近似匹配

使用 TF/IDF 的 准全文 索将文 或者文 中的字段作一大袋的 理。 match 可以告知我 大袋子中是否包含 的 条，但却无法告知 之 的 系。

思考下面 几个句子的不同：

- Sue ate the alligator.
- The alligator ate Sue.
- Sue never goes anywhere without her alligator-skin purse.

用 match 搜索 sue alligator 上面的三个文 都会得到匹配，但它却不能 定 个 是否只来自于一 境，甚至都不能 定是否来自于同一个段落。

理解分 之 的 系是一个 的 ，我 也无法通 一 方式去解决。但我 至少可以通 出 在彼此附近或者 是彼此相 的分 来判断一些似乎相 的分 。

个文 可能都比我 上面 个例子要 ： Sue 和 alligator 个 可能会分散在其他的段落文字中，我 可能会希望得到尽可能包含 个 的文 ，但我 也同 需要 些文 与分 有很高的相 度。

就是短 匹配或者近似匹配的所属 域。

> **TIP** 在 一章 ，我 是使用在match 中使用 的文 作 例子。

短 匹配

就像 match 于 准全文 索是一 最常用的 一 ，当 想 到彼此 近搜索 的 方法 ，就会想到 match_phrase 。

```
GET /my_index/my_type/_search
{
    "query": {
        "match_phrase": {
            "title": "quick brown fox"
        }
    }
}
```

似 match ， match_phrase 首先将 字符串解析成一个 列表，然后 些 行搜索，但只保留那些包含 全部 搜索 ，且 位置 与搜索 相同的文 。 比如 于 quick fox 的短 搜索可能不会匹配到任何文 ，因 没有文 包含的 quick 之后 跟着 fox 。

TIP

match_phrase 同 可写成一 型 phrase 的 match :

```
"match": {
    "title": {
        "query": "quick brown fox",
        "type":  "phrase"
    }
}
```

的位置

当一个字符串被分 后， 个分析器不但会返回一个 列表，而且 会返回各 在原始字符串中的 位置或者 序 系：

```
GET /_analyze?analyzer=standard
Quick brown fox
```

返回信息如下：

```
{
   "tokens": [
      {
         "token": "quick",
         "start_offset": 0,
         "end_offset": 5,
         "type": "<ALPHANUM>",
         "position": 1 ①
      },
      {
         "token": "brown",
         "start_offset": 6,
         "end_offset": 11,
         "type": "<ALPHANUM>",
         "position": 2 ①
      },
      {
         "token": "fox",
         "start_offset": 12,
         "end_offset": 15,
         "type": "<ALPHANUM>",
         "position": 3 ①
      }
   ]
}
```

① position 代表各 在原始字符串中的位置。

位置信息可以被存 在倒排索引中，因此 match_phrase 位置敏感的 ，就可以利用位置信息去匹配包含所有 ，且各 序也与我 搜索指定一致的文 ，中 不 其他 。

什 是短

一个被 定 和短 quick brown fox 匹配的文 ，必 足以下 些要求：

- quick 、 brown 和 fox 需要全部出 在域中。
- brown 的位置 比 quick 的位置大 1 。
- fox 的位置 比 quick 的位置大 2 。

如果以上任何一个 不成立， 文 不能 定 匹配。

TIP

本 上来 ，match_phrase 是利用一 低 的 span 族（query family）去做 位置敏感的匹配。 Span 是一 的 ，所以它 没有分 段；它 只 指定的 行精 搜索。

得 幸的是，match_phrase 已 足 秀，大多数人是不会直接使用 span 。然而，在一些 域，例如 利 索， 是会采用 低 去 行非常具体而又精心造的位置搜索。

混合起来

精 短 匹配 或 是 于 格了。也 我 想要包含 quick brown fox'' 的文 也能 匹配 quick fox,'' ，尽管情形不完全相同。

我 能 通 使用 slop 参数将 活度引入短 匹配中：

```
GET /my_index/my_type/_search
{
    "query": {
        "match_phrase": {
            "title": {
                "query": "quick fox",
                "slop":  1
            }
        }
    }
}
```

slop 参数告 match_phrase 条相隔多 然能将文 匹配 。 相隔多 的意思是 了 和文 匹配 需要移 条多少次？

我 以一个 的例子 始 。 了 quick fox 能匹配一个包含 quick brown fox 的文 ， 我 需要 slop 的 1:

```
           Pos 1         Pos 2         Pos 3
-----------------------------------------------
Doc:       quick         brown         fox
-----------------------------------------------
Query:     quick         fox
Slop 1:    quick                       fox
```

尽管在使用了 slop 短 匹配中所有的 都需要出 , 但是 些 也不必了匹配而按相同的序列排列。 有了足 大的 slop , 就能按照任意 序排列了。

了使 fox quick 匹配我 的文 , 我 需要 slop 的 3:

```
           Pos 1         Pos 2         Pos 3
-----------------------------------------------
Doc:       quick         brown         fox
-----------------------------------------------
Query:     fox           quick
Slop 1:    fox|quick      ①
Slop 2:    quick          fox
Slop 3:    quick                       fox
```

① 注意 fox 和 quick 在 中占据同 的位置。 因此将 fox quick 序成 quick fox 需要 , 或者 2 的 slop 。

多 字段

多 字段使用短 匹配 会 生奇怪的事。 想象一下 索引 个文 :

```
PUT /my_index/groups/1
{
    "names": [ "John Abraham", "Lincoln Smith"]
}
```

然后 行一个 Abraham Lincoln 的短 :

```
GET /my_index/groups/_search
{
    "query": {
        "match_phrase": {
            "names": "Abraham Lincoln"
        }
    }
}
```

令人 的是, 即使 Abraham 和 Lincoln 在 names 数 里属于 个不同的人名, 我 的文 也匹配了 。 一切的原因在Elasticsearch数 的索引方式。

在分析 John Abraham 的 候， 生了如下信息：

- Position 1: john
- Position 2: abraham

然后在分析 Lincoln Smith 的 候， 生了：

- Position 3: lincoln
- Position 4: smith

句 ， Elasticsearch 以上数 分析生成了与分析 个字符串 John Abraham Lincoln Smith 一几乎完全相同的 元。 我 的 示例 相 的 lincoln 和 abraham ， 而且 个 条 存在，并且它 正好相 ，所以 个 匹配了。

幸 的是， 在 的情况下有一 叫做 position_increment_gap 的 的解决方案，它在字段映射中配置。

```
DELETE /my_index/groups/ ①

PUT /my_index/_mapping/groups ②
{
    "properties": {
        "names": {
            "type":                "string",
            "position_increment_gap": 100
        }
    }
}
```

① 首先 除映射 groups 以及 个 型内的所有文 。

② 然后 建一个有正 的新的映射 groups 。

position_increment_gap 置告 Elasticsearch 数 中 个新元素 加当前 条 position 的指定 。所以 在当我 再索引 names 数 ，会 生如下的 果：

- Position 1: john
- Position 2: abraham
- Position 103: lincoln
- Position 104: smith

在我 的短 可能无法匹配 文 因 abraham 和 lincoln 之 的距 100 。 了匹配 个文 必 添加 100 的 slop 。

越近越好

于一个短 排除了不包含 切 短 的文 ， 而 近 — 一个 <code>slop</code> 大于 <code>0</code>— 的短 将 条的 近度考 到最 相 度 <code>_score</code> 中。 通 置一个像 <code>50</code> 或者 <code>100</code> 的高

<code>slop</code> ， 能 排除 距 太 的文 ， 但是也 予了那些 近的的文 更高的分数。

下列 quick dog 的 近 匹配了同 包含 quick 和 dog 的文 ， 但是也 了与 quick 和 dog 更加 近的文 更高的分数 ：

```
POST /my_index/my_type/_search
{
   "query": {
      "match_phrase": {
         "title": {
            "query": "quick dog",
            "slop":  50 ①
         }
      }
   }
}
```

① 注意高 slop 。

```
{
  "hits": [
     {
        "_id":      "3",
        "_score":   0.75, ①
        "_source": {
           "title": "The quick brown fox jumps over the quick dog"
        }
     },
     {
        "_id":      "2",
        "_score":   0.28347334, ②
        "_source": {
           "title": "The quick brown fox jumps over the lazy dog"
        }
     }
  ]
}
```

① 分数 高因 quick 和 dog 很接近

② 分数 低因 quick 和 dog 分

使用 近度提高相 度

然 近 很有用， 但是所有 条都出 在文 的要求 于 格了。 我 [full-text-search] 一章的 [match-precision] 也是同 的 ： 如果七个 条中有六个匹配， 那 个文 用 而言就已 足相 了， 但是 match_phrase 可能会将它排除在外。

相比将使用 近匹配作 要求， 我 可以将它作 信号— 使用， 作 多潜在 中的一个， 会 个文 的最 分 做出 献 (参考 [most-fields])。

上我 想将多个 的分数累 起来意味着我 用 bool 将它 合并。

我 可以将一个 的 match 作 一个 must 子句。 个 将决定 些文 需要被包含到 果集中。 我 可以用 minimum_should_match 参数去除 尾。 然后我 可以以 should 子句的形式添加更多特定 。 一个匹配成功的都会 加匹配文 的相 度。

```
GET /my_index/my_type/_search
{
  "query": {
    "bool": {
      "must": {
        "match": { ①
          "title": {
            "query":                "quick brown fox",
            "minimum_should_match": "30%"
          }
        }
      },
      "should": {
        "match_phrase": { ②
          "title": {
            "query": "quick brown fox",
            "slop":  50
          }
        }
      }
    }
  }
}
```

① must 子句从 果集中包含或者排除文 。

② should 子句 加了匹配到文 的相 度 分。

我 当然可以在 should 子句里面添加其它的 ， 其中 一个 只 某一特定方面的相 度。

性能 化

短 和 近 都比 的 query 代 更高。 一个 match 是看 条是否存在于倒排索引中，而一个 match_phrase 是必 算并比 多个可能重 的位置。

Lucene nightly benchmarks 表明一个 的 term 比一个短 大 快 10 倍，比 近 (有 slop 的短)大 快 20 倍。当然， 个代 指的是在搜索 而不是索引 。

TIP

通常，短　　的　外成本并不像　些数字所暗示的那　　人。事　上，性能上的差距只是明一个　　的　　term　　有多快。　准全文数据的短　　通常在几　秒内完成，因此　　上都是完全可用，即使是在一个繁忙的集群上。

在某些特定病理案例下，短　　可能成本太高了，但比　少　。一个典型例子就是DNA序列，在序列里很多同　的　　在很多位置重　出　。在　里使用高　　slop　　会到　致位置　算大量　加。

那　我　　如何限制短　　和　近近　　的性能消耗　？一　有用的方法是　少需要通　短　　的文　　数。

果集重新　分

在先前的章　中　　，我　　了而使用　近　　来　整相　度，而不是使用它将文　从　果列表中添加或者排除。一个　　可能会匹配成千上万的　果，但我　的用　很可能只　　果的前几　感　趣。

一个　　的　match　　已　通　排序把包含所有含有搜索　条的文　放在　果列表的前面了。事　上，我　只想　　些　部文　重新排序，来　同　匹配了短　　的文　一个　外的相　度升　。

search　API　通　　重新　分　明　支持　功能。重新　分　段支持一个代　更高的　分算法—比如　phrase　—只是　了从　个分片中　得前 K 个　果。然后会根据它　的最新　分 重新排序。

