**PROJECT REPORT ON**

**ZOMATO SMART SUGGESTIONS**

**(Master of Science in Computer Science)**

Name: Keerthana Kudumula

ID Num: 1151592

Logo, company name

Description automatically generated

**Content**

I was always fascinated by the food culture of Bengaluru. Restaurants from all over the world can be found here in Bengaluru. From United States to Japan, Russia to Antarctica, you get all type of cuisines here. Delivery, Dine-out, Pubs, Bars, Drinks, Buffet, Desserts you name it and Bengaluru has it. Bengaluru is best place for foodies. The number of restaurant are increasing day by day. Currently which stands at approximately 12,000 restaurants. With such an high number of restaurants. This industry hasn't been saturated yet. And new restaurants are opening every day. However it has become difficult for them to compete with already established restaurants. The key issues that continue to pose a challenge to them include high real estate costs, rising food costs, shortage of quality manpower, fragmented supply chain and over-licensing. This Zomato data aims at analyzing demography of the location. Most importantly it will help new restaurants in deciding their theme, menus, cuisine, cost etc for a particular location. It also aims at finding similarity between neighborhoods of Bengaluru on the basis of food. The dataset also contains reviews for each of the restaurant which will help in finding overall rating for the place.

**What is Recommendation System?**

The rapid growth of data collection has led to a new era of information. Data is being used to create more efficient systems and this is where Recommendation Systems come into play. Recommendation Systems are a type of information filtering systems as they improve the quality of search results and provides items that are more relevant to the search item or are realted to the search history of the user. They are active information filtering systems which personalize the information coming to a user based on his interests, relevance of the information etc. Recommender systems are used widely for recommending movies, articles, restaurants, places to visit, items to buy etc.

There are basically three types of recommender systems:-

* Demographic Filtering- They offer generalized recommendations to every user, based on movie popularity and/or genre. The System recommends the same movies to users with similar demographic features.
* Content Based Filtering- They suggest similar items based on a particular item. This system uses item metadata, such as genre, director, description, actors, etc. for movies, to make these recommendations.
* Collaborative Filtering- This system matches persons with similar interests and provides recommendations based on this matching. Collaborative filters do not require item metadata like its content-based counterparts.

Here I will be using Content Based Filtering

Content-Based Filtering: This method uses only information about the description and attributes of the items users has previously consumed to model user's preferences. In other words, these algorithms try to recommend items that are similar to those that a user liked in the past (or is examining in the present). In particular, various candidate items are compared with items previously rated by the user and the best-matching items are recommended.

This data set consists of restaurants of Bangalore,India collected from Zomato.

My aim is to create a content based recommender system in which when I will write a restaurant name, Recommender system will look at the reviews of other restaurants, and System will recommend us other restaurants with similar reviews and sort them from the highest rated.

**Problem**

The basic idea of analyzing the Zomato dataset is to get a fair idea about the factors affecting the establishment of different types of the restaurant at different places in Bengaluru, aggregate rating of each restaurant, Bengaluru being one such city has more than 12,000 restaurants with restaurants serving dishes from all over the world. With each day new restaurants opening the industry hasn't been saturated yet and the demand is increasing day by day. Inspite of increasing demand it, however, has become difficult for new restaurants to compete with established restaurants. Most of them serving the same food. Bengaluru being an IT capital of India. Most of the people here are dependent mainly on the restaurant food as they don’t have time to cook for themselves.

### What is sentiment analysis?

Sentiment analysis is the computational task of automatically determining what feelings a writer is expressing in text. Sentiment is often framed as a binary distinction (positive vs. negative), but it can also be a more fine-grained, like identifying the specific emotion an author is expressing (like fear, joy or anger).

Sentiment analysis is used for many applications, especially in business intelligence. Some examples of applications for sentiment analysis include:

#### Analyzing the social media discussion around a certain topic

* Evaluating survey responses
* Determining whether product reviews are positive or negative
* Sentiment analysis is not perfect, and as with any automatic analysis of language, you will have errors in your results. It also cannot tell you why a writer is feeling a certain way. However, it can be useful to quickly summarize some qualities of text, especially if you have so much text that a human reader cannot analyze all of it.

The goal is to analyzing the Zomato Bangalore dataset is to understand people’s choice of cuisine, dining type, preference for location according to price etc.

1. **Preprocessing** - Load the dataset, retrieve the reviews (aka documents) and scores, encode the target (scores) and split into a training and test set.
2. **Training Model** - We will train a simple LSTM model that will predict the rating based solely on the review comments.
3. **Evaluation** - We will plot the loss and accuracy progression through epochs, and display the classification results as a table.

