Summer 2019

Ads searching service

ming wu

Ads Searching Service

Ming Wu
Computer Engineering
Iowa State University

1. Introduction:

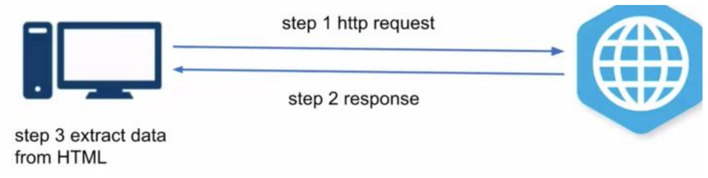
In this project, I modeled an advertisement searching service. Recently, big data and machine learning models have been widely used in online shopping or services. To better serve people, knowing one's behavior has become a good challenge. In my article, Ads Searching Service, I applied several machine learning algorithms including, the newly released word2vec model in 2013 by Google, which gives a new idea about comparing word similarities with vectors in space.

2. Web crawling:

All data I used in this project comes from web crawling. Web crawling is about making requests and extracting data from the response, which is also the first and foremost stage in any web Information Retrieval system.

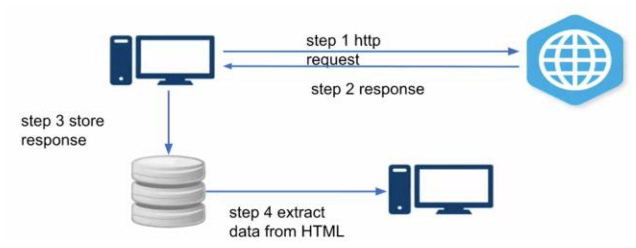
2.1 Two design choices:

2.1.1 Synchronously parsing HTML:



Parse HTML Page synchronously by three steps: Step 1: machine starts sending http request to a website. Step 2: The website receives request and response, all responses are stored in memory. Step 3: machine extracts data from HTML. which can easily be the web crawler design. For all information extracted from HTML, I will save all the data for further use.

2.1.2 Asynchronously parsing HTML page:



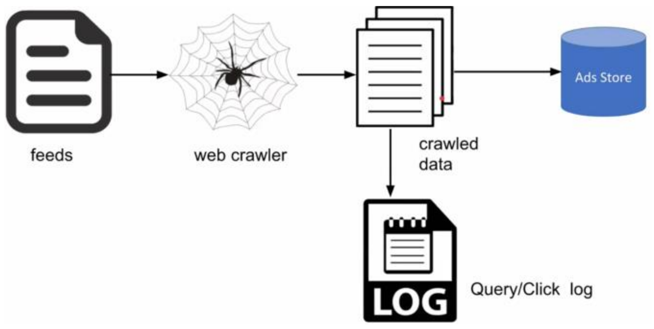
Parse HTML Page asynchronously by four steps: as seen in the synchronous way, repeat step 1 and step 2, for step 3: without doing HTML parsing directly, store responses to database. For extra step four: the data is extracted through HTML from another machine.

2.1.3 Compare and Contrast these two methods: parsing HTML synchronously would save disk space. However, if the user needs to extract more data, they would need to crawl again and single thread crawling is blocked by parsing HTML, which is also a waste of network bandwidth. In contrast, parsing HTML asynchronously would let you re-parse HTML if desired and unblock crawling from parsing HTML. However, the disadvantage would be requiring more storage space and more machines or CPUs to parse HTML. Also, in order to keep data fresh, repeatedly crawling work needs to be done regularly.

2.2 Web crawling challenge:

2.2.1 Where and what to start crawling:

For at least two layers of crawler, I need a list of website urls. However, websites are weighted differently. To better start with well-weighted websites, I used google website ranking as I always do. Another method I used was starting with feeds files, for instance, and making full use of topsites. After url crawling, I requested urls with different parameters. In the sample feeds file, there were three columns including: Query, Bid, Campaign Id. Bid price and Campaign are assigned ahead of time.



