SMQTK-Indexing

Release 0.18.0

Kitware, Inc.

Nov 29, 2021

CONTENTS

GitHub

Python toolkit for pluggable algorithms and data structures for multimedia-based machine learning.

CHAPTER

ONE

INSTALLATION

Please reference the SMQTK-Core installation documentation as such documentation for this package is nearly identical. Of course, replace uses of *smqtk-core* with *smqtk-indexing*.

CHAPTER

TWO

ALGORITHM INTERFACES

Here we list and briefly describe the high level algorithm interfaces which this SMQTK-Indexing package provides. There is at least one implementation available for each interface. Some implementations will require additional dependencies.

2.1 NearestNeighborsIndex

This interface defines a method to build an index from a set of `smqtk_descriptors.DescriptorElement` instances (*NearestNeighborsIndex.build_index()*) and a nearest-neighbors query function for getting a number of near neighbors to a query `DescriptorElement` (*NearestNeighborsIndex.nn()*).

Building an index requires that some non-zero number of `DescriptorElement` instances be passed into the *NearestNeighborsIndex.build_index()* method. Subsequent calls to this method should rebuild the index model, not add to it. If an implementation supports persistent storage of the index, it should overwrite the configured index.

The *NearestNeighborsIndex.nn()* method uses a single `DescriptorElement` to query the current index for a specified number of nearest neighbors. Thus, the *NearestNeighborsIndex* instance must have a non-empty index loaded for this method to function. If the provided query `DescriptorElement` does not have a set vector, this method will also fail with an exception.

This interface additionally requires that implementations define a *NearestNeighborsIndex.count()* method, which returns the number of distinct `DescriptorElement` instances are in the index.

class smqtk_indexing.interfaces.nearest_neighbor_index.**NearestNeighborsIndex**(**args: Any*, ***kwargs: Any*)

Common interface for descriptor-based nearest-neighbor computation over a built index of descriptors.

Implementations, if they allow persistent storage of their index, should take the necessary parameters at construction time. Persistent storage content should be (over)written `build_index` is called.

Implementations should be thread safe and appropriately protect internal model components from concurrent access and modification.

build_index(*descriptors: Iterable[smqtk_descriptors.interfaces.descriptor_element.DescriptorElement]*) → None

Build the index with the given descriptor data elements.

Subsequent calls to this method should rebuild the current index. This method shall not add to the existing index nor raise an exception to as to protect the current index.

Raises `ValueError` – No data available in the given iterable.

Parameters `descriptors` (*collections.abc.Iterable[smqtk.representation.DescriptorElement]*) – Iterable of descriptor elements to build index over.

abstract count() → int

Returns Number of elements in this index.

nn(*d: smqtk_descriptors.interfaces.descriptor_element.DescriptorElement*, *n: int = 1*) → Tuple[Tuple[smqtk_descriptors.interfaces.descriptor_element.DescriptorElement, ...], Tuple[float, ...]]

Return the nearest *N* neighbors to the given descriptor element.

Raises

- **ValueError** – Input query descriptor d has no vector set.
- **ValueError** – Current index is empty.

Parameters

- **d** – Descriptor element to compute the neighbors of.
- **n** – Number of nearest neighbors to find.

Returns Tuple of nearest N DescriptorElement instances, and a tuple of the distance values to those neighbors.

remove_from_index(*uids: Iterable[collections.abc.Hashable]*) → None

Partially remove descriptors from this index associated with the given UIDs.

Parameters **uids** – Iterable of UIDs of descriptors to remove from this index.

Raises

- **ValueError** – No data available in the given iterable.
- **KeyError** – One or more UIDs provided do not match any stored descriptors. The index should not be modified.

update_index(*descriptors: Iterable[smqtk_descriptors.interfaces.descriptor_element.DescriptorElement]*) → None

Additively update the current index with the one or more descriptor elements given.

If no index exists yet, a new one should be created using the given descriptors.

Raises **ValueError** – No data available in the given iterable.

Parameters **descriptors** (*collections.abc.Iterable[smqtk.representation.DescriptorElement]*) – Iterable of descriptor elements to add to this index.

2.2 LshFunctor

Implementations of this interface define the generation of a locality-sensitive hash code for a given DescriptorElement. These are used in smqtk_indexing.impls.nn_index.lsh.LSHNearestNeighborIndex instances.

class smqtk_indexing.interfaces.lsh_functor.**LshFunctor**(**args: Any*, ***kwargs: Any*)

Locality-sensitive hashing functor interface.

The aim of such a function is to be able to generate hash codes (bit-vectors) such that similar items map to the same or similar hashes with a high probability. In other words, it aims to maximize hash collision for similar items.

Building Models

Some hash functions want to build a model based on some training set of descriptors. Due to the non-standard nature of algorithm training and model building, please refer to the specific implementation for further information on whether model training is needed and how it is accomplished.

abstract get_hash(*descriptor: numpy.ndarray*) → numpy.ndarray
Get the locality-sensitive hash code for the input descriptor.

Parameters **descriptor** – Descriptor vector we should generate the hash of.

