首先我们需要将xml文件中有用的数据提取出来、在xml文件中，这包括句子的id、<text>和<aspectTerms>中每个<aspectTerm>的内容，即term, polarity, from和to。然后，需要将这些内容的信息转换成一个csv文件，每列的内容分别是：id，text，term，polarity，from和to。然后，我们可以对数据做进一步分析，采用xml这个库中etree.ElementTree的方法解析XML数据，得到关于text的一个列表text\_list和以term为键，polarity为值的一个字典opinion，通过nltk.FreqDist(opinion).most\_common(t)方法，获得最常见的t个aspects，这样可以分析评论最关心的是哪些方面。例如下图即在饭店数据集中，评论最关心的前20个方面。

在处理XML文件之前，我们发现文件中有非ASCII的空格字符，我们用space替换了这样的异常空格字符。

由于网站给的测试集没有情感标注，所以我们将数据集的90%设置为数据集，剩下10%中，40%用作验证集，60%则作为最终的测试集。所得到的训练集、验证集和测试集信息如下图所示。

为了更好地处理数据，我们创建了一个Reviews的类，它是PyTorch的Dataset类的子类。这个类用于处理包含评论、评论方面、目标标签等信息的数据。这个类包含以下参数：reviews, review\_aspects, targets, tokenizer, max\_len，它可以根据索引item从reviews、targets等属性中获取对应的评论内容和目标标签，使用tokenizer对象对评论内容进行编码，将评论内容转换为模型可以接受的输入格式，并进行必要的填充和截断操作。而编码结果以字典形式保存在encoding变量中。通过创建Reviews类的实例，可以将原始的评论数据集转换为适合用于训练和评估BERT模型的数据集对象。这个自定义数据集可以与PyTorch的数据加载器（如DataLoader）一起使用，用于批量加载和处理数据。

我们从Hugging Face的模型库中加载bert-base-cased预训练模型，并创建一个相应的tokenizer对象。'bert-base-cased'是BERT模型的基础版本之一。基础版本通常指的是相对较小的模型，具有较少的参数和计算资源需求。相比于其他版本（如large或xlarge），base版本在模型规模上更为轻量级，但仍具备良好的性能。

函数接收以下参数：df: 包含数据的DataFrame，其中包括文本、评论方面和目标标签等列。tokenizer: 用于将文本转换为模型输入的tokenizer对象。max\_len: 设定的最大文本长度，用于进行填充和截断文本以及batch\_size: 每个批次的样本数量。通过调用data\_loader函数，我们可以轻松地创建数据加载器对象，用于对数据集进行批量加载和处理。这些数据加载器可以与训练、验证和测试步骤一起使用，以便有效地训练和评估模型。

接着我们需要利用BERT构建情感分析模型。我们定义了一个名为SentimentClassifier的PyTorch模型类，它继承自nn.Module，表示这个类是一个神经网络模型。它接收一个参数 n\_classes，表示情感分类任务中的类别数量（在我们这个任务中是四类）。这里一行关键的实现是：self.bert = BertModel.from\_pretrained(PRE\_TRAINED\_MODEL\_NAME, output\_hidden\_states=True, output\_attentions=True, return\_dict=False)，它创建了一个BERT模型对象，并加载预训练的BERT权重。PRE\_TRAINED\_MODEL\_NAME是预训练模型的名称，它指定了要使用的BERT模型的类型，这里即'bert-base-cased'。output\_hidden\_states=True和output\_attentions=True表示要获取BERT模型的所有隐藏状态和注意力权重。return\_dict=False表示返回的结果将是BERT模型输出的元组形式。然后我们定义了一个Dropout层，用于在模型的输出中进行随机失活，丢弃概率为30%。接着我们定义了一个线性层（全连接层），将BERT模型的隐藏状态大小作为输入特征数，并将其映射到n\_classes个输出类别。然后我们定义了模型类的前向传播方法，它接收两个参数：一个是输入的token ID序列，用于表示输入的文本；另外一个是输入的注意力掩码，用于指示哪些部分是真实有效的文本。

在前向传播过程中，输入的token ID序列和注意力掩码通过BERT模型进行处理。通过调用self.bert(input\_ids=input\_ids, attention\_mask=attention\_mask)，我们可以获取BERT模型的输出元组，其中包含了BERT模型的多个输出，包括最后一层的隐藏状态、池化输出、隐藏状态的序列以及注意力权重。不过，在这个项目中，我们只关注池化输出（pooled\_output），它是BERT模型将整个输入序列的信息进行汇总后得到的表示。这个表示经过Dropout层进行随机失活后，通过线性层（全连接层）进行分类预测，最终返回预测结果。

最后，我们设置了训练过程中的一些参数和工具，包括优化器、学习率调度器和损失函数。我们对整个训练数据集进行了10次迭代训练。我们使用了AdamW优化算法，将模型的参数（model.parameters()）作为优化器的参数，并设置了学习率（lr=2e-5）和是否纠正偏差（correct\_bias=False）。接着，我们通过将训练数据加载器中的批次数量乘以训练的轮数，得到了需要执行的总的训练步数。然后，我们用get\_linear\_schedule\_with\_warmup创建了一个学习率调度器对象。它可以基于给定的优化器、预热步数和总的训练步数，根据训练的进展情况自动调整学习率。最后，我们的损失函数选择的是交叉熵损失函数。

这些代码段设置了训练过程中的关键组件。优化器用于更新模型的参数，学习率调度器用于调整学习率的变化，损失函数用于计算训练过程中的损失。这些组件将在训练循环中使用，以优化模型并计算损失。

最后，我们设计了一个训练循环函数，遍历数据加载器，每次迭代从数据加载器中获取一个batch的数据，并且设定了注意力掩码，使用模型进行前向传播，得到模型的预测输出，通过在预测输出张量的第一个维度上找到最大值，获得每个样本的预测类别。我们还利用了前面提到的交叉熵损失函数，计算模型的预测输出与目标标签之间的损失值。接着，我们通过比较预测类别和目标标签是否相等，并将结果求和，计算预测正确的数量。然后，我们在计算图中进行反向传播，计算损失相对于模型参数的梯度。循环结束后，计算整个epoch的平均损失值和准确率，并返回这些值。

在数据加载器迭代过程中，对每个数据样本进行以下操作：

提取文本信息到texts变量。

将输入数据转移到指定设备（如GPU）上。

调用模型进行预测，得到输出结果outputs。

使用torch.max()函数找到输出结果中概率最高的类别，即预测结果preds。

将当前数据样本的文本、预测结果、预测概率和真实值分别添加到对应的列表中。

最后，通过torch.stack()函数将列表中的张量沿指定维度堆叠，并将其转移到CPU上，最终返回文本、预测结果、预测概率和真实值。

在本次自然语言处理项目中，我们运用了BERT语言模型做了情感分析，得到了初步的结果。这个模型可以用来做简单的情感分类任务，但是它也存在准确度不够高（70%-80%），样本数比较少（只有两三千条数据）的不足。并且，随着训练次数的扩大，测试集的准确率还会略有下降。这需要我们在之后的工作中进一步发掘其中的原因。