1. 数学优化算法实现

1.1 线搜索

线搜索是一种迭代的求得某个函数的最值得方法，对于每次迭代，线搜索会计算得到搜索的方向以及沿这个方向移动的步长。下降方向可以通过多种方法计算，比如说：梯度下降法，牛顿法和拟牛顿法，计算的步长不一定是精确的。

1.1.1 二分法

二分法是最简单的线搜索方法，其主要的思想是在给定范围内取中点，在给定的足够小的值ε左右偏移并计算，然后更新搜索范围。

$$
\begin{aligned}
\lambda_k &= \frac{a_k + b_k}{2} - \varepsilon \\
\mu_k &= \frac{a_k + b_k}{2} + \varepsilon \\
if \;\; f(\lambda) > f(\mu_k) \;\;\; a_{k+1} &= \lambda_k \;\;\; b_{k+1} = b_k \\
else\; if \;\; f(\lambda) < f(\mu_k) \;\;\; a_{k+1} &= \lambda_k \;\;\; b_{k+1} = b_k
\end{aligned} \tag{1}
$$

实现二分法的代码如下：

```
def bisection_search(f, def_field, epsilon):
    half_epsilon = epsilon / 2.0
    lf = list(def_field)
    f_vals = [f(lf[0]), f(lf[1])]

    while True:
        mid_x = (lf[1] + lf[0]) / 2.0
        mid_esp = [mid_x - half_epsilon, mid_x + half_epsilon]
        tmp_f = [f(mid_esp[0]), f(mid_esp[1])]
        if tmp_f[0] > tmp_f[1]:
            f_vals[0] = tmp_f[0]
            lf[0] = mid_esp[0]
        else:
            f_vals[1] = tmp_f[1]
            lf[1] = mid_esp[1]

        if (lf[1] - lf[0]) <= epsilon * 1.000001:
            x_star = (lf[0] + lf[1]) / 2
            return x_star, f(x_star)
```

1.1.2 四等分搜索

四等分搜索的思想和二分思想相近，取中间三个点的最小值，该点两边两个点为新的区间，最小值落在该区间中。如果有两个相同的最小值，则最小值就在这两个最小值点之间。

实现代码如下

```
def equal_interval_search(f, def_field, epsilon):
    steps = np.linspace(def_field[0], def_field[1], 5)
    values = np.apply_along_axis(f, 0, steps)

    while True:
        min_id = values[1:4].argmin() + 1
        steps = np.linspace(steps[min_id - 1], steps[min_id + 1], 5)
        values = np.array([values[min_id - 1], f(steps[1]), values[min_id],
f(steps[3]), values[min_id + 1]])

        if (steps[4] - steps[0]) < epsilon * 0.1:
            return steps[2], values[2]

    return None
```

1.1.3 fibonacci搜索

Fibonacci搜索的主要算法流程如下：

斐波那契搜索的时间复杂度： $O(n) = log_2 n$
设查找的数组array的大小为m， 查询的值为value

```
    1. 先构建一个”数组元素符合斐波那契数列要求的且第一个大于等于要搜索的数组的大小“的数组F[n]
即F[n]-1>=m， F[n]为斐波那契数列
    2. 将要查找的数组大小为m的数组array扩展成数组大小为F[n]-1的数组temp。
    3. 将array中的值复制给temp，当temp的数组大小大于array时，temp中超出的元素全部复制为
array[m-1]
    4. 设k=n, low = 0, high = n-1, middle = low + F[n-1]-1
    5. 如果temp[middle] == value，则找到需要找的值
    6. 如果temp[middle] > value, 则high = middle-1, n = n-1, middle = low + F[n-
1]-1
    7. 如果 temp[middle] < value, 则low = middle +1，n = n-2, middle = low + F[n-
1]-1
    8. 重复5、6、7，直到找到或者low>high。
```

代码实现：

```
def fibonacci_search(f, def_field, epsilon):
    fib_values = np.array(fib_const_list)
    num = (def_field[1] - def_field[0]) / epsilon
    fidx_n = np.where(fib_values >= num)[0][0]

    step = (def_field[1] - def_field[0]) * fib_values[fidx_n - 1] * 1.0 /
fib_values[fidx_n]
    points = [def_field[0], def_field[1] - step, def_field[0] + step,
def_field[1]]
    f_values = np.apply_along_axis(f, 0, points)

    for i in range(fidx_n - 1, 1, -1):
        step = points[2] - points[1]
        if f_values[1] < f_values[2]:
            min_point = 1
            points[3] = points[2]
            f_values[3] = f_values[2]
            points[2] = points[1]
            f_values[2] = f_values[1]
```

```
            points[1] = points[0] + step
            f_values[1] = f(points[1])
        else:
            min_point = 2
            points[0] = points[1]
            f_values[0] = f_values[1]
            points[1] = points[2]
            f_values[1] = f_values[2]
            points[2] = points[3] - step
            f_values[2] = f(points[2])

    dst_p = (points[2] + points[1]) / 2

    return (dst_p, f(dst_p))
```

1.1.4 黄金分割法

三个原则（1）对称取点原则：选取x_1，x_2使得$x_1 - a = b - x_2$.（2）等比收缩原则：每次迭代时，让被删除的部分与原来区间的比值保持一个定值。（3）单点计算原则：每次迭代都只计算一次函数值。

1. step1：给定$a < b, \varepsilon > 0$.
2. step2：计算$x_1 := a + 0.382(b - a), x_2 := a + b - x_1$.
3. step3：计算$f_1 := f(x_1), f2 := f(x_2)$.
4. step4：如果$f_1 > f_2$,则令$a := x_1$,若$b - a < \varepsilon$,则转step5；否则令
   $f_1 := f_2, x_1 := x_2, x_2 := a + 0.618(b - a), f_2 := f(x_2)$,转step4.
   否则令$b := x2$,若$b - a < \varepsilon$,则转step5；否则令
   $f_2 := f_1, x_2 := x_1, x_1 := a + 0.382(b - a), f_1 := f(x_1)$,转step4.
5. step5:停止，输出$x^* = (a + b)/2$.

