

实验一：文本数据的分类与分析

- 实验目的
- 实验要求
- 实验内容
- 实验分工
- 实验环境
- 主要设计思想
 - TF-IDF 算法
 - 卡方检验
 - 朴素贝叶斯
 - 支持向量机
 - 性能评估方法
- 实验过程
 - 创建数据集
 - 数据预处理
 - 创建词典
 - 生成词向量
 - 朴素贝叶斯
 - 训练阶段
 - 测试阶段
 - 性能评估
 - 支持向量机
 - 训练和测试
 - 参数调整
 - 性能评估
 - 评估函数
- 总结和反思

实验目的

1. 掌握数据预处理的方法, 对训练集进行;
2. 掌握文本建模的方法, 对语料库档进行;
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估方法.

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/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, 以及 F-score.

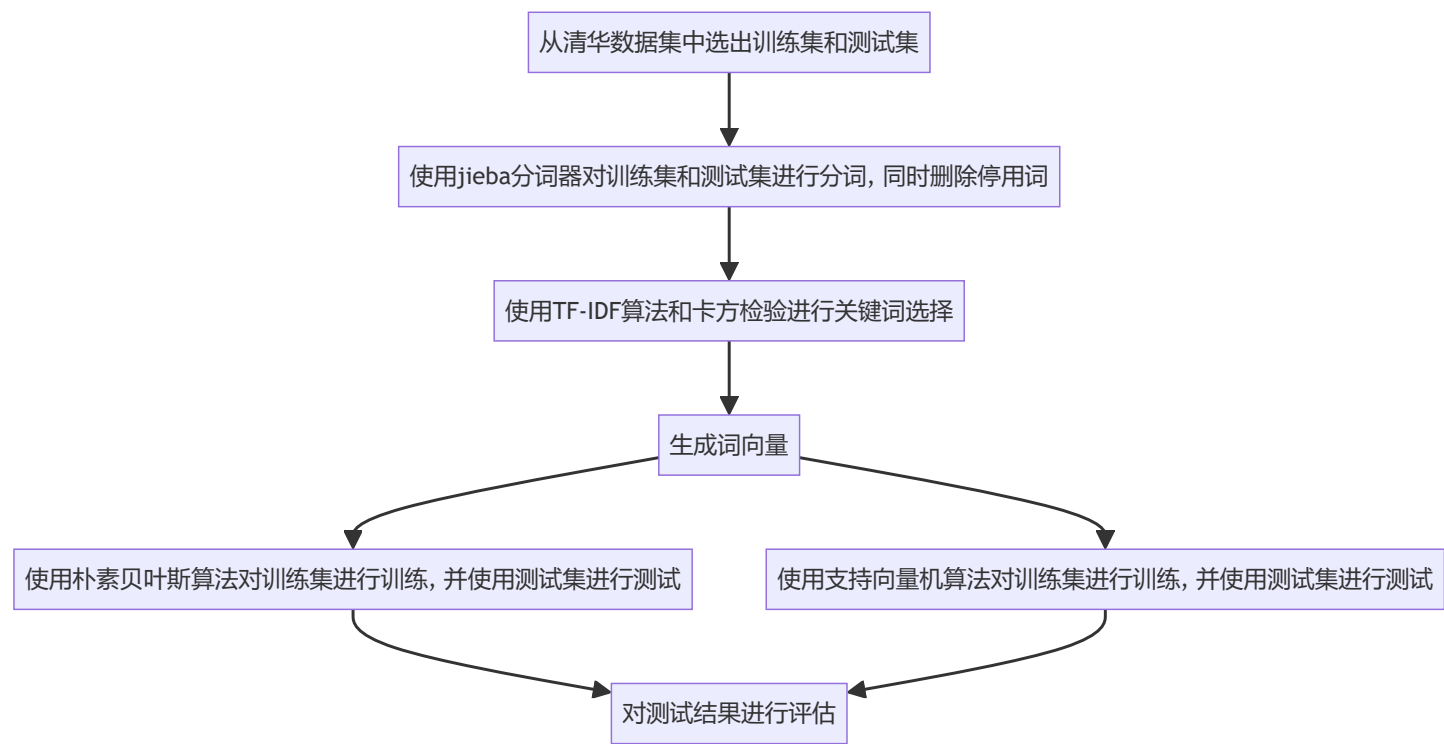
实验分工

没有组队, 所有步骤均由本人独立完成.

实验环境

- 操作系统: Windows10 64 位, Ubutun 20.04 LTS (分词时间较长, 所以在实验时放到服务器上运行)
- Python 环境: Python 3.9.7
- 数据集来源: [中文文本分类数据集 THUCNews](#)
- 分词工具: jieba
- 分类工具:
 - 手写实现朴素贝叶斯
 - 支持向量机: sklearn.svm

主要设计思想



TF-IDF 算法

本次实验使用 TF-IDF 进行特征提取.词频(term frequency, TF), 指的是某一个给定的词语在该文件中出现的频率.

$$TF_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

上式中 $n_{i,j}$ 表示第 j 篇文章中出现第 i 个词的频数, 而 $\sum_k n_{k,j}$ 表示统计第 j 篇文章中所有词的总数.

逆向文件频率(inverse document frequency, IDF), 某一特定词语的 IDF, 可以由一类文件中的总文件数目除以该类中包含该词语之文件的数目, 再将得到的商取对数得到. IDF 是一个词语普遍重要性的度量.

$$IDF_i = \log \frac{|D|}{|\{j : t_i \in d_j\}|}$$

上式中 D 表示一个文档的集合, 有 $\{d_1, d_2, d_3, \dots\} \in D$, 取模即是计算这个集合中文档的个数. t_i 表示第 i 个词, $j : t_i \in d_j$ 表示第 i 个单词属于文本 d_j , 对其取模即是计算包含单词 i 的文本的个数.

TF-IDF 值即是 TF 值与 IDF 值之积. TF-IDF 综合表征了该词在文档中的重要程度和文档区分度.

卡方检验

卡方检验, 是用途非常广的一种假设检验方法, 它在分类资料统计推断中的应用, 包括两个率或两个构成比比较的卡方检验; 多个率或多个构成比比较的卡方检验以及分类资料的相关分析等. 卡方检验表现了该词与文本的相关程度, 本实验中使用卡方检验来进行关键词选择.

