

Algorithms for Information Retrieval and Intelligence Web

UE21CS342BA2

AIRIW PROJECT

Bengaluru Restaurant Choice Assistant

Under the guidance of

Dr. Alpha Vijayan

Team Members

AAKAASH KURUNTH PES2UG21CS004

TEJAS B PES2UG21CS926

DATASET

- **zomato.csv**

All the column names and their meanings :

URL: contains the URL of the restaurant on the zomato website

address=contains the address of the restaurant in Bengaluru

name: contains the name of the restaurant

online_order=whether online ordering is available in the restaurant or not

book_table=table book option available or not

rate=contains the overall rating of the restaurant out of 5

votes=contains total number of rating for the restaurant as of the above-mentioned date

phone=contains the phone number of the restaurant

location=contains the neighbourhood in which the restaurant is located

rest_type=restaurant type

dish_liked=dishes people liked in the restaurant

cuisines=food styles, separated by a comma

approx_cost(for two people)=contains the approximate cost for a meal for two people

reviews_list=list of tuples containing reviews for the restaurant, each tuple consists of two values

BRIEF DESCRIPTION

The Restaurant Choice Assistant is a user-friendly web application designed to simplify the process of selecting a restaurant. Users can input food-related terms or criteria (such as “friendly atmosphere” or specific cuisines) into the search bar. The system intelligently retrieves restaurant information based on the user’s query and displays relevant options. The clean and straightforward user interface emphasizes text elements and search functionality, making it easy for users to discover dining options that align with their preferences.

FUNCTIONALITIES

The Restaurant Choice Assistant offers several key functionalities to enhance the user experience:

- **Search Bar and Query Input:**

Users can enter food-related terms, cuisine preferences, or any other relevant criteria into the search bar.

The system intelligently processes these queries to retrieve relevant restaurant information.

- **Dynamic Results Display:**

Upon clicking the “Go” button, the application displays a list of matching restaurants.

Each restaurant name is presented as a result, allowing users to quickly scan through options.

- **Intelligent Filtering and Sorting:**

The application may utilize AIRIW concepts such as lemmatisation, inverted indexes, phrase and wildcard queries, semantic matching, relevance feedback etc.

It ensures that the displayed options align with the user’s preferences or requirements.

- **User-Friendly Interface:**

The UI design is clean and straightforward, emphasizing text elements.

Users can easily interact with the search functionality without distractions.

In summary, the Restaurant Choice Assistant streamlines the restaurant selection process, making it convenient for users to discover dining options tailored to their liking.

TECHNOLOGIES USED

Front end: Tkinter

Database: CSV

Libraries used:

- Pandas
- NumPy
- NLTK
- Scikit-learn
- sklearn.metrics.pairwise
- Gensim
- Tkinter

CODE SNIPPET AND DETAILED EXPLANATION

PART 1: Pre-processing of raw data

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

DROPPING THE RATING COLUMN

```
In [4]: dataframe = dataframe.drop(['rating'], axis=1)
```

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.

MAKING ALL STRINGS LOWER CASE

```
In [10]: dataframe['review'] = dataframe['review'].str.lower()
```

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

REMOVING ERRONEOUS NUMBERS FROM THE REVIEWS

```
In [11]: dataframe.review = dataframe.review.str.replace('\d+', '')
```

3. Removing stopwords from the dataset.

REMOVING STOP WORDS

```
[12]: import nltk.corpus
stop = nltk.corpus.stopwords.words('english')

[13]: dataframe['review'] = dataframe['review'].apply(lambda x: ' '.join([word for word in x.split() if word not in (stop)]))
```

