This guide details the procedures for scraping parliamentary records from official websites and compiling datasets of conspiracist discourses. The code included herein may be adapted for other national Parliaments, specific time periods, and different research topics unrelated to conspiracism.

WEB SCRAPPING GUIDE FOR SESSION DIARIES

DIARIES OF THE CONGRESS OF DEPUTIES: SPAIN 1931-1977

To perform web scraping of the session logs of the Spanish Congress of Deputies between 1931 and 1975, an executable code has been written in Python that uses the Selenium library to automate the downloading of PDF documents from the website (https://app.congreso.es/est_sesiones/), specifically from the historical parliamentary sessions section.

The code used is presented and explained step by step below.

In [ ]:

```
# 1. Importing libraries
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import os
import time
```

First of all, the necessary libraries are imported to automate the Chrome browser that will navigate the conference website to interact with the file system and handle the dynamic loading of elements on the page.

Selenium WebDriver enables us to control Chrome; with *service* we configure the path of the chrome controller.; *By* is used to locate items on the web; *WebDriverWait* and *expected_conditions (as EC)* are used to wait for elements to load or interact; with *os* we interact with the file system (creating folders, for example); Finally, *time* allows us to pause (with *sleep*), which is necessary to give the website time to load correctly.

In [ ]:

```
# 2. Download folder settings
download_path = "D:\\DIARIOS\\31-77\\71-77"
if not os.path.exists(download_path):
   os.makedirs(download_path)
```

In this step, we define the folder where the downloaded files will be saved and create it if it doesn't exist. In our case, the folder was located on an external hard drive, as shown in the path.

In [ ]:

```
# 3. Configuring options for Chrome
chrome_options = webdriver.ChromeOptions()
chrome_options.add_experimental_option("prefs", {
   "download.default_directory": download_path,
   "download.prompt_for_download": False,
   "download.directory_upgrade": True,
   "safebrowsing.enabled": True
})
```

This configures the browser to download PDF files directly to the specified folder and, above all, allows downloads without user intervention and without asking questions or displaying security warnings before downloading, thereby speeding up the process.

In [ ]:

```
# 4. Chromedriver path and initialisation
chromedriver_path = 'C:\\Users\\Usuario\\Desktop\\chromedriver-win64\\chromedriver.exe'
service = Service(chromedriver_path)
driver = webdriver.Chrome(service=service, options=chrome_options)
```

This defines the path where the *chromedriver* executable is located and launches it, i.e. it opens a browser emulator that will

show how it progresses through the page in that download path process.

In [ ]:
```
# 5. Open the Congress website
driver.get('https://app.congreso.es/est_sesiones/')
try:
   legislaturas = WebDriverWait(driver, 10).until(
       EC.presence_of_all_elements_located((By.CLASS_NAME, 'list-group-item'))
   )
except Exception as e:
   print(f"Error al cargar las legislaturas: {e}")
```

Once the browser emulator has started, this line of code opens the page with the parliamentary sessions and waits until the available legislatures are visible on it, i.e. it waits until the list-group-item elements, which represent the legislatures, and if it doesn't load within 10 seconds, it prints an error message.

In [ ]:
```
# 7. Select the starting legislature and expansions
start_legislatura = "Legislatura 1971-1977"
start_clicking = False
def expand_all():
   while True:
       expanders = driver.find_elements(By.CLASS_NAME, "expandable-hitarea")
       new_clicks = 0
       for expander in expanders:
           if expander.is_displayed():
               try:
                   driver.execute_script("arguments[0].scrollIntoView();", expander)
                   time.sleep(0.5)
                   expander.click()
                   print("Expandiendo contenido adicional...")
                   time.sleep(1)
                   new_clicks += 1
               except Exception as e:
                   print(f"No se pudo expandir: {e}")
       if new_clicks == 0:
           break
```

Define the legislative session from which the documents should be downloaded and automatically click on all the '+' symbols (expandable-hitarea type) that expand content within a legislative session. In addition, use a loop to repeat the process until there are no more expandable elements.

In [ ]:
```
# 8. Going through all the other legislatures.
for legislatura in legislaturas:
   if start_clicking or start_legislatura in legislatura.text:
       start_clicking = True
       legislatura.click()
       print(f"Clic en: {legislatura.text}")
       time.sleep(1)

       # Expand all levels of +
       expand_all()

       # Wait and collect the documents
       try:
           indices = WebDriverWait(driver, 10).until(
               EC.presence_of_all_elements_located((By.CLASS_NAME, "file"))
           )
       except Exception as e:
           print(f"No se encontraron índices para {legislatura.text}. Error: {e}")
           continue
```

With these lines of code, all the legislatures found are scanned, and when the legislature marked as the start is found, it activates the processing of clicking on the selected legislature and begins to process the following ones; then, it waits for the documents (file) to appear, and if not, it moves on to the next legislature.

In [ ]:
```
# 9. Download documents
```

```
for indice in indices:
            try:
                indice_texto = indice.text
                driver.execute_script("arguments[0].scrollIntoView();", indice)
                WebDriverWait(driver, 10).until(EC.element_to_be_clickable(indice)).click()
                print(f"Haciendo clic en: {indice_texto}")
                 WebDriverWait(driver, 5).until(
                    EC.presence_of_element_located((By.TAG_NAME, "iframe"))
                )
                print("Visor PDF cargado.")

                iframes = driver.find_elements(By.TAG_NAME, "iframe")
                if iframes:
                    driver.switch_to.frame(iframes[0])
                  try:
                    download_button = WebDriverWait(driver, 5).until(
                        EC.element_to_be_clickable((By.ID, "download"))
                    )
                    driver.execute_script("arguments[0].scrollIntoView();", download_button)
                    download_button.click()
                    print(f"Descargando: {indice_texto}")
                    time.sleep(2)
                except Exception:
                    print(f" No se encontró botón de descarga para {indice_texto}")

                driver.switch_to.default_content()
                time.sleep(1)
            except Exception as e:
                print(f"Error al procesar el índice {indice_texto}: {e}")
```

In this section of the code, the browser scrolls down to the document and clicks on it; then, it waits for the PDF viewer (an iframe) to load and switches to it to click on the download button (id='download'); finally, it returns to the main context to continue clicking on the following PDFs and/or legislatures.

In [ ]:

```
# 10. Close the browser
driver.quit()
```

Finally, close the browser and terminate the script.

DIARIES OF THE CONGRESS OF DEPUTIES: SPAIN 1977 - Present

In this case, in order to scrape the session logs of the Spanish Congress of Deputies from 1977 to the present, the previous executable code has been adapted to the new page where these logs are located (https://www.congreso.es/es/busqueda-de-publicaciones).

The adapted code is presented and explained step by step below.

In [ ]:

```
# 1.  Importing libraries
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time
import os
import requests
from selenium.webdriver.support.ui import Select
```

These libraries allow you to: control the browser with Selenium; select elements (By and Select); Pause to wait for elements to load and become available with *WebDriverWait* and *expected_conditions (as EC)*; use advanced browser options thanks to *Options* and *Service*; Use *requests* to download files with custom headers; create folders (with *os*) and pause (with *time*).

In [ ]:

```
# 2. Configuring options for Chrome
```

```
options = Options()
options.headless = False  # Change to True if you don't want to see the browser
chromedriver_path = 'C:/Users/irene/OneDrive/Escritorio/chromedriver-win64/chromedriver.exe'
service = Service(chromedriver_path)
driver = webdriver.Chrome(service=service, options=options)
```

So, we configure Chrome with Selenium. In addition, eadless=False displays the browser, and if we change it to True, it runs in the background or invisible mode. Then, it defines the path of the chromedriver executable and launches Chrome with all those settings.

In [ ]:

```
# 3. Open the search page
url = "https://www.congreso.es/es/busqueda-de-publicaciones"
driver.get(url)
time.sleep(3)
```

Load the Congress page and wait 3 seconds for everything to load correctly.

In [ ]:

```
# 4. Selection of the legislature
def seleccionar_legislatura(legislatura_value):
  try:
      select_element = WebDriverWait(driver, 10).until(
          EC.presence_of_element_located((By.ID, "_publicaciones_legislatura"))
      )
      select = Select(select_element)
      select.select_by_value(legislatura_value)
      time.sleep(3)
      print(f" Legislatura {legislatura_value} seleccionada.")
  except Exception as e:
      print(f" Error al seleccionar legislatura: {str(e)}")
```

In this way, we wait for the drop-down menu of legislatures to appear, use *Select* to choose the option with the passed value, and pause for the content to load.

In [ ]:

```
# 5. Apply advanced filters
def aplicar_filtros_avanzados(fecha):
  try:
      filtros_avanzados_button = WebDriverWait(driver, 10).until(
          EC.element_to_be_clickable((By.ID, "headingFilter2"))
      )
      filtros_avanzados_button.click()
      print(" Filtros avanzados desplegados.")

      time.sleep(2)
      fechas_button = WebDriverWait(driver, 10).until(
          EC.element_to_be_clickable((By.ID, "headingThree2"))
      )
      fechas_button.click()
      print(" Sección 'Fechas' desplegada.")

      time.sleep(2)
      fecha_desde_input = WebDriverWait(driver, 10).until(
          EC.presence_of_element_located((By.ID, "_publicaciones_dateDesdeM"))
      )
      fecha_desde_input.clear()
      fecha_desde_input.send_keys(fecha)
      print(f" Fecha establecida a {fecha}.")

      buscar_button = WebDriverWait(driver, 10).until(
          EC.element_to_be_clickable((By.XPATH, '//span[@class="lfr-btn-label" and text()="Buscar"]'))
      )
      buscar_button.click()
      print(" Botón 'Buscar' clickeado.")

      time.sleep(3)
  except Exception as e:
      print(f" Error al aplicar filtros avanzados: {str(e)}")
```

With this function, we can display the advanced filter panel to edit the date from which to show results. To do this, enter a date in the 'Date from' field and then click on the 'Search' button. All this is done with *WebDriverWait* to ensure that the elements are ready before interacting with them.

In [ ]:

```
# 6. Funcion for downloading documents
def descargar_pdf(url_pdf, nombre_archivo):
  try:
      headers = {
          'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/91.0.4472.124 Safari/537.36',
          'Accept': 'application/pdf',
          'Accept-Encoding': 'gzip, deflate, br',
          'Connection': 'keep-alive',
          'Upgrade-Insecure-Requests': '1',
      }

      response = requests.get(url_pdf, headers=headers)

      if response.status_code == 200:
          if 'application/pdf' in response.headers.get('Content-Type', ''):
              with open(nombre_archivo, "wb") as file:
                  file.write(response.content)
              print(f" Descargado: {nombre_archivo}")
          else:
              print(f" El contenido no es un archivo PDF: {url_pdf}")
      else:
          print(f" Error al descargar (status code {response.status_code}): {url_pdf}")

  except Exception as e:
      print(f" Error al descargar {url_pdf}: {str(e)}")
```

Thanks to this function, and using *requests.get()*, we download the corresponding file in each case. Add a User-Agent and other headers to avoid being blocked by the server, check if the content is a PDF, and, if everything goes well, save the file with the specified name.

In [ ]:

```
# 7. Download folder
download_path = "E:\\DIARIOS"
os.makedirs(download_path, exist_ok=True)
```

With these lines, we define the folder where the downloaded files will be saved and create it if it doesn't exist. In our case, as before, the folder was located on an external hard drive, as can be seen in the path.

In [ ]:

```
# 8. Filters applied for browsing pages
seleccionar_legislatura("15")
aplicar_filtros_avanzados("23/01/2024")
```

This allows us to edit the legislative period we want to select, as well as a publication filter to mark the date from which we want the download to start. This makes it easier to correct the download in case of any errors and start again without having to repeat work already done.

In [ ]:

```
# 9. Download documents
while True:
  time.sleep(2)
  enlaces_pdf = driver.find_elements(By.LINK_TEXT, "PDF")

  if enlaces_pdf:
      print(f" Encontrados {len(enlaces_pdf)} PDFs en esta página.")
  else:
      print(" No se encontraron PDFs en esta página.")

  for enlace in enlaces_pdf:
      url_pdf = enlace.get_attribute("href")
      if url_pdf:
          if not url_pdf.startswith("http"):
              url_pdf = "https://www.congreso.es" + url_pdf
```

```
            print(f" Descargando: {url_pdf}")
            nombre_archivo = os.path.join(download_path, os.path.basename(url_pdf))
            descargar_pdf(url_pdf, nombre_archivo)

    try:
        boton_siguiente = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.XPATH, '//a[@class="page-link btn_pag" and text()=">"]'))
        )
        print(" Clic en 'Siguiente'")
        boton_siguiente.click()
        time.sleep(5)

        siguiente_pagina = driver.find_elements(By.XPATH, '//a[@class="page-link btn_pag" and text()=">"]')
        if not siguiente_pagina:
            print(" No hay más páginas, fin del proceso.")
            break
    except Exception as e:
        print(f" Error al hacer clic en el botón 'Siguiente': {str(e)}")
        break
```

With this loop, we automate the download by following these steps: first, wait 2 seconds; then, search for links with the text 'PDF'; go through each link and download the file using the *descargar_pdf* function and correct the link if it does not start with http; Next, download the file and name it accordingly. Finally, look for a '>' button (next page), click on it, and repeat until you can't find any more, then end the loop.

In [ ]:

```
# 10. Close the browser
driver.quit()
```

DIARIES OF THE CONGRESS OF DEPUTIES: Czech Republic

To scrape the session logs of the Czech Republic Parliament, we have once again adapted the code to the page where the logs are located (https://www.psp.cz/sqw/tisky.sqw?o=9&za=0). As in previous cases, Selenium is used to automate the downloading of PDF documents from the website.

The adapted code is presented and explained step by step below.

In [ ]:

```
# 1. Importing libraries
import os
import time
import requests
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from webdriver_manager.chrome import ChromeDriverManager
from urllib.parse import urljoin
import re
```

We import the libraries: *os* and *time* to handle paths and pauses; *requests* to download files; *selenium*, as mentioned above, to automate web browsing; *webdriver_manager* automatically installs the appropriate ChromeDriver; *urljoin* is useful if you want to combine URLs; and *re* to clean up file names.

In [ ]:

```
# 2. Configuring the base URL and download folder (+ avoiding duplicate links)
BASE_URL = "https://www.psp.cz/sqw/tisky.sqw?o=9&za=0"
DOWNLOAD_PATH = "D:\\DIARIOS\\RCH\\2021"
os.makedirs(DOWNLOAD_PATH, exist_ok=True)
visited_urls = set()
```

BASE_URL indicates where the crawl starts; then, with *DOWNLOAD_PATH*, we define the path of the folder where the PDFs are saved and, if it does not exist, we create it using *os.makedirs(…, exist_ok=True)*.

In addition, with *visited_urls = set()*, we create a set to keep track of the links already visited and avoid repeating downloads.

In [ ]:

```
# 3. Configuring the browser
def setup_browser():
    chrome_options = Options()
    chrome_options.add_argument("--no-sandbox")
    chrome_options.add_argument("--disable-dev-shm-usage")
    driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()), options=chrome_options)
    return driver

def sanitize_filename(filename):
    return re.sub(r'[\\/*?:"<>|]', "_", filename)
```

This is how we configure a Chrome browser in automated mode. Furthermore, with *no-sandbox* and *disable-dev-shm-usage* we avoid problems in other environments such as Linux. With *webdriver_manager* we avoid having to download ChromeDriver manually.

Finally, with *sanitise_filename(filename)* we prevent the file name from containing characters that are not permitted by the operating system, replacing characters that may cause errors when saving files (like *, ?, :) with underscores.

In [ ]:

```
# 4. Access to a subpage and location of PDFs
def download_pdf_files(driver, page_url):
    try:
        driver.get(page_url)
        time.sleep(2)
    except Exception as e:
        print(f" Error al acceder a la página {page_url}: {e}")
        return

    pdf_links = driver.find_elements(By.XPATH, '//a[contains(@href, "orig2.sqw?idd=")]')
    for pdf_link in pdf_links:
        pdf_url = pdf_link.get_attribute('href')
        download_file(pdf_url)
```

With this function, we enter a specific subpage and search for links that lead to PDF files.

In [ ]:

```
# 5. Download a specific PDFs
def download_file(file_url):
    file_name = os.path.join(DOWNLOAD_PATH, sanitize_filename(os.path.basename(file_url)) + ".pdf")

    if not os.path.exists(file_name):
        try:
            print(f" Descargando: {file_url}")
            file_response = requests.get(file_url, stream=True, timeout=15)
            file_response.raise_for_status()

            with open(file_name, "wb") as f:
                for chunk in file_response.iter_content(chunk_size=1024):
                    f.write(chunk)
            print(f" Descargado: {file_name}")
        except requests.RequestException as e:
            print(f" Error al descargar {file_url}: {e}")
    else:
        print(f" Archivo ya descargado: {file_name}")
```

This function aims to download a PDF file from a URL and save it to disk, avoiding doing so if the file already exists.

In [ ]:

```
# 6. Obtain links to documents on a page
def get_links_on_page(driver):
    links = []
    try:
        link_elements = driver.find_elements(By.XPATH, '//a[contains(@href, "tiskt.sqw")]')
        for link in link_elements:
            href = link.get_attribute('href')
            if href and href not in visited_urls:
                links.append(href)
                visited_urls.add(href)
    except Exception as e:
        print(f" Error al obtener enlaces: {e}")
```

```
    return links
```

On the main results page, we search for all links to document details and only add those that we have not yet visited.

