第 3 章 使用数据集工具

3.1 数据集工具介绍

在以往的自然语言处理任务中会花费大量的时间在数据处理上，针对不同的数据集往往需要不同的处理过程，各个数据集的格式差异大，处理起来复杂又容易出错。针对以上问题，HuggingFace 提供了统一的数据集处理工具，让开发者在处理各种不同的数据集时可以通过统一的 API 处理，大大降低了数据处理的工作量。

登录 HuggingFace 官网，单击顶部的 Datasets，即可看到 HuggingFace 提供的数据集，如图 3-1 所示。

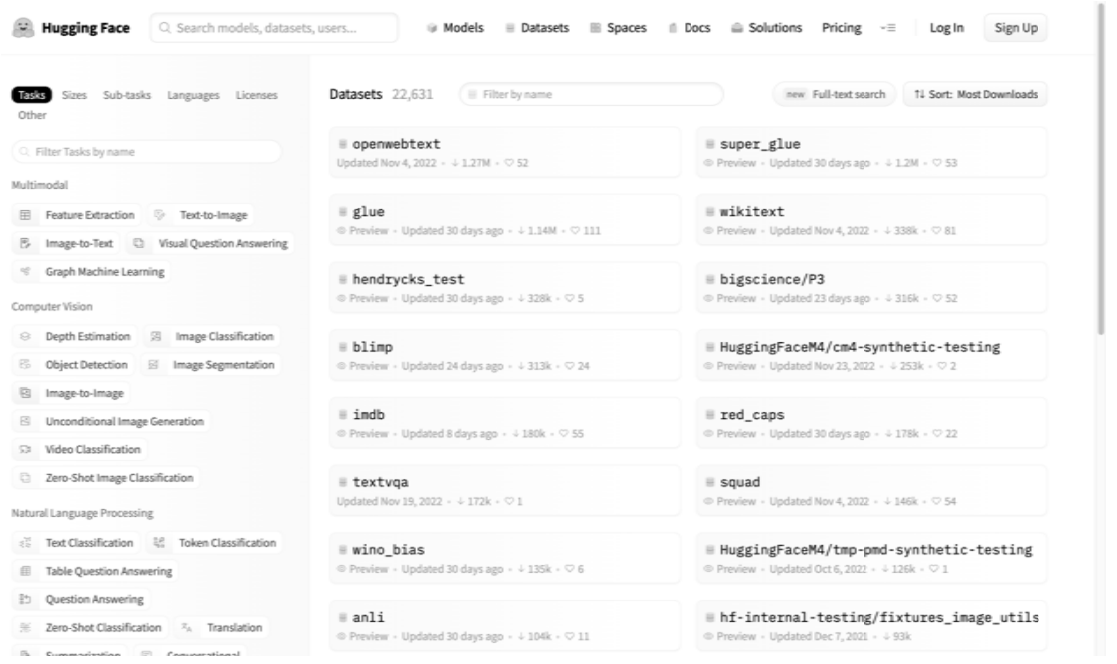

图 3-1　HuggingFace 数据集页面

在该界面左侧可以根据不同的任务类型、语言、体积、使用许可来筛选数据集，右侧为

具体的数据集列表，其中有经典的 glue、super_glue 数据集，问答数据集 squad，情感分类数据集 imdb，纯文本数据集 wikitext。

单击具体的某个数据集，进入数据集的详情页面，可以看到数据集的概要信息。以 glue 数据集为例，在详情页可以看到 glue 的各个数据子集的概要内容，每个数据子集的下方可能会有作者写的说明信息，如图 3-2 所示。

