

# 文本相似度实验报告

---

## 一、问题描述

### 1.1 待解决问题的解释

### 1.2 问题的形式化描述

## 二、系统

### 2.1 系统架构

### 2.2 各部分介绍

#### 2.2.1 文本预处理模块

#### 2.2.2 特征提取模块

#### 2.2.3 训练模块

#### 2.2.4 推理的 pipeline

### 2.3 算法的伪代码

## 三、实验

### 3.1 实验环境

### 3.2 数据

### 3.3 实验结果

## 四、总结和展望

### 4.1 总结

### 4.2 展望

王拓

班级：2021219112

学号：2021213490

## 一、问题描述

---

### 1.1 待解决问题的解释

文本相似度是指通过自然语言处理技术来衡量两段文本之间的语义相似性。

这种相似性不仅包括词汇层面的相似性，还包括句法和语义层面的相似性。例如，两段文本可能使用了不同的词汇，但表达的意思是相同的，或者两段文本在结构上存在差异，但传达的核心信息是一致的。文本相似度在自然语言处理领域具有广泛的应用，如信息检索、新闻推荐、智能客服等，可以帮助用户快速找到与其查询相关的信息，提高用户体验，具有很高的商业价值。

### 1.2 问题的形式化描述

给定两个文本  $D_1$  和  $D_2$ ，我们需要计算它们的相似度  $S(D_1, D_2)$ 。其中， $S(D_1, D_2)$  的取值范围是  $[0,1]$ 。当 $S(D_1, D_2)=1$  时，说明两段文本在语义上完全相同。当 $S(D_1, D_2) =0$  时，表示两者在语义上完全不同。在实际应用过程中，我们通过比较 $S(D_1, D_2)$  和一个阈值  $\theta$  判断这两个文本是否相似。如果  $S(D_1, D_2)>\theta$ ，我们认为这两段文本是相似的，否则我们认为两段文本是不相似的。

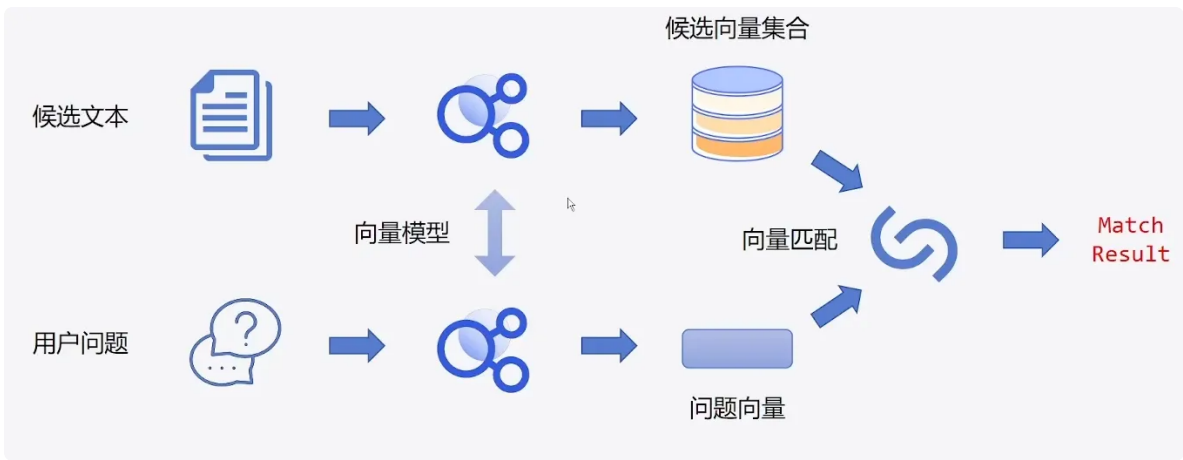
## 二、系统

### 2.1 系统架构

其实这种文本匹配有两种基本的思路，假设我们的数据集一个条目如下：

`{"sentence1": "找一部小时候的动画片", "sentence2": "求一部小时候的动画片。谢了", "label": "1"}`  
分别是sen1、sen2，还有对应的相似度label

