Business POI Recommendation Algorithm

Adrian Guzman afg30

December 2019

Abstract

Search engines can be found on almost every website on the internet. It has become so important for us to be able to navigate our data and quickly find what we are looking for. In this project, I create a search algorithm that will suggest businesses to a user based on categories and attributes specified by the user. The algorithm will use a weighting system to evaluate how well each business matches different parts of the query, which include category, predicate statements, and proximity. The goal of this implementation is to return businesses the user maybe did not directly search for, but that they might be interested in. Based on the experiments conducted, I conclude that my attempt to build an efficient business recommendation system was successful, as the algorithm consistently suggested relevant businesses.

1 Introduction and Background

In the age of big data, it has become increasingly important to be able to navigate data efficiently. More specifically, to search data efficiently and intelligently. This is where search engines come into play. Search engines allow us to navigate massive amounts of data extremely quickly and efficiently, giving us humans a shortcut for finding data we are interested in. The key to an optimal search engines is for it to be able to return data relevant to the user and their specified preferences. It will be my goal in the following project to architect and develop a search engine that satisfies this specification.

2 Problem Formulation

I will be creating a search engine that runs on business metadata contained in a data set provided by Yelp. The data set contains information on approximately 200,000 unique businesses. Each business is stored as a json object, with attributes such as business_id, name, address, latitude/longitude, stars, categories, etc.

The search engine will take an input of the form (category, keywords, literals, number_of_results), where *category* is the category of business to search for (eg.

'Restaurant', 'Salon'), *keywords* is a set of keywords describing the business (eg. 'Nightlife', 'Mexican'), *literals* is a set of predicate statements the business should ideally satisfy (eg. $stars > 5$, $RestaurantsDelivery = True$), and *number_of_results* is the number of business to return.

In addition, the user will be able to specify in the query if they want to include proximity weighting. If this option is set to true, then the algorithm will take into account the distance between the business and the user's current location and factor this into the final weighting, ultimately weighting businesses that are closer to the user higher.

The engine will then run this query over the business data and return business that *best match* the query. The idea here is not to just return results that match the query exactly, but to return businesses that the user might be interested in based on the query. To accomplish this the engine will use a weighting system, where weights are assigned to businesses based on how well they match individual parts of the query. This allows each query term to affect the final weight, but does not limit the results based on any individual query term (eg. the businesses returned do not have to match every query specification).

The first step will be to parse the provided data file into a KV-store indexed by *business_id* to provide quick access to any business's attributes we want. Once this index is created, it becomes apparent that we need to create another KV-store, this time indexed by category. Each business contains a list of categories the business has been classified under. These categories will be essential when the user constructs their query, so we want to be able to quickly find businesses labeled with a given category. We go ahead and create this new KV-store, this time indexed by category. The values for this KV-store will be a list of *business_ids* that belong to the respective category.

Now we have two KV-stores, one indexed by *business_id* with around 200,000 keys, and another indexed by categories with around 1600 keys.

Additionally, we need to create one more KV-store, which stores the range of all attributes with numeric values, which in this case is only *stars* and *review_count*. This KV-store will be used to normalize the respective values when executing ranged queries, as we will see later.

Now we have the data in the format we want and can proceed to create the search engine that will query said data.

2.1 Related Work

There were a few articles that I read on weighting, specifically on how to apply weighting to a search algorithm, but I did not use any of the techniques described in these articles as they did not seem to apply to my situation. I more just gained an understanding of how weighting works so that I could implement my own weighting formulas. The first article was an analysis of how Yelp actually uses review_count and rating to weight businesses. The second article was a more in depth explanation of how a query algorithm should weight individual portions of the query. The second was much more helpful in understanding how the weighting system works.

Jaisingh, Drishtii, et al. "A Brief Analysis of Yelp Search Ranking Factors." The Synpost, 13 Nov. 2018, synup.com/blog/analysis-of-yelp-search-ranking-factors/.
Broer, Rolf. "Search Engine Algorithm Basics." Moz, Moz, 11 Apr. 2012, moz.com/blog/search-engine-algorithm-basics.

