华东师范大学数据科学与工程学院实验报告

课程名称：当代人工智能	**年级：**2021 级	**上机实践成绩：**
指导教师：李翔	**姓名：**李睿恩	**学号：**10215501434
上机实践名称：文本分类		**上机实践日期：**2023.9.25
上机实践编号：1	**组号：**无	**上机实践时间：**17:31

一、 实验目的

- 完成 10 分类文本分类任务

二、 实验任务

- 将训练数据自行划分训练集和验证集（采用固定划分或多折交叉验证）
- 将文本映射成向量（可以使用 TF-IDF/Word2vec/Fasttext/Glove/Bert/XLNet）
- 通过训练集训练分类器（必须实现 Logistic 回归/SVM/MLP/决策树）

三、 实验环境

本次实验基于 JetBrains 的 DataSpell 集成开发环境，使用 Python 编程语言实现。本次实验还载入了 numpy, sklearn, nltk, genism, tensorflow 等模块以实现数据处理、机器学习等功能。

四、 实验过程

4.1 文本映射为向量的思路概述

从计算机的角度来看，文本的数据类型为字符串，而字符串并不利于我们进行数据分析。为此，我们需要将文本映射为元素均为实数的向量。在本次实验中，我们尝试了 **TF-IDF 方法**与 **Word2Vec 方法**作为文本映射向量的方式，并对二者进行了对比。在之后，我们针对 TextCNN 方法采取了 **Tokenizer** 的文本映射向量方法。

4.1.1 TF-IDF 方法

TF-IDF 是文本特征提取的常用统计方法之一，它通过计算某单词在文本中的出现次数与该文本中所有词的词数的商得到**词频**（Term Frequency），再计算总的文章数量与包含该关键词的文章的数量的商并取对数得到**逆文本频率**（Inverse Document Frequency），最后计算二者的乘积得到一个词语的 TF-IDF 值。

值得注意的是，一些词语在所有的文本中都会经常出现，这些词语大多都不具有太多的意义，这对于文本分类理应不会起到太多用处。这些词被称为“停用词”，而在度量相关性时，这些词的出现频率不应被纳入考虑范围内。

最后，我们可以基于该方法用一个向量还表示一段文本。对于一篇文章的所有实词，计

算出他们的 TF-IDF 值，并把这些值按照对应的实词在词汇表的位置依次排列，就可以得到一个代表了该段文本的向量。另外，由于我们没有被提供一个完整的词汇表，故我们在规划词汇表时，选择了将 train_data.txt 与 test.txt 中所有出现过的单词作为词汇表。这只是考虑了所有可能出现的词汇，而并非直接使用了测试集作为训练集，因此我们认为这样的操作是可以接受的。

在本次实验中，我们调用了 sklearn.feature_extraction.text 模块以实现 TF-IDF 方法。具体代码如图 1 所示。

```
# 初始化训练参数
# analyzer为word时，表示以单词为单位计算TF-IDF值
# stop_words为english时，表示去除英语中的停用词，避免缺乏实际意义的计算
# use_idf为True时，表示要计算idf值
# smooth_idf为True时，表示要在文档频率上加一来平滑idf，避免分母为0
# norm为None时，表示输出结果不需要任何标准化或归一化，为l2时，说明输出结果进行了归一化
tv1 = TfidfVectorizer(analyzer="word", stop_words="english", use_idf=True, smooth_idf=True, norm="l2")
tv1_fit = tv1.fit_transform(ALL)
# 得到每一个文本对应的TF-IDF向量
vsm_matrix = tv1_fit.toarray()
# vsm_matrix
textList = []
result_list = []
for i in range(LENGTH_TRAIN):
    textList.append(vsm_matrix[i])
textList = np.array(textList)
np.save('textVectorList.npy', textList)
for i in range(LENGTH_TRAIN, LENGTH_ALL):
    result_list.append(vsm_matrix[i])
result_list = np.array(result_list)
np.save('result_list.npy', result_list)
# result_list
```

图 1 TF-IDF 的实现

实现 TF-IDF 的主要的函数为 TfidfVectorizer，其中参数的意义解释如下。

analyzer：当它的值为 word 时，表示以单词为单位计算 TF-IDF 值。

stop_words：设置停用词，当它的值为 English 时，表示去除英语中的停用词。

use_idf：当它的值为 True 时，表示要计算 idf 值。

smooth_idf：当它的值为 True 时，表示需要在计算文档频率时额外增加 1，避免计算 idf 时存在分母为 0 的情况。

norm：决定输出是否需要进行归一化，当值为 l2 时，说明输出结果会被归一化。

最终得到的 vsm_matrix 为一个矩阵，其每一行都是一段文本所转换得出的向量，每一个向量都有 33359 维，每一列均代表一个单词的 TF-IDF 值。其前 8000 行均来自训练集，而后 2000 行来自测试集。运行的时间为 14.0756 秒。