卡方检验表格:

	属于该类别	不属于该类别	共计
文本中包含该词	a	b	a+b
文本中不包含该词	c	d	c+d
共计	a+c	b+d	n

计算卡法公式为:

$$\chi^2 = \frac{(a + b + c + d) \times (a \times d - b \times c)^2}{(a + c) \times (b + d) \times (a + b) \times (c + d)}$$

实验中的代码实现见 [创建词典](#)

朴素贝叶斯

朴素贝叶斯法(Naive Bayes model)是基于贝叶斯定理与特征条件独立假设的分类方法, 朴素贝叶斯分类器的公式为:

$$\nu_{NB} = \arg \max_{v_j \in V} P v_j \prod_i P(a_i | v_j)$$

朴素贝叶斯的训练公式为:

$$P(A_i | C) = \frac{N_{ic}}{N_c}$$

但是如果使用上述公式的话则会出现一个条件概率为零, 整个结果将变成 0 的问题, 所以需要对公式进行修正, 共有两种修正方法:

- Lapalace

$$P(A_i | C) = \frac{N_{ic} + 1}{N_c + c}$$

c: 类别总数

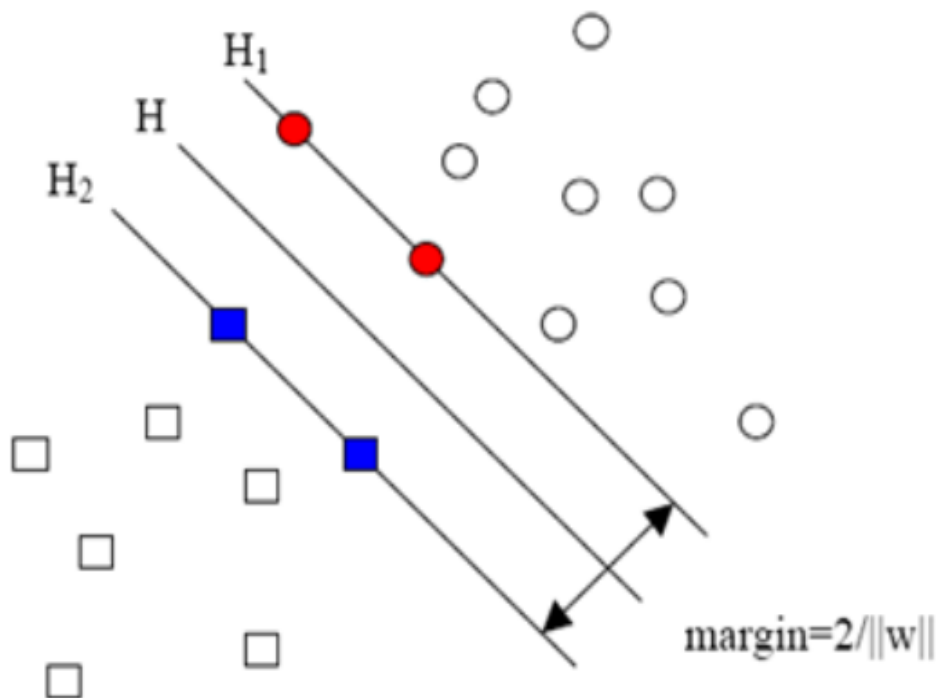
- m-estimate

$$P(A_i | C) = \frac{N_{ic} + mp}{N_c + m} = \frac{N_{ic} + 1}{N_c + |Vocabulary|}$$

本实验中的代码实现见 [朴素贝叶斯](#)

支持向量机

我们使用的 SVM(Support Vector Machine)分类算法, 它是最大 margin 分类算法中的一种. SVM 是一种可训练的机器学习方法, 在小样本中可以得到优秀的训练模型.



如上图所示, SVM 分类算法可以根据最边界的样本点, 也就是支持向量, 达到二分类的目的. 当数据的样本线性不可分时, 会通过核函数的映射, 把样本转换到线性可分的向量空间. 所以核函数的选择也会影响最终的效果. 如果遇到了多分类的问题, 可以通过多个 SVM 训练模型完成多分类的任务.

此外, SVM 与 logistic regression 非常相似. Logistic regression 虽然其名字当中是 regression 也就是回归, 但是实际上此算法是一个分类算法. 为了达到性能和效率兼备, 对不同情况运用不同算法的场景进行了描述:

n 为特征的数量, m 为训练样本的数量

- (1) 如果 n 相对 m 大很多, 可以使用 logistic regression 或者使用 SVM, 核函数选择线性核函数;
- (2) 如果 n 很小, m 的大小适中, 使用 SVM, 核函数选择 Gaussian 函数;
- (3) 如果 n 很小, m 很大, 需要添加更多的特征, 然后使用 logistic regression 或者使用 SVM, 不使用核函数.

本实验中的代码实现见 [支持向量机](#)

性能评估方法

在机器学习领域, 混淆矩阵(Confusion Matrix), 又称为可能性矩阵或错误矩阵. 混淆矩阵的结构一般如下图的表示的方法:

		Prediction	
		Positive	Negative
Reference	Positive	True Positive	False Negative
	Negative	False Positive	True Negative

混淆矩阵要表达的含义：

- 每一列代表了预测类别，每一列的总数表示预测为该类别的数据的数目；
- 每一行代表了数据的真实归属类别，每一行的数据总数表示该类别的数据实例的数目；

混淆矩阵的四个值含义：

- True Positive(TP): 真正类. 样本的真实类别是正类, 并且模型识别的结果也是正类;
- False Negative(FN): 假负类. 样本的真实类别是正类, 但是模型将其识别为负类;
- False Positive(FP): 假正类. 样本的真实类别是负类, 但是模型将其识别为正类;
- True Negative(TN): 真负类. 样本的真实类别是负类, 并且模型将其识别为负类.

正确率: 表示模型识别为正类的样本中, 真正为正类的样本所占比例, 公式为 $Precision = \frac{TP}{TP+FP}$

召回率: 召回率表现出在实际正样本中, 分类器能预测出多少, 公式为 $Recall = \frac{TP}{TP+FN}$

F1 Score: 综合了正确率和召回率, 公式为 $F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$

本实验中的代码实现见 [评估函数](#)

实验过程

创建数据集

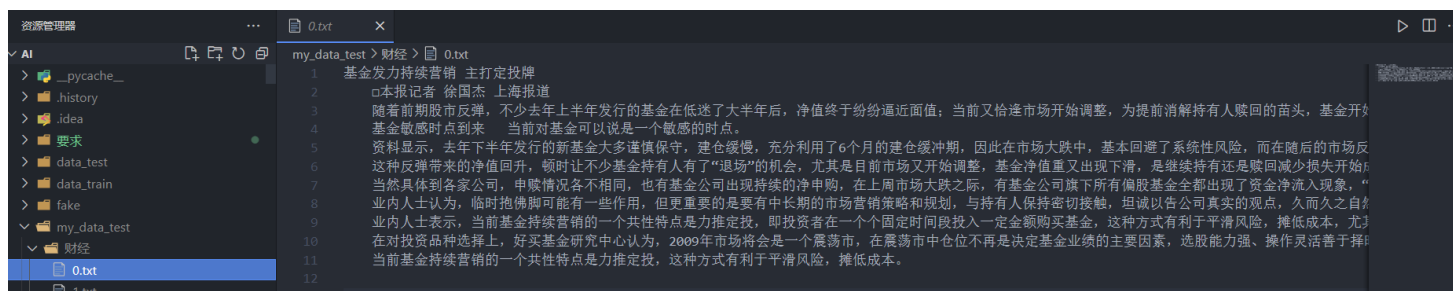
本次实验使用 [中文文本分类数据集 THUCNews](#), 选取原始数据集中的 ['财经', '房产', '社会', '时尚', '教育', '科技', '时政', '体育', '游戏', '娱乐'] 这十个类别, 每个类别按顺序选取**8000**篇训练集和**5000**篇测试集。

```
def create_datasets(src, train_dir, test_dir: str, train_size, test_size: int):
    for class_name in constants.class_list:
        dir_path = src + class_name
        file_list = os.listdir(dir_path)
        print(class_name + ':' + str(len(file_list)))

        if not os.path.exists(train_dir + class_name):
            os.makedirs(train_dir + class_name)
        for i in range(train_size):
            shutil.copy(dir_path + '/' + file_list[i], train_dir + class_name + '/' + str(i) + '.txt')

        if not os.path.exists(test_dir + class_name):
            os.makedirs(test_dir + class_name)
        for i in range(train_size, train_size + test_size):
            shutil.copy(dir_path + '/' + file_list[i], test_dir + class_name + '/' + str(i - train_size) + '.txt')
```

