# 《人工智能导论》大作业

任务名称：基于 Distilbert 的谣言检测模型

小组人员：黄俊植 522021910159

完成时间：2025 年 5 月 31 日

# 1．任务目标

基于谣言检测数据集，构建一个检测模型。该模型可以对数据集中的推文进行谣言检测与识别。实现了接口类文件 Classify.py 和训练文件 Train.py。其中接口类为 RumourDetectClass，提供一个接口与函数 Classify()，该函数的输入是一条字符串，输出是一个 int 值（值为对应的预测类别，即整数 0 或 1，0 代表非谣言、1 代表谣言）。

# 2．具体内容

## （1）实施方案

首先查阅资料，得知现在常用于模型检测的模型分为传统机器学习模型：支持向量机 SVM,随机森林 Random Froest,梯度提升树 XGBoost；深度学习模型：RNN/LSTM/GRU；Transformer 架构：BERT 等；以及各种混合集成方法。

对于各种模型的选择问题，我列出了以下表，显示各个模型的优势区间：

| 场景 | 优势模型 |
|---|---|
| 快速部署/有限资源 | TF-IDF + SVM/XGBoost |
| 中等数据量 | CNN/LSTM |
| 追求最高精度 | BERT/RoBERTa |
| 有传播图数据 | GNN + BERT 融合 |
| 多语言场景 | XLM-RoBERTa |

对于谣言检测模型，我们既要求它的分类准确率高，也要求它有合理的运行时间，所以我使用了 BERT 类型中的 Distilbert 模型，它既基于 BERT 架构，保证了模型的准确性，又将其进行精简，使用轻量化的架构：移除了 40%的参数，编码器数量减半，这使它保留了 95%的 BERT 性能，但速度却提升了 60%，非常满足任务需要。

BERT 在识别语义的精度是毋庸置疑的，但训练这种大型 NLP 模型的过程及其漫长，由于其庞大的规模，训练此类模型会持续数天。

Distilbert 通过教师-学生训练框架，将原本于 BERT 中的知识蒸馏出来，在学生（Distilbert）上实现与其相近的功能。**知识蒸馏的好处是显而易见的，因为蒸馏的模型在保证性能的情况下，参数更少、运行得更快、占用的空间更少。**【BERT 模型的知识蒸馏： DistilBERT 方法的理论和机制研究 - 知乎 https://zhuanlan.zhihu.com/p/444629182 】

我们还需要回答一个问题：为什么 Distilbert 适合谣言检测？Distilbert 首先继承了 BERT 的双向上下文理解能力，可以捕捉谣言中微妙的语义特征：如夸张表述，情绪化语言等，而这些是谣言的常见模式，如情感极端化词汇，虚假权威引用，紧急行动号召等；其次 Distilbert 体量小，可以在消费级 GPU（如本电脑的 RTX3060laptop）上部署，可随时随地对谣言进行检测。

在模型的具体实现上，我计划分为 4 个步骤完成：

1. 数据预处理：读取训练集和验证集，对文本进行必要的清洗
2. 编写 Train.py 训练 DistilBERT 模型。

3. 评估模型在验证集上的性能。
4. 保存模型，并编写 Classify.py 接口。

## （2）核心代码分析（Train.py 部分代码解释）

### 1. 自定义数据集类 (RumourDataset)

```
python
class RumourDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=128):
        # 文本清洗：移除控制字符但保留 URL 和话题标签
        cleaned_texts = [re.sub(r'[\x00-\x1F\x7F]', '', str(t)) for t in texts]
        self.encodings = tokenizer(
            cleaned_texts,
            truncation=True,
            padding=True,
            max_length=max_length,
            return_tensors="pt"
        )
        self.labels = labels.tolist()

    def __getitem__(self, idx):
        return {
            'input_ids': self.encodings['input_ids'][idx],
            'attention_mask': self.encodings['attention_mask'][idx],
            'labels': torch.tensor(self.labels[idx])
        }

    def __len__(self):
        return len(self.labels)
```

首先进行文本清洗，我们使用正则表达式[\x00-\x1F\x7F]移除控制字符（如回车、换行等），但保留 URL 和#话题标签等，因为这些可能在社交媒体的文本中有重要的语义。
其次使用文本编码：使用 tokenizer 将文本转换为模型可接受的格式：
truncation=True：截断超过 max_length 的文本
padding=True：填充短于 max_length 的文本
max_length=1024：设置最大序列长度
return_tensors="pt"：返回 PyTorch 张量

