实验介绍

【实验名称】：基于混合检索的电商信息检索系统

【实验目的】：

随着电子商务的迅速发展，商品信息检索已成为电商平台的核心功能之一。在实际使用中，用户可能会存在拼写错误、描述性表达、同义词等情况，传统的基于关键词匹配的检索方法往往无法满足用户多样化的查询需求。本实验旨在设计并实现一个基于混合检索技术的电商信息检索系统，该系统能够：

- 支持传统关键词检索和现代语义理解检索结合
- 提供智能拼写纠错功能，提升用户查询体验
- 实现同义词扩展，增强检索召回率
- 支持多维度过滤和排序，满足个性化搜索需求
- 提供直观的 Web 界面，便于用户交互和结果展示

【实验环境】：

操作系统: Windows 11

Python 版本: Python 3.96

核心依赖库:

fastapi==0.104.1

uvicorn[standard]==0.24.0

pandas==2.1.3

numpy==1.25.2

scikit-learn==1.3.2

datasets==2.14.6

python-multipart==0.0.6

sentence-transformers==2.2.2

torch>=1.9.0

前端技术:

HTML5 结构搭建页面框架

CSS3 实现样式设计

JavaScript 实现交互逻辑

数据集:

Amazon 商品数据集

https://huggingface.co/datasets/randomath/Amazon-combined?library=datasets

语义编码模型:

all-MiniLM-L6-v2

【参考文献】：

实验内容

【实验方案设计】：

一、系统整体架构

本实验设计了一个四层架构的电商信息检索系统：

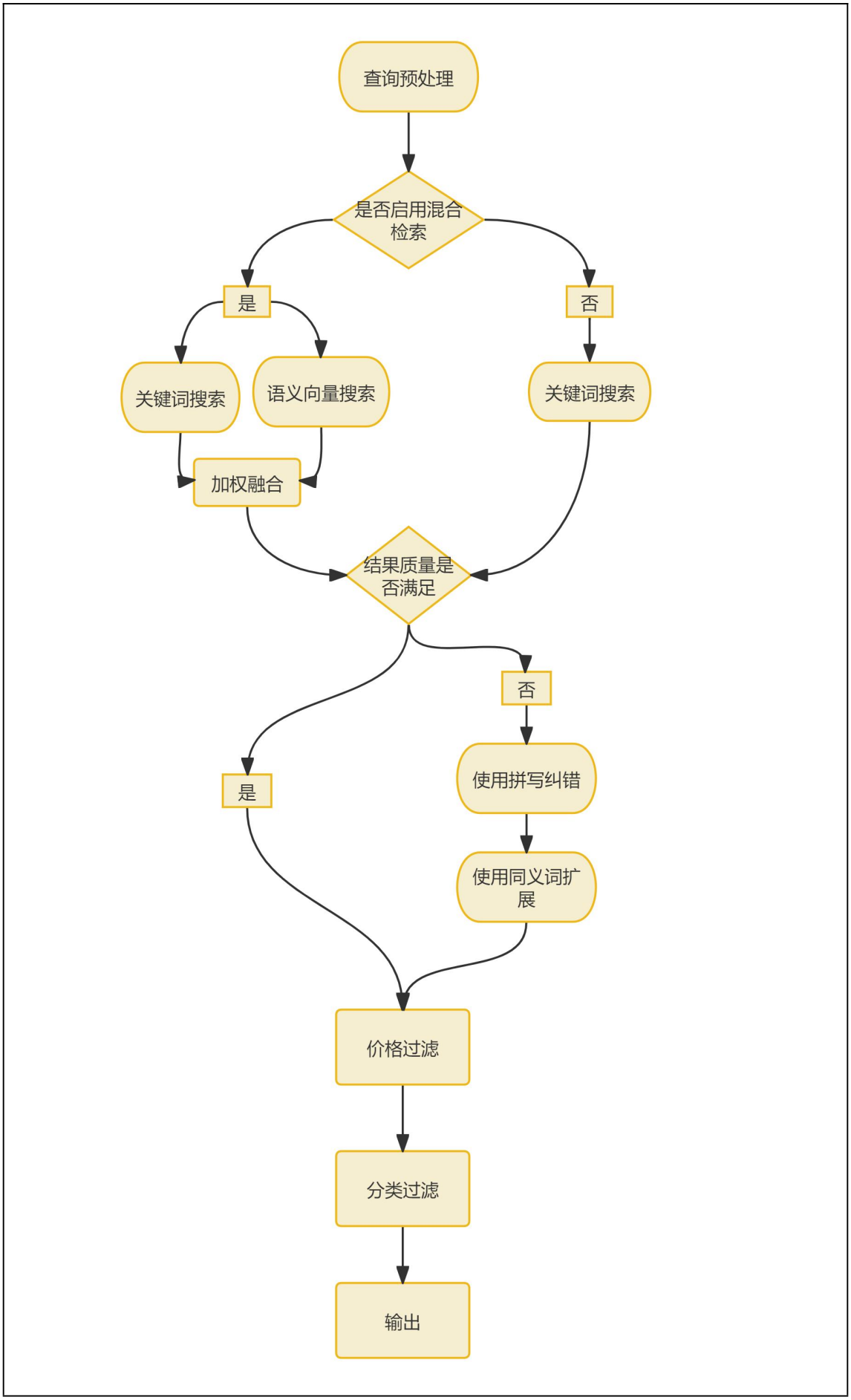
前端展示层：展现 web 界面，实现用户交互

API 服务层：使用 FastAPI 提供数据接口，实现前后端交互

搜索引擎层：检索算法的核心

数据处理层：数据预处理和存储

下图为简要的实现流程


查询预处理
是否启用混合检索
是
否
关键词搜索
语义向量搜索
关键词搜索
加权融合
结果质量是否满足
否
是
使用拼写纠错
使用同义词扩展
价格过滤
分类过滤
输出

二、检索方案设计

本系统采用关键词检索与语义检索相结合的混合检索策略。

1、TF-IDF 关键词检索

（1）TF-IDF 算法原理

TF-IDF (Term Frequency-Inverse Document Frequency) 算法是现代信息检索系统的基石，该算法的核心在于：一个词语对文档的重要性既取决于它在该文档中出现的频率，也取决于它在整个文档集合中的稀有程度。

词频（Term Frequency, TF）指一个词在文档中出现的频率，用于衡量词对文档的代表性。当一个词语在文档中反复出现时，我们有理由相信这个词语承载了该文档的重要语义信息。然而，仅仅依赖词频会导致常见词汇（如"的"、"是"、"在"等功能词）获得过高的权重，这些词汇在大多数文档中出现都很频繁，因此缺乏区分不同文档的能力。

为解决这个问题，引入了逆文档频率（Inverse Document Frequency, IDF）。其核心思想为：如果一个词语在很多文档中都出现，那么它对于区分不同文档的价值就相对较低；反之，如果一个词语只在少数文档中出现，那么它就具有更强的区分能力和更高的信息价值。这种设计使得 TF-IDF 能够自动降低高频通用词的权重，同时提升具有区分性的词汇或的重要性。

在具体的计算过程中，词频的计算需要考虑文档长度的影响。由于长文档中词语出现的绝对次数往往更高，简单的原始频率计算可能会偏向于较长的文档。因此，常用的词频计算方法包括相对频率（词语出现次数除以文档总词数）、对数归一化（对原始频率取对数）以及双重归一化等策略。这些归一化方法的选择会直接影响最终的检索效果，需要根据具体的应用场景和数据特征进行调整。

在逆文档频率的计算上，传统的 IDF 公式采用对数函数来平滑频率分布，避免出现极端的权重值。当某个词语在所有文档中都出现时，其 IDF 值趋向于零；当某个词语只在一个文档中出现时，其 IDF 值达到最大。这种对数缩放不仅使得权重分布更加合理，还确保了数值计算的稳定性。

（2）代码实现

本系统采用 scikit-learn 库中的 TfidfVectorizer 来实现 TF-IDF 算法

这里我们创建了 TF-IDF 向量化器，后训练并转换文档。

在 fit_transform(documents) 中实际上执行了两个操作：

①学习阶段（fit）

这一步会扫描所有文档，构建词汇表，然后计算每个词的文档频率，再过滤词汇，最终确定特征词汇表

②转换阶段（transform）

这一步会将每个文档转换为 TF-IDF 向量并计算词频（TF）和逆文档频率（IDF），生成最终的 TF-IDF 矩阵。

最终会生成矩阵结构来存储。

```
            self.tfidf_vectorizer = TfidfVectorizer(
                max_features=config.SEARCH_CONFIG['max_features'],
                stop_words='english',
                lowercase=True,
                ngram_range=(1, 2),
                min_df=2,
                max_df=0.8
            )
            self.tfidf_matrix = self.tfidf_vectorizer.fit_transform(documents)
```

获取关键词相似度时，程序首先会将查询也转换为 TF-IDF 向量，然后计算查询向量与所有文档向量的余弦相似度

```
        def _get_keyword_similarities(self, query: str) -> List[Tuple[int, float]]:
            """获取关键词相似度"""
            query_vector = self.tfidf_vectorizer.transform([query])
            similarities = cosine_similarity(query_vector, self.tfidf_matrix).flatten()
            top_indices = np.argsort(similarities)[::-1]

            return [(int(idx), float(similarities[idx])) for idx in top_indices
                    if similarities[idx] > 0]
```

2、语义检索

（1）Sentence Transformers

Sentence Transformers 是基于预训练 Transformer 模型的句子嵌入生成框架，它能够将可变长度的文本序列映射到固定维度的稠密向量空间中，使得语义相似的文本在向量空间中距离较近。

Transformer 架构通过自注意力机制实现了序列中任意两个位置之间的直接连接。这种设计使得模型能够捕获长距离的依赖关系，并且可以并行处理整个序列。

Sentence Transformers 在标准 Transformer 基础上引入了句子级别池化策略。标准的 Transformer 模型为序列中的每个 token 生成独立的向量表示，但对于文档检索等任务，我们需要的是整个文档或句子的统一表示。池化层实现了将 token 级别的表示聚合为文档级别的表示。

本系统采用的主要策略是均值池化，即将所有 token 向量的对应维度进行平均，因此能够综合考虑文档中所有词汇的贡献，避免因单个词汇的极端值而导致的表示偏差。均值池化保留了更多的整体信息且不依赖于特定的模型架构设计。

（2）all-MiniLM-L6-v2 模型

all-MiniLM-L6-v2 是一个高效的句子编码器模型，该模型专门设计用于将句子和短段落转换为 384 维的稠密向量表示。

all-MiniLM-L6-v2 采用了对比学习（contrastive learning）的训练目标。在训练过程中，模型需要学习识别哪些句子是配对的，哪些是随机组合的。具体来说，给定一个句子，模型要从一组随机采样的其他句子中预测出真正与其配对的句子。这种训练方式使模型的能力与信息检索系统所需的能力相匹配。

在实现语义检索模块时，不仅要保证语义理解的准确性，还要确保系统在实际部署中的稳定性和效率。all-MiniLM-L6-v2 模型基于 MiniLM 架构，是对原始 BERT 模型的知识蒸馏版本，在大幅减少参数量的同时保持了出色的语义理解能力。L6 表示模型有 6 个 Transformer 层，相比 BERT-base 的 12 层减少了一半，这使得模型在推理速度上有显著优势，22.7M 参数量使其在资源受限的环境中仍能高效运行。而 384 维的输出向量既包含了丰富的语义信息，又避免了过高的存储和计算开销，这在效果和效率实现取得了很好的平衡。

模型默认可以处理最长 256 个词片段（word pieces）的输入文本，超过这个长度的文本会被自动截断。电商商品的相关文本往往长度较短，因此这个长度设置可以满足要求。

这个模型可以直接从 hugging face 上导入，也可以将其下载到本地。

本实验将模型文件下载到了/models 目录下，系统采用手动构建的方式来加载模型。如果本地模型不完整，就会使用在线加载：

```
        try:
            # 手动构建模型
            transformer = Transformer(str(local_model_path))
            pooling = Pooling(
                transformer.get_word_embedding_dimension(),
                pooling_mode='mean'
            )
            self.model = SentenceTransformer(modules=[transformer, pooling])
        except Exception:
            # 在线加载
            self.model = SentenceTransformer(model_name)
```

（3）代码实现

模型加载成功后开始构建语义嵌入索引。在项目的/src/search_engine.py 代码的 build_embeddings 方法中，系统将所有商品的搜索文本一次性转换为向量表示：

```
    self.semantic_embeddings = self.model.encode(
        valid_docs,
        batch_size=16,
        show_progress_bar=True,
        convert_to_numpy=True,
        normalize_embeddings=True
    )
```

在 search 方法中。当用户输入查询时，系统首先将查询文本通过同样的模型编码：

```
query_embedding = self.model.encode([query], convert_to_numpy=True, normalize_embeddings=True)
```

然后通过余弦相似度计算查询向量与所有商品向量的相似性：

```
similarities = cosine_similarity(query_embedding, self.semantic_embeddings).flatten()
```

3、混合查询

不同的检索方法具有不同的优势和局限性。传统的 TD-IDF 是基于关键词的检索方法，在处理精确匹配和专业术语检索方面表现优异，但存在无法理解查询的深层语义意图，无法处理同义词、近义词等缺陷。语义检索方法虽然能够捕获查询和文档之间的语义相似性，但可能在精确匹配和对特定术语检索方面不如关键词方法直接有效。

因此我考虑综合两种方法来实现优势互补，通过对两种方法分配不同的相似度权重来计算整体的相似度并进行排序。由于用户对于不同物品的搜索需求不同，本系统**将相似度权重的决定权交给用户**。在网页中，用户可以通过拉动我们设置的比例条来自行决定两种检索方式的权重。比如，当用户希望搜索一些生活化的用品，如“简约的水杯”或“适合旅行穿的外套”，用户就可以把语义检索的权重调高；而假如用户是通过具体的产品名称、品牌、型号等关键词来表达自己的需求，就可以将关键词搜索的权重调高。

当用户调节了计算的权重后，系统会立刻调整显示的商品。这使得用户可以很方便地通过尝试来找到最合适的权重。

系统采用线性加权组合模型，其数学表达为

$$Score\_hybrid(q,d) = \alpha \times Score\_keyword(q,d) + \beta \times Score\_semantic(q,d)$$

其中α和β分别代表关键词检索和语义检索的权重系数，且满足α + β = 1。线性组合可以让用户直观地理解不同权重设置对检索结果的影响，同时根据不同的应用场景和用户偏好灵活调整权重分配。

```
def _execute_hybrid_search(self, query: str, top_k: int, filters: Dict = None):
```

```
        keyword_results = self._get_keyword_similarities(query)
        semantic_results = self._get_semantic_similarities(query)

        # 混合得分计算
        hybrid_scores = {}
        max_results = min(len(self.data), top_k * 3)

        for idx, score in keyword_results[:max_results]:
            hybrid_scores[idx] = score * self.keyword_weight

        for idx, score in semantic_results[:max_results]:
            if idx in hybrid_scores:
                hybrid_scores[idx] += score * self.semantic_weight
            else:
                hybrid_scores[idx] = score * self.semantic_weight
```

三、查询处理

1、拼写纠错模块

真实的用户查询往往包含各种形式的错误，这些错误如果不加处理，会严重影响检索系统的性能。这一模块实现了纠正这些错误的功能，使得检索系统能够在表达上存在拼写偏差的情况下仍能理解用户的真实意图。

该模块分为初始化和搜索两个阶段。在系统初始化的时候，系统会从所有商品数据中提取出所有单词，构建词汇表。在搜索过程中，首先系统会尝试直接搜索，如果结果不好，就会启动拼写纠错模块，使用 Levenshtein 距离算法，并告知用户纠错结果。如果纠错后结果变差，仍使用原始查询结果。

（1）初始化阶段

拼写纠错系统的生命周期始于 SpellCorrector 类的实例化过程。在系统启动时，该类通过初始化方法建立了核心的数据结构：

```
class SpellCorrector:
    def __init__(self, max_distance: int = 2):
        self.max_distance = max_distance
        self.word_freq = Counter()
        self.vocabulary = set()
```

max_distance 参数定义了拼写纠错的容错范围，设置为 2 意味着系统认为用户最多可能犯 2 个字符级别的错误，避免过度纠错。word_freq 使用 Counter 类来维护词频统计，不仅提供了高效的计数功能，也便于后续的候选词排序。vocabulary 采用 set 数据结构，确保了词汇查找的时间复杂度为 O(1)时间复杂度。

【构建词汇表】

当 HybridSearchEngine 初始化时，会调用 build_vocabulary 方法来构建拼写纠错所需的词汇表：

```
def build_vocabulary(self, documents: List[str]):
    for doc in documents:
        words = re.findall(r'\b\w+\b', doc.lower())
        for word in words:
            if len(word) > 2:
                self.word_freq[word] += 1
                self.vocabulary.add(word)

    min_freq = max(1, len(documents) // 1000)
    self.vocabulary = {word for word, freq in self.word_freq.items() if freq >= min_freq}
```

这个词汇表进行了以下三步处理：

① 提取词汇：从每个商品的描述中提取所有单词

② 统计频率：记录每个词出现的次数

③ 过滤低频词：只保留出现频率较高的词作为正确词汇

至此，初始化阶段处理完成，后面是搜索阶段的实现逻辑。

（2）拼写纠错触发机制

当用户发起搜索请求时，系统会先认为用户输入的是正确的，开始进行原始查询。拼写纠错并不会在一开始就执行，而是作为一种智能的后备策略，以避免对正确查询的不必要干预。

```
if use_hybrid and self.semantic_engine.model:
    results = self._execute_hybrid_search(processed_query, top_k, filters)
else:
    results = self._execute_keyword_search(processed_query, top_k, filters)
    query_info['search_strategy'] = 'keyword_only'
```

系统会从三个维度评估搜索结果的质量：结果是否为空、结果数量是否充足（少于期望数量的一半）、以及最高相关度分数是否过低（小于 0.1）。当这些条件中的任何一个成立时，系统才认为原始查询可能存在拼写错误，进而触发纠错流程。

```
if (enable_spell_correction and
        (not results or len(results) < top_k // 2 or
         (results and results[0]['score'] < 0.1))):
```

（3）纠错的主流程

代码首先通过正则表达式将查询分解为独立的词汇单元，然后对每个词汇独立进行纠错处理。对于每个词汇，系统调用 get_candidates 方法来获取可能的纠错候选，然后选择最佳候选作为纠正结果。corrections 列表记录所有的纠错操作，为用户提供纠错信息。

```
def correct_query(self, query: str) -> Tuple[str, List[str]]:
    words = re.findall(r'\b\w+\b', query.lower())
    corrected_words = []
```

```
    corrections = []

    for word in words:
        candidates = self.get_candidates(word)
        if candidates:
            best_word, distance = candidates[0]
            corrected_words.append(best_word)
            if distance > 0:
                corrections.append(f"{word} → {best_word}")
        else:
            corrected_words.append(word)

    corrected_query = ' '.join(corrected_words)
    return corrected_query, corrections
```

