SMQTK

Release 0.14.0

Kitware, Inc.

Jun 22, 2022

CONTENTS

GitHub

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

CHAPTER

ONE

INSTALLATION

There are two ways to get ahold of SMQTK. The simplest is to install via the **pip** command. Alternatively, the source tree can be acquired and build/install SMQTK via CMake or `setuptools`.

1.1 From `pip`

In order to get the latest version of SMQTK from PYPI:

```
$ pip install --upgrade smqtk
```

This method will install all of the same functionality as when installing from source, but not as many plugins will be functional right out of the box. This is due to some plugin dependencies not being installable through pip. We will see more on this in the section below.

1.1.1 Extras

A few extras are defined for the `smqtk` package:

- **docs**
  - Dependencies for building SMQTK documentation.
- **caffe**
  - Minimum required packages for when using with the Caffe plugin.
- **flann**
  - Required packages for using FLANN-based plugins.
  - There is not an adequate version in the standard PYPI repository (>=1.8.4). For FLANN plugin functionality, it is recommended to either use your system package manager or SMQTK from source.
- **postgres**
  - Required packages for using PostgreSQL-based plugins.
- **solr**
  - Required packages for using Solr-based plugins.

1.2 From Source

Acquiring and building from source is different than installing from **pip** because:

- Includes FLANN and libSVM[1] libraries and (patched) python bindings in the CMake build. CMake installation additionally installs these components
- CPack packaging support (make RPMs, etc.).[2]

The inclusion of FLANN and libSVM in the source is generally helpful due to their lack of [up-to-date] availability in the PYPI and system package repositories. When available via a system package manager, it is often not easy to use when dealing with a virtual environment (e.g. virtualenv or Anaconda).

The sections below will cover the quick-start steps in more detail:

- *System dependencies*
- *Getting the Source*
- *Installing Python dependencies*
- *CMake Build*
- *Building the Documentation*

1.2.1 Quick Start

```
$ # Check things out
$ cd /where/things/should/go/
$ git clone https://github.com/Kitware/SMQTK.git source
$ # Install python dependencies to environment
$ pip install -r source/requirements.txt
$ # SMQTK build
$ mkdir build
$ pushd build
$ cmake ../source
$ make -j2
$ popd
$ # Set up SMQTK environment by sourcing file
$ . build/setup_env.build.sh
$ # Running tests
$ python source/setup.py test
```

1.2.2 System dependencies

In order retrieve and build from source, your system will need at a minimum:

- git
- cmake >=2.8
- c++ compiler (e.g. gcc, clang, MSVC etc.)

In order to run the provided IQR-search web-application, introduced later when describing the provided web services and applications, the following system dependencies are additionally required:

- MongoDB[3]

[1] Included libSVM is a customized version based on v3.1
[2] These features are largely still in development and may not work correctly yet.
[3] This requirement will hopefully go away in the future, but requires an alternate session storage implementation.

1.2.3 Getting the Source

The SMQTK source code is currently hosted on GitHub here.

To clone the repository locally:

```
$ git clone https://github.com/Kitware/SMQTK.git /path/to/local/source
```

1.2.4 Installing Python dependencies

After deciding and activating what environment to install python packages into (system or a virtual), the python dependencies should be installed based on the `requirements.*.txt` files found in the root of the source tree. These files detail different dependencies, and their exact versions tested, for different components of SMQTK.

The the core required python packages are detailed in: `requirements.txt`.

In addition, if you wish to be able to build the Sphinx based documentation for the project: `docs/readthedocs-reqs.txt`. These are separated because not everyone wishes or needs to build the documentation.

Other optional dependencies and what plugins they correspond to are found in: `requirements/optional.txt`

Note that if **conda**[4] is being used, not all packages listed in our requirements files may be found in **conda**'s repository.

Installation of python dependencies via pip will look like the following:

```
$ pip install -r requirements.txt [-r docs/readthedocs-reqs.txt]
```

Where the `docs/readthedocs-reqs.txt` argument is only needed if you intend to build the SMQTK documentation.

Building NumPy and SciPy

If NumPy and SciPy is being built from source when installing from **pip**, either due to a wheel not existing for your platform or something else, it may be useful or required to install BLAS or LAPACK libraries for certain functionality and efficiency.

Additionally, when installing these packages using **pip**, if the `LDFLAGS` or `CFLAGS`/`CXXFLAGS`/`CPPFLAGS` are set, their build may fail as they are assuming specific setups[5].

Additional Plugin Dependencies

Some plugins in SMQTK may require additional dependencies in order to run, usually python but sometimes not. In general, each plugin should document and describe their specific dependencies.

For example, the ColorDescriptor implementation required a 3rd party tool to downloaded and setup. Its requirements and restrictions are documented in `python/smqtk/algorithms/descriptor_generator/colordescriptor/INSTALL.md`.

[4] For more information on the **conda** command and system, see the Conda documentation.
[5] This may have changed since wheels were introduced.

1.2.5 CMake Build

See the example below for a simple example of how to build SMQTK

Navigate to where the build products should be located. It is recommended that this not be the source tree. Build products include some C/C++ libraries, python modules and generated scripts.

If the desired build directory, and run the following, filling in <...> slots with appropriate values:

```
$ cmake <source_dir_path>
```

Optionally, the *ccmake* command line utility, or the GUI version, may be run in order to modify options for building additional modules. Currently, the selection is very minimal, but may be expanded over time.

1.2.6 Building the Documentation

All of the documentation for SMQTK is maintained as a collection of *reStructuredText_* documents in the `docs` folder of the project. This documentation can be processed by the **`Sphinx`** documentation tool into a variety of documentation formats, the most common of which is HTML.

Within the `docs` directory is a Unix `Makefile` (for Windows systems, a `make.bat` file with similar capabilities exists). This `Makefile` takes care of the work required to run **`Sphinx`** to convert the raw documentation to an attractive output format. For example:

```
make html
```

Will generate HTML format documentation rooted a `docs/_build/html/index.html`.

The command:

```
make help
```

Will show the other documentation formats that may be available (although be aware that some of them require additional dependencies such as **`TeX`** or **`LaTeX`**.)

Live Preview

While writing documentation in a mark up format such as `reStructuredText` it is very helpful to be able to preview the formatted version of the text. While it is possible to simply run the `make html` command periodically, a more seamless version of this is available. Within the `docs` directory is a small Python script called `sphinx_server.py`. If you execute that file with the following command:

```
python sphinx_server.py
```

It will run small process that watches the `docs` folder for changes in the raw documentation `*.rst` files and re-runs **`make html`** when changes are detected. It will serve the resulting HTML files at http://localhost:5500. Thus having that URL open in a browser will provide you with a relatively up to date preview of the rendered documentation.

CHAPTER

TWO

QUICK-STARTS

2.1 Creating a External Plugin

In this quick-start tutorial, we will show how to create a new interface implementation within an external python package and expose it to the SMQTK plugin framework via entry-points in the package's `setup.py` file.

Lets assume that we are adding an implementation of the *`Classifier`* interface to some package we will call `MyPackage`, wrapping the use a scikit-learn classifier in a simple way.

2.1.1 Implementing the interface

In `MyPackage`, lets imagine we start a new file, `new_classifier.py` such that the module is importable via the module path `MyPackage.plugins.new_classifier`. In the following code blocks we will incrementally build up a functional implementation.

To start, we need to import the base interface and create a new class inheriting from this interface:

```
from sklearn.linear_model import LogisticRegression
from smqtk.algorithms import Classifier


class SklearnLogisticRegressionClassifier (LogisticRegression, Classifier):
    """
    A new, simple implementation of SMQTK's Classifier interface wrapping
    Scikit-Learn's LogisticRegression classifier.
    """

    @classmethod
    def is_usable(cls):
        # Required by the ``smqtk.utils.plugin.Pluggable`` parent
        return True

    def get_config(self):
        # Required by the ``smqtk.utils.configuration.Configurable`` parent.
        return {
            'C': self.C,
            'class_weight': self.class_weight,
            'dual': self.dual,
            'fit_intercept': self.fit_intercept,
            'intercept_scaling': self.intercept_scaling,
            'max_iter': self.max_iter,
            'multi_class': self.multi_class,
            'n_jobs': self.n_jobs,
```

(continues on next page)

(continued from previous page)

```
            'penalty': self.penalty,
            'random_state': self.random_state,
            'solver': self.solver,
            'tol': self.tol,
            'verbose': self.verbose,
            'warm_start': self.warm_start,
        }

    def get_labels(self):
        # Required by the ``smqtk.algorithms.Classifier`` parent
        try:
            return self.classes_.tolist()
        except AttributeError:
            raise RuntimeError("No model yet fit.")

    def _classify_arrays(self, array_iter):
        # Required by the ``smqtk.algorithms.Classifier`` parent
        x = numpy.asarray(list(array_iter))
        proba_arr = self.predict_proba(x)
        for proba in proba_arr:
            yield dict(zip(self.classes_, proba))
```

Since our source material happens to be a class itself, our implementation can inherit from the Scikit-learn base classifier as well as from the SMQTK interface. In other cases, encapsulation may be a better approach.

The methods defined in our implementation are overrides of abstract methods declared in our parent, and higher, SMQTK interfaces. Documentation of abstract methods can usually be found in the interface sources as doc-strings and often include what is expected to be the input and output data-types as well as any exception conditions that are expected. For example, the *Classifier* interface documents `get_labels` as raising a `RuntimeError` specifically if no model is loaded to access class labels. Additionally, *Classifier* documents for the `_classify_arrays` method that the input parameter `array_iter` should be an iterable type containing instances of the *DescriptorElement* class and should return an iterable type (usually a generator) of specifically formatted dictionaries.

This implementation happens to be compliant with the defaults of the *Configurable* interface because all of its constructor parameters are already JSON compliant (with the occasional exception of the "random_state" parameter when a `RandomState` instance is used, but we will ignore that here for simplicity). Thus, `get_default_config` will return a JSON-compliant dictionary of the default parameters as defined in Scikit-learn's implementation, as well as `from_config` will appropriately return a new instance based on the given JSON-compliant dictionary.

```
>>> dflt_config = SklearnLogisticRegressionClassifier.get_default_config()
>>> dflt_config
{'C': 1.0,
 'class_weight': None,
 'dual': False,
 'fit_intercept': True,
 'intercept_scaling': 1,
 'max_iter': 100,
 'multi_class': 'warn',
 'n_jobs': None,
 'penalty': 'l2',
 'random_state': None,
 'solver': 'warn',
 'tol': 0.0001,
 'verbose': 0,
 'warm_start': False}
```

(continues on next page)

(continued from previous page)

```
>>> new_dflt_inst = SklearnLogisticRegressionClassifier.from_config(dflt_config)
>>> new_dflt_inst.get_config() == dflt_config
True
```

2.1.2 Exposing via entry-points

In order to allow the SMQTK plugin framework to become aware of our new implementation we will need to update `MyPackage`'s `setup.py` file to add an entry-point. Since we assumed above that we created our implementation in the module `MyPackage.plugins.new_classifier`, the following should be added:

```
setup(
    ...
    entry_points={
        ...
        'smqtk_plugins': [
            "MyPackage_plugins = MyPackage.plugins.new_classifier",
        ]
    }
)
```

Notes on adding entry-points:

- The value to the left of the ='s sign must be unique across installed module providing extensions for the entry-point. A safe method
- Multiple extensions may be specified. This may be useful if your implementations naturally belong in different locations within your package.
- Currently SMQTK only supports providing modules in its extensions. Otherwise a warning will be emitted and that extension will be ignored.

Now, after re-installing `MyPackage`, SMQTK's plugin framework should be able to discover this new implementation:

```
>>> from smqtk.algorithms import Classifier
>>> classifier.get_impls()
{..., MyPackage.plugins.new_classifier.SklearnLogisticRegressionClassifier,
 ...}
```

CHAPTER

THREE

SMQTK ARCHITECTURE OVERVIEW

SMQTK provides at its lowest level semantics for plugins and configuration. These are provided by some utility functions and two mixin classes: *smqtk.utils.plugin.Pluggable* and *smqtk.utils.configuration.Configurable*. These are explained further in the "Plugins and Configuration" section.

Subsequent to these two mixin classes, SMQTK provides two main categories of interfaces: algorithms and data representations. This organization of philosophy roughly aligns with the concept of data oriented design. Algorithms are usually interfaces defining a behavioral or transformative action(s), abstracting away how that behavior or transformation is achieved. Data representation interfaces define the encapsulation of some data structure, abstracting away where that data is stored..

Building upon algorithm and data representation interfaces, there is a sub-module providing some general web services: `smqtk.web`. Of likely interest is headless IQR web-service (`smqtk.web.iqr_service`). There is also a demonstration IQR web application with a simple UI as well as other headless web services (`smqtk.web.search_app`).

3.1 Plugins and Configuration

SMQTK provides plugin and configuration utilities to support the creation of interface classes that have a convenient means of accessing implementing types, paired ability to dynamically instantiate interface implementations based on a configuration derived by constructor introspection.

While these two primary mixin classes function independently and can be utilized on their own, their combination is symbiotic and allows for users of derivative interfaces to create tools in terms of the interfaces and leave the specific selection of implementations for configuration time.

Later, we will introduce the two categories of configurable and (usually) pluggable class classes found within SMQTK.

3.1.1 The `Pluggable` Mixin

Motivation: We want to be able to define interfaces to generic concepts and structures that higher level tools can be defined around without strictly catering themselves to any particular implementation, while additionally allowing freedom in implementation variety without overly restricting implementations.

In SMQTK, this is addressed via the *Pluggable* abstract mixin class:

```
import abc
from smqtk.utils.plugin import Pluggable

class MyInterface(Pluggable):

    @abc.abstractmethod
```

(continues on next page)

(continued from previous page)

```
    def my_behavior(self, x: str) -> int:
        """My fancy behavior."""

if __name__ == "__main__":
    # Discover currently available implementations and print out their names
    impl_types = MyInterface.get_impls()
    print("MyInterface implementations:")
    for t in impl_types:
        print(f"- {t.__name__}")
```

Interfaces and Implementations

Classes that inherit from the *Pluggable* mixin are considered either pluggable interfaces or plugin implementations depending on whether they fully implement abstract methods.

Interface implementations bundled within SMQTK are generally defined alongside their parent interfaces. However, other sources, e.g. other python packages, may expose their own plugin implementations via setting a system environment variable or via python extensions.

3.1.2 The `Configurable` Mixin

Motivation: We want generic helpers to enable serializable configuration for classes while minimally impacting standard class development.

SMQTK provides the *Configurable* mixin class as well as other helper utility functions in *smqtk.utils.configuration* for generating class instances from configurations. These use python's `inspect` module to determine constructor parameterization and default configurations.

Currently this module uses the JSON-serializable format as the basis for input and output configuration dictionaries as a means of defining a relatively simple playing field for communication. Serialization and deserialization is detached from these configuration utilities so tools may make their own decisions there. Python dictionaries are used as a medium in between serialization and configuration input/output.

Classes that inherit from *Configurable* *do* need to at a minimum implement the *get_config()* instance method.

3.1.3 Algorithms and Representations - The Combination

Interfaces found in SMQTK are generally binned into two categories: representations and algorithms.

Algorithms are interfaces to some function or operation, specifically parameterized through their constructor and generally parameterized via the algorithm's interface. The *SmqtkAlgorithm* base class inherits from both *Pluggable* and *Configurable* mixins so that all descendents gain access to the synergy they provide. These are located under the `smqtk.algorithms` sub-module.

Representations are interfaces to structures that are intended to specifically store some sort of data structure. Currently, the *SmqtkRepresentation* only inherits directly from *Configurable*, as there are some representational structures which desire configurability but to which variable implementations do not make sense (like *DescriptorElementFactory*). However most sub-classes do additionally inherit from *Pluggable* (like *DescriptorElement*). These are located under the `smqtk.representation` sub-module.

3.1.4 Implementing a Pluggable Interface

The following are examples of how to add and expose new plugin implementations for existing algorithm and representation interfaces.

SMQTK's plugin discovery via the *get_impls()* method currently allows for finding a plugin implementations in 3 ways:

- sub-classes of an interface type defined in the current runtime.
- within python modules listed in the environment variable specified by `YourInterface.PLUGIN_ENV_VAR`. (default SMQTK environment variable name is `SMQTK_PLUGIN_PATH`, which is defined in `Pluggable.PLUGIN_ENV_VAR`).
- within python modules specified under the entry point extensions namespace defined by `YourInterface.PLUGIN_NAMESPACE` (default SMQTK extension namespace is `smqtk_plugins`, which is defined in `Pluggable.PLUGIN_NAMESPACE`).

Within SMQTK

A new interface implementation within the SMQTK source-tree is generally implemented or exposed parallel to where the parent interface is defined. This has been purely for organizational purposes. Once we define our implementation, we will then expose that type in an existing module that is already referenced in SMQTK's list of entry point extensions.

In this example, we will show how to create a new implementation for the *Classifier* algorithm interface. This interface is defined within SMQTK at, from the root of the source tree, `python/smqtk/algorithms/classifier/_interface_classifier.py`. We will create a new file, `some_impl.py`, that will be placed in the same directory.

We'll define our new class, lets call it `SomeImpl`, in a file `some_impl.py`:

```
python/
└── smqtk/
    └── algorithms/
        └── classifier/
            ├── _interface_classifier.py
            ├── some_impl.py      # new
            └── ...
```

In this file we will need to define the `SomeImpl` class and all parent class abstract methods in order for the class to satisfy the definition of an "implementation":

```
from smqtk.algorithms import Classifier

class SomeImpl (Classifier):
    """
    Some documentation for this specific implementation.
    """

    # Our implementation-specific constructor.
    def __init__(self, paramA=1, paramB=2):
        ...

    # Abstract methods from Configurable.
    # (Classifier -> SmqtkAlgorithm -> Configurable)
    def get_config(self):
        # As per Configurable documentation, this returns the same non-self
```

(continues on next page)

(continued from previous page)

```
        # keys as the constructor.
        return {
            "paramA": ...,
            "paramB": ...,
        }

    # Classifier's abstract methods.
    def get_labels(self):
        ...

    def _classify_arrays(self, array_iter):
        ...
```

The final step to making this implementation discoverable is to add an import of this class to the existing hub of classifier plugins in `python/smqtk/algorithms/classifier/_plugins.py`:

```
...
from .some_impl import SomeImpl
```

With all abstract methods defined, this implementation will now be included in the returned set of implementation types for the parent *Classifier* interface:

```
>>> from smqtk.algorithms import Classifier
>>> Classifier.get_impls()
set([..., smqtk.algorithms.classifier.some_impl.SomeImpl, ...])
```

`SomeImpl` above should also be all set for configuration because it defines the one required abstract method *get_config()* and because its constructor is only anticipating JSON-compliant types. If more complicated types are desired by the constructor the additional methods would need to be overridden/extended as defined in the *smqtk.utils.configuration* module.

Within another python package

When implementing a pluggable interface in another python package, the proper method of export is via a package's entry point specifications using the namespace key defined by the parent interface (by default the `smqtk_plugins` value is defined by `smqtk.utils.plugin.Pluggable.PLUGIN_NAMESPACE`).

For example, let's assume that a separate python package, `OtherPackage` we'll call it, defines a *Classifier*-implementing sub-class `OtherClassifier` in the module `OtherPackage.other_classifier`. This module location can be exposed via the package's `setup.py` entry points metadata, using the `smqtk_plugins` key, like the following:

```
from setuptools import setup

...

setup(
    ...
    entry_points={
        'smqtk_plugins': 'my_plugins = OtherPackage.other_classifier'
    }
)
```

If this other package had multiple sub-modules in which SMQTK plugins were defined, the `smqtk_plugins` entry value may instead be a list:

```
setup(
    ...
    entry_points={
        'smqtk_plugins': [
            'classifier_plugins = OtherPackage.other_classifier',
            'other_plugins = OtherPackage.other_plugins',
        ]
    }
)
```

3.1.5 Reference

smqtk.utils.plugin

Helper functions and mixin interface for implementing class type discovery, filtering and a convenience mixin class.

This package provides a number of *discover_via_...* functions that return sets of type instances as found by the method described by that function.

These methods may be composed to create a pool of types that may be then filtered via the *filter_plugin_types* function to those types that are specifically "plugin types" for the given interface class. See the *is_valid_plugin* function documentation for what it means to be a "plugin" of an interface type.

While the above are defined in fairly general terms, the *Pluggable* class type defined last here is a mixin class that utilizes all of the above in a manner specific manner for the purposes of SMQTK. This mixin class defines the class-method `get_impls()` that will return currently discoverable plugins underneath the type it was called on. This discovery will follow the values of the `PLUGIN_ENV_VAR` and `PLUGIN_NAMESPACE` class variables defined in the interface class you are calling `get_impls()` from, using inherited values if not immediately specified.

Because these plugin semantics are pretty low level and commonly utilized, logging can be extremely verbose. Logging in this module, while still exists, is set to emit only at log level 1 or lower ("trace").

NOTE: The type annotations for *discover_via_subclasses* and *filter_plugin_types* are currently set to the broad *Type* annotation. Ideally these should use *Type[T]* instead, but there is currently a known issue with mypy where it aggressively assumes that an annotated type *must* be constructable, so it emits an error when the functions are called with an abstract *interface_type*. When this is resolved in mypy these annotations should be updated.

exception smqtk.utils.plugin.**NotAModuleError**
Exception for when the *discover_via_entrypoint_extensions* function found an entrypoint that was *not* a module specification.

exception smqtk.utils.plugin.**NotUsableError**
Exception thrown when a pluggable class is constructed but does not report as usable.

class smqtk.utils.plugin.**Pluggable**
Interface for classes that have plugin implementations

classmethod get_impls() → Set[Type[P]]
Discover and return a set of classes that implement the calling class.

See the `get_plugins` function for more details on the logic of how implementing classes (aka "plugins") are discovered.

The class-level variables `PLUGIN_ENV_VAR` and `PLUGIN_HELPER_VAR` may be overridden to change what environment and helper variable are looked for, respectively.

Returns Set of discovered class types that are considered "valid" plugins of this type. See *is_valid_plugin()* for what we define a "valid" type to be be relative to this class.

classmethod is_usable() → bool

Check whether this class is available for use.

Since certain plugin implementations may require additional dependencies that may not yet be available on the system, or other runtime conditions, this method may be overridden to check for those and return a boolean saying if the implementation is available for usable. When this method returns *True*, the class is declaring that it should be constructable and usable in the current environment.

By default, this method will return True unless a sub-class overrides this class-method with their specific logic.

NOTES:

- This should be a class method
- **When an implementation is deemed not usable, this should emit a** (user) warning, or some other kind of logging, detailing why the implementation is not available for use.

Returns Boolean determination of whether this implementation is usable in the current environment.

Return type bool

smqtk.utils.plugin.**discover_via_entrypoint_extensions**(*entrypoint_ns: str*) → Set[Type]

Discover and return types defined in modules exposed through the entry-point extensions defined for the given namespace by installed python packages.

Other installed python packages may define one or more extensions for a namespace, as specified by *ns*, in their "setup.py". This should be a single or list of extensions that specify modules within the installed package where plugins for export are implemented.

Currently, this method only accepts extensions that export a module as opposed to specifications of a specific attribute in a module. This is due to other methods of type discovery not necessarily honoring the selectivity that specific attribute specification provides (Looking at you *__subclasses__*...).

For example, as a single specification string:

```
...
entry_points={
    "smqtk_plugins": "my_package = my_package.plugins"
]
...
```

Or in list form of multiple specification strings:

```
...
entry_points = {
    "smqtk_plugins": [
        "my_package_mode_1 = my_package.mode_1.plugins",
        "my_package_mode_2 = my_package.mode_2.plugins",
    ]
}
...
```

Parameters **entrypoint_ns** – The name of the entry-point mapping in to look for extensions under.

Returns Set of discovered types from the modules and class types specified in the extensions under the specified entry-point.

smqtk.utils.plugin.**discover_via_env_var**(*env_var: str*) → Set[Type]

Discover and return types specified in python-importable modules specified in the the given environment variable.

We expect the given environment variable to define zero or more python module paths from which to yield all contained type definitions (i.e. things that descent from *type*). If there is an empty path element, it is skipped (e.g. "foo::bar:baz" will only attempt importing *foo*, *bar* and *baz* modules).

These python module paths should be separated with the same separator as would be used in the PYTHONPATH environment variable specification.

If a module defines no class types, then no types are included from that source for return.

An expected use-case for this discovery method is for modules that are not installed but otherwise accessible via the python search path. E.g. local modules, modules accessible through PYTHONPATH search path modification, modules accessible through *sys.path* modification.

Any errors raised from attempting to import a module are propagated upward.

Parameters **env_var** – The name of the environment variable to read from.

Raises **ModuleNotFoundError** – When one or more module paths specified in the given environment variable are not importable.

Returns Set of discovered types from the modules specified in the environment variable's contents.

smqtk.utils.plugin.**discover_via_subclasses**(*interface_type: Type*) → Set[Type]

Utilize the __subclasses__ to discover nested subclasses for a given interface type.

This approach will be able to observe any implementations that have been defined, anywhere at all, at the point of invocation, which can circumvent efforts towards specificity that other discovery methods may provide. For example, *discover_via_entrypoint_extensions* may return a single type that was specifically exported from a module whereas this method will, called afterwards, yield all the other types defined in that entry-point-imported module.

