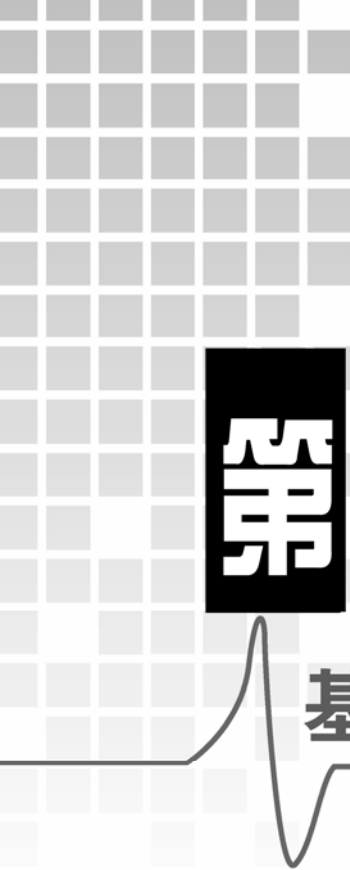




第2章

基于内容的推荐



基于内容的推荐是推荐系统的一种方法，它基于物品(如文章、音乐、电影等)的内容特征和用户的偏好，为用户提供个性化的推荐。这种推荐方法主要依靠对物品的内容进行分析和比较，以确定物品之间的相似性，以及与用户的兴趣匹配度。本章将详细讲解基于内容推荐的知识。



2.1 文本特征提取

文本特征提取是将文本数据转换为可供机器学习算法或其他自然语言处理任务使用的特征表示的过程。文本特征提取的目标是将文本中的信息转化为数值或向量形式，以便计算机可以理解和处理。



扫码看视频

2.1.1 词袋模型

词袋(bag-of-words)模型是一种常用的文本特征表示方法，用于将文本转换为数值形式，以便于机器学习算法的处理。该模型基于假设，认为文本中的词语顺序并不重要，只关注词语的出现频率。词袋模型的基本思想是将文本视为一个袋子(或集合)，并忽略词语之间的顺序。在构建词袋模型时，首先需要进行以下几个操作。

- ❑ 分词(tokenization): 将文本划分为词语或其他有意义的单元。通常使用空格或标点符号来分隔词语。
- ❑ 构建词表(vocabulary): 将文本中的所有词语收集起来构建一个词表，其中每个词语都对应一个唯一的索引。
- ❑ 计算词频(term frequency): 对于每个文本样本，统计每个词语在该样本中出现的频率。可以用一个向量表示每个样本的词频，其中向量的维度与词表的大小相同。

通过上述步骤，可以将每个文本样本转换为一个向量，其中向量的每个维度表示对应词语的出现频率或其他相关特征。这样就可以将文本数据转换为数值形式，供机器学习算法使用。

注意：词袋模型的优点是简单易用，适用于大规模文本数据，并能够捕捉到词语的出现频率信息。然而，词袋模型忽略了词语之间的顺序和上下文信息，可能会丢失一部分语义和语境的含义。

在 Python 中，有多种工具和库可用于实现词袋模型，具体说明如下。

1. scikit-learn

在 scikit-learn 库中提供了用于实现文本特征提取的类 CountVectorizer 和 TfidfVectorizer，例如，下面的实例演示了使用 scikit-learn 库实现词袋模型，并基于相似度计算进行推荐。读者可以根据自己的具体数据集和应用场景，自定义和扩展这个例子，构建更复杂和个性化的推荐系统。

源码路径: daima/2/skci.py

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# 电影数据集
movies = [
    'The Shawshank Redemption',
    'The Godfather',
    'The Dark Knight',
    'Pulp Fiction',
    'Fight Club'
]

# 电影简介数据集
synopsis = [
    'Two imprisoned men bond over a number of years, finding solace and eventual redemption through acts of common decency.',
    'The aging patriarch of an organized crime dynasty transfers control of his clandestine empire to his reluctant son.',
    'When the menace known as the Joker wreaks havoc and chaos on the people of Gotham, Batman must accept one of the greatest psychological and physical tests of his ability to fight injustice.',
    'The lives of two mob hitmen, a boxer, a gangster and his wife, and a pair of diner bandits intertwine in four tales of violence and redemption.',
    'An insomniac office worker and a devil-may-care soapmaker form an underground fight club that evolves into something much, much more.'
]

# 构建词袋模型
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(synopsis)

# 计算文本之间的相似度
similarity_matrix = cosine_similarity(X)

# 选择一部电影, 获取相似推荐
movie_index = 0 # 选择第一部电影作为例子
similar_movies = similarity_matrix[movie_index].argsort()[::-1][1:]

print(f"根据电影 '{movies[movie_index]}' 推荐的相似电影: ")
for movie in similar_movies:
    print(movies[movie])
```

在上述代码中, 首先, 定义了一个包含电影标题和简介的数据集。然后, 使用 `scikit-learn` 库中的 `CountVectorizer` 类来构建词袋模型, 将文本数据转换为词频向量表示。接下来, 使用 `cosine_similarity` 计算文本之间的余弦相似度, 得到一个相似度矩阵。最后, 选择一部电



影，根据它在相似度矩阵中的索引，获取相似度最高的电影推荐。执行代码后会输出：

```
根据电影 'The Shawshank Redemption' 推荐的相似电影：
Pulp Fiction
The Dark Knight
The Godfather
Fight Club
```

2. NLTK

在库 NLTK(natural language toolkit)中提供了用于实现文本分词和特征提取的函数和工具。例如，下面是一个使用 NLTK 库实现词袋模型的基础例子，功能是针对一个电影评论数据集，能根据评论内容来进行情感分类。

源码路径：`daima/2/nldk.py`

```
# 下载电影评论数据集
nltk.download('movie_reviews')

# 加载电影评论数据集
reviews = [(list(movie_reviews.words(fileid)), category)
            for category in movie_reviews.categories()
            for fileid in movie_reviews.fileids(category)]

# 构建词袋模型
all_words = [word.lower() for review in reviews for word in review[0]]
all_words_freq = FreqDist(all_words)
word_features = list(all_words_freq)[:2000]

# 定义特征提取函数
def extract_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
        features[word] = (word in document_words)
    return features

# 构建特征集
featuresets = [(extract_features(review), category) for (review, category) in reviews]

# 划分训练集和测试集
train_set = featuresets[:1500]
test_set = featuresets[1500:]

# 使用朴素贝叶斯分类器进行分类
classifier = nltk.NaiveBayesClassifier.train(train_set)
```



```
# 测试分类器的准确率
accuracy = nltk.classify.accuracy(classifier, test_set)
print("分类器准确率:", accuracy)

# 使用 SVM 分类器进行分类
svm_classifier = SklearnClassifier(SVC())
svm_classifier.train(train_set)

# 测试 SVM 分类器的准确率
svm_accuracy = nltk.classify.accuracy(svm_classifier, test_set)
print("SVM 分类器准确率:", svm_accuracy)
```

在上述代码中, 首先, 下载了 NLTK 库的电影评论数据集, 加载评论数据并进行词袋模型的构建。通过计算词频, 选择出现频率最高的 2000 个词语作为特征。接下来, 定义一个特征提取函数, 将每个评论文本转换为特征向量表示。然后, 构建特征集, 并将它划分为训练集和测试集。最后, 使用朴素贝叶斯分类器进行情感分类, 并计算分类器的准确率。另外, 使用 SVM 分类器进行分类并计算准确率。执行代码后会输出:

```
[nltk_data] Downloading package movie_reviews to
[nltk_data] C:\Users\apple\AppData\Roaming\nltk_data...
[nltk_data] Unzipping corpora\movie_reviews.zip.
分类器准确率: 0.78
SVM 分类器准确率: 0.616
```

上述例子演示了如何使用 NLTK 库实现词袋模型, 并应用于情感分类任务。大家可以根据自己的数据集和任务需求进行定制和扩展。

3. Gensim

Gensim 是一个用于主题建模和文本相似度计算的库, 也可以用于词袋模型的构建。例如, 下面的实例演示了使用 Gensim 库实现词袋模型的过程, 功能是针对一个新闻文章数据集, 能根据文章内容推荐相似的新闻。

源码路径: `daima/2/recommendation.py`

```
from gensim import models, similarities
from gensim.corpora import Dictionary

# 新闻文章数据集
documents = [
    "The economy is going strong with positive growth.",
    "Unemployment rates are decreasing, indicating a robust job market.",
    "Stock market is experiencing a bull run, with high trading volumes.",
    "Inflation remains low, providing stability to the economy."
]
```



```
# 分词和建立词袋模型
texts = [[word for word in document.lower().split()] for document in documents]
dictionary = Dictionary(texts)
corpus = [dictionary.doc2bow(text) for text in texts]

# 训练 TF-IDF 模型
tfidf = models.TfidfModel(corpus)
corpus_tfidf = tfidf[corpus]

# 构建相似度索引
index = similarities.MatrixSimilarity(corpus_tfidf)

# 选择一篇文章，获取相似推荐
article_index = 0 # 选择第一篇文章作为例子
similarities = index[corpus_tfidf[article_index]]

# 按相似度降序排列并打印推荐文章
sorted_indexes = sorted(range(len(similarities)), key=lambda i: similarities[i],
reverse=True)
print(f"根据文章 '{documents[article_index]}' 推荐的相似文章:")
for i in sorted_indexes[1:]:
    print(documents[i])
```

在上述代码中，首先，定义了一个包含新闻文章的数据集。随后，使用 **Gensim** 库对文章进行分词，并构建词袋模型。接下来，训练 **TF-IDF** 模型来计算每个词语的重要性，并使用 **TF-IDF** 模型转换文档向量，构建语料库。然后，构建相似度索引，将语料库中的每个文档转换为特征向量表示。最后，选择一篇文章作为例子，计算与其他文章的相似度，并将文章根据相似度降序排列，打印推荐的相似文章。执行代码后会输出：

