1. **深度学习框架、工具等的安装及配置过程**
   1. 深度学习框架的安装与配置

Anaconda的安装与配置：进入Anaconda官网进行下载，按照步骤进行安装，然后配置环境。(可以参照网上教程进行安装)。

在Anaconda中创建环境：conda create -n nlpwork python==3.8.13，成功创建python环境如图1.1所示。

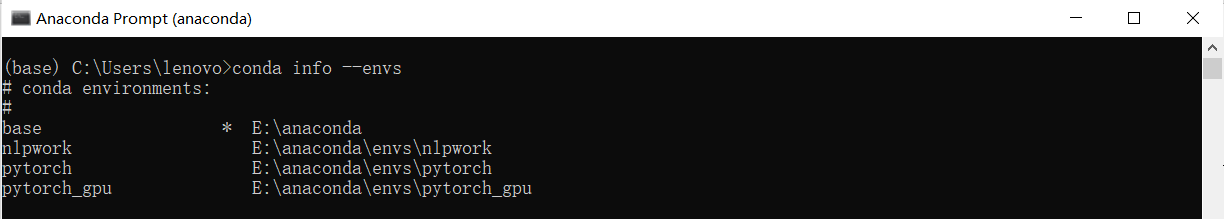


图 1. 1 成功创建python环境

输入指令激活已经创建的python环境：activate nlpwork，然后输入：conda list查看已经安装的python库。python库主要包括pandas、transformers、torch、torchtext、scikit-learn等，激活环境并查看安装的python库如图1.2所示。

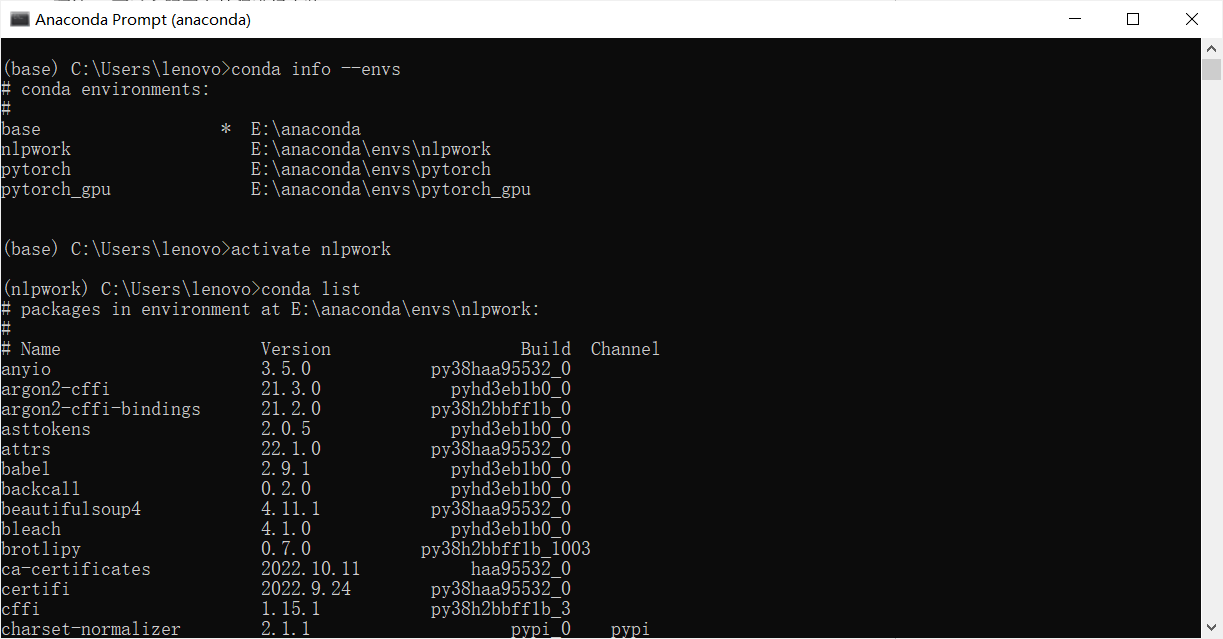


图 1. 2 激活环境并查看安装的python库

相关python库的安装使用：conda install package\_name,如果网速不行，可以使用清华源等国内的镜像源。

* 1. 安装torchtext库

在安装torchtext库时，因为torchtext库因为版本不同的问题导致某些函数的所在位置可能会发生变化。所以尽可能安装torchtext库的版本为0.9.0

如图1.3所示为所安装的torchtext库。

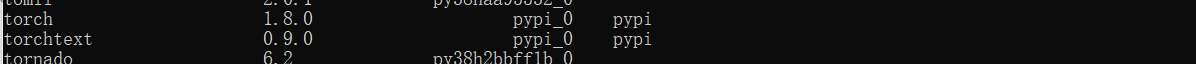


图 1. 3 所安装的torchtext库

* 1. 安装 spaCy

spaCy库是一个NLP领域的文本预处理Python库，包括分词(Tokenization)、词性标注(Part-of-speech Tagging, POS Tagging)、依存分析(Dependency Parsing)、词形还原(Lemmatization)、句子边界检测(Sentence Boundary Detection，SBD)、命名实体识别(Named Entity Recognition, NER)等功能。spaCy具有简单易用，支持多种语言、使用深度学习模型进行分词等任务、提供不同尺寸的模型可供选择。

输入：pip install spacy -i https://pypi.tuna.tsinghua.edu.cn/simple进行安装。

* 1. Jupyter Notebook 的安装

Jupyter Notebook是基于网页的用于交互计算的应用程序。其可被应用于全过程计算：开发、文档编写、运行代码和展示结果。在安装Anaconda时，会自动为用户安装Jupyter Notebook。如图1.4所示，Anaconda中的Jupyter Notebook。图1.5所示为Jupyter Notebook的主界面。

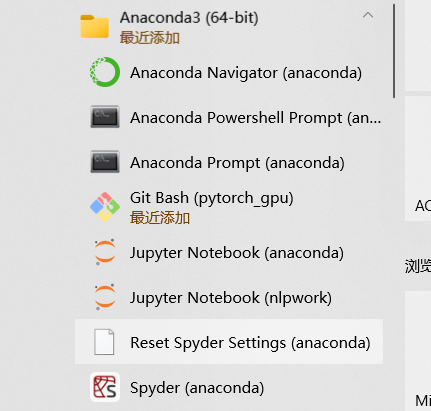


图 1. 4 Anaconda中的Jupyter Notebook

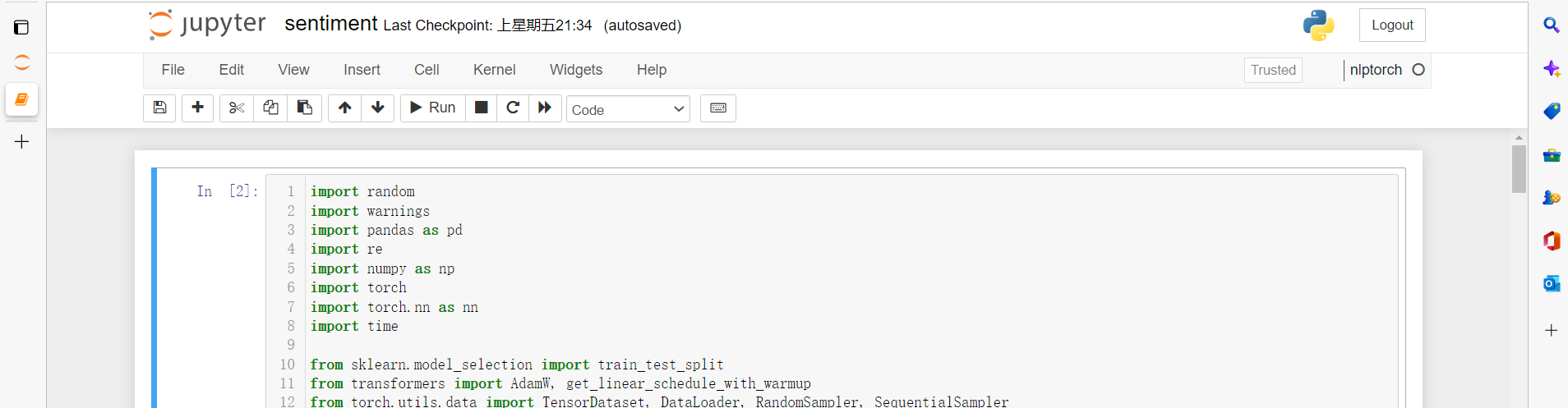


图 1. 5 Jupyter Notebook的主界面

* 1. PyCharm的安装

PyCharm是一种Python IDE（Integrated Development Environment，集成开发环境），带有一整套可以帮助用户在使用Python语言开发时提高其效率的工具，比如调试、语法高亮、项目管理、代码跳转、智能提示、自动完成、单元测试、版本控制。

