

Criterion C: Development

Python libraries used

A detailed description of each library is found in Criterion B

Python Library	Version
Beautifulsoup4	4.12.2
Pandas	2.1.1
Podcastparser	0.6.10
Urllib3	2.0.5
KeyBert	0.7.0
Db-sqlite3	0.0.1
Numpy	1.26.0
Gensim	4.3.2
Python-math	0.0.1
Requests	2.31.0
Flask	2.3.3
NLTK	3.8.1
Wordnet	0.0.1b2

List of techniques

Program Section	Complex Techniques
Use of API connection to validate inputs	<ul style="list-style-type: none">• Loops• Data structures• HTTP request• Web scraping• Error handling• Dictionary update
Use of NLTK lemmatisation and stopword removal to validate input	<ul style="list-style-type: none">• Pickle file opening• Stopword removal• Lemmatisation• List comprehension by filtering out stopwords
SQLite database creation, data query, and storage	<ul style="list-style-type: none">• Database Connection• Database Table Creation• Data retrieval• Loops• Data dictionary• Pandas Dataframe• Storing data in a data frame
Get podcast homepage URLs for each user input	<ul style="list-style-type: none">• HTTP request• Web scraping• String manipulation• 2D lists• Nested loops• Conditional logic• Data extraction• Data filtering
Get podcast homepage RSS feed URL	<ul style="list-style-type: none">• Loop• HTTP request• Data parsing with BeautifulSoup• Exception handling with 'try' and 'except'• Dictionary for data storage• Data extraction from HTML
Generate keywords from descriptions of each RSS feed episode	<ul style="list-style-type: none">• RSS parsing with 'podcastparser' library• HTTP request with 'urllib' library

	<ul style="list-style-type: none"> • Dictionary for data storage • Loop • Keyword Extraction with KeyBERT
Core NLP model: Word2Vector	<ul style="list-style-type: none"> • Error handling • Nested Loop • Word embedding (Word2Vec) • Numpy Operations • Centroid Calculations • Euclidean Distance calculation • Min-Max Normalisation function • Use of Gensim vector similarity function
Use of Flask to render HTML templates	<ul style="list-style-type: none"> • Flask application setup • Use of decorator • Setting a secret key
Fetch user input	<ul style="list-style-type: none"> • Flask request.form() • Flask session • Decorator for handling POST request
Use of CSS and HTML templates	<ul style="list-style-type: none"> • Use of @import rule • CSS selector • Font styling • Separation of concerns
Data table creation and visualisation	<ul style="list-style-type: none"> • List comprehension • Loop • Dictionary • Data transformation • Use of external libraries in rendering interactive tables • Defining parameters for table • Custom search functionality
Data table query	<ul style="list-style-type: none"> • DOM (document object model) selection • Event handling • Redirection • POST request • JSON serialisation • Promise handling
word cloud creation	<ul style="list-style-type: none"> • Jinja2 • External Library

	<ul style="list-style-type: none"> • JSON serialisation and deserialisation
Input autocomplete function	<ul style="list-style-type: none"> • jQuery Document ready function • Jinja2 • jQuery UI Autocomplete widget

SC: success criteria

Use of API connection to validate inputs

Looping through each user input

Sending request to google podcast API and using BS4 to interpret received data

```
def check_user_input(input1,input2,input3):
    user_input = [input1,input2,input3]
    result = {} #dictionary
    for each_input in user_input:
        iserror = 0
        base_url = 'https://podcasts.google.com/search/'
        search_url = base_url + each_input
        resp = requests.get(search_url)
        soup = BeautifulSoup(resp.text, 'lxml') #utilizes google podcast api to search for podcast results
        div_list = soup.find_all('div', class_="O9KIXe") #check if no podcast found using class property as web-scraping
        if len(div_list)!=0: #meaning that within class, there is a line: "no podcast found". So, the input is invalid
            iserror = 1
        result[each_input] = iserror

    error_msg = ''
    is_redirect = False
    for key in result.keys():
        if result[key] == 1:
            is_redirect = True
            error_msg += f"Input {key} is invalid. Please try again."
    return is_redirect, error_msg
```

Loops through dictionary to check if content of user input key contains 1 corresponding to an error whereby no podcast is found for that specific input

Using class property of webpage, I can check if podcast content is found. If the class contains a message, then no podcast is found for user input.