In [ ]:

```
# 7.Go through each link, download and go back.
def process_links(driver, current_page_url):
    links = get_links_on_page(driver)
    for link in links:
        print(f" Explorando: {link}")
        driver.get(link)
        time.sleep(2)
        download_pdf_files(driver, link)
        driver.get(current_page_url)
        time.sleep(2)
```

We go to each page of the document, download the PDF files, and then return to the previous page to continue with the rest.

In [ ]:

```
# 8. Define the base URL, link processing, and next button.
def navigate_and_download(driver):
    driver.get(BASE_URL)
    time.sleep(2)
    current_page_url = driver.current_url
    process_links(driver, current_page_url)

    while True:
        try:
            next_button = driver.find_element(By.XPATH, '//a[contains(@class, "next")]')
            next_page_url = next_button.get_attribute('href')
            print(f" Pasando a la siguiente página: {next_page_url}")
            driver.get(next_page_url)
            time.sleep(2)
            process_links(driver, next_page_url)
        except Exception as e:
            print(f" No se pudo encontrar el botón siguiente o se ha llegado al final: {e}")
            break
```

With the function *def navigate_and_download(driver)*, we define starting from the base URL, process the links, and then move forward page by page using the 'Next' button until there are no more pages.

In [ ]:

```
# 9. Starting point of the entire process
def crawl():
    driver = setup_browser()
    navigate_and_download(driver)
    driver.quit()
```

These lines are the command centre created by the browser. They start the process and close it when everything is finished with *driver.quit()*.

In [ ]:

```
# 10. Run the script
crawl()
print(" Descarga completada.")
```

We run the entire script and add a confirmation message to appear when the download is complete.

DIARIES OF THE EUROPEAN PARLIAMENT

In [ ]:

```
# 1. Importing libraries
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
```

```
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.support.ui import WebDriverWait
import time
import os
from urllib.parse import urljoin
from webdriver_manager.chrome import ChromeDriverManager
import requests
```

Imports all necessary libraries to automate Chrome, interact with web elements, handle waits, manage file paths, fetch PDF files, and control time delays. Selenium controls the browser, requests downloads PDFs, and os handles the filesystem.

In [ ]:

```
# 2. Configuring the base URL and download folder
DOWNLOAD_PATH = "D:/EUROPEAN_DIARIES"
os.makedirs(DOWNLOAD_PATH, exist_ok=True)

options = Options()
options.headless = False  # Run with visible browser
options.add_argument("--start-maximized")  # Start maximized

driver = webdriver.Chrome(
    service=Service(ChromeDriverManager().install()),
    options=options
)
wait = WebDriverWait(driver, 30)

BASE_URL = "https://www.europarl.europa.eu/plenary/en/minutes.html"
```

Creates the download folder and sets base URL for the European Parliament plenary minutes page. WebDriverWait is configured for waiting elements.

In [ ]:

```
# 3. PDF Download function
def download_pdf(pdf_url):
    file_name = os.path.join(DOWNLOAD_PATH, pdf_url.split("/")[-1])

    if os.path.exists(file_name):
        print("Already downloaded:", file_name)
        return

    r = requests.get(pdf_url, headers={"User-Agent": "Mozilla/5.0"})
    if r.status_code == 200:
        with open(file_name, "wb") as f:
            f.write(r.content)
        print("PDF downloaded:", file_name)
    else:
        print("Error downloading:", pdf_url)
```

Downloads a PDF from the provided URL if it doesn't already exist. Uses requests with a user-agent to mimic a browser and writes the PDF to DOWNLOAD_PATH.

In [ ]:

```
# 4. Zoom function
def apply_zoom(driver, percentage="75%"):
    driver.execute_script(f"document.body.style.zoom='{percentage}'")
```

Applies a zoom level to the webpage via JavaScript. Useful for adjusting view when elements are small or to make scraping more reliable.

In [ ]:

```
# 5. Open web page
driver.get(BASE_URL)
time.sleep(3)
apply_zoom(driver)
```

Opens the European Parliament plenary minutes page and waits 3 seconds for it to load. Then applies zoom using the previously defined function.

In [ ]:

```
# 6. Select legislature
driver.execute_script("""
document.querySelectorAll('span.ui-selectmenu-status')
    .forEach(el => el.innerText = '2024 - 2029');
document.querySelectorAll('input[type="hidden"]').forEach(el => {
    if (el.name && el.name.toLowerCase().includes('term')) {
        el.value = '2024-2029';
        el.dispatchEvent(new Event('change', { bubbles: true }));
    }
});
""")
time.sleep(1)
```

Sets the legislature term to “2024 - 2029” by modifying page elements via JavaScript. Updates both visible status and hidden input values to simulate a user selection.

In [ ]:

```
# 7. Set date filters
wait.until(lambda d: d.find_element(By.ID, "refSittingDateStart")).send_keys("01/01/2025")
wait.until(lambda d: d.find_element(By.ID, "refSittingDateEnd")).send_keys("31/01/2025")
wait.until(lambda d: d.find_element(By.ID, "sidesButtonSubmit")).click()
time.sleep(4)
```

Sets the date range filter for the plenary minutes (1–31 January 2025) and submits the form. Waits 4 seconds for the results to load.

In [ ]:

```
# 8. First result
first = wait.until(lambda d: d.find_element(
    By.XPATH, "//div[@class='notice']/p[@class='title']/a[1]"
))
driver.get(first.get_attribute("href"))
apply_zoom(driver)
time.sleep(1)
```

Navigates to the first search result and applies zoom. Waits 1 second to ensure the page loads before interacting with it.

In [ ]:

```
# 9. First download
# ============================
wait.until(lambda d: d.find_element(By.LINK_TEXT, "Index")).click()
wait.until(lambda d: d.find_element(By.LINK_TEXT, "Verbatim reports")).click()
time.sleep(0.5)

pdfs = driver.find_elements(By.XPATH, "//a[contains(text(),'Verbatim report of proceedings')]")
if pdfs:
    pdf_url = urljoin("https://www.europarl.europa.eu", pdfs[-1].get_attribute("href"))
    download_pdf(pdf_url)
else:
    print(" No PDFs found on the first day")
```

Clicks through to the “Index” and “Verbatim reports” pages, finds all PDFs labeled “Verbatim report of proceedings”, and downloads the last one (assumed final version). Prints a warning if none are found.

In [ ]:

```
# 10. Loop through days
while True:
    try:
        # Find Next button
        next_btns = driver.find_elements(By.LINK_TEXT, "Next")
        if not next_btns:
            print("No more Next buttons, finishing.")
            break

        # Click Next
        next_btns[0].click()
        time.sleep(0.5)
        apply_zoom(driver)
```

```
        # Download the last PDF of the new day
        pdfs = driver.find_elements(By.XPATH, "//a[contains(text(),'Verbatim report of proceedings')]")
        if pdfs:
            pdf_url = urljoin("https://www.europarl.europa.eu", pdfs[-1].get_attribute("href"))
            download_pdf(pdf_url)
        else:
            print(" No PDFs found on this day")
    except Exception as e:
        print(" END:", e)
        break

driver.quit()
```

Loops through all result pages by clicking “Next”, zooms the page, and downloads the last PDF of each day. Exits the loop when no more pages are found or if an error occurs. Closes the driver at the end.

DIARIES OF THE AUSTRIAN PARLIAMENT

In [ ]:

```
# 1. Importing libraries
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from webdriver_manager.chrome import ChromeDriverManager
import os
import time
import requests
```

Imports all necessary libraries to automate Chrome, interact with web elements, handle waits, manage file paths, fetch PDF files, and control time delays. Selenium controls the browser, requests downloads PDFs, and os handles the filesystem.

In [ ]:

```
# 2. Browser setup and configuration
options = Options()
# options.add_argument("--headless")

service = Service(ChromeDriverManager().install())

driver = webdriver.Chrome(service=service, options=options)

options = Options()
options.headless = False  # True = modo invisible
```

Configures and launches a Chrome browser instance using Selenium, automatically installing the correct ChromeDriver and optionally allowing headless execution.

In [ ]:

```
# 3. Open website
url = "https://www.parlament.gv.at/recherchieren/protokolle/?STENO_211utm_source=chatgpt.com"
driver.get(url)
time.sleep(3)
```

Navigates to the target website and pauses execution briefly to allow the page to fully load before interacting with it.

In [ ]:

```
# 4. Download Directory
download_path = "D:\\DIARIOS\\Austria\\XV"
os.makedirs(download_path, exist_ok=True)
```

Creates a local folder where downloaded PDF files will be stored, ensuring it exists before any downloads begin.

In [ ]:

```
# 5. Accept Cookies
def aceptar_cookies():
    try:
        boton = WebDriverWait(driver, 5).until(
            EC.element_to_be_clickable((By.XPATH, '//button[contains(text(), "Alle Akzeptieren")]'))
        )
        boton.click()
        print("Cookies accepted.")
        time.sleep(1)
    except:
        print("No cookie banner appeared.")
```

Waits for the cookie consent button to appear and clicks it if present, allowing the script to proceed without interruption from the cookie banner.

In [ ]:

```
# 6. Apply Date Filter
def aplicar_filtro_fecha(fecha_desde, fecha_hasta):
    try:
        # A. Open date filter
        boton_fecha = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.XPATH, '//button[contains(text(), "Datum von-bis")]'))
        )
        boton_fecha.click()
        print("Filtro de fecha abierto.")
        time.sleep(2)

        # B. Start date
        campo_inicio = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.ID, 'fromDatePicker'))
        )
        campo_inicio.click()
        time.sleep(0.5)
        campo_inicio.clear()
        campo_inicio.send_keys(fecha_desde.replace(".", ""))
        print(f"Start date: {fecha_desde}")
        time.sleep(1)

        # C. Search button
        boton_buscar = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.XPATH, '//button[contains(text(), "Treffer anzeigen")]'))
        )
        boton_buscar.click()
        print("Filter applied with start date.")
        time.sleep(4)

        # D. Reopen the date filter for the end date
        boton_fecha = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.XPATH, '//button[contains(text(), "Datum von-bis")]'))
        )
        boton_fecha.click()
        print("Date filter open for end date.")
        time.sleep(2)

        # E. End date
        campo_fin = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.ID, 'toDatePicker'))
        )
        campo_fin.click()
        time.sleep(0.5)
        campo_fin.clear()
        campo_fin.send_keys(fecha_hasta.replace(".", ""))
        print(f"End date: {fecha_hasta}")
        time.sleep(1)

        # F. Search button (with start date and end date)
        boton_buscar = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.XPATH, '//button[contains(text(), "Treffer anzeigen")]'))
        )
        boton_buscar.click()
        print("Filter applied with an end date.")
        time.sleep(4)

    except Exception as e:
        print(f"Error with the date filter by sections: {e}")
```

Opens the date filter menu, inputs a start and end date, and applies the filter step by step to refine the search results shown on the page.

In [ ]:

```
# 7. Download Individual PDF
def descargar_pdf(url_pdf, nombre_archivo):
    try:
        headers = {
            'User-Agent': 'Mozilla/5.0',
            'Accept': 'application/pdf',
        }
        response = requests.get(url_pdf, headers=headers)
        if response.status_code == 200 and 'application/pdf' in response.headers.get('Content-Type', ''):
            with open(nombre_archivo, "wb") as file:
                file.write(response.content)
            print(f" Downloaded: {nombre_archivo}")
        else:
            print(f" This is not a valid PDF: {url_pdf}")
    except Exception as e:
        print(f" Error downloading {url_pdf}: {str(e)}")
```

Sends an HTTP request to retrieve a PDF file from a given URL and saves it locally only if the response confirms it is a valid PDF.

In [ ]:

```
# 8. Extract and Download PDFs from Page
def descargar_pdfs_de_pagina():
    time.sleep(2)
    enlaces_pdf = driver.find_elements(By.XPATH, '//a[contains(@href, ".pdf")]')
    urls = set()
    for enlace in enlaces_pdf:
        url_pdf = enlace.get_attribute("href")
        if url_pdf and url_pdf.lower().endswith('.pdf') and url_pdf not in urls:
            urls.add(url_pdf)
            nombre_archivo = os.path.basename(url_pdf)
            ruta_guardado = os.path.join(download_path, nombre_archivo)
            print(f" Downloading: {url_pdf}")
            descargar_pdf(url_pdf, ruta_guardado)
```

Finds all PDF links on the current page, removes duplicates, and iterates through them to download each file into the specified folder.

In [ ]:

```
# 9. Pagination Handling
def obtener_pagina_actual_y_total():
    try:
        input_pagina = WebDriverWait(driver, 5).until(
            EC.presence_of_element_located((By.ID, "STENO_211-pageinput-num"))
        )
        valor_pagina = input_pagina.get_attribute('value') or input_pagina.get_attribute('placeholder') or '1'
        pagina_actual = int(valor_pagina)

        texto_total = WebDriverWait(driver, 5).until(
            EC.presence_of_element_located((By.CSS_SELECTOR, 'p.text.svelte-s20rjt'))
        ).text.strip()

        total_paginas = int(texto_total.split("von")[-1].strip())

        return pagina_actual, total_paginas
    except Exception as e:
        print(" No se pudo obtener la página actual y total:", e)
        return None, None

def ir_a_siguiente_pagina():
    pagina_actual, total_paginas = obtener_pagina_actual_y_total()
    if pagina_actual is None or total_paginas is None:
        print(" No se pudo obtener info de paginación, intentando avanzar de todos modos.")
    else:
        if pagina_actual >= total_paginas:
            print(f" Última página ({pagina_actual}) alcanzada.")
            return False
```

```
        else:
            print(f" Pasando a página {pagina_actual + 1} de {total_paginas}")

    try:
        boton_icono = WebDriverWait(driver, 5).until(
            EC.element_to_be_clickable((By.XPATH, '//i[contains(@class, "fa-chevron-right")]'))
        )
        boton_padre = boton_icono.find_element(By.XPATH, '..')

        boton_padre.click()
        time.sleep(3)
        return True
    except Exception as e:
        print(" No se pudo avanzar a siguiente página:", e)
        return False
```

Reads the current page number and total number of pages, then clicks the “next page” button to continue scraping until the last page is reached.

In [ ]:

```
# 10. Main Execution
aceptar_cookies()

fecha_inicio = "05.06.1979"
fecha_fin = "18.05.1983"
aplicar_filtro_fecha(fecha_inicio, fecha_fin)

while True:
    descargar_pdfs_de_pagina()
    if not ir_a_siguiente_pagina():
        print(" The execution ends because there are no more pages.")
        break

driver.quit()
```

Executes the full workflow by accepting cookies, applying the date filter, iterating through all result pages, downloading every PDF found, and stopping once no more pages remain before closing the browser.

DIARIES OF THE GERMAN PARLIAMENT

In [ ]:

```
# 1. Importing libraries
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from webdriver_manager.chrome import ChromeDriverManager
import time
import os
import requests
```

Import the libraries required to automate the web browser, locate and interact with elements on the Bundestag website, download PDF files and manage the downloaded files. Selenium is used for browser automation, webdriver_manager for managing ChromeDriver, while requests, os and time are used for downloading files, managing directories and controlling waiting times.

In [ ]:

```
# 2. Configuring the browser
options = Options()
options.headless = False  # Cambiar a True si no quieres ver el navegador

service = Service(ChromeDriverManager().install())
driver = webdriver.Chrome(service=service, options=options)
```

Configure and initialise the Chrome browser through Selenium. ChromeDriver is automatically installed and managed using webdriver_manager. The browser is configured in non-headless mode, allowing the scraping process to be visually monitored.

In [ ]:

```
# 3. Accessing the Bundestag parliamentary records
url = "https://www.bundestag.de/dokumente/protokolle/plenarprotokolle"
driver.get(url)
time.sleep(3)
```

Access the official German Bundestag webpage containing the plenary parliamentary protocols. A short delay is introduced to allow the webpage and its content to load before further interactions are performed.

In [ ]:

```
# 4. Defining the output directory
download_path = "D:\\TRABAJO\\DIARIOS\\Alemania\\2010"
os.makedirs(download_path, exist_ok=True)
```

Define the local directory in which the parliamentary PDF files will be stored. The directory is automatically created if it does not already exist. The output path is adjusted for each year of the parliamentary corpus.

In [ ]:

```
# 5. Downloading individual PDF files
def descargar_pdf(url_pdf, nombre_archivo):
    try:
        headers = {
            'User-Agent': 'Mozilla/5.0 (...)',
            'Accept': 'application/pdf',
        }

        response = requests.get(url_pdf, headers=headers)

        if response.status_code == 200 and \
           'application/pdf' in response.headers.get('Content-Type', ''):

            with open(nombre_archivo, "wb") as file:
                file.write(response.content)

            print(f" Descargado: {nombre_archivo}")

        else:
            print(f" No es un PDF válido: {url_pdf}")

    except Exception as e:
        print(f" Error al descargar {url_pdf}: {str(e)}")
```

Define a function to download individual parliamentary PDF files from their URLs. The request includes browser-like headers and verifies that the response is successful and corresponds to a PDF document before saving the file to the designated directory. Download errors and invalid responses are reported without interrupting the process.

In [ ]:

```
# 6. Applying the date filter
def aplicar_filtro_fecha(fecha_desde, fecha_hasta):
    try:
        input_desde = WebDriverWait(driver, 10).until(
            EC.presence_of_element_located(
                (By.ID, "from_440434-442112")
            )
        )

        input_desde.clear()
        input_desde.send_keys(fecha_desde)
```

Locate the starting-date field on the Bundestag webpage and enter the beginning of the desired period. WebDriverWait is used to ensure that the element is available before interacting with it.

