Optimization CW

Aylin Sadikhova

November 2024

1 Regularised Least Squares

1.1 Introduction

I am given a time vector t_{data} and a noisy data set vector x_{noisy} for approximating exponential decay. In this problem the goal is to fit a regularized polynomial approximation to $x'(t)$.

1.2 Problem

In the instructions part of the coursework the data has already been generated and a spline approximation to the derivative made. My task was to solve this regularized least square problem

$$min_c \left(\|f(x) - x'(t)\| + \lambda \int (f''(x))^2 dx \right) \tag{1}$$

From the lecture notes 3, polynomial fitting of a set of points $\mathbb{R}^2$ (u_i, y_i) for the relationship

$$\mathbf{u}_i^T \mathbf{A} \approx y_i \tag{2}$$

where $\mathbf{u}_i^T = [1, u_i, u_i^2, ..., u_i^d]$ can be solved with least squares given all u_i are different from each other. In our case, with a regularisation parameter $\mathbf{D}$, the solution is of the form

$$\mathbf{x}_{RSL} = \left(\mathbf{U}^T\mathbf{U} + \lambda \mathbf{D}^T\mathbf{D} \right)^{-1} \mathbf{U}^T\mathbf{b} \tag{3}$$

1.3 Constructing the Matrices

The relationship we are interested in is between t_{data} and $x'(t)$ approximation. As can be seen, $\mathbf{u}_i$ are the rows of a vandermode matrix $\mathbf{U}$. Therefore

$$\mathbf{U} = \begin{bmatrix} 1 & t_1 & t_1^2 & \cdots & t_1^n \\ \vdots & & & & \vdots \\ 1 & t_{100} & t_{100}^2 & \cdots & t_{100}^n \end{bmatrix} \tag{4}$$

