金融大数据处理技术 Project2 实验报告

151278034 王铄

程序运行和实验结果说明和分析

一、设计思路（步骤）

1. 训练集和数据集的分词

训练集：遍历三个文件夹中的每个文件。

```
/*遍历路径中文件夹内的每个文件*/
public void iteratorPath(String dir) {
    stock = new File(dir);
    files = stock.listFiles();
    if (files != null) {
        for (File file : files) {
            if (file.isFile()) {
                pathName.add(file.getName());
            }else if (file.isDirectory()) {
                iteratorPath(file.getAbsolutePath());
            }
        }
    }
}
```

由于基本都是标题与正文内容，故直接进行无关字符的过滤以及分词。下图是无关字符过滤的部分：

```
String titles = StringUtils.strip(tempseg.toString().replaceAll("[,.%/(A-Za-z0-9)]","")),"[]");
```

测试集：遍历每个 txt 文件，用数组存储每行的内容、切片，提取出标题所在的部分并分词。

```
data = new String[6];
for (int i = 0; i < list.size(); i++) {
    data = list.get(i).split("  ");
    if(data.length!=5||!data[4].contains("http")||data[3].contains("http")||data[3].isEmpty()) {
        continue;
    }
```

具体的分词做法与 Project1 相同，在这里不做重复讨论。结果输出至原路径，以供后续调用。

【此部分代码见 segmentation.java】

2. 计算训练集与数据集的 tfidf 值

$$\mathrm{tf_{i,j}} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$ 以上式子中分子是该词在文件中的出现次数，而分母则是在文件中所有字词的出现次数之和。

$$\mathrm{idf_i} = \log \frac{|D|}{|\{j : t_i \in d_j\}|}$$ |D|：语料库中的文件总数；分母为包含该词语的文件数目。如果该词语不在语料库中，就会导致分母为

零，因此一般情况下使用下式作为分母：

$$1+|\{d \in D : t \in d\}|$$

这一部分的实现代码如下（分母加一及取对数）：

```
for (int i = 0; i < docNum; i++) {
    HashMap<String, Float> temp = all_tf.get(FileList.get(i));
    Iterator iter = temp.entrySet().iterator();
    while (iter.hasNext()) {
        Map.Entry entry = (Map.Entry) iter.next();
        String word = entry.getKey().toString();
        if (dict.get(word) == null) {
            dict.put(word, 1);
        } else {
            dict.put(word, dict.get(word) + 1);
        }
    }
}
System.out.println("IDF for every word is:");
Iterator iter_dict = dict.entrySet().iterator();
while (iter_dict.hasNext()) {
    Map.Entry entry = (Map.Entry) iter_dict.next();
    float value = (float) Math.log(docNum / Float.parseFloat(entry.getValue().toString()));
    resIdf.put(entry.getKey().toString(), value);
    //这里输入的是key值和value值,每个词对应的idf
    System.out.println(entry.getKey().toString() + " == " + value);
}
return resIdf;
```

之后用 $\mathrm{tfidf}_{i,j} = \mathrm{tf}_{i,j} \times \mathrm{idf}_i$ 得到每个词的 tfidf 值。

对于训练集：计算每个文件中的每个词在全部三个文件夹（pos，neu，neg）中的次数并结合总词数进行计算。

对于测试集：计算每个标题中的词汇在所有标题中出现的词数并结合总词数进行计算。

输出数据以 “词+tfidf 值”的格式储存，为向量化做好准备。

【此部分代码见 Tfidf.java】

3. 加工向量化的标准词库(特征选择)

这里的目的是筛除对文本分类影响不大或无关的词语，保留影响较大的词语。

▶利用了 project1 中的 chi_words.txt。但是此处词汇较多，导致向量的维数比较大。而且有些专有名词等词汇实际上是无用的，所以运用了一些方法将多余的特征词删去了。具体操作是用 excel 打开 chi_words.txt，之后筛选带有一些特殊字的词语如“一”，“二”、“百”、“千”等带有数字的词，“东”、“西”、“南”、“北”等带有方位的词（可能是地名）等并分别删去。经过一些类似的处理后，留下了 1000 个自认为比较“有用”的词汇，作为特征词汇。即，之后的向量化操作是基于这 1000 个特征词汇来的，生成的向量都是 1000 维向量。