4. 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

```
In [14]: words = set(nltk.corpus.words.words())

In [15]: dataframe['review'] = dataframe['review'].apply(lambda x: ' '.join([word for word in x.split() if word in (words)]))

In [16]: dataframe

Out[16]:
```

	name	review
0	Jalsa	beautiful place dine dinner family restaurant ...
12	Spice Elephant	dinner family turned ambience really nice staf...
26	San Churro Cafe	ambience good enough went quick bite first big...
46	Addhuri Udupi Bhojana	great food proper style full place half good f...
81	Grand Village	good restaurant buffet great service overwhelm...
...
1315206	Calcutta North Indian Meals	center probably famous north object
1315268	Chime - Sheraton Grand Bengaluru Whitefield Ho...	nice friendly place staff awesome service bad ...
1315289	The Nest - The Den Bengaluru	great ambience looking nice good selection nes...
1315306	Nawabs Empire	place good ordered negative review would object
1315570	SeeYa Restaurant	good food take bit time get food coz ordered c...

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

RESETING INDEX

```
: dataframe.reset_index(drop=True, inplace = True)

: import nltk
```

6. 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.

LEMMETIZING WITH POS TAG

SIZE OF DICTIONARY BEFORE LEMMETIZATION

```
In [21]: d = set()
for words in dataframe.review.str.findall(r"\w+").map(set):
    for word in words:
        d.add(word)
print(len(d))

4889
```

24]: *#estimating the size of the dictionary AFTER LEMMETIZATION*

```
d = set()
for words in dataframe.review.str.findall(r"\w+").map(set):
    for word in words:
        d.add(word)
print(len(d))

4633
```

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]]

- Though a reliable data structure, it has no positional information and can be improved in that aspect.

2. CREATING INVERTED INDEX

```
In [27]: new_list = []
for i in range(dataframe.shape[0]):
    for j in dataframe.iloc[i,1].split():
        new_list.append([j,i])
new_list = sorted(new_list)
dict_index = {}
words = []
for i in new_list:
    if i[0] not in words:
        words.append(i[0])
        dict_index[i[0]] = [1,[i[1]]]
    else:
        if i[1] not in dict_index[i[0]][1]:
            dict_index[i[0]][0] += 1
            dict_index[i[0]][1].append(i[1])
```



```
In [28]: dict_index
```

```
Out[28]: {'aa': [3, [3925, 5024, 6230]],  
          'abandon': [1, [1241]],  
          'able': [12,  
                  [736, 815, 1645, 1902, 2228, 4302, 5089, 6007, 6016, 6474, 6657, 6737]],  
          'absolute': [33,  
                       [234,  
                        503,  
                        560,  
                        767,  
                        768,  
                        1018,  
                        1020,  
                        1080,  
                        1508,  
                        1581,  
                        1673,  
                        1774,  
                        1983,  
                        2068,  
                        2225,
```

2. Created a bigram inverted index

- Data Structure used: key is bigram, value is list of all possible words corresponding with that bigram

CREATING BIGRAM INVERTED INDEX

```
In [29]: bigrams = {}  
words = []  
for i in range(dataframe.shape[0]):  
    for word in dataframe.iloc[i,1].split():  
        if word not in words:  
            words.append(word)  
            new = '$'+word+'$'  
            for i in range(len(word)):  
                if new[i:i+2] not in bigrams:  
                    bigrams[new[i:i+2]] = [word]  
            else:  
                bigrams[new[i:i+2]].append(word)
```

```
In [30]: bigrams
```

```
Out[30]: {'$b': ['beautiful',  
                'best',  
                'bad',  
                'bite',  
                'big',  
                'buffet',  
                'bar',  
                'back',  
                'bit',  
                'busy',  
                'book',  
                'bunch',  
                'barbecue',  
                'break',  
                'base',  
                'bath',  
                'branch',  
                'board',  
                'become',  
                'beak']
```

3. Inverted index with positional information

- a. 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.

