



山东大学
SHANDONG UNIVERSITY

实验报告

课 程: Data Mining

实验题目: Homework 1

姓 名: 陈昕

学 号: 201814806

班 级: 2018 级计算机学硕班

实验时间: 2018/10/8 – 2018/11/5

一、 实验目的

用 git 创建项目并上传到 GitHub 上，并学会使用 git 的基本命令。

理解课堂上学到的 Vector space model 和 KNN 分类方法，并在 20 Newsgroups 数据集上实践课堂上学到的文本处理、tf-idf、cosine similarity、反向索引及 KNN 分类等知识。

任务：

1. 预处理文本数据集，并且得到每个文本的 VSM 表示。
2. 实现 KNN 分类器，测试其在 20Newsgroups 上的效果。

二、 实验环境

硬件环境：

intel i5-7400 CPU @ 3.00 GHz
8.00 GB RAM

软件环境：

Windows 10 64 位 家庭版
python 3.6.6
Anaconda custom (64-bit)
Jupyter notebook 5.7.0
nltk 3.3.0

三、 实验过程

（我不能保证在报告描述中变量名与实际程序中完全一致。）

准备工作

从 <https://git-scm.com/downloads> 上可以下载到最新的 git 发行版。

在 [GitHub](#) 上创建账号，然后按 start a project 创建项目。

整个实验主程序的入口是 main.ipynb，可以通过 jupyter notebook 运行。

在任务正式开始之前，先从 <http://qwone.com/~jason/20Newsgroups/> 上下载了 [20news-18828.tar.gz](#) 数据集，并用 7zip 将其解压。

任务一：预处理文本数据集，并且得到每个文本的 VSM 表示。

我在这个任务中使用 nltk(Natural Language Toolkit)来进行文本的预处理，将文本转化为词语集合，并自己编写了一个类 VectorSpaceModel 来把词语集合转换为 VSM 表示。

根据面向对象的单一职责原则，VectorSpaceModel 类没有文本预处理功能，文本预处理过程在 main.ipynb 中进行。

在 main.ipynb 中的文本预处理和转化为 VSM 的过程如下：

首先，用 os 模块获取 20Newsgroups 数据集的所有文件路径，filepaths。

将 filepaths 按照 8: 2 的比例划分为训练集 training_set_fns 和测试集 test_set_fns (fn 是 file name 的缩写)。

然后，用文件读取函数 open() 打开 training_set_fns 中的所有文档。文档的编码方式为 latin1。

Nltk 包中提供了 stopwords, tokenizer 和 stemmer。先用 RegexpTokenizer 从文档中取出所有单词，然后把所有词转换为小写并用 stemmer 取词干，最后过滤掉所有属于 stopwords 的停词。

对测试集中的所有文档进行上述处理，得到一个分词后的文件列表，命名为 tokenized_docs。这样就结束了文本预处理过程。

VSM 表示的各种功能是在 VectorSpaceModel 类里实现的。

在 main.ipynb 里，遍历 tokenized_docs，对每个文档用 VectorSpaceModel 包装，并将其中的词语加入计算 DF，并在遍历结束后计算 IDF。完成后即可计算各个 VSM 里每个词的 tf-idf 权重，再把向量转为单位向量。

我实现了 sub-linear 和 maximum 两种 tf normalization 方法，还可以选择直接使用 tf。这些选择定义在一个枚举类 TF_Scale 中。

在 main.ipynb 中，我选择了使用 maximum 方法来 normalize 词频。

VectorSpaceModel.py 文件中：

包括两个类 TF_Scale 和 VectorSpaceModel。

枚举类 TF_Scale 里定义了 TF 的 3 种 normalization 方式：RAW 代表不 normalization，SUB_LINEAR 表示使用 $tf(t, d) = \begin{cases} 1 + \log c(t, d), & \text{if } c(t, d) > 0 \\ 0, & \text{otherwise} \end{cases}$ ，MAXIMUM 表示使用 $tf(t, d) = \alpha + (1 - \alpha) \frac{c(t, d)}{\max_t c(t, d)}$ 。

类 VectorSpaceModel 的类属性包括：

- rawDF (整个词典和文档频率的字典)
- _IDF (值为 IDF 值的字典)
- alpha (进行 Maximum TF scaling 时的 α)

对象属性包括：

- rawTF (基于内置 Counter 的词频统计字典)
- _maxCountInTF (进行 Maximum TF scaling 时的最高词频)
- vector (存放存在于文档中的词以及它们的权重)。

