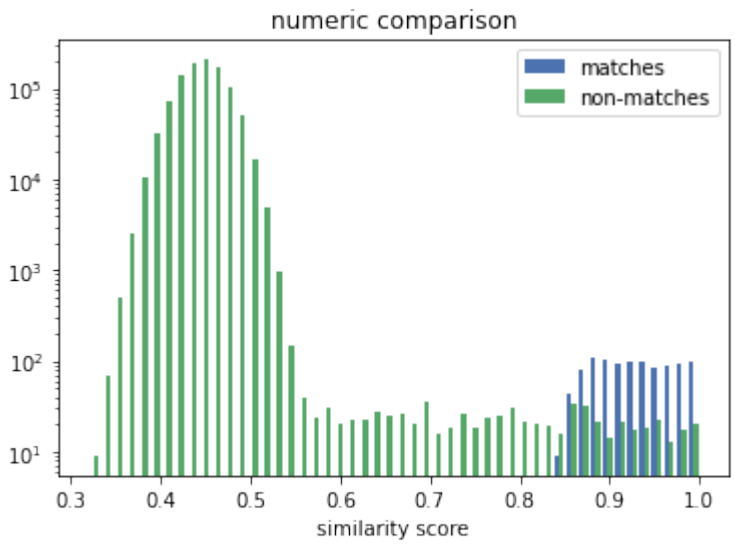
(continued from previous page)

```
sims_matches_numeric, sims_non_matches_numeric = grouped_sim_scores_from_clks(clks_a_
↪numeric, clks_b_numeric, 0.0)
```

[13]:
```
# NBVAL_IGNORE_OUTPUT
plt.style.use('seaborn-deep')

plt.hist([sims_matches_unigram, sims_non_matches_unigram], bins=50, label=['matches',
↪'non-matches'])
plt.legend(loc='upper right')
plt.yscale('log')
plt.xlabel('similarity score')
plt.title('uni-gram comparison')
plt.show()

plt.hist([sims_matches_bigram, sims_non_matches_bigram], bins=50, label=['matches', 'non-
↪matches'])
plt.legend(loc='upper right')
plt.yscale('log')
plt.xlabel('similarity score')
plt.title('bi-gram comparison')
plt.show()

plt.hist([sims_matches_numeric, sims_non_matches_numeric], bins=50, label=['matches',
↪'non-matches'])
plt.legend(loc='upper right')
plt.yscale('log')
plt.xlabel('similarity score')
plt.title('numeric comparison')
plt.show()
```







The distribution for the numeric comparison is very different to the uni/bi-gram one. The similarity scores of the matches (the correct linkage) in the n-gram case are mixed-in with the scores of the non-matches, making it challenging for a solver to decide if a similarity score denotes a match or a non-match.

The numeric comparison produces similarity scores for matches that mirrors the distribution of the numeric distances. More importanty, there is a good separation between the scores for the matches and the ones for the non-matches. The former are all above 0.8, whereas the latter are almost all (note the log scale) below 0.6.

In the next step, we will see how well the solver can find a linkage solution for the different CLKs.

[14]:
```
def mapping_from_clks(clks_a, clks_b, threshold):
    """computes a mapping between clks_a and clks_b using the anonlink library"""
    results_candidate_pairs = anonlink.candidate_generation.find_candidate_pairs(
            [clks_a, clks_b],
            anonlink.similarities.dice_coefficient,
            threshold
    )
```

(continues on next page)

(continued from previous page)

```
    solution = anonlink.solving.greedy_solve(results_candidate_pairs)
    return set( (a,b) for ((_, a),(_, b)) in solution)

true_matches = set((i,i) for i in range(1000))

def describe_matching_quality(found_matches):
    """computes and prints precision and recall of the found_matches"""
    tp = len(true_matches & found_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))
```

[15]:
```
print('results for numeric comparisons')
print('threshold 0.6:')
describe_matching_quality(mapping_from_clks(clks_a_numeric, clks_b_numeric, 0.6))
print('threshold 0.7:')
describe_matching_quality(mapping_from_clks(clks_a_numeric, clks_b_numeric, 0.7))
print('threshold 0.8:')
describe_matching_quality(mapping_from_clks(clks_a_numeric, clks_b_numeric, 0.8))
```

```
results for numeric comparisons
threshold 0.6:
Precision: 0.907, Recall: 0.905
threshold 0.7:
Precision: 0.907, Recall: 0.905
threshold 0.8:
Precision: 0.915, Recall: 0.905
```

[16]:
```
print('results for unigram comparisons')
print('threshold 0.6:')
describe_matching_quality(mapping_from_clks(clks_a_unigram, clks_b_unigram, 0.6))
print('threshold 0.7:')
describe_matching_quality(mapping_from_clks(clks_a_unigram, clks_b_unigram, 0.7))
print('threshold 0.8:')
describe_matching_quality(mapping_from_clks(clks_a_unigram, clks_b_unigram, 0.8))
```

```
results for unigram comparisons
threshold 0.6:
Precision: 0.343, Recall: 0.336
threshold 0.7:
Precision: 0.396, Recall: 0.327
threshold 0.8:
Precision: 0.558, Recall: 0.130
```

As expected, we can see that the solver does a lot better when given the CLKs generated with the numeric comparison technique.

The other thing that stands out is that the results in with the numeric comparison are stable over a wider range of thresholds, in contrast to the unigram comparison, where different thresholds produce different results, thus making it more challenging to find a good threshold.

Conclusions

The overall quality of the linkage result is heavily influence by the right choice of comparison technique for each individual feature. In summary: - *n-gram comparison* is best suited for fuzzy string matching. It can account for localised errors like spelling mistakes. - *exact comparison* produces high similiarity only for exact matches, low otherwise. This can be useful if the data is noise-free and partial similarities are not relevant. For instance credit card numbers, even if they only differ in one digit they discribe different accounts and are thus just as different then numbers which don't have any digits in common. - *numeric comparison* provides a measure of similiarity that relates to the numerical distance of two numbers. Example use-cases are measurements like height or weight, continuous variables like salary.