代码实现

```
def golden_section_search(f, def_field, epsilon, args=None):
    golden_const = 0.618

    step = (def_field[1] - def_field[0]) * 0.618
    points = [def_field[0], def_field[1] - step, def_field[0] + step,
def_field[1]]
    f_values = np.zeros(len(points))
    for i in range(len(points)):
        f_values[i] = f(points[i])

    while (points[2] - points[1]) > epsilon:
        # step = (points[2] - points[1])
        step = (points[2] - points[0]) * (1 - golden_const)
        if f_values[1] < f_values[2]:
            min_point = 1
            points[3] = points[2]
            f_values[3] = f_values[2]
            points[2] = points[1]
            f_values[2] = f_values[1]
            points[1] = points[0] + step
            f_values[1] = f(points[1])
        else:
            min_point = 2
            points[0] = points[1]
```

```
            f_values[0] = f_values[1]
            points[1] = points[2]
            f_values[1] = f_values[2]
            points[2] = points[3] - step
            f_values[2] = f(points[2])

    dst_p = (points[2] + points[1]) / 2

    return dst_p, f(dst_p)
```

1.1.5 非精确搜索

这里使用的是多项式拟合法，多项式$y = a_0 + a_x + a_2x^2 + \ldots + a_nx^n = \sum_{i=0}^{n} a_ix^i$

考虑输入样本数据如下：

$$\begin{aligned} x &= [x_1, x_2, \ldots, x_m]^T \\ y &= [y_1, y_2, \ldots, y_n]^T \end{aligned} \tag{2}$$

构建方程组：

$$\begin{aligned} y_1 &= a_0 + a_1x_1 + a_2x_1^2 + \ldots + a_nx_1^n \\ y_2 &= a_0 + a_1x_2 + a_2x_2^2 + \ldots + a_nx_2^n \\ &\cdots \\ y_m &= a_0 + a_1x_m + a_2x_m^2 + \ldots + a_nx_m^n \end{aligned} \tag{3}$$

令：

$$\begin{aligned} A &= \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{pmatrix} \\ b &= \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} \\ f &= \begin{pmatrix} y1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \end{aligned} \tag{4}$$

令$Ab = f$，拟合的过程就是求解b,按照数学表达式来看，$b = A^{-1}f$，A可逆的时候可以直接求解，当不可逆的时候可以考虑正交分解：$A_{[m,n+1]} = Q_{[m,n+1]}R_{[n+1,n+1]}$

代码实现如下：

```
def quadratic_polynomial(f, x0, v):
    a = 0.
    fa = f(x0)
    b = 1.
    fb = f(x0 + b * v)
    c = 0.
    fc = 0.
```

```
    st = 1.

    if fa > fb:
        # c: the first point f(c) > f(b)
        # lambda = 2, 4,8,...,a,b,c
        while True:
            c = b + st
            fc = f(x0 + c * v)
            if fc >= fb:
                break

            fa = fb
            fb = fc
            a = b
            b = c
            st *= 2.
    else:
        # a=0,b=lambda, c=2*lambda, lambda=1/2, 1/4, ...
        c = b
        fc = fb
        while True:
            st *= 1. / 2
            b = a * st

            fb = f(x0 + b * v)
            if fb < fc:
                break
            fc = fb
            c = b

    # pdb.set_trace()
    lambd = 1 / 2.0 * (fa * (c ** 2 - b ** 2) + fb * (a ** 2 - c ** 2) + fc * (b
** 2 - a ** 2))
    lambd /= (fa * (c - b) + fb * (a - c) + fc * (b - a))

    f_lamb = f(lambd)
    if f_lamb < fb:
        return lambd

    return b
```

1.1.6 对比分析

这里对二分搜索和四等分搜索使用的函数是$f_1(x) = 8x^3 - 2x^2 - 7x + 3$，Fibonacci搜索使用的函数是$f_2(x) = x^2 - 6x + 2$，黄金分割法使用的函数是$f_4(x) = e^x - 5x$，对于非精确线搜索检验了$f_1(X), f_3(x)$。得到的结果如下图所示。

```
D:\Anaconda\envs\pytorch36\python.exe D:/ProgrammingSpace/Python/OptimizationHomework/LineSearch.py
fibonacci搜索: expect 2.98; real: (3.0042918454935426, -6.99998158006226)
黄金分割搜索: expect 1.609; real: (1.600831365085648, -3.047004910644432)
二分搜索: (0.6290235072374344, -0.2034169653883815)
四等分搜索: (0.62890625, -0.2034144401550293)
expect x:0.63
nr: dst: [-0.07142857 -0.21428571]
nr: rst: (matrix([[-0.07142857],
        [-0.21428571]]), matrix([[1.85714286]]))
非精确搜索多项式拟合法, expect x: 0.63, 0.52; real: 0.5227272727272727
非精确搜索多项式拟合法, expect x: 0.609, 0.531; real: 1.0

Process finished with exit code 0
```

1.2 牛顿法的实现

1.2.1 牛顿法

牛顿法的基本思想是用一个二次函数去近似目标函数f(x)，然后精确地求出这个二次函数的极小点作为新的迭代点。

二次函数的选取：$f(x_k+s)\approx f(x_k)+\bigtriangledown f(x_k)^Ts+\frac{1}{2}s^T\bigtriangledown^2f(x_k)s=:m_k(s)$

迭代方向的选择：$d_k^N:=arg\,min\,m_k(s)$

牛顿方程：$\bigtriangledown^2f(x_k)s+\bigtriangledown f(x_k)=0$

牛顿方向：$d_k^N=-[\bigtriangledown^2f(x_k)]^{-1}\bigtriangledown f(x_k)=0$

牛顿步：$x_{k+1}x_k-G_k^{-1}g_k$

牛顿法的基本步骤：

- 取初始点x_0及精确参数$\varepsilon\geq0$，令$k=0$
- 若$||g_k||\leq\varepsilon$，算法终止，否则进入下一步
- 令$x_{k+1}=x_k-G_k^{-1}g_k,k=k+1$，转步骤2

1.2.2 拟牛顿法

牛顿法的搜索方向是$d^t=-G_k^{-1}g_t$，拟牛顿法要做的就是不计算二阶导及其逆矩阵，设法构造一个矩阵U，使得其逼近G_k^{-1}。假设$f(x)$是二次函数，于是Hesse矩阵的G是个常数矩阵，任意两点$x^{(k)}$和$x^{(k+1)}$处的梯度之差是：

$$\begin{aligned}&\nabla f\left(x^{(k+1)}\right)-\nabla f\left(x^{(k)}\right)=G\cdot\left(x^{(k+1)}-x^{(k)}\right)\\=>\ &x^{(k+1)}-x^{(k)}=G^{-1}\cdot\left[\nabla f\left(x^{(k+1)}\right)-\nabla f\left(x^{(k)}\right)\right]\\=>\ &x^{(k+1)}-x^{(k)}=G_{k+1}\cdot\left[\nabla f\left(x^{(k+1)}\right)-\nabla f\left(x^{(k)}\right)\right]\\&or\ \Delta x_k=G_{k+1}\cdot\Delta g_k\end{aligned}\tag{5}$$

