gala Documentation

Release 0.5dev

Juan Nunez-Iglesias

Jun 07, 2017

Contents

Gala is a python library for performing and evaluating image segmentation, distributed under the open-source, BSD-like Janelia Farm license. It implements the algorithm described in Nunez-Iglesias et al., PLOS ONE, 2013.

If you use this library in your research, please cite:

> Nunez-Iglesias J, Kennedy R, Plaza SM, Chakraborty A and Katz WT (2014) Graph-based active learning of agglomeration (GALA): a Python library to segment 2D and 3D neuroimages. *Front. Neuroinform. 8:34.* doi:10.3389/fninf.2014.00034

If you use or compare to the GALA algorithm in your research, please cite:

> Nunez-Iglesias J, Kennedy R, Parag T, Shi J, Chklovskii DB (2013) Machine Learning of Hierarchical Clustering to Segment 2D and 3D Images. *PLoS ONE 8(8): e71715.* doi:10.1371/journal.pone.0071715

Gala supports n-dimensional images (images, volumes, videos, videos of volumes...) and multiple channels per image.

Contents:

CHAPTER 1

Installation

Requirements

After version 0.3, Gala requires Python 3.5 to run. For a full list of dependencies, see the *requirements.txt* file.

Optional dependencies

- vigra/vigranumpy (1.9.0)

In its original incarnation, this project used Vigra for the random forest classifier. Installation is less simple than scikit-learn, which has emerged in the last few years as a truly excellent implementation and is now recommended. Tests in the test suite expect scikit-learn rather than Vigra. You can also use any of the scikit-learn classifiers, including their newly-excellent random forest.

Installing with distutils

Gala is a Python library with limited Cython extensions and can be installed in two ways:

- Use the command `python setup.py build_ext -i` in the gala directory, then add the gala directory to your PYTHONPATH environment variable, or
- Install it into your preferred python environment with `python setup.py install`.

Installing requirements

Though you can install all the requirements yourself, as most are available in the Python Package Index (PyPI) and can be installed with simple commands, the easiest way to get up and running is to use [miniconda](http://conda.pydata.org/miniconda.html). Once you have the *conda* command, you can create a fully-functional gala environment with *conda env create -f environment.yml* (inside the gala directory).

Installing with Buildem

Alternatively, you can use Janelia's own buildem system to automatically download, compile, test, and install requirements into a specified buildem prefix directory. (You will need CMake.)

```
$ cmake -D BUILDEM_DIR=/path/to/platform-specific/build/dir <gala directory>
$ make
```

You might have to run the above steps twice if this is the first time you are using the buildem system.

On Mac, you might have to install compilers (such as gcc, g++, and gfortran).

Testing

The test coverage is rather tiny, but it is still a nice way to check you haven't completely screwed up your installation. The tests do cover the fundamental functionality of agglomeration learning.

We use pytest for testing. Run the tests by building gala in-place and running the `py.test` command. (You need to have installed pytest and pytest-cov for this to work. Both are readily available in PyPI.)

Alternatively, you can run individual test files independently:

```
$ cd tests
$ python test_agglo.py
$ python test_features.py
$ python test_watershed.py
$ python test_optimized.py
$ python test_gala.py
```

CHAPTER 2

Getting Started

An example script, `example.py`, exists in the `tests/example-data` directory. We step through it here for a quick rundown of gala's capabilities.

First, import gala's submodules:

```
from gala import imio, classify, features, agglo, evaluate as ev
```

Next, read in the training data: a ground truth volume (`gt_train`), a probability map (`pr_train`) and a superpixel or watershed map (`ws_train`).

```
gt_train, pr_train, ws_train = (map(imio.read_h5_stack,
                                    ['train-gt.lzf.h5', 'train-p1.lzf.h5',
                                     'train-ws.lzf.h5']))
```

A *feature manager* is a callable object that computes feature vectors from graph edges. The object has the following responsibilities, which it can inherit from `classify.base.Null`:

- create a (possibly empty) *feature cache* on each edge and node, precomputing some of the calculations needed for feature computation;
- maintain the feature cache throughout node merges during agglomeration; and,
- compute the feature vector from the feature caches when called with the inputs of a graph and two nodes.

