Assignment 2

本次实验实现了三种方法的线性分类，分别用 least square model 和 perceptron 来对数据点进行两类分类，还实现了 logistic 方法对文本进行四类分类。

0. 综述

0.1 结构

- source python file: 调用、运行其他文件中的 class
  - `source.py`
- solution files: **每个文件均独立实现了一个部分，并各自封装成 class**
  - `least_square_model.py`: 实现了 least square model 的两类数据点分类
  - `perceptron.py`: 实现了 perceptron algorithm 的两类数据点分类
  - `logistic.py`: 实现了 logistic regression 的四类文本分类

0.2 库

- 除了其他库以外，还使用了 `argparse` 来运行项目，需要通过以下命令行安装：

```
pip install argparse
```

0.3 运行方法

- 在下文的每个关键部分，都会有获得对应结果的命令行，在项目的目录直接运行即可
- 打印 help message：

```
python source.py -h
```

1. Part I

1.1 least square model

代码结构

- 代码都在 `class LSM` 中
- `self.train()`训练出$\mathrm{w} = [w_1 \quad w_2]$和 b 的值；
- `self.predict()`进行预测；
- `self.plot()`画图；
- `self.accuracy()`计算正确率；
- `self.run()`运行整个流程

运行

```
python source.py --algorithm least_square
```

- 输出训练结果$\mathrm{w} = [w_1 \quad w_2]$和 b 的值，以及正确率

实现

- 由于两类数据的 label 是 True(1)和 False(0)，因此两类数据的分界线的 y 值应为 0.5，所以拟合曲线为$\mathrm{w}_1 x_1+\mathrm{w}_2 x_2+\mathrm{b}=0.5$。其中$\mathrm{w}=[w_1 \quad w_2]$和 b 是需要求解的值
- 根据以下课本公式可以迅速计算出拟合向量w。其中，$\Phi^{\dagger}$是 X 的伪逆，可以通过 numpy 的 linalg.pinv()函数快速求得。而t_k就是 y。

$$\mathrm{w}=\Phi^{\dagger} t_k \tag{3.35}$$

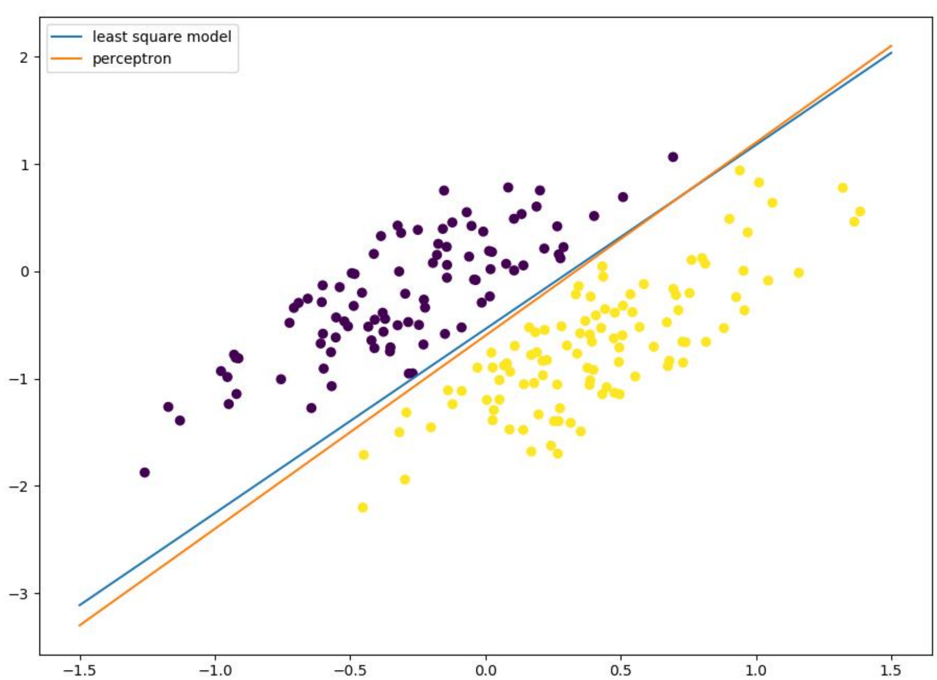
- 为了能更快地求解 w 和 b，令$\hat{X}=[X_{N\times 2}| \quad 1_{N\times 1}]$，也就是在 X 矩阵右边添加一列 1，再获得对应的伪逆$\widehat{\Phi^{\dagger}}$，再根据 3.35 计算出$\widehat{\mathrm{w}}$，而且$\widehat{\mathrm{w}}_{1\times 3}=[w_{1\times 2}| \quad b]$
- 进行预测的时候，对于数据点$\mathrm{X}_{N\times 2}$，可以计算出：

$$y_{N\times 1}=[X_{N\times 2}| \quad 1_{N\times 1}]\times \widehat{\mathrm{w}}_{1\times 3}{}^{T}=X_{N\times 2}\times w_{1\times 2}{}^{T}+b。$$

