# 1 Problem1

## 1.1 [Q1]

Acoording to the Plugin classifier's definition:

$$\eta_1(x) \geq \eta_0(x)$$

Hence, we can have the following:

$$\mathbb{P}_{Y|X}^{(1|x)} \geq \mathbb{P}_{Y|X}^{(0|x)}$$

$$\Leftrightarrow \frac{\mathbb{P}_{X|Y}^{(x|1)}\mathbb{P}_Y(1)}{\mathbb{P}_X(x)} \geq \frac{\mathbb{P}_{X|Y}^{(x|0)}\mathbb{P}_Y(0)}{\mathbb{P}_X(x)}$$

$$\Leftrightarrow \mathbb{P}_{X|Y}^{(x|1)}\mathbb{P}_Y(1) \geq \mathbb{P}_{X|Y}^{(x|0)}\mathbb{P}_Y(0)$$

Since, we have $\mathbb{P}_Y(1) = 0.5$, we have

$$\mathbb{P}_Y(1) = \mathbb{P}_Y(0) = 0.5$$

Thus we can arrive at:

$$\Leftrightarrow \mathbb{P}_{X|Y}^{(x|1)} \geq \mathbb{P}_{X|Y}^{(x|0)}$$

From definition, $\mathbb{P}_{X|Y}^{(x|0)} \sim N(0,1)$ and $\mathbb{P}_{X|Y}^{(x|1)} \sim N(2,1)$.

$$\Leftrightarrow \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x-2)^2}{2}\right) \geq \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$$

Take the log,

$$\Leftrightarrow -\frac{(x-2)^2}{2} \geq -\frac{x^2}{2}$$

$$\Leftrightarrow 4x - 4 \geq 0$$

Hence, the decision boundary is linear. And $w_0 = -4, w_1 = 4$.

## 1.2 [Q2]

Since, we have $\mathbb{P}_Y(1) = 0.5$, we have

$$\mathbb{P}_Y(1) = \mathbb{P}_Y(0) = 0.5$$

We can still use the Equivalent formula in [Q1], thus we have,

$$\Leftrightarrow \mathbb{P}_{X|Y}^{(x|1)} \geq \mathbb{P}_{X|Y}^{(x|0)}$$

$$\Leftrightarrow \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \geq \sqrt{\frac{2}{\pi}} \exp\left(-2x^2\right)$$

$$\Leftrightarrow \exp(-\frac{x^2}{2}) \geq 2\exp(-2x^2)$$

Take the log, we have,

$$\Leftrightarrow -\frac{x^2}{2} \geq \ln 2 - 2x^2$$

$$\Leftrightarrow \frac{3}{2}x^2 - \ln 2 \geq 0$$

Thus, the decision boundary is not linear, it's parabola.
Only the last statement "Decision Boundary is not linear" is correct. Since parabola is symmetric about $x = 0$. No matter $x \geq \frac{1}{2}, -1$ or $x \leq -\frac{1}{2}, 1$, we can't make sure it's label is "1".

## 1.3 [Q3]

Now we have the decision boundary

$$x^2 \geq \frac{2}{3} \ln 2$$

Thus we can have the following statement is true

- Classfy as class 1 if $x \geq \sqrt{\frac{2}{3} \ln 2}$ or $x \leq -\sqrt{\frac{2}{3} \ln 2}$

## 1.4 [Q4]

$$\begin{aligned}
L(a_{i,k}, b_{i,k}) &= \prod_{k=1}^{K} \mathbb{P}(x|y = y_k) \\
&= \prod_{k=1}^{K} \prod_{i=1}^{2} \mathbb{P}(x_i|y = y_k) \\
&= \prod_{k=1}^{K} \prod_{i=1}^{2} \frac{1}{b_{i,k} - a_{i,k}} \\
&= \prod_{i=1}^{2} \left( \frac{1}{b_{i,k} - a_{i,k}} \right)^{K}
\end{aligned}$$

