****

《软件工程实践》暑期实践

题 目：\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 学号 | 姓名 | 专业班级 | 工作占比 | 个人成绩 |
| 2008010328 | 张明劼 | 软件2003 | 100% |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 评阅项目 | 评阅内容 | 百分比 | 得分 |
| 问题难度 | 对要解决问题的难度进行分类 | 20% |  |
| 实习结果 | 对最终的实现过程和结果进行评价 | 50% |  |
| 能力水平 | 应用理论分析问题、解决问题的能力； | 10% |  |
| 文字表述 | 要求报告文字表述正确、清楚； | 10% |  |
| 平时表现 | 实习过程中的听课、讨论、手册等内容 | 10% |  |
| 实习报告得分 | | |  |
| 评阅人： | | | |

2023年7月19日

目 录

[第1章 序言 1](#_Toc140654142)

[第2章 相关工作综述 2](#_Toc140654143)

[2.1 主题分类 2](#_Toc140654144)

[2.2 情感分类 2](#_Toc140654145)

[第3章 实习内容 4](#_Toc140654146)

[3.1 docker基础 4](#_Toc140654147)

[3.2 Huggingface 18](#_Toc140654148)

[3.3 情感分类流程 30](#_Toc140654149)

[3.4 Setfit模型 39](#_Toc140654150)

[第4章 总结与收获 44](#_Toc140654151)

# 第1章 序言

在这份实习报告的序言中，我将简要介绍我在实习期间主要从事的工作内容，即主题分类和情感分类。

在实习期间，我有幸参与了一个关于自然语言处理领域的重要任务，即NER主题分类和情感分类。NER是一项用于从文本中提取出具有特定意义的命名实体的任务，例如人名、地名、组织机构等。而主题分类和情感分类则是对文本进行进一步分析和理解的任务。

通过对命名实体进行识别和分类，我们可以更好地理解文本所含信息，并为相关应用如智能搜索、舆情分析、信息抽取等提供支持。NER主题分类的目标是按照事先定义好的一系列主题对命名实体进行分类，使我们能够快速定位文本中涉及的特定主题。而情感分类任务则着重分析文本中所表达的情感倾向，有助于我们理解人们对某个主题或事件的态度和情感观点。

在实习期间，我参与了数据的收集和整理工作，学习并运用了各种主题分类算法和模型，如基于机器学习的方法和深度学习模型。通过对大量文本数据的处理和训练，我不断提高了主题分类和情感分类的准确性和效率。

实习过程中，我还积极参与团队讨论和交流，与同事们共同解决问题、分享经验，不断改进和优化我们的模型和算法。这个过程让我更加深入地理解了主题分类和情感分类的挑战与机遇，并培养了团队合作和沟通能力。

总结而言，这次实习给予了我宝贵的机会去探索和应用主题分类和情感分类技术。我在技术和团队协作方面都有所收获，并相信这些经验将对我的职业发展起到积极的推动作用。

在这篇报告中，第2章介绍了本文的相关研究工作，包括常见的主题分类以及情感分类算法。第3章介绍了这几周在企业的学习收获；第4章是对本文的总结和工作展望。

# 第2章 相关工作综述

## 2.1 主题分类

主题分类是一种文本分类任务，旨在将给定的文本按照其所属的主题或话题进行分类。它涉及对文本内容的分析和理解，以确定其所涉及的主要概念、主题或领域。

主题分类在许多领域都有着广泛的应用。在信息检索和搜索引擎中，主题分类可以帮助用户更准确地获取与其兴趣相关的信息。在舆情分析和社交媒体监测中，主题分类能够帮助了解公众对某个事件、产品或服务的态度和观点。在文本挖掘和知识管理中，主题分类可用于组织和归档大量的文本数据，使之更容易被搜索和浏览。

主题分类的实现可以使用各种方法和技术。传统的基于机器学习的方法包括朴素贝叶斯分类器、支持向量机和决策树等。这些方法通常依赖于手动设计的特征，并在训练集上进行监督学习。然而，随着深度学习的发展，神经网络模型如卷积神经网络（CNN）和循环神经网络（RNN）也被广泛应用于主题分类任务。这些模型能够自动学习特征表示，并在大规模数据上进行端到端的训练，具有较好的分类性能。

主题分类任务也面临一些挑战。首先，语言的多样性和表达方式的灵活性使得文本中的主题分类具有一定的主观性和不确定性。此外，文本中可能存在歧义、多义词和语义变化等问题，需要针对特定领域和语境进行解决。此外，缺乏标注数据和领域知识也是主题分类中常见的挑战之一。

总体而言，主题分类是一项重要的自然语言处理任务，对于有效组织和理解文本内容至关重要。通过不断改进算法和技术，并结合领域专业知识，我们可以提高主题分类的准确性和效率，以满足各种实际应用的需求。

## 2.2 情感分类

情感分类是一种自然语言处理任务，旨在对文本进行情感倾向的分类，即判断文本所表达的情感是积极、消极还是中立。情感分类的目标是通过分析文本的情感表达、情绪和语义信息来理解其中蕴含的情感色彩和情感观点。

情感分类广泛应用于许多领域。在社交媒体监测中，情感分类可以帮助企业和品牌了解用户对产品、服务或事件的看法和反馈。在舆情分析中，情感分类可用于快速评估公众对特定话题、政策或事件的态度。在市场调研和用户体验评估中，情感分类可以帮助分析人员了解用户对产品或服务的满意度和喜好。

情感分类的实现方法有很多种。传统的基于规则和词典的方法涉及使用预定义的情感词汇和规则对文本进行情感分类。这些方法相对简单直观，但通常难以处理上下文和语义的复杂性。近年来，深度学习模型如卷积神经网络（CNN）、循环神经网络（RNN）和注意力机制等也被广泛应用于情感分类任务。这些模型能够自动学习文本中的情感特征和上下文关系，从而提高分类性能。

然而，情感分类任务也面临一些挑战。首先，情感是主观的，不同人对相同文本可能有不同的情感倾向。其次，语义的复杂性和多义性使得情感分类具有一定的歧义和难度。此外，缺乏标注数据和领域知识也可能限制情感分类的准确性和适应性。

总体而言，情感分类是一个重要的自然语言处理任务，对于理解人们对特定主题、事件或产品的情感态度至关重要。通过不断改进算法和模型，并结合领域知识和背景信息，可以提高情感分类的效果和鲁棒性，以满足各种实际应用的需求。

# 第3章 实习内容

## 3.1 docker基础

参考文档：

<https://docker.easydoc.net/doc/81170005/cCewZWoN/lTKfePfP>

<https://www.notion.so/Pipeline-0843ccfce01d4804946cc6f04af4b32c#c11187a143664dc0a30f1b27d826d642>

