有监督学习-分类&回归

- 使用PLA、LR、knn、朴素贝叶斯实现二分类，得到最高准确率为88.3。
- 使用PLA、LR、朴素贝叶斯、knn实现五分类，得到最高准确率为43.3。
- 使用线性回归模型、对Tag处理，对特征进行提取，进行特征工程处理，实现最高相关系数为0.6569。

（1）算法原理

1. 数据处理

分类：

- 对数据单词的大小写、标点符号进行处理，去掉多余的标点符号，将所有单词统一换成小写，去掉没用的低频词，比如“the”。
- 使用word2vec对单词进行处理，然后对每个句子的单词进行加权求和，实现从高维度降为低维度，提高模型训练速度，但是由于后期模型训练效果很差，估计是因为数据准确性丢失太多，因此后来放弃这种数据处理方法。
- 使用tf-idf矩阵，在使用这种方式时，由于在模型中使用效果不好，而且字典化比较麻烦，因此到后面弃用这种表示方法。
- 使用onehot矩阵，但是由于词向量达到80000+维，因此建立这么大的矩阵不理想，难以继续后面的调参，而且稀疏矩阵很多位置都为0，并没有太多意义，因此使用字典存储onehot，简化矩阵表示，使用梯度下降法时，只需要更新每个句子的出现词的对应的权值就好，训练速度快，没有丢失数据准确性，充分利用数据，准确率高，方便后面的调参。

回归：

- 对Tag进行处理，将Tag出现词建立一个onehot矩阵，同时考虑到某些词对模型的提升效果不好，可能会下降，因此我选择遍历Tag每一个词的onehot向量，最后选出效果最好的22个词，即22个特征向量，加到原有的6个特征向量组成新的矩阵，相关系数提升0.01。
- 经过分析比较，发现原有的6个特征列中Average_Score、Review_Total_Negative_Word_Counts、Review_Total_Positive_Word_Counts三个特征列对模型的训练非常重要，并且还没有充分处理，因此选择对三者进行特征交集处理，比如Review_Total_Negative_Word_Counts，Review_Total_Positive_Word_Counts两列分别代表负面、正面的词的个数，因此可以将两列中每一行负面和正面的单词数相加得出一个总数，分别求出当前行负面单词数和正面单词数占总数的比率，从而得出负面单词率和正面单词率的两个特征列，依次类推，适当对三列特征进行各种交集运算处理，充分利用有用的特征，实验证明，非常有效提高相关系数。（实际上，提升准确率和相关系数在于模型和数据特征处理，如果数据处理得好，即使是简单的模型也能得到很好效果。）

2. PLA

- PLA基于线性回归的方法，使用w向量权值与每一行句子的句子onehot向量x相乘，得出计算结果y，如果y大于等于0，预测为1，如果y小于0，预测为0，由于使用onehot矩阵时，每次w与x相乘时，x中所有为0的位置与对应的w向量的权值相乘时为0，没有计入结果y中，只有x中为1的对应权值才累加到y中，因此为了提升训练速度，使用字典存储onehot，因此在计算y时，每次选择句子中出现的词的向量权值进行累加，得到y，效果跟使用onehot矩阵一样，但效率高。
- 由计算结果y得到预测结果y1后，与正确结果比较，如果相同则不用更新，否则需要进行更新，对句子中每一个词的权值w1进行更新，引入一个学习率rate，调整更新的幅度，使用梯度下降法更新权值，每次迭代遍历所有句子，判断每次遍历预测结果是否正确，正确则继续，否则进行更新，直到梯度下降条件终止。
- 两种梯度下降方法：
    - 使用不变的学习率rate，每一次迭代计算训练数据的准确率，直到某一次准确率达到1.0，停止训练，然而这样往往会过拟合，因此最终测试准确率在0.873。
    - 使用可变的学习率rate，每一次迭代对rate=rate/迭代数计算，进行30次迭代，训练的梯度步伐会越来越小，更好达到最终点，训练准确率只有90.03，但是测试准确率上升到0.883。

3. 逻辑回归

- 逻辑回归同PLA一样，使用基于线性回归和onehot字典，将w与句子向量x相乘得出计算结果y后，使用sigmoid激活函数得出预测结果y1，根据逻辑回归的梯度下降公式推导，计算出实际值与预测结果y1的误差error，对句子中的每一个词的权值w1进行更新。
- 使用两种梯度下降方法：
    - 不变学习率rate，每次迭代计算训练准确率，只有预测不准确进行更新，直到预测准确率达到0.99，停止训练，与PLA一样，这种方法会过拟合，测试准确率只有0.853。
    - 可变学习率rate，每次迭代rate=rate/迭代数，30次迭代，所有数据都更新，训练准确率只有90.66，测试准确率上升为0.878。

4. 朴素贝叶斯

- 基于概率公式，以二分类为例，首先计算label值为1的概率p_1，用p1_dic字典存储label值为1对应的句子中各个词出现的次数，用p1_total统计label值为1对应的句子的单词总数，针对每一行句子，判断该句子的label是否为1，如果是1，则该句子中的各个词在p1_dic对应的出现次数+1，该句子的单词总数叠加到p1_total中。统计完毕后，将p1_dic中各个词的出现次数除以p1_total作为各个词在label值为1的所有单词中的频率。
- 为了避免结果出现下溢，使用对数存储上一点提到的频率，方便在下面将相乘改为相加。
- 同样，计算出label值为0的相应结果。
- 在预测时，针对当前句子，计算label为1概率时，将句子中各个词在p1_dic中对应的value值相加，最后在加上math.log(p_1)，得到label为1的概率，同理得到label为0的概率，根据概率的大小比较得出预测结果，五分类也是同样的方法。

5. 线性回归

- 首先进行特征处理，对原有特征进行特征工程处理，再加上偏置单元，扩展为31列特征，对Tag进行onehot处理，经过比较，去除没用的特征词，最后选出22列特征，最终扩展为53列特征。
- 使用线性回归，对特征矩阵的每一行x，与向量w相乘，得到预测结果y，求出误差error=true_y-y，使用梯度下降法，学习率rate随着迭代次数增加而下降，迭代次数设为30，不断更新优化w，直到效果最好。

6. knn

分类：

- 使用word2Vec处理数据降维为500维，使用欧式距离为度量方式，设置k值为150，选出150个与预测句子欧式距离最近的句子，对句子进行加权处理，同时给距离一个惩罚系数，距离越大，惩罚系数越高，距离越短，权值越大，重要性越强，选出权值最高的label值。

回归：

- 使用余弦相似度为度量方式，设置k值为150，选出150个与预测句子余弦相似度最大的句子，针对余弦相似度进行重要性处理，数值越大，越重要，其权值越重要，对句子的label值进行加权处理，得出最终的预测结果。