### 2. 评估指标计算 (compute_metrics)

```
python
def compute_metrics(pred):
    labels = pred.label_ids
```

```
    preds = pred.predictions.argmax(-1)
    acc = accuracy_score(labels, preds)
    f1 = f1_score(labels, preds)
    return {'accuracy': acc, 'f1': f1}
```

首先从预测结果中提取真实标签(label_ids)和预测类别(argmax(-1))，之后计算关键指标：accuracy_score，f1_score 等
返回字典供 Trainer 使用。

## 3. 训练参数配置 (TrainingArguments)

```
python
training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=8,
    per_device_train_batch_size=64,
    per_device_eval_batch_size=128,
    learning_rate=2e-5,
    warmup_ratio=0.1,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=50,
    eval_strategy='steps',
    eval_steps=200,
    save_strategy='steps',
    save_steps=200,
    load_best_model_at_end=True,
    metric_for_best_model='f1',
    fp16=torch.cuda.is_available(),
    report_to='none'
)
```

参数 值说明
num_train_epochs 8 训练轮次
per_device_train_batch_size 64 训练批次大小
learning_rate 2e-5 经典 BERT 微调学习率
warmup_ratio0.1 前 10%训练步作为预热
eval_strategy 'steps' 按步数而非 epoch 评估
eval_steps 200 每 200 训练步评估一次
metric_for_best_model 'f1' 用 F1 值选择最佳模型
fp16 自动检测 使用混合精度加速训练
load_best_model_at_end True 训练结束时加载最佳模型

## 4. 训练器配置 (Trainer)

```
python
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics
)
```

实现以下功能：
整合模型、参数、数据集和评估方法
自动处理训练循环、评估和模型保存
支持分布式训练和混合精度

## 5. 模型保存与评估

```
python
# 保存最佳模型
model.save_pretrained('rumour_model')
tokenizer.save_pretrained('rumour_model')

# 验证集评估
val_results = trainer.predict(val_dataset)
val_preds = np.argmax(val_results.predictions, axis=-1)

# 输出详细分类报告
print(classification_report(val_df['label'], val_preds))
```

保存模型，输出评估结果。

# （3）核心代码分析（Classify.py 部分代码解释）

## 1. 类初始化 (__init__ 方法)

```
python
def __init__(self, model_path='rumour_model'):
    # 设备配置 (优先 GPU 0 号卡)
    self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    # 加载 tokenizer 和模型
    self.tokenizer = DistilBertTokenizer.from_pretrained(model_path)
    self.model = DistilBertForSequenceClassification.from_pretrained(model_path)

    self.model.to(self.device)
```

```
        self.model.eval()
        print(f"模型加载完成，使用设备: {self.device}")
```

自动检测并使用 GPU（优先 0 号卡），否则使用 CPU。从指定路径加载 tokenizer 和分类模型，使用 from_pretrained 确保兼容训练时保存的格式

模型配置如下：

model.to(device)：将模型移至目标设备（GPU/CPU）

model.eval()：切换到评估模式（关闭 dropout 等训练专用层）

状态反馈：打印加载完成信息和使用的设备

## 2. 文本清洗 (_clean_text)

```
python
def _clean_text(self, text):
    # 基本文本清洗（保留 URL 和话题标签）
    return re.sub(r'[\x00-\x1F\x7F]', '', text)
```

清理范围：使用正则表达式移除控制字符（ASCII 0-31 和 127）

保留重要特征： URL，话题标签（#话题）等

处理空文本：隐式处理 None 类型（通过 str(text)）

## 3. 分类接口 (classify)

```
python
def classify(self, text: str) -> int:
    # 清理文本
    cleaned_text = self._clean_text(text)

    # Tokenize
    inputs = self.tokenizer(
        cleaned_text,
        truncation=True,
        padding=True,
        max_length=512,
        return_tensors="pt"
    ).to(self.device)

    # 推理
    with torch.no_grad():
        outputs = self.model(**inputs)

    # 解析结果
    logits = outputs.logits
    return torch.argmax(logits, dim=-1).item()
```

调用_clean_text 进行基础清洗
确保输入为字符串类型
文本编码：
truncation=True：截断超过 512 个 token 的文本
padding=True：填充短文本（实际单文本可省略）
max_length=512：使用 BERT 最大长度
return_tensors="pt"：返回 PyTorch 张量
.to(self.device)：将输入移至模型所在设备
模型推理：
with torch.no_grad()：禁用梯度计算（节省内存，提升速度）
model(**inputs)：前向传播获取输出
结果解析：
outputs.logits：获取原始预测分数
torch.argmax(dim=-1)：在类别维度取最大值索引
.item()：从单元素张量提取整数值

