本科生实验报告

机器学习技术综合训练

实验 3：基于贝叶斯方法的文本分类

姓名： 杨豪

班级： 软件 2101

时间： 2023 年 4 月 17 日

学号： 2206213297

目录

实验 3：基于贝叶斯方法的文本分类

3.1 实验内容

分别用 Term Frequency 与 Bernoulli 方法实现基于贝叶斯方法的文本分类算法，并在给定的数据集上进行训练与测试。

要求：

1. 代码补充完整 (Bernoulli 方法选做)
2. 调整预处理函数，看看部分预处理操作的有/无对结果有什么影响
3. 实验文档内容需要包括：实验原理、代码及对应简要说明、实验结果 (准确率)

提示：

- 未登录词：即只在测试集中出现过，而没有在训练集中出现过的词。可以直接跳过这个词，当它不存在。因为已得到的贝叶斯模型中不含与它相关的知识，这个词对分类没有帮助。
- 数据集相对比较小，两种还原方法的速度差异并不明显，但在大数据集上的速度差异比较明显。
- 计算过程中尽量多用矩阵操作，速度较快。

3.2 实验原理

3.2.1 背景

词袋模型 Bag of Words(BoW) ，将句子转化成长度与词汇表长度一致的向量

- 优点：简单方便
- 缺点：
  - 一段文本只会用到词汇表中的一部分词，对大文本库，通过这种方法获得的向量会很稀疏 (即包含很多 0)
  - 文本上下文之间的关联 (即文本中单词的顺序) 信息被抹除了
  - 对中文文本需要引入额外的分词工具进行词组切分

停用词，即在文本中极为常见或无实际意义，无法起到分类作用的词，例如：so, and, or, the, a,... 构建文本向量时，通常要将这些停用词删掉不放入词汇表中，以减少向量的维度 (臃肿程度)。除了停用词，标点符号、数字也可以认为是与分类无关的内容，可将之删去

词干提取 (Stemming)，所得未必是真实的单词，计算复杂度较低、速度较快；词形还原 (Lemmatization)，所得必然是真实的单词，计算复杂度较高、速度较慢。

3.2.2 分类方法

文档类别的集合为 C ，共计 k 类：$C=\{c_1,c_2,...,c_k\}$

训练集的词汇表为 D ，共计 m 词：$D=\{d_1,d_2,...,d_m\}$

待分类的一个文档内容为 $text$：$text=w_1,w_2,...,w_n$

目标：$text$ 所属的类别 c_{text}

由 Bayes 公式，

$$c_{text}=\text{argmax}_{c\in C}P(c\mid text)=\text{argmax}_{c\in C}\frac{P(text\mid c)P(c)}{P(text)}$$

对同一 text，分母相同，只需要比较分子，即可取 $c_{text}=\text{argmax}_{c\in C}P(text\mid c)P(c)$

采用频率逼近概率的思想：

$$P(c)=\frac{\text{这类文档数量}}{\text{所有文档数量}}=\frac{N(c,text)}{N(text)}$$

Term Frequency

由 TF 方法的思想

$$P(text\mid c)=P(w_1,w_2,\cdots,w_n\mid c)=\prod_{i=1}^{n}P(w_i\mid c),w_i\in D$$

采用频率去逼近概率:$P(w_i\mid c)=\dfrac{N(w_i\text{ in }W_c)}{N(W_c)}$,但实际计算中常取 $\dfrac{N(w_i\text{ in }W_c)+1}{N(W_c)+m}$, 这样既能防止 $P(w_i\mid c)=0$ 又能保持 $\displaystyle\sum_{j=1}^{m}P(d_j\mid c)=1$ （拉普拉斯平滑）

故

$$c_{text}=\text{argmax}_{c\in C}\prod_{i=1}^{n}P(w_i\mid c)P(c)$$

程序中连乘易趋向于 0，通过取对数解决

$$c_{text}=\text{argmax}_{c\in C}\left[\ln P(c)+\sum_{i=1}^{n}\ln P(w_i\mid c)\right]$$