Feature managers can be chained through the `features.Composite` class.

```
fm = features.moments.Manager()
fh = features.histogram.Manager()
fc = features.base.Composite(children=[fm, fh])
```

With the feature manager, and the above data, we can create a *region adjacency graph* or *RAG*, and use it to train the agglomeration process:

```
g_train = agglo.Rag(ws_train, pr_train, feature_manager=fc)
(X, y, w, merges) = g_train.learn_agglomerate(gt_train, fc)[0]
y = y[:, 0] # gala has 3 truth labeling schemes, pick the first one
```

`X` and `y` above have the now-standard scikit-learn supervised dataset format. This means we can use any classifier that satisfies the scikit-learn API. Below, we use a simple wrapper around the scikit-learn `RandomForestClassifier`.

```
rf = classify.DefaultRandomForest().fit(X, y)
```

The composition of a feature map and a classifier defines a *policy* or *merge priority function*, which will determine the agglomeration of a volume of hereby unseen data (the *test* volume).

```
learned_policy = agglo.classifier_probability(fc, rf)

pr_test, ws_test = (map(imio.read_h5_stack,
                        ['test-p1.lzf.h5', 'test-ws.lzf.h5']))
g_test = agglo.Rag(ws_test, pr_test, learned_policy, feature_manager=fc)
```

The best expected segmentation is obtained at a threshold of 0.5, when a merge has even odds of being correct or incorrect, according to the trained classifier.

```
g_test.agglomerate(0.5)
```

The RAG is a *model* for the segmentation. To extract the segmentation itself, use the `get_segmentation` function. This is a map of labels of the same shape as the original image.

```
seg_test1 = g_test.get_segmentation()
```

Gala transparently supports multi-channel probability maps. In the case of EM images, for example, one channel may be the probability that a given pixel is part of a cell boundary, while the next channel may be the probability that it is part of a mitochondrion. The feature managers work identically with single and multi-channel features.

```
# p4_train and p4_test have 4 channels
p4_train = imio.read_h5_stack('train-p4.lzf.h5')
# the existing feature manager works transparently with multiple channels!
g_train4 = agglo.Rag(ws_train, p4_train, feature_manager=fc)
(X4, y4, w4, merges4) = g_train4.learn_agglomerate(gt_train, fc)[0]
y4 = y4[:, 0]
rf4 = classify.DefaultRandomForest().fit(X4, y4)
learned_policy4 = agglo.classifier_probability(fc, rf4)
p4_test = imio.read_h5_stack('test-p4.lzf.h5')
g_test4 = agglo.Rag(ws_test, p4_test, learned_policy4, feature_manager=fc)
g_test4.agglomerate(0.5)
seg_test4 = g_test4.get_segmentation()
```

For comparison, gala allows the implementation of many agglomerative algorithms, including mean agglomeration (below) and LASH.

```
g_testm = agglo.Rag(ws_test, pr_test,
                    merge_priority_function=agglo.boundary_mean)
g_testm.agglomerate(0.5)
seg_testm = g_testm.get_segmentation()
```

Evaluation

The gala library contains numerous evaluation functions, including edit distance, Rand index and adjusted Rand index, and our personal favorite, the variation of information (VI):

```
gt_test = imio.read_h5_stack('test-gt.lzf.h5')
import numpy as np
results = np.vstack((
    ev.split_vi(ws_test, gt_test),
    ev.split_vi(seg_testm, gt_test),
    ev.split_vi(seg_test1, gt_test),
    ev.split_vi(seg_test4, gt_test)
    ))
print(results)
```

This should print something like:

```
[[ 0.1845286   1.64774412]
 [ 0.18719817  1.16091003]
 [ 0.38978567  0.28277887]
 [ 0.39504714  0.2341758 ]]
```

Each row is an evaluation, with the first number representing the undersegmentation error or false merges, and the second representing the oversegmentation error or false splits, both measured in bits.

