NEA Project: Hansard Data Project

Name: Freddie Stather
Computer Science NEA
Candidate Number: 7184
Centre Number: 57371

Analysis:

The problem:

Lots of people vote, and many people are interested in politics. But many people just don't know what their MP speaks about inside of parliament, within debates, speeches, questions, or what their MP stands for, or represents. There is a lot of information on what an MP says in Parliament, all speeches are recorded in Hansard, but there is currently no simple way to access this information.

This program is designed to solve that. The program is designed to provide an easier way to analyse speeches given by an MP in Parliament, and present that data in an easy to understand way. The program will use methods such as word clouds, graphs, semantic similarity, and other graphical representations that would help the user understand what their MP says and what they represent. This would help them be more aware of their politician's views and opinions, whether they be right or wrong, and to make more informed decisions when it comes to voting.

The Survey:

To gather information about the potential end users of the project, a survey has been made to get people's opinions on the topic. The survey (https://www.surveymonkey.co.uk/r/87X5WJD)[1] has a few questions that tells us about how active they are in local politics, if they have seen any speeches by their local MP, and how much they care about politics in general. The survey responders can tell us what they would like to be added to the project, which can help to influence the design phase of the project.

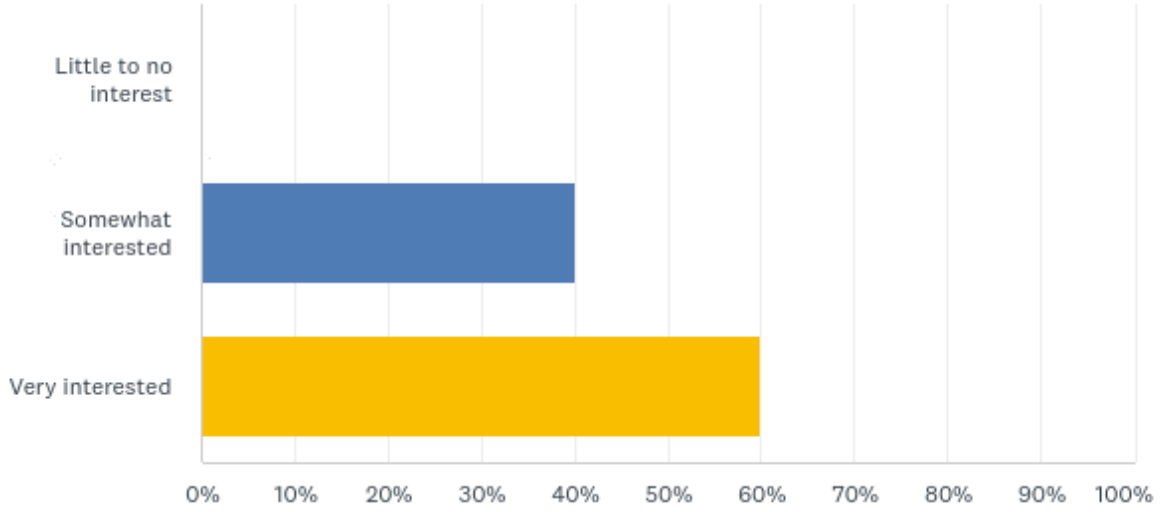
Using an example from the survey, people would want to see a visual representation of the speeches, like word clouds, or graphs. This would influence their vote in elections, and help make more informed decisions about their voting choices.

Most of the people who responded to the survey said they wanted the ability to search through the speeches done by MPs, which means it should be added to the

[1] https://www.surveymonkey.co.uk/r/87X5WJD

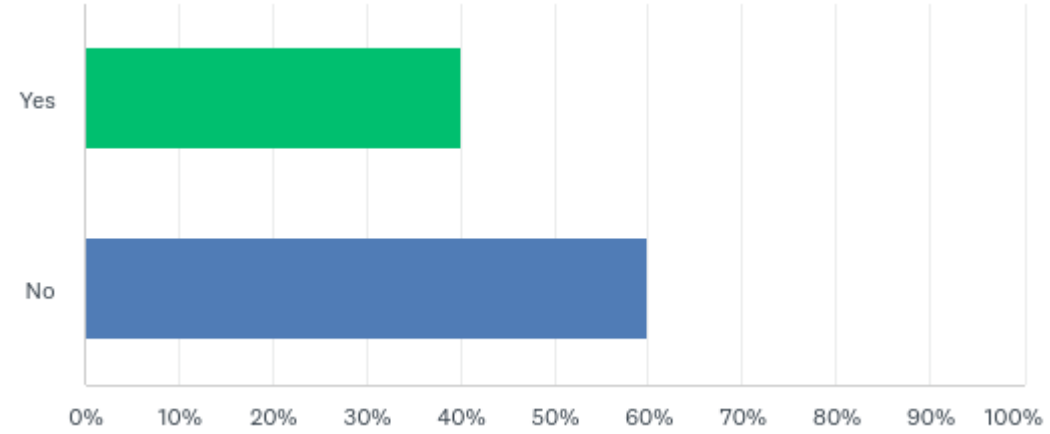
webpage, as this would make the program more relevant to the end user, and make the program better as a result.

Question 1: How interested are you in UK politics?



No one who responded to the survey said they had little to no interest in UK politics. Everyone who responded either were somewhat interested in politics, or very interested. This means that the project has some use, and isn't meaningless. It also means that we can get good responses to the other questions, as they are invested in politics.

Question 2: Have you ever read or listened to a speech given in Parliament by your local MP?



More people put no as the response to this question. This means that people have no idea what their MP talks about in Parliament, meaning this project is of good use

to them as everyone who answered the survey said they had an interest in politics, but has no clue of what is talked about in Parliament.

Question 3: Where did you read or listen to a speech given by your local MP?

BBC Parliament

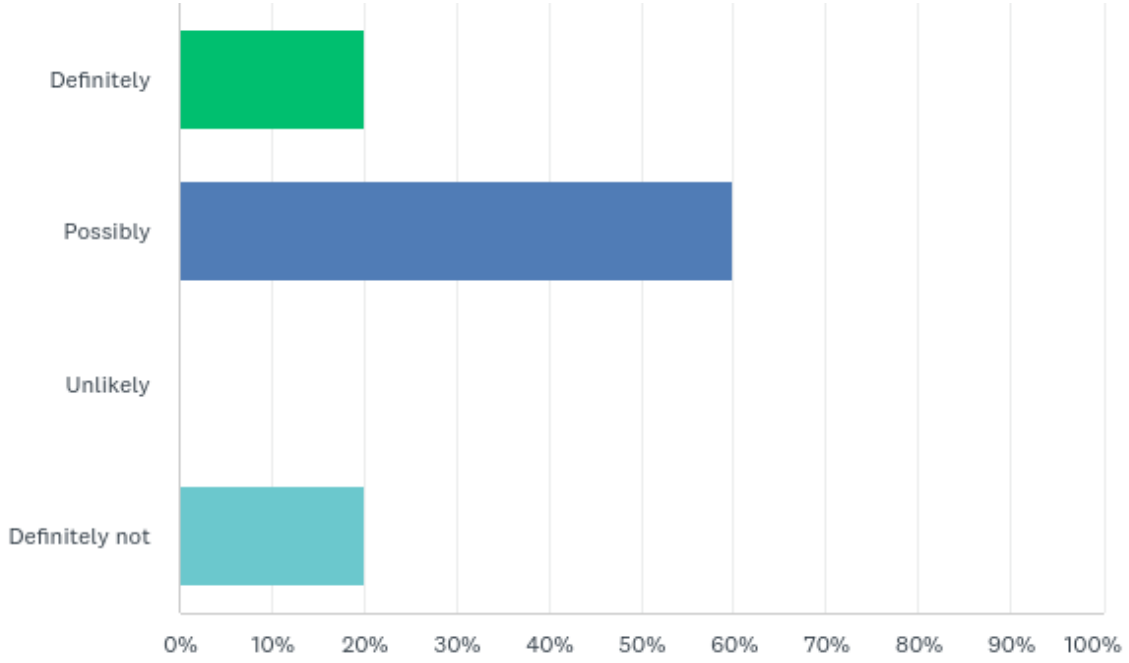
15/01/2023 20:24

He was on TV once

15/01/2023 20:21

More people skipped this question than answered it, meaning they have not read or listened to a speech by an MP. But the people that did answer said they had seen their local MP on TV, like Politics Live, or on the BBC Parliament channel. They had to seek that information out, so an easier place to store it all would be beneficial for them, and make it easier for them to find info about MPs.

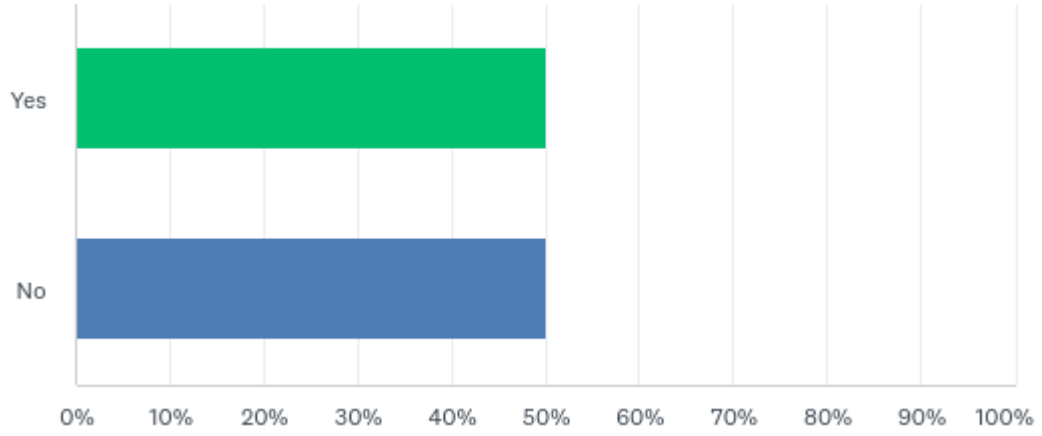
Question 4: Would having better information about your MP's speeches in Parliament influence your vote?



More people said that having this information would help to influence their vote than said not. This means that the project will not fall of deaf ears and have people that will use it to hopefully make better voting decisions in later elections.

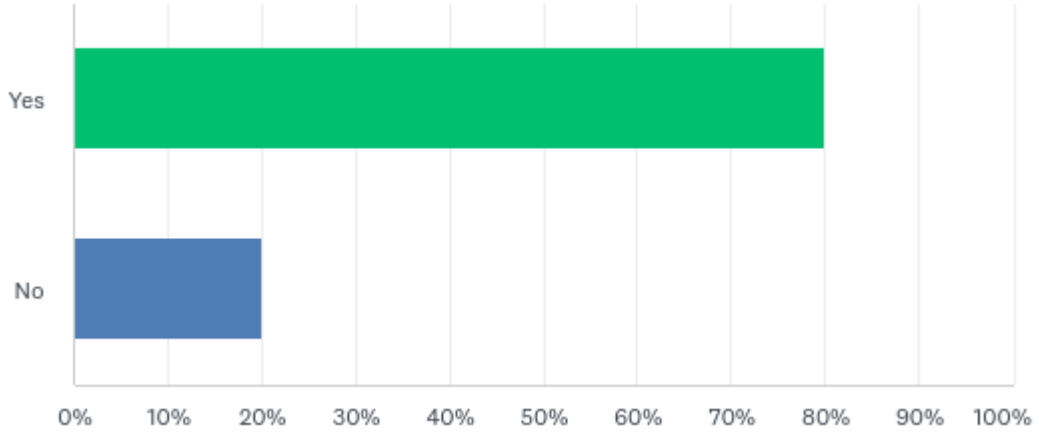
A few people said that having more information would not help influence their vote. This project could still be useful to them, as their opinions might change over time, by getting more information.

Question 5: Would you like more information about the speeches given by your local MP in Parliament?



An equal amount of people put yes and no for this question. The programme can still be of use for the people who said no, as their opinion could change over time. For the people who said yes, the programme is of more use to them as the data would be more meaningful, getting more out of the programme than the people who responded no.

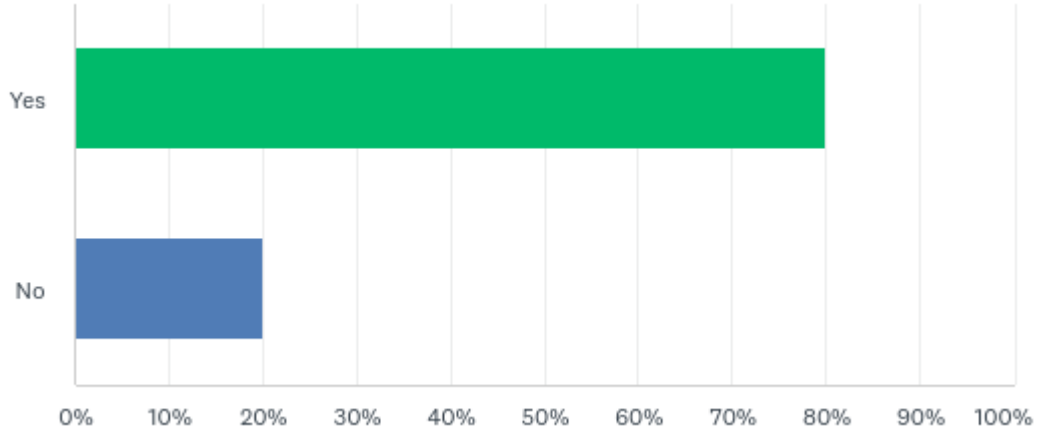
Question 6: Would you like the ability to search through the speeches given by your local MP?



Most people responded yes to this question, which means the programme has to include a search function to allow the users to search through speeches done by their local MP. This has to be done to meet the demands set by the survey respondents.

The people who said no probably have no interest in this programme anyway, so their demands do not have to be met.

Question 7: Would you like a visual representation of the speeches given by your local MP? For example, a word cloud.



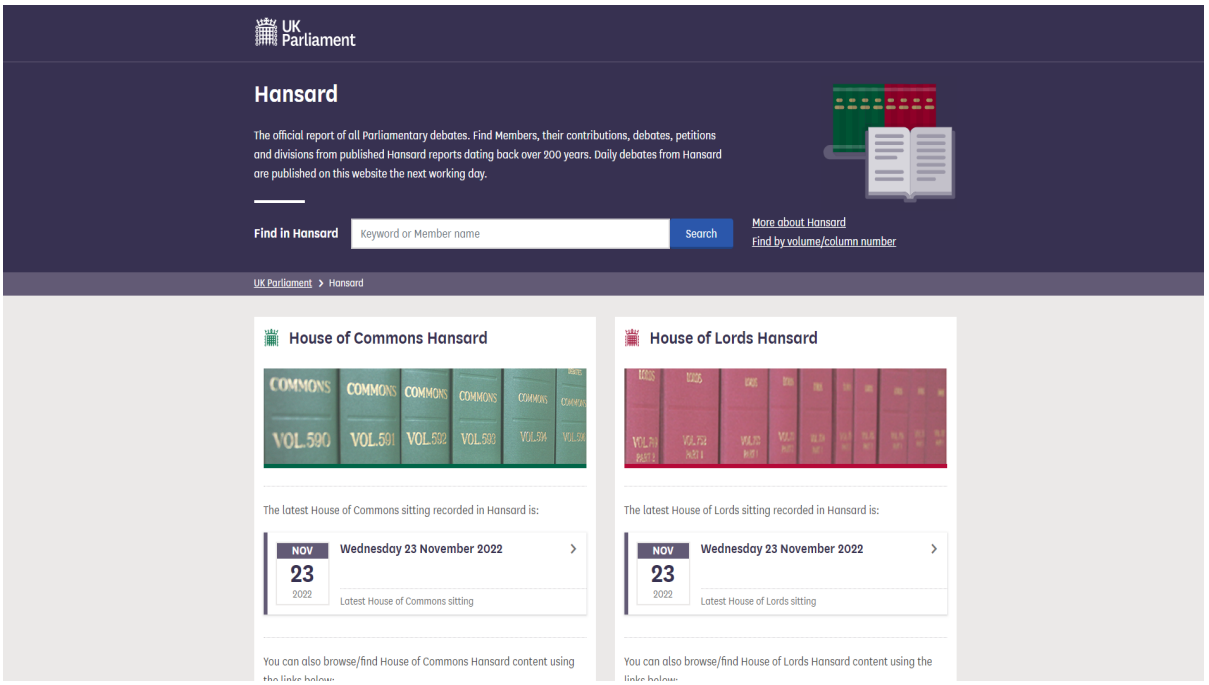
More people said yes than no to this question, which means that the programme should have to include the ability to present word clouds for the most common topics that MPs discuss to make it more relevant for the end user.

Question 8: Do you have any suggestions on how the public could better understand their MP speeches?

Only one person responded to this question. The response was to add a way to search for a particular topic, and see relevant speeches for those topics. The programme has a search function which allows the user to search for speeches containing certain words and topics. When the user searches for a topic, the date of the speech shows up next to that speech, satisfying the user's requests.

Background:

There is already a partial solution to this problem, but it is complex, and not very user friendly. The current solution is the official government Hansard.[2]



The official Government Hansard website

Hansard contains all Parliamentary speeches from around the 1800s, to the present day. The techniques that are used in my program could also be applied to historical data present on Hansard, but I am choosing to focus only on MPs that are currently sitting in the house. This makes it more relevant for the end user, as they would most likely care about sitting MPs. The problem with the official solution is that there are no ways of producing that data in any forms that aren't plain

Output

URL for this: **https://www.theyworkforyou.com/api/getMP?postcode=GL208DF&output=json**


```
{
    "member_id": "42617",
    "house": "1",
    "constituency": "Tewkesbury",
    "party": "Conservative",
    "entered_house": "2019-12-13",
    "left_house": "9999-12-31",
    "entered_reason": "general_election",
    "left_reason": "still_in_office",
    "person_id": "10505",
    "lastupdate": "2019-12-13 05:21:12",
    "title": "",
    "given_name": "Laurence",
    "family_name": "Robertson",
```

text, which makes it hard for the end user to sort through and use all that data, especially if their MP has served in the house for a long time. My project will apply some modern text analysis methods to the Hansard data, which will allow the end user to form a greater understanding of what their MP stands for and represents. The code will display the results of the analysis in the form of plain text, but also in the form of word clouds, graphs, and charts, all to make it easier for the end user to navigate through and understand the data. The programme will also provide an advanced searching capability which will use semantic similarity. The project will be developed using python, using jupyter notebook for prototyping and experimentation, and python flask will be used to create a website to display the final results. The programme will also have to use HTML and Javascript to create the webpage.

Text Analysis:

Text analytics is the process of extracting meaningful insights and information from text data using a combination of natural language processing techniques, and statistical analysis. It is a way to analyse and understand large amounts of unstructured text data, such as social media posts, customer reviews, or survey responses, in order to identify patterns, trends, and relationships that might not be immediately apparent.

Word Clouds

A word cloud is a visual representation of the most frequently used words in a piece of text. It is created by arranging and formatting the words in a way that allows the most frequently used words to stand out.

Word clouds are often used as a way to quickly and visually summarise the main points or themes in a piece of text. They can be used to highlight the most important words or concepts in a document, and to show the relationships between different words and concepts.

The words are then displayed in a visual format, with the size of the word proportional to its frequency in the text. The word cloud is usually arranged so that the most frequently used words are larger and more prominent, while less frequently used words are smaller and less prominent.

The image above is of *Martin Luther King's 'I Have a Dream'* speech, in which the most common words are the bigger blobs on the cloud

Stopwords are common words that are filtered out or removed from text. The words that are filtered are considered to be of little value in terms of conveying meaning or relevance in a given context, and by having them removed, can improve the efficiency of the word cloud, and make it have more meaning.

