D.A.R.W.I.N.

A Natural Language Car Information Search Engine

Gu Yucheng, Duanmu Mingliang

Dec 14, 2021

Abstract

D.A.R.W.I.N. stands for **D**ocumented **A**utomobile **R**etrieval system **W**ith **I**nformation **N**eural network. In this project, a perpendicular IR system on second-hand cars is proposed. A second-hand car dataset is created based on online data. The proposed IR system is designed to support natural language inputs as queries, and returns the ranking on our dataset. Results shows that the MAP of the enhanced algorithm is doubled compared to the BM25 baseline.

1 Introduction

Current car dealer websites are mostly filter-based, which means people must decide on a series of parameters like mileage, drivetrain, make and engine before being presented with choices. However, the fact is most people have no concept of these values, especially for those buying a car for the first time. A descriptive sentence that implicitly conveys the needs of a car buyer is more common in the real world, for instance, "I want a small car which supports Apple CarPlay", "I want a big car that can take my family of four and a dog for long trips". Our system is acting like a sales manager focusing on such fuzzy search to better help nonprofessional car buyers choose the best deal according to their needs. According to our investigation, there are currently no such system on the internet. Darwin is therefore proposed and implemented, in hope to fill the vacancy.

2 Related Works

In this project, the goal is to achieve a perpendicular IR system. There are some works in IR field that does a similar job, but of course on different topics. For instance, in "Rapidly Deploying a Neural Search Engine for the COVID-19 Open Research Dataset: Preliminary Thoughts and Lessons Learned", Zhang et al. [2020] presented a search engine on COVID-19 Open Research Dataset. They used Anserini for indexing, and used seq2seq models to build the ranker. These are the methods we can draw lesson from, as we currently have enough GPU resource at-hand, using neural networks is a possible approach. However, they are also vague on the evaluation, as they insist that the evaluation requires a correct result on the dataset, which they were not sure of. In our case since the dataset is not large, we could have better evaluation methods. In "MedKnowts: Unified Documentation and Information Retrieval for Electronic Health Records", Murray et al. [2021] also works on a specific field, which is clinical documentations. They discussed heavily on the implementations, which is helpful for us when building the web interface.

In their "Document Ranking with a Pretrained Sequence-to-Sequence Model", Nogueira et al. [2020] also used seq2seq models in document ranking. Instead of working on perpendicular fields,

they tested their method on MS MARCO passage and Robust04 datasets, which makes the results more generalizable. The seq2deq model is a encoder-decoder architectured model transforming text to text and then do a reranking. We believe this method is applicable to our situation, if computational power allows.

Prakash and Patel [2015] discussed techniques for query understanding in their report "Techniques for Deep Query Understanding". According to them, query understanding have a few phases including query correction, query suggestion, query expansion, query classification and semanic tagging. We believe for our preprocess of query to work, query expansion and semanic tagging are necessary. These are the aspects we will pay most attention to.

Finally, other methods are also worth considering. Liu et al. [2017] proposed a cascade procedure in their "Cascade ranking for operational e-commerce search". The procedure cascades a few ranking functions in series for training and inference. This method effectively deals with queries involving multiple factors of preferences or constraints.

3 Data

We developed a scraper script that scrapes data from Cars.com. The website maintains a relatively well- structured HTML code, so it is not difficult to extract all the features we need from different locations on the page. Totally 9,975 second-hand cars are collected as dataset, and after cleaning, there were about 9,200 remain.

The formatted data is in JSON form, with only four fields, in which two fields are dictionaries of features. The first step of preprocessing is to expand all features as the columns of the dataset. The second step is to separate simple information like year, car make, and turbo from the name and engine of the car. The third step is extracting drivetrain, fuel type, emission, number and type of cylinders, average MPG and transmission number from the original feature column. In this step, regular expression is frequently used to obtain information from original data, since the format is not uniform across each line of data, for example, "2.0L", "2.0 L", "2.0" can all lead to a 2.0 emission volume. Also, we are mapping drivetrain and fuel type into less categories to simplify classification, for example, "E85 Flex Fuel" and "Flexible Fuel" can be treated as in the same class. In addition, we classify the columns with only logical values indicating whether the car has a certain function into three groups: safety, comfort, and multimedia configuration, so the further filter criteria can be built on these groups.