这就是拟牛顿法，不同的拟牛顿法区别就在于如何确定U。

(1) DFP法

令G=D，假设已知D_t，希望用叠加的方式求D_{t+1},即$D_{t+1}=D_t+\Delta D_t$，带入得到：$\Delta D_t\Delta g_t=\Delta x_t-D_t\Delta g_t$

假设满足这个等式的：$\Delta D_t=\Delta x_t\cdot q_t^T-D_t\Delta g_t\cdot w_t^T$

首先，通过对比发现：$q_t^T\cdot\Delta g_t=w_t^T\cdot\Delta g_t=I_n$

其次，要保证ΔD_t是对称的，参照ΔD_t的表达式，最简单的是令：$q_t = \alpha_t \Delta x_t$，$w_t = \beta_t D_t \Delta g_t$

第二个条件带入第一个得到：

$$\alpha_t = \frac{1}{\Delta g_t^T \Delta x_t}$$
$$\beta_t = \frac{1}{\Delta g_t^T D_t \Delta g_t}$$

然后带回ΔD_t的表达式：$\Delta D_t = \frac{\Delta x_t \Delta x_t^T}{\Delta g_t^T \Delta x_t} - \frac{D_t \Delta g_t \Delta g_t^T D_t}{\Delta g_t^T D_t \Delta g_t}$

以上算法的复杂度是$O(n^2)$

DFP算法的流程：

- 给定初始点$x^{(0)}$，误差ε，令$D_0 = I_n, t = 0$
- 计算搜索方向$d^{(t)} = -D_t^{-1} g_t$
- 从点$x^{(t)}$出发，沿着$d^{(t)}$做一维搜索，获得最优步长并更新参数：

$$\begin{aligned} \lambda_t &= \arg\min_{\lambda} f\left(x^{(t)} + \lambda \cdot d^{(t)}\right) \\ x^{(t+1)} &= x^{(t)} + \lambda_t \cdot d^{(t)} \end{aligned} \tag{6}$$

- 若$\left|g_{t+1}\right| < \varepsilon$，就停止迭代，否则转5
- 计算$\Delta g = g_{t+1} - g_t, \Delta x = x^{(t+1)} - x^{(t)}$，更新G

$$D_{t+1} = D_t + \frac{\Delta x \Delta x^T}{\Delta g^T \Delta x} - \frac{D_t \Delta g \Delta g^T D_t}{\Delta g^T D_t \Delta g} \tag{7}$$

- 令$t = t + 1$，转2

(2) BFGS

拟牛顿条件是：

$$\begin{aligned} \Delta x_t &= B_{t+1}^{-1} \cdot \Delta g_t \\ \Delta g_t &= B_{t+1} \cdot \Delta x_t \end{aligned} \tag{8}$$

BFGS与BFP是对偶的，所以迭代公式只要把Δx_t和Δg_t调换一下就好。

$$\Delta B_t = \frac{\Delta g_t \Delta g_t^T}{\Delta x_t^T \Delta g_t} - \frac{B_t \Delta x_t \Delta x_t^T B_t}{\Delta x_t^T B_t \Delta x_t} \tag{9}$$

为了不计算求逆，引入Sherman-Morrison公式：

$$\begin{aligned} \left(A + uv^T\right)^{-1} &= A^{-1} - \frac{\left(A^{-1}u\right)\left(v^T A^{-1}\right)}{1 + v^T A^{-1} u} \\ => B_{t+1}^{-1} &= \left(I_n - \frac{\Delta x_t \Delta g_t^T}{\Delta x_t^T \Delta g_t}\right) B_t^{-1} \left(I_n - \frac{\Delta g_t \Delta x_t^T}{\Delta x_t^T \Delta g_t}\right) + \frac{\Delta x_t \Delta x_t^T}{\Delta x_t^T \Delta g_t} \end{aligned} \tag{10}$$

以下是BFGS的算法步骤：

- 给定初始点$x^{(0)}$,误差ε，令$B_0 = I_n, t = 0$
- 计算搜索方向$d^{(t)} = -B_t^{-1} g_t$
- 从点$x^{(t)}$出发，沿着$d^{(t)}$做一维搜索，获得最优步长并更新参数：

$$\lambda_t = \arg\min f\left(x^{(t)} + \lambda \cdot d^{(t)}\right)$$

$$x^{(t+1)} = x^{(t)} + \lambda_t \cdot d^{(t)} \tag{11}$$

- 若$|g_{t+1}| < \varepsilon$，就停止迭代，否则转5
- 计算$\Delta g = g_{t+1} - g_t, \Delta x = x^{(t+1)} - x^{(t)}$，更新$B^{-1}$，然后

$$B_{t+1}^{-1} = \left(I_n - \frac{\Delta x_t \Delta g_t^T}{\Delta x_t^T \Delta g_t}\right) B_t^{-1} \left(I_n - \frac{\Delta g_t \Delta x_t^T}{\Delta x_t^T \Delta g_t}\right) + \frac{\Delta x_t \Delta x_t^T}{\Delta x_t^T \Delta g_t} \tag{12}$$

- 令$t = t + 1$，转2

1.2.3 代码实现

代码实现见'\OptimizationHomework\NewtonMethod.py'，测试所用的函数为$f = 4x_1x_2 + 2x_1x_2 + 2x_2^2 + x_1 + x_2$

```
D:\Anaconda\envs\pytorch36\python.exe D:/ProgrammingSpace/Pytho

nr: dst: [[-0.07142857 -0.21428571]]

牛顿法搜索: rst: (matrix([[-0.07142857],
        [-0.21428571]]), 1.8571428571428572)

nr: rst: (matrix([[-0.07142857],
        [-0.21428571]]), matrix([[1.85714286]]))

dfp rst: (matrix([[-0.07140773],
        [-0.21430442]]), 1.8571428588008674)

bfgs rst: (matrix([[-0.07142916],
        [-0.2142496 ]]), 1.8571428597106978)
bfgs rst: (matrix([[-0.07141976],
        [-0.214284  ]]), 1.857142857489643)

Process finished with exit code 0
```

1.3 最速下降法

最速下降法的原理相较于牛顿法更简单，它是使用函数的一阶特性的，搜索方向是$d_k = - \bigtriangledown f(x_k)$，算法的基本框架如下：

- 取初始点x_0及精确参数$\varepsilon \geq 0$，令$k = 0$
- 若$||g_k|| \leq \varepsilon$，算法终止，否则进入下一步
- 取$d_k = -g_k$，利用线搜索步长规则选取步长α_k
- 令$x_{k+1} = x_k + \alpha_k d_k, k = k + 1$，转步骤2

