机器学习实验报告

实验二：新闻文本分类

一、 实验内容

实验以对脱敏处理后的新闻文本数据进行类别分类为主要任务，数据文件组织格式为：train.csv（训练数据）、testA.csv（测试数据），其中训练数据为 20 万条标注的文本数据，测试数据为 5 万条未标注的文本数据。脱敏处理将文本数据中的所有文字替换为数字，原始文本与数字的映射关系被隐藏。实验要求使用 BERT 或 BERT 系列的模型对这些脱敏新闻文本数据进行分类，保存对测试数据集的推断结果，并上传至阿里天池平台进行实验结果评估。

字段表：

训练集：

1. 字段 label：对每一行文本数据的类别标注，共 14 个候选分类类别：财经、彩票、房产、股票、家居、教育、科技、社会、时尚、时政、体育、星座、游戏、娱乐。
2. 字段 text：脱敏的新闻文本，每一行代表一条文本，文本按照字符级别进行了匿名处理。

测试集：

字段 text：脱敏的新闻文本，每一行代表一条文本 ，文本按照字符级别进行了匿名处理。

训练数据格式如下：

label	text
6	57 44 66 56 2 3 3 37 5 41 9 57 44 47 45 33 13 63 58 31 17 47 0 1 1 69 26 60 62 15 21 12 49 18 38 20 50 23 57 44 45 33 25 28 47 22 52 35 30 14 24 69 54 7 48 19 11 51 16 43 26 34 53 27 64 8 4 42 36 46 65 69 29 39 15 37 57 44 45 33 69 54 7 25 40 35 30 66 56 47 55 69 61 10 60 42 36 46 65 37 5 41 32 67 6 59 47 0 1 1 68

结果提交：

需提交对测试数据集每一条文本进行 14 种类别分类的结果，label 编号为 0 至 13 的整数，每个整数对应一个类别，将推断结果保存为 CSV 文件，CSV 文件应只有一个 label 字段，每一行代表测试数据对应行文本的分类结果。

评测标准：

评价标准为类别 F1 值的均值，提交结果与实际测试集的类别进行对比，结果越大越好。F1 值的计算公式为：

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F\text{-}score = \frac{2Precision * Recall}{Precision + Recall}$$

实验要求：

1. 使用 NumPy、Pandas、Matplotlib 等工具对数据进行分析。
2. 根据数据的特点设计合理的数据预处理方案。
3. 使用 BERT 或 BERT 系列的模型完成文本分类任务。
4. 训练或微调模型并进行超参数调整。
5. 用最终模型生成预测结果并保存。
6. 上传最终预测结果的 CSV 文件获取竞赛测评、排名结果。
7. 根据实验内容编写实验报告。

二、 实验环境

代码格式：Jupyter Notebook（ipynb 格式文件）

运行环境：AutoDL GPU 租赁平台 JupyterLab IDE

Python 版本：Python 3.8

GPU: RTX A5000 24G

CUDA 版本: 11.3

安装包版本：

- PyTorch 1.11.0
- Numpy 1.22.4
- Pandas 1.5.3
- Seaborn 0.12.2
- Matplotlib 3.7.1
- (Hugging Face) transformers 4.29.2
- (Hugging Face) datasets 2.12.0
- (Hugging Face) evaluate 0.2.2
- (Hugging Face) accelerate 0.17.1

三、 实验步骤

（一） 加载数据集

使用 datasets 库的 load_dataset 函数加载远程服务器上的数据集，组织为 Hugging Face Dataset 的标准格式，训练集与测试集的结构如下：

```
[ ]  train_data
```

```
DatasetDict({
    train: Dataset({
        features: ['label', 'text'],
        num_rows: 200000
    })
})
```

```
[ ]  test_data
```

```
DatasetDict({
    train: Dataset({
        features: ['text'],
        num_rows: 50000
    })
})
```

训练集与测试集都为 CSV 格式文件，训练集有两个字段：label 表示每条文本的分类标注，text 表示每一条经过脱敏处理的文本。测试集只有 text 字段，与训练集的 text 字段定义相同。

（二） 数据集探索

1) 数据格式转换、查看数据集

使用 set_format 方法将 Dataset 类型的对象转换为 DataFrame 对象以便于进行数据探索，格式转换完成后查看数据集：

```
[ ]  # 训练集头5条数据
     train_df.head()
```

	label	text
0	2	2967 6758 339 2021 1854 3731 4109 3792 4149 15...
1	11	4464 486 6352 5619 2465 4802 1452 3137 5778 54...
2	3	7346 4068 5074 3747 5681 6093 1777 2226 7354 6...
3	2	7159 948 4866 2109 5520 2490 211 3956 5520 549...
4	3	3646 3055 3055 2490 4659 6065 3370 5814 2465 5...

```
[ ]  #  测试集头5条数据
     test_df.head()
```

	text
0	5399 3117 1070 4321 4568 2621 5466 3772 4516 2...
1	2491 4109 1757 7539 648 3695 3038 4490 23 7019...
2	2673 5076 6835 2835 5948 5677 3247 4124 2465 5...
3	4562 4893 2210 4761 3659 1324 2595 5949 4583 2...
4	4269 7134 2614 1724 4464 1324 3370 3370 2106 2...