```
根据文章 'The economy is going strong with positive growth.' 推荐的相似文章:
Stock market is experiencing a bull run, with high trading volumes.
Inflation remains low, providing stability to the economy.
Unemployment rates are decreasing, indicating a robust job market.
```

2.1.2 n-gram 模型

在推荐系统中，**n-gram** 模型是一种基础的文本建模技术，能捕捉词序列的局部信息。它是一种基于概率的统计模型，用于预测给定文本序列中下一个词或字符的可能性。**n-gram** 模型中的 **n** 表示模型考虑的词语或字符的数量。例如，一个 **2-gram** 模型(也称为 **bigram** 模型)会考虑每个词的上下文中的前一个词，而一个 **3-gram** 模型(也称为 **trigram** 模型)会考虑前两个词。**n-gram** 模型的基本假设是：前词的出现仅依赖前面的 **n-1** 个词。通过观察大量文

本数据可知，**n-gram** 模型可以学习到不同词语之间的频率和概率分布，从而对下一个词的出现进行预测。

在 Python 程序中，NLTK 库提供了一些工具和函数，用于构建 **n-gram** 模型并进行文本生成和预测。例如，下面的实例演示了使用库 NLTK 实现 *n-gram* 模型的过程。

源码路径：`daima/2/ngram.py`

```
import nltk
nltk.download('punkt')
from nltk import ngrams

# 商品列表
products = [
    "Apple iPhone 12",
    "Samsung Galaxy S21",
    "Google Pixel 5",
    "Apple iPad Pro",
    "Samsung Galaxy Tab S7",
    "Microsoft Surface Pro 7"
]

# 构建 n-gram 模型
n = 2 # n-gram 模型中考虑的词语数量
product_tokens = [product.lower().split() for product in products]
product_ngrams = [list(ngrams(tokens, n)) for tokens in product_tokens]

# 用户输入查询
query = "Apple iPhone"

# 根据查询匹配推荐商品
query_tokens = query.lower().split()
query_ngrams = list(ngrams(query_tokens, n))

recommended_products = []
for i in range(len(products)):
    count = 0
    for query_ngram in query_ngrams:
        if query_ngram in product_ngrams[i]:
            count += 1
    if count == len(query_ngrams):
        recommended_products.append(products[i])

print("Recommended Products:")
for product in recommended_products:
    print(product)
```




上述代码的具体说明如下。

- ❑ 导入 NLTK 库，并从中导入 `ngrams` 函数。
- ❑ 定义商品列表：创建了一个包含不同商品名称的列表。
- ❑ 构建 `n-gram` 模型：将商品名称分成单词，并使用 `ngrams` 函数生成 `n-gram` 序列。这里指定 `n` 的值为 2，表示使用二元组(bigram)作为 `n-gram` 模型。
- ❑ 用户查询输入：定义一个查询字符串，例如“Apple iPhone”。
- ❑ 根据查询匹配推荐商品：将查询字符串分成单词，并生成相应的 `n-gram` 序列。然后遍历商品列表，并对每个商品的 `n-gram` 序列进行匹配。如果查询的所有 `n-gram` 都在商品的 `n-gram` 序列中出现，则认为该商品与查询相关，并将它添加到推荐列表中。
- ❑ 输出推荐商品：最后打印出推荐的相关商品列表。

执行代码后会输出：

```
Recommended Products:  
Apple iPhone 12
```

注意：这只是一个简化的例子，用于说明如何使用 NLTK 库实现基于 `n-gram` 的推荐系统。实际的推荐系统可能包含更多的步骤和复杂的算法，能处理更大规模的数据和更复杂的推荐逻辑。

2.1.3 特征哈希

特征哈希(feature hashing)是一种常用的特征处理技术，用于将高维特征向量映射到固定长度的哈希表中。在推荐系统中，特征哈希可用于处理稀疏的特征数据，减少内存消耗并加快计算速度。特征哈希的基本原理如下。

- ❑ 特征表示：在推荐系统中，通常使用特征来表示用户和物品，例如用户的年龄、性别、浏览历史，物品的类别、标签等。这些特征可以形成一个高维的特征向量。
- ❑ 特征哈希函数：特征哈希使用哈希函数将高维特征向量映射到固定长度的哈希表中。哈希函数将特征的取值范围映射到一个固定大小的哈希表索引。通常哈希函数的输出是一个整数，表示特征在哈希表中的位置。
- ❑ 哈希表存储：哈希表可以使用数组或其他数据结构来表示。每个特征都对应哈希表中的一个位置，可以将特征的取值作为索引，将特征的计数或权重作为值存储在哈希表中。
- ❑ 特征编码：对于每个样本，通过特征哈希函数将特征向量映射到哈希表中，并根

据哈希表的索引位置将特征编码为一个固定长度的特征向量。这个特征向量可以作为输入用于训练推荐系统的模型。

特征哈希的主要优点是简单高效，适用于处理大规模的稀疏特征数据。它可以减少内存消耗，因为哈希表的大小是固定的，不受原始特征向量维度的影响。此外，特征哈希还能加快计算速度，因为哈希函数比完整的特征向量计算速度更快。

在 Python 程序中，可以使用类 `sklearn.feature_extraction.FeatureHasher` 实现特征哈希处理，并将哈希后的特征用于推荐系统的特征工程和模型训练。假设现在有一个电影推荐系统，其中每部电影有以下特征：电影名称、电影类型、导演、演员。我们可以使用特征哈希来处理这些特征，并将它们转换为固定长度的特征向量。下面是一个使用特征哈希处理上述电影特征的例子。

源码路径: `daima/2/teha.py`

```
from sklearn.feature_extraction import FeatureHasher

# 电影数据集
movies = [
    {"movie_id": 1, "title": "Movie A", "genre": "Action", "director": "Director X",
     "actors": ["Actor A", "Actor B"]},
    {"movie_id": 2, "title": "Movie B", "genre": "Comedy", "director": "Director Y",
     "actors": ["Actor B", "Actor C"]},
    {"movie_id": 3, "title": "Movie C", "genre": "Drama", "director": "Director Z",
     "actors": ["Actor A", "Actor C"]}
]

# 将列表类型的特征转换为字符串
for movie in movies:
    movie["actors"] = ", ".join(movie["actors"])

# 特征哈希处理
hasher = FeatureHasher(n_features=5, input_type="dict")
hashed_features = hasher.transform({movie["movie_id"]: movie for movie in movies}).values()

# 打印特征哈希处理后的特征向量
for i, movie in enumerate(movies):
    print(f"Movie ID: {movie['movie_id']}")
    print(f"Title: {movie['title']}")
    print(f"Hashed Features: {hashed_features[i].toarray()[0]}")
    print("-----")
```

上述代码的具体说明如下。

(1) 创建电影数据集，其中包含每部电影的一些特征，如电影 ID、标题、类型、导演和演员列表。



(2) 需要对演员列表进行处理，将它从列表类型转换为以逗号分隔的字符串，以确保特征是字符串类型。

(3) 使用类 `FeatureHasher` 进行特征哈希处理。其中指定了哈希处理后的特征向量的长度(`n_features`)为 5，并设置输入类型为字典(`input_type="dict"`)。

(4) 将电影数据集转换为字典形式，并使用电影 ID 作为字典的键，电影特征作为字典的值。

(5) 使用特征哈希器对字典形式的电影特征进行转换，得到哈希处理后的特征向量。再循环遍历每部电影，打印出电影 ID、标题以及对应的哈希处理后的特征向量。

执行代码后，会输出每部电影的 ID、标题以及对应的特征哈希向量。具体输出的内容取决于电影数据集的内容，每行包含一部电影的信息。本例输出内容如下：

```
Movie ID: 1
Title: Movie A
Hashed Features: [-3.  0.  1.  0.  1.]
-----
Movie ID: 2
Title: Movie B
Hashed Features: [-2.  0.  0. -1. -1.]
-----
Movie ID: 3
Title: Movie C
Hashed Features: [-3.  0. -1. -2. -1.]
-----
```

总体来说，上述代码演示了如何使用特征哈希对电影特征进行处理，将其转换为固定长度的特征向量。这种方法适用于处理高维稀疏特征的情况，能提高计算效率和降低存储成本。

2.2 TF-IDF(词频-逆文档频率)

TF-IDF(Term Frequency-Inverse Document Frequency)是一种用于评估文本中词语重要性的统计算法，它结合了词频(TF)和逆文档频率(IDF)两个指标。

在推荐系统中，逆文档频率通常与词频结合使用，形成 TF-IDF 特征表示。TF-IDF 通过考虑一个词语在当前文本中的频率(TF)，以及它在整个文本集中的普遍性和独特性(IDF)，综合评估词语的重要性。TF-IDF 的计算公式如下：

$$\text{TF-IDF} = \text{TF} * \text{IDF}$$

其中，TF 是词频，IDF 是逆文档频率。



扫码看视频

通过计算一个词语的 TF-IDF 值，我们可以确定该词语在文档中的重要性。当一个词语的词频较高且在整个文档集中出现的次数较少时，它的 TF-IDF 值将更高，表示它在该文档中具有更高的重要性。

TF-IDF 常用于信息检索、文本挖掘和推荐系统等任务中，用于计算文档之间的相似度或衡量词语的重要性，以便于进行文本分析和自动化处理。

2.2.1 词频计算

词频是指一个词语在文本中出现的频率，用于衡量一个词语在给定文本中的重要程度。词频可以通过计算一个词语在文本中出现的次数来获取。在推荐系统中，词频计算是一种基础的文本特征计算方法，用于评估文本中词语的重要性和出现的频率。

在 Python 中，可以使用各种库和方法来计算词频。例如，下面的实例演示了使用库 NLTK 来计算词频的过程。

源码路径: daima/2/cipin.py

```
import nltk
from nltk import FreqDist

# 推荐系统的用户评价数据
reviews = [
    "This movie is great!",
    "I love this movie so much.",
    "The acting in this film is superb.",
    "The plot of this movie is confusing.",
    "I didn't enjoy this film."
]

# 将所有评价合并为一个字符串
text = ' '.join(reviews)

# 分词
tokens = nltk.word_tokenize(text)

# 计算词频
freq_dist = FreqDist(tokens)

# 输出词频统计结果
for word, frequency in freq_dist.items():
    print(f"Word: {word}, Frequency: {frequency}")
```

在上述代码中，用户对电影的评价数据存储在 reviews 列表中。首先，将所有评价合并为一个字符串。然后，使用 nltk.word_tokenize() 方法对字符串进行分词，得到一个词语列



表。接下来，使用 `FreqDist` 类计算词频，生成一个词频分布对象。最后，通过遍历词频分布对象，打印出每个词语及其对应的词频。执行代码后会输出：