Take the log, we can arrive at

$$\ell(a_{i,k}, b_{i,k}) = K \sum_{i=1}^{2} \ln \frac{1}{b_{i,k} - a_{i,k}}$$

we have know that $a_{i,k} \leq x_i \leq b_{i,k}$. Since $\ell(a_{i,k}, b_{i,k})$ is decreasing as $x_i$ goes larger. The arrive the largest $\ell(a_{i,k}, b_{i,k})$, we have,

$$\hat{b_{i,k}} = min\, b_{i,k} = max\, x_{i,k}$$

$$\hat{a_{i,k}} = max\, a_{i,k} = min\, x_{i,k}$$

## 1.5 [Q5]

we know that

$$h_B(x) = arg\,\underset{k}{max}\,\mathbb{P}(x_i|y = y_k)\mathbb{P}(y = y_k)$$

Thus,

$$h_B(x) = \begin{cases} \prod_{i=1}^{2} \frac{\hat{\pi_k}}{b_{i,k}-a_{i,k}}, & a_{i,k}, b_{i,k} \in S \\ \\ 0, & others \end{cases} \tag{1}$$

where, $S = \{(a_{i,k}, b_{i,k})|a_{1,k} \le x_1 \le b_{1,k} \cap a_{2,k} \le x_2 \le b_{2,k}\}$ Let's assume that there are 2 classes $y_{k1}, y_{k2}$

• If there is no overlapping between two areas. As is shown in Figure 1,

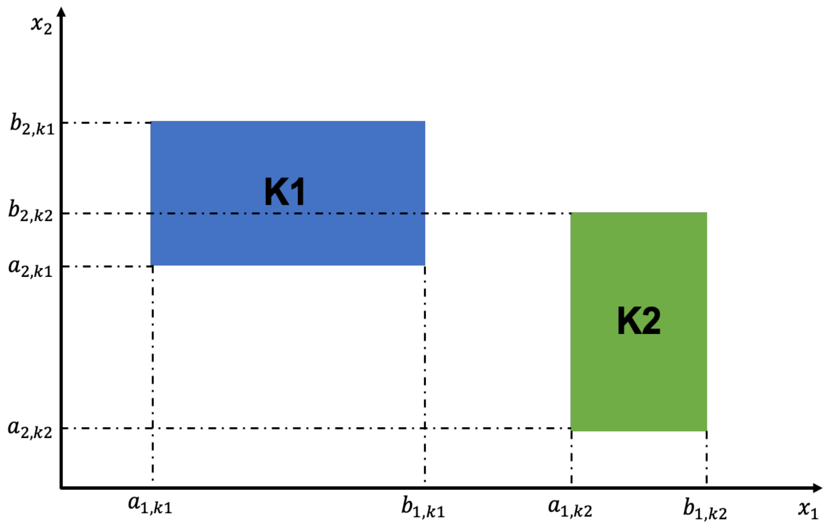


Figure 1: No Overlapping

the decision boundary is two no-overlapping rectangles.

• If there is overlapping between two areas. As is shown in Figure 2, the black region is the overlapping area. Note that if we assume $\eta_{k1} \ge \eta_{k2}$, namely:

$$\frac{\hat{\pi_{k1}}}{\prod_{i=1}^{2}(b_{i,k1} - a_{i,k1})} \ge \frac{\hat{\pi_{k2}}}{\prod_{i=1}^{2}(b_{i,k2} - a_{i,k2})}$$

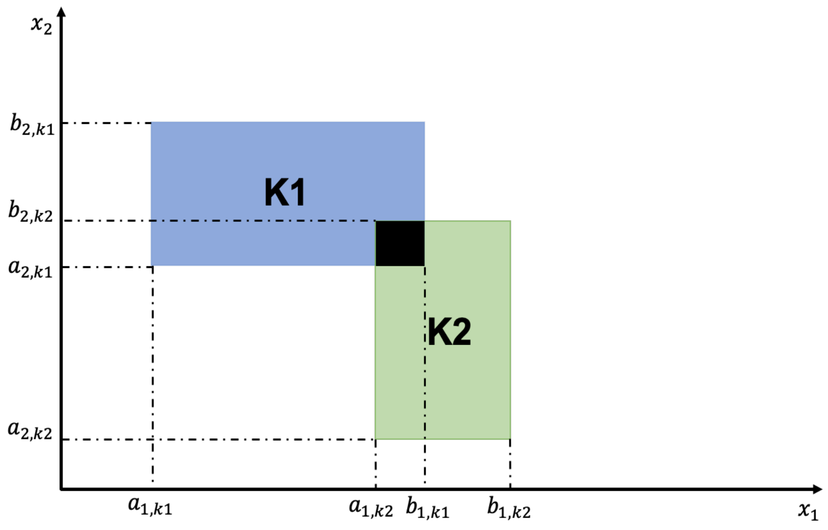


Figure 2: Overlapping

Then the decision boundary for class $k1$ is the blue region plus black region. And boundary for the class $k2$ is the green region subtract the black region, vice versa.

