Problem 1

(1)
Solving the maximum entropy distribution with constraints is essentially

$$\begin{aligned} \max &\int p(x)\ln p(x)\mathrm{d}x \\ \text{s.t.} &\int p(x)\mathrm{d}x = 1 \\ &\int xp(x)\mathrm{d}x = 0 \\ &\int x^2p(x)\mathrm{d}x = 1 \end{aligned} \tag{1}$$

Lemma(functional derivative): We first show how to take the derivative of a function. Suppose we have

$$F(p(x)) = \int p(x)f(x)\mathrm{d}x. \tag{2}$$

By substituting $p(x)$ with $p(x) + \epsilon\eta(x)$, we have

$$F(p(x) + \epsilon\eta(x)) = \int p(x)f(x)\mathrm{d}x + \epsilon\int \eta(x)f(x)\mathrm{d}x, \tag{3}$$

and thus,

$$\frac{\partial F}{\partial p(x)} = f(x). \tag{4}$$

Similarly, we have

$$\begin{aligned} G(p(x)) &= \int p(x)\ln p(x)\mathrm{d}x \\ \frac{\partial G}{\partial p(x)} &= \ln p(x) + 1. \end{aligned} \tag{5}$$

From the fact that the above optimization question is a concave maxmization one, we construct its Lagrangian function and set its derivative w.r.t. λ to zero.

$$\begin{aligned} L(x,\lambda) &= \int p(x)\ln p(x)\mathrm{d}x - \lambda_1(\int p(x)\mathrm{d}x - 1) - \lambda_2\int xp(x)\mathrm{d}x - \lambda_3(\int x^2p(x)\mathrm{d}x - 1) \\ \frac{\partial L}{\partial \lambda} &= \ln p(x) + 1 - \lambda_1 - \lambda_2 x - \lambda_3 x^2 \\ &= 0. \end{aligned} \tag{6}$$

We obtain

$$p(x) = \exp(-1 + \lambda_1 + \lambda_2 x + \lambda_3 x^2) \tag{7}$$

To solve λ, we substitute the above $p(x)$ into constraints

$$\begin{aligned} \int \exp(-1 + \lambda_1 + \lambda_2 x + \lambda_3 x^2)\mathrm{d}x &= 1 \\ \int x\exp(-1 + \lambda_1 + \lambda_2 x + \lambda_3 x^2)\mathrm{d}x &= 0 \\ \int x^2\exp(-1 + \lambda_1 + \lambda_2 x + \lambda_3 x^2)\mathrm{d}x &= 1 \end{aligned} \tag{8}$$

We notice that $\lambda = (1 - \frac{1}{2}\ln(2\pi\sigma^2),\ 0,\ \frac{1}{2\sigma^2})$ from $N(0,1)$ does satisfy above equations(and thus no need to check conditions for Lagrangian method).
This shows that $X\ N(0,1)$ is the maximum entropy distribution under such constraints.

Problem 1 continued on next page. . .

(2)
Similar to **(1)**, we have

$$\begin{aligned} p(x) &= \exp(-1 + \sum_{0}^{n} \lambda_i x^i) \\ \text{s.t.} \int p(x)\mathrm{d}x &= 1 \\ \int x^i p(x)\mathrm{d}x &= m_i \quad \text{for} 1 \leq i \leq n \end{aligned} \tag{9}$$

Notice that in some cases, such distribution does not exist and the maximum entropy only serves as an upper bound.

Problem 2

I omitted solution for (1) and (2), since they are all included in (3), below shows some of the important code.

- my implemetation in detail is in the appendix.
- code below assumes beta_0(bias) is zero
    - to cope with the problem which assumes true beta being (-2, 1) instead of (0, -2, 1)
    - if bias also has to be estimated, just change IRLS() function
    - experiment shows this assumption does not make much difference

```
#!/usr/bin/env python -W ignore::DeprecationWarning
from hw_1_2 import gen_label, IRLS, get_fisher_information
import numpy as np
np.random.seed(1234)

n = 100
X = np.random.normal(size=(n, 2))
```

