手机屏幕的截图

中度可信度描述已自动生成

现代信息检索大作业

成员1：张 兆 202228013229029

成员2：陈国鑫 2022E8013282125

成员3：顾颂恩 202228015029012

成员4：周姿能 202228013229058

成员5：程 爽 202228013229057

解决方案及代码开源地址：<https://github.com/zhangzhao219/UCAS-IR-Project>

2022年12月

目录

[1 引言 1](#_Toc120436406)

[1.1 任务描述 1](#_Toc120436407)

[1.2 赛题分析 1](#_Toc120436408)

[1.3 评价指标 1](#_Toc120436409)

[2 实现方案 3](#_Toc120436410)

[2.1 技术原理 3](#_Toc120436411)

[2.1.1 Re-ranker 3](#_Toc120436412)

[2.1.2 BERT 3](#_Toc120436413)

[2.1.3 ColBERT 4](#_Toc120436414)

[2.1.4 Warmup 5](#_Toc120436415)

[2.2 预训练模型 6](#_Toc120436416)

[2.2.1 ELECTRA 7](#_Toc120436417)

[2.2.2 SimLM 7](#_Toc120436418)

[2.2.3 CoT-MAE 8](#_Toc120436419)

[2.3 实验效果对比 9](#_Toc120436420)

[3 实验步骤 11](#_Toc120436421)

[3.1 提交说明 11](#_Toc120436422)

[3.2 文件说明 11](#_Toc120436423)

[3.3 训练流程 13](#_Toc120436424)

[3.4 预测流程 15](#_Toc120436425)

[4 参考文献 17](#_Toc120436426)

1. 引言
   1. 任务描述

本次大作业要求使用采样后的TREC 2019训练数据，在TREC 2020 Passage Ranking的子赛道Passage Re-ranking进行检索竞赛。在Passage Re-ranking子任务中，每个查询提供1000篇段落进行重新排名。这1000个段落是基于BM25检索生成的，没有应用于整个集合的词干。

标准答案集合的评分分为四个等级:（1）完全相关：该段落专用于查询，并包含确切的答案；（2）高度相关：段落提供大量的信息与查询有关，但答案可能有点不清楚，或隐藏在无关的信息中；（3）相关：段落看起来与查询相关，但没有回答查询（提供一些信息与查询有关）；（4）不相关：段落与查询无关。其中根据官网的说明，如果对于二值分类来说，（1）和（2）被认为是相关的，（3）和（4）被认为是不相关的。

* 1. 赛题分析

根据我们前期调研，发现应用于TREC检索任务的主要模型可以分为三类：（1）“trad”：传统信息检索方法如BM25、RM3；（2）“nn”：使用深度学习的方法或者词向量如Duet；（3）“nnlm”：使用大规模预训练语言模型如Bert、XLNet

通过对比上述的三种方案的评价指标，我们发现大规模预训练语言模型“nnlm”效果整体大于其他神经网络模型“nn”和非神经网络模型“trad”。其中语言模型BERT在Passage Ranking的两个子赛道中效果都最好。因此我们选择Bert作为我们这次大作业的baseline，并根据信息检索课程所学知识对baseline进行改进提高算法性能。

* 1. 评价指标

此次比赛所使用的评价指标为在验证集上得到的NDCG@10，具体计算方法如下式所示：

根据Trec官方网站的说明，为了避免不同评价实现方式的差异，官网提供了评测脚本[trec\_eval-9.0.7.tar.gz](https://trec.nist.gov/trec_eval/trec_eval-9.0.7.tar.gz)，从而保证评价指标的唯一性。该评测脚本是使用C语言编写的，可以在1秒钟内返回评测结果，安装使用也十分简单。