Bernoulli

由 Bernoulli 方法的思想

$$P(text \mid c) = P(d_1, d_2, \cdots, d_n \mid c) = \prod_{j=1}^{m} P(d_j \mid c)^b (1 - P(d_j \mid c))^{1-b}, d_j \in D, b = \begin{cases} 1 \text{ if } d_j \in text \\ 0 \text{ else} \end{cases}$$

采用频率去逼近概率：

$$P(d_j \mid c) = \frac{\text{这类文档中出现该词的文档个数}}{\text{这类文档的总个数}} = \frac{N(C_{d_j})}{N(C)}$$

同上，实际计算中常用 $P(d_j \mid c) = \dfrac{N(C_{d_j}) + 1}{N(C) + 2}$

故

$$c_{text} = \text{argmax}_{c \in C} \prod_{j=1}^{m} P(d_j \mid c)^b (1 - P(d_j \mid c))^{1-b} P(c)$$

同上，对该结果取对数

$$c_{text} = \text{argmax}_{c \in C} \left[\ln P(c) + \sum_{j=1}^{m} \ln P(d_j|c)^b (1 - P(d_j|c))^{1-b} \right]$$

3.3 框架代码解读、补充与修改

题目已给定的代码（包含部分自己的修改）有三个部分

- 主函数和全局变量，预定义训练类型和预处理类型、文本种类、停用词，调用各函数
- preprocess 函数。根据给定的预处理类型清洗、整理数据：将输入的句子转化为单词词组，并统一为小写、去标点、去停用词、去数字、还原；
- load 函数，根据指定路径读取 csv 格式下确定格式的训练集或测试集并根据预处理类型调用 preprocess 函数
- words2dic 函数，扫描给定的训练集并生成词汇表字典
- train_TF 函数，根据给定训练集生成词汇表字典，并由实验原理通过统计词频计算出 $P(w_i \mid c)$ 和 $P(c)$
- train_Bernoulli 函数，根据给定训练集生成词汇表字典，由实验原理统计词频并计算出 $P(d_j \mid c)$ 和 $P(c)$s
- test 函数，根据给定的训练方法和 $P(w_i \mid c)$、$P(c)$ 在测试集上计算准确率

主要修改了 train 和 test 函数，分条叙述如下

3.3.1 train_TF 函数

根据实验原理填充如下

Listing 1: train_TF

```
def train_TF(train_x, train_y):
    # ...... Some code......

    for words, cate in  zip(train_x, train_y):
        for word in words:
            words_frequency[dictionary[word]][categories[cate]] += 1
        category_sents[categories[cate]] += 1

    # p(c) (维度:类别数x1)
    p_c = category_sents /  len(train_y)

    # n(w_c) 每类下的词总数(维度:类别数x1)
    category_words = np. sum(words_frequency, 0)

    # p(w_i|c) (维度:词汇数x类别数)
    p_stat = (words_frequency + 1) / (category_words +  len(dictionary))

    return p_stat, dictionary, p_c
```

算法很简单。一次扫描即可得到 category_sents 和 words_frequency 因为字典基于训练集搭建，所以数据肯定都在字典中，不必判断；为了提高训练速度采用**向量**的方式整合了大量相似的运算：以向量为单位求出 p_stat、p_c 和 category_words

3.3.2 train_Bernoulli 函数

根据实验原理填充如下

Listing 2: train_Bernoulli

```
def train_Bernoulli(train_x, train_y):
# ...... Some Code ......
    for word in dictionary:
    for words, cate in  zip(train_x, train_y):
        if word in words:
            words_frequency[dictionary[word]][categories[cate]] += 1
```

```
        category_sents[categories[cate]] += 1

    # p(c) (维度:类别数x1)
    p_c = category_sents /  len(train_y)

    # p(w_i|c) (维度:词汇数x类别数)
    p_stat = (words_frequency + 1) / (category_sents + 2)

    return p_stat, dictionary, p_c
```

同上，但每个文档只计算一次，所以需要一个和 dictionary 长度相同的向量以判断同一文档下该单词是否被计算过。且本次算法不再需要 category_words