Below are code used to plot

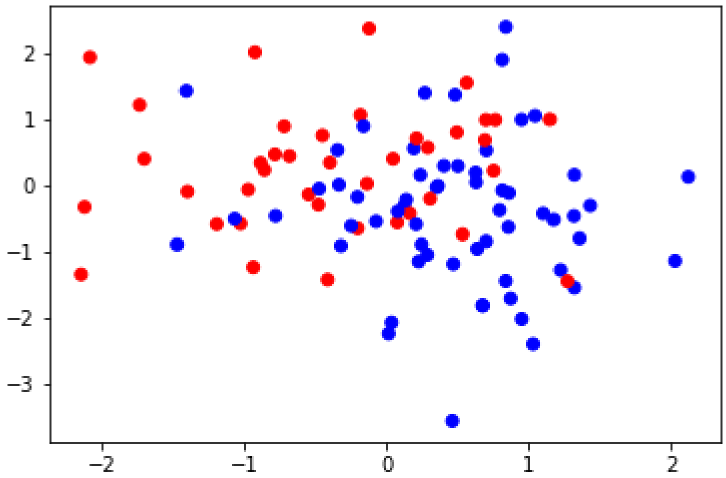
- asymptotical distribution, and
- scatter plot of estimated beta

Note that levels for (3) are [1, ..., 10] while for (4), [1, 2, ..., 1024]

```
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
def plot_new(X, weight, area, small_contour):
    func_z = multivariate_normal(mean=np.asarray((-2, 1))
                                 , cov=np.linalg.inv(get_fisher_information(X,
(-2,1))))
    xlist = np.linspace(area[0], area[1], 100)
    ylist = np.linspace(area[2], area[3], 100)
    X, Y = np.meshgrid(xlist, ylist)
    Z = np.empty(shape=X.shape)
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Z[i][j] = func_z.pdf([X[i][j], Y[i][j]])
    # note that countors in two graphs are different w.r.t. levels
    if not small_contour:
        levels = [(1 << i) for i in range(1, 11)]
        plt.contour(X, Y, Z, levels, colors='k')
        plt.scatter(weight[:, 0], weight[:, 1])
        plt.show()
    else:
        levels = [i / 5 for i in range(1, 11)]
        plt.contour(X, Y, Z, levels, colors='k')
        plt.scatter(weight[:, 0], weight[:, 1])
        plt.show()
```

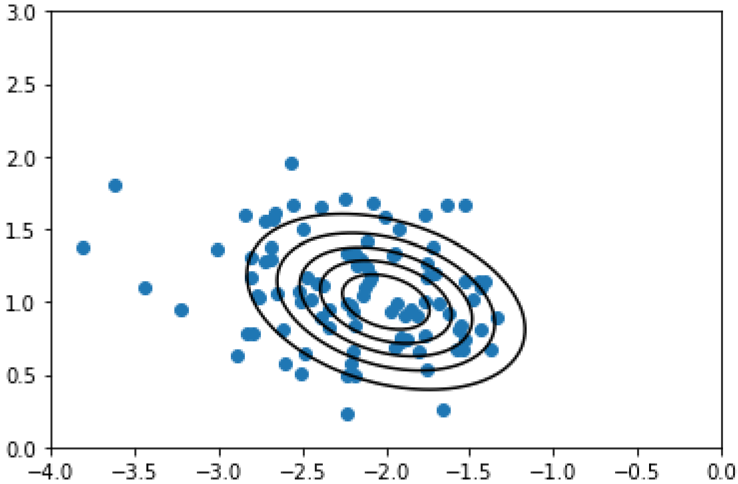
```
weight = np.empty(shape=(n, 2))
for i in range(100):
    y = gen_label(X)
    weight[i] = IRLS(X, y)
    if(i == 0):
        color = ['b' if y[i] == 0 else 'r' for i in range(0, 100)]
        plt.scatter(X[:, 0], X[:, 1], color=color)
        print("estimated weight in the first run is: ", weight[0])
```

```
estimated weight in the first run is:  [-1.37086595  0.66987777]
```



Above plot visulizes a possible observation Y from true parameter

```
plot_new(X, weight, area=[-4.,0.,0.,3.], small_contour=1)
```



