PUFFINN

Michael Vesterli and Martin Aumüller

Jan 17, 2020

CONTENTS:

1 Locality Sensitive Hashing 3

2 Algorithm Description 5

3 C++ Documentation 7

4 Python Documentation 15

Python Module Index 17

Index 19

Parameterless and Universal Fast FInding of Nearest Neighbors

PUFFINN is a library which uses Locality Sensitive Hashing (LSH) to find close neighbors in high-dimensional space. The specific approach used is described below and has a number of desirable properties, namely that it is easy to use and provides guarantees for the expected accuracy of the result. It is also very performant. Knowledge of LSH is not necessary to use the library when using the recommended default parameters in `Index`.

CHAPTER

ONE

LOCALITY SENSITIVE HASHING

A more thorough introduction to LSH should be found elsewhere, but a brief overview is presented here.

Locality Sensitive Hash functions are hash functions with the property that similar points have a higher probability of having a hash collision than distant points. These hash functions are grouped into families, from which multiple functions are drawn randomly. By concatenating multiple such functions, the collision probability of distant points becomes very low, while the collision probability of similar points remains reasonably high.

To use these functions to search for nearest neighbors, it is first necessary to construct an index which contains hashes of every point for multiple concatenated LSH functions. The nearest neighbors of a query point can then be found by hashing the query using the same hash functions. Any point with a colliding hash is then considered a candidate for being among the nearest neighbors. The most similar points among the candidates are then found by computing the actual similarity. In this way, most of the dataset does not need to be considered.

CHAPTER

TWO

ALGORITHM DESCRIPTION

PUFFINN uses an adaptive query mechanism which adjusts the length of the concatenated hashes depending on the similarity of the nearest neighbors and the index size. If the index is small or if the neighbors are distant, the collision probability needs to be increased, which is done by reducing the length of the hashes. This ensures that the target recall is achieved regardless of the difficulty of the query.

PUFFINN also uses a filtering step to further reduce the number of candidates. This is done by computing a number of sketches for each point using 1-bit LSH functions. Points are then only considered if the hamming similarity of a randomly selected pair of sketches is above a set treshold, which depend on the similarity. This significantly reduces the number of candidates, but reduces the recall slightly.

It is also possible to draw hash functions from different sources. By default, they are sampled independently, but it is also possible to use a precalculated set of functions and concatenate them in various ways. This reduces the number of necessary hash computations but yields hashes of lower quality. It is sometimes more efficient to use one of these alternative hash sources.

CHAPTER

THREE

C++ DOCUMENTATION

template<typename **TSim**, typename **THash** = **typename** *TSim*::DefaultHash, typename **TSketch** = **typename** *TSim*::DefaultSket
class Index : **private** puffinn::ChunkSerializable

An index constructed over a dataset which supports approximate near-neighbor queries for a specific similarity measure.

Basic usage consists of using the `insert`, `rebuild` and `search` methods in that order. These methods are the only ones in the library that need to be called during typical use.

The *`Index`* is generic over the similarity measure and two LSH families, one used for searching and one used for the following filtering step. The LSH families default to good choices for the similarity measure and should usually not be explicitly set.

Parameters

- `TSim`: The similarity measure. Currently *`CosineSimilarity`* and *`JaccardSimilarity`* are supported. Depending on the similarity measure, points are stored internally using different `Formats`. The `Format` specifies which types of input are supported.
- `THash`: The family of Locality-Sensitive hash functions used to search for near neighbor candidates. Defaults to a family chosen by the similarity measure.
- `TSketch`: The family of 1-bit Locality-Sensitive hash functions used to further filter candidates. Defaults to a family chosen by the similarity measure.

Public Functions

Index(**typename** TSim::Format::Args *dataset_args*, uint64_t *memory_limit*, **const** *HashSourceArgs*<THash> &*hash_args* = IndependentHashArgs<THash>(), **const** *HashSourceArgs*<TSketch> &*sketch_args* = IndependentHashArgs<TSketch>())

Construct an empty index.

