Trip pal: A search engine that combines all aspects for a trip

Rui Zhou **Zefan Fu** **Feiyang Liu**

1 PROBLEM

Preparing for a perfect trip and getting all the information easily of a dream destination is always a problem for travel enthusiasts. In the real world, when travelers are planning for a trip, they need to go to different websites to gather information. For example, they need to go to Accuweather to find local weather of their destination, go to Yelp to find restaurant, go to Google map to find shopping locations, and go through Wikipedia to go through attraction information. The whole lookup process is very messy and time-consuming, and general search engines such as Google and Bing cannot fulfill this specific interest-oriented search. Therefore, we want to build a search engine that could provide a one-stop service that can provide many information that a traveler wants to know.

In order to build such a search engine, we formulate the problem into an information retrieval model. Firstly, we selected several open datasets that meet our needs, which include the Yelp open dataset that have restaurant and shop information [1], the countryside vegetarian restaurant information [2], and the countryside museum, zoo information [3]. Each of the dataset has over 30,000 records. We also embedded some APIs such as weather info, local time, and Google Map into our website in order to enrich our system. Secondly, we made some pre-process to the raw dataset we fetched online and stored them into MySQL database. Finally, we used the document ranking knowledge we learned in class such as TF-IDF and cosine similarity to rank our retrieved documents. We also did some research on IR document ranking and used some other methods such as Kendall's Tau to further adjust and improve our ranking results. Besides, we made some process on the query such as stemming, stopword removing, word spell checking before doing the document retrieval in order to make our system more user friendly.

2 METHOD

2.1 Frontend

Our frontend is developed based on HTML5, CSS, JQuery, Jinja2, and Javascript. We provide a general search page which can return all categories of relevant results that matches user's query. The system also provides advanced search pages for users to search their query in a specific category. For example, if the user go to the advanced restaurant search page and search for "alcohol", the system will return top 5(can be changed by users) restaurants that sell alcohol based on the ranking. Besides the basic functionality, we also add some other tricks to make our system more user-friendly. Users can set the number of result they want to be retrieved, and each result will be displayed on the web page with general info such as name, address, type, along with a "more info" button. When the user click the "more info" button, they will be directed to a new web page which contains more information about that result, such as the weather, current local time, open hours, ratings, price, and a Google Map with a marker on that location. This is very useful for trip planning, as travelers don't need to go to different websites to gather information. In addition, if a user want to restrict the search result into one specific city, he/she only needs to add phrase "in xxx" when doing the query, and our system will only return the result in that specific city. In this way, users can easily find information they want. For example, a Chinese history lover can not only find the top Chinese history museum across the country, but could also look for Chinese history museum near Ann Arbor. The snapshots of our system is in the appendix.

2.2 Backend

In the backend, we mainly focus on data processing, page ranking and database maintenance. Before we set up the website, we did some preprocessing work to the raw data. After filtering spam information and stemming the text, we build a relational database using the processed data and use sphinxsearch to parse the query. In order to improve the accuracy, we use TF-IDF algorithm after getting a rudimentary result from sphinxsearch. After that, we use Kendall's Tau to evaluate and improve the search result.

2.2.1 Preprocessing.

We have two datasets for this website and they are directly downloaded from the internet. They didn't fit our website so good at first, so we did some preprocessing on them.

The first thing we did is filtering out the spam information in the datasets. The spam information can be deceptive while ranking. Two main problem we encountered when dealing with spam are keywords stuffing and misleading words. For example, an auto-shop mentioned around the shop there are many delicious food

such as pizza, hotdog and hamburger, so when we search the keyword "pizza", this shop ranks quite high because it has a good rating. Apparently, that rate is for auto-service rather than food, so in this case our rank was manipulated by the misleading word. In another case, a pizza store has some bad comments and low rate, but it has a high rank which is beyond our expectation. That is because when searching the keyword "pizza", it simply counts the matching keywords, so the phrase such as "awful pizza" also counts, making the store ranking surprisingly high. Most of the spam information comes from the column such as "comments", "introduction" and so on, so when using sphinxsearch to count matches, we just count the matches in the column "name", "category", "menu" and so on.