## 2 Problem2

### 2.1 [Q1]

I set the $\alpha$ as 0.0001, and the number of iterations required for convergence is 1924. The Figure 3 shows the "cost" decreasing with iterations.

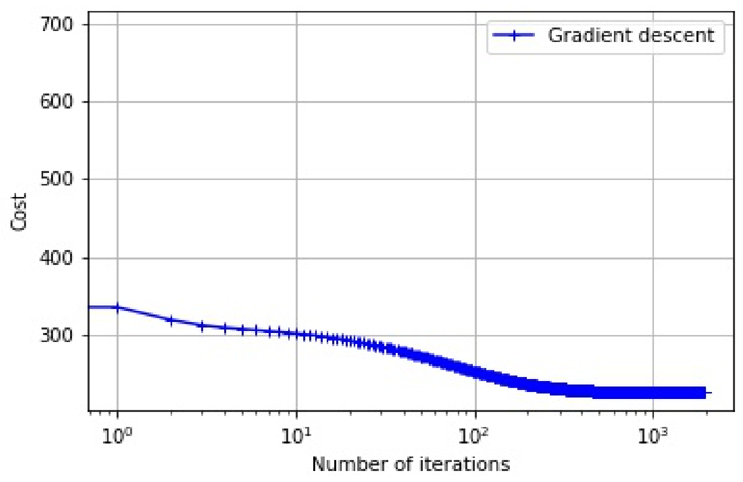


Figure 3: The "cost" decreasing with iterations.

## 2.2 [Q2]

The gradient of $\theta$ is:

$$\nabla_\theta \ell(\theta) = \sum_{i=1}^{N} x_i \left( y_i - \frac{1}{1+\exp(-\theta^T x_i)} \right)$$

Take the derivative of $\theta$ again, we can arrive:

$$\begin{aligned}
\nabla_\theta^2 \ell(\theta) &= \nabla_\theta \sum_{i=1}^{N} x_i y_i - \nabla_\theta \sum_{i=1}^{N} \frac{1}{1+\exp(-\theta^T x_i)} x_i \\
&= 0 - \nabla_\theta \sum_{i=1}^{N} \frac{1}{1+\exp(-\theta^T x_i)} x_i \\
&= -\sum_{i=1}^{N} \left[ \frac{1}{1+\exp(-\theta^T x_i)} \left( 1 - \frac{1}{1+\exp(-\theta^T x_i)} \right) \nabla_\theta \left( -\theta^T x_i \right) \right] x_i \\
&= \sum_{i=1}^{N} \left[ \frac{1}{1+\exp(-\theta^T x_i)} \left( 1 - \frac{1}{1+\exp(-\theta^T x_i)} \right) x_i \right] x_i \\
&= \sum_{i=1}^{N} x_i x_i^T \frac{1}{1+\exp(-\theta^T x_i)} \left( \frac{1}{1+\exp(-\theta^T x_i)} \right)
\end{aligned}$$

## 2.3 [Q3]

The code below is used to calculate the $\nabla_\theta^2 \ell(\theta)$

```
def log_Hessian(theta, x):
    g = logistic_func(theta,x) #1000*1
    n = g.shape[0]
    return np.dot(np.dot(np.dot(x.T,np.diag(g.reshape(n))),np
                                            .diag(1-g.reshape(n))),x)
```

The code below is used to implement Newton's method.

```
def Newton_method(theta, x, y, tol, maxiter):
    nll_vec = []
    nll_vec.append(neg_log_like(theta, x, y))
    nll_delta = 2.0*tol
    iter = 0
    while (nll_delta > tol) and (iter < maxiter):
        theta=theta-np.dot(np.linalg.inv(log_Hessian(theta,x)
                                            ),log_grad(theta, x, y
                                            ))#(1000,1)
        nll_vec.append(neg_log_like(theta, x, y))
        nll_delta = nll_vec[-2]-nll_vec[-1]
        iter += 1
    return theta, np.array(nll_vec)
```

I set $\alpha$ as **0.0001**. The number of iterations is **8**. And the comparision between **Gradient Descent** and **Newton's Method** is shown in Figure 4.

