数据挖掘与最优化: Assignment 4

组长:	231502004 李子	完成内容	T3	贡献度	20%
组员:	221300066 季千焜	完成内容	T1	贡献度	20%
组员:	221098024 李瑛琦	完成内容	T4	贡献度	20%
组员:	221098145 李傲雪	完成内容	T2	贡献度	20%
组员:	221098071 单佳仪	完成内容	T2	贡献度	20%

目录

实验进度

我们完成了所有内容。

T1

以下是实现缩放点积注意力机制的自定义函数，使用 PyTorch 框架：

```python
import torch
import torch.nn.functional as F

def scaled_dot_product_attention(Q, K, V):
    """
    缩放点积注意力机制实现

    参数:
        Q: 查询张量 (Query), 形状为 [batch_size, num_heads, seq_len_q, depth_k]
        K: 键张量 (Key), 形状为 [batch_size, num_heads, seq_len_k, depth_k]
        V: 值张量 (Value), 形状为 [batch_size, num_heads, seq_len_v, depth_v]

    返回:
        注意力加权后的输出张量, 形状为 [batch_size, num_heads, seq_len_q, depth_v]
    """
    # 1. 计算Q和K的点积
    matmul_qk = torch.matmul(Q, K.transpose(-2, -1))  # [batch_size, num_heads, seq_len_q, seq_len_k]

    # 2. 缩放操作: 除以sqrt(d_k)
    d_k = K.size(-1)
    scaled_attention_logits = matmul_qk / torch.sqrt(torch.tensor(d_k, dtype=torch.float32))

    # 3. 应用softmax获取权重
    attention_weights = F.softmax(scaled_attention_logits, dim=-1)  # [batch_size, num_heads, seq_len_q,
     seq_len_k]

    # 4. 用权重加权V
    output = torch.matmul(attention_weights, V)  # [batch_size, num_heads, seq_len_q, depth_v]

    return output
```

使用示例：

```python
# 创建测试数据
batch_size = 2
num_heads = 4
```

```python
seq_len_q = 5
seq_len_kv = 7
depth_k = 64
depth_v = 80

Q = torch.randn(batch_size, num_heads, seq_len_q, depth_k)
K = torch.randn(batch_size, num_heads, seq_len_kv, depth_k)
V = torch.randn(batch_size, num_heads, seq_len_kv, depth_v)

# 计算注意力输出
output = scaled_dot_product_attention(Q, K, V)
print(output.shape)  # 输出: torch.Size([2, 4, 5, 80])
```

T2

Transformer 使用正弦和余弦函数来构造位置编码。其中，pos 表示位置，i 表示特征维度，$PE(p,\cdot)\in R^{d_{model}}$ 表示给第 p 个词嵌入向量添加的位置矫正向量。

$$PE(pos,2i)=sin(\frac{1}{10000^{2i/d_{model}}}pos) \tag{1}$$

$$PE(pos\ 2i+1)=cos(\frac{1}{10000^{2i/d_{model}}}pos) \tag{2}$$

实现 Transformer 中的正余弦位置编码的自定义函数编码如下：

```python
import numpy as np
import matplotlib.pyplot as plt

def positional_encoding(pos, d):
    """
    实现Transformer中的正余弦位置编码

    参数:
    pos - 位置(整数)
    d - 编码维度(整数)

    返回:
    pe - 位置编码向量(numpy数组)
    """
    pe = np.zeros(d)
    for i in range(0, d, 2):
        freq = pos / (10000 ** (i / d))# 计算正弦和余弦的频率
        pe[i] = np.sin(freq)# 偶数位置用正弦
        if i + 1 < d:
            pe[i+1] = np.cos(freq)# 奇数位置用余弦
    return pe
```