代码实现见'\OptimizationHomework\OptimalGradient.py'，测试所用的函数为$f = \frac{1}{2}(x_1^2 + 2x_2^2)$，得到结果如下所示

```
D:\Anaconda\envs\pytorch36\python.exe D:/ProgrammingSpace/Python/OptimizationHomework/OptimalGradient.py
solve_direct: (matrix([[0.],
        [0.]]), 0.0)
optimal_grandient_for_f1: (matrix([[ 0.00975461],
        [-0.00243865]]), 5.3523240480965005e-05)
newton_search_for_quad(f1): (matrix([[0.],
        [0.]]), matrix([[0.]]))

Process finished with exit code 0
```

1.4 共轭梯度法

1.4.1 线性共轭方向法

共轭方向：设A是nxn对称正定矩阵，若非零向量$d_1, d_2, \ldots, d_n$关于A共轭，那么满足$d_i^T A d_j = 0, i \neq j$

线性共轭方向法基本框架：

- 取初始点x_0和搜索方向d_0满足$(d_0, g_0) < 0$，精度参数$\varepsilon \geq 0$,令$k = 0$
- 若$||g_k|| \leq \varepsilon$，算法终止，否则进入下一步
- 计算步长$\alpha_k = arg\ min\ f(x_k + \alpha d_k)$
- 令$x_{k+1} = x_k + \alpha_k d_k$
- 求d_{k+1}，使得d_{k+1}与$d_0, d_1, \ldots, d_k$关于A共轭，令$k = k + 1$，返回步骤2

1.4.2 线性共轭梯度法

迭代公式有：

线性组合系数：$\beta_{k-1} = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}}$

共轭搜索方向：$d_k = -g_k + \beta_{k-1} d_{k-1}$

精确线搜索步长：$\alpha_k = \frac{g_k^T g_k}{d_k^T A d_k}$

新的迭代点：$x_{k+1} = x_k + \alpha_k d_k$

算法的整体框架如下：

- 取初始点x_0及终止参数$\varepsilon \geq 0$，计算$g_0 = Ax_0 - b, d_0 = -g_0$，令k=0
- 若$||g_k|| \leq \varepsilon$，算法终止，否则进入下一步
- 依次计算：$\alpha_k = \frac{{g_k}^T g_k}{{d_k}^T A d_k}, x_{k+1} = x_k + \alpha_k d_k$，
  $g_{k+1} = Ax_{k+1} - b, \beta_k = \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k}, d_{k+1} = -g_{k+1} + \beta_k d_k$，令$k = k + 1$，转到步骤2.

1.4.3 实现

代码实现见'\OptimizationHomework\ConjugateGradientMethod.py'，测试的函数是$f = 1 + x_1 - x_2 + x_1^2 + 2x_2^2$，测试的结果如下所示。

```
D:\Anaconda\envs\pytorch36\python.exe D:/ProgrammingSpace/Python/Optimi
conj_grandient_method_for_f2: (matrix([[-0.50343844],
        [ 0.25429095]]), 0.6250486473720456)
conj_grandient_method_for_f2_direct: (matrix([[-0.5 ],
        [ 0.25]]), 0.625)
Fletcher_Reeves_conj.
expect: x2 = [0.4, 0.01].
Real: (matrix([[0.00513356],
        [0.00048761]]), 1.5917643339460515e-05)
Fletcher_Reeves_conj.
expect: x2 = [0.4, 0.01].
Real: (matrix([[0.00245507],
        [0.00119691]]), 8.89252726085435e-06)

Process finished with exit code 0
```

2. 机器学习优化器的实现

一般地，机器学习优化器中定义的参数有：待优化的参数θ，目标函数$J(\theta)$，以及学习率α。优化算法的分类主要有固定学习率和自适应学习率两种算法。固定学习率的算法主要有：BGD、SGD、SGDM、NAG；自适应学习率算法主要有：AdaGrad、AdaDelta、Adam、Nadam。目前使用的最多的两种优化器是SGD和Adam。

以下是各种算法的对比：

算法	优点	缺点	适用情况
BGD	目标函数为凸函数时，可以找到全局最优解	收敛速度慢，需要用到全部数据，内存消耗大	不适用于大规模数据集，不能在线更新
SGD	避免冗余数据的干扰，收敛速度加快，能过够在线学习	更新值的方差较大，收敛过程中可能落入鞍点，选择合适的学习率可能比较困难	适用于需要在线更新的模型，适用于大规模训练样本的情况（在NLP和网络表示学习很常用）
Momentum	能够在相关方向加速SGD，抑制震荡，从而加快收敛	需要人工设定学习率	适用于有可靠的初始化参数
Adagrad	实现学习率的自动更改	仍依赖于人工设置一个全局学习率，中后期的梯度接近于0，使得训练提前结束	需要加快收敛，训练复杂网络时，适用于处理稀疏梯度
Adadelta	不需要预设一个默认的学习率，训练初中期加速效果不错，可以避免参数更新时两边单位不统一的问题	在局部极小值附近震荡，可能不收敛	需要加快收敛，训练复杂网络时
Adam	速度快，对内存需求较小，为不同的参数计算不同的自适应学习率	在局部极小值附近震荡，可能不收敛	需要加快收敛，训练复杂网络时，善于处理稀疏梯度和处理非平稳目标的优点，也适用于大多非凸优化的问题，适用于大数据集和高维空间

以下实现几种常见的优化器。

2.1 BGD（批量梯度下降）

假设训练样本的总数为m，样本为$(x^1, y^1), \ldots, (x^n, y^n)$，模型参数为$\theta$，损失函数为$J(\theta)$，在第i个样本$x^i, y^i$上损失函数关于参数的梯度为$\bigtriangledown_\theta J_i(\theta, x^i, y^i)$，学习率为α，使用BGD更新参数为：

$$\theta_{t+1} = \theta_t - \frac{1}{m}\alpha_t \cdot \sum_{i=1}^{n} \nabla_\theta J_i\left(\theta, x^i, y^i\right) \tag{13}$$

由于每一次进行参数的更新，将计算整个数据集样本上的所有数据，导致运算速度很慢，尤其是对于大样本的数据集。但是由于下降的方向是总体的平均梯度，所以将会得到一个全局最优解。