The asymptotical distribution serves as a good distribution from plot above

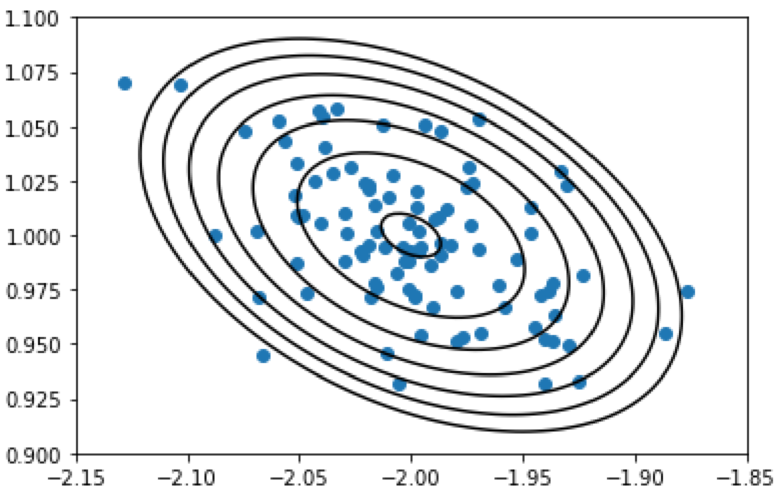
```
print("Covariance matrix form fisher information is \n"
      , np.linalg.inv(get_fisher_information(X, (-2,1))))
print("Covariance matrix from expirical data is \n"
      , np.cov(np.transpose(weight)))
```

```
Covariance matrix form fisher information is
 [[ 0.19350715 -0.0440933 ]
 [-0.0440933   0.10205911]]
Covariance matrix from expirical data is
 [[ 0.25623927 -0.04276872]
 [-0.04276872  0.12578528]]
```

, but there are still some discrapancy in estimating the covariance matrix

```
n = 10000
X = np.random.normal(size=(n, 2))
weight = np.empty(shape=(100, 2))
for i in range(100):
    y = gen_label(X)
    weight[i] = IRLS(X, y)
```

```
plot_new(X, weight, area=[-2.15, -1.85, 0.9, 1.1], small_contour=0)
```



The asymptotical distribution fits emperical data better from plot above, when # of data points is larger.

This can also be shown from the estimation of covariance matrix

```
print("Covariance matrix form fisher information is \n"
      , np.linalg.inv(get_fisher_information(X, (-2,1))))
print("Covariance matrix from expirical data is \n"
      , np.cov(np.transpose(weight)))
```

```
Covariance matrix form fisher information is
 [[ 0.00174037 -0.00053367]
 [-0.00053367  0.00096422]]
Covariance matrix from expirical data is
 [[ 0.00202756 -0.00069806]
 [-0.00069806  0.00105249]]
```

Problem 3

Similar to 2, I only estimate beta with bias = 0, in all the tests below, I use

- warmup of learning rate and learning rate decay for non-adaptive methods
- L_1 difference when visulizing losses, since likelihood function often deminishes to zero, log(L) -> NaN

```
from hw_1_3 import *
_, beta_est = nag(X, Y_gt, 0.01, d, beta_0) # to get estimated beta using NAG
```

```
NAG ended with L_1 diff as:  1.1847692899228934
Total time: 89.13268494606018 total steps: 10001
```

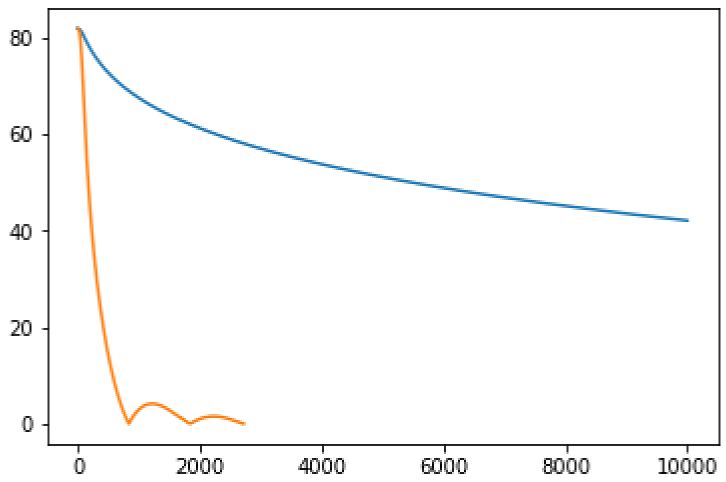
```
gd_loss = gd(X, Y_gt, 0.01, d, beta_est)
```

```
vanilla GD ended with L_1 diff as:  42.08500024090012
Total time: 86.5514760017395 total steps: 10001
```

```
nag_loss, _ = nag(X, Y_gt, 0.01, d, beta_est)
```

```
NAG ended with L_1 diff as:  0.0009671487964988679
Total time: 24.532387495040894 total steps: 2722
```

