Final Project
Convex Clustering

220041040 Qiuyan YUAN

221041041 Yining ZHONG

117030035 Xinrong JIANG

116020277 Sixuan XIANG

220041032 Hao HAN

2021.12.28

—

MDS6106
Introduction to Optimization

—

Abstract

Convex clustering is an interesting topic worthy of study. It can solve some of the problems of traditional unsupervised clustering methods, such as poor model effect, the need to manually set some important parameters, and so on. In this project, by setting reasonable convex optimization goals, we can make each sample point not far from the cluster center to which it belongs, and at the same time make it possible to distinguish between different cluster centers. We try to choose different parameters to define our problem, use AGM, Newton CG, BFGS, L-BFGS, and other different methods to optimize this problem, and the difference between using different matrix norms for the penalty term of the loss function, using Mini batch SGD and other methods is studied. Some meaningful and interesting observation results are obtained.

1. Introduction

This report is going to investigate different optimization models and utilize minimization methodologies to solve convex clustering problems for large-scale unsupervised learning.

In some pattern recognition problems, the training data consists of a set of input vectors X without any corresponding target values. The goal in such unsupervised learning problems may be to discover groups of similar examples within the data, where it is called clustering.

Let $\mathbb{R}^{d\times n} \ni A = (a_1, a_2, \dots, a_n)$ be a given data matrix with n observations and d features.

The convex clustering model for these n observations solves the following convex optimization problem:

$$\min_{X\in\mathbb{R}^{d\times n}} \frac{1}{2}\sum_{i=1}^{n} \|x_i - a_i\|^2 + \lambda \sum_{i=1}^{n} \sum_{j=i+1}^{n} \|x_i - x_j\|_p \qquad (1)$$

where $\lambda > 0$ is a regularization parameter and $\|\cdot\|_p$ denotes the p-norm. Here, $\|\cdot\|$ denotes the standard Euclidean norm (as usual). The p-norm above with $p \geq 1$ ensures the convexity of the model. Typically, p is chosen to be 1, 2, or ∞. (Since this model is strongly convex, the optimal solution for a given positive λ is unique.)

After solving (1) and obtaining the optimal solution $X^* = (x_1^*, \dots, x_n^*)$, we assign ai and aj to the same cluster if and only $\|x_i^* - x_j^*\| \leq \varepsilon$ for a given tolerance $\varepsilon > 0$.

After several group discussions, we finish the final project together, and the details of the work division are as follows.

Participation	name
Data preparation, AGM, and Weighted AGM	Qiuyan YUAN
Newton-CG, and Weighted Newton-CG	Yining ZHONG
BFGS, and L-BFGS	Sixuan XIANG
SGD, and Mini-batch SGD	Hao HAN
Norm regularization	Xinrong JIANG

Table 1: Individual work

2. Data Preparation

- Data preparation
  a) Self-generated Data Sets

We generated seven 2-dimension datasets with different shapes and sizes for the project. Figure 1 displays the distribution of them on a two-dimensional plane.

Dataset	n	classes	description
200_random_4	200	4	4 random center points, the spread of the data points is controlled by the selected variance.
2000_random_4	2000	4	4 random center points, the spread of the data points is controlled by the selected variance.

200_moon_2	200	2	Moon-shaped data sets that generated by sklearn.datasets.make_moons().
200_circle_2	200	2	Ring-shaped data sets that generated by sklearn.datasets.make_moons().
200_dot_3	200	3	3 random center points, the spread of the data points is controlled by the same variance.
2000_dot_3	2000	3	3 random center points, the spread of the data points is controlled by the same variance.
200_dot_6	200	6	6 random center points, the spread of the data points is controlled by the same variance.

Table 2: Self-generated Data Sets

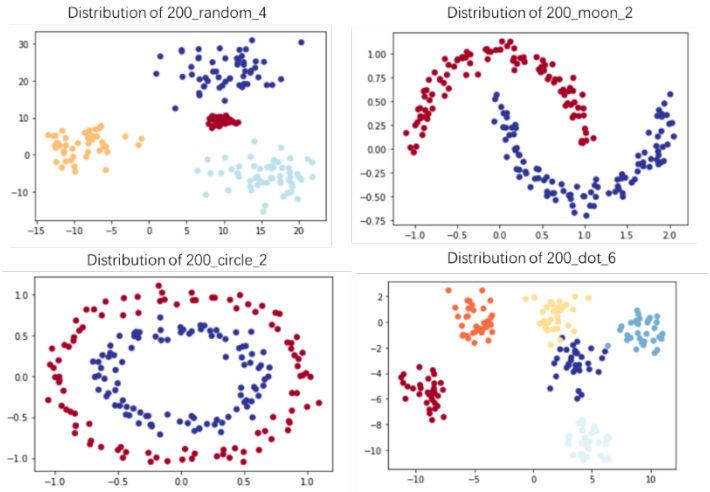


Figure 1: Distribution of 4 types of artificial data sets

b) Real Data Sets

All the provided datasets are used for the project. The basic information of the datasets is summarized in Table x. However, we do not have enough computing resources to run the algorithm under the large data set. We mainly use the wine data set.

Dataset	d	n	classes
wine	13	178	3
vowel	10	528	11
segment	19	2310	7
mint	784	60000	10

Table 3: Real Data Sets

3. Experiment

- AGM
  - Introduction to Accelerated Gradient Method

The Accelerated Gradient Method is based on acceleration techniques (extrapolation step) and Momentum. In this project, we apply the Accelerated Gradient Method (AGM) with fixed step size and basic extrapolation strategy.

Algorithm1. Accelerated Gradient Method (AGM)

Begin

Initialization: Choose a point $x_0 \in R^n$ and set $x^{-1} = x^0$

for k = 0, 1, 2, … **do**

select an extrapolation parameter β_k:

$$\beta_k = \frac{\theta_k(1-\theta_{k-1})}{\theta_{k-1}} = \frac{\theta_k}{\theta_{k-1}} - \theta_k \text{ and } \theta_{-1} = \theta_0 = 1, \theta_k = t_k^{-1}$$

$$t_k = \frac{1+\sqrt{1+4t_{k-1}^2}}{2}, \beta_k = \frac{t_{k-1}-1}{t_k}, t_{-1} = t_0 = 1$$

compute the step $y^{k+1} = x^k + \beta_k(x^k - x^{k-1})$

select step size $\alpha_k = 1/L, L = 1 + n\lambda/\delta$

set $x^{k+1} = y^{k+1} - \alpha_k \nabla f(y^{k+1})$.

if $\|\nabla f(x^{k+1})\| \leq tol$ **do**

return x^{k+1}

end

By applying the accelerated gradient method to test datasets, we can find that for the datasets with large dimensions and sample size, AGM can't converge with the low tolerance level. The reason is that only based on the first-order information, our step update is of low efficiency. Even the AGM methods have relaxed the Armijo condition, the decrease of the objective function value can't be guaranteed.

■ Experiment

a) Clustering results on different datasets

1) Cluster Result

Six artificial data sets are used to test the AGM algorithm. They are 200_random_4, 200_dot_3, 200_dot_6, 2000_random_4, 200_moon_2, 200_circle_2. According to the experimental results, the AGM algorithm performs well on small, scattered data sets, such as 200_random_4 and 200_dot_3. In a small data set with 6 clusters, 200_dot_6, the clustering results are reasonable but not completely classified into 6 classes. The left image of the figure below represents the original distribution of the artificial data set, and the right image represents the clustering result of the AGM.

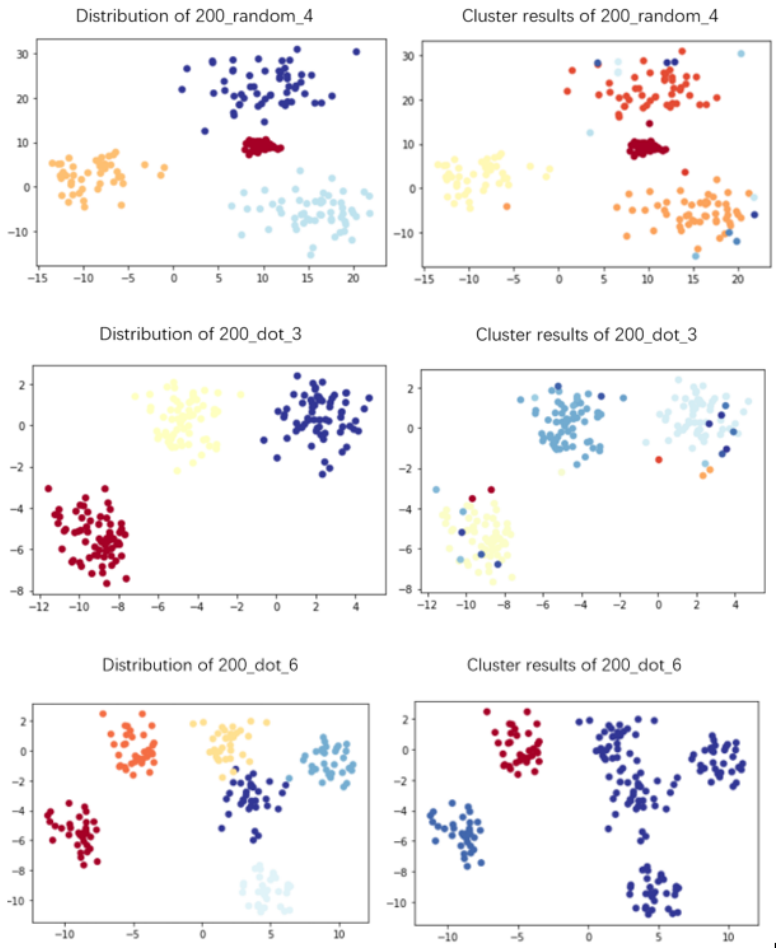


Figure 2: original distribution and cluster results

However, on large data sets, such as the 2000_random_4 data set which has 2-dimension, 2000 data points, and 4 clusters, the AGM algorithm cannot converge. After Iteration reaches 30000 and the running time exceeds 10 hours, the norm gradient is still greater than 1000, and effective clustering cannot be performed.

Figure 3: original distribution and cluster results of 2000_random_4

On data sets with special shapes, such as moon-shaped and circular data sets, although AGM converges, the clustering effect is not significant, and the two types of data cannot be distinguished.

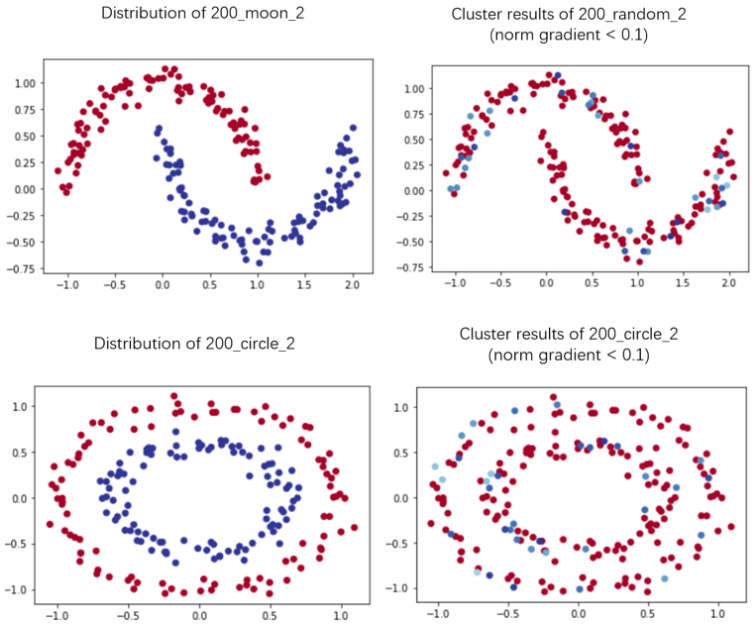


Figure 4: original distribution and cluster results

2) CPU Time and Convergence rate

AGM used on large datasets such as 2000_random_4 and circle shape datasets such as 200_circle_2 cannot converge even though the iteration reaches 30000. The CPU time for different data with 200 data points is around 10 minutes with AGM. It is too slow to apply to real data sets.

Dataset Name	Iteration Num	CPU Time	Norm Gradient
200_random_4	13486	358.22s	< 0.1
200_dot_3	13473	752.88s	< 0.1
200_dot_6	11530	635.88s	< 0.1
2000_random_4	30000	62955.85s	1171
200_moon_2	26651	587.12s	< 0.1
200_circle_2	30000	641.45s	1.04

Table 4: performance of AGM on different datasets

- Newton-CG
  - Introduction to Newton-CG Method

The Newton-CG Method is a truncated Newton method, which is to approximate the solution of the Newton equation to get the idea of the inspection direction because solving the equation is time-consuming. We require an approximate solution to be found with a certain number of iterations.

In this method, we choose CG Method to approximate the Newton function $\nabla^2 f_k p_k + \nabla f_k = 0$

Algorithm2. Newton-CG Method

Given initial point x_0

for k = 0, 1, 2, … **do**

Define tolerance $\epsilon_k = \min(0.5, \sqrt{||\nabla f_k||})||\nabla f_k||$;

Set $z_0 = 0, r_0 = \nabla f_k, d_0 = -r_0 = -\nabla f_k$;

for $j = 0,\ 1,\ 2\ \ldots$

if $d_j^T B_k d_j \leq 0$

if $j = 0$

return $p_k = -\nabla f_k;$

else

return $p_k = z_j;$

Set $\alpha_j = r_j^T r_j / d_j^T B_k d_j;$

Set $z_{j+1} = z_j + \alpha_j d_j;$

Set $r_{j+1} = r_j + \alpha_j B_k d_j;$

if $\|r_{j+1}\| < \epsilon_k$

return $p_k = z_{j+1};$

Set $\beta_{j+1} = r_{j+1}^T r_{j+1} / r_j^T r_j;$

Set $d_{j+1} = -r_{j+1} + \beta_{j+1} d_j;$

end (for)

Set $x_{k+1} = x_k + \alpha_k p_k,$ where α_k satisfies the Wolfe, Goldstein, or Armijo backtracking conditions (using $\alpha_k = 1$ if possible);

end

- Calculate the gradient and hessian

$$\min_{X \in \mathbb{R}^{d \times n}} f_{\text{clust}}(X) := \frac{1}{2} \sum_{i=1}^{n} \|x_i - a_i\|^2 + \lambda \sum_{i=1}^{n} \sum_{j=i+1}^{n} \varphi_{\text{hub}}(x_i - x_j)$$

We can use a vector to represent X, so $X \in R^{dn}, x_i \in R^d, i = 1,2,\ldots,n,\ X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$

Then the gradient of x_i is,

$$\frac{\partial f}{\partial x_i} = x_i - a_i + \lambda \sum_{j=1}^{n} \varphi'_{hub}(x_i - x_j),\ \varphi'_{hub}(y) = \begin{cases} \frac{1}{\delta} y & ||y|| \leq \delta \\ \frac{y}{||y||} & ||y|| > \delta \end{cases}$$

The whole gradient can be represented like, $\frac{\partial f}{\partial X} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$

Then the hessian matrix is $\begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix}$,

where $\frac{\partial^2 f}{\partial x_i \partial x_i} = I + \lambda \sum_{j=1}^{n} \varphi''_{hub}(x_i - x_j)$ and $\frac{\partial^2 f}{\partial x_i \partial x_j} = -\lambda \varphi''_{hub}(x_i - x_j)$.

