Assignment 2

方煊杰

复旦大学

16307130335@fudan.edu.cn

准备

package requirements:

- matplotlib
- numpy

```
#install packages
$pip install matplotlib
$pip install numpy
```

实现

二分类问题

1. 最小二乘法（Least square model）

```
def least_square(d):
    T = []
    X = []
    for i in range(0, len(d.X)):
        T.append(([1,0] if(d.y[i]) else [0, 1]))
        X.append([1, d.X[i][0], d.X[i][1]])
    X_pinv = np.linalg.pinv(X)
    W = X_pinv.dot(T)
    #[W1-W2]^T * [1, x, y]
    k = (W[1][0]- W[1][1])/(W[2][1] - W[2][0])
    b = (W[0][0]- W[0][1])/(W[2][1] - W[2][0])
    return k, b

def straight_fun(k, b, x):
    return k*x+b

def main():
    d = get_linear_seperatable_2d_2c_dataset()
    k, b = least_square(d)
```

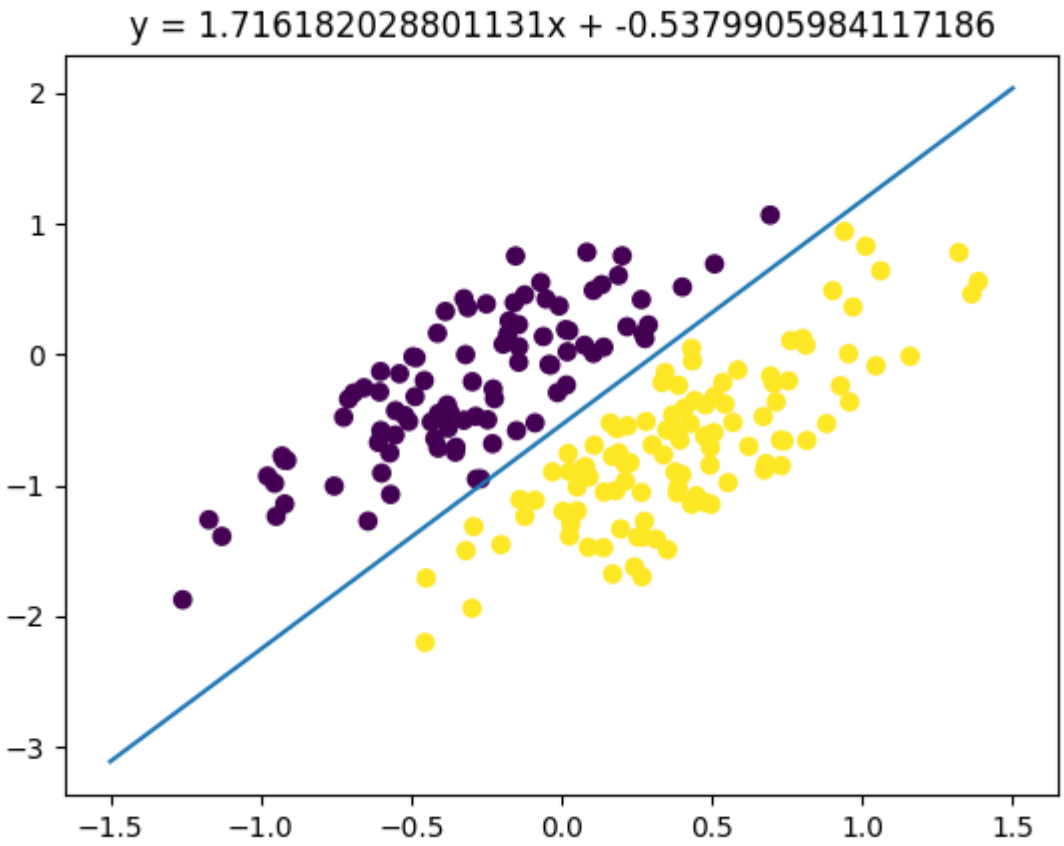
```
    plt.plot([-1.5, 1.5], [straight_fun(k, b ,-1.5), straight_fun(k, b ,1.5)])
    d.plot(plt).show()
```

