**3.1 CORPUS CREATION**

Using the corpus, we can build a sentiment classifier that will determine the positive, negative and neutral sentiments for a document. To perform sentiment analysis we created our own corpus. The data for finding sentiment is usually present on web like Hindi articles. Therefore Hindi articles were used to extract words. Those words were further translated into English. We used the VADER(valence aware dictionary and sentiment reasoner) created by Hutto, C.J. and Gilbert , E.E. which is a tool for sentiment analysis that is lexical and rule based. that is used for sentiment analysis of text which has both the negative as well as positive polarity. After finding the polarity of the English words using VADER , these were translated back to Hindi language. A database was used to store the Hindi words along with their respective polarity. It consist of two columns for word and polarity. If the polarity of the word is greater then zero it is stored in positive polarity , if polarity is less than zero it is stored in negative polarity else it is stored in neutral polarity. Also the synonyms and antonyms of the words are stored along with their polarity using web scraping.

1. Web scraping to extract words

|  |
| --- |
| for url in url\_list: |
|  | r = requests.get(url) |
|  | htmlContent = r.content |
|  |  |
|  | soup = BeautifulSoup(htmlContent, 'html.parser') |
|  |  |
|  | text = soup.get\_text() |
|  | l = text.split() # all website content |
|  | l2 = [] # all hindi content from the website |
|  | list = [] # final list |
|  |  |
|  | for element in l: |
|  | # devnagiri unicode : \u0900 to \u0963 and \u0972 to \u097f |
|  | if '\u0900' <= element <= '\u0963' or '\u0972' <= element <= '\u097f': |
|  | l2.append(element) |
|  |  |
|  | print('Creating sublists with hindi words and their respective polarities...') |
|  | for word in l2: |
|  | temp\_list = [] # for sublists |
|  | sia = SentimentIntensityAnalyzer() |
|  | eng\_word = hin\_to\_eng(word) |
|  | polarity = sia.polarity\_scores(eng\_word).get('compound') |
|  | temp\_list.append(word) |
|  | temp\_list.append(polarity) |
|  |  |
|  |  |
|  | cursorObj.execute('''INSERT INTO chatbot(word, polarity) VALUES (?, ?)''', temp\_list) |
|  |  |
|  | con.commit() |
|  |  |
|  | list.append(temp\_list) |
|  | print(temp\_list) |
|  |  |
|  | print('\nURL: ', url, '\nDATA STORED IN DATABASE SUCCESSFULLY\n') |

1. Web scraping to extract synonyms and antonyms of the previously stored words in the database

|  |
| --- |
| for i , j in dict\_words.items(): |
|  | eng\_word = hin\_to\_eng(i) |
|  | #print(eng\_word) |
|  | api\_url = 'https://api.api-ninjas.com/v1/thesaurus?word={}'.format(eng\_word) |
|  | response = requests.get(api\_url, headers={'X-Api-Key': 'xgCnBVIqBPJa5D9C0PFJHw==GCGiHFsbH3vJqxFy'}) |
|  | if response.status\_code == requests.codes.ok: |
|  | syn\_ant=response.text # a list of tuples |
|  | res = json.loads(syn\_ant) # dictionary |
|  |  |
|  | for k , l in res.items(): |
|  | if k=="synonyms": |
|  | for m in l: |
|  | temp\_list1=[] #for sub\_lists for synonyms |
|  | hin\_word=eng\_to\_hin(m) |
|  | # devnagiri unicode : \u0900 to \u0963 and \u0972 to \u097f |
|  | if '\u0900' <= hin\_word <= '\u0963' or '\u0972' <= hin\_word <= '\u097f': |
|  | temp\_list1.append(hin\_word) |
|  | temp\_list1.append(j) |
|  | cursorObj.execute('''INSERT INTO chatbot(word, polarity) VALUES (?, ?)''', temp\_list1) |
|  | print("iteration completed for synonyms") |
|  | con.commit() |
|  | elif k=="antonyms": |
|  | for m in l: |
|  | temp\_list2=[]# for sub\_lists for antonyms |
|  | hin\_word=eng\_to\_hin(m) |
|  | # devnagiri unicode : \u0900 to \u0963 and \u0972 to \u097f |
|  | if '\u0900' <= hin\_word <= '\u0963' or '\u0972' <= hin\_word <= '\u097f': |
|  | temp\_list2.append(hin\_word) |
|  | temp\_list2.append(j\*(-1)) |
|  | cursorObj.execute('''INSERT INTO chatbot(word, polarity) VALUES (?, ?)''', temp\_list2) |
|  | print("iteration completed for antonyms") |
|  | con.commit() |
|  | else: |
|  | print("Error:", response.status\_code, response.text) |
|  |  |

