blocklib

Release 0.1.9

Confidential Computing Team

Apr 11, 2023

CONTENTS

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

Note that it is part of the anonlink system which includes libraries for encoding, command line tools and Rest API:

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

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

Install with pip:

```
pip install blocklib
```

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Tutorials

`blocklib` library is a Python-implementaion of various blocking techniques in record linkage. The tutorial `tutorial_blocking.ipynb` shows current supported blocking methods and how to use and assess them.

1.1.1 Blocking API

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

There are two main metrics to evaluate a blocking technique - reduction ratio and pair completeness.

Reduction Ratio

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

$$\text{reduction ratio} = 1 - \frac{\text{number of comparisons after blocking}}{N^2}$$

Pair Completeness

Pair completeness measure how many true matches are maintained after blocking. It is evalauted as

$$\text{pair completeness} = 1 - \frac{\text{number of true matches after blocking}}{\text{number of all true matches}}$$

Different blocking techniques have different methods to partition datasets in order to reduce as much number of comparisons as possible while maintain high pair completeness.

In this tutorial, we demonstrate how to use blocking in privacy preserving record linkage.

Load example Northern Carolina voter registration dataset:

[1]:
```
import blocklib
```

[2]:
```
# NBVAL_IGNORE_OUTPUT
import pandas as pd

df_alice = pd.read_csv('data/alice.csv')
df_alice.head()
```

[2]:
```
     recid givenname     surname       suburb    pc
0   761859      kate     chapman     brighton  4017
1  1384455      lian       hurse   carisbrook  3464
2  1933333   matthew       russo       bardon  4065
3  1564695  lorraine      zammit  minchinbury  2770
4  5971993      ingo  richardson  woolsthorpe  3276
```

In this dataset, `recid` is the voter registration number, with that we are able to verify the quality of a linkage between snapshots of this dataset taken at different times. `pc` refers to postcode.

The next step is to configure how to block the data. There are two privacy preserving blocking methods currently supported by `blocklib`:

1. Probabilistic signature (p-sig)
2. LSH based Λ-fold redundant (lambda-fold)

This tutorial will demonstrate using both of these, starting with probabilistic signatures.

Blocking Methods - Probabilistic signature (p-sig)

The high level idea behind this blocking method is that it uses signatures as the blocking key and places records having the same signatures into the same block. You can find the original paper here: Scalable Entity Resolution Using Probabilistic Signatures on Parallel Databases.

An example blocking configuration using probabilistic signatures:

[3]:
```
blocking_config = {
    "type": "p-sig",
    "version": 1,
    "config": {
        "blocking-features": ['givenname', 'surname'],
        "filter": {
            "type": "ratio",
            "max": 0.02,
            "min": 0.00,
        },
        "blocking-filter": {
            "type": "bloom filter",
            "number-hash-functions": 20,
            "bf-len": 2048,
        },
        "signatureSpecs": [
            [
                 {"type": "characters-at", "config": {"pos": [0]}, "feature": "givenname
↪"},
                 {"type": "characters-at", "config": {"pos": [0]}, "feature": "surname"},
            ],
            [
                {"type": "metaphone", "feature": "givenname"},
                {"type": "metaphone", "feature": "surname"},
            ]
        ]
    }
```

(continues on next page)

(continued from previous page)

```
}
```

The blocking config can be fully validated to ensure all required types are present.

[4]:
```
blocklib.validation.validate_blocking_schema(blocking_config)
```

[4]:
```
BlockingSchemaModel(version=1, type=<BlockingSchemaTypes.psig: 'p-sig'>,␣
↪config=PSigConfig(record_id_column=None, blocking_features=['givenname', 'surname'],␣
↪null_sentinel='', filter=PSigFilterRatioConfig(type='ratio', max=0.02, min=0.0),␣
↪blocking_filter=PSigBlockingBFFilterConfig(type='bloom filter', number_of_hash_
↪functions=20, bloom_filter_length=2048, compress_block_key=False),␣
↪signatures=[[PSigCharsAtSignatureSpec(type=<PSigSignatureTypes.chars_at: 'characters-at
↪'>, feature='givenname', config=PSigCharsAtSignatureConfig(pos=[0])),␣
↪PSigCharsAtSignatureSpec(type=<PSigSignatureTypes.chars_at: 'characters-at'>, feature=
↪'surname', config=PSigCharsAtSignatureConfig(pos=[0]))],␣
↪[PSigMetaphoneSignatureSpec(type=<PSigSignatureTypes.metaphone: 'metaphone'>, feature=
↪'givenname'), PSigMetaphoneSignatureSpec(type=<PSigSignatureTypes.metaphone: 'metaphone
↪'>, feature='surname')]]))
```

Step 1 - Generate Signatures

For a record `r`, a signature is a sub-record derived from record `r` with a signature strategy. An example signature strategy is to concatenate the initials of first and last name, e.g., the signature for record `"John White"` is `"JW"`.