Lemmatization is the process of reducing words to their base form. The purpose of this is to group together different inflected forms of a word so they can be analysed as a single item. For example, the word 'walked' can be lemmatized to 'walk', walk is the base form.

Semantic Similarity:

Semantic similarity measures the similarity between words, for example, the word 'cat' is similar to the word 'feline', but is not similar to the word 'tractor'. This is useful in the project to see all the words that an MP has said, and filter them by a certain type, like the type of word and emotion of that word (angry, happy etc). This will make it easier to filter the data, and present that data visually. This also makes it easier for the end user to explore all the data.

Embeddings

Embeddings are a type of data representation that are used to capture the relationships between different items in a dataset. Word embeddings are often used to represent words as high-dimensional vectors. A word embedding is a learned representation of a word in the form of a vector, usually with lots of dimensions. The values in the vector capture the context and meaning of the word, based on the other words in the sentence or text. This allows similar words to have similar embeddings. In a similar fashion a sentence embedding is a vector representing a whole sentence and a document embedding represents an entire document.

Topic modelling

Topic modelling automatically identifies topics in a collection of texts. The goal of topic modelling is to extract the themes or topics that are discussed across different text documents. Statistical methods are used to identify common patterns and group the words into topics. Once the topics have been identified, the output can be used to analyse and understand large collections of text, like MP speeches. For example,

using topic modelling, the user can understand how often their MP speaks about climate change. It is also possible to create topics from the embeddings.

List of objectives:

1. Make a webpage, using the TheyWorkForYou API that is better designed and easier to use than the government Hansard website.
- The search function will have a dropdown menu, with a list of MPs, and what constituency they currently serve in.
- When the user selects an MP it comes up with their picture, taken from the TheyWorkForYou API, as well as their name, and when they served in Parliament.
- It also shows their most recent speeches made in parliament, and a way to search for speeches that contain a specific word, or set of words.
2. Explore applying some modern text analysis methods to MP's speeches. This will include:
- Word Clouds
- Semantic similarity search
- Topic modelling
3. Solve the problem of people not knowing what their MP stands for and talks about in Parliament, and getting people to make more informed voting choices
- The webpage will have options to present the data graphically, which is more than what the current solution has, and to make it easier for the end user to understand the data and work through it.
- The webpage will have a nice, user friendly design, that is simple to use and to understand, it will be simple to use and navigate.

The Prototype/Design Phase:

Jupyter Notebook

Jupyter Notebook is a way of developing python code that can be used to combine code, text, visualisations like graphs into a single file. This makes it ideal for the design phase of my project as it can be easily understood, and makes it easy to

experiment with graphical presentations. It also makes it easy to test and debug the code, as you can test line by line, as opposed to testing the whole code at once.

To help my implementation phase, the initial prototyping will be done using Jupyter Notebook. This will allow me to easily explore the options for obtaining and displaying data. In the next section, I will use what I have learned from the prototype, and turn it into a web-based interface.

The Data and The Hansard API

The way the programme accesses the Hansard data is through the Hansard API. An API is a set of rules, protocols, and tools that allow different software systems to communicate with each other. There already is an API that I can use on the TheyWorkForYou[3] website that can be accessed in python using the 'requests' module.
To access the API from Python, I need a library or package that can handle making HTTP requests. One library for this is the "**requests**" library, which can be installed by pip installing. Once you have the library installed, you can use it to make a variety of different types of requests, such as **GET, POST, PUT,** and **DELETE** requests.

Some APIs require authentication to get to work, so a key will have to be used in the header of the requests with a variable to define it. I was able to acquire a free API key from https://www.theyworkforyou.com/ as this project is for educational purposes.

Here is an example of how to access the TheyWorkForYou API to see how many uses of their services you have left per month:

```
import requests
APIkey = 'DN8s9LBm8jMBFZihXEG2gqzx'
def getQuota():
    quotaurl = 'https://www.theyworkforyou.com/api/getQuota'
    response = requests.get(quotaurl, params={'key':APIkey})
    return response.json()['quota']
print(getQuota())
```

The programme is using the requests library to access the Hansard API, and the postocdes_uk library to validate postcodes.

[3] https://www.theyworkforyou.com/api/

The function ***get_MP_info*** will take an input of either a postcode or a constituency, and output the corresponding MP based on that area, and return their information from Hansard. The Hansard API returns its results in JSON format, which is converted into a python dictionary.

```
import requests
import postcodes_uk

def is_valid_postcode(postcode):
    """
    This function takes one argument and returns True or False.

    Args:
    postcode (str): The postcode

    Returns:
    bool: True if a valid postcode, otherwise returns False.
    """
    #making sure the postcode is in uppercase, required for
validation
    postcode = postcode.upper()
    # using the postcodes_uk library to validate the postcode
    ret = postcodes_uk.validate(postcode)
    return ret

def get_MP_info(postcode_or_constituency):
    """
    This function takes one argument, either the postcode or the
constituency name, and returns the MP information from Hansard.

    Args:
    postcode_or_constituency (str): Postcode or a Constituency

    Returns:
    dict: The MP information from Hansard
    """
    # checks to see in the input is a valid postcode
    params = {'key':APIkey}
    if is_valid_postcode(postcode_or_constituency):
        params['postcode'] = postcode_or_constituency
    else:
```

```
        params['constituency'] = postcode_or_constituency

    url = 'https://www.theyworkforyou.com/api/getMP'
    response = requests.get(url, params=params)
    return response.json()
```

As an example, we can get the information for the MP for Tewkesbury as follows:

```
get_MP_info('Tewkesbury')
```

```
{'member_id': '42617',
 'house': '1',
 'constituency': 'Tewkesbury',
 'party': 'Conservative',
 'entered_house': '2019-12-13',
 'left_house': '9999-12-31',
 'entered_reason': 'general_election',
 'left_reason': 'still_in_office',
 'person_id': '10505',
 'lastupdate': '2019-12-13 05:21:12',
 'title': '',
 'given_name': 'Laurence',
 'family_name': 'Robertson',
 'full_name': 'Laurence Robertson',
 'url': '/mp/10505/laurence_robertson/tewkesbury',
 'image': '/people-images/mpsL/10505.jpg',
 'image_height': 160,
 'image_width': 120,
 'office': [{'moffice_id':
'uk.parliament.data/Member/253/Committee/11/1',
   'dept': 'Panel of Chairs',
   'position': 'Member',
   'from_date': '2020-01-15',
   'to_date': '9999-12-31',
   'person': '10505',
   'source': ''}]}
```

Typing in a constituency, or a postcode gets us the same results either way, but inputting a postcode is easier. We can achieve the same results by running:

```
get_MP_info('GL208DF')
```

Allowing the user to enter a postcode, instead of a constituency is easier as constituency names are often long and hard to remember, and also hard to spell as

the program requires you to spell it perfectly, for example ‘Bermondsey and Old Southwark’.

```
def get_Hansard(person_id):
    """
    This function takes one argument and returns the MP's person
ID from Hansard.

    Args:
    personID (int): gets the relevant data from Hansard

    Returns:
    int: The information from Hansard about the person_id
    """
    params = {'key':APIkey}
    params['person'] = person_id
    url = 'https://www.theyworkforyou.com/api/getHansard'
    response = requests.get(url, params=params)
    data = response.json()
    params['page'] = 1
    # loops until we have all the pages
    while True:
        params['page'] += 1
        response = requests.get(url, params=params)
        data0 = response.json()
        if len(data0['rows']) == 0:
            break
        data['rows'].extend(data0['rows'])
    return data
```

This function returns all the speeches for the relevant MP. The API will only allow us to download a small number of speeches, so it loops through pages until all the speeches have been obtained.

The data is returned as a python dictionary. The value of **data[‘rows’]** is a list, and each entry of the list corresponds to one speech, and is itself a python dictionary. The actual words of the speech are in **data[‘rows’][i][‘body’]**, and are in HTML.

A library called ‘**beautifulsoup**’ can be used to parse the HTML, and convert that into plain text.

```
from bs4 import BeautifulSoup
```

```
BeautifulSoup(data['rows'][0]['body'], "html.parser").text
```

Word Clouds

As an example, I have selected 3 MPs who are all from different political parties.

1. Lawrence Robertson, Tewkesbury, Conservative Party. Tewkesbury is a rural constituency in Gloucestershire.
2. Lucy Powell, Manchester Central, Labour Party. This is an inner city constituency in the north west of England
3. Caroline Lucas, Brighton, Green Party. She is the only Green MP.

I have chosen these three MPs because they all represent a diverse range of political parties and constituencies. We will expect these differences to show up in their word cloud, as they all speak about different topics, being from such different constituencies with different demands.

Word clouds are generated from frequency counts of words. Before counting the number of words, we need to preprocess the speeches. We perform the following steps in a list of speeches:

1. Concatenate the speeches into one string of text
2. Remove all punctuation
3. Remove everything inside of parenthesis. This is a feature of Hansard where if an MP refers to another member of the house, their name is added in parenthesis. This will remove examples such as '(a)' or '(b)' if a list is present in the data.
4. Convert all the text into lowercase
5. Split the text by whitespace to create a working list of words
6. Lemmatize the words, meaning reducing each word to its base form. For example, 'running', 'ran', and 'runs' are all reduced to 'run' because run is the root.

```
import string
from nltk.stem import WordNetLemmatizer
import re
def preprocess_speeches(speeches):
    #Concatenates the speeches into one string of text
    text = '\n'.join(speeches)
    #Removes all punctuation
    text = text.translate(str.maketrans('', '',
string.punctuation))
    #Removes text inside parenthesis
    text = re.sub(r'\d+', '', text)
```

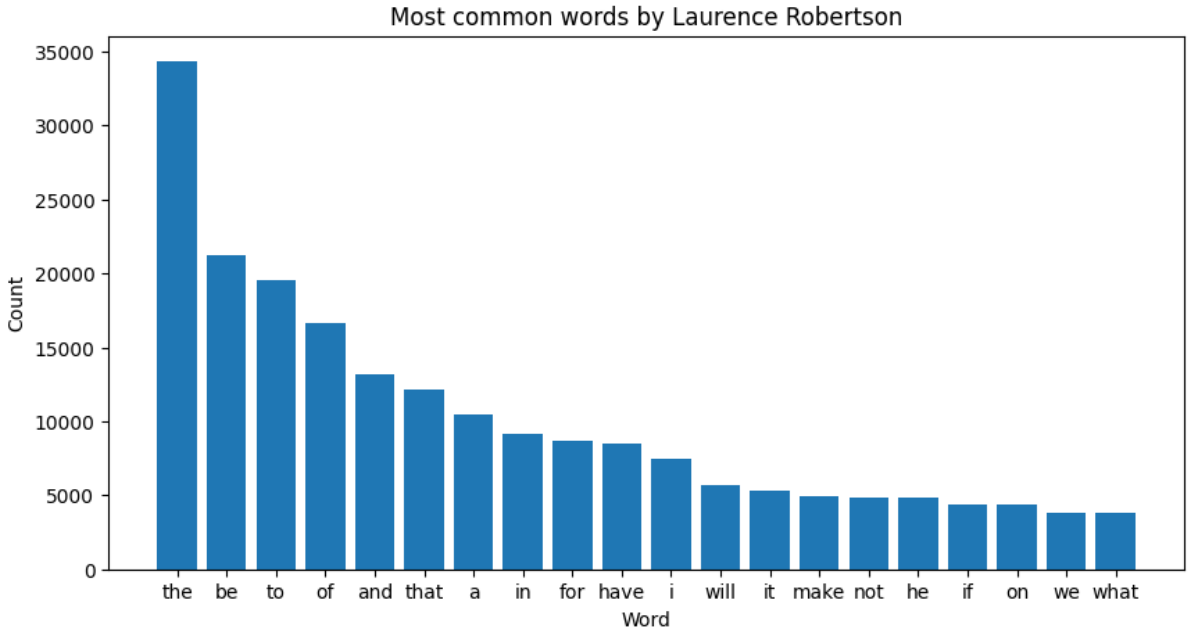
```
    #Converts text into lowercase, and splits by whitespace
    words = text.lower().split()
    #Reduce each word to its root
    lemmatizer = WordNetLemmatizer()
    words = [lemmatizer.lemmatize(w, pos='v') for w in words]
    return words
```

In the code snippet above, it gives the python function for preprocessing words.

Given the list of words, it is easy to generate a frequency count.

```
from collections import Counter
def frequency_count(words):
    words_counter = Counter(words)
    return words_counter
```

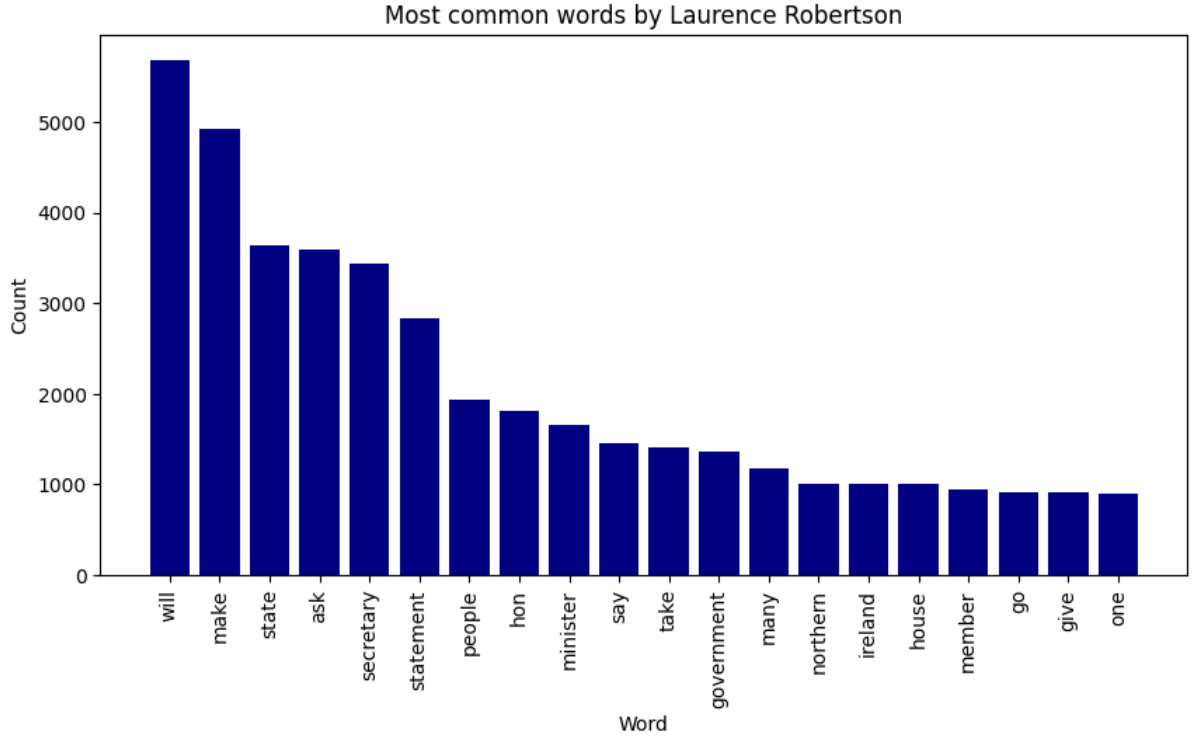
This function returns a counter object. From this, we can extract the most common words. In the image below, we have charted the most common words said by Laurence Robertson. As you can see, all of the words have very little meaning in the context of speeches in Parliament. These words are known as stopwords, and must be removed from the text.

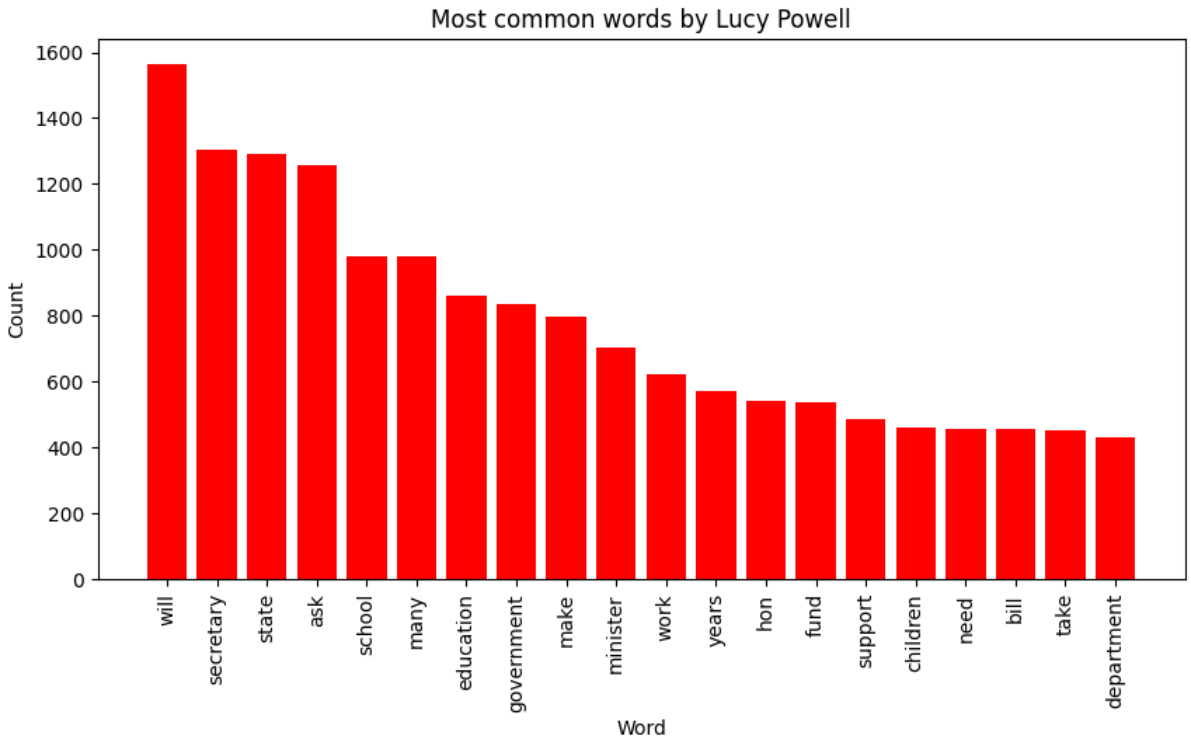


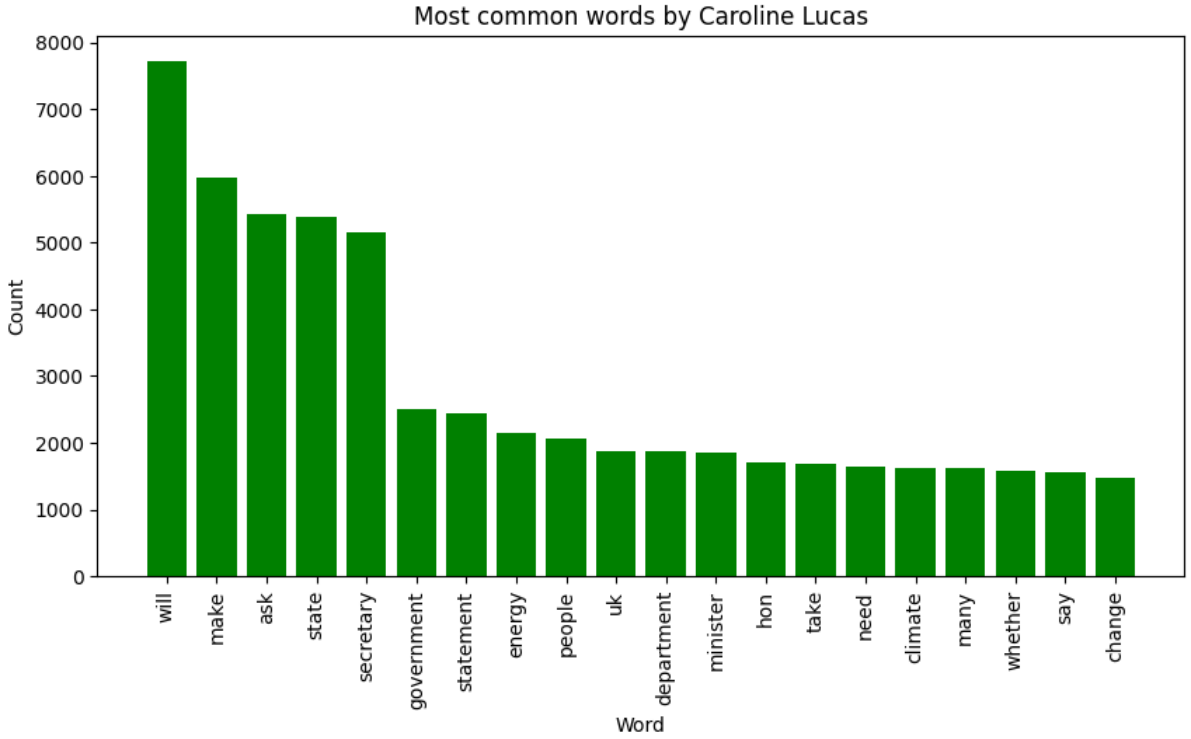
I will modify the frequency_count function to take a list of stopwords as an input, and remove those words from the frequency count. The python wordcloud package comes with a list of stopwords.

```
def frequency_count(words, stopwords = []):
    words_counter = Counter([w for w in words if not w in
stopwords])
    return words_counter
```

