摘 要

本文根据搜索引擎四大过程，先通过爬虫爬取知乎网问答信息，预处理数据，建立搜索引擎相关索引，并摒弃传统人工拟合排序公式的方式，将 LTR 思想作为搜索结果排序指导思想，训练 LambdaMART 模型用于结果排序，利用训练好的模型实现了一个可视化问答搜索系统。解决了传统搜索引擎在网页排序中要考虑的因素越来越多的，从而无法继续采用人工拟合的方式计算相关度的问题。

关键词：机器学习，LTR，问题答案推荐，LambdaMART，文本处理，关键词提取，爬虫，搜索引擎，索引

ABSTRACT

In this paper, we follow the four process of search engine to crawl the question and answer information of zhihu.com through crawler, preprocess data, establish the related index of search engine, and abandon the traditional artificial fitting sorting formula. The LTR thought is used as the guiding idea of the search results, and the LambdaMART model is trained for the result sort, and the training is used well. A visual question answering search system is realized by using the trained model. There are more and more factors to be considered in the traditional search engine, which can not continue to use artificial fitting to calculate the degree of correlation.

Keywords: Machine learning，LTR，Question answer recommendation，LambdaMART，Text processing，Keyword extraction，Crawler，Search engine，Index

目 录

第一章 绪 论

1.1 研究工作背景和意义

互联网高速发展的时代,搜索引擎作为互联网接口，相关技术也如雨后春笋般从出不穷。传统搜索引擎包含了搜索引擎爬虫网页爬取，索引建立，内容检索，结果排序四个过程，其中结果排序起初来用人工拟合公式的方式计算网页的相关度。而现今在当前热门的机器学习领域的带领下，出现了机器学习和搜索引擎结合下的产物 LTR，解决了网页排序中要考虑的因素越来越多的，从而无法继续采用人工拟合的方式计算相关度的问题。

所以就有了本课题，基于机器学习实现一个问答推荐系统，即用户输入问题，显示相关答案，本质上和搜索引擎相似。本文设计的是一个问题答案匹配的搜索引擎，数据源来自于知乎网，需要自己编写爬虫爬取，使用合适的模型，训练该模型后，利用训练好的模型为核心，实现一个问题答案搜索引擎。

1.2 机器学习国内外研究历史和现状

当谷歌的 Alphago 出现在世人面前并且击败了九段围棋高手李世石之后，机器学习算是正式进入了大众的视野，各大高校也纷纷开始了有关课题的学习和研究工作。那究竟什么是机器学习呢，我们先从学习的定义说起。 按照人工智能大师西蒙的说法，学习就是系统在不断重复的工作中对本身能力的增强或者改进，使得下一次的执行相同任务或类似的任务的时候，会比之前做的更好更出色或者效率更高。

顾名思义，机器学习是研究如何使用机器来模拟人类学习活动的一门学科。稍为严格的提法是：机器学习是一门研究机器获取新知识和新技能，并识别现有知识的学问。这里所说的“机器”，指的就是计算机；现在是电子计算机，以后还可能是中子计算机、光子计算机或神经计算机等。

机器学习属于人工智能领域，人工智能是从 20 世纪 40 年代和 50 年代开始[1]，逐渐走上历史舞台。1956 年正式被确立为一门学科。而之后人工智能领域的发展，多多少少可以看到机器学习的身影。机器学习是人工智能发展到一定阶段的必然产物。

从 20 世纪 50 年代到 70 年代，当时人工智能领域的学者们普遍认为，只要给机器赋予逻辑推理能力，那么机器就有了智能，当时具有代表性的成果主要有“逻辑推理家”程序以及之后的“通用问题求解”程序等。其中“逻辑推理家”[10]程序在 1952 年就证明了《数学原理》的 38 条定理，又过了 11 年，就证明了全部 52 条定理，而且定理 2.85 甚至

比罗素和怀特海的证明更为巧妙，这个程序也给作者带来了 1975 年的图灵奖。

然而随着研究的深入，学者们发现，光有逻辑推理能力，是完全不够的。又掀起了一股新的认知狂潮[11]，认为要让机器智能，必须得先让机器有知识，于是从 20 世纪 70 年代开始，学者们的研究方向转向了赋予机器人类的知识，大量的专家系统出现，但是人们也逐渐发现，由人来把知识总结出来再教授给计算机的这个过程，是相当困难的，于是有的学者就想：机器要是能自己学习就好了。这也就是机器学习产生的必然性。

1.3 LTR 研究背景和现状

在互联网刚刚诞生的时候，最开始出现的是门户网站，后来才有了搜索引擎，而当时的搜索引擎采用的检索模型是十分简单的，依赖的特征值往往很少。如今互联网发展至今，对页面排序的时候，考虑的因素越来越多，比如 pagerank，查询文档匹配单词数，URL 地址长度等。此时若指望人工将几十种考虑因素拟合出排序公式是不太现实的，而机器学习正好十分适合这种工作。这是 LTR 诞生的原因之一。

另一方面原因是：对于监督学习来说，需要大量已经标记好的数据用于给模型训练，在此基础上才能自动学习，而如果单靠人工标注大量的训练数据明显不太现实。对于搜索引擎来说，虽然无法靠人工标注大量数据，但其实有很好的代替品，那就是用户的点击记录，用户点击某页面次数越多，代表了该页面对于用户来说更加相关，就是说应该排序更加靠前，尽管点击次数多不一定完全和相关性有直接联系，但实践表明，这种点击数据对于机器学习系统来说是可行而有效。

1.4 本文的主要创新点

本论文以 LTR 思想中的 LambdaMART 模型为搜索结果排序模型，以传统搜索引擎基本原理和结构为重点研究对象，实现了一个基于机器学习的问答推荐算法系统，主要创新点有:

1. 将机器学习引入搜索引擎中，即使用 LTR 的思想完成对搜索结果的排序，使得在特征值过多的情况下，不需要人工拟合排序公式。
2. 用于爬取知乎网数据的爬虫，使用 redis 实现了断点续传的类似功能，使得爬虫可以在出现任何意外情况间断的情况下继续工作，加快爬取效率。
3. 使用了多个索引结构，包括答案关键词的正逆向索引，问题相关答案索引，问题关键词逆向索引等，使得进一步缩减搜索时间。

1.5 本论文的结构安排

本文章节内容安排如下：

第一章绪论，阐述了本文工作内容，当前机器学习的国内外研究现状和 LTR 研究背景和现状。

第二章机器学习基础，阐述了机器学习的基础理论，介绍了机器学习的四大分类及各自特点，简单阐述了机器学习的基本原理和相关问题。

第三章爬虫基础，简单说明了爬虫的基本原理，介绍了爬虫在爬取过程中采用的不同搜索策略，介绍了一般爬虫在爬取数据中常见的问题和对应的解决方案等。

第四章搜索引擎基础，先简单介绍搜索引擎中使用到的四大过程，然后分别详细介绍文本的处理过程，包括网页文本提取，中文的分词，去除停用词，除噪。详细说明搜索引擎中使用的索引的组织结构和建立方法。详细说明了搜索引擎中的各种检索模型。最后详细介绍了 LTR 的分类和各自的思想特征。

第五章问答推荐系统研究与设计，论文主体部分，详细阐述了问答推荐系统的设计与结构。先简单描述代码结构，以搜索引擎四大过程为顺序，详细论述了爬虫的设计，数据预处理过程，索引的建立，相关度的标记，特征值的选取，用户交互界面的设计和系统的优化等方面。

第六章系统测试，说明了测试环境，列出测试用例和相应结果，并对结果进行分析。

第七章全文总结和展望，对论文全文的内容进行总结并对已有系统的后续工作进行展望。

第二章 机器学习基础

2.1 机器学习基础

机器学习，即是一种从数据中获取想要的信息的一种手段，或者说，从数据中学习一些新的知识的手段。机器学习的定义多种多样，但有一个广泛被认可的一个定义就是：对于某类任务 T 和性能度量 P，如果一个计算机程序在 T 上以 P 衡量的性能随着经验 E 而自我完善，那么我们称这个计算机程序在从经验 E 学习。这简短的定义，是 Mitchell 在 1998 年给出的，一直在业界被广泛流传。本文中使用到的 LTR 就是机器学习的一个分支，意在让机器学会如何排序搜索结果。

2.2 机器学习分类

机器学习分为四大类[1]：监督式学习，非监督式学习，半监督式学习，增强学习。

监督式学习，基本上实现的就是分类的功能， 它从有标签的训练数据中学习，然后给定某个新数据，预测它的标签。这里的标签，其实就是某个事物的分类。在监督学习中，输入的数据被称作训练样本，每组样本都会有一个明确的标识或者结果。LTR 属于监督式学习范畴。

非监督学习，与监督学习恰恰相反，非监督学习面对的训练数据都是没有标签的，给定数据，从数据中学习，能学到什么，这取决于数据本身具有什么样的特征。

半监督式学习，这种学习方式借鉴了监督学习和非监督学习，去其糟粕取其精华，半监督学习通过对部分标签数据的学习，从而对之后的数据进行分类，但不像监督学习一样，一旦学习完毕，就不再更新自身，而半监督学习会根据之前的经验继续对数据进行学习。

增强学习，计算机通过观察来学习做什么样的动作。每个动作对环境都会产生影响，计算机会根据观察到的周围环境的影响当作反馈来做出判断。我们熟知的谷歌的 alphago 使用的算法就属于增强学习的范畴。

2.3 机器学习基本原理

机器学习中数据并非常规意义上的数值，它是对某个对象的某性质的描述，被描述的性质被称作属性，而用来描述的数据就叫做特征值，不同特征值组成的向量就是一条数据，可以看作一个实例对象。数据的不同属性之间相互独立，每个属性都代表了一个

不同的维度，而由这些维度共同组成了一个特征向量空间，每一个实例在这个空间中，都可以被看作一个向量。

传统机器学习中，我们需要用许多特征值向量来训练各种各样的模型，使得他们的参数对训练数据有较好的拟合程度。由于一个算法不可能完全拟合所有训练数据(一般情况下)，也不可能对所有数据的预测都准确无误，所以产生了误差性能这一重要指标来表示一个训练模型的好坏程度。误差的定义是：模型的预测输出和真实输出之间的差异。进一步可分为训练误差和测试误差。训练误差指的是模型在训练数据集上的误差，测试误差是在新的数据集上的误差。

模型训练中，还可能会出现过拟合现象，当模型过于的拟合了训练集数据，导致缺少了泛化能力的时候，就被称作过拟合。过拟合现象的解决方式有很多种，最常见的就是正则化，在目标函数后增加正则项。

假设函数即机器学习模型，本质上，未训练的模型即是一个参数未确认的假设函数，有的模型假设函数可能很简单，也可能很复杂。

除了传统的机器学习，目前机器学习较为热门的两个方向是：SVM 和 DeepLearning，即支持向量机和深度学习。

什么是深度学习，深度学习是使用深度神经网络，用大量的数据训练网络，使之具备相应的能力能抓取数据中的特征，与传统机器学习不同的是，深度学习需要大量的数据，即”大数据”，而且深度学习提取出来的数据的特征的相关知识，是存在于机器空间的，人类无法感知无法理解，对人类来说，就是一个黑盒，性能好，但却缺乏解释性。

第三章 爬虫基础

3.1 爬虫基本原理

互联网上的信息储存在成千上万的服务器之中，早期，人们在做搜索引擎的时候，网站的信息都是通过人工录入的，但随着互联网的发展，这些人工录入的数据的量，早已不能满足人们，于是便有了爬虫的诞生，爬虫诞生之初是为了搜索引擎所服务的，而不是现在大家常见的盗取他人网站等不良用途。

爬虫是一个自动提取网页信息的程序，它为搜索引擎提供搜索数据的来源，是搜索引擎的重要组成部分。传统意义上的爬虫，是由一个或者若干的起始网页开始，按照一定规则，不断从当前的链接中过滤筛选出所需要的链接放入爬取队列，再根据一定的规则，选择队列中的链接继续爬取，直到队列为空，从而爬取到海量的数据，为搜索引擎提供的数据保证。爬虫一般运行流程图如图 3-1 所示：