Using request and web-scraping tools that search through classes ^[1], this function validates if user input returns podcast results in Google podcast API. [**SC: 1) d) iii) (2)**] is achieved by using sessions ^[2] as a flag to check if no podcast results are returned. This is more efficient as all the processing is done on an external server.

Use of NLTK lemmatization and stopwords removal to validate input

```
# Define a list of trivial words
trivial_words = get_stop_words()
```

with statement ensures that file opened is closed after use

open() function opens the file for reading

'rb' specifies how file is read: (read binary)

handle stores file object returned by **open()**

```
def get_stop_words():
    with open('data/Stopwords.pickle', 'rb') as handle:
        stopwords = pickle.load(handle)
    return stopwords
```

.load() used to deserialise the file object in **handle**

[SC: 1) d) iii) (1)] is achieved by defining a list of stopwords.

WordNetLemmatizer() is imported from NLTK library

```
lemmatizer = WordNetLemmatizer()
singular_words = [lemmatizer.lemmatize(word.strip()) for word in word_list]
```

WordNetLemmatizer() is imported from NLTK library

.strip() removes any leading and trailing spaces in the word

Loops through each user input in **word_list**

While Regex handles common cases when converting plural words to their singular form, exceptions like 'bless' pose challenges. Using NLTK lemmatization ^[3] processes words to their singular form by interpreting the meaning in context, validating inputs more accurately and achieving [SC: 1) d) iii) (1)].

```
# Remove trivial words
cleaned_words = [word for word in singular_words if word.lower() not in trivial_words]
```

[SC: 1) d) iii) (1)] is met by filtering out stopwords after lemmatization.

SQLite database creation, data query, and storage

Database creation

Sqlite3.connect connects to database in current working directory. If it doesn't exist, it implicitly creates the database.

calling **conn.cursor** allows me to execute SQL statements such as querying and creating tables.

```
conn = sqlite3.connect('KEYWORD_MAP.db') #connecting to a database
cursor = conn.cursor()
cursor.execute('CREATE TABLE IF NOT EXISTS KEYWORD(user_input, keywords)') #creating a database table
conn.commit()
```

.commit() is able to execute the above statements.

Using the **CREATE TABLE** statement by calling the **.execute()** function, I am able to create a new table called **KEYWORD** containing user inputs and keywords from relevant podcasts.

Data query

pd.read_sql() reads the data from SQL database into a pandas dataframe named **df_result** using SQL query statement **SELECT** to get **user_input** and **keywords** from **KEYWORD** table.

Establishes connection to the database

```
df_result = pd.read_sql('SELECT user_input, keywords FROM KEYWORD', conn)

#comparing user_input from database with new user_input
for i in range(df_result['user_input'].count()):
    if(df_result['user_input'][i]==user_input):
        return(user_input,df_result['keywords'][i])
```

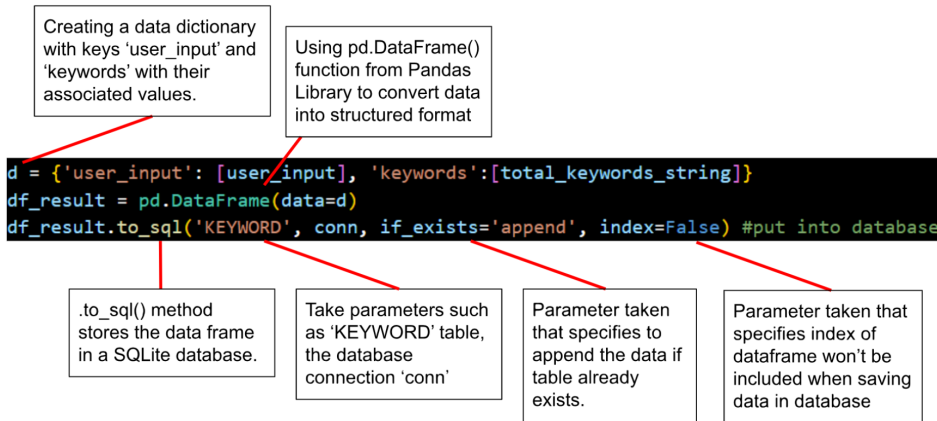
Return the user input and its relevant keywords stored in the dataframe by accessing it from

If the user inputs within dataframe match with the one entered by user in current session then keywords can just be fetched from dataframe

