AI第二次试验 实验报告

王宇飞 PB14011018

本次实验包括两个部分，电影评价分类和图像分割。

Part1 电影评价分类

实现数据集的特征抽取

这次试验我分别采取了不同的特征抽取方式，以下会一一说明。

本次实验最简单的特征抽取方式就是不加任何特殊处理，直接将给定的数据文件读入作为数据写入到内存中，作为之后分类操作的数据来源。这样做的话，必须要使用稀疏矩阵来保存数据；否则`50000*80000`的矩阵将会耗费大量的内存空间，并且使算法的运行极为缓慢。稀疏矩阵的实现可以采用`scipy.sparse`库中的`csr_matrix`，因为根据数据集的性质，数据集对应的数据存储矩阵应该是一个列稀疏的矩阵，因此应该使用行作为主元。

对于这一部分，完整的代码如下：

```
def getFeature():
    fileData = open("data")
    row = []
    col = []
    data = []
    evalRes = []
    rowIndex = -1
    fileList = fileData.readlines()
    random.shuffle(fileList)
    for line in fileList:
        line = line.rstrip('\n')
        dataList = re.split(' |:', line)

        if int(dataList[0]) >= 7:
            evalRes.append(1)
        else:
            if int(dataList[0]) <= 4:
                evalRes.append(-1)
            else:
                continue
        del dataList[0]

        rowIndex = rowIndex + 1
        row.extend([rowIndex] * int(len(dataList) / 2))
        col.extend(map(int, dataList[::2]))
        data.extend(map(int, dataList[1::2]))

    featureMatrix = csr_matrix((data, (row, col)))
    return featureMatrix, evalRes
```

这里在实现的时候有几个部分需要注意：

1. 一次循环可以处理一行来节省算法的运行开销；
2. 在合并列表的时候，一定要使用`extend`函数来扩展列表，而不能直接将两个列表相加，否则两个列表相加的时候会产生一个新的列表对象，这种方式会极大地拖慢算法的运行速度；
3. 每次处理的时候，最好先使用`shuffle`函数将给定的数据集乱序，否则原始数据集中前25000条都是正例而后15000条都是负例，这样会在划分5fold的时候影响算法的训练效果（比如某一个测试集中可能全部都是正例）。

这样处理的特征集在利用朴素贝叶斯算法进行分类的时候，平均正确率可以达到大约72%左右，具体可以参见之后的说明。

之后，为了提高算法的正确率和减少算法的运行时间，我采用了简单的特征选择算法，利用`sklearn.feature_selection`中的函数，只需要添加一行：

```
featureMNew = SelectKBest(chi2, k=20000).fit_transform(featureMatrix,
evalRes)
```

之后返回`featureMNew`即可。这样处理之后原来的数据集特征数降至20000，减少了`3/4`，只留下了相对重要的部分，减少了算法的运行时间，并且在这种情况下，使用朴素贝叶斯算法的分类平均正确率可以提高至86%左右，具体可以参见之后的说明。

然而，20000个特征对于SVM来说还是太多了，在这种情况下的算法运行时间仍然是完全不可接受的。因此，在实现`softsvm`的过程中，我使用了`gensim`来在给定的数据集上训练一个`word2vec`模型来得到词向量，然后将词向量的加权平均来作为`doc2vec`的结果，即给每一条评论产生一个单独的向量。我没有直接使用`doc2vec`模型来进行训练，因为训练集的评论数目太小，直接进行训练可能不会产生较好的结果。训练的文件在源代码`w2v.py`中已经给出，此处不再赘述。在训练`word2vec`模型的过程中，我将维度数指定为`150`，其他部分采用了`gensim`中`word2vec`的默认参数。另外，这里有一点需要注意：在训练的时候，为了去除没有意义的词，我采用了默认的训练选项将出现次数小于5次的词全部去除，因此在统计词频的时候需要进行特殊处理：

```
    featureMatrix = np.zeros([len(texts), 150])
    i = -1
    for line in texts:
        i = i + 1
        count = 0
        for word in line:
            try:
                featureMatrix[i, :] = featureMatrix[i, :] +
np.array(model[word])
            except KeyError:
                count += 1;
        featureMatrix[i, :] /= (len(line) - count)
```

利用`try`和`except`，记录不在模型中的词的数量，并且在之后的总数量中减去即可，这样即可得到每条评论所对应的正确的向量。另外，这里也可以使用上文中提及的特征提取方法来进一步减少特征的数量。