- 上述两个公式可以使用任意一个。当$\mathrm{y}\geq 0.5$时，判为 True；当$\mathrm{y}<0.5$时，判为 False。

结果

- 分界线方程为
$0.8272788927030406x_1-0.4820461226254367x_2+0.24066371802669287=0.5$
- 正确率：1.0
- 含有分界线的图如下：



1.2 perceptron

代码结构

- 代码都在 `class Perceptron` 中
- `self.preprocess_y()`预处理 y

- `self.train()`训练出$\mathrm{w} = [w_1 \quad w_2]$和 b 的值；
- `self.predict()`进行预测；
- `self.plot()`画图；
- `self.accuracy()`计算正确率；
- `self.run()`运行整个流程

运行

```
python source.py --algorithm perceptron
```

- 输出训练结果$\mathrm{w} = [w_1 \quad w_2]$和 b 的值，以及正确率

实现

- 对 y 进行预处理，把 False 改为-1；另外 True 原本就对应 1。因此两类数据的分界线的 y 值应为 0，所以拟合曲线为$\mathrm{w}_1 x_1 + \mathrm{w}_2 x_2 + \mathrm{b} = 0$。其中$\mathrm{w} = [w_1 \quad w_2]$和 b 是需要求解的值
- 为了计算方便，同样令$\hat{X} = [X_{N\times2} | \quad 1_{N\times1}]$，也就是在 X 矩阵右边添加一列 1，再计算出$\widehat{\mathrm{w}}$，那么可以根据$\widehat{\mathrm{w}}_{1\times3} = [w_{1\times2} | \quad b]$获得 w 和 b
- 初始值$\widehat{\mathrm{w}}_{1\times3} = 1_{1\times3}$
- 当$\widehat{\mathrm{w}}_{1\times3} \times \hat{X}^T \times y < 0$时，按照下方公式更新 w，直至$\widehat{\mathrm{w}}_{1\times3}$不再有变化，而$\widehat{\mathrm{w}}_{1\times3} = [w_{1\times2} | \quad b]$，那么可以获得 w 和 b
$$\widehat{\mathrm{w}}_{1\times3} = \widehat{\mathrm{w}}_{1\times3} + \eta \times \hat{X} \times y$$
- 当进行预测的时候，对于数据点$\mathrm{X}_{N\times2}$，可以计算出：
$$y_{N\times1} = [X_{N\times2} | \quad 1_{N\times1}] \times {\widehat{\mathrm{w}}_{1\times3}}^T = X_{N\times2} \times {w_{1\times2}}^T + b。$$
- 上述两个公式可以使用任意一个。当$\mathrm{y} \geq 0$时，判为 True；当$\mathrm{y} < 0$时，判为 False。

结果

- 分界线方程为
$$0.90091832x_1 - 0.5003462x_2 - 0.2999999999999999 = 0$$
- 正确率：1.0
- 含有分界线的图在上文，和 least square model 的分界线在同一个图中

2. Part II

2.1 代码结构

- 代码都在 `class Logistic` 中
- `self.get_vocabulary()`获得词典
- `self.preprocess_X()`预处理 X
- `self.preprocess_y()`预处理 y
- `self.softmax()`计算 softmax 的值
- `self.loss()`计算 loss
- `self.graident()`按照公式计算 gradient
- `self.check_graident()`检验 gradient 公式的正确性
- `self.shuffle_dataset()`打乱数据集的顺序
- `self.accuracy()`计算正确率；

- `self.train()`训练模型，并获得 loss、train_accuracy、validate_accuracy 的 list
- `self.show()`训练出模型，并画出 loss、train_accuracy、validate_accuracy 的变化
- `self.show_batch_diff()`训练出不同 `batch size` 的模型，并画出 loss、train_accuracy、validate_accuracy 的变化
- `self.show_lamb_diff()`训练出不同 `lamb` 的模型，并画出 loss、train_accuracy、validate_accuracy 的变化
- `self.show_alpha_diff()`训练出不同 `alpha` 的模型，并画出 loss、train_accuracy、validate_accuracy 的变化

