幽默文本检测

18308133 刘显彬 19335301 庄鹏标

2021 年 11 月 17 日

1 概述

在本次实验，我们小组完成了简单的文本清理工作，使用了朴素贝叶斯和前馈神经网络模型完成了幽默文本分类和幽默文本评分的任务。在实验过程中，遇到了许多问题包括预测准确率不理想，模型过拟合，计算时间较长等问题。我们采取了不同的学习率调整策略，使用优化器，调整模型参数等多种方法解决问题，最终获得了理想的结果。

2 实验原理

2.1 文本数据预处理

文本预处理必不可少。文本预处理对于过滤噪声，降低模型复杂度，提高预测准确率，节约计算成本都有极大帮助。文本预处理这一步骤需要与下游任务相结合，综合考虑后续需要的操作来决定预处理的工作。如在本次幽默文本检测的数据竞赛中，就可以考虑幽默这个主题。如，我们觉得有些单词与幽默的主题没有什么关系，就可以考虑去掉。同时，这也可以对我们训练好的词向量进行降维，降低模型复杂度，避免出现过拟合。文本预处理的方法如下 [3]：

- **脏数据清洗**。首先我们要检测是否文本具有我们需要的属性，如果我们发现某些文本的标签或是其他属性缺失，我们考虑要把这部分文本分离出来或是补充数据
- **去除数据中的非文本部分**。在本次使用的英文文本中有许多特殊的符号，包括中文，标点符号等我们不需要的文本内容，因此我们首先要去除这部分内容。我们可以直接使用 python 的正则表达式处理，只保留英文和数字。
- **分词**。由于英文单词间由空格分隔，所以分词比较简单，只需要根据空格分词，调用 split() 函数即可完成。
- **去除停用词**。在人类的语言中，有许多功能词，这部分词出现比较普遍，但对于辨别文本与文本的区别没什么帮助，这就是停用词。在英语文本中，”a”，”the” 等冠词，“over”, “under”, “above”等介词，还有一些虚词、代词或者没有特定含义的动词、名词就是停用词。我们往往希望能去掉这些停用词。我们可以通过总结出来的停用词表，来去除停用词。
- **词干化**。英文中同一个词可能因为单复数，时态等原因有不同的形式，但都具有相同的词干。如 leaf 和 leaves，词干化得到 leav。我们将词还原成词干的形式，突出文本的特征。

- **特征处理**。经过前面的步骤，我们已经获得了足够干净的文本，接下来就是要将文本转化成数字的表示形式。我们使用的是 Word2Vec 模型获得词向量。Word2Vec 使用一系列的文档的词语去训练模型，把文章的词映射到一个固定长度的连续向量。得到词向量后，将每个句子中的词向量相加取平均值，得到每个句子的平均词向量用以表示句子的向量。

2.2 基于朴素贝叶斯的文本分类算法

2.2.1 原理

1. 在分类问题中，我们要把一个文本 X 划分为某个类别，类别有很多种 Y=y1,y2,y3……ym。我们无法直接确定 X 的标签，只能根据 X 是标签 yi 的概率大小来判断。因此我们的目标就是计算 P（yi|X）

2. 根据条件概率的公式 $P(y_i|X) = \frac{P(X|y_i)*P((y_i))}{P(X)}$。对于不同的 $y_i, P(X)$ 相同，因此我们只需要比较 $P(X|y_i)*P((y_i))$ 即可。其中 $P(y_i)$ 是类别的概率我们可以用训练集中，标签为 0 和 1 的文本数量来估计，即 $P(y_i) = \frac{n_i}{n_0+n_1}$, 其中，n0 表示标签为 0 的文本数量，n1 表示标签为 1 的文本数量。

3. 根据条件独立性，$P(X|y_i) = \Pi_j^n P(X_j|y_i)$, 即我们可以认为各个单词之间是没有联系。

4. $P(X_j|y_i)$ 通过统计训练集中，不同词语在该类别的出现频率来估计。对每个标签类别，我们都可以获得一个这样的词表，用于估计单词出现的概率

2.2.2 举例说明

使用一个例子来说明如上的原理，对句子“welcome to beijing”判断它是标签 0，还是标签 1。

1. 我们计算并比较 $P(0|welcome\ to\ beijing)$ 和 $P(1|welcome\ to\ beijing)$。

2. 根据条件概率的公式,我们问题转变为比较 $P(welcome\ to\ beijing|0)*P(0)$ 和 $P(welcome\ to\ beijing|0)*P(1)$

