# 知识工程-作业4 文本分类

2024214500 叶璨铭

## 代码与文档格式说明

本文档使用Jupyter Notebook编写，所以同时包括了实验文档和实验代码。

本次实验项目采用了类似于 Quarto + nbdev 的方法来同步Jupyter Notebook代码到python文件, 因而我们的实验文档导出为pdf和html格式可以进行阅读，而我们的代码也导出为python模块形式，可以作为代码库被其他项目使用。 我们这样做的好处是，避免单独管理一堆 .py 文件，防止代码冗余和同步混乱，py文件和pdf文件都是从.ipynb文件导出的，可以保证实验文档和代码的一致性。

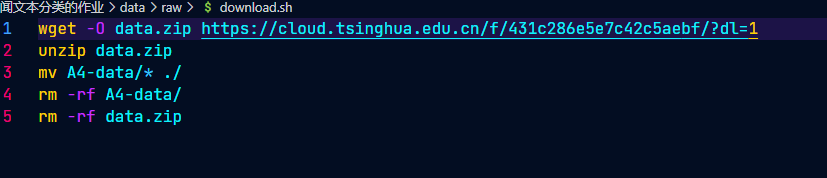
本文档理论上支持多个格式，包括ipynb, html, docx, pdf, md 等，但是由于 quarto和nbdev 系统的一些bug，我们目前暂时只支持ipynb文件，以后有空的时候解决bug可以构建一个[在线文档网站](https://thu-coursework-machine-learning-for-big-data-docs.vercel.app/)。您在阅读本文档时，可以选择您喜欢的格式来进行阅读，建议您使用 Visual Studio Code (或者其他支持jupyter notebook的IDE, 但是VSCode阅读体验最佳) 打开 ipynb格式的文档来进行阅读。

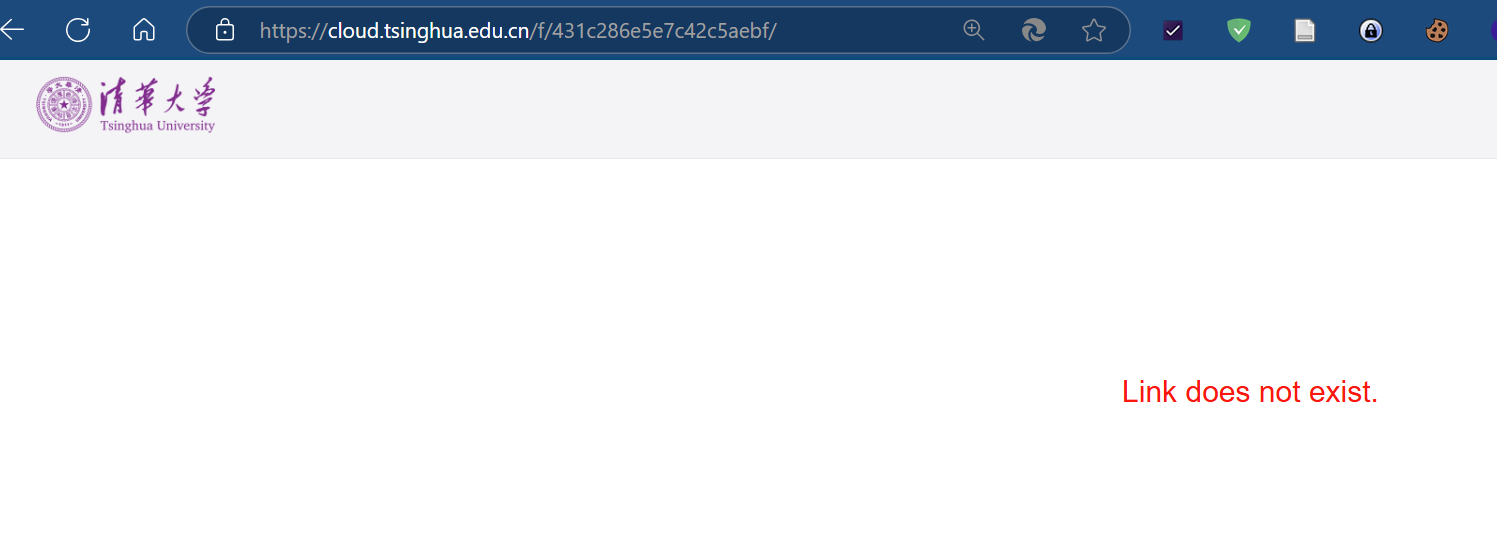
为了记录我们自己修改了哪些地方，使用git进行版本控制，这样可以清晰地看出我们基于助教的代码在哪些位置进行了修改，有些修改是实现了要求的作业功能，而有些代码是对助教的代码进行了重构和优化。我将我在知识工程课程的代码，在作业截止DDL之后，开源到 <https://github.com/2catycm/THU-Coursework-Knowledge-Engineering.git> ，方便各位同学一起学习讨论。