All crawled data is saved in file format, one part of which is stored as advertisements for future use, the rest is generated as query or click log file.

2.2.2 Network I/O is the bottleneck

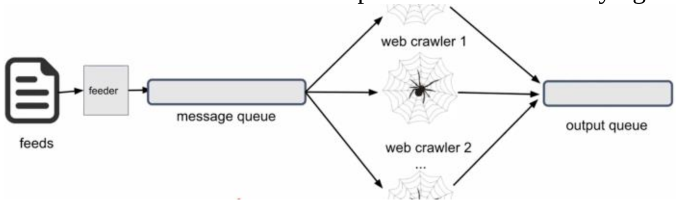
The most expensive thing of web crawling is sending HTTP requests from different websites. For example, if one site takes 2 - 3 seconds to respond, then you are looking at making 20 - 30 requests a minute, which wastes CPUs when waiting for responses from the website. To solve this problem, I used multi-threaded crawler and non-blocking IO.

2.2.3 Avoid Bot Detection

When crawling a website, it is possible that all request responses will have the same 503 error, which refers to service unavailable. Generally, 503 is a temporary state, it may represent server overload or maintenance. There are multiple ways to avoid bot detection, and one way is spoofing http headers for the user agent. Another way is proxy service by rotating IPs using a list of over 100 proxy servers.

2.2.4 The crawler needed to be resilient

The crawler may crash due to uncaught exceptions. When crawling large sets of data, it is possible for either the crawler to crash or crawl the same url. Trying to handle failed urls and avoiding the same url is essential. To solve this problem, I log crawled the url, some of which is stored in memory. Log failed url and exception for retrying.



2.2.5 Distributed Crawler Design

When crawling data sets becomes large, for instance, a billion, it cannot be finished by several machines. After doing page ranking and getting feeds files, I pushed the feeder to the message queue, which can be Kafka. All N multi-thread web crawlers are paralleled to read information from message queues and send HTTP requests to an output queue. The Resumer of the output queue stores all responses to the Hadoop class. The algorithm for multiple web crawlers remains the same. To remove duplicates in the distributed system, in other words, to remove duplicates of urls, I did MapReduce in feeds files and made sure all urls sent to message queue were unique.

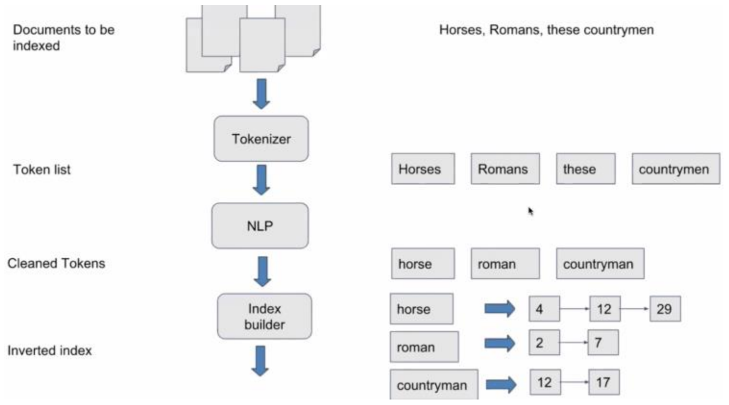
3. Information Retrieval:

Finding material of an unstructured nature that satisfies an information need from within large collections.