2.2 处理 data

- 遍历 `data` 中的 `text`
  - 通过 `text.translate(str.maketrans("", "", string.punctuation))`把 `punctuation` 去掉
  - 通过 `re.sub('[' + string.whitespace + ']', ' ', text)`把 `whitespace` 去掉
  - 通过 `text.lower().split(' ')`把 `text` 小写化，并以空格分割成一个个词
  - 储存每个 `text` 对应的 `words` 列表
- 新建大小为(data 量*字典量)的全 0 数据，每行代表一篇文章对应字典中的词的存在情况
  - 0 表示字典中这个词在这篇文章中不存在
  - 1 表示字典中这个词在这篇文章中存在
- 遍历每篇文章的词列表
  - 如果这个词在词典中，则对应位置标 1

2.3 处理 target

```
dataset = np.zeros([len(y_data), self.categories])
dataset[np.arange(len(y_data)), y_data] = 1
```

- 新建(target 量*类型量)的全 0 数组
- 把每行的 target 值对应的位置标为 1

2.4 梯度公式推导

- Loss 的标量公式如下：

$$\mathrm{L} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk}\log\hat{y}_{nk} + \lambda\sum_{m=1}^{M}\sum_{k=1}^{K} {W_{nk}}^2$$

- 其中：

$$\hat{y}_{nk} = \frac{\exp(Z_{nk})}{\sum_{\hat{k}=1}^{K}\exp(Z_{n\hat{k}})}$$

- 通过链式求导来求解 loss 对 w 的偏导：

$$\frac{\partial \mathrm{L}}{\partial W_{ij}} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk}\frac{1}{\hat{y}_{nk}}\frac{\partial \hat{y}_{nk}}{\partial z_{nj}}\frac{\partial z_{nj}}{\partial W_{ij}} + 2\lambda W_{ij}$$

- 其中：

$$z_{nj} = \sum_{m=1}^{M} \mathrm{X}_{nm} W_{mj} + b_j$$

- 那么在求偏导前首先要求出：

$$\frac{\partial \hat{y}_{nk}}{\partial z_{nj}} = \frac{\frac{\partial \exp(Z_{nk})}{\partial z_{nj}} \sum_{\hat{k}=1}^{K} \exp(Z_{n\hat{k}}) - \exp(Z_{nk})\exp(Z_{nj})}{(\sum_{\hat{k}=1}^{K} \exp(Z_{n\hat{k}}))^2}$$
$$= \frac{\exp(Z_{nk}) I_{jk} \sum_{\hat{k}=1}^{K} \exp(Z_{n\hat{k}}) - \exp(Z_{nk})\exp(Z_{nj})}{(\sum_{\hat{k}=1}^{K} \exp(Z_{n\hat{k}}))^2}$$
$$= \frac{\exp(Z_{nk})}{\sum_{\hat{k}=1}^{K} \exp(Z_{n\hat{k}})} \left(\frac{I_{jk} \sum_{\hat{k}=1}^{K} \exp(Z_{n\hat{k}})}{\sum_{\hat{k}=1}^{K} \exp(Z_{n\hat{k}})} - \frac{\exp\left(Z_{nj}\right)}{\sum_{\hat{k}=1}^{K} \exp(Z_{n\hat{k}})} \right) = \hat{y}_{nk}\left(I_{jk} - \hat{y}_{nj}\right)$$

$$\frac{\partial z_{nj}}{\partial W_{ij}} = \mathrm{X}_{ni}$$

- 另外还有：

$$\sum_{k=1}^{K} y_{nk} = 1$$

- 现在可以代入求解 loss 对 w 的偏导：

$$\frac{\partial \mathrm{L}}{\partial W_{ij}}$$
$$= -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} \frac{1}{\hat{y}_{nk}} \frac{\partial \hat{y}_{nk}}{\partial z_{nj}} \frac{\partial z_{nj}}{\partial W_{ij}} + 2\lambda W_{ij} =$$
$$-\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} \frac{1}{\hat{y}_{nk}} \hat{y}_{nk}\left(I_{jk} - \hat{y}_{nj}\right)\mathrm{X}_{ni} + 2\lambda W_{ij} =$$
$$-\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk}\left(I_{jk} - \hat{y}_{nj}\right)\mathrm{X}_{ni} + 2\lambda W_{ij}$$
$$= -\frac{1}{N} \sum_{n=1}^{N} \mathrm{X}_{ni}(y_{nj} - \sum_{k=1}^{K} y_{nk}\hat{y}_{nj}) + 2\lambda W_{ij} = -\frac{1}{N} \sum_{n=1}^{N} \mathrm{X}_{ni}\left(y_{nj} - \hat{y}_{nj}\right) + 2\lambda W_{ij}$$