可见训练集与测试集的 text 字段都为空格与数字分隔的字符串，训练集的 label 字段为 0 到 13 的整数，对应 14 种分类类别。

2） 统计文本长度，查看数据集描述

```
[ ]  train_df.head()
```

	label	text	length
0	2	2967 6758 339 2021 1854 3731 4109 3792 4149 15...	1057
1	11	4464 486 6352 5619 2465 4802 1452 3137 5778 54...	486
2	3	7346 4068 5074 3747 5681 6093 1777 2226 7354 6...	764
3	2	7159 948 4866 2109 5520 2490 211 3956 5520 549...	1570
4	3	3646 3055 3055 2490 4659 6065 3370 5814 2465 5...	307

```
[ ]  test_df.head()
```

	text	length
0	5399 3117 1070 4321 4568 2621 5466 3772 4516 2...	607
1	2491 4109 1757 7539 648 3695 3038 4490 23 7019...	1861
2	2673 5076 6835 2835 5948 5677 3247 4124 2465 5...	901
3	4562 4893 2210 4761 3659 1324 2595 5949 4583 2...	3786
4	4269 7134 2614 1724 4464 1324 3370 3370 2106 2...	316

```
[ ]  train_df.describe()
```

	label	length
count	200000.000000	200000.000000
mean	3.210950	907.207110
std	3.084955	996.029036
min	0.000000	2.000000
25%	1.000000	374.000000
50%	2.000000	676.000000
75%	5.000000	1131.000000
max	13.000000	57921.000000

```
[ ]  test_df.describe()
```

	length
count	50000.000000
mean	909.844960
std	1032.313375
min	14.000000
25%	370.000000
50%	676.000000
75%	1133.000000
max	41861.000000

- 训练集共 200,000 条新闻，每条新闻平均 907 个字符，最短的句子长度为 2,最长的句子长度为 57921,其中 75%以下的数据长度在 1131 以下。
- 测试集共 50,000 条新闻，每条新闻平均 909 个字符，最短句子长度为 14，最长句子 41861,75%以下的数据长度在 1133 以下。

