



信息检索实验报告

Shandong University
July 13, 2020

Gao Dechen

Contents

1	实验概述	3
1.1	实验描述	3
1.2	实验环境	3
2	实验设计	4
2.1	数据量分析	4
2.2	Two-stage Scheme	5
2.3	Ensemble Learning	5
2.3.1	Base Learner	5
2.3.2	Ensemble Modeling	6
3	数据预处理	7
3.1	文本预处理	7
3.1.1	法语字母	7
3.1.2	数字处理	8
3.1.3	特殊字符处理	9
3.1.4	Lemmatization	9
3.2	进程池优化	9
3.3	负例数据集导出	11
3.3.1	负例比例设定	11
3.3.2	数据集格式	12
4	特征工程	12
4.1	Corpus 初始化	12
4.2	特征运算	13
5	Recall	16
6	Ensemble Learning	17
6.1	MRR 验证策略	17
6.2	XGBoost Model	19
6.2.1	Grid Search	19

6.2.2	Optimized Parameters	21
6.3	LightGBM Model	21
6.3.1	Optimized Parameters	21
6.3.2	Feature Importances	22
6.4	CatBoost	23
6.4.1	Visual Tools	23
6.4.2	Bayesian Optimization	23
6.5	Blending	25
7	实验总结	26

1 实验概述

1.1 实验描述

给定一个文档集 $D = \{d_1, d_2, d_3 \cdots d_n\}$ 和一个查询 q ，输出 q 对文档集 D 的全排序结果。

- 文档集： $D = \{d_1, d_2, \cdots, d_n\}$ ，其中 d_i 为字符串格式原文档
- 训练集、验证集： $Q = \{q_1, q_2, \cdots, q_n\}$ ，对字符串格式查询 q_i ，数据集中包含 q_i 对应的召回文档集排序结果 $Label_i$

1.2 实验环境

- 调试环境主要采用 iPython，优势在于数据预处理、模型缓存在调试过程中可以驻留内存，提高开发效率。
- 早期调试使用 Google Colaboratory，原因是线上使用便捷，外网服务器下载模型、数据速度快。
- 后期由于大量的数据预处理实验，换用华为云服务器。此时 iPython 工具采用 Jupyter Notebook。
- 考虑到实验模型训练耗时少，而数据预处理、特征工程耗时多。故而运算均未使用 GPU，而是使用多线程处理器。

调试环境	Jupyter Notebook
	Google Colaboratory
解释器	Python 3.6
编辑器	Visual Studio Code

表 1: 软件环境

	CPU	RAM	OS
Google Colab	2 * 2.30 GHz	12.72 GB	Ubuntu 18.04
Huawei Cloud	8 * 3.0 GHz	64.0 GB	Ubuntu 18.04
Local Host	4 * 2.80 GHz	16.0 GB	Windows 10

表 2: 硬件环境

2 实验设计

2.1 数据量分析

进行实验设计之前，首先对数据进行大致分析，以确定所采用的方法。

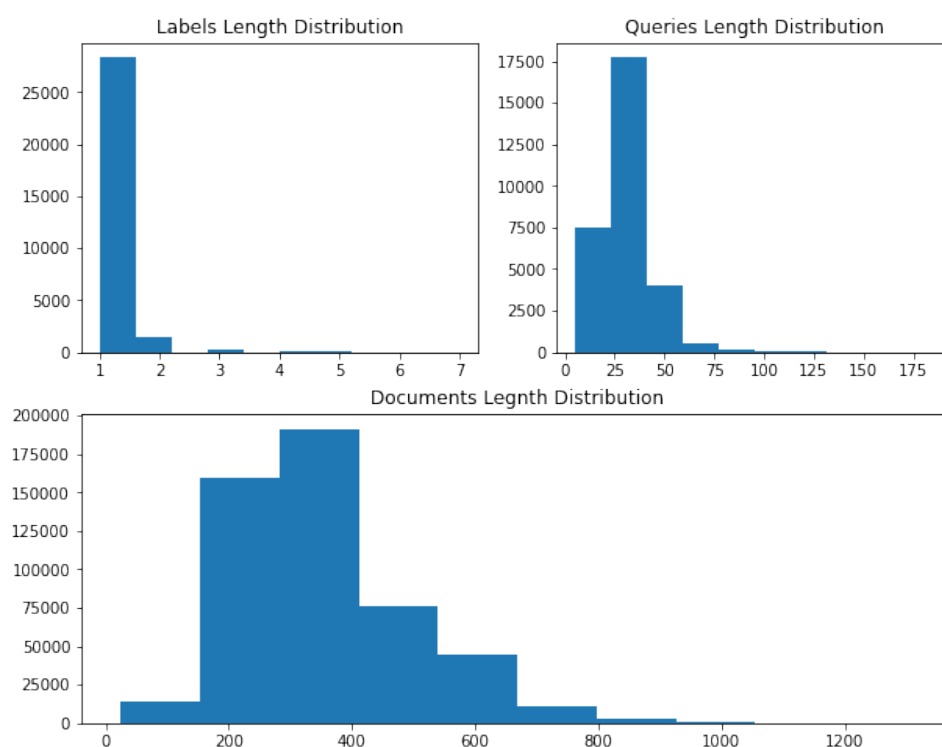


图 1: 数据集统计

如图 1所示，我们分别统计出关于训练集标注文档数量、查询长度以及文档集文档长度的分布情况。

文档集包含 500000 条文档，其中最长的文档仅为 1312；而 30000 条训练集中最长的查询为 186。此外，训练集当中召回文档最多仅 7 条，而多数召回列表长度为 1。

上述数据表明：

- 数据量小，训练集正例数量，模型调试时应注意避免过拟合，并且设置适当的负例
- 文本长度短，意味着可以在有限的运算资源内，考虑更为复杂的特征算法

2.2 Two-stage Scheme

已知最终测试集查询数量为 600，结合章节 2.1 所述，如果直接采用特征工程以及排序策略， 3×10^9 的特征运算量以及 5×10^5 大小的模型输入显然务所接受。

如图 2 所示，我们采用经典的 Two-stage Scheme：

- Recall 阶段通过 BM25、Cosine 等运算快的特征召回 10^2 级别的文档（具体的召回数量设置接下来会在章节 5 当中说明）。
- Rank 阶段将 Recall 阶段召回的文档集和，进一步排序。此时文档集规模此时允许引入更多特征运算。由于本次实验采用的指标为 $MRR@10$ ，故而 Re-rank 输出只取前 10 项。

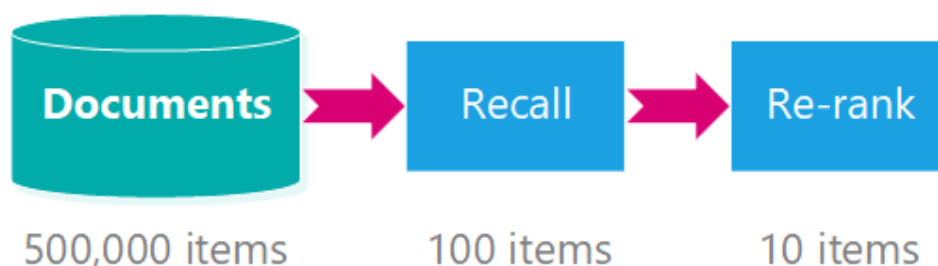


图 2: Two-stage Scheme

2.3 Ensemble Learning

2.3.1 Base Learner

如图 3 所示，目前在 Web30K 以及 Yahoo! 上取得 SoTA 指标的模型均为 LambdaMART。而且目前 LightGBM 文档、社区质量高。故而可以作为较强的 Base Learner candidate。