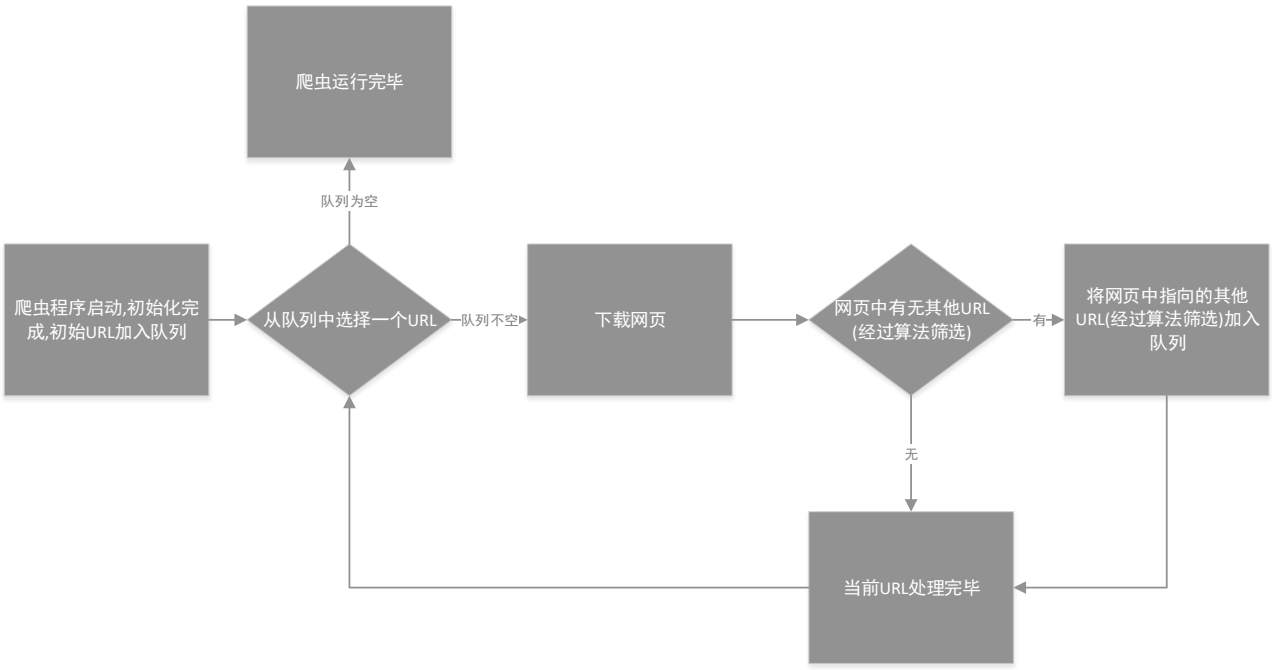

图 3-1 爬虫一般运行流程图

3.2 爬虫搜索策略分类

搜索策略，即选取什么样的网页爬取的策略。根据爬虫在爬取数据过程中采取的搜索策略不同，一般分为以下三种策略：

1. 广度优先搜索：广度优先搜索，和其他广度优先的算法一样，优先保证算法广度，即在爬取过程中，先将当前层次的所有网页都爬取完毕，再进行下一层次的网页内容爬取。这个算法的核心思想是，认为越靠近初始网页的网页，相关程度越高，所以根据广度优先算法获取到的网页，整体上相关程度会高一些，但实则随着网页层次的深入，大量的不相关网页也会被爬取，降低爬取速度。
2. 深度优先搜索：深度优先搜索，和其他深度优先算法一样，优先保证算法深度，即在爬取过程中，先从初始页面，获取到一个链接后，直接爬取该链接内容，继续再从这个链接获取下一个链接，这样处理完一条路线后，再次处理下一条路线。这个算法也并不复杂，但是由于深度优先算法，优先爬取离初始页面远的页面，这样爬取到的页面相关程度很低，价值不高，一般很少采用。但由于知乎网问答页面组织方式的特殊性，本系统的爬虫采用的是深度优先算法，后面会详细叙述。
3. 最佳优先搜索：这个搜索策略下，会有一个分析算法，用于分析相应链接与主题相关程度，用于评判一个链接的好坏程度，该搜索策略下，会从队列中选取分析算法筛选出来的有效链接开始爬取网页，每次都只爬取分析算法认为有效的链接，减少了爬取无用网页带来的爬取速度减慢问题。算法核心在于如何设计分析算法。最佳优先搜索策略本质上是一个局部最优算法。

3.3 爬虫常见问题以及解决办法

爬虫基本原理十分简单，但由于爬虫的辨识度太高，即爬虫的机械重复程度很高，很容易识别出爬虫，一些网站就会对一些来历不明或者用意不明的爬虫进行识别和限制，一旦识别出非搜索引擎爬虫，就会采用各种各样的手段限制你的动作，比如限制请求数目，暂时封禁相应 IP 访问，暂时封禁相关帐号，暂时封禁相关网段等。

解决办法有很多，下面简单说明几个方法：

1. 构造合理的 HTTP 请求头

HTTP 的请求头是在你每次向网络服务器发送请求时，传递的一组属性和配置信息。HTTP 定义了十几种的请求头类型，不过大多数都不常用。只有下面的 7 个字段被大多数浏览器用来初始化所有网络请求，表 3-1 是我浏览谷歌的时候的请求头情况。

表 3-1 HTTP 请求头

属性	内容
Host	https://www.google.com
Connection	keep-alive
Accept	text/html，application/xhml+xml，application/xml;q=0.9，

	image/webp，*/*;q=0.8
User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
Referrer	https://www.google.com
Accept-Encoding	gzip， deflate， br
Accept-Language	zh-CN，zh;q=0.8，zh-TW;q=0.7，zh-HK;q=0.5，en-US;q=0.3，en;q=0.2

虽然网站会对请求头都检查一遍，但最重要的还是 User-Agent 参数，你需要把 User-Agent 改成不容易引起怀疑的内容才行，如果你使用了一些三方库的时候，一般都需要手动修改 user-Agent 的值，不然很容易被检测出并查封，比如 python 的标准库 urlib 中，默认的 user-agent 值为 Python-urllib/3.4，这很容易就被查封。

2. 设置 cookie

Cookie 是存储在用户侧的一个被加密的数据，一般 cookie 对于服务器来说，都代表了一个用户的相关数据，即可以用来识别用户，如果你的爬虫被网站封禁了，如果网站在你的用户侧存储了 cookie，那么有可能网站就是通过 cookie 找到了你，并封禁了你的 cookie 的相关访问。这种情况最简单的方式就是直接禁止服务器写入 cookie，防止网站根据 cookie 而封禁访问。

但有些网站是必须有 cookie 才能访问的，因为需要保持用户的登录状态(如知乎)，这时候我们可以保存多个 cookie，当有 cookie 被封禁的时候，我们可以切换到其他 cookie 即可。

3. 降低访问频率

一般情况下，为了提升爬虫效率，大部分情况下都会将爬虫设计成多线程，使得爬虫在短时间内爬取到大量页面上的数据，会给服务器带来压力，因此一般网站都会有过载保护，会阻止访问速度异常的连接，解决这种情况的最好办法就是相应降低访问速率。

4. 注意隐藏的元素

网站为了反爬虫，一些网站会在表单中故意设计一些对用户隐藏的元素。这些隐含字段可以让字段的值对浏览器可见，但是对用户不可见，除非查看网页源代码。如果你的爬虫无差别的把隐藏元素都爬出来了，那么就很容易被网站识别出来，只要被网站发现，就有可能立马被封账号或者 IP，所以在提交表单一般需要先看一下元素的相关属性再进行爬取。

5. 使用代理

由于 IP 的不可伪造性，很多网站对爬虫的封禁，都是针对 IP 的，如果你的 IP 被网站封禁了，那么最有用的手段就是采用代理 IP 访问的方式来伪装你的请求头。代理 IP 可以在很多云服务网站中找到。

第四章 搜索引擎基础

4.1 搜索引擎基本原理

互联网发展如此迅速，很大程度上是搜索引擎的功劳，如今的搜索引擎已经不是从前的简单只是根据网页的几个简单特征值来计算网页内容的相似度，也已经不是一起那个数据量少的可怜的检索时代了。现在的搜索引擎十分强大，里面涉及的原理十分复杂，但我们可以通过了解搜索引擎的基本原理，来逐渐了解一下搜索引擎内部的工作原理。

搜索引擎一般分为以下四个部分：

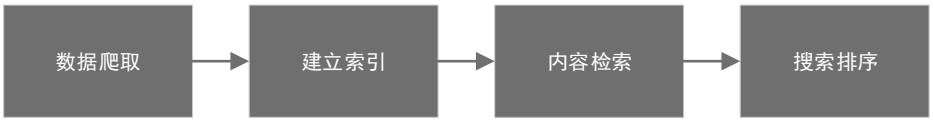

图 4-1 搜索引擎组成部分

4.2 数据爬取

搜索引擎的数据爬取，即网站资源的获取，早期互联网刚刚发展起来的时候，网站资源的获取，完全靠人工获取并输入，像早期的各大门户网站等。随着互联网发展，如何在互联网中搜集想要的信息就成了一大挑战，于是就有了爬虫。网络爬虫技术是搜索引擎最根本的技术。通过爬虫，我们可以将网络上数以百亿计的网页信息存到本地，为搜索引擎提供数据支持。

4.3 文本特征值处理

爬虫爬取数据完毕后，下一步工作就是对爬取到的文档进行一定程度上的文本特征值处理，一般包括：提取文本，分词，去停用词，除噪等。

4.3.1 提取文本

提取文本，即从源代码中分析提取出所需文本的过程，爬虫爬取到的数据都是网站的源代码表示形式(如 HTML)，包含大量的不可见格式标签，这些标签都只是用于给浏览器解析，而展现给用户特定的样式，无实际意义，所以需要从中剔除掉，提取出用户可见的文本。除了用户可见的文本需要提取外，一般搜索引擎还会提取一些特殊的文字信息，

如 meta 标签等。

4.3.2 分词

分词，即把文本中的所有词语分离出来，分词根据语种的不同，有不一样的分词算法。中文分词指的是将汉字序列切分出一个个的词语。中文分词比英文分词要难许多，我们知道，在英文中，单词之间是按照空格分割的，而中文只有字，句，段有明显的自然分隔符，而词语没有。中文分词的算法主要有两类： 基于词典分词算法，基于统计的机器学习算法。

4.3.3 去停用词

停用词，即一些在语言中大量出现，而又对实际语言内容没有任何影响的词语，比如中文中的“的”，“得”，“地”等助词，或者是“啊”，“呀”等感叹词，包括一些转折介词等，都被称为停用词，而英文中也有类似的词语，如“the”，“to”，“of”等，在分词处理后，我们还需要去除停用词，进一步减少无用词语带来的效率损失问题，减少无意义的计算和检索时间。

4.3.4 除噪

除噪，即进一步去除无意义内容，这些内容被称作噪声，或者噪点，如网页中的导航文字，版权信息，广告栏等，这些内容对于排名计算毫无意义，只能增加计算量，所以在更进一步处理之前，需要除去这些无意义内容。搜索引擎会在这个阶段将这些内容识别并去除，这个过程就被称为除噪。

4.4 索引

搜索引擎中之所以需要索引，是为了加快搜索引擎检索的速度，有了索引，在检索过程中就不需要对所有文档进行遍历操作，从而大幅度增加检索速度。

4.4.1 索引组织方式

建立索引[4]，即如何把爬虫爬取到了网页信息保存到本地，这里主要涉及到索引数据结构的设计，索引的分类和建立索引的方式也有很多种，下面介绍一种简单的索引组织方式。

倒排索引，又称逆向索引，先简单介绍一下基本概念：

文档：文本形式的待检索对象。

文档编号：搜索引擎内部，标记每个文档的唯一 ID 编号。

单词编号：搜索引擎内部，用于标记每个单词的唯一 ID 编号。

单词词典：文档集合中出现过的所有单词构成的字符串集合，单词词典内每条索引都记载了单词本身和相关信息等，常用的存储方式有：哈希链表形式，树形结构等。

倒排列表：出现了某个单词的所有文档的文档集合以及相关信息，倒排列表可以记录哪些文档包含了某个单词，倒排列表由倒排索引项组成，每个倒排索引项由文档 ID，单词出现次数等信息组成。

由于倒排比较简单，下面给出一个简单例子便于直观理解，见表 4-1：

表 4-1 例子文档内容

文档编号	文档内容
1	今天小明从学校回家
2	今天小兰从学校出发
3	学校小卖铺有辣条卖
4	小明回家路上买辣条
5	小兰不喜欢吃辣条

根据以上的文档内容，即可生成对应的倒排索引，索引内容见表 4-2：

表 4-2 倒排列表内容

单词 ID	单词	文档频率	倒排列表(文档 ID，出现次数)
1	今天	2	(1，1)，(2，1)
2	小明	2	(1，1)，(4，1)
3	小兰	2	(2，1)，(5，1)
4	辣条	2	(3，1)，(5。1)
5	学校	3	(1，1)，(2，1)，(3，1)
6	回家	2	(1，1)，(4，1)
7	从	2	(1，1)，(2，1)
8	卖	1	(3，1)
9	路上	1	(4，1)
10	不喜欢	1	(5，1)
11	出发	1	(2，1)

从例子中，我们可以看出，倒排索引将文档中的每个单词和文档都对应生成一个唯一 ID，并存储下对应关系，接着记录下每个单词在所有文档中出现的次数，并记录为文

档频率，并把每个单词出现的文档 ID 和出现次数都记录下来。

4.4.2 建立索引

理解了简单的倒排索引的组成和结构，下面介绍索引生成的几种方式：

a) 两遍文档遍历法

生成索引步骤：

1. 遍历第一遍文档，收集全局信息。包括：文档集合的文档总数 N，文档集合包含的所有单词数 M，每个单词的文档频率 DF。把所有单词对应的 DF 值相加，即可计算出最终索引的所需内存大小，分配所需内存大小，以供使用。
2. 遍历第二遍文档，逐个单词建立倒排索引。每扫描到一个单词，查询对应 ID，如没有则生成唯一 ID 并保存对应关系，接着修改该单词在当前文档 ID 下出现次数+1，即最终会得到这个单词在文档出现的次数 TF。最后第二遍遍历扫描结束后，第一步分配的内存将会全部占满，每个单词对应两个指针，分别指向了内存区域的片段，即为对应单词的索引项。

缺点：一次性需要大量的内存，对机器性能要求高。

b) 排序法

生成索引步骤：

1. 读入一个文档，并对读入文档进行编号，标记唯一 ID，解析文档内容
2. 将单词对应相应单词 ID。建立相应的索引项(单词 ID，文档 ID，单词频率)
3. 将索引项存放到中间结果存储区。
4. 处理下一个文档，重复 1 步骤。
5. 如果中间结果存储区占用完毕，则对中间结果存储区的所有索引项依据单词 ID 为主项，文档 ID 为辅助项进行排序。
6. 将排序好的中间结果存储区合并写入最终索引文件。

4.4.3 搜索引擎中的索引

前面简单介绍了什么是索引，而在搜索引擎中，我们使用的索引有所不同，一般的搜索引擎的索引部分，都是由正向索引和反向索引(倒排索引)构成。

正向索引，这个索引所记录的内容是每个文档对应的信息，比如关键词 ID，关键词出现的次数，格式，位置，TF 值等，具体保存什么信息，根据实际情况有所不同。实际搜索引擎中，索引中的关键词全部是以 ID 形式保存，而关键词 ID 和关键词对应关系，会保存在另外一个索引中。这种每个文档 ID 对应一串关键词 ID 的索引，被称为正向索引。