## 数据下载

cd data/raw  
sh download.sh

但是老师给的链接过期了，





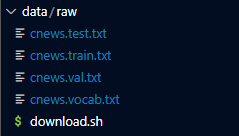
本次实验使用数据集来自清华大学2016年构建的新闻文本分类数据集 THUCNews，共包含14个类别的74万篇新闻文档，可以在 <http://thuctc.thunlp.org/message> 获取，均为UTF-8纯文本格式

根据说明，我们进入thunlp的链接，填写研究者信息之后，可以看到下载链接



不过这个数据集特别大，我们这次实验的是子集，所以只用非常好的学长在群里面分享的数据子集，代替脚本中的下载步骤，然后我们就可以解压了。

由于文件层级不一样，我们不用脚本，直接解压。



## 数据预处理

我们看下 main.py 文件

使用到数据的地方是

word2vec\_model = load\_word2vec\_model(file="./data/raw/cnews.train.txt", vector\_size=vector\_size)  
...  
train\_dataset = MyDataset("./data/raw/cnews.train.txt", text\_vocab=text\_vocab, pad\_token=pad\_token, unk\_token=unk\_token, max\_length=max\_length)

load\_word2vec\_model gensim库会直接处理这个txt，我们稍后再下一节实现

实际上训练for循环里面，对于MyDataset的数据要求是这样的

for text, label in train\_loader:  
 text = text.to(device)  
 label = label.to(device)  
 prediction = model(text)  
 loss = loss\_function(prediction, label)  
 loss.backward()  
 optimizer.step()

所以我们需要去 dataset.py 实现 MyDataset 类，让每一个item是一个text和label的pair

我们首先用ruff格式化一下 dataset.py 方便开发 ruff format dataset.py

注意看，\_\_init\_\_调用了 load 函数需要我们实现

def \_\_init\_\_(...):  
 self.text, self.label = self.load(file)

随后检查了这两个数量要一样多，建立了 label2index, word2index, 然后调用了pad。 助教用的注释规范太长了，我们使用fastai规范来重新注释。

在注释的过程中，我们很快就发现，助教的代码的类型不严谨，self.text 有时候是tensor，有时候是list[list[int]]， 语义不规范，导致VSCode报了很多错，我们先重构一下助教的代码，增加合适的注释和类型提示。

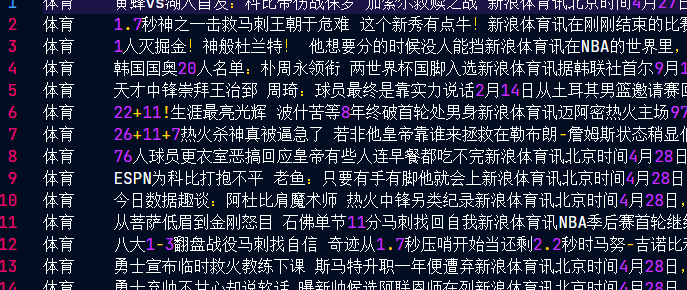
清晰的类型注解也是能够帮助我们更好的理解代码的，提高我们对作业的理解，所以不惜花一点时间。

import torch  
from torch.utils.data import Dataset, DataLoader  
from typing import Optional  
  
  
class MyDataset(Dataset):  
 def \_\_init\_\_(  
 self,  
 file: str, # 文件路径  
 text\_vocab: dict, # 文本词汇  
 max\_length: int = 1024, # 最大长度  
 pad\_token: str = "<PAD>", # 填充标记  
 unk\_token: str = "<UNK>", # 未知标记  
 label2index: Optional[dict] = None, # 标签映射  
 ) -> None:  
 # 先写self是更加规范的。  
 self.text\_vocab = text\_vocab  
 self.pad\_token = pad\_token  
 self.unk\_token = unk\_token  
 self.max\_length = max\_length  
 # 直接保存的参数写完了，接下来才写计算逻辑  
  