```
Word: This, Frequency: 1
Word: movie, Frequency: 3
Word: is, Frequency: 3
Word: great, Frequency: 1
Word: !, Frequency: 1
Word: I, Frequency: 2
Word: love, Frequency: 1
Word: this, Frequency: 4
Word: so, Frequency: 1
Word: much, Frequency: 1
Word: ., Frequency: 4
Word: The, Frequency: 2
Word: acting, Frequency: 1
Word: in, Frequency: 1
Word: film, Frequency: 2
Word: superb, Frequency: 1
Word: plot, Frequency: 1
Word: of, Frequency: 1
Word: confusing, Frequency: 1
Word: did, Frequency: 1
Word: n't, Frequency: 1
Word: enjoy, Frequency: 1
```

本实例展示了如何使用词频计算来分析用户评价数据。通过统计词语出现的频率，我们可以了解哪些词语在用户评价中出现得更频繁，从而帮助推荐系统更好地理解用户的喜好和偏好。基于词频的分析结果，推荐系统可以提供与用户评价相关的电影推荐或者进一步完成文本情感分析等任务。

2.2.2 逆文档频率计算

逆文档频率是推荐系统中常用的一种特征权重计算方法，它衡量了一个词语在文本集合中的重要程度。

下面是一个使用 Python 计算逆文档频率的例子，其中假设有一个文本集合存储在列表 `documents` 中。

源码路径: `daima/2/niwen.py`

```
import math
from collections import Counter

# 文本集合
```

```

documents = [
    "This is the first document.",
    "This document is the second document.",
    "And this is the third one.",
    "Is this the first document?"
]

# 分词并去重
word_sets = [set(document.lower().split()) for document in documents]

# 计算逆文档频率
idf = {}
num_documents = len(documents)
for word in set(word for word_set in word_sets for word in word_set):
    count = sum(1 for word_set in word_sets if word in word_set)
    idf[word] = math.log(num_documents / (count + 1))

# 输出逆文档频率
for word, idf_value in idf.items():
    print(f"Word: {word}, IDF: {idf_value}")

```

在上述代码中，首先对每个文本进行分词，并去除重复的词语，得到一个词语集合。然后遍历所有词语的集合，计算每个词语的逆文档频率。逆文档频率的计算公式是 $\log(N / (n + 1))$ ，其中， N 表示文本集合中的文档数， n 表示包含当前词语的文档数。最后，打印输出每个词语及其对应的逆文档频率。执行代码后会输出：

```

Word: this, IDF: -0.2231435513142097
Word: third, IDF: 0.6931471805599453
Word: second, IDF: 0.6931471805599453
Word: document?, IDF: 0.6931471805599453
Word: first, IDF: 0.28768207245178085
Word: is, IDF: -0.2231435513142097
Word: one., IDF: 0.6931471805599453
Word: document, IDF: 0.6931471805599453
Word: and, IDF: 0.6931471805599453
Word: document., IDF: 0.28768207245178085
Word: the, IDF: -0.2231435513142097

```

注意：通过逆文档频率的计算，可以帮助推荐系统识别那些在整个文本集合中相对不常见但在当前文本中出现较多的词语。这些词语通常具有一定的独特性和重要性，因此，在推荐系统中起到一定的权重作用。通过将逆文档频率与词频结合，可以构建出更具表达力的特征表示，用于执行推荐系统中的任务，例如文本相似度计算、文本分类等。



2.2.3 TF-IDF 权重计算

TF-IDF 是一种常用的特征权重计算方法,通过将词频与逆文档频率相乘得到特征权重,用于衡量一个词语在文本中的重要性。和前面介绍的 IDF 相比,TF-IDF 的优势在于,它不仅考虑了词语在单个文档中的出现频率(TF),还考虑了词语在整个文档集中的稀有程度(IDF),从而更准确地衡量词语的重要性。这种结合使得 TF-IDF 能够突出显示在特定文档中重要但在整体文档集中不常见的词语,增强了特征的区分能力。例如,下面是一个使用 Python 程序计算 TF-IDF 权重的例子。

源码路径: daima/2/quan.py

```
from sklearn.feature_extraction.text import TfidfVectorizer

# 文本集合
documents = [
    "This is the first document.",
    "This document is the second document.",
    "And this is the third one.",
    "Is this the first document?"
]

# 创建 TF-IDF 向量化器
vectorizer = TfidfVectorizer()

# 对文本集合进行向量化
tfidf_matrix = vectorizer.fit_transform(documents)

# 输出词语和对应的 TF-IDF 权重
feature_names = vectorizer.get_feature_names()
for i in range(len(documents)):
    doc = documents[i]
    feature_index = tfidf_matrix[i, :].nonzero()[1]
    tfidf_scores = zip(feature_index, [tfidf_matrix[i, x] for x in feature_index])
    for word_index, score in tfidf_scores:
        print(f"Document: {doc}, Word: {feature_names[word_index]}, TF-IDF Score: {score}")
```

在上述代码中,使用了库 `scikit-learn` 中的类 `TfidfVectorizer` 来计算 TF-IDF 权重。首先,创建一个 TF-IDF 向量化器对象 `vectorizer`。然后,将文本集合 `documents` 传入向量化器的 `fit_transform()` 方法,得到 TF-IDF 矩阵 `tfidf_matrix`。最后,遍历每个文本和对应的 TF-IDF 向量,打印输出词语和对应的 TF-IDF 权重。执行代码后会输出:


```

Document: This is the first document., Word: document, TF-IDF Score: 0.46979138557992045
Document: This is the first document., Word: first, TF-IDF Score: 0.5802858236844359
Document: This is the first document., Word: the, TF-IDF Score: 0.38408524091481483
Document: This is the first document., Word: is, TF-IDF Score: 0.38408524091481483
Document: This is the first document., Word: this, TF-IDF Score: 0.38408524091481483
Document: This document is the second document., Word: second, TF-IDF Score:
0.5386476208856763
Document: This document is the second document., Word: document, TF-IDF Score:
0.6876235979836938
Document: This document is the second document., Word: the, TF-IDF Score: 0.281088674033753
Document: This document is the second document., Word: is, TF-IDF Score: 0.281088674033753
Document: This document is the second document., Word: this, TF-IDF Score:
0.281088674033753
Document: And this is the third one., Word: one, TF-IDF Score: 0.511848512707169
Document: And this is the third one., Word: third, TF-IDF Score: 0.511848512707169
Document: And this is the third one., Word: and, TF-IDF Score: 0.511848512707169
Document: And this is the third one., Word: the, TF-IDF Score: 0.267103787642168
Document: And this is the third one., Word: is, TF-IDF Score: 0.267103787642168
Document: And this is the third one., Word: this, TF-IDF Score: 0.267103787642168
Document: Is this the first document?, Word: document, TF-IDF Score: 0.46979138557992045
Document: Is this the first document?, Word: first, TF-IDF Score: 0.5802858236844359
Document: Is this the first document?, Word: the, TF-IDF Score: 0.38408524091481483
Document: Is this the first document?, Word: is, TF-IDF Score: 0.38408524091481483
Document: Is this the first document?, Word: this, TF-IDF Score: 0.38408524091481483

```

2.3 词嵌入

词嵌入(word embedding)是一种将词语映射到连续向量空间的技术,用于表示词语的语义和语法信息。它是自然语言处理(NLP)中的一项重要技术,对于推荐系统的构建和改进具有重要意义。在传统的基于计数的表示方法中,每个词语表示为一个独立的向量,无法捕捉到词语之间的语义关系。而词嵌入通过将词语映射到一个低维连续向量空间中,使得相似的词语在向量空间中的距离更小,因此能够更好地表示词语之间的语义相似性。



扫码看视频

2.3.1 分布式表示方法

分布式表示方法是一种将词语或文本表示为连续向量的技术,在推荐系统中被广泛应用于词嵌入和文本表示任务。它通过捕捉词语或文本的上下文信息来构建向量表示,使得具有相似语义或语法特征的词语及文本在向量空间中距离更近。

在 Python 中,有多种分布式表示方法可供使用,其中最常见的是 Word2Vec 和 GloVe。



1. Word2Vec

Word2Vec 是一种基于神经网络的词嵌入方法，它通过学习词语上下文的分布模式来生成词向量。Word2Vec 包括两种模型：连续词袋模型(continuous bag-of-words, CBOW)和跳字模型(Skip-gram)。CBOW 模型是根据上下文词语预测目标词语，Skip-gram 模型则是根据目标词语预测上下文词语。

2. GloVe

GloVe(global vectors for word representation)是一种基于全局统计信息的词嵌入方法，它利用词语的共现矩阵来捕捉词语之间的关系。GloVe 通过最小化损失函数来学习词语的向量表示，使得在向量空间中具有相似共现模式的词语距离更小。

与 Word2Vec 不同，GloVe 是基于全局词汇共现矩阵进行训练的，而不是仅依赖于局部上下文窗口。GloVe 的核心思想是：词与词之间的关系可以通过它们在上下文窗口中的共现频率来捕捉。具体来说，GloVe 会首先构建一个词汇共现矩阵，其中的每个元素表示两个词同时出现在上下文窗口中的频率。然后通过优化目标函数，将这些共现信息映射到低维的词嵌入空间。最终得到的词嵌入向量具有良好的语义表示能力，可以用于推荐系统中的相似度计算、文本分类等任务。

分布式表示方法可以应用于推荐系统中的多个任务，如文本分类、文本聚类、推荐算法中的特征表示等。通过将词语或文本转换为连续向量表示，可以更好地捕捉到语义和语法的特征，从而提高推荐系统的性能和准确性。

注意：词嵌入模型的训练需要大量的文本数据，并且模型的选择和参数调整也会对结果产生影响。因此，在应用词嵌入技术时，需要根据具体的任务和数据进行合适的模型选择和参数调整，以获得更好的效果。

2.3.2 使用 Word2Vec 模型

使用库 Gensim 可以方便地实现 Word2Vec 模型，再通过输入语料库进行训练，可以得到每个词语的分布式表示。下面的实例是假设现在有一个电影推荐系统，使用 Word2Vec 模型实现分布式表示方法并计算电影的相似度。

源码路径：daima/2/fenbu1.py

