Least Squares Curve Fitting

Saran Ahluwalia and Mountain Chan

October 9, 2020

Linear Least Squares

Theoretical Background

For the problem of fitting a linear least squares model to an arbitrary set of datum, $\{(x_1, y_1), ..., (x_n, y_n)\}$ that we want to fit to an arbitrary function of the form

$$\sum_i c_i f_i(x) \tag{1}$$

An ubiquitous variant for curve fitting is to fit the data to a polynomial of order m. That is,

$$P_m(x) = \sum_{k=0}^{m} c_k x^k \tag{2}$$

In the following exposition, our main objective will be to fit the the error function, $\text{erf}\,(x)$:

$$\text{erf}\,(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{3}$$

This exercise illustrates the methods for fitting a well-explored function, rather than to learn and to subsequently generalize a fitted model to an aggregated dataset.

Polynomial Fitting Experiments

In this section we cover the methods for fitting a full polynomial to the $\text{erf}\,(x)$. Briefly, the polynomial in question will possess powers for every natural number up to $n - 1$.

Plotted below are the errors of polynomial fits given by $\log |\text{erf}\,(x) - P_m(x)|$ for a polynomial of order m. Plotted in this figure are polynomials with $m = 1, ..., 5$.

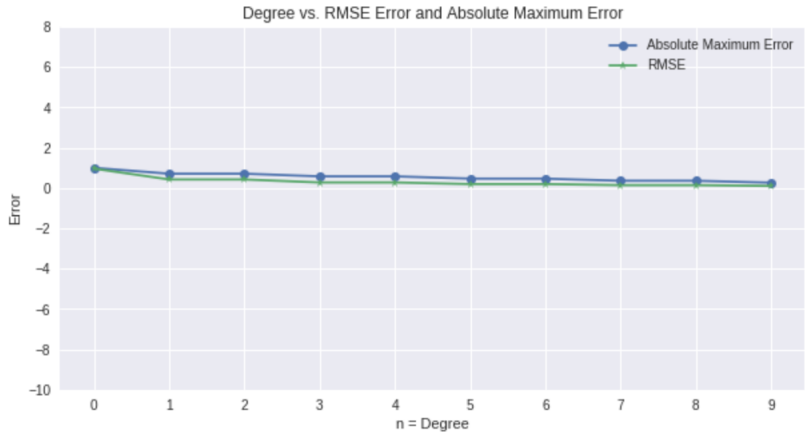

Figure 1: Full Polynomial Fit Error Plot

Degree	Absolute Error	RMSE
0	0.936952448494691	0.9611880730344562
1	0.35195967704345704	0.4243971642561228
2	0.2169221826275908	0.42439716425612284
3	0.14846721277653022	0.27588551185182014
4	0.10938021110524809	0.27588551185182014
5	0.07371128045209922	0.19567704352744125

Table 1: Full Polynomial Fit Errors Values

From the aforementioned data observed, fig. 1, the error of each of the fit decreases as the order of the polynomial increases. This is expected as higher-order polynomials have more inflection points - hence more stability - over the image of the function, before there is an asymptotic explosion.

Note that the errors have approximate symmetry about $x = 0.5$. In addition, the polynomials seem to be grouped closely into pairs of even and odd functions with a common error at $x = 0.5$. Below is a table that tabulates the root mean squared error (RMSE) and the maximum absolute error (MAE) for each degree of polynomial.

The data collected in table 1, supports our hypothesis that the error decreased as polynomial order increased. We have included a graphical illustration of the error in fig. 1.

From this figure, we noted that the RMSE and MAE are similar. We also noted, (although we did not include the that for $P_{10}(x)$, the RMSE and maximum errors are approximately $e^{-36} \approx 10^{-16}$. This may be due to the representation error due to how Python represents floats to IEEE-754 "double precision" standards. 754 doubles contain 53 bits of precision. Hence, during the computation of the input the computer strives to convert 0.1 to the closest fraction it can of the form $J/2^N$ where J is an integer containing exactly 53 bits and N is the number of significant digits.