数据集示例:



数据预处理

- 使用 jieba 分词器对文本进行分词, 并且开启了 jieba 的 paddle 模式, 使分词结果更加准确;
- 将一个类的分词结果存储到一个文件中, 一行代表一个文本的分词结果, 方便后续操作;
- 除了老师给出的停用词外我又从网上搜集了一些中文停用词 (共 943 个) 和停用符号 (共 77 个)。

```

def del_stop_words(src, out: str):
    """
    删除停用词
    并将同一类的数据整合到一个文件中，方便后续处理

    src: 待处理文件路径
    out: 处理后文件路径
    """
    # 整个数据集分词后整合的结果
    _all = ''
    for class_name in constants.class_list:
        print(f'正在处理 {class_name} 类')

        if not os.path.exists(src + class_name):
            raise Exception(f'{src + class_name} 文件夹不存在')

        if not os.path.exists(out + class_name):
            os.makedirs(out + class_name)

        # 单个种类分词并合成整合
        all_single = ''
        for _file in os.listdir(src + class_name):
            single = ''
            with open(src + class_name + '/' + _file, 'r', encoding='utf-8') as f:
                lines = f.read()
                words = ps.cut(lines, use_paddle=True)
                for word, flag in words:
                    # 若为名词且不在停用词表中，则加入写入串
                    if flag == 'n' and not is_stop_word(word):
                        single += (word + ' ')
            all_single += single + '\n'

        with open(out + class_name + '/all.txt', 'w', encoding='utf-8') as f:
            f.write(all_single)

        _all += all_single

    with open(out + '/all.txt', 'w', encoding='utf-8') as f:
        f.write(_all)

```

停用词和停用符号:

stop_words_ch.txt

901 一起
902 一样
903 一面
904 一旦
905 一来
906 一切
907 一般
908 一天
909 一致
910 一下
911 一方面
912 其一
913 之一
914 同一
915 不一
916 万一
917 另一方面
918 一些
919 有效
920 哟
921 哎哟
922 有些
923 哪些
924 那些
925 这些
926 欢迎
927 以便
928 自从
929 咋
930 相对而言
931 愿意
932 宁愿
933 自个儿
934 自各儿
935 以后
936 大约
937 以及
938 自家
939 自己
940 越是
941 普遍
942 开展
943 换句话说
944

stop_sign.txt

```
1 ~
2 ^
3 ~
4 <
5 =
6 >
7 |
8 -
9 -
10 --
11 ,
12 ;
13 :
14 !
15 ?
16 /
17 .
18 ...
19 '
20 "
```

分词后的文本示例:

data_train > 财经 > all.txt

```
1 资金 橡胶 期价 主力 合约 关口 分析师 橡胶 现货 需求 国 天气 汽车 方案 沪胶 后市 强势 日 强势 橡胶 期货 价格 资金 主力 合约 整数 关口 终盘 结算
2 银行 基金 新军 基金 市场 基民 成绩 银行 系 基金公司 旗下 基金 数据 民生 蓝筹 民生 旗下 基金 A股 市场 基金 煤炭 行业 民生 蓝筹 基金 收益 同期 业
3 油价 压力 记者 海外市场 油价 话题 现实 桶 高点 金融危机 国际 油价 油价 压力 产业 国际 油价 之举 区间 国际 油价 强势 局势 因素 油价 风向标 油价
4 商品房 被告 称 原告 被告 业主 律师 被告 处 房屋 按揭 按揭 被告 楼款 被告 律师 答辩状 溱日 湾畔 基地产 人员 被告 空白 合同 合同 按揭 楼 款 补
5 炒房客 农民工 身份证 首套房 楼市 新政 市场 心态 节目 实录 房地产市场 住房 价格 城市 房价 通知 政策 组合拳 楼市 房价 市场 动向 楼市 新政 炒家 桂
6 手 账户 储户 损失 手机 信息 短信 时 纠纷案 一审 银行 储户 责任 该县 银行 支行 卡 账户 手机 短信 银行 账户 现金 店面 短信 短信 情况 银行 身份
7 存量 房贷 机客 潮 存量 房 贷款利率 消息 人 网站 存量 房 房贷 利率 房贷 利率 比例 基准 利率 人 压力 客服 人员 电话 存量房 贷款利率 人 数量 人
8 骗子 手机 银行卡 账户 账户 手 密码 钱 警方 案件 化名 海 江湖 眼皮 账户 警方 省市 手机 网银 团伙 省 诈骗案 低息 公司 公司 老板 公司 际 广告 化
9 银行 高管 薪酬 记者 银行 年报 高管 薪酬 高管 薪酬 榜首 行长 年薪 行长 高薪 董事长 天价 年薪 董事长 股份制 银行 银行 高管 薪酬 风景 薪酬 行长 生
10 银行 存款 储户 漏洞 存款 储户 银行 漏洞 诈骗案 被告人员 诈骗案 存款 储户 漏洞 损失 储户们 法院 储户 判决书 诈骗案 受害者 时 存折 存款 钱 存折
11 故障 图 记者 银行 取款机 钱 人 秘密 银行 损失 钱 女士 家具 取款机 钱 现金 零钱 取款机 屏幕 钞票 取款机 零用钱 女士 电子 系统 银行 记录 同学
12 定期存款 银行 储户 银行 人员 储户 妻子 存款 人员 事 该行 人员 对方 记者 储户 该行 大堂 经理 该行 活动 储户 银行 储户 钱 储户 储户 钱 银行 储
13 房贷 新政 楼市 买方 市场 角色 爷 电话 人 记者 房屋 中介 门店 房价 楼市 卖方 市场 强势 地位 家 市场 研究中心 分析师 记者 楼市 新政 市场 印象
14 幌 身份 信息 骗子 卡 记者 公司 员工 幌子 人员 身份 信息 信用卡 记者 警方 骗子 老总 警方 警方 信用卡 银行 欠款 金额 民警 信用卡 两人 工作 身份
15 人 悬念 记者 新址 大火 世人 大火 焦点 三湘 烟花 姓名 人员 新址 烟花 公司 信息 警方 事 火灾 事故 责任 火灾 事故 情况 业内人士 金额 责任 人 记者
16 银行 媒体 消息 银行 二套房 政策 漏洞 客户 二套房 记者 银行 客户 贷款 银行 银行 客户 记者 时 股份制 银行 房贷 部 负责人 客户 房子 刚性 需求 银行
17 银行 党委会 领导 员工 福利 记者 通讯员 名行 领导 国有资产 银行 全体 员工 人 集体 保险 福利 领导 银行 职工 赃款 精品 案件 记者 案情 国有资产 国
18 银行 密码 富豪 电 家 豪宅 中年 富豪 亡父 遗产 户口 户口 密码 支票簿 富豪 信件 通知 密码 户口 价值 股票 现金 集团 成员 中年 男子 黑钱 罪名 案件
19 信贷 经理 记者 人士 票据 银行 银行 家 银行 信贷 经理 嫌疑犯 时间 银行 承兑汇 人士处 银行 支行 信贷 经理 银行业务 知识 时间 商业银行 处 信贷 经
20 身份证 银行 身份 身份证 银行 业务 银行 身份 系统 公安部门 公民 信息系统 公安部门 皮球 银行 身份证 身份 网点 身份 证件 时 银行 信息 系统 业务 作
21 概念 噱头 收益 词汇 银行 专家 概念 银行 噱头 产品 收益 产品 球 产品 投资者 银行 足球 概念 产品 足球 上市公司 股票 年 收益 银行 银行 银行 主意
22 第三方 资金 通道 记者 时间 球迷 化名 球 球 赌球 网络 赌球 道 赌球 网络 赌球 网站 记者 球客 庄家 过程 平台 身影 平台 赌球 网站 资金 通道 赌球 贝
23 常识 消息 声 报道 山寨 版 TM机 常识 人 山寨 TM机 部门 负责人 人 卡号 密码 消费者 损失 记者 山寨 部门 真仿 责任 消费者 心思 机身 标志 市民 常识
24 高层 银行业务 际 银行业务 目标 远景 员工 情绪 银行 成本 意图 记者 银行 员工 情绪 高层 岗位 工作 员工 速度 措施 安心 作用 渠道 外界 传闻 消息 之
```

