Linear Modeling - Least Square Approach
TF4063

Fadjar Fathurrahman

The material in this note is based on this book [1]. We only show only some portion of the code to illustrate the idea described in the note. You can find the original MATLAB codes accompanying the book at:
`https://github.com/sdrogers/fcmlcode`

1 *Linear model and its loss function*

Given pair of data (x, t) where x are inputs dan t are targets, a linear model with parameter (w_0, w_1) can be written as:

$$t = f(x; w_0, w_1) = w_0 + w_1 x \tag{1}$$

We are now left with the task of choosing the best parameters (w_0, w_1) fir this model. We need to quantify how good the model is. One metric that we can use to quantify this is the squared difference (or error) between target and model prediction. For n-th data we can write

$$\mathcal{L}_n \equiv (t_n - f(x_n; w_0, w_1))^2 \tag{2}$$

By averaging contributions from all data:

$$\mathcal{L} = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n = \frac{1}{N} \sum_{n=1}^{N} (t_n - f(x_n; w_0, w_1))^2 \tag{3}$$

We will call this quantity as loss function and we want to this quantity to be as small as possible. Finding model parameters by minimizing loss functions such as in Eq. (3) is known as least square approach to linear regression.

2 *Minimizing loss function*

We can find the parameters (w_0, w_1) by using minimization procedures:

$$\arg\min_{w_0, w_1} \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n \tag{4}$$

For our particular case of Eq. (3), we can found this analytically, i.e. calculating the first derivatives of $\mathcal{L}$ with respect to w_0 and w_1, equating them to zero, and solve the resulting equations for w_0 and w_1. For

more general cases, we can use various numerical optimization procedures such as gradient descent methods.
We begin by writing our loss function as:

$$\begin{aligned}\mathcal{L} &= \frac{1}{N}\sum_{n=1}^{N}\left(t_n - (w_0 + w_1 x_n)\right)^2 \\ &= \frac{1}{N}\sum_{n=1}^{N}\left(w_1^2 x_n^2 + 2w_1 x_n(w_0 - t_n) + w_0^2 - 2w_0 t_n + t_n^2\right)\end{aligned}$$

Now we find the first derivatives of $\mathcal{L}$ with respect to w_0, w_1 and equating them to zero.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_1} &= 2w_1\frac{1}{N}\left(\sum_{n=1}^{N} x_n^2\right) + \frac{2}{N}\left(\sum_{n=1}^{N} x_n(w_0 - t_n)\right) = 0 \\ \frac{\partial \mathcal{L}}{\partial w_0} &= 2w_0 + 2w_1\frac{1}{N}\left(\sum_{n=1}^{N} x_n\right) - \frac{2}{N}\left(\sum_{n=1}^{N} t_n\right) = 0\end{aligned}$$

We obtain

$$\begin{aligned}w_1 &= \frac{\overline{xt} - \bar{x}\bar{t}}{\overline{x^2} - \bar{x}^2} \\ w_0 &= \bar{t} - w_1\bar{x}\end{aligned} \tag{5}$$

where symbols with overline denotes their average value, for examples

$$\begin{aligned}\bar{x} &= \frac{1}{N}\sum_{n=1}^{N} x_n \\ \bar{t} &= \frac{1}{N}\sum_{n=1}^{N} t_n\end{aligned}$$

Example: olympic100m dataset

Now we want to implement least square approach to linear regression based on Eq. 2. For this purpose, we need a simple data to work on. We choose to work with `olympic100m` data which describes winning time of men's 100m sprint Olympic Games. You can choose to work with other data or synthetic data.

Year	Seconds
1896	12.00
1900	11.00
1904	11.00
...	...
2008	9.69

The data is plotted in Figure 1.