Returns Generated bit-vector as a numpy array of booleans.

2.3 HashIndex

This interface describes specialized *NearestNeighborsIndex* implementations designed to index hash codes (bit vectors) via the hamming distance metric function. Implementations of this interface are primarily used with the `smqtk_indexing.impls.nn_index.lsh.LSHNearestNeighborIndex` implementation.

Unlike the *NearestNeighborsIndex* interface from which this interface is very similar to, *HashIndex* instances are built with an iterable of `numpy.ndarray` and *HashIndex.nn()* returns a `numpy.ndarray`.

class smqtk_indexing.interfaces.hash_index.**HashIndex**(**args: Any*, ***kwargs: Any*)
Specialized `NearestNeighborsIndex` for indexing unique hash codes bit-vectors) in memory (numpy arrays) using the hamming distance metric.

Implementations of this interface cannot be used in place of something requiring a `NearestNeighborsIndex` implementation due to the speciality of this interface.

Only unique bit vectors should be indexed. The `nn` method should not return the same bit vector more than once for any query.

build_index(*hashes: Iterable[numpy.ndarray]*) → None
Build the index with the given hash codes (bit-vectors).

Subsequent calls to this method should rebuild the current index. This method shall not add to the existing index nor raise an exception to as to protect the current index.

Raises **ValueError** – No data available in the given iterable.

Parameters **hashes** – Iterable of hash vectors (boolean-valued) to build index over.

abstract count() → int

Returns Number of elements in this index.

nn(*h: numpy.ndarray*, *n: int = 1*) → Tuple[numpy.ndarray, Sequence[float]]
Return the nearest *N* neighbor hash codes as bit-vectors to the given hash code bit-vector.

Distances are in the range [0,1] and are the percent different each neighbor hash is from the query, based on the number of bits contained in the query (normalized hamming distance).

Raises **ValueError** – Current index is empty.

Parameters

- **h** – Hash code vectors (boolean-valued) to compute the neighbors of. Should be the same bit length as indexed hash codes.
- **n** – Number of nearest neighbors to find.

Returns Tuple of nearest N hash codes and a tuple of the distance values to those neighbors.

remove_from_index(*hashes: Iterable[numpy.ndarray]*) → None
Partially remove hashes from this index.

Parameters **hashes** – Iterable of numpy boolean hash vectors to remove from this index.

Raises

- **ValueError** – No data available in the given iterable.
- **KeyError** – One or more UIDs provided do not match any stored descriptors.

update_index(*hashes: Iterable[numpy.ndarray]*) → None
Additively update the current index with the one or more hash vectors given.

If no index exists yet, a new one should be created using the given hash vectors.

Raises **ValueError** – No data available in the given iterable.

Parameters **hashes** – Iterable of numpy boolean hash vectors to add to this index.

CHAPTER

THREE

EXAMPLES

3.1 Nearest Neighbor Computation with Caffe

The following is a concrete example of performing a nearest neighbor computation using a set of ten butterfly images. This example has been tested using Caffe version rc2,) and may work with the master version of Caffe from GitHub.

To generate the required model files `image_mean_filepath` and `network_model_filepath`, run the following scripts:

```
caffe_src/ilsvrc12/get_ilsvrc_aux.sh
caffe_src/scripts/download_model_binary.py  ./models/bvlc_reference_caffenet/
```

Once this is done, the nearest neighbor index for the butterfly images can be built with the following code:

```
from smqtk.algorithms.nn_index.flann import FlannNearestNeighborsIndex

# Import some butterfly data
urls = ["http://www.comp.leeds.ac.uk/scs6jwks/dataset/leedsbutterfly/examples/{:03d}.jpg
↪".format(i) for i in range(1,11)]
from smqtk.representation.data_element.url_element import DataUrlElement
el = [DataUrlElement(d) for d in urls]

# Create a model.  This assumes that you have properly set up a proper Caffe environment␣
↪for SMQTK
from smqtk.algorithms.descriptor_generator import get_descriptor_generator_impls
cd = get_descriptor_generator_impls()['CaffeDescriptorGenerator'](
        network_prototxt_filepath="caffe/models/bvlc_reference_caffenet/deploy.prototxt",
        network_model_filepath="caffe/models/bvlc_reference_caffenet/bvlc_reference_
↪caffenet.caffemodel",
        image_mean_filepath="caffe/data/ilsvrc12/imagenet_mean.binaryproto",
        return_layer="fc7",
        batch_size=1,
        use_gpu=False,
        gpu_device_id=0,
        network_is_bgr=True,
        data_layer="data",
        load_truncated_images=True)

# Set up a factory for our vector (here in-memory storage)
from smqtk.representation.descriptor_element_factory import DescriptorElementFactory
from smqtk.representation.descriptor_element.local_elements import␣
↪DescriptorMemoryElement
```

(continues on next page)

(continued from previous page)

```
factory = DescriptorElementFactory(DescriptorMemoryElement, {})

# Compute features on the first image
descriptor_iter = cd.generate_elements(el, descr_factory=factory)
index = FlannNearestNeighborsIndex(distance_method="euclidean",
                                   random_seed=42, index_filepath="nn.index",
                                   parameters_filepath="nn.params",
                                   descriptor_cache_filepath="nn.cache")
index.build_index(descriptor_iter)
```

3.2 NearestNeighborServiceServer Incremental Update Example

3.2.1 Goal and Plan

In this example, we will show how to initially set up an instance of the `NearestNeighborServiceServer` web API service class such that it can handle incremental updates to its background data. We will also show how to perform incremental updates and confirm that the service recognizes this new data.

