JMCtools Documentation

Release 1.0.0

Ben Farmer

Jul 31, 2018

Contents:

Python tools for perfoming Monte-Carlo studies of joint distribution functions that are built piecemeal from scipy.stats distribution objects (or any objects that 'quack' like them).

CHAPTER 1

Installation

This project is hosted on github. You can download and install it in the usual ways, for example:

```
git clone https://github.com/bjfar/JMCtools.git
pip install ./JMCtools
```

It is recommended to use pip to install the package since this is compatible with anaconda environments.

Once installed the package can be imported in python using the module name `JMCtools`.

CHAPTER 2

Quick start

The principle pipeline which JMCtools is designed to streamline is the following:

1. *Combine independent distribution functions into a joint distribution*
2. *Sample from the joint distribution*
3. *Build relationships between model parameters and distribution parameters*
4. *Find maximum likelihood estimators* for these parameters (for many samples/trials, in parallel)
5. *Construct test statistics*

Getting more advanced, we can further combine joint distributions into experiments, define test statistics to compute in analysis objects, and loop over the whole procedure to compute trial_corrections.

A fast introduction to the package, then, is to see an example of this in action. So let's get to it!

2.1 Combine independent distribution functions into a joint distribution

Suppose we have several independent random variables, which can each be modelled by an object from scipy.stats. JMCtools provides the `JointModel` class for the purpose of packaging these variables together into one single distribution-function-like object, which has similar (although not identical) behaviour and function to the native scipy.stats objects.

For example, we can create the joint PDF for two normal random variables as follows:

```
import JMCtools.distributions as jtd
import scipy.stats as sps
import numpy as np

joint = jtd.JointModel([sps.norm,sps.norm])
```

2.2 Sample from the joint distribution

Now that we have an object describing our joint PDF, we can sample from it in a scipy.stats manner:

```
null_parameters = [{'loc': 3, 'scale': 1},
                   {'loc': 1, 'scale': 2}]
samples = joint.rvs((10000,),null_parameters)
```

We can also evaluate the joint PDF, and compare it to our samples to check that they seem reasonable:

```
# Compute 2D PDF over grid
nxbins=100
nybins=100
x = np.linspace(-2,8,nxbins)
y = np.linspace(-6,10,nybins)
X, Y = np.meshgrid(x, y)
dxdy = (x[1]-x[0]) * (y[1]-y[0])
PDF = joint.pdf([X,Y],null_parameters)

# Construct smallest intervals containing certain amount of probability
outarray = np.ones((nxbins,nybins))
sb = np.argsort(PDF.flat)[::-1]
outarray.flat[sb] = np.cumsum(PDF.flat[sb] * dxdy)

# Make plot!
import matplotlib.pyplot as plt
fig= plt.figure(figsize=(5,4))
ax = fig.add_subplot(111)
ax.contourf(X, Y, outarray, alpha=0.3, levels=[0,0.68,0.95,0.997])
ax.scatter(*samples,lw=0,s=1)
ax.set_xlabel("x")
ax.set_ylabel("y")
fig.savefig("example_2D_joint.svg")
```

Fig. 1: Contours of the PDF of the joint distribution, with samples overlayed.

2.3 Build relationships between model parameters and distribution parameters

In JMCtools a *model* consists of two main components: a `JointModel`, and a list of functions which take some abstract parameters and return the arguments needed to evaluate the distribution functions managed by the `JointModel`. These are combined via the `ParameterModel` class.

For example, if we leave the variances of our two normal distributions fixed, and take the means as independent parameters, we can construct a simple two parameter model as follows:

```
import JMCtools.models as jtm

def pars1(a):
   return {'loc': a, 'scale':1}

def pars2(b):
```

(continues on next page)

(continued from previous page)

```
   return {'loc': b, 'scale':2}

model = jtm.ParameterModel(joint,[pars1,pars2])
```

For the purposes of efficiently finding maximum likelihood estimators for these parameters, the `ParameterModel` class automatically infers the block structure of the model. That is, it figures out which blocks of parameters are needed to evaluate which distribution functions. In our example the two parameters independently fix the means of each normal distribution, so our 2D model can be broken down into two independent 1D models. We can see that the `ParameterModel` object has noticed this by inspecting its `blocks` attribute:

```
print(model.blocks)
```

which produces:

```
>>> {(deps=['a'], submodels=[0]), (deps=['b'], submodels=[1])}
```

Here the output is telling us that one parameter block depends on the parameter a, and fixes the arguments of the 0th component of the `JointModel`, and a second parameter block depends on b and fixes the 1th joint distribution component.