1. `T.append(([1,0] if(d.y[i]) else [0, 1])), X.append([1, d.X[i][0], d.X[i][1]])`:获得sample向量集X和目标向量集T.
2. `X_pinv = np.linalg.pinv(X),W = X_pinv.dot(T)`:

$$W = X^{pinv} \times T$$

3. 根据超平面公式获得直线参数k,b:

$$(w_k - w_j)^T x = 0$$



2. 感知器算法（Perceptron algorithm）

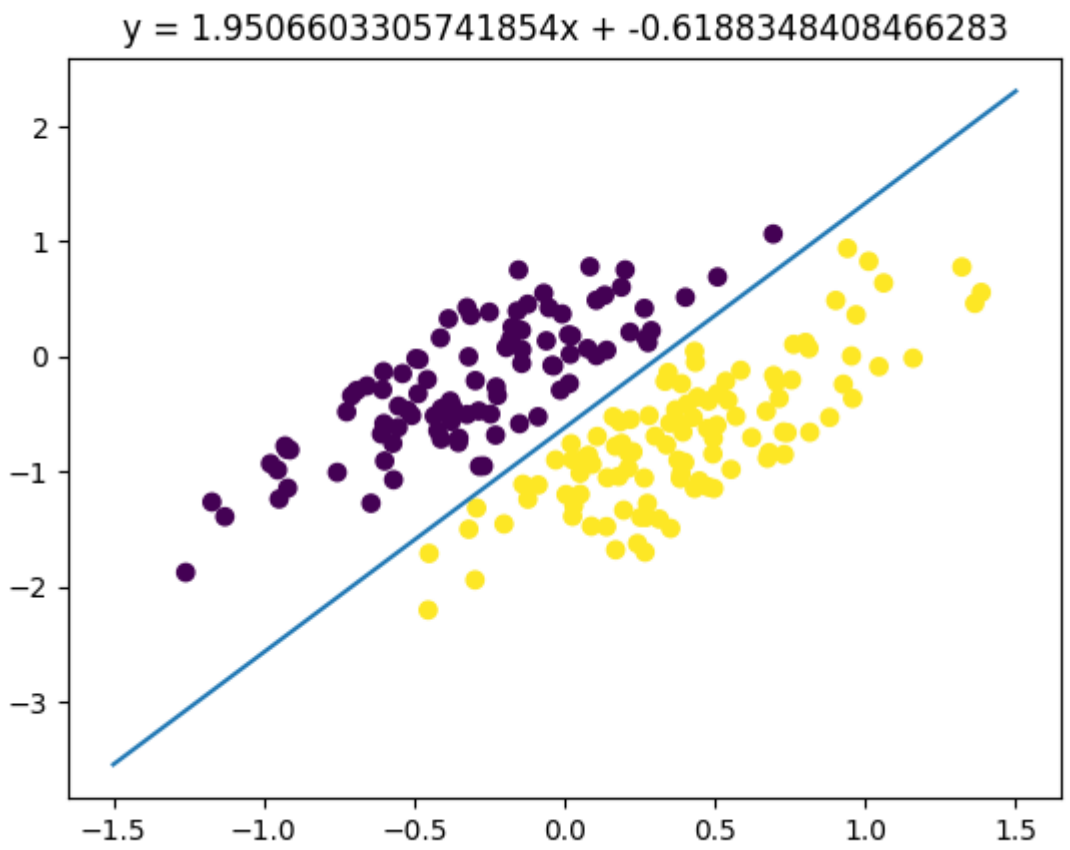
```
def normalization(X):
    # mean
    u = np.mean(X, axis=0)
    # standard
    v = np.std(X, axis=0)
    X = (X - u) / v
    X = np.hstack((np.ones((X.shape[0], 1)), X))
    return X

def perceptron(d):
    num = len(d.X)
```

```
    W = (np.random.randn(1, 3))[0]
    y = d.y
    X = normalization(d.X)
    while 1:
        loss_exist = 0
        for i in range(0, num):
            E = W[0] + W[1]* d.X[i][0] + W[2]* d.X[i][1]
            E = E * (1 if(y[i]) else -1)
            if E < 0:
                loss_exist = 1
                W += (X[i] if(y[i]) else -X[i])

        if loss_exist == 0:
            k = -W[1]/W[2]
            b = -W[0]/W[2]
            return k, b
```