下图为一个正向索引的例子，见表 4-3：

表 4-3 正向索引例子

文档 ID	索引项
1	今天，小明，学校，回家
2	今天，小兰，学校，出发
3	学校，小卖部，辣条，卖
4	小明，回家，路上，辣条
5	小兰，不喜欢，辣条

反向索引，因为正向索引无法直接应用到搜索中，假设用户搜索了关键词 3，那么搜索引擎需要检索库中所有文档，得到所有包含关键词 3 的相关文档，这样的时间消耗对于用户来说根本不切实际，不可取，所以我们必须提前建立反向索引，供用户搜索使用，可以大幅度减少搜索时间。这里的反向索引保存的信息一般包括：关键词 ID，包含对应关键词的文档 ID 序列。

下表为一个反向索引的例子，见表 4-4：

表 4-4 逆向索引例子

关键词	文档
今天	1，2
小明	1，4
小兰	2，5
辣条	3，5
学校	1，2，3
回家	1，4
从	1，2
卖	3
路上	4
不喜欢	5
出发	2

4.5 检索模型和搜索排序

索引建立完毕后，搜索引擎的前期工作基本已经完成，接下来已经可以向外提供搜索服务了，当用户输入要搜索的内容时候，搜索引擎会根据之前生成的反向索引迅速的

获得相关文档，接下来的工作就是搜索排序，据相关度排序文档，以便用户优先看到自己想要的内容。

4.5.1 初始子集的选取

当粗略获得了所有相关的文档后，往往相关文档的数量十分的庞大，甚至几十万，几百万，这么大的数据量进行匹配计算的话，所需要的时间无疑会更长，而且用户并不关心也不可能关心排名十分靠后的文档，显然十分的浪费计算机的性能。为了进一步提升搜索速度，进一步满足用户需求，一般在实际搜索引擎中只计算权值较高的文档返回给用户。

4.5.1 检索模型

经过了初始子集的选取后，剩下数量不是很庞大的文档序列，接下来要做的是利用检索模型，计算出文档相关值，根据相关程度排序搜索结果。把文档集合划分为 4 个部分，第一部分是文档出现了用户查询的关键词也同时是被用户认为是相关的，第二部分是文档出现了用户查询的关键词却不被用户认为是相关的，第三部分是文档没有出现用户查询的关键词但却被用户认为是相关的，第四部分是文档没有出现用户查询的关键词，也不被用户认为是相关的。检索模型种类很多，判断好的模型的标准是：应该尽量提升第一部分和第三部分的文档的排名，降低第二部分和第四部分的排名。

下面介绍几种检索模型[9]：

4.5.1.1 布尔模型

布尔模型是检索模型中最简单的，文档和用户搜索的内容的相似度计算是通过布尔代数计算得出的。

布尔模型中用户的搜索被视为逻辑表达式，比如用户要查询计算机硬件或者软件相关的内容，那么可以用以下逻辑表达式表达搜索：

计算机 AND (硬件 OR 软件)

根据这个逻辑表达式，包含计算机关键词的文档中，还包含硬件或者还包含软件关键词，即被认为和用户查询相关，否则不相关。

从上面可以看出，布尔模型十分简单直观，但是存在明显的缺陷，即：无法排序，检索结果是二元的，文档只被划分为相关和不相关两个部分，其检索结果过于的粗糙。

4.5.1.2 空间向量模型

空间向量模型目前属于较为成熟的模型之一，被广泛应用到各个领域的各个方面。

4.5.1.2.1 文档表示方法

在空间向量模型中，文档有自己特殊的表示方法，会把每个文档表示为一个由 N 维特征值组成的一个向量，特征值的选取没有具体要求，但都会根据一定依据计算各自特征值的权重，由这 N 维向量就组成了文档。

下面是空间向量模型的文档表示例子，见表 4-5：

表 4-5 空间向量模型文档表示

文档 ID	文本长度	搜索关键词 TF*IDF	包含搜索关键词的最小窗口大小
文档 1	56	1.54	20
文档 2	74	1.22	60
文档 3	34	0.57	7
文档 4	22	0.36	4

通过特征转换，所有文档和用户的查询都可以转换为对应的特征值向量表示。

4.5.1.2.2 相关性计算

给定用户的搜索向量和文档向量，即可计算之间的相关性了。余弦相似度是一种较为合适的指标来当作文本之间的相似度。

设文本D_1和文本D_2的空间向量表示分别为：

$D_1 = (w1_1, w1_2 .. w1_n)$

$D_2 = (w2_1, w2_2 .. w2_n)$

则余弦相似度计算公式如式 4-1 所示：

$$\mathrm{sim}(D_1, D_2) = \frac{D_1 \cdot D_2}{\|D_1\| \times \|D_2\|} = \frac{\sum_i (w_{1i} \times w_{2i})}{\sum_i {w_{1i}}^2 \times \sum_i {w_{2i}}^2} \quad \text{(式 4-1)}$$

余弦相似度公式本身已经对特征值向量长度做了规范化，主要是为了惩罚长文档。但是存在明显的过分惩罚的现象，如果同时存在长文档和短文档，短文档的相关系数会大大高于长文档。

4.6 LTR

LTR，全称 Learning To Rank，前面介绍了一些检索模型，他们的功能都是用来实现对文档的的排序，使得用户更关心的内容排名更加的靠前，而 LTR 也是一种可以实现对搜索结果进行排序的方法，这是几年机器学习兴起之后，变的热门的一种排序方法。这是机器学习和搜索引擎领域结合下产生的全新排序方法。

从前面介绍的检索模型可以看出，前面的检索模型大部分考虑的特征值比较少，而且一般都是人工拟合排序公式，在搜索引擎刚刚发展的时代考虑的因素不多，还适用，而如今互联网发展如此迅猛，作为连接互联网重要的枢纽的搜索引擎，之前单一的特征值和人工拟合，已经不能满足人们的搜索需求。随着搜索排序要考虑的因素越来越多，人工拟合已经变的不切实际，而这种类型的工作又很适合用机器学习来处理，于是便有了机器学习的勇武之地，这是 LTR 诞生的原因之一。

还有一方面的原因是，机器学习分为监督学习和非监督学习，我们的排序能通过监督学习来实现，而监督学习需要大量的训练数据，才能完成模型的训练。而对于搜索引擎来说，有一个天然而又直接的指标来完成对搜索结果的标注工作，那就是用户的点击记录，用户会倾向于点击相关的搜索结果，所以我们可以认为，用户点击量越高的结果，应该就是相关度越高的搜索结果。依次为依据，我们就可以标注我们的搜索结果，从而作为训练数据的来源。

机器学习的排序系统由 4 个步骤组成：标注训练数据，文档特征值提取，训练模型，采用模型排序。

从目前研究的情况来说，LTR 一般被分为 3 大类[2]：单文档方法(PointWise)，文档对方法(PairWise)，文档列表方法(ListWise)。

4.6.1 单文档方法(PointWise)

单文档方法中，训练数据是以单个文档为处理对象的，首先将文档转换为特征值向量后，机器学习模型会根据从训练数据中学习到的回归(或者分类)分类函数对单一文档进行打分，打分结果即为搜索结果相关系数，下面是一个简单的例子，见表 4-6：

表 4-6 单文档方法特征值例子

文档 ID	查询 ID	PageRank 值	余弦值	文档长度	相关值
1	2	3	0.6	30	0.8
3	2	4	0.13	352	0.5
2	3	5	0.24	35	0.7
3	3	1	0.36	45	0.5

2	4	5	0.3	87	0.3
4	4	8	0.4	12	0.8

例子中，采用了 3 个特征值，单文档方法中，模型会根据每一条训练数据，尽可能的拟合一个回归函数来接近训练数据，模型的输入是特征值向量(例子中是 3 维的特征值向量)，输出是相关值。

4.6.2 文档对方法(PairWise)

文档对方法则是换了一个角度对搜索结果进行分析。对于搜索排序来说，最关键的是确认文档的先后顺序，即两个文档谁在前谁在后的问题。只要能确定任意两个文档的先后顺序，我们即可以完成对搜索结果的排序。总的来说，文档对方法相对于单文档方法来说，将重点转移到了文档顺序是否合理进行判断。

这种机器学习排序方式，最终的训练目标是判断任意两个文档的组成的文档对是否已经满足顺序关系。具体的学习过程如下，首先还是人工标定好训练数据集。将文档转换为特征值向量，根据人工标定的相关度，即可得到多对文档顺序对，将文档对作为训练数据，训练模型。

下面是一个简单的文档对方法的例子：

原始人工标定数据如上表 4-6，则我们可以得到以下的文档顺序对，见表 4-7：

表 4-7 文档顺序对

查询 ID	文档 ID	顺序关系	文档 ID
2	1	>	3
3	2	>	3
4	4	>	2

这就是我们要作为训练数据的文档对，机器学习的模型有很多，不管选择何种模型，最终训练的效果就是输入一个文档对，模型可以判断该顺序是否合理，如果合理，则文档 A 在文档 B 前面，否则不合理。通过这样的方式，我们就可以完成对搜索结果的排序工作。

文档对方法摒弃了单文档方法的只对单一文档打分的办法，做出了改进，但依旧存在一些明显的问题，比如说对每个文档的位置考虑都是同样的权重，或者说对每个位置的文档都是一样的待遇，这样就会出现一个问题，实际应用中，用户往往会更多的关注搜索结果的前几名的情况，即搜索结果的前列文档的顺序十分重要，如果前面文档的顺序出错，代价应该明显高于后面的文档顺序出错才对，而简单的文档对方法并没有体现这一点。

4.6.3 文档列表方法(ListWise)

文档列表方法与前面两种方法不同，单文档方法每次只是单独考虑一个文档，文档对方法则是对一次搜索结果的任意两个文档作为训练对象。文档列表方法摒弃了前面两种方法的思想，把一次搜索结果的所有文档看作一个整体，用于作为训练数据。

文档对方法要做的是，根据 N 个训练数据(每个训练数据就是一次查询结果的所有文档)，训练得到一个最优评分函数，对于一个新的用户搜索，用评分函数对每个文档进行打分，即为相关度，最后根据相关度排序即为最终排序结果。

文档列表方法具体的实现算法有很多种，就不一一介绍了，后面会详细介绍到使用的 LambdaMART 模型，就是文档列表方法的具体实现算法之一。

第五章 问答推荐系统研究与设计

前面已经介绍了搜索引擎相关的技术与搜索引擎的一般实现过程和模块之间是如何联系，如何工作的，下面会详细介绍本系统是如何实现一个问题答案搜索系统的。

本系统包括功能和主要任务有:

1. 编写爬虫爬取知乎网问答数据,用于构建训练和测试数据集。
2. 预处理原始数据。
3. 生成搜索相关索引。
4. 使用数据集训练机器模型。
5. 用训练好的机器模型排序，从而实现一个可视化界面。

5.1 代码结构

本系统主要有以下模块，结构如图 5-1 所示：

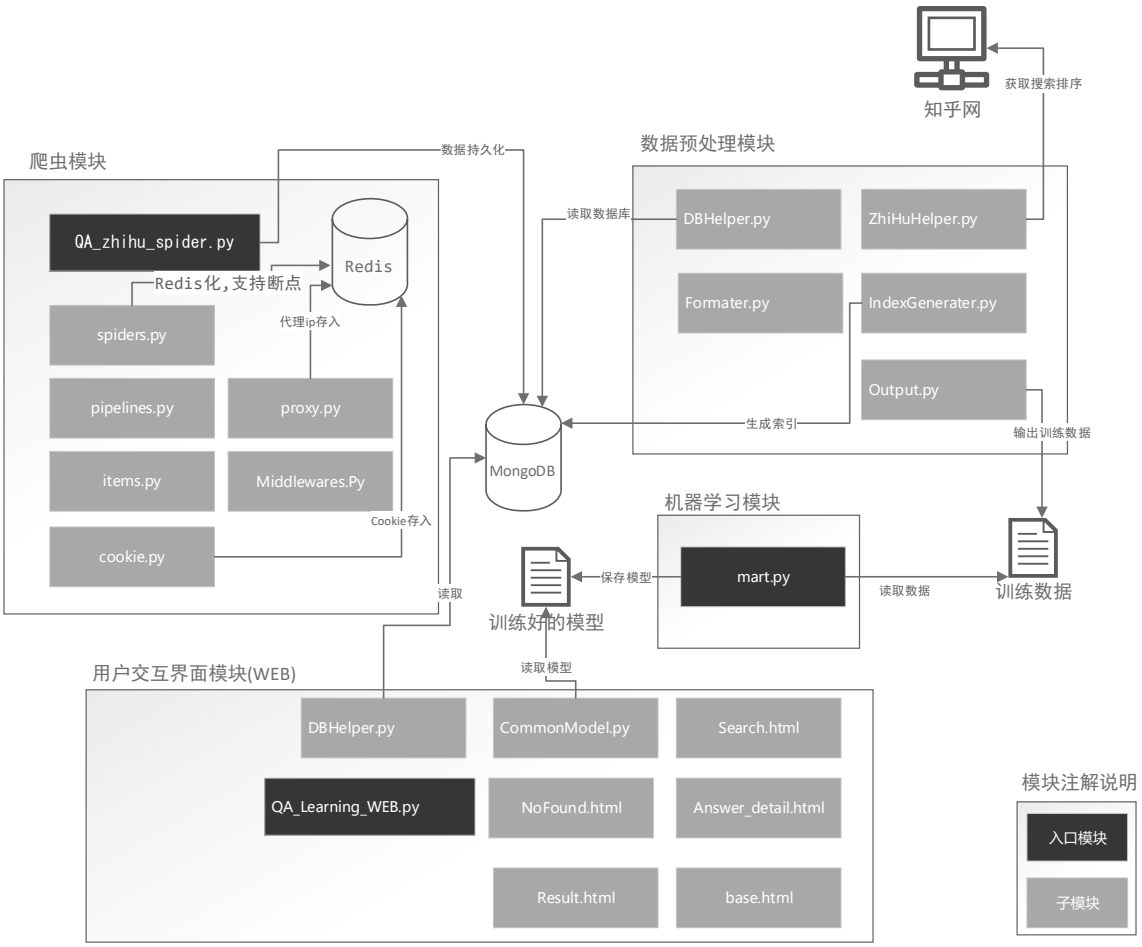

图 5-1 系统模块结构图

1. 爬虫模块
2. 数据预处理模块
3. 机器学习模块
4. 用户交互界面模块

以上模块分为 3 个独立的程序：

1. 爬虫程序：由爬虫模块组成，主要负责爬取知乎网站的问答数据，并存入数据库中。
2. 机器学习程序：由数据预处理模块和机器学习模块组成，主要负责把数据库中的数据预处理后，交给机器学习模型训练得到训练好的模型并生成相应搜索索引。
3. WEB 交互页面：由用户交互界面模块组成，主要负责使用训练好的模型，实现一个可以搜索答案的 WEB 页面。