In [ ]:

```
# 7. Defining the end date
input_hasta = WebDriverWait(driver, 10).until(
    EC.presence_of_element_located(
        (By.ID, "to_440434-442112")
    )
)
```

```
input_hasta.clear()
input_hasta.send_keys(fecha_hasta)
```

Locate the ending-date field and enter the final date of the selected period. Together with the starting date, this defines the temporal range of parliamentary records to be retrieved.

In [ ]:

```
# 8. Waiting for the filtered results
time.sleep(5)
```

Wait for the Bundestag webpage to update the results after the date range has been entered.

In [ ]:

```
# 9. Identifying PDF links on each page
def descargar_pdfs_de_pagina():
    time.sleep(2)
    urls_descargadas = set()

    enlaces = driver.find_elements(
        By.XPATH,
        '//a[contains(@href, ".pdf")]'
    )
```

Identify all links on the current Bundestag webpage whose URLs point to PDF documents. A set is created to keep track of URLs already identified during the current page.

In [ ]:

```
# 10. Extracting the PDF URLs
enlaces_pdf = []

for e in enlaces:
    href = e.get_attribute("href")

    if href and href.lower().endswith('.pdf'):
        enlaces_pdf.append(href)
```

Extract the URLs of the identified PDF documents and retain only links whose addresses explicitly end in .pdf. The URLs are collected before the download process begins to avoid problems caused by changes to the webpage during navigation.

In [ ]:

```
# 11. Downloading the PDFs from the current page
for url_pdf in enlaces_pdf:
    if url_pdf not in urls_descargadas:
        urls_descargadas.add(url_pdf)

        nombre_archivo = os.path.basename(url_pdf)
        ruta_guardado = os.path.join(
            download_path,
            nombre_archivo
        )

        descargar_pdf(url_pdf, ruta_guardado)
```

Download each unique PDF identified on the current page. The original filename is extracted from the URL and combined with the predefined output directory before the document is downloaded.

In [ ]:

```
# 12. Navigating to the next page
def ir_a_siguiente_pagina():
    try:
        botones = WebDriverWait(driver, 10).until(
            EC.presence_of_all_elements_located(
                (By.XPATH, "//i[contains(@class,'icon-chevron')]")
            )
        )
```

Identify the navigation controls used to move between pages of parliamentary records. The script waits until the relevant navigation elements are available on the webpage.

In [ ]:

```
# 13. Selecting and activating the next-page button
boton_siguiente = botones[-1]

boton_clickable = boton_siguiente.find_element(
    By.XPATH,
    "./ancestor::button | ./ancestor::a | ./parent::*"
)

driver.execute_script(
    "arguments[0].scrollIntoView(true);",
    boton_clickable
)

time.sleep(1)

driver.execute_script(
    "arguments[0].click();",
    boton_clickable
)
```

Select the last navigation control, which corresponds to the next-page button, and identify its clickable parent element. The browser scrolls to the element before triggering a JavaScript click to move to the next page of parliamentary records.

In [ ]:

```
# 14. Defining the date range
fecha_inicio = "01.01.2010"
fecha_fin = "31.12.2010"
```

Define the temporal boundaries of the parliamentary records to be collected. Dates are entered using the German website's DD.MM.YYYY format. The parameters are adjusted for each year included in the corpus.

In [ ]:

```
# 15. Applying the filter and starting the scraping process
aplicar_filtro_fecha(fecha_inicio, fecha_fin)

while True:
    descargar_pdfs_de_pagina()

    if not ir_a_siguiente_pagina():
        break
```

Apply the predefined date range and begin the scraping process. For each page, the script downloads all available parliamentary PDFs and then moves to the next page. The process continues until no further page is available.

In [ ]:

```
# 16. Closing the browser
driver.quit()
```

Close the automated browser once all parliamentary records within the selected period have been processed.

DIARIES OF THE CZECH PARLIAMENT

In [ ]:

```
# 1. Importing the required libraries
import os
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin
```

The script begins by importing the libraries required to access the parliamentary website, process the HTML page and download the PDF documents. requests is used to retrieve the webpage and PDF files, BeautifulSoup to parse the HTML

structure, urljoin to construct complete URLs from relative links, and os to manage folders and downloaded files.

In [ ]:
```
# 2. Defining the source website and destination folder
URL = "https://www.psp.cz/eknih/2010ps/tesnopis/index.htm"

CARPETA_DESTINO = r"D:\TRABAJO\DIARIOS\RCH\2010-2013"

os.makedirs(CARPETA_DESTINO, exist_ok=True)
```

The script defines the webpage containing the Czech parliamentary records as the main source. It also specifies the local folder where the downloaded PDF documents will be stored. If the destination folder does not already exist, it is automatically created.

In [ ]:
```
# 3. Accessing the parliamentary index
print("Leyendo índice...")

respuesta = requests.get(URL)
respuesta.raise_for_status()

soup = BeautifulSoup(respuesta.text, "html.parser")
```

The script sends an HTTP request to the parliamentary website and retrieves the HTML content of the index page. raise_for_status() checks whether the request was successful, while BeautifulSoup parses the HTML so that the links contained on the page can be identified and processed.

In [ ]:
```
# 4. Identifying the PDF documents
pdfs = []

for enlace in soup.find_all("a", href=True):

    href = enlace["href"]

    if href.lower().endswith(".pdf"):
        pdfs.append(urljoin(URL, href))

print(f"Encontrados {len(pdfs)} PDFs.")
```

The script searches the HTML page for all hyperlinks and examines their href attributes. Links whose addresses end in .pdf are identified as parliamentary documents and added to the pdfs list. urljoin() converts relative links into complete URLs, ensuring that each document can be accessed directly.

In [ ]:
```
# 5. Preparing the PDF downloads
for i, pdf_url in enumerate(pdfs, start=1):

    nombre = os.path.basename(pdf_url)

    ruta = os.path.join(CARPETA_DESTINO, nombre)

    if os.path.exists(ruta):
        print(f"[{i}/{len(pdfs)}] Ya existe: {nombre}")
        continue
```

The script processes the identified PDFs one by one. For each document, it extracts the original filename from its URL and constructs the corresponding local storage path. Before downloading a document, it checks whether the file already exists in the destination folder. This prevents previously downloaded documents from being downloaded again.

In [ ]:
```
# 6. Downloading and saving the PDFs
print(f"[{i}/{len(pdfs)}] Descargando {nombre}")

r = requests.get(pdf_url)
r.raise_for_status()
```

```
with open(ruta, "wb") as f:
    f.write(r.content)
```

Each PDF is retrieved directly from its URL using an HTTP request. After confirming that the request was successful, the binary content of the document is written to the previously defined local folder.

In [ ]:

```
# 7. Completing the download process
print("\n✅ Descarga finalizada.")
```

Once all identified PDF files have been processed, the script indicates that the download process has been completed.

DIARIES OF THE SLOVAKIAN PARLIAMENT

In [ ]:

```
# 1. Importing the required libraries
import os
import time
import requests
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from webdriver_manager.chrome import ChromeDriverManager
```

The script imports the libraries required to access the Slovak parliamentary website, navigate through its pages and download the parliamentary documents. Selenium is used to interact with the website dynamically, while requests is used later to download the documents. os manages local folders and files, time introduces waiting periods during page loading, and webdriver_manager automatically manages the Chrome driver required by Selenium.

In [ ]:

```
# 2. Configuring the browser and download preferences
def setup_browser():

    chrome_options = Options()

    prefs = {
        "download.default_directory": r"D:\TRABAJO\DIARIOS\SLV",
        "download.prompt_for_download": False,
        "download.directory_upgrade": True,
        "plugins.always_open_pdf_externally": True
    }

    chrome_options.add_experimental_option("prefs", prefs)

    driver = webdriver.Chrome(
        service=Service(ChromeDriverManager().install()),
        options=chrome_options
    )

    driver.maximize_window()

    return driver
```

The setup_browser() function configures the Chrome browser used for the scraping process. It establishes the default download directory, disables download confirmation prompts and configures PDF files to be opened externally rather than within the browser. The function then initializes the Chrome driver and maximizes the browser window.

In [ ]:

```
# 3. Starting the browser
driver = setup_browser()
```

The configured browser is launched by calling the setup_browser() function. The resulting Selenium driver is stored in the driver variable and is used throughout the scraping process to navigate and interact with the parliamentary website.

In [ ]:

```
# 4. Selecting the parliamentary legislature
# ESTE ES PARA LA LEGISLATURA 2016-2020
#driver.get("https://www.nrsr.sk/dl/Browser/Default?legId=13&termNr=7")

# ESTE ES PARA LA LEGISLATURA 2020-2025
driver.get("https://www.nrsr.sk/dl/Browser/Default?legId=13&termNr=8")
```

The script specifies the parliamentary legislature from which the records will be collected. The commented URL corresponds to the 2016–2020 legislature, while the active URL accesses the 2020–2025 legislature. In this version of the script, the 2020–2025 legislature is therefore selected for data collection.

In [ ]:

```
# 5. Establishing an explicit wait
wait = WebDriverWait(driver, 20)
```

A maximum waiting time of 20 seconds is established for elements that need to become available on the webpage. This allows the script to wait for dynamically loaded elements before attempting to interact with them.

In [ ]:

```
# 6. Locating the parliamentary session transcript section
boton = wait.until(
    EC.element_to_be_clickable((
        By.XPATH,
        "//a[contains(., 'Prepis zo schôdze')]"
    ))
)
```

The script searches for the link containing the text “Prepis zo schôdze”, which corresponds to the parliamentary session transcript section. Selenium waits until this element is available and clickable before proceeding.

In [ ]:

```
# 7. Accessing the session transcript list
# PRIMERO HACER CLICK
driver.execute_script("arguments[0].click();", boton)

# Esperar a que cargue la lista
time.sleep(3)
```

The script activates the identified link using JavaScript. It then waits three seconds to allow the list of available parliamentary session documents to load.

In [ ]:

```
# 8. Creating a list to store document URLs
urls = []
```

An empty list called urls is created to store the URLs of the parliamentary documents identified during the scraping process.

In [ ]:

```
# 9. Collecting documents from the first page
# ---------- PAGINA 1 ----------

enlaces = driver.find_elements(
    By.XPATH,
    "//a[contains(@href,'documentId=')]"
)

for e in enlaces:

    url = e.get_attribute("href")

    if url not in urls:
        urls.append(url)

print(f"Página 1: {len(urls)} URLs acumuladas")
```

The script searches the first page for links whose URLs contain the parameter documentId=, which identifies the parliamentary documents available for download. Each URL is extracted and added to the urls list, provided that it has not already been collected.

In [ ]:

```
# 10. Initializing the pagination process
# ---------- RESTO DE PAGINAS ----------

pagina = 2
```

After processing the first page, the script initializes the pagination process by setting the current page number to 2.

In [ ]:

```
# 11. Accessing subsequent pages
while True:

    print(f"Página {pagina}")

    try:

        boton = wait.until(
            EC.element_to_be_clickable((
                By.XPATH,
                f"//span[text()='{pagina}']"
            ))
        )

    except:
        print("No hay más páginas.")
        break
```

The script enters a loop to process the remaining pages of parliamentary records. For each page, it searches for the corresponding page-number element and waits until it becomes clickable. If the requested page number cannot be found, the script assumes that there are no additional pages and terminates the loop.

In [ ]:

```
# 12. Navigating to the next page
driver.execute_script("arguments[0].click();", boton)

time.sleep(3)
```

Once the page-number element has been identified, the script clicks it using JavaScript and waits three seconds for the new page of records to load.

In [ ]:

```
# 13. Extracting document URLs from each subsequent page
enlaces = driver.find_elements(
    By.XPATH,
    "//a[contains(@href,'documentId=')]"
)

for e in enlaces:

    url = e.get_attribute("href")

    if url not in urls:
        urls.append(url)

print(f"URLs acumuladas: {len(urls)}")
```

The script repeats the same URL extraction procedure for each subsequent page. It identifies all links containing documentId=, extracts their URLs and adds new URLs to the cumulative list. This process ensures that documents from all available pages are collected without duplicating URLs.

In [ ]:

```
# 14. Moving to the next page
pagina += 1
```

After processing the current page, the page counter is increased by one so that the next iteration of the loop attempts to access the following page. The loop continues until no additional page can be located.

In [ ]:

```
# 15. Reporting the total number of documents identified
print(f"\nTOTAL URLS: {len(urls)}")
```

Once all available pages have been processed, the script reports the total number of unique parliamentary document URLs collected.

In [ ]:

```
# 16. Creating a requests session
session = requests.Session()
```

A persistent requests session is created to manage the subsequent document downloads. Using a session allows the script to maintain shared connection and authentication information across multiple requests.

In [ ]:

```
# 17. Transferring the browser cookies to the download session
# Copiar las cookies del navegador
for cookie in driver.get_cookies():
    session.cookies.set(cookie["name"], cookie["value"])
```

The script retrieves the cookies stored in the Selenium browser and transfers them to the requests session. This allows the subsequent download requests to use the same session information established while navigating the parliamentary website. This step is particularly useful when documents require information from the active browser session to be accessed.

In [ ]:

```
# 18. Defining the destination folder
CARPETA = r"D:\TRABAJO\DIARIOS\SLV\2020-2025"
os.makedirs(CARPETA, exist_ok=True)
```

The script defines the local folder where the parliamentary documents corresponding to the 2020–2025 legislature will be stored. If the folder does not already exist, it is automatically created.

In [ ]:

```
# 19. Processing each parliamentary document
for url in urls:

    document_id = url.split("documentId=")[1]

    print("Descargando:", document_id)
```

The script iterates through all previously collected document URLs. For each URL, it extracts the value associated with documentId=, which serves as the document identifier, and displays it while the file is being downloaded.

In [ ]:

```
# 20. Downloading the document
respuesta = session.get(url, timeout=30)

if respuesta.status_code != 200:
    print("Error")
    continue
```

Each parliamentary document is requested through the previously created session. A timeout of 30 seconds is established for each request. If the server does not return a successful status code (200), the script reports an error and moves to the next document.

In [ ]:

```
# 21. Checking the server-provided filename
disposition = respuesta.headers.get("Content-Disposition", "")

nombre = f"{document_id}.docx"
```

```
if "filename=" in disposition:
    nombre = disposition.split("filename=")[1].strip('"')
```

The script examines the Content-Disposition header of the response to determine whether the server provides an original filename for the document. If such a filename is available, it is used. Otherwise, the document ID is used to generate a filename with the .docx extension.

In [ ]:

```
# 22. Constructing the local file path
ruta = os.path.join(CARPETA, nombre)
```

The filename is combined with the previously defined destination folder to create the complete local path where the document will be stored.

In [ ]:

```
# 23. Checking whether the document already exists
if os.path.exists(ruta):
    print("Ya existe")
    continue
```

Before saving the document, the script checks whether a file with the same name already exists in the destination folder. If it does, the download is skipped to avoid creating duplicate files.

In [ ]:

```
# 24. Saving the downloaded document
with open(ruta, "wb") as f:
    f.write(respuesta.content)
```

The content returned by the server is written to the local file in binary mode. This saves the parliamentary document in the designated folder.

In [ ]:

```
# 25. Closing the browser
driver.quit()
print("\n¡Proceso terminado!")
```

Once all documents have been processed, the Selenium browser is closed and the scraping session is terminated. Also, the script displays a final message indicating that the document collection process has been completed.

DIARIES OF THE FRENCH PARLIAMENT

In [ ]:

```
// 1. Importing libraries and defining the scraper parameters
const fs = require("fs");
const path = require("path");
const https = require("https");
const cheerio = require("cheerio");
const iconv = require("iconv-lite");
const { Document, Packer, Paragraph, TextRun, AlignmentType } = require("docx");
```

Import the libraries required to retrieve, parse and store the French parliamentary records. The https module is used to download the webpages, cheerio to parse their HTML structure, and iconv-lite to correctly decode webpages using different character encodings. The fs and path modules manage the output files and folders, while the docx package is used to generate Word documents.

In [ ]:

```
// 2. Defining the source and output parameters
const URL_INDEX = "https://PONER/AQUI/LA/URL/DEL/INDICE/DE/LA/ASSEMBLEE";
const DIR_OUT = "/AQUI/PONER/LA/RUTA/DE/SALIDA";
const FORCE_OVERWRITE = false;
const DELAY_MS = 800;
```

Define the main parameters of the scraping process, including the URL of the French National Assembly index page and the directory where the resulting documents will be stored. The scraper is configured not to overwrite documents that have already been downloaded, and an 800-millisecond delay is introduced between requests.

In [ ]:

```
# 3. Defining the document formatting parameters
const FONT_SIZES = {
  header: 16,
  date: 18,
  h1: 16,
  h2: 16,
  h3: 14,
  h4: 14,
  h5: 12,
  h6: 12,
  body: 11
};
```

Define the font sizes used when reconstructing the parliamentary records in Word format. Different sizes are assigned to dates, headings and body text in order to preserve the hierarchical structure of the original parliamentary documents.

In [ ]:

```
# 4. Defining retry parameters
const MAX_RETRIES = 4;
const RETRY_DELAY_MS = 3000;
```

Set the maximum number of retries and the waiting time between attempts when a webpage cannot be retrieved because of temporary HTTP or connection errors.

