|  |
| --- |
| **Algorithms for Information Retrieval and Intelligence Web** |

**ASSIGNMENT 1**

**Zomato Restaurant IR System**

**Team Members**

**Anushka Jalori PES1UG20CS071**

**Ayushi Soumya PES1UG20CS097**

**Gaurav Mahajan PES1UG20CS150**

**Dataset used: ratings.csv**

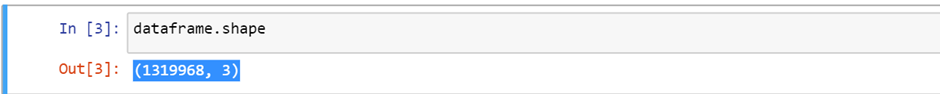
**Overview of the database**

The database has 3 columns

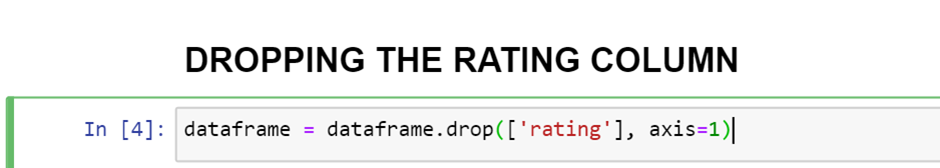
* Name:- Corresponds to the name of the restaurant.
* Rating:- Has the rating out of a scale of 5.
* Review:- has the review for the particular restaurant.

**PART 1: Pre-processing of raw data**

Initial size of database: (1319968, 3)

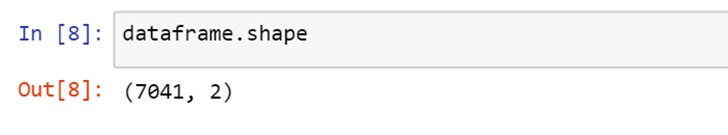


Dropped the rating column since we are only dealing with textual data in our IR system.



Treat all different branches of different restaurants as the same entity by combining the reviews and treating each restaurant as a separate document.

This drastically reduces the size of the database.



**Preprocessing reviews column**

The following are the steps we have taken:

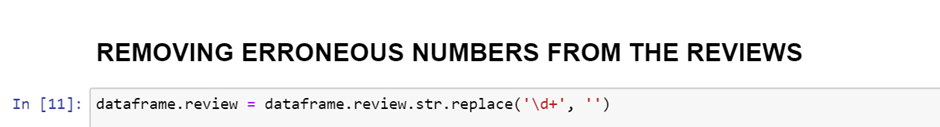
1. Making all strings lowercase

This may lead to some words losing their meaning, however, we move forward

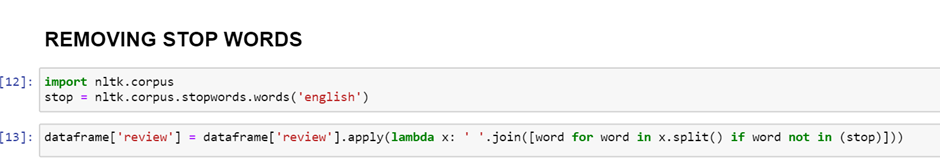
with the assumption that not many proper nouns are used to a limited extent.



1. Removing erroneous numbers from the reviews section as they do not contribute much to the meaning.



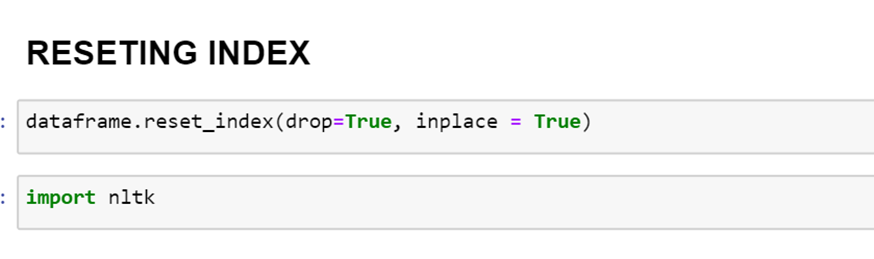
1. Removing stopwords from the dataset.