除此之外，经常用于数据挖掘比赛的 CatBoost、XGBoost 等同样可以作为 Base Learner 进行实验。

考虑到实验数据集给出了 raw text，故而特征工程可以引入语言模型。语言模型的引入方式有两种：一类是利用语言模型生成的词向量；二是将语言模型利用下游任务进行 fine-tune。

目前 Transformer 在 NLP 领域的多项任务上取得 SoTA 水平，多项工作证明 BERT 在 Ranking 问题上的有效性 [5][4][3]。尽管 BERT 生成的词向量作为特征工程输入的方式有效，但是 fine-tune 的效果则更优秀 [5]。

但是技术处于 SoTA 水准并不意味着适用于我们实验的问题，考虑到 LightGBM 依旧处于 Ranking 问题当中的经典解决方案，且 BERT 作为 Base Learner 带来的指标提升，较之时间成本并不合理，我还是选择优先调试集成其他技术方案。

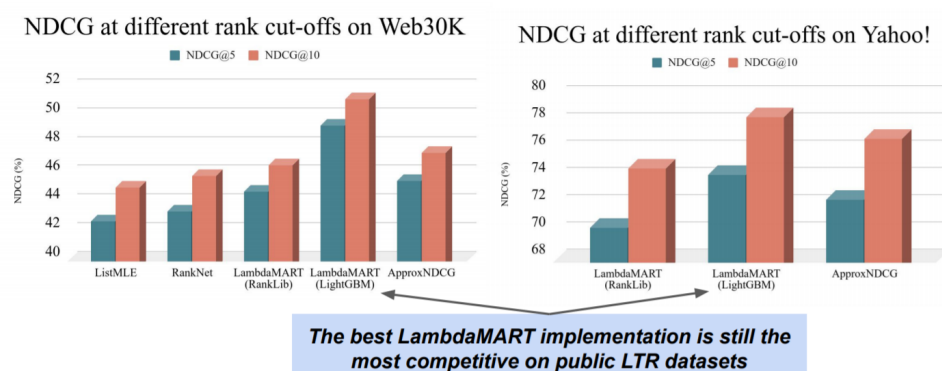


图 3: Ranking 技术

2.3.2 Ensemble Modeling

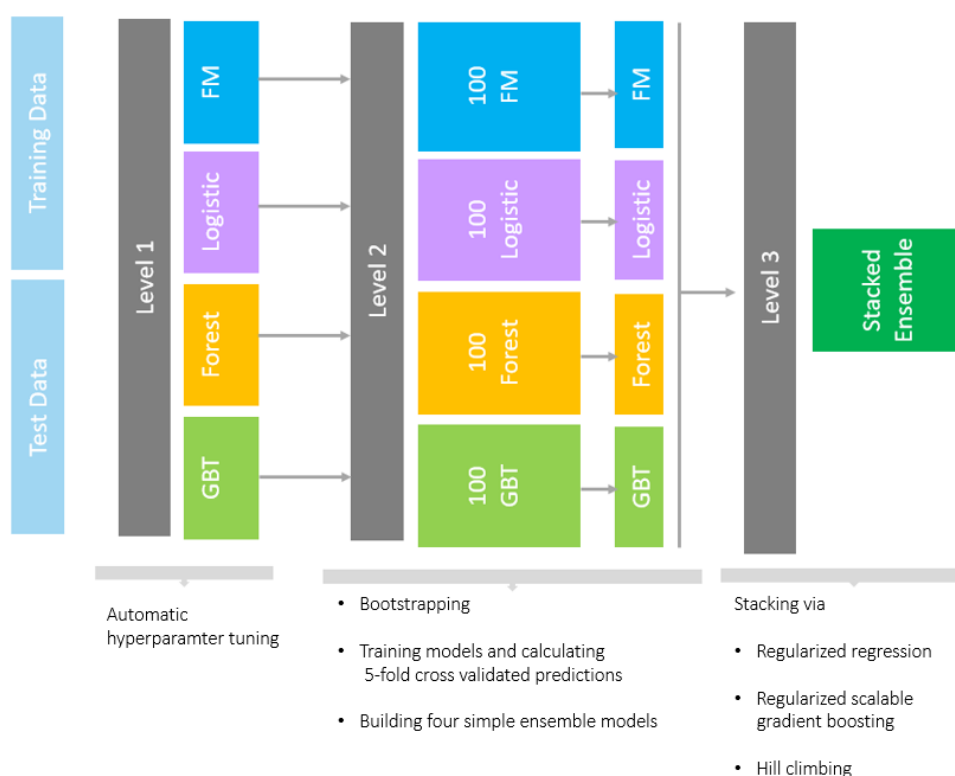


图 4: Model Stacking

目前 Ensemble Learning 方法主要包括 Bagging、Boosting、Stacking、Blending。其中 Stacking、Blending 属于最常用且效果较好的方案。都可以为 Base Learner 集成效果带来提升，尽管有一定差异，但并无绝对的优劣之分，故而我们进行对比试验。

Stacking 技术如图 4 所示。

Level 1、Level 2 分别通过 k-Fold Cross-validation 调参、训练不同的 Base Learner，多个 Base Learner 在 validation fold 数据上的输出构成 Level 3 Stacking 步骤的 feature，而不同模型在原测试数据上的输出则取平均构成 Level 3 当中进行拟合的 label。

3 数据预处理

3.1 文本预处理

文本预处理包括：tokenization、stemming、stop words deletion、special characters deletion、lowering。最终封装预处理接口如下，代码详见 preprocess.py。

```
def preprocess(self, text):  
    '''  
        Preprocess text  
    '''  
    token_words = self.tokenize(text)  
    token_words = self.stem(token_words)  
    token_words = self.delete_stopwords(token_words)  
    token_words = self.delete_characters(token_words)  
    token_words = self.to_lower(token_words)  
    return token_words
```

接下来讨论预处理过程中考虑到的细节问题。部分细节的优化其实对最终指标影响较小，但是我们也应当对细节多加完善。

3.1.1 法语字母

如图 5 所示，数据集当中出现了一定量的法语字母。但是由于法语字母可以使用英语字母代替，所以查询、文档会产生同单词却不匹配的情况。我们通过 unidecode 来同一转换为英文字母编码来解决这一问题。


```
'16': 'The approach is based on a theory of justice that considers
ate. Restorative justice that fosters dialogue between victim and o
'49': 'Colorâ\x80\x94urine can be a variety of colors, most often
e colors can be the result of a disease process, several medication
s.',
'60': 'Inborn errors of bile acid synthesis can produce life-threa
ical disease presenting later in childhood or in adult life.he neur
s). The most useful screening test for many of these disorders is a
electrospray ionisation tandem mass spectrometry.',
'389': 'The word convict here (elegcw /elegxo) means to bring to l
of believing or doing something wrong, but it does not mean that th
similar.',
'616': 'In-home tutors can earn anywhere from $10 to $80 an hour,
\x99s experience. Tutors often charge more for older students or th
'723': 'Calculators may be used on the COMPASS Pre-gebra, Algebr
ed below. Electronic writing pads or pen-input devicesâ\x80\x94The
```