3. 计算 $P(0) = \frac{n_0}{n_0+n_1}$, $P(1) = \frac{n_1}{n_0+n_1}$，其中 n0 表示标签为 0 的文本数量，n1 表示标签为 1 的文本数量

4. 用条件独立性计算 $P(welcome\ to\ beijing|0) = P(welcome|0)*P(to|0)*P(beijing|0)$, 其中 n(welcome) 表示在类别 0 的数据集中,$P(welcome|0) = \frac{n(welcome)}{N}$，welcome 这个单词出现的次数；N 表示类别 0 的数据集中，单词的总数。对其他词的计算同理。

5. 用同样的方法计算 $P(welcome\ to\ beijing|1)$

6. 比较 $P(welcome\ to\ beijing|0)$ 和 $P(welcome\ to\ beijing|1)$ 概率大的就是句子的标签

2.3 Feedback Network 和 Back Propagation

2.3.1 前向神经网络

我们知道，将输入值的集合唯一映射为输出集合的关系，叫做函数：$X \xrightarrow{f} Y$，其中 X 可以是一段文字，Y 可以是这段文字对应的幽默程度估计，这种估计关系就是一种函数。但这种关系非常难用准确的数学函数式子表示，所以我们希望：

- $f(x)$ 本身是一个黑箱子，它可以自行调整自己的内部结构
- 它可以处理不限于线性的函数映射，对于非线性的映射它也可以有一定的处理能力

对于上面的问题，我们可以分别用下面的方法解决：

- 自行调整：我们可以借鉴逻辑回归的经验，借助梯度下降这一方法来实现
- 线性预测：我们可以简单地用 $Y = Wx + b$ 来很表达线性的函数结构，其中的 W, b 就是需要用梯度下降让模型自行学习的参数
- 非线性：引入非线性的函数，与上一步中的 Y 串联输出，自此，我们就完成了从 $X \to Y$ 的简单非线性映射
- 双层叠加：单次的线性预测 + 非线性输出并不足以表现非线性（因为最终结果和中间的线性预测结果仍然是唯一相关的），因此，我们采用一种简单的方式：将该结果，重复上面两个过程，这样就可以将第一次的非线性信息利用起来，让我们的黑箱子打破非线性的结构，正式拥有非线性的预测能力

形式化来讲：我们的黑箱子中的线性预测单元称为：神经元；神经元输出通过的非线性函数称为激活函数（这一过程称为激活）；我们把通过一次线性预测 + 一次激活称为一层中间层/隐藏层。

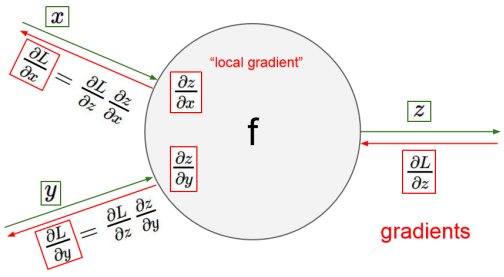
但不难想象的是，随着从 $X \to Y$ 的实际映射复杂度上升，有限的双层结构不能充分表达实际映射，此时我们有两种做法以增加复杂度：（1）将层内神经元的规模变大：但神经元的运算是线性的，这样并不足以弥补非线性部分。（2）增加层数：模仿双层叠加，将更多层串联在一起，这样便能不断增加黑箱子的非线性能力。

至此，我们得以初步窥视神经网络的基础结构：以神经元预测和激活过程为重复单元：给出图示如下：

2.3.2 后向传播过程

现在我们来关注如何对神经网络的进行梯度推导，我们用到的核心公式为链式法则：

$$\frac{\partial Z}{\partial X} = \frac{\partial Y}{\partial X}\frac{\partial Z}{\partial Y} \tag{1}$$



从局部看一个单独的神经元的话，也就是说，我们计算一个神经元处的梯度，也就是要算，损失函数 L 对该神经元参数 W 的偏微分，假设该神经元的输入为 X，输出为 Y，由链式法则：

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial W} \tag{2}$$

如此一来，我们便将一个全局的梯度计算，分解成了两部分：神经元的局部梯度：$\frac{\partial Y}{\partial W}$，以及 L 对该神经元输出 Y 的全局梯度：$\frac{\partial L}{\partial Y}$，但幸运的是，由 $Y = Wx + b$ 可知，$\frac{\partial Y}{\partial W} = X$，所以问题变成了 Y 的全局梯度求解，我们重复这个过程，不难看出：每一层 (i 层) 的梯度计算，都依赖于该层的输出 (Y_i) 的梯度计算。因此，我们可以从最后一层的梯度开始，然后不断往前面的层计算梯度，由于计算梯度的方向和神经网络的输出计算方向相反，因此，我们把这个过程叫做：**后向传播过程**；同时我们发现，全局梯度计算可以分解成局部梯度的乘积，所以我们可以先对单独的激活层、线性层、损失层等，先计算它们的局部梯度作为基础。