Parameters

- `dataset_args`: Arguments specifying how the dataset should be stored, depending on the format of the similarity measure. When using *`CosineSimilarity`*, it specifies the dimension that all vectors must have. When using *`JaccardSimilarity`*, it specifies the universe size. All tokens must be integers between 0, inclusive, and the paramter, exclusive.
- `memory_limit`: The number of bytes of memory that the index is permitted to use. Using more memory almost always means that queries are more efficient.
- `hash_args`: Arguments used to construct the source from which hashes are drawn. This also includes arguments that are specific to the hash family specified in `THash`. It is recommended to use the default value.

- `sketch_args`: Similar to `hash_args`, but for the hash family specified in `TSketch`. It is recommended to use the default value.

Index(std::istream &*in*)
Deserialize an index.

It is assumed that the input data is a serialized index using the same version of PUFFINN.

void **deserialize_chunk**(std::istream &*in*)
Deserialize a single chunk.

void **serialize**(std::ostream &*out*, bool *use_chunks* = false) **const**
Serialize the index to the output stream to be loaded later. Supports splitting the serialized data into chunks, which can be accessed using the `serialize_chunks` method. This is primarily useful when the serialized data cannot be written to a file directly and storing the index twice in memory is infeasible.

Parameters

- `use_chunks`: Whether to split the serialized index into chunks. Defaults to false.

SerializeIter **serialize_chunks**() **const**
Get an iterator over serialized chunks in the dataset. See `serialize` for its use.

template<typename **T**>
void **insert**(**const** *T* &*value*)
Insert a value into the index.

Before the value can be found using the `search` method, `rebuild` must be called.

Parameters

- `value`: The value to insert. The type must be supported by the format used by `TSim`.

template<typename **T**>
T **get**(uint32_t *idx*)
Retrieve the n'th value inserted into the index.

Since the value is converted back from the internal storage format, it is unlikely to be equal to the inserted value due to normalization, rounding and other adjustments.

void **rebuild**()
Rebuild the index using the currently inserted points.

This is done in parallel by default. The number of threads used can be specified using the OMP_NUM_THREADS environment variable.

template<typename **T**>
std::vector<uint32_t> **search**(**const** *T* &*query*, unsigned int *k*, float *recall*, *FilterType* *filter_type* = *FilterType*::Default) **const**
Search for the approximate `k` nearest neighbors to a query.

Return The indices of the `k` nearest found neighbors. Indices are assigned incrementally to each point in the order they are inserted into the dataset, starting at 0. The result is ordered so that the most similar neighbor is first.

Parameters

- `query`: The query value. It follows the same constraints as when inserting a value.

- `k`: The number of neighbors to search for.
- `recall`: The expected recall of the result. Each of the nearest neighbors has at least this probability of being found in the first phase of the algorithm. However if sketching is used, the probability of the neighbor being returned might be slightly lower. This is given as a number between 0 and 1.
- `filter_type`: The approach used to filter candidates. Unless the expected recall needs to be strictly above the `recall` parameter, the default should be used.

std::vector<uint32_t> **search_from_index**(uint32_t *idx*, unsigned int *k*, float *recall*, *FilterType* *filter_type* = *FilterType*::Default) **const**

Search for the approximate `k` nearest neighbors to a value already inserted into the index.

This is similar to `search(get(idx))`, but avoids potential rounding errors from converting between data formats and automatically removes the query index from the search results .

template<typename **T**>
std::vector<unsigned int> **search_bf**(**const** *T* &*query*, unsigned int *k*) **const**

Search for the k nearest neighbors to a query by computing the similarity of each inserted value.

`rebuild` does not need to be called before a point is considered.

Return The indices of the `k` nearest neighbors. Indices are assigned incrementally to each point in the order they are inserted into the dataset, starting at 0. The result is ordered so that the most similar neighbor is first.

Parameters

- `query`: The query value. It follows the same constraints as when inserting a value.
- `k`: The number of neighbors to search for.

struct CosineSimilarity

Measures the cosine of the angle between two unit vectors.

This is also known as the angular distance. The supported LSH families are *CrossPolytopeHash*, *FHTCrossPolytopeHash* and *SimHash*.

