Machine Learning HW1

309505018 郭俊廷

1. Bayesian Linear Regression

(1)

Q: Why we need the basis function $\phi(x)$ for linear regression? And what is the benefit for applying basis function over linear regression? (5%)

A:

x 代表所有的feature集合，basis function $\phi(x)$ 可以針對要 train的 feature作 preprocessing，以及做scaling，把每個feature scale在0到1之間。

(2)

Q: Prove that the predictive distribution just mentioned is the same with the form

$$p(t|x, X, T) = N(T|m(x), s^2(x))$$

Where

$$m(x) = \beta\phi(x)^T S \ \ \phi(x_n)t_n \qquad s^2(x) = \beta^{-1} + \phi(x)^T S\phi(x).$$

Here, the matrix S^{-1} is given by $S^{-1} = \alpha I + \beta \ {}^{N}{}_{n=1} \phi(x_n)\phi(x_n)^T$ (15%)

A:

$$p(t|x, X, T) = \int_{-\infty}^{\infty} p(t, w|x, X, T)dw$$

Weight is given

$$= \int_{-\infty}^{\infty} p(t|w, x, X, T)p(w|x, X, T)dw$$

Target t and Training data X,T are unrelated, Weight w and testing data x also unrelated,

$$= \int_{-\infty}^{\infty} p(t|w, x)p(w|X, T)dw$$

$p(w|X, T) = p(T|X, w)p(w)$

$$= \int_{-\infty}^{\infty} p(t|w, x)p(T|X, w)p(w)dw$$

$p(t|w, x) = N(t|y(w, x), \beta^{-1}I), \quad p(T|X, w) = N(T|y(X, w), \beta^{-1}I), \quad p(w) = N(w|0, \alpha^{-1}I)$

$$= \int_{-\infty}^{\infty} N(t|w^T\phi(x), \beta^{-1}I)N(T|y(X, w), \beta^{-1}I)N(w|0, \alpha^{-1}I)dw$$

$$= N(t|\beta\phi(x)^T S\sum_{n=1}^{N}\phi(x_n)t_n, \beta^{-1} + \phi(x)^T S\phi(x)) \qquad Q.E.D$$

(3)
Q: Could we use linear regression function for classification? Why or why not? Explain it! (10%)

A:
可以，我們可以在model的output加一個 Sigmoid Function，把輸出控制在0到1，0到0.5是一個class，0.5到1是另一個class，就可以做分類了。

2. Feature Select

(a)
Q: In the feature selection stage, please apply polynomials of order M = 1 and M = 2 over the dimension D = 7 input data. Please evaluate the corresponding RMS error on the training set (15%) Code Result

A:
In M=1, Root Mean Square (RMS) Error is 0.0595042

n [50]:
```
# Gradient Descent M=1
# ydata = b + w*xdata
b = 0.0
w = np.ones(7)
lr = 1
epoch = 20000
b_lr = 0.0
w_lr = np.zeros(7)
```

n [51]:
```
for e in range(epoch):
  # Calculate the value of the loss function
  error = Y - b - np.dot(x_data, w) #shape: (500,)
  loss = np.mean(np.square(error)) # Mean Square Error

  # Calculate gradient
  b_grad = -2*np.sum(error)*1 #shape: ()
  w_grad = -2*np.dot(error, x_data) #shape: (7,)

  # update learning rate
  b_lr = b_lr + b_grad**2
  w_lr = w_lr + w_grad**2

  # update parameters.
  b = b - lr/np.sqrt(b_lr) * b_grad
  w = w - lr/np.sqrt(w_lr) * w_grad

  # Print "Root Mean Square Error" per 1000 epoch
  if (e+1) % 1000 == 0:
    print('epoch:{}\n Loss:{}'.format(e+1, np.sqrt(loss)))
```

```
epoch:2000
 Loss:0.0595042087913924
epoch:4000
 Loss:0.05950420877764978
epoch:6000
 Loss:0.05950420877764953
epoch:8000
 Loss:0.05950420877764953
epoch:10000
 Loss:0.05950420877764953
epoch:12000
 Loss:0.05950420877764953
epoch:14000
 Loss:0.05950420877764953
epoch:16000
 Loss:0.05950420877764953
epoch:18000
 Loss:0.05950420877764953
epoch:20000
 Loss:0.05950420877764953
```

In M=2, Root Mean Square (RMS) Error is 0.0590381

In [55]:
```
# Gradient Descent M=2
# ydata = b + w2*xdata + w1*xdata^2
b = 0.0
w1 = np.ones(7)
w2 = np.ones(7)
lr = 1
epoch = 20000
b_lr = 0.0
w1_lr = np.zeros(7)
w2_lr = np.zeros(7)
```