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(gd_loss)
plt.plot(nag_loss)
plt.show()
```



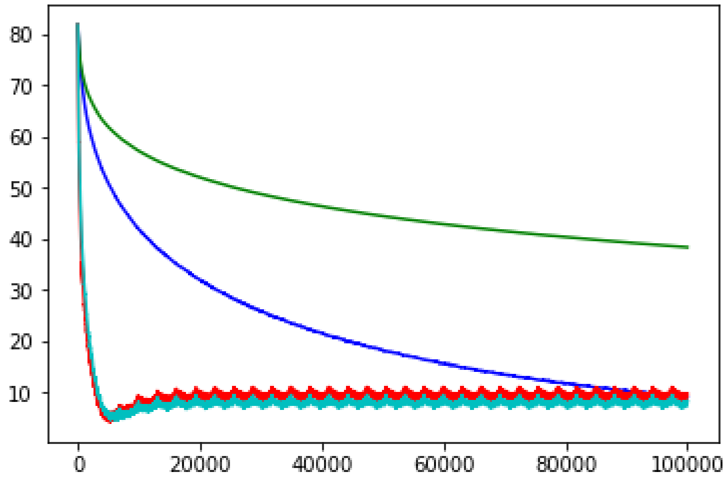
We see from above plot that

- NAG converges faster

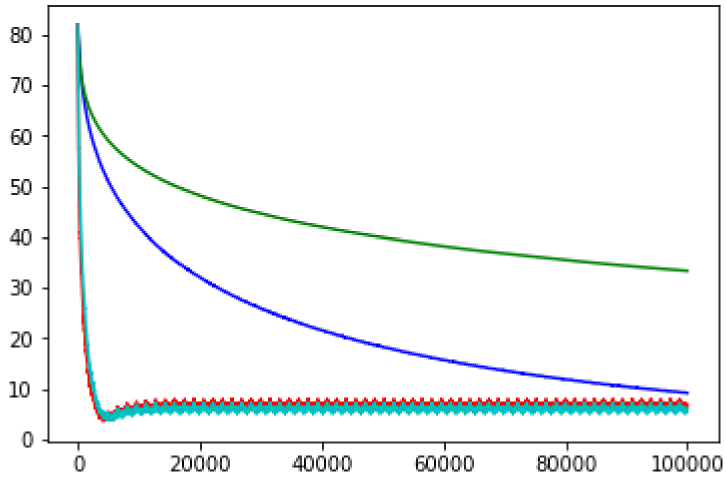
- NAG yields better results at termination time
- L_1 loss from NAG shows disturbance
    - such disturbance goes smaller as time goes

```
import matplotlib.pyplot as plt
%matplotlib inline
for batch_size in [32, 64, 128]:
    sgd_loss = sgd(X, Y_gt, 0.01, d, batch_size, beta_est)
    adagrad_loss = adagrad(X, Y_gt, 0.01, d, 1e-8, batch_size, beta_est)
    rmsprop_loss = rmsprop(X, Y_gt, 0.01, d, 1e-8, batch_size, beta_est)
    adam_loss = adam(X, Y_gt, 0.01, d, 0.9, 0.999, 1e-8, batch_size, beta_est)
    plt.clf()
    plt.plot(sgd_loss, color='b')
    plt.plot(adagrad_loss, color='g')
    plt.plot(rmsprop_loss, color='r')
    plt.plot(adam_loss, color='c')
    plt.show()
```

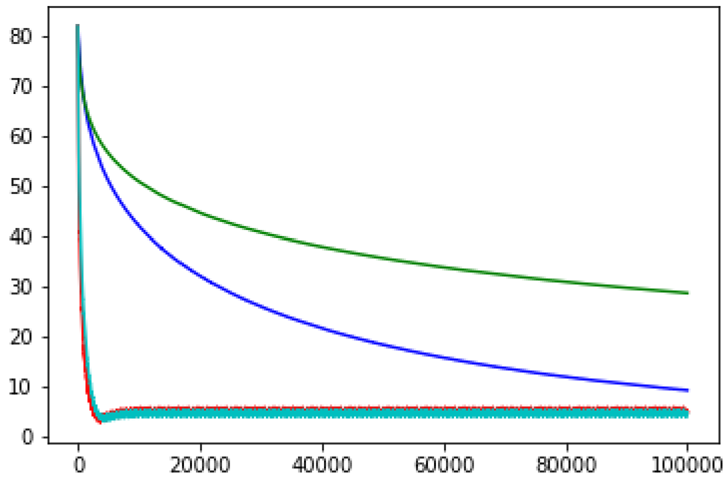
```
SGD ended with L_1 diff as:  9.089506974905682
Total time: 2.2679333686828613
AdaGrad ended with L_1 diff as:  38.38244586420935
Total time: 2.4318063259124756
RMSprop ended with L_1 diff as:  9.675100723387237
Total time: 2.524564504623413
Adam ended with L_1 diff as:  8.363492159318252
Total time: 3.2511160373687744
```



