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

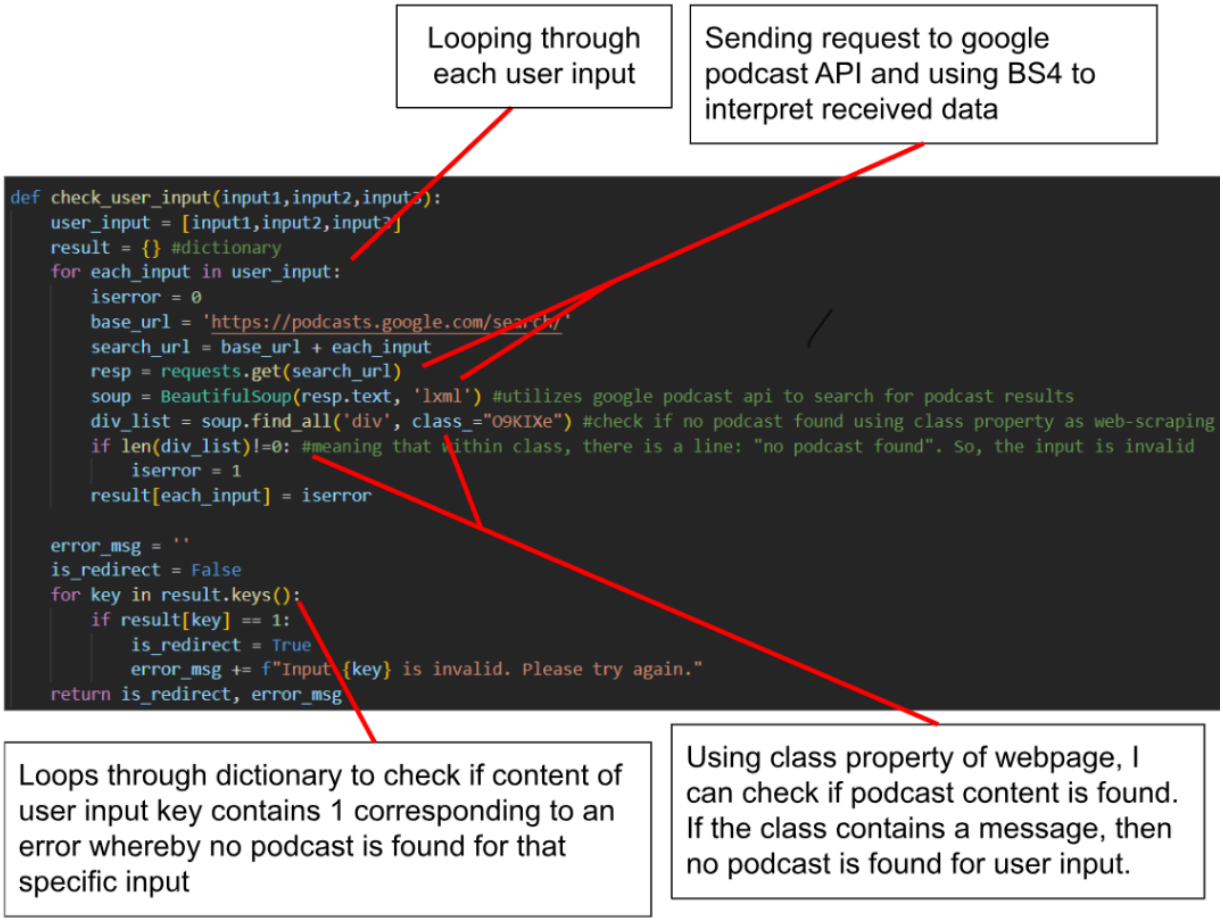
Program Section	Complex Techniques
Use of API connection to validate inputs	• Loops • Data structures • HTTP request • Web scraping • Error handling • Dictionary update
Use of NLTK lemmatisation and stopword removal to validate input	• Pickle file opening • Stopword removal • Lemmatisation • List comprehension by filtering out stopwords
SQLite database creation, data query, and storage	• 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	• HTTP request • Web scraping • String manipulation • 2D lists • Nested loops • Conditional logic • Data extraction • Data filtering
Get podcast homepage RSS feed URL	• 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	• RSS parsing with ‘podcastparser’ library • HTTP request with ‘urllib’ library

