香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

The Chinese University of Hong Kong, Shenzhen
SDS · School of Data Science
Andre Milzarek · Fall Semester 2020/21

MDS 6106 – Introduction to Optimization

Final Project: Report

Name: ***Peng Deng*** Student ID: ***220041042***

Name: ***Yihang Li*** Student ID: ***220041006***

Name: ***Binbo Chen*** Student ID: ***220041052***

Name: ***Cheng Cheng*** Student ID: ***220041014***

Contents

1 Introduction

1.1 Background of Binary Classification Problems

For the binary classification problems with two classes $\mathcal{C}_1$ and $\mathcal{C}_2$, we assume that our training data consists of feature vectors $a_i \in \mathbb{R}^n, i \in \{1, 2, \ldots, m\}$ and corresponding class labels $z_i \in \{0, 1\}$ or $b_i = 2z_i - 1 \in \{-1, 1\}$. Here, each label b_i or z_i indicates whether the data point a_i belongs to the class $\mathcal{C}_1$ or $\mathcal{C}_2$.

The training data provides implicit information on how to distinguish the two classes and which elements to assign to which class. Based on this implicit information, we want to train a model that allows to assign new feature vectors correctly to one of the two classes. Therefore, our goal is to choose and learn a parametric model ℓ_θ which allows to separate the two classes. The model parameters θ in ℓ_θ have to be adapted to the training data such that as many feature vectors as possible are classified correctly by our trained model ℓ_θ. After the parameter estimation has been performed, the model can be used to predict the label and class of a new data point $a \in \mathbb{R}^n$ via evaluating the function $\ell_\theta(a)$

There are different possibilities on how to build such a model ℓ_θ. Support vector machines try to separate the two data classes $\mathcal{C}_1$ and $\mathcal{C}_2$ via a hyperplane, i.e, points on the left side of the hyperplane are modeled to belong to class $\mathcal{C}_1$ while points on the right side of the hyperplane should belong to class $\mathcal{C}_2$ (or vice versa). Hence, $\ell_\theta = \ell_{(x,y)}$ is chosen as a linear function

$$\ell_{(x,y)} : \mathbb{R}^{n+1} \to \mathbb{R}, \quad \ell_{(x,y)}(a) := a^\top x + y$$

and our task is to choose the hyperplane parameters $\theta = (x, y) \in \mathbb{R}^n \times \mathbb{R}$ to separate the classes $\mathcal{C}_1$ and $\mathcal{C}_2$ and to represent the given data $(a_i, b_i) \in \mathbb{R}^n \times \{-1, 1\}$ optimally. A new data point a can then be classified via

$$\begin{cases} +1 & \text{if } \ell_{(x,y)}(a) > 0 \\ -1 & \text{if } \ell_{(x,y)}(a) \leq 0 \end{cases} \quad \text{or} \quad \begin{cases} \mathcal{C}_1 & \text{if } \ell_{(x,y)}(a) > 0 \\ \mathcal{C}_2 & \text{if } \ell_{(x,y)}(a) \leq 0 \end{cases}$$

In order to improve the separation of the classes $\mathcal{C}_1$ and $\mathcal{C}_2$ and to reduce potential misclassifications, one typically considers the following hinge-loss formulation

$$\min_{x,y} \frac{\lambda}{2}\|x\|^2 + \sum_{i=1}^{m} \max\left\{0, 1 - b_i\left(a_i^\top x + y\right)\right\} \tag{1}$$

to determine the parameters x and y. Here, $\lambda > 0$ is a regularization parameter that balances the margin and the misclassification error.

1.2 Objectives

Our goal is to utilize minimization methodologies(such as Gradient Descent, Accelerated Gradient Descent, Globalized BFGS, Globalized L-BFGS, Stochastic Gradient Descent and so on) to solve support vector machine and logistic regression models for our own synthetic datasets and other large-scale binary classification tasks from LIBSVM.

2 Data Preparation

In this part, we discuss two categories of data: self-generated data point clouds in the two-dimensional plane and datasets coming from LIBSVM.

2.1 Generate Synthetic Data

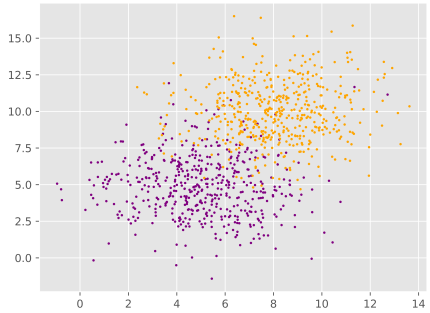
The way of generating data point clouds can be shown as following: set the center c_1, c_2 for each of the two class $\mathcal{C}_1$ and $\mathcal{C}_1$.We will generate m_1, m_2 numbers of points for class $\mathcal{C}_1$ and class $\mathcal{C}_2$.Then, we randomly

choose ε_1, ε_2, δ_1, δ_2 where $\varepsilon_1, \varepsilon_2 \sim N\left(0, \sigma_1^2\right)$, $\delta_1, \delta_2 \sim N\left(0, \sigma_2^2\right)$, and set

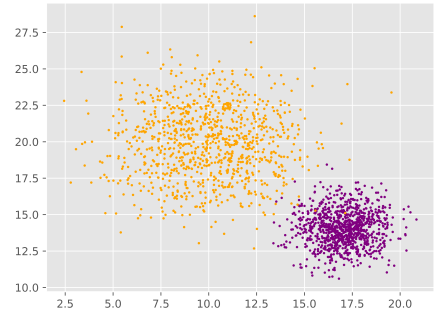
$$a_i = c_1 + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \end{pmatrix}, \quad a_j = c_2 + \begin{pmatrix} \delta_1 \\ \delta_2 \end{pmatrix}$$

where $i = 1, 2, 3, ..., m_1, j = 1, 2, 3, ..., m_2$.

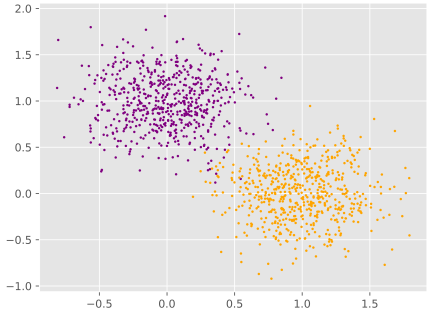
As in Figure 1, we generate four datasets 1a-1d by choosing different parameters.