SEARCH FUNCTION TAKES CARE OF CREATING VECTORS, WILDCARD QUERIES, SIMILARITY SCORES AND RANKING RESULTS

```
In [38]: def search(query, feedback=False):
        query = preprocess(query)
        query_terms = query.split()
        # Generate vocab and query vector
        vocab = list(set(query_terms))
        query_vector = np.zeros(len(vocab))
        for i, term in enumerate(vocab):
            query_vector[i] = query_terms.count(term)

        # Generate phrase vectors
        phrase_vectors = []
        for i in range(len(query_terms)-1):
            if query_terms[i] == '*' and '*' in query_terms[i+1:]:
                j = i+1+query_terms[i+1:].index('*')
                phrase = ' '.join(query_terms[i:j+1])
                phrase_terms = phrase.split()
                phrase_vector = np.zeros(len(vocab))
                for k, term in enumerate(vocab):
                    if term in phrase_terms:
                        phrase_vector[k] = phrase_terms.count(term)
                phrase_vectors.append((phrase, phrase_vector))

        # Generate wildcard regexes
        wildcard_regexes = []
        for term in query_terms:
            if '*' in term:
                regex = term.replace('*', '\w+')
                wildcard_regexes.append(regex)
```

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.

FUNCTION TO CALCULATE COSINE SIMILARITY

```
In [37]: def preprocess(text):
text = str(text).lower()
text = re.sub(r'^\w\s', '', text)
return text

def cosine_similarity(query_vector, doc_vector):
dot_product = np.dot(query_vector, doc_vector)
query_norm = np.linalg.norm(query_vector)
doc_norm = np.linalg.norm(doc_vector)
if query_norm == 0 or doc_norm == 0:
return 0
else:
return dot_product / (query_norm * doc_norm)
```

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.

```
# Compute similarity scores
scores = []
for i, review in enumerate(dataframe['review']):#####
    terms = review.split()
    doc_vector = np.zeros(len(vocab))
    for j, term in enumerate(vocab):
        doc_vector[j] = terms.count(term)
    # Check phrase queries
    phrase_match = True
    for phrase, phrase_vector in phrase_vectors:
        if phrase not in review:
            phrase_match = False
            break
        phrase_score = cosine_similarity(phrase_vector, doc_vector)
        phrase_match = phrase_match and (phrase_score > 0)
    if not phrase_match:
        continue
    # Check wildcard queries
    for regex in wildcard_regexes:
        if not any(re.match(regex, t) for t in terms):
            continue
    score = cosine_similarity(query_vector, doc_vector)
    scores.append((i, score))
```

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.

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.

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.

RESULTS OF SEARCH

```
Out[41]: [('Butterly',
  'reasonably price tasty food perfect first look cheese cake nice ambience lot good food one place hangout second reasonab
  ly price tasty food perfect first look cheese cake nice ambience lot good food one place hangout second reasonably price ta
  sty food perfect first look cheese cake nice ambience lot good food one place hangout second reasonably price tasty food pe
  rfect first look cheese cake nice ambience lot good food one place hangout second object',
  0.9999999999999998),
 ('Cakeport',
  'havent give chance even taste ordered orange butterscotch havent give chance even taste ordered orange butterscotch can
  not rate store amazing awesome variety use go absolutely best always average dry cake pleasurable good tasty take white for
  est wow good taste wide range object',
  0.9999999999999998),
 ('Foreign CaFé\hx83A\hx82A\hx82A\hx83A\hx82A\hx82A\hx83A\hx83A\hx82A\hx82A\hx83A\hx82A\hx82A',
  'foreign one bad ever order disgust service pathetic support take whatever order dont order order via bogo offer must try
  chiffon cake worth food food really tasty neatly order cold coffee sandwich quantity way less today foreign object',
  0.9999999999999998),
 ('The Chocolate Heaven - Cakes',
  'tasty cake proper delivery order via order tasty cake proper delivery order via order nice decent place real good coffee
  try place quite long back still nice decent place real good coffee try place quite long back still nice decent place real g
  ood coffee try place quite long back still nice decent place real good coffee try place quite long back still tasty cake pr
```