2.4 梯度推导

2.4.1 神经元

$$Y = Wx + b \tag{3}$$

$$\frac{\partial Y}{\partial X} = W \tag{4}$$

$$\frac{\partial Y}{\partial W} = X \tag{5}$$

2.4.2 激活层

$$Y = \frac{1}{e^{-X}} \tag{6}$$

$$\frac{\partial Y}{\partial X} = \frac{e^{-X}}{Y^2} = Y(Y-1) \tag{7}$$

2.4.3 损失函数

损失函数我们用了三种：二元交叉熵（BCE），均方误差（MSE），均方根误差（RMSE）：

其中 BCE 我们在逻辑回归中已经完成推导，在此不再赘述。MSE 推导如下，假设样本数为 N，真实标签为 $Label$，预测值为 Y：

$$L = \frac{1}{N}\sum_{i}^{N}(Label_i - Y_i)^2 \tag{8}$$

$$\frac{\partial L}{\partial Y_i} = \frac{2 * (Y_i - Label)}{N} \tag{9}$$

RMSE 的推导如下：

$$L_{MSE} = \frac{1}{N}\sum_{i}^{N}(Label_i - Y_i)^2 \tag{10}$$

$$L_{RMSE} = \sqrt{L_{MSEL}} \tag{11}$$

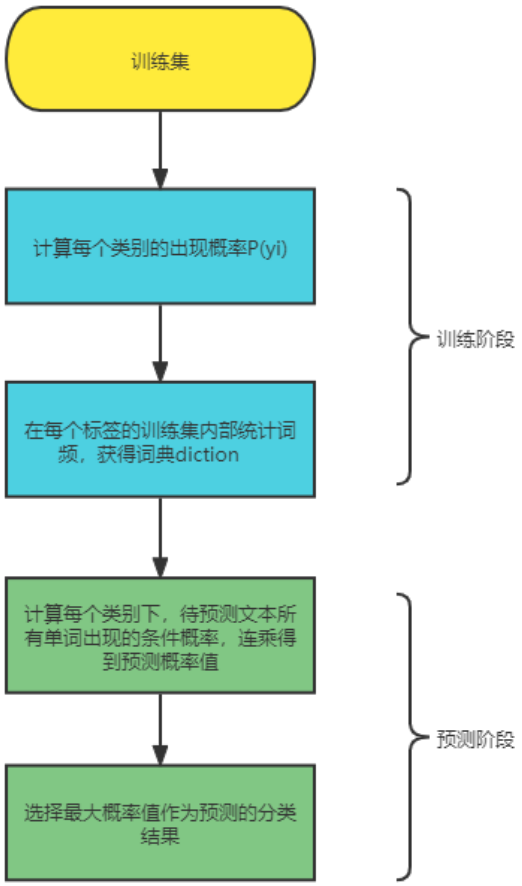
$$\frac{\partial L}{\partial Y_i} = \frac{1}{2}\cdot\frac{1}{L}\cdot\frac{\partial L_{MSE}}{\partial Y_i} \tag{12}$$

