实验 5：BERT 实践实验

大数据分析实践实验报告

姓　　名: 郑坤武

学　　号: 202200150184

班　　级: 22 级公信班

学　　院: 政治学与公共管理学院

2025 年 11 月 18 日

1 实验目的

本实验基于开源预训练语言模型 **BERT** 与 HuggingFace 生态（**transformers** 与 **datasets**），在 **GLUE–MRPC** 句子对复述识别任务上进行微调，构建一个完整的“预训练模型 → 下游任务”教学范例。实验重点包括：

- 掌握 **BertTokenizerFast** 与 **BertForSequenceClassification** 的基本用法（模型加载、分词编码、前向计算与损失获取）。
- 通过 **datasets** 库加载 GLUE 中的 **MRPC** 数据集，完成句子对的批量编码与划分（训练集 / 验证集 / 测试集）。
- 实现完整的 BERT 微调流程：**动态 padding 的 DataLoader**、训练循环、验证与测试评估（Accuracy 与 F1），理解预训练模型在下游任务中的迁移能力。
- 将微调后的模型与分词器保存到本地文件夹，形成可复用的 **bert_mrpc_finetuned** 模型资源，便于后续加载与推理。

2 实验环境

- **操作系统：** macOS 12.x
- **开发工具：** VS Code + 终端 (Terminal)
- **语言与运行环境：** Python 3.12，虚拟环境 `venv`
- **主要依赖库：**
  - PyTorch（`torch`）
  - Transformers（`transformers`）
  - Datasets（`datasets`）
  - Scikit-learn（`scikit-learn`）
  - tqdm（`tqdm`）

3 具体实验步骤与结果分析

3.1 项目初始化与环境配置

在本地 `macOS` 上新建实验目录并创建 Python 虚拟环境，使用 pip 安装所需依赖。终端中主要命令如下：

```
// 创建实验目录并进入
mkdir -p ~/bigdata_exp5_bert
cd ~/bigdata_exp5_bert

// 创建并激活虚拟环境
python3 -m venv venv
```

```
source venv/bin/activate

// 升级 pip 并安装依赖
pip install --upgrade pip
pip install torch transformers datasets scikit-learn tqdm
```

完成后，在 VS Code 中打开该目录，后续所有 Python 脚本（如 `exp5_mrpc_bert.py`）均放在此文件夹中，方便统一管理。

3.2 数据集与任务介绍

本实验采用 GLUE（General Language Understanding Evaluation）基准中的 **MRPC (Microsoft Research Paraphrase Corpus)** 数据集，其核心是“句子对是否为语义复述”的二分类任务。MRPC 中每条样本由三个字段构成：

- `sentence1`：第一个英文句子。
- `sentence2`：第二个英文句子。
- `label`：标签，`1` 表示两句为语义等价（复述），`0` 表示不等价。

实验目标是：给定一对句子 (s_1, s_2)，判断它们是否表达了相同的含义，即构建一个基于 BERT 的句子对分类模型，并在 MRPC 的验证集与测试集上评估其准确率与 F1 指标。

3.3 模型与分词器加载

在脚本中首先设置随机种子与设备选择逻辑，保证实验可复现并优先利用硬件加速（CUDA / MPS）：

```
// 随机种子与设备选择
def set_seed(seed:  int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

def get_device() -> torch.device:
    if torch.cuda.is_available():
        print("使用设备: CUDA")
        return torch.device("cuda")
    if  hasattr(torch.backends, "mps") and torch.backends.mps.is_available():
        print("使用设备: MPS (Apple 芯片)")
        return torch.device("mps")
    print("使用设备: CPU")
    return torch.device("cpu")
```

随后加载预训练好的 **BERT Base** 模型与对应分词器，并移动到指定设备：

```
print("正在加载 GLUE MRPC 数据集...")
raw_datasets = load_dataset("glue", "mrpc")
```

```
3
4  print("正在加载 bert-base-uncased 模型...")
5  tokenizer = BertTokenizerFast.from_pretrained("bert-base-uncased")
6  model = BertForSequenceClassification.from_pretrained(
7      "bert-base-uncased", num_labels=2
8  )
9  model.to(device)
```