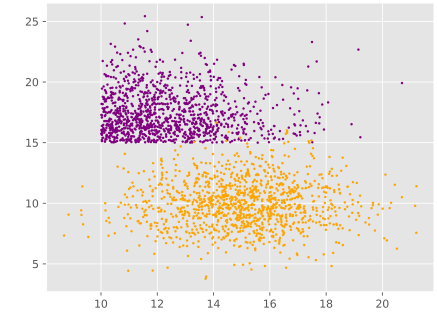
(a) Dataset1 with $c_1 = (5, 5)$, $c_2 = (8, 10)$, $\sigma_1 = 2$ $\sigma_2 = 2$ **and** $m_1 = m_2 = 500$



(b) Dataset2 with $c_1 = (17, 14)$, $c_2 = (10, 20)$, $\sigma_1 = 1.2$ $\sigma_2 = 2.5$ **and** $m_1 = m_2 = 1000$



(c) Dataset3 with $c_1 = (0, 1)$, $c_2 = (1, 0)$, $\sigma_1 = 0.3$ $\sigma_2 = 0.3$ **and** $m_1 = m_2 = 600$



(d) Dataset4 with $c_1 = (10, 15)$, $c_2 = (15, 10)$, $\sigma_1 = 3$ $\sigma_2 = 2$ **and** $m_1 = m_2 = 1200$

Figure 1 Self-Generated Datasets

2.2 Data From LIBSVM

Table 1 are the description of the datasets from LIBSVM we used in later numerical comparison.

Table 1 A description of the datasets used in the numerical comparison

data set	m	n	data set	m	n
a9a	32561	122	mushrooms	8124	112
breast-cancer	683	10	news20	19996	1355191
covtype	581012	54	phishing	11055	68
gisette	6000	5000	rcv1	20242	47236

2.3 Data Processing

For self-synthesized data, it is already clean, and in normalized form, so no further processing is needed. For data from LIBSVM, the data is in '.mat' format and is sparse. Thus, we basically use 'scipy.io.loadmat' to load the sparse matrix. Then, because some of the data set only provides two datasets respectively contain features and labels, splitting data into train set and test set is needed. Here we split data into two groups $A_{\text{train}} = (a_i)_{i\in\mathcal{T}}$ and $A_{\text{test}} = (a_i)_{i\in\mathcal{T}^C}$ where $\mathcal{T}^C = \{1,\dots,m\}\backslash\mathcal{T}$, and set $|\mathcal{T}| = 70\% \cdot m$, where m is the total number of records. Then because the features in the data sets are of different scale, normalization is needed. We simply use sklearn.preprocessing.MaxAbsScaler to normalize the data: Maxabsscaler works in a way that the training data lies within the range [-1,1] by dividing through the largest maximum value in each feature. It is meant for data that is already centered at zero or sparse data.

3 Support Vector Machines

We first consider a smooth variant of the support vector machine problem (1) that was already introduced in the lectures:

$$\min_{x,y} f_{\text{svm}}(x,y) := \frac{\lambda}{2}\|x\|^2 + \sum_{i=1}^{m} \varphi_+\left(1 - b_i\left(a_i^\top x + y\right)\right) \tag{2}$$

Here, $\varphi_+(t)$ denotes a Huber-type version of the max-function $\max\{0,t\}$:

$$\varphi_+(t) = \begin{cases} \frac{1}{2\delta}(\max\{0,t\})^2 & \text{if } t \le \delta \\ t - \frac{\delta}{2} & \text{if } t > \delta \end{cases}$$

For the convenience of further implementation, we define $\varphi_+(t)$, f_{svm}, ∇f_{svm} in our python scripts as follows:

```python
def phi_plus(t):
    if t <= delta:
        return 1/(2*delta) * (max(0, t)**2)
    else:
        return t - delta/2

def f_svm(xk, m, Lambda):
    x = xk[:-1]
    y = xk[-1][0]
    return Lambda/2 * np.linalg.norm(x)**2 + sum(map(phi_plus, 1 - b * (np.dot(a,x) + y).reshape(-1)))
def df(xk, m, Lambda):
    x = xk[:-1]
    y = xk[-1][0]
    grad = np.zeros((x.size+1, 1))
    t = 1 - b * (np.dot(a,x) + y).reshape(-1)
    dphi_list = np.zeros(m)
    dphi_list[t>delta] = 1
    dphi_list[(0<t) & (t<delta)] = t[(0<t) & (t<delta)]/delta
    grad[:-1] = Lambda*x + np.dot(dphi_list * (-b), a).reshape(-1,1)
    grad[x.size] = np.sum(-dphi_list * b)
    return grad
```

3.1 Implement GM and AGM

We implement the basic gradient method(GM) with backtracking ($\gamma = 0.1, \sigma = 0.5$, and $s = 1$) and the accelerated gradient method (AGM) for the smoothed support vector machine problem (2).

Also, for the AGM we follow the basic extrapolation strategy

$$\alpha_k = \frac{1}{L}, \quad \beta_k = \frac{t_{k-1}-1}{t_k}, \quad t_k = \frac{1}{2}\left(1 + \sqrt{1 + 4t_{k-1}^2}\right), \quad t_{-1} = t_0 = 1$$

and estimates the Lipschitz constant shown in Algorithm 1.