[ ]:

1.2 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`).

1.2.1 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.

1.2.2 Example Schema

```
{
  "version": 3,
  "clkConfig": {
    "l": 1024,
    "kdf": {
      "type": "HKDF",
      "hash": "SHA256",
      "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
↪G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",
```

(continues on next page)

(continued from previous page)

```
      "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"}
      }
    },
    {
      "identifier": "GENDER M or F",
      "format": {
        "type": "enum",
        "values": ["M", "F"]
```

(continues on next page)

(continued from previous page)

```
      },
      "hashing": {
        "comparison": {
          "type": "ngram",
          "n": 1
        },
        "strategy": {
          "bitsPerFeature": 400
        },
        "hash": {"type": "doubleHash"}
      }
    }
  ]
}
```

A more advanced example can be found here.

1.2.3 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
ignored	boolean	yes	if set, clkhash will ignore this feature
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

name	type	optional	description
type	string	no	has to be 'numeric'
thresholdDistance	number	no	positive number, if distance is not more than this, two values will produce overlapping tokens
resolution	integer	no	produce 2 * resolution + 1 tokens
fractional_precision	integer	yes	quantisation of floats

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.

1.3 Development

1.3.1 API Documentation

Bloom filter

Generate a Bloom filter

`clkhash.bloomfilter.`**`blake_encode_ngrams`**(*ngrams: Iterable[str], keys: Sequence[bytes], ks: Sequence[int], l: int, encoding: str*) → bitarray.bitarray

Computes the encoding of the ngrams using the BLAKE2 hash function.

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

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

Our construction

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

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

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

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

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

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

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

Parameters

- **ngrams** – list of n-grams to be encoded
- **keys** – secret key for blake2 as bytes
- **ks** – ks[i] is k value to use for ngram[i]
- **l** – length of the output bitarray (has to be a power of 2)
- **encoding** – the encoding to use when turning the ngrams to bytes

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

`clkhash.bloomfilter.`**`crypto_bloom_filter`**(*record: Sequence[str], comparators: List[*clkhash.comparators.AbstractComparison*], schema:* clkhash.schema.Schema, *keys: Sequence[Sequence[bytes]]*) → Tuple[bitarray.bitarray, str, int]

Computes the composite Bloom filter encoding of a record.

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

Parameters

- **record** – plaintext record tuple. E.g. (index, name, dob, gender)
- **comparators** – A list of comparators. They provide a 'tokenize' function to turn string into appropriate tokens.
- **schema** – Schema
- **keys** – Keys for the hash functions as a tuple of lists of bytes.

Returns

3-tuple:

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

`clkhash.bloomfilter.double_hash_encode_ngrams(ngrams: Iterable[str], keys: Sequence[bytes], ks: Sequence[int], l: int, encoding: str) → bitarray.bitarray`

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

> Using the method from: Schnell, R., Bachteler, T., & Reiher, J. (2011). A Novel Error-Tolerant Anonymous Linking Code. http://grlc.german-microsimulation.de/wp-content/uploads/2017/05/downloadwp-grlc-2011-02.pdf

Parameters

- **ngrams** – list of n-grams to be encoded
- **keys** – hmac secret keys for md5 and sha1 as bytes
- **ks** – ks[i] is k value to use for ngram[i]
- **l** – length of the output bitarray
- **encoding** – the encoding to use when turning the ngrams to bytes

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

`clkhash.bloomfilter.double_hash_encode_ngrams_non_singular(ngrams: Iterable[str], keys: Sequence[bytes], ks: Sequence[int], l: int, encoding: str) → bitarray.bitarray`

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

The original construction of [Schnell2011] displays an abnormality for certain inputs:

> An n-gram can be encoded into just one bit irrespective of the number of k.

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

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

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

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

and thus

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

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

Parameters

- **ngrams** – list of n-grams to be encoded
- **keys** – tuple with (key_sha1, key_md5). That is, (hmac secret keys for sha1 as bytes, hmac secret keys for md5 as bytes)
- **ks** – ks[i] is k value to use for ngram[i]
- **l** – length of the output bitarray
- **encoding** – the encoding to use when turning the ngrams to bytes

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

`clkhash.bloomfilter.fold_xor(bloomfilter: bitarray.bitarray, folds: int) → bitarray.bitarray`

Performs XOR folding on a Bloom filter.

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

Parameters

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

Returns folded bloom filter

`clkhash.bloomfilter.hashing_function_from_properties(fhp: clkhash.field_formats.FieldHashingProperties) → Callable[[Iterable[str], Sequence[bytes], Sequence[int], int, str], bitarray.bitarray]`

Get the hashing function for this field :param fhp: hashing properties for this field :return: the hashing function

`clkhash.bloomfilter.stream_bloom_filters(dataset: Iterable[Sequence[str]], keys: Sequence[Sequence[bytes]], schema: clkhash.schema.Schema) → Iterable[Tuple[bitarray.bitarray, str, int]]`

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

Parameters

- **dataset** – An iterable of indexable records.
- **schema** – An instantiated Schema instance
- **keys** – A tuple of two lists of secret keys used in the HMAC.

Returns Generator yielding bloom filters as 3-tuples

CLK

Generate CLK from data.

`clkhash.clk.chunks(seq: Sequence[clkhash.clk.T], chunk_size: int) → Iterable[Sequence[clkhash.clk.T]]`

Split seq into chunk_size-sized chunks.

Parameters

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

`clkhash.clk.generate_clk_from_csv(input_f: TextIO, secret: AnyStr, schema: clkhash.schema.Schema, validate: bool = True, header: Union[bool, AnyStr] = True, progress_bar: bool = True, max_workers: Optional[int] = None) → List[bitarray.bitarray]`

Generate Bloom filters from CSV file, then serialise them.