Odd Polynomial Fitting Experiment

Next, in order to improve the fit of the polynomial we exploited the fact that the $\text{erf}\,(x)$ is an odd function. Hence, we decided to use odd polynomials in order to better approximate the $\text{erf}\,(x)$. Let the odd polynomial be represented as:

$$O_j(x) = \sum_{k=1}^{j} c_k x^k \tag{4}$$

Where $k = 2n + 1$ for $n \in \mathbb{N}$. In this experiment, we chose $j = 1, ..., 5$ - results in polynomial degrees of 1 to 9.

Plotted below are the absolute errors of the odd polynomials.

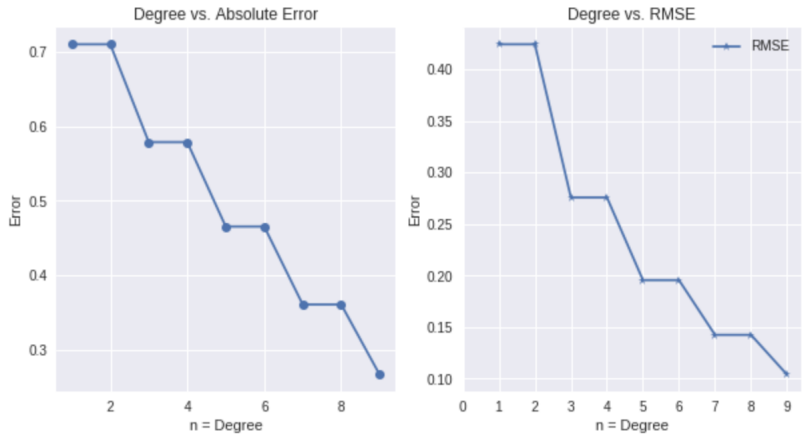

Figure 2: Odd Polynomial Fit Error Plot

Degree	Absoulte Error	RMSE
1	0.7104739391052999	0.4233780400130702
3	0.5789483075561241	0.2768012060313645
5	0.46551661953661194	0.19821158545191792

Table 2: Odd Polynomial Fit Errors Values

We noted that the magnitude of the errors for the odd function were smaller than the polynomial. This is due to the fact that the pre-image of the odd function was in the interval of $x \in [-10, 10]$. Had we chosen a purely non-negative pre-image, the results may be different.

In table 2, the values of the RMSE and the MAE are listed for the odd polynomial. An intriguing trend elucidated from the corresponding polynomial of order $\frac{i+1}{2}$ was that errors were more volatile at lower degrees and more stable at higher degrees. This trend was interpreted to imply that the previous polynomial achieved numerical stability around a point. Conversely, with the odd polynomial the number of terms is halved. Hence, there will be less points that are interpolated as the function alternates between increasing and decreasing and vice versa. However, if we observe the error, we can see that although the absolute error are clearly different, the RMSE are consistent. We speculate that this may be due to the fact that the even degree for the full polynomial have coefficients that have an order of 10^{-11} to 10^{-16} whereas the coefficient for the odd degree have coefficients that have an order of 10^{-1} degree. The vast difference in the order of magnitude contributed little to no effects to the polynomial. Hence we have a RMSE that is very similar to each other.

Observe that even though the RMSE are closely the same, there are a huge differences in absolute error between full polynomial and odd polynomial. Thus even though the fitting are similar, absolute error tells us that full polynomial fitting is a much better fitting than the odd polynomial fitting but that is not the case when we plotted the fitting over the expected values. Hence we think that absolute error might not be a idea method for measuring error in this case, and RMSE might be a better fit for measuring and comparing error.

Exponential Function Fitting Experiment

Inspection of the horizontal asymptote for the error function motivated the exploration of another means to improve the fit to the error function. The $\lim_{x\to\pm\infty} \operatorname{erf}(x) = \pm 1$. This poses a conundrum when applying the previous models that as it is necessary for a polynomial of order greater than zero to have the property $\left|\lim_{x\to\pm\infty} P_m(x)\right| = \infty$.