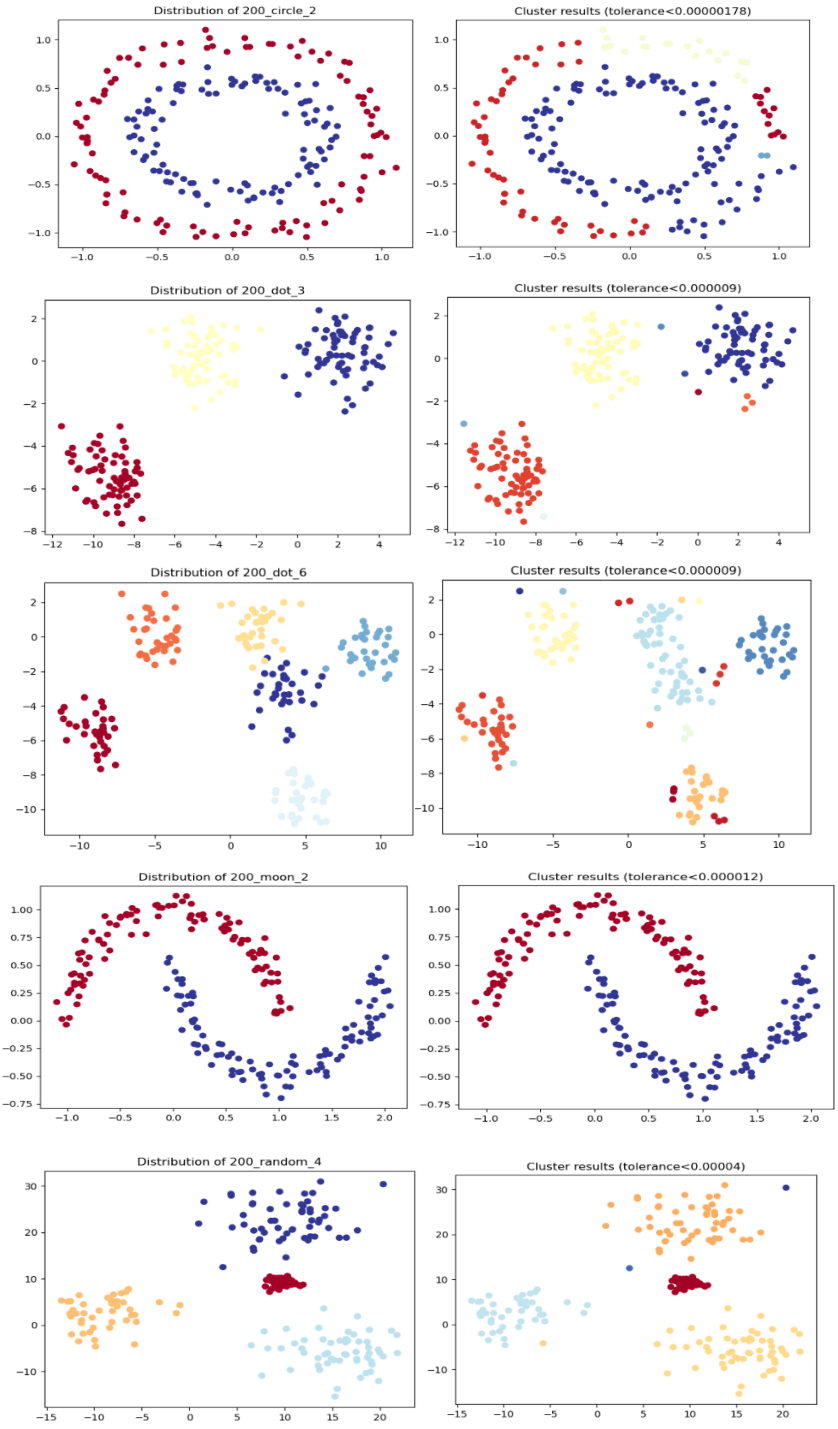
For which $\varphi''_{hub}(y) = \begin{cases} \frac{1}{\delta} I & ||y|| \le \delta \\ \frac{I}{||y||} - \frac{y \cdot y^T}{||y||^3} & ||y|| > \delta \end{cases}$

According to these formulas, try to implement them in python.

- Experiment

a) Clustering results on different datasets

1) Cluster Result



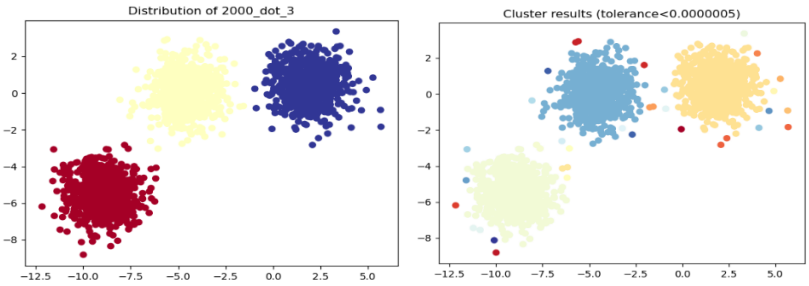


Figure 7: The clustering results

These are the results visualized after clustering in a different dataset. The pictures on the left are the distribution of the original data, and pictures on the right side are clustering results using Newton-CG Method. After experiments, we can find in the circle dataset, the result is not quite good although after too many tests trying on different tolerance. However, the result in the moon dataset is perfect in that all points are clustered in the right classes. In the other datasets, the solutions are also ok that except for some outliers, most of the classes are distinguished.

All these are visualizations of artificial data sets, but for real datasets, they can't be visualized like above. Then we use the convergence line and relative error to evaluate.

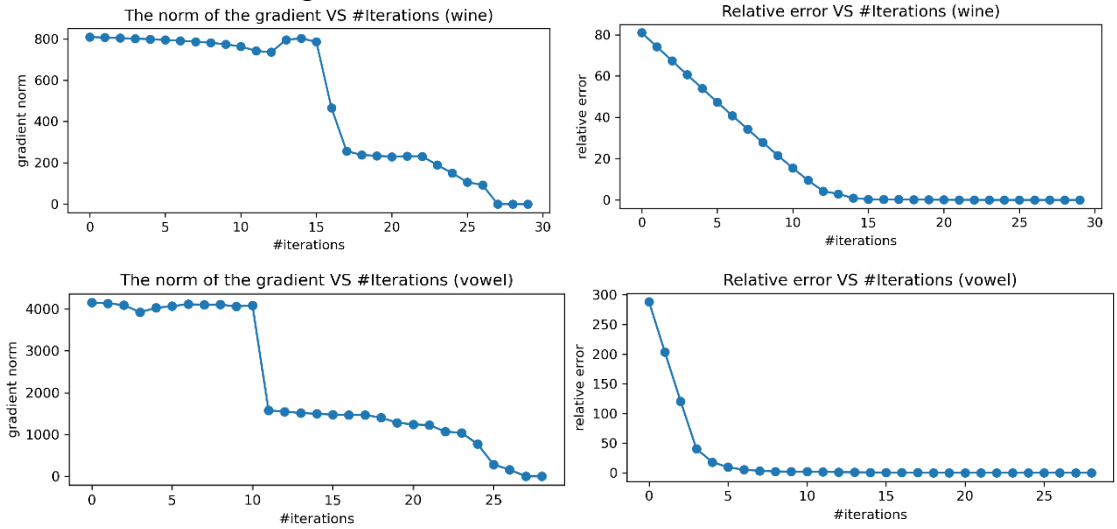


Figure 8: The relationship between Relative Error/Gradient Norm and #Iterations

From the convergence and relative error lines we can see that using Newton-CG, there is a vain and rapid decline during the iterations. Then it gradually converges to 0. After we can compare the CPU time and the number of iterations.

2) CPU Time and iteration number

We can compare the performance of Newton-CG on different data sets.

Dataset Name	Iteration Num	CPU Time	Norm Gradient
200_circle_2	15	44 s	<6.7e-5
200_dot_3	60	2 min 46 s	<2e-5
200_dot_6	76	3 min 20 s	<2e-5
200_moon_2	19	1 min	<5.7e-5
200_random_4	193	8 min 31 s	<7.9e-6
2000_dot_3	14	86 min 37 s	<2.2e-6

Table 5: Artificial datasets (no weight, tol=1e-3, lamda=0.5)

Dataset Name	d	n	Iteration Num	CPU Time	Norm Gradient	Tol
wine	13	178	29	1.63 min.	<1.7e-5	1e-3
vowel	10	528	28	20.19min	<0.0006	1e-3

Table 6: Real datasets (no weight, tol=1e-3 lamda=0.5)

We can see that although the number of iterations of Newton-CG is not too large, the size of the dataset can lead to a long time to run an iteration, like 2000_dot_3, which needs more than one hour to finish, so we only run one large size dataset.

b) The impact of regularization parametersλ

Set λ =0.05, 0.1, 0.2, 0.5, 0.8, 1, 3, 5, 7.5, 10. And using the dataset "200_moon_2", "200_dot_6" to run different λ and compare them.

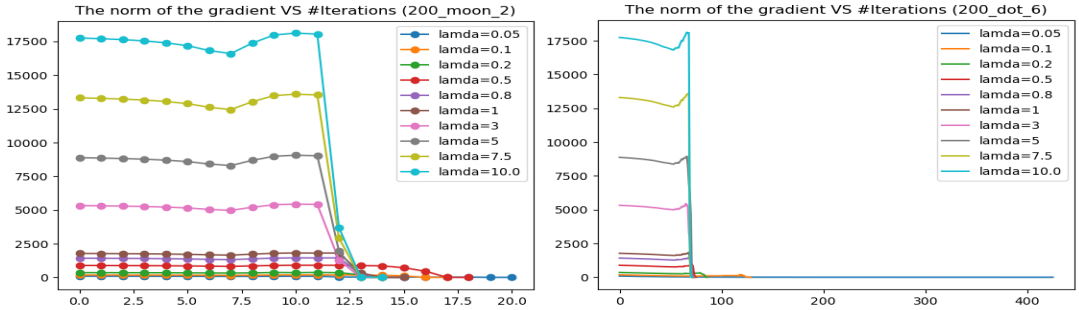


Figure 9: The relationship between Relative Error/Gradient Norm and #Iterations

λ	0.05	0.1	0.2	0.5	0.8	1	3	5	7.5	10
200_moon_2	21	18	16	19	16	16	15	15	15	15
200_dot_6	426	130	87	76	74	73	73	74	72	72

Table 7: The iterations of different λ

From the two plots of different datasets with different λ, we find that the bigger λ, the bigger the initial gradient and the more dramatic decline in convergence. And we compare the number of iterations then find that the slower λ is, the more iterations need. And this is also understandable, the smaller λ means that the smaller the difference in different directions, so it may need more attempts to find a better direction.

- Weighted Models
  - Introduction of weighted model

Now, we change the objective function, giving weights based on the input data A, so we also need to change the gradient and hessian. But in fact, we only need to add weight to the functions.

The gradient of x_i need to be changed like

$$\frac{\partial f}{\partial x_i} = x_i - a_i - \lambda \sum_{j=1}^{i-1} w_{ji} \varphi''_{hub}(x_j - x_i) + \lambda \sum_{j=i+1}^{n} w_{ij} \varphi''_{hub}(x_j - x_i).$$

Then the whole formula is,

$$\frac{\partial f}{\partial x_i} = x_i - a_i + \lambda \sum_{j=1}^{i-1} w_{ji} \varphi''_{hub}(x_i - x_j) + \lambda \sum_{j=i+1}^{n} w_{ij} \varphi''_{hub}(x_j - x_i)$$

The $\frac{\partial^2 f}{\partial x_i \partial x_i} = I + \lambda \sum_{j=1}^{n} w\varphi''_{hub}(x_i - x_j)$

And $\frac{\partial^2 f}{\partial x_i \partial x_j} = -\lambda\, w\varphi''_{hub}(x_i - x_j)$

The $w = \begin{cases} w_{ij} & j > i \\ w_{ji} & j < i \\ 0 & j = i \end{cases}$

- AGM V.S. Weighted AGM

Use wine data set to test AGM and weighted AGM (λ = 0.05, to = 1e-1). The norm gradient of AGM started to drop significantly after 6000 iterations, while that of the weighted methods drop steadily from the beginning. Although the number of iterations of the two methods is similar, the weighted method is faster with less CPU time on Wine. The weighted AGM can achieve a better result and converge faster.

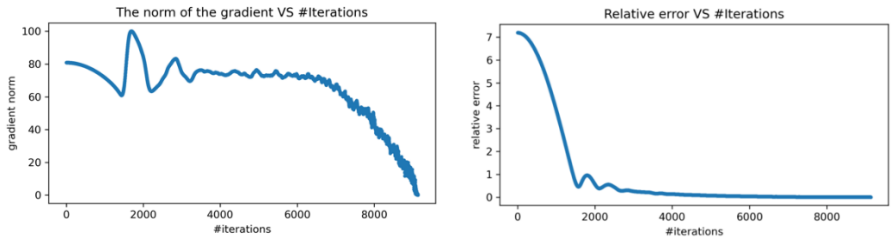


Figure 5: norm gradient and relative error of AGM

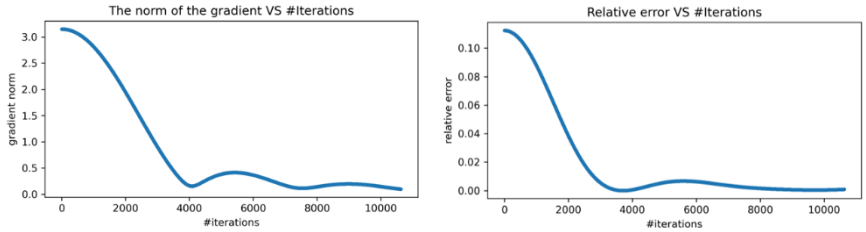


Figure 6: norm gradient and relative error of Weighted AGM

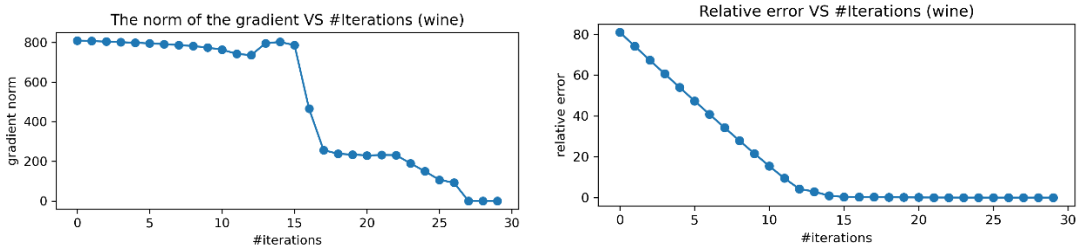
Method	Iteration Num	CPU Time	Norm Gradient
Weighted	10633	47.15 min	<0.1
No weighted	9136	169.01 min	<0.1

Table 8: compare weighted and no weighted AGM

- Newton-CG V.S. Weighted Newton-CG

1) Clustering results (Wine (lamda=0.05, tol=1e-3, k=5))

No weighted



Weighted

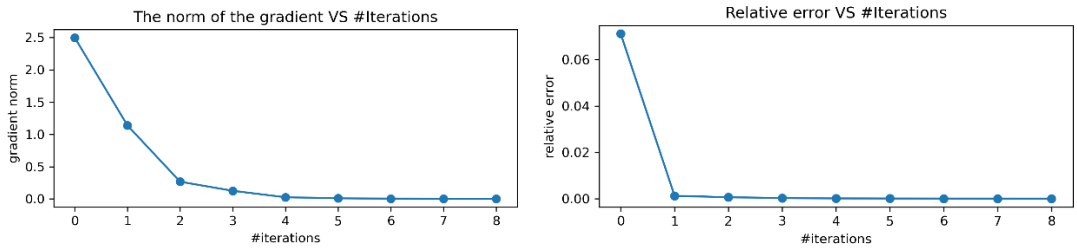


Figure 10: The relationship between Relative Error/Gradient Norm and #Iterations

Comparing the weighted model and no weighted model, we can find that the no weighted model needs more iteration times and the gradients change from a large number to 0, but for the weighted model, the gradient changes are not so violent because of adding weights.

2) CPU time

Method	Iteration Num	CPU Time	Norm Gradient
No weighted	43	2 min 48 s	<0.0003
Weighted	8	25 s	<0.0008

Table 9: The comparison between weighted model and no weighted model

From the CPU time and the number of iterations, we can also find that the weighted model can do better on the wine dataset.