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

Parameters

- **input_f** – A file-like object of csv data to hash.

- **secret** – A secret.
- **schema** – Schema specifying the record formats and hashing settings.
- **validate** – Set to *False* to disable validation of data against the schema. Note that this will silence warnings whose aim is to keep the hashes consistent between data sources; this may affect linkage accuracy.
- **header** – Set to *False* if the CSV file does not have a header. Set to *'ignore'* if the CSV file does have a header but it should not be checked against the schema.
- **progress_bar** (*bool*) – Set to *False* to disable the progress bar.
- **max_workers** (*int*) – Passed to ProcessPoolExecutor except for the special case where the value is 1, in which case no processes or threads are used. This may be useful or required on platforms that are not capable of spawning subprocesses.

Returns A list of Bloom filters as bitarrays and a list of corresponding popcounts.

`clkhash.clk.generate_clks(pii_data: Sequence[Sequence[str]], schema: clkhash.schema.Schema, secret: AnyStr, validate: bool = True, callback: Optional[Callable[[int, Sequence[int]], None]] = None, max_workers: Optional[int] = None) → List[bitarray.bitarray]`

`clkhash.clk.hash_chunk(chunk_pii_data: Sequence[Sequence[str]], keys: Sequence[Sequence[bytes]], schema: clkhash.schema.Schema) → Tuple[List[bitarray.bitarray], Sequence[int]]`

Generate Bloom filters (ie hash) from chunks of PII. It also computes and outputs the Hamming weight (or popcount) – the number of bits set to one – of the generated Bloom filters.

Parameters

- **chunk_pii_data** – An iterable of indexable records.
- **keys** – A tuple of two lists of keys used in the HMAC. Should have been created by *generate_key_lists*.
- **schema** (*Schema*) – Schema specifying the entry formats and hashing settings.

Returns A list of Bloom filters as bitarrays and a list of corresponding popcounts

key derivation

`clkhash.key_derivation.generate_key_lists(secret: Union[bytes, str], num_identifier: int, num_hashing_methods: int = 2, key_size: int = 64, salt: Optional[bytes] = None, info: Optional[bytes] = None, kdf: str = 'HKDF', hash_algo: str = 'SHA256') → Tuple[Tuple[bytes, ...], ...]`

Generates *num_hashing_methods* derived keys for each identifier for the secret using a key derivation function (KDF).

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

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

Parameters

- **secret** – a secret (either as bytes or string)
- **num_identifier** – the number of identifiers

- **num_hashing_methods** – number of hashing methods used per identifier, each of them requiring a different key
- **key_size** – the size of the derived keys
- **salt** – salt for the KDF as bytes
- **info** – optional context and application specific information as bytes
- **kdf** – the key derivation function algorithm to use
- **hash_algo** – the hashing algorithm to use (ignored if *kdf* is not 'HKDF')

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

`clkhash.key_derivation.hkdf(secret: bytes, num_keys: int, hash_algo: str = 'SHA256', salt: Optional[bytes] = None, info: Optional[bytes] = None, key_size: int = 64) → Tuple[bytes, ...]`

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

Parameters

- **secret** – input keying material
- **num_keys** – the number of keys the kdf should produce
- **hash_algo** – The hash function used by HKDF for the internal HMAC calls. The choice of hash function defines the maximum length of the output key material. Output bytes <= 255 * hash digest size (in bytes).
- **salt** – HKDF is defined to operate with and without random salt. This is done to accommodate applications where a salt value is not available. We stress, however, that the use of salt adds significantly to the strength of HKDF, ensuring independence between different uses of the hash function, supporting "source-independent" extraction, and strengthening the analytical results that back the HKDF design. Random salt differs fundamentally from the initial keying material in two ways: it is non-secret and can be re-used. Ideally, the salt value is a random (or pseudorandom) string of the length HashLen. Yet, even a salt value of less quality (shorter in size or with limited entropy) may still make a significant contribution to the security of the output keying material.
- **info** – While the 'info' value is optional in the definition of HKDF, it is often of great importance in applications. Its main objective is to bind the derived key material to application- and context-specific information. For example, 'info' may contain a protocol number, algorithm identifiers, user identities, etc. In particular, it may prevent the derivation of the same keying material for different contexts (when the same input key material (IKM) is used in such different contexts). It may also accommodate additional inputs to the key expansion part, if so desired (e.g., an application may want to bind the key material to its length L, thus making L part of the 'info' field). There is one technical requirement from 'info': it should be independent of the input key material value IKM.
- **key_size** – the size of the produced keys

Returns Derived keys

random names

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

Names and ages are based on Australian and USA census data, but are not correlated. Additional functions for manipulating the list of names - producing reordered and subset lists with a specific overlap

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

TODO: Generate realistic errors TODO: Add RESTful api to generate reasonable name data as requested

class clkhash.randomnames.**Distribution**(*resource_name: str*)

Bases: object

Creates a random value generator with a weighted distribution

generate() → str

Generates a random value, weighted by the known distribution

load_csv_data(*resource_name: str*) → None

Loads the first two columns of the specified CSV file from package data. The first column represents the value and the second column represents the count in the population.

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

Bases: object

Randomly generated PII records.

SCHEMA = <Schema (v3): 4 fields>

generate_random_person(*n: int*) → Iterable[Tuple[str, str, str, str]]

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

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

generate_subsets(*sz: int*, *overlap: float = 0.8*, *subsets: int = 2*) → Tuple[List, ...]

Return random subsets with nonempty intersection.