The use of this discovery method may also result in different returns depending on the import state at the time of invocation. E.g. further imports may increase the quantity of returns from this function.

This function uses depth-first-search when traversing sub-class tree.

Reference: https://docs.python.org/3/library/stdtypes.html#class.__subclasses__

NOTE: subclasses are retained via weak-references, so if a normal condition is exposing types from something that otherwise raised an exception or if a local definition is leaking, apparently an *import gc; gc.collect()* wipes out the return as long as it's not referenced, of course as long as its reference is not retained by something.

Parameters **interface_type** – The interface type to recursively find sub-classes under.

Returns Set of recursive subclass types under *interface_type*.

smqtk.utils.plugin.**filter_plugin_types**(*interface_type: Type*, *candidate_pool: Collection[Type]*) → Set[Type]

Filter the given set of types to those that are "plugins" of the given interface type.

See the documentation for *is_valid_plugin()* for what we define a "plugin type" to be relative to the given *interface_type*.

We consider that there may be duplicate type instances in the given candidate pool. Due to this we will consider an instance of a type only once and return a set type to contain the validated types.

Parameters

- **interface_type** – The parent type to filter on.

- **candidate_pool** – Some iterable of types from which to collect interface type plugins from.

Returns Set of types that are considered "plugins" of the interface types following the above listed rules.

smqtk.utils.plugin.**is_valid_plugin**(*cls: Type*, *interface_type: Type*) → bool

Determine if a class type is a valid candidate for plugin discovery.

In particular, the class type `cls` must satisfy several conditions:

1. It must not literally be the given interface type.
2. It must be a strict subtype of `interface_type`.
3. It must not be an abstract class. (i.e. no lingering abstract methods or properties if the *abc.ABCMeta* metaclass has been used).
4. If the cls is a subclass of Pluggable, it must report as usable via its is_usable() class method.

Logging for this function, when enabled can be very verbose, and is only active with a logging level of 1 or lower.

Parameters

- **cls** – The class type whose validity is being tested
- **interface_type** – The base class under consideration

Returns `True` if the class is a valid candidate for discovery, and `False` otherwise.

Return type bool

smqtk.utils.configuration

Helper interface and functions for generalized object configuration, to and from JSON-compliant dictionaries.

While this interface and utility methods should be general enough to add JSON-compliant dictionary-based configuration to any object, this was created in mind with the SMQTK plugin module.

Standard configuration dictionaries should be JSON compliant take the following general format:

```
{
    "type": "one-of-the-keys-below",
    "ClassName1": {
        "param1": "val1",
        "param2": "val2"
    },
    "ClassName2": {
        "p1": 4.5,
        "p2": null
    }
}
```

The "type" key is considered a special key that should always be present and it specifies one of the other keys within the same dictionary. Each other key in the dictionary should be the name of a `Configurable` inheriting class type. Usually, the classes named within a block inherit from a common interface and the "type" value denotes a selection of a specific sub-class for use, though this is not required property of these constructs.

class smqtk.utils.configuration.**Configurable**

Interface for objects that should be configurable via a configuration dictionary consisting of JSON types.

classmethod from_config(*config_dict: Dict*, *merge_default: bool = True*) → C

Instantiate a new instance of this class given the configuration JSON-compliant dictionary encapsulating initialization arguments.

This base method is adequate without modification when a class's constructor argument types are JSON-compliant. If one or more are not, however, this method then needs to be overridden in order to convert from a JSON-compliant stand-in into the more complex object the constructor requires. It is recommended that when complex types *are* used they also inherit from the *Configurable* in order to hopefully make easier the conversion to and from JSON-compliant stand-ins.

When this method *does* need to be overridden, this usually looks like the following pattern:

```
class MyClass (Configurable):

    @classmethod
    def from_config(cls, config_dict, merge_default=True):
        # Optionally guarantee default values are present in the
        # configuration dictionary.  This statement pairs with the
        # ``merge_default=False`` parameter in the super call.
        # This also in effect shallow copies the given non-dictionary
        # entries of ``config_dict`` due to the merger with the
        # default config.
        if merge_default:
            config_dict = merge_dict(cls.get_default_config(),
                                     config_dict)

        #
        # Perform any overriding here.
        #

        # Create and return an instance using the super method.
        return super(MyClass, cls).from_config(config_dict,
                                               merge_default=False)
```

This method should not be called via super unless an instance of the class is desired.

Parameters

- **config_dict** (*dict*) – JSON compliant dictionary encapsulating a configuration.
- **merge_default** (*bool*) – Merge the given configuration on top of the default provided by `get_default_config`.

Returns Constructed instance from the provided config.

abstract get_config()

Return a JSON-compliant dictionary that could be passed to this class's `from_config` method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn't make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method's returned dictionary may leave those parameters out. In such cases, the object's `from_config` class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns JSON type compliant configuration dictionary.

Return type dict

classmethod get_default_config() → Dict[str, Any]

Generate and return a default configuration dictionary for this class. This will be primarily used for generating what the configuration dictionary would look like for this class without instantiating it.

By default, we observe what this class's constructor takes as arguments, turning those argument names into configuration dictionary keys. If any of those arguments have defaults, we will add those values into the configuration dictionary appropriately. The dictionary returned should only contain JSON compliant value types.

It is not be guaranteed that the configuration dictionary returned from this method is valid for construction of an instance of this class.

Returns Default configuration dictionary for the class.

Return type dict

```
>>> # noinspection PyUnresolvedReferences
>>> class SimpleConfig(Configurable):
...     def __init__(self, a=1, b='foo'):
...         self.a = a
...         self.b = b
...     def get_config(self):
...         return {'a': self.a, 'b': self.b}
>>> self = SimpleConfig()
>>> config = self.get_default_config()
>>> assert config == {'a': 1, 'b': 'foo'}
```

smqtk.utils.configuration.**cls_conf_from_config_dict**(*config: Dict*, *type_iter: Iterable[Type[T]]*) → Tuple[Type[T], Dict]

Helper function for getting the appropriate type and configuration sub-dictionary based on the provided "standard" SMQTK configuration dictionary format (see above module documentation).

Parameters

- **config** – Configuration dictionary to draw from.
- **type_iter** – An iterable of class types to select from.

Raises **ValueError** –

This may be raised if:

- type field not present in `config`.
- type field set to `None`
- type field did not match any available configuration in the given config.
- Type field did not specify any implementation key.

Returns Appropriate class type from `type_iter` that matches the configured type as well as the sub-dictionary from the configuration. From this return, `type.from_config(config)` should be callable.

smqtk.utils.configuration.**cls_conf_to_config_dict**(*cls: Type*, *conf: Dict*) → Dict

Helper function for creating the appropriate "standard" smqtk configuration dictionary given a *Configurable*-implementing class and a configuration for that class.

This very simple function simply arranges a semantic class key and an associated dictionary into a normal pattern used for configuration in SMQTK:

```
>>> class SomeClass (object):
```

... pass >>> cls_conf_to_config_dict(SomeClass, {0: 0, 'a': 'b'}) == { ... 'type': 'smqtk.utils.configuration.SomeClass', ... 'smqtk.utils.configuration.SomeClass': {0: 0, 'a': 'b'} ... } True

Parameters

- **cls** (*type[Configurable]*) – A class type implementing the *Configurable* interface.
- **conf** (*dict*) – SMQTK standard type-optioned configuration dictionary for the given class and dictionary pair.

Returns "Standard" SMQTK JSON-compliant configuration dictionary

Return type dict

smqtk.utils.configuration.**configuration_test_helper**(*inst: C*, *config_ignored_params: Union[Set, FrozenSet] = frozenset({})*, *from_config_args: Sequence = ()*) → Tuple[C, C, C]

Helper function for testing the get_default_config/from_config/get_config methods for class types that in part implement the Configurable mixin class. This function also tests that `inst`'s parent class type's `get_default_config` returns a dictionary whose keys' match the constructor's inspected parameters (except "self" of course).

This constructs 3 additional instances based on the given instance following the pattern:

```
inst-1  ->  inst-2  ->  inst-3
        ->  inst-4
```

This refers to `inst-2` and `inst-4` being constructed from the config from `inst`, and `inst-3` being constructed from the config of `inst-2`. The equivalence of each instance's config is cross-checked with the other instances. This is intended to check that a configuration yields the same class configurations and that the config does not get mutated by nested instance construction.

This function uses assert calls to check for consistency.

We return all instances constructed in case the caller wants to make additional instance integrity checks.

Parameters

- **inst** (Configurable) – Configurable-mixin inheriting class to test.
- **config_ignored_params** (*set[str]*) – Set of parameter names in the instance type's constructor that are ignored by `get_default_config` and `from_config`. This is empty by default.
- **from_config_args** (*tuple*) – Optional additional positional arguments to the input `inst.from_config` method after the configuration dictionary.

Returns Instance 2, 3, and 4 as described above.

Return type (*Configurable*,*Configurable*,*Configurable*)

smqtk.utils.configuration.**from_config_dict**(*config: Dict*, *type_iter: Iterable[Type[C]]*, **args: Any*) → C

Helper function for instantiating an instance of a class given the configuration dictionary `config` from available types provided by `type_iter` via the `Configurable` interface's `from_config` class-method.

`args` are additionally positional arguments to be passed to the type's `from_config` method on return.

Example: >>> from smqtk.representation import DescriptorElement >>> example_config = { ... 'type': 'smqtk.representation.descriptor_element.local_elements.DescriptorMemoryElement', ... 'smqtk.representation.descriptor_element.local_elements.DescriptorMemoryElement': {}, ... } >>> inst = from_config_dict(example_config, DescriptorElement.get_impls(), ... 'type-str', 'some-uuid') >>> from smqtk.representation.descriptor_element.local_elements import DescriptorMemoryElement >>> isinstance(inst, DescriptorMemoryElement) True

Raises

- **`ValueError`** –

  This may be raised if:

  - type field not present in `config`.
  - type field set to `None`
  - type field did not match any available configuration in the given config.
  - Type field did not specify any implementation key.

- **`AssertionError`** – This may be raised if the class specified as the configuration *type*, is present in the given `type_iter` but is not a subclass of the `Configurable` interface.
- **`TypeError`** – Insufficient/incorrect initialization parameters were specified for the specified `type`'s constructor.

Parameters

- **`config`** – Configuration dictionary to draw from.
- **`type_iter`** – An iterable of class types to select from.
- **`args`** (*object*) – Other positional arguments to pass to the configured class' `from_config` class method.

Returns Instance of the configured class type as specified in `config` and as available in `type_iter`.

`smqtk.utils.configuration.`**`make_default_config`**(*configurable_iter: Iterable[Type[C]]*) → Dict[str, Union[None, str, Dict]]

Generated default configuration dictionary for the given iterable of Configurable-inheriting types.

For example, assuming the following simple class that descends from `Configurable`, we would expect the following behavior:

```
>>> # noinspection PyAbstractClass
>>> class ExampleConfigurableType (Configurable):
...     def __init__(self, a, b):
...         ''' Dummy constructor '''
>>> make_default_config([ExampleConfigurableType]) == {
...     'type': None,
...     'smqtk.utils.configuration.ExampleConfigurableType': {
...         'a': None,
...         'b': None,
...     }
... }
True
```

Note that technically `ExampleConfigurableType` is still abstract as it does not implement `get_config`. The above call to `make_default_config` still functions because we only use the `get_default_config` class method and do not instantiate any types given to this function. While functionally acceptable, it is generally not recommended to draw configurations from abstract classes.

Parameters **configurable_iter** – An iterable of class types class types that sub-class `Configurable`.

Returns Base configuration dictionary with an empty `type` field, and containing the types and initialization parameter specification for all implementation types available from the provided getter method.

`smqtk.utils.configuration.`**to_config_dict**(*c_inst:* smqtk.utils.configuration.Configurable) → Dict

Helper function that transforms the configuration dictionary retrieved from `configurable_inst` into the "standard" SMQTK configuration dictionary format (see above module documentation).

For example, with a simple DataFileElement: >>> from smqtk.representation.data_element.file_element import DataFileElement >>> e = DataFileElement(filepath='/path/to/file.txt', readonly=True) >>> to_config_dict(e) == { ... "type": "smqtk.representation.data_element.file_element.DataFileElement", ... "smqtk.representation.data_element.file_element.DataFileElement": { ... "filepath": "/path/to/file.txt", ... "readonly": True, ... "explicit_mimetype": None, ... } ... } True

Parameters **c_inst** (`Configurable`) – Instance of a class type that subclasses the `Configurable` interface.

Returns Standard format configuration dictionary.

Return type dict

Reload Use Warning

While the `smqtk.utils.plugin.get_plugins()` function allows for reloading discovered modules for potentially new content, this is not recommended under normal conditions. When reloading a plugin module after `pickle` serializing an instance of an implementation, deserialization causes an error because the original class type that was pickled is no longer valid as the reloaded module overwrote the previous plugin class type.

3.2 Data Abstraction

An important part of any algorithm is the data its working over and the data that it produces. An important part of working with large scales of data is where the data is stored and how its accessed. The `smqtk.representation` module contains interfaces and plugins for various core data structures, allowing plugin implementations to decide where and how the underlying raw data should be stored and accessed. This potentially allows algorithms to handle more data that would otherwise be feasible on a single machine.

class `smqtk.representation.`**SmqtkRepresentation**

Interface for data representation interfaces and implementations.

Data should be serializable, so this interface adds abstract methods for serializing and de-serializing SMQTK data representation instances.

3.2.1 Data Representation Structures

The following are the core data representation interfaces.

Note: It is required that implementations have a common serialization format so that they may be stored or transported by other structures in a general way without caring what the specific implementation is. For this we require that all implementations be serializable via the `pickle` (and thus `cPickle`) module functions.

DataElement

class smqtk.representation.**DataElement**

Abstract interface for a byte data container.

The primary "value" of a `DataElement` is the byte content wrapped. Since this can technically change due to external forces, we cannot guarantee that an element is immutable. Thus `DataElement` instances are not considered generally hashable. Specific implementations may define a `__hash__` method if that implementation reflects a data source that guarantees immutability.

UUIDs should be cast-able to a string and maintain unique-ness after conversion.

clean_temp()

Clean any temporary files created by this element. This does nothing if no temporary files have been generated for this element yet.

abstract content_type()

Returns Standard type/subtype string for this data element, or None if the content type is unknown.

Return type str or None

classmethod from_uri(*uri*)

Construct a new instance based on the given URI.

This function may not be implemented for all DataElement types.

Parameters **uri** (*str*) – URI string to resolve into an element instance

Raises

- **NoUriResolutionError** – This element type does not implement URI resolution.
- **smqtk.exceptions.InvalidUriError** – This element type could not resolve the provided URI string.

Returns New element instance of our type.

Return type *DataElement*

abstract get_bytes()

Returns Get the bytes for this data element.

Return type bytes

abstract is_empty()

Check if this element contains no bytes.

The intend of this method is to quickly check if there is any data behind this element, ideally without having to read all/any of the underlying data.

Returns If this element contains 0 bytes.

Return type bool

is_read_only()

Returns If this element can only be read from.

Return type bool

md5()
Get the MD5 checksum of this element's binary content.

Returns MD5 hex checksum of the data content.

Return type str

abstract set_bytes(*b*)
Set bytes to this data element.

Not all implementations may support setting bytes (check `writable` method return).

This base abstract method should be called by sub-class implementations first. We check for mutability based on `writable()` method return.

Parameters **b** (*bytes*) – bytes to set.

Raises **ReadOnlyError** – This data element can only be read from / does not support writing.

sha1()
Get the SHA1 checksum of this element's binary content.

Returns SHA1 hex checksum of the data content.

Return type str

sha512()
Get the SHA512 checksum of this element's binary content.

Returns SHA512 hex checksum of the data content.

Return type str

to_buffered_reader()
Wrap this element's bytes in a `io.BufferedReader` instance for use as file-like object for reading.

As we use the `get_bytes` function, this element's bytes must safely fit in memory for this method to be usable.

Returns New BufferedReader instance

Return type io.BufferedReader

uuid()
UUID for this data element.

This many take different forms from integers to strings to a uuid.UUID instance. This must return a hashable data type.

By default, this ends up being the hex stringification of the SHA1 hash of this data's bytes. Specific implementations may provide other UUIDs, however.

Returns UUID value for this data element. This return value should be hashable.

Return type collections.abc.Hashable

abstract writable()

Returns if this instance supports setting bytes.

Return type bool

write_temp(*temp_dir=None*)
Write this data's bytes to a temporary file on disk, returning the path to the written file, whose extension is guessed based on this data's content type.

It is not guaranteed that the returned file path does not point to the original data, i.e. writing to the returned filepath may modify the original data.

NOTE: The file path returned should not be explicitly removed by the user. Instead, the `clean_temp()` method should be called on this object.

Parameters **temp_dir** (*None or str*) – Optional directory to write temporary file in, otherwise we use the platform default temporary files directory. If this is an empty string, we count it the same as having provided None.

Returns Path to the temporary file

Return type str

DataSet

class smqtk.representation.**DataSet**
Abstract interface for data sets, that contain an arbitrary number of `DataElement` instances of arbitrary implementation type, keyed on `DataElement` UUID values.

This should only be used with DataElements whose byte content is expected not to change. If they do, then UUID keys may no longer represent the elements associated with them.

abstract add_data(**elems*)
Add the given data element(s) instance to this data set.

NOTE: Implementing methods should check that input elements are in fact DataElement instances.

Parameters **elems** (smqtk.representation.DataElement) – Data element(s) to add

abstract count()

Returns The number of data elements in this set.

Return type int

abstract get_data(*uuid*)
Get the data element the given uuid references, or raise an exception if the uuid does not reference any element in this set.

Raises **KeyError** – If the given uuid does not refer to an element in this data set.

Parameters **uuid** (*collections.abc.Hashable*) – The uuid of the element to retrieve.

Returns The data element instance for the given uuid.

Return type *smqtk.representation.DataElement*

abstract has_uuid(*uuid*)
Test if the given uuid refers to an element in this data set.

Parameters **uuid** (*collections.abc.Hashable*) – Unique ID to test for inclusion. This should match the type that the set implementation expects or cares about.

Returns True if the given uuid matches an element in this set, or False if it does not.

Return type bool

abstract uuids()

Returns A new set of uuids represented in this data set.

Return type set

DescriptorElement

class smqtk.representation.**DescriptorElement**(*type_str*, *uuid*)

Abstract descriptor vector container.

This structure supports implementations that cache descriptor vectors on a per-UUID basis.

UUIDs must maintain unique-ness when transformed into a string.

Descriptor element equality based on shared descriptor type and vector equality. Two descriptor vectors that are generated by different types of descriptor generator should not be considered the same (though, this may be up for discussion).

Stored vectors should be effectively immutable.

classmethod from_config(*config_dict*, *type_str*, *uuid*, *merge_default=True*)

Instantiate a new instance of this class given the desired type, uuid, and JSON-compliant configuration dictionary.

Parameters

- **type_str** (*str*) – Type of descriptor. This is usually the name of the content descriptor that generated this vector.
- **uuid** (*collections.abc.Hashable*) – Unique ID reference of the descriptor.
- **config_dict** (*dict*) – JSON compliant dictionary encapsulating a configuration.
- **merge_default** (*bool*) – Merge the given configuration on top of the default provided by `get_default_config`.

Returns Constructed instance from the provided config.

Return type *DescriptorElement*

classmethod get_default_config()

Generate and return a default configuration dictionary for this class. This will be primarily used for generating what the configuration dictionary would look like for this class without instantiating it.

By default, we observe what this class's constructor takes as arguments, aside from the first two assumed positional arguments, turning those argument names into configuration dictionary keys. If any of those arguments have defaults, we will add those values into the configuration dictionary appropriately. The dictionary returned should only contain JSON compliant value types.

It is not be guaranteed that the configuration dictionary returned from this method is valid for construction of an instance of this class.

Returns Default configuration dictionary for the class.

Return type dict

classmethod get_many_vectors(*descriptors*)

Get an iterator over vectors associated with given descriptors.

Note Most subclasses should override internal method *_get_many_vectors* rather than this external wrapper function. If a subclass does override this classmethod, it is responsible for appropriately handling any valid DescriptorElement, regardless of subclass.

Parameters **descriptors** (*collections.abc.Iterable[smqtk.representation.descriptor_element.DescriptorElement]*) – Iterable of descriptors to query for.

Returns Iterable of vectors associated with the given descriptors or None if the descriptor has no associated vector. Results are returned in the order that descriptors were given.

Return type list[numpy.ndarray | None]

abstract has_vector()

Returns Whether or not this container current has a descriptor vector stored.

Return type bool

abstract set_vector(*new_vec*)
Set the contained vector.

If this container already stores a descriptor vector, this will overwrite it.

Parameters **new_vec** (*numpy.ndarray*) – New vector to contain.

Returns Self.

Return type DescriptorMemoryElement

type()

Returns Type label type of the DescriptorGenerator that generated this vector.

Return type str

uuid()

Returns Unique ID for this vector.

Return type collections.abc.Hashable

abstract vector()

Returns Get the stored descriptor vector as a numpy array. This returns None of there is no vector stored in this container.

Return type numpy.ndarray or None

DescriptorSet

class smqtk.representation.**DescriptorSet**
Index of descriptors, keyed and query-able by descriptor UUID.

Note that these indexes do not use the descriptor type strings. Thus, if a set of descriptors has multiple elements with the same UUID, but different type strings, they will bash each other in these indexes. In such a case, when dealing with descriptors for different generators, it is advisable to use multiple indices.

abstract add_descriptor(*descriptor*)
Add a descriptor to this index.

Adding the same descriptor multiple times should not add multiple copies of the descriptor in the index (based on UUID). Added descriptors overwrite indexed descriptors based on UUID.

Parameters **descriptor** (smqtk.representation.DescriptorElement) – Descriptor to index.

abstract add_many_descriptors(*descriptors*)
Add multiple descriptors at one time.

Adding the same descriptor multiple times should not add multiple copies of the descriptor in the index (based on UUID). Added descriptors overwrite indexed descriptors based on UUID.

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

abstract clear()
Clear this descriptor index's entries.

abstract count()

Returns Number of descriptor elements stored in this index.

Return type int

abstract get_descriptor(*uuid*)
Get the descriptor in this index that is associated with the given UUID.

Parameters **uuid** (*collections.abc.Hashable*) – UUID of the DescriptorElement to get.

Raises **KeyError** – The given UUID doesn't associate to a DescriptorElement in this index.

Returns DescriptorElement associated with the queried UUID.

Return type *smqtk.representation.DescriptorElement*

abstract get_many_descriptors(*uuids*)
Get an iterator over descriptors associated to given descriptor UUIDs.

Parameters **uuids** (*collections.abc.Iterable[collections.abc.Hashable]*) – Iterable of descriptor UUIDs to query for.

Raises **KeyError** – A given UUID doesn't associate with a DescriptorElement in this index.

Returns Iterator of descriptors associated to given uuid values.

Return type collections.abc.Iterable[*smqtk.representation.DescriptorElement*]

get_many_vectors(*uuids*)
Get underlying vectors of descriptors associated with given uuids.

Parameters **uuids** (*collections.abc.Iterable[collections.abc.Hashable]*) – Iterable of descriptor UUIDs to query for.

Raises KeyError: When there is not a descriptor in this set for one or more input UIDs.

Returns List of vectors for descriptors associated with given uuid values.

Return type list[numpy.ndarray | None]

abstract has_descriptor(*uuid*)
Check if a DescriptorElement with the given UUID exists in this index.

Parameters **uuid** (*collections.abc.Hashable*) – UUID to query for

Returns True if a DescriptorElement with the given UUID exists in this index, or False if not.

Return type bool

items()
alias for iteritems

abstract iterdescriptors()
Return an iterator over indexed descriptor element instances. :rtype: collections.abc.Iterator[smqtk.representation.DescriptorElement]

abstract iteritems()
Return an iterator over indexed descriptor key and instance pairs. :rtype: collections.abc.Iterator[(collections.abc.Hashable,

smqtk.representation.DescriptorElement)]

abstract iterkeys()
Return an iterator over indexed descriptor keys, which are their UUIDs. :rtype: collections.abc.Iterator[collections.abc.Hashable]

keys()
alias for iterkeys

abstract remove_descriptor(*uuid*)
Remove a descriptor from this index by the given UUID.

Parameters **uuid** (*collections.abc.Hashable*) – UUID of the DescriptorElement to remove.

Raises **KeyError** – The given UUID doesn't associate to a DescriptorElement in this index.

abstract remove_many_descriptors(*uuids*)
Remove descriptors associated to given descriptor UUIDs from this index.

Parameters **uuids** (*collections.abc.Iterable[collections.abc.Hashable]*) – Iterable of descriptor UUIDs to remove.

Raises **KeyError** – A given UUID doesn't associate with a DescriptorElement in this index.

DetectionElement

class smqtk.representation.**DetectionElement**(*uuid*)
Representation of a spatial detection.

classmethod from_config(*config_dict*, *uuid*, *merge_default=True*)
Override of *smqtk.utils.configuration.Configurable.from_config()* with the added runtime argument `uuid`. See parent method documentation for details.

Parameters

- **config_dict** (*dict*) – JSON compliant dictionary encapsulating a configuration.
- **uuid** (*collections.abc.Hashable*) – UUID to assign to the produced DetectionElement.
- **merge_default** (*bool*) – Merge the given configuration on top of the default provided by `get_default_config`.