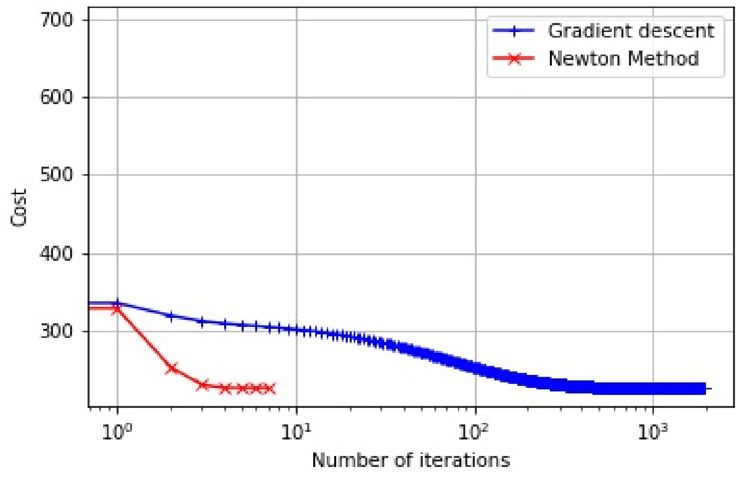


Figure 4: Comparision between Gradient Descent and Newton's Method

## 2.4 [Q4]

The code for SGD is below.

```
def Sto_Grad_Desc(theta, x, y, alpha, tol, maxiter):
    nll_vec = []
    nll_vec.append(neg_log_like(theta, x, y))
    nll_delta = 2.0*tol
    iter = 0
    index_pool = list(range(x.shape[0]))
    flunc = 3
    while (flunc > 0) and (iter < maxiter):
        # randomly pick a sample
        shuffle(index_pool)
        loss = 0
        for index in index_pool:
            if not iter < maxiter:
                return theta, np.array(nll_vec)
            x_i = x[index]
            y_i = y[index]
            # update theta
            theta = theta - alpha * log_grad(theta, x_i, y_i)
            loss += neg_log_like(theta, x_i, y_i)
            iter += 1
        nll_vec.append(loss)
        nll_delta = nll_vec[-2] - nll_vec[-1]
        # this part is used to prevent the inflence of
                                        fluncation
        if nll_delta > tol:
            flunc = 3
        else:
            flunc -= 1
    return theta, np.array(nll_vec), iter
```

In the code, I firstly randomly disorganize the order. And then traverse samples and update $\theta$ one by one. This part is the main idea of "SGD". In the end, note that the iterative procedure is fluncating, thus in order not to jump out the while loop, I set parameter "flunc" to ensure even if current "nll_delta" is smaller, we can't reckon it as convergence. It may be fluncating. Instead, let it do more iterations until the consecutive three "nll_delta" is less than "tol".
In the end, I set $\alpha$ as **0.001**, the maximum number of iterations as **maxiter = 100000**. And the comparision between **Gradient Descent** and **SGD** is shown in Table 1, it shows the speed, number of iterations and final cost.

Table 1: Comparision between GD and SGD

| | iterations | Speed | Cost |
|---|---|---|---|
| GD | 1924 | 0.270897 | 226.365 |
| SGD | 100000 | 4.556203 | 223.393 |

## 2.5 [Q5]

In 100,000 samples, the parameter I set for each algorithm is listed in Table 2.