	• Dictionary for data storage • Loop • Keyword Extraction with KeyBERT
Core NLP model: Word2Vector	• 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	• Flask application setup • Use of decorator • Setting a secret key
Fetch user input	• Flask request.form() • Flask session • Decorator for handling POST request
Use of CSS and HTML templates	• Use of @import rule • CSS selector • Font styling • Separation of concerns
Data table creation and visualisation	• List comprehension • Loop • Dictionary • Data transformation • Use of external libraries in rendering interactive tables • Defining parameters for table • Custom search functionality
Data table query	• DOM (document object model) selection • Event handling • Redirection • POST request • JSON serialisation • Promise handling
word cloud creation	• Jinja2 • External Library

	• JSON serialisation and deserialisation
Input autocomplete function	• jQuery Document ready function • Jinja2 • jQuery UI Autocomplete widget

SC: success criteria

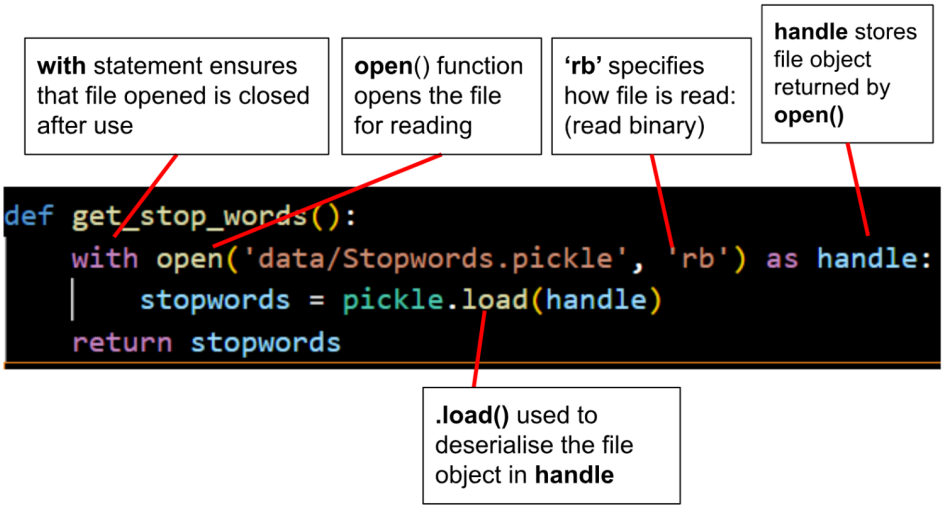
Use of API connection to validate inputs



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 stopword removal to validate input

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



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

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)].**

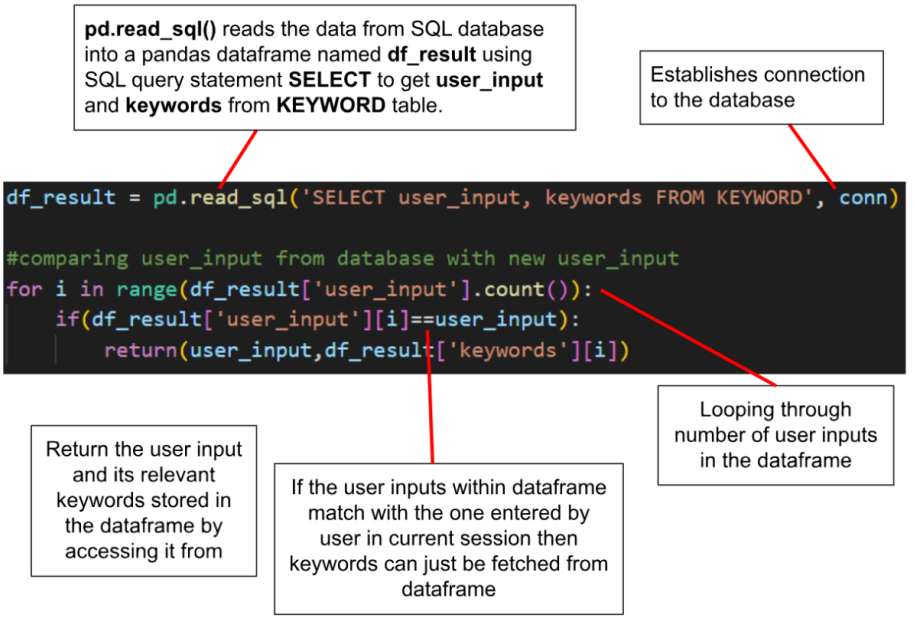
```
# Remove trivial words
cleaned_words = [word for word in singular_words if word.lower() not in trivial_words]
```

