

大作业报告 --- 电影评论情感分析

许天祺 袁晨圃 李知谦 齐胡鑫

 Repo cauphenuny/ucas-ml

一、 运行效果

1.1 WebUI

实时显示对于评论的预测

启动方法参见 README.md



二、 特征工程

2.1 训练集数据分析

2.1.1 数据集格式

数据集：Kaggle Sentiment Analysis on Movie Reviews

- `train.tsv` 含 PhraseId/SentenceId/Phrase/Sentiment
 - Sentiment: 影评的情感倾向 (0~4, 负面~正面)
 - 源自 Stanford Sentiment Treebank (SST) 的研究
- `test.tsv` 类似, 但不含 Sentiment
 - 每个句子的所有解析子集都对应一个标签
 - 细粒度情感分析

PhraseId	SentenceId	Phrase	Sentiment
83635	4323	An exhilarating serving of movie fluff .	2
83636	4323	An exhilarating serving of movie fluff	3
83637	4323	An exhilarating	3
83638	4323	serving of movie fluff	2
83639	4323	of movie fluff	1
83640	4323	movie fluff	2

表 1 训练集数据片段

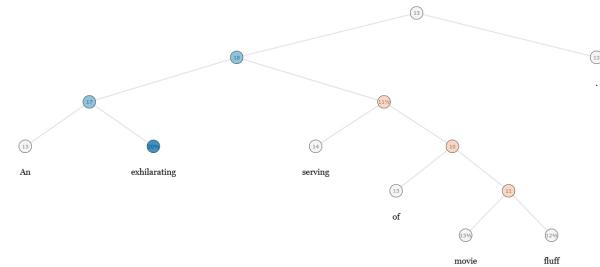


图 2 Stanford Sentiment Treebank

2.1.2 缺失与异常值处理

经检查，不存在缺失/异常值

2.1.3 标签分布

情感标签	含义	样本数	占比(约)
0	Negative	7,072	4.5%
1	Somewhat Negative	27,273	17.4%
2	Neutral	79,582	50.8%
3	Somewhat Positive	32,927	21.0%
4	Positive	9,206	5.9%

表 2 训练集情感分布统计

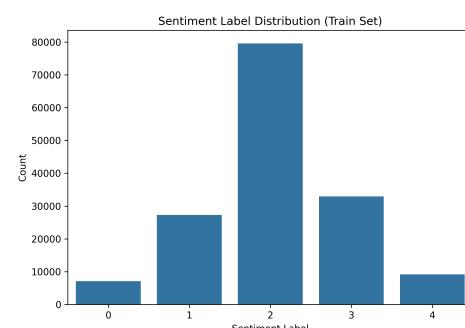


图 3 分布可视化

这是一个以 *Neutral* 为主导、极端情感样本稀缺的严重不均衡多分类数据集。

2.1.3.1 数据明显「类别不平衡」

- Neutral (2) \approx 占一半
- Negative / Positive < 6%
- 多数类样本 \approx 少数类样本的 10 倍以上

模型在训练时会天然偏向预测 Neutral。

2.1.3.2 符合真实世界分布

大多数评论是中性或轻微情感，极端情绪本来就少，数据分布符合真实用户行为，而非人工平衡数据。

2.1.3.3 在这个数据集上，Accuracy 指标意义有限

如果一个模型什么都不学，永远预测 $\text{Sentiment} = 2$ 它的 accuracy 就是： $\approx 50.8\%$

这意味着：55% 以下 → 模型几乎没学，60% → 只是比瞎猜稍好

2.1.3.4 极端情感 (0 / 4) 是最难学的

Negative + Positive =10.4%

但它们在实际应用中最重要

模型很容易把: $0 \rightarrow 1/2$, $4 \rightarrow 3/2$ 的情感“强度”被削弱

2.2 文本分词与向量化

2.2.1 TF-IDF

首先提取单词，然后去掉文本中的停用词(Stop Words)，然后对剩余词语进行ngram计数，取出频率最高的一些unigram和bigram作为词。

TF-IDF：是一种衡量某个词对特定文档重要性的统计方法

1. 计算词频，衡量某个词在文档中的重要性

$$\text{TF}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

- ## 2. 计算逆文档频率，惩罚常见词，提升稀有词权重

$$\text{IDF}(t) = \log\left(\frac{N}{1 + \text{DF}(t)}\right)$$

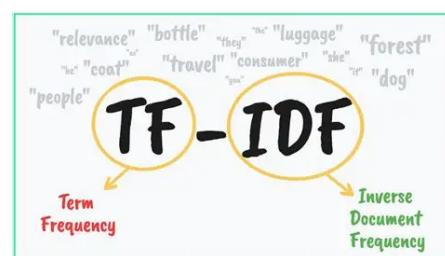


图 4 TF-IDF

- ### 3. 计算 TF-IDF 向量

$$\text{TF-IDF}(t, d) \equiv \text{TF}(t, d) \times \text{IDF}(t)$$

问题？不含有位置信息、词素大小与未登录词之间存在两难问题

2.2.2 BPE Tokenization

BPE (Byte Pair Encoding) 通过迭代合并高频字符对构建子词词表，有效平衡词典大小与 Out of Vocabulary 问题。

BPE 只是一种高效的编码，不含有语义信息，到语义空间的映射由 Embedding 层学习获得。

例 初始语料：“low lower lowest”

- 基础拆分: low|lower|lowest
 - 合并高频对 "lo": lo w|low e r|low e s t
 - 合并高频对 "low": low|low e r|low e s t
 - 最终词表包含: low, e, r, st, er, est, lowest 等子词, 每一个子词对应一个整数 id

训练 BPE tokenizer 的流程：

阶段	核心操作
1. 预分词	按特殊标记分割文本，再用正则规则提取基础词元，BPE 的合并不越过基础词元边界
2. 频率统计	统计所有相邻词元对的共现频率
3. 迭代合并	重复合并当前最高频的词对，将其加入词表
4. 词表构建	记录所有子词及其唯一 ID，生成 <code>vocab.json</code> 和 <code>merges.json</code>

三、模型设计与实现

3.1 传统 ML 方法

3.1.1 Logistic Regression

将 TF-IDF 处理后的文本通过逻辑回归 (Logistic Regression, LR) 进行分类。

用于多分类的 LR:

对于每一个类别 k ，计算一个得分

$$z_k = w_k^\top x + b_k$$

类别 k 的预测概率为：

$$P(y = k|x) = \frac{\exp(z_k)}{\sum_{j=1}^K \exp(z_j)} \quad \text{i.e. Softmax}(z_k)$$

(这也是为什么多分类 LR 又叫 Softmax Regression)

损失函数 (Cross Entropy Loss):

$$-\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(P(y = k|x^{(i)}))$$

3.2 深度神经网络

3.2.1 LSTM

步骤	公式	作用
1. 遗忘门	$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$	决定丢弃多少旧信息 (0 到 1 之间)
2. 输入门	$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$	决定哪些新信息进入细胞状态
3. 候选状态	$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$	生成当前时刻的备选记忆
4. 细胞更新	$C_t = f_t C_{t-1} + i_t \tilde{C}_t$	更新长期记忆 (核心步骤)
5. 输出门	$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$	决定输出哪些部分到下一时刻
6. 最终输出	$h_t = o_t * \tanh(C_t)$	生成当前时刻的隐藏状态

