Tutorial-Copy2

November 24, 2021

1 Tutorial for the Bayesian machine scientist

This tutorial illustrates how to program a Bayesian machine scientist, using the code provided here. The tutorial assumes general knowledge of Python programming. We start by importing all necessary Python modules:

[1]:
```
import sys
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

import matplotlib.pyplot as plt
from copy import deepcopy
from ipywidgets import IntProgress
from IPython.display import display,display_latex

sys.path.append('./')
sys.path.append('./Prior/')
from mcmc import *
from parallel import *
from fit_prior import read_prior_par
```

1.1 Loading and preparing the data

We then load the data. In this particular case, we load the salmon stocks data. The features (independent variables) are loaded into a Pandas `DataFrame` named `x`, whereas the target (dependent) variable is loaded into a Pandas `Series` named `y`. Data should **always** be loaded in these formats to avoid problems.

[2]:
```
XLABS = [
    'x1',
    'x2'
]
def f1(x):
    return 2.5+x+np.sin(x)
def f2(x1,x2):
    return x1*(1+x2)*np.cos(x2)
```

```
x1=np.arange(-2.0,2.0,0.5)
x2=np.arange(-2.0,2.0,0.5)
x=pd.DataFrame(columns={'x1','x2'})
for a in x1:
    for b in x2:
        x=x.append({'x1':a,'x2':b},ignore_index=True)

y = pd.Series(f2(x['x1'],x['x2']))#+np.random.normal(0,0.1,size=len(x.x1)))
x,y
```

[2]:
```
(     x1   x2
 0  -2.0 -2.0
 1  -2.0 -1.5
 2  -2.0 -1.0
 3  -2.0 -0.5
 4  -2.0  0.0
 ..  …  …
 59  1.5 -0.5
 60  1.5  0.0
 61  1.5  0.5
 62  1.5  1.0
 63  1.5  1.5

 [64 rows x 2 columns],
 0    -0.832294
 1     0.070737
 2    -0.000000
 3    -0.877583
 4    -2.000000
         …
 59    0.658187
 60    1.500000
 61    1.974561
 62    1.620907
 63    0.265265
 Length: 64, dtype: float64)
```

1.2 Initializing the Bayesian machine scienstist

We start by initializing the machine scientist. This involves three steps: - **Reading the prior hyperparameters.** The values of the hyperparameters depend on the number of variables `nv` and parameters `np`considered during the search. Many combinations of `nv` and `np` have hyperparameters calculated in the `Prior` directory. Otherwise, the hyperparameters should be fit. - **Setting the "temperatures" for the parallel tempering.** If you don't know what parallel tempering is, you can read it in the Methods section of the paper, or just leave it as is in the code. In general, more temperatures (here 20) lead to better sampling of the expression space (we use a maximum of 100 different temperatures) - **Initializing the (parallel) scientist.**

[3]:
```
%%time
from machinescientist import machinescientist
# Read the hyperparameters for the prior
prior_par = read_prior_par('./Prior/final_prior_param_sq.named_equations.nv2.
 ↪np2.2016-09-09 18:49:43.038278.dat')


best_description_lengths,lowest_mdl,best_model =␣
 ↪machinescientist(x=x,y=y,XLABS=XLABS,n_params=2,prior_par=prior_par,resets=2,
                   steps_prod=7000
                   )
```

```
Run 1
-23693.6917510055 ((((((((((x1 * (_a0_ * _a1_)) / x1) + _a1_) ** 2) * ((x2 /
_a1_) * x1)) + (x1 * _a1_)) * cos(x2)) * (_a1_ * _a1_)) / (x2 / x2)) + (_a0_ *
_a0_))
Run 2
-23831.1721859678 (((_a0_ + ((x1 / _a1_) + x2)) * cos(x2)) * x1)
CPU times: user 35min 12s, sys: 24.2 s, total: 35min 36s
Wall time: 34min 37s
```