（2） 伪代码

1. PLA

```
Algorithm PLA
    Input: train_list,label_list
    Output: weights
  rate <- 0.1 #学习率
  cycle <- 30 #迭代次数
  for k = 0 to cycle do
    rate <- rate / (k+1) #更改学习率
    for i = 0 to train_size do
        word_list <- train_list[i][:]
        y <- 0
        for word in word_list do
            y += weights[word]
        sign_y = sign(y)
        if sign_y = label_list[i] then
            continue
        else
            #梯度下降，更新权值
            for word in word_list do
                weights[word] += rate*label_list[i]*1
  return weights
```

2. 逻辑回归

```
Algorithm Logistics Regression
    Input: train_list,label_list
    Output: weights
  rate <- 0.1
  id <- 1
  while True:
    rate = rate/id
    id += 1
    for i = 0 to train_size do
```

```
        word_list <- train_list[i][:]
        y <- 0
        for word in word_list do
            y += weights[word] * 1
        #使用sigmoid激活函数
        sig_y <- sigmoid(y)
        #求误差
        error <- label_list[i] - sig_y
        #梯度下降，更新权值
        for word in word_list
            weights[word] += rate*error*1
    if id >= 30 then
        break
  return weights
```

3.朴素贝叶斯

```
Algorithm NB
    Input: train_list,test_list,train_label,test_label
    Output: result
  p1_dic <- {} #存储label值为1对应各个词出现的频率
  p0_dic <- {} #存储label值为0对应各个词出现的频率
  p1_total <- 1.0 #统计label值为1的词的总数
  p0_total <- 1.0 #统计label值为0的词的总数
  for i = 0 to train_size do
    word_list <- train_list[i][:]
    #如果label为1，则增加句子的词总数和更新各个词出现的次数
    if train_label[i] = 1 then
        p1_total += len(word_list)
        for word in word_list do
            p1_dic[word] += 1
    #如果label为0，则增加句子的词总数和更新各个词出现的次数
    elif train_label[i] = 0 then
        p0_total += len(word_list)
        for word in word_list do
            p0_dic[word] += 1

  for every key in p1_dic do
      #对数处理，防止下溢
      p1_dic[key] <- log(p1_dic[key] / p1_total)
  for every key in p0_dic do
      p0_dic[key] <- log(p0_dic[key] / p0_total)

  #求各自label值的频率
  p1_rate <- count(label=1) / train_label_size
  p0_rate <- count(label=0) / train_label_size

  result <- []
  for i = 0 to test_size do
      p1 <- log(p1_rate)
      p0 <- log(p0_rate)
```

```
      for word in test_list[i] do
          p1 += p1_dic[word]
          p0 += p0_dic[word]
      #选择概率最高的作为预测值
      if p1 > p0 then
          test_y <- 1
      else
          test_y <- 0
      result.append(test_y)
  return result
```

4. 线性回归

```
Algorithm Linear Regression
    Input: train_array,test_array,train_label,test_label
    Output: result
  alpha <- 0.05
  cycle <- 30
  for i = 0 to cycle do
    alpha <- alpha/(i+1) #随着迭代数增加，不断减少学习率
    for k = 0 to train_size do
        x <- train_array[k][:]
        y <- np.dot(x,weights)
        error <- train_label[k] - y #误差
        weights += alpha*error*x #梯度下降，更新权值

  result <- []
  for i = 0 to test_size do
    x <- test_array[i][:]  #x向量
    y <- np.dot(x,weights)  #预测值
    result.append(y)
  return result
```

5. knn

```
Algorithm knn
    Input: train_list,test_list,train_size,test_size
    Output: result
  k_value <- 150 #选择的k个参考句子
  weight <- 5  #加权处理的幂值
  result <- []
  for i = 0 to test_size do
    cosine_similarity <- {}
    #计算测试句子与每一个训练集句子的余弦相似度
    for j = 0 to train_size do
        m <- np.linalg.norm(train_list[j]) * np.linalg.norm(test_list[i])
        if m != 0 then
```

```
            cosine_similarity[j] <- np.dot(train_list[j],test_list[i]) / m
        else
            cosine_similarity[j] <- 0
    #对数据按余弦相似度从大到小排序
    sorted(cosine_similarity[j])
    #得到前k_value个句子作为最终预测的参考
    data <- cosine_similarity.getmax(k_value)

    sum <- 0
    for k = 0 to k_value do
        sum += pow(data[k][1]*15+1,weight)

    test_score <- 0
    for k = 0 to k_value do
        index <- data[k][0]
        #余弦相似度越大，占的比例越高，对其重要性进行幂次运算，加大其所占的权重
        w <- pow(data[k][1]*15+1,weight)
        #权重比例
        weight1 <- w/sum
        #加权处理，得到最终的预测结果
        test_score += train_label[index]*weight1

    result.append(test_score)
```

（3） 关键代码

1. PLA

- 基于训练集对数据进行训练，得到最佳的权值weights

```
    cycles = 30 #迭代次数
    alpha = 0.1 #学习率
    for j in range(cycles):
        correct_num = 0
        print(j+1)
        #随着迭代次数增加，不断减少学习率
        alpha = alpha/(j+1)
        for i in range(train_size):
            tmp_list = train_list[i][:]
            y = 0.0
            #针对句子中的每个词，增加这个词对应的权值
            for k in tmp_list:
                y += (weights[k] * 1)
            #如果y大于等于0，预测为1，y小于0，预测为0
            if y >= 0.0:
                sign_y = 1.0
            else:
                sign_y = 0.0
            #如果预测结果正确，不更新权值
            if sign_y == float(label_list[i]):
```

```
                correct_num += 1
                continue
            #预测不正确，更新权值
            else:
                #如果结果为1，则更新值为1.0，否则为-1.0
                if float(label_list[i]) == 1.0:
                    e = 1.0
                else:
                    e = -1.0
                #对每个词进行权值的更新
                for k in tmp_list:
                    weights[k] = weights[k] + e*alpha
        print(float(correct_num) / train_size)
```

- 利用得到权值，对测试集数据进行预测，对每一行句子，将每一个词的权值累加，根据得到结果，如果大于等于0，预测为1，否则预测为0。

```
    #遍历每一个句子，得到预测结果
    for i in range(test_size):
        tmp_list = test_list[i][:]
        test_y = 0.0
        #对于句子的每一词，如果存在对应的权值，直接相加
        for k in tmp_list:
            if  k in weights.keys():
                test_y += weights[k] * 1
        #如果结果大于0，预测为1，否则预测为0
        if test_y >= 0.0:
            sign_y = 1.0
        else:
            sign_y = 0.0
        if sign_y == float(label_list1[i]):
            accurate_number += 1
```