- Weighted AGM V.S. Weighted Newton CG

Weighted Newton CG performs far better than weighted AGM, no matter on iteration num or on CPU time. AGM is only based on the first-order information, the step update is of low efficiency. Because of using the second-order derivative information, the number of iterations and CPU time are dramatically better than AGM.

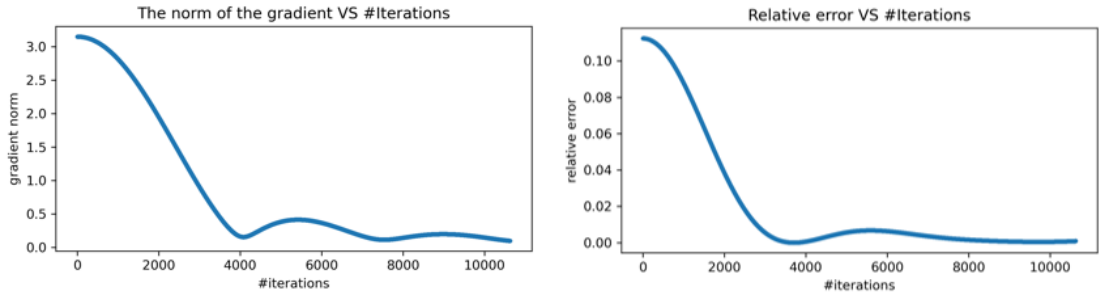


Figure 11: norm gradient and relative error of Weighted AGM

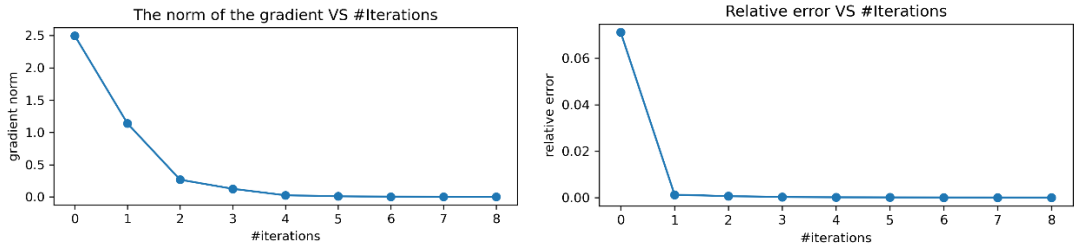


Figure 12: norm gradient and relative error of Weighted Newton CG

Method	Iteration Num	CPU Time	Norm Gradient
Weighted AGM	10633	47.15 min	<0.1
Weighted Newton CG	8	0.42min	<0.0008

Table 10: compare weighted AGM and weighted Newton CG

4. Extensions and Stochastic Optimization

- BFGS, L-BFGS
  - Introduction

Newton's method is very computationally expensive. While the computation of the gradient scales as O(n), the computation of the inverse Hessian scales as $O(n^3)$ (computing the Hessian scales as $O(n^2)$, inverting it as scales as O(n3)). As the dimensions of our problem increase, the overhead in memory and time gets out of hand very quickly. It's clear at this point that the benefit of an increased convergence rate will be far outweighed by the large cost of the additional computation time. To address this issue, quasi-Newton methods are introduced, where we can have faster convergence than gradient descent, but a lower operational cost per iteration than Newton's method.

In quasi-Newton methods, instead of computing the actual Hessian, we just approximate it with a positive-definite matrix B, which is updated from iteration to iteration using information computed from previous steps. We immediately see that this scheme would yield a much less costly algorithm compared to Newton's method because instead of computing a large number of new quantities at each iteration, we're largely making use of previously computed quantities.

The specific update for B is given by the specific quasi-Newton method used.

Here, we will focus on one of the most popular methods, known as the BFGS method:

The Globalized BFGS-Method with Armijo step size

Begin

Initialization: $x^0 \in R^n$ and asymmetric pos. def. matrix $H_0 \in R^{n*n}$, and choose $\sigma, \gamma \in (0,1)$.

for k = 0, 1, 2, … **do**

$d^k = -H_k \nabla f(x^k)$

$Computing\ the\ stepsize\ \alpha_k\ using\ Armijo\ line\ search$

$x^{k+1} = x^k + \alpha_k d^k$

if $\|\nabla f(x^{k+1})\| \leq tol$ **do**

return x^{k+1}

else do

$s^k = x^{k+1} - x^k, \quad y^k = \nabla f(x^{k+1}) - \nabla f(x^k)$

if $(s^k)^T y^k < 0$ **do**

$H_{k+1} = H_k$

else do

$$H_{k+1} = H_k + \frac{(s^k - H_k y^k)(s^k)^T + s^k(s^k - H_k y^k)^T}{(s^k)^T y^k} - \frac{((s^k - H_k y^k)^T y^k}{((s^k)^T y^k)^2}$$

end

  - Experiment

Initial settings:

tol	maxiter	lambda	delta	s	sigma	gamma
1e-3	1e5	0.5	0.001	1	0.5	0.1

Table 11: Global variables

a) Tests on Artificial Data Set

We consider two artificial data set: data_200_dot_3 and data_200_dot_6. Visualization of the BFGS

performance on these two data sets is shown below. We find that BFGS gives a satisfying classification for both problems. Significant decline of the gradient norm occurs within ten iterations and the relative error approach zero within five iterations.

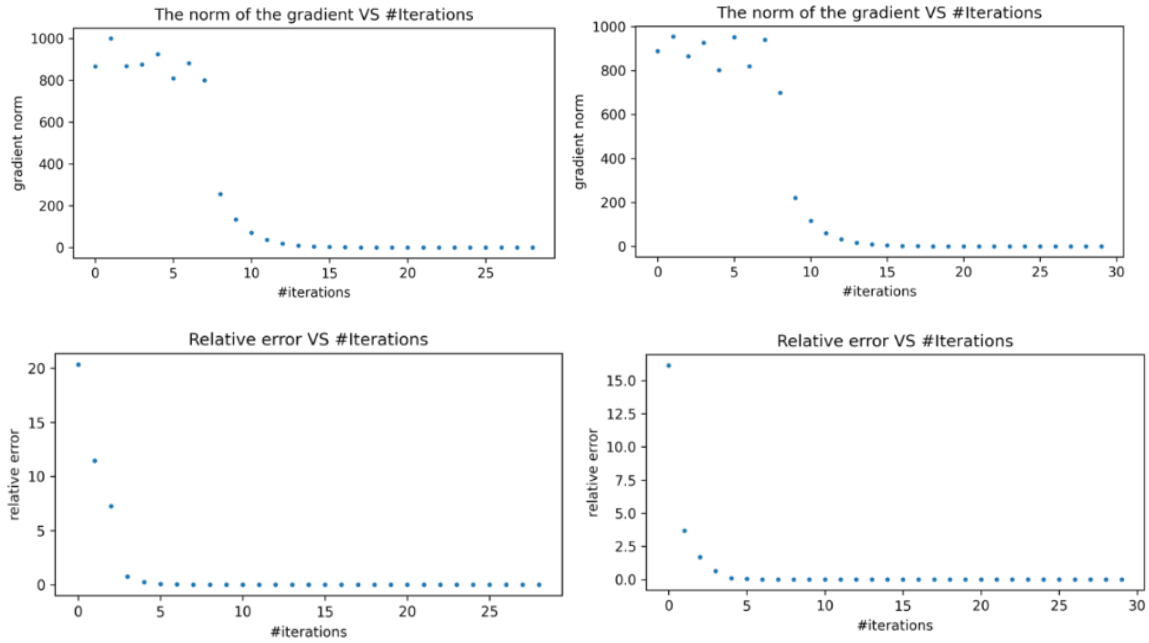


Figure 13: The relationship between Relative Error/Gradient Norm and #Iterations (left: 200_dot_3; right: 200_dot_6)

b) Comparison with Newton-CG

Utilizing Newton-CG, running the experiments on the same data sets (data_200_dot_3 and data_200_dot_6) gives the following results.

As we excepted, the BFGS method greatly reduces CPU time. In both tests, the running time of BFGS is less than one second, while Newton-CG needs two to three minutes. Moreover, the number of iterations of BFGS is also less than half of Newton-CG.

Dataset Name	Iteration Number		CPU Time		Norm Gradient	
	Newton-CG	BFGS	Newton-CG	BFGS	Newton-CG	BFGS
200_dot_3	60	28	2 min 46 s	0.13 s	< 2e-5	< 2e-5
200_dot_6	76	29	3 min 20 s	0.14 s	< 2e-5	< 2e-5

Table 12: Newton-CG vs. BFGS

So far, we have built the approximations B_k and H_k of the Hessian and inverse of the Hessian as full $n \times n$ matrices. However, if n is large, this might not be possible or too expensive. However, we do not need H_k explicitly, we are only interested in $d^k = -H_k \nabla f(x^k)$.

For a vector $v \in R^n$, it holds that:

$$H_{k+1}v = H_k\left[v - \frac{(s^k)^\top v}{(s^k)^\top y^k}\cdot y^k\right] + \frac{(s^k)^\top v}{(s^k)^\top y^k}s^k - \frac{(y^k)^\top H_k\left[v - \frac{(s^k)^\top v}{(s^k)^\top y^k}\cdot y^k\right]}{(s^k)^\top y^k}s^k$$

Hence, $H_{k+1}v$ can be computed recursively: $\beta_k = \frac{(s^k)^\top v}{(s^k)^\top y^k}$, $q^k = v - \beta_k y^k$, $p^k = H_k q^k$, $H_{k+1}v = p^k + \left[\beta_k - \frac{(y^k)^\top p^k}{(s^k)^\top y^k}\right]s^k$

The idea of the limited memory BFGS method is to store only the last m pairs.

$$\{y^{k-m}, y^{k-m+1}, \dots, y^k\} \quad \text{and} \quad \{s^{k-m}, s^{k-m+1}, \dots, s^k\}$$

and to use this information to build a much cheaper BFGS-type approximation of the Hessian.

c) Experiments

In this part, we are going to compare the BFGS method and L-BFGS method by testing the wine data set.

Both methods converge eventually. However, the L-BFGS method converges slower than the BFGS method. In the first thirty iterations, the gradient decline of L-BFGS method is inapparent. In the BFGS method, we find a significant decline within 7 iterations. It is a trade-off between accuracy and saving storage space. To save storage space, the L-BFGS only store and use the last m pairs of $\{y^{k-m}, y^{k-m+1}, \dots, y^k\}$ and $\{s^{k-m}, s^{k-m+1}, \dots, s^k\}$. Since the L-BFGS does not utilize full information, and this might be the reason that it didn't find the right way until the thirtieth iteration.

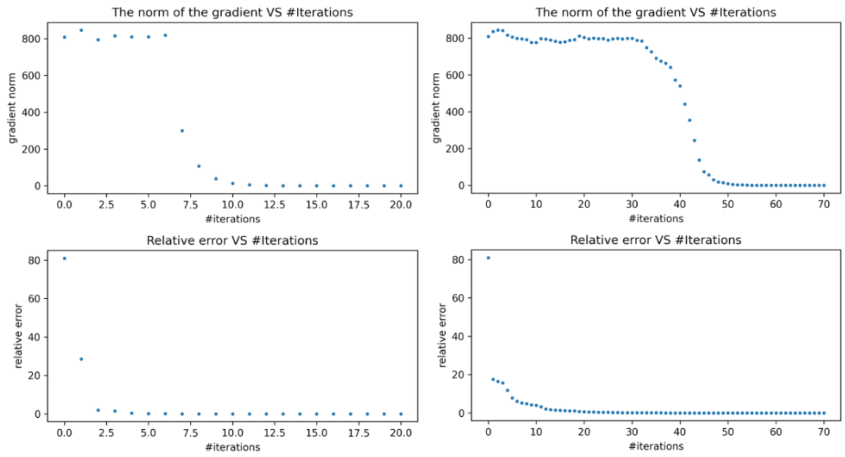


Figure 14: BFGS vs. L-BFGS
(left: BFGS; right: L-BFGS)

- SGD，Mini-Batch SGD
  - Introduction

a) Stochastic gradient descent[1]

Stochastic gradient descent (often abbreviated SGD) is an iterative method for optimizing an objective function with suitable smoothness properties. It can be regarded as a stochastic approximation of gradient descent optimization since it replaces the actual gradient (calculated from the entire data set) with an estimate thereof (calculated from a randomly selected subset of the data).

Specifically, we often use empirical risk $E_n(f)$ to measure the training set performance in machine learning.

$$E_n(f) = \frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i), y_i)$$

To minimize the empirical risk, gradient descent (GD) is often used. Each iteration updates the weights w based on the gradient of $E_n(f_w)$.

$$w_{t+1} = w_t - \eta \frac{1}{n}\sum_{i=1}^{n} \nabla_w Q(z_i, w_t)$$

where η is an adequately chosen learning rate.

The stochastic gradient descent algorithm (SGD) is a drastic simplification. Instead of computing the gradient of $E_n(f_w)$ exactly, each iteration estimates this gradient based on a single randomly picked example z_t:

$$w_{t+1} = w_t - \eta_t \nabla_w Q(z_t, w_t)$$

The stochastic process $\{w_t, t = 1, \ldots\}$ depends on the examples randomly picked at each iteration. Since the stochastic algorithm does not need to remember which examples were visited during the previous iterations, it can process examples on the fly in a deployed system.

b) Mini-Batch Stochastic Gradient Descent

During training, Mini-Batch Stochastic Gradient Descent (mini-batch SGD) processes a group of examples per iteration. For notational simplicity, assume that n is divisible by the number of mini-batches m. Then we partition the examples into m mini-batches, each of size $b = n/m$.

Given a random minibatch $I \subset \{1, \ldots, n\}$ of size b, we can define the objective function on I as

$$Q_I(w) = \frac{1}{|I|} \sum_{i \in I} Q_i(w) \text{ we have } Q(w) = \mathbf{E}_I[Q_I(w)]$$

In the simple case that $\Omega = \mathbb{R}^d$ the mini-batch SGD employs the following stochastic update rule: at each iteration t, we pick mini-batch $I_t \subset \{1, \ldots, n\}$ of size b at random and let

$$w_t = w_{t-1} - \eta_t \nabla Q_{I_t}(w)$$

Although b times more examples are processed in an iteration, the mini-batch training can converge much slower than that of standard SGD with the same number of processed examples. In practice, the convergence rate slows down dramatically in terms of the number of examples processed, when we use a large mini-batch size [2].

■ Experiment

a) Data

After preliminary experiments, the gradient descent algorithm takes a long time (perhaps more than 3 hours) to reach convergence under large data sets (the amount of data usually exceeds several thousand). We do not have enough computing resources to run the algorithm under the large data set, so we consider using the wine data set and the artificial data set (200 two-dimensional points, 6 cluster centers) generated in the data preparation stage as the experimental data sets.

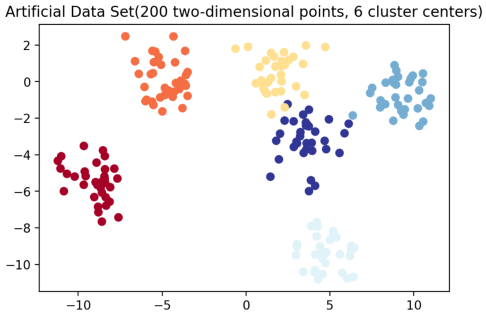


Figure 15: Display of true labels of the artificial data set

b) The impact of different batch sizes

In the Mini-Batch SGD algorithm, the batch size is a very important hyperparameter. We are here to discuss the influence of different batch sizes on convergence speed, time, convergence effect, and other factors.

