RentRunner: A Domain-Specific Search Engine for Apartment Hunting

1 Introduction

Domain-specific search engine, or vertical search engine, commonly exists in the websites with specific topics, like shopping, health information, and scholastic literatures. With the specific knowledge in a particular domain, the search engine have a better understanding of user's request and make more meaningful recommendations [2]. However, we have observed a lack of such fine-tuned search engine in the area of apartment search. Craigslist, as a popular website that host the information of ranging from apartments to used automobiles, fails to handle many problems within the domain of apartment search. For instance, no post filter exists in Craigslist, and, as a result, spams and low-quality posts are everywhere which make the search more difficult. Another issue with Craigslist is its strictly keyword-based retrieving algorithms. A user might want to add a specific constraint into her query, like "1-bedroom" and "below $600", a strictly keyword based model will sometimes eliminate the relevant results simply because the keyword is not found in the main body of the posts, which is usually the case with Craigslist, because user can specify information like number of bedrooms, rent in tags.

In this project, we have designed and implemented a domain-specific search engine for apartment hunting, RentRunner, based on the data extracted from it. Through enabling the features of duplicate detection, posts quality evaluation and natural language processing, we target to address the issues stated in above within Craigslist and provides a better user experience.

2 Solution

2.1 System Overview

The overall structure of the RentRunner system can be seen on Figure 1. Documents on apartment postings are collected from Craigslist apartments section - we initially collected all the active Ann Arbor posts available in a list, then followed the links to individual posts to collect the post body, longitude, and latitude, and other information. Individual pages were fetched using the Python package Scrapy, and specific pieces of information from each page were parsed out and written out in a JSON file. The preprocessing module classifies the postings and de-duplicates the posts, and the created metadata is added to the JSON file. This file is then indexed by Solr and made available through a web interface. The web frontend accepts natural language search input from the user and passes it to the natural language processing module, which extracts constraints within the natural language query and
converts it to a new query in Solr syntax with specific filters. The results from Solr are passed back to the web frontend and used to point the locations of each post on

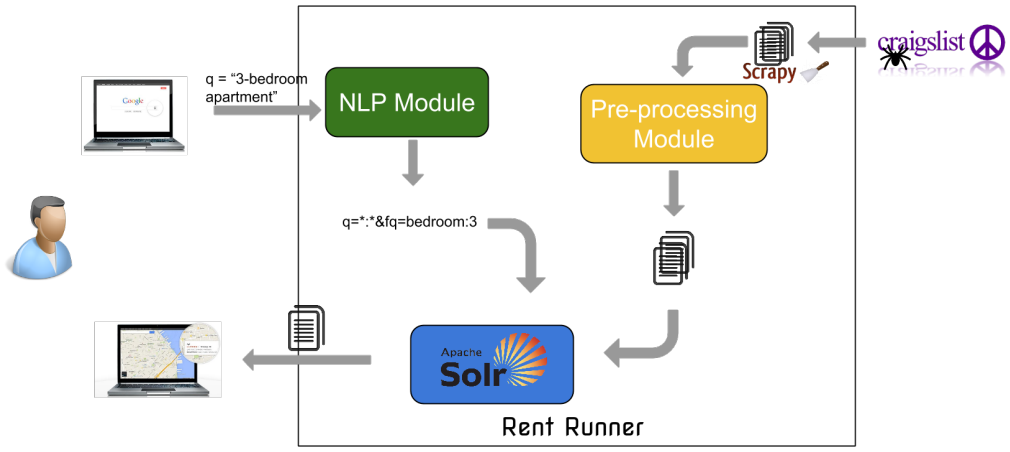

Figure 1: RentRunner System Architecture

the map.

2.2 Data Extraction

Data was collected using the Scrapy python package, with the initial apartment page as the seed page. Results were listed in brief summary on the list page according to post date and additional results paged. Each result contains an id which was extracted and appended to a url to provide a link to the full post. The full posts were retrieved; DOM elements on the page were located using Xpath conventions and their contents transferred to Python objects, to be written to JSON file. Specifically, longitude and latitude information were extracted from the details page of each post. Around 2000 posts were gathered in each of 2 periods, for a total of around 4000 postings.

2.3 Data Preprocessing

2.3.1 Motivation

Before ranking the documents, we wanted to identify near duplicates as well as cluster the documents based on their quality. By using near duplicate detection, our application can help users visualize which ads come from the same business, and potentially provide a greater variety of more relevant results. In terms of clustering, we wanted to provide the user a way to identify higher quality of ad posts, and filter out the posts that had some spam like features. With both these preprocessing steps, users will have an easier time of identifying a variety of higher quality posts.