- 通过链式求导来求解 loss 对 b 的偏导，首先要求出

$$\frac{\partial z_{nj}}{\partial b_j} = 1$$

- 带入求解 loss 对 b 的偏导：

$$\frac{\partial \mathrm{L}}{\partial b_j} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk}\frac{1}{\hat{y}_{nk}}\frac{\partial \hat{y}_{nk}}{\partial z_{nj}}\frac{\partial z_{nj}}{\partial b_j} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk}\frac{1}{\hat{y}_{nk}}\hat{y}_{nk}\left(I_{jk} - \hat{y}_{nj}\right)$$

$$= -\frac{1}{N}\sum_{n=1}^{N}\left(y_{nj} - \hat{y}_{nj}\right)$$

- 各用一行代码实现 numpy 的向量化运算

```
w_gradient = 2 * self.lamb * w - X_train.T @ (y_train - y_pred) / X_train.shape[0]
b_gradient = - np.sum(y_train - y_pred, axis=0) / X_train.shape[0]
```

2.5 L2 正则化

- 根据上文的 loss 公式，可以看出使用了 L2 正则化，正则化只包含了 w，而没有 b。另外，正则化的系数为λ
- 公式中正则化只包含了 w，而没有 b，因为：当模型过于复杂的时候，会出现 over-fitting 的情况，在训练集上达到很高的正确率，但是泛化能力却降低了。因此需要正则化，给模型的复杂度加上 penalty，当模型越复杂时，penalty 越大，这样可以降低 over-fitting 的程度。而对于线性拟合，决定模型复杂度的是 w。当 w 的 L2 很大的时候，预测结果的变化波动很大，使得复杂度变高，而 b 的大小只会使数据整体偏移，而不会对结果的波动变化有影响，也不会使模型复杂度有影响。因此不需要正则化 b。

2.6 检验梯度计算的正确性

运行

```
python source.py --algorithm logistic --n check
```

- 检验梯度计算的正确性

原理

- 按照导数的定义计算梯度：

$$\nabla \mathrm{W} = \left[\frac{\partial \mathrm{L}}{\partial W_{ij}}\right]_{N\times K} = \left[\frac{L\left(W+\delta_{ij}\right) - L(W)}{\left|\delta_{ij}\right|}\right]_{N\times K}$$

$$\nabla b = \left[\frac{\partial \mathrm{L}}{\partial b_j}\right]_{K} = \left[\frac{L\left(b+\delta_{j}\right) - L(b)}{\left|\delta_{j}\right|}\right]_{K}$$

- 如果导数定义结果和公式计算结果的任意对应值的差 error，小于某个给定的小阈值ε，则可以认为梯度计算正确

$$\forall \mathrm{i,j} \quad \mathrm{st.} \left|\nabla \mathrm{W}_{ij} - \nabla \widehat{\mathrm{W}}_{ij}\right| < \varepsilon \quad and \quad \left|\nabla \mathrm{b}_k - \nabla \hat{\mathrm{b}}_k\right| < \varepsilon$$

实现

- 由于计算量很大，因此首先 shuffle 一下数据集，取出前 8 条数据，并循环 7 个 epoch，并计算梯度用于验证
- 在每个 epoch 中，用公式计算出 w、b 的梯度，并通过导数定义计算出 w、b 的梯度，把每个对应值的差记录下来
- 如果最大的差值小于给定的小阈值ε，则说明梯度公式是正确的

结果

- 在导数定义公式中的$|\delta_{ij}| = 1e - 6$的情况下，最大差值可以小于阈值$\varepsilon = 1\mathrm{e} - 7$，说明梯度公式是正确的