Except for batch size, other hyperparameter settings of artificial data set and wine data set are as follows. We realized that convergence results usually require decreasing learning rates satisfying the conditions $\sum_t \eta_t^2 < \infty$ and $\sum_t \eta_t = \infty$ [2]. So we choose $\alpha = 1/k$ (k = #iterations) as our learning rate.

hyperparameter	artificial data set	wine data set
λ	0.1	0.1
δ	0.0001	0.0001
Learning rate (α)	1/k (k = #iterations)	0.1/k (k = #iterations)
Gradient tolerance (tol)	0.1	0.1
Max iteration number	30000	20000

Table 13: hyperparameter settings of artificial data set and wine data set

1) CPU Time

Iteration Num	BS=1(SGD)	BS=25	BS=50	BS=100	BS=200(GD)
10000	0.86s	11.79s	23.02s	45.71s	90.67s
20000	1.72s	23.56s	46.19s	91.59s	181.67s
30000	2.60s	35.35s	69.23s	137.30s	272.65s

Table 14: relationship between CPU Time and #iterations/batch size(manual data set)

Iteration Num	BS=1(SGD)	BS=50	BS=100	BS=178(GD)
10000	1.49s	38.15s	75.14s	132.47s
20000	2.95s	76.27s	150.23s	265.15s

Table 15: the relationship between CPU Time and #iterations/batch size (wine data set)

When the batch size is 1, Mini-Batch SGD is SGD. When the batch size is equal to the number of data in the data set, Mini-Batch SGD is GD. From the table, we can find that when the batch size is unchanged, the time taken for every 10,000 iterations is approximate. When the batch size is changed, the larger the batch size, the more CPU time is consumed, and the time consumed is approximately linearly related to the batch size.

2) Convergence rate

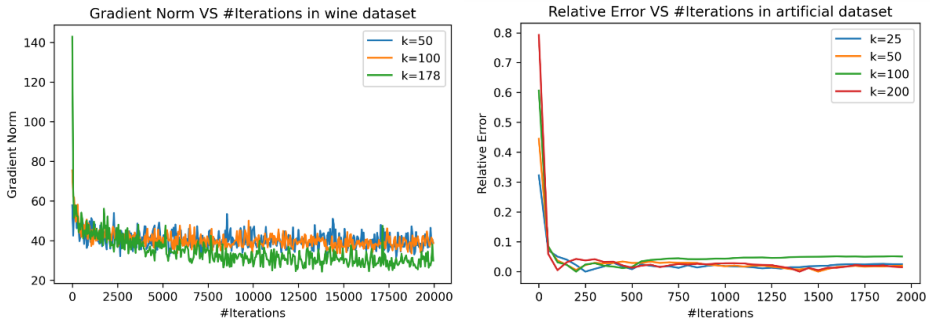


Figure 16: The relationship between Relative Error/Gradient Norm and #Iterations

We can see from the figure that, in general, the larger the batch size, the smaller the relative error, and the smaller the gradient norm after the same number of iterations. However, we must take into account that when the batch size is different, the time spent in each iteration is different. Therefore, the batch

size should be a certain number between $[1, the\ amount\ of\ data\ in\ the\ data\ set]$, so that the Mini-Batch SGD algorithm can achieve the effect of GD in a relatively short time. If the batch size is too small, the convergence will be very unstable. If the batch size is too large, it will take a lot of time, especially on a large data set.

3) Experimental result

Because the data in the artificial data set is two-dimensional, we can display them on a two-dimensional plane. The clustering results corresponding to different k are selected as follows.

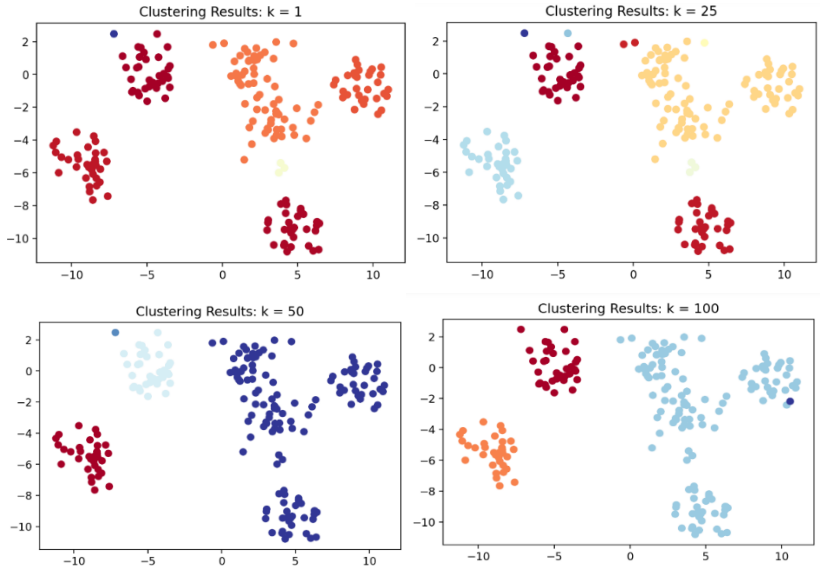


Figure 17: Clustering results of different k

It can be seen from the figure that under different batch sizes, the original clustering problem can achieve a reasonable clustering result, which can indicate that the difference in batch size does not affect the convergence of the function.

- Types of norm regularizations
  - Introduction

Regularization is a very important technique in machine learning to prevent overfitting. The difference between L1 and L2 is that L2 is the sum of the square of the weights, while L1 is the sum of the weights. And the maximum norm is the maximum weight in the vector. These three types of norm regularizations in Huber function are as follows:

$$\| x \|_1 = \sum_{i=1}^{N} |x_i|,\ \| x \|_2 = (\sum_{i=1}^{N} |x_i|^2)^{\frac{1}{2}},\ \| x \|_\infty = \max_i |x_i|, i = \arg\max_i |x_i|$$

The differences between their properties can be promptly summarized as follows:

L2 regularization	L1 regularization
Computational efficiency due to having analytical solutions	Computational inefficient on non-sparse cases
Non-sparse outputs	Sparse outputs
No feature selection	Built-in feature selection

Table 16: the difference between L1 regularization and L2 regularization

From the computational efficiency perspective, L1-norm does not have an analytical solution, but L2-norm does. This allows the L2-norm solutions to be calculated computationally efficiently. However, L1-norm solutions have sparsity properties that allow them to be used along with sparse algorithms,

making the calculation more computationally efficient.
Sparsity refers to that only very few entries in a matrix (or vector) are non-zero. L1-norm has the property of producing many coefficients with zero values or very small values with few large coefficients.
Built-in feature selection is frequently mentioned as an important property of the L1-norm, which the L2-norm does not. This is a result of the L1-norm, which tends to produce sparse coefficients (explained below). L2-norm produces non-sparse coefficients, so it does not have this property.
Moreover, L2-norm has unique solutions while L1-norm does not. Similar to L1 norm regularization, the maximum norm may have several solutions. It treats a matrix as if it were a vector, with the entries written in a rectangular array instead of a line. In a convex problem, the corners maybe not be on the axis. So, the solutions will give lots of coefficients shrunk to have absolute value, but not sparse solutions.
Even though we put lots of effort into how to get a specific correct hessian of the Huber function for different norm regularization, we failed. Since the L1 norm is not differentiable, it is hard to get the Hessian matrix, and we can just mainly focus on AGM instead of Newton-CG. When calculating the gradient, a subgradient is utilized. The subgradient of the L1 norm is as follows:

$$\frac{\partial}{\partial \mathbf{x}} \| \mathbf{x} \|_1 = \operatorname{sign}(\mathbf{x}) = \begin{cases} +1 & x_i > 0 \\ -1 & x_i < 0 \\ [-1,1] & x_i = 0 \end{cases}$$

Similarly, the subgradient of the maximum norm is as follows:

$$\frac{\partial \| x \|_\infty}{\partial x} = \operatorname{sign}(x_i) \cdot e_i, \frac{\partial \| x \|_\infty^2}{\partial x} = 2 \cdot x_i \cdot e_i, i = \operatorname{argmax} |x_i|$$

Then, the gradient of the Huber function in L1 norm and L infinite norm can be denoted respectively as follows:

$$\varphi'_{\text{hub}} = \begin{cases} \frac{1}{\delta} \cdot \| y \| \cdot \operatorname{sign}(y), & \| y \| \leqslant \delta \\ \operatorname{sign}(y), & \| y \| > \delta \end{cases}$$

$$\varphi'_{\text{hub}} == \begin{cases} \frac{1}{\delta} y_i \cdot e_i, & \| y \| \leqslant \delta \\ \operatorname{sign}(y_i) \cdot e_i, & \| y \| > \delta \end{cases}, i = \operatorname{argmax} |x_i|$$

- Experiment

We test AGM on data_200_dot_6 with 3 different norm types. Set lamda=0.05, tol=1e-1, max iteration = 30000. The results are shown in the following table. When we use L1 norm and L-inf norm, it cannot converge even reaching the max iteration. The run time is also longer when comparing the performance of other norms with L2 norm.

Norm type	Iteration Num	CPU Time	Norm Gradient
L1 norm	30000	754.01s	0.96
L2 norm	11530	239.04s	< 0.1
L-inf norm	30000	1249.03s	0.51

Table 17: performance of AGM on different types of norm regularizations w.r.t dataset 200_dot_6

Compared among different types of norm regularization, the results show that the ℓ_2-$(\|\cdot\|_2)$ norm regularization method based on the Huber function has the fastest convergence. The performance in relative error has no highlight.

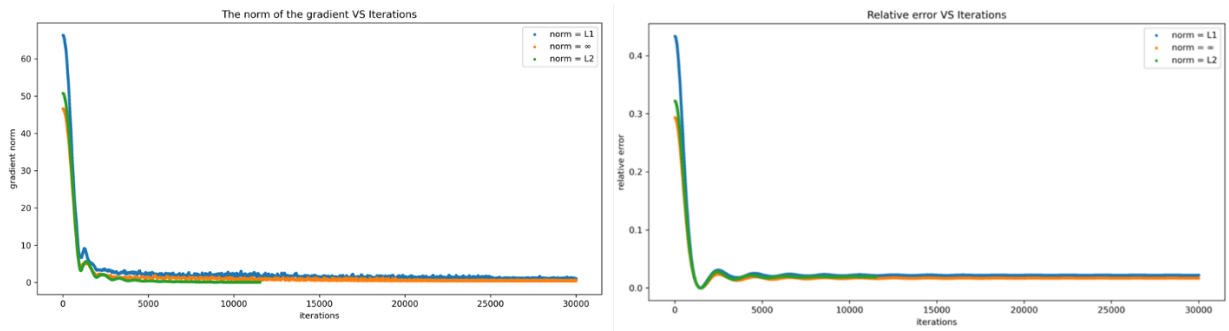


Figure 17: The Gradient Norm & relative error vs Iterations

5. Conclusion

From our experiments, we learned a lot about AGM, Newton CG, different optimization methods (BFGS/L-BFGS……), matrix norms, mini-batch SGD, etc. AGM algorithm cannot perform well on large-scale or special shapes data sets. It is also really time-consuming with loose tolerance, which is only based on first-order information. Apply weighted methods to improve the AGM, it can achieve better results and better runtimes.

Regarding Newton-CG, we find that the number of iterations of the Newton method is significantly smaller than that of AGM, which of course is due to the use of the second-order derivative information. But in the experiment, how to speed up and store the Hessian matrix has become a very important issue. In the future, we can continue to try to store the hessian matrix in a dictionary and optimize the codes to achieve speed improvements.

In the BFGS method, instead of computing the actual Hessian, we just approximate it with a positive-definite matrix B. Therefore, it can achieve faster convergence than gradient descent, but the lower operational cost per iteration than Newton's method. The L-BFGS method aims to build a much cheaper BFGS-type approximation of the Hessian by storing only the last m pairs of $\{y^{k-m}, y^{k-m+1}, \dots, y^k\}$ and $\{s^{k-m}, s^{k-m+1}, \dots, s^k\}$. Although the L-BFGS is much cheaper, it takes more iterations and a longer time to converge.

Mini-batch SGD can be regarded as a stochastic approximation of gradient descent optimization since it replaces the actual gradient (calculated from the entire data set) with an estimate thereof (calculated from a randomly selected subset of the data). Generally, it can increase the calculation speed without greatly losing accuracy. This method is especially useful on large data sets. However, choosing a good batch size is also a question worth discussing.

6. References

References

[1] Bottou, Léon. "Stochastic gradient descent tricks." Neural networks: Tricks of the trade. Springer, Berlin, Heidelberg, 2012. 421-436.

[2] Li, Mu, et al. "Efficient mini-batch training for stochastic optimization." Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. 2014.

7. Appendix

In [ ]:
```
#Load libraries
import pandas
import numpy as np
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt
import random
import math
```

In [ ]:
```
import time
from sklearn.preprocessing import normalize
from scipy.io import loadmat
from scipy import sparse
from scipy.sparse import csc_matrix
import scipy
```

1.1 Input Data

In [ ]:
```
#generate random x0
def generate_random_mat(X):
  #??????????
  shape = X.shape
  csc_matrix((data, indices, indptr),shape)
  return

def generate_random_array(X):
  A=np.random.rand(X.shape[0],X.shape[1])
  return A
```

In [ ]:
```
#input dataset wine(mat)
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:
```
wine_data_mat = loadmat("/content/drive/MyDrive/wine_data.mat",mat_dtype=True
wine_label_mat = loadmat("/content/drive/MyDrive/vowel_label.mat",mat_dtype=T
wine_data=wine_data_mat['A']
wine_label=wine_label_mat['b']
wine_data
```

In [ ]:
```
type(wine_data)
```

In [ ]:
```
#input 200_dot_3
data_200_dot_3 = np.loadtxt('/content/drive/MyDrive/op_data/200_dot_3.txt')
x_200_dot_3 = data_200_dot_3[0]
y_200_dot_3 = data_200_dot_3[1]
c_200_dot_3 = data_200_dot_3[2]
data_200_dot_3 = np.stack((x_200_dot_3,y_200_dot_3), axis=1)
#normalization
#data_200_dot_3 = normalize(data_200_dot_3, axis=0, norm='max')
```

In [ ]:

```
A=csc_matrix((np.array([1,1]),(np.array([0,1]),np.array([0,1]))),shape=(2,2))
print(A)
x0=np.concatenate(list(map(lambda i:A.getcol(i).toarray(),range(A.shape[1]))))
print(x0)
```

1.2 Data Generation

In [ ]:
```
#Data Generation1: 多组随机分布，可重叠
np.random.seed(2)
def data_generation(n,d,range,theta):
  arr_1 = np.random.uniform(range[0][0],range[0][1],(d,1))
  arr_2 = np.random.uniform(range[1][0],range[1][1],(d,1))
  data_reference = np.hstack((arr_1,arr_2))
  data_group = np.random.choice(d,n)
  data_generation = list()
  for i in data_group:
    a = [data_reference[i][0]+random.normalvariate(0,theta[i]),data_reference
    data_generation.append(a)
  return data_group, np.array(data_generation)

label_generation1, data_generation1 =  data_generation(20000,4,[[-10,35],[-20
```

In [ ]:
```
label_generation1[:10]
```

In [ ]:
```
data_generation1[:10]
```

In [ ]:
```
x=data_generation1[:,0]
y=data_generation1[:,1]
c=label_generation1

fig, ax = plt.subplots()
scatter = ax.scatter(x, y, c=c, cmap=plt.cm.RdYlBu)
plt.show()
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:
```
np.savetxt('/content/drive/MyDrive/op_data/20000_random_4.txt',(x,y,c),fmt="%
```