 # 加载原始文本和标签  
 # 这里还没变成张量，不要搞混淆了  
 raw\_text, raw\_labels = self.load(file)  
 assert len(raw\_text) == len(raw\_labels), "text: {}, label: {}".format(  
 len(raw\_text), len(raw\_labels)  
 )  
 # assert condition, error\_message 才是规范写法，助教写print有误。  
  
 # 初始化或使用标签映射  
 if label2index is None:  
 self.label2index = dict(  
 zip(sorted(set(raw\_labels)), range(len(set(raw\_labels))))  
 )  
 else:  
 self.label2index = label2index  
  
 # 转换标签为整数  
 # convert\_label2index 函数不应该暴露到外面，而且只有一行，直接在这里实现  
 self.\_labels = [self.label2index[label] for label in raw\_labels]  
 assert len(self.\_labels) == len(raw\_labels), "\_labels: {}, raw\_labels: {}".format(  
 len(self.\_labels), len(raw\_labels)  
 )  
  
 # 转换文本为词索引  
 indexed\_text = self.word2index(raw\_text)  
 assert len(indexed\_text) == len(raw\_text), "indexed\_text: {}, raw\_text: {}".format(  
 len(indexed\_text), len(raw\_text)  
 )  
  
 # 填充并转换为张量  
 # 合理的接口设计不应该使用 self传递参数，而是应该明确传递。  
 padded\_text = self.pad(indexed\_text)  
 self.\_text\_tensor = torch.tensor(padded\_text)  
   
 def \_\_len\_\_(self) -> int: # 返回数据集大小  
 return len(self.\_text\_tensor)  
  
 def \_\_getitem\_\_(self, item: int # 数据索引  
 ) -> tuple[torch.Tensor, int]: # 返回(文本张量,标签)  
 return self.\_text\_tensor[item], self.\_labels[item]

现在我们严格区分了 \_text\_tensor 和 raw\_text，杜绝了类型问题。

现在可以开始按照init中调用的顺序来实现，首先是load函数

简单查看一下文件数据，比如cnews.val.txt，



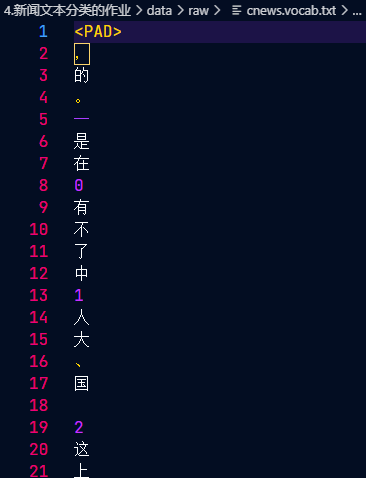
我们可以看到数据是，类别 \t 文本 的形式。

def load(  
 self,  
 file: str, # 输入文件路径  
) -> tuple[list[str], list[str]]: # 返回(文本列表,标签列表)  
 """  
 read file and load into text (a list of strings) and label (a list of class labels)  
 """  
 text, label = [], []  
 with open(file, "r", encoding="utf-8") as f:  
 for line in f:  
 # 每行格式: 标签\t文本内容  
 label\_txt, content = line.strip().split("\t")  
 text.append(content)  
 label.append(label\_txt)  
 return text, label

接下来 word2index

这里我们需要把句子的每个单词转换为int

cnews.vocab.txt 是这样的



其实我们没有搞中文分词，直接单字成词，所以要用 .split("") 直接把每个字分开。然后不再vocab里面的要用<UNK>标注。

助教在main中这样写

# add unk\_token and pad\_token  
unk\_index = text\_vocab[unk\_token] = len(text\_vocab)  
pad\_index = text\_vocab[pad\_token] = len(text\_vocab)

实际上数据中已经有 <PAD> 不过没关系，这是因为助教其实用gensim的word2vec\_model.wv.key\_to\_index作为vocab，而不是原来的那个vocab文件。这个是从训练集提取的。

这样搞才是对的，因为待会这里的int tensor还要被word2vec处理为float dense tensor，需要按照人家model的定义来。

def word2index(  
 self,  
 text: list[str], # 输入文本列表  
) -> list[list[int]]: # 返回词索引列表的列表  
 """  
 convert loaded text to word\_index with text\_vocab  
 self.text\_vocab is a dict  
 """  
 \_text = []  
 for sentence in text:  
 # 将句子分词并转换为词索引  
 words = sentence.strip().split("")  
 # 如果词不在词表中，使用UNK的索引  
 indices = [  
 self.text\_vocab.get(word, self.text\_vocab[self.unk\_token]) for word in words  
 ]  
 \_text.append(indices)  
 return \_text