所有上文中所提及的函数都在文件`getFeature.py`中。为了方便，我没有将函数编写为`getFeature(comment)`的形式。

实现朴素贝叶斯分类器

```
def nBayesClassifier(trainData, trainLabel, testData, testLabel, threshold):
    trainData = trainData.astype(np.bool).astype(np.int32)
    trainLabelP = np.where(trainLabel == 1, 1, 0)
    trainLabelN = np.where(trainLabel == -1, 1, 0)

    pCount = np.sum(trainLabelP)
    nCount = np.sum(trainLabelN)

    trainResultP = sparse.csc_matrix(trainLabelP).dot(trainData).todense()
    trainResultP = trainResultP.astype(np.float)
    trainResultN = sparse.csc_matrix(trainLabelN).dot(trainData).todense()
    trainResultN = trainResultN.astype(np.float)
    trainResultAll = trainResultP + trainResultN
```

```
    trainResultAll = trainResultAll / (pCount + nCount)
    trainResultAll[np.isnan(trainResultAll)] = 0.5
    trainResultAll[np.isinf(trainResultAll)] = 0.5

    trainResultP = trainResultP / pCount
    trainResultP[np.isnan(trainResultP)] = 0.5
    trainResultP[np.isinf(trainResultP)] = 0.5

    trainResultN = trainResultN / nCount
    trainResultN[np.isnan(trainResultN)] = 0.5
    trainResultN[np.isinf(trainResultN)] = 0.5

    testData = testData.astype(np.bool).astype(np.int32)
    tempP = testData.multiply(trainResultP).tocsr()
    tempN = testData.multiply(trainResultN).tocsr()
    tempAll = testData.multiply(trainResultAll).tocsr()
    testResult = np.zeros([1, testData.get_shape()[0]])

    for i in range(testData.get_shape()[0]):
        tempM1 = tempP.getrow(i).toarray()
        tempM2 = tempN.getrow(i).toarray()
        tempM3 = tempAll.getrow(i).toarray()

        tempM1 = np.multiply(tempM1, 1 / tempM3)
        tempM2 = np.multiply(tempM2, 1 / tempM3)
        a = np.prod(tempM1[~np.isnan(tempM1)]) * (float(pCount) / (pCount +
nCount))
        b = np.prod(tempM2[~np.isnan(tempM2)]) * (float(nCount) / (pCount +
nCount))
        testResult[0, i] = a / (a + b)

    testResult = np.asarray(np.where(testResult > threshold, 1,
-1)).astype(np.int32)
    accu = np.sum(np.where(testResult == testLabel, 1, 0)) /
float(testLabel.shape[0])

    return testResult, accu
```

朴素贝叶斯分类器的实现我没有使用`sclearn`中的相关库，而是自己编写的代码。我第一次使用numpy编写朴素贝叶斯分类器的时候，大部分代码都没有向量化，算法中充斥着大量的for循环，这也直接导致了算法的代码运行效率极其低下，之后我重写了整个代码，也就是上面的这个版本。在这一部分代码中，仍然有没有完全向量化的部分，即其中的一个for循环，算法的性能瓶颈也主要在这里。

为了将算法向量化来提高速度，我使用了很多奇怪的技巧，例如说稀疏矩阵没法用`np.where`，直接用`nonzero`加上if将矩阵二值化也不够快，所以我使用类型转换的方式来将矩阵二值化：

```
trainData = trainData.astype(np.bool).astype(np.int32)
```

另外，还有几点需要注意，例如为了保证结果的精度，需要在将分子和分母连乘之前先进行除法来保证结果的精度，否则在朴素贝叶斯公式中，分子项的连乘会导致结果过于小从而使精度不能满足要求。除此之外，在朴素贝叶斯公式的使用中还要特别注意处理零的情况，以及在除法操作中要特别注意类型的转换问题。

上述算法在使用5fold交叉验证得到的结果如下：

```
[[ 0.74    0.7316  0.7256  0.7437  0.731 ] 0.5
 [ 0.7381  0.7288  0.7236  0.7415  0.7269] 0.6
 [ 0.734   0.7265  0.7209  0.738   0.7225] 0.7
 [ 0.7309  0.7247  0.719   0.7359  0.72  ] 0.75
 [ 0.7288  0.7221  0.716   0.7341  0.7185] 0.8
 [ 0.7269  0.7193  0.7114  0.7319  0.7156] 0.85
 [ 0.724   0.7135  0.7085  0.7273  0.7112]] 0.9
```