As a quick aside, it is useful to see what happens if the model parameters correlate the arguments of the joint distribution components:

```
def pars3(a,b):
   return {'loc': a+b, 'scale':1}

model2 = jtm.ParameterModel(joint,[pars3,pars2])
print(model2.blocks)
```

which produces:

```
>>> {(deps=['a', 'b'], submodels=[0, 1])}
```

So now, there is only one parameter block, that depends on both parameters and fixes the arguments of both joint distribution components. The important difference is that now this block will require a 2D optimisation in order to locate the maximum likelihood estimators for a and b, whereas previously they could be found by two independent 1D optimisations (which is much faster).

2.4 Find maximum likelihood estimators

Now that we have a `ParameterModel`, we can use it to find maximum likelihood estimators for all the parameters that we have defined. This is made simple by the `find_MLE_parallel()` member function, which can find MLEs for each simulated dataset, splitting the task over multiple processes if desired.

We simulated some data earlier using the `JointModel` class, but we can also simulate it directly from the `ParameterModel`:

```
null_parameters = {'a':3, 'b':1}
Ntrials = 10000
Ndraws = 1
data = model.simulate((Ntrials,Ndraws),null_parameters)
```

Note that the shape of the simulated data is important. The length of the last dimension is interpreted as the number of draws from the joint distribution per *trial* or pseudoexperiment. In this way, one can easily find the MLEs given

multiple independent draws. But for simplicity we here do just one draw per experiment. (For more complicated scenarios where different components of the joint distribution require different numbers of "draws", one must manually construct the appropriate `JointModel` before wrapping it in a `ParameterModel`.)

Finding the MLEs requires setting some options for the chosen optimisation method and then simply calling `find_MLE_parallel()`

```
# Set starting values and step sizes for Minuit search
options = {'a':3, 'error_a':1, 'b': 1, 'error_b': 2}
Lmax, pmax = model.find_MLE_parallel(options,data,method='minuit',Nprocesses=3)
# Note that Lmax are the log-likelihoods of the MLEs,
# and pmax are the parameter values.
```

2.5 Construct test statistics

The final step is to construct some test statistic of interest! Here we will compute a likelihood ratio test statistic:

```
Lnull = model.logpdf(null_parameters)
LLR = -2*(Lnull - Lmax) # log-likelihood ratio

# Plot!
n, bins = np.histogram(LLR, bins=100, normed=True)
q = np.arange(0,9,0.01)
fig = plt.figure(figsize=(5,4))
ax = fig.add_subplot(111)
ax.plot(bins[:-1],n,drawstyle='steps-post',label="Minuit",c='r')
ax.plot(q,sps.chi2.pdf(q, 2),c='k',label="Asymptotic")
ax.set_xlabel("LLR")
ax.set_ylabel("pdf(LLR)")
ax.set_ylim(0.001,2)
ax.set_xlim(0,9)
ax.set_yscale("log")
ax.legend(loc=1, frameon=False, framealpha=0,prop={'size':14})

fig.savefig('quickstart_LLR.svg')
```

Fig. 2: Simulated distribution of likelihood ratio test statistic (red), and expected distribution according to asymptotic theory (black).

And that's it! Everything this package is good for is just an application of the above pipeline to different problems.

The unadulterated code for the above examples can be viewed *here*.

2.6 Limitations

Please note the following:

- The *grid* optimisation method is very fast for low-dimensional problems, and very slow for dimensions larger than about 2 due to the curse of dimensionality. Note also that results will be poor if the resolution of the grid is not sufficiently below the variance of the MLEs.