Public Types

using Format = *UnitVectorFormat*

using DefaultHash = *FHTCrossPolytopeHash*

using DefaultSketch = *SimHash*

struct JaccardSimilarity

Measures the Jaccard Similarity between two sets.

This is defined as the size of the intersection divided by the size of the union. The supported LSH families are *MinHash* and *MinHash1Bit*.

Public Types

using Format = *SetFormat*

using DefaultSketch = *MinHash1Bit*

using DefaultHash = *MinHash*

struct UnitVectorFormat
A format for storing real vectors of unit length.

Currently, only `std::vector<float>` is supported as input type. The vectors do not need to be normalized before insertion.

Each number is stored using a 16-bit fixed point format. Although this slightly reduces the precision, the inaccuracies are very unlikely to have an impact on the result. The vectors are stored using 256-bit alignment.

struct SetFormat
A format for storing sets.

Currently, only `std::vector<uint32_t>` is supported as input type. Each integer in this set represents a token and must be between 0 and the number of dimensions specified when constructing the `LSHTable`.

class SimHash
A one-bit hash function, which creates a random hyperplane at the origin and hashes points depending on which side of the plane the point is located on.

Public Types

using Args = *SimHashArgs*

struct SimHashArgs
SimHash does not take any arguments.

class CrossPolytopeHash
An implementation of cross-polytope LSH using random rotations.

See *FHTCrossPolytopeHash* for a more efficient hash function using pseudo-random rotations instead.

This LSH applies a random rotation to vectors and then maps each vector to the closest axis. This yields multiple bits per hash function. Since there is no easy way to calculate collision probabilities, they are estimated via sampling instead.

Public Types

using Args = *CrossPolytopeArgs*

struct CrossPolytopeArgs
Arguments for the cross-polytope LSH.

Public Members

unsigned int **estimation_repetitions**
: Number of samples used to estimate collision probabilities.

float **estimation_eps**
: Granularity of collision probability estimation.

class FHTCrossPolytopeHash
: An implementation of cross-polytope LSH using fast-hadamard transforms.

  See *CrossPolytopeHash* for a description of the hash function.

  Using repeated applications of random +/-1 diagonal matrices and hadamard transforms, a pseudo-random rotation can be calculated more efficiently than using a random rotation. In practice, using three applications of each transform gives hashes of similar quality as the ones produced using a true random rotation.

Public Types

using Args = *FHTCrossPolytopeArgs*

struct FHTCrossPolytopeArgs
: Arguments for the fast-hadamard cross-polytope LSH.

Public Members

int **num_rotations**
: Number of iterations of the fast-hadamard transform.

unsigned int **estimation_repetitions**
: Number of samples used to estimate collision probabilities.

float **estimation_eps**
: Granularity of collision probability estimation.

class MinHash
: A multi-bit hash function for sets, which selects the minimum token when ordered by a random permutation.

  In practice, each token is hashed using a standard hash function, after which the token with the smallest hash is selected. The higher the Jaccard similarity, the higher the probability of two sets containing the same minimum token.

Public Types

using Args = *MinHashArgs*

struct MinHashArgs
: Arguments for *MinHash*.

class MinHash1Bit
: *MinHash*, but only use 1 bit to make it suitable for sketching.

Public Types

using Args = *MinHash*::*Args*

template<typename **T**>
struct HashSourceArgs

Arguments that can be supplied with data from the `LSHTable` to construct a HashSource.

Parameters

- `T`: The used LSH family.

Subclassed by *puffinn::HashPoolArgs< T >*, *puffinn::IndependentHashArgs< T >*, *puffinn::TensoredHashArgs< T >*

template<typename **T**>
struct IndependentHashArgs : **public** puffinn::*HashSourceArgs*<*T*>

Describes a hash source where all hash functions are sampled independently.

Public Members

T::Args **args**

Arguments for the hash family.

template<typename **T**>
struct HashPoolArgs : **public** puffinn::*HashSourceArgs*<*T*>

Describes a hash source which precomputes a pool of a given size.

Each hash is then constructed by sampling from this pool. This reduces the number of hashes that need to be computed, but produces hashes of lower quality.