```
import pandas as pd
from gensim.models import Word2Vec

# 电影数据
```



```

movies_list = [
    "The Dark Knight",
    "Inception",
    "Interstellar",
    "The Shawshank Redemption",
    "Pulp Fiction",
    "Fight Club"
]

# 对电影标题进行分词
movies_tokens = [movie.lower().split() for movie in movies_list]

# 训练 Word2Vec 模型
model = Word2Vec(sentences=movies_tokens, vector_size=100, window=5, min_count=1)

# 获取电影的分布式表示
def get_movie_embedding(title):
    tokens = title.lower().split()
    embedding = []
    for token in tokens:
        if token in model.wv:
            embedding.append(model.wv[token])
    if embedding:
        return sum(embedding) / len(embedding)
    else:
        return None

# 选择一部电影，获取相似推荐
movie_title = 'The Dark Knight'
movie_embedding = get_movie_embedding(movie_title)
if movie_embedding is not None:
    similar_movies = model.wv.most_similar([movie_embedding], topn=5)
    similar_movie_titles = [movie[0] for movie in similar_movies]
    print(f"根据电影 '{movie_title}' 推荐的相似电影: ")
    print(similar_movie_titles)
else:
    print(f"找不到电影 '{movie_title}' 的分布式表示。")

```

在上述代码中，首先，导入所需的库，包括 `pandas` 和 `Word2Vec`。随后，创建一个包含电影标题的列表 `movies_list`。接着，对电影标题进行分词处理，生成一个包含分词结果的列表 `movies_tokens`。然后，使用 `Word2Vec` 模型进行训练，传入分词后的电影标题列表 `movies_tokens`，设置向量维度为 100，窗口大小为 5，最小词频为 1。最后，定义函数 `get_movie_embedding()`，用于获取电影的词嵌入表示。在该函数中，首先检查电影是否在训练集中存在，如果存在，就返回对应的词嵌入向量，否则返回空向量。

执行代码后会输出模型的训练进度和结果。而在调用函数 `get_movie_embedding()` 时，



会输出电影的词嵌入向量或空向量。例如：

```
根据电影 'The Dark Knight' 推荐的相似电影：
['the', 'knight', 'dark', 'interstellar', 'inception']
```

2.3.3 使用 GloVe 模型

在 Python 中可以使用库 Gensim 或者直接下载预训练的 GloVe 向量进行词嵌入操作。下面实例的功能是使用 GloVe 预训练模型(glove-wiki-gigaword-100)来计算电影标题的相似度。

源码路径：`daima/2/fenbu2.py`

```
from gensim.models import KeyedVectors

# 加载预训练的 GloVe 模型
glove_model = KeyedVectors.load_word2vec_format('glove-wiki-gigaword-100.txt',
binary=False)

# 定义电影标题列表
movies = ['The Dark Knight', 'Inception', 'Interstellar', 'The Matrix', 'Fight Club']

# 计算相似度矩阵
similarity_matrix = [[glove_model.wv.similarity(movie1, movie2) for movie2 in
movies] for movie1 in movies]

# 打印相似度矩阵
print("相似度矩阵: ")
for i in range(len(movies)):
    for j in range(len(movies)):
        print(f"{movies[i]} 与 {movies[j]} 的相似度: {similarity_matrix[i][j]}")
```

在本实例中，使用了 GloVe 官方网站中的数据集文件 glove-wiki-gigaword-100.txt，需要读者自己去下载。上述代码计算了电影标题之间的相似度矩阵，并打印了每对电影之间的相似度值。

2.4 主题模型

主题模型(topic model)是一种用于分析文本数据的统计模型，它旨在发现文本背后的潜在主题或话题结构。主题模型假设每个文档都由多个主题组成，并且每个主题都由一组相关的单词表示。通过分析文档中单词的分布模式，主题模型可以识别出这些主题，并用它们来描述和表示文本数据。在基于内容



扫码看视频

的推荐系统中，主题模型可以帮助理解文本数据中的主题信息，并将其应用于推荐过程中。

2.4.1 潜在语义分析

潜在语义分析(Latent Semantic Analysis, LSA)是一种主题模型方法，用于在文本数据中发现潜在的语义结构。LSA 基于矩阵分解技术，可以将文本数据转换为低维的语义空间表示。

LSA 的核心思想是通过奇异值分解(Singular Value Decomposition, SVD)来降低文本数据的维度，并捕捉文本之间的语义关系。下面是一个使用 LSA 实现主题模型的例子。

源码路径: daima/2/qian.py

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD

# 假设有一组文档数据
documents = [
    "I like to watch movies",
    "I prefer action movies",
    "Documentaries are informative",
    "I enjoy romantic movies",
    "Comedies make me laugh",
]

# 将文档数据向量化为 TF-IDF 矩阵
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(documents)

# 使用 LSA 进行主题建模
lsa = TruncatedSVD(n_components=2)
lsa.fit(X)

# 输出每个主题的关键词
feature_names = vectorizer.get_feature_names()
for topic_idx, topic in enumerate(lsa.components_):
    print(f"主题 {topic_idx+1}:")
    top_words = [feature_names[i] for i in topic.argsort()[::-1]]
    print(", ".join(top_words))
```

在上述代码中，首先，使用库 `scikit-learn` 中的类 `TfidfVectorizer` 将文档数据转换为 TF-IDF 矩阵，该矩阵反映了单词在文档中的重要性。然后，使用 `TruncatedSVD` 进行 LSA 主题建模，设置主题数为 2。最后，打印输出每个主题的关键词，以了解每个主题所代表的语义内容。执行代码后会输出：



主题 1:

movies, action, prefer, romantic, enjoy

主题 2:

informative, documentaries, are, enjoy, romantic

LSA 可以帮助我们在文本数据中发现主题和语义关系,从而应用于推荐系统中。例如,可以根据用户的偏好和文本数据的主题进行推荐,提供个性化的推荐结果。

注意: LSA 是一种无监督学习方法,它依赖于文本数据本身的特征。在实际应用中,可以结合其他特征和技术,如用户反馈、协同过滤等,以构建更加精确的推荐系统。

2.4.2 隐含狄利克雷分布

隐含狄利克雷分布(Latent Dirichlet Allocation, LDA)是一种概率主题模型,用于发现文本数据中的潜在主题和主题分布。LDA 假设每个文档包含多个主题,每个主题又由多个单词组成,然后通过统计方法推断文档的主题分布和单词的主题分布。例如,下面是一个使用 LDA 实现主题模型的例子。

源码路径: `daima/2/yinhan.py`

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation

# 假设有一组文档数据
documents = [
    "I like to watch movies",
    "I prefer action movies",
    "Documentaries are informative",
    "I enjoy romantic movies",
    "Comedies make me laugh",
]

# 将文档数据向量化为词频矩阵
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(documents)

# 使用 LDA 进行主题建模
lda = LatentDirichletAllocation(n_components=2, random_state=42)
lda.fit(X)

# 输出每个主题的关键词
feature_names = vectorizer.get_feature_names()
```



```
for topic_idx, topic in enumerate(lda.components_):
    print(f"主题 {topic_idx+1}:")
    top_words = [feature_names[i] for i in topic.argsort()[::-6:-1]]
    print(", ".join(top_words))
```

在上述代码中，首先，使用库 `scikit-learn` 中的类 `CountVectorizer` 将文档数据转换为词频矩阵，该矩阵反映了每个单词在文档中的出现次数。然后，使用 `LatentDirichletAllocation` 进行 LDA 主题建模，设置主题数为 2。最后，打印输出每个主题的关键词，以了解每个主题所代表的语义内容。执行代码后会输出：

```
主题 1:
are, informative, documentaries, enjoy, romantic
主题 2:
movies, me, make, comedies, laugh
```

LDA 可以帮助我们在文本数据中发现潜在的主题结构，从而应用于推荐系统中。例如，可以根据用户的兴趣和文本数据的主题分布进行推荐，提供个性化的推荐结果。

需要注意的是，LDA 是一种无监督学习方法，它基于概率模型进行推断，依赖于文本数据本身的特征。在实际应用中，可以结合其他特征和技术，如用户行为数据、协同过滤等，以构建更加精确的推荐系统。同时，还有其他主题模型方法可供选择，如潜在语义分析(LSA)也可以用于发现文本数据的主题结构。选择适合问题需求和数据特点的主题模型方法是推荐系统设计的重要考虑因素。

2.4.3 主题模型的应用

假设我们运营一个电商平台，现在希望通过主题模型来实现基于内容的商品推荐。我们可以使用 LDA 主题模型来分析商品的文本描述，从中发现商品的潜在主题，然后根据用户的偏好向其推荐相关主题的商品。下面的实例演示了使用 LDA 主题模型实现商品推荐的过程。

源码路径：`daima/2/product.py`

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation

# 假设有一组商品数据，每种商品有一个文本描述
products = [
    {"product_id": 1, "description": "High-performance gaming laptop with powerful graphics card."},
    {"product_id": 2, "description": "Wireless noise-canceling headphones for immersive audio experience."},
```



```
{ "product_id": 3, "description": "Smart home security camera with real-time monitoring." },
{ "product_id": 4, "description": "Compact and lightweight digital camera for travel photography." },
{ "product_id": 5, "description": "Stylish and durable backpack for everyday use." },
]

# 提取商品描述文本
documents = [product["description"] for product in products]

# 将商品描述向量化为词频矩阵
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(documents)

# 使用 LDA 进行主题建模
lda = LatentDirichletAllocation(n_components=3, random_state=42)
lda.fit(X)

# 对每种商品进行主题预测
for i, product in enumerate(products):
    description = product["description"]
    X_new = vectorizer.transform([description])
    topic_probabilities = lda.transform(X_new)
    topic_idx = topic_probabilities.argmax()
    product["topic"] = topic_idx

# 根据用户偏好推荐商品
user_preferences = [1, 2] # 假设用户偏好的主题是 1 和 2
recommended_products = [product for product in products if product["topic"] in user_preferences]

# 输出推荐的商品
print("推荐的商品: ")
for product in recommended_products:
    print(f"商品 ID: {product['product_id']}, 描述: {product['description']}")
```

