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。

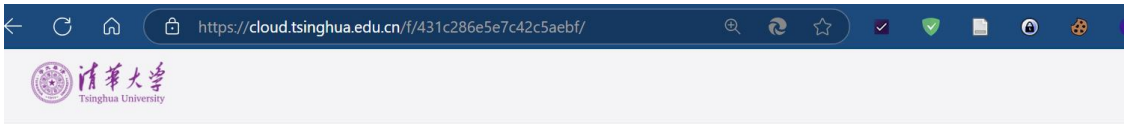
本文档理论上支持多个格式，包括 ipynb, html, docx, pdf, md 等，但是由于 quarto 和 nbdev 系统的一些 bug，我们目前暂时只支持 ipynb 文件，以后有空的时候解决 bug 可以构建一个在线文档网站。您在阅读本文档时，可以选择您喜欢的格式来进行阅读，建议您使用 Visual Studio Code（或者其他支持 jupyter notebook 的 IDE，但是 VSCode 阅读体验最佳）打开 ipynb 格式的文档来进行阅读。

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，



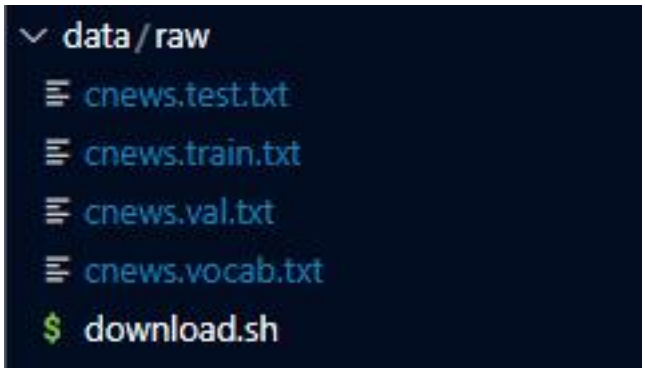
Link does not exist.