5.2 数据的爬取

5.2.1 爬虫框架选取

目前现成的爬虫框架中有许多已经比较成熟又易用的框架，所以我采取了直接使用框架的形式编写本项目需要的爬虫。目前市面上的爬虫框架有许多种类：

1. 分布式爬虫： Nutch
2. JAVA 单机爬虫： Crawler4j，WebMagic，WebCollector
3. Python 爬虫： Scrapy[5]，Crawley

优缺点对比：

1. Nutch 本身依赖 hadoop 运行， hadoop 在集群机器数量较少的时候，表现不如单机，我个人只有一台个人电脑，故放弃该框架。
2. Crawler4j， WebMagic， WebCollector 都是比较成熟的 JAVA 爬虫框架，开发难度也不大，但并不支持简单的分布式运行(即，数量少的集群机器情况下)。
3. Scrapy 是 python 下十分成熟的爬虫框架，虽然本身没有分布式的功能，但可以通过简单的添加才实现分布式的简单功能，而且 scrapy 也很好的支持中间件的编写，这样在处理一些爬虫遇到的常见问题的时候，也会更加得心应手。

经过以上的优缺点对比，最后我选择了 scrapy 爬虫框架，不仅是因为这个框架完善的功能，也作为了刚刚开始学习 python 的一个起点，为后面使用 python 写机器学习模块代码积累一些基础。

5.2.2 数据结构设计

首先是数据库的选择，目前主流的两种类型的数据库分别是SQL和NOSQL数据库，本项目本身的存储数据并不复杂，但数据量很大，需要更快的读取速度，需要较为灵活和方便才存储，所以最后决定了选择 NOSQL 型数据库作为本项目的数据库，而 NOSQL 数据库中我选择了使用人数较多的 MongoDB，学习资料完善，有利于初次接触 NOSQL 的开发者更好更快的开发出自己的数据库。

其次是数据结构的设计，首先通过知乎网站的源代码内容的筛选，目前版本的知乎网站，能爬取到的内容如下：

问题相关：

1. 问题 ID
2. 问题标题
3. 问题详细说明
4. 问题创建时间
5. 问题最近更新时间
6. 问题关注者数量
7. 问题作者
8. 问题评论数
9. 问题回答数
10. 问题被浏览次数
11. 问题所属主题

答案相关：

1. 答案 ID
2. 答案回答的原问题的 ID
3. 答案作者
4. 答案创建时间
5. 答案更新时间
6. 答案点赞数
7. 答案详细说明
8. 答案的简述
9. 答案评论数

话题相关：

1. 话题 ID
2. 话题名字
3. 话题详细说明
4. 话题下的子话题列表
5. 话题的父话题

根据以上能爬取到的网站内容，设计相应 python 数据结构，全部以类的形式表示相应对象(存放在 item.py 中)。

5.2.3 爬虫代码设计

通过对知乎网站结构的研究，最终决定采用依据主题为主线，顺藤摸瓜的方式来爬取知乎网站的问答数据。

5.2.3.1 知乎网站结构

1. 问题页面

当前版本的知乎问题页面的 URL 有着统一的格式：

https://www.zhihu.com/question/QID，其中 QID 表示问题 ID。这个页面下包含了所有问题相关的基础信息内容，但无法获取到完整的相关答案信息。需要通过请求特定页面才能获取到答案信息，后面会有详细说明。

2. 答案页面

当前版本知乎答案页面的 URL 也有着统一的格式：https://www.zhihu.com/question/QID/answer/AID，其中 QID 表示问题 ID，AID 表示答案 ID，也可以直接使用 https://www.zhihu.com/answer/AID，会自动跳转到相应 https://www.zhihu.com/question/QID/answer/AID 格式页面。但以上页面并不能获取到当前问题下所有答案。

3. 话题页面

当前版本知乎话题页面的 URL 也有着统一的格式： https://www.zhihu.com/topic/TID，其中 TID 表示话题 ID。

在知乎，所有的问题都会属于一个或者多个话题中，即答案和主题是一对多的关系，而每个话题都包含有若干的子话题，和若干的父话题，知乎中的所有话题是通过父子关系构成的一个如图 5-2 有根的无循环有向图。根据这个无循环有向图，可以做到遍历所有的话题，从而获得每个话题下的问题和答案，这样就可以爬取到网站的问答数据。

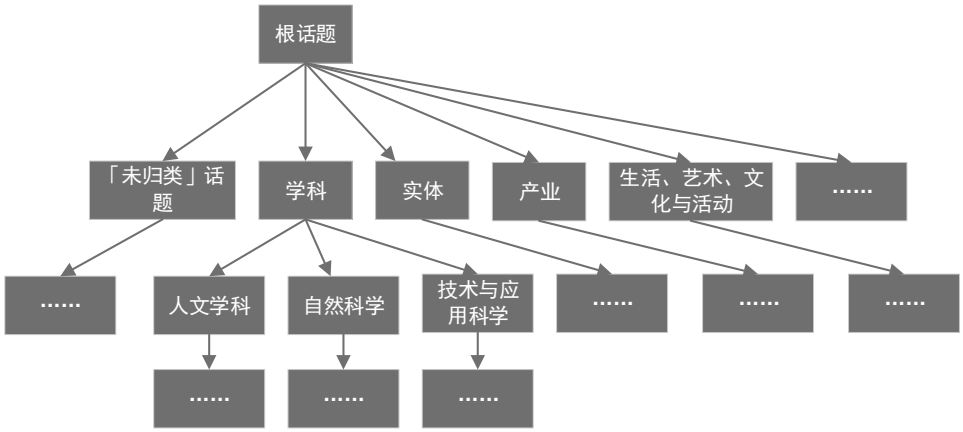

图 5-2 知乎网站话题构成的有根无循环有向图

5.2.3.2 可选的爬取知乎网策略对比

1. 暴力遍历所有问题和答案的 ID 来爬取：这种策略代码实现十分简单，但不可行，首先我们并不知道知乎网问题和答案的 ID 是如何生成的，只能通过穷举法来爬取，这样不仅有大量的无效 ID 使得我们的代码效率十分低下，而且极易被网站的反爬虫检测并使得我们的爬虫很快就失效，综上所述，本方法不可取。
2. 通过首页的信息流来爬取问题和答案：知乎网首页会推送各种各样热门或者符合当前用户的问题和答案等，我们可以通过这个信息流来获取到很多问题和答案，但由于首

页信息流是当前热门问题答案，有很大的局限性，会导致我们爬取到的问题和答案覆盖面很低，只停留在多个话题，而且也只能爬取到最近的一些热门问题和答案，时间范围也很狭窄，综上所述，本方法不可取。

3. 通过话题的无循环有向图遍历：由于整个知乎网站的所有话题都源自于一个叫[根话题]的话题，所以可以通过从根话题遍历的方式遍历完所有的话题，可以采用深度遍历算法或者广度遍历算法，这里优先考虑深度算法，由于每个话题中的热门问答都是来自于其子话题中的热门问答，意味着话题层级越往上，问答信息就越会向热门方向靠拢，这会有一定局限性，而由于基本知乎的问题标签都集中在话题树的叶子节点上，所以采用深度优先算法，更能得到全面的数据。

5.2.3.3 数据获取方法详细说明

1. 话题数据获取

根据前面叙述，知乎现阶段网站中，话题组成了一个有根的无循环有向图，前面论述了采用广度优先算法获取到的数据更能满足我们的要求，但这里由于知乎网站话题中问题的设计原因，我们只能采用深度优先算法，不然会出现大量重复数据，导致程序会花大量时间用来去除重复数据。具体原因是因为知乎网站中，每个话题下的问题，定义为当前话题和当前话题的所有子话题中的问题，所以如果直接遍历较为高层次的话题的话，再遍历其子话题，会出现大量重复问题数据出现，所以只能采用深度优先算法来遍历话题。

起始页面(即根话题)： https://www.zhihu.com/topic/19776749/organize/entire，在页面源代码中，class 为“zm-side-section-inner parent-topic”的 div 里，包含了本话题的所有父话题，该 div 中的 class 为“zm-item-tag”的 div 的内容，即为父话题的文本表示。而 class 为“zm-side-section-inner child-topic”的 div 里。包含了当前话题的所有子话题。依据以上的说明，我们可以利用 xpath 的语法写出爬虫代码，如代码 5-1 所示：

代码 5-1 利用 xpath 获取网页源码信息爬虫代码

```
    def parse_topic(self， response):
        # 当前话题 ID
        now_topic_id = re.match(r"(?:https://www.zhihu.com/topic/)(\d+)(?:/organize/entire)" ，
response.url).group(1)
        # 当前话题名称
        now_topic_name = response.xpath("//h1[@class='zm-editable-content']/text()")[0].extract().strip()
……省略部分代码，省略代码获取了当前话题的描述，子话题，父话题等信息，并保存当前话题信息……
```

```
        yield item
        if len(children_list) != 0:
            # 有子话题
            for c in children_list:
                tid = c.extract().strip()
                yield Request('https://www.zhihu.com/topic/' + tid + '/organize/entire' ,
callback=self.parse_topic)
        # 深度优先，先遍历所有话题，从最底层的子话题开始搜索问题
        if len(children_list) != 0:
            for c in children_list:
                tid = c.extract().strip()
                yield Request('https://www.zhihu.com/topic/' + tid + '/top-answers',
                              callback=self.parse_topic_top_questions)
                yield Request('https://www.zhihu.com/topic/' + tid + '/unanswered',
                              callback=self.parse_topic_top_questions)
        # 还要搜索本身话题的问题(当然，可能重复)
        yield Request('https://www.zhihu.com/topic/' + now_topic_id + '/top-answers',
                      callback=self.parse_topic_top_questions)
        yield Request('https://www.zhihu.com/topic/' + now_topic_id + '/unanswered',
                      callback=self.parse_topic_top_questions)
```

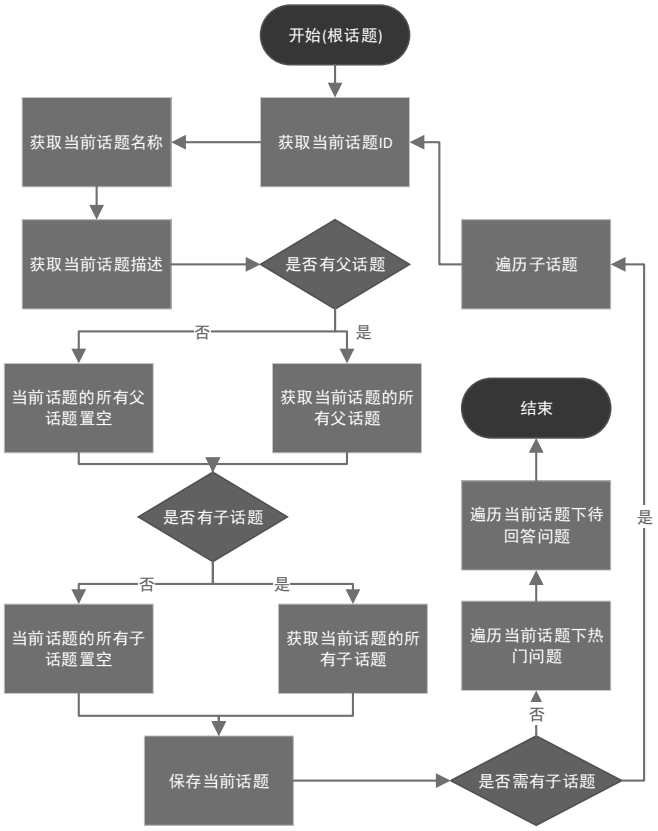

图 5-2 话题爬取程序流程图

以上代码即采用了深度优先遍历算法，遍历了知乎上所有话题，并且对每个话题也做了问题数据的爬取，算法简单流程图如图 5-2 所示。

2. 问题数据获取

在遍历话题的时候，回调了 parse_topic_top_questions 函数，即上面流程图的遍历当前话题下热门话题和遍历当前话题下待回答问题部分，这个函数是用于爬取问题相关数据的具体实现函数。

知乎中，每个话题的问题都会被收纳在两个页面中，分别是精华问题(又称热门问题)和等待回答问题(未回答问题)，但由于精华问题页面和等待回答问题页面组织方式完全相同，所以只需要相同处理即可.我们可以直接遍历每个话题中的这两个分类问题，即可获得当前话题下的所有问题 ID，再根据问题 ID，直接得到该问题详细页面的 URL 进行进一步数据的获取，详细过程如 5-2 代码所示：

代码 5-2 解析话题中问题页面

```
def parse_topic_top_questions(self,  response):
    """
    解析话题精华问题
```

```
        :param response:
        :return:
        """
        questions_url = response.xpath(r"//h2/a[@class='question_link']/@href")
        # 处理当前页的所有问题
        for url in questions_url:
            yield Request('https://www.zhihu.com' + url.extract().strip(),  callback=self.parse_question)
        # 下一页
        next = response.xpath(r"//div[@class='zm-invite-pager']/span[last()]/a/@href")
        if len(next) != 0:
            next_url = next[0].extract().strip()
            yield Request(response.url.split('?')[0] + next_url,  callback=self.parse_topic_top_questions)
```