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

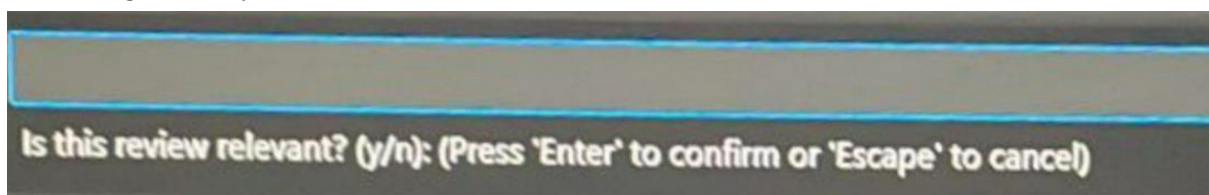
RELEVANCE FEEDBACK AND RE-RANKING

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.

```
# Re-rank results using relevance feedback
if feedback:
    relevant_docs = []
    nonrelevant_docs = []
    for i, (name, review, score) in enumerate(results):
        print(f'Review {i+1}:')
        print(name)
        print(review)

        print(f'Similarity score: {score}')
        feedback = input('Is this review relevant? (y/n): ')
        if feedback.lower() == 'y':
            relevant_docs.append(i)
        else:
            nonrelevant_docs.append(i)
    relevant_scores = [score for i, score in enumerate(scores) if i in relevant_docs]
    nonrelevant_scores = [score for i, score in enumerate(scores) if i in nonrelevant_docs]
    if len(relevant_scores) > 0:
        mean_relevant_score = sum(relevant_scores) / len(relevant_scores)
    else:
        mean_relevant_score = 0
    if len(nonrelevant_scores) > 0:
        mean_nonrelevant_score = sum(nonrelevant_scores) / len(nonrelevant_scores)
    else:
        mean_nonrelevant_score = 0
    alpha = 0.1
    beta = 0.1
    new_scores = []
    for i, (name, review, score) in enumerate(results):
        if i in relevant_docs:
            new_score = (1-alpha)*score + alpha*mean_relevant_score
        elif i in nonrelevant_docs:
            new_score = (1-beta)*score - beta*mean_nonrelevant_score
        else:
            new_score = score
        new_scores.append((i, new_score))
    new_scores = sorted(new_scores, key=lambda x: x[1], reverse=True)
    results = []
    for i, score in new_scores:
        review = dataframe.loc[i]['review']
        name = dataframe.loc[i]['name']
        results.append((name, review, score))
    return 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.

LOADING PRETRAINED MODEL FOR SEMANTIC MATCHING

```
In [42]: import gensim.downloader as api
import numpy as np

# Load pre-trained Word2Vec model
model = api.load('word2vec-google-news-300')
```

```
In [57]: def tokenize(query):
    query = preprocess(query)
    query_terms = query.split()
    # Generate vocab and query vector
    query_tokens = list(set(query_terms))
    return query_tokens
```