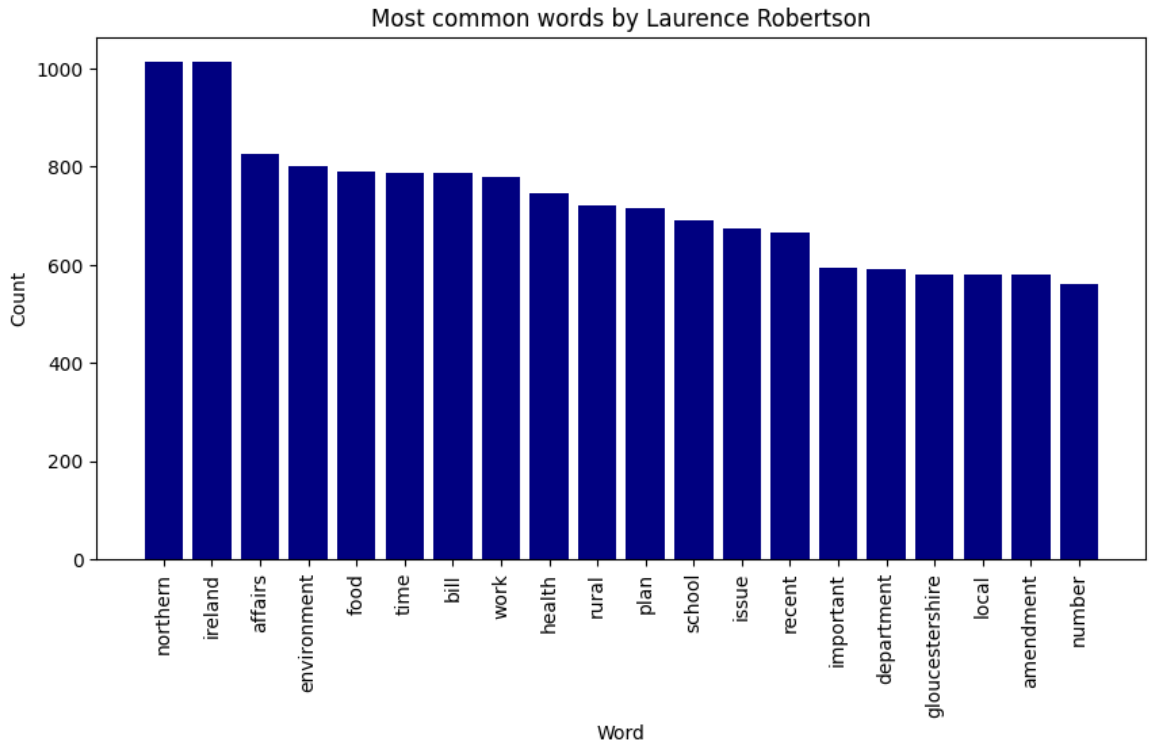
In the charts below, we can see the effects of introducing stopwords into the frequency count. Even though the list of stopwords is good at getting rid of some words, it is insufficient at getting rid of all the meaningless words. I have generated one graph for each of the three MPs.







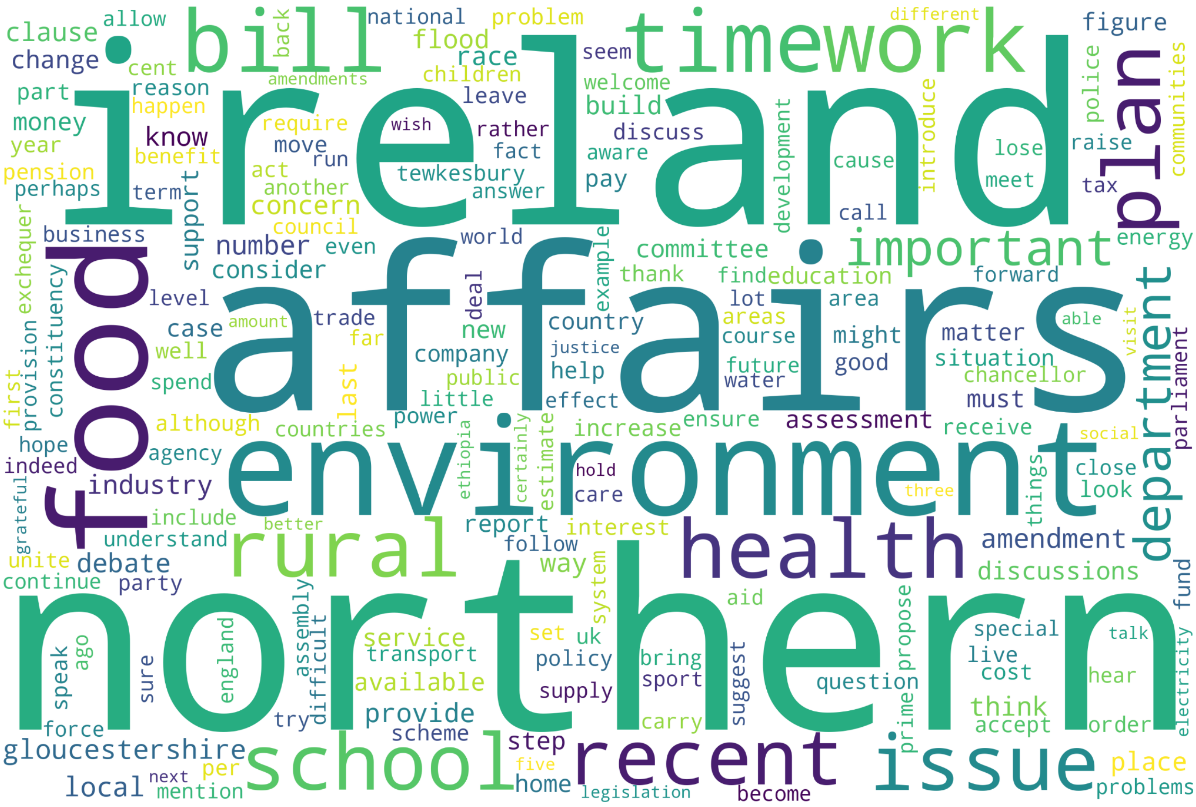
While these graphs are an improvement over the previous one, they are still not perfect. We still see lots of Parliamentary language like ‘secretary’ or ‘will’, which are not meaningful. This is why the code still needs to introduce an extended list of stopwords, that include these words, among others. The image below is an example of the new chart that excluded these words. As you can see, these words have more meaning, and the list is not full of words that are in the list of stopwords.



This is a jupyter notebook cell to produce a word cloud with the 250 most common words spoken by Laurence Robertson. The stopwords have already been removed so do not appear in word_freqs.

```
wordcloud = WordCloud()
wordcloud = WordCloud(width = 3000, height =
2000,background_color='white')
wordcloud.generate_from_frequencies({x[0] : x[1] for x in
word_freqs.most_common(250)})
plt.figure(figsize=(40, 30))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```

This code produces this:



This is quite good, but there is one major issue remaining. The words 'northern' and 'ireland' are said together in many speeches to make 'northern ireland'. This is an example of a common bigram.

Bigrams:

Bigrams are pairs of words that appear consecutively in a text. Bigrams are used to extract meaningful information from text data. If we include both bigrams and individual words, we can capture some of the context in which the words appear and get a better understanding of the overall meaning of some speeches. Some examples might include 'northern Ireland', and 'climate change'.

In the code below, I have modified the preprocess_speeches function to extract both words and bigrams from the speeches.

```
from nltk.util import ngrams
def preprocess_speeches(speeches, bigrams=True):
    #Concatenates the speeches into one string of text
    text = '\n'.join(speeches)
    #Removes all punctuation
```

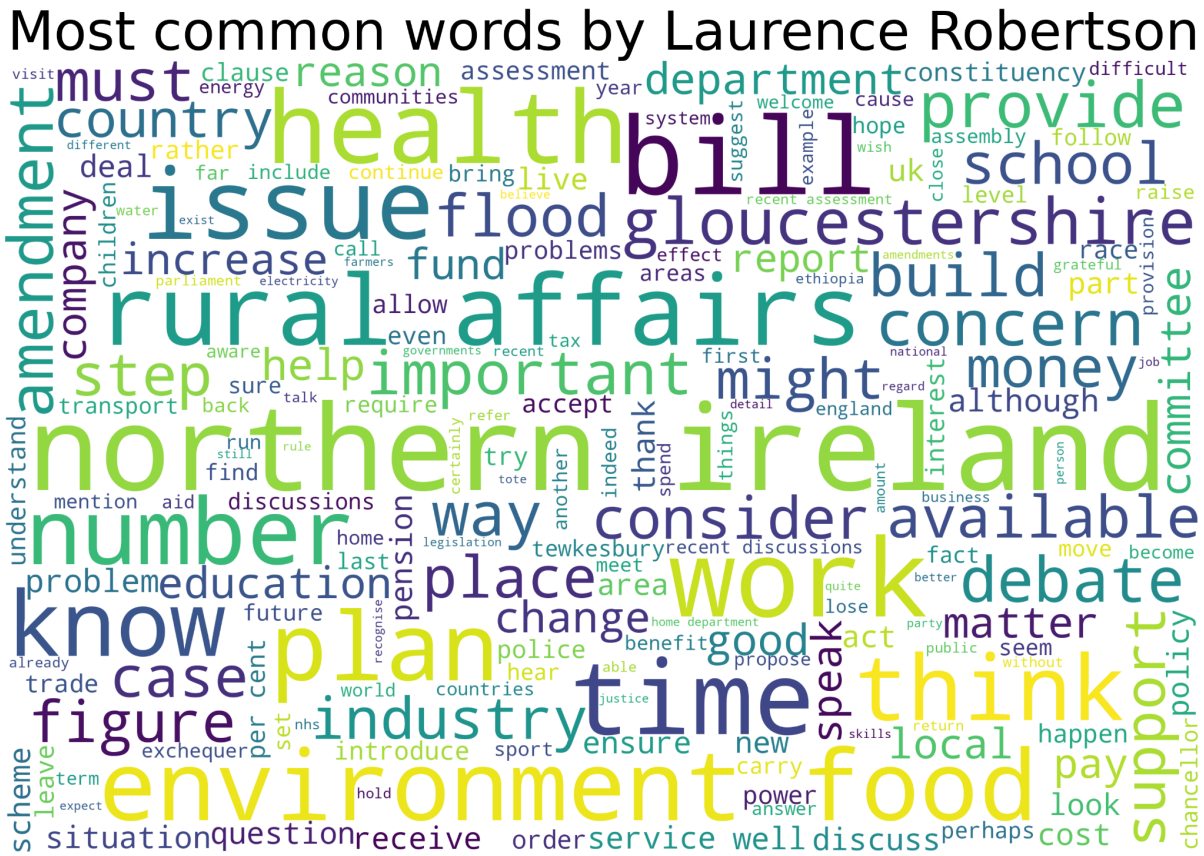
```
    text = text.translate(str.maketrans('', '',
string.punctuation))
    #Removes text inside parenthesis
    text = re.sub(r'\d+', '', text)
    #Converts text into lowercase, and splits by whitespace
    words = text.lower().split()
    words = [w for w in words if len(w) > 1]
    lemmatizer = WordNetLemmatizer()
    #Reduce each word to its root
    words = [lemmatizer.lemmatize(w, pos='v') for w in words]
    if bigrams == False:
        return words
    bigrams = ngrams(words, 2)
    return words, bigrams
```

In this code, we have counted the number of words, and the number of bigrams, and gets a dictionary with the 250 most common of both.

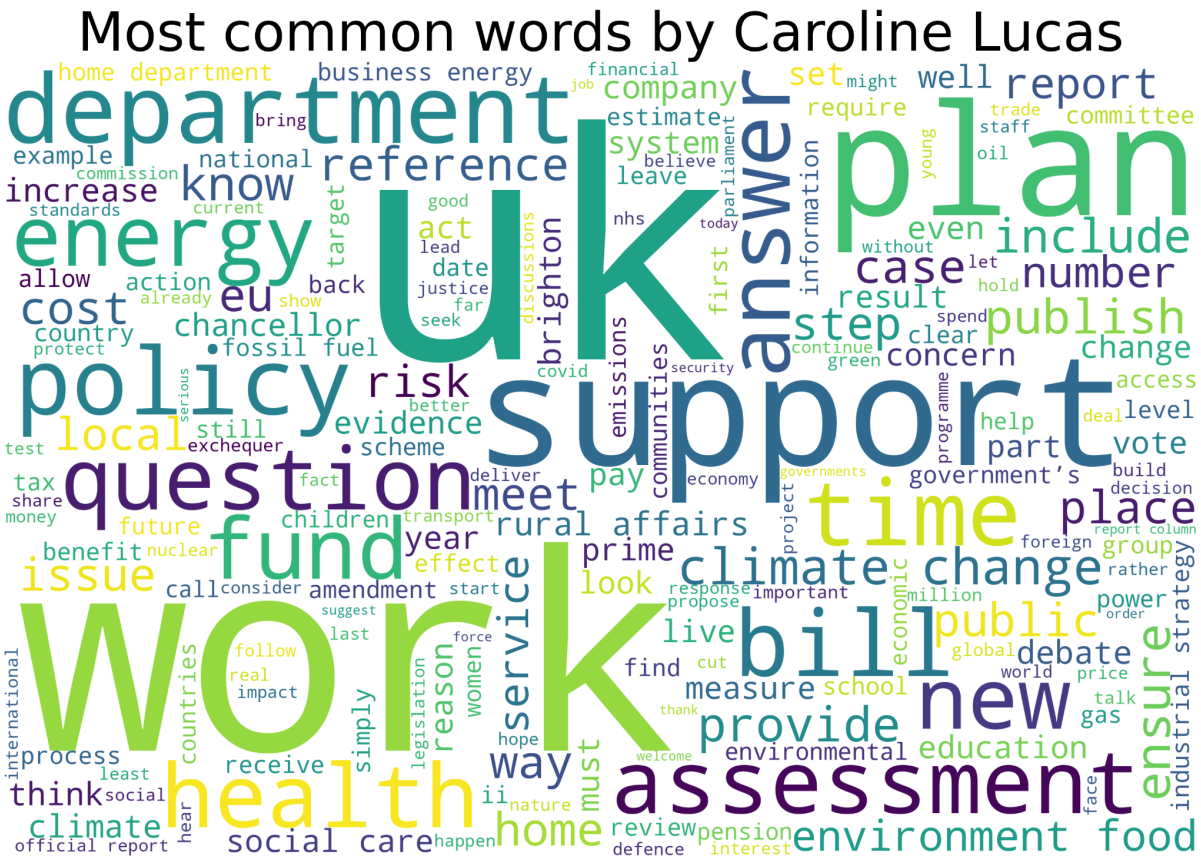
```
def bigrams_frequency_count(words, bigrams, stopwords = []):
    #Counts words as before, excluding stopwords
    words_counter = Counter([w for w in words if not w in
stopwords])
    #Counts bigrams, excluding any that are in the list of
stopwords
    bigrams_counter = Counter([' '.join(w) for w in bigrams if not
w[0] in stopwords and not w[1] in stopwords])
    #Gets a dictionary of the 250 most common words
    words_counter = words_counter.most_common(250)
    words_counter = {w[0] : w[1] for w in words_counter}
    #Gets a dictionary with the 250 most common bigrams
    bigrams_counter = bigrams_counter.most_common(250)
    bigrams_counter = {w[0] : w[1] for w in bigrams_counter}
    #Removes counts from single words, if they appear in the list
of bigrams
    for w in words_counter.keys():
        for b in bigrams_counter.keys():
            bg = b.split()
            if w == bg[0] or w == bg[1]:
                words_counter[w] -= bigrams_counter[b]
    return Counter(words_counter) + Counter(bigrams_counter)
```

The resulting code produces this:

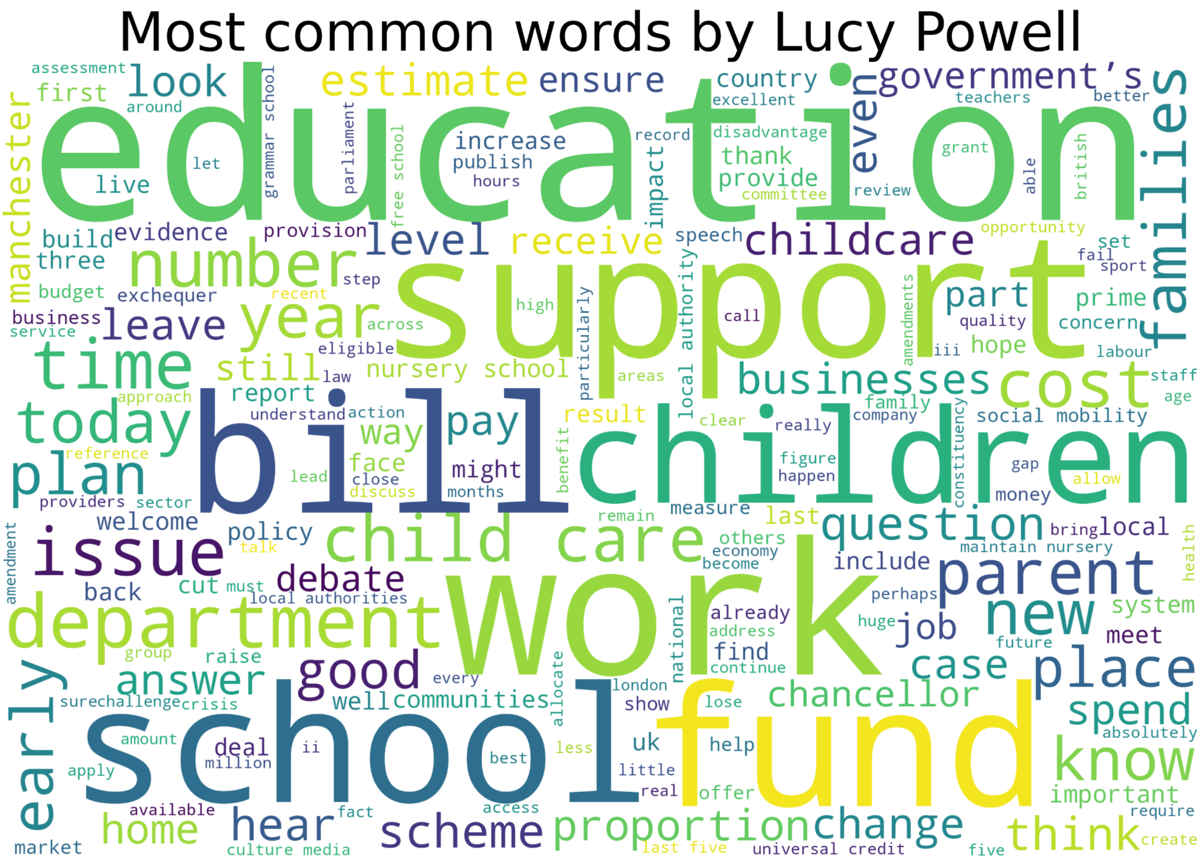
Below are the three MPs with their new word clouds with bigrams included:



In Laurence Robertson's word cloud, we can see a strong interest in Northern Ireland, rural affairs, and money. He was previously the shadow minister for Northern Ireland, and the shadow minister for trade and industry.



In Caroline Lucas' word cloud, we can see a strong interest in work, climate change, energy usage, and the health of the nation. She was previously on the Environment Audit Committee, so her interest reflects on the word cloud.

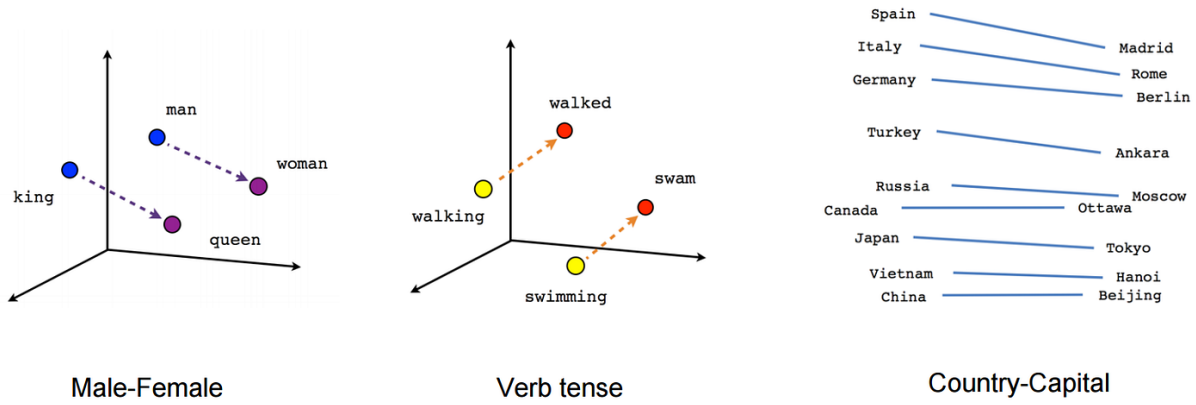


In Lucy Powell's word cloud, we can see a strong interest in schools, children, and education. She has been the shadow minister for education twice, and is currently on the education committee.

Vector Embeddings

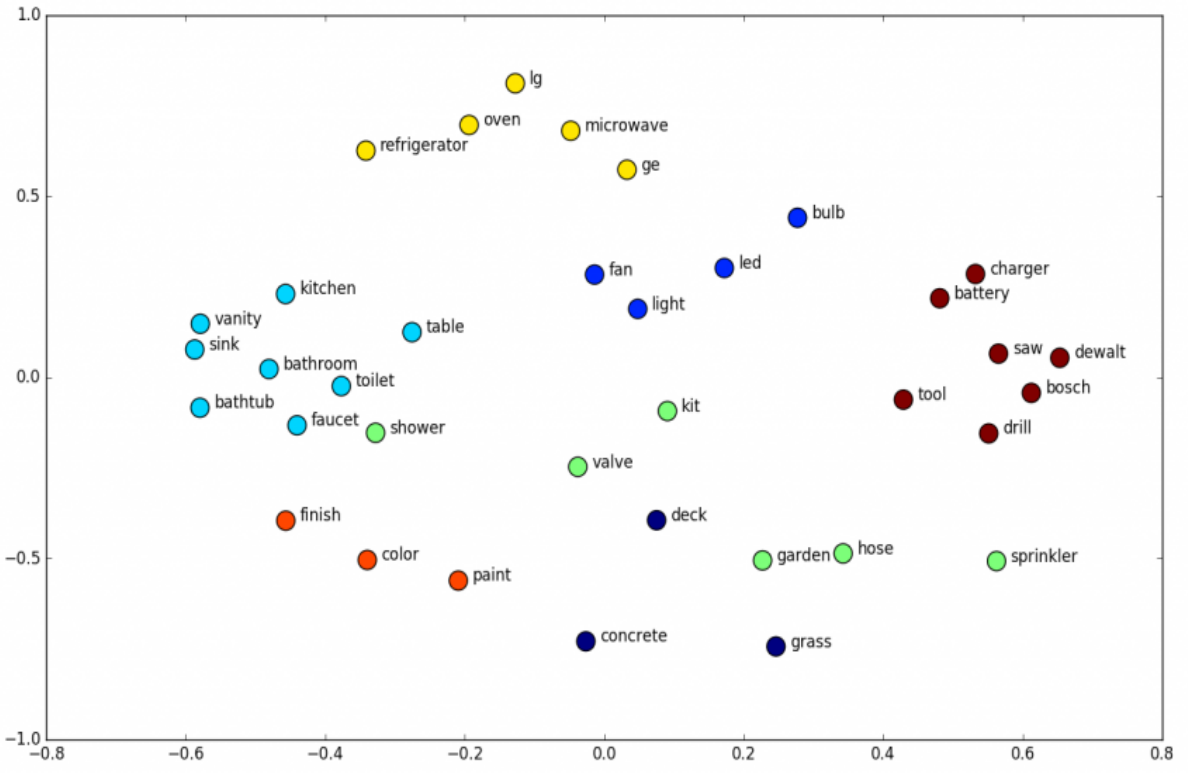
An embedding is a mapping from words to a high-dimensional vector space. An embedding captures the semantic and syntactic relationships between words in a way that allows the natural language of text to be processed more effectively.

Each word in a given text is assigned a unique vector. The vectors are learned by training a machine learning model on a large text base.



Male-Female Verb tense Country-Capital

How are vectors compared?



Vectors can be compared using cosine similarity. This is the cosine of the angle between vectors. This is calculated by taking the dot product of the vectors and dividing it by the product of their magnitudes. The values can range from -1, meaning it is totally dissimilar, to 1, meaning it is totally similar. 0 Means there is no similarity.

NLTK - Natural Language Toolkit

NLTK is a python module for working with language data. It provides a wide range of tools and resources for language processing, including machine learning. It contains pretrained word embeddings that can be accessed as follows.