While the exact signs can vary based on the order of polynomial and the signs of its terms, a polynomial will most certainly never have a horizontal asymptote as $x \to \pm\infty$. To fit this property of the error function, we should choose a model that has a horizontal asymptote.

One such function that possesses such a horizontal asumptope is the exponential decay function. The exponential decay function has a horizontal asymptote at 0. At this juncture, it should be noted that this asymptote will vary depending on whether or not a constant is multiplied to the function or by simply adding as constant as another term in the model.

The problem arises that the rest of the exponential does not have the behavior that we observe in the error function. We can absole this issue by using the product of the exponential decay and another function, which can also have a horizontal asymptote as the exponential term usually has a greater affect than other terms. The model that was chosen for this report is the following

$$c_1 + e^{-t^2}\left(c_2 + c_3 z + c_4 z^2 + c_5 z^3\right) \tag{5}$$

where $z = (1+t)^{-1}$.

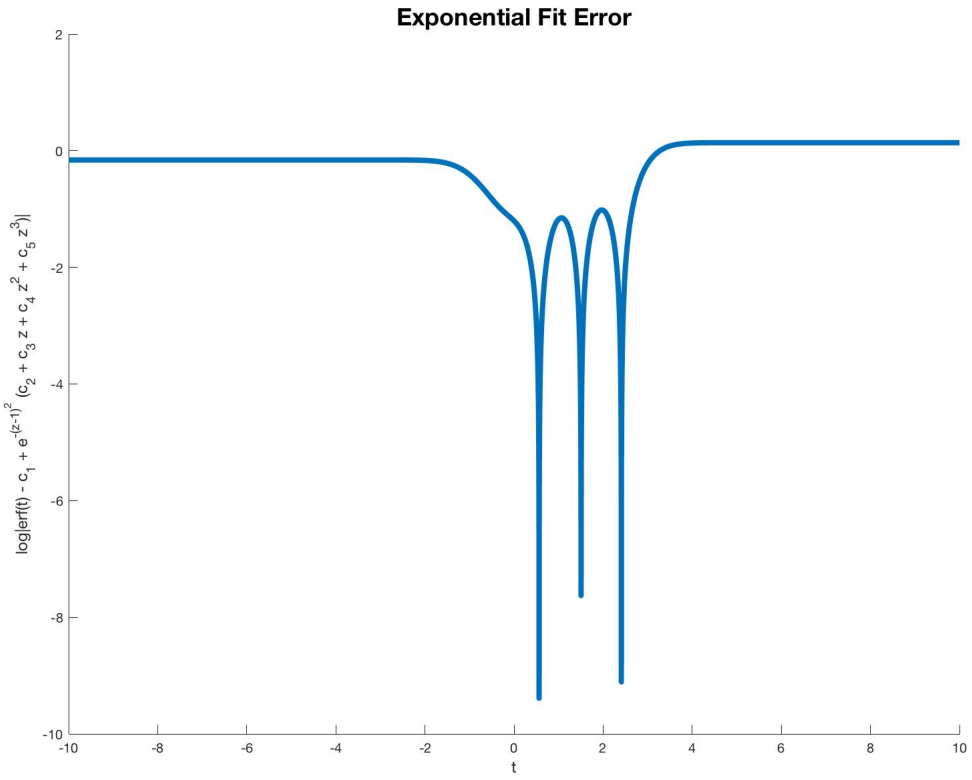

Figure 3: Exponential Fit Error Plot

Plotted in fig. 3 is the logarithm of the MAE of the exponential fit to the error function. This is due to the asymptotic nature of the error function and the exponential. That is, as x increases, the error will decrease, with the anticipation that the error converges to zero.

Additional improvements may be made by fitting a nonlinear model, resulting in parameters in places such as exponents.