3.3.3 test 函数

根据给定的训练方法在给定的测试集上测试训练出的准确度。注意到所谓的求和完全可以通过向量内积实现，而多个向量内积又可以表示为矩阵乘法的形式，因此可以优化算法。

```
def test(data_x, data_y, p_stat, dictionary, p_c, type_train):
    # ...... Some Code ......
    if type_train == 'TF':
    for i, (words, cate) in  enumerate( zip(data_x, data_y)):
        for word in words:
            if word in dictionary:
                word_vec[dictionary[word]][i] += 1
        real[i] = categories[cate]
    res = np.dot(np.transpose(word_vec), np.log(p_stat)) + np.log(p_c)
    count =  len(data_y) - np.count_nonzero(real - np.argmax(res, axis=1))
    elif type_train == 'Bernoulli':
        for i, (words, cate) in  enumerate( zip(data_x, data_y)):
            for word in dictionary:
                if word in words:
                    word_vec[dictionary[word]][i] = 1
            real[i] = categories[cate]
        res = np.dot(np.transpose(word_vec), np.log(p_stat)) + np.dot(1 -
            np.transpose(word_vec), np.log(1 - p_stat)) + np.log(p_c)
        count =
            len(data_y) - np.count_nonzero(real - np.argmax(res, axis=1))
```

```
    # ...... Some Code ......
```

3.4 结果展示

通过简单的修改 main 即可让程序一次得到不同数据清洗方法和计算方法的结果，如下表所示。

accuracy	stemmer		lemmatizer	
	train	test	train	test
TF	92.91%	87.42%	93.51%	87.34%
Bernoulli	92.85%	86.64%	93.42%	87.11%

3.5 附录

3.5.1 源代码

Listing 3: Bayes.py

```
import copy
import os
import nltk
import string
import numpy as np
from nltk.stem import WordNetLemmatizer
from nltk.stem.porter import PorterStemmer

# 种类
categories = {'World': 0, 'Sci/Tech': 1, 'Sports': 2, 'Business': 3}
# 还原方法
types_word = ['stemmer', 'lemmatizer']
# 训练方法
types_train = ['TF', 'Bernoulli']
# 停用词
stopwords =  set(nltk.corpus.stopwords.words('english'))
# 词干提取/词形还原
stemmer, lemmatizer = PorterStemmer(), WordNetLemmatizer()

```

```python

def preprocess(sent, type_word):
    """
    将输入的句子转化为单词词组,并统一为小写、去标点、去停用词、去数字、还原
    """
    # 统一为小写
    sent = sent.lower()
    # 去标点
    remove =  str.maketrans('', '', string.punctuation)
    sent = sent.translate(remove)
    # 转化为单词词组
    words = nltk.word_tokenize(sent)
    # 去停用词
    words = [w for w in words if not (w in stopwords)]
    # 去数字
    words = [w for w in words if not w.isdigit()]
    # 还原:词干提取/词形还原
    if type_word == 'stemmer':
        words = [stemmer.stem(w) for w in words]

    elif type_word == 'lemmatizer':
        words = [lemmatizer.lemmatize(w) for w in words]

    return words


def load(path, type_word):
    """
    path:数据集路径
    根据指定路径读取训练集或测试集
    """
    data_x, data_y = [], []
    with  open(path, 'r') as f:
        lines = f.readlines()
        length =  len(lines)
        for i, line in  enumerate(lines):
```

```python
            tmp = line.split('|')
            data_x.append(preprocess(tmp[1].strip(), type_word))
            data_y.append(tmp[0])
            if i % 1000 == 0:
                print('loading:{}/{}'. format(i, length))

    return data_x, data_y


def words2dic(train_x):
    """
    将训练集中的单词转化为词2id(从0开始)的字典
    """
    dictionary = {}
    i = 0
    for words in train_x:
        for word in words:
            if not word in dictionary:
                dictionary[word] = i
                i += 1
    return dictionary


def train_TF(train_x, train_y):
    # 词汇表
    dictionary = words2dic(train_x)

    # n(w_i in w_c) 词频-种类 矩阵(维度:词汇数x类别数)
    words_frequency = np.zeros(( len(dictionary),  len(categories)), dtype=
        int)

    # n(c,text) 每类下的句总数(维度:类别数x1)
    category_sents = np.zeros( len(categories), dtype= int)

    for words, cate in  zip(train_x, train_y):
        for word in  set(words):
            words_frequency[dictionary[word]][categories[cate]] += 1
```

```
            category_sents[categories[cate]] += 1

    # p(c) (维度:类别数x1)
    p_c = category_sents /  len(train_y)

    # n(w_c) 每类下的词总数(维度:类别数x1)
    category_words = np. sum(words_frequency, 0)

    # p(w_i|c) (维度:词汇数x类别数)
    p_stat = (words_frequency + 1) / (category_words +  len(dictionary))

    return p_stat, dictionary, p_c


def train_Bernoulli(train_x, train_y):
    # 词汇表
    dictionary = words2dic(train_x)

    # n(w_i in w_c) 词频-种类 矩阵(维度:词汇文档数x类别数)
    words_frequency = np.zeros(( len(dictionary),  len(categories)), dtype=
        int)

    # n(c,text) 每类下的句总数(维度:类别数x1)
    category_sents = np.zeros( len(categories), dtype= int)

    for word in dictionary:
        for words, cate in  zip(train_x, train_y):
            if word in words:
                words_frequency[dictionary[word]][categories[cate]] += 1
            category_sents[categories[cate]] += 1

    # p(c) (维度:类别数x1)
    p_c = category_sents /  len(train_y)

    # p(w_i|c) (维度:词汇数x类别数)
    p_stat = (words_frequency + 1) / (category_sents + 2)