3） 查看文本数据的长度分布

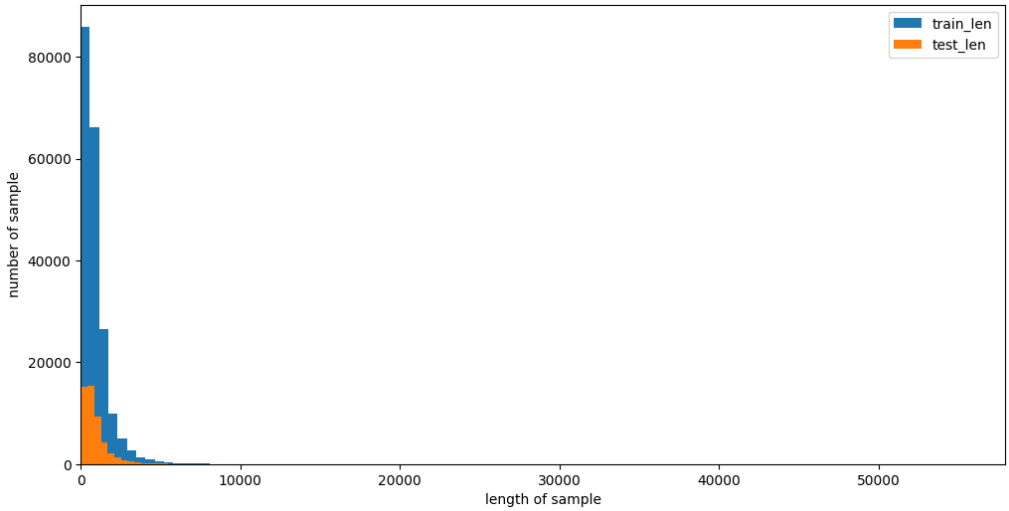


图 1 长度分布直方图

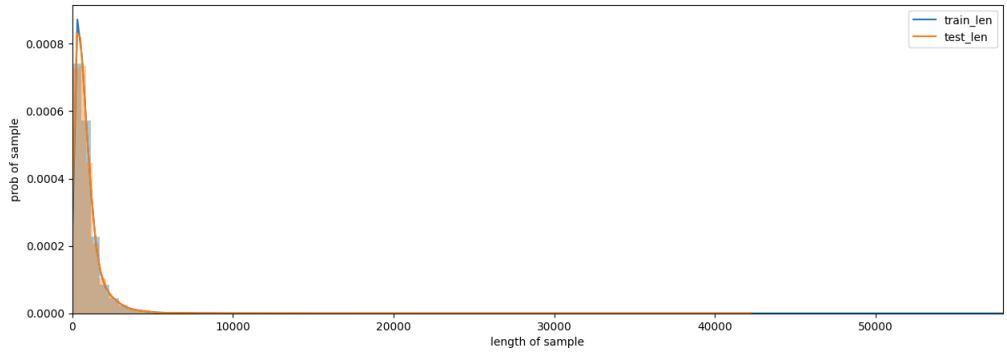


图 2 长度分布频率直方图

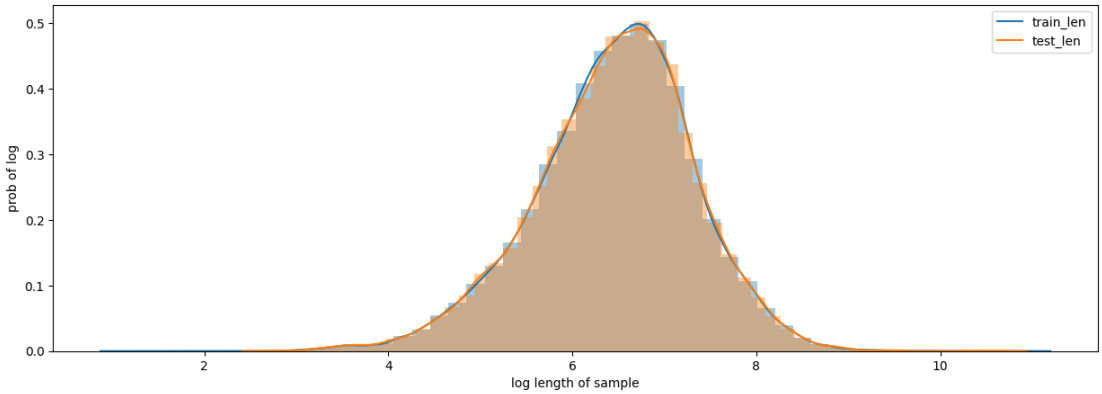


图 3 截断后长度分布图

由此可见训练集和测试集在长度上近似同一分布，经过 Scipy 的拟合优度检验 p 值为 0，并不是一个正态分布。

4) 查看类别分布情况

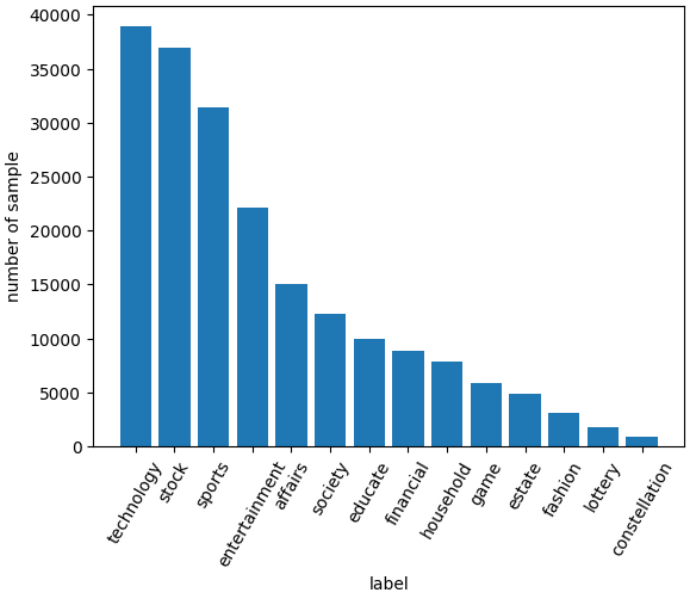


图 3 类型分布直方图

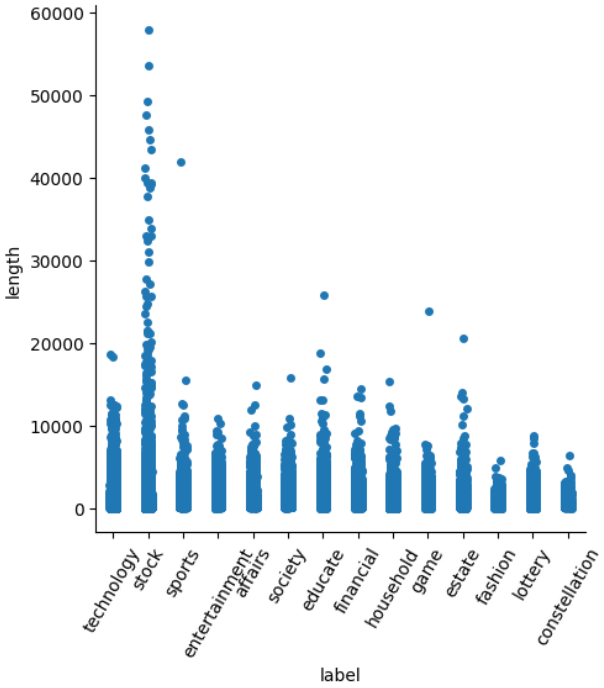


图 4 类型-文本长度分布图

- 数据集类别分布存在较为不均匀的情况。在训练集中科技类新闻最多，其次是股票类新闻，最少的新闻是星座新闻。
- 不同类别的新闻文本长度不同，尤其是股票类新闻，出现了一些极长的新闻文本，其余类型的新闻文本的长度分布相近。
- 由于类别不均衡，可能会严重影响模型的精度。

