pescador Documentation

Release

Brian McFee and Eric Humphrey

February 23, 2016

Contents

CHAPTER 1

Simple example

This document will walk through the basics of training models using pescador.

Our running example will be learning from an infinite stream of stochastically perturbed samples from the Iris dataset.

Before we can get started, we'll need to introduce a few core concepts. We will assume some basic familiarity with scikit-learn and generators.

1.1 Batch generators

Not all python generators are valid for machine learning. Pescador assumes that generators produce output in a particular format, which we will refer to as a *batch*. Specifically, a batch is a python dictionary containing *np.ndarray*. For unsupervised learning (e.g., MiniBatchKMeans), valid batches contain only one key: *X*. For supervised learning (e.g., SGDClassifier), valid batches must contain both *X* and *Y* keys, both of equal length.

Here's a simple example generator that draws random batches of data from Iris of a specified *batch_size*, and adds gaussian noise to the features.

```
import numpy as np

def noisy_samples(X, Y, batch_size=16, sigma=1.0):
    '''Generate an infinite stream of noisy samples from a labeled dataset.

    Parameters
    ----------
    X : np.ndarray, shape=(n, d)
        Features

    Y : np.ndarray, shape=(n,)
        Labels

    batch_size : int > 0
        Size of the batches to generate

    sigma : float > 0
        Variance of the additive noise

    Yields
    ------
    batch
    '''

```

```
    n, d = X.shape

    while True:
        i = np.random.randint(0, n, size=m)

        noise = sigma * np.random.randn(batch_size, d)

        yield dict(X=X[i] + noise, Y=Y[i])
```

In the code above, *data_stream* is an iterator that can be sampled indefinitely because *noisy_samples* contains an infinite loop. Each iterate of *data_stream* will be a dictionary containing the sample batch's features and labels.

1.2 StreamLearner

Many scikit-learn classes provide an iterative learning interface via *partial_fit()*, which can update an existing model after observing a new batch of samples. Pescador provides an additional layer (*StreamLearner*) which interfaces between batch generators and *partial_fit()*.

The following example illustrates how to use *StreamLearner*.

```
from __future__ import print_function

import sklearn.datasets
from sklearn.cross_validation import ShuffleSplit
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score

import pescador

# Load the Iris dataset
data = sklearn.datasets.load_iris()
X, Y = data.data, data.target

# Get the space of class labels
classes = np.unique(Y)

# Generate a single 90/10 train/test split
for train, test in ShuffleSplit(len(X), n_iter=1, test_size=0.1)

    # Instantiate a linear classifier
    estimator = SGDClassifier()

    # Wrap the estimator object in a stream learner
    model = pescador.StreamLearner(estimator, max_batches=1000)

    # Build a data stream
    batch_stream = noisy_samples(X[train], Y[train])

    # Fit the model to the stream
    model.iter_fit(batch_stream, classes=classes)

    # And report the accuracy
    print('Test accuracy: {:.3f}'.format(accuracy_score(Y[test],
                                                        model.predict(X[test]))))
```

A few things to note here:

- Because *noisy_samples* is an infinite generator, we need to provide an explicit bound on the amount of samples to draw when fitting. This is done in line 20 with the *max_batches* parameter to *StreamLearner*.
- *StreamLearner* objects transparently wrap the methods of their contained *estimator* object, so *model.predict(X[test])* and *model.estimator.predict(X[test])* are equivalent.

CHAPTER 2

Advanced example

This document will walk through advanced usage of pescador.

We will assume a working understanding of the simple example in the previous section.

2.1 Streamers

Generators in python have a couple of limitations for common stream learning pipelines. First, once instantiated, a generator cannot be "restarted". Second, an instantiated generator cannot be serialized directly, so they are difficult to use in distributed computation environments.