2. 逻辑回归

- 利用字典化的onehot，针对每一个句子，只累加句子中的每个词的权值，经过sigmoid函数处理，然后对每个词的权值进行梯度下降更新。

```
    train_size = len(train_list)
    alpha = 0.1 #学习率
    id = 1 #迭代的序号
    while True:
        correct_number = 0
        print(id)
        # 不断更新学习率
        alpha = alpha / id
        id += 1
        #遍历所有的句子，不断训练权值
        for j in range(train_size):
            tmp_list = train_list[j][:]
```

```
            y = 0.0
            #累加每个词的权值
            for word in tmp_list:
                y += weights[word] * 1
            y = round(y,6)
            #经过sigmoid激活函数处理
            value = sigmoid(y)
            if (round(value) == int(label_list[j])):
                correct_number += 1
            #误差
            error = float(label_list[j]) - value
            #对句子的每一个词，更新词的权值
            for word in tmp_list:
                weights[word] = weights[word] + alpha * error * 1
        rate = float(correct_number)/train_size
        print( float(correct_number)/train_size )
        if id >= 30:  #迭代次数达到30次，结束训练
            break
```

训练好模型后，对测试集结果进行预测，对线性求和的结果进行sigmoid函数处理，使结果数值范围在（0,1），四舍五入得到预测结果。

```
accurate_number = 0 #统计预测正确的个数
#遍历测试集
for i in range(test_size):
    tmp_list = test_list[i][:]
    test_y = 0.0 #初始化预测结果
    #对句子中每个词的权值进行累加
    for word in tmp_list:
        if word in weights.keys():
            test_y += weights[word] * 1
    #对小数点后6位进行四舍五入
    test_y = round(test_y,6)
    #经过sigmoid激活函数后的预测结果与正确结果比较，正确则统计正确数加1
    if round(sigmoid(test_y)) == int(label_list1[i]):
        accurate_number += 1
```

3. 朴素贝叶斯

- 对训练集数据进行处理，同时统计每个label对应的总词数，以及每个词的频数，分别用p_total, p_list表示。

```
train_size = len(file1_list) #训练集句子的行数
train_list = []
p1_list = {}  #存储label为1的词的频数
p0_list = {} #存储label为0的词的频数
p1_total = 1.0 #存储label为1的词总数
p0_total = 1.0 #存储label为0的词总数
#遍历每一行句子
for i in range(train_size):
```

```
        words = file1_list[i].split()
        #label为1的处理
        if int(label_list1[i]) == 1:
            #增加词的总数
            p1_total += len(words)
            #增加每个词的频数
            for word in words:
                if word not in p1_list.keys():
                    p1_list[word] = 1
                else:
                    p1_list[word] += 1
        #label为0的处理
        else:
            #增加词的总数
            p0_total += len(words)
            #增加每个词的频数
            for word in words:
                if word not in p0_list.keys():
                    p0_list[word] = 1
                else:
                    p0_list[word] += 1
        train_list.append(words)
```

- 统计label为1和0的各个词的频率，为了体现词频率的重要性，进行加权处理，乘以10000000，同时为了防止下溢，因此对频率进行对数处理，在后面的预测中，化相乘为求和。

```
    #计算每个词在对应label中的频率,加权对数处理
    for key in p1_list.keys():
        p1_list[key] = math.log((float(p1_list[key]) / p1_total)*10000000 )
    for key in p0_list.keys():
        p0_list[key] = math.log((float(p0_list[key]) / p0_total)*10000000 )
```

- 对每一个测试句子，分别统计各个label的概率大小，选择概率最大的label作为预测结果。

```
    correct_num = 0 #统计准确个数
    for i in range(test_size):
        p1 = math.log( p_1 ) #label为1的可能性
        p0 = math.log( 1.0 - p_1) #label为0的可能性
        #针对每个词，分别累加到对应的可能性
        for word in test_list[i]:
            if word in p1_list.keys():
                p1 += p1_list[word]
            if word in p0_list.keys():
                p0 += p0_list[word]
        #比较可能性大小，可能性大的预测为对应的label
        if p1 > p0:
            test_y = 1
        else :
            test_y = 0
        #预测正确，统计个数加1

        if test_y == int(label_list2[i]):
```

```
            correct_num += 1
```

4. 线性回归

- 为了便于后面的线性计算，将二维列表的训练和测试集转为array，并且统一转化为float类型。

```
train_array = np.array(train_list) #转化为array
train_tag1 = train_array[:, 6] #取出第7列Tag
train_array = np.delete(train_array,6,axis=1) #删除第7列
train_array = np.array(train_array,dtype=float) #将值由str转为float

test_array = np.array(test_list) #转化为array
test_tag1 = test_array[:, 6] #取出第7列Tag
test_array = np.delete(test_array,6,axis=1) #删除第7列
test_array = np.array(test_array,dtype=float) #将值由str转为float
```

- 为了能够更好地线性拟合数据，针对6个特征列数据大小不一情况，对数据进行标准化。

```
#标准化
for i in range(6):
    word_list = train_array[:,i] #每一特征列的数据
    max_value = max(word_list) #求该列的最大值
    min_value = min(word_list) #求该列的最小值
    ran = max_value - min_value #该列的值范围大小
    word_size = len(word_list)
    for k in range(word_size):
        #根据标准化公式，得出每个数据在0-1的比例
        word_list[k] = float(word_list[k] - min_value) / ran
    train_array[:,i] = word_list #将处理的列更改到train_array
```

- 处理Tag数据，首先对Tag进行预处理，构建好onehot矩阵后，使用循环遍历方式后，将每一列分贝插入到模型特征列中，观察对相关系数的影响，经过比较，最后选出22个对相关系数有提升意义的词。

```
#对相关系数有利的词
train_word = [' Budget Single Room Non Smoking ', ' Business trip ', ' Couple ',
              ' Deluxe Queen Guestroom ', ' Double Queen Waterfront ',
              ' Double or Twin Room with Sea View ', ' Duplex ', ' Embassy Suite ',
              ' Family with young children ', ' Group ', ' King Duplex Suite ',
              ' King Hilton Sea View ', ' King Room with Balcony ',
              ' Large Double Room ', ' Leisure trip ', ' Prestige Suite ',
              ' Privilege Junior Suite with Spa Access ',
              ' Privilege Room with 1 Queen Size bed ', ' Queen Guestroom ',
              ' Solo traveler ', ' Stayed 3 nights ', ' Studio King Family ']
```

- 针对选出的词，进行onehot矩阵构建。

```
train_one_hot = [] #onehot矩阵
#遍历每个训练集句子
for i in range(train_tag_size):
    tmp = [0] * train_word_size #初始化全0
    #遍历句子中的每一个词
    for word in train_tag[i]:
        #如果词存在词列表，进行下一步
        if word in train_word:
            index = train_word.index(word) #得出下标
            tmp[index] = 1 #在当前下标上将值置为1
    train_one_hot.append(tmp) #加入到onehot矩阵
```

