|  |
| --- |
| **SSR ：Super smart RAG use token vector inverted merge engine for retrieval**  **Yinyue** |
|  |
|  |
|  |
|  |
|  |
|  |

Abstract

鉴于当前RAG系统基于BM25全文检索会遇到的同义词匹配以及文本段向量匹配时会各自遇到的召回率较低等问题，本文提出并实现了一种基于Token向量匹配的RAG系统：SSR（Super smart RAG）。首先我们搭建了Token向量检索引擎，然后在此基础上实现了结合大模型内外部知识进行两次生成的结果生成模块，从而组成完整RAG系统。在SciQ、BoolQ、OpenBookQA三个客观题数据集上，分别对比LLM不使用RAG，以及分别使用BM25索引、全文向量索引、token 向量索引作为SSR基础检索引擎的效果。最终发现在BoolQ数据集上，SSR都取得了较好的效果。

Introduction

当前的RAG系统，面对文本类型的数据，通常会利用对文本段进行分词，利用倒排索引构建文本段的字面索引，或者对文本段进行向量化，利用向量检索引擎构建向量索引，随后在构建好的索引上，寻找与用户问句最相关的文本段落，作为生成环节的候选参考。

但是字面的倒排索引需要面对的明显的问题是如果不在查询时对用户的问题或者在构建索引时对文本段落，进行额外的同义词扩展的处理，是无法检索出虽然与问题中语义相同，但是字面不同的文本的。（找一些参考文献）

而对文本段构建向量索引，虽然可以一定程度上解决语义匹配的问题，但是当待检索的文本数据量达到一定程度的时候，其召回率会下降，甚至低于字面匹配的倒排索引的方式。

在本工作中，我们介绍SSR，一种新型的RAG框架。它在Token的向量构建的索引基础上，利用倒排索引合并的方式作为基础的召回引擎；并且采用模型内外部知识融合反思的方式生成最终的结果。

首先会对框架的整体架构以及索引构建和查询两个主要流程进行介绍础。随后会对检索引擎模块和使用内外部知识融合反思的生成模块分别进行介绍。随后会通过在BoolQ数据集上，分别对比不使用RAG的LLM，使用BM25索引，全文向量索引，token向量索引三种不同检索引擎的RAG的结果进行对比。最后会对当前存在的问题挑战以及后续工作进行描述。

Related Work

SSR主要分为两个主要的模块，基础索引模块以及两阶段的生成模块。在索引构建模块中，采用parquet的文件格式，存储了Token级别的向量索引、每个分段的原始的文本等正排信息。在此基础上实现了索引的构建、查询两大流程。两阶段的生成模块则通过将第一次不依赖检索知识的生成结果和将检索结果一起送入LLm进行第二次生成，达到融合内外部知识的目的。