In [ ]:
```
np.loadtxt('/content/drive/MyDrive/op_data/20000_random_4.txt')
```

In [ ]:
```
#Data Generation2: 月牙形
np.random.seed(2)
def data_generation(n,range,theta):
  random.seed(3)
  a = random.uniform(range[0][0]/3,range[0][1]/3)
  b = random.uniform(range[0][0]/2,range[0][1]/2)
  x1 = np.linspace(range[0][0], (range[0][0]+range[0][1])/2, int(n/2), dtype=
  noise = np.random.normal(0,theta,x1.shape).astype(np.float32)
  y1 = -np.square(x1) / a + b + noise

  x2 = np.linspace((range[0][0]+range[0][1])/2, range[0][1], int(n/2), dtype=
```

```
  y2 = np.square(x2 - ((range[0][0]+range[0][1])/2)) / a - b + noise

  group1 = np.hstack((x1,y1))
  group2 = np.hstack((x2,y2))
  data_generation = np.vstack((group1,group2))

  data_group = np.hstack((np.zeros(int(n/2)),np.ones(int(n/2))))
  return data_group, data_generation

label_generation2, data_generation2 = data_generation(600,[[-40,40]],20)
```

In [ ]:
```
x=data_generation2[:,0]
y=data_generation2[:,1]
c=label_generation2

fig, ax = plt.subplots()
scatter = ax.scatter(x, y, c=c, cmap=plt.cm.RdYlBu)
plt.show()
```

In [ ]:
```
from sklearn.datasets import make_moons
data_generation2, label_generation2 = make_moons(n_samples=200, noise=0.07, r
x=data_generation2[:,0]
y=data_generation2[:,1]
c=label_generation2

fig, ax = plt.subplots()
scatter = ax.scatter(x, y, c=c, cmap=plt.cm.RdYlBu)
plt.show()
```

In [ ]:
```
np.savetxt('/content/drive/MyDrive/op_data/20000_moon_2.txt',(x,y,c),fmt="%f"
```

In [ ]:
```
np.loadtxt('/content/drive/MyDrive/op_data/20000_moon_2.txt')
```

In [ ]:
```
#Data Generation3: 环形
np.random.seed(2)
from sklearn.datasets import make_circles
data_generation3, label_generation3 = make_circles(n_samples=200, noise=0.07,
x=data_generation3[:,0]
y=data_generation3[:,1]
c=label_generation3

fig, ax = plt.subplots()
scatter = ax.scatter(x, y, c=c, cmap=plt.cm.RdYlBu)
plt.show()
```

In [ ]:
```
np.savetxt('/content/drive/MyDrive/op_data/20000_circle_2.txt',(x,y,c),fmt="%
```

In [ ]:
```
data = np.loadtxt('/content/drive/MyDrive/op_data/20000_circle_2.txt')
data
```

In [ ]:
```
x = data[0]
y = data[1]
label = data[2]
```

In [ ]:
```
label
```

In [ ]:
```
#Data Generation4: 点簇型
from sklearn.datasets import make_blobs
data_generation4, label_generation4 = make_blobs(n_samples=200, n_features=2,
x=data_generation4[:,0]
y=data_generation4[:,1]
c=label_generation4

fig, ax = plt.subplots()
scatter = ax.scatter(x, y, c=c, cmap=plt.cm.RdYlBu)
plt.show()
```

In [ ]:
```
np.savetxt('/content/drive/MyDrive/op_data/200_dot_6.txt',(x,y,c),fmt="%f",de
```

In [ ]:
```
np.loadtxt('/content/drive/MyDrive/op_data/200_dot_6.txt')
```

In [ ]:
```
#数据集生成 https://blog.csdn.net/qq_42730750/article/details/117286533
```

2.1 AGM

In [ ]:
```
from numpy.linalg import norm
def f(x,a,lambda1,delta):
  f_1 = np.sum((norm(x[i]-a[i]))**2 for i in range(len(x)))
  x_diff = list()
  f_2 = 0
  for i in range(len(a)):
    # for j in range(i+1,len(a)):
    #   x_diff.append(x[i]-x[j])
    xi_diff = x[i] - x[i+1:]
    xi_diff_norm = norm(xi_diff, ord=2, axis=1)
    mask = xi_diff_norm <= delta
    f_2 += np.sum(xi_diff_norm[mask] ** 2 / (2*delta))
    f_2 += np.sum(xi_diff_norm[~mask] - delta/2)

  #f_2 = phi(x_diff,delta)
  return 1/2*(f_1) + lambda1 * f_2

def g(x,a,lambda1,delta):
  grad_1 = x - a
  grad_2 = np.zeros(x.shape)
  for i in range(x.shape[0]):
    xi_diff = x[i] - x[i+1:]
    xi_diff_norm = norm(xi_diff, ord=2, axis=1)
    mask = xi_diff_norm <= delta

    grad_2[i] += lambda1 * np.sum(xi_diff[mask] / delta, axis=0)
    grad_2[i] += lambda1 * np.sum(xi_diff[~mask] / np.expand_dims(xi_diff_nor

  return grad_1 + grad_2
```

In [ ]:
```
def function(x,A,lamd,delta):
    '''
```

```
    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of function 'f_clust'.

    Returns
    -------
    type: float
        DESCRIPTION: The solution of f_clust.

    PS: A, row, col, lamd, delta all store as global variables.
    '''
    row=A.shape[0]
    col=A.shape[1]
    f1=[np.power(np.linalg.norm(x[i*row:(i+1)*row]-A[:, i].reshape(-1, 1)),2)
    f2=0
    for i in range(col):
        for j in range(i+1,col):
            n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
            f2+=0.5/delta*np.power(n,2) if n<=delta else n-delta*0.5
    return 0.5*sum(f1)+lamd*f2

def gradient(x,A,lamd,delta):
    '''

    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of gradient function.
    Returns
    -------
    type: numpy.ndarray
          shape: (row*col,1)
          DESCRIPTION: The solution of gradient function.

    '''
    row=A.shape[0]
    col=A.shape[1]
    solution1=np.asarray(np.concatenate(list(map(lambda i:x[i*row:(i+1)*row]-
    solution2=np.zeros([row*col,1])
    for i in range(col):
        for j in range(col):
            n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
            solution2[i*row:(i+1)*row]+=(x[i*row:(i+1)*row]-x[j*row:(j+1)*row
    return solution1+lamd*solution2
```

In [ ]:

```
gradient(x,A,0.05,1e-4)
```

In [ ]:

```
def AGM(obj,grad,x,options):
    alpha = options['alpha']
    tol = options['tol']
    isprint = options['isprint']

    k = 0
    x_pre = x
    t = 1
    t_pre = 1

    gradient = grad(x,A,lambda1,delta)
```

```
    x_lst = [x]
    gradient_lst = [norm(gradient)]
    while norm(gradient) >= tol and k < 30000:
        # accelerate
        beta = (t_pre -1)/t
        y = x + beta * (x - x_pre)
        # update
        x_pre = x
        x = y - alpha * grad(y,A,lambda1,delta)
        t_pre = t
        t = 1/2 * (1 + np.sqrt(1+4*t**2))
        gradient = grad(x,A,lambda1,delta)
        x_lst.append(x)
        gradient_lst.append(norm(gradient))
        k = k + 1
        if isprint:
            print("Iteration:",k,"obj:",obj(x,A,lambda1,delta),"norm_gradient
    return x,x_lst,gradient_lst
```

In [ ]:

```
def invoke_AGM(f,g,x_init,options):
  global lambda1
  global delta
  lambda1 = options['lambda']
  delta = options['delta']

  L = 1+x_init.shape[0]*lambda1/delta
  alpha = 1/L
  options["alpha"] = alpha
  solution, x_lst, gradient_lst = AGM(f,g,x_init,options)

  print("Stepsize:",alpha,"NumIteration:",len(x_lst)-1)
  return solution,x_lst,gradient_lst
```

In [ ]:

```
def plot_convergence_figure(obj, x_lst, gradient_lst, options, x_init):
    lambda1 = options['lambda']
    alpha = options['alpha']

    assert len(x_lst) == len(gradient_lst)
    iteration_num = len(gradient_lst)

    #The norm of the gradient VS #iterations
    plt.figure(dpi = 300, figsize=(6, 6.5))
    ax1 = plt.subplot(2, 1, 1)
    scatter = ax1.scatter(range(iteration_num), gradient_lst, s = 5)
    plt.title("The norm of the gradient VS #Iterations")
    ax1.set_xlabel('#iterations')
    ax1.set_ylabel('gradient norm')

    #Relative error VS #Iterations
    ax2 = plt.subplot(2, 1, 2)
    best_function_value = obj(x_lst[-1],x_init,lambda1,delta)
    scatter = ax2.scatter(range(iteration_num), [abs(obj(each,x_init,lambda1,
    plt.title("Relative error VS #Iterations")
    ax2.set_xlabel('#iterations')
    ax2.set_ylabel('relative error')

    plt.tight_layout()
    plt.show()
```

In [ ]:

```
#test data_200_dot_3
```

```
T1 = time.time()
options = {
    "lambda":0.05,
    "delta":1e-4,
    "tol":1e-1,
    "isprint":True
}

A=data_200_dot_3
solution,x_lst,gradient_lst = invoke_AGM(f,g,A,options)

T2 = time.time()
run_time = (T2-T1)/60
```

In [ ]:
```
run_time
```

In [ ]:
```
plot_convergence_figure(f,x4_lst,gradient4_lst,options,data_200_dot_3)
```

In [ ]:
```
#test data_200_dot_3 with random x0
T1 = time.time()
options = {
    "lambda":0.05,
    "delta":1e-4,
    "tol":1e-1,
    "isprint":True
}

A=generate_random_array(data_200_dot_3)
solution,x_lst,gradient_lst = invoke_AGM(f,g,A,options)

T2 = time.time()
run_time = (T2-T1)/60
```

In [ ]:
```
#test data_200_dot_3
T1 = time.time()
options = {
    "lambda":0.05,
    "delta":1e-4,
    "tol":1e-1,
    "isprint":True
}

A=data_200_dot_3
solution,x_lst,gradient_lst = invoke_AGM(function,gradient,A,options)

T2 = time.time()
run_time = (T2-T1)/60
```

In [ ]:
```
#test wine
T1 = time.time()
options = {
    "lambda":0.05,
    "delta":1e-4,
    "tol":1e-1,
    "isprint":True
}
A=wine_data
```

```
x0=np.concatenate(list(map(lambda i:A.getcol(i).toarray(),range(A.shape[1]))))
solution,x_lst,gradient_lst = invoke_AGM(function,gradient,x0,options)
T2 = time.time()
run_time = (T2-T1)/60
```

Weighted AGM

In [ ]:

```
def function(x,A,lamd,delta):
    '''
    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of function 'f_clust'.

    Returns
    -------
    type: float
        DESCRIPTION: The solution of f_clust.

    PS: A, row, col, lamd, delta all store as global variables.
    '''
    v=0.5
    row=A.shape[0]
    col=A.shape[1]
    f1=[np.power(np.linalg.norm(x[i*row:(i+1)*row]-A[:, i].reshape(-1, 1)),2)
    f2=0
    for i in range(col-1):
        for j in index_list[i]:
            n=np.linalg.norm(x[i*row:(i+1)*row]-x[(j+i)*row:(j+i+1)*row])
            w=np.exp((-v)*A_dist_list[i][j])
            f2+=w*0.5/delta*np.power(n,2) if n<=delta else w*(n-delta*0.5)
            #f2+=w*n
    return 0.5*sum(f1)+lamd*f2

def gradient(x,A,lamd,delta):
    '''

    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of gradient function.
    Returns
    -------
    type: numpy.ndarray
          shape: (row*col,1)
          DESCRIPTION: The solution of gradient function.

    '''
    v=0.5
    row=A.shape[0]
    col=A.shape[1]
    solution1=np.asarray(np.concatenate(list(map(lambda i:x[i*row:(i+1)*row]-
    solution2=np.zeros([row*col,1])
    for i in range(col-1):
        for j in index_list[i]:
            n=np.linalg.norm(x[i*row:(i+1)*row]-x[(j+i)*row:(j+i+1)*row])
            w=np.exp((-v)*A_dist_list[i][j])
            solution2[i*row:(i+1)*row]+=w*(x[i*row:(i+1)*row]-x[(j+i)*row:(i+
```

```
            #solution2[i*row:(i+1)*row]+=w*(x[i*row:(i+1)*row]-x[(j+i)*row:(j
    return solution1+lamd*solution2
```

In [ ]:

```
def find_k_near(A,k):
  row=A.shape[0]
  col=A.shape[1]
  A_dist_list = list()
  index_list = list()
  for i in range(col-1):
    if k > col-i-1:
      k = col-i-1
    A_diff = sparse.hstack([A[:,i] for _ in range(col-i-1)]) - A[:, i+1:]
    A_dist = np.array(A_diff.power(2).sum(axis=0)).squeeze(axis=0)
    A_dist_list.append(A_dist)
    index = np.argpartition(A_dist,k)[k:]
    index_list.append(index)
  return A_dist_list, index_list
```

In [ ]:

```
from numpy.linalg import norm
def W_AGM(obj,grad,x,options):
    global A_dist_list
    global index_list

    alpha = options['alpha']
    tol = options['tol']
    isprint = options['isprint']
    ################################################
    A_dist_list, index_list = find_k_near(A,7)
    ################################################
    k = 0
    x_pre = x
    t = 1
    t_pre = 1

    gradient = grad(x,A,lambda1,delta)
    x_lst = [x]
    gradient_lst = [norm(gradient)]
    while norm(gradient) >= tol:
        # accelerate
        beta = (t_pre -1)/t
        y = x + beta * (x - x_pre)
        # update
        x_pre = x
        x = y - alpha * grad(y,A,lambda1,delta)
        t_pre = t
        t = 1/2 * (1 + np.sqrt(1+4*t**2))
        gradient = grad(x,A,lambda1,delta)
        x_lst.append(x)
        gradient_lst.append(norm(gradient))
        k = k + 1
        if isprint:
            print("Iteration:",k,"obj:",obj(x,A,lambda1,delta),"norm_gradient
    return x,x_lst,gradient_lst
```

In [ ]:

```
def invoke_W_AGM(f,g,x_init,options):
  global lambda1
  global delta
  lambda1 = options['lambda']
  delta = options['delta']
```

```
  L = 1+x_init.shape[0]*lambda1/delta
  alpha = 1/L
  options["alpha"] = alpha
  solution, x_lst, gradient_lst = W_AGM(f,g,x_init,options)

  print("Stepsize:",alpha,"NumIteration:",len(x_lst)-1)
  return solution,x_lst,gradient_lst
```

In [ ]:
```
#test wine
T1 = time.time()
options = {
    "lambda":0.05,
    "delta":1e-4,
    "tol":1e-1,
    "isprint":True
}
A=wine_data
x0=np.concatenate(list(map(lambda i:A.getcol(i).toarray(),range(A.shape[1]))))
solution,x_lst,gradient_lst = invoke_W_AGM(function,gradient,x0,options)
T2 = time.time()
run_time = (T2-T1)/60
```

In [ ]:
```
print(run_time)
```

test find_k_near

In [ ]:
```
A=csc_matrix((np.array([1,1,1]),(np.array([0,1,1]),np.array([0,0,1])))),shape=
x=np.concatenate(list(map(lambda i:A.getcol(i).toarray(),range(A.shape[1])))),
np.array(A.todense())
```

In [ ]:
```
A_dist_list, index_list = find_k_near(A,2)
```

Other

In [ ]:
```
import numpy as np
from numpy.linalg import norm

class UnionFindSet:
    def __init__(self, n):
        self.parent = list(range(n))

    #合并index1和index2所属集合
    def union(self, index1: int, index2: int):
        self.parent[self.find(index2)] = self.find(index1)

    #查找index结点的父结点（含路径压缩）
    def find(self, index: int) -> int:
        if self.parent[index] != index:
            self.parent[index] = self.find(self.parent[index])
        return self.parent[index]


X = solution4
#print(X)
```

```
uf = UnionFindSet(X.shape[0])
# clustering
tolerance = 3 # TBD

for i in range(X.shape[0]):
    for j in range(i+1, X.shape[0]):
        # 两两比较距离
        if norm(X[i] - X[j]) < tolerance:
            uf.union(i, j)
# 聚类结果
label_result4 = list()
for i in range(X.shape[0]):
    label_result4.append(uf.find(i)) # 第i条数据的聚类类别

# Visualization
print(label_result4)
```

