第 7 章 文本特征提取与模型复用

目录

第 7 章 文本特征提取与模型复用

在前面几章的示例介绍中，我们所用到的数据集都是已经处理好的数据，换句话说这些数据集的每个特征维度都已经转换成了可用于计算的数值形式。但是在实际的建模任务中，我们拿到的数据集可能并不是这样的形式。例如接下来要完成的一个任务，对中文垃圾邮件进行分类。

7.1 词袋模型

例如，对于下面这样一个邮件（样本），应该采用什么样的方式来对其进行量化呢？同时，我们知道在建模过程中需要保证每个样本的特征维度数都一样，但是这里每一封邮件的长度却并不同，这又该怎么处理呢？接下来，笔者就开始介绍机器学习中的第一种文本向量化方法——词袋模型（Bag Of Words, BOW）。

“股权分置已经牵动全国股民的心，是机会？是陷阱？是机会您应该如何把握？是陷阱您应该如何规避？ 请点击此网址索取和-讯专家团针对股权分置的操作指导：http://www.spam.com/gwyqxjqxx/”

7.1.1 理解词袋模型

什么是词袋模型呢？其实词袋模型这个叫法非常形象，凸出了模型的核心思想。所谓词袋模型就是，首先将训练样本中所有不重复的词放到这个袋子中构成一个词表（字典）；然后再以这个词表为标准来遍历每一个样本，如果词表中对应位置的词出现在了样本中，那么词表对应位置就用 1 来表示，没有出现就用 0 来表示；最后，对于每个样本来说都将其向量化成了一个和词表长度一样的只含有 0 和 1 的向量。

如图 7-1 所示为一个直观的词袋模型转换示意图。左边为原始数据集（包含两个样本），中间为词表，右边为向量化的结果。

图 7-1 词袋模型原理图

其中[1 1 0 1]的含义就是，样本“机器学习 是 人工智能 的 子集”中，有 3 个词都出现在了词表当中，分别是“机器学习”、“人工智能”和“子集”。

具体步骤可以总结为以下 3 步：

1) 文本分词

首先需要将原始数据的每个样本都进行分词处理（英文语料可以跳过这步）；

2) 构造词表

然后在所有的分词结果中去掉重复的部分，保证每个词语只出现一次，且同时要以任意一种顺序来固定词表中每个词的位置。

3) 文本向量化

遍历每个数据样本，若词表中的词出现在该样本中，则对应位置为 1，没出现则为 0。

在如图 7-1 中，对样本“机器学习 是 人工智能 的 子集”来说，其中有 3 个词都出现在了词表中，所以词表中每个词的对应位置为 1，而‘深度学习’这个词并没有出现在样本中，所以对应位置为 0。

可以看出，向量化后每个样本特征维度的长度都和词表长度相同（图 7-1 中为 4）。虽然这样做的好处是词表包含了样本中所有出现过的词，但是这却很容易导致维度灾难。因为通常一个一般大小的中文数据集，都可能会出现数万个词语（而这意味着转化后向量的维度也有这么大）。所以实际处理中，在分词结束后通常还会进行词频统计这一步，即统计每个词在数据集中出现的次数，然后只选择其中出现频率最高的前 K 个词作为最终的词表。最后，通常也会将一些无意义的虚词，即停用词（Stop Words）去掉，例如“的，啊，了，”等。

7.1.2 文本分词

经过 7.1.1 节的介绍可以知道，向量化的第一步是需要对文本进行分词。下面笔者将开始介绍一款常用的开源分词工具 jieba。当然，使用 jieba 库的前提是先要安装，读者可以先进入到对应的虚拟环境中，然后通过命令 pip install jieba 进行安装。

这里先用下面这段文本来进行分词处理并做词频统计：

央视网消息：当地时间 11 日，美国国会参议院以 88 票对 11 票的结果通过了一项动议，允许国会“在总统以国家安全为由决定征收关税时”发挥一定的限制作用。这项动议主要针对加征钢铝关税的 232 调查，目前尚不具有约束力。动议的主要发起者——共和党参议员鲍勃·科克说，11 日的投票只是一小步，他会继续推动进行有约束力的投票。

可以看到，这段文本当中还包含了很多标点符号和数字，显然暂时不需要这些内容，所以在分词的时候可以通过正则表达式来进行过滤。同时，jieba 库分别提供了两种分词模式来应对不同场景下的中文分词，下面分别进行介绍。完整代码见 Chapter07/01_cut_words.py 文件。

1) 普通分词模式

普通分词模式指的就是按照常规的分词方法，将一个句子分割成多个词语的组成形式，代码如下：

```
1 import jieba,re
2 def cutWords(s, cut_all=False):
3     cut_words = []
4     s = re.sub("[A-Za-z0-9\: \·\—\, \。\“ \”]", "", s)
5     seg_list = jieba.cut(s, cut_all=cut_all)
6     cut_words.append(" ".join(seg_list))
7     print(cut_words)
```