1. 使用`numpy`工具正则化sample向量
2. `W = (np.random.randn(1, 3))[0]`:随机生成参数矩阵W, 规模为（1 * 3)
3. 通过检测对所有的sample是否都满足下述条件来判断是否结束训练:

$$w \times x_n^T \times t_n > 0$$

4. 如果存在sample不满足条件，则使用随机梯度下降算法更新参数矩阵



多分类逻辑回归问题

1. 实现源文件多热点向量化以及目标分类单热点向量化

Transform the documents into multi-hot vector representation, and also the targets into one-hot representation

1.1: 分割字符串

```
import string

def split_doc(s):
    for i in s:
        if i in string.punctuation:
            s = s.replace(i, "")
        elif i in string.whitespace:
            s = s.replace(i, " ")
    ans = s.split()
    # to lower case
    for i in range(0, len(ans)):
        ans[i] = ans[i].lower()
    return ans
```

1. 使用`string.functuation`以及`string.replace`来忽略所有的标点符号
2. 使用`string.whitespace`以及`string.replace`将所有特殊类符号（\n, \t...) 替换为空格符
3. 使用`string.split()`将字符串切割为单词list
4. 最后遍历所有的单词，调用`string.lower()`将所有的单词转为小写

1.2: 获得单词表

```
def handler_doc(data):
    data_set = []
    data_count = {}
    store_key = []
    vocabulary = {}
    for s in data:
        ans = split_doc(s)
        for x in ans:
            data_count[x] = data_count.get(x, 0) + 1
            # only choose the words which occur at least
            # 10 times in the overall training set.
            if data_count[x] == 10:
                store_key.append(x)
        data_set.append(ans)
    store_key.sort()
    count = 0
    for key in store_key:
        vocabulary[key] = count
        count += 1
    data_vec = to_vector(data_set, vocabulary)
```

```
    return data_vec, vocabulary
```

- 对于每个切分的单词，用字典统计出现次数:`data_count[x] = data_count.get(x, 0) + 1`
- 当单词出现`min_times=10`时，存进单词表(减少向量维度)
- 给单词表排序，获得字典

1.3: 获得多热点模式向量

```
def to_vector(data_set, vocabulary):
    data_vec = []
    for s in data_set:
        vec = [0]*len(vocabulary)
        for i in s:
            if i in vocabulary:
                vec[vocabulary[i]] = 1
        vec.insert(0, 1) # 将bias 加到向量头部
        data_vec.append(vec)
    data_vec = np.array(data_vec)
    print("Generate doc_vec, Shape: ", data_vec.shape)
    return data_vec
```

- 对于每个sample, 构建一个与字典长度相同的向量`vec = [0]*len(vocabulary)`
- 每个sample中出现的词语，在向量中相应位置设为1，得到multi-hot vector
- 在每个向量的头部增加`bias = 1`，*（为了减少求导时额外考虑的`b`）*
- 最终得到的数据向量集的规模为:

$$N \times (D+1)$$

N为$sample$的数量，D为向量维度（即字典长度）

1.4: 获得单热点目标向量

```
def get_target(data, num):
    target = []
    for i in range(0, len(data)):
        vec = [0]*num
        vec[data[i]] = 1
        target.append(vec)
    target = np.array(target)
    print("Generate Target(one hot), Shape: ", target.shape)
    return target
```

- 对于每个sample，构建一个长度为类型数量的向量：`vec = 0*num`
- 在这个数据集中，类型名直接为数字，所以不用考虑对类型的映射，可以直接把类型名对应到向量 index,构建单热点目标向量：`vec[data[i]] = 1`
- 最终得到的目标向量集规模为:

$$N \times K$$

N为$sample$数量，K为类型数量

2. 使用numpy实现逻辑回归分类算法

Differentiate the loss function for logistic regression,and then implement the calculation in vectorized style in numpy.

ps:下面所有的公式和书本以及作业介绍所给出的有所不同，是针对自己的代码所复现的

2.1: 获得类别概率分布矩阵(Ynk)

```
def softmax(X, W):
    data = np.exp(np.dot(X, W.T))
    sum_data = np.sum(data, axis=1).T
    data = data/sum_data[:,None]
    return data
```

因为题目要求不使用显式循环，所以在计算**概率分布**矩阵时使用numpy工具，在这里解释一下这短短几行代码的计算原理。

- 首先获得"激活" a[n] (1 * K):

$$a_n = x_n \times W^T \quad [x_n : 1 \times (D+1)\,;\, W : K \times (D+1)]$$

所以对于所有的sample向量使用矩阵计算：`a = np.dot(X, W.T)` (N * K)