Looping through number of user inputs in the dataframe

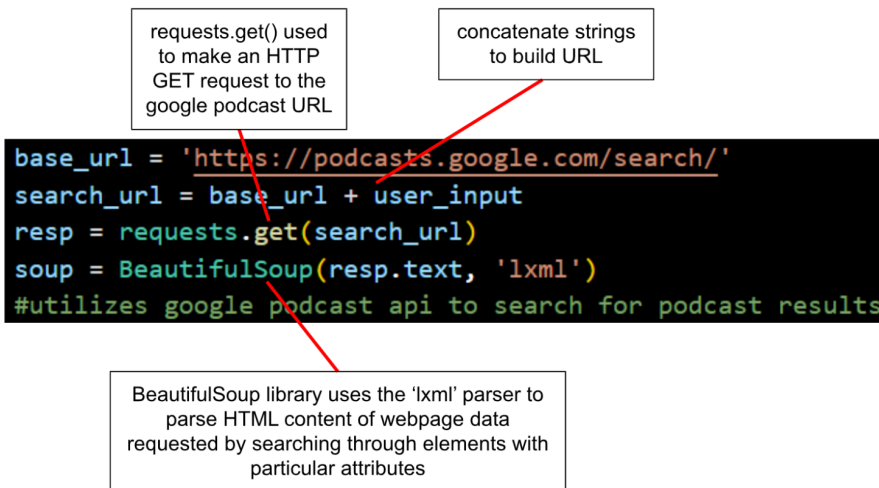
KeyBERT() language model ^[4] causes extended processing time for each input. To optimise user experience, saving inputs and their keywords in a database eliminates the need for processing user input, thus achieving [SC: 1) d) i)].

Storage



A data dictionary is often used for a data frame. Surpassing the manual SQL insertion approach, pandas .to_sql() method efficiently saves the data frame to a SQLite database ^[5], hence achieving [SC: 1) d) i)].

Get podcast homepage URLs for each user input



[SC: 2) b)]

```
results = soup.find_all('a', {'role': 'listitem'})
```

BeautifulSoup's `.find_all()` ^[6] finds podcast items in the soup content, identifying `<a>` elements with a 'listitem' role attribute, returning a list.

```
for result in results:
    podcast_url_part = result.get('href')[2:] #get the links of each podcast item
```

[SC: 2) c)] is achieved from the above webscraping method.

Retrieval of the podcast homepage URLs uses the same techniques.

```
new_homepage_urls = list(set(homepage_urls))
```

Converting a list to a set is simple, readable and has an average time complexity of $O(n)$ ^[7] whilst automatically eliminating redundant elements. [SC: 2) d)]

Get podcast homepage RSS feed URL

Loops through
homepage URLs

```
for pc_url in new_homepage_urls:
    google_podcast_url = pc_url
    url_getrssfeed = 'https://getrssfeed.com'
    headers = {'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.0.0 Safari/537.36'}
```

Dictionary used to store
additional HTTP request headers
for POST method

Used to identify client making
request containing info about its
browser and operating system

Used to mimic a Windows 10 machine
using a chrome browser, so that request
is treated as if it were made from web
browser

[SC: 2) e)] is achieved by first defining meta-data used for the POST request which returns the RSS feed for each podcast homepage. ^[8]

Requests method
sends a HTTP POST
request

Request sent
to this URL

Data parameter
contains dictionary
used to send data as
part of POST request

headers parameter
includes custom
headers to mimic a
browser agent

```
#to get podcast homepage rss url
r = requests.post(url_getrssfeed, data={"url":google_podcast_url}, headers=headers)
soup_getrssafterpost = BeautifulSoup(r.text, 'lxml')
try:
    rss_url = soup_getrssafterpost.find('div', {'class': 'mt-4'}).a['href']
except:
    print(f"Cannot retrieve rss feed from this {google_podcast_url}")
    continue
```

'try' and 'except'
block used in
error handling

Tries to find <div> element with class attribute set to 'mt-4'.
Then, retrieves the URL from this corresponding to the RSS
feed URL

Finally, BeautifulSoup library parses the RSS feed to identify important elements like the RSS feed URL.