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

Parameters

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

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

Returns tuple of subsets

load_data() → None

Loads databases from package data

Uses data files sourced from http://www.quietaffiliate.com/free-first-name-and-last-name-databases-csv-and-sql/ https://www.census.gov/topics/population/genealogy/data/2010_surnames.html https://www.abs.gov.au/AUSSTATS/abs@.nsf/DetailsPage/3101.0Jun%202016

```
randomname_schema = {'clkConfig': {'kdf': {'hash': 'SHA256', 'info':
'c2NoZW1hX2V4YW1wbGU=', 'keySize': 64, 'salt': 'SCbL2zHNnmsckfzchsNkZY9XoHk96P/
G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==', 'type': 'HKDF'}, 'l':
1024}, 'features': [{'identifier': 'INDEX', 'ignored': True}, {'identifier': 'NAME
freetext', 'format': {'type': 'string', 'encoding': 'utf-8', 'case': 'mixed',
'minLength': 3}, 'hashing': {'comparison': {'type': 'ngram', 'n': 2, 'positional':
False}, 'strategy': {'bitsPerToken': 15}, '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':
{'bitsPerToken': 30}, 'hash': {'type': 'doubleHash'}}}, {'identifier': 'GENDER M or
F', 'format': {'type': 'enum', 'values': ['M', 'F']}, 'hashing': {'comparison':
{'type': 'ngram', 'n': 1, 'positional': False}, 'strategy': {'bitsPerToken': 60},
'hash': {'type': 'doubleHash'}}}], 'version': 3}
```

```
randomname_schema_bytes = b'{\n "version": 3,\n "clkConfig": {\n "l": 1024,\n "kdf":
{\n "type": "HKDF",\n "hash": "SHA256",\n "salt": "SCbL2zHNnmsckfzchsNkZY9XoHk96P/
G5nUBrM7ybymlEFsMV6PAeDZCNp3rfNUPCtLDMOGQHG4pCQpfhiHCyA==",\n "info":
"c2NoZW1hX2V4YW1wbGU=",\n "keySize": 64\n }\n },\n "features": [\n {\n "identifier":
"INDEX",\n "ignored": true\n },\n {\n "identifier": "NAME freetext",\n "format": {\n
"type": "string",\n "encoding": "utf-8",\n "case": "mixed",\n "minLength": 3\n },\n
"hashing": {\n "comparison": {\n "type": "ngram",\n "n": 2\n },\n "strategy": {\n
"bitsPerToken": 15\n },\n "hash": {"type": "doubleHash"}\n }\n },\n {\n
"identifier": "DOB YYYY/MM/DD",\n "format": {\n "type": "date",\n "description":
"Numbers separated by slashes, in the year, month, day order",\n "format":
"%Y/%m/%d"\n },\n "hashing": {\n "comparison": {\n "type": "ngram",\n "n": 1,\n
"positional": true\n },\n "strategy": {\n "bitsPerToken": 30\n },\n "hash": {"type":
"doubleHash"}\n }\n },\n {\n "identifier": "GENDER M or F",\n "format": {\n "type":
"enum",\n "values": ["M", "F"]\n },\n "hashing": {\n "comparison": {\n "type":
"ngram",\n "n": 1\n },\n "strategy": {\n "bitsPerToken": 60\n },\n "hash": {"type":
"doubleHash"}\n }\n }\n ]\n}\n'
```

property schema_types: Sequence[*clkhash.field_formats.FieldSpec*]

clkhash.randomnames.**random_date**(*year: int*, *age_distribution: Optional[*clkhash.randomnames.Distribution*]*) → datetime.datetime

Generate a random datetime between two datetime objects.

Parameters

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

Returns random datetime between start and end

clkhash.randomnames.**save_csv**(*data: Iterable[Tuple[Union[str, int], ...]]*, *headers: Iterable[str]*, *file: TextIO*) → None

Output generated data to file as CSV with header.

Parameters

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

schema

Schema loading and validation.

exception clkhash.schema.**MasterSchemaError**

Bases: Exception

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

class clkhash.schema.**Schema**(*fields: Sequence[clkhash.field_formats.FieldSpec], l: int, xor_folds: int = 0, kdf_type: str = 'HKDF', kdf_hash: str = 'SHA256', kdf_info: Optional[bytes] = None, kdf_salt: Optional[bytes] = None, kdf_key_size: int = 64*)

Bases: object

Linkage Schema which describes how to encode plaintext identifiers.

Variables

- **fields** – the features or field definitions
- **l** (*int*) – The length of the resulting encoding in bits. This is the length after XOR folding.
- **xor_folds** (*int*) – The number of XOR folds to perform on the hash.
- **kdf_type** (*str*) – The key derivation function to use. Currently, the only permitted value is 'HKDF'.
- **kdf_hash** (*str*) – The hash function to use in key derivation. The options are 'SHA256' and 'SHA512'.
- **kdf_info** (*bytes*) – The info for key derivation. See documentation of `key_derivation.hkdf()` for details.
- **kdf_salt** (*bytes*) – The salt for key derivation. See documentation of `key_derivation.hkdf()` for details.
- **kdf_key_size** (*int*) – The size of the derived keys in bytes.

exception clkhash.schema.**SchemaError**(*msg: str, errors: Optional[Sequence[clkhash.field_formats.InvalidSchemaError]] = None*)

Bases: Exception

The user-defined schema is invalid.

clkhash.schema.**convert_to_latest_version**(*schema_dict: Dict[str, Any], validate_result: Optional[bool] = False*) → Dict[str, Any]

Convert the given schema to latest schema version.

Parameters

- **schema_dict** – A dictionary describing a linkage schema. This dictionary must have a *'version'* key containing a master schema version. The rest of the schema dict must conform to the corresponding master schema.
- **validate_result** – validate converted schema against schema specification

Returns schema dict of the latest version

Raises *SchemaError* – if schema version is not supported

`clkhash.schema.from_json_dict(dct: Dict[str, Any], validate: bool = True) → clkhash.schema.Schema`

Create a Schema of the most recent version according to dct

if the provided schema dict is of an older version, then it will be automatically converted to the latest.

Parameters

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

Raises *SchemaError* – An exception containing details about why the schema is not valid.

Returns the Schema

`clkhash.schema.from_json_file(schema_file: TextIO, validate: bool = True) → clkhash.schema.Schema`

Load a Schema object from a json file.