Another thing we did when preprocessing the datasets is we use nltk library in python to remove the stopwords and stemming the text. When user searches the keyword "to", there may be thousands of results with nearly same rank. We consider this kind of searching a "meaningless" search since it can't provide any useful information to the user and is distracting. So, before we set up the database, we removed all the stopwords. After doing so, we stem the text so when user searches "Chinese", all results related to "Chinese" and "China" will show up.

When a user enters a query, we also make a similar pre-process on it. Besides removing stopwords and stemming, we checked for "in xxx" phrase in the query to see whether the user wants to look in a specific city. We also add spell checking function to auto-correct spell errors. For example, if a user wants to look for American cuisine but typed "Americen", our system will indicate the spell error and return results based on "American".

2.2.2 Building the database

After finishing the preprocessing job, we started to build the database using mysql. Since we have two datasets, we built two tables with the following design.

Table 1: cityrestuarant: The primary key is business_id and no other key constraints

Field	Type	Field	Type
business_id	varchar	is_open	Int
address	text	categories	Text
attributes	text	city	varchar
review_count	int	name	varchar
longitude	float	latitude	Float
type	varchar	stars	Double

Table 2: museums: The primary key is museum_ID and no other key constraints

Field	Type	Field	Type
museum_id	varchar	museum_name	varchar
street_address	text	zip_code	varchar
state	varchar	city	varchar
phone	varchar	Longitude	float
revenue	float	latitude	float
income	float	museum_type	varchar

After building the two tables and importing the datasets into them, we used sphinxsearch to build indexes on the databases and parse the query. When a query was sent to backend, we use Python to call sphinxsearch and get the preliminary result. In sphinxsearch, we have different matching and ranking configurations for different tables. For the table museum, we match the query in the fields Museum_name and Museum_type and we rank the results based on firstly stars and secondly how many query matches are found in the tuple, since the stars reflect the quality of the shop directly so we should consider it first. For the table cityrestuarant, we match the query in the fields categories, attributes, name and type, and we rank the result based on number of matches and then revenue, since revenue can show whether a museum is well maintained but is not the most important thing to consider.

Because the datasets we use is not json format, so after getting the search result from sphinxsearch, we used some Python code to jsonify the results, making it easy to read and process for the frontend.

2.3 Ranking with TF-IDF

Although we get a rudimentary search result from sphinxsearch, the recall is not very high and sometimes the ranking is not very accurate. We need to use TF-IDF to improve the ranking result. In order not to make longer document weight more, we used normalized TF-IDF:

$$w_{ik} = \frac{\mathrm{tf_{ik}} \log\left(\frac{N}{n_k}\right)}{\sqrt{\sum_{k=1}^{t} \left[\log\left(\frac{N}{n_k}\right)\right]^2 (\mathrm{tf_{ik}})^2}}$$

Where $\mathrm{tf_{ik}}$ is the frequency of term k in document i, N is total number of documents, n_k is the number of documents that contains term k.

After computer w_{ik} for each i and each k, we used cosine similarity to computer the document similarity as the following equation:

$$Sim(D_i, D_j) = \sum_{k=1}^{t} w_{ik}\, w_{jk}$$

Then we re-rank the result using $Sim(D_i, Q)$. After doing so we found that in some case the ranking result improved a lot, while in some case, the performance decrease. In order to further improve the ranking performance, we combined the ranking result given by sphinxsearch and the ranking result given by normalized TF-IDF, compute the final ranking score using the following equation. The documents have higher ranking score rank higher.

$$\text{Ranking Score} = (1 - R)\log\left(\frac{10}{\text{Rank given by Sphinxsearch}}\right) + R\sum_{k=1}^{t} w_{ik}\, w_{jk}$$

Where R is a weight factor and range in (0, 1).