3 Algorithm Description

The algorithm follows the basic structure below:

```
def mainAlgorithm(category, keywords, literals, k):

    #Step One
    Search businesses by matching categories/keywords:
        Weight business appropriately
        Only return businesses matching > 1 of the categories/keywords

    #Step Two
    Modify the weights obtained above for each business by:
        How well they match each provided literal
        Proximity to user (if specified)

    #Step Three
    return top k businesses by respective weights
```

So it's pretty simple in principle, but now I'll take a deeper look into how each of the weighting functions works.

3.1 Categorical Weighting

This function has the option to use linear or fuzzy search (matching syntactically similar strings). The default is fuzzy because this allows for more freedom when comparing categories against the search query.

The function begins by iterating through the KV-store indexed by category, comparing each category with each category specified in the query. Remember this KV-store takes the form:

{'categoryOne': ['businessID', 'businessID', ...],
'categoryTwo': ['businessID', 'businessID', ...]}

Where each of the businessIDs in the list belongs to the corresponding category.

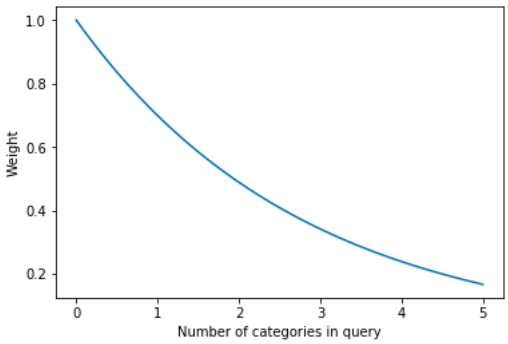
```
category_score_dict = {}
iterate through each category in KV-store:
    ratio = 0
    for queryCategory in queryCategories:
        ratio += compute fuzzy similarity between category and queryCategory
    weight = ratio normalized by number of categories in query
```

```
    assign weight to category
filter out categories with weights less than threshold
    threshold = (1/1.43)**(number of query categories))
```

In short this function assigns weights to each category available based on its similarity with the query categories. If the query categories are ['restaurant', 'japanese'] we would get the following KV-store, with the weight denoting how similar each category is to both of the query categories.

```
{'active': 0.16, 'golf': 0.0, 'life': 0.09, 'sum': 0.18, 'food': 0.0, 'specialty': 0.14, 'restaurant': 0.69, 'dim':
0.0, 'seafood': 0.23, 'chinese': 0.31, 'ethnic': 0.08, 'import': 0.24, 'sushi': 0.17, 'bars': 0.17, 'japanese': 0.5
9, 'services': 0.15, 'insurance': 0.43, 'financial': 0.21, 'heater': 0.24, 'kitchen': 0.08, 'home': 0.09, 'shopping
': 0.15, 'local': 0.08, 'water': 0.25, 'garden': 0.16, None: 0.0, 'plumbing': 0.15, 'bath': 0.17, 'shipping': 0.15,
```

Then it filters the list so only categories with a weight higher than a calculated threshold remain (essentially retaining matching categories). The threshold is set using the following function ((1/1.43)**(number of query categories)):



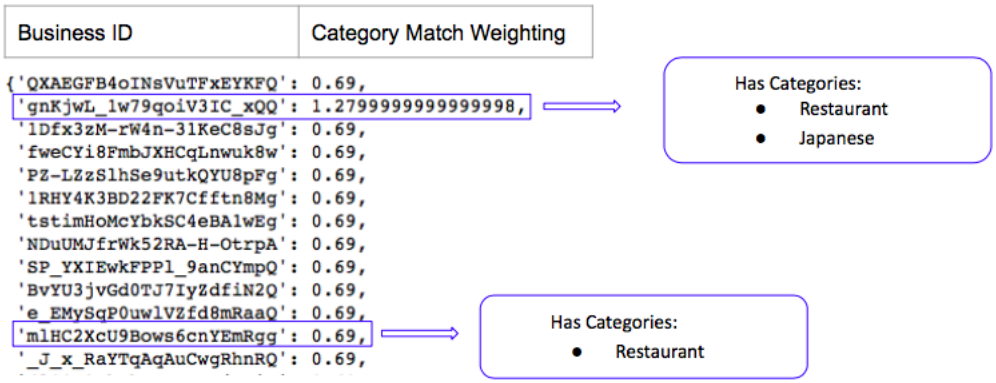
This function is used because as there are more query categories, the similarity between each business category and the query categories will presumably decrease, as similarity is calculated as one value denoting similarity between each category and *all* of the query categories.

After filtering the categories by this threshold, we get the assumed matching categories:

```
{'restaurant': 0.69, 'japanese': 0.59, 'assistant': 0.5}
```

You can see that not all of the categories will actually be relevant, as we are using fuzzy search, however these 'irrelevant' categories will receive a lower weighting and consequentially will be much less likely to make it into the final cut of suggested businesses.

The function finishes off by assigning each of the filtered categories weights to businesses that belong to the respective category, where businesses that belong to more than one of the categories recieve the sum of all weights of categories they belong to. This would return a KV-store like such:



We observe that the business with the higher weight belongs to both of the query categories, while the rest belong to only one of the categories and as such are assigned a lower weight.

3.2 Literals Weighting

The KV-store returned above is then passed into the literal weighting function. This function iterates through each business in the KV-store, evaluates how well the business satisfies each predicate statement, and operates on the preexisting weight accordingly.

```
for business in businessWeightKVStore:
    for literal in queryLiterals:
        #val = value of attribute of business
        #val_max = maximum value of attribute across all businesses
        #threshold = threshold value set in literal query
        normalizingFx = ((val_max+val)/val_max - (threshold/val_max))/2

        if literal condition is satisfied:
            if operator is equality:
                binary/ranged valued attribute:
                    increase weight of business by 1
                categorical valued attribute:
                    increase weight of business by normalized fuzzy similarity
            if operator is ranged:
                increase weight of business by normalizingFx()
        else literal condition not satisfied:
            if attribute is string typed:
                decrease weight by normalized fuzzy similarity
            if attribute is ranged(numerical):
                decrease weight by (1 - normalizingFx())
```

The logic here for operating on the weight is pretty difficult to understand by staring at the code. It is better understood through analyzing some examples.

Assuming that more stars is better, stars=4 and 5 satisfies the query so we increase weight by 0.5 or 1 respectively. stars=3 and 2 do not satisfy query so we decrease weight by 0.4 and 0.6 respectively. Since 2 matches the query less than 3, the weight is decreased more.

```
query = 'stars >= 4' #Max of stars is 5

#Business Stars #Adjust weight by
4               :               0.5
5               :               1
3               :              -0.4
2               :              -0.6
```

Assuming that more reviews is better, reviews=4000 matches the query threshold so weight is increased by 0.5. Reviews = 5000 and 6500 do not satisfy the query so the weight is decreased by 0.44 and 0.35 respectively. Notice the weight is decreased less for the business with more reviews. Reviews = 3000 and 500 satisfy the query so weight is increased by 0.43 and 0.28 respectively. Notice the reviews=3000 is closer to the threshold so the weight is increased more.

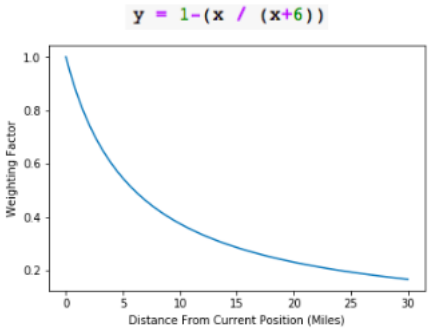
```
query = 'review_count <= 4000' #Max of review count is 8000

#Business Stars #Adjust weight by
4000            :             0.5
5000            :            -0.44
6500            :            -0.35
3000            :             0.43
500             :             0.28
```

3.3 Proximity Weighting

This functionality is actually nested inside of the literals weighting function but I will analyze it separately for the sake of transparency.

By default the algorithm uses proximity weighting, but the user has the ability to disable this functionality. If it is enabled, the algorithm first retrieves the latitude and longitude of the machine the query is being executed on. Then it iterates through each business (in the same loop literals are weighted in, avoiding multiple traversals of the data) and computes the distance in miles between the two sets of coordinates using the Haversine formula. This distance is then plugged into the following function:



This returns a weight scaled by distance, where the weight is really only relevant if the business is within 20-25 miles of the users current location. The business weight is then incremented by this proximity weight.

Now that we have finished weighting each of the businesses, all we have left to do is return the k businesses with the highest weights. This is done easily by sorting the business-weight KV-store by weight and taking the first k tuples. Some important attributes such as 'stars', 'review_count', etc. are added to each tuple, and the KV-store is then converted to a DataFrame for easy visualization.

3.4 Performance

Without proximity weighting the algorithm runs in approximately O(1) time, taking around 1 second for 200,000 businesses. The number of categories searched does not affect run time because of the category indexed KV-store, however the amount of literals in the query slightly affects the run time. O(1) time is only noticeably affected when the number of literals exceeds 15 or so.

With proximity weighting the algorithm runs in $O(1) < x < O(\log n)$ time, because additional resources are required to compute the Haversine formula for each business. The algorithm takes around 2 seconds with proximity weighting for all 200,000 businesses.

As far as correctness goes, there is no quantitative measure I can think of that will assess how well the algorithm performed. The best way to do this in my opinion is to study the results of sample queries and judge whether appropriate (relevant to the user) businesses were returned.

4 Experimental Demo

Now we have an algorithm, let's see how well it works! We'll start out with a baseline query of ['restaurants', 'japanese'] to make sure our categorical weighting works as expected:

	Name	Stars	Number of Reviews	Score	City	Categories
gnKjwL_1w79qoiV3IC_xQQ	Musashi Japanese Restaurant	4.0	170	128.0	Charlotte	Sushi Bars, Restaurants, Japanese
v-scZMU6jhnmV955RSzGJw	No. 1 Sushi Sushi	4.5	106	128.0	Pittsburgh	Japanese, Sushi Bars, Restaurants
PkDghu4aan2_wxrhXjTEgg	MiraKuru	3.5	16	128.0	Calgary	Nightlife, Italian, Restaurants, Japanese, Lou...
SJBzyJDCR_f6dx5tpYAABA	Kibo Sushi House	4.0	15	128.0	Toronto	Sushi Bars, Japanese, Restaurants
jX9DocoiY4Bo9EUkaTSqvg	China AAA	4.5	149	128.0	Henderson	Restaurants, Hawaiian, Chinese, Japanese, Poke...
ecJri9ozyke4dOCWulZiRQ	Nishikawa Ramen	4.0	427	128.0	Chandler	Asian Fusion, Japanese, Restaurants, Ramen, Ta...
4B8VnRAstRRshxiUzm9yPw	Maiko Sushi - DDO	4.0	51	128.0	Dollard-des-Ormeaux	Restaurants, Sushi Bars, Japanese

We get back a bunch of Japanese Restaurants, which is what we expected. Now we add the predicates $stars >= 4$ and $review\_count >= 100$ to the query:

	Name	Stars	Number of Reviews	Score	City	Categories
hihud--QRriCYZw1zZvW4g	Gangnam Asian BBQ Dining	4.5	3449	129.250587	Las Vegas	Barbeque, Korean, Asian Fusion, Tapas/Small Pl...
UPlYuRaZvknlNOd1w8kqRQ	Monta Ramen	4.0	2604	129.149976	Las Vegas	Noodles, Ramen, Restaurants, Japanese
7sb2FYLS2sejZKxRYF9mtg	Sakana	4.5	1659	129.143376	Las Vegas	Buffets, Sushi Bars, Japanese, Restaurants
pH0BLkL4cbxKzu471VZnuA	SUSHISAMBA - Las Vegas	4.0	2355	129.135062	Las Vegas	Bars, Dim Sum, Japanese, Asian Fusion, Restaur...
A-uZAD4zP3rRxb44WUGV5w	Soho Japanese Restaurant	4.5	1512	129.134571	Las Vegas	Restaurants, Japanese, Asian Fusion, Sushi Bars
sqRX-XLIhx4rs2c1TpBf8A	Raku	4.5	1457	129.131277	Las Vegas	Desserts, Food, Seafood, Restaurants, Japanese
1qkKfqhO8z2XMzLLDFE96Q	Kodo Sushi Sake	5.0	414	129.118807	Scottsdale	Sushi Bars, Japanese, Restaurants
S-oLPRdhlyL5HAknBKTUcQ	Harumi Sushi	4.5	1107	129.110314	Phoenix	Seafood, Sushi Bars, Restaurants, Japanese
umXvdus9LbC6oxtLdXeIFQ	Sweets Raku	4.5	1088	129.109176	Las Vegas	Food, Desserts, Wine Bars, Bars, American (Tra...
uqLqbAvBdRDc-qS4hpslXw	Yama Sushi	4.0	1912	129.108529	Las Vegas	Sushi Bars, Restaurants, Japanese

Now we get Japanese restaurants with high ratings as well as lots of reviews, which is again what we expect. The red boxes highlight the weighting at work. Gangnam Asian BBQ has a slightly lower rating than Kodo Sushi Sake, so the algorithm assigns it a lower weight for that attribute. However it has many more reviews than Kodo, and the algorithm weighting determines that this difference in number of reviews is more significant than the difference in rating, so when the weights for both predicates are combined, Gangnam receives a higher weighting and thus appears higher in the recommendation list.

Now we can expand the literal search by adding a literal with a True/False value. Now our predicates are $stars >= 4$ and $review\_count >= 100$ and $RestaurantsDelivery = True$

	Name	Stars	Number of Reviews	Score	City	Delivery	Categories
O3OH5IEFMPtz7mPKakPZ3Q	Japaneiro	4.5	420	130.069166	Las Vegas	True	Asian Fusion, Seafood, Restaurants, French, Ja...
YkOCo5ipV2he2WXlAlZb-A	Jaburritos Sushi Burritos	4.5	376	130.066531	Las Vegas	True	Sushi Bars, Mexican, Asian Fusion, Restaurants...
w6zW6glyg1sI5V6Wag_SYg	Teppan Bento	4.5	326	130.063536	Las Vegas	True	Asian Fusion, Japanese, Teppanyaki, Restaurants
EvxTD0ETbbFbFjUMWhHblw	Oh Curry	4.5	312	130.062698	Las Vegas	True	Japanese Curry, Japanese, Restaurants
UpSHAMogY9ubMGbuNAV5nw	Sushi Wa	4.5	171	130.054253	Vaughan	True	Asian Fusion, Sushi Bars, Restaurants, Japanese

We see that now we don't get any 5 star restaurants, and the average number of reviews for the suggested restaurants is much lower than previously, however the suggested restaurants all offer delivery, so we got a list of suitable Japanese restaurants. Again, the suggested results aren't required to offer delivery, but they are weighted higher if they do. If many literals are added, the interaction of the weighting for each attribute will ideally suggest businesses that match the most literals as closely as possible.

Now let's turn on the proximity filtering for our previous query of japanese restaurants with more than 4 stars

	Name	Stars	Number of Reviews	Score	City	Categories
jmTirQw-n4V_Z4g1-RE9iw	Kung Fu Tea	4.0	46	130.436428	Cleveland	Bubble Tea, Japanese, Restaurants, Coffee & Te...
X6chSU0KBGHGNxZ8UH_qag	Kenko Sushi & Teriyaki	4.0	126	130.431321	Cleveland	Asian Fusion, Sushi Bars, Bubble Tea, Food, Et...
rRTz0zVbjmrEY8dEZmPJTA	Otani Noodle - Uptown	4.0	167	130.352284	Cleveland	Restaurants, Ramen, Japanese, Noodles
v2a8hl1Tsi1SgFKFWD39qQ	Wasabi Restaurant	4.5	11	129.903012	Cleveland	Sushi Bars, Chinese, Japanese, Restaurants
APPZiUz_CEvR_--qY-tbig	Pacific East Japanese Restaurant	4.0	358	129.846690	Cleveland Heights	Sushi Bars, Restaurants, Malaysian, Japanese
YyyqcbVyoBo4wPsgowUtUQ	SASA	4.0	97	129.715050	Cleveland	Japanese, Restaurants
d_tubLVx2pnTZXvXFfcOeQ	Ariyoshi Japanese Restaurant	4.0	19	129.641076	Cleveland Heights	Japanese, Restaurants
cuYl-talioYVb619BlmokA	Pho Ha Nam	4.5	37	129.562714	Cleveland	Japanese, Restaurants, Vietnamese