In [ ]:

```
# 5. Detecting the webpage character encoding
function decode_buffer(buf, c_type) {
  let enc = null;
  const m_head = /charset=([\w-]+)/i.exec(c_type || "");
  if (m_head) enc = m_head[1];

  if (!enc) {
    const head_str = buf.toString("latin1").slice(0, 4000);
    const m_meta = /<meta[^>]+charset=["']?\s*([\w-]+)/i.exec(head_str) ||
                   /charset=([\w-]+)/i.exec(head_str);
    if (m_meta) enc = m_meta[1];
  }

  enc = (enc || "windows-1252").toLowerCase();
  if (enc === "iso-8859-1" || enc === "latin1") enc = "windows-1252";
  if (!iconv.encodingExists(enc)) enc = "windows-1252";

  return iconv.decode(buf, enc);
}
```

Detect the character encoding used by the downloaded French parliamentary webpage. The script first checks the HTTP headers and, if the encoding is not specified there, examines the HTML metadata. When no valid encoding is identified, windows-1252 is used as the default. This step ensures that French characters and accents are correctly preserved during text extraction.

In [ ]:

```
# 6. Downloading an individual webpage
function fetch_single(url) {
  return new Promise((resolve, reject) => {
    const req = https.get(
      url,
      { headers: { "User-Agent": "Mozilla/5.0" } },
      (res) => {
        if (res.statusCode !== 200) {
          reject(new Error(`HTTP ${res.statusCode} en ${url}`));
          res.resume();
          return;
        }
```

```
        const chunks = [];
        res.on("data", (c) => chunks.push(c));
        res.on("end", () =>
          resolve(
            decode_buffer(
              Buffer.concat(chunks),
              res.headers["content-type"]
            )
          )
        );
      }
    );

    req.on("error", reject);
  });
}
```

Download a parliamentary webpage using an HTTPS request. The response is collected in chunks and combined into a single buffer before being decoded using the previously defined character-encoding procedure. Requests are identified as coming from a standard web browser through the User-Agent header.

In [ ]:

```
# 7. Introducing delays between requests
const sleep = (ms) => new Promise((r) => setTimeout(r, ms));
```

Define a utility function that pauses the execution of the scraper for a specified number of milliseconds. This function is later used to control the frequency of requests to the parliamentary website.

In [ ]:

```
# 8. Retrying failed requests
async function fetch_html(url) {
  let last_err;

  for (let attempt = 0; attempt <= MAX_RETRIES; attempt++) {
    try {
      return await fetch_single(url);
    } catch (err) {
      last_err = err;

      const is_temp =
        /HTTP (5\d\d|429)/.test(err.message) ||
        Boolean(err.code);

      if (!is_temp || attempt === MAX_RETRIES) break;

      await sleep(RETRY_DELAY_MS * (attempt + 1));
    }
  }

  throw last_err;
}
```

Retrieve a webpage with an automatic retry mechanism. Temporary server errors, such as HTTP 5xx responses or rate-limit responses (HTTP 429), as well as connection errors, trigger additional attempts. The waiting time increases after each failed attempt, while non-temporary errors are not repeatedly requested.

In [ ]:

```
# 9. Mapping French session names to numerical identifiers
const MAP_ORD = {
  premiere: 1,
  deuxieme: 2,
  seconde: 2,
  troisieme: 3,
  quatrieme: 4,
  cinquieme: 5,
  sixieme: 6,
  septieme: 7,
  huitieme: 8,
  neuvieme: 9,
  dixieme: 10,
  onzieme: 11,
```

```
  douzieme: 12
};
```

Define a mapping between French ordinal terms used in parliamentary URLs and numerical session identifiers. This allows session numbers to be standardised when constructing the output filenames.

In [ ]:

```
# 10. Extracting the session number
function extract_session_num(p_name) {
  const pts = p_name.replace(/\/+$/, "").split("/").filter(Boolean);
  const last_p = pts[pts.length - 1] || "";

  if (/^\d+$/.test(last_p))
    return parseInt(last_p, 10);

  return MAP_ORD[last_p.split("-")[0]] || 1;
}
```

Extract the parliamentary session number from the URL path. If the final component of the URL is numeric, it is directly converted into a number. Otherwise, the French ordinal term is converted using the predefined mapping.

In [ ]:

```
# 11. Mapping French month names
const MAP_MONTHS = {
  janvier: "01",
  "février": "02",
  fevrier: "02",
  mars: "03",
  avril: "04",
  mai: "05",
  juin: "06",
  juillet: "07",
  "août": "08",
  aout: "08",
  septembre: "09",
  octobre: "10",
  novembre: "11",
  "décembre": "12",
  decembre: "12"
};
```

Define a correspondence between French month names and their numerical representation. Both accented and non-accented forms are included to account for possible variations in the source webpages.

In [ ]:

```
# 12. Parsing French dates
function parse_fr_date(t) {
  t = t
    .replace(/\u00A0/g, " ")
    .replace(/\s+/g, " ")
    .trim()
    .toLowerCase();

  const rx = new RegExp(
    "(\\d{1,2})(?:er)?\\s+(" +
    Object.keys(MAP_MONTHS).join("|") +
    ")\\s+(\\d{4})"
  );

  const m = rx.exec(t);

  if (!m) return null;

  return {
    y: m[3],
    m: MAP_MONTHS[m[2]],
    d: m[1].padStart(2, "0")
  };
}
```

Convert dates written in French into a standard numerical format. The function accounts for the French notation used for the first day of the month (1er) and returns the year, month and day in a structured format.

In [ ]:

```
# 13. Extracting parliamentary document links from the index
function get_links(html, base) {
  const $ = cheerio.load(html);
  const seen = new Set();
  const links = [];

  $("ul.crs-index-days").first().children("li").each((_, li) => {
    const dt = parse_fr_date($(li).find("h2").first().text());

    $(li).find("a.link.h5").each((__, a) => {
      const h = $(a).attr("href");
      if (!h) return;

      const full = new URL(h, base).href;

      if (seen.has(full)) return;
      seen.add(full);
```

Parse the French National Assembly index page and identify the links corresponding to individual parliamentary sessions. The date associated with each session is extracted from the index, while duplicate URLs are removed to prevent the same document from being downloaded more than once.

In [ ]:

```
# 14. Creating standardised output filenames
const p_name = new URL(full).pathname;
const s_num = extract_session_num(p_name);

const fallback =
  (p_name.replace(/\/+$/, "").split("/").pop() || "doc")
  .replace(/[^\w.-]+/g, "_");

const f_name = dt
  ? path.join(
      dt.y,
      dt.m,
      `${dt.y}_${dt.m}_${dt.d}-Sesion${s_num}`
    )
  : fallback;

links.push({ url: full, fname: f_name });
```

Construct a standardised filename for each parliamentary session using its date and session number. Documents are organised into year and month folders, creating a chronological structure for the resulting corpus. If a date cannot be extracted, a cleaned version of the URL is used as a fallback filename.

In [ ]:

```
# 15. Cleaning extracted text runs
function clean_runs(rs) {
  const merged = [];

  for (const r of rs) {
    const last = merged[merged.length - 1];

    if (last && last.b === r.b && last.i === r.i)
      last.t += r.t;
    else
      merged.push({ ...r });
  }

  for (const r of merged)
    r.t = r.t.replace(/\s+/g, " ");

  if (merged.length) {
    merged[0].t = merged[0].t.replace(/^\s+/, "");
    merged[merged.length - 1].t =
      merged[merged.length - 1].t.replace(/\s+$/, "");
  }
```

```
  return merged.filter((r) => r.t.length);
}
```

Clean and consolidate consecutive text segments extracted from the HTML. Segments with the same bold and italic formatting are merged, redundant whitespace is removed and empty segments are discarded. This produces cleaner text while preserving relevant formatting information.

In [ ]:

```
# 16. Extracting text while preserving formatting
function extract_r($, el, opts = {}) {
```

Recursively extract the textual content of an HTML element while preserving formatting information such as bold and italics. The function also recognises line breaks and can either preserve them or convert them into spaces depending on the extraction options.

In [ ]:

```
# 17. Identifying parliamentary speakers
function extract_speaker($, p_node) {
  const rs = [];

  $(p_node).contents().each((_, ch) => {
    if (ch.type === "text") {
      const txt = ch.data.replace(/\u00A0/g, " ");

      if (txt.trim())
        rs.push({ t: txt, b: true, i: false });

    } else if (ch.type === "tag") {
      const $ch = $(ch);
      const tg = ch.tagName.toLowerCase();

      const is_i =
        $ch.hasClass("italique") ||
        tg === "i" ||
        tg === "em";

      const txt = $ch.text().replace(/\u00A0/g, " ");

      if (txt.trim())
        rs.push({
          t: txt,
          b: !is_i,
          i: is_i
        });
    }
  });

  return rs;
}
```

Extract the names and labels identifying parliamentary speakers from the dedicated speaker paragraphs in the French parliamentary HTML structure. Speaker information is preserved as bold text, while italic formatting is retained when present.

In [ ]:

```
# 18. Parsing the complete parliamentary record
function parse_doc(html) {
  const $ = cheerio.load(html);
  const ps = [];

  const head = $(".crs-cr-head").first();
```

Parse the HTML structure of an individual French parliamentary record and reconstruct its relevant textual components. The parser distinguishes between the document header, section headings, speaker identification and the substantive text of parliamentary interventions.

In [ ]:

```
# 19. Extracting the document header and date
```

```
head.find(".crs-h-huge-title, h1.crs-h-compte-rendu").each((_, el) => {
  const txt = $(el)
    .text()
    .replace(/\u00A0/g, " ")
    .replace(/\s+/g, " ")
    .trim();

  if (!txt) return;

  ps.push({
    rs: [{ t: txt, b: true, i: false }],
    center: true,
    size: $(el).is("h1")
      ? FONT_SIZES.date
      : FONT_SIZES.header
  });
});
```

Extract the title and date information from the parliamentary record header. These elements are centred and formatted with larger font sizes in the reconstructed Word document.

In [ ]:

```
# 20. Extracting parliamentary sections and headings
const body = $(".crs-cr-content").first();

if (!body.length && !head.length)
  throw new Error("Bad structure");

body.find(".crs-inter").each((_, el) => {
  const $el = $(el);

  if ($el.parents(".crs-inter").length) return;

  $el.children("h1, h2, h3, h4, h5, h6").each((__, t_node) => {
    const rs = extract_r($, t_node, { force_b: true });

    if (rs.length)
      ps.push({
        rs,
        center: false,
        size: FONT_SIZES[t_node.tagName.toLowerCase()]
      });
  });
```

Identify the main parliamentary sections within the document and extract their hierarchical headings. Nested sections are excluded from being processed twice, while the original heading level is retained through the corresponding font size.

In [ ]:

```
# 21. Extracting speaker names and interventions
const spk_p = $el.find("p.orateur").first();

const txt_ps = $el.children("p")
  .toArray()
  .filter((p) => !$(p).hasClass("orateur"));

const spk_rs = spk_p.length
  ? extract_speaker($, spk_p)
  : [];
```

Separate speaker identification from the substantive text of the parliamentary intervention. Speaker paragraphs are identified through the orateur class, while the remaining paragraphs are treated as the content of the intervention.

In [ ]:

```
# 22. Preserving paragraph and line structure
for (const tp of txt_ps) {
  for (const line of extract_r($, tp, { breaks: true })) {
    if (line.length)
      ps.push({
        rs: line,
        center: false
      });
```

```
  }
}
```

Extract the text of each parliamentary intervention while preserving line breaks. Each resulting line is stored as a separate text element, allowing the original structure of the parliamentary record to be approximately reconstructed in the output document.

In [ ]:

```
# 23. Converting extracted elements into Word paragraphs
function build_docx(ps) {
  const chs = ps.map((p) => {
    const sz = p.size || FONT_SIZES.body;

    const is_head =
      sz > FONT_SIZES.body && !p.center;

    return new Paragraph({
      alignment: p.center
        ? AlignmentType.CENTER
        : AlignmentType.JUSTIFIED,

      spacing: {
        after: 120,
        before: is_head ? 240 : 0
      },

      children: p.rs.map((r) =>
        new TextRun({
          text: r.t,
          bold: r.b || undefined,
          italics: r.i || undefined,
          size: sz * 2
        })
      )
    });
  });

  return new Document({
    sections: [{ children: chs }]
  });
}
```

Convert the extracted parliamentary elements into a Word document. The reconstruction preserves the main formatting characteristics of the source, including headings, font sizes, bold and italic text, paragraph alignment and spacing.

In [ ]:

```
# 24. Generating and saving each parliamentary document
async function generate_file(html, out_path) {
  const ps = parse_doc(html);
  const doc = build_docx(ps);
  const buf = await Packer.toBuffer(doc);

  fs.writeFileSync(out_path, buf);

  return ps.length;
}
```

Parse the downloaded HTML, reconstruct its content as a Word document and save the resulting file to the specified output path. The number of extracted paragraphs is returned as a basic measure of the amount of content processed.

In [ ]:

```
# 25. Running the scraper
async function run_scraper(fetch_fn = fetch_html) {
  const idx_html = await fetch_fn(URL_INDEX);
  const links = get_links(idx_html, URL_INDEX);

  fs.mkdirSync(DIR_OUT, { recursive: true });

  let c_ok = 0,
      c_skip = 0,
      c_err = 0;
```

Initiate the complete scraping process by downloading the parliamentary index and extracting the links to individual parliamentary sessions. The output directory is created if necessary, and counters are initialised to track successfully processed, skipped and failed documents.

In [ ]:

```
# 26. Downloading and processing each session
for (let i = 0; i < links.length; i++) {
  const { url, fname } = links[i];

  const p_out =
    path.join(DIR_OUT, `${fname}.docx`);

  if (!FORCE_OVERWRITE &&
      fs.existsSync(p_out)) {
    c_skip++;
    continue;
  }

  try {
    const h = await fetch_fn(url);

    fs.mkdirSync(
      path.dirname(p_out),
      { recursive: true }
    );

    await generate_file(h, p_out);
    c_ok++;

  } catch (e) {
    c_err++;
  }

  if (i < links.length - 1 &&
      DELAY_MS > 0)
    await sleep(DELAY_MS);
}
```

Iterate through all identified parliamentary sessions. Existing documents are skipped unless overwriting is explicitly enabled. Each new session is downloaded, parsed and converted into a Word document, while errors are counted without interrupting the processing of the remaining sessions. A delay is introduced between requests.

In [ ]:

```
# 27. Executing the scraper and exporting functions
if (require.main === module) {
  run_scraper().catch(() => process.exit(1));
}

module.exports = {
  decode_buffer,
  get_links,
  parse_doc,
  build_docx,
  generate_file,
  run_scraper
};
```

Execute the scraper when the file is run directly and export the main functions so that individual components can be reused or tested from other JavaScript files.

DIARIES OF THE BRITISH PARLIAMENT

In [ ]:

```
# 1. Importing libraries and modules
'use strict';

const path = require('path');
const fs = require('fs');
const { Packer } = require('docx');
```

Import the main Node.js libraries required to manage file paths and files, together with the docx package used to generate Word documents. The path module is used to construct output folders and file paths, while fs is used to create directories and save the generated documents.

In [ ]:

```
# 2. Importing the Hansard API and processing functions
const {
  SECTIONS_OF_INTEREST,
  getSectionsForDay,
  getSectionExtId,
  fetchDebateByExtId,
} = require('./src/hansardApi');

const { parseDebateFromApi } = require('./src/parseDebateFromApi');
const { buildCombinedDocx } = require('./src/buildDocx');
const { isDateString, dateRange, sleep } = require('./src/dateUtils');
```

Import the functions required to retrieve and process parliamentary debates from the Hansard API. These functions identify the parliamentary sections available for a given day, retrieve the corresponding debate identifiers, download the debate content, parse the API response and convert the extracted sections into Word documents. Additional utility functions are used to validate dates, generate date ranges and introduce delays between requests.

In [ ]:

```
# 3. Setting the request delay
const DELAY_MS = Number(process.env.HANSARD_DELAY_MS ?? 300);
```

Define the delay between consecutive requests to the Hansard API. The default value is 300 milliseconds, although it can be modified through an environment variable. This prevents the script from sending requests continuously and allows the scraping process to run at a controlled rate.

In [ ]:

```
# 4. Retrieving the content of a parliamentary section
async function get_sec_content(extId) {
  const data = await fetchDebateByExtId(extId);
  const blocks = parseDebateFromApi(data);
  const title = (data.Overview && data.Overview.Title) || 'Sección';
  return { title, blocks };
}
```

Retrieve a specific parliamentary debate using its external identifier. The API response is then parsed into structured text blocks, while the title of the parliamentary section is extracted from the response. If no title is available, a default label is assigned.

In [ ]:

```
# 5. Processing a single parliamentary day
async function proc_date(date, out_dir, house = 'Commons') {
  const secs = await getSectionsForDay(date, house);

  if (secs.length === 0) {
    return { gen: [], errs: [] };
  }
```

Process the parliamentary record for a specific date. First, the script queries the Hansard API to identify the parliamentary sections available for that day in the House of Commons. If no sections are available, the date is skipped without generating an output document.

In [ ]:

```
# 6. Selecting the sections of interest
const data_arr = [];
const errs = [];

for (const [api_n, url_n] of Object.entries(SECTIONS_OF_INTEREST)) {
  if (!secs.includes(api_n)) continue;
```

Compare the sections available on the selected date with a predefined list of sections of interest. Only sections included in this predefined list are processed. This allows the database to focus on the relevant parliamentary material rather than retrieving

every section available through the API.