It is typically possible to choose a pool size which performs better than independent hashing, but using independent hashes is a better default.

Public Functions

constexpr HashPoolArgs(unsigned int *pool_size*)

HashPoolArgs(std::istream &*in*)

Public Members

T::Args **args**

Arguments for the hash family.

unsigned int **pool_size**

The size of the pool in bits.

template<typename **T**>
struct TensoredHashArgs : **public** puffinn::*HashSourceArgs*<*T*>

Describes a hash source where hashes are constructed by combining a unique pair of smaller hashes from two sets.

This means that the number of necessary hashes is only the square root of the number used for independent hashing. However, this hash source does not perform well when targeting a high recall.

enum puffinn::**FilterType**

Approaches to filtering candidates.

Values:

Default

The most optimized and recommended approach, which stops shortly after the required expected recall has been achieved.

None

A simple approach without sketching. Use this if it is very important that the *expected* recall is above the given treshold. However it currently looks at every table in the internal structure before checking whether the recall target has been achieved.

Simple

A simple approach which mirrors `None`, but with filtering. It is only intended to be used to fairly assess the impact of sketching on the result.

CHAPTER

FOUR

PYTHON DOCUMENTATION

class puffinn.**Index**(*similarity_measure*, *dimensions*, *memory_limit*, *kwargs*)

An index constructed over a dataset which supports approximate near-neighbor queries for a specific similarity measure. It can be serialized using pickle.

Parameters

- **metric** (*str*) – The name of the metric used to measure the similarity of two points. Currently `"angular"` and `"jaccard"` are supported, which respectively map to `CosineSimilarity` and `JaccardSimilarity` in the C++ API.
- **dimensions** (*integer*) – The required number of dimensions of the input. When using the `"angular"` metric, all input vectors must have this length. When using the `"jaccard"` metric, all tokens in the input sets must be integers between 0, inclusive, and dimensions, exclusive.
- **memory_limit** (*integer*) – The number of bytes of memory that the index is permitted to use. Using more memory almost always means that queries are more efficient.
- **kwargs** – Additional arguments used to setup hash functions. None of these are necessary. The hash family, hash source and their arguments are given by specifying `"hash_function"`, `"hash_args"`, `"hash_source"` and `"source_args"` respectively.
- **kwargs.hash_function** – The hash function can be either `"simhash"`, `"crosspolytope"`, `"fht_crosspolytope"`, `"minhash"` or `"1bit_minhash"`, depending on the metric. See the C++ documentation on the corresponding types for details.
- **kwargs.hash_args** – Arguments for the used hash function. The supported arguments when using "crosspolytope" are "estimation_repetitions" and "estimation_eps". Using "fht_crosspolytope", "num_rotations" can also be specified. The other hash functions do not take any arguments. See the C++ documentation on the hash functions for details.
- **kwargs.hash_source** – The supported hash sources are `"independent"`, `"pool"` and `"tensor"`. See the C++ documentation on `HashSourceArgs` for details.
- **kwargs.source_args** – Arguments for the hash source. Most hash sources do not take arguments. If `"pool"` is selected, the size of the pool can be specified as the `"pool_size"`.

insert(*value*)

Insert a value into the index.

Before the value can be found using the search method, *rebuild()* must be called.

Parameters **value** (*list[integer]*) – The value to insert.

get(*idx*)

Retrieve a value that has been inserted into the index.

The value is converted back from the internal storage format, which means that it is unlikely to be equal to the inserted value due to normalization, rounding and other adjustments.

Parameters **idx** (*integer*) – The value to retrieve by insertion order.

rebuild()

Rebuild the index using the currently inserted points.

This is done in parallel.

search(*query*, *k*, *recall*, *filter_type = "default"*)

Search for the approximate k nearest neighbors to a query.