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

SQLite database creation, data query, and storage

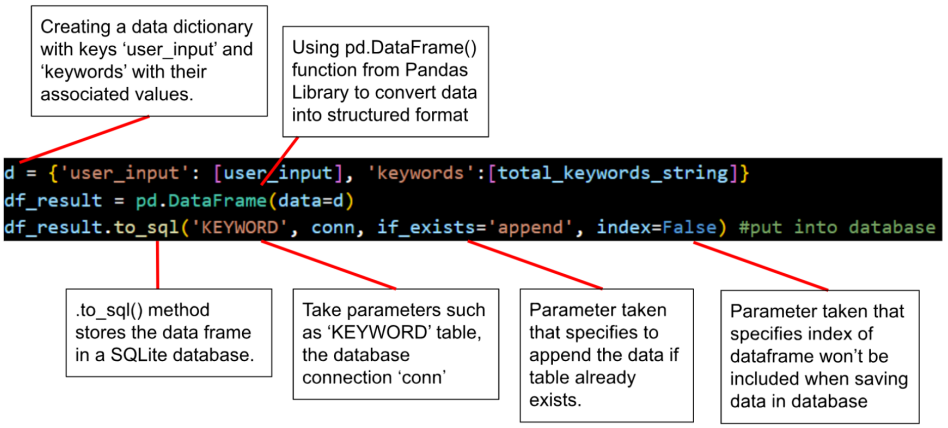
Database creation

Data query



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
```

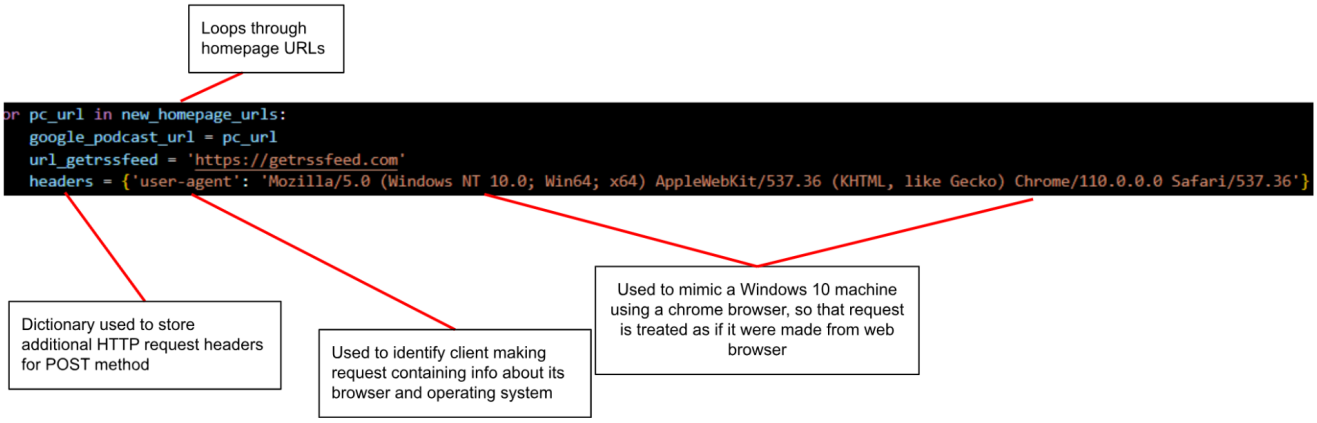
[SC: 2) c)] is achieved from the above webscraping method.

Retrieval of the podcast homepage URLs uses the same techniques.

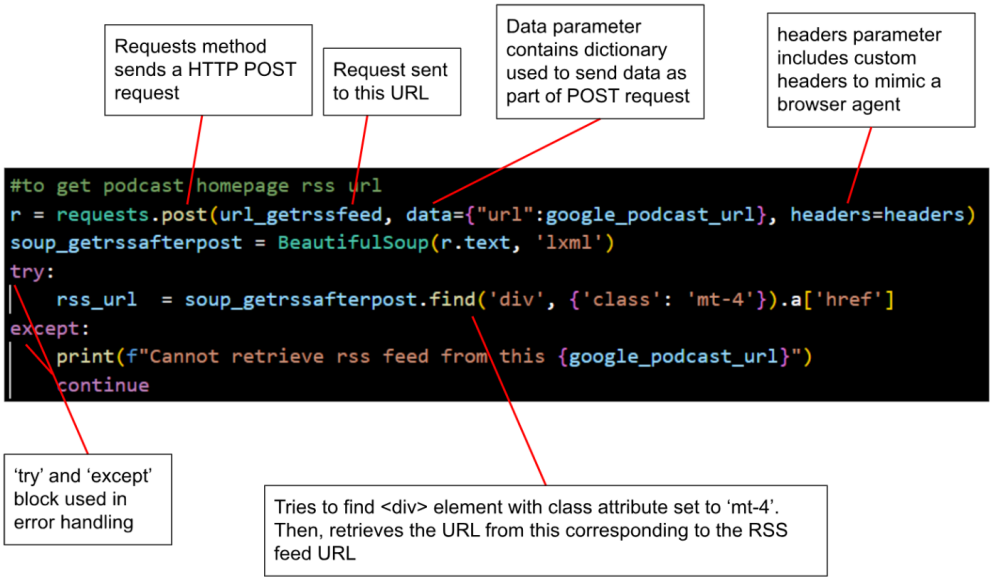