2.3.2 Near Duplicate Detection

In order to perform near duplicate detection of the craigslist ads, we used a python package called simhash. First, the ads were separated into 3-gram shingles, where each 3-gram phrase is turned into a hash using a hash function. Then, to save on storage, a process called MinHashing is used where each document is now represented by a set of minimum hashes, which are generated by passing the all the hashes representing the 3-gram phrases of the document into another set of hash functions. From here, the size of the intersection for each pair of documents is estimated by computing the proportion of minimum hashes that are identical between the pair of documents.

2.3.3 Quality Based Clustering

Although the ads on craigslist weren't labeled as spam or not spam, we wanted to visually identify spam-like posts through clustering. To that end, we clustered the posts

into 3 clusters: the first cluster indicating "higher quality" ads, the second cluster indicated "medium quality" ads, and the last cluster indicating "lower quality" ads, with the idea being that the "lower quality" cluster would contain all the spam-like posts. The 3 features that we decided were most indicative of post quality were the length of the ad, the ratio of all caps words to the length of the ad, and the ratio of exclamation marks to the length of the ad.
In terms of the clustering method, we used a hard clustering method called the Buckshot Algorithm, in order to prevent the k-means clustering algorithm from generating suboptimal clusters. First, we took out a sample of size 45, which was the square root of the total amount of ads. Then, we used hierarchical agglomerative clustering on the sample of ads with the metric as Euclidean distance, and the linkage criteria as ward. The reason why we used ward as the linkage criteria is because it minimizes the total sum of squares around the centroid when the two clusters are merged, and is the linkage criteria most similar to k-means. Since we wanted 3 clusters as the final output, we cut off the hierarchical tree at 3 clusters. Finally, we computed the mean centroids of all 3 clusters generated from hierarchical agglomerative clustering, and used them as the initial centroids for the k-means clustering. To run the hierarchical agglomerative clustering and k-means algorithm, we used python's scikit-learn library. Below is a visualization of the final clusters, where the green cluster corresponds to the "higher quality" cluster, the red cluster corresponds to the "medium quality" quality cluster, and the blue cluster corresponds to the "lower quality" cluster.

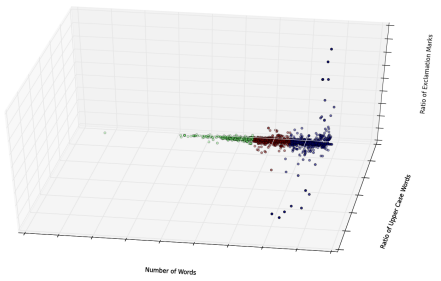

Figure 2: Clustering Visualization

Even though it wouldn't make sense to represent the clusters by the most frequently occurring words, since all the clusters are about the same topic, one example of an ad from the "lower quality" cluster is:

"One Bedrooms for $599 For a limited time ONLY!!!!!!!!!!!!!!!!!! Come in and secure your home TODAY!!!!!!!!!!!!!!!!!!"

2.4 Natural Language Query Processing Module

2.4.1 Motivation

One important feature we think is both promising and critical for a domain specific search engine is the support of natural language query. Unlike general-purpose search engines which target to handle queries in any domain, like Google and Bing, search space for a domain-specific is quite limited, in other words, as search engine designers, we have some, if not all, knowledge about what retrieved documents are like (in our case, posts about apartments), and what information can possibly go into the query (e.g., in our case, number of bedrooms, range of prices). Hence, it is possible for us to enable the feature handling some simple human language queries in our search engine. Granted that existing search engine sometimes provides "Advanced Search" besides the simple search box that allows the user to intentionally specify some constraints for the returned results. The learning curve of "Advanced Search" is manifest and proportionally increases as number of possible features increases.

Therefore, we decided to created a NLP module to discover the latent constraints buried in the human language queries. For simplicity, we implemented 3 potential constraints: number of bedrooms, number of bathrooms and rent. More constraints can be included in a similar way as long as the crawler can be programmed to extract more features from the web pages. The NLP module translates the natural language query into more queries with specific filter queries (fq) before being sent to the search engine. We will demonstrate how effective it is in handling natural language queries in Section 3.

