distributions Documentation

Release 2.0.0

Salesforce.com

November 18, 2014

Contents

CHAPTER 1

Overview

Distributions implements low-level primitives for Bayesian MCMC inference in Python and C++ including:

- special numerical functions `distributions.<flavor>.special`,
- samplers and density functions from a variety of distributions, `distributions.<flavor>.random`,
- conjugate component models (e.g., gamma-Poisson, normal-inverse-chi-squared) `distributions.<flavor>.models`, and
- clustering models (e.g., CRP, Pitman-Yor) `distributions.<flavor>.clustering`.

Python implementations are provided in up to three flavors:

- Debug `distributions.dbg` are pure-python implementations for correctness auditing and error checking, and allowing debugging via pdb.
- High-Precision `distributions.hp` are cython implementations for fast inference in python and numerical reference.
- Low-Precision `distributions.lp` are inefficent wrappers of blazingly fast C++ implementations, intended mostly as wrappers to check that C++ implementations are correct.

Our typical workflow is to first prototype models in python, then prototype faster inference applications using cython models, and finally implement optimized scalable inference products in C++, while testing all implementations for correctness.

1.1 Feature Model API

Feature models are contained in modules in python and structs in C++. Below write `Model.thing` to denote `module.thing` in python and `Model::thing` in C++.

Most functions consume explicit entropy sources in C++ or `global_rng` implicitly in python

Below `json` denotes a python dict/list/number/string suitable for serialization with the `json` package.

Each feature model API consist of:

- Datatypes.
  - `Shared` - shared global model state including fixed parameters, hyperparameters, and, for datatypes with dynamic support, shared sufficient statistics.
  - `Value` - observation state, i.e., datum
  - `Group` - local component state including sufficient statistics and possibly group parameters

- `Sampler` - partially evaluated per-group sampling function (optional in python)
- `Scorer` - cached per-group scoring function (optional in python)
- `Mixture` - vectorized scoring functions for mixture models (optional in python)

- Shared operations. These should be simple and fast:

```
shared = Model.Shared()
shared.protobuf_load(message)
shared.protobuf_dump(message)
shared.load(json)                              # python only
shared.dump() -> json                          # python only

Shared.from_dict(json) -> shared               # python only
Shared.from_protobuf(json, message)            # python only
Shared.to_protobuf(message) -> json            # python only

shared.add_value(value)
shared.add_repeated_value(value)
shared.remove_value(value)
shared.realize()
shared.plus_group(group) -> shared             # optional
```

- Group operations. These should be simple and fast. These may consume entropy:

```
group = Model.Group()
group.protobuf_load(message)
group.protobuf_dump(message)
group.load(json)                               # python only
group.dump() -> json                           # python only

Group.from_values(shared, values) -> group     # python only
Group.from_dict(json) -> group                 # python only
Group.from_protobuf(json, message)             # python only
Group.to_protobuf(message) -> json             # python only

group.init(shared)
group.add_value(shared, value)
group.add_repeated_value(shared, value, count)
group.remove_value(shared, value)
group.merge(shared, other_group)
group.sample_value(shared)
group.score_value(shared)
group.vaidate()                                # C++ only
```

- Sampling. These may consume entropy:

```
sampler = Model.Sampler()
sampler.init(shared, group)
sampler.eval(sampler) -> value
group.sample_value(shared) -> value
Model.sample_group(shared, group_size) -> group   # python only
```

- Scoring. These may also consume entropy, e.g. when implemented using monte carlo integration):

```
scorer = Model.Scorer()
scorer.init(shared, group)
scorer.eval(shared, value) -> float
group.score_value(shared, value) -> float
```

- Mixture Slaves (optional in python). These provide batch operations on a collection of groups.:

```
mixture = Model.Mixture()
mixture.groups().push_back(group)              # C++ only
mixture.append(group)                          # python only
mixture.init(shared)
mixture.add_group(shared)
mixture.remove_group(shared, groupid)
mixture.add_value(shared, groupid, value)
mixture.remove_value(shared, groupid, value)
mixture.score_value(shared, value, scores_accum)
mixture.score_data(shared) -> float
mixture.score_data_grid(shareds, scores_out)   # C++ only
```