下面调用该函数，令 d=500，在同一张图上绘制位置为 1 和位置为 100 的编码图形（横坐标为编码维度 1，⋯，d，纵坐标为编码具体数值）。

```python
# 设置编码维度
d = 500

# 计算位置1和位置100的编码
pos1_encoding = positional_encoding(1, d)
pos100_encoding = positional_encoding(100, d)

# 绘制图形
plt.figure(figsize=(12, 6))
```

```
    plt.plot(range(1, d+1), pos1_encoding, label='Position 1', alpha=0.7)
    plt.plot(range(1, d+1), pos100_encoding, label='Position 100', alpha=0.7)
    plt.xlabel('Encoding Dimension')
    plt.ylabel('Encoding Value')
    plt.title('Positional Encoding for Position 1 and Position 100 (d=500)')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()
```

位置为 1 和位置为 100 的编码图形如下所示：

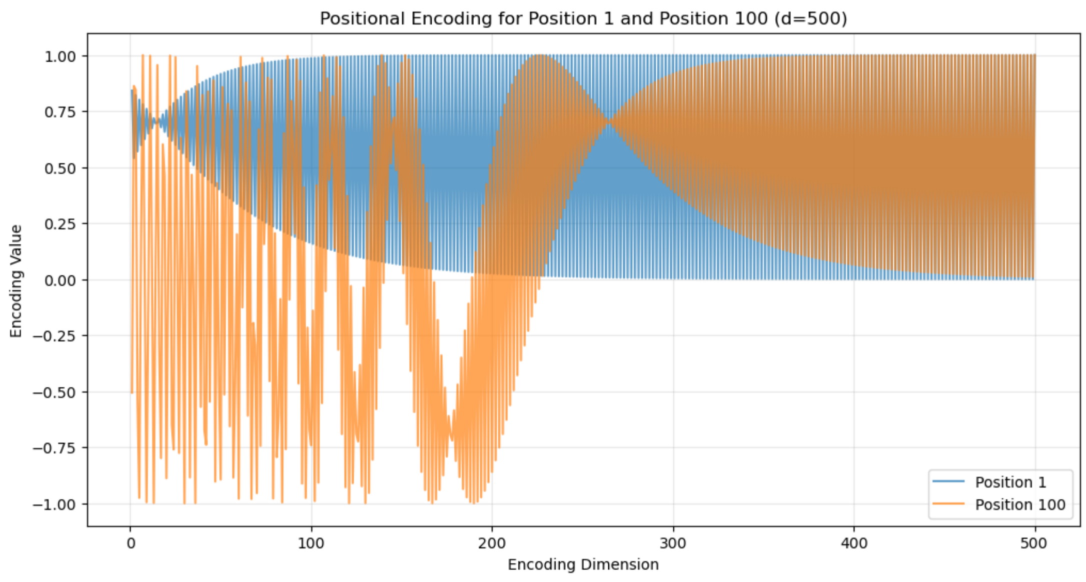


图 1: 位置编码图形

T3

之前的一个项目里用过 GPT2 微调，因此直接采用了之前的 GPT2 代码进行推理。参考的项目在：https://github.com/graykode/gpt-2-Pytorch