Algorithm 1: The Accelerated Gradient Method for Unknown Lipschitz Constants

Initialization: Choose $x^0 \in \mathbb{R}^n$, set $x^{-1} = x^0$ and $t_{-1} = t_0 = 1$. Choose $\alpha_{-1} > 0$ and $\eta \in (0, 1)$

for $k = 0, 1, 2, \ldots$ **do**

Compute the extrapolation parameter $\beta_k = t_k^{-1}(t_{k-1} - 1)$ and set $y^{k+1} = x^k + \beta_k\left(x^k - x^{k-1}\right)$

Set $\alpha_k = \alpha_{k-1}$ and $\bar{x}^{k+1} = y^{k+1} - \alpha_k \nabla f\left(y^{k+1}\right)$

while $f\left(\bar{x}^{k+1}\right) - f\left(y^{k+1}\right) > -\frac{\alpha_k}{2}\|\nabla f\left(y^{k+1}\right)\|^2$ **do**

Set $\alpha_k = \eta\alpha_k$ and recompute $\bar{x}^{k+1} = y^{k+1} - \alpha_k \nabla f\left(y^{k+1}\right)$

Set $t_{k+1} = \frac{1}{2} \cdot \left(1 + \sqrt{1 + 4t_k^2}\right)$ and $x^{k+1} = \bar{x}^{k+1}$

3.2 Implement the globalized BFGS

We implement the globalized BFGS method for smoothed SVM problem (2) with armijo line search and we choose $H_0 = I$ as initial matrix for the BFGS-updates. In order to guarantee positive definiteness of the BFGS updates, the pair $\left\{s^k, y^k\right\}$ should only be added to the current curvature pairs if the condition $\left(s^k\right)^\top y^k > 10^{-14}$ is satisfied. (Notice that here $y^k = \nabla f\left(x^{k+1}\right) - \nabla f\left(x^k\right)$ denotes the vector within the BFGS-framework and not the bias term in the model $\ell_{(x^k, y^k)}$)).

3.3 Test on Synthetic Datasets and Performance

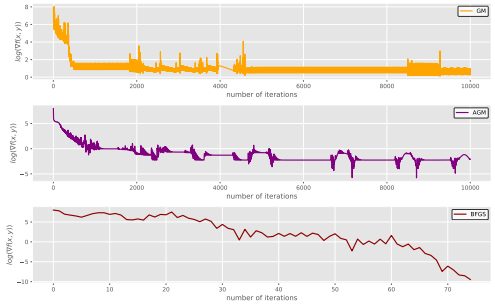
We choose $\delta = 10^{-3}$, $tol = 10^{-4}$ and $\lambda = 0.1$, run the three different methods on our four synthetic datasets. For each dataset, we plot and compare the convergence w.r.t. the norm of the gradient $\nabla f_{\text{svm}}\left(x^k, y^k\right)$ as in Figure 2.

We also compare their performance with respect to the number of iterations, the required cpu-time and computed their corresponding classification accuracy as follows:

$$\tau\left(x^*, y^*\right) := 100\% \cdot \frac{\sum_{i \in \mathcal{T}^*} |p_i + b_i|}{2\left|\mathcal{T}^{\mathrm{C}}\right|} \text{ where } p_i = \begin{cases} +1 & \text{if } \ell_{(x^*, y^*)}(a_i) > 0 \\ -1 & \text{if } \ell_{(x^*, y^*)}(a_i) \leq 0 \end{cases} \quad \forall i \in \mathcal{T}^{\mathrm{C}}$$

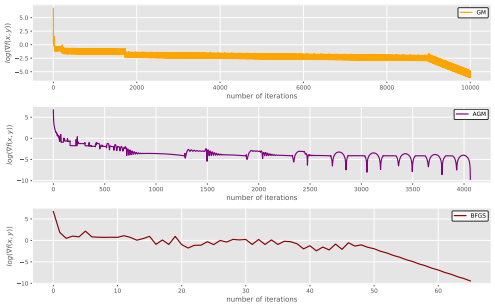
where (x^*, y^*) is a solution returned by the algorithms. And we split the dataset into two disjoint (and randomly selected) groups $A_{\text{train}} = (a_i)_{i \in \mathcal{T}}$ and $A_{\text{test}} = (a_i)_{i \in \mathcal{T}}$ where $\mathcal{T}^{\mathrm{C}} = \{1, \ldots, m\} \backslash \mathcal{T}$ denotes the complement of the index set $\mathcal{T} \subset \{1, \ldots, m\}$. In the optimization problem (2), we then only access data from the set A_{train}. We then validate the obtained result on the test data and define the so-called testing accuracy above. The higher the testing accuracy the more data points have been classified correctly by our model $\ell_{(x^*, y^*)}$.

And Table 2 is the summary results of those performance.

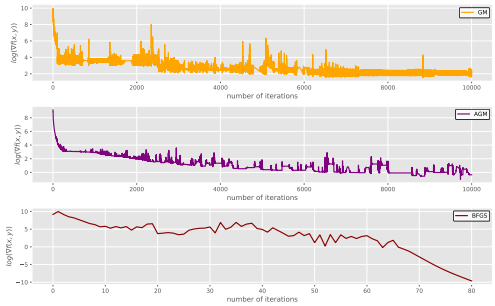


(a) Performance for Dataset1

(b) Performance for Dataset2



(c) Performance for Dataset3



(d) Performance for Dataset4

Figure 2 Performance for Synthetic Datasets on Different Methods-SVM

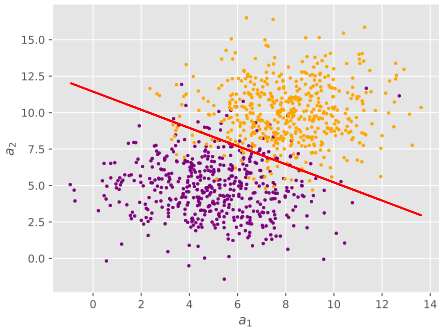
Table 2 The performance of synthetic datasets for SVM

datasets	methods	accuracy	iter	cpu-time	datasets	methods	accuracy	iter	cpu-time
dataset 1	GM	0.922	10000*[1]	329.54s	dataset 3	GM	0.992	10000*	271.28s
	AGM	0.923	10000*	20.11s		AGM	0.992	4061	10.23s
	BFGS	0.923	74	3.08s		BFGS	0.992	65	2.22s
dataset 2	GM	0.996	10000*	749.91s	dataset 4	GM	0.99	10000*	778.71s
	AGM	0.997	10000*	39.09s		AGM	0.995	10000*	46.21s
	BFGS	0.997	95	9.15s		BFGS	0.995	80	6.55s

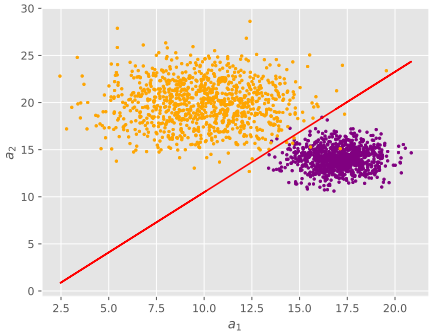
As we can see from Figure 2 and Table 2, the GM method didn't converge in any dataset, but the accuracy of GM is high in all datasets after 10000 iteration times. For AGM, It is converged on dataset 3 but no other three datasets and the accuracy of AGM is also high. Besides, the BFGS is the best method to solve this problem on the four datasets. It converged in all of the datasets quickly in serveral seconds, which is much faster than GM and AGM. Therefore, BFGS is the best method in terms of speed and convergence, as well as accuracy.