```
In [62]: def semantic_matching(query, dataframe):
    # Tokenize query
    query_tokens = tokenize(query)
    # Convert query terms to vectors
    query_vectors = [model[word] for word in query_tokens if word in model.key_to_index]
    # Compute mean vector of query
    query_vector = np.mean(query_vectors, axis=0)
    # Compute similarity between query vector and document vectors
    results = []
    for i, row in dataframe.iterrows():
        document = row['review']
        name = row['name']
        document_tokens = tokenize(document)
        document_vectors = [model[word] for word in document_tokens if word in model.key_to_index]
        if len(document_vectors) > 0:
            document_vector = np.mean(document_vectors, axis=0)
            similarity = cosine_similarity(np.squeeze(np.asarray(query_vector)), np.squeeze(np.asarray(document_vector)))
            results.append((name, document, similarity))
    # Sort results by similarity score
    results = sorted(results, key=lambda x: x[2], reverse=True)
    return results
```

RESULTS OF SEMANTIC MATCHING

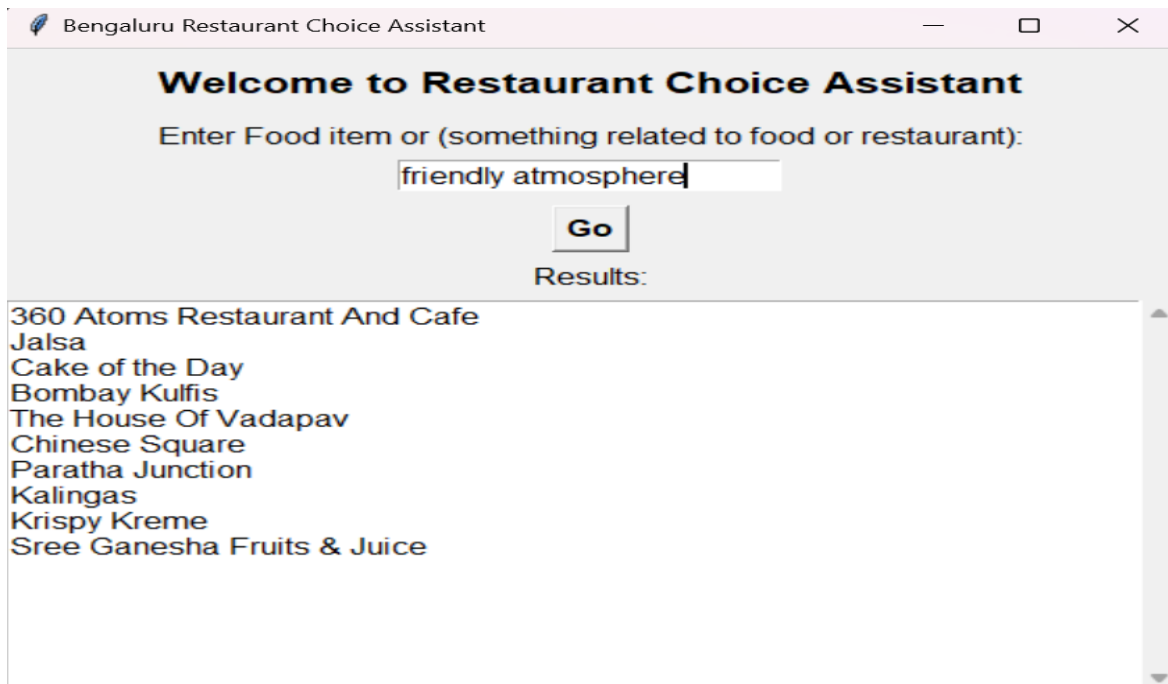
```
In [77]: results = semantic_matching("tasty cakes", dataframe)
```

```
In [84]: results
```

```
Out[84]: [('Delicious Desserts',
      'try deliciously taste price quality try chocolate try chocolate truffle tasty homemade chocolate cake really delicious h
omemade good variety try deliciously taste price quality try chocolate try chocolate truffle tasty homemade chocolate cake
really delicious homemade good variety delicious object',
      0.7930152),
      ('Cupcake Noggins',
      'prepare delicious red velvet order chocolate salt caramel cake go feed sweet cupcake small cute outlet serve prepare del
icious red velvet cupcake small cute outlet serve prepare delicious red velvet order chocolate salt caramel cake go feed sw
eet cupcake small cute outlet serve cupcake object',
      0.74693733),
      ('Nawabi Zaica',
      'taste tell order pepper dum yummy nice food extremely spicy even though id tell order dum chilly chicken nice order butt
er non new tasty spice nice atmosphere best menu food delicious chai good taste tell order pepper dum yummy nice food extre
mely spicy even though id tell order dum chilly chicken nice order butter non new tasty spice nice atmosphere best menu foo
d delicious chai good order chicken chicken chicken curry give taste tell order pepper dum yummy nice food extremely spicy
even though id tell order dum chilly chicken nice order butter non new tasty spice nice atmosphere best menu food delicious
chai good order chicken chicken chicken curry give taste tell order pepper dum yummy nice food extremely spicy even though
id tell order dum chilly chicken nice order butter non new tasty spice nice atmosphere best menu food delicious chai good o
bject',
      0.74022244)]
```