思考：为什么不用数值计算的方式求梯度？

- 原因：通过公式计算快很多
- 公式计算快：可以通过 numpy 实现向量化运算
- 导数定义计算慢：如果要向量化运算，则每次运算需要$(\mathrm{N} \times \mathrm{K}) \times (\mathrm{N} \times \mathrm{K})$的空间（N 为数据量，K 为类型数），空间代价是巨大的，计算耗时也比较长；如果不用向量化运算，需要用两层 for 循环，每次求解梯度时需要$\mathrm{N} \times \mathrm{K} + \mathrm{K}$次导数运算，计算耗时长
- 因此使用公式计算梯度

2.7 logistic regression

运行

```
python source.py --algorithm logistic --n run
```

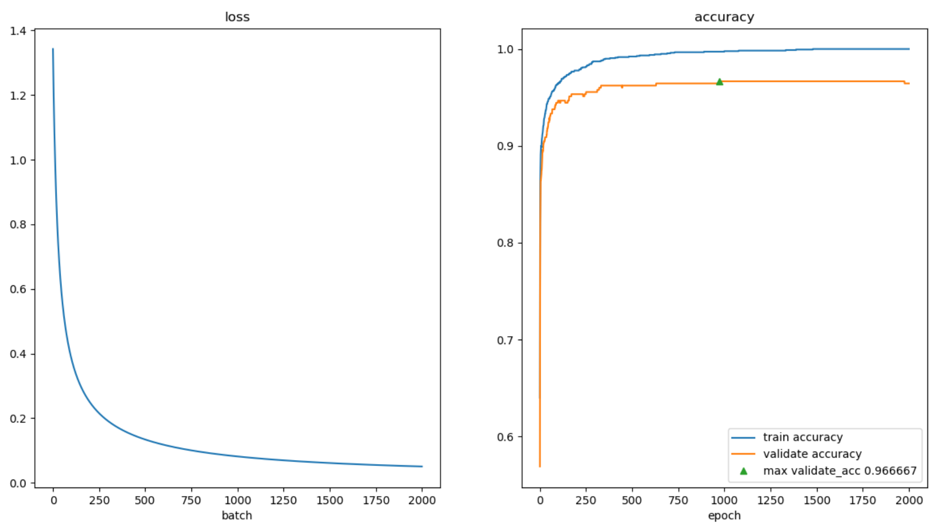
- 输出每个 epoch 的在训练集上的正确率和检验集上的正确率
- 最后用达到最高检验集正确率的 w、b，在测试集上计算正确率，并输出

实现

- **把训练集的 4/5 作为训练集，剩下的作为检验集，在后面的模型训练中也是如此划分**
- 正则化系数为 1e-4，学习率为 0.1，运行的 epoch 数为 2000，是 full batch 的
- 初始化 w 和 b 为 0
- 循环 2000 个 epoch，在每个 epoch 中先 shuffle 训练集，再计算梯度，更新 w、b，记录 loss、训练集上的正确率、检验集上的正确率
- 最后用达到最高检验集正确率的 w、b，在测试集上计算正确率，并输出
- 画出 loss、训练集上的正确率、检验集上的正确率的变化曲线

结果

- 训练集上的正确率达到了 1，检验集最高正确率为 0.966667
- 达到最高检验集正确率的 w、b 在测试集上的正确率为 0.925134



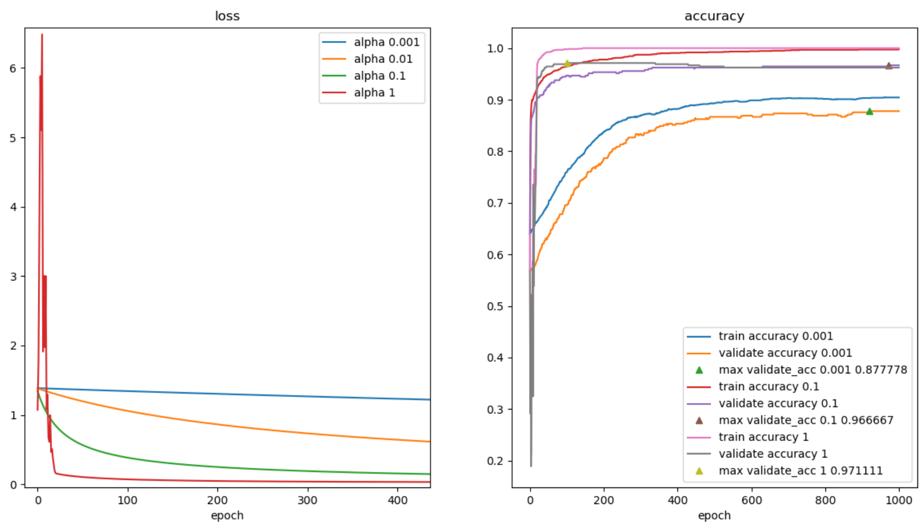
2.8 如何决定学习率

运行

```
python source.py --algorithm logistic --n alpha
```

- 训练 4 个不同学习率的模型，学习率为 0.001、0.01、0.1、1
- 输出每个 epoch 的在训练集上的正确率和检验集上的正确率
- 最后用达到最高检验集正确率的 w、b，在测试集上计算正确率，并输出
- 画出 loss、训练集上的正确率、检验集上的正确率的变化曲线