Parameters

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

Raises *SchemaError* – When the schema is invalid.

Returns the Schema

`clkhash.schema.validate_schema_dict(schema: Dict[str, Any]) → None`

Validate the schema.

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

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

Raises

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

field_formats

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

`class clkhash.field_formats.BitsPerFeatureStrategy(bits_per_feature: int)`

Bases: *clkhash.field_formats.StrategySpec*

Have a fixed number of filter insertions for a feature, irrespective of the actual number of tokens.

This strategy allows to reason about the importance of a feature, irrespective of the lengths of the feature values. For example, in the BitsPerTokenStrategy the name 'Bob' affects only have the number of bits in the Bloom filter than 'Robert'. With this BitsPerFeatureStrategy, both names set the same number of bits in the filter, thus allowing to adjust importance on a per feature basis.

Variables **bits_per_feature** (*int*) – total number of insertions for this feature, will be spread across all tokens.

bits_per_token(*num_tokens: int*) → List[int]

Return a list of integers, one for each of the *num_tokens* tokens, defining how often that token gets inserted into the Bloom filter.

Parameters **num_tokens** (*int*) – number of tokens in the feature's value

Returns [k, …] with k's >= 0

class clkhash.field_formats.**BitsPerTokenStrategy**(*bits_per_token: int*)

Bases: *clkhash.field_formats.StrategySpec*

Insert every token the same number of times.

This is the strategy from the original Schnell paper. The provided value *bits_per_token* (the 'k' value in the paper) defines the number of hash functions that are used to insert each token into the Bloom filter.

One important property of this strategy is that the total number of inserted bits for a feature relates to the length of its value. This can have privacy implications, as the number of bits set in a Bloom filter correlate to the number of tokens of the PII.

Variables **bits_per_token** (*int*) – how often each token should be inserted into the filter

bits_per_token(*num_tokens: int*) → List[int]

Return a list of integers, one for each of the *num_tokens* tokens, defining how often that token gets inserted into the Bloom filter.

Parameters **num_tokens** (*int*) – number of tokens in the feature's value

Returns [k, …] with k's >= 0

class clkhash.field_formats.**DateSpec**(*identifier: str, hashing_properties:* clkhash.field_formats.FieldHashingProperties, *format: str, description: Optional[str] = None*)

Bases: *clkhash.field_formats.FieldSpec*

Represents a field that holds dates.

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

Variables **format** (*str*) – The format of the date.

OUTPUT_FORMAT = '%Y%m%d'

classmethod from_json_dict(*json_dict: Dict[str, Any]*) → *clkhash.field_formats.DateSpec*

Make a DateSpec object from a dictionary containing its properties.

Parameters

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

validate(*str_in: str*) → None

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

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

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

Raises

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

class clkhash.field_formats.**EnumSpec**(*identifier: str, hashing_properties: clkhash.field_formats.FieldHashingProperties, values: Iterable[str], description: Optional[str] = None*)

Bases: *clkhash.field_formats.FieldSpec*

Represents a field that holds an enum.

The finite collection of permitted values must be specified.

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

classmethod from_json_dict(*json_dict: Dict[str, Any]*) → *clkhash.field_formats.EnumSpec*

Make a EnumSpec object from a dictionary containing its properties.

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

validate(*str_in: str*) → None

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

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

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

Raises *InvalidEntryError* – When entry is invalid.

class clkhash.field_formats.**FieldHashingProperties**(*comparator: clkhash.comparators.AbstractComparison, strategy: clkhash.field_formats.StrategySpec, encoding: str = 'utf-8', hash_type: str = 'blakeHash', prevent_singularity: Optional[bool] = None, missing_value: Optional[clkhash.field_formats.MissingValueSpec] = None*)

Bases: object

Stores the settings used to hash a field.

This includes the encoding and tokenisation parameters.

Variables

- **comparator** (AbstractComparison) – provides a tokenizer for desired comparison strategy
- **encoding** (*str*) – The encoding to use when converting the string to bytes. Refer to Python's documentation for possible values.
- **hash_type** (*str*) – hash function to use for hashing
- **prevent_singularity** (*bool*) – the 'doubleHash' function has a singularity problem
- **num_bits** (*int*) – dynamic k = num_bits / number of n-grams
- **k** (*int*) – max number of bits per n-gram

- **missing_value** (MissingValueSpec) – specifies how to handle missing values

replace_missing_value(*str_in: str*) → str

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

Parameters **str_in** (*str*) – input string

Returns str_in or the missingValue replacement value

class clkhash.field_formats.**FieldSpec**(*identifier: str, hashing_properties: Optional[*clkhash.field_formats.FieldHashingProperties*], description: Optional[str] = None*)

Bases: object

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

Variables

- **identifier** (*str*) – The name of the field.
- **description** (*str*) – Description of the field format.
- **hashing_properties** (FieldHashingProperties) – The properties for hashing. None if field ignored.

format_value(*str_in: str*) → str

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

There are several reasons why this might be necessary:

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

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

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

classmethod from_json_dict(*field_dict: Dict[str, Any]*) → *clkhash.field_formats.FieldSpec*

Initialise a *FieldSpec* object from a dictionary of properties.

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

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

is_missing_value(*str_in: str*) → bool

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

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

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

abstract validate(*str_in: str*) → None

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

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

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

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

class clkhash.field_formats.**Ignore**(*identifier: Optional[str] = None*)

Bases: *clkhash.field_formats.FieldSpec*

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

validate(*str_in: str*)

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

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

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

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

class clkhash.field_formats.**IntegerSpec**(*identifier: str, hashing_properties:* clkhash.field_formats.FieldHashingProperties, *description: Optional[str] = None, minimum: Optional[int] = None, maximum: Optional[int] = None, **kwargs: Dict[str, Any]*)

Bases: *clkhash.field_formats.FieldSpec*

Represents a field that holds integers.