现在可以写pad，目的是为了让每个句子长度一样，不够的补<PAD>，太长的截断。

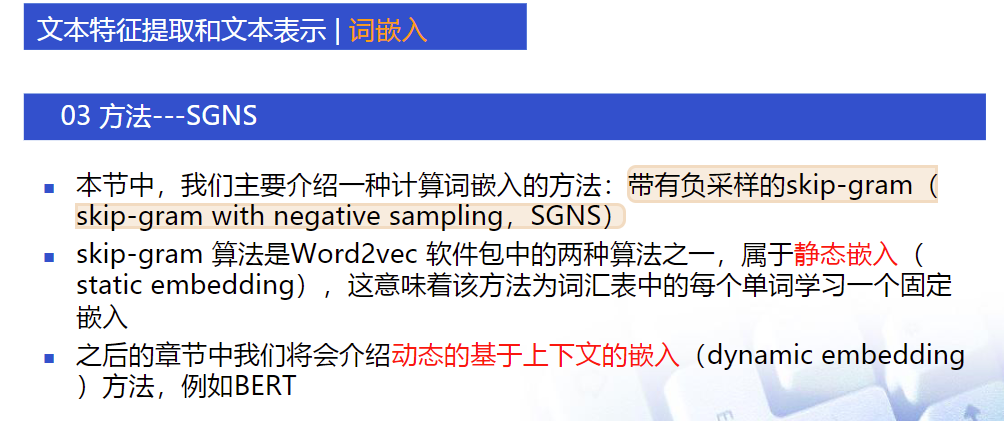
用到 self.text\_vocab[self.pad\_token]

def pad(  
 self,  
 text: list[list[int]], # 待填充的词索引列表  
) -> list[list[int]]: # 返回填充后的词索引列表  
 """  
 pad word indices to max\_length  
 """  
 pad\_text = []  
 for \_text in text:  
 # 如果长度超过max\_length则截断  
 if len(\_text) > self.max\_length:  
 pad\_text.append(\_text[: self.max\_length])  
 else:  
 # 如果长度小于max\_length则用pad\_token的索引填充  
 pad\_text.append(  
 \_text  
 + [self.text\_vocab[self.pad\_token]] \* (self.max\_length - len(\_text))  
 )  
 return pad\_text

## 基于gensim工具包训练带有负采样的 skip-gram

在本节中，我们将使用gensim工具包来训练一个带有负采样的skip-gram模型。

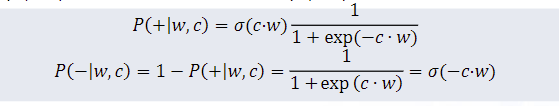
我们首先复习一下课件



Skip-gram模型是一种用于词向量训练的模型，属于word2vec的一种，通过预测给定词语的上下文词语来学习词向量。负采样是一种加速训练过程的方法，通过减少计算量来提高训练效率。

其中课件说的“静态向量”应该是指词向量不参与后续训练。

具体学习的原理是，最大化目标词和上下文词的余弦相似度，最小化目标词和负样本词的余弦相似度。 负样本太多了，所以从词汇表中采样出来。



现在我们可以实现代码了。 我们首先找到官方仓库的链接，<https://github.com/piskvorky/gensim> ，根据指南，直接pip 安装即可。readme提到这个库已经是稳定阶段，不再增加新功能。

# 安装gensim工具包  
pip install --upgrade gensim

阅读 main.py 我们可以看到，使用到word2vec的代码如下：

word2vec\_model = load\_word2vec\_model(file="./data/raw/cnews.train.txt", vector\_size=vector\_size)  
word\_embeddings = get\_word\_embeddings(word2vec\_model, vector\_size=vector\_size)

因此我们首先到 util.py 实现 load\_word2vec\_model 函数。

# | export  
import os  
import gensim  
from util import load\_text  
  
  
def load\_word2vec\_model(file=None, vector\_size=100):  
 # train word2vec with gensim  
 if os.path.exists("word2vec"):  
 word2vec\_model = gensim.models.word2vec.Word2Vec.load("word2vec")  
 else:  
 text = load\_text(file)  
 # Train word2vec model with gensim  
 word2vec\_model = gensim.models.word2vec.Word2Vec(  
 sentences=text, vector\_size=vector\_size, window=5, min\_count=1, workers=4  
 )  
 word2vec\_model.save("word2vec")  
 return word2vec\_model