Table 2: Parameter for three algorithm

| | $\alpha$ | tol | maxiter |
|---|---|---|---|
| GD | 0.000001 | 1e-6 | 10000 |
| Newton | | 1e-6 | 10000 |
| SGD | 0.0001 | 1e-6 | 1000000 |

For Gradient Descent: Final GD cost: 23830.493 after 2554 iterations in 30.937585 seconds

$$[1.80175736 - 0.26290611 - 0.33760097]$$

For Newton's Method: Final Newton cost: 23830.493 after 8 iterations in 600.265969 seconds

$$[1.80207019 - 0.26293566 - 0.33762014]$$

For SGD: Final SGD cost: 23795.735 after 1000000 iterations in 39.221972 seconds

$$[1.75690746 - 0.26546559 - 0.33761656]$$

From these data, we can conclude the Newton's Method is the most time-consuming. It takes nearly 600 seconds (10 minutes) to compute 100,000 2-d data. Besides, for Gradient Desecnt and SGD, apparently, GD takes more iterations to complete, while SGD takes less iterations. Although it seems that SGD (39.221972 seconds) takes more time than GD (30.937585 seconds), but SGD iterates 1000,000 times while GD only iterates 2554 times, actually, If the precision is not so strict, we can let SGD iterates less and get a not so bad result.

For example, agian I set SGD parameter **maxiter** as **100,000**, the other two parameters as the same as before. we get the new result as follows:
For SGD: Final SGD cost: 28488.893 after 100000 iterations in 4.063194 seconds

$$[0.7247389 - 0.17032292 - 0.29207908]$$

From the result ,we can see that the time is extremely fast, only takes 4.063194 seconds. And the Final SGD cost is not so bad, 28488.893 is closed to GD's cost (23830.493).
Thus, we can conclude that
In terms of **Precision:** Stochastic Gradient Descent less precise than Newton's Method = Gradient Descent
In terms of **Speed:** Newton's Method slower than Gradient Descent slower than Stochastic Gradient Descent.

# 3 Code

Gradient descent

September 28, 2020

## 1 Loading modules

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from math import exp
import time
from numpy import *
from random import shuffle
```

## 2 Auxiliary functions

[2]:
```python
# the logistic function
def logistic_func(theta, x):
    t = x.dot(theta)
    g = np.zeros(t.shape)
    # split into positive and negative to improve stability
    g[t>=0.0] = 1.0 / (1.0 + np.exp(-t[t>=0.0]))
    g[t<0.0] = np.exp(t[t<0.0]) / (1.0+ np.exp(t[t<0.0]))
    return g

# function to compute log-likelihood
def neg_log_like(theta, x, y):
    g = logistic_func(theta,x)
#     print(g.shape)
    return -sum(np.log(g[y>0.5])) - sum(np.log(1-g[y<0.5]))

# function to compute the gradient of the negative log-likelihood
def log_grad(theta, x, y):
    g = logistic_func(theta,x)
    return -x.T.dot(y-g)

def log_Hessian(theta, x):
    g = logistic_func(theta,x) #1000*1
    n = g.shape[0]
```

1

```
    return np.dot(np.dot(np.dot(x.T,np.diag(g.reshape(n))),np.diag(1-g.
↪reshape(n))),x)
```

## 3 Gradient descent for Logistic Regression

[24]:
```
# implementation of gradient descent for logistic regression
def grad_desc(theta, x, y, alpha, tol, maxiter):
    nll_vec = []
    nll_vec.append(neg_log_like(theta, x, y))
    nll_delta = 2.0*tol
    iter = 0
    while (nll_delta > tol) and (iter < maxiter):
#          print(theta)
        theta = theta - (alpha * log_grad(theta, x, y))
        nll_vec.append(neg_log_like(theta, x, y))
        nll_delta = nll_vec[-2]-nll_vec[-1]
        iter += 1
    return theta, np.array(nll_vec)

# function to compute output of LR classifier
def lr_predict(theta,x):
    # form Xtilde for prediction
    shape = x.shape
    Xtilde = np.zeros((shape[0],shape[1]+1))
    Xtilde[:,0] = np.ones(shape[0])
    Xtilde[:,1:] = x
    return logistic_func(theta,Xtilde)

def Newton_method(theta, x, y, tol, maxiter):
    nll_vec = []
    nll_vec.append(neg_log_like(theta, x, y))
    nll_delta = 2.0*tol
    iter = 0
    while (nll_delta > tol) and (iter < maxiter):
        theta=theta-np.dot(np.linalg.inv(log_Hessian(theta,x)),log_grad(theta,␣
↪x, y))#(1000,1)
        nll_vec.append(neg_log_like(theta, x, y))
        nll_delta = nll_vec[-2]-nll_vec[-1]
        iter += 1
    return theta, np.array(nll_vec)