$$= \frac{Y_i - Label}{N \cdot L} \tag{13}$$

3 伪代码/实验过程

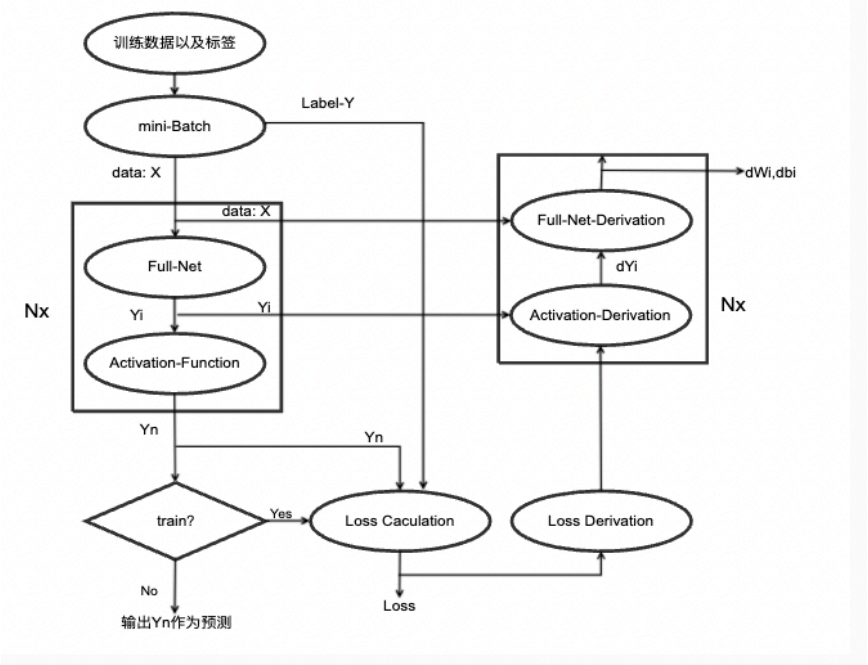
3.1 朴素贝叶斯的文本分类

朴素贝叶斯因为内容较为简单，所以不在此展示代码，我们直接展示流程图如下：其中有训练（统计词频）以及预测（计算后验条件概率）两大阶段。



3.2 神经网络

由于每一部分的偏微分过程的伪代码不比数学公式或者实际的代码更清晰，所以这一部分将不给出推导公式的伪代码。我们直接给出计算过程的流程图（如图所示）：



其中，注意到图中的 Nx，表示这一层结构将会串联重复 N 次，上一层的结果将作为下一层的输入；对称地，在反向传播过程也会重复 N 次，每一层的梯度 dy 将作为前一层的上游梯度；还有图中横跨 Forward 和 BackWard 过程的箭头：这表示计算梯度的时候，我们需要用到前向传播中的数据，正如上一节梯度推导所示。

4 关键代码分析

4.1 朴素贝叶斯

下面是词频统计，展示分类为 0 的部分：

```
for row in trainSentence0:
        n0 += len(row)
        for word in row:
            if word in dict0:
                dict0[word] += 1  # 如果单词已经在词表中，则数量+1
            else:
                dict0[word] = 1  # 如果单词没有在词表中，则加入
```

这里是预测阶段即根据公式 $P(X|y_i) = \Pi_j^n P(X_j|y_i)$，计算条件概率，再乘以类的概率。其中在计算每个单词的出现概率时，为了避免出现次数为 0，而概率为 0，对所有单词出现次数 +1，对分母 +2。代码如下：

```
def predict(dict0, dict1, data, n0, n1):
    result = []
     for row in data:  # 逐个文本进行预测
        p0 = 0
        p1 = 0
```

```
        for word in row:  # 逐个词语计算概率
            if word in dict0:
                p0 = p0 * (dict0[word] + 1) / (n0 + 2)
            else:
                p0 = p0 * 1 / (n0 + 2)

            if word in dict1:
                p1 = p1 * (dict1[word] + 1) / (n1 + 2)
            else:
                p1 = p1 * 1 / (n1 + 2)

        if p0 > p1:
            result.append(0)
        else:
            result.append(1)
    return result
```

4.2 神经网络

我们采用的基础结构如下：

```
class NerualNet:
    def __init__(featureDim=None, outputDim=1, HiddenDims=[]):
        self.W = []   # 每层的参数W
        self.b = []   # 每层的参数b
        self.activation = [] # 每层的激活函数句柄
        self.backwardActivation = [] #每层梯度计算句柄

        # 随机初始化：高斯随机分布，均值0，方差1
        HiddenDims.insert(0, featureDim)
        HiddenDims.append(outputDim)
        for i in range(len(HiddenDims)-1):
            self.W.append((np.random.randn(HiddenDims[i], HiddenDims[i+1])
            self.b.append(np.random.randn(1, HiddenDims[i+1]).astype('float64'))
        self.layers = len(self.W)
```