在上述代码中，第 4 行代码作用是将所有字母、数字、冒号、逗号、句号等过滤掉；第 5-6 行用来完成分词处理的过程，当 cut_all = False，表示普通分词模式。根据上述代码分词结束后便能看到如下所示的结果：

```
['央视网 消息 当地 时间 日 美国国会参议院 以票 对票 的 结果 通过 了 一项 动议 允许 国会 在 总统 以 国家 安全 为 由 决定 征收 关税 时 发挥 一定 的 限制 作用 这项 动议 主要 针对 加征 钢铝 关税 的 调查 目前 尚 不 具有 约束力 动议 的 主要 发起者 共和党 参议员 鲍勃 科克 说 日 的 投票 只是 一 小步 他会 继续 推动 进行 有 约束力 的 投票']
```

但是，对于有的句子来说可以有不同的分词方法，例如“美国国会参议院”这段描述，既可以分成“美国 国会 参议院”，也可以是“美国国会 参议院”，甚至都可以直接是“美国国会参议院”，不同的人可能有不同的切分方式。因此， jieba 还提供了另外一种全分词模式。

2) 全分词模式

当把上面代码中 cut_all 设置为 True 后，便可以开启全分词模式，分词后的结果如下：

```
['央视 央视网 视网 消息 当地 时间 日 美国 美国国会 美国国会参议院 国会 参议 参议院 议院 以票 对 票 的 结果 通过 了 一项 动议 允许 许国 国会 在 总统 以 国家 家安 安全 为 由 决定 征收 关税 时 发挥 一定 的 限制 制作 作用 这项 动议 主要 针对 加征 钢 铝 关税 的 调查 目前 尚 不 不具 具有 约束 约束力 动议 的 主要 发起 发起者 共和 共和党 党参 参议 参议员 议员 鲍 勃 科克 说 日 的 投票 只是 一小 小步 他 会 继续 推动 进行 有 约束 约束力 的 投票']
```

可以看出对于有的句子，分词后的结果确实看起来结结巴巴的，而这就是全分词模式的作用。在分词结束后，下面就开始对分词结果进行词频统计并构造词表。

7.1.3 构造词表

上面介绍到，分词后通常还会先进行词频统计，以选取出现频率最高的前 K 个词来构造词表。对词频统计需要用到另外一个包 collection 中的 Counter 计数器（如果没有安装可自行安装，命令为 pip install collection）。但是需要注意的是，像上面那样分词后的形式并不能做词频统计，因为 Counter 是将 list 中的一个元素视为一个词，所以要对上面的代码略微进行修改。完整代码见 Chapter07/01_cut_words.py 文件。

```
1 def wordsCount(s):
2     cut_words = ""
3     s = re.sub("[A-Za-z0-9\：\·\—\，\。\“ \”]", "", s)
4     seg_list = jieba.cut(s, cut_all=False)
5     cut_words += (" ".join(seg_list))
6     all_words = cut_words.split()
7     c = Counter()
8     for x in all_words:
9         if len(x) > 1 and x != '\r\n':
10            c[x] += 1
```

在上述代码中，前 6 行便是先用来对文本进行分词处理；然后再通过后面 4 行代码来完成词频统计。经过上述代码的词频统计后，便可以取前 K 个词来构造词表，代码如下：

```
1     #此处接上面代码
2     vocab = []
3     print('\n 词频统计结果：')
4     for (k, v) in c.most_common(5):  # 输出词频最高的前 5 个词
5         print("%s:%d" % (k, v))
6         vocab.append(k)
7     print("词表：", vocab)
```

这样便能得到前 K（这里取的 5）个词构成的词表，结果如下：

```
词频统计结果：
动议:3
关税:2
主要:2
约束力:2
投票:2
词表： ['动议', '关税', '主要', '约束力', '投票']
```

7.1.4 文本向量化

通过上面的操作，便能得到一个最终的词表（Vocabulary）。最后一步的向量化工作则是遍历每个样本，查看词表中每个词是否出现在当前样本中，如果出现则词表对应维度用 1 表示，没出现用 0 表示。完整代码见 Chapter07/02_vectorization.py 文件，关键代码如下所示：

```
1 def vetorization(s):
2     #此处接文本分词和词频统计代码
3     x_vec = []
4     for item in x_text:
5         tmp = [0] * len(vocab)
6         for i, w in enumerate(vocab):
7             if w in item:
8                 tmp[i] = 1
9         x_vec.append(tmp)
10    print("词表：", vocab)
11    print("文本：", x_text)
12    print(x_vec)
```

在上述代码中，第 3 行里 x_text 表示原始文本分词后的结果；第 4 行里 tmp 表示先初始化一个长度为词表长度的全 0 向量；第 6-8 行表示开始遍历每一句文本中的每一个词，判断其是否存在于词表中，如果存在则将 tmp 向量对应处置为 1。

