torch-two-sample Documentation

Release 0.1

Josip Djolonga

Sep 26, 2017

Contents

CHAPTER 1

Overview

This package implements a total of six two sample tests:

- The classical Friedman-Rafsky test *[FR79]*.
- The classical k-nearest neighbours (kNN) test *[FR83]*.
- The differentiable Friedman-Rafsky test *[DK17]*.
- The differentiable k-nearest neighbours (kNN) test *[DK17]*.
- The maximum mean discrepancy (MMD) test *[GBR+12]*.
- The energy test *[SzekelyR13]*.

These tests accept as input two samples and produce a statistic that should be large when the samples come from different distributions. The first two of these are *not* differentiable, and can be only used for statistical testing, but not for learning implicit generative models.

Incidentally, there is also code for marginal inference in models with cardinality potentials, or spanning-tree constraints, which are internally used for the implementation of the smooth graph tests. The code for inference in cardinality potentials has been adapted from the code accompanying *[SST+12]*, while the original algorithm comes from *[TSZ+12]*. Please consider citing these papers if you use the smoothed k-NN test, which relies on this inference method.

CHAPTER 2

Installation

Once you install PyTorch (following these instructions) , you can install the package as:

```
python setup.py install
```

CHAPTER 3

Usage

Statistical testing

First, one has to create the object that implements the test. For example:

```
>>> n = 128  # Sample size.
>>> fr_test = SmoothFRStatistic(n, n, cuda=False, compute_t_stat=True)
```

Note that we have used `compute_t_stat=True`, which means that `fr_test.__call__` will compute a t-statistic, so that we can obtain an approximate p-value from it using the CDF of a standard normal. We have implemented t-statistics only for the smooth graph tests.

Let us work out a concrete example. We will draw three different samples from 5 dimensional Gaussians, so that the first two come from the same distribution, while the last one from a distribution with a different mean.

```
>>> # We fix the seeds so that the result is reproducible.
>>> torch.manual_seed(0)
>>> numpy.random.seed(0)
>>> dim = 5  # The data dimension.
>>> sample_1 = torch.randn(n, dim)
>>> sample_2 = torch.randn(n, dim)
>>> sample_3 = torch.randn(n, dim)
>>> sample_3[:, 0] += 1  # Shift the first coordinate by one.
```

We can now compute the t-statistics (the returned matrices will be discussed shortly).

```
>>> t_val_12, matrix_12 = fr_test(Variable(sample_1), Variable(sample_2),
>>>                               alphas=[4.], ret_matrix=True)
>>> t_val_13, matrix_13 = fr_test(Variable(sample_1), Variable(sample_3),
>>>                               alphas=[4.], ret_matrix=True)
```

The approximate p-values can be then computed from the normal's CDF as follows:

```
>>> from scipy.stats import norm   # To evaluate the CDF of a normal.
>>> print('1 vs 2', 1 - norm.cdf(t_val_12.data[0]))
```

```
1 vs 2 0.834159503205
>>> print('1 vs 3', 1 - norm.cdf(t_val_13.data[0]))
1 vs 3 1.31049615604e-11
```

We also provide the means to compute the p-value by sampling from the permutation null. Namely, if you provide set `ret_matrix=True` in the invocation of `fr_test`, the function returns a second return value, which can be then used to compute the p-value. Every test has a `pval` method, which accepts this second return value and returns the p-value estimated from random permutations. We would like to point out that this method has been written in cython, so that it should execute reasonably fast. Concretely, we can compute the (sampled) exact p-value as follows.

```
>>> np.seed(0)  # The method is internally using numpy.
>>> print('1 vs 2', fr_test.pval(matrix_12, n_permutations=1000))
1 vs 2 0.853999972343
>>> print('1 vs 3', fr_test.pval(matrix_13, n_permutations=1000))
1 vs 3 0.0
```

