基于 MacBERT 的技术问答社区重复问题识别

摘 要

建立一个基于自然语言处理技术的自动标重系统会对技术问答平台的日常维护起到极大作用。本文探究的是如何使用 NLP 技术建立一个判断问题是否重复的分类模型。数据处理和模型构建使用 Python3.8，MacBERT 模型构建使用 PyTorch 深度学习框架，并使用 GPU 进行训练加速。

针对问题一（**中文文本分类问题**），本文采用基于 MacBERT 的文本相似度预测模型，将问题一的文本相似度概率预测问题抽象为是否为重复问题的二分类问题，并在实际数据进行预测。首先使用 CPM 中文分词模型（CPM-tokenizer）对问题文本进行分词处理并嵌入向量化，然后将需要预测的两个问题的嵌入向量输入到 MacBERT 中，得到了相似性特征向量，然后将特征向量输入到全连接层中，得到最后的分类结果。由于原始数据存在正负样本数量极不平衡问题，我们将所有的正样本的 70%分为训练集，并将相同数量的负样本也分入训练集，将剩下的全部正样本和负样本都作为测试集。**最终结果为 F1-score 为 72.18%，准确度为 91.01%，Loss 为 0.00004。**

针对问题二（**推荐问题**），将问题一中的深度神经网络全连接层进行改进，使其输出为重复问题的概率，针对每一个问题在所有的问题数据中随机采样 100 个问题作为候选对象，并依据概率的大小进行排序，得到推荐列表，以完成问题二的相似问题推荐任务。**最终结果为 R 值为 0.2460。**

关键词：重复问题识别；MacBERT 模型；文本相似性；self-attention；Transformer

目录

1.问题重述

1.1 问题的背景

技术社区问答平台作为用户互相分享交流的社区平台，近年来逐步成为用户寻找技术类疑难解答的首要渠道。各分类技术性问题的文本数据量不断攀升，给问答平台的日常运营维护带来了挑战。随着新用户的不断加入以及用户数量的增加，新用户提出的疑问可能已经在平台上被其他用户提出并解答过，但由于技术性问题的复杂性，各个用户提问的切入角度不同，用问题标题关键词匹配的搜索系统无法指引新用户至现有的问题。于是，新用户会提出重复的问题，而这些问题会进一步增加平台上的文本量，导致用户重复响应相同的问题。对于这种现象，通常的做法是及时找到新增的重复问题并打上标签，然后在搜索结果中隐藏该类重复问题，保证对应已解决问题出现的优先度。所以，建立一个基于自然语言处理技术的自动标重系统会对问答平台的日常维护起到极大帮助。

目前，问答平台上的问题标重主要依靠用户人工辨别。平台用户会对疑似重复的问题进行投票标记，然后平台内的管理员和资深用户（平台等级高的用户）对该问题是否被重复提问进行核实，若确认重复则打上重复标签。该过程较为繁琐，依赖用户主观判断，存在时间跨度大、工作量大、效率低等问题，增加了用户的工作量且延长了新用户寻求答案所需的时间。因而，如能建立一个检测问题重复度的模型，通过配对新提出问题与文本库中现存问题，找出重复的问题组合，就能提高重复问题标记效率，提高平台问题的文本质量，减少问题冗余。同时，平台用户也能及时地根据重复标签提示找到相关问题并查看已有的回复。

1.2 问题的提出

根据题目中给出的描述，需要在论文中，建立一个判断问题是否重复的分类模型并解决以下问题：

1）输出样本问题组为重复问题的概率：

2）从附件问题列表中，给出与目标问题重复概率最大的前 10 个问题的编号。

2 问题分析

2.1 问题一分析

对于问题一，属于**文本分类问题**。

传统的机器学习分类方法将整个文本分类问题就拆分成了特征工程和分类器两部分。特征工程分为文本预处理、特征提取、文本表示三个部分，最终目的是把文本转换成计算机可理解的格式，并封装足够用于分类的信息，即很强的特征表达能力。传统做法主要问题的文本表示是高纬度高稀疏的，特征表达能力很弱，而且神经网络很不擅长对此类数据的处理；此外需要人工进行特征工程，成本很高。

应用深度学习解决大规模文本分类问题最重要的是解决文本表示，再利用CNN/RNN/BERT 等网络结构自动获取特征表达能力，去掉繁杂的人工特征工程，端到端的解决问题。本文首先采用 CPM 中文分词模型（CPM-tokenizer）对问题文本进行分词处理并嵌入向量化，然后将需要预测的两个问题的嵌入向量输入到 MacBERT 中，得到了相似性特征向量，然后将特征向量输入到全连接层中，得到最后的分类结果。

2.2 问题二分析

对于问题二，属于**推荐问题**。

推荐系统是根据用户需求、兴趣等，通过推荐算法从海量数据中挖掘出用户感兴趣的项目（如信息、服务、物品等），并将结果以个性化列表的形式推荐给用户。推荐系统的核心是推荐算法，它利用用户与项目之间的二元关系，基于用户历史行为记录或相似性关系帮助发现用户可能感兴趣的项目。

传统的推荐算法主要分为以下四种方式：关联规则推荐、协同过滤推荐（collaborative filtering recommendation）、基于内容的推荐（content-based recommendation）、混合推荐(hybrid recommendation)。经典的协同过滤方法采用浅层模型无法学习到用户和项目的深层次特征；基于内容的推荐方法利用用户已选择的项目来寻找其它类似属性的项目进行推荐，但是这种方法需要有效的特征提取，传统的浅层模型依赖于人工设计特征（先验知识），其有效性及可扩展性非常有限，制约了基于内容的推荐方法的性能。

因此本文采用的是**基于深度学习的推荐系统**。基于深度学习的推荐系统通常将各类用户和项目相关的数据作为输入，利用深度学习模型学习到用户和项目的隐表示，并基于这种隐表示为用户产生项目推荐。一个基本的架构如图 1 所示，包含输入层、模型层和输出层．输入层的数据主要包括：根据 CPM 生成的嵌入向量。在模型层，使用的深度学习模型比较广泛，包括自编码器、受限玻尔兹曼机、卷积神经网络、循环神经网络等，本文才好用的是 BERT 模型。在输出层，通过利用学习到的句式隐表示，通过内积、Softmax、相似度计算等方法产生项目的推荐列表。

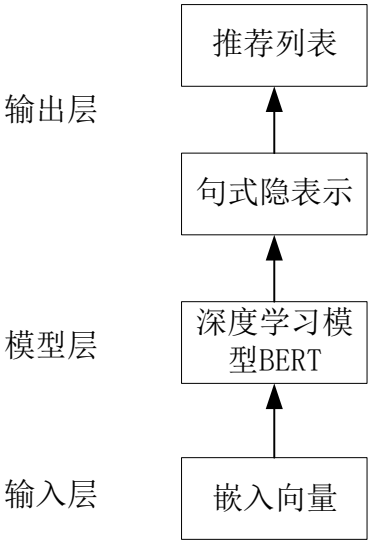


图 1 推荐系统架构

针对文本，输出相似性大的文本，针对一个问题，对所有的 ID 做分类问题，选择 100 个做分类问题。

3.模型构建

3.1 模型整体架构

本文使用了 CPM 中文分词模型和 MacBERT 模型，首先用 CPM 中文分词模型对问题文本进行分词处理，而后用一个 8 层 Transformer Encoder 的 MacBERT 模型进行训练和测试，得到分类结果。图 2 中左图为我们构建的模型，右图为 Transformer Encoder 的内部结构。模型的整体架构如图 2：