代码实现,这里测试的优化目标函数是$y = x_1 + 2x^2$这一凸优化问题。

```
def bgd_multi():
    # 训练集，每个样本有2个分量
    x = np.array([(1, 1), (1, 2), (2, 2), (3, 1), (1, 3), (2, 4), (2, 3), (3,
                                                                            3)])
    y = np.array([3, 5, 6, 5, 7, 10, 8, 9])
```

```
    # 初始化
    m, dim = x.shape
    theta = np.zeros(dim)  # 参数
    alpha = 0.01  # 学习率
    threshold = 0.0001  # 停止迭代的错误阈值
    iterations = 1500  # 迭代次数
    error = 0  # 初始错误为0
    train_loss_list = []
    # 迭代开始
    for i in range(iterations):
        error = 1 / (2 * m) * np.dot((np.dot(x, theta) - y).T,
                                     (np.dot(x, theta) - y))
        train_loss_list.append(error)
        # 迭代停止
        if abs(error) <= threshold:
            break

        theta -= alpha / m * (np.dot(x.T, (np.dot(x, theta) - y)))

    print('多元变量：', '迭代次数：%d' % (i + 1), 'theta：', theta, 'error：%f' %
error)
    return train_loss_list
```

2.2 SGD（随机梯度下降）

随机梯度下降不像SGD那样每次对全部的样本进行运算，而是每次更新时只选择一个样本(x^i, y^i)计算其梯度，参数更新公式为：

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta J_i\left(\theta, x^i, y^i\right) \tag{14}$$

SGD由于每次参数更新仅仅需要一个样本的梯度，训练速度很快，由于每次迭代并不是都向着整体最优的方向，导致梯度的波动非常大，更容易从一个局部最优跳到另一个局部最优，准确率下降。所以对于SGD，学习率选择比较困难，学习率太高的话就会波动过大，所有的参数都是用的相同的学习率

代码实现：

```
def sgd():
    # 训练集，每个样本有2个分量
    x = np.array([(1, 1), (1, 2), (2, 2), (3, 1), (1, 3), (2, 4), (2, 3), (3,
3)])
    y = np.array([3, 5, 6, 5, 7, 10, 8, 9])

    # 初始化
    m, dim = x.shape
    theta = np.zeros(dim)  # 参数
    alpha = 0.01  # 学习率
    threshold = 0.0001  # 停止迭代的错误阈值
    iterations = 1500  # 迭代次数
    error = 0  # 初始错误为0
    train_loss_list = []
    # 迭代开始
    for i in range(iterations):

        error = 1 / (2 * m) * np.dot((np.dot(x, theta) - y).T, (np.dot(x, theta)
- y))
        # 迭代停止
        train_loss_list.append(error)
```

```
        if abs(error) <= threshold:
            break

        j = np.random.randint(0, m)

        theta -= alpha * (x[j] * (np.dot(x[j], theta) - y[j]))

    print('迭代次数：%d' % (i + 1), 'theta：', theta, 'error：%f' % error)
    return train_loss_list
```

2.3 Momentum-SGD（带动量的SGD）

Momentum的算法思想是：参数更新时在一定程度上保留之前更新的方向，同时又利用当前batch的梯度微调最终的更新方向，简言之就是通过积累之前的动量来加速当前的梯度。其算法的迭代公式为：

$$
\begin{aligned}
g &= momentum * g + \alpha \left(h_\theta \left(x^i\right) - y^i\right) x^i \\
\theta_j &= \theta_j - g
\end{aligned}
\tag{15}
$$

代码实现：

```
def Momentum_sgd():
    # 训练集，每个样本有三个分量
    x = np.array([(1, 1), (1, 2), (2, 2), (3, 1), (1, 3), (2, 4), (2, 3), (3,
                                                                            3)])
    y = np.array([3, 5, 6, 5, 7, 10, 8, 9])

    # 初始化
    m, dim = x.shape
    theta = np.zeros(dim)  # 参数
    alpha = 0.01  # 学习率
    momentum = 0.1  # 冲量
    threshold = 0.0001  # 停止迭代的错误阈值
    iterations = 1500  # 迭代次数
    error = 0  # 初始错误为0
    gradient = 0  # 初始梯度为0

    train_loss_list = []
    # 迭代开始
    for i in range(iterations):
        j = i % m
        error = 1 / (2 * m) * np.dot((np.dot(x, theta) - y).T,
                                     (np.dot(x, theta) - y))
        train_loss_list.append(error)
        # 迭代停止
        if abs(error) <= threshold:
            break

        gradient = momentum * gradient + alpha * (x[j] *
                                                  (np.dot(x[j], theta) - y[j]))
        theta -= gradient

    print('迭代次数：%d' % (i + 1), 'theta：', theta, 'error：%f' % error)
    return train_loss_list
```

2.4 Adam算法

Adam是一种自适应学习率的方法，在Momentum的一阶矩估计的基础上加上了二阶矩估计。利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率。

Adam的算法流程如下，具体可以见论文《Adam : A Method for Stochastic Optimization》。

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
  $v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

```
def adam():
    # 训练集，每个样本有三个分量
    x = np.array([(1, 1), (1, 2), (2, 2), (3, 1), (1, 3), (2, 4), (2, 3), (3,
                                                                            3)])
    y = np.array([3, 5, 6, 5, 7, 10, 8, 9])

    # 初始化
    m, dim = x.shape
    theta = np.zeros(dim)  # 参数
    alpha = 0.01  # 学习率
    momentum = 0.1  # 冲量
    threshold = 0.0001  # 停止迭代的错误阈值
    iterations = 3000  # 迭代次数
    error = 0  # 初始错误为0

    b1 = 0.9  # 算法作者建议的默认值
    b2 = 0.999  # 算法作者建议的默认值
    e = 0.00000001  #算法作者建议的默认值
    mt = np.zeros(dim)
    vt = np.zeros(dim)
    train_loss_list = []
    for i in range(iterations):
        j = i % m
        error = 1 / (2 * m) * np.dot((np.dot(x, theta) - y).T,
                                     (np.dot(x, theta) - y))
        train_loss_list.append(error)
        if abs(error) <= threshold:
            break

        gradient = x[j] * (np.dot(x[j], theta) - y[j])
        mt = b1 * mt + (1 - b1) * gradient
```

```
        vt = b2 * vt + (1 - b2) * (gradient**2)
        mtt = mt / (1 - (b1**(i + 1)))
        vtt = vt / (1 - (b2**(i + 1)))
        vtt_sqrt = np.array([math.sqrt(vtt[0]),
                             math.sqrt(vtt[1])])  # 因为只能对标量进行开方
        theta = theta - alpha * mtt / (vtt_sqrt + e)

    print('迭代次数：%d' % (i + 1), 'theta：', theta, 'error：%f' % error)
    return train_loss_list
```