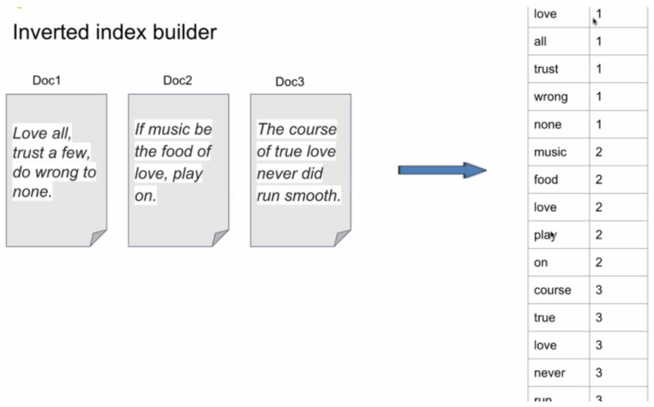
3.1 Inverted index

3.1.1 Inverted index construction

For each term t, we must store a list of all documents that contain t. Identify each doc by a docID, which is a document serial number. In this project, docID is the url.

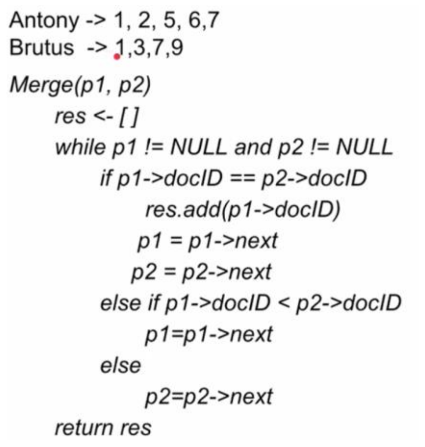


Inverted index builder requires four major steps. The first one is **tokenization**, which cuts character sequence into word tokens. I implemented tokenizer to remove all the punctuations and other irrelevant items, which output as a list of token. The second step is called **normalization**, which maps text and query terms to the same form. The third step is **stemming** in order to match different forms of a root. I applied **NLP** to clean up all tokens. For instance, I transferred all words to lowercase, converted plural to singular, removed stop words, which means omitting very common words. After these steps, each input document became normalized and well-prepared as an input of index builder.


Inverted index builder
Doc1
Doc2
Doc3
Love all, trust a few, do wrong to none.
If music be the food of love, play on.
The course of true love never did run smooth.
love 1
all 1
trust 1
wrong 1
none 1
music 2
food 2
love 2
play 2
on 2
course 3
true 3
love 3
never 3

3.1.2 Process a query with inverted index example

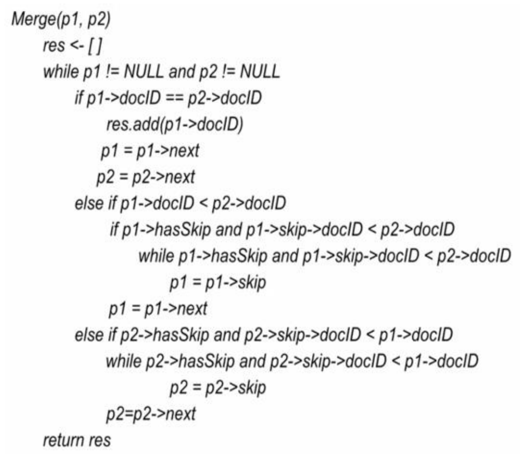
Consider processing the query: Antony and Brutus. Firstly, locate Antony in the dictionary and retrieve its postings. Similarly, locate Brutus in the dictionary and retrieve its postings. Then, merge the two postings, meaning intersect the document sets. In simplest terms, merge the two sorted lists. After that, iterate the two posting lists simultaneously with two pointers respectively.



```
Antony -> 1, 2, 5, 6,7
Brutus  -> 1,3,7,9
Merge(p1, p2)
    res <- [ ]
    while p1 != NULL and p2 != NULL
        if p1->docID == p2->docID
            res.add(p1->docID)
            p1 = p1->next
            p2 = p2->next
        else if p1->docID < p2->docID
            p1=p1->next
        else
            p2=p2->next
    return res
```