4.1.2 Word2Vec 方法

Word2Vec 是一种用于将单词映射到高维向量空间的自然语言处理技术，它通过学习大量文本数据中的单词上下文信息，将单词表示为实数向量。基于此，为了衡量一段文本的对应的向量，我们通过对一段文本中所有的词语对应向量求平均值，来代表这段文本。

我们删去停用词后，使用了 gensim.models 模块以实现 Word2Vec 方法，具体实现如图 2.

```
for text in rawList:
    tokens = word_tokenize(text.lower())
    filtered_tokens = []
    for word in tokens:
        if word not in stop_words:
            filtered_tokens.append(word)
    tokenized_texts.append(filtered_tokens)

model = Word2Vec(tokenized_texts, vector_size=100, window=5, min_count=1, sg=1)
text_vectors = []
for tokenized_text in tokenized_texts:
    text_vector = np.zeros(model.vector_size)
    word_count = 0

    for word in tokenized_text:
        if word in model.wv:
            text_vector = text_vector + model.wv[word]
            word_count = word_count + 1

    if word_count > 0:
        text_vector = text_vector / word_count

    text_vectors.append(text_vector)

text_vectors = np.array(text_vectors)
```

图 2 Word2Vec 的实现

实现 Word2Vec 方法的主要函数为 Word2Vec，其中参数的解释意义如下：

tokenized_texts: 包含了已经被分词的文本数据的列表。

vector_size: 指定每个单词的向量维度。

window: 模型在训练过程中，模型考虑了上下文的词语个数。

min_count: 只有出现次数不少于 min_count 次的单词才会被考虑。

sg: 这指代了我们使用了哪一种模型。sg=1 表示使用了 Skip-gram 模型，即通过给定一个词来预测它的上下文。sg=0 表示使用了 CBOW 模型，即通过给定上下文词来预测目标词。

最终得到的 text_vectors 是一个矩阵，其每一行都是每段文本所对应的向量。在图 2 所示的代码中，这里只计算了 train_data.txt 中的文本。在使用初始参数时，其运行时间为 14.6322 秒。

4.2 TF-IDF 与 Word2Vec 方法对比——以 Logistic 回归为例

本次实验中，我们将会使用多种分类模型来实现文本的 10 分类，但如果我们将所有分类模型都分别套用 TF-IDF 与 Word2Vec，我们的实验将会变得冗余。为此，我们计划选择一个在本次任务中效果最好的文本转换向量方式。我们以被广泛使用的 **Logistic 回归**为实验的分类方法，测试两个文本转换向量方式的优劣，观察他们的准确率、精确度、召回率与 F1-Score。

4.2.1 TF-IDF 下的 Logistic 回归

我们对 train_data.txt 中的数据随机划分 5 次，每次的 87.5%会作为训练集，剩下的 12.5%会作为测试集。训练后，观察测试集的平均准确率、平均精确度、平均召回率与平均 F1-Score。其中，准确率是指分类模型正确预测的样本数量占总样本数量的比例；精确度是指在所有被模型预测为某类别的样本中，实际正是该类别的样本的比例；召回率是指所有实际为某类别的样本中，模型成功预测为该类别的样本的比例，F1-Score 是精确度和召回率的调和平均数。在多分类问题中，classification_report 函数会返回每个分类下的上述数值的加权平均数。Logistic 回归的具体实现如图 3.

```
accuracyTotal = 0
precisionTotal = 0
recallTotal = 0
f1Total = 0
LOOP_NUMBER = 5
target_names = ['class_0', 'class_1', 'class_2', 'class_3', 'class_4', 'class_5', 'class_6', 'class_7', 'class_8', 'class_9']

for loop in range(LOOP_NUMBER):
    text_train, text_test, label_train, label_test = train_test_split(textList, labelList, test_size=0.125, random_state=None)
    model = LogisticRegression(max_iter=1000)
    model.fit(text_train, label_train)
    accuracy = model.score(text_test, label_test)

    y_pred = model.predict(text_test)
    classification_rep = classification_report(label_test, y_pred, target_names=target_names, output_dict=True)

    # 提取相应的指标值
    precision = classification_rep['weighted avg']['precision']
    recall = classification_rep['weighted avg']['recall']
    f1 = classification_rep['weighted avg']['f1-score']

    accuracyTotal += accuracy
    precisionTotal += precision
    recallTotal += recall
    f1Total += f1
```

图 3 Logistic 回归

我们通过随机划分五次训练集与测试集，调用 LogisticsRegression 函数测试 TF-IDF 下的 Logistic 方法效果，并计算准确率、精确度、召回率与 F1-Score 的平均值。最终的结果如图 4.

