

**课 程 报 告**

**题目： 基于用户大数据的搜索链接推荐系统**

**院（系）： 网络空间安全学院**

**课程名称： 信息检索（公选课）**

**学 号：**

**姓 名：**

**任课老师：**

**报告日期： 2022年 月 日**

**目录**

1**绪论** 1

1.1课题背景 1

1.2课题目的与意义 1

1.3国内外研究现状 1

2**课题内容** 3

2.1课题需求 3

2.2数据的获取 3

2.3数据的分析处理与储存 4

2.3.1 query的提取 4

2.3.2 query与link相关度计算 4

2.4处理后数据的利用 6

3**遇到的困难与解决方案** 8

3.1数据收集部分 8

3.1数据处理部分 8

4**总结与展望** 9

**参考文献** 10

**附录** 11

**1绪论**

**1.1课题背景**

随着物联网、社交网络、云计算等技术不断融入我们的生活以及现有的计算能力、储存空间、网络带宽的高速发展，人类积累的数据在互联网、通信、金融、商业、医疗等诸多领域不断增长和积累。

互联网搜索引擎支持数以十亿次web搜索，每天处理数万TB字节数据。全世界通信的主干网上一天就有数万TB字节数据在传输。数据的量级不断升级、应用的不断深入和大数据不可忽略的价值让我们不得不探索如何才能让我们更好的受益于这些数据。

信息冗余、信息真假、信息安全、信息处理、信息统一等问题也随着大数据给人们带来价值的同时也造成了一系列的问题。人们不仅希望能够从大数据中提取出有价值的信息，更希望发现能够有效支持生产生活中需要决策的更深层次的规律。

互联网中有着海量的数据，如何从数以亿计的页面中获取符合需求的信息成为了一个重要的问题，而搜索引擎的出现，很好地解决了这些问题。所谓搜索引擎，就是根据用户需求与一定算法，运用特定策略从互联网检索出指定信息反馈给用户的一门检索技术。搜索引擎依托于多种技术，如网络爬虫技术、检索排序技术、网页处理技术、大数据处理技术、自然语言处理技术等，为信息检索用户提供快速、高相关性的信息服务。搜索引擎技术的核心模块一般包括爬虫、索引、检索和排序等，同时可添加其他一系列辅助模块，以为用户创造更好的网络使用环境。

使用普通的搜索引擎，当给定一个搜索词汇后，服务器返回一系列的链接与其简介，用户需要进行二次筛选获得所需要的内容。在一系列的链接中，符合用户需求的链接往往所占数量较少，因此用户通常需要浏览多个搜索结果后才能获得所需要的内容。因此搜索词与网页链接间的相关性的强弱，是决定搜索引擎使用体验的一个重要因素。

**1.2课题目标与意义**

常规搜索引擎根据用户点击量对各页面进行排名计算，但由于给出的一系列网站页面中，有很多的页面可能包含大量广告、付费等无价值信息，当用户点击这种页面后，往往浏览时间较短，并未获得有价值的信息，但依旧为网站增加了点击量，导致网站排名进一步增加。

本课题的目标，是根据已获得的数据，通过分析计算每个页面的浏览时长，计算出搜索词与该页面间的相关性，最终通过一个简单的搜索引擎，实现为指定搜索词寻找相关性最高的网站页面，省去用户二次筛选的步骤。

**1.3国内外研究现状**

在国际互联网市场上，搜索引擎主要以Google为主，市场占有率在90%以上，而在国内市场，搜索引擎主要以百度、Bing为主。虽然互联网中有着大量的搜索引擎，但无一例外的是，这些搜索引擎都需要用户进行二次筛选搜索结果，其原始数据的获取方式通常为使用爬虫爬取各个网页，分析页面内容，从页面中提取出关键词，当用户搜索该关键词时，返回出包含该关键词的页面，并且根据用户的点击量优化页面排名。

**2 课题内容**

**2.1 课题需求**

该项目需要大量的用户浏览器记录，其包括每个页面的url地址以及其访问的时间，用户每次访问浏览器产生的所有数据应保存在同一个用户id下，并且将获取到的这些数据发送到服务端保存在数据库中。