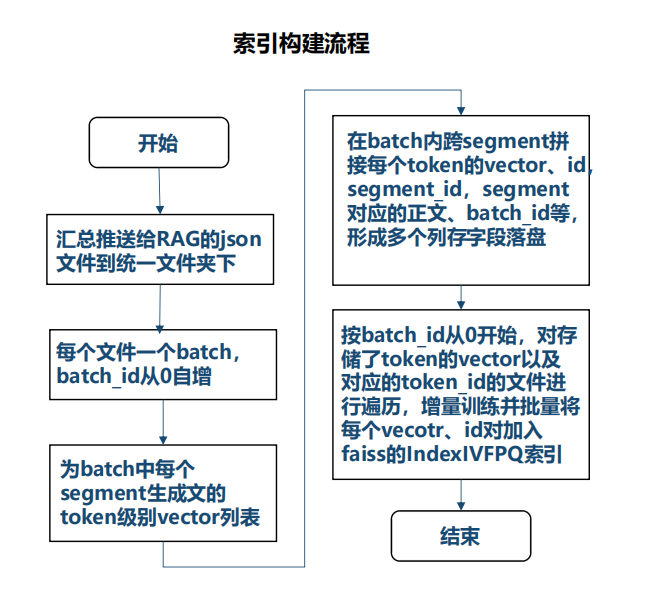


图2-1 索引构建流程

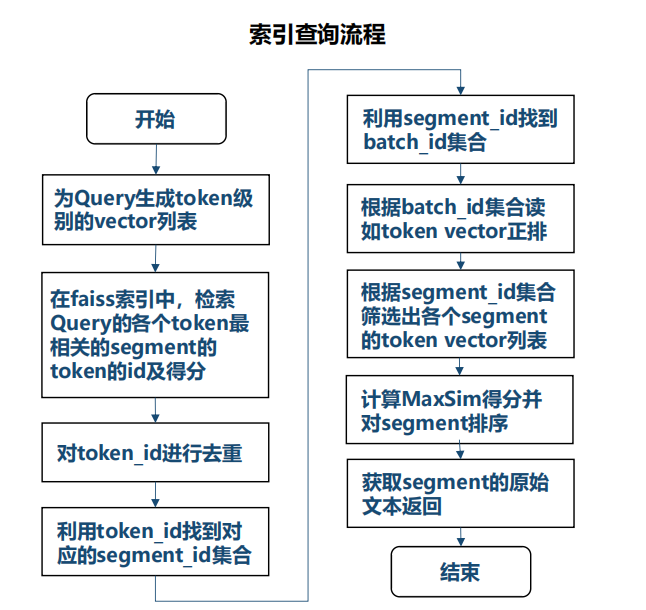


图2-2 索引查询流程

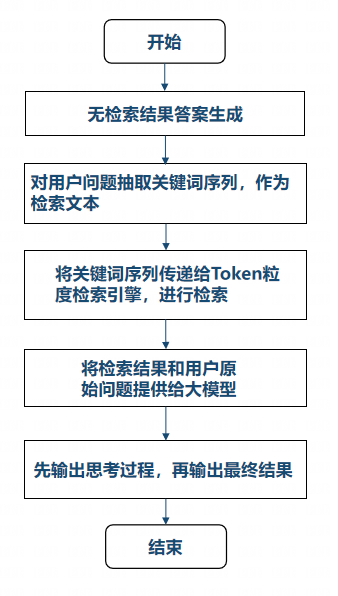


图2-3 两阶段生成流程

Token粒度的向量检索引擎

设计思路

在传统搜索引擎中，一般通过对文本进行不同粒度的分词后，构建词到文档的倒排索引，随后使用跳表等方式提升倒排索引的合并速度，再使用BM25等算法计算最终的文档和查询问题的相关性。而如faiss等向量检索引擎，对需要构建索引的文本整体进行固定维度向量的向量化，随后利用局部敏感哈希、聚类等技术，提升和查询问题向量的相似度计算过程。

基于文本的全文检索，无论是对文档还是对于查询问题，都是在字面级别进行操作，意味着当检索词和待检索文本中存在的词字面不同时，哪怕语义相同，也无法被匹配到。比如一篇文档中存在“开心”、“高兴”这样的词语，使用“愉快”这个词去进行检索的时候就无法命中，造成召回结果的缺失。

而faiss等向量检索引擎，在被用在rag的时候，将无论长短的文本，统一压缩到统一长度的向量空间。AI工具公司EyeLevel.ai的数据科学家Daniel Warfield和前IBM Watson高级工程师Dr. Benjamin Fletcher博士在RAG技术规模化研究过程中发现一个现象：向量搜索的准确性随着数据量的增加而显著下降。

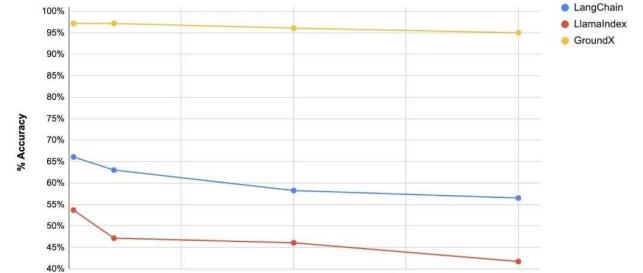


图3-1 EyeLevel.ai文档数量与RAG准确率关系

在图3-1中，可以看到GroundX系统使用相较LangChain、LlamaIndex而言，准确率的下降相对较低。与GroundX的RAG系统实现是通过了对于某些关键实体构建单独的向量索引，会通过在这些实体的向量索引上进行检索，与全文向量检索组合使用，提升RAG系统最终的准确率。