**Loading the dataset: Load the data and import the libraries.**

1. **Data Cleaning:**

* Deleting redundant columns.
* Renaming the columns.
* Dropping duplicates.
* Cleaning individual columns.
* Remove the NaN values from the dataset

1. **Text Preprocessing**

* Cleaning unnecessary words in the reviews
* Removing links and other unncessary items
* Removing Symbols

1. **Recommendation System**
2. RangeIndex: 51717 entries, 0 to 51716
3. Data columns (total 17 columns):
4. # Column Non-Null Count Dtype
5. --- ------ -------------- -----
6. 0 url 51717 non-null object
7. 1 address 51717 non-null object
8. 2 name 51717 non-null object
9. 3 online\_order 51717 non-null object
10. 4 book\_table 51717 non-null object
11. 5 rate 43942 non-null object
12. 6 votes 51717 non-null int64
13. 7 phone 50509 non-null object
14. 8 location 51696 non-null object
15. 9 rest\_type 51490 non-null object
16. 10 dish\_liked 23639 non-null object
17. 11 cuisines 51672 non-null object
18. 12 approx\_cost(for two people) 51371 non-null object
19. 13 reviews\_list 51717 non-null object
20. 14 menu\_item 51717 non-null object
21. 15 listed\_in(type) 51717 non-null object
22. 16 listed\_in(city) 51717 non-null object
23. dtypes: int64(1), object(16)
24. memory usage: 6.7+ MB

### Data Cleaning and Feature Engineering

<class 'pandas.core.frame.DataFrame'>

Int64Index: 43499 entries, 0 to 51716

Data columns (total 14 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 address 43499 non-null object

1 name 43499 non-null object

2 online\_order 43499 non-null object

3 book\_table 43499 non-null object

4 rate 43499 non-null object

5 votes 43499 non-null int64

6 location 43499 non-null object

7 rest\_type 43499 non-null object

8 cuisines 43499 non-null object

9 approx\_cost(for two people) 43499 non-null object

10 reviews\_list 43499 non-null object

11 menu\_item 43499 non-null object

12 listed\_in(type) 43499 non-null object

13 listed\_in(city) 43499 non-null object

dtypes: int64(1), object(13)

memory usage: 5.0+ MB

**Columns in a Dataset:**

Index(['address', 'name', 'online\_order', 'book\_table', 'rate', 'votes',

'location', 'rest\_type', 'cuisines', 'approx\_cost(for two people)',

'reviews\_list', 'menu\_item', 'listed\_in(type)', 'listed\_in(city)'],

dtype='object')

**Changing the columns names:**

Index(['address', 'name', 'online\_order', 'book\_table', 'rate', 'votes',

'location', 'rest\_type', 'cuisines', 'cost', 'reviews\_list',

'menu\_item', 'type', 'city'],

dtype='object')

**Changing the some datatypes like float to string:**

Data columns (total 14 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 address 43499 non-null object

1 name 43499 non-null object

2 online\_order 43499 non-null object

3 book\_table 43499 non-null object

4 rate 43499 non-null object

5 votes 43499 non-null int64

6 location 43499 non-null object

7 rest\_type 43499 non-null object

8 cuisines 43499 non-null object

9 cost 43499 non-null float64

10 reviews\_list 43499 non-null object

11 menu\_item 43499 non-null object

12 type 43499 non-null object

13 city 43499 non-null object

dtypes: float64(1), int64(1), object(12)

memory usage: 5.0+ MB

**Unique Ratings of the dataset:**

array(['4.1/5', '3.8/5', '3.7/5', '3.6/5', '4.6/5', '4.0/5', '4.2/5',

'3.9/5', '3.1/5', '3.0/5', '3.2/5', '3.3/5', '2.8/5', '4.4/5',

'4.3/5', 'NEW', '2.9/5', '3.5/5', '2.6/5', '3.8 /5', '3.4/5',

'4.5/5', '2.5/5', '2.7/5', '4.7/5', '2.4/5', '2.2/5', '2.3/5',

'3.4 /5', '-', '3.6 /5', '4.8/5', '3.9 /5', '4.2 /5', '4.0 /5',

'4.1 /5', '3.7 /5', '3.1 /5', '2.9 /5', '3.3 /5', '2.8 /5',

'3.5 /5', '2.7 /5', '2.5 /5', '3.2 /5', '2.6 /5', '4.5 /5',

'4.3 /5', '4.4 /5', '4.9/5', '2.1/5', '2.0/5', '1.8/5', '4.6 /5',

'4.9 /5', '3.0 /5', '4.8 /5', '2.3 /5', '4.7 /5', '2.4 /5',

'2.1 /5', '2.2 /5', '2.0 /5', '1.8 /5'], dtype=object)