核心函数：Loss，根据输入，前向传播并计算 Loss，同时完成反向传播的梯度计算，返回梯度：

```
def loss(X, Labels=None, lossfunction='RMSELoss'):
    dW_all, db_all, X_fullnet = [],[],[] # X_Fullnet记录了神经元的输入
    datas = [] # datas保存了激活层的输入和输出
    Y = None

    # forward
    for i in range(self.layers):
        data = {}
        Y,X = self.Fullnet(X,self.W[i],self.b[i])
        X_fullnet.append(X)
```

```
        acfunc = self.activation[i] # 取得这一层的激活函数句柄
        if acfunc != 'linear' and acfunc != None:
            # 最后一层激活函数如果非None（进行回归时最后一层不加激活函数）
            # data保存了激活层的输入和输出，用于反向传播,结构上是一个字典
            data = acfunc(Y) # 函数原型示例： {'Y':Y,'X':X} = sigmod(X)
            datas.append(copy.deepcopy(data))

    if Label is None:
        # 直接返回预测值
        return datas[-1]['Y']

    '''loss 计算 '''
    if lossfunction == 'RMSELoss':
        lossf = self.RMSELoss
        backLoss = self.backwardRMSELoss
    else:
        lossf = self.MLELoss # BCELoss
        backLoss = self.backwardMLELoss

    # datas[-1] struct : {'X':最后激活层输入, 'Y'; 预测值, 'loss':loss}
    datas[-1] = lossf(datas[-1], Y=Labels)


    '''backward'''
    dy = backLoss(datas[-1], Y=Labels) # loss层的微分，dy 将是后面梯度累积的结果
    for i in range(self.layers-1, -1, -1):
        dy = self.backwardActivation[i](datas[-1], dy_upper=dy) # 将loss的梯度作为上游，计算激活层梯度
        dW, db, dy = self.backFullnet(X_fullnet[i], self.W[i], self.b[i], dy_upper=dy)
        dW_all.insert(0, dW)
        db_all.insert(0, db)
    return {"dW":dW_all, "db":db_all, "loss":datas[-1]['loss']}
```

loss 传入一个批次的 X，和其对应的 label，用于计算梯度，train 函数将对总训练集分 mini-Batch，并调用 loss，随后进行梯度更新：

```
def train(self, X, label, lrate=0.01, epochs=10, batchSize=0, lossfunction="RMSELoss", optim=None)

    for epoch in range(epochs):
        for miniX, minY in DataLoader(np.append(X,labels, axis=1)).Batch(batchSize,shuffle=False):
            cache = self.Loss(miniX, minY, lossfunction=lossfunction)
            dW, db, loss = cache["dW"], cache["db"], cache["loss"]

            if optim is not None:
                dW, db = optim(dW, db)
            for i in range(self.layers):
                self.W[i] -= lrate*(dW[i])#+lamda*self.W[i])
                self.b[i] -= lrate*(db[i])#+lamda*self.b[i])
```

预测过程只需要简单调用下 Loss 函数即可，在不传入 Label 时，loss 将终止在前向传播过程，并返回预测值，因此不再展示。

下面给出 (R)MSELoss，ReLU 的实现（sigmod、似然函数损失等均已在逻辑回归实现），将 $\sum(Y-label)^2$ 展开为：$|Y|^2+|label|^2-2Y\cdot label$ 的向量化运算已在第一次实验推导。

```
def ReLU(self, X):
    # groupReLU是ReLU正负半轴的斜率（k1，k2），默认为（1，0），
    # 即正半轴为y=x，负半轴y=0x
    return {'Y':(X*(X>0)*self.groupReLU[0] + X*(X<0)*self.groupReLU[1]).astype('float64'), 'X':X}

def backwardReLU(self, data, dy_upper=0):
    # dy is upper derivate
    # return d(ReLu_y)/d(ReLU_x)
    X = data['X']
    return ((X > 0)*self.groupReLU[0] + (X<0)*self.groupReLU[1])*(dy_upper)

def MSEloss(self, data, Y):
    # Y是标签
    Yp = data['Y'] # 预测值
    loss = (Y.T.dot(Y)+Yp.T.dot(Yp)-2*Y.T.dot(Yp))/len(Y)
    data['loss'] = loss
    return data

def backwardMSE(self, data, Y):
    Yp = data['Y']
    return 2*(Yp-Y)

def RMSELoss(self, data, Y):
    Yp = data['Y']
    loss = np.sqrt((Y.T.dot(Y)+Yp.T.dot(Yp)-2*Y.T.dot(Yp))/len(Y))
    data['loss'] = loss
    return data

def backwardRMSE(self, data, Y):
    Yp = data['Y']
    Loss = data['loss']
    return (Yp-Y)/Loss
```