详细安装过程参考<https://blog.csdn.net/qq_37278346/article/details/127011079>

1. **采用的深度学习模型的详细描述**
   1. BERT模型简介

BERT是2018年10月由Google AI研究院提出的一种预训练模型，全称是Bidirectional Encoder Representation from Transformers。BERT在机器阅读理解顶级水平测试SQuAD1.1中表现出惊人的成绩: 全部两个衡量指标上全面超越人类，并且在11种不同NLP测试中创出SOTA表现，包括将GLUE基准推高至80.4% (绝对改进7.6%)，MultiNLI准确度达到86.7% (绝对改进5.6%)，成为NLP发展史上的里程碑式的模型成就。首次提出BERT模型的论文BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding发表在自然语言处理顶会NAACL上，并获得了该届会议的最佳长论文。BERT的基础建立在transformer之上，拥有强大的语言表征能力和特征提取能力。BERT模型在GLUE基准测试中的效果见表1。

表1 BERT模型在GLUE基准测试中的效果

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| System | MNLI-(m/mm)  392k | QQP  363k | QNLI  108k | SST-2  67k | CoLA  8.5k | STS-B  5.7k | MRPC  3.5k | RTE  2.5k | Average |
| Pre-OpenAl SOTA | 80.6/80.1 | 66.1 | 82.3 | 93.2 | 35.0 | 81.0 | 86.0 | 61.7 | 74.0 |
| BiLSTM+ELMo+Attn | 76.4/76.1 | 64.8 | 79.8 | 90.4 | 36.0 | 73.3 | 84.9 | 56.8 | 71.0 |
| OpenAI GPT | 82.1/81.4 | 70.3 | 87.4 | 91.3 | 45.4 | 80.0 | 82.3 | 56.0 | 75.1 |
| BERTBASE | 84.6/83.4 | 71.2 | 90.5 | 93.5 | 52.1 | 85.8 | 88.9 | 66.4 | 79.6 |
| BERTLARGE | 86.7/85.9 | 72.1 | 92.7 | 94.9 | 60.5 | 86.5 | 89.3 | 70.1 | 82.1 |

* 1. 模型结构

BERT模型是一个双向编码模型，采用了Transformer Encoder block进行连接，最后在BERT后面连接上特定任务的分类器。BERT模型结构可以说就是Transformer的encoder部分，BERT模型有两种，一种叫BERT-base对应的是12层encoder，另一种叫做BERT-large对应的是24层encoder，两者主要是在模型复杂度上不同，BERT-base总的参数量为110M，BERT-large总的参数量为340M。

BERT是基于Transformer的双向预训练语言模型。模型的使用分为两个部分：pre-training(预训练)和fine-tuning(微调)。在pre-training过程中，模型将使用无标签的数据(unlabled data)来对BERT的两个预训练任务(Masked LM and NSP)进行训练。在fine-tuning过程中，BERT模型将用预训练模型初始化所有参数，这些参数将针对于下游任务，如分类任务等，使用labeled data进行训练。针对不同的下游任务可以训练出不同的模型，但是它们都是由同一个预训练模型进行初始化而来的。BERT模型除了输出层外，在预训练和微调中使用了相同的架构，相同的预训练模型参数用于初始化不同的下游任务的模型，在微调中，对所有参数进行微调。如图2.1所示为BERT模型的预训练和微调的架构。

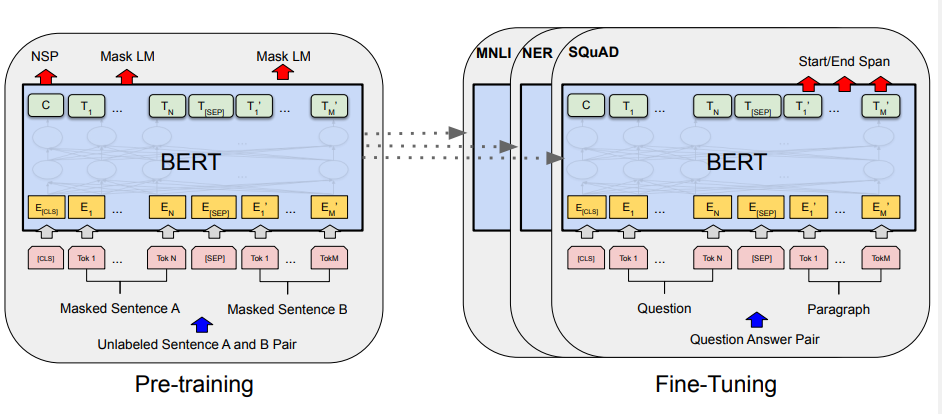


图 2. 1 BERT模型的预训练和微调的架构，图中[CLS]是在每个输入示例前添加的一个特殊符号，[SEP]是一个特殊的分隔符。

BERT模型使用示例如下图2.2所示。

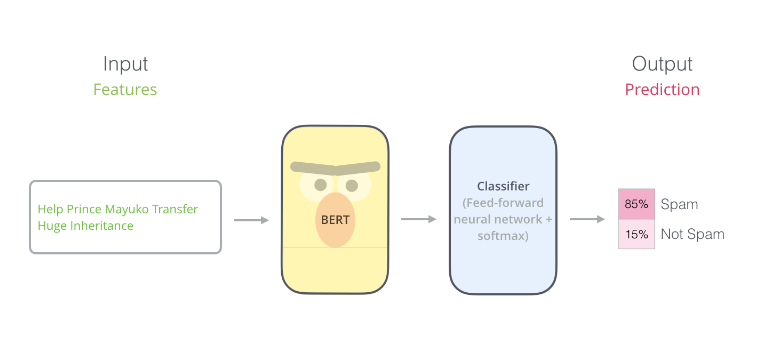


图 2. 2 BERT模型使用示例

* 1. 模型的输入和输出

BERT模型的输入的特殊之处在于，在一句话最开始拼接了一个[CLS] token，如图2.3所示。这个特殊的[CLS] token经过BERT得到的向量表示通常被用作当前的句子表示。除此之外，其余输入的特殊单词类似Transformer。BERT模型将一串单词作为输入，这些单词在多层encoder中不断向上流动，每一层都会经过自注意力机制和前馈神经网络。BERT模型输入的文本是在进行token时是有自己的方式的，使用的是WordPieces作为最小的处理单元，语句处理成tokens后就需要考虑位置编码以及tokens, CLS，SEP进行Embedding编码。与Transformer不同的地方在于位置编码的使用，Transformer中的位置编码使用三角函数进行表示，而BERT模型的位置编码将在预训练过程中得到。

