垃圾短信识别

Spam SMS Recognition

实验时间：2024年4月16日
截止时间：2024年4月29日

CONTENT

目　录

01 问题描述
Problem description

02 实验内容
Experimental Content

03 代码实现
Code implementation

04 模型训练和预测
Model training and prediction

问题描述

Problem description

问题描述
Problem description

垃圾短信 (Spam Messages，SM) 是指未经过用户同意向用户发送不愿接收的商业广告或者不符合法律规范的短信。
大数据时代的到来使得大量个人信息数据得以沉淀和积累，但是庞大的数据量缺乏有效的整理规范。
在面对巨大的短信数据时，为了保证更良好的用户体验，如何从数据中挖掘出更多有意义的信息为人们免受垃圾短信骚扰成为当前亟待解决的问题。

实验要求：
1) 任务提供包括数据读取、基础模型、模型训练等基本代码
2) 参赛选手需完成核心模型构建代码，并尽可能将模型调到最佳状态
3) 模型单次推理时间不超过 10 秒

参考资料：
- Numpy：https://www.numpy.org/
数值计算相关的开源库
- Pandas: https://pandas.pydata.org/
数据分析的开源库
- Sklearn: https://scikit-learn.org/
机器学习算法库(基于Numpy 和 Python)
- jieba: https://github.com/fxsjy/jieba
中文分词组件，本次实验的主要工具
- 四川大学机器智能实验室停用词库：
https://github.com/goto456/stopwords/blob/master/scu_stopwords.txt

2 实验内容

Experimental Content

数据集　　数据处理　　模型搭建

实验内容
Experimental Content

数据集：
- 该数据集包括了约 7.87 万条数据，有 3 个字段 label、message 和 msg_new， 分别代表了短信的类别、短信的内容和分词后的短信
- 中文分词工具 [jieba](https://github.com/fxsjy/jieba)
- 0 代表正常的短信，1 代表恶意的短信
- 正常短信和恶意短信举例：

label	message（短信内容）	msg_new（短信分词后）
0	人们经常是失去了才发现它的珍贵	人们 经常 是 失去 了 才 发现 它 的 珍贵
1	本人现在承办驾驶证业务!招收学员，一对 一教学	本人 现在 承办 驾驶证 业务 ! 招收 学员 ， 一对 一 教学

根据 data path 利用 pandas 库读取 csv 格式的数据集

```
sms = pd.read_csv(data_path, encoding='utf-8')
```

1. 分词

- `jieba.cut` 方法接受四个输入参数: 需要分词的字符串；cut_all 参数用来控制是否采用全模式；HMM 参数用来控制是否使用 HMM 模型；use_paddle 参数用来控制是否使用paddle模式下的分词模式，paddle模式采用延迟加载方式，通过enable_paddle接口安装paddlepaddle-tiny，并且import相关代码；
- `jieba.cut_for_search` 方法接受两个参数：需要分词的字符串；是否使用 HMM 模型。该方法适合用于搜索引擎构建倒排索引的分词，粒度比较细
- 待分词的字符串可以是 unicode 或 UTF-8 字符串、GBK 字符串。注意：不建议直接输入 GBK 字符串，可能无法预料地错误解码成 UTF-8
- `jieba.cut` 以及 `jieba.cut_for_search` 返回的结构都是一个可迭代的 generator，可以使用 for 循环来获得分词后得到的每一个词语(unicode)，或者用
- `jieba.lcut` 以及 `jieba.lcut_for_search` 直接返回 list
- `jieba.Tokenizer(dictionary=DEFAULT_DICT)` 新建自定义分词器，可用于同时使用不同词典。`jieba.dt` 为默认分词器，所有全局分词相关函数都是该分词器的映射。

代码示例

```
# encoding=utf-8
import jieba

jieba.enable_paddle()# 启动paddle模式。 0.40版之后开始支持，早期版本不支持
strs=["我来到北京清华大学","乒乓球拍卖完了","中国科学技术大学"]
for str in strs:
    seg_list = jieba.cut(str,use_paddle=True) # 使用paddle模式
    print("Paddle Mode: " + '/'.join(list(seg_list)))

seg_list = jieba.cut("我来到北京清华大学", cut_all=True)
print("Full Mode: " + "/ ".join(seg_list))  # 全模式

seg_list = jieba.cut("我来到北京清华大学", cut_all=False)
print("Default Mode: " + "/ ".join(seg_list))  # 精确模式

seg_list = jieba.cut("他来到了网易杭研大厦")  # 默认是精确模式
print(", ".join(seg_list))

seg_list = jieba.cut_for_search("小明硕士毕业于中国科学院计算所，后在日本京都大学深造")  # 搜索引擎模式
print(", ".join(seg_list))
```