这样，根据 vetorization 函数便能够对输入的文本进行向量化表示，结果如下：

```
s=['文本分词工具可用于对文本进行分词处理','常见的用于处理文本的分词处理工具有很多']
vetorization(s)
词表： ['文本', '分词', '处理', '工具', '用于', '进行', '常见', '很多']
文本：[['文本','分词','工具','可','用于', '对','文本','进行', '分词', '处理'], ['常见', '的', '用于', '处理', '文本', '的', '分词', '处理', '工具', '有','很多']]
[[1, 1, 1, 1, 1, 1, 0, 0], [1, 1, 1, 1, 1, 0, 1, 1]]
```

从上面的结果可以看出，这里选择了出现频率最高的前 8 个词来构造词表，然后得到了每个样本的向量化表示。

到此，笔者就介绍完了第一种基本的文本向量化表示方法，即判断样本中的每一个词是否出现在词表中，如果出现则词表对应位置就用 1 来表示，没有包含则用 0 表示。最终就会得到一个仅包含 0，1 的向量来表示这一样本。但这是这样做的弊端之一就是没有考虑到词的出现频率，即不管一个词出现了多少次，最后都仅仅用 1 来表示其出现过。但在一些场景下，词频又是十分重要的。因此，接下来再来看另外一个种同时考虑词频的词袋表示模型。

7.1.5 考虑词频的文本向量化

如图 7-2 所示，最上面为原始样本，中间为词表，最下边为两种词袋模型的表示结果。其中左边的表示方法就是我们在上面介绍的第一种文本表示方法，它只考虑词表中的单词是否出现，而不关心出现次数；而右边的表示方法同时还考虑到了每个词的出现频率。

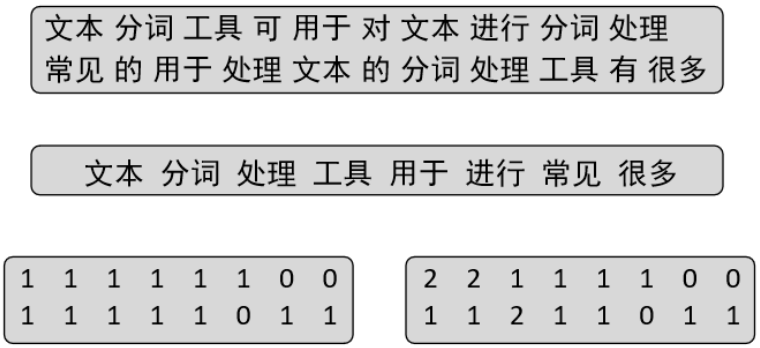


图 7-2 考虑词频的词袋模型原理图

因此，根据这一原理只需要将 7.1.4 节中的代码稍作修改即可实现这一结果。完整代码见 Chapter07/03_vectorization_with_freq.py 文件，关键代码如下：

```
1 def vectorization_with_freq(s):
2     #此处接文本分词和词频统计代码
3     x_vec = []
4     for item in x_text:
5         tmp = dict(zip(vocab, [0] * len(vocab)))
6         for w in item:
7             if w in vocab:
8                 tmp[w] += 1
9         x_vec.append(list(tmp.values()))
10    print("词表：", vocab)
11    print("文本：", x_text)
12    print(x_vec)
```

在上述代码中，第 5 行用来初始化一个字典，其 key 为词表中的每一个词，value 初始化为 0 表示每个词出现的次数为 0；第 6-8 行用来遍历样本中的每一个词，如果其出现在词表中，则对字典中对应词的计数值加 1；最后第 9 行用来取字典对应的所有 value 值做这条文本的向量化表示。

这样，根据 vetorization_with_freq 函数便能够对输入的文本进行向量化表示，结果如下：

```
s = ['文本分词工具可用于对文本进行分词处理', '常见的用于处理文本的分词处理工具有很多']
vectorization_with_freq(s)
词表： ['文本', '分词', '处理', '工具', '用于', '进行', '常见', '很多']
文本： [['文本', '分词', '工具', '可', '用于', '对', '文本', '进行', '分词', '处理
'], ['常见','的','用于', '处理','文本','的','分词','处理','工具','有','很多']]
[[2, 2, 1, 1, 1, 1, 0, 0], [1, 1, 2, 1, 1, 0, 1, 1]]
```

这样，便得到了考虑词频的文本向量化表示。不过，其实这一方法在 sklearn 中已经实现了。接下来就通过 sklearn 中的方法再进行一次示例。在 sklearn 中，可以通过导入 CountVectorizer 这一类方法来完成上诉步骤，代码如下：

```
from sklearn.feature_extraction.text import CountVectorizer
count_vec = CountVectorizer(max_features=8)
x = count_vec.fit_transform(s).toarray()
vocab = count_vec.vocabulary_
vocab = sorted(vocab.items(),key=lambda x:x[1])
```