首先if是判断模型是否已经训练成功，如果训练成功，直接加载模型，否则重新训练模型。

训练的代码我们参考了文档 <https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html#sphx-glr-auto-examples-tutorials-run-word2vec-py> 的“Training Your Own Model” 章节，使用训练数据 text 作为输入，设置参数 vector\_size（向量维度）、窗口大小（window=5）、min\_count=1（忽略所有频次小于1的词）和 workers=4（并行训练用的线程数）。

## TextCNN 模型实现

在 main.py 里面，

word\_embeddings = get\_word\_embeddings(word2vec\_model, vector\_size=vector\_size)  
model = TextCNN(  
 word\_embeddings, vector\_size, label2index, pad\_index, max\_length=1024  
).to(device)

把 gensim 的 word\_embeddings 传入到了 TextCNN的初始化中。

这个实际上是一个 np.array 矩阵，在util.py中看到

def get\_word\_embeddings(  
 word2vec\_model, vector\_size=100, pad\_token="<PAD>", unk\_token="<UNK>"  
):  
 ...  
 word\_embeddings = np.zeros((len(text\_vocab), vector\_size))  
 ...  
 return word\_embeddings

每一行是vocab index对应的那个词向量

现在我们可以打开 cnn.py，老规矩，先把助教的代码规范化一下，不仅ruff format，还把类型注释搞对，知道每个函数的输入输出和参数的定义和类型。

这里代码不多，不需要特别重构。

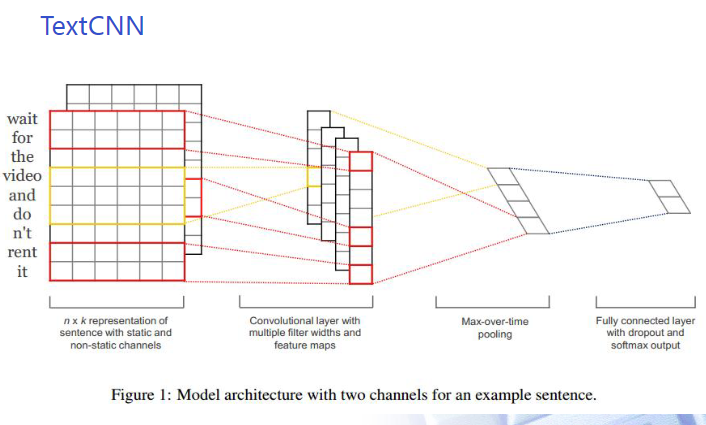
现在我们直接开始写TextCNN.

首先我们处理好外面传进来的 word\_embeddings ， 直接用 <https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html> nn.Embedding.from\_pretrained

由于老师讲解强调SGNS是静态嵌入，我们就当做静态嵌入，freeze=True 不参与训练。

现在我们看看卷积怎么实现

回顾老师课件



filter\_size 表示纵向上，认为前后多少个词是有关系的，比如红色框框是2， 红色框框对应了卷积核也是那么多，卷起来就是乘法求和，得到单个数。如果一个位置想要得到很多数，那就需要多个卷积核。

随后我们查看 nn.Conv1d 的文档 <https://pytorch.org/docs/stable/generated/torch.nn.Conv1d.html>

in\_channels 对应 打横的 embedding size， out\_channels 对应一个位置输出多少个数， kernel\_size 对应纵向的filter\_size

每一个 fitler\_size 卷积完之后，会得到比原来单词数量稍微短一点的向量，为了和位置无关的得到一个全局句子的特征表示，需要做一个pooling，比如课件提到的max pooling。

所以说我们分类器的输入有 channels \* len(filter\_size) 这么多个 （不算batch size，linear只对最后一个维度操作）

import numpy as np  
import torch  
from torch import nn  
  
  
class TextCNN(nn.Module):  
 def \_\_init\_\_(  
 self,  
 word\_embeddings: np.ndarray, # 预训练词向量矩阵(N\*D)  
 vector\_size: int, # 词向量维度 D  
 label2index: dict, # 标签到索引的映射  
 pad\_index: int, # 填充token的索引  
 filter\_size: list[int] = [2, 3, 4, 5], # CNN卷积核大小  
 channels: int = 64, # CNN输出通道数  
 max\_length: int = 1024, # 最大序列长度  
 ) -> None:  
 super(TextCNN, self).\_\_init\_\_()  
 # Initialize embedding layer with pre-trained word\_embeddings  
 self.embedding = nn.Embedding.from\_pretrained(  
 torch.FloatTensor(word\_embeddings), freeze=True, padding\_idx=pad\_index  
 )  
 # Build a stack of 1D CNN layers for each filter size  
 self.convs = nn.ModuleList(  
 [  
 nn.Conv1d(in\_channels=vector\_size, out\_channels=channels, kernel\_size=k)  
 for k in filter\_size  
 ]  
 )  
 # Final linear layer for label prediction; number of classes equals len(label2index)  
 num\_class = len(label2index)  
 self.linear = nn.Linear(channels \* len(filter\_size), num\_class)