2.4.2 Natural Language Processing

For handling natural language query, we first use Stanford NLP Parser to generate a linguistic parse tree from the query [1]. The linguistic parse tree provides the syntactic structure information as Figure 3 of a given string (usually sentences).

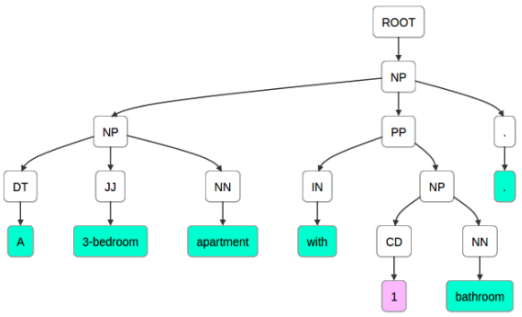

Figure 3: Sample Linguistic Parse Tree

From the tree structure, we can easily obtain the functions and dependencies of each word in a sentence. As mentioned before, the objective of handling the natural language queries in our project is to find the constraints. We pay more attention on noun phrases (NP), especially the descriptive words, like adjective (JJ), within the noun phrases, where constraints might frequently occur here.

The way we extract constraints for number of bathrooms and number of bedrooms are similar. Given the common way to describe such a constraint in human language are enumerable, like "1-bedroom", "2-ba", "3 bedrooms", etc. Once an adjective is found, we check if it is in the form of "-bedroom", or "-bathroom" and so on. If it is, we extract the number and add a filter query on the field "bedroom" or "bathroom", like "fq=bedroom:2". Strings like "3 bedrooms", are simply a combination of Cardinal Number (CD) and noun (NN) or nouns (NNS) in the noun phrases. Once such a combinations are found, we check if the noun or nouns are "bedroom", "bathroom" and so on.

"Rent" is a little bit different because, first it is often not a simple numeric string, but involves a range, for example "below $500", "higher than $1000", and second it does not simply reside in a noun phrase. In search engine Apache Solr, range of fields is supported in filter query. For example, "below $500" can be translated to "[* TO 500]". Thus, we need to correctly extract this constraint. To extract rent constraint, we have discovered a set of possible sentence structures that can hopefully tackle most of the situations. Take "the rent is lower than $600" as an example.

```
(ROOT
  (S
    (NP (DT the) (NN rent))
    (VP (VBZ is)
      (ADJP
        (ADJP (JJR lower))
        (PP (IN than)
          (NP ($ $) (CD 600)))))))
```

Figure 4: Parse Example

The sentence is a combination of noun phrase (the rent) and verb phrase (is lower than $600). Within the verb phrase, the verb "is" is definitely useless, and the adjective phrase(ADJP) is useful. With the adjective, the noun phrase $100 will certainly be extracted, the comparative adjec-

tive (JJR) "higher" is also an important feature to extract. We have enumerated a set of English words describing the notion of "greater than" and "lower than", also we managed to handle the negation such as "not higher than" as well. Besides the constraints, we also discover that the orders and locations of keywords matter. For example, "small bedroom huge living room" might be given a equivalent score as "huge bedroom small livingroom" using Okapi BM25 ranking function. However, the semantic meaning of two documents are exactly opposite. We handle this problem, we have decided to extract each individual noun phrase as a separate query and set a constraint that the words within a noun phrases must be within a certain distance in the retrieved document using the proximity search feature provided by Apache Solr, such as, "q=quiet%20bedroom∼10".

2.5 Solr and Web Frontend

A collection was created in Solr for the craigslist set and its schema.xml file modified to accept the data elements in Scrapy's JSON output. Most elements were indexed as text, while some pieces of information such as rent, longitude, and latitude were assigned float values. In addition, longitude and latitude were merged into a LatLon Location field to allow for geolocation filtering through Solr. Attributes such as number of bathrooms, number of bedrooms, and etc could all be filtered through the Solr query interface. On the front end, user input in natural language would be passed to a webmethod implemented in Python. The concepts in the text would be discovered through natural language methods and used to assign filter values on a Solr query. Communication with Solr was enabled by JQuery's $.ajax method, with a base url pointing to Solr's select method and query urls specifying parameters such as location, rent, recentness, and etc. Relevant results were returned in json format, sorted by relevance to the Python webmethod. This results is passed through to the Javascript frontend and collected in an array.

The front end requests the current location of user, and use the location to retrieve only nearby locations. Returned results containing longitude and latitude data were used to instantiate marker objects in the Google Maps API - then were plotted on map. Metadata generated through clustering of the posts assigns a different marker color for each grade of post. Clicking on each marker would bring up the post body of the listing, and switching to the listview would bring up results in order of relevance in a list. In the list view, next to the summary of each post, are tags identifying the bedroom and bathroom count, as well as the rent at each location.

