1. Given that $R_{tr}(B) = \frac{1}{N}\sum_{i=1}^{N}(y_i - B^T x_i)^2$

$R_{te}(B) = \frac{1}{M}\sum_{i=1}^{M}(\tilde{y}_i - B^T \tilde{x}_i)^2$

At regression model with p parameters

$y = XB + \epsilon$, where $\epsilon_i \overset{iid}{\sim} N(0, 6^2)$

solving $y = XB + \epsilon$,

I get the least square estimate of parameter B

$\hat{B} = (X'X)^{-1}X'y$

we know that $E(\hat{B}) = B$

Let $(X_1, y_1) \cdots (X_N, y_N)$ be the training data.

$(\tilde{X}_1, \tilde{y}_1) \cdots (\tilde{X}_M, \tilde{y}_M)$ be the test data

$E(y) = E(X\hat{B}) + E(\epsilon)$

$E(y) = XB$

So expectation of y under both training data and test data give the same result.

when $\epsilon_i \overset{iid}{\sim} N(0, 6^2)$

$y_i \sim N(X_i B, 6^2)$

$\tilde{y}_i \sim N(\tilde{X}_i B, 6^2)$

We know that $s_1^2 = \frac{1}{N-1}\sum_{i=1}^{N}(y_i - \hat{B}^T x_i)^2$ is unbiased

s_1^2 is estimation of 6^2.

$E(s_1^2) = E\left(\frac{1}{N-1}\sum_{i=1}^{N}(y_i - \hat{B}^T x_i)^2\right)$

we already know that

$E(R_{tr}(\hat{B})) = E\left(\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{B}^T x_i)^2\right)$

$= \frac{N-1}{N} E\left(\frac{1}{N-1}\sum_{i=1}^{N}(y_i - \hat{B}^T x_i)^2\right)$

$= \frac{N-1}{N} E(s_1^2)$

$= \frac{N-1}{N} 6^2$

Similarly, let $s_2^2 = \frac{1}{M-1}\sum_{i=1}^{M}(\tilde{y}_i - \hat{B}^T \tilde{x}_i)^2$

which is an unbiased estimation of 6^2

$E(s_2^2) = E\left(\frac{1}{M-1}\sum_{i=1}^{M}(\tilde{y}_i - \hat{B}^T \tilde{x}_i)^2\right)$

$E(R_{te}(\hat{B})) = E\left(\frac{1}{M}\sum_{i=1}^{M}(\tilde{y}_i - \hat{B}^T \tilde{x}_i)^2\right)$

$= \frac{M-1}{M} E\left(\frac{1}{M-1}\sum_{i=1}^{M}(\tilde{y}_i - \hat{B}^T \tilde{x}_i)^2\right)$

$= \frac{M-1}{M} 6^2$

Since $E(R_{tr}(\hat{B})) = \frac{N-1}{N} 6^2$

and $E(R_{te}(\hat{B})) = \frac{M-1}{M} 6^2$

and we have $N \geq M \geq p$

we have $\frac{N-1}{N} \leq \frac{M-1}{M}$

thus $E(R_{tr}(\hat{B})) \leq E(R_{te}(\hat{B}))$

2 (a) Set $h(t) = f(x + t(y-x))$ for $t \in (0,1)$

$h'(t) = \nabla f(x + t(y-x))^T (y-x)$.

Use equation (1)

$h'(t) - h'(0) = \nabla f(x - t(y-x))^T (y-x) - \nabla f(x)^T (y-x)$

$h(1) - h(0) \ldots$

$$= (\nabla f(x) - \nabla f(x + t(y-x)))^T \cdot (y-x)$$

$$\geq mt \|x-y\|_2^2$$

$$h'(t) \geq h'(0) + mt\|x-y\|_2^2$$

use FTC, I have

$$h(1) - h(0) = \int_0^1 h'(t)\,dt$$

$$f(y) - f(x) \geq \int_0^1 h'(0) - mt\|x-y\|_2^2\,dt$$

$$\leq h'(0) + \frac{m}{2}\|x-y\|_2^2$$

$$\geq \nabla f(x)^T(y-x) + \frac{m}{2}\|x-y\|_2^2$$

$$f(y) \geq f(x) + \nabla f(x)^T(y-x) + \frac{m}{2}\|x-y\|_2^2$$

Thus I proved that (1) is equivalent to

$$f(y) \geq f(x) + \nabla f(x)^T(y-x) + \frac{m}{2}\|y-x\|_2^2$$

(b) assume $g(x) = f(x) - \frac{m}{2}\|x\|_2^2$, where $\nabla g(x) = \nabla f(x) - mx$.