def Sto_Grad_Desc(theta, x, y, alpha, tol, maxiter):
    nll_vec = []
    nll_vec.append(neg_log_like(theta, x, y))
    nll_delta = 2.0*tol
    iter = 0
    index_pool = list(range(x.shape[0]))
    flunc = 3
    while (flunc > 0) and (iter < maxiter):
        # randomly pick a sample
        shuffle(index_pool)
        loss = 0
        for index in index_pool:
            if not iter < maxiter:
                return theta, np.array(nll_vec)
            x_i = x[index]
            y_i = y[index]
            # update theta
            theta = theta - alpha * log_grad(theta, x_i, y_i)
            loss += neg_log_like(theta, x_i, y_i)
            iter += 1
        nll_vec.append(loss)
        nll_delta = nll_vec[-2] - nll_vec[-1]
        # this part is used to prevent the inflence of fluncation
        if nll_delta > tol:
            flunc = 3
        else:
            flunc -= 1
    return theta, np.array(nll_vec), iter
```

## 4 Generating dataset Sample Number 1000

[4]:
```
np.random.seed(2020) # Set random seed so results are repeatable (do not change)
x,y = datasets.make_blobs(n_samples=1000,n_features=2,centers=2,cluster_std=6.0)

## build classifier form Xtilde
shape = x.shape
xtilde = np.zeros((shape[0],shape[1]+1))
xtilde[:,0] = np.ones(shape[0])
xtilde[:,1:] = x
```

## 5 Run Gradient Descent

[5]:
```
# Initialize theta to zero
theta = np.zeros(shape[1]+1)
# Run gradient descent
#alpha = 0.0001,maxiter=10000
alpha = 0.0001
tol = 1e-6
maxiter = 10000
```

3

```
start_time = time.time()
#using Gradient Descent
theta_gd, cost_gd = grad_desc(theta,xtilde,y,alpha,tol,maxiter)
print(theta_gd)
stop_time = time.time()
print("Final GD cost: %.3f after %d iterations in %f seconds" %␣
 ↪(cost_gd[-1],cost_gd.size,stop_time-start_time))
```

```
[ 1.89927372 -0.29532015 -0.36595797]
Final GD cost: 226.365 after 1924 iterations in 0.256847 seconds
```

## 6 Run Newton Method

[6]:
```
# Initialize theta to zero
theta = np.zeros(shape[1]+1)
# Run gradient descent
#alpha = 0.0001,maxiter=10000
alpha = 0.0001
tol = 1e-6
maxiter = 10000

#using Newton's Method
start_time = time.time()
theta_ngd, cost_ngd = Newton_method(theta,xtilde,y,tol,maxiter)
print(theta_ngd)
stop_time = time.time()
print("Final Newton cost: %.3f after %d iterations in %f seconds" %␣
 ↪(cost_ngd[-1],cost_ngd.size,stop_time-start_time))
```

```
[ 1.90256321 -0.2956653  -0.36619489]
Final Newton cost: 226.365 after 8 iterations in 0.027727 seconds
```

## 7 Run SGD

[7]:
```
# Initialize theta to zero
theta = np.zeros(shape[1]+1)
# Run gradient descent
#alpha = 0.0001,maxiter=10000
alpha = 0.001
tol = 1e-6
maxiter = 100000

start_time = time.time()
#using SGD
theta_sgd, cost_sgd, iter = Sto_Grad_Desc(theta,xtilde,y,alpha,tol,maxiter)
```

4

```
print(theta_sgd)
stop_time = time.time()
print("Final SGD cost: %.3f after %d iterations in %f seconds" %␣
 ↪(cost_sgd[-1],iter,stop_time-start_time))
```

```
[ 1.85609907 -0.28564398 -0.34971833]
Final SGD cost: 223.393 after 100000 iterations in 4.278880 seconds
```

## 8 Generating dataset Sample Number 100,000

[9]:
```
np.random.seed(2020) # Set random seed so results are repeatable (do not change)
x,y = datasets.make_blobs(n_samples=100000,n_features=2,centers=2,cluster_std=6.
 ↪0)