3 Evaluation

3.1 Accuracy

In this section, we are evaluating how relevant the retrieved documents are after enabling the features we explained in the last section in our domain-specific search engine. In our experiment, 4 Ph.D. students from Electrical Engineering and Computer Science Department in University of Michigan are invited to each make 5 random searches on both our RentRunner search engine and the search engine of Craigslist and identify the relevance of retrieved results to their queries. To guarantee that the test queries are truly generated randomly by the volunteers, no instruction is given to the volunteers before the testing, in other words, volunteers can try whatever queries they want. The content of the queries can be seen in Appendix. Mean Average Precision and Precision are used as the metrics to measure the accuracy of the retrieval. Since the ground truth of relevances of all

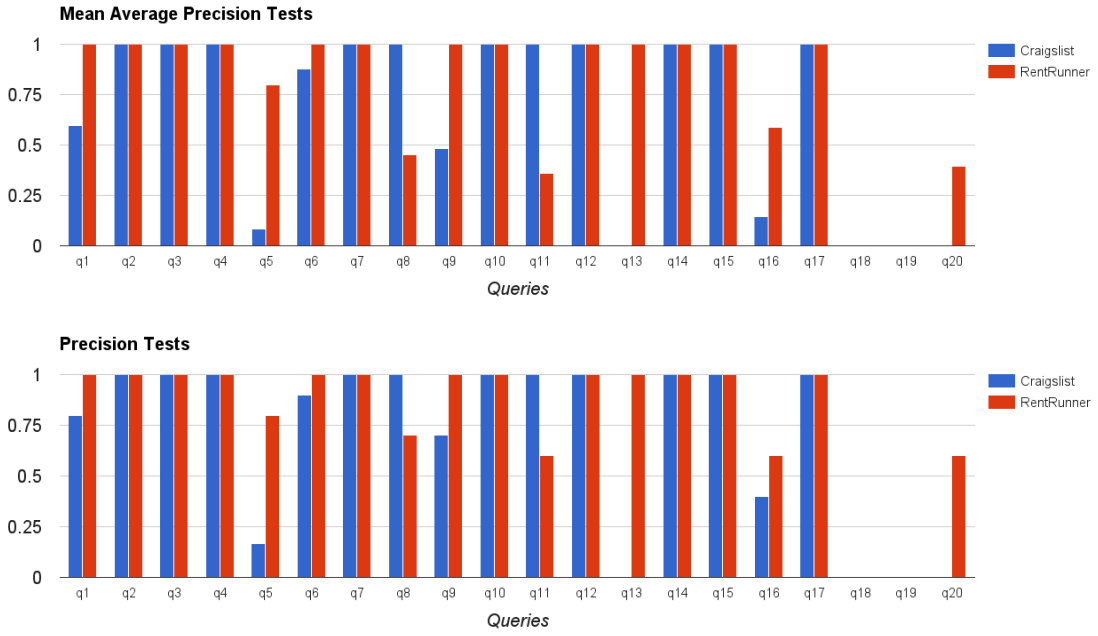

Figure 5: MAP and Precision statistics

documents is not known, Recall can not be measured. Only top 10 returned results are involved when evaluating the accuracy. For those queries who received less than 10 retrieved results, we simply include all the retrieved results in calculating MAP. The results are shown in Figure 5 and will be discussed in section 3.3.

3.2 Performance

In this section, we are verifying that the overall performance is not affected by adding the feature of Natural Language Query Processing. Note that both web page crawling and features of duplicate detection and quality evaluation are executed offline, the response time of real time search should not be affected any of them. The Apache Solr Search Engine and Natural Language Processing Module are both running together on a Lenovo Z510 laptop with Intel i5-4200M CPU, 8GB RAM and 1TB HDD Hard Drive. The 20 random queries from last section are each sent 5 times sequentially to both RentRunner and Craigslist and response time is measured. To minimize the differences of network latency, experimental searches are sent from a remote machine to both the server running RentRunner and the server of Craigslist. The measured average response time statistics for both search engine are shown in Table 1 and will be discussed in section 3.3.