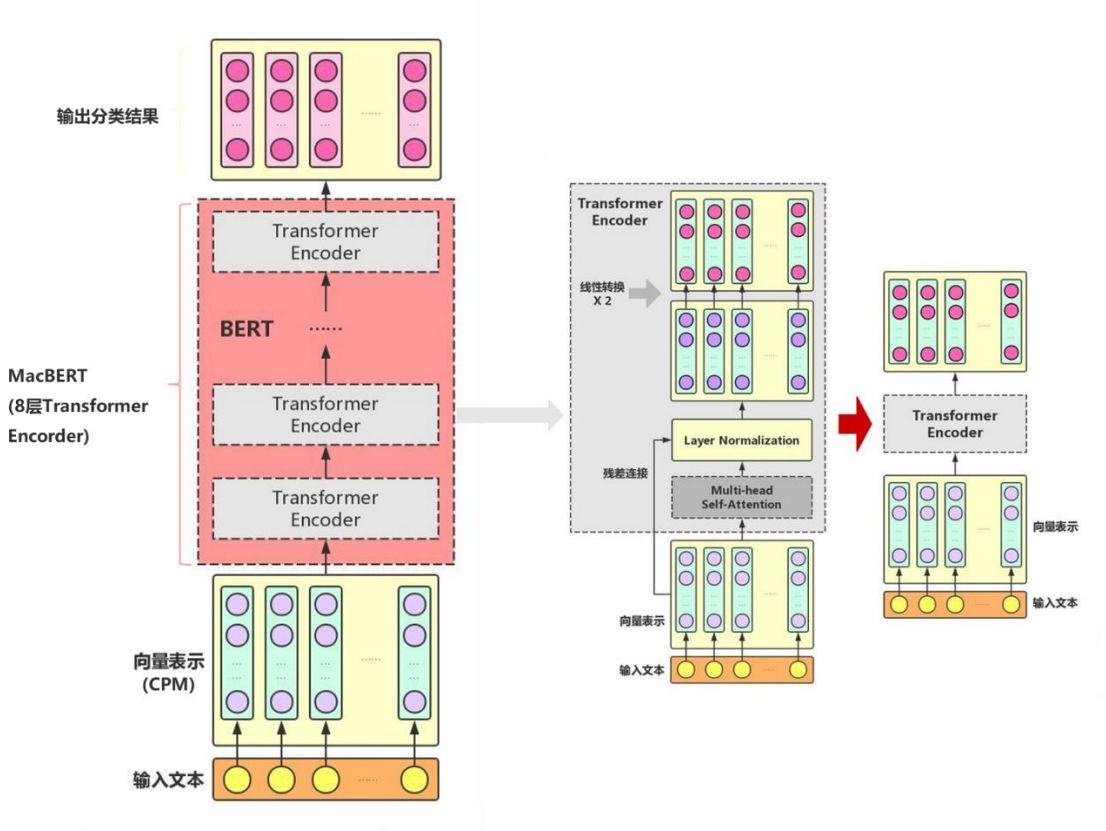


图 2 模型的整体架构

首先使用 CPM 中文分词模型（CPM-tokenizer）对问题文本进行分词处理并嵌入向量化，然后将需要预测的两个问题的嵌入向量输入到 MacBERT 中，得到了相似性特征向量，然后将特征向量输入到全连接层中，得到最后的分类结果。数据流程图如图 3。

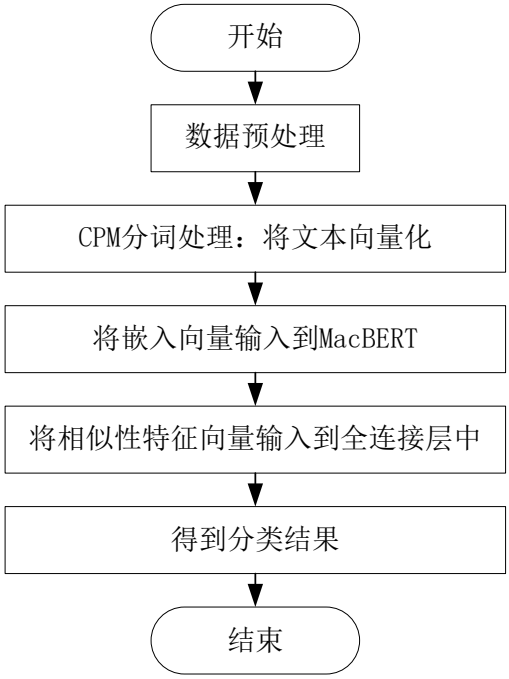


图 3 数据流程图

3.2 CPM

Chinese Pre-trained Language Model（CPM）是最大的中文预训练语言模型，它可以促进后续的中文自然语言处理任务，如会话、文章生成、完形填空和语言理解。通过对各种中文自然语言处理任务的实验表明，CPM 算法在少数镜头（甚至是零镜头）的情况下对许多自然语言处理任务都有很好的处理效果。随着参数的增加，CPM 在大多数数据集上表现得更好，这表明较大的模型在语言生成和语言理解方面更为熟练。

CPM 是一个从左到右的转换器解码器，类似于 GPT 的模型架构（Radford 等人，2019）。我们预先训练了三个不同尺寸的模型。为了使 CPM 适应汉语语料库，我们建立了一个新的子词表，并调整了训练批量。表 1 是模型尺寸。

	n_{param}	n_{layers}	d_{models}	n_{heads}	d_{head}
CPM-small	109M	12	768	12	64
CPM-Medium	334M	24	1024	16	64
CPM-Large	2.6B	32	2560	32	80

表 1：模型尺寸。n_{param} 是参数的数量。n_{layers} 增加层数。d_{models} 是隐藏状态的维度，在每一层中都是一致的。n_{heads} 是每层中的关注头数量。d_{head} 是每个注意头的尺寸。

词汇构建：以往关于汉语预训练模型的研究通常采用 BERT-Chinese 的子词词汇（Devlin et al.，2019），将输入文本分割成一个新的文本字符级序列。然而，汉语词汇通常包含多个字符，在字符级序列中会丢失一些重要的语义。为了解决这个问题，我们构造了一个新的子词词汇，包含单词和字符。例如，一些常用词会被添加到词汇表中。

训练策略：由于中文词分布的稀疏性比英文词分布的稀疏性更严重，我们采用了大批量的方法使模型训练更加稳定。与 GPT-3 2.7B（Brown 等人，2020）中使用的批量大小（100 万代币）相比，我们的批量大小（300 万代币）要大两倍。

对于训练过程中不能存储在单个 GPU 中的最大模型，我们将模型沿宽度维度跨 GPU 进行划分，以便于大规模训练，减少节点间的数据传输。

3.3 Transformer

Transformer 的结构和 Attention 模型一样，Transformer 模型中也采用了 encoer-decoder 架构。但其结构相比于 Attention 更加复杂，论文中 encoder 层由 6 个 encoder 堆叠在一起，decoder 层也一样。图 4 是 Transformer 结构图。

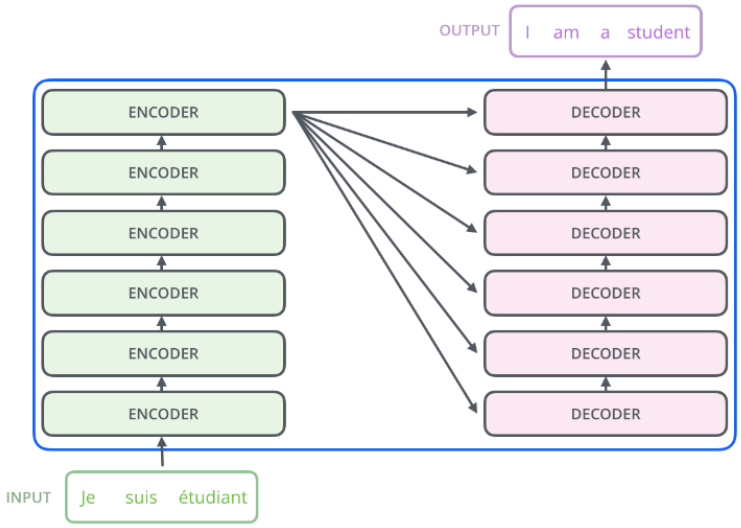


图 4 Transformer 结构

绝大部分的序列处理模型都采用 encoder-decoder 结构，其中 encoder 将输入序列 $(x_1, x_2, ..., x_n)$ 映射到连续表示 $\vec{z} = (z_1, z_2, ..., z_n)$，然后 decoder 生成一个输出序列 $(y_1, y_2, ..., y_m)$，每个时刻输出一个结果。Transformer 模型延续了这个模型，每一个 encoder 和 decoder 的内部结构如下图 5：