3.1.3 Posting lists with skip pointer

To identify where I placed skips pointers and the optimal length of skip span see figure below: More skips mean shorter skip spans, which we are more likely to skip. It also means lots of comparisons to skip pointers and lots of space storing skip pointers. Fewer skips mean few pointer comparisons, but then long skip spans mean that there will be fewer opportunities to skip. The most common way is to square root the length of the posting list to get the skip spam.



```
Merge(p1, p2)
    res <- [ ]
    while p1 != NULL and p2 != NULL
        if p1->docID == p2->docID
            res.add(p1->docID)
            p1 = p1->next
            p2 = p2->next
        else if p1->docID < p2->docID
            if p1->hasSkip and p1->skip->docID < p2->docID
                while p1->hasSkip and p1->skip->docID < p2->docID
                    p1 = p1->skip
            p1 = p1->next
        else if p2->hasSkip and p2->skip->docID < p1->docID
            while p2->hasSkip and p2->skip->docID < p1->docID
                p2 = p2->skip
            p2=p2->next
    return res
```

4. MapReduce

Map divides the input into ranges and creates a map task to transfer each partition. The input for Map can be any string. The output can be either keys or values. Shuffle distributes partitions to different machines by keys. Reduce collects the various results and combines them to answer the larger problem that the master node needs to solve.

4.1 MapRedue:

Label each document with number which is my input. Split each document with space. Count the number of word appearance and saved in the map. Shuffle documents with same key to same machine. In key list of values, add values together as reduce part

5. Query rewrite

The goal of query rewrite is to find queries related to the issued one, which would allow us to retrieve relevant ads that were not matches by the original query. The approach is to find K nearest neighbors of original query, semantically similar queries. Refer to intuition, if we can find the vector representation of the query, then we can calculate the similarity by cosine of the two vectors.

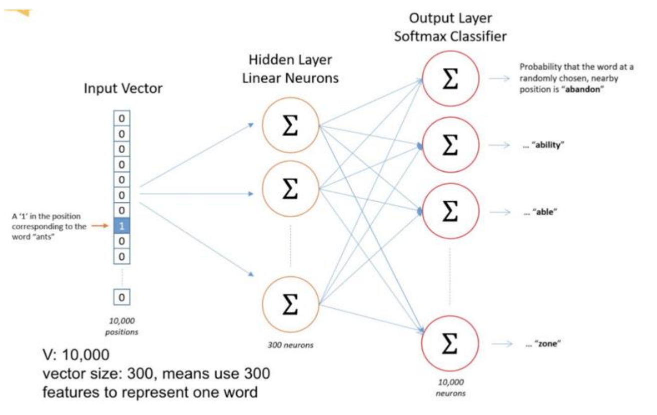
5.1 word2vec - skip gram model

The definition of word2vec is that for a given word in a sentence, what is the probability of each and every other word in our vocabulary appearing anywhere within a small window around the input word?

The window size is important, because it will affect the collection of training samples. To use the skip gram model training, text corps are needed with the vocabulary of V unique words as training data. Input word representation is one-hot vector for each word, this vector will have V components (one for every word in our vocabulary) and we will place a "1" in the position corresponding to the word, and 0s in all of the other positions. The output is a single vector containing for every word in our vocabulary, the probability that each word would appear near the input word. Last but not least, intuition is that if two words have similar contexts, then our network is motivated to learn similar word vectors for these two words.

5.2 word2vec training:

What I used for training was a hidden layer of Linear Neurons. The targeted words will be placed as a "1" in the position corresponding to the word. Suppose there's a vocabulary V with the size of n, which means the input one-hot vector will be the same size as n, and the hidden layer would have m neurons. The reason I use word2vec is that I want each word to have a representation of the vector so that I could determine the similarity by calculating the cosine of two vectors. The size of the output vector depends on the number of neurons in the hidden layer. The output layer should remain the same as the input vocabulary, which should be n.