(Results may vary since there is some randomness involved in training a random forest, and the datasets are small.)

As mentioned earlier, many other evaluation functions are available. See the documentation for the `evaluate` package for more information.

```
# rand index and adjusted rand index
ri = ev.rand_index(seg_test1, gt_test)
ari = ev.adj_rand_index(seg_test1, gt_test)
# Fowlkes-Mallows index
fm = ev.fm_index(seg_test1, gt_test)
```

Other options

Gala supports a wide array of merge priority functions to explore your data. We can specify the median boundary probability with the `merge_priority_function` argument to the RAG constructor:

```
g_testM = agglo.Rag(ws_test, pr_test,
                    merge_priority_function=agglo.boundary_median)
```

A user can specify their own merge priority function. A valid merge priority function is a callable Python object that takes as input a graph and two nodes, and returns a real number.

To be continued...

That's a quick summary of the capabilities of Gala. There are of course many options under the hood, many of which are undocumented... Feel free to push me to update the documentation of your favorite function!

CHAPTER 3

API Reference

gala.agglo: RAG Agglomeration

gala.classify: Classifier tools

gala.classify.**concatenate_data_elements**(*alldata*)
Return one big learning set from a list of learning sets.

A learning set is a list/tuple of length 4 containing features, labels, weights, and node merge history.

gala.classify.**default_classifier_extension**(*cl*, *use_joblib=True*)
Return the default classifier file extension for the given classifier.

Parameters **cl** : sklearn estimator or VigraRandomForest object

A classifier to be saved.

use_joblib : bool, optional

Whether or not joblib will be used to save the classifier.

Returns **ext** : string

File extension

Examples

```
>>> cl = RandomForestClassifier()
>>> default_classifier_extension(cl)
'.classifier.joblib'
>>> default_classifier_extension(cl, False)
'.classifier'
```

gala.classify.**get_classifier**(*name='random forest'*, **args*, ***kwargs*)
Return a classifier given a name.

Parameters **name** : string

The name of the classifier, e.g. 'random forest' or 'naive bayes'.

***args, **kwargs :**

Additional arguments to pass to the constructor of the classifier.

Returns **cl** : classifier

A classifier object implementing the scikit-learn interface.

Raises **NotImplementedError**

If the classifier name is not recognized.

Examples

```
>>> cl = get_classifier('random forest', n_estimators=47)
>>> isinstance(cl, RandomForestClassifier)
True
>>> cl.n_estimators
47
>>> from numpy.testing import assert_raises
>>> assert_raises(NotImplementedError, get_classifier, 'perfect class')
```

gala.classify.**load_classifier**(*fn*)

Load a classifier previously saved to disk, given a filename.

Supported classifier types are: - scikit-learn classifiers saved using either pickle or joblib persistence - vigra random forest classifiers saved in HDF5 format

Parameters **fn** : string

Filename in which the classifier is stored.

Returns **cl** : classifier object

cl is one of the supported classifier types; these support at least the standard scikit-learn interface of *fit()* and *predict_proba()*

gala.classify.**sample_training_data**(*features*, *labels*, *num_samples=None*)

Get a random sample from a classification training dataset.

Parameters **features: np.ndarray [M x N]**

The M (number of samples) by N (number of features) feature matrix.

labels: np.ndarray [M] or [M x 1]

The training label for each feature vector.

num_samples: int, optional

The size of the training sample to draw. Return full dataset if *None* or if num_samples >= M.

Returns feat: np.ndarray [num_samples x N]

The sampled feature vectors.

lab: np.ndarray [num_samples] or [num_samples x 1]

The sampled training labels

gala.classify.**save_classifier**(*cl*, *fn*, *use_joblib=True*, ***kwargs*)

Save a classifier to disk.

Parameters **cl** : classifier object

Pickleable object or a classify.VigraRandomForest object.

fn : string

Writeable path/filename.

use_joblib : bool, optional

Whether to prefer joblib persistence to pickle.

kwargs : keyword arguments

Keyword arguments to be passed on to either *pck.dump* or *joblib.dump*.