- 使用softmax函数获得每个分类在所有分类中的概率：

$$p(C_k \mid x_n) = y_k(x_n) = \frac{e^{a_{nk}}}{\sum_{k=1}^{K} e^{a_{nk}}}$$

`data = np.exp(a)`: 对矩阵里的每个元素求e的指数运算

`sum_data= np.sum(data, axis=1).T`: 将矩阵按照每一行求和，得到存放结果的行向量(1 * N)

`data = data/sum_data[:,None]`: 等同于每一行的元素除以每一行的和

2.2: 交叉熵损失函数（cross entropy loss function）

```
def cross_entropy(data, target, step = 500):
    W = np.random.randn(len(target[0]), len(data[0]))
    print("Generate W, Shape: " + str(W.shape))
    for _ in range(0, step):
        loss = 0
        ynk = softmax(data, W)
        for i in range(0, len(data)):
            y_Xi = ynk[i]  # 1 X 4
            loss -= np.dot(target[i].T, np.log(y_Xi))
        print("step:", _," loss: ", loss/len(data))
        W = W - gradient_cross(ynk, target, data)
    return W
```

- 随机初始化参数矩阵W: `W = np.random.randn(len(target[0]), len(data[0]))`, 规模为（K * (D+1)）
- 根据交叉熵函数计算出损失 `loss -= np.dot(target[i].T, np.log(y_Xi))`

$$E = -\frac{1}{N}\sum_{i=1}^{N} t_i \times ln(y_i)$$

$$t_i : Target[i] \quad y_i : Ynk[i]$$

2.3：梯度计算（gradient）

```
def gradient_cross(ynk, target, data, a=0.001):
    matrix = (ynk - target).T
    gradient = np.dot(matrix, data)
    #print("Generate gradient, Shape: ", gradient.shape)
    return a * gradient
```

根据损失函数对每个分类参数向量Wk的求导结果：

$$\bigtriangledown E_k = \sum_{n=1}^{N}(y_{nk} - t_{nk}) \times x_n = (y_k - t_k)^T \times X$$

$$y_k, t_k : Ynk, Target \text{ 矩阵第 } k \text{列；} \ X : sample \text{向量集（} N * (D+1) \text{）}$$

整体矩阵结果可以写为：

$$\bigtriangledown E = (y - t)^T \times X$$

2.4: 使用正则化来防止过拟合

- 叉熵损失函数变更为：

$$E = -\frac{1}{N}\sum_{i=1}^{N} t_i \times ln(y_i) + \lambda||W||^2$$

- 梯度函数变为：

$$\bigtriangledown E = (y - t)^T \times X + 2\lambda W$$

Problem 1: 是否增加关于bias的惩罚项?

正则化是为了减少单个特征对于学习的影响从而避免陷入局部极小值，由于bias不属于sample原本的特征而只是偏置，所以不增加关于bias的惩罚项

Problem 2: 如何判断梯度计算的正确性?

