实验一　文本数据的分类与分析

【实验目的】

1. 掌握数据预处理的方法，对训练集数据进行预处理；
2. 掌握文本建模的方法，对语料库的文档进行建模；
3. 掌握分类算法的原理，基于有监督的机器学习方法，训练文本分类器；
4. 利用学习的文本分类器，对未知文本进行分类判别；
5. 掌握评价分类器性能的评估方法。

【实验类型】

数据挖掘算法的设计与编程实现。

【实验要求】

1. 文本类别数：>=10 类；
2. 训练集文档数：>=50000 篇；每类平均 5000 篇。
3. 测试集文档数：>=50000 篇；每类平均 5000 篇。
4. 分组完成实验，组员数量<=3，个人实现可以获得实验加分。

【实验内容】

利用分类算法实现对文本的数据挖掘，主要包括：

1. 语料库的构建，主要包括利用爬虫收集 Web 文档等；

2. 语料库的数据预处理，包括文档建模，如去噪，分词，建立数据字典，使用词袋模型或主题模型表达文档等；

注：使用主题模型，如 LDA 可以获得实验加分；

3. 选择分类算法（朴素贝叶斯（必做）、SVM/其他等），训练文本分类器，理解所选的分类算法的建模原理、实现过程和相关参数的含义；

4. 对测试集的文本进行分类

5. 对测试集的分类结果利用正确率和召回率进行分析评价：计算每类正确率、召回率，计算总体正确率和召回率。

【实验分析与总结】

1. 数据获取

从清华新闻语料库中获取 10 类每类一万文本数据，共总计 10 万数据。其中，一万数据对半分为训练集和测试集，将其拷贝至项目资源文件中。这么做的原因是能够利用贝叶斯的极大似然估计。

```
for class_name, class_name_en in class_list.items():
    dir_path = 'D:/下载/THUCNews/THUCNews/' + class_name
    file_list = os.listdir(dir_path)
    print(class_name + ':' + str(len(file_list)))

    if not os.path.exists('source_data_train/' + class_name_en):
        os.mkdir('source_data_train/' + class_name_en)
    for i in range(5000):
        print(i)
        shutil.copy(dir_path + '/' + file_list[i], 'source_data_train/' + class_name_en + '/' + str(i) + '.txt')

    if not os.path.exists('source_data_test/' + class_name_en):
        os.mkdir('source_data_test/' + class_name_en)
    for i in range(5000, 10000):
        print(i)
        shutil.copy(dir_path + '/' + file_list[i], 'source_data_test/' + class_name_en + '/' + str(i - 5000) + '.txt')
```

图 1 数据获取

```
1  老债券基金申赎平稳
2      □本报记者 易非 深圳报道
3      “从节后的情况来看，老的债券基金规模变化比较平稳，净赎回不明显。而新成立的债券基金净赎回接近一成。”某基金公司市场人士告诉中国证券报记者。
4      债券基金是2008年基金市场上最大赢家，这一年央行多次降息打开了债券市场上涨的空间，国债指数、企债指数分别上涨了9.4%、17.11%，债券型基金也
5      在这种业绩的刺激下，债券型基金规模急剧膨胀，去年四季度，债券型基金净申购比例为21.88%。但细究起来，债券型基金内部也是苦乐不均，其中仅投
6      债券基金受热捧的情况正有所改变。今年1月初国债、企债指数均出现大幅回调，始于2007年末的债券牛市行情面临高位调整压力，债券型基金除少部分持
7      “在基金市场中，高风险类基金品种与低风险类基金品种增里占比具有一定的跷跷板效应。”债券基金投资人士表示。虽然债券市场没有以前热了，但他并不
8
9
```

图 2 数据示例

2. 数据预处理

利用 jeiba 库，将数据文本分为仅有名词的干净文本。其中，除了利用 jieba 库去除了专有名词、人名等无用名词外，还根据老师所给的停用词表删除了停用词。不仅仅如此，停用词操作也同时考虑到了名词和符号拼接的特殊情况。

```
def isStopWord(word_str: str):
    if word_str in stop_words:
        return True
    for i in word_str:
        if i in stop_sign:
            return True
    return False
```