Returns None

Notes

For joblib persistence, *compress=3* is the default.

gala.features: Feature definitions

gala.morpho: Morphological operations

gala.evaluate: Segmentation evaluation

gala.imio: Image IO

gala.viz: Visualization tools

CHAPTER 4

Release notes

0.3

0.3.2

- Bug fix: missing import in `test_gala.py`. This was caused by rebasing commits from post-0.3 onto 0.3.

0.3.1

This is a major bug fix release addressing issue #63 on GitHub. You can read more there and in the related mailing list thread, but the gist is that the "learning mode" parameter did nothing in previous releases of gala. The gala library in fact was not implementing the algorithm described in the GALA paper, but rather, a variant of LASH with memory across epochs. (LASH only retains data from the most recent learning epoch.) It remains to be determined whether the "strict" learning mode described in our paper indeed yields improvements in segmentation accuracy.

Note that the included tests pass when using scikit-learn 0.16, but not with the recently-released 0.17, because of changes in the implementation of `GaussianNB`.

0.3.0

Announcing the third release of gala!

I want to thank Paul Watkins, Sean Colby, Larissa Heinrich, Joergen Kornfeld, and Jan Funke for their bug reports and mailing list discussions, which prompted almost all of the improvements in this release.

I must also thank the Saalfeld lab for financial support while I was making these improvements.

This release focuses on performance improvements, but also includes some API and behavior changes.

This is the last release of gala supporting Python 2. Upcoming work will focus on asynchronous learning to enable interactive proofreading, for which Python 3.4 and 3.5 offer compelling features and libraries. If you absolutely *need* Python 2.7 support in gala, get in touch!

On to the changes in this version!

Major changes:

- 2x memory reduction and 3x RAG construction speedup.
- Add support for masked volumes: use a boolean array of the same shape as the image to inspect only `True` positions.
- **API break:** The label "0" is no longer considered a boundary label; volumes with a single-voxel-thick boundary are no longer supported.
- **API break:** The Ultrametric Contour Map (UCM) is gone, because it is inaccurate without a voxel-thick boundary, and was computationally expensive to maintain.

Minor changes:

- Add `paper_em` and `snemi3d` default feature managers (in `gala.features.default`) to reproduce previous gala results.
- Bug fix: passing a label array of type floating point no longer causes a crash. (But you really should use integers for labels!)

0.2

0.2.3

Minor feature addition: enable exporting segmentation results *after* agglomeration is complete.

0.2.2

This maintenance release contains several bug fixes:

- package Cython source files (.pyx) for PyPI
- package the gala-segment command-line interface for PyPI
- include viridis in `requirements.txt`
- update libtiff usage

0.2

This release owes much of its existence to Neal Donnelly (@NealJMD on GitHub), who bravely delved into gala and reduced its memory and time footprints by over 20% each. The other highlights are Python 3 support and much better continuous integration.

Major changes:

- gala now uses an ultrametric tree backend to represent the merge hierarchy. This speeds up merges and will allow more sophisticated editing operations in the future.
- gala is now **fully compatible with Python 3.4**! That's a big tick in the "being a good citizen of the Python community" box. =) The downside is that a lot of the operations are slower in Py3.
- As mentioned above, gala is 20% faster and 20% smaller than before. That's thanks to extensive benchmarking and Cythonizing by @NealJMD
- We are now measuring code coverage, and although it's a bit low at 40%, the major gala functions (RAG building, learning, agglomerating) are covered. And we're only going up from here!
- We now have documentation on ReadTheDocs!

Minor changes:

- @anirbanchakraborty added the concepts of "frozen nodes" and "frozen edges", which are never merged. This is useful to temporarily ignore mitochondria during the first stages of agglomeration, which can dramatically reduce errors. (See A Context-aware Delayed Agglomeration Framework for EM Segmentation.)
- @anirbanchakraborty added the inclusiveness feature, a measure of how much a region is "surrounded" by another.
- The *gala.evaluate* module now supports the Adapted Rand Error, as used by the SNEMI3D challenge.
- Improvements to the *gala.morphology* module.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index