综上，这也促使我们有了进一步对所有的Token都构建向量索引，然后先通过Token级别的向量相似度匹配，找到和query相关性最高的Token集合，再以类似文本检索倒排索引合并的方式，聚合Token集合所关联的文档段落的想法。

然而这样的实现方式，Token的量级比文档的量级一般是百倍、千倍，势必会造成存储时的空间、查询时的时间消耗高等问题。

* 1. 空间维度上的问题与挑战

3.2.1 语义表征的维度

在各类大语言模型当中，语义表征的维度一般是千级别的。

如果在我们的Token粒度检索引擎中依然使用上千级别的Token向量，则会造成巨大的存储消耗。

表3-1 不同大模型重要属性参数



根据资料，Colbert在计算问题和文档相关性的时候，也是采用了Token级别的向量，利用MaxSim Score的打分方式计算相关性。其Token的语义向量维度只有128维，相较其他模型上千维的维度，缩小了接近10倍。因为，我们在进行检索引擎的Token向量化的过程中，借助Colbert，使得获得的Token也是128维，降低存储空间。

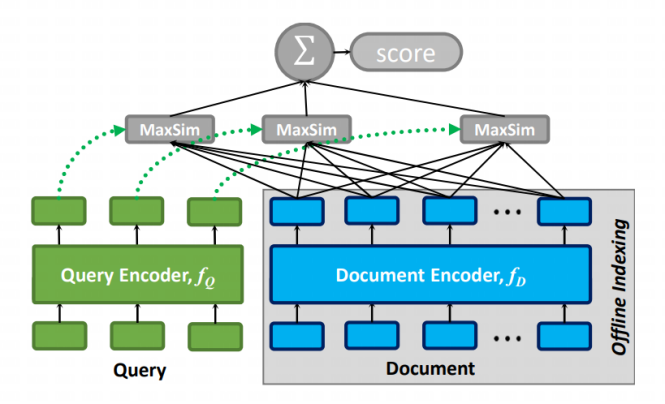


图3-2 Colbert的MaxSim打分方式

3.2.2 浮点数的精度

除了向量维度的大小，对于空间影响比较大以外，向量每个维度上的精度高低对于存储也有比较大的影响。表征一个浮点数，当使用float64的时候，会比使用float32的时候多一倍的存储空间。因为寻找一个合适的精度，可以进一步提升存储空间的利用率。

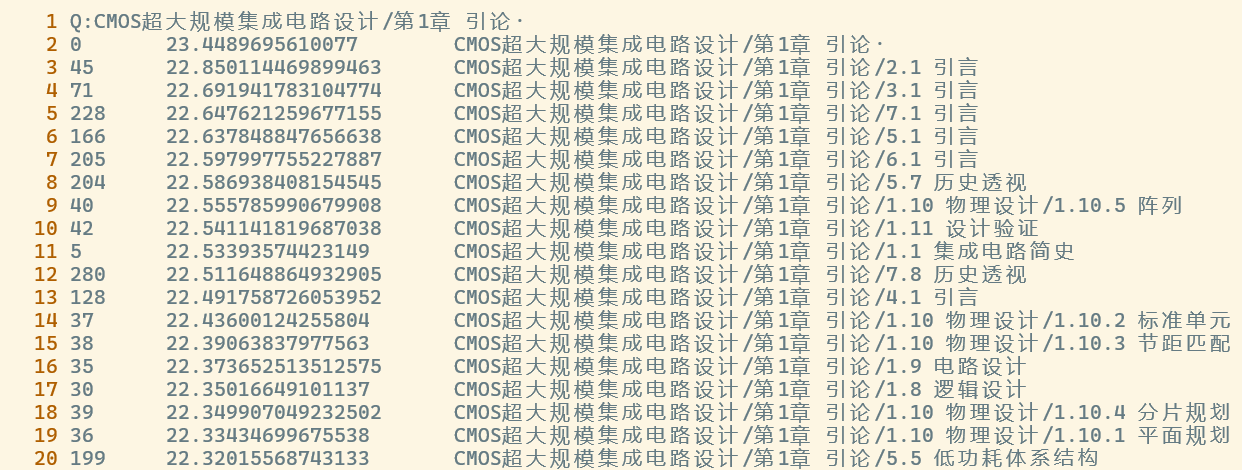
经过对比，float16和float64在利用MaxSim Score计算文本相关性的时候，差距不大。因此最终选择了对于经过Colbert产出的向量结果，进行float16的精度压缩。