每一行从左至右分别为使用数据集中第`i*10000~i*10000+10000`条数据作为验证集，剩下的数据作为训练集得到的正确率；每一行的右侧则标明了后验概率所使用的阈值。总而言之，随着后验概率阈值的提升，算法的正确率是在不断下降的。上述的数据直接使用原始数据进行训练和测试而没有使用特征选择，最高正确率在74%左右。

之后我加入了特征提取将特征降至20000维，重新进行了测试，得到的结果如下：

```
[[ 0.871   0.8675  0.8647  0.8675  0.8618] 0.5
 [ 0.8664  0.8652  0.8635  0.8628  0.8594] 0.6
 [ 0.8623  0.8627  0.8608  0.8583  0.8562] 0.7
 [ 0.8588  0.8618  0.8593  0.8546  0.8535] 0.75
 [ 0.856   0.8587  0.857   0.8504  0.8502] 0.8
 [ 0.8513  0.8553  0.8525  0.8459  0.845 ] 0.85
 [ 0.8453  0.8507  0.8459  0.8416  0.8375]] 0.9
```

数据的格式和上文相同。可以看到，随着后验概率的提升，算法的正确率仍然在不断下降；加入了特征选择之后，算法的最高正确率提高到了将近87%。选择0.5作为阈值较为合适。

实现最小二乘分类器

由于助教明确说明可以使用`sclearn`中的相关库，因此这一部分的代码编写要相对简单：

```
def lsClassifier(trainData, trainLabel, testData, testLabel, lambdaS):
    reg = linear_model.Ridge(alpha=lambdaS)
    reg.fit(trainData, trainLabel.tolist())

    W = reg.coef_
    testResult = np.array(testData.dot(W))
    testResult = np.where(testResult > 0, 1, -1).astype(np.int32)
    accu = np.sum(np.where(testResult == testLabel, 1, 0)) /
float(testLabel.shape[0])

    return testResult, accu
```

计算验证结果的代码和计算正确率的代码和上文相似。这里，`fit`函数可以直接接受稀疏矩阵作为参数，它针对稀疏矩阵也有特殊的计算方法（和密集矩阵有所不同）。由于如何求解二次规划问题并不是本次试验的重点，因此这里不再赘述。

由于完整地跑完一组验证数据非常耗费时间，因此这里我只在进行了特征抽取的情况下运行了算法，得到的结果如下：

```
[[ 0.8877  0.8942  0.8913  0.8905  0.8888] 0.0001
 [ 0.8877  0.8942  0.8913  0.8906  0.8887] 0.01
 [ 0.8876  0.8942  0.8912  0.8908  0.8889] 0.1
 [ 0.8878  0.8945  0.891   0.8905  0.8889] 0.5
 [ 0.8881  0.8944  0.8912  0.8904  0.889 ] 1
 [ 0.888   0.8942  0.8913  0.8906  0.8891] 5
 [ 0.8881  0.8947  0.8913  0.8914  0.8887] 10
 [ 0.8877  0.8962  0.8919  0.893   0.8882] 100
 [ 0.8843  0.8881  0.8882  0.8901  0.883 ] 1000
 [ 0.8678  0.8688  0.8679  0.8719  0.866 ] 5000
 [ 0.8522  0.8539  0.8539  0.8553  0.852 ]] 10000
```

数据的格式和上文相似，每一行从左至右分别为使用数据集中第`i*10000~i*10000+10000`条数据作为验证集，剩下的数据作为训练集得到的正确率；每一行的右侧标明了所使用的`lambda`的值。可以看到，在`lambda`值较大的时候，算法的正确率会有明显的下降；选择合适的`lambda`值可以提高算法的正确率。从测试数据来看，选择5左右的值作为`lambda`值较为合适，在此时算法的正确率大约为89%。