简介

将开发的应用程序自动部署的开源引擎

在虚拟化的容器执行环境中增加了一个应用程序部署引擎。提供一个轻量、快速的环境运行开发者的程序，方便高效的将程序部署到测试环境与生产环境。

作用

打包、分发、部署

打包：就是把你软件运行所需的依赖、第三方库、软件打包到一起，变成一个安装包

分发：你可以把你打包好的“安装包”上传到一个镜像仓库，其他人可以非常方便的获取和安装

部署：拿着“安装包”就可以一个命令运行起来你的应用，自动模拟出一摸一样的运行环境，不管是在 Windows/Mac/Linux。

方便部署环境

简单、轻量化：

几分钟就可以将程序Docker化

取出管理程序开销，Docker高性能，一台宿主机可以运行更多的容器。

逻辑分离

开发人员只关心容器中运行的程序，运维只关心如何管理容器

保证开发人员代码的各种环境下一致

快速、高效的开发生命周期

缩短代码开发、测试到部署、上线运行的周期。

程序具有可移植性、易于构建，并易于协作。

优势：

常规开发部署方式：自己在windows开发🡪测试 到Linux服务器配置运行环境

问题：服务器上跑有问题

Dokcer：确保不同机器上跑的都是一致的运行环境。

优点：

方便分发、部署，方便传播给其他人的安装。

快速安装测试、学习软件，用完就丢，不把时间安排在安装软件上。

多版本软件共存，不污染系统

重要概念

镜像：软件安装包，方便进行传播、安装

容器：软件安装后的状态，每个软件的运行环境都是独立、隔离的

核心组件

Docker客户端与服务器

Docker是一个CS架构，客户端只需向Docker服务器或守护进程发出请求，服务器或守护进程将完成所有的工作并返回结果。

图示

描述已自动生成

Docker镜像

镜像是构建Docker的基石，用户基于镜像来运行自己的容器。

镜像可视为容器的“源代码”。镜像体积小，易于分享、存储以及更新

Registry

Docker用Registry来保存用户构建的镜像。分为公有和私有两种，Docker公司运营的公共Registry叫做Docker Hub。从上面可以下载到开源的镜像。也可以自己上传保存自己的镜像。

也可以在Docker Hub上保存自己的私有镜像。

也可以架设自己的私有Registry。

Docker容器

Docker可以帮助构建和部署容器，只需要将自己的程序或服务打包至容器即可。容器是基于镜像启动起来的，容器可以运行一个或者多个进程。我们可以认为，镜像是Docker生命周期中的构建或者打包阶段，容器则是启动和执行阶段。总结：

Docker=一个镜像格式+一系列标准的操作+一个执行环境

标准集装箱概念：将货物运世界各地。

Docker将货物换为软件。每个容器包含一个软件镜像，容器里的软件镜像可以进行一些操作：创建、启动、关闭、重启、销毁。

Docker在进行上述操作时，不关心容器里到底装了什么，所有的容器都按照相同的方式将内容“装载”。

Docker技术组件

可以运行于任何安装了现代Linux系统内核的X64主机，推荐的内核版本是3.8或者更高。Docker开销较低，可用于台式、服务器、笔记本。包含几个部分

一个原生的Linux容器格式，Docker中成为libcontainer，其是目前Docker容器的默认格式。

Linux内核命名控件(namespace)，用于隔离文件系统、内核和网络

文件系统隔离：每个容器有自己的root文件系统

进程隔离：每个容器运行在自己的进程环境中

网络隔离：容器间的虚拟网络接口和IP地址都是分开的

资源隔离和分组：使用cgroups（即control group, Linux 的内核特性之一）将CPU和内存之类的资源独立分配给每个Docker容器

写时复制：文件系统都是通过写时复制创建的，意味着文件系统是分层的、快速的，并且占用控件磁盘更小。

日志：容器产生的STDOUT、STDERR和STDIN这些IO流都会收集并计入日志，用来进行日志分析和故障排除。（标准输出，标准输入和标准错误）

交互式的shell：用户可以创建一个伪tty终端，将其连接到STDIN,为容器提供一个交互式的shell。

写时复制（Copy-on-write，简称COW）是一种计算机程序设计领域的优化策略。其核心思想是，如果有多个调用者（callers）同时请求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。这过程对其他的调用者都是透明的。此作法主要的优点是如果调用者没有修改该资源，就不会有副本（private copy）被创建，因此多个调用者只是读取操作时可以共享同一份资源。

Docker配置

Docker操作

首先创建一个docker

sudo docker run

--gpus all

-it

-u $(id -u):$(id -g)

-p 99:22

-p 999:6667

-p 9999:6666

-v /home/zmj:/workspace/zmj

-v /opt/src:/workspace/src

--name=zmj\_docker

--ipc=host

--ulimit memlock=-1

--ulimit stack=67108864

--restart=always d3a03924e49d

具体后续在run命令中进行

进入docker



docker是一个完整的ubuntu操作系统，可以获取其主机名，查看/etc/hosts，和ubuntu版本等操作

文本

描述已自动生成

使用exit退出docker。

Docker ps

用于查看Docker容器

docker ps 只会显示运行中的容器， -a 参数可以显示所有容器

docker ps --help # 查看命令的具体参数，其他的命令雷同

图形用户界面, 文本

描述已自动生成

容器命名

使用-name

一 个合法的容器名称只能 包含以下字符:小写字母a~z、大写字母A~z、数字0~9、下划线、圆点、横线。

在很多容器命令中，我们都可以使用容器的名称替代容器 ID

容器的命名必须是唯一的



Docker start

exit退出后，发现docker已经关闭

这时使用docker start 进行启动

图片包含 文本

描述已自动生成

Docker attach

docker直接start的话不会出现交互式命令行，可使用attach去重新附在容器的会话上



守护式容器

daemonized container 没有交互式会话，非常适合运行应用程序以及服务。大部分时候以守护式容器来运行容器

在下图中，使用-d创建守护式容器

-c 参数，shell 命令 sh 的参数，表示从 string 中读取命令

会1s打印一个hello world 直到容器或其进程停止工作。

屏幕上有字

描述已自动生成

容器内部在干什么

docker-logs获取容器的日志

图片包含 文本

描述已自动生成

可使用-f参数控制输出

# 一些其他参数

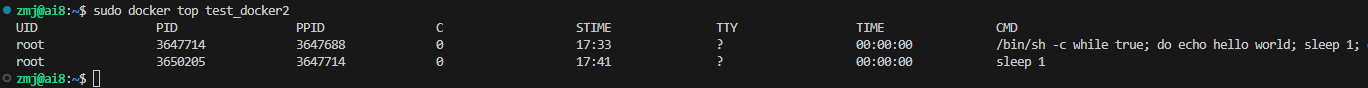
docker logs --tail 10 my\_daemon # 获取日志最后10行