For this example, we will use the `LSHNearestNeighborIndex` implementation as it is one that currently supports live-reloading its component model files. Along with it, we will use the `ItqFunctor` and `PostgresDescriptorSet` implementations as the components of the `LSHNearestNeighborIndex`. For simplicity, we will not use a specific *HashIndex*, which causes a `LinearHashIndex` to be constructed and used at query time.

All scripts used in this example's proceedure have a command line interface that uses dash options. Their available options can be listed by giving the `-h`/`--help` option. Additional debug logging can be seen output by providing a `-d` or `-v` option, depending on the script.

This example assumes that you have a basic understanding of:

- JSON for configuring files
- how to use the `bin/runApplication.py`
- **SMQTK's NearestNeighborServiceServer application and algorithmic/data-structure components.**
  - *NearestNeighborsIndex*, specific the implementation `LSHNearestNeighborIndex`
  - `DescriptorSet` abstract and implementations with an updatable persistance storage mechanism (e.g. `PostgresDescriptorSet`).
  - *LshFunctor* abstract and implementations

Dependencies

Due to our use of the `PostgresDescriptorSet` in this example, a minimum installed version of PostgreSQL 9.4 is required, as is the `psycopg2` python module (`conda` and `pip` installable). Please check and modify the configuration files for this example to be able to connect to the database of your choosing.

Take a look at the `etc/smqtk/postgres/descriptor_element/example_table_init.sql` and `etc/smqtk/postgres/descriptor_set/example_table_init.sql` files, located from the root of the source tree, for table creation examples for element and index storage:

```
$ psql postgres -f etc/smqtk/postgres/descriptor_element/example_table_init.sql
$ psql postgres -f etc/smqtk/postgres/descriptor_set/example_table_init.sql
```

3.2.2 Proceedure

[1] Getting and Splitting the data set

For this example we will use the Leeds butterfly data set (see the `download_leeds_butterfly.sh` script). We will split the data set into an initial sub-set composed of about half of the images from each butterfly catagory (418 total images in the `2.ingest_files_1.txt` file). We will then split the data into a two more sub-sets each composed of about half of the remaining data (each composing about 1/4 of the original data set, totaling 209 and 205 images each in the `TODO.ingest_files_2.txt` and `TODO.ingest_files_3.txt` files respectively).

[2] Computing Initial Ingest

For this example, an "ingest" consists of a set of descriptors in an index and a mapping of hash codes to the descriptors.

In this example, we also train the LSH hash code functor's model, if it needs one, based on the descriptors computed before computing the hash codes. We are using the ITQ functor which does require a model. It may be the case that the functor of choice does not require a model, or a sufficient model for the functor is already available for use, in which case that step may be skipped.

Our example's initial ingest will use the image files listed in the `2.ingest_files_1.txt` test file.

[2a] Computing Descriptors

We will use the script `bin/scripts/compute_many_descriptors.py` for computing descriptors from a list of file paths. This script will be used again in later sections for additional incremental ingests.

The example configuration file for this script, `2a.config.compute_many_descriptors.json` (shown below), should be modified to connect to the appropriate PostgreSQL database and the correct Caffe model files for your system. For this example, we will be using Caffe's `bvlc_alexnet` network model with the `ilsvrc12` image mean be used for this example.

```
{
    "descriptor_factory": {
        "smqtk.representation.descriptor_element.postgres.PostgresDescriptorElement": {
            "binary_col": "vector",
            "db_host": "/dev/shm",
            "db_name": "postgres",
            "db_pass": null,
            "db_port": null,
            "db_user": null,
            "table_name": "descriptors",
            "type_col": "type_str",
            "uuid_col": "uid"
        },
        "type": "smqtk.representation.descriptor_element.postgres.
↪PostgresDescriptorElement"
    },
    "descriptor_generator": {
        "smqtk.algorithms.descriptor_generator.caffe_descriptor.CaffeDescriptorGenerator
↪": {
            "network_model": {
                "type": "smqtk.representation.data_element.file_element.DataFileElement",
                "smqtk.representation.data_element.file_element.DataFileElement": {
```

(continues on next page)