总结：

1. 训练集共 200,000 条新闻，每条新闻平均 907 个字符，最短的句子长度为 2,最长的句子长度为 57921,其中 75%以下的数据长度在 1131 以下。测试集共 50,000 条新闻，每条新闻平均 909 个字符，最短句子长度为 14，最长句子 41861,75%以下的数据长度在 1133 以下。
2. 训练集和测试集在长度近似满足同一分布，但不是正态分布
3. 数据集类别分布存在较为不均匀的情况。在训练集中科技类新闻最多，其次是股票类新闻，最少的新闻是星座新闻。
4. 数据集使用的是脱敏数据，原文本的上下文联系被破坏，且与各种文本类型的预训练模型使用的训练语料有很大的不同，因此需要对数据集进行预处理再作为预训练模型的输入进行微调任务。

（三） 模型前置训练：掩码语言建模（MLM）任务

由于本次实验使用的数据集都是脱敏处理后的数据集，所有的原始文本被替换为数字，每一个数字替代原始文本的一个字符，因此原始文本的上下文关系可能有一定程度的破坏，针对 MLM 任务进行训练能够更具针对性地捕捉训练数据的上下文关系。BERT 等预训练模型使用的训练语料都是基于文本的，与本次实验所使用的训练数据存在很大的不同，因此不能直接将其作为预训练模型的输入进行文本分类的微调任务。除此之外，BERT 等预训练模型所使用的分词器也是基于大规模的文本语料进行训练的，对于多个数字组成的文本可能并不能正确分词，因此还需要针对本次实验所使用的训练数据进行统计，重新训练一个分词器以建立新的词表和映射关系。 以下所有实验步骤均使用 Hugging Face API，分词器与模型分别加载互不影响。

本次实验选用的模型为：distilbert-base-uncased，即 BERT 模型的蒸馏版本模型 DistilBERT。DistilBERT 模型保留了 BERT 模型 97%的性能，但模型大小下降了 40%，运算速度提高了 60%，可以保证显存的消耗和算力的要求在一个可接受的范围内。Hugging Face Trainer 默认使用 AdamW 优化器。

本次实验没有对模型结构进行任何改动，用于 MLM 任务的 DistilBERT 模型结构过长，具体模型结构图在同目录下 img 文件夹中。

1） 重新训练分词器 Tokenizer：

```
[ ]  #  将训练集组织为训练词袋
     def  get_training_corpus():
             return  (
                     train_data["train"][i  :  i  +  1000]["text"]
                     for  i  in  range(0,  len(train_data["train"]),  1000)
             )


     training_corpus  =  get_training_corpus()
```

```
[ ]  #  分词器未训练时的分词结果
     tokens  =  tokenizer.tokenize(example)
     tokens[:10]
```

```
['296', '##7', '67', '##58', '339', '2021', '1854', '37', '##31', '410']
```

```
[ ]  #  训练分词器，因为脱敏数据中的数字大部分没有超过10000，所以限定词表大小为10000
     tokenizer  =  tokenizer.train_new_from_iterator(training_corpus,  10000)
```

```
[ ]  #  训练分词器后
     tokens  =  tokenizer.tokenize(example)
     tokens[:10]
```

```
['2967', '6758', '339', '2021', '1854', '3731', '4109', '3792', '4149', '1519']
```

2） 预处理数据集：

```
[ ]  #  分词函数
     def  tokenize_function(examples):
             result  =  tokenizer(examples["text"])
             if  tokenizer.is_fast:
                     result["word_ids"]  =  [result.word_ids(i)  for  i  in  range(len(result["input_ids"]))]
             return  result
```

```
[ ]  #  对数据集进行分词
     tokenized_datasets  =  train_data.map(
             tokenize_function,  batched=True,  remove_columns=["text",  "label"]
     )
     tokenized_datasets
```