In [57]:
```
for e in range(epoch):
  # Calculate the value of the loss function
  x_data_square = np.square(x_data) #shape: (500,7)
  error = Y - b - np.dot(x_data, w2)-np.dot(x_data_square, w1) #shape: (500,)
  loss = np.mean(np.square(error)) # Mean Square Error

  # Calculate gradient
  b_grad = -2*np.sum(error)*1 #shape: ()
  w1_grad = -2*np.dot(error, x_data_square) #shape: (7,)
  w2_grad = -2*np.dot(error, x_data) #shape: (7,)

  # update learning rate
  b_lr = b_lr + b_grad**2
  w1_lr = w1_lr + w1_grad**2
  w2_lr = w2_lr + w2_grad**2

  # update parameters.
  b = b - lr/np.sqrt(b_lr) * b_grad
  w1 = w1 - lr/np.sqrt(w1_lr) * w1_grad
  w2 = w2 - lr/np.sqrt(w2_lr) * w2_grad

  # Print "Root Mean Square Error" per 1000 epoch
  if (e+1) % 2000 == 0:
    print('epoch:{}\n Loss:{}'.format(e+1, np.sqrt(loss)))
```

```
epoch:2000
 Loss:0.059039211771226806
epoch:4000
 Loss:0.05903883567163798
epoch:6000
 Loss:0.05903859268360228
epoch:8000
 Loss:0.059038435591609636
epoch:10000
 Loss:0.05903833399516438
epoch:12000
 Loss:0.059038268276338304
epoch:14000
 Loss:0.05903822576043161
epoch:16000
 Loss:0.05903819825341032
epoch:18000
 Loss:0.05903818045607411
epoch:20000
 Loss:0.05903816894066609
```

(b)
Q: How will you analysis the weights of polynomial model M = 1 and select the most contributive feature? Code Result, Explain (10%)

A:
In M=1 model, y = w*x_data + b, Loss function = Y - b - w*x
跟據我的 gradient descent 跑出來的結果, 在epoch=20000時, 此時的weight
weight = [0.0929, 0.0778, 0.0238, 0.0063, 0.0674, 0.3694, 0.0243]
X_data = [GRE, TOEFL, University ranking, SOP, LOR, CGPA, Research]

可以看出 CGPA這個 feature對於 school admittion最具有影響，所以 CGPA is the most contributive feature.

Code Result:

```
In [58]: for e in range(epoch):
           # Calculate the value of the loss function
           error = Y - b - np.dot(x_data, w) #shape: (500,)
           loss = np.mean(np.square(error)) # Mean Square Error

           # Calculate gradient
           b_grad = -2*np.sum(error)*1 #shape: ()
           w_grad = -2*np.dot(error, x_data) #shape: (7,)

           # update learning rate
           b_lr = b_lr + b_grad**2
           w_lr = w_lr + w_grad**2

           # update parameters.
           b = b - lr/np.sqrt(b_lr) * b_grad
           w = w - lr/np.sqrt(w_lr) * w_grad

           # Print "Root Mean Square Error" per 2000 epoch
           if (e+1) % 2000 == 0:
             print('epoch:{}\n Loss:{}'.format(e+1, np.sqrt(loss)))
         print("w: ",w)
         print("b: ",b)
```

```
epoch:2000
 Loss:0.05950420879343555
epoch:4000
 Loss:0.059504208777649815
epoch:6000
 Loss:0.05950420877764953
epoch:8000
 Loss:0.05950420877764953
epoch:10000
 Loss:0.05950420877764953
epoch:12000
 Loss:0.05950420877764953
epoch:14000
 Loss:0.05950420877764953
epoch:16000
 Loss:0.05950420877764953
epoch:18000
 Loss:0.05950420877764953
epoch:20000
 Loss:0.05950420877764953
w:  [0.09292532 0.07778323 0.02376547 0.00634455 0.06743497 0.36936137
 0.02430748]
b:  0.3482198690275523
```

3. Maximum Likelihood Approach

(a)
Q: Which basis function will you use to further improve your regression model, Polynomial, Gaussian, Sigmoidal, or hybrid? Explain (5%)

A:
我的 Linear Regression Model 採用的是二次 Polynomial Function, 因為我使用 Gradient Descent去最佳化我的 loss function, 所以不需要Gaussian跟Sigmoidal。

(b)
Q: Introduce the basis function you just decided in (a) to linear regression model and analyze the result you get. (Hint: You might want to discuss the phenomenon when model becomes too complex.) Code Result, Explain (10%)

A:
我採用了 Polynomial Function 一次函數跟二次函數兩種方法，
一次式：y = w*x_data + b， 二次式：y = w1*x_data^2 + w2*x_data + b
x_data = (500,7) 500個人, 7個feature, w=(7,) 每個feature的weight, b=常數
定義了loss function = y - w*x_data - b 和
loss function = y - w1*x_data^2 - w2*x_data - b

