# SMQTK-Classifier

***Release 0.19.0***

**Kitware, Inc.**

**Oct 07, 2021**

# CONTENTS

GitHub

This package provides interfaces and implementations around the classification of inputs into some form of labeled probabilistic values. Additional data structure abstractions are defined here to standardize this behavior into common terms.

CHAPTER

ONE

# INSTALLATION

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

CHAPTER

TWO

# CLASSIFIER INTERFACES

Here we list and briefly describe the high level algorithm interfaces which SMQTK-Classifier provides. Some implementations will require additional dependencies that cannot be packaged with SMQTK-Classifier.

## 2.1 ClassifyDescriptor

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

**class** smqtk_classifier.interfaces.classify_descriptor.**ClassifyDescriptor**(**args: Any*, ***kwargs: Any*)

Interface for algorithms that classify input descriptors into discrete labels and/or label confidences.

**static _assert_array_dim_consistency**(*array_iter: Union[numpy.ndarray, Iterable[numpy.ndarray]]*) → Union[numpy.ndarray, Iterable[numpy.ndarray]]

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 matrices, 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** **array_iter** (*collections.abc.Iterable[numpy.ndarray] | np.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: Union[numpy.ndarray, Iterable[numpy.ndarray]]*) → Iterator[Dict[Any, float]]

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.

*Remember:* A single-pass *Iterator* is a valid *Iterable*. If an implementation needs to pass over the input multiple times, either ensure you are receiving an ndarray, or *not* and Iterator.

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** – Iterable of arrays to be classified.

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

**classify_arrays**(*array_iter: Union[numpy.ndarray, Iterable[numpy.ndarray]]*) → Iterator[Dict[Any, float]]

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** **array_iter** (*collections.abc.Iterable[numpy.ndarray] | np.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.

**classify_elements**(*descr_iter: Iterable[smqtk_descriptors.interfaces.descriptor_element.DescriptorElement], factory: smqtk_classifier.classification_element_factory.ClassificationElementFactory = <smqtk_classifier.classification_element_factory.ClassificationElementFactory object>, overwrite: bool = False, d_elem_batch: int = 100*) → Iterator[*smqtk_classifier.interfaces.classification_element.ClassificationElement*]

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.

**classify_one_element**(*descr_elem: smqtk_descriptors.interfaces.descriptor_element.DescriptorElement*, *factory: smqtk_classifier.classification_element_factory.ClassificationElementFactory = <smqtk_classifier.classification_element_factory.ClassificationElementFactory object>*, *overwrite: bool = False*) → *smqtk_classifier.interfaces.classification_element.ClassificationElement*

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**() → Sequence[collections.abc.Hashable]

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.

**Raises** **RuntimeError** – No model loaded.

## 2.2 ClassifyDescriptorSupervised

This interface is a class of classifiers that are trainable via supervised training.

**class** smqtk_classifier.interfaces.classify_descriptor_supervised.**ClassifyDescriptorSupervised**(**args: Any*, ***kwargs: Any*)

Class of classifiers that are trainable via supervised training, i.e. are given specific descriptor examples for class labels.

**abstract has_model**() → bool

**Returns** If this instance currently has a model loaded. If no model is present, classification of descriptors cannot happen (needs to be trained).

**train**(*class_examples: Mapping[collections.abc.Hashable, Iterable[smqtk_descriptors.interfaces.descriptor_element.DescriptorElement]]*) → None

Train the supervised classifier model.

If a model is already loaded, we will raise an exception in order to prevent accidental overwrite.

If the same label is provided to both `class_examples` and `kwds`, the examples given to the reference in `kwds` will prevail.

**Parameters** **class_examples** – Dictionary mapping class labels to iterables of DescriptorElement training examples.

**Raises**

- **ValueError** – There were no class examples provided.
- **ValueError** – Less than 2 classes were given.
- **RuntimeError** – A model already exists in this instance. Following through with training would overwrite this model. Throwing an exception for information protection.

## 2.3 ClassifyImage

This interface represents algorithms that classify image instances into discrete labels or label confidences. The Images are formatted as `np.ndarray`.

**class** smqtk_classifier.interfaces.classify_image.**ClassifyImage**(**args: Any*, ***kwargs: Any*)

Interface for algorithms that classify input images into discrete labels and/or label confidences. Images are expected to be formatted in the format of *np.ndarray* matrices.

**abstract classify_images**(*img_iter: Union[numpy.ndarray, Iterable[numpy.ndarray]]*) → Iterator[Dict[Any, float]]

Classify an input iterable of images, in the form of *np.ndarray* matricies into a parallel iterable of label-to-confidence mappings (dictionaries).

We expect input image matrices to come in either the *[H, W]* or *[H, W, C]* dimension formats.

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** – Iterable of images, as numpy arrays, to be classified.

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

**Returns** Iterator of dictionaries, parallel in association to the input images. Each dictionary should map labels to associated confidence values.

`abstract get_labels()` → Sequence[collections.abc.Hashable]
Get the sequence of class labels that this classifier can classify images into. This includes the negative or background label if the classifier embodies such a concept.

**Returns** Sequence of possible classifier labels.

**Raises** `RuntimeError` – No model loaded.

## 2.4 ClassifyImageSupervised

This interface defines a specialization of image classifiers that are trainable via supervised learning.

`class smqtk_classifier.interfaces.classify_image_supervised.ClassifyImageSupervised(*args: Any, **kwargs: Any)`
Class of classifiers that are trainable via supervised training, i.e. are given specific Image examples for class labels.

`abstract has_model()` → bool

**Returns** If this instance currently has a model loaded. If no model is present, classification of images cannot happen (needs to be trained).

`train(class_examples: Mapping[collections.abc.Hashable, Union[numpy.ndarray, Iterable[numpy.ndarray]]])` → None
Train the supervised classifier model.