1. Upon closer inspection, we realised that there were many words that were slang and not a part of the English dictionary. For that reason we decided to only keep those words that are present in the dictionary



1. Deleting rows lead to improper indexes. We reset the indexes so we can later treat the indexes as the document id.



1. Lemmatizing with POS index

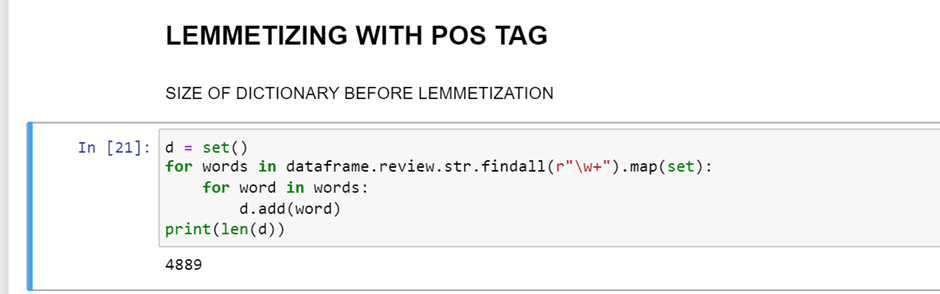
**from nltk.stem import WordNetLemmatizer**

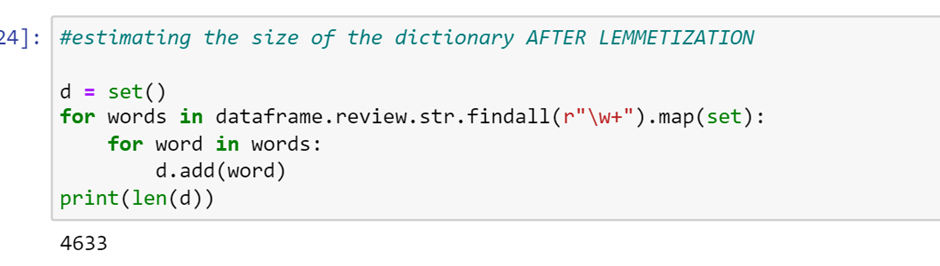
Initially we tried lemmatizing without the pos index, however, the size of the

dictionary did not reduce as much as we expected and was still large.

So, our next step was to lemmatize using POS indexes which mark the words as

adjectives, nouns etc and lemmatize accordingly.



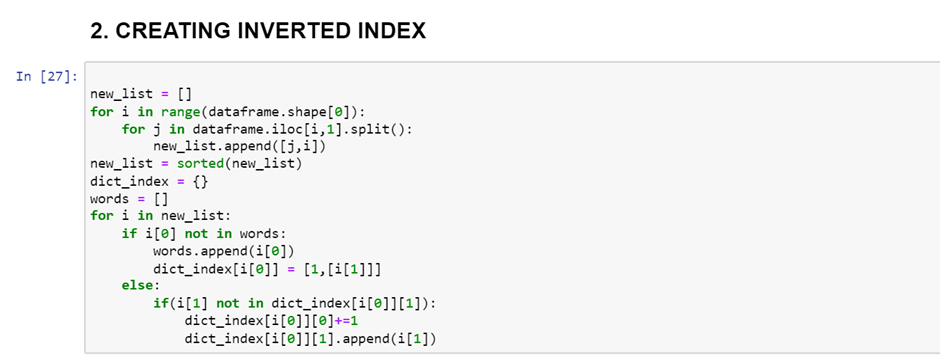


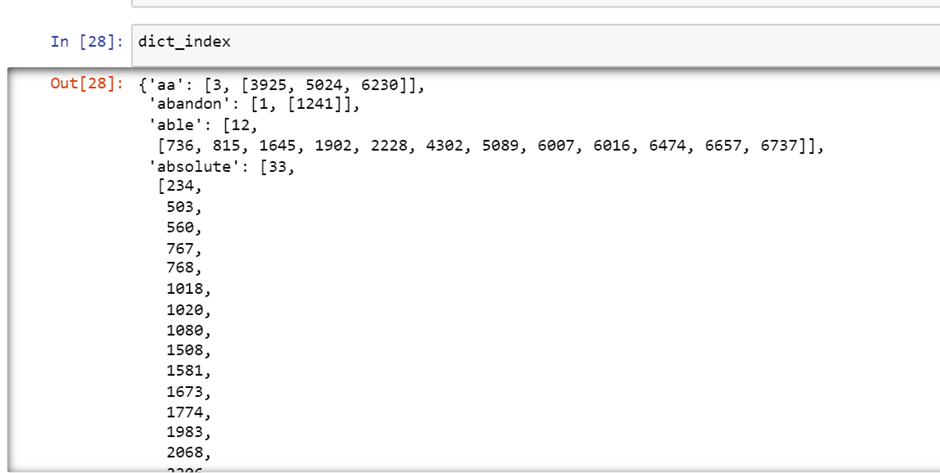
**PART 2: Generate Inverted Index (variation in data structures)**

