gp2FE

July 19, 2022

0.1 Gaussian process for two way fixed effects using gpytorch + pyro

import libraries

[ ]:
```
import torch
import gpytorch
import pyro
from pyro.infer.mcmc import NUTS, MCMC
from matplotlib import pyplot as plt
import seaborn as sns
import numpy as np
from gpytorch.priors import GammaPrior, NormalPrior
from gpytorch.lazy import InterpolatedLazyTensor
from gpytorch.utils.broadcasting import _mul_broadcast_shape
from gpytorch.constraints import Positive
import time
from gpytorch.kernels.kernel import Kernel
```

Define model The model has separated components for x, group and time. for x it is just a usual GP with SE kernel. for group and time i both used gp with the index kernel that has non zero variance only for the same group/time.

Input: [x, group, t]
y = GP(0, K_x, K_group, K_t)
K_x(x,x') = os^2 exp(-0.5(x-x')^2 / ls^2)
K_g(g,g') = sigma^2_g iff g==g' else 0
K_t(t,t') = sigma^2_t iff t==t' else 0

[ ]:
```
num_samples = 100
warmup_steps = 100
torch.set_default_tensor_type(torch.DoubleTensor)

class myIndexKernel(Kernel):
    r"""
    A kernel for discrete indices. Kernel is defined by a lookup table.

    .. math::

        \begin{equation}
```

```
            k(i, j) = v_i if i==j otherwise 0
        \end{equation}

    where :math:`\mathbf v` is a  non-negative vector.
    These parameters are learned.

    Args:
        :attr:`num_tasks` (int):
            Total number of indices.
        :attr:`batch_shape` (torch.Size, optional):
        :attr:`prior` (:obj:`gpytorch.priors.Prior`):
            Prior for :math:`v` vector.

    Attributes:
        raw_var:
            The element-wise log of the :math:`\mathbf v` vector.
    """

    def __init__(self, num_tasks, prior=None, **kwargs):
        super().__init__(**kwargs)
        self.num_tasks = num_tasks
        self.register_parameter(name="raw_var", parameter=torch.nn.
↪Parameter(torch.randn(*self.batch_shape, 1)))
        if prior is not None:
            self.register_prior("raw_var_prior", prior, lambda m: m.var,
             lambda m, v: m._set_var(v))

        self.register_constraint("raw_var", Positive())

    @property
    def var(self):
        return self.raw_var_constraint.transform(self.raw_var)

    @var.setter
    def var(self, value):
        self._set_var(value)

    def _set_var(self, value):
        self.initialize(raw_var=self.raw_var_constraint.
↪inverse_transform(value))

    def _eval_covar_matrix(self):
        return torch.diag_embed(self.var*torch.ones((self.num_tasks,)))

    def forward(self, i1, i2, **params):

        i1, i2 = i1.long(), i2.long()
        covar_matrix = self._eval_covar_matrix()
        batch_shape = _mul_broadcast_shape(i1.shape[:-2], i2.shape[:-2], self.
↪batch_shape)

        res = InterpolatedLazyTensor(
            base_lazy_tensor=covar_matrix,
            left_interp_indices=i1.expand(batch_shape + i1.shape[-2:]),
            right_interp_indices=i2.expand(batch_shape + i2.shape[-2:]),
        )
        return res

class FEGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood, num_group, num_time):
        super(FEGPModel, self).__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ZeroMean()
        self.x_covar_module = gpytorch.kernels.ScaleKernel(gpytorch.kernels.
↪RBFKernel())
        self.group_covar_module = myIndexKernel(num_tasks=num_group,␣
↪prior=GammaPrior(1,1))
        self.t_covar_module = myIndexKernel(num_tasks=num_time,␣
↪prior=GammaPrior(1,1))
    def forward(self, x):
        # separate kernel for [x, group , time]
        mean_x = self.mean_module(x[:,0])
        covar_x = self.x_covar_module(x[:,0])
        covar_group = self.group_covar_module(x[:,1])
        covar_t = self.t_covar_module(x[:,2])
        covar = covar_x + covar_group + covar_t
        return gpytorch.distributions.MultivariateNormal(mean_x, covar)
```

Define data consider 50 units clustered by 10 groups, repeated observed over 10 time periods. These units are associated with time varying covariate x_{it}~N(0,1), time-invariant group-level fixed effects e_g(g)~N(0,1) and group-invariant time-level fixed effects e_t(t)~N(0,1). The observation model is y_{it} = 2sin(4x) + e_g(g) + e_t(t) + noise, noise~N(0,0.1^2)