Python Code

0.0.1 Least Square Method

```
# -*- coding: utf-8 -*-
"""Least Squre Regression and Horner Method.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1GGBkGZoJzEEJ7XnyLiHdVncjOTxcOjTv
"""

import sys
import time
import datetime
import os
import math
import numpy as np
import matplotlib as mpl
```

```
from matplotlib import pyplot as plt
from matplotlib import rc
from matplotlib.pyplot import legend

def polyLeastSquares(m, X):
    '''
    polynomial least squares curve fitting
    m = degree of the fitting polynomial

    X = array which contains the data points
    and is of shape (nDatapoints, 2)

    returns the fitted weights vector, which is of shape
    (m + 1,)

    Usage:
    m = 9
    w = polyLeastSquares(m, Xt) # Xt = training data
    # returns the weight vector w
    '''
    nDatapoints = X.shape[0]
    assert X.shape[1] == 2, "Error: Shape assertion failed."

    # fill the Vandermonde matrix V
    V = np.ones((nDatapoints, m + 1))

    # column vector
    tmp = np.ones((nDatapoints,))

    for i in range(m):
        tmp = np.multiply(tmp, X[:, 0])
        V[:, i + 1] = tmp

    # fill the right hand side
    b = np.ones((nDatapoints, 1))
    b[:, 0] = X[:, 1]

    A = np.matmul(V.transpose(), V)
    b = np.matmul(V.transpose(), b)

    # solve linear system A * w = b for the weights vector w
    w = np.linalg.solve(A, b)
    w = w.reshape((m + 1,))

    return w

def poly_horner(x, coeff):

    result = coeff[-1]
    for i in range(-2, -len(coeff) - 1, -1):
        result = result * x + coeff[i]
    return result

if __name__ == '__main__':

    ######################################################################
```

```
    # randomize training data

    Xt = np.random.uniform(-10,10,size=(21, 2))

    assert Xt.shape == (21, 2), "Error: Shape assertion failed."

    print("Training data shape =", Xt.shape)

    seedValue = 523456789

    ######################################################################

    # polynomial fitting orders
    jobs = np.arange(0, 5).reshape(-1)

    p1 = None
    for m in jobs:

        # polynomial least squares fitting
        w = polyLeastSquares(m, Xt)
        print("fitted weights =", w)
        # use poly d to validate that Horner's method works
        # p1 = np.poly1d(w)
        # print the polynomial coefficients to compare with regression
        # print('Rebuilt polynomial from coefficients = {}'.format(p1))

        # create fitted model

        nModelPoints = 800
        xVals = np.linspace(-10.0, 10.0, nModelPoints)
        yVals = np.array([poly_horner(x, w) for x in xVals])
        Xm = np.zeros((nModelPoints, 2))
        Xm[:, 0] = xVals
        Xm[:, 1] = yVals
```