In [ ]:

```
# 7. Retrieving and parsing each selected section
try {
  const extId = await getSectionExtId(api_n, date, house);

  if (!extId) {
    errs.push({ sec: url_n, msg: 'No GUID' });
    continue;
  }

  const content = await get_sec_content(extId);
  data_arr.push(content);
} catch (e) {
  errs.push({ sec: url_n, msg: e.message });
}
```

For each selected parliamentary section, the script retrieves its external identifier (GUID) and uses it to obtain the full debate content. The extracted section is added to the dataset for the corresponding day. If an identifier cannot be found or an error occurs during retrieval, the problem is recorded and the script continues processing the remaining sections.

In [ ]:

```
# 8. Combining the sections into a single document
if (data_arr.length === 0) {
  return { gen: [], errs };
}

const doc = buildCombinedDocx(data_arr, {
  title: `Hansard - ${date}`,
  description: `[https://hansard.parliament.uk/commons/${date}`,
});
```

Once all relevant sections for a given day have been retrieved, they are combined into a single Word document. The document is labelled with the corresponding date and includes the Hansard parliamentary record as its source.

This ensures that the parliamentary material corresponding to each date is stored as a single document while preserving the different sections retrieved from Hansard.

In [ ]:

```
# 9. Saving the parliamentary record
const fname = `parlamento_ingles_${date}.docx`;
const out_path = path.join(out_dir, fname);

const buf = await Packer.toBuffer(doc);
fs.mkdirSync(out_dir, { recursive: true });
fs.writeFileSync(out_path, buf);

return { gen: [out_path], errs };
```

Convert the generated document into a Word file and save it to the corresponding output directory. Directories are automatically created when necessary. Each document is named using the date of the parliamentary record, allowing the resulting corpus to be organised chronologically.

In [ ]:

```
# 10. Organising documents by year and month
function get_out_dir(base_dir, date) {
  const [y, m] = date.split('-');
  return path.join(base_dir, y, m);
}
```

Organise the output documents into folders according to year and month. This creates a hierarchical structure that facilitates the management and later processing of the parliamentary corpus.

In [ ]:

```
# 11. Processing a range of dates
async function proc_date_range(s_date, e_date, base_dir) {
```

```
  const dates = [...dateRange(s_date, e_date)];
  const sum = { tot: dates.length, w_sess: 0, w_o_sess: 0, docs: 0, errs: [] };
```

Generate all dates between the specified starting and ending dates and process them sequentially. The script also creates a summary structure to keep track of the number of dates processed, dates with parliamentary sessions, generated documents and errors.

In [ ]:

```
# 12. Processing each date sequentially
for (let i = 0; i < dates.length; i++) {
  const d = dates[i];

  try {
    const { gen, errs } = await proc_date(d, get_out_dir(base_dir, d));

    if (gen.length > 0) {
      sum.w_sess++;
      sum.docs += gen.length;
    } else if (errs.length === 0) {
      sum.w_o_sess++;
    }
```

Iterate through the selected date range and process each date individually. The script records whether a parliamentary document was successfully generated or whether the date contained no relevant parliamentary session.

In [ ]:

```
# 13. Recording errors and controlling the request rate
for (const { sec, msg } of errs) {
  sum.errs.push({ date: d, msg: `${sec}: ${msg}` });
}

if (DELAY_MS > 0 && i < dates.length - 1) {
  await sleep(DELAY_MS);
}
```

Record any errors associated with individual parliamentary sections while allowing the process to continue with subsequent dates. A delay is introduced between requests to control the frequency of API calls.

In [ ]:

```
# 14. Defining the input dates and output directory
function parse_args(argv) {
  const def_dir = "/AQUI/PONER/LA/RUTA/DE/SALIDA";
  const [a, b, c] = argv;
```

Define the command-line arguments used to specify the period to be downloaded and the output directory. The script can operate either on a single date or on a range between two dates.

In [ ]:

```
# 15. Date validation and single-date or date-range processing
if (!a) {
  return {
    s_date: '2010-01-20',
    e_date: '2010-01-20',
    out_dir: def_dir
  };
}

if (!isDateString(a)) {
  throw new Error('Bad date');
}

if (b && isDateString(b)) {
  return {
    s_date: a,
    e_date: b,
    out_dir: c || def_dir
  };
}
```

```
return {
  s_date: a,
  e_date: a,
  out_dir: b || def_dir
};
```

Validate the date provided by the user and determine whether the script should process a single date or a date range. If two valid dates are provided, all dates between them are processed; otherwise, only the specified date is used. A default output directory is provided when no alternative path is specified.

In [ ]:

```
# 16. Main execution function
async function main() {
  let args;

  try {
    args = parse_args(process.argv.slice(2));
  } catch (e) {
    process.exitCode = 1;
    return;
  }

  const { s_date, e_date, out_dir } = args;

  try {
    const sum = await proc_date_range(s_date, e_date, out_dir);
    if (sum.errs.length > 0) process.exitCode = 1;
  } catch (e) {
    process.exitCode = 1;
  }
}
```

Execute the complete scraping process using the dates and output directory defined by the command-line arguments. Errors occurring during argument validation or document processing are handled without stopping the script unexpectedly.

In [ ]:

```
# 17. Starting the script and exporting functions
if (require.main === module) {
  main();
}

module.exports = {
  proc_date,
  proc_date_range,
  get_out_dir,
  parse_args,
};
```

Run the main() function when the file is executed directly. The main processing functions are also exported so that they can be reused or tested from other JavaScript files.

EXTRACTION OF JSON FILES

In [ ]:

```
# 1. Define the metadata for the dataset
import json

metadata_json = '''
{
 "@context": "https://schema.org/",
 "@type": "Dataset",
 "author": [
   {
     "affiliation": {
       "name": "Research Institute for Artificial Intelligence"
     },
     "identifier": "0000-0002-3752-644X",
     "name": "Cerquides Bueno, Jesus"
   },
```

```
    {
      "affiliation": {
        "name": "Research Institute for Artificial Intelligence"
      },
      "identifier": "0009-0004-6195-1106",
      "name": "Mateos Marín, Paula"
    }
  ],
  "name": "Spanish Congress parliamentary records (1977-2024)",
  "datePublished": "2024-05-17",
  "url": "https://zenodo.org/record/11172651"
}
'''
```

We declare a multi-line string with JSON-LD format and describe a dataset following the schema.org standard, so that it is understandable by people and machines (such as Google Scholar, Zenodo, etc.).

It includes: Resource type (@type), authors with their ORCIDs and affiliations, dataset name, publication date, and access URL.

In [ ]:

```
# 2. Converting JSON to a Python dictionary
metadata = json.loads(metadata_json)
```

The JSON string is converted to a Python dictionary. With this, we can now access its values with metadata[‘name’], metadata[‘author’], etc.

In [ ]:

```
# 3. Show the main JSON data on screen
print("Nombre del dataset:", metadata["name"])
print("Fecha de publicación:", metadata["datePublished"])
print("Autores:")
for author in metadata["author"]:
    print(f" - {author['name']} ({author['identifier']})")

print("URL del dataset:", metadata["url"])
```

This way, we print the name of the dataset, the date, the list of authors with their ORCID identifier, and the link to access the dataset in the output.

In [ ]:

```
# 4. Download ZIP files from Zenodo
import requests

urls = [
    "https://zenodo.org/api/records/11212304/files-archive",
    "https://zenodo.org/api/records/11195944/files-archive",
    "https://zenodo.org/api/records/11172652/files-archive"
]
```

We import *requests* to perform HTTP downloads. We then define the three Zenodo URLs (API) that return ZIP files of the datasets.

In [ ]:

```
# 5. Local download of files
for idx, url in enumerate(urls, start=1):
    print(f"Descargando archivo {idx} desde {url}...")
    response = requests.get(url, stream=True)

    if response.status_code == 200:
        filename = f"archivo_{idx}.zip"
        with open(filename, "wb") as file:
            for chunk in response.iter_content(chunk_size=8192):
                file.write(chunk)
        print(f" Archivo guardado como {filename}")
    else:
        print(f" Error al descargar {url}, código de estado: {response.status_code}")
```

With these lines, we go through each URL, attempt to download the file using requests.get(url, stream=True), and with that stream=True condition, we are able to download it in parts, which is ideal for large files. If the response is correct (status_code == 200), it saves the file to disk as file_1.zip, file_2.zip, etc. Conversely, if there is an error, it displays it in the console.

In [ ]:
```
# 6. Preparation for ZIP extraction
import zipfile
import os

zip_files = ["archivo_1.zip", "archivo_2.zip", "archivo_3.zip"]
extract_folder = "D:\\DIARIOS\\archivos_extraidos"
os.makedirs(extract_folder, exist_ok=True)
```

First, we import *zipfile* to handle compressed files; we define the names of the downloaded files; we prepare a folder where they will be extracted and, finally, with *os.makedirs(extract_folder, exist_ok=True)*, the folder is created if it did not previously exist.

In [ ]:
```
# 7. Extracting ZIP files and reading .txt files
for zip_file in zip_files:
   print(f" Extrayendo {zip_file} en {extract_folder}...")

   with zipfile.ZipFile(zip_file, 'r') as zip_ref:
       zip_ref.extractall(extract_folder)

       for file_name in zip_ref.namelist():
           file_path = os.path.join(extract_folder, file_name)

           if file_name.endswith(".txt"):
               with open(file_path, "r", encoding="utf-8") as f:
                   contenido = f.read()
                   print(f" Contenido de {file_name}:\n")
                   print(contenido[:500])
                   print("\n" + "-"*80 + "\n")
```

This will go through the three downloaded ZIP files and extract each one completely into the specified folder. It will then go through the contents of the ZIP file and, if a file ends in .txt, open it with UTF-8 encoding. Afterwards, only the first 500 characters and a separator line will be printed on the screen.

READING JSON FILES

In [ ]:
```
# 1. Importing libraries
import zipfile
import os
import json
```

With these imports, we can work with .zip files (thanks to *zipfile*); manipulate paths and files with *os*; and read and work with .json files with *json*.

In [ ]:
```
# 2. Define work routes
zip_folder = r"D:\DIARIOS\ZIPS+77"
extract_folder = r"D:\DIARIOS\archivos_extraidos"
os.makedirs(extract_folder, exist_ok=True)
```

Define the path where all the downloaded .zip files are located and with *extract_folder* specify the path where the files inside those ZIPs will be extracted. If the folder does not exist, create it with *os.makedirs*.

In [ ]:
```
# 3. Search for all ZIP files in the input folder
zip_files = [f for f in os.listdir(zip_folder) if f.endswith(".zip")]
```

Go through all the files in the zip_folder, leaving only those ending in .zip. The result is a list such as: ['77-01.zip', '78-03.zip',

…]

In [ ]:

```
# 4. Verify that there are ZIP files, extract and read the JSON files they contain.
if not zip_files:
   print(" No se encontraron archivos ZIP en", zip_folder)
else:
   for zip_file in zip_files:
       zip_path = os.path.join(zip_folder, zip_file)
       print(f" Extrayendo {zip_file} en {extract_folder}...")

       try:
           with zipfile.ZipFile(zip_path, 'r') as zip_ref:
               zip_ref.extractall(extract_folder)

               for file_name in zip_ref.namelist():
                   file_path = os.path.join(extract_folder, file_name)

                   if file_name.endswith(".json"):
                       with open(file_path, "r", encoding="utf-8") as f:
                           try:
                               data = json.load(f)
                               print(f" Contenido de {file_name}:\n")
                               print(json.dumps(data, indent=4, ensure_ascii=False)[:500])
                               print("\n" + "-"*80 + "\n")
                           except json.JSONDecodeError:
                               print(f" Error al leer {file_name}: No es un JSON válido.")
        except zipfile.BadZipFile:
           print(f" {zip_file} no es un archivo ZIP válido o está corrupto.")
```

Now, we go through each .zip file found, build its full path with os.path.join(…) and print a message indicating which file is being extracted to the *extract_folder*.

With *zip_ref.namelist()* you get a list of all the files inside the ZIP and, for each one, the full path *file_path* is constructed. If it is .json, it is opened and an attempt is made to load it with *json.load()* . It is displayed in the console formatted and indented, but only the first 500 characters, so as not to saturate the output. If the file does not have a valid JSON format, a warning is thrown.

CONSTRUCTION OF THE DATABASE

DATABASE WITH JSON FILES

This code aims to process speeches from Spanish Parliament sessions (in JSON format) to detect conspiracy theories using regular expressions, extract metadata, and save everything in a .csv file.

In [ ]:

```
# 1. Importing libraries
import json
import os
import re
import csv
import pandas as pd
from datetime import datetime
```

With *json* and *os* we can load and manage files; with *re* we can work with regular expressions; to save results we use *csv*; thanks to *pandas* we can display the results as a table and, finally, with *datetime* we can convert dates to a readable format.

In [ ]:

```
# 2. Define folders and files to be processed
folder_path = r"D:\DIARIOS\archivos_extraidos"
files_to_process = ["XIV.json", "XV.json"]
```

We indicate the path to the .json files and specify which documents we want to analyse. In our case, the files ‘XIV.json’ and ‘XV.json’ will be processed, corresponding to the fourteenth and fifteenth legislatures respectively, as we want to detect mentions of conspiracy theories related to the COVID-19 pandemic.

In [ ]:

```
# 3. Function to read JSON files
def read_json_files(folder_path, files):
    all_data = []
    print(f"\U0001F465 Inicio de la lectura de archivos JSON...")
    for file in files:
        print(f"\U0001F4C2 Procesando archivo: {file}")
        file_path = os.path.join(folder_path, file)
        if os.path.exists(file_path):
            with open(file_path, 'r', encoding='utf-8') as f:
                data = json.load(f)
                if isinstance(data, list):
                    all_data.extend(data)
                else:
                    print(f" Advertencia: {file} no es una lista.")
        else:
            print(f" El archivo {file} no existe.")
    print(f" Lectura de archivos completada. Total de documentos: {len(all_data)}")
    return all_data

data = read_json_files(folder_path, files_to_process)
```

It goes through the specified files, loads them, and combines them into a single list. It also checks whether the file exists and whether it is a list of dictionaries. With *all_data*, it returns a list of all the speeches found.

Finally, we define *data* to contain all the documents/texts from the 14th and 15th legislatures.

In [ ]:

```
# 4. List of patterns of conspiracy theories related to COVID-19.
conspiracy_patterns = [
    r"vacunas.*(microchip|control|experimento|engaño|grafeno)",
    r"PCR.*(fraude|manipulación|mentira|falsa pandemia)",
    r"5G.*(covid|control|manipulación)",
    r"agenda 2030.*(dominio|población|farsa)",
    r"Nuevo Orden Mundial.*(control|élite|manipulación)",
    r"plandemia|falsa pandemia|dictadura sanitaria",
    r"big pharma.*(mentira|engaño|beneficio económico)",
    r"OMS.*(corrupción|manipulación|engaño)"
]

regex_patterns = [re.compile(pattern, re.IGNORECASE) for pattern in conspiracy_patterns]
```

This list compiles regular expressions designed to search for phrases such as ‘vaccines and microchips’, ‘5G causes COVID’, ‘Agenda 2030 and domination’, etc. Each line searches for matches that suggest a conspiracy theory.

Then, with *regex_patterns*, the patterns are compiled to improve efficiency and make them case-insensitive.

In [ ]:

```
# 5. Auxiliary regular expressions
orador_pattern = re.compile(r"(?:El señor|La señora)\s([A-Za-zÁÉÍÓÚÑáéíóúñ]+(?: [A-Za-zÁÉÍÓÚÑáéíóúñ]+)*)")
fecha_pattern = re.compile(r"[a-zA-Z]+ (\d{1,2}) de ([a-zA-Z]+) de (\d{4})")
legislatura_pattern = re.compile(r"([IVXLCDM]+)\s*LEGI(SATURA)?", re.IGNORECASE)
sesion_pattern = re.compile(r"\\nSesi\\u00f3n n\\u00fam\.\s*(\d+)")
```

These expressions detect information from the context of the speech, such as the speaker (*speaker_pattern*), the date (*date_pattern*), the legislature (*legislature_pattern*) and the session (*session_pattern*).

In [ ]:

```
# 6. Function for formatting the date
def format_fecha(fecha):
    match = fecha_pattern.search(fecha)
    if match:
        dia = match.group(1).zfill(2)
        mes_texto = match.group(2).lower()
        anio = match.group(3)
        meses = {
            "enero": "01", "febrero": "02", "marzo": "03", "abril": "04", "mayo": "05",
            "junio": "06", "julio": "07", "agosto": "08", "septiembre": "09", "octubre": "10",
            "noviembre": "11", "diciembre": "12"
        }
        mes = meses.get(mes_texto, "00")
```

```
        return f"{dia}/{mes}/{anio}"


    try:
        parsed_date = datetime.strptime(fecha, "%Y%m%d")
        return parsed_date.strftime("%d/%m/%Y")
    except ValueError:
        return fecha
```

Convert dates such as “11 March 2024” to the format “11/03/2024”. To do this, use a dictionary of months to convert the month to a number. Also try to convert formats such as “20240411”.

In [ ]:

```
# 7. Auxiliary functions for extracting information
def extract_legislatura(text)
def extract_sesion(text)
def extract_orador(text)
```

These functions use their respective patterns to extract the legislature, session number, and speaker.

In [ ]:

```
# 8. Extract the context of the mention
def extract_context(text, match):
    start = max(match.start() - 50, 0)
    end = min(match.end() + 50, len(text))
    return text[start:end].strip()
```

Extract a window of 50 characters before and after the text that matches the conspiracy theory.