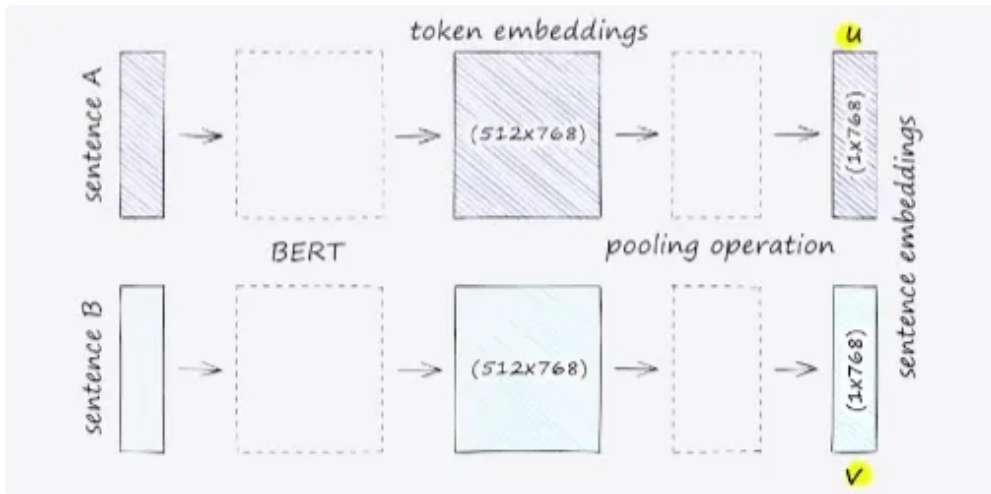
- 思路一：输入句子对，对是否相似进行学习。我们可以把数据变成：  
**CLS+SEN1+SEP+SNE2+SEP** 输入到我们的模型，然后让它对对应的标签进行学习，这样就在推理的时候就可以输入两个句子，然后对其进行标签的预测。**优点是模型可以看见两个句子的信息，缺点是把两个句子组合起来消耗资源大**
- 思路二：分别对两个句子进行编码，然后过一遍模型分别获取其向量表示，最后根据余弦相似度判断两个句子是否相似。**优点是开销比较小**



本系统包括以下几个核心组件：

- 文本预处理模块：**首先是加载 dataset, 构建tokenizer,然后针对我们数据的进行分词，例如 `k:v[sen1:sen2]`,然后把构建好的数据变成 token,例如：`{'input_ids': [[101, 2617, 2617, 779, 6206,`





## 2.2 各部分介绍

### 2.2.1 文本预处理模块

```

1  import torch
2
3  tokenizer = AutoTokenizer.from_pretrained("/home/wangtuo/workspace/Homework/Huggingface/models--hfl--chinese-macbert-base")
4
5  # 处理数据函数
6  def process_function(examples):
7      sentences = []
8      labels = []
9      for sen1, sen2, label in zip(examples['sentence1'], examples['sentence2'], examples['label']):
10         sentences.append(sen1)
11         sentences.append(sen2)
12         labels.append(1 if int(label) == 1 else -1)
13
14     tokenizer_examples = tokenizer(sentences, max_length=128, truncation=True, padding='max_length')
15     tokenizer_examples = {k: [v[i: i + 2] for i in range(0, len(v), 2)] for k, v in tokenizer_examples.items()}
16     tokenizer_examples['labels'] = labels
17     return tokenizer_examples
18
19 tokenizer_datasets = datasets.map(process_function, batched=True, remove_columns=datasets["train"].column_names)
20 tokenizer_datasets

```

分词器:

- 使用 `AutoTokenizer` 构建分词器，分词器包括了对句子单词的切分，实现中文词到字典 ids 的映射，`token_types` 表示输入属于哪一个句子，然后是 `attention_mask` 表示输入的那一部分是有效的，哪部分是 padding 的。

数据处理函数，处理我们输入的 datasets:

- 重要的是·我们要把数据字典变成 `k:[v[1,2]]` 就是一个字典的索引对应两个句子，`sen1` 和 `sen2`。
- 然后我们把数据过一遍分词器。

### 2.2.2 特征提取模块

```

1 class DualModel(BertPreTrainedModel):
2
3     def __init__(self, config: PretrainedConfig, *inputs, **kwargs):
4         super().__init__(config, *inputs, **kwargs)
5         self.bert = BertModel(config)
6         self.post_init()
7
8     def forward(
9         self,
10        input_ids: Optional[torch.Tensor] = None,
11        attention_mask: Optional[torch.Tensor] = None,
12        token_type_ids: Optional[torch.Tensor] = None,
13        position_ids: Optional[torch.Tensor] = None,
14        head_mask: Optional[torch.Tensor] = None,
15        inputs_embeds: Optional[torch.Tensor] = None,
16        labels: Optional[torch.Tensor] = None,
17        output_attentions: Optional[bool] = None,
18        output_hidden_states: Optional[bool] = None,
19        return_dict: Optional[bool] = None,
20    ):
21        return_dict = return_dict if return_dict is not None else self.config.use_return_dict
22
23        # 分别获取sen1 和 sen2 的输出
24        senA_input_ids, senB_input_ids = input_ids[:, 0], input_ids[:, 1]
25        senA_attention_mask, senB_attention_mask = attention_mask[:, 0], attention_mask[:, 1]
26        senA_token_type_ids, senB_token_type_ids = token_type_ids[:, 0], token_type_ids[:, 1]
27
28        # 获取向量表示
29        senA_outputs = self.bert(
30            senA_input_ids,
31            attention_mask=senA_attention_mask,
32            token_type_ids=senA_token_type_ids,
33            position_ids=position_ids,
34            head_mask=head_mask,
35            inputs_embeds=inputs_embeds,
36            output_attentions=output_attentions,
37            output_hidden_states=output_hidden_states,
38            return_dict=return_dict,
39        )
40
41        senA_pooled_output = senA_outputs[1]
42
43        senB_outputs = self.bert(
44            senB_input_ids,