图3-2 float64时相关性排序示例

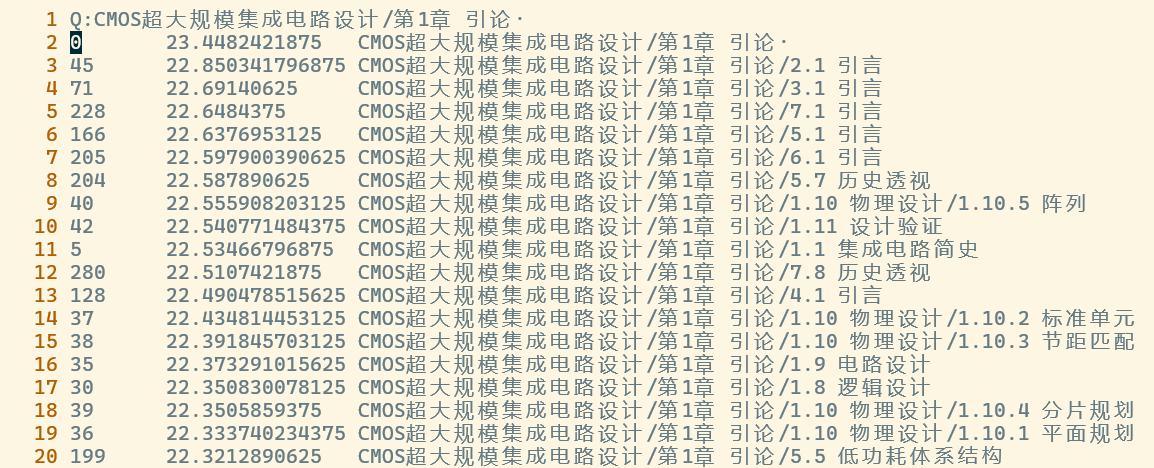


图3-3 float16时相关性排序示例

* 1. 时间维度上的问题与挑战

**3.3.1 Token向量索引结构**

当faiss使用最简单的IndexFlatL2时，因为是暴力检索，虽然精度比较高，但是要都加载到内存中，随着向量数量增加，检索匹配的时间增长明显。

为了降低向量检索的耗时，使用IndexIVFPQ这样通过聚类+乘积量化的方式，可以将检索时间降低到十几ms。

如图3-5所示，展示了索引构建的IndexIVFPQ索引构建的内部流程。

原始数据需要首先转换为如第3.2节中所示，形成128维，使用float16存储的向量。

随后对于产出的向量使用k-means进行聚类，因为是对token维度进行向量索引，所以可以依据token id的量级，加上同义词的参考进行聚类中心数量的选择。相比较文档全文向量而言，有更明确的聚类数量的参考。

当聚类完成之后，就要对每一个向量选择其最近的聚类中心，分配到改聚类中心所属的聚类簇中，形成倒排链表。每个链表表示同一个聚类簇中的向量。

分配完成之后，计算每一个向量和和中心向量计算残差向量。如果残差的方差越小，说明该簇内的向量越集中，越有利于后续进一步进行PQ子量化过程。

对于每一个残差向量，进一步分解为m个段每个段的子向量维度为nbits。然后对于每个段内的子向量，再进行k-means聚类，本次聚类中心数量为2^nbits，每个中心按照自增编码形成PQ码表。这样每个子向量都有一个所属的子聚类中心。一个原来的向量就形成了对应m个PQ码的新的量化向量。