Therefore I get

$$(\nabla g(x) - \nabla g(y))^T \cdot (x-y) = (\nabla f(x) - \nabla f(y) - \frac{m}{2}(x-y))^T (x-y)$$

$$= (\nabla f(x) - \nabla f(y))^T (x-y) - (m(x-y))^T (x-y)$$

$$\geq 0$$

Thus $g(x)$ is convex in domain of f.

Therefore $\nabla^2 g(x) \geq 0$ and I get $\nabla^2 f(x) - mI \geq 0$

which means that I proved that (1) is equivalent to $\nabla^2 f(x) \geq mI$

3 (a) $Pr(q_2 = \text{Happy}) = 0.8$

(b) $Pr(O_2 = \text{frown}) = 0.8 \times 0.1 + 0.2 \times 0.5$

$= 0.08 + 0.1 = 0.18$

(c) $Pr(q_2 = \text{Happy} \mid O_2 = \text{frown}) = \frac{0.8 \times 0.1}{0.18} = \frac{4}{9}$

(d) $Pr(O_{100} = \text{yell}) = Pr(O_{100} = \text{yell} \cap q_{100} = \text{happy}) + Pr(O_{100} = \text{yell} \cap q_{100} = \text{angry})$

$= 0.2$

(e) $q_1 = \text{happy}$

$p(f_2 \mid h_2) = 0.8 \times 0.1 = 0.08$

$p(f_2 \mid a_2) = 0.2 \times 0.5 = 0.1$ so $q_2 = \text{angry}$

$p(f_3 \mid h_3) = 0.2 \times 0.1 = 0.02$

$p(f_3 \mid a_3) = 0.8 \times 0.5 = 0.4$ so $q_3 = \text{angry}$.

the most likely sequence is happy, angry, angry.

4. (a) The optimization problem now becomes $R(\theta) + \lambda J(\theta)$

which equals $\sum_{i=1}^{n} \sum_{k=1}^{k} (y_{ik} - f_k(x_i))^2 + \lambda(\sum_{km} B_{km}^2 + \sum_{ml} \alpha_{ml}^2)$

taking the gradient of $R(\theta) + \lambda J(\theta)$

I get $\frac{\partial R_i + \partial \lambda J(\theta)}{\partial B_{km}} = \frac{\partial R_i}{\partial B_{km}} + \frac{\partial \lambda J(\theta)}{\partial B_{km}}$

$= \delta_{ki} Z_{mi} + 2\lambda \sum_{km} B_{km}$

and $$\frac{\partial R_i + \partial \lambda J(\theta)}{\partial a_{ml}} = S_{mi} X_{il} + 2\lambda \sum_{ml} a_{ml}$$

According to the problem, a gradient update at the $(r+1)$st iteration has the form

$$B_{km}^{(r+1)} = B^{(r)}_{km} - \frac{\partial R}{\partial B^{(r)}_{km}}, \quad a^{(r+1)}_{lm} = a^{(r)}_{ml} - \frac{\partial R}{\partial a^{(r)}_{ml}}$$

taking the new gradient update. $\frac{\partial R + \partial \lambda J(\theta)}{\partial B_{km}}$ and $\frac{\partial R + \partial \lambda J(\theta)}{\partial a_{ml}}$ into the gradient update.

I get $$B^{(r+1)}_{km} = B^{(r)}_{km} - \frac{\partial R}{\partial B^{(r)}_{km}} - \frac{\partial \lambda J(\theta)}{\partial B^{(r)}_{km}}$$

$$= B^{(r)}_{km} + 2(y_{ik} - f_k(x_i))\, g'_k(B_k^{(r)T} z_i) z_{mi} - 2\lambda \sum_{km} B^{(r)}_{km}$$

$$a^{(r+1)}_{ml} = a^{(r)}_{ml} + \sum_{k=1}^{k} 2(y_{ik} - f_k(x_i))\, g'_k(B_k^T z_i) B_{km}\, \sigma'(a_m^{(r)T} x_i) x_{il} - 2\lambda \sum_{ml} a^{(r)}_{ml}$$

Above are the gradient update for this regularized problem

(b) Stochastic gradient descent replaces the actual gradient, which is calculated by the entire dataset, with an estimate. The estimate is calculated from a randomly subset of the data. This could largely reduce the computational burden when n is large, achieving faster iterations in trade for a lower convergence rate

In [152…

```
import pandas as pd;
import numpy as np;
import matplotlib.pyplot as plt;
import seaborn as sns;
import time
```