比较定义梯度和计算梯度的差别,二者差距在一个较小的值（0.0001）

使用导数定义：

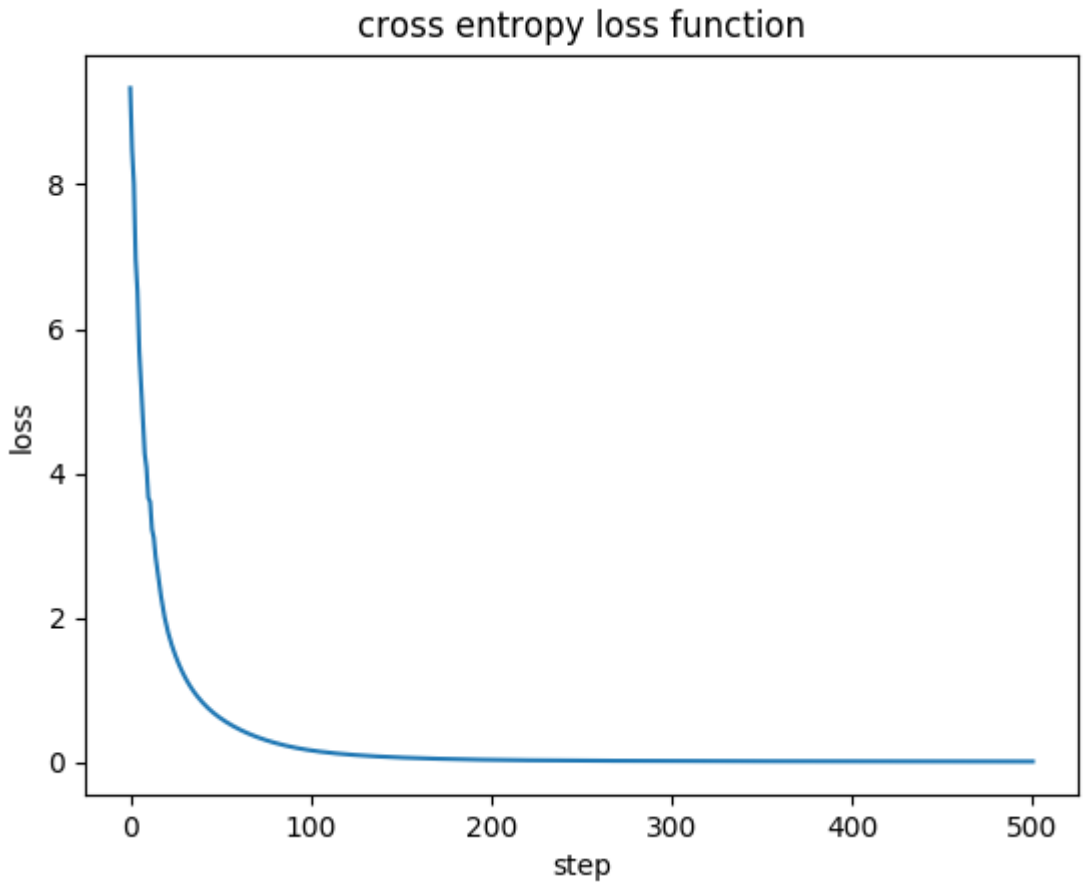
$$dE_{nk} = \frac{E(W + \varepsilon_{nk}) - E(W)}{\varepsilon}$$

对于每个分量检测与梯度函数对应分量的差值是否都在较小差值之内

```
def check(gradient, dE):
    for i in range(0, len(gradient[0])):
        for j in range(0, len(gradient)):
            if gradient[i][j] - dE[i][j] > 0.0001
                return False
    return True
```

3. 完成逻辑回归训练，给出损失分布图

Finish the training for the logistic regression model on the training dataset, include the plot for the loss curve you obtained during the training of the model.



Problem 3: 如何确定学习率参数

- 为了将W矩阵中的元素都保持在 [0, 1] 中，输出梯度矩阵看了一下元素数值，因为在这道 题目中向量的维度为2000多，输出矩阵单个元素最大值可达到几十，因此选取学习参数为**0.001**
- 在运行过程中发现如果学习参数较大，导致W矩阵元素过大，会使得`np.exp()，np.log()`调用返回`nan`值，如果参数太小，loss下降得巨慢。

Problem 4: 何时终止迭代训练

- 设立迭代次数`step`
- 设立损失标准，如：当`loss<0.001`时,停止迭代

在这道题里我使用了第一种，通过打印出每次迭代时loss的变化从而确定较好的迭代次数（从图里也可以很清楚地看出来）

4. 随机梯度下降或批量梯度下降

stochastic gradient descent or batched gradient descent

4.1: 修改损失函数代码

```
def cross_entropy(data, target, step = 500):
    ......
    ......
    for _ in range(0, step):
        loss = 0
        n_list = np.random.randint(len(data), size=50)
        for i in range(0, len(data)):
            if i in n_list:
                y_Xi = ynk[i]  # 1 X 4
                loss -= np.dot(target[i], np.log(y_Xi).T)
        print("step: ", _," loss: ", loss/len(data))
        loss_vector.append(loss/len(data))
        W = W - gradient_cross(ynk, target, data)
    return W, loss_vector

def gradient_cross(ynk, target, data, n_list, a=0.001,):
    matrix = (ynk - target).T
    for i in range(0, len(matrix)):
        if i not in n_list:
            data[i] = [0]* len(data[0])
    gradient = np.dot(matrix, data)
    #print("Generate gradient, Shape: ", gradient.shape)
    return a * gradient
```

- `n_list = np.random.randint(len(data), size=50)`: 随机生成 [0, sizeof (sample)) 区间内部分整数
- 计算损失时，只对下标属于`n_list`的sample进行计算
- 计算梯度时，将下标不属于`n_list`的sample置为零向量，即不参与对参数矩阵的更新作用

Problem 5: 从另外两种梯度下降方法中观察到了什么?