2.2.4 Kendall's Tau

We now have a comprehensive equation to compute the ranking score, but we don't know the best value for R. So, we need an evaluation system to help us evaluate the ranking result and improve the performance. After doing some research, we find Kendall's Tau is a good method to evaluate how close two ranking is. It has several properties:

•If agreement between 2 ranks is perfect, then KT = 1

•If disagreement is perfect, then KT = -1

•If rankings are uncorrelated, then KT = 0 on average

And it is computed by the following equation:

$$\text{tau} = \frac{n_c - n_d}{0.5n(n-1)}$$

Where n is the number of documents, n_c is the number of agreement and n_d is the number of disagreement.

In order to use Kendall's Tau, we firstly manually rank some queries and set them as standard. Secondly, we tuned the value of R and rank on the same queries. Finally, we computed the Kendall's Tau between the ranks and the standard ranks, and we select the R whose ranks has the biggest Kendall's Tau to the standard ranks. At last the value of R is 0.8.

3 RESULTS

3.1 Query on 3 cities with combined store and museum results

Since our store and restaurant dataset only supports three cities, Madison, Pittsburgh, and Las Vegas, we can obtain combined results of both museum and store type data when we search query related to the three cities.

3.1.1 Precision

All the results that we have returned contains at least on word match in its fields. When the query contains multiple words, we have chosen to return a document if all the words in the query has a match. For example, if we query "Las Vegas", word "Las" returns 22937 documents with total hits as 27302 and word "Vegas" returns 22930 documents with total hits as 27577. The "match all" forces the final returned results to be the intersection of the two separate results.

In addition, we have enabled matching root of query words. For example, if we search "Chinese", document with "China" in it will show up in our returned result even when there is no "Chinese" in the document.

Although we can search cuisine keyword like "Chinese", "Mexican" or "Thai", there is no information provided for us to know how many returned results are actually related to this cuisine since all we know is the name of the restaurant or the attribute field contains this word. Here we assume that once there is a match in the name or attribute, it is a true document. Then, our precision is 1 since all returned document has a match in either name or city or attribute field. Similar situation for query as "gift shop" or "hair" as we have demonstrated in the presentation.

If we limit the query word to only the name of these cities, we can obtain information about the number of true results returned so that we can calculate precision. There are cases that the name of a museum may contain another city's name.

The number of returned documents are listed below in tables.

Table 3: Store and Restaurant type return results

City	Madison	Pittsburgh	Las Vegas
True documents	2711	5275	22930
Total returned	2714	5275	22930
Precision	99.89%	100%	100%

Table 4: Museum type return results

City	Madison	Pittsburgh	Las Vegas
True documents	77	75	53
Total returned	122	80	53
Precision	63.11%	93.75%	100.00%

After carefully finding out what is wrong, we have obtained information listed in below tables. For museums in Madison, the list is too long to be displayed but we have all the matching filed counted. Among the 45 wrong results, 31 have a match in Museum Name but the location is in other cities. Madison is also a name for person besides a name for a city. That's why the precision is low. In addition to 31 errors in Museum Name, there are 6 errors in Institution Name and 8 errors in Legal Name.

Table 5: Pittsburgh Museum error results

City	Matching Content	Matching field
SEWICKLEY	TUSKEGEE AIRMEN MEMORIAL OF THE GREATER PITTSBURGH REGION	Museum Name & Institution Name
BRADFORD	UNIVERSITY OF PITTSBURGH	Institution Name
CHICAGO	3511 N PITTSBURGH AVE	Street Address
CONNELLSVILLE	107 S PITTSBURGH ST	Street Address
CONNELLSVILLE	299 SOUTH PITTSBURGH STREET	Street Address

Table 6: Madison store and restaurant error results

City	Matching Content	Matching field
Las Vegas	Madison Avenue Bar & Grille	Name
Pittsburgh	Madison Ave Specialty Cakes	Name
Pittsburgh	814 Madison Ave	Address

All the errors occur when there is a match in other fields but the location is not the same city. The "Madison Avenue Bar & Grille" is located in Las Vegas although it has Madison in its name.

From these errors, we came up with a solution to add a "in" word in the query. For example, we type in "art museum in Madison" if we want to search art museum only in city Madison. When we identify that there is word "in" in the query. We separate the city name out and finds a match only in the city field.