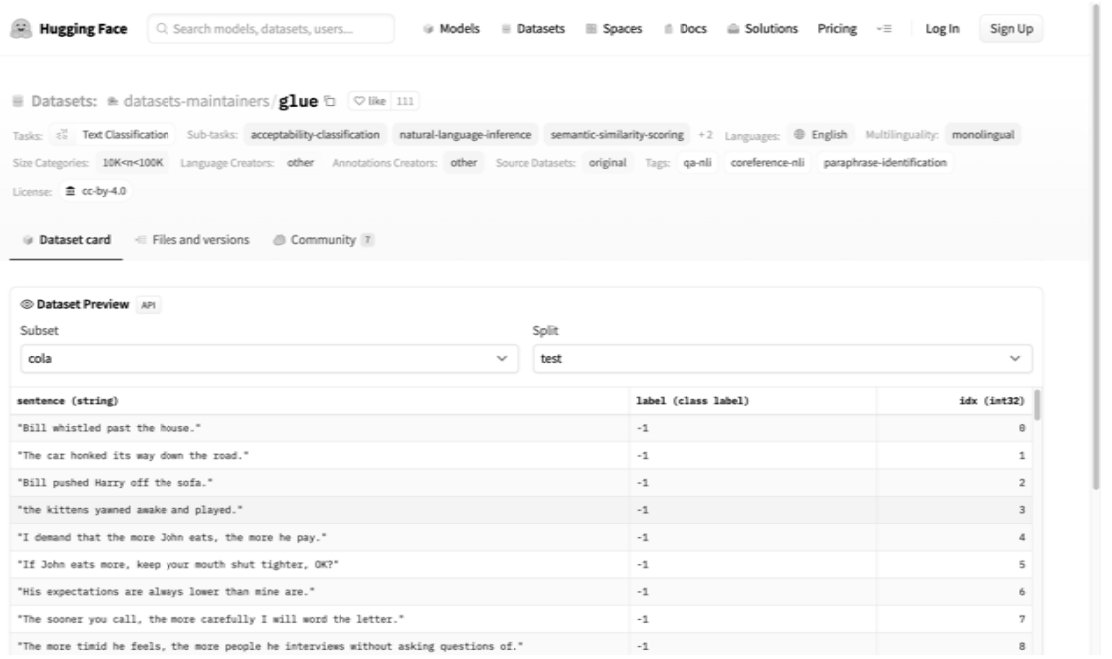

图 3-2　数据集详情页面

不要担心，你不需要熟悉所有的数据集，这些数据集大多是英文的，本书重点关注中文的数据集。出于简单起见，本书只会使用几个简单的数据集来完成后续的实战任务，具体可参看接下来的代码演示。

3.2　使用数据集工具

3.2.1　数据集加载和保存

1. 在线加载数据集

使用 HuggingFace 数据集工具加载数据往往只需一行代码，以加载名为 seamew/ChnSentiCorp 数据集为例，代码如下：

```
#第 3 章/加载数据集
from datasets import load_dataset
```

```
dataset = load_dataset(path='seamew/ChnSentiCorp')
dataset
```

注意：由于 HuggingFace 把数据集存储在谷歌云盘上，在国内加载时可能会遇到网络问题，所以本书的配套资源中已经提供了保存好的数据文件，使用 load_from_disk()函数加载即可。关于 load_from_disk()函数可参见本章“从本地磁盘加载数据集”一节。

可以看到，要加载一个数据集是很简单的，使用 load_dataset()函数，把数据集的名字作为参数传入即可。运行结果如下：

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 9600
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 0
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 1200
    })
})
```

可以看到 seamew/ChnSentiCorp 共分为 3 部分，分别为 train、validation 和 test，分别代表训练集、验证集和测试集，并且每条数据有两个字段，即 text 和 label，分别代表文本和标签。

还可以看到 3 部分分别的数据量，其中验证集的数据量为 0 条，说明虽然作者切分出了验证集这部分，但并没有向其中分配数据，这是一个空的部分。

加载数据集的 load_dataset()函数还有一些其他参数，通过下面这个例子说明，代码如下：