In [ ]:
```
print(label_generation4)
```

In [ ]:
```
#orginal data
x=data_generation4[:,0]
y=data_generation4[:,1]
c=label_generation4

fig, ax = plt.subplots()
scatter = ax.scatter(x, y, c=c, cmap=plt.cm.RdYlBu)
plt.show()
```

In [ ]:
```
#predict data
x=data_generation4[:,0]
y=data_generation4[:,1]
c=label_result4

fig, ax = plt.subplots()
scatter = ax.scatter(x, y, c=c, cmap=plt.cm.RdYlBu)
plt.show()
```

In [ ]:
```
#test data_generation1
options = {
    "lambda":0.05,
    "delta":1e-4,
    "tol":1e-2,
    "isprint":True
}
solution_1,x1_lst,gradient1_lst = invoke_AGM(data_generation1,options)
```

In [ ]:
```
X = solution_1
#print(X)
uf = UnionFindSet(X.shape[0])
# clustering
tolerance = 4 # TBD

for i in range(X.shape[0]):
    for j in range(i+1, X.shape[0]):
        # 两两比较距离
        if norm(X[i] - X[j]) < tolerance:
            uf.union(i, j)
```

```
# 聚类结果
label_result1 = list()
for i in range(X.shape[0]):
    label_result1.append(uf.find(i)) # 第i条数据的聚类类别

# Visualization
print(label_result1)
print(set(label_result1))
```

In [ ]:

```
#orginal data
x=data_generation1[:,0]
y=data_generation1[:,1]
c=label_generation1

fig, ax = plt.subplots()
scatter = ax.scatter(x, y, c=c, cmap=plt.cm.RdYlBu)
plt.show()
```

In [ ]:

```
#predict data
x=data_generation1[:,0]
y=data_generation1[:,1]
c=label_result1

fig, ax = plt.subplots()
scatter = ax.scatter(x, y, c=c, cmap=plt.cm.RdYlBu)
plt.show()
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Dec 12 14:29:25 2021

@author: Evelyn
"""
#如果要运行程序，请把本文件放到解压的数据文件下

from scipy.io import loadmat
import numpy as np
from scipy.sparse import csc_matrix
import time

# --------------------my version(slow version)-------------------------------
def function(x):
    '''
    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of function 'f_clust'.

    Returns
    -------
    type: float
        DESCRIPTION: The solution of f_clust.

    PS: A, row, col, lamd, delta all store as global variables.
    '''
    f1=[np.power(np.linalg.norm(x[i*row:(i+1)*row]-A[:, i].reshape(-1, 1)),2) for i in range(col)]
    f2=0
    for i in range(col):
        for j in range(i+1,col):
            n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
            f2+=0.5/delta*np.power(n,2) if n<=delta else n-delta*0.5
    return 0.5*sum(f1)+lamd*f2

def gradient(x):
    '''

    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of gradient function.
    Returns
    -------
    type: numpy.ndarray
          shape: (row*col,1)
          DESCRIPTION: The solution of gradient function.

    '''

    solution1=np.asarray(np.concatenate(list(map(lambda i:x[i*row:(i+1)*row]-A[:, i].reshape(-1, 1),range(col))),axis=0))
    solution2=np.zeros([row*col,1])
    for i in range(col):
        for j in range(col):
            n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
            solution2[i*row:(i+1)*row]+=(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])/delta
```

```
    if n<=delta else (x[i*row:(i+1)*row]-x[j*row:(j+1)*row])/n
60      return solution1+lamd*solution2
61
62  def hessian(x):
63      '''
64
65      Parameters
66      ----------
67      x : numpy.ndarray
68          shape: (row*col,1)
69          DESCRIPTION: The independent variable of hessian funciton
70      Returns
71      -------
72      type: numpy.ndarray
73            shape: (row*col,row*col)
74            DESCRIPTION: The solution of hessian function.
75
76      '''
77      mat=[]
78      hub={}
79      for i in range(col):
80          hubij=np.zeros([row,row])
81          for j in range(col):
82              n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
83              hubij+=np.identity(row)/delta if n<=delta else np.identity(row)/n-
    (x[i*row:(i+1)*row]-x[j*row:(j+1)*row])@(x[i*row:(i+1)*row]-x[j*row:
    (j+1)*row]).T/np.power(n,3)
84          hub[i]=hubij
85
86      for i in range(col):
87          r=[]
88          for j in range(col):
89              if i==j:
90                  r.append(np.identity(row)+lamd*hub[i])
91              else:
92                  n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
93                  hubij=np.identity(row)/delta if n<=delta else np.identity(row)/n-
    (x[i*row:(i+1)*row]-x[j*row:(j+1)*row])@(x[i*row:(i+1)*row]-x[j*row:
    (j+1)*row]).T/np.power(n,3)
94                  r.append(-lamd*hubij)
95          mat.append(np.concatenate(r,axis=1))
96      return np.concatenate(mat,axis=0)
97
98  # ---------------------------new version(quick)------------------------------
99  def function1(x):
100     '''
101     Parameters
102     ----------
103     x : numpy.ndarray
104         shape: (row*col,1)
105         DESCRIPTION: The independent variable of function 'f_clust'.
106
107     Returns
108     -------
109     type: float
110         DESCRIPTION: The solution of f_clust.
111
112     PS: A, row, col, lamd, delta all store as global variables.
113     '''
114     f1=[np.power(np.linalg.norm(x[i*row:(i+1)*row]-A[:, i].reshape(-1, 1)),2) for i
    in range(col)]
115     f2=0
```

```
    for i in range(col-1):
        xi_xj_norm=np.asarray(np.linalg.norm([x[i*row:(i+1)*row]-x[j*row:(j+1)*row] for j in range(i+1,col)],axis=1,ord=2))
        mask=xi_xj_norm<=delta
        f2+=sum(xi_xj_norm[mask]**2*0.5/delta)
        f2+=sum(xi_xj_norm[~mask]-delta*0.5)
    return 0.5*sum(f1)+lamd*f2

def gradient1(x):
    '''

    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of gradient function.
    Returns
    -------
    type: numpy.ndarray
          shape: (row*col,1)
          DESCRIPTION: The solution of gradient function.

    '''

    solution1=np.asarray(np.concatenate(list(map(lambda i:x[i*row:(i+1)*row]-A[:,i].reshape(-1, 1),range(col))),axis=0))
    solution2=np.zeros([row*col,1])
    for i in range(col):
        xi_xj=np.asarray([x[i*row:(i+1)*row]-x[j*row:(j+1)*row] for j in range(col)]).reshape((col,row),order='C')
        m=np.all(np.equal(xi_xj, 0), axis=1)
        xi_xj=xi_xj[~m] #去掉0项
        xi_xj_norm=np.linalg.norm(xi_xj,axis=1,ord=2)
        mask=xi_xj_norm<=delta
        solution2[i*row:(i+1)*row]+=(np.sum(xi_xj[mask]/delta,axis=0)).reshape(row,1)
        solution2[i*row:(i+1)*row]+=(np.sum(xi_xj[~mask]/np.expand_dims(xi_xj_norm[~mask],axis=1),axis=0)).reshape(row,1)
    return solution1+lamd*solution2


def hessian1(x):
    '''

    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of hessian funciton
    Returns
    -------
    type: numpy.ndarray
          shape: (row*col,row*col)
          DESCRIPTION: The solution of hessian function.

    '''
    mat=[]
    hub={}
    for i in range(col):
        hubij=np.zeros([row,row])
        xi_xj=np.asarray([x[i*row:(i+1)*row]-x[j*row:(j+1)*row] for j in range(col)]).reshape((col,row),order='C')
        m=np.all(np.equal(xi_xj, 0), axis=1)
```

```
        xi_xj=xi_xj[~m] #去掉0项
        xi_xj_norm=np.linalg.norm(xi_xj,axis=1,ord=2)
        mask=xi_xj_norm<=delta
        hubij+=np.sum([(np.identity(row)/delta if judge==True else np.identity(row)/xi_xj_norm[j]-xi_xj[j].reshape(row,1)@xi_xj[j].reshape(1,row)/(xi_xj_norm[j]**3)) for j,judge in enumerate(mask)],axis=0)
        hub[i]=hubij

    for i in range(col):
        r=[]
        for j in range(col):
            if i==j:
                r.append(np.identity(row)+lamd*hub[i])
            else:
                n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
                hubij=np.identity(row)/delta if n<=delta else np.identity(row)/n-(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])@(x[i*row:(i+1)*row]-x[j*row:(j+1)*row]).T/np.power(n,3)
                r.append(-lamd*hubij)
        mat.append(np.concatenate(r,axis=1))
    return np.concatenate(mat,axis=0)

# 测试以上函数
# print(gradient(x))
# print(hessian(x))
# print(function(x))
# xk=gradient(x)
# B=hessian(x)
# print((xk.T@B@xk)[0][0])

class Newton_CG:
    def __init__(self,func,grad,hess,x0,tol,maxiter):
        self.func=func
        self.grad=grad
        self.hess=hess
        self.x0=x0    #initial point
        self.x_sequence=[x0]  #sequence of {x_k}
        self.norm_grad_x_sequence=[np.linalg.norm(self.grad(x0))] #sequence of {||grad_x_k||}
        self.cg_max=10  #the max iterations of CG
        self.epsilon=tol #the tolerance of newton-CG
        self.maxiter=maxiter  #the max iteration of newton-CG

    def backtracking(self,x,d):
        alpha=s
        x_1=x+alpha*d
        while self.func(x_1)>self.func(x)+gamma*alpha*np.dot(self.grad(x).T,d)[0,0]:
            alpha=sigma*alpha
            x_1=x+alpha*d
        return alpha

    def cal_dk(self,xk):
        B=self.hess(xk)
        vj=np.zeros([row*col,1])
        rj=self.grad(xk)
        pj=-rj
        cg_tol=min(1,np.power(np.linalg.norm(rj),0.1))*np.linalg.norm(rj)
        dk=-rj
        for j in range(0,self.cg_max):
            judge=(pj.T@B@pj)[0][0]
            if judge<=0:
```

```
                dk=vj if j>0 else dk
                break
            norm_rj=np.linalg.norm(rj) #the norm of old rj
            sigmaj=np.power(norm_rj,2)/judge
            vj+=sigmaj*pj
            rj+=sigmaj*B@pj #new rj
            norm_rj_1=np.linalg.norm(rj) #the norm of new rj
            if norm_rj_1<cg_tol:
                dk=vj
                break
            betaj_1=np.power(norm_rj_1,2)/np.power(norm_rj,2)
            pj=-rj+betaj_1*pj
        return dk


    def Newton_CG(self):
        xk=self.x0
        print("INITIAL OBJ:",self.func(x0))
        for k in range(self.maxiter):
            dk=self.cal_dk(xk)
            alphak=self.backtracking(xk,dk)
            xk=xk+alphak*dk
            self.x_sequence.append(xk)
            norm_grad_xk=np.linalg.norm(self.grad(xk))
            self.norm_grad_x_sequence.append(norm_grad_xk)
            print("ITER:{:0>6d}".format(len(self.x_sequence)-1),end='    ')
            # print("OBJ:",self.func(xk),end='    ')
            print("NORM:",norm_grad_xk,end='\n')
            if norm_grad_xk<=self.epsilon:
                return xk

T1 = time.time() #计时

# 测试Newton_CG

#设置一些全局变量，可自行修改
lamd=0.5
delta=0.001
s=1
sigma=0.5
gamma=0.1
maxiter=10000

#稀疏矩阵
file='.\wine\wine_data.mat'
dataA=loadmat(file,mat_dtype=True)
file='.\wine\wine_label.mat'
datab=loadmat(file,mat_dtype=True)
A=dataA['A']
b=datab['b']
row=A.shape[0]
col=A.shape[1]
x0=np.concatenate(list(map(lambda i:A.getcol(i).toarray(), range(col))),axis=0)
tol=0.001
maxiter=10000

# ----------------测试wine
lamda=0.05
delta=1e-4
temp=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
```

```
temp.Newton_CG()
T2 = time.time()
print('程序运行时间:{:.2f}分' .format((T2 - T1)/60))
np.savez('.\\compare\\wine_no_weight',temp.x_sequence, temp.norm_grad_x_sequence)

# ---------------测试一个自己建的很小的数据集
# A=np.array([[0.,0.5,1.,0.5,1.,0.5]]*4)
# col=A.shape[1]
# row=A.shape[0]
# x0=A.reshape([-1,1],order='F')
# temp=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
# temp.Newton_CG()

# --------------测试人工生成数据集
# data=np.loadtxt('.\\op_data\\200_circle_2.txt')
# A=data[0:2]
# x0=A.reshape([-1,1],order='F')
# row=A.shape[0]
# col=A.shape[1]
# cir_200=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
# x_star=cir_200.Newton_CG()
# T2 = time.time()
# print('程序运行时间:{:.2f}分' .format((T2 - T1)/60))
# np.savez('cir_200',cir_200.x_sequence,cir_200.norm_grad_x_sequence)

# data=np.loadtxt('.\\op_data\\200_dot_3.txt')
# A=data[0:2]
# x0=A.reshape([-1,1],order='F')
# row=A.shape[0]
# col=A.shape[1]
# dot_200=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
# dot_200.Newton_CG()
# T2 = time.time()
# print('程序运行时间:{:.2f}分' .format((T2 - T1)/60))
# np.savez('dot_200',dot_200.x_sequence,dot_200.norm_grad_x_sequence)

# T1 = time.time()
# data=np.loadtxt('.\\op_data\\200_moon_2.txt')
# A=data[0:2]
# x0=A.reshape([-1,1],order='F')
# row=A.shape[0]
# col=A.shape[1]
# moon_200=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
# moon_200.Newton_CG()
# T2 = time.time()
# print('程序运行时间:{:.2f}分' .format((T2 - T1)/60))
# np.savez('moon_200',moon_200.x_sequence,moon_200.norm_grad_x_sequence)

# T1 = time.time()
# data=np.loadtxt('.\\op_data\\200_random_4.txt')
# A=data[0:2]
# x0=A.reshape([-1,1],order='F')
# row=A.shape[0]
# col=A.shape[1]
# random_200=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
# random_200.Newton_CG()
# T2 = time.time()
# print('程序运行时间:{:.2f}分' .format((T2 - T1)/60))
# np.savez('random_200',random_200.x_sequence,random_200.norm_grad_x_sequence)

