中文文本分类

北京邮电大学

数据仓库与数据挖掘大作业

王浩宇
2017140491
计算机技术
49 组

一、 实验要求

搜集至少 100 万份中文文本，且至少 10 类。使用朴素贝叶斯和另外一种分类方法进行中文文本分类。

其中，训练集和测试集之比为 1：1。

二、 数据收集

本次实验的文本数据是基于 Scrapy 框架，自行编写的爬虫程序，在中国新闻网的滚动新闻页面中抓取新闻文本。

爬虫程序的文件结构如图 2-1 所示。

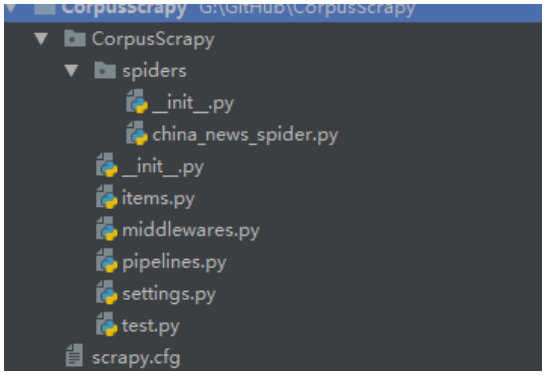

图 2-1 爬虫的程序文件结构

基于中国新闻网的滚动新闻页面的 url 均为“http://www.chinanews.com/scroll-news/yyyy/mmdd/news.shtml”结构，因此，可以通过构造不同的“yyyy/mmdd/”组合来抓取不同的历史新闻。url 的构造函数如图 2-2 和图 2-3 所示。

```
16        scroll_domain = 'http://www.chinanews.com/scroll-news/'
```

图 2-2 滚动新闻页面的主域名

```
28    def url_generator(self, date):
29        end_date = datetime.date.today()
30        cur_date = datetime.date(date[0], date[1], date[2])
31
32        while cur_date <= end_date:
33            temp = str(cur_date).split('-')
34            url = '%s/%s/%s%s/news.shtml' % (self.scroll_domain,
35                                             temp[0], temp[1], temp[2])
36            yield url
37            cur_date += datetime.timedelta(1)
```

图 2-3 滚动新闻的 url 的构造函数

然后，对于每个构造好的合法 url，访问并解析索引页（图 2-4），获取新闻标签和具体新闻的 url（图 2-4），抓取相应的新闻文本（图 2-5），按照新闻的类别标签分别保存到本地。（图 2-5 与图 2-6）

```python
def parse(self, response):
    new_xpath_query = '//*[@id="content_right"]/div[3]/ul/li/div[2]/a/@href'
    title_xpath_query = '//*[@id="content_right"]/div[3]/ul/li/div[1]/a/text()'

    link_list = response.xpath(new_xpath_query).extract()
    class_list = response.xpath(title_xpath_query).extract()

    for news_url, article_class in zip(link_list, class_list):
        url = response.urljoin(news_url)
        yield scrapy.Request(url=url,
                             callback=self.page_parse, meta={'article_class': article_class})
        #使用Request的meta来传递额外的参数
```

图 2-4 每日新闻索引页面的解析函数

```python
def page_parse(self, response):
    content_query = '//p/text()'
    article_class = response.meta['article_class']
    article_class = re.sub(r'\s', '', article_class)

    if article_class not in self.title:
        if not article_class or len(article_class.strip()) <= 0:
            article_class = '其他'
        self.title[article_class] = 0

    self.title[article_class] += 1

    content = ''.join(response.xpath(content_query).extract())
    content = re.sub(r'\s', '', content)
    title = '%s-%d.txt' % (article_class, self.title[article_class])
    title = re.sub(r'\s', '', title)

    item = NewsItem()
    item['title'] = title
    item['content'] = content
    item['article_class'] = article_class

    yield item
```

图 2-5 具体的新闻文本的获取函数

```python
class NewsPipeline(object):
    def create_dir(self, path):
        if not os.path.exists(path):
            os.mkdir(path)

    def process_item(self, item, spider):
        if item['content'] and len(item['content'].strip()) > 0:
            cur_path = os.path.join('./data', item['article_class'])
            self.create_dir(cur_path)
            with open(os.path.join(cur_path, item['title']), 'wb') as f:
                f.write(item['content'].encode('utf-8'))
#           line = json.dumps(dict(item)) + '\n'
#           self.file.write(line)
        return item

    def open_spider(self, spider):
        self.create_dir('data')
```

