

# CS6200 Information Retrieval - Final Project Report

## Text Message Search

NUID: 001059643 - Madhuri Palanivelu

Github Source Code: <https://github.com/gonemad97/TextMessageSearchIR>

### Introduction:

In this day and age, it is difficult to find anyone who doesn't use mobile phones and the many popular mobile apps that they entail. Some of these popular apps may include Whatsapp, WeChat, Telegram, Messenger, etc. These particular apps pertain to Text and Voice Messaging platforms. These apps provide means for effective and simple communication with anyone, anywhere across the globe.

The features that these apps provide include sending pictures, videos, contacts, the user's geographic location and documents from their device. Another feature that these apps have in common, is the *Search-in-Conversation feature*. This feature allows users to search for a particular conversation that they may have had with someone in their contacts. This means, they'd be able to type in a particular phrase, and find conversations they've had with people in their contacts where that particular phrase was used. Sometimes, even the most popular and best of applications can have bug fixes and updates every now and then. Likewise, a particular problem that has been discovered, has to do with this Search-in-Conversation feature.

In this report, an alternative approach to the implementation of this problem will be discussed with a probable solution and a conversational search engine is built as a result. This should be considered as a stand-alone prototype of a *conversational search engine* that uses the existing app's Search features as inspiration. It is an alternative perspective to the feature from the eyes of a consumer.

### Problem Identified:

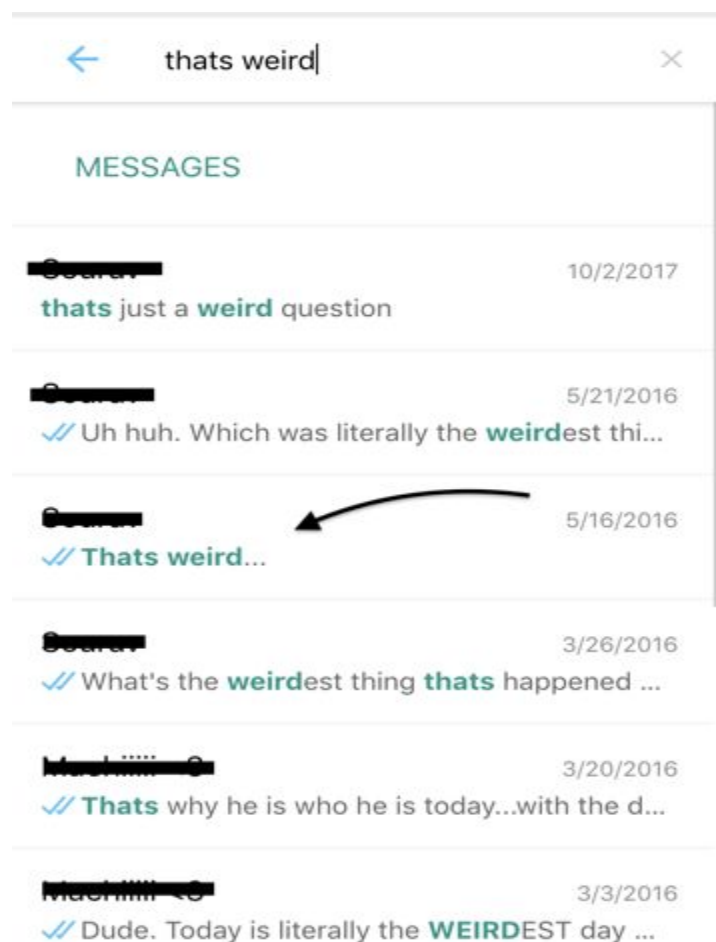
Searching through old text messages for information is always useful when the user is trying to remember or find something they may have spoken with one of their contacts about. The priority of this information depends on the user, but regardless, it is some sort

of information need. The issue identified is in regards to the relevance and ranking technique used, when displaying results to the user.