下方展示本次实验得到的三个结果的评价结果：

图形用户界面, 文本

描述已自动生成

图1 评价结果

从图1可以看出，实验得到的三个结果的NDCG@10分别达到了0.7155，0.6739和0.7615，三个结果的含义将在后续进行说明。

1. 实现方案
   1. 技术原理

通过广泛调研阅读，我们尝试了多种预训练模型和后续的结构进行微调，并且在学习率参数调节上尝试学习率预热方法。本小节首先介绍我们采用的核心损失函数Re-ranker，随后介绍我们使用的三种微调结构，包括ColBERT和使用Bert输出层的两种输出输出分数，最后介绍warmup学习率预热方法。

* + 1. Re-ranker

Re-ranker是一个交叉编码器，它重新排列上一阶段检索模型的前 个结果，以查询 和段落 的串联作为输入，并输出一个实值分数 。给定标记的正对 和检索的top-k预测中随机抽取的 个强负样本段落，采用列表级损失来训练re-ranker。其损失函数为：



对于我们的检索任务来说，数据集中并没有 个强负样本段落，对于每一个查询来说仅有一个正样本和一个负样本。因此上式也可以看作退化成Softmax损失函数。

* + 1. BERT

自2018年谷歌提出BERT以来，预训练语言模型在自然语言处理领域取得了很大的成功，在多种NLP任务上取得了SOTA效果。BERT本质上是一个基于Transformer架构的编码器，其取得成功的关键因素是利用多层Transoformer中的自注意力机制（Self-Attention）提取不同层次的语义特征，具有很强的语义表征能力。如图2所示，BERT的训练分为两部分，一部分是基于大规模语料上的预训练（Pre-training），一部分是在特定任务上的微调（Fine-tuning）。

图示

描述已自动生成

图2 Bert的两阶段训练

Bert相关预训练模型的输出由四部分组成，其中包括模型各层输出的隐藏状态和序列的第一个token的最后一层的隐藏状态（pooler output，是由线性层和Tanh激活函数进一步处理产生的）。其中模型最后一层输出的隐藏状态（last\_hidden\_state）通常用于命名实体识别，而pooler output的输出通常用于句子分类。因此我们将两种形式的输出作为我们的两种模型微调的结构，后续进行检索实验。

* + 1. ColBERT

在预训练模型兴起之后，full interaction的形式变得非常常见，即把query和passage都输入预训练模型，通过attention在整个神经网络的每一层都进行交互计算，实验结果也证明这样的相似度计算方式非常有效。但缺点是计算速度慢，对于低延迟的应用场景不适用。更严重的问题是，passage无法进行线下预计算和建立索引。因此，这种方案效果虽好，但不能应用在大规模文本检索任务上。原因在于一个q需要和大量的d进行匹配，计算量和文档个数直接相关，在线上对计算资源依赖大。因此以ColBERT等模型提出了所谓的late interaction。前半程先用各种方法进行预计算编码得到query和passage的向量表示，而不是交互信息，然后在后半程对编码后的向量做full interaction，达到性能与速度折衷的目的。Colbert把表示层尽量离线算出，减少在线实时计算量，只在最后层进行MaxSim计算，并且是不需要训练的固定计算，也就是说，这一部分需要的算力资源不多。ColBERT的主要思想是对query与passage在token-level的编码进行匹配计算，并通过MaxSim运算符取出最大值并求和作为最终的分数。具体公式如下：

其中和不是一个向量，而是对每个token编码的向量组，CNN的含义是linear层的降维，Filter的含义是去掉标点符号的token表示。相关性分数的计算：

即对query中的每个token与passage中的所有token计算Sim值并取出最大值，再将其求和作为最终分数。直观上感觉这种算法与BM25等文本匹配算法类似，但计算粒度更为精细。

ColBERT的网络架构如图3所示：

图示

描述已自动生成

图3 ColBERT网络架构图

* + 1. Warmup

Warmup是针对学习率优化的一种策略，主要过程是：在预热期间，学习率从0线性（也可非线性）增加到优化器中的初始预设值，之后使其学习率从优化器中的初始值线性降低到0。如图4所示：