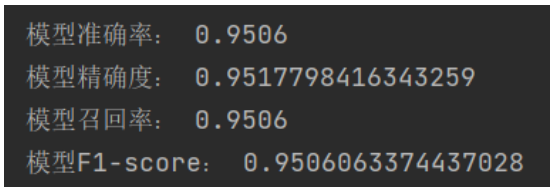

图 4 TF-IDF 下 Logistic 回归在训练集与测试集上的效果

可以发现，TF-IDF 下的 Logistic 有 95%左右的准确率、精确度、召回率与 F1-Score。这说明其在训练集与测试集中有不错的表现。

4. 2. 2 Word2Vec 下的 Logistic 回归

同上述过程，我们基于上文中给出的 Word2Vec 结果，随机划分 5 次训练集与测试集，调用 Logistic 分类模型，得到模型的准确率、精确度、召回率与 F1-Score 的平均值，最终的结果如图 5.

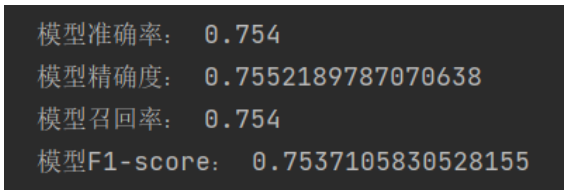

图 5 Word2Vec 下 Logistic 回归在训练集与测试集上的效果

这样的预测结果并不理想。为此，我们需要对 Word2Vec 作出一定的调整。

4. 2. 3 调整参数后 Word2Vec 下的 Logistic 回归

在上文中，我们的 Word2Vec 中的超参数是基于一些技术文档中给出的默认值设置的，这可能并不适配于我们的模型。为此，我们可以多次调整超参数，找出尽可能符合题目要求的模型。

经过多次尝试，我们调整超参数为 **vector_size=200, window=30, min_count=15**。这样的操作可以让 Word2Vec 在考虑文本中前后单词之间的相关性时，考虑到比原先的参数更多的信息，从而提升模型的准确率。调整超参数后，我们得到的结果如图 6.

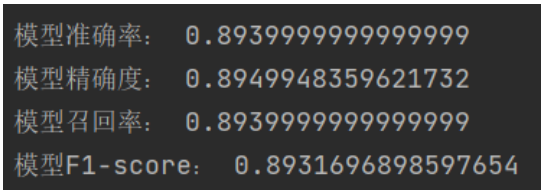

图 6 调整参数后 Word2Vec 下 Logistic 回归在训练集与测试集上的效果

我们发现其效果得到了很明显的提升。然而，这样调整参数也让该模型需要考虑更多的因素，导致 Word2Vec 的运行时长更久。在使用初始参数时，其运行时间为 14.6322 秒，而

在调整参数后，其运行时间增加到了 27.9425 秒。

4. 2. 4 基于 TF-IDF 加权后 Word2Vec 下的 Logistic 回归

实际上，直接将一个文本中的所有单词对应的 Word2Vec 向量的平均值作为衡量一个文本的向量是有显著缺点的，这忽略了每个词各自的重要程度。

一种优化的做法是，对于每一个单词所对应的 Word2Vec 向量，我都乘以该词在该语料库中的 TF-IDF 值，从而获得考虑了每个词重要程度的向量。其具体实现如图 7.

```
tokenized_texts = [[word for word in word_tokenize(text.lower()) if word not in stop_words] for text in rawList]
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(rawList)

model = Word2Vec(tokenized_texts, vector_size=200, window=30, min_count=15, sg=1)
text_vectors = []

for i, tokenized_text in enumerate(tokenized_texts):
    text_vector = np.zeros(model.vector_size)

    for word in tokenized_text:
        if word in model.wv:
            word_index = tfidf_vectorizer.vocabulary_.get(word, -1)
            if word_index != -1:
                tfidf_value = tfidf_matrix[i, word_index]
                text_vector = text_vector + model.wv[word] * tfidf_value

    text_vectors.append(text_vector)

text_vectors = np.array(text_vectors)
text_vectors
```

图 7 基于 TF-IDF 加权后 Word2Vec 下的 Logistic 回归

基于此运行的 Logistic 回归的效果如图 8 所示。

```
模型准确率:  0.914
模型精确度:  0.9147123069922802
模型召回率:  0.914
模型F1-score:  0.9138230037113402
```

图 8 基于 TF-IDF 加权后 Word2Vec 下的 Logistic 回归在训练集与测试集上的效果

可以发现，最终的效果得到了微弱的提升。这说明该方法总体上应当是有效的。

4. 3 TF-IDF 与 Word2Vec 对比分析

我们发现，经过多次调参与方法的优化，Word2Vec 的效果总是不及 TF-IDF 理想。但是，这样的现象并不符合常理，因为 TF-IDF 方法极其传统，它只是以单词的出现频率作为衡量文本的方式，而 Word2Vec 还额外考虑了文本中词的上下文关系。

实际上，如果是一个现实生活中遇到的文本分类问题，我们有理由相信 Word2Vec 可以具有比 TF-IDF 更加优越的性能。Word2Vec 之所以在处理该问题时没有良好的性能，可能有