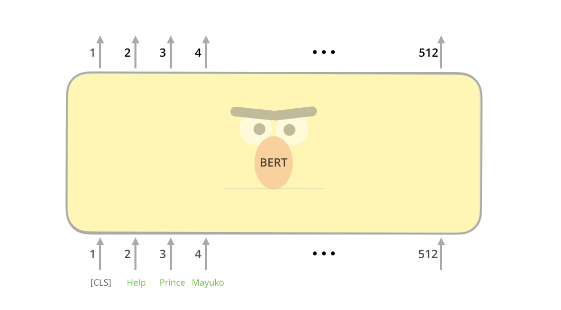


图 2. 3 输入的句子的开端拼接了一个[CLS] token

BERT的输入表示如图2.4所示。

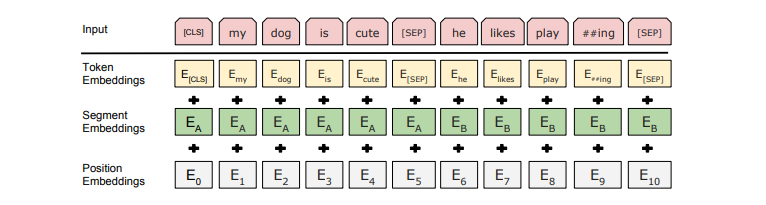


图 2. 4 BERT的输入表示

BERT模型输入的所有token经过编码以后，会在每个位置输出一个大小为hidden\_size的向量。原论文中在文本分类任务中，在获取[CLS]对应的向量以后，连接上一个分类器就可以进行分类。如图2.5所示为单个句子文本分类任务的实现。BERT模型不仅可以使用最后一层的BERT的输出，还可以使用每一个encoder layer的每一个token的向量作为特征。比如，直接提取每一个encoder的token表示当作特征，输入现有的特定任务神经网络中进行训练。如图2.6所示。

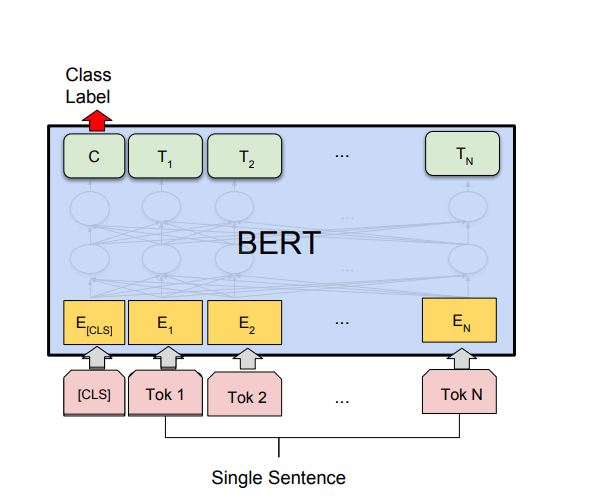


图 2. 5 单个句子文本分类任务的实现

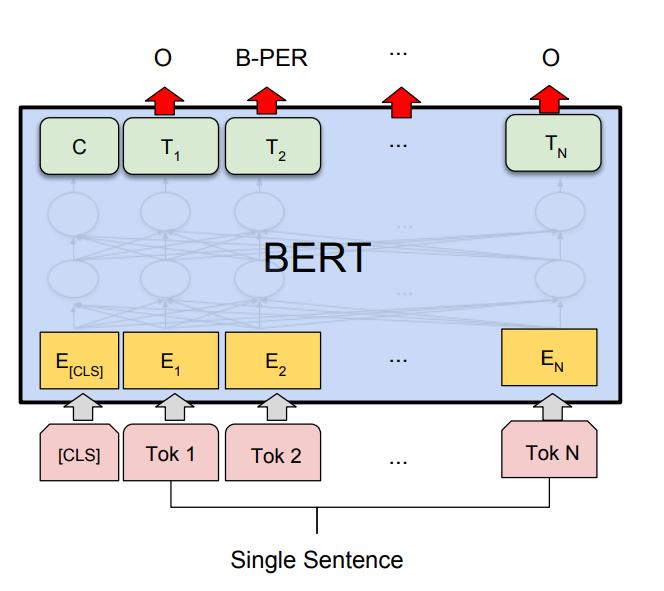


图 2. 6

* 1. 模型的预训练过程

BERT模型预训练的过程使用的是Masked language model(掩码语言模型)。通过将输入文本序列的部分单词随机遮掩掉(即Mask掉)，然后让模型去预测这些被Mask的单词。如图2.7所示为BERT的掩码语言模型架构。这样做的好处在于可以加入噪声，然后再加入噪声以后进行训练可以让模型更加的稳定。但是这也带来了模型收敛较慢的后果。

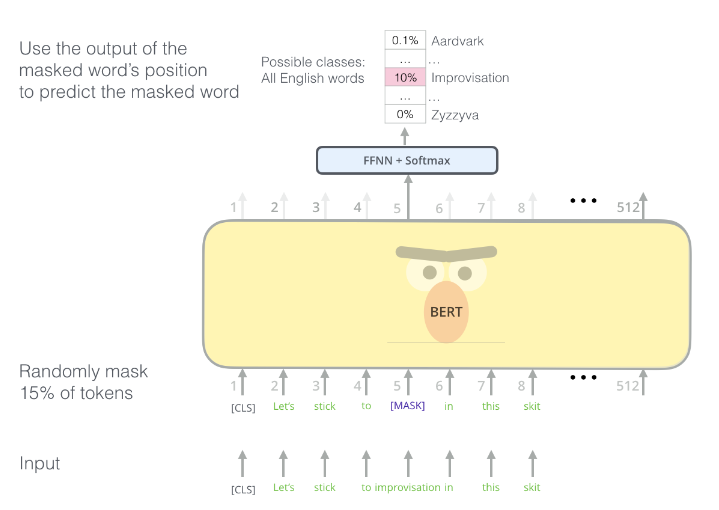


图 2. 7 BERT的掩码语言模型架构

* 1. 损失函数

考虑到所要完成的任务为二分类或者多分类任务，所以损失函数选择使用交叉熵损失函数。交叉熵是信息论中的一个重要的概念，主要应用于度量两个概率分布之间的差异性。交叉熵能够衡量同一个随机变量中的两个不同概率分布的差异程度，在机器学习中就表示为真实概率分布与预测概率分布之间的差异。交叉熵的值越小，模型预测效果就越好。计算公式见下式。