- 对原有特征列进行特征工程处理，对Review_Total_Negative_Word_Counts、Review_Total_Positive_Word_Counts两列进行处理，对于每一行，首先计算出negative和positive的个数，将两者相加为total，然后计算rate1=negative的个数/total，rate2=positive的个数/total，得出两列新的比率，作为两个新的特征，依次类推，对其他特征列也进行特征多项式处理，最后加上第一列偏置单元，得到的特征列为31列。

```
#训练集
negative1 = train_array[:,3] #negative列
positive1 = train_array[:,5] #positive列
size1 = len(negative1)
negative1_rate = [] #存储比率
positive1_rate = []
#遍历每一行句子
for i in range(size1):
    total = negative1[i] + positive1[i] + 1 #求和
    ne_rate = float(negative1[i]) / total #求negative的负比率
    po_rate = float(positive1[i]) / total #求的positive的正比率
    negative1_rate.append(ne_rate)
    positive1_rate.append(po_rate)

train_array = np.insert(train_array,7,negative1_rate,axis=1) #将负比率存储在第8列
train_array = np.insert(train_array,8,positive1_rate,axis=1) #将正比率存储在第9列
```

- 将扩展的特征列与处理Tag后的onehot列合并。

```
train_one_hot = np.array(train_one_hot,dtype=float)  #转化float格式
test_one_hot = np.array(test_one_hot,dtype=float)
train_array = np.append(train_array,train_one_hot,axis=1) #按照列格式合并两个array
test_array = np.append(test_array,test_one_hot,axis=1)
```

- 离散化，如果仅仅是一条直线，很难直接拟合数据，很难取得更好的效果，因此将训练数据的二维array进行离散化，将矩阵上下分成两部分，同时构建两个新的特征矩阵，特征矩阵1上部分为矩阵的上部分数据，下部分为全0，特征矩阵2上部分为全0，下部分为矩阵的下部分数据，从而对直线进行分段化，更好地拟合数据。

```
length1 = len(train_array[0]) #列数
```

```
    #遍历所有列数
    for i in range(length1):
        array1 = train_array[:,i]
        length2 = len(array1) #列长度
        half = int(length2/2) #列中间的下标
        arr1 = [] #列表1
        arr2 = [] #列表2
        #列的上部分处理
        for j in range(0,half):
            arr1.append(0)
            arr2.append(array1[j])
        #列的下部分处理
        for j in range(half,length2):
            arr1.append(array1[j])
            arr2.append(0)
        #将arr1更新到当前的列位置
        train_array[:,i] = np.array(arr1)
        #将arr2插入到后面
        train_array = np.insert(train_array,i+length1,arr2,axis=1)
```

- 进行多次迭代，根据迭代次数不断更新学习率，利用梯度下降法，不断更新权值，得出最优解。

```
    weight_size = len(train_array[0])
    weights = np.ones(weight_size) #初始化权值
    alpha = 0.05 #学习率
    cycle = 30 #迭代次数
    for i in range(cycle):
        print(i+1)
        alpha = alpha / (i+1) #更新学习率
        total_error = 0
        #遍历所有行
        for k in range(train_size):
            x = train_array[k][:]
            y = np.dot(x , weights) #线性求和
            error = train_label[k] - y #求误差
            weights = weights + alpha * error * x #更新权值
            total_error += error
```

- 利用解析解求出最优权值weights。

```
    train_mat = np.matrix(train_array) #转化矩阵
    train_label_mat = np.matrix(train_label).T #转化为矩阵，求逆
    xtx = train_mat.T * train_mat #求X.T*X
    if np.linalg.det(xtx) != 0.0:
        weights_list = xtx.I*(train_mat.T*train_label_mat) #解析解公式
```

5. knn

- 同线性回归一样，首先对六列特征列进行标准化，然后进行特征工程处理，接着对Tag进行onehot处理，选出有用的tag词汇，加入特征集，代码不再重复张贴。
- 遍历测试集的每一行进行预测，首先采取余弦相似度求出当前行与训练集所有行的余弦相似度，然后按照从大到小进行排序，选出前k个，根据权值处理求和得到最终预测结果。

```
    for x in range(test_size):   #遍历每个句子
        print(x+1)
        sorted_data = measure(train_array,test_array,x,train_size,measure_choice)   #采用对应的测量方式得到最佳的k个样本
        test_score = get_final_emotion(train_array,sorted_data,k_value,weight_coefficient,measure_choice)  #得到预测样本的各个概率
        test_score_list.append(test_score) #预测结果列表
        score_list.append(float(test_array[x][-1])) #标准结果列表
        print(abs(test_score - test_array[x][-1])) #计算误差，观察预测过程误差范围
```

- 采用余弦相似度计算。

```
#余弦相似度
def get_cosine_similarity(train_array,test_array,x,sample_list_size):
    cosine_similarity = {}
    for i in range(sample_list_size):
        m = np.linalg.norm(train_array[i]) * (np.linalg.norm(test_array[x]))
        if (m != 0):
            cosine_similarity[i] = np.dot(train_array[i], test_array[x]) / m   #应用余弦公式
        else:
            cosine_similarity[i] = 0
    return cosine_similarity
```

- 根据k个最近样本，对每个样本进行加权处理，同时根据余弦相似度大小，余弦相似度越大，赋予权值越高，最后加权求和得出预测结果。

```
    test_score = 0.0
    #余弦相似度的求解方法
    if measure_choice == 2:
        sum1 = 0
        for i in range(k_value):
            sum1 += math.pow((sorted_data[i][1] * 15 + 1), weight_coefficient) #求出总权值
            # sum1 += math.pow(math.e,(sorted_data[i][1]*15+1))
        for i in range(k_value):
            index = sorted_data[i][0]  #得到下标
            min = math.pow((sorted_data[i][1] * 15 + 1), weight_coefficient)
            # min = math.pow(math.e,(sorted_data[i][1]*15+1))
            weight = min / sum1  #权值占比
            test_score += train_array[index][-1]*weight
```