其中 num_labels=2 对应 MRPC 的二分类设置。

3.4 数据预处理与编码

由于 MRPC 是句子对任务，需要对两句输入联合编码。使用 BertTokenizerFast 提供的批处理接口对数据进行分词与截断：

```
1  def preprocess_function(examples, tokenizer, max_length:  int = 128):
2      return tokenizer(
3          examples["sentence1"],
4          examples["sentence2"],
5          truncation=True,
6          max_length=max_length,
7      )
8
9  tokenized_datasets = raw_datasets. map(
10     lambda examples: preprocess_function(examples, tokenizer),
11     batched=True,
12     desc="Tokenization",
13 )
14
15 // 删去无关列，并重命名标签
16 tokenized_datasets = tokenized_datasets.remove_columns(
17     ["idx", "sentence1", "sentence2"]
18 )
19 tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
```

这样模型在前向计算时即可直接接收 input_ids、attention_mask、token_type_ids 与 labels 等字段。

3.5 DataLoader 设计与动态 padding

为了兼顾效率与易用性，本实验采用 **动态 padding** 方式：不同 batch 的最大长度可以不同，同一 batch 内对齐到该批次的最大序列长度。

```
1  @dataclass
2  class Batch:
3      input_ids: torch.Tensor
4      attention_mask: torch.Tensor
5      token_type_ids: torch.Tensor
6      labels: torch.Tensor
7
```

```python
def collate_fn(features: List[Dict], pad_token_id:  int) -> Batch:
    input_ids = [f["input_ids"] for f in features]
    attention_mask = [f["attention_mask"] for f in features]
    token_type_ids = [f["token_type_ids"] for f in features]
    labels = torch.tensor([f["labels"] for f in features], dtype=torch. long)

    max_len =  max( len(x) for x in input_ids)

    def pad(seqs, pad_value):
        return torch.stack(
            [
                torch.tensor(
                     list(seq) + [pad_value] * (max_len -  len(seq)), dtype=torch. long
                )
                for seq in seqs
            ]
        )

    input_ids = pad(input_ids, pad_token_id)
    attention_mask = pad(attention_mask, 0)
    token_type_ids = pad(token_type_ids, 0)

    return Batch(
        input_ids=input_ids,
        attention_mask=attention_mask,
        token_type_ids=token_type_ids,
        labels=labels,
    )

def create_dataloaders(tokenized_datasets, tokenizer, batch_size:  int = 16):
    pad_token_id = tokenizer.pad_token_id
    train_loader = DataLoader(
        tokenized_datasets["train"], batch_size=batch_size,
        shuffle=True,
        collate_fn=lambda batch: collate_fn(batch, pad_token_id),
    )
    eval_loader = DataLoader(
        tokenized_datasets["validation"], batch_size=batch_size,
        shuffle=False,
        collate_fn=lambda batch: collate_fn(batch, pad_token_id),
    )
    test_loader = DataLoader(
        tokenized_datasets["test"], batch_size=batch_size,
        shuffle=False,
        collate_fn=lambda batch: collate_fn(batch, pad_token_id),
    )
    return train_loader, eval_loader, test_loader
```

动态 padding 能有效减少无效的 [PAD] 计算，提升训练速度和内存利用率。

3.6 训练策略与循环实现

训练阶段使用 AdamW 优化器与线性 warmup–decay 学习率调度器，训练轮数为 3 轮。关键超参数如下：

- 学习率：2e-5
- Batch size：16
- 训练轮数：3
- 优化器：torch.optim.AdamW
- 学习率调度：get_linear_schedule_with_warmup
- 梯度裁剪：max_norm=1.0

训练函数与评估函数的核心结构如下：

```python
def train_one_epoch(model, dataloader, optimizer, scheduler, device, epoch_idx:  int):
    model.train()
    running_loss = 0.0
    progress_bar = tqdm(dataloader, desc=f"Epoch {epoch_idx+1} 训练中")

    for batch in progress_bar:
        batch = Batch(
            input_ids=batch.input_ids.to(device),
            attention_mask=batch.attention_mask.to(device),
            token_type_ids=batch.token_type_ids.to(device),
            labels=batch.labels.to(device),
        )
        outputs = model(
            input_ids=batch.input_ids,
            attention_mask=batch.attention_mask,
            token_type_ids=batch.token_type_ids,
            labels=batch.labels,
        )
        loss = outputs.loss
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        if scheduler is not None:
            scheduler.step()
        running_loss += loss.item()
        progress_bar.set_postfix({"loss": f"{loss.item():.4f}"})
    return running_loss /  len(dataloader)

def evaluate(model, dataloader, device, desc:  str = "验证集评估"):
    model. eval()
    all_labels, all_preds = [], []
    with torch.no_grad():
```

```
        progress_bar = tqdm(dataloader, desc=desc)
        for batch in progress_bar:
            batch = Batch(
                input_ids=batch.input_ids.to(device),
                attention_mask=batch.attention_mask.to(device),
                token_type_ids=batch.token_type_ids.to(device),
                labels=batch.labels.to(device),
            )
            outputs = model(
                input_ids=batch.input_ids,
                attention_mask=batch.attention_mask,
                token_type_ids=batch.token_type_ids,
            )
            logits = outputs.logits
            preds = torch.argmax(logits, dim=-1)
            all_labels.extend(batch.labels.cpu().numpy().tolist())
            all_preds.extend(preds.cpu().numpy().tolist())
    acc = accuracy_score(all_labels, all_preds)
    f1 = f1_score(all_labels, all_preds)
    return acc, f1
```