```
In [ ]:
```


Screenshot of the UI and the output of our restaurant choice assistant



Bengaluru Restaurant Choice Assistant

Welcome to Restaurant Choice Assistant

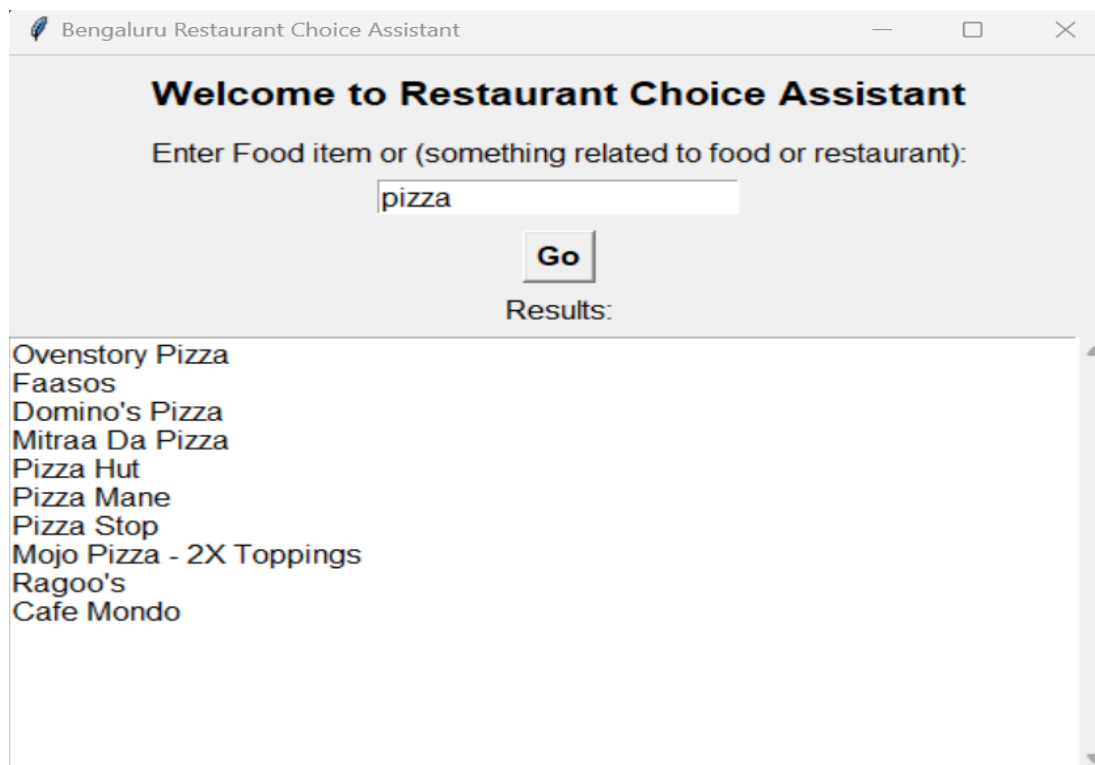
Enter Food item or (something related to food or restaurant):

friendly atmosphere

Go

Results:

- 360 Atoms Restaurant And Cafe
- Jalsa
- Cake of the Day
- Bombay Kulfis
- The House Of Vadapav
- Chinese Square
- Paratha Junction
- Kalingas
- Krispy Kreme
- Sree Ganesha Fruits & Juice



Bengaluru Restaurant Choice Assistant

Welcome to Restaurant Choice Assistant

Enter Food item or (something related to food or restaurant):

pizza

Go

Results:

- Ovenstory Pizza
- Faasos
- Domino's Pizza
- Mitraa Da Pizza
- Pizza Hut
- Pizza Mane
- Pizza Stop
- Mojo Pizza - 2X Toppings
- Ragoo's
- Cafe Mondo

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.