We also visualize the classification. Because for each data set, the visualized figures are relatively similar if using different methods (GM, SVM, BFGS), here we just show the figure of BFGS in Figure 3.

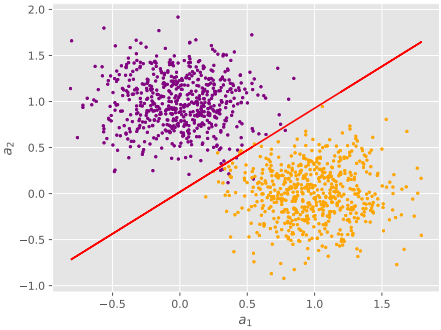
[1] * means the max iteration time reached



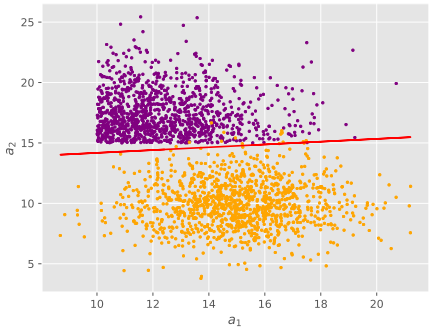
(a) Visualization for Dataset1 with BFGS



(b) Visualization for Dataset2 with BFGS



(c) Visualization for Dataset3 with BFGS



(d) Visualization for Dataset4 with BFGS

Figure 3 Visualization for Synthetic Datasets on BFGS-SVM

4 Logistic Regression

We now consider a second model for binary classification - the so-called logistic regression model. Here, the corresponding optimization problem is given by:

$$\min_{x,y} f_{\log}(x,y) = \frac{1}{m}\sum_{i=1}^{m} \log\left(1+\exp\left(-b_i\cdot\left(a_i^\top x+y\right)\right)\right) + \frac{\lambda}{2}\|x\|^2 \tag{3}$$

In the case of logistic regression, the sigmoid function $\sigma:\mathbb{R}\to\mathbb{R}, \sigma(a)=\frac{1}{1+\exp(-a)}$ is chosen as underlying probability model. In logistic regression, we want to train the linear model $\ell_{(x,y)}(a)=a^\top x+y$ such that

$$\sigma\left(\ell_{(x,y)}\left(a_i\right)\right)=\sigma\left(a_i^\top x+y\right)\approx\begin{cases}1 & \text{if } a_i \text{ belongs to class } \mathcal{C}_1, \text{ i.e., } b_i=+1\\ 0 & \text{if } a_i \text{ belongs to class } \mathcal{C}_2, \text{ i.e., } b_i=-1\end{cases}$$

A new data point $a\in\mathbb{R}^n$ can then be classified via

$$\begin{cases}+1 & \text{if } \sigma\left(\ell_{(x,y)}(a)\right)>\frac{1}{2}\\ -1 & \text{if } \sigma\left(\ell_{(x,y)}(a)\right)\le\frac{1}{2}\end{cases} \quad \text{or} \quad \begin{cases}\mathcal{C}_1 & \text{if } \sigma\left(\ell_{(x,y)}(a)\right)>\frac{1}{2}\\ \mathcal{C}_2 & \text{if } \sigma\left(\ell_{(x,y)}(a)\right)\le\frac{1}{2}\end{cases}$$

The optimization problem (3) is based on a corresponding maximum likelihood approach to estimate the optimal model parameters x and y for this probabilistic strategy.

Also, for the convenience of further implementation, we define $f_{\log}(x,y)$ and $\nabla f_{\log}(x,y)$ in our python scripts as follows:

```python
def f_logit(xk, Lambda):
    x, y = xk[:-1], xk[-1]
    Log = np.log(1+np.exp(-b*(a.dot(x)+y)))
    return Lambda/2 * np.linalg.norm(x)**2 + Log.sum()/m

def df_logit(xk, Lambda):
    x = xk[:-1]
    y = xk[-1]

    grad = np.zeros((x.size+1, 1)).reshape(-1)

    t = np.exp(-b.reshape(-1)*(a.dot(x)+y).reshape(-1))

    df_Log = np.dot((t /(1+t))*(-b), a)
    grad[:-1] = Lambda * x + df_Log/m
    df_Log = (t /(1+t))*(-b)
    grad[x.size] = df_Log.sum()/m
    return grad.reshape(-1,)
```

4.1 Implement GM and AGM

Here we set $\lambda = 0.1$, We firstly use basic GM with backtracking ($\gamma = 0.1, \sigma = 0.5, s = 1$) on the logistics regression. Then we apply the AGM on the logistic regression as is applied before on the SVM but here we take $L = \frac{1}{4m}\sum_{i=1}^{m}\|a_i\|^2$ as the Lipschitz constant of $\nabla f_{\log}$.

4.2 Implement the globalized L-BFGS

We implement the L-BFGS with armijo line search. Note that here, $s^{k-1} = x^k - x^{k-1}, \quad y^{k-1} = \nabla f\left(x^k\right) - \nabla f\left(x^{k-1}\right)$, $\rho_k = \left(\left(s^k\right)^\top y^k\right)^{-1}$, $\gamma^k = \frac{\left(s^{k-1}\right)^\top y^{k-1}}{\|y^{k-1}\|^2}$, $H_k^0 = \gamma^k I$.Here we set the memory parameter $m_{L-BFGS} = 5$. And in the two loop recursion, if $k < m_{L-BFGS}$, we choose to iterate k times instead of m_{L-BFGS} times.

