Single Partition Regression Analysis

Rhys Hawkins and Malcolm Sambridge

September 25, 2012

This tutorial describes how to analyse continuous data using Monte-Carlo Markov-Chain based regression.

Pre-requisites

This tutorial is for the Python version of the rjmcmc library. The examples rely on the Matplotlib library for plotting. The versions used in the development of this tutorial are as follows:

- Python 2.7.1
- rjmcmc 0.1.0
- Matplotlib 1.1.0

The Data

For this tutorial we will use a non-trivial (in the sense that it will require a higher order polynomial to fit the function correctly) synthetic dataset with added noise.

The function that is used is an exponentially increasing sine wave over the domain 0 . . . 10, ie:

$$y = e^{\frac{x}{3}} \sin \frac{2x}{3} \tag{1}$$

The actual dataset unevenly (though with fairly good coverage) samples this function and adds some Gaussian noise and these values are save to an ASCII text file. A plot of the synthetic data points with the true function is shown in Figure 1.

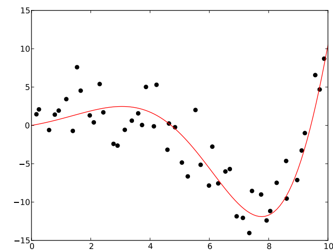

Figure 1: The Synthetic Data.

Loading the Data

For those unfamiliar with Python, we discuss briefly the loading of data from a simple ASCII text file. The code for loading the data and doing a simple plot is shown in Listing 1.

Listing 1: Loading the Data.

```
1  #
2  # Import the libraries we will need for analysis and plotting.
3  #
4  import rjmcmc
5  import matplotlib
6  import matplotlib.pyplot
7
8  #
9  # Open our data file which consists of one (x, y) coordinater per line
```

```
10  # separated by whitespace
11  #
12  f = open('data.txt', 'r')
13  lines = f.readlines()
14
15  x = []
16  y = []
17
18  for line in lines:
19      columns = line.split()
20
21      x.append(float(columns[0]))
22      y.append(float(columns[1]))
23
24  f.close()
25
26  fig = matplotlib.pyplot.figure()
27
28  matplotlib.pyplot.plot(x, y, 'ko')
29
30  fig.savefig('ch1-loading.pdf', format='PDF')
31  matplotlib.pyplot.show()
```

The ASCII file contains an x, y pair per line separated by a space. On line 12, we use the built in function `open` to open the file. One lines 15 and 16 we initialize 2 list that will contain the x and y coordinates in the file. On line 18 we loop through the lines in the file and add each x, y pair to the 2 separate lists.

From line 26 onwards, we plot the data using the matplotlib library and save the plot to a PDF file. The plot resulting from this script can be seen in Figure 2.

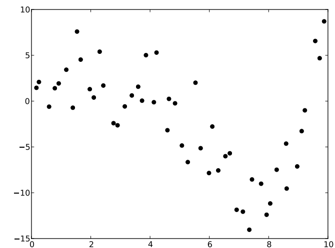

Figure 2: Noisy data and true curve used in test problem.

Running the default analysis

For performing an regression analysis on a continuous dataset the function is called `regression_single1d`. The parameters for this function are as follows with default values shown where applicable:

dataset The dataset object to run the analysis on. This is an rjmcmc.dataset1d object which wraps the x and y vectors you load from the file and includes individual point noise values. This is the only parameter which doesn't have a default value.

burnin = *10000* The number of initial samples to throw away.

total = *50000* The total number of samples to use for the analysis.

max_order = *5* The maximum order of polynomial to use to fit the data.

xsamples = *100* The number of points to sample along the x direction for the curve.

ysamples = *100* The number of points to sample along the y directory for the statistics such as mode, median and confidence intervals. This is the number of bins for the histograms in the y direction.

confidence_interval = *0.95* The confidence interval to use for minimum and maximum confidence intervals. This should be a value between 0 and 1.

For this analysis we are only going to use the default values and the listing is shown in Listing 2.