5 结果

5.1 数据集划分

我们数据集划分的思路与之前的实验类似，为了方便比较。我们直接使用了 K=10 的 K 折划分，打乱后，选择第一折作为验证集。

5.2 幽默文本分类

分类问题中使用了朴素贝叶斯和前馈神经网络两种方法，对结果进行比较。朴素贝叶斯的结果如下：

表 1: 随机选取几次的结果：

times	accuracy
1	0.6532
2	0.6106
3	0.6287
4	0.560

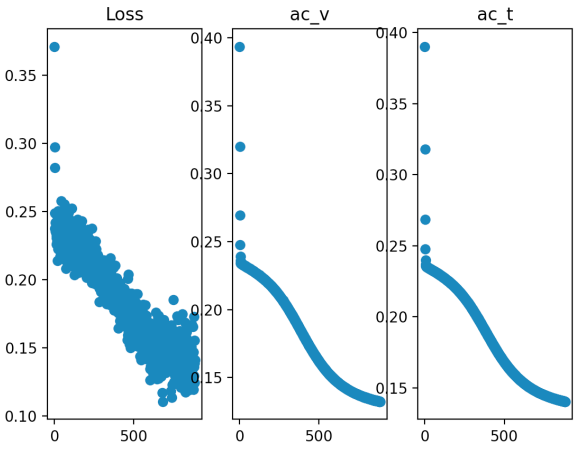
准确率非常低（因此我们后面直接放弃了，我们做实验的时候认为，样本量过少，不足以反映实际概率）

前馈神经网络在分类上使用的损失函数使用的是似然函数。比较准确率和 Loss 大小，考虑学习率，隐藏层维度，激活函数的影响：**准确率**结果如下：

表 2: epochs 设置为 150 时，不同参数下的性能

lrate	hiddenDims	activation-function	val accuracy
0.001	[64, 32]	['sigmod','sigmod', 'sigmod']	0.783
0.01	[64, 32]	['sigmod','sigmod', 'sigmod']	0.81
0.001	[64, 32]	['sigmod','ReLU', 'sigmod']	0.798

可见准确率不算低，但是耗时非常久，如下图的 Loss 曲线可知：几乎耗尽了所有的 epochs（150）才勉强收敛。从左到右，依次为：miniBatch Loss，validation Loss，trainSet Loss，每次**迭代**（不是 epochs）记录一次。



因此从**时间复杂度**而言，朴素贝叶斯分类的计算时间相比前馈神经网络计算时间短的多。前馈神经网络计算时间主要受到隐藏层维度和迭代次数 epochs 的影响。在两层隐藏层 [64, 32] 的条件下，迭代 100 个 epochs 用时大概 80 秒，随着隐藏层增加计算用时会显著提升。

稳定性：朴素贝叶斯分类是基于概率公式计算得到预测结果，因此稳定性较强，在数据集较小的情况下分类结果也比较稳定；前馈神经网络则受到影响的可能因素较多，包括初始化参数，学习率，迭代次数等，但是在目前的神经网络中，尽管较多情况都会陷入到局部的极值点中，但整体表现也都比朴素贝叶斯的表现更好。

5.3 文本评分：回归预测

我们将最后一层的激活函数去除，因此就可以实现回归预测（因为线性映射的结果就是整个实数域），损失函数使用了 MSEloss（实际使用上与 RMSELoss 结果差不多）：

表 3: epochs 设置为 150 时，不同参数下的平均性能

lrate	hiddenDims	activation-function	RMSELoss
0.001	[64, 32]	['sigmod','sigmod', 'None']	0.554
0.01	[64, 32]	['sigmod','sigmod', 'None']	0.543
0.001	[64, 32]	['sigmod','ReLU', 'None']	0.548

表现已经较好，主要问题和分类的问题一致（时间较长，受初始参数影响较大），因为使用的模型（神经网络）、调整方式都一样，有共通性，所以稍微侧重分类任务来分析模型性能。

