Ask Yahoo!

Pranav Chaphekar
Courant Institute
New York University
pc2353@nyu.edu

Urjit Patel
Center for Data Science
New York University
up276@nyu.edu

Sean D Rosario
Center for Data Science
New York University
sdr375@nyu.edu

I. Context & Motivation

Search Engines has now become the essence of everyday life to get the required information in easiest way. Today, we now are so dependent on these interfaces that it is hard to imagine a single hour without access to a Search Engine, whether it be Google, Bing, or Yahoo. Because of these very high dependence on the Search Engines, it is essential that Search Engines provides the latest and most accurate information in the fastest and understandable format to the user. Due to high competition, being unable to provide so can have serious repercussions in terms of user retention.

We believe that, connecting people to already acknowledged information source could prove a great help for any individual. Yahoo! Answers is one of them. It contains more than 4 Million questions and its best answers. Providing a Search Engine which takes the user question and provides required information from this information source could prove very useful. In contrast to our current Search Engines which takes the users questions and shows the result in terms of links to other pages or web sites, we developed a search engine which directly provides the well proven answer, thanks to Yahoo.

II. Introduction

Our main objective was to develop a search engine which can work as a answering machine for user query. We aimed to develop it user friendly, easy to interact and quick for user convenience and satisfaction. We provide user with nice front end with search box in middle, where user can type his/her question and get back answers immediately below the search box. While user is writing question, we assist user by suggesting next most probable word. We also understand that user tends to make mistakes while typing questions, hence we also correct the query with spell checker. Once we receive the question, we display the best answer from our data set for his question below the search box.

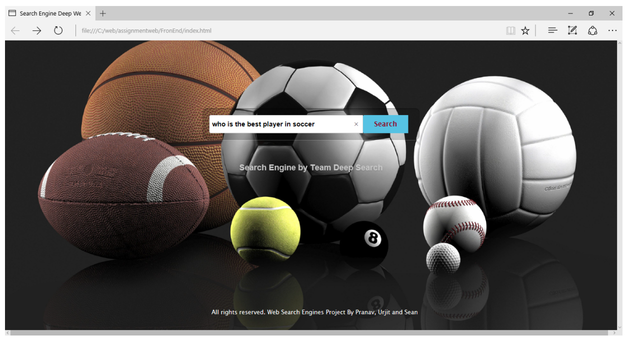
Search Engine Deep We
who is the best player in soccer
Search
Search Engine by Team Deep Search
All rights reserved. Web Search Engines Project By Pranav, Urjit and Sean

Fig. 1. Ask Yahoo! Front End

Technically, our development starts with data cleaning, data processing and creating necessary index files from the raw Yahoo! Answers corpus[1]. We used Hadoop MapReduce to create the index files required for probabilistic models which are explained in detail in following sections. On receiving the user query, we rank all questions in our database using cosine ranker and display the answer for the top ranked question to user. We evaluate the performance of our tool on a test set created with the help of our friends at NYU.

III. Data Understanding and Processing

We used Yahoo Data set for our Question Answer Search Engine. Initially, the raw data was divided in two big xml files having 4 million question-answers and 12 GB of size. We implemented the XML parser to parse the entire data set and converts into bunch of tab separated text files after doing the data cleaning. Data cleaning includes removing unnecessary characters, and formatting. From many features of the original xml file such as question, best answer, other answers, main category, sub category, technical terms, post time etc., we only took the question, best answer, and main category.

We decided to focus on the Sports category and filtered out all the sport related questions from the original large corpus. Sport categories included were as below,

$1.Basketball$
$2.Football(American)$
$3.Football(Soccer)$
$4.Cricket$
$5.Baseball$
$6.Tennis$
$7.Golf$
$8.Rugby$
$9.Cycling$
$10.AutoRacing$
$11.Swimming - Diving$
$12.FantasySports$
$13.WinterSports$
$14.OtherSports$

We have around 200,000 total-question set and 1,200,000 total unique terms.

IV. Architecture

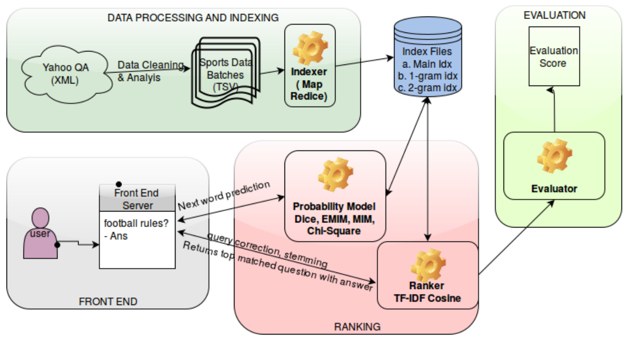
DATA PROCESSING AND INDEXING
Yahoo QA (XML)
Data Cleaning & Analyis
Sports Data Batches (TSV)
Indexer (Map Redice)
Index Files a. Main Idx b. 1-gram idx c. 2-gram idx
EVALUATION
Evaluation Score
Evaluator
Front End Server
football rules? - Ans
user
Next word prediction
query correction, stemming
Returns top matched question with answer
Probability Model Dice, EMIM, MIM, Chi-Square
Ranker TF-IDF Cosine
FRONT END
RANKING

Fig. 2. Ask Yahoo! Architecture

Above diagram shows the main components in our search engine, which are Indexing, Ranking, Evaluation and Front End. We process Yahoo Question Answers data in advance using data processing and indexers to extract the necessary information to rank all questions based on user queries. User query tokens are processed and received by ranker to score set of question-answer documents. Probability models process the query tokens to predict next most probable word in the user query. Evaluation is done on a set of user questions where we compare the top matched question-answer document with the golden truth.

