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

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

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

In [1]:

```
#!/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

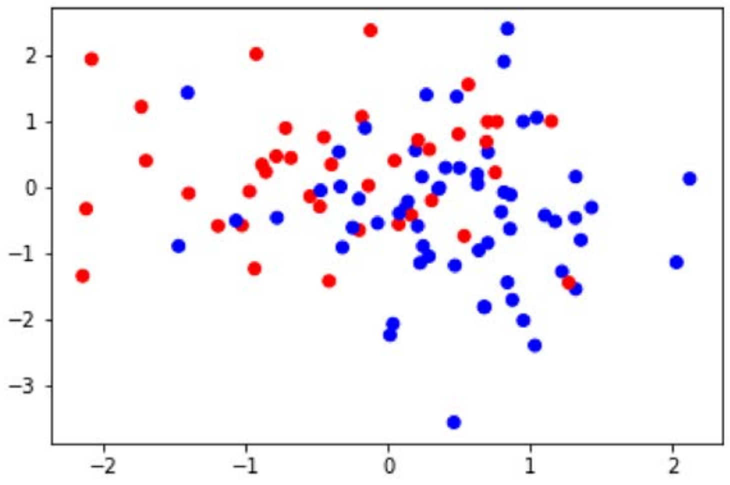
Note that levels for (3) are [1, ..., 10] while for (4), [1, 2, ..., 1024]

In [2]:

```
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()
```

In [3]:

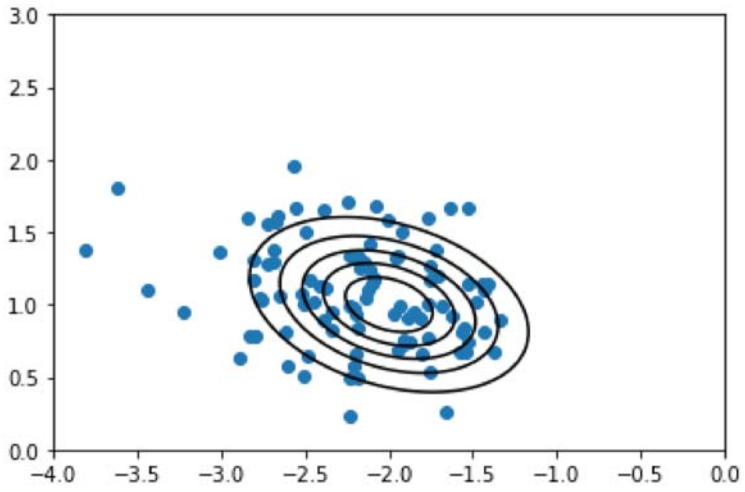
```
weight = np.empty(shape=(n, 2))
for i in range(100):
    y = gen_label(X)
    if(i == 0):
        color = ['b' if y[i] == 0 else 'r' for i in range(0, 100)]
        plt.scatter(X[:, 0], X[:, 1], color=color)
    weight[i] = IRLS(X, y)
```



Above plot visulizes a possible observation Y from true parameter

In [4]:

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



The asymptotical distribution serves as a good distribution from plot above

In [5]:

```
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]]
```

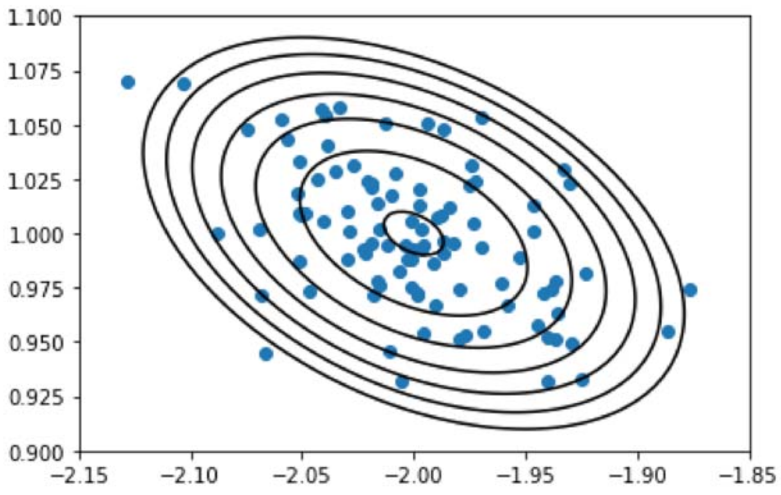
, but there are still some discrapancy in estimating the covariance matrix

In [6]:

```
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)
```

In [7]:

```
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

In [8]:

```
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]]
```

In [ ]:

```

```

Problem 3

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

- warmup of learning rate
- learning rate decay for non-adaptive methods

In [3]:

```
from hw_1_3 import *
_, beta_est = nag(X, Y_gt, 0.01, d, beta_0)
```