In [ ]:

```
# 9. Classify the mention according to its tone
def classify_mention(text):
    if any(word in text for word in ["control", "engaño", "manipulación"]):
        return "A"  # Explicit
    elif any(word in text for word in ["dicen", "algunos creen", "se habla"]):
        return "B"  # Implicit
    elif any(word in text for word in ["no es cierto", "falso", "desmentido"]):
        return "C"  # Refutation
    return "B"  # By default, implicit
```

Classify the context as:

- A: Explicit (if there are words such as deception, control…)
- B: Implicit (‘they say’, ‘some believe’…)
- C: Refutation (false, not true)

If nothing clear is detected, ‘B’ is assumed by default.

In [ ]:

```
# 10. Process documents and detect mentions
def process_documents(data, regex_patterns):
    results = []
    print(" Procesando documentos para menciones de teorías de conspiración...")
    for idx, item in enumerate(data):
        print(f" Documento {idx + 1}/{len(data)}...")
        if isinstance(item, dict):
            text = item.get("texto", "").lower()
            fecha = item.get("fecha", "Desconocida")
            sesion = extract_sesion(text)  # Usamos la función modificada para extraer el número de sesión
            partido = item.get("partido", "No identificado")
            ideologia = item.get("ideologia", "No especificada")
            fecha_formateada = format_fecha(fecha)
            legislatura = extract_legislatura(text)
            orador = extract_orador(text)

            for pattern in regex_patterns:
                match = pattern.search(text)
                if match:
                    contexto = extract_context(text, match)
```

```
                    clasificacion = classify_mention(contexto)
                    results.append([fecha_formateada, legislatura, sesion, orador, partido, ideologia,
match.group(0), contexto, clasificacion])
                    print(f" Mención detectada: '{match.group(0)}' → {contexto}")
    print(f" Se encontraron {len(results)} menciones de teorías de conspiración.")
    return results
```

It goes through each data document, extracts metadata (date, session, legislature, speaker, political party, etc.), searches for matches with conspiracy patterns, and saves the results as rows (lists).

In [ ]:

```
# 11. Generate CSV
detected_mentions = process_documents(data, regex_patterns)

output_file = "conspiraciones.csv"
with open(output_file, mode="w", newline="", encoding="utf-8") as file:
    writer = csv.writer(file)
    writer.writerow(["Fecha", "Legislatura", "No. de Sesión", "Orador", "Partido", "Ideología", "Cita",
"Contexto", "Clasificación"])
    writer.writerows(detected_mentions)

print(f" Archivo guardado: {output_file}")
```

Create the file conspiracies.csv. Write the header and then each row detected, resulting in a table with 9 columns.

In [ ]:

```
# 12. Show results
df = pd.read_csv(output_file)
print(df.head())
```

Load the .csv with Pandas and display the first results as a table.

DATABASE WITH PDF FILES

This script automates the reading of previously downloaded PDF files of Congressional journals using web scraping, identifies mentions of conspiracy theories, and visualises the results with graphs and a word cloud.

In [ ]:

```
# 1. Importing libraries
import os
import re
import csv
import fitz  # PyMuPDF
import pandas as pd
from datetime import datetime
import seaborn as sns
import matplotlib.pyplot as plt
from wordcloud import WordCloud
```

The necessary imported libraries are: *os* to interact with the file system (read folders, join paths); *re* for working with regular expressions; *csv* for saving data in .csv format; *fitz* from the *PyMuPDF* module, used to read and extract text from PDF files; *pandas as pd* for manipulating data in table-like structures (DataFrame); *datetime* for working with dates and times (although not used directly in this code); *seaborn* for generating attractive graphs (uses matplotlib underneath); *matplotlib.pyplot* for creating custom graphs; *WordCloud* for creating word clouds.

In [ ]:

```
# 2. Path to PDFs
base_folder_path = r"D:\DIARIOS\77-24"
subfolders = ["XIV", "XV"]
```

Indicamos la ruta principal donde están los PDFs y con *subfolders* especificamos dentro de esa ruta qué documentoscorrespondientes a las legislaturas queremos analizar.

In [ ]:

```
# 3. Patterns of conspiracy theories
conspiracy_patterns = [
    r"vacunas.*(microchip|control|experimento|engaño|grafeno)",
    r"PCR.*(fraude|manipulación|mentira|falsa pandemia)",
    r"5G.*(covid|control|manipulación)",
    r"agenda 2030.*(dominio|población|farsa)",
    r"Nuevo Orden Mundial.*(control|élite|manipulación)",
    r"plandemia|falsa pandemia|dictadura sanitaria",
    r"big pharma.*(mentira|engaño|beneficio económico)",
    r"OMS.*(corrupción|manipulación|engaño)"
]
regex_patterns = [re.compile(p, re.IGNORECASE) for p in conspiracy_patterns]
```

This list groups together regular expressions that detect phrases related to conspiracy theories. They are compiled with re.IGNORECASE so that it does not matter whether they are in upper or lower case.

In [ ]:

```
# 4. Other patterns for extracting metadata
orador_pattern = re.compile(r"(?:El señor|La señora)\s+([A-ZÁÉÍÓÚÑ]+(?: [A-ZÁÉÍÓÚÑ]+)*)", re.IGNORECASE)
legislatura_pattern = re.compile(r"([IVXLCDM]+)\s*LEGI(SLATURA)?", re.IGNORECASE)
sesion_pattern = re.compile(r"Sesi[oó]n n[uú]m\.?\s*(\d+)", re.IGNORECASE)
fecha_pattern = re.compile(r"(\d{1,2}) de ([a-zA-Z]+) de (\d{4})", re.IGNORECASE)
```

Patrones para detectar el nombre del orador, el número de legislatura, el número de sesión y la fecha en formato textual (00 de … de xxxx).

In [ ]:

```
# 5. Conversion of months
meses = {
    "enero": "01", "febrero": "02", "marzo": "03", "abril": "04", "mayo": "05",
    "junio": "06", "julio": "07", "agosto": "08", "septiembre": "09", "octubre": "10",
    "noviembre": "11", "diciembre": "12"
}
```

Converts month names in text to their numerical value (for example: ‘March’ = ‘03’).

In [ ]:

```
# 6. Auxiliary functions
def extract_fecha_from_text(text):
    match = fecha_pattern.search(text)
    if match:
        dia = match.group(1).zfill(2)
        mes = meses.get(match.group(2).lower(), "00")
        anio = match.group(3)
        return f"{dia}/{mes}/{anio}"
    return "Fecha no encontrada"

def extract_legislatura(text):
    match = legislatura_pattern.search(text)
    return match.group(1) if match else "Desconocida"

def extract_sesion(text):
    match = sesion_pattern.search(text)
    return match.group(1) if match else "Desconocida"

def extract_orador(text):
    match = orador_pattern.search(text)
    if match:
        return match.group(1).strip()
    return "Anónimo"

def extract_context(text, match):
    start = max(match.start() - 50, 0)
    end = min(match.end() + 50, len(text))
    return text[start:end].strip()

def classify_mention(text):
    if any(word in text for word in ["control", "engaño", "manipulación"]):
        return "A"
    elif any(word in text for word in ["dicen", "algunos creen", "se habla"]):
        return "B"
```

```
    elif any(word in text for word in ["no es cierto", "falso", "desmentido"]):
        return "C"
    return "B"
```

The first functions are used to extract the date, legislative session, session, and speaker using the regular expressions already defined.

On the other hand, *def extract_context(text, match)* returns the text surrounding a match to provide context. Finally, with *def classify_mention(text)*, we classify mentions as:

- “A” → explicit
- “B” → implicit
- “C” → refutation.

In [ ]:

```
# 7. Reading PDFs and extracting text
def read_all_pdfs(base_path, subfolders):
    all_data = []
    print(" Iniciando la lectura de PDFs...")
    for subfolder in subfolders:
        folder_path = os.path.join(base_path, subfolder)
        print(f" Explorando carpeta: {folder_path}")
        for file in os.listdir(folder_path):
            if file.lower().endswith(".pdf"):
                file_path = os.path.join(folder_path, file)
                print(f" Leyendo: {file}")
                try:
                    text = ""
                    with fitz.open(file_path) as doc:
                        for page in doc:
                            text += page.get_text()
                        first_page_text = doc[0].get_text() if doc.page_count > 0 else ""
                        fecha_extraida = extract_fecha_from_text(first_page_text)

                    all_data.append({
                        "texto": text,
                        "fecha": fecha_extraida,
                        "partido": "No identificado",
                        "ideologia": "No especificada"
                    })
                except Exception as e:
                    print(f" Error al leer {file}: {e}")
    print(f" Lectura completada. Total de documentos: {len(all_data)}")
    return all_data
```

With this function, we open each PDF in the subfolders and extract the full text, the date from the first page, and return a list of dictionaries with text, date, party, and ideology.

In [ ]:

```
# 8. Process documents and search for patterns
def process_documents(data, regex_patterns):
    results = []
    print(" Procesando documentos para menciones de teorías de conspiración...")
    for idx, item in enumerate(data):
        print(f" Documento {idx + 1}/{len(data)}...")
        if isinstance(item, dict):
            text = item.get("texto", "").lower()
            fecha = item.get("fecha", "Desconocida")
            sesion = extract_sesion(text)
            partido = item.get("partido", "No identificado")
            ideologia = item.get("ideologia", "No especificada")
            legislatura = extract_legislatura(text)
            orador = extract_orador(text)

            for pattern in regex_patterns:
                match = pattern.search(text)
                if match:
                    contexto = extract_context(text, match)
                    clasificacion = classify_mention(contexto)
                    results.append([
                        fecha, sesion, orador, partido,
                        ideologia, match.group(0), contexto, clasificacion
```

```
                    ])
                    print(f" Mención detectada: '{match.group(0)}' → {contexto}")
    print(f" Se encontraron {len(results)} menciones de teorías de conspiración.")
    return results
```

It takes the data read and, for each document, converts the text to lowercase; extracts the session, legislature and speaker; searches for conspiratorial pattern matches; extracts context and classifies it; and finally saves each mention as a row of results.

In [ ]:

```
# 9. Run the analysis
data = read_all_pdfs(base_folder_path, subfolders)
detected_mentions = process_documents(data, regex_patterns)
```

Performs the above functions: Reads PDFs, processes and detects mentions of conspiracies.

In [ ]:

```
# 10. Save results to a CSV file
output_file = "conspiraciones_pdf.csv"
with open(output_file, mode="w", newline="", encoding="utf-8") as file:
    writer = csv.writer(file)
    writer.writerow([
        "Fecha", "No. de Sesión", "Orador", "Partido",
        "Ideología", "Cita", "Contexto", "Tema de Conspiración"
    ])
    for row in detected_mentions:
        writer.writerow(row)


print(f" Archivo guardado: {output_file}")
```

These lines create a CSV file with a header containing all the relevant variables: Date, Session, Speaker, Political Party, Ideology, Quote, Context, and Classification.

In [ ]:

```
# 11. Graphic representations

## Bar chart: Most frequent conspiracy topics
plt.figure(figsize=(10, 6))
conspiracy_counts = df['Cita'].value_counts().head(10)
sns.barplot(x=conspiracy_counts.index, y=conspiracy_counts.values,
            palette=sns.color_palette("viridis", n_colors=len(conspiracy_counts)))
plt.title("Temas de conspiración más frecuentes")
plt.xticks(rotation=45, ha="right")
plt.ylabel("Frecuencia")
plt.tight_layout()
plt.show()

## Pie chart: Distribution of explicit mentions, implicit mentions, and refutations
classification_map = {"A": "Menciones explícitas", "B": "Menciones implícitas", "C": "Refutaciones"}
classification_counts = df['Tema de Conspiración'].value_counts()
labels = [classification_map.get(k, k) for k in classification_counts.index]
sizes = classification_counts.values
colors = ['#ff9999', '#66b3ff', '#99ff99'][:len(sizes)]

plt.figure(figsize=(8, 8))
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=140, colors=colors)
plt.title('Distribución de Menciones Explicitas, Implícitas y Refutaciones')
plt.axis('equal')
plt.show()

## Word cloud: frequently used words in context
texto_completo = " ".join(df["Contexto"].dropna().astype(str))
wordcloud = WordCloud(width=800, height=400, background_color="white", colormap="viridis",
max_words=100).generate(texto_completo)

plt.figure(figsize=(12, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.title("Nube de palabras: términos frecuentes en el discurso")
plt.tight_layout()
```

```
plt.show()
```

FINAL DATABASES

COVID DATABASE

In [ ]:

```
# 1. Importing libraries
import json
import os
import re
import csv
import pandas as pd
from datetime import datetime
import unicodedata
```

Import the libraries: *os*, *json* and *csv*, which are used to manage files; *re* to search for text, *pandas* to work with tables, *datetime* to work with dates (datetime) and *unicodedata* to remove accents.

In [ ]:

```
# Load base of Members of Parliament (MPs) with politic party and ideology
diputados_df = pd.read_excel(r"C:\Users\irene\OneDrive\Escritorio\RESULTADOS\diputados_XIV.xlsx")
```

A pre-prepared Excel file containing the names, parties and ideologies of the MPs is loaded so that each speaker can be labelled.

In [ ]:

```
def quitar_tildes(texto):
    return ''.join(
        c for c in unicodedata.normalize('NFD', texto)
        if unicodedata.category(c) != 'Mn'
    )

def buscar_partido_ideologia(nombre):
    nombre_normalizado = quitar_tildes(nombre.upper())
    diputados_df['NOMBRE_NORMALIZADO'] = diputados_df['NOMBRE'].apply(lambda x: quitar_tildes(str(x).upper()))
    fila = diputados_df[diputados_df['NOMBRE_NORMALIZADO'] == nombre_normalizado]
    if not fila.empty:
        partido = fila.iloc[0]['PARTIDO']
        ideologia = fila.iloc[0]['IDEOLOGIA']
        return partido, ideologia
    else:
        return "No identificado", "No especificada"
```

With these two functions, first: accents are removed and the text is normalised, which is useful for comparing names even if they have accents; and second: the speaker's party and ideology are searched for in Excel by comparing names without accents or capitalisation differences.

In [ ]:

```
# 2. Define folders and files to be processed
folder_path = r"D:\DIARIOS\archivos_extraidos"
files_to_process = ["XIV.json"]
```

Define la carpeta donde están los archivos JSON y cuál procesar (en este caso, el archivo XIV.json).

In [ ]:

```
# 3. Function to read JSON and assign legislature
def read_json_files(folder_path, files):
    all_data = []
    print(f"\U0001F465 Inicio de la lectura de archivos JSON...")
    for file in files:
        legislatura = os.path.splitext(file)[0]
        print(f"\U0001F4C2 Procesando archivo: {file} (Legislatura {legislatura})")
        file_path = os.path.join(folder_path, file)
```

```
        if os.path.exists(file_path):
            with open(file_path, 'r', encoding='utf-8') as f:
                data = json.load(f)
                if isinstance(data, list):
                    for item in data:
                        item["legislatura"] = legislatura
                    all_data.extend(data)
                else:
                    print(f" Advertencia: {file} no es una lista.")
        else:
            print(f" El archivo {file} no existe.")
    print(f" Lectura de archivos completada. Total de documentos: {len(all_data)}")
    return all_data

data = read_json_files(folder_path, files_to_process)
```

It opens the JSON files, adds the name of the legislature to them, and combines all the documents into a list called data.

In [ ]:

```
# 4. List of patterns of conspiracy theories related to COVID-19
conspiracy_patterns = [
    # Terms related to conspiracy theories
    r"(conjura[s]?|complot[s]?|contubernio[s]?|conspiración[es]?|estado[s]? profundo[s]?|teoría[s]?
conspirativa[s]?|teoría[s]? de la conspiración|élite[s]?|conspiranoia[s]?|conspiranoic[oa]s?)",
    # General terms related to COVID
    r"(COVID|covid|covid[- ]?19|coronavirus|estado de
alarma|pandemia[s]?|vacuna[s]?|experimental[es]?|PCR[s]?|mascarilla[s]?|confinamiento[s]?|cuarentena[s]?|SARS[
- ]?CoV[- ]?2|virus)",
    # Terms commonly used in COVID conspiracy theories
    r"(plandemia[s]?|dictadura[s]? sanitaria[s]?|falso[s]? positivo[s]?|microchip[s]?|grafeno|big
pharma|nanopartícula[s]?|laboratorio|virus chino|Soros|Gates|gobierno de la mentira)",
    # Terms of general discrediting
    r"(corrupción[es]?|experimento[s]?|control[es]?|manipulación[es]?|manipulad[oa]s?
|mentira[s]?|engaño[s]?|censura)"
]

regex_patterns = [re.compile(pattern, re.IGNORECASE) for pattern in conspiracy_patterns]
```

Define regular expressions to detect words and phrases typical of conspiracy theories about COVID (such as ‘plandemic’ or ‘graphene’) and compile them for faster searching.

In [ ]:

```
# 5. Auxiliary regular expressions
fecha_pattern = re.compile(r"[a-zA-Z]+ (\d{1,2}) de ([a-zA-Z]+) de (\d{4})")
```

Pattern for recognising dates written in words (e.g. “12 March 2020”).

In [ ]:

```
# 6. Function to format the date
def format_fecha(fecha):
    match = fecha_pattern.search(fecha)
    if match:
        dia = match.group(1).zfill(2)
        mes_texto = match.group(2).lower()
        anio = match.group(3)
        meses = {
            "enero": "01", "febrero": "02", "marzo": "03", "abril": "04", "mayo": "05",
            "junio": "06", "julio": "07", "agosto": "08", "septiembre": "09", "octubre": "10",
            "noviembre": "11", "diciembre": "12"
        }
        mes = meses.get(mes_texto, "00")
        return f"{dia}/{mes}/{anio}"
    try:
        parsed_date = datetime.strptime(fecha, "%Y%m%d")
        return parsed_date.strftime("%d/%m/%Y")
    except ValueError:
        return fecha
```

Convert dates in different formats (such as “20200312” or “12 March 2020”) to a standard dd/mm/yyyy format.