输出:

```
【全模式】: 我/ 来到/ 北京/ 清华/ 清华大学/ 华大/ 大学

【精确模式】: 我/ 来到/ 北京/ 清华大学

【新词识别】: 他，来到，了，网易，杭研，大厦    （此处，“杭研”并没有在词典中，但是也被Viterbi算法识别出来了）

【搜索引擎模式】: 小明，硕士，毕业，于，中国，科学，学院，科学院，中国科学院，计算，计算所，后，在，日本，京
```

2. 添加自定义词典

载入词典

- 开发者可以指定自己自定义的词典，以便包含 jieba 词库里没有的词。虽然 jieba 有新词识别能力，但是自行添加新词可以保证更高的正确率
- 用法： jieba.load_userdict(file_name) # file_name 为文件类对象或自定义词典的路径
- 词典格式和 `dict.txt` 一样，一个词占一行；每一行分三部分：词语、词频（可省略）、词性（可省略），用空格隔开，顺序不可颠倒。`file_name` 若为路径或二进制方式打开的文件，则文件必须为 UTF-8 编码。
- 词频省略时使用自动计算的能保证分出该词的词频。

例如：

```
创新办 3 i
云计算 5
凯特琳 nz
台中
```

- 更改分词器（默认为 `jieba.dt`）的 `tmp_dir` 和 `cache_file` 属性，可分别指定缓存文件所在的文件夹及其文件名，用于受限的文件系统。
- 范例：
  - 自定义词典：https://github.com/fxsjy/jieba/blob/master/test/userdict.txt
  - 用法示例：https://github.com/fxsjy/jieba/blob/master/test/test_userdict.py
    - 之前： 李小福 / 是 / 创新 / 办 / 主任 / 也 / 是 / 云 / 计算 / 方面 / 的 / 专家 /
    - 加载自定义词库后： 李小福 / 是 / 创新办 / 主任 / 也 / 是 / 云计算 / 方面 / 的 / 专家 /

调整词典

- 使用 `add_word(word, freq=None, tag=None)` 和 `del_word(word)` 可在程序中动态修改词典。
- 使用 `suggest_freq(segment, tune=True)` 可调节单个词语的词频，使其能（或不能）被分出来。
- 注意：自动计算的词频在使用 HMM 新词发现功能时可能无效。

代码示例：

```
>>> print('/'.join(jieba.cut('如果放到post中将出错。', HMM=False)))
如果/放到/post/中将/出错/。
>>> jieba.suggest_freq(('中', '将'), True)
494
>>> print('/'.join(jieba.cut('如果放到post中将出错。', HMM=False)))
如果/放到/post/中/将/出错/。
>>> print('/'.join(jieba.cut('「台中」正确应该不会被切开', HMM=False)))
「/台/中/」/正确/应该/不会/被/切开
>>> jieba.suggest_freq('台中', True)
69
>>> print('/'.join(jieba.cut('「台中」正确应该不会被切开', HMM=False)))
「/台中/」/正确/应该/不会/被/切开
```

实验内容
Experimental Content

数据处理：

停用词：

停用词是指在信息检索中，为节省存储空间和提高搜索效率，在处理自然语言数据（或文本）之前或之后会自动过滤掉某些字或词，这些字或词即被称为 Stop Words（停用词）。

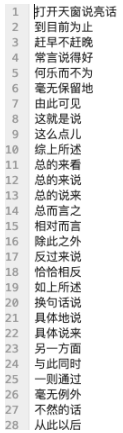
这些停用词都是人工输入、非自动化生成的，生成后的停用词会形成一个停用词库。

本次实验中采用的是[四川大学机器智能实验室停用词库](https://github.com/goto456/stopwords/blob/master/scu_stopwords.txt)