docker logs --tail 0 -f my\_daemon # 跟踪某个容器最新日志不必读取整个日志

docker logs --tail 0 -ft my\_daemon # 参数 -t 为日志加上时间戳

获取docker内部运行的进程

sudo docker top



在容器内部运行进程

docker exec 在容器内部额外启动新进程，可以在容器内运行的进程有两个类型：

后台任务（无交互）



交互任务

-t -i 标志容器为我们执行的进程创建了TTY并捕捉了STDIN。



停止容器



创建自动重启Docker

将restart 设为always,若由于某种错误使容器自动停止运行，可自动重启

我们还可以将 --restart 的参数设置为 on-failure，这样容器只会在退出代码为 非 0 时重启。

当然还可以设置重启次数 --restart=on-failure:5

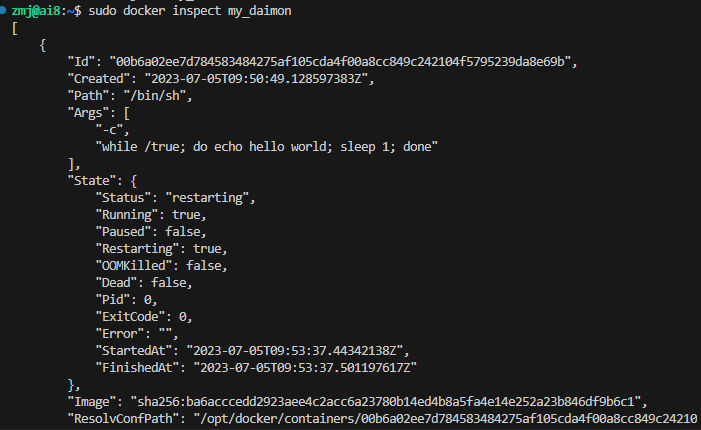
文本

描述已自动生成

深入容器

除了通过 docker ps 命令获取信息，还可以通过 docker inspect 来获得更多的容器信息。

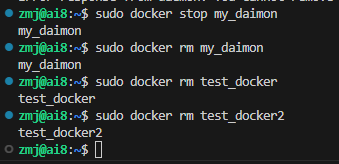
包括名称、命令、网络配置以及很多有用的数据。



# 也可以使用 -f 参数来选定查看结果 sudo docker inspect --format='{{ .State.Runing }}' my\_daemon

删除容器

sudo docker rm



容器进程必须停止后再删除。

Docker run

运行一个docker

基本参数

-it: 以交互模式运行容器；

--gpus all：使用GPU（docker19.03及以上版本，安装nvidia-container-toolkit）

or

--gpu '"device=1"' 使用本机的第1块gpu

--name="pytorch\_docker": 为容器指定一个名称；

--ipc=host：docker环境使用的命名空间 host共享宿主机的命名空间；private 在内存中专门指定当前docker的命名空间

-v /home/local\_path:/home/docker\_path ## 挂载宿主机目录到容器中； 通过docker环境访问主机的目录，即docker与主机文件交互的目录

-p 8080:8080， 指定容器暴露的端口；主机端口:docker 端口，一般主机端口可以随意更改，docker端口不要更改

--rm 退出时自动删除容器

--ulimit memlock=-1 --ulimit stack=67108864

--ulimit限制docker进程对资源的使用

--restart=always 容器挂掉时总是重启

Docker镜像以及仓库

什么是docker镜像

docker镜像是由文件系统堆叠而成的，最底端是一个引导文件系统，即bootfs，很典型的Linux/Unix引导文件系统。但Docker用户几乎永远不会与其交互。实际上，当容器启动时，他会被移入内存中，而引导文件系统则会被卸载（unmount），流出更多内存供initrd磁盘镜像使用。

Docker镜像的第二层是root文件系统rootfs，它位于引导文件系统之上，rootfs可以使一种或多种操作系统（比如Debian或者Ubuntu系统）。

在传统的Linux引导过程中，root文件系统会最先以只读的方式加载，引导结束后并完成完整性检查后，他才会被切换到读写模式。但是在Docker中，root的文件系统永远只读。而且Docker利用联合加载技术又在root文件系统上加载更多的只读文件系统。（联合加载：一次性同时加载多个文件系统，但是在外部看起来只有一个文件系统。）

Docker将这样的文件系统称为镜像，一个镜像可以放在另一个镜像的顶部，位于下面的镜像称为父镜像（parent image），可以以此类推，最底部的镜像称为基础镜像（base image）。最后，当一个镜像启动容器时，Docker会在该镜像最顶层加载一个读写文件系统，我们想在Docker中运行的程序就是在这个读写层进行的。

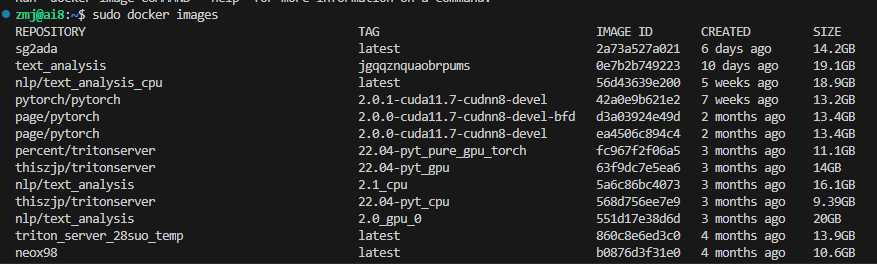
Docker第一次启动一个容器时，初始的读写层是空的。当文件系统发生变化时，这些变化都会用到这一层中。比如，如果想修改一个文件，这个文件首先会从该读写层下面的只读层复制到该读写层。该文件的只读版本依旧存在，不过已经被该读写层的该文件副本所隐藏。

通常这种机制称为写时复制（copy on write）,这是Docker如此强大的技术之一。每个只读镜像层都是只读的，而且以后永远不会变化。当创建一个新容器是，Docker都会构建出一个镜像栈，并在栈的最顶端添加一个读写层。这个读写层再加上其下面的镜像层以及一些配置数据，就构成了一个容器。

图示

描述已自动生成

sudo docker images查看可用镜像



本地镜像都保存在宿主机 /var/lib/docker 目录下。每个镜像都保存在 Docker 所采用的存储驱动目录下面，如 aufs 或者 devicemapper。

镜像是从仓库下载而来，而仓库存在于Registry中，默认的Registry是由Docker公司运营的公共Registry服务，即DockerHub。可以把仓库想象为类似Git仓库，其包括镜像、层以及关于镜像的元数据。