```

```python
    return p_stat, dictionary, p_c


def test(data_x, data_y, p_stat, dictionary, p_c, type_train):
    """
    批量数据测试,计算准确率
    """
    # 统计预测正确的数目
    count = 0
    real = np.zeros( len(data_y))
    word_vec = np.zeros(( len(dictionary),  len(data_y)))
    # 计算argmax(...)
    if type_train == 'TF':
        for i, (words, cate) in  enumerate( zip(data_x, data_y)):
            for word in words:
                if word in dictionary:
                    word_vec[dictionary[word]][i] += 1
            real[i] = categories[cate]
        res = np.dot(np.transpose(word_vec), np.log(p_stat)) + np.log(p_c)
        count =
           len(data_y) - np.count_nonzero(real - np.argmax(res, axis=1))
    elif type_train == 'Bernoulli':
        for i, (words, cate) in  enumerate( zip(data_x, data_y)):
            for word in dictionary:
                if word in words:
                    word_vec[dictionary[word]][i] = 1
            real[i] = categories[cate]
        res = np.dot(np.transpose(word_vec), np.log(
            p_stat)) + np.dot(1 - np.transpose(word_vec), np.log(1 - p_stat
                )) + np.log(p_c)
        count =
           len(data_y) - np.count_nonzero(real - np.argmax(res, axis=1))

    print('Accuracy: {}/{} {}%'. format(count,
           len(data_y),  round(100*count/ len(data_y), 2)))


```

```python
if __name__ == '__main__':
    p_stat, dictionary, p_c = [], [], []

    for type_word in types_word:
        train_x, train_y = load(
            os.getcwd()+'\\data\\news_category_train_mini.csv', type_word)
        test_x, test_y = load(
            os.getcwd() + '\\data\\news_category_test_mini.csv', type_word)
        for type_train in types_train:

            if type_train == 'TF':
                p_stat, dictionary, p_c = train_TF(train_x, train_y)
            elif type_train == 'Bernoulli':
                p_stat, dictionary, p_c = train_Bernoulli(train_x, train_y)
            print(type_word, type_train)
            # 训练集上的准确率
            test(train_x, train_y, p_stat, dictionary, p_c, type_train)
            # 测试集上的准确率
            test(test_x, test_y, p_stat, dictionary, p_c, type_train)
```

3.5.2 控制台输出

```
stemmer TF
Accuracy: 9484/10208 92.91%
Accuracy: 2231/2552 87.42%
stemmer Bernoulli
Accuracy: 9478/10208 92.85%
Accuracy: 2211/2552 86.64%
```

(a) stemmer

```
lemmatizer TF
Accuracy: 9546/10208 93.51%
Accuracy: 2229/2552 87.34%
lemmatizer Bernoulli
Accuracy: 9536/10208 93.42%
Accuracy: 2223/2552 87.11%
```

(b) lemmatizer