实现支持向量机分类器

同样地，支持向量机也可以使用`sclearn`中的相关库：

```
def softsvm(trainData, trainLabel, testData, testLabel, sigma, Ci):
    if sigma == 0:
        clf = SVC(C=Ci, kernel='linear')
    else:
        clf = SVC(C=Ci, kernel='rbf', gamma=sigma)
    clf.fit(trainData, trainLabel)

    testResult = np.array(clf.predict(testData))
    testResult = np.where(testResult > 0, 1, -1).astype(np.int32)
    accu = np.sum(np.where(testResult == testLabel, 1, 0)) /
float(testLabel.shape[0])

    return testResult, accu
```

根据要求，依据传入的`sigma`值的不同，将选择不同的核函数。这一部分的数据集处理方式我在上文中有所提及，即使用`word2vec`来得到词向量，加权平均之后来得到每一条评论对应的向量。因此，这一部分的数据集`trainData`和`testData`不再是稀疏矩阵，而是150列的密集矩阵。

另外，为了得到实验文档中提及的d值，我是用了下列的方式来对数据进行采样，估计合适的d值：

```
shapeM = featureM.shape
dSum = 0
for i in range(shapeM[0] / 1000):
    for j in range(shapeM[0] / 1000):
        temp = featureM[i, :] - featureM[j, :]
        dSum += np.sum(temp * temp)
d = dSum / ((shapeM[0] / 1000) * (shapeM[0] / 1000))
sigmaL = [0, 0.01 * d, 0.1 * d, d, 10 * d, 100 * d]
```

即简单地取前50组数据来计算d值，否则计算所有数据的时间开销是完全不可接受的。

由于SVM的运行时间比线性分类器更长，在我的实验设备上运行完一组数据（1fold，1sigma，1C）大概需要2~5分钟，按照文档中的要求（5fold，5sigma，4C）运行完所有实验数据大概需要10个小时左右，这个时间实在是太长了；因此，我只运行了部分数据来作为验证。实验得到的数据如下：

```
0 1 0 0.8252
0 1 1 0.8437
0 1 2 0.7681
0 1 3 0.8106
0 1 4 0.8435
0.0829217483721 1 0 0.8286
0.0829217483721 1 1 0.8517
0.0829217483721 1 2 0.776
0.0829217483721 1 3 0.8177
0.0829217483721 1 4 0.8509
0.829217483721 1 0 0.8063
0.829217483721 1 1 0.8333
0.829217483721 1 2 0.7357
0.829217483721 1 3 0.7658
0.829217483721 1 4 0.8459
...
0 10 0 0.8246
0 10 1 0.8443
0 10 2 0.7699
0 10 3 0.8114
0 10 4 0.8433
0.0829217483721 10 0 0.8342
0.0829217483721 10 1 0.8538
0.0829217483721 10 2 0.7855
0.0829217483721 10 3 0.8225
0.0829217483721 10 4 0.855
0.829217483721 10 0 0.8047
0.829217483721 10 1 0.8278
0.829217483721 10 2 0.7265
0.829217483721 10 3 0.7689
0.829217483721 10 4 0.834
```

上述的数据中，第一列为sigma值，第二列为C值，第三列为fold数，最后一列为正确率。每个fold的正确率相差较大，可能是因为这里没有像之前一样对输入数据进行shuffle的原因，随机化之后效果应该能更好。从上述的数据中可以看到，在使用线性核函数的情况下，适当地提高C是可以提高算法的正确率的；而在使用rbf核函数的情况下，C的选择产生的影响依赖于sigma值。从上述的数据中可以看出，在算法的参数值选择合适的情况下，算法的正确率大致在82%左右，证明使用`word2vec`加权向量和的方式进行特征抽取是有效的。

在不同数据集上使用交叉验证选择各个算法的参数

以最小二乘分类器为例，交叉验证的代码如下：

```
lambdaS = [0.0001, 0.01, 0.1, 0.5, 1, 5, 10, 100, 1000, 5000, 10000]
shapeM = featureM.get_shape()
sliceL = len(evalRes) / 5
lsResult = np.zeros([len(lambdaS), 5])
for index in range(len(lambdaS)):
    for i in range(5):
        trainData = sparse.vstack([featureM[0:i*sliceL, :],
featureM[(i+1)*sliceL:shapeM[0], :]])
        trainLabel = evalRes[0:i*sliceL] + evalRes[(i+1)*sliceL:shapeM[0]]
        testData = featureM[i*sliceL:(i+1)*sliceL, :]
        testLabel = evalRes[i*sliceL:(i+1)*sliceL]
        testResult, accu = lsClassifier(trainData, np.array(trainLabel),
testData, np.array(testLabel), lambdaS[index])
        lsResult[index, i] = accu
        print lambdaS[index], i, accu

print lsResult
```