4.3 Test on Synthetic Datasets and Performance

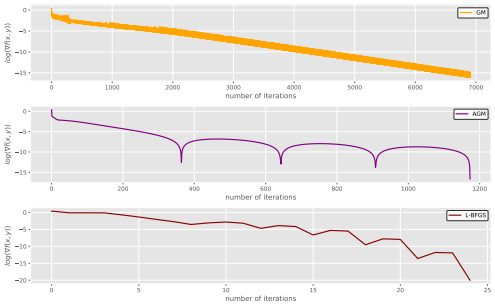
We choose $tol = 10^{-7}$ and $\lambda = 0.1$, then ran the methods on each synthesized dataset. Then we plot the convergence w.r.t. the norm of the gradient $\nabla f_{\text{svm}}\left(x^k, y^k\right)$ as in Figure 4.

As for the required cpu-time, numbers of iteration and accuracy of the classification, they are shown in Table 3. As we can see from Figure 4a-4d, and Table 3, all those three methods converged in our four synthetic datasets. Though they have different iteration times over different datasets. Overall, L-BFGS performed much better than both other two methods. Also, one interesting phenomenon is that the trend of AGM, and the huge iteration times for GM to converge.

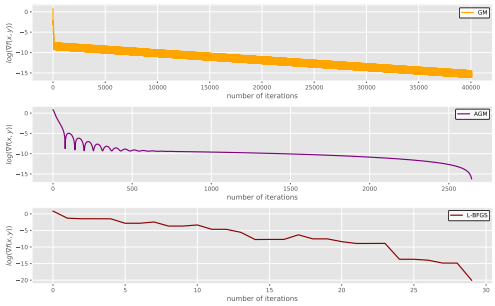
Table 3 The performance of synthetic datasets for Logistic

datasets	methods	accuracy	iter	cpu-time	datasets	methods	accuracy	iter	cpu-time
dataset 1	GM	0.922	6903	4.31s	dataset 3	GM	0.991	124	0.64s
	AGM	0.922	1173	0.63s		AGM	0.991	21	0.69S
	L-BFGS	0.922	24	0.52s		L-BFGS	0.991	13	0.52s
dataset 2	GM	0.994	40068	27.01s	dataset 4	GM	0.989	79722	62.91s
	AGM	0.994	2644	1.16s		AGM	0.989	3073	1.44s
	L-BFGS	0.944	29	0.86s		L-BFGS	0.989	26	0.72S

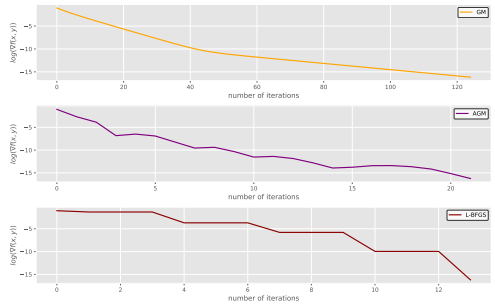
As same as before, we visualize the result of classification of L-BFGS of logistic regression in Figure 5.



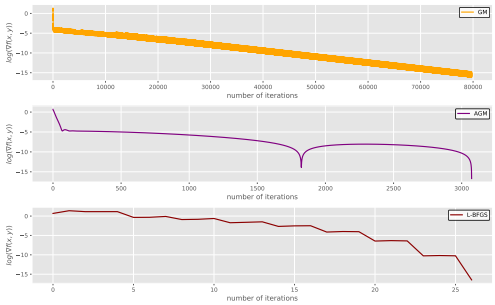
(a) Performance for Dataset1



(b) Performance for Dataset2

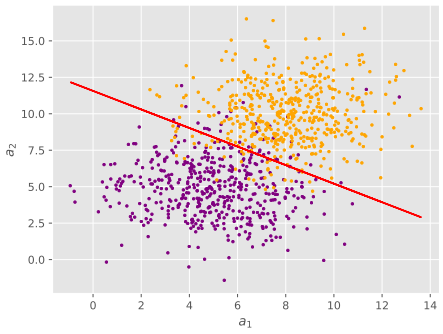


(c) Performance for Dataset3

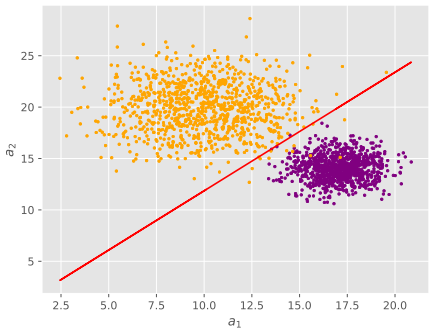


(d) Performance for Dataset4

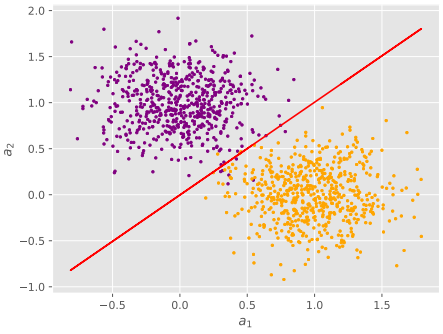
Figure 4 Performance for Synthetic Datasets on Different Methods-Logistic



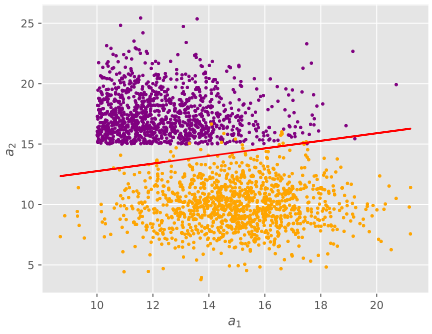
(a) Visualization for Dataset1 with L-BFGS



(b) Visualization for Dataset2 with L-BFGS



(c) Visualization for Dataset3 with L-BFGS



(d) Visualization for Dataset4 with L-BFGS

Figure 5 Visualization for Synthetic Datasets on lBFGS-Logistic

Algorithm 2: The L-BFGS Method

Initialization: set $\mathbf{x}^0 = 0$, $\mathbf{x}^0 \in \mathbb{R}^{n+1}$, $d^0 = -\nabla f(\mathbf{x}^0)$. And use backtracking ($\gamma = 0.1, \sigma = 0.5, s = 1$) to decide α_0.