`blocklib` provides the following signature strategies:

- `feature-value`: the signature is generated by returning the selected feature
- `characters-at`: the signature is generated by selecting a single character or a sequence of characters from selected feature
- `metaphone`: the signature is generated by phonetic encoding the selected feature using metaphone

The output of this step is a reversed index where keys are generated signatures / blocking key, and the values are lists of corresponding record IDs. A record ID could be row index or the actual record identifier if it is available in the dataset.

Signature strategies are defined in the `signatureSpecs` section. For example, in the above configuration, we are going to generate two signatures for each record. The first signature produces initials:

```
[
     {"type": "characters-at", "config": {"pos": [0]}, "feature": "givenname"},
     {"type": "characters-at", "config": {"pos": [0]}, "feature": "surname"}
]
```

The second signature is generated by a combination of how the two components of a person's name sounds:

```
[
    {"type": "metaphone", "feature": "givenname"},
    {"type": "metaphone", "feature": "surname"}
]
```

That is phonetic encoding of first name and last name.

One signature corresponds to one block. I will use signature and block interchangeably but they mean the same thing.

Step 2 - Filter Signatures

Signature strategies can create blocks with many records, and blocks with just one record. To impose limits on the minimum and maximum block size `blocklib` provides configurable filtering.

For example, in the above configuration, the filter is configured as:

```
{
    "type": "ratio",
    "max": 0.02,
    "min": 0.001
}
```

`blocklib` will filter out all signatures / blocks whose number of records is greater than 2% of number of total records or is less than 0.1% of number of total records. Note these percentages are based on the data provided to `blocklib` so only use on roughly symmetric sized record linkage.

Absolute filtering is also supported to filter by number of records. An example `filter` configuration:

```
{
    "type": "count",
    "max": 100,
    "min": 5
}
```

Step 3 - Anonymization

Given the aim of privacy preserving record linkage, the signatures themselves (e.g. `"JW"`) are not going to be shared, instead following the `p-sig` paper, the signatures all get encoded into a Bloom Filter. Here we use one Bloom Filter and map all filtered signatures into that Bloom Filter.

```
"blocking-filter": {
    "type": "bloom filter",
    "number-hash-functions": 20,
    "bf-len": 2048,
}
```

After anonymization, the signature becomes the set of indices of bits 1 in the bloom filter and hence can preserve the privacy of data for each data provider.

Blocking Data

Now that we have configured how the P-Sig blocking will work, we can carry out our blocking job with `blocklib`. Note `blocklib` only accept list of tuples or lists as input data, so some pre-processing may be necessary. Example data input for `blocklib`:

```
[
    [761859, 'kate', 'chapman', 'brighton', 4017],
    [1384455, 'lian', 'hurse', 'carisbrook', 3464],
    [1933333, 'matthew', 'russo', 'bardon', 4065],
    [1564695, 'lorraine', 'zammit', 'minchinbury', 2770],
    [5971993, 'ingo', 'richardson', 'woolsthorpe', 3276]
]
```

Step 1 - Generate Candidate Blocks for Party A - Alice

[5]:
```
alice = df_alice.to_dict(orient='split')
print("Example PII", alice['data'][0])
```

```
Example PII [761859, 'kate', 'chapman', 'brighton', 4017]
```

[6]:
```
from blocklib import generate_candidate_blocks

block_obj_alice = generate_candidate_blocks(alice['data'], blocking_config, header=alice[
↪'columns'])
block_obj_alice.print_summary_statistics()
```

```
Statistics for the generated blocks:
        Number of Blocks:   5028
        Minimum Block Size: 1
        Maximum Block Size: 61
        Average Block Size: 1.8341
        Median Block Size:  1
        Standard Deviation of Block Size:  3.8372
        Coverage:           100.0%
Individual statistics for each strategy:
Strategy: 0
        Number of Blocks:   503
        Minimum Block Size: 1
        Maximum Block Size: 61
        Average Block Size: 9.167
        Median Block Size:  6
        Standard Deviation of Block Size:  9.3427
        Coverage:           100.0%
Strategy: 1
        Number of Blocks:   4534
        Minimum Block Size: 1
        Maximum Block Size: 7
        Average Block Size: 1.017
        Median Block Size:  1
        Standard Deviation of Block Size:  0.1584
        Coverage:           100.0%
```

You can print the statistics of the blocks in order to inspect the block distribution and decide if this is a good blocking result.

`generate_candidate_blocks` returns a `CandidateBlockingResult`, the attribute we are most interested in is `blocks`, a `dict` that maps signatures to lists of records.