每个镜像仓库可以存放多个镜像，比如pytorch仓库包含了很多版本。

为区分统一仓库的不同镜像，使用TAG标签方便区分。

从镜像 pytorch/pytorch:1.13.0-cuda11.6-cudnn8-devel 启动一个容器

电脑屏幕截图

描述已自动生成

在冒号后，指定容器的标签。

Docker Hub有两种仓库，一种是用户仓库和顶层仓库。用户仓库的镜像都是由Docker用户创建，而顶层仓库则是由Docker内部人员来管理。

用户仓库的命名和仓库名由两部分组成：用户名/仓库名。

顶层仓库只包含仓库名部分，比如ubuntu仓库，顶层仓库由Docker公司和选定的能提供优质基础镜像的厂商管理。用户可以基于这些基础镜像构建自己的镜像。

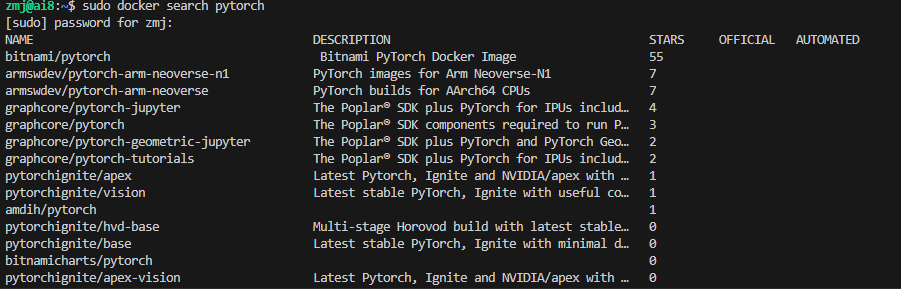
拉取镜像

使用docker run命令启动镜像时，若容器不在本地，Docker会优先从Docker Hub下载镜像。如果没有指定具体的镜像标签，那么Docker会自动下载latest标签的镜像。

也可使用docker pull 命令自己预先拉取镜像到本地。使用docker pull命令可以节省一个新镜像启动一个容器所需要的时间。

查找镜像

可以通过docker search 查找所有Docker Hub上公共的可用镜像。



构建镜像

如果你想要使用容器，步骤应该如下

1. docker search 查找可用的镜像

2. docker pull 下载到本地

3. docker run 开启一个容器

创建Docker Hub账号

注册

邮箱激活

测试是否运行正常

docker login

使用Docker commit 命令创建镜像

首先基于官方ubuntu镜像创建一个新容器

sudo docker run -it ubuntu /bin/bash

在容器中安装Apache

# 网络问题应该是无法安装成功的，这里只是给出一个流程

apt-get -yqq update

apt-get -y install apache2

我们想将这个容器作为一个 web 服务器使用，所以我们想把它的工作状态保存下来。这样不必每次都创建一个新容器并在此在里面安装 Apache了。

# 退出容器 exit

# 执行 docker commit 命令。可以通过 docker ps -l -q 命令获得刚创建的容器 ID

sudo docker commit docker\_id username/docker\_name

# 当然与 git 提交类似，你可以通过参数添加其他的信息 # -m 提交信息 # --author 作者信息。

用Dockerfile构建镜像

使用Dockerfile构建镜像是比较推荐的方式。接下来展示一个基础示例与一个基于pytorch构建的Dockerfile例子。

Dockerfile基于基本的DSL语法指令构建一个Docker镜像。之后使用docker build 命令基于该Dockerfile的指令构建新的镜像

首先创建一个目录并在里面创建初始的Dockerfile。我们将创建一个包含简单web服务器的Docker镜像。

文本

描述已自动生成

创建了一个static\_web的目录来保存Dockerfile，这个目录就是我们的构建环境。Docker称这个环境为上下文（）

## 3.2 Huggingface

Haggingface

Huggingface是什么

Hugging Face Hub和 Github 类似，都是Hub(社区)。Hugging Face可以说的上是机器学习界的Github。Hugging Face为用户提供了以下主要功能：

模型仓库（Model Repository）：Git仓库可以让你管理代码版本、开源代码。而模型仓库可以让你管理模型版本、开源模型等。使用方式与Github类似。

模型（Models）：Hugging Face为不同的机器学习任务提供了许多预训练好的机器学习模型供大家使用，这些模型就存储在模型仓库中。

数据集（Dataset）：Hugging Face上有许多公开数据集。

基本介绍

hugging face在NLP领域最出名，其提供的模型大多都是基于Transformer的。为了易用性，Hugging Face还为用户提供了以下几个项目：

Transformers(github, 官方文档): Transformers提供了上千个预训练好的模型可以用于不同的任务，例如文本领域、音频领域和CV领域。该项目是HuggingFace的核心，可以说学习HuggingFace就是在学习该项目如何使用。

Datasets(github, 官方文档): 一个轻量级的数据集框架，主要有两个功能：①一行代码下载和预处理常用的公开数据集； ② 快速、易用的数据预处理类库。

Accelerate(github, 官方文档): 帮助Pytorch用户很方便的实现 multi-GPU/TPU/fp16。

Space：Space提供了许多好玩的深度学习应用，可以尝试玩一下。

NLP简介

NLP 是语言学和机器学习领域，专注于理解与人类语言相关的一切。 NLP 任务的目标不仅是单独理解单个单词，而且能够理解这些单词的上下文。

常见NLP任务：

分类：获取评论的情绪，检测电子邮件是否是垃圾邮件，确定句子语法是否正确或两个句子在逻辑上是否相关

对每个单词进行分类：识别句子的语法成分（名词、动词、形容词）或命名实体（人、地点、组织）

生成文本内容：用自动生成的文本完成提示，用屏蔽词填充文本中的空白

从文本中提取答案：给定问题和上下文，根据上下文中提供的信息提取问题的答案

从输入文本生成新句子：将文本翻译成另一种语言，总结文本

NLP的难点：

计算机处理信息的方式与人类不同。 例如，当我们读“我饿了”这句话时，我们可以很容易地理解它的含义。 同样，给定两个句子，例如“我饿了”和“我很难过”，我们可以轻松确定它们的相似程度。 对于机器学习（ML）模型来说，此类任务更加困难。 需要以某种方式处理文本，使模型能够从中学习。

Huggingface Transformers

直接使用预训练模型进行推理

提供了大量预训练模型可供使用

使用预训练模型进行迁移学习

Transformer 模型用于解决各种 NLP 任务。Transformers 库提供了创建和使用这些共享模型的功能。 模型中心包含数千个预训练模型，任何人都可以下载和使用。 也可以将自己的模型上传到 Hub

Transformers 库中最基本的对象是 pipeline() 函数。 它将模型与其必要的预处理和后处理步骤连接起来，使我们能够直接输入任何文本并获得易于理解的答案

示例

#引入sentiment-analysis 模型

from transformers import pipeline

classifier = pipeline("sentiment-analysis")

classifier("I've been waiting for a HuggingFace course my whole life.")