|  |
| --- |
|  |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  | |
|  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |

**3.2Applied rules**

**3.2.1 Pre-processing phase**

The dataset present on web consist of numbers , one length terms and all which mostly don’t contribute in sentiment analysis therefore such data are thoroughly processed to remove unwanted content. This phase contributes to sentence segmentation where paragraph are segmented into sentences. Further sentence tokenization is carried out where tokens are taken out of the sentence. Lastly stop words are removed as they don’t play major part in sentiment extraction.

* + 1. **Negation rule**

The negation operator (नहीं ,न,etc) present in the text inverts the sentimental meaning of the word following the negation words. The method to handle negation in sentiment analysis include considering a window of words of a fix size (typically 4 to 6) coming across the negation operator and reversing the polarity of all the words in the window. “ ! “ symbol is added to every word in the window for reversing it till either the sentence is completed or an violating expectation or a contrast or conjunction or a delimiter is encountered. Negation may be applied either in forward or in backward direction based on sentence structure.

**CASE 1:** ifa sentence has only one single negate word then all the words before the negation word are negated and the reverse polarity of the negated words are considered.

**Example :** 



**CASE 2 :** Forward negation is applied when a sentence has a negate word, conjunction and index of conjunction is more than the index of negated word . All the words after the conjunction will be negated.

**Examples :**





**CASE 3 :** If a sentence has “न” multiple times in sub-sentences separated by commas. For each “न” the negation is applied in forward direction until a delimiter is encountered.