求如下所示：

```
GET /my_index/my_type/_search
{
    "query": {
        "match": {  ①
            "title": {
                "query":                "quick brown fox",
                "minimum_should_match": "30%"
            }
        }
    },
    "rescore": {
        "window_size": 50, ②
        "query": {         ③
            "rescore_query": {
                "match_phrase": {
                    "title": {
                        "query": "quick brown fox",
                        "slop":  50
                    }
                }
            }
        }
    }
}
```

① `match` 决定 些文 将包含在最 果集中，并通 TF/IDF 排序。

② `window_size` 是 一分片 行重新 分的 部文 数量。

③ 目前唯一支持的重新打分算法就是 一个 ，但是以后会有 加更多的算法。

相

短 和 近 都很好用，但 有一个 点。它 于 格了： 了匹配短 ，所有 都必 存在，即使使用了 `slop` 。

用 `slop` 得到的 序的 活性也需要付出代 ，因 失去了 之 的 系。即使可以 `sue` 、`alligator` 和 `ate` 相 出 的文 ，但无法分 是 *Sue ate* 是 *alligator ate* 。

当 相互 合使用的 候，表 的含 比 独使用更 富。 个子句 *I'm not happy I'm working* 和 *I'm happy I'm not working* 包含相同 的 ，也 有相同的 近度，但含 截然不同。

如果索引 而不是索引独立的 ，就能 些 的上下文尽可能多的保留。

句子 `Sue ate the alligator` ，不 要将 一个 （或者 *unigram* ）作 索引

```
["sue", "ate", "the", "alligator"]
```

也要将 个 以及它的 近 作 个 索引：

```
["sue ate", "ate the", "the alligator"]
```

些 （或者 *bigrams* ）被称 *shingles* 。

TIP

Shingles 不限于 ； 也可以索引三个 （ *trigrams* ）：

```
["sue ate the", "ate the alligator"]
```

Trigrams 提供了更高的精度，但是也大大 加了索引中唯一的数量。在大多数情况下，Bigrams 就 了。

当然，只有当用 入的 内容和在原始文 中 序相同 ，shingles 才是有用的； `sue alligator` 的 可能会匹配到 个 ，但是不会匹配任何 shingles 。

幸 的是，用 向于使用和搜索数据相似的 造来表 搜索意 。但 一点很重要：只是索引 bigrams 是不 的；我 然需要 unigrams ，但可以将匹配 bigrams 作 加相 度 分的信号。

生成 Shingles

Shingles 需要在索引 作 分析 程的一部分被 建。我 可以将 unigrams 和 bigrams 都索引到 个字段中， 但将它 分 保存在能被独立 的字段会更清晰。unigrams 字段将 成我 搜索的基部分，而 bigrams 字段用来提高相 度。

首先，我　需要在　建分析器　使用 shingle　　元　　器：

```
DELETE /my_index

PUT /my_index
{
    "settings": {
        "number_of_shards": 1,  ①
        "analysis": {
            "filter": {
                "my_shingle_filter": {
                    "type":             "shingle",
                    "min_shingle_size": 2, ②
                    "max_shingle_size": 2, ②
                    "output_unigrams":  false   ③
                }
            },
            "analyzer": {
                "my_shingle_analyzer": {
                    "type":             "custom",
                    "tokenizer":        "standard",
                    "filter": [
                        "lowercase",
                        "my_shingle_filter" ④
                    ]
                }
            }
        }
    }
}
```

① 参考 [relevance-is-broken] 。

②　　最小/最大的 shingle 大小是 2，所以　　上不需要　置。

③ shingle　　元　　器　　出 unigrams，但是我　想　unigrams 和 bigrams 分　。

④ my_shingle_analyzer 使用我　常　的 my_shingles_filter　　元　　器。

首先，用 analyze API　　下分析器：

```
GET /my_index/_analyze?analyzer=my_shingle_analyzer
Sue ate the alligator
```