```
from nltk.data import find
import gensim
#reading in vectors database
word2vec_sample =
str(find('models/word2vec_sample/pruned.word2vec.txt'))
model =
gensim.models.KeyedVectors.load_word2vec_format(word2vec_sample,
binary=False)
```

The model is a mapping from words to 300-dimensional vectors. This is used to embed sentences as follows.

```
def sentence2vec(sentence, model, stopwords):
    #maps every words to a vector, sums the vectors, then
normalises
    v = model['house'] - model['house']
    cnt = 0
    for w in sentence.split(' '):
        if model.has_index_for(w) and not w in stopwords:
            v += model.get_vector(w)
            cnt += 1
    if cnt > 0:
        v /= norm(v)
    return v
```

An alternative approach is the SentenceTransformers Python library for generating sentence embeddings using a pre-trained transformer like BERT. It provides an easy to use interface for encoding text data into dense vector representations.

The library contains a large number of embedding models, shown here: https://www.sbert.net/docs/pretrained_models.html. I chose to use the model “average_word_embeddings_komninos”, as that gave the best trade off between speed, memory, and quality. Many of the other models are neural networks, which would require an expensive GPU. This is used as below.

```
from sentence_transformers import SentenceTransformer
```

```
sbert_model =
SentenceTransformer('average_word_embeddings_komninos')

def sentence2vec(sentence, model, stopwords):
    text = " ".join(x for x in sentence.lower().split() if not x
in stopwords)
    return model.encode(text)

text_embeddings = sentence2vec(sentence, sbert_model, stopwords)
```

I chose to use BERT as this was a simpler solution and was more compatible with Topic Modelling.

Semantic similarity

Below are the functions required for computing the cosine similarity:

```
import numpy as np
import spacial

def norm(vec):
    #returns the magnitude of the vector
    return np.linalg.norm(vec)

def similarity(vec1, vec2):
    #returns the cosine similarity of the vector
    if norm(vec1) < 0.1 or norm(vec2) < 0.1:
        return 0.0
    return 1.0 - spatial.distance.cosine(vec1, vec2)
```

In the code below, we have created a search function, that allows the user to find speeches by one MP that contain a certain word, or something similar in context to that word.

- The first line creates a vector representation of the input text using a pre-trained model and a set of stopwords. The resulting 'vectors' variable is a list of vectors, where each vector represents a sentence or a document.
- The code then defines a function 'search', that takes in four arguments 'words', 'model', 'vectors', and 'stopwords'. The function creates a vector representation 'v' of the query using the 'sentence2vec' function.

- The function calculates the similarity scores between the query vector ‘v’ and all the vectors in ‘vectors’ using the ‘similarity’ function.
- Finally, the function returns the indices of the 16 most similar vectors in ‘vectors’ based on their similarity scores. The indices are sorted in descending order using the ‘np.argsort(scores)[::-1]’ and the top 16 indices are selected using ‘[:16]’
- The code calls the search function with the query “‘horse riding”’. Using the pre-trained ‘model’, the list of vectors ‘vectors’, and the set of ‘stopwords’, it then prints the resulting ‘idxs’ variable, which is a list of the top 16 most similar vector indices in ‘vectors’ to the query “‘yes”’

```
def search(words, model, vectors, stopwords):
    v = sentence2vec(words, model, stopwords)
    scores = [similarity(v,dv) for dv in vectors]
    idxs = np.argsort(scores)[::-1][:16]
    return idxs

idxs = search("horse riding", model, vectors, stopwords)
print(idxs)
```

For example, we have used the phrase ‘horse riding’. The search will now find the top 16 most similar speeches that are semantically similar to “horse riding”. The most similar speech is this one:

'The Minister will be aware that as more and more building takes place in villages, more traffic is put on the road, which presents a danger to horse riders. Just last year alone, 46 horses were killed and 130 riders were injured. One way in which more access could be provided is by allowing horse riders to use footpaths, for example, and there are many other ideas. Will she work with me and others who are concerned about this issue to try to improve access to bridleways for horse riders?'

Another example is the phrase ‘horse racing’. This should produce different results, as the phrase has a different meaning to the previous phrase. This is the most similar speech that includes the phrase ‘horse racing’:

'What progress she has made on replacing the horserace betting levy as a means of funding horse racing; and if she will make a statement.'

We can also search for words that are not directly found in the text. You can search for almost anything, and a speech that relates to that word or phrase comes up. For

example, the input to get this speech was ‘horseback’. The word horseback was not said in this speech, but the search found something similar and related to the topic:

'The Minister says that I did not. I discussed at great length in Committee my problems with another matter in which I have a constituency interest—the horse racing industry. As we approach the great week at Cheltenham, it is probably appropriate to remind the House that the way in which the OFT was looking at the horse racing industry was very clumsy.'

BERTopic

BERTopic is a topic modelling technique that is based on the BERT language model. It uses BERT to encode the text in a corpus into high-dimensional vectors. These vectors are transformed into lower dimensional vectors using UMAP and then clustering techniques are applied to find clusters which are topics.

The resulting clusters are then assigned a topic label based on the most representative words in each cluster. BERTopic does not require a fixed number of topics to be defined in advance, unlike traditional topic modelling techniques such as Latent Dirichlet Allocation. Instead, it uses a technique called topic coherence to determine the optimal number of topics based on the quality of the topics generated.

The code to find topics is below.

```
from bertopic import BERTopic
from umap import UMAP
# Initiate UMAP
umap_model = UMAP(n_neighbors=15,
                  n_components=5,
                  min_dist=0.0,
                  metric='cosine',
                  random_state=100)# Initiate BERTopic
topic_model = BERTopic(umap_model=umap_model, language="english",
calculate_probabilities=True, nr_topics=11)
# Run BERTopic model
text = preprocess_speeches(speeches, stopwords = stopwords,
min_length = 25)
topics, probabilities = topic_model.fit_transform(text)
topic_model.get_topic_info()
```

The UMAP line initialises a UMAP model with 15 nearest neighbours, 5 components, a minimum distance of 0.0, and the cosine similarity metric. 'random_state=100' sets a fixed seed for random number generation, ensuring reproducibility of the results.

The BERTopic line initialises a BERTopic model, passing in the 'umap_model' object created earlier, setting the language to English, enabling probability calculation, and specifying the number of topics to be 11.

The next line preprocesses the 'speeches' text data using a function called 'preprocess_speeches'. The function takes in the speeches, a list of stopwords, and a minimum length for the words to keep. The output of the function is assigned to a variable called 'text'.

The next line fits the 'text' data to the 'topic_model' object and returns the topics and their corresponding probabilities. The 'topics' variable contains the top topic per document, while 'probabilities' contains the probability score of that topic.

The final line retrieves the topic information from the 'topic_model' object, such as the most representative words for each topic and their corresponding probabilities. It returns a dataframe with this information.

The resulting code produces this:

	Topic	Count	Name
0	-1	122	-1_drugs_public_crime_important
1	0	73	0_racing_tote_race_sport
2	1	57	1_schools_school_special_education
3	2	219	2_houses_tewkesbury_flood_areas
4	3	71	3_nuclear_industry_energy_gas
5	4	11	4_manufacturing_industry_companies_goods
6	5	19	5_pubs_pub_smoking_smoke
7	6	74	6_ethiopia_aid_country_world
8	7	574	7_ireland_northern_bill_amendment

The image above shows a list of topics, and how many times a word relevant to a topic was said. Topic -1 consists of all documents that are not assigned to a topic, whereas topics 1-8 are the topics found by BERT. For example, Laurence Robertson

uses words relevant to Northern Ireland 574 times, and words relevant to Tewkesbury 219 times, which gives the user a picture about how much their MP talks about certain topics.

Implementation

Webpage and GUI

The webpage will have the following features:

- A scrollable list of all the constituencies in the UK, on the left side of the screen, and allowing the user to search through them, to find the constituency they need
- When a user selects a constituency, the MP's picture will be displayed above the list of constituencies, and to the left of the tabs, which display the word clouds and the speeches
- The page will have tabs for the following:
    - Word clouds, which display a selection of relevant topics that the searched MP discusses
        - There will be up to 8 word clouds, one for each of the relevant topics that the chosen MP discusses, all arranged in a grid. When a user click on a word cloud, the list of speeches for that topic will be displayed, along with the date of when that speech was said
    - Topic modelling, to find words and phrases of a similar topic, and display them using bar charts
    - A search function to find speeches that contain certain words or phrases

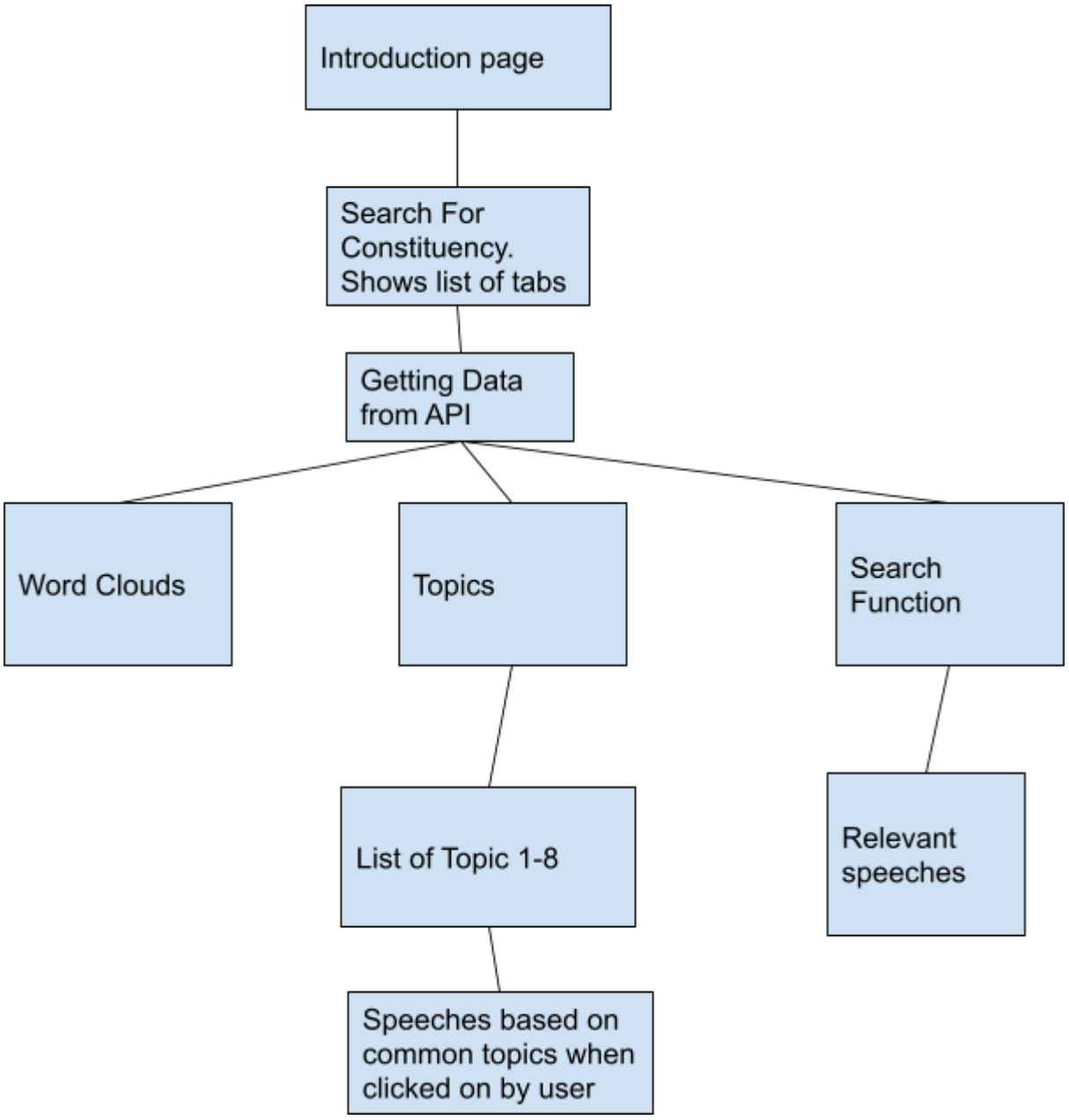
The grid for the word clouds will look something like this:

Topic 1	Topic 2	Topic 3	Topic 4
Topic 5	Topic 6	Topic 7	Topic 8

When the user clicks on a topic, a selection of speeches will be presented underneath the grid, as well as enlarging the word cloud to be easier to read. For example, if Topic 1 was about drugs, speeches about that selected topic will be shown, as well as the word cloud for that chosen topic.

The search function will take an input from the user, and present speeches that contain the word they searched, or topics that have a similar semantic similarity to that word, if the word searched for is not present in the data.

Overview of the Project



The project flow will look something like this, with the introduction page leading into the first search function. This will allow the user to pick and choose what section of the programme they want to explore next, with word clouds, topics and the speech search function. Each tab does something different and offers the user different ways to find out about their chosen MP. The topics tab and the search tab each do more when interacted with. The topics tab displays a list of relevant speeches based on

the common topics that the MP talks about. While the search tab displays relevant speeches based on what the user inputted.

The web page will be developed using Flask, which is a lightweight web application framework for Python. It's designed to be simple, flexible, and is a popular choice for developing web applications, and using APIs. Flask provides a simple way to incorporate APIs using request handling, allowing it to process the data sent by the client.

The Code

The main routine to emit a webpage is below. This takes as input the MP, data for the wordcloud and data from the form on the webpage. The routine determines if the user has selected an MP. If so it sends MP data to the webpage for rendering. If not it renders instructions on how to use the page.