0.0.2 Part a

```
# -*- coding: utf-8 -*-
"""Full Polynomial Fitting.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1FkBcNp7JTzrLj0lGBgc3BN-jkFGXxRdu
"""

import sys
import time
import datetime
import os
import math
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
from matplotlib import rc
from matplotlib.pyplot import legend
```

```
plt.rcParams['figure.figsize'] = [20, 15]


# From Handbook of Mathematical Functions, formula 7.1.26.
def erf(x):
    # save the sign of x
    sign = 1 if x >= 0 else -1
    x = abs(x)

    # constants
    a1 =  0.254829592
    a2 = -0.284496736
    a3 =  1.421413741
    a4 = -1.453152027
    a5 =  1.061405429
    p  =  0.3275911

    # A&S formula 7.1.26
    t = 1.0/(1.0 + p*x)
    y = 1.0 - (((((a5*t + a4)*t) + a3)*t + a2)*t + a1)*t*math.exp(-x*x)
    return sign*y # erf(-x) = -erf(x)

def polynomial_fitted_y_values(x, w):
    x = [x ** j for j in np.arange(0, w.size)]
    wTx = w[:, np.newaxis].T.dot(x).T
    return wTx[:, 0]

def least_squares(M, x, t):
    x = np.array([x ** j for j in np.arange(0, M + 1)])
    return np.linalg.solve(x.dot(x.T), x.dot(t))

def rmse(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())

def mean_absolute_error(predictions, targets):
  return np.absolute(predictions - targets).mean()

def absolute_error(predictions, targets):
    difference = predictions - targets
        return np.absolute(np.amax(difference))

def main():

    x = np.linspace(-10, 10, 21)

    t =  np.array([erf(xVal) for xVal in x])
    xs = np.linspace(-10, 10, 21)

    t_ideal = np.array([erf(xVal) for xVal in xs])

    l2_diff = []
    rms_diff = []
    mae_diff = []
    abs_errors = []
```

```
    M = np.arange(0, 10).reshape(-1)
    fig_row = np.ceil(M.shape[0] / 2)
    fig_col = np.ceil(len(M) / fig_row)
    fig = plt.figure()
    for i, m in enumerate(M):
        w = least_squares(m, x, t)
        fig.add_subplot(fig_row, fig_col, i + 1)
        plt.plot(x, t, 'b.', label='Training data')
        plt.plot(xs, t_ideal, 'g-', label='f(x) = erf(x)')
        # Compute Various errors here
        l2_norm = np.linalg.norm((t_ideal - polynomial_fitted_y_values(xs, w)), 2)
        rms_error = rmse(polynomial_fitted_y_values(xs, w), t_ideal)
        mae = mean_absolute_error(polynomial_fitted_y_values(xs, w), t_ideal)
        abs_error = absolute_error(polynomial_fitted_y_values(xs, w), t_ideal)

        rms_diff.append(rms_error)
        mae_diff.append(mae)
        abs_errors.append(abs_error)

        plt.plot(xs, polynomial_fitted_y_values(xs, w), 'y-', label='Polynomial Curve fitting')
        plt.legend()
        plt.title('M = {0}'.format(m))
        plt.xlim(-10, 10)
        plt.ylim(-6.5, 6.5)

    fig, ax = plt.subplots(figsize=(10, 5))
    ax.plot(np.arange(0, 10), abs_errors, marker='o', label='Absolute Maximum Error')
    ax.plot(np.arange(0, 10), rms_diff, marker="*", label='RMSE')
    ax.set_title('Degree vs. RMS Error and Absolute Maximum Error')
    ax.set_xlabel('n = Degree')
    ax.set_ylabel('Error')
    plt.legend()

    plt.yticks(np.arange(-10,10, 2))
    plt.xticks(np.arange(0,10,1))
    plt.show()

if __name__ == '__main__':
  main()
```