init写好了，forward自然也不难。

def forward(  
 self,  
 inputs: torch.Tensor, # 输入张量(N\*L)  
 ) -> torch.Tensor: # 返回预测logits(N\*K)， 不需要softmax  
 # Embedding layer  
 x = self.embedding(inputs) # 得到 (N\*L\*D)  
 # Convolutional layer  
 x = x.transpose(1, 2) # 卷积需要将词向量维度放在最后 (N\*D\*L)  
 x = [conv(x) for conv in self.convs]  
 x = [nn.functional.gelu(i) for i in x] # 每一个 i是 (N\*C\*Li) ， Li = L - ki + 1  
 # Pooling layer  
 x = [  
 nn.functional.max\_pool1d(  
 i,  
 kernel\_size=i.size(2), # 对 Li 去做 max\_pooling  
 ).squeeze(2)  
 for i in x # 每一个 i是 (N\*C\*Li)  
 ] # 每一个 item 变为 (N\*C)  
 # Concatenate all pooling results  
 x = torch.cat(x, dim=1) # 把每一个 item 拼接起来，变为 (N, C\*len(filter\_size))  
 # Linear layer  
 x = self.linear(x) # 分类，得到 (N\*K)  
 return x

这里我们使用了relu作为激活函数，这个老师没有提到，但是我觉比较需要。

检查代码，其实不规范的是，init里面 max\_length 没有用到，因为前面dataset已经处理过了，不过为了规范，我们还是改一下，forward的时候检查一下。

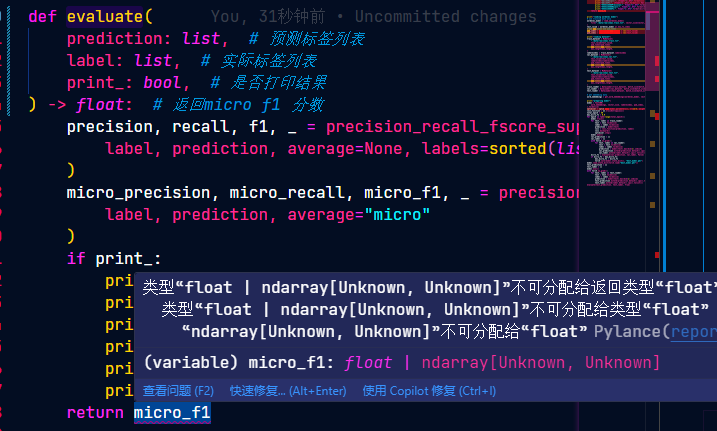
# check max\_length  
if inputs.size(1) > self.max\_length:  
 inputs = inputs[:, : self.max\_length]

## 评价指标

复习老师课件，macro就是直接每个类别的P，R，f1平均起来，而micro对每个类别的TP，FP，FN求和，然后计算P，R，f1。

认为样本量大的时候micro更重要，样本量小的时候macro更重要。

注意到 main.py evaluate 类型不够严谨。

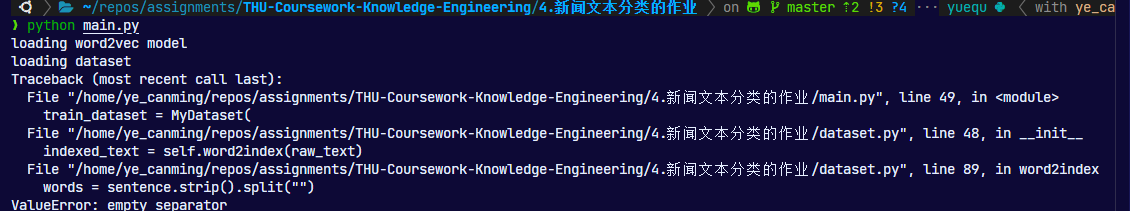


这是因为 sklearn.metrics.precision\_recall\_fscore\_support , 参考文档 <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html> 可能返回的不是float，看文档就懂了，没有 average 的时候是各个类别的list，有average的时候是一个数。