图表, 折线图

描述已自动生成

图4 Warmup训练技巧示意图

由于刚开始训练时，模型的权重是随机初始化的，此时若选择一个较大的学习率，可能带来模型的不稳定或振荡。选择Warmup预热学习率的方式，可以使得开始训练的几个step内学习率较小，在预热的小学习率下，模型可以慢慢趋于稳定，等模型相对稳定后再选择预先设置的学习率进行训练,，使得模型收敛速度变得更快，模型效果更佳。同时在模型训练的后期，模型已经基本训练稳定，此时使用高学习率也可能产生震荡的现象，采用更小的学习率进行训练可以使得模型尽可能接近最优点。

* 1. 预训练模型

我们在[huggingface](https://huggingface.co/)上寻找合适的开源预训练模型作为我们的主干网络，总共找到了三种不同类型的预训练模型共七种，如表1所示：

表1 选取的预训练模型

|  |  |  |
| --- | --- | --- |
| 类型 | 模型名称 | 模型网站 |
| C | albert-base-v2 | <https://huggingface.co/albert-base-v2> |
| bert-base-uncased | <https://huggingface.co/bert-base-uncased> |
| electra-base-discriminator | <https://huggingface.co/google/electra-base-discriminator> |
| M | simlm-base-msmarco | <https://huggingface.co/intfloat/simlm-base-msmarco> |
| cocodr-base-msmarco | <https://huggingface.co/OpenMatch/cocodr-base-msmarco> |
| R | cotmae\_base\_msmarco\_reranker | <https://huggingface.co/caskcsg/cotmae_base_msmarco_reranker> |
| simlm-msmarco-reranker | <https://huggingface.co/intfloat/simlm-msmarco-reranker> |

其中，模型类型为C指这种预训练模型只在普通的预料上进行预训练过，与Re-ranking任务甚至msmarco数据集都没有任何联系；模型类型为M指该预训练模型在msmarco数据集上经过训练，模型类型为R指该预训练模型是使用msmarco数据集并在Re-ranking任务上进行的预训练。

下面介绍三种后续使用的预训练模型。

* + 1. ELECTRA

ELECTRA采用RTD进行预训练。ELECTRA的作者分析了Bert模型的MLM预训练任务的缺点，提出了Replaced Token Detection预训练任务，其训练过程类似GAN，如图5所示。ELECTRA由两个部分组成，第一部分是生成器，生成器将句子中的部分单词进行替换，例如图中将painting替换成了car。第二部分是判别器，判别器用于判断一个句子中每一个单词是否被替换，训练的过程会预测所有的单词，比BERT更为高效。

图示, 示意图

描述已自动生成

图5 ELECTRA结构

* + 1. SimLM

一、预训练阶段

图示

描述已自动生成

图6 SimLM预训练模型架构

如图6所示，在预训练阶段有两种随机的操作，一种是随机mask掉一系列的词，另一个是通过ELECTRA分割的生成器生成mask部分的词。由于随机性这种操作可能会得到相同的句子。根据输入编码器和解码器的不同，上面的操作会采用不同的概率和进行两次：

结构中的编码器是多层的transformer模型，可以使用BERT等模型进行初始化。解码器是2层的transformer模型。预训练任务采用替换后的掩码语言建模。在预训练结束后丢弃解码器只保留编码器做后续的有监督微调任务。

二、微调阶段

图示

描述已自动生成

图7 SimLM微调模型架构

如图7所示，在模型的微调阶段，使用预训练阶段的模型初始化retriver1和retriever2，而对于re-ranker采用ELECTRA预训练模型进行微调。整个微调阶段比较直观而且不需要联合训练或者定期重建索引。

* + 1. CoT-MAE

图示

描述已自动生成

图8 CoT-MAE模型架构

CoT-MAE的设计是为了联合学习文本段内部token的语义信息以及两段文本段和之间的语义关系。为此其结构采用了非对称编码器解码器结构。其中编码器采用的是一个较深的网络并有足够的参数，能够学习到较好的文本表征建模能力。解码器采用的是浅层的网络，目的是希望解码尽量依赖于编码器传入的上下文，从而强迫编码器学习一个更好的文本表征。