3.1.2 Recall

As can be seen from above, all true results are returned. Since we limited that we only return results with all words in the query find a match, recall is 100%.

3.2 Query on 20 cities with only museum results

We also want to know how our search engine performs when we search other cities for the museum information. When we are searching name of other cities, only museum type document will be returned and we do not need to consider shop and restaurant type.

We have applied same methodology with section 3.1. The precision of 20 queries are listed in below table.

Table 7: precision of 20 queries

City	True documents	Total returned	Precision
Los Angeles	162	178	91.01%
San Diego	97	106	91.51%
San Francisco	121	125	96.80%
Boston	104	109	95.41%
New York	305	395	77.22%
Washington	253	369	68.56%
Seattle	92	96	95.83%
Atlanta	82	89	92.13%
Ann Arbor	36	39	92.31%
Detroit	63	69	91.30%
Fargo	21	25	84.00%
Houston	139	151	92.05%
Orange	44	61	72.13%
Orlando	21	22	95.45%
Napa	16	16	100.00%
Austin	108	115	93.91%
Champaign	12	19	63.16%
Berkeley	37	41	90.24%
Chicago	200	202	99.01%
Miami	81	93	87.10%

From these data, we can see that many query words that are common in address and museum names are having a low precision whereas other words are having precision above 90%.

To improve, we use the same "in" key word to separate the city name and finds a match of the city name only in city field.

3.4 Compare performance before and after "in" key word

When we find "in" keyword in the query, we take out the city name and finds a match only in city field. However, it could not enhance our precision to 100% for all query. For example, museums in Orangeburg and Orangeville will be returned when we search "museum in orange". But we also want root match be used in shop and restaurant type of documents.

3.4.1 3 cities with combined store and museum results

The enhanced precisions are in below table

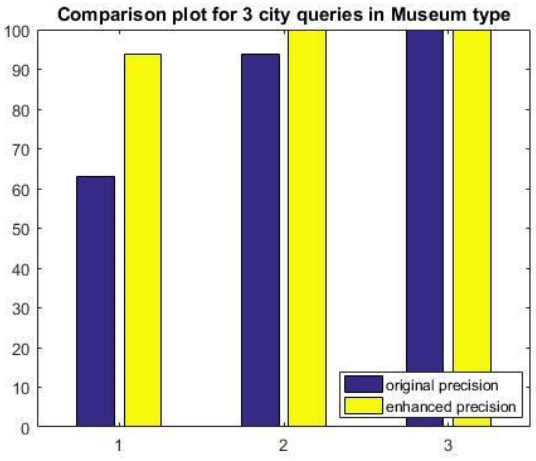
Table 8: Store and Restaurant type return results

City	Madison	Pittsburgh	Las Vegas
True documents	2711	5275	22930
Total returned	2711	5275	22930
Precision	100%	100%	100%

Table 9: Museum type return results

City	Madison	Pittsburgh	Las Vegas
True documents	77	75	53
Total returned	82	75	53
Precision	93.90%	100.00%	100.00%

The plot of museum numbers is in below figure to provide a comparison.