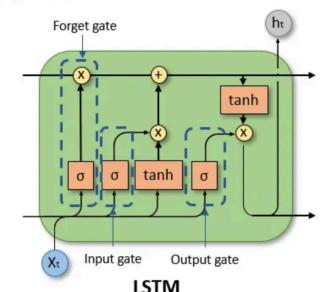


图 5 LSTM Block

Multilayer LSTM:

每个时间步在 $l - 1$ 层的输出作为该位置在 l 层的输入

3.2.2 Transformer LM

3.2.2.1 Transformer Block

与原始的 Transformer 相比，我们实现的 transformer block 有以下改动：

- Decoder Only，而不是 transformer 提出时的 Encoder - Decoder 结构
- Pre-Norm，将 norm 从 add 之后提到 attn/ffn 之前
- RoPE 位置编码
- Normalization: RMSNorm
- Activation: SwiGLU

① 为什么用这些设置？

经过调研，这些是当前主流 LLM 都在使用的结构

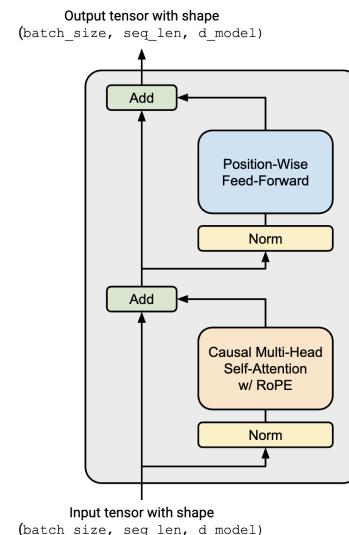


图 6 Decode Transformer Block

主要计算流程

设输入张量为 x ，该 Block 的计算流程如下：

$$\begin{aligned}
 x_{\text{norm}1} &= \text{Norm}(x) \\
 \text{Attn}(x_{\text{norm}1}) &= \underset{h=1}{\overset{\text{num_heads}}{\text{Concat}}} \left(\text{Softmax} \left(\frac{f_{\text{RoPE}}(Q_h) f_{\text{RoPE}}(K_h)^\top + \text{CausalMask}}{\sqrt{d_k}} \right) V_h \right) \\
 x' &= x + \text{Attn}(x_{\text{norm}1}) \\
 x_{\text{norm}2} &= \text{Norm}(x') \\
 \text{FFN}(x_{\text{norm}2}) &= \sigma(x_{\text{norm}2} W_1 + b_1) W_2 + b_2 \\
 \text{Output} &= x' + \text{FFN}(x_{\text{norm}2})
 \end{aligned}$$

与 LSTM 相比，它可以并行地处理整个序列，同时通过 Attention 机制捕捉长距离依赖关系。

一些用到的组件

① RMSNorm: Root-Mean Square Layer Normalization

$$\begin{aligned}
 \text{RMS}(x) &= \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \epsilon} \\
 \text{RMSNorm}(x)_i &= \frac{x_i}{\text{RMS}(x)} \cdot \gamma_i
 \end{aligned}$$

与 LayerNorm 相比，仅保留缩放，减少计算开销，收敛更快 [1]

① SwiGLU: Swish Gated Linear Unit

$$\text{SiLU}(x) = x \cdot \sigma(x)$$

$$\text{SwiGLU}(x, W_1, W_2, W_3) = W_2(\text{SiLU}(W_1 x)) \odot W_3 x$$

通过门控机制，模型可以动态控制哪些信息应该通过，增加了非线性表达的自由度。[2], [3]

② RoPE: Rotary Position Embedding

RoFormer: Enhanced Transformer with Rotary Position Embedding, Su et al., 2021

一种位置编码，核心思想是将特征向量看作是 2D 平面上的复数，并对其进行旋转，从而编码相对距离 $m - n$ 。[4]

二维的旋转：

$$f(x, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

其中 θ 是预定义的频率常数

推广到高维：将 d 维向量分成 $d/2$ 组，每组内部应用旋转

注意力分数 QK^\top 是关于旋转不变的，只跟相对位置 $m - n$ 有关

$$\langle f_q(q, m), f_k(k, n) \rangle = g(q, k, m - n)$$

所以加入 RoPE 之后，距离相同的两对 token 编码相同。

3.2.2.2 将 Transformer 用于分类任务

- 去掉 Output Embedding 层 (lm head)，改为分类头 (classification head)
- 具体地，从经过 Transformer 块之后的 token 序列中取出一个 token，接入一个 MLP 分类器
 - 对于 Decoder-Only 架构，选择最后一个 token
 - 对于 Encoder-Decoder 或者 Encoder-Decoder 架构，选择第一个 token ([CLS])

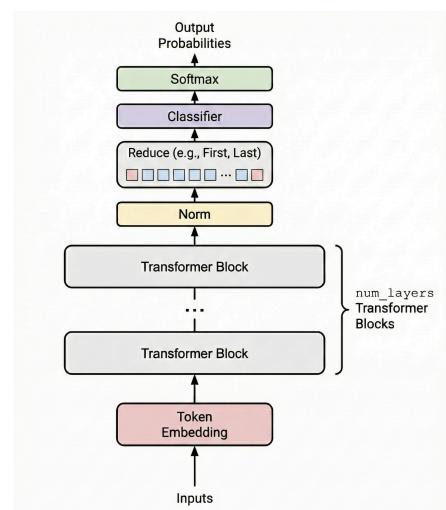


图 7 Transformer LM for Classification

3.2.3 模型实现

我们从零开始实现了整个模型和训练 pipeline:
tokenizer, model, dataloader, optimizer, trainer ...

```

1 .
2 └── tinyllm
3   └── cpp
4     └── bpe.hpp

```

tiktoken, transformers

~~torch.nn.*~~

~~torch.optim.*~~

`torch.nn.Parameter`

```
5 |   |   └── export.cpp
6 |   |   └── include
7 |   |   └── utils
8 |   └── network
9 |       └── functional.py
10 |      └── layers.py
11 |      └── models.py
12 |      └── multiplatform.py
13 |   └── optimize
14 |       └── functional.py
15 |       └── lr_scheduler.py
16 |           └── optimizers.py
17 |   └── tests
18 |       └── ...
19 |   └── tokenize
20 |       └── pretokenizer.py
21 |           └── tokenizer.py
22 |       └── train
23 |           └── checkpoint.py
24 |           └── dataset.py
25 |           └── train.py
26 └── ...
```