> 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。

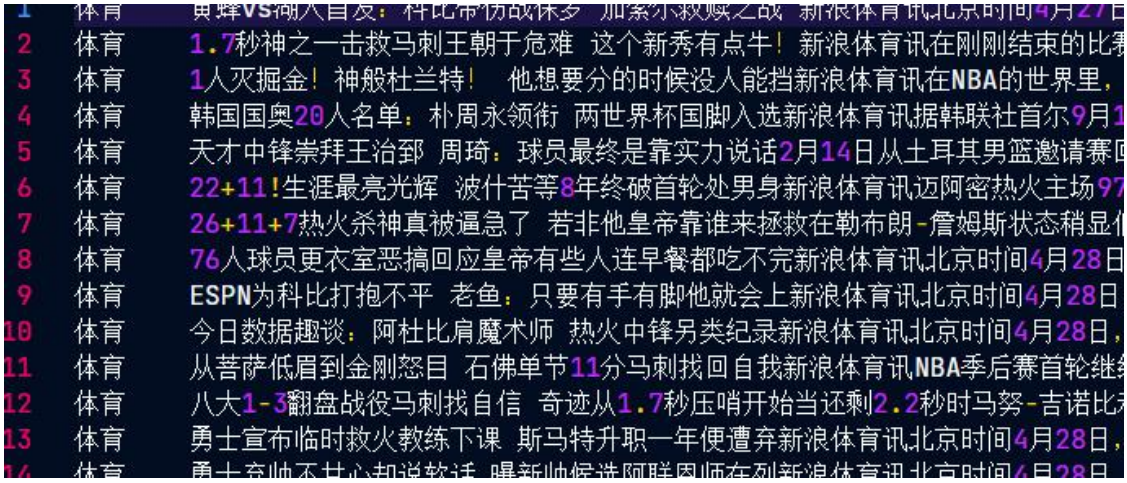
```
import torch
from torch.utils.data import Dataset, DataLoader
from typing import Optional


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_label
s))))
            )
        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: {}, r
aw_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`，



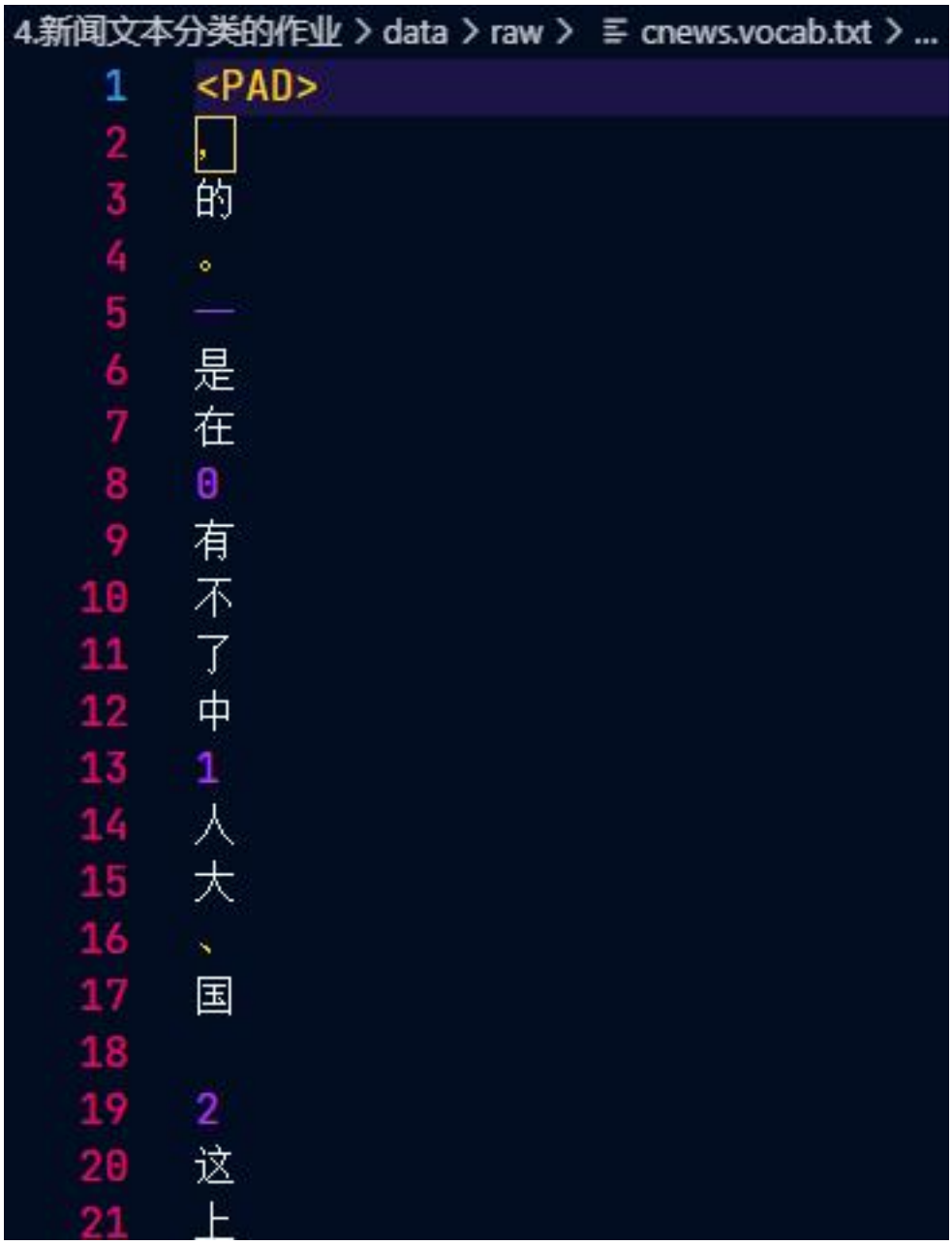
我们可以看到数据是，类别 `\t` 文本 的形式。