## （4）调参过程

### 1）.学习率 learning_rate:核心杠杆

```
Learning_rate=1e-5
              precision    recall  f1-score   support

           0       0.80      0.80      0.80       226
           1       0.75      0.75      0.75       179
    accuracy                           0.78       405
   macro avg       0.77      0.78      0.78       405
weighted avg       0.78      0.78      0.78       405
准确率: 0.7778
F1 分数: 0.7500
Learining_rate=2e-5
              precision    recall  f1-score   support

           0       0.82      0.82      0.82       226
           1       0.77      0.77      0.77       179
    accuracy                           0.80       405
   macro avg       0.79      0.79      0.79       405
weighted avg       0.80      0.80      0.80       405
准确率: 0.7975
F1 分数: 0.7697
Learning_rate=3e-5:
              precision    recall  f1-score   support

           0       0.84      0.85      0.84       226
```

```
           1       0.80      0.79      0.80       179
    accuracy                           0.82       405
   macro avg       0.82      0.82      0.82       405
weighted avg       0.82      0.82      0.82       405
准确率: 0.8222
F1 分数: 0.7978
Learning_rate=4e-5
              precision    recall  f1-score   support
           0       0.83      0.87      0.85       226
           1       0.83      0.78      0.80       179
    accuracy                           0.83       405
   macro avg       0.83      0.83      0.83       405
weighted avg       0.83      0.83      0.83       405
准确率: 0.8321
F1 分数: 0.8046
Learning_rate=5e-5
              precision    recall  f1-score   support
           0       0.85      0.88      0.87       226
           1       0.84      0.81      0.83       179
    accuracy                           0.85       405
   macro avg       0.85      0.85      0.85       405
weighted avg       0.85      0.85      0.85       405
准确率: 0.8494
F1 分数: 0.8262
Learning_rate=6e-5
              precision    recall  f1-score   support
           0       0.85      0.90      0.87       226
           1       0.86      0.79      0.83       179
    accuracy                           0.85       405
   macro avg       0.85      0.85      0.85       405
weighted avg       0.85      0.85      0.85       405
准确率: 0.8519
F1 分数: 0.8256
Learning_rate=7e-5
              precision    recall  f1-score   support
           0       0.84      0.88      0.86       226
           1       0.83      0.78      0.81       179
    accuracy                           0.83       405
   macro avg       0.83      0.83      0.83       405
weighted avg       0.83      0.83      0.83       405
准确率: 0.8346
F1 分数: 0.8069
```

从上述测试可以看出，Learing_rate=6e-5 时，准确率最高，训练效果最好。

## 2）.per_device_train_batch_size (训练批次大小) - 梯度质量

per_device_train_batch_size=16

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.88 | 0.85 | 0.86 | 226 |
| 1 | 0.81 | 0.85 | 0.83 | 179 |
| accuracy | | | 0.85 | 405 |
| macro avg | 0.84 | 0.85 | 0.85 | 405 |
| weighted avg | 0.85 | 0.85 | 0.85 | 405 |

准确率: 0.8469
F1 分数: 0.8306
per_device_train_batch_size=32

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.87 | 0.88 | 0.87 | 226 |
| 1 | 0.84 | 0.84 | 0.84 | 179 |
| accuracy | | | 0.86 | 405 |
| macro avg | 0.86 | 0.86 | 0.86 | 405 |
| weighted avg | 0.86 | 0.86 | 0.86 | 405 |

准确率: 0.8593
F1 分数: 0.8403
per_device_train_batch_size=64

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.85 | 0.90 | 0.87 | 226 |
| 1 | 0.86 | 0.79 | 0.83 | 179 |
| accuracy | | | 0.85 | 405 |
| macro avg | 0.85 | 0.85 | 0.85 | 405 |
| weighted avg | 0.85 | 0.85 | 0.85 | 405 |

准确率: 0.8519
F1 分数: 0.8256
per_device_train_batch_size=96

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.85 | 0.85 | 0.85 | 226 |
| 1 | 0.81 | 0.80 | 0.81 | 179 |
| accuracy | | | 0.83 | 405 |
| macro avg | 0.83 | 0.83 | 0.83 | 405 |
| weighted avg | 0.83 | 0.83 | 0.83 | 405 |

准确率: 0.8321
F1 分数: 0.8090
从上述测试可以看出，per_device_train_batch_size=32 时，准确率最高，训练效果最好。

## 3）.warmup_ratio (预热比例) - 稳定训练

warmup_ratio=0.05,

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.86 | 0.85 | 0.86 | 226 |
| 1 | 0.81 | 0.83 | 0.82 | 179 |

| | | | | |
|---|---|---|---|---|
| accuracy | | | 0.84 | 405 |
| macro avg | 0.84 | 0.84 | 0.84 | 405 |
| weighted avg | 0.84 | 0.84 | 0.84 | 405 |