[7]:
```
list(block_obj_alice.blocks.keys())[0]
```

[7]:
```
'(1920, 1031, 142, 401, 1560, 671, 1830, 941, 52, 1211, 1470, 581, 1740, 851, 2010, 1121,
↪ 232, 491, 1650, 761)'
```

To protect privacy, the signature / blocking key is not the original signature such as `JW`. Instead, it is a list of mapped indices of bits set to 1 in the Bloom Filter for the original signature. Next we want to do the same thing for another party - *enter Bob*.

Step2 - Generate Candidate Blocks for Party B - Bob

[8]:
```
# NBVAL_IGNORE_OUTPUT
df_bob = pd.read_csv('data/bob.csv')
bob = df_bob.to_dict(orient='split')
block_obj_bob = generate_candidate_blocks(bob['data'], blocking_config, header=bob[
↪'columns'])
block_obj_bob.print_summary_statistics()
```

```
Statistics for the generated blocks:
        Number of Blocks:   5018
        Minimum Block Size: 1
        Maximum Block Size: 59
        Average Block Size: 1.8378
        Median Block Size:  1
        Standard Deviation of Block Size:  3.8383
        Coverage:           100.0%
Individual statistics for each strategy:
Strategy: 0
        Number of Blocks:   500
        Minimum Block Size: 1
        Maximum Block Size: 59
        Average Block Size: 9.222
        Median Block Size:  6
        Standard Deviation of Block Size:  9.3374
        Coverage:           100.0%
Strategy: 1
        Number of Blocks:   4529
        Minimum Block Size: 1
        Maximum Block Size: 4
        Average Block Size: 1.0181
        Median Block Size:  1
        Standard Deviation of Block Size:  0.1445
        Coverage:           100.0%
```

Generate Final Blocks

Now we have *candidate* blocks from both parties, we can generate final blocks by only including signatures that appear in both parties. Instead of directly comparing signatures, the algorithm maps the list of signatures into a Bloom Filter for each party called the candidate blocking filter, and then creates the combined blocking filter by only retaining the bits that are present in both candidate filters.

[9]:
```
from blocklib import generate_blocks

filtered_blocks_alice, filtered_blocks_bob = generate_blocks([block_obj_alice, block_obj_
↪bob], K=2)
print('Alice: {} out of {} blocks are in common'.format(len(filtered_blocks_alice),␣
↪len(block_obj_alice.blocks)))
print('Bob:   {} out of {} blocks are in common'.format(len(filtered_blocks_bob),␣
↪len(block_obj_bob.blocks)))
```

```
Alice: 2794 out of 5028 blocks are in common
Bob:   2794 out of 5018 blocks are in common
```

Assess Blocking

We can assess the blocking result when we have ground truth. There are two main metrics to assess blocking result as mentioned in the beginning of this tutorial. Here is a recap:

- reduction ratio: relative reduction in the number of record pairs to be compared.
- pair completeness: the percentage of true matches after blocking

[10]:
```
# NBVAL_IGNORE_OUTPUT
from blocklib.evaluation import assess_blocks_2party

subdata1 = [x[0] for x in alice['data']]
subdata2 = [x[0] for x in bob['data']]

rr, pc = assess_blocks_2party([filtered_blocks_alice, filtered_blocks_bob],
                              [subdata1, subdata2])
print(f'reduction ratio:   {round(rr, 3)}')
print(f'pair completeness: {pc}')
```

```
assessing blocks: 100%|| 2794/2794 [00:00<00:00, 687485.94key/s]
```

```
reduction ratio:   0.996
pair completeness: 1.0
```

Blocking Methods - LSH Based Λ-fold Redundant

Now we look the other blocking method that we support - LSH Based Λ-fold Redundant blocking. This blocking method uses a list of selected bits selected randomly from Bloom Filter for each record as block keys. Λ refers the degree of redundancy i.e. we will conduct LSH-based blocking Λ times, each forms a blocking group. Then those blocking groups are combined into one blocking results. This will make a record redundant Λ times but will increase the recall.