图 3 判断停用词

```python
def creatTexts():
    for class_name, class_name_en in class_list.items():
        # 生成文件夹目录
        if not os.path.exists('data_test/' + class_name_en):
            os.mkdir('data_test/' + class_name_en)
        if not os.path.exists('data_train/' + class_name_en):
            os.mkdir('data_train/' + class_name_en)
        for i in range(5000):
            print(class_name + ':' + str(i))
            # 生成测试集
            string_to_write = ''
            with open('source_data_test/' + class_name_en + '/' + str(i) + '.txt', 'r', encoding='utf-8') as f:
                lines = f.read()
                # print(lines)
                words = ps.cut(lines, use_paddle=True)
                for word, flag in words:
                    # print('%s %s' % (word, flag))
                    # 若为名词且不在停用词表中，则加入写入串
                    if flag == 'n' and not isStopWord(word):
                        string_to_write += (word + ' ')
            with open('data_test/' + class_name_en + '/' + str(i) + '.txt', 'w', encoding='utf-8') as f:
                f.write(string_to_write)
            # 生成训练集
            string_to_write = ''
            with open('source_data_train/' + class_name_en + '/' + str(i) + '.txt', 'r', encoding='utf-8') as f:
                lines = f.read()
                # print(lines)
                words = ps.cut(lines, use_paddle=True)
                for word, flag in words:
                    # print('%s %s' % (word, flag))
```

图 4 分词并生成干净文本

```
1  债券 基金 本报 记者 情况 债券 基金 规模 债券 基金公司 市场 人士 证券 记者 证券 记者 基金公司 情况 债券 基金 整体 情况 资金 股票 方向 基
```

图 5 干净文本示例

随后，将处理后的文件在每个类下整合为一个 all.txt 文件，以避免重复 IO

```python
    all_data_test = ''
    all_data_train = ''
    for class_name, class_name_en in class_list.items():
        string_to_write_test = ''
        string_to_write_train = ''
        for i in range(5000):
            print(class_name + ':' + str(i))
            with open('data_test/' + class_name_en + '/' + str(i) + '.txt', 'r', encoding='utf-8') as f:
                string_to_write_test += (f.read() + '\n')
            with open('data_train/' + class_name_en + '/' + str(i) + '.txt', 'r', encoding='utf-8') as f:
                string_to_write_train += (f.read() + '\n')
        with open('data_test/' + class_name_en + '/all.txt', 'w', encoding='utf-8') as f:
            f.write(string_to_write_test)
        with open('data_train/' + class_name_en + '/all.txt', 'w', encoding='utf-8') as f:
            f.write(string_to_write_train)
        all_data_test += string_to_write_test
        all_data_train += string_to_write_train
    with open('data_test/all.txt', 'w', encoding='utf-8') as f:
        f.write(all_data_test)
    with open('data_train/all.txt', 'w', encoding='utf-8') as f:
        f.write(all_data_train)
```

图 6 数据整合

3. 建立词典

词典的生成即特征提取部分，按如下方案执行：对于每一个类的训练集，先利用 jieba 内置的 TF/IDF 方法加权，得到前 1000 个关键词(得到 1000*2 的一个矩阵)，对于每个词，其权值再与卡方检验值相乘得到**综合权值**，每个类排序后的前 500 个词为该类的关键特征词。

首先需要将训练集中整合的干净文档转为 Python 的 Counter 数据结构方便后续处理：

```
def init():
    for CLASS_NAME_EN in class_list.values():
        with open('data_train/' + CLASS_NAME_EN + '/all.txt', 'r', encoding='utf-8') as file:
            contents[CLASS_NAME_EN] = ''.join(file.readlines())
            # 实际上，基于Counter数据结构的word_counter就是词频统计结果
            word_counters[CLASS_NAME_EN] = Counter(contents[CLASS_NAME_EN].split())
        if not os.path.exists('pkls/' + CLASS_NAME_EN):
            os.mkdir('pkls/' + CLASS_NAME_EN)
        with open('pkls/' + CLASS_NAME_EN + '/TF.pkl', 'wb') as file:
            pickle.dump(dict(word_counters[CLASS_NAME_EN]), file)
```