但我们觉得神经网络的优化空间还有很多，因此不在这里继续调参以找到更好的结果，后面一部分，将体现我们的优化对于模型稳定性和运行速度等的提升。

6 问题和优化

下面的内容都将主要用分类任务来说明，因为优化调整是两个任务共同受益的。

6.1 学习率策略

经过使用我们发现，上面基础版的神经网络在两方面表现不是很好：（1）Loss 的曲线经常会卡在某一个值上，无法下降，就如同“砸到一顿墙”，但偶尔会突破这个屏障（因为我们采用随机初始化参数），我们认为这很可能碰到了局部极值点或者是鞍点，并且难以逃脱；（2）收敛速度慢，多次测试都要上百多 epochs。于是我们收集资料想办法突破上面两个问题，这就带来了我们的第一次优化：调整不同阶段的学习率，有助于模型更快达到极值点（收敛更快）以及让模型拥有更多的机会（比如 Momentum）去跨越局部极值点，去探索更多的参数分布区域。我们实现了 $adagrad, RMSProp, Adam$ 等优化器，最后发现 Adam[1] 的优化效果最好：

```
def Adam(self, beta1=0.9, beta2=0.999, epsilon=1e-8):
    self.gW2 = copy.deepcopy(self.gW)
    self.gb2 = copy.deepcopy(self.gb)

    def adam(dW, db):
        self.t += 1
        gW = copy.deepcopy(self.gW)
        gb = copy.deepcopy(self.gb)
```

```
        for i in range(len(dW)):
            # momentum
            self.gW[i] = beta1*self.gW[i] + (1-beta1)*dW[i]
            self.gb[i] = beta1*self.gb[i] + (1-beta1)*db[i]
            # rmsprop
            self.gW2[i] = beta2*self.gW2[i] + (1-beta2)*np.square(dW[i])
            self.gb2[i] = beta2*self.gb2[i] + (1-beta2)*np.square(db[i])
            # update
            gW[i] = self.gW[i] / (1-pow(beta1, self.t))
            gW[i] = gW[i] / ( np.sqrt(self.gW2[i]/(1-beta1**self.t)) +epsilon)

            gb[i] = self.gb[i] / (1-pow(beta1, self.t))
            gb[i] = gb[i] / ( np.sqrt(self.gb2[i]/(1-beta2**self.t)) +epsilon)

        return gW, gb
    return adam
```

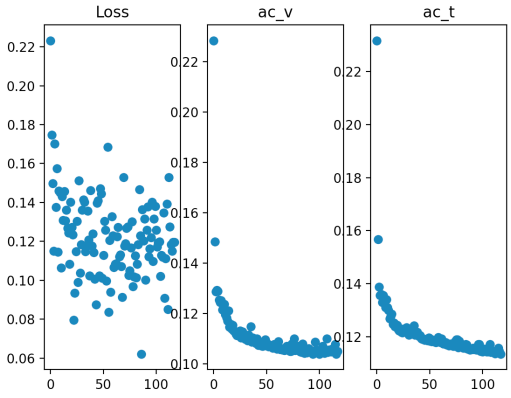
结果对比：采用了学习率优化器前后的模型表现：**运行耗时极大改善**

表 4: 使用了 adam 作为优化器之后的表现：epochs:20

hiddendim	activation	accuracy
[64]	['sigmod', 'sigmod']	0.834
[64,32]	['sigmod','sigmod', 'sigmod']	0.838
[64, 32]	['sigmod', 'Relu', 'sigmod']	0.843
[64, 32， 16]	['sigmod', 'Relu','sigmod','sigmod']	0.814
[100， 64, 32， 16]	['sigmod', 'Relu', 'sigmod', 'Relu', 'sigmod']	0.78

可以直观感受到的是：层数较少的时候，模型表现都非常稳定，而且随激活函数改变的变化不大，但是层数深了之后，明显发生过拟合，这一点我们稍后便会提到；而且，虽然比起没使用优化器之前，准确率只提高了 2% 4%，但是，有一点非常重要的是，训练 epochs 数，明显下降，收敛速度明显提升，将没使用优化器之前轮次耗费从 150 减到了 20，这一时间性能上的改良与朴素贝叶斯拉近了距离。而这也是我们认为这个实验的几个优化中，提升最明显的一次优化。收敛结果：

图 1: 每次迭代记录的 mini-Loss，Val-Loss，trainSet-Loss



然后在这个基础之上，加入了学习率的影响探讨：
选定隐藏层为 [64,32]，激活函数为 [sigmod, ReLU, sigmod],batchsize=128，lossfunctino= 似然函数:

表 5: 学习率对分类结果的影响：

lrate	accuracy
0.0001	0.831
0.001	0.8448
0.005	0.8389
0.01	0.843
0.05	0.8413

选定隐藏层为 [64,32]，激活函数为 [sigmod, ReLU, None],batchsize=128，lossfunction=RMSE:

表 6: 学习率对回归结果的影响：

lrate	Loss
0.0001	0.538
0.001	0.543
0.005	0.548
0.01	0.533
0.05	0.538

我们发现，学习率对于结果影响并不十分明确，反而是，不同的初始化参数造成的结果更大：表现在每次不同的随机初始化的结果会有“二级分化”的现象，有的会被卡在某一个极值点（0.60 左右），有的则可以突破到（0.83）第二个极值点处。所以我们有了第二次优化：

6.2 降低结果对初始参数的依赖性

随着参数的不同初始化，我们还发现，初始化的好坏，会影响到最终的表现：在所有超参数都一致的情况下，模型有时候很快就被极值点卡住，有时候却像是没经过这些极值点直接绕过去；我们的理解是，

不同初始化时的参数梯度下降的走的路径不一样。因此，我们尝试了（1）不同的初始化策略，（Xavier，Kaiming 初始化方法等，最后保留了 $Kaiming/he$ 初始化）（2）Batch Norm