**Removing ‘/5’ from rates:**

array([800. , 300. , 600. , 700. , 550. , 500. , 450. , 650. ,

400. , 900. , 200. , 750. , 150. , 850. , 100. , 1.2 ,

350. , 250. , 950. , 1. , 1.5 , 1.3 , 199. , 1.1 ,

1.6 , 230. , 130. , 1.7 , 1.35, 2.2 , 1.4 , 2. ,

1.8 , 1.9 , 180. , 330. , 2.5 , 2.1 , 3. , 2.8 ,

3.4 , 50. , 40. , 1.25, 3.5 , 4. , 2.4 , 2.6 ,

1.45, 70. , 3.2 , 240. , 6. , 1.05, 2.3 , 4.1 ,

120. , 5. , 3.7 , 1.65, 2.7 , 4.5 , 80. ])

**Unique cities:**

array(['Banashankari', 'Bannerghatta Road', 'Basavanagudi', 'Bellandur',

'Brigade Road', 'Brookefield', 'BTM', 'Church Street',

'Electronic City', 'Frazer Town', 'HSR', 'Indiranagar',

'Jayanagar', 'JP Nagar', 'Kalyan Nagar', 'Kammanahalli',

'Koramangala 4th Block', 'Koramangala 5th Block',

'Koramangala 6th Block', 'Koramangala 7th Block', 'Lavelle Road',

'Malleshwaram', 'Marathahalli', 'MG Road', 'New BEL Road',

'Old Airport Road', 'Rajajinagar', 'Residency Road',

'Sarjapur Road', 'Whitefield'], dtype=object)

**Checking NULL Values:**

address 0

name 0

online\_order 0

book\_table 0

rate 0

votes 0

location 0

rest\_type 0

cuisines 0

cost 0

reviews\_list 0

menu\_item 0

type 0

city 0

dtype: int64

**Some of the common text preprocessing / cleaning steps are:**

* Lower casing
* Removal of Punctuations
* Removal of Stopwords
* Removal of URLs
* Spelling correction

### Term Frequency-Inverse Document Frequency

Term Frequency-Inverse Document Frequency (TF-IDF) vectors for each document. This will give you a matrix where each column represents a word in the overview vocabulary (all the words that appear in at least one document) and each column represents a restaurant, as before.

TF-IDF is the statistical method of evaluating the significance of a word in a given document.

TF — Term frequency(tf) refers to how many times a given term appears in a document.

IDF — Inverse document frequency(idf) measures the weight of the word in the document, i.e if the word is common or rare in the entire document. The TF-IDF intuition follows that the terms that appear frequently in a document are less important than terms that rarely appear. Fortunately, scikit-learn gives you a built-in TfIdfVectorizer class that produces the TF-IDF matrix quite easily.

### One Hot Encoding

Let's look at the distribution of ratings

4.0 397337

5.0 372425

3.0 178102

1.0 149367

2.0 65870

3.5 55691

4.5 50292

2.5 17432

1.5 9333

Name: score, dtype: int64

Clearly this is categorical data, with some heavy class imbalance. This is something to address if you want to improve performance. We will go ahead and encode that into binary labels:

A picture containing text, electronics, keyboard, white

Description automatically generated

### Train Test Split

Finally, we split the data into train and test sets; the latter will be used to evaluate our model.

### Building an LSTM Model

At this point, we are ready to build our model. Similar to the original kernel, we will go through the following steps:

* Fit the Keras Tokenizer
* Build an embedding matrix
* Tokenize and pad our training data
* Train the model

Once we are done training the model, we evaluate how well it performs. This part is covered in the next section.

### Helper functions to create fast text embedding

def build\_matrix(word\_index, path):

def get\_coefs(word, \*arr):

def load\_embeddings(path):

def build\_model(embedding\_matrix):

words = Input(shape=(None,))

x = Embedding(\*embedding\_matrix.shape, weights=[embedding\_matrix], trainable=False)(words)

x = SpatialDropout1D(0.3)(x)

x = Bidirectional(CuDNNLSTM(256, return\_sequences=True))(x)

hidden = concatenate([GlobalMaxPooling1D()(x),GlobalAveragePooling1D()(x), ])

hidden = Dense(512, activation='relu')(hidden)

result = Dense(9, activation='softmax')(hidden)

model = Model(inputs=words, outputs=result)

model.compile(loss='categorical\_crossentropy',optimizer='adam',metrics=['accuracy'])

**Creating the tokenizer**

300-dimensional pretrained FastText English word vectors released by Facebook.