【此部分生成文档见 chi_words.txt（保留了原来的文件名）】

4. 依据词库进行向量化操作

依据 chi_words 的词汇，这里先用名为 dic 的 list 记录特征词：

```
    //用list记录模型的词
    String READ_PATH = ("/home/wangshuo/pj2/chi_words.txt");
    ArrayList<String> dic = new ArrayList<String>();
    File file = new File(READ_PATH);
    BufferedReader br = new BufferedReader(new FileReader(file));
    String t;
    while ((t = br.readLine()) != null) {
        dic.add(t);
    }
    br.close();
```

之后在输入路径的文件中对各个词进行搜索，若某个词在 dic 中，则将 tfidf 值赋在相应的位置，生成向量。在生成一个向量后立即输出至结果文件之后进行下一个文件的读取。

```
    File file1 = new File(INPUT_PATH);
    for (int i = 0; i < vector.dimension; i++) {
        vector[i] = "0";
    }
    BufferedReader br1 = new BufferedReader(new FileReader(file1));
    BufferedWriter bw1 = new BufferedWriter(new FileWriter(OUT_PATH,
    String temp;
    String[] temp1 = new String[2];
    int k = 0;
    while ((temp = br1.readLine()) != null) {
        temp1 = temp.split(" ");
        k = dic.indexOf(temp1[0]);
        if (k != -1) {
            vector[k] = temp1[1];
        }
    }
    br1.close();
}
```

对于训练集的每个文件夹以及测试集所在文件夹做相似的操作。之后将各个元素的值整体乘以相同的倍数并取整，方便后续观察和计算。需要补充的是，在对训练集文件夹操作时，还需要在每行向量输出完成后再把表示情感的标签值也输入进去并附在每行末尾。

【此部分代码见 vectorization.java；向量输出结果见 training_vectors.txt 以及 testing_vectors.txt】

5. Knn 算法设计和实现

利用第 4 步得到的结果，读取两个由向量组成的 txt 文件。

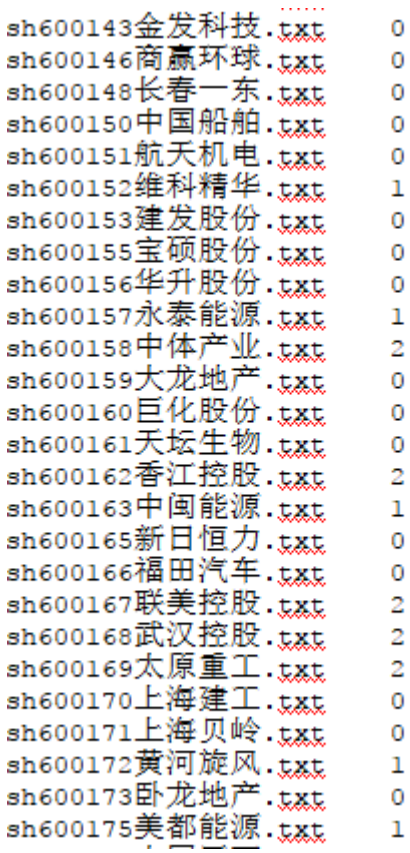
Mapper:针对测试集中的单个向量（行号作为 Key），计算其与训练集中各个向量的欧氏距离。将此距离与训练集对应向量的 label（type）值传给 Reducer。

Reducer:对于每个 mapper 传来的所有 distance 和 type 进行整合，并针对欧氏距离进行排序。

```
public static class KNNReducer extends
        Reducer<Text, Text, Text, IntWritable> {

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        List<DistanceType> list = new ArrayList<>();
        for (Text value : values) {
            list.add(new DistanceType(value.toString()));
        }
        Collections.sort(list);
```