We tested out many different data structures ranging from lists, dictionaries, nested data structures etc.

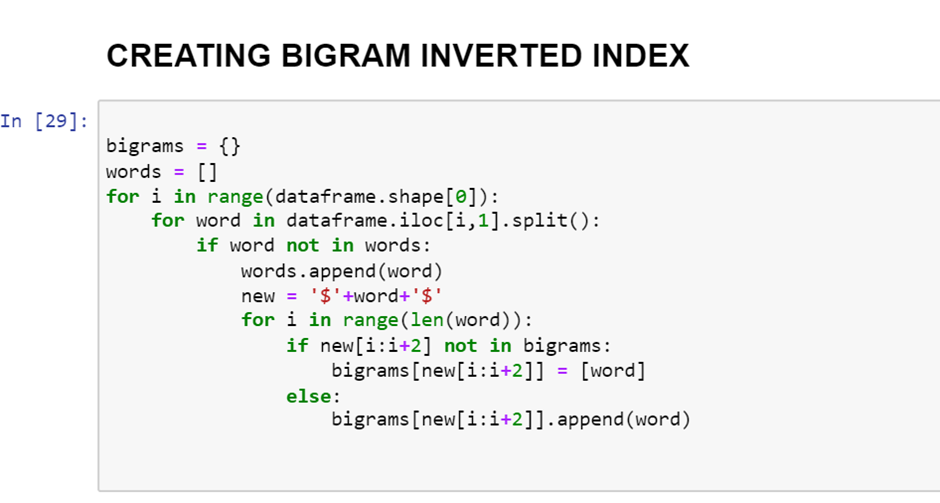
Here is an overview of all the structures used:-

1. **Dictionary with key as word and values as [doc\_frequency, [list of doc ids]]**
   1. Though a reliable data structure, it has no positional information and can be improved in that aspect.





1. **Created a bigram inverted index** 
   1. Data Structure used: key is bigram, value is list of all possible words corresponding with that bigram





1. **Inverted index with positional information**
2. Format of the data structure:

Each dictionary term is of the following format

**'abandon': [3, {3925: [5, 21, 37, 53, 69, 85, 101, 117, 133, 149], 5024: [1, 28], 6230: [4]}]**

Key: Dictionary term

Value: [first term id doc\_frequenct followed by a dictionary where key is doc\_id and value is the positional information of the word in the document]

**PART 3: Handling wild card and phrase queries**

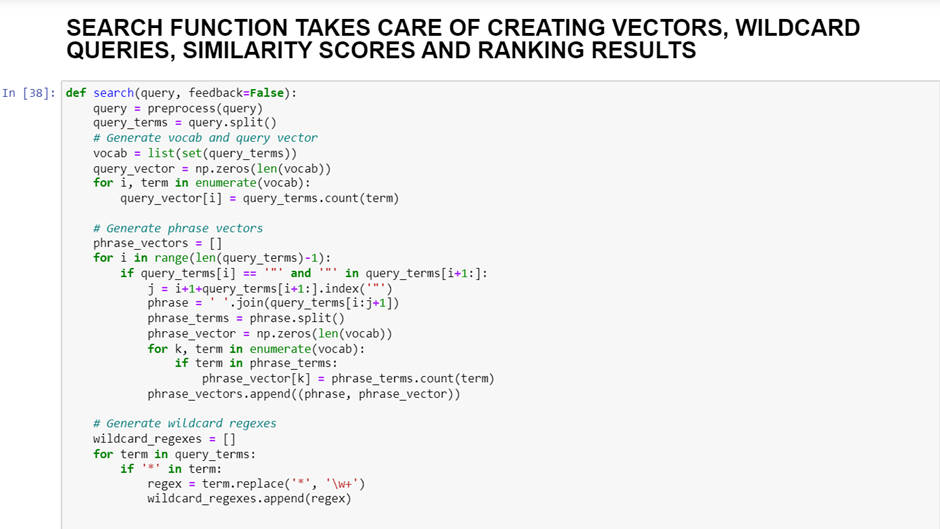
Here, we have written a super-function called search which takes two parameters :

1. Query - a compulsory parameter that handles the queries of the end-user
2. Feedback - an optional parameter with a default value of false, when set to true may ask for user feedback.