Attention 和 RoPE 的实现:

```
1 def scaled_dot_product_attention( python
2     query: Float[Tensor, "... len_q
3         dim_k"],
4     key: Float[Tensor, "... len_k
5         dim_k"],
6     value: Float[Tensor, "... len_k
7         dim_v"],
8     mask: Bool[Tensor, "... len_q
9         len_k"] | None = None,
10 ) -> Float[Tensor, "... len_q dim_v"]:
11     scores = einops.einsum(query, key,
12         "... len_q dim_k, ... len_k dim_k -
13         > ... len_q len_k")
14     scores = scores / key.shape[-1] **
15         0.5
16     if mask is not None:
17         scores.masked_fill_(~mask,
18             float("-inf"))
19     attn_value = softmax(scores, dim=-1)
20     return einops.einsum(attn_value,
21         value, "... len_q len_k, ... len_k
22         dim_v -> ... len_q dim_v")
```

```
1 class RoPE(nn.Module):
2     def _update_rotation(self, max_seq_len: int):
3         positions = torch.arange(max_seq_len,
4                                   dtype=torch.float32)
4         angles = einops.einsum(positions, self.freqs,
5                               "max_seq_len, half_d_k -> max_seq_len half_d_k")
6         rotate_x = torch.stack([torch.cos(angles),
7                                torch.sin(angles)], dim=-1).flatten(start_dim=-2)
8         rotate_y = torch.stack([-torch.sin(angles),
9                                torch.cos(angles)], dim=-1).flatten(start_dim=-2)
10        if self.device:
11            rotate_x = rotate_x.to(self.device)
12            rotate_y = rotate_y.to(self.device)
13        self.register_buffer("rotate_x", rotate_x,
14                             persistent=False)
15        self.register_buffer("rotate_y", rotate_y,
16                             persistent=False)
17        self.max_seq_len = max_seq_len
```

代码 2 RoPE 位置: tinyllm/tinyllm/network/layers.py

代码 1 Attention, 位置: tinyllm/tinyllm/network/functional.py

Transformer Block 实现: ([tinyllm/tinyllm/network/models.py](#))

```
1 class TransformerBlock(Module):
2     def __init__(
3         self,
4         d_model: int,
5         num_heads: int,
6         d_ff: int | None = None,
7         rope_theta: float | None = None,
8         rope_len: int | None = None,
9         device: torch.device | str | None = None,
10        dtype: torch.dtype | None = None,
11        norm_type: Literal["rms", "none"] = "rms",
12        norm_location: Literal["pre", "post"] = "pre",
```

```

13     ffn_activate: Literal["swiglu", "silu"] = "swiglu",
14     causal: bool = True,
15   ):
16     super().__init__()
17     self.ln1 = RMSNorm(d_model, device=device, dtype=dtype) if norm_type == "rms" else Identical()
18     self.attn = MultiheadSelfAttention(
19       ...
20     )
21     self.ln2 = RMSNorm(d_model, device=device, dtype=dtype) if norm_type == "rms" else Identical()
22     self.ffn = FeedForward(d_model, d_ff, activate=ffn_activate, device=device, dtype=dtype)
23     self.norm_location = norm_location
24
25   def forward(
26     self,
27     x: Float[torch.Tensor, "... seq_len d_model"],
28     len: Int[torch.Tensor, "..."] | None = None,
29   ):
30     if self.norm_location == "pre":
31       x = x + self.attn(self.ln1(x), sequence_length=len)
32       x = x + self.ffn(self.ln2(x))
33     elif self.norm_location == "post":
34       x = self.ln1(x + self.attn(x, sequence_length=len))
35       x = self.ln2(x + self.ffn(x))
36     else:
37       raise NotImplementedError(f"unsupported norm_location: {self.norm_location}")
38   return x

```

BPE Tokenizer 实现: (tinyllm/tinyllm/cpp/bpe.hpp)

```

1  inline auto
2  encode(const py::list& words, const py::list& merges, const py::dict& vocab, int num_threads, bool verbose = false) cpp
3  -> std::vector<int> {
4    std::vector<int> token_ids;
5    std::vector<std::vector<std::string>> words_vec;
6    std::unordered_map<std::pair<std::string, std::string>, int, pair_hash> merges_rank;
7    for (size_t rank = 0; const auto& item : merges) {
8      py::tuple merge = item.cast<py::tuple>();
9      std::string first = py::bytes(merge[0]).cast<std::string>(),
10        second = py::bytes(merge[1]).cast<std::string>();
11      merges_rank[std::make_pair(first, second)] = rank++;
12    }
13    for (auto item : words) {
14      py::tuple word = item.cast<py::tuple>();
15      std::vector<std::string> word_tokens;
16      for (auto token : word) {
17        word_tokens.push_back(token.cast<std::string>());
18      }
19      words_vec.push_back(word_tokens);
20    }
21    transform(
22      words_vec,
23      [&merges_rank](std::vector<std::string> word) {
24        std::vector<std::pair<std::pair<std::string, std::string>, int>> valid_pairs;
25        do {
26          valid_pairs.clear();
27          for (size_t i = 0; i + 1 < word.size(); i++) {
28            auto pair = std::make_pair(word[i], word[i + 1]);
29            if (merges_rank.contains(pair))
30              valid_pairs.push_back(std::make_pair(pair, merges_rank.at(pair)));
31          }
32          if (!valid_pairs.size()) break;
33          std::sort(
34            valid_pairs.begin(), valid_pairs.end(),
35            [&merges_rank](const auto& a, const auto& b) -> bool {
36              return a.second < b.second;

```

```

37             });
38         word = merge_token(word, valid_pairs[0].first);
39     } while (valid_pairs.size());
40     return word;
41 },
42 num_threads, verbose, "BPE encoding");
43 for (const auto& word : words_vec) {
44     for (const auto& token : word) {
45         py::bytes token_bytes = py::bytes(token);
46         if (vocab.contains(token_bytes)) {
47             token_ids.push_back(vocab[token_bytes].cast<int>());
48         } else {
49             throw std::runtime_error("Token not found in vocabulary: " + token);
50         }
51     }
52 }
53 return token_ids;
54 }

```

此外，我们尝试了使用 transformer 库微调模型，作为对比

```

1 class TransformersClassifier(Classifier):
2     def __init__(self, model_name, num_classes, **kwargs):
3         super().__init__()
4         self.model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=num_classes,
5 **kwargs)
6
7     def forward(self, x, len):
8         if len is not None:
9             seq_len = x.size(-1)
10            attention_mask = (torch.arange(seq_len, device=x.device).unsqueeze(0) < len.unsqueeze(-1)).to(x.dtype)
11        else:
12            attention_mask = (x != 0).to(x.dtype)
13
14        outputs = self.model(input_ids=x, attention_mask=attention_mask)
15        return outputs.logits

```

代码 3 app/classifier/transfomers.py

3.3 训练 pipeline

3.3.1 整体架构概览

- 入口：scripts/train.py 负责参数解析、数据加载、模型选择、优化器/调度器/损失函数构建，随后交给 Trainer 统一训练与评估。
- 数据处理：app/dataloader 将 TSV 文本为张量，完成分割、填充、对齐。
- 模型后端：三个可选模型，使用统一接口 Classifier，共享训练循环与推理接口。
 - LSTMClassifier
 - TinyLLMClassifier
 - TransformersClassifier
- 训练循环：Trainer 实现学习率调度、梯度平滑、定期验证、W&B 记录、最优权重保存与提交文件生成。