以下三个原因。

1. 缺乏充分的文本预处理

在使用 Word2Vec 方法前的分词过程中，我们缺乏对数据进行完整预处理的步骤。我们观察到最后被用于 Word2Vec 方法的词语中，有大量词语的结构并不符合自然语言的规律，如“38ddd”, “wet/smooth”以及大量的标点符号。之所以我们没有对这些词语进行完整的预处理，主要是因为其中不符合自然语言规律的词汇形态各异，难以使用一个统一的标准进行处理。但这可能会影响该模型判断词和词之间的关系度，从而导致模型效果不佳。

2. 参数有待进一步调整

在 Word2Vec 中，我们注重了 vector_size, window 和 min_count 三个参数的调整。实际上，如果将这些参数的数字进一步地增大，我们应当可以得到效果更加理想的模型。然而，这样的参数调整会面临两个挑战，其一是调整参数的过程较为耗时耗力，对三个参数同时进行调整并观察其效果并不是一个可以快速完成的工作，其二是参数数值的增加会导致代码运行的时间进一步增加，从而导致过低的运行效率。在这里，我们认为 TF-IDF 的运行时间与模型效果已经足够在本题中使用，为提升 Word2Vec 的效果而消耗更多时间可能会得不偿失。

3. 以衡量词语的向量来衡量文本的方式有待调整

Word2Vec 主要是对每一个单词产生一个向量。因此，如何寻找一个合适的方法，将大量表示词语的向量综合起来表示一段文本，是一个非常困难的议题。尽管在网络上，不少人提出了以平均值的方式或以 TF-IDF 加权的方式来使得 Word2Vec 更具成效的方法，但可能会存在更好的衡量方式。

4. 模型不适用于本题

本次的 10 分类问题是一次作业，是基于一个现成的、规模较小的数据集，这与实际应用存在不小的差距。因此，我们不能否认该数据集可能与 Word2Vec 的实际应用场合存在出入。一种可能的情况是，在本题所提供的数据集中，由于每一段文本的特征与类别区分过于明显，以致于个别单词的出现足以让计算机预测该段文本的分类情况，此时使用 Word2Vec 的确可能不是一个更加优秀的选择。意即，不同的模型会有不一样的适配场合，而 Word2Vec 可能恰好不适用于本题。

基于上述分析，我们认为 Word2Vec 将在本题中不是一个最优方法。在本题中我们将**全部采用 TF-IDF 作为文本转换为向量的方法**。

4.4 训练分类器

在选定了 TF-IDF 作为本题文本转换为向量的方法后，我们将会对向量选择最合适的分类器。在这里，我们将分别训练 Logistic 回归、SVM、MLP、决策树与 TextCNN，观察其分类效果并选择分类效果最优的分类器作为最终使用的分类器。

4.4.1 Logistic 回归

Logistic 回归通常是用于解决二分类问题的统计学习方法。在这里，我们使用的 Logistic

回归是多类别 Logistic 回归，是 Logistic 回归在处理多类别分类问题时的扩展。在二分类的情况下，模型会输出一个介于 0 和 1 之间的值，表示样本属于正类别的概率，若概率大于 0.5 则认为该样本属于正类别，否则认为该样本属于另一类别，而多分类问题是基于此的扩展，它会选择具有最高置信度的类别作为最终的预测结果。

实现与分类效果见本报告 4.2.1.

4. 4. 2 支持向量机(Support Vector Machine)

支持向量机(Support Vector Machine, SVM)是一种监督学习算法，它旨在找到一个最优的超平面，该超平面可以将数据集中的数据点根据其类别进行分隔。在二分类问题中，SVM 寻找最优超平面使得两个类别的数据点之间的间隔最大化，而多分类问题是其扩展。

在这里，我们先通过使用 sklearn.svm 中的 LinearSVC 模块，直接优化线性模型以实现支持向量机。该模型默认会进行 L2 正则化，可以避免过拟合，且不需要调整过多参数便可以直接使用。代码的整体框架与实现 Logistic 回归时类似，具体实现与实现效果如图 9.

```
text_train, text_test, label_train, label_test = train_test_split(textList, labelList, test_size=0.125,
 random_state=None)
svm_classifier = LinearSVC(dual='auto')
svm_classifier.fit(text_train, label_train)
y_pred = svm_classifier.predict(text_test)
```

```
平均运行时间为:  4.321963691711426
模型准确率:  0.9620000000000001
模型精确度:  0.9626589359714325
模型召回率:  0.9620000000000001
模型F1-score:  0.9619274685443928
```

图 9 线性核函数下的支持向量机代码实现

可以观察到，其准确率、精确度、召回率与 F1-Score 均有 96%，且平均每次运行的时间为 4.3220 秒，表现良好。

然而，就在我们试图调用 SVC 模块并更换核函数来观察其他核函数下的支持向量机时，代码的运行却极其地耗时。即使只是运行一次调用 SVC 模块的支持向量机，我们都无法在短时间内观察到结果。考虑到 96%左右的准确率、精确度、召回率与 F1-Score 已经是非常不错的表现，牺牲如此大量的时间去换取可能没有过多提升的预测结果是得不偿失的，因此我们不再调用 SVC 模块观察其他核函数下的结果。