Minimum and maximum values may be specified.

Variables

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

classmethod from_json_dict(*json_dict: Dict[str, Any]*) → *clkhash.field_formats.IntegerSpec*

Make a IntegerSpec object from a dictionary containing its properties.

Parameters

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

validate(*str_in: str*) → None

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

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

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

Raises *InvalidEntryError* – When entry is invalid.

exception clkhash.field_formats.**InvalidEntryError**

Bases: ValueError

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

field_spec = None

exception clkhash.field_formats.**InvalidSchemaError**

Bases: ValueError

Raised if the schema of a field specification is invalid.

For example, a regular expression included in the schema is not syntactically correct.

field_spec_index = None

json_field_spec = None

class clkhash.field_formats.**MissingValueSpec**(*sentinel: str, replace_with: Optional[str] = None*)

Bases: object

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

Variables

- **sentinel** (*str*) – sentinel is the string that identifies a missing value e.g.: 'N/A', ''. The sentinel will not be validated against the feature format definition
- **replace_with** (*str*) – defines the string which replaces the sentinel whenever present, can be 'None', then sentinel will not be replaced.

classmethod from_json_dict(*json_dict: Dict[str, Any]*) → *clkhash.field_formats.MissingValueSpec*

class clkhash.field_formats.**StrategySpec**

Bases: object

Stores the information about the insertion strategy.

A strategy has to implement the 'bits_per_token' function, which defines how often each token gets inserted into the Bloom filter.

abstract bits_per_token(*num_tokens: int*) → List[int]

Return a list of integers, one for each of the *num_tokens* tokens, defining how often that token gets inserted into the Bloom filter.

Parameters **num_tokens** (*int*) – number of tokens in the feature's value

Returns [k, …] with k's >= 0

classmethod from_json_dict(*json_dict: Dict[str, Union[str, SupportsInt]]*) → *clkhash.field_formats.StrategySpec*

class clkhash.field_formats.**StringSpec**(*identifier: str, hashing_properties:* clkhash.field_formats.FieldHashingProperties, *description: Optional[str] = None, regex: Optional[str] = None, case: str = 'mixed', min_length: int = 0, max_length: Optional[int] = None*)

Bases: *clkhash.field_formats.FieldSpec*

Represents a field that holds strings.

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

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

Variables

- **encoding** (*str*) – The encoding to use when converting the string to bytes. Refer to Python's documentation for possible values.
- **regex** – Compiled regular expression that entries must conform to. Present only if the specification is regex- based.
- **case** (*str*) – The casing of the entries. One of *'lower'*, *'upper'*, or *'mixed'*. Default is *'mixed'*. Present only if the specification is not regex-based.
- **min_length** (*int*) – The minimum length of the string. *None* if there is no minimum length. Present only if the specification is not regex-based.
- **max_length** (*int*) – The maximum length of the string. *None* if there is no maximum length. Present only if the specification is not regex-based.

classmethod from_json_dict(*json_dict: Dict[str, Any]*) → *clkhash.field_formats.StringSpec*

Make a StringSpec object from a dictionary containing its properties.

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

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

validate(*str_in: str*) → None

Validates an entry in the field.

Raises *InvalidEntryError* iff the entry is invalid.

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

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

Raises

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

clkhash.field_formats.**fhp_from_json_dict**(*json_dict: Dict[str, Any]*) → *clkhash.field_formats.FieldHashingProperties*

Make a *FieldHashingProperties* object from a dictionary.

Parameters **json_dict** (*dict*) – Conforming to the *hashingConfig* definition in the *v2* linkage schema.

Returns A *FieldHashingProperties* instance.

clkhash.field_formats.**spec_from_json_dict**(*json_dict: Dict[str, Any]*) → *clkhash.field_formats.FieldSpec*

Turns a dictionary into the appropriate FieldSpec object.

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

Raises *InvalidSchemaError* –

Returns An initialised instance of the appropriate FieldSpec subclass.

comparators

class clkhash.comparators.**AbstractComparison**

Bases: object

Abstract base class for all comparisons

abstract tokenize(*word: str*) → Iterable[str]

The tokenization function.

Takes a string and returns an iterable of tokens (as strings). This should be implemented in a way that the intersection of two sets of tokens produced by this function approximates the desired comparison criteria.

Parameters **word** – The string to tokenize.

Returns Iterable of tokens.

class clkhash.comparators.**ExactComparison**

Bases: *clkhash.comparators.AbstractComparison*

Enables exact comparisons

High similarity score if inputs are identical, low otherwise.

Internally, this is done by treating the whole input as one token. Thus, if you have chosen the 'bitsPerToken' strategy for hashing, you might want to adjust the value such that the corresponding feature gets an appropriate representation in the filter.

tokenize(*word: str*) → Iterable[str]

The tokenization function.

Takes a string and returns an iterable of tokens (as strings). This should be implemented in a way that the intersection of two sets of tokens produced by this function approximates the desired comparison criteria.

Parameters **word** – The string to tokenize.

Returns Iterable of tokens.

class clkhash.comparators.**NgramComparison**(*n: int, positional: Optional[bool] = False*)

Bases: *clkhash.comparators.AbstractComparison*

Enables 'n'-gram comparison for approximate string matching. An n-gram is a contiguous sequence of n items from a given text.

For Example: the 2-grams of 'clkhash' are ' c', 'cl', 'lk', 'kh', 'ha', 'as', 'sh', 'h '. Note the white- space in the first and last token. They serve the purpose to a) indicate the beginning and end of a word, and b) gives every character in the input text a representation in two tokens.

'n'-gram comparison of strings is tolerant to spelling mistakes, e.g., the strings 'clkhash' and 'clkhush' have 6 out of 8 2-grams in common. One wrong character will affect 'n' 'n'-grams. Thus, the larger you choose 'n', the more the error propagates.