When deployed, the scraper script can be reused daily to retrieve real-time data as backend. For simplicity, we will keep using the initially retrieved data as our dataset. Table 3 shows some basic statistics of the dataset.

Item	**Statistic**
Total cars	9204
Average price	39279
Number of cars under $60000 (Cheap & Moderate)	8225
Number of cars with full configurations	252
Percentage of 4-wheel drive car	64.05%
Percentage of cars with model after year 2016	83.04%
Percentage of cars with emission volume over 3.0	57.14%
Percentage of electric cars	1.54%

Table 1: Statistics of the dataset

4 Methods

Our proposed system is a combination of natural language processing techniques and information retrieval techniques. The system accepts natural-language-like inputs. For instance, users can query: "I want a car with strong horsepower", or "I want a blue car with steerwheel heating".

4.1 Architecture

The overall architecture of Darwin is shown in Figure 1.

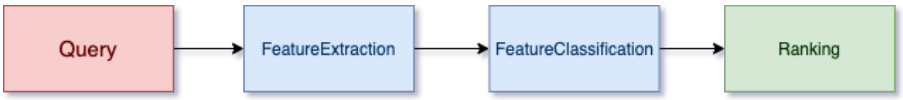

Figure 1: Architecture of the System.

Denote the initial query as $\mathcal{Q}$. The query is first converted by a feature extractor, which finds specific patterns in the query, and extract these patterns as $\bar{\mathbf{E}}$. The extracted patterns, each representing a feature that the user is querying, are then send to a feature classifier. The classifier uses a recurrent neural network R to classify these features into four distinct subclasses. The generated classes are $\bar{\mathbf{C}} = R(\bar{\mathbf{E}})$

After this step, the system has split the initial query $\mathcal{Q}$ into a group of (feature, class) pairs $[\bar{\mathbf{E}}, \bar{\mathbf{C}}]$. These pairs are then sent to the ranking function for further ranking.

Since the first two stages in the architecture are still under development, we also provide an option to pass the query as normal to the ranker. The ranker will then scan through each word w in $\mathcal{Q}$ using the traditional method.

4.2 Dependency Match

To handle natural language inputs, the input query is first passed through a dependency parser. When describing a certain features of a car, it is common that certain short adjective-noun phrases are used. The purpose of dependency match is to capture these phrases in the input query. The data structure of a query after the parser is shown in Figure 2.

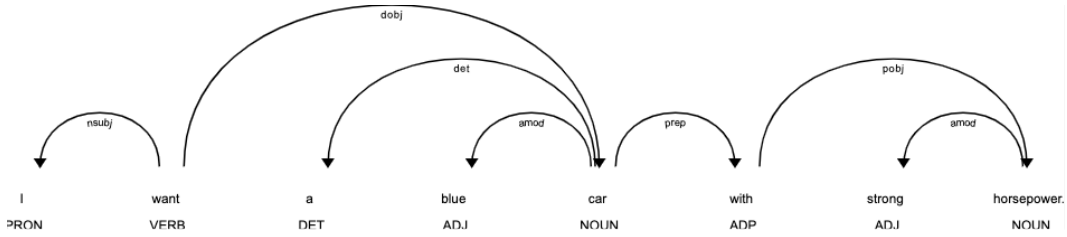

Figure 2: Dependency of a Query.

In Figure 2, syntactic dependencies in the example query are visualised using arcs starting from the head token and ends at the dependent token. The word "want" only has outgoing arcs, therefore it is the root of the dependency tree. By traversing through the arcs, the tree structure $\mathcal{T}$ can be obtained. Dependency matching checks over $\mathcal{T}$ for specified patterns. For example, one of our specified pattern looks for *pobj* after *amod*. In the figure, the pattern would extract [horsepower $\longrightarrow$ strong], i.e. **strong horsepower**.

In out implementation, SpaCy is used to parse the dependency tree and obtain matches of specified dependencies.

4.3 Feature Classification using RNN

After the features $\bar{\mathbf{E}}$ are extracted, to better use these features, we will figure out which feature class does this feature belong to. A recurrent neural network (RNN) is used to achieve this purpose.