注意：所给出的停词库为 txt 形式，故需要用python的文件读取对数据集进行载入。

下面是一些停用词：

```
['嘿', '很', '乎', '会', '或', '既', '及', '啦', '了', '们', '你',
'您', '哦', '砰', '啊', '你', '我', '他', '她', '它']
```




```
def read_stopwords(stopwords_path):
```

.splitlines()
可能会用到的函数

实验内容
Experimental Content

数据处理：
文本向量化：CountVectorizer和TfidfVectorizer

CountVectorizer：目前拥有的数据是长度不统一的文本数据，而绝大多数机器学习算法需要的输入是向量，因此文本类型的数据需要经过处理得到向量。
我们可以借助 sklearn 中 CountVectorizer 来实现文本的向量化，CountVectorizer 实际上是在统计每个词出现的次数，这样的模型也叫做词袋模型。

实验内容
Experimental Content

CountVectorizer:

Reference: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

Parameters:

input : *{'filename', 'file', 'content'}, default='content'*
- If `'filename'`, the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze.
- If `'file'`, the sequence items must have a 'read' method (file-like object) that is called to fetch the bytes in memory.
- If `'content'`, the input is expected to be a sequence of items that can be of type string or byte.

encoding : *str, default='utf-8'*
If bytes or files are given to analyze, this encoding is used to decode.

decode_error : *{'strict', 'ignore', 'replace'}, default='strict'*
Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given `encoding`. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

strip_accents : *{'ascii', 'unicode'} or callable, default=None*
Remove accents and perform other character normalization during the preprocessing step. 'ascii' is a fast method that only works on characters that have a direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) means no character normalization is performed.

Both 'ascii' and 'unicode' use NFKD normalization from `unicodedata.normalize`.

lowercase : *bool, default=True*
Convert all characters to lowercase before tokenizing.

preprocessor : *callable, default=None*
Override the preprocessing (strip_accents and lowercase) stage while preserving the tokenizing and n-grams generation steps. Only applies if `analyzer` is not callable.

tokenizer : *callable, default=None*
Override the string tokenization step while preserving the preprocessing and n-grams generation steps. Only applies if `analyzer == 'word'`.

stop_words : *{'english'}, list, default=None*
If 'english', a built-in stop word list for English is used. There are several known issues with 'english' and you should consider an alternative (see Using stop words).

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if `analyzer == 'word'`.

If None, no stop words will be used. In this case, setting `max_df` to a higher value, such as in the range (0.7, 1.0), can automatically detect and filter stop words based on intra corpus document frequency of terms.

token_pattern : *str or None, default=r"(?u)\b\w\w+\b"*
Regular expression denoting what constitutes a "token", only used if `analyzer == 'word'`. The default regexp select tokens of 2 or more alphanumeric characters (punctuation is completely ignored and always treated as a token separator).

If there is a capturing group in token_pattern then the captured group content, not the entire match, becomes the token. At most one capturing group is permitted.

ngram_range : *tuple (min_n, max_n), default=(1, 1)*
The lower and upper boundary of the range of n-values for different word n-grams or char n-grams to be extracted. All values of n such such that min_n <= n <= max_n will be used. For example an `ngram_range` of `(1, 1)` means only unigrams, `(1, 2)` means unigrams and bigrams, and `(2, 2)` means only bigrams. Only applies if `analyzer` is not callable.

analyzer : *{'word', 'char', 'char_wb'} or callable, default='word'*
Whether the feature should be made of word n-gram or character n-grams. Option 'char_wb' creates character n-grams only from text inside word boundaries; n-grams at the edges of words are padded with space.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

Changed in version 0.21.

Since v0.21, if `input` is `filename` or `file`, the data is first read from the file and then passed to the given callable analyzer.

max_df : *float in range [0.0, 1.0] or int, default=1.0*
When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words). If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

min_df : *float in range [0.0, 1.0] or int, default=1*
When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

max_features : *int, default=None*
If not None, build a vocabulary that only consider the top `max_features` ordered by term frequency across the corpus. Otherwise, all features are used.

This parameter is ignored if vocabulary is not None.

vocabulary : *Mapping or iterable, default=None*
Either a Mapping (e.g., a dict) where keys are terms and values are indices in the feature matrix, or an iterable over terms. If not given, a vocabulary is determined from the input documents. Indices in the mapping should not be repeated and should not have any gap between 0 and the largest index.

binary : *bool, default=False*
If True, all non zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts.

dtype : *dtype, default=np.int64*
Type of the matrix returned by fit_transform() or transform().