创建词典

- 将每个类的词典保存到二进制文件中, 方便存储和读入;
- 对每个类中词频大于 30 的词计算 TF-IDF 值, 并且将其和这些词的卡方相乘得到新的权值, 选取权值前 1000 的词组成关键词表;

```

def init_word_counters(path, output: str) -> (dict, dict):
    """
    初始化每个分类的词典和词频统计器
    """
    words = {}
    words_cnt = {}

    for class_name in constants.class_list:
        with open(path + class_name + '/all.txt', 'r', encoding='utf-8') as file:
            words[class_name] = ''.join(file.readlines())
            words_cnt[class_name] = Counter(words[class_name].split())

        if not os.path.exists(output + class_name):
            os.makedirs(output + class_name)

        with open(output + class_name + '/tf.pkl', 'wb') as file:
            pickle.dump(dict(words_cnt[class_name]), file)

    return words, words_cnt

def calculate_chi_square(data_path, word, class_str: str):
    """
    计算卡方统计量
    """
    a = 0
    b = 0
    c = 0
    d = 0

    for class_name in constants.class_list:
        with open(data_path + class_name + '/all.txt', 'r', encoding='utf-8') as file:
            lines = file.readlines()

            if class_name == class_str:
                for line in lines:
                    if word in line.split():
                        a += 1
                    else:
                        c += 1
            else:
                for line in lines:
                    if word in line.split():
                        b += 1
                    else:
                        d += 1

    if b == 0:

```

```

    return 1

    return (a + b + c + d) * ((a * d - b * c) ** 2) / ((a + c) * (a + b) * (d + b) * (c + d))

def extract_keywords(data_path, output_path: str):
    """
    提取关键词并计算权重
    """
    words, words_cnt = init_word_counters(data_path, output_path)
    all_keywords = []

    for class_name in constants.class_list:
        # 统计词频大于30的词，并计算tf-idf值
        top_words = words_cnt[class_name].most_common()
        top_k = next((i for i, (word, freq) in enumerate(top_words) if freq < 30), 2000)
        tags = jieba.analyse.extract_tags(words[class_name], topK=top_k, withWeight=True)
        tags_dict = dict(tags)
        index = 0

        # 计算卡方，并将每个词的tf-idf值乘以卡方值得到新的权重
        for tag_word, value in tags_dict.items():
            index += 1
            tags_dict[tag_word] = value * calculate_chi_square(data_path, tag_word, class_name)

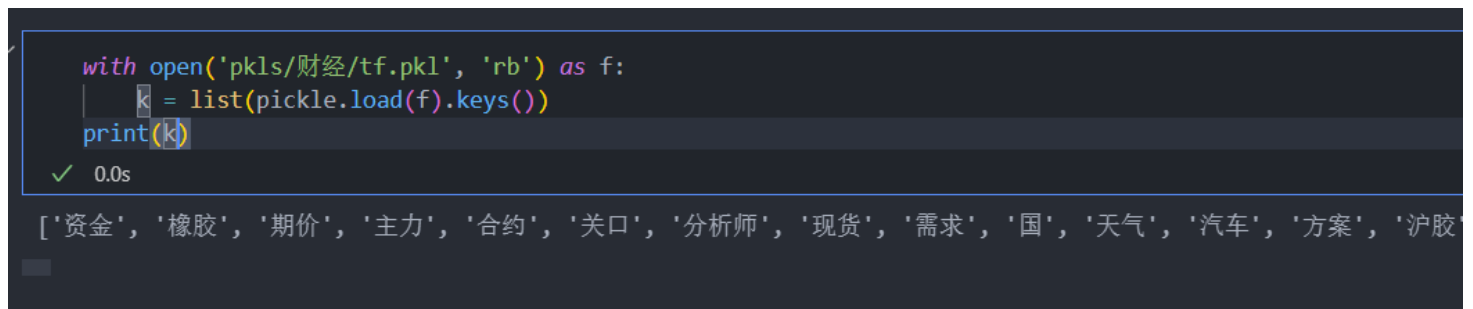
        # 选取前1000的词作为关键词
        all_keywords += [k for k, v in Counter(tags_dict).most_common(1000)]

    all_dict = Counter(all_keywords)

    with open(output_path + 'keywords.pkl', 'wb') as f:
        pickle.dump(all_dict, f)

```

一个类别的词典示例:



```

with open('pkls/财经/tf.pkl', 'rb') as f:
    k = list(pickle.load(f).keys())
print(k)

```

✓ 0.0s

['资金', '橡胶', '期价', '主力', '合约', '关口', '分析师', '现货', '需求', '国', '天气', '汽车', '方案', '沪胶']