在上述代码中，首先使用 `CountVectorizer` 将商品描述转换为词频矩阵，然后使用 `LatentDirichletAllocation` 进行 LDA 主题建模，设置主题数为 3。接下来，对每种商品进行主题预测，并将预测结果存储在商品数据中。最后，根据用户的偏好选择相应的主题，并推荐属于这些主题的商品。执行代码后会输出：

```
推荐的商品:
商品 ID: 2, 描述: Wireless noise-canceling headphones for immersive audio experience.
商品 ID: 4, 描述: Compact and lightweight digital camera for travel photography.
商品 ID: 5, 描述: Stylish and durable backpack for everyday use.
```

本实例展示了使用主题模型进行商品推荐的过程。通过分析商品描述的潜在主题，可以根据用户的偏好推荐与其兴趣相关的商品。这种基于内容的推荐方法可以帮助推荐系统提供个性化的商品推荐，增加用户的购买体验和满意度。

注意：这只是一个简单的示例，在实际应用中可能需要考虑更多的因素，如用户历史行为、商品属性等，以构建更准确和更有效的推荐系统。此外，还可以根据具体情况使用其他主题模型算法和技术，如潜在语义分析(LSA)和 BERT 等。

2.5 文本分类和标签提取

文本分类是指将文本数据按照预定义的类别或标签进行分类的任务。文本分类是自然语言处理(NLP)领域中的一个重要问题，具有广泛的应用，例如情感分析、垃圾邮件过滤、新闻分类等。在 Python 中，有多种方法可以用于文本分类和标签提取，其中常用的有 3 种：传统机器学习、卷积神经网络和循环神经网络。



扫码看视频

2.5.1 传统机器学习

在 Python 中，可以使用机器学习技术实现文本分类和标签提取。文本分类是将文本数据分为不同的预定义类别或标签的任务，而标签提取是从文本中提取关键标签或关键词的任务。接下来，将简要介绍两种实现文本分类和标签提取的机器学习方法。

1. 朴素贝叶斯分类器

朴素贝叶斯分类器(naive bayes classifier)是一种简单且有效的文本分类方法。它基于朴素贝叶斯定理和特征独立性假设，将文本特征与类别之间的条件概率进行建模。常见的朴素贝叶斯分类器包括多项式朴素贝叶斯(multinomial naive bayes)和伯努利朴素贝叶斯(bernoulli naive bayes)。下面是一个使用朴素贝叶斯分类器进行文本分类和标签提取的例子，功能是对电影评论信息进行文本分类。

源码路径：`daima/2/pusu.py`

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB

# 文本数据
texts = [
```




```
"This movie is great!",
"I loved the acting in this film.",
"The plot of this book is intriguing.",
"I didn't enjoy the music in this concert.",
]

# 对文本进行特征提取
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(texts)

# 标签数据
labels = ['Positive', 'Positive', 'Positive', 'Negative']

# 创建朴素贝叶斯分类器模型并进行训练
clf = MultinomialNB()
clf.fit(X, labels)

# 进行文本分类和标签提取
test_text = "The acting in this play was exceptional."
test_X = vectorizer.transform([test_text])
predicted_label = clf.predict(test_X)

print(f"文本: {test_text}")
print(f"预测标签: {predicted_label}")
```

在上述代码中，使用了库 `scikit-learn` 中的类 `CountVectorizer` 进行文本特征提取，并使用类 `MultinomialNB` 实现了朴素贝叶斯分类器。通过将训练好的模型应用于新的文本，可以进行文本分类和标签提取。执行代码后会输出：

```
文本: The acting in this play was exceptional.
预测标签: ['Positive']
```

2. 支持向量机

支持向量机(Support Vector Machine, SVM)是一种强大的文本分类算法，它可以通过构建高维特征空间并找到最佳的分割超平面来实现分类。SVM 在文本分类中的应用主要包括线性支持向量机(linear SVM)和核支持向量机(kernel SVM)。核函数可以帮助 SVM 处理非线性问题，如径向基函数核(radial basis function kernel)。下面是一个简单的实例，演示了使用支持向量机实现音乐推荐的文本分类的用法。实例中使用音乐的特征描述作为模型的输入，并将音乐的推荐标签作为目标变量进行训练。

源码路径: daima/2/xiang.py

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# 音乐数据
music_features = [
    "This song has a catchy melody and upbeat rhythm.",
    "The lyrics of this track are deep and thought-provoking.",
    "The vocals in this album are powerful and emotional.",
    "I don't like the repetitive beats in this song.",
]

# 推荐标签数据
recommendations = ['Pop', 'Indie', 'Rock', 'Electronic']

# 对音乐特征进行文本特征提取
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(music_features)

# 创建支持向量机分类器模型并进行训练
clf = SVC()
clf.fit(X, recommendations)

# 进行音乐推荐
test_music = "I love the electronic beats in this track."
test_X = vectorizer.transform([test_music])
predicted_recommendation = clf.predict(test_X)

print(f"音乐特征: {test_music}")
print(f"推荐标签: {predicted_recommendation}")
```

在上述代码中,使用了库 `scikit-learn` 中的类 `TfidfVectorizer` 来提取音乐特征的文本表示,然后使用 `SVC` 来构建支持向量机分类器模型,并进行音乐推荐的标签预测。用户可以根据实际情况调整训练数据和测试数据,并使用更复杂的特征提取方法和模型调参来提高预测的准确性。执行代码后会输出:

```
音乐特征: I love the electronic beats in this track.
推荐标签: ['Electronic']
```



2.5.2 卷积神经网络

卷积神经网络(Convolutional Neural Network, CNN)是一种在推荐系统中广泛应用的深度学习模型,它在图像处理任务上取得了巨大的成功,并且在自然语言处理领域也得到了广泛应用。CNN 在推荐系统中常用于文本分类、图像推荐和音乐推荐等任务,能够从输入数据中提取特征并进行高效的模式识别。

下面简要介绍 CNN 在推荐系统中的应用和一些关键概念。

- ❑ **卷积层(convolutional layer):** 卷积层是 CNN 的核心组成部分,它通过应用卷积操作来提取输入数据的局部特征。在文本分类任务中,卷积层可以识别关键词组合或短语,捕捉文本中的局部模式。
- ❑ **池化层(pooling layer):** 池化层用于降低卷积层输出的维度,并保留最重要的特征。常用的池化操作包括最大池化(max pooling)和平均池化(average pooling),它们可以减小数据,并提取最显著的特征。
- ❑ **全连接层(fully connected layer):** 全连接层用于将卷积层和池化层提取的特征映射到输出标签空间。在推荐系统中,全连接层可以将提取的特征与用户行为数据进行关联,实现个性化推荐。
- ❑ **嵌入层(embedding layer):** 在文本推荐中,嵌入层用于将离散的文本输入转换为连续的向量表示。它可以学习单词之间的语义关系,并捕捉文本中的语义信息。
- ❑ **激活函数(activation function):** 激活函数引入了非线性特性,使得 CNN 能够学习更复杂的模式和特征。常用的激活函数包括 ReLU、Sigmoid 和 Tanh。

下面将通过一个具体实例详细讲解使用卷积神经网络对花朵图像进行分类的过程。本实例将使用 `keras.Sequential` 模型创建图像分类器,并使用 `preprocessing.image_dataset_from_directory` 加载数据。

源码路径: `daima/2/cnn02.py`

1. 准备数据集

本实例使用包含大约 3700 张鲜花照片的数据集,数据集包含 5 个子目录,每个类别一个目录:

```
flower_photo/  
  daisy/  
  dandelion/  
  roses/  
  sunflowers/  
  tulips/
```


(1) 下载数据集，代码如下：

```
import pathlib
dataset_url =
"https://storage.googleapis.com/download.tensorflow.org/example_images/flower_
photos.tgz"
data_dir = tf.keras.utils.get_file('flower_photos', origin=dataset_url, untar=True)
data_dir = pathlib.Path(data_dir)
image_count = len(list(data_dir.glob('*/*.jpg')))
print(image_count)
```

执行代码后会输出：

```
3670
```

这说明在数据集中共有 3670 张图像。

(2) 浏览数据集 **roses** 目录中的第一张图像，代码如下：

```
roses = list(data_dir.glob('roses/*'))
PIL.Image.open(str(roses[0]))
```

执行代码后显示数据集 **roses** 目录中的第一张图像，如图 2-1 所示。



图 2-1 roses 目录中的第一张图像

(3) 也可以浏览数据集 **tulips** 目录中的第一张图像，代码如下：

```
tulips = list(data_dir.glob('tulips/*'))
PIL.Image.open(str(tulips[0]))
```

执行效果如图 2-2 所示。



图 2-2 tulips 目录中的第一张图像

2. 创建数据集

下面使用 `image_dataset_from_directory()` 方法从磁盘中加载数据集中的图像，然后从头开始编写自己的加载数据集的代码。

(1) 首先为加载器定义加载参数，代码如下：

```
batch_size = 32
img_height = 180
img_width = 180
```

(2) 在现实中通常使用验证拆分法创建神经网络模型，在本实例中将使用 80% 的图像进行训练，使用 20% 的图像进行验证。使用 80% 的图像进行训练的代码如下：

```
train_ds = tf.keras.preprocessing.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset="training",
    seed=123,
    image_size=(img_height, img_width),
    batch_size=batch_size)
```

执行代码后会输出：

```
Found 3670 files belonging to 5 classes.
Using 2936 files for training.
```

使用 20% 的图像进行验证的代码如下：

```
val_ds = tf.keras.preprocessing.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    subset="validation",
    seed=123,
    image_size=(img_height, img_width),
    batch_size=batch_size)
```

执行代码后会输出：

```
Found 3670 files belonging to 5 classes.
Using 734 files for validation.
```

可以在数据集的属性 `class_names` 中找到类名，每个类名和目录名称的字母顺序对应。例如执行下面的代码：

```
class_names = train_ds.class_names
print(class_names)
```

执行后会显示类名：

```
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

(3) 可视化数据集中的数据，通过如下代码显示训练数据集中的前 9 张图像：

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```

执行效果如图 2-3 所示。

(4) 接下来通过将这些数据集传递给训练模型 `model.fit`，手动迭代数据集并检索批量图像。代码如下：