- The *minuit* optimisation method needs a little help from you in order to get reliable results. The starting guess needs to put the minimiser in the correct global minima, and the step size needs to be small enough that Minuit doesn't jump out of this minima. For more tips on using Minuit see the iminuit documentation.
- There are currently no global optimisers implemented. Thus, if finding MLEs for your parameters requires solving a difficult global optimisation problem then this package cannot help you, sorry!

CHAPTER 3

Examples from quick start

Return to *Quick start* guide.

```
# make_joint
import JMCtools.distributions as jtd
import scipy.stats as sps
import numpy as np

joint = jtd.JointModel([sps.norm,sps.norm])
# sample_pdf
null_parameters = [{'loc': 3, 'scale': 1},
                   {'loc': 1, 'scale': 2}]
samples = joint.rvs((10000,),null_parameters)
# check_pdf
# Compute 2D PDF over grid
nxbins=100
nybins=100
x = np.linspace(-2,8,nxbins)
y = np.linspace(-6,10,nybins)
X, Y = np.meshgrid(x, y)
dxdy = (x[1]-x[0]) * (y[1]-y[0])
PDF = joint.pdf([X,Y],null_parameters)

# Construct smallest intervals containing certain amount of probability
outarray = np.ones((nxbins,nybins))
sb = np.argsort(PDF.flat)[::-1]
outarray.flat[sb] = np.cumsum(PDF.flat[sb] * dxdy)

# Make plot!
import matplotlib.pyplot as plt
fig= plt.figure(figsize=(5,4))
ax = fig.add_subplot(111)
ax.contourf(X, Y, outarray, alpha=0.3, levels=[0,0.68,0.95,0.997])
ax.scatter(*samples,lw=0,s=1)
ax.set_xlabel("x")
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel("y")
fig.savefig("example_2D_joint.svg")
# build_model
import JMCtools.models as jtm

def pars1(a):
   return {'loc': a, 'scale':1}

def pars2(b):
   return {'loc': b, 'scale':2}

model = jtm.ParameterModel(joint,[pars1,pars2])
# block_structure
print(model.blocks)
# alt_model
def pars3(a,b):
   return {'loc': a+b, 'scale':1}

model2 = jtm.ParameterModel(joint,[pars3,pars2])
print(model2.blocks)
# sim_data
null_parameters = {'a':3, 'b':1}
Ntrials = 10000
Ndraws = 1
data = model.simulate((Ntrials,Ndraws),null_parameters)
# find_MLEs
# Set starting values and step sizes for Minuit search
options = {'a':3, 'error_a':1, 'b': 1, 'error_b': 2}
Lmax, pmax = model.find_MLE_parallel(options,data,method='minuit',Nprocesses=3)
# Note that Lmax are the log-likelihoods of the MLEs,
# and pmax are the parameter values.
# compute_stats
Lnull = model.logpdf(null_parameters)
LLR = -2*(Lnull - Lmax) # log-likelihood ratio

# Plot!
n, bins = np.histogram(LLR, bins=100, normed=True)
q = np.arange(0,9,0.01)
fig = plt.figure(figsize=(5,4))
ax = fig.add_subplot(111)
ax.plot(bins[:-1],n,drawstyle='steps-post',label="Minuit",c='r')
ax.plot(q,sps.chi2.pdf(q, 2),c='k',label="Asymptotic")
ax.set_xlabel("LLR")
ax.set_ylabel("pdf(LLR)")
ax.set_ylim(0.001,2)
ax.set_xlim(0,9)
ax.set_yscale("log")
ax.legend(loc=1, frameon=False, framealpha=0,prop={'size':14})

fig.savefig('quickstart_LLR.svg')
```

CHAPTER 4

JMCtools package

4.1 Submodules

4.2 JMCtools.common module

4.3 JMCtools.distributions module

4.4 JMCtools.models module

4.5 Module contents

These tools are fairly simple wrappers of scipy.stats objects, made for convenience rather than speed. Effort has been taken to make them pretty efficient, but if you use them to construct and analyse monstrously large joint PDFs then things will not be fast. For such intense usage you will need a more specialised (and more complicated!) toolkit, such as ROOT.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

J