图 2-6 按类别标签保存新闻文本

三、 数据处理

对于已经获取的原始新闻文本，需要进行预处理才可以进行具体的分类训练和测试工作。具体的预处理过程大体可分为三步：

1、 中文分词，保留名词词语并去除停用词。

本次实验使用的是结巴分词工具，对原始的新闻文本信息进行词性标注和分词，只保留名词的同时去除停用词。

2、 利用处理之后的词语，使用词袋模型表示文本。

3、 使用卡方检验进行特征选择，以防止维度爆炸影响准确度。由于原始新闻文本的词典大小为 62 万，注意到很多词语出现的频率非常低，因此数据稀疏性很大，所以可以使用稀疏矩阵来表示原始语料（图 3-1），以节省内存的使用。同时，将类别和词语映射为数字，方便进行后续的表示和计算工作。

```
55  def generate_SparseMatrix(corpus_data):
56      w2i = dict() #词语序号映射
57      i2w = dict()
58      c2i = dict() #类别序号映射
59      i2c = dict()
60      m, n = len(corpus_data['dict']), len(corpus_data['class'])
61      wf_matrix = lil_matrix((m, n))
62      df_matrix = lil_matrix((m, n))
63      index_word = 0
64      index_class = 0
65      for c in corpus_data['class']:
66          i2c[index_class] = c #映射类别序号
67          c2i[c] = index_class
68          index_class += 1
69
70      for w, cp in corpus_data['occurs'].items():
71          i2w[index_word] = w #映射词语序号
72          w2i[w] = index_word
73          for class_name in cp:
74              i, j = index_word, c2i[class_name]
75              if w in corpus_data['word_frequency'][class_name]:
76                  wf_matrix[i, j] = corpus_data['word_frequency'][class_name][w]
77              if w in corpus_data['doc_frequency'][class_name]:
78                  df_matrix[i, j] = corpus_data['doc_frequency'][class_name][w]
79          index_word += 1
80
81      corpus_data['w2i'] = w2i
82      corpus_data['i2w'] = i2w
83      corpus_data['c2i'] = c2i
84      corpus_data['i2c'] = i2c
85      corpus_data['wf_matrix'] = wf_matrix
86      corpus_data['df_matrix'] = df_matrix
87      return corpus_data
```

图 3-1 构建原始语料库的稀疏矩阵

其中，图 3-1 中的函数构建了两个矩阵：词频矩阵 wf 和文档频率矩阵 df。矩阵每一行表示的是不同的词语，每一列表示的是不同的类别；元素$wf_{i,j}$表示词语 i 在类别 j 中出现的频率，元素$df_{i,j}$表示包含词语 i 并且属于类别 j 的文档数目。

这样，通过构建好的稀疏矩阵以及映射表，就可以通过公式 3-1，进行卡方检

验的计算了（图 3-2）。

$$\chi^2(t,c) = \frac{N \times (AD - CB)^2}{(A+C) \times (B+D) \times (A+B) \times (C+D)} \quad 3\text{-}1$$

其中，公式 3-1 的各个参数的含义如下：

N：训练数据集文档总数；

A：包含词条 t，同时属于类别 c 的文档的数量；

B：包含词条 t，但是不属于类别 c 的文档的数量；

C：属于类别 c，但是不包含词条 t 的文档的数量；

D：不属于类别 c，同时也不包含词条 t 的文档的数量。

由于做出原假设是“词条 t 与类别 c 不相关”，因此选择的过程也变成了为每个词语 t 计算它与类别 c 的卡方值，然后从大到小排个序，此时卡方值越大越相关。最后，对每个类别选取前 1000 个特征词语，最后将所有类别的特征词语去重合并（图 3-3），便得到了整个语料库的特征词典，字典的大小为 8200 个特征词语。

```
89  def chiSquared_Test(df_matrix):
90      m, n = df_matrix.shape
91      csr_df = df_matrix.tocsr()
92      N = csr_df.sum()
93      chi_matrix = lil_matrix((m, n))
94      row_sum = []
95      for i in range(m):
96          row_sum.append(csr_df[i, :].sum())
97      for j in range(n):
98          col_sum = csr_df[:, j].sum()
99          for i in range(m):
00              A = csr_df[i, j]
01              B = row_sum[i] - A
02              C = col_sum - A
03              D = N - A - B - C
04              chi_matrix[i, j] = (N * (A * D - C * B) ** 2) / ((A + C) * (B + D) * (A + B) * (C + D))
05              print('%s:  (%d, %d) has done.' % (str(time.asctime(time.localtime())), i, j))
06      return chi_matrix
```