（4）创新点&优化

1. 使用字典简化onehot矩阵，在处理分类问题上，PLA和逻辑回归均使用onehot计算，由于词维度达到80000+维，因此迭代过程慢，效率低，对于onehot矩阵，只有词出现的位置为1，其他位置为0，为了提高效率，使用字典存储onehot，key值为词，value值为词的onehot值，遍历每一行句子，直接计算出现的词的w权值，其他没有出现的词由于其onehot为0，故可以忽略不计，提高运算效率，并且经过适当的迭代次数和学习率调整，准确率可以达到88%。
2. 回归特征工程处理：回归项目的六个特征中有三个特征很重要，对最后的结果影响很大，如果用线性回归实现，仅仅依靠六个特征难以很好地拟合数据，因此经过验证，发现Average_Score，Review_Total_Negative_Word_Counts，Review_Total_Positive_Word_Counts三列数据对最后相关系数影响很大，可以对这三列数据进行多项式处理，扩展成为新的特征，比如negative和positive列代表负面和正面单词的数目，我们可以统计负面和正面的总数，然后分别计算负面和正面的占总数的比率，扩展为新的两个特征，依次类推，从而扩展更多的特征列，相关系数由0.55提升到0.64。
3. 回归Tag处理，Tag的词向量维度有1700+维，如果全部添加为特征，用线性回归无法很好地更新权值，因此对Tag词向量构造onehot矩阵，然后遍历Tag词向量，每一个循环依次将每一维度的Tag向量作为特征加入到训练集，测出每一个Tag词向量对相关系数的效果，用这样的方法最后选出22个词向量，加入训练集，最后将相关系数提升0.01。
4. 回归数据离散化，如果仅仅是线性式子，难以很好地拟合数据，然而，将特征列进行离散，分成多段，扩展新的特征列，可以让直线分段化，从而更好地拟合数据。
5. 朴素贝叶斯简化词向量处理：按照贝叶斯公式，需要针对基于label的条件下计算各自词的频率，如果使用词向量叠加，最后除以词总数得到，由于分类数据中词的维度有80000+，循环遍历统计效率比较低，所以针对每一个label值用字典存储词对应的频率，key值为词，value值为频率值，预测时只需要计算对应的词的频率的乘积即可。
6. 朴素贝叶斯概率权值和对数处理：由于部分词出现频率很低，如果直接相乘的话容易出现下溢，预测效果很差，故在此基础上首先对词的概率乘上经过多次测试得出的权值10000000，然后再进行对数处理，这样概率相乘就变成了相加，二分类准确率由0.65提升到0.85，五分类准确率是0.385。