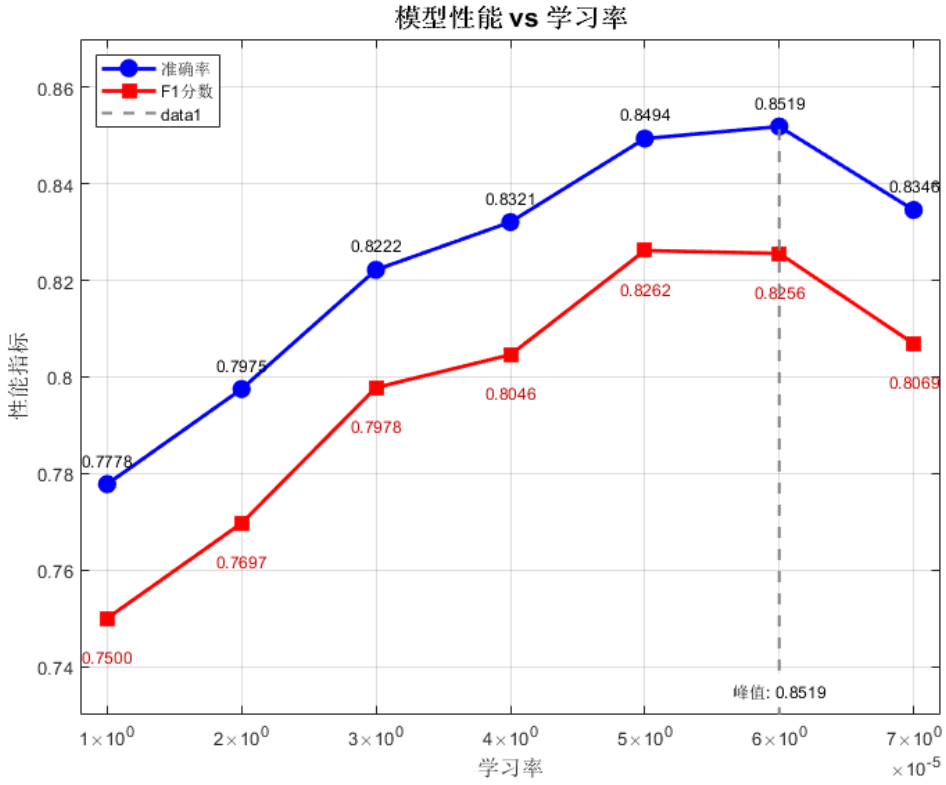
准确率: 0.8420
F1 分数: 0.8232
warmup_ratio=0.1,

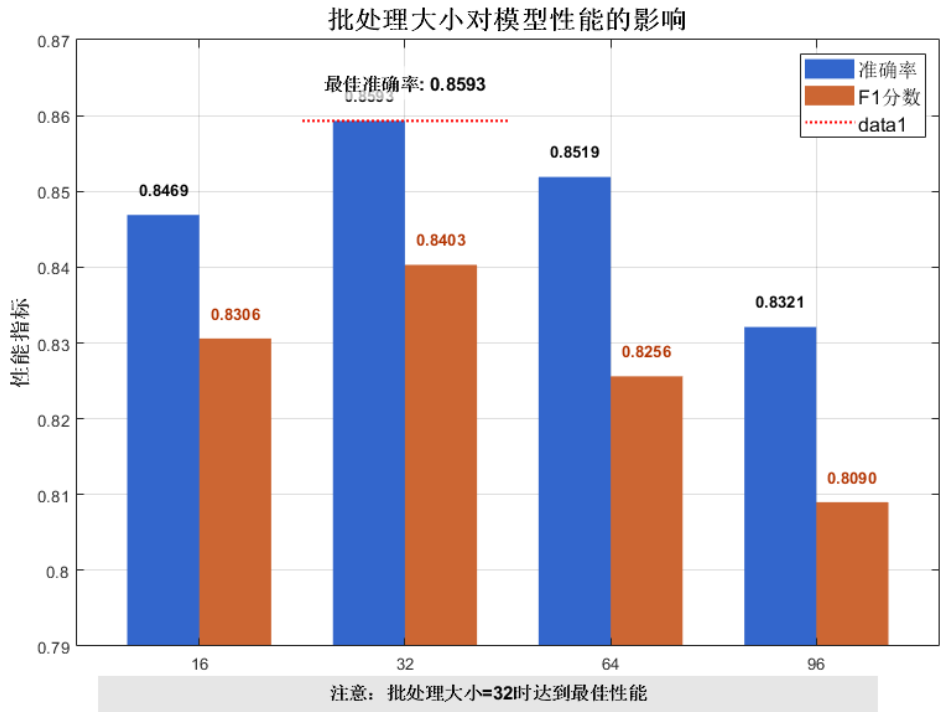
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.89 | 0.86 | 0.88 | 226 |
| 1 | 0.83 | 0.87 | 0.85 | 179 |
| accuracy | | | 0.86 | 405 |
| macro avg | 0.86 | 0.86 | 0.86 | 405 |
| weighted avg | 0.87 | 0.86 | 0.86 | 405 |