在上述代码中，第 2 行里的 max_features=8 表示取频率最高的前 8 个词来构造词表；最后 2 行代码分别用来获得词表，以及将词表进行排序。

这样，根据上述代码便能够对文本进行向量化表示，结果如下：

```
[('分词', 0), ('处理', 1), ('工具', 2), ('常见', 3), ('很多', 4), ('文本', 5),
('用于', 6), ('进行', 7)]
[[2 1 1 0 0 2 1 1]
 [1 2 1 1 1 1 1 0]]
```

可以发现，通过 CountVectorizer 类得到的文本向量与上面我们自己编写代码输出的结果存在着一点差别。细心的读者可以发现，导致这一差别的主要原因便是词表的顺序不一样。由于最后的文本向量化形式会依赖于每个词在词表中的位置顺序，所以根据不同顺序的词表最后得到的向量必定存在着不同。但这本质上并没有什么不同，只要词表一样那两者得到的向量表示都是等价的。

7.1.6 小结

在本节中，笔者首先介绍了第一种将文本转化为向量的词袋模型；接着介绍了一款常用的中文分词工具 jieba 库，并演示了如何通过 jieba 进行分词处理并进行词频统计；然后介绍了如何实现词袋模型的最后一步向量化表示；最后还介绍了另外一种词袋模型表示方法，该方法同时考虑到了词语的出现频率，并且在长文本的表示中用得较多。

7.2 基于贝叶斯算法的垃圾邮件分类

接下来，笔者就采用第 2 种词袋模型表示方法通过朴素贝叶斯算法来对垃圾邮件进行分类。下面用到的是一个中文的邮件分类数据集，包含垃圾邮件和非垃圾邮件两类，即一个 2 分类任务。其中 ham_5000.utf8 和 spam_5000.utf8 这两个文件中分别包含有 5000 封正常邮件和垃圾邮件，文件中每行分别表示一封邮件，示例如下

“我的意中人是一个盖世英雄，有一天他会踩着七色的云彩来娶我，我猜中了前头，可是我猜不着这结局”世间一切美好都有有效期限吧，坦然面对，接受幸福的彩排。

总的来说，要完成这一文本分类任务，首先需要载入原始文本并对其中的每一个样本进行分词处理；接着通过上面介绍的 CountVectorizer 类来完成文本的向量化表示，并制作完成每个样本对应的类别构成一个完整的数据集；最后再来根据朴素贝叶斯算法来完成最后的分类任务。完整代码见 Chapter07/04_bag_of_word_cla.py 文件。

7.2.1 载入原始文本

首先需要完成的便是编写一个用于载入本地文本并同时进行分词的函数，代码如下：

```
def load_data_and_cut(file_path='./data/ham_100.utf8'):
    x_cut = []
    with open(file_path, encoding='utf-8') as f:
        for line in f:
            line = line.strip('\n')
            seg_list = jieba.cut(clean_str(line), cut_all=False)
```

```
            tmp = " ".join(seg_list)
            x_cut.append(tmp)
    return x_cut
```

在上述代码中，第 3 行用来打开一个本地的文本文件，并采用 utf-8 的编码格式进行读取；第 4 行用来读取文本中的每一行文本；第 5 行用来去掉每行文本末尾的换行符；第 6-7 行用来对文本进行清洗以及分词处理，其中函数 clean_str 的作用是去掉一个字符串中的所有非中文字符；最后再返回处理好的结果。

7.2.2 制作数据集

在完成原始文本的载入后，就需要根据样本的数量来分别构造垃圾邮件和正常邮件对应的类别标签，同时再对文本进行向量化表示，代码如下：

```
def get_dataset(top_k_words=1000):
    x_pos = load_data_and_cut(file_path='./data/ham_5000.utf8')
    x_neg = load_data_and_cut(file_path='./data/spam_5000.utf8')
    y_pos, y_neg = [1] * len(x_pos), [0] * len(x_neg)
    x, y = x_pos + x_neg, y_pos + y_neg
    X_train, X_test, y_train, y_test = \
        train_test_split(x, y, test_size=0.3, random_state=42)
    count_vec = CountVectorizer(max_features=top_k_words)
    X_train = count_vec.fit_transform(X_train)
    X_test = count_vec.transform(X_test)
    # print(len(count_vec.vocabulary_)) # 输出词表长度
    return X_train, X_test, y_train, y_test
```

在上述代码中，第 2-3 行分别用来载入原始的正常邮件和垃圾邮件；第 4-5 行分别用来构造样本标签，以及将两个类别的特征输入和标签放到一起；第 6-7 行用来将原始数据进行打乱，同时划分成训练集与测试集两个部分；第 8-10 行则是先实例化类 CountVectorizer，然后通过训练集来构造词表并对训练集进行向量化，最后再用通过训练集得到的词表来对测试集进行向量化；第 12 行则是将最后的结果进行返回。