类 `VectorSpaceModel` 的方法都比较简单，包括：

- `calTF` (根据 `TF_Scale` 所指定的方法计算 `tf`)
- `calIDF` (计算 `IDF`)
- `accumulateDocumentFrequency` (添加新文档信息到 `rawDF`)
- `calWeight` (计算 `vector`)
- `normalize` (把向量单位化)
- `getTerms` (返回文档中包含的所有词)
- `getCorpus` (返回全局的所有词语)
- `dot` 和 `dotProduct` (用两个标准化后的 `VSM` 进行点乘)。

我没有选择使用矩阵来记录各 `VSM` 表示，因为一个文档基本不可能拥有语料库中的所有词，用矩阵表示有极大可能得到稀疏的矩阵，不必要的 0 会占用过多的内存空间。

The screenshot shows a Jupyter Notebook interface with a JupyterLab window on the left and a Windows Task Manager window on the right. The Jupyter Notebook contains the following code:

```
In [3]: # generate file path of training set and testing set
subdirs = os.listdir(PATH)
training_set_fns = []
test_set_fns = []
for dir_ in subdirs:
    files = os.listdir(PATH+dir_)
    splitting_pos = math.floor(len(files)*0.8)
    training_set_fns.extend(os.path.join(dir_, fn) for fn in files[:splitting_pos])
    test_set_fns.extend(os.path.join(dir_, fn) for fn in files[splitting_pos:])

In [4]: print('The training set contains %d files' % len(training_set_fns))
print('The testing set contains %d files' % len(test_set_fns))

The training set contains 15056 files
The testing set contains 3772 files

In [5]: # read documents of training set
training_docs = []
for fn in training_set_fns:
    with open(PATH+fn, encoding='latin1') as f:
        training_docs.append(f.read())

In [6]: # preprocess training docs
stop_words = set(stopwords.words('english'))
def filter_stop_words(doc):
    return filter(lambda v: v not in stop_words, doc)

tokenized_docs = []
tokenizer = RegexpTokenizer(r'\w+')
stemmer = SnowballStemmer('english', ignore_stopwords=True)
for doc in training_docs:
    word_tokens = tokenizer.tokenize(doc) # Tokenization
    word_stems = [stemmer.stem(w.lower()) for w in word_tokens if w not in stop_words]
    word_without_stopwords = filter_stop_words(word_stems)
    tokenized_docs.append(word_without_stopwords)

In [7]: # get the VSM representation of each training document
vsms = []
for doc in tokenized_docs:
    vsm = VectorSpaceModel(doc)
    VectorSpaceModel.accumulateDocumentFrequency(vsm.getTerms())
    vsms.append(vsm)

VectorSpaceModel.calIDF()
for vsm in vsms:
    vsm.calWeight(TF_Scale.MAXIMUM)
    vsm.normalize()

In [8]: len(vsms)
Out[8]: 15056
```

The Windows Task Manager window shows the following table of running processes:

名称	状态	CPU	内存	磁盘	网络	GPU	GPU 引擎
TeamViewer 13 (32 位)		3.7%	860.0 MB	0 MB/秒	0 Mbps	0.6%	GPU 0 - 3D
Python (4)		0%	618.4 MB	0 MB/秒	0 Mbps	0%	
Python		0%	551.4 MB	0 MB/秒	0 Mbps	0%	
Python		0%	53.5 MB	0 MB/秒	0 Mbps	0%	
Python		0%	7.1 MB	0 MB/秒	0 Mbps	0%	
控制台窗口主进程		0%	6.3 MB	0 MB/秒	0 Mbps	0%	
Google Chrome (21)		0%	595.8 MB	0 MB/秒	0 Mbps	0%	GPU 0 - 3D
Adobe Acrobat DC (32 位) (3)		0%	93.9 MB	0 MB/秒	0 Mbps	0%	
Antimalware Service Executable		0.2%	89.7 MB	0.1 MB/秒	0 Mbps	0%	
µTorrent (32 位) (3)		7.9%	77.2 MB	6.9 MB/秒	35.4 Mb	0%	
Windows 资源管理器		0%	61.0 MB	0.1 MB/秒	0 Mbps	0%	
Microsoft Word		0%	48.8 MB	0 MB/秒	0 Mbps	0%	
TeamViewer 13 (32 位)		0%	41.3 MB	0 MB/秒	0 Mbps	0%	
TIM (32 位)		0%	41.1 MB	0 MB/秒	0 Mbps	0%	
桌面窗口管理器		0.4%	39.0 MB	0 MB/秒	0 Mbps	0.1%	GPU 0 - 3D

如图所示，在用 `VectorSpaceModel` 的对象列表来替代矩阵的情况下，尽管列表里包含 15056 个文档且仅使用 `stopwords` 过滤，内存消耗也只需要约 600MB，不会撑爆 8G 内存。