这个函数完成了上述工作，从上述两种页面中，一页一页分析出当前所有问题的 ID，并回调 parse_topic_top_questions 进行更进一步的数据提取。

在问题的详细页面，根据对页面元素的分析，我们可以很轻易提取到需要的信息，包括标题，创建时间，更新时间，点赞数，浏览次数，评论数等。节选部分代码如代码 5-3 所示，对应简单程序流程图如图 5-3 所示。

代码 5-3 xpath 获取问题详细页面信息

```
        # 问题标题
        question_title = response.xpath(r"//h1[@class='QuestionHeader-title']/text()")[0].extract().strip()
        # 问题详细内容
        question_content = re.search(r"(?:editableDetail\":)(.*)(?:\"visitCount)" , response.xpath(r"//div[@id='data']/@data-state")[0].extract()).group(1)
        # 问题创建时间
        question_create_time = response.xpath(r"//div[@class='QuestionPage']/meta[@itemprop='dateCreated']/@content").extract()[0]
        # 问题被浏览次数
        question_view_count = response.xpath(r"//div[@class='NumberBoard QuestionFollowStatus-counts']/div[@class='NumberBoard-item']/div[@class='NumberBoard-value']/text()").extract()[0]
        # 问题关注者数量
        question_follower_count = response.xpath(r"//div[@class='QuestionPage']/meta[@itemprop='zhihu:followerCount']/@content").extract()[0]
```

```
        # 问题回答数目
        question_answer_count =
int(response.xpath(r"//div[@class='QuestionPage']/meta[@itemprop='answerCount']/@content").extract()[0]
)
        # 问题评论数目
        question_comment_count =
response.xpath(r"//div[@class='QuestionPage']/meta[@itemprop='commentCount']/@content").extract()[0]
        # 问题所属话题 ID 列表
        topic_list = response.xpath(r"//a[@class='TopicLink']")
……
```

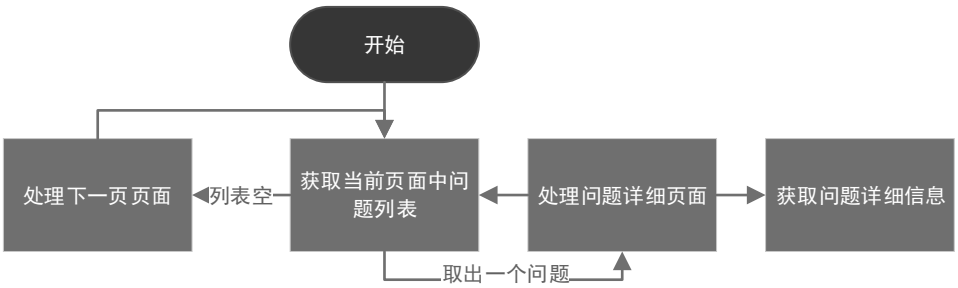

图 5-3 问题爬取流程图

3. 答案数据获取

在获取问题相关数据的时候，回调了 parse_answer，而这个函数是用来获取回答当前问题的答案数据的函数，这里对于答案数据的获取和前面的通过 HTML 源代码的获取的方式有所不同。问题和话题的获取，都是通过页面的源代码根据 xpath 语法获取到相应文本。而答案数据的获取有所不同，这里的 more_answer 实际上是一个 ajax 发出的获取数据的 url，根据这个基础 URL，加上问题 ID 和相关参数后，服务器会返回一个 json 格式的数据，这个 json 格式数据包含了多个答案的详细信息内容。

Json 数据格式如下：

```
{
    ...
    Data:[
        {...
            id:答案 ID,
            author:作者 ID,
            question:{
                id:问题 ID,
```

```
                ...}
            created_time:答案创建时间,
            updated_time:答案更新时间,
            voteup_count:答案点赞数,
            comment_count:答案评论数,
            excerpt:答案简介,
            content:答案详细内容,
            ...}, ...
    ]
    ...
}
```

根据以上返回的 json 数据，我们可以很简单的转换为要保存的对象，相关代码略。

5.2.4 爬虫中间件设计

爬虫在运行中，会遇到很多问题，需要我们增加额外的代码来处理相关问题，其中最常见的问题就是当网站服务器检测到某 IP 的访问行为很可能是爬虫，会被网站禁止访问或者做出其他相应手段。为了解决这一问题，我主要采取的是使用代理 IP 进行数据爬取的方式，利用海量代理 IP 池，当某 IP 被 Ban 时候，从代理池中获取其他 IP 即可。除了 Ban 的问题外，由于知乎网站的特殊性，我们必须保持用户的持续登录状态，才能爬取到数据，所以我们还需要保存用户的 cookie 相关。

在 Scrapy 框架中，要处理一些非爬取数据问题相关问题的时候，只需要添加中间件即可，这样可以具有很良好的模块化，使得我们的代码耦合性更加低，更具扩展性。

5.2.4.1 代理中间件

1. 首先我们需要获取代理 IP，相关代码如代码 5-4 所示：

代码 5-4 代理 IP 获取

```
def initIPPOOLS(rconn):
    """把有效的 IP 存入  REDIS 数据库"""
    ipNum=len(rconn.keys('IP*'))
    if ipNum<IPPOOLNUM:
        IPPOOLS=GetIPPOOLS(IPPOOLNUM)
        for ipall in IPPOOLS:
```

```
            try:
                ip=ipall.split(':')[0]
                port=ipall.split(':')[1]
                telnetlib.Telnet(ip，port=port，timeout=2) #检验代理 ip 是否有效
            except:
                logger.warning("The ip is not available !( IP:%s )" % ipall)
            else:
                rconn.set("IP:%s:10"%(ipall)，ipall)        #10 is status
    else:
        logger.warning("The number of   the IP is %s!" % str(ipNum))
```

以上代码通过从 GetIPPOOLS 函数中获取代理 IP。并检测代理 IP 有效性，将有效的 IP 存入 redis 数据库，方便后面代码的取用。 GetIPPOOLS 函数只是简单的从服务商中发包获取代理 IP 而已，我这里使用的是大象代理。

2. 然后需要从 redis 数据库中获取存入的代理 IP，取出并使用，相关代码(middlewares.py)如代码 5-5 所示：

代码 5-5 代理中间件

```
def process_request(self， request， spider):
        ipNum = len(self.rconn.keys('IP*'))
        if ipNum < 50:
            proxy_thread = threading.Thread(target=initIPPOOLS， args=(self.rconn，))
            proxy_thread.setDaemon(True)
            proxy_thread.start()
        if self.TIMES == 3:
            baseIP = random.choice(self.rconn.keys('IP:*'))
            ip = str(baseIP， 'utf-8').replace('IP:'， '')
            try:
                IP， PORT， status = ip.split(':')
                request.meta['status'] = status
                telnetlib.Telnet(IP， port=PORT， timeout=2)  # 测试 ip 是否有效
            except:
                updateIPPOOLS(self.rconn， IP + ':' + PORT， status)
            else:
                self.IP = "http://" + IP + ':' + PORT
```

```
                self.TIMES = 0
                updateIPPOOLS(self.rconn， IP + ':' + PORT， status， 1)
        else:
            self.TIMES += 1
        if self.IP is not "":
            request.meta["proxy"] = self.IP
```

process_request 函数在每次爬虫要爬取页面的时候，都会调用，作用类似 Hook，如果 redis 数据库中 IP 池不够 50，则继续获取。如果当前爬虫重试次数达到 3 次以上，则更换 IP。更换 IP 前也需要检查 IP 可用性，最后，在请求头的 proxy 字段，添加我们的代理 IP 即可。

5.2.4.1 cookie 中间件

Cookie 中间件的设计，依旧采用了 redis 数据库作为存储方便爬虫读取。cookie 获取的代码详见 cookie.py，cookie 的中间件详见 middlewares.py 中的 CookiesMiddleware 类，由于代码简单，这里不详细说明。

5.3 数据预处理

爬虫模块已经将数据存入了 MongoDB 数据库，接下来需要对爬取到的原始数据做一些预处理。

5.3.1 文本特征值提取

爬取到的原始数据中，大部分数据都是整数形式，但存在一些文本形式的内容，包括：

- 问题中的 title 字段，content 字段
- 答案中的 excerpt 字段，content 字段
- 话题中的 name 字段，desc 字段

暂时不使用话题相关数据，所以暂时只提取问题和答案的相关文本特征值。

1. 去除无用 html 标签

由于在爬取原始文本数据的时候，存在一些无用的 html 标签，不属于有效文本，应该去除，包含需要剔除的 html 标签的文本包括：问题中的 content 字段，答案中的 content 字段.核心检测并去除 html 标签代码如代码 5-6 所示：

代码 5-6 去除 html 无用标签

```
def del_html(content):
    """
    删除 html 标签，返回删除后文本
    :param content: 原始文本
    :return: 删除标签后文本
    """
    re_html = re.compile('</?\w+[^>]*>')
    s = re_html.sub('', content)
return s
```

采用了一个正则表达式，用于匹配所有 html 标签格式的文本，然后用 sub 函数，剔除从左开始，第一个符合正则表达式的本文，即剔除最外围的 html 标签，由于我们需要保存一些其他的 html 标签，因为正文中的标签可能包含了用户的意图，比如文本加粗，图片 URL 等，诸如此类的标签我们需要保存，所以只去除了最外围的 html 标签。

2. 关键词提取

关键词的提取，我采用了广泛被使用十分实用的算法 TF-IDF，用于提取文本形式数据的关键词信息。这里使用了 python 的第三方库结巴[6].代码如代码 5-7 所示：

代码 5-7 关键词提取

```
def handler_each_question(question):
    # 问题 ID
    qid = question['question_id']
    # 问题内容文本
    title = question['title']
    content = question['content']
    all_info = []
    title_info = jieba.analyse.extract_tags(title, topK=8, withWeight=True, allowPOS=['ns', 'n', 'vn', 'v', 'nr'])
    title_words = [i[0] for i in title_info]
    title_info.sort(key=lambda d: d[1])

    if len(content) > 1:
        # 获取 10 个关键词
        content_info = jieba.analyse.extract_tags(content, topK=10, withWeight=True,
                                                  allowPOS=['ns', 'n', 'vn', 'v', 'nr'])
        content_info.sort(key=lambda d: d[1])
```

```
    #       把标题和说明的关键词融合成 10 个
    all_info.extend(title_info)
    if len(all_info) < 10:
        need = 10 - len(all_info)
        for info in content_info:
            if need != 0:
                if info[0] not in title_info:
                    all_info.append([info[0], info[1] * SHRINK_PARM])
                    need -= 1
return qid, word_info_to_id_info(all_info)
```

代码从问题的 title 字段提取 8 个关键词，从问题的 content 字段提取了 10 个关键词，然后以 title 中的关键词为主，用 content 的关键词来补全 10 个关键词。同时这一步，我们也去除了停用词，allowPOS 参数允许我们设置仅保留的词性，即可去除停用词。

其中 word_info_to_id_info 函数，将关键词转换为关键词 ID，具体逻辑是，当关键词不在数据库中时，将生成一个唯一 ID 并返回，如果数据库中已经存在关键词，则查询对应 ID 并返回。关键词提取程序流程图如图 5-4 所示：

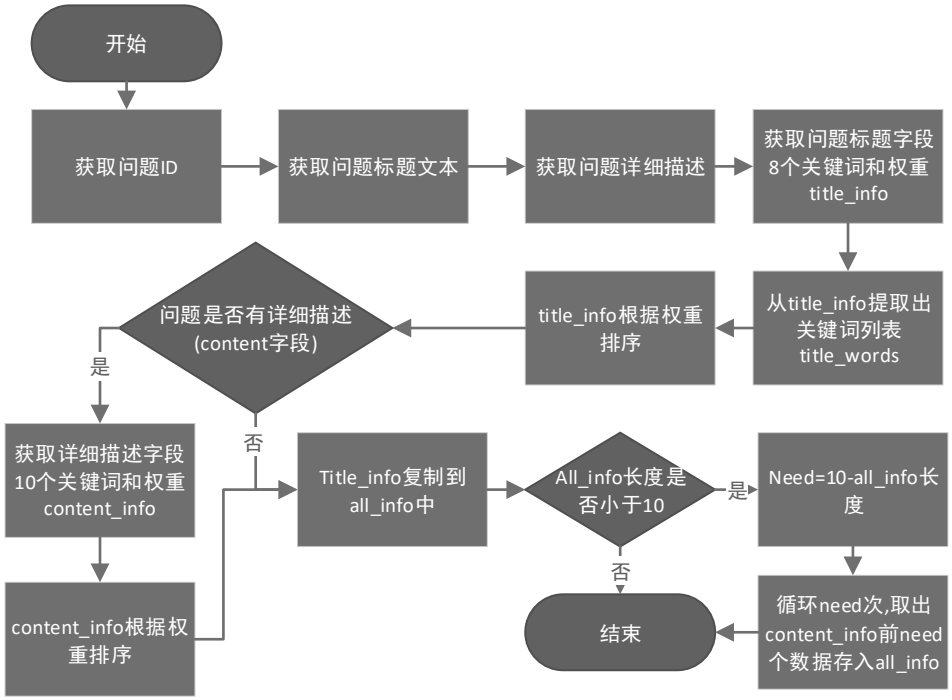

图 5-4 问题关键词提取流程图

5.3.2 特征值规格化

有些提取出来的特征值会出现取值特别大，需要进行一些标准化或者正则化的方式，

使得特征值缩小到一定范围，或减少方差等，因为有的模型对于数据的范围敏感，所以一般我们都需要对较大数据范围的数据或者较零散的数据进行预处理。

我的特征值中，有如下几个特征值需要处理：

1. 答案创建时间
2. 答案更新时间
3. 答案点赞数
4. 答案评论数
5. 答案长度

针对某个问题，可能存在有些问题本身就是热门问题，有些问题本身就是冷门问题，而导致的答案投票数上有很大差异的情况，比如关于人生的问题中可能会出现某答案点赞数超过几万，而一些冷门问题，比如一些专业性极强的问题中，可能会出现最高票答案只有十几票的情况所以我们需要让这些特征值更具相对性，从而避免最后训练出来的模型只会让几万的高票答案排序靠前的情况。也为了让以上特征值缩小范围，所以采用了 MinMaxScaler 的预处理方式。如图 5-5 是以答案点赞数为例子的算法程序流程图，其他特征值同理。