```
for image_batch, labels_batch in train_ds:
    print(image_batch.shape)
    print(labels_batch.shape)
    break
```

执行代码后会输出：

```
(32, 180, 180, 3)
(32,)
```




通过上述输出可知, `image_batch` 是形状的张量(32, 180, 180, 3), 这是一批 32 张形状图像: 180×180×3(最后一个维度是指颜色通道 RGB); 而 `labels_batch` 是形状的张量(32,)。



图 2-3 训练数据集中的前 9 张图像

3. 配置数据集

(1) 接下来将配置数据集以提高性能, 确保本实例使用缓冲技术从磁盘生成数据, 而不会导致 I/O 阻塞。下面是加载数据时建议使用的两种重要方法。

- ❑ `Dataset.cache()`: 当从磁盘加载图像后, 将图像保存在内存中。这将确保数据集在训练模型时不会成为瓶颈。当数据集太大而无法放入内存时, 可以使用此方法来创建高性能的磁盘缓存。
- ❑ `Dataset.prefetch()`: 在训练过程中, 能够使数据预处理操作和模型执行操作重叠进行, 从而提高训练效率。

(2) 然后进行数据标准化处理。因为 RGB 通道值在[0, 255]范围内，这对于神经网络来说并不理想。一般来说，应该设法使输入值变小。在本实例中将重新缩放图层，将值标准化为[0, 1]范围内，代码如下：

```
normalization_layer = layers.experimental.preprocessing.Rescaling(1./255)
```

(3) 通过调用 `map()` 方法将该层应用于数据集，代码如下：

```
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
print(np.min(first_image), np.max(first_image))
```

执行代码后会输出：

```
0.0 0.9997713
```

或者，可以在模型定义中包含该层，这样可以简化部署。本实例将使用第二种方法。

4. 创建模型

本实例的模型由三个卷积块组成，每个块都有一个最大池层。模型中有一个全连接层，上面有 128 个单元，由激活函数激活。该模型尚未针对高精度进行调整，本实例的目标是展示一种标准方法。代码如下：

```
num_classes = 5

model = Sequential([
    layers.experimental.preprocessing.Rescaling(1./255, input_shape=(img_height,
img_width, 3)),
    layers.Conv2D(16, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(32, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes)
])
```

5. 编译模型

(1) 本实例中使用了 `optimizers.Adam` 优化器和 `losses.SparseCategoricalCrossentropy()` 损失函数。要想查看每个训练时期的训练准确率和验证准确率，需要传递 `metrics` 参数。代码如下：



```
model.compile(optimizer='adam',  
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
              metrics=['accuracy'])
```

(2) 使用模型的函数 `summary()` 查看网络中的所有层，代码如下：

```
model.summary()
```

6. 训练模型

开始训练模型，代码如下：

```
epochs=10  
history = model.fit(  
    train_ds,  
    validation_data=val_ds,  
    epochs=epochs  
)
```

执行代码后会输出：

```
Epoch 1/10  
92/92 [=====]  
///省略部分结果  
Epoch 10/10  
92/92 [=====] - 1s 10ms/step - loss: 0.0566 - accuracy: 0.9847 -
```

7. 可视化训练结果

在训练集和验证集上创建损失图和准确度图，然后绘制可视化结果，代码如下：

```
acc = history.history['accuracy']  
val_acc = history.history['val_accuracy']  
  
loss = history.history['loss']  
val_loss = history.history['val_loss']  
  
epochs_range = range(epochs)  
  
plt.figure(figsize=(8, 8))  
plt.subplot(1, 2, 1)  
plt.plot(epochs_range, acc, label='Training Accuracy')  
plt.plot(epochs_range, val_acc, label='Validation Accuracy')  
plt.legend(loc='lower right')  
plt.title('Training and Validation Accuracy')  
  
plt.subplot(1, 2, 2)
```



```
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

执行代码后的效果如图 2-4 所示。

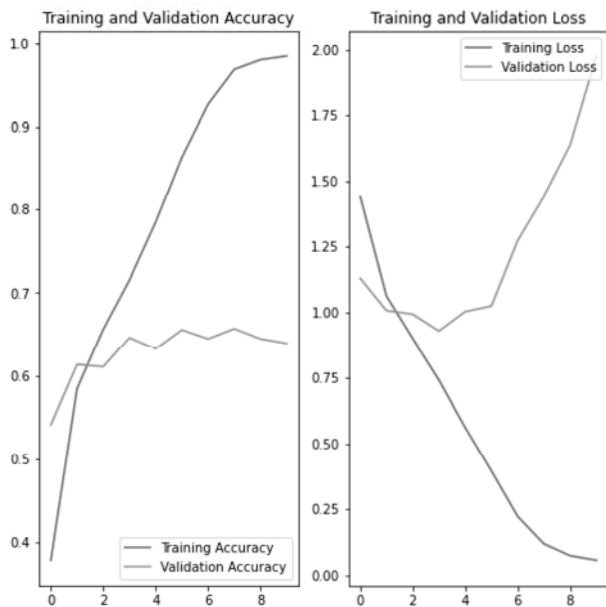


图 2-4 可视化损失图和准确度图

8. 过拟合处理：数据增强

从可视化损失图和准确度图中可以看出，训练准确率和验证准确率相差很大，模型在验证集上的准确率只有 60% 左右。训练准确率会随着时间线性增加，而验证准确率在训练过程中只停滞在 60% 左右。此外，训练准确率和验证准确率之间的差异是显而易见的，这是过度拟合的迹象。

当训练样例数量较少时，模型有时会从训练样例的噪声或不需要的细节中学习，这在一定程度上会对模型在新样例上的性能产生负面影响，这种现象称为过拟合。它意味着该模型将很难在新数据集上泛化。在训练过程中有多种方法可以对抗过拟合。

过拟合通常发生在训练样本较少时，数据增强采用的方法是从现有示例中生成额外的训练数据，方法是使用随机变换来增强它们，从而产生看起来可信的图像，使模型能够看到数据的不同变体，从而提高其泛化能力。



(1) 通过使用 `tf.keras.layers.experimental.preprocessing` 实现数据增强，可以将其像其他层一样包含在模型中，并在 GPU 上运行。代码如下：

```
data_augmentation = keras.Sequential([
    layers.experimental.preprocessing.RandomFlip("horizontal",
                                                    input_shape=(img_height,
                                                                    img_width,
                                                                    3)),
    layers.experimental.preprocessing.RandomRotation(0.1),
    layers.experimental.preprocessing.RandomZoom(0.1),
])
```

(2) 对同一张图像多次应用数据增强技术，下面是可视化数据增强的代码：

```
plt.figure(figsize=(10, 10))
for images, _ in train_ds.take(1):
    for i in range(9):
        augmented_images = data_augmentation(images)
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(augmented_images[0].numpy().astype("uint8"))
        plt.axis("off")
```

执行代码后的效果如图 2-5 所示。

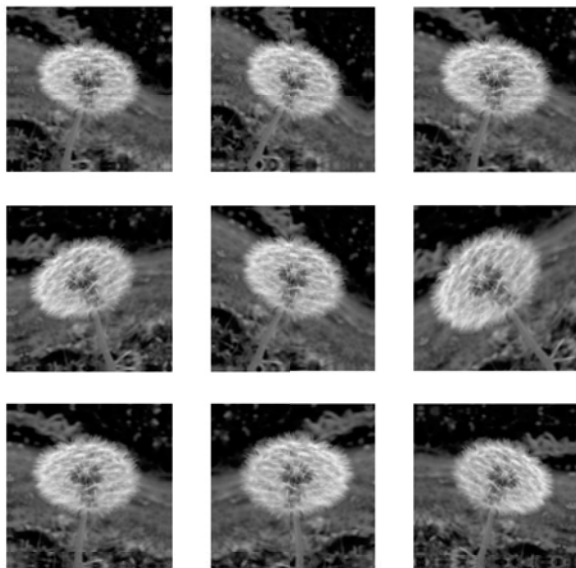


图 2-5 数据增强效果

9. 过拟合处理：将 Dropout 引入网络

接下来介绍另一种减少过拟合的技术：将 Dropout 引入网络，这是一种正则化处理形式。当将 Dropout 应用于一个层时，它会在训练过程中从该层随机删除(通过将激活设置为 0)许多输出单元。Dropout 会将一个小数作为其输入值，例如 0.1、0.2、0.4 等，这意味着从应用层中随机丢弃 10%、20% 或 40% 的输出单元。下面的代码首先创建了一个新的神经网络 layers.Dropout，然后使用增强图像对其进行训练。

```
model = Sequential([
    data_augmentation,
    layers.experimental.preprocessing.Rescaling(1./255),
    layers.Conv2D(16, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(32, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Dropout(0.2),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes)
])
```

10. 重新编译和训练模型

经过前面的过拟合处理后，接下来重新编译和训练模型。重新编译模型的代码如下：

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
model.summary()
Model: "sequential_2"
```

重新训练模型的代码如下：

```
epochs = 15
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs
)
```

执行代码后会输出：

```
Epoch 1/15
92/92 [=====] - 2s 13ms/step - loss: 1.2685 - accuracy:
0.4465 - val_loss: 1.0464 - val_accuracy: 0.5899
```




```
///省略部分代码
Epoch 15/15
92/92 [=====] - 1s 11ms/step - loss: 0.4930 - accuracy:
0.8096 - val_loss: 0.6705 - val_accuracy: 0.7384
```

在使用数据增强和 Dropout 处理后,过拟合比以前少了,训练准确率和验证准确率更接近。接下来重新可视化训练结果,代码如下:

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

执行代码后的效果如图 2-6 所示。

11. 预测新数据

最后使用最新创建的模型对未包含在训练集或验证集中的图像进行分类处理,代码如下:

```
sunflower_url = "https://storage.googleapis.com/download.tensorflow.org/
example_images/592px-Red_sunflower.jpg"
sunflower_path = tf.keras.utils.get_file('Red_sunflower', origin=sunflower_url)

img = keras.preprocessing.image.load_img(
    sunflower_path, target_size=(img_height, img_width)
)
img_array = keras.preprocessing.image.img_to_array(img)
img_array = tf.expand_dims(img_array, 0) # Create a batch

predictions = model.predict(img_array)
```

```
score = tf.nn.softmax(predictions[0])

print(
    "This image most likely belongs to {} with a {:.2f} percent confidence."
    .format(class_names[np.argmax(score)], 100 * np.max(score))
)
```

执行代码后会输出：

