RAG 系统优化实战:我是如何将检索准确率提升 25% 的

作者: 李晨
发布时间: 2024年6月18日
标签: #RAG #NLP #检索优化 #大语言模型

前言

最近半年,我在星辰科技的工作主要围绕公司内部的智能问答系统展开。这是一个典型的 RAG(Retrieval-Augmented Generation)应用场景——用户提问,系统从海量文档中检索相关内容,然后用 LLM 生成回答。

刚接手这个项目时,系统的检索准确率只有 64%,用户经常抱怨"答非所问"。经过三个月的优化,我们把这个数字提升到了 89%,响应延迟也从 2.3 秒降到了 1.4 秒。

这篇文章想分享一下我在这个过程中踩过的坑和积累的经验,希望对做类似系统的同学有所帮助。

问题诊断:为什么检索效果这么差?

在开始优化之前,我花了两周时间分析系统的问题。我随机抽取了 500 条用户查询,逐条检查检索结果,发现了几个主要问题:

1. Chunk 策略太粗糙

原系统简单地按 512 token 固定长度切分文档,完全不考虑语义完整性。这导致很多重要信息被切断了,比如:

```
Chunk 1: "...我们的退款政策如下:1. 购买后7天内可以"
Chunk 2: "无理由退款。2. 超过7天但在30天内,如果产品..."
```

用户问"多久可以退款",第一个 chunk 被检索到了,但信息不完整,LLM 只能回答"7天",完全遗漏了后面的条款。

2. Embedding 模型选择不当

原系统用的是通用的 `text-embedding-ada-002`,对我们的业务领域(电商客服)针对性不强。我做了个对比实验:

- 通用模型:相似度得分 0.68
- 领域微调模型:相似度得分 0.82

差距还是很明显的。

3. 只用了单一检索策略

系统只做了稠密向量检索(Dense Retrieval),对于一些关键词匹配场景效果不好。比如用户问"订单号 123456 的物流信息",向量检索经常召回不相关的内容,因为它过度关注语义,忽略了精确匹配的重要性。

4. 缺少重排序机制

检索出来的 top 10 文档直接喂给 LLM,没有二次排序。这导致真正相关的文档可能排在后面,被截断了(我们只取 top 3 给 LLM)。

优化方案:一步步提升检索效果

第一步:改进 Chunk 策略

我设计了一个语义感知的分块算法:

1. 先用 spaCy 做句子分割
2. 计算相邻句子的语义相似度
3. 当相似度低于阈值时,在这里切分
4. 保证每个 chunk 大小在 256–512 token 之间
5. 重叠 50 token,避免信息丢失

实现代码大概长这样(简化版):

```python
```

```python
def semantic_chunking(text, min_size=256, max_size=512, overlap=50):
    sentences = nlp(text).sents
    chunks = []
    current_chunk = []
    current_size = 0

    for i, sent in enumerate(sentences):
        sent_tokens = len(sent)

        if current_size + sent_tokens > max_size:
            # 检查语义相似度
            if i < len(sentences) - 1:
                similarity = calculate_similarity(sent, sentences[i+1])
                if similarity < 0.5:  # 低相似度,可以切分
                    chunks.append(' '.join(current_chunk))
                    current_chunk = [str(sent)]
                    current_size = sent_tokens
                    continue

        current_chunk.append(str(sent))
        current_size += sent_tokens

    if current_chunk:
        chunks.append(' '.join(current_chunk))

    return chunks
```

这个改动让检索准确率从 64% 提升到了 71%,效果立竿见影。

第二步:混合检索(Hybrid Search)

我实现了稠密向量 + 稀疏向量的混合检索:

- **稠密向量(Dense):** 用 embedding 模型,擅长语义匹配
- **稀疏向量(Sparse):** 用 BM25 算法,擅长关键词匹配

两路检索结果用加权融合:

```python
```

```python
def hybrid_search(query, top_k=10, alpha=0.7):
    # 稠密检索
    dense_results = vector_db.search(
        query_embedding=embed(query),
        top_k=top_k
    )

    # 稀疏检索
    sparse_results = elasticsearch.search(
        query=query,
        top_k=top_k
    )

    # RRF (Reciprocal Rank Fusion) 融合
    final_scores = {}
    for rank, doc in enumerate(dense_results):
        final_scores[doc.id] = alpha / (rank + 60)

    for rank, doc in enumerate(sparse_results):
        if doc.id in final_scores:
            final_scores[doc.id] += (1-alpha) / (rank + 60)
        else:
            final_scores[doc.id] = (1-alpha) / (rank + 60)

    # 按分数排序
    sorted_docs = sorted(final_scores.items(), key=lambda x: x[1], reverse=True)
    return sorted_docs[:top_k]
```