最终得到关键词共**6507**个, 示例如下:

```

with open('pkls/keywords.pkl', 'rb') as f:
    keywords = list(pickle.load(f).keys())
print(keywords)
print(len(keywords))

```

✓ 0.0s

```

['基金', '市场', '投资者', '收益', '股票', '银行', '净值', '债券', '经理', '资产', '业绩', '仓位',
6507

```

生成词向量

为了方便存储和加快后续使用时读入的速度, 在实验中我将训练集和测试集的词典按照关键词表生成了一个词向量矩阵, 并且因为矩阵很稀疏, 所以使用 `scipy` 存储成二进制的稀疏矩阵.

```

def create_word_vectors(dataset, output: str, dataset_size: int, _keywords: list):
    """
    生成词向量
    """
    max_idx = len(_keywords)
    print(max_idx)

    arr = np.zeros(shape=(dataset_size * len(constants.class_list), max_idx))
    index = 0

    for class_name in constants.class_list:
        print(class_name)
        arr, index = process_data(dataset + class_name + '/all.txt', _keywords, arr, index)

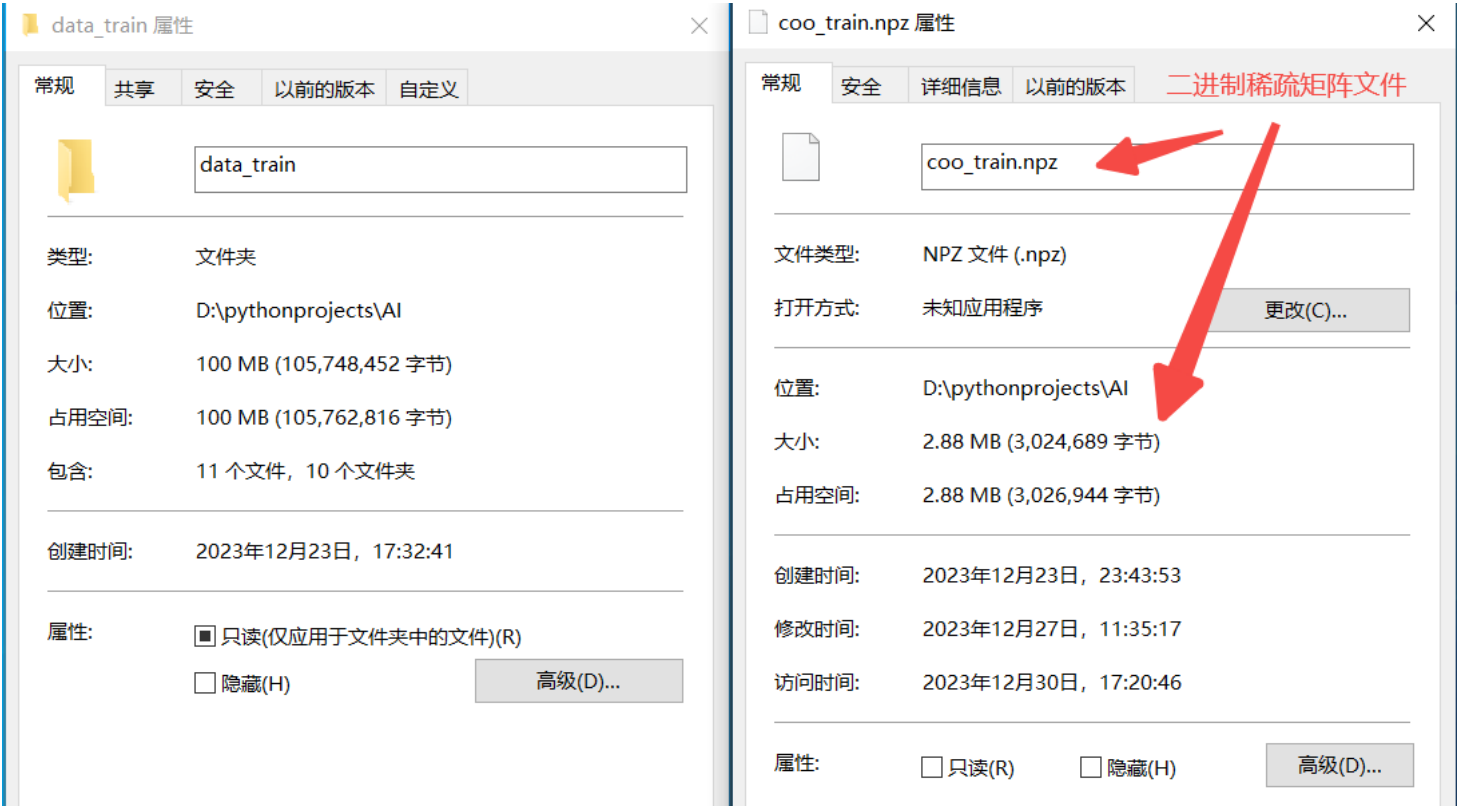
    res = coo_matrix(arr)
    save_npz(output, res)

def process_data(path, _keywords, arr, index):
    with open(path, 'r', encoding='utf-8') as file:
        lines = file.readlines()

    for line in lines:
        for word in line.split():
            if word not in _keywords:
                continue
            else:
                word_index = _keywords.index(word)
                arr[index][word_index] += 1
        index += 1
    return arr, index

```

以训练集为例, 分词后的数据集大小为 100 MB , 经过稀疏矩阵压缩后, 存储文件大小仅为 2.88 MB , 大大减少了存储空间和加快了后续的读取速度.



朴素贝叶斯

训练阶段

训练时为避免后续出现 0 概率问题, 我使用了拉普拉斯修正 (在实验中也试过 m -估计, 但是表现效果没有拉普拉斯效果好, 所以最终选用拉普拉斯), 训练结束后将概率矩阵保存成二进制文件方便后续预测使用.

```

def train_bayes(_keywords: list, datasets, output: str):
    """
    朴素贝叶斯训练
    """
    max_idx = len(_keywords)
    word_df = pd.DataFrame(np.zeros((max_idx, len(constants.class_list))), columns=constants.class_list,
                           index=_keywords)
    bayes_df = pd.DataFrame(np.zeros((max_idx, len(constants.class_list))), columns=constants.class_list,
                             index=_keywords)

    # 构建关键词词频矩阵
    for _, class_name_1 in enumerate(constants.class_list):
        with open(datasets + class_name_1 + '/tf.pkl', 'rb') as file:
            tf_dict = pickle.load(file)
            for word in word_df.index:
                if tf_dict.get(word) is None:
                    continue
                else:
                    word_df.at[word, class_name_1] = tf_dict.get(word)

    # 构建条件概率矩阵
    for word in bayes_df.index:
        for class_name_1 in constants.class_list:
            # 拉普拉斯修正
            bayes_df.at[word, class_name_1] = (word_df.at[word, class_name_1] + 1) / (
                word_df[class_name_1].sum() + len(constants.class_list))

    with open(output, 'wb') as file:
        pickle.dump(bayes_df, file)