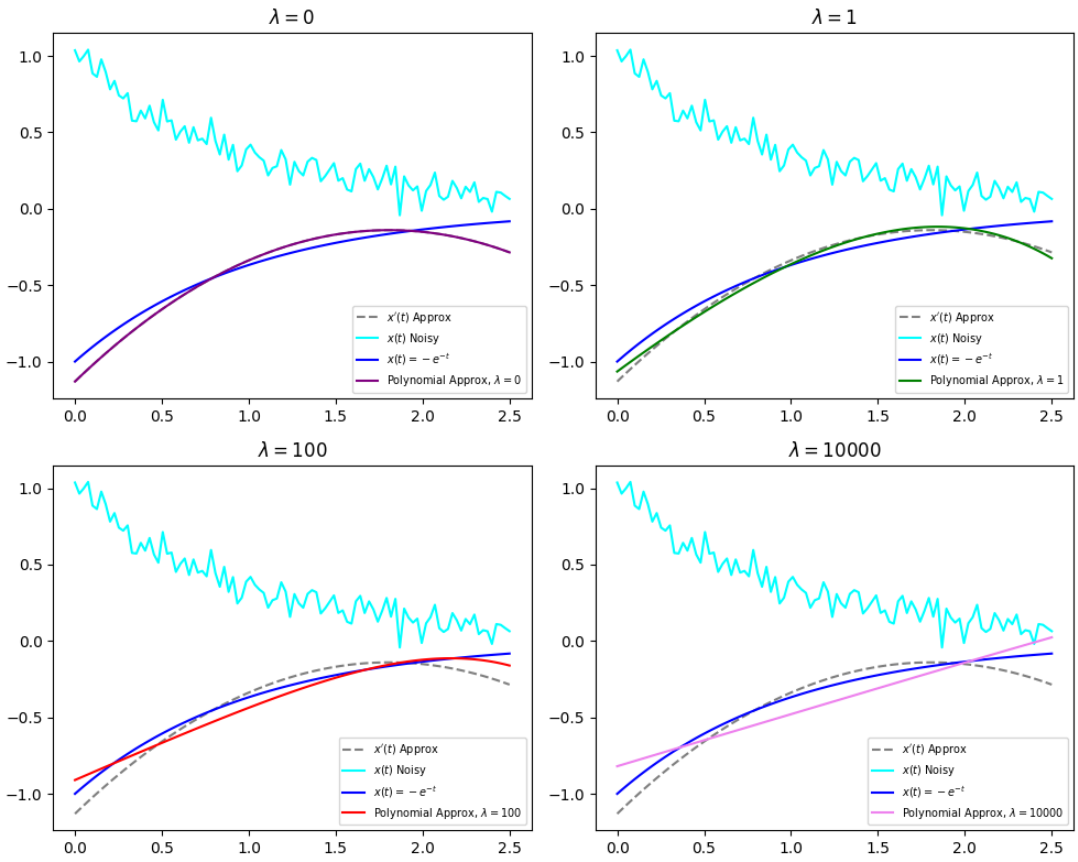

Figure 1: The Approximation for Different Values of λ

To find **D** I took the derivative of $f(x) = a_0 + a_1 t + a_2 t^2 + \cdots + a_n t^n$ twice with respect to t. Taking advantage of the pattern $[0, 0, 1\times 2a_2, 2\times 3a_3, \cdots, n(n-1)a_n]$ being the coefficients of $0, 0, 1, x, \cdots, x^{n-2}$, I have constructed the matrix for the regularisation parameter.

1.4 Analysis

Thus, finding all the matrices we need and computing x_{RLS} in python, we can plot it for different values of λ and see how it affects the fit of the polynomial. As can be seen in Figure 1, the polynomial approximation gets closer and closer to the true value of $x'(t) = -e^{-t}$ for larger λ to a certain point, up to and including $\lambda = 100$ after which it becomes a worse approximation.

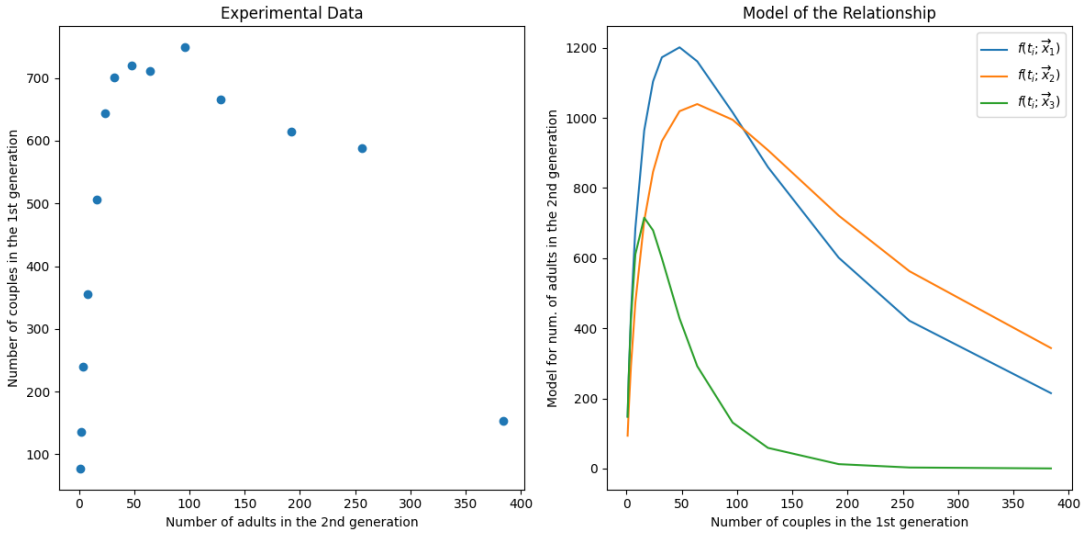

Figure 2: Weevil population density effect on their fertility

2 Gauss-Newton Method

2.1 Part a)

Figure 2

2.2 Part b)

From the lecture 5 notes

$$J(\mathbf{x}^k) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}^k) & \cdots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}^k) \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1}(\mathbf{x}^k) & \cdots & \frac{\partial f_m}{\partial x_n}(\mathbf{x}^k) \end{bmatrix} \tag{5}$$

Plugging in variables in our case

$$J(\mathbf{x}^k) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(t_1; \mathbf{x}^k) & \cdots & \frac{\partial f_1}{\partial x_n}(t_1; \mathbf{x}^k) \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1}(t_m; \mathbf{x}^k) & \cdots & \frac{\partial f_m}{\partial x_n}(t_m; \mathbf{x}^k) \end{bmatrix} \tag{6}$$

2.3 Part c)

Output for x_1, x_2 and x_3 respectively:
array([97.01908765, 0.30193299, 0.46562079]), With 14 iterations
array([97.01908765, 0.30193299, 0.46562079]), with 12 iterations

array([nan, nan, nan]), with 6 iterations. It produced a convergent sequence Code is in the appendix.