```
new_homepage_urls = list(set(homepage_urls))
```

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

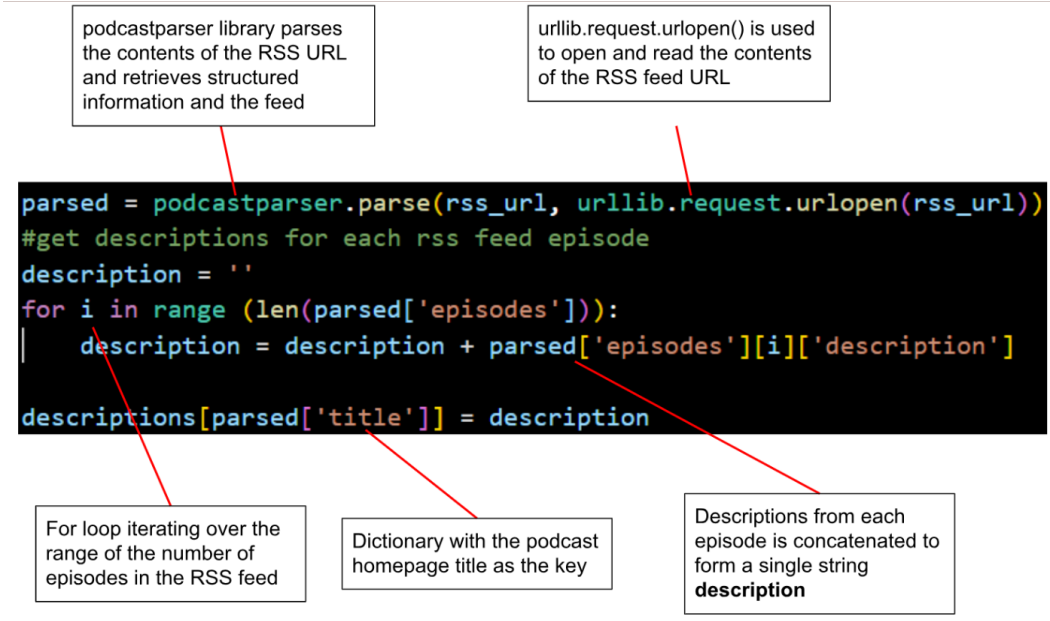


[SC: 2) e)] is achieved by first defining meta-data used for the POST request which returns the RSS feed for each podcast homepage. [8]



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



[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
```