（1）初始化策略：Xavier Glorot 提及，激活值的方差是逐层递减的，这导致反向传播中的梯度也逐层递减，因此，Xavier 初始化提出，将正态分布的初始化参数乘以权重 $1/\sqrt{fan_{in}}$，fan_{in} 为参数的输入维度，以维持传播过程中保持高斯的方差。但是我们也注意到 Xavier 对 relu 做激活的时候，是不能满足该性质的，但我们实际上要用到 relu 作为激活函数，因此我们采用了 Kaiming 初始化 [4]：权重调节为 $\sqrt{2/fan_{in}}$。

（2）Batch Normalization[2]：尽管 Batch norm 是用来调整层间的数据分布，减少层之间的耦合性，从而加快模型对数据分布改变的适应，加快收敛，但是 BN 模型本身对层间的输入数据进行调整，减少了（超）参数对模型学习过程的影响，所以我们认为这能够改善模型过于依赖初始参数的现象。BatchNorm 的原理如下：具体代码见:$code/NerualNet.py/NerualNet(class)/BatchNorm$

比起未采用优化策略之前：更稳定，这正符合我们的预期，因为采用了初始化策略和 Batch normalization，使得我们的模型对于参数/超参的依赖变小了，不同学习率，不同初始化之下的准确率都更稳定了；

稳定性增强：我们使用了从 lr（0.001-1），随机初始化参数，隐藏层保持一到两层，batchsize：128，发现：平均准确率几乎都集中在 0.83～0.84，不会出现卡在 0.6 左右的情况，可见我们模型对超参数的依赖变小。回归任务的时候也变稳定了，loss 也几乎都是落下 0.54.

6.3 模型过拟合

在训练中，我们观察到了过拟合的现象：训练集的 Loss 仍在下降，但验证集的 Loss 已经向上走，我们采用的方法是，减少学习的轮次，模拟 early-stop，在使用了优化器后，大致较好的 epochs 数不超过 25，否则就开始出现明显的过拟合。这时候的验证集上的准确率几乎总能稳定在 0.83～0.84，但实际上，我们这时候在 Private Board 上的表现却不是非常好，只有 0.79。后来我们意识到，这很有可能正是神经网络的局限性导致的：因为需要训练足够多参数，所以 8000+ 的训练集规模并不足以完全训练，反而因为样本数少，模型更倾向于拟合训练集，并且模型复杂度越大（层数加深，维度加大），越来越容易拟合训练集，验证集表现丝毫没有提高，我们参考了这个思路，后来调整参数的时候并没有过多去增加复杂度，一～二层隐藏层的网络完全足够撑起 0.83 的验证集正确率。

7 不足

7.1 任务模型选取

开始的时候我们没有注意到样本数据量可能不足，可能不能发挥神经网络的统计优势，而且容易发生过拟合，神经网络可能并没有特别适合这种情况下的任务；但我们耗费了过多的精力在神经网络的调整，导致可能发挥更好的贝叶斯模型并没有调节到位。

7.2 优化方向

过于注重一般性的优化（比如学习率策略、初始化策略，BN 模型等都是针对神经网络这一模型的），而没有过多去思考，与任务本身相关的优化：经过助教师兄的指导，我们后来知道：幽默文本分类/预测这样主题性明确的任务，一开始可能更适合的思考方向为主题模型，再去查阅相关的资料，而我们几乎

一开始就直奔神经网络这个工具去。对于文本处理并不上心，只是剔除停用词，进行词干化等基本的手段就不再优化。思路上，并没有与任务主题接轨。

7.3 模型调整

对于神经网络超参调整本身，我们几乎是瞎调，没有一个合理的思路，导致调整的时候过乱，最终并没能发挥已有工具的最优性能。

8 总结，分工

本次任务通过两个不同的模型（朴素贝叶斯和神经网络）完成了分类任务、回归任务，发现了神经网络的强大性能，也对对遇到的诸如收敛过慢，过拟合等问题，对神经网络进行调整和优化（如加入了优化器，参数初始化模型，BatchNorm 等），思考了神经网络强大性能的同时带来的局限性。

庄鹏标主要负责文本处理，朴素贝叶斯部分的编写；刘显彬主要负责神经网络部分的基础架构，以及部分优化。经决定选定神经网络为最终框架，最后共同进行神经网络的调整。

参考文献

[1] Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[2] Sergey Ioffe & Christian Szegedy(2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

[3] https://zhuanlan.zhihu.com/p/53277723

[4] Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification