> 主讲老师：Fox
> 有道云笔记地址：https://note.youdao.com/s/PJRhfowf

1. 相关性的概述

什么是相关性（Relevance）

搜索是用户和搜索引擎的对话，用户关心的是搜索结果的相关性

- 是否可以找到所有相关的内容
- 有多少不相关的内容被返回了
- 文档的打分是否合理
- 结合业务需求，平衡结果排名

相关性是指在搜索引擎中，描述一个文档与查询语句匹配程度的度量标准。这种相关性通过为每个匹配查询条件的文档计算一个相关性评分（_score）来实现，评分越高表示文档与查询语句的匹配程度越高

如下例子：显而易见，查询JAVA多线程设计模式，文档id为2,3的文档的算分更高

关键词	文档ID
JAVA	1,2,3
设计模式	1,2,3,4,5,6
多线程	2,3,7,9

Elasticsearch使用评分算法，根据查询条件与索引文档的匹配程度来确定每个文档的相关性。同时，为了满足各种特定的业务需求，Elasticsearch也充分允许用户自定义评分。

在下面示例中，_score就是Elasticsearch检索返回的评分，其值可以衡量每个文档与查询的匹配程度，即相关性。每个文档都有对应的评分，该得分由正浮点数表示。文档评分越高，则该文档的相关性越高。

计算相关性评分

Elasticsearch使用布尔模型查找匹配文档，并用一个名为“实用评分函数”的公式来计算相关性。这个公式借鉴了TF-IDF（词频-逆向文档频率）和向量空间模型，同时加入了一些现代的新特性，如协调因子、字段长度归一化以及词/查询语句权重提升。

Elasticsearch 5之前的版本，评分机制或者打分模型是基于TF-IDF实现的。从Elasticsearch 5之后，默认的打分机制改成了Okapi BM25。其中BM是Best Match的缩写，25是指经过25次迭代调整之后得出的算法，它是由TF-IDF机制进化来的。

传统TF-IDF和BM25都使用逆向文档频率来区分普通词（不重要）和非普通词（重要），使用词频来衡量某个词在文档中出现的频率。两种机制的逻辑相似：首先，文档里的某个词出现得越频繁，文档与这个词就越相关，得分越高；其次，某个词在集合中所有文档里出现的频次越高，则它的权重越低、得分越低。也就是说，某个词在集合中所有文档里越罕见，其得分越高。BM25在传统TF-IDF的基础上增加了几个可调节的参数，使得它在应用上更佳灵活和强大，具有较高的实用性。

TF-IDF评分公式：

- **TF是词频(Term Frequency)**

检索词在文档中出现的频率越高，相关性也越高。

```
词频（TF） =  某个词在文档中出现的次数 /  文档的总词数
```

- **IDF是逆向文本频率(Inverse Document Frequency)**

每个检索词在索引中出现的频率，频率越高，相关性越低。总文档中有些词比如“是”、“的”、“在”在所有文档中出现频率都很高，并不重要，可以减少多个文档中都频繁出现的词的权重。

```
逆向文本频率（IDF）= log (语料库的文档总数 / (包含该词的文档数+1))
```

- **字段长度归一值（field-length norm)**

检索词出现在一个内容短的 title 要比同样的词出现在一个内容长的 content 字段权重更大。

以上三个因素——词频（term frequency）、逆向文本频率（inverse document frequency）和字段长度归一值（field-length norm）——是在索引时计算并存储的，最后将它们结合在一起计算单个词在特定文档中的权重。

BM25 就是对 TF-IDF 算法的改进，对于 TF-IDF 算法，TF(t) 部分的值越大，整个公式返回的值就会越大。BM25 就针对这点进行来优化，随着TF(t) 的逐步加大，该算法的返回值会趋于一个数值。

- BM 25的公式

示例：通过Explain API查看TF-IDF

```
PUT /test_score/_bulk
{"index":{"_id":1}}
{"content":"we use Elasticsearch to power the search"}
{"index":{"_id":2}}
{"content":"we like elasticsearch"}
{"index":{"_id":3}}
{"content":"Thre scoring of documents is caculated by the scoring formula"}
{"index":{"_id":4}}
{"content":"you know,for search"}

GET /test_score/_search
{
  "explain": true,
  "query": {
    "match": {
      "content": "elasticsearch"
    }
  }
}

GET /test_score/_explain/2
{
  "query": {
    "match": {
      "content": "elasticsearch"
    }
  }
}
```

Elasticsearch自定义评分

自定义评分是用来优化Elasticsearch默认评分算法的一种有效方法，可以更好地满足特定应用场景的需求。

自定义评分的核心是通过修改评分来修改文档相关性，在最前面的位置返回用户最期望的结果。

Elasticsearch自定义评分的主要作用如下：

1) 排序偏好：通过在搜索结果中给每个文档自定义评分，可以更好地满足搜索用户的排序偏好。

2) 特殊字段权重：通过给特定字段赋予更高的权重，可以让这些字段对搜索结果的影响更大。

3) 业务逻辑需求：根据业务需求，可以定义复杂的评分逻辑，使搜索结果更符合业务需求。

4) 自定义用户行为：可以使用用户行为数据（如点击率）作为评分因素，提高用户搜索体验。

搜索结果相关性与自定义评分的关系

搜索引擎本质是一个匹配过程，即从海量的数据中找到匹配用户需求的内容。判定内容与用户查询的相关性一直是搜索引擎领域的核心研究课题之一。如果搜索引擎不能准确地识别用户查询的意图并将相关结果排在前面的位置，那么搜索结果就不能满足用户的需求，从而影响用户对搜索引擎的满意度。

如上图所示，左侧圆圈代表用户期望通过搜索引擎获取的结果，右侧圆圈代表用户最终得到的结果。左右两个圆的交集部分即为预期结果与实际结果的相关性。

2. 自定义评分的策略

然而，如何实现这样的自定义评分策略，以确保搜索结果能够最大限度地满足用户需求呢？我们可以从多个层面，包括索引层面、查询层面以及后处理阶段着手。

以下是几种主要的自定义评分策略：

- Index Boost: 在索引层面修改相关性
- boosting: 修改文档相关性
- negative_boost: 降低相关性
- function_score: 自定义评分
- rescore_query：查询后二次打分

Index Boost: 在索引层面修改相关性

Index Boost这种方式能在跨多个索引搜索时为每个索引配置不同的级别。所以它适用于索引级别调整评分。

实战举例：一批数据里有不同的标签，数据结构一致，要将不同的标签存储到不同的索引(A、B、C)，并严格按照标签来分类展示（先展示A类，然后展示B类，最后展示C类），应该用什么方式查询呢？具体实现如下。借助indices_boost提升索引的权重，让A排在最前，其次是B，最后是C。

```
PUT my_index_100a/_doc/1
{
  "subject":"subject 1"
}
PUT my_index_100b/_doc/1
{
  "subject":"subject 1"
}
PUT my_index_100c/_doc/1
{
  "subject":"subject 1"
}

POST my_index_100*/_search
{
  "query": {
    "term": {
      "subject.keyword": {
        "value": "subject 1"
      }
    }
  }
}

POST my_index_100*/_search
{
  "query": {
    "term": {
      "subject.keyword": {
        "value": "subject 1"
      }
    }
  },
  "indices_boost": [
    {
      "my_index_100a": 1.5
    },
    {
      "my_index_100b": 1.2
```

```
    },
    {
      "my_index_100c": 1
    }
  ]
}
```

boosting: 修改文档相关性

boosting可在查询时修改文档的相关度。boosting值所在范围不同，含义也不同。

若boosting值为0～1，如0.2，代表降低评分；

若boosting值＞1，如1.5，则代表提升评分。

适用于某些特定的查询场景，用户可以自定义修改满足某个查询条件的结果评分。

```
POST /blogs/_bulk
{"index":{"_id":1}}
{"title":"Apple iPad","content":"Apple iPad,Apple iPad"}
{"index":{"_id":2}}
{"title":"Apple iPad,Apple iPad","content":"Apple iPad"}

GET /blogs/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "title": {
              "query": "apple,ipad",
              "boost": 4
            }
          }
        },
        {
          "match": {
            "content": {
              "query": "apple,ipad",
              "boost": 1
            }
          }
        }
      ]
    }
  }
}
```

negative_boost: 降低相关性

若对某些返回结果不满意，但又不想将其排除(must_not)，则可以考虑采用negative_boost的方式。

原理说明如下：

- negative_boost仅对查询中定义为negative的部分生效。
- 计算评分时，不修改boosting部分评分，而negative部分的评分则乘以negative_boost的值。
- negative_boost取值为0～1.0，如0.3。

案例：要求苹果公司的产品信息优先展示

```
POST /news/_bulk
{"index":{"_id":1}}
{"content":"Apple Mac"}
{"index":{"_id":2}}
{"content":"Apple iPad"}
{"index":{"_id":3}}
{"content":"Apple employee like Apple Pie and Apple Juice"}


GET /news/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "content": "apple"
        }
      }
    }
  }
}

# 利用must not排除不是苹果公司产品的文档
GET /news/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "content": "apple"
        }
      },
      "must_not": {
        "match":{
          "content": "pie"
        }
      }
    }
  }
```

```
}
# 利用negative_boost降低相关性
GET /news/_search
{
  "query": {
    "boosting": {
      "positive": {
        "match": {
          "content": "apple"
        }
      },
      "negative": {
        "match": {
          "content": "pie"
        }
      },
      "negative_boost": 0.2
    }
  }
}
```

function_score: 自定义评分

该方式支持用户自定义一个或多个查询语句及脚本，达到精细化控制评分的目的，以对搜索结果进行高度个性化的排序设置。适用于需进行复杂查询的自定义评分业务场景。

案例1：商品信息如下，如何同时根据销量和浏览人数进行相关度提升？

商品	销量	浏览人数
A	10	10
B	20	20

想要提升相关度评分，则将每个文档的原始评分与其销量和浏览人数相结合，得到一个新的评分。例如，使用如下公式：

评分=原始评分×（销量+浏览人数）

这样，销量和浏览人数较高的文档就会有更高的评分，从而在搜索结果中排名更靠前。这种评分方式不仅考虑了文档与查询的匹配度（由_score表示），还考虑了文档的销量和浏览人数，非常适用于电子商务等场景。

该需求可以借助script_score实现，代码如下，其评分是基于原始评分和销量与浏览人数之和的乘积计算的结果。

```
PUT my_index_products/_bulk
{"index":{"_id":1}}
{"name":"A","sales":10,"visitors":10}
{"index":{"_id":2}}
{"name":"B","sales":20,"visitors":20}
{"index":{"_id":3}}
{"name":"C","sales":30,"visitors":30}

#基于function_score实现自定义评分检索
POST my_index_products/_search
{
  "query": {
    "function_score": {
      "query": {
        "match_all": {}
      },
      "script_score": {
        "script": {
          "source": "_score*(doc['sales'].value+doc['visitors'].value)"
        }
      }
    }
  }
}
```

rescore_query：查询后二次打分

二次评分是指重新计算查询所返回的结果文档中指定文档的得分。

Elasticsearch会截取查询返回的前N条结果，并使用预定义的二次评分方法来重新计算其得分。但对全部有序的结果集进行重新排序的话，开销势必很大，使用rescore_query可以只对结果集的子集进行处理。该方式适用于对查询语句的结果不满意，需要重新打分的场景。

```
PUT my_index_books-demo/_bulk
{"index":{"_id":"1"}}
{"title":"ES实战","content":"ES的实战操作，实战要领，实战经验"}
{"index":{"_id":"2"}}
{"title":"MySQL实战","content":"MySQL的实战操作"}
{"index":{"_id":"3"}}
{"title":"MySQL","content":"MySQL一定要会"}

GET my_index_books-demo/_search
{
  "query": {
    "match": {
      "content": "实战"
    }
  }
}

# 查询content字段中包含”实战“的文档，权重为0.7。
# 并对文档中title为MySQL的文档增加评分，权重为1.2，
# window_size为50，表示取分片结果的前50进行重新算分
GET my_index_books-demo/_search
{
  "query": {
    "match": {
      "content": "实战"
    }
  },
  "rescore": {
    "query": {
      "rescore_query": {
        "match": {
          "title": "MySQL"
        }
      },
      "query_weight": 0.7,
      "rescore_query_weight": 1.2
    },
    "window_size": 50
  }
```

```
40 }
```

通过rescore_query我们可以对检索结果进行二次评分，增加自己更复杂的评分逻辑，提供更准确的结果排序，但是相应的也会增加查询的计算成本与响应时间。

多字段搜索场景优化

多字段搜索的三种场景：

- **最佳字段(Best Fields)： 多个字段中返回评分最高的**

当字段之间相互竞争，又相互关联。例如，对于博客的 title和 body这样的字段，评分来自最匹配字段

- **多数字段(Most Fields)：匹配多个字段，返回各个字段评分之和**

处理英文内容时的一种常见的手段是，在主字段(English Analyzer)，抽取词干，加入同义词，以匹配更多的文档。相同的文本，加入子字段（Standard Analyzer），以提供更加精确的匹配。其他字段作为匹配文档提高相关度的信号，匹配字段越多则越好。

- **混合字段(Cross Fields)： 跨字段匹配，待查询内容在多个字段中都显示**

对于某些实体，例如人名，地址，图书信息。需要在多个字段中确定信息，单个字段只能作为整体的一部分。希望在任何这些列出的字段中找到尽可能多的词。

最佳字段搜索

将任何与任一查询匹配的文档作为结果返回，采用字段上最匹配的评分作为最终评分返回。

> 官方文档：https://www.elastic.co/guide/en/elasticsearch/reference/8.14/query-dsl-dis-max-query.html

案例

```
DELETE /blogs
PUT /blogs/_doc/1
{
    "title": "Quick brown rabbits",
    "body":  "Brown rabbits are commonly seen."
}

PUT /blogs/_doc/2
{
    "title": "Keeping pets healthy",
    "body":  "My quick brown fox eats rabbits on a regular basis."
}
# 搜索棕色的狐狸
POST /blogs/_search
{
    "query": {
        "bool": {
            "should": [
                { "match": { "title": "Brown fox" }},
                { "match": { "body":  "Brown fox" }}
            ]
        }
    }
}

```

思考：查询结果不符合预期，为什么？

bool should的算法过程：

- 查询should语句中的两个查询
- 加和两个查询的评分
- 乘以匹配语句的总数
- 除以所有语句的总数

上述例子中，title和body属于竞争关系，不应该将分数简单叠加，而是应该找到单个最佳匹配的字段的评分。

使用dis max query查询

```
POST /blogs/_search
{
    "query": {
        "dis_max": {
            "queries": [
                { "match": { "title": "Brown fox" }},
                { "match": { "body":  "Brown fox" }}
            ]
        }
    }
}
```

可以通过tie_breaker参数调整

Tie Breaker是一个介于0-1之间的浮点数。0代表使用最佳匹配;1代表所有语句同等重要。

1. 获得最佳匹配语句的评分_score 。
2. 将其他匹配语句的评分与tie_breaker相乘
3. 对以上评分求和并规范化

最终得分=最佳匹配字段+其他匹配字段*tie_breaker

```
POST /blogs/_search
{
    "query": {
        "dis_max": {
            "queries": [
                { "match": { "title": "Quick pets" }},
                { "match": { "body":  "Quick pets" }}
            ]
        }
    }
}


POST /blogs/_search
{
    "query": {
        "dis_max": {
            "queries": [
                { "match": { "title": "Quick pets" }},
                { "match": { "body":  "Quick pets" }}
            ],
            "tie_breaker": 0.1
        }
    }
}
```

使用 best_fields 查询

best_fields策略获取最佳匹配字段的得分, final_score = max(其他匹配字段得分， 最佳匹配字段得分)

采用 best_fields 查询，并添加参数 tie_breaker=0.1，final_score = 其他匹配字段得分 * 0.1 + 最佳匹配字段得分

Best Fields是默认类型，可以不用指定，等价于dis_max查询方式

```
POST /blogs/_search
{
  "query": {
    "multi_match": {
      "type": "best_fields",
      "query": "Brown fox",
      "fields": ["title","body"],
      "tie_breaker": 0.2
    }
  }
}
```

案例

```
PUT /employee
{
    "settings" : {
        "index" : {
            "analysis.analyzer.default.type": "ik_max_word"
        }
    }
}

POST /employee/_bulk
{"index":{"_id":1}}
{"empId":"1","name":"员工001","age":20,"sex":"男","mobile":"19000001111","salary":23343,"deptName":"技术部","address":"湖北省武汉市洪山区光谷大厦","content":"i like to write best elasticsearch article"}
{"index":{"_id":2}}
{"empId":"2","name":"员工002","age":25,"sex":"男","mobile":"19000002222","salary":15963,"deptName":"销售部","address":"湖北省武汉市江汉路","content":"i think java is the best programming language"}
{"index":{"_id":3}}
{"empId":"3","name":"员工003","age":30,"sex":"男","mobile":"19000003333","salary":20000,"deptName":"技术部","address":"湖北省武汉市经济开发区","content":"i am only an elasticsearch beginner"}
{"index":{"_id":4}}
{"empId":"4","name":"员工004","age":20,"sex":"女","mobile":"19000004444","salary":15600,"deptName":"销售部","address":"湖北省武汉市沌口开发区","content":"elasticsearch and hadoop are all very good solution, i am a beginner"}
{"index":{"_id":5}}
{"empId":"5","name":"员工005","age":20,"sex":"男","mobile":"19000005555","salary":19665,"deptName":"测试部","address":"湖北省武汉市东湖隧道","content":"spark is best big data solution based on scala, an programming language similar to java"}
{"index":{"_id":6}}
{"empId":"6","name":"员工006","age":30,"sex":"女","mobile":"19000006666","salary":30000,"deptName":"技术部","address":"湖北省武汉市江汉路","content":"i like java developer"}
{"index":{"_id":7}}
{"empId":"7","name":"员工007","age":60,"sex":"女","mobile":"19000007777","salary":52130,"deptName":"测试部","address":"湖北省黄冈市边城区","content":"i like elasticsearch developer"}
{"index":{"_id":8}}
```

```
{"empId":"8","name":"员工008","age":19,"sex":"女","mobile":"19000008888","salary":60000,"deptName":"技术部","address":"湖北省武汉市江汉大学","content":"i like spark language"}
{"index":{"_id":9}}
{"empId":"9","name":"员工009","age":40,"sex":"男","mobile":"19000009999","salary":23000,"deptName":"销售部","address":"河南省郑州市郑州大学","content":"i like java developer"}
{"index":{"_id":10}}
{"empId":"10","name":"张湖北","age":35,"sex":"男","mobile":"19000001010","salary":18000,"deptName":"测试部","address":"湖北省武汉市东湖高新","content":"i like java developer, i also like elasticsearch"}
{"index":{"_id":11}}
{"empId":"11","name":"王河南","age":61,"sex":"男","mobile":"19000001011","salary":10000,"deptName":"销售部","address":"河南省开封市河南大学","content":"i am not like java"}
{"index":{"_id":12}}
{"empId":"12","name":"张大学","age":26,"sex":"女","mobile":"19000001012","salary":11321,"deptName":"测试部","address":"河南省开封市河南大学","content":"i am java developer, java is good"}
{"index":{"_id":13}}
{"empId":"13","name":"李江汉","age":36,"sex":"男","mobile":"19000001013","salary":11215,"deptName":"销售部","address":"河南省郑州市二七区","content":"i like java and java is very best, i like it, do you like java"}
{"index":{"_id":14}}
{"empId":"14","name":"王技术","age":45,"sex":"女","mobile":"19000001014","salary":16222,"deptName":"测试部","address":"河南省郑州市金水区","content":"i like c++"}
{"index":{"_id":15}}
{"empId":"15","name":"张测试","age":18,"sex":"男","mobile":"19000001015","salary":20000,"deptName":"技术部","address":"河南省郑州市高新开发区","content":"i think spark is good"}


GET /employee/_search
{
  "query": {
    "multi_match": {
      "query": "elasticsearch beginner 湖北省 开封市",
      "type": "best_fields",
      "fields": [
        "content",
        "address"
      ]
    }
```

```
  },
  "size": 15
}


# 查看执行计划
GET /employee/_explain/3
{

  "query": {
    "multi_match": {
      "query": "elasticsearch beginner 湖北省 开封市",
      "type": "best_fields",
      "fields": [
        "content",
        "address"
      ]
    }
  }
}

GET /employee/_explain/3
{

  "query": {
    "multi_match": {
      "query": "elasticsearch beginner 湖北省 开封市",
      "type": "best_fields",
      "fields": [
        "content",
        "address"
      ],
      "tie_breaker": 0.1
    }
  }
}

```

使用多数字段搜索

most_fields策略获取全部匹配字段的累计得分（综合全部匹配字段的得分），等价于bool should查询方式

```
GET /employee/_explain/3
{

  "query": {
    "multi_match": {
      "query": "elasticsearch beginner 湖北省 开封市",
      "type": "most_fields",
      "fields": [
        "content",
        "address"
      ]
    }
  }
}
```

案例

```
DELETE /titles
PUT /titles
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "english",
        "fields": {
          "std": {
            "type": "text",
            "analyzer": "standard"
          }
        }
      }
    }
  }
}

POST titles/_bulk
{ "index": { "_id": 1 }}
{ "title": "My dog barks" }
{ "index": { "_id": 2 }}
{ "title": "I see a lot of barking dogs on the road " }

# 结果与预期不匹配
GET /titles/_search
{
  "query": {
    "match": {
      "title": "barking dogs"
    }
  }
}
```

用广度匹配字段title包括尽可能多的文档——以提升召回率——同时又使用字段title.std 作为信号将相关度更高的文档置于结果顶部。

```
GET /titles/_search
{
  "query": {
    "multi_match": {
      "query": "barking dogs",
      "type": "most_fields",
      "fields": [
        "title",
        "title.std"
      ]
    }
  }
}
```

每个字段对于最终评分的贡献可以通过自定义值boost 来控制。比如，使title 字段更为重要,这样同时也降低了其他信号字段的作用：

```
#增加title的权重
GET /titles/_search
{
  "query": {
    "multi_match": {
      "query": "barking dogs",
      "type": "most_fields",
      "fields": [
        "title^10",
        "title.std"
      ]
    }
  }
}
```

跨字段搜索

搜索内容在多个字段中都显示，类似bool+dis_max组合

```
DELETE /address
PUT /address
{
    "settings" : {
        "index" : {
            "analysis.analyzer.default.type": "ik_max_word"
        }
    }
}

PUT /address/_bulk
{ "index": { "_id": "1"} }
{"province": "湖南","city": "长沙"}
{ "index": { "_id": "2"} }
{"province": "湖南","city": "常德"}
{ "index": { "_id": "3"} }
{"province": "广东","city": "广州"}
{ "index": { "_id": "4"} }
{"province": "湖南","city": "邵阳"}

#使用most_fields的方式结果不符合预期，不支持operator
GET /address/_search
{
  "query": {
    "multi_match": {
      "query": "湖南常德",
      "type": "most_fields",
      "fields": ["province","city"]
    }
  }
}

# 可以使用cross_fields，支持operator
#与copy_to相比，其中一个优势就是它可以在搜索时为单个字段提升权重。
GET /address/_search
{
  "query": {
    "multi_match": {
      "query": "湖南常德",
```

```
        "type": "cross_fields",
        "operator": "and",
        "fields": ["province","city"]
      }
    }
}
```

还可以用copy...to 解决，但是需要额外的存储空间

```
DELETE /address
# copy_to参数允许将多个字段的值复制到组字段中，然后可以将其作为单个字段进行查询
PUT /address
{
  "mappings" : {
      "properties" : {
        "province" : {
          "type" : "keyword",
          "copy_to": "full_address"
        },
        "city" : {
          "type" : "text",
          "copy_to": "full_address"
        }
      }
    },
    "settings" : {
        "index" : {
            "analysis.analyzer.default.type": "ik_max_word"
        }
    }
}

PUT /address/_bulk
{ "index": { "_id": "1"} }
{"province": "湖南","city": "长沙"}
{ "index": { "_id": "2"} }
{"province": "湖南","city": "常德"}
{ "index": { "_id": "3"} }
{"province": "广东","city": "广州"}
{ "index": { "_id": "4"} }
{"province": "湖南","city": "邵阳"}

GET /address/_search
{
  "query": {
    "match": {
      "full_address": {
        "query": "湖南常德",
```

```
        "operator": "and"
      }
    }
  }
}

```