**Examples :**





|  |
| --- |
|  |
| def negation(string): |
|  | neg\_count = 0 |
|  | conj\_count = 0 |
|  | conj\_list = ['और', 'कि', 'लेकिन', 'पर', 'मगर', 'चाहे', 'या', 'तो', 'क्योंकि', 'जब कि', 'एवं', 'इसलिए', 'या फिर', 'नहीं तो', 'जैसे कि'] |
|  | neg\_list = ['नहीं', 'न', 'नदारद'] |
|  | # example\_string = 'यह मूवी अच्छी नहीं है' |
|  | # example\_string = 'पूरी फ़िल्म इस तरह की नहीं बन पायी कि आम आदमी उसे पूरे समय रुचि से देखे' |
|  | # example\_string = 'न कहानी ढंग की है, न ही पटकथा और न ही निर्देशन' |
|  |  |
|  |  |
|  |  |
|  | string\_list = string.split() |
|  |  |
|  | for i in neg\_list: |
|  | for j in string\_list: |
|  | if i == j: |
|  | neg\_count += 1 |
|  | # print(i, ' --> negative count = ', neg\_count) |
|  |  |
|  | for i in conj\_list: |
|  | for j in string\_list: |
|  | if i == j: |
|  | conj\_count += 1 |
|  | # print(i, ' --> conjunction count = ', conj\_count) |
|  |  |
|  |  |
|  |  |
|  |  |
|  | # case 1: if sentence has only one negative word |
|  | # example\_string = 'यह मूवी अच्छी नहीं है' |
|  | if neg\_count >= 1 and conj\_count == 0: |
|  | for i in neg\_list: |
|  | if i in string\_list: |
|  | for j in range(string\_list.index(i)): |
|  | string\_list[j] = '!' + string\_list[j] |
|  |  |
|  | for i in neg\_list: |
|  | for j in string\_list: |
|  | if i == j: |
|  | string\_list.remove(j) |
|  | return string\_list |
|  | # case 2: if index of conjunction is greater than that of negative word |
|  | # example\_string = 'पूरी फ़िल्म इस तरह की नहीं बन पायी कि आम आदमी उसे पूरे समय रुचि से देखे' |
|  | elif neg\_count == 1 and conj\_count == 1: |
|  | for i in conj\_list: |
|  | for j in neg\_list: |
|  | if i in string\_list and j in string\_list and string\_list.index(i) > string\_list.index(j): |
|  | for j in range(string\_list.index(i) + 1, len(string\_list)): |
|  | string\_list[j] = '!' + string\_list[j] |
|  | for i in neg\_list: |
|  | for j in string\_list: |
|  | if i == j: |
|  | string\_list.remove(j) |
|  | return string\_list |
|  | # case 3: if sentence has 'न' multiple times, separated by commas |
|  | # example\_string = 'न कहानी ढंग की है, न ही पटकथा और न ही निर्देशन' |
|  | elif neg\_count > 1 and conj\_count >= 1: |
|  | for i in range(len(string\_list)): |
|  | if '!' in string\_list[i]: |
|  | string\_list[i] = string\_list[i][1:] |
|  | if string\_list[i] in neg\_list or string\_list[i] in conj\_list: |
|  | string\_list[i] = string\_list[i] |
|  | else: |
|  | string\_list[i] = '!' + string\_list[i] |
|  | for i in neg\_list: |
|  | for j in string\_list: |
|  | if i == j: |
|  | string\_list.remove(j) |
|  | return string\_list |
|  | else: |
|  | return False |
|  |  |
|  | # print(negation('यह मूवी अच्छी नहीं है')) |
|  | # print(negation('पूरी फ़िल्म इस तरह की नहीं बन पायी कि आम आदमी उसे पूरे समय रुचि से देखे')) |
|  | # print(negation('न कहानी ढंग की है, न ही पटकथा और न ही निर्देशन')) |
|  |  |

* + 1. **Discourse relation**

Discourse relation is used to establish a coherent relation, linking phrases and clauses in a text. Linguistic constructs like conditional, modals , connectives, etc. can change the accompanying sentiment contained in the text at the clausal or phrasal level in addition to the sentence level. The interaction of different discourse segments is reflected by coherent relation. Violated expectations like etc. are handled.

Violating expectation conjunctions oppose or refute the neighboring discourse segment which is categorized as Conj\_After and Conj\_Infer. Conj\_After is the set of conjunctions that give more importance to the discourse segment that follows them so the discourse segments after the Conj\_After are given preferences and the previous sentences are dropped. 

Conj\_Infer tend to draw a conclusion therefore the discourse segment following them are given more weight.