式中表示样本的真实分布，表示模型所预测的分布。

1. **源代码及注释**

**3.1 导入数据并对数据进行初步处理**

1. import random
2. import warnings
3. import pandas as pd
4. import re
5. import numpy as np
6. import torch
7. import torch.nn as nn
8. import time
9. from sklearn.model\_selection import train\_test\_split
10. from transformers import AdamW, get\_linear\_schedule\_with\_warmup
11. from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
12. from bs4 import BeautifulSoup
13. from torchtext.legacy import data
14. from torchtext.legacy import datasets
15. from transformers import BertTokenizer, BertModel
16. *#读取数据集*
17. warnings.filterwarnings('ignore')
18. data = pd.read\_csv('./Desktop./zkm./labeledTrainData.tsv',header=0, delimiter="\t", quoting=3)
19. *# print(type(data))*
20. def dataSet\_preprocess(review):
21. *#任务一：去掉html标记。*
22. raw\_text = BeautifulSoup(review,'html').get\_text()
23. *#任务二：去掉非字母字符,sub(pattern, replacement, string) 用空格代替*
24. letters = re.sub('[^a-zA-Z]',' ',raw\_text)
25. *#str.split(str="", num=string.count(str)) 通过指定分隔符对字符串进行切片，如果参数 num 有指定值，则仅分隔 num 个子字符串*
26. *#这里是先将句子转成小写字母表示，再按照空格划分为单词list*
27. words = letters.lower().split()
28. *#     #获取停用词，从数据集中去掉停用词*
29. *#     stop\_words\_file = './Desktop./zkm./english.txt'*
30. *#     stopwords = get\_stopwords(stop\_words\_file)*
31. *#     words = [word for word in words if word not in stopwords]*
32. *#     ' '.join(words)*
33. return words
34. *#从停用词文件中获取停用词表*
35. def get\_stopwords(stop\_words\_file):
36. with open(stop\_words\_file,encoding='utf-8') as f:
37. stopwords = f.read()
38. stopwords\_list = stopwords.split('\n')
39. custom\_stopwords\_list = [i for i in stopwords\_list]
40. return custom\_stopwords\_list
41. review\_data = []
42. sentiment\_data = []
43. *#review\_data存放评论*
44. for review in data['review']:
45. review\_data.append(' '.join(dataSet\_preprocess(review)))
46. *#sentiment\_data存放每条评论相对应的情感倾向：1为积极的，0为消极的*
47. for sentiment in data['sentiment']:
48. sentiment\_data.append(sentiment)
49. *# print(type(review\_data))  #list*
50. data["review"] = pd.DataFrame(review\_data)
51. data["sentiment"] = pd.DataFrame(sentiment\_data)
52. print(data)
53. print(type(data))
54. *#截断数据操作*
55. data["review"] = data["review"].str[:2000]
56. *# data = pd.DataFrame.truncate(data, after = 100)*
57. *#获取review中字符串的最大长度*
58. length\_of\_the\_messages = data["review"].str.split("\\s+")
59. print(length\_of\_the\_messages.str.len().max())
60. data\_labels = pd.DataFrame(data.sentiment.values)
61. data\_reviews = pd.DataFrame(data.review.values)
62. *# print(type(data\_labels))*
63. *# print(data\_reviews)*
64. """
65. 将训练数据分为两组，一组是训练集(包含80%的样本)；一组是测试集(包含20%的样本)
66. """
67. X = data.review.values[:100]  *# review*
68. Y = data.sentiment.values[:100]  *# label*
69. X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.2)
70. *# print(X\_train)*
71. *# print(Y\_train)*
72. *# print(X\_test)*
73. *# print(Y\_test)*

运行结果：

IMDB数据集中包含25000条标注好的评论，其中积极评论和消极评论各15000条。将数据集按照8：2的比例划分为训练集和测试集。此部分代码将数据导入以后对数据进行清洗，去除html标记和非字母字符等。图3.1所示为数据集内容、数据集大小、数据集中文本的最大长度。

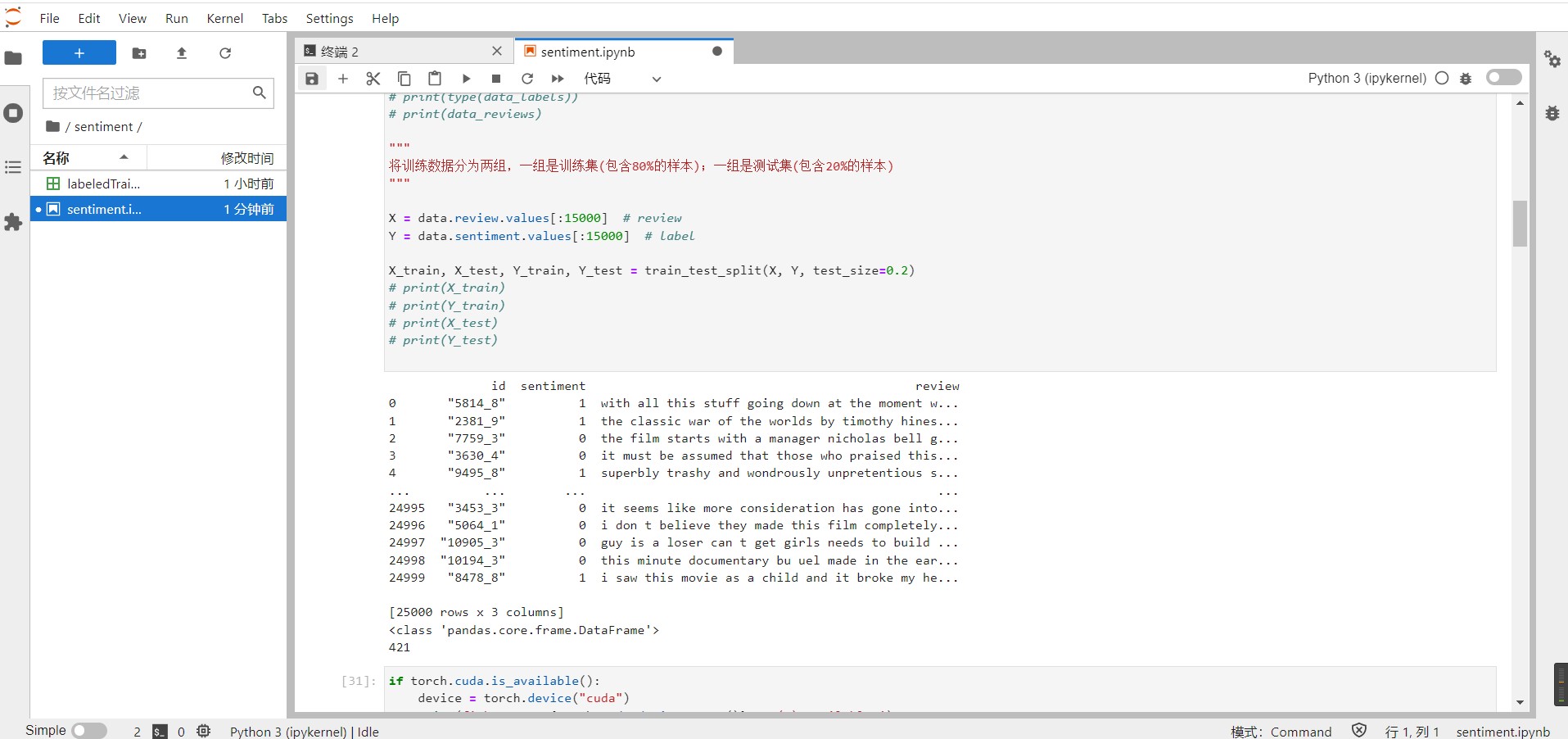


图 3. 1 数据集内容、数据集大小、数据集中文本的最大长度

**3.2 查看设备硬件条件**

1. """
2. 查看设备硬件条件，有GPU就使用GPU，否则使用CPU
3. """
4. if torch.cuda.is\_available():
5. device = torch.device("cuda")
6. print(f'There are {torch.cuda.device\_count()} GPU(s) available.')
7. print('Device name:', torch.cuda.get\_device\_name(0))
8. else:
9. print('No GPU available, using the CPU instead.')
10. device = torch.device("cpu")

运行结果：

在代码调试过程中发现：该模型对于硬件条件的要求比较高，本人电脑内存16GB，显卡：NVIDIA GeForce GTX 1050，并不能满足条件，会出现内存爆掉的情况，模型跑不起来。因此在云服务器上租了一块GPU：NVIDIA GeForce RTX 3090，24G显存，90G内存上可以很好地运行，在batch\_size设为16时，显存使用了15G，40个batch大概耗时12s。图3.2所示为batch\_size设为16时，设备的使用情况。图3.3为上述代码的运行结果。