```
SGD ended with L_1 diff as:  9.119980024670259
Total time: 3.530276298522949
AdaGrad ended with L_1 diff as:  33.32072177295851
Total time: 3.88824725151062
RMSprop ended with L_1 diff as:  6.808032554722099
Total time: 4.166238307952881
Adam ended with L_1 diff as:  6.135580491122062
Total time: 5.348047733306885
```




```
SGD ended with L_1 diff as:  9.130085547588333
Total time: 4.361374616622925
AdaGrad ended with L_1 diff as:  28.583852256471978
Total time: 4.767404556274414
RMSprop ended with L_1 diff as:  4.8515965149099864
Total time: 4.812380075454712
Adam ended with L_1 diff as:  4.491760740311056
Total time: 5.9629807472229
```



We conclude from above plot that

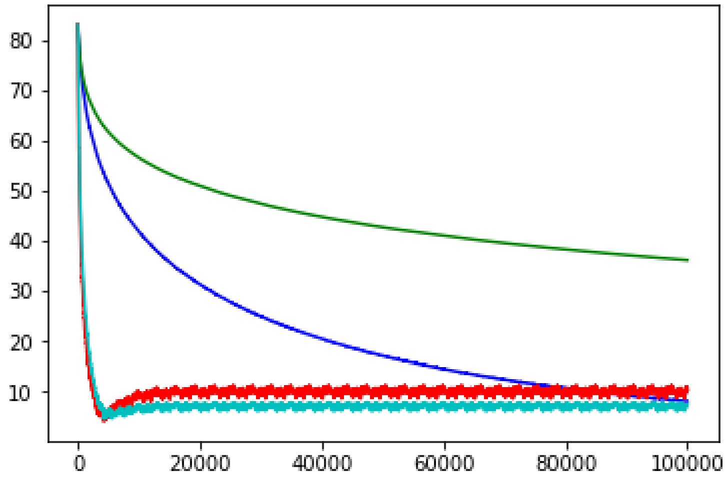
- for convergence speed: AdaGrad < SGD < RMSprop = Adam
  - only in this specific setting
  - Adam is slightly better than RMSprop
- AdaGrad does suffer from gradient vanishing
- generally speaking, all algorithms performs better under larger batch size
  - except for SGD, which hits the limit of 9.1x
- smaller batch size causes disturbance in L_1 loss after convergence of RMSprop and Adam
  - with batch size grows, such disturbance gets smaller
- the gap between final result of SGD and (Adam or RMSprop) goes larger for larger batch size

```
from hw_1_3 import *
np.random.seed(1234)
sparse_rate = 0.3
M = np.random.uniform(size=(n,d)) < sparse_rate
X[M] = 0.
Y_gt = gen_label(X, beta_0)
_, beta_est = nag(X, Y_gt, 0.01, d, beta_0)
```

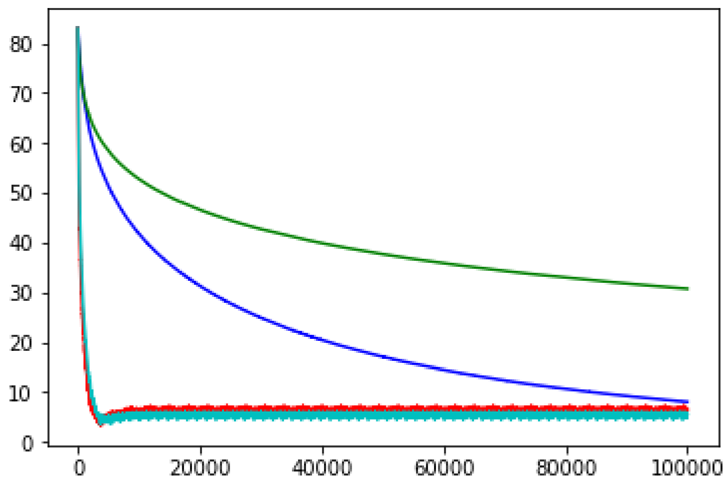
```
NAG ended with L_1 diff as:  1.8321787783273666
Total time: 91.06195950508118 total steps: 10001
```