- 每一次迭代计算loss及更新W矩阵的时间加快
- loss 下降的速率比标准梯度下降快很多
- loss曲线图相对标准梯度有较大波动

Problem 6: 比较三种梯度下降方法的优缺点

方法	优点	缺点
标准梯度下降	每一次更新的幅度大，在样本不多的情况下，收敛速度快；每次更新保证朝着正确的方向	每一次学习时间长；数据量大时消耗内存大；有多个局部极小值时不能保证收敛至全局最小值
批量梯度下降	在提高收敛速度的情况下，也基本保持了每次更新朝着正确的方向	选择批量数量时需要多次试验
随机梯度下降	在样本量大的情况下收敛过程快；学习速率高；在有多个局部极小值时能够较好得收敛至全局最小值	每次更新可能不朝着正确方向进行，导致波动；增加学习迭代次数

使用

1. 工具

使用`argparse`工具来封装整个作业，便于检验输出结果

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--methods", "-m", default="lsq", choices=["lsq", "ptr",
"logistic"])
    parser.add_argument("--step", "-s", default=500, type=int)
    parser.add_argument("--gMethods", "-g", default=1, choices=[1, 2], type=int)
    parser.add_argument("--number", "-n", default=50, type=int)
    args = parser.parse_args()
    if args.methods != "logistic":
        part1(args.methods)
    else:
        part2(args.step, args.gMethods, args.number)
```

2. 示例

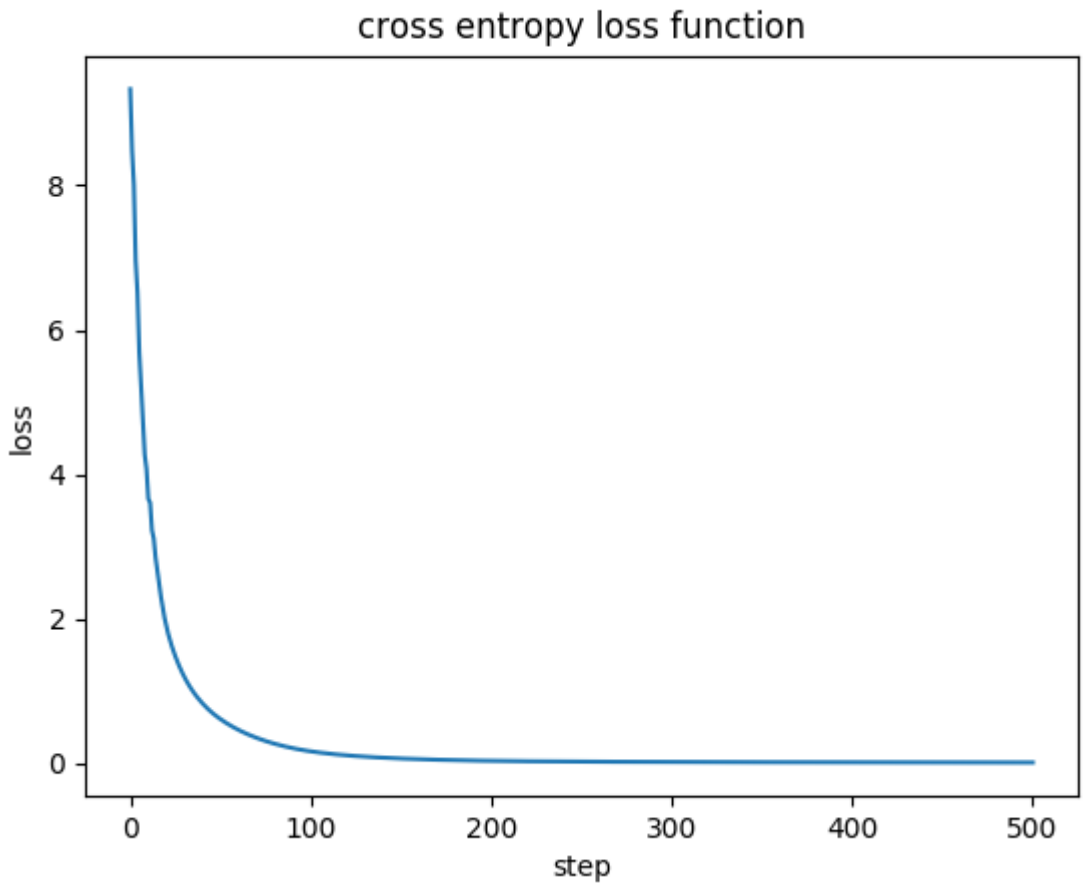
```
# least square model
$ python source.py -m lsq
#perceptron algorithm
$ python source.py -m ptr