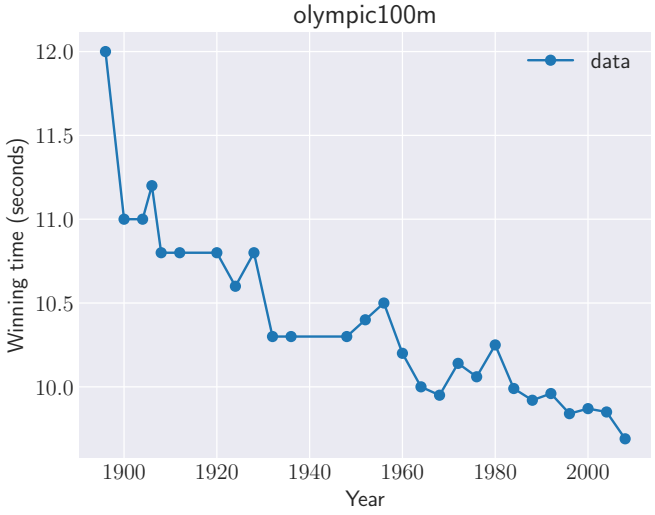


Figure 1: *xy* plot of olympic100m data

From Figure 1 we see that there is a tendency that winning time to decrease as time progress. We might want to apply linear regression to this case. The following Python program implements the equation for this simple dataset.

```
import numpy as np

import matplotlib.pyplot as plt
import matplotlib
matplotlib.style.use("seaborn-darkgrid")
plt.rcParams.update({
    "text.usetex": True,
    "font.size": 14}
)

# Load the data
# set the actual path for your case
DATA_PATH = "olympic100m.txt"
data = np.loadtxt(DATA_PATH, delimiter=",")

x = data[:,0]
t = data[:,1]

# Calculate the parameters
tbar = np.average(t)
xbar = np.average(x)
xtbar = np.average(x*t)
x2bar = np.average(x**2)

w1 = (xtbar - xbar*tbar)/(x2bar - xbar**2)
w0 = tbar - w1*xbar
```

```
print("Model parameters:")
print("w0 = %18.10f" % w0)
print("w1 = %18.10f" % w1)

t_pred = w0 + w1*x

# Plotting stuffs
# ....
```

From the previous program, we get the following result for the weights (paramaters of the model)

```
Model parameters:
w0 =      36.4164559025
w1 =      -0.0133308857
```

We can visualize the result by plotting the data and the a line described linear equation Eq. (1). The result is shown in Fig. 2.

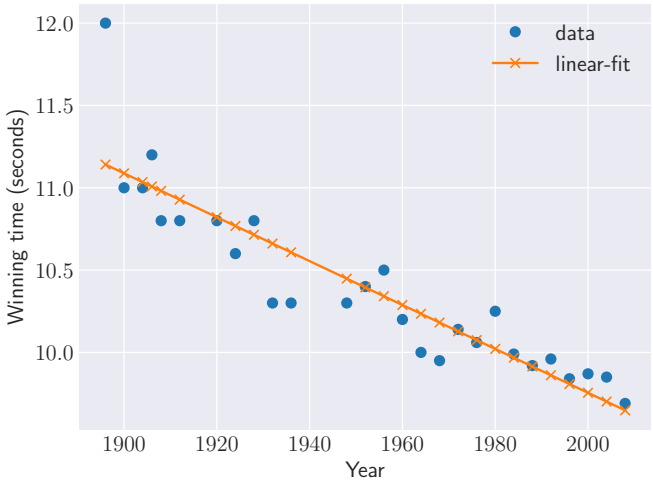


Figure 2: Result of linear regression for `olympic100m`.

3 *Using matrix and vector notation*

We will rewrite our previous problem in matrix and vector notation. This will give us more flexibility and enable us to generalize to more complex situations. We start by defining inputs and model parameters as vectors.

$$\mathbf{x}_n = \begin{bmatrix} 1 \\ x_n \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

Using this definition, we can write our previous linear model in Eq. (1) as:

$$f(x_n; w_0, w_1) = \mathbf{w}^\mathsf{T} \mathbf{x}_n \tag{6}$$

The expression for loss function, Eq. (3), becomes

$$\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}\left(t_n - \mathbf{w}^\mathsf{T}\mathbf{x}_n\right)^2 \tag{7}$$

We now arrange several input vector into a matrix:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \mathbf{x}_2^\mathsf{T} \\ \vdots \\ \mathbf{x}_N^\mathsf{T} \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix}$$

As with inputs, we now define target vectors as

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{bmatrix} \tag{8}$$

With this definition we can write the loss function as

$$\mathcal{L} = \frac{1}{N}\left(\mathbf{t} - \mathbf{X}\mathbf{w}\right)^\mathsf{T}\left(\mathbf{t} - \mathbf{X}\mathbf{w}\right) \tag{9}$$