结果

[{'label': 'POSITIVE', 'score': 0.9598047137260437}]

也可以如此使用

classifier(

["I've been waiting for a HuggingFace course my whole life.", "I hate this so much!"])

结果

[{'label': 'POSITIVE', 'score': 0.9598047137260437},

{'label': 'NEGATIVE', 'score': 0.9994558095932007}]

该pipeline选定并加载一个预训练模型，可以对语句的情感进行分析。加载后的模型会放入缓存，故再次运行不需要重新加载。

刚刚的pipeline中，具体三个步骤：

文本被预处理为模型可以理解的格式。(Tokenzier)

预处理后的输入将传递给模型。

模型的预测是经过后处理的，因此您可以理解它们。

零样本学习（Zero-shot classification）

对尚未进行标记的文本进行分类，这是现实生活中的常见场景，因为注释文本通常非常耗时并需要专业领域知识。对于这个用例，只需要指定标签，因此不必依赖预训练模型的标签。

示例

from transformers import pipeline

classifier = pipeline("zero-shot-classification")

classifier(

"This is a course about the Transformers library",

candidate\_labels=["education", "politics", "business"],

)

结果

{'sequence': 'This is a course about the Transformers library',

'labels': ['education', 'business', 'politics'],

'scores': [0.8445963859558105, 0.111976258456707, 0.043427448719739914]}

除此之外，还可用于Text generation等方面，这里不一一赘述了。

微调

所有 Transformer 模型（GPT、BERT、BART、T5 等）都已被训练为语言模型。 这意味着他们已经以自我监督的方式接受了大量原始文本的培训。

但是这些模型可能对特定的实际任务并不实用，因此可能需要进行迁移学习，模型会针对给定任务以监督方式（使用人工注释的标签）进行微调。

Transformers工作原理

预训练是从头开始训练模型的行为：权重随机初始化，并且在没有任何先验知识的情况下开始训练。

从base model 到Pretrained language model历经极大的消耗，对环境也有极大的污染。

针对已有的预训练模型进行微调，即能提高准确率，也能减少训练模型的时间。

例如，可以利用在英语上训练的预训练模型，然后在 arXiv 语料库上对其进行微调，从而产生基于科学/研究的模型。 微调只需要有限数量的数据：预训练模型获得的知识是“转移 的”，因此称为转移学习。

当然，如果拥有巨大量的数据的情况下，从头训练模型也许会更好。

Transformer 模型结构

分为两部分：

Encoder编码器（左）：编码器接收输入并构建其表示（其特征）。 这意味着模型经过优化以从输入中获取理解。

解码器（右）：解码器使用编码器的表示（特征）以及其他输入来生成目标序列。 这意味着该模型针对生成输出进行了优化。

按照前面的步骤，Encoder接收来自tokenizer得到的序列化结果，并进一步将其转化为计算机能理解的语言（包含特征等） Decoder接收Encoder的结果并进一步输出为我们能够看懂的结果。

使用场景：

Encoder-only models:适合需要理解输入的任务，例如句子分类和命名实体识别。

Decoder-only models: 适合文本生成等生成任务。

Encoder-decoder models or sequence-to-sequence models: 适合需要输入的生成任务，例如翻译或摘要。

一般来说，Encoder可以独立工作，而Decoder则依赖于Encoder提供的上下文语境等信息。

Attention layers

Transformers的核心之一，让模型关注于特定的文字。

举例来说，翻译英语的"I like this course"，模型需要关注"你"才能获得"喜欢"的正确翻译，因为法文的特性。此外，翻译this时，也需要关注course。

综上，同样的概念适用于与自然语言处理的任何任务，单词的含义受上下文的影响，上下文可以使正在研究的单词之前或者之后的任何单词。

Trainner

基本参数：

model: Union[PreTrainedModel, nn.Module] = None,

集成了 transformers.PreTrainedMode 或者torch.nn.module的模型

args: TrainingArguments = None,

超参数的定义，这部分也是trainer的重要功能，大部分训练相关的参数都是这里设置的，非常的方便：

data\_collator: Optional[DataCollator] = None,

train\_dataset: Optional[Dataset] = None,

eval\_dataset: Optional[Union[Dataset, Dict[str, Dataset]]] = None,

tokenizer: Optional[PreTrainedTokenizerBase] = None,

model\_init: Optional[Callable[[], PreTrainedModel]] = None,

compute\_metrics: Optional[Callable[[EvalPrediction], Dict]] = None,

callbacks: Optional[List[TrainerCallback]] = None,

optimizers: Tuple[torch.optim.Optimizer, torch.optim.lr\_scheduler.LambdaLR] = (None, None),

preprocess\_logitsformetrics: Optional[Callable[[torch.Tensor, torch.Tensor], torch.Tensor]] = None,

具体流程

分为三大步：预处理、通过模型传递输入、后处理

使用分词器（Tokenzier）进行预处理

将输入拆分为单词、子单词或者符号，成为标记（token）

将每一个token映射到一个整数

添加可能对模型有用的其它输入

所有预处理都需要以模型训练完全相同的方式完后才能完成，需要从Model Hub中下载相关信息。并且使用AutoTokenizer类及其方法from\_pretrained()。自动获取与模型标记器相关的数据，并进行缓存。

动手尝试

使用模型

我们可以像使用标记器一样下载预训练模型。Transformers提供了一个AutoModel类，该类还具有from\_pretrained()方法：

from transformers import AutoModel

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

model = AutoModel.from\_pretrained(checkpoint)

这个架构只包含基本转换器模块：给定一些输入，它输出我们将调用的内容隐藏状态（hidden states），亦称特征（features）。对于每个模型输入，我们将检索一个高维向量，表示Transformer模型对该输入的上下文理解。

虽然这些隐藏状态本身可能很有用，但它们通常是模型另一部分（称为头部(head)）的输入。 在Chapter 1中，可以使用相同的体系结构执行不同的任务，但这些任务中的每个任务都有一个与之关联的不同头。

Transformers模块的矢量输出通常较大。它通常有三个维度：

Batch size: 一次处理的序列数（在我们的示例中为2）。

Sequence length: 序列的数值表示的长度（在我们的示例中为16）。

Hidden size: 每个模型输入的向量维度。

输入上图中tokenizer的输出的值并作为Transformers的输入。

模型头

模型头将Transformer的输出作为输入，并将其投影到不同的维度，一般如下图所示：

Embedding和Layers组成Transformer模型。嵌入Embedding层将标记化输入的每个输入ID转换为表示关联标记(token)的向量。后续层使用注意机制操纵这些向量，以生成句子的最终表示。

对于我们的示例，我们需要一个带有序列分类头的模型（能够将句子分类为肯定或否定）。因此，我们实际上不会使用AutoModel类，而是使用AutoModelForSequenceClassification，因为AutoModel类只会传入一个模型，将其转化为向量，却没有进行分类操作，无法成为有价值的结果。

from transformers import AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

model = AutoModelForSequenceClassification.from\_pretrained(checkpoint)

outputs = model(\*\*inputs)