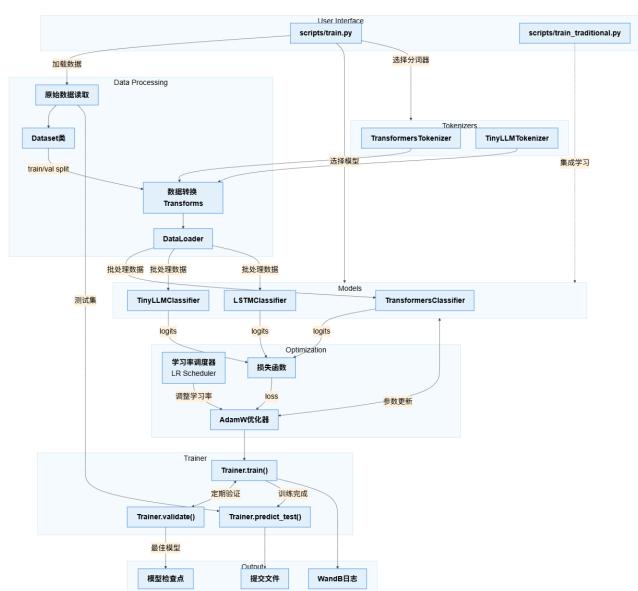


图 8 训练 pipeline 架构

3.3.2 数据预处理

- 标签策略：按 SentenceId 取最长短语作为该句标签，确保划分时标签一致，不让同一个句子不同短语同时出现在训练/验证集。
- 划分：8:2 训练/验证，SentenceId 分层抽样。

```

1 def split(self, test_size: float = 0.2, random_state: int | None = None):
    df = self.data
    longest = (
        df.assign(_len=df["Phrase"].str.len())
        .sort_values(["SentenceId", "_len"], ascending=[True, False])
        .drop_duplicates("SentenceId")
    )
    train_ids, val_ids = train_test_split(
        sentence_ids,
        test_size=test_size,
        random_state=random_state,
        stratify=sentence_labels,
    )

```

代码 4 app/dataloader/dataset.py

- 数据加载：入口脚本构建 Dataset → split → DataLoader（train 数据集打乱；collate_fn 负责 padding 与长度）。

```

1 dataset = dataloader.Dataset(
2     pd.read_csv(train_path, sep="\t"),
3     transform=dataloader.transform.to_tensor(tokenizer, device=args.device),
4 )
5 train, valid = dataset.split(test_size=0.2, random_state=42)
6 train_dataloader = DataLoader(
7     train,
8     batch_size=args.batch_size,
9     shuffle=True,
10    collate_fn=dataloader.transform.collate_padding(device=args.device),
11 )
12 valid_dataloader = DataLoader(
13     valid,
14     batch_size=args.batch_size,
15     shuffle=False,
16     collate_fn=dataloader.transform.collate_padding(device=args.device),
17 )
18 test = pd.read_csv(test_path, sep="\t", dtype=str, na_filter=False)

```

代码 5 scripts/train.py

3.3.3 Tokenizer

- TinyLLM 模型使用的 tokenizer：从 ckpts/tokenizer/<name>-<vocab_size> 导入自训 BPE。
- HuggingFace 使用的 tokenizer（Transformers/LSTM）：TransformersTokenizer，自动处理 pad/eos。

app/dataloader/transform.py 中包含两个用于统一数据格式的函数：

- 张量化：to_tensor 将文本编码为 input_ids

```

1 def to_tensor(tokenizer: Tokenizer, device: torch.device | str | None = ACCL_DEVICE):
2     def transform(text: str, label: int):
3         tokenized = tokenizer.encode(text)
4         input_ids = torch.tensor(tokenized, dtype=torch.int64, device=device)
5         output_label = torch.tensor(label, dtype=torch.int64, device=device)
6         return input_ids, output_label
7     return transform

```

代码 6 app/dataloader/transform.py

- 对齐: collate_padding 统一 batch 长度并给出 lengths/labels

```

1 def collate_padding(device: torch.device | str | None = ACCL_DEVICE):
2     def collate(batch: list[dict[str, torch.Tensor]]):
3         input_ids = [item["text"] for item in batch]
4         labels = torch.tensor([item["label"] for item in batch], dtype=torch.int64, device=device)
5         max_len = max(len(ids) for ids in input_ids)
6         lengths = torch.tensor([len(ids) for ids in input_ids], dtype=torch.int64, device=device)
7         padded_input_ids = torch.zeros((len(batch), max_len), dtype=torch.int64, device=device)
8         for i, ids in enumerate(input_ids):
9             padded_input_ids[i, : len(ids)] = ids
10        return {"input_ids": padded_input_ids, "lengths": lengths, "labels": labels}
11    return collate

```

代码 7 app/dataloader/transform.py

3.3.4 模型后端与接口

- 统一接口: Classifier 提供 forward 与 predict, forward 由具体的子类实现, predict 会负责 tokenizer 编码、padding 以及 argmax logits, 确保三种后端一致的推理路径。

```

1 class Classifier(torch.nn.Module, ABC):
2     @abstractmethod
3     def forward(
4         self,
5         x: Int[torch.Tensor, "... seq_len"],
6         len: Int[torch.Tensor, "..."] | None = None,
7     ) -> Float[torch.Tensor, "... num_classes"]:
8         pass
9
10    def predict(self, phrases: list[str], tokenizer: Tokenizer) -> list[int]:
11        # ...

```

代码 8 app/classifier/base.py

- TransformersClassifier: HF AutoModelForSequenceClassification, 根据长度生成 attention mask

```

1 class TransformersClassifier(Classifier):
2     def __init__(self, model_name: str, num_classes: int, **model_args,):
3         super().__init__()
4         self.model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=num_classes,
5             ignore_mismatched_sizes=True, **model_args)
6
7     def forward(
8         self,
9         x: Int[torch.Tensor, "... seq_len"],
10        len: Int[torch.Tensor, "..."] | None = None,
11    ) -> Float[torch.Tensor, "... num_classes"]:
12        if len is not None:
13            seq_len = x.size(-1)
14            attention_mask = (torch.arange(seq_len, device=x.device).unsqueeze(0) < len.unsqueeze(-1)).to(x.dtype)
15        else:
16            attention_mask = (x != 0).to(x.dtype)
17        outputs = self.model(input_ids=x, attention_mask=attention_mask)
18
19    return outputs.logits

```

代码 9 app/classifier/transformers.py

- LSTMClassifier: Embedding + (Bi)LSTM，支持 first/mean/last 聚合后接 MLP

```

1 packed_x = pack_padded_sequence(x, len.cpu(), batch_first=True, enforce_sorted=False)
2 lstm_out_packed, _ = self.lstm(packed_x)
3 lstm_out, _ = pad_packed_sequence(lstm_out_packed, batch_first=True)
4 ... # reduction -> classifier MLP

```

代码 10 app/classifier/lstm.py

- TinyLLMClassifier: tinyllm Transformer，可设非因果/因果掩码，first/mean/last 聚合后接轻量 MLP，支持加载/冻结/定步解冻底座。

```

1 self.model = models.TransformerModel(..., causal=causal)
2 def forward(self, x, len=None):
3     x = self.model(x, len=len, lm_head=False)
4     x = self.reduction(x, len)
5     return self.classifier(x)