Implicit model learning

We go through a simple example of learning an implicit generative model on MNIST. While here we present only the minimal code necessary to train a model, we provide in our repository a jupyter notebook that builds on the code below and should be directly executable on your machine. As the base measure we will use a 10-dimensional Gaussian, and the following generative model:

```
>>> generator = nn.Sequential(
>>>   nn.Linear(10, 128),  # Receive a 10 dimensional noise vector as input.
>>>   nn.ReLU(),  # Then, a single hidden layer of 10 units with ReLU.
>>>   nn.Linear(128, 28 * 28),  # The output has 28 * 28 dimensions.
>>>   nn.Sigmoid())  # Squash the output to [0, 1].
```

To optimize we will use the `torch.optim.Adam` optimizer *[KB15]*.

```
>>> optimizer = torch.optim.Adam(generator.parameters(), lr=1e-3)
```

As a loss function we will use the smoothed 1-NN loss with a batch size of 256.

```
>>> batch_size = 256
>>> loss_fn = SmoothKNNStatistic(
>>>    batch_size, batch_size, False, 1, compute_t_stat=True)
```

Next, let us load the MNIST dataset using `torchvision`.

```
>>> from torchvision.datasets import MNIST
>>> from torchvision.transforms import ToTensor
>>> dataset = MNIST('mnist_dir', transform=ToTensor(), download=True)
```

We can then train the model for 50 epochs.

```
>>> from torch.utils.data import DataLoader
>>>
>>> for epoch in range(1, 51):
>>>   data_loader = DataLoader(dataset, batch_size=batch_size, drop_last=True)
>>>   # Note that we drop the last batch as they all have to be of equal size.
>>>   noise_tensor = torch.FloatTensor(batch_size, 10)
>>>   losses = []
>>>   noise = Variable(noise_tensor)
>>>   for batch, _ in data_loader:
>>>     batch = batch.view(batch_size, -1)  # We want one observation per row.
>>>     noise_tensor.normal_()
>>>     optimizer.zero_grad()
>>>     loss = loss_fn(Variable(batch), generator(noise), alphas=[0.1])
>>>     loss.backward()
>>>     losses.append(loss.data[0])
>>>     optimizer.step()
>>>   print('epoch {0:>2d}, avg loss {1}'.format(epoch, np.mean(losses)))
```

CHAPTER 4

Statistics

Differentiable statistics

These tests can be used for both learning implicit models and statistical two sample testing.

Non-differentiable statistics

These tests can be only used for statistical two-sample testing.

CHAPTER 5

Marginal inference

Spanning-tree distributions

Cardinality potentials

Bibliography

Indices and tables

- genindex
- modindex
- search

Bibliography

[DK17] J. Djolonga and A. Krause. Learning Implicit Generative Models Using Differentiable Graph Tests. *ArXiv e-prints*, September 2017. arXiv:1709.01006.

[FR79] Jerome H Friedman and Lawrence C Rafsky. Multivariate generalizations of the wald-wolfowitz and smirnov two-sample tests. *Annals of Statistics*, pages 697–717, 1979.

[FR83] Jerome H Friedman and Lawrence C Rafsky. Graph-theoretic measures of multivariate association and prediction. *Annals of Statistics*, pages 377–391, 1983.

[GBR+12] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773, 2012.

[KB15] Diederik Kingma and Jimmy Ba. Adam: a method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*. 2015.

[SST+12] Kevin Swersky, Ilya Sutskever, Daniel Tarlow, Richard S Zemel, Ruslan R Salakhutdinov, and Ryan P Adams. Cardinality restricted boltzmann machines. In *Advances in Neural Information Processing Systems (NIPS)*, 3293–3301. 2012.

[SzekelyR13] Gábor J Székely and Maria L Rizzo. Energy statistics: a class of statistics based on distances. *Journal of Statistical Planning and Inference*, 143(8):1249–1272, 2013.

[TSZ+12] Daniel Tarlow, Kevin Swersky, Richard S Zemel, Ryan Prescott Adams, and Brendan J Frey. Fast exact inference for recursive cardinality models. *Uncertainty in Artificial Intelligence (UAI)*, 2012.