图 3-2 卡方检验的计算

```
117 def test_chiSquared():
118     cur_path = os.path.abspath('.')
119     model_path = os.path.join(cur_path, 'model')
120     chi_matrix = mmread(os.path.join(model_path, 'chi.mat.mtx'))
121     chi_matrix = chi_matrix.tocsr()
122     m, n = chi_matrix.shape
123     maxK = 1000
124     feature_list = []
125     for j in range(n):
126         class_list = []
127         for i in range(m):
128             class_list.append((i, chi_matrix[i, j]))
129         class_list = sorted(class_list, key=lambda item: item[1])
130         feature_list.append(class_list[-maxK:])
131 
132     feature_pool = set()
133     for word_list in feature_list:
134         for x in word_list:
135             feature_pool.add(x[0])
136 
137     with open(os.path.join(model_path, 'feature.pool'), 'wb') as f:
138         for x in feature_pool:
139             f.write(('%s\n' % x).encode('utf-8'))
140 
141     return feature_pool
142 
```

图 3-3 按照卡方检验的值对每个词语排序并选取合并

四、 朴素贝叶斯分类器

朴素贝叶斯分类器的原理如公式 4-1 所示，很简洁明了。

$$v_{NB} = argmax_{a_i \in A}\{P(a_i)P(b_1|a_i)P(b_2|a_i)\dots P(b_n|a_i)\} \quad 4\text{-}1$$

为了防止乘法造成的数值过小问题，对公式 4-1 进行了改进，得到了形如公式 4-2 所示的公式。

$$v_{NB} = argmax_{a_i \in A}\left\{log_{10}\left(P(a_i)\right) + \sum_{b_i \in B} log_{10}\left(P(b_i|a_i)\right)\right\}$$

$$= argmax_{a_i \in A}\left\{P_{log_{10}}(a_i) + \sum_{b_i \in B} P_{log_{10}}(b_i|a_i)\right\} \quad 4\text{-}2$$

本次实验的贝叶斯分类器大体分为两个步骤：

第一步，通过预处理的文本数据，训练得到在类别a_i中出现特征b_j的统计频率以及类别a_i的统计频率。（图 4-1）

特别的，对于零概率问题，使用拉普拉斯平滑（加一平滑）技术进行处理。整个计算过程如公式 4-3 所示。

$$P_{log_{10}}(b_i|a_i) = \frac{c(a_i,b_i)+1}{c(a_i)+|V|} \quad 4\text{-}3$$

```
72  def train_NB(path):
73  #     tfidf_matrix = load_tfidf(path).tocsr()
74      wf_mat = load_wfmat(path).tocsr()
75      df_mat = load_dfmat(path).tocsr()
76      feature_string, feature_index = load_feature(path)
77
78      m, n = wf_mat.shape
79      abs_V = wf_mat.sum()
80      abs_D = df_mat.sum()
81      pwNB_mat = [dict() for x in range(n)]
82      pcNB_mat = [0 for x in range(n)]
83      for j in range(n):
84          #计算P(c)
85          pcNB_mat[j] = math.log10(df_mat[: , j].sum() / abs_D)
86          #计算P(w|c)
87          col_sum = wf_mat[feature_index, j].sum()
88          for i in feature_index:
89              pwNB_mat[j][i] = math.log10((wf_mat[i, j] + 1) / (col_sum + abs_V))
90
91      pickle.dump(pwNB_mat, open(os.path.join(path, 'pwNB.model'), 'wb'), True)
92      pickle.dump(pcNB_mat, open(os.path.join(path, 'pcNB.model'), 'wb'), True)
93
```