Indexer

Indexer are the objects which we create to incorporate our prior knowledge about the raw data into easy accessible objects before the client facing, to make sure that we can get the information as fast as possible and as accurate as possible during user interaction. Indexer is an important component of any web search engine as it would be the main reference for knowledge source during the user interaction with search engine. It becomes very essential how and what information from raw corpus we store into our indexers.

We are creating three index file in our search engine.
- Main Index File (With Split and Merge)
- Unigram Index File (With Hadoop Map Reduce)
- Bigram Index file (With Hadoop Map Reduce)

A. Main Index File

We read all the questions and answers for Sports domain and create main index file. We use split and merge logic to deal with large corpus of questions and answers. We are mainly storing three objects in our main index file.

1. Word Dictionary We create a word dictionary which has id to word mapping for each unique word appearing in our question set

$wordid(int)->word(sting)$

2. Posting list This is the main object of our index file, which is the dictionary containing word id mapping with all question ids in which it appears. We use this object to find the top documents for user query.

$wordid(int)->list\_of\_question\_ids(intlist)$

3. Documents Documents are the objects where we store the raw text data for our questions and answers. We store these data with the help of three dictionaries,

1. Question Dictionary
$word\_id(int)->word(sting)$

2. Answer Dictionary
$word\_id(int)->word(sting)$

3. Term Frequency and Document Frequency Dictionaries
$word\_id(int)->value(double)$

- **Split and Merge :** We apply split and merge logic to make sure that we can create index files from large corpus of questions and answers without running out of the memory. While doing Split, we keep the posting list in different index files and then after scanning is done through the entire corpus, we merge these small index files into one index so that we can get the full fledge posting list. While splitting we de-refence the objects that we no longer need(E.g. Posting List, which is the main reason for out of memory error).

B. Unigram count Index File

It contains the dictionary for unigram counts. Each key is a word and the value corresponding to that key is a number for how many times that word appears in the corpus

C. Bigram count Index File

It contains the dictionary for bigram counts. Each key is group of two words and value corresponding to that key is a number for how many times that pair appears together in the corpus.

- **Hadoop Map Reduce:**

We used Hadoop Map Reduce to create index file for unigram count and bigram count. We utilize NYU's HPC dumbo server for our purpose.

Mapper : Mapper reads one by one each lines and creates a key-value pair with tab as a separator. For unigrams, the key is a word and for bigrams, the key is pair of words appearing together in same question. Value is just 1 for all keys.

Sort Mechanism : Hadoop has a sort mechanism running between mappers and reducers, which sorts the key,value pair based on the key value.

Reducer : Reducer receives the sorted key,value pairs and sum over the values which is 1 for the same key. Hence, output of the reducer is the key and value which is the count of that particular key (unigram or bigram).

Ranking

Ranker is the component which uses the query tokens as an input and scores the set of documents to find the top search results for a given user query.

Document Selection: An important factor is which set of documents we consider for scoring. Since we have more than 110K set of questions and answers, it would be a very costly operation to score all the documents. We use the posting list object from main index file to get the subset of documents for ranking. We first applied the greedy approach where we considered only those documents in which all the query terms appears (after stemming and query correction). But we notice that, it is very likely that user will have few words which would not present in our true data set, but we should not reject that question from the consideration. We can also not consider all the joint set of documents from all the query tokens as posting list for stop words such as why, what, the would be very long.

Hence, we came up with the solution, where we first find the important words apart from the stop words from the query tokens, and then took the joint set of document ids from the posting list of those important terms.

Scoring documents: We used TF:IDF Cosine similarity to calculate the score for each document. We have the TF and IDf values for each term of the document stored in our main index file.

$$\text{cosine }(D_i,\text{Q}) = \frac{\sum_{j=1}^{t} d_{ij} Q_j}{\sqrt[2]{\sum_{j=1}^{t} d_{ij}^2 \sum_{j=1}^{t} Q_j^2}}$$

Where t is the total number of terms. i is the index of the document.

Evaluation

To evaluate our search engine, we had to come up with evaluation logic which can give us some score based on the performance on user questions. As we are mainly concerned with only the top result for user question, we can not apply the conventional methods such as precision, recall etc to evaluate the performance. Hence we asked few of our friends to help us generate the test set. We took help from 5 friends and provided each of them a set of 10 true questions from our dataset. We asked them to write all 10 questions with some other way such that meaning remains the same. By doing so, we generated 5 different versions for each of the true 10 questions. Our final test queries set was of 50 questions, which we pass to our search engine one by one. We notice that how many questions out of 50, our search engine could able to give us true result which one of the 10 original question.

$$\text{Score} = \frac{1}{|D|}\sum_{i=1}^{|D|} \mathbb{1}(if\_predicted\_correct)$$

Where D is the total number of test queries.