```
#第 3 章/加载 glue 数据集
load_dataset(path='glue', name='sst2', split='train')
```

这里加载了经典的 glue 数据集，熟悉 glue 的读者可能已经知道 glue 分了很多数据子集，可以以参数 name 指定要加载的数据子集，在上面的例子中加载了 sst2 数据子集。

还可以使用参数 split 直接指定要加载的数据部分，在上面的例子中加载了数据的 train 部分。

运行结果如下：

```
Dataset({
    features: ['sentence', 'label', 'idx'],
    num_rows: 67349
```

```
    })
```

可以看到，glue 的 sst2 数据子集的 train 部分有 67 349 条数据，每条数据都具有 sentence、label 和 idx 字段。

2. 将数据集保存到本地磁盘

加载了数据集后，可以使用 save_to_disk()函数将数据集保存到本地磁盘，代码如下：

```
#第 3 章/将数据集保存到磁盘
dataset.save_to_disk(
    dataset_dict_path='./data/ChnSentiCorp')
```

3. 从本地磁盘加载数据集

保存到磁盘以后可以使用 load_from_disk()函数加载数据集，代码如下：

```
#第 3 章/从磁盘加载数据集
from datasets import load_from_disk
dataset = load_from_disk('./data/ChnSentiCorp')
```

3.2.2　数据集基本操作

1. 取出数据部分

为了便于做后续的实验，这里取出数据集的 train 部分，代码如下：

```
#使用 train 数据子集做后续的实验
dataset = dataset['train']
```

2. 查看数据内容

可以查看部分数据样例，代码如下：

```
#第 3 章/查看数据样例
for i in [12, 17, 20, 26, 56]:
    print(dataset[i])
```

运行结果如下：

```
    {'text': '轻便,方便携带,性能也不错,能满足平时的工作需要,对出差人员来讲非常不错',
'label': 1}
    {'text': '很好的地理位置,一塌糊涂的服务,萧条的酒店。', 'label': 0}
    {'text': '非常不错,服务很好,位于市中心区,交通方便,不过价格也高! ', 'label': 1}
    {'text': '跟住招待所没什么太大区别。绝对不会再住第 2 次的酒店! ', 'label': 0}
    {'text': '价格太高,性价比不够好。我觉得今后还是去其他酒店比较好。', 'label': 0}
```

到这里，可以看出数据是什么内容了，这是一份购物和消费评论数据，字段 text 表示消费者的评论，字段 label 表明这是一段好评还是差评。

3. 数据排序

可以使用 sort()函数让数据按照某个字段排序，代码如下：

```
#第 3 章/排序数据
#数据中的 label 是无序的
print(dataset['label'][:10])
#让数据按照 label 排序
sorted_dataset = dataset.sort('label')
print(sorted_dataset['label'][:10])
print(sorted_dataset['label'][-10:])
```

运行结果如下：

```
[1, 1, 0, 0, 1, 0, 0, 0, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

可以看到，初始数据是乱序的，使用 sort()函数后，数据按照 label 排列为有序的了。

4. 打乱数据

和 sort()函数相对应，可以使用 shuffle()函数再次打乱数据，代码如下：

```
#第 3 章/打乱数据顺序
shuffled_dataset=sorted_dataset.shuffle(seed=42)
shuffled_dataset['label'][:10]
```

运行结果如下：

```
[0, 1, 0, 0, 1, 0, 1, 0, 1, 0]
```

可以看到，数据再次被打乱为无序。

5. 数据抽样

可以使用 select()函数从数据集中选择某些数据，代码如下：

```
#第 3 章/从数据集中选择某些数据
dataset.select([0, 10, 20, 30, 40, 50])
```

运行结果如下：

```
Dataset({
    features: ['text', 'label'],
    num_rows: 6
})
```

选择出的数据会再次组装成一个数据子集，使用这种方法可以实现数据抽样。

6. 数据过滤

使用 filter()函数可以按照自定义的规则过滤数据，代码如下：

```
#第 3 章/过滤数据
def f(data):
    return data['text'].startswith('非常不错')
dataset.filter(f)
```

filter()函数接受一个函数作为参数，在该函数中确定过滤数据的条件，在上面的例子中数据过滤的条件是评价以“非常不错”开头，运行结果如下：

```
Dataset({
    features: ['text', 'label'],
    num_rows: 13
})
```