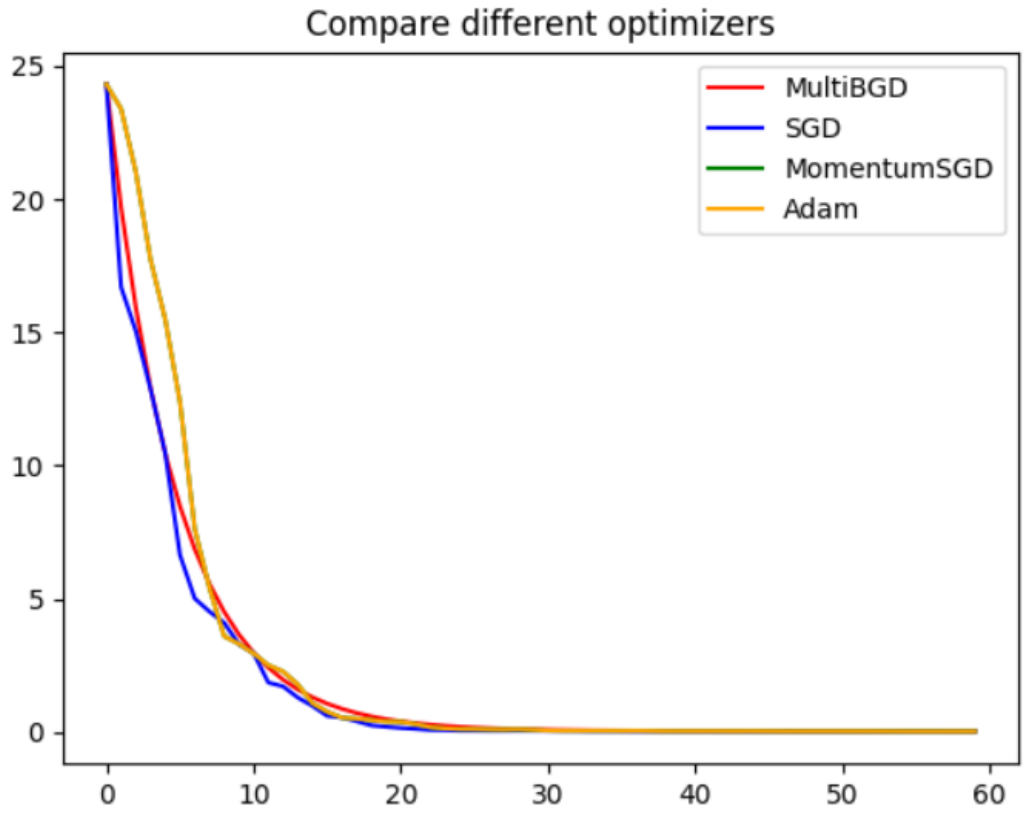
2.5 对比

对同一个函数$y = x_1 + 2x^2$进行这四种优化器的测试，得到如下的结果：

```
D:\Anaconda\envs\pytorch36\python.exe D:/ProgrammingSpace/Python/OptimizationHomework/OptimizerOfML.py
单变量： 迭代次数： 19 theta： 0.999842 error1： 0.000000
多元变量： 迭代次数：454 theta： [1.01339617 1.98981777] error：0.000100
迭代次数：500 theta： [1.01293366 1.99003996] error：0.000094
迭代次数：385 theta： [1.01340148 1.98984903] error：0.000100
迭代次数：1987 theta： [1.01326533 1.98965863] error：0.000100

Process finished with exit code 0
```

将损失函数画在同一副图中，得到以下的结果：



3. 最优化在机器学习中的应用

这里主要介绍的是SVM算法的实现。支持向量机(SVM)是一种监督学习方式对数据进行二元分类的广义线性分类器，其决策边界时对学习样本求解的最大边距超平面。

超平面是在样本空间中的一个平面，在多维空间中，超平面可以用向量的形式表现为$\vec{w}\vec{x}+b=0$，记作$(\vec{w},b)$的形式，其中$\vec{w}$是法向量，而b为移位项。考虑任意一点$\vec{x}$到超平面之间的距离，可以表示为：$\gamma=\frac{\vec{w}\cdot\vec{x}+b}{||\vec{w}||}$，超平面能将样本空间划分为两个部分。假设超平面$(\vec{w},b)$将所有的样本正确分类，即对于任意的$(\vec{x_i},y_i)\in D$，若$y_i=+1$，那么$\vec{w}\cdot\vec{x}+b<0$。

对于$(\vec{x_i},y_i)\in D$，可以将上面的距离公式的分子部分带入y_i改写成：$\gamma_i=\frac{y_i(\vec{w}\cdot\vec{x}+b)}{\|\vec{w}\|}$，定义间隔γ为所有样本中离超平面最近的那个，即$\gamma=min\ \gamma_i$，那么$\gamma_i\geq\gamma$总是成立的，带入上式可得：$\frac{y_i(\vec{w}\cdot\vec{x}+b)}{\|\vec{w}\|}\geq\gamma$。

因此，SVM的问题就变成了找到最大间隔的超平面，使得γ最大，即：

$$\begin{aligned}&\max_{(\vec{w},b)}\min_i\frac{1}{\|\vec{w}\|}\left|\vec{w}\cdot\vec{x}_i+b\right|\\&s.t.\quad y_i\left(\vec{w}\cdot\vec{x}_i+b\right)>0,\quad i=1,2,\ldots,m\end{aligned}\tag{16}$$

因为间隔始终大于0.即$|\vec{w}\cdot\vec{x_i}+n|>0$，则可以通过缩放$(\vec{w},b)$使得：$\min_i\left|\vec{w}\cdot\vec{x}_i+b\right|=1$，那么上面的问题就变成了$max\ \frac{1}{||\vec{w}||}$，上式可以重写为：

$$\begin{aligned}&\min\frac{1}{2}\|\vec{w}\|^2\\&s.t.\quad y_i\left(\vec{w}\cdot\overrightarrow{x_i}+b\right)\geq1,\quad i=1,2,\ldots,m\end{aligned}\tag{17}$$

是一个凸二次规划问题，这样就能用现有的求解优化问题的方法。

将SVM的问题用拉格朗日函数的形式重写为：

$$\mathcal{L}(\vec{w},b,\vec{\alpha})=\underbrace{\frac{1}{2}\|\vec{w}\|^2}_{f(\vec{w},b)}+\sum\alpha_i\underbrace{\left(1-y_i\left(\vec{w}\cdot\vec{x}_i-b\right)\right)}_{h_i(\vec{w},b)\leq0}\tag{18}$$