（4）候选词生成

如果输入词已存在于词汇表中，直接返回该词本身，编辑距离为 0，避免了对正确词汇的不必要计算。对于不在词汇表中的词汇，系统采用长度预筛选策略，通过比较字符串长度差异来快速排除明显不可能的候选。

只有通过预筛选的词汇才会进入 Levenshtein 距离计算阶段。系统调用 levenshtein_distance 方法来精确计算编辑距离。符合距离要求的候选词会被加入候选列表，然后进行排序。排序策略为(x[1], -self.word_freq.get(x[0], 0))，即编辑距离越小越好，在距离相同时，选择高频词汇。

```
def get_candidates(self, word: str) -> List[Tuple[str, int]]:
    if word in self.vocabulary:
        return [(word, 0)]

    candidates = []
    word_lower = word.lower()
```

```
7.
8.          for vocab_word in self.vocabulary:
9.              if abs(len(vocab_word) - len(word_lower)) > self.max_distance:
10.                 continue
11.
12.             distance = self.levenshtein_distance(word_lower, vocab_word)
13.             if distance <= self.max_distance:
14.                 candidates.append((vocab_word, distance))
15.
16.         candidates.sort(key=lambda x: (x[1], -self.word_freq.get(x[0], 0)))
17.         return candidates[:5]
```

（5）Levenshtein 距离算法

Levenshtein 距离算法是计算两个字符串之间差异程度的经典算法。该算法的核心思想是通过计算将一个字符串转换为另一个字符串所需的最少单字符编辑操作次数来量化两个字符串的相似程度，允许的编辑操作包括字符的插入、删除和替换。

Levenshtein 距离算法的标准实现方法是动态规划解法，其将复杂的字符串比较问题分解为一系列简单的子问题。算法构建一个二维动态规划表，其中每个单元格 dp[i][j]表示将第一个字符串的前 i 个字符转换为第二个字符串的前 j 个字符所需的最少编辑操作数。对于每个位置(i,j)，算法考虑三种可能的操作路径：从位置(i-1,j)进行删除操作、从位置(i,j-1)进行插入操作、从位置(i-1,j-1)进行替换操作（如果字符相同则无需操作）。算法选择这三种路径中成本最低的一种作为当前位置的最优解。这种分解策略使得问题的求解过程变得系统化和可追踪，同时保证了算法的正确性和效率。

/src/search_engine.py 代码的 levenshtein_distance 方法采用了经典的动态规划算法来计算编辑距离。它将一个复杂的字符串匹配问题分解为若干个子问题，并通过状态转移方程来避免重复计算。

```
1.      def levenshtein_distance(self, s1: str, s2: str) -> int:
```

```
        """计算编辑距离"""
        if len(s1) < len(s2):
            return self.levenshtein_distance(s2, s1)

        if len(s2) == 0:
            return len(s1)

        previous_row = list(range(len(s2) + 1))
        for i, c1 in enumerate(s1):
            current_row = [i + 1]
            for j, c2 in enumerate(s2):
                insertions = previous_row[j + 1] + 1
                deletions = current_row[j] + 1
                substitutions = previous_row[j] + (c1 != c2)
                current_row.append(min(insertions, deletions, substitutions))
            previous_row = current_row

        return previous_row[-1]
```

（6）纠错结果质量评估

系统不会盲目采用纠错后的结果，只有当纠错结果的最高分数超过原始结果时才进行替换。

```
    corrected_query, corrections = self.spell_corrector.correct_query(processed_query)
    if corrections:
        if use_hybrid and self.semantic_engine.model:
            corrected_results = self._execute_hybrid_search(corrected_query, top_k, filters)
        else:
            corrected_results = self._execute_keyword_search(corrected_query, top_k, filters)

```

```
        if corrected_results and (not results or corrected_results[0]['score'] > results[0]['score']):
            results = corrected_results
            query_info['processed_query'] = corrected_query
            query_info['corrections'] = corrections
            query_info['search_strategy'] += '_spell_corrected'
```

2、同义词扩展模块

与拼写纠错专注于修正用户的输入错误不同，同义词扩展旨在理解用户查询的语义意图，通过添加相关的同义词来扩大搜索范围，从而发现更多相关的结果。

本项目中的同义词扩展功能通过 SynonymExpander 类实现，采用预定义词典的方式，与拼写纠错形成了互补的查询优化策略。

（1）系统初始化阶段

同义词扩展采用了预定义词典方法。词典被按照商品类别进行组织，涵盖了电商平台中常见的产品类型和描述性词汇。

```
class SynonymExpander:
    def __init__(self):
        self.synonyms = {
            # 电子产品
            'phone': ['smartphone', 'mobile', 'cellphone', 'iphone', 'android'],
            ……
            'large': ['big', 'huge', 'xl'],
        }
```

对于词典，系统在初始化阶段构建了双向的同义词映射关系，解决了同义词关系的对称性问题。例如，当用户搜索’smartphone’时，系统不仅能找到’smartphone’对应的同义词，还能通过 reverse_synonyms 找到将’smartphone’作为同义词的其他词汇。

```
self.reverse_synonyms = {}
for word, syns in self.synonyms.items():
```

```
    for syn in syns:
        if syn not in self.reverse_synonyms:
            self.reverse_synonyms[syn] = []
        self.reverse_synonyms[syn].append(word)
```

（2）触发逻辑

同义词扩展在整个搜索流程中被设计为第三级优化策略，位于原始搜索和拼写纠错之后。只有在前面的策略（原始搜索和拼写纠错）都未能产生满意结果时才会被激活。但在处理逻辑上，同义词扩展采用**结果合并策略**而非纠错模块使用的替换策略，来增加搜索覆盖率。

```
=if (enable_synonym_expansion and
        (not results or len(results) < top_k // 2 or
         (results and results[0]['score'] < 0.1))):

    expanded_query = self.synonym_expander.expand_query(query_info['processed_query'])
    if expanded_query != query_info['processed_query']:
        if use_hybrid and self.semantic_engine.model:
            expanded_results = self._execute_hybrid_search(expanded_query, top_k, filters)
        else:
            expanded_results = self._execute_keyword_search(expanded_query, top_k, filters)

        if expanded_results and (not results or expanded_results[0]['score'] > results[0]['score']
* 0.8):
            combined_results = self._merge_results(results, expanded_results, top_k)
            if len(combined_results) > len(results):
                results = combined_results
                query_info['expansions'] = expanded_query.split()
                query_info['search_strategy'] += '_synonym_expanded'
```