Model	**Score**
Without Query Correction	0.34
With Query correction	0.46
Query Correction+ Stop word removal	0.70

Front End

We designed a front end with html, java script, css where user can interact with our search engine. User can ask question in a box given in the center and hit enter or search for the answer. On receiving the user's question, we display the top questions matched, along with its best answer in the html format.

Extra Features

Query Correction We implemented the algorithm to find edit distance for query token. Once user write a query, we find the nearest word from our dictionary with least edit distance. This dictionary is generated on the basis of the unique terms in the corpus; which makes sure that nouns, etc. are also handled properly. Also we correct the words like "nooooo" which has many unwanted "o"s in it using pattern matching; which makes sure that most of the typing errors are corrected.

Next word suggestion We also developed a probability model which uses the unigram and bigram count index files and finds the most probable next word for the user query. We tried different four probability models,

1. Dice Co-efficient Gives more preference to the commonly co-occuring words.

2. Mutual Information(MIM) Give more importance to the specialised words.

3. Expected Mutual Information(EMIM) Gives more preference to the commonly co-occuring words.

4. Chi-Square Give more importance to the specialised words.

We prefer Dice Co-efficient over others because it's computationally faster and it's easy to implement; and in our search engine, we expect user to ask the common Questions and not specialized Questions, hence we feel that Dice co-efficient serves the purpose.

V. Challenges Faced

1. Big Corpus We had to deal with around 200000 documents, which is really big number. We faced some memory problems during our initial face, which we later solved with split-merge and Hadoop MapReduce.

2. Data Cleaning As we are dealing with real natural language corpus, we noticed too much noise in the data, such as incorrect spellings, unwanted characters and symbols. We did the cleaning part on questions as we are ranking QA pairs based on questions, but we would still see scope of cleaning in answers part and verifying the truthfulness of the answers.

3. Front end integration for query completion We also found some difficulties in integrating our query correction and completion model with the HTML front end due to lack of expertise in front end part; and hence we could not integrate that in our final model

VI. Future Work

There is a possibility of future expansion and improvement in our current tool. Listed below are few methods that we would like to implement,

1. Word Vectors : Currently we are using inverted indexer with word TF-IDF values to find the top documents for user query. In this method, we do not capture the context or semantic meaning of the word. We believe that using word vectors can make difference here. We can use any available methods such as count based vectors or Deep Learning neural methods such as embedding through language modeling [8] or word2vec [7] (CBOW or Skipgram) or glove to get the contextual word vectors.

2. Neural methods to generate Sentence Representation: We can use techniques such as simple average over all word vectors, Convolution, Recurrent Neural Networks to get the Sentence Vectors, which can later be compared using any similarity methods to get the score of each document for user query.

3. Attention Neural Methods for QA: Currently we are displaying the answer from the available knowledge source without any modifications. In case the answers are very big or some articles, then we can use Recurrent Neural Networks with Attention [6], to find the most important line from the document for user's question.

Context based answering using RNN Search Engine for Articles and Stack overflow

VII. Individual Contribution

- **Urjit:-** Project Idea and proposal, Data set selection (Yahoo), Cluster Analysis within the data set and sports subcategories selection, Hadoop MapReduce Implementation for Probabilistic model (unigram and bigram index files), Data Cleaning in MapReduce, IndexerInvertedQuery(Next Document), Evaluation logic,Project Report.
- **Pranav:-** IndexerInvertedQuery(Split and Merge, ConstructIndex), Ranker(Cosine), Front End(HTML Output), Probability model(Next Word Prediction), XML Parsing, Query Correction(Edit Distance) and Cleaning.
- **Sean:-** Evaluation Platform Implementation, Tried query completion integration with front end, Python to Java interfacing.

VIII. Acknowledgment

We would like to thank Professor Cong Yu and Professor Fernando for their guidance in project selection and data set selection.

References

[1] Yahoo Dataset : https://webscope.sandbox.yahoo.com/catalog.php?datatype=l

[2] W. Bruce Croft, Donald Metzler, and Trevor Strohman, ”Information Retrieval in Practice By”, published by Pearson Education, 2015

[3] Stopwords list: http://www.ai.mit.edu/projects/jmlr/papers/volume5/lewis04a/a11-smart-stop-list/english.stop

[4] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman, Mining of Massive Datasets, 2014

[5] Wikibooks contributors, ”Algorithm Implementation/Strings/Levenshtein distance,” Wikibooks, The Free Textbook Project

[6] Karl Moritz Hermann, Tom Koisk, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, Phil Blunsom. 2015. Teaching machines to read and comprehend. In Advances in Neural Information Processing Systems, pages 1684-1692.

[7] Mikolov et. al. 2013. Efficient Estimation of Word Representations in Vector Space

[8] Bengio et. al. 2003. A Neural Probabilistic Language Model.

[9] Porter's Stemming Algorithm refer http://snowball.tartarus.org/algorithms/porter/stemmer.html