令$L(\vec{w},b,\vec{\alpha})$对于$\vec{w}$和$b$的偏导数等于0：

$$\begin{cases}\frac{\partial\mathcal{L}}{\partial\vec{w}}=\vec{w}-\sum\alpha_iy_i\vec{x}_i=0\\\frac{\partial\mathcal{L}}{\partial b}=\sum\alpha_iy_i=0\end{cases}\tag{19}$$

可得：

$$\begin{aligned}\vec{w}&=\sum\alpha_iy_i\vec{x}_i\\0&=\sum\alpha_iy_i\end{aligned}\tag{20}$$

将上式带入$L(\vec{w},b,\vec{\alpha})$可得：

$$\begin{aligned}\mathcal{L}(\vec{w},b,\vec{\alpha})=&\frac{1}{2}\left(\sum\alpha_iy_i\vec{x}_i\right)\cdot\left(\sum\alpha_iy_i\overrightarrow{x_i}\right)\\&+\sum\alpha_i\left(1-y_i\left(\left(\sum\alpha_iy_i\overrightarrow{x_i}\right)\cdot\overrightarrow{x_i}-b\right)\right)\\=&\sum_{i=1}^m\alpha_i-\frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m\alpha_i\alpha_jy_iy_j\left(\overrightarrow{x_i}\cdot\overrightarrow{x_j}\right)\end{aligned}\tag{21}$$

这样对偶问题就变成：

$$\begin{aligned}&\max_{\vec{\alpha}\geq0}\min_{\vec{z}}\mathcal{L}(\vec{z},\vec{\alpha})\\\Longrightarrow&\max_{\vec{\alpha}\geq0}\sum_{i=1}^m\alpha_i-\frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m\alpha_i\alpha_jy_iy_j\left(\vec{x}_i\cdot\vec{x}_j\right)\\&\text{s.t.}\quad\sum\alpha_iu_i=0,\alpha_i>0\end{aligned}\tag{22}$$

解出$\vec{\alpha}$后，求出$\vec{w}$与b即可得到模型：

$$\begin{aligned} f(\vec{x}) &= \vec{w} \cdot \vec{x} + b \\ &= \sum_{i=1}^{m} \alpha_i y_i \left(\vec{x}_i \cdot \vec{x}\right) + b \end{aligned} \tag{23}$$

还需要满足上面提到的KKT条件：

$$\begin{cases} \alpha_i \geq 0 \\ y_i f\left(\vec{x}_i\right) - 1 \geq 0 \\ \alpha_i \left(y_i f\left(\vec{x}_i\right) - 1\right) = 0 \end{cases} \tag{24}$$

软间隔

上面我们考虑的都是数据可以分隔的情况，那么当数据集并不是完全能被超平面分隔，即有些点不满足约束$y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1$，那么可以设定一个惩罚项：

$$\xi_i = \max\left\{1 - y_i\left(\vec{w} \cdot \vec{x}_i + b\right), 0\right\} \tag{25}$$

那么原来的 SVM 的优化问题就会变成：

$$\begin{aligned} &\min_{\vec{w},b} \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{m} \max\left\{1 - y_i\left(\vec{w} \cdot \vec{x}_i + b\right), 0\right\} \\ &s.t. \quad 1 - y_i\left(\vec{w} \cdot \vec{x}_i + b\right) \leq 0, \quad i = 1, 2, \ldots, m \end{aligned} \tag{26}$$

其中的 $\max(1 - z, 0)$ 称为“折页损失”（hinge loss）。接着引入“松弛变量” $\xi_i \geq 0$，将上式重写为：

$$\begin{aligned} & \min_{\vec{w},b} \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{m} \xi_i \\ \text{s.t.} \quad & y_i\left(\vec{w} \cdot \vec{x}_i + b\right) \geq 1 - \xi_i \\ & \xi_i \geq 0, i = 1, 2, \ldots, m \end{aligned} \tag{27}$$

核方法

当遇到样本是非线性可分时，一般的线性 SVM 就无法很好的分类，这时候可以采用核技巧（kernel trick），将样本映射至高维空间，变成高维空间中线性可分的即可。

考虑一个映射函数$\phi(\vec{x})$，将 d*d* 维特征映射至 m 维：

$$\phi(\vec{x}) : \mathbb{R}^d \to \mathbb{R}^m \tag{28}$$

定义核函数：

$$K\left(\vec{x}_i, \vec{x}_j\right) = \phi\left(\vec{x}_i\right) \cdot \phi\left(\vec{x}_j\right) \tag{29}$$

核技巧的思想是用核函数 K 避免在高维 m*m* 空间中计算映射函数的内积，这样能够极大简化运算。函数 K 能够满足成为核函数的充分必要条件是 K是半正定，即：

$$\sum_{i,j=1}^{m} c_i c_j K\left(\vec{x}_i, \overrightarrow{x_j}\right) \geq 0 \tag{30}$$