To find the best value of $\mathbf{w}$ we can follow similar procedure that we have used in the previous part. We need to find the solution of $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 0$

$$\mathcal{L} = \frac{1}{N}\left(\mathbf{t}^\mathsf{T}\mathbf{t} + (\mathbf{X}\mathbf{w})^\mathsf{T}\mathbf{X}\mathbf{w} - \mathbf{t}\mathbf{X}\mathbf{w} - (\mathbf{X}\mathbf{w})^\mathsf{T}\mathbf{t}\right) \tag{10}$$

$$= \frac{1}{N}\left(\mathbf{w}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{w} - 2\mathbf{w}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{t} + \mathbf{t}^\mathsf{T}\mathbf{t}\right) \tag{11}$$

Equating these to zeros we have

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{2}{N}\left(\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{w} - \mathbf{X}^\mathsf{T}\mathbf{t}\right) = 0 \tag{12}$$

So we have

$$\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{w} = \mathbf{X}^\mathsf{T}\mathbf{t} \tag{13}$$

or

$$\mathbf{w} = \left(\mathbf{X}^\mathsf{T}\mathbf{X}\right)^{-1}\mathbf{X}^\mathsf{T}\mathbf{t} \tag{14}$$

The following snippet describes how Eq. (14) is implemented.

```
# Load the data
DATAPATH = "olympic100m.txt"
data = np.loadtxt(DATAPATH, delimiter=",")
```

```
Ndata = len(data) # data.shape[0]

x = data[:,0]
t = data[:,1]

# Build the input matrix
X = np.zeros((Ndata,2))
X[:,0] = 1.0
X[:,1] = data[:,0]

# Calculate the model parameters
XtX = X.T @ X
XtXinv = np.linalg.inv(XtX)
w = XtXinv @ X.T @ t

print("Model parameters:")
print("w0 = %18.10e" % w[0])
print("w1 = %18.10e" % w[1])

t_pred = X @ w

# Plot the results ...
```

The result should be the same as using Equation (2):

```
Model parameters:
w0 =   3.6416455903e+01
w1 =  -1.3330885711e-02
```

4 *Generalization to more complex models*

We can use more "complex" models than (6). For example the quadratic equation:

$$f(x; w_0, w_1, w_2) = w_0 + w_1 x + w_2 x^2 \tag{15}$$

Note that the model is still linear in parameter, so this model is also a linear model. Using matrix and vector notation, we can fit the data to this equation simply by adding one column to the matrix $\mathbf{X}$:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix} \tag{16}$$

The following snippets shows how to implement this:

```
# Build matrix X
X = np.zeros( (Ndata,3) )
X[:,0] = 1.0
X[:,1] = data[:,0]
X[:,2] = np.power( data[:,0], 2 )

t = data[:,1] # target

XtX = X.transpose() @ X
XtXinv = np.linalg.inv(XtX)
w = XtXinv @ X.transpose() @ t

print("Model parameters:")
print("w0 = %18.10e" % w[0])
print("w1 = %18.10e" % w[1])
print("w2 = %18.10e" % w[2])

t_pred = X @ w
```

This scheme also applies to higher order polynomials. The following snippet shows how to build the matrix $\mathbf{X}$, find the parameters $\mathbf{w}$ and also how to make prediction given model parameter and input.

```
def fit_polynomial(x, t, Npoly):
    Ndata = len(x)
    # Npoly is degree of the polynomial
    X = np.zeros( (Ndata,Npoly+1) )
    X[:,0] = 1
    for i in range(1,Npoly+1):
        X[:,i] = np.power( x, i )
    XtX = X.transpose() @ X
    XtXinv = np.linalg.inv(XtX)
    w = XtXinv @ X.transpose() @ t
    return X, w

Npoly = 3
X, w = fit_polynomial(x, t, Npoly)

# Define new input from first x to last x where the model
↪  will be evaluated
NptsPlot = 200
x_eval = np.linspace(x[0], x[-1], NptsPlot)
# Build X matrix for new input
X_eval = np.zeros( (NptsPlot,Npoly+1) )
X_eval[:,0] = 1.0
for i in range(1,Npoly+1):
    X_eval[:,i] = np.power( x_eval, i )

# Evaluate the model for the new input
t_eval = X_eval @ w
```

In the following figure we show the result of fitting `olympic100m` data to 3rd order polynomial.