Listing 2: Running the Default Analysis.

```
#
# Import the libraries we will need for analysis and plotting.
#
import rjmcmc
import matplotlib
import matplotlib.pyplot

#
# Open our data file which consists of one (x, y) coordinate per line
# separated by whitespace
#
f = open('data.txt', 'r')
lines = f.readlines()

x = []
y = []

for line in lines:
    columns = line.split()

    x.append(float(columns[0]))
    y.append(float(columns[1]))

f.close()

#
# Estimate our error standard deviation
#
sigma = 3.0
n = [sigma] * len(x)

#
# Create the rjmcmc dataset
#
data = rjmcmc.dataset1d(x, y, n)

#
# Run the default analysis
#
results = rjmcmc.regression_single1d(data)

#
# Retrieve the mean curve for plotting
#
xc = results.x()
meancurve = results.mean()

#
# Plot the data with black crosses and the mean with a red line
#
fig = matplotlib.pyplot.figure()
matplotlib.pyplot.plot(x, y, 'ko', xc, meancurve, 'r-')
fig.savefig('ch2-analyse.pdf', format='PDF')
matplotlib.pyplot.show()
```

The preamble (lines 1 ... 24) consists of loading the file as in the previous section.

An important part of the analysis is estimating the error in the data. This is specified as a error value per data point and can be thought of a weighting as to how well the fit will attempt to fit an individual point. If the value is low, then the fit will be tight and conversely if the value is high then the fit will be loose. On lines 29 and 30 we set a value of 3.0 for all data points. Use this value for now, but try other values greater than 0.0 to see the effect.

On line 35 we construct the dataset1d object from the x, y and n lists we created. These lists must be the same length.

On line 40 we run the analysis with this dataset1d object. The regression_single1d function returns a resultset1d object which contains various results and diagnostics about the analysis. For this simple analysis we simply take the x sampling coordinates and the mean of the fits. And plot the mean with the original data points to

see how representative the mean is. This plot is shown in Figure 3.

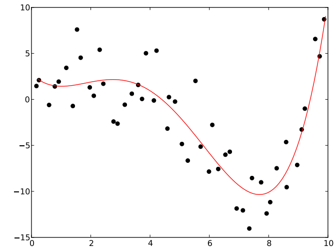

Figure 3: The Default Analysis Plot.

Order Analysis

A question we may ask is what order best represents the underlying function. We can develop some understanding of this by constraining the maximum allowable order of the fit and observing when the order histogram converges and/or the mean of the fits converges. The script to do this for this dataset is shown in Listing 3.

Listing 3: Order Analysis.

```
#
# Import the libraries we will need for analysis and plotting.
#
import rjmcmc
import matplotlib
import matplotlib.pyplot
from mpl_toolkits.mplot3d import axes3d, Axes3D
import sys

#
# Open our data file which consists of one (x, y) coordinater per line
# separated by whitespace
#
f = open('data.txt', 'r')
lines = f.readlines()

x = []
y = []

for line in lines:
    columns = line.split()

    x.append(float(columns[0]))
    y.append(float(columns[1]))

f.close()

#
# Estimate our error standard deviation
#
sigma = 3.0
n = [sigma] * len(x)

#
# Create the rjmcmc dataset
#
data = rjmcmc.dataset1d(x, y, n)

#
# Run a series of analyses with varying maximum allowed order
#
results = []
burnin = 100
total = 1000
orderlimit = 5
for maxorder in range(orderlimit + 1):
    print maxorder
    results.append(rjmcmc.regression_single1d(data, burnin, total, maxorder))

colours = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'w']
formats = map(lambda x: x + '-', colours)

#
# Plot the data with black crosses the curves from each of the analyses
# with a different colour
#
fig = matplotlib.pyplot.figure(1)
ax = fig.add_subplot(111)

orders = []
legendtitles = []
for result in results:

    order = result.order_histogram()
    if order == None: # The max order = 0 case will return None so
        order = [total]

    ax.plot(result.x(), result.mean(), formats[len(orders)])

    #
    # Create the order histogram data (append zeros for orders not allowed
    # in the analyses
    #
    legendtitles.append('Max. Order %d' % len(orders))
    orders.append(order + [0] * (orderlimit + 1 - len(order)))

ax.plot(x, y, 'ko')
```

```
78  legendtitles.append('Data')
79  legend = ax.legend(legendtitles, 'lower left')
80  fig.savefig('ch3-orderanalysis.pdf', format='PDF')
81
82  #
83  # Plot a 3D bar chart showing the progression of the order histogram
84  # as the analysis maximum order is increased.
85  #
86  fig = matplotlib.pyplot.figure(2)
87  ax = Axes3D(fig)
88  xs = range(orderlimit + 1)
89  for maxorder in xs:
90      ax.bar(xs,
91              orders[maxorder],
92              zs=maxorder,
93              zdir = 'y',
94              color=colours[maxorder])
95
96  ax.set_xlabel('Order')
97  ax.set_ylabel('Maximum Order')
98  ax.set_zlabel('Count')
99
100 fig.savefig('ch3-orderanalysishist.pdf', format='PDF')
101
102 matplotlib.pyplot.show()
```

In lines 41 ... 46 we set some parameters and run several analyses with different allowed maximum polynomial order and store the results.