这里的 `alpha` 参数我通过网格搜索确定的,最终设置为 0.7(偏重语义检索)。

混合检索让准确率又提升了 8 个点,达到 79%。

第三步:引入重排序模型(Reranker)

检索出 top 20 候选后,我用了一个 cross-encoder 模型做重排序。Cross-encoder 比 bi-encoder 更准确,因为它能同时看到 query 和 document,捕捉更细粒度的相关性。

我选择的是 `bge-reranker-large`,在我们的测试集上 NDCG@10 达到 0.91。

python

```python
from transformers import AutoModelForSequenceClassification, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('BAAI/bge-reranker-large')
model = AutoModelForSequenceClassification.from_pretrained('BAAI/bge-reranker-large')

def rerank(query, candidates, top_k=3):
    pairs = [[query, doc.content] for doc in candidates]
    inputs = tokenizer(pairs, padding=True, truncation=True, return_tensors='pt')
    scores = model(**inputs).logits.squeeze()

    # 按分数排序
    ranked_indices = torch.argsort(scores, descending=True)[:top_k]
    return [candidates[i] for i in ranked_indices]
```

重排序让准确率从 79% 跃升到 86%,这是单个改动中提升最大的。

第四步:Query 改写与扩展

有些用户的问题表达不够清晰,或者过于简短。比如用户问"能退吗",系统不知道是问退款、退货还是退换货政策。

我用 LLM 做了 query 改写:

```python
def query_rewrite(query):
    prompt = f"""你是一个智能客服助手。用户的问题可能表达不清楚,请你:
1. 补全问题的上下文
2. 明确用户的真实意图
3. 生成3个可能的改写版本

原始问题:{query}

改写后的问题(JSON格式):"""

    response = llm.generate(prompt)
    rewrites = json.loads(response)
    return rewrites
```

然后对每个改写版本都做检索,最后合并结果。这个方法在处理模糊查询时特别有效,又提升了 3 个点。

第五步:领域知识微调 Embedding 模型

最后,我用我们自己的数据(3万对 query-document 相关性标注)微调了 embedding 模型。

训练数据构造:

- 正样本:用户点击的文档
- 负样本:检索到但用户没点击的文档
- 困难负样本:BM25 高分但语义不相关的文档(这个很重要!)

用 `sentence-transformers` 库很容易实现:

```python
from sentence_transformers import SentenceTransformer, InputExample, losses
from torch.utils.data import DataLoader

model = SentenceTransformer('BAAI/bge-large-zh-v1.5')

train_examples = [
    InputExample(texts=[query, pos_doc], label=1.0),
    InputExample(texts=[query, neg_doc], label=0.0),
    ...
]

train_dataloader = DataLoader(train_examples, shuffle=True, batch_size=16)
train_loss = losses.CosineSimilarityLoss(model)

model.fit(
    train_objectives=[(train_dataloader, train_loss)],
    epochs=3,
    warmup_steps=100
)
```

微调后的模型在我们的测试集上,召回率@10 从 72% 提升到 84%,准确率最终达到了 89%。

工程优化:降低响应延迟

提升准确率的同时,我们也要关注性能。原系统响应延迟 2.3 秒,用户体验很差。我做了几个优化:

1. 向量索引优化

从 Flat 索引换成了 HNSW(Hierarchical Navigable Small World)索引。虽然构建时间长了,但检索速度提升了 5 倍。

```python
```

```
import faiss

# 原来:Flat 索引
index = faiss.IndexFlatIP(dimension)

# 优化后:HNSW 索引
index = faiss.IndexHNSWFlat(dimension, 32)  # 32 是 M 参数
index.hnsw.efConstruction = 200
index.hnsw.efSearch = 50
```