结果



分析

- 通过上述 loss 值变化，可以发现，学习率越大，loss 下降的越快，但是 loss 的变化也更不稳定；而训练集和检验集的正确率也上升的更快
- 如果学习率过大，则可能无法收敛

决定学习率的策略

- 学习率要尽量大，使得收敛更快；但是学习率不能过大，而使得不能收敛
- 可以先在一个较小的数据集上运行多次，获得一个大致的学习率范围，再在这个范围中，用全部数据集上运行多次，获得更准确的学习率

2.9 什么时候结束训练

停止条件

- 前一周期所有的ΔW_{ij}都太小，小于某个指定的阈值ε
- 前一周期错误率小于某个指定的阈值ε
- 超过某个指定的 epoch 数N_{epoch}

指定 epoch 数

- 我使用的方法是指定 epoch 数，取为 1000
- 如何指定 epoch 数？epoch 数越小，训练所需时间越短；但是 epoch 数要足够大，能让 validation 集上的正确率达到最大值，并且能显示出由于 overfitting 而下降的趋势，这样才能确认获得最好的 validation 集正确率

2.9 不同的 batch size

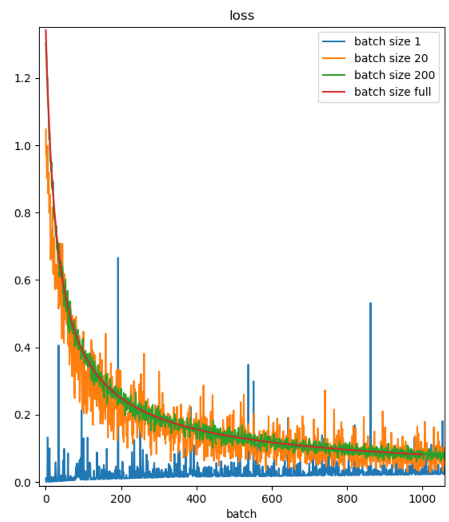
运行

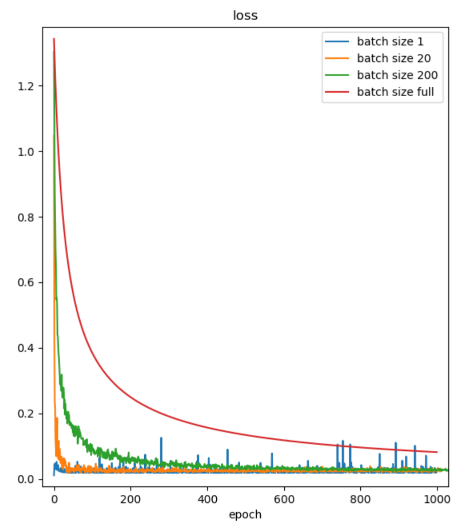
```
python source.py --algorithm logistic --n batch
```

- 训练 4 个不同 batch size 的模型，batch size 为 1、20、200、full batch
- 在训练模型的过程中输出对应的正确率
- 最后画图