图 5: French Characters

3.1.2 数字处理

是否删除数字？如何删除？

考虑到数字既可以作为文本语义的一部分影响检索结果，也可以作为字符串的一部分。在这里我简单的选择了保留全部数字。

在过滤数字的实验版本中，出现的一类错误是，由于我将字符串转换为 Python 内部的数字数据格式，所以会把 `infinity` 这样的字符串同样判别为数字。例如查询文本出现了 "what is infinity"，结果经过分词、删除停用词、删除数字，该 query 最终为空串。所以如果需要过滤数字，采用正则表达式。预处理如图 6 所示。

```
training_pd.loc[26755]
```

Last executed at 2020-06-16 10:47:00 in 4ms

```
Unnamed: 0          26755
query_id          759040
query_text        infinity
query_label        532706
Name: 26755, dtype: object
```

图 6: Query 26755

3.1.3 特殊字符处理

是否删除特殊字符，以及如何删除？

documents 中出现了"____" 这样无意义的特殊字符，这样全部由特殊字符组成的特殊字符串会继续出现在分词结果当中，但是这种字符对语义理解没有任何作用，应当删除。

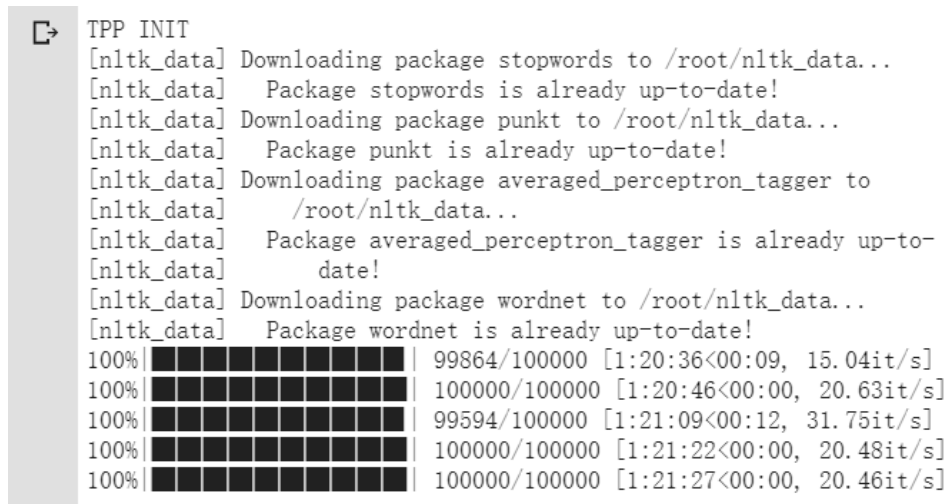
对于"fine-tune" 这样的专用字符串，其中的特殊字符是有意义的，所以不该删除。故而目前我采用的原则是，分词之后只去除“只包含特殊字符”的字符串。

3.1.4 Lemmatization

Lemmatization 是否适用于 Ranking 问题？答案是肯定的。因为理想的检索不应当被词型限制，而是着重于语义。

3.2 进程池优化

如 Colab CPU 性能仅为 $2 * 2.30\text{GHz}$ ，在处理 50 万条文档时，文本预处理极为耗时，如图 7所示，例如 5 线程耗时约 80 分钟。



```
TPP INIT
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data] /root/nltk_data...
[nltk_data] Package averaged_perceptron_tagger is already up-to-
[nltk_data] date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Package wordnet is already up-to-date!
100% [Progress Bar] 99864/100000 [1:20:36<00:09, 15.04it/s]
100% [Progress Bar] 100000/100000 [1:20:46<00:00, 20.63it/s]
100% [Progress Bar] 99594/100000 [1:21:09<00:12, 31.75it/s]
100% [Progress Bar] 100000/100000 [1:21:22<00:00, 20.48it/s]
100% [Progress Bar] 100000/100000 [1:21:27<00:00, 20.46it/s]
```

图 7: Colab Text Preprocessing

如图 8所示，换用 $8 * 3.0\text{GHz}$ 之后，速度提升约 10 倍。

```

Last executed at 2020-06-16 10:29:49 in 8m 15.65s

[nltk_data] /root/nltk_data...
[nltk_data] Package averaged_perceptron_tagger is already up-to-
[nltk_data] date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Package wordnet is already up-to-date!
100%|██████████| 62500/62500 [07:54<00:00, 131.62it/s]
100%|██████████| 62500/62500 [07:53<00:00, 131.90it/s]
100%|██████████| 62500/62500 [07:55<00:00, 131.41it/s]
100%|██████████| 62500/62500 [07:55<00:00, 131.46it/s]
100%|██████████| 62500/62500 [07:54<00:00, 131.76it/s]
100%|██████████| 62500/62500 [07:55<00:00, 131.51it/s]
100%|██████████| 62500/62500 [08:00<00:00, 130.06it/s]
100%|██████████| 62500/62500 [08:05<00:00, 128.76it/s]

```

图 8: 8 CPU Cores Preprocessing

我们使用 Pool 来进行多进程优化。首先编写单进程 worker，此回调函数可以进行复用，对于任何多进程处理任务都可以采取此方案。

```

from multiprocessing import Manager, Process, Pool

def func(proc_idx, data_list, start_idx, end_idx, ret_dict):
    doc_pd = pd.DataFrame({}, columns=['doc_id', 'doc_text'])
    pd_idx = 0
    for (doc_id, doc_text) in tqdm(data_list[start_idx:end_idx]):
        doc_pd.loc[pd_idx, 'doc_id'] = doc_id
        doc_pd.loc[pd_idx, 'doc_text'] = '
        - '.join(tpp.preprocess(unidecode(doc_text)))
        pd_idx = pd_idx + 1
    ret_dict[proc_idx] = doc_pd

```

多进程入口即构造现成池 Pool，以及进程共享通信的数据结构。接下来根据 CPU 内核数量，通过 apply_async 创建异步非阻塞的多进程实例。最终组合各进程的返回值，得到数据预处理的结果。

```

# Make pool according to number of CPU cores
pool = Pool(proc_num)
# Processes shared data