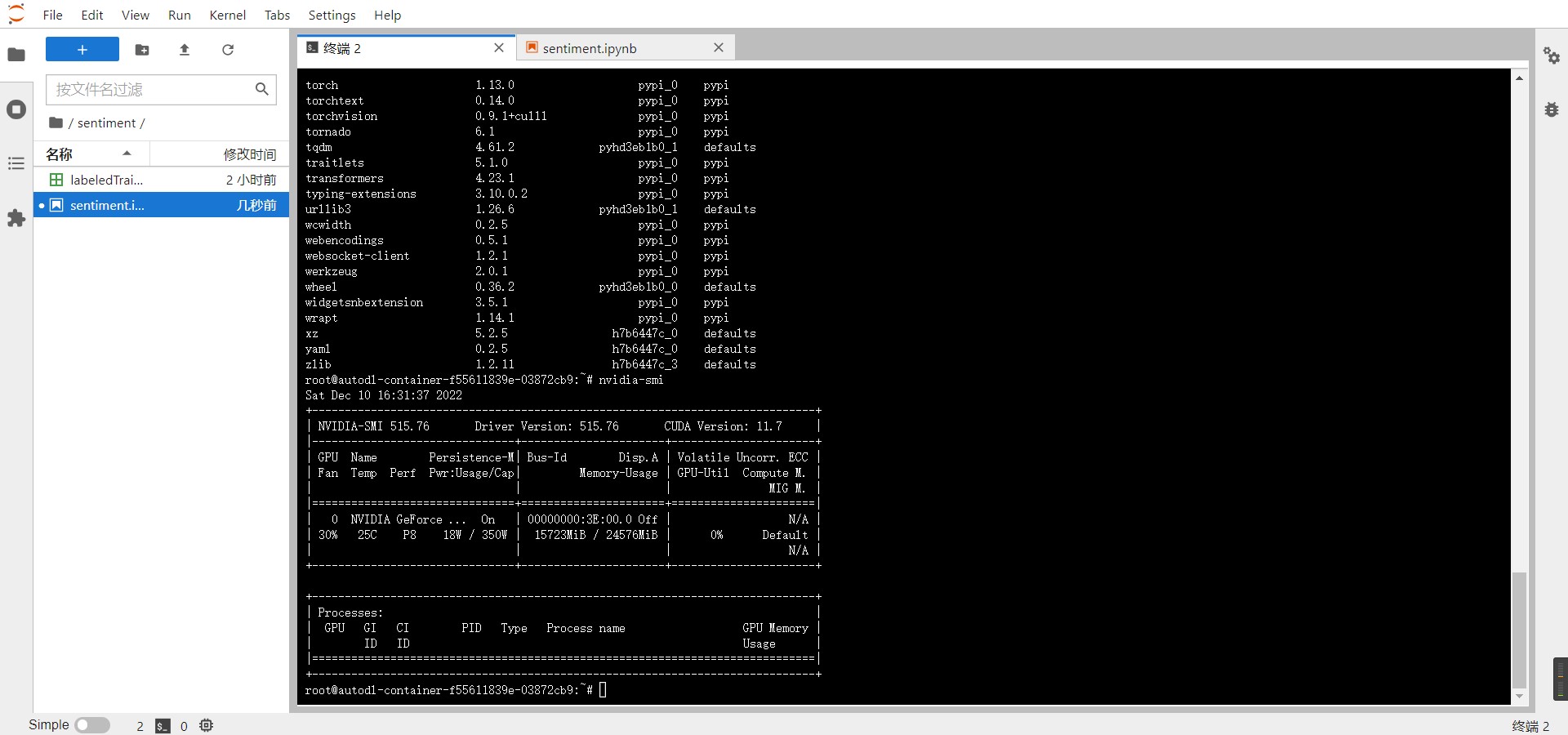


图 3. 2 batch\_size设为16时，设备的使用情况

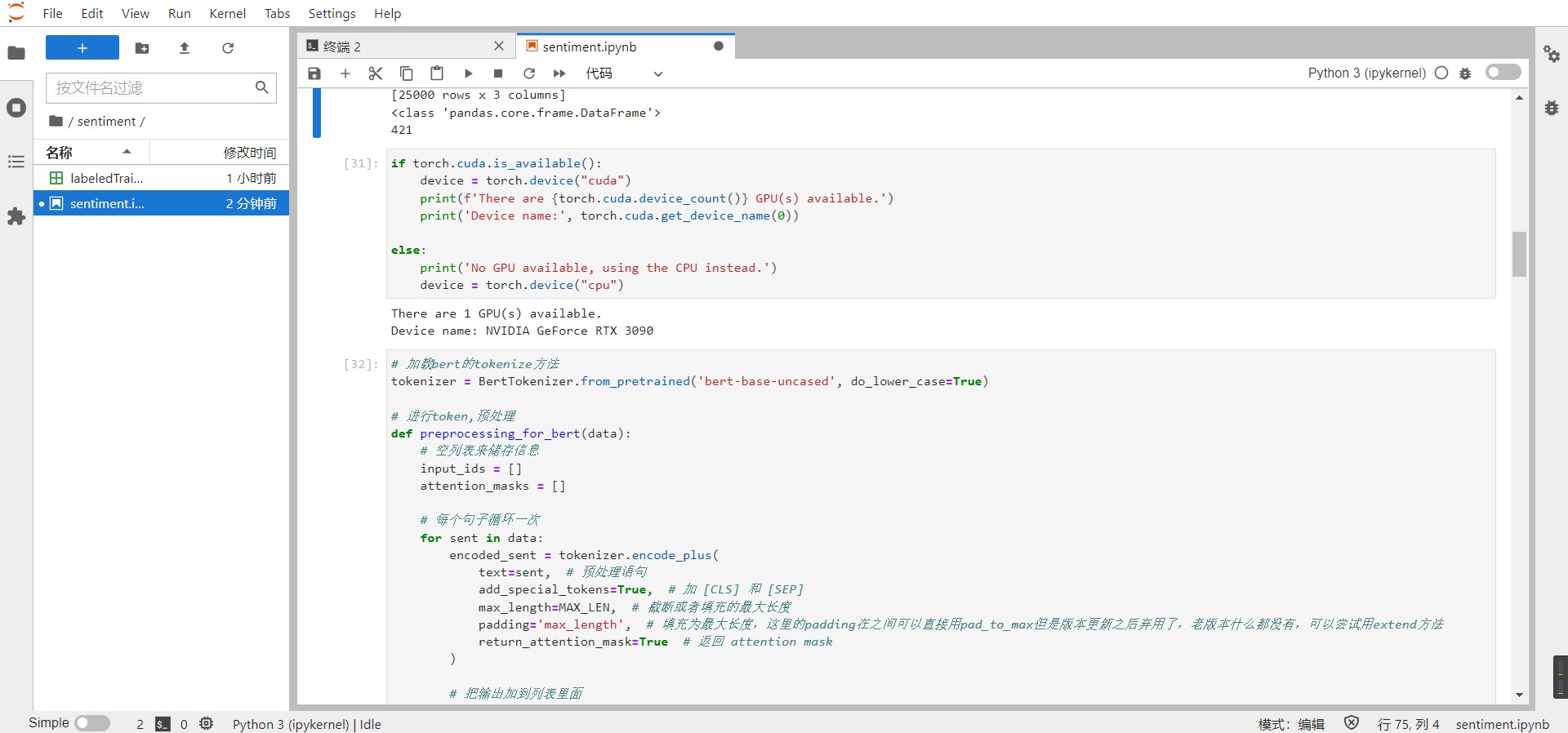


图 3. 3 运行结果

**3.3 数据预处理**

1. *# 载入bert的tokenize方法*
2. tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased', do\_lower\_case=True)
3. *# 数据预处理，将数据转换为tensor*
4. def preprocess\_data(data):
5. *# 空列表来储存信息*
6. input\_ids = []
7. attention\_masks = []
8. *# 每个句子循环一次*
9. for sent in data:
10. encoded\_sent = tokenizer.encode\_plus(
11. text=sent,
12. add\_special\_tokens=True,  *# 加入特殊字符[CLS] 和 [SEP]*
13. max\_length=MAX\_LEN,  *# 截断的最大长度*
14. padding='max\_length',  *# 填充为最大长度*
15. return\_attention\_mask=True  *# 返回 attention mask*
16. )
17. *# 把输出加到list里面*
18. input\_ids.append(encoded\_sent.get('input\_ids'))
19. attention\_masks.append(encoded\_sent.get('attention\_mask'))
20. *#转换为tensor*
21. input\_ids = torch.tensor(input\_ids)
22. attention\_masks = torch.tensor(attention\_masks)
23. return input\_ids, attention\_masks
24. *# Encode 准备好的数据*
25. encoded\_comment = [tokenizer.encode(sent, add\_special\_tokens=True) for sent in data.review.values]
26. *#文本的最大长度*
27. MAX\_LEN = max([len(sent) for sent in encoded\_comment])
28. *# print(MAX\_LEN) #MAX\_len = 485*
29. *# 在训练集和测试集上运行 preprocess\_data 转化为指定输入格式*
30. train\_inputs, train\_masks = preprocess\_data(X\_train)
31. test\_inputs, test\_masks = preprocess\_data(X\_test)
32. *# 转化为tensor类型*
33. train\_labels = torch.tensor(Y\_train)
34. test\_labels = torch.tensor(Y\_test)
35. *# 超参数用于微调, 个人建议batch size 16 or 32较好.*
36. batch\_size = 16
37. *# 为训练集建立 DataLoader*
38. train\_data = TensorDataset(train\_inputs, train\_masks, train\_labels)
39. train\_sampler = RandomSampler(train\_data)
40. train\_dataloader = DataLoader(train\_data, sampler=train\_sampler, batch\_size=batch\_size)
41. *# print(train\_dataloader)*
42. *# 给测试集建立 DataLoader*
43. test\_data = TensorDataset(test\_inputs, test\_masks, test\_labels)
44. test\_sampler = SequentialSampler(test\_data)