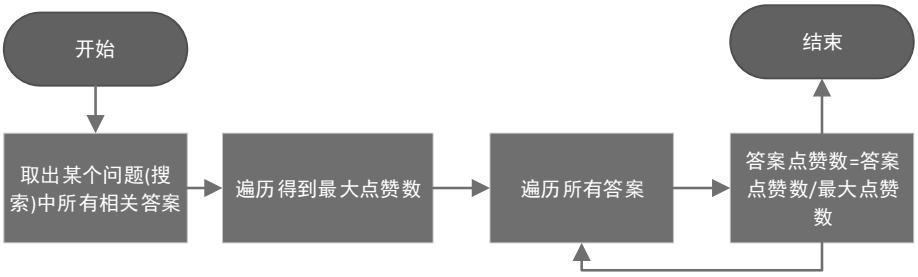

图 5-5 答案点赞数预处理流程图

经过以上预处理后，相关特征值的范围将会保证在 0 到 1 之间。

5.3.3 建立正向索引

答案正向索引，只用于生成答案逆向索引，索引项结构见表 5-1：

表 5-1 逆向索引结构

项	类型	说明
answer_id	整数	答案 ID
content_info	数组	格式：[关键词 ID，TF-IDF 值]，表示当前答案中 TF-IDF 值前十的关键词信息

生成正向索引核心代码如代码 5-8 所示：

代码 5-8 生成正向索引

```
def generate_forward_index():
    """
    生成问题正向索引，并存入数据库
    :return:
    """
    db_helper = DBHelper.get_instance()
    r = db_helper.get_all_answers(db_helper.get_forward_index_count())
    # 重新生成索引
    for answer in r:
        res = handler_each_answer(answer)
        db_helper.add_forward_indexes(res[0]， res[1])
r.close()
```

正向索引生成支持断点生成，首先获取当前已经生成的正向索引数量，由于正向索引和答案是一一对应关系的，所以生成的正向索引数量应该等于答案数量，所以可以接着之前的生成的部分，继续生成索引。其中 res 即为索引项。

其中提取索引项核心代码位于 handler_each_answer 下，对答案的文本，我们也去除了停用词，并提取了 15 个关键词。

根据以上的答案正向索引，我们可以很容易的查询到某个答案的所有关键词信息和相应的 TF-IDF 值。为后面的逆向索引的生成，提供了更快的查询方式。

5.3.4 建立逆向索引

有了正向索引，接下来可以生成答案的逆向索引。逆向索引的结构见表 5-2：

表 5-2 逆向索引结构

项	类型	说明
key_word_id	整数	关键词 ID
content	数组	格式：[文档 ID，关键词 TF-IDF 值]，表示所有包含该关键词的文档信息

逆向索引的代码比较简单，只是将正向索引换了一种组织方式，核心代码如代码 5-9 所示：

代码 5-9 生成逆向索引

```
def generate_reverse_index():
    """
    生成反向索引(前提是已经生成好了正向索引)
    :return:
    """
    db_helper = DBHelper.get_instance()
    r = db_helper.get_forward_indexes()
    for f_index in r:
        a_id = f_index['answer_id']
        content_info = f_index['content_info']
        if content_info:
            for content in content_info:
                db_helper.add_reverse_index_content(content[0], a_id, content[1])
r.close()
```

从数据库中，逐行读取正向索引，重新写入逆向索引即可，其中 add_reverse_index_content 代码如代码 5-10 所示：

代码 5-10 添加逆向索引

```
    def add_reverse_index_content(self, key_word_id, answer_id, weight):
        """
        :param key_word_id:索引关键字 ID
        :param answer_id:答案 ID
        :param weight:关键字对应权重
        :return:
        """
        self.reverse_index.update_one({"word_key_id": key_word_id} , {"$addToSet": {"content":
[answer_id, weight]}}, upsert=True)
```

采用 update 形式修改数据库， upsert=True 可以确保在数据库中没有相应条目的时候，会直接将数据插入数据库， addToSet 是 MongoDB 的一种特殊插入方式，表示把当前项中的指定 key 对应的 val 当作数组来对待，对该数据插入数据。

生成了逆向索引之后，我们就可以很方便的利用逆向索引，进行搜索结果的初始子集的选取了，我们可以通过关键词，直接查询到包含该关键词的所有答案 ID，而不需要

遍历所有答案进行搜索，然后我们就可以对这些答案进行评分，最后根据评分排序了。

5.3.5 相关度标记

问题和答案的数据准备工作前面已经基本完成，接下来我们需要人工评分，即为机器学习提供标签。这一步我直接使用了知乎网的搜索结果作为排序结果，关于知乎搜索结果的代码实现，我使用了 github 上一个非官方的接口 zhihu-oauth[12]，用于获取知乎网实际搜索结果排序情况。

经过以上步骤，我们的数据准备工作基本完成，接下来需要利用机器学习，拟合我们现有的数据。

5.4 机器学习模型

机器学习中，专门用于排序的领域叫 LTR，包含了三大类方法：单文档方法，文档对方法，文档列表方法，其中文档列表方法表现较好。

机器学习模块部分的代码，依旧采用 python 语言，三方库采用了目前主流且相关资料比较多的 scikit-learn[5]方便代码的书写和对机器学习的相关学习。

5.4.1 LambdaMART 模型介绍说明

机器学习中模型多种多样，经过选择，最后可选取的模型如下：

1. 单文档方法：线性回归模型 ElasticNet，根据 scikit-learn 的文档说明，和当前数据的大小，特征值数量，最终选择的线性回归模型。
2. 文档列表方法： LambdaMART，在文档列表方法中，表现十分优秀的算法之一。

最终参考相关文献， LambdaMART 的表现更佳，最终决定采用 LambdaMART 作为训练模型。下面简单介绍一下 LambdaMART 模型

LambdaMART[2]是 ListWise 算法之一，出自微软的 Chris Burges 之手，LambdaMART 分为两个部分，底层的训练模型采用 MART(又称 GBDT)，求解的梯度采用 Lambda。

Lambda 最初在 LambdaRank 模型中被提出，Lambda 指的是下次迭代的方向和强度，即梯度。

对于每个文档d_iLambda 为$\lambda_{\mathrm{i}}=\sum_{\mathrm{j(i,\ j)}\in I}\lambda_{ij}-\sum_{\mathrm{j(j,\ i)}\in I}\lambda_{ji}$，也就是说，每个文档的梯度取决于其他同一查询中的其他不同 label 的文档。

MART 的原理是，直接在函数空间中对函数求解，求解结构由若干课树组成，每棵树拟合的目标就是损失函数的梯度，而在 LambdaMART 中，就是 Lambda。具体算法伪代码表示如下：

Procedure LambdaMART($\{\vec{x}, y\}$, N, L, η)

For i: o -> $|\{\vec{x}, y\}|$ do

$F_0(\vec{x}_i) = BaseModel(\vec{x}_i)$

End for

For k: 0-> N do

For i: 0 ->$|\{\vec{x}, y\}|$ do

$y_i = \lambda_i$

$w_i = \frac{\partial y_i}{\partial \overrightarrow{F_{k-1(\lambda)}}}$

End for

$\{R_{Lk}\}_{l=1}^{L}$

For l: 0 -> L do

$\gamma_{lk} = \frac{\sum_{\vec{x}_i \in R_{lk}} y_i}{\sum_{\vec{x}_i \in R_{lk}} w_i}$

End for

For I : 0 ->$|\{\vec{x}, y\}|$ do

$F_k(\vec{x}_i) = F_{k-1}(\vec{x}_i) + \eta \sum_l \gamma_{lk} I(\vec{x}_i \epsilon R_{lk})$

End for

End for

procedure

最终选择 LambdaMART 是因为：

1. 适用于排序场景，直接求解，而非传统的通过分类或者回归实现。
2. 损失函数可导： 通过损失函数的转换，将类似于 NDCG 这种无法求导的评价指标转换成可以求导的函数。
3. 组合特征：因为采用树模型，因此可以学到不同特征组合情况
4. 特征选择：因为是基于 MART 模型，因此也具有 MART 的优势，可以学到每个特征的重要性，可以做特征选择。
5. 适用于正负样本比例失衡的数据：因为模型的训练对象具有不同 label 的文档 pair，而不是预测每个文档的 label，因此对正负样本比例失衡不敏感。

5.4.2 特征值选取

考虑到问题和答案关联性较大，排序的时候，除了要有答案本身的特征值之外，还应该需要该答案对应的问题相关的特征值，这样才能让模型更好的拟合他们之间的关系。如：当用户搜索“编程如何入门”的时候，我们不仅需要查找相关的答案，我们也应该寻

找有关问题对应的答案，如“编程入门书籍有哪些”等问题所对应的答案。

特征值的选定由人工完成，包括答案相关特征值和问题相关特征值，见表 5-3 和表 5-4：

表 5-3 答案相关特征值列表

特征值	类型	范围	预处理方式	详细说明
Create_time	浮点数	[0,1]	MinMaxScaler	答案创建的时间
Update_time	浮点数	[0,1]	MinMaxScaler	答案更新的时间
Comment_count	浮点数	[0,1]	MinMaxScaler	答案评论数
Voteup_count	浮点数	[0,1]	MinMaxScaler	答案点赞数
Len	浮点数	[0,1]	MinMaxScaler	答案文本长度
Ans_key_word_hit	整数	正整数	无	答案关键词中和搜索关键词匹配个数，即答案命中关键词个数
Ans_key_word_hit_query_tfidf	浮点数	正实数	无	答案命中关键词和搜索关键词的 TF-IDF 相乘所得，共 10 个，如 Ans_key_word_hit 小于 10，那么后面补 0

表 5-4 问题相关特征值列表

特征值	类型	范围	预处理方式	详细说明
Ques_key_word_hit	整数	正整数	无	当前答案所回答的问题中关键词和搜索关键词匹配个数，即问题命中关键词个数
Ques_key_word_hit_query_tfidf	浮点数	正实数	无	答案对应问题的关键词和搜索关键词的 TF-IDF 相乘所得，共 10 个

综上所述，特征值一共 27 个。以上特征值，不仅考虑到了答案自身的相关属性，也考虑到了答案所回答问题的相关属性。

5.4.3 模型的训练

采用 scikit-learn 预设的树模型，可以有效简化代码复杂度，核心代码如代码 5-11 所示：

代码 5-11 模型训练函数

```
def learn(train， n_trees=10， learning_rate=0.1， k=10， validate=False):
    scores = train[:， 0]
    queries = train[:， 1]]
    features = train[:， 2:]
    ensemble = Ensemble(learning_rate)
    model_output = np.zeros(len(features))
    time.clock()
    for i in range(n_trees):
        start = time.clock()
        lambdas = compute_lambdas(model_output， scores， queries， k)
        start = time.clock()
        tree = DecisionTreeRegressor(max_depth=6)
        tree.fit(features， lambdas)
        ensemble.add(tree)
        prediction = tree.predict(features)
        model_output += learning_rate * prediction
        start = time.clock()
        train_score = score(model_output， scores， queries， 10)
    print "Finished sucessfully."
return ensemble
```

首先将传入的 train 对象拆分为 score(评分)，query(查询 ID)，feature(特征值)三个部分， Ensemble 即是算法中的若干树的对象，然后根据子树数量开始循环，首先先计算 Lambda 的值，然后用 scikit-learn 库中自带的决策树作为子树，然后让决策树拟合 Lambda，将拟合后的树加入 Ensemble，然后根据当前决策树的预测结果，可以算出 model_output 作为下个子树计算 Lambda 的输入。所有子树拟合完毕后，训练过程结束。

5.5 用户交互界面

为了方便使用机器学习模块训练出来的模型，所以用户交互页面也选择了 python 作为基础语言，采用了主流的 Flask[8]框架来实现 web 的后台，前端采用了简单实用的 bootstrap[7]作为 web 显示和交互。

5.5.1 搜索主页

搜索主页包含的组件有:

1. 用户搜索框
2. 搜索按钮
3. 导航栏
4. 作者信息和相关说明

后台调用相关函数：

1. result：负责调用模型，得到搜索答案排序列表后，渲染页面并返回给用户显示，代码如 5-12 所示：

代码 5-12 答案排序结果页面

```
def result():
model = ModelA(r'C:\Users\qianzise\Desktop\LambdaMart-master\model_zhihu2')
answer_id_list=model.getBestAnswerList(request.Form['question'])

if answer_id_list is None or len(answer_id_list)==0:
    return render_template('NoFound.Html', search_question=request.Form['question'])
answer_list = get_answers_by_ids(answer_id_list[0:100])
return render_template('Result.Html',  resultList=answer_list)
```

调用训练好的模型，搜索列表如果为空，则渲染 NoFound 页面，否则渲染 Result 页面返回给用户显示。其中 getBestAnswerList 为核心函数如代码 5-13 所示：