```

测试阶段

预测时因为训练集和测试集中每个类别的样本均相等, 所以在我的代码中贝叶斯分类器公式可以简化成:

$$\nu_{NB} = \arg \max_{v_j \in V} \Pi_i P(a_i | v_j)$$

```
def _predict(text_pos: int, column, row, _keywords, bayes_df):
    pred = {}
    for class_name in constants.class_list:
        pred[class_name] = 1
        for v in column[row[text_pos]:row[text_pos + 1]]:
            w = _keywords[v]
            pred[class_name] *= bayes_df.at[w, class_name]

    res = sorted(pred.items(), key=lambda kv: (kv[1], kv[0]), reverse=True)
    return res[0][0]

def predict_bayes(npz_path, confusion_matrix_path: str, test_size: int, _keywords: list, bayes_train_path:
    """
    朴素贝叶斯预测
    """

    test = load_npz(npz_path).tocsr()
    column = test.indices
    row = test.indptr

    with open(bayes_train_path, 'rb') as file:
        bayes_df = pickle.load(file)

    confusion_matrix = pd.DataFrame(np.zeros((len(constants.class_list), len(constants.class_list))),
                                    columns=constants.class_list, index=constants.class_list)

    class_index = 0
    for class_name in constants.class_list:
        for i in range(test_size):
            s = _predict(class_index * test_size + i,
                          column, row, _keywords, bayes_df)
            confusion_matrix.at[class_name, s] += 1
            class_index += 1
    confusion_matrix.to_csv(confusion_matrix_path)
```

性能评估

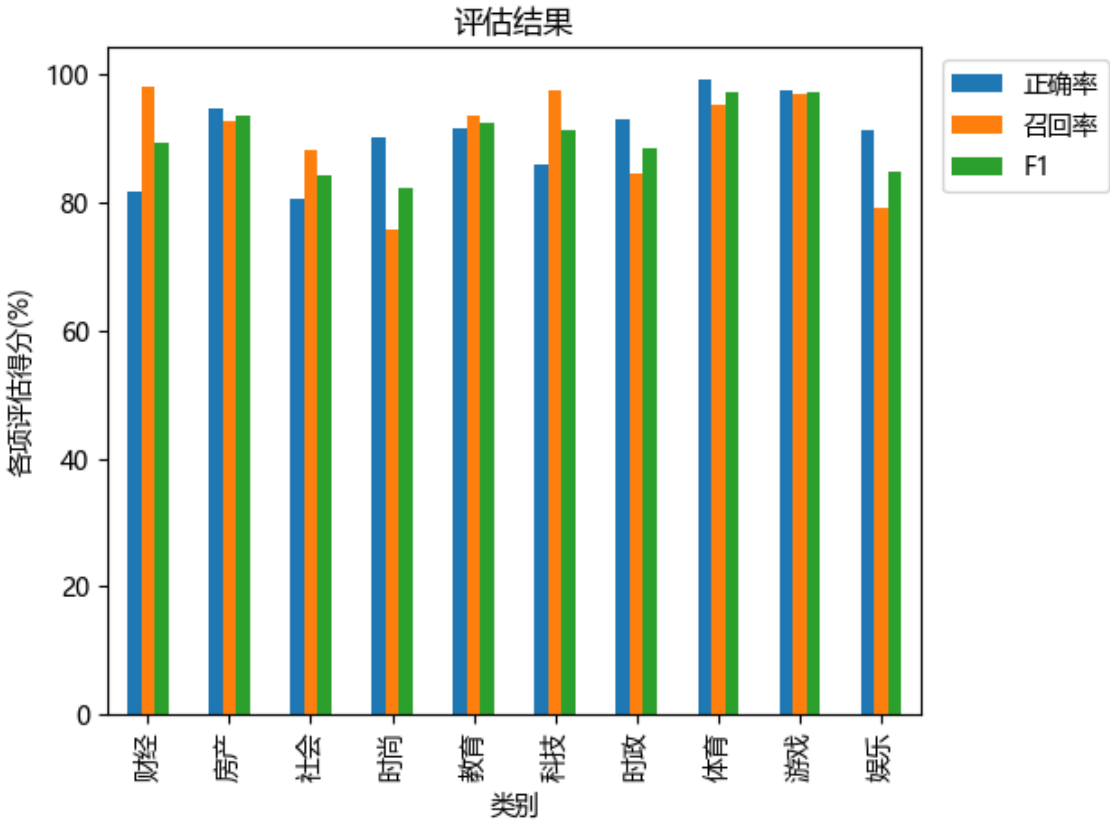
混淆矩阵:

	财经	房产	社会	时尚	教育	科技	时政	体育	游戏	娱乐
财经	4914	24	2	1	16	1	36	0	5	1
房产	199	4636	37	17	11	7	59	2	14	18
社会	124	46	4415	13	182	63	75	6	10	66
时尚	19	3	319	3790	55	625	28	4	9	148

	财经	房产	社会	时尚	教育	科技	时政	体育	游戏	娱乐
教育	175	8	70	2	4677	11	30	2	6	19
科技	53	9	1	11	5	4883	2	2	31	3
时政	272	135	177	3	109	48	4226	0	5	25
体育	55	1	52	7	15	3	26	4762	10	69
游戏	31	11	11	20	8	18	12	4	4856	29
娱乐	166	20	396	331	26	16	41	13	29	3962

评估得分:

	财经	房产	社会	时尚	教育	科技	时政	体育	游戏	娱乐	整体
正确率	81.79	94.75	80.57	90.35	91.63	86.04	93.19	99.31	97.61	91.29	90.65
召回率	98.28	92.72	88.30	75.80	93.54	97.66	84.52	95.24	97.12	79.24	90.24
F1	89.28	93.72	84.26	82.44	92.58	91.48	88.64	97.23	97.36	84.84	90.45



支持向量机

训练和测试

训练使用 `sklearn.svm` 的 `SVC` 和 `LinearSVC` 函数.

我在本实验中 SVM 的核模型选取为线性核, 因为在实验中我使用高斯核进行调参时发现训练时间过长 (训练一次最短也需要 40+分钟), 并且高斯核的训练结果也不太理想, 和线性核的结果差不多, 所以综合考虑我使用了线性核, 并且设置迭代次数保证每次完全收敛 (不过也保留了高斯核的代码, 可以在训练时通过参数控制使用哪一种核函数)

训练结束后同样把模型保存成二进制文件以供后面测试使用.

```
def train_svc(npz_path, output: str, train_size: int, c: float = 0.1, gamma: float = 0.1, max_iter: int =
    kernel: str = 'linear'):
    """
    svm训练, 自定义核函数
    """

    coo_train = load_npz(npz_path)
    train_arr = np.array([int(i / train_size) for i in range(train_size * len(constants.class_list))]) #

    if kernel == 'linear':
        print(f"kernel: linear, C: {c}, max_iter: {max_iter}")
        model = LinearSVC(C=c, max_iter=max_iter)
    elif kernel == 'rbf':
        print(f"kernel: rbf, C: {c}, gamma: {gamma}")
        model = SVC(kernel='rbf', C=c, gamma=gamma)
    else:
        raise Exception('kernel type error, only support linear and rbf')

    start = time.time()
    model.fit(coo_train.tocsr(), train_arr)
    end = time.time()
    print('Train time: %s seconds\n' % (end - start))

    with open(output, 'wb') as f:
        pickle.dump(model, f)
```