Attributes:

vocabulary_ : *dict*
A mapping of terms to feature indices.

fixed_vocabulary_ : *bool*
True if a fixed vocabulary of term to indices mapping is provided by the user.

stop_words_ : *set*
Terms that were ignored because they either:
- occurred in too many documents (`max_df`)
- occurred in too few documents (`min_df`)
- were cut off by feature selection (`max_features`).

This is only available if no vocabulary was given.

Methods

`build_analyzer`()	Return a callable to process input data.
`build_preprocessor`()	Return a function to preprocess the text before tokenization.
`build_tokenizer`()	Return a function that splits a string into a sequence of tokens.
`decode`(doc)	Decode the input into a string of unicode symbols.
`fit`(raw_documents[, y])	Learn a vocabulary dictionary of all tokens in the raw documents.
`fit_transform`(raw_documents[, y])	Learn the vocabulary dictionary and return document-term matrix.
`get_feature_names_out`([input_features])	Get output feature names for transformation.
`get_metadata_routing`()	Get metadata routing of this object.
`get_params`([deep])	Get parameters for this estimator.
`get_stop_words`()	Build or fetch the effective stop words list.
`inverse_transform`(X)	Return terms per document with nonzero entries in X.
`set_fit_request`(*[, raw_documents])	Request metadata passed to the `fit` method.
`set_params`(**params)	Set the parameters of this estimator.
`set_transform_request`(*[, raw_documents])	Request metadata passed to the `transform` method.
`transform`(raw_documents)	Transform documents to document-term matrix.

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This document is the second document.',
...     'And this is the third one.',
...     'Is this the first document?',
... ]
>>> vectorizer = CountVectorizer()
>>> X = vectorizer.fit_transform(corpus)
>>> vectorizer.get_feature_names_out()
array(['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third',
       'this'], ...)
>>> print(X.toarray())
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
>>> vectorizer2 = CountVectorizer(analyzer='word', ngram_range=(2, 2))
>>> X2 = vectorizer2.fit_transform(corpus)
>>> vectorizer2.get_feature_names_out()
array(['and this', 'document is', 'first document', 'is the', 'is this',
       'second document', 'the first', 'the second', 'the third', 'third one',
       'this document', 'this is', 'this the'], ...)
 >>> print(X2.toarray())
 [[0 0 1 1 0 0 1 0 0 0 0 1 0]
 [0 1 0 1 0 1 0 1 0 0 1 0 0]
 [1 0 0 1 0 0 0 0 1 1 0 1 0]
 [0 0 1 0 1 0 1 0 0 0 0 0 1]]
```

实验内容
Experimental Content

与CountVectorizer 类似，我们有TFidfVectorizer:
TF-IDF 算法是创建在这样一个假设之上的：
对区别文档最有意义的词语应该是那些在文档中出现频率高的词语，因此选择特征空间坐标系取 TF 词频作为测度，就可以体现同类文本的特点。
另外考虑到单词区别不同类别的能力，TF-IDF 法认为一个单词出现的文本频数越小，它区别不同类别文本的能力就越大。
因此引入了逆文本频度 IDF 的概念，以 TF 和 IDF 的乘积作为特征空间坐标系的取值测度，并用它完成对权值 TF 的调整，调整权值的目的在于突出重要单词，抑制次要单词。
在本质上 IDF 是一种试图抑制噪声的加权，并且单纯地认为文本频率小的单词就越重要，文本频率大的单词就越无用。

Reference: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

- TF：词频。

$$TF(w) = \frac{\text{词}w\text{在文档中出现的次数}}{\text{文档的总词数}}$$

- IDF：逆向文件频率。有些词可能在文本中频繁出现，但并不重要，也即信息量小，如 is, of, that 这些单词，这些单词在语料库中出现的频率也非常大，我们就可以利用这点，降低其权重。

$$IDF(w) = ln\frac{\text{语料库的总文档数}}{\text{语料库中词}w\text{出现的文档数}}$$

- TF-ID 综合参数：TF - IDF = TF * IDF

实验内容
Experimental Content

数据处理：

划分训练集和测试集：

一般的数据集会划分为两个部分：

- 训练数据：用于训练，构建模型
- 测试数据：在模型检验时使用，用于评估模型是否有效

划分比例：

- 训练集：70% 80% 75%
- 测试集：30% 20% 25%