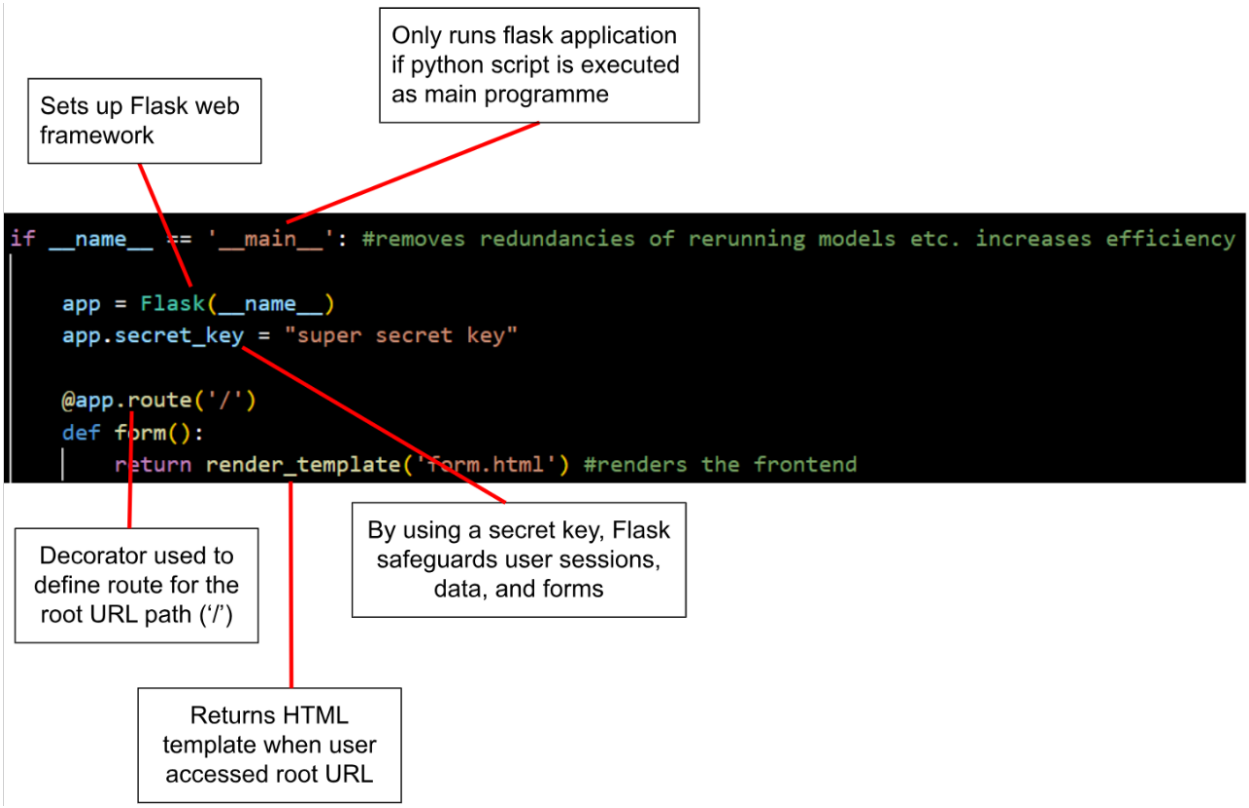
To achieve **[SC: 3) b) ii)],** the centroid is calculated from the mean of all the keyword vector pools.Calculating the mean distance of each keyword vector pool to the centroid, the function can determine the degree of input relevance.

```
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)].**

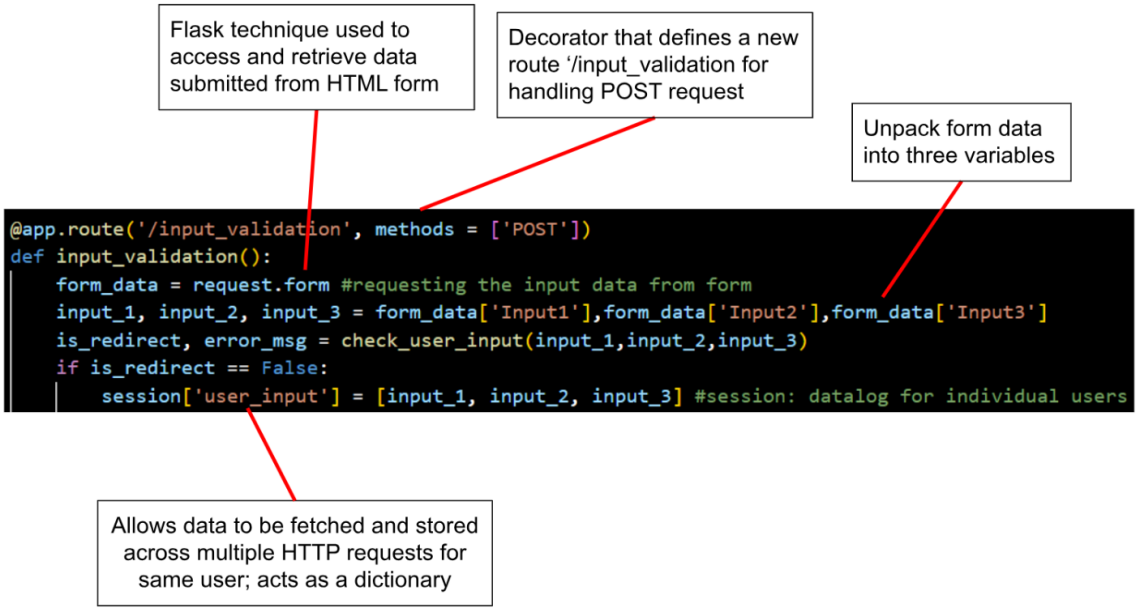
Use of Flask to render HTML templates