```

```

45         attention_mask=senB_attention_mask,
46         token_type_ids=senB_token_type_ids,
47         position_ids=position_ids,
48         head_mask=head_mask,
49         inputs_embeds=inputs_embeds,
50         output_attentions=output_attentions,
51         output_hidden_states=output_hidden_states,
52         return_dict=return_dict,
53     )
54
55     senB_pooled_output = senB_outputs[1] # [batch, hidden]
56
57     # 计算相似度
58     cos = CosineSimilarity()(senA_pooled_output, senB_pooled_output)
59     # [batch,1]
60
61     # 计算loss
62     loss = None
63     if labels is not None:
64         loss_fct = CosineEmbeddingLoss(0.3)
65         loss = loss_fct(senA_pooled_output, senB_pooled_output, labels)
66
67     output = (cos,)
68     return ((loss,) + output) if loss is not None else output

```

- `__init__` 方法中，创建一个 Bert 的实例。
- `forward` 方法，接收输入ID、注意力掩码、token类型ID、位置ID、头掩码、嵌入输入、标签、输出注意力、输出隐藏状态和返回字典等。
- 然后首先从输入ID中分离出两个句子的输入ID、注意力掩码和token类型ID。然后分别调用bert模型来获取两个句子的输出和池化输出（即句子的嵌入表示）。
- 之后使用余弦损失函数计算损失，使用余弦相似度计算来相似度

### 2.2.3 训练模块

```

1  train_args = TrainingArguments(output_dir="./dual_model",      # 输出文件夹
2                                per_device_train_batch_size=32,  # 训练时的b
   atch_size
3                                per_device_eval_batch_size=32,   # 验证时的ba
   tch_size
4                                logging_steps=10,                # log 打印
   的频率
5                                evaluation_strategy="epoch",     # 评估策略
6                                save_strategy="epoch",           # 保存策略
7                                save_total_limit=3,              # 最大保存
   数
8                                learning_rate=2e-5,              # 学习率
9                                weight_decay=0.01,               # weight_d
   ecay
10                               metric_for_best_model="f1",       # 设定评估
   指标
11                               load_best_model_at_end=True)      # 训练完成
   后加载最优模型

```

## 2.2.4 推理的 pipeline



```

1  from typing import Any
2
3
4  class SentenceSimilarityPipeline:
5
6      def __init__(self, model, tokenizer) -> None:
7          self.model = model.bert
8          self.tokenizer = tokenizer
9          self.device = model.device
10
11     def preprocess(self, senA, senB):
12         return self.tokenizer([senA, senB], max_length = 128, truncation=True, return_tensors="pt", padding = True)
13
14     def predict(self, inputs):
15         inputs = {k: v.to(self.device) for k, v in inputs.items()}
16         return self.model(**inputs)[1]
17
18     def postprocess(self, logits):
19         cos = CosineSimilarity()(logits[None, 0, :1], logits[None, 1, :]).squeeze().cpu().item()
20         return cos
21
22     def __call__(self, senA, senB, return_vector=False):
23         inputs = self.preprocess(senA, senB)
24         logits = self.predict(inputs)
25         result = self.postprocess(logits)
26         if return_vector:
27             return result, logits
28         else:
29             return result

```

- 模型的 pipeline 包括初始化、数据预处理、预测、后处理、以及调用的预测。
- 在初始化方面，要求输入我们的模型还有 tokenizer
- 在调用 pipeline 中，只需要输入 SenA 和 SenB 即可，最终将会返回两者的余弦相似度。

## 2.3 算法的伪代码

```

1  输入：文本D1， 文本D2， 阈值 $\theta$ 
2  输出：相似度分数 $S(D1, D2)$ ， 相似性判断
3
4  预处理模块：
5      对D1进行预处理，得到预处理后的文本D1_pre
6      对D2进行预处理，得到预处理后的文本D2_pre
7
8  特征提取模块：
9      使用词嵌入模型，将D1_pre转换为特征向量V1
10     使用词嵌入模型，将D2_pre转换为特征向量V2
11
12  相似度计算模块：
13     计算V1和V2的余弦相似度，得到相似度分数 $S(D1, D2)$ 
14
15  阈值设定与判断模块：
16     如果 $S(D1, D2) \geq \theta$ ：
17         输出： $S(D1, D2)$ ， "文本D1和D2相似"
18     否则：
19         输出： $S(D1, D2)$ ， "文本D1和D2不相似"