In lines 51 ... 78 we plot the mean of the fits for each of the constrained analyses. This plot is shown in Figure 4. As can be seen from this plot, there is very little difference between the Max. Order 3, 4, 5 curves which implies that the data is cubic.

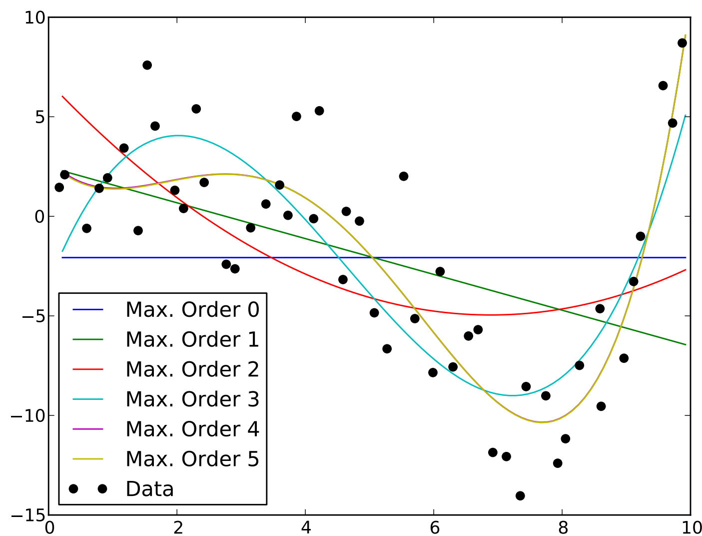

Figure 4: The Mean Curves for different maximum orders.

In lines 80 ... 99 we plot the progression of the order histogram. The order histogram reports the count of polynomials of each order where the order is randomly choosen from a probability distribution determined by the data itself. This plot is shown in Figure 5.

Looking at the figure, it can be seen that when the maximum permitted order is 0, all the curves sampled are 0th order as ex-

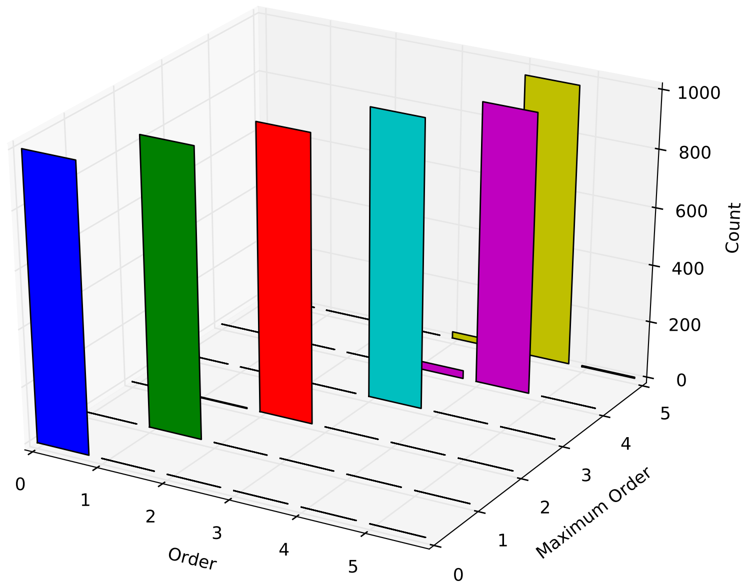

Figure 5: Posterior PDFs of the polynomial order parameter as a function of maximum allowed order.

pected. As this maximum order is increased, the distribution of orders changes until the step from 4 to 5 where there is little difference. It should be noted that if a tighter noise parameter is used, that this will change slightly as higher order polynomials will be more favoured to fit the data better.

Confidence Intervals

So far we have only plotted the mean of the fits, however this gives us no indication of distribution of the fit (this can be thought of as the confidence of the fit). There are a number of ways in which we can look at this and one of these is to look at the curves generated during the analysis. The listing to do this is in Listing 4.