Returns Constructed instance from the provided config.

Return type *DetectionElement*

abstract get_bbox()

Returns The spatial bounding box of this detection.

Return type smqtk.representation.AxisAlignedBoundingBox

Raises **NoDetectionError** – No detection AxisAlignedBoundingBox set yet.

abstract get_classification()

Returns The classification element of this detection.

Return type smqtk.representation.ClassificationElement

Raises **NoDetectionError** – No detection ClassificationElement set yet or the element is empty.

classmethod get_default_config()

Generate and return a default configuration dictionary for this class. This will be primarily used for generating what the configuration dictionary would look like for this class without instantiating it.

By default, we observe what this class's constructor takes as arguments, turning those argument names into configuration dictionary keys. If any of those arguments have defaults, we will add those values into the configuration dictionary appropriately. The dictionary returned should only contain JSON compliant value types.

It is not be guaranteed that the configuration dictionary returned from this method is valid for construction of an instance of this class.

Returns Default configuration dictionary for the class.

Return type dict

```
>>> # noinspection PyUnresolvedReferences
>>> class SimpleConfig(Configurable):
...     def __init__(self, a=1, b='foo'):
...         self.a = a
...         self.b = b
...     def get_config(self):
...         return {'a': self.a, 'b': self.b}
>>> self = SimpleConfig()
>>> config = self.get_default_config()
>>> assert config == {'a': 1, 'b': 'foo'}
```

abstract get_detection()

Returns The paired spatial bounding box and classification element of this detection.

Return type (smqtk.representation.AxisAlignedBoundingBox, smqtk.representation.ClassificationElement)

Raises **NoDetectionError** – No detection AxisAlignedBoundingBox and ClassificationElement set yet.

abstract has_detection()

Returns Whether or not this container currently contains a valid detection bounding box and classification element (must be non-zero).

Return type bool

abstract set_detection(*bbox*, *classification_element*)

Set a bounding box and classification element to this detection element.

Parameters

- **bbox** (*smqtk.representation.AxisAlignedBoundingBox*) – Spatial bounding box instance.
- **classification_element** (*smqtk.representation.ClassificationElement*) – The classification of this detection.

Raises **ValueError** – No, or invalid, AxisAlignedBoundingBox or ClassificationElement was provided.

Returns Self

Return type *DetectionElement*

3.2.2 Data Support Structures

Other data structures are provided in the [smqtk.representation](/python/smqtk/representation) module to assist with the use of the above described structures:

ClassificationElementFactory

class smqtk.representation.**ClassificationElementFactory**(*type*, *type_config*)

Factory class for producing ClassificationElement instances of a specified type and configuration.

classmethod from_config(*config_dict*, *merge_default=True*)

Instantiate a new instance of this class given the configuration JSON-compliant dictionary encapsulating initialization arguments.

This method should not be called via super unless and instance of the class is desired.

Parameters

- **config_dict** (*dict*) – JSON compliant dictionary encapsulating a configuration.
- **merge_default** (*bool*) – Merge the given configuration on top of the default provided by get_default_config.

Returns Constructed instance from the provided config.

Return type *ClassificationElementFactory*

get_config()

Return a JSON-compliant dictionary that could be passed to this class's from_config method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn't make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method's returned dictionary may leave those parameters out. In such cases, the object's from_config class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns JSON type compliant configuration dictionary.

Return type dict

classmethod get_default_config()

Generate and return a default configuration dictionary for this class. This will be primarily used for generating what the configuration dictionary would look like for this class without instantiating it.

It is not be guaranteed that the configuration dictionary returned from this method is valid for construction of an instance of this class.

Returns Default configuration dictionary for the class.

Return type dict

new_classification(*type*, *uuid*)
Create a new ClassificationElement instance of the configured implementation.

Parameters

- **type** (*str*) – Type of classifier. This is usually the name of the classifier that generated this result.
- **uuid** (*collections.abc.Hashable*) – UUID to associate with the classification.

Returns New ClassificationElement instance.

Return type smqtk.representation.ClassificationElement

type

Type type | smqtk.representation.ClassificationElement

DescriptorElementFactory

class smqtk.representation.**DescriptorElementFactory**(*d_type*, *type_config*)
Factory class for producing DescriptorElement instances of a specified type and configuration.

classmethod from_config(*config_dict*, *merge_default=True*)
Instantiate a new instance of this class given the configuration JSON-compliant dictionary encapsulating initialization arguments.

This method should not be called via super unless and instance of the class is desired.

Parameters

- **config_dict** (*dict*) – JSON compliant dictionary encapsulating a configuration.
- **merge_default** (*bool*) – Merge the given configuration on top of the default provided by `get_default_config`.

Returns Constructed instance from the provided config.

Return type *DescriptorElementFactory*

get_config()
Return a JSON-compliant dictionary that could be passed to this class's `from_config` method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn't make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method's returned dictionary may leave those parameters out. In such cases, the object's `from_config` class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns JSON type compliant configuration dictionary.

Return type dict

classmethod get_default_config()
Generate and return a default configuration dictionary for this class. This will be primarily used for generating what the configuration dictionary would look like for this class without instantiating it.

It is not be guaranteed that the configuration dictionary returned from this method is valid for construction of an instance of this class.

Returns Default configuration dictionary for the class.

Return type dict

new_descriptor(*type_str*, *uuid*)
Create a new DescriptorElement instance of the configured implementation

Parameters

- **type_str** (*str*) – Type of descriptor. This is usually the name of the content descriptor that generated this vector.
- **uuid** (*collections.abc.Hashable*) – UUID to associate with the descriptor

Returns New DescriptorElement instance

Return type *smqtk.representation.DescriptorElement*

DetectionElementFactory

class smqtk.representation.**DetectionElementFactory**(*elem_type*, *elem_config*)
Factory class for producing DetectionElement instances of a specified type and configuration.

classmethod from_config(*config_dict*, *merge_default=True*)
Instantiate a new instance of this class given the configuration JSON-compliant dictionary encapsulating initialization arguments.

This base method is adequate without modification when a class's constructor argument types are JSON-compliant. If one or more are not, however, this method then needs to be overridden in order to convert from a JSON-compliant stand-in into the more complex object the constructor requires. It is recommended that when complex types *are* used they also inherit from the `Configurable` in order to hopefully make easier the conversion to and from JSON-compliant stand-ins.

When this method *does* need to be overridden, this usually looks like the following pattern:

```
class MyClass (Configurable):

    @classmethod
    def from_config(cls, config_dict, merge_default=True):
        # Optionally guarantee default values are present in the
        # configuration dictionary.  This statement pairs with the
        # ``merge_default=False`` parameter in the super call.
        # This also in effect shallow copies the given non-dictionary
        # entries of ``config_dict`` due to the merger with the
        # default config.
        if merge_default:
            config_dict = merge_dict(cls.get_default_config(),
                                     config_dict)

        #
        # Perform any overriding here.
        #

        # Create and return an instance using the super method.
        return super(MyClass, cls).from_config(config_dict,
                                               merge_default=False)
```

This method should not be called via super unless an instance of the class is desired.

Parameters

- **config_dict** (*dict*) – JSON compliant dictionary encapsulating a configuration.

- **merge_default** (*bool*) – Merge the given configuration on top of the default provided by `get_default_config`.

Returns Constructed instance from the provided config.

get_config()

Return a JSON-compliant dictionary that could be passed to this class's `from_config` method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn't make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method's returned dictionary may leave those parameters out. In such cases, the object's `from_config` class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns JSON type compliant configuration dictionary.

Return type dict

classmethod get_default_config()

Generate and return a default configuration dictionary for this class. This will be primarily used for generating what the configuration dictionary would look like for this class without instantiating it.

By default, we observe what this class's constructor takes as arguments, turning those argument names into configuration dictionary keys. If any of those arguments have defaults, we will add those values into the configuration dictionary appropriately. The dictionary returned should only contain JSON compliant value types.

It is not be guaranteed that the configuration dictionary returned from this method is valid for construction of an instance of this class.

Returns Default configuration dictionary for the class.

Return type dict

```
>>> # noinspection PyUnresolvedReferences
>>> class SimpleConfig(Configurable):
...     def __init__(self, a=1, b='foo'):
...         self.a = a
...         self.b = b
...     def get_config(self):
...         return {'a': self.a, 'b': self.b}
>>> self = SimpleConfig()
>>> config = self.get_default_config()
>>> assert config == {'a': 1, 'b': 'foo'}
```

new_detection(*uuid*)

Create a new DetectionElement instance o the configured implementation.

Parameters **uuid** (*collections.abc.Hashable*) – UUID to assign the element.

Returns New DetectionElement instance.

Return type *DetectionElement*

3.3 Algorithms

3.3.1 Algorithm Interfaces

class smqtk.algorithms.**SmqtkAlgorithm**
Parent class for all algorithm interfaces.

property name

Returns The name of this class type.

Return type str

Here we list and briefly describe the high level algorithm interfaces which SMQTK provides. There is at least one implementation available for each interface. Some implementations will require additional dependencies that cannot be packaged with SMQTK.

Classifier

This interface represents algorithms that classify `DescriptorElement` instances into discrete labels or label confidences.

class smqtk.algorithms.classifier.**Classifier**
Interface for algorithms that classify input descriptors into discrete labels and/or label confidences.

static _assert_array_dim_consistency(*array_iter*)
Assert that arrays are consistent in dimensionality across iterated arrays.

Currently we only support iterating single dimension vectors. Arrays of more than one dimension (i.e. 2D matries, etc.) will trigger a ValueError.

Includes a short-cut where if the input is a non-object 2D ndarray, dimensionality must already be consistent, so the ndarray (which is an Iterable) is just returned. Otherwise, we return a generator that checked dimensionality of the input iterable during iteration.

Parameters **| np.ndarray array_iter** (*collections.abc.Iterable[numpy.ndarray]*) – Iterable numpy arrays.

Raises

- **AttributeError** – Individual arrays are not numpy.ndarray-like.
- **ValueError** – Not all input arrays were of consistent dimensionality.

Returns Iterable of the same arrays in the same order, but validated to be of common dimensionality.

abstract _classify_arrays(*array_iter*)
Overridable method for classifying an iterable of descriptor elements whose vectors should be classified.

At this level, all input arrays are guaranteed to be of consistent dimensionality.

Each classification mapping should contain confidence values for each label the configured model contains. Implementations may act in a discrete manner whereby only one label is marked with a `1` value (others being `0`), or in a continuous manner whereby each label is given a confidence-like value in the [0, 1] range.

Parameters **array_iter** (*collections.abc.Iterable[numpy.ndarray]*) – Iterable of arrays to be classified.

Returns Iterable of dictionaries, parallel in association to the input descriptor vectors. Each dictionary should map labels to associated confidence values.

Return type collections.abc.Iterable[dict[collections.abc.Hashable, float]]

classify_arrays(*array_iter*)

Classify an input iterable of numpy arrays into a parallel iterable of label-to-confidence mappings (dictionaries).

Each classification mapping should contain confidence values for each label the configured model contains. Implementations may act in a discrete manner whereby only one label is marked with a `1` value (others being `0`), or in a continuous manner whereby each label is given a confidence-like value in the [0, 1] range.

Parameters | **np.ndarray array_iter** (*collections.abc.Iterable[numpy.ndarray]*) – Iterable of descriptor vectors, as numpy arrays, to be classified.

Raises **ValueError** – Input arrays were not all of consistent dimensionality.

Returns Iterable of dictionaries, parallel in association to the input descriptor vectors. Each dictionary should map labels to associated confidence values.

Return type collections.abc.Iterable[dict[collections.abc.Hashable, float]]

classify_elements(*descr_iter*, *factory=<smqtk.representation.classification_element_factory.ClassificationElementFactor object>*, *overwrite=False*, *d_elem_batch=100*)

Classify an input iterable of descriptor elements into a parallel iterable of classification elements.

Classification element UIDs are inherited from the descriptor element it was generated from.

We invoke `classify_arrays` for actual generation of classification results. See documentation for this method for further details. # We invoke `classify_arrays` for factory-generated classification # elements that do not yet have classifications stored, or on all input # descriptor elements if the `overwrite` flag is True.

Selective Iteration For situations when it is desired to access specific generator returns, like when only one descriptor element is provided in order to get a single element out, it is strongly recommended to expand the returned generator into a sequence type first. For example, expanding out the generator's returns into a list (`list(g.generate_elements([e]))[0]`) is recommended over just getting the "next" element of the returned generator (`next(g.generate_elements([e]))`). Expansion into a sequence allows the generator to fully execute, which includes any functionality after the final `yield` statement in any of the underlying iterators that may perform required clean-up.

Non-redundant Processing Certain classification element implementations, as dictated by the input factory, may be connected to persistent storage in the background. Because of this, some classification elements may already "have" classification results on construction. This method, by default, only computes new classification results for descriptor elements whose associated classification element does not report as already containing results. If the `overwrite` flag is True then classifications are computed for all input descriptor elements and results are set to their respective classification elements regardless of existing result storage.

Parameters

- **descr_iter** (*collections.abc.Iterable[DescriptorElement]*) – Iterable of DescriptorElement instances to be classified.
- **factory** (smqtk.representation.ClassificationElementFactory) – Classification element factory. The default factory yields MemoryClassificationElement instances.
- **overwrite** (*bool*) – Recompute classification of the input descriptor and set the results to the ClassificationElement produced by the factory.
- **d_elem_batch** (*int*) – The number of descriptor elements to collect before requesting the whole batch's vectors at once via `DescriptorElement.get_many_vectors` method.

Raises

- **ValueError** – Either: (A) one or more input descriptor elements did not have a stored vector, or (B) input descriptor element arrays were not all of consistent dimensionality.
- **IndexError** – Implementation of `_classify_arrays` either under or over produced classifications relative to the number of input descriptor vectors.

Returns Iterator of result ClassificationElement instances. UUIDs of generated ClassificationElement instances will reflect the UUID of the DescriptorElement it was computed from.

Return type collections.abc.Iterator[smqtk.representation.ClassificationElement]

classify_one_element(*descr_elem*, *factory=<smqtk.representation.classification_element_factory.ClassificationElement object>*, *overwrite=False*)

Convenience method around `classify_elements` for the single-input case.

See documentation for the *Classifier.classify_elements()* method for more information.

Parameters

- **descr_elem** (DescriptorElement) – Iterable of DescriptorElement instances to be classified.
- **factory** (smqtk.representation.ClassificationElementFactory) – Classification element factory. The default factory yields MemoryClassificationElement instances.
- **overwrite** (*bool*) – Recompute classification of the input descriptor and set the results to the ClassificationElement produced by the factory.

Raises

- **ValueError** – The input descriptor element did not have a stored vector.
- **IndexError** – Implementation of `_classify_arrays` either under or over produced classifications relative to the number of input descriptor vectors.

Returns ClassificationElement instances. UUIDs of the generated ClassificationElement instance will reflect the UUID of the DescriptorElement it was computed from.

Return type smqtk.representation.ClassificationElement

abstract get_labels()

Get the sequence of class labels that this classifier can classify descriptors into. This includes the negative or background label if the classifier embodies such a concept.

Returns Sequence of possible classifier labels.

Return type collections.abc.Sequence[collections.abc.Hashable]

Raises **RuntimeError** – No model loaded.

DescriptorGenerator

This interface represents algorithms that generate whole-content descriptor vectors for one or more given input *DataElement* instances. The input *DataElement* instances must be of a content type that the *DescriptorGenerator* supports, referenced against the `valid_content_types()` method (required by the `ContentTypeValidator` mixin class).

The *DescriptorGenerator.generate_elements()* method also requires a *DescriptorElementFactory* instance to tell the algorithm how to generate the *DescriptorElement* instances it should return. The returned *DescriptorElement* instances will have a type equal to the name of

the *DescriptorGenerator* class that generated it, and a UUID that is the same as the input *DataElement* instance.

If a *DescriptorElement* implementation that supports persistent storage is generated, and there is already a descriptor associated with the given type name and UUID values, the descriptor is returned without re-computation.

If the `overwrite` parameter is `True`, the *DescriptorGenerator* instance will re-compute a descriptor for the input *DataElement*, setting it to the generated *DescriptorElement*. This will overwrite descriptor data in persistent storage if the *DescriptorElement* type used supports it.

class smqtk.algorithms.descriptor_generator.**DescriptorGenerator**

Base abstract Feature Descriptor interface.

generate_arrays(*data_iter*)

Generate descriptor vector elements for **all** input data elements.

Descriptor arrays yielded out will be parallel in association with the data elements input.

Selective Iteration For situations when it is desired to access specific generator returns, like when only one data element is provided in order to get a single array out, it is strongly recommended to expand the returned generator into a sequence type first. For example, expanding out the generator's returns into a list (`list(g.generate_arrays([e]))[0]`) is recommended over just getting the "next" element of the returned generator (`next(g.generate_arrays([e]))`). Expansion into a sequence allows the generator to fully execute, which includes any functionality after the final `yield` statement in any of the underlying iterators.

Parameters **data_iter** (*collections.abc.Iterable[smqtk.representation.DataElement]*) – Iterable of DataElement instances to be described.

Raises

- **RuntimeError** – Descriptor extraction failure of some kind.
- **ValueError** – Given data element content was not of a valid type with respect to this descriptor generator implementation.

Returns Iterator of result numpy.ndarray instances.

Return type collections.abc.Iterator[numpy.ndarray]

generate_elements(*data_iter*, *descr_factory=<smqtk.representation.descriptor_element_factory.DescriptorElementFactor object>*, *overwrite=False*)

Generate DescriptorElement instances for the input data elements, generating new descriptors for those elements that need them, or optionally all input data elements.

Descriptor elements yielded out will be parallel in association with the data elements input. Descriptor element UUIDs are inherited from the data element it was generated from.

Selective Iteration For situations when it is desired to access specific generator returns, like when only one data element is provided in order to get a single element out, it is strongly recommended to expand the returned generator into a sequence type first. For example, expanding out the generator's returns into a list (`list(g.generate_elements([e]))[0]`) is recommended over just getting the "next" element of the returned generator (`next(g.generate_elements([e]))`). Expansion into a sequence allows the generator to fully execute, which includes any functionality after the final `yield` statement in any of the underlying iterators that may perform required clean-up.

Non-redundant Processing Certain descriptor element implementations, as dictated by the input factory, may be connected to persistent storage in the background. Because of this, some descriptor elements may already "have" a vector on construction. This method, by default, only computes new descriptor vectors for data elements whose associated descriptor element does not report as already containing a vector. If

the `overwrite` flag is True then descriptors are computed for all input data elements and are set to their respective descriptor elements regardless of existing vector storage.

Parameters

- **data_iter** (*collections.abc.Iterable*[smqtk.representation.DataElement]) – Iterable of DataElement instances to be described.
- **descr_factory** (smqtk.representation.DescriptorElementFactory) – DescriptorElementFactory instance to drive the generation of element instances with some parametrization.
- **overwrite** (*bool*) – By default, if a factory-produced DescriptorElement reports as containing a vector, we do not compute a descriptor again for the associated data element. If this is `True`, however, we will generate descriptors for all input data elements, overwriting the vectors previously stored in the factory-produces descriptor elements.

Raises

- **RuntimeError** – Descriptor extraction failure of some kind.
- **ValueError** – Given data element content was not of a valid type with respect to this descriptor generator implementation.
- **IndexError** – Underlying vector-producing generator either under or over produced vectors.

Returns Iterator of result DescriptorElement instances. UUIDs of generated DescriptorElement instances will reflect the UUID of the DataElement it was generated from.

Return type collections.abc.Iterator[*smqtk.representation.DescriptorElement*]

generate_one_array(*data_elem*)

Convenience wrapper around `generate_arrays` for the single-input case.

See the documentation for the *DescriptorGenerator.generate_arrays()* method for more information.

Parameters **data_elem** (smqtk.representation.DataElement) – DataElement instance to be described.

Raises

- **RuntimeError** – Descriptor extraction failure of some kind.
- **ValueError** – Given data element content was not of a valid type with respect to this descriptor generator implementation.

Returns Descriptor vector the given data as a `numpy.ndarray` instance.

Return type numpy.ndarray

generate_one_element(*data_elem*, *descr_factory=<smqtk.representation.descriptor_element_factory.DescriptorElement object>*, *overwrite=False*)

Convenience wrapper around `generate_elements` for the single-input case.

See documentation for the *DescriptorGenerator.generate_elements()* method for more information

Parameters

- **data_elem** (smqtk.representation.DataElement) – DataElement instance to be described.

- **descr_factory** (smqtk.representation.DescriptorElementFactory) – DescriptorElementFactory instance to drive the generation of element instances with some parametrization.
- **overwrite** (*bool*) – By default, if a factory-produced DescriptorElement reports as containing a vector, we do not compute a descriptor again for the associated data element. If this is `True`, however, we will generate descriptors for all input data elements, overwriting the vectors previously stored in the factory-produces descriptor elements.

Raises

- **IndexError** – Underlying vector-producing generator either under or over produced vectors.
- **RuntimeError** – Descriptor extraction failure of some kind.
- **ValueError** – Given data element content was not of a valid type with respect to this descriptor generator implementation.

Returns Result DescriptorElement instance. UUID of the generated DescriptorElement instance will reflect the UUID of the DataElement it was generated from.

Return type *smqtk.representation.DescriptorElement*

ImageReader

class smqtk.algorithms.image_io.**ImageReader**

Interface for algorithms that load a raster image matrix from a data element.

is_valid_element(*data_element*)

Check if the given DataElement instance reports a content type that matches one of the MIME types reported by `valid_content_types`.

This override checks if the `DataElement` has the `matrix` property as the `MatrixDataElement` would provide, and that its value of an expected type.

Parameters **data_element** (smqtk.representation.DataElement) – Data element instance to check.

Returns True if the given element has a valid content type as reported by `valid_content_types`, and False if not.

Return type bool

load_as_matrix(*data_element*, *pixel_crop=None*)

Load an image matrix from the given data element.

Matrix Property Shortcut. If the given DataElement instance defines a `matrix` property this method simply returns that. This is intended to interface with instances of `smqtk.representation.data_element.matrix.MatrixDataElement`.

Loading From Bytes. When not loading from a short-cut matrix, matrix return format is `ImageReader` implementation dependant. Implementations of this interface should specify and describe their return type.

Aside from the exceptions documented below, other exceptions may be raised when an image fails to load that are implementation dependent.

Parameters

- **data_element** (smqtk.representation.DataElement) – DataElement to load image data from.

- **pixel_crop** (*None|smqtk.representation.AxisAlignedBoundingBox*) – Optional bounding box specifying a pixel sub-region to load from the given data. If this is provided it must represent a valid sub-region within the loaded image, otherwise a RuntimeError is raised. Handling of non-integer aligned boxes are implementation dependant.

Raises

- **RuntimeError** – A crop region was specified but did not specify a valid sub-region of the image.
- **AssertionError** – The `data_element` provided defined a `matrix` attribute/property, but its access did not result in an expected value.
- **ValueError** –

  This error is raised when:

  - The given `data_element` was not of a valid content type.
  - A `pixel_crop` bounding box was provided but was zero volume.
  - `pixel_crop` bounding box vertices are not fully represented by integers.

Returns Numpy ndarray of the image data. Specific return format is implementation dependant.

Return type numpy.ndarray

class `smqtk.algorithms.image_io.pil_io.`**PilImageReader**(*explicit_mode=None*)

Image reader that uses PIL to load the image.

This implementation may additionally raise an `IOError` when failing to to load an image.

get_config()

Return a JSON-compliant dictionary that could be passed to this class's `from_config` method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn't make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method's returned dictionary may leave those parameters out. In such cases, the object's `from_config` class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns JSON type compliant configuration dictionary.

Return type dict

classmethod is_usable()

Check whether this class is available for use.

Since certain plugin implementations may require additional dependencies that may not yet be available on the system, or other runtime conditions, this method may be overridden to check for those and return a boolean saying if the implementation is available for usable. When this method returns *True*, the class is declaring that it should be constructable and usable in the current environment.

By default, this method will return True unless a sub-class overrides this class-method with their specific logic.

NOTES:

- This should be a class method
- **When an implementation is deemed not usable, this should emit a** (user) warning, or some other kind of logging, detailing why the implementation is not available for use.

Returns Boolean determination of whether this implementation is usable in the current environment.

Return type bool

valid_content_types()

Returns A set valid MIME types that are "valid" within the implementing class' context.

Return type set[str]

HashIndex

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

Unlike the `NearestNeighborsIndex` interface from which this interface descends, `HashIndex` instances are build with an iterable of `numpy.ndarray` and `nn` returns a `numpy.ndarray`.

class smqtk.algorithms.nn_index.hash_index.**HashIndex**

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*)

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** (*collections.abc.Iterable[numpy.ndarray[bool]]*) – Iterable of descriptor elements to build index over.

abstract count()

Returns Number of elements in this index.

Return type int

nn(*h*, *n=1*)

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** (*numpy.ndarray[bool]*) – Hash code to compute the neighbors of. Should be the same bit length as indexed hash codes.
- **n** (*int*) – Number of nearest neighbors to find.

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

Return type (tuple[numpy.ndarray[bool]], tuple[float])

remove_from_index(*hashes*)
Partially remove hashes from this index.

Parameters **hashes** (*collections.abc.Iterable[numpy.ndarray[bool]]*) – 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*)
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** (*collections.abc.Iterable[numpy.ndarray[bool]]*) – Iterable of numpy boolean hash vectors to add to this index.