代码 5-13 获取答案排序结果

```
def getBestAnswerList(self,  question_title):
    self.Question_title = question_title
    self.Db = DBHelper.Get_instance()
    tags  =  jieba.Analyse.extract_tags(question_title  ,  topK=self.Question_tag_len  ,
```

```
withWeight=True，allowPOS=[‘ns’， ‘n’， ‘vn’， ‘v’， ‘nr’])
        # 问题关键词 ID 列表
        self.Search_ques_key_word_ids = [self.Db.get_key_word_id(t[0]) for t in tags]
        self.Search_ques_key_word_weights = [t[1] for t in tags]

        while len(self.Search_ques_key_word_ids) < self.Question_tag_len:
            self.Search_ques_key_word_ids.Append(0)
            self.Search_ques_key_word_weights.Append(0)
        answer_list = self._getAboutAnswer()
        question_list = self._getAboutQuestion()
        # 特征值列表，每个数组第一位表示答案 ID
        feature = []
        for ans in answer_list:
            afeature = self._getAnswerFeature(ans)
            if afeature[1] in question_list:
                #                   答案回答的问题在相关问题中，
                _question_feature = self._getQuestionFeature(afeature[1])
                # 答案 ID
                _f = [afeature[0]]
                _f.extend(afeature[2:])
                _f.extend(_question_feature[2:])
                _f.append(_question_feature[1])
                featureX.Append(_f)
            else:
                _question_feature = [0 for I in range(self.Question_feature_len)]
                _f = [afeature[0]]
                _f.extend( afeature[2:])
                _f.extend(_question_feature)
                featureX.Append(_f)
        feature=np.Array(feature)
        feature=self._data_preprocess(feature)
        _ids，_featureX=np.Hsplit(featureX，(1，))
        scores=self.Predict(_featureX)
        score_list = np.Hstack((array(list(_ids.Astype(np.Int32)))， array(scores.Reshape(-1， 1))))
```

```
        sl = list(score_list.Tolist())
        sl.Sort(key=lambda d: d[1],  reverse=True)
return sl
```

首先将浏览器传入的搜索问题去除停用词并提取关键词，得到关键词列表，然后读取数据库中关键词表，将关键词转换为关键词 ID，然后调用_getAboutAnswer 获取初始子集，即所有相关的答案，后面会详细说明，然后调用_getAboutQuestion，获取相关问题列表，用于计算相关特征值，然后循环初始子集，计算相关特征值。最后将特征值输入模型，得到预测结果，排序后返回。下面是特征值提取算法流程图：

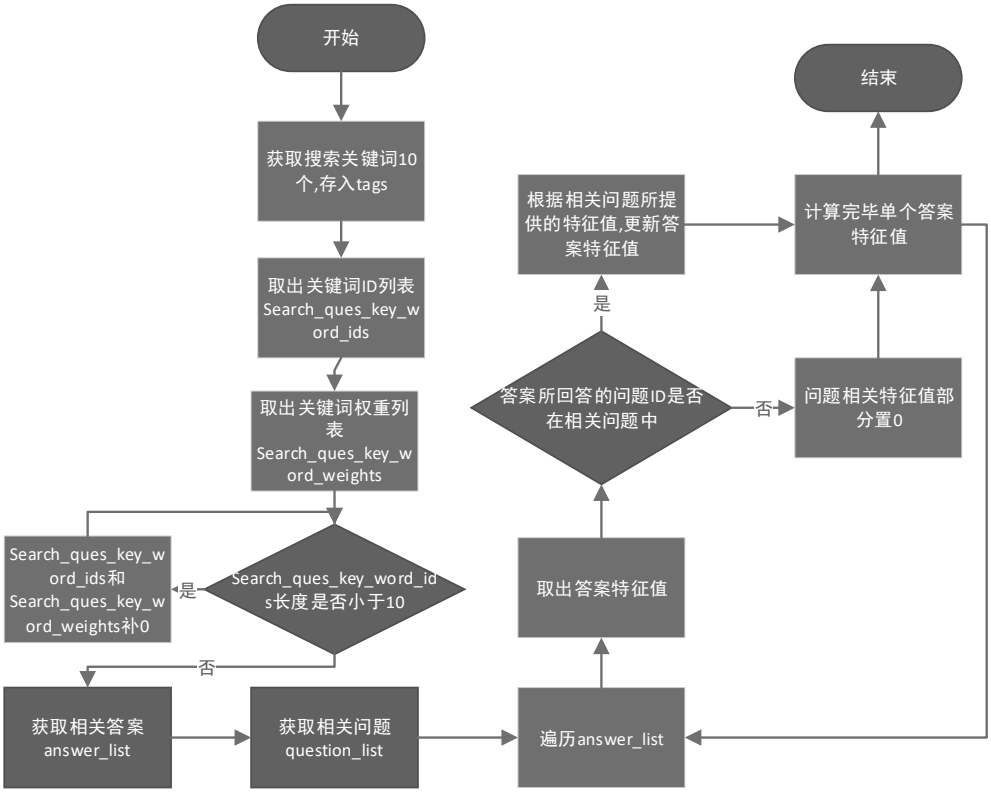

图 5-6 特征值提取流程图

由于数据库中存有的答案数据量很大，我们排序的时候不可能都一一评分，需要筛选，即初始子集选取，相关代码如代码 5-14 所示：

代码 5-14 初始子集选取

```
    def _getAboutAnswer(self):
        “””
        获取相关答案
        :return:  相关答案
        “””
```

```
        ans_id_list = []
        for qid in self.Search_ques_key_word_ids:
            # 有关键词
            o = self.Db.get_answers_by_key_word_id(qid)
            for content in o:
                ans_id_list.Append(content[0])
        ans_id_list = set(ans_id_list)
        ans_list = []
        for ans_id in ans_id_list:
            ans_list.Append(self.Db.get_answer_by_id(ans_id))
return ans_list
```

这里的初始子集的选取，简单的采取了并集的处理方式，即选择包含任意关键词的所有答案(非重复)。

5.5.2 搜索结果

搜索结果页面包括元素：

1. 答案列表：
   a) 答案缩略内容
   b) 答案创建时间
   c) 答案更新时间
   d) 答案评论数
   e) 答案点赞数

后台调用相关函数：

Answer：负责检索数据库中答案详细信息，渲染后返回给用户显示，代码如 5-15 所示：

代码 5-15 获取答案详细信息

```
def answer(answer_id):
    ans = db.get_answer_by_id(answer_id)
    ans['update_time'] = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(ans['update_time']))
    ans['create_time'] = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(ans['create_time']))
    ques = db.get_question_by_id(str(ans['question_id']))
return render_template('Answer_detail.html', answer=ans, question=ques)
```

5.5.3 答案详情

答案详情页面包括元素：

1. 答案详细内容
2. 答案创建时间
3. 答案更新时间
4. 答案评论数
5. 答案点赞数

相关代码略

5.6 优化

本系统中存在许多需要优化的地方，经过优化，能使得我们的代码运行的更加高效更加快速。下面举例两个可优化的地方。

5.6.1 爬虫优化

爬虫在运行过程中，经常会遇到被服务器 BAN，或者其他一些不可抗力因素而终止爬取，不管防范措施做的如何到位，几乎不可能只运行一次就把大量的数据一次性爬取到，如果我们的爬虫没有经过任何优化，那么就需要从头重新运行爬虫，而之前已经爬取的数据将被舍弃，这是一种效率极为低下的爬取方式，为了解决这个问题，我们必须要优化一下爬虫，使得我们的爬虫支持断点工作，即爬虫运行时，可以继续上次未完成的工作继续爬取数据。

我们可以将需要爬取的页面队列保存起来，这样就算爬虫终止，只需要在爬虫启动的时候，重新读取保存起来的队列即可。依据这个思路，我采用 Redis 数据库做持久化，之所以选择 Redis 是因为这个数据库的数据直接存在内存中，读取速度极快，而我的队列数据量并不大，所以很适合使用 Redis。

根据 OOP 设计原则，爬虫支持断点功能代码，写在基类 RedisSpider 中，而无需修改原来的爬虫代码。详细代码在 spider.py 中，核心代码如 5-16 所示：

代码 5-16 支持断点爬虫核心代码

```
    def next_requests(self):
        """Returns a request to be scheduled or none."""
        use_set = self.settings.getbool('REDIS_START_URLS_AS_SET', defaults.START_URLS_AS_SET)
```

```
        fetch_one = self.server.spop if use_set else self.server.lpop


        found = 0
        while found < self.redis_batch_size:
            data = fetch_one(self.redis_key)
            if not data:
                break
            req = self.make_request_from_data(data)
            if req:
                yield req
                found += 1
            else:
                self.logger.debug("Request not made from data: %r", data)
        if found:
            self.logger.debug("Read %s requests from '%s'", found, self.redis_key)
```

next_requests 函数功能是获取下一个爬取页面请求，会随机从 Redis 数据库获取到相应 key 对应的 val，即保存的 URL 列表中的一项，如代码 5-17 所示：

代码 5-17 next_requests 函数

```
    def make_request_from_data(self, data):
        url = bytes_to_str(data, self.redis_encoding)
        return self.make_requests_from_url(url)
```

make_request_from_data 函数负责将存储在 redis 数据库中的二进制数据转换成 URL 后再转为一个 requests 请求。

5.6.2 数据库优化

数据库中不仅存储了大量爬虫爬取下来的数据，而且在爬取过程中，还需要频繁的读写操作，需要有较高读取性能才能保证代码的流畅运行。最简单的方式就是给表加上索引项。

AnswerInfo:

功能说明：存储了所有爬虫爬取到的原始答案数据，在爬虫爬取数据阶段，需要大量写该表，在预处理阶段，需要快速读取指定答案相关信息。

采取措施：数据爬取阶段无相应措施，在预处理阶段，添加索引项 answer_id。索引

创建代码：

db.AnswerInfo.ensureIndex({answer_id：1})

QuestionInfo：

功能说明：存储了所有爬虫爬取到的原始问题数据，在爬虫爬取数据阶段，需要大量写该表。在预处理阶段，需要快速读取某问题相关信息。

采取措施：数据爬取阶段无相应措施，在预处理阶段，添加索引项 question_id，索引创建代码：

db.QuestionInfo.ensureIndex({question_id：1})

KeyWord：

功能说明：存储了关键词和关键词 ID 的对应关系，在预处理阶段生成。需要大量读写。

采取措施：添加索引项 id，索引创建代码：

db.KeyWord.ensureIndex({id：1})

ForwardIndex：

功能说明：存储了答案正向索引，和答案有一对一关系，在预处理阶段生成，在生成逆向索引的时候，需要大量读特定答案的正向索引。

采取措施：添加索引项 answer_id，索引创建代码：

db.ForwardIndex.ensureIndex({answer_id：1})

ReverseIndex：

功能说明：存储了答案的逆向索引，在搜索阶段需要大量的读特定 ID 的关键词相关文档列表。

采取措施：添加索引项 key_word_id，索引创建代码：

db.ReverseIndex.ensureIndex({key_word_id：1})

经过以上索引，数据库访问的速度明显加快，也减少了硬盘读取压力。

第六章 系统测试

6.1 测试环境

测试电脑采用的是我个人的笔记本电脑，电脑配置如表 6-1：

表 6-1 测试环境配置

操作系统	Windows 10 专业版 (Build 16299), 64-bit
主板	Notebook W35xSS_370SS
处理器	Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz 四核
内存	16.00 GB
硬盘 1	WDC WD10JPVX-22JC3T0(1000GB)
硬盘 2	intelssdsc2cw120a3INTEL SSDSC2CW120A3(120GB)
浏览器	Firefox 60.0.1 (64 位)

6.2 测试用例

1. 先测试搜索数据库内现存问题的情况：

测试用例 1，见表 6-2：

表 6-2 测试用例 1

输入	北京有哪些靠谱的搬家公司?
输出 1	答案 ID： 261817513 内容： 之前在北京神马搬家公司平台上找过 2 次，都还算靠谱….也是朋友介绍的…
输出 2	答案 ID： 265551165 内容： 今天无意中浏览到这个帖子，本人在北京工作八年，跟搬家公司也打过几次交道，明白其中的一些潜规则。所以今天跟大家分享一下我的经验…
输出 3	答案 ID： 12340610 内容： 四通搬家。凡是有背景的物流公司，都能提供常规搬家服务之外的便利。比如小货车横穿长安街，交警会假装没看见。
输出 4	答案 ID：127465040 内容： 刚用的 hellomover。 在路上看见他家的货车，觉得干净….
…	…

说明	从第 7 个结果开始，搜索到的答案开始与搬家无关，与关键词公司有关，初步判断是库中数据不够多的缘故

测试用例 2，见表 6-3：

表 6-3 测试用例 2

输入	人类文明最有可能被什么毁灭
输出 1	答案 ID： 26009101 内容： 外星生物入侵 虽然听着很扯淡，但外星人是有可能存在的……
输出 2	答案 ID： 26013240 内容：首先反对 @yolfilm 的部分答案，抗生素的实效并不会引起人类文明的毁灭……
输出 3	答案 ID： 88587753 内容： 我觉得人类可能会毁灭于伊斯兰极端……
…	…
说明	从第 45 个结果开始，搜索到相关问题的答案，如： 如何评价《帝国的毁灭》， 世界四大古代文明是不是只有中华文明没有毁灭

2. 测试搜索问题在数据库中不存在的情况：

测试用例 3，见表 6-4：

表 6-4 测试用例 3

输入	人生的意义
输出 1	答案 ID： 123031464 内容： 优雅的活着，让每一分钟有意义。
输出 2	答案 ID： 24305527 内容： 你现在做的每一件小事在未来都会影响到你的生活、无论这件事在现在看意义多大
输出 3	答案 ID： 24334442 内容： 优去年夏天某次散步思考人生意义，顿悟到人生根本没有意义。
…	…
说明	搜索结果都来自相关问题，如： 人生中你经历过哪些顿悟？等