（3）核心逻辑

首先，原始查询被分解为词汇集合，然后，系统对每个词汇进行双向同义词查找：既查找该词的同义词，又查找将该词作为同义词的其他词汇。

```
    def expand_query(self, query: str, max_expansions: int = 2) -> str:
        words = re.findall(r'\b\w+\b', query.lower())
        expanded_terms = set(words)

        for word in words:
            if word in self.synonyms:
                expanded_terms.update(self.synonyms[word][:max_expansions])
            if word in self.reverse_synonyms:
                expanded_terms.update(self.reverse_synonyms[word][:max_expansions])

        return ' '.join(expanded_terms)
```

（4）结果合并策略

系统将比较结果数量是否增加。只有当合并后的结果数量确实超过原有结果时，系统才认为同义词扩展产生了价值。

```
    combined_results = self._merge_results(results, expanded_results, top_k)
    if len(combined_results) > len(results):
        results = combined_results
        query_info['expansions'] = expanded_query.split()
        query_info['search_strategy'] += '_synonym_expanded'

    def _merge_results(self, results1: List[Dict], results2: List[Dict], top_k: int) -> List[Dict]:
        seen_ids = set()
        merged = []

        all_results = results1 + results2
        all_results.sort(key=lambda x: x['score'], reverse=True)
```

```
13.
14.         for result in all_results:
15.             product_id = result['product_id']
16.             if product_id not in seen_ids:
17.                 seen_ids.add(product_id)
18.                 merged.append(result)
19.                 if len(merged) >= top_k:
20.                     break
21.
22.         return merged
```

四、数据集处理

本项目采用的数据集是亚马逊商品数据，该数据集有 2.98k rows，涵盖商品名称、类别、描述、评分、评价等特征。

数据集链接：[randomath/Amazon-combined · Datasets at Hugging Face]

数据处理的代码在/src/data_preprocessor.py。

处理后的数据保存在/data/processed_amazon_data.csv

（1）商品价格数据清洗

原始数据中的价格格式不统一，需要清洗。

```
1.     def clean_price(self, price_str) -> Optional[float]:
2.         if pd.isna(price_str) or price_str is None:
3.             return None
4.
5.         try:
6.             if isinstance(price_str, (int, float)):
7.                 return float(price_str) if price_str > 0 else None
8.
9.             price_str = str(price_str).strip()
```

```
            price_clean = re.sub(r'[^\d.,]', '', price_str)

            if not price_clean:
                return None

            # 处理价格格式
            if ',' in price_clean and '.' in price_clean:
                parts = price_clean.split('.')
                if len(parts) == 2 and len(parts[1]) <= 2:
                    price_clean = price_clean.replace(',', '')
            elif ',' in price_clean:
                if price_clean.count(',') == 1 and len(price_clean.split(',')[1]) <= 2:
                    price_clean = price_clean.replace(',', '.')
                else:
                    price_clean = price_clean.replace(',', '')

            price = float(price_clean)
            return price if 0 < price < 1000000 else None
        except:
            return None
```

首先，方法处理了各种可能的输入类型，包括数值类型、字符串类型以及空值情况。对于字符串类型的价格，系统使用正则表达式移除所有非数字、非逗号、非小数点的字符，以处理带有货币符号、空格等杂质的价格字符串。

价格格式的处理逻辑为，当价格字符串同时包含逗号和小数点时，系统会判断小数点后的位数来确定逗号的作用：如果小数点后只有 1-2 位数字，说明逗号是千位分隔符，应该被移除；否则可能是某种特殊格式。当只有逗号时，系统会根据逗号后的位数判断其是千位分隔符还是小数点的替代符号。最后，系统设置了合理的价格范围（0 到 1,000,000），过滤掉明显不合理的数值。

（2）描述文本清洗

原始描述数据往往包含 HTML 标签、不规范的格式和冗余信息。

```
    def clean_description(self, desc) -> str:
        if pd.isna(desc) or desc is None:
            return ""

        try:
            if isinstance(desc, str):
                try:
                    desc = ast.literal_eval(desc)
                except:
                    pass

            if isinstance(desc, list):
                full_desc = " ".join(str(item) for item in desc if item)
            else:
                full_desc = str(desc)

            # 移除 HTML 标签和多余空格
            clean_desc = re.sub(r'<[^>]+>', '', full_desc)
            clean_desc = re.sub(r'\s+', ' ', clean_desc).strip()
            return clean_desc[:500]  # 限制长度
        except:
            return ""
```

描述字段可能以字符串形式存储列表结构，系统首先尝试使用 ast.literal_eval 来解析这种情况。如果解析成功得到列表，系统会将列表中的所有非空元素连接成完整的描述文本。HTML 标签的移除和空格的标准化均使用正则表达式实现。同时控制长度。

（3）搜索文本的构建

搜索引擎需要一个综合的文本字段来进行全文检索，因此需要将多个字段的信息整合为统一的搜索文本：

```
    def create_search_text(self, row: pd.Series) -> str:
        text_parts = []

        # 标题
        title = row.get('title', '')
        if title and str(title) != 'nan':
            text_parts.append(str(title))

        # 主分类
        main_category = row.get('main_category', '')
        if main_category and str(main_category) != 'nan':
            text_parts.append(str(main_category))

        # 描述
        description = row.get('description_clean', '')
        if description:
            text_parts.append(description)

        # 分类列表
        categories = row.get('categories_clean', [])
        if isinstance(categories, list):
            text_parts.extend(categories)

        return " ".join(text_parts).lower()
```

（4）综合处理流程

```
    def process_data(self, df: pd.DataFrame) -> pd.DataFrame:
        self.logger.info("开始数据预处理...")
        processed_df = df.copy()

        # 清洗价格
        processed_df['price_clean'] = processed_df['price'].apply(self.clean_price)

        # 清洗描述
        processed_df['description_clean'] = processed_df['description'].apply(self.clean_description)

        # 解析分类
        processed_df['categories_clean'] = processed_df['categories'].apply(self.parse_list_field)

        # 创建搜索文本
        processed_df['search_text'] = processed_df.apply(self.create_search_text, axis=1)

        # 添加产品 ID
        processed_df['product_id'] = range(len(processed_df))

        # 过滤无效数据
        processed_df = processed_df[
            processed_df['title'].notna() &
            (processed_df['title'].astype(str) != 'nan') &
            (processed_df['title'].astype(str).str.strip() != '')
            ].copy()

        processed_df = processed_df.reset_index(drop=True)
        processed_df['product_id'] = range(len(processed_df))