获取到数据后，对同一个用户id下的url地址进行分析，提取其中的搜索词query，并根据页面的浏览时长计算出url与该query之间的相关性并将url与其相关性储存在该query下。

计算并储存url与query之间的相关性后，当向系统查询一个新的query1时，将该query1与已处理过的query进行比较，计算其字符串相似度，寻找相似度最高的query2，并查询出该query2所储存的url，查询出相关性最高的url，最终返回该url，即实现了为指定搜索词推荐最佳网页的效果。

**2.2 数据的获取**

根据课题需求，首先需要收集用户的浏览器记录，使用第三方软件搜集浏览器记录容易被系统识别为恶意软件，并且对操作系统、使用的浏览器版本等有较高的要求，因此排除制作第三方软件的可能性。除了第三方软件、能直接获取到浏览器记录的方式便是通过浏览器本身，得益于主流浏览器，如chrome、edge等扩展插件的使用，可以很方便的开发出插件来收集浏览器的记录。本项目的插件面向chrome浏览器开发，同时支持edge浏览器使用。

浏览器插件的开发需要使用到JavaScript语言，将指定代码通过插件注入到浏览器页面并随页面加载开始执行：

var url = window.location.href;// 获取页面url

var time = new Date().getTime();    // 获取页面时间

将这两行代码注入到前端页面中，便能获取到新建页面的url地址以及页面创建时的时间戳。

从浏览器捕获到所需的信息后，还需要将数据上传到服务器进行储存。js前端向向服务器发送数据，可以选择使用ODBC，直接通过js连接MySQL数据库，将数据插入数据库中，但这种做法会带来明显的安全问题，用户可以提取出扩展插件的代码，获取数据库的账户和密码，可能造成数据泄露等严重后果。

除了上述做法，最安全的是创建一个服务器程序，作为连接用户端插件和MySQL数据库。js向浏览器发送数据，可以使用websocket与服务器建立通话，也可以向服务器发送http的GET请求，将数据携带在请求体中。因为仅需要单向通话，不需要进行大量的数据交换，因此本项目中选用发送GET请求的方式向服务器发送数据。所使用到的GET请求如下：

var req = `http://$host$:$port$/?id=${id} &time=${time} &url=${url}`;

其中id是浏览器打开时随机分配的一个长度为10的字符串，用来将用户的单次使用行为进行标记，time为前面所获取到的页面打开时间，url即为页面的url地址。向服务器发起该请求，便实现了数据的发送。（完整代码见附录）

服务器端，使用python创建一个服务器用于接受用户浏览器插件发送的请求，对GET请求进行切片操作，便可得到url，id，time三个重要数据，将这些数据储存再MySQL数据库中，便完成了数据的获取。

**2.3 数据的分析处理与储存**

**2.3.1 query的提取**

得到原始数据后，对获取的url解码，便能获得形如：

"https://cn.bing.com/search?q=搜索示例&cvid=b6ef47128d6e458db049049"

的url链接，在全部的url链接中，有一部分链接指向的是用户使用搜索引擎时所创建的页面，分析链接可以发现，该链接包含着GET请求，在请求体中附带着搜索词（query），浏览器版本等多种信息，对该类型的url进行匹配和提取，便能获取到链接中的搜索词。

template = "https://cn.bing.com/search?q="

if (url[0 : len(template)] == template):

    lindex = len(template)

    rindex = url.find("&")

    query = url[lindex: rindex]

    return (query, 'bing')

使用模板template对不同的搜索引擎的搜索链接进行匹配，若匹配成功，则能够表示该链接为搜索链接，进一步通过字符串识别，获取query在url中的位置，最后切片操作得到完整的query。

将提取出的query存储在MySQL数据库中，按照存储的数据为query分配唯一的id，并将该id作为储存相关url的数据表的表名，当需要更新某一query下的url记录时，只需要先查询数据表，获取query的id，接着查询id对于的数据表即可。

**2.3.2 query与url相关度计算**

用户每一次打开浏览器，随机分配一个uid，随后所有的浏览器链接（url）都与该uid关联存储，与其同时存储的，还有每个连接（url）创建的时间戳（time），计算第i个url与第i+1个url间的时间差（time\_dif），即可得到第i个页面的浏览时间:

若该url为数据表中的最后一个url，无法使用时间差的方法计算该页面的浏览时间，根据用户行为分析：用户浏览此页面后关闭了浏览器，代表着用户已从该页面及以前的页面获取到了自己所需要的信息，且在此页面获取到与查询query相关信息的可能性最大。因此，默认为最后一个页面设置为较长的浏览时间，使其计算相关度时有较大的权重。

不难分析，一个页面刚被打开就被切换为其他页面，代表该页面内包含的信息不符合用户所需要的内容，即该url与用户输入的query相关性较低，浏览时间长则表示该页面的内容为用户所需内容，即该url与用户输入的query相关性较高。相关性relevance与浏览页面的时间T正相关。

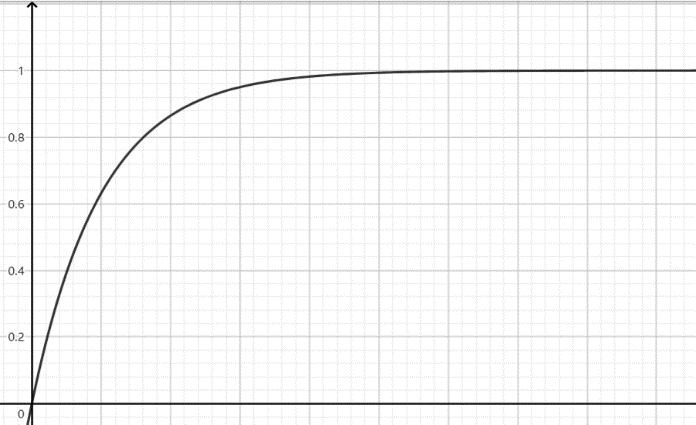
 又有相关性R小于等于1大于0，且T与R并非简单的线性关系，因此使用一下函数曲线拟合二者的相关性：

图2-1 链接相关性与浏览时间的相关性曲线

公式中k为常数，用于调整R随时间t变化的速率，当W=1时，假设t=20s（20000ms）时相关性R取值为80%（0.8），计算得到：

公式中（0≤≤1）由该url以除浏览时间以外的性质决定，主要由url在用户所浏览的所有页面url\_list中的位置、页面的性质等多种因素决定。

对于用户使用浏览器，每次打开浏览器后往往不是简单的搜索、查找选择包含所需内容的页面、退出浏览器这样的单一流程，因此，同一个uid下的各个url往往并不属于一个query，所以需要考虑该url与query的相对位置，设url与query的距离（ = …-2, -1, 0, 1, 2…）与其影响因子之间的关系由下表确定：

表2-1 距离与影响因子的关系

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 距离 | … | -2 | -1 | 0 | 1 | 2 | … |
| 影响因子 | 0 | 0.1 | 0.4 | 1 | 0.6 | 0.3 | 0 |

后续页面的相关性对先前的页面也存在影响，影响因子由后第i个页面的相关性决定，当后续页面的相关性较高时，靠前的页面的相关性应适当降低，而当后续页面的相关性很低时，则后续页面对先前页面的影响则很低，其影响程度由下表确定（表中的为计算影响因子前的相关性，即仅由页面性质、页面浏览时间、页面与query的距离决定，与后续页面的相关性无关）：

表2-2 后续页面相关性的影响程度表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0~0.2 | 0.2~0.4 | 0.4~0.6 | 0.6~0.8 | 0.8~1 |
|  | 1 | 0.95 | 0.9 | 0.85 | 0.8 |

又由于相关性R是根据浏览时间确定的，有一种情况需要特殊考虑：有播放视频的页面存在。若出现一个长达一分钟的视频，其相关性将高达99%，但视频对于搜索结果来说往往相关性较差，为了保留大多数数据的可靠性，还需要对视频结果进行单独处理。对于视频网站，降低其时间差（time\_dif）的权重，将其时间差的权重置为普通网站的，并取消其对前序页面的影响因子。

在各种特殊情况中，还有一种情况较为特殊：用户输入一个query，需求为查询某网站的官网链接，进入该链接后进行多次跳转。在这种情况下，官网链接的相关性应当很高，但使用上述算法，则会导致按照浏览时间排序，因为官网首页的浏览时间很短，会导致真正的官网首页计算所得相关性往往很低。为解决该情况，需要对同一个query下的多个url检测其官网链接，若均属于同一网站下，且有官网首页的存在，则提升官网url的相关性，降低甚至取消其他页面的相关性。