图 7 文件数据初始化

利用 TF/IDF 与卡方值处理文本代码如下，利用 Counter 数据结构取得每类训练集中综合权值最大的前 500 个词

```
init()
# 先利用jieba内置的TF/IDF方法加权，得到前1000个关键词(得到1000*2的一个矩阵)，其权值再与卡方检验值相乘，每个类排序后的前500个词为关键词
for class_name_en in class_list.values():
    topK = 1000
    if len(word_counters[class_name_en]) < 1000:
        topK = len(word_counters[class_name_en])
    tags = jieba.analyse.extract_tags(contents[class_name_en], topK=topK,
                                      withWeight=True)
    tags_dic = dict(tags)
    index = 0
    for tag, value in tags_dic.items():
        print("class: %s index: %d tag: %s\t\t weight: %f" % (class_name_en, index, tag, value))
        index += 1
        tags_dic[tag] = value * chi_square(tag, class_name_en)
        # tags_dic[tag] = value
    key_words = Counter(tags_dic).most_common(500)
    with open('data_train/' + class_name_en + '/dict.txt', 'w', encoding='utf-8') as f:
        for i in key_words:
            f.write(i[0] + ':' + str(i[1]) + '\n')
```

图 8 每类的字典生成

对于卡方检验，有如下定义：

假设欲得到词 W 和类 C 的关系，设置如下参数

Parm_A: 包含 W 且属于 C 的文档数

Parm_B: 包含 W 但不属于 C 的文档数

Parm_C: 不包含 W 却属于 C 的文档数

Parm_D: 既不包含 W 也不属于 C 的文档数

则有：

表格 1 卡方检验表

特征选择	属于 C	不属于 C	总计
包含 W	A	B	A+B
不包含 W	C	D	C+D
总数	A+C	B+D	N

因此卡方值为

$$\chi = \frac{\left(\mathrm{parm_A} + \mathrm{parm_B} + \mathrm{parm_C} + \mathrm{parm_D}\right) \times \left(\mathrm{parm_A} * \mathrm{parm_D} - \mathrm{parm_B} * \mathrm{parm_C}\right)^2}{\left(\mathrm{parm_A} + \mathrm{parm_C}\right) \times \left(\mathrm{parm_A} + \mathrm{parm_B}\right) \times \left(\mathrm{parm_D} + \mathrm{parm_B}\right) \times \left(\mathrm{parm_C} + \mathrm{parm_D}\right)}$$

求卡方值函数代码编写如下：

```
def chi_square(word: str, class_str: str):
    parm_A = 0
    parm_B = 0
    parm_C = 0
    parm_D = 0
    for CLASS_NAME_EN in class_list.values():
        with open('data_train/' + CLASS_NAME_EN + '/all.txt', 'r', encoding='utf-8') as file:
            lines = file.readlines()
            # all文件中，一行代表一个文件
            if CLASS_NAME_EN == class_str:
                for j in lines:
                    if word in j.split():
                        parm_A += 1
                    else:
                        parm_C += 1
            else:
                for j in lines:
                    if word in j.split():
                        parm_B += 1
                    else:
                        parm_D += 1
    # print('chisq-statistic=%.4f, p-value=%.4f, df=%i expected_frep=%s' % kf)
    # 返回卡方值，卡方值越大说明相关性越高，该词在该类中越关键
    if parm_B == 0:
        return 1
    return (parm_A + parm_B + parm_C + parm_D) * ((parm_A * parm_D - parm_B * parm_C) ** 2) / (
            (parm_A + parm_C) * (parm_A + parm_B) * (parm_D + parm_B) * (parm_C + parm_D))
```

图 9 求卡方值

生成总体关键词表，则需要将各个类的字典中的 500 个词进行整合去重，利用 Counter 数据结构即可方便的实现这个功能，随后将总体特征关键词表保存于本地

```
V = []
for class_name_en in class_list.values():
    with open('data_train/' + class_name_en + '/dict.txt', 'r', encoding='utf-8') as f:
        for line in f.readlines():
            V.append(line.split(':')[0])
V_dict = Counter(V)

with open('data_train/key_words.txt', 'w', encoding='utf-8') as f:
    f.write('\n'.join(V_dict.keys()))
with open('pkls/key_words.pkl', 'wb') as f:
    pickle.dump(V_dict, f)
```