4.3.1 Network Structure

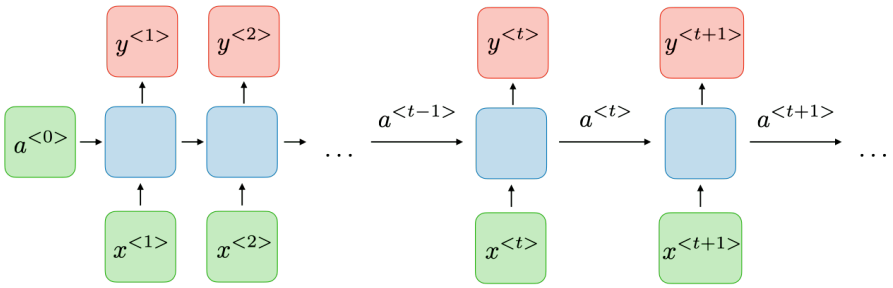

Figure 3: The Architecture of a General RNN. Credit: CS230

Figure 3 shows the architecture of a general RNN. RNN acts like a usual multi-layer perceptron, excepts that its input and output can be variable length sequences. To achieve this, we define the activation $a^{<t>}$ and output $y^{<t>}$ at each timestep t, where we input the t-th element of the input sequence to the network. The activation and outputs are defined as

$$\begin{cases} a^{<t>} = g_1\left(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a\right) \\ y^{<t>} = g_2\left(W_{ya}a^{<t>} + b_y\right) \end{cases} \tag{1}$$

Where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are all coefficients, and g_1, g_2 are activation functions. In practice, the network's input-output mapping can be one-to-one, one-to-many, many-to-one and many-to-many. In our implementation, we choose a many-to-one structure as shown in Figure 4. The hidden size

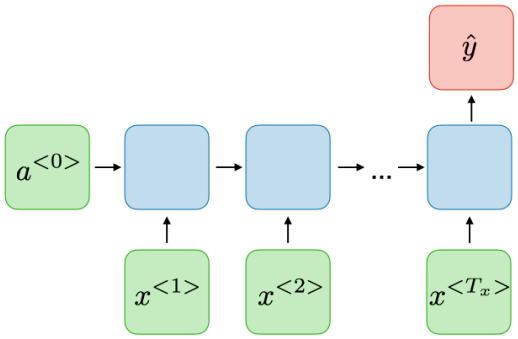

Figure 4: The Architecture of a Many-to-one RNN. Credit: CS230

of the network is set to be 128 based on the prepared training set. The activation function for each hidden state is linear. A LogSoftMax layer follows the output to generate probabilties of each class.

For simplicity, the RNN is designed as a character-level network. The input is a character set in one-hot encoding. We choose 52 English characters adding a few puncts, in total 57 characters as the character set, so each line of input is a 1×57 tensor.

4.3.2 Train Method

Since we have limited time, it is impossible for us to manually curate and annotate a dedicated training dataset. Therefore we took a different approach to bootstrap from existing corpuses.

The main corpuses used for the training are the posts from r/askcarsales on Reddit. We made an assumption that each input feature consists of two words, a noun with a adjective modifying it. With the assumption, we extract all two-word patterns either by fixing the noun or fixing the adjective. Additionally, the extracted patterns are assigned to a class according to the fixed term. For instance, we would assume all phrases ending with "engine" is witin the "power" class.

This method may lead to a model overfitting on the keywords we set. However, we believe that a carefully selected group of keywords can match more than 80% of the queries. Additionally, since the strong correlation between the subsequent two words, this method does extract many useful corpus from the original corpus. Table 2 shows some samples from the dataset. After collection

Regex Pattern	Example	Class
`[\S]*[\s]engine`	EcoDiesel engine	power
`[\S]*[\s]price`	walk-away price	price
`[\S]*[\s]stereo`	JBL stereo	configuration
`black[\s][\S]*`	black Terrain	exterior

Table 2: Samples from the Generated Dataset

and simple cleansing, we have a dataset with in total around 9,000 lines of data, which is enough for a RNN network. Since the dataset is not large, we use bootstrap method on all data to train. The training loss is shown in Figure 5, and the confusion matrix of the final network on the train set is shown in Figure 6.