Here are the types of issues that are to be addressed by the proposed conversational search engine:

### 1. Ranking of the Results:

Let's consider the following example:



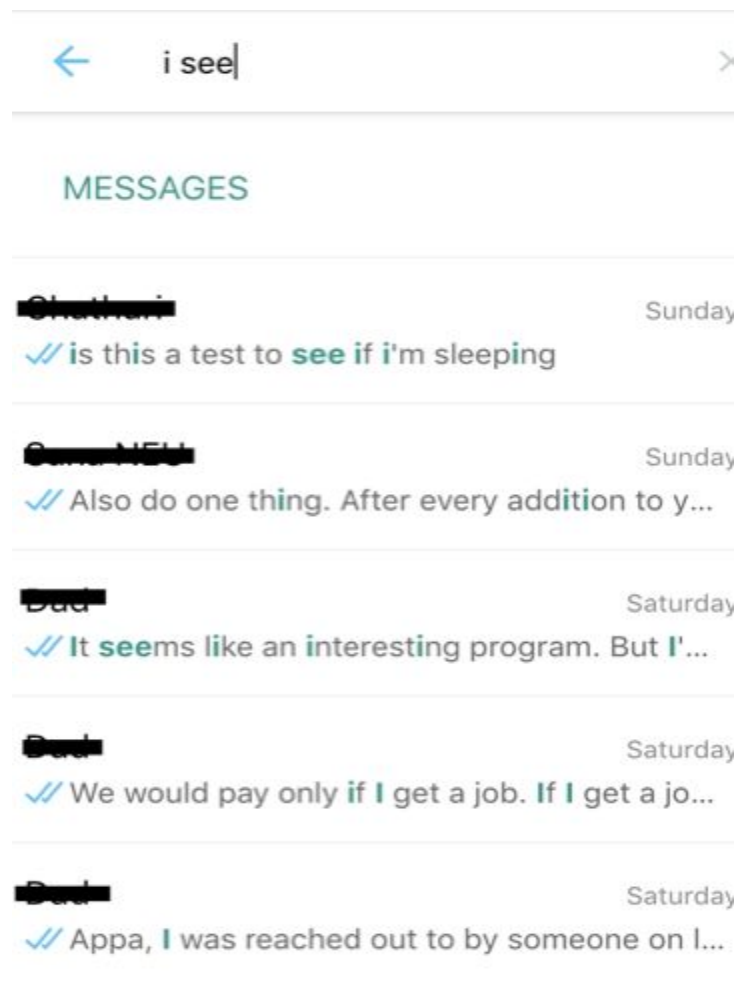
Here the search query entered by the user is "thats weird". The anticipated results that a user would expect after entering this phrase, would be a list of conversations where this *exact phrase* was used. But as displayed in the results of this example, the first occurrence of "thats weird" is placed third in line. It can be noted that,

there are more conversations containing the same phrase further down the ranked list of results. As observed, the conversation having any remote relation to this phrase is displayed based on the date it was texted by either a person in the user's contacts or by the user themselves.

Most of the time, the user won't really know the exact date they said a particular phrase and they're probably just looking for the person they conversed with. So in this regard, the method of **ranking** can be altered to something different - like the relevance or the number of times the phrase was used with a particular contact.

## 2. Types of Results Returned:

Note the following example:



In this example, note the very first search result and how it relates to the search query. Saying “i see” would generally mean the user sees or understands something. But the first result doesn’t bring the pronoun “I” anywhere. Words like “is”, “if” and “sleeping” are highlighted because of the letter “i” in them. This is quite flawed and can be considered an *abnormality in the highlighted terms*. The ranking is affected here as well because of the dates.

In the previous example, “that’s” and “weird” are considered as separate terms in the results too. There is even a case where “weirdest” is considered though it isn’t the exact term the user is looking for. When “i” is required, “i’m” is returned. Though these kinds of results don’t completely stray from the essence of what the user is looking for, they shouldn’t be present in the *top K results* when there are results that perfectly fit the user’s search query.

### 3. Synonymous Words:

Sometimes, a user may not remember the exact word they used in the place of another word. For example, maybe in an actual conversation, they said, “She’s been silent”, but in the search query, they input, “She’s been quiet”. In this case, random results with the terms split up in the top results, won’t help the user find what they are looking for. This is a classic *perception problem* (converting RIN to PIN)

This also goes without saying for *text abbreviations* like “brb”, “lol”, “gn”, etc. Sometimes the exact abbreviation could be used and at other times, maybe the whole full phrase would have been used in a conversation. This can also be addressed.

These issues get in the way of a user trying to satisfy their information need quickly, without frustration. This frustration stems from not being presented with the exact results they required in the first place, as a part of the initial results.

Let’s take a look at the proposed solution to these issues.

## Solution Proposed:

The most prominent solution to this entire issue, is the use of the *exact search query* itself in the event of a user searching for it. If the user asks for a particular phrase, every result having the exact phrase should be displayed in the top couple of ranked results.