```
import matplotlib.pyplot as plt
%matplotlib inline
for batch_size in [32, 64, 128]:
    sgd_loss = sgd(X, Y_gt, 0.01, d, batch_size, beta_est)
    adagrad_loss = adagrad(X, Y_gt, 0.01, d, 1e-8, batch_size, beta_est)
    rmsprop_loss = rmsprop(X, Y_gt, 0.01, d, 1e-8, batch_size, beta_est)
    adam_loss = adam(X, Y_gt, 0.01, d, 0.9, 0.999, 1e-8, batch_size, beta_est)
    plt.clf()
    plt.plot(sgd_loss, color='b')
    plt.plot(adagrad_loss, color='g')
    plt.plot(rmsprop_loss, color='r')
    plt.plot(adam_loss, color='c')
    plt.show()
```

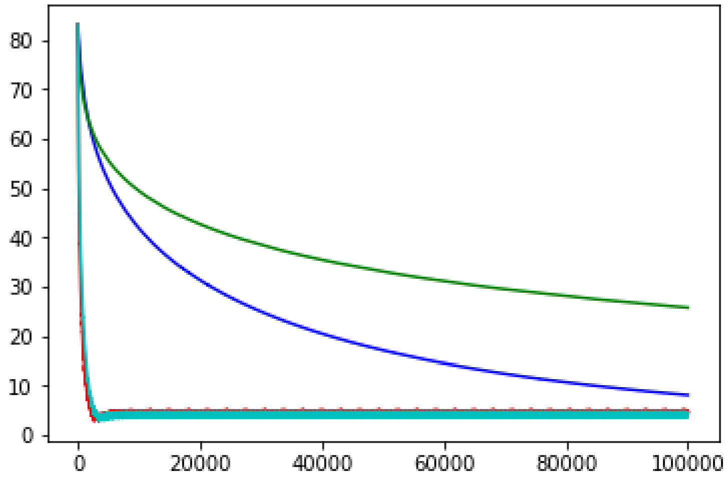
```
SGD ended with L_1 diff as:  8.10032203795655
Total time: 2.1782093048095703
AdaGrad ended with L_1 diff as:  36.15153057116418
Total time: 2.404568672180176
RMSprop ended with L_1 diff as:  10.432561255866384
Total time: 2.516232967376709
Adam ended with L_1 diff as:  7.549515797378077
Total time: 3.3620452880859375
```




```
SGD ended with L_1 diff as:  8.09541736789895
Total time: 3.456721067428589
AdaGrad ended with L_1 diff as:  30.749326373660313
Total time: 3.843719482421875
RMSprop ended with L_1 diff as:  6.578302521043001
Total time: 3.987337589263916
Adam ended with L_1 diff as:  5.615495204777702
Total time: 4.9308106899261475
```




```
SGD ended with L_1 diff as:  8.094199734747038
Total time: 4.287536144256592
AdaGrad ended with L_1 diff as:  25.73220545695067
Total time: 4.7193779945373535
RMSprop ended with L_1 diff as:  4.727903580408079
Total time: 4.527889251708984
Adam ended with L_1 diff as:  4.183012679913681
Total time: 5.861323118209839
```



We can also derive conclusions similar to (2), besides

- gap between Adam and RMSprop bacomes larger in smaller batch_size
- L_1 diff at convergence remains the same for Adam and RMSprop regardless of sparsity
- the limit for SGD becomes smaller (8.1 v.s. 9.1)