(continued from previous page)

```
                    "filepath": "/home/purg/dev/caffe/source/models/bvlc_alexnet/bvlc_alexnet.caffemodel",
                    "readonly":true
                }
            },
            "network_prototxt": {
                "type": "smqtk.representation.data_element.file_element.DataFileElement",
                "smqtk.representation.data_element.file_element.DataFileElement": {
                    "filepath": "/home/purg/dev/caffe/source/models/bvlc_alexnet/deploy.prototxt",
                    "readonly": true
                }
            },
            "image_mean": {
                "type": "smqtk.representation.data_element.file_element.DataFileElement",
                "smqtk.representation.data_element.file_element.DataFileElement": {
                    "filepath": "/home/purg/dev/caffe/source/data/ilsvrc12/imagenet_mean.binaryproto",
                    "readonly": true
                }
            },
            "return_layer": "fc7",
            "batch_size": 256,
            "use_gpu": false,
            "gpu_device_id": 0,
            "network_is_bgr": true,
            "data_layer": "data",
            "load_truncated_images": false,
            "pixel_rescale": null,
            "input_scale": null,
            "threads": null
        },
        "type": "smqtk.algorithms.descriptor_generator.caffe_descriptor.CaffeDescriptorGenerator"
    },
    "descriptor_set": {
        "smqtk.representation.descriptor_set.postgres.PostgresDescriptorSet": {
            "db_host": "/dev/shm",
            "db_name": "postgres",
            "db_pass": null,
            "db_port": null,
            "db_user": null,
            "element_col": "element",
            "multiquery_batch_size": 1000,
            "pickle_protocol": -1,
            "read_only": false,
            "table_name": "descriptor_set",
            "uuid_col": "uid"
        },
        "type": "smqtk.representation.descriptor_set.postgres.PostgresDescriptorSet"
    }
}
```

For running the script, take a look at the example invocation in the file `2a.run.sh`:

```
#!/usr/bin/env bash
set -e
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
cd "${SCRIPT_DIR}"

../../../bin/scripts/compute_many_descriptors.py \
  -v \
  -c 2a.config.compute_many_descriptors.json \
  -f 2.ingest_files_1.txt \
  --completed-files 2a.completed_files.csv
```

This step yields two side effects:

- Descriptors computed are saved in the configured implementation's persistant storage (a postgres database in our case)
- **A file is generated that mapping input files to their `DataElement` UUID values, or otherwise known as their SHA1 checksu**
  - This file will be used later as a convenient way of getting at the UUIDs of descriptors processed for a particular ingest.
  - **Other uses of this file for other tasks may include:**
    - interfacing with other systems that use file paths as the primary identifier of base data files
    - want to quickly back-reference the original file for a given UUID, as UUIDs for descriptor and classification elements are currently the same as the original file they are computed from.

[2b] Training ITQ Model

To train the ITQ model, we will use the script: `./bin/scripts/train_itq.py`. We'll want to train the functor's model using the descriptors computed in *step 2a*. Since we will be using the whole index (418 descriptors), we will not need to provide the script with an additional list of UUIDs.

The example configuration file for this script, `2b.config.train_itq.json`, should be modified to connect to the appropriate PostgreSQL database.

```
{
    "descriptor_set": {
        "smqtk.representation.descriptor_set.postgres.PostgresDescriptorSet": {
            "db_host": "/dev/shm",
            "db_name": "postgres",
            "db_pass": null,
            "db_port": null,
            "db_user": null,
            "element_col": "element",
            "multiquery_batch_size": 1000,
            "pickle_protocol": -1,
            "read_only": false,
            "table_name": "descriptor_set",
            "uuid_col": "uid"
        },
        "type": "smqtk.representation.descriptor_set.postgres.PostgresDescriptorSet"
```

(continues on next page)

(continued from previous page)

```
    },
    "itq_config": {
        "bit_length": 256,
        "itq_iterations": 50,
        "mean_vec_filepath": "2b.itq.256bit.mean_vec.npy",
        "random_seed": 0,
        "rotation_filepath": "2b.itq.256bit.rotation.npy"
    },
    "parallel": {
        "index_load_cores": 4,
        "use_multiprocessing": true
    },
    "uuids_list_filepath": null
}
```

`2b.run.sh` contains an example call of the training script:

```
#!/usr/bin/env bash
set -e
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
cd "${SCRIPT_DIR}"

../../../bin/scripts/train_itq.py -v -c 2b.config.train_itq.json
```

This step produces the following side effects:

- **Writes the two file components of the model as configured.**
  - **We configured the output files:**
    - `2b.itq.256bit.mean_vec.npy`
    - `2b.nnss.itq.256bit.rotation.npy`

[2c] Computing Hash Codes

For this step we will be using the script `bin/scripts/compute_hash_codes.py` to compute ITQ hash codes for the currently computed descriptors. We will be using the descriptor index we added to before as well as the `ItqFunctor` models we trained in the previous step.

This script additionally wants to know the UUIDs of the descriptors to compute hash codes for. We can use the `2a.completed_files.csv` file computed earlier in *step 2a* to get at the UUIDs (SHA1 checksum) values for the computed files. Remember, as is documented in the `DescriptorGenerator` interface, descriptor UUIDs are the same as the UUID of the data from which it was generated from, thus we can use this file.