for $k = 1, 2, ..., max_{iter}$ **do**

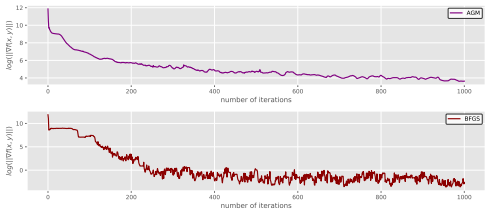
- set $\mathbf{x}^k = \mathbf{x}^{k-1} + \alpha_{k-1} d^{k-1}, s^{k-1} = \mathbf{x}^k - \mathbf{x}^{k-1}, y^{k-1} = \nabla f(\mathbf{x}^k) - \nabla f(\mathbf{x}^{k-1})$
- Break when $\|\nabla f(\mathbf{x}^k)\| <= tol$
- **if** $(s^k)^\top y^k > 10^{-14}$ **then**
  - $\gamma^k = \frac{\left(s^{k-1}\right)^\top y^{k-1}}{\|y^{k-1}\|^2}$;
- **else**
  - $\gamma^k = 1$;
- set $q = \nabla f\left(\mathbf{x}^k\right)$
- **for** $i = k-1, k-2, \ldots, k - m_{L-BFGS}$ **do**
  - set $\alpha_i = \rho_i \cdot \left(s^i\right)^\top q$ and
  - **if** $(s^i)^\top y^i > 10^{-14}$ **then**
    - $q = q - \alpha_i y^i$;
  - **else**
    - $q = q$;
- Set $r = H_k^0 q$
- **for** $i = k - m_{L-BFGS}, k - m_{L-BFGS} + 1, \ldots, k-1$ **do**
  - **if** $(s^i)^\top y^i > 10^{-14}$ **then**
    - $\beta = \rho_i \cdot \left(y^i\right)^\top r$ and $r = r + (\alpha_i - \beta) s^i$;
  - **else**
    - $r = r$;
- set $r = H_k \nabla f\left(\mathbf{x}^k\right), d^k = -r$, use backtracking ($\gamma = 0.1, \sigma = 0.5, s = 1$) to decide α_k.

5 Performance on LIBSVM Datasets

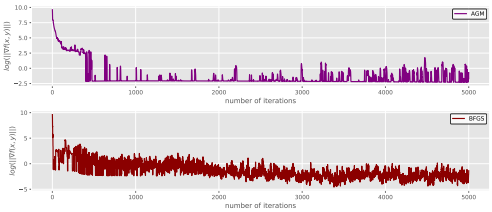
5.1 For SVM

We choose the same parameters as before and test different methods on the a9a, mushroom and breast-cancer datasets for smoothed SVM problems. The corresponding performances are as in Figure 6 and Table 4. As is showen in Figure 6,for the dataset a9a, we have not observe convergence when using AGM and BFGS, and we found that BFGS performs better than AGM with same numbers of iteration and descend faster. Using the data set mushroom, we also have not met convergence with 5000 times of iteration. What is worth mentioning is that we observe a sharp descend at first several iteration while AGM descended slower. As for dataset breast-cancer, the BFGS converges within 100 of iterations and the AGM converges with relative more iterations. As for basic gradient method, we have not observe convergence with maximum iteration times of 10000. As is mentioned before, here we did not use basic gradient method when handling a9a and mushroom due to limitation of computing power. As for Table 4, because a9a and mushroom do not converge, we focus on breast-cancer and found that BFGS spent less time. To make a brief summary, here we mainly observe that BFGS converges faster and works better than AGM from the perspective of iteration numbers and cpu time.

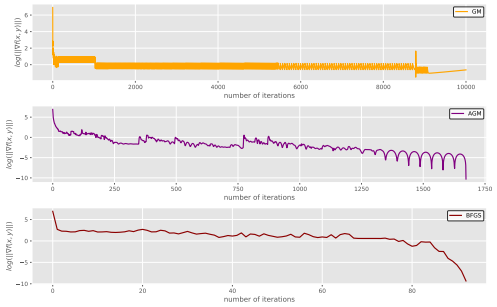
[2]Due to limitation of time and computing resources, we suspend these parts.



(a) Performance for a9a Dataset



(b) Performance for mushroom Dataset



(c) Performance for breast-cancer Dataset

Figure 6 Performance for LIBSVM Datasets on Different Methods-SVM

Table 4 The performance of LIBSVM datasets for SVM

datasets	methods	accuracy	iter	cpu-time	datasets	methods	accuracy	iter	cpu-time
breast cancer	GM	0.956	10000	154.97s	a9a	GM	#	#	#
	AGM	0.956	1673	2.18s		AGM	0.850	1000*	51.91s
	BFGS	0.956	92	0.59 S		BFGS	0.850	1000*	176.33s
mashroom	GM	#[2]	#	#	#	#	#	#	#
	AGM	1.000	5000	53.31s					
	BFGS	1.000	5000	124.17s					

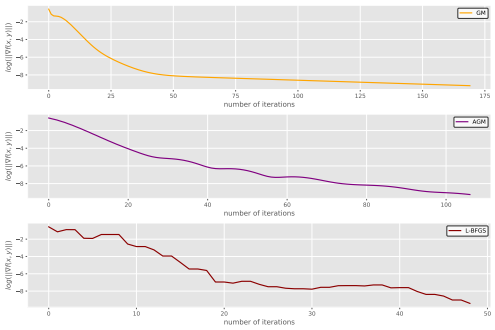
5.2 For Logistic Regression

For logistic regression, we test breast-cancer, mushroom, a9a, covtype, Phishing, news20, Rcv1 and gisette. The performances can be seen in Figure 7 and Table 5. As is shown in 7, generally, L-BFGS requires less numbers of iteration. As for the other two method, one might work better than the other. We think it might depends on the feature of data. As for the cpu time in 5, cpu time of each method have no relative obvious mode in our experiment. We believe there should be relations between cpu time and method, maybe not in a relative intuitive relation and further study is needed. Besides, all of the methods reaches same accuracy for the same datasets if converging.

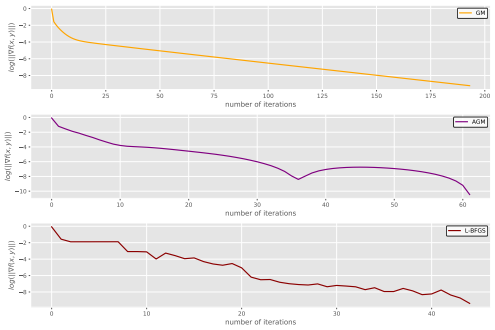
6 Extensions

6.1 Further Improve Performance

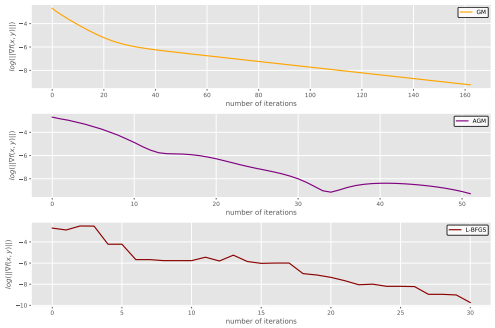
There are many strategies to further improve the performance of our implementation. Such as choosing different linesearch strategies, parameters (within the algorithm), or update rules. Here, we just take Logistic Regression model with L-BFGS method as the example to adjust the parameter λ in order to obtain a better performance. As we can see from Figure 8, the accuracy decreases with the increasing of λ, and the accuracy goes to a steady state in large λ. For dataset phishing and breast-cancer, the accuracy will close to 0.5 when the λ is relatively large. This is because λ is the Regularization parameter, which means the higher λ, the smaller $\|x\|$ would be. In this case, x will be close to 0, which means the model is not good. Thus, the



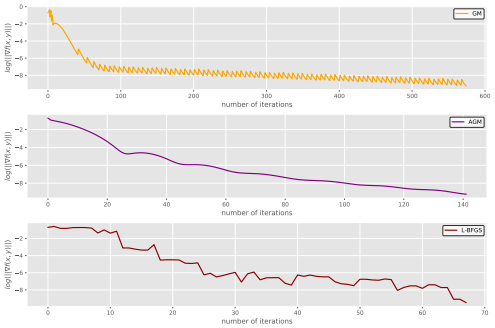
(a) Performance for mashroom Dataset



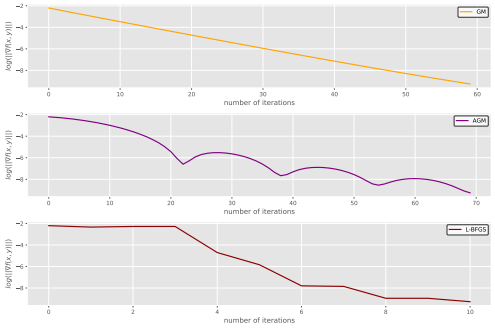
(b) Performance for breast cancer Dataset



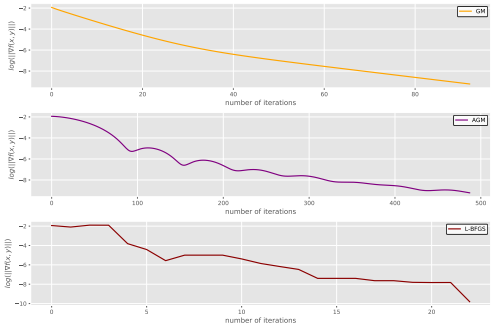
(c) Performance for covtype Dataset



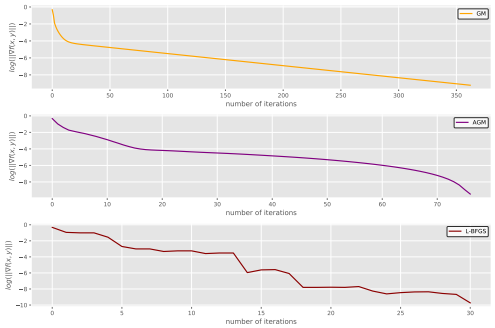
(d) Performance for phishing Dataset



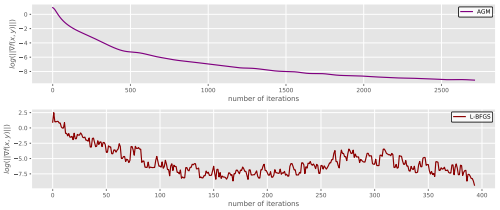
(e) Performance for rcv1 Dataset



(f) Performance for news20 Dataset



(g) Performance for a9a Dataset



(h) Performance for gisette Dataset

Figure 7 Performance for LIBSVM Datasets on Different Methods-Logistic

accuracy will be close to 0.5, seems to guess randomly.

Table 5 The performance of LIBSVM datasets for Logistic

datasets	methods	accuracy	iter	cpu-time	datasets	methods	accuracy	iter	cpu-time
breast-cancer	GM	0.93	193	0.47s	mushroom	GM	0.96	169	0.60s
	AGM	0.93	61	0.27s		AGM	0.96	106	0.37s
	L-BFGS	0.93	44	0.41s		L-BFGS	0.96	48	0.93s
a9a	GM	0.80	362	4.34s	covtype	GM	0.63	162	16.79s
	AGM	0.80	76	0.76s		AGM	0.63	51	3.54s
	L-BFGS	0.80	30	1.36s		L-BFGS	0.63	30	12.52s
Phishing	GM	0.92	574	3.08s	news20	GM	0.89	92	38.54s
	AGM	0.92	141	0.55s		AGM	0.89	487	190.98s
	L-BFGS	0.92	67	1.06s		L-BFGS	0.89	22	107.54s
Rcv1	GM	0.84	59	8.18s	gisette	GM	#	#	#
	AGM	0.84	69	6.64s		AGM	0.97	2712	953.09s
	L-BFGS	0.84	10	5.53s		L-BFGS	0.97	393	605.16s