这里特别需要注意的地方就是，一定要先划分数据，然后再进行向量化。这就同第 4 章中介绍的标准化流程一样，一定要用在训练集上得到的标准化参数（指词表）来对测试集进行标准化（指向量化）。

7.2.3 训练模型

在制作完成数据集后，就可以定义朴素贝叶斯模型，然后进行训练与预测，代码如下：

```
from sklearn.naive_bayes import MultinomialNB
def train(X_train, X_test, y_train, y_test):
    model = MultinomialNB()
    model.fit(X_train, y_train)
    y_pre = model.predict(X_test)
    print(classification_report(y_test, y_pre))
```

在上述代码中，第 3 行用来定义一个多项式的朴素贝叶斯模型；第 4-5 行用来训练模型和对测试集进行测试；最后可以得到如下所示的评估结果：

```
1                precision    recall  f1-score   support
2     accuracy                           0.96      3001
3    macro avg       0.96      0.96      0.96      3001
4 weighted avg       0.96      0.96      0.96      3001
```

同时，在实例化类 MultinomialNB 时，还可以通过其对应的模型参数 alpha 来设定平滑系数的取值。在默认情况下，alpha=1 即拉普拉斯平滑。

7.2.4 复用模型

在实际的运用环境中，不可能每次在对新数据进行预测时都从头开始训练一个模型。通常，模型在第 1 次训练完成后都会被保存下来。只要后续不需要再对模型做任何改动，那么在对新数据进行预测时，只需要载入已有的模型进行复用即可[①]。

1) 保存模型

首先需要定义一个函数来对传入的模型进行保存，代码如下：

```
1 import joblib
2 def save_model(model, dir='MODEL'):
3     if not os.path.exists(dir):
4         os.mkdir(dir)
5     joblib.dump(model, os.path.join(dir, 'model.pkl'))
```

在上述代码中，第 3-4 行用来判断当前是否存在'MODEL'这个目录，如果不存在则创建；第 5 行用来将传入的模型以 model.pkl 的名称保存到'MODEL'这个目录中。此时，只需要在 7.2.3 中第 5 行代码前插入代码 save_model(model, dir='MODEL')即可完成对模型的保存。

2) 复用模型

在复用模型之前，需要先定义个函数来对已有的模型进行载入，代码如下：

```
1 def load_model(dir='MODEL'):
2     path = os.path.join(dir, 'model.pkl')
3     if not os.path.exists(path):
4         raise FileNotFoundError(f"{path} 模型不存在，请先训练模型！")
5     model = joblib.load(path)
6     return model
```

在上述代码中，第 2-4 行用来判断给定的路径中是否存在一个名为 model.pkl 的模型文件，如果不存在则进行提示；第 5-6 行用来返回载入后的模型。此时，只要已存在的模型载入成功，那么便可以直接用起来对新数据进行预测，代码如下：

```
1 def predict(X):
2     model = load_model()
3     y_pred = model.predict(X)
4     print(y_pred)
```

到此，对于文本数据的数据预处理过程、向量化过程、以及模型的训练和复用过程就介绍完了。不过，如果读者这里稍微举一反三的话就会发现，在保存模型的时候不仅仅是要对最后的分类或者回归模型进行保存，最开始的数据集预处理模型同样需要保存，例如这里的

① Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

CountVectorizer 模型。因为新输入的数据一般也都是原始数据，需要对其进行相应的标准化（这里是向量化）处理，同时还必须使用通过训练集得到的参数来对新数据进行标准化，所以也需要对标准化时的模型进行保存。

7.2.5 小结

在这节中，笔者首先以一个真实的垃圾邮件数据集为例，详细介绍了如何通过 sklearn 中的朴素贝叶斯模型（MultinomialNB）来完成文本的分类任务，包括载入原始本文数据、制作数据集、划分数据集等；然后还介绍了如何通过 joblib 模块来完成模型的复用；最后还分析了复用模型时不仅仅需要保存最后的回归或者分类模型，同时还应该保存数据预处理过程中所用到的所有模型。

7.3 考虑权重的词袋模型

在 7.1 节中，笔者介绍了两种基本的用于文本表示的词袋模型表示方法，两者之间的唯一区别就是一个考虑的词频而另外一个没有考虑。下面我们再介绍另外一种应用更为常见和广泛使用的词袋模型表示方式——TF-IDF 表示方法。

7.3.1 理解 TF-IDF 模型

之所以陆续的会出现不同的向量化表示形式，其最终目的都只有一个，即尽可能准确的对原始文本进行表示。TF-IDF 为词频-逆文档频率（Term Frequence - Inverse Document Frequence）的简称。首先需要明白的是 TF-IDF 实际上是 TF 与 IDF 两者的乘积。同时，出现 TF-IDF 的原因在于，通常来说在一个样本中一次词出现的频率越高，其重要性应该对应越高，即考虑到词频对文本向量的影响；但是如果仅仅只是考虑到这一个因素则同样会带来一个新的弊端，即有的词不只是在某个样本中出现的频率高，其实它在整个数据集中的出现频率都很高，而这样的词往往也是没有意义的。因此，TF-IDF 的做法是通过词的逆文档频率来加以修正调整。