```
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]

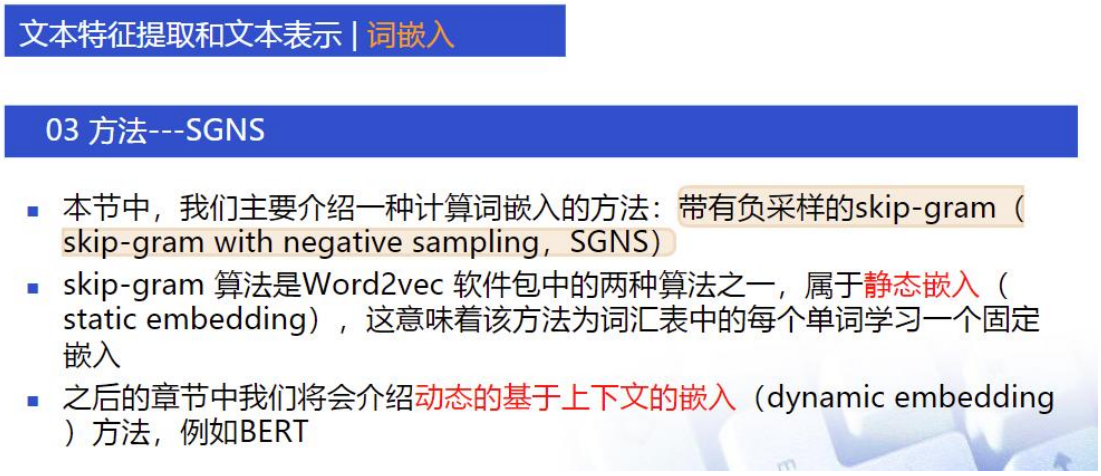
```
def pad(
    self,
    text: list[list[int]],  # 待填充的词索引列表
) -> list[list[int]]:  # 返回填充后的词索引列表
    """
    pad word indices to max_length
    """
    pad_text = []
    for _text in text:
        # 如果长度超过max_length 则截断
        if len(_text) > self.max_length:
            pad_text.append(_text[: self.max_length])
        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。

$$P(+|w,c) = \sigma(c \cdot w) \frac{1}{1+\exp(-c \cdot w)}$$

$$P(-|w,c) = 1 - P(+|w,c) = \frac{1}{1+\exp(c \cdot w)} = \sigma(-c \cdot w)$$

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=vecto
r_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_coun
t=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=vecto
r_size)
model = TextCNN(
```

```
    word_embeddings, vector_size, label2index, pad_index, max_length=10
24
).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="<UN
K>"
):
    ...
    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

TextCNN

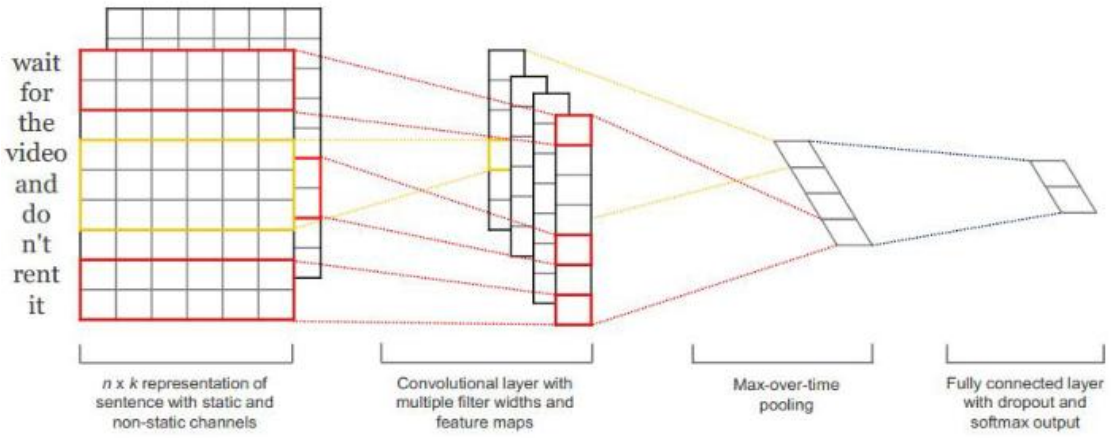


Figure 1: Model architecture with two channels for an example sentence.

`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_id
x=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=channel
s, kernel_size=k)
                for k in filter_size
            ]
        )
        # Final linear layer for label prediction; number of classes eq
uals 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。

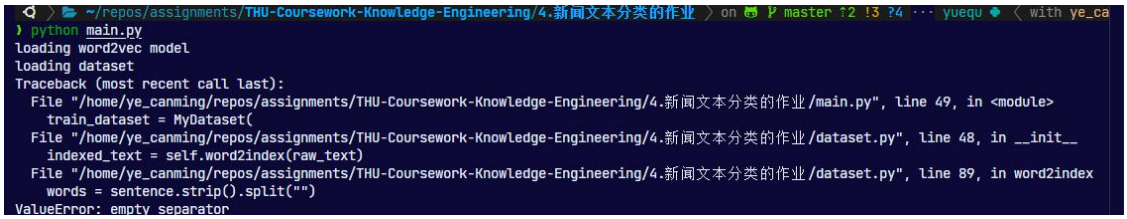
这是因为 sklearn.metrics.precision_recall_fscore_support ，参考文档 https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision

_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



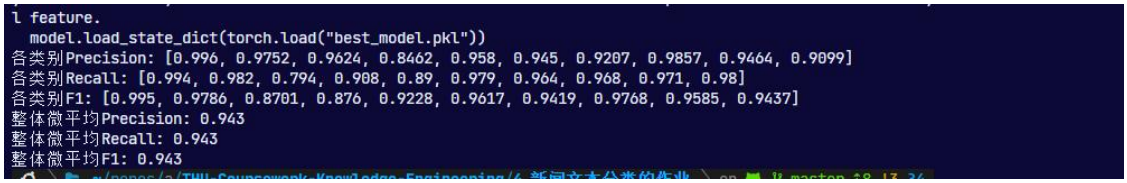
原来 split 写的不好。我们的目的是把 str 变成 char 的列表，在 Java 里面确实经常写 split("")，但是 Python 里面认为这个不对，所以我们要用 python 的方式来写。

```
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。

$$\begin{bmatrix} k_{11} \cdot a_{11} + k_{12} \cdot a_{12} + k_{21} \cdot a_{21} + k_{22} \cdot a_{22} & \cdots & r_{1(p-1)} \\ k_{11} \cdot a_{21} + k_{12} \cdot a_{22} + k_{21} \cdot a_{31} + k_{22} \cdot a_{32} & \cdots & r_{2(p-1)} \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix}$$

(a) Result of Classic Convolution

$$\begin{bmatrix} \phi_{11}(a_{11}) + \phi_{12}(a_{12}) + \phi_{21}(a_{21}) + \phi_{22}(a_{22}) & \cdots & r_{1(p-1)} \\ \phi_{11}(a_{21}) + \phi_{12}(a_{22}) + \phi_{21}(a_{31}) + \phi_{22}(a_{32}) & \cdots & r_{2(p-1)} \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix}$$

(b) Result of KAN Convolution

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

```python
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。