We can conveniently extract these UUIDs with the following commands in script `2c.extract_ingest_uuids.sh`, resulting in the file `2c.uuids_for_processing.txt`:

```
#!/usr/bin/env bash
set -e
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
cd "${SCRIPT_DIR}"

cat 2a.completed_files.csv | cut -d',' -f2 >2c.uuids_for_processing.txt
```

With this file, we can now complete the configuration for our computation script:

```
{
    "plugins": {
        "descriptor_set": {
            "smqtk.representation.descriptor_set.postgres.PostgresDescriptorSet": {
                "db_host": "/dev/shm",
                "db_name": "postgres",
                "db_pass": null,
                "db_port": null,
                "db_user": null,
                "element_col": "element",
                "multiquery_batch_size": 1000,
                "pickle_protocol": -1,
                "read_only": false,
                "table_name": "descriptor_set",
                "uuid_col": "uid"
            },
            "type": "smqtk.representation.descriptor_set.postgres.PostgresDescriptorSet"
        },
        "lsh_functor": {
            "smqtk.algorithms.nn_index.lsh.functors.itq.ItqFunctor": {
                "mean_vec_cache": {
                    "type": "smqtk.representation.data_element.file_element.
↪DataFileElement",
                    "smqtk.representation.data_element.file_element.DataFileElement": {
                        "filepath": "2b.itq.256bit.mean_vec.npy",
                        "readonly":true
                    }
                },
                "rotation_cache": {
                "type": "smqtk.representation.data_element.file_element.DataFileElement",
                    "smqtk.representation.data_element.file_element.DataFileElement": {
                        "filepath": "2b.itq.256bit.rotation.npy",
                        "readonly":true
                    }
                },
                "bit_length": 256,
                "itq_iterations": 50,
                "normalize": null,
                "random_seed": 0
            },
            "type": "smqtk.algorithms.nn_index.lsh.functors.itq.ItqFunctor"
        }
    },
    "utility": {
        "hash2uuids_input_filepath": null,
        "hash2uuids_output_filepath": "2c.hash2uuids.pickle",
        "pickle_protocol": -1,
        "report_interval": 1.0,
        "use_multiprocessing": true,
        "uuid_list_filepath": "2c.uuids_for_processing.txt"
    }
}
```

We are not setting a value for `hash2uuids_input_filepath` because this is the first time we are running this script, thus we do not have an existing structure to add to.

We can now move forward and run the computation script:

```
#!/usr/bin/env bash
set -e
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
cd "${SCRIPT_DIR}"

../../../bin/scripts/compute_hash_codes.py -v -c 2c.config.compute_hash_codes.json
```

This step produces the following side effects:

- **Writed the file `2c.hash2uuids.pickle`**
  - This file will be copied and used in configuring the `LSHNearestNeighborIndex` for the `NearestNeighborServiceServer`

[2d] Starting the `NearestNeighborServiceServer`

Normally, a *NearestNeighborsIndex* instance would need to be have its index built before it can be used. However, we have effectively already done this in the preceeding steps, so are instead able to get right to configuring and starting the `NearestNeighborServiceServer`. A default configuration may be generated using the generic `bin/runApplication.py` script (since web applications/servers are plugins) using the command:

```
$ runApplication.py -a NearestNeighborServiceServer -g 2d.config.nnss_app.json
```

An example configuration has been provided in `2d.config.nnss_app.json`. The `DescriptorSet`, `DescriptorGenerator` and *LshFunctor* configuration sections should be the same as used in the preceeding sections.

Before configuring, we are copying `2c.hash2uuids.pickle` to `2d.hash2uuids.pickle`. Since we will be overwriting this file (the `2d` version) in steps to come, we want to separate it from the results of *step 2c*.

Note the highlighted lines for configurations of note for the `LSHNearestNeighborIndex` implementation. These will be explained below.

```
{
    "descriptor_factory": {
        "smqtk.representation.descriptor_element.postgres.PostgresDescriptorElement": {
            "binary_col": "vector",
            "db_host": "/dev/shm",
            "db_name": "postgres",
            "db_pass": null,
            "db_port": null,
            "db_user": null,
            "table_name": "descriptors",
            "type_col": "type_str",
            "uuid_col": "uid"
        },
        "type": "smqtk.representation.descriptor_element.postgres.
↪PostgresDescriptorElement"
    },
    "descriptor_generator": {
```

(continues on next page)