#标准梯度下降
$ python source.py -m logistic -g 1
#批量梯度下降
$ python source.py -m logistic -g 2 -n 50
#随机梯度下降
$ python source.py -m logistic -g 2 -n 1
```

结果

```
def forecast(data, W, real_target):
    correct = 0
    ynk = softmax(data, W)
    for i in range(0, len(data)):
        res = ynk[i].tolist()
        index = res.index(max(res))
        if index == real_target[i]:
            correct += 1
    print("correct: ", correct, "/", len(data), "=", correct/len(data))
```

1.标准梯度下降

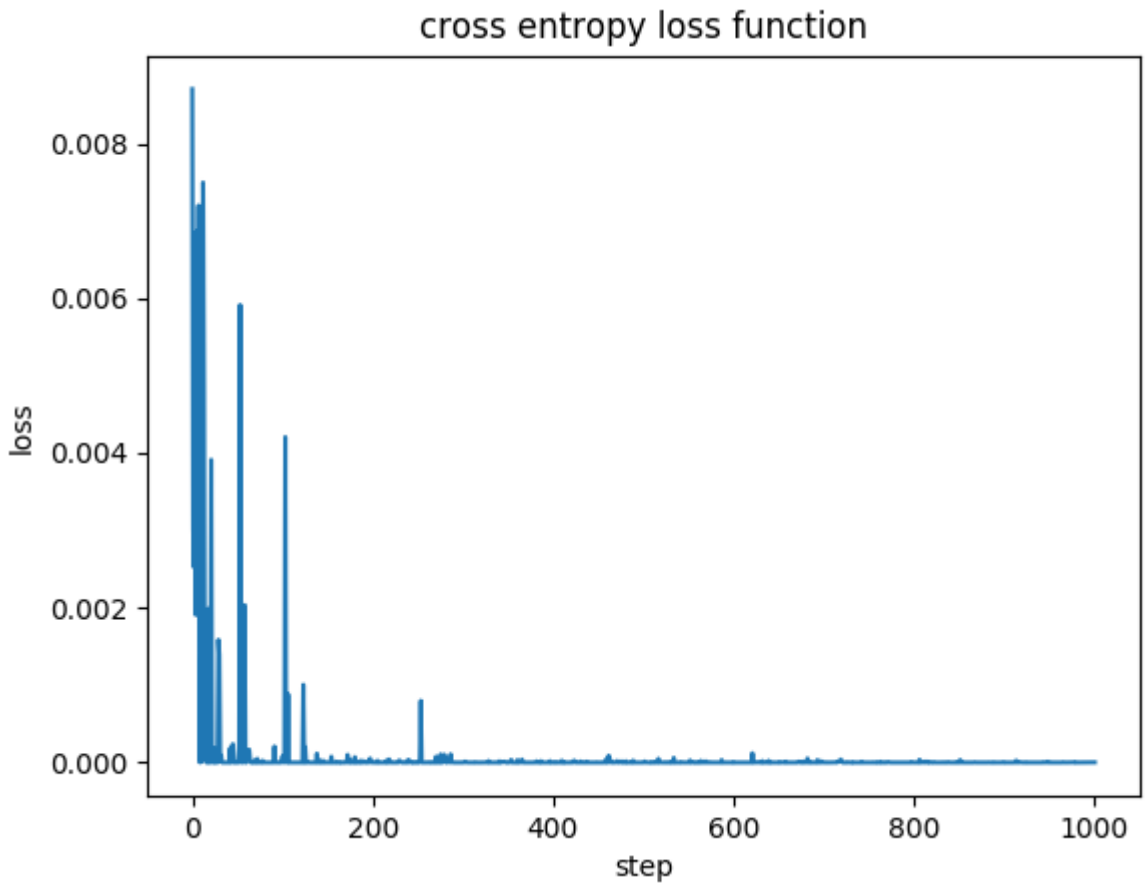
```
$ python source.py -m logistic -s 1000
```