45. test\_dataloader = DataLoader(test\_data, sampler=test\_sampler, batch\_size=batch\_size)
46. print("The data is ready!")

运行结果：

该部分代码对数据进行预处理，处理成tensor形式后将测试集和数据集封装成train\_dataloader和test\_dataloader作为模型的输入，同时在这部分设置超参数batch\_size为16。

图3.4所示为数据预处理成功以后输出提示。

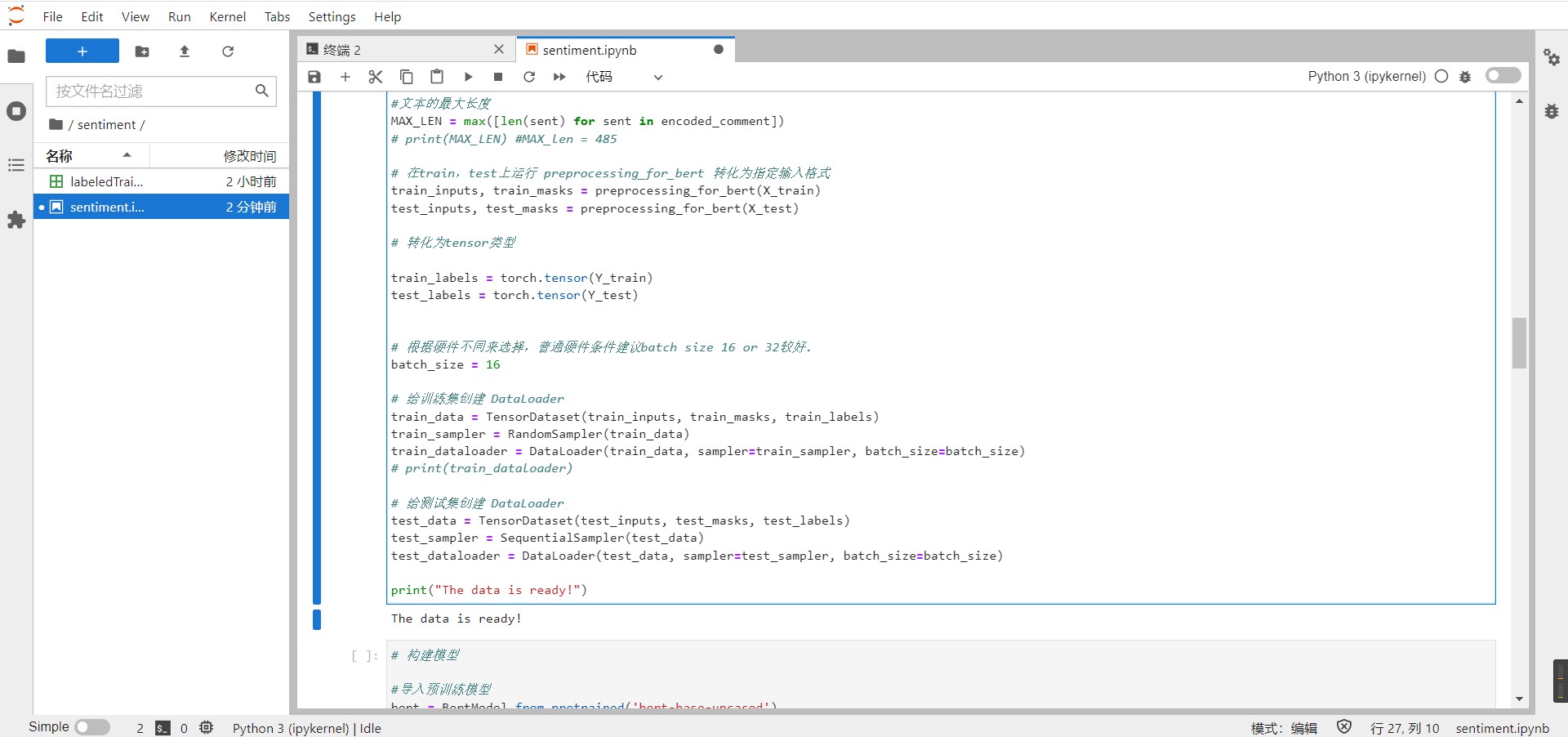


图 3. 4 数据预处理成功以后输出提示

**3.4 数据预处理**

1. *#构建分类模型*
2. *#首先导入bert预训练模型*
3. bert = BertModel.from\_pretrained('bert-base-uncased')
4. *#微调Bert模型，在预训练模型之后加上两个全连接层构成一个分类器，实现分类功能*
5. class BertClassifier(nn.Module):
6. def \_\_init\_\_(self):
7. super(BertClassifier, self).\_\_init\_\_()
8. *# hidden size of Bert默认768，分类器隐藏维度，输出维度为2(因为最后的评论分为积极消极两类)*
9. dim\_input, classify\_hidden, dim\_output = 768, 100, 2
10. *# 实体化Bert模型*
11. self.bert = BertModel.from\_pretrained('bert-base-uncased')
12. *# 分类器，使用ReLU激活函数和两层全连接层*
13. self.classifier = nn.Sequential(
14. nn.Linear(dim\_input, classify\_hidden),  *# 全连接*
15. nn.ReLU(),  *# 激活函数*
16. nn.Linear(classify\_hidden, dim\_output)  *# 全连接*
17. )
18. def forward(self, input\_ids, attention\_mask):
19. *# 搭建网络*
20. *# 输入*
21. outputs = self.bert(input\_ids=input\_ids,
22. attention\_mask=attention\_mask)
23. *#提取[CLS]的最后一层的输出*
24. last\_hidden\_state\_cls = outputs[0][:, 0, :]
25. *# 全连接，计算然后分类，最后输出label*
26. logits = self.classifier(last\_hidden\_state\_cls)
27. return logits
28. """
29. 初始化我们的模型，包括优化器、学习率，设置epochs
30. """
31. def initialize\_model(epochs=2):
32. *# 初始化分类器*
33. bert\_classifier = BertClassifier()
34. *# 使用device进行运算，GPU or CPU*
35. bert\_classifier.to(device)
36. *# 优化器，使用AdamW，设置学习率和eps*
37. optimizer = AdamW(bert\_classifier.parameters(),
38. lr=5e-5,  *# 默认学习率*
39. eps=1e-8
40. )
41. *# 训练的总步数*
42. total\_steps = len(train\_dataloader) \* epochs
43. *# 学习率预热*
44. scheduler = get\_linear\_schedule\_with\_warmup(optimizer,
45. num\_warmup\_steps=0,
46. num\_training\_steps=total\_steps)
47. return bert\_classifier, optimizer, scheduler
48. *# 实例化损失函数，因为是分类任务，所以使用交叉熵损失函数比较好*
49. loss\_fn = nn.CrossEntropyLoss()  *# 交叉熵*