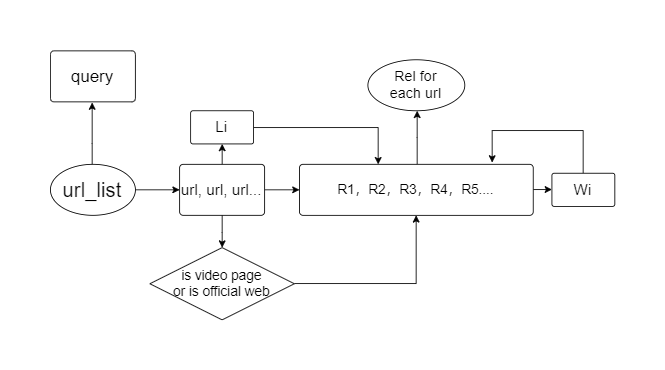
对上述各种情况进行归纳，得到如下图的相关性计算流程：

图2-2 url与query相关度计算流程图

【算法部分见附录】

按照上述流程，即可计算出url\_list中每一个url与每一个query之间的相关性，按照相关性由大到小排序，记录相关性最大的5个url（不足5个时以实际个数为准），对于每一个url，储存时还需要考虑原数据库中是否已保存该url，若已存在，使用公式：

累计计算多次记录。

最终，从数据库中已有的url和新增的url中找出相关性最大的5个url，存入该query所对应的数据库中。

**2.4 处理后数据的利用**

完成上述对原始数据的获取后，便可得到一系列的query和与之对应的相关性数据表：



图2-3 url与query相关度数据表

在该数据表中按照相关性大小储存着url数据，当需要查询query下的url时，首先需要获取到query在数据库中的id，再查询id下的表单即可。

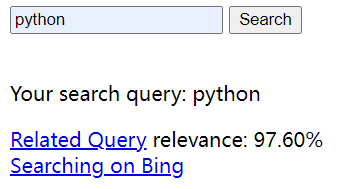
 为用户创建一个简单的搜索引擎：

图2-4 搜索引擎的实现

当用户输入一个query后，在数据库中找到对应的query，查询到相似度最大的url后返回该url，即实现了为给的query查找相关性最高的网页的目的。

但在实际使用中，用户的搜索词并不一定已经被服务器处理过，且这种情况的概率远大于能够直接找到相应query的概率，所以，对于不存在的query，也需要经过处理获得相关性较高的页面。

当用户所查的query0不在数据库中时，将该query0与服务器中保存的每一个query进行相似度计算：

query1的长度为len1，query2的长度为len2

计算出query1与query2的最长公共子字符串长度lcs\_len

则query1与query2之间的相似度：

计算出query0与每一个query后，得到相似度最大的三个query，依次对三个query进行处理：

与query0的相似度为

查询下相关性最高的，其相关性为

则与query0的相关性为

完成全部计算后，即可得到相关性最高为relevance的url，为用户返回该url即可。

**3 遇到的困难和解决方案**

**3.1 数据收集部分**

设计数据收集部分时，一开始将向服务器发送数据的部分与采集部分都放置在了content\_script.js中，随页面的加载注入到页面中执行，在本地测试时可以正常使用，插件能够正常连接服务器端，服务器也可以正常编码插件发送的二进制数据。但当服务端部署在远程服务器上时，却出现服务器端发送的二进制数据不能够正常编码的情况，因此导致插件与服务器间的http连接不能正常建立。

为排除该问题，先后更换了三次服务器，仍然不能解决。于是选择放弃http发送GET请求，选用websocket的方式与服务器进行通信，当在本地测试时，使用ws协议可以正常通信，但部署在远程服务器上时，ws协议无法建立，始终报错被页面阻止，使用wss协议后，浏览器上不再出现报错，但与使用http请求出现了相同的问题，二进制数据无法编码。

最后通过大量测试得出结果，在http协议的页面下，使用http请求发送数据与使用websocket发送数据一切正常，但在https协议的页面下，http请求无法发起，ws协议的websocket连接无法建立，wss协议连接出现二进制数据无法编码。查询http与https协议相关内容后才了解到，https为加密协议，在https页面下无法发起http，且wss协议也为加密协议。