用Gradient descent去最佳化loss function, 當Gradient越大, learning rate就越大, 下降越快, 當Gradient越小, learning rate就越小, 下降越慢, 越來越逼近最佳解

最後得到的結論是使用二次式：y = w1*x_data^2 + w2*x_data + b 所得到的RMS Error會比較使用一次式還要小。

使用Gradient descent最佳化不能用在三次以上或更複雜的Model, 因為Gradient descent有可能找到函數的Local Minimum，這時候就不會是最佳解。

Code Result:
一次式：y = w*x_data + b

n [50]:
```
# Gradient Descent M=1
# ydata = b + w*xdata
b = 0.0
w = np.ones(7)
lr = 1
epoch = 20000
b_lr = 0.0
w_lr = np.zeros(7)
```

n [51]:
```
for e in range(epoch):
  # Calculate the value of the loss function
  error = Y - b - np.dot(x_data, w) #shape: (500,)
  loss = np.mean(np.square(error)) # Mean Square Error

  # Calculate gradient
  b_grad = -2*np.sum(error)*1 #shape: ()
  w_grad = -2*np.dot(error, x_data) #shape: (7,)

  # update learning rate
  b_lr = b_lr + b_grad**2
  w_lr = w_lr + w_grad**2

  # update parameters.
  b = b - lr/np.sqrt(b_lr) * b_grad
  w = w - lr/np.sqrt(w_lr) * w_grad

  # Print "Root Mean Square Error" per 1000 epoch
  if (e+1) % 1000 == 0:
    print('epoch:{}\n Loss:{}'.format(e+1, np.sqrt(loss)))
```

```
epoch:2000
 Loss:0.0595042087913924
epoch:4000
 Loss:0.05950420877764978
epoch:6000
 Loss:0.05950420877764953
epoch:8000
 Loss:0.05950420877764953
epoch:10000
 Loss:0.05950420877764953
epoch:12000
 Loss:0.05950420877764953
epoch:14000
 Loss:0.05950420877764953
epoch:16000
 Loss:0.05950420877764953
epoch:18000
 Loss:0.05950420877764953
epoch:20000
 Loss:0.05950420877764953
```

二次式：y = w1*x_data^2 + w2*x_data + b

In [55]:
```
# Gradient Descent M=2
# ydata = b + w2*xdata + w1*xdata^2
b = 0.0
w1 = np.ones(7)
w2 = np.ones(7)
lr = 1
epoch = 20000
b_lr = 0.0
w1_lr = np.zeros(7)
w2_lr = np.zeros(7)
```

In [57]:
```
for e in range(epoch):
  # Calculate the value of the loss function
  x_data_square = np.square(x_data) #shape: (500,7)
  error = Y - b - np.dot(x_data, w2)-np.dot(x_data_square, w1) #shape: (500,)
  loss = np.mean(np.square(error)) # Mean Square Error

  # Calculate gradient
  b_grad = -2*np.sum(error)*1 #shape: ()
  w1_grad = -2*np.dot(error, x_data_square) #shape: (7,)
  w2_grad = -2*np.dot(error, x_data) #shape: (7,)

  # update learning rate
  b_lr = b_lr + b_grad**2
  w1_lr = w1_lr + w1_grad**2
  w2_lr = w2_lr + w2_grad**2

  # update parameters.
  b = b - lr/np.sqrt(b_lr) * b_grad
  w1 = w1 - lr/np.sqrt(w1_lr) * w1_grad
  w2 = w2 - lr/np.sqrt(w2_lr) * w2_grad

  # Print "Root Mean Square Error" per 1000 epoch
  if (e+1) % 2000 == 0:
    print('epoch:{}\n Loss:{}'.format(e+1, np.sqrt(loss)))
```

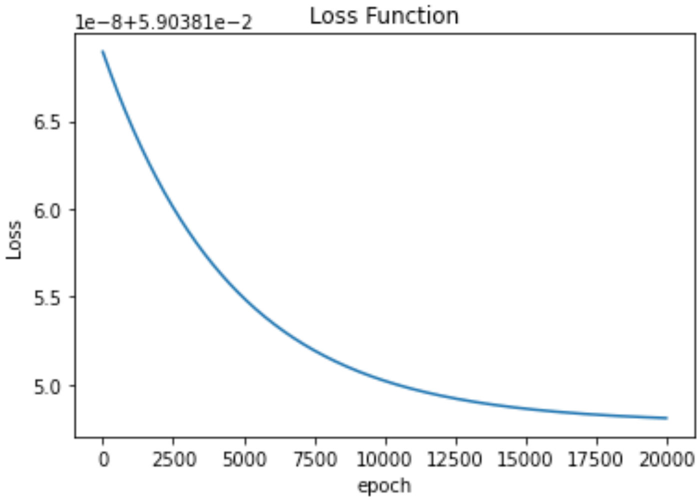
```
epoch:2000
 Loss:0.059039211771226806
epoch:4000
 Loss:0.05903883567163798
epoch:6000
 Loss:0.05903859268360228
epoch:8000
 Loss:0.059038435591609636
epoch:10000
 Loss:0.05903833399516438
epoch:12000
 Loss:0.059038268276338304
epoch:14000
 Loss:0.05903822576043161
epoch:16000
 Loss:0.05903819825341032
epoch:18000
 Loss:0.05903818045607411
epoch:20000
 Loss:0.05903816894066609
```