测试时直接读入训练好的模型文件和测试集的稀疏矩阵, 并且使用 `sklearn.metrics` 包直接获取到混淆矩阵.

```

def predict_svc(test_npz, model_path, confusion_matrix_path: str, test_size: int):
    """
    svm预测
    """
    coo_test = load_npz(test_npz)
    test_arr = np.array([int(i / test_size) for i in range(test_size * len(constants.class_list))])

    with open(model_path, 'rb') as f:
        model = pickle.load(f)

    start = time.time()
    pre = model.predict(coo_test.tocsr())
    end = time.time()
    print('Test time: %s seconds\n' % (end - start))

    c = metrics.confusion_matrix(test_arr, pre)
    confusion_matrix = pd.DataFrame(c, columns=constants.class_list,
                                    index=constants.class_list)
    confusion_matrix.to_csv(confusion_matrix_path)

```

参数调整

使用网格调参, 因为使用了线性核, 所以只有一个惩罚因子 c 需要调整, 设定 c 的范围

为 $[0.001, 0.01, 0.1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100]$, 在此范围内使用下述代码进行调参, 并且注意还要调整迭代次数保证每次训练时完全收敛.

```

C_values = [0.001, 0.01, 0.1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100]

for C in C_values:
    # 训练模型，迭代次数也不断增加保证每次都能收敛
    max_iter = 100000
    if C >= 1:
        max_iter *= C
    svm.train_svc(c=C, max_iter=max_iter, npz_path='coo_train.npz', output='pkls/svm_train.pkl', train_size=
        kernel='linear')

    # 预测并评估模型
    svm.predict_svc('coo_test.npz', 'pkls/svm_train.pkl', 'confusion_matrix_svm.csv', 5000)
    score = evaluate.evaluate('confusion_matrix_svm.csv')

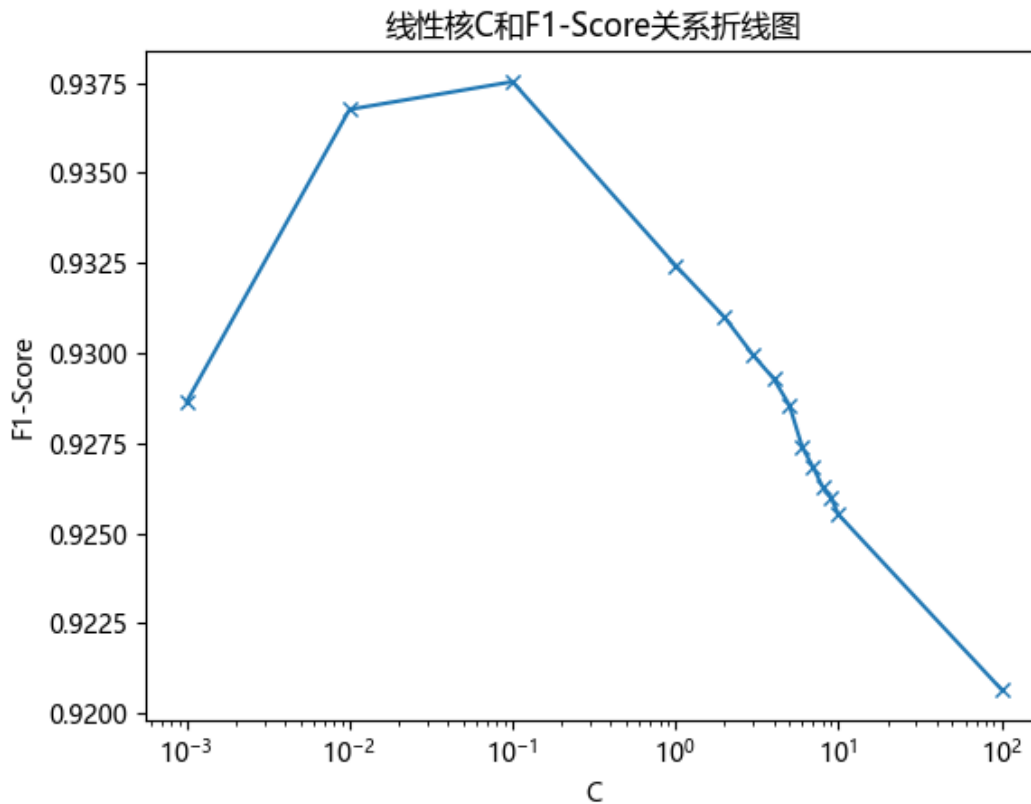
    scores.append({'C': C, 'score': score})

# 更新最佳得分和参数组合
if score > best_score:
    best_score = score
    best_params = {'C': C}

print("Best Parameters:", best_params)
print("Best Score:", best_score)