图 4-1 训练朴素贝叶斯模型

第二步，对于每个输入的文本数据，根据 4-2 计算得到该文本在所有类别下的最大似然度，并将该类别作为输出值输出，并统计混淆矩阵（图 4-3）。最后计算模型的准确度、查全率（召回率）以及 F 测度（图 4-4）。

```
70      confuse_mat = [[0 for x in range(n)] for x in range(n)]
71      data = pickle.load(open(data_path, 'rb'))
72      print('%s: begin to calculate...' % time.asctime())
73      for class_index, doc_list in data.items():
74          print('%s: %s is beginning..' % (time.asctime(), i2c[class_index]))
75          for doc in doc_list:
76              maxV = float('-inf')
77              best = None
78              for j in range(n):
79                  cur_p = pcNB[j]
80                  for w, freq in doc.items():
81                      if w in w2i:
82                          i = w2i[w]
83                          if i in pwNB[j]:
84                              cur_p += pwNB[j][i] * freq
85                  if cur_p > maxV:
86                      maxV = cur_p
87                      best = j
88              if class_index == best:
89                  correct_doc_cnt += 1
90              confuse_mat[class_index][best] += 1
91              doc_total += 1
```

图 4-2 测试朴素贝叶斯模型的程序代码

分类器在 50 万文本的测试集的测试表现良好，准确度可以达到 87.7%。

	文化	体育	证券	娱乐	汽车	军事	法治	I T	教育	房产
文化	42556	1060	262	2684	353	2476	746	733	3108	771
体育	555	83086	215	875	345	328	184	325	330	191
证券	374	440	48503	552	838	376	681	2619	164	2083
娱乐	5227	1464	239	76369	217	327	316	543	469	182
汽车	122	212	258	181	48316	203	341	442	94	149
军事	757	301	137	181	130	30541	196	163	110	98
法治	1453	927	539	1591	2064	1009	50666	1639	2969	2255
I T	1590	822	2101	819	1322	807	1347	57904	725	977
教育	485	237	38	133	92	124	249	161	15247	128
房产	645	374	696	287	729	146	1298	415	555	31787

图 4-3 混淆矩阵

```
0 文化:
Precision =  0.777293    Recall  =  0.791533   F-score  =  0.784348
1 体育:
Precision =  0.961265    Recall  =  0.934359   F-score  =  0.947621
2 证券:
Precision =  0.856489    Recall  =  0.915358   F-score  =  0.884946
3 娱乐:
Precision =  0.894743    Recall  =  0.912719   F-score  =  0.903641
4 汽车:
Precision =  0.960213    Recall  =  0.888064   F-score  =  0.922730
5 军事:
Precision =  0.936438    Recall  =  0.840493   F-score  =  0.885875
6 法治:
Precision =  0.778136    Recall  =  0.904362   F-score  =  0.836514
7 IT:
Precision =  0.846376    Recall  =  0.891599   F-score  =  0.868399
8 教育:
Precision =  0.902510    Recall  =  0.641412   F-score  =  0.749883
9 房产:
Precision =  0.860690    Recall  =  0.823050   F-score  =  0.841449
avg:
Precision =  0.877415    Recall  =  0.854295   F-score  =  0.862541
```

4-4 准确度、召回率和 F 测度

五、 SVM 分类器

本次实验使用 libsvm 工具包进行 svm 分类器的设计与实现。

为了提高 SVM 的性能，采用 TFIDF 进行词向量的构建（图 5-1）。TFIDF 公式如公式 5-1 所示。

$$\mathrm{TF}-\mathrm{IDF}(t,c)=\frac{c(w,c)}{N(c)}\times log_2\frac{c(w,d)}{N(D)} \qquad 5\text{-}1$$

```
def calc_tfidf(path):
    wf_matrix = load_corpusData(path).tocsr()
    df_matrix = load_dfmat(path).tocsr()
    feature_index = load_feature(path)

    m, n = wf_matrix.shape
    tfidf_matrix = [dict() for x in range(n)]
    abs_D = df_matrix.sum()
    df_sum = dict()
    for i in feature_index:
        df_sum[i] = df_matrix[i, :].sum()
    for j in range(n):
        col_sum = wf_matrix[feature_index, j].sum()
        for i in feature_index:
            TF = wf_matrix[i, j] / col_sum
            IDF = math.log2(abs_D / (df_sum[i]))
            tfidf_matrix[j][i] = TF * IDF

    pickle.dump(tfidf_matrix, open(os.path.join(r'.\data', 'tfidf.train.list'), 'wb'), True)
    return tfidf_matrix
```