在预训练损失计算部分，CoT-MAE分别计算了自监督MLM损失以及语义监督MLM损失。其中自监督的损失和普通的MLM一样，通过将文本段随机mask掉15%，通过没有mask的部分预测mask的部分，其损失函数为：



其中为文本段中mask掉的部分，表示文本段中没有mask的部分。

语义监督MLM则考虑从另一段文本段的没有mask的文本以及上一阶段编码器的上下文embedding 学习mask部分的文本。其损失为：



其中是编码器的最后一层隐藏层输出的上下文embedding，为文本段中mask掉的部分，表示文本段中没有mask的部分。

总的损失为考虑A和B的对偶的形式：



* 1. 实验效果对比

针对前面提出的三类共七种预训练模型，三种后续的微调结构以及Warmup训练策略，我们在NVIDIA Tesla V100 32G \* 4上对这42种组合分别进行训练，batch\_size统一设置为256，得到的训练结果如表2所示：

表2 训练与预测结果（NDCG@10）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | no warmup | | | warmup | | |
| 模型架构 | ① | ② | ③ | ① | ② | ③ |
| albert-base | 0.6658 | 0.6482 | 0.3932 | 0.6630 | 0.6244 | 0.4688 |
| bert-base | 0.6214 | 0.6209 | 0.5508 | 0.6368 | 0.6215 | 0.5163 |
| electra-base | 0.7067 | **0.7155** | 0.4300 | 0.6911 | 0.7035 | 0.3550 |
| simlm-msmarco | 0.6672 | 0.6523 | 0.5997 | **0.6739** | 0.6553 | 0.6049 |
| cocodr-msmarco | 0.6432 | 0.6443 | 0.6699 | 0.6537 | 0.6423 | 0.6578 |
| cotmae-reranker | 0.7605 | 0.7368 | 0.6237 | **0.7615** | 0.7550 | 0.5803 |
| simlm-reranker | 0.7472 | 0.7368 | 0.2427 | 0.7456 | 0.7222 | 0.1766 |

其中，①指后续的模型架构为预训练模型的pooler\_output经过全连接层后直接输出分数，②指后续的模型架构为预训练模型的last\_hidden\_state经过全连接层后直接输出分数，③指使用ColBERT的网络结构。

我们在三种预训练模型上取得的结果分别为：在Re-ranking任务上进行训练后的预训练模型为0.7615；在msmarco数据集上经过训练的预训练模型为0.6739；普通的在其他语料上预训练的模型为0.7155。

从实验结果可以看出，在Re-ranking任务上进行训练后的预训练模型（cotmae\_base\_msmarco\_reranker和simlm-msmarco-reranker）的表现最好，NDCG@10最高可以达到0.7615。这种结果应该是比较符合常理的，因为模型实际上已经在Re-ranking任务上经过了大规模数据的训练，对于我们的小规模数据集几乎不需要怎么进行微调即可取得很好的结果。但是对于在msmarco数据集上经过训练的预训练模型（simlm-base-msmarco和cocodr-base-msmarco）来说，对于Re-ranking任务上的表现甚至还不如普通的在其他语料上预训练的模型（albert-base-v2, bert-base-uncased和electra-base-discriminator）。这可能是因为在msmarco数据集上经过训练的预训练模型的任务与Re-ranking并不匹配，同时我们的训练数据是经过采样后的少量数据，模型并不能学到大规模数据的知识，从而泛化性能并不是很好。

对比三种模型的架构，也可以发现在大多数的情况下，pooler\_output的输出要比last\_hidden\_state的输出更适用于Re-ranking任务，但是也并不绝对，在普通的预训练模型的最好结果是在last\_hidden\_state的输出上取得的。而ColBERT事实上并不适用于Re-ranking任务，ColBERT更聚焦于Retrieval任务，更适合于粗排序，对于Re-ranking任务的精排序并不适合。