```
@bp.route('/', methods=('GET', 'POST'))
def main(selected_constituency=None, MP=None, wordclouddata=None,
form=None):
    """
    Renders the main page with optional parameters for
selected_constituency, MP, wordclouddata, and form.
    """
    global constituencies
    log.info('Working')
    display_tab = 'wordcloud'
    if constituencies is None:
        log.info('Requesting constituencies')
        constituencies = getConstituencies()
        log.info('Completed requesting constituencies')
        instructions = True
    else:
        instructions = False
    if not form is None and form.validate_on_submit():
        searchTerm = form.searchTerm.data
        display_tab = 'search'
    else:
        searchTerm = None
    most_similar = None
    if MP is None:
        instructions = True
```

```
        representative_docs = None
        topicsData = None
        SimpleMP = None
    else:
        representative_docs = MP.representative_docs
        topicsData = [[{'word': w[0], 'value': 1.0} for w in
MP.topic_model.get_topic(
            i)] for i in MP.topic_model.get_topics().keys() if i
!= -1]
        SimpleMP = HansardSimpleMP(MP)
        selected_constituency = MP.constituency
        instructions = False
        if not searchTerm is None:
            most_similar = MP.find_most_similar(searchTerm)
    if instructions:
        log.info('Rendering initial page with instructions')
    else:
        log.info('Rendering template')
    log.info('Using %.2f %% of memory',
psutil.virtual_memory().percent)
    ret = render_template('main/main.html',
                          constituencies=constituencies,

selected_constituency=selected_constituency,
                          MP=SimpleMP,
                          wordclouddata=wordclouddata, form=form,
most_similar=most_similar,
                          display_tab=display_tab,
topics_data=topicsData,

representative_docs=representative_docs,
instructions=instructions)
    log.info('Completed rendering template')
    return ret
```

The code below deals with the search requests for semantic similarity search or when the user selects a constituency.

```
@bp.route('/search', methods=('GET', 'POST'))
def search():
    """
```

```
    Performs a search based on user-selected constituency,
retrieves word cloud data, and returns the main page.
    """
    constituency = None
    select = request.form.get('constituency')
    log.info("select=%s", select)
    if select == 'Select constituency':
        return redirect(url_for('main.dropdown'))
    if select is None:
        log.info('Getting MP from session. SELECT IS NONE')
        MP = session["MP"]
        constituency = session["constituency"]
    else:
        constituency = str(select)
        if constituency == session.get("constituency", None):
            log.info('Getting MP from session. CONSTITUENCY
UNCHANGED')
            MP = session["MP"]
        else:
            log.info('GETTING MP FROM HANSARD')
            MP = HansardMP(constituency)
            session["MP"] = MP
            session["constituency"] = constituency

    form = SearchTermForm()
    log.info('Getting word cloud frequencies for %s',
MP.full_name)
    freqs = MP.wordcloud_freqs
    log.info('Completed getting wordcloud frequencies for %s',
MP.full_name)
    wordclouddata = [{'word': x[0], 'value': x[1]}
                     for x in freqs.most_common(64)]
    wordclouddata.sort(key=lambda x: x['value'], reverse=True)
    return main(selected_constituency=constituency,
                    MP=MP, wordclouddata=wordclouddata, form=form)
```

The Flask routes index and main are used to render the main page of the web application and perform a search based on user-selected constituencies. The index route redirects to the main.main endpoint, while the main route renders the main page with optional parameters for selected_constituency, MP, wordclouddata, and form. The main route first sets the display_tab variable to 'wordcloud'. If the

constituencies global variable is None, the function calls the getConstituencies function to retrieve a list of constituencies. The function then checks if the form argument is not None and if it has been validated. If the form has been validated, the function sets the searchTerm variable to the value of the searchTerm field in the form. Otherwise, the searchTerm variable is set to None. The function then sets the most_similar variable to None. If the MP argument is None, the instructions variable is set to True, and the representative_docs, topicsData, and SimpleMP variables are set to None. Otherwise, the instructions variable is set to False, and the representative_docs, topicsData, and SimpleMP variables are set to the appropriate values. If the searchTerm variable is not None, the function calls the find_most_similar method of the MP object to find the most similar document to the search term. Finally, the function renders the main/main.html template with the appropriate variables.

The search route is used to perform a search based on a user-selected constituency, retrieve wordcloud data, and return the main page. The function first sets the constituency variable to None. If the user has not selected a constituency, the function redirects to the main.dropdown endpoint. If the user has selected a constituency, the function checks if the selected constituency is the same as the constituency stored in the session. If it is, the function retrieves the MP object from the session. Otherwise, the function creates a new HansardMP object for the selected constituency and stores it in the session. The function then creates a SearchTermForm object and retrieves the wordcloud frequencies for the selected constituency. Finally, the function renders the main route with the appropriate variables.

The HansardMP Class

The HansardMP class contains all the information we require about an MP. It has several class methods that call out to the HansardAPI to get the data or BERT to get embeddings, create topic models or run semantic search.

```
class HansardMP:
    """
    Gets and stores all the info on the MP
    """
    def __init__(self, postcode_or_constituency, minLength=25):
        """
        Initialises the info about the chosen MP
        """
        self.info = getMP(postcode_or_constituency)
        self.minLength = minLength
        self._wordcloud_freqs = None
```

```
        self._speeches = None
        self.stopwords = get_stopwords()
        self._embeddings = None
        self._topic_model = None
        self._representative_docs = None
        self._sentenceTransformer = HansardSentenceTransformer
        log.info('Completed MP initialisation for %s',
self.full_name)

    @property
    def sentenceTransformer(self):
        """
        Returns the function that converts text into vectors
        """
        return self._sentenceTransformer

    @property
    def constituency(self):
        """
        Returns the constituency chosen by the user
        """
        return self.info['constituency']

    @property
    def person_id(self):
        """
        Returns the ID from Hansard for the chosen MP
        """
        return self.info['person_id']

    @property
    def party(self):
        """
        Returns the political party for the chosen MP
        """
        return self.info['party']

    @property
    def image(self):
        """
        Returns the URL of the image of the chosen MP
        """
```

```
        return 'https://www.theyworkforyou.com' +
self.info['image']

    @property
    def full_name(self):
        """
        Returns the full name of the chosen MP
        """
        return self.info['full_name']
```

The HansardMP class is used to get and store information about MPs. The class takes a postcode or constituency as an argument, and has several properties that return information about the chosen MP, such as their constituency, political party, and image URL.

The class has an __init__ method that initialises the info attribute with the MP's information obtained from the getMP function.

The minLength argument is an optional parameter that sets the minimum length of speeches to be considered for the analysis.

The class also initialises several other attributes, such as **_wordcloud_freqs**, **_speeches**, **_embeddings**, **_topic_model**, and **_representative_docs**, which are used to store information about the MP.

The class has several properties that return information about the MP. The sentenceTransformer property returns the function that converts the text into vectors. The constituency property returns the constituency chosen by the user. The person_id property returns the ID from Hansard for the chosen MP. The party property returns the political party for the chosen MP. The image property returns the URL of the image of the chosen MP. The full_name property returns the full name of the chosen MP.

```
    def get_speeches(self):
        """
        Gets all the speeches for the chosen MP from the Hansard
API.
        Sets self._speeches to be a list, and each element of the
list, and each entry is a dictionary
        with keys timestamps, and text.
        Timestamp is a unix timestamp.
        Speeches are presented in the order of latest speech to the
```

```
earliest.
        """
        log.info('Getting speeches for %s', self.full_name)
        self._speeches = getSpeeches(self.person_id,
self.minLength)
        log.info('Completed getting speeches for %s',
self.full_name)

    @property
    def speeches(self):
        """
       Returns the speeches.
       If they have not yet been retrieved, retrieve them
        """
        if self._speeches is None:
            self.get_speeches()
        return self._speeches
```

The get_speeches method takes three arguments: self, which represents the instance of the class, and self.person_id, and self.minLength, which are used to get speeches for the chosen MP. The method logs an info message using the log module, indicating that it is getting speeches for the person. It then calls the getSpeeches function with the MP's ID from Hansard, and the minimum length of the speeches, and assigns the result to the _speeches attribute of the instance. Finally, the method logs another info message indicating that it has completed getting speeches for the chosen MP.

The speeches method is a property, which means it is accessed like an attribute rather than a method. It checks if the _speeches attribute is None. If it is, it calls the get_speeches method to retrieve the speeches.

The use of a property for the speeches method allows the speeches to be retrieved only when they are needed, and ensures that they are retrieved if they have not been retrieved yet.

```
    def get_wordcloud_freqs(self):
        """
        Computes the wordcloud frequencies.
        Sets self._wordcloud_freqs to be a Counter.
        The Keys are the word/bigrams, and the values are the
Counts of each word
        """
```

```
        log.info('Preprocessing %d speeches for %s',
                 len(self.speeches), self.full_name)
        text = [x['text'] for x in self.speeches]
        words, bigrams = preprocess_speeches(text, bigrams=True)
        bigrams = list(bigrams)
        log.info('Frequency counting %d words and %d bigrams for
%s',
                 len(words), len(bigrams), self.full_name)
        freqs = bigrams_frequency_count(
            words, bigrams, stopwords=self.stopwords)
        self._wordcloud_freqs = freqs

    @property
    def wordcloud_freqs(self):
        """
        Returns the wordcloud frequencies.
        If they have not been retrieved, retrieve them
        """
        if self._wordcloud_freqs is None:
            self.get_wordcloud_freqs()
        return self._wordcloud_freqs
```

The get_wordcloud_freqs method takes one argument and is used to preprocess speeches, count the frequency of words and bigrams, and store the results in the _wordcloud_freqs attribute. The method first logs an informational message indicating the number of speeches being processed and the name of the person giving the speeches. It then extracts the text from each speech and passes it to the preprocess_speeches function to tokenize the text into words and bigrams. The method then counts the frequency of words and bigrams using the bigrams_frequency_count function and stores the results in the _wordcloud_freqs attribute.

The wordcloud_freqs method is a property method that returns the _wordcloud_freqs attribute. If the attribute is None, the method calls the get_wordcloud_freqs method to calculate the frequency counts.

```
    def get_embeddings(self):
        """
        Computes the embeddings.
        Sets self._embeddings to be a list.
        Each entry of the list is a dictionary with keys
        timestamp - a unix timestamp.
```

```
        text - a speech with all the stopwords removed
        vector - a 300 dimensional vector with norm 1.0 that
represents the text
        idx - the index of this speech in self._speeches
        """
        data = preprocess_speeches_for_embeddings(
            self.speeches, stopwords=self.stopwords,
min_length=self.minLength)
        vectors = self.sentenceTransformer.encode([x['text'] for x
in data], normalize_embeddings=True)
        for i in range(len(data)):
            data[i]['vector'] = vectors[i]
        self._embeddings = data
        log.info('Computed embeddings for %s', self.full_name)

    @property
    def embeddings(self):
        """
        Returns the embeddings.
        If they have not been computed, compute them
        """
        if self._embeddings is None:
            self.get_embeddings()
        return self._embeddings
```

The get_embeddings method is a method of a class that computes embeddings for a list of speeches. The method first calls a preprocess_speeches_for_embeddings function to preprocess the speeches and remove stopwords. The result is a list of dictionaries, where each dictionary contains a timestamp and a text with stopwords removed. The method then uses the sentenceTransformer library to encode the text into a 300-dimensional vector with norm 1.0. The resulting vectors are added to each dictionary in the list. Finally, the method sets the _embeddings attribute of the class to the list of dictionaries and logs a message.

The embeddings method is a property of the same class that returns the embeddings. If the _embeddings attribute is None, the method calls the get_embeddings method to compute the embeddings. Otherwise, it returns the _embeddings attribute.

```
    def find_most_similar(self, sentence):
        """
        The input is a sentence as a text string.
```

```
        This function does the following:
        1. Produces a vector for the sentence
        2. Score the vector against existing embeddings by cosine
similarity
        3. Sort by score, to find the most similar 10 vectors
        4. Recover the original most similar speeches

        Returns a list. Each entry of the list is a dictionary with
keys
        text - a speech by the MP
        date - the date the MP said the speech in a readable format
        """
        log.info('Finding vector for sentence %s', sentence)
        vector = self.sentenceTransformer.encode([sentence],
normalize_embeddings=True)[0]
        log.info('Found vector sentence')
        scores = [cosineSimilarityNormalised(vector,
self.embeddings[i]['vector'])
                  for i in range(len(self.embeddings))]
        idxs = np.argsort(scores)[-10:][::-1]
        most_similar = []
        for i in idxs:
            # log.info('score %.3f', scores[i])
            id = self.embeddings[i]['idx']
            text = self.speeches[id]['text']
            t =
datetime.fromtimestamp(self.speeches[id]['timestamp'])
            t = t.strftime('%d/%m/%Y')
            most_similar.append({'text': text,  'date': t})
            # log.info(text)
        return most_similar
```

The find_most_similar function is a method of a class that uses the sentence transformer library to encode sentences into vectors. The function takes a sentence as input and produces a vector for the sentence using the encode method of the sentence transformer. The normalize_embeddings parameter is set to True to normalise the embeddings. The function then calculates the cosine similarity between the input sentence vector and the vectors of the existing embeddings. The scores are sorted in descending order, and the indices of the top 10 scores are selected. The function then recovers the original speeches corresponding to the top 10 scores and returns them as a list of dictionaries, where each dictionary contains the text of a speech and the date it was given.

```
   def get_topic_model(self):
        """
        Computes a BERTopic topic model
        for example see:
https://maartengr.github.io/BERTopic/getting_started/tips_and_tric
ks/tips_and_tricks.html#speed-up-umap
        """
        # Initiate UMAP
        umap_model = UMAP(n_neighbors=15,
                          n_components=5,
                          min_dist=0.0,
                          metric='cosine',
                          random_state=100)
        # Initiate BERTopic
        self._topic_model = BERTopic(umap_model=umap_model,
embedding_model=self.sentenceTransformer,
                                     language="english",
calculate_probabilities=True, nr_topics=9, verbose=False)

    @property
    def topic_model(self):
        """
        Returns the topic modelling.
        If they have not been retrieved, get the topic models and
return them
        """
        if self._topic_model is None:
            self.get_topic_model()
            self.run_topic_model()

        return self._topic_model
```

The get_topic_model function is a method of a class that computes a BERTopic topic model. The function first initialises a UMAP model with specific parameters, including the number of neighbours, the number of components, the minimum distance, the metric, and the random state. UMAP is a dimensionality reduction technique that is used to reduce the dimensionality of the input data to a lower number of dimensions. The resulting lower-dimensional data is then used as input to the BERTopic model.

Next, the function initialises a BERTopic model with the UMAP model, the sentence-transformers library for embedding, the language, the number of topics,

and the verbosity level. BERTopic is a topic modelling algorithm that is based on the BERT language model. The algorithm uses UMAP for dimensionality reduction and clustering to group similar documents together into topics.

The resulting topic model is stored in the _topic_model attribute of the class instance. The _topic_model attribute is initialised as None and is only set when the get_topic_model function is called. The topic_model property of the class returns the topic model. If the _topic_model attribute is None, the get_topic_model function is called to compute the topic model.

```
   def run_topic_model(self):
        """
        Runs the topic modelling
        Assigns a topic to each speech
        """
        text, embeddings = [x['text'] for x in self.embeddings],
np.array([x['vector'] for x in self.embeddings])
        log.info('Running topic model for %s on %d extracts',
                 self.full_name, len(text))
        self.topics, self.probabilities =
self.topic_model.fit_transform(
            text, embeddings)
        log.info('Completed running topic model %s',
self.full_name)
```

The run_topic_model method is a method of a class that performs topic modelling on a set of text extracts. The method takes no arguments other than self, which refers to the instance of the class. The method first extracts the text and embeddings from the instance's embeddings attribute. The text variable contains a list of the text extracts, and the embeddings variable contains a numpy array of the embeddings for each extract.

The method then logs a message indicating that the topic modelling is being run for the instance's full_name attribute and the number of extracts being analysed. The method then calls the fit_transform method of the instance's topic_model attribute, passing in the text and embeddings variables. The fit_transform method returns two variables: topics and probabilities. The topics variable is a numpy array that contains the topic assigned to each extract, and the probabilities variable is a numpy array that contains the probability of each extract belonging to its assigned topic.

The method then logs a message indicating that the topic modelling has been completed for the instance's full_name attribute. Finally, the method assigns the topics and probabilities variables to the instance's topics and attributes.

```
   def get_representative_docs(self):
        """
         Retrieves representative documents for each topic from
the topic model.

    Returns:
    - tables: List of lists, where each inner list contains the
date and text of a representative document
              for a particular topic.
        """
        tables = []
        log.info('Getting representative docs')
        for i in self.topic_model.get_topics().keys():
            if i == -1:
                continue
            tables.append([])
            reps = self.topic_model.get_representative_docs(i)
            for r in reps:
                j = next(j for j in range(len(self.embeddings))
                         if self.embeddings[j]['text'] == r)
                idx = self.embeddings[j]['idx']
                t =
datetime.fromtimestamp(self.speeches[idx]['timestamp'])
                t = t.strftime('%d/%m/%Y')
                tables[-1].append([t,
self.speeches[idx]['text'].strip()])
        return tables

    @property
    def representative_docs(self):
        """
        Property that returns the representative documents for
each topic from the topic model.
    If the representative documents have not been retrieved yet,
it calls the `get_representative_docs()`
    method to retrieve them and stores them in a private variable
for future access.
    Returns:
    - tables: List of lists, where each inner list contains the
date and text of a representative document
              for a particular topic.
```

```
        """
        if self._representative_docs is None:
            self._representative_docs =
self.get_representative_docs()
        return self._representative_docs
```

The get_representative_docs function is a method of a class that retrieves representative documents for each topic from a topic model. The function first initialises an empty list called tables. The function then enters a loop that iterates through each topic in the topic model. If the topic is -1, the function continues to the next iteration of the loop. Otherwise, the function appends an empty list to tables to represent the current topic. The function then calls the get_representative_docs method of the topic model to retrieve the representative documents for the current topic. The function then enters another loop that iterates through each representative document. For each representative document, the function finds the index of the document in the embeddings and retrieves the date and text of the document from the speeches. The function then appends the date and text of the document to the inner list of tables that represents the current topic.

The representative_docs property is another method of the same class that returns the representative documents for each topic from the topic model. If the representative documents have not been retrieved yet, the property calls the get_representative_docs() method to retrieve them and stores them in a private variable called _representative_docs for future access. The property then returns the _representative_docs variable.

Hansard.py

The Hansard.py file deals with the interactions between the webpage and the HansardAPI.

```
def getQuota():
    url = 'https://www.theyworkforyou.com/api/getQuota'
    response = requests.get(url, params={'key': APIkey})
    return response.json()
```

The getQuota function is a Python function that sends a GET request to the TheyWorkForYou API to retrieve the user's quota limit. The function uses the requests library to send the request and the APIkey variable to authenticate the request.

The function first initialises the url variable with the API endpoint for retrieving the quota limit. The function then sends a GET request to the API using the requests.get method and passes the url and params arguments. The params argument is a dictionary that contains the key parameter with the user's API key.

The function then returns the response from the API as a JSON object using the response.json() method.