```
Downloading data from
https://storage.googleapis.com/download.tensorflow.org/example_images/592px-Red_sunflower.jpg
122880/117948 [=====] - 0s 0us/step
This image most likely belongs to sunflowers with a 99.36 percent confidence.
```



图 2-6 可视化结果

需要注意的是，数据增强和 Dropout 层在推理时处于非活动状态。

2.5.3 循环神经网络

循环神经网络(Recurrent Neural Network, RNN)是一种常用于处理序列数据的神经网络模型。在推荐系统中，RNN 被广泛应用于序列建模和推荐任务，例如用户行为序列分析、时间序列数据预测、文本生成等。



RNN 的特点是能够处理具有时间依赖性的数据，通过记忆过去的信息来影响当前的输出。与传统的前馈神经网络不同，RNN 引入了循环连接，使得信息可以在网络内部进行传递和更新。这种循环连接的设计使得 RNN 在处理序列数据时具有优势。

在 Python 中，可以使用多种库和框架来构建与训练 RNN 模型，其中最常用的是 TensorFlow 和 PyTorch。这些工具提供了丰富的 RNN 实现，包括常用的 RNN 变体[如长期记忆网络(LSTM)和门控循环单元(GRU)]，以及各种辅助函数和工具，方便进行模型构建、训练和评估。实例文件 `xun.py` 的功能是使用循环神经网络(LSTM)实现文本分类。

源码路径: `daima/2/xun.py`

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

# 自定义数据集类
class SentimentDataset(Dataset):
    def __init__(self, texts, labels):
        self.texts = texts
        self.labels = labels

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]
        return text, label

# 自定义循环神经网络模型
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        embedded = self.embedding(x)
        output, _ = self.lstm(embedded)
        output = self.fc(output[:, -1, :]) # 取最后一个时刻的输出
        return output
```



```

# 准备数据
texts = ["I love this movie", "This film is terrible", "The acting was superb"]
labels = [1, 0, 1] # 1 代表正面情感, 0 代表负面情感

# 构建词汇表
vocab = set(' '.join(texts))
char_to_idx = {ch: i for i, ch in enumerate(vocab)}

# 创建数据集和数据加载器
dataset = SentimentDataset(texts, labels)
data_loader = DataLoader(dataset, batch_size=1, shuffle=True)

# 定义超参数
input_size = len(vocab)
hidden_size = 128
output_size = 2 # 正面和负面两种情感
num_epochs = 10

# 实例化模型
model = LSTMModel(input_size, hidden_size, output_size)

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

# 训练模型
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
criterion.to(device)

for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0
    for inputs, labels in data_loader:
        inputs = [char_to_idx[ch] for ch in inputs[0]]
        inputs = torch.tensor(inputs).unsqueeze(0).to(device)
        labels = torch.tensor(labels).to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss/len(data_loader):.4f}")

```



在上述代码中，首先定义了一个数据集类 `SentimentDataset` 来处理情感分类的文本数据。然后定义了一个简单的 LSTM 模型 `LSTMModel`，其中包含一个嵌入层、一个 LSTM 层和一个全连接层。接着使用自定义数据集类加载样本文本和相应的标签，并根据需要将文本转换为整数索引序列。再使用数据加载器迭代数据，并在每个批次上训练模型。在训练过程中，迭代数据加载器，将每个样本的输入文本转换为整数索引序列，并将其作为输入传递给模型进行训练。最后使用交叉熵损失函数计算损失，并使用反向传播和优化器更新模型的参数。执行代码后会输出：

```
Epoch 1/10, Loss: 0.7174
Epoch 2/10, Loss: 0.5884
Epoch 3/10, Loss: 0.5051
Epoch 4/10, Loss: 0.4218
Epoch 5/10, Loss: 0.3467
Epoch 6/10, Loss: 0.2571
Epoch 7/10, Loss: 0.1835
Epoch 8/10, Loss: 0.1147
Epoch 9/10, Loss: 0.0616
Epoch 10/10, Loss: 0.0392
```

注意：可以根据实际数据和需求对代码进行适当的修改，包括修改数据集类，调整模型结构，修改超参数等。

2.6 文本情感分析

文本情感分析是一种将自然语言文本的情感倾向性进行分类或评估的技术，它可以帮助我们了解文本中所表达的情感，例如积极、消极或中性，从而在推荐系统中更好地理解用户的喜好和情感偏好。在 Python 中，有多种方法可以进行文本情感分析，其中常用的方法有两种：机器学习方法和深度学习方法。



扫码看视频

2.6.1 机器学习方法

机器学习模型能够通过训练数据学习文本的特征表示，并通过对新的文本数据进行预测来判断情感类别。使用机器学习方法实现文本情感分析的基本流程如下。

(1) 准备数据集，包括带有标签的文本样本，例如电影评论数据集，其中每个样本都有一个情感标签(积极或消极)。可以使用公开可用的数据集，如 IMDB 电影评论数据集。

(2) 对文本数据进行预处理,包括文本分词、移除停用词、词干化等操作。这可以通过使用自然语言处理库(如 NLTK、spaCy)来完成。

(3) 选择合适的特征表示方法。常用的特征表示方法包括词袋模型、TF-IDF 等。词袋模型是将文本表示为词汇表中单词的计数向量,而 TF-IDF 考虑了单词的频率和在整个文本集合中的重要性。

(4) 使用机器学习算法构建分类模型。常用的机器学习算法包括朴素贝叶斯(naive bayes)、支持向量机(SVM)、决策树(decision tree)等。这些算法可以通过使用机器学习框架(如 scikit-learn)进行构建和训练。

对于使用词袋模型表示的文本数据,可以将每个文本样本表示为特征向量,其中每个维度表示一个单词在文本中出现的次数。对于使用 TF-IDF 表示的文本数据,可以将每个文本样本表示为特征向量,其中每个维度表示一个单词的 TF-IDF 值。

(5) 对模型进行训练和优化。可以使用训练集进行模型的训练,通过调整模型的参数和使用交叉验证等技术来优化模型的性能。

(6) 使用测试集来评估模型的性能,包括准确率、精确率、召回率等指标。可以使用混淆矩阵来可视化模型的分类结果。

(7) 使用训练好的模型对新的文本数据进行情感分析。即将新的文本转换为特征向量,并通过模型的预测输出来判断文本的情感类别。

总结起来,使用机器学习方法实现文本情感分析需要准备数据集,进行数据预处理,选择特征表示方法,构建和训练模型,最后对新数据进行预测。这样的方法可以用于自动分析和理解大量文本数据中的情感倾向,为情感分析任务提供了一种可行的解决方案。

下面是一个使用机器学习方法实现商品情感分析的例子,其中涉及数据准备,文本预处理和机器学习模型训练等步骤。

源码路径: daima/2/jiqi.py

```
# 读取训练数据集
data = pd.read_csv('reviews.csv')

# 划分训练集和测试集
train_data, test_data, train_labels, test_labels = train_test_split(data['review'],
data['sentiment'], test_size=0.2, random_state=42)

# 文本向量化
vectorizer = TfidfVectorizer()
train_vectors = vectorizer.fit_transform(train_data)
test_vectors = vectorizer.transform(test_data)

# 构建支持向量机模型
```




```
svm = SVC()
svm.fit(train_vectors, train_labels)

# 在测试集上进行预测
predictions = svm.predict(test_vectors)

# 评估模型性能
accuracy = svm.score(test_vectors, test_labels)
print("Accuracy:", accuracy)
```

在上述代码中，首先，读取包含评论和情感标签的训练数据集(例如 `reviews.csv`)。随后，使用 `train_test_split()` 函数将数据集划分为训练集和测试集。接下来，使用 `TfidfVectorizer` 将文本数据转换为 TF-IDF 特征向量，这是一种常用的文本向量化方法。然后，使用支持向量机(SVM)作为分类器训练模型并对测试集进行预测。最后，计算模型在测试集上的准确率并将其作为评估指标，以衡量模型的性能。执行代码后会输出：

```
Accuracy: 1.0
```

注意：数据集的质量和规模对模型的性能有很大影响，在本实例中的数据集文件 `reviews.csv` 中只提供了很少的数据，建议读者进一步收集充足的数据并进行数据预处理，以提高模型的准确率和泛化能力。

2.6.2 深度学习方法

使用深度学习方法实现文本情感分析是一种常见且有效的技术。深度学习模型能够自动学习文本中的特征表示，并通过大量的训练数据来提高模型的性能。在文本情感分析中，常用的深度学习模型包括卷积神经网络(CNN)、循环神经网络(RNN)和长短期记忆网络(LSTM)等。这些模型可以通过使用深度学习框架(如 TensorFlow、Keras、PyTorch)进行构建和训练，具体说明如下。

- ❑ 对于 CNN 模型，可以使用卷积层来提取文本中的局部特征，然后通过池化层进行下采样，最后连接全连接层进行分类。
- ❑ 对于 RNN 或 LSTM 模型，可以利用序列数据的时间依赖性来捕捉文本中的上下文信息。可以将文本序列作为输入，经过嵌入层将单词转换为向量表示，然后通过 RNN 或 LSTM 层进行序列建模，最后连接全连接层进行分类。

在构建模型后，需要进行模型的训练和优化。可以使用训练集进行模型的训练，通过反向传播算法和优化算法(如随机梯度下降)来更新模型的参数，使得模型能够更好地拟合数据。在模型训练完成后，可以使用测试集来评估模型的性能，包括准确率、精确率、召回

率等指标。可以使用混淆矩阵来可视化模型的分类结果。最后，可以使用训练好的模型对新的文本数据进行情感分析。将新的文本输入到模型中，通过模型的预测输出来判断文本的情感类别。

实例文件 `film.py` 的功能是在 IMDB 大型电影评论数据集上训练循环神经网络，以进行文本情感分析。本实例使用 LSTM 模型在 IMDB 数据集上进行情感分析，首先会进行训练和评估工作，以获取模型的损失和准确率，并可以在新数据上进行情感预测。实例文件 `film.py` 的具体实现流程如下。