Pescador provides the *Streamer* object to circumvent these issues. *Streamer* simply provides an object container for an uninstantiated generator (and its parameters), and an access method *generate()*. Calling *generate()* multiple times on a streamer object is equivalent to restarting the generator, and can therefore be used to simply implement multiple pass streams. Similarly, because *Streamer* can be serialized, it is simple to pass a streamer object to a separate process for parallel computation.

Here's a simple example, using the generator from the previous section.

```
import pescador

streamer = pescador.Streamer(noisy_samples, X[train], Y[train])

batch_stream2 = streamer.generate()
```

Iterating over *streamer.generate()* is equivalent to iterating over *noisy_samples(X[train], Y[train])*.

Additionally, Streamer can be bounded easily by saying *streamer.generate(max_batches=N)* for some *N* maximum number of batches.

2.2 Stream re-use and multiplexing

The *mux()* function provides a powerful interface for randomly interleaving samples from multiple input streams. *mux* can also dynamically activate and deactivate individual *Streamers*, which allows it to operate on a bounded subset of streams at any given time.

As a concrete example, we can simulate a mixture of noisy streams with differing variances.

```
for train, test in ShuffleSplit(len(X), n_iter=1, test_size=0.1)

    # Instantiate a linear classifier
```

```
    estimator = SGDClassifier()

    # Wrap the estimator object in a stream learner
    model = pescador.StreamLearner(estimator, max_batches=1000)

    # Build a collection of streams with different variance scales
    streams = [noisy_samples(X[train], Y[train], sigma=sigma)
               for sigma in [0.5, 1.0, 2.0, 4.0]]

    # Build a mux stream, keeping only 2 streams alive at once
    batch_stream = pescador.mux(streams,
                                1000,   # Generate 1000 batches in total
                                2,      # Keep 2 streams alive at once
                                lam=16) # Use a poisson rate of 16


    # Fit the model to the stream
    model.iter_fit(batch_stream, classes=classes)

    # And report the accuracy
    print('Test accuracy: {:.3f}'.format(accuracy_score(Y[test],
                                                        model.predict(X[test]))))
```

In the above example, each *noisy_samples* streamer is infinite. The *lam=16* argument to *mux* says that each stream should produce some *n* batches, where *n* is sampled from a Poisson distribution of rate *lam*. When a stream exceeds its bound, it is deactivated, and a new stream is activated to fill its place.

Setting *lam=None* disables the random stream bounding, and *mux()* simply runs each active stream until exhaustion.

Streams can be sampled with or without replacement according to the *with_replacement* option. Setting this parameter to *False* means that each stream can be active at most once.

Streams can also be sampled with non-uniform weighting by specifying a vector *pool_weights*.

Finally, exhausted streams can be removed by setting *prune_empty_seeds* to *True*. If *False*, then exhausted streams may be reactivated at any time.

Note that because *mux()* itself is a generator, it too can be wrapped in a *Streamer* object.

CHAPTER 3

API Reference

3.1 The Streamer object

3.2 The StreamLearner object

3.3 Stream manipulation

Utility functions for stream manipulations

mux(seed_pool, n_samples, k[, lam, ...])	Stochastic multiplexor for generator seeds.
buffer_batch(generator, buffer_size)	Buffer an iterable of batches into larger (or smaller) batches
buffer_streamer(streamer, buffer_size, ...)	Buffer a stream of batches
batch_length(batch)	Determine the number of samples in a batch.

3.3.1 util.mux

util.**mux**(*seed_pool*, *n_samples*, *k*, *lam=256.0*, *pool_weights=None*, *with_replacement=True*, *prune_empty_seeds=True*, *revive=False*)

Stochastic multiplexor for generator seeds.

Given an array of Streamer objects, do the following:

1. Select `k` seeds at random to activate
2. Assign each activated seed a sample count ~ Poisson(lam)
3. Yield samples from the streams by randomly multiplexing from the active set.
4. When a stream is exhausted, select a new one from the pool.