```

```
30.         self.logger.info(f"数据预处理完成: {len(processed_df)} 条记录")
31.         return processed_df
```

【实验结果分析】：

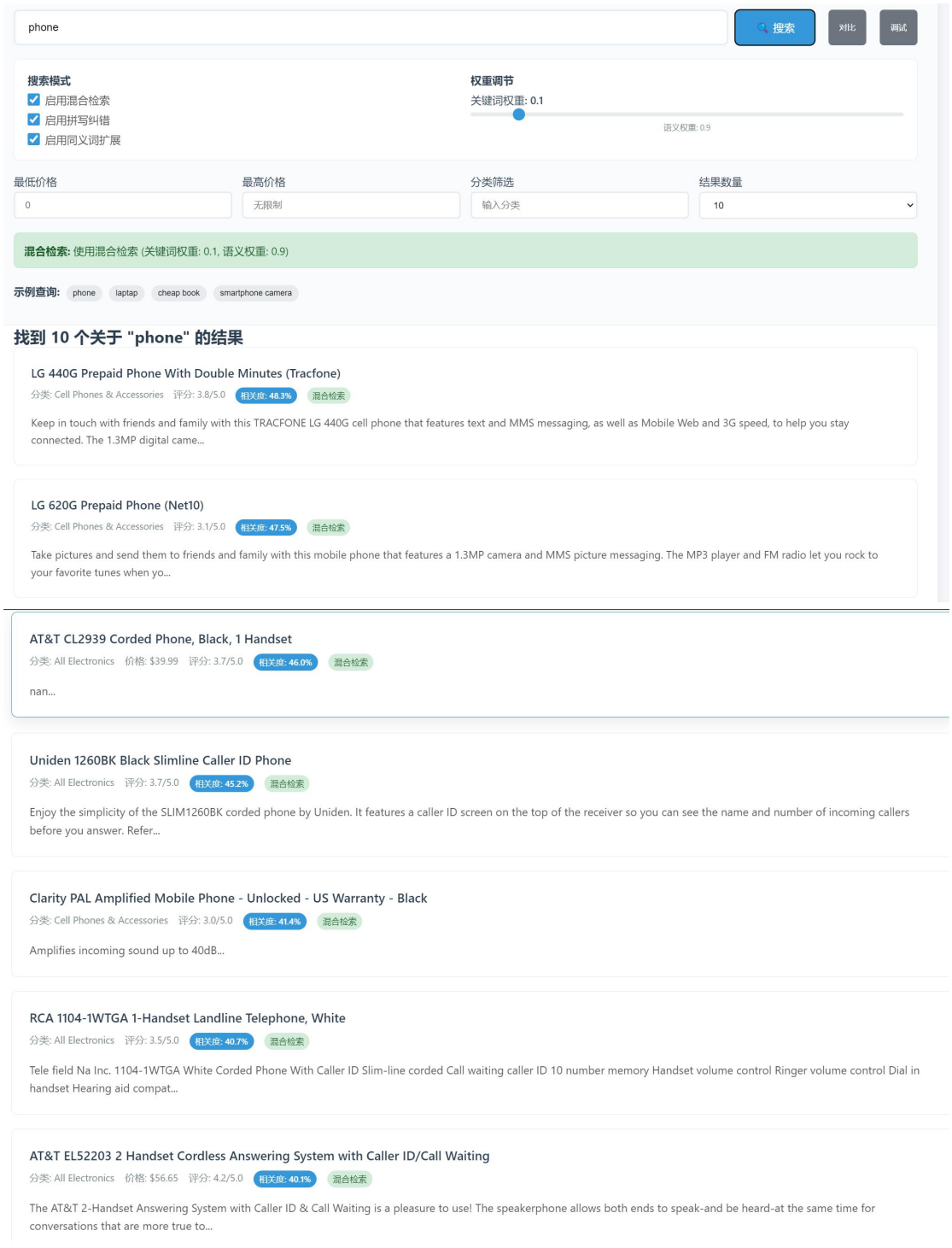
在 python 终端中执行 pip install -r requirements.txt，下载所需依赖。

然后执行 python run.py api ，可以看到模型导入成功，词汇表也成功生成。

```
(ir) PS D:\AIlearning\contentSecurity\informationRetrieval> python run.py api
✅sentence-transformers 导入成功
启动API服务...
INFO:     Will watch for changes in these directories: ['D:\\AIlearning\\contentSecurity\\informationRetrieval']
INFO:     Uvicorn running on http://localhost:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [7948] using WatchFiles
✅sentence-transformers 导入成功
INFO:     Started server process [26148]
INFO:     127.0.0.1:42129 - "OPTIONS /api/search HTTP/1.1" 200 OK
Batches: 100%|██████████| 1/1 [00:00<00:00, 11.46it/s]
INFO:     127.0.0.1:42129 - "POST /api/search HTTP/1.1" 200 OK
Batches: 100%|██████████| 1/1 [00:00<00:00,  4.05it/s]
INFO:     127.0.0.1:42132 - "POST /api/search HTTP/1.1" 200 OK
```