```
sklearn.model_selection.train_test_split(x, y, test_size, random_state )
```

- `x`：数据集的特征值
- `y`： 数据集的标签值
- `test_size`： 如果是浮点数，表示测试集样本占比；如果是整数，表示测试集样本的数量。
- `random_state`： 随机数种子,不同的种子会造成不同的随机采样结果。相同的种子采样结果相同。
- `return` 训练集的特征值 `x_train` 测试集的特征值 `x_test` 训练集的目标值 `y_train` 测试集的目标值 `y_test`。

实验内容
Experimental Content

模型搭建原理：

利用朴素贝叶斯算法原理:

朴素贝叶斯实现分类的原理是基于贝叶斯公式，给定一个样本，计算该样本条件下每个类别的条件概率。

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots, x_n \mid y)}{P(x_1, \ldots, x_n)}$$

由于假设每个特征是独立的，所以该公式可以化为:

$$P(x_i|y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|y),$$

由于分母是确定的，结果只和分子有关。

$$P(y \mid x_1, \ldots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

求出最大的条件概率，其对应的类别就是该样本所属的类别。

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

实验内容
Experimental Content

模型搭建实现：

采用 sklearn.naive_bayes 搭建一个简单的模型：

MultinomialNB 是一种常用于文本分类的朴素贝叶斯方法，下面以此为例训练一个朴素贝叶斯分类器

```
from sklearn.naive_bayes import MultinomialNB
nb = MultinomialNB()
%time nb.fit(X_train_dtm, y_train)  # 计算训练时间
```

```
CPU times: user 154 ms, sys: 4.02 ms, total: 158 ms
Wall time: 162 ms
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

构建一个 PipeLine：

构建 PipleLine 可以将数据处理和数据分类结合在一起，这样输入原始的数据就可以得到分类的结果，方便直接对原始数据进行预测。

实验内容
Experimental Content

Pipeline:
利用一系列数据变换(算法)和一个预测器构成的一套流程。

class sklearn.pipeline.**Pipeline**(*steps*, ***, *memory=None*, *verbose=False*) [source]

A sequence of data transformers with an optional final predictor.

`Pipeline` allows you to sequentially apply a list of transformers to preprocess the data and, if desired, conclude the sequence with a final predictor for predictive modeling.

Intermediate steps of the pipeline must be 'transforms', that is, they must implement `fit` and `transform` methods. The final estimator only needs to implement `fit`. The transformers in the pipeline can be cached using `memory` argument.

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps using their names and the parameter name separated by a `'__'`, as in the example below. A step's estimator may be replaced entirely by setting the parameter with its name to another estimator, or a transformer removed by setting it to `'passthrough'` or `None`.

For an example use case of `Pipeline` combined with `GridSearchCV`, refer to Selecting dimensionality reduction with Pipeline and GridSearchCV. The example Pipelining: chaining a PCA and a logistic regression shows how to grid search on a pipeline using `'__'` as a separator in the parameter names.

Read more in the User Guide.

New in version 0.5.

Parameters:

steps : *list of tuples*
List of (name of step, estimator) tuples that are to be chained in sequential order. To be compatible with the scikit-learn API, all steps must define `fit`. All non-last steps must also define `transform`. See Combining Estimators for more details.

memory : *str or object with the joblib.Memory interface, default=None*
Used to cache the fitted transformers of the pipeline. The last step will never be cached, even if it is a transformer. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute `named_steps` or `steps` to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming.

verbose : *bool, default=False*
If True, the time elapsed while fitting each step will be printed as it is completed.

Attributes:

`named_steps` : *Bunch*
Access the steps by name.

`classes_` : ***ndarray of shape (n_classes,)***
The classes labels.

`n_features_in_` : ***int***
Number of features seen during first step `fit` method.

`feature_names_in_` : ***ndarray of shape (n_features_in_,)***
Names of features seen during first step `fit` method.