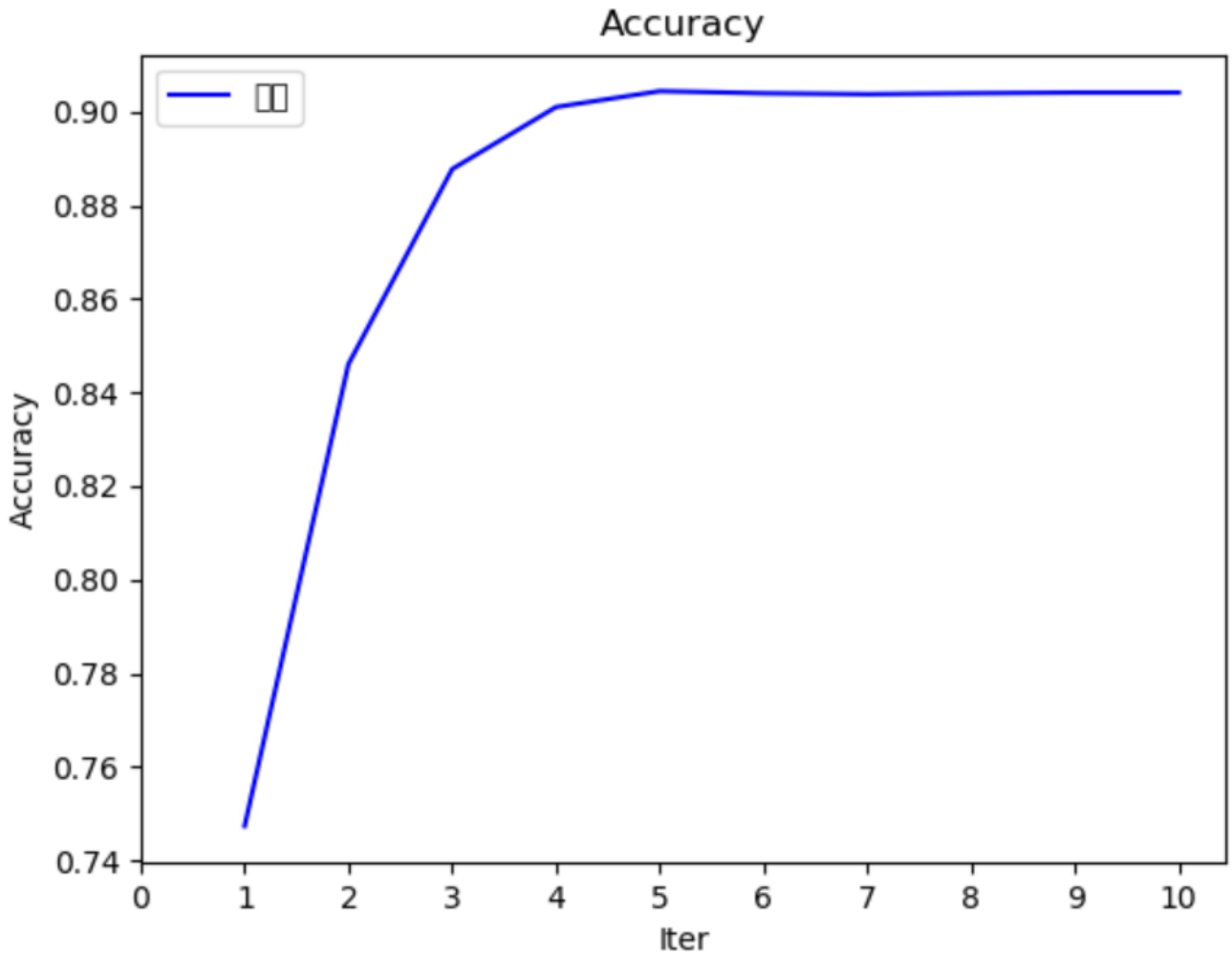
（5）实验结果及分析

1. PLA

A. 二分类：

*使用梯度下降法，学习率随着迭代次数增加而不断减少，统计每一次迭代的训练数据的准确率，观察PLA训练过程。

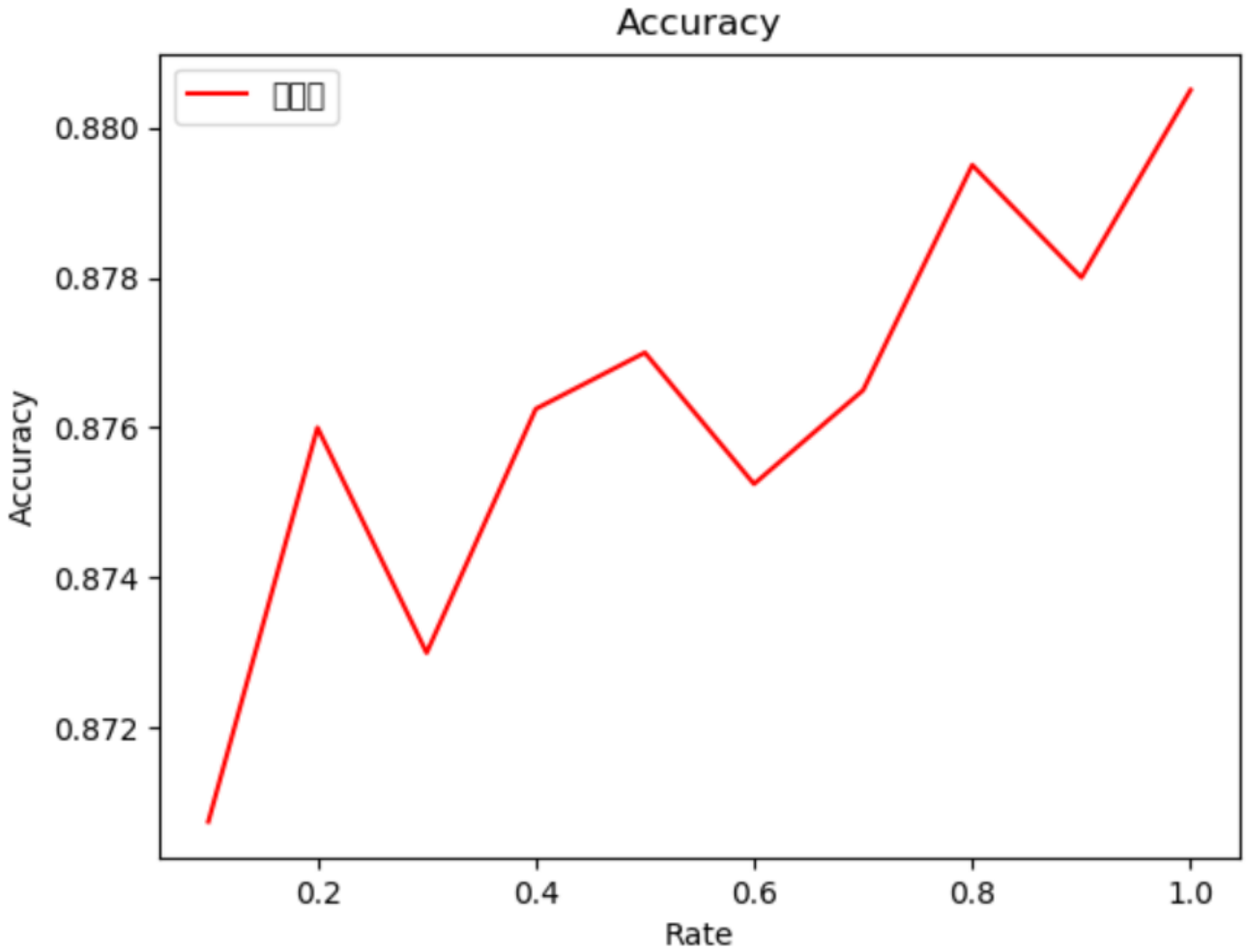
迭代次数	1	2	3	4	5	6	7	8	9	10
准确率	0.74735	0.84605	0.88775	0.901	0.9044	0.90395	0.90375	0.90395	0.9041	0.9041



由图可知，训练过程中PLA算法准确率逐渐上升，说明梯度下降法有明显效果，随着迭代次数增加，准确率上升趋势变换，根据9、10次迭代可知准确率保持稳定，说明梯度下降完成。

* 比较不同的初始学习率（每次迭代，学习率更新为学习率/当前迭代次数）对验证集准确率的影响。

学习率	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**
准确率	0.87075	0.876	0.873	0.87625	0.877	0.87525	0.8765	0.8795	0.878	0.8805



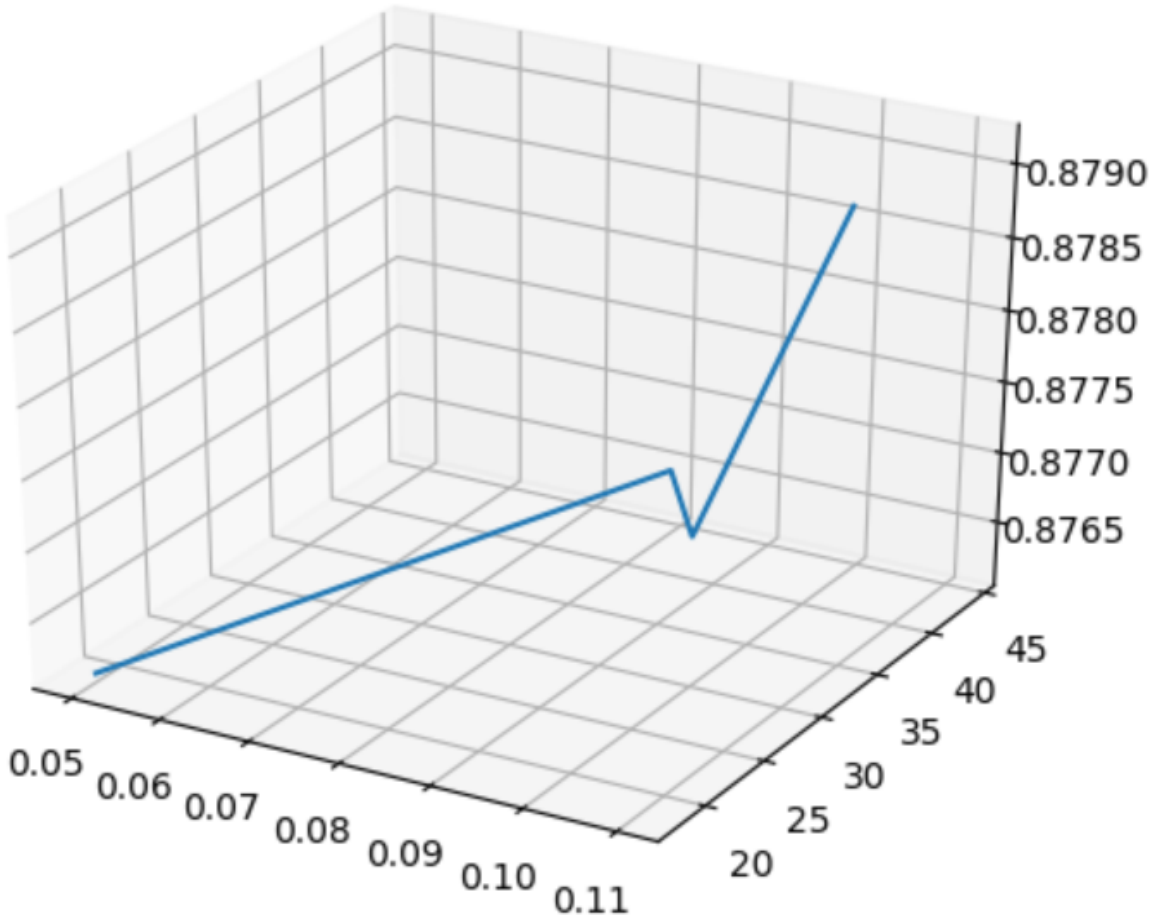
通过比较分析，可以知道学习率为1.0时，验证集准确率最高。

2. 逻辑回归

A. 二分类：

*梯度训练过程与PLA相似，学习率随着迭代次数不断更新降低，比较经过调参各项的准确率。

初始学习率	迭代次数	验证集准确率	测试集准确率
0.1	30	0.874	0.8771
0.05	18	0.875	0.8761
0.11	35	0.87375	0.8792
0.08	44	0.8735	0.8765