Generate keywords from descriptions of each RSS feed episode

podcastparser library parses
the contents of the RSS URL
and retrieves structured
information and the feed

urllib.request.urlopen() is used
to open and read the contents
of the RSS feed URL

```
parsed = podcastparser.parse(rss_url, urllib.request.urlopen(rss_url))
#get descriptions for each rss feed episode
description = ''
for i in range(len(parsed['episodes'])):
    description = description + parsed['episodes'][i]['description']
descriptions[parsed['title']] = description
```

For loop iterating over the
range of the number of
episodes in the RSS feed

Dictionary with the podcast
homepage title as the key

Descriptions from each
episode is concatenated to
form a single string
description

[SC: 2) f)] is achieved by using the podcastparser library to parse each RSS feed URL to concatenate the descriptions of each episode together. ^[9]

```
kw_model = KeyBERT() #model using tone, word frequency, etc to find keywords from text
keywords = kw_model.extract_keywords(descriptions[i])
```

[SC: 2) f) ii)] & [SC: 3) a)] is achieved using KeyBERT which extracts podcast keywords from descriptions from each homepage using the .extract_keywords() method in NLP.

Core NLP model: Word2Vector

```
def word_to_vector(keyword_pool):
    each_keyword_vector_pool = []
    if keyword_pool is None:
        return [] # Return an empty list if keyword_pool is None
    for each_keyword in keyword_pool:
        try:
            # 'model' is your pre-trained Word2Vec model
            vector = model[each_keyword]
            each_keyword_vector_pool.append(vector)
        except KeyError:
            # Handle the case where the keyword is not in the model's vocabulary
            continue
    return each_keyword_vector_pool
```

[SC: 3) b)] is achieved using the Word2Vec model from the Stanford GloVe project.^[10] Using a pre-existing unsupervised learning algorithm from a large corpus to create word vectors is much more efficient than training my own, optimising the backend functionality of my program.

```
centroid_2 = pre_centroid_arr.mean(axis=0)
distance = np.sqrt(sum((pre_centroid_arr[0]-centroid_2)**2))
+np.sqrt(sum((pre_centroid_arr[1]-centroid_2)**2))
+np.sqrt(sum((pre_centroid_arr[2]-centroid_2)**2))
avg_distance = distance/3
```

To achieve [SC: 3) b) ii)], the centroid is calculated from the mean of all the keyword vector pools. Calculating the mean distance of each keyword vector pool to the centroid, the function can determine the degree of input relevance.

```
def C2_min_max_normalisation(C_dis):
    #normalisation to 0-1, the larger the more relevant
    return 1-((C_dis-C2_min)/(C2_max-C2_min))

c1_relevance,c2_relevance = C1_min_max_normalisation(distance1),
C2_min_max_normalisation(distance2)

final_centroid = c2.reshape(300)
#matching centroid vector with list of similar words
centroid_input1 = model.similar_by_vector(final_centroid)
centroid_input1 = np.array(centroid_input1)
```

Relevance is determined by min-max normalisation, ensuring a consistent scale expressed as a percentage.

Our hypothesis test confirmed the correlation between centroid distance and input relevance and thus confidence in recommendation. Therefore, using the Gensim .similar_by_vector function ^[11], the closest recommended keyword can be determined from euclidean distance to the centroid vector, thus achieving **[SC: 3) b) iii)]**.

Use of Flask to render HTML templates

The diagram illustrates the use of Flask to render HTML templates. It features a central code block with several annotations pointing to specific parts of the code:

- Only runs flask application if python script is executed as main programme**: Points to the `if __name__ == '__main__':` condition.
- Sets up Flask web framework**: Points to the `app = Flask(__name__)` line.
- Decorator used to define route for the root URL path ('/')**: Points to the `@app.route('/')` decorator.
- By using a secret key, Flask safeguards user sessions, data, and forms**: Points to the `app.secret_key = "super secret key"` line.
- Returns HTML template when user accessed root URL**: Points to the `return render_template('form.html')` line.

The code block contains the following Python code:

```
if __name__ == '__main__': #removes redundancies of rerunning models etc. increases efficiency

    app = Flask(__name__)
    app.secret_key = "super secret key"

    @app.route('/')
    def form():
        return render_template('form.html') #renders the frontend
```

```
return render_template('result.html',c2_relevance=c2_relevance,
| | | | | similar_word=similar_word, search_url=search_url)
```

render_template() function passes keyword arguments to HTML template to generate dynamic content, achieving **[SC: 5) a)]**. This method also follows separation of concerns such that application logic in python is separated from HTML presentation, enhancing maintainability and readability.

Fetch user input

Flask technique used to access and retrieve data submitted from HTML form

Decorator that defines a new route '/input_validation' for handling POST request

Unpack form data into three variables

```
@app.route('/input_validation', methods = ['POST'])
def input_validation():
    form_data = request.form #requesting the input data from form
    input_1, input_2, input_3 = form_data['Input1'],form_data['Input2'],form_data['Input3']
    is_redirect, error_msg = check_user_input(input_1,input_2,input_3)
    if is_redirect == False:
        session['user_input'] = [input_1, input_2, input_3] #session: datalog for individual users
```

Allows data to be fetched and stored across multiple HTTP requests for same user; acts as a dictionary

[SC: 1) c) d)] is achieved using Flask ^[12] to define routes where input data can be validated using `check_user_input()` function.

Use of CSS and HTML templates

```
@import url('https://fonts.googleapis.com/css2?family=Poppins&display=swap');

html {
| | height: 100%;
}
body {
| | margin-top:5rem;
| | font-family: 'Poppins', sans-serif;
| | background: linear-gradient(□#141e30, □#243b55);
}
```

main.css specifies features like background and font colours. ^[13]

```
<head>
  <link rel="stylesheet" href="static/form_display.css">
  <link rel="stylesheet" href="static/button.css">
  <link rel="stylesheet" href="static/main.css">
</head>
```

By using external CSS files linked in the <head> section, HTML content can be separated from CSS styles; if I alter the style I won't alter the content of the webpage, achieving **[SC: 1) a) b)]**. This type of modularity is also implemented by separating each HTML page and connecting them via Flask as shown in main.py. Use of separated functions means that they can be accessed from different python files, extending usage for different applications.

Data table creation and visualisation

.fetchall() returns a list of tuple records containing user input and its keywords and stores in records

```
# Fetch all records from the query result and convert to a list of dictionaries
records = cur.fetchall()
# keywords = [row_to_dict(record) for record in records]
keywords = [dict(user_input=record[0], keywords=record[1]) for record in records]
#This makes it easier to work with the data in a more structured way
```

dict() converts the tuple elements into a dictionary with user_input as the key and keywords as the value

Goes through each record of tuple data in the list

To achieve **[SC: 5) b)]**, data from database is first converted into a dictionary.

Grid.js library creates interactive tables with specific parameters: ^[14]

```
<script src="https://unpkg.com/gridjs/dist/gridjs.umd.js"></script>
```

Rendering and creating a grid using the Grid.js library by setting parameters for the grid

Use of jinja to receive keywords passed from `.render_template()` in `previous_inputs()` and storing it as a json format

```
const userData = {{ keywords | tojson | safe }};
new gridjs.Grid({
  columns: [
    { id: 'user_input', name: 'User Input' },
    { id: 'keywords', name: 'Keywords' },
  ],
  data: userData,
```

Defines the columns of the grid where the 'id' property specifies an identifier for each column

The data source for the grid uses `userData` which is a list of dictionaries stored in `keywords`

An interactive gridjs table is created with defined columns and corresponding data.

`search` is an object that defines how search functionality works

`selector` is a property of `search` taking in three parameters defining which cells in the grid should be searchable

The arrow function defines the following function linked to the `selector`

`.includes()` checks if `cellIndex` is 0 or 1, restricting search to either the first or second column within grid

```
search: {
  selector: (cell, rowIndex, cellIndex) => [0, 1].includes(cellIndex) ? cell : null,
},
sort: true,
pagination: true,
}).render(document.getElementById('database'));
```

Other parameters like `sort` and `pagination` are defined as true to allow the functionalities to be afforded

`.getElementById()` renders the grid js table in the HTML document where the identifier is 'database'

If cell not in these two columns, return null to the `selector`

Lastly, further functionalities like searching and sorting are granted using Grid.js library, thus conveniently rendering professional data tables, achieving **[SC: 5) b)]**. The external library greatly simplifies creating interactive tables. The configurability also allows the developer to personalise the appearance and table behaviour to suit project aim.

Data table query

`.addEventListener()` listens for a 'keyup' event that is triggered when a key on keyboard is pressed

Element within the class '`.gridjs-search-input`' is selected where user search queries are stored

```
const searchButton_1 = document.querySelector('.gridjs-search-input');

searchButton_1.addEventListener('keyup', function(event) {
  if (event.key === 'Enter') {
    getUserInputOnSearch();
  }
});
```

If the key pressed is the enter key then a function is called

Firstly, to achieve **[SC: 4) b)]**, the programme must 'listen' for user input.

```
// Function to perform a POST request when Enter key is pressed
function getUserInputOnSearch() {
  var user_query = document.querySelector('.gridjs-search-input');
  var user_query = user_query.value;
```

Next, `user_query` stores the query element for subsequent POST requests.