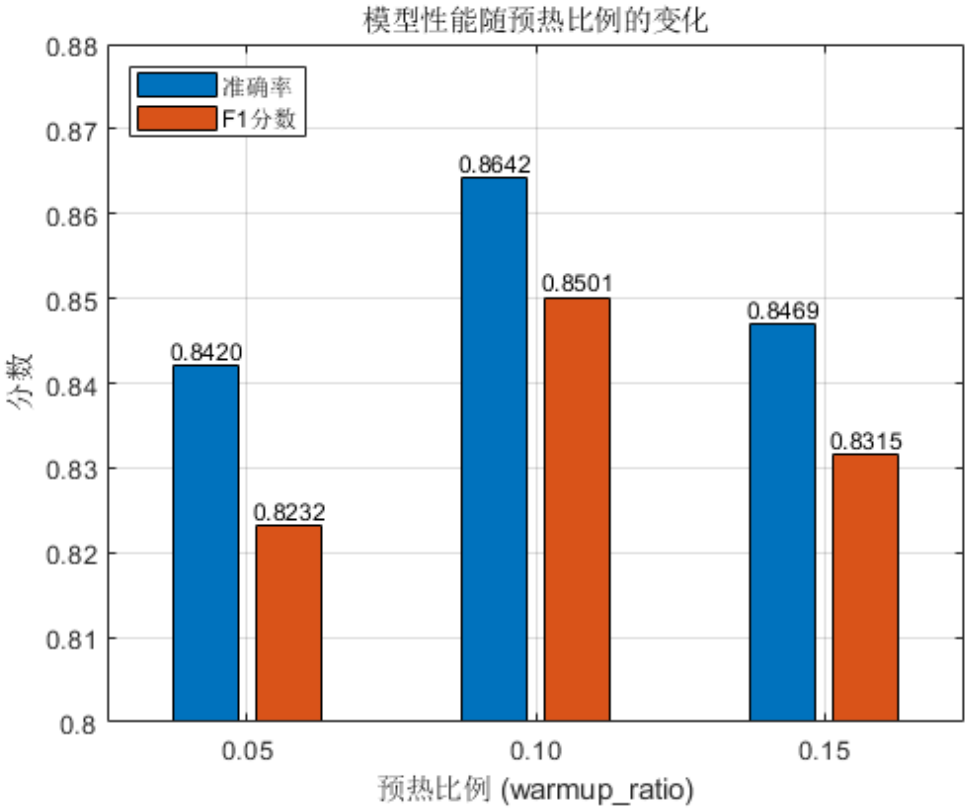
准确率: 0.8642
F1 分数: 0.8501
warmup_ratio=0.15,

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.88 | 0.84 | 0.86 | 226 |
| 1 | 0.81 | 0.85 | 0.83 | 179 |
| accuracy | | | 0.85 | 405 |
| macro avg | 0.84 | 0.85 | 0.85 | 405 |
| weighted avg | 0.85 | 0.85 | 0.85 | 405 |

准确率: 0.8469
F1 分数: 0.8315
从上述测试可以看出，warmup_ratio=0.1 时，准确率最高，训练效果最好。
综上所述，最优参数是：
Learing_rate=6e-5，per_device_train_batch_size=32，warmup_ratio=0.1。
以下是参数对模型性能的影响图：


模型性能 vs 学习率
准确率
F1分数
data1
性能指标
学习率
峰值: 0.8519
0.7778
0.7975
0.8222
0.8321
0.8494
0.8519
0.8346
0.7500
0.7697
0.7978
0.8046
0.8262
0.8256
0.8069


批处理大小对模型性能的影响
最佳准确率: 0.8593
准确率
F1分数
data1
性能指标
0.8469
0.8306
0.8593
0.8403
0.8519
0.8256
0.8321
0.8090
16
32
64
96
注意：批处理大小=32时达到最佳性能