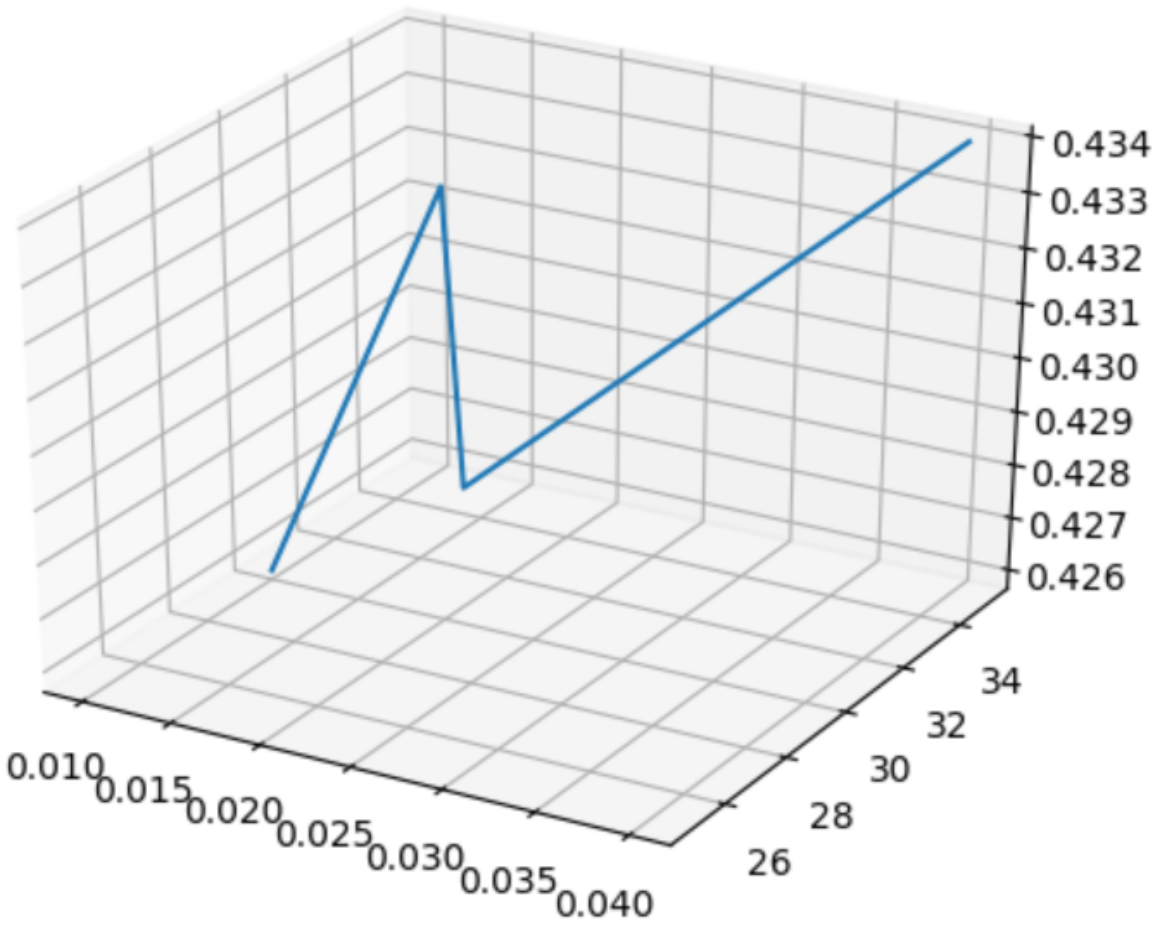


相比之下，逻辑回归二分类效果不如PLA，并且后期调参效果不理想。

B. 五分类：

*比较不同的学习率以及不同的迭代数对准确率的影响，经过比较，选出几个效果比较好的调参结果。

初始学习率	迭代次数	验证集准确率	测试集准确率
0.01	30	0.4241	0.4258
0.02	30	0.428	0.4336
0.03	25	0.4276	0.4308
0.04	35	0.42925	0.434



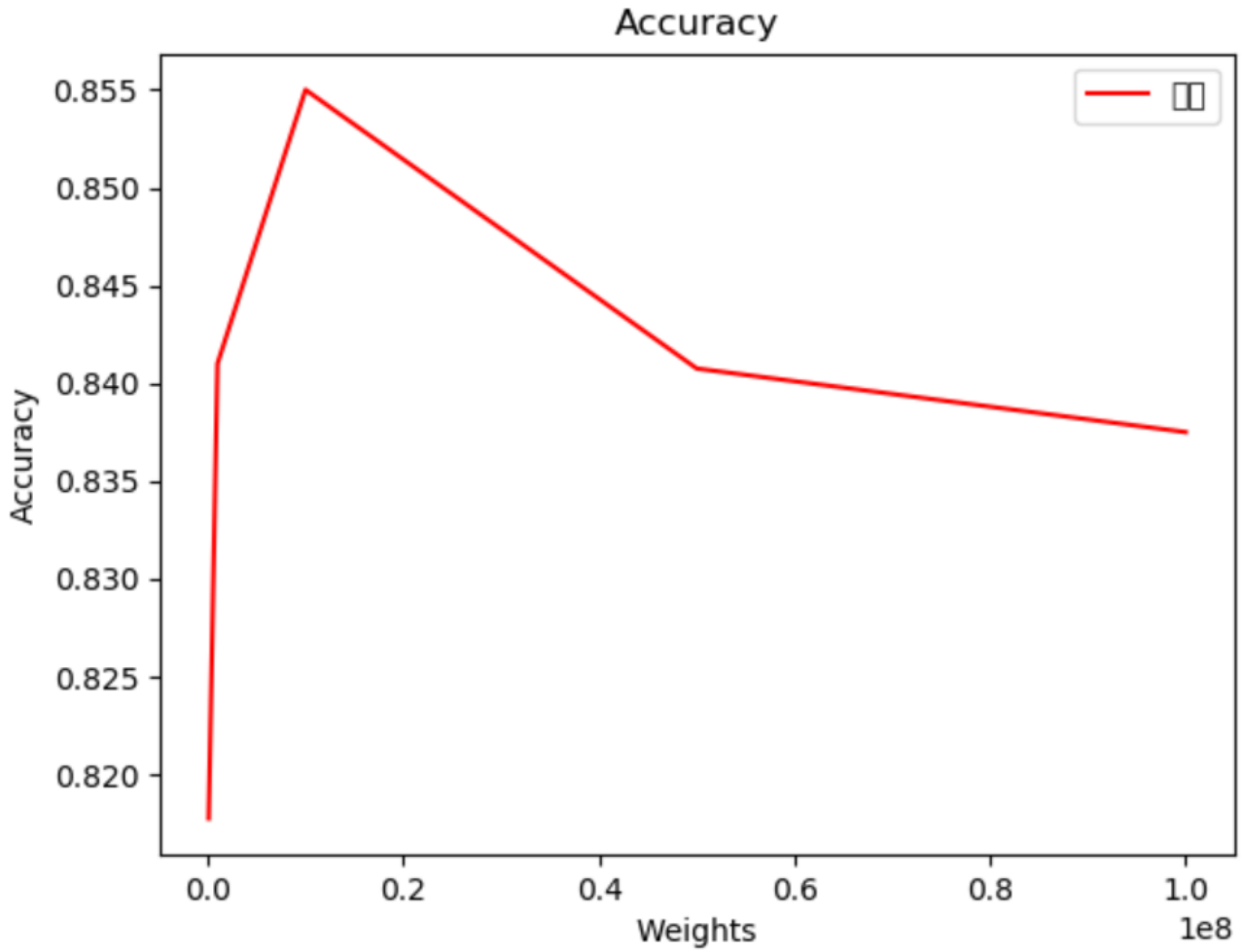
由此可见，当学习率为0.02或者0.04，迭代次数在30-35，五分类的准确率比较高，达到0.43以上。