```

代码 11 app/classifier/tinyllm.py

3.3.5 训练配置与调度

- 选择后端模型: --classifier {transformers,lstm,tinyllm}
- 设置超参: --epoch/--batch_size/--lr/--valid_interval/--output_dir.....
- 支持 checkpoint I/O 与 W&B。
- 优化器: TinyLLM 实现的 AdamW (betas=(0.9,0.999), eps=1e-8, weight_decay=0.01)。
- 学习率: constant 或 cosine 调度, warmup_ratio 线性预热; 若提供外部调度器, 则 _apply_warmup 直接调用其 update。
- 损失: cross_entropy; 若启用 label_smoothing, 仅训练期套平滑, 验证阶段始终使用原始损失函数。

3.3.6 优化与正则策略

- Dropout: LSTM 堆叠层间 (除最后层) 与头部 MLP; Transformer 采用 HF/tinyllm 默认配置。
- Label smoothing: 缓解过拟合与过度自信, 验证阶段关闭以便真实评估。
- 权重衰减: 0.01, 抑制大权重。

- 冻结/解冻 (TinyLLM): 可加载 base ckpt 后冻结底座, `release_steps` 自动解冻, 兼顾稳定性与收敛速度。
- 验证与选优: 每 `valid_interval` step 触发 `validate`, 输出 loss/acc/分类报告/混淆矩阵, 新低则保存 best (若开启)。

3.3.7 训练-验证-测试流程

- 构建 DataLoader → 选择 tokenizer & 模型 → 创建优化器/调度器/损失。
- 训练循环 (`Trainer.train`):
 1. 如果达到 `valid_step`, 则运行验证 `validate()`;
 2. 预热/调度 lr;
 3. 前向→loss→backward→step;
 4. 可选 step hook (如 TinyLLM 解冻);
 5. W&B 日志。
- 验证: `Trainer.validate` 评估 loss/acc/分类报告/混淆矩阵, 更新 best。
- 测试/提交: 若提供 `--submit_file`, 调用 `model.predict` 生成 CSV 提交文件。

四、实验过程

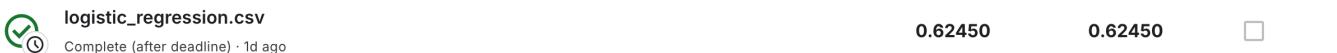
4.1 Logistic Regression

训练参数:

```
1 make_pipeline(  
2     TfidfVectorizer(ngram_range=(1,2), max_features=50000),  
3     LogisticRegression(max_iter=2000, multi_class="multinomial", solver="saga"),  
4 )
```

python

结果: Test Accuracy = 0.62450, 作为 baseline



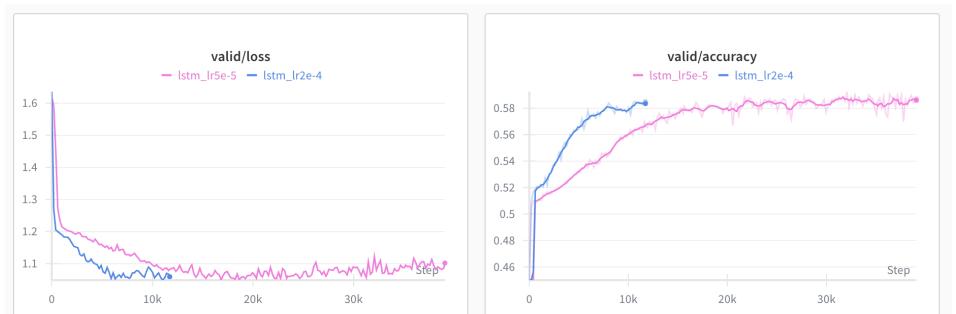
尝试了与深度学习方法集成, 但效果不如直接使用深度学习模型。

4.2 LSTM 模型

我们训练了一个简单的 LSTM 模型:

```
1 LSTMClassifier(  
2     embedding_dim=128,  
3     hidden_dim=128,  
4     output_dim=5,  
5     n_layers=2,  
6     bidirectional=True,  
7     dropout=0.3,  
8 )
```

python



最终 Accuracy 在 0.58 左右

4.3 Transformer 模型

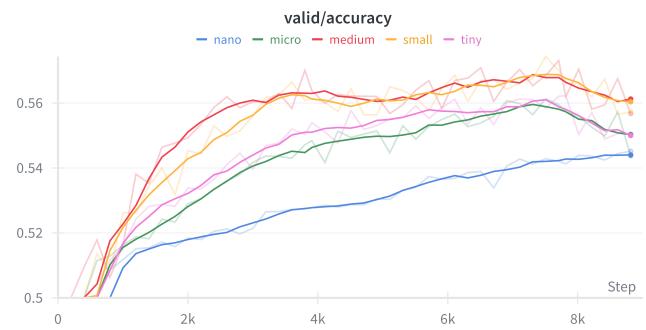
4.3.1 直接训练

我们首先尝试直接在本题数据集上从头开始训练一个 Transformer 模型

1. 模型大小实验

(参数):

大小	num_layers	d_model	num_heads	val/acc
nano	4	64	2	0.544 (-2.5%)
micro	4	128	4	0.558
tiny	4	256	8	0.560 (+0.4%)
small	4	512	16	0.567 (+1.6%)
medium	8	512	16	0.567 (+1.6%)
large	12	768	16	N/A
x-large	16	1024	16	N/A

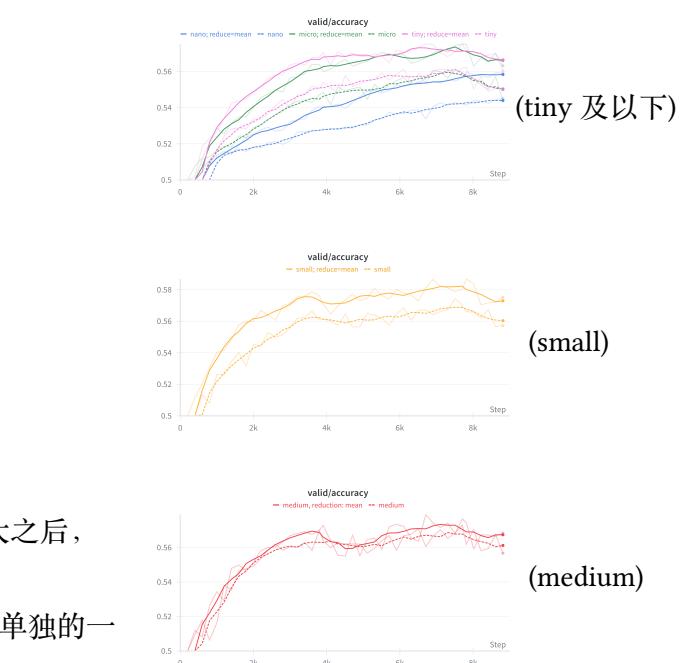


实验结果：随着模型大小提升，准确率稳定提升，但除 nano 外其他模型均会过拟合

2. 规约方法方法实验：

- last: 选择最后一个 token 的特征接到分类头
- mean: 选择所有 token 的特征的平均值接到分类头

模型大小	baseline(last)	mean
nano	0.544	0.558 (+2.6%)
micro	0.558	0.573 (+2.7%)
tiny	0.560	0.571 (+2.0%)
small	0.567	0.580 (+2.3%)
medium	0.567	0.573 (+1.1%)