而用字典进行向量单位化和点乘也是节省时间与内存的，因为如果一个词的词频为 0，那么它不会影响这两个过程的结果，反而会花费不必要的时间。

任务二：实现 KNN 分类器，测试其在 20Newsgroups 上的效果。

我的 KNN 分类器实现在 `KNNClassifier.py` 中，测试代码依旧写在 `main.ipynb` 中。

在 `main.ipynb` 中的进行的分类过程如下：

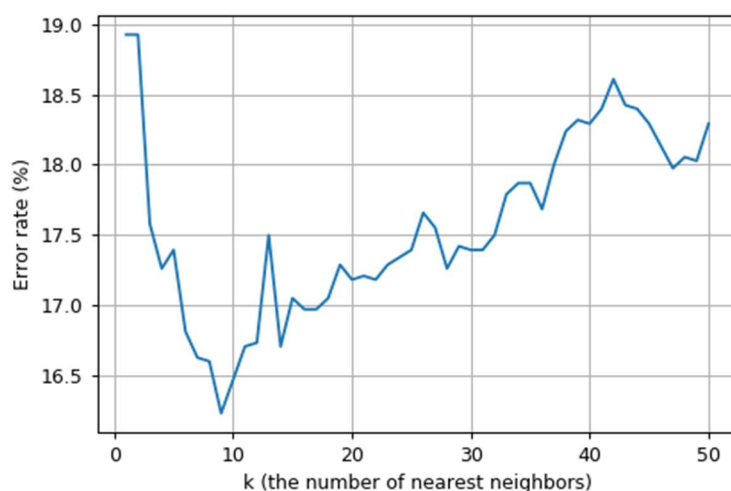
首先，读取测试集 `test_set_fns`，并转化为 VSM 表示。

接着构建 KNN 分类器。训练集的标签 `labels` 来自它们的文件夹名，可通过 `os.path.dirname` 得到。用训练集数据和 `labels` 就可以构建 `KNNClassifier`，而 `train` 函数中构建了反向索引。

然后，以从 1 开始的 50 个 `k` 作为参数，用分类器对测试集进行分类。

在分类完成后，对比分类器生成的 `label` 和测试集的实际 `label`（也就是文件夹名），计算错误率。

通过 `numpy` 包的方法，找到最佳 `k` 值，并用 `matplotlib` 画出错误率曲线，如下图所示：



结果显示在 `k=9` 时错误率最小，约 16.2%。之后虽然有些波动，但趋势是向上的，也就是说 `k` 并不是越大越好。

KNNClassifier.py 文件中：

该文件只包括 `KNNClassifier` 一个类。

类 `KNNClassifier` 中，没有类属性。

对象属性包括：

- `vsms`（训练数据集的 `vsm` 列表）
- `labels`（每个数据对应的标签）
- `invertedList`（作为反向索引的字典）。

对象方法包括：

- `buildInvertedList`（构建反向索引）
- `train`（内部调用 `buildInvertedList` 方法）
- `classify`（使用 `inverted list` 和 `heap` 来加速找到 `k` 近邻，用 `Counter` 求出 `k` 近邻中最多的 `label`，作为分类结果）

`train` 方法的存在是为了我想象中的 `Classifier` 接口，它应该有 `train` 和 `classify` 两个方法，所有 `client` 都应该只需要调用这两个方法，减小调用方的成本。

`classify` 方法是分类的主方法，要被调用多次，所以我使用反向索引和 `heapq` 模块提供的 `nlargest` 方法来进行加速。

四、 实验结论

在本次实验中，我动手实现了课堂上学到的文本预处理、vector space model 构建和 KNN 分类，感觉对这些知识点的理解更深入了。

在读取文件时，我发现 utf8 有些无法解析的字符，在网上查找后发现换用 latin1 编码就能解析全部文件了。这是因为基本上各种编码都兼容 ASCII 字符，而各种文本编辑器优先把只包含 ASCII 字符的文件是 utf8 编码的，但这样在碰到 latin1 字符时就会出现解析错误。

我还发现了 nltk 这个强大的自然语言处理工具，用它进行了分词、取词干、过滤停词的操作。它还有词性标记、解析、获取语义等功能，但我在本次实验中没有用到。

在实现 vector space model 时，我实现了 sub-linear 和 maximum 两种 tf normalization 方式，并用 dict 作为 vector，且实现了 vector 点乘。

在进行 knn 分类时，我对从 1-50 的 k 进行了测试，最高正确率约 84%，此时 k 取 9。错误率先下降再上升，出现了明显的转折点，所以存在最佳的 k 值。