果然，我　得到了 3 个　　：

- sue ate
- ate the
- the alligator

在我　可以　　建一个使用新的分析器的字段。

多字段

我 曾 到将 unigrams 和 bigrams 分 索引更清晰，所以 title 字段将 建成一个多字段（参考 [multi-fields] ）：

```
PUT /my_index/_mapping/my_type
{
    "my_type": {
        "properties": {
            "title": {
                "type": "string",
                "fields": {
                    "shingles": {
                        "type":     "string",
                        "analyzer": "my_shingle_analyzer"
                    }
                }
            }
        }
    }
}
```

通 个映射， JSON 文 中的 title 字段将会被以 unigrams (title)和 bigrams (title.shingles)被索引， 意味着可以独立地 些字段。

最后，我 可以索引以下示例文 ：

```
POST /my_index/my_type/_bulk
{ "index": { "_id": 1 }}
{ "title": "Sue ate the alligator" }
{ "index": { "_id": 2 }}
{ "title": "The alligator ate Sue" }
{ "index": { "_id": 3 }}
{ "title": "Sue never goes anywhere without her alligator skin purse" }
```

搜索 Shingles

了理解添加 shingles 字段的好 ， 我 首先来看 The hungry alligator ate Sue 行 match 的 果：

```
GET /my_index/my_type/_search
{
   "query": {
        "match": {
           "title": "the hungry alligator ate sue"
        }
   }
}
```

个　返回了所有的三个文　，但是注意文　1 和 2 有相同的相　度　分因　他　包含了相同的　：

```
{
  "hits": [
     {
        "_id": "1",
        "_score": 0.44273707, ①
        "_source": {
           "title": "Sue ate the alligator"
        }
     },
     {
        "_id": "2",
        "_score": 0.44273707, ①
        "_source": {
           "title": "The alligator ate Sue"
        }
     },
     {
        "_id": "3", ②
        "_score": 0.046571054,
        "_source": {
           "title": "Sue never goes anywhere without her alligator skin purse"
        }
     }
  ]
}
```

① 个文　都包含 the 、 alligator 和 ate ，所以　得相同的　分。

② 我　可以通　置 minimum_should_match 参数排除文　3 ，参考 [match-precision] 。

在在　里添加 shingles 字段。不要忘了在 shingles 字段上的匹配是充当一　信号—　了提高相　度分—所以我　然需要将基本 title 字段包含到　中：

```
GET /my_index/my_type/_search
{
   "query": {
      "bool": {
         "must": {
            "match": {
               "title": "the hungry alligator ate sue"
            }
         },
         "should": {
            "match": {
               "title.shingles": "the hungry alligator ate sue"
            }
         }
      }
   }
}
```

然匹配到了所有的 3 个文 ， 但是文 2 在排到了第一名因 它匹配了 shingled ate sue.

```
{
  "hits": [
     {
        "_id": "2",
        "_score": 0.4883322,
        "_source": {
           "title": "The alligator ate Sue"
        }
     },
     {
        "_id": "1",
        "_score": 0.13422975,
        "_source": {
           "title": "Sue ate the alligator"
        }
     },
     {
        "_id": "3",
        "_score": 0.014119488,
        "_source": {
           "title": "Sue never goes anywhere without her alligator skin purse"
        }
     }
  ]
}
```

即使 包含的 hungry 没有在任何文 中出 ， 我 然使用 近度返回了最相 的文 。

Performance性能

shingles 不 比短 更 活，而且性能也更好。 shingles 跟一个 的 `match` 一 高效，而不用 次搜索花 短 的代 。只是在索引期 因 更多 需要被索引会付出一些小的代 ， 也意味着有 shingles 的字段会占用更多的磁 空 。 然而，大多数 用写入一次而 取多次，所以在索引期 化我 的 速度是有意 的。

是一个在 Elasticsearch 里会 常 到的 ：不需要任何前期 行 多的 置，就能 在搜索的 候有很好的效果。 一旦更清晰的理解了自己的需求，就能在索引 通 正 的 的数据建模 得更好 果和性能。