Input Vector
Hidden Layer
Linear Neurons
Output Layer
Softmax Classifier
Probability that the word at a randomly chosen, nearby position is "abandon"
... "ability"
... "able"
... "zone"
A '1' in the position corresponding to the word "ants"
10,000 positions
300 neurons
10,000 neurons
V: 10,000
vector size: 300, means use 300 features to represent one word

Each neuron was pointed by a weighted line from input vector. The hidden layer is a weighted matrix with n rows (one for every word in our vocabulary) and m columns (one for every hidden neuron). So, the size for the matrix is n multiplied by m, which is the size of the vocabulary multiplied by number of neurons. The output word vector would become a lookup table, in which each row is the number of input words and each column is its feature. Notice that the hidden layer matrix is the lookup table we want. The goal is to learn this hidden layer weight matrix which can help calculate word vector.

Given an example of word2vec, suppose there's a vocabulary with the size of n, and the feature vector size is m. The hidden layer should be a matrix with n rows and m columns. We will get the same size matrix output after doing multiplication. Then, we will use output vector as an input to multiply the output weights and send it to the activation function Softmax. The function for Softmax is below:

$$\text{softmax} = \frac{e^x}{\sum e^x}$$

If you randomly pick a word nearby the targeted word, you will get the probability.

In order to train our model, we need a target function. There's an activation function for output layer. In this part, we cannot apply gradient descent directly, instead, we minimize the target function. Wi represents the targeted word, and the o1 to oc represents words that we are going to use to calculate the probability appearing near the targeted word. We apply the maximum likelihood to get the hierarchical Softmax to solve the computation cost issue. Apply Backpropagation and Stochastic Gradient Descent to solve the weighted values. We randomly define a default weight value to start training. As we already have the input vector, we then do forward calculation, which multiplies the input by default weight and we will then get the output for each layer. For the very last layer, once we have calculated the weight(y value) and gone backwards to do partial differential to get the real weight, that becomes the gradient. Last step, use stochastic to update the weight value until converged, which means the minimized value stays unchanged. (figure next)

Minimize target function

$$\begin{aligned} E &= -\log p(w_{O,1}, w_{O,2}, \ldots, w_{O,C} | w_I) \\ &= -\log \prod_{c=1}^{C} \frac{exp(u_{c,j_c^*})}{\sum_{j'=1}^{V} exp(u'_j)} \\ &= -\sum_{c=1}^{C} u_{j_c^*} + C \cdot \log \sum_{j'=1}^{V} exp(u_{j'}) \end{aligned}$$

5.3 Train query rewrite in two levels

To calculate query rewrite with word2vec by two levels. In term level, I replaced query term with similar terms. I extracted the term from the query or title and created new training data. After that I trained the word2vec model with phrase data. To determine if the training model was good enough, I tested phrases with similar meanings to see if they could be determined as synonyms. The machine will tell how similar the words are near the targeted ones. In order words, how similar the environment is around the targeted words to determine if those phrases are synonyms. Similarly, in phrase level, after I extracted phrase from query or title and created new training data, I linked those phrases with underscore. For training online, I will record all searching queries in a specific time limit, for instance, 5 minutes. I used query groups in seeds file to model this. Then, I repeated extracting phrase from query and replaced phrase with similar phrase trained earlier.

For phrase detection, I extracted bigram (two words) from text corpus and calculated score:

$$\begin{aligned} \text{score}(w_1, w_2) &= \frac{p(w_1, w_2)}{p(w_1) * p(w_2)} \\ &= \frac{\frac{\text{count}(\text{w}_1, \text{w}_2)}{\text{N}}}{\frac{count(w_1)}{N} * \frac{count(w_2)}{N}} \\ &= \frac{count(w_1, w_2)}{count(w_1) * count(w_2)} \end{aligned}$$

In this calculation, count(w) is number of times word w shown in text corpus and count(w1,w2) is number of times word w1, w2 shown together in text corpus. N refers to the unique words in text corpus. I selected bigram with score larger than threshold as phrase and stored raw queries(keys) and queries with phrases (values) in key value store.