实际上，SVC 模块是一个自由度更高的模块，理论上说，它更具有泛用性与实用性。在本题中，调用 SVC 模块的代码运行时间过长，可能由以下原因导致。

1. TF-IDF 得到的向量维数过高

在我们运行的 TF-IDF 下，我们每一段文本的向量都有 33359 维，这是一个非常大的维数，这可能使得我们的计算非常地消耗算力。

2. LinearSVC 只考虑线性核函数的情况，算法得到了优化

LinearSVC 是一个只考虑了线性核函数的情况，这使得它可能会得到更好地优化。而

SVC 模块由于具有泛用性，可能因此会失去针对性地优化，使得它运行速度较低。

在使用了 TF-IDF 的背景下，我们只能放弃对 SVC 模块的使用。实际上，在文本分类任务中，线性核函数已经被认为具有不错的效果了。

4.4.3 多层感知机(Multilayer Perceptron)

多层感知机是一种前馈人工神经网络，是一种深度学习。它由三个或更多的层组成，这些层包括输入层、输出层和一个或多个隐藏层。每个层中的神经元与下一层中的所有神经元都有连接，但同一层内的神经元之间没有连接。

在这里，我们使用 sklearn.neural_network 中的 MLPClassifier 模块，进行多层感知机的训练。该模型会默认使用 ReLU(Rectified Linear Unit，修正线性单元)作为激活函数，默认使用 L2 正则化来防止过拟合。我们还规定 hidden_layer_sizes=(10,)，即该神经网络只有一个隐藏层，且该隐藏层中有 10 个神经元，规定 max_iter=1000，即训练的最大迭代次数，规定 random_state=42 作为确定的随机种子。代码的整体框架与实现 Logistic 回归时类似，多层感知机的具体实现与效果见图 10.

```
mlp_classifier = MLPClassifier(hidden_layer_sizes=(10,), max_iter=1000, random_state=42)
mlp_classifier.fit(text_train, label_train)
y_pred = mlp_classifier.predict(text_test)
```

```
平均运行时间为:  204.26693372726442
模型准确率:  0.9636000000000001
模型精确度:  0.9642711530691942
模型召回率:  0.9636000000000001
模型F1-score:  0.9635754716251934
```

图 10 多层感知机的代码实现

可以观察到，其准确率、精确度、召回率与 F1-Score 均可达到 96%，但每一次多层感知机的训练都需要耗费 204.2669 秒的时间，耗时较长。

我们观察到，如果忽略运行时间，多层感知机对文本分类有着非常优秀的表现。但是，它的运行时间却是一个实实在在无法忽略的问题。我们认为，有以下两个原因使得这次的代码运行时间较长。

1. 没有调用 GPU 运行深度学习

我们调用的是 sklearn 中的模块，它的优点是模块封装完全，我们需要调整的参数不多，调用便捷。但缺点是，我们无法选择调用 GPU 来运行我们的深度学习。在使用 TensorFlow 或 PyTorch 时，我们可以选择调用 GPU，而在 sklearn 模块下，我们只能使用默认的 CPU 来进行深度学习。然而，我们尝试了在 TensorFlow 模块中调用 GPU，却调用失败，这可能是因为运行代码的计算机默认使用了其内置的集成显卡，而没有调用其独立显卡。

2. 参数设置过大

我们设置的隐藏层只有一层，但我们也让隐藏层有 10 个神经元。实际上，如果我们做出

了良好的正则化，那么神经元越多，模型的容量就会被提升，模型也可以学习更复杂的模式。不过，它的一个显著代价就是可能会大大增加模型的训练时间。

基于上述分析，我们调整了参数，运行了只有 5 个神经元的多层感知机，得到了如图 11 的结果。

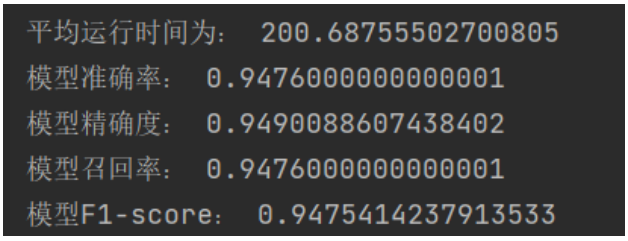

图 11 调整参数后的多层感知机运行结果

我们观察到，运行时间只有略微下降，为 200.6876 秒，但平均准确率、精确度、召回率与 F1-Score 却要比有 10 个神经元的时候要低约 2%.

我们再将隐藏层的神经元数量调整回 10，但此时我们将激活函数改为之前运行过的效果不错的 Logistic 函数，期待不一样的激活函数可以带来更好的预测效果。代码运行结果如图 12.