7.3.2 TF-IDF 计算原理

TF-IDF 的计算过程总体上可以分为两步，先统计词频，然后计算逆文档频率，最后将两者相乘得到 TF-IDF 值[①]。

1) 统计词频

$$TF = \text{某个词在该样本中出现的次数} \tag{7-1}$$

2) 计算逆文档频率

$$IDF = \log\left(\frac{\text{总的样本数}}{\text{包含有该词的样本数+1}}\right) \tag{7-2}$$

其中 log 表示取自然对数。

根据式(7-2)可以发现，如果一个词越是常见，那么对应的分母就越大，逆文档频率就越小。分母之所以要加 1，是为了避免分母为 0 时（当使用自定义词表时）的平滑处理。这就是最原始的 IDF 计算方式。不过这种做法的一个瑕疵就是，当所有样本中都含有某个词的时候，计算出来的 IDF 就为负数。因此，sklearn 在实现 IDF 计算时采用了另外一种平滑处理的方式

[①] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

$$IDF = \log\left(\frac{\text{总的样本数+1}}{\text{包含有该词的样本数+1}}\right)+1 \tag{7-3}$$

这样就同时避免了上面所出现的两种情况。在后面的计算示例中，笔者也将采用式(7-3)中的公式来计算 IDF 值。

3) 计算 TF-IDF

$$TF\text{–}IDF = TF \times IDF \tag{7-4}$$

最后，根据计算得到的 TF 和 IDF 值便可以根据式(7-4)来计算 TF-IDF 值。同时，对于数据集中的每一个词都能计算得到对应的 TF-IDF 值，最后将所有的值组合成一个矩阵便得到了文本的向量化表示。

注意：对于样本中的每一个词，如果其没有出现在词表中，那么对应的 TF-IDF 值为 0。

7.3.3 TF-IDF 计算示例

现在假设有如下 4 个样本（每个样本为列表中的一个元素）

```
corpus = ['this is the first document',
          'this document is the second document',
          'and this is the third one',
          'is this the first document']
```

同时，其对应的词表为

```
vocabulary = ['this', 'document', 'first', 'is', 'second', 'the',
              'and', 'one']
```

1) 统计词频

首先，根据已知的样本和词表，可以得到如下所示的一个词频统计矩阵

```
[[1 1 1 1 0 1 0 0]
 [1 2 0 1 1 1 0 0]
 [1 0 0 1 0 1 1 1]
 [1 1 1 1 0 1 0 0]]
```

其中矩阵中的每一行表示对应样本中，词表中各个词出现的次数。例如第 1 行中的前 4 个 1 表示词表中的前 4 个词均在样本“this is the first document”中出现；第 5 个 0 表示词表中的“second”并没有在第一个样本中出现；第 6 个 1 表示词表中的“the”出现在第 1 个样本中；最后两个 0 表示词表中“and”和“one”这两个词也没有出现在第一个样本中。词频矩阵中的其它 3 行也是同理。

2) 计算逆文档频率

由式(7-3)可知，对于词表中的每一个词，根据其在整个样本中的出现情况都可以计算得到一个 IDF 值。因此，对于整个词表来说，可以计算得到如下所示的一个 IDF 向量

```
[1.    1.223    1.510    1.    1.916    1.    1.916    1.916]
```

例如对于单词“document”来说它出现在了 3 个样本中，因此其计算过程为

$$\log\left(\frac{4+1}{3+1}\right)+1 \approx 1.223 \tag{7-5}$$

3) 计算 TF-IDF

在计算得到样本中每一个词的词频，以及词表中每一个词的 IDF 值后，便可以根据(7-4)计算得到样本中每一个词的 TF-IDF 值，最终得到如下所示的 TF-IDF 权重矩阵

```
[[1.    1.223  1.510  1.   0.   1.   0.     0.   ]
 [1.    2.446  0.     1.   1.916  1.   0.     0.   ]
 [1.    0.     0.     1.   0.   1.   1.916  1.916 ]
 [1.    1.223  1.510  1.   0.   1.   0.     0.   ]]
```