**3.5 训练函数和评估函数**

1. *#训练函数*
2. def train(model, train\_dataloader, test\_dataloader=None, epochs=2, evaluation=False):
3. *#开始训练*
4. for epoch\_i in range(epochs):
5. print(f"{'Epoch':^7} | {'每40个Batch':^9} | {'训练集 Loss':^12} | {'测试集 Loss':^10} | {'测试集准确率':^9} | {'时间':^9}")
6. print("-" \* 80)
7. *# 计算每个epoch消耗的时间*
8. t0\_epoch, t0\_batch = time.time(), time.time()
9. *# 每一个epoch都需要重置变量 total\_loss, batch\_loss, batch\_counts*
10. total\_loss, batch\_loss, batch\_counts = 0, 0, 0
11. *# 把model设置为训练模式*
12. model.train()
13. *# 分batch训练*
14. for step, batch in enumerate(train\_dataloader):
15. batch\_counts += 1
16. *# 把batch加载到GPU*
17. b\_input\_ids, b\_attn\_mask, b\_labels = tuple(t.to(device) for t in batch)
18. *# 梯度归零*
19. model.zero\_grad()
20. *# 训练*
21. logits = model(b\_input\_ids, b\_attn\_mask)
22. *# 损失计算并累加*
23. loss = loss\_fn(logits, b\_labels)
24. batch\_loss += loss.item()
25. total\_loss += loss.item()
26. *# 反向传播*
27. loss.backward()
28. *# 归一化，防止梯度爆炸*
29. torch.nn.utils.clip\_grad\_norm\_(model.parameters(), 1.0)
30. *# 更新参数和学习率*
31. optimizer.step()
32. scheduler.step()
33. *# 输出每40个batch的消耗时间和计算损失*
34. if (step % 40 == 0 and step != 0) or (step == len(train\_dataloader) - 1):
35. *# 开始计时*
36. time\_elapsed = time.time() - t0\_batch
37. *# 输出结果*
38. print(
39. f"{epoch\_i + 1:^7} | {step:^10} | {batch\_loss / batch\_counts:^14.6f} | {'-':^12} | {'-':^13} | {time\_elapsed:^9.2f}")
40. *# 重置batch参数*
41. batch\_loss, batch\_counts = 0, 0
42. t0\_batch = time.time()
43. *# 计算平均loss：训练集的平均loss*
44. avg\_train\_loss = total\_loss / len(train\_dataloader)
45. print("-" \* 80)
46. *# 可以通过控制evalution是否为TRUE，决定是否需要汇总评估*
47. if evaluation:
48. *# 每个epoch之后评估一下性能*
50. test\_loss, test\_accuracy = evaluate(model, test\_dataloader)
51. *# 输出整个训练集上的时间消耗情况*
52. time\_elapsed = time.time() - t0\_epoch
53. print(
54. f"{epoch\_i + 1:^7} | {'-':^10} | {avg\_train\_loss:^14.6f} | {test\_loss:^12.6f} | {test\_accuracy:^12.2f}% | {time\_elapsed:^9.2f}")
55. print("-" \* 80)
56. print("\n")
57. *# 评价函数：在测试集上观察模型的效果*
58. def evaluate(model, test\_dataloader):
59. *# 把model设置为评估模式*
60. model.eval()
61. *# 测试集上准确率和误差*
62. test\_accuracy = []
63. test\_loss = []
64. *# 测试集上的每个batch*
65. for batch in test\_dataloader:
66. *# 放到GPU上*
67. b\_input\_ids, b\_attn\_mask, b\_labels = tuple(t.to(device) for t in batch)
68. *# 计算结果（不更新梯度）*
69. with torch.no\_grad():
70. logits = model(b\_input\_ids, b\_attn\_mask)  *# 放入模型中运行*
71. *# 计算误差*
72. loss = loss\_fn(logits, b\_labels.long())
73. test\_loss.append(loss.item())
74. *# 得到预测结果，返回一行中最大值的序号*
75. preds = torch.argmax(logits, dim=1).flatten()
76. *# 计算准确率，预测正确的个数/测试集总的样本数*
77. accuracy = (preds == b\_labels).cpu().numpy().mean() \* 100
78. test\_accuracy.append(accuracy)
79. *# 计算整体的平均正确率和loss*
80. val\_loss = np.mean(test\_loss)
81. val\_accuracy = np.mean(test\_accuracy)
82. return val\_loss, val\_accuracy

**3.6 训练模型**

1. print("服务器准备就绪")
2. bert\_classifier, optimizer, scheduler = initialize\_model(epochs=6)
3. print("开始训练:\n")
4. train(bert\_classifier, train\_dataloader, test\_dataloader, epochs=6, evaluation=True)
5. print("训练完成！")