|  |
| --- |
| #conjunction\_list1 for conj\_after words |
|  | conjunction\_list1 = ["और", "व", "तथा", "एवं", \ |
|  | "यदि", "अगर", "तो", "चूंकि ", \ |
|  | "क्योंकि", "लेकिन", "परन्तु", "मगर", "पर", "फिर भी", \ |
|  | "ताकि", "अथवा", "या", "या तो", "या फिर", "वरना", \ |
|  | "के बजाय",'बावजूद'] |
|  | #conjunction\_list2 for conj\_infer or conclusive conjunction |
|  | conjunction\_list2=["इसीलिए"] |
|  |  |
|  | def discourse\_relation(sentence): |
|  | sentence\_in\_list=sentence.split() |
|  |  |
|  |  |
|  | # any func to check the existence of conjunction in sentence |
|  | check1=any(item in sentence\_in\_list for item in conjunction\_list1) |
|  | check2=any(item in sentence\_in\_list for item in conjunction\_list2) |
|  |  |
|  | #case1 for conj\_after |
|  | if check2: |
|  | for i in sentence\_in\_list: |
|  | for j in conjunction\_list2: |
|  | if(i==j): |
|  | return sentence\_in\_list[sentence\_in\_list.index(i)+1: ] |
|  |  |
|  | #case2 for conj\_infer |
|  | elif check1: |
|  | for i in sentence\_in\_list: |
|  | for j in conjunction\_list1: |
|  | if(i==j): |
|  | del sentence\_in\_list[:sentence\_in\_list.index(i)+1] |
|  | return sentence\_in\_list |
|  | else: |
|  | return sentence\_in\_list |
|  |  |
|  |  |
|  | # print(discourse\_relation("कहने को तो यह दो घंटे की फिल्म है लेकिन यह दो घंटे किसी सजा से कम नहीं है")) |
|  | # print(discourse\_relation("जो भी कहो कुल मिलाकर ब्रेक के बाद ब्रेक से पिछली ही अच्छी है")) |
|  | #print(discourse\_relation("वो लड़की बहुत अच्छी है")) |
|  | #print(discourse\_relation("तुम नहीं आये इसीलिए मैं भी नहीं गयी।")) |
|  | #print(discourse\_relation("तुम गलत हो कुल मिलाकर यह अच्छा है।")) |
|  | # print(discourse\_relation('मैं बदसूरत लड़का हूँ')) |

* + 1. **Intensifiers**

In grammar , intensifiers are those words that are added in front of an adjective or adverb and thus they make their meaning more stronger.

Examples : अत्यधिक , बहुत etc

Therefore as an adjective or adverb having intensifiers in front of them will have the polarity twice their previous polarity.

|  |
| --- |
| intensifiers\_list = ["बिल्कुल", "अत्यधिक", "बहुत"] |
|  |  |
|  | def intensifiers(input): |
|  | word\_list = input.split() |
|  | for i in range(len(intensifiers\_list)): |
|  | for j in range(len(word\_list)): |
|  | if intensifiers\_list[i] == word\_list[j]: |
|  | word\_list[j+1] = "\*" + word\_list[j+1] |
|  |  |
|  | if("बहुत" in word\_list): |
|  | word\_list.remove("बहुत") |
|  | # print(word\_list) |
|  |  |
|  | if("बिल्कुल" in word\_list): |
|  | word\_list.remove("बिल्कुल") |
|  | # print(word\_list) |
|  |  |
|  | if("अत्यधिक" in word\_list): |
|  | word\_list.remove("अत्यधिक") |
|  | # print(word\_list) |
|  |  |
|  | return word\_list |

* 1. **Use of slangs**

Slangs abbreviations are mostly used by the Internet users in their messages. Slang is a type of language of non-standard words and phrases such as “gr8”, “smh”, “xoxo”. The primary motivation behind using slang words includes its usefulness as it becomes easy for others to interpret and saves a lot of time. There exists large number of slangs that are categorized as positive or negative in terms of sentiment in chat. Therefore detection, translation and identification of slang's polarity have become very important.

Therefore Previous database is used for handling slangs in this project that stores around 35 slang words that are generally in use in our chat and are assigned tentative polarity.

**4. Creation co-occurrence relation**

Co-occurrence analysis is a technique often used in text mining , comparative genomics and promoter analysis. For our evaluation, we considered a collection of Hindi articles from web. Before using that text , pre processing was done by removing the stop words from the dataset as they don’t take part in further processing.