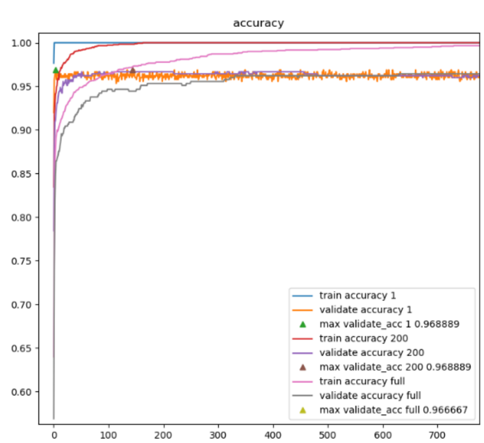
结果

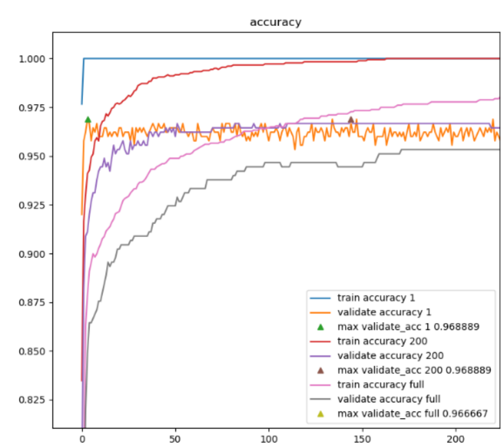
- 左图为统一运行的 batch 数量的结果。右图为统一运行的 epoch 数量的结果





- 左图为不同 batch size 在训练集、检验集上的正确率变化；右图为左图的放大。





分析

- batch size 越小，loss 的下降方向越不准确，loss 的波动越大
- 除了 batch size 为 1 的情况外，loss 的下降大致趋势是一样的

- batch size 越小，loss 下降的越快，收敛得更快

优缺点

- batch size 较大：loss 下降的方向更准确，波动小，但 loss 下降的慢，所用的数据量大
- batch size 较小：loss 下降的波动大，而 loss 下降更快，所用的数据量少
- 而 mini batch 则是两种情况的折中，loss 下降准确，波动小，且 loss 下降的快，收敛更快

2.10 不同的 lambda

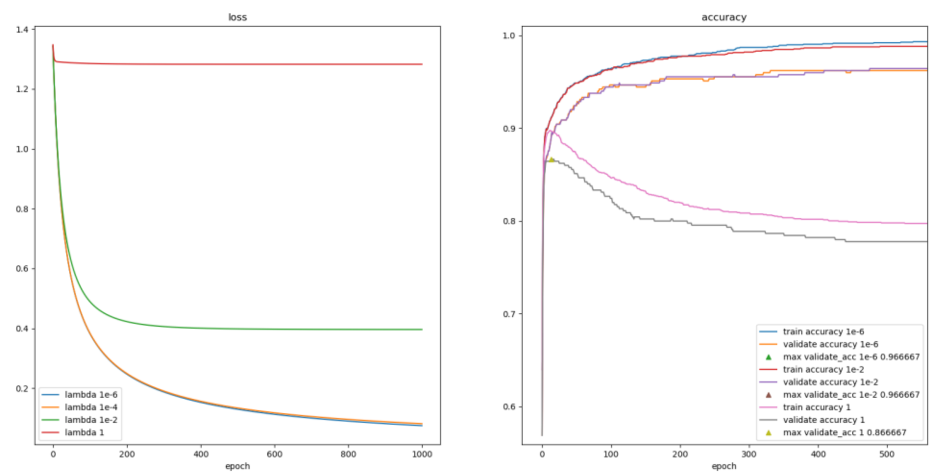
运行

```
python source.py --algorithm logistic --n lambda
```