```

```

manager = Manager()
ret_dict = manager.dict()

for proc_idx in range(proc_num):
    start_idx = proc_idx * chunk_size
    end_idx = min(data_list_len, start_idx + chunk_size)
    pool.apply_async(func, args=(proc_idx, data_list, start_idx, end_idx,
        ret_dict))

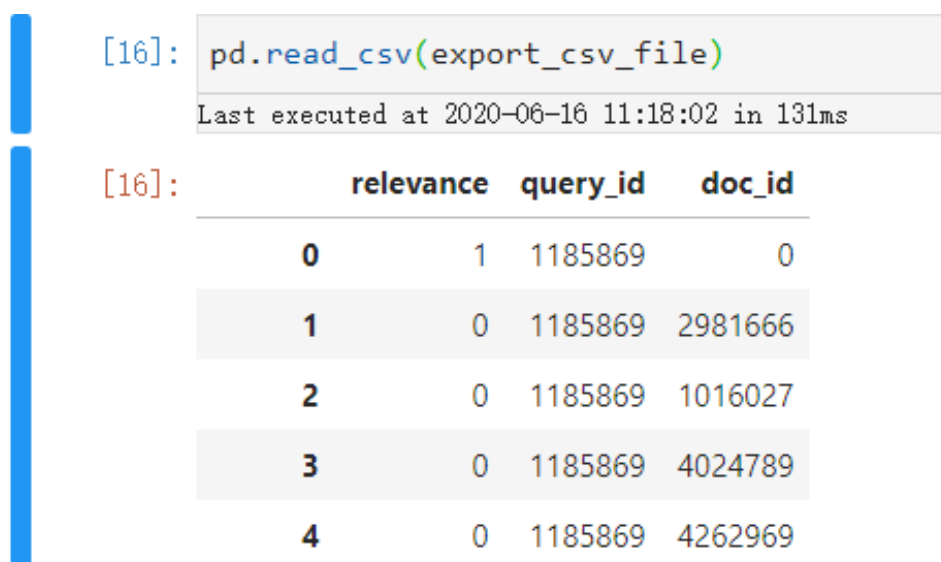
pool.close()
pool.join()
# Concat results from each process
doc_frame = pd.concat([ret_dict[i] for i in range(proc_num)], axis=0)

```

3.3 负例数据集导出

3.3.1 负例比例设定

负例过少，或者过多，都会影响 Re-rank 阶段的排序效果。这里我们引入 NEGATIVE_SCALE 常数，表明负例数量与正例的比例。



```

[16]: pd.read_csv(export_csv_file)
Last executed at 2020-06-16 11:18:02 in 131ms

```

	relevance	query_id	doc_id
0	1	1185869	0
1	0	1185869	2981666
2	0	1185869	1016027
3	0	1185869	4024789
4	0	1185869	4262969

图 9: Exported Dataset

章节 2.1 中已经表明，每组查询正例数量多为 1，最多为 7。而章节 2.2 当中，我

们设定的 Recall 阶段召回量为 10^2 数量级, 那么根据最后的实际召回用于 Re-rank 的数量, 如 50、100, 我们便分别构造 NEGATIVE_SCALE 约为 50、100 的负例数据集进行对照试验。

3.3.2 数据集格式

导出数据格式如图 9 所示, 为 NEGATIVE_SCALE=30 时的验证集, DataFrame 每一项为三元组:

$$(query\ id, doc\ id, relevance)$$

其中 $relevance=0$ 时, 表示该 query-doc pair 为负例; 若非零, 则代表相关度。

注意, 此处的相关度只在同一个 query group 当中才有意义, 因为该值不表示绝对的, 只隐含同一个 query group 下的偏序关系。例如对同一项 query 而言, $relevance=7$ 的文档, 相关度一定大于 $relevance=6$ 的文档。

实验开始时我曾经错误的理解了 relevance 项, 最终在 SVM-rank Documentation[1] 当中看到 "the target values are used to generated pairwise preference constraints"。

根据章节 2.1 所述, relevance 的最大值即为 7, 我们将按照 label 当中文档的出现顺序, 为每一个 query-doc pair 分配降序 relevance; 负例 relevance 分配为 0。

接下来经过章节 4, 导出数据将会扩展如下, 即最终用于模型训练的数据格式:

$$(query\ id, doc\ id, relevance, feat_1, feat_2, \dots, feat_k)$$

其中 $feat_i$ 代表特征工程取得的第 i 项特征。

4 特征工程

4.1 Corpus 初始化

我们使用 gensim 来实现 Word2Vec、BM25、TF-IDF。首先针对文档集简历语料库以及词向量的索引, 写入磁盘, 用于之后的复用。

```
# Initialize corpus and models
dictionary = corpora.Dictionary(text_pool)
corpus = [dictionary.doc2bow(line) for line in text_pool]
tfidf_model = models.TfidfModel(corpus, dictionary=dictionary)
corpus_tfidf = tfidf_model[corpus]

# Write to disk
dictionary.save(dict_path + 'dictionary.dict')
tfidf_model.save(dict_path + 'tfidf.model')
corpora.MmCorpus.serialize(dict_path + 'corpus.mm', corpus)
num_features = len(dictionary.token2id.keys())

# Similarities of sparse matrix
index = similarities.SparseMatrixSimilarity(corpus_tfidf,
    num_features=num_features)
index.save(dict_path + 'index.index')
```

4.2 特征运算

如图 10所示，为特征工程初步选用的特征计算部分。由于章节 3.3.2目前只包含 query-doc、relevance 三元组，所以首先将预处理后的文本映射入 DataFrame，并且计算词向量。

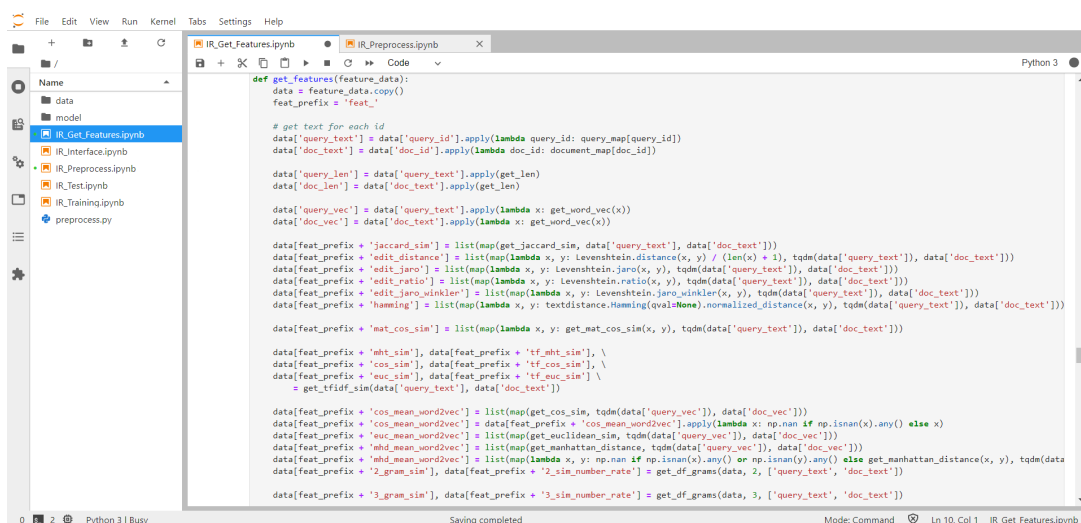


图 10: Getting Features

初步设计特征种类 46 项 [6], 大致涵盖了如下范围:

- Jaccard similarity
- Levenshtein distance
- TF-IDF
- 2-gram
- 3-gram
- Word vectors similarities
- Sparse matrix cosine similarity
- BM25 in group and overall
- Token-based statistics

在负例常数 `NEGATIVE_SCALE=50` 的数据集当中, query-doc pair 数量为 1633938。故而我们同样的采用了线程池优化。首先多线程运算, 然后合并特征。经过测试, 如图 11 所示, 163*46 项特征, 8 进程运算耗时 6 分 24 秒。

```
[6]: '''  
      Basic settings for training features generation  
      '''  
NEGATIVE_SCALE = 50 # 100 150 200  
csv_folder = './data/csv/'  
train_folder = './data/train/'  
dict_file = csv_folder + 'training.csv'  
export_file = csv_folder + 'export_training_' + str(NEGATIVE_SCALE) + '.csv'  
feature_file = train_folder + 'features_' + str(NEGATIVE_SCALE) + '.csv'  
  
init_dict(dict_file)  
generate_features(export_file, feature_file, new_set=True)  
  
Last executed at 2020-06-16 12:38:11 in 6m 24.25s  
  
Chunk size: 204243, Length: 1633938  
CPU core: 8  
Cores used: 8
```

图 11: Getting Features Timing

接下来根据 Pearson 相关度计算相关系数:

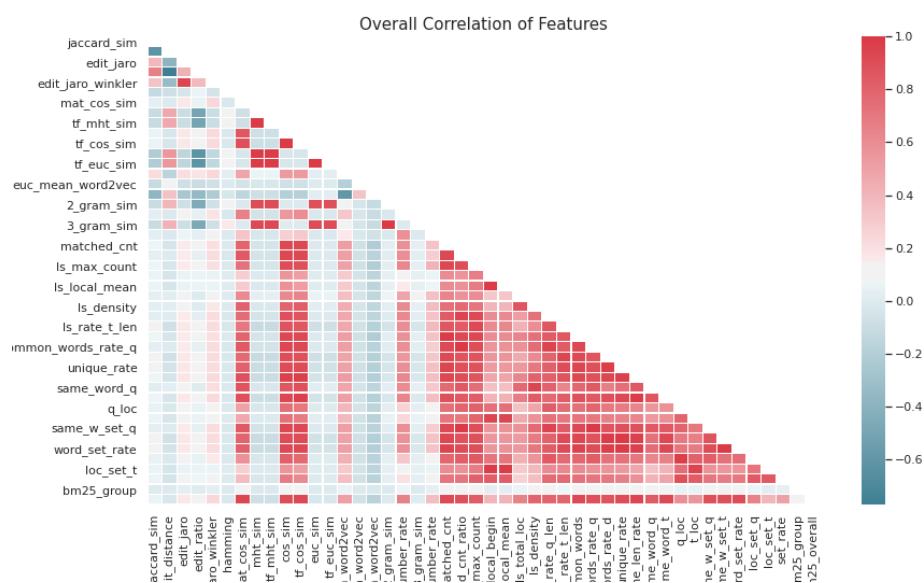


图 12: Correlation Matrix

可以看到图 12 热力图中出现大量强关联，意味着存在大量的冗余特征。而且冗余的部分是基于字符串 token 设计的统计学特征，例如 common words ratio、unique words ratio 等特征。

据此我们去除冗余特征，最终获得 20 类特征，如图 13 所示，可见冗余状况有所改善。

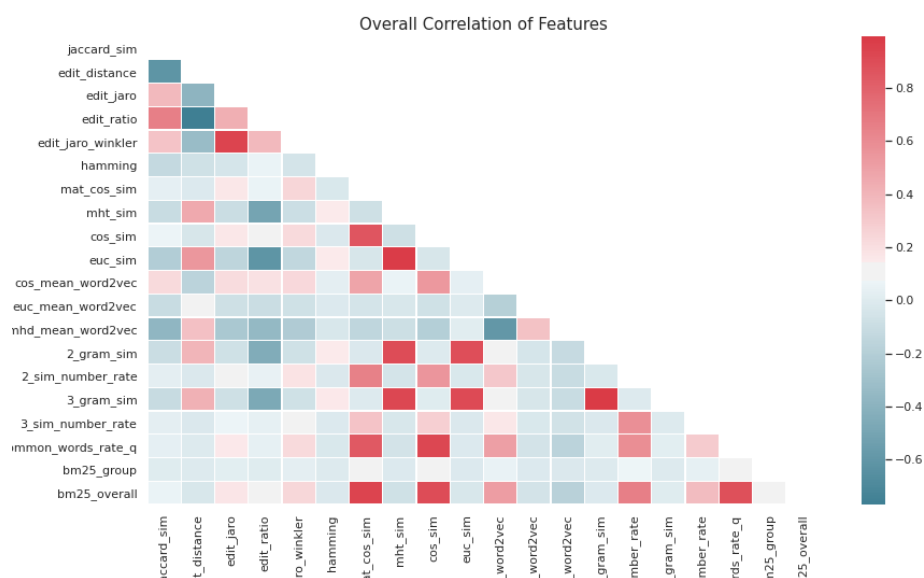


图 13: Correlation Matrix After Cleaning

5 Recall

Recall 阶段的意义在于缩减 Re-rank 阶段特征工程、模型排序的运算量。我们规定, RECALL_SIZE 为文档集全排序之后, 所取出得召回集和的大小。也就是进一步送入 Re-rank 阶段的文档数量。

Recall 相对于文档集而言, 本质同样是一次 ranking 操作。故而我们设计 Recall 方案需要保证:

- Recall 排序方案的运算量可以接受, 已知文档集为 50 万。
- Recall 在可接受的运算量下, RECALL_SIZE 内的召回率应可以接受。

为此我们选取了 BM25 以及 Cosine Similarity 两类特征进行初步的对照试验。原因在于 BM25 本身就是用于排序检索的一类常用算法, 而 Cosine 测定可以利用章节 4.1 所得的向量索引, 加速运算。且两种策略的运算量可以接受。

```
def recall_bm25(query_text):  
    '''  
        BM25 recalling strategy  
    '''  
    scores = g_bm25_model.get_scores(query_text.split())  
    sort_idx = np.argsort(scores)[::-1]  
    return list(map(lambda x: str(doc_idx2id[x]), sort_idx))  
  
def recall_sparse_mat_sim(query_text):  
    '''  
        Sparse matrix similarity  
    '''  
    query_vec = g_dictionary.doc2bow(query_text.split())  
    scores = g_index[query_vec]  
    sort_idx = np.argsort(scores)[::-1]  
    return list(map(lambda x: str(doc_idx2id[x]), sort_idx))
```

接下来我们通过 8 进程加速运算 BM25 以及 Cosine 相似度。在 validation 集上对 3000 条查询统计召回用时, BM25 耗时 4.3 分钟, 而 Cosine 耗时仅 2.2 分钟。

```
recall_bm25 recalling time cost: 4.355580401420593
```

```
recall_sparse_mat_sim recalling time cost: 2.1816662112871805
```