数据集的 text 字段的文本数据在经过分词处理后不需要作为模型的输入，对于 MLM 任务，数据集的 label 字段也是不需要的，因此在数据集的 map 过程中可将它们丢弃减少显存消耗。由于使用的 DistilBERT 模型的可接受最大输入长度为 512，而本次实验中使用的数据平均长度都超过了 512，为了尽可能地保留大部分的信息，可以使用滑动窗口重叠数据，对原始数据集进行分块处理。

```
[ ] # 对数据进行分块的大小
    chunk_size = 128
```

```
[ ] # 滑动窗口，数据重叠进行分块
    def group_texts(examples):
        # Concatenate all texts
        concatenated_examples = {k: sum(examples[k], []) for k in examples.keys()}
        # Compute length of concatenated texts
        total_length = len(concatenated_examples[list(examples.keys())[0]])
        # We drop the last chunk if it's smaller than chunk_size
        total_length = (total_length // chunk_size) * chunk_size
        # Split by chunks of max_len
        result = {
            k: [t[i : i + chunk_size] for i in range(0, total_length, chunk_size)]
            for k, t in concatenated_examples.items()
        }
        # Create a new labels column
        result["labels"] = result["input_ids"].copy()
        return result
```

```
[ ] # 准备好进行MLM训练的数据集
    lm_datasets = tokenized_datasets.map(group_texts, batched=True)
    lm_datasets
```

```
Map:   0%|          | 0/140000 [00:00<?, ? examples/s]
Map:   0%|          | 0/60000 [00:00<?, ? examples/s]
DatasetDict({
    train: Dataset({
        features: ['input_ids', 'attention_mask', 'word_ids', 'labels'],
        num_rows: 993138
    })
    validation: Dataset({
        features: ['input_ids', 'attention_mask', 'word_ids', 'labels'],
        num_rows: 427397
    })
})
```

在进行分块操作后原始数据集将发生膨胀，为了使训练时长在可接受的范围内的同时不损失过多的性能，可以对膨胀后的数据集进行降采样到原始数据集的大小，作为最终的 MLM 任务训练数据集。

```
wwm_probability = 0.2


def whole_word_masking_data_collator(features):
    for feature in features:
        word_ids = feature.pop("word_ids")

        # Create a map between words and corresponding token indices
        mapping = collections.defaultdict(list)
        current_word_index = -1
        current_word = None
        for idx, word_id in enumerate(word_ids):
            if word_id is not None:
                if word_id != current_word:
                    current_word = word_id
                    current_word_index += 1
                mapping[current_word_index].append(idx)

        # Randomly mask words
        mask = np.random.binomial(1, wwm_probability, (len(mapping),))
        input_ids = feature["input_ids"]
        labels = feature["labels"]
        new_labels = [-100] * len(labels)
        for word_id in np.where(mask)[0]:
            word_id = word_id.item()
            for idx in mapping[word_id]:
                new_labels[idx] = labels[idx]
                input_ids[idx] = tokenizer.mask_token_id
        feature["labels"] = new_labels

    return default_data_collator(features)
```

```
# 下采样数据集，达到与之前的训练集相同的数据量
train_size = 200000
test_size = int(0.1 * train_size)

downsampled_dataset = lm_datasets["train"].train_test_split(
    train_size=train_size, test_size=test_size,
)
downsampled_dataset
```

```
DatasetDict({
    train: Dataset({
        features: ['input_ids', 'attention_mask', 'word_ids', 'labels'],
        num_rows: 200000
    })
    test: Dataset({
        features: ['input_ids', 'attention_mask', 'word_ids', 'labels'],
        num_rows: 20000
    })
})
```

3） 加载训练所需组件，进行 MLM 任务训练：

使用 Hugging Face API 的 Trainer 进行训练，需要加载数据整理器，评估指标，分词器和对应模型。MLM 任务在训练过程中不需要进行特殊的指标计算。

```
[ ] # 定义数据整理器
    from transformers import DataCollatorForLanguageModeling

    data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm_probability=0.15)    # 掩码进行词元遮掩的概率
```

```
[ ] # 加载模型用于MLM任务
    from transformers import AutoModelForMaskedLM
    model = AutoModelForMaskedLM.from_pretrained(model_checkpoint)
```

定义 Trainer 使用的超参数，开启训练：

训练超参数：

```
[ ] # 定义Trainer使用的参数
    from transformers import TrainingArguments

    batch_size = 64
    model_name = model_checkpoint.split("/")[-1]

    training_args = TrainingArguments(
            output_dir=f"{model_name}-finetuned-MLM",
            num_train_epochs=10,
            overwrite_output_dir=True,
            evaluation_strategy="epoch",    # 每个epoch后推断
            learning_rate=2e-5,
            weight_decay=0.01,    # 每层权重衰减值
            save_total_limit=2,    # 训练时保存的最大模型数量
            save_strategy="epoch",    # 每个epoch后保存模型
            per_device_train_batch_size=batch_size,    # 训练数据的batch size
            per_device_eval_batch_size=batch_size,
            push_to_hub=False,    # 不把模型推送到Hub
            fp16=True,    # 开启fp16加速
            logging_strategy="epoch",    # 每个epoch输出日志
            logging_dir="/root/tf-logs",    # 日志目录，AutoDL平台需要指定该目录以供TensorBoard使用
    )
```

```
[ ] from transformers import Trainer
    # 定义Trainer
    trainer = Trainer(
            model=model,
            args=training_args,
            train_dataset=downsampled_dataset["train"],
            eval_dataset=downsampled_dataset["test"],
            data_collator=data_collator,
            tokenizer=tokenizer,
    )
```

```
[ ] trainer.train()
```