```

```
349
350 # T1 = time.time()
351 # data=np.loadtxt('.\\op_data\\200_dot_6.txt')
352 # A=data[0:2]
353 # x0=A.reshape([-1,1],order='F')
354 # row=A.shape[0]
355 # col=A.shape[1]
356 # dot_200_6=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
357 # dot_200_6.Newton_CG()
358 # T2 = time.time()
359 # print('程序运行时间:{:.2f}分' .format((T2 - T1)/60))
360 # np.savez('dot_200_6',dot_200_6.x_sequence, dot_200_6.norm_grad_x_sequence)
361
362 # T1 = time.time()
363 # lamda_list=[0.05,0.1,0.2,0.5,0.8,1,3,5,7.5,10.]
364 # data=np.loadtxt('.\\op_data\\200_dot_6.txt')
365 # A=data[0:2]
366 # x0=A.reshape([-1,1],order='F')
367 # row=A.shape[0]
368 # col=A.shape[1]
369 # tol=0.1
370 # for i in lamda_list:
371 #     lamd=i
372 #     dot_2000=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
373 #     dot_2000.Newton_CG()
374 #     np.savez('dot_200_6_'+str(i),dot_2000.x_sequence,dot_2000.norm_grad_x_sequence)
375 # T2 = time.time()
376 # print('程序运行时间:{:.2f}分' .format((T2 - T1)/60))
377 # np.savez('dot_2000',dot_2000.x_sequence,dot_2000.norm_grad_x_sequence)
378
379 # ------------------------测试真实数据
380 # file='.\\vowel\\vowel_data.mat'
381 # dataA=loadmat(file,mat_dtype=True)
382 # A=dataA['A']
383 # row=A.shape[0]
384 # col=A.shape[1]
385 # x0=np.concatenate(list(map(lambda i:A.getcol(i).toarray(), range(col))),axis=0)
386 # tol=0.001
387 # vowel=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
388 # vowel.Newton_CG()
389 # T2 = time.time()
390 # print('程序运行时间:{:.2f}分' .format((T2 - T1)/60))
391 # np.savez('vowel',vowel.x_sequence, vowel.norm_grad_x_sequence)
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Dec 26 22:49:35 2021

@author: Evelyn
"""
from scipy.io import loadmat
import numpy as np
from scipy.sparse import csc_matrix
import time

def get_weighted_sparse(k):
    for i in range(col-1):
        w=np.array([np.exp(-v*np.power(np.linalg.norm((A[:, i].reshape(-1,1)-A[:, j].reshape(-1,1)).toarray(),ord=2),2)) for j in range(i+1,col)])
        if k>=col-i-1:
            k=col-i-1
        index=np.argpartition(w,-k)[-k:]
        for j in range(k):
            Weight[(i,index[j]+i+1)]=w[index[j]]
            Weight[(index[j]+i+1,i)]=w[index[j]]
    return Weight

def get_weighted(k):
    for i in range(col-1):
        w=np.array([np.exp(-v*np.power(np.linalg.norm((A[:, i].reshape(-1,1)-A[:, j].reshape(-1,1))),2)) for j in range(i+1,col)])
        if k>=col-i-1:
            k=col-i-1
        index=np.argpartition(w,-k)[-k:]
        for j in range(k):
            Weight[(i,index[j]+i+1)]=w[index[j]]
            Weight[(index[j]+i+1,i)]=w[index[j]]
    return Weight

# --------------------my version(slow version)-----------------------------
def function(x):
    '''
    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of function 'f_clust'.

    Returns
    -------
    type: float
        DESCRIPTION: The solution of f_clust.

    PS: A, row, col, lamd, delta all store as global variables.
    '''
    f1=[np.power(np.linalg.norm(x[i*row:(i+1)*row]-A[:, i].reshape(-1, 1)),2) for i in range(col)]
    f2=0
    for i in range(col):
        for j in range(i+1,col):
            n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
            f2+=Weight.get((i,j),0)*0.5/delta*np.power(n,2) if n<=delta else Weight.get((i,j),0)*(n-delta*0.5)
    return 0.5*sum(f1)+lamd*f2

def gradient(x):
```

```
    '''

    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of gradient function.
    Returns
    -------
    type: numpy.ndarray
          shape: (row*col,1)
          DESCRIPTION: The solution of gradient function.

    '''

    solution1=np.asarray(np.concatenate(list(map(lambda i:x[i*row:(i+1)*row]-A[:,i].reshape(-1, 1),range(col))),axis=0))
    solution2=np.zeros([row*col,1])
    for i in range(col):
        for j in range(col):
            n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
            solution2[i*row:(i+1)*row]+=Weight.get((i,j),0)*(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])/delta if n<=delta else Weight.get((i,j),0)*(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])/n
    return solution1+lamd*solution2

def hessian(x):
    '''

    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of hessian funciton
    Returns
    -------
    type: numpy.ndarray
          shape: (row*col,row*col)
          DESCRIPTION: The solution of hessian function.

    '''
    mat=[]
    hub={}
    for i in range(col):
        hubij=np.zeros([row,row])
        for j in range(col):
            n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
            hubij+=Weight.get((i,j),0)*np.identity(row)/delta if n<=delta else Weight.get((i,j),0)*np.identity(row)/n-(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])@(x[i*row:(i+1)*row]-x[j*row:(j+1)*row]).T/np.power(n,3)
        hub[i]=hubij

    for i in range(col):
        r=[]
        for j in range(col):
            if i==j:
                r.append(np.identity(row)+lamd*hub[i])
            else:
                n=np.linalg.norm(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])
                hubij=Weight.get((i,j),0)*np.identity(row)/delta if n<=delta else Weight.get((i,j),0)*np.identity(row)/n-(x[i*row:(i+1)*row]-x[j*row:(j+1)*row])@(x[i*row:(i+1)*row]-x[j*row:(j+1)*row]).T/np.power(n,3)
```

```
                r.append(-lamd*hubij)
        mat.append(np.concatenate(r,axis=1))
    return np.concatenate(mat,axis=0)

class Newton_CG:
    def __init__(self,func,grad,hess,x0,tol,maxiter):
        self.func=func
        self.grad=grad
        self.hess=hess
        self.x0=x0    #initial point
        self.x_sequence=[x0]  #sequence of {x_k}
        self.norm_grad_x_sequence=[np.linalg.norm(self.grad(x0))] #sequence of {||grad_x_k||}
        self.cg_max=10  #the max iterations of CG
        self.epsilon=tol #the tolerance of newton-CG
        self.maxiter=maxiter  #the max iteration of newton-CG

    def backtracking(self,x,d):
        alpha=s
        x_1=x+alpha*d
        while self.func(x_1)>self.func(x)+gamma*alpha*np.dot(self.grad(x).T,d)[0,0]:
            alpha=sigma*alpha
            x_1=x+alpha*d
        return alpha

    def cal_dk(self,xk):
        B=self.hess(xk)
        vj=np.zeros([row*col,1])
        rj=self.grad(xk)
        pj=-rj
        cg_tol=min(1,np.power(np.linalg.norm(rj),0.1))*np.linalg.norm(rj)
        dk=-rj
        for j in range(0,self.cg_max):
            judge=(pj.T@B@pj)[0][0]
            if judge<=0:
                dk=vj if j>0 else dk
                break
            norm_rj=np.linalg.norm(rj) #the norm of old rj
            sigmaj=np.power(norm_rj,2)/judge
            vj+=sigmaj*pj
            rj+=sigmaj*B@pj #new rj
            norm_rj_1=np.linalg.norm(rj) #the norm of new rj
            if norm_rj_1<cg_tol:
                dk=vj
                break
            betaj_1=np.power(norm_rj_1,2)/np.power(norm_rj,2)
            pj=-rj+betaj_1*pj
        return dk


    def Newton_CG(self):
        xk=self.x0
        print("INIT OBJ:",self.func(x0))
        for k in range(self.maxiter):
            dk=self.cal_dk(xk)
            alphak=self.backtracking(xk,dk)
            xk=xk+alphak*dk
            self.x_sequence.append(xk)
            norm_grad_xk=np.linalg.norm(self.grad(xk))
            self.norm_grad_x_sequence.append(norm_grad_xk)
            print("ITER:{:0>6d}".format(len(self.x_sequence)-1),end='    ')
```

```
            print("OBJ:",self.func(xk),end='    ')
            print("NORM:",norm_grad_xk,end='\n')
            if norm_grad_xk<=self.epsilon:
                return xk

#读取数据，只用wine数据进行了测试，其他可自行修改
file='.\wine\wine_data.mat'
dataA=loadmat(file,mat_dtype=True)

file='.\wine\wine_label.mat'
datab=loadmat(file,mat_dtype=True)

A=dataA['A']
b=datab['b']

#设置一些全局变量，可自行修改
lamd=0.05
delta=0.001
row=A.shape[0]
col=A.shape[1]
Weight={}
s=1
sigma=0.5
gamma=0.1
v=0.5

T1 = time.time() #计时

# 测试Newton_CG(Weighted 版本)
# ----------------测试wine
x0=np.concatenate(list(map(lambda i:A.getcol(i).toarray(), range(col))),axis=0)
tol=0.001
delta=1e-4
maxiter=10000

Weight=get_weighted_sparse(5)
temp=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
temp.Newton_CG()
T2 = time.time()
print('程序运行时间:{:.2f}分' .format((T2 - T1)/60))
np.savez('.\\compare\\wine_weight',temp.x_sequence, temp.norm_grad_x_sequence)

# ---------------测试一个自己建的很小的数据集
# A=np.array([[0.,0.5,1.,0.5,1.,0.5]]*4)
# col=A.shape[1]
# row=A.shape[0]
# Weighted=get_weighted(3)
# x0=A.reshape([-1,1])
# temp=Newton_CG(function, gradient, hessian, x0, tol, maxiter)
# temp.Newton_CG()
```

```python
import numpy as np
from numpy.linalg import norm
import matplotlib.pyplot as plt
import os
from scipy.io import loadmat
from scipy.sparse import csc_matrix
import time

# load data
data_200_dot_3 = np.loadtxt('op_data/200_dot_3.txt')
data_200_dot_6 = np.loadtxt('op_data/200_dot_6.txt')
data_2000_dot_3 = np.loadtxt('op_data/2000_dot_6.txt')
wine_data_mat = loadmat("wine_data.mat",mat_dtype=True)
wine_label_mat = loadmat("wine_label.mat",mat_dtype=True)
wine_data=wine_data_mat['A']
wine_label=wine_label_mat['b']


def function(x):
    '''
    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of function 'f_clust'.

    Returns
    -------
    type: float
        DESCRIPTION: The solution of f_clust.

    PS: A, row, col, lamd, delta all store as global variables.
    '''
    f1=[np.power(np.linalg.norm(x[i*row:(i+1)*row]-A[:, i].reshape(-1, 1)),2) for i in range(col)]
    f2=0
    for i in range(col-1):
        xi_xj_norm=np.asarray(np.linalg.norm([x[i*row:(i+1)*row]-x[j*row:(j+1)*row] for j in range(i+1,col)],axis=1,ord=2))
        mask=xi_xj_norm<=delta
        f2+=sum(xi_xj_norm[mask]**2*0.5/delta)
        f2+=sum(xi_xj_norm[~mask]-delta*0.5)
    return 0.5*sum(f1)+lamd*f2


def gradient(x):
    '''

    Parameters
    ----------
    x : numpy.ndarray
        shape: (row*col,1)
        DESCRIPTION: The independent variable of gradient function.
    Returns
    -------
    type: numpy.ndarray
          shape: (row*col,1)
          DESCRIPTION: The solution of gradient function.

    '''

```

```
    solution1=np.asarray(np.concatenate(list(map(lambda i:x[i*row:(i+1)*row]-A[:,i].reshape(-1, 1),range(col))),axis=0))
    solution2=np.zeros([row*col,1])
    for i in range(col):
        xi_xj=np.asarray([x[i*row:(i+1)*row]-x[j*row:(j+1)*row] for j in range(col)]).reshape((col,row),order='C')
        m=np.all(np.equal(xi_xj, 0), axis=1)
        xi_xj=xi_xj[~m] #去掉0项
        xi_xj_norm=np.linalg.norm(xi_xj,axis=1,ord=2)
        mask=xi_xj_norm<=delta
        solution2[i*row:(i+1)*row]+=(np.sum(xi_xj[mask]/delta,axis=0)).reshape(row,1)
        solution2[i*row:(i+1)*row]+=(np.sum(xi_xj[~mask]/np.expand_dims(xi_xj_norm[~mask],axis=1),axis=0)).reshape(row,1)
    return solution1+lamd*solution2


def plot_convergence_figure(obj, x_lst, gradient_lst, x_init):

    assert len(x_lst) == len(gradient_lst)
    iteration_num = len(gradient_lst)

    #The norm of the gradient VS #iterations
    plt.figure(dpi = 300, figsize=(6, 6.5))
    ax1 = plt.subplot(2, 1, 1)
    scatter = ax1.scatter(range(iteration_num), gradient_lst, s = 5)
    plt.title("The norm of the gradient VS #Iterations")
    ax1.set_xlabel('#iterations')
    ax1.set_ylabel('gradient norm')

    #Relative error VS #Iterations
    ax2 = plt.subplot(2, 1, 2)
    best_function_value = obj(x_lst[-1])
    scatter = ax2.scatter(range(iteration_num), [abs(obj(each) - best_function_value) / (max(1, abs(best_function_value))) for each in x_lst], s = 5)
    plt.title("Relative error VS #Iterations")
    ax2.set_xlabel('#iterations')
    ax2.set_ylabel('relative error')

    plt.tight_layout()
    plt.show()



class UnionFindSet:
    def __init__(self, n):
        self.parent = list(range(n))

    #合并index1和index2所属集合
    def union(self, index1: int, index2: int):
        self.parent[self.find(index2)] = self.find(index1)

    #查找index结点的父结点（含路径压缩）
    def find(self, index: int) -> int:
        if self.parent[index] != index:
            self.parent[index] = self.find(self.parent[index])
        return self.parent[index]


def BFGS(func, grad, x0, options):
    epsilon = options['tol']
    maxiter = options['maxiter']
```

```
117     k = 0  #number of iterations
118     n = np.shape(x0)[0]
119     Bk = np.eye(n) #initial Hessian matrix is I
120     alpha = s
121     x_lst = [x0]
122     norm_grad_lst = [norm(grad(x0))]
123 
124 
125     while k < maxiter:
126         gk1 = grad(x0)
127         if norm(gk1) < epsilon:
128             break
129         dk = -1.0*np.linalg.solve(Bk,gk1)
130         while func(x0 + alpha * dk) > func(x0) + gamma * alpha * np.dot(gk1.T, dk)
    [0,0]:
131             alpha = sigma * alpha
132 
133         x1 = x0 + alpha * dk
134         x_lst.append(x1)
135         norm_gk1 = norm(grad(x1))
136         norm_grad_lst.append(norm_gk1)
137         print("ITER:{:0>6d}".format(len(x_lst)-1),end='    ')
138         print("OBJ:", func(x1),end='    ')
139         print("NORM:", norm_gk1,end='\n')
140 
141         #BFGS
142         sk = x1 - x0
143         yk = gk1 - grad(x0)
144 
145         if np.dot(sk.T,yk) > 0:
146             Bs = np.dot(Bk,sk)
147             ys = np.dot(yk.T,sk)
148             sBs = np.dot(np.dot(sk.T,Bk),sk)
149             Bk = Bk - Bs.reshape((n,1))*Bs/sBs + yk.reshape((n,1))*yk/ys
150 
151         k += 1
152         x0 = x1
153     x = x0
154     return x_lst ,norm_grad_lst, x, k
155 
156 
157 def LBFGS(fun, grad, x0, options):
158     epsilon = options['tol']
159     maxiter = options['maxiter']
160     rho = 0.55
161     sigma = 0.4
162 
163     H0 = np.eye(np.shape(x0)[0])
164 
165     x_lst = [x0]
166     norm_grad_lst = [norm(grad(x0))]
167 
168     #s和y用于保存近期m个，这里m取6
169     s = []
170     y = []
171     m = 6
172 
173     k = 1
174     gk = np.mat(grad(x0))#计算梯度
175     dk = -H0 * gk
176     while (k < maxiter):
```

```
        gk1 = grad(x0)
        if norm(gk1) < epsilon:
            break
        n = 0
        mk = 0
        gk = np.mat(grad(x0))#计算梯度
        while (n < 20):
            newf = fun(x0 + rho ** n * dk)
            oldf = fun(x0)
            if (newf < oldf + sigma * (rho ** n) * (gk.T * dk)[0, 0]):
                mk = n
                break
            n = n + 1

        #LBFGS校正
        x = x0 + rho ** mk * dk
        #print x

        #保留m个
        if k > m:
            s.pop(0)
            y.pop(0)

        #计算最新的
        sk = x - x0
        yk = grad(x) - gk

        s.append(sk)
        y.append(yk)

        #two-loop的过程
        t = len(s)
        qk = grad(x)
        a = []
        for i in range(t):
            alpha = (s[t - i - 1].T * qk) / (y[t - i - 1].T * s[t - i - 1])
            qk = qk - alpha[0, 0] * y[t - i - 1]
            a.append(alpha[0, 0])
        r = H0 * qk

        for i in range(t):
            beta = (y[i].T * r) / (y[i].T * s[i])
            r = r + s[i] * (a[t - i - 1] - beta[0, 0])


        if (yk.T * sk > 0):
            dk = -r

        k = k + 1
        x_lst.append(x)
        norm_gk1 = norm(grad(x))
        norm_grad_lst.append(norm_gk1)
        print("ITER:{:0>6d}".format(len(x_lst)-1),end='    ')
        print("OBJ:", fun(x),end='    ')
        print("NORM:", norm_gk1,end='\n')
        x0 = x

    x = x0
    return x_lst, norm_grad_lst, x,  k