图 10 生成所有文档的关键词表

得到关键词维度为 3685

4. 生成词向量

通过所有文档的关键词表，首先生成 50000*3685 的词向量矩阵，由于数据稀疏性很大，因此采用 scipy 库的稀疏矩阵进行存储，生成的词向量稀疏矩阵用二进制保存在本地

```
with open('pkls/key_words.pkl', 'rb') as f:
    key_words_dic = pickle.load(f)
key_words = list(key_words_dic.keys())
test_arr = np.zeros(shape=(50000, 3685))
train_arr = np.zeros(shape=(50000, 3685))
test_index = 0
train_index = 0
for class_name_en in class_list.values():
    with open('data_test/' + class_name_en + '/all.txt', 'r', encoding='utf-8') as f:
        content = f.readlines()
        for text in content:
            print(class_name_en + ':' + str(test_index))
            for w in text.split():
                if w not in key_words:
                    continue
                else:
                    index = key_words.index(w)
                    test_arr[test_index][index] += 1
            test_index += 1
    with open('data_train/' + class_name_en + '/all.txt', 'r', encoding='utf-8') as f:
        content = f.readlines()
        for text in content:
            print(class_name_en + ':' + str(train_index))
```

图 11 生成词向量

```
coo_test = coo_matrix(test_arr)
# print(coo_test)
save_npz('coo_test.npz', coo_test)
coo_train = coo_matrix(train_arr)
# print(coo_train)
save_npz('coo_train.npz', coo_train)
```

图 12 生成并保存稀疏矩阵

5. 贝叶斯分类器

贝叶斯分类器的主要公式如下所示：

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

其中a代表类别，b代表特征关键词

而在本实验中，测试集、训练集的每个类的数据量都是相同的（5000 个），即$P(a_i)$的值将不影响训练结果，因此可以利用贝叶斯公式的极大似然估计，公式可以简化为：

$$v_{NB} = argmax_{a_i \in A}\{P(b_1|a_i)P(b_2|a_i)\dots P(b_n|a_i)\} \quad (2)$$

贝叶斯分类器的训练过程关键在于构造每一类（10 类）的词频矩阵以便于 m-估计，由于在字典生成的过程中，干净文档转为 Counter 数据结构后实际上就生成了词频向量，因此通过读取先前存储的二进制文件即可较为容易的生成词频矩阵，随后，为了解决 0 概率问题，使用m-估计来估计每个词相对于10个类各自的条件概率，公式如下（TF 为词频）：

$$P(b_n|a_i) = \frac{TF(a_i, b_n) + 1}{TF(a_i) + |V|} \quad (3)$$

```python
def train_Bays():
    global word_df, key_words, df_sum, bayes_df

    word_df = pd.DataFrame(np.zeros((3685, 10)), columns=class_list.values(),
                           index=key_words)
    bayes_df = pd.DataFrame(np.zeros((3685, 10)), columns=class_list.values(),
                            index=key_words)

    # 构造关键词词频矩阵
    for CLASS_NAME_EN in class_list.values():
        with open('pkls/' + CLASS_NAME_EN + '/TF.pkl', 'rb') as F:
            TF_dic = pickle.load(F)
            for tup in word_df.itertuples():
                if TF_dic.get(tup[0]) is None:
                    continue
                else:
                    word_df.at[tup[0], CLASS_NAME_EN] = TF_dic.get(tup[0])
    df_sum = np.array(word_df).sum()
    word_df.to_csv('TF_Matrix.csv')
    with open('pkls/TF_Matrix.pkl', 'wb') as F:
        pickle.dump(word_df, F)

    # 构造条件概率矩阵
    for tup in bayes_df.itertuples():
        for CLASS_NAME_EN in class_list.values():
            # m-估计
            bayes_df.at[tup[0], CLASS_NAME_EN] = (word_df.at[tup[0], CLASS_NAME_EN] + 1) / (
                    word_df[CLASS_NAME_EN].sum() + df_sum)
    bayes_df.to_csv('Bayes.csv')
    with open('pkls/Bayes.pkl', 'wb') as F:
        pickle.dump(word_df, F)
```