The *ranking* can be done in terms of an approximate relevance to the user's request and be based on the *frequency of the exact phrase* in the conversation with a contact.

If there are very few results with the phrase the user is looking for, there could be a chance that they are looking for the same query but with a different *synonymous word* in the place of one in their query. Same goes for a *text abbreviation*. In this regard, some more results can be presented to the user with the synonym or the text abbreviation in place of more results that don't exactly fit the user's initial search query.

Before diving into the exact implementation of these features, let's first take a look at the dataset we'll be using to create a working conversational search engine prototype to apply these solutions to the issue at hand.

## Dataset:

In order to create a conversational search engine, it would make sense to create a corpus consisting of *actual human conversational data*. It isn't possible to obtain a set of actual text conversations between two people, because it violates privacy. So one way to obtain a dialogue corpus, would be to use the *transcripts of movies* and screenplays.

In order to collect this data, the movies (arranged in alphabetical order) need to be crawled from this website:

<https://www.scripts.com/>

This website contains all kinds of movies in various languages, transcribed in English, which would help increase the size of the dataset.

An important point to note with this dataset, is that this conversational search engine would be created to suit this dataset. *This means that the movie names should be*

*considered as the person a user would be texting, and the transcribed movie script should be considered as the conversation the user would have had with that person from the user's contact.*

After crawling and obtaining all the possible movie links from the website, a Python dictionary containing the movie titles as the keys and their respective scripts as the values, is created. Limited **tokenizing** is done for the scripts (conversation), in order to keep the authenticity of the text conversation.

Apostrophes are left in the words that have them in order to display appropriate results. An example for this could be results pertaining to “lets” or “let’s”. Both have different grammatical meanings. Apostrophes were also kept in consideration of words like “doesn’t”, “wasn’t”, etc.

The crawling is a process of about 3 hours. It retrieves 23,907 movies along with their scripts. It is required to save the crawled data in a structure so that it can be referred to while searching for resulting documents for the user's query. The structure used in this implementation is a **Python Pickle object**. Pickling is done for serializing and deserializing a Python object, in this case the dictionary containing the movie details, and saved onto the disk. It converts the dictionary into a character stream so that it can be reconstructed in another Python script. In our case, this other Python script would be responsible for deserializing the dictionary and using the information to fetch the user's search query results.

Another website was crawled for creating another smaller pickle file for 96 English words with a **synonym**, to be used for amplifying the results later.

## **Code Implementation Strategy:**

Now to get onto the nitty-gritty of the implementation. To begin with, these are the tools and libraries used to create this conversational search engine:

- Python BeautifulSoup (*Crawling*)
- Elasticsearch with Python (*Backend*)
- Python Flask (*User Interface*)

Crawling was discussed in the previous section, so Backend and UI is discussed below.

## **Backend:**

After the list of dictionaries containing the movie details is deserialized, each dictionary is inserted into a new Elasticsearch index. Initial indexing of all **23,907 records** takes about 3 minutes. After it has been indexed once, it does not need to be reindexed.

Next, by utilizing *Elasticsearch's Search API* and tuning the parameters a bit, the resultant set is created. The search functionality will contain the index of movies and their scripts and the type of query that is getting retrieved from the index. In light of the issue of partially retrieved phrases at hand, the query type is selected as “*match\_phrase*”, and the user's query will be used as the matching parent against the rest of each script.

“*match\_phrase*” will make sure to look for the exact phrase amongst the content indexed and it will not consider the terms of the phrase separately. Along with the type of query, a highlighting functionality is also introduced to help differentiate the query phrase in the final resulting items.

The highlight is applied with the default unified *highlighter* that uses BM25 to break the text into sentences and assign scores to them. A fragment size is entered as 100 as the size of the highlighted fragment in terms of character length. Finally, HTML tags <mark> and </mark> are set as the scripts pre and post tags in order to help differentiate the highlighted phrases from the rest of the script.

The “*hits*” field of the result set returned by Elasticsearch will contain more fields like “*score*” and “*\_source*”. “*\_source*”, which holds the JSON body that was initially indexed, can be manipulated to return the result item titles and content and the “*score*” field will hold the score assigned to that item. Two dictionaries are maintained to hold these scores and movie titles with their newly highlighted scripts. These dictionaries will be used next, for displaying the results of the user's search query in a ranked list.