```
def getConstituencies():
    log.info('Getting constituencies')
    #log.info(getQuota())
    url = 'https://www.theyworkforyou.com/api/getConstituencies'
    log.info('Created constituencies url')
    response = requests.get(url, params={'key': APIkey})
    log.info('Received response')
    return [x['name'] for x in response.json()]
```

The getConstituencies function is a Python function that retrieves a list of constituencies from a remote API and returns a list of constituency names. The function first logs an informational message to indicate that it is about to retrieve the constituencies. It then creates a URL to retrieve the constituencies from the remote API. The URL is created using the theyworkforyou.com API and is passed an API key as a parameter. The function then sends a GET request to the URL using the requests library. The params parameter is used to pass the API key to the API. The response from the API is then converted to a JSON object using the json() method. Finally, the function returns a list of constituency names by iterating over the JSON object and extracting the name property of each constituency object.

```
def is_valid_postcode(postcode):
    """
    This function takes one argument and returns True or False.

    Args:
    postcode (str): The postcode

    Returns:
    bool: True if a valid postcode, otherwise returns False.
    """
    #making sure the postcode is in uppercase, required for
validation
    postcode = postcode.upper()
    # using the postcodes_uk library to validate the postcode
```

```
    ret = postcodes_uk.validate(postcode)
    return ret
```

The is_valid_postcode function takes a single argument, postcode, which is a string representing a postcode. The function first converts the postcode to uppercase, as this is required for validation. The function then uses the postcodes_uk library to validate the postcode. The postcodes_uk library is a Python library that provides functions for working with UK postcodes. The validate function of the library takes a postcode as an argument and returns True if the postcode is valid, and False otherwise.

The is_valid_postcode function returns the value returned by the validate function of the postcodes_uk library.

```
def getMP(postcode_or_constituency):
    """
    This function takes one argument, either the postcode or the
constituency name, and returns the MP information from Hansard.

    Args:
    postcode_or_constituency (str): Postcode or a Constituency

    Returns:
    dict: The MP information from Hansard
    """
    # checks to see in the input is a valid postcode
    params = {'key':APIkey}
    if is_valid_postcode(postcode_or_constituency):
        params['postcode'] = postcode_or_constituency
    else:
        params['constituency'] = postcode_or_constituency

    url = 'https://www.theyworkforyou.com/api/getMP'
    response = requests.get(url, params=params)
    return response.json()

functools.lru_cache(maxsize = 128)
```

The getMP function is a Python function that takes one argument, either the postcode or the constituency name, and returns the MP information from Hansard. The function first checks if the input is a valid postcode. If it is a valid postcode, the function adds the postcode to the params dictionary with a key of 'postcode'. If it is

not a valid postcode, the function assumes it is a constituency name and adds it to the params dictionary with a key of 'constituency'.

The function then makes an API call to the TheyWorkForYou API using the requests.get method. The url variable contains the URL for the API endpoint, and the params variable contains the parameters to be passed to the API. The response from the API call is returned as a JSON object using the response.json() method.

The functools.lru_cache decorator is used to cache the results of previous function calls. This decorator caches the results of the function for up to 128 function calls. This means that if the function is called with the same argument multiple times, the function will only make one API call and return the cached result for subsequent calls.

```
def getHansard(person_id):
    """
    This function takes one argument and returns the MP's person
ID from Hansard.

    Args:
    personID (int): gets the relevant data from Hansard

    Returns:
    dict: The information from Hansard about the person_id
    """
    params = {'key':APIkey}
    params['person'] = person_id
    params['num'] = 512
    url = 'https://www.theyworkforyou.com/api/getMP'
    response = requests.get(url, params=params)
    data = response.json()
    params['page'] = 1
    # loops until we have all the pages
    with tqdm(total=data['info']['total_results']) as pbar:
        pbar.update(len(data['rows']))
        while True:
            params['page'] += 1
            response = requests.get(url, params=params)
            data0 = response.json()
            pbar.update(len(data0['rows']))
            if len(data0['rows']) == 0:
                break
```

```
            data['rows'].extend(data0['rows'])
    return data
```

The getHansard function is a function that takes person_id as an argument and returns information about the MP from the Hansard API. The function first creates a dictionary called params that contains the API key, the person_id, and the number of results to return per page. The function then creates a URL for the API endpoint and sends a GET request to the endpoint using the requests library. The response is then converted to JSON format and stored in the data variable.

The function then enters a loop that continues until all pages of results have been retrieved. The loop updates a progress bar using the TQDM library and sends a GET request to the API endpoint for each page of results. The response is then converted to JSON format and the rows of data are appended to the data variable. The loop continues until there are no more rows of data to retrieve.

Finally, the function returns the data variable, which contains information about the MP from the Hansard API.

```
def deduplicate(lst):
    deduplicated = []
    for item in lst:
        if item not in deduplicated:
            deduplicated.append(item)
    return deduplicated
```

The deduplicate function takes a list as an argument and returns a new list with all duplicate items removed. The function first initialises an empty list called deduplicated. The function then iterates through each item in the input list. For each item, the function checks if it is already in the deduplicated list. If it is not, the function appends the item to the deduplicated list. If it is already in the deduplicated list, the function skips it and moves on to the next item.

Finally, the function returns the deduplicated list.

```
import time
def make_timestamp(_date, _time):
    try:
        if not _time is None:
            return time.mktime(time.strptime("%s %s" % (_date,
_time), "%Y-%m-%d %H:%M:%S"))
```

```
        else:
            return time.mktime(time.strptime("%s" % (_date),
"%Y-%m-%d"))
    except:
        if not _time is None:
            return make_timestamp(_date, None)
        raise(ValueError, 'Date and time to not make sense. %s
%s\n' % _date, _time)
```

The make_timestamp function is a function that takes two arguments: _date and _time. The _date argument is a string representing a date in the format YYYY-MM-DD, and the _time argument is a string representing a time in the format HH:MM:SS. The function returns a Unix timestamp representing the date and time.

The function first checks if the _time argument is not None. If it is not None, the function uses the time.strptime function to parse the _date and _time strings and convert them to a time tuple. The function then uses the time.mktime function to convert the time tuple to a Unix timestamp.

If the _time argument is None, the function uses the time.strptime function to parse the _date string and convert it to a time tuple representing the start of the day. The function then uses the time.mktime function to convert the time tuple to a Unix timestamp.

If the function encounters an error while parsing the date and time strings, it calls itself recursively with the _time argument set to None. If the recursive call also encounters an error, the function raises a ValueError with an error message.

```
def getSpeeches(person_id, minLength=25):
    """
    This function takes one argument and returns the MP's person
ID from Hansard.

    Args:
    person_id (int): gets the relevant data from Hansard

    Returns:
    list: Each list entry is one speech given by the MP and a
timestamp
    """
    data = getHansard(person_id)
    speeches = []
```

```
    for x in data['rows']:
        body = x['body']
        text = BeautifulSoup(body, "html.parser").text
        if len(text.split()) < minLength:
            continue
        if len(speeches) > 0 and text == speeches[-1]['text']:
            continue
        speeches.append({'timestamp' : make_timestamp(x['hdate'],
x['htime']), 'text' : text})
    return deduplicate(speeches)
```

The getSpeeches function takes two arguments: person_id, which is an integer representing the ID of the MP whose speeches are to be retrieved, and minLength, which is an optional integer representing the minimum length of a speech in words. If minLength is not provided, it defaults to 25.

The function first calls the getHansard function with the person_id argument to retrieve data from Hansard. The data is stored in the data variable. The function then initialises an empty list called speeches to store the extracted speeches. The function then enters a loop that iterates through each row in the data variable. For each row, the function extracts the speech text from the HTML using the BeautifulSoup library and checks if the length of the speech text is greater than or equal to the minLength argument. If the length is less than minLength, the function skips to the next iteration of the loop. If the length is greater than or equal to minLength, the function checks if the speech text is a duplicate of the previous speech. If it is a duplicate, the function skips to the next iteration of the loop. If it is not a duplicate, the function adds the speech to the speeches list along with a timestamp.

Finally, the function calls the deduplicate function to remove any duplicate speeches and returns the resulting list of speeches with timestamps.

Wordcloud.py

This file deals with the creation of frequency counts for the wordcloud and text preprocessing routines,

```
def preprocess_speeches(speeches, bigrams=True):
    #Concatenates the speeches into one string of text
    text = '\n'.join(speeches)
    #Removes all punctuation
```

```
    text = text.translate(str.maketrans('', '',
string.punctuation))
    #Removes text inside parenthesis
    text = re.sub(r'\d+', '', text)
    #Converts text into lowercase, and splits by whitespace
    words = text.lower().split()
    words = [w for w in words if len(w) > 1]
    #Reduce each word to its root
    log.info('Running lemmatizer')
    words = [lemmatizer.lemmatize(w, pos='v') for w in words]
    log.info('Completed running lemmatizer')
    if bigrams == False:
        return words
    bigrams = ngrams(words, 2)
    return words, bigrams
```

The preprocess_speeches function is a Python function that takes a list of speeches and an optional bigrams parameter. The function first concatenates all the speeches into a single string of text using the join method. It then removes all punctuation from the text using the translate method and the string.punctuation constant. Next, it removes any text inside parentheses using a regular expression that matches any digit. The function then converts the text to lowercase and splits it into words using whitespace as a delimiter. It filters out any words that are one character long and reduces each word to its root using a lemmatizer. Finally, if the bigrams parameter is False, the function returns a list of words. Otherwise, it generates a list of bigrams using the ngrams function.

```
def preprocess_speeches_for_embeddings(speeches, stopwords=[],
min_length=25):
    out = []
    for i in range(len(speeches)):
        text = speeches[i]['text']
        #Removes all punctuation
        text = text.translate(str.maketrans('', '',
string.punctuation))
        #Removes text inside parenthesis
        text = re.sub(r'\d+', '', text)
        #Lower case
        text = text.lower()
        #Remove stopwords
        text = " ".join(w for w in text.split() if not w in
stopwords)
```

```
        if len(text.split()) >= min_length:
            out.append({'idx' : i, 'timestamp' :
speeches[i]['timestamp'], 'text' : text})
    return out
```

The preprocess_speeches_for_embeddings function takes three arguments: speeches, which is a list of speeches, stopwords, which is a list of words to be removed from the speeches, and min_length, which is the minimum length of a speech after preprocessing.

The function initialises an empty list called out, which will be used to store the preprocessed speeches. The function then enters a loop that iterates over each speech in the speeches list. For each speech, the function gets the text of the speech and removes all punctuation using the translate method of the string class. The function then removes all text inside parentheses using a regular expression. The text is then converted to lowercase using the lower method of the string class. The function then removes all stopwords from the text using a list comprehension. If the length of the resulting text is greater than or equal to the min_length argument, the function appends a dictionary containing the index, timestamp, and preprocessed text of the speech to the out list.

Finally, the function returns the list containing the speeches after being preprocessed.

```
def bigrams_frequency_count(words, bigrams, stopwords = []):
    #Counts words as before, excluding stopwords
    words_counter = Counter([w for w in words if not w in
stopwords])
    #Counts bigrams, excluding any that are in the list of
stopwords
    bigrams_counter = Counter([' '.join(w) for w in bigrams if not
w[0] in stopwords and not w[1] in stopwords])
    #Gets a dictionary of the 250 most common words
    words_counter = words_counter.most_common(250)
    words_counter = {w[0] : w[1] for w in words_counter}
    #Gets a dictionary with the 250 most common bigrams
    bigrams_counter = bigrams_counter.most_common(250)
    bigrams_counter = {w[0] : w[1] for w in bigrams_counter}
    #Removes counts from single words, if they appear in the list
of bigrams
    for w in words_counter.keys():
        for b in bigrams_counter.keys():
```

```
            bg = b.split()
            if w == bg[0] or w == bg[1]:
                words_counter[w] -= bigrams_counter[b]
    return Counter(words_counter) + Counter(bigrams_counter)
```

The bigrams_frequency_count function takes three arguments: words, which is a list of words, bigrams, which is a list of bigrams, and stopwords, which is an optional list of stopwords. The function returns a Counter object that contains the frequency count of the 250 most common words and bigrams, excluding any stopwords.

The function first creates a Counter object called words_counter that counts the frequency of each word in the words list, excluding any stopwords. The stopwords list is optional and defaults to an empty list if not provided. Similarly, the function creates a Counter object called bigrams_counter that counts the frequency of each bigram in the bigrams list, excluding any bigrams that contain stopwords.

The function then gets the 250 most common words and bigrams from their respective Counter objects and creates dictionaries that map each word or bigram to its frequency count. The most_common method of the Counter object returns a list of tuples, where each tuple contains a word or bigram and its frequency count. The function uses a dictionary comprehension to create a dictionary that maps each word or bigram to its frequency count.

The function then removes the counts of single words that appear in the list of bigrams. For each word in the words_counter dictionary, the function checks if it appears in any of the bigrams in the bigrams_counter dictionary. If it does, the function subtracts the frequency count of the bigram from the frequency count of the word.

Finally, the function returns the sum of the words_counter and bigrams_counter Counter objects.

__init__.py

This is the __init__ file for the module. As an optimisation many of the functions used later are imported here so they are available to use later.

```
from flask import Flask
from flask_bootstrap import Bootstrap5
from flask_session import Session
```

```
import logging
import psutil
import time
print('Importing SentenceTransformer. %.3f%% memory usage' %
psutil.virtual_memory().percent)
_t0 = time.time()
from sentence_transformers import SentenceTransformer
_t1 = time.time()
print('\tTook %.3f seconds ' % (_t1-_t0))
print('Loading Specific SentenceTransformer. %.3f%% memory usage'
% psutil.virtual_memory().percent)
_t0 = time.time()
HansardSentenceTransformer =
SentenceTransformer('sentence-transformers/average_word_embeddings
_komninos')
_t1 = time.time()
print('\tTook %.3f seconds ' % (_t1-_t0))
#HansardSentenceTransformer =
SentenceTransformer('nreimers/MiniLM-L6-H384-uncased')
print('Importing bertopic. %.3f%% memory usage' %
psutil.virtual_memory().percent)
_t0 = time.time()
from bertopic import BERTopic
_t1 = time.time()
print('\tTook %.3f seconds ' % (_t1-_t0))
print('Importing UMAP. %.3f%% memory usage' %
psutil.virtual_memory().percent)
_t0 = time.time()
from umap import UMAP
_t1 = time.time()
print('\tTook %.3f seconds ' % (_t1-_t0))
print('Importing nltk ngrams and WordNetLemmatizer. %.3f%% memory
usage' % psutil.virtual_memory().percent)
_t0 = time.time()
from nltk import ngrams
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
_ = lemmatizer.lemmatize('cat') #Force it to do initial setup work
now
_t1 = time.time()
print('\tTook %.3f seconds ' % (_t1-_t0))
```

```
logging.basicConfig(level=logging.INFO, format="%(asctime)s
%(levelname)s : %(message)s", datefmt='%m/%d/%Y %I/%M:%S %p')
log = logging.getLogger(__file__)


def create_app(test_config=None):
    app = Flask(__name__, instance_relative_config=True)
    app.logger.info('Starting create_app')
    app.config.from_mapping(
        SECRET_KEY = 'dev',
    )


    app.logger.info('Importing main')

    from . main import bp
    app.logger.info('Registering Main')
    app.register_blueprint(bp)

    app.logger.info('Bootstrap5')
    bootstrap = Bootstrap5(app)
    app.logger.info('Completed create_app')
    app.config["SESSION_PERMANENT"] = False
    app.config["SESSION_TYPE"] = "filesystem"
    Session(app)
    app.logger.info('Completed session_app')
    app.jinja_env.cache = {}
    return app
```

The code defines a Flask app using the create_app function. The function takes an optional test_config argument. The function initialises the Flask app and sets its SECRET_KEY configuration option.

The function then imports various libraries, including Flask Bootstrap, Flask Session, logging, psutil, time, SentenceTransformer, BERTopic, UMAP, and nltk. It also initialises a WordNetLemmatizer object from nltk.

The code then logs various messages using the logging module.

Next, the function registers a blueprint named bp from the main module. The main module contains the main logic of the app.

The function then initialises a Bootstrap5 object and sets up session management using the Flask Session extension.

Finally, the function returns the Flask app object.

run.py

```
import argparse
from flaskr import create_app

import argparse

# create an argument parser object
parser = argparse.ArgumentParser(description='Specify a port
number')
# add an argument for the port number
parser.add_argument('--port', type=int, default=8081, help='the
port number to use')
# add an argument for the debug flag
parser.add_argument('--debug', action='store_true', help='enable
debug mode')


app = create_app()

if __name__ == '__main__':
    # parse the arguments from the command line
    args = parser.parse_args()
    # access the port number entered by the user (or the default
value)
    port_number = args.port
    # access the debug flag entered by the user (or False by
default)
    debug_mode = args.debug
    if debug_mode:
        app.run(debug=True, port=port_number)
    else:
        app.run(host='0.0.0.0', port=port_number, threaded=True)
```

The Python script starts by importing the argparse module and the create_app function from the flaskr module. The argparse module is used to create an argument

parser object that allows the user to specify a port number and enable debug mode when running the Flask application.

The script creates an argument parser object using the ArgumentParser class from the argparse module. The description parameter is used to provide a brief description of the script's purpose. The script then adds two arguments to the argument parser object using the add_argument method. The first argument is for the port number, and the second argument is for the debug flag.

The type parameter is used to specify the data type of the port number argument, which is an integer. The default parameter is used to specify the default value of the port number argument, which is 8081. The help parameter is used to provide a brief description of the port number argument.

The action parameter is used to specify that the debug flag argument is a boolean flag. The store_true value is used to indicate that the flag should be set to True if it is present in the command line arguments. The help parameter is used to provide a brief description of the debug flag argument.

The script then creates a Flask application using the create_app function from the flaskr module. The create_app function returns a Flask application object that is used to handle HTTP requests.

The script checks if the __name__ variable is equal to '__main__'. This is used to ensure that the script is being run as the main program and not being imported as a module.

If the script is being run as the main program, the script parses the command line arguments using the parse_args method of the argument parser object. The parse_args method returns an object that contains the values of the command line arguments.

The script then accesses the port number and debug flag entered by the user (or the default values) using the attributes of the object returned by the parse_args method.

Finally, the script runs the Flask application using the run method of the Flask application object. If the debug flag is True, the Flask application is run in debug mode with the specified port number. Otherwise, the Flask application is run with the specified port number and threaded mode.

Main.html

This is the code for the website itself. As an optimization the css is embedded directly in the html. This is written in Jinja template language. The code inside {} is evaluated in Python before the webpage code is created.