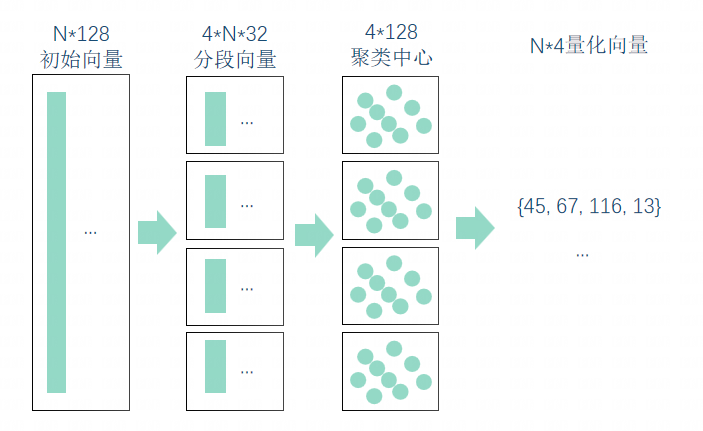


图3-4 PQ流程

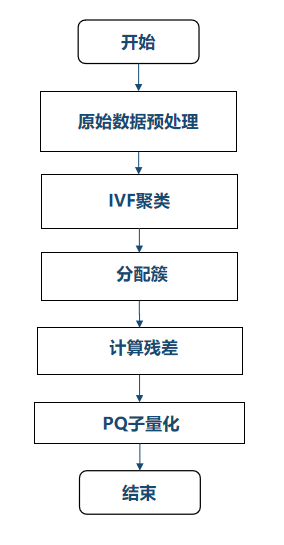


图3-5 IndexIVFPQ索引构建流程

如图3-6所示，展示了索引查询的IndexIVFPQ索引构建的内部流程。

查询问句利用Colbert的Query向量生成网络，生成同样是128维的向量。

分别在顶层的聚类中心计算相似度，找到距离最近的前n个聚类簇。在这些簇中进行后续的查找。

对选定的每个聚类中心，分别计算向量残差，形成残差向量。

对于残差向量，和索引时PQ计算相同，划分为m个段，每个段向量维度nbits。然后计算每个段的子向量和该段内的2^nbits个子聚类中心的距离，形成预计算的距离表。再通过遍历之前索引时通过PQ量化后的各个子向量，用该子向量所属的子聚类中心来查找刚才预计算的距离表，作为当前段的近似距离。累加m段距离作为整个向量的近似距离。

对于计算后的近似距离进行排序，获得最终相似向量序列。

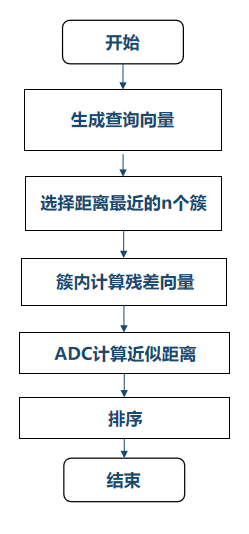


图3-6 IndexIVFPQ索引检索流程

**3.3.2 列存**

如果依然使用每行一个记录的方式去存储token和所在的文本段，文本段所在的长篇文本，每个文本段的id以及原文等信息的话，定位每次所需的信息就需要顺序遍历文件去定位获取。在根据token id去查找文本段，或者根据文本段id去查找原文、所在长篇文本id等信息时，逐行扫描会消耗大量时间。如果改成h5列存的方式，对数据做好分区，可以在几ms内完成如根据文本段获取原始文本的过程。