Secondly Bi-gram language model was used that includes finding bigrams which are two words such as “ is awesome” , “ user friendly” occurring together in any dataset. Using this model co-occurrence words in our dataset were fetched , their polarity was calculated using VADER. If the polarity was non- zero then the co-occurrence word was stored in a co-occurrence table in our database else it was skipped.

|  |
| --- |
|  |
| from nltk.sentiment import SentimentIntensityAnalyzer |
|  | from translator import eng\_to\_hin, hin\_to\_eng |
|  | import sqlite3 |
|  | import codecs |
|  |  |
|  | comparo\_list = [] |
|  | con = sqlite3.connect('chatbot.db') |
|  | cursorObj = con.cursor() |
|  |  |
|  |  |
|  | with codecs.open('co\_occurrence\_testfile.txt', encoding='utf-8') as f: |
|  | content = f.read() |
|  |  |
|  | print('Data read from file...') |
|  |  |
|  | sentences = content.split('। ') |
|  |  |
|  | print('Removing unwanted symbols from content...') |
|  | for sentence in sentences: |
|  | if '।' in sentence: |
|  | sentence = sentence.replace('।', '') |
|  | elif '\"' in sentence: |
|  | sentence = sentence.replace('\"', '') |
|  | elif '\'' in sentence: |
|  | sentence = sentence.replace('\'', '') |
|  | elif ',' in sentence: |
|  | sentence = sentence.replace(',', '') |
|  |  |
|  | words\_list = sentence.split() |
|  |  |
|  | for i in range(len(words\_list)): |
|  | if words\_list[i] != 'sw' and words\_list[i-1] != 'sw': |
|  | t = (words\_list[i-1], words\_list[i]) |
|  | comparo\_list.append(t) |
|  |  |
|  |  |
|  | print('Converting tuples into string...') |
|  | for i in comparo\_list: |
|  | str = ' ' |
|  | str = str.join(i) |
|  | comparo\_list[comparo\_list.index(i)] = str |
|  |  |
|  |  |
|  | print('Fetching polarity of co-occurring words...') |
|  | for i in comparo\_list: |
|  | sia = SentimentIntensityAnalyzer() |
|  | eng\_word = hin\_to\_eng(i) |
|  | polarity = sia.polarity\_scores(eng\_word).get('compound') |
|  | if polarity != 0.0: |
|  | co\_oc\_list = [] |
|  | print(i, ' --> ', polarity) |
|  | co\_oc\_list.append(i) |
|  | co\_oc\_list.append(polarity) |
|  | cursorObj.execute('''INSERT INTO co\_occurrence\_words(word, polarity) VALUES (?, ?)''', co\_oc\_list) |
|  | con.commit() |
|  | else: |
|  | print('Word NOT found') |
|  |  |
|  | print('\nPROCESS COMPLETE') |
|  |  |

**5. Implementation**

Initially input was taken from the user and the polarity of each word was taken out of the corpus developed by us. The rules mentioned above, such as removal of stop words, negation and discourse relation handling, handling intensifiers, slangs were applied. A co-occurrence table was developed that consists of co-occurrence words and their polarity in the dataset. Bi-gram method is then applied to the processed sentence that gives the co-occurrence words present in that sentence. These words are then inserted in the co-occurrence table if polarity is non-zero.

If a word is present in the created database, the total polarity will be calculated by adding the word's polarity in the database, its co-occurrence polarity (if any), and its polarity after applying intensifiers (if any). If it isn't present in the database, its total polarity will be calculated by translating it to English, adding the fetched compound polarity and its polarity after intensifiers (if any).