可以看到，mean 方法明显比 last 方法好，但模型变大之后，规约方法的影响减小

可能是因为在模型层数以及参数量不够大的情况下，单独的一个 token 无法很好地融合整个句子的信息

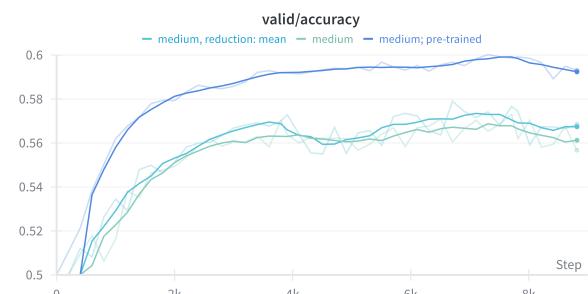
4.3.2 两阶段训练

为了充分发挥大模型的性能，我们采用两阶段训练方法：

先在通用文本上预训练，然后在本题数据集上微调

具体地，我们在 OpenWebText (24.2GB) [5] / TinyStories (1GB) [6] 数据集上训练 next-token prediction 任务
加入预训练阶段后，模型性能有明显提升，尽管预训练的任务和数据集都与本题不完全匹配

模型大小	baseline	+mean	+pretrain
tiny	0.560	0.571 (+2.0%)	0.588 (+5.0%)
medium	0.567	0.573 (+1.1%)	0.597 (+5.3%)



试验一下先冻结 base_model，只训练分类头，然后在中途解冻的效果

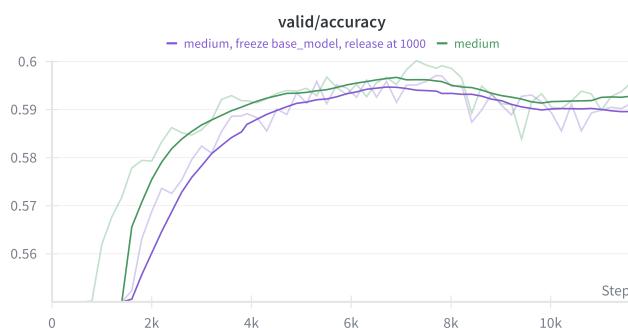


图 11 freeze

结果变差了，可能是由于模型自身有效训练时间变短了

使用更大的模型，更久的预训练：

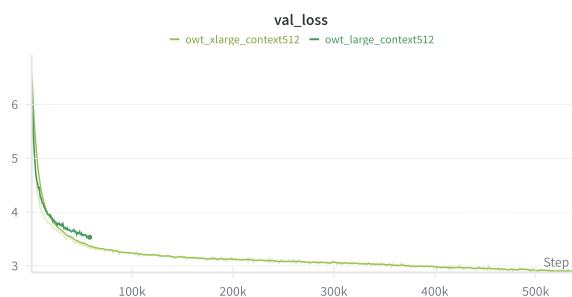


图 12 x-large 预训练 Loss 曲线

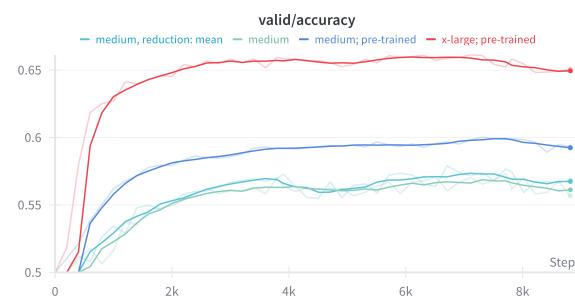


图 13 微调，红线：x-large，蓝线：medium

结果：

模型大小	val/acc
tiny	0.588
medium	0.597 (+1.5%)
x-large	0.660 (+12.2%)

最终在 x-large 模型上达到了最高 0.66 的验证集准确率，提交结果 (test acc: 0.67854)：



4.4 微调 transformers 库提供的预训练模型

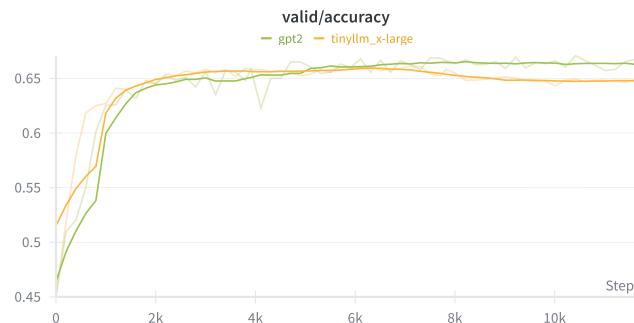


图 14 tinyllm vs gpt2

在过拟合之前我们的 tinyllm 模型与 gpt2 不相上下，甚至正确率略好一些

4.4.1 学习率调度实验

在微调大规模预训练 Transformers 模型时，学习率（Learning Rate）的设置不仅影响收敛速度，更决定了训练的成败。本实验针对 RoBERTa-Large 模型进行了详尽的超参数搜索。

4.4.1.1 学习率过大导致的类别塌陷

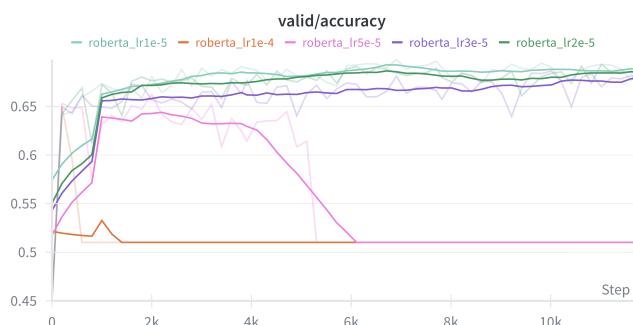


图 15 不同学习率结果

实验初期，我们尝试使用较高的学习率 $lr = 1 \times 10^{-4}$ 。结果模型在约 1000 个 iteration 后，预测结果开始集中于类别 2，准确率降至 0.51 左右（等于类别 2 的占比），此后也一直保持在这一水平。

```

Epoch 1: 51%
Validation Loss: 1.2861
Validation Accuracy: 0.5102
Distribution of predictions: [0, 0, 31022, 0, 0]
Classification Report:
/home/nvme01/chenpu/duas-ml-local/.venv/lib/python3.12/site-pac
being set to 0.0 in labels with no predicted samples. Use `zero_
warn_nrf(average=_modifier_, f"metric_.canalize()")` instead.