generator.py 代码：

```python
import torch
import random
import numpy as np
from GPT2.model import GPT2LMHeadModel
from GPT2.utils import load_weight
from GPT2.config import GPT2Config
from GPT2.sample import sample_sequence
from GPT2.encoder import get_encoder

def truncate_repetition_10(tokens, k=10):
    seen = set()
    for i in range(len(tokens) - k + 1):
        snippet = tuple(tokens[i:i + k])
        if snippet in seen:
            return tokens[:i]
        seen.add(snippet)
    return tokens

def text_generator(state_dict, args):
    if args.quiet is False:
        print(args)

    if args.batch_size == -1:
        args.batch_size = 1
    assert args.nsamples % args.batch_size == 0

    seed = random.randint(0, 2147483647)
    np.random.seed(seed)
    torch.random.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    enc = get_encoder()
    config = GPT2Config()
    model = GPT2LMHeadModel(config)
    model = load_weight(model, state_dict)
    model.to(device)
    model.eval()

    if args.length == -1:
        args.length = config.n_ctx // 2
    elif args.length > config.n_ctx:
        raise ValueError("Can't get samples longer than window size: %s" % config.n_ctx)
    if args.quiet is False:
        print(args.text)
    context_tokens = enc.encode(args.text)

    generated = 0
    for _ in range(args.nsamples // args.batch_size):
        out = sample_sequence(
            model=model, length=args.length,
            context=context_tokens if not args.unconditional else None,
            start_token=enc.encoder['<|endoftext|>'] if args.unconditional else None,
            batch_size=args.batch_size,
            temperature=args.temperature,
            top_k=args.top_k,
            device=device
        )

        out = out[:, len(context_tokens):].tolist()
        for i in range(args.batch_size):
            # print("iii:", i)
            generated += 1
            post_out = out[i]
            if enc.encoder['<|endoftext|>'] in post_out:
                post_out = post_out[:post_out.index(enc.encoder['<|endoftext|>'])]
            # 从前往后截断重复
            # print(enc.decode(post_out))
            post_out = truncate_repetition_10(post_out, 10)
            text = enc.decode(post_out)
            if args.quiet is False:
                print("=" * 40 + " SAMPLE " + str(generated) + " " + "=" * 40)
            print(text)
```

推理参数：temperature=0.7，top_k=20，end_by_endoftext=True

得到如下结果：

```
PS F:\Study\ml\gpt2\CAAonLLM> python main.py --text "Once upon a time"
F:\Study\ml\gpt2\CAAonLLM\main.py:37: FutureWarning: You are using `torch.load` with `
    weights_only=False` (the current default value), which uses the default pickle module
    implicitly. It is possible to construct malicious pickle data which will execute arbitrary
    code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
    models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This
    limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be
    loaded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`.
    We recommend you start setting `weights_only=True` for any use case where you don't have full control of the
    loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  state_dict = torch.load('gpt2-pytorch_model.bin', map_location='cpu' if not torch.cuda.
    is_available() else None)
Namespace(train=False, train_epochs=10, malicious_repeat_time=7, text='Once upon a time', quiet=
    False, nsamples=1, unconditional=False, batch_size=-1, length=-1, temperature=0.7, top_k=20,
    end_by_endoftext=True, autotest=False)
Once upon a time
100%|                                                                          |
                                                    512/512 [00:03<00:00, 160.53it/s]
======================================== SAMPLE 1 ========================================
 the great powers of history were at war with one another. The Great Powers of the world were
    divided into three classes. The First Order, the Dark Ones, and the White Mantle.

A large number of the great powers were the Dark Ones. The Dark Ones were the dark ones with the
    highest power. The White Mantle was the most powerful of the three. The Dark Ones were the
    most powerful of the three. The greatest power of all was the Dark One, and he was the Dark
    One.

This was the world of the great powers of the world.
```

文本开头表现：模型采用了经典的童话式开头，在风格模仿上具有一定准确性，展现出基础的叙事能力。

世界观构建尝试：文本引入了三个阵营（First Order, Dark Ones, White Mantle），体现出模型在搭建虚构世界结构方面的意图。

语言连贯性：整体语言语法正确，表达通顺，基本具备英文叙述的基本逻辑。

主要缺点：

- **信息重复：**多次重复描述 “Dark Ones” 的强大，缺乏语义推进；
- **逻辑空洞：**存在自指或语义封闭的句子（如：“he was the Dark One”）；
- **前后矛盾：**只关心文本续写，而不关心基本逻辑是否正确，经常会和前一句话矛盾。
- **词汇匮乏：**表达方式和句型单一，语言风格机械；
- **结尾空泛：**最后一语 “*This was the world of the great powers of the world.*” 几乎没有提供新的信息。