3. 测试数据库中不存在的关键词的情况：

测试用例 4，见表 6-5：

表 6-5 测试用例 4

输入	Minecraft

输出	空，直接跳转到 NoFound 页面

4. 测试输入空

测试用例 6，见表 6-6：

表 6-6 测试用例 5

输入	直接搜索，不输入任何内容
输出	空，直接跳转到 NoFound 页面

6.3 测试结果分析

性能方面：

1. 搜索数据库中存在的问题中，准确率(NDCG)平均 0.75，表现还算可以。

2. 搜索数据库中不存在的问题时，准确率(NDCG)平均 0.6，表现欠佳。

3. 由于爬虫爬取是问答数据并不完整，所以存在一些问题搜索结果准确率很低的情况。

UI 方面：

未采用分页方式显示，数据返回速度慢，且过于冗长。

第七章 全文总结和展望

7.1 总结

纵观全文，文本主要研究工作内容如下：

1. 分析知乎网站组织结构，包括话题详细页面结构，问题详细页面结构，答案详细数据获取方式，话题组成的无循环有向图，话题页面中热门问题和待回答问题的页面结构等。
2. 使用 Scrapy 框架编写爬虫，使用 redis 数据库实现了爬虫的断点爬取，全自动切换代理 IP，切换 cookie 等反反爬虫手段，并爬取知乎网问题答案相关数据，使用 MongoDB 做数据持久化。
3. 深入了解搜索引擎基础原理，包括文本特征值提取，关键词提取，文档存储方法等。
4. 深入了解搜索引擎中索引的原理和使用，包括文档正向索引，文档逆向索引，并使用爬取数据建立需要的搜索引擎索引。
5. 进行数据的预处理，包括提取数据特征值，提取文本特征值，个别特征值进行 MinMaxScaler 缩放处理。
6. 深入了解 LTR 机器学习算法，使用 scikit-learn 编写 LambdaMART 模型并使用爬取数据训练模型。
7. 利用训练好的模型在 Flask 框架和 bootstrap 下实现一个 WEB 端答案推荐系统。

7.2 后续工作展望

目前实现的系统还存在一些问题，还不够完善，后续还需要一些工作来完善相关优化相关功能，后续的完善优化工作包括：

1. 优化模型，提高排序准确率。
2. 优化特征值提取，有利于提高模型的准确率。
3. 优化数据库读写方式，加快数据库的读取速度。

致谢

在此要感谢我的指导老师俸志刚老师，之前在大二的时候就有缘与老师结识，和老师度过了一个学期的美好的教学时光。这次也有幸选到了俸老师的毕业设计题目，从选题到系统设计再到具体代码的实现，俸老师每周都会定时和我通过 QQ 的形式相互交流学习。虽然与老师只是萍水相逢，但经过这一年的交流，已经感觉到了俸老师是一名教学严谨，工作踏实，知识面广又十分幽默和蔼的老师。本文的完成，也得益于俸老师的大力指导和帮助。

参考文献

[1]. 机器学习干货站.机器学习简史[OL]. http://www.52ml.net/11881.html: smallroof, 2014.4.1

[2]. Christopher J. C. Burges, Krysta M. Svore, Paul N. Bennett. Learning to Rank Using an Ensemble of Lambda-Gradient Models[J].JMLR,2011,14:25-35

[3]. 真实的归宿.机器学习排序[OL]: https://blog.csdn.net/hguisu/article/details/7989489:2012.9.18

[4]. Hoohack: 搜索引擎索引的数据结构和算法[OL]. http://blog.jobbole.com/101792/:2016.5.31

[5]. scikit-learn[CP]. http://scikit-learn.org/

[6]. fxsjy.jieba[CP]. https://github.com/fxsjy/jieba

[7]. Bootstrap.Bootstrap[CP]. https://www.bootcss.com

[8]. Pallets.Flask[CP]. http://flask.pocoo.org/

[9]. 张俊林.这就是搜索引擎核心技术详解[M].电子工业出版社,2011.99-123

[10].McCorduck, Pamela.Machines Who Think 2nd[M].Natick, MA: A. K. Peters, Ltd., 2004.123-125

[11].McCorduck, Pamela.Machines Who Think 2nd[M].Natick, MA: A. K. Peters, Ltd., 2004.266-276,296-300,314,421

[12].7sDream. zhihu-oauth [CP]. https://github.com/7sDream/zhihu-oauth

JMLR: Workshop and Conference Proceedings 14 (2011) 25–35 Yahoo! Learning to Rank Challenge

Learning to Rank Using an Ensemble of Lambda-Gradient Models

Christopher J. C. Burges CBURGES@MICROSOFT.COM
Krysta M. Svore KSVORE@MICROSOFT.COM
Paul N. Bennett PAUBEN@MICROSOFT.COM
Andrzej Pastusiak ANDRZEJP@MICROSOFT.COM
Qiang Wu QIANGWU@MICROSOFT.COM
Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA

Editor: Olivier Chapelle, Yi Chang, Tie-Yan Liu

Abstract

We describe the system that won Track 1 of the Yahoo! Learning to Rank Challenge.
Keywords: Learning to Rank, Gradient Boosted Trees, Lambda Gradients, Web Search

1. Introduction and Summary

The Yahoo! Learning to Rank Challenge, Track 1, was a public competition on a Machine Learning for Information Retrieval task: given a set of queries, and given a set of retrieved documents for each query, train a system to maximize the Expected Reciprocal Rank (Chapelle et al., 2009) on a blind test set, where the training data takes the form of a feature vector $\mathbf{x} \in \mathcal{R}^d$ with label $y \in \mathcal{Y}$, $\mathcal{Y} \equiv \{0, 1, 2, 3, 4\}$ (a more positive number denoting higher relevance) for each query/document pair (the original, textual data was not made available). The Challenge setup, background information, and results have been extensively covered elsewhere and we refer to Chapelle and Chang (2011) for details. In this paper we summarize the work that resulted in the winning system.[1] We limit the work described in this paper to the work done specifically for the Challenge; the work was done over a four week period prior to the end of the Challenge.

Our approach used a linear combination of twelve ranking models, eight of which were bagged LambdaMART boosted tree models, two of which were LambdaRank neural nets, and two of which were MART models using a logistic regression cost. LambdaRank is a method for learning arbitrary information retrieval measures; it can be applied to any algorithm that learns through gradient descent (Burges et al., 2006). LambdaRank is a listwise method, in that the cost depends on the sorted order of the documents. We briefly summarize the ideas here, where the $\mathbf{x}_i, i = 1, \ldots, m_q$, are taken to be the set of labeled feature vectors for a given query q and corresponding documents $d_i, i = 1, \ldots, m_q$. The key LambdaRank insight is to define the gradient of the cost with respect to the score that

1. Our team was named Ca3Si2O7, the chemical formula for Rankinite. We donated half of the $8000 prize money to the NIPS Foundation and half to the International Machine Learning Society (the organizers of ICML) for student scholarships.

the model assigns to a given $\mathbf{x}_i$ *after* all of the $\mathbf{x}_i, i = 1, ..., m_q$, have been sorted by their scores s_i; thus the gradients take into account the rank order of the documents, as defined by the current model. LambdaRank is an empirical algorithm, in that the form that the gradients take was chosen empirically: the λ's are those gradients, and the contribution to a given feature vector $\mathbf{x}_i$'s λ_i from a pair $\mathbf{x}_i, \mathbf{x}_j$, $y(\mathbf{x}_i) \neq y(\mathbf{x}_j)$, is just the gradient of the logistic regression loss (viewed as a function of $s_i - s_j$) multiplied by the change in Z caused by swapping the rank positions of the two documents while keeping all other documents fixed, where Z is the information retrieval measure being learned (Burges et al., 2006; Donmez et al., 2009). λ_i is then the sum of contributions for all such pairs. Remarkably, it has been shown that a LambdaRank model trained on Z, for Z equal to Normalized Cumulative Discounted Gain (NDCG) (Jarvelin and Kekalainen, 2000), Mean Reciprocal Rank, or Mean Average Precision (three commonly used IR measures), given sufficient training data, consistently finds a local optimum of that IR measure (in the space of the measure viewed as a function of the model parameters) (Donmez et al., 2009).

While LambdaRank was originally instantiated using neural nets, it was found that a boosted tree multiclass classifier ("McRank") gave improved performance (Li et al., 2007); combining these ideas led to LambdaMART, which instantiates the LambdaRank idea using gradient boosted decision trees (Friedman, 2001; Wu et al., 2009). This work showed that McRank's improved performance over LambdaRank (instantiated in a neural net) is due to the difference in the expressiveness of the underlying models (boosted decision trees versus neural nets) rather than being due to an inherent limitation of the lambda-gradient idea.[2] For a self-contained description of these algorithms we refer the reader to Burges (2010).

Regarding our ensemble approach, four of the LambdaMART rankers (and one of the nets) were trained by optimizing for the Expected Reciprocal Rank (ERR) measure (Chapelle et al., 2009), and four (and the other net) were trained by optimizing for NDCG. For the LambdaMART models, we also generated extended training sets by randomly deleting feature vectors for each query in the training data, and concatenating the resulting data into a new training set; four of the eight LambdaMART models were trained using such data (see below for details). We performed parameter sweeps to find the best parameters for the boosted trees (such as the number of leaves and the learning rate): the parameter sweeps resulted in variations in accuracy of up to 0.08 in the absolute value of NDCG. Once the best parameters were found, we performed 10-fold bagging without replacement for each LambdaMART model, thus using all available training data for that model. Note that this bagging step was done after all parameters had been fixed. The logistic regression models optimized for the graded relevance probabilities that ERR assigns (see below), and their outputs were also used as features in two of the LambdaMART models.

We explored various approaches to linearly combine the 12 resulting rankers, using the provided training set. For the combination method, we investigated (1) linear LambdaRank, (2) a method to optimally combine any pair of rankers given any IR metric, and (3) simple averaging. We found that (3) — simply adding the normalized model scores — worked as well as the other approaches, and somewhat surprisingly, that including the less accurate models (which were the logistic regression and neural net models) also helped.

2. In fact we also trained McRank models to include in our ensemble for the Challenge, but we found the performance to be sufficiently low that they were not included; this was likely a bug, but time did not permit pursuit.

外文资料译文

Lambda 梯度模型下的机器学习排序

Christopher J. C. Burges cburges@microsoft.com
Krysta M. Svore ksvore@microsoft.com
Paul N. Bennett pauben@microsoft.com
Andrzej Pastusiakandrzejp@microsoft.com
Qiang Wu qiangwu@microsoft.com

绪论

我们用本文描述的系统赢得了雅虎机器学习排序挑战第一场比赛的胜利.

关键词:机器学习排序， 梯度提升树， Lambda 渐进，网页搜索

1. 介绍和总结

雅虎学习排名挑战赛的第一场，是一场针对信息检索的机器学习公开竞赛：给定一组查询,并为每个查询提供一组检索文档,训练系统以最大化期望的 ERR ,在测试集中，训练数据中对于每个查询/文档对(原始文本数据不可用)采用特征向量 x∈Rd 的形式，标签采用 y∈Y，Y≡{0,1,2,3,4})数值越大表示相关性越高)。 挑战赛的计划，背景,，比赛结果等已在其他地方广泛报道,详细信息我们参考了 Chapelle 和 Chang(2011)的报道。在本文中，我们总结了为了获胜而做的工作。我们将本文中描述的工作局限于专门针对比赛所做的工作;比赛结束前，我们的工作已经完成了四周。

我们使用了 12 个排序模型的线性组合，其中 8 个是 LambdaMART boosted tree 模型，其中两个是 LambdaRank 神经网络，其中两个是使用逻辑回归函数作为代价函数的 MART 模型。 LambdaRank 是一种可以学习任意信息检索方法的机器学习方法;它可以应用到使用梯度下降学习的任何算法。 LambdaRank 是一种文档列表方法，因为代价值取决于文档的排序顺序。我们简单总结下算法，其中$x_i i = 1, \dots, m_q$是给定查询 q 的对应文档$d_i, i = 1, \dots, m_q$的特征值向量集合。LambdaRank 的关键是定义代价函数的梯度能符合模型根据分数s_i将给定的x_i排序到所有$x_i, i = 1, \dots, m_q$后面所给出的结果;正如前面定义，梯度考虑了文档的排名顺序。LambdaRank 是一种经验算法,因为梯度的形式是根据经验所得：代价函数梯度为 λ，代价函数的梯度取决于给定的特征值向量x_i对应的λ_i，而λ_i取值取决

于文档对向量$x_i, x_j, y(x_i) \neq y(x_j)$, λ为代价函数梯度乘以因为只交换排序结果中两个文档所导致的Z的变化量，其中Z是信息检索度量。λ_i是所有这些文档排序对的贡献总和。需要注意的是，已证明，在Z上训练出来的LambdaRank模型，其中的Z等于归一化累积折现增益（NDCG），平均倒数秩或平均精度(三个常用IR度量)有效的训练数据，都发现了IR测量的局部最优解(在测量空间中被看作是模型参数的函数)。

虽然 LambdaRank 最初是使用神经网络实现的，但我们发现使用增强树多分类分类器(McRank)可以提高性能。结合以上想法就产生了 LambdaMART 算法，算法使用 GBDT 实现了 LambdaRank 的思想。工作表明，相对于 LambdaRank(使用神经网络实现)使用 McRank 性能有所提高，这是由于基础模型(BDT 与神经网络)之间的差异导致，而不是因为 lambda 梯度设计思想的限制。算法详细描述见 Burges(2010)。

我们的集成方法，使用四个 LambdaMART 排序模型(组成其中一个网络)通过对 ERR 度量进行优化来训练，并使用四个 LambdaMART(组成另一个网络)针对 NDCG 进行优化。对于 LambdaMART 模型，我们还通过随机删除训练数据中每个查询的特征向量，并将结果数据连接到一个新的训练集中生成扩展训练集;八个 LambdaMART 模型中的四个使用这些数据进行了训练。我们进行了参数调优，用以找出最佳参数（如叶子节点数量和学习率）：经过参数调优 NDCG 绝对值的精度变化高达 0.08。