LshFunctor

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

class smqtk.algorithms.nn_index.lsh.functors.**LshFunctor**
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*)
Get the locality-sensitive hash code for the input descriptor.

Parameters **descriptor** (*numpy.ndarray[float]*) – Descriptor vector we should generate the hash of.

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

Return type numpy.ndarray[bool]

NearestNeighborsIndex

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

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

The `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 `count` method, which returns the number of distinct `DescriptorElement` instances are in the index.

class smqtk.algorithms.nn_index.**NearestNeighborsIndex**

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*)

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()

Returns Number of elements in this index.

Return type int

nn(*d*, *n=1*)

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** (*smqtk.representation.DescriptorElement*) – Descriptor element to compute the neighbors of.
- **n** (*int*) – Number of nearest neighbors to find.

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

Return type (tuple[*smqtk.representation.DescriptorElement*], tuple[float])

remove_from_index(*uids*)
Partially remove descriptors from this index associated with the given UIDs.

Parameters **uids** (*collections.abc.Iterable[collections.abc.Hashable]*) – 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*)
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.

ObjectDetector

This interface defines a method to generate object detections (*DetectionElement*) over a given *DataElement*.

class smqtk.algorithms.object_detection.**ObjectDetector**
Abstract interface to an object detection algorithm.

An object detection algorithm is one that can take in data and output zero or more detection elements, where each detection represents a spatial region in the data.

This high level interface only requires detection element returns (spatial bounding-boxes with associated classification elements).

detect_objects(*data_element*, *de_factory=<smqtk.representation.detection_element_factory.DetectionElementFactory object>*, *ce_factory=<smqtk.representation.classification_element_factory.ClassificationElementFactory object>*)
Detect objects in the given data.

UUIDs of detections are based on the hash produced from the combination of:

- Detection bounding-box bounding coordinates
- Classification label set predicted for a bounding box.

Parameters

- **data_element** (smqtk.representation.DataElement) – Source data from which to detect objects within.
- **de_factory** (smqtk.representation.DetectionElementFactory) – Factory for generating DetectionElement instances. The default factory yields MemoryClassificationElement instances.
- **ce_factory** (smqtk.representation.ClassificationElementFactory) – Factory for generating ClassificationElement instances for detections. The default factory yields MemoryClassificationElement instances.

Raises **ValueError** – Given data element content was not of a valid content type that this class reports as valid for object detection.

Returns Iterator over result DetectionElement instances as generated by the given DetectionElementFactory, containing classification elements as generated by the given ClassificationElementFactory.

Return type collections.abc.Iterable[*smqtk.representation.DetectionElement*]

RankRelevancy

This interface defines one method: `rank`. The `rank` method takes examples of relevant and not-relevant example descriptor vectors as `numpy.ndarray` sequences and uses them to compute relevancy scores (on a `[0, 1]` scale) on a provided pool of other descriptor vectors.

class smqtk.algorithms.rank_relevancy.**RankRelevancy**

Algorithm that can rank a given pool of descriptors based on positively and negatively adjudicated descriptors.

abstract rank(*pos: Sequence[numpy.ndarray]*, *neg: Sequence[numpy.ndarray]*, *pool: Sequence[numpy.ndarray]*) → Sequence[float]

Assign a relevancy score to each input descriptor in *pool* based on the positively and negatively adjudicated descriptors in *pos* and *neg* respectively.

Parameters

- **pos** – Sequence of positively adjudicated descriptor vectors.
- **neg** – Sequence of negatively adjudicated descriptor vectors.
- **pool** – A sequence of descriptor vectors that we want to rank by topical relevancy relative to the given positive and negative examples.

Returns An ordered sequence of float values denoting the relevancy of *pool* elements

RankRelevancyWithFeedback

This interface defines one method: `rank_with_feedback`. Like `RankRelevancy.rank()`, `rank_with_feedback` takes examples of relevant and not-relevant example descriptor vectors as `numpy.ndarray` sequences and uses them to compute relevancy scores (on a `[0, 1]` scale) on a provided pool of other descriptor vectors. However, it also expects a sequence of corresponding UIDs for the pool vectors and additionally returns a sequence of UIDs, possibly not all from the pool, on which feedback would be most useful.

class smqtk.algorithms.rank_relevancy.**RankRelevancyWithFeedback**

Similar to the *RankRelevancy* algorithm but with the added feature of also returning a sequence of elements from which feedback would be "most useful".

What "most useful" means may be flexible but generally refers to the goal of reducing the amount of adjudications required in order to separate true-positive examples from true-negative examples in provided pools via the assigned relevancy scores. E.g. other elements may be adjudicated in some quantity to achieve some level of relevant sample separation, but if the feedback requests are instead adjudicated, less elements may need to be adjudicated to achieve and equivalent level of separation.

Feedback requests ought to be returned in a form that is meaningful for the user to be able to properly convey the proper information to the adjudicating agent to actually perform adjudications. Additionally, we want to be able to request feedback from elements that may not be present in the given pool of descriptors.

Towards that end, this algorithm should be given a sequence of UIDs for the given pool of descriptors. This allows the implementation to potentially coordinate with an outside source of descriptor references such that the returned feedback requests may be interpreted uniformly.

abstract _rank_with_feedback(*pos: Sequence[numpy.ndarray]*, *neg: Sequence[numpy.ndarray]*, *pool: Sequence[numpy.ndarray]*, *pool_uids: Sequence[collections.abc.Hashable]*) → Tuple[Sequence[float], Sequence[collections.abc.Hashable]]

Implement *rank_with_feedback()*. *pool* and *pool_uids* have already been checked to be of equal length.

See also:

rank_with_feedback()'s doc-string for the meanings of the parameters and their return values

rank_with_feedback(*pos: Sequence[numpy.ndarray]*, *neg: Sequence[numpy.ndarray]*, *pool: Sequence[numpy.ndarray]*, *pool_uids: Sequence[collections.abc.Hashable]*) → Tuple[Sequence[float], Sequence[collections.abc.Hashable]]

Assign a relevancy score to each input descriptor in *pool* based on the positively and negatively adjudicated descriptors in *pos* and *neg* respectively, additionally returning a sequence of UIDs of those descriptors for which adjudication feedback would be "most useful".

Parameters

- **pos** – Sequence of positively adjudicated descriptor vectors.
- **neg** – Sequence of negatively adjudicated descriptor vectors.
- **pool** – A sequence of descriptor vectors that we want to rank by topical relevancy relative to the given positive and negative examples.
- **pool_uids** – A sequence of hashable UID values, parallel in association with descriptors in *pool*.

Returns Ordered sequence of float values denoting relevancy of *pool* elements, as well as a sequence of `Hashable` values referencing in-pool or out-of-pool descriptors we recommend for adjudication feedback. In the latter sequence, descriptors are ordered by usefulness, most to least.

Raises **ValueError** – *pool* and *pool_uids* are of different length

See also:

RankRelevancyWithFeedback class doc-string for discussion on "most useful" meaning.

RelevancyIndex

This interface defines two methods: `build_index` and `rank`. The `build_index` method is, like a `NearestNeighborsIndex`, used to build an index of `DescriptorElement` instances. The `rank` method takes examples of relevant and not-relevant `DescriptorElement` examples with which the algorithm uses to rank (think sort) the indexed `DescriptorElement` instances by relevancy (on a `[0, 1]` scale).

class smqtk.algorithms.relevancy_index.**RelevancyIndex**

Abstract class for IQR index implementations.

Similar to a traditional nearest-neighbors algorithm, An IQR index provides a specialized nearest-neighbors interface that can take multiple examples of positively and negatively relevant exemplars in order to produce a [0, 1] ranking of the indexed elements by determined relevancy.

abstract build_index(*descriptors*)

Build the index based on the given iterable of descriptor elements.

Subsequent calls to this method should rebuild the index, not add to it.

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()

Returns Number of elements in this index.

Return type int

abstract rank(*pos*, *neg*)

Rank the currently indexed elements given `pos` positive and `neg` negative exemplar descriptor elements.

Parameters

- **pos** (*collections.abc.Iterable[smqtk.representation.DescriptorElement]*) – Iterable of positive exemplar DescriptorElement instances. This may be optional for some implementations.
- **neg** (*collections.abc.Iterable[smqtk.representation.DescriptorElement]*) – Iterable of negative exemplar DescriptorElement instances. This may be optional for some implementations.

Raises **NoIndexError** – If index ranking is requested without an index to rank.

Returns Map of indexed descriptor elements to a rank value between [0, 1] (inclusive) range, where a 1.0 means most relevant and 0.0 meaning least relevant.

Return type dict[*smqtk.representation.DescriptorElement*, float]

3.3.2 Algorithm Models and Generation

Some algorithms require a model, of a pre-existing computed state, to function correctly. Not all algorithm interfaces require that there is some model generation method as it is as times not appropriate or applicable to the definition of the algorithm the interface is for. However some implementations of algorithms desire a model for some or all of its functionality. Algorithm implementations that require extra modeling are responsible for providing a method or utility for generating algorithm specific models. Some algorithm implementations may also be pre-packaged with one or more specific models to optionally choose from, due to some performance, tuning or feasibility constraint. Explanations about whether an extra model is required and how it is constructed should be detailed by the documentation for that specific implementation.

For example, part of the definition of a `NearestNeighborsIndex` algorithm is that there is an index to search over, which is arguably a model for that algorithm. Thus, the `build_index()` method, which should build the index model, is part of that algorithm's interface. Other algorithms, like the `DescriptorGenerator` class of algorithms, do not have a high-level model building method, and model generation or choice is left to specific implementations to explain or provide.

DescriptorGenerator Models

The `DescriptorGenerator` interface does not define a model building method, but some implementations require internal models. Below are explanations on how to build or get modes for `DescriptorGenerator` implementations that require a model.

ColorDescriptor

ColorDescriptor implementations need to build a visual bag-of-words codebook model for reducing the dimensionality of the many low-level descriptors detected in an input data element. Model parameters as well as storage location parameters are specified at instance construction time, or via a configuration dictionary given to the `from_config` class method.

The storage location parameters include a data model directory path and an intermediate data working directory path: `model_directory` and `work_directory` respectively. The `model_directory` should be the path to a directory for storage of generated model elements. The `work_directory` should be the path to a directory to store cached intermediate data. If model elements already exist in the provided `model_directory`, they are loaded at construction time. Otherwise, the provided directory is used to store model components when the `generate_model` method is called. Please reference the constructor's doc-string for the description of other constructor parameters.

The method `generate_model(data_set)` is provided on instances, which should be given an iterable of `DataElement` instances representing media that should be used for training the visual bag-of-words codebook. Media content types that are supported by `DescriptorGenerator` instances is listed via the `valid_content_types()` method.

Below is an example code snippet of how to train a ColorDescriptor model for some instance of a ColorDescriptor implementation class and configuration:

```
# Fill in "<flavor>" with a specific ColorDescriptor class.
cd = ColorDescriptor_<flavor>(model_directory="data", work_directory="work")

# Assuming there is not model generated, the following call would fail due to
# there not being a model loaded
# cd.generate_one_element(some_data, some_factory)

data_elements = [...]  # Some iterable of DataElement instances to media content
# Generates model components
cd.generate_model(data_elements)

# Example of a new instance, given the same parameters, that will load the
# existing model files in the provided ``model_directory``.
new_cd = ColorDescriptor_<flavor>(model_directory="data", work_directory="work")

# Since there is a model, we can now generate descriptors for new data
new_cd.generate_one_element(new_data, some_factory)
```

CaffeDefaultImageNet

This implementation does not come with a method of training its own models, but requires model files provided by Caffe: the network model file and the image mean binary protobuf file.

The Caffe source tree provides two scripts to download the specific files (relative to the caffe source tree):

```
# Downloads the network model file
scripts/download_model_binary.py models/bvlc_reference_caffenet

# Downloads the ImageNet mean image binary protobuf file
data/ilsvrc12/get_ilsvrc_aux.sh
```

These script effectively just download files from a specific source.

If the Caffe source tree is not available, the model files can be downloaded from the following URLs:

- Network model: http://dl.caffe.berkeleyvision.org/bvlc_reference_caffenet.caffemodel
- Image mean: http://dl.caffe.berkeleyvision.org/caffe_ilsvrc12.tar.gz

NearestNeighborsIndex Models (k nearest-neighbors)

`NearestNeighborsIndex` interfaced classes include a `build_index` method on instances that should build the index model for an implementation. Implementations, if they allow for persistant storage, should take relevant parameters at construction time. Currently, we do not package an implementation that require additional model creation.

The general pattern for `NearestNeighborsIndex` instance index model generation:

```
descriptors = [...]  # some number of descriptors to index

index = NearestNeighborsIndexImpl(...)
# Calling ``nn`` should fail before an index has been built.

index.build_index(descriptors)

q = DescriptorElementImpl(...)
neighbors, dists = index.nn(q)
```

RelevancyIndex Models

`RelevancyIndex` interfaced classes include a `build_index` method in instances that should build the index model for a particular implementation. Implementations, if they allow for persistant storage, should take relevant parameters at construction time. Currently, we do not package an implementation that requires additional model creation.

The general pattern for `RelevancyIndex` instance index model generation:

```
descriptors = [...]  # some number of descriptors to index

index = RelevancyIndexImpl(...)
# Calling ``rank`` should fail before an index has been built.

index.build_index(descriptors)

rank_map = index.rank(pos_descriptors, neg_descriptors)
```

3.4 Web Service and Demonstration Applications

Included in SMQTK are a few web-based service and demonstration applications, providing a view into the functionality provided by SMQTK algorithms and utilities.

3.4.1 runApplication

This script can be used to run any conforming (derived from *SmqtkWebApp*) SMQTK web based application. Web services should be runnable via the `bin/runApplication.py` script.

Runs conforming SMQTK Web Applications.

```
usage: runApplication [-h] [-v] [-c PATH] [-g PATH] [-l] [-a APPLICATION] [-r]
                      [-t] [--host HOST] [--port PORT] [--use-basic-auth]
                      [--use-simple-cors] [--debug-server] [--debug-smqtk]
                      [--debug-app] [--debug-ns DEBUG_NS]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

Application Selection

-l, --list List currently available applications for running. More description is included if SMQTK verbosity is increased (-v | –debug-smqtk)

Default: False

-a, --application Label of the web application to run.

Server options

-r, --reload Turn on server reloading.

Default: False

-t, --threaded Turn on server multi-threading.

Default: False

--host Run host address specification override. This will override all other configuration method specifications.

--port Run port specification override. This will override all other configuration method specifications.

--use-basic-auth Use global basic authentication as configured.

Default: False

--use-simple-cors Allow CORS for all domains on all routes. This follows the "Simple Usage" of flask-cors: https://flask-cors.readthedocs.io/en/latest/#simple-usage

Default: False

Other options

--debug-server Turn on server debugging messages ONLY. This is implied when -v|–verbose is enabled.

Default: False

--debug-smqtk Turn on SMQTK debugging messages ONLY. This is implied when -v|–verbose is enabled.

Default: False

--debug-app Turn on flask app logger namespace debugging messages ONLY. This is effectively enabled if the flask app is provided with SMQTK and "–debug-smqtk" is passed. This is also implied if -v|–verbose is enabled.

Default: False

--debug-ns Specify additional python module namespaces to enable debug logging for.

Default: []

3.4.2 SmqtkWebApp

This is the base class for all web applications and services in SMQTK.

class smqtk.web.**SmqtkWebApp**(*json_config*)

Base class for SMQTK web applications

classmethod from_config(*config_dict*, *merge_default=True*)

Override to just pass the configuration dictionary to constructor

get_config()

Return a JSON-compliant dictionary that could be passed to this class's `from_config` method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn't make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method's returned dictionary may leave those parameters out. In such cases, the object's `from_config` class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns JSON type compliant configuration dictionary.

Return type dict

classmethod get_default_config()

Generate and return a default configuration dictionary for this class. This will be primarily used for generating what the configuration dictionary would look like for this class without instantiating it.

This should be overridden in each implemented application class to add appropriate configuration.

Returns Default configuration dictionary for the class.

Return type dict

classmethod impl_directory()

Returns Directory in which this implementation is contained.

Return type str

run(*host=None*, *port=None*, *debug=False*, ***options*)
Override of the run method, drawing running host and port from configuration by default. 'host' and 'port' values specified as argument or keyword will override the app configuration.

3.4.3 Sample Web Applications

Descriptor Similarity Service

- Provides a web-accessible API for computing content descriptor vectors for available descriptor generator labels.
- Descriptor generators that are available to the service are based on the a configuration file provided to the server.

class smqtk.web.descriptor_service.**DescriptorServiceServer**(*json_config*)
Simple server that takes in a specification of the following form:

/<descriptor_type>/<uri>[?...]

See the docstring for the `DescriptorServiceServer.compute_descriptor()` method for complete rules on how to form a calling URL.

Computes the requested descriptor for the given file and returns that via a JSON structure.

Standard return JSON:

```
{
    "success": <bool>,
    "descriptor": [ <float>, ... ]
    "message": <string>,
    "reference_uri": <uri>
}
```

Additional Configuration

Note: We will look for an environment variable *DescriptorService_CONFIG* for a string file path to an additional JSON configuration file to consider.

---

generate_descriptor(*de*, *cd_label*)
Generate a descriptor for the content pointed to by the given URI using the specified descriptor generator.

Raises

- **ValueError** – Content type mismatch given the descriptor generator
- **RuntimeError** – Descriptor extraction failure.

Returns Generated descriptor element instance with vector information.

Return type *smqtk.representation.DescriptorElement*

generator_label_configs

Type dict[str, dict]

get_config()
Return a JSON-compliant dictionary that could be passed to this class's `from_config` method to produce an instance with identical configuration.

In the most cases, this involves naming the keys of the dictionary based on the initialization argument names as if it were to be passed to the constructor via dictionary expansion. In some cases, where it doesn't make sense to store some object constructor parameters are expected to be supplied at as configuration values (i.e. must be supplied at runtime), this method's returned dictionary may leave those parameters out. In such cases, the object's `from_config` class-method would also take additional positional arguments to fill in for the parameters that this returned configuration lacks.

Returns JSON type compliant configuration dictionary.

Return type dict

classmethod get_default_config()
Generate and return a default configuration dictionary for this class. This will be primarily used for generating what the configuration dictionary would look like for this class without instantiating it.

Returns Default configuration dictionary for the class.

Return type dict

get_descriptor_inst(*label*)
Get the cached content descriptor instance for a configuration label :type label: str :rtype: smqtk.algorithms.descriptor_generator.DescriptorGenerator

classmethod is_usable()
Check whether this class is available for use.

Since certain plugin implementations may require additional dependencies that may not yet be available on the system, or other runtime conditions, this method may be overridden to check for those and return a boolean saying if the implementation is available for usable. When this method returns *True*, the class is declaring that it should be constructable and usable in the current environment.

By default, this method will return True unless a sub-class overrides this class-method with their specific logic.

NOTES:

- This should be a class method
- **When an implementation is deemed not usable, this should emit a** (user) warning, or some other kind of logging, detailing why the implementation is not available for use.

Returns Boolean determination of whether this implementation is usable in the current environment.

Return type bool

resolve_data_element(*uri*)
Given the URI to some data, resolve it down to a DataElement instance.

Raises **ValueError** – Issue with the given URI regarding either URI source resolution or data resolution.

Parameters **uri** (*str*) – URI to data

Returns DataElement instance wrapping given URI to data.

Return type *smqtk.representation.DataElement*

IQR Demo Application

Interactive Query Refinement or "IQR" is a process whereby a user provides one or more exemplar images and the system attempts to locate additional images from within an archive that a similar to the exemplar(s). The user then adjudicates the results by identifying those results that match their search and those results that do not. The system then uses those adjudications to attempt to provide better, more closely matching results refined by the user's input.

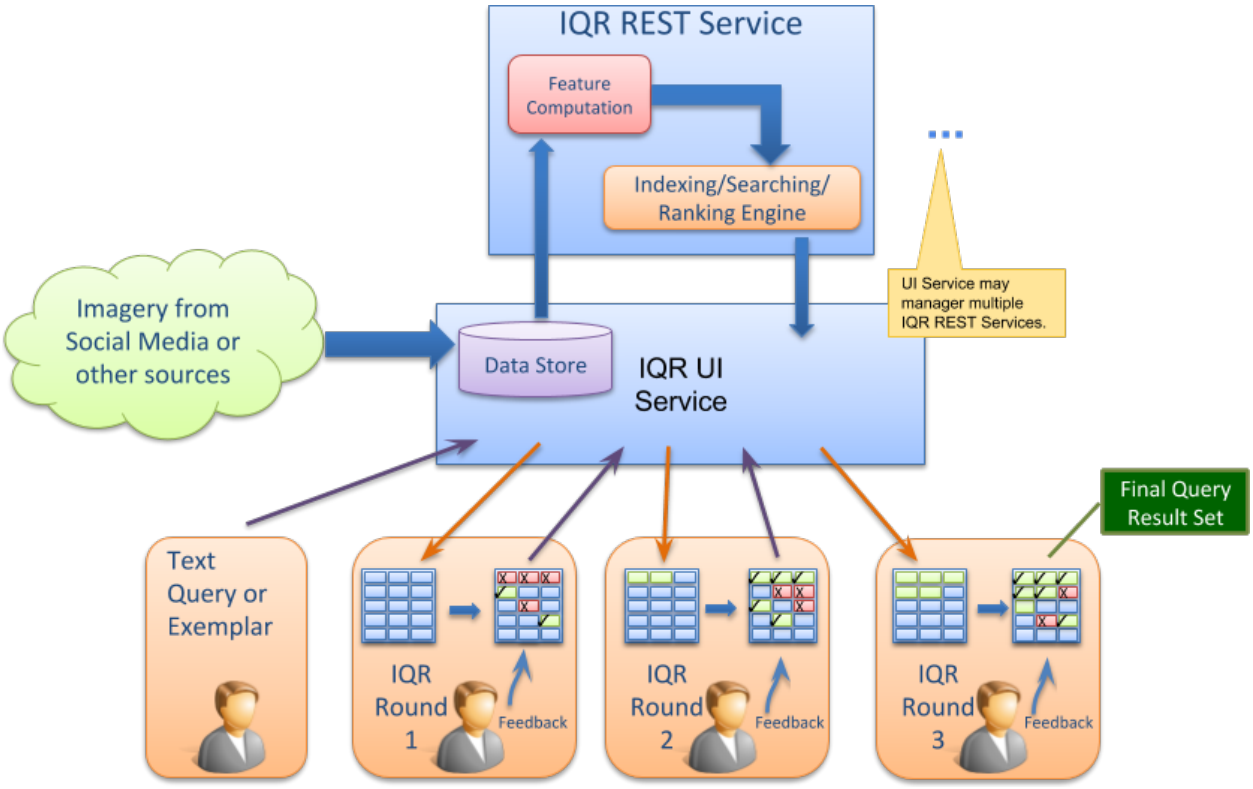

Fig. 1: **SMQTK IQR Workflow**
Overall workflow of an SMQTK based Interactive Query Refinement application.

The IQR application is an excellent example application for SMQTK as it makes use of a broad spectrum of SMQTK's capabilities. In order to characterize each image in the archive so that it can be indexed, the `DescriptorGenerator` algorithm is used. A `NearestNeighborsIndex` algorithm is used to understand the relationship between the images in the archive and a `RelevancyIndex` algorithm is used to rank results based on the user's positive and negative adjudications.

SMQTK comes with a pair of web-based application that implements an IQR system using SMQTK's services as shown in the *SMQTK IQR Workflow* figure.

Running the IQR Application

The SMQTK IQR demonstration application consists of two web services: one for hosting the models and processing for an archive, and a second for providing a user-interface to one or more archives.

In order to run the IQR demonstration application, we will need an archive of imagery. SMQTK has facilities for creating indexes that support 10's or even 100's or 1000's of images. For demonstration purposes, we'll use a modest archive of images. The Leeds Butterfly Dataset will serve quite nicely. Download and unzip the archive (which contains over 800 images of different species of butterflies).

SMQTK comes with a script, `iqr_app_model_generation`, that computes the descriptors on all of the images in your archive and builds up the models needed by the *`NearestNeighborsIndex`* and *`RelevancyIndex`* algorithms.

```
usage: iqr_app_model_generation [-h] [-v] -c PATH PATH -t TAB GLOB [GLOB ...]
```

Positional Arguments

GLOB Shell glob to files to add to the configured data set.

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

-c, --config Path to the JSON configuration files. The first file provided should be the configuration file for the `IqrSearchDispatcher` web-application and the second should be the configuration file for the `IqrService` web-application.

-t, --tab The configuration "tab" of the `IqrSearchDispatcher` configuration to use. This informs what dataset to add the input data files to.

The `-c/--config` option should be given the 2 paths to the configuration files for the `IqrSearchDispatcher` and `IqrService` web services respectively. These provide the configuration blocks for each of the SMQTK algorithms (*`DescriptorGenerator`*, `NearestNeighborIndex`, etc.) required to generate the models and indices that will be required by the application. For convenience, the same configuration files will be provided to the web applications when they are run later.

The SMQTK source repository contains sample configuration files for both the `IqrSearchDispatcher` and `IqrService` services. They can be found at `source/python/smqtk/web/search_app/sample_configs/config.IqrSearchApp.json` and `source/python/smqtk/web/search_app/sample_configs/config.IqrRestService.json` respectively. The `iqr_app_model_generation` script is designed to run from an empty directory and will create the sub-directories specified in the above configurations requires when run.

Since these configuration files drive both the generation of the models and the web applications themselves, a closer examination is in order.

Present in both configuration files are the `flask_app` and `server` sections which control Flask web server application parameters. The `config.IqrSearchApp.json` contains the additional section `mongo` that configures the MongoDB server the UI service uses for storing user session information.