Listing 4: Confidence Intervals.

```
#
# Import the libraries we will need for analysis and plotting.
#
import rjmcmc
import matplotlib
import matplotlib.pyplot
from mpl_toolkits.mplot3d import axes3d, Axes3D

#
# Open our data file which consists of one (x, y) coordinater per line
# separated by whitespace
#
f = open('data.txt', 'r')
lines = f.readlines()

x = []
y = []

for line in lines:
    columns = line.split()

    x.append(float(columns[0]))
```

```
    y.append(float(columns[1]))

f.close()

#
# Estimate our error standard deviation
#
sigma = 3.0
n = [sigma] * len(x)

#
# Create the rjmcmc dataset
#
data = rjmcmc.dataset1d(x, y, n)

#
# This is our callback function which samples the curves generated
# during the analysis
#
sample_x = None
sample_curves = []
sample_i = 0
sample_rate = 5
def sampler_cb(x, y):
    global sample_x, sample_curves, sample_i, sample_rate

    if sample_i == 0:
        sample_x = x

    if sample_i % sample_rate == 0:
        sample_curves.append(y)

    sample_i = sample_i + 1

#
# Run a series of analyses with varying maximum allowed order
#
results = []
burnin = 100
total = 1000
maxorder = 5
results = rjmcmc.regression_single1d_sampled(data,
                                             sampler_cb,
                                             burnin,
                                             total,
                                             maxorder)

#
# Plot the data with black crosses, the sample curves as faint lines, and
# the mean as a red line
#
fig = matplotlib.pyplot.figure()
ax = fig.add_subplot(111)

yc = 0.5
yalpha = 1.0/((1.0 - yc) * float(len(sample_curves)))
for sy in sample_curves:

    ax.plot(sample_x, sy,
            color = str(yc),
            alpha = yalpha,
            linestyle = '-',
            linewidth = 10)

ax.plot(results.x(), results.mean(), 'r-')
ax.plot(x, y, 'ko')
fig.savefig('ch4-confidence.pdf', format='PDF')

matplotlib.pyplot.show()
```

In this script we call a slightly different function called `regression_single1d_sampled` which accepts a callback function. We define this function on lines 46 ... 55. This function accepts an x and y list which is the discretization of the current fitting polynomial being used. In this function we sample every 5th polynomial and store them.

On lines 70 ... 90 we plot the data as black dots, and the mean fit with a red line over plots of all the fits we sampled. This plot is shown in Figure 6.

The sampled fits are plotted with transparency so that where they overlap this will show increased density implying that where these sampled polynomial ensemble appears darker, we can have higher confidence that the underlying function passes through that region.

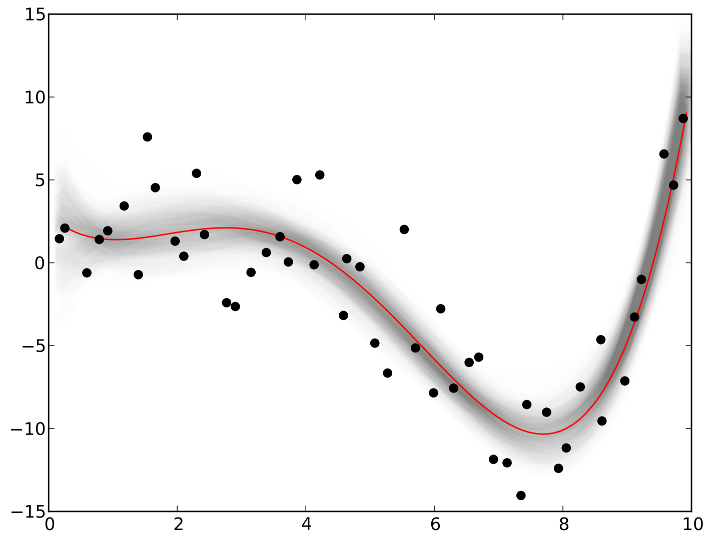

Figure 6: Sampled Confidence Intervals.

Estimating data noise

With the hierarchical Bayesian approach we include the standard deviation of the noise on the observations as an unknown. In the above examples the noise σ was set to 3 units, but the actual σ of the data noise in Figure 1 is 2.5. Can we use the data to detect the true standard deviation of its noise? The hierarchical Bayesian sampling scheme is implemented with the script in Listing 5. Inference on the noise is implemented by introducing a new parameter, $\lambda = \frac{\sigma_{est}}{\sigma_{true}}$, defined as the ratio of the estimated noise to the real noise.