To elucidate more on how the scoring is done by Elasticsearch, it uses the *Lucene Practical Scoring Function* (LPSF) to assign a strong relevance score that can be used

for ranking the documents based on the user's search query. The LPSF is a similarity model that is based on Term Frequency, Inverse Document Frequency and also uses the Vector Space Model in case of multi-termed queries. Query normalization and a coordination factor is also considered as a part of the LPSF. This helps deepen the relevance scoring.

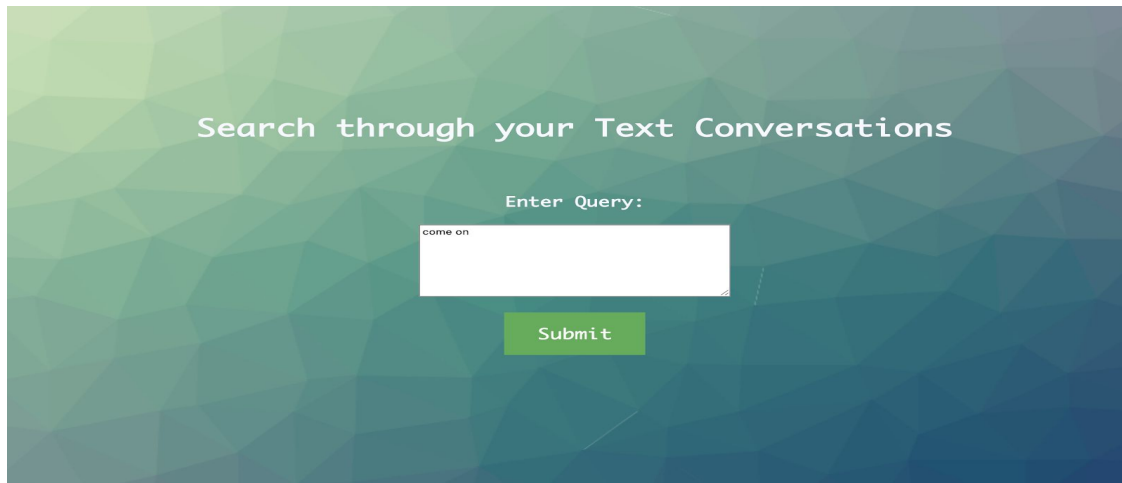
A heap is used to display the top movie names (aka User's Contact) and the fragmented script surrounding the highlighted phrase. The heap uses the scores to resemble a *max heap* and display the top 10 results to the user. If the query is a single word, and if the results returned for that query is less than 10, the synonym of that word is also searched for and the top few results are added at the end of the results based on how many more results are required to have 10 results in total. Either way, the remaining list of 10 results will be sent to the Flask application in order to create the appropriate User Interface.

Another module was also added where *textual abbreviations* were stored. In the case of this particular dataset, there weren't results pertaining to "brb" or "k" actually used in a movie script. So this module will convert those entries, if entered by a user to simulate a text conversation, into their full forms. The abbreviations can easily be added to the result when the dataset is actually text conversations. Here it was avoided in order to preserve better results in light of the dataset of movie transcripts used.

## **User Interface:**

*Flask* helps create the user interface by using the backend modules. The interface consists of *two web pages* for the user to view. The first and initial landing page will prompt the user to enter their query. The query from the text field is retrieved from the HTML and entered into the backend modules through Flask.



A search interface with a green and blue geometric background. At the top, the text "Search through your Text Conversations" is displayed. Below it, the label "Enter Query:" is followed by a text input field containing the text "come on". A green "Submit" button is positioned below the input field.