观察输出，可以看到输出了一个2\*2的向量，对应两个句子的两个标签。

后处理

模型的输出看起来没有什么意义，我们的模型预测第一句为[-1.5607, 1.6123]，第二句为[ 4.1692, -3.3464]。这些不是概率，而是logits，即模型最后一层输出的原始非标准化分数。

想要转换为概率他们需要经过SoftMax层

import torch

predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)

print(predictions)

这里就可以看出他们预测的概率

第一句为[0.0402, 0.9598]，第二句为[0.9995, 0.0005]。这些是可识别的概率分数。

可使用下列代码查看标签的具体意思：

model.config.id2label

模型

创建与使用模型，使用AutoModel类，其对库中各种可用模型的简单包装。

初始化BERT模型

加载配置对象 from transformers import BertConfig, BertModel

# Building the config

config = BertConfig()

# Building the model from the config

model = BertModel(config)

config包含很多用于构建模型的属性，比如说hiddensize属性定义了hidden状态向量的大小，numhiddenlayers定义了Transformer模型的层数。

该模型可以在这种状态下使用，但会输出胡言乱语；首先需要对其进行训练。这将需要很长的时间和大量的数据，并将产生不可忽视的环境影响。为了避免不必要的重复工作，必须能够共享和重用已经训练过的模型。

保存模型和加载模型一样简单—我们使用 savepretrained() 方法，类似于 frompretrained() 方法：

model.save\_pretrained("directory\_on\_my\_computer")

将模型与模型属性两个文件保存

Tokenizer

标记器(Tokenizer)是 NLP 管道的核心组件之一。它们有一个目的：将文本转换为模型可以处理的数据。由于模型只能处理数字，而输入一般为文本，故需要Tokenzier。

Tokenzier例子

1.基于词的(Word-based)

以词拆分文本

可以通过应用Python的split()函数，使用空格将文本标记为单词

使用这种标记器，我们最终可以得到一些非常大的“词汇表”，其中词汇表由我们在语料库中拥有的独立标记的总数定义。

每个单词都分配了一个 ID，从 0 开始一直到词汇表的大小。该模型使用这些 ID 来识别每个单词。每一个单词都会有一个标识符，这将生成大量的标志，且当词义相近时（比如dog和dogs）模型很难区分。最后，还需要一个自定义标记（token）来表示词汇中单词，这被称为“未知”标记(token)，通常表示为“[UNK]”或”“

2.基于字符(Character-based)

将文本拆分为字符，而不是单词，通常有两个好处：

词汇量小得多

词汇外（未知）标记少的多，因为每个单词都可以从字符创建。

这种方法也并非完美，由于现在表示的是基于字符而不是单词，可能意义不大，每个字符也没有多大意义。这又因语言而定，比如说，中文中，每个字符比拉丁语言中的字符包含更多信息。

另外，我们模型会最终处理大量的token，虽然使用基于单词的tokenizer，单词只是单个标记，转化为字符时，它很容易编程10个或者更多的字符。

3.字词标记化

不将常用词拆分更小子词，将稀有词分解为更有意义的子词。

比如，annoyingly可能认为是一个罕见的词，分解为 annoying 和ly

这种方法在土耳其语等粘着型语言(agglutinative languages)中特别有用，可以通过将子词串在一起来形成（几乎）任意长的复杂词。

加载和保存

from transformers import BertTokenizer

tokenizer = BertTokenizer.from\_pretrained("bert-base-cased")

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from\_pretrained("bert-base-cased")

使用

tokenizer("Using a Transformer network is simple")

保存

tokenizer.save\_pretrained("directory\_on\_my\_computer")

全部过程：

1.标记化

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from\_pretrained("bert-base-cased")

sequence = "Using a Transformer network is simple"

tokens = tokenizer.tokenize(sequence)

print(tokens)

此方法输出token

['Using', 'a', 'transform', '##er', 'network', 'is', 'simple']

2.由词符（token）到输入ID

输入ID的转化有tokenzier的converttokensto\_ids()方法实现：

ids = tokenizer.convert\_tokens\_to\_ids(tokens)

print(ids)

结果：

[7993, 170, 11303, 1200, 2443, 1110, 3014]

3.解码：从词汇索引中，我们想要得到一个字符串，这可以通过decode()实现，如下：

decoded\_string = tokenizer.decode([7993, 170, 11303, 1200, 2443, 1110, 3014])

print(decoded\_string)

结果：

'Using a Transformer network is simple'

微调一个预训练模型

使用MRPC数据集作为示例，有5801对句子，每个句子有一个标签，指示他们是否同义

导入数据集

简单查看一下

导入tokenizer

在话传给模型前，对两句话需要进行预处理

可以看到，包含了输入词id(inputids) 和 注意力遮罩(attentionmask) ，讨论类型标记ID(tokentypeids)。在这个例子中，类型标记ID(tokentypeids)的作用就是告诉模型输入的哪一部分是第一句，哪一部分是第二句。

转换回来：

用类型标记ID对BERT进行预训练,并且使用第一章的遮罩语言模型，还有一个额外的应用类型，叫做下一句预测. 这项任务的目标是建立成对句子之间关系的模型。

tokenized\_dataset = tokenizer(

raw\_datasets["train"]["sentence1"],

raw\_datasets["train"]["sentence2"],

padding=True,

truncation=True,

)

缺点：返回字典（字典的键是输入词id(inputids) ， 注意力遮罩(attentionmask) 和 类型标记ID(tokentypeids)，字典的值是键所对应值的列表）而且只有当在转换过程中有足够的内存来存储整个数据集时才不会出错

为了将数据保存为数据集，我们将使用Dataset.map()方法，如果我们需要做更多的预处理而不仅仅是标记化，那么这也给了我们一些额外的自定义的方法。这个方法的工作原理是在数据集的每个元素上应用一个函数，因此让我们定义一个标记输入的函数：

def tokenize\_function(example):

return tokenizer(example["sentence1"], example["sentence2"], truncation=True)

这个函数输入一个字典，返回包括输入词id(inputids) ， 注意力遮罩(attentionmask) 和 类型标记ID(tokentypeids) 键的新字典。如果键所对应的值包含多个句子，那么依旧可以工作。在map()时可以使用batched=True，显著增加标记速度。

注意：在标记函数中省略了Padding参数，因为填充到最大长度效率不高。

可见向数据集中添加了新的键。

最后一件我们需要做的事情是，当我们一起批处理元素时，将所有示例填充到最长元素的长度——我们称之为动态填充。

动态填充

负责在批处理中的数据整理为一个batch的函数成为collate函数，构建DataLoader是传递一个参数，默认是一个函数，将数据转化为PyTorch张量，并将其拼接。