每轮训练结束后在验证集上计算 Accuracy 与 F1，记录最佳 F1 对应的模型参数，用于最终测试集评估。

3.7 测试集评估与结果分析

在完成 3 轮训练后，将最佳验证效果对应的模型参数加载回来，在 MRPC 测试集上进行最终评估，得到如下指标：

- 测试集准确率：**0.8435**
- 测试集 F1 分数：**0.8854**

从结果可以看出，在没有特别复杂的调参与模型改造的情况下，单纯对 `bert-base-uncased` 进行少量 epoch 的微调，就可以在 MRPC 任务上取得较高的 F1 分数，说明 BERT 预训练模型已经在语义表征上学习到了丰富的语言知识，对小数据集任务具有良好的迁移能力。

如图 2 所示，展示训练过程与最终测试结果。

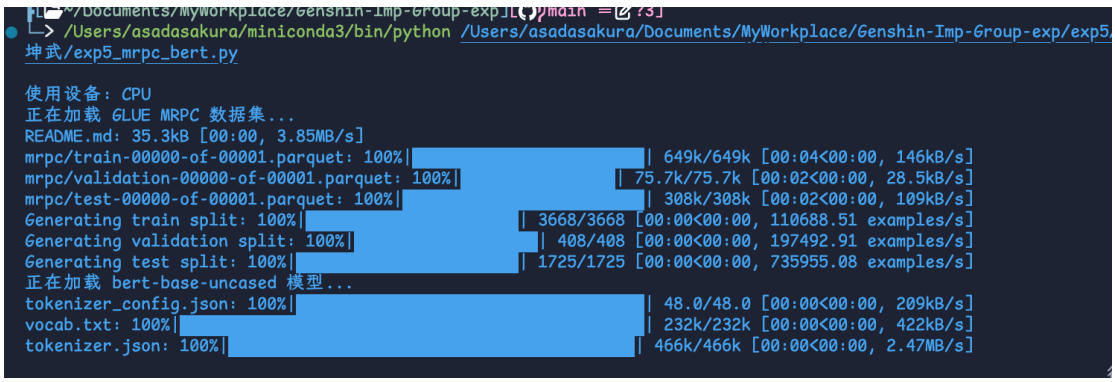

图 1: 终端中 BERT 微调过程示例

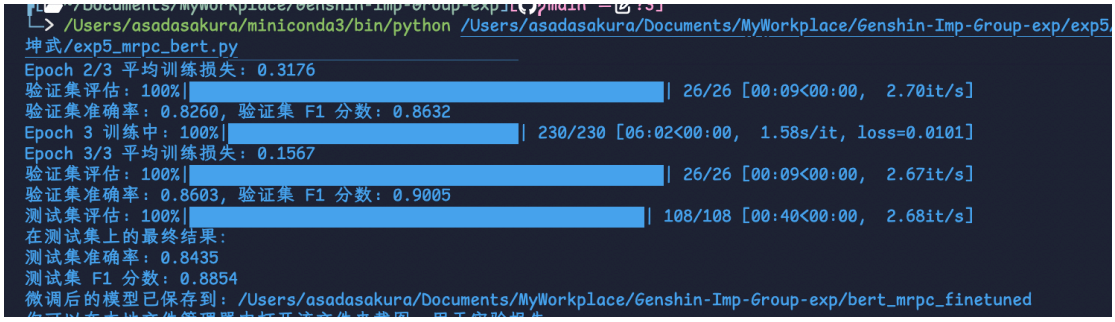

图 2: 终端中 BERT 微调测试集评估结果示例

3.8 模型保存与本地文件结构

实验最后将微调后的模型与分词器保存到本地目录 `bert_mrpc_finetuned` 中：

```
save_dir = "bert_mrpc_finetuned"
os.makedirs(save_dir, exist_ok=True)
model.save_pretrained(save_dir)
tokenizer.save_pretrained(save_dir)
print(f"微调后的模型已保存到: {os.path.abspath(save_dir)}")
```