图 13 训练贝叶斯分类器

训练数据集，则直接读取条件概率矩阵中的值，利用公式 2 对其概率相乘得到后验概率。排序 10 个类的后验概率，最高者即预测类。测试集采用词向量的稀疏矩阵进行读取，以加快测试速度。

```
def Bays(text_pos: int):
    global word_df, bayes_df
    v_NB = {}

    for CLASS_NAME_EN in class_list.values():
        v_NB[CLASS_NAME_EN] = 1
        for v in column_index[row_pointers[text_pos]:row_pointers[text_pos + 1]]:
            # print(np.array(df).sum())
            w = key_words[v]
            v_NB[CLASS_NAME_EN] *= bayes_df.at[w, CLASS_NAME_EN]

    # print(v_NB)
    res = sorted(v_NB.items(), key=lambda kv: (kv[1], kv[0]), reverse=True)
    return res[0][0]
```

图 14 利用贝叶斯分类器进行测试

训练与测试的主要过程则如下所示，结果将生成混淆矩阵：

```
if __name__ == '__main__':
    start = time.time()
    train_Bays()
    end = time.time()
    train_time = end - start

    confusion_matrix = pd.DataFrame(np.zeros((10, 10)), columns=class_list.values(),
                                    index=class_list.values())

    start = time.time()
    class_index = 0
    for class_name_en in class_list.values():
        for i in range(5000):
            # with open('data_test/' + class_name_en + '/' + str(i) + '.txt', 'r', encoding='utf-8') as f:
            #     content = f.read()
            s = Bays(class_index * 5000 + i)
            print('class:' + class_name_en + ' pre:' + s + ' id:' + str(i))
            # 横向为预测
            # 纵向为真实值
            confusion_matrix.at[class_name_en, s] += 1
        class_index += 1
    end = time.time()
    print('\n\nTrain time: %s Seconds' % train_time)
    print('Test time: %s Seconds' % (end - start))
    print(confusion_matrix)
    confusion_matrix.to_csv('Confusion_Matrix.csv')

    with open('pkls/Confusion_Matrix.pkl', 'wb') as file:
        pickle.dump(confusion_matrix, file)
```

图 15 贝叶斯分类器主要运行过程

得到的混淆矩阵如下所示：

表格 2 贝叶斯混淆矩阵

	Economics	House	Society	Fashion	Education	Technology	Politics	PE	Game	Entertainment
Economics	3990	490	62	9	65	327	19	0	0	38
House	68	4037	52	0	45	748	34	1	5	10
Society	43	236	3921	4	476	170	32	3	13	102
Fashion	6	102	5	4491	89	140	0	3	23	141
Education	28	108	63	13	4066	625	14	3	7	73
Technology	8	121	54	2	125	4609	26	1	21	33
Politics	57	721	198	0	257	310	3419	2	6	30
PE	0	19	16	1	42	47	7	4819	10	39
Game	14	118	11	4	71	239	5	19	4432	87
Entertainment	1	21	33	22	29	341	7	0	8	4538

评价数据如下：

```
----------财经----------
召回率：0.7980
精确率：0.9466
F1值 ：0.8660
----------房产----------
召回率：0.8074
精确率：0.6759
F1值 ：0.7358
----------社会----------
召回率：0.7842
精确率：0.8881
F1值 ：0.8329
----------时尚----------
召回率：0.8982
精确率：0.9879
F1值 ：0.9409
----------教育----------
召回率：0.8132
精确率：0.7723
F1值 ：0.7922
----------科技----------
召回率：0.9218
精确率：0.6100
F1值 ：0.7342
----------时政----------
召回率：0.6838
精确率：0.9596
F1值 ：0.7986
----------体育----------
召回率：0.9638
精确率：0.9934
F1值 ：0.9784
----------游戏----------
召回率：0.8864
精确率：0.9794
F1值 ：0.9306
----------娱乐----------
召回率：0.9076
精确率：0.8914
F1值 ：0.8994
平均召回率：0.8464
平均精确率：0.8705
```