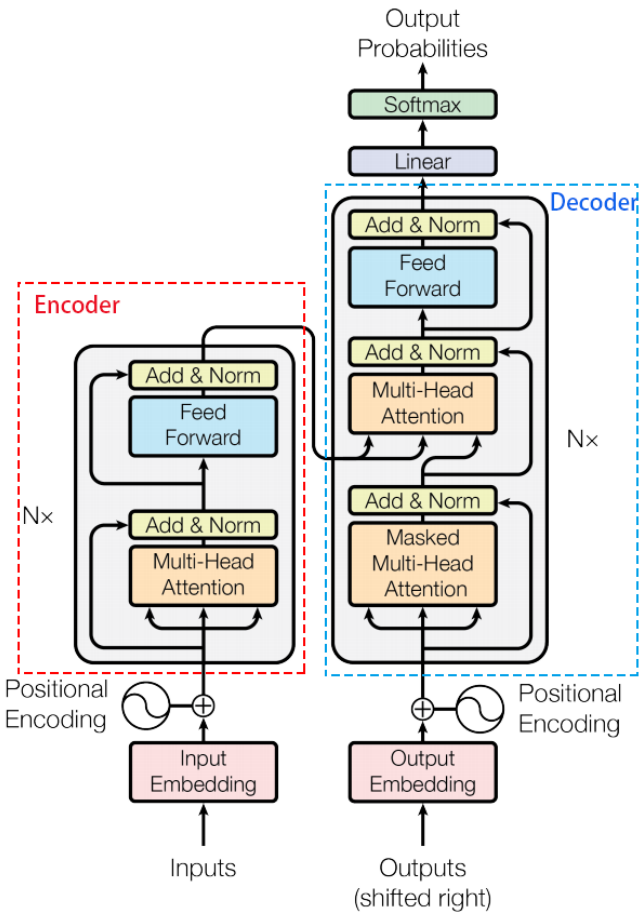


图 5 Encoder-Decoder 架构

encoder，包含两层，一个 self-attention 层和一个前馈神经网络，self-attention 能帮助当前节点不仅仅只关注当前的词，从而能获取到上下文的语义。

decoder 也包含 encoder 提到的两层网络，但是在这两层中间还有一层 attention 层，帮助当前节点获取到当前需要关注的重点内容。

3.3.1 Encoder 层结构

首先，模型需要对输入的数据进行一个 embedding 操作，也可以理解为类似 w2c 的操作，enmbedding 结束之后，输入到 encoder 层，self-attention 处理完数据后把数据送给前馈神经网络，前馈神经网络的计算可以并行，得到的输出会输入到下一个 encoder。

3.3.2 Decoder 层结构

Decoder 也是 N=6 层，每层包括 3 个 sub-layers：

第一个是 Masked multi-head self-attention，也是计算输入的 self-attention，但是因为是生成过程，因此在时刻 i 的时候，大于 i 的时刻都没有结果，只有小于 i 的时刻有结果，因此需要做 Mask

第二个 sub-layer 是全连接网络，与 Encoder 相同

第三个 sub-layer 是对 encoder 的输入进行 attention 计算。

同时 Decoder 中的 self-attention 层需要进行修改，因为只能获取到当前时刻之前的输入，因此只对时刻 t 之前的时刻输入进行 attention 计算，这也称为 Mask 操作。

3.3.3 self-attention 机制

self-attention 是 Transformer 用来将其他相关单词的“理解”转换成我们正在处理的单词的一种思路，其思想和 attention 类似。

Scaled Dot-Product Attention

在 Transformer 中使用的 Attention 是 Scaled Dot-Product Attention, 是归一化的点乘 Attention，假设输入的 query q、key 维度为 d_k ,value 维度为 d_v，那么就计算 query 和每个 key 的点乘操作，并除以 $\sqrt{d_k}$ ，然后应用 Softmax 函数计算权重。

$$Attention(Q,K_i,V_i)=Soft\max(\frac{Q^TK_i}{\sqrt{d_k}})V_i \tag{1}$$

在实践中，将 query 和 keys、values 分别处理为矩阵 Q,K,V，那么计算输出矩阵为：

$$Attention(Q,K,V)=Soft\max(\frac{Q^TK}{\sqrt{d_k}})V \tag{2}$$

其中 $Q\in\mathbb{R}^{m\times d_k}$ ，$K\in\mathbb{R}^{m\times d_k}$ ，$K\in\mathbb{R}^{m\times d_k}$，输出矩阵维度为 $\mathbb{R}^{m\times d_v}$。

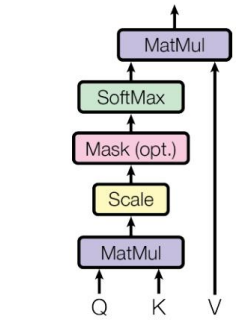
那么 Scaled Dot-Product Attention 的示意图如下所示，Mask 是可选的(opt.)，如果是能够获取到所有时刻的输入(K, V), 那么就不使用 Mask；如果是不能获取到，那么就需要使用 Mask。使用了 Mask 的 Transformer 模型也被称为 Transformer Decoder，不使用 Mask 的 Transformer 模型也被称为 Transformer Encoder。图 5 是 Scaled Dot-Product Attention 示意图。

Multi-Head Attention

如果只对 Q、K、V 做一次这样的权重操作是不够的，这里提出了 Multi-Head Attention，操作包括：

1) 首先对 Q、K、V 做一次线性映射，将输入维度均为 d_{model} 的 Q,K,V 矩阵映射到 $Q\in\mathbb{R}^{m\times d_k}$ ，$K\in\mathbb{R}^{m\times d_k}$ ，$K\in\mathbb{R}^{m\times d_k}$
2) 然后在采用 Scaled Dot-Product Attention 计算出结果
3) 多次进行上述两步操作，然后将得到的结果进行合并
4) 将合并的结果进行线性变换

图 6 是 Multi-Head Attention 机制示意图：



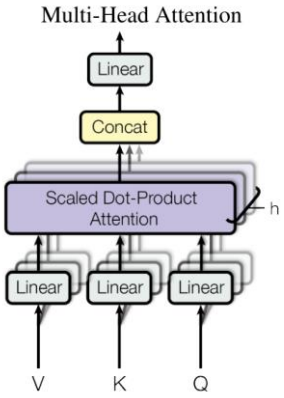


图 5 Scaled Dot-Product Attention 示意　　图 6 Multi-Head Attention 示意图

在图 4 的 Encoder-Decoder 架构中，有三处 Multi-head Attention 模块，分别是：

1) Encoder 模块的 Self-Attention，在 Encoder 中，每层的 Self-Attention 的输入 $Q=K=V$ ，都是上一层的输出。Encoder 中的每个 position 都能够获取到前一层的所有位置的输出。
2) Decoder 模块的 Mask Self-Attention，在 Decoder 中，每个 position 只能获取到之前 position 的信息，因此需要做 mask，将其设置为 $-\infty$
3) Encoder-Decoder 之间的 Attention，其中 Q 来自于之前的 Decoder 层输出，K,V 来自于 encoder 的输出，这样 decoder 的每个位置都能够获取到输入序列的所有位置信息。

3.3.4 Position-wise Feed-forward Networks

在进行了 Attention 操作之后，encoder 和 decoder 中的每一层都包含了一个全连接前向网络，对每个 position 的向量分别进行相同的操作，包括两个线性变换和一个 ReLU 激活输出

$$FFN(x)=\max(0,xW_1+b_1)W_2+b_2 \tag{3}$$

其中每一层的参数都不同。

3.3.5 Positional Encoding

Transformer 模型中缺少一种解释输入序列中单词顺序的方法，它跟序列模型还不不一样。为了处理这个问题，transformer 给 encoder 层和 decoder 层的输入添加了一个额外的向量 Positional Encoding，维度和 embedding 的维度一样，这个向量采用了一种很独特的方法来让模型学习到这个值，这个向量能决定当前词的位置，或者说在一个句子中不同的词之间的距离。这个位置向量的具体计算方法有很多种，本文的计算方法如下：

$$PE(pos,2i)=\sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{4}$$

$$PE(pos,2i+1)=\cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{5}$$

其中 pos 是指当前词在句子中的位置，i 是指向量中每个值的 index，可以看出，在偶数位置，使用正弦编码，在奇数位置，使用余弦编码。

最后把这个 Positional Encoding 与 embedding 的值相加，作为输入送到下一层。

3.4 BERT

Bidirectional Encoder Representations from Transformers（BERT）是一种预训练语言表示的方法，这意味着我们在大型文本语料库（如维基百科）上训练通用的"语言理解"模型，然后将该模型用于我们关心的下游 NLP 任务（如问题解答）。BERT 优于以前的方法，因为它是第一个无人监督的、深度双向的培训前 NLP 系统。无人监督意味着 BERT 仅使用普通文本语料库接受训练，这一点很重要，因为网络上以多种语言公开了大量纯文本数据。