In the first step, we split the query into a set of query terms and create a set of unique terms in the query and store it in the vocab list. Then , we go on to vectorize these terms and use a loop to iterate over each term in the vocabulary list and count the number of occurrences of that term in the query. This count is then stored in the corresponding index of the query vector.

Next step, we take up handling of phrase queries. This code extracts phrases from the search query, generates phrase vectors that represent the frequency of each term in the phrase, and stores them in a list along with the original phrase, similar to the query vector generated in the previous code snippet.

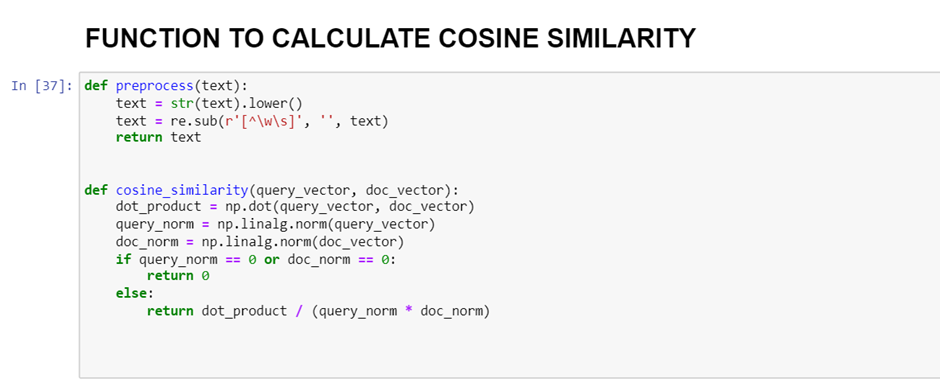
We create an empty list named "wildcard\_regexes" to store the regular expressions. The loop iterates over each term in the query and checks if it contains an asterisk using the "in" keyword. If the term contains an asterisk, the code replaces the asterisk with the regular expression pattern "\w+", which matches one or more alphanumeric characters. This creates a regular expression that can match any word that contains the original search term as a substring. The code then appends the generated regular expression to the "wildcard\_regexes" list.

****

**PART 4: Retrieve relevant text using similarity index**

We define a function that performs two preprocessing steps on a string of text: converting it to lowercase and removing any non-alphanumeric characters.

We also define a function that calculates the cosine similarity between two vectors using their dot product and Euclidean norms.

****

We then compute similarity scores between the query vector and each document vector in a dataframe by counting the number of times each term in the query appears in each document, and calculating the cosine similarity between the resulting query and document vectors. The code also checks for phrase queries and incorporates their matching scores in the final similarity score.

We check for wildcard queries in the query terms, and if any are present, then check whether the document contains any terms that match the wildcard pattern. If a match is found, the code calculates the cosine similarity score between the query vector and the document vector, and appends the index of the document and its score to a list of scores.

****

**PART 5: Retrieve relevant text using likelihood language model**

There are many problems with a straightforward probabilistic language model that computes n-gram probabilities. The context issue is the main one. The following word in complex sentences is influenced by the context that comes before it.

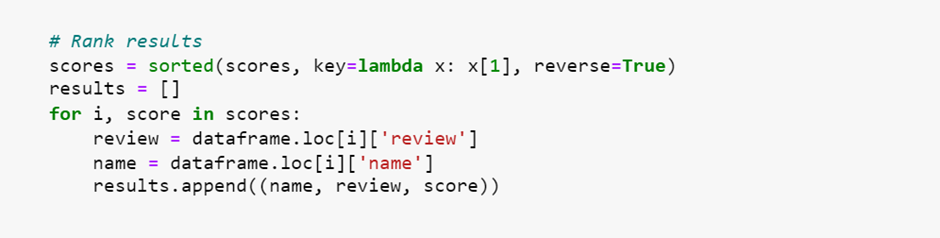
Therefore, even if the value of n is very high, it could not be clear from the previous n words what the next word would be.