The first line of the file contains the number of words in the vocabulary and the size of the vectors. Each line contains a word followed by its vectors, like in the default fast Text text format. Each value is space separated. Words are ordered by descending frequency.

**Training**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param # Connected to

=============================================================================

input\_1 (InputLayer) (None, None) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

embedding\_1 (Embedding) (None, None, 300) 30280200 input\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

spatial\_dropout1d\_1 (SpatialDro (None, None, 300) 0 embedding\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

bidirectional\_1 (Bidirectional) (None, None, 512) 1142784 spatial\_dropout1d\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

global\_max\_pooling1d\_1 (GlobalM (None, 512) 0 bidirectional\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

global\_average\_pooling1d\_1 (Glo (None, 512) 0 bidirectional\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

concatenate\_1 (Concatenate) (None, 1024) 0 global\_max\_pooling1d\_1[0][0]

global\_average\_pooling1d\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 512) 524800 concatenate\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 9) 4617 dense\_1[0][0]

=============================================================================

Total params: 31,952,401

Trainable params: 1,672,201

Non-trainable params: 30,280,200

Train on 1049637 samples, validate on 116627 samples

Epoch 1/10

1049637/1049637 [==============================] - 687s 655us/step - loss: 0.9478 - acc: 0.6451 - val\_loss: 0.6368 - val\_acc: 0.7801

Epoch 00001: val\_acc improved from -inf to 0.78012, saving model to model.h5

Epoch 2/10

1049637/1049637 [==============================] - 685s 653us/step - loss: 0.6022 - acc: 0.7875 - val\_loss: 0.4353 - val\_acc: 0.8541

Epoch 00002: val\_acc improved from 0.78012 to 0.85407, saving model to model.h5

Epoch 3/10

1049637/1049637 [==============================] - 685s 652us/step - loss: 0.4702 - acc: 0.8362 - val\_loss: 0.3226 - val\_acc: 0.8939

Epoch 00003: val\_acc improved from 0.85407 to 0.89391, saving model to model.h5

Epoch 4/10

1049637/1049637 [==============================] - 685s 653us/step - loss: 0.3953 - acc: 0.8630 - val\_loss: 0.2734 - val\_acc: 0.9111

Epoch 00004: val\_acc improved from 0.89391 to 0.91111, saving model to model.h5

Epoch 5/10

1049637/1049637 [==============================] - 685s 653us/step - loss: 0.3452 - acc: 0.8803 - val\_loss: 0.2428 - val\_acc: 0.9217

Epoch 00005: val\_acc improved from 0.91111 to 0.92170, saving model to model.h5

Epoch 6/10

1049637/1049637 [==============================] - 685s 653us/step - loss: 0.3116 - acc: 0.8917 - val\_loss: 0.2155 - val\_acc: 0.9305

Epoch 00006: val\_acc improved from 0.92170 to 0.93045, saving model to model.h5

Epoch 7/10

1049637/1049637 [==============================] - 685s 653us/step - loss: 0.2856 - acc: 0.9006 - val\_loss: 0.2060 - val\_acc: 0.9342

Epoch 00007: val\_acc improved from 0.93045 to 0.93423, saving model to model.h5

Epoch 8/10

1049637/1049637 [==============================] - 685s 653us/step - loss: 0.2641 - acc: 0.9079 - val\_loss: 0.1842 - val\_acc: 0.9404

Epoch 00008: val\_acc improved from 0.93423 to 0.94036, saving model to model.h5

Epoch 9/10

1049637/1049637 [==============================] - 686s 654us/step - loss: 0.2504 - acc: 0.9124 - val\_loss: 0.1710 - val\_acc: 0.9456

Epoch 00009: val\_acc improved from 0.94036 to 0.94559, saving model to model.h5

Epoch 10/10

1049637/1049637 [==============================] - 686s 654us/step - loss: 0.2362 - acc: 0.9175 - val\_loss: 0.1660 - val\_acc: 0.9473

Epoch 00010: val\_acc improved from 0.94559 to 0.94725, saving model to model.h5

### Training Evaluation

Let's take a look at how well the model is training.

Chart, line chart

Description automatically generated