```
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



[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

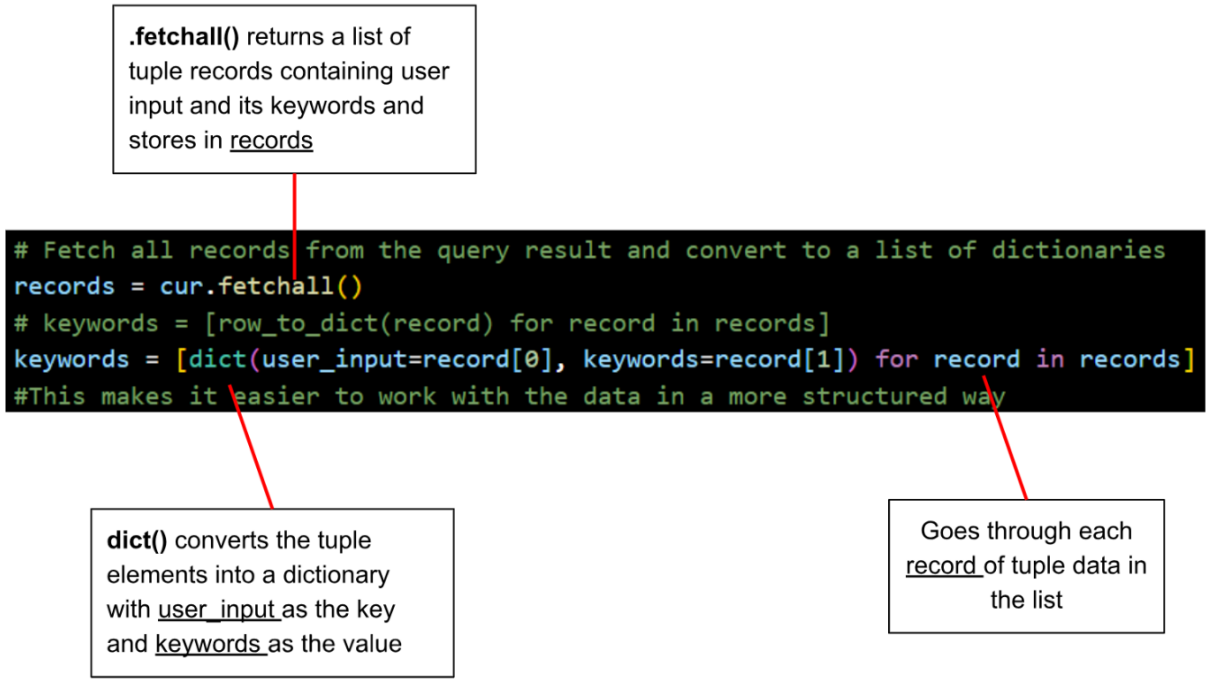
```
@import url('https://fonts.googleapis.com/css2?family=Poppins&display=swap');

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



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'));
```

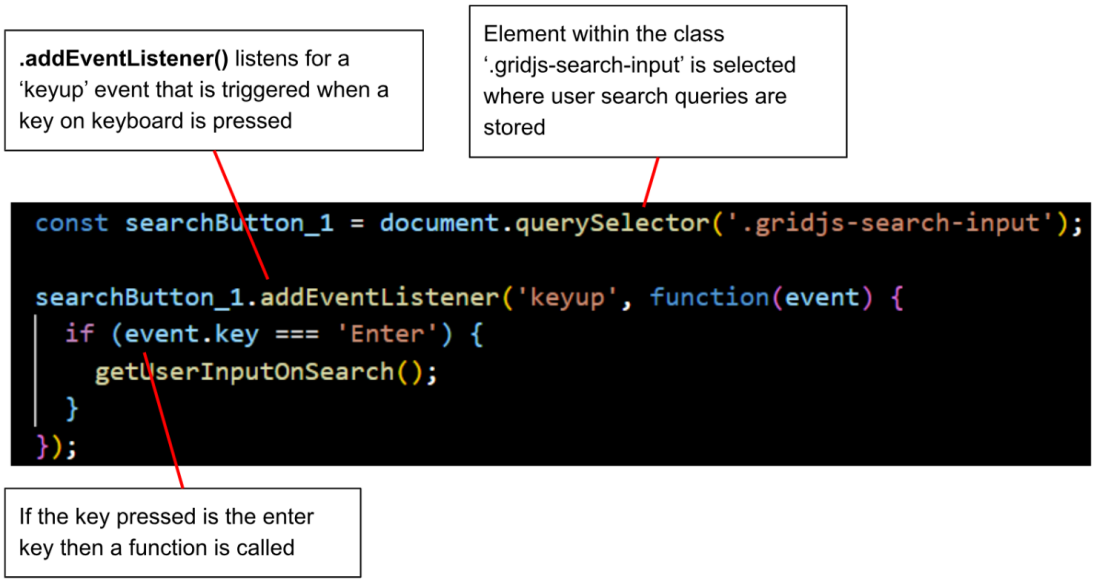
If cell not in these two columns, return null to the selector