查阅chrome扩展开发文档后得知，当浏览器打开时，会为插件打开一个background页面，可以在background页面上运行js代码，且不受前台页面的影响。于是选择将发送数据部分与采集数据部分进行分离，将发送数据部分写入到background页面上，创建listener函数监听信息，当注入到页面中的代码采集到数据后，将数据打包在message中广播，listener接收到message后，将数据通过http请求发送到服务器。

当完成插件制作时，为该项目分配的时间已经过去了三分之一，影响到了后续的开发。

**3.2 数据处理部分**

项目最开始的设想为收集一万条左右的原始数据，保证最终获得的query数量以及相关性的准确度。但最终插件发布后安装量极低，到整个项目完成时获取到的原始数据仅有七百余条，提取出的query数量有限，且query大部分为计算机编程相关，当搜索其他领域词汇时，常出现无法找到相关页面的情况。

**4 总结与展望**

通过对《信息检索》这门课程的学习，对数据处理、机器学习等方面都有了很多的了解。特别是其中的推荐算法和机器学习部分，为我带来了不少的启发，感谢魏老师能够带来如此精彩并且有价值的课程。在该课程的影响下，我对人工智能和深度学习有了很大的兴趣，也开始了该领域的学习。感谢魏老师的幸苦付出。

对于项目的选题，最开始的设想是通过深度学习训练一个模型来处理数据，但因为在这方面的学习刚刚起步，在即将结束课程时依旧不能很好的运用深度学习的知识，最终只能改变选题，转而使用常规的算法进行处理数据。

在后续的学习中，我会尝试用最开始的思路实现该选题，继续加强人工智能领域的学习，可能以后会结合机器学期和网安专业知识做别的项目。

最后，还要感谢为我提供项目所需数据的几位好友，在他们的帮助下，我得以完成了数据的分析等操作。

**参考文献**

[1]龙佳. 论搜索引擎的特点与发展态势[J]. 电脑知识与技术,2019,15(1):200-201.

[2]续扬. 大数据时代的新媒体发展[J]. 中国传媒科技, 2013(4):2.

[3] [【干货】Chrome插件(扩展)开发全攻略 - 我是小茗同学 - 博客园 (cnblogs.com)](https://www.cnblogs.com/liuxianan/p/chrome-plugin-develop.html)

[4] [最长公共子序列（LCS） - ranjiewen - 博客园 (cnblogs.com)](https://www.cnblogs.com/ranjiewen/p/5559490.html)

[5] [HTTP 与 HTTPS 的区别 | 菜鸟教程 (runoob.com)](https://www.runoob.com/w3cnote/http-vs-https.html)

**附录**

项目完整代码请访问github：[noxke/search-recommend: Analyze the relevance between query and url based on browser records (github.com)](https://github.com/noxke/search-recommend)

部分重要代码如下：

【js扩展通过GET请求向服务器发送数据】

const id = Math.random().toString(36).slice(-10);    // 浏览器打开时创建随机id

chrome.extension.onMessage.addListener(

    function(message) {

        if (opt) {

            url = message[0];

            time = message[1];

            const Http = new XMLHttpRequest();

            var req = `http://$host$:$port$/?id=${id} &time=${time} &url=${url}`;

            Http.open("GET", req);

            Http.send();

        }

    }

);

【相关度计算算法】

def analysis\_data(uid, url\_list):

    '''对于每个uid的所有访问数据进行处理'''

    query\_dict= {}  # {index : query}

    link\_dict = {}  # {index : (time\_dif, url)}

    for index, data in enumerate(url\_list):

        url = data[0]

        query = extract\_query(url)

        if (query != None):

            query\_dict[index] = query[0]

            update\_query(query[0])

        else:

            if (index + 1 >= len(url\_list)):    # 数据表中最后一个数据，默认时间差为60秒

                time\_dif = 60 \* 1000

            else:

                time\_dif = url\_list[index + 1][1] - url\_list[index][1]

            if (is\_video\_web(url)): # 视频网站降低时间差权重

                time\_dif = time\_dif / 1000

            link\_dict[index] = time\_dif

    # 更新每个url和query之间的relevance

    query\_index\_ls = [i for i in query\_dict]