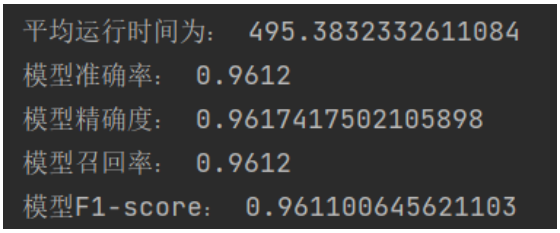

图 12 激活函数为 Logistic 函数的多层感知机运行结果

我们发现其平均运行时间达到了 495.3832 秒，这是之前训练的两个模型的耗时的两倍，但模型的准确率、精确度、召回率与 F1-Score 却没有得到显著提升。

这说明，深度学习是一个对参数敏感，且非常消耗算力的过程。尽管多层感知机有着良好的运行结果，但无论我们如何设置它的隐藏层，代价都是消耗非常多的算力与时间。这使得我们在实际应用中需要综合考量它的优劣。

4. 4. 4 决策树

决策树是一种常用的机器学习算法，它可以用于分类和回归任务。决策树是一种树形结构，其中每个内部节点代表一个属性上的决策测试，每个分支代表一个决策结果，而每个叶节点代表一个预测的类别。通过递归地分裂数据集，决策树试图将数据集划分为尽可能纯净的子集。

在这里，我们使用 sklearn.tree 中的 DecisionTreeClassifier 模块。我们先运行其中参数均

为默认参数的决策树模型，代码的整体框架与实现 Logistic 回归时类似，具体代码与运行结果如图 13.

```
dt_classifier = DecisionTreeClassifier(random_state=42)
dt_classifier.fit(text_train, label_train)
y_pred = dt_classifier.predict(text_test)
accuracy = dt_classifier.score(text_test, label_test)
```

```
平均运行时间为： 39.13006644248962
模型准确率： 0.7867999999999999
模型精确度： 0.7888546919511834
模型召回率： 0.7867999999999999
模型F1-score： 0.7866261113542182
```

图 13 默认参数下的决策树代码实现

我们发现，平均每次决策树运行的时间为 39.1301 秒，但平均准确率、精确度、召回率与 F1-Score 均只有约 78%，效果不佳。

我们初步怀疑，这是因为我们对于决策树的参数没有一个非常好的调整与优化。然而，研究了决策树中可以调节的参数后，我们发现决策树模型中可以调节的参数大多数都涉及减小模型的复杂度与避免模型的过拟合（如 max_depth, min_samples_split, min_samples_leaf, max_leaf_nodes）。但目前我们的模型遇到的最严重的问题在于模型的预测效果极其不理想，而调整这些参数对模型的预测效果可能没有显著的影响。因此，我们放弃调整这些参数，只调整衡量决策树分裂质量的标准，即 criterion 改为 entropy，观察代码运行结果。代码运行结果如图 14.

```
平均运行时间为： 31.50426330566406
模型准确率： 0.7426
模型精确度： 0.7496185196236086
模型召回率： 0.7426
模型F1-score： 0.7441254927342695
```

图 14 Criterion 被调整为 entropy 的决策树运行结果

我们发现，平均运行时间有所下降，但模型的准确率、精确度、召回率与 F1-Score 也下降到了约 74%，这是一个更加不理想的结果。

实际上，决策树常常被认为不是一个非常理想的文本分类模型。这可能是因为，TF-IDF 向量化的文本数据是高维且稀疏的，特征空间会非常大且大部分特征的取值为 0，决策树在处理这种高维稀疏的数据时可能没有一个很良好的表现。因此，如果数据呈现离散型特征或低维的特征，那么决策树可以是一个不错的方法，但在本题中它的表现就略显逊色了。

4.4.5 *TextCNN(Text Convolutional Neural Networks)

本次实验更多的是一种对于机器学习、深度学习的体验，而上述的各种方法更多还是侧重于传统机器学习方法。因此，我们计划尝试 TextCNN 的神经网络方法，调用 tensorflow 与 keras 模块，以获得更多深度学习方面的体验。我们对于其根本原理并不做过多深究，这里只是尝试调用 tensorflow 与 keras 模块，为未来学习作铺垫。

TextCNN 是一种可以被用于文本分类任务的深度学习模型，它采用了卷积神经网络(Convolutional Neural Networks)来处理文本数据，基本思想是通过卷积层来提取文本中的局部特征，然后通过池化层降维并保留最显著的特征，最后将处理后的特征输入到全连接层进行分类。

运行 TextCNN 需要我们使用 tensorflow.keras 中的部分模块。调用的所有模块如图 15.

```
import numpy as np
import time
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Conv1D, GlobalMaxPooling1D, Dense
```