- Testing metadata. Example model parameters and datasets are automatically discovered by unit test infrastructures, reducing the cost of per-model test-writing:

```
# in python
for example in Model.EXAMPLES:
    shared = Model.shared_load(example['shared'])
    values = example['values']
    ...

// in C++
Model::Shared shared = Model::Shared::EXAMPLE();
...
```

1.2 Clustering Model API

- Sampling and scoring:

```
model = Model()
model.sample_assignments(sample_size)
model.score_counts(counts)
model.score_add_value(...)
model.score_remove_value(...)
```

- Mixture driver (optional in python). These provide batch operations on a collection of groups. Clustering mixture drivers, referencing a `clustering` model:

```
mixture = model.Mixture()
mixture.counts().push_back(count)                    # C++ only
mixture.init(model)                                  # C++ only
mixture.init(model, counts)                          # python only
mixture.remove_group(shared, groupid)
mixture.add_value(shared, groupid, value) -> bool
mixture.remove_value(shared, groupid, value) -> bool
mixture.score_value(shared, value, scores_out)
mixture.score_data(shared) -> float
```

Mixture drivers and slaves coordinate using the pattern:

```
# driver is a single clustering model
# slaves is a list of feature models

def add_value(driver, slaves, groupid, value):
    added = driver.mixture.add_value(driver.shared, groupid, value)
```

```
        for slave in slaves:
            slave.mixture.add_value(slave.shared, groupid, value)
            if added:
                slave.mixture.add_group(slave.shared)

    def remove_value(driver, slaves, groupid, value):
        removed = driver.mixture.remove_value(driver.shared, groupid, value)
        for slave in slaves:
            slave.mixture.add_value(slave.shared, groupid, value)
            if removed:
                slave.mixture.remove_group(slave.shared, groupid)
```

See `examples/mixture/main.py` for a working example.

- Testing metadata (python only). Example model parameters and datasets are automatically discovered by unit test infrastructures, reducing the cost of per-model test-writing:

```
ExampleModel.EXAMPLES = [ ...model specific... ]
```

1.3 Source of Entropy

The C++ methods explicity require a random number generator `rng` everywhere entropy may be consumed. The python models try to maintain compatibility with `numpy.random` by hiding this source either as the global `numpy.random` generator, or as single `global_rng` in wrapped C++.

CHAPTER 2

Installation

You may build distributions in several ways:

- as a standalone C++ library
- as a standalone Python package
- as a Python package wrapping the dynamically-linked C++ library

Note: On OSX, distributions builds with newer versions of clang, but some systems default to g++. You can force distributions to use clang by setting the `CC` environment variable before running any `pip`, `cmake`, or `make` commands with `export CC=clang`.

2.1 Python Standalone

Install numpy and scipy. Then:

```
pip install distributions
```

2.2 C++ Standalone

Install requirements:

```
sudo apt-get install cmake libeigen3-dev
```

To install in `./lib`:

```
make install
```

Alternatively, set a custom install location:

```
CMAKE_INSTALL_PREFIX=/my/prefix make install
```

2.3 Python wrapping libdistributions

Follow instructions for C++ Standalone. Install numpy and scipy. Then:

```
LIBRARY_PATH=/my/prefix/lib pip install distributions
```

Warning: When using wrapped libdistributions, the dynamic linker must be able to find the library. The environment variables used to do this differ from platform to platform.

On Linux, you might run python as follows:

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/my/prefix/lib python
```

On OSX, you'll need a different flag:

```
DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/my/prefix/lib python
```

If you use virtualenv with virtualenvwrapper and use the virtualenv root as your prefix, it is convenient to add a postactivate hook to set this environment. On Linux, this would look like this:

```
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$VIRTUAL_ENV/lib' >> $VIRTUAL_ENV/bin/postactivate
```

2.4 Developer Quick Start

This will install both the static and dynamic versions of libdistributions within a virtualenv, then install the distributions Python package built to wrap libdistributions.

Install cmake. Install numpy, scipy, cython, and nosetests so that they're available within a python virtualenv. Activate that virtualenv. Then:

```
make test
```

The top-level `Makefile` provides many targets useful for development.