In [ ]:

```
# 7. Auxiliary functions
def extract_presidente(text):
    patron_presidente = r"PRESIDENCIA DEL(?: EXCMO\.| EXCMA\.)?(?: SR\.| SRA\.)?(?: D\.| DÑA\.| DOÑA|
DON)?\s+([A-ZÁÉÍÓÚÑÜ\s]+)"
    match = re.search(patron_presidente, text.upper())
    if match:
        nombre = match.group(1).strip()
        nombre = re.sub(r"\s+", " ", nombre)
        return nombre.title()
    return None

def extract_orador(texto_antes_mencion, presidente=None):
    posibles = re.findall(r"([A-ZÁÉÍÓÚÑÜ\s]{2,}):", texto_antes_mencion)
    candidatos = []
    for p in posibles:
        palabras = p.strip().split()
        palabras_validas = [w for w in palabras if len(w) > 1]
        if len(palabras_validas) >= 2:
            candidato = " ".join(palabras_validas).title()
            candidatos.append(candidato)
    if candidatos:
        return candidatos[-1]
    elif presidente:
        return presidente
    else:
        return "Desconocido"

def extract_context(text, match):
    start_pos = match.start()
    end_pos = match.end()

    # Increase margin
    pre_window = 1500
    post_window = 1500
    start_context = max(0, start_pos - pre_window)
    end_context = min(len(text), end_pos + post_window)

    # Extract fragment with more margin
    snippet = text[start_context:end_context]

    # Find the nearest paragraph break before and after
    parrafo_inicio = snippet[:start_pos - start_context].rfind('\n\n')
    parrafo_fin = snippet[end_pos - start_context:].find('\n\n')

    context_start = start_context + parrafo_inicio + 2 if parrafo_inicio != -1 else start_context
    context_end = end_pos + parrafo_fin if parrafo_fin != -1 else end_context

    context = text[context_start:context_end].strip()

    # Ensure that the quotation is included (in case the paragraph break omits it).
    if match.group(0).lower() not in context.lower():
        context = text[start_context:end_context].strip()

    cita = match.group(0)

    # Highlight all occurrences of the quote in context
    context_resaltado = re.sub(re.escape(cita), f"**{cita}**", context, flags=re.IGNORECASE)

    return context_resaltado

def tiene_mentalidad_conspirativa(texto):
    texto = texto.lower()
    patrones_mentales = [
        r"ocultan?", r"manipulan?", r"esconden?", r"controlan?", r"engañan?",
        r"mentira(s)?", r"engaño(s)?", r"verdad escondida", r"no se dice todo",
        r"intereses oscuros", r"poder oculto", r"información oculta", r"no nos cuentan",
        r"sospechoso(s)?", r"falta de transparencia", r"intereses que no se revelan",
        r"engañosa(s)?", r"engaños?", r"manipulación", r"manipulado(s)?", r"secretismo"
     ]
    for patron in patrones_mentales:
        if re.search(patron, texto):
            return True
    return False

def classify_mention(context):
    context_lower = context.lower()
```

```
    refutacion_patrones = [
        r"\b(bulo(s)?|mentira(s)?|falsedad(es)?|desinformación|es falso|es mentira|no es verdad|ha sido
desmentido|carece de fundamento|infundado|infundada|no tiene base|ha sido refutado|teoría sin
base|conspiración infundada)\b",

r"\b(no\s+hay\s+(pruebas|evidencia|fundamento)|nunca\s+se\s+demostró|no\s+es\s+cierto|no\s+es\s+real|es\s+una\
s+invención)\b"
    ]
    if any(re.search(p, context_lower) for p in refutacion_patrones):
        return "C"
    if tiene_mentalidad_conspirativa(context):
        return "B"
    mencion_teoria_patrones = [

r"\b(nuevo\s+orden\s+mundial|plandemia|élite(s)?|gran\s+reseteo|agenda\s+2030|microchip(s)?|dictadura\s+sanita
ria|teoría\s+conspirativa|teoría\s+de\s+la\s+conspiración|conspiración\s|conspiración\s+global|élite(s)?\s+mun
dial(es)?|gobierno\s+en\s+la\s+sombra|plandemic)\b",

r"\b(vacuna(s)?\s+como\s+instrumento\s+de\s+control|el\s+virus\s+fue\s+creado|los\s+medios\s+están\s+comprados
|falsa\s+pandemia)\b"
    ]
    if any(re.search(p, context_lower) for p in mencion_teoria_patrones):
        return "A"
    return "N"

def determinar_expresion(contexto, clasificacion):
    contexto = contexto.lower()

    # Very explicit expression patterns (direct quote from the theory)
    patrones_explicitos_directos = [
        r"\bplandemia\b", r"\bla pandemia es falsa\b", r"\bel covid no existe\b",
        r"\bnos han implantado\b", r"\bnos quieren controlar con microchips\b",
        r"\bvacuna como instrumento de control\b", r"\bla vacuna tiene grafeno\b",
        r"\blas élites nos controlan\b", r"\bel nuevo orden mundial\b"
        ]

    # Implicit language typical of political/parliamentary discourse
    patrones_implicitos = [
        r"\bnos mienten\b", r"\bno nos cuentan\b", r"\bintereses ocultos\b",
        r"\bfalta de transparencia\b", r"\bno es toda la verdad\b",
        r"\bmanipulación\b", r"\bengaño\b", r"\bsecretismo\b",
        r"\binformación que no se dice\b", r"\bnos han ocultado\b",
        r"\bsospechamos que\b", r"\bparece que hay algo detrás\b",
        r"\bdecisiones que no se explican\b", r"\bverdad escondida\b",
        r"\bcontrolan la narrativa\b", r"\bpoder en la sombra\b"
    ]

    # If it is a rebuttal (classification C)
    if clasificacion == "C":
        # Solo se considera explícita si el lenguaje reproduce textualmente la teoría para negarla
        if any(re.search(p, contexto) for p in patrones_explicitos_directos):
            return "Explícita"
        else:
            return "Implícita"

    # If the language is very clear and direct: explicit
    if any(re.search(p, contexto) for p in patrones_explicitos_directos):
        return "Explícita"

    # If there is suggestive, indirect or ambiguous language: implicit
    if any(re.search(p, contexto) for p in patrones_implicitos):
        return "Implícita"

    # If classified as A, B, or C but doesn't use plain language
    if clasificacion in ["A", "B", "C"]:
        return "Implícita"

    # If there is no evidence of a conspiracy theory
    return "No procede"
```

Support functions for:

- *extract_presidente*: attempts to identify who is chairing the session based on the text.
- *extract_orador*: detects who is speaking just before the mention found.
- *extract_context*: extracts the text fragment around the word found and highlights the quote in bold.

- *tiene_mentalidad_conspirativa*: identifies language that suggests conspiratorial thinking (such as ‘they are hiding it from us,’ ‘they are deceiving us,’ etc.).
- *classify_mention*: classifies each mention as A (theory), B (mindset), C (refutation) or N (none).
- *determinar_expresion*: indicates whether the mention is Explicit, Implicit or Not applicable, depending on how the conspiratorial idea is expressed.

In [ ]:

```
# 8. Process documents
def process_documents(data, regex_patterns):
    results = []
    print(" Procesando documentos para menciones de teorías de conspiración...")
    for idx, item in enumerate(data):
        text_original = item.get("texto", "")
        text = text_original.lower()
        fecha = item.get("fecha", "Desconocida")
        legislatura = item.get("legislatura", "Desconocida")
        doc_id = item.get("id", "Sin ID")
        doc_id_str = str(doc_id)

        numero_sesion_match = re.search(r"(?:sesión\s*)?(?:n[úu]m(?:\.|ero)?)[\s:]*?(\d+)", text,
re.IGNORECASE)
        numero_sesion = numero_sesion_match.group(1) if numero_sesion_match else "Desconocido"

        documento_match = re.search(r"\b(?:BOCG[_-]|DS(?:CD|CG)?[_-])[A-Z0-9_-]+", text)
        documento = documento_match.group() if documento_match else item.get("cve", "Desconocido")

        fecha_formateada = format_fecha(fecha)
        presidente = extract_presidente(text)

        print(f" Documento {idx + 1}/{len(data)}")
        ya_detectados = set()

        for grupo_index, pattern in enumerate(regex_patterns, start=1):
            for match in pattern.finditer(text):
                contexto = extract_context(text, match)
                texto_hasta_mencion = text_original[:match.start()]
                orador_nombre = extract_orador(texto_hasta_mencion, presidente)
                clave = f"{orador_nombre}_{doc_id_str}"
                if clave in ya_detectados:
                    continue
                ya_detectados.add(clave)
                partido, ideologia = buscar_partido_ideologia(orador_nombre)
                clasificacion = classify_mention(contexto)
                dic_conspiracion = "Sí" if clasificacion in ["A", "B", "C"] else "No"
                grupo_texto = f"Grupo {grupo_index}"
                modo_exp_implic = determinar_expresion(contexto, clasificacion) if dic_conspiracion == "Sí"
else "No procede"

                results.append([
                    doc_id_str, fecha_formateada, legislatura, documento, numero_sesion, orador_nombre,
                    partido, ideologia, grupo_texto, match.group(0), contexto, clasificacion,
dic_conspiracion, modo_exp_implic
                ])

        # Second pass for conspiracy mentality in full text
        orador_general = extract_orador(text_original, presidente)
        clave_general = f"{orador_general}_{doc_id_str}"
        if clave_general not in ya_detectados:
            clasificacion_general = classify_mention(text_original)
            if clasificacion_general == "B":
                partido, ideologia = buscar_partido_ideologia(orador_general)
                results.append([
                    doc_id_str, fecha_formateada, legislatura, documento, numero_sesion, orador_general,
                    partido, ideologia, "Intervención completa", "", text_original[:1000],
clasificacion_general, "Sí", "Implícita"
                ])
                ya_detectados.add(clave_general)

    print(f" Se encontraron {len(results)} menciones de teorías de conspiración.")
    return results
```

Go through all the JSON texts, look for matches with conspiracy theories, identify the speaker and their party, classify the mention, and save everything in a list of results.

In [ ]:

```
# 9. Generate CSV
detected_mentions = process_documents(data, regex_patterns)
output_folder = r"C:\Users\irene\OneDrive\Escritorio\RESULTADOS"
os.makedirs(output_folder, exist_ok=True)
output_file = os.path.join(output_folder, "conspiraciones.csv")
columnas = ["ID", "Fecha", "Legislatura", "Documento", "Numero de Sesion", "Orador", "Partido", "Ideologia",
"Grupo", "Cita", "Contexto", "Clasificacion", "Si/No Conspiracion", "Explicita/Implicita"]
df_resultados = pd.DataFrame(detected_mentions, columns=columnas)
df_resultados.to_csv(output_file, sep=';', quoting=csv.QUOTE_ALL, index=False, encoding='utf-8')
print(f" Archivo guardado: {output_file}")
```

Convert the results into a *pandas* DataFrame and save them to a CSV file with columns such as date, speaker, politic party, quote, ranking, etc.

In [ ]:

```
# 10. Show result
df = pd.read_csv(output_file, delimiter=';')
print(df.head())
```

Load the newly created CSV and display the first few rows to verify that everything was processed correctly.

POWER OUTAGE DATABASE (“APAGÓN DE ABRIL”)

In [ ]:

```
# 1. Importing libraries
import json
import os
import re
import csv
import pandas as pd
from datetime import datetime
import unicodedata
```

Import libraries to manage files (os, json, csv), search for text (re), work with data (pandas), dates (datetime), and remove accents (unicodedata).

In [ ]:

```
# Load base of Members of Parliament (MPs) with politic party and ideology
diputados_df = pd.read_excel(r"C:\Users\irene\OneDrive\Escritorio\RESULTADOS\diputados_XV.xlsx")
```

Load a pre-prepared Excel file with the names of the MPs, their party and ideology, which will be used to identify and classify the speakers in the text.

In [ ]:

```
def quitar_tildes(texto):
    return ''.join(
        c for c in unicodedata.normalize('NFD', texto)
        if unicodedata.category(c) != 'Mn'
    )

def buscar_partido_ideologia(nombre):
    nombre_normalizado = quitar_tildes(nombre.upper())
    diputados_df['NOMBRE_NORMALIZADO'] = diputados_df['NOMBRE'].apply(lambda x: quitar_tildes(str(x).upper()))
    fila = diputados_df[diputados_df['NOMBRE_NORMALIZADO'] == nombre_normalizado]
    if not fila.empty:
        partido = fila.iloc[0]['PARTIDO']
        ideologia = fila.iloc[0]['IDEOLOGÍA']
        return partido, ideologia
    else:
        return "No identificado", "No especificada"
```

With these two functions, first: accents are removed and the text is normalised, which is useful for comparing names even if they have accents; and second: the speaker's party and ideology are searched for in Excel by comparing names without accents or capitalisation differences.

In [ ]:

```
# 2. Define folders and files to be processed
folder_path = r"D:\DIARIOS\APAGON_MAYO"
files_to_process = ["APAGON_MAYO.json"]
```

Specify the folder where the JSON files are located and which one to process (in this case, APAGON_MAYO.json).

In [ ]:

```
# 3. Reading JSON files
def read_json_files(folder_path, files):
    all_data = []
    print(f" Inicio de la lectura de archivos JSON...")
    for file in files:
        print(f" Procesando archivo: {file}")
        file_path = os.path.join(folder_path, file)
        if os.path.exists(file_path):
            with open(file_path, 'r', encoding='utf-8') as f:
                data = json.load(f)
                if isinstance(data, list):
                    all_data.extend(data)
                else:
                    print(f" Advertencia: {file} no es una lista.")
        else:
            print(f" El archivo {file} no existe.")
    print(f" Lectura de archivos completada. Total de documentos: {len(all_data)}")
    return all_data

data = read_json_files(folder_path, files_to_process)
```

It opens the JSON files, goes through them one by one, loads their content, and saves everything in a data list. It also prints the progress and the total number of documents loaded.

In [ ]:

```
# 4. Patterns of mention
apagon_patterns = [
    r"\bapag[oó]n(es)?\b",
    r"\bcolapso energ[eé]tico\b",
    r"\bfallo(s)? el[eé]ctrico(s)?\b",
    r"\bproblema(s)? el[eé]ctrico(s)?\b",
    r"\bcrisis energ[eé]tica\b",
    r"\binterrupci[oó]n de suministro\b",
    r"\bdesconexi[oó]n el[eé]ctrica\b",
    r"\bfalta de energ[ií]a\b",
    r"\bfalla de luz\b",
    r"\bcorte de luz\b",
    r"\bapag[oó]n general\b",
    r"\bracionamiento energ[eé]tico\b",
    r"\belectricidad\b",
    r"\bluz\b",
    r"\bsuministro\b",
    r"\benerg[ií]a\b",
    r"\bservicio el[eé]ctrico\b",
    r"\binterrupci[oó]n\b",
    r"\bcolapso\b",
    r"\bincidente(s)? el[eé]ctrico(s)?\b",
    r"\bsin (luz|electricidad|energ[ií]a)\b",
    r"\brenovables\b",
    r"\bnucleares\b",
    r"\bMarruecos\b",
    r"\bexplosi[oó]n\b",
    r"\bAlemania\b",
    r"\bincendio\b",
    r"\bRusia\b",
    r"\bruso\b",
    r"\bataque\b",
    r"\batmosf[eé]rico(a)?\b",
    r"\bchantaje\b",
    r"\bsimulacro\b"
]

regex_patterns = [re.compile(pattern, re.IGNORECASE) for pattern in apagon_patterns]
```

Define a list of words and expressions that may refer to power cuts or energy crises (such as ‘power cut’, “blackout”, ‘energy collapse’, etc.) and convert them into regular expressions to search for them in the text.

In [ ]:

```
# 5. Auxiliary regex for dates
fecha_pattern = re.compile(r"[a-zA-Z]+ (\d{1,2}) de ([a-zA-Z]+) de (\d{4})")
```

Regular expression to recognise dates written in words, for example “15 May 2023”.

In [ ]:

```
# 6. Date formatting
def format_fecha(fecha):
    if not fecha:
        return "00/00/0000"
    match = re.search(r"(\d{1,2}) de ([a-zA-Záéíóúñ]+) de (\d{4})", fecha, re.IGNORECASE)
    if match:
        dia = match.group(1).zfill(2)
        mes_texto = match.group(2).lower()
        anio = match.group(3)
        meses = {
            "enero": "01", "febrero": "02", "marzo": "03", "abril": "04", "mayo": "05",
            "junio": "06", "julio": "07", "agosto": "08", "septiembre": "09", "octubre": "10",
            "noviembre": "11", "diciembre": "12"
        }
        mes = meses.get(mes_texto, "00")
        return f"{dia}/{mes}/{anio}"
    for fmt in ("%Y%m%d", "%Y-%m-%d", "%d-%m-%Y", "%d/%m/%Y"):
        try:
            parsed_date = datetime.strptime(fecha, fmt)
            return parsed_date.strftime("%d/%m/%Y")
        except ValueError:
            continue
    return "00/00/0000"
```

Converts different date formats such as “20230515” or “15 May 2023” to the uniform format dd/mm/yyyy. If it cannot be interpreted, it returns “00/00/0000”.