打开 html 网页，即可看到系统成功运行

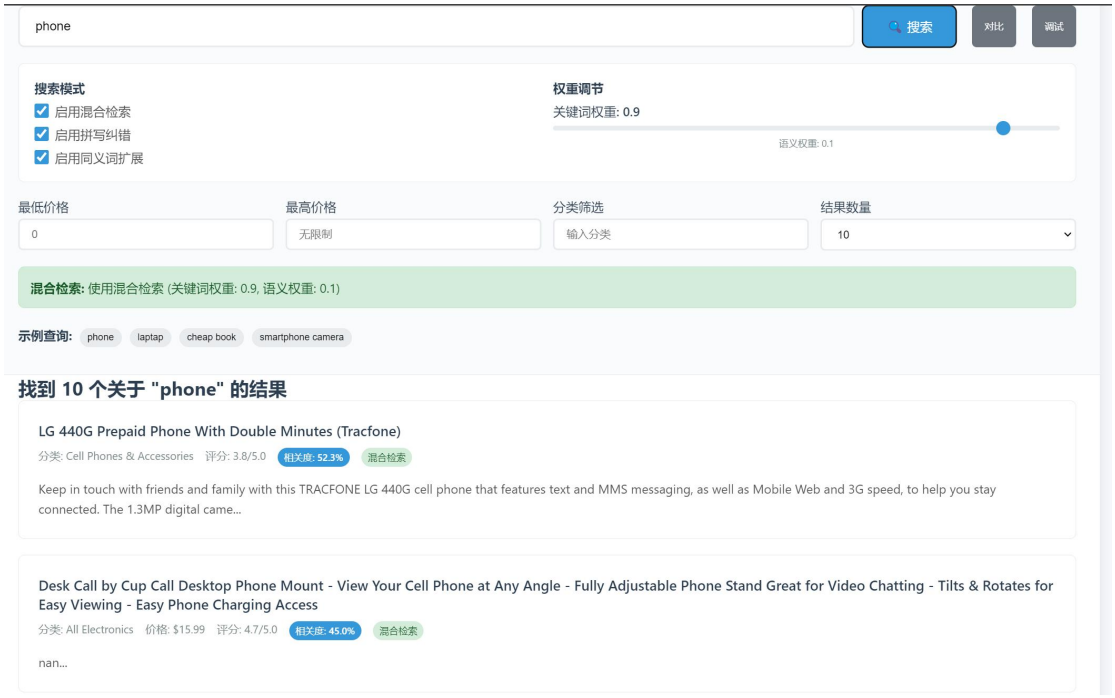
现在，我在搜索框中输入 phone，并设置关键词权重为 0.1 的时候，搜索出来的结果部分截图为：



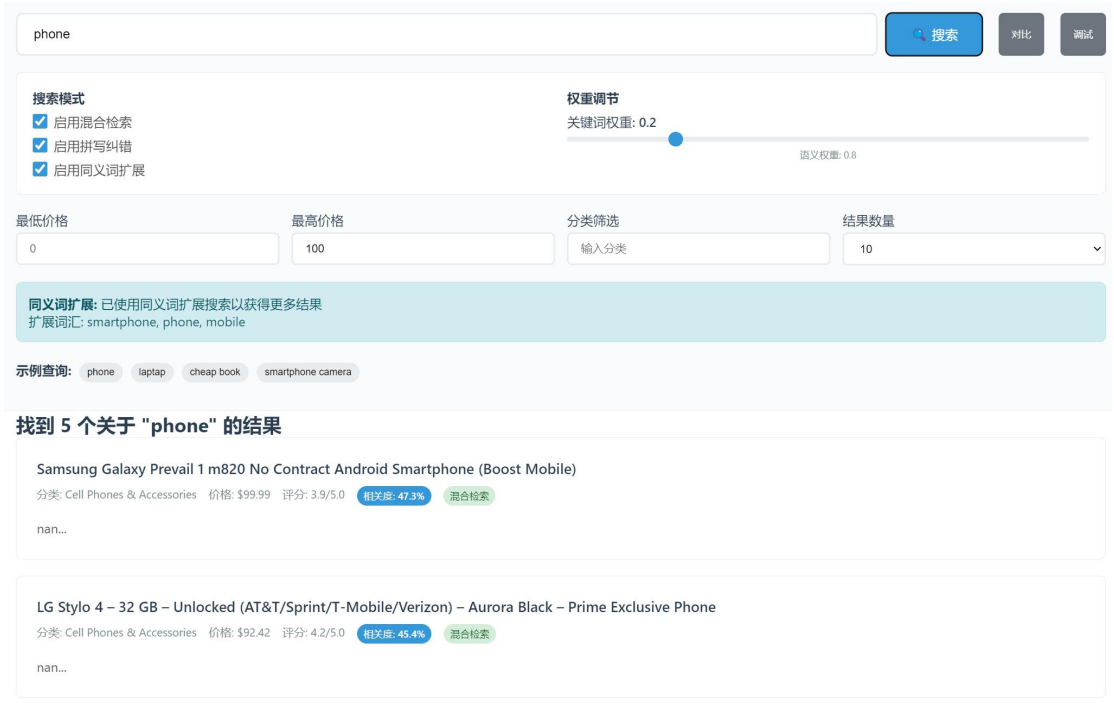
搜索出来的前两个商品均为手机，这表明系统成功理解了我的要求

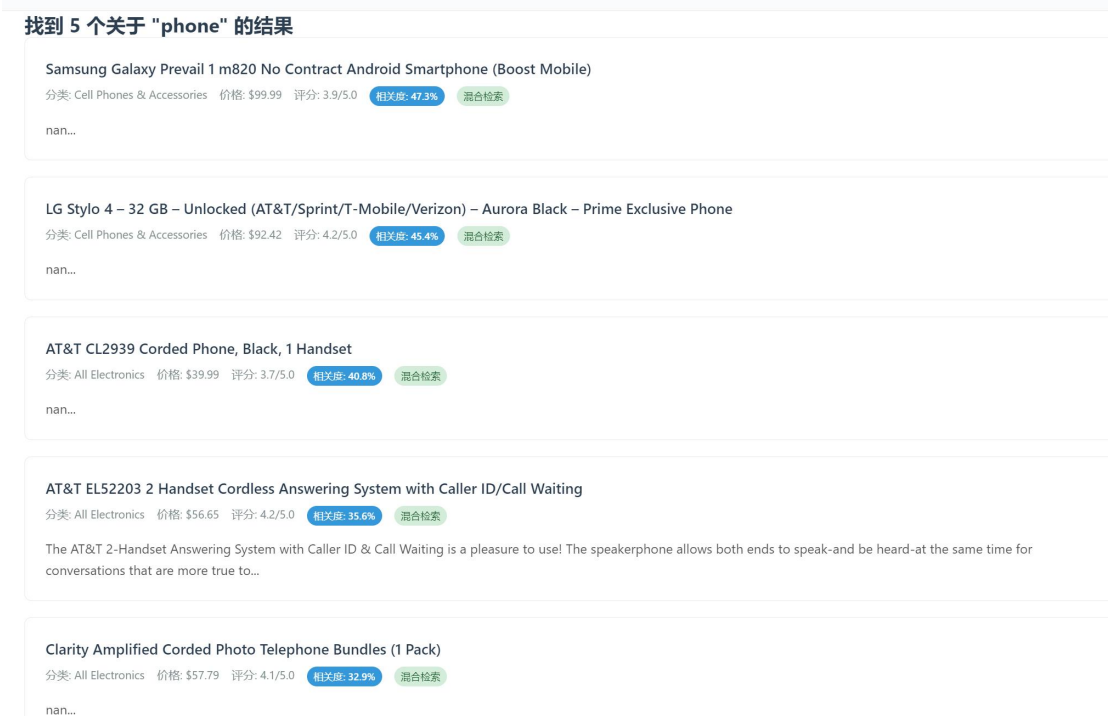
而**当我将关键词权重调为 0.9 时**，可以看到商品的相关度发生了变化，且排名第二的商品变为了一个桌面手机支架。这表明此时关键词搜索占据了主要，对于“phone”的高匹配