2.4 Part d)

Keeping the same stopping criteria and starting at $\mathbf{x}_1$. When taking constant step sizes of 0.8, 0.6 and 0.4 respectively, the output, (Damped Gauss Newton, number of iterations), is:
(array([97.01908765, 0.30193299, 0.46562079]), 21)
(array([97.01908765, 0.30193299, 0.46562079]), 36)
(array([97.01908765, 0.30193299, 0.46562079]), 63)
Similarly, when starting at $\mathbf{x}_2$
(array([97.01908765, 0.30193299, 0.46562079]), 21)
(array([97.01908765, 0.30193299, 0.46562079]), 35)
(array([97.01908765, 0.30193299, 0.46562079]), 62)
Code is in the appendix. The output arrays have stayed the same, the only thing that changed was the number of iterations. Another interesting result is that the number of iterations have stayed similar across $\mathbf{x}_1$ and $\mathbf{x}_2$.

2.5 Part e)

There is an optimal starting point and optimal number of couples in the first generation that produce a maximum number of adults in the second generation. Judging from the Figure 2 it is $\mathbf{x}_1$ and around 75 couples. Above and below that the graph steeply declines.

3 Appendix

3.1 Problem 1 Code

```
import numpy as np
import matplotlib.pyplot as plt
from jedi.inference.imports import import_module_by_names
from scipy.interpolate import UnivariateSpline


# Load the data from the CSV file
data = np.loadtxt('Optimization/t_data_x_noisy.csv', delimiter=',',
    skiprows=1)

# Split the columns into time and noisy data
t_data = data[:, 0]
x_noisy = data[:, 1]

# Fit the noisy data using a spline to approximate x'(t)
spline_fit = UnivariateSpline(t_data, x_noisy, s=0.5)  # Smoothing
    spline
spline_derivative = spline_fit.derivative()  # Derivative of the
    spline
```

```
17
18 # Get the derivative values from the spline approximation
19 x_prime_approx = spline_derivative(t_data) # This is the
       approximation of x'(t)
20
21 x_true = np.exp(-t_data)
22
23
24 # Polynomial fitting
25 def poly_fitting(n, l):
26     # A matrix whose rows are u_i = [1, u_i, u_i^2, ..., u_i^n]
27     U = np.vander(t_data, n+1, increasing=True)  # Polynomial
       Fitting Matrix
28
29     # Finding the Regularization Matrix
30     D = np.zeros((n-1, n+1))
31     for i in range(2, n+1):
32         D[i-2][i] = i * (i-1)
33
34     x_0 = np.matmul(U.transpose(), U) + l * np.matmul(D.transpose()
       , D)
35     x_1 = np.linalg.inv(x_0)
36     x_RLS = np.matmul(np.matmul(x_1, U.transpose()), x_prime_approx
       )
37     return U, x_RLS
38
39
40 def f(n, l):
41     U, x_RLS = poly_fitting(n, l)
42     return np.matmul(U, x_RLS)
43
44
45 # Plotting
46 fig = plt.figure(figsize=(10, 8))
47 # Grid of figures
48 ax1 = fig.add_subplot(221)
49 ax2 = fig.add_subplot(222)
50 ax3 = fig.add_subplot(223)
51 ax4 = fig.add_subplot(224)
52 # Adding plots
53 ax1.plot(t_data, x_prime_approx, ls='--', c='gray', label="$x'(t)$
       Approx")
54 ax2.plot(t_data, x_prime_approx, ls='--', c='gray', label="$x'(t)$
       Approx")
55 ax3.plot(t_data, x_prime_approx, ls='--', c='gray', label="$x'(t)$
       Approx")
56 ax4.plot(t_data, x_prime_approx, ls='--', c='gray', label="$x'(t)$
       Approx")
57
58 # Added the noisy and derivative of true data to each axes for
       comparison
59 ax1.plot(t_data, x_noisy, c='cyan', label='$x(t)$ Noisy')
60 ax1.plot(t_data, -x_true, c='blue', label='$x(t) = -e^{-t}$')
61
62 ax2.plot(t_data,x_noisy, c='cyan', label='$x(t)$ Noisy')
63 ax2.plot(t_data, -x_true, c='blue', label='$x(t) = -e^{-t}$')
64
```

```
ax3.plot(t_data,x_noisy, c='cyan', label='$x(t)$ Noisy')
ax3.plot(t_data, -x_true, c='blue', label='$x(t) = -e^{-t}$')

ax4.plot(t_data,x_noisy, c='cyan', label='$x(t)$ Noisy')
ax4.plot(t_data, -x_true, c='blue', label='$x(t) = -e^{-t}$')

# Plotting the regularised square approximations for different
    lambda
ax1.plot(t_data, f(5, 0), c='purple', label='Polynomial Approx, $\\
    lambda = 0$')
ax2.plot(t_data, f(5, 1), c='green', label='Polynomial Approx, $\\
    lambda = 1$')
ax3.plot(t_data, f(5, 100), c='red', label='Polynomial Approx, $\\
    lambda = 100$')
ax4.plot(t_data, f(5, 10000), c='violet', label='Polynomial Approx,
     $\\lambda = 10000$')

# Making the axes prettier
ax1.set_title("$\\lambda = 0$")
ax2.set_title("$\\lambda = 1$")
ax3.set_title("$\\lambda = 100$")
ax4.set_title("$\\lambda = 10000$")

ax1.legend(fontsize="7", loc ="lower right")
ax2.legend(fontsize="7", loc ="lower right")
ax3.legend(fontsize="7", loc ="lower right")
ax4.legend(fontsize="7", loc ="lower right")
fig.tight_layout()
plt.show()
```