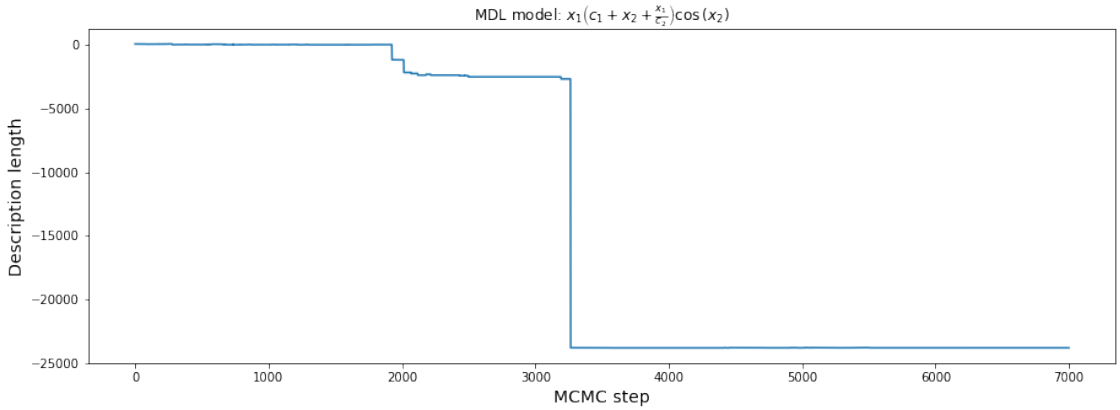
So let's take a look at the objects we stored. Here is the best model sampled by the machine scientist:

[4]:
```
print('Best model:\t', best_model)
print('Desc. length:\t', lowest_mdl)
```

```
Best model:      (((_a0_ + ((x1 / _a1_) + x2)) * cos(x2)) * x1)
Desc. length:    -23831.1721859678
```

And here is the trace of the description length:

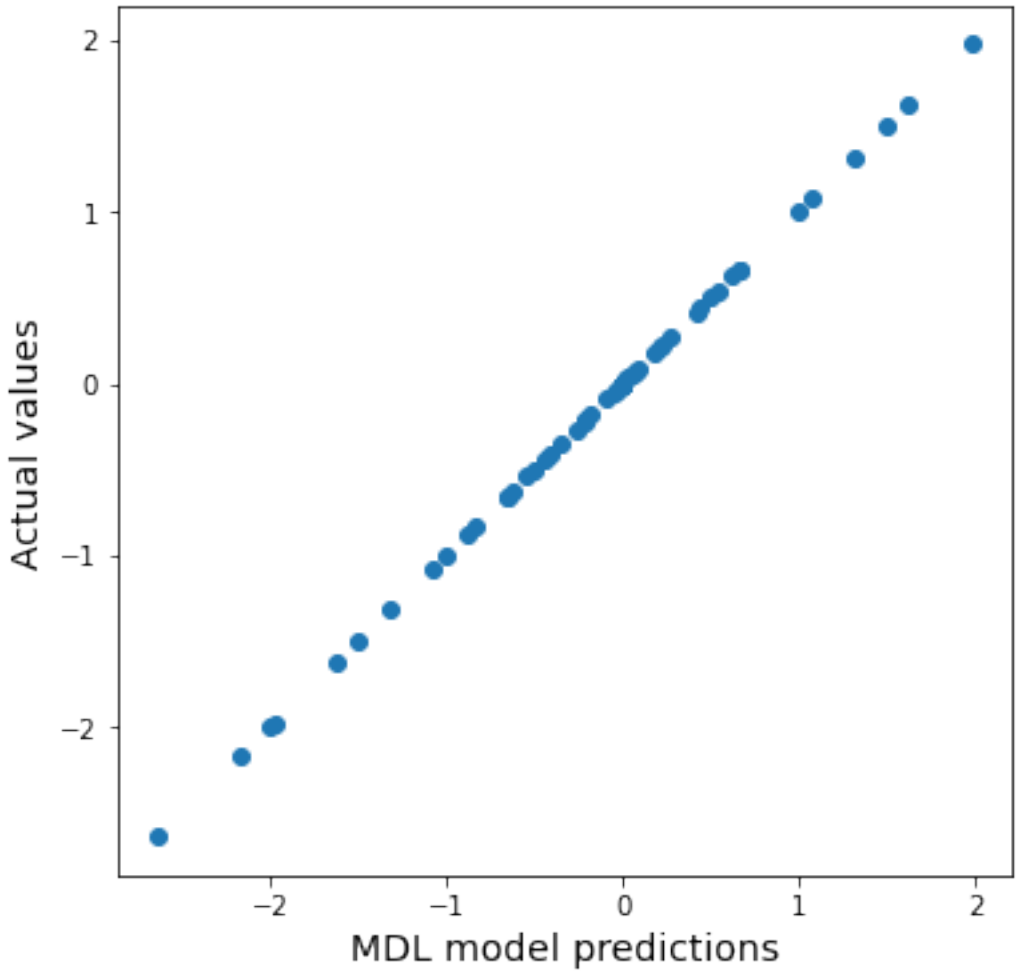
[5]:
```
plt.figure(figsize=(15, 5))
plt.plot(best_description_lengths)
plt.xlabel('MCMC step', fontsize=14)
plt.ylabel('Description length', fontsize=14)
plt.title('MDL model: $%s$' % best_model.latex())
plt.show()
```