度让系统找到了这个手机支架，可是我要搜索的是手机。

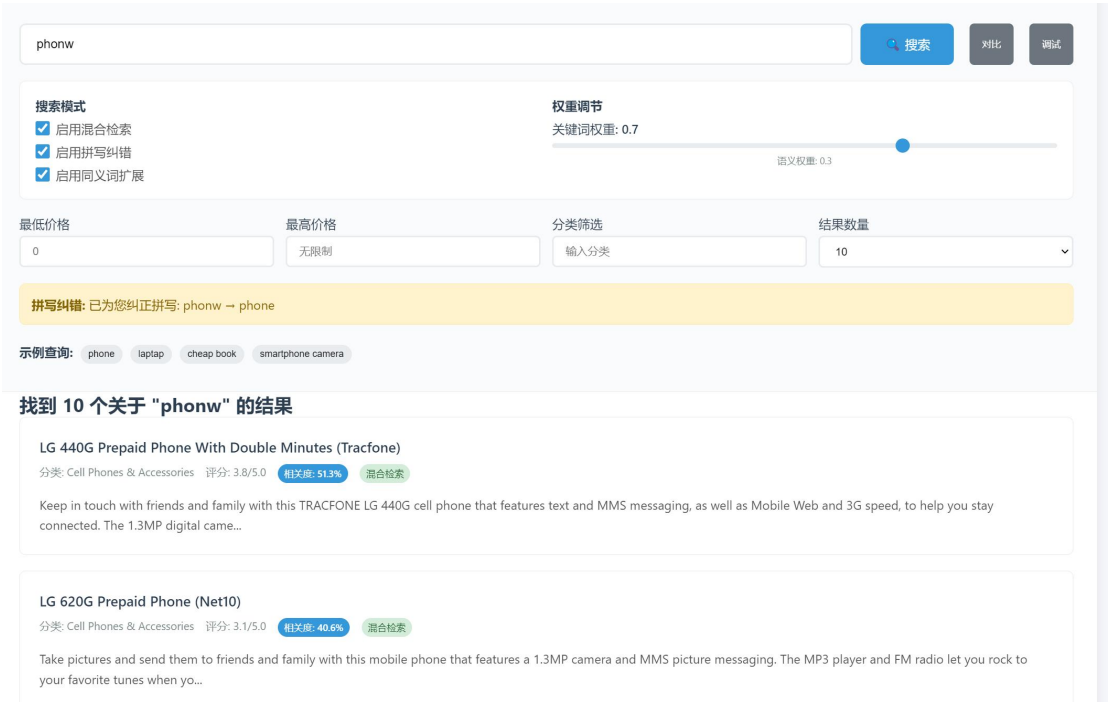


接下来我**将词义理解的权重设置为 0.8**（即关键词权重为 0.2），并设置最高价格为 100。可以看到，手机的价格小于 100 美金的商品很少，因此系统搜索出来的结果的条目不足，触发了**同义词扩展功能**，扩展出了 smartphone、phone、mobile 合并处理





接下来在输入框中输入 phonw 测试**拼写纠错功能**，可以看到已经纠正拼写 phonw 为 phone



【实验总结】：

本次实验中，通过亲身设计一个信息检索系统，让我对现代信息检索系统的核心加数和实现方法有了更为清晰的认识。

在实验中，我首先采用的是 TF-IDF 的传统检索的方式来进行信息的检索，该方法能很好地实现基于关键词的检测，包括对具体的产品名称、品牌、型号等查询。但是在测试的过程中，我发现这种搜索方法不能很好地理解用户的语义，也无法包容用户的拼写错误等。经过进一步的探索，我引入了预训练好的深度学习模型来完善其缺点。同时也加入了拼写纠错和同义词扩展模块，能在各个层面上多样化地满足用户的搜索需求，实现了一个相对完备的电商信息检索系统。

从实验结果来看，混合检索策略在不同类型的查询中都表现良好。对于"phone"这样的通用查询，语义检索权重较高时能够准确返回手机类商品；而关键词权重较高时，虽然也能找到相关商品，但由于关键词检索的局限性，检索出的结果可能会包含一些边缘相关的结果（如手机支架）等。同时，在实验结果模块，我也验证了其他的功能的实现，拼写纠错功能能够成功将"phonw"纠正为"phone"，同义词扩展功能在结果不足时能够扩展查询词汇。

最初在使用 all-MiniLM-L6-v2 模型时，我是直接用的 huggingface 上提供的引用的代码来实时获取模型，但是由于网络不稳定，常常会出现访问不到模型的情况，出现报错：(ProtocolError('Connection aborted.', ConnectionResetError(10054, '远程主机强迫关闭了一个现有的连接。', None, 10054, None)), '(Request ID: 21d59e8e-d9e6-421e-b811-c2a674d52b5f)')

于是我采用了将模型文件下载到本地，尽管可是还是反复失败，查询资料并多次尝试后使用本地的文件并手动组装模型，才成功。这让我对深度学习模型的使用有了更切身的体验，丰富了问题处理的经验。

评语及评分（指导教师）

【评语】：

评分：

日期：

附件：

实验报告说明

1．实验名称：要用最简练的语言反映实验的内容。

2．实验目的：目的要明确，要抓住重点。

3．实验环境：实验用的软硬件环境（配置）。

4．实验方案设计（思路、步骤和方法等）：这是实验报告极其重要的内容。包括概要设计、详细设计和核心算法说明及分析，系统开发工具等。应同时提交程序或设计电子版。

对于**设计型和综合型实验**，在上述内容基础上还应该画出流程图、设计思路和设计方法，再配以相应的文字说明。

对于**创新型实验**，还应注明其创新点、特色。

5．实验结果分析：即根据实验过程中所见到的现象和测得的数据，进行对比分析并做出结论（可以将部分测试结果进行截屏）。

6．实验总结：对本次实验的心得体会，所遇到的问题及解决方法，其他思考和建议。

7．评语及评分：指导教师依据学生的实际报告内容，用简练语言给出本次实验报告的评价和价值。