## 3. 工作总结

### （1）收获、心得

这是我第一次尝试解决人工智能方面的问题，有许多的感受和收获。首先由于我曾经是其他学院的学生，并没有学习过信安的专业课，也只参与了一些其他语言（C++，Java）的项目工作，所以对于 Python 的使用方面，我相当不熟练，这也给我带来了很大的困难，但是在遇到问题，解决问题的过程中，我逐渐对代码的编写得心应手了起来，同时也发现曾经编写其他项目的经验和这里都可以相通，这些都支撑着我完成了整段代码的编写工作，也让我收获到了许多之前没有的宝贵知识，经验。

我在代码编写的过程中尝试了许多模型，比如 LSMT 模型和 BERT 模型等，这些也让我学习到了关于人工智能模型的知识，以及各个模型的适用场景以及优势区间，也让我学习到了一些人工智能模型的设计思路以及调参的思路。

通过这次大作业，我也意识到了人工智能在各行各业的重要意义，尤其是在对抗谣言方面，通过技术创新为信息真实性验证提供了全新解决方案。随着多模态大模型的发展，未来的防谣系统将具备更接近人类的语境理解和逻辑推理能力，在保障言论自由的同时，构筑数字时代的"信息免疫系统"。

但同时，我也认识到了人工智能目前的很多局限性，比如我在识别“鸡是鸭子生的”这样一个命题时，许多模型都给出了 TRUE 的判断，这表明许多的人工智能模型依旧不具备完备性或完好性，需要我们谨慎使用，也要推动它不断进步。

### （2）遇到问题及解决思路

我遇到的第一个困难是在模型的选择上，我首先选择的模型并非 Distilbert

模型，而是 LSMT 模型，但是在实际训练的情况下，我的正确率一直在 55%上下浮动，在加入更多层和注意力机制下依然没有多少改善（在附录会附上第一次的代码）。之后才选择了精确度更高的 BERT 类模型。

其次在代码的编写上，我之前是其他专业的学生，并没有学过信安的多少专业课，曾经也没有使用 Python 的经验（之前的专业项目一直都是 C++或 Java 的代码），所以编写代码的过程显得比较困难，但是在学习他人代码以及 LLM 的帮助下，我也是顺利完成了代码的编写工作，同时，过去编写其他项目的编程经验以及习惯也帮助我扫清了很多障碍，其实思路都是相通的。

在训练模型的时候，参数的选择十分重要，而调整参数的过程却比较痛苦。我使用的方法是分段画图，首先找出参数效果最好的区段，然后在该区段二分的步骤，确定了一些较优的参数设置。

我在训练的时候还遇到了 GPU 占用低的问题，经过检查后发现，我下载的 Torch 版本为 CPU，在重新更换 GPU 版的 Torch 后训练时间由 12 分钟压缩到了 20 秒，有了质的飞跃。

## 4. 课程建议

老师上课讲的很好，希望可以多增加一些实践机会（比如像大作业这种类似的作业），我感觉这次大作业对我的成长帮助极大。再次感谢两位老师以及助教们的付出，让我有了很大的收获。