```
1 {
2     "flask_app": {
```

(continues on next page)

(continued from previous page)

```
3           "BASIC_AUTH_PASSWORD": "demo",
4           "BASIC_AUTH_USERNAME": "demo",
5           "SECRET_KEY": "MySuperUltraSecret"
6       },
7       "server": {
8           "host": "127.0.0.1",
9           "port": 5000
10      },
11      "mongo": {
12          "database": "smqtk",
13          "server": "127.0.0.1:27017"
14      },
15      "iqr_tabs": {
16          "LEEDS Butterflies": {
17              "working_directory": "workdir",
18              "data_set": {
19                  "smqtk.representation.data_set.memory_set.DataMemorySet": {
20                      "cache_element": {
21                          "smqtk.representation.data_element.file_element.
    ↪DataFileElement": {
22                              "explicit_mimetype": null,
23                              "filepath": "workdir/butterflies_alexnet_fc7/data.
    ↪memorySet.cache",
24                              "readonly": false
25                          },
26                          "type": "smqtk.representation.data_element.file_element.
    ↪DataFileElement"
27                      },
28                      "pickle_protocol": -1
29                  },
30                  "type": "smqtk.representation.data_set.memory_set.DataMemorySet"
31              },
32              "iqr_service_url": "http://localhost:5001"
33          }
34      }
35  }
```

The `config.IqrSerchApp.json` configuration has an additional block "iqr_tabs" (line 15). This defines the different archives, and matching IQR REST service describing that archive, the UI is to provide an interface for. In our case there will be only one entry, "LEEDS Butterflies" (line 16), representing the archive that we are currently building. This section describes the data-set container that contains the archive imagery to show in the UI (line 18) as well as the URL to the RESTful service providing the IQR functions for the archive (line 32).

In the `config.IqrRestService.json` configuration file (shown below) we see the specification of the algorithm and representation plugins the RESTful IQR service app will use under `iqr_service -> plugins`. Each of these of these blocks is passed to the SMQTK plugin system to create the appropriate instances of the algorithm or data representation in question. The blocks located at lines 35, 66, and 147 configure the three main algorithms used by the application: the descriptor generator, the nearest neighbors index, and the relevancy index. For example the `nn_index` block that starts at line 66 specifies two different implementations: `FlannNearestNeighborsIndex`, which uses the Flann library, and `LSHNearestNeighborIndex`, configured to use the Iterative Quantization hash function (paper). The `type` element on line 135 selects the `LSHNearestNeighborIndex` to be used for this configuration.

(jump past configuration display)

```
1   {
```

(continues on next page)

(continued from previous page)

```
    "flask_app": {
        "BASIC_AUTH_PASSWORD": "demo",
        "BASIC_AUTH_USERNAME": "demo",
        "SECRET_KEY": "MySuperUltraSecret"
    },
    "server": {
        "host": "127.0.0.1",
        "port": 5001
    },
    "iqr_service": {
        "plugins": {
            "classification_factory": {
                "smqtk.representation.classification_element.memory.
↪MemoryClassificationElement": {},
                "type": "smqtk.representation.classification_element.memory.
↪MemoryClassificationElement"
            },
            "classifier_config": {
                "smqtk.algorithms.classifier.libsvm.LibSvmClassifier": {
                    "normalize": 2,
                    "svm_label_map_uri": null,
                    "svm_model_uri": null,
                    "train_params": {
                        "-b": 1,
                        "-c": 2,
                        "-s": 0,
                        "-t": 0
                    }
                },
                "type": "smqtk.algorithms.classifier.libsvm.LibSvmClassifier"
            },
            "descriptor_factory": {
                "smqtk.representation.descriptor_element.local_elements.
↪DescriptorMemoryElement": {},
                "type": "smqtk.representation.descriptor_element.local_elements.
↪DescriptorMemoryElement"
            },
            "descriptor_generator": {
                "type": "smqtk.algorithms.descriptor_generator.caffe_descriptor.
↪CaffeDescriptorGenerator",
                "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": "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": "bvlc_alexnet/deploy.prototxt",
```

(continues on next page)

(continued from previous page)

```
                            "readonly": true
                        }
                    },
                    "image_mean": {
                        "type": "smqtk.representation.data_element.file_element.
↪DataFileElement",
                        "smqtk.representation.data_element.file_element.
↪DataFileElement": {
                            "filepath": "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
                }
            },
            "descriptor_set": {
                "smqtk.representation.descriptor_set.memory.MemoryDescriptorSet": {
                    "cache_element": {
                        "smqtk.representation.data_element.file_element.
↪DataFileElement": {
                            "explicit_mimetype": null,
                            "filepath": "workdir/butterflies_alexnet_fc7/descriptor_
↪set.pickle",
                            "readonly": false
                        },
                        "type": "smqtk.representation.data_element.file_element.
↪DataFileElement"
                    },
                    "pickle_protocol": -1
                },
                "type": "smqtk.representation.descriptor_set.memory.
↪MemoryDescriptorSet"
            },
            "neighbor_index": {
                "smqtk.algorithms.nn_index.lsh.LSHNearestNeighborIndex": {
                    "descriptor_set": {
                        "smqtk.representation.descriptor_set.memory.
↪MemoryDescriptorSet": {
                            "cache_element": {
                                "smqtk.representation.data_element.file_element.
↪DataFileElement": {
                                    "explicit_mimetype": null,
                                    "filepath": "workdir/butterflies_alexnet_fc7/
↪descriptor_set.pickle",
                                    "readonly": false
                                },
                                "type": "smqtk.representation.data_element.file_
↪element.DataFileElement"
```

(continues on next page)

(continued from previous page)

```
                            },
                            "pickle_protocol": -1
                        },
                        "type": "smqtk.representation.descriptor_set.memory.
↪MemoryDescriptorSet"
                    },
                    "distance_method": "cosine",
                    "hash2uuids_kvstore": {
                        "smqtk.representation.key_value.memory.MemoryKeyValueStore": {
                            "cache_element": {
                                "smqtk.representation.data_element.file_element.
↪DataFileElement": {
                                    "explicit_mimetype": null,
                                    "filepath": "workdir/butterflies_alexnet_fc7/
↪hash2uuids.mem_kvstore.pickle",
                                    "readonly": false
                                },
                                "type": "smqtk.representation.data_element.file_
↪element.DataFileElement"
                            }
                        },
                        "type": "smqtk.representation.key_value.memory.
↪MemoryKeyValueStore"
                    },
                    "hash_index": {
                        "type": null
                    },
                    "hash_index_comment": "'hash_index' may also be null to default␣
↪to a linear index built at query time.",
                    "lsh_functor": {
                        "smqtk.algorithms.nn_index.lsh.functors.itq.ItqFunctor": {
                            "bit_length": 64,
                            "itq_iterations": 50,
                            "mean_vec_cache": {
                                "smqtk.representation.data_element.file_element.
↪DataFileElement": {
                                    "explicit_mimetype": null,
                                    "filepath": "workdir/butterflies_alexnet_fc7/
↪itqnn/mean_vec.npy",
                                    "readonly": false
                                },
                                "type": "smqtk.representation.data_element.file_
↪element.DataFileElement"
                            },
                            "normalize": null,
                            "random_seed": 42,
                            "rotation_cache": {
                                "smqtk.representation.data_element.file_element.
↪DataFileElement": {
                                    "explicit_mimetype": null,
                                    "filepath": "workdir/butterflies_alexnet_fc7/
↪itqnn/rotation.npy",
                                    "readonly": false
                                },
                                "type": "smqtk.representation.data_element.file_
↪element.DataFileElement"
                            }
```

(continues on next page)

(continued from previous page)

```
                            },
                            "type": "smqtk.algorithms.nn_index.lsh.functors.itq.ItqFunctor
↪"
                        },
                        "read_only": false
                    },
                    "type": "smqtk.algorithms.nn_index.lsh.LSHNearestNeighborIndex"
                },
                "relevancy_index_config": {
                    "smqtk.algorithms.relevancy_index.libsvm_hik.LibSvmHikRelevancyIndex
↪": {
                        "autoneg_select_ratio": 1,
                        "cores": null,
                        "descr_cache_filepath": null,
                        "multiprocess_fetch": false
                    },
                    "type": "smqtk.algorithms.relevancy_index.libsvm_hik.
↪LibSvmHikRelevancyIndex"
                }
            },
            "session_control": {
                "positive_seed_neighbors": 500,
                "session_expiration": {
                    "check_interval_seconds": 30,
                    "enabled": false,
                    "session_timeout": 3600
                }
            }
        }
}
```

Once you have the configuration file set up the way that you like it, you can generate all of the models and indexes required by the application by running the following command:

```
iqr_app_model_generation \
    -c config.IqrSearchApp.json config.IqrRestService.json \
    -t "LEEDS Butterflies" /path/to/butterfly/images/*.jpg
```

This will generate descriptors for all of the images in the data set and use them to compute the models and indices we configured, outputting to the files under the `workdir` directory in your current directory.

Once it completes, you can run the `IqrSearchApp` and `IqrService` web-apps. You'll need an instance of MongoDB running on the port and host address specified by the `mongo` element on line 13 in your `config.IqrSearchApp.json` configuration file. You can start a Mongo instance (presuming you have it installed) with:

```
mongod --dbpath /path/to/mongo/data/dir
```

Once Mongo has been started you can start the `IqrSearchApp` and `IqrService` services with the following commands in separate terminals:

```
# Terminal 1
runApplication -a IqrService -c config.IqrRestService.json

# Terminal 2
runApplication -a IqrSearchDispatcher -c config.IqrSearchApp.json
```

After the services have been started, open a web browser and navigate to `http://localhost:5000`. Click lick on

the `login` button in the upper-right and then enter the credentials specified in the default login settings file `source/python/smqtk/web/search_app/modules/login/users.json`.

Fig. 2: *Click on the login element to enter your credentials*

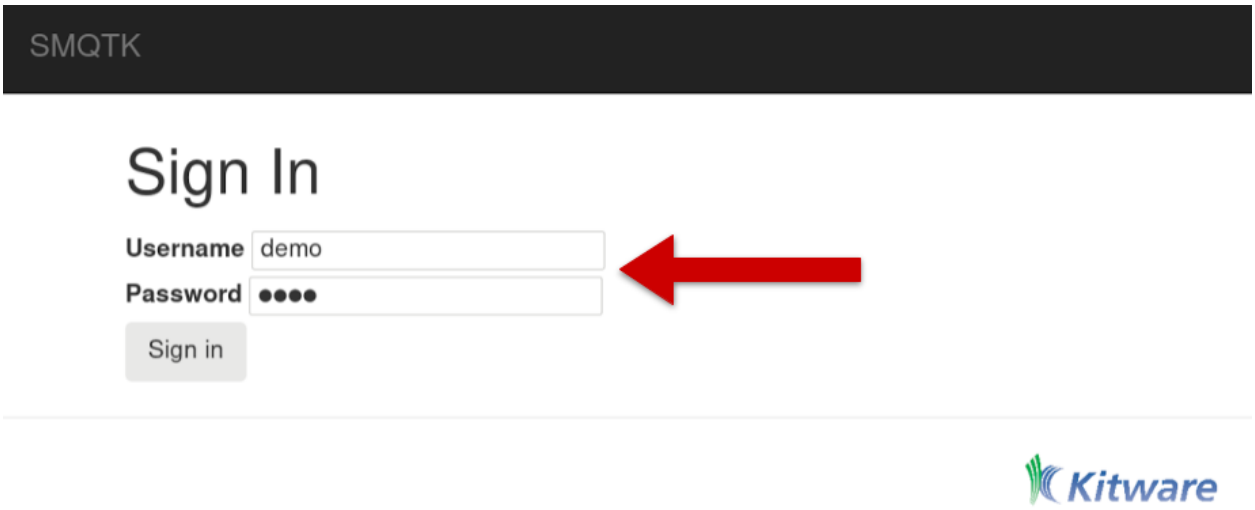

Fig. 3: *Enter demo credentials*

Once you've logged in you will be able to select the `LEEDS Butterfly` link. This link was named by line 16 in the `config.IqrSearchApp.json` configuration file. The `iqr_tabs` mapping allows you to configure interfacing with different IQR REST services providing different combinations of the required algorithms – useful for example, if you want to compare the performance of different descriptors or nearest-neighbor index algorithms.

To begin the IQR process drag an exemplar image to the grey load area (marked `1` in the next figure). In this case we've uploaded a picture of a Monarch butterfly (`2`). Once uploaded, click the `Initialize Index` button (`3`) and the system will return a set of images that it believes are similar to the exemplar image based on the descriptor computed.

The next figure shows the set of images returned by the system (on the left) and a random selection of images from the archive (by clicking the `Toggle Random Results` element). As you can see, even with just one exemplar the system is beginning to learn to return Monarch butterflies (or butterflies that look like Monarchs)

At this point you can begin to refine the query. You do this by marking correct returns at their checkbox and incorrect returns at the "X". Once you've marked a number of returns, you can select the "Refine" element which will use your adjudications to retrain and rerank the results with the goal that you will increasingly see correct results in your result set.

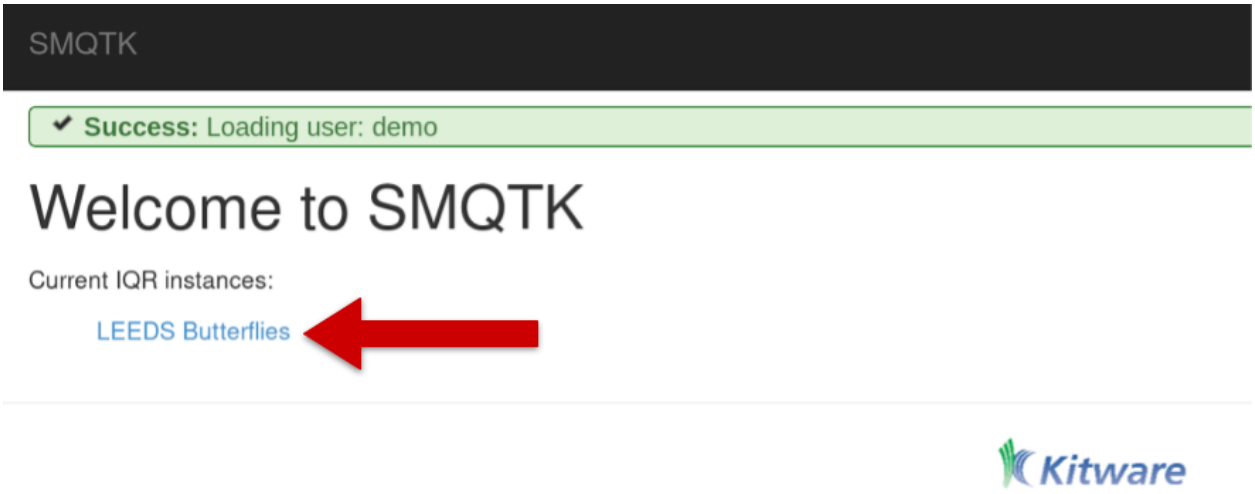

Fig. 4: *Select the "LEEDS Butterflies" link to begin working with the application*

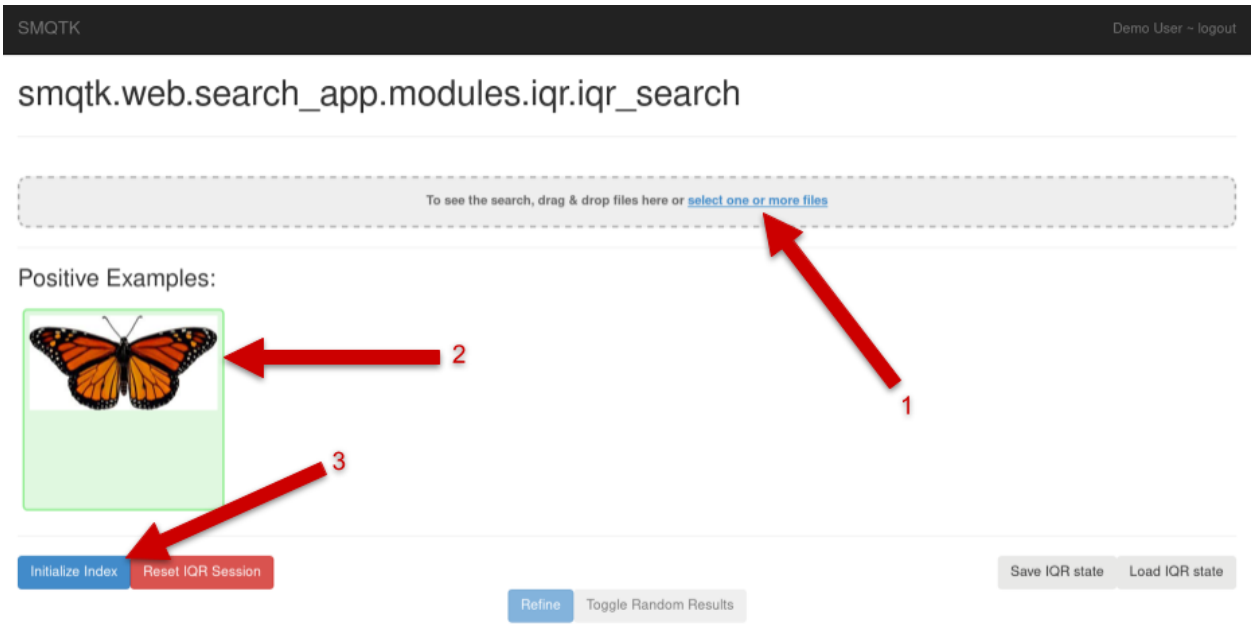

Fig. 5: *IQR Initilization*

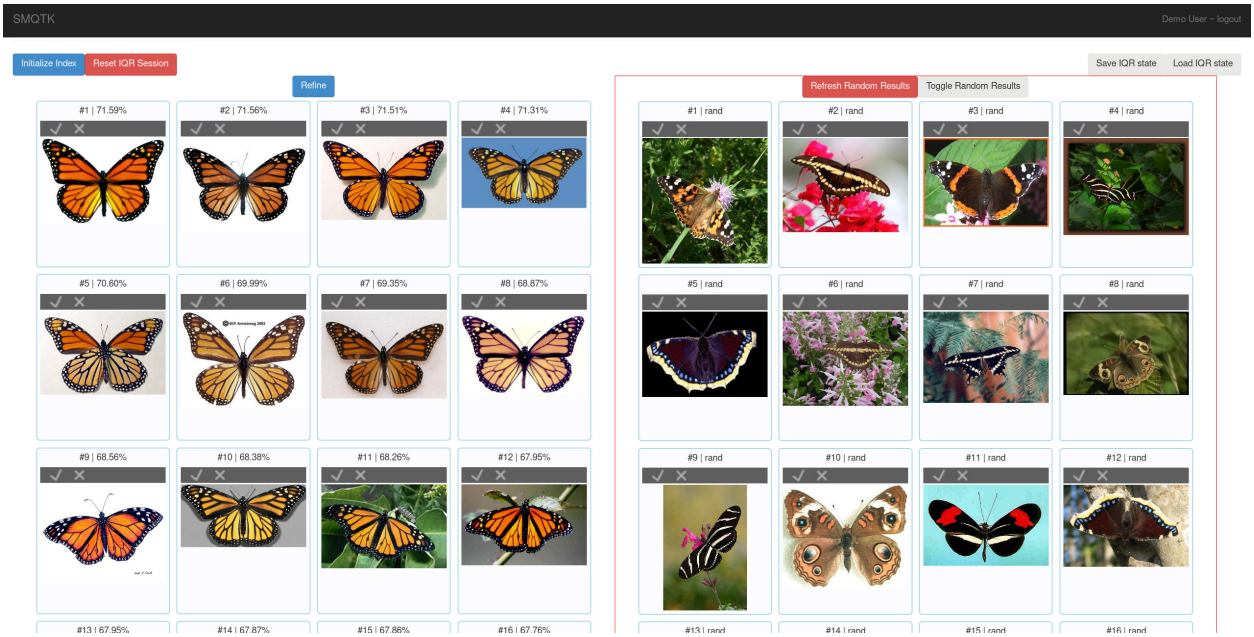

Fig. 6: *Initial Query Results and Random Results*

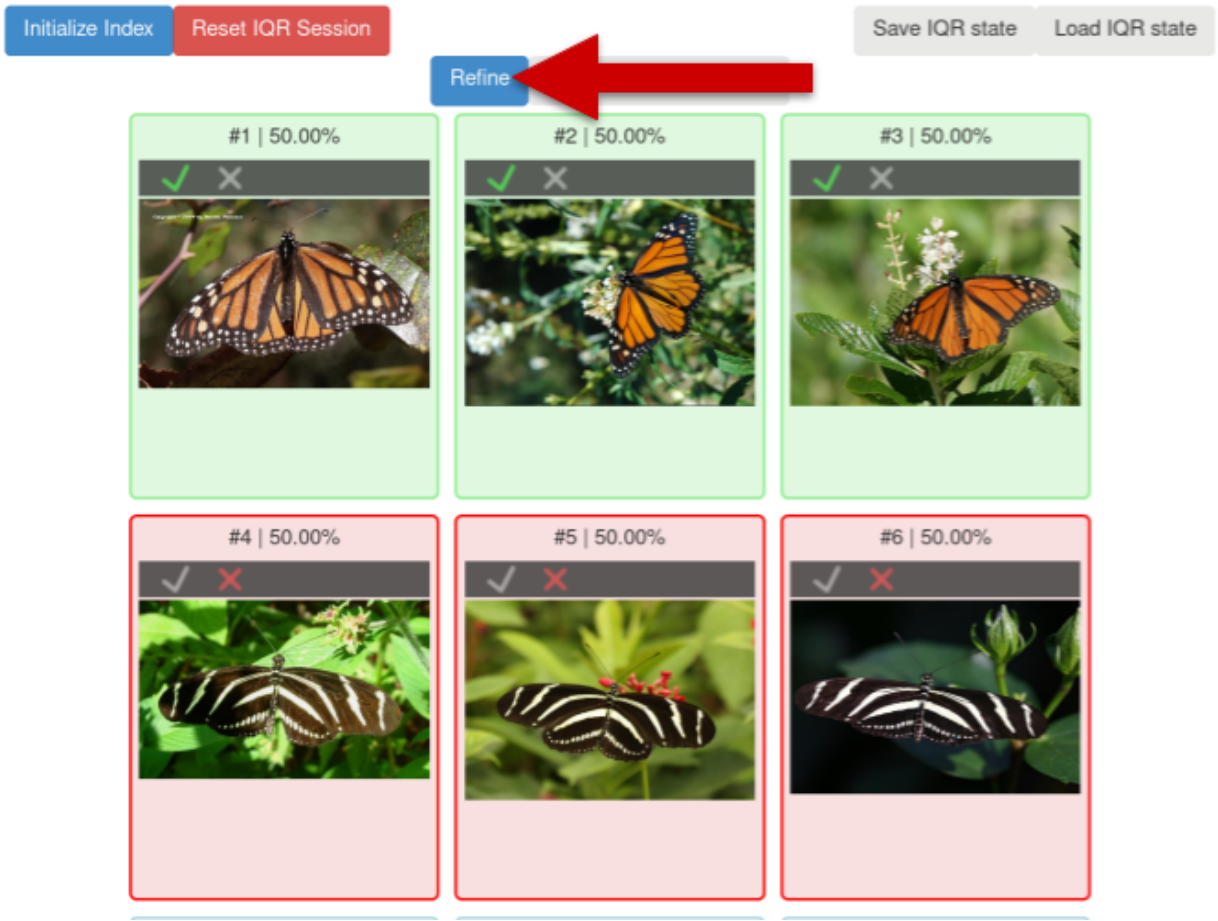

Fig. 7: *Query Refinement*

You can continue this process for as long as you like until you are satisfied with the results that the query is returning. Once you are happy with the results, you can select the `Save IQR State` button. This will save a file that contains all of the information requires to use the results of the IQR query as an image classifier. The process for doing this is described in the next session.

Using an IQR Trained Classifier

Before you can use your IQR session as a classifier, you must first train the classifier model from the IQR session state. You can do this with the `iqrTrainClassifier` tool:

```
usage: iqrTrainClassifier [-h] [-v] [-c PATH] [-g PATH] [-i IQR_STATE]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

-i, --iqr-state Path to the ZIP file saved from an IQR session.