(continued from previous page)

```
        "smqtk.algorithms.descriptor_generator.caffe_descriptor.CaffeDescriptorGenerator
↪": {
            "network_model": {
                "type": "smqtk.representation.data_element.file_element.DataFileElement",
                "smqtk.representation.data_element.file_element.DataFileElement": {
                    "filepath": "/home/purg/dev/caffe/source/models/bvlc_alexnet/bvlc_
↪alexnet.caffemodel",
                    "readonly":true
                }
            },
            "network_prototxt": {
                "type": "smqtk.representation.data_element.file_element.DataFileElement",
                "smqtk.representation.data_element.file_element.DataFileElement": {
                    "filepath": "/home/purg/dev/caffe/source/models/bvlc_alexnet/deploy.
↪prototxt",
                    "readonly": true
                }
            },
            "image_mean": {
                "type": "smqtk.representation.data_element.file_element.DataFileElement",
                "smqtk.representation.data_element.file_element.DataFileElement": {
                    "filepath": "/home/purg/dev/caffe/source/data/ilsvrc12/imagenet_mean.
↪binaryproto",
                    "readonly": true
                }
            },
            "return_layer": "fc7",
            "batch_size": 256,
            "use_gpu": false,
            "gpu_device_id": 0,
            "network_is_bgr": true,
            "data_layer": "data",
            "load_truncated_images": false,
            "pixel_rescale": null,
            "input_scale": null,
            "threads": null
        },
        "type": "smqtk.algorithms.descriptor_generator.caffe_descriptor.
↪CaffeDescriptorGenerator"
    },
    "flask_app": {
        "BASIC_AUTH_PASSWORD": "demo",
        "BASIC_AUTH_USERNAME": "demo",
        "SECRET_KEY": "MySuperUltraSecret"
    },
    "nn_index": {
        "smqtk.algorithms.nn_index.lsh.LSHNearestNeighborIndex": {
            "lsh_functor": {
                "type": "smqtk.algorithms.nn_index.lsh.functors.itq.ItqFunctor",
                "smqtk.algorithms.nn_index.lsh.functors.itq.ItqFunctor": {
                    "mean_vec_cache": {
                        "type": "smqtk.representation.data_element.file_element.
↪DataFileElement",
```

(continues on next page)

(continued from previous page)

```
                        "smqtk.representation.data_element.file_element.DataFileElement
↪": {
                            "filepath": "2b.itq.256bit.mean_vec.npy",
                            "readonly":true
                        }
                    },
                    "rotation_cache": {
                    "type": "smqtk.representation.data_element.file_element.
↪DataFileElement",
                        "smqtk.representation.data_element.file_element.DataFileElement
↪": {
                            "filepath": "2b.itq.256bit.rotation.npy",
                            "readonly":true
                        }
                    },
                    "bit_length": 256,
                    "itq_iterations": 50,
                    "normalize": null,
                    "random_seed": 0
                }
            },
            "descriptor_set": {
                "smqtk.representation.descriptor_set.postgres.PostgresDescriptorSet": {
                    "db_host": "/dev/shm",
                    "db_name": "postgres",
                    "db_pass": null,
                    "db_port": null,
                    "db_user": null,
                    "element_col": "element",
                    "multiquery_batch_size": 1000,
                    "pickle_protocol": -1,
                    "read_only": false,
                    "table_name": "descriptor_set",
                    "uuid_col": "uid"
                },
                "type": "smqtk.representation.descriptor_set.postgres.
↪PostgresDescriptorSet"
            },
            "hash2uuids_kvstore": {
                "type": "smqtk.representation.key_value.memory.MemoryKeyValueStore",
                "smqtk.representation.key_value.memory.MemoryKeyValueStore":  {
                    "cache_element": {
                        "type": "smqtk.representation.data_element.file_element.
↪DataFileElement",
                        "smqtk.representation.data_element.file_element.DataFileElement
↪": {
                            "filepath": "2d.hash2uuids.pickle",
                            "readonly": true
                        }
                    }
                }
            },
```

(continues on next page)

(continued from previous page)

```
110             "hash_index_comment": "'hash_index' may also be null to default to a linear␣
↪index built at query time.",
111             "hash_index": {"type": null},
112             "distance_method": "hik",
113             "read_only": true
114         },
115         "type": "LSHNearestNeighborIndex"
116     },
117     "server": {
118         "host": "127.0.0.1",
119         "port": 5000
120     }
121 }
```

Emphasized line explanations:

- On line 55, we are using the `hik` distance method, or histogram intersection distance, as it has been experimentally shown to out perform other distance metrics for AlexNet descriptors.
- On line 56, we are using the output generated during *step 2c*. This file will be updated during incremental updates, along with the configured `DescriptorSet`.
- On line 58, we are choosing not to use a pre-computed *HashIndex*. This means that a `LinearHashIndex` will be created and used at query time. Other implementations in the future may incorporate live-reload functionality.
- **On line 61, we are telling the `LSHNearestNeighborIndex` to reload its implementation-specific model files when it detects**
  - We listed `LSHNearestNeighborIndex` implementation's only model file on line 56 and will be updated via the `bin/scripts/compute_hash_codes.py`
- On line 72, we are telling the implementation to make sure it does not write to any of its resources.