T4

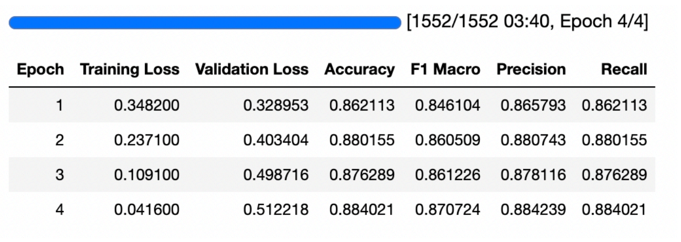
基于 financial phrasebank 微调英文预训练模型实现金融新闻或推文的情感分析:

```
from datasets import load_dataset
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer,
    pipeline
)
import numpy as np
from sklearn.metrics import classification_report, accuracy_score
import torch

# 设置随机种子保证可复现性
seed = 42
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(seed)

# 加载数据集
dataset = load_dataset("atrost/financial_phrasebank")

# 提取各划分的数据
train_data = dataset["train"]
val_data = dataset["validation"]
test_data = dataset["test"]

# 提取文本和标签
X_train = train_data["sentence"]
y_train = train_data["label"]
X_val = val_data["sentence"]
y_val = val_data["label"]
X_test = test_data["sentence"]
y_test = test_data["label"]

print(f"训练集: {len(X_train)}, 验证集: {len(X_val)}, 测试集: {len(X_test)}")
print(f"标签分布 - 训练集: {np.unique(y_train, return_counts=True)}")


# 加载FinBERT模型和tokenizer
```

```
model_name = "ProsusAI/finbert"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(
    model_name,
    num_labels=3,
    id2label={0: "negative", 1: "neutral", 2: "positive"},
    label2id={"negative": 0, "neutral": 1, "positive": 2}
)

# 对文本进行编码
def tokenize_function(examples):
    return tokenizer(
        examples,
        padding="max_length",
        truncation=True,
        max_length=128
    )

train_encodings = tokenize_function(X_train)
val_encodings = tokenize_function(X_val)
test_encodings = tokenize_function(X_test)

# 创建PyTorch数据集
class FinancialDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

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

train_dataset = FinancialDataset(train_encodings, y_train)
val_dataset = FinancialDataset(val_encodings, y_val)
test_dataset = FinancialDataset(test_encodings, y_test)

# 定义训练参数
training_args = TrainingArguments(
    output_dir='./results2',           # 输出目录
    eval_strategy="epoch",            # 每轮结束后评估
```

```
        save_strategy="epoch",              # 每轮结束后保存
        learning_rate=2e-5,                 # 学习率
        per_device_train_batch_size=8,      # 训练批次大小
        per_device_eval_batch_size=64,      # 评估批次大小
        num_train_epochs=4,                 # 训练轮数
        weight_decay=0.01,                  # 权重衰减
        load_best_model_at_end=True,        # 训练结束时加载最佳模型
        metric_for_best_model="accuracy",   # 最佳模型指标
        logging_dir='./logs2',               # 日志目录
        logging_steps=50,                   # 每50步记录一次日志
        report_to="none",                   # 禁用外部报告
        seed=seed,                          # 随机种子
        warmup_ratio=0.1,                   # 预热比例
    )

    # 定义评估指标
    def compute_metrics(p):
        predictions, labels = p
        predictions = np.argmax(predictions, axis=1)

        acc = accuracy_score(labels, predictions)
        report = classification_report(
            labels, predictions,
            target_names=['negative', 'neutral', 'positive'],
            output_dict=True
        )

        return {
            'accuracy': acc,
            'f1_macro': report['macro avg']['f1-score'],
            'precision': report['weighted avg']['precision'],
            'recall': report['weighted avg']['recall'],
        }

    # 创建Trainer
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        compute_metrics=compute_metrics,
    )

    # 训练模型
    print("\n开始微调FinBERT模型...")
    trainer.train()

    # 保存微调后的模型
    trainer.save_model("./finbert_finetuned")
    print("微调完成! 模型已保存到 finbert_finetuned 目录")

    # 在测试集上评估
    print("\n在测试集上评估微调后的模型...")
    test_results = trainer.predict(test_dataset)
    test_metrics = test_results.metrics

    print(f"\n测试集性能:")
    print(f"准确率: {test_metrics['test_accuracy']:.4f}")
    print(f"F1宏平均: {test_metrics['test_f1_macro']:.4f}")
    print(f"精确率: {test_metrics['test_precision']:.4f}")
    print(f"召回率: {test_metrics['test_recall']:.4f}")

    # 详细分类报告
    predictions = np.argmax(test_results.predictions, axis=1)
    print("\n详细分类报告:")
    print(classification_report(
        y_test, predictions,
        target_names=['negative', 'neutral', 'positive']
    ))
```