Let's see an example config, this time selecting the blocking features using column indices instead of column names:

[11]:
```
blocking_config = {
    "type": "lambda-fold",
    "version": 1,
    "config": {
        "blocking-features": [1, 2],
        "Lambda": 5,
        "bf-len": 2048,
        "num-hash-funcs": 10,
        "K": 40,
        "random_state": 0,
        "input-clks": False
    }
}

blocklib.validation.validate_blocking_schema(blocking_config)
```

[11]:
```
BlockingSchemaModel(version=1, type=<BlockingSchemaTypes.lambdafold: 'lambda-fold'>,␣
↪config=LambdaConfig(record_id_column=None, blocking_features=[1, 2], null_sentinel='',␣
↪Lambda=5, bloom_filter_length=2048, number_of_hash_functions=10, K=40, block_
↪encodings=False, random_state=0))
```

(continues on next page)

(continued from previous page)

Now let's explain the meaning of each argument:

- blocking-features: a list of feature indice that we are going to use to generate blocks
- Lambda: this number denotes the degree of redundancy - H^i, $i = 1, 2, ..., \Lambda$ where each H^i represents one independent blocking group
- bf-len: length of Bloom Filter for each record
- num-hash-funcs: number of hash functions used to map record to Bloom Filter
- K: number of bits we selected from Bloom Filter for each record
- random_state: control random seed

Then we can carry out the blocking job and assess the result just like above steps

[12]:
```
print('Generating candidate blocks for Alice:')
block_obj_alice = generate_candidate_blocks(alice['data'], blocking_config)
block_obj_alice.print_summary_statistics()
print()
print('Generating candidate blocks for Bob: ')
block_obj_bob = generate_candidate_blocks(bob['data'], blocking_config)
block_obj_bob.print_summary_statistics()
```

```
Generating candidate blocks for Alice:
Statistics for the generated blocks:
        Number of Blocks:   6050
        Minimum Block Size: 1
        Maximum Block Size: 873
        Average Block Size: 3.8107
        Median Block Size:  1
        Standard Deviation of Block Size:  20.9703

Generating candidate blocks for Bob:
Statistics for the generated blocks:
        Number of Blocks:   6085
        Minimum Block Size: 1
        Maximum Block Size: 862
        Average Block Size: 3.7888
        Median Block Size:  1
        Standard Deviation of Block Size:  20.715
```

[13]:
```
filtered_blocks_alice, filtered_blocks_bob = generate_blocks([block_obj_alice, block_obj_
↪bob], K=2)
print('Alice: {} out of {} blocks are in common'.format(len(filtered_blocks_alice),␣
↪len(block_obj_alice.blocks)))
print('Bob:   {} out of {} blocks are in common'.format(len(filtered_blocks_bob),␣
↪len(block_obj_bob.blocks)))
```

```
Alice: 4167 out of 6050 blocks are in common
Bob:   4167 out of 6085 blocks are in common
```

[14]:
```
# NBVAL_IGNORE_OUTPUT
rr, pc = assess_blocks_2party([filtered_blocks_alice, filtered_blocks_bob],
                              [subdata1, subdata2])
print(f'reduction ratio:   {round(rr, 3)}')
print(f'pair completeness: {pc}')
```

```
assessing blocks: 100%|| 4167/4167 [00:00<00:00, 347130.33key/s]
```

```
reduction ratio:   0.872
pair completeness: 1.0
```

[ ]:

1.2 Blocking Schema

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

Currently we support two blocking methods:

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

which are proposed by the following publications:

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

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

1.2.1 Basic Structure

A blocking schema consists of three parts:

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

1.2.2 Example Schema

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

1.2.3 Schema Components

type

String value which describes the blocking method.

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

version

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

config

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

Specific configuration of supported blocking methods can be found here:

- *config of p-sig*
- *config of lambda-fold*

Probabilistic Signature Configuration

attribute	type	description
blocking-features	list[integer]	specify which features are used in blocks generation
filter	dictionary	filtering threshold
blocking-filter	dictionary	type of filter to generate blocks
null-sentinel	Any	defines the object that represents the NULL value in the dataset. Defaults to the empty string “”
signatureSpecs	list of lists	signature strategies where each list is a combination of signature strategies

Filter Configuration

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

Blocking-filter Configuration

A blocking filter is represented as a string describing the bit positions in the Bloom filter set to one, e.g.: “(3, 265, 403, 665, 927, 165, 41, 303, 565, 827, 965, 203, 465, 727, 865, 103, 365, 627, 503, 765)”. This representation consumes a considerable amount of space. If the indices are not needed for further processing, you can tell blocklib to replace these strings with a 5 byte hash by setting the *compress-block-key* flag.

attribute	type	description
type	string	currently we only support “bloom filter”
number-hash-functions	integer	this specifies how many bits will be flipped for each signature
bf-len	integer	defines the length of blocking filter, for bloom filter usually this is 1024 or 2048
compress-block-key	boolean	optional. Replace the block key by a 5 bytes hash versions of itself.