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

As with other tools from SMQTK the configuration file is a JSON file. An default configuration file may be generated by calling `iqrTrainClassifier -g example.json`, but pre-configured example file can be found `here` and is shown below:

```
{
    "classifier": {
        "smqtk.algorithms.classifier.libsvm.LibSvmClassifier": {
            "normalize": 2,
            "svm_label_map_uri": "workdir/iqr_classifier/label_map",
            "svm_model_uri": "workdir/iqr_classifier/model",
            "train_params": {
                "-b": 1,
                "-c": 2,
                "-s": 0,
                "-t": 0
            }
        },
        "type": "smqtk.algorithms.classifier.libsvm.LibSvmClassifier"
    }
}
```

The above configuration specifies the classifier that will be used, in this case the `LibSvmClassifier`. Let us assume the IQR session state was downloaded as `monarch.IqrState`. The following command will train a classifier leveraging the descriptors labeled by the IQR session that was saved:

```
iqrTrainClassifier.py -c config.iqrTrainClassifier.json -i monarch.IqrState
```

Once you have trained the classifier, you can use the `classifyFiles` command to actually classify a set of files.

```
usage: smqtk-classify-files [-h] [-v] [-c PATH] [-g PATH] [--overwrite]
                            [-l LABEL]
                            [GLOB [GLOB ...]]
```

Positional Arguments

GLOB Series of shell globs specifying the files to classify.

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

Classification

--overwrite When generating a configuration file, overwrite an existing file.

Default: False

-l, --label The class to filter by. This is based on the classifier configuration/model used. If this is not provided, we will list the available labels in the provided classifier configuration.

Again, we need to provide a JSON configuration file for the command. As with `iqrTrainClassifier`, there is a sample configuration file in the repository:

```
{
    "classification_factory": {
        "smqtk.representation.classification_element.memory.
↪MemoryClassificationElement": {},
        "type": "smqtk.representation.classification_element.memory.
↪MemoryClassificationElement"
    },
    "classifier": {
        "smqtk.algorithms.classifier.libsvm.LibSvmClassifier": {
            "normalize": 2,
            "svm_label_map_uri": "workdir/iqr_classifier/label_map",
            "svm_model_uri": "workdir/iqr_classifier/model",
            "train_params": {
```

(continues on next page)

(continued from previous page)

```
                "-b": 1,
                "-c": 2,
                "-s": 0,
                "-t": 0
            }
        },
        "type": "smqtk.algorithms.classifier.libsvm.LibSvmClassifier"
    },
    "descriptor_factory": {
        "smqtk.representation.descriptor_element.local_elements.
↪DescriptorMemoryElement": {},
        "type": "smqtk.representation.descriptor_element.local_elements.
↪DescriptorMemoryElement"
    },
    "descriptor_generator": {
        "type": "smqtk.algorithms.descriptor_generator.caffe_descriptor.
↪CaffeDescriptorGenerator",
        "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": "bvlc_alexnet.caffemodel",
                    "readonly":true
                }
            },
            "network_prototxt": {
                "type": "smqtk.representation.data_element.file_element.
↪DataFileElement",
                "smqtk.representation.data_element.file_element.DataFileElement": {
                    "filepath": "bvlc_alexnet/deploy.prototxt",
                    "readonly": true
                }
            },
            "image_mean": {
                "type": "smqtk.representation.data_element.file_element.
↪DataFileElement",
                "smqtk.representation.data_element.file_element.DataFileElement": {
                    "filepath": "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
        }
    }
}
```

Note that the `classifier` block on lines 7-18 is the same as the `classifier` block in the `iqrTrainClassfier` configuration file. Further, the `descriptor_generator` block on lines 25-39 matches the descriptor generator used for the IQR application itself (thus matching the type of descriptor used to train the classifier).

Once you've set up the configuration file to your liking, you can classify a set of labels with the following command:

```
smqtk-classify-files -c config.classifyFiles.json -l positive /path/to/butterfly/
↪images/*.jpg
```

If you leave the `-l` argument, the command will tell you the labels available with the classifier (in this case *positive* and *negative*).

SMQTK's `smqtk-classify-files` tool can use this saved IQR state to classify a set of files (not necessarily the files in your IQR Applicaiton ingest). The command has the following form:

3.5 Utilities and Applications

Also part of SMQTK are support utility modules, utility scripts (effectively the "binaries") and service-oriented and demonstration web applications.

3.5.1 Utility Modules

Various unclassified functionality intended to support the primary goals of SMQTK. See doc-string comments on sub-module classes and functions in [`smqtk.utils`](/python/smqtk/utils) module.

3.5.2 Utility Scripts

Located in the [`smqtk.bin`](/python/smqtk/bin) module are various scripts intended to provide quick access or generic entry points to common SMQTK functionality. These scripts generally require configuration via a JSON text file and executable entry points are installed via the `setup.py`. By rule of thumb, scripts that require a configuration also provide an option for outputting a default or example configuration file.

Currently available utility scripts in alphabetical order:

classifier_kfold_validation

Helper utility for cross validating a supervised classifier configuration. The classifier used should NOT be configured to save its model since this process requires us to train the classifier multiple times.

- **plugins**
  - **supervised_classifier** Supervised Classifier implementation configuration to use. This should not be set to use a persistent model if able (this utility will repeatedly train a new model for each fold).
  - **descriptor_set** Index to draw descriptors to classify from.
- **cross_validation**

  - **truth_labels** Path to a CSV file containing descriptor UUID the truth label associations. This defines what descriptors are used from the given index. We error if any descriptor UUIDs listed here are not available in the given descriptor index. This file should be in [uuid, label] column format.
  - **num_folds** Number of folds to make for cross validation.
  - **random_seed** Optional fixed seed for the
  - **classification_use_multiprocessing** If we should use multiprocessing (vs threading) when classifying elements.
- **pr_curves**
  - **enabled** If Precision/Recall plots should be generated.
  - **show** If we should attempt to show the graph after it has been generated (matplotlib).
  - **output_directory** Directory to save generated plots to. If None, we will not save plots. Otherwise we will create the directory (and required parent directories) if it does not exist.
  - **file_prefix** String prefix to prepend to standard plot file names.
- **roc_curves**
  - **enabled** If ROC curves should be generated
  - **show** If we should attempt to show the plot after it has been generated (matplotlib).
  - **output_directory** Directory to save generated plots to. If None, we will not save plots. Otherwise we will create the directory (and required parent directories) if it does not exist.
  - **file_prefix** String prefix to prepend to standard plot file names.

```
usage: classifier_kfold_validation [-h] [-v] [-c PATH] [-g PATH]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

classifier_model_validation

Utility for validating a given classifier implementation's model against some labeled testing data, outputting PR and ROC curve plots with area-under-curve score values.

This utility can optionally be used train a supervised classifier model if the given classifier model configuration does not exist and a second CSV file listing labeled training data is provided. Training will be attempted if `train` is set to true. If training is performed, we exit after training completes. A `SupervisedClassifier` sub-classing implementation must be configured

We expect the test and train CSV files in the column format:

... <UUID>,<label> ...

The UUID is of the descriptor to which the label applies. The label may be any arbitrary string value, but all labels must be consistent in application.

Some metrics presented assume the highest confidence class as the single predicted class for an element:

- confusion matrix

The output UUID confusion matrix is a JSON dictionary where the top-level keys are the true labels, and the inner dictionary is the mapping of predicted labels to the UUIDs of the classifications/descriptors that yielded the prediction. Again, this is based on the maximum probability label for a classification result (T=0.5).

See Scikit-Learn PR and ROC curve explanations and examples:

- http://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html
- http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

```
usage: classifier_model_validation [-h] [-v] [-c PATH] [-g PATH]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

classifyFiles

Based on an input, trained classifier configuration, classify a number of media files, whose descriptor is computed by the configured descriptor generator. Input files that classify as the given label are then output to standard out. Thus, this script acts like a filter.

```
usage: classifyFiles [-h] [-v] [-c PATH] [-g PATH] [--overwrite] [-l LABEL]
                     [GLOB [GLOB ...]]
```

Positional Arguments

GLOB Series of shell globs specifying the files to classify.

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

Classification

--overwrite When generating a configuration file, overwrite an existing file.

Default: False

-l, --label The class to filter by. This is based on the classifier configuration/model used. If this is not provided, we will list the available labels in the provided classifier configuration.

compute_classifications

Script for asynchronously computing classifications for DescriptorElements in a DescriptorSet specified via a list of UUIDs. Results are output to a CSV file in the format:

uuid, label1_confidence, label2_confidence, ...

CSV columns labels are output to the given CSV header file path. Label columns will be in the order as reported by the classifier implementations `get_labels` method.

Due to using an input file-list of UUIDs, we require that the UUIDs of indexed descriptors be strings, or equality comparable to the UUIDs' string representation.

```
usage: compute_classifications [-h] [-v] [-c PATH] [-g PATH]
                               [--uuids-list PATH] [--csv-header PATH]
                               [--csv-data PATH]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

Input Output Files

--uuids-list Path to the input file listing UUIDs to process.

--csv-header Path to the file to output column header labels.

--csv-data Path to the file to output the CSV data to.

compute_hash_codes

Compute LSH hash codes based on the provided functor on all or specific descriptors from the configured index given a file-list of UUIDs.

When using an input file-list of UUIDs, we require that the UUIDs of indexed descriptors be strings, or equality comparable to the UUIDs' string representation.

We update a key-value store with the results of descriptor hash computation. We assume the keys of the store are the integer hash values and the values of the store are `frozenset` instances of descriptor UUIDs (hashable-type objects). We also assume that no other source is concurrently modifying this key-value store due to the need to modify the values of keys.

```
usage: compute_hash_codes [-h] [-v] [-c PATH] [-g PATH] [--uuids-list PATH]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

I/O

--uuids-list Optional path to a file listing UUIDs of descriptors to computed hash codes for. If not provided we compute hash codes for all descriptors in the configured descriptor index.

compute_many_descriptors

Descriptor computation helper utility. Checks data content type with respect to the configured descriptor generator to skip content that does not match the accepted types. Optionally, we can additionally filter out image content whose image bytes we cannot load via `PIL.Image.open`.

```
usage: compute_many_descriptors [-h] [-v] [-c PATH] [-g PATH] [-b INT]
                                [--check-image] [-f PATH] [-p PATH]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

-b, --batch-size Number of files to batch together into a single compute async call. This defines the granularity of the checkpoint file in regards to computation completed. If given 0, we do not batch and will perform a single `compute_async` call on the configured generator. Default batch size is 0.

Default: 0

--check-image If se should check image pixel loading before queueing an input image for processing. If we cannot load the image pixels via `PIL.Image.open`, the input image is not queued for processing

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

Required Arguments

-f, --file-list Path to a file that lists data file paths. Paths in this file may be relative, but will at some point be coerced into absolute paths based on the current working directory.

-p, --completed-files Path to a file into which we add CSV format lines detailing filepaths that have been computed from the file-list provided, as the UUID for that data (currently the SHA1 checksum of the data).

computeDescriptor

Compute a descriptor vector for a given data file, outputting the generated feature vector to standard out, or to an output file if one was specified (in numpy format).

```
usage: computeDescriptor [-h] [-v] [-c PATH] [-g PATH] [--overwrite]
                         [-o OUTPUT_FILEPATH]
                         [input_file]
```

Positional Arguments

input_file Data file to compute descriptor on

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

--overwrite Force descriptor computation even if an existing descriptor vector was discovered based on the given content descriptor type and data combination.

Default: False

-o, --output-filepath Optional path to a file to output feature vector to. Otherwise the feature vector is printed to standard out. Output is saved in numpy binary format (.npy suffix recommended).

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

createFileIngest

Add a set of local system files to a data set via explicit paths or shell-style glob strings.

```
usage: createFileIngest [-h] [-v] [-c PATH] [-g PATH] [GLOB [GLOB ...]]
```

Positional Arguments

GLOB

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

descriptors_to_svmtrainfile

Utility script to transform a set of descriptors, specified by UUID, with matching class labels, to a test file usable by libSVM utilities for train/test experiments.

The input CSV file is assumed to be of the format:

uuid,label ...

This is the same as the format requested for other scripts like `classifier_model_validation.py`.

This is very useful for searching for -c and -g parameter values for a training sample of data using the `tools/grid.py` script, found in the libSVM source tree. For example:

<smqtk_source>/TPL/libsvm-3.1-custom/tools/grid.py -log2c -5,15,2 -log2c 3,-15,-2 -v 5 -out libsvm.grid.out -png libsvm.grid.png -t 0 -w1 3.46713615023 -w2 12.2613240418 output_of_this_script.txt

```
usage: descriptors_to_svmtrainfile [-h] [-v] [-c PATH] [-g PATH] [-f PATH]
                                   [-o PATH]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

IO Options

-f Path to the csv file mapping descriptor UUIDs to their class label. String labels are transformed into integers for libSVM. Integers start at 1 and are applied in the order that labels are seen in this input file.

-o Path to the output file to write libSVM labeled descriptors to.

generate_image_transform

Utility for transforming an input image in various standardized ways, saving out those transformed images with standard namings. Transformations used are configurable via a configuration file (JSON).

Configuration details: {

”crop”: {

“center_levels”: null | int # If greater than 0, crop out one or more increasing smaller images # from a base image by cutting off increasingly larger portions of # the outside perimeter. Cropped image dimensions determined by the # dimensions of the base image and the number of crops to generate.

”quadrant_pyramid_levels”: null | int # If greater than 0, generate a number of crops based on a number of # quad-tree partitions made based on the given number of levels. # Partitions for all levels less than the level provides are also # made.

”tile_shape”: null | [width, height] # If not null and is a list of two integers, crop out tile windows # from the base image that have the width and height specified. # If the image width or height is not evenly divisible by the tile # width or height, respectively, then the crop out as many tiles as # neatly fit starting from the axis origin. The remaining pixels are # ignored.

”tile_stride”: null | [x, y] # If not null and is a list of two integers, crop out sub-images of # the above width and height (if given) with this stride. When not # this is not provided, the default stride is the same as the tile # width and height.

},

"brightness_levels": null | int # Generate a number of images with different brightness levels using # linear interpolation to choose levels between 0 (black) and 1 # (original image) as well as between 1 and 2. # Results will not include contrast level 0, 1 or 2 images.

"contrast_levels": null | int # Generate a number of images with different contrast levels using # linear interpolation to choose levels between 0 (black) and 1 # (original image) as well as between 1 and 2. # Results will not include contrast level 0, 1 or 2 images.

}

```
usage: generate_image_transform [-h] [-v] [-c PATH] [-g PATH] [-i IMAGE]
                                [-o OUTPUT]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

Input/Output

-i, --image Image to produce transformations for.

-o, --output Directory to output generated images to. By default, if not told otherwise, we will write output images in the same directory as the source image. Output images share a core filename as the source image, but with extra suffix syntax to differentiate produced images from the original. Output images will share the same image extension as the source image.

iqr_app_model_generation

Train and generate models for the SMQTK IQR Application.

This application takes the same configuration file as the IqrService REST service. To generate a default configuration, please refer to the `runApplication` tool for the `IqrService` application:

runApplication -a IqrService -g config.IqrService.json

```
usage: iqr_app_model_generation [-h] [-v] -c PATH PATH -t TAB GLOB [GLOB ...]
```

Positional Arguments

GLOB Shell glob to files to add to the configured data set.

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

-c, --config Path to the JSON configuration files. The first file provided should be the configuration file for the `IqrSearchDispatcher` web-application and the second should be the configuration file for the `IqrService` web-application.

-t, --tab The configuration "tab" of the `IqrSearchDispatcher` configuration to use. This informs what dataset to add the input data files to.

iqrTrainClassifier

Train a supervised classifier based on an IQR session state dump.

Descriptors used in IQR, and thus referenced via their UUIDs in the IQR session state dump, must exist external to the IQR web-app (uses a non-memory backend). This is needed so that this script might access them for classifier training.

Click the "Save IQR State" button to download the IqrState file encapsulating the descriptors of positively and negatively marked items. These descriptors will be used to train the configured SupervisedClassifier.

```
usage: iqrTrainClassifier [-h] [-v] [-c PATH] [-g PATH] [-i IQR_STATE]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

-i, --iqr-state Path to the ZIP file saved from an IQR session.

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

make_balltree

Script for building and saving the model for the `SkLearnBallTreeHashIndex` implementation of `HashIndex`.

```
usage: make_balltree [-h] [-v] [-c PATH] [-g PATH]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

minibatch_kmeans_clusters

Script for generating clusters from descriptors in a given descriptor set using the mini-batch KMeans implementation from Scikit-learn (http://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html).

By the nature of Scikit-learn's MiniBatchKMeans implementation, euclidean distance is used to measure distance between descriptors.

```
usage: minibatch_kmeans_clusters [-h] [-v] [-c PATH] [-g PATH] [-o PATH]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

output

-o, --output-map Path to output the clustering class mapping to. Saved as a pickle file with -1 format.

proxyManagerServer

Server for hosting proxy manager which hosts proxy object instances.

This takes a simple configuration file that looks like the following:

[server]
port = <integer>
authkey = <string>

```
usage: proxyManagerServer [-h] [-v] [-c PATH] [-g PATH]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

removeOldFiles

Utility to recursively scan and remove files underneath a given directory if they have not been modified for longer than a set amount of time.

```
usage: removeOldFiles [-h] [-d BASE_DIR] [-i INTERVAL] [-e EXPIRY] [-v]
```

Named Arguments

-d, --base-dir	Starting directory for scan.
-i, --interval	Number of seconds between each scan (integer).
-e, --expiry	Number of seconds until a file has "expired" (integer).
-v, --verbose	Display more messages (debugging). Default: False

runApplication

Generic entry point for running SMQTK web applications defined in [`smqtk.web`](/python/smqtk/web).

Runs conforming SMQTK Web Applications.

```
usage: runApplication [-h] [-v] [-c PATH] [-g PATH] [-l] [-a APPLICATION] [-r]
                      [-t] [--host HOST] [--port PORT] [--use-basic-auth]
                      [--use-simple-cors] [--debug-server] [--debug-smqtk]
                      [--debug-app] [--debug-ns DEBUG_NS]
```

Named Arguments

-v, --verbose	Output additional debug logging. Default: False

Configuration

-c, --config	Path to the JSON configuration file.
-g, --generate-config	Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

Application Selection

-l, --list	List currently available applications for running. More description is included if SMQTK verbosity is increased (-v | –debug-smqtk) Default: False
-a, --application	Label of the web application to run.

Server options

-r, --reload Turn on server reloading.

Default: False

-t, --threaded Turn on server multi-threading.

Default: False

--host Run host address specification override. This will override all other configuration method specifications.

--port Run port specification override. This will override all other configuration method specifications.

--use-basic-auth Use global basic authentication as configured.

Default: False

--use-simple-cors Allow CORS for all domains on all routes. This follows the "Simple Usage" of flask-cors: https://flask-cors.readthedocs.io/en/latest/#simple-usage

Default: False

Other options

--debug-server Turn on server debugging messages ONLY. This is implied when -v|–verbose is enabled.

Default: False

--debug-smqtk Turn on SMQTK debugging messages ONLY. This is implied when -v|–verbose is enabled.

Default: False

--debug-app Turn on flask app logger namespace debugging messages ONLY. This is effectively enabled if the flask app is provided with SMQTK and "–debug-smqtk" is passed. This is also implied if -v|–verbose is enabled.

Default: False

--debug-ns Specify additional python module namespaces to enable debug logging for.

Default: []

summarizePlugins

Print out information about what plugins are currently usable and the documentation headers for each implementation.

```
usage: summarizePlugins [-h] [-v] [--defaults DEFAULTS]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

--defaults Optionally generate default configuration blocks for each plugin structure and output as JSON to the specified path.

Default: False

train_itq

Tool for training the ITQ functor algorithm's model on descriptors in a set.

By default, we use all descriptors in the configured set (`uuids_list_filepath` is not given a value).

The `uuids_list_filepath` configuration property is optional and should be used to specify a sub-set of descriptors in the configured set to train on. This only works if the stored descriptors' UUID is a type of string.

```
usage: train_itq [-h] [-v] [-c PATH] [-g PATH]
```

Named Arguments

-v, --verbose Output additional debug logging.

Default: False

Configuration

-c, --config Path to the JSON configuration file.

-g, --generate-config Optionally generate a default configuration file at the specified path. If a configuration file was provided, we update the default configuration with the contents of the given configuration.

CHAPTER

FOUR

EXAMPLES

4.1 Simple Feature Computation with ColorDescriptor

The following is a concrete example of performing feature computation for a set of ten butterfly images using the *CSIFT* descriptor from the ColorDescriptor software package. It assumes you have set up the `colordescriptor` executable and python library in your *PATH* and *PYTHONPATH*. Once set up, the following code will compute a *CSIFT* descriptor:

```
# 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 you have set up the colordescriptor executable.
from smqtk.algorithms.descriptor_generator import get_descriptor_generator_impls
cd = get_descriptor_generator_impls()['ColorDescriptor_Image_csift'](model_directory=
↪'data', work_directory='work')
cd.generate_model(el)

# 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
factory = DescriptorElementFactory(DescriptorMemoryElement, {})

# Compute features on the first image
result = cd.generate_one_element(el[0], factory)
result.vector()

# array([ 0.        ,  0.01254855,  0.        , ...,  0.0035853 ,
#         0.        ,  0.00388408])
```

4.2 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
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)
```

4.3 NearestNeighborServiceServer Incremental Update Example

4.3.1 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
```

4.3.2 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": {
                    "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
```

(continues on next page)

(continued from previous page)

```
                }
            },
            "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 check**
  - 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"
    },
    "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"
```

(continues on next page)

(continued from previous page)

```
        },
        "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": {
        "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": {
```

(continues on next page)

(continued from previous page)

```
                "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",
                        "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": {
```

(continues on next page)

(continued from previous page)

```
83                  "smqtk.representation.descriptor_set.postgres.PostgresDescriptorSet":
↪{
84                      "db_host": "/dev/shm",
85                      "db_name": "postgres",
86                      "db_pass": null,
87                      "db_port": null,
88                      "db_user": null,
89                      "element_col": "element",
90                      "multiquery_batch_size": 1000,
91                      "pickle_protocol": -1,
92                      "read_only": false,
93                      "table_name": "descriptor_set",
94                      "uuid_col": "uid"
95                  },
96                  "type": "smqtk.representation.descriptor_set.postgres.
↪PostgresDescriptorSet"
97              },
98              "hash2uuids_kvstore": {
99                  "type": "smqtk.representation.key_value.memory.MemoryKeyValueStore",
100                 "smqtk.representation.key_value.memory.MemoryKeyValueStore":  {
101                     "cache_element": {
102                         "type": "smqtk.representation.data_element.file_element.
↪DataFileElement",
103                         "smqtk.representation.data_element.file_element.
↪DataFileElement": {
104                             "filepath": "2d.hash2uuids.pickle",
105                             "readonly": true
106                         }
107                     }
108                 }
109             },
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 det**

  - 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",
  "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": {
        "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":
↪{
```

(continues on next page)

(continued from previous page)

```
                        "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

# Extract UUIDs of files/descriptors just generated
cat 4.completed_files.csv | cut -d, -f2 > 4.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 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

FIVE

RELEASE PROCESS AND NOTES

5.1 Steps of the SMQTK Release Process

Three types of releases are expected to occur: - major - minor - patch

See the `CONTRIBUTING.md` file for information on how to contribute features and patches.

The following process should apply when any release that changes the version number occurs.

5.1.1 Create and merge version update branch

Patch Release

A patch release should only contain fixes for bugs or issues with an existing release. No new features or functionality should be introduced in a patch release. As such, patch releases should only ever be based on an existing release point.

1. Create a new branch off of the `release` branch named something like `release-patch-{NEW_VERSION}`.

- Increment patch value in `python/smqtk/__init__.py` file's `__version__` attribute.
- Rename the `docs/release_notes/pending_patch.rst` file to `docs/release_notes/v{VERSION}.rst`, matching the value in the `__version__` attribute. Add a descriptive paragraph under the title section summarizing this release.
- Add new release notes RST file reference to `docs/release_notes.rst`.

2. Tag branch (see *Tag new version* below).
3. Merge version bump branch into `release` and `master` branches.

Major and Minor Releases

Major and minor releases may add one or more trivial or non-trivial features and functionalities.

1. Create a new branch off of the `master` or `release` named something like `release-[major, minor]-{NEW_VERSION}`.

a) Increment patch value in `VERSION` file.

b) Rename the `docs/release_notes/pending_release.rst` file to `docs/release_notes/v{VERSION}.rst`, matching the value in the `VERSION` file. Add a descriptive paragraph under the title section summarizing this release.

c) Add new release notes RST file reference to `docs/release_notes.rst`.

2. Create a pull/merge request for this branch with master as the merge target. This is to ensure that everything passes CI testing before making the release. If there is an issue then branches should be made and merged into this branch until the issue is resolved.
3. Tag branch (see *Tag new version* below) after resolving issues and before merging into `master`.
4. Reset the release branch (–hard) to point to the new branch/tag.
5. Merge version bump branch into the `master` branch.

5.1.2 Tag new version

Release branches should be tagged in order to record where in the git tree a particular release refers to. The branch off of `master` or `release` is usually the target of such tags.

Currently the `From GitHub` method is preferred as it creates a "verified" release.

From GitHub

Navigate to the releases page on GitHub and click the `Draft a new release` button in upper right.

Fill in the new version in the `Tag version` text box (e.g. `v#.#.#`) and use the same string in the `Release title` text box. The "@" target should be the release branch created above.

Copy and past this version's release notes into the `Describe this release` text box.

Remember to check the `This is a pre-release` check-box if appropriate.

Click the `Public release` button at the bottom of the page when complete.