```

```
237
238 #global variables
239 lamd = 0.5
240 delta = 0.001
241
242 #global variables for backtracking
243 s=1
244 sigma=0.5
245 gamma=0.1
246
247
248 options = {
249     "tol": 1e-3,
250     "maxiter": 1e5
251 }
252
253 # wine BFGS
254 A = wine_data
255 row = A.shape[0]
256 col = A.shape[1]
257 x0 = np.concatenate(list(map(lambda i:A.getcol(i).toarray(), range(col))),axis=0)
258
259
260 T1 = time.time()
261 temp_wine_1 = BFGS(function, gradient, x0, options)
262 T2 = time.time()
263 run_time_wine_1 = (T2-T1)/60
264
265 run_time_wine_1
266
267 plot_convergence_figure(function, temp_wine_1[0], temp_wine_1[1], x0)
268
269 # wine L-BFGS
270 T1 = time.time()
271 temp_wine_2 = LBFGS(function, gradient, x0, options)
272 T2 = time.time()
273 run_time_wine_2 = (T2-T1)/60
274
275 run_time_wine_2
276
277 plot_convergence_figure(function, temp_wine_2[0], temp_wine_2[1], x0)
278
279 # data_200_dot_3 BFGS
280 A = data_200_dot_3[0:2]
281 x0 = A.reshape([-1,1],order='F')
282 row = A.shape[0]
283 col = A.shape[1]
284
285 T1 = time.time()
286 x0=A.reshape([-1,1],order='F')
287 temp_200_3 = BFGS(function, gradient, x0, options)
288 T2 = time.time()
289 run_time_200_3 = (T2-T1)/60
290
291 run_time_200_3
292
293 plot_convergence_figure(function, temp_200_3[0], temp_200_3[1], x0)
294
295 X = temp_200_3[2].reshape([-1,2], order='F')
296 #print(X)
297 uf = UnionFindSet(X.shape[0])
```

```
298 # clustering
299 tolerance = 3 # TBD
300 
301 for i in range(X.shape[0]):
302     for j in range(i+1, X.shape[0]):
303         # 两两比较距离
304         if norm(X[i] - X[j]) < tolerance:
305             uf.union(i, j)
306 # 聚类结果
307 label_result_200_3 = list()
308 for i in range(X.shape[0]):
309     label_result_200_3.append(uf.find(i)) # 第i条数据的聚类类别
310 
311 #orginal data
312 x=data_200_dot_3[0, :]
313 y=data_200_dot_3[1, :]
314 c=label_result_200_3
315 
316 fig, ax = plt.subplots()
317 scatter = ax.scatter(x, y, c=c, cmap=plt.cm.RdYlBu)
318 plt.show()
319 
320 
321 #data_200_dot_6 BFGS
322 A = data_200_dot_6[0:2]
323 x0 = A.reshape([-1,1],order='F')
324 x_dot_200_6 = data_200_dot_6[0]
325 y_dot_200_6 = data_200_dot_6[1]
326 row = A.shape[0]
327 col = A.shape[1]
328 
329 T1 = time.time()
330 x0 = A.reshape([-1,1],order='F')
331 temp_200_6 = BFGS(function, gradient, x0, options)
332 T2 = time.time()
333 run_time_200_6 = (T2-T1)/60
334 
335 run_time_200_6
336 
337 plot_convergence_figure(function, temp_200_6[0], temp_200_6[1], x0)
338 
339 
340 X = temp_200_6[2].reshape([-1,2], order='F')
341 #print(X)
342 uf = UnionFindSet(X.shape[0])
343 # clustering
344 tolerance = 6 # TBD
345 
346 for i in range(X.shape[0]):
347     for j in range(i+1, X.shape[0]):
348         # 两两比较距离
349         if norm(X[i] - X[j]) < tolerance:
350             uf.union(i, j)
351 # 聚类结果
352 label_result_200_6 = list()
353 for i in range(X.shape[0]):
354     label_result_200_6.append(uf.find(i)) # 第i条数据的聚类类别
355 
356 fig, ax = plt.subplots()
357 scatter = ax.scatter(x_dot_200_6, y_dot_200_6, c=label_result_200_6,
```

```
    cmap=plt.cm.RdYlBu)
358 plt.show()
```

```
#!/usr/bin/env python
# coding: utf-8

# In[11]:


#Load libraries
import pandas as pd
import numpy as np
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from scipy.io import loadmat
from scipy.sparse import csc_matrix
import tensorflow as tf
import matplotlib.pyplot as plt
from numpy.linalg import norm
import random
import math
import time


# In[2]:


#load data 200 two-dimensional points, 6 cluster centers
data_dot_200_6 = np.loadtxt(r"C:\Users\HAOHAN\Desktop\优化
\data\op_data\200_dot_6.txt")
x_dot_200_6 = data_dot_200_6[0]
y_dot_200_6 = data_dot_200_6[1]
label_dot_200_6 = data_dot_200_6[2]

fig, ax = plt.subplots(dpi=200)
plt.title("Artificial Data Set(200 two-dimensional points, 6 cluster centers)")
scatter = ax.scatter(x_dot_200_6, y_dot_200_6, c=label_dot_200_6, cmap=plt.cm.RdYlBu)
plt.show()


# In[61]:


#load wine data
file=r"C:\Users\HAOHAN\Desktop\优化\data\datasets\datasets\wine\wine_data.mat"
data_wine = loadmat(file,mat_dtype=True)["A"].todense().T


# In[6]:


class UnionFindSet:
    def __init__(self, n):
        self.parent = list(range(n))

    #合并index1和index2所属集合
    def union(self, index1: int, index2: int):
        self.parent[self.find(index2)] = self.find(index1)

    #查找index结点的父结点（含路径压缩）
    def find(self, index: int) -> int:
        if self.parent[index] != index:
```

```
            self.parent[index] = self.find(self.parent[index])
        return self.parent[index]


# In[136]:


def f(x,a,lambda1,delta):
    f_1 = np.sum((norm(x[i]-a[i]))**2 for i in range(len(x)))
    x_diff = list()
    f_2 = 0
    for i in range(len(a)):
        # for j in range(i+1,len(a)):
        #   x_diff.append(x[i]-x[j])
        xi_diff = x[i] - x[i+1:]
        xi_diff_norm = norm(xi_diff, ord=2, axis=1)
        mask = xi_diff_norm <= delta
        f_2 += np.sum(xi_diff_norm[mask] ** 2 / (2*delta))
        f_2 += np.sum(xi_diff_norm[~mask] - delta/2)

    #f_2 = phi(x_diff,delta)
    return 1/2*(f_1) + lambda1 * f_2

def g(x,a,lambda1,delta):
    grad_1 = x - a
    grad_2 = np.zeros(x.shape)
    for i in range(x.shape[0]):
        xi_diff = x[i] - x[i+1:]
        xi_diff_norm = norm(xi_diff, ord=2, axis=1)
        mask = xi_diff_norm <= delta
        grad_2[i] += lambda1 * np.sum(xi_diff[mask] / delta, axis=0)
        grad_2[i] += lambda1 * np.sum(xi_diff[~mask] /
np.expand_dims(xi_diff_norm[~mask], axis=1), axis=0)

    return grad_1 + grad_2

def g_MiniBatch(x,a,lambda1,delta,batch_size):
    data_index_array = np.random.randint(0, x.shape[0], batch_size)
    grad_1 = x - a
    grad_2 = np.zeros(x.shape)
    for i in data_index_array:
        xi_diff = x[i] - x[i+1:]
        xi_diff_norm = norm(xi_diff, ord=2, axis=1)
        mask = xi_diff_norm <= delta
        grad_2[i] += lambda1 * np.squeeze(np.asarray(np.sum(xi_diff[mask] / delta,
axis=0)))
        grad_2[i] += lambda1 * np.squeeze(np.asarray(np.sum(xi_diff[~mask] /
np.expand_dims(xi_diff_norm[~mask], axis=1), axis=0)))

    return grad_1 + grad_2


# In[279]:


def GradientMethod(obj,grad,x,options,a,MiniBatch=False):
    T1 = time.time() #计时
    global lambda1
    global delta
    lambda1 = options['lambda']
    delta = options['delta']
    alpha = options['alpha']
```

```
    tol = options['tol']
    isprint = options['isprint']
    max_iter = options["max_iter"]
    if MiniBatch:
        batch_size = options["batch_size"]
        gradient = grad(x,a,lambda1,delta,batch_size)
    else:
        gradient = grad(x,a,lambda1,delta)
    x_lst = [x]
    gradient_lst = [norm(gradient)]
    k = 0
    while norm(gradient) >= tol and k < max_iter:
        k += 1
        x = x - alpha * gradient / k
        if MiniBatch: gradient = grad(x,a,lambda1,delta,batch_size)
        else: gradient = grad(x,a,lambda1,delta)
        if isprint and (k % 10000 == 1):
            print("Iteration:",k-
1,"obj:",obj(x,a,lambda1,delta),"norm_gradient:",norm(gradient), "time:
{:.2f}s".format(time.time() - T1))
        x_lst.append(x)
        gradient_lst.append(norm(gradient))
    print("Iteration:",k-
1,"obj:",obj(x,a,lambda1,delta),"norm_gradient:",norm(gradient))
    T2 = time.time() #计时
    print('程序运行时间:{:.4f}分' .format((T2 - T1)/60))
    return x,x_lst,gradient_lst


# In[14]:


def plot_convergence_figure(obj, x_lst, gradient_lst, options, x_init):
    lambda1 = options['lambda']
    alpha = options['alpha']

    assert len(x_lst) == len(gradient_lst)
    iteration_num = len(gradient_lst)

    #The norm of the gradient VS #iterations
    plt.figure(dpi = 300, figsize=(6, 6.5))
    ax1 = plt.subplot(2, 1, 1)
    scatter = ax1.scatter(range(iteration_num), gradient_lst, s = 5)
    plt.title("The norm of the gradient VS #Iterations")
    ax1.set_xlabel('#iterations')
    ax1.set_ylabel('gradient norm')

    #Relative error VS #Iterations
    ax2 = plt.subplot(2, 1, 2)
    best_function_value = obj(x_lst[-1],x_init,lambda1,delta)
    scatter = ax2.scatter(range(iteration_num), [abs(obj(each,x_init,lambda1,delta) -
best_function_value) / (max(1, abs(best_function_value))) for each in x_lst], s = 5)
    plt.title("Relative error VS #Iterations")
    ax2.set_xlabel('#iterations')
    ax2.set_ylabel('relative error')

    plt.tight_layout()
    plt.show()


# In[ ]:

```

```
177
178 data = data_dot_200_6.T[:,:2]
179 delta_value = 1e-4
180 solution_list_all, x_list_all, gradient_list_all, function_value_list_all = [], [],
    [], []
181 for batch_size in [1, 25, 50, 100, 200]:
182     options = {
183     "lambda":lambda_val,
184     "delta":delta_value,
185     "tol":1e-1,
186     "isprint":True,
187     "alpha": 1,
188     "max_iter": 30000,
189     "batch_size": batch_size
190     }
191     print(batch_size)
192     solution_list, x_list, gradient_list =
    GradientMethod(f,g_MiniBatch,data,options,data,MiniBatch=True)
193     tmp = []
194     for each in x_list:
195         tmp.append(f(each,data,lambda_val,delta_value))
196     function_value_list_all.append(tmp)
197     solution_list_all.append(solution_list)
198     x_list_all.append(x_list)
199     gradient_list_all.append(gradient_list)
200
201
202 # In[ ]:
203
204
205 plt.figure(dpi = 300)
206 k_list = [1, 25, 50, 100, 200]
207 for i in range(len(function_value_list_all)):
208     plt.plot([each * 50 for each in range(len(function_value_list_all[i]
    [10:5000:50]))], [np.sqrt(norm(each - min(function_value_list_all[i][10:5000:50])) /
    max(1, abs(min(function_value_list_all[i][10:5000:50])))) for each in
    function_value_list_all[i][10:5000:50]], label="k="+str(k_list[i]))
209 plt.legend()
210 plt.xlabel("#Iterations")
211 plt.ylabel("Relative Error")
212 plt.title("Relative Error VS #Iterations in artifitial dataset")
213 plt.show()
214
215
216 # In[224]:
217
218
219 X = solution_data_dot_200_6_SGD
220 #print(X)
221 uf = UnionFindSet(X.shape[0])
222 # clustering
223 tolerance = 0.7 # TBD
224
225 for i in range(X.shape[0]):
226     for j in range(i+1, X.shape[0]):
227         # 两两比较距离
228         if norm(X[i] - X[j]) < tolerance:
229             uf.union(i, j)
230 # 聚类结果
231 label_data_dot_200_6 = list()
232 for i in range(X.shape[0]):
233
```

```
    label_data_dot_200_6.append(uf.find(i)) # 第i条数据的聚类类别

# Visualization
print(len(set(label_data_dot_200_6)))

#predict data
fig, ax = plt.subplots()
scatter = ax.scatter(x_dot_200_6, y_dot_200_6, c=label_data_dot_200_6, cmap=plt.cm.RdYlBu)
plt.show()


# In[280]:


data = data_wine
delta = delta_value
solution_list_all_wine, x_list_all_wine, gradient_list_all_wine  = [], [], []
for batch_size in [1, 50, 100, 178]:
    options = {
    "lambda":lambda_val,
    "delta":delta_value,
    "tol":1e-1,
    "isprint":True,
    "alpha": 0.1,
    "max_iter": 20000,
    "batch_size": batch_size
    }
    print(batch_size)
    solution_list, x_list, gradient_list = GradientMethod(f,g_MiniBatch,data,options,data,MiniBatch=True)
    solution_list_all_wine.append(solution_list)
    x_list_all_wine.append(x_list)
    gradient_list_all_wine.append(gradient_list)


# In[281]:


function_value_list_all_wine = []
for i in range(len(x_list_all_wine)):
    tmp = []
    for each in x_list_all_wine[i]:
        tmp.append(f(each,data,lambda_val,delta_value))
    function_value_list_all_wine.append(tmp)


# In[285]:


plt.figure(dpi = 300)
k_list = [1, 50, 100, 178]
for i in range(len(function_value_list_all_wine)):
    plt.plot([each * 50 for each in range(len(function_value_list_all_wine[i][:20000:50]))], [np.sqrt(norm(each - min(function_value_list_all_wine[i][:20000:50])) / max(1, abs(min(function_value_list_all_wine[i][:20000:50])))) for each in function_value_list_all_wine[i][:20000:50]], label="k="+str(k_list[i]))
plt.legend()
plt.xlabel("#Iterations")
plt.ylabel("Relative Error")
plt.title("Relative Error VS #Iterations in wine dataset")
plt.show()
```

```
290
291
292 # In[289]:
293
294
295 plt.figure(dpi = 300)
296 k_list = [1, 50, 100, 178]
297 for i in range(len(function_value_list_all_wine)):
298     plt.plot([each * 50 for each in range(len(gradient_list_all_wine[i]
    [:20000:50]))], gradient_list_all_wine[i][:20000:50], label="k="+str(k_list[i]))
299 plt.legend()
300 plt.xlabel("#Iterations")
301 plt.ylabel("Gradient Norm")
302 plt.title("Gradient Norm VS #Iterations in wine dataset")
303 plt.show()
304
305
306 # In[ ]:
```