3.2 Problem 2 Code

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.ma.core import zeros

# Load data from csv file
data = np.loadtxt('Optimization\i_t_i_Y_i.csv', delimiter=',',
                  skiprows=1)

# Split the cols into i, number of couples in the first gen, number
    of adults in the 2nd gen
i = data[:, 0]
t = data[:, 1]
Y = data[:, 2]

x1 = [200, 0.3, 0.5]
x2 = [120, 0.25, 0.5]
x3 = [180, 0.2, 0.7]


def f(t_i, x):
    x_1, x_2, x_3 = x[0], x[1], x[2]
    return x_1 * t_i * np.exp(-x_2 * t ** x_3)


f1 = f(t, x1)
```

```
f2 = f(t, x2)
f3 = f(t, x3)


# Defining the error
def r(x):
    return f(t, x) - Y


# Building the Jacobian
def Jacobian(x):
    # Grad of r
    r_x1 = t * np.exp(-x[1] * t ** x[2])  # w.r.t x1
    r_x2 = -(t ** x[2]) * f(t, x)  # w.r.t x2
    r_x3 = -x[1] * (t ** x[2]) * np.log(t) * f(t, x)  # w.r.t x3
    return np.array([r_x1, r_x2, r_x3]).transpose()


def grad_g(x):
    return 2 * np.matmul(Jacobian(x).transpose(),  r(x))

def GaussNewton(x):
    xk1 = x
    iter = 0
    while np.linalg.norm(grad_g(xk1)) > 1e-6:
        J = Jacobian(xk1)
        xk1 = xk1 - 0.5 * np.matmul(np.linalg.inv(np.matmul(J.T, J)
    ), grad_g(xk1))
        iter += 1
    return xk1, iter


def Damped_GaussNewton(x, t):
    xk1 = x
    iter = 0
    while np.linalg.norm(grad_g(xk1)) > 1e-6:
        J = Jacobian(xk1)
        a = - np.linalg.inv(np.matmul(J.T, J))
        d = np.matmul(a, np.matmul(J.T, r(xk1)))
        xk1 = xk1 + t * d
        iter += 1
    return xk1, iter

print(Damped_GaussNewton(x2, .8))
print(Damped_GaussNewton(x2, .6))
print(Damped_GaussNewton(x2, .4))

# Creating the figure
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
ax = axes[0]  # First axes
ax1 = axes[1]  # Second axes

# Scatter Plot
ax.scatter(t, Y)
ax.set_xlabel('Number of adults in the 2nd generation')
ax.set_ylabel('Number of couples in the 1st generation')
ax.set_title('Experimental Data')
```

```
81
82 # Plot of the model
83 ax1.plot(t, f1, label='$f(t_i;\\overrightarrow{x}_1)$')  # \\LaTeX
       is awesome, soo prettyy
84 ax1.plot(t, f2, label='$f(t_i;\\overrightarrow{x}_2)$')
85 ax1.plot(t, f3, label='$f(t_i;\\overrightarrow{x}_3)$')
86 ax1.set_title('Model of the Relationship')
87 ax1.set_xlabel('Number of couples in the 1st generation')
88 ax1.set_ylabel('Model for num. of adults in the 2nd generation')
89
90 # Making the plot prettier
91 plt.legend()
92 plt.tight_layout()
93 plt.show()
```

3.3 References

Lecture notes from class
"Learning Scientific Programming with Python" - Christian Hill, ISBN 978-1-108-74591-8