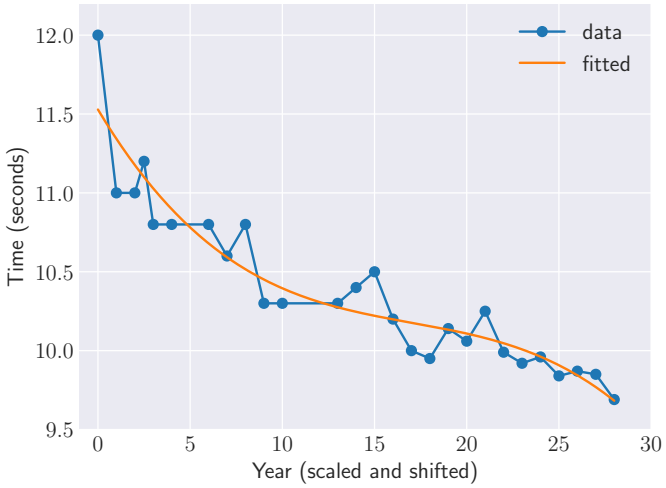


Figure 3: 3rd order polynomial fit to `olympic100m` data.

Note that we have shifted and scaled the x-axis (the input) to avoid numerical problems (round-off errors) with high order polynomial. The following snippet can be used for this purpose.

```
# Load the data
DATAPATH = "olympic100m.txt"
data = np.loadtxt(DATAPATH, delimiter=",")

t = data[:,1] # Target
# Rescale the data to avoid numerical problems with large
↪ numbers
x = data[:,0]
x = x - x[0] # shift
x = 0.25*x # scale
```

5 *Model selection*

Evaluating the value of loss according to Eq. (7) on the observation data for different polynomial order we have the following result.

```
Order:   1 Loss:    0.05031
Order:   2 Loss:    0.03796
Order:   3 Loss:    0.02961
Order:   4 Loss:    0.02706
Order:   5 Loss:    0.02350
Order:   6 Loss:    0.02202
Order:   7 Loss:    0.01970
Order:   8 Loss:    0.01698
```

We note that the loss is decreasing as we increase the polynomial order. Using this result, we might decide to choose 8-th order polynomial as the best model. However, this is not a good idea. The result of

8-th order fit is shown in Figure 4. It can be seen that the model is not

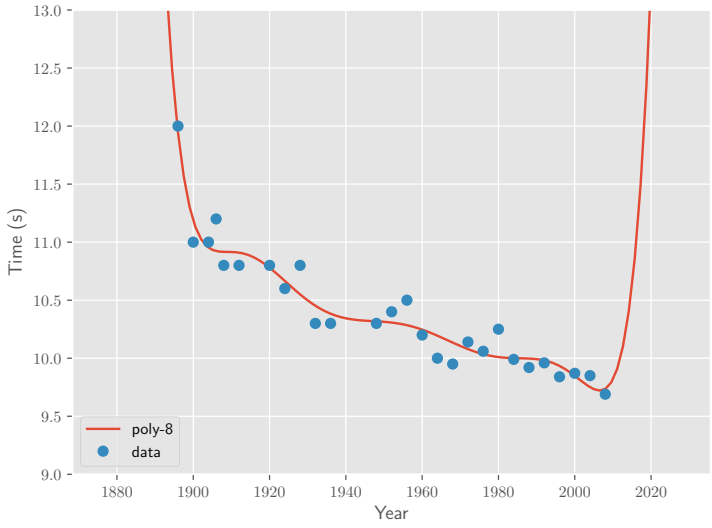


Figure 4: 8-th order polynomial fit to `olympic100m` data.

behaved very well if we try to predict t for other years other than the ones in our original data.

One solution to mitigate this problem is to split the data that we have into two subsets: the *training* and *validation* data. The model parameters $\mathbf{w}$ is calculated (or trained) using training data while the loss is calculated using validation data.