In [153…

```
from sklearn.utils import shuffle
```

In [154…

```
Y = pd.read_csv("target.txt",names=["target"])
names = [str(int) for int in list(range(1,123))]
names = ["X"+int for int in names]
X = pd.read_csv("features.txt",names=names)
```

In [155…

```
X = X.values
Y = Y.values
```

a

In [157…

```
def softmargin(n,w,b,C,x,y):
    temp_max_sum = 0
    # j = (0,122) feature
    # i = (0,6414) number of cases
    # x[i] case i in x
    # x[i][j] case i feature j in x
    # w[j] each parameter
    temp_w_sum = np.dot(w,w)
    for i in range(n):
        temp_max_sum += max(0,1-y[i]*(np.dot(x[i],w)+b))
    return 0.5*temp_w_sum + C*temp_max_sum
```

In [158…

```
def partial_wj(w,j,xi,yi,b):
    if yi*(np.dot(xi,w)+b) >= 1:
        return 0
    else:
        return -yi*xi[j]
```

In [159…

```
def gradient_wj(w,j,C,x,y,b):
    temp_sum = 0
    for i in range(len(y)):
        temp_sum += partial_wj(w,j,x[i],y[i],b)
    return w[j] + C*temp_sum
```

In [160…

```
def partial_b(w,xi,yi,b):
    if yi*(np.dot(xi,w)+b) >= 1:
        return 0
    else:
        return -yi
```

In [161…

```python
def gradient_b(w,b,C,x,y):
    temp_sum = 0
    for i in range(len(y)):
        temp_sum += partial_b(w,x[i],y[i],b)
    return C*temp_sum
```

In [162…

```python
def BatchGradientDescent(x,y):
    k = b = 0
    eps = 0.25
    eta = 0.0000003
    d = len(x[0])
    w = np.zeros(d)
    n = len(y)
    C = 100
    cost = 10000
    f_old = softmargin(n,w,b,C,x,y)
    f = []
    while cost > eps:
        for j in range(d):
            w[j] = w[j] - eta*gradient_wj(w,j,C,x,y,b)
        b = b - eta*gradient_b(w,b,C,x,y)
        f_new = softmargin(n,w,b,C,x,y)
        cost = abs(f_new - f_old)*100/f_old
        f_old = f_new
        f.append(f_new[0])
        if k%10 == 0:
            print("cost:", cost)
            print("Iteration:", k)
        k += 1
    return f
```

In [163…

```python
Start = time.time()
BGD_cost = BatchGradientDescent(X,Y)
End = time.time()
print("The running time for BGD is:", End-Start)
```

```
cost: [38.89291868]
Iteration: 0
cost: [0.39451094]
Iteration: 10
cost: [0.31617411]
Iteration: 20
cost: [0.30402508]
Iteration: 30
cost: [0.30396612]
Iteration: 40
cost: [0.30691618]
Iteration: 50
The running time for BGD is: 370.30693888664246
```

In [164…

```python
import random
random.seed(10086)
from sklearn.utils import shuffle
X1,Y1 = shuffle(X,Y)
```

```python
def StochasticGradientDescent(x,y):
    k = b = 0
    i = 1
    eps = 0.001
    eta = 0.0001
    d = len(X[0])
    w = np.zeros(d)
    n = len(y)
    C = 100
    f_old = softmargin(n,w,b,C,x,y)
    cost_old = 0.5*f_old
    cost_new = 1000
    f = []
    while cost_new > eps:
        for j in range(d):
            w[j] = w[j] - eta*(w[j]+C*partial_wj(w,j,x[i],y[i],b))
        b = b - eta*C*partial_b(w,x[i],y[i],b)
        f_new = softmargin(n,w,b,C,x,y)
        if k == 0:
            cost_new = 0.5*(abs(f_new - f_old)*100/f_old)
        else:
            cost_new = 0.5*(abs(f_new - f_old)*100/f_old) + 0.5*cost_old
        f_old = f_new
        cost_old = cost_new
        i = (i%n)+1
        f.append(f_new[0])
        if k%500 ==0:
            print("cost:", cost_new)
            print("Iteration:", k)
        k += 1
    return f
```

In [165…

```python
Start = time.time()
SGD_cost = StochasticGradientDescent(X1,Y1)
End = time.time()
print("The running time for SGD is:", End-Start)
```

```
cost: [1.59674145]
Iteration: 0
cost: [1.30766404]
Iteration: 500
cost: [0.05772389]
Iteration: 1000
cost: [0.0298056]
Iteration: 1500
cost: [0.83853692]
Iteration: 2000
cost: [0.69810909]
Iteration: 2500
cost: [1.00549747]
Iteration: 3000
cost: [0.00212938]
Iteration: 3500
cost: [0.72745517]
Iteration: 4000
cost: [0.07439594]
Iteration: 4500
The running time for SGD is: 245.87482714653015
```