Craigslist	RentRunner
538.5	1107.6

Table 1: Average Response Time in ms

3.3 Discussion

In general, the results show that RentRunner retrieves better results than Craigslist at a reasonable cost. From the experimental results, we have observed that the Mean Average Precision and Precision of RentRunner are better than Craigslist for the majority of the test queries and almost as good for the rest of the queries. For many queries with specific constraints like "Rent is below $1000" (Q13), the statistics show that we have successfully extracted the constraints and retrieved the correct results, whereas such queries are clearly handled

incorrectly in Craigslist. However, there are still a few exceptions that we currently could not handle well especially when the user is providing the queries that have syntax errors or are ambiguous. Take query "rent between 600 and 800" (Q8) for instance, verb "is" is missing in the sentence and the Stanford NLP parser labeled rent as a verb and the subject is missing in this sentence. Hence, our module can not find a field for filter query for this query and, as a consequence, failed to extract this constraint. Another example of such failure is the query "single room apartment" (Q16). It is ambiguous in that the word "room" is too generic, and the system does not know if it is "bedroom" or "bathroom". We can address the issue of ambiguity using the techniques introduced in [3]. In addition, better user experience are reported by the volunteers with the feature of quality evaluation and duplicate detection being implemented on the search machine. The performance tests show that the response time of RentRunner is almost twice as much as that of Craigslist but still acceptable, given that the whole system is running on a personal computer and no specific optimization on performance is done. The module of Natural Language Processing did not significantly slow down the search engine, although techniques should be investigated in the future about how to make this module more efficient.

4 Conclusion and Future Work

4.1 Conclusion

In conclusion, RentRunner appears to be a significant improvement over Craigslist, both in terms of the relevance of the retrieved results and the enhanced user experience. Being able to parse natural language queries allows Rent Runner to handle queries with specific constraints such as upper and lower bounds on rent price, and specific number of bathrooms or bedrooms, and also take into account the order of the terms in the natural language query, whereas Craigslist is unable to handle natural language queries and in general returns less relevant results. In addition, RentRunner lets users view the quality of the results and if they belong to the same business or are duplicates. Thus, even though RentRunner has a longer response time and has some areas of potential improvement, it is still a more user friendly search engine than Craigslist overall.

4.2 Future Work

For future work, some features we could expand on are filtering out results based on more specifications in the query, implementing a more suitable ranking function, and improving the relevance of the results by using relevance feedback. Since each craigslist ad includes location information in terms of the longitude and latitude coordinates, one possibility for improvement would be to allow users to input a query such as: "Within 5 miles of (-83.702364 42.249842), and then return the results that satisfy that location based constraint.
For improving the ranking of our results, we could implement BM25L ranking function as developed by Lv and Zhai, to prevent longer and more informative ads from being unfairly [4]. The equation for BM25L is shown below:

$$S(Q,D) = \sum_{t \in Q \cap D} \log\left(\frac{N+1}{df(t)+0.5}\right) * \frac{(k_1+1)(c_{td}+\delta)}{k_1+(c_{td}+\delta)}$$

$$Where\ c_{td} = \frac{c(t,D)}{1-b+b(\frac{|D|}{avdl})}$$

This ranking function modifies the length normalization component of BM25 by adding a constant , such that longer documents are more favored. Therefore, since

it would be bad to over penalize longer apartment ads, BM25L might be a good improvement over the default ranking function in Solr.
Finally, to further improve the ranking of our results from a set of queries, we could represent queries as vectors and use Rocchio feedback to generated updated query vectors, once we have users to label the results of the queries as relevant or not relevant.

References

[1] Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, volume 6, pages 449–454, 2006.

[2] Daniel R Fesenmaier, Karl W Wöber, and Hannes Werthner. *Destination recommendation systems: Behavioral foundations and applications*. CABI, 2006.

[3] Fei Li and HV Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1), 2014.

[4] Yuanhua Lv and ChengXiang Zhai. When documents are very long, bm25 fails! In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 1103–1104. ACM, 2011.

Appendix

List of queries generated by volunteers

1. parking included
2. safe apartment
3. apartment bus
4. Ann Arbor apartment
5. 2-bedroom apartment with private bathroom
6. on central campus
7. 1-bedroom apartment
8. rent between 600 and 800
9. 1 bedroom apartment
10. quiet apartment
11. apartment university of michigan
12. sublet apartment
13. rent is below 1000
14. furnished apartment
15. near north campus
16. single room apartment
17. heat included
18. $500 per month 2-bedroom apartment
19. $800 per month studio apartment
20. roommate wanted for a 2-bedroom apartment with shared bathroom