因为这里设定 KNN 中的 K 值为 3，即寻找到欧氏距离最短的 3 个点，再对其所对应的 type 进行考查。三个点进行等权投票，计算出每个向量的 KNN 结果（归属于哪一类）并输出文件名和类型号。此处 0 表示 Positive,1 表示 neutral，2 表示 negative。结果部分截图如下：



sh600143金发科技.txt	0
sh600146商赢环球.txt	0
sh600148长春一东.txt	0
sh600150中国船舶.txt	0
sh600151航天机电.txt	0
sh600152维科精华.txt	1
sh600153建发股份.txt	0
sh600155宝硕股份.txt	0
sh600156华升股份.txt	0
sh600157永泰能源.txt	1
sh600158中体产业.txt	2
sh600159大龙地产.txt	0
sh600160巨化股份.txt	0
sh600161天坛生物.txt	0
sh600162香江控股.txt	2
sh600163中闽能源.txt	1
sh600165新日恒力.txt	0
sh600166福田汽车.txt	0
sh600167联美控股.txt	2
sh600168武汉控股.txt	2
sh600169太原重工.txt	2
sh600170上海建工.txt	0
sh600171上海贝岭.txt	0
sh600172黄河旋风.txt	1
sh600173卧龙地产.txt	0
sh600175美都能源.txt	1

【此部分代码见 KNNMapreduce.java；预测结果见 knn_test_result.txt】

6. SVM 算法设计和实现

① 生成正确的输入格式（稀疏矩阵的输入）

SVM 对于训练集和数据集的输入格式是有规定的。即训练集要按照“Label position:value position:value .....”进行输入的。每个 Position 对应于此向量中数值不为 0 的元素所在位置，每行代表一个向量。由此来输出稀疏矩阵。

但是在之前的处理中，训练集和数据集是以完整的矩阵的形式输出的。需要对此做一个转化。为了简单起见，在这里写了两个 C 程序，分

别将训练集和数据集作为输入数据，输出所得的稀疏矩阵格式。以训练集为例，C 程序代码如下：

```c
#include<stdio.h>
int main(){
    int a[500][1001];
    int i,j;
    FILE *fp;

    for(i=0; i<500; i++){
        for(j=0; j<1001; j++){
            scanf("%d",&a[i][j]);
        }
    }

    if((fp=fopen("svm_train_matrix3.txt","w"))==NULL) {
        printf("File cannot be opened/n");
        exit(1);
    }

    else{
        for(i=0; i<500; i++){
            fprintf(fp,"%d ", a[i][1000]);
            for(j=0; j<1000; j++){
                if(a[i][j]!=0) {
                //  printf("%d:%d ", j-1, a[i][j]);
                    fprintf(fp,"%d:%d ", j+1, a[i][j]);
                }
            }
            fprintf(fp,"\n");
        }
    }
    fclose(fp);
    return 0;
}
```

这里一开始将数据以文件形式读取，在 Devcpp 下运行会出现报错，后来改为将文本复制后粘贴读取，也会使程序崩溃。后来发现是因为开放的数组规模太大，超过了内存限制。所以这里每次读取 500 行（程序不崩溃的上限为 518），进行标准化的输出，省略了所有值为 0 的部分。

这里也是进行了反复操作。因为训练集数据共有 1500 行，而测试集有 3200 多行。所以一共要重复跑 10 次 C 程序，并且要将输出文件合并。生成的标准化结果如下：

```
3 26:473 102:250 167:620
3 66:123 206:26 403:75 789:105 880:132 907:88
3 26:423 31:639 102:223 317:533 631:335
1 1:497 3:382 4:352 256:829 284:532 911:1276
3 17:78 20:50 27:107 28:120 34:96 59:340 66:159 94:110 98:96 104:88 107:116 108:116 112:78 119
1 13:461 72:829 182:1247 415:1276 483:844 632:1467 637:923
1 17:210 20:136 27:289 28:324 34:260 59:183 66:429 104:119 112:210 129:252 139:284 198:291 206
2 97:593 730:1932 761:2096
2 11:64 22:509 23:52 74:74 80:95 82:107 104:97 105:680 118:80 128:64 135:108 145:430 185:102 1
2 11:133 23:54 74:308 86:65 88:88 118:83 128:67 130:118 290:93 311:102 316:125 348:60 453:78 5
2 1:354 3:273 97:190 167:366 206:314 260:358 270:161 865:200
2 16:687 391:1483 396:1183 575:1292
2 30:356 102:197 104:184 138:251 522:386 626:366 767:325
2 12:211 17:206 20:133 28:317 34:254 59:179 66:630 73:231 112:206 119:158 129:123 140:140 198:
3 13:282 18:259 63:1851 203:1365 206:539 209:203 226:630 479:488 567:733
2 26:656 36:739 102:346 136:625 257:401
2 97:1908
1 18:370 63:879 203:1298 267:929 715:1213
2 19:158 21:259 23:283 30:127 48:193 70:271 71:259 76:314 105:152 122:142 129:69 164:73 166:39
3 74:573 129:399 209:162 345:502 448:591 927:697
2 209:425 347:754
2 16:776 20:213 97:258 129:197 170:513 206:141 209:480 257:231 423:183
3 3:71 4:32 13:28 20:27 23:12 25:28 33:53 35:17 36:27 38:19 61:21 80:46 88:20 96:21 99:16 112:
1 13:161 37:619 63:706 75:326 203:1042 206:102 237:388 431:382 694:506 784:420 796:975 800:471
2 8:167 22:76 23:27 40:64 92:71 95:67 104:25 105:268 118:42 122:55 140:30 145:181 167:34 186:3
3 346:1641 545:1393 882:464
```