As an example, let's say that we want to use the data for year larger than 1979 as validation data and otherwiseas training data. The following snippet shows how to do this.

```
def fit_polynomial(x, t, Npoly):
    Ndata = len(x)
    # Npoly is degree of the polynomial
    X = np.zeros( (Ndata,Npoly+1) )
    X[:,0] = 1
    for i in range(1,Npoly+1):
        X[:,i] = np.power( x, i )
    XtX = X.transpose() @ X
    XtXinv = np.linalg.inv(XtX)
    w = XtXinv @ X.transpose() @ t
    return X, w

def predict_polynomial(w, x_eval):
    Npoly = w.shape[0] - 1
    Ndata_eval = x_eval.shape[0]
    # Build X matrix for new input
    X_eval = np.zeros( (Ndata_eval,Npoly+1) )
    X_eval[:,0] = 1.0
    for i in range(1,Npoly+1):
```

```
        X_eval[:,i] = np.power( x_eval, i )
    # evaluate
    t_eval = X_eval @ w
    return t_eval

# Load the data
DATAPATH = "olympic100m.txt"
data = np.loadtxt(DATAPATH, delimiter=",")

t_full = data[:,1] # Target
x_full = data[:,0]
# Data indices for validation and training data
idx_val = x_full > 1979
idx_train = x_full <= 1979
#
x_val = x_full[idx_val]
t_val = t_full[idx_val]
#
x = x_full[idx_train]
t = t_full[idx_train]

# Shift and rescale the data to avoid numerical
# problems with large numbers
x = x - x_full[0]
x = 0.25*x
# also do this for validation input
x_val = x_val - x_full[0]
x_val = 0.25*x_val

for Npoly in range(1,9):
    X, w = fit_polynomial(x, Npoly)
    t_val_pred = predict_polynomial(w, x_val)
    loss = np.sum( (t_val_pred - t_val)**2/len(t_val) )
    print("Npoly = %2d   loss = %10.5f" % (Npoly, loss))
```

The result is shown below.

```
Npoly =  1   loss =    0.10130
Npoly =  2   loss =    0.16763
Npoly =  3   loss =    1.06188
Npoly =  4   loss =    4.45706
Npoly =  5   loss =    5.51420
Npoly =  6   loss = 1533.43686
Npoly =  7   loss =   61.91356
Npoly =  8   loss = 6023.27314
```

From this result, we can say that first order polynomial is the best model.

However, note that the above result is very sensitive on the choice of validation data. This is particularly problematic if our data is small. A technique called *cross-validation* can be used to mitigate this problem. K-fold cross-validation splits the data into K equally (or as close to equal as possible) sized blocks.

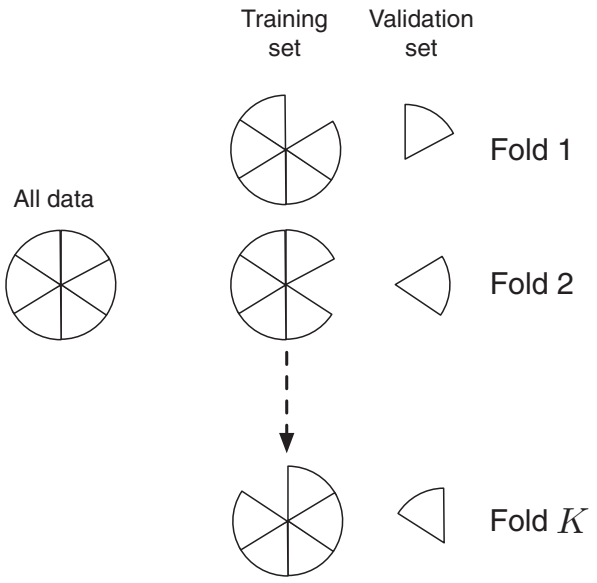


Figure 5: Dividing data into train and validation sets for cross validation.

Each block takes its turn as a validation set for a training set comprised of the other $K-1$ blocks. Averaging over the resulting k loss values gives us our final loss value.

An extreme case of K-fold cross-validation is where $K = N$, the number of observations in our dataset: each data observation is held out in turn and used to test a model trained on the other This particular form of cross-validation is also known as Leave-One-Out Cross-Validation (LOOCV).

> **Task 1**
> Implement LOOCV on `olympic100m` dataset. Which polynomial gives the minimum loss?

One drawback of illustrating model selection on a real dataset is that we don't know what the "true" model is and therefore don't know if our selection techniques are working. We can overcome this by generating a synthetic dataset.

> **Task 2**
> Create a synthetic dataset from, for example generated from 3rd order polynomial plus random noise term. Fit polynomial from first to, say 7th order, to this data. Plot order vs LOOCV loss. Which polynomial order give the minimum loss?