Reference: https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

```
>>> from sklearn.svm import SVC
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.datasets import make_classification
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.pipeline import Pipeline
>>> X, y = make_classification(random_state=0)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
...                                                     random_state=0)
>>> pipe = Pipeline([('scaler', StandardScaler()), ('svc', SVC())])
>>> # The pipeline can be used as any other estimator
>>> # and avoids leaking the test set into the train set
>>> pipe.fit(X_train, y_train).score(X_test, y_test)
0.88
>>> # An estimator's parameter can be set using '__' syntax
>>> pipe.set_params(svc__C=10).fit(X_train, y_train).score(X_test, y_test)
0.76
```

`decision_function`(X, **params)	Transform the data, and apply `decision_function` with the final estimator.
`fit`(X[, y])	Fit the model.
`fit_predict`(X[, y])	Transform the data, and apply `fit_predict` with the final estimator.
`fit_transform`(X[, y])	Fit the model and transform with the final estimator.
`get_feature_names_out`([input_features])	Get output feature names for transformation.
`get_metadata_routing`()	Get metadata routing of this object.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(Xt, **params)	Apply `inverse_transform` for each step in a reverse order.
`predict`(X, **params)	Transform the data, and apply `predict` with the final estimator.
`predict_log_proba`(X, **params)	Transform the data, and apply `predict_log_proba` with the final estimator.
`predict_proba`(X, **params)	Transform the data, and apply `predict_proba` with the final estimator.
`score`(X[, y, sample_weight])	Transform the data, and apply `score` with the final estimator.
`score_samples`(X)	Transform the data, and apply `score_samples` with the final estimator.
`set_output`(*[, transform])	Set the output container when `"transform"` and `"fit_transform"` are called.
`set_params`(**kwargs)	Set the parameters of this estimator.
`set_score_request`(*[, sample_weight])	Request metadata passed to the `score` method.
`transform`(X, **params)	Transform the data, and apply `transform` with the final estimator.

代码实现

Code implementation

代码实现

Code implementation

完成读取停用词的代码

```python
import os
os.environ["HDF5_USE_FILE_LOCKING"] = "FALSE"

# ---------- 停用词库路径，若有变化请修改 -------------
stopwords_path = r'scu_stopwords.txt'
# ------------------------------------------------

def read_stopwords(stopwords_path):
    """
    读取停用词库
    :param stopwords_path: 停用词库的路径
    :return: 停用词列表，如 ['嘿', '很', '乎', '会', '或']
    """
    stopwords = []
    # ----------- 请完成读取停用词的代码 ------------

    #-----------------------------------------------

    return stopwords

# 读取停用词
stopwords = read_stopwords(stopwords_path)
```

代码实现

Code implementation

完成读取停用词的代码之后，实现pipline_list，并最终在测试集上进行评估

```
# ---------------- 导入相关的库 ----------------
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import BernoulliNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import ComplementNB

# pipline_list用于传给Pipline作为参数
pipeline_list = [
    # ------------------------- 需要完成的代码 -----------------------------

    # ========================= 以下代码仅供参考 ============================
    ('cv', CountVectorizer(token_pattern=r"(?u)\b\w+\b", stop_words=stopwords)),
    ('classifier', MultinomialNB())
    # ======================================================================

    # ----------------------------------------------------------------------
]
```

模型训练与预测

Model training and prediction

模型训练

model train

可以尝试从以下几个方面去优化模型：

1. 文本向量化可以选择 CountVectorizer 或者 TfidfVectorizer，适当调节里面的参数，如 ngram_range
2. 更换更好的停用词库，请放在 results 目录下
2. 尝试进行数据进行归一化，可以采用 StandardScaler 或者 MaxAbsScaler
3. 适当调节分类器的参数，提高模型的表现

注意事项

Notifications

注意：

1. 点击左侧栏 `提交结果` 后点击 `生成文件` 则只需勾选 `predict()` 函数的cell。**注意不要勾选训练模型的代码。**
2. 请导入必要的包和第三方库 (包括此文件中曾经导入过的)。
3. 请加载你认为训练最佳的模型，即请按要求填写模型路径。
4. `predict()` 函数的输入和输出请不要改动。
5. 测试时记得填写你的模型路径及名称, 如果采用 离线任务 请将模型保存在 **results** 文件夹下。

谢谢