

---

# **blocklib**

***Release 0.1.1***

**Feb 13, 2020**



---

## Contents

---

|          |                           |           |
|----------|---------------------------|-----------|
| <b>1</b> | <b>Table of Contents</b>  | <b>3</b>  |
| 1.1      | Tutorials . . . . .       | 3         |
| 1.2      | Blocking Schema . . . . . | 9         |
| 1.3      | Development . . . . .     | 13        |
| 1.4      | Devops . . . . .          | 14        |
| 1.5      | References . . . . .      | 14        |
| <b>2</b> | <b>External Links</b>     | <b>15</b> |
|          | <b>Bibliography</b>       | <b>17</b> |



*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 anonlink system which includes libraries for encoding, command line tools and Rest API:

- [clckhash](#)
- [anonlink-client](#)
- [anonlink-entity-service](#)

Blocklib is Apache 2.0 licensed, supports Python version 3.5+ and run on Windows, OSX and Linux.

Install with pip:

```
pip install blocklib
```



## 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 Nothern Carolina voter registration dataset:

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

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

|   | 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. So we are able to verify the quality of a linkage between snapshots of this dataset taken at different times. `pc` refers to postcode.

Next step is to config a blocking job. Before we do that, let's look at the blocking methods we are currently supporting:

1. Probabilistic signature (p-sig)
2. LSH based  $\Lambda$ -fold redundant (lambda-fold)

Let's firstly look at P-sig

### Blocking Methods - Probabilistic signature (p-sig)

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

Detailed steps and explanations are in the following.

Let's see an example of configuration for p-sig

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

(continues on next page)



(continued from previous page)

```

    ]
  }
}

```

### Step1 - Generate Signature

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".

We provide 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 list 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 is a combination of 3 different signature strategies

```

{"type": "characters-at", "config": {"pos": [0]}, "feature-idx": 1},
{"type": "characters-at", "config": {"pos": [0]}, "feature-idx": 2},
{"type": "feature-value", "feature_idx": 4}

```

It combines the initials of first and last name and postcode.

The second signature is generated by a combination of 2 signature strategies:

```

{"type": "metaphone", "feature-idx": 1},
{"type": "metaphone", "feature-idx": 2},

```

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.*

### Step2 - Filter Too Frequent Signatures

A signature is assumed to identify a record as uniquely as possible. Therefore, we need to filter out some too frequent signatures since they can uniquely identify the record. On the otherside, we want to be resilient to frequency attack, so we need to filter out too rare signature that only contains very few records. The configuration of filtering is in the `filter` part. For example, in the above configuration, the filter section is configured as:

```

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

```

Then we 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 that we also support absolute filtering configuration i.e. filter by number of counts. For example:

```

"filter": {
  "type": "count",
  "max": 100,
}

```

(continues on next page)

(continued from previous page)

```

    "min": 5,
}

```

### Step3 - Anonymization

Given we want to do privacy preserving record linkage, the signatures need to be hashed to avoid leaking of PII information. The most frequent used data structure of such encoding is Bloom Filter. Here we use one Bloom Filter and map all filtered signatures into that Bloom Filter. The configuration of Bloom Filter is in `block-filter` section:

```

"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.

### Carry out Blocking Job

Okay, once you have a good understanding of the P-Sig blocking, we can carry out our blocking job with `blocklib`. First, we need to process the data since `blocklib` only accept list of tuples or lists as input data. An example data input for `blocklib` is

```

[
    [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]
]

```

### Step1 - Generate Candidate Blocks for Party A - Alice

```

[3]: # NBVAL_IGNORE_OUTPUT
data_alice = df_alice.to_dict(orient='split')['data']
print("Example PII", data_alice[0])

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

```

```

[4]: # NBVAL_IGNORE_OUTPUT
from blocklib import generate_candidate_blocks

block_obj_alice = generate_candidate_blocks(data_alice, blocking_config)

Number of Blocks:    5011
Minimum Block Size:  1
Maximum Block Size:  61
Average Block Size:  1
Median Block Size:   1
Standard Deviation of Block Size:  3.8424497488253815

```

The statistics of blocks are printed for you to inspect the block distribution and decide if this is a good blocking result. Here both average and median block sizes are 1 which is resilient to frequency attack.

You can get the blocking instance and blocks/reversed indice in the `block_obj_alice`. Let's look at the first block in the reversed indices:

```
[5]: # NBVAL_IGNORE_OUTPUT
print(block_obj_alice.state)
list(block_obj_alice.blocks.keys())[0]

<blocklib.pprlpsig.PPRLIndexPSignature object at 0x11df2a3d0>

[5]: (1698, 1315, 932, 549)
```

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

### Step2 - Generate Candidate Blocks for Party B - Bob

```
[6]: # NBVAL_IGNORE_OUTPUT
df_bob = pd.read_csv('data/bob.csv')
data_bob = df_bob.to_dict(orient='split')['data']
block_obj_bob = generate_candidate_blocks(data_bob, blocking_config)
print(block_obj_bob.state)
print(list(block_obj_bob.blocks.keys())[0])
print(list(block_obj_bob.blocks.values())[1])