SignatureSpecs Configurations

It is better to illustrate this one with an example:

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

(continues on next page)

(continued from previous page)

```
  ]
}
```

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

The following specifies the current supported signature strategies:

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

Finally a full example of p-sig blocking schema:

```
{
 "type": "p-sig",
 "version": 1,
 "config": {
     "blocking-features": [1],
     "filter": {
         "type": "ratio",
         "max": 0.02,
         "min": 0.00,
     },
     "blocking-filter": {
         "type": "bloom filter",
         "number-hash-functions": 4,
         "bf-len": 2048,
     },
     "null-sentinel": None,
     "signatureSpecs": [
         [
              {"type": "characters-at", "config": {"pos": [0]}, "feature": 1},
              {"type": "characters-at", "config": {"pos": [0]}, "feature": 2},
         ],
         [
             {"type": "metaphone", "feature": 1},
             {"type": "metaphone", "feature": 2},
         ]
     ]
   }
 }
```

LSH based λ-fold Configuration

attribute	type	description
blocking-features	list[integer]	specify which features are used in blocks generation
null-sentinel	Any	defines the object that represents the NULL value in the dataset. Defaults to the empty string ""
Lambda	integer	denotes the degree of redundancy - H^i, $i = 1, 2, ..., \Lambda$ where each H^i represents one independent blocking group
bf-len	integer	length of bloom filter
num-hash-funcs	integer	number of hash functions used to map record to Bloom filter
K	integer	number of bits we will select from Bloom filter for each reocrd
random_state	integer	control random seed
input-clks	boolean	input data is CLKS if true else input data is not CLKS

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

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

1.3 Python API

1.3.1 Block Generator

Module that implements final block generations.

blocklib.blocks_generator.**check_block_object**(*candidate_block_objs: Sequence[*blocklib.candidate_blocks_generator.CandidateBlockingResult*]*)

Check candidate block objects type and their states type.

Raises **TypeError** – if conditions aren't met.

Parameters **candidate_block_objs** – A list of candidate block result objects from 2 data providers

blocklib.blocks_generator.**generate_blocks**(*candidate_block_objs: Sequence[*blocklib.candidate_blocks_generator.CandidateBlockingResult*]*, *K: int*) → List[Dict[Any, List[Any]]]

Generate final blocks given list of candidate block objects from 2 or more than 2 data providers.

Parameters

- **candidate_block_objs** – A list of CandidateBlockingResult from multiple data providers
- **K** – it specifies the minimum number of occurrence for records to be included in the final blocks

Returns List of dictionaries, filter out records that appear in less than K parties

blocklib.blocks_generator.**generate_blocks_psig**(*reversed_indices: Sequence[Dict]*, *block_states: Sequence[*blocklib.pprlpsig.PPRLIndexPSignature*]*, *threshold: int*)

Generate blocks for P-Sig

Parameters

- **reversed_indices** – A list of dictionaries where key is the block key and value is a list of record IDs.
- **block_states** – A list of PPRLIndex objects that hold configuration of the blocking job
- **threshold** – int which decides a pair when number of 1 bits in bloom filter is large than or equal to threshold

Returns A list of dictionaries where blocks that don't contain any matches are deleted

blocklib.blocks_generator.**generate_reverse_blocks**(*reversed_indices: Sequence[Dict]*)

Invert a map from "blocks to records" to "records to blocks".

Parameters **reversed_indices** – A list of dictionaries where key is the block key and value is a list of record IDs.

Returns A list of dictionaries where key is the record ID and value is a set of blocking keys the record belongs to.

1.3.2 Blocks

class blocklib.candidate_blocks_generator.**CandidateBlockingResult**(*blocking_result: blocklib.pprlindex.ReversedIndexResult*, *state:* blocklib.pprlindex.PPRLIndex)

Object for holding candidate blocking results.

Variables

- **blocks** – a dictionary that contains a mapping from the block ID to the record IDs in that block.
- **state** – A PPRLIndex state that contains the configuration of blocking
- **stats** – a dictionary containing the summary statistics of the generated blocks

print_summary_statistics(*output: typing.TextIO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *round_ndigits: int = 4*)

Print the summary statistics of this candidate blocking result to 'output'. :param output: a file like object to write to. Defaults to sys.stdout :param round_ndigits: round floating point numbers to ndigits precision. Defaults to 4.

blocklib.candidate_blocks_generator.**generate_candidate_blocks**(*data: Sequence[Tuple[str, ...]]*, *blocking_schema: Dict*, *header: Optional[List[str]] = None*) → *blocklib.candidate_blocks_generator.CandidateBlockingRes*

Parameters

- **data** – list of tuples E.g. ('0', 'Kenneth Bain', '1964/06/17', 'M')
- **blocking_schema** – A description of how the signatures should be generated. See *Blocking Schema*
- **header** – column names (optional) Program should throw exception if block features are string but header is None

Returns A 2-tuple containing A list of "signatures" per record in data. Internal state object from the signature generation (or None).