## build classifier form Xtilde
shape = x.shape
xtilde = np.zeros((shape[0],shape[1]+1))
xtilde[:,0] = np.ones(shape[0])
xtilde[:,1:] = x
```

## 9 Run SGD in Sample Number 100,000

[13]:
```
# Initialize theta to zero
theta = np.zeros(shape[1]+1)
# Run gradient descent
#alpha = 0.0001,maxiter=10000
alpha = 0.0001
tol = 1e-6
maxiter = 100000

start_time = time.time()
#using SGD
theta_sgd, cost_sgd, iter = Sto_Grad_Desc(theta,xtilde,y,alpha,tol,maxiter)

print(theta_sgd)
stop_time = time.time()
print("Final SGD cost: %.3f after %d iterations in %f seconds" %␣
 ↪(cost_sgd[-1],iter,stop_time-start_time))
```

```
[ 0.7247389  -0.17032292 -0.29207908]
Final SGD cost: 28488.893 after 100000 iterations in 4.063194 seconds
```

## 10 Run Gradient Descent

[23]:
```
# Initialize theta to zero
theta = np.zeros(shape[1]+1)
# Run gradient descent
#alpha = 0.0001,maxiter=10000
alpha = 0.000001
tol = 1e-6
maxiter = 10000

start_time = time.time()
#using Gradient Descent
theta_gd, cost_gd = grad_desc(theta,xtilde,y,alpha,tol,maxiter)
print(theta_gd)
stop_time = time.time()
print("Final GD cost: %.3f after %d iterations in %f seconds" %␣
 ↪(cost_gd[-1],cost_gd.size,stop_time-start_time))
```

```
[ 1.80175736 -0.26290611 -0.33760097]
Final GD cost: 23830.493 after 2554 iterations in 30.937585 seconds
```

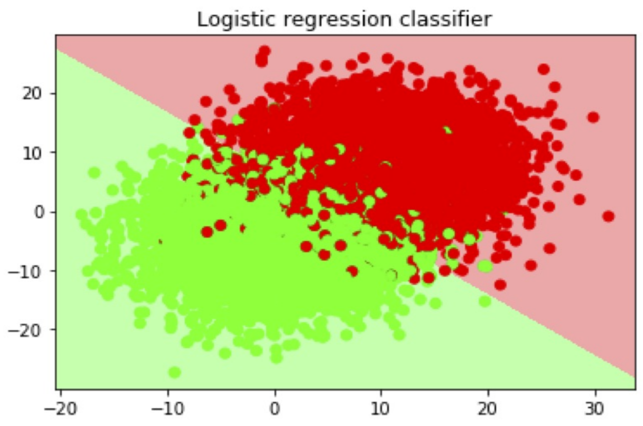
## 11 Run Newton Method

[53]:
```
# Initialize theta to zero
theta = np.zeros(shape[1]+1)
# Run gradient descent
#alpha = 0.0001,maxiter=10000
tol = 1e-6
maxiter = 10000

#using Newton's Method
start_time = time.time()
theta_ngd, cost_ngd = Newton_method(theta,xtilde,y,tol,maxiter)
print(theta_ngd)
stop_time = time.time()
print("Final Newton cost: %.3f after %d iterations in %f seconds" %␣
 ↪(cost_ngd[-1],cost_ngd.size,stop_time-start_time))
```

```
[ 1.80207019 -0.26293566 -0.33762014]
Final Newton cost: 23830.493 after 8 iterations in 600.265969 seconds
```

[128]:
```
## Plot the decision boundary.
# Begin by creating the mesh [x_min, x_max]x[y_min, y_max].
h = .02  # step size in the mesh
x_delta = (x[:, 0].max() - x[:, 0].min())*0.05 # add 5% white space to border
y_delta = (x[:, 1].max() - x[:, 1].min())*0.05
x_min, x_max = x[:, 0].min() - x_delta, x[:, 0].max() + x_delta
```

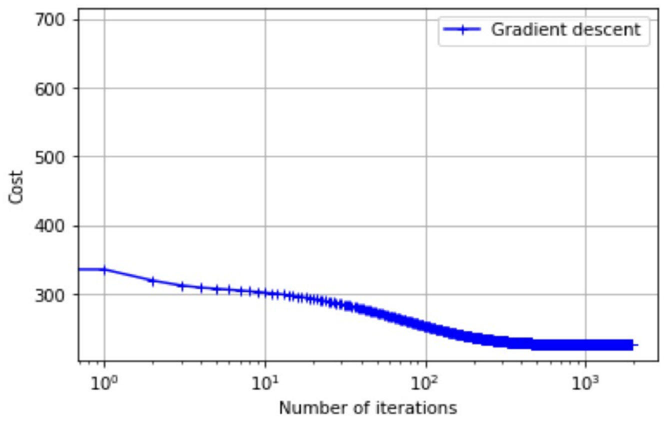
```
y_min, y_max = x[:, 1].min() - y_delta, x[:, 1].max() + y_delta
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = lr_predict(theta_gd,np.c_[xx.ravel(), yy.ravel()])