最终输出的结果矩阵`lsResult`已经在上文中展示过，计算每行的平均值并且比较大小，即可得到合适的参数。每次运行相应的算法时，主函数会从原始数据拼接训练集和验证集，并且导入函数来完成对应算法的调用。剩余算法的交叉验证代码具体可以参见`main.py`，此处不再赘述。

Part2 基于k-means聚类算法的图片色彩分割

在合理利用`numpy`的情况下，这一部分的代码也相当简洁，一共只有40行左右。图片的读入及处理我使用了pillow（PIL）。

算法的代码如下：

```
def nearestDistance(pointMat, clusterCenters):
    disResult = np.mat(np.zeros([np.shape(pointMat)[0],
np.shape(clusterCenters)[0]]))
    for i in range(np.shape(clusterCenters)[0]):
        temp = pointMat - np.matlib.repmat(clusterCenters[i, :],
np.shape(pointMat)[0], 1)
        disResult[:, i] = np.sum(np.multiply(temp, temp), axis=1)
    disMinIndex = np.mat(np.argmin(disResult, axis=1))
    return disMinIndex

def initializeClusterCenters(pointMat, k):
    clusterCenters = np.zeros([k, 3])
    for i in range(k):
        clusterCenters[i, :] = pointMat[np.random.randint(0,
np.shape(pointMat)[0]), :]

    return clusterCenters

def kmeans(pointMat, k, clusterCenters):
    subCenter = np.mat(np.zeros([np.shape(pointMat)[0], 1], dtype=np.int32))
    for count in range(50):
        disMinIndex = nearestDistance(pointMat, clusterCenters)
        subCenter = disMinIndex
        for i in range(k):
            clusterCenters[i, :] = np.mean(pointMat[np.where(subCenter == i)
[0], :], axis=0)

    return subCenter, clusterCenters

k = 3
fp = open("sea.JPG", "rb")
image = im.open(fp)
imageData = np.mat(image.getdata())
clusterCenters = initializeClusterCenters(imageData, k)
subCenter, clusterCenters = kmeans(imageData, k, clusterCenters)

clusterCenters = clusterCenters.astype(np.int32)

m, n = image.size
draw = ImageDraw.Draw(image)
for i in range(m*n):
    draw.point((i % m, i / m), tuple(clusterCenters[int(subCenter[i])]))
del draw

image.show()
```

同样地，在实现算法的时候我尽可能地避免去使用for循环，而尽量使算法尽可能地向量化。代码中的两个主要的for循环长度是聚类中心的数量，在一般情况下这个值都不是很大，不会过于影响算法的效率。

算法中有三个函数：`initializeClusterCenters`，`nearestDistance`和`kmeans`。`initializeClusterCenters`从图片中随机选取k个点，将他们的色彩值作为聚类的中心；`nearestDistance`则负责计算图片中的每一个点离哪一个聚类点最近，并返回一个`disMinIndex`，它记录了图片中每一个点对应的距离最近的聚类点在`clusterCenters`中的索引值。`kmeans`则是主函数，它主要负责调用`nearestDistance`和`means`函数来重新计算聚类中心。得到真正的聚类中心之后，我使用了`draw.point`来改变原图片中每一个点的RGB值，最后将图像显示出来。

这里有一点需要提及，我原本通过比较`subCenter`和`disMinIndex`是否相同来判断算法是否收敛，但最后我发现由于这两个数组过于长（长度为图片中的像素数），这一步比较所用的时间甚至比`kmeans`算法其余部分所用时间的总和都要长。利用聚类中心是否变化（即`clusterCenters`）来判断算法是否收敛又会导致需要使用一些玄学参数，因此最终我索性通过控制循环次数来作为`kmeans`算法的终止条件。一般而言，在聚类中心不多的情况下（10个以内），50次循环已经足够算法收敛到合适的值。

运行程序得到的结果如下：

原图：

k=2:

k=3:

k=5：

k=10：

k=20：

可以看到，在k较小的时候，可以清晰地看出图片中的颜色种类数，验证了算法的正确性；在k较大的时候，算法生成的图片上虽然还能够看到明显的颜色区块，但是已经基本上能够还原原图中的大部分细节。