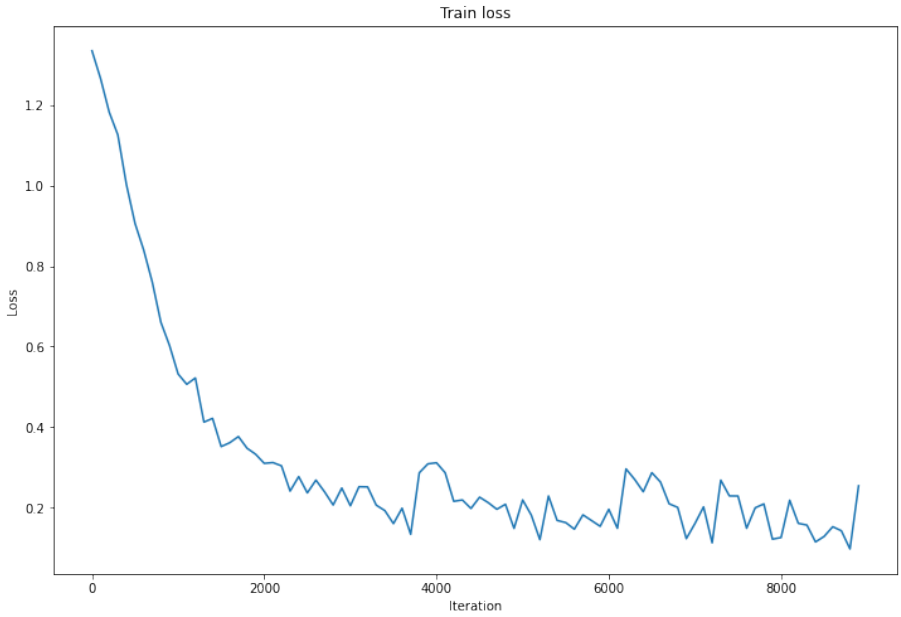

Figure 5: Train Loss of the RNN.

From the feature map, the power and configuration classes are well-fitted, while the price and exterior classes are sometimes confused. This is understandable, because in both classes there would

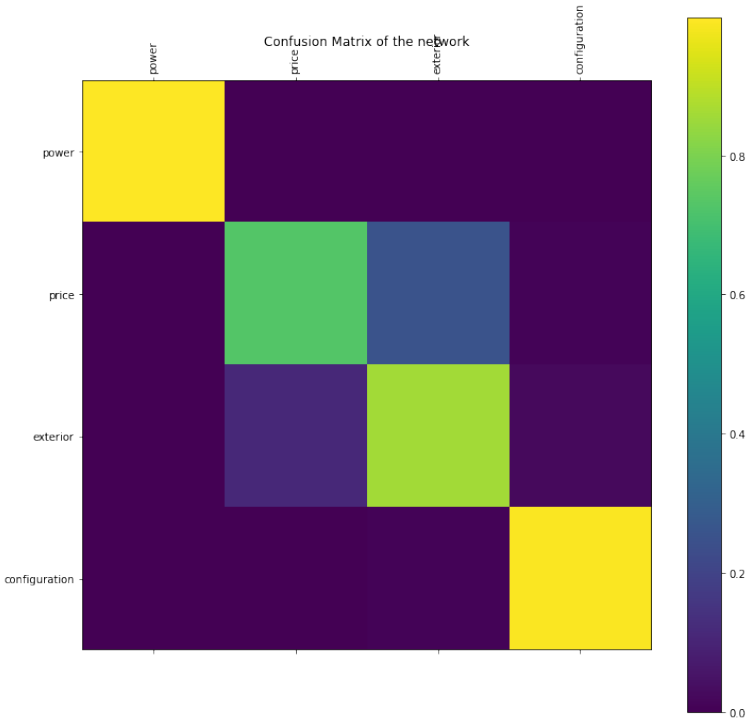

Figure 6: Confusion Matrix on the Train Set.

exist phrases with an adjective modifying the word "car", "SUV", "sedan", etc. The network will easily be confused by the latter noun. But after some tests, this network is confirmed to be robust enough for the coarse feature classification task.