    for k, index in enumerate(query\_index\_ls):

        query = query\_dict[index]

        rel\_dict = {}   # {index: relevance}

        rel\_list = []   # [(index, relevance)]

        lr = {-3: 0, -2: 0.1, -1: 0.4, 0: 1, 1: 0.6, 2: 0.3, 3: 0}

        for i, time\_dif in link\_dict.items():

            url = url\_list[i]

            rel = 0

            if (i < index): # url在query之前

                if (k == 0):

                    l = -1

                elif (i > query\_index\_ls[k - 1]):

                    l = -1

                else:

                    if (k - 1 == 0):

                        l = -2

                    elif (i > query\_index\_ls[k - 2]):

                        l = -2

                    else:

                        l = -3

            else:   # url位于query之后

                if (k == len(query\_index\_ls) - 1):

                    l = 0

                elif (k < query\_index\_ls[k + 1]):

                    l = 0

                else:

                    if (k + 1 == len(query\_index\_ls) - 1):

                        l = 1

                    elif (i < query\_index\_ls[k + 2]):

                        l = 1

                    else:

                        if (k + 2 == len(query\_index\_ls) - 1):

                            l = 2

                        elif (i < query\_index\_ls[k + 3]):

                            l = 2

                        else:

                            l = 3

            rel = lr[l] \* (1 - math.pow(math.e, -0.00008 \* time\_dif))

            rel\_dict[i] = rel

        if (k == len(query\_index\_ls) - 1):

                rindex = len(url\_list) - 1

        else:

            rindex = query\_index\_ls[k + 1] - 1

        for i in range(index + 1, rindex + 1):

            rel = rel\_dict[i]

            lindex = i + 1

            url = url\_list[i][0]

            if (is\_of\_web(url)):

                of\_web = official\_web(url)

            else:

                of\_web = ""

            for j in range(lindex, rindex + 1):

                rel2 = rel\_dict[j]

                url2 = url\_list[j][0]

                if (official\_web(url2) == of\_web):

                    rel = math.sqrt(rel + rel2 - rel \* rel2)

                else:

                    if (rel2 <= 0.2):

                        wi = 1

                    elif (rel2 <= 0.4):

                        wi = 0.95

                    elif (rel2 <= 0.6):

                        wi = 0.9

                    elif (rel2 <= 0.8):

                        wi = 0.85

                    else:

                        wi = 0.8

                    rel = rel\* wi

            rel\_list.append((i, rel))

        # 将url按照相关性排序插入到对应的query表单中，只插入前五条

        rel\_list.sort(key=lambda x : x[1], reverse=True)

        if (len(rel\_list) > 5):

            rel\_list = rel\_list[0 : 5]

        url\_ls = []

        for i, r in rel\_list:

            url = url\_list[i][0]

            url\_ls.append((url, r))

        update\_query\_rel(query, url\_ls)

    # 将处理过的数据的is\_processed全部标记为1

    for data in url\_list:

        update\_url(uid, data[1])

    for url, relevance in url\_list:

        tmp = url\_ls

        for u, rel in tmp:

            if url == u:

                relevance = math.sqrt(relevance + rel - relevance \* rel)

                url\_ls.remove((u, rel))

        url\_ls.append((url, relevance))

    url\_ls.sort(key=lambda x : x[1], reverse=True)

    if (len(url\_ls) > 5):

        url\_ls = url\_ls[0:5]

【计算两个query间的相似度】

def query\_similarity(query1 : str = "", query2: str = "") -> float:

    '''计算两个query的相似度'''

    query1 = query1.lower()

    query2 = query2.lower()

    len1 = len(query1)

    len2 = len(query2)

    dp = [[0 for j in range(len1 + 1)] for i in range(len2 + 1)]

    for i in range(1, len2 + 1):

        for j in range(1, len1 + 1):

            if (query1[j - 1] == query2[i - 1]):

                dp[i][j] = dp[i - 1][j - 1] + 1

            else:

                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    lcs\_len = dp[-1][-1]

    if (len1 \* len2 == 0):

        if (len1 == 0 and len2 == 0):

            return 1.0

        else:

            return 0

    return (lcs\_len \* lcs\_len) / (len1 \* len2)