```
{% extends "bootstrap_base.html" %}

{% block styles %}
{{super()}}
<!-- <link rel="stylesheet" href="{{ url_for('static',
filename='style/newstyle.css') }}"> -->
<style>
    #container {
        position: absolute;
        top: 0;
        right: 20%;
        left: 0;
        bottom: 0;
    }

    #topleft_div {
        position: absolute;
        top: 0;
        right: 0;
        left: 0%;
        width: 25%;
        bottom: 45%;
        background-color: #ffffff;
        text-align: center;
    }

    #bottomleft_div {
        position: absolute;
        top: 45%;
        right: 0;
        left: 0%;
        width: 25%;
        bottom: 0%;
        background-color: #ffffff;
        text-align: center;
    }

    #top_div {
```

```
        position: absolute;
        top: 0;
        right: 0;
        left: 25%;
        width: 100%;
        bottom: 65%;
        background-color: #009900;
        text-align: center;
    }

    #bottom_div {
        position: absolute;
        top: 0%;
        right: 0;
        left: 25%;
        width: 95%;
        bottom: 0;
        background-color: #ffffff;
        text-align: center;
        color: #FFFFFF;
    }

    #Wordcloud {
        position: absolute;
        top: 10%;
        right: 0;
        left: 0%;
        width: 100%;
        bottom: 0;
        background-color: #ffffff;
        text-align: center;
        color: #FFFFFF;
    }

    /* Style the tab */
    .tab {
        overflow: hidden;
        border: 1px solid #ccc;
        background-color: #f1f1f1;
    }

    /* Style the buttons inside the tab */
```

```
.tab button {
    background-color: inherit;
    float: left;
    border: none;
    outline: none;
    cursor: pointer;
    padding: 14px 16px;
    transition: 0.3s;
    font-size: 17px;
}

/* Change background color of buttons on hover */
.tab button:hover {
    background-color: #ddd;
}

/* Create an active/current tablink class */
.tab button.active {
    background-color: #ccc;
}

/* Style the tab content */
.tabcontent {
    display: none;
    padding: 6px 12px;
    border: 1px solid #ccc;
    border-top: none;
}

h2,
h1,
h3,
p {
    color: #000000;
}


table,
td,
th {
    color: #000000;
}
```

```
    table,
    th,
    td {
        border: 1px solid;
    }

    th,
    tr {
        text-align: justify;
    }


    .grid-container {
        display: grid;
        grid-template-columns: auto auto auto auto;
        background-color: #2196F3;
        padding: 10px;
    }

    .grid-item {
        background-color: rgba(255, 255, 255, 0.8);
        border: 1px solid rgba(0, 0, 0, 0.8);
        padding: 20px;
        font-size: 30px;
        text-align: center;
    }

    div#loading_div {
        display: none;
        cursor: wait;
    }

    div#loading_search_div {
        display: none;
        cursor: wait;
    }
</style>
{% block title %} Hansard Explorer {% endblock %}

{% block html_attribs %} lang="en"{% endblock %}
```

The code is an HTML/CSS template that defines the layout and styles of a web page. It uses absolute positioning to create four divs that occupy different parts of the page. The #topleft_div and #bottomleft_div divs are positioned on the left side of the page, while the #top_div and #bottom_div divs are positioned on the right side of the page. The #Wordcloud div is positioned in the centre of the page.

The #topleft_div and #bottomleft_div divs have a white background colour and are centred horizontally. The #top_div div has a green background colour and is positioned below the #topleft_div and #bottomleft_div divs. The #bottom_div div has a white background colour and is positioned below the #top_div div. The #Wordcloud div has a white background colour and is positioned in the centre of the page.

The code also defines a tabbed interface using the .tab class. The tabbed interface has multiple tabs, each of which is represented by a button with the .tab button class. The active tab is represented by a button with the .tab button.active class. The content of each tab is represented by a div with the .tabcontent class.

Finally, the code defines a grid layout using the .grid-container and .grid-item classes. The grid layout has four columns and an auto-generated number of rows. Each cell in the grid is represented by a div with the .grid-item class.

```
{% block content %}
<div id="container">
    <div id="topleft_div">
        {% if selected_constituency != None %}
        <h3>{{ selected_constituency }}</h3>
        {% endif %}
        {% if MP != None %}
        <figure class="figure">
            <img src="{{ MP.image }}" alt="image of {{
MP.full_name }}" class="figure-img img-fluid rounded">
            <figcaption class="figure-caption">{{ MP.full_name
}}</figcaption>
        </figure>
        <h3>{{ MP.party }}</h3>
        {% endif %}
    </div>
    <div id="bottomleft_div">
        <form method="POST" action="{{url_for('main.search') }}">
            <select class="form-select" name="constituency"
method="GET" action="/">
                <option selected="selected" disables hidden>Select
```

```
constituency</option>
                {% for constituency in constituencies %}
                <option
value="{{constituency}}">{{constituency}}</option>"
                {% endfor %}
            </select>
            <button type="submit" class="btn btn-primary"
onclick="loadingMP();">Go</button>
        </form>
    </div>
    <div id="bottom_div">
        <div id="loading_div">
            <br>
            <img style="display: inline-block; height: 300px;
width: auto; text-align:center"

src="https://ichef.bbci.co.uk/news/976/cpsprodpb/917A/production/_
108824273_chimps2.jpg.webp">
            <br>
            <h2> Loading </h2>
            <img style="display: block;-webkit-user-select:
none;margin: auto;background-color: hsl(0, 0%, 90%);"

src="https://mir-s3-cdn-cf.behance.net/project_modules/disp/ab79a2
31234507.564a1d23814ef.gif">
        </div>
        <div id="content_div">
            {% if instructions %}
            <h2> Welcome to Hansard Explorer </h2>
            <br>
            <p> To select a constituency, use the dropdown menu on
the left.</p>
            <p> When the MP has finished loading, select one of
the tabs above to get more information about their
                speeches. </p>
            <p> The Word Cloud tab produces a word cloud of all
the speeches from the selected MP over 25 words, and
                puts them into a word cloud. This is the tab that
is first shown when the MP has finished loading in
            </p>
            <p> The Topics tab produces a selection of common
topics that the selected MP discusses, when you click on
```

```
                one of the topics, a list of speeches will show
up</p>
            <p> If you have selected the Search tab, search for
the word you would like to find in one of the speeches,
                and a list of relevant speeches will show up. </p>
            {% else %}
            <div class="tab">
                <button class="tablinks" onclick="openTab(event,
'Wordcloud')">Wordcloud</button>
                <button class="tablinks" onclick="openTab(event,
'Topics')">Topics</button>
                <button class="tablinks" onclick="openTab(event,
'Search')">Search</button>
            </div>
            <div id="Wordcloud" class="tabcontent"></div>
            <div id="Topics" class="tabcontent">
                {% if topics_data != None %}
                <h3> Click on topics to display representative
speeches </h3>
                <div class="grid-container">
                    {% for i in range(topics_data |length) %}
                    <div class="grid-item"
id="TopicWordcloud{{i}}" onclick="displayTopicRep({{i}})"></div>
                    {% endfor %}
                </div>
                <div id="TopicRep">
                    {% for i in range(representative_docs |length)
%}
                    <div id="TopicRep{{i}}">
                        <h2>
                            Topic {{i+1}}
                        </h2>
                        <table>
                            <tr>
                                <th>Date</th>
                                <th>Speech</th>
                            </tr>
                            {% for k in
range(representative_docs[i] |length) %}
                            <tr>

<td>{{representative_docs[i][k][0]}}</td>
```

```
<td>{{representative_docs[i][k][1]}}</td>
                            </tr>
                            {% endfor %}
                        </table>
                    </div>
                    {% endfor %}
                </div>
                {% endif %}
            </div>
            <div id="Search" class="tabcontent">
                {% if form != None %}
                <form method="POST" action="{{
url_for('main.search') }}" onsubmit="loadingSearch();">
                    {{ form.hidden_tag() }}
                    {{ form.searchTerm.label }} {{ form.searchTerm
}}
                    {{ form.submit }}
                </form>
                {% endif %}
                <div id="loading_search_div">
                    <h2> Searching for most similar speeches </h2>
                    <img style="display:
block;-webkit-user-select: none;margin: auto;background-color:
hsl(0, 0%, 90%);"

src="https://mir-s3-cdn-cf.behance.net/project_modules/disp/ab79a2
31234507.564a1d23814ef.gif">
                </div>
                <br>
                {% if most_similar != None %}
                <div>
                    <table>
                        <tr>
                            <th>Date</th>
                            <th>Speech</th>
                        </tr>
                        {% for i in range(most_similar|length) %}
                        <tr>
                            <td>{{ most_similar[i].date }}</td>
                            <td>{{ most_similar[i].text }}</td>
                        </tr>
```

```
                        {% endfor %}
                    </table>
                </div>
                {% endif %}
            </div>
            {% endif %}
        </div>
    </div>
</div>
{% endblock %}
```

The page is divided into three main sections: the top left section, the bottom left section, and the bottom section.

The top left section displays the name of the selected constituency and the name and party of the MP representing that constituency.

The bottom left section contains a form that allows the user to select a constituency. The form includes a dropdown menu with a list of constituencies and a "Go" button to submit the form.

The bottom section is divided into two parts: the loading screen and the content section. The loading screen is displayed when the user submits the form to select a constituency. It includes a loading animation and a message indicating that the page is loading. The content section is displayed once the page has finished loading. It includes tabs to view information about the MP's speeches.

The tabs include the Wordcloud tab, which displays a word cloud of the MP's speeches; the Topics tab, which displays common topics that the MP discusses and allows the user to view representative speeches for each topic; and the Search tab, which allows the user to search for speeches containing a specific word or phrase.

The HTML code also includes JavaScript functions to handle the tabs and display the loading screen and search results.

```
{% block scripts %}
{{super()}}
{% if instructions %}
{% else %}
<script>
    {% if display_tab == 'wordcloud' %}
    document.getElementById("Wordcloud").style.display = "block"
```

```
    {% endif %}
    {% if display_tab == 'search' %}
    document.getElementById("Search").style.display = "block"
    {% endif %}
</script>
{% endif %}
<script>
    function openTab(evt, cityName) {
        var i, tabcontent, tablinks;
        tabcontent =
document.getElementsByClassName("tabcontent");
        for (i = 0; i < tabcontent.length; i++) {
            tabcontent[i].style.display = "none";
        }
        tablinks = document.getElementsByClassName("tablinks");
        for (i = 0; i < tablinks.length; i++) {
            tablinks[i].className =
tablinks[i].className.replace(" active", "");
        }
        document.getElementById(cityName).style.display = "block";
        evt.currentTarget.className += " active";
    }
</script>
<script>
    function loadingMP() {
        document.getElementById("loading_div").style.display =
"block"
        document.getElementById("content_div").style.display =
"none"
        document.getElementById("topleft_div").style.display =
"none"
    }
</script>
<script>
    function loadingSearch() {

document.getElementById("loading_search_div").style.display =
"block"
    }
</script>
{% if wordclouddata != None %}
<script
```

```
src="https://cdn.anychart.com/releases/v8/js/anychart-base.min.js"
></script>
<script
src="https://cdn.anychart.com/releases/v8/js/anychart-tag-cloud.mi
n.js"></script>
<script>
    anychart.onDocumentReady(function () {
        var data = [
            {% for W in wordclouddata %}
            { "x": "{{W['word']}}", "value": {{ W['value']}}},
        {% endfor %}
        ];

    // create a tag (word) cloud chart
    var chart = anychart.tagCloud(data);

    // set a chart title
    chart.title("Word cloud for {{MP.full_name}}");
    // set an array of angles at which the words will be laid out
    chart.angles([0])
    // enable a color range
    chart.colorRange(true);
    // set the color range length
    chart.colorRange().length('80%');

    // display the word cloud chart
    chart.container("Wordcloud");
    chart.draw();
    });
</script>
{% endif %}
{% if topics_data != None %}
{% for i in range(topics_data |length) %}
<script>
    anychart.onDocumentReady(function () {
        var data = [
            {% for W in topics_data[i] %}
            { "x": "{{W['word']}}", "value": {{ W['value']}}},
        {% endfor %}
        ];

    // create a tag (word) cloud chart
```

```
    var chart = anychart.tagCloud(data);