源码路径： `daima/2/film.py`

(1) 导入必要的库，包括 PyTorch 库(`torch` 和相关模块)、PyTorch 文本库(`torchtext`)以及 NumPy 库(`numpy`)。

(2) 定义 LSTM 模型。这是一个继承自 `nn.Module` 的子类，在模型的初始化方法 `__init__()` 中定义模型的各个层和参数，其中包括嵌入层(`nn.Embedding`)、LSTM 层(`nn.LSTM`)、全连接层(`nn.Linear`)以及 dropout 层(`nn.Dropout`)。在模型的前向传播方法 `forward()` 中，定义数据在模型中的流动路径：首先将输入文本通过嵌入层进行嵌入，然后将嵌入向量输入到 LSTM 层中，获取最后一个时间步的隐藏状态(`hidden[-1, :, :]`)并进行 dropout 操作，最后通过全连接层得到输出结果。对应的实现代码如下：

```
# 定义模型
class LSTMModel(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab_size, output_dim,
                 num_layers, bidirectional, dropout):
        super(LSTMModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=num_layers,
                             bidirectional=bidirectional, dropout=dropout)
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim,
                             output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, text):
        embedded = self.dropout(self.embedding(text))
        output, (hidden, _) = self.lstm(embedded)
        hidden = self.dropout(torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1))
        return self.fc(hidden.squeeze(0))
```

(3) 设置随机种子，以保证实验的可复现性。通过设定一个固定的随机种子，可以确保每次运行代码得到的随机数序列相同，从而使得实验结果具有可重复性。对应的实现代码如下：



```
# 设置随机种子
SEED = 1234
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

(4) 定义超参数, 包括嵌入维度(EMBEDDING_DIM)、隐藏层维度(HIDDEN_DIM)、输出维度(OUTPUT_DIM)、LSTM 层数(NUM_LAYERS)、是否双向 LSTM(BIDIRECTIONAL)、dropout 率(DROPOUT)和批量大小(BATCH_SIZE)等。对应的实现代码如下:

```
# 定义超参数
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
NUM_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.5
BATCH_SIZE = 64
```

(5) 使用 torchtext 库加载 IMDB 数据集。其中, Field()用于定义文本数据的预处理方式, 包括分词方法和是否将文本转换为小写; LabelField()用于定义标签数据的处理方式; IMDB.splits()用于将数据集划分为训练集和测试集。代码如下:

```
# 加载 IMDB 数据集
TEXT = Field(tokenize='spacy', lower=True)
LABEL = LabelField(dtype=torch.float)
train_data, test_data = IMDB.splits(TEXT, LABEL)
```

(6) 构建词汇表(vocabulary)。通过调用 build_vocab()方法并传入训练集数据, 可以构建词汇表。此外, 通过指定 vectors 参数为 “glove.6B.100d”, 可以加载预训练的词向量(glove.6B.100d)并将其应用于嵌入层。对应的实现代码如下:

```
# 构建词汇表
TEXT.build_vocab(train_data, vectors="glove.6B.100d")
LABEL.build_vocab(train_data)
```

(7) 创建数据加载器(data iterator)。通过调用 BucketIterator.splits()方法, 可以将训练集和测试集的数据打包成数据加载器, 用于后续的模式训练和评估; 其中指定了批量大小(batch_size)和设备(device)。对应的实现代码如下:

```
# 创建数据加载器
train_iterator, test_iterator = BucketIterator.splits(
    (train_data, test_data),
    batch_size=BATCH_SIZE,
    device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')
)
```


(8) 初始化 LSTM 模型。首先, 根据词汇表的大小(`len(TEXT.vocab)`)确定模型中嵌入层的输入维度。然后, 使用超参数和词汇表的大小创建一个 LSTM 模型实例。对应的实现代码如下:

```
# 初始化模型
vocab_size = len(TEXT.vocab)
model = LSTMModel(EMBEDDING_DIM, HIDDEN_DIM, vocab_size, OUTPUT_DIM, NUM_LAYERS,
                  BIDIRECTIONAL, DROPOUT)
```

(9) 加载预训练的词向量, 并将其赋值给嵌入层的权重。通过 `TEXT.vocab.vectors` 可以获取到词向量。对应的实现代码如下:

```
# 加载预训练的词向量
pretrained_embeddings = TEXT.vocab.vectors
model.embedding.weight.data.copy_(pretrained_embeddings)
```

(10) 定义损失函数(`nn.BCEWithLogitsLoss()`)和优化器(`optim.Adam`)。`nn.BCEWithLogitsLoss()`是用于二分类问题的损失函数, 它结合了 Sigmoid 激活函数和二元交叉熵损失。`optim.Adam`是一种常用的优化器, 用于参数的优化。对应的实现代码如下:

```
# 定义损失函数和优化器
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters())
```

(11) 将模型和损失函数移到 GPU 上进行计算(如果可用), 其中通过 `torch.cuda.is_available()`判断是否有可用的 GPU 设备。对应的实现代码如下:

```
# 将模型移到 GPU (如果可用)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
criterion = criterion.to(device)
```

(12) 定义模型的训练函数。训练函数接收模型(`model`)、数据加载器(`iterator`)、优化器(`optimizer`)和损失函数(`criterion`)作为输入。在函数内部, 首先将模型设为训练模式(`model.train()`), 然后遍历数据加载器中的每个批次数据。在每个批次中, 首先将优化器的梯度置 0(`optimizer.zero_grad()`), 然后获取批次数据的文本(`batch.text`)和标签(`batch.label`)。再通过模型预测文本的情感得分(`predictions`), 将其压缩为一维张量(`squeeze(1)`)。接着计算预测值与真实标签之间的损失(`loss`)和准确率(`acc`), 通过反向传播和优化器更新模型参数(`loss.backward()`和`optimizer.step()`)。最后, 累积损失和准确率到 `epoch_loss` 和 `epoch_acc` 中, 返回平均损失和平均准确率。对应的实现代码如下:

```
# 训练模型
def train(model, iterator, optimizer, criterion):
```



```
model.train()
epoch_loss = 0
epoch_acc = 0

for batch in iterator:
    optimizer.zero_grad()
    text = batch.text
    predictions = model(text).squeeze(1)
    loss = criterion(predictions, batch.label)
    acc = binary_accuracy(predictions, batch.label)
    loss.backward()
    optimizer.step()
    epoch_loss += loss.item()
    epoch_acc += acc.item()

return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

(13) 定义模型的评估函数。评估函数与训练函数的结构类似，唯一的区别在于模型设为评估模式(`model.eval()`)并使用 `torch.no_grad()` 上下文管理器来禁用梯度计算。这是因为在评估过程中不计算梯度，可以加快运算速度并减少内存消耗。模型评估函数返回平均损失和平均准确率。对应的实现代码如下：

```
# 评估模型
def evaluate(model, iterator, criterion):
    model.eval()
    epoch_loss = 0
    epoch_acc = 0

    with torch.no_grad():
        for batch in iterator:
            text = batch.text
            predictions = model(text).squeeze(1)
            loss = criterion(predictions, batch.label)
            acc = binary_accuracy(predictions, batch.label)
            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

(14) 定义计算准确率函数。给定模型的预测值(`preds`)和真实标签(`y`)，函数首先通过 `Sigmoid()` 函数将预测值映射为 0~1 的概率，并对其进行四舍五入。然后将四舍五入后的预测值与真实标签进行比较，计算正确预测的个数。再除以总样本数，得到准确率。对应的实现代码如下：

```
# 计算准确率
def binary_accuracy(preds, y):
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float()
    acc = correct.sum() / len(correct)
    return acc
```

(15) 训练模型。在每个训练周期(epoch)中, 首先调用训练函数(train())对模型进行训练, 并获取训练损失和准确率。然后调用评估函数(evaluate())对模型进行评估, 并获取验证损失和准确率。如果当前的验证损失(valid_loss)比之前记录的最佳验证损失(best_valid_loss)更小, 就将当前模型保存为最佳模型(model.pt)。最后, 打印每个训练周期的训练损失、训练准确率、验证损失和验证准确率。对应的实现代码如下:

```
# 开始训练
N_EPOCHS = 5
best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, test_iterator, criterion)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'model.pt')

    print(f'Epoch: {epoch+1:02} ')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc:.2%}')
    print(f'\t Val. Loss: {valid_loss:.3f} | Val. Acc: {valid_acc:.2%}')
```

(16) 加载之前保存的最佳模型参数(model.pt), 以便后续在新数据上进行预测。对应的实现代码如下:

```
# 加载保存的最佳模型
model.load_state_dict(torch.load('model.pt'))
```

(17) 在新数据上进行情感预测。编写函数 predict_sentiment(), 在新数据上进行情感预测。该函数接收模型(model)和待预测的句子(sentence)作为输入。在函数内部, 首先将模型设为评估模式(model.eval())。然后对句子进行分词, 并将分词后的单词转换为对应的索引。接着将索引转换为 PyTorch 张量, 并将其移动到相同的设备(GPU 或 CPU)上。为了与模型的输入形状匹配, 需要对张量进行维度调整(unsqueeze(1))。然后通过模型进行预测, 将输出的概率值通过 Sigmoid 函数进行映射, 得到情感预测值(范围为 0~1)。最后, 返回预测值(prediction.item())。在测试部分, 给定一个测试句子(test_sentence), 调用 predict_sentiment() 函数进行情感预测, 并将结果打印出来。对应的实现代码如下:



```
# 在新数据上进行预测
def predict_sentiment(model, sentence):
    model.eval()
    tokenized = [tok.text for tok in spacy_en.tokenizer(sentence)]
    indexed = [TEXT.vocab.stoi[t] for t in tokenized]
    tensor = torch.LongTensor(indexed).to(device)
    tensor = tensor.unsqueeze(1)
    prediction = torch.sigmoid(model(tensor))
    return prediction.item()

# 测试模型
test_sentence = "This movie is terrible!"
prediction = predict_sentiment(model, test_sentence)
print(f'Test Sentence: {test_sentence}')
print(f'Predicted Sentiment: {prediction:.4f}')
```