```

每次记录测试结果的总体 F1 值, 将调参结果绘制折线图如下



通过上图可以看到最佳参数为 $C=0.1$

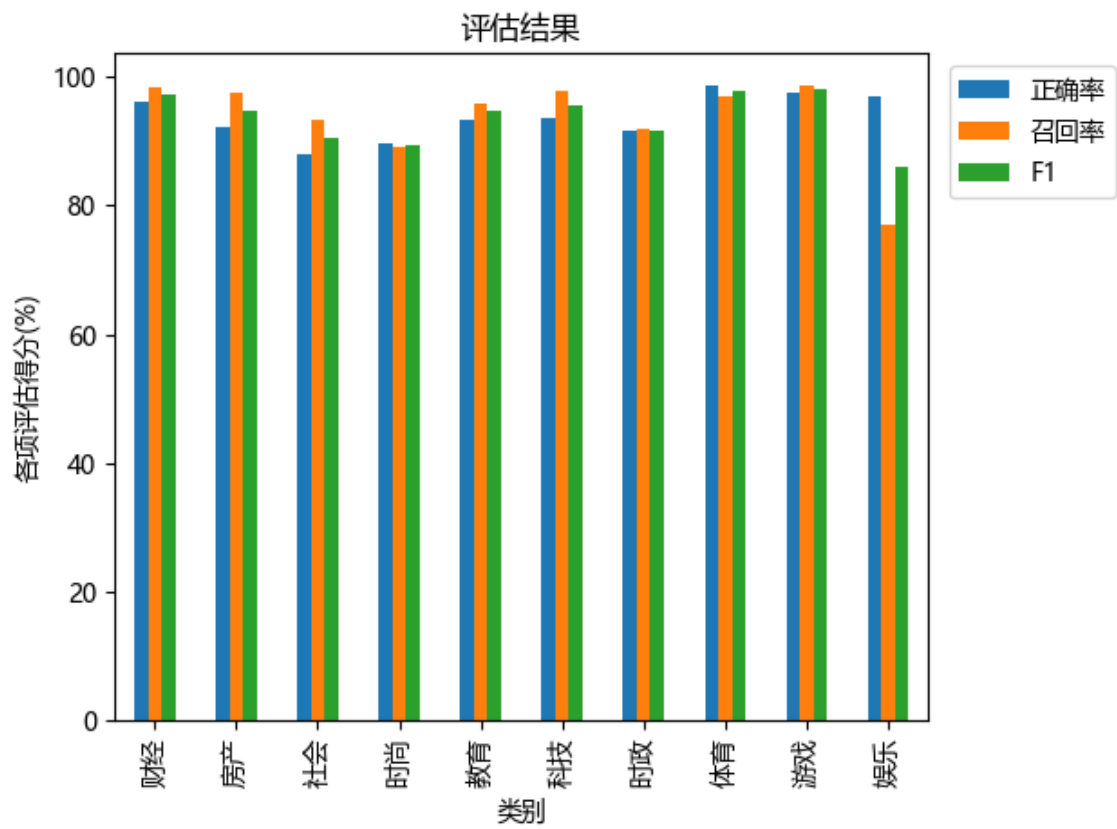
性能评估

混淆矩阵:

	财经	房产	社会	时尚	教育	科技	时政	体育	游戏	娱乐
财经	4919	52	3	3	6	1	11	0	4	1
房产	19	4884	11	13	6	7	46	0	9	5
社会	36	61	4672	11	67	39	82	7	7	18
时尚	23	19	127	4453	86	196	26	10	20	40
教育	1	5	44	5	4797	21	112	3	5	7
科技	5	13	6	53	6	4888	14	2	12	1
时政	60	144	81	12	58	33	4590	4	3	15
体育	7	31	20	20	18	4	5	4857	13	25
游戏	6	4	6	21	5	8	5	2	4934	9
娱乐	45	81	336	373	85	21	116	35	52	3856

评估结果:

	财经	房	社会	时尚	教育	科技	时政	体育	游戏	娱乐	整体
正确率	96.06	92.26	88.05	89.71	93.44	93.68	91.67	98.72	97.53	96.96	93.81
召回率	98.38	97.68	93.44	89.06	95.94	97.76	91.80	97.14	98.68	77.12	93.70
F1	97.20	94.89	90.67	89.38	94.67	95.67	91.74	97.92	98.10	85.91	93.75



评估函数

根据混淆矩阵计算每个类别和整体的正确率、召回率和 F1 值, 将分析结果绘图并返回整体的 F1 值方便 SVM 的调参

```

def draw_pic(eval_table: DataFrame, pic_path: str = 'evaluation.png'):
    """
    对测评结果进行绘图
    """
    data = pd.DataFrame(np.zeros((len(constants.class_list), 3)), index=constants.class_list,
                        columns=eval_table.index.values)
    for class_name in constants.class_list:
        for metric in eval_table.index.values:
            data.at[class_name, metric] = eval_table.at[metric, class_name]

    data.plot(kind='bar', legend=True)
    plt.legend(bbox_to_anchor=(1.01, 1), loc=2)

    plt.xlabel('类别')
    plt.ylabel('各项评估得分(%)')
    plt.title('评估结果')

    plt.tight_layout()
    plt.savefig(pic_path)


def evaluate(csv_path: str, output_path='evaluation.csv') -> float:
    """
    根据混淆矩阵计算正确率, 召回率和F1值
    返回整体的F1值
    """
    df = pd.read_csv(csv_path, index_col=0).apply(
        pd.to_numeric, errors='coerce').astype(int)
    pd.set_option('display.max_columns', None)
    print("混淆矩阵: \n", df, "\n")

    eval_table = pd.DataFrame(np.zeros((3, len(constants.class_list) + 1)), columns=constants.class_list +
                              index=['正确率', '召回率', 'F1'])

    precision_all = 0
    recall_all = 0
    for class_name in constants.class_list:
        precision = df.at[class_name, class_name] / df[class_name].sum()
        precision_all += precision
        eval_table.at['正确率', class_name] = round(precision * 100, 2)

        recall = df.at[class_name, class_name] / df.loc[class_name, :].sum()
        recall_all += recall
        eval_table.at['召回率', class_name] = round(recall * 100, 2)

        f1 = 2 * (precision * recall) / (precision + recall)
        eval_table.at['F1', class_name] = round(f1 * 100, 2)

```

```
precision_all = precision_all / len(constants.class_list)
recall_all = recall_all / len(constants.class_list)
f1_all = 2 * (precision_all * recall_all) / (precision_all + recall_all)

eval_table.at['正确率', '整体'] = round(precision_all * 100, 2)
eval_table.at['召回率', '整体'] = round(recall_all * 100, 2)
eval_table.at['F1', '整体'] = round(f1_all * 100, 2)

pd.set_option('display.float_format', '{:.2f}'.format)
print("评估结果: \n", eval_table, "\n")
eval_table.to_csv(output_path)

draw_pic(eval_table)

return f1_all
```

总结和反思

1. 数据集的选择

通过与同学实验结果进行比较, 发现从网上爬取的文本数据质量要高一些. 并且在实验中也没有充分利用已有数据集, 新建数据集的时候只是按顺序选取, 或许用 `sklearn` 中随机分割训练集和测试集的函数能充分利用已有数据, 这样实验结果可能会更好一些.

2. SVM 的结果

在本实验中 SVM 的测试结果不太理想, 可能还是原始数据集的问题, 并且后续才了解到训练时可以加速, 这样可以大大减少训练的时间.

3. 评估结果分析

同种算法比较, 可以看到在朴素贝叶斯和 SVM 中均是时尚和娱乐这两类召回率过低, 猜想可能是原始数据集中本身就存在一些分类比较模糊的文本或者说这些种类本身就比较类似. 可以由此得知除了数据集的选择外, 样本的分类也应当避免各个种类过于相似, 比如我一开始的实验分类同时包含有股票和财经两个分类, 结果最终的测试结果财经的召回率仅为 60% 左右.

1. 数据文件的存储

一开始我都是使用文本文件存储, 但是发现每次生成的文件太大, 读入速度慢. 上网查询资料后得知可以存储成二进制文件, 这样不仅减小了文件大小, 同时也大大加快了文件的读取速度, 所以在此次实验中我除了分词后的文件、测试后的混淆矩阵和评估后的评估结果外, 其他文件均以二进制文件存储.

5. 提取关键词

提取关键词的时候我同时使用了 TF-IDF 和卡方检验, 因为只是想着把这两个参数都用上, 所以参照 TF-IDF 的算法只是把这两个参数简单相乘. 后续想到或许将二者以某种算法结合起来得到的新权值或许会更准确一些, 不过因为查

阅资料没有找到类似的算法, 所以最终也只是采用了简单相乘的方法. 这一点可能会导致关键词提取的不准确, 从而导致后续的训练结果不理想.