    // set a chart title
    chart.title("Topic {{i+1}}");
    // set an array of angles at which the words will be laid out
    chart.angles([0])
    // enable a color range
    chart.colorRange(false);
    // set the color range length
    // chart.colorRange().length('80%');
    chart.normal().stroke("#000000");
    //chart.normal().fontWeight(600);
    // display the word cloud chart
    chart.container("TopicWordcloud{{i}}");
    chart.draw();
    });
</script>
{% endfor %}
{% endif %}
{% if topics_data != None %}
<script>
    const topicTables = new Array();
    {% for i in range(representative_docs | length) %}
    topicTables.push(document.getElementById("TopicRep{{i}}"));
    topicTables[{{ i }}].style.display = "none";
    {% endfor %}
    function displayTopicRep(i) {
        var j;
        for (j = 0; j < topicTables.length; j++) {
            if (j == i) {
                topicTables[i].style.display = "block";
            }
            else {
                topicTables[j].style.display = "none";
            }
        }
    }
</script>
{% endif %}
{% endblock %}
```

The code block defines several JavaScript functions and scripts that are used to display and manipulate HTML elements in a web page. The scripts include functions to open tabs, display loading animations, and create word cloud charts.

The first script block defines a function called openTab that is used to switch between tabs in a web page. The function takes two arguments: evt, which represents the event that triggered the function, and cityName, which is the name of the tab to be opened. The function first hides all the tab content by setting their display property to "none". It then removes the "active" class from all the tab links. Finally, it shows the content of the selected tab and adds the "active" class to the selected tab link.

The second script block defines a function called loadingMP that is used to display a loading animation while the word cloud chart is being generated. While it is loading, the contents of the page are hidden, and are replaced by a loading screen.

The third script block defines a function called loadingSearch that is used to display a loading animation while the programme generates the results.

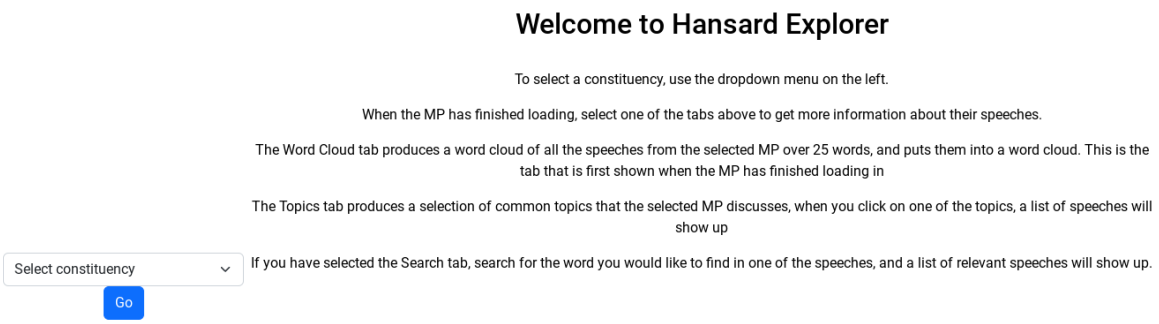
The fourth script block creates a word cloud chart using the AnyChart library. The script first loads the AnyChart library and sets up the data for the chart. It then creates a tag cloud chart using the data and sets the chart title, angles, and colour range. Finally, it displays the chart in the HTML element with the ID "Wordcloud".

The fifth script block creates one or more word cloud charts for the topics in the search results. The charts are created only if the topics_data variable is not None. The script first sets up the data for each chart and creates a tag cloud chart using the data. It then sets the chart title, angles, and stroke colour. Finally, it displays the chart in the HTML element with the ID "TopicWordcloud".

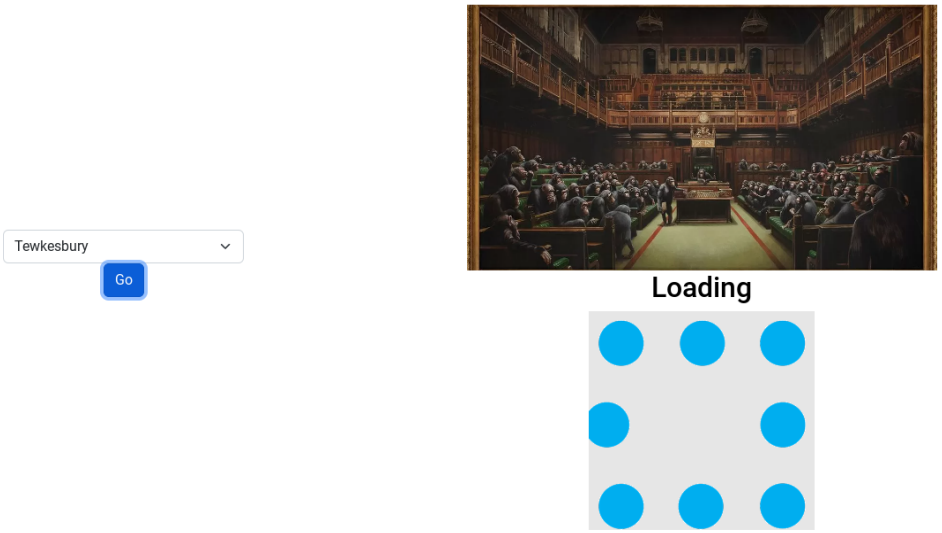
The sixth script block defines a function called displayTopicRep that is used to display the representative documents for a selected topic. The function takes one argument, i, which represents the index of the selected topic. The function first hides all the representative document tables and then shows the table for the selected topic.

The final webpage

The final webpage will look like the design plan, and will function like the table, with the search function to the left, and instructions to the right. The instructions only appear at the first instance of loading up the website, as the user will be familiar with how the web page works.



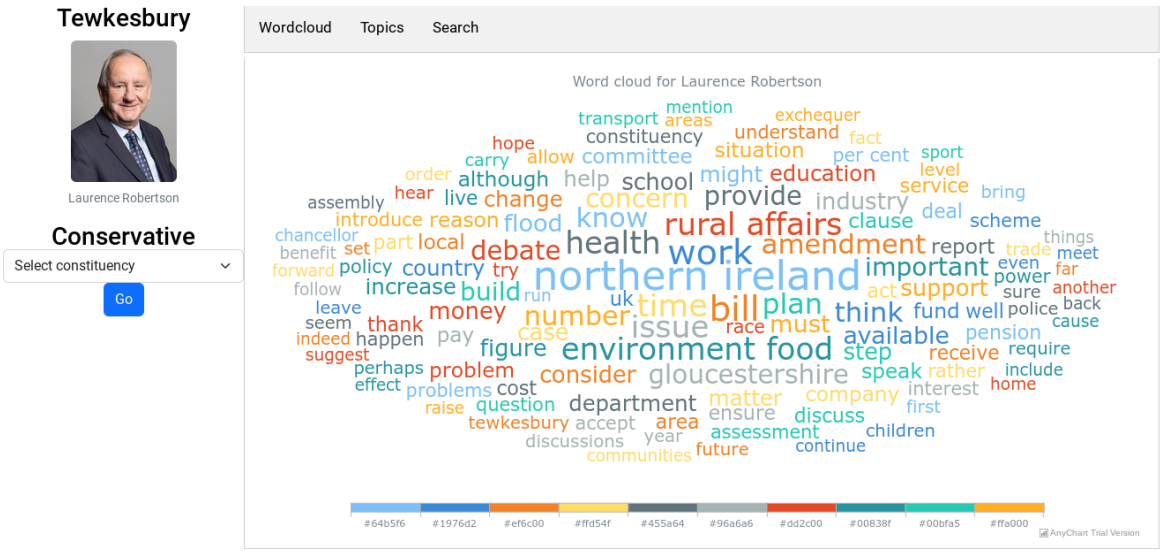
When the user selects a constituency using the dropdown search menu, everything disappears, and a loading screen is added, showing the user that the programme is working, and loading, because the user will be waiting for a while as some MPs have thousands of speeches that need to be pre-processed.



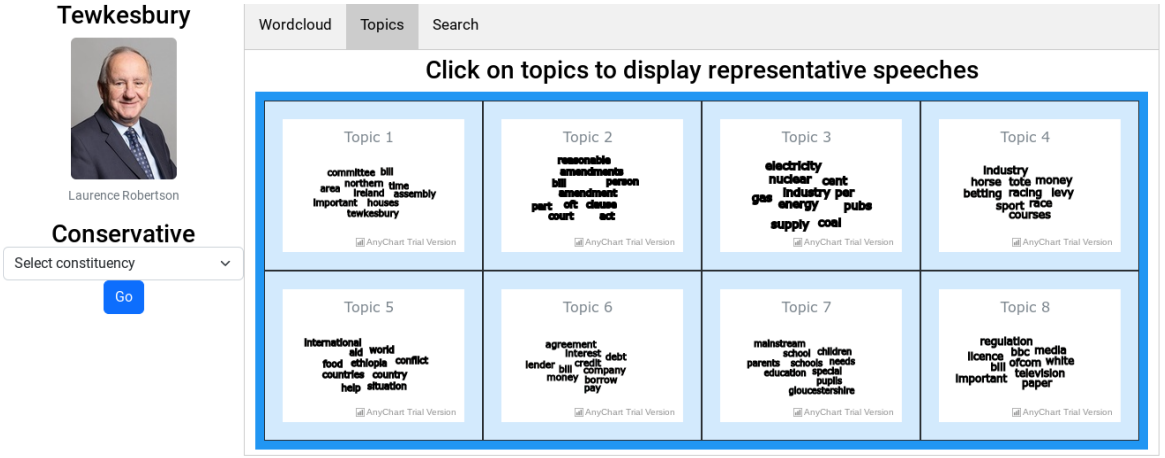
When the webpage loads in, a picture of the MP is loaded in on the left, that shows their name, constituency they serve in, and the party they are affiliated with. It also has the dropdown menu below it, in case the user wants to select another MP. On the top of the page, a list of tabs are displayed that lead to other parts of the programme that will be discussed later. In the middle of the page, a word cloud full of all the words that are present in the speeches of the chosen MP. The more frequent the words appear, the bigger they are in the word cloud. This also includes the standard list of stop words, and the extra list of stop words that had to be added to

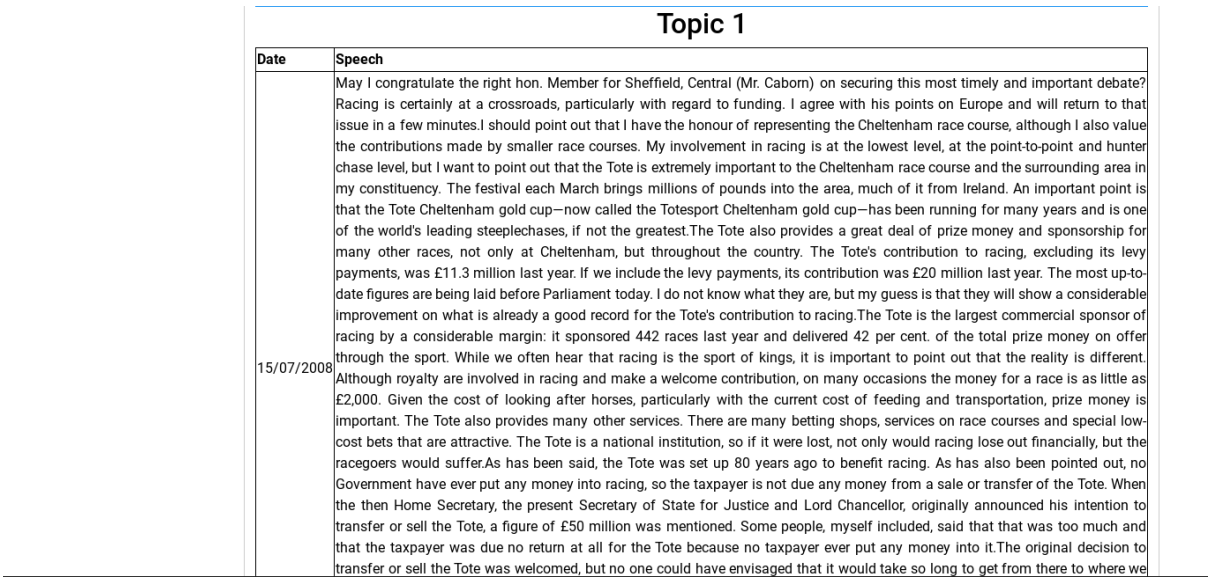
the programme to stop the bombardment of parliamentary language clogging up the meaningful words in the cloud.

Laurence Robertson (Tewkesbury)



The second tab, produces a selection of topics, up to a limit of eight, of common topics that the chosen MP discusses. When the user clicks on one of the topics, a selection of speeches that are relevant to that topic are shown, along with the date of when that speech was said. For example, one of the topics this MP talks about is horse racing and betting. When the user clicks on that topic, a selection of speeches that discuss the topic are shown. The user can go back to any of the tabs at any time, with no loading between them.

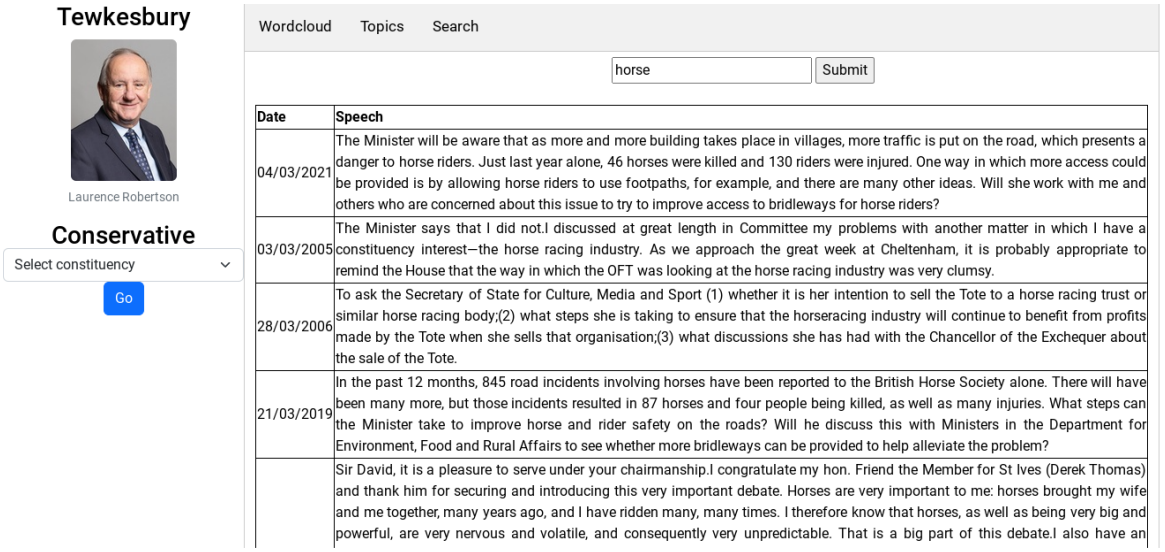




Topic 1

Date	Speech
15/07/2008	May I congratulate the right hon. Member for Sheffield, Central (Mr. Caborn) on securing this most timely and important debate? Racing is certainly at a crossroads, particularly with regard to funding. I agree with his points on Europe and will return to that issue in a few minutes.I should point out that I have the honour of representing the Cheltenham race course, although I also value the contributions made by smaller race courses. My involvement in racing is at the lowest level, at the point-to-point and hunter chase level, but I want to point out that the Tote is extremely important to the Cheltenham race course and the surrounding area in my constituency. The festival each March brings millions of pounds into the area, much of it from Ireland. An important point is that the Tote Cheltenham gold cup—now called the Totesport Cheltenham gold cup—has been running for many years and is one of the world's leading steeplechases, if not the greatest.The Tote also provides a great deal of prize money and sponsorship for many other races, not only at Cheltenham, but throughout the country. The Tote's contribution to racing, excluding its levy payments, was £11.3 million last year. If we include the levy payments, its contribution was £20 million last year. The most up-to-date figures are being laid before Parliament today. I do not know what they are, but my guess is that they will show a considerable improvement on what is already a good record for the Tote's contribution to racing.The Tote is the largest commercial sponsor of racing by a considerable margin: it sponsored 442 races last year and delivered 42 per cent. of the total prize money on offer through the sport. While we often hear that racing is the sport of kings, it is important to point out that the reality is different. Although royalty are involved in racing and make a welcome contribution, on many occasions the money for a race is as little as £2,000. Given the cost of looking after horses, particularly with the current cost of feeding and transportation, prize money is important. The Tote also provides many other services. There are many betting shops, services on race courses and special low-cost bets that are attractive. The Tote is a national institution, so if it were lost, not only would racing lose out financially, but the racegoers would suffer.As has been said, the Tote was set up 80 years ago to benefit racing. As has also been pointed out, no Government have ever put any money into racing, so the taxpayer is not due any money from a sale or transfer of the Tote. When the then Home Secretary, the present Secretary of State for Justice and Lord Chancellor, originally announced his intention to transfer or sell the Tote, a figure of £50 million was mentioned. Some people, myself included, said that that was too much and that the taxpayer was due no return at all for the Tote because no taxpayer ever put any money into it.The original decision to transfer or sell the Tote was welcomed, but no one could have envisaged that it would take so long to get from there to where we

The final tab allows the user to search for a certain word or phrase present in the speeches of their chosen MP. This is relevant to the end user, as people who responded to the survey wanted the ability to search through the data. As an example, if the user wants to find out if their chosen MP talks about horses, they can input the word into the search box in the tab, and relevant speeches will be shown below the search box. The only downside is that the list of topics change each time you search for the same MP. The image above is an example of speeches on a topic.



Date	Speech
04/03/2021	The Minister will be aware that as more and more building takes place in villages, more traffic is put on the road, which presents a danger to horse riders. Just last year alone, 46 horses were killed and 130 riders were injured. One way in which more access could be provided is by allowing horse riders to use footpaths, for example, and there are many other ideas. Will she work with me and others who are concerned about this issue to try to improve access to bridleways for horse riders?
03/03/2005	The Minister says that I did not.I discussed at great length in Committee my problems with another matter in which I have a constituency interest—the horse racing industry. As we approach the great week at Cheltenham, it is probably appropriate to remind the House that the way in which the OFT was looking at the horse racing industry was very clumsy.
28/03/2006	To ask the Secretary of State for Culture, Media and Sport (1) whether it is her intention to sell the Tote to a horse racing trust or similar horse racing body;(2) what steps she is taking to ensure that the horseracing industry will continue to benefit from profits made by the Tote when she sells that organisation;(3) what discussions she has had with the Chancellor of the Exchequer about the sale of the Tote.
21/03/2019	In the past 12 months, 845 road incidents involving horses have been reported to the British Horse Society alone. There will have been many more, but those incidents resulted in 87 horses and four people being killed, as well as many injuries. What steps can the Minister take to improve horse and rider safety on the roads? Will he discuss this with Ministers in the Department for Environment, Food and Rural Affairs to see whether more bridleways can be provided to help alleviate the problem?
	Sir David, it is a pleasure to serve under your chairmanship.I congratulate my hon. Friend the Member for St Ives (Derek Thomas) and thank him for securing and introducing this very important debate. Horses are very important to me: horses brought my wife and me together, many years ago, and I have ridden many, many times. I therefore know that horses, as well as being very big and powerful, are very nervous and volatile, and consequently very unpredictable. That is a big part of this debate.I also have an

As you can see, the speeches in the images above are all relevant to the search term, and are laid out in order of semantic similarity, with the most relevant

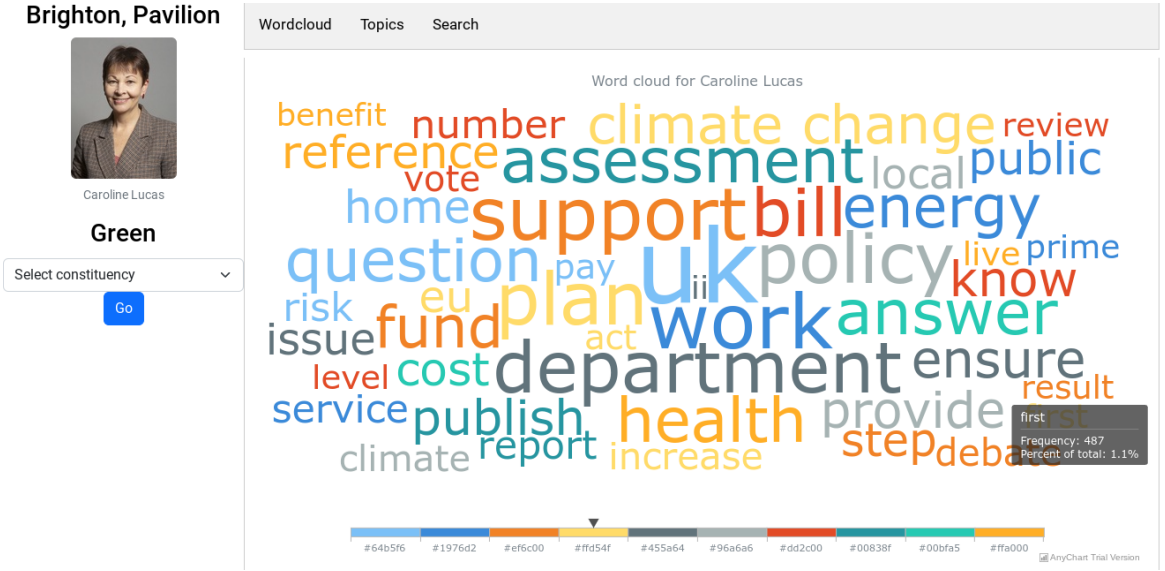
These features were all part of the initial plan for the web page, and all work as intended.

The only downside to the code is that it takes ages to load up and pre-process all the speeches, for some MPs that have thousands of speeches, it could take even longer.

Testing Various MPs

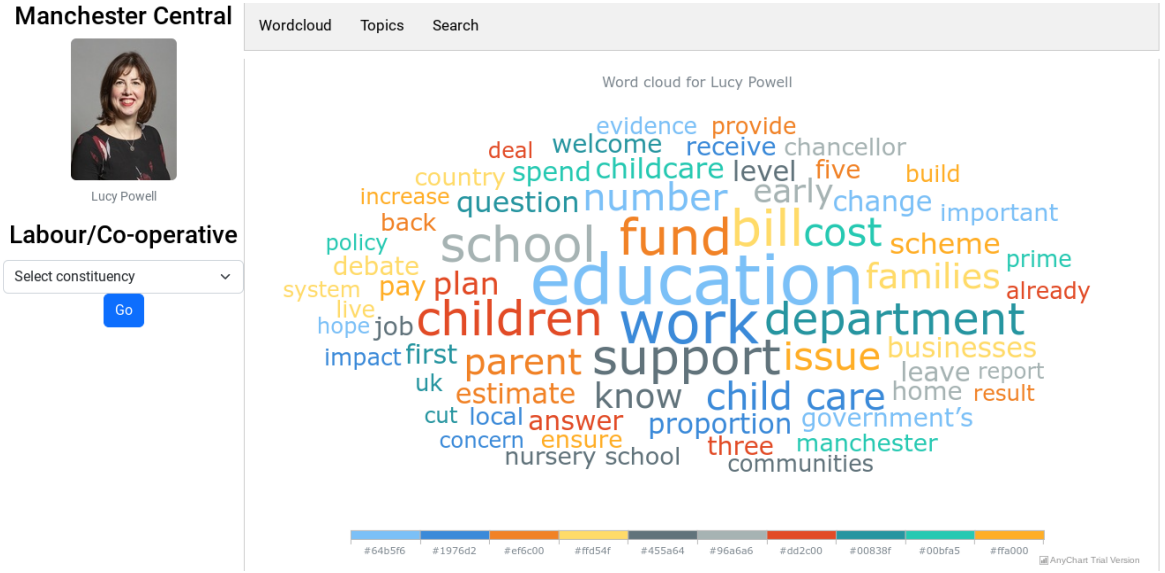
To make sure the project works properly, testing on multiple MPs must be done to ensure the code works. For example, testing the code on an MP from each of the big three parties, Conservative, Labour and Green. This shows the end user that each MP talks about different things, and that all parties stand for different things. For the testing, I will use the three MPs that have been tested on throughout the project. Those being Laurence Robertson from Tewkesbury, who is a Conservative MP, Lucy Powell from Manchester Central, who is a Labour MP, and Caroline Lucas from Brighton, who is a Green MP.

Caroline Lucas (Brighton, Pavilion)

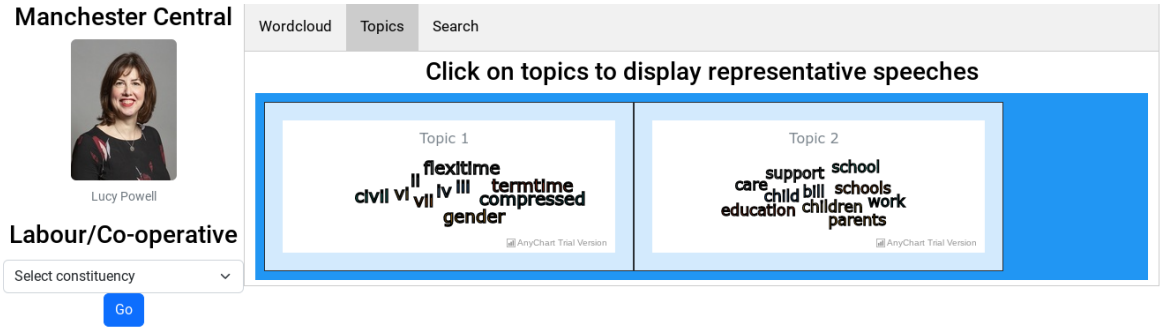


She talks more about the climate than Laurence Robertson, since she is the only Green MP, she would care more about the environment than the other MPs.

Lucy Powell (Manchester Central)



Since she is a Labour MP, she cares more about childcare and education than the other parties because Labour's focus is the people.



Lucy Powell has fewer speeches than the other MPs, so the number of topics she has are fewer. In this instance the topic modelling has not performed as expected and produced one topic that is uninformative.

Evaluation

I have created a visualisation framework for MP's speeches that fulfils the requirements set out at the start of the project. In this section I will examine how I met the objectives defined at the start of the project.

1. *Make a webpage, using the TheyWorkForYou API that is better designed and easier to use than the government Hansard website.*
    - This has been achieved using Python Flask.
- *The search function will have a dropdown menu, with a list of MPs, and what constituency they currently serve in.*
    - I implemented a dropdown list of constituencies. A user must select one of those. This is easier to navigate than selecting an MP.
- *When the user selects an MP it comes up with their picture, taken from the TheyWorkForYou API, as well as their name, and when they served in Parliament.*
    - This has been achieved.
- *It also shows their most recent speeches made in parliament, and a way to search for speeches that contain a specific word, or set of words.*
    - I chose not to display the most recent speeches, but I did implement the semantic search feature.
2. *Explore applying some modern text analysis methods to MP's speeches. This will include:*
- *Word Clouds*
- *Semantic similarity search*
- *Topic modelling*
    - These were all explored in jupyter notebook, and implemented in the web page.
3. *Solve the problem of people not knowing what their MP stands for and talks about in Parliament, and getting people to make more informed voting choices*
    - The word clouds and the topic modelling gives people a broad overview of their MPs interests.
- *The webpage will have options to present the data graphically, which is more than what the current solution has, and to make it easier for the end user to understand the data and work through it.*
    - This has been achieved.

- *The webpage will have a nice, user friendly design, that is simple to use and to understand, it will be simple to use and navigate.*
    - Use of Flask, HTML, and Javascript provides a smooth user experience when the data is loaded. Unfortunately, the web page remains slow to load data, despite my best efforts at optimising, the speed of the backend is limited by the hardware. A professional implementation would make use of GPUs for the machine learning processing.

Initial feedback from users was positive. I incorporated the following changes suggested by users:

- Added a loading screen, so the user knows the web page hasn't died
- Added instruction to the initial landing page
- Performed optimisations to speed up the backend processing

These optimisations included:

- Embedding the CSS into the HTML to reduce the page load time
- Initialise the SentenceTransformer, and the Lemmatizer inside __init__.py. This moved the slow loading of these components to the initialisation of the web page instead of happening when the user interacts with the web page

Further optimizations would have installed the Hansard data locally using an SQL database. I chose not to do this as the amount of data that would have needed to be downloaded would have been too large.

Artefacts

The video of the project in action can be found here: https://youtu.be/Z8ewyXamuIU which goes over how the project looks, and how it runs, showing all of the code and the final web page.

The code itself is housed on github https://github.com/Fredsterface/computingproject

The website is hosted on AWS at http://3.10.107.222:8080/