但是pylance不知道，我们告诉它一定是float就行。

assert isinstance(micro\_f1, float)  
return micro\_f1

## 训练模型

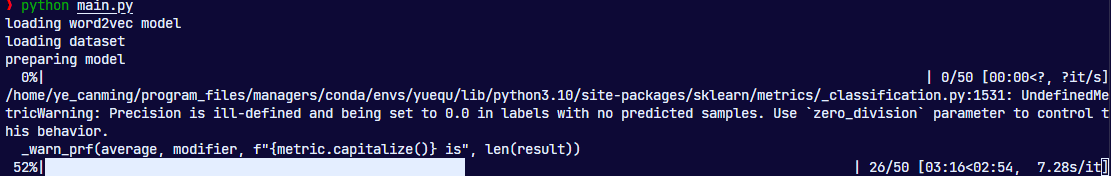


原来split写的不好。我们的目的是把str变成char的列表，在Java里面确实经常写 split("")，但是Python里面认为这个不对，所以我们要用python的方式来写。

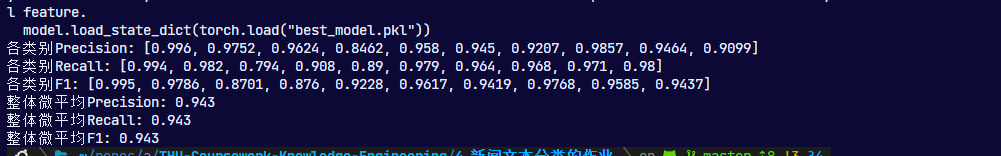
words = list(sentence.strip())

这样就行了。

现在可以成功训练了



训练成功我们得到了结果



各类别Precision: [0.996, 0.9752, 0.9624, 0.8462, 0.958, 0.945, 0.9207, 0.9857, 0.9464, 0.9099]  
各类别Recall: [0.994, 0.982, 0.794, 0.908, 0.89, 0.979, 0.964, 0.968, 0.971, 0.98]  
各类别F1: [0.995, 0.9786, 0.8701, 0.876, 0.9228, 0.9617, 0.9419, 0.9768, 0.9585, 0.9437]  
整体微平均Precision: 0.943  
整体微平均Recall: 0.943  
整体微平均F1: 0.943

相比老师给出的结果

各类别Precision: [0.999, 0.9418, 0.9822, 0.8036, 0.9374, 0.9798, 0.9197, 0.9554, 0.9171, 0.9406]  
各类别Recall: [0.99, 0.987, 0.719, 0.929, 0.899, 0.972, 0.928, 0.964, 0.973, 0.981]  
各类别F1: [0.9945, 0.9639, 0.8303, 0.8618, 0.9178, 0.9759, 0.9238, 0.9597, 0.9442, 0.9604]  
整体微平均Precision: 0.9342  
整体微平均Recall: 0.9342  
整体微平均F1: 0.9342