输出结果如下:
训练集: 3100, 验证集: 776, 测试集: 970
标签分布 - 训练集: (array([0, 1, 2]), array([382, 1852, 866]))
开始微调 FinBERT 模型...



[1552/1552 03:40, Epoch 4/4]

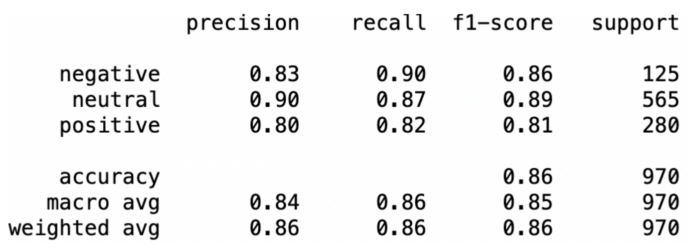
Epoch	Training Loss	Validation Loss	Accuracy	F1 Macro	Precision	Recall
1	0.348200	0.328953	0.862113	0.846104	0.865793	0.862113
2	0.237100	0.403404	0.880155	0.860509	0.880743	0.880155
3	0.109100	0.498716	0.876289	0.861226	0.878116	0.876289
4	0.041600	0.512218	0.884021	0.870724	0.884239	0.884021

图 2: 微调模型

在测试集上评估微调后的模型...
测试集性能:
准确率: 0.8608
F1 宏平均: 0.8531

精确率: 0.8625

召回率: 0.8608



```
              precision    recall  f1-score   support

    negative       0.83      0.90      0.86       125
     neutral       0.90      0.87      0.89       565
    positive       0.80      0.82      0.81       280

    accuracy                           0.86       970
   macro avg       0.84      0.86      0.85       970
weighted avg       0.86      0.86      0.86       970
```

图 3: 详细分类报告

基于同一训练数据，使用 Doc2Vec 得到的文本向量预测新闻情绪，比较预训练模型与 Doc2Vec 模型的样本外预测准确率:

```python
from gensim.models.doc2vec import Doc2Vec, TaggedDocument  # 添加这行导入
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score
# 2 Doc2Vec模型实现
# 2.1 数据准备
# df = pd.DataFrame({
#     'text': dataset['train']['sentence'],
#     'label': dataset['train']['label']
# })

# 2.2 划分训练集和测试集
# train_df, test_df = train_test_split(df, test_size=0.2, random_state=42)
train_df = pd.DataFrame({
    'text': X_train,  # 使用FinBERT的训练集文本
    'label': y_train   # 使用FinBERT的训练集标签
})
test_df = pd.DataFrame({
    'text': X_test,    # 使用FinBERT的测试集文本
    'label': y_test     # 使用FinBERT的测试集标签
})


# 2.3 准备Doc2Vec的输入格式
train_docs = [TaggedDocument(words=doc.split(), tags=[str(i)])
    for i, doc in enumerate(train_df['text'])]
# 2.4 训练Doc2Vec模型
model = Doc2Vec(train_docs, vector_size=100, window=5, min_count=2, workers=4, epochs=20)
```