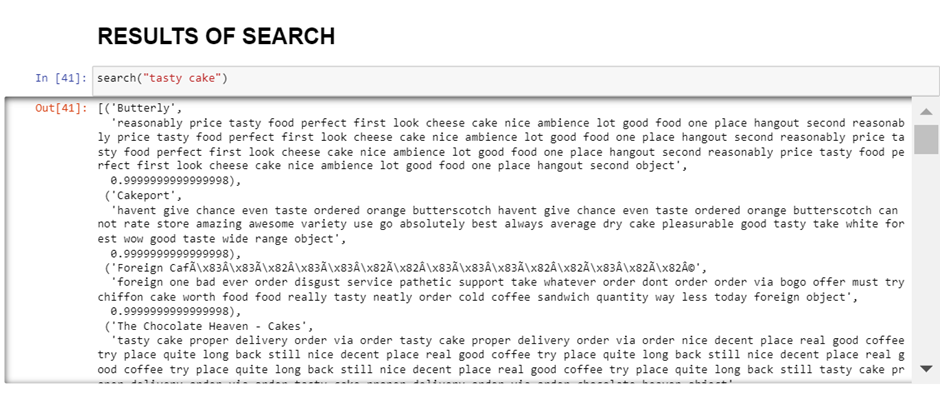
Scale and resource issues are another issue with this strategy.

Even if most permutations never appear in the text, the number of feasible permutations rises sharply as the size (n) increases. Unrepeatable n-grams cause a sparsity issue. Most words have the same probability since the probability distribution's granularity might be quite low.

**PART 6: Ranking of retrieved documents**

Now, we rank the results based on the similarity scores, from highest to lowest, and generate a list of tuples that include the restaurant name, review, and score for each document that matched the query.

****

****

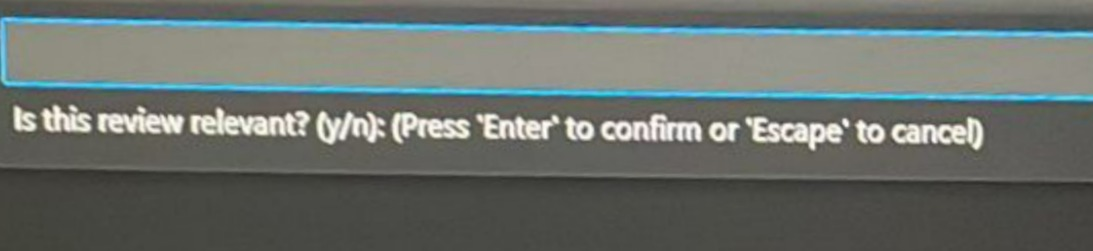
**PART 7: Advanced search: relevance feedback, semantic matching, re-ranking of results, finding out query intention**

**RELEVANCE FEEDBACK AND RE-RANKING**

We sort the scores in descending order and return a list of tuples representing the search results. If the feedback parameter is True, the function prompts the user to indicate whether each search result is relevant or not, computes the mean similarity scores for relevant and non-relevant reviews, and re-ranks the search results using a modified similarity score that incorporates these mean scores. The function then returns the re-ranked search results.