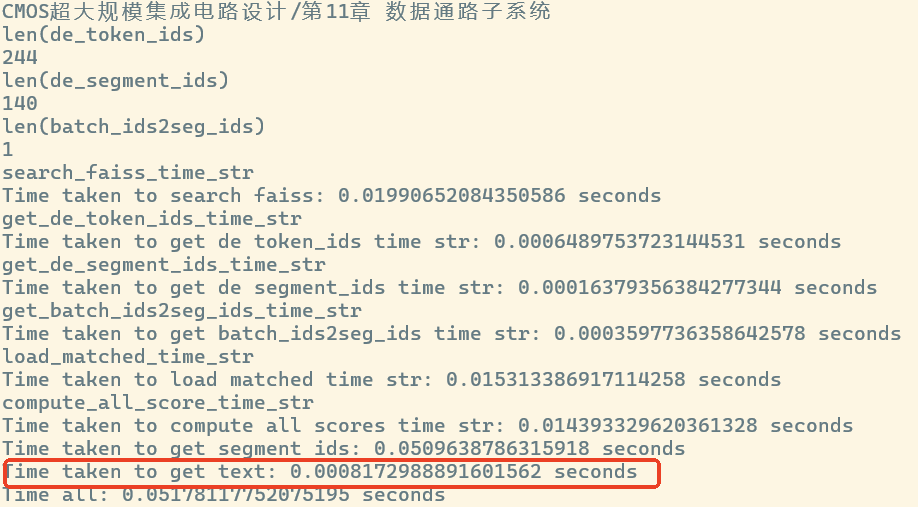


图3-6 IndexIVFPQ索引检索流程

两阶段的答案生成

设计思路

我们观察到，当直接大语言模型提供检索的结果进行生成的话，回答有时会过多参考检索结果。即使检索结果没有错误，但是如果所问的问题需要较完整的结构化知识去回答时，也会过多的偏向检索的知识，而忽略掉其他更完整的知识框架。所以我们希望从以下几个方便来提升生成结果的完整性。

**问题预处理**

在用户的问题中，可能会存在一些为了让模型生成结果更为符合诉求的生成时指令，比如生成长度小于100个字等。这些文本实际上对于检索过程没有实际用处，反而会为检索带来冗余信息。

为了能够让检索结果更精准，在检索之前使用大模型对用户的问题进行关键词序列的提取，这样可以保证检索过程中没有冗余词的干扰，提升召回结果的准确率。

内外部知识融合

为了解决过于依赖检索结果的问题，我们使用两次生成的方式。第一次生成时，仅将用户的原始问题输入大模型，不进行相关文本段的检索。随后将用户的原始问题、第一次生成的结果、检索结果一同作为大模型的输入，进行二次生成。

第一次的生成结果因为没有检索结果的干扰，主要代表的是大模型通过模型蕴含在内部参数中的知识所回答出来的结果。这样就可以保证在第二次输入的时候，也有明确的无干扰的答案同样作为参考。使得如果检索结果对第一次生成答案有帮助的话，就会在第一次生成结果基础上进行补充；如果检索结果有误或者对答案生成无帮助，那么同样可以保留第一次的检索结果。

预思考

如DeepSeek-R1这样的天生具有推理思考能力的大模型，其会在给出最终的答案之前，有一段较长的预先思考的过程。此类模型的思考过程是通过训练实现的原生能力，这样的思路对无原生推理部分的模型同样有借鉴意义。对于没有这样原生能力的模型，我们人为的在Prompt中，增加额外的要求，将输出的格式限制为：

<thoughts>

填充思考内容，本内容不作为最终答案，只作为最终答案生成前的思考试错。

</thoughts>

<answer>

填充最终的答案。

</answer>

通过这样在最终答案生成之前，生成思考过程，可以有效的提升生成结果的准确率

实验结果

数据集

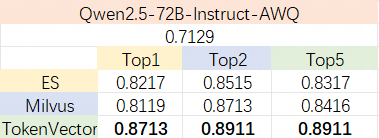
BoolQ是一个从Google搜索引擎查询中收集起来的15942个问题。每个问题都有“是”或“否”两个选项。

测试方法与测试结果

为了对比基于Token向量的检索引擎与基于BM25的全文检索和基于全文向量的检索效果的优劣，我们通过将SSR的检索引擎部分进行替换，保留生成模块的逻辑。对BoolQ数据集的前100条问题进行测试。

基座模型使用的是Qwen2.5-72B-Instruct-AWQ。在不使用获得的准确率是0.7129。其余使用不同检索引擎以及采用不同TopN文档进行二次生成的准确率，如表5-1所示，使用Token向量的检索引擎在不同的TopN准确率均高于使用ES的全文文本检索以及使用Milvus的全文向量检索。

表5-1 BoolQ测试效果



结论

我们通过在BoolQ测试集上的测试，证明了基于Token向量检索的SSR，超越了基于全文文本索引和全文向量索引的模式。从一定程度上解决了全文索引需要额外同义词的转换，以全文向量索引随着文本数量增加召回准确率快速下降的问题。证明了SSR的有效性。