【此部分代码见 train_matrix.c 和 test_matrix.c；生成的结果见 svm_train_matrix.txt 和 svm_test_matrix.txt】

② 生成训练结果

由于生成训练模型的过程极其复杂，

这里使用了课件中提供的网址里面的 jar 包和 Java 程序（LIBSVM）。自己编写了 java 程序，调用 LIBSVM 内置函数，将 svm_train_matrix.txt 中的训练集稀疏矩阵进行训练，并输出训练模型。模型的部分截图如下：

```
svm_type c_svc
kernel_type rbf
gamma 0.001
nr_class 3
total_sv 995
rho -0.18497475517009188 0.04955303729668811 0.22034295230821285
label 3 1 2
nr_sv 344 289 362
SV
0.8146619864722087 1.0 26:473.0 102:250.0 167:620.0
0.8147011062159081 1.0 66:123.0 206:26.0 403:75.0 789:105.0 880:132.0 907:88.0
0.8147011062159379 1.0 26:423.0 31:639.0 102:223.0 317:533.0 631:335.0
0.0 0.049804684127423116 17:78.0 20:50.0 27:107.0 28:120.0 34:96.0 59:340.0 66:159.0 94:110.0
0.8147309053319681 1.0 13:282.0 18:259.0 63:1851.0 203:1365.0 206:539.0 209:203.0 226:630.0 4
0.814739865354346 1.0 74:573.0 129:399.0 209:162.0 345:502.0 448:591.0 927:697.0
0.814770104389853 1.0 3:71.0 4:32.0 13:28.0 20:27.0 23:12.0 25:28.0 33:53.0 35:17.0 36:27.0 3
0.814770104389853 1.0 346:1641.0 545:1393.0 882:464.0
0.8149269945669924 1.0 23:813.0 26:766.0 80:733.0 102:404.0 140:907.0 410:705.0 768:2953.0
0.8149269945669924 1.0 4:99.0 11:142.0 23:116.0 102:116.0 105:125.0 140:130.0 207:145.0 209:9
0.8149269945669924 1.0 102:435.0 119:552.0 297:1462.0 767:718.0 863:994.0 880:785.0
0.8149269945669924 1.0 5:158.0 26:216.0 28:144.0 52:293.0 74:161.0 102:114.0 129:112.0 206:80
1.0 1.0 5:23.0 6:23.0 16:21.0 22:23.0 35:45.0 38:25.0 67:34.0 74:24.0 86:61.0 90:25.0 99:44.0
0.8149269945669924 1.0 23:74.0 118:113.0 189:132.0 346:185.0 348:81.0 470:178.0 750:133.0 882
0.8149269945669924 1.0 209:225.0 270:153.0 534:715.0 789:604.0
0.8149269945669924 1.0 52:420.0 102:163.0 147:373.0 550:355.0 920:491.0
0.8149269945669924 1.0 23:161.0 99:209.0 104:149.0 118:246.0 164:167.0 424:337.0 435:440.0 55
0.8149411049960753 1.0 243:335.0 525:574.0 568:281.0
0.8149411049960753 1.0 129:381.0 280:731.0 383:1268.0
0.8149411049960753 1.0 404:177.0 453:178.0
0.8149411049960753 1.0 13:172.0 90:236.0 264:279.0 271:278.0 423:141.0
```