```python
import numpy as np
import numpy.ma as ma
from scipy.interpolate import griddata
from numpy.random import uniform, seed
from matplotlib import cm

def gen_label(X):
   return np.random.binomial(1, 1.0 / (1.0 + np.exp(-np.matmul(X, (-2.0, 1.0)))))

def IRLS(X, y):
    weight = np.zeros(shape=(2))
    # bias = np.log(np.mean(y) / (1.0 - np.mean(y)))
    bias = 0
    #print(bias)
    threshold = 1e-6
    change = 1e6
    while(change >= threshold):
        ita = np.matmul(X, weight) + bias
        #print(ita)
        miu = 1.0 / (1.0 + np.exp(-ita))
        #print(miu)
        s = np.multiply(miu, 1.0 - miu)
        z = ita + (y - miu) / s
        #print(s, z)
        s = np.diag(s).copy()
        XTSX_inv = np.linalg.inv(np.matmul(np.matmul(np.transpose(X), s), X))
        XTSz = np.matmul(np.matmul(np.transpose(X), s), z)
        weight_new = np.matmul(XTSX_inv, XTSz)
        change = np.linalg.norm(weight_new - weight)
        weight = weight_new
    return weight

def get_fisher_information(X, beta):
    prob = 1.0 / (1.0 + np.exp(-np.matmul(X, (-2.0, 1.0))))
    w = np.multiply(prob, 1 - prob)
    w = np.diag(w).copy()
    return np.matmul(np.matmul(np.transpose(X), w), X)
```

```python
import numpy as np
import time
np.random.seed(1234)

n, d = 100000, 100
X = np.random.normal(size=(n,d))
beta_0 = np.random.normal(size=d)

def get_prob(X, beta):
    return 1.0 / (1.0 + np.exp(- np.matmul(X, beta)))

def gen_label(X, beta):
    return np.random.binomial(1, get_prob(X, beta))

Y_gt = gen_label(X, beta_0)

def get_gradient(X, Y_gt, Y_pred, batch_size):
    grad_beta = np.dot(np.transpose(X), Y_gt - Y_pred)
    return grad_beta / batch_size

def get_likelihood(X, Y_gt, prob):
    res = 1.0
    for i in range(X.shape[0]):
        if Y_gt[i] == 1.0:
            res *= prob[i]
        else:
            res *= 1.0 - prob[i]
    return res

def lr_scheduler(lr_init, step, decay):
    warm_up_step = 100.0
    lr_decay = 1e-6
    if step <= warm_up_step:
        return lr_init * step / warm_up_step
    if not decay:
        return lr_init
    return np.power(1 - lr_decay, step - 100) * lr_init

def get_batch(X, Y_gt, start_ele, batch_size):
    if start_ele + batch_size >= n:
        X_batch = np.concatenate((X[start_ele: ], X[ :start_ele + batch_size - n]))
        Y_gt_batch = np.concatenate((Y_gt[start_ele: ], Y_gt[ :start_ele + batch_size - n]))
    else:
        X_batch = X[start_ele: (start_ele + batch_size)]
        Y_gt_batch = Y_gt[start_ele: (start_ele + batch_size)]
    return X_batch, Y_gt_batch

# batch size = n
def gd(X, Y_gt, lr_init, d, beta_est):
    start = time.time()
    loss = []
    beta = np.zeros(shape=d)
    step = 1
    while True:
        Y_pred = get_prob(X, beta)
        grad_beta = get_gradient(X, Y_gt, Y_pred, X.shape[0])
        lr = lr_scheduler(lr_init, step, 1)
        beta += lr * grad_beta
        step += 1
        loss_this = np.sum(np.absolute(beta_est - beta))
        loss.append(loss_this)
        if loss_this < 1e-3 or step > 1e4:
            print("vanilla GD ended with L_1 diff as: ", np.sum(np.absolute(beta_est - beta)))
            print("Total time:", time.time() - start, "total steps:", step)
            break;
    return loss

def nag(X, Y_gt, lr_init, d, beta_est):
    start = time.time()
    loss = []
    beta = np.zeros(shape=d)
    step = 1
```