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

In [4]:

```
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
```

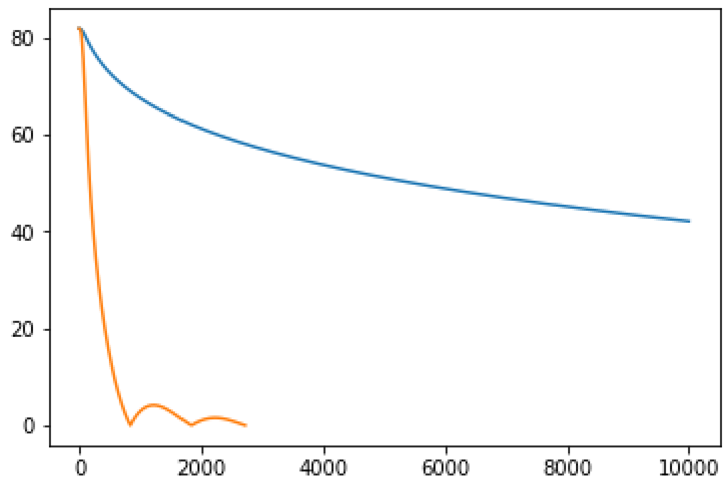
In [11]:

```
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
```

In [12]:

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



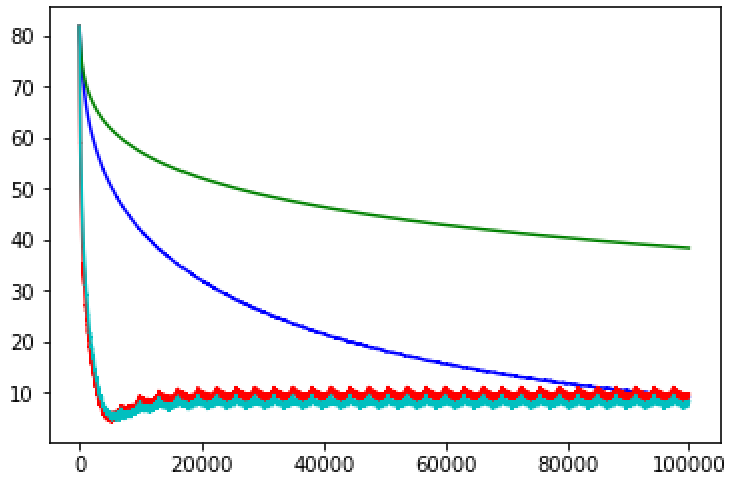
We see from above plot that

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

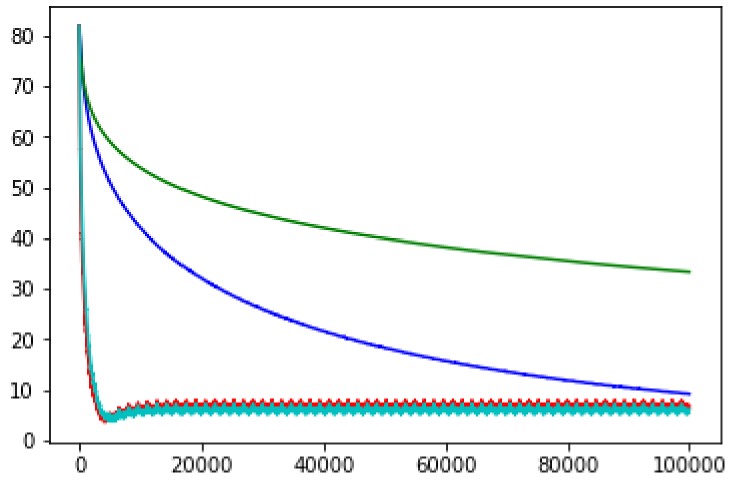
In [13]:

```
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()
```

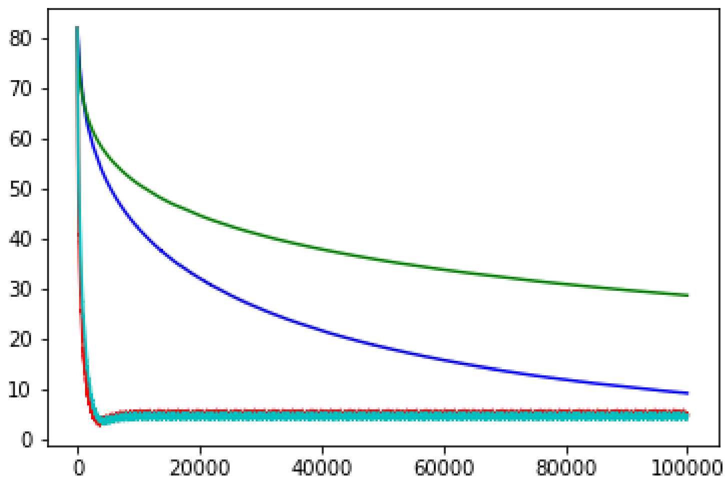
```
SGD ended with L_1 diff as:  9.089506974905682
Total time: 2.2679333686828613
AdaGrad ended with L_1 diff as:  38.38244586420935
Total time: 2.4318063259124756
RMSprop ended with L_1 diff as:  9.675100723387237
Total time: 2.524564504623413
Adam ended with L_1 diff as:  8.363492159318252
Total time: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

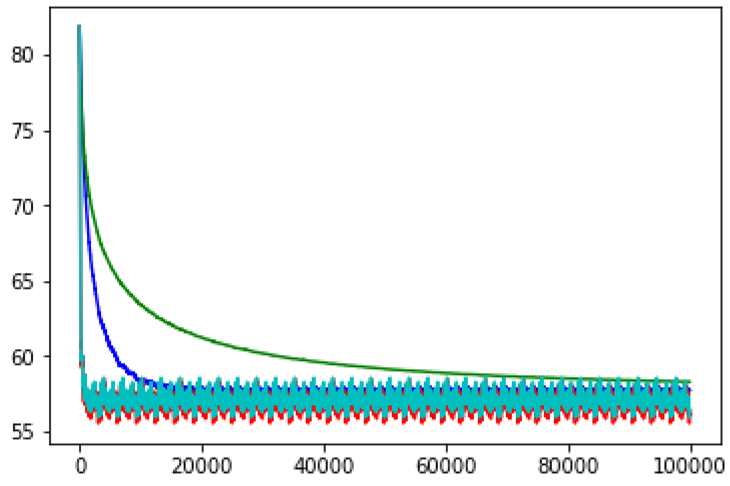
In [14]:

```
np.random.seed(1234)
sparse_rate = 0.3
M = np.random.uniform(size=(n,d)) < sparse_rate
X[M] = 0.
```

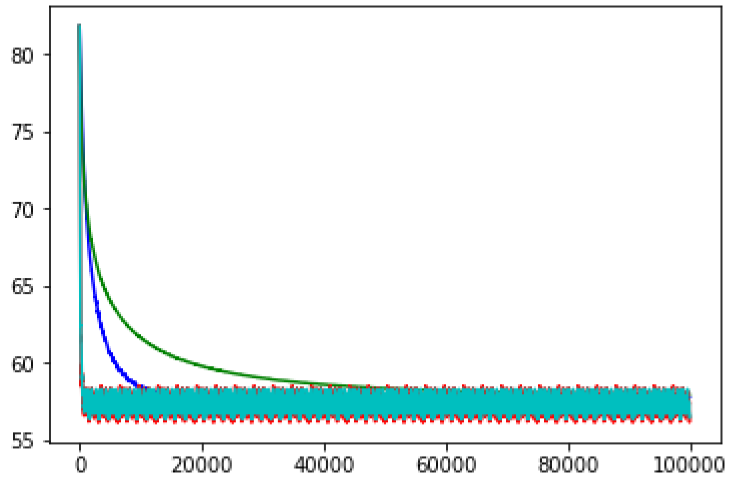
In [15]:

```
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()
```

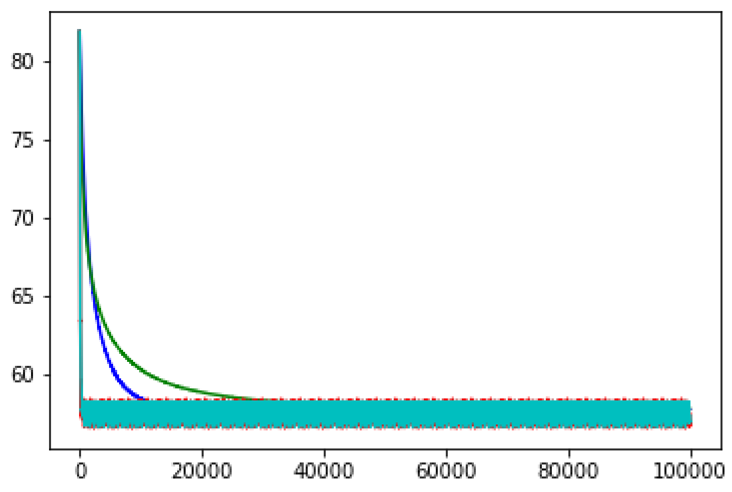
```
SGD ended with L_1 diff as:  57.74325215623408
Total time: 2.2330245971679688
AdaGrad ended with L_1 diff as:  58.30687856300753
Total time: 2.4275107383728027
RMSprop ended with L_1 diff as:  56.09351403571297
Total time: 2.569098472595215
Adam ended with L_1 diff as:  56.477997851590615
Total time: 3.358017683029175
```



```
SGD ended with L_1 diff as:  57.78443471597452
Total time: 3.466726303100586
AdaGrad ended with L_1 diff as:  57.95036400912456
Total time: 3.9653944969177246
RMSprop ended with L_1 diff as:  56.533170866174935
Total time: 4.007282972335815
Adam ended with L_1 diff as:  56.664281028993386
Total time: 5.384597063064575
```



```
SGD ended with L_1 diff as:  57.797680800123366
Total time: 4.19179105758667
AdaGrad ended with L_1 diff as:  57.83287014075975
Total time: 4.435136318206787
RMSprop ended with L_1 diff as:  56.886631457256904
Total time: 4.495975494384766
Adam ended with L_1 diff as:  56.93494475625021
Total time: 6.0927064418792725
```



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

- variation in loss gets larger in sparse cases
- all algorithm hits a limit sooner or later
    - such limit is irrelivant to batch size, thus an inherent limit in this problem

```
In [ ]:
```