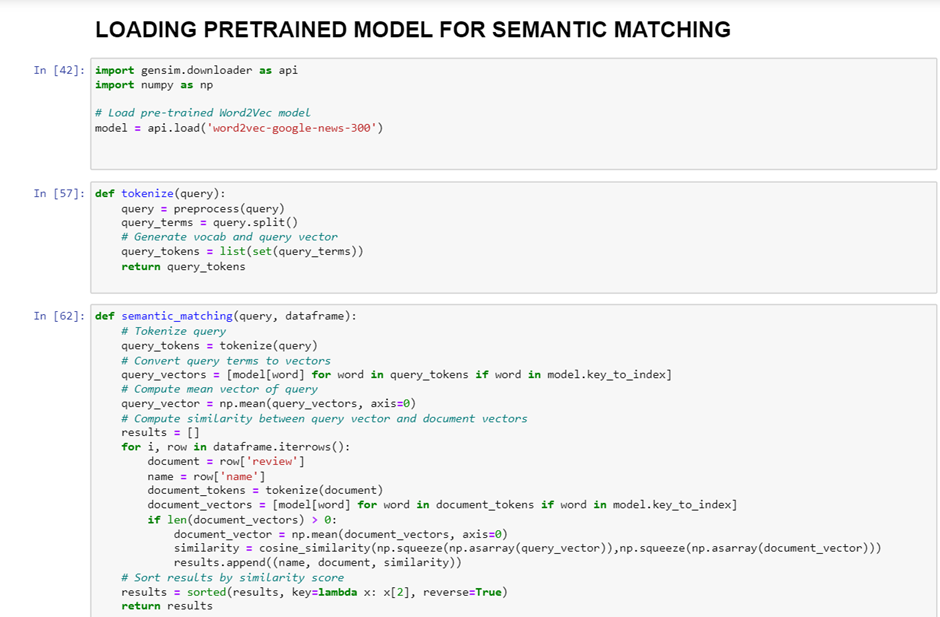
****

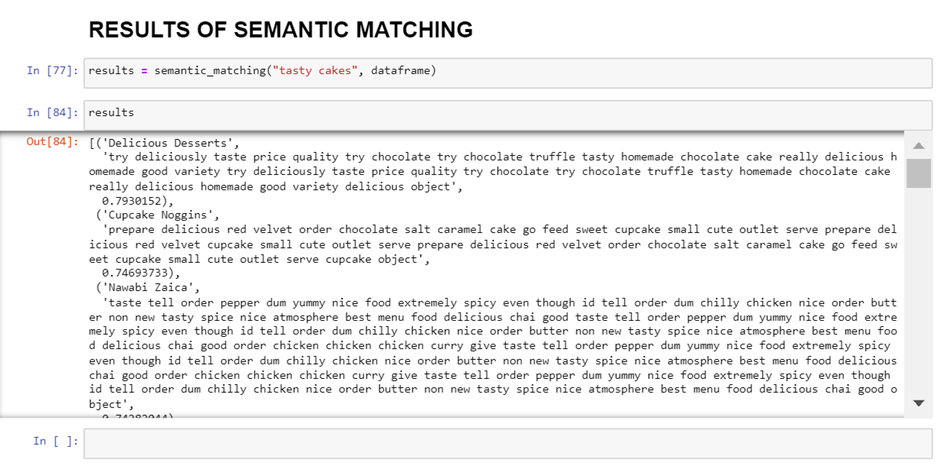
When the parameter gets set to True, this dialog box appears accepting the user feedback which is given as **y** if the results are relevant and **n** if it is not.

****

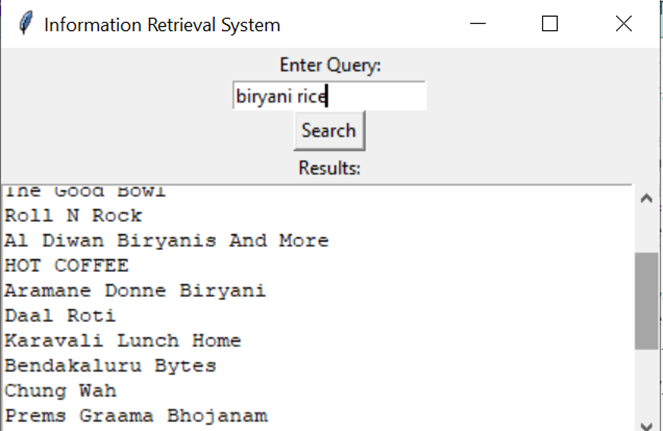
**SEMANTIC MATCHING**

We perform semantic matching using word embeddings to find relevant documents based on a query. We first tokenize the query, then convert the query terms to vectors using a pre-trained word embedding model. We then compute the mean vector of the query and compare it with the mean vectors of the documents to compute the cosine similarity. Finally, we return the top documents sorted by similarity score.

****

****

**SIMPLE GUI**

****

**Results**

Our information retrieval, thus, can take care of a variety of queries ranging from Index generation, wild card query implementation details, phrase queries implementation details, ranking of retrieved results, semantic matching using pre trained models, take in user feedback, a simple GUI etc.

**Conclusion**

The user can thus type in any query in mind and get restaurants that correspond to his/her needs. Our system will provide results based on any type of query provided by the user on a simple GUI interface.