Overall, to generate query rewrite, first, I found synonyms of raw queries and found a way to quantify similarity between terms. Second, train word2vec model with SparkMLlib since its fit my requirement. The input data is cleansed ads titles and queries in seed files. Third, I processed all queries offline. For each term in one query, find top 5 synonym terms and Construct synonyms for each query by different combination of 5 synonyms. Last, I stored query's synonyms in key value store.

6. Query intent extraction

The goal for query intent extraction is to generate subqueries which preserve the intent of the original query in the best way and allow us to retrieve more relevant ads. The logistic regression classifier is used to determine the goodness of each subquery. My intuition is that historically good subqueries have more clicks on relevant ads, which contain terms in subqueries (overlap subqueries). Query Intent Extraction is also defined as **supervised learning**. Given a long query, the machine generates subqueries by sliding windows with different window sizes and records the number of clicked pairs. To determine the coherence between any pair of tokens in a query based on search log, I calculated the MCI, which means the mutual click intent. Tokens in good subqueries have strong coherence.

$$MCI_{i,j} = \frac{N_{i,j}^{i,j}}{N_{i,j}^{i,j}+N_{i}^{i,j}+N_{j}^{i,j}+1}$$

$N_{i,j}^{i,j}$	number of historical clicked <query, ads keyword> pairs that query contains both token vi , vj and ads keyword contain vi , vj
$N_{k}^{i,j}$	number of historical clicked <query, ads keyword> pairs that query contains both token vi , vj and ads keyword contain vk

Note that the number is aggregated across all <query, ads keyword> pairs

The data structure I used here is undirected graph. For every input query, each token represents a node in the graph. The edge 'E' is the MCI score between any pair of tokens. In the undirected graph V, I cut the whole graph into two parts, Vs and V\Vs. For the features I selected, f1 represents the sum of MCI in subquery Vs. When f1 has a larger value, the subset Vs has better subquery. f2 is the sum of subquery divided by edge weights, which is equivalent to normalization. Similarly, when f2 has a larger value, Vs has better subqueries. f3 gives the maximum value of MCI in subset Vs. A better f3 value means that Vs has better subqueries. f4 is a max score based on crossing subset. When f4 has larger value then f3, it means the good subqueries do not belong to any subset. I have to redo the cutting.

query : Beach Camping Outdoor Chair

$$f_1 = \frac{\sum_{v_i, v_j \in V_s} MCI_{i,j}}{\sum_{v_i, v_j \in V} MCI_{i,j}}$$

Vs: sub query tokens
Es : sub query edges

$$f_2 = \frac{\sum_{v_i, v_j \in V_s} MCI_{i,j}/|E_s|}{\sum_{v_i, v_j \in V} MCI_{i,j}/|E|}$$

$$f_3 = \frac{\max_{v_i, v_j \in V_s} MCI_{i,j}}{\max_{v_i, v_j \in V} MCI_{i,j}}$$

$$f_4 = \frac{\max_{v_i \in V_s, v_j \in V \setminus V_s} MCI_{i,j}}{\max_{v_i, v_j \in V} MCI_{i,j}}$$

6.1 Click Intent Rank(CIR)

CIR quantify the contribution of each token to query intent and indicate how important token v is in the query. The intuition is that important tokens can generate good sub query. Given a query q with n tokens, a directed graph G(V,E) is constructed. The function is below:

$$e_{i,j} = P(v_j \mid v_i) = \frac{N_{i,j}^{i,j} + 1}{N_{i,j}^{i,j} + N_{i}^{i,j} + 1}$$

which indicates how likely is having the token vj in keywords if the token vi is already existed in the query and keyword given clicked pair found in search log.