图 16 贝叶斯分类器评价数据

6. SVM

调用 sklearn 的 SVM 分类模型，由于其接口支持 scipy 的稀疏矩阵，因此将词向量直接传入模型训练即可。经测试，在使用 rbf 核函数的前提下，参数 C 在 1-10 范围内，参数 gamma 在[10,1,0.1,0.01,0.001]范围内时，组合 C=6, gamma=0.001 的分类效果最好，运行代码如下：

```
coo_test = load_npz('coo_test.npz')
# print(coo_test)
coo_train = load_npz('coo_train.npz')
# print(coo_train)
class_arr = np.array([int(i / 5000) for i in range(50000)])

model = SVC(kernel='rbf', C=6, gamma=0.001)
start = time.time()
model.fit(coo_train.tocsr(), class_arr)
end = time.time()
print('Train time: %s Seconds' % (end - start))
start = time.time()
pre = model.predict(coo_test.tocsr())
end = time.time()
print('Test time: %s Seconds' % (end - start))
print(pre)
with open('pkls/svm_pre.pkl', 'wb') as f:
    pickle.dump(pre, f)
```

图 17SVM 主要代码

混淆矩阵如下：

表格 3SVM 混淆矩阵

	Economics	House	Society	Fashion	Education	Technology	Politics	PE	Game	Entertainment
Economics	4764	48	34	20	12	20	96	1	1	4
House	71	4615	65	41	26	10	160	2	4	6
Society	42	61	4590	31	53	50	134	6	18	15
Fashion	3	3	6	4893	21	7	31	3	3	30
Education	18	26	103	113	4419	60	231	3	16	11
Technology	10	10	53	44	22	4756	82	4	12	7
Politics	41	82	95	8	38	35	4688	1	9	3
PE	0	1	16	13	4	6	30	4915	6	9
Game	2	9	13	41	11	29	73	4	4801	17
Entertainment	2	5	30	117	9	15	39	3	17	4763

评价数据如下：

```
Train time: 151.20289421081543 Seconds
Test time: 233.29223203659058 Seconds
[0 0 0 ... 9 9 9]
混淆矩阵为：
 [[4764   48   34   20   12   20   96    1    1    4]
 [  71 4615   65   41   26   10  160    2    4    6]
 [  42   61 4590   31   53   50  134    6   18   15]
 [   3    3    6 4893   21    7   31    3    3   30]
 [  18   26  103  113 4419   60  231    3   16   11]
 [  10   10   53   44   22 4756   82    4   12    7]
 [  41   82   95    8   38   35 4688    1    9    3]
 [   0    1   16   13    4    6   30 4915    6    9]
 [   2    9   13   41   11   29   73    4 4801   17]
 [   2    5   30  117    9   15   39    3   17 4763]]
准确率为：
 0.94408
精确率为：
 [0.96184131 0.94958848 0.91708292 0.91956399 0.95752979 0.95348837
 0.84255931 0.99453662 0.98240229 0.97903392]
均值0.9458

召回率为：
 [0.9528 0.923  0.918  0.9786 0.8838 0.9512 0.9376 0.983  0.9602 0.9526]
均值0.9441

F1值为：
 [0.95729931 0.93610548 0.91754123 0.94816394 0.91918877 0.95234281
 0.8875426  0.98873466 0.97117427 0.96563609]
---------------分类报告---------------
              precision    recall  f1-score   support

           0       0.96      0.95      0.96      5000
           1       0.95      0.92      0.94      5000
           2       0.92      0.92      0.92      5000
           3       0.92      0.98      0.95      5000
           4       0.96      0.88      0.92      5000
           5       0.95      0.95      0.95      5000
           6       0.84      0.94      0.89      5000
           7       0.99      0.98      0.99      5000
           8       0.98      0.96      0.97      5000
           9       0.98      0.95      0.97      5000

    accuracy                           0.94     50000
   macro avg       0.95      0.94      0.94     50000
weighted avg       0.95      0.94      0.94     50000
```