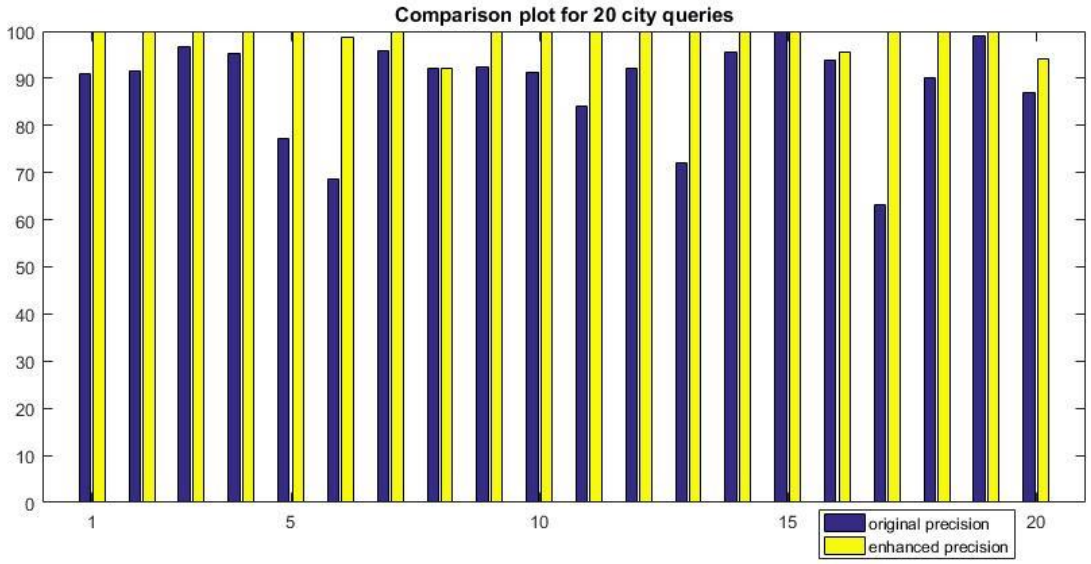


Figure 1 Comparison plot for "in" key word enhancement in 20 queries of museum type

3.4.1 20 cities with only museum results

The enhanced precisions are in below table

Table 10: enhanced precision of 20 queries

City	True documents	Total returned	Precision
Los Angeles	162	162	100.00%
San Diego	97	97	100.00%
San Francisco	121	121	100.00%
Boston	104	104	100.00%
New York	305	305	100.00%
Washington	253	256	98.83%
Seattle	92	92	100.00%
Atlanta	82	89	92.13%
Ann Arbor	36	36	100.00%
Detroit	63	63	100.00%
Fargo	21	21	100.00%
Houston	139	139	100.00%
Orange	44	44	100.00%
Orlando	21	21	100.00%
Napa	16	16	100.00%
Austin	108	113	95.58%
Champaign	12	12	100.00%
Berkeley	37	37	100.00%
Chicago	200	200	100.00%
Miami	81	86	94.19%

The plot of these numbers is in Figure 1 to provide a comparison.

4 DISCUSSION

In summary, our search engine has been able to return relevant documents based on user queries of city names and attributes. We rank the documents by a combination of star rating and TF-IDF. For the user interface, we provide additional map and weather information for each document returned.

We also provide advance search page for user to query on certain type of information only whereas all types of information are returned and shown on the general search page. In addition, we applied a "in" key word method to enhance our precision by splitting the city name after "in" out and match it only with city field in the database. With such method, searching "cake in Madison" will not return cake store in other city with its address contains the word "Madison".

For future improvement, hotel and flight dataset can be added to our database to create a more useful search engine for trip planning. Our current functionality is not enough for really planning a trip. If we want to solve the problem described in the first section, we also need more data about stores and restaurant in other cities. In addition, we can record how many times users have clicked "more information" button of a returned document. If many users want to see map and attribute field for a shop or restaurant, we would like it to be given a higher rank for future queries. To achieve this, we can combine the star rating, TF-IDF and the users clicks to rank documents.

REFERENCES

[1] Yelp Dataset Challenge. (n.d.). Retrieved April 23, 2017, from https://www.yelp.com/dataset_challenge

[2] Vegetarian & Vegan Restaurants. Retrieved April 23, 2017, from https://www.kaggle.com/datafiniti/vegetarian-vegan-restaurants

[3] Museums, Aquariums, and Zoos. Retrieved April 23, 2017, from https://www.kaggle.com/imls/museum-directory

APPENDIX

You searched:cake

Restaurant and Shop Info:

Name: The Cake & Cookie Spot

City: Pittsburgh

Address: 2108 Murray Ave

Rating:5.0

Categories:['Desserts', 'Bakeries', 'Food']

More info

Museum Advanced Search Page

Area or museum type... Search

Single Restaurant Info:

Name: The Cake & Cookie Spot

Address:2108 Murray Ave

City:Pittsburgh

Rating:5

Alcohol:No

Good for meal:No

Business parking:Yes

Ambience:Not available

Categories:['Desserts', 'Bakeries', 'Food']