4.4 Feature Search (Ranking)

After proprocessing the raw data, we use the Pyserini package introduced in Assignment 2 to index the documents. Our implementation of ranking function consists of two main parts, the traditional BM25 score part and an enhancement score part based on our neural network's transformation of the query. We do a simple addition on two score parts to determine the final ranking of the document.

4.4.1 BM25

The BM25 part is similar to what we implement in Assignment 2, which takes the form of

$$\sum_{t \in Q,D} \ln\left(\frac{N - df + 0.5}{df + 0.5}\right) \cdot \frac{(k_1 + 1)tf}{k_1((1-b) + b\frac{dl}{avdl}) + tf} \cdot \frac{(k_3 + 1)qtf}{k_3 + qtf}$$

where we use the tuned hyperparameters in Assignment 2: $k_1 = 1.2, b = 0.75, k_3 = 1.2$. Since we use the basic BM25 as our baseline, we simply concatenate every feature in our clean dataset to form a long string as the content of the document. For numerical features, we directly convert them into string; for seven logic values on configurations, we basically use the name of the feature and add "no" before if the logic value is False.

4.4.2 Enhancement Score

The enhancement part takes in the output of our neural network, which is a dictionary with key of extracted feature phrases from the query, and values of four categories of features we defined: configuration, exterior, power, and price. We compare the extracted query features to the feature set of current document to give extra credits to the document if it satisfies the conditions. For seven logic values on configurations and two numerical values on power, we give 0.1 extra score if the current document conforms to query features, while for features like car make, drivetrain, exterior colorand price, we reward the document 0.4 extra points.

To judge whether our features from neural network are consistant to the ones in current document, we divide the work into two situations: simple features (exterior color, car make, etc. that can be represented by simple and accurate words) and extent features (power, configurations that comes with various adjectives before them).

For simple features, we simply match them through finding the intersection between current document's feature set and certain key-value pairs in the output of neural network. If there exists intersection, we will give additional score to the document due to its relevance to the query in terms of feature. For extent features, we roughly define several mappings from common vocabularies/numerical values to three levels of extent: cheap/low, moderate/average, and expensive/high. We first match keywords in the output of neural network to classify them into three categories, then we manually define threshold for numerical values so as to judge whether current document's feature falls into the same category. Finally we can decide whether to give extra score to the document based on the classification result for extent features.

4.5 Interface

For deployment, we designed an interface using the package Streamlit. Users can use this web interface like using an ordinary search engine. In addition to this, two main features are included in this interface.

4.5.1 Development Mode

To boost the algorithm tuning process, a development mode is incorporated in the interface. The development mode provides batch query, result annotation and other feature to help the development process.

4.5.2 Caching Techniques

To maintain performance of the backend, the main objects in the interface, including the FeatureExtractor, the FeatureClassifier and the Ranker, all have a two-level cache. The first level of cache is the object itself, using an experimental primitive decorator of Streamlit. In the entire lifecycle of a Streamlit app, these three objects are shared across different sessions. Hence only the first access to the object requires to init the object, which greatly speeds up the load time of subsequent accesses.

The second level of cache is the data interaction with the objects. Consider every stage t in the presented pipeline architecture of Figure 1 as $O_t = p_t(I_t)$ where O_t is the output, I_t the input and p_t the pipeline stage. For the time-consuming interactions such as feature classification, a PipelineCache object is prepared to store all existent (I_t, O_t) pairs. If I_t exists in the cache, it will not be processed by p_t again, but instead the corresponding O_t in the cache will be directly returned. This design has two benefits:

1. After a time of usage, the system will become faster and faster.

2. The caches provides an interface to directly apply feedbacks. If the prediction of Feature-Classifier went awry for a certain I_f, we can just modify the (I_f, O_f) pair in the cache. This interface is also implemented using Streamlit, and can be access in the development menu.