训练结果：

[31250/31250 1:13:54, Epoch 10/10]

Epoch	Training Loss	Validation Loss
1	5.450100	3.942292
2	3.614300	3.164917
3	3.148800	2.919345
4	2.942100	2.759429
5	2.809400	2.646808
6	2.721800	2.582737
7	2.660300	2.527395
8	2.617500	2.500902
9	2.586900	2.484974
10	2.573500	2.462837

在训练完成后，保存之前训练好的分词器和模型到本地，在之后的文本分类微调任务中将使用它们。

（四） 模型文本分类任务微调

在前面的 MLM 训练任务中，重新训练的分词器 Tokenizer 针对本次实验的训练数据进行了学习，建立了新的词表；重新训练的 DistilBERT 模型在针对本次实验的训练数据进行 MLM 任务训练后，学习到了这些脱敏数据潜在的上下文关系。可以将脱敏数据看成“另一种语言”，而之前的训练过程使得模型学习到了这“另一种语言”的相关知识，可以针对文本分类任务进行微调。具体模型结构图也存放于 img 文件夹下。

1） 使用训练好的分词器进行分词：

```
[ ]  #  加载之前训练的分词器而非默认分词器
     tokenizer  =  AutoTokenizer.from_pretrained('./tokenizer')
```

```
[ ]  def  preprocess_function(examples):
            return  tokenizer(examples["text"],  truncation=True)
```

```
[ ]  tokenized_datasets  =  train_data.map(preprocess_function,  batched=True,  remove_columns=["text"])
```

2） 加载训练所需组件，进行微调：

加载对应任务的数据整理器：

```
[ ]  #  使用文本分类任务相关的数据整理器
     from  transformers  import  DataCollatorWithPadding

     data_collator  =  DataCollatorWithPadding(tokenizer=tokenizer)
```

加载准确度评估指标，编写在微调过程中输出评估指标结果的函数：

```
[ ]  #  加载评估指标——准确度
     import  evaluate

     accuracy  =  evaluate.load("accuracy")
```

```
[ ]  import  numpy  as  np

     #  定义在训练过程中输出的准确度指标
     def  compute_metrics(eval_pred):
             predictions,  labels  =  eval_pred
             predictions  =  np.argmax(predictions,  axis=1)
             return  accuracy.compute(predictions=predictions,  references=labels)
```

加载之前保存的在 MLM 任务上训练好的模型，建立标签映射：

```
#  建立数字标签-文本标签的一一映射
id2label  =  {0:  '科技',  1:  '股票',  2:  '体育',  3:  '娱乐',  4:  '时政',  5:  '社会',
                      6:  '教育',  7:  '财经',  8:  '家居',  9:  '游戏',  10:  '房产',
                      11:  '时尚',  12:  '彩票',  13:  '星座'}
label2id  =  {'科技':  0,  '股票':  1,  '体育':  2,  '娱乐':  3,  '时政':  4,  '社会':  5,
                      '教育':  6,  '财经':  7,  '家居':  8,  '游戏':  9,  '房产':  10,
                      '时尚':  11,  '彩票':  12,  '星座':  13}
```

```
[ ]  from  transformers  import  AutoModelForSequenceClassification,  TrainingArguments,  Trainer
     #  加载之前在MLM任务上训练好保存在本地的DistilBERT模型
     model  =  AutoModelForSequenceClassification.from_pretrained(
             "./DistilBERT-MLM",  num_labels=14,  id2label=id2label,  label2id=label2id
     )
```

Hugging Face API 使用 AutoModelForMaskedML 函数加载模型检查点以进行 MLM 训练任务，使用 AutoModelSequenceClassification 函数加载模型检查点以进行文本分类任务，实际上 AutoModelSequenceClassification 函数就是在原本的模型结构的基础上加上了一个用于分类任务的模型头，需要传入标签到索引的映射关系确定需要进行分类的种类，模型之前的层的权重被冻结用于微调。