From Git on the Command Line

Create a new git tag using the new version number (format: `v<MAJOR.<MINOR>.<PATCH>`) on the merge commit for the version update branch merger:

```
$ git tag -a -m "[Major|Minor|Patch] release v#.#.#"
```

Push this new tag to GitHub (assuming origin remote points to SMQTK on GitHub:

```
$ git push origin v#.#.#
```

To add the release notes to GitHub, navigate to the tags page on GitHub and click on the "Add release notes" link for the new release tag. Copy and paste this version's release notes into the description field and the version number should be used as the release title.

5.1.3 Create new version release to PYPI

Make sure the source is checked out on the newest version tag, the repo is clean (no uncommited files/edits), and the `build` and `dist` directories are removed:

```
$ git checkout <VERSION_TAG>
$ rm -r dist python/smqtk.egg-info
```

Create the `build` and `dist` files for the current version with the following command(s) from the source tree root directory:

```
$ python setup.py sdist
```

Make sure your `$HOME/.pypirc` file is up-to-date and includes the following section with your username/password:

```
[pypi]
username = <username>
password = <password>
```

Make sure the `twine` python package is installed and is up-to-date and then upload dist packages created with:

```
$ twine upload dist/*
```

5.2 Release Notes

5.2.1 SMQTK v0.2 Release Notes

This is a minor release if SMQTK that provides both new functionality and fixes over the previous version v0.1.

The highlights of this release are new and updated interface classes, an updated plugin system, new HBase and PostgreSQL DataElement implementations, and a new wrapper for Caffe CNN descriptor extraction.

Additional one-off scripts were added for reference as well as a more generally usable utility for listing out available plugins for the running system and environment.

Additional notes about the release are provided below.

Updates / New Features since v0.1

General

- Added `SmqtkObject`, `SmqtkAlgorithm` and `SmqtkRepresentation` interfaces for high level classification of sub-classes and encapsulation if high level general functionality (like logging).
- Removed GENIE and MASIR archive directories. There is a tag for the last hash where they were present in this repository. Not removed from history so cloning the SMQTK repo is still large.
- Removed geospace web application sub-module (moved elsewhere).

Documentation

- Update documentaiton to reStructured text files and added support for building Sphinx documentation pages.

Plugins

- Added `Pluggable` interface, intended for abstract classes whose implementations are expected to be provided via dynamic plugins, and propagated its use within the code base.
- `get_plugins` function now ensures that loaded classes descend from `Pluggable` and check that they are currently usable.

Data Elements

- Added HBase backend.
- Added PostgreSQL backend.
- Added asynchronous conversion of an iterable of `DataElement` instances into a numpy matrix. Supports `multiprocessing` and `threading` approaches.

Data Sets

- Added default implementation of `contains` method to abstract interface.
- Separated out original `DataFileSet` into separate file-based and in-memory implementations.
- Added file caching of memory-based data sets.

Descriptor Generators

- Expanded construction parameters for ColorDescriptor implementations so as to remove most class-level variables.
- Added `CaffeDefaultImageNet` implementation and support files. This is intended to be used with the `cnn_feature_extractor` binary optionally built with SMQTK.

Nearest Neighbors

- Removed model FLANN implementation model filepath defaults, allowing purely in-memory use without model persistance.

Web Tools

- Added static file hosting flask blueprint in the IQR demo for serving arbitrary directories as a source of static files. Removed need to write generated files into source tree in order to host them.
- Fixed base flask app interface to be `Pluggable`.

Python Utilities

- Shifted some functions around into locations where it makes more sense for them to live
  - `smqtk.utils.safe_create_dir` -> `smqtk.utils.file_utils.safe_create_dir`
  - `smqtk.utils.touch` -> `smqtk.utils.file_utils.touch`

Tools / Scripts

- Added plugin summarization script for listing names and description of currently available plugins for the various SMQTK interfaces.
- Changed IQR model generation example script to use the same configuration file that would be passed to the IQR web app (simplification).
- Added machine specific ITQ code generation scripts

Fixes since v0.1

IQR web application demo

- Fixed preview cache to clean up after itself.

Code Index

- Fixed the way `MemoryCodeIndex` updated descriptor count so as not to count descriptor overwrites as new descriptors.

Descriptor Generators

- Fixed ColorDescriptor implementation use of `pyflann.FLANN.nn_index` when the distance method is "hik" (inverted results order and distance values).
- Fixed ColorDescriptor `is_usable` check to catch stdout/stderr output.

Nearest Neighbors

- Fixed issue with FLANN implementation where containing directories for output files were not being created first.

Relevancy Index

- Fixed bug in `LibSvmHikRelevancyIndex` where negative distance values would cause an error.

IQR Utils

- Fixed incorrect default `RelevancyIndex` configuration.

Tests

- Fixed tests due to DataSet implementation split

Tools / Scripts

- Fixed various bugs in compute scripts

Miscellaneous

- Removed various unnecessary print statements originally for debugging.
- Removed redundant uses of metaclass declarations.

5.2.2 SMQTK v0.2.1 Release Notes

This is a minor release with a necessary bug fix for installing SMQTK. This release also has a minor documentation update regarding Caffe AlexNet default model files and how/where to get them.

Updates / New Features since v0.2

Documentation

- Added segment on acquiring necessary Caffe model files for use with the current caffe wrapper implementation.

Fixes since v0.2

Build

- Fix an issue where the CMake was trying to install directories no longer in the source tree due to earlier removal.

5.2.3 SMQTK v0.2.2 Release Nodes

This minor release primarily adds classifier algorithm and classification representation support, a new service web application for nearest-neighbors algorithms, as well as additional documentation.

Also, this release adds a few more command line tools, especially of note is `iqrTrainClassifier.py` that can train a classifier based on the saved state of the IQR demonstration application (also a new feature).

Updates / New Features since v0.2.1

Classifiers

- Added generic Classifier algorithm interface.
- Added SupervisedClassifier intermediate interface.

Classification Elements

- Added classification result encapsulation interface.
- Added in-memory implementation
- Added PostgreSQL implementation
- Added file-based implementation
- Added ClassificationElementFactory implementation.

Data Elements

- Added DataFileElement implementation the optional use of the tika module for file content type extraction. Falls back to previous method when tika module not found or fails.

Descriptor Elements

- Moved additional implementation specific documentation into `docs/` directory.
- Moved additional implementation specific configuration and example files into `etc/smqtk/`.
- Moved `PostgresDescriptorElement` implementation out of nested sub-module into a single module in implementations directory.

Descriptor Generators

- Removed `PARALLEL` class variable (parameterized in pertinent implementation constructors).
- Added `CaffeDescriptorGenerator` implementation, which is more generalized and model agnostic, using the Caffe python interface.

Documentation

- Added web-service documentation directory and moved applicable documentation files there.
- Added more/better documentation on IQR demonstration application.
- Added documentation on saving IQR state and training/using a supervised classifier based on it.

Tools / Scripts

- Added descriptor compute script that reads from a file-list text file specifying input data file paths, and asynchronously computes descriptors. Uses JSON configuration file for algorithm and element backend specification.
- Added tool for training a supervised classifier based on an IQR session state.
- Added tool for classifying a sequence of input file paths, outputting paths that classified as the input label (highest confidence).
- Converted `iqr_app_model_generation.py` to run as a command line tool with arguments, rather than an example script.

Web / Services

- Added `NearestNeighborServiceServer`, which provides web-service that returns the nearest *N* neighbors to the given descriptor element.

- Added ability to save IQR state via a new button in web interface. This file is used with the IQR classifier training script.

Fixes since v0.2.1

Custom LibSVM

- Fixed an issue where `svm_save_model` would crash when saving a 2-class SVM model.
- Fixed an issue where `svm_save_model` would save an extra, unexpected file when saving a 2-class SVM model.

Descriptor Elements

- Fix threading joining in `elements_to_matrix` (when using non-multiprocessing mode).
- Fixed configuration use in `DescriptorElementFactory.from_config`.

Data Sets

- Removed `is_usable` abstract method. Redundant with `Pluggable` base class.

Docs

- Made `sphinx_server.py` executable.
- Fixed whitespacing issue with `docs/algorithms.rst` that prevented display of ToC sections.
- Updated/Fixed various class/function doc-strings.

Utils

- Fixed `smqtk.utils.plugin.get_plugins` to handle skipping intermediate interfaces between the base class and implementation classes, as well as to skip implementation classes that do not fully implement abstract methods.

5.2.4 SMQTK v0.3.0 Release Notes

This minor release primarily adds a new modular LSH nearest-neighbor index algorithm implementation. This new implementation strictly replaces the now deprecated and removed `ITQNearestNeighborsIndex` implementation because of its increased modularity and flexibility. The old `ITQNearestNeighborsIndex` implementation had been hard-coded and its previous functionality can be reproduced with the new implementation (`ItqFunctor` + `LinearHashIndex`).

The `CodeIndex` representation interface has also been depricated as its function has been replaced by the combination of the `LSHNearestNeighborIndex` implementation.

Updates / New Features since v0.2.2

CodeIndex

- Deprecated/Removed because of duplication with `DescriptorIndex`, `HashIndex` and LSH algorithm.

Custom LibSVM

- Fix compiler error on Windows with Visual Studio < 2013. Log2 doesn't exist until that VS version. Added stand-in.

DescriptorIndex

- Added initial Solr backend implementation.

Documentation

- Updated documentation to references to `CodeIndex` and update references to `ITQNearestNeighborsIndex` to `LSHNearestNeighborIndex`.

HashIndex

- Added new `HashIndex` algorithm interface for efficient neighbor indexing of hash codes (bit vectors).
- Added linear (brute force) implementation.
- Added ball-tree implementation (uses `sklearn.neighbors.BallTree`)

LshFunctor

- Added new interface for LSH hash code generation functor.
- Added ITQ functor (replaces old `ITQNearestNeighborsIndex` functionality).

NearestNeighborIndex

- Added generalized LSH implementation: `LSHNearestNeighborIndex`, which uses a combination of `LshFunctor` and `HashIndex` for modular assembly of functionality.
- Removed deprecated `ITQNearestNeighborsIndex` implementation (reproducible using the new `LSHNearestNeighborIndex` with `ItqFunctor` and `LinearHashIndex`).

Tests

- Added tests for DescriptorIndex abstract and in-memory implementation.
- Removed tests for deprecated `CodeIndex` and `ITQNearestNeighborsIndex`
- Added tests for `LSHNearestNeighborIndex` + high level tests using ITQ functor with linear and ball-tree hash indexes.

Tools / Scripts

- Added optional global default config generation to `summarizePlugins.py`
- Updated `summarizePlugins.py`, removing `CodeIndex` and adding `LshFunctor` and `HashIndex` interfaces.

Utilities

- Added `cosine_distance` function (inverse of `cosine_similarity`)
- Updated `compute_distance_kernel` to be able to take `numba.jit` compiled functions

Web / Services

- Added query sub-slice return option to NearestNeighborServiceServer web-app.

Fixes since v0.2.2

DescriptorElement

- Fixed mutibility of stored descriptors in DescriptorMemoryElement implementation.

Tools / Scripts

- Added `Classifier` interface plugin summarization to `summarizePlugins.py`.

Utilities

- Fixed bug with `smqtk.utils.bit_utils.int_to_bit_vector[_large]` when give a 0-value integer.

Web / Services

- Fixed issue with IQR alerts not showing whitespace correctly.
- Fixed issue with IQR reset not resetting everything, which caused the application to become unusable.

5.2.5 SMQTK v0.4.0 Release Notes

This is a minor release that provides various minor updates and fixes as well as a few new command-line tools and a new web service application.

Among the new tools include a couple classifier validation scripts for checking the performance of a classification algorithm fundamentally as well as against a specific test set.

A few MEMEX program specific scripts have been added in a separated directory, defining an ingestion process from an ElasticSearch instance through descritpor and hash code computation.

Finally, a new web service has been added that exposes the IQR process for external tools. The existing IQR demo web application still functions as it did before, but does not yet use this service under the hood.

Updates / New Features since v0.3.0

Classifiers

- Updated supervised classifier interface to no assume presence of a "negative" class.
- Fixed libSVM implementation train method to not assume "negative" class.

Compute Functions

- Refactored `compute_many_descriptors.py` main work function into a new sub-module of SMQTK in in order to allow higher level compute function to be accessible from the SMQTK module API.
- Added function for asynchronously computing LSH codes for some number of input descriptor elements.

Descriptor Index

- Update to postgresql backend to lazy-connect during batch executions, preventing a connection from being made if nothing is being added.

Documentation

- Added `CONTRIBUTING.md` file.
- Added example of setting up a NearestNeighborServiceServer with live-reload enabled and how to add/process incremental ingests.

IQR

- Revised IqrSession class for generalized use (pruned down attributes to what is needed). Fixed IqrSearchApp due to changes.

Tools / Scripts

- Added CLI script for hash code generation and output to file. This script is primarily for support of LSHNearestNeighborIndex live-reload functionality.
- Added script for asynchronously computing classifications on descriptors in an index via a list of descriptor UUIDs.
- Added script for cross validating a classifier configuration for some truthed descriptors within an index. Can generate PR and ROC curves.

- Added some MEMEX specific scripts for processing and updating data from a known Solr index source.
- Added MEMEX-specific script for fetching image data from an ElasticSearch instance and transfering it locally.
- Added script for validating a classifier implementation with a model against a labeled set of descriptors. This script can also be used to conveniently train a classifier if it is a supervised classifier type.

Utilities

- Added helper wrapper for generalized asynchronous function mapping to an input stream.
- Added helper function for loop progress reporting and timing.
- Added helper function for JSON configuration loading.
- Added helper for utilities, encapsulating standard argument parser and configuration loading/generation steps.
- Renamed "merge_config" to "merge_dict" and moved it to the smqtk.utils module level.

Web

- Added IQR mostly-RESTful service application. Comes with companion text file outlining web API.

Fixes since v0.3.0

ClassificationElement

- Fixed memory implementation serialization bug.

HashIndex

- Fixed SkLearnBallTreeHashIndex model load/save functions to not use pickle due to save size issues. Now uses `numpy.savez` instead, providing better serialization and run time.

5.2.6 SMQTK v0.5.0 Release Notes

This is a minor release that provides minor updates and fixes as well as a new Classifier implementation, new parameters for some existing algorithms and added scripts that were the result of a recent hackathon.

The new classifier implementation, the `IndexLabelClassifier`, was created for the situation where the resultant vector from DescriptorGenerator is actually classification probabilities. An example where this may be the case is when a CNN model and configuration for the Caffe implementation yields a class probability (or Softmax) layer.

The specific scripts added from the hackathon are related to classifying entities based on associated image content.

Updates / New Features since v0.4.0

Classifier

- Added classifier that applies a list of text labels from file to vector from descriptor as if it were the classification confidence values.

Descriptor Generators

- Added `input_scale` pass-through option in the Caffe wrapper implementation.
- Added default descriptor factory to yield in-memory descriptors unless otherwise instructed.

Descriptor Index

- Added warning logging message when PostgreSQL implementation file fails to import the required python module.

libSVM

- Tweaked some default parameters in grid.py

LSH Functors

- Added descriptor normalization option to ITQ functor class.

Scripts

- Added new output features to classifier model validation script: confusion matrix and ROC/PR confidence interval.
- Moved async batch computation scripts for descriptors, hash codes and classifications to `bin/`.
- Added script to transform a descriptor index (or part of one) into the file format that libSVM likes: `descriptors_to_svmtrainfile.py`
- Added script to distort a given image in multiple configurable ways including cropping and brightness/contrast transformations.
- Added custom scripts resulting from MEMEX April 2016 hackathon.
- Changed MEMEX update script to collect source ES entries based on crawl time instead of insertion time.

Utilities

- Added async functionality to kernel building functions

Fixes since v0.4.0

CMake

- Removed `SMQTK_FIRST_PASS_COMPLETE` stuff in root CMakeLists.txt

Scripts

- Changed `createFileIngest.py` so that all specified data elements are added to the configured data set at the same time instead of many additions.

5.2.7 SMQTK v0.6.0 Release Notes

This minor release provides bug fixes and minor updates as well as Docker wrapping support for RESTful services, a one-button Docker initialization script for a directory of images, and timed IQR session expiration.

The `docker` directory is intended to host container Dockerfile packages as well as other associated scripts relating to docker use of SMQTK. With this release, we provide a Dockerfile, with associated scripts and default configurations, for a container that hosts a Caffe install for descriptor computation, replete with AlexNet model files, as well as the NearestNeighbor and IQR RESTful services. This container can be used with the `docker/smqtk_services.run_images.sh` for image directory processing, or with existing model files and descriptor index.

The IQR Controller class has been updated to optionally time-out sessions and clean itself over time. This is required for any service that is going to stick around for any substantial length of time as resources would otherwise build up and the host machine would run out of RAM.

Updates / New Features since v0.5.0

CMake

- Added scripts that were missing from install command.

Descriptor Index

- Changed functions that used to take `*uuids` list expansion as an argument and changed them to take iterables, which no longer causes sequencification of input iterables and is already compatible with all included implementations except Solr.
- Update Solr implementation functions that used to take `*uuid` list expansion to properly handle input iterables of arbitrary sizes.
- DescriptorIndex instances, when iterated over, now yield DescriptorElement instances instead of just the UUID keys.

Docker

- Added docker container formula for running SMQTK NearestNeighbor and IQR services.
- Added a script to setup SMQTK web services over a directory of images, performing all necessary Docker setup and data processing. This is intended for demo purposes only and not for large scale processing.

IQR

- Added optional session expiration feature to `IqrController` class to allow for long-term self clean-up.

Nearest Neighbors Index

- Changed ITQ fit method to by default use multiprocessing over threading, which in general is faster (more through-put).

Utilities

- Removed by-index access in `elements_to_matrix`, allowing arbitrary input as long as the __len__ and __iter__ functions are defined.
- Changed much of the debug messages in `smqtk.utils.parallel` to "trace" level (level 1).

Scripts

- Simplified the `train_itq.py` script a little.

Web Apps

- Added configuration of IqrController session expiration monitoring to IQR RESTful (ish) service.

Fixes since v0.5.0

Descriptor Index

- Fixed PostgreSQL back-end bug when iterating over descriptors that caused inconsistent/duplicate elements in iterated values.

IQR

- Fixed how IqrController used and managed session UUID values.

Utilities

- Fixed bug in `int_to_vector` functions dealing with vector size estimation.

Web Apps

- Fixed bugs in IQR classifier caching and refreshing from dirty state
- Fixed how the NearestNeighbor service descriptor computation method errors regarding descriptor retrieval in order to not obfuscate the error.

5.2.8 SMQTK v0.6.1 Release Notes

This is a patch release with bug fixs for the Docker wrapping of RESTful services introduced in v0.6.0.

Fixes since v0.6.0

Docker

- Fixed issue where *smqtk_services.run_images.sh* wasn't properly pulling containers from Dockerhub.
- Fixed typo in default configuration files installed into the container.
- Fixed IQR service function layout to be more explicit in errors caught and raised which maintaining thread safety.

5.2.9 SMQTK v0.6.1 Release Notes

This is a patch release with a bug fix for Caffe descriptor generation introduced in v0.6.0.

Fixes since v0.6.0

Descriptor Generation

- Fixed bug in Caffe wrapper image array loading where loaded arrays were not in the correctly associated with data identifiers.

5.2.10 SMQTK v0.7.0 Release Notes

This minor release incorporates various fixes and enhancements to representation and algorithms interfaces and implementations.

A new docker image has been added to wrap the IQR web interface and headless services. This image can either be used as a push-button image ingestion and IQR interface container, or as a fully feature environment to play around with SMQTK, Caffe deep-learning-based content description and IQR.

A major departure has happened for some representation structures, like DataElements, as they are no longer considered hashable and now have interfaces reflecting their mutability. Representation structures, by their nature of having arbitrary backends, may be modifiable my external agents interacting in a separate manner with the backend being used. This has also opened up the ability to provide algorithm implementations with DataElement instances instead of filepaths for desired byte content and many implementations have transitioned over to using this pattern. There is nothing fundamentally wrong with requesting file-path input, however it is restricting as to where configuration files or data models may come from.

Updates / New Features since v0.6.2

Algorithms

Descriptor Generators

- Added KWCNN DescriptorGenerator plugin

Build System

- Added `setup.py` script in support of installation by `pip`. Updated CMake code to install python components via this scripts.
- Added `SMQTK_BUILD_FLANN` and `SMQTK_BUILD_LIBSVM` to CMake for optionally building libSVM and Flann (both default ON).

Classifier Interface

- Added default `ClassificationElementFactory` that uses the in-memory back-end.

Compute Functions

- Added minibatch kmeans based descriptor clustering function with CLI interface.

Descriptor Elements

- Revised implementation of in-memory representation, doing away with global cache.
- Added optimization to Postgres backend for a slightly faster `has_vector` implementation.

Descriptor Generator

- Removed lingering assumption of `pyflann` module presence in `colordescriptor.py`.

Devops::Ansible

- Added initial Ansible roles for SMQTK and Caffe dependency.

Devops::Docker

- Revised default IQR service configuration file to take into account recently added session expiration support. Defaults were used before, but now it needs to be specifically enabled as by default expiration is not enabled.
- Added IQR / playground docker container setup. Includes: - CPU + NVIDIA GPU capable docker file. - Optional input image tiling. - Optional startup of RESTfule NN and IQR services.

Documentation

- Updated build and installation documentation.
- Added missing utility script documentation hooks.
- Standardized utility script definition of argument parser generation function for documentation use.

Girder

- Added initial simple Girder plugin to link to an external IQR webapp instance.

Misc.

- Added algo/rep/iqr imports to top level `__init__.py` to make basic functionality available without special imports.

Representation

Data Elements

- Added plugin for Girder-hosted data elements

- Added `from_uri` member function as well as global function to handle instance construction or selection via URI string specification.
- Postgres data element will now automatically create its configured table if it doesn't exist and authentication and sufficient privileges.

Descriptor Element

- Postgres descriptor element will now automatically create its configured table if it doesn't exist and authentication and sufficient privileges.

Descriptor Index

- Postgres descriptor index will now automatically create its configured table if it doesn't exist and authentication and sufficient privileges.

Scripts

- Add script to conveniently make Ball-tree hash index model given an existing `hash2uuids.pickle` model file required for the `LSHNearestNeighborsIndex` implementation.
- `compute_many_descriptor.py` batch size parameter now defaulted to 0 instead of 256.
- Add script to cluster an index of descriptors via mini-batch kmeans (scikit-learn).
- Added script wraping the use of the mini-batch kmeans descriptor clustering function.
- Added scripts and notebooks for retrieving MEMEX-specific data from ElasticSearch.
- Moved-command line scripts to the *smqtk.bin* sub-module in order to use `setuptool` support for cross-platform executable generation.
- `classifier_kfold_validation` utility now only uses MemoryClassificationElement instead of letting it be configurable.
- Added script for finding nearest neighbors of a set of UUIDs given a nearest neighbors index.
- Added script to add GirderDataElements to a data set

Utilities

- Started a module containing URL-base utility functions, initially adding a url-join function similar in capability to `os.path.join`.
- Added fixed tile cropping to image transform tool.
- Added utility functions to detect mimetypes of files via `file-magic` or `tika` optional dependencies.

Web

- Updated/Rearchitected IqrSearchApp (now IqrSearchDispatcher) to be able to spawn multiple IQR configurations during runtime in addition to any configured in the input configuration JSON file. This allows external applications to manage configuration storage and generation.
- Added directory for Girder plugins and added an initial one that, given a folder with the correct metadata attached, can initialize an IQR instance based on that configuration, and then link to IQR web interface (uses existing/updated IqrSearch web app).
- Added ability to automatically login via a valid Girder token and parent Girder URL for token/user verification. This primarily allows restricted external IQR instance creation and automatic login from Girder redirects.
- Mongo session information block at bottom IQR app page now only shows up when running server in debug mode.
- Added document showing complete use case with IQR RESTful webservice using the IQR docker image with LEEDS Butterfly data. Includes expected results users should be able to replicate.

Fixes since v0.6.2

Documentation

- Fixed issues caused by moving scripts out of *./bin/* to *./python/smqtk/bin*.

Scripts

- Fix logging bug in *compute_many_descriptors.py* when file path has unicode in it.
- Removed final loop progress report from *compute_many_descriptors.py* as it did not report valid statistics.
- Fixed deprecated import of `flask.ext.basicauth` module.
- Fixed DescriptorFileElement cache-file save location directory when configured to use subdirectories. Now no longer creates directories to store only a single file. Previous file-element roots are not compatible with this change and need to be re-ingested.
- Fixed IQR web app url prefix check

Metrics

- Fixed cosine distance function to return angular distance.

Utilities

- `SmqtkObject` logger class accessor name changed to not conflict with `flask.Flask` logger instance attribute.

Web

- Fixed Flow upload browse button to not only allow directory selection on Chrome.

5.2.11 SMQTK v0.8.0 Release Notes

This minor release represents the merger of a public release that added a Girder-based implementation of the DataElement interface. We also optimized the use of the PostgreSQL DescriptorIndex implementation to use named cursors for large queries.

Updates / New Features since v0.7.0

Data Structures

- Revise *GirderDataElement* to use *girder_client* python module and added the the use of girder authentication token values in lieu of username/password for communication authorization.
- Add the optional use of named cursors in PostgreSQL implementation of the *DescriptorIndex* interface. Assists with large selects such that the server only sends batches of results at a time instead of the whole result pool.
- Added PostgreSQL implementation of the KeyValueStore interface.

Girder

- Initial SMQTK Girder plugin to support image descriptor processing via girder-worker.
- Initial SMQTK Girder plugin implementing a resource and UI for SMQTK nearest neighbors and IQR.