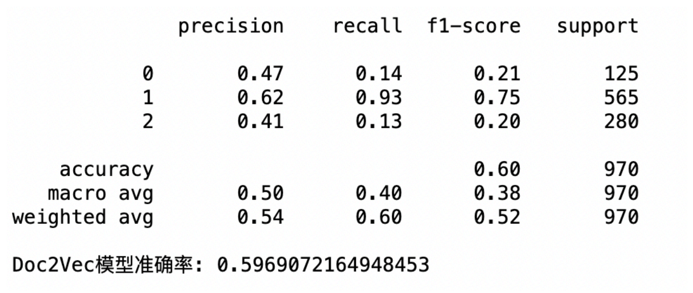
```python

# 2.5 提取文档向量
train_vectors = [model.infer_vector(doc.split()) for doc in train_df['text']]
test_vectors = [model.infer_vector(doc.split()) for doc in test_df['text']]

# 2.6 训练分类器
clf = LogisticRegression(max_iter=1000)
clf.fit(train_vectors, train_df['label'])
```

Out:

LogisticRegression(max_iter=1000) In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.

On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```python
# 2.7 预测并评估
predictions = clf.predict(test_vectors)
doc2vec_accuracy = accuracy_score(test_df['label'], predictions)
print(classification_report(test_df['label'], predictions))
print(f"Doc2Vec模型准确率:", doc2vec_accuracy)
```

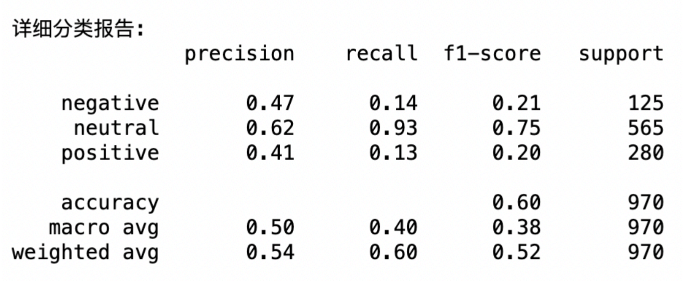


```
              precision    recall  f1-score   support

           0       0.47      0.14      0.21       125
           1       0.62      0.93      0.75       565
           2       0.41      0.13      0.20       280

    accuracy                           0.60       970
   macro avg       0.50      0.40      0.38       970
weighted avg       0.54      0.60      0.52       970

Doc2Vec模型准确率: 0.5969072164948453
```

图 4: Doc2vec 模型预测准确度

```python
# 2.10 计算详细的分类报告
class_report = classification_report(
    test_df["label"],
    predictions,
    target_names=["negative", " neutral", "positive"]
    # output_dict=True
)
print("\n详细分类报告:")
print(class_report)
```

```python
# 3.1 比较两个模型的性能
def compare_models():
    FinBERT_ac = test_metrics['test_accuracy']
    doc2vec_ac = doc2vec_accuracy
```



```
详细分类报告:
              precision    recall  f1-score   support

    negative       0.47      0.14      0.21       125
     neutral       0.62      0.93      0.75       565
    positive       0.41      0.13      0.20       280

    accuracy                           0.60       970
   macro avg       0.50      0.40      0.38       970
weighted avg       0.54      0.60      0.52       970
```

图 5: 详细分类报告

```python

        print("\n===== 模型性能对比 =====")
        print(f"FinBERT模型准确率: {FinBERT_ac:.4f}")
        print(f"Doc2Vec模型准确率: {doc2vec_ac:.4f}")
        print(f"性能差异: {FinBERT_ac - doc2vec_ac:.4f}")

        if FinBERT_ac > doc2vec_ac:
            print("FinBERT模型性能优于Doc2Vec模型")
        else:
            print("Doc2Vec模型性能优于FinBERT模型")

    # 3.2 执行比较
    if __name__ == "__main__":
        compare_models()
```

===== 模型性能对比 =====

FinBERT 模型准确率: 0.8608

Doc2Vec 模型准确率: 0.5969

性能差异: 0.2639

FinBERT 模型性能优于 Doc2Vec 模型