```
    beta_tmp = beta
    while True:
        y_beta = beta + ((step - 2.0) / (step + 1.0)) * (beta - beta_tmp)
        Y_pred = get_prob(X, y_beta)
        grad_beta = get_gradient(X, Y_gt, Y_pred, X.shape[0])
        lr =  lr_scheduler(lr_init, step, 1)
        beta_tmp = beta
        beta = y_beta + lr * grad_beta
        step += 1
        loss_this = np.sum(np.absolute(beta_est - beta))
        loss.append(loss_this)
        if loss_this < 1e-3 or step > 1e4:
            print("NAG ended with L_1 diff as: ", np.sum(np.absolute(beta_est - beta)))
            print("Total time:", time.time() - start, "total steps:", step)
            break;
    return loss, beta

def adagrad(X, Y_gt, lr_init, d, eps, batch_size, beta_est):
    start = time.time()
    loss = []
    beta = np.zeros(shape=d)
    step = 1
    g_beta = np.zeros(shape=d)
    while True:
        start_ele = ((step - 1) * batch_size) % n
        X_batch, Y_gt_batch = get_batch(X, Y_gt, start_ele, batch_size)
        Y_pred = get_prob(X_batch, beta)
        grad_beta = get_gradient(X_batch, Y_gt_batch, Y_pred, batch_size)
        g_beta += np.square(grad_beta)
        lr =  lr_scheduler(lr_init, step, 0)
        beta += lr * np.multiply((1.0 / np.sqrt(g_beta + eps)), grad_beta)
        step += 1
        loss.append(np.sum(np.absolute(beta_est - beta)))
        if step > 1e5:
            print("AdaGrad ended with L_1 diff as: ", np.sum(np.absolute(beta_est - beta)))
            print("Total time:", time.time() - start)
            break;
    return loss

def rmsprop(X, Y_gt, lr_init, d, eps, batch_size,beta_est):
    start = time.time()
    loss = []
    beta = np.zeros(shape=d)
    step = 1
    g_beta = np.zeros(shape=d)
    while True:
        start_ele = ((step - 1) * batch_size) % n
        X_batch, Y_gt_batch = get_batch(X, Y_gt, start_ele, batch_size)
        Y_pred = get_prob(X_batch, beta)
        grad_beta = get_gradient(X_batch, Y_gt_batch, Y_pred, batch_size)
        g_beta = 0.9 * g_beta + 0.1 * np.square(grad_beta)
        lr =  lr_scheduler(lr_init, step, 0)
        beta += lr * np.multiply((1.0 / np.sqrt(g_beta + eps)), grad_beta)
        step += 1
        loss.append(np.sum(np.absolute(beta_est - beta)))
        if step > 1e5:
            print("RMSprop ended with L_1 diff as: ", np.sum(np.absolute(beta_est - beta)))
            print("Total time:", time.time() - start)
            break;
    return loss

def sgd(X, Y_gt, lr_init, d, batch_size, beta_est):
    start = time.time()
    loss = []
    beta = np.zeros(shape=d)
    step = 1
    while True:
        start_ele = ((step - 1) * batch_size) % n
        X_batch, Y_gt_batch = get_batch(X, Y_gt, start_ele, batch_size)
        Y_pred = get_prob(X_batch, beta)
        grad_beta = get_gradient(X_batch, Y_gt_batch, Y_pred, batch_size)
        lr =  lr_scheduler(lr_init, step, 1)
        beta += lr * grad_beta
        step += 1
```

```python
        loss.append(np.sum(np.absolute(beta_est - beta)))
        if step > 1e5:
            print("SGD ended with L_1 diff as: ", np.sum(np.absolute(beta_est - beta)))
            print("Total time:", time.time() - start)
            break;
    return loss

def adam(X, Y_gt, lr_init, d, b_1, b_2, eps, batch_size, beta_est):
    start = time.time()
    loss = []
    beta = np.zeros(shape=d)
    step = 1
    m_beta = np.zeros(shape=d)
    v_beta = np.zeros(shape=d)
    while True:
        start_ele = ((step - 1) * batch_size) % n
        X_batch, Y_gt_batch = get_batch(X, Y_gt, start_ele, batch_size)
        Y_pred = get_prob(X_batch, beta)
        grad_beta = get_gradient(X_batch, Y_gt_batch, Y_pred, batch_size)
        lr =  lr_scheduler(lr_init, step, 0)
        m_beta = b_1 * m_beta + (1 - b_1) * grad_beta
        v_beta = b_2 * v_beta + (1 - b_2) * np.square(grad_beta)
        beta += lr * (m_beta / (1.0 - np.power(b_1, step)))
                / np.sqrt(eps + v_beta / (1.0 - np.power(b_2, step)))
        step += 1
        loss.append(np.sum(np.absolute(beta_est - beta)))
        if step > 1e5:
            print("Adam ended with L_1 diff as: ", np.sum(np.absolute(beta_est - beta)))
            print("Total time:", time.time() - start)
            break;
    return loss
```