Parameters **seed_pool** : iterable of Streamer

The collection of Streamer objects

n_samples : int > 0 or None

The number of samples to generate. If `None`, sample indefinitely.

k : int > 0

The number of streams to keep active at any time.

lam : float > 0 or None

Rate parameter for the Poisson distribution governing sample counts for individual streams. If `None`, sample infinitely from each stream.

pool_weights : np.ndarray or None

Optional weighting for `seed_pool`. If `None`, then weights are assumed to be uniform. Otherwise, `pool_weights[i]` defines the sampling proportion of `seed_pool[i]`.

Must have the same length as `seed_pool`.

with_replacement : bool

Sample Streamers with replacement. This allows a single stream to be used multiple times (even simultaneously). If `False`, then each Streamer is consumed at most once and never revisited.

prune_empty_seeds : bool

Disable seeds from the pool that produced no data. If `True`, Streamers that previously produced no data are never revisited. Note that this may be undesireable for streams where past emptiness may not imply future emptiness.

revive: bool

If `with_replacement` is `False`, setting `revive=True` will re-insert previously exhausted seeds into the candidate set.

This configuration allows a seed to be active at most once at any time.

3.3.2 util.buffer_batch

`util.`**`buffer_batch`**(*generator*, *buffer_size*)

Buffer an iterable of batches into larger (or smaller) batches

Parameters **generator** : iterable

The generator to buffer

buffer_size : int > 0

The number of examples to retain per batch.

Yields batch

A batch of size at most *buffer_size*

3.3.3 util.buffer_streamer

`util.`**`buffer_streamer`**(*streamer*, *buffer_size*, **args*, ***kwargs*)

Buffer a stream of batches

Parameters **streamer** : pescador.Streamer

The streamer object to buffer

buffer_size : int > 0

the number of examples to retain per batch

Yields batch

A batch of size at most *buffer_size*

See also:

buffer_batch

3.3.4 util.batch_length

util.**batch_length**(*batch*)

Determine the number of samples in a batch.

Parameters **batch** : dict

A batch dictionary. Each value must implement *len*. All values must have the same *len*.

Returns **n** : int >= 0 or None

The number of samples in this batch. If the batch has no fields, n is None.

Raises **RuntimeError**

If some two values have unequal length

3.4 Parallelism

ZMQ-baesd stream multiplexing

zmq_stream(streamer[, max_batches, ...])	Parallel data streaming over zeromq sockets.

3.4.1 zmq_stream.zmq_stream

zmq_stream.**zmq_stream**(*streamer*, *max_batches=None*, *min_port=49152*, *max_port=65535*, *max_tries=100*, *copy=False*, *timeout=None*)

Parallel data streaming over zeromq sockets.

This allows a data generator to run in a separate process from the consumer.

A typical usage pattern is to construct a *Streamer* object from a generator (or *util.mux* of several *Streamer's), and then use 'zmq_stream* to execute the stream in one process while the other process consumes data, e.g., with a *StreamLearner* object.

Parameters **streamer** : *pescador.Streamer*

The streamer object

max_batches : None or int > 0

Maximum number of batches to generate

min_port : int > 0

max_port : int > min_port

The range of TCP ports to use

max_tries : int > 0

The maximum number of connection attempts to make

copy : bool

Set *True* to enable data copying

Yields batch

Data drawn from *streamer.generate(max_batches)*.

CHAPTER 4

Changes

4.1 Changes

4.1.1 v0.1.2

- Added `pescador.mux` parameter *revive*. Calling with *with_replacement=False, revive=True* will use each seed at most once at any given time.
- Added `pescador.zmq_stream` parameter *timeout*. Setting this to a positive number will terminate dangling worker threads after *timeout* is exceeded on join. See also: `multiprocessing.Process.join`.

4.1.2 v0.1.1

- `pescador.mux` now throws a `RuntimeError` exception if the seed pool is empty

4.1.3 v0.1.0

Initial public release

CHAPTER 5

Contribute

- Issue Tracker
- Source Code

5.1 Indices and tables

- genindex
- modindex
- search

p

u

z

B

M

P

U

Z