We can now start the service using:

```
$ runApplication.py -a NearestNeighborServiceServer -c 2d.config.nnss_app.json
```

We can test the server by calling its web api via curl using one of our ingested images, `leedsbutterfly/images/001_0001.jpg`:

```
$ curl http://127.0.0.1:5000/nn/n=10/file:///home/purg/data/smqtk/leedsbutterfly/images/
↪001_0001.jpg
{
  "distances": [
    -2440.0882132202387,
    -1900.5749250203371,
    -1825.7734497860074,
    -1771.708476960659,
    -1753.6621350347996,
    -1729.6928340941668,
    -1684.2977819740772,
    -1627.438737615943,
    -1608.4607088603079,
    -1536.5930510759354
  ],
  "message": "execution nominal",
```

(continues on next page)

(continued from previous page)


```
  "neighbors": [
    "84f62ef716fb73586231016ec64cfeed82305bba",
    "ad4af38cf36467f46a3d698c1720f927ff729ed7",
    "2dffc1798596bc8be7f0af8629208c28606bba65",
    "8f5b4541f1993a7c69892844e568642247e4acf2",
    "e1e5f3e21d8e3312a4c59371f3ad8c49a619bbca",
    "e8627a1a3a5a55727fe76848ba980c989bcef103",
    "750e88705efeee2f12193b45fb34ec10565699f9",
    "e21b695a99fee6ff5af8d2b86d4c3e8fe3295575",
    "0af474b31fc8002fa9b9a2324617227069649f43",
    "7da0501f7d6322aef0323c34002d37a986a3bf74"
  ],
  "reference_uri": "file:///home/purg/data/smqtk/leedsbutterfly/images/001_0001.jpg",
  "success": true
}
```


If we compare the result neighbor UUIDs to the SHA1 hash signatures of the original files (that descritpors were computed from), listed in the *step 2a* result file `2a.completed_files.csv`, we find that the above results are all of the class `001`, or monarch butterflies.

If we used either of the files `leedsbutterfly/images/001_0042.jpg` or `leedsbutterfly/images/001_0063.jpg`, which are not in our initial ingest, but in the subsequent ingests, and set `.../n=832/...` (the maximum size we will see in ingest grow to), we would see that the API does not return their UUIDs since they have not been ingested yet. We will also see that only 418 neighbors are returned even though we asked for 832, since there are only 418 elements currently in the index. We will use these three files as proof that we are actually expanding the searchable content after each incremental ingest.

We provide a helper bash script, `test_in_index.sh`, for checking if a file is findable via in the search API. A call of the form:

```
$ ./test_in_index.sh leedsbutterfly/images/001_0001.jpg 832
```

... performs a curl call to the server's default host address and port for the 832 nearest neighbors to the query image file, and checks if the UUIDs of the given file (the sha1sum) is in the returned list of UUIDs.

[3] First Incremental Update

Now that we have a live `NearestNeighborServiceServer` instance running, we can incrementally process the files listed in `3.ingest_files_2.txt`, making them available for search without having to shut down or otherwise do anything to the running server instance.

We will be performing the same actions taken in steps *2a* and *2c*, but with different inputs and outputs:

1. Compute descriptors for files listed in `3.ingest_files_2.txt` using script `compute_many_descriptors.py`, outputting file `3.completed_files.csv`.
2. Create a list of descriptor UUIDs just computed (see `2c.extract_ingest_uuids.sh`) and compute hash codes for those descriptors, overwriting `2d.hash2uuids.pickle` (which causes the server the `LSHNearestNeighborIndex` instance to update itself).

The following is the updated configuration file for hash code generation. Note the highlighted lines for differences from *step 2c* (notes to follow):

```
{
    "plugins": {
```

(continues on next page)

(continued from previous page)

```
        "descriptor_set": {
            "smqtk.representation.descriptor_set.postgres.PostgresDescriptorSet": {
                "db_host": "/dev/shm",
                "db_name": "postgres",
                "db_pass": null,
                "db_port": null,
                "db_user": null,
                "element_col": "element",
                "multiquery_batch_size": 1000,
                "pickle_protocol": -1,
                "read_only": false,
                "table_name": "descriptor_set",
                "uuid_col": "uid"
            },
            "type": "smqtk.representation.descriptor_set.postgres.PostgresDescriptorSet"
        },
        "lsh_functor": {
            "smqtk.algorithms.nn_index.lsh.functors.itq.ItqFunctor": {
                "mean_vec_cache": {
                    "type": "smqtk.representation.data_element.file_element.
↪DataFileElement",
                    "smqtk.representation.data_element.file_element.DataFileElement": {
                        "filepath": "2b.itq.256bit.mean_vec.npy",
                        "readonly":true
                    }
                },
                "rotation_cache": {
                "type": "smqtk.representation.data_element.file_element.DataFileElement",
                    "smqtk.representation.data_element.file_element.DataFileElement": {
                        "filepath": "2b.itq.256bit.rotation.npy",
                        "readonly":true
                    }
                },
                "bit_length": 256,
                "itq_iterations": 50,
                "normalize": null,
                "random_seed": 0
            },
            "type": "smqtk.algorithms.nn_index.lsh.functors.itq.ItqFunctor"
        }
    },
    "utility": {
        "hash2uuids_input_filepath": "2d.hash2uuids.pickle",
        "hash2uuids_output_filepath": "2d.hash2uuids.pickle",
        "pickle_protocol": -1,
        "report_interval": 1.0,
        "use_multiprocessing": true,
        "uuid_list_filepath": "3.uuids_for_processing.txt"
    }
}
```

Line notes:

- Lines 31 and 32 are set to the model file that the `LSHNearestNeighborIndex` implementation for the server was configured to use.
- Line 36 should be set to the descriptor UUIDs file generated from `3.completed_files.csv` (see `2c.extract_ingest_uuids.sh`)