## 5.附录

## 5.1 首次尝试失败的 LSMT 模型代码（train.py）

```
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from sklearn.metrics import accuracy_score, f1_score, classification_report, confusion_matrix
from torch.utils.data import Dataset, DataLoader
import re
import os
import time
from tqdm import tqdm
import platform
from collections import Counter
import math
from torch.optim.lr_scheduler import ReduceLROnPlateau

# 设置环境变量
```

```
os.environ['TOKENIZERS_PARALLELISM'] = 'false'

class RumourDataset(Dataset):
    def __init__(self, texts, labels, vocab, max_length=128):
        self.texts = texts
        self.labels = labels
        self.vocab = vocab
        self.max_length = max_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        # 改进文本清洗：保留特殊符号（URL、话题标签）
        cleaned_text = re.sub(r'[^\x20-\x7E]', '', str(text))  # 仅保留可打印 ASCII 字符
        # 分词并转换为索引
        tokens = cleaned_text.split()
        indices = [self.vocab.get(token, self.vocab['<unk>']) for token in tokens[:self.max_length]]

        # 填充序列
        padded_indices = indices + [self.vocab['<pad>']] * (self.max_length - len(indices))
        attention_mask = [1] * len(indices) + [0] * (self.max_length - len(indices))

        return {
            'input_ids': torch.tensor(padded_indices, dtype=torch.long),
            'attention_mask': torch.tensor(attention_mask, dtype=torch.long),
            'labels': torch.tensor(self.labels[idx], dtype=torch.long)
        }

class PositionalEncoding(nn.Module):
    #位置编码
    def __init__(self, d_model, dropout=0.1, max_len=128):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x):
```

```
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)

class ImprovedLSTMClassifier(nn.Module):
    加入更多层和注意力机制
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers, dropout):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.pos_encoder = PositionalEncoding(embedding_dim, dropout)

        # 双向 LSTM
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=n_layers,
                            bidirectional=True, batch_first=True, dropout=dropout)

        # 自注意力机制
        self.attention = nn.MultiheadAttention(embedding_dim, num_heads=4, dropout=dropout)

        # 更深的分类器
        self.fc1 = nn.Linear(hidden_dim * 2, hidden_dim)  # 双向 LSTM 输出维度是 2 倍
        self.fc2 = nn.Linear(hidden_dim, hidden_dim // 2)
        self.fc3 = nn.Linear(hidden_dim // 2, output_dim)

        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(embedding_dim)

    def forward(self, text, attention_mask=None):
        embedded = self.embedding(text)
        embedded = self.pos_encoder(embedded)

        # 应用注意力机制
        embedded = embedded.permute(1, 0, 2)  # [seq_len, batch_size, emb_dim]
        attn_output, _ = self.attention(embedded, embedded, embedded)
        embedded = embedded + attn_output  # 残差连接
        embedded = self.layer_norm(embedded)
        embedded = embedded.permute(1, 0, 2)  # 恢复维度

        # LSTM 处理
        lstm_output, (hidden, cell) = self.lstm(embedded)

        # 使用最后一个时间步的输出
        output = self.dropout(lstm_output[:, -1, :])

        # 多层分类器
        output = F.relu(self.fc1(output))
```

```
        output = self.dropout(output)
        output = F.relu(self.fc2(output))
        output = self.dropout(output)
        output = self.fc3(output)

        return output

def build_vocab(texts, min_freq=1, special_tokens=None):
    #构建词汇表，降低词频阈值
    counter = Counter()
    for text in texts:
        cleaned_text = re.sub(r'[^\x20-\x7E]', '', str(text))
        tokens = cleaned_text.split()
        counter.update(tokens)

    # 创建词汇表
    vocab = {'<pad>': 0, '<unk>': 1}
    if special_tokens:
        for token in special_tokens:
            vocab[token] = len(vocab)

    for token, count in counter.items():
        if count >= min_freq:
            vocab[token] = len(vocab)

    return vocab

def train_model(model, dataloader, optimizer, criterion, device):
    model.train()
    total_loss, total_acc = 0, 0

    for batch in tqdm(dataloader, desc="Training"):
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        optimizer.zero_grad()
        predictions = model(input_ids, attention_mask)
        loss = criterion(predictions, labels)

        loss.backward()
        # 梯度裁剪防止爆炸
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
```

```
            total_loss += loss.item()
            total_acc += (predictions.argmax(1) == labels).sum().item()

    return total_loss / len(dataloader), total_acc / len(dataloader.dataset)

def evaluate_model(model, dataloader, criterion, device):
    model.eval()
    total_loss, total_acc = 0, 0
    all_preds, all_labels = [], []

    with torch.no_grad():
        for batch in tqdm(dataloader, desc="Evaluating"):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            predictions = model(input_ids, attention_mask)
            loss = criterion(predictions, labels)

            total_loss += loss.item()