1.3 Making predictions with the Bayesian machine scientist

Finally, we typically want to make predictions with models. In this regard, the interface of the machine scientist is similar to those in Scikit Learn: to make a prediction we call the `predict(x)` method, with an argument that has the same format as the training `x`, that is, a Pandas `DataFrame` with the exact same columns.

[6]:
```
plt.figure(figsize=(6, 6))
plt.scatter(best_model.predict(x), y)
#plt.plot((-6, 0), (-6, 0))
plt.xlabel('MDL model predictions', fontsize=14)
plt.ylabel('Actual values', fontsize=14)
plt.show()
```



1.4 Further refinements

The examples above are only intended to illustrate how a basic MCMC would be implemented. In practice, there are other considerations that we kept in mind in all the experiments reported in the manuscriot, and that anyone using the code should too: - **Equilibration**: One should not start sampling until the MCMC has converged to the stationary distribution. Although determining when a sample is in equilibrium, a necessary condition is that the description length is not increasing or, more typically, decreasing. The trace of the description length should be flat (except for fluctuations) before we start collecting samples. - **Thinning**: MCMC samples should be thinned, so only one in, say, 100 samples are kept for the trace. Otherwise, one is getting highly correlated samples, which may lead to, for example, erroneous estimates of confidence intervals. - **Getting trapped**: Despite the parallel tempering, the MCMC can get trapped in local minima of the description length. For this, we typically keep track of the number of steps since the last `tree_swap()` move was accepted for each temperature. If a particular temperature has *not* accepted swaps in a long time, then we anneal the whole system, that is, we increase all temperatures and decrease them slowly back to equilibrium so as to escape the local minima. Using several restarts of the MCMC and comparing the results is also a convenient check. - **Memory issues**: By default, the

machine scientist keeps a cache of all visited models, so as to avoid duplicates of previously considered models, as well as to speed up the process of obtaining the maximum likelihood estimators of the model parameters. For long MCMC chains this becomes memory intensive, so it may be convenient to periodically clean this cache (or, at least, old models in this cache) by reinitializing the `fit_pat` and `representative` attributes of the `Parallel` instance.

[ ]:
```
%%javascript
IPython.notebook.save_notebook()
IPython.notebook.kernel.execute('nb_name = "' + IPython.notebook.notebook_name␣
↪+ '"')
```

[8]:
```
save=input()
if save=='T':
    import os
    from datetime import datetime
    os.system(f"jupyter nbconvert --output-dir='./data/Tests/nd2pdf/' --output␣
↪'{nb_name[:-6]}_date_{datetime.now()}.pdf' --to pdf {nb_name}")
```

```
"""
if i%2000==0:
    print('Delete cache',len(pms.trees['1'].fit_par),len(pms.trees['1'].representative))
    pms.trees['1'].fit_par={}
    pms.trees['1'].representative={}
```

print(‘Delete cache’,len(pms.trees[‘1’].fit_par),len(pms.trees[‘1’].representative))
print(pms.trees[‘1’].fit_par) pms.trees[‘1’].fit_par={} pms.trees[‘1’].representative={} """