接下来对 `RECALL_SIZE` 的设定进行实验，以 10 为粒度绘图，查看不同 `RECALL_SIZE` 下的召回率对比如图 14 所示，可见 BM25 恒优于 Cosine 相似度。

当 `RECALL_SIZE=50`，BM25 的召回率约为 77.38%，`RECALL_SIZE=50` 时，BM25 召回率达到了 83.18%。并且 BM25 对于 3000 条查询的文档全排序仅需 4.3 分钟，性能上也可以接受。

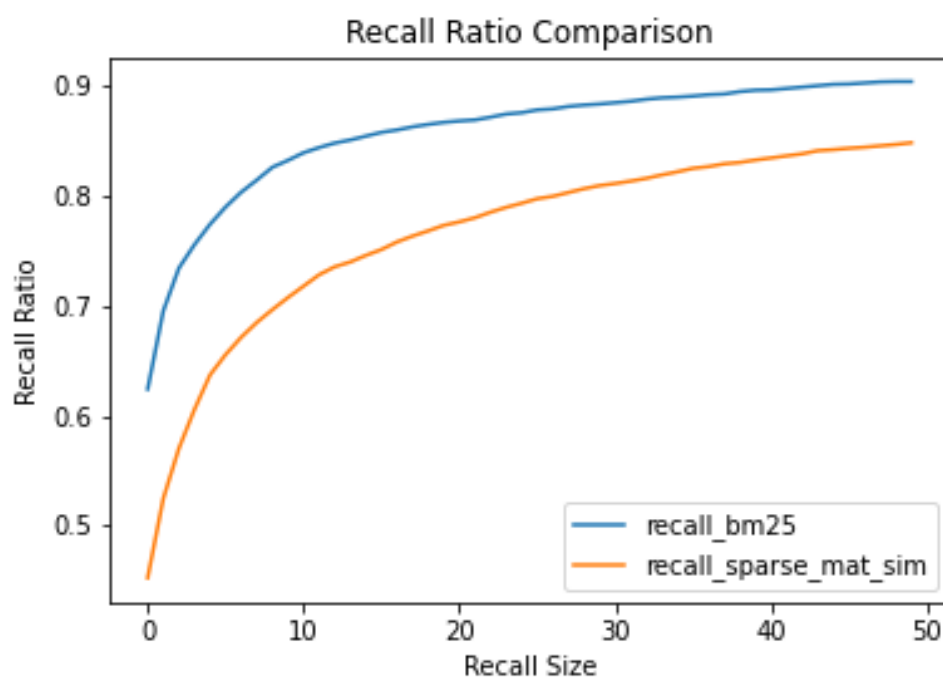


图 14: Recall Ratio Comparison

综上所述，我们选定的 Stage 1 召回策略即为 BM25。

6 Ensemble Learning

6.1 MRR 验证策略

接下来开始训练调参多个 Base Learner。

首先根据数据集格式要求，获得 `group` 分组信息，即根据 `query id` 划分 `group` 并得到每一组的大小：

```
def get_group(fold_part):
    fold_part['group'] =
        fold_part.groupby('query_id')['query_id'].transform('size')
    group = fold_part[['query_id', 'group']].drop_duplicates()['group']
    return group
```

首先使用 `NEGATIVE_SCALE=50` 的训练集全集, 以及默认参数训练 `LightGBM`、`XGBoost` 模型。将章节 5 阶段召回的数据进行模型 `Re-rank`, 并计算 `MRR` 指标。训练完成后, 分别计算 `LGB`、`XGB` 模型相关度预测得分, 如图 15 所示。

```
[56]: test_pd
```

Last executed at 2020-06-16 16:24:56 in 14ms

```
[56]:
```

	query_id	doc_id	lgb_pred	xgb_pred	recall_pred
0	404021	2451807	0.199839	6.047806	50
1	404021	583441	0.199839	6.053606	49
2	404021	265855	0.199839	6.048067	48
3	404021	5784331	0.199839	5.982104	47
4	404021	6841132	0.199839	5.954121	46
...
149995	733477	2179098	0.197580	3.850322	5
149996	733477	1291089	0.191662	3.738608	4
149997	733477	1517357	0.197580	3.999970	3
149998	733477	3485818	0.187168	3.116307	2
149999	733477	4917972	0.197580	4.063635	1

150000 rows × 5 columns

图 15: Default Hyper-parameters MRR

经过得分排序, 得到最终的 `Re-rank` 结果。将此结果与验证集 `label` 对比, 计算

MRR 得分。此时单模型与 BM25 MRR 指标分别如下：

```
MRR scoring:
LGB: 0.3595121693121697
XGB: 0.3831346560846564
BM25: 0.4211854497354501
```

可以看到不经过调参的模型输出得分，竟然连 BM25 的指标都无法达到。接下来需要对 Base Learner 进行调参。

6.2 XGBoost Model

6.2.1 Grid Search

XGBoost 单模型训练的接口实现如下：

```
def train(params, train_x, train_y, valid_x, valid_y, train_group,
          valid_group):
    model = xgb.sklearn.XGBRanker(**params)
    model.fit(train_x, train_y, train_group, verbose=False,
              eval_set=[(valid_x, valid_y)], eval_group=[valid_group],
              early_stopping_rounds=100)
    return model
```

我们引入 GridSearchCV 进行 Hyper-parameters 调整。GridSearchCV 顾名思义，即网格搜索、交叉验证相结合。

首先我们调整 `n_estimators`，其中 `cv_params` 是我们试图进行对比测试的参数列表，随后我们构造 GridSearchCV 实例并进行拟合：

```
cv_params = {'n_estimators': [400, 500, 600, 700, 800, 900, 1000]}
other_params = {'learning_rate': 0.1, 'n_estimators': 500,
                'max_depth': 5, 'min_child_weight': 1, 'seed': 0,
                'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0,
                'reg_alpha': 0, 'reg_lambda': 1, 'objective':
```

```
'rank:pairwise', 'eval_metric':'ndcg', 'random_state':  
    ↳ 2020,}
```

```
model = xgb.sklearn.XGBRanker(**other_params)  
optimized_GBM = GridSearchCV(estimator=model, param_grid=cv_params,  
    ↳ scoring=ndcg_score, cv=5, verbose=1, n_jobs=8)  
optimized_GBM.fit(train_x, train_y, group=train_group)
```

网格搜索结束之后，输出最佳 estimator 如图 16所示，即在给定的 candidates 当中，最佳的 estimator 数量为 400。

```
[26]: print(optimized_GBM.best_estimator_)  
Last executed at 2020-06-16 17:25:26 in 5ms  
XGBRanker(base_score=0.5, booster='gbtree', colsample_bylevel=1,  
    colsample_bynode=1, colsample_bytree=0.8, eval_metric='ndcg', gamma=0,  
    gpu_id=-1, importance_type='gain', interaction_constraints='',  
    learning_rate=0.1, max_delta_step=0, max_depth=5, min_child_weight=1,  
    missing=nan, monotone_constraints='()', n_estimators=400, n_jobs=0,  
    num_parallel_tree=1, random_state=2020, reg_alpha=0, reg_lambda=1,  
    scale_pos_weight=None, seed=0, subsample=0.8, tree_method='exact',  
    validate_parameters=1, verbosity=None)
```