为了解决句子长度统一的问题，必须定义一个collate函数，该函数会每一个batch句子填充到正确的长度。transformers库通过DataCollatorWithPadding为我们提供了这样的参数，实例化时，需要一个标记器（用来知道使用哪个词来填充，以及模型期望填充在左边或者右边）

## 3.3 情感分类流程

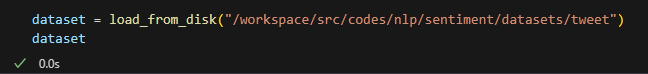
首先导入包datasets中的load\_from\_disk

datasets是NLP中技术的开源供应商开发的数据集加载库。其格式可能有所不同，但是有相关文档可以参阅。

手机屏幕的截图

描述已自动生成

导入本地的数据集



计算指标函数

文本

描述已自动生成

首先声明全局变量val\_history并置为空值。

定义函数，传入EvalPrediction类型的数据。

EvalPrediction

需要引入transformers库，这是Huggingface开发的库。Transformers提供了数以千计针对于各种任务的预训练模型模型，开发者可以根据自身的需要，选择模型进行训练或微调，也可阅读api文档和源码，快速开发新模型。

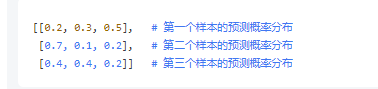
其包括两个变量：

图形用户界面, 文本, Word

中度可信度描述已自动生成

predictions表示模型预测的值（二维数组），而label\_ids表示真实的值（一维数组）。

比如说，一共有0,1,2三种结果，则predictions为



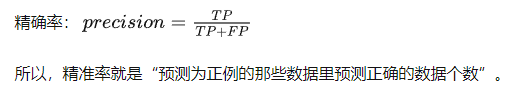
使用argmax函数，设定axis=1，算出每行最大值索引，作为预测结果。



[F1分数](https://so.csdn.net/so/search?q=F1%E5%88%86%E6%95%B0&spm=1001.2101.3001.7020)（F1-score）是分类问题的一个衡量指标。一些多分类问题的机器学习竞赛，常常将F1-score作为最终测评的方法。它是精确率和召回率的调和平均数，最大为1，最小为0。

F_{1}=2\cdot \frac{precision\cdot recall}{precision+recall}

精确率和召回率：



文本

描述已自动生成

在这里，使用sklearn库中的f1\_score，以达到衡量模型的指标



注意：average=macro的意思是：

文本

描述已自动生成

计算准确度，并返回





加载数据

图形用户界面, 文本

描述已自动生成

python中三引号可以将复杂的字符串进行复制:

python三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。

三引号的语法是一对连续的单引号或者双引号（通常都是成对的用）。

读取数据，不概述了

文本

描述已自动生成

读取结果：



创建了一个名为datasets的DatasetDict对象。

图形用户界面

描述已自动生成

DatasetDict是Hugging Face库中的一个数据结构，用于管理多个数据集（datasets）。它类似于Python的字典（dict），但是专门设计用于处理多个数据集，并提供了一些方便的功能和方法。

将数据从文件中取出并读入Dataset中。

文本

描述已自动生成

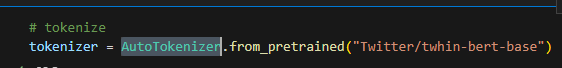
最终结果：

文本

描述已自动生成

使用AutoTokenizer来加载模型

tokenizer进行分词。其主要的任务是将文本输入转化为模型可以接受的输入，因为模型只能输入数字，所以 tokenizer 会将文本输入转化为数值型的输入



进行数据预处理：也就是使用分词器将词汇分开。

文本

描述已自动生成

可以看出，数据在处理后转化为了数字

文本

描述已自动生成



Attention Mask：

在自然语言处理中，Attention Mask是一个与输入对应的二进制向量（0,1），用于指示模型在进行注意力机制时要关注哪些位置，而忽略哪些位置。

对于序列任务（例如文本分类、命名实体识别等），输入通常由单词或字符组成，并且长度可能不一致。为了使所有的输入具有相同的长度，我们需要进行填充（padding）操作。填充后的序列通常包含特殊的填充符号（如0），以示区分。

Attention Mask的作用就是告诉模型哪些部分是真实的输入内容，哪些部分是填充的。它通常是一个与输入序列相同长度的二进制向量，其中1表示真实的输入，0表示填充。

在本次学习中，使用的模型支持不同长度输入，故不需要1,0表示，默认全为1。

生成data\_collator

图形用户界面, 文本, 网站

描述已自动生成

准备模型

文本, 日程表

描述已自动生成

并定义名称

0 为负面的，1 为中立 2为 积极的。

选择参数：

文本

描述已自动生成

TrainingArguments是Hugging Face的transformers库中的一个类，它用于定义训练过程中的各种参数。以下是你提供的代码中使用的参数及其含义：

"test/results": 指定训练结果的保存路径。

do\_eval=True: 设置为True时，在训练过程中会进行评估。

evaluation\_strategy="steps": 设置评估策略为按步骤进行评估。

learning\_rate=0.0001: 学习率的初始值设为0.0001。

per\_device\_eval\_batch\_size=16: 每个设备上用于评估的批处理大小为16。

per\_device\_train\_batch\_size=16: 每个设备上用于训练的批处理大小为16。

num\_train\_epochs=20: 训练轮数设置为20。

overwrite\_output\_dir=True: 如果设置为True，则会覆盖以前保存的结果。

logging\_steps=10: 每隔10个步骤记录一次日志信息。

eval\_steps=20: 每隔20个步骤进行一次评估。

文本

描述已自动生成

设置模型参数并开始训练。

图形用户界面, 文本

描述已自动生成

## 3.4 Setfit模型

**SetFit: 一个基于**[Sentence Transformers](https://sbert.net/)**的高效的少样本微调 (fine-tune) 框架**

!<https://huggingface.co/blog/assets/103_setfit/setfit_curves.png>

优势

🗣 **无需提示词或语言器 (verbalisers)**: 目前的少样本微调技术都需要手工设计的提示或语言器，用于将训练样本转换成适合目标语言模型的格式。SetFit 通过直接从少量标注训练样本中生成丰富的嵌入，完全省去了提示。

🏎 **快速训练**: SetFit 不需要使用像 T0 或 GPT-3 这样的大规模语言模型就能达到高准确率。因此，典型情况下，它的训练和推理会快一个数量级或以上。

🌎 **支持多语言**: SetFit 可与 Hub 上的任一 Sentence Tranformer 一起使用，这意味着如果你想让它支持多语言文本分类，你只要简单地微调一个多语言的 checkpoint 就好了。