Listing 5: Hierarchical Bayesian sampling to determine the data noise standard deviation

```
#
# Import the libraries we will need for analysis and plotting.
#
import rjmcmc
import matplotlib
import matplotlib.pyplot

#
# Open our data file which consists of one (x, y) coordinate per line
# separated by whitespace
#
f = open('data.txt', 'r')
lines = f.readlines()

x = []
y = []

for line in lines:
    columns = line.split()

    x.append(float(columns[0]))
    y.append(float(columns[1]))

f.close()

#
# Estimate our error standard deviation
#
```

```
29  sigma = 3.0
30  n = [sigma] * len(x)
31
32  #
33  # Create the rjmcmc dataset
34  #
35  data = rjmcmc.dataset1d(x, y, n)
36
37  lambda_min = 0.5
38  lambda_max = 2.0
39  lambda_std = 0.05
40
41  data.set_lambda_range(lambda_min, lambda_max)
42  data.set_lambda_std(lambda_std)
43
44
45  #
46  # Run the default analysis
47  #
48  results = rjmcmc.regression_single1d(data)
49
50  #
51  # Retrieve the mean curve for plotting
52  #
53  xc = results.x()
54  meancurve = results.mean()
55
56  #
57  # Retrieve the results of the hierarchical
58  #
59  p = results.proposed()
60  a = results.acceptance()
61
62  print 'Lambda Acceptance Rate:', float(a[1])/float(p[1]) * 100.0
63
64  lh = results.lambda_history()
65
66  #
67  # Plot the data with black crosses and the mean with a red line
68  #
69  fig = matplotlib.pyplot.figure(1)
70  matplotlib.pyplot.plot(x, y, 'ko', xc, meancurve, 'r-')
71
72
73  fig = matplotlib.pyplot.figure(2)
74  matplotlib.pyplot.plot(range(len(lh)), lh)
75
76  fig = matplotlib.pyplot.figure(3)
77
78  a = matplotlib.pyplot.subplot(111)
79  lsamples = lh[10000:]
80
81  n, bins, patches = a.hist(lsamples, 100, range=(lambda_min, lambda_max))
82  a.set_title('Histogram of Lambda')
83  a.set_xlabel('Lambda')
84  a.set_ylabel('Count')
85  fig.savefig('ch5-hierarchical.pdf', format='PDF')
86
87  print 'Lambda average:', sum(lsamples)/float(len(lsamples))
88  matplotlib.pyplot.show()
```

In this script we set up a uniform prior on λ over a pre-determined range and use a Gaussian distribution to perturb the λ values during the Markov chain. The range of the values of λ as well as the standard deviation of the perturbation are parameter that must be chosen. These are set in lines 37-42.

We call the function `regression_single1d` to do the work. We plot the posterior PDF of the noise σ as a histogram. This plot is shown in Figure 7.

The histogram shows the support of the data for a range of λ values. Clearly there is information in the data on the likely values of noise. Where is the peak of the histogram ? How does this compare to the ratio of the estimated to true σ? Usually the ability of the data to constrain noise parameters will trade-off with the model complexity given, in this case, by the order of the polynomial. You can edit the script by changing the estimated noise and rerun to see what happens.

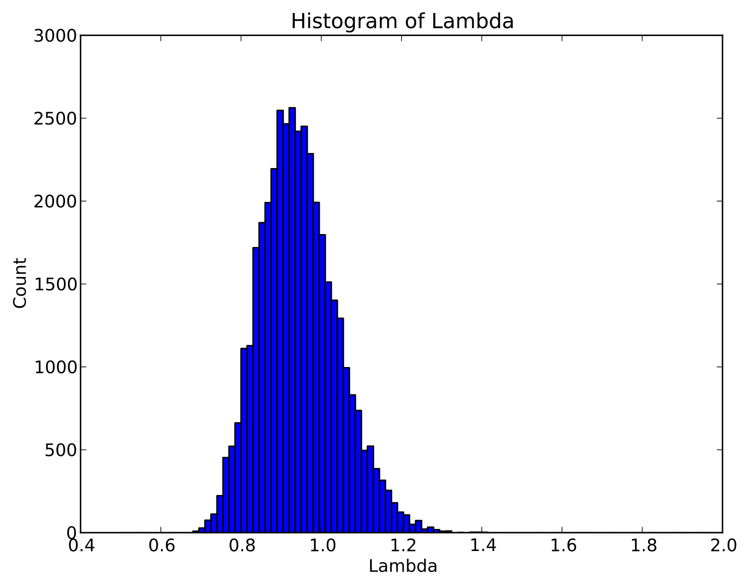

Figure 7: Posterior PDF of the noise standard deviation parameter λ.