例如对于第 2 个样本来说：

词表中的第 1 个词“this”在该样本中的出现的次数为 1，所以其 TF-IDF 值为

$$1\times1=1 \tag{7-6}$$

词表中的第 2 个词“document” 在该样本中的出现的次数为 2，所以其 TF-IDF 值为

$$2\times1.223=2.446 \tag{7-7}$$

词表中的第 3 个词“first” 在该样本中的出现的次数为 0，所以其 TF-IDF 值为

$$0\times1.510=0 \tag{7-8}$$

同样，对于其它样本 TF-IDF 值的计算也可以按照上述过程进行，读者们可以自行进行验算。这样，我们就将原始的文本表示转换成了 TF-IDF 形式的数值表示。

7.3.4 TF-IDF 示例代码

对于上述整个计算过程，可以使用 sklearn 中的 CountVectorizer 类和 TfidfTransformer 类来完成，完整代码见 Chapter07/05_tf_idf.py 文件。关键代码如下：

```
1 if __name__ == '__main__':
2     count = CountVectorizer(vocabulary=vocabulary)
3     count_matrix = count.fit_transform(corpus).toarray()
4     tfidf_trans = TfidfTransformer(norm=None)
5     tfidf_matrix = tfidf_trans.fit_transform(count_matrix)
6     idf_vec = tfidf_trans.idf_
7     print(tfidf_matrix.toarray())
```

在上述代码中，第 1 行用来实例化类 CountVectorizer，并同时传入词表；第 2 行用来对原始数据进行词频统计；第 3-4 行代码用来计算整个 TF-IDF 矩阵。同时，count_martrix 是词频统计矩阵；tfidf_matrix 是 TF-IDF 权重矩阵，也就是 7.3.3 节计算 TF-IDF 中的结果；idf_vec 是 IDF 向量。

但在默认情况下，第 3 行代码中的参数 norm 的值为’l2’，也就是说此时 TfidfTransformer 会对 TF-IDF 权重矩阵的每一行进行标准化，即标准化后每一行的模为 1。同时，在上面示例中，设置 norm 为 None 只是为了复现 7.3.2 中 TF-IDF 的计算过程，方便读者理解。

最后，需要解释一下的地方是上述代码第 2 行和第 4 行后面的 toarray()方法。根据 7.1 节介绍的内容可以知道，利用词袋模型来表示文本通常来说维度会比较高，当样本较多时词表中可能会有数万或者是数十万个词。因此，在这种情况下对于每个样本来说，其通过词袋模型转换后的特征向量中都会存在大量的 0，从而使得最后得到的特征矩阵非常稀疏（Sparse）。所以，为了提高存储效率，在 sklearn 中这样的稀疏矩阵都会采用稀疏方式进行存储。例如 7.3.4 节中，tfidf_matrix 的第 1 行采用稀疏表示的结果为

```
1 (0, 5)  1.0
2 (0, 3)  1.0
3 (0, 2)  1.5108256237659907
4 (0, 1)  1.2231435513142097
5 (0, 0)  1.0
```

其中第 1 行的含义是原始矩阵中第 0 行第 5 列的值是 1；同理第 3 行的含义是原始矩阵中第 0 行第 2 列的值是 1.510。注意，这里的所索引都是从 0 开始。并且可以发现，对于原始矩阵中取值为 0 的位置在稀疏矩阵中并没有被体现出来，这也就极大的节省了变量的存储空间。所以，当有需要查看原始非稀疏矩阵的结果时，就可以通过 toarray()方法来转换得到。不过在 sklearn 中，不管样本特征采用的是稀疏表示方法还是非稀疏表示方法，都可以直接用来进行建模。

7.3.5 小结

在本节中，笔者首先介绍了什么是 TF-IDF，以及为什么需要使用到 TF-IDF；接着介绍了 TF-IDF 的计算原理，并同时用真实的示例演示了 TF-IDF 的整个计算过程；最后介绍了如何通过 sklearn 中的 CountVectorizer 类和 TfidfTransformer 类来完成整个计算过程。

7.4 词云图

在介绍完文本的向量化表示方法后，这里再顺便介绍一个实用的对文本按权重（频率）进行可视化的 Python 包 word cloud。根据 word cloud，可以将词语以权重大小或者是词频高低来生成词云图，如图 7-3 所示。

图 7-3 词云图

图 7-3 所展示的词云图是根据宋词分词统计后所形成的结果，其中字体越大表示其出现的频率越高或者是 TF-IDF 权重越大。

7.4.1 生成词云图

在生成词云图之前，首先需要统计得到词频或者是 TF-IDF 权重。接下来，以一个宋词数据集为例进行介绍。完整代码见 Chapter07/06_word_cloud.py 文件。

1) 载入原始文本

首先，这里需要载入原始的宋词数据，代码如如下：

```
def load_data_and_cut(file_path='./data/QuanSongCi.txt'):
    cut_words = ""
    with open(file_path, encoding='utf-8') as f:
        for line in f:
            line = line.strip('\n')
            if len(line) < 20:
                continue
            seg_list = jieba.cut(clean_str(line), cut_all=False)
            cut_words += (" ".join(seg_list))
        all_words = cut_words.split()
    return all_words
```