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'

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.

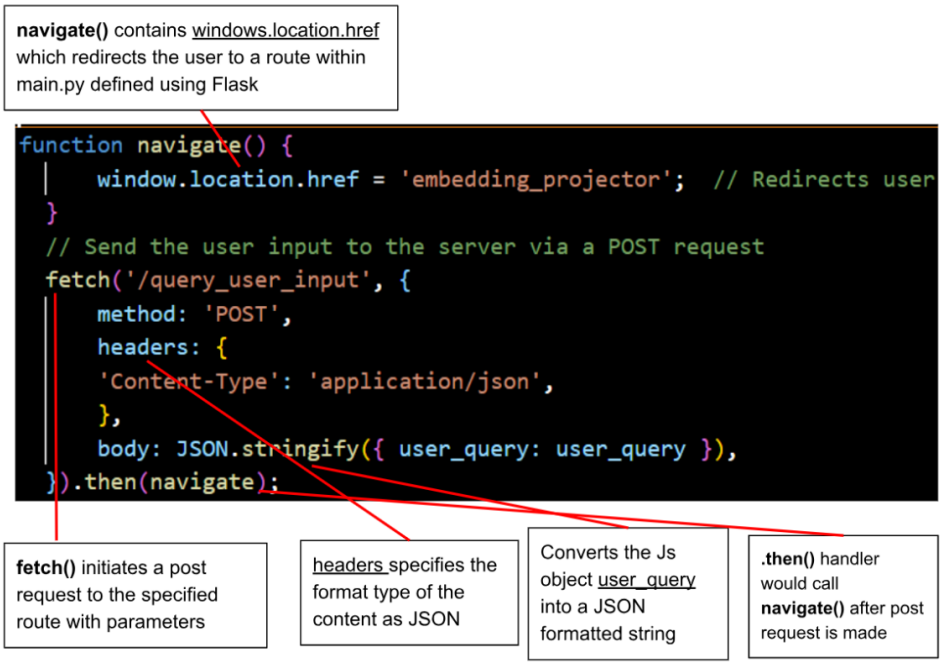
Data table query



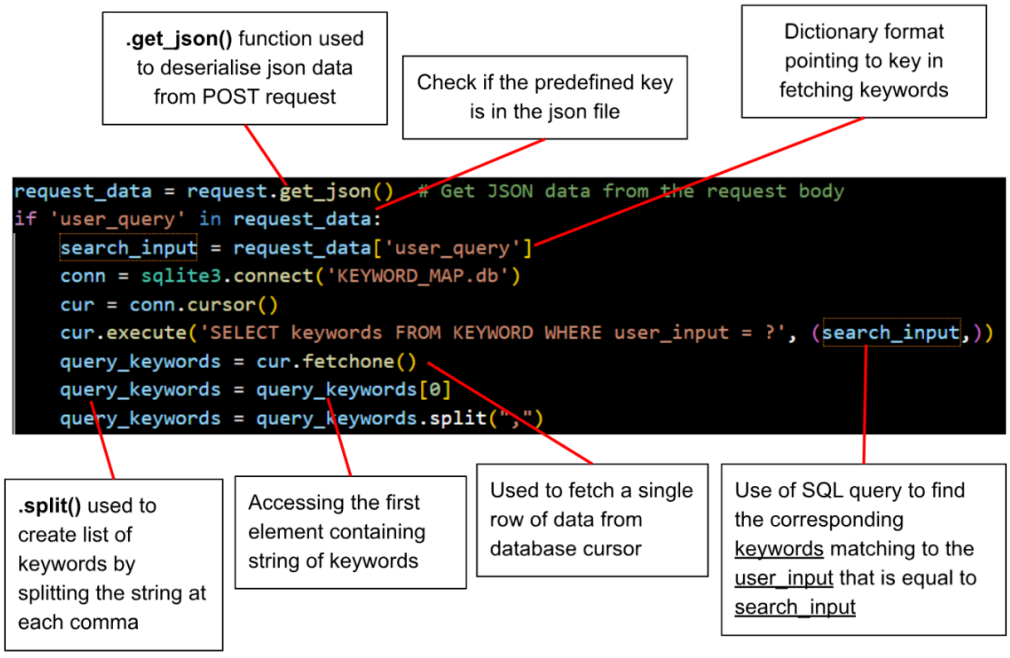
Firstly, to achieve **[SC: 4) b)],** the programme must 'listen' for user input.

```
// 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.



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.



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