预先训练的陈述也可以是无上下文的，也可以是上下文的，上下文陈述可以进一步是单向的或双向的。无上下文模型（如 word2vec 或 GloVe）为词汇中的每个单词生成一个单一的"单词嵌入"表示。

BERT 是建立在训练前语境陈述（包括半监督序列学习、生成预训练、ELMo 和 ULMFit）的最新工作基础上的，但关键是这些模型都是单向的或浅双向的。这意味着每个单词仅使用左侧（或右侧）的单词上下文。

BERT 使用一个简单的方法：我们屏蔽输入中 15% 的单词，通过深双向 transformer 编码器运行整个序列，然后仅预测蒙面单词。

然后，我们将在大型语料库上训练一个大模型（12 层到 24 层 transformer），时间很长（1M 更新步骤），这就是 BERT。

使用 BERT 分为两个阶段：预处理和微调。在预训练过程中，通过不同的预训练任务对模型进行未标记数据的训练。对于微调，首先使用预先训练好的参数初始化 BERT 模型，然后使用下游任务的标记数据对所有参数进行微调。每个下游任务都有单独的微调模型，即使它们是用相同的预训练参数初始化的。图 7 中是一个问答示例。

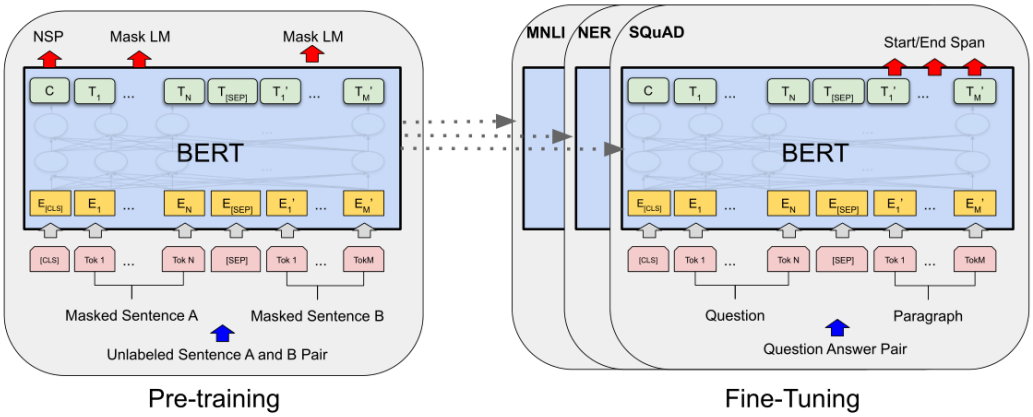


图 7：BERT 的整体预训练和微调程序。除了输出层之外，在预训练和微调中都使用了相同的体系结构。相同的预训练模型参数用于为不同的下游任务初始化模型。在微调期间，所有参数都会被微调。[CLS]是添加在每个输入示例前面的特殊符号，[SEP]是特殊的分隔符标记（例如分隔问题/答案）。

BERT 的一个显著特征是它跨不同任务的统一体系结构。预先训练的体系结构和最终的下游体系结构之间的差别最小。

对于给定的 Token，其输入表示是通过对相应的 Token、段和位置嵌入求和来构造的。这种结构的可视化如图 8 所示。

其中：

1) Token Embeddings 是词向量，第一个单词是 CLS 标志，可以用于之后的分类任务
2) Segment Embeddings 用来区别两种句子，因为预训练不光做 LM 还要做以两个句子为输入的分类任务
3) Position Embeddings 和之前文章中的 Transformer 不一样，不是三角函数而是学习出来的。

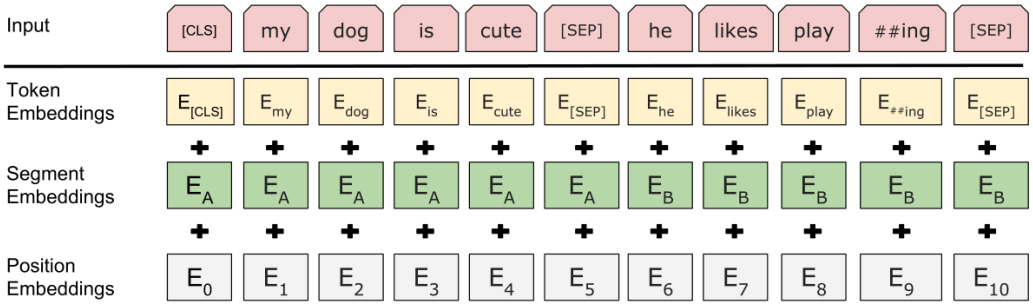


图 8:BERT 输入表示。输入嵌入是 Token 嵌入、分段嵌入和位置嵌入的总和。

3.4.1 预训练

第一步预训练的目标就是做语言模型，从上文模型结构中看到了这个模型的不同，即 bidirectional。这里作者用了一个加 mask 的 trick。

在训练过程中作者随机 mask 15%的 token，而不是把像 cbow 一样把每个词都预测一遍。最终的损失函数只计算被 mask 掉那个 token。

Mask 如何做也是有技巧的，如果一直用标记[MASK]代替（在实际预测时是碰不到这个标记的）会影响模型，所以随机 mask 的时候 10%的单词会被替代成其他单词，10%的单词不替换，剩下 80%才被替换为[MASK]。要注意的是 Masked LM 预训练阶段模型是不知道真正被 mask 的是哪个词，所以模型每个词都要关注。

因为序列长度太大（512）会影响训练速度，所以 90%的 steps 都用 seq_len=128 训练，余下的 10%步数训练 512 长度的输入。

第二个预训练任务，目的是让模型理解两个句子之间的联系。训练的输入是句子 A 和 B，B 有一半的几率是 A 的下一句，输入这两个句子，模型预测 B 是不是 A 的下一句。预训练的时候可以达到 97-98%的准确度。

3.4.2 微调

微调是很简单的，因为转换器中的自我关注机制允许 BERT 对许多下游任务进行建模，不管它们是涉及单个文本还是通过交换适当的输入和输出来实现文本对。对于涉及文本对的应用，一种常见的模式是在应用双向交叉注意之前对文本对进行独立编码，如 Parikh et al.（2016）；Seo 等人（2017 年）。BERT 使用了自我注意机制来统一这两个阶段，因为用自我注意编码串联的文本对有效地包含了两句话之间的双向交叉注意。

对于每个任务，我们只需将特定于任务的输入和输出插入到 BERT 中，并端到端地微调所有参数。在输入时，预训练的句子 A 和句子 B 类似于（1）复述中

的句子对，（2）蕴涵中的假设前提对，（3）问答中的问题-段落对，以及（4）退化的文本-∅在文本分类或序列标记中配对。在输出处，表示被馈送到 Token 级任务的输出层，例如序列标记或问答，并且[CLS]表示被馈送到分类的输出层，例如蕴涵或情感分析。与预训练相比，微调相对便宜。

3.5 MacBERT

在本文中，我们利用了以前的 BERT 模型，并提出了一个简单的修改，导致微调任务的显著改进，我们称这个模型为 MacBERT（MLM 为修正 BERT）。MacBERT 与 BERT 共享相同的预训练任务，但有一些修改。对于 MLM 任务，我们执行以下修改。

1) 我们使用全词掩蔽和 Ngram 掩蔽策略来选择候选标记进行掩蔽，单词级单图到 4-gram 的百分比分别为 40%、30%、20%、10%。
2) 我们建议使用类似的词语进行掩蔽，而不是使用[MASK]标记进行掩蔽，该标记从未出现在微调阶段。使用同义词工具箱（Wang 和 Hu，2017）获得相似的单词，该工具箱基于 word2vec（Mikolov et al.，2013）相似性计算。如果选择一个 N-gram 来屏蔽，我们将分别找到相似的单词。在极少数情况下，当没有相似词时，我们会降级使用随机词替换。
3) 我们使用 15%的输入词进行掩蔽，其中 80%将替换为相似词，10%替换为随机词，其余 10%保留为原始词。