5 Evaluation and Results

5.1 Baseline generation

For evaluation, we used Pyserini to index and run a BM25 base line first. The features we extracted are concated into a single string as the document. The querys are searched on this document, we show a part of the results here:

```
("..." represents features omitted here due to page limits.)
1 - I want a red car with heated seats.
    > doc 3064 - 2008 Chevrolet Corvette Indy Pace Car, 19995, Red, ...
    > doc 1821 - 2011 Dodge Ram 1500 20 WHEELS SEATS PKG, 17985, Gray, ...
    > doc 3085 - 2004 Lincoln Town Car Ultimate L, 7995, Gold, ...
2 - I want a blue car with CarPlay.
    > doc 3064 - 2008 Chevrolet Corvette Indy Pace Car, 19995, Red, ...
    > doc 3085 - 2004 Lincoln Town Car Ultimate L, 7995, Gold, ...
    > doc 3102 - 2008 BMW 128 i, 8900, Blue, ...
```

To evaluate the effectiveness of this baseline, the queried results are manually marked as relevant/irrelavant. The web interface has incorporated such feature. Then, we calculate the MAP of the baseline. We use the same procedure for the enhanced model too. See Figure 7 for the performance of the baseline. The evaluation shows that for some queries BM25 performs well, but it doesn't perform well on most of the queries. For the queries do work well, we also suspect the effectiveness as they actually reflect some bias in our data. For example, query 14 asks that "I want to keep warm in winter and start the car remotely". Since Cars.com requires the buyer's location, even if we've set the diameter as infinite, the crawled data is still scattered in a large circle around Ann Arbor. And most cars in Michigan do have heated devices since winters in Michigan are cold.

5.2 Comparison between baseline and enhanced model

With the help of our web-based experiment program, we run the 20 sample queries again using the enhanced ranking function, and manually annoate the top 5 retrieved results for each query as relevant or not. We use MAP as the evaluation metric to test the performance of our new model against the baseline. The line chart below clearly shows the difference of MAP between each query.

The avaerge MAP for baseline is 0.24 while our enhanced model delivers a 0.47 average MAP. From the graph we can see in over half of the queries, the enhanced model performs better than the baseline. By checking the content of queries, we find our proposed model has high accuracy in matching the color, power, price, and drivetrain. However, when user specify a car make name in the query, our system cannot recognize the make information and send back filtered results. Also, if the query makes a detailed requirement on one specific configuration, for example heated seats, our system perfroms bad because we are grouping configurations into three categories instead of evaluating on each configuration.

We consider hyperparameters of the ranking function as a great influence on the performance of model, which mainly refers to the extra credit added to the document for satisfying the query

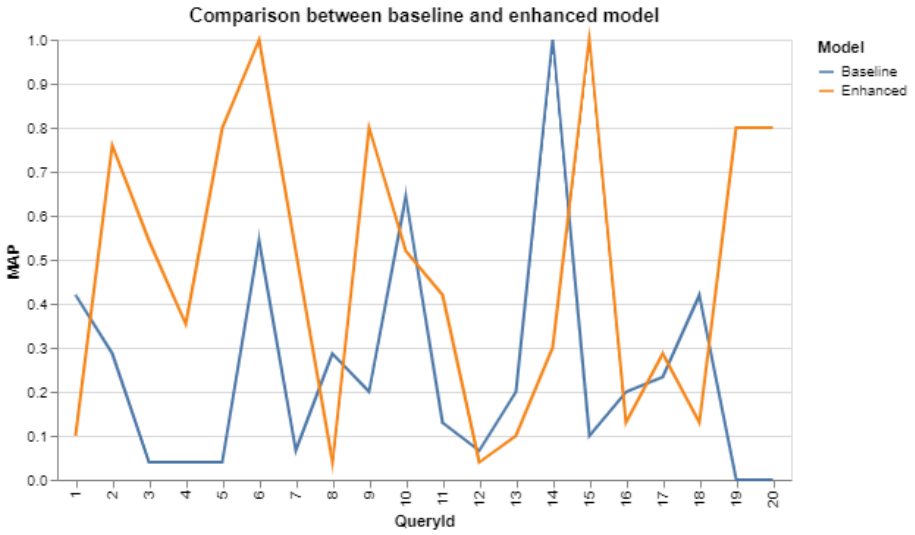

Figure 7: MAP comparison of baseline and enhanced model

requirements. In this model, we use a small weight for configuration and large weight for price, color, drivetrain and power, which explains why we have high accuracy in terms of these four aspects. Another limit on performance is the neural network, even if we apply rules for filtering car makes at the ranking step, our effort may be in vain if the network model cannot extract such feature from the query. One reason for unsuccessful feature extraction is the flexibility of the query, which means if we have several attributes before the car keyword, the position of the make may influence its chance of being detected. The other reason is that the same property of a car can take different forms of expressions, with the limited size of training set, we cannot cover all the cases.