1.3.3 Base PPRL Index

class blocklib.pprlindex.**PPRLIndex**(*config: Union[*blocklib.validation.psig_validation.PSigConfig, blocklib.validation.lambda_fold_validation.LambdaConfig*]*)

Base class for PPRL indexing/blocking.

build_reversed_index(*data: Sequence[Sequence]*, *header: Optional[List[str]] = None*)

Method which builds the index for all database.

Parameters

- **data** – list of tuples, PII dataset
- **header** – file header, optional

Return type ReversedIndexResult

See derived classes for actual implementations.

get_feature_to_index_map(*data: Sequence[Sequence]*, *header: Optional[List[str]] = None*)

Return feature name to feature index mapping if there is a header and feature is of type string.

classmethod select_reference_value(*reference_data: Sequence[Sequence]*, *ref_data_config: Dict*)

Load reference data for methods need reference.

set_blocking_features_index(*blocking_features*, *feature_to_index: Optional[Dict[str, int]] = None*)

Set value of member variable blocking features index.

self.blocking_features could be string (column name) or int (column index) self.blocking_features_index must be int (column index)

1.3.4 Signature Generator

blocklib.signature_generator.**generate_by_char_at**(*attr_ind: int*, *dtuple: Sequence*, *pos: List[Any]*)

Generate signatures by select subset of characters in original features.

```
>>> res = generate_by_char_at(2, ('harry potter', '4 Privet Drive', 'Little Whinging
↪', 'Surrey'), [0, 3])
>>> assert res == 'Lt'
>>> res = generate_by_char_at(2, ('harry potter', '4 Privet Drive', 'Little Whinging
↪', 'Surrey'), [":4"])
>>> assert res == 'Litt'
```

blocklib.signature_generator.**generate_by_feature_value**(*attr_ind: int*, *dtuple: Sequence*)

Generate signatures by simply return original feature at attr_ind.

blocklib.signature_generator.**generate_by_metaphone**(*attr_ind: int*, *dtuple: Sequence*)

Generate a phonetic encoding of features using metaphone.

```
>>> generate_by_metaphone(0, ('Smith', 'Schmidt', 2134))
'SM0XMT'
```

blocklib.signature_generator.**generate_signatures**(*signature_strategies: List[List[Union[*blocklib.validation.psig_validation.PSigCharsAtSignat blocklib.validation.psig_validation.PSigMetaphoneSignatureSpec, blocklib.validation.psig_validation.PSigFeatureValueSignatureSpec*]]]*, *dtuple: Sequence*, *null_sentinel: Any*, *feature_to_index: Optional[Dict[str, int]] = None*)

Generate signatures for one record.

Parameters

- **signature_strategies** – A list of PSigSignatureModel instances each describing a strategy to generate signatures.
- **dtuple** – Raw data to generate signatures from
- **null_sentinel** – String that represents the NULL value in the dataset
- **feature_to_index** – Mapping from feature name to feature index

Return signatures set of str

1.3.5 P-Sig

class blocklib.pprlpsig.**PPRLIndexPSignature**(*config: Union[*blocklib.validation.psig_validation.PSigConfig, *Dict]*)

Class that implements the PPRL indexing technique:

Reference scalability entity resolution using probability signatures on parallel databases.

This class includes an implementation of p-sig algorithm.

build_reversed_index(*data: Sequence[Sequence]*, *header: Optional[List[str]] = None*)

Build inverted index given P-Sig method.

Configuration

class blocklib.pprlpsig.**PSigConfig**(*, *filter: Union[*blocklib.validation.psig_validation.PSigFilterRatioConfig, blocklib.validation.psig_validation.PSigFilterCountConfig*]*, *signatureSpecs: List[List[Union[*blocklib.validation.psig_validation.PSigCharsAtSignatureSpec, blocklib.validation.psig_validation.PSigMetaphoneSignatureSpec, blocklib.validation.psig_validation.PSigFeatureValueSignatureSpec*]]]*, ***extra_data: Any*)

blocking_filter: *blocklib.validation.psig_validation.PSigBlockingBFFilterConfig*

filter: Union[*blocklib.validation.psig_validation.PSigFilterRatioConfig*, *blocklib.validation.psig_validation.PSigFilterCountConfig*]

signatures: List[List[Union[*blocklib.validation.psig_validation.PSigCharsAtSignatureSpec*, *blocklib.validation.psig_validation.PSigMetaphoneSignatureSpec*, *blocklib.validation.psig_validation.PSigFeatureValueSignatureSpec*]]]

class blocklib.validation.psig_validation.**PSigBlockingBFFilterConfig**(*, *type: Literal['bloom filter']*, ***extra_data: Any*)

bloom_filter_length: int

compress_block_key: Optional[bool]

number_of_hash_functions: int

type: Literal['bloom filter']

class blocklib.validation.psig_validation.**PSigCharsAtSignatureConfig**(*, *pos: List[Union[blocklib.validation.constrained_t str]]*)

pos: List[Union[blocklib.validation.constrained_types.PositiveInt, str]]

class blocklib.validation.psig_validation.**PSigCharsAtSignatureSpec**(*, *type: typing.Literal[<PSigSignatureTypes.chars_at: 'characters-at'>]*, *feature: typing.Union[int, str]*, *config: blocklib.validation.psig_validation.PSigCharsAtSign*

config: *blocklib.validation.psig_validation.PSigCharsAtSignatureConfig*

type: Literal[<PSigSignatureTypes.chars_at: 'characters-at'>]

class blocklib.validation.psig_validation.**PSigConfig**(*, *filter: Union[blocklib.validation.psig_validation.PSigFilterRatioConfig, blocklib.validation.psig_validation.PSigFilterCountConfig]*, *signatureSpecs: List[List[Union[blocklib.validation.psig_validation.PSigCharsAtS blocklib.validation.psig_validation.PSigMetaphoneSignatureSpec, blocklib.validation.psig_validation.PSigFeatureValueSignatureSpec]]]*, ***extra_data: Any*)

blocking_features: Union[List[int], List[str]]

blocking_filter: *blocklib.validation.psig_validation.PSigBlockingBFFilterConfig*

filter: Union[*blocklib.validation.psig_validation.PSigFilterRatioConfig*, *blocklib.validation.psig_validation.PSigFilterCountConfig*]

null_sentinel: Any

record_id_column: Optional[int]

signatures: List[List[Union[*blocklib.validation.psig_validation.PSigCharsAtSignatureSpec*, *blocklib.validation.psig_validation.PSigMetaphoneSignatureSpec*, *blocklib.validation.psig_validation.PSigFeatureValueSignatureSpec*]]]

class blocklib.validation.psig_validation.**PSigFeatureValueSignatureSpec**(*, *type: typing.Literal[<PSigSignatureTypes.feature 'feature-value'>]*, *feature: typing.Union[int, str]*)

type: Literal[<PSigSignatureTypes.feature_value: 'feature-value'>]

class blocklib.validation.psig_validation.**PSigFilterConfigBase**(*, *type: str*)

type: str

class blocklib.validation.psig_validation.**PSigFilterCountConfig**(*, *type: Literal['count']*, *max: blocklib.validation.constrained_types.PositiveInt*, *min: blocklib.validation.constrained_types.PositiveInt*)

max: blocklib.validation.constrained_types.PositiveInt

min: blocklib.validation.constrained_types.PositiveInt

type: Literal['count']

class blocklib.validation.psig_validation.**PSigFilterRatioConfig**(*, *type: Literal['ratio']*, *max: blocklib.validation.constrained_types.UnitFloat*, *min: blocklib.validation.constrained_types.UnitFloat = 0.0*)

max: blocklib.validation.constrained_types.UnitFloat

min: blocklib.validation.constrained_types.UnitFloat

type: Literal['ratio']

class blocklib.validation.psig_validation.**PSigMetaphoneSignatureSpec**(**, type: typing.Literal[<PSigSignatureTypes.metaphone: 'metaphone'>], feature: typing.Union[int, str]*)

type: Literal[<PSigSignatureTypes.metaphone: 'metaphone'>]

class blocklib.validation.psig_validation.**PSigSignatureSpecBase**(**, type: str, feature: Union[int, str]*)

feature: Union[int, str]

type: str

class blocklib.validation.psig_validation.**PSigSignatureTypes**(*value*)

An enumeration.

chars_at = 'characters-at'

feature_value = 'feature-value'

metaphone = 'metaphone'

1.3.6 Lambda Fold

class blocklib.pprllambdafold.**PPRLIndexLambdaFold**(*config: Union[*blocklib.validation.lambda_fold_validation.LambdaConfig*, Dict]*)

Class that implements the PPRL indexing technique:

An LSH-Based Blocking Approach with a Homomorphic Matching Technique for Privacy-Preserving Record Linkage.

This class includes an implementation of Lambda-fold redundant blocking method.

build_reversed_index(*data: Sequence[Any]*, *header: Optional[List[str]] = None*)

Build inverted index for PPRL Lambda-fold blocking method.