If a model is already loaded, we will raise an exception in order to prevent accidental overwrite.

If the same label is provided to both `class_examples` and `kwds`, the examples given to the reference in `kwds` will prevail.

**Parameters** `class_examples` – Dictionary mapping class labels to iterables of Image training examples.

**Raises**

- `ValueError` – There were no class examples provided.
- `ValueError` – Less than 2 classes were given.
- `RuntimeError` – A model already exists in this instance. Following through with training would overwrite this model. Throwing an exception for information protection.

## 2.5 Classification Element

Data structure used by Classifier

**class** smqtk_classifier.interfaces.classification_element.**ClassificationElement**(**args: Any*, ***kwargs: Any*)

Classification result encapsulation.

Contains a mapping of arbitrary (but hashable) label values to confidence values (floating point in `[0,1]` range). If a classifier does not produce continuous confidence values, it may instead assign a value of `1.0` to a single label, and `0.0` to the rest.

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

Element equality based on classification labels and values, not the type or UUID.

Since this base class defines `__getstate__` and `__setstate__` methods implementing classes must also extend these methods to support serialization. These methods have been marked as abstract to facilitate this requirement.

**Parameters**

- **type_name** – Name of the type of classifier this classification was generated by.
- **uuid** – Unique ID reference of the classification

**classmethod from_config**(*config_dict: Dict*, *type_name: str*, *uuid: collections.abc.Hashable*, *merge_default: bool = True*) → C

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** – JSON compliant dictionary encapsulating a configuration.
- **type_name** – Name of the type of classifier this classification was generated by.
- **uuid** – Unique ID reference of the classification
- **merge_default** – Merge the given configuration on top of the default provided by `get_default_config`.

**Returns** Constructed instance from the provided config.

**abstract get_classification**() → Dict[Any, float]

Get classification result map, returning a label-to-confidence dict.

We do no place any guarantees on label value types as they may be represented in various forms (integers, strings, etc.).

Confidence values are in the [0,1] range.

**Raises** **NoClassificationError** – No classification labels/confidences yet set.

**Returns** Label-to-confidence dictionary.

**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

`abstract has_classifications()` → bool

**Returns** If this element has classification information set.

`max_label()` → collections.abc.Hashable

Get the label with the highest confidence.

**Note on type annotation:** We are using *Hashable* here instead of *CLASSIFICATION_VALUE_T* (*Any*) due to the detachment from the dictionary type. When attached, *Any* is effectively *Hashable* in context of dictionary keys. Here however we want to try to make sure the given value is applicable with a dictionary.

**Raises** `NoClassificationError` – No classification set.

**Returns** The label with the highest confidence.

`abstract set_classification(`*m: Optional[Mapping[Any, float]] = None*, ***kwds: float*`)` → Dict[Any, float]

Set the whole classification map for this element. This will strictly overwrite the entire label-confidence mapping (vs. updating it)

Label/confidence values may either be provided via keyword arguments or by providing a dictionary mapping labels to confidence values. Non-string labels must be provided via an input dictionary (`m` parameter).

NOTE TO IMPLEMENTORS: This abstract method will aggregate input into a single dictionary, checks that there is anything in it and return it. Thus, a `super` call should be made, which will return a dictionary.

**Parameters** `m` – New labels-to-confidence mapping to set.

**Raises** `ValueError` – The given label-confidence map was empty.

**Returns** Input/combined Hashable-to-float mapping as a new dictionary.

CHAPTER

THREE

# CLASSIFIER IMPLEMENTATIONS

Here we describe the implementations

CHAPTER

FOUR

# RELEASE PROCESS AND NOTES

## 4.1 Steps of the SMQTK Release Process

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

## 4.2 Release Notes

### 4.2.1 v0.15.0

#### Updates / New Features

CI

- Add Github actions workflow for CI.

Misc.

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

#### Fixes

- Update CI configurations to use Poetry.
- Fixing docs and adding RTD

### 4.2.2 v0.16.0

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

#### Updates / New Features

Dependencies

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

Testing

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

#### Fixes

### 4.2.3 v0.17.0

This minor release introduces the pending deprecation of the `Classifier` and the `SupervisedClassifier` interface names, subsequently renamed to `ClassifyDescriptor` and `ClassifyDescriptorSupervised`, respectively. We also introduce a new, image-specific interface pair: `ClassifyImage` and `ClassifyImageSupervised`. These interfaces follow closely to the API of the original descriptor-based interfaces but instead take in image matrices as the primary input to be described.

#### Updates / New Features

Interfaces

- Classifier and SupervisedClassifier split into ClassifyImage, ClassifyImageSupervised, ClassifyDescriptor, and ClassifyDescriptorSupervised
- Rather than just taking in Descriptors, the SMQTK-Classifier can work directly with both Descriptors and Images
- Standardized Image input to follow the format of numpy matrices

#### Fixes

### 4.2.4 v0.18.0

This minor release improves modularity in the classify interfaces by removing *extra_params* and includes the utilization of the *postgres* helpers from SMQTK-Dataprovider.

#### Updates / New Features

- Removed *extra_params* training parameter from *_train* functions because it affected the modularity of the interface.

Implementations

- Modified *smqtk_classifier.impls.classification_element.postgres* to use the helper from *smqtk_dataprovider.utils.postgres*.

#### Fixes

### 4.2.5 v0.19.0

#### Updates / New Features

- Update `smqtk-descriptors` requirement to the latest patched version.

#### Fixes

- Fix usage of `DescriptorElement` as per dep update.

CHAPTER

FIVE

# INDICES AND TABLES

- genindex
- modindex
- search

# INDEX

## Symbols



## C



## F



## G



## H



## M



## S



## T