```

图 16 类别塌陷

原因在于：

- RoBERTa 拥有 $125M \sim 3.5B$ 参数，其预训练权重已处于高度优化的极小值区域。过大的更新步长会产生灾难性遗忘，破坏模型已掌握的语义提取能力。
- 由于随机初始化的分类头在初期会产生巨大的梯度，配合高学习率，这股冲击力会直接传导至底层权重，导致模型为了快速降低 Loss 而选择“全输出单一高频标签”的局部最优解。
- 值得注意的是（见图 15） 5×10^{-5} 这一在 Base 模型上常用的学习率，在训练后期（约 5000 step 后）也出现了准确率急剧下滑的崩溃现象，进一步证明了 Large 模型对高学习率的容忍度极低。

RoBERTa 和 BERT 的论文中都给出了研究者使用的超参数设置。

Hyperparam	RACE	SQuAD	GLUE
Learning Rate	1e-5	1.5e-5	{1e-5, 2e-5, 3e-5}
Batch Size	16	48	{16, 32}
Weight Decay	0.1	0.01	0.1
Max Epochs	4	2	10
Learning Rate Decay	Linear	Linear	Linear
Warmup ratio	0.06	0.06	0.06

Table 10: Hyperparameters for finetuning RoBERTa_{LARGE} on RACE, SQuAD and GLUE.

图 17 RoBERTa 论文给出的超参数设置 [7]

A.3 Fine-tuning Procedure

For fine-tuning, most model hyperparameters are the same as in pre-training, with the exception of the batch size, learning rate, and number of training epochs. The dropout probability was always kept at 0.1. The optimal hyperparameter values are task-specific, but we found the following range of possible values to work well across all tasks:

- Batch size: 16, 32
- Learning rate (Adam): 5e-5, 3e-5, 2e-5
- Number of epochs: 2, 3, 4

We also observed that large data sets (e.g., 100k+ labeled training examples) were far less sensitive to hyperparameter choice than small data sets. Fine-tuning is typically very fast, so it is reasonable to simply run an exhaustive search over the above parameters and choose the model that performs best on the development set.

图 18 BERT 论文给出的超参数设置建议 [8]

4.4.1.2 学习率调度策略

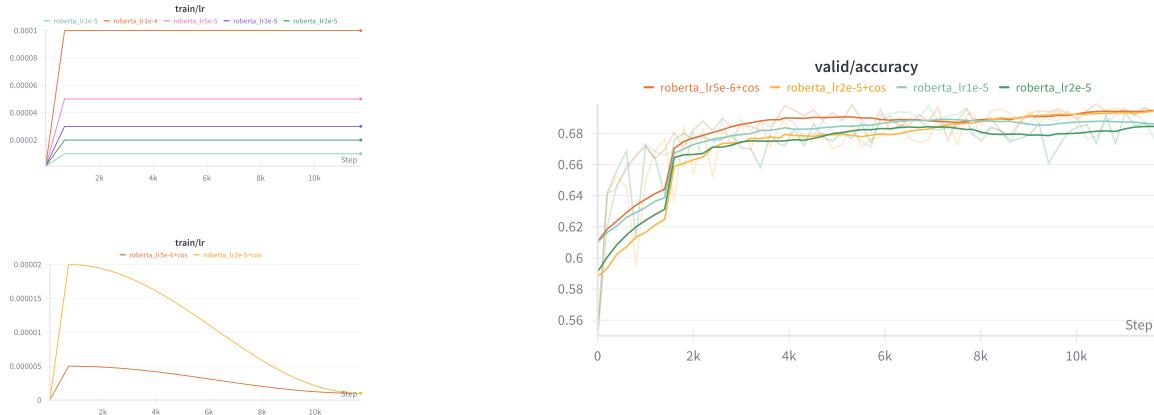
遵循论文中的设置，为了平衡训练早期的稳定性和后期的精准度，我们引入了带预热（Warmup）的调度机制。

- 线性预热：在前 6% 的步数中，学习率从 0 线性增加至设定值（如 2×10^{-5} ）。这为随机初始化的分类头提供了“缓冲期”，避免初始梯度冲击破坏底座权重。
- 余弦衰减：预热结束后，学习率按余弦曲线缓慢下降。相比恒定学习率（Constant），余弦衰减在训练后期能以更细微的步长进行参数微调，有效提高了最终的验证集准确率。

通过对比实验，我们确定了适用于本任务的超参数空间：

学习率	调度策略	训练表现	验证集峰值精度	稳定性
1×10^{-4}	Warmup+Const	极速崩溃 (1k step)	0.51	极差
5×10^{-5}	Warmup+Const	中期崩溃 (4k step)	0.64	不稳定
2×10^{-5}	Warmup+Const	缓慢上升，略有震荡	0.67	良好

学习率	调度策略	训练表现	验证集峰值精度	稳定性
2×10^{-5}	Warmup+Cosine	平滑收敛，泛化最优	0.69	极佳
5×10^{-6}	Warmup+Cosine	极其稳定但收敛缓慢	0.68	极佳



最终最好的结果 (test acc: 0.71186):

roberta_lr1e-5.csv
Complete (after deadline) · 1d ago · <https://wandb.ai/cauphenuny-university-of-chinese-academy-of-sciences/UCAS%2...> 0.71186 0.71186

五、分析与总结

5.1 最终模型性能对比

5.1.1 核心指标对比

下表展示了四个模型在验证集上的最佳性能指标。可以看到，不同架构的模型在情感分析任务上表现出了显著差异。

模型架构	架构类型	Val Accuracy	Val Loss	Macro-F1	极端情感召回 (Class 0/4)
LSTM	RNN	58.50%	1.0592	0.41	极差 (13% / 21%)
TinyLLM	Decoder-only	65.95%	0.8335	0.53	一般 (23% / 36%)
GPT-2	Decoder-only	66.89%	0.7949	0.57	一般 (30% / 45%)
RoBERTa	Encoder-only	69.81%	0.7256	0.62	较好 (47% / 57%)

表 4 四大模型最佳性能指标对比

结论：虽然 Accuracy 的差距看似只有 11% (59% vs 70%)，但 Macro-F1 的差距高达 0.21，这说明架构选择对性能有决定性影响。

5.1.2 架构分析

1. LSTM (RNN 架构):

- 验证集 Loss 始终较高 (停留在 1.05 左右)。

- 模型难以捕捉文本深层的语义特征，性能在 58% 左右触达天花板。
 - 受限于序列建模能力，在长文本和复杂语义理解上表现较差。
2. **Decoder-only Transformer 架构 (TinyLLM / GPT-2):**
- 两个 Decoder-only 模型表现接近：TinyLLM (65.95%) 和 GPT-2 (66.89%) 的准确率仅相差约 1%。
 - 验证集 Loss 稳定在 0.79-0.83 之间，GPT-2 训练过程更加平稳，TinyLLM 由于预训练知识不足，在微调后期出现过拟合现象。
 - 在极端情感识别上表现一般，Class 0/4 的召回率在 23-45% 之间。
3. **Encoder-only Transformer 架构 (RoBERTa):**
- 准确率达到 69.81%，比 Decoder-only 模型高出约 3-4 个百分点。
 - 验证集 Loss 最低 (0.73)，收敛更稳定。
 - 在极端情感识别上表现突出，Class 0/4 的召回率达到 47%/57%，显著优于 Decoder-only 模型。

5.1.3 极端情感捕捉能力分析 (Recall Analysis)

本任务最大的难点在于 类别不平衡 (Neutral 占 50%)。我们重点关注模型对 **Negative (0)** 和 **Positive (4)** 的识别能力。