图 15 运行 TextCNN 所调用的模块

（请忽视红色的下划线，这些模块实际上已经被安装，系 DataSpell 的问题）

我们先试图基于 TF-IDF 向量化的文本进行 TextCNN 的运行。我们使用了 128 个卷积核，每个卷积核的大小为 5，激活函数为 ReLU，并使用了全局最大池化层以提取出每个卷积核的最强特征，并设置有 10 个神经元的输出层，利用 Softmax 激活函数进行多分类。最后，使用 Adam 作为优化器和稀疏分类交叉熵作为损失函数来编译模型，训练模型时取用数据的 12.5%作为验证集，其余数据作为训练集，并且训练 5 个周期，以准确率作为模型训练后的评估指标。具体代码与效果见图 16.

```
start_time = time.time()
model = Sequential()
model.add(Conv1D(128, 5, activation='relu', input_shape=(X.shape[1], 1)))
model.add(GlobalMaxPooling1D())
model.add(Dense(10, activation='softmax'))
# 编译模型
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# 训练模型
model.fit(X.reshape(X.shape[0], X.shape[1], 1), y, epochs=5, validation_split=0.125)
# 评估模型
loss, accuracy = model.evaluate(X.reshape(X.shape[0], X.shape[1], 1), y)
end_time = time.time()
print("精确度为:", accuracy)
print("运行时间为:", end_time - start_time)
```

```
精确度为: 0.140625
运行时间为: 1215.3077547550201
```

图 16 TextCNN 的实现与效果

我们发现，其不但速度极慢，且训练期间的准确率也一直显示在 10%左右，这是一个非常不理想的效果。

经过分析，我们认为导致 TextCNN 性能如此不理想的一个很大的原因在于我们使用了 TF-IDF 作为其输入。TF-IDF 是一个维度较高且极为稀疏的矩阵，它不一定适用于卷积神经网络。因此，我们计划对 TextCNN 更换一种文本转换为向量的方式。

在这里，我们采用了 keras 模块中可以将文本转化为数字的实用工具 **Tokenizer** 作为文本转化为向量的方式。我们规定词汇表大小为 10000，向量最大长度为 100，观察运行效果，其具体实现如图 17.

```python
# 设置词汇表大小
max_words = 10000
# 设置文本最大长度
maxlen = 100
# 使用Tokenizer进行文本转换为向量的操作
# 可以过滤大量标点符号
tokenizer = Tokenizer(num_words=max_words, filters='!"#$%&()*+,-./:;<=>?@[\]^_`{|}~\t\n')
tokenizer.fit_on_texts(X)
sequences = tokenizer.texts_to_sequences(X)
X = pad_sequences(sequences, maxlen=maxlen)
# 将标签转换为数组
Y = np.array(Y)
# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.125, random_state=42)
# 构建模型
model = Sequential()
model.add(Embedding(max_words, 128, input_length=maxlen))
model.add(Conv1D(128, 5, activation='relu'))
model.add(GlobalMaxPooling1D())
model.add(Dense(10, activation='softmax'))
# 编译模型
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# 训练模型
model.fit(X_train, y_train, epochs=5, validation_data=(X_test, y_test))
# 评估模型
loss, accuracy = model.evaluate(X_test, y_test)
```

```
Epoch 1/5
219/219 [==============================] - 7s 28ms/step - loss: 1.5835 - accuracy: 0.5283 - val_loss: 0.6800 - val_accuracy: 0.7810
Epoch 2/5
219/219 [==============================] - 6s 27ms/step - loss: 0.4437 - accuracy: 0.8586 - val_loss: 0.3928 - val_accuracy: 0.8560
Epoch 3/5
219/219 [==============================] - 6s 27ms/step - loss: 0.1813 - accuracy: 0.9489 - val_loss: 0.3236 - val_accuracy: 0.8890
Epoch 4/5
219/219 [==============================] - 6s 28ms/step - loss: 0.0652 - accuracy: 0.9909 - val_loss: 0.3104 - val_accuracy: 0.8960
Epoch 5/5
219/219 [==============================] - 6s 28ms/step - loss: 0.0237 - accuracy: 0.9987 - val_loss: 0.3023 - val_accuracy: 0.9030
32/32 [==============================] - 0s 5ms/step - loss: 0.3023 - accuracy: 0.9030
精确度为: 0.902999997138977
运行时间为: 32.246939182281494
```

图 17 使用 Tokenizer 将文本转化为向量后的 TextCNN

观察到，运行时间为 32.2469 秒，精确度则有约 90%.

我们进一步调整其中的参数，将词汇表大小设置为 50000，向量最大长度拉长为 200，允许向量承载更多文本特征，再次运行代码，观察效果，运行结果如图 18.

```
Epoch 1/5
219/219 [==============================] - 22s 96ms/step - loss: 1.5082 - accuracy: 0.5657 - val_loss: 0.5668 - val_accuracy: 0.8310
Epoch 2/5
219/219 [==============================] - 21s 94ms/step - loss: 0.3673 - accuracy: 0.8930 - val_loss: 0.3118 - val_accuracy: 0.8980
Epoch 3/5
219/219 [==============================] - 21s 95ms/step - loss: 0.1354 - accuracy: 0.9666 - val_loss: 0.2549 - val_accuracy: 0.9110
Epoch 4/5
219/219 [==============================] - 21s 95ms/step - loss: 0.0446 - accuracy: 0.9931 - val_loss: 0.2272 - val_accuracy: 0.9230
Epoch 5/5
219/219 [==============================] - 21s 95ms/step - loss: 0.0163 - accuracy: 0.9989 - val_loss: 0.2241 - val_accuracy: 0.9290
32/32 [==============================] - 0s 7ms/step - loss: 0.2241 - accuracy: 0.9290
精确度为: 0.9290000200271606
运行时间为: 106.37823724746704
```