6 Discussion

From the experimental results, we find our model performs fair in most queries while still have zero accuracy in certain queries. Therefore, we don' t think the performance is satisfactory enough for an end-user. Since we are doing a fuzzy search, the variation of queries will be dramatic in real world situations because each user has his/her own way of expression. In our model, we only use twenty sample queries to make annotations on a small set of documents, and the sample queries all follow the simple structure of "I want..." so that the simple neural network model can easily figure out distinct features. In reality, user may speak out their car needs in a story-telling form, like "I am a housewife finding an SUV to pick up my children from school". We think the idea of fuzzy searching as a future trend for smarter search engines, but our effort is not enough to do useful science due to the limitation of feature detection in highly flexible queries.

Our approach makes a 92% improvement (0.47) in terms of average MAP across 20 example queries against the baseline (0.24), which is out of our original expectation. Though we are well aware that pure BM25 is not useful in dealing with numerical features as well as fuzzy queries, we did not expect much on our improved model because we realized the difficulty to extract features from vague expression is great halfway through the project. Our model does well in extracting salient features like drivetrain, exterior color, and make because they are often single attribute words before the keyword "car", we use regular expression to filter out these features and set as input of our neural network model. Our solution also does well in selecting the proper price and

determining the scale of feature (high/low power, poor/rich configuration), which is achieved by the extent word bank created by our own. We figure out a set of adjectives people always use in daily life and use them to map a wide range of query terms to three categories, which yields a good result for most cases.

However, the weakness of our solution is magnified if the sequence of words in the query is changed, or rare words and numerical values are included, or a more narrative tone is used together with too much details. For example, the retrieved results demonstrate high randomness for the query "I am a student searching for a light-color sedan capable of driving in snowy days". Our model is still at the level of understanding word-level meanings but not the implications behind words. Moreover, we are mostly focusing on a few basic queries and make a small number of annotations to support it, which is far from constructing a huge dataset for training a robust model.

7 Conclusion

In this project, we propose the idea to make a search engine to provide support for fuzzy queries, so that nonprofessionals can be exposed to good deals even though they are not familiar with car terminologies. We scrap data from Cars.com and do some basic data cleaning to make the features of each car tidy and structural. In order to better understand the keypoints in users' descriptions, we use some natural language processing techniques to recognize components and dependencies of a sentence, and train a small neural network based on RNN to transform the original query into key-value pairs indicating the content and classification of the mentioned features. Our ranking function combines traditional BM25 score and a new enhancement score, which is a set of rules we define to match our processed queries with features in the original documents. The experimental results show that our system has an improvement on fuzzy searching than the simple BM25 ranker, and the improvement is greater on well-structured queries with right vocabularies. The neural network model we train can identify salient features from unprofessional descriptions of car needs, while it still makes mistakes on colloquial, detailed, special-to-usecase expressions. At the same time, the calculation of enhancement score in the ranker is too limited to a small subset of vocabulary to deal with complex features described in human language.

Though we are not making great progress in improving the quality of fuzzy search, we still think it is a promising topic to provide more customized service and optimize search engines in substantial domains. When searching for domain-specific information, fuzzy search conveys the idea of an equal treatment of all user types regardless of their knowledge background. We think NLP techniques are essential to a better interpretation of user descriptions, maybe future researches can apply topic modeling, summarization or neural network based language models to extract features as much as possible.

Here is the link to our project source code.

8 Acknowledgements

Our thanks to Darwin Marsh, this project wouldn't be proposed were it not for the Ford that he sold to us. It was a perfect car for us and we had great times.