Parameters

- **query** (*list[integer]*) – The query value.
- **k** (*integer*) – The number of neighbors to search for.
- **recall** (*float*) – The expected recall of the result. Each of the nearest neighbors has at least this probability of being found in the first phase of the algorithm. However if sketching is used, the probability of the neighbor being returned might be slightly lower. This is given as a number between 0 and 1.
- **filter_type** (*string*) – The approach used to filter candidates. Unless the expected recall needs to be strictly above the recall parameter, the default should be used. The supported types are “default”, “none” and “simple”. See `FilterType` for more information.

PYTHON MODULE INDEX

p

puffinn, 15

INDEX

G

get() (*puffinn.Index method*), 15

I

Index (*class in puffinn*), 15
insert() (*puffinn.Index method*), 15

P

puffinn (*module*), 15
puffinn::CosineSimilarity (*C++ class*), 9
puffinn::CosineSimilarity::DefaultHash (*C++ type*), 9
puffinn::CosineSimilarity::DefaultSketch (*C++ type*), 9
puffinn::CosineSimilarity::Format (*C++ type*), 9
puffinn::CrossPolytopeArgs (*C++ class*), 10
puffinn::CrossPolytopeArgs::estimation_eps (*C++ member*), 11
puffinn::CrossPolytopeArgs::estimation_repetitions (*C++ member*), 11
puffinn::CrossPolytopeHash (*C++ class*), 10
puffinn::CrossPolytopeHash::Args (*C++ type*), 10
puffinn::Default (*C++ enumerator*), 13
puffinn::FHTCrossPolytopeArgs (*C++ class*), 11
puffinn::FHTCrossPolytopeArgs::estimation_eps (*C++ member*), 11
puffinn::FHTCrossPolytopeArgs::estimation_repetitions (*C++ member*), 11
puffinn::FHTCrossPolytopeArgs::num_rotations (*C++ member*), 11
puffinn::FHTCrossPolytopeHash (*C++ class*), 11
puffinn::FHTCrossPolytopeHash::Args (*C++ type*), 11
puffinn::FilterType (*C++ enum*), 12
puffinn::HashPoolArgs (*C++ class*), 12
puffinn::HashPoolArgs::args (*C++ member*), 12
puffinn::HashPoolArgs::HashPoolArgs (*C++ function*), 12
puffinn::HashPoolArgs::pool_size (*C++ member*), 12
puffinn::HashSourceArgs (*C++ class*), 12
puffinn::IndependentHashArgs (*C++ class*), 12
puffinn::IndependentHashArgs::args (*C++ member*), 12
puffinn::Index (*C++ class*), 7
puffinn::Index::deserialize_chunk (*C++ function*), 8
puffinn::Index::get (*C++ function*), 8
puffinn::Index::Index (*C++ function*), 7, 8
puffinn::Index::insert (*C++ function*), 8
puffinn::Index::rebuild (*C++ function*), 8
puffinn::Index::search (*C++ function*), 8
puffinn::Index::search_bf (*C++ function*), 9
puffinn::Index::search_from_index (*C++ function*), 9
puffinn::Index::serialize (*C++ function*), 8
puffinn::Index::serialize_chunks (*C++ function*), 8
puffinn::JaccardSimilarity (*C++ class*), 9
puffinn::JaccardSimilarity::DefaultHash (*C++ type*), 10
puffinn::JaccardSimilarity::DefaultSketch (*C++ type*), 10
puffinn::JaccardSimilarity::Format (*C++ type*), 10
puffinn::MinHash (*C++ class*), 11
puffinn::MinHash1Bit (*C++ class*), 11
puffinn::MinHash1Bit::Args (*C++ type*), 12
puffinn::MinHash::Args (*C++ type*), 11
puffinn::MinHashArgs (*C++ class*), 11
puffinn::None (*C++ enumerator*), 13
puffinn::SetFormat (*C++ class*), 10
puffinn::SimHash (*C++ class*), 10
puffinn::SimHash::Args (*C++ type*), 10
puffinn::SimHashArgs (*C++ class*), 10
puffinn::Simple (*C++ enumerator*), 13
puffinn::TensoredHashArgs (*C++ class*), 12

puffinn::UnitVectorFormat (*C++ class*), 10

R

rebuild() (*puffinn.Index method*), 16

S

search() (*puffinn.Index method*), 16