对于类似 NSP 的任务，我们执行句子顺序 ALBERT（Lan 等人，2019）提出的预测（SOP）任务，通过切换两个连续句子的原始顺序来创建负样本。

我们调查了最近自然语言处理领域中有代表性的预训练语言模型的技术。表 2 描述了这些模型以及所提出的 MacBERT 模型的总体比较。

	BERT	ERNIE	XLNet	RoBERTa	ALBERT	ELECTRA	MacBERT
Type	AE	AE	AR	AE	AE	AE	AE
Embeddings	T/S/P	T/S/P	T/S/P	T/S/P	T/S/P	T/S/P	T/S/P
Masking	T	T/E/Ph	-	T	T	T	WWM/NM
LM Task	MLM	MLM	PLM	MLM	MLM	Gen-Dis	Mac
Paired Task	NSP	NSP	-	-	SOP	-	SOP

表 2：预先训练的语言模型的比较。（AE：自动编码器，AR：自回归，T：令牌，S：段，P：位置，W：词，E：实体，Ph：短语，WWM：全词掩蔽，NM： N-gram 掩蔽，NSP：下一句预测，SOP：句序预测，MLM：掩蔽 LM，PLM：置换 LM，Mac：MLM 作为校正）

4.模型训练

4.1 数据预处理

针对附件 2 中正样本数量 1005 个，负样本数量 6222 个，因此正负样本数量存在极不平衡问题，并且负样本存在数据缺失问题—只有 questionID 没有 duplicates，我们在所有的 ID 中随机生成一个作为 duplicates 对象。

4.2 硬件和时间

我们用 1 个英伟达 TAITAN RTX 图形处理器在一台机器上训练我们的模型。对于使用本文中描述的超参数的基本模型，每个训练步骤大约需要 1.06 秒。我们对基础模型进行了总共 6100 步或 3.1 小时的训练。

4.3 优化器

我们使用了 AdamW 优化器[3]，其中 $\beta_1 = 0.9$，$\beta_2 = 0.999$，$\varepsilon = 10^{-8}$.我们在训练过程中根据以下公式改变了学习率:

$$lrate = d_{\mathrm{model}}^{-0.5} . \min(step\_num^{-0.5}, step\_num.warmup\_step^{-1.5}) \quad (6)$$

这对应于对于第一个预热步骤训练步骤线性地增加学习速率，并且随后与步骤数的平方根倒数成比例地减少学习速率。我们用了 $warmup\_step = 4000$。

4.4 正则化

在训练过程中，我们采用了三种类型的正则化，分别为：Residual Dropout、LayerNorm 和 L2 正则化。

4.4.1 Residual Dropout

我们将剔除[4]应用于每个子层的输出，然后将其添加到子层输入并进行归一化。此外，我们对编码器和解码器堆栈中的嵌入和位置编码的总和应用了丢弃。对于基本模型，我们使用 $P_{drop} = 0.1$ 的比率。图 9 为 dropout 示意图。

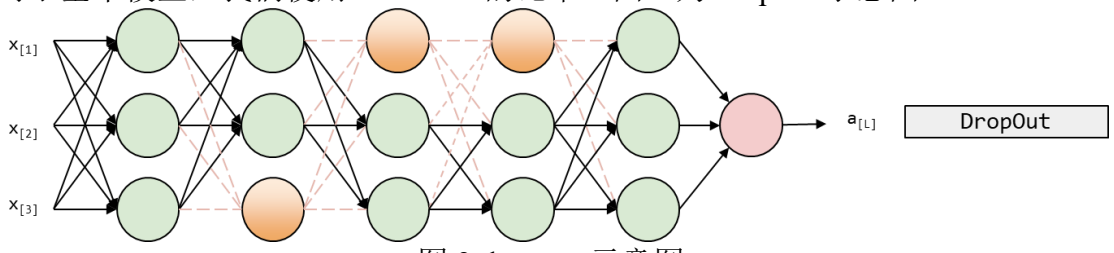


图 9 dropout 示意图

4.4.2 LayerNorm

LayerNorm[5]过程中，令 $x = (x_1, x_2, ..., x_H)$ 是归一化层的大小为 H 的输入的矢量表示。LayerNorm 重新心并重新缩放输入 x 为：

$$h = g \odot N(x) + b, \quad N(x)\frac{x-\mu}{\delta}, \quad \mu = \frac{1}{H}\sum_{i=1}^{H} x_i, \quad \delta = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(x_i-\mu)^2} \tag{7}$$

其中 h 是 LayerNorm 层的输出。$\odot$ 是一个点生产操作。μ 和 δ 为均值和输入的标准偏差。偏差 b 和增益 g 是具有相同尺寸 H 的参数。图 10 为 Normalization 的示意图。

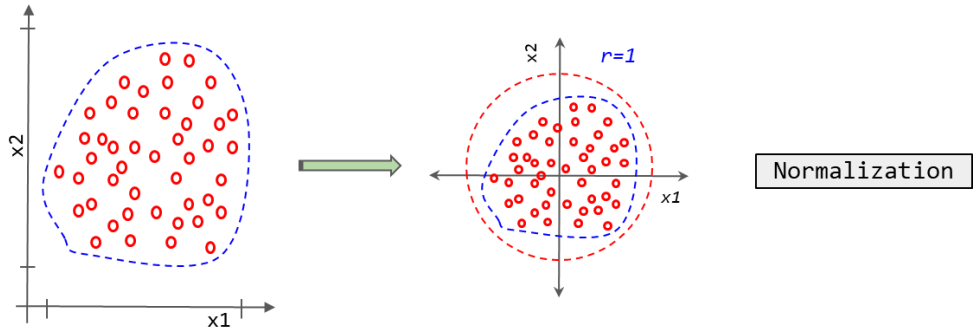


图 10 Normalization 的示意图

4.4.3 L2 正则化

L2 正则化[6]的形式是在原先的损失函数后边再加多一项：$\frac{1}{2}\lambda\theta_i^2$，那加上 L2 正则项的损失函数就可以表示为：$L(\theta) = L(\theta) + \lambda\sum_{i}^{n}\theta_i^2$，其中 θ 就是网络层的待学习的参数，λ 则控制正则项的大小，较大的取值将较大程度约束模型复杂度，反之亦然。

4.5 超参数设置

超参数设置如表 3：

表 3 超参数设置

超参数	值
学习率	1×10^{-4}
Epochs	100
Batchsize	24

4.6 训练策略

4.6.1 学习率衰减

本文采用的是按步长衰减，公式如下：

$$lr = lr_{base} \cdot \tau^{\left\lfloor \frac{step}{stepsize} \right\rfloor} \tag{8}$$

其中 lr_{base} 是基础学习率。τ 小于 1 衰减率。stepsize 是个阈值，当前迭代步伐 step 除以他，向下取整作为 τ 的指数。衰减系数和 stepsize 都是经验值。本文采用的是每 20 个 epoch 衰减 10%。

4.6.2 早停策略

早停法的基本含义是在训练中计算模型在验证集上的表现，当模型在验证集上的表现开始下降的时候，停止训练，这样就能避免继续训练导致过拟合的问题。其主要步骤如下：

1) 将原始的训练数据集划分成训练集和验证集
2) 只在训练集上进行训练，并每个一个周期计算模型在验证集上的误差，例如，每 15 次 epoch（mini batch 训练中的一个周期）
3) 当模型在验证集上的误差比上一次训练结果差的时候停止训练
4) 使用上一次迭代结果中的参数作为模型的最终参数

在我们的模型中，每 10 个 epoch 保存一次网络权重，根据实际的性能表现选择最佳网络权重，根据实际训练情况，epoch 在 40-55 之间时 F1-score 的值最大且稳定，因此将 epoch 大于 50 的部分舍去，只保留 epoch 等于 50 时的网络权重，并将此时的权重用于测试集性能评估。

4.7 微调

为了进行公平的比较，对于每个数据集，我们保持相同的超参数（如最大长度、预热步骤等），并且只将每个任务的初始学习速率从 1e-5 调整为 5e-5。请注意，初始学习速率是在原始的中文 BERT 上调整的，并且可以通过单独调整学习速率来获得另一个增益。为了保证实验结果的可靠性，我们做了十次同样的实验。最佳初始学习率是通过选择最佳的平均开发集性能来确定的。我们报告了最高和平均分数来评估最高和平均绩效。