图 16: Best Estimator for n_estimators

但是由于 cv_params 划分粒度较大，所以我们在 400 附近进一步缩小参数粒度，最终获得的最佳 n_estimators = 400。

以此论推，最终我们得到的最优参数组合如表 3所示。我们在调参的过程当中，是对参数进行分组调试的，将控制功能有关的参数一并进行测试。根据乘法原理，同时测试所有参数，测试的参数组合则会过多，测试时间会无法接受。

6.2.2 Optimized Parameters

n_estimators	400
max_depth	5
min_child_weight	1
gamma	0.2
subsample	0.8
colsample_bytree	0.8
reg_alpha	0.1
reg_lambda	0.05
learning_rate	0.1

表 3: XGBoost Best Params

经过上述调参，单模型 XGBoost 在验证集上测试结果为：MRR=0.4284。

6.3 LightGBM Model

6.3.1 Optimized Parameters

单模型 LightGBM 训练接口如下，分组信息同样可由 DataFrame.groupby 方法获得。

```
def lgb_train(hyper_params, train_x, train_y, valid_x, valid_y,
              group_train, group_valid):
    training_set = lgb.Dataset(train_x, label=train_y, group=group_train)
    validation_set = lgb.Dataset(valid_x, valid_y,
                                  reference=training_set, group=group_valid)
    model = lgb.train(hyper_params, training_set,
                      valid_sets=[training_set, validation_set], verbose_eval=500)
    return model
```

尽管章节 6.1 当中 LightGBM 默认参数的 MRR 指标不理想，但是 LightGBM 的训练速度和内存占用属于 Base Learner 当中最强的模型。最终调参结果如下。

n_estimators	400
max_depth	7
num_leaves	120
max_bin	5
min_data_in_leaf	75
feature_fraction	1.0
bagging_fraction	1.0
lambda_l1	0.0
lambda_l2	0.0
learning_rate	0.1

表 4: LightGBM Best Params

采取上表参数，单模型 LightGBM 在验证集上测试结果为：MRR=0.3824。

6.3.2 Feature Importances

输出 LightGBM feature importances 如图 17所示，可见 BM25、公共 token 比例等特征权重较高。

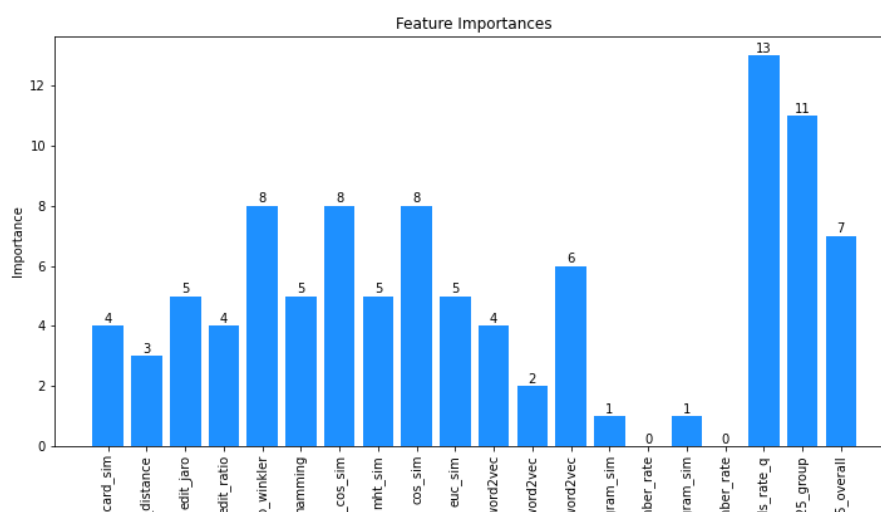


图 17: LightGBM Feature Importances

6.4 CatBoost

6.4.1 Visual Tools

CatBoost 属于较新的 Boosting 算法，生态也非常成熟完善，算法库功能齐全，甚至包括 grid search、plotting（图 18）等功能。进一步降低了编码成本。默认参数验证结果仅为 $MRR=0.2865$ 。

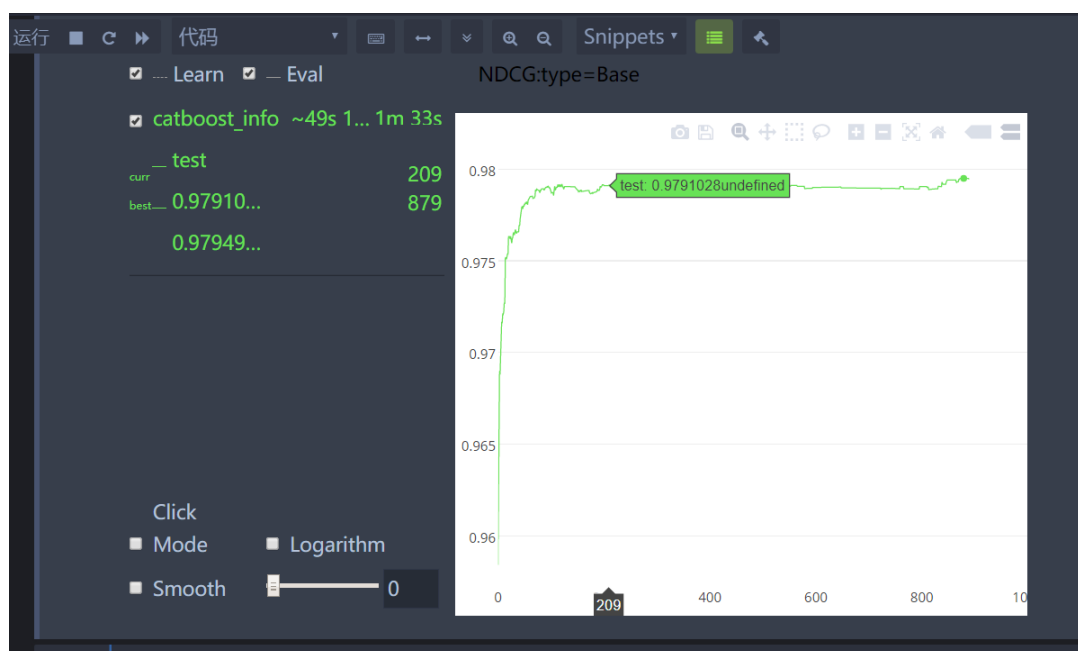


图 18: CatBoost Training

6.4.2 Bayesian Optimization

由于 CatBoost 单模型训练慢，参数组合测试耗时长。这里我们尝试 Bayesian Optimization[2]。如下代码所示，我们将单模型训练方法封装为 Bayesian Optimization 实例的回调参数，给出待优化参数及其候选取值范围，然后开始 Bayesian 优化迭代过程。

```
from bayes_opt import BayesianOptimization
# Bayesian Optimization
def bayesian_cat_train(l2_leaf_reg, bagging_temperature, learning_rate):
    params = {'loss_function': 'YetiRank',
              'learning_rate': learning_rate,
              'iterations': 800,
```



```
        'depth': 8,
        'use_best_model': True,
        'random_seed': 2020,
        'eval_metric': 'NDCG',
        'l2_leaf_reg': l2_leaf_reg,
        'bagging_temperature': bagging_temperature,
        'thread_count': 64,
        'early_stopping_rounds': 100,
    }

    model = cat_train(params, train_x, train_y, valid_x, valid_y,
                      train_group, valid_group)
    mrr = get_pred_mrr(model)
    return mrr

cat_opt = BayesianOptimization(bayesian_cat_train, {
                                'l2_leaf_reg': (1, 9),
                                'bagging_temperature': (1, 10),
                                'learning_rate': (0.1, 0.2),
                            })

cat_opt.maximize(init_points=5, n_iter=15)
```