图 18 调整参数后的 Tokenizer 化的 TextCNN 运行结果

可以观察到，运行时间显著增长了，为 106.3782 秒，但精确度有了进一步提升，约为 93%，这说明 Tokenizer 向量化的向量维度越高，其能承载的特征越多，最后预测的效果就越好，但训练模型也需要消耗更多的时间。

4.5 最终选择的分类器

在我们尝试了基于 Word2Vec 的 Logistic 回归、基于 TF-IDF 的支持向量机、基于 TF-IDF 的多层感知机、基于 TF-IDF 的决策树、基于 TF-IDF 的 Logistic 回归与基于 Tokenizer 的 TextCNN 后，我们发现我们所使用的基于 TF-IDF 的多层感知机模型在多折交叉验证中的表现较好。因此，我们将对 test.txt 中的数据通过我们配置的基于 TF-IDF 的多层感知机进行预测。预测结果见 results.txt，预览见图 19.

```
id, pred
0, 6
1, 1
2, 4
3, 1
4, 6
5, 7
6, 0
7, 4
8, 9
9, 7
10, 6
```

图 19 预测结果示例图

五、 总结与分析

在本次实验中，我们分析了不同文本转化为向量的方法，分析了不同的分类器，体验了编写与分析机器学习、深度学习的代码，理解了多折交叉验证的思想。

以下是对文本映射为向量的方法的概括性理解。

TF-IDF：一个可以根据词语在一段文本与整个语料库中出现频率来将文本映射为向量的

方式。优点是便于理解，且如果涉及的文本问题主要取决于词语出现的频率，则可以有出色的表现，如该方法在本题中有着出色的表现。缺点是，只考虑了词语出现的频率但没有考虑词语之间可能存在的联系，缺乏泛用性，如果抛开本题的背景，可能该方法会失去亮点。

Word2Vec：一个可以根据词语上下文来将词语转化为向量的方法。优点是，可以综合考量了一个词语所在的上下文，更具有泛用性。缺点是，它在衡量一段由大量词语组成的文本上可能表现不会非常出彩。在本题中，它之所以没有非常优秀的表现，可能是因为本题中的文本分类标准仅仅是依靠个别出现频率较高的词语，或我们设置的参数过小，或以 Word2Vec 衡量文本的方式并不合适。

Tokenizer：一个可以根据词语在一段文本的出现频率来将文本映射为向量的方式。它的优点是可以限制向量的特征数量，使得模型的规模适中，并且它是 keras 模块是一部分，可以快速地在神经网络中被使用。缺点是，它也只考虑了词汇的出现频率，没有考虑词语之间的上下文关系。在本题中，我们只配合了需要调用 keras 模块的 TextCNN 来使用。

而在对分类方法的理解中，我们认为除了更适合离散型数据与低维数据的**决策树**在文本分类任务中可能总体表现欠佳以外，其他方法在文本分类中均有着不错的表现。而在其中，**Logistic 回归**与**支持向量机**作为传统机器学习方法的代表，他们运行时间较短，效果良好，可解释性强。**多层感知机**与 **TextCNN** 由于本身为深度学习算法，因此准确性可能比其他方法要高，但模型会比传统机器学习要复杂，耗时相比其他算法会更久，可解释性也较弱。

不过，在实际使用中，对模型的总体评价也与数据本身的特点和实际需求有着非常大的联系，这使得我们在实际运用中需要综合考量各种因素，选择最合适当下情况的算法，而不是单一地评价一个算法的耗时或在一个验证集上的准确度。

六、 参考技术博客

[1] tf-idf 原理 & TfidfVectorizer 参数详解及实战
URL: https://blog.csdn.net/a7303349/article/details/126575663
[2] sklearn-TfidfVectorizer 彻底说清楚
URL: https://zhuanlan.zhihu.com/p/67883024
[3] python 逻辑回归模型多分类
URL: https://blog.51cto.com/u_16175495/6811063
[4] “机器学习”系列之 SVM（支持向量机）
URL: https://aistudio.baidu.com/projectdetail/1691063?ad-from=1694
[5] 多层感知机（MLP）简介
URL: https://blog.csdn.net/fg13821267836/article/details/93405572
[6] TextCNN 原理、结构、代码
URL: https://blog.51cto.com/u_15899958/5909754
[7] TextCNN
URL: https://blog.csdn.net/weixin_46125345/article/details/120907765
[8] python 函数——Keras 分词器 Tokenizer
URL: https://cloud.tencent.com/developer/article/1694921