表 4 为 CMRC 2018（简体中文）结果。括号中显示了 10 次独立运行的平均分数。整体最佳表现用黑体表示（标记基本水平）。预训练权重方式：Mac Bert Chinese base。

表 4 CMRC 2018（简体中文）结果

CMRC2018	DEV		Test		Challenge	
	EM	F1	EM	F1	EM	F1
BERT	65.5	84.5	70.0	87.0	18.6	43.3
	（64.4）	（84.0）	（68.7）	（86.3）	（17.0）	（41.3）
BERT-wwm	66.3	85.6	70.5	87.4	21.0	47.0
	（65.0）	（84.7）	（69.1）	（86.7）	（19.3）	（43.9）
BERT-wwm-ext	67.1	85.7	71.4	87.7	24.0	47.3
	（65.6）	（85.0）	（70.0）	（87.0）	（20.0）	（44.6）
RoBERTa-wwm-ext	67.4	87.	72.6	89.4	26.2	51.0
	（66.5）	（86.5）	（71.4）	（88.8）	（24.6）	（49.1）
ELECTRA-base	68.4	84.8	73.1	87.1	22.6	45.0
	（**68.0**）	（84.6）	（72.7）	（86.9）	（21.7）	（43.8）
MacBERT-base	**68.5**	**87.9**	**73.2**	**89.5**	**30.2**	**54.0**
	（67.3）	**（87.1）**	**（72.4）**	**（89.2）**	**（26.4）**	**（52.2）**

5.模型评价

5.1 问题一评价

5.1.1 F1-score

使用 F1-score 对分类模型进行评价：

$$F_1 = \frac{1}{n}\sum_{i=1}^{n}\frac{2P_iR_i}{P_i+R_i} \tag{9}$$

其中 P_i 为第 i 类的查准率， R_i 为第 i 类的查全率。

TP：被检索到正样本，实际也是正样本（正确识别）

FP：被检索到正样本，实际是负样本（一类错误识别）

FN：未被检索到正样本，实际是正样本。（二类错误识别）

TN：未被检索到正样本，实际也是负样本。（正确识别）

查准率 Precision：被正确检索的样本数与被检索到样本总数之比。即：TP/（TP+FP）。

查全率 Recall：被正确检索的样本数与应当被检索到的样本数之比。即：TP/（TP+FN）。

5.1.2 Accuracy

准确率（Accuracy）是最常用的指标，可以总体上衡量一个预测的性能。其表达为分类正确的样本数与样本总数之比。即：（TP+TN）/（ALL）。

5.1.3 Loos function

我们采用了交叉熵损失函数 （Cross-entropy loss function），交叉熵损失函数的标准形式如下:

$$C = -\frac{1}{n}\sum_x [y\ln a + (1-y)\ln(1-a)] \tag{10}$$

公式中 x 表示样本， y 表示实际的标签， a 表示预测的输出， n 表示样本总数量。

其特点如下：（1）本质上也是一种对数似然函数，可用于二分类和多分类任务中。（2）当使用 sigmoid 作为激活函数的时候，常用交叉熵损失函数而不用均方误差损失函数，因为它可以完美解决平方损失函数权重更新过慢的问题，具有“误差大的时候，权重更新快；误差小的时候，权重更新慢”的良好性质。

5.2 问题二评价

对于每个问题的预测结果采用 top K 列表对其进行评估，评估公式如下：

$$R = \frac{N_{\det ected}}{N_{total}} \tag{11}$$

其中 $N_{\det ected}$ 为在 top K 列表结果中正确检测到的重复问题编号数量， N_{total} 为该样本实际拥有的重复问题数量。评估时 K 取 10，若样本中无重复问题则不会计分。

6.结果

6.1 问题一结果

F1-score

F1-score 随 epoch 变化曲线如图 11 所示，最终 F1-score 稳定在 **72.18%**。当 epoch 在 40-55 之间时 F1-score 的值最大且稳定，因此将 epoch 大于 50 的部分舍去。

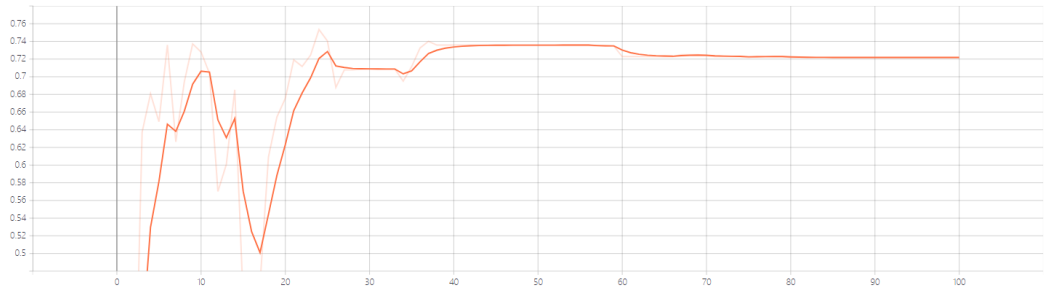


图 11 F1-score 随 epoch 变化曲线

Accuracy

Accuracy 随 epoch 变化曲线如图 12 所示，最终 Accuracy 稳定在 **91.01%**。

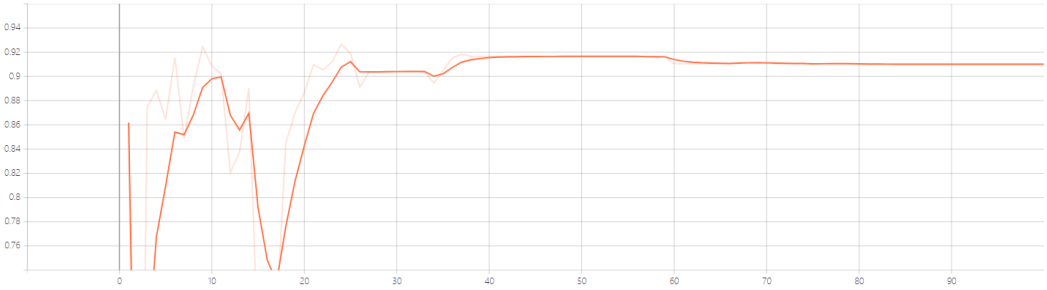


图 12 Accuracy 随 epoch 变化曲线

Loss

Loss 随 epoch 变化曲线如图 13 所示，最终 Loss 稳定在 **0.00004**。

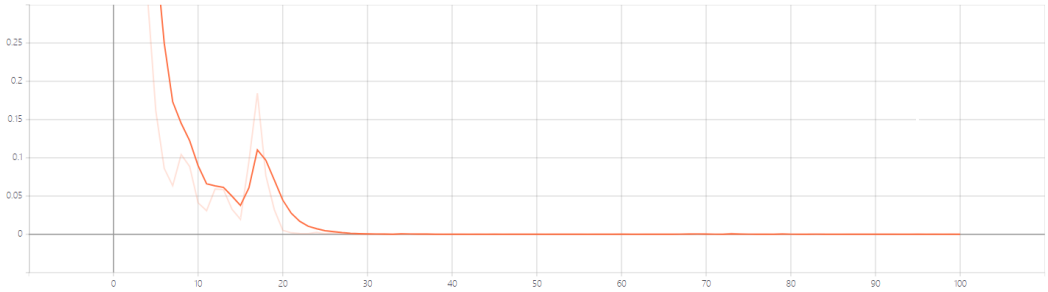


图 13 Loss 随 epoch 变化曲线

6.2 问题二结果

问题二采用的是问题一训练出的模型，其 R 值为 0.2460。

7.结论

本文采用基于 MacBERT 的文本相似度预测模型，该模型将 MLM 语言模型任务作为一种语言修正方式进行了修正，缓解了预训练和微调阶段的差异。针对问题一的最终结果为 F1-score 为 72.18%，准确度为 91.01%，Loss 为 0.00004。针对问题二的最终结果为 R=0.2460。本文使用了先进的算法 MacBERT 和 GPU 进行加速，效果显著。今后，我们可以研究一种有效的方法来确定掩蔽比，而不是启发式的掩蔽比，以进一步提高预训练语言模型的性能。