常用的核函数有多项式核函数（Polynomial function）：

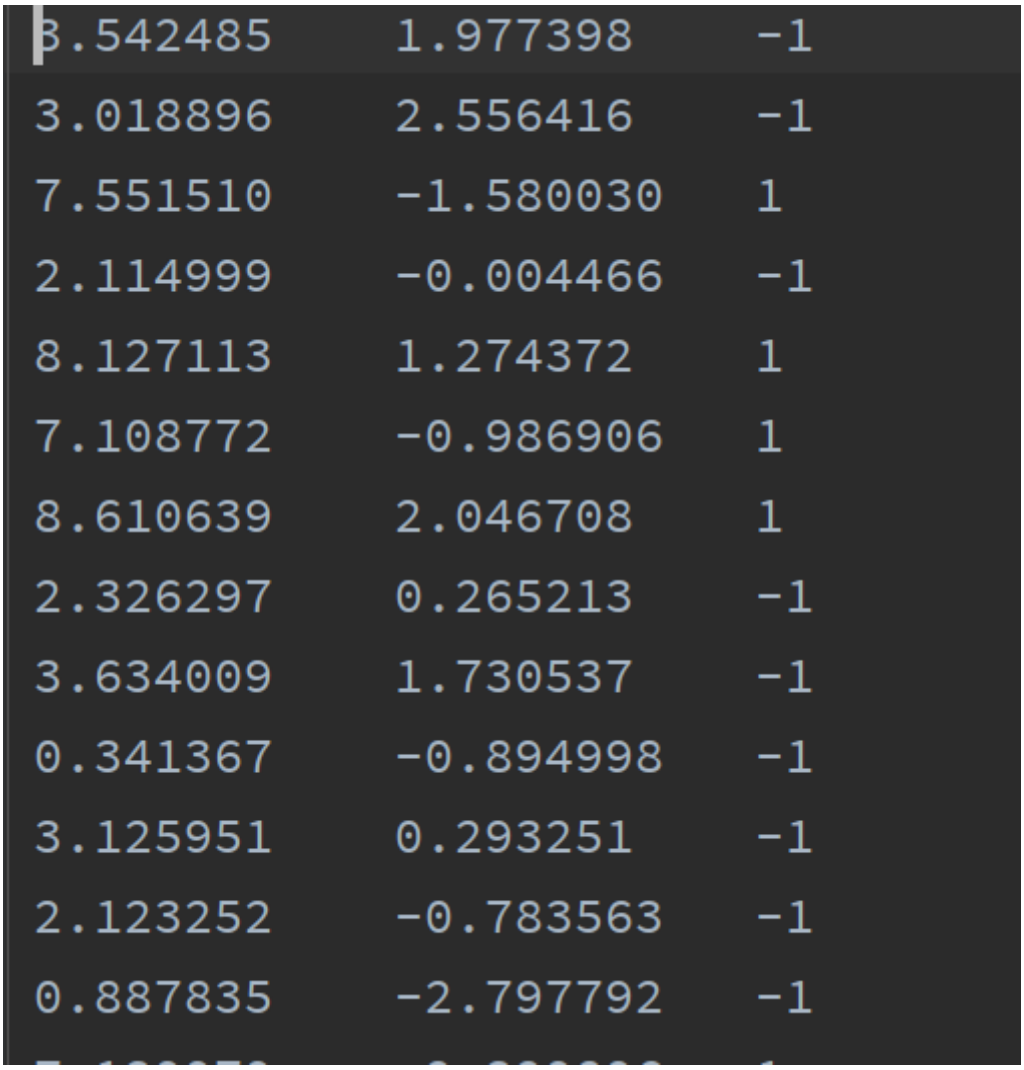
$$K\left(\vec{x}_i, \vec{x}_j\right) = \left(\overrightarrow{x_i} \cdot \overrightarrow{x_j} + 1\right)^d \tag{31}$$

以及高斯核函数（Guassian radial basis function），通常称为 RBF Kernel：

$$K\left(\vec{x}_i, \vec{x}_j\right) = \exp\left(-\frac{\left\|\vec{x}_i - \vec{x}_j\right\|^2}{2\sigma^2}\right) \tag{32}$$

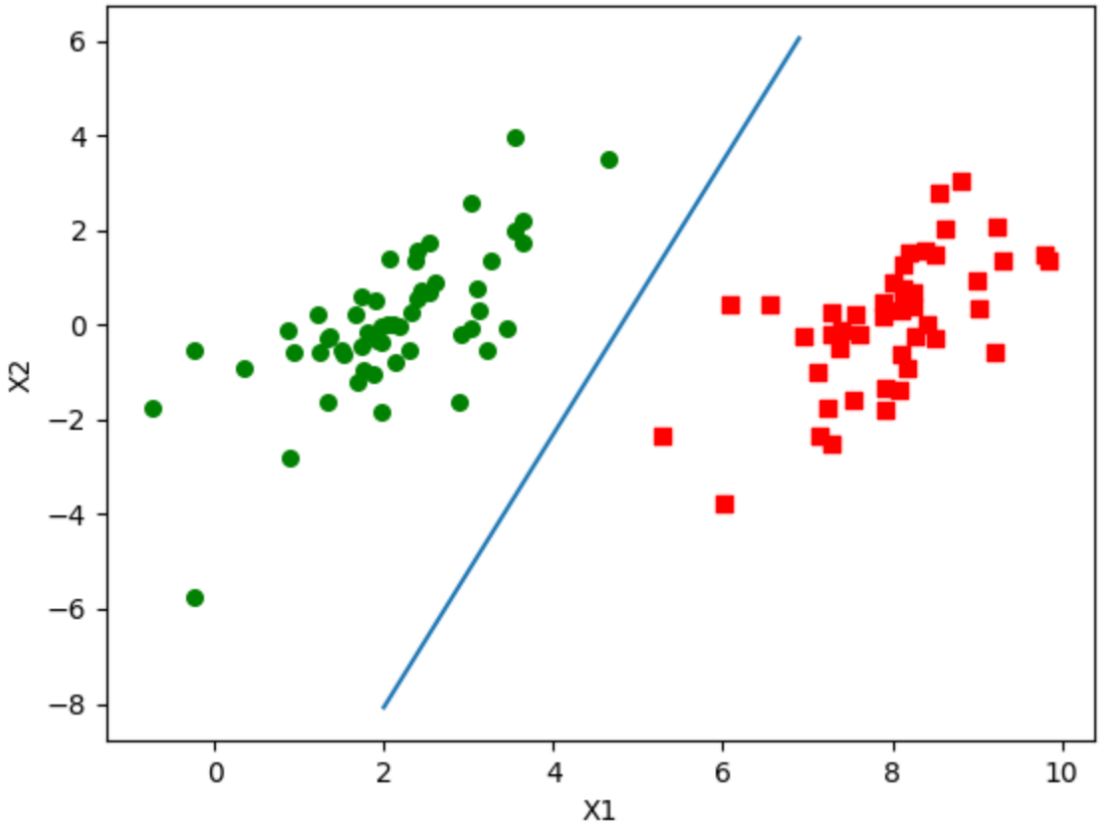
代码实现

这里使用SVM实现了一个简单的二分类问题，使用的数据集如下图所示，标签只有(-1,1),两个维度表示其特征。



3.542485	1.977398	-1
3.018896	2.556416	-1
7.551510	-1.580030	1
2.114999	-0.004466	-1
8.127113	1.274372	1
7.108772	-0.986906	1
8.610639	2.046708	1
2.326297	0.265213	-1
3.634009	1.730537	-1
0.341367	-0.894998	-1
3.125951	0.293251	-1
2.123252	-0.783563	-1
0.887835	-2.797792	-1

实现的代码位于'OptimizationHomework\OptimizationInMachineLearning\svm-Demo'，实验的结果如下所示，SVM能很好的分类数据集中的数据。


6
4
2
0
−2
−4
−6
−8
X2
0
2
4
6
8
10
X1