A positional n-gram also encodes the position of the n-gram within the word. The positional 2-grams of 'clkhash' are '1 c', '2 cl', '3 lk', '4 kh', '5 ha', '6 as', '7 sh', '8 h '. Positional n-grams can be useful for comparing words where the position of the characters are important, e.g., postcodes or phone numbers.

Variables

- **n** – the n in n-gram, non-negative integer

- **positional** – enables positional n-gram tokenization

tokenize(*word: str*) → Iterable[str]

Produce *n*-grams of *word*.

Parameters **word** – The string to tokenize.

Returns Iterable of n-gram strings.

class clkhash.comparators.**NonComparison**

Bases: *clkhash.comparators.AbstractComparison*

Non comparison.

tokenize(*word: str*) → Iterable[str]

Null tokenizer returns empty Iterable.

FieldSpec Ignore has hashing_properties = None and get_tokenizer has to return something for this case, even though it's never called. An alternative would be to use an Optional[Callable]].

Parameters **word** – not used

Returns empty Iterable

class clkhash.comparators.**NumericComparison**(*threshold_distance: float*, *resolution: int*, *fractional_precision: int = 0*)

Bases: *clkhash.comparators.AbstractComparison*

enables numerical comparisons of integers or floating point numbers.

The numerical distance between two numbers relate to the similarity of the tokens produces by this comparison class. We implemented the idea of Vatsalan and Christen (Privacy-preserving matching of similar patients, Journal of Biomedical Informatics, 2015).

The basic idea is to encode a number's neighbourhood such that the neighbourhoods of close numbers overlap. For example, the neighbourhood of x=21 is 19, 20, 21, 22, 23, and the neighbourhood of y=23 is 21, 22, 23, 24, 25. These two neighbourhoods share three elements. The overlap of the neighbourhoods of two numbers increases the closer the numbers are to each other.

There are two parameters to control the overlap.

- ***threshold_distance*: the maximum distance which leads to an non-empty overlap. Neighbourhoods for points which** are further apart have no elements in common. (*)
- ***resolution*: controls how many tokens are generated. (the *b* in the paper). Given an interval of size** *threshold_distance* we create 'resolution tokens to either side of the mid-point plus one token for the mid-point. Thus, 2 * *resolution* + 1 tokens in total. A higher resolution differentiates better between different values, but should be chosen such that it plays nicely with the overall Bloom filter size and insertion strategy.

(*) the reality is a bit more tricky. We first have to quantize the inputs to multiples of *threshold_distance* / (2 * *resolution*), in order to get comparable neighbourhoods. For example, if we choose a *threshold_distance* of 8 and a *resolution* of 2, then, without quantization, the neighbourhood of x=25 would be [21, 23, 25, 27, 29] and for y=26 [22, 24, 26, 28, 30], resulting in no overlap. The quantization ensures that the inputs are mapped onto a common grid. In our example, the values would be quantized to even numbers (multiples of 8 / (2 * 2) = 2). Thus x=25 would be mapped to 26. The quantization has the side effect that sometimes two values which are further than *threshold_distance* but not more than *threshold_distance* + 1/2 quantization level apart can share a common token. For instance, a=24.99 would be mapped to 24 with a neighbourhood of [20, 22, 24, 26, 28], and b=16 neighbourhood is [12, 14, 16, 18, 20].

We produce the output tokens based on the neighbourhood in the following way. Instead of creating a neighbourhood around the quantized input with values dist_interval = *threshold_distance* / (2 * *resolution*) apart, we

instead multiply all values by (2 * *resolution*). This saves the division, which can introduce numerical inaccuracies. Thus, the tokens for x=25 are [88, 96, 104, 112, 120].

We are dealing with floating point numbers by quantizing them to integers by multiplying them with 10 ** *fractional_precision* and then rounding them to the nearest integer.

Thus, we don't support to full range of floats, but the subset between 2.2250738585072014e-(308 - fractional_precision - log(resolution, 10)) and 1.7976931348623157e+(308 - fractional_precision - log(resolution, 10))

Variables

- **threshold_distance** – maximum detectable distance. Points that are further apart won't have tokens in common.
- **resolution** – controls the amount of generated tokens. Total number of tokens will be 2 * resolution + 1
- **fractional_precision** – number of digits after the point to be considered

tokenize(*word: str*) → Iterable[str]

The tokenization function.

Takes a string and returns an iterable of tokens (as strings). This should be implemented in a way that the intersection of two sets of tokens produced by this function approximates the desired comparison criteria.

Parameters **word** – The string to tokenize.

Returns Iterable of tokens.

clkhash.comparators.**get_comparator**(*comp_desc: Dict[str, Any]*) → *clkhash.comparators.AbstractComparison*

Creates the comparator as defined in the schema. A comparator provides a tokenization method suitable for that type of comparison.

This function takes a dictionary, containing the schema definition. It returns a subclass of AbstractComparison.

1.3.2 Testing

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

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

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

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

Note several tests will be skipped by default.

1.3.3 Type Checking

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

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

1.4 Devops

1.4.1 Azure Pipeline

`clkhash` is automatically built and tested using Azure Pipeline for Windows environment, in the project *Anonlink <https://dev.azure.com/data61/Anonlink>*

Two pipelines are available:

- *Build pipeline <https://dev.azure.com/data61/Anonlink/_build?definitionId=2>*,
- *Release pipeline <https://dev.azure.com/data61/Anonlink/_release?definitionId=1>*.

Build Pipeline

The build pipeline is described by the script *azurePipeline.yml* which is using template resources from the folder *.azurePipeline*.

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 `clkhash`,
- 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 3.9` (i.e. the wheel for x86 and x64 and the *tar.gz* source distribution).

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

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

Most of the complexity is abstracted into the template in *.azurePipeline/wholeBuild.yml*.