可以看到，上述代码和 7.1.3 节中的基本一样，所以在此就不再赘述。

2) 统计词频

接下来，通过 Counter 计数器来完成分词结果中词频的统计，代码如下：

```
def get_words_freq(all_words, top_k=500):
    c = Counter()
    for x in all_words:
        if len(x) > 1 and x != '\r\n':
            c[x] += 1
    vocab = {}
    for (k, v) in c.most_common(top_k):
        vocab[k] = v
    return vocab
```

在上述代码中，第 2 行用来定义一个计数器；第 5 行用来对每个词进行计数；第 6-8 行用来查找出现频率最高的前 top_k 个词，并将词和出现频率以字典的形式进行存储。

3) 生成词云图

在得到词频字典后，便可以通过 WordCloud 类来完成词云图的生成，代码如下：

```
def show_word_cloud(word_fre):
    word_cloud = WordCloud(font_path='./data/simhei.ttf',
                           background_color='white', max_font_size=70)
    word_cloud.fit_words(word_fre)
    plt.imshow(word_cloud)
    plt.xticks([])  # 去掉横坐标
    plt.yticks([])  # 去掉纵坐标
    plt.tight_layout()
    plt.show()
```

在上述代码中，第 2 行用来载入汉字字体，因为 word cloud 默认不支持汉字；第 4 行用来生成词云图；第 5-6 行用来展示最后生成的词云图。在运行完上述代码后，便可以得到如图 7-3 所示的词云图。可以发现，全词中出现频率最高的几个词便是“人家”、“东风”、“何处”、“风流”等。

7.4.2 自定义样式

通过 word cloud 除了能够生成类似 7-3 所示的矩形词云图以外，更多场景下我们希望能够生成自定义样式的词云图。例如一个人的形状、一个建筑的形状等。在 word cloud 中，只需要在实例化对象 WordCloud 时传入一个掩码矩阵即可完成这一想法。完整代码见 Chapter07/07_word_cloud.py 文件。

```
1 def show_word_cloud(word_fre):
2     from PIL import Image
3     img = Image.open('./data/dufu.png')
4     img_array = np.array(img)
5     word_cloud = WordCloud(font_path='./data/simhei.ttf',
6                    background_color='white', max_font_size=70,mask=img_array)
7     word_cloud.fit_words(word_fre)
8     plt.imshow(word_cloud)
9     plt.xticks([])  # 去掉横坐标
10    plt.yticks([])  # 去掉纵坐标
11    plt.tight_layout()
12    plt.show()
```

在运行完上述代码后，便可以生成一个自定义形状的词云图，如图 7-4 右所示。

图 7-4 自定义词云图样式

在上述代码中，第 3-4 行用来打开一张图片，并同时转换为一个矩阵；第 6 行在实例化类 WordCloud 时需要将这个矩阵赋值到 mask 参数。这样便能够生成自定义样式的词云图。这里需要注意的一个地方就是，选择的这张图片的背景一定要是纯白色的，因为 WordCloud 的填充原理就是在图片的非白色趋于进行填充。如果使用的是一张非白色背景的图片，那么最后生成的词云图依旧是一个矩形。

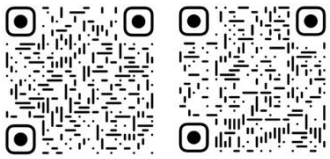
7.4.3 小结

在本节中，笔者首先介绍了什么是词云图；接着介绍了如何根据得到的词频统计结果通过 word cloud 工具来生成词云图；最后还介绍了如何生成自定义形状的词云图。

总结以下，在本章中笔者首先介绍了两种文本处理领域中常见的词袋模型，一步一步详细介绍了整个文本向量化的处理流程；接着介绍了如何使用朴素贝叶斯算法来完成中文垃圾邮件的分类任务，同时也介绍了如何复用模型，包括存储模型和载入模型；其次介绍了文本

处理领域中使用最为频繁的 TF-IDF 表示方法以及对应的计算过程；最后介绍了一种常见的文本可视化手段，即以词和其对应的词频为参数，通过 word cloud 工具来生成相应的词云图。

本次内容就到此结束，感谢您的阅读！如果你觉得上述内容对你有所帮助，欢迎分享至一位你的朋友！若有任何疑问与建议，请添加笔者微信'nulls8'或加群进行交流。**青山不改，绿水长流，我们月来客栈见！**



扫码关注@月来客栈可获得更多优质内容！

代码仓库： https://github.com/moon-hotel/MachineLearningWithMe

2021年

第一章：机器学习环境安装　　PDF内容

Python版本为3.6，各个Python包版本见`requirements.txt`，使用如下命令即可安装：

```
pip install -r requirements.txt
```

第二章：从零认识线性回归　　代码　　PDF内容

第三章：从零认识逻辑回归　　代码　　PDF内容

第四章：模型的改善与泛化　　代码　　PDF内容

第五章：K 近邻算法与原理　　代码　　PDF内容

第六章：朴素贝叶斯算法　　PDF内容

第七章：文本特征提取与模型复用

第八章：决策树与集成模型

第九章：支持向量机

第十章：聚类算法