运行结果：如图3.5所示为模型成功运行截图

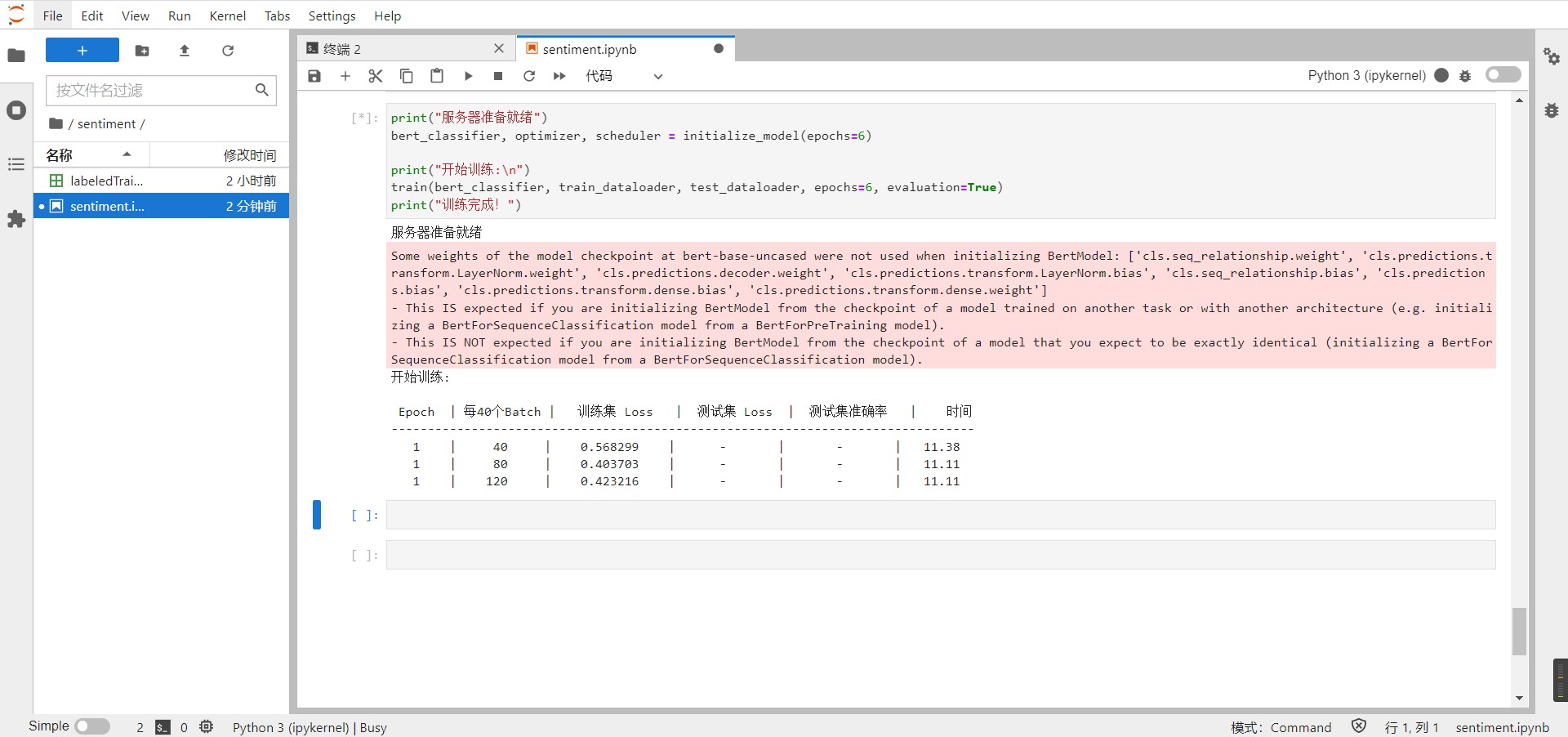


图 3. 5 模型成功运行截图

红底框中出现的信息表示该模型的部分参数并没有用到，说明实现的任务与该预训练模型不完全对应，主要是BERT模型的参数量太大，但是并不影响模型的正常运行，同时也说明有着很大的进步空间，倘若与预训练模型完全对应，所有的参数都使用到了，预测的准确度将会有新的提升。

1. **训练曲线（Loss-Epoch）**

如图4.1所示为每个batch的train\_loss曲线图(batch\_size = 16)。图中只记录了三个epoch的训练曲线，在6个epoch训练结束以后，训练集的每个batch的平均损失为：0.013438，测试集准确率为：92.45%。

图 4. 1 每个batch的train\_loss曲线图。(batch\_size = 16)

1. **运行或运行结果截图**

运行结果如图5.1所示。

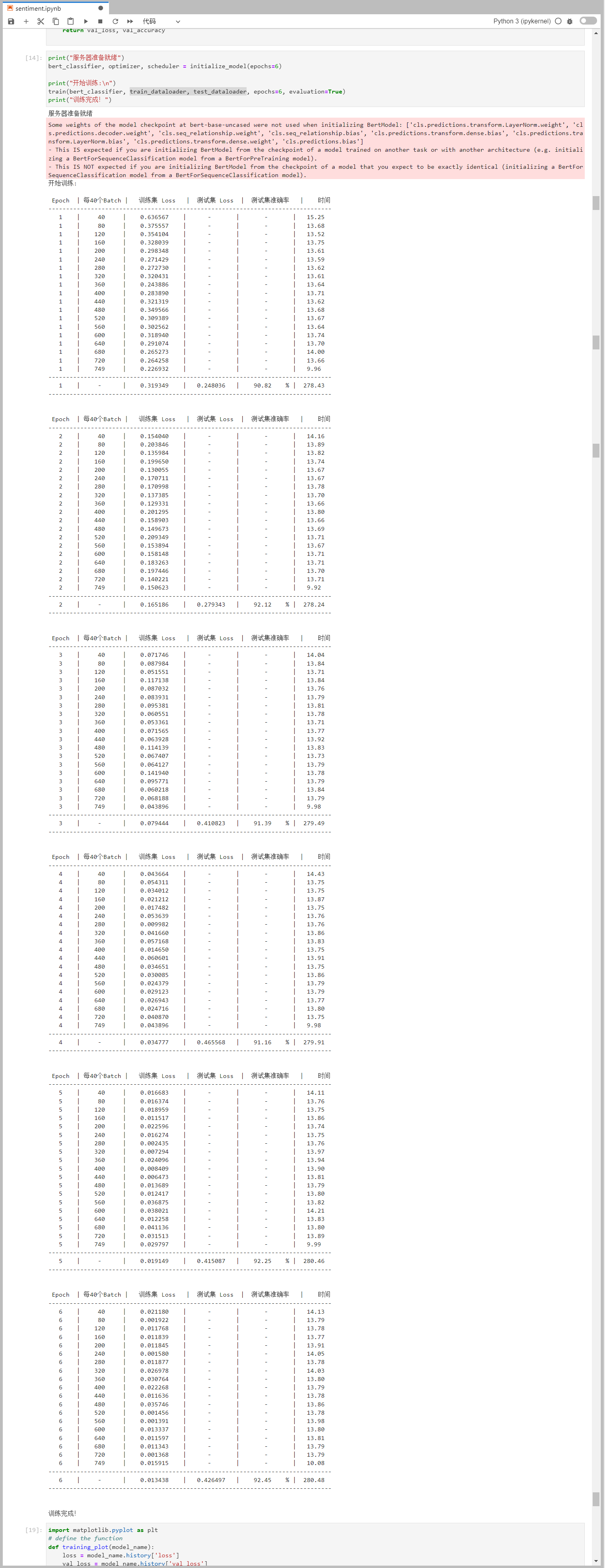


图 5. 1 运行结果

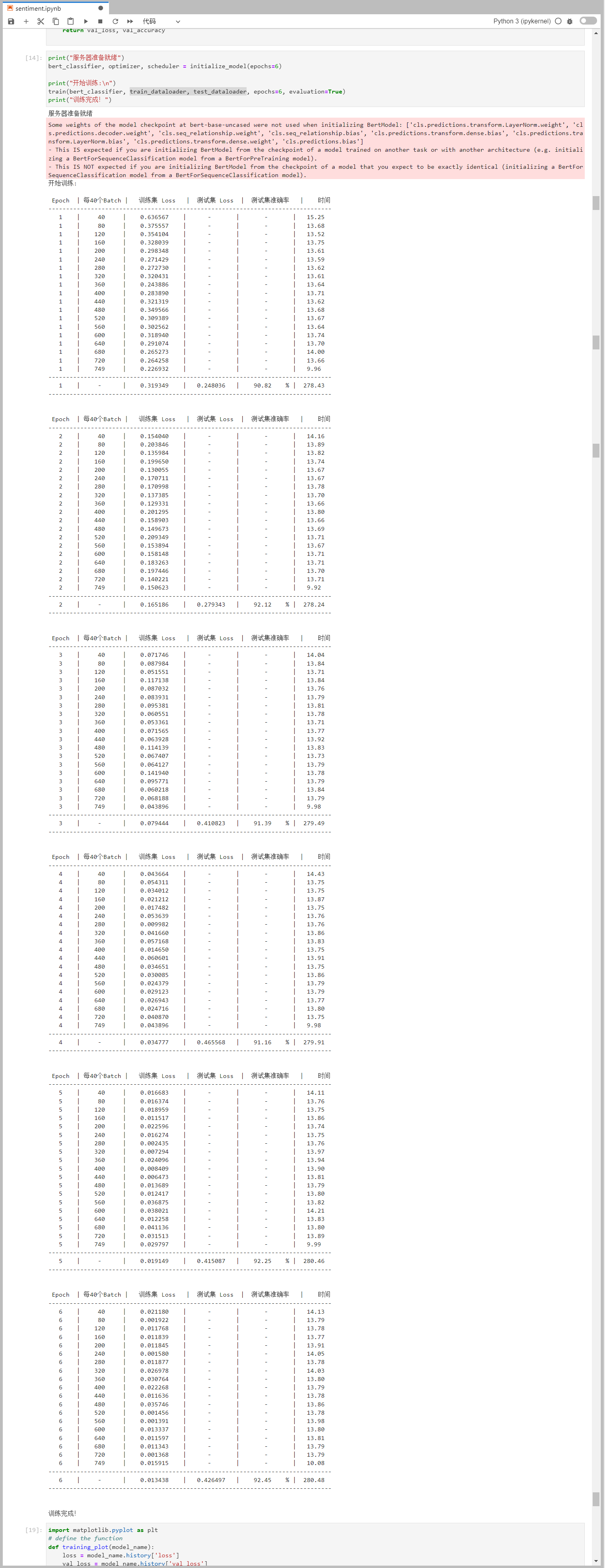


图 5. 1 运行结果