6.2 PageRank

PageRank gives web pages a ranking score based on links from other pages. The intuition here is that higher scores given for more links, and links from other high ranking pages. To implement this, start each page with a rank of 1.0. On each iteration: each page contributes to its neighbors its own rank divided by the number of its neighbors:

$$\text{contrib}_{\text{p}} = \frac{rank_p}{neighbors_p}$$

After that, set each page's new rank based on the sum of its neighbor's contribution:

$$\text{new} - \text{rank} = \sum contribs * d + (1-d)$$

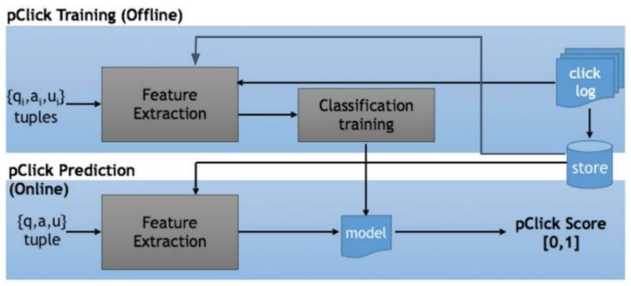
7. Ads Ranking

To better rank ads, I calculated relevance score, which measures how relevant query is to keywords in ads. In my project, this is the number of words match query over total number of words in key words. The method used in industry is that use human judge to label (query, Ad) pairs extracted from search log and use supervised learning with more features: term frequency, inverted document frequency, click through rate and impressions.

pClick indicates the "clickability" which can be defined as the probability of click or estimated Click Through Rate(CTR). Given a {query, ad, user context} tuple, determine its "clickability". Essentially it's a classification problem using supervised learning where the label is 1 for clicked ads, 0 for non-clicked ads. pClick Features extracted from search log and stored in key-value store. The features I selected including, User IP (certain groups love ads), User DeviceId (Certain users love ads), AdId, QueryClassification_+_AdClassification (if query category matches ads category, the probability of click will be high), etc.

7.1 PClick Model training

Feature engineering pipeline includes four parts: generate search log, generate features from search log, store features in key value store and generate training data. Data for both online and offline remains the same {q, a, u}. q refers to query, a refers to adds candidate, u refers to users(ip, divides id). The data is from click log. I did extraction of features first, and stored them to key values store. One part data comes from key value store and users. After that, I trained model. Referring to online training, the input remains the same as {query, ads, users}. I generated key according to these, and read features from key values store. I sent vectors which match keys to model. The model will give the prediction of pClick, which value is between 0 and 1, indicating the probability of being clicked.



8. Evaluation

There are two ways to evaluate the word2vec algorithm. One way is that since we have the values for word similarities, we can allow people to decide how similar they are, which is human judgement. Another way is comparing with an open source benchmark, for instance, Glove, http://wordvectors.org/suit.php. Overall, in my project, I got really good results for this calculation.

For MCI and CIR online training, using the AB test to determine if the good subqueries would generate more clicks than bad subqueries. I calculated click rate for these two different subqueries. My click rate has a wide range which indicates that my MCI and CIR are well-performed.

For the whole project, evaluation is based on CTR, RPM (revenue per 1000 impressions), CPM (cost per 1000 impressions), which are all good on over 90% searches.

8. Conclusion:

In this project, I developed a search Ads workflow, which supports the following: Query understanding, Ads selection from inverted index, Ads ranking, Ads filter, Ads pricing and Ads allocation. I implemented Feature engineering for Query understanding and click prediction. In addition, I implemented Query rewrite with word2vector algorithm. I finally predicted Query intent with pageRank as features and click probability with impression count, click count and category as features. Through this project, the data has demonstrated how a website like Amazon can serve people by differentiating advertisements according to individual’s behaviors.

9. Citation:

Robinson, H. (2010, April 26). CAP Confusion: Problems with partition tolerance. Retrieved from http://blog.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/

Scalable, H. (2012, September 18). DISTRIBUTED ALGORITHMS IN NOSQL DATABASES. Retrieved from https://highlyscalable.wordpress.com/2012/09/18/distributed-algorithms-in-nosql-databases/