6 *Regularization*

We can define a measure of complexity of out linear model by

$$\sum_i w_i^2 \text{ or } \mathbf{w}^\mathsf{T}\mathbf{w} \tag{17}$$

As opposed to minimizing loss function (7), we can minimize a regularized loss function by adding penalty for overcomplexity:

$$\mathcal{L}' = \mathcal{L} + \lambda \mathbf{w}^\mathsf{T}\mathbf{w}$$

where the parameter λ control the trade off between model accuracy and model complexity. To find the best parameter, we can proceed by using similar procedure as before. The regularized loss function is written as

$$\mathcal{L}' = \frac{1}{N}\mathbf{w}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{w} - \frac{2}{N}\mathbf{w}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{t} + \frac{1}{N}\mathbf{t}^\mathsf{T}\mathbf{t} + \lambda\mathbf{w}^\mathsf{T}\mathbf{w} \tag{18}$$

First derivative of the loss function with respect to model paramater $\mathbf{w}$ is

$$\frac{\partial \mathcal{L}'}{\partial \mathbf{w}} = \frac{2}{N}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{w} - \frac{2}{N}\mathbf{X}^\mathsf{T}\mathbf{t} + 2\lambda\mathbf{w}$$

Equating this to zero:

$$(\mathbf{X}^\mathsf{T}\mathbf{X} + N\lambda\mathbf{I})\mathbf{w} = \mathbf{X}^\mathsf{T}\mathbf{t}$$

we obtain

$$\mathbf{w} = (\mathbf{X}^\mathsf{T}\mathbf{X} + N\lambda\mathbf{I})^{-1}\mathbf{X}^\mathsf{T}\mathbf{t} \tag{19}$$

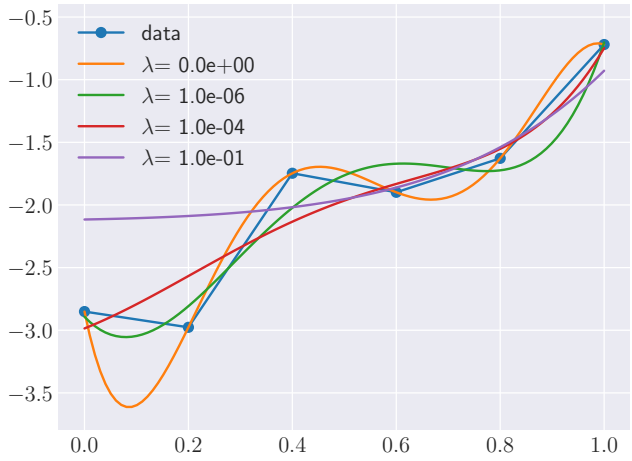
We can implement this by simple modification to our code.

```
def fit_polynomial_ridge(x, t, Npoly, λ=0.0):
    Ndata = len(x)
    # Npoly is degree of the polynomial
    X = np.zeros( (Ndata,Npoly+1) )
    X[:,0] = 1.0
    for i in range(1,Npoly+1):
        X[:,i] = np.power( x, i )
    XtX = X.transpose() @ X + Ndata*λ*np.eye(Ndata)
    XtXinv = np.linalg.inv(XtX)
    w = XtXinv @ X.transpose() @ t
    return X, w
```

As an example we will implement this for synthetic data generated from linear function plus noise term.

```
np.random.seed(1234)
Ntrain = 6 # training data
x = np.linspace(0.0, 1.0, Ntrain)
y = 2*x - 3
NoiseVar = 0.1
noise = math.sqrt(NoiseVar)*np.random.randn(x.shape[0])
t = y + noise # add some noise
```

Fitting 5-th order polynomial to this data and using various values for regularization parameter λ, we obtain the following result. Best value



for λ can be obtained by using validation data as we have done before.

The following snippet can be used to generate the figure:

```
Npoly = 5
plt.clf()
plt.plot(x, t, marker="o", label="data")
x_eval = np.linspace(x[0], x[-1], 100)
for λ in [0.0, 1e-6, 1e-4, 1e-1]:
    X, w = fit_polynomial_ridge(x, t, Npoly, λ=λ)
    t_eval = predict_polynomial(w, x_eval)
    plt.plot(x_eval, t_eval, label="$\\lambda$={:8.1e}".format(λ))
plt.xlim(-0.05, 1.05)
plt.ylim(-3.8, -0.4)
plt.grid(True)
plt.legend()
```

Task 3

Apply regularized linear regression to olympic100m data or synthetic data or other simple dataset of your choice. Determine the best model parameters (along with regularization parameter λ) and describe the procedure that you have used to determine

them.

7 *Linear regression in Scikit Learn Python package*

Scikit Learn [2] provides many algorithms for regression problems. For linear models such as discussed previously, we can use several classes from `linear_model` module:

- ordinary linear regression
- ridge linear regression

References

[1] Simon Rogers and Mark Girolami. *A First Course in Machine Learning*. CRC Press, Boca Raton, 2nd edition, 2017.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.