在本地实际运行中，模型保存路径为：

`/Users/.../bert_mrpc_finetuned`

该目录中包含配置文件、权重文件与分词器相关文件等，可通过本地图形界面截图展示，如图 3 所示。

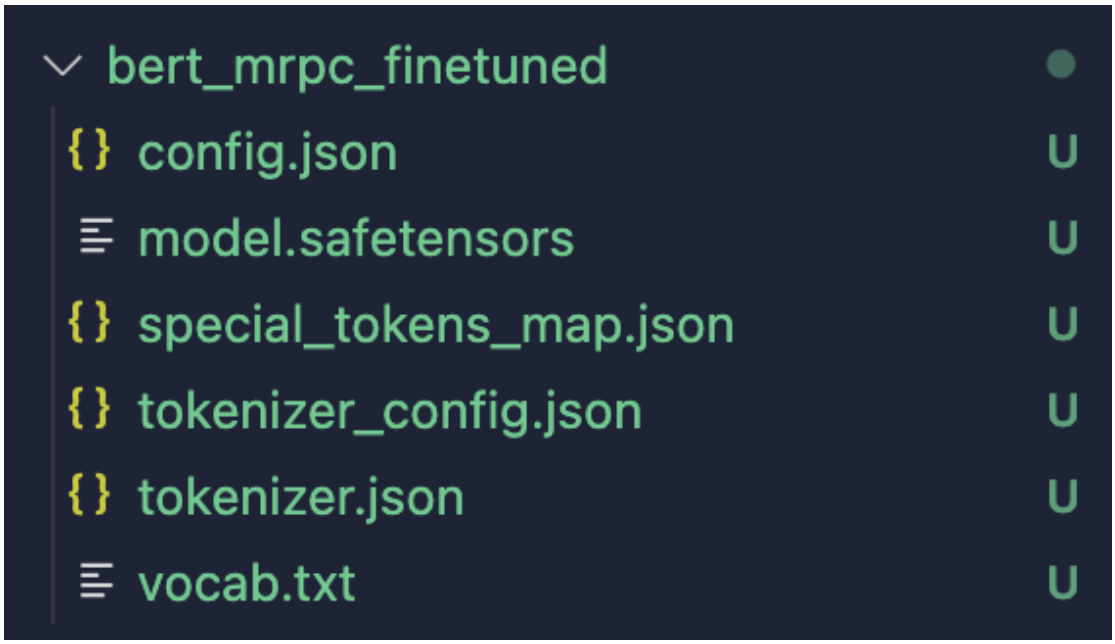

图 3: 本地 `bert_mrpc_finetuned` 目录结构示意

通过该步骤，验证了微调模型可以顺利落地到本地文件系统，为后续加载、推理或继续微调提供了基础。

4 实验总结与收获

本实验从零开始在本地 macOS 环境中，基于 HuggingFace 生态完成了 BERT 在 GLUE–MRPC 任务上的微调，实现了“预训练模型 + 下游任务”的典型范例。整个过程涵盖了数据加载与预处理、动态 padding 的 DataLoader 设计、训练与验证评估、测试集性能分析以及模型持久化保存等关键环节。

通过本次实验，我对以下几点有了更深刻的理解：

- **预训练模型的优势：**与从头训练模型相比，直接在 `bert-base-uncased` 上做少量微调，就能在小样本任务上取得较高的准确率与 F1，体现出大规模预训练在表示学习上的强大能力。
- **工程化流程意识：**通过自定义 `collate_fn`、设计训练 / 评估函数与保存模型等步骤，初步建立起一个可复用的 NLP 训练脚手架，后续可以方便迁移到其他文本分类、句子匹配等任务上。
- **调参与扩展空间：**当前实验仅使用了默认的 BERT Base 与简单的超参数配置，后续可以尝试更深层的模型（如 RoBERTa、DeBERTa 等）、不同学习率 / batch size、更多 epoch 或加入早停（early stopping）机制，以探索性能进一步提升的可能性。

总体而言，本实验帮助我从“会用预训练模型”迈向“能够自己搭建完整微调流程”的阶段，为后续学习更复杂的 NLP 任务（如问答、文本生成、多任务学习等）打下了实践基础。