(c)
Q: Apply N-fold cross-validation in your training stage to select at least one hyper-parameter(order, parameter number, ...) for model and do some discussion(underfitting, overfitting). Code Result, Explain (10%)

A:
從下圖可以看到，隨著epoch次數增加，loss 慢慢地收斂，loss越大收斂較快，loss越小時收斂越較慢，既沒有overfitting，也沒有underfitting。

Code Result:

In [32]:
```
plt.plot(plt_epoch, plt_loss)
plt.title('Loss Function')
plt.ylabel('Loss')
plt.xlabel('epoch')
plt.show()
```



4. Maximum A Posterior Approach

(a)
Q: What is the key difference between maximum likelihood approach and maximum a posterior approach? Explain (5%)

A:
Maximum Likelihood 認為所有 θ 出現的機率是均等的，只考慮當下的機率，所以會把「這一次的實驗」所算出來的機率當作這整個事件的機率。

Maximum A Posterior 會先算出 θ 出現的機率，包括 θ 怎麼來的，手中先握著一個先驗機率 (Prior Probability) ，再透過不斷觀察新的實驗來更新手上的機率。

(b)
Q: Use Maximum a posterior approach method to retest the model in 2 you designed. You could choose Gaussian distribution as a prior. Code Result (10%)

A:

In [175]:
```
#calculate MAP theta
hypothesis = np.linspace(0, 1, 101)
theta_hat_1 = hypothesis[np.argmax(w1)]
theta_hat_2 = hypothesis[np.argmax(w2)]
print(theta_hat_1,theta_hat_2)
```

```
0.02 0.05
```

In [176]:
```
# Gradient Descent M=2 for Maximum a posterior approach
# ydata = b + w2*xdata + w1*xdata^2
b = 0.0
w1 = np.ones(7)
w2 = np.ones(7)
lr = 1
epoch = 20000
b_lr = 0.0
w1_lr = np.zeros(7)
w2_lr = np.zeros(7)
```

In [180]:
```
pltmap_loss =[]
pltmap_epoch = []
for e in range(epoch):
  # Calculate the value of the loss function
  x_data_square = np.square(x_data) #shape: (500,7)
  error = Y - b - theta_hat_1*np.dot(x_data, w2)- theta_hat_2*np.dot(x_data_square, w1)
  loss = np.mean(np.square(error)) # Mean Square Error

  # Calculate gradient
  b_grad = -2*np.sum(error)*1 #shape: ()
  w1_grad = -2*np.dot(error, x_data_square) #shape: (7,)
  w2_grad = -2*np.dot(error, x_data) #shape: (7,)

  # update learning rate
  b_lr = b_lr + b_grad**2
  w1_lr = w1_lr + w1_grad**2
  w2_lr = w2_lr + w2_grad**2

  # update parameters.
  b = b - lr/np.sqrt(b_lr) * b_grad
  w1 = w1 - lr/np.sqrt(w1_lr) * w1_grad
  w2 = w2 - lr/np.sqrt(w2_lr) * w2_grad

  pltmap_loss.append(np.sqrt(loss))
  pltmap_epoch.append(e)
  # Print "Root Mean Square Error" per 2000 epoch
  if (e+1) % 2000 == 0:
    print('epoch:{}\n MAP Loss:{}'.format(e+1, np.sqrt(loss)))
```

```
epoch:2000
 MAP Loss:0.059299922504985496
epoch:4000
 MAP Loss:0.05927349890374746
epoch:6000
 MAP Loss:0.05925087590137326
epoch:8000
 MAP Loss:0.05923119619268617
epoch:10000
 MAP Loss:0.059213856288005434
epoch:12000
 MAP Loss:0.05919842368600604
epoch:14000
 MAP Loss:0.05918458189042279
epoch:16000
 MAP Loss:0.05917209379455025
epoch:18000
 MAP Loss:0.05916077722045926
epoch:20000
 MAP Loss:0.059150488517141976
```

(c)
Q: Compare the result between maximum likelihood approach and maximum a posterior approach. Is it consistent with your conclusion in (a)? Explain (5%)

A:
根據我的計算結果
maximum likelihood MSE error = 0.0590381
maximum a posterior MSE error = 0.05915
兩者的Error 差不多，使用Maximum Likelihood 較佳一點