1. 实验步骤
   1. 提交说明

由于文件中包含了原始的预训练模型、训练好的模型和镜像，附件比较大。如果失效，下载地址为：

<https://drive.google.com/drive/folders/1jEgrCuCsCVIS1ACE_Brwo0mOm4L1dWNe?usp=sharing>

解压zip文件至本地目录中：

unzip zhangzhao-IR.zip -d zhangzhao-IR

附件提供了Docker和python两种方法运行程序，推荐使用Docker。如果使用docker，需要将IR.tar.gz解压到zhangzhao-IR/images目录下：

tar xvzf IR.tar.gz -C zhangzhao-IR/images

* 1. 文件说明

切换到zhangzhao-IR目录下，有如下文件，文件的具体名称和含义如下列所示：

.

├── README\_submit.md # 说明文件

├── file

│   ├── bert\_cat.py # 模型文件1

│   ├── bert\_sequence\_classification.py # 模型文件2

│   ├── colbert.py # 模型文件3

│   ├── data # 原始数据

│   │   ├── 2019 # 2019年原始数据

│   │   │   ├── 2019qrels-pass.txt

│   │   │   ├── collection.train.sampled.tsv

│   │   │   ├── msmarco-passagetest2019-43-top1000.tsv

│   │   │   ├── qidpidtriples.train.sampled.tsv

│   │   │   └── queries.train.sampled.tsv

│   │   └── 2020 # 2020年原始数据

│   │       ├── 2020qrels-pass.txt

│   │       └── msmarco-passagetest2020-54-top1000.tsv

│   ├── dataset.py # 数据处理文件

│   ├── ensemble\_rank.py # 模型集成文件（未使用）

│   ├── ensemble\_score.py # 模型集成文件（未使用）

│   ├── log.py # 日志配置文件

│   ├── main.py # 主文件

│   ├── models # 训练好的模型文件

│   │   ├── cotmae\_base\_msmarco\_reranker-bert\_sequence\_classification

│   │   │   └── best.pt

│   │   ├── electra-base-discriminator-bert\_cat

│   │   │   └── best.pt

│   │   └── simlm-base-msmarco-bert\_sequence\_classification

│   │       └── best.pt

│   ├── only\_predict # 仅预测脚本

│   │   ├── run\_cotmae\_base\_msmarco\_reranker-bert\_sequence\_classification.sh