The provided `3.run.sh` script is an example of the commands to run for updating the indices and models:

```
#!/usr/bin/env bash
set -e
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
cd "${SCRIPT_DIR}"

# Compute descriptors for new files, outputing a file that matches input
# files to thair SHA1 checksum values (their UUIDs)
../../../bin/scripts/compute_many_descriptors.py \
  -d \
  -c 2a.config.compute_many_descriptors.json \
  -f 3.ingest_files_2.txt \
  --completed-files 3.completed_files.csv

# Extract UUIDs of files/descriptors just generated
cat 3.completed_files.csv | cut -d, -f2 > 3.uuids_for_processing.txt

# Compute hash codes for descriptors just generated, updating the target
# hash2uuids model file.
../../../bin/scripts/compute_hash_codes.py -v -c 3.config.compute_hash_codes.json
```

After calling the `compute_hash_codes.py` script, the server logging should yield messages (if run in debug/verbose mode) showing that the `LSHNearestNeighborIndex` updated its model.

We can now test that the `NearestNeighborServiceServer` using the query examples used at the end of *step 2d*. Using images `leedsbutterfly/images/001_0001.jpg` and `leedsbutterfly/images/001_0042.jpg` as our query examples (and `.../n=832/...`), we can see that both are in the index (each image is the nearest neighbor to itself). We also see that a total of 627 neighbors are returned, which is the current number of elements now in the index after this update. The sha1 of the third image file, `leedsbutterfly/images/001_0082.jpg`, when used as the query example, is not included in the returned neighbors and thus found in the index.

[4] Second Incremental Update

Let us repeat again the above process, but using the third increment set (highlighted lines different from `3.run.sh`):

```
#!/usr/bin/env bash
set -e
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
cd "${SCRIPT_DIR}"

# Compute descriptors for new files, outputing a file that matches input
# files to thair SHA1 checksum values (their UUIDs)
../../../bin/scripts/compute_many_descriptors.py \
  -d \
  -c 2a.config.compute_many_descriptors.json \
  -f 4.ingest_files_3.txt \
  --completed-files 4.completed_files.csv
```

(continues on next page)

(continued from previous page)

```
13
14 # Extract UUIDs of files/descriptors just generated
15 cat 4.completed_files.csv | cut -d, -f2 > 4.uuids_for_processing.txt
16
17 # Compute hash codes for descriptors just generated, updating the target
18 # hash2uuids model file.
19 ../../../bin/scripts/compute_hash_codes.py -v -c 4.config.compute_hash_codes.json
```

After this, we should be able to query all three example files used before and see that they are all now included in the index. We will now also see that all 832 neighbors requested are returned for each of the queries, which equals the total number of files we have ingested over the above steps. If we increase `n` for a query, only 832 neighbors are returned, showing that there are 832 elements in the index at this point.

CHAPTER

FOUR

RELEASE PROCESS AND NOTES

4.1 Steps of the SMQTK Release Process

Please reference the SMQTK-Core release process documentation as that same process is applicable here, of course replacing uses of *smqtk-core* with *smqtk-indexing*.

4.2 Release Notes

4.2.1 SMQTK v0.15.0 Release Notes

This is the initial release of `smqtk-indexing`, spinning off from v0.14.0 of the monolithic SMQTK library.

Updates / New Features

CI

- Add Github actions workflow for CI.

Misc.

- Updated to use now publicly available `smqtk-dataprovider` package from PYPI.
- Updated to use now publicly available `smqtk-descriptors` package from PYPI.

Fixes

CI

- Fix mypy typechecking issues due to the increased strenuousness of the mypy options.

4.2.2 v0.16.0

This minor release updates the build system used to Poetry, updates the `smqtk-core` package dependency to a version >= 0.18.0 (the current release) and makes use of its importlib metadata pass-through.

Updates / New Features

Dependencies

- Remove dependency on `setuptool`'s `pkg_resources` module. Taking the stance of bullet number 5 in from Python's Packaging User-guide with regards to getting this package's version. The "needs to be installed" requirement from before is maintained.
- Added `ipython` (and appropriately supporting version of `jedi`) as development dependencies. Minimum versioning is set to support python 3.6 (current versions follow NEP 29 and thus require python 3.7+).

Documentation

- Revised documentation from the raw-from-mono-repo form into the start something more appropriate to this specific package.

Misc.

- Now standardize to using Poetry for environment/build/publish management.
  - Collapsed pytest configuration into the `pyproject.toml` file.

Testing

- Added terminal-output coverage report in the standard pytest config in the `pyproject.toml` file.

Fixes

- Update CI configurations to use Poetry.

4.2.3 v0.17.0

Updates / New Features

Dependencies

- Updated SMQTK-* deps to the latest patched versions.

Examples

- Added an example jupyter notebook detailing the process of building and querying against a `NearestNeighborsIndex` implementation.

Implementations

- Add default values for `FaissNearestNeighborsIndex` constructor parameters regarding descriptor and uid-map storage.

Fixes

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX