_op _cov

The GaussianProcess package

Release 0.0

Anand Patil

April 5, 2007

CONTENTS

1 Introduction **1**

2 Installation **3**

2.1 Dependencies 3

2.2 How to install 3

3 Tutorial **5**

3.1 Prerequisites 5

3.2 A first look at Gaussian processes 5

3.3 The role of the covariance function 9

3.4 Nonparametric regression: observing Gaussian processes 9

3.5 Integrals, derivatives and transforms: linear operations and constraints 9

3.6 The array aspect and performance 9

3.7 Incorporating Gaussian processes in probability models with PyMC 10

3.8 Writing your own covariance functions 10

4 API reference **11**

CHAPTER

ONE

Introduction

Gaussian processes (GPs) are probability distributions for functions. They're useful if a functional form is unknown a priori.

GPs are not hard to understand at a conceptual level, but implementing them on a computer can require fairly involved linear algebra. This makes it hard for beginners to get started, and even when you have experience it's annoying.

This package provides Python classes that represent the components of the Gaussian process. They are meant to support many types of Gaussian process usage, from intuitive exploration to high-performance deployment in MCMC, with smooth transitions between.

The package also provides a class which produces Gaussian process-valued PyMC parameters and a PyMC sampling method to handle it. That means the Gaussian process-related objects can be directly incorporated into larger probability models.

You'll need to understand Python, numpy and a little bit of Bayesian statistics. Refs.

CHAPTER

TWO

Installation

2.1 Dependencies

- *Python version 2.4* or later.
- *numpy*, Numerical Python. The core package for numerical work in Python.
- Optional: *matplotlib/ pylab*, Matlab-like scientific plotting utilities for Python. Install this if you want to use any of the graphical functionality described in this manual.
- Optional: *PyMC*, Bayesian statistics and MCMC support for Python. Install this if you want to embed Gaussian processes in larger probability models.

2.2 How to install

Type this into a shell:

```
python setup.py install
```

etc.

CHAPTER

THREE

Tutorial

To understand GPs, you need to play around with them. You can be as good at math as you want, but to really know what they're like you have to build experience. For that reason, after a brief overview of what GP's can be good for, this tutorial will immediately show you how to actually instantiate a GP. That way you have one to play around with throughout the rest of the tutorial. All the code in the tutorial is in the folder `examples`, which is distributed with this package.

3.1 Prerequisites

You need to know a bit about Python and numpy. Also a bit about Bayesian statistics and the normal distribution. Refs.

3.2 A first look at Gaussian processes

Gaussian processes are probability distributions for functions. The statement 'f has a Gaussian process distribution with mean M and covariance C' is usually written as follows:

$$f \sim \text{GP}(M, C). \tag{3.1}$$

Gaussian processes have two parameters, which are analogous to the parameters of the normal distribution:

- M is the mean function. Like the mean parameter of the normal distribution, M gives the central tendency for f. In Bayesian statistics, M is usually considered a prior guess for f. $M(x)$ gives the expectation of $f(x)$.
- C is the covariance function. Its role is harder to understand than that of the mean function, but among other things it regulates:
  - the amount by which f may deviate from M
  - the smoothness of f
  - the wiggliness of f.

  $C(x, y)$ gives the covariance of $f(x)$ and $f(y)$; $C(x, x)$ gives the variance of $f(x)$.

An intuitive understanding of covariance functions is essential for appropriate application of Gaussian processes, but for the time being don't worry about them too much. Section 3.3 is all about covariance functions.

A preliminary example- use Steve's paper

3.2.1 Instantiating a Gaussian process

The rest of this tutorial will be much easier to understand with an actual Gaussian process in hand to experiment with. The `GaussianProcess` module supports multivariate as well as univariate GP's; the variables x and y in section 3.2 can be vectors just as well as scalars. For the time being, however, let's concentrate on univariate GP's, as they're easier to visualize.

Instantiating a covariance function

The first component of a GP that we will generate is a covariance function. The covariance function of a univariate GP is a function of two variables. In this example we will use the function

$$C(x,y) = a\, e^{-\left(\frac{|x-y|}{s}\right)^p}, \tag{3.2}$$

where a, s and p are tunable parameters. This is known as the 'power family' of covariance functions. It's not considered the best for most applications, but it's simple to write down.

Not every function of two variables is acceptable as a covariance function. In fact, our function C is only acceptable if $a > 0$, $s > 0$ and $0 < p \leq 2$. If you want to write your own covariance function at some point, this module provides a function called `isPosDef` that can help you determine if your function is acceptable. That function is described in more detail later. However, this module provides a small library of readymade covariance functions implemented in Fortran, including the popular Matérn covariance function, in the package `cov_funs`.

The code shown below will produce an instance of class `Covariance` called C. It will also display the covariance function $C(x,y)$ as a filled contour plot on the mesh ($-1 < x < 1$,$-1 < y < 1$).

```
# GaussianProcess/examples/cov.py

from GaussianProcess import *
from numpy import *

def exp_cov(x,y,p,a,s):
    """
    a and s must be positive
    p must be positive and less than 2
    """

    C = zeros((len(x), len(y)))
    for i in xrange(len(x)):
        for j in xrange(len(y)):
            C[i,j] = a * exp(-(abs(x-y) / s) ** p)
    return C

C = Covariance(eval_fun = exp_cov, p = .49, a = 1., s = .3)

x=arange(-1.,1.,.1)
contourf(x, x, C(x,x))
colorbar()
```

The first argument, `eval_fun`, gives the Python function from which the covariance function will be made, in this case `exp_cov`. The `Covariance` class will raise an error if it is instantiated around an unacceptable function. The extra arguments `p, a` and `s` will be passed to `exp_cov`.

At this stage, the covariance function exposes a very simple user interface. In fact, it behaves a lot like its input function `exp_cov`. Try calling it. The only differences are: you only call it with x and y meshes, not with the extra arguments p, s and a; and you can just call it with one mesh, in which case it'll assume that the other mesh is the same and return a square matrix. Oh yeah- if you call it with array arguments, it'll return a matrix which gives it evaluated over all the values in the mesh. This behavior is handy. Input functions have to behave that way, too.

The last line plots the covariance as a filled contour plot evaluated on a mesh. The width of the covariance function band controls how tightly nearby evaluations of f are coupled to each other. You'll notice it looks like a band. If there's a wide band, $f(x)$ and $f(y)$ will tend to have similar values. If there's a narrower band, $f(x)$ and $f(y)$ won't be as tightly correlated. The height of the covariance function band controls the overall amplitude of f's deviation from M. Try playing around with the parameters of `exp_cov` and see how C looks.

Instantiating a mean function

The second component we will generate is the mean function. The mean function of a univariate GP can be interpreted as a prior guess for the GP, so it's a univariate function also. Unlike the covariance function, there are no restrictions whatsoever on the mean function; any univariate function is fine. We will use the parabola

$$M(x) = ax^2 + bx + c. \tag{3.3}$$

The following code will produce an instance of class `Mean` called M, which will be associated with C. This association is unnecessary in this part of the tutorial, but it's essential for nonparametric regression and other conditioning operations, which we'll discuss later.

```
# GaussianProcess/examples/mean.py

from GaussianProcess import *
from numpy import *

def exp_cov(x,y,p,a,s):
    """
    a and s must be positive
    p must be positive and less than 2
    """

    C = zeros((len(x), len(y)))
    for i in xrange(len(x)):
        for j in xrange(len(y)):
            C[i,j] = a * exp(-(abs(x-y) / s) ** p)
    return C

C = Covariance(eval_fun = exp_cov, p = .49, a = 1., s = .3)

def linfun(x, a, b, c):
    return a * x ** 2 + b * x + c

M = Mean(eval_fun = linfun, C = C, a = 1., b = .5, c = 2.)
x=arange(-1.,1.,.1)
plot(x, M(x), 'k-')
```

As with `Covariance`, the first argument to `Mean`'s init method is a Python function, in this case `linfun`. The second argument, `C`, is the `Covariance` instance with which the mean function is associated. The extra arguments `a`, `b` and `c` will be passed to `linfun`.

Like C, M behaves a lot like an ordinary numpy universal function. The last line plots $M(x)$ on $-1 < x < 1$, and as expected the plot is a parabola. Mean functions are much less magical than covariance functions. As stated before, they're just an initial guess for f. In fact, they can be removed from the GP and added back in at will:

$$f \sim \text{GP}(M, C) \tag{3.4}$$

$$\Rightarrow f = M + g, \quad \text{where} \quad g \sim \text{GP}(0, C). \tag{3.5}$$

Drawing realizations

Finally, let's generate some realizations (draws) from the Gaussian process defined by M and C and take a look at them. The following code will generate a list of instances of class `Realization` called `f_list`:

```
# GaussianProcess/exales/realizations.py

from GaussianProcess import *
from numpy import *

def exp_cov(x,y,p,a,s):
    """
    a and s must be positive
    p must be positive and less than 2
    """

    C = zeros((len(x), len(y)))
    for i in xrange(len(x)):
        for j in xrange(len(y)):
            C[i,j] = a * exp(-(abs(x-y) / s) ** p)
    return C

C = Covariance(eval_fun = exp_cov, p = .49, a = 1., s = .3)

def linfun(x, a, b, c):
    return a * x ** 2 + b * x + c

M = Mean(eval_fun = linfun, C = C, a = 1., b = .5, c = 2.)

f_list=[]
for i in range(3):
    f = Realization(M, C)
    f_list.append(f)

x=arange(-1.,1.,.1)

plot_envelope(M, C, mesh=x)

for f in f_list
    plot(x, f(x))
```

The init method of `Realization` takes only two arguments, a mean function and a covariance function. Each element of `f_list` is a Gaussian process realization, which is a randomly-drawn function. Like ordinary numpy universal functions, GP realizations can be called with either values or ndarrays as arguments.

The third-to-last last line calls the function `plot_envelope`, which kind of summarizes the distribution. The dashdot black line in the middle is M, and the gray band is the ± 1 standard deviation envelope generated by C. Each realization is a callable function, and their values over a mesh are plotted superimposed on the envelope.

Experiment!

As stated before, no matter how much math you know, the only way you'll be able to use GPs with confidence is by getting some experience with them. Now is a good time to have fun playing with the parameters of the mean and covariance functions. Change parameters of the covariance function, see how it looks using the code in section 3.2.1, and try to guess how the realizations will look. See if you're right. Try generating M around a different function. Try generating C around one of the covariance functions from `cov_funs`, or if you're feeling adventurous try writing your own.

3.3 The role of the covariance function

Various visualization methods of covariance, relationship between covariance's shape and differentiability of realizations.

3.4 Nonparametric regression: observing Gaussian processes

The condition function, what it does. Some ecological examples.

3.5 Integrals, derivatives and transforms: linear operations and constraints

Define linear operations, talk about derivatives, antiderivatives, integrals and Fourier transforms. Say that you can constrain the value of any linear operation, or softly constrain it. Introduce the lintrans argument.

3.6 The array aspect and performance

Conceptually, Gaussian process realizations are random functions, and the parameters of the Gaussian process are functions. To build intuition as quickly as possible, this tutorial has tried to stay faithful to the concepts by focussing on the *functional* aspect of `Covariance`, `Mean` and `Realization` for as long as possible.

From the package author's point of view, this was the hard part. Making `Realization` act like a function is particularly difficult; its return values have to be evaluated on-demand (lazily) using a fairly complicated and expensive algorithm whose expense increases with the number of calls made so far.

You may have noticed by now that `Covariance` is a subclass of `numpy.matrix`, and that `Mean` and `Realization` are subclasses of `numpy.ndarray`. That's because these objects are usually represented on a computer as arrays and matrices rather than functions. Certain properties of the Gaussian process distribution make this representation work well, though it requires a fair bit of linear algebra. Unfortunately, using the array representation of the Gaussian process is much faster than using the functional representation we have considered so far. If you need more performance from the Gaussian process than you've been getting so far, you may need to take a look at this section.

This section will try to introduce you to the array aspect of the GP, while insulating you from as much linear algebra as possible. The first subsection will tell you the bare minimum you need to know to use the array aspect, and subsequent sections will go into more depth.

3.6.1 Utilitarian overview: the base mesh

The base mesh is where you should put evaluations you know ahead of time you're going to want. Redo examples with a base mesh. Indexing and slicing of objects. If you do MCMC, you're NEVER allowed to change the base mesh. DON'T DO IT. Talk about drawing realizations but specifying values on a base mesh.

3.6.2 Some light theory

$$\begin{aligned} &x \sim \mathrm{N}(\mu, V) && x, y, V \text{ are scalars} \\ &\vec{x} \sim \mathrm{N}(\vec{\mu}, C) && \vec{x} \text{ and } \vec{\mu} \text{ are vectors, } C \text{ is a scalar} \\ &f \sim \mathrm{GP}(M, C) && f \text{ and } M \text{ are functions of one variable, } C \text{ is a function of two variables} \end{aligned} \tag{3.6}$$

3.7 Incorporating Gaussian processes in probability models with PyMC

3.8 Writing your own covariance functions

CHAPTER

FOUR

API reference

Doxygen-style class reference here.