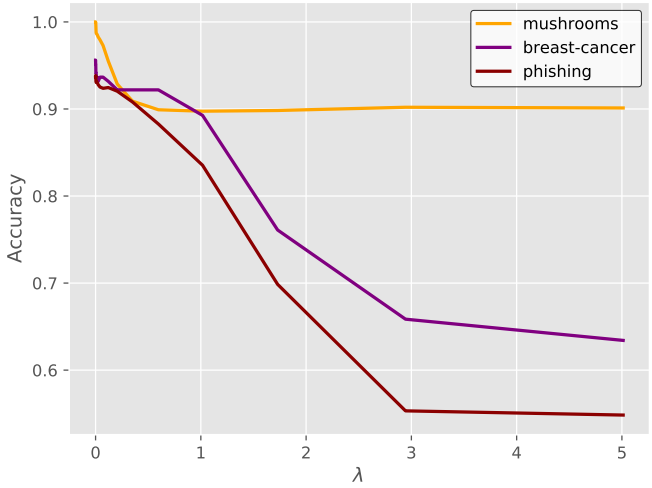


Figure 8 Adjustment of λ of L-BFGS method in Logistic Regression

6.2 Stochastic Optimization

6.2.1 Background Information

Both the support vector machine and the logistic regression problem can be expressed as a so-called empirical risk minimization problem of the form

$$\min_{x,y} f(x,y) = \frac{1}{m}\sum_{i=1}^{m} f_i(x,y)$$

where each $f_i : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ corresponds to our loss model being evaluated at a certain data point $(a_i, b_i) \in \mathbb{R}^n \times \{-1, +1\}$. If the number of data points m is large or if the dimension $n+1$ of the problem is large, the evaluation of the full gradient $\nabla f(x,y)$ can be very time-consuming and - in some cases - is not even possible. Stochastic gradient methods are based on the idea of using simpler stochastic approximations of the gradient in each iteration. In particular, in each iteration we sample one index i_k from $\{1, \ldots, m\}$ uniformly

at random and perform the update

$$\begin{pmatrix} x^{k+1} \\ y^{k+1} \end{pmatrix} = \begin{pmatrix} x^{k} \\ y^{k} \end{pmatrix} - \alpha_k \nabla f_{i_k}\left(x^k, y^k\right)$$

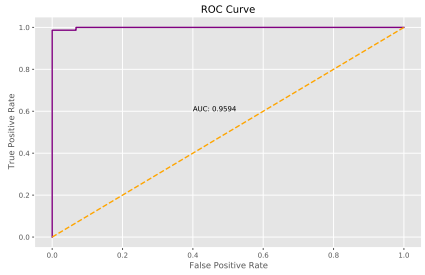
Convergence of this simple approach can be guaranteed if the step size $\alpha_k = \eta\beta_k$ are diminishing and satisfy $\sum_{k=0}^{\infty} \beta_k = \infty, \sum_{k=0}^{\infty} \beta_k^2$, and $\eta > 0$. There are also mini-batch version, where we select a (larger) subset $\mathcal{S}_k \subset \{1, \ldots, m\}$ at random and set

$$\begin{pmatrix} x^{k+1} \\ y^{k+1} \end{pmatrix} = \begin{pmatrix} x^{k} \\ y^{k} \end{pmatrix} - \frac{\alpha_k}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}_k} \nabla f_i\left(x^k, y^k\right)$$

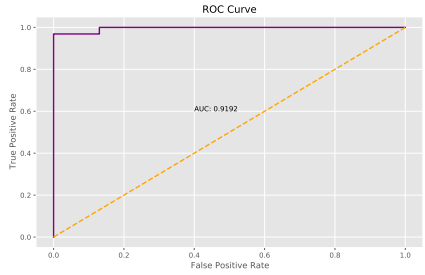
6.2.2 Implementation and Performance of SGD

For the implementation of SGD, we choose the diminishing step as $\alpha_k = \eta\beta_k$, where $\eta = 10$ and $\beta_k = \frac{1}{log(k+2)}$, and the batch size as 66. Figure 9 are the performance for mushroom and phishing datasets by using SGD methods for solving Logistic problem.

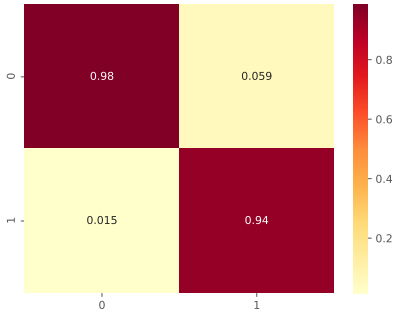
As we see, Figure 9a and Figure 9b are the ROC Curve for mushroom and phishing respectively. The AUC values of are 0.9594 and 0.9192 accordingly, they are both quite close to 1, which shows a good performance of classification. We can also see this from Figure 9c and Figure 9d, they are the corresponding confusion matrix for this classification problem. As we see, the true positive rate(top left) and true negative rate(bottom right) for each dataset are both close to 1, and at the same time the false positive rate(top right, also is type I error) and the false negative rate(bottom left, also is type II error) are quite low(close to 0), which shows our classification results on these two datasets are quite well.



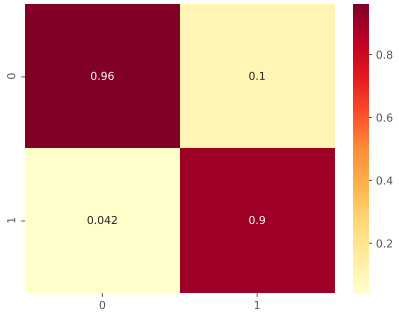
(a) ROC Curve for mushroom Dataset



(b) ROC Curve for phishing Dataset



(c) Confusion Matrix for mushroom Dataset



(d) Confusion Matrix for phishing Dataset

Figure 9 Performance for LIBSVM Datasets on SGD Methods-Logistic

7 Conclusion

In this project, we have implemented two models including Support Vector Machine (SVM) and Logistic Regression as a classification model to solve the binary classification problem. Firstly, we generated four synthetic datasets. And then we tested our models with different optimization methods on the synthetic datasets. The synthetic datasets helped us to debug and improve the quality of our python scripts. Then, we use the models and optimization method on LIBSVM Datasets, which is bigger and more sophisticated. For more work, we tune the hyperparameter λ of logistic regression problem and also implement the mini batch stochastic gradient method. Later we test its performance on two LIBSVM Datasets and plot the corresponding ROC curve and confusion matrix.

Due to the time limitation, we have a lot of works unfinished. Thus, we put them as the future work. For example, we just adjusted the parameter Λ in Further Improve Performance part, but there are many other strategies can be applied to improve the performance of the models and algorithms. Furthermore, the interpretation of the result and the performance should goes deeper, which means the more principles should be investigated. Fortunately, all of our works are done via github, so all the files are administrated well. As a result, we will keep updating and optimizing this project in the future via this repository (https://github.com/Yihang-Li/MDS6106Project).

Appendices

All of our supporting materials can be found in our github repository.