3. 朴素贝叶斯

由于朴素贝叶斯算法比较固定，因此并没有太多的参数可调，可以对概率进行加权处理。

A. 二分类

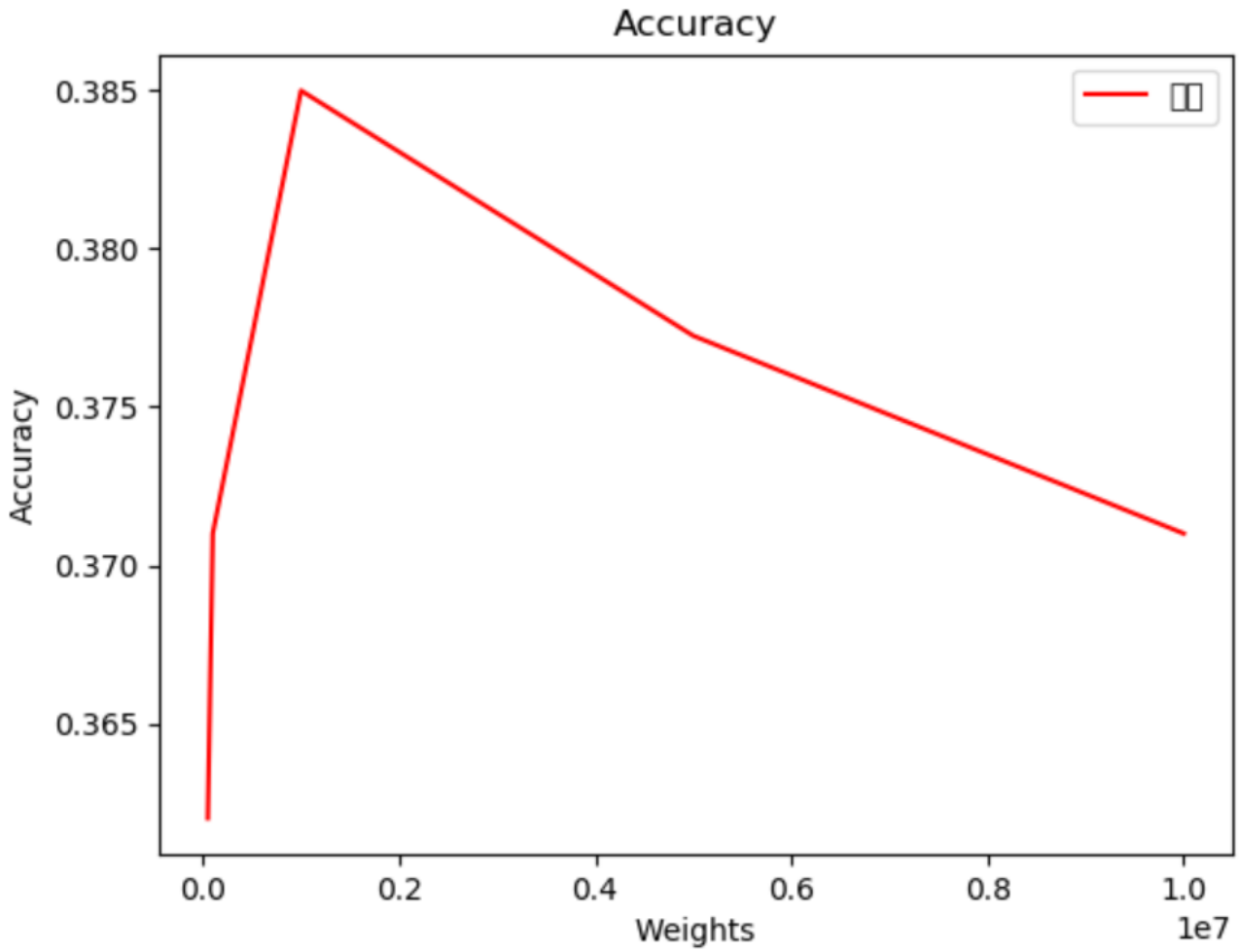
权值	**10 0000**	**100 0000**	**1000 0000**	**5000 0000**	**10000 0000**
准确率	0.81775	0.841	0.855	0.84075	0.8375



由图可得随着权值增大，准确率越高，当超过一定值时，准确率下降，并且NB二分类效果比PLA、逻辑回归要差。

B. 五分类

权值	**5 0000**	**10 0000**	**100 0000**	**500 0000**	**1000 0000**
准确率	0.362	0.371	0.385	0.37725	0.371



根据分析，可以知道NB的最高准确率为0.385，权值为100 0000，跟逻辑回归的最高值有一定差距，由于NB没有其他参数可调，因此这是NB的优化后的结果。

4. 线性回归

A. 回归

比较对特征进行的各种数据处理，从而得出对验证集和测试集的相关系数影响，下面各项处理都是基于上一行的处理的基础上进行下一步的处理。

数据处理	验证集相关系数	测试集相关系数
初始六列特征	0.55	0.562
特征工程处理	0.648	0.651
数据标准化	0.646	0.643
数据离散化	0.647	0.649
加入Tag处理	0.651	0.656

由表可知，对原有特征进行特征工程处理和加入Tag处理后，相关系数提升效果最好，到达0.656，而数据标准化和数据离散化对相关系数影响不大。

5. knn

由于knn算法在本次项目中的准确率和相关系数比较低，因此后期优化放弃了knn算法。

A. 分类：knn分类的效果不够理想，准确率只有0.66，而且运行时间很长。

k	度量方式	准确率
150	欧式距离	0.66

B. 回归：跟线性回归相比，回归效果较差，运行效率较低。

k	度量方式	相关系数
150	余弦相似度	0.48