Parameters

- **data** – list of lists
- **header** – file header, optional

Returns reversed index as ReversedIndexResult

Configuration

class blocklib.pprllambdafold.**LambdaConfig**(*, *Lambda: int*, *K: int*, *random_state: int*, ***extra_data: Any*)

K: int

Lambda: int

block_encodings: bool

bloom_filter_length: int

number_of_hash_functions: int

random_state: int

class blocklib.validation.lambda_fold_validation.**LambdaConfig**(*, *Lambda: int*, *K: int*, *random_state: int*, ***extra_data: Any*)

K: int

Lambda: int

block_encodings: bool

blocking_features: Union[List[int], List[str]]

bloom_filter_length: int

null_sentinel: Any

number_of_hash_functions: int

random_state: int

record_id_column: Optional[int]

1.3.7 Internal

`blocklib` uses pydantic for validation.

class blocklib.validation.**BlockingSchemaModel**(*, *version: int*, *type:* blocklib.validation.BlockingSchemaTypes, *config: Union[*blocklib.validation.psig_validation.PSigConfig, blocklib.validation.lambda_fold_validation.LambdaConfig*]*)

config: Union[*blocklib.validation.psig_validation.PSigConfig*, *blocklib.validation.lambda_fold_validation.LambdaConfig*]

classmethod config_gen(*config_to_validate*, *values*) → Union[*blocklib.validation.psig_validation.PSigConfig*, *blocklib.validation.lambda_fold_validation.LambdaConfig*]

type: *blocklib.validation.BlockingSchemaTypes*

classmethod validate_config(*config_to_validate*, *values*)

version: int

class blocklib.validation.**BlockingSchemaTypes**(*value*)

An enumeration.

lambdafold = 'lambda-fold'

psig = 'p-sig'

blocklib.validation.**load_schema**(*file_name: str*)

blocklib.validation.**validate_blocking_schema**(*config: Dict*) → *blocklib.validation.BlockingSchemaModel*

Validate blocking schema data with pydantic.

Raises **ValueError** – exceptions when passed an invalid config.

1.3.8 Encoding

Class to implement privacy preserving encoding.

blocklib.encoding.**flip_bloom_filter**(*string: str*, *bf_len: int*, *num_hash_funct: int*)

Hash string and return indices of bits that have been flipped correspondingly.

Parameters

- **string** – string: to be hashed and to flip bloom filter
- **bf_len** – int: length of bloom filter
- **num_hash_funct** – int: number of hash functions

Returns bfset: a set of integers - indices that have been flipped to 1

blocklib.encoding.**generate_bloom_filter**(*list_of_strs: List[str]*, *bf_len: int*, *num_hash_funct: int*)

Generate a bloom filter given list of strings.

Parameters

- **return_cbf_index_sig_map** –
- **list_of_strs** –
- **bf_len** –
- **num_hash_funct** –

Returns bloom_filter_vector if return_cbf_index_sig_map is False else (bloom_filter_vector, cbf_index_sig_map)

1.4 Development

1.4.1 Testing

Make sure you have all the required dependencies before running the tests:

```
$ poetry install
```

Now run the unit tests and print out code coverage with *pytest*:

```
$ poetry run pytest --cov=blocklib
```

1.4.2 Type Checking

`blocklib` uses static typechecking with `mypy`. To run the type checker as configured to run in the CI:

```
$ poetry run mypy blocklib --ignore-missing-imports --strict-optional --no-implicit-
↪optional --disallow-untyped-calls
```

1.5 Devops

1.5.1 GitHub Actions

`blocklib` is automatically built and tested using GitHub actions.

There are currently three workflows:

Testing

The testing workflow is defined in the script `.github/workflows/python-test.yml`.

It consists of two jobs

- *Unit tests* - tests the library using `pytest` with different combinations of python versions and operating systems.
- *Notebook tests* - tests the tutorial notebooks using `pytest`.

Type Checking

The type checking workflow is defined in the script `.github/workflows/typechecking.yml`. It runs typechecking with mypy.

Build and Publish

The build and publish workflow is defined in the script `.github/workflows/build_publish.yml`.

It consists of two jobs:

- *Build distribution Packages* - packages blocklib into wheels and saves the build artifacts.
- *Publish to PyPI* - uploads the built artifacts to PyPI.

Note: The *Publish to PyPI* job is only triggered on a GitHub release.

1.6 References

CHAPTER

TWO

EXTERNAL LINKS

- blocklib on Github
- blocklib on Pypi

BIBLIOGRAPHY

[Zhang2018] Y Zhang, KS Ng, T Churchill, P Christen - Proceedings of the 27th ACM (2018). Scalable Entity Resolution Using Probabilistic Signatures on Parallel Databases

[Karapiperis20144] Karapiperis, D. and Verykios, V.S. - IEEE Transactions on Knowledge and Data Engineering, 27(4), pp.909-921.(2014) `<https://www.computer.org/csdl/journal/tk/2015/04/06880802/13rRUxASubY>`_An LSH-Based Blocking Approach with a Homomorphic Matching Technique for Privacy-Preserving Record Linkage

PYTHON MODULE INDEX

b

INDEX

R

S

T

V