8.参考文献

[1]Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[J]. arXiv preprint arXiv:1706.03762, 2017.
[2] Cui Y, Che W, Liu T, et al. Revisiting pre-trained models for chinese natural language processing[J]. arXiv preprint arXiv:2004.13922, 2020.
[3] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In ICLR, 2015.
[4] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdi-nov. Dropout: a simple way to prevent neural networks from overfitting. Journal of MachineLearning Research, 15(1):1929–1958, 2014.
[5] Xu J, Sun X, Zhang Z, et al. Understanding and improving layer normalization[J]. arXiv preprint arXiv:1911.07013, 2019.
[6] Loshchilov I, Hutter F. Decoupled weight decay regularization[J]. arXiv preprint arXiv:1711.05101, 2017.
[7] Li Q, Peng H, Li J, et al. A Survey on Text Classification: From Shallow to Deep Learning[J]. arXiv preprint arXiv:2008.00364, 2020.
[8] Lu J, Wu D, Mao M, et al.Recommender system application developments[J]. Decision Support Systems, 2015,74(C):12-32.
[9] Paul Covington,Jay Adams,and EmreSargin .2016. Deep neural networks for youtube recommendations.In Proceedingsof the 10th ACM Conference on Recommender Systems.ACM,191–198.
[10] Cheng H T, Koc L, Harmsen J, etal. Wide & Deep Learning for Recommender Systems[J]. 2016:7-10.
[11] Zhang, Yongfeng, and Xu Chen. "Explainable recommendation: A survey and new perspectives." *arXiv preprint arXiv:1804.11192* (2018).
[12] Kim, Donghyun, et al. "Convolutional matrix factorization for document context-aware recommendation." *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 2016.

附录

附录一：preprocess

```
#该段代码功能：数据预处理
from numpy.core.defchararray import replace
# import pkuseg
from transformers import BertTokenizer, BertModel
import torch
from torch.utils.data import Dataset

import pandas as pd
import numpy as np


# 超参数
TEST_RATIO = 0.7
num_random = 10

# tokenizer = BertTokenizer.from_pretrained("weight")
# model =
BertModel.from_pretrained("weight/chinese_macbert_base.zip")

# 读生数据
# 将句子存入内存：所有句子有id且不连续，所以维持dataframe的存在
rawCsv = pd.read_csv('src/data/attachment1.csv')
# print(rawCsv.head())

# 建立text id和dataFrame id的索引，HashMap
idIndex = {}
for i in range(rawCsv.shape[0]):
    index = rawCsv.iloc[i, 0]
    idIndex[index] = i


# 读取标记数据
# 先验知识：
# 句子tokenize后词向量数量最大2038，最小11，直接设定max length = 2048，即词向量个数
# 有些行dpulicate中存在多个句子id，通过循环处理
# 三列的数据类型：np.int64, str, np.int64
# 第二列在eval后变成列表
```

```
rawPairDataCsv = pd.read_csv('src/data/attachment2.csv')
processedData = []
# print(rawPairDataCsv.head())
# print(type(rawPairDataCsv.iloc[0, 0]))
# print(type(eval(rawPairDataCsv.iloc[0, 1])))
# print(type(rawPairDataCsv.iloc[0, 2]))
for i in range(rawPairDataCsv.shape[0]):
    label = rawPairDataCsv.iloc[i, 2]
    if int(label) == 0:
        pairQuestionId = np.random.choice(list(idIndex.keys()),
size=1)
        processedData.append({'id1': rawPairDataCsv.iloc[i, 0],
'id2': pairQuestionId[0], 'label': label})
        continue

    questionId = rawPairDataCsv.iloc[i, 0]
    pairQuestionIdList = eval(rawPairDataCsv.iloc[i, 1])

    if not len(pairQuestionIdList) == 1:
        for pairID in pairQuestionIdList:
            pairID = int(pairID)

            processedData.append({'id1': questionId, 'id2': pairID,
'label': label})
    else:
        processedData.append({'id1': questionId, 'id2':
int(pairQuestionIdList[0]), 'label': label})


dataIndex = list(range(len(processedData)))

trainIndex = np.random.choice(dataIndex,
size=int(len(processedData) * TEST_RATIO), replace=False)
testIndex = list(set(dataIndex) - set(trainIndex))


# idx1 = processedData[0]['id1']
# idx2 = processedData[0]['id2']
# text1 = rawCsv.iloc[idIndex[idx1], 2]
# text2 = rawCsv.iloc[idIndex[idx2], 2]
# tensor_dict = tokenizer(text1, text2, padding=True,
```

```
max_length=2048, return_tensors='pt')


# 构建数据集
class TrainSet(Dataset):

    def __init__(self, trainIndex) -> None:
        super().__init__()

        self.textIndex = [processedData[i] for i in trainIndex]
        self.length = len(trainIndex)

    def __getitem__(self, ids):
        id1 = self.textIndex[ids]['id1']
        id2 = self.textIndex[ids]['id2']
        label = self.textIndex[ids]['label']

        text1 = rawCsv.iloc[idIndex[id1], 2]
        text2 = rawCsv.iloc[idIndex[id2], 2]
        return text1, text2, label

    def __len__(self):
        return self.length


class TestSet(Dataset):

    def __init__(self, testIndex) -> None:
        super().__init__()

        self.textIndex = [processedData[i] for i in testIndex]
        self.length = len(testIndex)

    def __getitem__(self, ids):
        id1 = self.textIndex[ids]['id1']
        id2 = self.textIndex[ids]['id2']
        label = self.textIndex[ids]['label']

        text1 = rawCsv.iloc[idIndex[id1], 2]
        text2 = rawCsv.iloc[idIndex[id2], 2]
        return text1, text2, label
```

```
    def __len__(self):
        return self.length


groundtruth_raw  = {}

for i in range(rawPairDataCsv.shape[0]):
    id1 = int(rawPairDataCsv.iloc[i, 0])
    label = rawPairDataCsv.iloc[i, 2]
    if int(label) == 0:
        continue
    id2list = eval(rawPairDataCsv.iloc[i, 1])
    if id1 not in groundtruth_raw.keys():
        groundtruth_raw[id1] = []
    for j in id2list:
        groundtruth_raw[id1].append(int(j))
        if int(j) not in groundtruth_raw.keys():
            groundtruth_raw[int(j)] = []
        groundtruth_raw[int(j)].append(id1)

# 根据attachment2.csv构建数据集
# for i in range(rawCsv.shape[0]):
#     print('\r', "{}/{}".format(i, rawCsv.shape[0]), end='')
#     index = rawCsv.iloc[i, 0]
#     # englishText = rawCsv.iloc[i, 1]
#     chineseText = rawCsv.iloc[i, 2]
#     # chineseTextCutted = seg.cut(chineseText)

#     # text2Tokenize = ['[CLS]'] + chineseTextCutted + ['[SEP]']
#     tokenizedText = tokenizer(chineseText, padding=True,
truncation=True, max_length=2048)

#     pass

# print('\n {}  {}'.format(np.array(length).min(),
np.array(length).max()))

# print(type(rawCsv.iloc[0, 2]))  输出为str
# print(type(rawCsv.iloc[0, 0]))  输出为numpy.int64
# print(rawCsv.shape)

list(idIndex.keys())
```

```
select_dataset = []
for i in range(rawCsv.shape[0]):
    id1 = rawCsv.iloc[i, 0]
    random_select_id = np.random.choice(list(idIndex.keys()),
size=num_random, replace=False)
    for id2 in random_select_id:
        select_dataset.append({'id1': id1, 'id2':id2})

    if id1 in groundtruth_raw.keys():
        for id2 in groundtruth_raw[id1]:
            select_dataset.append({'id1':id1, 'id2':id2})


class DatasetIndex:

    def __init__(self) -> None:
        self.trainIndex = trainIndex
        self.testIndex = testIndex
        self.select_dataset = select_dataset
        self.groundtruth = groundtruth_raw


class Random_Select_Dataset(Dataset):
    def __init__(self, select_dataset):
        self.data = select_dataset
        self.length = len(self.data)

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

    def __getitem__(self, ids):
        id1 = self.data[ids]['id1']
        id2 = self.data[ids]['id2']
        # label = self.data[ids]['label']

        text1 = rawCsv.iloc[idIndex[id1], 2]
        text2 = rawCsv.iloc[idIndex[id2], 2]
        return text1, text2, id1, id2
```