0.0.3 Part b

```
# -*- coding: utf-8 -*-
"""Odd Polynomial Fitting.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/18XhvH9qgWAfOKfFO69W1Kdm29B2kOKi2
"""

import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [20, 15]

def polynomial_fitted_y_values(x, w):
```

```
    x = [x ** j for j in np.arange(0, w.size)]
    wTx = w[:, np.newaxis].T.dot(x).T
    return wTx[:, 0]


def least_squares(M, x, t):

    x_domain = np.array([x ** j for j in np.arange(0, M + 1)])
    xOdd = x_domain.copy()
    #xOdd = xOdd.tolist()
    index = []
    for i in range(x_domain.shape[0]):
      if i % 2 == 0:
        index.append(i)
    if M == 0:
        xOdd[i] = np.nan
    else:
      xOdd = np.delete(x_domain, index, 0)

    return np.linalg.solve(xOdd.dot(xOdd.T), xOdd.dot(t))

def main():

    x = np.linspace(-10, 10, 21)

    t =  np.array([erf(xVal) for xVal in x])
    xs = np.linspace(-10, 10, 500)
    t_ideal = np.array([erf(xVal) for xVal in xs])
    l2_diff = []
    rms_diff = []
    abs_errors = []

    M = np.arange(1, 10).reshape(-1)
    fig_row = np.ceil(M.shape[0] / 2)
    fig_col = np.ceil(len(M) / fig_row)
    fig = plt.figure()
    for i, m in enumerate(M):
        index = []
        w = least_squares(m, x, t)
        # index for all the even coefficient
        index = np.arange(0, len(w))
        # padding w with zero's in the even coefficient position
        w = np.insert(w, index, 0)
        rms_error = rmse(polynomial_fitted_y_values(xs, w), t_ideal)
        abs_error = absolute_error(polynomial_fitted_y_values(xs, w), t_ideal)

        # l2_norm = np.linalg.norm((t_ideal - polynomial_fitted_y_values(xs, w)), 2)

        # l2_diff.append(l2_norm)
        rms_diff.append(rms_error)
        abs_errors.append(abs_error)

        fig.add_subplot(fig_row, fig_col, i + 1)
        plt.plot(x, t, 'b.', label='Training data', markersize = 12)
        plt.plot(xs, t_ideal, 'g-', label='f(x) = erf(x)')
        plt.plot(xs, polynomial_fitted_y_values(xs, w), 'r-', label='Polynomial Curve fitting')
```

```
        plt.legend()
        plt.title('M = {0}'.format(m))
        plt.xlim(-10, 10)
        plt.ylim(-6.5, 6.5)

    fig, (ax1,ax2) = plt.subplots(1,2,figsize=(10, 5))
    ax1.plot(np.arange(1, 10), abs_errors, marker='o', label='Absolute Error')
    ax2.plot(np.arange(1, 10), rms_diff, marker="*", label='RMSE')
    ax1.set_title('Degree vs. Absolute Error')
    ax2.set_title('Degree vs. RMSE')
    ax1.set_xlabel('n = Degree')
    ax2.set_xlabel('n = Degree')
    ax1.set_ylabel('Error')
    ax2.set_ylabel('Error')
    plt.legend()

    plt.xticks(np.arange(0,10,1))
    plt.show()

if __name__ == '__main__':
  main()
```

0.0.4 Part c

Listing 1: Exponential Model Code Sample

```
function [model_rmse, max_absolute_error, x, errors_x_domain] =
    erf_nonlinear()
%erf_fitting using linear least squares to fit error function

% Points for evaluation
x = -10:0.1:10;

% Evaluate error function at points for fitting
y_true = erf(x);

% Evaluate error function more generally for error test points
errors_x_domain = -10:0.001:10;
true_erf_values = erf(errors_x_domain);


% Create exponential model
exponential_model = @(z) 1 + exp(-(z-1).^2) * 1  + exp(-(z-1).^2) .* z.^2 +
    ...
    exp(-(z-1).^2) .* z.^2 + exp(-(z-1).^2) .* z.^3;

exp_figure = figure;
exp_axes = axes;
hold(exp_axes);

exp_fit = linlsqfit(x, y_true, exponential_model);

% Evaluate model at error testing points
coefficients = splitfunction(exponential_model);
value_evaluations = 0;
for term=1:5
```

```
29      value_evaluations = value_evaluations + ...
30          exp_fit(term) * coefficients{term}(errors_x_domain);
31  end
32
33  model_evaluation = value_evaluations;
34
35  % Compute Error
36  exponential_model_error = model_evaluation - true_erf_values;
37
38  model_rmse = rms(exponential_model_error);
39  max_absolute_error = max(abs(exponential_model_error));
40
41  plot(exp_axes, errors_x_domain, log(abs(exponential_model_error)), '
        LineWidth', 4);
42
43  func_title = title('Exponential Fit Error');
44  set(func_title, 'FontSize', 18);
45
46  x_label = xlabel('t');
47  y_label_ = ylabel(['log|erf(t) - c_1 + e^{-(z-1)^2} ' ...
48      '(c_2 + c_3 z + c_4 z^2 + c_5 z^3)|']);
49  set(x_label, 'FontSize', 12);
50  set(x_label, 'FontSize', 12);
51
52  % Present axes for visual
53  hold(exp_axes);
54
55
56  end
```