工作流程

SetFit 的两阶段训练过程

SetFit 主要包含两个阶段：首先在少量标注样例 (典型值是每类 8 个或 16 个样例) 上微调一个 Sentence Transformer 模型。然后，用微调得到的 Sentence Tranformer 的模型生成文本的嵌入 (embedding) ，并用这些嵌入训练一个分类头 (classification head) 。

SetFit 利用 Sentence Transformer 的能力去生成基于句对 (paired sentences) 的稠密嵌入。

第一步微调阶段，它使用对比训练 (contrastive training) 来最大化利用有限的标注数据。首先，通过选择类内 (in-class) 和类外 (out-class) 句子来构造正句对和负句对，然后在这些句对上训练 Sentence Transformer 模型并生成每个样本的稠密向量。

第二步，根据每个样本的嵌入向量和各自的类标签，训练分类头。推理时，未见过的样本通过微调后的 Sentence Transformer 并生成嵌入，生成的嵌入随后被送入分类头并输出类标签的预测。

简言之，第一步使用Sentence Transformer，生成每一个样本（评论）的向量表示，而在第二步则将第一部导出的向量进行分类。

只需要把基础 Sentence Transformer 模型换成多语言版的，SetFit 就可以无缝地在多语言环境下运行。在实验中，SetFit 在德语、日语、中文、法语以及西班牙语中，在单语言和跨语言的条件下，都取得了不错的分类性能。

**注：**

对比训练即将正句对x+（分类相同的句子），与负句对x-（分类不同的句子）

比如在政治言论中，0代表不是，1代表是， 0与0，1与1是正键值对。10是负键值对。

流程

丹麦语效果不好

# 导入库

! pip install datasets

! pip install sentence\_transformers

! pip install setfit

! pip install transformers

from datasets import load\_dataset

from sentence\_transformers.losses import CosineSimilarityLoss

from setfit import SetFitModel, SetFitTrainer

# 导入数据集

dataset = load\_dataset("danish\_political\_comments")

# dataset.save\_to\_disk("./danish\_comments")

查看一下数据集的情况

dataset

结果，可以看到数据集有三个feature，id sentence 和target，要用的是sentence和targe

DatasetDict({

train: Dataset({

features: ['id', 'sentence', 'target'],

num\_rows: 9008

})

})

丹麦数据集不进行数据处理的话一共有四类型标签，1,2,3,4；故随机每类型的标签8份数据。验证集没类型的标签100份数据。

# 随机分割测试集与验证集

train\_ds = dataset["train"].shuffle(seed=42).select(range(8 \* 4))

test\_ds = dataset["train"].shuffle(seed=44).select(range(100 \* 4))

#进行trainer的初始化，注意setFit模型输入参数为text与label，故需要进行转化，否则会报错。

column\_mapping={"sentence": "text", "target": "label"}

model = SetFitModel.from\_pretrained("sentence-transformers/paraphrase-mpnet-base-v2")

trainer = SetFitTrainer(

model=model,

column\_mapping=column\_mapping,

train\_dataset=train\_ds,

eval\_dataset=test\_ds,

loss\_class=CosineSimilarityLoss,

batch\_size=16,

num\_iterations=20, # Number of text pairs to generate for contrastive learning

num\_epochs=1 # Number of epochs to use for contrastive learning

)

训练

trainer.train()

trainer.evaluate()

西班牙语

from sklearn.model\_selection import train\_test\_split

spanish\_dataset = load\_from\_disk("/workspace/zmj/dataset/Violencia-politica-genero")

查看数据集

DatasetDict({

train: Dataset({

features: ['id', 'text', 'task'],

num\_rows: 32090

})

})

#数据预处理

def compute\_res(example):

return {"task": 0 if example["task"] == 'no violencia política de género' else 1 }

spanish\_dataset=spanish\_dataset.map(compute\_res)

spanish\_dataset["train"]["task"][:3]

#分割训练集以及验证集

train\_ds, test\_ds = train\_test\_split(spanish\_dataset["train"], test\_size=0.2, random\_state=42)

train\_ds = Dataset.from\_dict(Dataset.from\_dict(train\_ds)[:500])

test\_ds = Dataset.from\_dict(test\_ds)

column\_mapping={"text": "text", "task": "label"}

model = SetFitModel.from\_pretrained("sentence-transformers/paraphrase-mpnet-base-v2")

trainer = SetFitTrainer(

model=model,

column\_mapping=column\_mapping,

train\_dataset=train\_ds,

eval\_dataset=test\_ds,

metric="accuracy",

loss\_class=CosineSimilarityLoss,

batch\_size=16,

num\_iterations=20, # Number of text pairs to generate for contrastive learning

num\_epochs=1 # Number of epochs to use for contrastive learning

)

#进行训练

trainer.train()

trainer.evaluate()

训练结果

可以直接使用SetFit库来加载SetFit模型

from setfit import SetFitModel, SetFitTrainer

此外，需要导入sentence transform模型

# 第4章 总结与收获

在本次实习中，我主要参与了主题分类和情感分类的任务，并使用了一些相关工具和框架进行模型的开发和评估。此外，我还学习了Docker和Hugging Face等技术，为NLP领域的工作提供了便利。

主题分类： 在主题分类任务中，我采用了setfit模型来进行主题分类。setfit是一个强大的机器学习库，它提供了丰富的算法和功能，用于文本数据的特征提取、特征选择和模型训练等步骤。通过使用setfit模型，我能够从给定的文本中提取重要的特征，并使用不同的分类算法进行主题分类。在实践中，我尝试了多种特征表示方法和模型参数的调整，以优化主题分类模型的准确性和性能。

情感分类： 在情感分类任务中，我选用了预训练模型来进行情感分类。预训练模型是基于海量数据进行训练的通用语义理解模型，可以在各种NLP任务上取得良好的效果。我使用Hugging Face提供的Transformers库，该库提供了方便的接口和封装，使得加载和微调预训练模型变得更加简单。通过微调预训练模型并进行情感分类任务的训练，我成功实现了准确且高性能的情感分类模型。

Docker和Hugging Face学习： 除了主题分类和情感分类任务，我还学习了Docker和Hugging Face等技术。Docker是一种容器化平台，可以将应用程序及其依赖项打包成独立的容器，方便在不同环境中部署和运行。我学会了使用Docker构建和管理NLP模型的开发环境，提高了工作的灵活性和可移植性。同时，我也学习了Hugging Face这个流行的NLP工具集合，它提供了各种预训练模型、数据集和训练工具，为NLP任务的研究和应用提供了便利。

通过本次实习，我深入学习了主题分类和情感分类的理论知识，并通过使用setfit和预训练模型成功地完成相关任务。同时，掌握了Docker和Hugging Face等工具和技术，为模型的开发和部署提供了更高效和便捷的方式。这次实习经历对于我的技术提升和职业发展都具有重要意义，使我对NLP领域有了更深入的了解和实践经验。