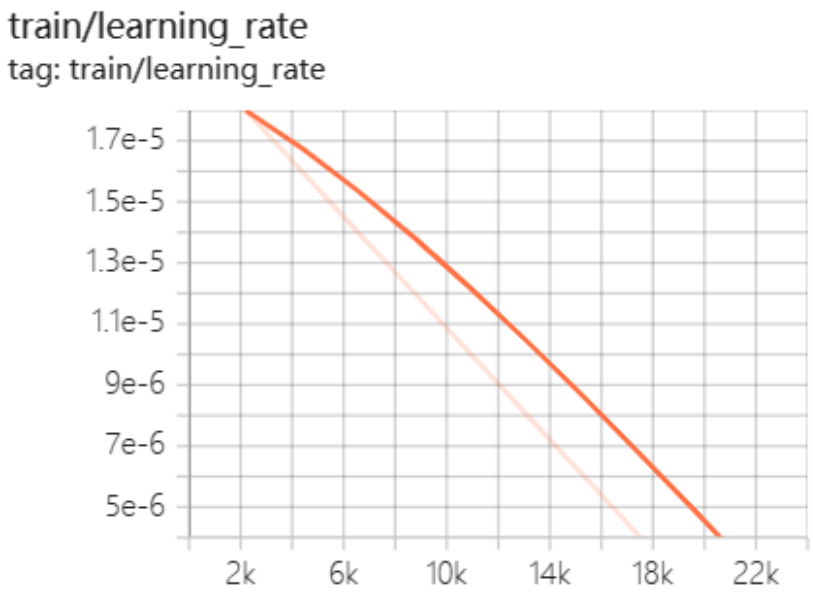
定义 Trainer 使用的超参数，传入数据整理器、分词器、指标计算函数和分词后的数据集，调用 train 方法开启微调过程：

训练超参数：

```
[ ]  #  定义Trainer使用的参数
     training_args  =  TrainingArguments(
             output_dir="distilbert-base-mlm-finetuned-classifier",
             learning_rate=2e-5,
             per_device_train_batch_size=64,
             per_device_eval_batch_size=64,
             num_train_epochs=10,
             overwrite_output_dir=True,
             weight_decay=0.01,
             evaluation_strategy="epoch",
             save_total_limit=2,
             save_strategy="epoch",
             push_to_hub=False,
             fp16=True,
             logging_strategy="epoch",
             logging_dir="/root/tf-logs",
     )
```

```
     trainer  =  Trainer(
             model=model,
             args=training_args,
             train_dataset=tokenized_datasets["train"],
             eval_dataset=tokenized_datasets["validation"],
             tokenizer=tokenizer,
             data_collator=data_collator,
             compute_metrics=compute_metrics,
     )
```

Hugging Face Trainer 默认使用线性衰减的学习率调度：

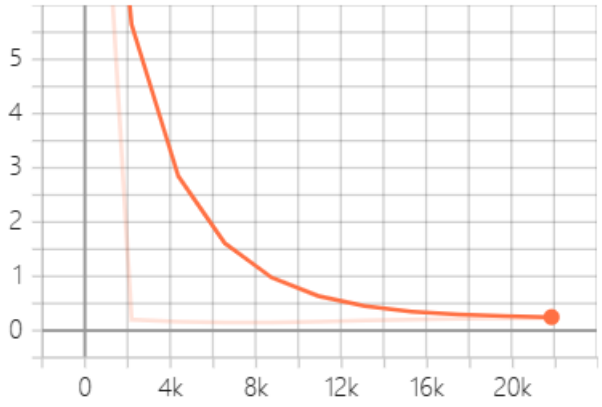


训练结果：

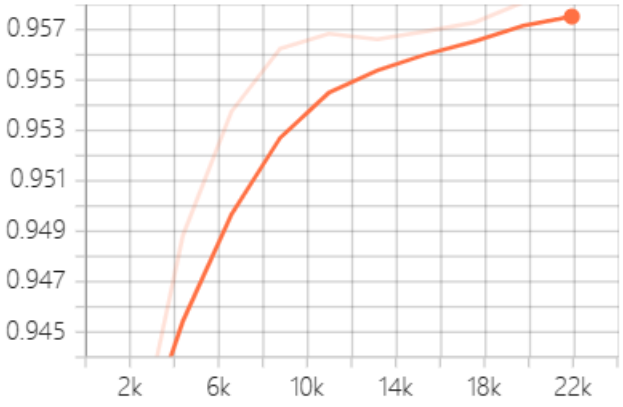
[21880/21880 2:51:57, Epoch 10/10]

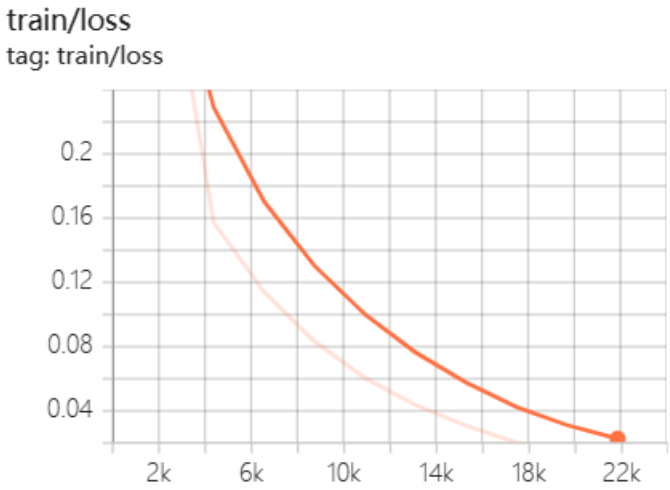
Epoch	Training Loss	Validation Loss	Accuracy
1	0.348600	0.200055	0.939750
2	0.157200	0.167829	0.948833
3	0.114000	0.153521	0.953750
4	0.083200	0.151610	0.956250
5	0.060400	0.166850	0.956833
6	0.043600	0.187417	0.956617
7	0.030800	0.200048	0.956933
8	0.020300	0.212938	0.957283
9	0.014500	0.220913	0.958083
10	0.010100	0.225648	0.958050