图 5-1 计算 TFIDF

然后，为了满足 libsvm 的数据格式规范，需要将数据格式化（图 5-2）。

```
    wi2di = dict()
    max_di = -1
    with open(save_path, 'wb') as f:
        line_cnt = 0
        for class_index, doc_list in data.items():
            for doc in doc_list:
                if line_cnt % 3000 == 0:
                    print('%s: %d lines has done...' % (time.asctime(), line_cnt))
                temp_list = []
                di_list = []
                for w, wf in doc.items():
                    if w not in w2i:
                        continue
                    i = w2i[w]
                    if i in feature_index:
                        if i not in wi2di:
                            max_di += 1
                            wi2di[i] = max_di
#                       for k in range(wf):
                        temp_list.append(tfidf_mat[class_index][i])
                        di_list.append(wi2di[i])
                if len(temp_list) < 10:
                    continue
                temp_list = z_score(temp_list)
                if np.float64('nan') == temp_list[0]:
                    continue
                pair_list = zip(di_list, temp_list)
                pair_list = sorted(pair_list, key=lambda item:item[0])
                output_line = '%d ' % class_index
                for pair in pair_list:
                    output_line += '%d:%f ' % (pair[0], pair[1])
                output_line += '\n'
                f.write(output_line.encode('utf-8'))
                line_cnt += 1
```

图 5-2 数据格式化

整个模型的训练过程，只需要使用 libsvm 自带的工具包中编译好的 svm-train 工具，输指定的核函数以及参数的值，便可以完成训练。

特别的，由于完整的训练数据是 500000*8200 维的矩阵，完整的训练一次模型需要耗时十几个小时。由于时间有限，鉴于 svm 只需要找到足够的支撑向量便可以完成训练的特点，本次实验的 svm 模型的训练使用了对训练数据进行随机抽样的训练方式。

在 RBF 核的训练中，通过对 C=[1, 5, 10]和 gamma=[0.1, 0.01, 0.001]这几组参数的尝

试，选择了准确率最高的[C=10, gamma=0.001]的参数组合，作为 RBF 核的模型参数，得到的准确率为 96.4%；在使用线性核函数训练模型时，测试了 C=[1, 5, 10]这几个参数，选取了准确率最高的参数 C=10，准确率为 97.5%。对比两个核函数的性能，选择了线性核函数作为最终模型。

最后，统计了模型在 50 万文本测试集中的混淆矩阵（5-3），以及相应的准确度、召回率以及 F 测度（图 5-4）。

	文化	体育	证券	娱乐	汽车	军事	法治	I T	教育	房产
文化	45636	600	138	1056	188	294	277	244	270	236
体育	323	76841	137	412	157	225	108	117	157	113
证券	38	21	46795	63	104	23	59	255	14	223
娱乐	946	498	46	68221	101	140	235	178	115	50
汽车	43	29	81	23	47875	70	45	157	21	54
军事	202	147	46	54	98	31058	50	72	70	37
法治	141	144	38	166	76	135	49413	91	203	143
I T	35	16	142	58	243	28	48	57056	17	71
教育	76	34	15	48	18	19	55	21	20509	32
房产	75	45	143	30	143	31	63	65	55	33831

图 5-3 混淆矩阵

```
0 文化:
Precision =  0.932508    Recall  =  0.960455   F-score  =  0.946275
1 体育:
Precision =  0.977745    Recall  =  0.980427   F-score  =  0.979085
2 证券:
Precision =  0.983192    Recall  =  0.983481   F-score  =  0.983336
3 娱乐:
Precision =  0.967262    Recall  =  0.972765   F-score  =  0.970006
4 汽车:
Precision =  0.989194    Recall  =  0.976981   F-score  =  0.983049
5 军事:
Precision =  0.975624    Recall  =  0.969865   F-score  =  0.972736
6 法治:
Precision =  0.977507    Recall  =  0.981332   F-score  =  0.979416
7 IT:
Precision =  0.988599    Recall  =  0.979401   F-score  =  0.983979
8 教育:
Precision =  0.984731    Recall  =  0.956978   F-score  =  0.970656
9 房产:
Precision =  0.981149    Recall  =  0.972435   F-score  =  0.976772
avg:
Precision =  0.975751    Recall  =  0.973412   F-score  =  0.974531
```

图 5-4 准确度、召回率、F 测度

六、 实验的不足与反思

在验收过程中，老师指出了特征词典的构建过程中，应该再去除人名和单字，这样特征的维度可以再降低很多，可以进一步提升分类器的整体性能。