[ ]:
```
# simulate synthetic data
pyro.set_rng_seed(12345)
rng = np.random.default_rng(12345)
n = 50 # number of units
m = 10 # number of groups
T = 10 # number of time
noise_scale = 0.1**2 # noise

# x/fixed effects
x = np.random.normal(loc=0,scale=1, size=(n,T))
group_fes = np.random.normal(loc=0,scale=1, size=(m,))
time_fes = np.random.normal(loc=0,scale=1, size=(T,))
```

```
train_x = np.zeros((n*T,3))
train_y = np.zeros((n*T,))
for i in range(n):
    for t in range(T):
        group = int(i/m)
        train_x[i*T+t,0] = x[i,t]
        train_x[i*T+t,1] = group
        train_x[i*T+t,1] = t
        train_y[i*T+t] = 2*np.sin(4*x[i,t]) + group_fes[group] \
             + time_fes[t] + noise_scale*np.random.normal(size=(1,))

train_x = torch.tensor(train_x)
train_y = torch.tensor(train_y.reshape((-1,)))
```

0.2 Build gpytorch model and find MAP

[ ]:
```
# Use a positive constraint instead of usual GreaterThan(1e-4) so that␣
 ↪LogNormal has support over full range.
likelihood = gpytorch.likelihoods.GaussianLikelihood(
        noise_constraint=Positive(),
        noise_prior=GammaPrior(1,5))
model = FEGPModel(train_x, train_y, likelihood, num_group=m, num_time=T)

model.x_covar_module.base_kernel.register_prior("lengthscale_prior",␣
 ↪GammaPrior(1, 2), "lengthscale")
model.x_covar_module.register_prior("outputscale_prior", GammaPrior(1, 1),␣
 ↪"outputscale")

mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

hypers = {
    'likelihood.noise_covar.noise': torch.tensor(0.1),
    'x_covar_module.base_kernel.lengthscale': torch.tensor(1.0),
    'x_covar_module.outputscale': torch.tensor(1.0),
    'group_covar_module.var': torch.tensor(1.0),
    't_covar_module.var': torch.tensor(1.0)
}

model.initialize(**hypers)

# Find optimal model hyperparameters
model.train()
likelihood.train()

# Use the adam optimizer
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)  # Includes␣
 ↪GaussianLikelihood parameters

for i in range(50):
    optimizer.zero_grad()
    output = model(train_x)
    loss = -mll(output, train_y)
    loss.backward()
    print('Iter %d/50 - Loss: %.3f' % (i + 1, loss.item()))
    optimizer.step()

for param_name, param in model.named_parameters():
    print(f'Parameter name: {param_name:42} value = {param.item()}')
```

```
Iter 1/50 - Loss: 6.290
Iter 2/50 - Loss: 4.979
Iter 3/50 - Loss: 3.997
Iter 4/50 - Loss: 3.306
Iter 5/50 - Loss: 2.841
Iter 6/50 - Loss: 2.528
Iter 7/50 - Loss: 2.309
Iter 8/50 - Loss: 2.145
Iter 9/50 - Loss: 2.017
Iter 10/50 - Loss: 1.914
Iter 11/50 - Loss: 1.828
Iter 12/50 - Loss: 1.756
Iter 13/50 - Loss: 1.696
Iter 14/50 - Loss: 1.644
Iter 15/50 - Loss: 1.600
Iter 16/50 - Loss: 1.563
Iter 17/50 - Loss: 1.531
Iter 18/50 - Loss: 1.503
Iter 19/50 - Loss: 1.480
Iter 20/50 - Loss: 1.460
Iter 21/50 - Loss: 1.442
Iter 22/50 - Loss: 1.428
Iter 23/50 - Loss: 1.415
Iter 24/50 - Loss: 1.404
Iter 25/50 - Loss: 1.395
Iter 26/50 - Loss: 1.387
Iter 27/50 - Loss: 1.380
Iter 28/50 - Loss: 1.375
Iter 29/50 - Loss: 1.370
Iter 30/50 - Loss: 1.366
Iter 31/50 - Loss: 1.362
Iter 32/50 - Loss: 1.359
Iter 33/50 - Loss: 1.357
Iter 34/50 - Loss: 1.354
Iter 35/50 - Loss: 1.352
Iter 36/50 - Loss: 1.351
Iter 37/50 - Loss: 1.350
Iter 38/50 - Loss: 1.348
Iter 39/50 - Loss: 1.348
Iter 40/50 - Loss: 1.347
Iter 41/50 - Loss: 1.346
Iter 42/50 - Loss: 1.346
Iter 43/50 - Loss: 1.345
Iter 44/50 - Loss: 1.345
Iter 45/50 - Loss: 1.344
Iter 46/50 - Loss: 1.344
Iter 47/50 - Loss: 1.344
Iter 48/50 - Loss: 1.344
Iter 49/50 - Loss: 1.344
Iter 50/50 - Loss: 1.344
Parameter name: likelihood.noise_covar.raw_noise           value =
-0.037334923091634796
Parameter name: x_covar_module.raw_outputscale             value =
1.8889086804256978
Parameter name: x_covar_module.base_kernel.raw_lengthscale value =
-0.7774248377758834
Parameter name: group_covar_module.raw_var                 value =
-0.41871831124693054
Parameter name: t_covar_module.raw_var                     value =
-3.186221034556271
```

Define pyro model and run

[ ]:
```python
# define pyro model
def pyro_model(x, y):
    with gpytorch.settings.fast_computations(False, False, False):
        sampled_model = model.pyro_sample_from_prior()
        output = sampled_model.likelihood(sampled_model(x))
        pyro.sample("obs", output, obs=y)
        loss = -mll(output,y)
    return loss

model.double()
likelihood.double()

# Sample model hyperparameters
model.train()
likelihood.train()

nuts_kernel = NUTS(pyro_model, adapt_step_size=True, jit_compile=False)
mcmc_run = MCMC(nuts_kernel, num_samples=num_samples, warmup_steps=warmup_steps)
```

```
start = time.time()
mcmc_run.run(train_x, train_y)
end = time.time()
print(end - start)
```

```
Sample: 100%|| 200/200 [02:41,  1.24it/s, step size=8.80e-01, acc.
```

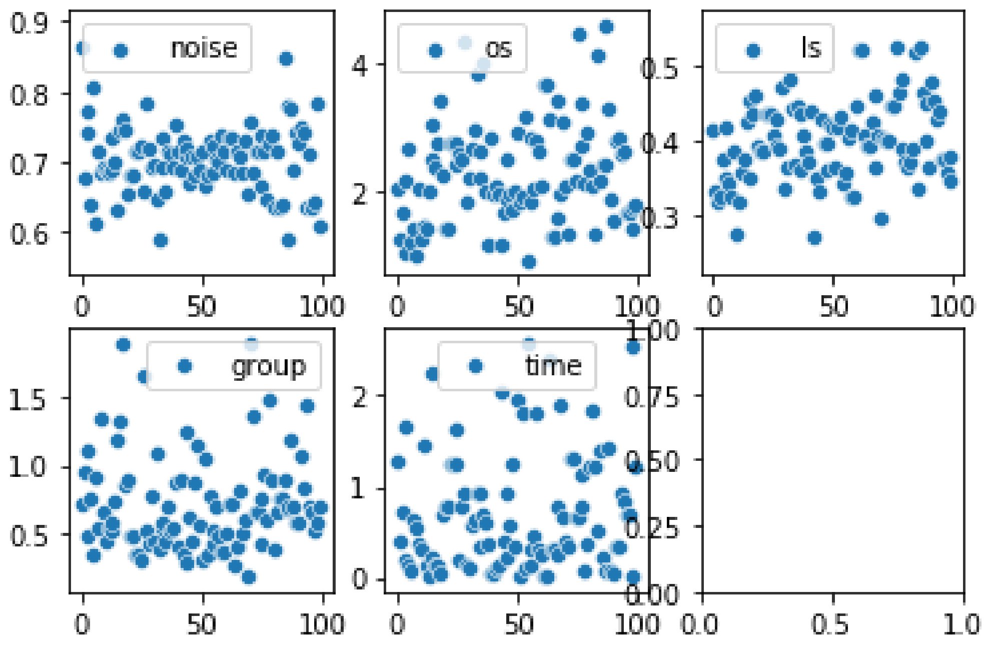
```
161.90221405029297
```

Plot model parameter posteriors

[ ]:
```
mcmc_samples = mcmc_run.get_samples()
# plot sampler trace
param_list = ['likelihood.noise_covar.noise_prior',  \
    'x_covar_module.outputscale_prior', 'x_covar_module.base_kernel.
 ↪lengthscale_prior',\
        'group_covar_module.raw_var_prior', 't_covar_module.raw_var_prior']
labels = ["noise", "os","ls", "group", "time"]
fig, axes = plt.subplots(nrows=2, ncols=3)
for i in range(5):
        samples = mcmc_samples[param_list[i]].numpy().reshape(-1)
        sns.scatterplot(range(num_samples) ,samples, ax=axes[int(i/3),␣
 ↪int(i%3)])
        axes[int(i/3)][int(i%3)].legend([labels[i]])
plt.show()
```

```
/Users/yahoo/anaconda3/lib/python3.7/site-packages/seaborn/_decorators.py:43:
FutureWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
  FutureWarning
/Users/yahoo/anaconda3/lib/python3.7/site-packages/seaborn/_decorators.py:43:
FutureWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
  FutureWarning
/Users/yahoo/anaconda3/lib/python3.7/site-packages/seaborn/_decorators.py:43:
FutureWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
  FutureWarning
/Users/yahoo/anaconda3/lib/python3.7/site-packages/seaborn/_decorators.py:43:
```

```
FutureWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
  FutureWarning
/Users/yahoo/anaconda3/lib/python3.7/site-packages/seaborn/_decorators.py:43:
FutureWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
  FutureWarning
```


noise
os
ls
group
time