可以看到，满足评价以“非常不错”开头的数据共有 13 条。

7. 训练测试集拆分

可以使用 train_test_split()函数将数据集切分为训练集和测试集，代码如下：

```
#第 3 章/切分训练集和测试集
dataset.train_test_split(test_size=0.1)
```

参数 test_size 表明测试集占数据总体的比例，例子中占 10%，可知训练集占 90%，运行结果如下：

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 8640
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 960
    })
})
```

可以看到，数据集被切分为 train 和 test 两部分，并且两部分数据量的比例满足 9:1。

8. 数据分桶

可以使用 shared ()函数把数据均匀地分为 *n* 部分，代码如下：

```
#第 3 章/数据分桶
dataset.shard(num_shards=4, index=0)
```

（1）参数 num_shards 表明要把数据均匀地分为几部分，例子中分为 4 部分。

（2）参数 index 表明要取出第几份数据，例子中为取出第 0 份。

运行结果如下：

```
Dataset({
```

```
    features: ['text', 'label'],
    num_rows: 2400
})
```

因为原数据集数量为9600条，均匀地分为4份后每一份是2400条，和上面的输出一致。

9. 重命名字段

使用rename_column()函数可以重命名字段，代码如下：

```
#第3章/字段重命名
dataset.rename_column('text', 'text_rename')
```

运行结果如下：

```
Dataset({
    features: ['text_rename', 'label'],
    num_rows: 9600
})
```

原始字段text现在已经被重命名为text_rename。

10. 删除字段

使用remove_columns()函数可以删除字段，代码如下：

```
#第3章/删除字段
dataset.remove_columns(['text'])
```

运行结果如下：

```
Dataset({
    features: ['label'],
    num_rows: 9600
})
```

可以看到字段text现在已经被删除。

11. 映射函数

有时希望对数据集总体做一些修改，可以使用map()函数遍历数据，并且对每条数据都进行修改，代码如下：

```
#第3章/应用函数
def f(data):
    data['text'] = 'My sentence: ' + data['text']
    return data
maped_datatset = dataset.map(f)
print(dataset['text'][20])
print(maped_datatset['text'][20])
```

map()函数是很强大的一个函数，map()函数以一个函数作为入参，在该函数中确定要对

数据进行的修改，可以是对数据本身的修改，例如例子中的代码就是对 text 字段增加了一个前缀，也可以进行增加字段、删除字段、修改数据格式等操作，运行结果如下：

```
非常不错,服务很好,位于市中心区,交通方便,不过价格也高!
My sentence: 非常不错,服务很好,位于市中心区,交通方便,不过价格也高!
```

经过 map()函数的映射后 text 字段多了一个前缀，而原始数据则没有。

12. 使用批处理加速

在使用过滤和映射这类需要使用一个函数遍历数据集的方法时，可以使用批处理减少函数调用的次数，从而达到加速处理的目的。在默认情况下是不使用批处理的，由于每条数据都需要调用一次函数，所以函数调用的次数等于数据集中数据的条数，如果数据的数量很多，则需要调用很多次函数。使用批处理函数，能够一批一批地处理数据，让函数调用的次数大大减少，代码如下：