附录二：model

```
#该段代码功能：构建 MacBERT 模型
import torch
import torch.nn.functional as F
from torch import nn
from transformers import BertTokenizer, BertModel


class TextSimilarityModel(nn.Module):

    def __init__(self, pretrain_dir, device):
        super().__init__()
        self.pretrainedModel =
BertModel.from_pretrained(pretrain_dir)
        self.fc = nn.Linear(768 * 512, 2)

        self.device = device

    def __init__(self, token):
        input_ids = token['input_ids'].to(self.device)
        attention_mask = token['attention_mask'].to(self.device)
        token_type_ids = token['token_type_ids'].to(self.device)

        enchancedEmbeddings = self.pretrainedModel(input_ids,
attention_mask=attention_mask,
token_type_ids=token_type_ids)['last_hidden_state']
        enchancedEmbeddings = enchancedEmbeddings.faltten(1)
        classificationVector = self.fc(F.relu(enchancedEmbeddings))

        return classificationVector
```

附录三：evalution

```
# 该段代码功能：模型评估
import torch
from sklearn.metrics import f1_score

def evaluate1(model, test_loader, tokenizer, device):
    model.eval()

    print("evaluating ...")
```

```
    predicted = []
    ground_truth = []
    with torch.no_grad():
        for data in test_loader:
            texts1, texts2, labels = data
            labels = labels.to(device)
            encoding = tokenizer(texts1, texts2,
padding='max_length', truncation=True, max_length=512,
return_tensors='pt')
            output = model(encoding)

            _, logits = torch.max(output, 1)

            predicted = predicted + logits.numpy().tolist()
            ground_truth = ground_truth + labels.numpy().tolist()

    f1 = f1_score(ground_truth, predicted)
    print("f1 score: {:.5f}".format(f1))
```

附录四：train

```
# 该段代码功能：模型训练
from transformers import BertTokenizer, BertModel
import torch
import pandas as pd
import numpy as np
from torch import nn, optim
from torch.nn import functional as F
from torch.utils.data import DataLoader
from preprocess import TrainSet, TestSet, DatasetIndex
from evaluation import evaluate1


class TextSimilarityModel(nn.Module):

    def __init__(self, bertModel, device):
        super().__init__()
        self.pretrainedModel = bertModel
        self.fc = nn.Linear(768 * 512, 2)

        self.device = device

    def forward(self, token):
```

```
        input_ids = token['input_ids'].to(self.device)
        attention_mask = token['attention_mask'].to(self.device)
        token_type_ids = token['token_type_ids'].to(self.device)

        enchancedEmbeddings = self.pretrainedModel(input_ids,
attention_mask=attention_mask,
token_type_ids=token_type_ids)['last_hidden_state']
        enchancedEmbeddings = enchancedEmbeddings.flatten(1)
        classificationVector = self.fc(F.relu(enchancedEmbeddings))

        return classificationVector

if __name__ == "__main__":
    device = "cuda:0"
    # tokenizer 预训练模型初始化
    pretrainDir = "weight/chinese_macbert_base_torch"
    LR = 1e-4
    L2 = 1e-4
    EPOCHS = 50
    BATCHSIZE = 32

    tokenizer = BertTokenizer.from_pretrained(pretrainDir)
    bert = BertModel.from_pretrained(pretrainDir)

    # 添加全连层进行分类
    model = TextSimilarityModel(bert, device)
    model.to(device)

    # 数据集初始化
    datasetIndex = DatasetIndex()
    train_set = TrainSet(datasetIndex.trainIndex)
    test_set = TestSet(datasetIndex.testIndex)

    train_loader = DataLoader(train_set, batch_size=BATCHSIZE,
shuffle=True, num_workers=2)
    test_loader = DataLoader(test_set, batch_size=BATCHSIZE,
shuffle=True, num_workers=2)

    # 定义损失函数，优化器等
    loss_fn = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters(), lr=LR,
weight_decay=L2)
```

```
    batch_count = 0
    print("开始训练，训练集长度：{}".format(train_set.length))
    for epoch in range(EPOCHS):

        model.train()

        running_loss = []
        for data in train_loader:
            batch_count += 1

            optimizer.zero_grad()
            texts1, texts2, labels = data
            labels = labels.to(device)

            # position encoding 最大长度512
            encoding = tokenizer(texts1, texts2,
padding='max_length', truncation=True, max_length=512,
return_tensors='pt')

            output = model(encoding)
            loss = loss_fn(output, labels)
            loss.backward()
            optimizer.step()
            if batch_count % 10 == 0:
                print("Iter {} of epoch {}: loss:
{:.5f}".format(batch_count, epoch, loss.item()))

            running_loss.append(loss.item())

        print("Epoch {} loss: {:.5f}".format(epoch,
np.array(running_loss).mean()))
        evaluate1(model, test_loader, tokenizer, device)
        del running_loss

        # 保存权重
        torch.save(model, "model/model1.pth")

    # TODO: 完成F1 score计算
```

附录五：test

```
# 该段代码功能：模型测试
from torch.utils.data import dataloader
from transformers import BertTokenizer, BertModel
import torch
import pandas as pd
import numpy as np
from torch import nn, optim
from torch.nn import functional as F
from torch.utils.data import DataLoader
from preprocess import TrainSet, TestSet, DatasetIndex,
Random_Select_Dataset, dataIndex, groundtruth_raw
from evaluation import evaluate1


class TextSimilarityModel(nn.Module):

    def __init__(self, bertModel, device):
        super().__init__()
        self.pretrainedModel = bertModel
        self.fc = nn.Linear(768 * 512, 2)

        self.device = device

    def forward(self, token):
        input_ids = token['input_ids'].to(self.device)
        attention_mask = token['attention_mask'].to(self.device)
        token_type_ids = token['token_type_ids'].to(self.device)

        enchancedEmbeddings = self.pretrainedModel(input_ids,
attention_mask=attention_mask,
token_type_ids=token_type_ids)['last_hidden_state']
        enchancedEmbeddings = enchancedEmbeddings.flatten(1)
        classificationVector = self.fc(F.relu(enchancedEmbeddings))

        return classificationVector

if __name__ == "__main__":
    device = "cuda:0"
    # tokenizer 预训练模型初始化
    pretrainDir = "path/"
    LR = 1e-4
    L2 = 1e-4
```

```
    EPOCHS = 50
    BATCHSIZE = 32

    tokenizer = BertTokenizer.from_pretrained(pretrainDir)
    # bert = BertModel.from_pretrained(pretrainDir)

    # 添加全连层进行分类
    model = torch.load('model1_epoch40.pth')
    print(model)
    model.to(device)

    # 数据集初始化
    datasetIndex = DatasetIndex()
    select_data =
Random_Select_Dataset(datasetIndex.select_dataset)
    result = {}
    random_loader = DataLoader(select_data, batch_size=BATCHSIZE,
shuffle=True, num_workers=2)

    progress = 0
    with torch.no_grad():
        model.eval()
        for data in random_loader:
            progress += BATCHSIZE
            print('\r', '{}/{}'.format(progress, select_data.length),
end='')
            texts1, texts2, ids1, ids2 = data
            ids1 = ids1.numpy().tolist()
            ids2 = ids2.numpy().tolist()
            encoding = tokenizer(texts1, texts2,
padding='max_length', truncation=True, max_length=512,
return_tensors='pt')
            output = model(encoding)
            output = F.softmax(output, 1).cpu().numpy()[:,
1].tolist()
            for i in range(len(ids1)):
                id1 = ids1[i]
                id2 = ids2[i]
                if id1 not in result.keys():
                    result[id1] = []
                result[id1].append([id2, output[i]])
```

```
print()

HR = []
id_length = len(result)
cc = 0
for id1 in result.keys():
    cc += 1
    print('\r', '{}/{}'.format(cc, id_length), end='')
    id1result = np.array(result[id1])
    id2_candidate = np.array(list(map(int, id1result[:, 0])))
    score = id1result[:, 1]
    index = np.argsort(-score)
    id2_top10 = id2_candidate[index].tolist()[:10]

    if int(id1) not in datasetIndex.groundtruth.keys():
        R = 0
    else:
        R = 0
        ground_truth = datasetIndex.groundtruth[int(id1)]
        for gt in ground_truth:
            if gt not in id2_top10:
                continue
            else:
                position = id2_top10.index(gt)
                R += int(position < 10)
        R = R / len(ground_truth)

    HR.append(R)

print()
print(np.array(HR).mean())
```