```

## 三、实验

### 3.1 实验环境

```

1  torch                1.12.1+cu113
2  torchaudio           0.12.1+cu113
3  torchvision          0.13.1+cu113
4  transformers         4.36.2

```

### 3.2 数据

在本实验中，我们使用了SimCLUE数据集的一个子集，该数据集是一个大规模的中文语义理解与匹配数据集，包含超过300万条数据。我们选取了其中的一个分支train\_pair\_1w.json，其中包含10,000条数据。我们将这10,000条数据分为两部分：8000条作为训练集，2000条作为测试集。数据集的下载链接 [https://github.com/CLUEbenchmark/SimCLUE/blob/main/datasets/train\\_pair\\_1w.json](https://github.com/CLUEbenchmark/SimCLUE/blob/main/datasets/train_pair_1w.json)

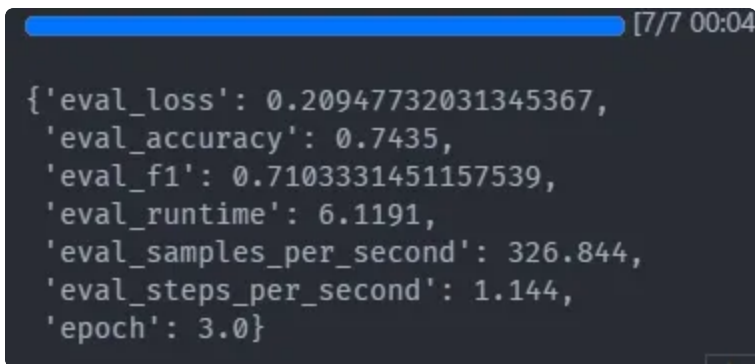
### 3.3 实验结果

训练时的数据：



Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.187900	0.213186	0.748500	0.693853
2	0.176400	0.203245	0.757000	0.697761
3	0.159400	0.201321	0.767500	0.723049
4	0.154200	0.197839	0.772500	0.721030
5	0.147800	0.196796	0.763000	0.710269

可以看到准确率和 F1 值都在不断地提高，但是在第 4 到第 5 个 Epoch 时，准确率没有提高，可能是发生了过拟合  
测试集的数据



```
{'eval_loss': 0.20947732031345367,  
  'eval_accuracy': 0.7435,  
  'eval_f1': 0.7103331451157539,  
  'eval_runtime': 6.1191,  
  'eval_samples_per_second': 326.844,  
  'eval_steps_per_second': 1.144,  
  'epoch': 3.0}
```

可以达到 0.7435 的准确率，0.7103 的 f1 值。这个结果表明，我们的模型在文本相似度任务上具有良好的性能。

## 四、总结和展望

### 4.1 总结

本实验中，我们构建了一个基于双塔BERT模型的文本相似度检测系统。通过使用SimCLUE数据集的一个子集进行训练和测试，我们的模型在文本相似度任务上展现出了良好的性能。在测试集上，模型达到了0.7435的准确率和0.7103的F1值，这表明我们的模型能够有效地捕捉文本之间的语义相似性。

实验过程中，我们使用了PyTorch和相关库来搭建和训练模型，同时采用了Hugging Face的Transformers库来简化模型构建和数据处理。我们的模型设计允许两个句子独立地通过BERT模型，然后计算它们嵌入表示的余弦相似度，这种结构不仅提高了计算效率，而且能够有效地衡量句子对的相似性。

### 4.2 展望

尽管我们的模型在文本相似度任务上取得了不错的成绩，但仍有一些方向可以进行进一步的优化和探索：

- 数据集扩展：可以考虑使用更大规模的数据集来训练模型，以提高模型的泛化能力和性能。

- 模型调整：可以尝试调整模型的结构或超参数，例如使用不同的预训练模型，或者调整损失函数和优化器，以进一步提高模型的准确率和鲁棒性。
- 多任务学习：文本相似度检测可以与其他NLP任务结合，例如文本分类、情感分析等，通过多任务学习来提高模型的综合性能。
- 解释性分析：尽管深度学习模型在许多任务上取得了优异的性能，但它们的决策过程往往是黑箱的。可以尝试引入模型解释性技术，如注意力机制可视化，来更好地理解模型是如何做出预测的。
- 实时应用：将模型部署到实际应用中，例如在线客服、智能问答系统等，以提供实时文本相似度检测服务。

通过这些展望，我们希望能够进一步改进文本相似度检测模型，并在更广泛的应用场景中发挥作用。