由于选定了合适的参数范围，Bayesian 优化迭代的过程中，我们可以看到测试指标发生质的变化，从 $MRR=0.28$ 提升到 $MRR=0.3828$ ，如图 19所示，我们最终取第 18 次迭代的参数作为最终 CatBoost 的训练参数。

Last executed at 2020-06-17 12:33:33 in 1h 49m 19s					
iter	target	baggin...	l2_lea...	learni...	
1	0.3774	8.418	7.165	0.1286	
2	0.3768	4.715	5.565	0.1135	
3	0.374	5.455	4.538	0.1146	
4	0.3811	6.014	3.58	0.1678	
5	0.3707	2.442	3.977	0.127	
6	0.3798	3.535	7.113	0.1029	
7	0.3782	1.743	1.669	0.1482	
8	0.3817	5.398	3.677	0.1758	
9	0.3805	6.031	3.594	0.1544	
10	0.3823	5.702	3.571	0.1971	
11	0.3818	5.856	3.291	0.1998	
12	0.3803	5.284	3.253	0.2	
13	0.3821	4.868	3.663	0.2	
14	0.3811	4.522	3.297	0.1263	
15	0.308	4.402	3.838	0.2	
16	0.3158	4.795	2.912	0.142	
17	0.3607	5.112	3.946	0.1011	
18	0.3828	5.751	3.958	0.1375	
19	0.3718	6.137	4.164	0.109	
20	0.3665	1.812	1.521	0.1099	

图 19: CatBoost Bayesian Optimization

最终得到 $MRR=0.3828$ 。

6.5 Blending

通过对 Base Learners 输出加权 Blending 获得最终的排序结果。由于不同 Base Learner 预测取值不同，首先进行 normalization 统一到 $[0, 1]$ 内的实数范围内，然后进行加权平均值处理，如图 20所示，最终输出排序关键字作为 Re-rank 结果。

接下来枚举测试不同的加权组合，最终通过 LGB 加权 9, XGB 加权 1，得到测试结果 $MRR=0.4308$ ，如图 21所示。可见比任意单模型得得分都要高。当然由于数据集较小，我认为类似操作没有任何意义。

	query_id	doc_id	lgb_pred	xgb_pred	cat_pred	recall_pred	pred
0	404021	2451807	0.999546	0.987537	0.914845	1.00	0.975482
1	404021	583441	0.999546	0.992392	0.906713	0.98	0.969663
2	404021	265855	0.999546	0.990219	0.915339	0.96	0.966276
3	404021	5784331	0.999546	0.973772	0.904767	0.94	0.954521
4	404021	6841132	0.999546	0.976786	0.909102	0.92	0.951358
...
149995	733477	2179098	0.915886	0.547937	0.655948	0.10	0.554943
149996	733477	1291089	0.915886	0.593837	0.685402	0.08	0.568781
149997	733477	1517357	0.965091	0.582305	0.696489	0.06	0.575971
149998	733477	3485818	0.915886	0.417827	0.553499	0.04	0.481803
149999	733477	4917972	0.915886	0.630588	0.710584	0.02	0.569264

150000 rows × 7 columns

图 20: Blending

```
[43]: def get_weighted_pred(a0, a1, a2, a3):
test_pd['pred'] = (a0 * test_pd['lgb_pred'] + a1 * test_pd['xgb_pred'] + a2 * test_pd['cat_pred'] + a3 * test_pd['recall_pred'])
pred_mrr = MRR(get_np_pred('pred'), eval_labels)
return pred_mrr

print(get_weighted_pred(9, 1, 0, 0))
Last executed at 2020-06-17 13:27:27 in 3.04s
0.4308230158730158
```

图 21: Blending Weights

7 实验总结

本次实验耗时较长，不过也是大学期间最有趣的一次实验。如图 22所示，我从 5 月 28 日开始处理数据，中间一直在记录进展，就这样持续约 10 天。

之所以说实验过程非常有趣，是因为问题自由度非常高，数据集仅包含 raw text，可发挥的空间非常大；而且是以竞技的形式给出，优化的方向非常明确。

查阅资料、数据处理、特征工程、模型训练，从头开始编码进度非常缓慢，刚开始使用 Colab 的低性能 CPU，尽管 iPython 提供了极大的便利，不过每一次更换预处理方法、特征种类重新计算的时间成本都很高。让我意识到有些优化方案并非不敢想，

只是不敢做。生产工具的落后极大的限制了工作效果。



图 22: Diary

后续更换服务器之后实验效率提升显著。未来是云计算、云存储的世界，这一点果真切实体会到了。低成本的部署，便捷的运算工作，对生产力的提高显而易见。

验收前一天我才开始 MRR 的测试，这也是比较遗憾的一件事，实在没有时间细致调整参数，最后只能使用 XGBoost 单模型。尽管指标上并不理想，但是这是第一次进行完整的特征工程、模型训练、调参工作。体会到了其中时间成本的开销所在，进一步理解了课堂上所学原理之时，掌握了常用的机器学习工具、方法。以后对于此类工作定当更加轻车熟路。

课内知识与工业要求其实天壤之别，就算是前人总结好的方法规律，在实际的运用当中也可能面临诸多的思辨和对比。将知识运用到实践当中，是令人愉快的，因为这一步是充满创造力的，需要真正动手、动脑的。当看到自己的方法对实际的问题有所助益，这是最令人骄傲快乐的时刻。

References

- [1] Thorsten Joachims. “Optimizing search engines using clickthrough data”. In: (2002), pp. 133–142.
- [2] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. “Practical bayesian optimization of machine learning algorithms”. In: (2012), pp. 2951–2959.
- [3] Rodrigo Nogueira et al. “Multi-stage document ranking with BERT”. In: *arXiv preprint arXiv:1910.14424* (2019).
- [4] Yifan Qiao et al. “Understanding the Behaviors of BERT in Ranking”. In: *arXiv preprint arXiv:1904.07531* (2019).
- [5] Shuguang Han et al. “Learning-to-Rank with BERT in TF-Ranking”. In: *arXiv preprint arXiv:2004.08476* (2020).
- [6] Weilong Chen et al. “An Effective Approach for Citation Intent Recognition Based on Bert and LightGBM”. In: ().