我们的性能提高了1%，有可能是因为我们用了激活函数relu，不知道助教的实现和我们是不是这个区别。

参考TextCNN的论文仓库实现 <https://github.com/delldu/TextCNN/blob/master/model.py>

可以看到这个实现里面有relu，比我们多一个dropout。

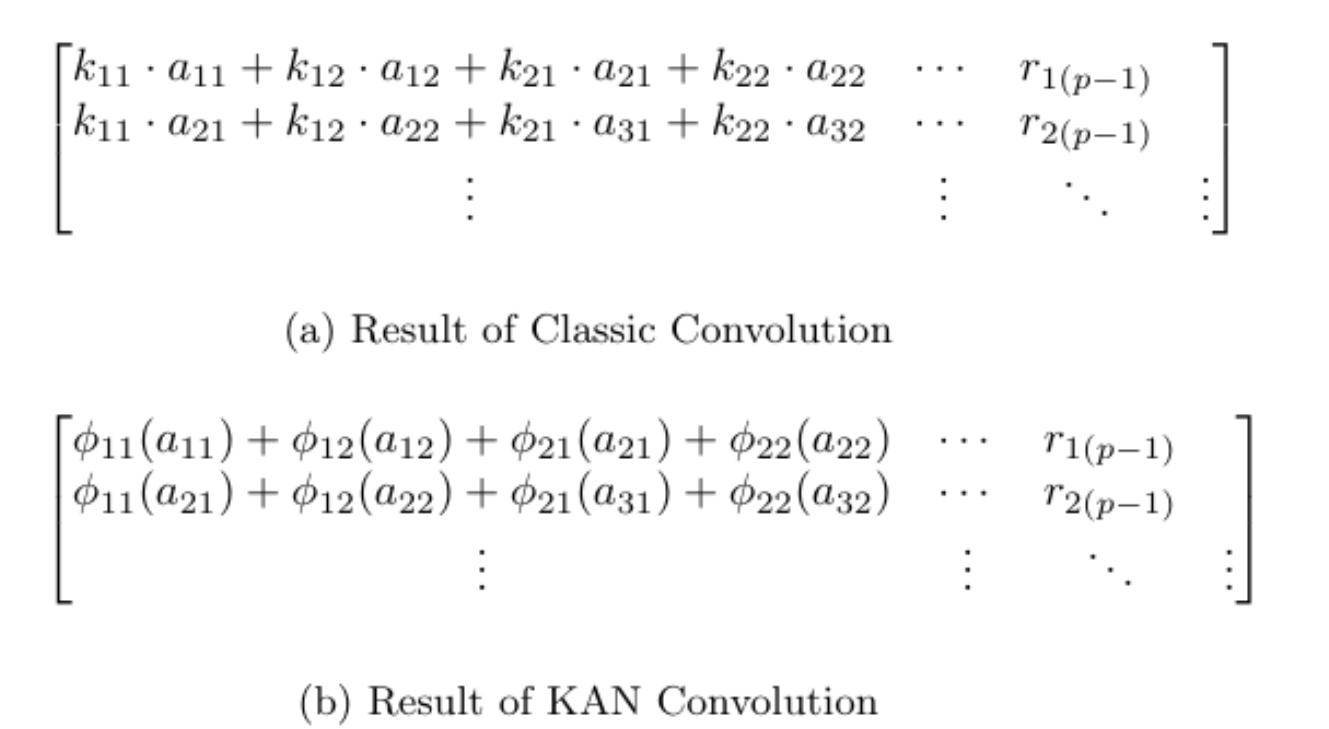
有可能我们数据量少，所以去掉了dropout效果更好。

## 探究进一步提高TextCNN性能的思路

近期爆火的明星网络KAN，网上褒贬不一，理论上这个网络确实很创新，但是实测效果很多人说视觉领域不一定优于MLP，需要做一些改进才行。

我们正好来试试文本分类任务，用卷积KAN代替卷积。

其实公式很简单，所谓的KAN就是把矩阵乘法的求和不变，乘法换成了可学习激活函数。



我们调用开源库 ckan。 因为这个作者不太会搞pypi包，弄得有点乱，我用submodule和软链接的方式引入。

git submodule add https://github.com/AntonioTepsich/Convolutional-KANs.git  
cd Convolutional-KANs  
pip install pyprof  
cd ..  
ln -s Convolutional-KANs/kan\_convolutional

由于这个库没有搞Conv1d 我们做一个转换器

import torch  
import torch.nn as nn  
  
class Conv1dViaConv2d(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size,   
 stride=1, padding=0, dilation=1, groups=1, bias=True,   
 conv\_2d=nn.Conv2d):  
 super(Conv1dViaConv2d, self).\_\_init\_\_()  
 self.conv2d = conv\_2d(  
 in\_channels,  
 out\_channels,  
 (1, kernel\_size),  
 stride=(1, stride),  
 padding=(0, padding),  
 dilation=(1, dilation),  
 groups=groups,  
 bias=bias,  
 )  
  
 def forward(self, x):  
 # 调整输入维度  
 x = x.unsqueeze(2) # 添加一个高度维度  
 # 执行 Conv2d  
 x = self.conv2d(x)  
 # 移除多余维度  
 x = x.squeeze(2)  
 return x

# 示例用法  
input\_data = torch.randn(1, 3, 10) # (batch\_size, in\_channels, length)  
conv1d\_layer = Conv1dViaConv2d(3, 2, 3)  
output\_data = conv1d\_layer(input\_data)  
print(output\_data.shape) # 输出：torch.Size([1, 2, 8])

torch.Size([1, 2, 8])

由于时间关系，这个并没有跑通。这个库有很多bug，我们这样做了之后不能直接跑通，我们以后再探究。

或者在 <https://github.com/2catycm/THU-Coursework-Knowledge-Engineering.git> 看到我们的更新。