```
step:  997  loss:  0.002991996846352663
step:  998  loss:  0.002988714089197318
step:  999  loss:  0.0029854388550205322
Generate doc_vec, Shape:  (1496, 5635)
correct:  1208 / 1496 = 0.8074866310160428
```

2. 批量梯度下降

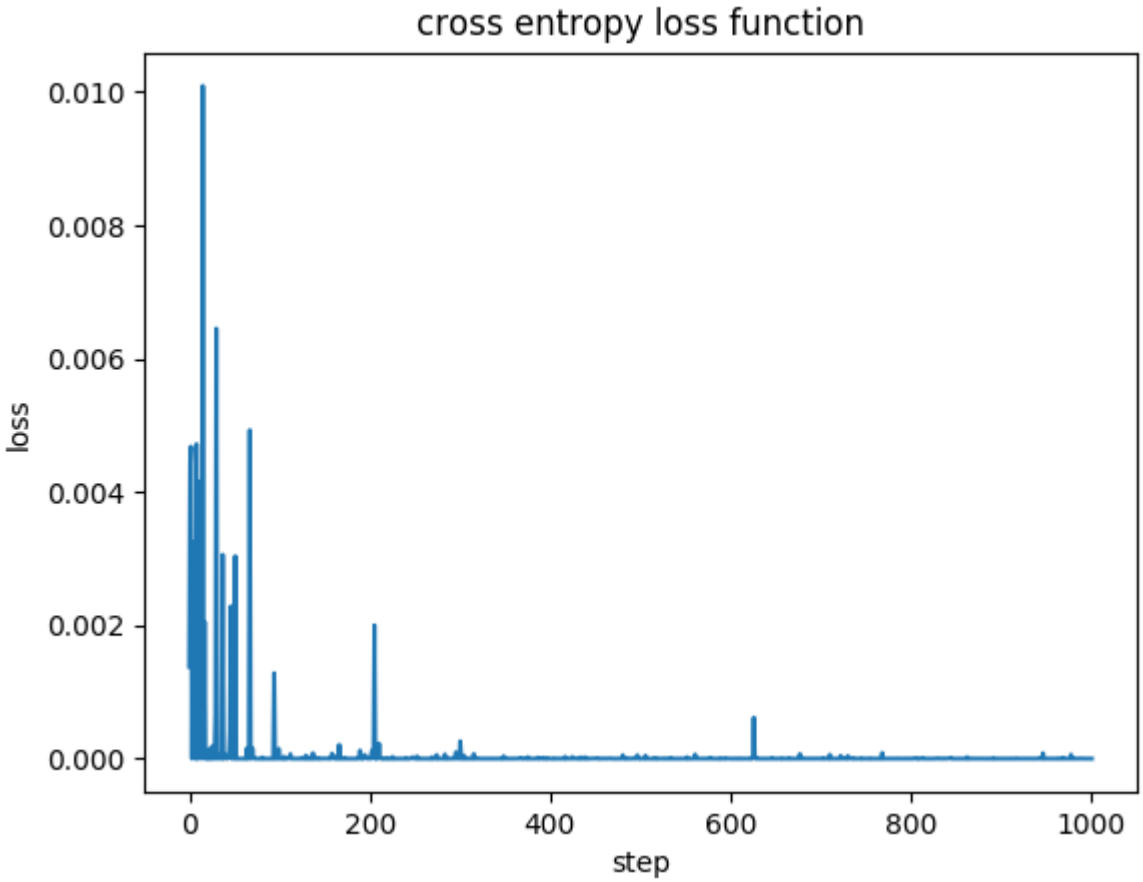
```
$ python source.py -m logistic -s 1000 -g 2 -n 50
```




```
step:  997  loss:  1.64396140795705e-10
step:  998  loss:  1.0547546870376503e-07
step:  999  loss:  2.1255096960013034e-06
Generate doc_vec, Shape:  (1496, 5635)
correct:  1177 / 1496 = 0.7867647058823529
```

3. 随机梯度下降

```
$ python source.py -m logistic -s 1000 -g 2 -n 1
```




```
step:  997  loss:  6.487029977214675e-08
step:  998  loss:  4.3974120690538234e-18
step:  999  loss:  3.588187629998967e-08
Generate doc_vec, Shape:  (1496, 5635)
correct:  1209 / 1496 = 0.8081550802139037
```