In [166…

```
def MiniBatchGradientDescent(x,y):
    k = b = 0
    eps = 0.01
    eta = 0.00001
    d = len(X[0])
    w = np.zeros(d)
    n = len(y)
    C = 100
    f_old = softmargin(n,w,b,C,x,y)
    cost_old = 0.5*f_old
    cost_new = 1000
    f = []
    batch_size = 20
    l = 1
    while cost_new >eps:
        a = int(l)*batch_size+1
        c = min(n, (int(l)+1)*batch_size)
        for j in range(d):
            w[j] = w[j] - eta*gradient_wj(w,j,C,x[a:c],y[a:c],b)
        b = b - eta*gradient_b(w,b,C,x[a:c],y[a:c])
        l = (l+1)%((n+batch_size-1)/batch_size)
        f_new = softmargin(n,w,b,C,x,y)
        if k == 0:
            cost_new = 0.5*(abs(f_new - f_old)*100/f_old)
        else:
            cost_new = 0.5*(abs(f_new - f_old)*100/f_old) + 0.5*cost_old
        f_old = f_new
        cost_old = cost_new
        f.append(f_new[0])
        if k%200 == 0:
            print("cost:", cost_new)
            print("Iteration:", k)
        k += 1
    return f
```

In [167…

```
Start = time.time()
MBGD_cost = MiniBatchGradientDescent(X1,Y1)
End = time.time()
print("The running time for MBGD is:", End-Start)
```

```
cost: [2.70376515]
Iteration: 0
cost: [0.13064641]
Iteration: 200
cost: [0.06286008]
Iteration: 400
The running time for MBGD is: 34.10087561607361
```

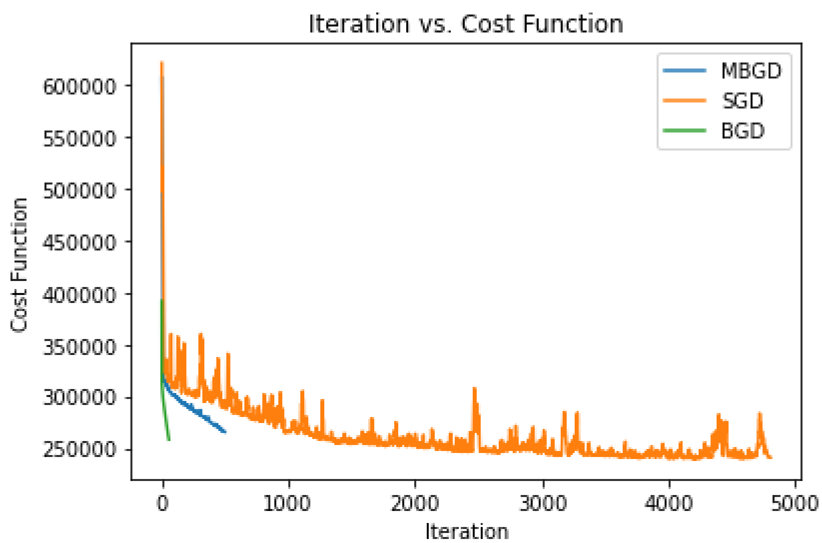
b

In [168…

```
plt.plot(list(range(len(MBGD_cost))),MBGD_cost, label = "MBGD")
plt.plot(list(range(len(SGD_cost))),SGD_cost, label = "SGD")
plt.plot(list(range(len(BGD_cost))),BGD_cost, label = "BGD")
plt.legend()
plt.title("Iteration vs. Cost Function")
plt.xlabel("Iteration")
plt.ylabel("Cost Function")
```

Out[168...

```
Text(0, 0.5, 'Cost Function')
```



We can see that Batch Gradient Descent has the lowest iteration. This is because Batch Gradient Descent takes information from the whole dataset, so it needs the least amount of iterations. Howerver, it takes the longest amount of time to converge, 370 seconds.

For Stochastic Gradient Descent, it takes the highest amount of iterations. Stochastic Gradient Descent calculates the gradient for one case each time, so it uses less time to find parameter. Because it collect information from randomly selected points, it uses less computational cost to find enough information tha Batch Gradient Descent. It takes the medium amount of time to converge, 245 seconds

For Minibatch Gradient Descent. it combines the advantage of Stochastic Gradient Descent and Batch Gradient Descent, so it takes the least amount of time to converge, 34 seconds