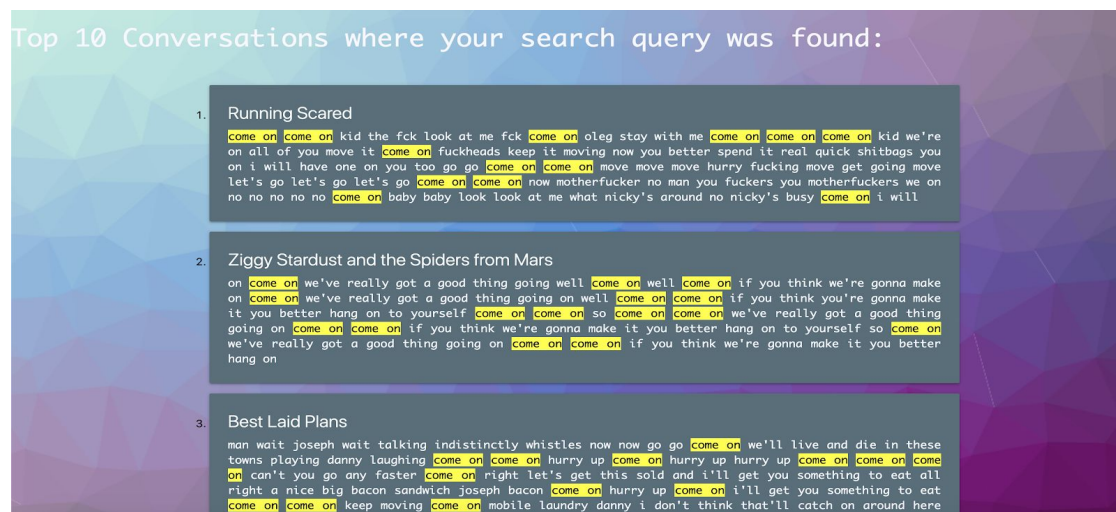
Search through your Text Conversations

Enter Query:

come on

Submit

The resulting list is stripped of the `<mark>` tags at this point as they are recognized in the HTML code written for the second webpage and replaced by the result with highlighted query terms. This is handled by a *JavaScript* function in the HTML code. The result list of top 10 movies (aka conversations) is passed onto the HTML code for the second webpage. This webpage will hold all the results crafted with *extensive CSS*. There will be a button allowing the user to go back a page and search for a new query if they'd like.

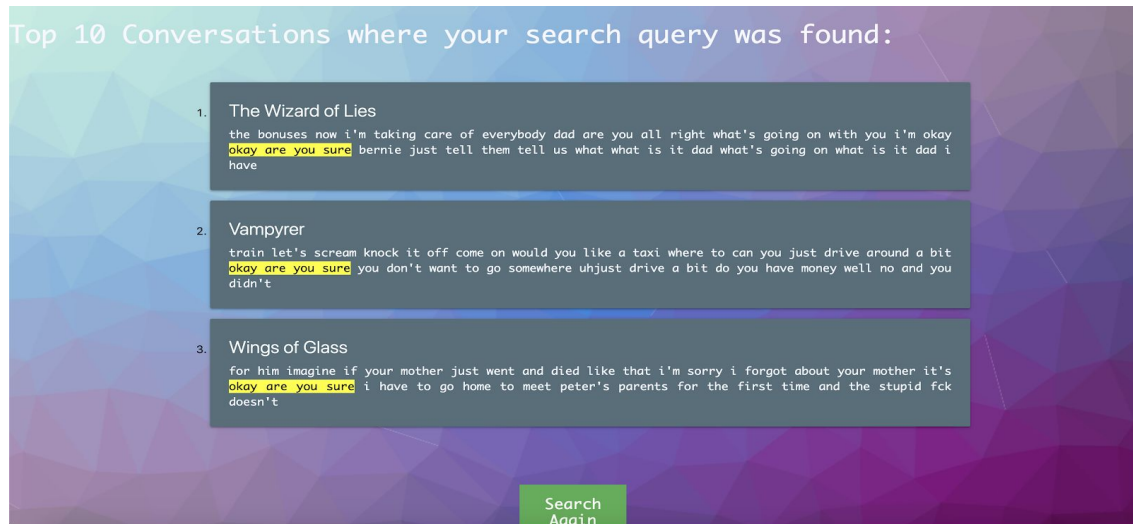
A screenshot of a webpage titled "Top 10 Conversations where your search query was found:". It lists three conversations with the search term "come on" highlighted in yellow. 1. Running Scared: "come on come on kid the fck look at me fck come on oleg stay with me come on come on come on kid we're on all of you move it come on fuckheads keep it moving now you better spend it real quick shitbags you on i will have one on you too go go come on come on move move move hurry fucking move get going move let's go let's go let's go come on come on now motherfucker no man you fuckers you motherfuckers we on no no no no no come on baby baby look look at me what nicky's around no nicky's busy come on i will". 2. Ziggy Stardust and the Spiders from Mars: "on come on we've really got a good thing going well come on well come on if you think we're gonna make on come on we've really got a good thing going on well come on come on if you think you're gonna make it you better hang on to yourself come on come on so come on come on we've really got a good thing going on come on come on if you think we're gonna make it you better hang on to yourself so come on we've really got a good thing going on come on come on if you think we're gonna make it you better hang on". 3. Best Laid Plans: "man wait joseph wait talking indistinctly whistles now now go go come on we'll live and die in these towns playing danny laughing come on come on hurry up come on hurry up hurry up come on come on come on can't you go any faster come on right let's get this sold and i'll get you something to eat all right a nice big bacon sandwich joseph bacon come on hurry up come on i'll get you something to eat come on come on keep moving come on mobile laundry danny i don't think that'll catch on around here".