References

Edwin Zhang, Nikhil Gupta, Rodrigo Nogueira, Kyunghyun Cho, and Jimmy Lin. Rapidly deploying a neural search engine for the COVID-19 open research dataset: Preliminary thoughts and

lessons learned. *CoRR*, abs/2004.05125, 2020. URL https://arxiv.org/abs/2004.05125.

Luke S. Murray, Divya Gopinath, Monica Agrawal, Steven Horng, David A. Sontag, and David R. Karger. Medknowts: Unified documentation and information retrieval for electronic health records. *CoRR*, abs/2109.11451, 2021. URL https://arxiv.org/abs/2109.11451.

Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Document ranking with a pretrained sequence-to-sequence model. *CoRR*, abs/2003.06713, 2020. URL https://arxiv.org/abs/2003.06713.

Abhay Prakash and Dhaval Patel. Techniques for deep query understanding. *CoRR*, abs/1505.05187, 2015. URL http://arxiv.org/abs/1505.05187.

Shichen Liu, Fei Xiao, Wenwu Ou, and Luo Si. Cascade ranking for operational e-commerce search. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2017. doi: 10.1145/3097983.3098011. URL http://dx.doi.org/10.1145/3097983.3098011.

A Ineffective Methods

We've initially tried with some methods to translate the query into standarized forms. For instance, our system doesn't recognize inputs such as "I want a car with good performance in cold weathers". We want the system to be able to translate that into queries like "A car with strong heating system, heated steerwheels, gasoline-driven, four-wheel-drive". We've tried two methods:

A.1 Seq2seq Model

A seq2seq (sequence to sequence) model is a special kind of RNN model that excels at complex language tasks, such as translation, question answering and text summarization. We initially tried to extract some data from the reddit corpus to train a seq2seq model. However, the content in the reddit corpus has too many distractions, and require plenty of works to be clensed.

A.2 Closest Words in Corpus

A substitution of seq2seq model was also proposed. In this method, we tried a package called sense2vec which helps at finding the most similar vectors for multi-word phrases in the given corpus, based on POS tags and other labels. We also implemented another snippet using spaCy's own `most_similar` method. Both performed poorly, because the effectiveness entirely depends on whether the corpus have the queried phrase or not. For instance, the reddit corpus does not have the phrase "strong horsepower", so both methods entirely fail on this query. Additionally, the latter method is extremely slow since it must iterate over the entire corpus. We therefore turned to the RNN, which only require the corpus to have two-words phrase, and is guranteed to have almost constant time complexity since it is not dependent on the size of the corpus.

B Plan & Retrospective

B.1 Future Works

B.1.1 Algorithm Performance Improvements

1. The feature extraction module can have more parallel rules to capture more kinds of two-word, even three-word phrases.

2. The feature classification network should be retrained with better data and a valid test set to prevent overfitting. Given enough time and human resource, a manually annotated dataset can greatly improve the performance of the RNN.

3. A seq2seq network can be trained to translate more customized inputs to standarized descriptions. We didn't implement it at last because the train data is harder to obtain, but with enough time it is doable.

B.1.2 System Performance Improvemnts

1. The first level cache hasn't been tested to be thread-safe. When accessed by multiple users at the same time, problems may occur. This needs further testing.

2. Each uncached query takes around 5 seconds on our dataset. This can further be improved by pre-calculated cache and multiprocessing.

B.1.3 Other Works

1. Provide a minimal Docker image for fast deployment.

2. Remove Pyserini dependency. Since Pyserini requires JAVA, the deployment thus become complex. Replacing Pyserini with Python code or customed binaries will solve this. Since we don't depend heavily on Pyserini, this is achievable.

B.2 Retrospective

1. The pre-process of queries is actually the hardest part in this project. We initially planned to use around two weeks on the NLP input interpretation task, and two weeks on algorithm finetuing. It would be better if we could initially balance between these two tasks.

2. Looking at a retrospective, though collecting data costs plenty of time, we should have started the algorithm part earlier, even if the data was incomplete. We didn't realize that we would require plenty of corpus data in the process of developing the algorithm, and it would be better if we could have rescheduled the timeline in mid October.