Number of Blocks:    5004
Minimum Block Size: 1
Maximum Block Size: 59
Average Block Size: 1
Median Block Size: 1
Standard Deviation of Block Size: 3.841316259098635
<blocklib.pprlpsig.PPRLIndexPSignature object at 0x11f533150>
(587, 571, 579, 563)
[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 signature, the algorithm will firstly map the list of signatures into a Bloom Filter for for each party called the candidate blocking filter, and then creates the combined blocking filter by only retaining the bits that are present in all candidate filters.

```
[7]: # NBVAL_IGNORE_OUTPUT
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: 5011 out of 5011 blocks are in common
Bob: 5001 out of 5001 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 we 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

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

subdata1 = [x[0] for x in data_alice]
subdata2 = [x[0] for x in data_bob]

rr, pc = assess_blocks_2party([filtered_blocks_alice, filtered_blocks_bob],
                             [subdata1, subdata2])

assessing blocks: 100%|| 2803/2803 [00:00<00:00, 82074.75key/s]
```

## Blocking Methods - LSH Based $\Lambda$ -fold Redundant

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

Let's see an example config of it:

```
[9]: 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
    }
}
```

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 -  $H^i$ ,  $i = 1, 2, \dots, \Lambda$  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

```
[10]: # NBVAL_IGNORE_OUTPUT
print('Generating candidate blocks for Alice:')
block_obj_alice = generate_candidate_blocks(data_alice, blocking_config)
```

(continues on next page)

(continued from previous page)

```
print()
print('Generating candidate blocks for Bob: ')
block_obj_bob = generate_candidate_blocks(data_bob, blocking_config)
```

```
Generating candidate blocks for Alice:
Number of Blocks: 6050
Minimum Block Size: 1
Maximum Block Size: 873
Average Block Size: 3
Median Block Size: 1
Standard Deviation of Block Size: 20.970313750521722
```

```
Generating candidate blocks for Bob:
Number of Blocks: 6085
Minimum Block Size: 1
Maximum Block Size: 862
Average Block Size: 3
Median Block Size: 1
Standard Deviation of Block Size: 20.71496408472215
```

```
[11]: # NBVAL_IGNORE_OUTPUT
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
```

```
[12]: # NBVAL_IGNORE_OUTPUT
rr, pc = assess_blocks_2party([filtered_blocks_alice, filtered_blocks_bob],
    [subdata1, subdata2])
print('RR={}'.format(rr))
print('PC={}'.format(pc))
```

```
assessing blocks: 100%| 4167/4167 [00:00<00:00, 7218.27key/s]
```

```
RR=0.8823915973988634
PC=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  $\lambda$ -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": [1, 2],
    "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   |
|----------------------|--|
| <i>“p-sig”</i>       | Probability Signature blocking method from <a href="#">Scalable Entity Resolution Using Probabilistic Signatures on Parallel Databases</a>           |
| <i>“lambda-fold”</i> | LSH based Lambda Fold Redundant blocking method from <a href="#">Scalable Entity Resolution Using Probabilistic Signatures on Parallel Databases</a> |

#### **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 u  |
| filter            | dictionary    | filtering threshold   |
| blocking-filter   | dictionary    | type of filter to generate blocks   |
| signatureSpecs    | list of lists | signature strategies where each list is a combination of signature strategies |

## Filter Configuration

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

## Blocking-filter Configuration

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

## SignatureSpecs Configurations

It is better to illustrate this one with an example:

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

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



## LSH based $\lambda$ -fold Configuration

| attribute         | type          | description   |
|-------------------|---------------|---|
| blocking-features | list[integer] | specify which features to used in blocks generation   |
| Lambda            | integer       | denotes the degree of redundancy - $H^i, i = 1, 2, \dots, \Lambda$ 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   |
| ran-dom_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 Development

### 1.3.1 Testing

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

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

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

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

### 1.3.2 Type Checking

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

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

## 1.4 Devops

### 1.4.1 Azure Pipeline

blocklib is automatically built and tested using Azure Pipeline in the project *blocklib* <<https://dev.azure.com/data61/blocklib>>

**The current pipeline is available:**

- Build pipeline <[https://dev.azure.com/data61/Anonlink/\\_build?definitionId=5](https://dev.azure.com/data61/Anonlink/_build?definitionId=5)>.

#### Build Pipeline

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

There is 1 top level stage in the build pipeline:

- *Unit tests and build* - tests the library using `pytest` with different versions of Python and package

The *Build & Test* job does:

- install the requirements,
- run tests on Ubuntu 18.04 OS, for Python 3.5, Python 3.6, Python 3.7 and Python 3.8
- publish the test results,
- publish the code coverage (on Azure only),
- package and publish the artifacts.

## 1.5 References

## CHAPTER 2

---

### 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