```
#第 3 章/使用批处理加速
def f(data):
    text=data['text']
    text=['My sentence: ' + i for i in text]
    data['text']=text
    return data
maped_datatset=dataset.map(function=f,
                           batched=True,
                           batch_size=1000,
                           num_proc=4)
print(dataset['text'][20])
print(maped_datatset['text'][20])
```

在这段代码中，调用了数据集的 map()函数，对数据进行了映射操作，但这次除了数据处理函数之外，还额外传入了很多参数，下面对这些参数进行讲解。

（1）参数 batched=True 和 batch_size=1000：表示以 1000 条数据为一个批次进行一次处理，这将把函数执行的次数削减约 1000 倍，提高了运行效率，但同时对内存会提出更高的要求，读者需要结合自己的运算设备调节合适的值，通常来讲，1000 是个合适的值。

（2）参数 num_proc=4：表示在 4 条线程上执行该任务，同样是和性能相关的参数，读者可以结合自己的运算设备调节该值，一般设置为 CPU 核心数量。

当使用批处理处理数据时，每次传入处理函数的就不是一条数据了，而是一个批次的数据。在上面的例子中，一个批次为 1000 条数据，在编写处理函数时需要注意，以上代码的运行结果如下：

```
非常不错,服务很好,位于市中心区,交通方便,不过价格也高!
My sentence: 非常不错,服务很好,位于市中心区,交通方便,不过价格也高!
```

可以看到，数据处理的结果和使用单条数据映射时的结果一致，使用批处理仅仅是性能

上的考量，不会影响数据处理的结果。

13. 设置数据格式

使用 set_format()函数修改数据格式，代码如下：

```
    #第 3 章/设置数据格式
    dataset.set_format(type='torch', columns=['label'], output_all_columns=
True)
    dataset[20]
```

（1）参数 type 表明要修改为的数据类型，常用的取值有 numpy、torch、tensorflow、pandas 等。

（2）参数 columns 表明要修改格式的字段。

（3）参数 output_all_columns 表明是否要保留其他字段，设置为 True 表明要保留。

运行结果如下：

```
    {'label': tensor(1), 'text': '非常不错,服务很好,位于市中心区,交通方便,不过价格
也高! '}
```

字段 label 已经被修改为 PyTorch 的 Tensor 格式。

3.2.3 将数据集保存为其他格式

1. 将数据保存为 CSV 格式

可以把数据集保存为 CSV 格式，便于分享，同时数据集工具也有加载 CSV 格式数据的方法，代码如下：

```
    #第 3 章/导出为 CSV 格式
    dataset = load_dataset(path='seamew/ChnSentiCorp', split='train')
    dataset.to_csv(path_or_buf='./data/ChnSentiCorp.csv')
    #加载 CSV 格式数据
    csv_dataset = load_dataset(path='csv',
                           data_files='./data/ChnSentiCorp.csv',
                           split='train')
    csv_dataset[20]
```

运行结果如下：

```
    {'Unnamed: 0': 20, 'text': '非常不错,服务很好,位于市中心区,交通方便,不过价格也
高! ', 'label': 1}
```

可以看到，保存为 CSV 格式后再加载，多了一个 Unnamed 字段，在这一列中实际保存的是数据的序号，这和保存的 CSV 文件内容有关系。如果不想要这一列，则可以直接到 CSV 文件去删除第 1 列，删除时可以使用数据集的删除列功能，在此不再赘述。

2. 保存数据为 JSON 格式

除了可以保存为 CSV 格式外，也可以保存为 JSON 格式，方法和 CSV 格式大同小异，代码如下：

```
#第 3 章/导出为 JSON 格式
dataset=load_dataset(path='seamew/ChnSentiCorp', split='train')
dataset.to_json(path_or_buf='./data/ChnSentiCorp.json')
#加载 JSON 格式数据
json_dataset=load_dataset(path='json',
                          data_files='./data/ChnSentiCorp.json',
                          split='train')
json_dataset[20]
```

运行结果如下：

```
{'text': '非常不错，服务很好，位于市中心区，交通方便，不过价格也高！', 'label': 1}
```

可以看到，保存为 JSON 格式并不存在多列的问题。

3.3　小结

本章讲解了 HuggingFace 数据集工具的使用，包括数据的加载、保存、查看、排序、抽样、过滤、拆分、映射、列重命名等操作。