`navigate()` contains `windows.location.href` which redirects the user to a route within `main.py` defined using Flask

```
function navigate() {  
    window.location.href = 'embedding_projector'; // Redirects user  
}  
  
// Send the user input to the server via a POST request  
fetch('/query_user_input', {  
    method: 'POST',  
    headers: {  
        'Content-Type': 'application/json',  
    },  
    body: JSON.stringify({ user_query: user_query }),  
}).then(navigate);
```

`fetch()` initiates a post request to the specified route with parameters

`headers` specifies the format type of the content as JSON

Converts the Js object `user_query` into a JSON formatted string

`.then()` handler would call `navigate()` after post request is made

The `navigate() {...}.then(navigate)`, employs asynchronous programming that initiates a network request before further operations, enhancing readability. If the POST request fails, it can be debugged separately from the navigation function.

`.get_json()` function used to deserialise json data from POST request

Check if the predefined key is in the json file

Dictionary format pointing to key in fetching keywords

```
request_data = request.get_json() # Get JSON data from the request body  
if 'user_query' in request_data:  
    search_input = request_data['user_query']  
    conn = sqlite3.connect('KEYWORD_MAP.db')  
    cur = conn.cursor()  
    cur.execute('SELECT keywords FROM KEYWORD WHERE user_input = ?', (search_input,))  
    query_keywords = cur.fetchone()  
    query_keywords = query_keywords[0]  
    query_keywords = query_keywords.split(",")
```

`.split()` used to create list of keywords by splitting the string at each comma

Accessing the first element containing string of keywords

Used to fetch a single row of data from database cursor

Use of SQL query to find the corresponding keywords matching to the user_input that is equal to search_input

Lastly, [SC: 4) b)] is achieved by fetching the user input from the POST request and mapping it to its keywords from the database.

```
if query_keywords is not None:
    session['keyword'] = query_keywords
    session['searchQuery'] = search_input
    session['route'] = 1
    #keywordVector = word_to_vector(query_keywords)
    return "success"
```

Using sessions with specific keys in Flask allows storage and access across HTTP requests, so data can be accessed in different Flask routes, promoting organisation and continuity between pages and functions.

word cloud creation

Jinja **flash()** function allows search_input to be accessed as a variable via Jinja in an HTML document

```
if session['route'] == 1:
    search_input = session['searchQuery']
    search_input = ''.join(str(search_input))
    flash(search_input)
    route = 1
```

To achieve [SC: 5) a)], the session is a flag that determines if 'embedding_projector.html' is called from this route as the HTML document renders different content based on the originating route.

Jinja2 conditional statement

`get_flashed_messages()`
used to retrieve flash messages

```
{% if route == 1 %}
  {% with message = get_flashed_messages() %}
    <a href="http://127.0.0.1:5000" class="home-button">Home</a>
```

Thus, Jinja2 ^[15] is used to display the flashed data in the HTML document.

```
<script src="https://cdn.jsdelivr.net/npm/TagCloud@2.2.0/dist/TagCloud.min.js"></script>
```

To achieve either [SC: 5) a)] or [SC: 5) c)], TagCloud Js library ^[16] takes a list of words and displays an animated word cloud, allowing users to quickly grasp the important keywords relevant to their search input.

`TagCloud()` initialises the TagCloud library to create a word cloud visualisation.

`'contents'` specifies where the cloud will be rendered

`JSON.parse()` parses the JSON-formatted string and converts it to a Js object.

```
const myTags = JSON.parse('{{ keywordsForCloud | tojson | safe }}');
var tagCloud = TagCloud('.contents', myTags, {
  radius: 270,
  // animation speed
  maxSpeed: "fast",
  initSpeed: "fast",
  direction: 135,
  left: 0,
  // interact with cursor movement
  keep: true,
});
```

The rest of the parameters are to customise the visualisation and interaction of the word cloud.