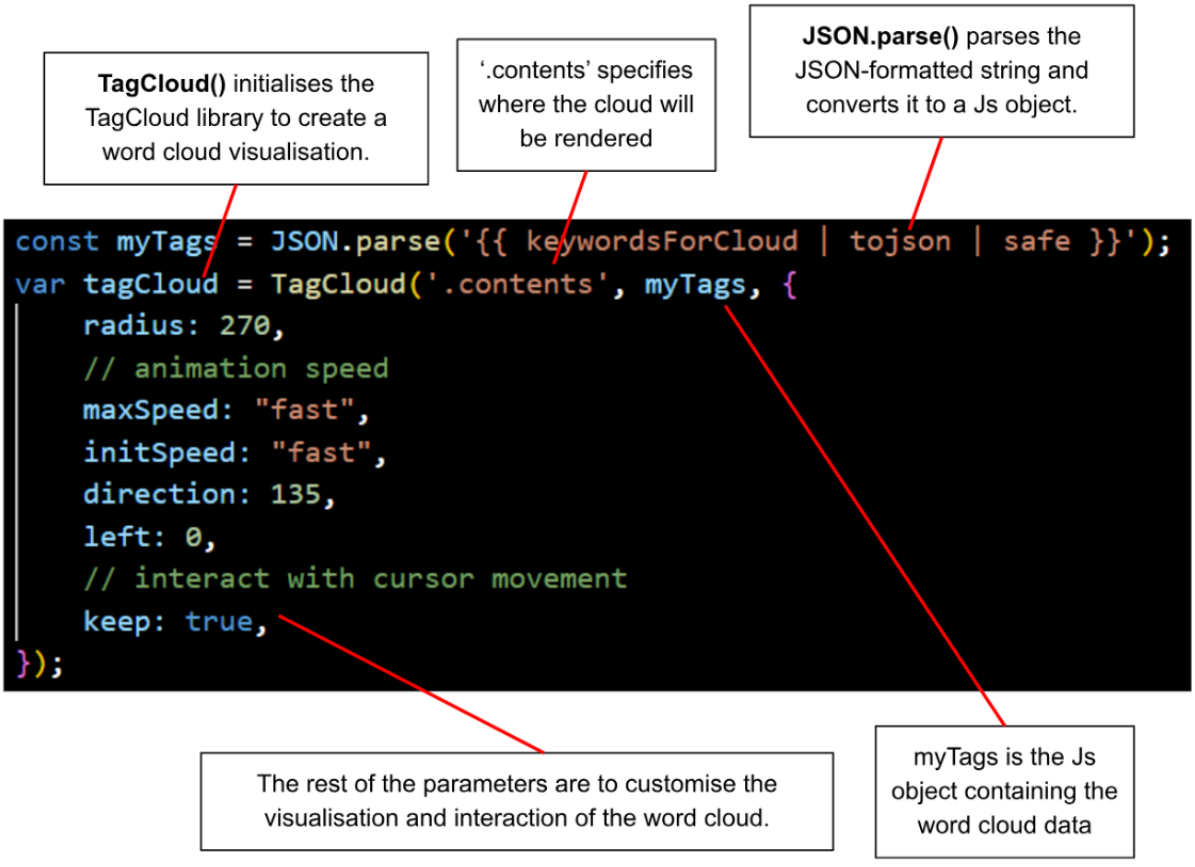
```
if session['route'] == 1:
    search_input = session['searchQuery']
    search_input = ''.join(str(search_input))
    flash(search_input)
    route = 1
```

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.

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.



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.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.