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00'])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

## Plot the training points
plt.scatter(x[:, 0], x[:, 1], c=y, cmap=cmap_bold)

## Show the plot
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("Logistic regression classifier")
plt.show()
```



[13]:
```
# plt.semilogx((cost_ngd),'rx-',label='Newton Method')
# plt.semilogx((cost_sgd),'g*-',label='SGD')
```

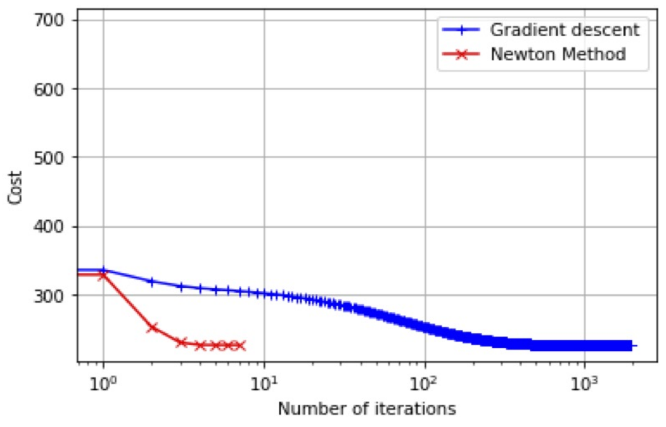
7

```
plt.semilogx((cost_gd),'b+-',label='Gradient descent')
plt.grid(True)
plt.xlabel('Number of iterations')
plt.ylabel('Cost')
plt.legend()
plt.savefig('/Users/gexueren/Desktop/6254/Code/Latex/HW#6/Q1.jpg')
```



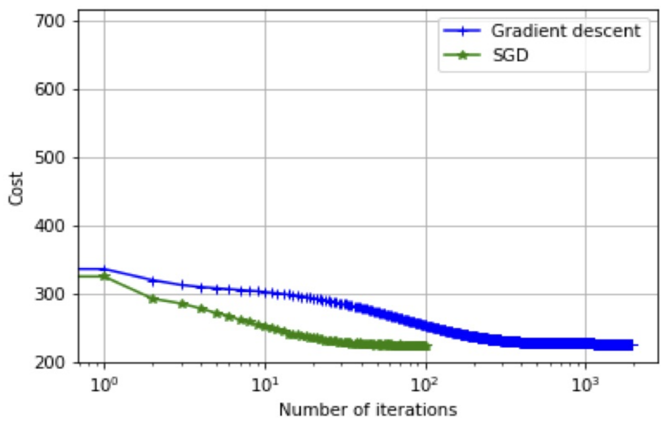
[12]:
```
plt.semilogx((cost_gd),'b+-',label='Gradient descent')
plt.semilogx((cost_ngd),'rx-',label='Newton Method')
plt.grid(True)
plt.xlabel('Number of iterations')
plt.ylabel('Cost')
plt.legend()
plt.savefig('/Users/gexueren/Desktop/6254/Code/Latex/HW#6/Q2.jpg')
```

8



```
[41]: plt.semilogx((cost_gd),'b+-',label='Gradient descent')
      plt.semilogx((cost_sgd),'g*-',label='SGD')
      plt.grid(True)
      plt.xlabel('Number of iterations')
      plt.ylabel('Cost')
      plt.legend()
      plt.savefig('/Users/gexueren/Desktop/6254/Code/Latex/HW#6/Q4.jpg')
```

9



Gradient descent
SGD
Cost
Number of iterations
700
600
500
400
300
200
$10^0$
$10^1$
$10^2$
$10^3$

```
[ ]:
```

10