│   │   ├── run\_electra-base-discriminator-bert\_cat.sh

│   │   └── run\_simlm-base-msmarco-bert\_sequence\_classification.sh

│   ├── pretrained # 预训练模型文件

│   │   ├── caskcsg

│   │   │   └── cotmae\_base\_msmarco\_reranker

│   │   │       ├── config.json

│   │   │       ├── pytorch\_model.bin

│   │   │       ├── special\_tokens\_map.json

│   │   │       ├── tokenizer\_config.json

│   │   │       └── vocab.txt

│   │   ├── google

│   │   │   └── electra-base-discriminator

│   │   │       ├── config.json

│   │   │       ├── pytorch\_model.bin

│   │   │       ├── tokenizer.json

│   │   │       ├── tokenizer\_config.json

│   │   │       └── vocab.txt

│   │   └── intfloat

│   │       └── simlm-base-msmarco

│   │           ├── config.json

│   │           ├── pytorch\_model.bin

│   │           ├── special\_tokens\_map.json

│   │           ├── tokenizer.json

│   │           ├── tokenizer\_config.json

│   │           └── vocab.txt

│   ├── requirements.txt # 依赖库文件

│   ├── result # 运行结果

│   │   ├── log\_2022\_11\_20\_11\_47\_09 # 运行结果日志

│   │   ├── log\_2022\_11\_21\_05\_01\_28 # 运行结果日志

│   │   ├── log\_2022\_11\_21\_05\_05\_06 # 运行结果日志

│   │   ├── result\_2019qrels\_2022\_11\_20\_11\_47\_09 # 在2019年数据上进行验证的TREC结果文件

│   │   ├── result\_2019qrels\_2022\_11\_21\_05\_01\_28 # 在2019年数据上进行验证的TREC结果文件

│   │   ├── result\_2019qrels\_2022\_11\_21\_05\_05\_06 # 在2019年数据上进行验证的TREC结果文件

│   │   ├── result\_2020qrels\_2022\_11\_20\_11\_47\_09 # 在2020年数据上进行推理后的TREC结果文件

│   │   ├── result\_2020qrels\_2022\_11\_21\_05\_01\_28 # 在2020年数据上进行推理后的TREC结果文件

│   │   └── result\_2020qrels\_2022\_11\_21\_05\_05\_06 # 在2020年数据上进行推理后的TREC结果文件

│   ├── rich\_progress.py # 命令行美化配置

│   ├── train\_and\_predict # 训练+训练后预测脚本

│   │   ├── run\_cotmae\_base\_msmarco\_reranker-bert\_sequence\_classification.sh

│   │   ├── run\_electra-base-discriminator-bert\_cat.sh

│   │   └── run\_simlm-base-msmarco-bert\_sequence\_classification.sh

│   └── trec\_eval-9.0.7.tar.gz # 评测脚本

├── images # Docker镜像路径

│   └── IR.tar # Docker镜像

├── install\_docker.sh # Docker安装脚本

├── install\_python.sh # Python依赖库安装脚本

├── predict\_docker.sh # Docker仅预测脚本

├── predict\_python.sh # Python仅预测脚本

├── train\_and\_predict\_docker.sh # Docker训练+预测脚本

├── train\_and\_predict\_python.sh # Python训练+预测脚本

* 1. 训练流程

为了避免将2019年的数据和2020年的数据进行混用，在最终提交版本的代码中首先将2019年的数据与2020年的数据分开，保证在训练过程中仅使用2019年的数据进行训练和验证，只有在训练完成后预测的时候才去选择2020年的数据进行测试。

├── 2019 # 2019年原始数据

│   ├── 2019qrels-pass.txt

│   ├── collection.train.sampled.tsv

│   ├── msmarco-passagetest2019-43-top1000.tsv

│   ├── qidpidtriples.train.sampled.tsv

│   └── queries.train.sampled.tsv

└── 2020 # 2020年原始数据

    ├── 2020qrels-pass.txt

    └── msmarco-passagetest2020-54-top1000.tsv

对于数据的预处理，我们通过collection.train.sampled.tsv和queries.train.sampled.tsv内部的文本与ID的对应关系，将qidpidtriples.train.sampled.tsv中每条查询的一个正样本和负样本的ID替换成实际的文档，随后通过预训练模型自带的tokenizer分词器进行处理，输入到dataloader中进行训练和测试。整个过程完全在线进行，不需要提前对原始数据进行处理。

训练的主入口文件是file/main.py，其中提供了22个超参数用于对代码中的可选项进行选择，保证代码运行过程的鲁棒性。

训练脚本如下所示：

# Variables

SEED=42

GPU='0 1 2 3'

TRAIN\_BATCH=256

MODEL=bert\_sequence\_classification

BERT='pretrained/caskcsg/cotmae\_base\_msmarco\_reranker'

TIMESTAMP=run3

# 使用2019年的数据进行训练

python main.py \

--train \

--batch ${TRAIN\_BATCH} --datetime ${TIMESTAMP} --epoch 20 --gpu ${GPU} --lr 3e-5 --seed 42 --early\_stop 20 \

--data\_folder\_dir 2019 --train\_document collection.train.sampled.tsv --train\_query queries.train.sampled.tsv --qid\_pid qidpidtriples.train.sampled.tsv --test\_data\_file msmarco-passagetest2019-43-top1000.tsv --test\_result\_file 2019qrels-pass.txt \