真实标签	LSTM Recall	TinyLLM Recall	GPT-2 Recall	RoBERTa Recall
0 (极负)	13%	23%	30%	47%
1 (负面)	40%	52%	57%	61%
2 (中性)	81%	80%	81%	79%
3 (正面)	40%	57%	55%	62%
4 (极正)	21%	36%	45%	57%

表 5 各类别召回率 (Recall) 详细对比

数据洞察：

- **LSTM** 在极端情感识别上表现最差，倾向于把所有极端样本都预测为中性，Class 0/4 的召回率仅 13%/21%。
- **TinyLLM** 与 **GPT-2** 表现接近，在极端情感识别上有所提升，但仍明显低于 Encoder-only 架构。两个模型的 Class 0/4 召回率在 23-45% 之间。
- **RoBERTa** 在极端情感识别上表现最佳，Class 0/4 的召回率达到 47%/57%，比 Decoder-only 模型高出约 15-20 个百分点。这进一步验证了双向注意力机制在理解型任务上的优势。

5.1.4 混淆矩阵形态分析

对比四个模型的混淆矩阵，可以观察到一些特征：

- 中心坍缩 (LSTM):** 预测结果高度集中在 Label 2 (Neutral) 列，呈现垂直长条状。模型采取了“由于不确定，所以猜中性”的保守策略。
- 对角线强化 (Transformer):** 混淆矩阵的对角线明显变亮，特别是在 Label 0 和 Label 4 的角落。这代表模型真正理解了语义，而非利用数据分布漏洞。双向注意力机制使得模型能够更好地捕捉全局语义信息。
- 邻类漂移 (Common Issue):** 所有模型的主要错误都集中在相邻类别（如把 0 预测成 1，把 4 预测成 3）。几乎没有模型会将 0 预测成 4。这说明模型都成功学到了情感的连续性特征。

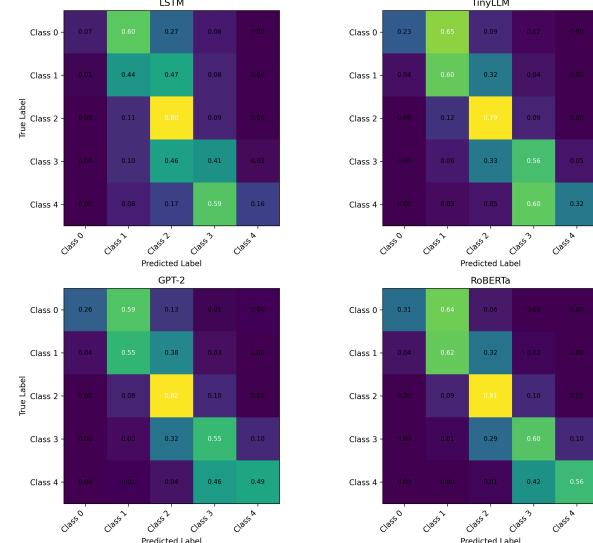


图 21 四大模型混淆矩阵对比(归一化后)

5.2 总结

通过之前的实验以及模型表现，我们得出以下结论：

- 架构决定上限：** LSTM 受限于结构，无法有效处理长文本和复杂语义；Transformer 架构显著优于 RNN。
- Decoder-only vs Encoder-only 的架构差异：**
 - Decoder-only 架构 (TinyLLM/GPT-2)** 通过因果掩码进行自回归训练，更适合生成任务。在分类任务中，对全局语义的理解相对受限，准确率在 66% 左右。
 - Encoder-only 架构 (RoBERTa)** 通过双向注意力机制可以同时利用上下文的所有信息，对文本的全局语义理解更强，更适合理解型任务，准确率达到 70%，在极端情感识别上表现尤为突出。
- 预训练的重要性：** 大规模预训练使模型携带的丰富先验知识是解决 **Cold Start** 和 类别不平衡问题的关键。预训练使得模型能够更好地泛化到小样本场景。
- Accuracy 区分度不足：** LSTM 的 58% Accuracy 背后是接近 0 的极端情感召回率；Transformer 模型的约 70% Accuracy 代表了更均衡、更可用的模型。

5.2.1 最终提交结果

模型	提交结果	排名
Logistic Regression	0.62450	284 / 861
集成 (LR+RoBERTa)	0.68507	6 / 861
LSTM	0.58886	508 / 861
TinyLLM	0.67854	8 / 861
RoBERTa	0.71186	3 / 861

logistic_regression.csv Complete (after deadline) · 1d ago	0.62450	0.62450	<input type="checkbox"/>
merge.csv Complete (after deadline) · 2d ago	0.68507	0.68507	<input type="checkbox"/>
lstm_2layer_lr1e-4.csv Complete (after deadline) · now	0.58886	0.58886	<input type="checkbox"/>
tinyllm_x-large_lr2e-5.csv Complete (after deadline) · 8h ago	0.67854	0.67854	<input type="checkbox"/>
roberta_lr1e-5.csv Complete (after deadline) · 1d ago · https://wandb.ai/cauphenuny-university-of-chinese-academy-of-sciences/UCAS%2...	0.71186	0.71186	<input type="checkbox"/>

六、附录

6.1 Contributions

(代码: <https://github.com/cauphenuny/ucas-ml>)

- 袁晨圃: 设计并实现代码框架, 实现 LSTM 模型、TinyLLM 各个组件和 WebUI, 完成主要实验, 撰写报告模型设计和实验过程部分
- 李知谦: 实现 transformers 微调模型和 Trainer, 撰写报告训练 pipeline 与实验过程部分
- 许天祺: 分析数据集、总结与分析模型实验结果, 撰写报告数据集与结果分析部分
- 齐胡鑫: 实现传统 ML 模型, 完成对应实验和撰写报告相应部分

6.2 References

- [1] B. Zhang and R. Sennrich, “Root Mean Square Layer Normalization,” *CoRR*, 2019, [Online]. Available: <http://arxiv.org/abs/1910.07467>
- [2] S. Elfwing, E. Uchibe, and K. Doya, “Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning,” *CoRR*, 2017, [Online]. Available: <http://arxiv.org/abs/1702.03118>
- [3] N. Shazeer, “GLU Variants Improve Transformer,” *CoRR*, 2020, [Online]. Available: <https://arxiv.org/abs/2002.05202>
- [4] J. Su, Y. Lu, S. Pan, B. Wen, and Y. Liu, “RoFormer: Enhanced Transformer with Rotary Position Embedding,” *CoRR*, 2021, [Online]. Available: <https://arxiv.org/abs/2104.09864>
- [5] A. Gokaslan, V. Cohen, E. Pavlick, and S. Tellex, “OpenWebText Corpus.” 2019.
- [6] R. Eldan and Y. Li, “TinyStories: How Small Can Language Models Be and Still Speak Coherent English?” [Online]. Available: <https://arxiv.org/abs/2305.07759>
- [7] Y. Liu *et al.*, “RoBERTa: A Robustly Optimized BERT Pretraining Approach,” 2019, [Online]. Available: <https://arxiv.org/abs/1907.11692>
- [8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” 2019, [Online]. Available: <https://arxiv.org/abs/1810.04805>
- [9] R. Socher *et al.*, “Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank,” pp. 1631–1642, Oct. 2013.