图 18SVM 评价数据

7. 逻辑回归

调用 sklearn 的逻辑回归模型，其接口同样支持 scipy 的稀疏矩阵。经测试，在参数 C=0.01, max_iter=3000 时，逻辑回归模型的分类效果最好，运行代码如下：

```
coo_test = load_npz('coo_test.npz')
# print(coo_test)
coo_train = load_npz('coo_train.npz')
# print(coo_train)
class_arr = np.array([int(i / 5000) for i in range(50000)])

model = LogisticRegression(C=0.01, max_iter=3000)
start = time.time()
model.fit(coo_train, class_arr)
end = time.time()
print('Train time: %s Seconds' % (end - start))
start = time.time()
pre = model.predict(coo_test)
end = time.time()
print('Test time: %s Seconds' % (end - start))
with open('pkls/LR_pre.pkl', 'wb') as f:
    pickle.dump(pre, f)
```

图 19 逻辑回归主要代码

混淆矩阵如下：

表格 4 逻辑回归混淆矩阵

	Economics	House	Society	Fashion	Education	Technology	Politics	PE	Game	Entertainment
Economics	4594	93	65	34	27	41	132	3	6	5
House	101	4434	94	65	39	21	231	4	5	6
Society	45	56	4535	38	69	81	135	7	18	16
Fashion	5	5	8	4906	19	10	15	2	5	25
Education	32	21	133	143	4298	99	223	10	22	19
Technology	13	16	67	58	32	4674	103	6	20	11
Politics	46	72	104	21	40	48	4650	2	9	8
PE	0	1	20	25	3	8	18	4908	9	8
Game	4	4	17	66	20	34	43	6	4784	22
Entertainment	2	1	39	142	10	20	33	5	23	4725

评价数据如下：

```
Train time: 5.81696891784668 Seconds
Test time: 0.04600167274475098 Seconds
混淆矩阵为：
 [[4594   93   65   34   27   41  132    3    6    5]
 [ 101 4434   94   65   39   21  231    4    5    6]
 [  45   56 4535   38   69   81  135    7   18   16]
 [   5    5    8 4906   19   10   15    2    5   25]
 [  32   21  133  143 4298   99  223   10   22   19]
 [  13   16   67   58   32 4674  103    6   20   11]
 [  46   72  104   21   40   48 4650    2    9    8]
 [   0    1   20   25    3    8   18 4908    9    8]
 [   4    4   17   66   20   34   43    6 4784   22]
 [   2    1   39  142   10   20   33    5   23 4725]]
准确率为：
 0.93016
精确率为：
 [0.9487815  0.94280247 0.89236521 0.89232448 0.94316436 0.92811755
 0.83288555 0.9909146  0.97612732 0.9752322 ]
均值0.9323

召回率为：
 [0.9188 0.8868 0.907  0.9812 0.8596 0.9348 0.93   0.9816 0.9568 0.945 ]
均值0.9302

F1值为：
 [0.93355009 0.91394414 0.89962309 0.93465422 0.89944543 0.93144679
 0.87876784 0.98623531 0.96636703 0.95987811]
---------------分类报告---------------
              precision    recall  f1-score   support

           0       0.95      0.92      0.93      5000
           1       0.94      0.89      0.91      5000
           2       0.89      0.91      0.90      5000
           3       0.89      0.98      0.93      5000
           4       0.94      0.86      0.90      5000
           5       0.93      0.93      0.93      5000
           6       0.83      0.93      0.88      5000
           7       0.99      0.98      0.99      5000
           8       0.98      0.96      0.97      5000
           9       0.98      0.94      0.96      5000

    accuracy                           0.93     50000
   macro avg       0.93      0.93      0.93     50000
weighted avg       0.93      0.93      0.93     50000
```

图 20 逻辑回归评价数据

8. 实验总结与反思

本次实验中存在最大的问题是数据预处理和数据清洗部分。在实验结果中可以看到，时尚等类别的分类结果无论在何种分类方式下都不甚理想，推测其主要原因还是在于数据预处理时没有做好。例如在生成词典时，应当将所有词进行 TF/IDF 加权排序，然而由于性能和时间的原因，只能用较小的 1000 个样本实现加权排序。其次，TF/IDF 权值和卡方值直接相乘的效果有待商榷，若二者进行加权相乘，字典生成的结果可能更好。