Fixes since v0.7.0

Data Structures

- Added locking to PostgreSQL *DescriptorElement* table creation to fix race condition when multiple elements tried to create the same table at the same time.
- Fix unconditional import of optional *girder_client* dependency.

Dependencies

- Pinned Pillow version requirement to 4.0.0 due to a large-image conversion issue that appeared in 4.1.x. This issue may have been resolved in newer versions of Pillow.

Scripts

- Various fixes to IQR model generation process due to changes made to algorithm input parameters (i.e. taking *DataElement* instances instead of filepaths).
- Fixes *build_iqr_models.sh* to follow symlinks when compiling input image file list.

Tests

- Fix missing abstract function override in KeyValueStore test stub.
- Fix test girder_client.HttpError import issue.

5.2.12 SMQTK v0.8.1 Release Notes

This patch release addresses a bug with PostgreSQL implementations incorrectly calling a helper class.

Fixes since v0.8.0

Descriptor Index Plugins

- Fix bug in PostgreSQL plugin where the helper class was not being called appropriately.

Utilities

- Fix bug in PostgreSQL connection helper where the connection object was being called upon when it may not have been initialized.

5.2.13 SMQTK v0.9.0 Release Notes

This minor release represents an update to supporting python 3 versions as well as adding connection pooling support to the PostgreSQL helper class.

Updates / New Features since v0.8.1

General

- Added support for Python 3.
- Made some optimizations to the Postgres database access.

Travis CI

- Removed use of Miniconda installation since it wasn't being utilized in special way.

Fixes since v0.8.1

Tests

- Fixed ambiguous ordering check in libsvm-hik implementation of RelevancyIndex algorithm.

5.2.14 SMQTK v0.10.0 Release Notes

This minor release represents the merger of public release request 88ABW-2018-3703. This large updated adds a number of functionality improvements and API changes, docker image improvements and expansions (see the new classifier service), FAISS algorithm wrapper improvements, `NearestNeighborIndex` update and removal support, a switch to `py.test` testing framework, generalized classification probability adjustment function, code clean-up, bug fixes and more.

Updates / New Features since v0.9.0

Algorithms

- Classifier
  - Added *ClassifierCollection* support class. This assists with aggregating multiple SMQTK classifier implementations and applying one or more of those classifiers to input descriptors.
  - Split contents of the *__init__.py* file into multiple component files. This file was growing too large with the multiple abstract classes and a new utility class.
  - Changed *classify* abstract method to raise a *ValueError* instead of a *RuntimeError* upon being given an empty *DescriptorElement*.
  - Updated SupervisedClassifier abstract interface to use the template pattern with the train method. Now, implementing classes need to define `_train`. The `train` method is not abstract anymore and calls the `_train` method after the input data consolidation.
  - Update API of classifier to support use of generic extra training parameters.
  - Updated libSVM classifier algorithm to weight classes based on the geometric mean of class counts divided by specific class count to more properly handle weighting even if there is class imbalance.
- Hash Index
  - Made to be its own interface descending from *SmqtkAlgorithm* instead of *NearestNeighborsIndex*. While the functionality of a NN-Index and a HashIndex are very similar, all method interfaces are different in terms of the types they accept and return and the HashIndex implementation redefined and documented them to the point where there was no shared functionality.
  - Switched to using the template method for abstract methods.
  - Add update and remove methods to abstract interface. Implemented new interface methods in all subclasses.
  - Added model concurrency protection to implementations.
- Nearest-Neighbors
  - Switched to using the template method for abstract methods.
  - Add update and remove methods to abstract interface. Implemented new interface methods in all subclasses.
  - Fix imports in FAISS wrapper module.

  - Added model concurrency protection to implementations.
  - FAISS
    * Add model persistence via optionally provided *DataElement*.
    * Fixed use of strings for python 2/3 compatibility.
    * Changed default factory string to "IVF1,Flat".
    * Added initial GPU support to wrapper. Currently only supports one GPU with explicit GPU ID specification.

Representations

- Descriptor Index
  - Added *__contains__* method to abstract class to call the *has* method. This should usually be more efficient than scanning the iteration of the index which is what was happening before. For some implementations, at worst, the runtime for checking for inclusion will be the same (some implementations may *have* to iterate).
- Descriptor Element
  - Interface
    * Hash value for an element is now only composed of UID value. This is an initial step in deprecating the use of the type-string property on descriptor elements.
    * Equality check between elements now just vector equality.
    * Added base implementation of *__getstate__* and *__setstate__*. Updated implementations to handle this as well as be backward compatible with their previous serialization formats.
    * Added a return of self to vector setting method for easier in-line setting after construction.
  - PostgreSQL
    * Updated to use PsqlConnectionHelper class.
- KeyValueStore
  - Added *remove* and *remove_many* abstract methods to the interface. Added implementations to current subclasses.
  - Added *__getitem__* implementation.

Docker

- Caffe
  - Updated docker images for CPU or GPU execution.
  - Updated Caffe version built to 1.0.0.
- Added Classifier service docker images for CPU or GPU execution.
  - Inherits from the Caffe docker images.
  - Uses MSRA's ResNet-50 deep learning models.
- IQR Playground
  - Updated configuration files.
  - Now only runs IQR RESTful service and IQR GUI web app (removed nearest- neighbors service).
  - Simplified source image mount point to */images*.

  - Updated *run_container.*.sh* helper scripts.
  - Change deep-learning model used from AlexNet to MSRA's RestNet-50 model.
- Versioning changes to, by default, encode date built instead of arbitrary separate versioning compared to SMQTK's versioning.
- Classifier and IQR docker images now use the local SMQTK checkout on the host system instead of cloning from the internet.

IQR module

- Added serialization load/save methods to the *IqrSession* class.

Scripts

- *generate_image_transform*
  - Added stride parameter to image tile cropping feature to allow for more than just discrete, abutting tile cropping.
- *runApplication*
  - Add ability to get more than individual app description from providing the *-l* option. Now includes the title portion of each web app's doc-string.
- Added *smqtk-make-train-test-sets*
  - Create train/test splits from the output of the *compute_many_descriptors* tool, usually for training and testing a classifier.

Testing

- Remove use of *nose-exclude* since there are now actual tests in the web sub-module.
- Switch to using *pytest* as the test running instead of *nose*. Nose is now in "maintenance mode" and recommends a move to a different testing framework. Pytest is a popular a new powerful testing framework alternative with a healthy ecosystem of extensions.
- Travis CI
  - Removed use of Miniconda installation since it wasn't being utilized in special way.
- Added more tests for Flask-based web services.

Utilities module

- Added mimetype utilities sub-module.
- Added a web utilities module.
  - Added common function for making response Flask JSON instances.
- Added an *iter_validation* utility submodule.
- Plugin utilities
  - Updated plugin discovery function to be more descriptive as to why a module or class was ignored. This helps debugging and understanding why an implementation for an interface is not available at runtime.
- PostgreSQL
  - Added locking to table creation upsert call.
- Added probability utils submodule and initial probability adjustment function.

Web

- Added new classifier service for managing multiple SMQTK classifier instances via a RESTful interface as well as describe arbitrary new data with the stored classifiers. This service also has the ability to take in saved IQR session states and train a new binary classifier from it.
  - Able to query the service with arbitrary data to be described and classified by one or more managed classifiers.
  - Able to get and set serializations of classifier models for archival.
  - Added example directory of show how to run and to interact with the classifier service via *curl*.
  - Optionally take a new parameter on the classify endpoint to adjust the precision/recall balance of results.
- IQR Search Dispatcher (GUI web app)
  - Refactored to use RESTful IQR service.
  - Added GUI and JS to load an IQR state from file.
  - Update sample JSON configuration file at *python/smqtk/web/search_app/sample_configs/config.IqrSearchApp.json*.
  - Added */is_ready* endpoint for determining that the service is alive.
- IQR service
  - Added ability to an IQR state serialization into a session.
  - Added sample JSON configuration file to *python/smqtk/web/search_app/sample_configs/config.IqrRestService.json*.
  - Added */is_ready* endpoint for determining that the service is alive.
  - Move class out of the *__init__.py* file and into its own dedicated file.
  - Make IQR state getter endpoint return a JSON containing the base64 of the state instead of directly returning the serialization bytes.
  - Added endpoints to update, remove from and query against the global nearest-neighbors index.

Fixes since v0.9.0

Algorithms

- Nearest-Neighbor Index
  - LSH
    - Fix bug where it was reporting the size of the nested descriptor index as the size of the neighbor index when the actual index state is defined by the hash-to-uids key-value mapping.

Representations

- DataElement
  - Fixed bug where *write_temp()* would fail if the *content_type()* was unknown (i.e. when it returned *None*).
- Descriptor Index
  - PostgreSQL
    - Fix bug where an instance would create a table even though the *create_table* parameter was set to false.
- Descriptor Elements
  - PostgreSQL implementation
    - Fix set_vector method to be able to take in sequences that are not explicitly numpy arrays.

- KeyValue
  - PostgreSQL
    - Fix bug where an instance would create a table even though the *create_table* parameter was set to false.

Scripts

- *classifier_model_validation*
  - Fixed confidence interval plotting.
  - Fixed confusion matrix plot value range to the [0,1] range which causes the matrix colors to have meaning across plots.

Setup.py

- Add *smqtk-* to some scripts with camel-case names in order to cause them to be successfully removed upon uninstallation of the SMQTK package.

Tests

- Fixed ambiguous ordering check in libsvm-hik implementation of RelevancyIndex algorithm.

Web

- IQR Search Dispatcher (GUI web app)
  - Fix use of *StringIO* to using *BytesIO*.
  - Protect against potential deadlock issues by wrapping intermediate code with try/finally clauses.
  - Fixed off-by-one bug in javascript *DataView* construction.
- IQR Service
  - Gracefully handle no-positive-descriptors error on working index initialization.
  - Fix use of *StringIO* to using *BytesIO*.

5.2.15 SMQTK v0.11.0 Release Notes

This minor release includes a number of security and stability fixes for algorithms and the IQR demo web application.

Updates / New Features since v0.10.0

Documentation

- Updated IQR Demo Application documentation RST file and images to reflect the current state of SMQTK and that process.

Fixes since v0.10.0

Algorithms

- Classifiers
  - SVM
    - Fixed broken large model saving in Python 2, creating parity with Python 3.
- Nearest-Neighbors
  - FAISS
    - Fixed use of strings for compatibility with Python 2.
    - Fixed broken large model saving in Python 2, creating parity with Python 3.
  - FLANN
    - Fixed broken large model saving in Python 2, creating parity with Python 3.
  - Hash Index
    - Scikit-Learn BallTree
      - Fix `save_model` and `load_model` methods for additional compatibility with scikit-learn version 0.20.0.
  - LSH
    - Fix issue with update and remove methods when constructed with a key-value store structure that use the `frozenset` type.
    - Fix issue with on-the-fly linear hash index build which was previously not correctly setting a set of integers.

Descriptor Generator Plugins

- Fix issue with `CaffeDescriptorGenerator` where the GPU would not be appropriately used on a separate thread/process after initialization occurs on the main (or some other) thread.

Docker

- IQR Playground
  - Updated README for better instruction on creating the docker image first.
- Caffe image
  - Resolved an issue with upgrading pip for a newer version of matplotlib.

Documentation

- Removed module mocking in sphinx `conf.py` as it has been shown to be brittle to changes in the source code. If we isolate and document a use-case where the mocking becomes relevant again we can bring it back.

Misc.

- Update requests and flask package version in `requirements.txt` and `devops/docker/smqtk_wrapper_python/requirements.txt` files due to GitHub security alert.
- Updated package versions for packages in the `requirements.docs.txt` requirements file.

Utilities

- Fixed broken large file writing in Python 2, creating parity with Python 3.
- Fixed `iqr_app_model_generation.py` script for the current state of SMQTK functionality.

- Fixed double logging issue in `python/smqtk/bin/classifyFiles.py` tool.

Web

- IQR Search Demo App
  - Fixed input element autocomplete property value being set from disabled" to the correct value of "off".
  - Fix CSRF vulnerability in demo web application front-end.
  - Fixed sample configuration files for the current state of associated tools.

5.2.16 SMQTK v0.12.0 Release Notes

This minor release includes minor fixes and known dependency version updates.

Fixes

Docker

- Fix issue with IQR playground image where matplotlib was attempting to use the TkAgg backend by default by adding a `matplotlibrc` file to specify the use of the `Agg` backend.

Misc

- Update requirements versions for: Flask, Flask-Cors, Pillow
- Update Travis-CI configuration to assume less default values.

Web

- IQR Service
  - Broaden base64 parsing error catch. Specific message of the error changed with python 3.7.

5.2.17 SMQTK v0.13.0 Release Notes

This release incorporates updates and fixes performed on the VIGILANT project and approved for public release (case number 88ABW-2019-5287). Some of the major updates and fixes in this include:

- Object detection algorithm interface and supporting DetectionElement interface and implementations.
- Revised plugin implementation accessor via the mixin class instead what used to be manually implemented side-car functions for every interface. Also moved some configuration specific functions out of the plugin utility module and into a configuration utility submodule, where the `Configurable` mixin class has also moved to.
- Moves unit tests out of the installed SMQTK package and into a dedicated sub-directory in the repository.

Updates / New Features

Algorithms

- Added `ImageReader` algorithm interface
  - Added matrix reading short-cut if DataElement instance provided has a `matrix` attribute/property.
  - Added PIL (pillow) implementation with tests.
  - Added GDAL implementation with tests.
- Descriptor Generators

  - Change `CaffeDescriptorGenerator` constructor to take `DataElement` instances rather than URIs.

- HashIndex
  - SkLearnBallTreeHashIndex
    - Fixed numpy load call to explicitly allow loading pickled components due to a parameter default change in numpy version 1.16.3.
- Object Detection
  - Added initial abstract interface.
  - Added "ImageMatrixObjectDetector" interface for object detectors that specifically operate on image data and standardizes the use of an "ImageReader" algorithm to provide the pixel matrix as input.
- Nearest Neighbors
  - FAISS
    - Gracefully handle addition of duplicated descriptors to avoid making index unusable due to an unexpected external failure.
    - Make use of new `get_many` method of key-value stores to improve runtime performance.
    - Make use of new `get_many_vectors` classmethod of DescriptorElement to improve runtime performance.
  - LSH Hash Functor
    - Use `ProgressReporter` in itq to avoid bugs from deprecated `report_progress` function

Compute Functions

- Add `compute_transformed_descriptors` function to `compute_functions.py` for conducting searches with augmented copies of an image

Misc.

- Updated numpy version in requirements.txt to current versions. Also split versioning between python 2 and 3 due to split availability.
- Resolve python static analysis warnings and errors.

Representation

- Added `AxisAlignedBoundingBox` class for describing N-dimensional euclidean spatial regions.
- Added `DetectionElement` interface, and in-memory implementation, with associated unit tests.
- Added `DetectionElementFactory` class for factory construction of `DetectionElement` instances.
- Add use of `smqtk.utils.configuration.cls_conf_from_config_dict` and `smqtk.utils.configuration.cls_conf_to_config_dict` to appropriate methods in factory classes.
- Add `get_many` method to `KeyValueStore` interface class and provide an optimized implementation of it for the `PostgresKeyValueStore` implementation class.
- Add `get_many_vectors` classmethod for efficiently retrieving vectors from several descriptor elements at once
- Add efficient implementation of `_get_many_vectors` for Postgres descriptor elements.
- Updated `MemoryKeyValueStore.add_many` to use `dict.update` method instead of manually updating keys.
- Removed unnecessary method override in `DataFileElement`.

- Added `MatrixDataElement` representation that stores a `numpy.ndarray` instance internally, generating bytes on-the-fly when requested.
- `DataMemoryElement` now raises a TypeError if a non-bytes-line object is passed during construction or setting of bytes. Configuration mixin hooks have been updated to convert to and from strings for JSON-compliant dictionary input and output. Fixed various usages of DataMemoryElement to actually pass bytes.

Tests

- Moved tests out of main package tree.
- Added use of `pytest-runner` in `setup.py`, removing `run_tests.sh` script. New method of running tests is `python setup.py test`.

Utilities

- Added to `Pluggable` interface the `get_impls` method, replacing the separate `get_*_impls` functions defined for each interface type. Removed previous `get_*_impls` functions from algorithm and representation interfaces, adjusting tests and utilities as appropriate.
- Renamed `smqtk.utils.configurable` to `smqtk.utils.configuration`. Ramifications fixed throughout the codebase. Added documentation to doc-strings.
- Added `cls_conf_from_config_dict` and `cls_conf_to_config_dict` intermediate helper functions to `smqtk.utils.configuration` for the `from_config_dict` and `to_config_dict` subproblems, respectively. This was motivated by duplicated functionality in element factory class `from_config` and `get_config` methods.
- Moved some helper functions from `smqtk.utils.plugin``to ``smqtk.utils.configuration` as those functions more specifically had to do with configuration dictionary construction and manipulation. Ramifications fixed throughout the codebase.
- Updated `smqtk.utils.plugin.get_plugins` signature and return. Now more simply takes the interface class (previously referred to as the base-class) instead of the original first two positional, string arguments as they could be easily introspected from the interface class object. Ramifications fixed throughout the codebase.
- Added `ContentTypeValidator` interface for algorithms that operate on raw `DataElement` instances, providing methods for validating reported content types against a sub-class defined set of "valid" types. Applied to `DescriptorGenerator` interface.
- Replace usage of `smqtk.utils.bin_utils.report_progress` with the `ProgressReporter` class throughout package.
- Removed bundled "jsmin" in favor of using pip installed package.
- Moved `merge_dict` out of `smqtk/utils/__init__.py` and into its own module.
- Created `combinatorics` utils module, moved `ncr` function to here.
- Renamed various utility modules that included `_utils` in their name to not include `_utils` for the sake of reducing redundancy.
- Removed `FileModificationMonitor` utility class due to having no current use anywhere as well as its tests non-deterministically failing (issues with timing and probably lack of sufficient use of mock, time to fix not worth its lack of use). The `watchdog` python package should be used instead.
- Added entry-point extension method of plugin discovery.
- Added warning to `smqtk.utils.file.safe_file_write` when used on Windows platforms.

Fixes

Algorithms

- Nearest Neighbors
  - FAISS
    - Fix issue with storing and retrieving index IDs as numpy types by casting to python native integers due to an incompatibility with some KeyValueStore implementations (specificially an issue with the PostgreSQL implementation).

Misc.

- Removed some unused imports.

Representation

- Fixed bug with `ClassificationElement.max_label` where an exception would be raised if there was no label with associated confidence greater than 0.
- Fix some postgres test comparisons due to not being able to `byte` case Binary instances in python 3. Instead using the `getquoted` conversion for the sake of actual/expected comparisons.

Tests

- Moved `--cov` options from pytest.ini file into the runner script. This fixes debugger break-pointing in some IDEs (e.g. PyCharm).
- Fix various minor testing errors.

Utilities

- Fix `ZeroDivisionError` in `smqtk.utils.bin_utils.report_progress`. Also added deprecation warning to this function.

5.2.18 SMQTK v0.14.0 Release Notes

Notable updates with this release: * Simplification and vectorization of a few algorithm APIs. * New algorithm implementations and updates to existing ones. * Beginning to use `docker-compose` configuration to define the build

configurations of various images, beginning with an image to provide FAISS as a TPL dependency.

- Renamed `DescriptorIndex` to `DescriptorSet` to reduce confusion on implied functionality.

Notable fixes with this release: * Fixed issue with `smqtk.utils.parallel.parallel_map` to not hang on

keyboard interrupts.

Updates / New Features

Algorithms

- Classifier
  - Overhauled interface API to have the abstract method be a many-to-many iterator instead of the previous one-to-one signature.
  - Updated implementations and usages of this interface throughout SMQTK.
  - Added wrapper for scikit-learn LogisticRegression classifier.

- DescriptorGenerator
  - Overhauled interface API to have the abstract method be a many-to-many iterator instead of the previous one-to-one signature.
  - Updated colordescriptor implementation for interface API update.
  - Updated caffe implementation for interface API update.
  - Updated KWCNN implementation for interface API update.
- NearestNeighborsIndex
  - FAISS
    - Exposed `nprobe` parameter for when using IVF type indices to be utilized at query time.
- RelevancyIndex
  - Added `NoIndexError` exception for when attempting to perform ranking before an index is built.
  - Added `SupervisedClassifierRelevancyIndex` to enable using any available supervised classifier implementation to satisfy the RelevancyIndex API.

Compute Functions

- Updated `smqtk.compute_functions.compute_many_descriptors` to utilize new DescriptorGenerator API.

Docker

- Started use of docker-compose YAML file to organize image building.
- Added FAISS TPL image to be copied from by utilizing images.
- IQR "Playground"
  - Fixed compute test scripts to use updated DescriptorGenerator API.

Documentation

- Updated `docs/algorithminterfaces.rst` to reflect the new DescriptorGenerator API.
- Updated `docs/algorithmmodels.rst` to reflect the new DescriptorGenerator API.
- Updated the `docs/examples/caffe_build_index.rst` example to use the new DescriptorGenerator API.
- Updated the `docs/examples/simple_feature_computation.rst` example to use the new DescriptorGenerator API.

IQR

- Remove forcing of relevancy scores in `refine` when a result element is contained in the positive or negative exemplar or adjudication sets. This is because a user of an `IqrSession` instance can determine this intersection optionally outside of the class, so this forcing of the values is a loss of information.
- Added accessor functions to specific segments of the relevancy result predictions: positively adjudicated, negatively adjudicated and not-adjudicated elements.

Misc.

- Cleaned up various test warnings.

Representation

- AxisAlignedBoundingBox
  - Added `intersection` method.

- Data Element
  - Added PostgreSQL implementation.
- DataSet
  - Added PostgreSQL implementation, storing data representation natively in the database.
- DetectionElement
  - Added individual component accessors.
- Renamed "DescriptorIndex" to "DescriptorSet" in order to better represent what the structure and API represents. "Index" can carry the connotation that more is happening within the structure than actually is.

Tests

- Updated colordescriptor DescriptorGenerator tests to "skip" when deemed not available so that the tests are not just hidden when the optional dependencies are not present.
- Updated dummy classes used in classifier service unit tests to match the new DescriptorGenerator API.
- Update IQR service unit tests stub class for the new DescriptorGenerator API and iteration properties.
- Updated various class unit tests to make use of new configuration test helper function.
- Added a skip mark to `ContextualReadWriteLock` class unit tests which currently fail non-deterministically. This class is currently not used within SMQTK and a user-warning is now emitted when an attempted construction of this class occurs.

Tools / Scripts

- Updated the `smqtk.bin.classifyFiles` tool to use the new DescriptorGenerator API.
- Updated the `smqtk.bin.computeDescriptor` tool to use the new DescriptorGenerator API.
- Updated the `smqtk.bin.iqr_app_model_generation` tool to use the new DescriptorGenerator API.
- Updated some old MEMEX scripts to use the new DescriptorGenerator API.

Utils

- Added additional description capability to ProgressReporter.
- Added a return of self in the `ContentTypeValidator.raise_valid_element()` method.
- Added helper function for testing Configurable mixing instance functionality.
- Promoted service proxy helper class from IQR service server to a general web utility.
- Update random character generator to use `random.SystemRandom` which, at least for Posix systems, uses a source suitable for cryptographic purposes.
- Expanded debug logging enabling options in `runApplication` tool.
- Added `--use-simple-cors` option to the `runApplication` tool to enable CORS for all domains on all routes.

Web

- Added endpoints IQR headless service for expanded getter methods added to IqrSession class.
- Changed IQR web service endpoint to retrieve nearest-neighbors to a GET method instead of the previous POST method, as the previous method did not make sense for the request being made.
- Fixed usage of DescriptorGenerator instances in the classifier service for the API update.
- Updated `smqtk.web.descriptor_service` to use the new DescriptorGenerator API.

- Updated `smqtk.web.iqr_service` to use the new DescriptorGenerator API.
- Updated `smqtk.web.nearestneighbor_service` to use the new DescriptorGenerator API.

Fixes

Algorithms

- DescriptorGenerator
  - Caffe
    - Fix configuration overrides to correctly handle configuration from JSON.
    - Coerce unicode arguments to Net constructor to strings (or bytes in python 3).
    - Fixed numpy load call to explicitly allow loading pickled components due to a parameter default change in numpy version 1.16.3.
- HashIndex
  - SkLearnBallTreeHashIndex
    - Fixed numpy load call to explicitly allow loading pickled components due to a parameter default change in numpy version 1.16.3.
- ImageMatrixObjectDetector
  - Add `abstractmethod` decorator to intermediate implementation of `get_config` method.

Documentation

- Add missing reference to v0.13.0 change notes.

Tests

- Fixed PostgreSQL KeyValueStore implementation unit test that became non-deterministic in Python 3+.

Utilities

- Fixed issue with ProgressReporter when reporting before the first interval period.
- Fixed issue with `smqtk.utils.parallel.parallel_map` function where it could hang during threading-mode when a keyboard interrupt occurred.
- Fixed incorrectly calling the module-level debug logging function to use the locally passed logger, cleaning up a duplicate logging issue.

Web

- Classifier Service
  - Fix configuration of CaffeDescriptorGenerator.
- IQR Service
  - Fix configuration of CaffeDescriptorGenerator.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

R

S

T

U

V

W