            total_acc += (predictions.argmax(1) == labels).sum().item()
            all_preds.extend(predictions.argmax(1).cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    return (
        total_loss / len(dataloader),
        total_acc / len(dataloader.dataset),
        np.array(all_preds),
        np.array(all_labels)
    )

def main():
    # 加载数据
    train_df = pd.read_csv('train.csv')
    val_df = pd.read_csv('val.csv')

    # 检查数据平衡性
    train_pos = train_df['label'].sum()
    train_neg = len(train_df) - train_pos
    val_pos = val_df['label'].sum()
    val_neg = len(val_df) - val_pos

    print(f"训练集: {len(train_df)}条，谣言比例: {train_df['label'].mean():.2f}")
```

```
print(f"   - 谣言: {train_pos}条，非谣言: {train_neg}条")
print(f"验证集: {len(val_df)}条，谣言比例: {val_df['label'].mean():.2f}")
print(f"   - 谣言: {val_pos}条，非谣言: {val_neg}条")

# 处理类别不平衡 - 计算类别权重
class_weights = [1.0, float(train_neg) / train_pos]  # 增加少数类权重

# 构建词汇表（降低词频阈值）
special_tokens = ['http', 'https', 'www', 'com', 'rt', '@user', '#hashtag']
vocab = build_vocab(train_df['text'], min_freq=1, special_tokens=special_tokens)
print(f"词汇表大小: {len(vocab)}")

# 保存词汇表
torch.save(vocab, 'vocab.pth')

# 创建数据集
train_dataset = RumourDataset(train_df['text'].values, train_df['label'].values, vocab)
val_dataset = RumourDataset(val_df['text'].values, val_df['label'].values, vocab)

# 设备配置
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"使用设备: {device}")

# 根据操作系统设置数据加载 workers
num_workers = 0 if platform.system() == 'Windows' else 4

# 创建数据加载器
train_loader = DataLoader(
    train_dataset,
    batch_size=64,  # 减小批次大小以提升精度
    shuffle=True,
    num_workers=num_workers,
    pin_memory=True
)
val_loader = DataLoader(
    val_dataset,
    batch_size=128,
    num_workers=num_workers,
    pin_memory=True
)

model = ImprovedLSTMClassifier(
    vocab_size=len(vocab),
    embedding_dim=256,  # 增加嵌入维度
```

```
        hidden_dim=256,        # 增加隐藏单元
        output_dim=2,
        n_layers=3,            # 增加 LSTM 层数
        dropout=0.4            # 适当增加 dropout 防止过拟合
).to(device)

# 优化器和损失函数 - 添加类别权重处理不平衡
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0005, weight_decay=1e-4)
criterion = nn.CrossEntropyLoss(weight=torch.tensor(class_weights).to(device))

# 学习率调度器
scheduler = ReduceLROnPlateau(
        optimizer,
        mode='max',
        factor=0.5,
        patience=2,
        verbose=True
)

# 训练参数
num_epochs = 20  # 增加训练轮数
best_f1 = 0
train_losses, val_losses = [], []
train_accs, val_accs = [], []

# 早停参数
early_stop_patience = 4
no_improve_count = 0

# 训练循环
for epoch in range(num_epochs):
        start_time = time.time()

        # 训练
        train_loss, train_acc = train_model(model, train_loader, optimizer, criterion, device)
        train_losses.append(train_loss)
        train_accs.append(train_acc)

        # 验证
        val_loss, val_acc, val_preds, val_labels = evaluate_model(model, val_loader, criterion, device)
        val_losses.append(val_loss)
        val_accs.append(val_acc)

        # 计算 F1 分数
```

```
        f1 = f1_score(val_labels, val_preds)

        # 更新学习率
        scheduler.step(f1)

        epoch_time = time.time() - start_time

        print(f"\nEpoch {epoch+1}/{num_epochs} | Time: {epoch_time:.1f}s")
        print(f"Train Loss: {train_loss:.4f} | Train Acc: {train_acc:.4f}")
        print(f"Val Loss: {val_loss:.4f} | Val Acc: {val_acc:.4f} | F1: {f1:.4f}")

        # 打印混淆矩阵
        cm = confusion_matrix(val_labels, val_preds)
        print("混淆矩阵:")
        print(f"TN: {cm[0,0]} | FP: {cm[0,1]}")
        print(f"FN: {cm[1,0]} | TP: {cm[1,1]}")

        # 保存最佳模型
        if f1 > best_f1:
            best_f1 = f1
            no_improve_count = 0
            torch.save({
                'model_state_dict': model.state_dict(),
                'vocab_size': len(vocab),
                'embedding_dim': 256,
                'hidden_dim': 256,
                'output_dim': 2,
                'n_layers': 3,
                'dropout': 0.4,
                'class_weights': class_weights
            }, 'best_lstm_model.pth')
            print(f"最佳模型已保存，F1: {f1:.4f}")
        else:
            no_improve_count += 1
            print(f"F1 未提升，早停计数器: {no_improve_count}/{early_stop_patience}")

            # 早停检查
            if no_improve_count >= early_stop_patience:
                print(f"验证集 F1 在{early_stop_patience}个 epoch 内未提升，停止训练")
                break

# 最终评估
print("\n 最终评估结果:")
print(classification_report(val_labels, val_preds))
```

```
    print(f"准确率: {accuracy_score(val_labels, val_preds):.4f}")
    print(f"F1 分数: {f1_score(val_labels, val_preds):.4f}")

    # 文本方式输出训练统计
    print("\n 训练统计:")
    print("Epoch | Train Loss | Train Acc | Val Loss | Val Acc | Val F1")
    for epoch in range(len(train_losses)):
                print(f"{epoch+1:5d}  |  {train_losses[epoch]:10.4f}  |  {train_accs[epoch]:9.4f}  |
{val_losses[epoch]:8.4f} | {val_accs[epoch]:7.4f} | {f1_score(val_labels, val_preds):.4f}")

if __name__ == "__main__":
    main()
```