由图可见，训练结果生成了 995 个支持向量。类型为 1，2，3 的支持向量分别有 289，362，344 个。

【此部分代码见 svm.java；生成的结果见 model.txt】

③ 利用测试集和模型进行预测

▶ 假设 A,B,C 三类。在训练的时候我选择 A,B； A,C； B,C 所对应的向量作

为训练集，然后得到 3 个训练结果，在测试的时候，把对应的向量分别对 3 个结果进行测试，然后采取投票形式，最后得到一组结果。

- 投票是这样的：

  A=B=C=0;

  (A,B)-classifier 如果是 A win,则 A=A+1;otherwise,B=B+1;

  (A,C)-classifier 如果是 A win,则 A=A+1;otherwise, C=C+1;

  (B,C)-classifier 如果是 C win,则 C=C+1;otherwise,B=B+1;

  The decision is the Max(A,B,C)

Mapper: 计算每个测试向量在三个分类下分别归于哪一类。

Reducer: 读取每个 Key 在三个分类下的结果并进行投票。

Reducer 部分代码如下：

```
public void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {
    double[] vote=new double[3];
    double f;
    //多个分类器投票
  for (Text val : values) {
      String[] v=val.toString().split(":");
      f=Double.parseDouble(v[1]);
      String[] label=v[0].split("v");
      if(f>0){
          vote[Integer.parseInt(label[0])-1]+=f;
      }else{
          vote[Integer.parseInt(label[1])-1]+=-f;
      }
  }

  if(getMax(vote)==0){
      result.set("Positive");
  }else if(getMax(vote)==1){
      result.set("Negative");
  }else{
      result.set("Neutral");
  }
  context.write(key, result);
```

部分结果的截图：

100	test0100	Negative
101	test0101	Negative
102	test0102	Positive
103	test0103	Neutral
104	test0104	Negative
105	test0105	Negative
106	test0106	Negative
107	test0107	Negative
108	test0108	Positive
109	test0109	Positive
110	test0110	Negative
111	test0111	Positive

112	test0112	Negative
113	test0113	Neutral
114	test0114	Positive
115	test0115	Positive
116	test0116	Neutral
117	test0117	Negative
118	test0118	Neutral
119	test0119	Negative
120	test0120	Negative
121	test0121	Negative
122	test0122	Positive
123	test0123	Neutral
124	test0124	Positive
125	test0125	Negative
126	test0126	Negative

【结果文档见 svm_result.txt】

二、性能、扩展性等方面存在的不足和可能的改进之处

1. 特征选择时没有用到 tfidf 值，是一开始考虑到不同的文本中会有相同的词出现，排序结果会出现多个重复词汇。所以利用之前的 chi_words 进行认为挑选，会有一些主观的因素，影响向量化的结果和训练的准确性。
2. 预测结果未能实现准确度的考查。
3. 在由向量化结果向 SVM 所需的标准化的稀疏矩阵转化时虽然使用了 C 语言作为辅助工具，但执行效率上还是不如 mapreduce 来得快，不能很大批量地处理数据。（上限是 518*1001 的 Int 型数组）
4. 程序的串行化程度低，需要反复调整数据格式，降低了效率。

三、提交文件说明

1. segmentation.java，用于测试集和数据集的数据清洗和分词。
2. Tfidf.java，用于计算训练集和测试集的 tfidf 值。
3. chi_words.txt，是处理过的特征文本。
4. vectorization.java，用于生成训练集和测试集的向量化结果。
5. training_vectors.txt 和 testing_vectors.txt，是训练集和测试集的向量化结果。
6. KNNMapreduce.java，是 knn 的算法程序。
7. knn_test_result.txt，是 Knn 的执行输出结果。
8. train_matrix.c 和 test_matrix.c，是用于生成标准化的稀疏矩阵的 C 程序。
9. svm_train_matrix.txt 和 svm_test_matrix.txt，是训练集和测试集标准化的稀疏矩阵的输出结果。
10. svm_result.txt，svm 的执行结果。