Top 10 Conversations where your search query was found:

1. Running Scared  
come on come on kid the fck look at me fck come on oleg stay with me come on come on come on kid we're on all of you move it come on fuckheads keep it moving now you better spend it real quick shitbags you on i will have one on you too go go come on come on move move move hurry fucking move get going move let's go let's go let's go come on come on now motherfucker no man you fuckers you motherfuckers we on no no no no no come on baby baby look look at me what nicky's around no nicky's busy come on i will
2. Ziggy Stardust and the Spiders from Mars  
on come on we've really got a good thing going well come on well come on if you think we're gonna make on come on we've really got a good thing going on well come on come on if you think you're gonna make it you better hang on to yourself come on come on so come on come on we've really got a good thing going on come on come on if you think we're gonna make it you better hang on to yourself so come on we've really got a good thing going on come on come on if you think we're gonna make it you better hang on
3. Best Laid Plans  
man wait joseph wait talking indistinctly whistles now now go go come on we'll live and die in these towns playing danny laughing come on come on hurry up come on hurry up hurry up come on come on come on can't you go any faster come on right let's get this sold and i'll get you something to eat all right a nice big bacon sandwich joseph bacon come on hurry up come on i'll get you something to eat come on come on keep moving come on mobile laundry danny i don't think that'll catch on around here

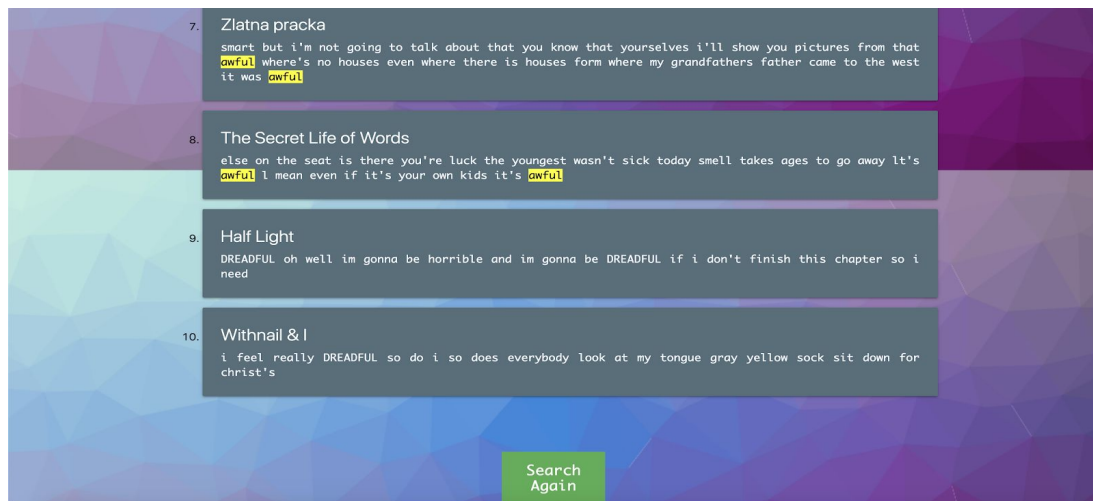
*Results are perfectly highlighted to show the full phrase that the user wanted to search for.*

This is the result returned if the user enters an abbreviation in their query:



*Results for the query, "k are you sure"*

This result shows the result of a single query with results less than 10. The query's synonym is capitalized to differentiate between the highlighted word and its synonym results.



*Results for the query, "awful", where there were 8 results and 2 results with "DREADFUL"*

## Conclusion:

This conversational search engine was built as a prototype application that targeted a particular problem that was perceived as a flaw in most existing texting platforms. It is nonetheless a stand-alone application for this course. Concepts learned throughout the course helped make this application a success and it also increased learning about Elasticsearch in general.