`myTags` is the Js object containing the word cloud data

Input autocomplete function

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.js">
</script>
<script src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.8.16/jquery-ui.js">
</script>
```

To achieve **[SC: 1) d) ii)]**, the first script loads the jQuery library ^[17] making it easy to manipulate HTML document objects. The second script is used to create an interactive autocomplete box.

```
<script>
    $( function() {
        var availableTags1 = [
            {% for input_record in input_records %}
                "{{input_record}}",
            {% endfor %}
        ];
        $( "#tags1" ).autocomplete({
            source: availableTags1
        });
    });
};
```

Use of \$ defines a function in javascript that works with DOM (document object model) elements. **.autocomplete()** calls the variable availableTags1 and, using Jinja2, flashes the input records corresponding to the user input defined by the identifier "tags1", thus achieving **[SC: 1) d) ii)]**.

Word count: 972

Works Cited

DB Browser for SQLite, <https://sqlitebrowser.org/>. Accessed 1 July 2023.

Jinja — Jinja Documentation (3.1.x), <https://jinja.palletsprojects.com/en/3.1.x/>.

Accessed 15 November 2023.

Breuss, Martin. “Beautiful Soup: Build a Web Scraper With Python – Real Python.”

Real Python, <https://realpython.com/beautiful-soup-web-scraper-python/>.

Accessed 5 July 2023.

“[Flask教學] Flask Session 使用方法和介紹.” *Max行銷誌*, 23 September 2020,

<https://www.maxlist.xyz/2019/06/29/flask-session/>. Accessed 10 June 2023.

“GloVe: Global Vectors for Word Representation.” *Stanford NLP Group*,

<https://nlp.stanford.edu/projects/glove/>. Accessed 20 July 2023.

Grootendorst, Maarten. “KeyBERT - KeyBERT.” *Maarten Grootendorst*,

[https://maartengr.github.io/KeyBERT/api/keybert.html#keybert._model.Key](https://maartengr.github.io/KeyBERT/api/keybert.html#keybert._model.KeyBERT.extract_embeddings)

[BERT.extract_embeddings](https://maartengr.github.io/KeyBERT/api/keybert.html#keybert._model.KeyBERT.extract_embeddings). Accessed 25 June 2023.

“How to style buttons with CSS.” *W3docs*,

<https://www.w3docs.com/snippets/css/how-to-style-buttons-with-css.html>.

Accessed 10 September 2023.

Jain, Sandeep. "NLP Gensim Tutorial - Complete Guide For Beginners."

GeeksforGeeks, 7 November 2022,

<https://www.geeksforgeeks.org/nlp-gensim-tutorial-complete-guide-for-beginners/>. Accessed 2 August 2023.

Jain, Sandeep. "Python | Lemmatization with NLTK." *GeeksforGeeks*, 3 January 2023, <https://www.geeksforgeeks.org/python-lemmatization-with-nltk/>.

Accessed 18 June 2023.

"jQuery Tutorial." *W3Schools*, <https://www.w3schools.com/jquery/default.asp>.

Accessed 15 December 2023.

m, jose. "👉." *YouTube*, 30 August 2022,

https://github.com/miguelgrinberg/flask-gridjs/blob/main/templates/ajax_table.html. Accessed 10 November 2023.

Min, Cong. "Animated text sphere in JavaScript using TagCloud.js." *DEV*

Community, 2 August 2021,

<https://dev.to/asmitbm/animated-text-sphere-in-javascript-using-tagcloud-js-1p72>. Accessed 4 December 2023.

Perl, Thomas. "podcastparser · PyPI." *PyPI*, <https://pypi.org/project/podcastparser/>.

Accessed 15 July 2023.

“Python網頁設計: Flask使用筆記(二)- 搭配HTML和CSS.” *Yanwei Liu*, 5 April 2019,
<https://yanwei-liu.medium.com/python%E7%B6%B2%E9%A0%81%E8%A8%AD%E8%A8%88-flask%E4%BD%BF%E7%94%A8%E7%AD%86%E8%A8%98-%E4%BA%8C-89549f4986de>. Accessed 10 August 2023.

“Python Requests post Method.” *W3Schools*,
https://www.w3schools.com/python/ref_requests_post.asp. Accessed 4
June 2023.

“Python Requests post Method.” *W3Schools*,
https://www.w3schools.com/python/ref_requests_post.asp. Accessed 10
July 2023.

“Python set() Function.” *W3Schools*,
https://www.w3schools.com/python/ref_func_set.asp. Accessed 7 July
2023.