Description of what is tested:

Python Version	Operating System	Standard pytest	Notebooks
pypy3	ubuntu-20.04	Yes	No
3.7	ubuntu-20.04	Yes	Yes
3.7	macos-10.15	Yes	Yes
3.7	vs2017-win2016 (x64)	Yes	No
3.7	vs2017-win2016 (x86)	Yes	No
3.8	ubuntu-20.04	Yes	Yes
3.8	macos-10.15	Yes	Yes

Build Artifacts

A pipeline artifact named **Release** is created by the build pipeline which contains the universal wheels and the source distributions 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 `clkhash` `Pypi` repository,
- *PYPI_PASSWORD*: password to push an artifact to `clkhash` `Pypi` repository for the user *PYPI_LOGIN*.

1.5 Research Notes

1.5.1 On the length of a CLK

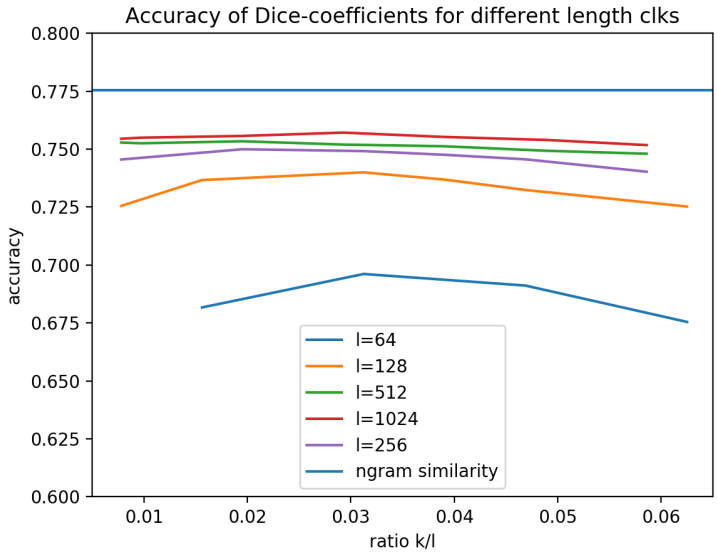
Note: This research note was authored by Wilko Henecka and first published on Apr 3, 2018 at https://github.com/data61/clkhash/issues/81

In literature, the length of a CLK `l` is either fixed to 1000 or 100. Depending on who is writing the paper. I read somewhere (unfortunately I cannot find it again…) that 100 is just as good as 1000. That made me suspicious. There should be some difference. But the more interesting question is, what is a good length for a CLK?

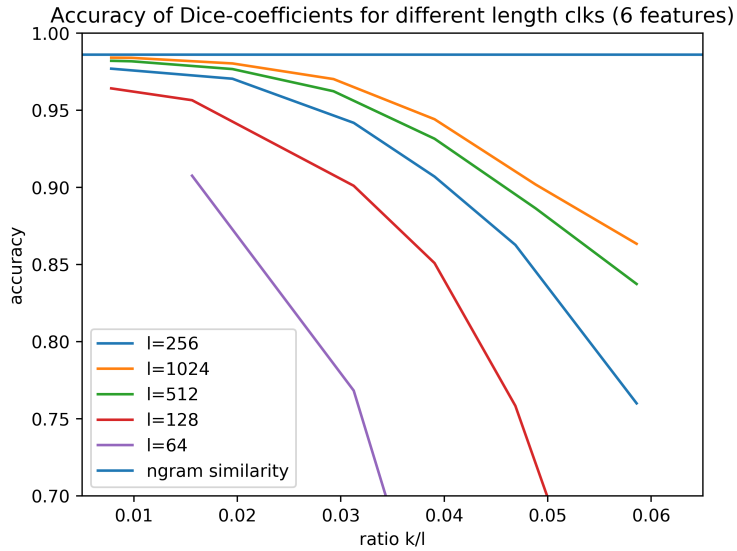
Only one way to find out… experiments!

I generated 100000 PII record pairs with the febrl tool (with the default perturbations). By accuracy in this context I mean: is the similarity score of the correct pair higher than all the other similarity score with the remaining entries in the dataset. The line for n-gram similarity is a upper bound. That's computed by generating the n-grams and computing the Dice coefficient directly from the n-grams. Essentially, if you wouldn't have any collisions in a Bloom filter, then it would compute exactly the same.

First experiment with the first three features: firstname, lastname, street number.



and second experiment with 6 features: firstname, lastname, street number, address_1, address_2, dob



1.5.2 Discussion

As expected, there is a difference between the result for different choices of l. Interestingly though, it seems that you get diminished returns when increasing the bloom filter size l.

The step from 64 to 128 is more significant than the one from 512 to 1024.

There is a slight decrease in accuracy for small values of k/l. This is most likely explained by the higher impact of collisions. Let's say you encode every n-gram with exactly one bit, then, in case of an collision, you cannot differentiate between the two different n-grams any more.

We can see a significant drop-off in accuracy for higher values of the k/l ratio. This is due to the fact, that the Bloom filter gets saturated for higher k, that is, the overwhelming majority of bits is set to 1.

1.5.3 Conclusion

These experiments suggests, that a good choice for k and l depend on the number of n-grams that have to be encoded in the Bloom filter. k should be chosen such that, in expectation, somewhere about half the bits in the filter will be set, in order to achieve good accuracy. However, k should also not be a small single digit number, thus in this case, l should be increased.

The most promising approach to allow the use of small Bloom filters with good accuracy is to control the number of n-grams per entity (think of SOUNDEX encoding of strings, or geo-encoding of addresses).

However, this is not very helpful in terms of privacy. As shown in 'Who Is 1011011111…1110110010? Automated Cryptanalysis of Bloom Filter Encryptions of Databases with Several Personal Identifier' and similar publications, the success of attacks on Bloom filters appears to be inverse proportional to the information embedded.

1.6 References

CHAPTER

TWO

EXTERNAL LINKS

- clkhash on Github
- clkhash on PyPi

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

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

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

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

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

PYTHON MODULE INDEX

C

INDEX

H

I

J

L

M

N

O

R

S

T

V