- 训练 4 个不同 lambda 的模型，为 1e-6、1e-4、1e-2、1
- 在训练模型的过程中输出对应的正确率
- 最后画图

结果

- 左图为不同 lambda 的 loss 变化。右图训练集、检验集上正确率的变化



分析

- lambda 越大，penalty 越大，loss 下降慢、下降量越少，准确率越快达到最大值，值越低

- lambda 越小，panalty 越小，loss 下降快，下降量多，准确率更高

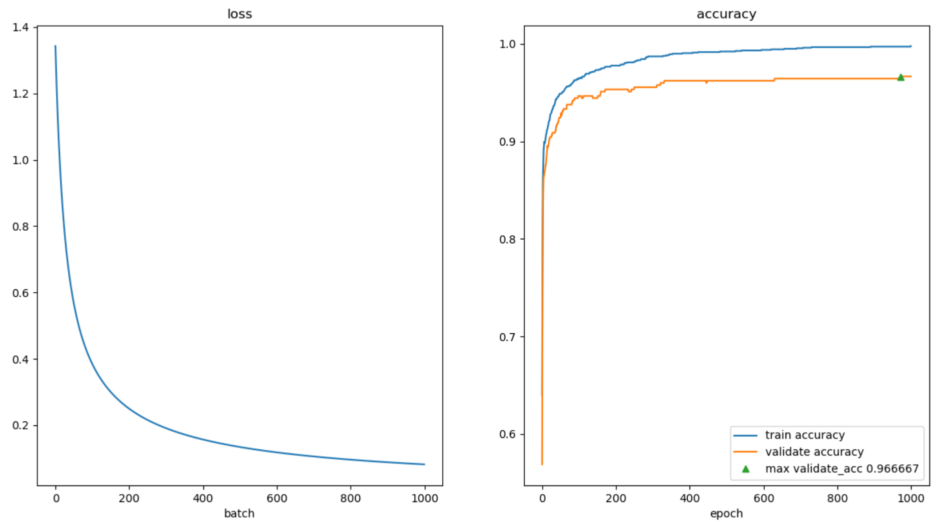
- 但是 lambda 越小，越容易 over-fitting

2.10 不同的 batch size 的测试集结果

- 以下左图均为 loss 变化，右图均为训练集、检验集上的正确率变化

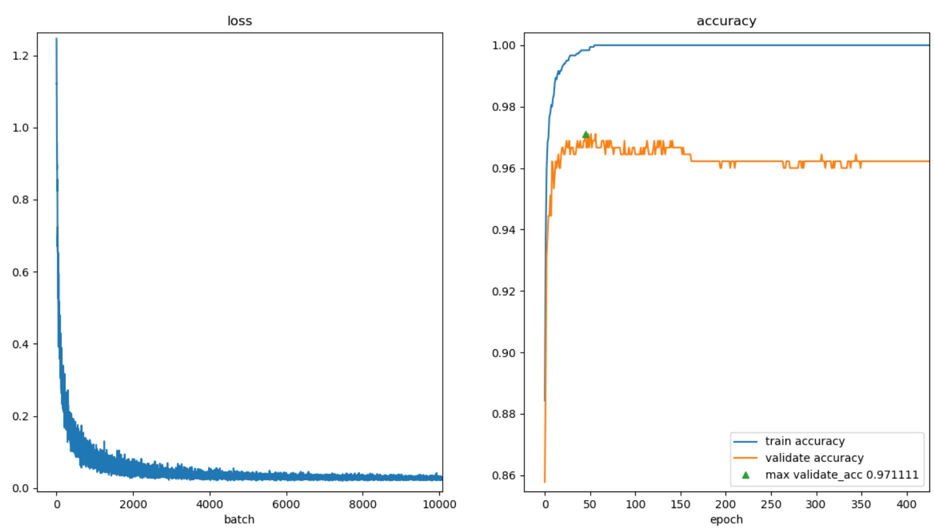
full batch

- 测试集正确率：0.925134



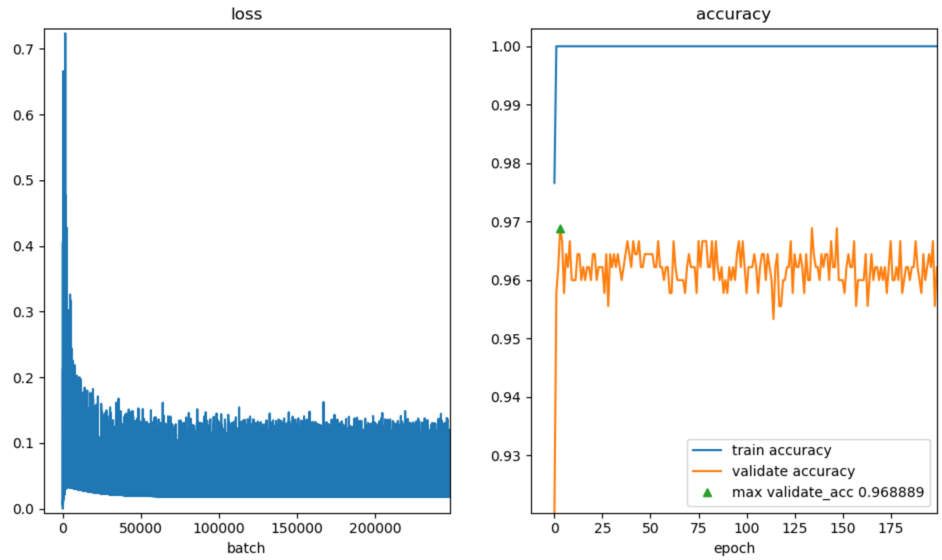
mini batch

- 测试集正确率：0.924465



full batch

- 测试集正确率：0.922460


loss
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
50000
100000
150000
200000
batch
accuracy
1.00
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0
25
50
75
100
125
150
175
epoch
train accuracy
validate accuracy
max validate_acc 0.968889