--save --bert ${BERT} --model ${MODEL} --warmup 0.1

脚本的超参数设置，首先保证是训练模式，随后设置了训练的batch\_size大小、输出文件的时间戳（用于隔离不同的结果文件）、训练轮数、使用的GPU、学习率、随机种子以及早停轮数（如果在长时间内模型在验证集的效果上没有提升，则认为已经没有再次训练的必要了）。然后设置了训练阶段的数据集，可以看出使用的都是2019年的数据集，2020年的数据并没有参与到训练过程的任何部分；最后设置了使用的预训练模型、后面的微调策略以及warmup策略的学习速率。

设置好脚本后即可通过运行脚本直接训练，训练过程中会同时在控制台与文件中输出日志，从而保证训练结果的可观察性。同时如果添加了--board选项，会在tensorboard中输出关键的指标，如学习率、损失以及目前的NDCG@10指标等。

文本

描述已自动生成

图9 训练日志示意图

在训练的过程中，每训练一轮就调用评价脚本进行一轮测试，如果测试的结果比上一轮的结果更好，就会保存一个模型到本地，或者将上一个保存的模型替换掉。这样训练结束后最终保存下来的模型是在2019年数据上测试的最好的模型。中间并没有2020年的数据参与。

* 1. 预测流程

预测流程的超参数配置大致如下所示：

# Variables

SEED=42

GPU='0 1 2 3'

TEST\_BATCH=1024

MODEL=bert\_sequence\_classification

BERT='pretrained/caskcsg/cotmae\_base\_msmarco\_reranker'

TIMESTAMP=run3

# 测试2020年的数据

python main.py \

--predict \

--batch ${TEST\_BATCH}  --datetime ${TIMESTAMP} --gpu ${GPU} \

--data\_folder\_dir 2020 --test\_data\_file msmarco-passagetest2020-54-top1000.tsv --test\_result\_file 2020qrels-pass.txt \

--load --bert ${BERT} --model ${MODEL}

其中的时间戳会去寻找刚刚对应时间戳下训练好的模型，测试的数据均为2020年的数据，其余的参数与训练大致相同。测试过程中也会同时在文件中和控制台输出对应的预测信息，并且在最后一行打印测试指标。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图10 测试日志示意图

1. 参考文献

[1] Yu Y, Xiong C, Sun S, et al. COCO-DR: Combating Distribution Shifts in Zero-Shot Dense Retrieval with Contrastive and Distributionally Robust Learning[J]. arXiv preprint arXiv:2210.15212, 2022.

[2] Khattab O, Zaharia M. Colbert: Efficient and effective passage search via contextualized late interaction over bert[C]//Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval. 2020: 39-48.

[3] Santhanam K, Khattab O, Saad-Falcon J, et al. Colbertv2: Effective and efficient retrieval via lightweight late interaction[J]. arXiv preprint arXiv:2112.01488, 2021.

[4] Wu X, Ma G, Lin M, et al. Contextual mask auto-encoder for dense passage retrieval[J]. arXiv preprint arXiv:2208.07670, 2022.

[5] Wang L, Yang N, Huang X, et al. Simlm: Pre-training with representation bottleneck for dense passage retrieval[J]. arXiv preprint arXiv:2207.02578, 2022.

[6] Craswell N, Mitra B, Yilmaz E, et al. Overview of the TREC 2019 deep learning track[J]. arXiv preprint arXiv:2003.07820, 2020.

[7] Craswell, N., Mitra, B., Yilmaz, E., and Campos, et al. Overview of the TREC 2020 deep learning track[J]. arXiv preprint arXiv: 2102.07662, 2021.

[8] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

[9] Clark, K., Luong, M. T., Le, Q. V., & Manning, C. D. (2020). Electra: Pre-training text encoders as discriminators rather than generators. arXiv preprint arXiv:2003.10555.