|  |
| --- |
|  |
| def total\_polarity(words): |
|  | import intensifiers |
|  | from nltk.sentiment import SentimentIntensityAnalyzer |
|  | from translator import hin\_to\_eng |
|  | from discourse\_relation import discourse\_relation |
|  | from negation import negation |
|  | from polarity\_finder import polarity\_finder |
|  | from co\_occurrence\_polarity import co\_occurrence\_polarity |
|  |  |
|  |  |
|  | # words = "मैं बहुत बदसूरत लड़का हूँ" |
|  | # words = "मैं बदसूरत लड़का पहुंच गई हूँ" |
|  | polarity\_after\_intensifiers = 0.0 |
|  | total\_polarity = 0.0 |
|  | co\_oc\_polarity = 0.0 |
|  |  |
|  |  |
|  | ################################## DISCOURSE RELATION ######################################## |
|  |  |
|  | discourse\_list = discourse\_relation(words) |
|  | discourse\_str = ' ' |
|  | discourse\_str = discourse\_str.join(discourse\_list) |
|  |  |
|  |  |
|  | ################################ NEGATION, INTENSIFIERS AND CO-OCCURRENCE WORDS ################################### |
|  |  |
|  | string\_list = discourse\_str.split() |
|  | post\_negation\_list = negation(discourse\_str) |
|  | co\_oc\_list = [] |
|  | final\_list = [] |
|  |  |
|  | if negation(discourse\_str) != False: |
|  | for i in post\_negation\_list: |
|  | temp\_list = [] |
|  | if polarity\_finder(i) == None: |
|  | sia = SentimentIntensityAnalyzer() |
|  | eng\_word = hin\_to\_eng(i) |
|  | polarity = sia.polarity\_scores(eng\_word).get('compound') |
|  | if '!' in i: |
|  | polarity = polarity \* -1 |
|  | i = i.replace('!', '') |
|  | temp\_list.append(i) |
|  | temp\_list.append(polarity) |
|  | else: |
|  | temp\_list.append(i) |
|  | temp\_list.append(polarity) |
|  |  |
|  | final\_list.append(temp\_list) |
|  | else: |
|  | polarity = polarity\_finder(i) |
|  | temp\_list.append(i) |
|  | temp\_list.append(polarity) |
|  | final\_list.append(temp\_list) |
|  | for i in final\_list: |
|  | total\_polarity = total\_polarity + i[1] |
|  |  |
|  | else: |
|  | after\_intensifiers = intensifiers.intensifiers(discourse\_str) |
|  |  |
|  | for i in after\_intensifiers: |
|  | if i[0] == "\*": |
|  | j = i[1:] |
|  | if polarity\_finder(j) == None: |
|  | sia = SentimentIntensityAnalyzer() |
|  | eng\_word = hin\_to\_eng(j) |
|  | polarity\_after\_intensifiers = polarity\_after\_intensifiers + (sia.polarity\_scores(eng\_word).get('compound')\*2) |
|  | after\_intensifiers.remove(i) |
|  | else: |
|  | polarity\_after\_intensifiers = polarity\_after\_intensifiers + (polarity\_finder(j)\*2) |
|  | after\_intensifiers.remove(i) |
|  |  |
|  | for i in after\_intensifiers: |
|  | temp\_co\_oc\_list = [] |
|  | temp\_co\_oc\_list.append(after\_intensifiers[after\_intensifiers.index(i)-1]) |
|  | temp\_co\_oc\_list.append(i) |
|  | co\_oc\_list.append(temp\_co\_oc\_list) |
|  |  |
|  | co\_oc\_list.remove(co\_oc\_list[0]) |
|  | co\_oc\_final\_list = [] |
|  |  |
|  | for i in co\_oc\_list: |
|  | co\_oc\_string = ' ' |
|  | co\_oc\_string = co\_oc\_string.join(i) |
|  | co\_oc\_final\_list.append(co\_oc\_string) |
|  |  |
|  | for i in co\_oc\_final\_list: |
|  | if co\_occurrence\_polarity(i) != None: |
|  | co\_oc\_polarity = co\_oc\_polarity + co\_occurrence\_polarity(i) |
|  | split\_string = i.split() |
|  | for j in split\_string: |
|  | for k in after\_intensifiers: |
|  | if j == k: |
|  | after\_intensifiers.remove(k) |
|  |  |
|  | for i in after\_intensifiers: |
|  | if polarity\_finder(i) == None: |
|  | sia = SentimentIntensityAnalyzer() |
|  | eng\_word = hin\_to\_eng(i) |
|  | total\_polarity = total\_polarity + (sia.polarity\_scores(eng\_word).get('compound')) |
|  | else: |
|  | total\_polarity = total\_polarity + (polarity\_finder(i)) + co\_oc\_polarity + polarity\_after\_intensifiers |
|  |  |
|  | return total\_polarity |
|  |  |
|  |  |