Now we get restaurants near our location that also satisfy the query as close as possible. Most of these locations are within a few blocks of me, which is pretty neat.

Now we can apply proximity filtering to the previous query of Japanese restaurants with predicates $stars >= 4$ and $review\_count >= 100$ and $RestaurantsDelivery = True$

	Name	Stars	Number of Reviews	Score	City	Delivery	Categories
rRTz0zVbjmrEY8dEZmPJTA	Otani Noodle - Uptown	4.0	167	131.352284	Cleveland	True	Restaurants, Ramen, Japanese, Noodles
YyyqcbVyoBo4wPsgowUtUQ	SASA	4.0	97	130.715050	Cleveland	True	Japanese, Restaurants
d_tubLVx2pnTZXvXFfcOeQ	Ariyoshi Japanese Restaurant	4.0	19	130.641076	Cleveland Heights	True	Japanese, Restaurants
jmTirQw-n4V_Z4g1-RE9iw	Kung Fu Tea	4.0	46	130.436428	Cleveland	False	Bubble Tea, Japanese, Restaurants, Coffee & Te...
X6chSU0KBGHGNxZ8UH_qag	Kenko Sushi & Teriyaki	4.0	126	130.431321	Cleveland	False	Asian Fusion, Sushi Bars, Bubble Tea, Food, Et...
k-XN0KuOWPU5xZIfXOVSIA	Ninja City Kitchen and Bar	3.5	158	130.375120	Cleveland	True	Nightlife, Bars, Japanese, Restaurants, Asian ...

Now there are 4 weights at work here (not including the categorical weights): stars, review count, delivery, and proximity. It's difficult to justify and explain the interaction between these 4 weights, but looking at the results here we can conclude that we got relevant results to the query criteria. The top result satisfies all criteria, and is just down the street from me, the second result,

SASA, doesn't quite satisfy the > 100 review count, but it is very close which is why it is weighted so high. Kung Fu Tea actually doesn't satisfy the delivery requirement, but is weighted so highly because it is so close to me. Ninja City doesnt satisfy the rating requirement of >= 4, but it is weighted so highly because the rating is very close to the set threshold, and it satisfies both other criteria.

5 Conclusion and Future Work

Based on the experiments conducted in the previous section and the results observed, I conclude that the algorithm does perform as expected, and honestly better than I expected. It suggests almost exactly what I'm looking for most of the time, which I find really interesting and useful.

Depending on the user preference, one might want to tone down the proximity weighting to increase the strictness of adherence to the literals.

One major downfall to this search engine is that you must have some prior knowledge of the categories. Fuzzy search attempts to lower the knowledge barrier here, but it is only so helpful. For example, the UPS stores has categories ['postal', 'delivery', 'shipping', etc] but it does not have 'mail' anywhere in its categories. So if a user searches for 'mail', the UPS store will not be suggested, when it definitely should be. The solution to this lies in NLP neural networks. This allows the algorithm to determine similarity based not on syntax, but on meaning. For example using fuzzy comparison on mail and postal returns 0.4, while using an NLP neural net returns a similarity of 0.7. There are many python libraries that allow access to NLP nets, so the implementation is easy, and I actually did implement this, but there is one major downfall: run time. Using the NLP net to compare each category with the query categories increases the run time dramatically, to over a few minutes. Obviously this won't work, which is why I did not include this in the presentation.

One possible solution here would be to distribute the work across threads and/or processors.

I enjoyed working on this project and I think I created a useful product that can enjoy some practical applications. Obviously it needs some adjustments and fine tuning, but I will definitely keep it in my catalog.