eval/loss
tag: eval/loss



eval/accuracy
tag: eval/accuracy





在微调过程完成后保存模型用于测试集数据推断。本次实验的模型在 RTX A5000 24G 的设备上，开启 FP16 选项的条件下，训练总时长（MLM 训练+文本分类微调）大约为 6 个小时。

（五） 推断测试集，保存结果

加载测试数据集，逐条文本进行推断，保存结果：

```
from transformers import AutoTokenizer
# 加载之前训练的分词器
tokenizer = AutoTokenizer.from_pretrained("./tokenizer")
```

```
from transformers import AutoModelForSequenceClassification
# 加载之前保存到本地的文本分类模型
model = AutoModelForSequenceClassification.from_pretrained("./DistilBERT-Classifier").to("cuda")
```

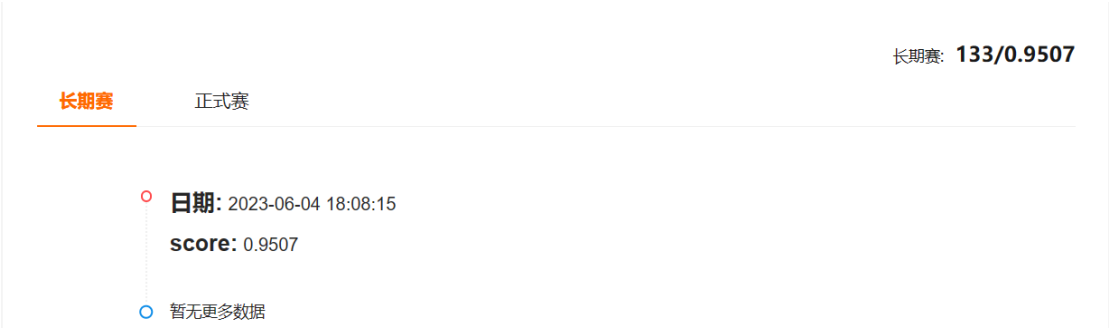
```
import pandas as pd
df = pd.DataFrame(data=None, columns=["label"])
```

```
import torch
# 进行推断，运行时间较长，在10分钟左右，且会占用大量的显存（大约21G左右），请确保你至少有24G的空闲显存可使用
count = 0
for sentence in test_data["train"]["text"]:
        inputs = tokenizer(sentence, truncation=True, return_tensors="pt")
        with torch.no_grad():
                logits = model(**inputs.to("cuda")).logits
        predicted_class_id = logits.argmax().to("cpu").item()
        count = count + 1
        df.loc[count] = [predicted_class_id,]
```

```
df.to_csv('distilbert-classifier.csv',index=False)
```

四、 评测结果

最终得分 0.9507 分，长期赛排名 133 名



五、 总结

由于本次实验使用的数据集中的数据都是经过脱敏处理的数据，因此如何对数据集进行恰当的预处理是重中之重，对于使用预训练模型进行微调任务来说，经过脱敏的数据集是不能够直接使用的，因为预训练模型在训练时使用的语料与它大不相同，预训练模型不知道任何关于这个数据集的知识，同样的，预训练模型使用的分词器也不知道任何对于脱敏数据的分词策略和映射关系。因此，预训练模型和分词器都需要针对这个脱敏数据集进行训练或微调。另一种可行的方案是，将脱敏后的数据作为索引，使用分词器将它们映射为分词器词表中对应的词元，也就是说，将数字转换为文本，再利用预训练模型良好的泛化性，针对文本分类任务进行微调。实验表明，使用相同模型的情况下，先转文本再微调的验证集准确率在 94%左右，先进行 MLM 任务训练再微调的验证集准确率接近 96%。本次实验的方案仍有改进的余地，主要集中在 MLM 任务的训练上，通过使用完整的膨胀后的数据集、训练更多的轮次、调整学习率调度策略等方式可能会等到更好的结果。另一种可能的改进方式是着手于对数据集的预处理操作，由于本数据集每条文本的平均长度过长，对数据使用截断操作作为模型的输入可能会丢失大量的信息，因此对原始训练数据进行词嵌入训练、使用 TF-IDF 编码等操作对输入数据进行“降维”，可能能够得到更优质的训练数据，从而得到更好的结果。同样，使用多模型进行集成提升预测效果也是一种可选的方案。