In [ ]:

```
# 7. Auxiliary functions
def extract_presidente(text):
    patron_presidente = r"PRESIDENCIA DEL(?: EXCMO\.| EXCMA\.)?(?: SR\.| SRA\.)?(?: D\.| DÑA\.| DOÑA|
DON)?\s+([A-ZÁÉÍÓÚÑÜ\s]+)"
    match = re.search(patron_presidente, text.upper())
    if match:
        nombre = match.group(1).strip()
        nombre = re.sub(r"\s+", " ", nombre)
        return nombre.title()
    return None

def extract_orador(texto_antes_mencion, presidente=None):
    posibles = re.findall(r"([A-ZÁÉÍÓÚÑÜ\s]{2,}):", texto_antes_mencion)
    candidatos = []
    for p in posibles:
        palabras = p.strip().split()
        palabras_validas = [w for w in palabras if len(w) > 1]
        if len(palabras_validas) >= 2:
            candidato = " ".join(palabras_validas).title()
            candidatos.append(candidato)
    if candidatos:
        return candidatos[-1]
    elif presidente:
        return presidente
    else:
        return "Desconocido"

def extract_context(text, match):
    start_pos = match.start()
    end_pos = match.end()
    pre_window = 1500
    post_window = 1500
    start_context = max(0, start_pos - pre_window)
```

```
    end_context = min(len(text), end_pos + post_window)
    snippet = text[start_context:end_context]
    parrafo_inicio = snippet[:start_pos - start_context].rfind('\n\n')
    parrafo_fin = snippet[end_pos - start_context:].find('\n\n')
    context_start = start_context + parrafo_inicio + 2 if parrafo_inicio != -1 else start_context
    context_end = end_pos + parrafo_fin if parrafo_fin != -1 else end_context
    context = text[context_start:context_end].strip()
    if match.group(0).lower() not in context.lower():
        context = text[start_context:end_context].strip()
    cita = match.group(0)
    context_resaltado = re.sub(re.escape(cita), f"**{cita}**", context, flags=re.IGNORECASE)
    return context_resaltado
```

Support functions for:

extract_presidente: attempts to identify who is chairing the session.

extract_orador: detects who is speaking before the mention.

extract_context: extracts the text fragment around the word found and highlights the quote in bold.

In [ ]:

```
# 8. Document processing (modified to use regular expression patterns)
def process_documents(data, regex_patterns):
    results = []
    print(" Procesando documentos para menciones de apagón...")
    for idx, item in enumerate(data):
        text_original = item.get("Texto", "")
        text = text_original
        fecha = item.get("Fecha", "").strip()
        if not fecha:
            fecha_match = re.search(r"(\d{1,2} de [a-záéíóúñ]+ de \d{4})", text_original, re.IGNORECASE)
            if fecha_match:
                fecha = fecha_match.group(1)
        fecha_formateada = format_fecha(fecha)
        legislatura = item.get("Legislatura", "Desconocida")
        doc_id = item.get("ID", "Sin ID")
        numero_sesion = item.get("Numero de Sesion", "Desconocido")
        documento = item.get("Documento", "Desconocido")
        presidente = extract_presidente(text)
        print(f" Documento {idx + 1}/{len(data)}")
        ya_detectados = set()

        # Direct use of regex patterns without simplified list
        for pat in regex_patterns:
            for match in pat.finditer(text):
                contexto = extract_context(text, match)
                texto_hasta_mencion = text[:match.start()]
                orador_nombre = extract_orador(texto_hasta_mencion, presidente)
                clave = f"{orador_nombre}_{doc_id}"
                if clave in ya_detectados:
                    continue
                ya_detectados.add(clave)
                partido, ideologia = buscar_partido_ideologia(orador_nombre)
                results.append([
                    str(doc_id), fecha_formateada, legislatura, documento, numero_sesion,
                    orador_nombre, partido, ideologia, match.group(0), contexto
                ])

    print(f" Se encontraron {len(results)} menciones relacionadas con apagón.")
    return results
```

Scan the entire text of the documents:

- Search for words related to blackouts using the patterns.
- Extract the context in which they appear.
- Detect who is speaking, their party and ideology (from Excel).
- Save all this information (document, date, speaker, quote, context) in a *results* list.

In [ ]:

```
# 9. Generate CSV
detected_mentions = process_documents(data, regex_patterns)
output_folder = r"C:\Users\irene\OneDrive\Escritorio\RESULTADOS"
os.makedirs(output_folder, exist_ok=True)
output_file = os.path.join(output_folder, "apagon.csv")
columnas = ["ID", "Fecha", "Legislatura", "Documento", "Numero de Sesion", "Orador", "Partido", "Ideologia",
"Cita", "Contexto"]
df_resultados = pd.DataFrame(detected_mentions, columns=columnas)
df_resultados.to_csv(output_file, sep=';', quoting=csv.QUOTE_ALL, index=False, encoding='utf-8')
print(f" Archivo guardado: {output_file}")
```

Convert the results into a table (DataFrame) and save them in a CSV file called *apagon.csv*, with the main columns: ID, Date, Legislature, Speaker, Politic Party, Quote, Context, etc.

In [ ]:

```
# 10. Show results
df = pd.read_csv(output_file, delimiter=';')
print(df.head())
```

Read the generated CSV and display the first rows on the console to verify that processing was completed successfully.

POST-FRANCOISM DATABASE

In [ ]:

```
# 1. Importing libraries
import json
import os
import re
import csv
import pandas as pd
from datetime import datetime
import unicodedata
```

Import libraries to manage files (*os, json, csv*), search for text (*re*), work with data (*pandas*), dates (*datetime*) and remove accents (*unicodedata*).

In [ ]:

```
# Load base of Members of Parliament (MPs) with politic party and ideology
diputados_df = pd.read_excel(r"D:\TRABAJO\DIPUTADOS\DIPUTADOS+77.xlsx")
```

Load a pre-prepared Excel file with the names of the MPs, their party and ideology, which will be used to identify and classify the speakers in the text.

In [ ]:

```
def quitar_tildes(texto):
    return ''.join(
        c for c in unicodedata.normalize('NFD', texto)
        if unicodedata.category(c) != 'Mn'
    )

def buscar_partido_ideologia(nombre):
    nombre_normalizado = quitar_tildes(nombre.upper())
    diputados_df['NOMBRE_NORMALIZADO'] = diputados_df['NOMBRE'].apply(lambda x: quitar_tildes(str(x).upper()))
    fila = diputados_df[diputados_df['NOMBRE_NORMALIZADO'] == nombre_normalizado]
    if not fila.empty:
        partido = fila.iloc[0]['PARTIDO']
        ideologia = fila.iloc[0]['IDEOLOGIA']
        return partido, ideologia
    else:
        return "No identificado", "No especificada"
```

With these two functions, first: accents are removed and the text is normalised, which is useful for comparing names even if they have accents; and second: the speaker's party and ideology are searched for in Excel by comparing names without accents or capitalisation differences.

In [ ]:

```
# 2. Define folders and files to be processed
folder_path = r"D:\TRABAJO\DIARIOS\archivos_extraidos"
files_to_process = ["C.json", "I.json", "II.json", "III.json", "IV.json", "V.json", "VI.json", "VII.json",
                    "VIII.json", "IX.json", "X.json", "XI.json", "XII.json", "XIII.json", "XIV.json",
"XV.json"]
```

Specify the folder where the JSON files are located and which one to process (in this case, all legislatures covering the period after 1977).

In [ ]:

```
# 3. Function to read JSON and assign legislature
def read_json_files(folder_path, files):
    all_data = []
    print(f"\U0001F465 Inicio de la lectura de archivos JSON...")
    for file in files:
        legislatura = os.path.splitext(file)[0]
        print(f"\U0001F4C2 Procesando archivo: {file} (Legislatura {legislatura})")
        file_path = os.path.join(folder_path, file)
        if os.path.exists(file_path):
            with open(file_path, 'r', encoding='utf-8') as f:
                data = json.load(f)
                if isinstance(data, list):
                    for item in data:
                        item["legislatura"] = legislatura
                    all_data.extend(data)
                else:
                    print(f" Advertencia: {file} no es una lista.")
        else:
            print(f" El archivo {file} no existe.")
    print(f" Lectura de archivos completada. Total de documentos: {len(all_data)}")
    return all_data

data = read_json_files(folder_path, files_to_process)
```

Read each JSON, add the legislature field according to the file name, and put everything together in a data list, with progress messages.

In [ ]:

```
# 4. List of patterns of conspiracy theories
conspiracy_patterns = [
    # Términos relacionados con teorías conspirativas

r"(negacionista[s]?|negacionismo|terraplanista[s]?|terraplanismo|conjura[s]?|complot[s]?|contubernio[s]?|consp
iración[es]?|estado[s]? profundo[s]?|teoría[s]? conspirativa[s]?|teoría[s]? de la
conspiración|élite[s]?|conspiranoia[s]?|conspiranoic[oa]s?)",
    # Términos generales relacionados con el COVID
    r"(COVID|covid|covid[- ]?19|coronavirus|estado de
alarma|pandemia[s]?|vacuna[s]?|experimental[es]?|PCR[s]?|mascarilla[s]?|confinamiento[s]?|cuarentena[s]?|SARS[
- ]?CoV[- ]?2|virus)",
    # Términos que suelen usarse en teorías de conspiración sobre el COVID
    r"(plandemia[s]?|dictadura[s]? sanitaria[s]?|falso[s]? positivo[s]?|microchip[s]?|grafeno|big
pharma|nanopartícula[s]?|laboratorio|virus chino|Soros|Gates|gobierno de la mentira)",
    # Términos de desacreditación general
    r"(corrupción[es]?|experimento[s]?|control[es]?|manipulación[es]?|manipulad[oa]s?
|mentira[s]?|engaño[s]?|censura)"
]

regex_patterns = [re.compile(pattern, re.IGNORECASE) for pattern in conspiracy_patterns]
```

Define key words/expressions related to COVID and conspiracies (denialism, ‘plandemic’, ‘graphene’, etc.) and compile them as regex to search for them in the texts.

In [ ]:

```
# 5. Auxiliary regular expressions
fecha_pattern = re.compile(r"[a-zA-Z]+ (\d{1,2}) de ([a-zA-Z]+) de (\d{4})")
```

Pattern for recognising dates such as “12 May 2020”.

In [ ]:

```
# 6. Function to format the date
def format_fecha(fecha):
    match = fecha_pattern.search(fecha)
    if match:
        dia = match.group(1).zfill(2)
        mes_texto = match.group(2).lower()
        anio = match.group(3)
        meses = {
            "enero": "01", "febrero": "02", "marzo": "03", "abril": "04", "mayo": "05",
            "junio": "06", "julio": "07", "agosto": "08", "septiembre": "09", "octubre": "10",
            "noviembre": "11", "diciembre": "12"
        }
        mes = meses.get(mes_texto, "00")
        return f"{dia}/{mes}/{anio}"
    try:
        parsed_date = datetime.strptime(fecha, "%Y%m%d")
        return parsed_date.strftime("%d/%m/%Y")
    except ValueError:
        return fecha
```

Converts dates to dd/mm/yyyy format from expressions with month in text or from YYYYMMDD strings. If it does not fit, it returns the original.

In [ ]:

```
# 7. Auxiliary functions
def extract_presidente(text):
    patron_presidente = r"PRESIDENCIA DEL(?: EXCMO\.| EXCMA\.)?(?: SR\.| SRA\.)?(?: D\.| DÑA\.| DOÑA|
DON)?\s+([A-ZÁÉÍÓÚÑÜ\s]+)"
    match = re.search(patron_presidente, text.upper())
    if match:
        nombre = match.group(1).strip()
        nombre = re.sub(r"\s+", " ", nombre)
        return nombre.title()
    return None

def extract_orador(texto_antes_mencion, presidente=None):
    posibles = re.findall(r"([A-ZÁÉÍÓÚÑÜ\s]{2,}):", texto_antes_mencion)
    candidatos = []
    for p in posibles:
        palabras = p.strip().split()
        palabras_validas = [w for w in palabras if len(w) > 1]
        if len(palabras_validas) >= 2:
            candidato = " ".join(palabras_validas).title()
            candidatos.append(candidato)
    if candidatos:
        return candidatos[-1]
    elif presidente:
        return presidente
    else:
        return "Desconocido"

def extract_context(text, match):
    start_pos = match.start()
    end_pos = match.end()

    # Increase margin
    pre_window = 1500
    post_window = 1500

    start_context = max(0, start_pos - pre_window)
    end_context = min(len(text), end_pos + post_window)

    snippet = text[start_context:end_context]

    # Find the nearest paragraph break before and after
    parrafo_inicio = snippet[:start_pos - start_context].rfind('\n\n')
    parrafo_fin = snippet[end_pos - start_context:].find('\n\n')

    context_start = start_context + parrafo_inicio + 2 if parrafo_inicio != -1 else start_context
    context_end = end_pos + parrafo_fin if parrafo_fin != -1 else end_context

    context = text[context_start:context_end].strip()

    # Ensure that the quote is included (in case the paragraph break omits it).
```

```
    if match.group(0).lower() not in context.lower():
        context = text[start_context:end_context].strip()

    cita = match.group(0)
    # Resaltar todas las apariciones de la cita en el contexto
    context_resaltado = re.sub(re.escape(cita), f"**{cita}**", context, flags=re.IGNORECASE)

    return context_resaltado

def tiene_mentalidad_conspirativa(texto):
    texto = texto.lower()
    patrones_mentales = [
        r"ocultan?", r"manipulan?", r"esconden?", r"controlan?", r"engañan?",
        r"mentira(s)?", r"engaño(s)?", r"verdad escondida", r"no se dice todo",
        r"intereses oscuros", r"poder oculto", r"información oculta", r"no nos cuentan",
        r"sospechoso(s)?", r"falta de transparencia", r"intereses que no se revelan",
        r"engañosa(s)?", r"engaños?", r"manipulación", r"manipulado(s)?", r"secretismo"
    ]
    for patron in patrones_mentales:
        if re.search(patron, texto):
            return True
    return False

def classify_mention(context):
    context_lower = context.lower()
    refutacion_patrones = [
        r"\b(bulo(s)?|mentira(s)?|falsedad(es)?|desinformación|es falso|es mentira|no es verdad|ha sido
desmentido|carece de fundamento|infundado|infundada|no tiene base|ha sido refutado|teoría sin
base|conspiración infundada)\b",

r"\b(no\s+hay\s+(pruebas|evidencia|fundamento)|nunca\s+se\s+demostró|no\s+es\s+cierto|no\s+es\s+real|es\s+una\
s+invención)\b"
    ]
    if any(re.search(p, context_lower) for p in refutacion_patrones):
        return "C"
    if tiene_mentalidad_conspirativa(context):
        return "B"
    mencion_teoria_patrones = [

r"\b(nuevo\s+orden\s+mundial|plandemia|élite(s)?|gran\s+reseteo|agenda\s+2030|microchip(s)?|dictadura\s+sanita
ria|teoría\s+conspirativa|teoría\s+de\s+la\s+conspiración|conspiración\s|conspiración\s+global|élite(s)?\s+mun
dial(es)?|gobierno\s+en\s+la\s+sombra|plandemic)\b",

r"\b(vacuna(s)?\s+como\s+instrumento\s+de\s+control|el\s+virus\s+fue\s+creado|los\s+medios\s+están\s+comprados
|falsa\s+pandemia)\b"
    ]
    if any(re.search(p, context_lower) for p in mencion_teoria_patrones):
        return "A"
    return "N"

def determinar_expresion(contexto, clasificacion):
    contexto = contexto.lower()

    # Very explicit expression patterns (direct theory quote)
    patrones_explicitos_directos = [
        r"\bplandemia\b", r"\bla pandemia es falsa\b", r"\bel covid no existe\b",
        r"\bnos han implantado\b", r"\bnos quieren controlar con microchips\b",
        r"\bvacuna como instrumento de control\b", r"\bla vacuna tiene grafeno\b",
        r"\blas élites nos controlan\b", r"\bel nuevo orden mundial\b"
    ]

    # Implicit language typical of political/parliamentary discourse
    patrones_implicitos = [
        r"\bnos mienten\b", r"\bno nos cuentan\b", r"\bintereses ocultos\b",
        r"\bfalta de transparencia\b", r"\bno es toda la verdad\b",
        r"\bmanipulación\b", r"\bengaño\b", r"\bsecretismo\b",
        r"\binformación que no se dice\b", r"\bnos han ocultado\b",
        r"\bsospechamos que\b", r"\bparece que hay algo detrás\b",
        r"\bdecisiones que no se explican\b", r"\bverdad escondida\b",
        r"\bcontrolan la narrativa\b", r"\bpoder en la sombra\b"
    ]

    # If it is a rebuttal (classification C)
    if clasificacion == "C":
        # It is only considered explicit if the language reproduces the theory verbatim in order to refute it.
        if any(re.search(p, contexto) for p in patrones_explicitos_directos):
            return "Explícita"
```

```
        else:
            return "Implícita"

    # If the language is very clear and direct: explicit
    if any(re.search(p, contexto) for p in patrones_explicitos_directos):
        return "Explícita"

    # If there is suggestive, indirect or ambiguous language: implicit
    if any(re.search(p, contexto) for p in patrones_implicitos):
        return "Implícita"

    # If you have been classified as A, B or C but do not use clear language
    if clasificacion in ["A", "B", "C"]:
        return "Implícita"

    # If there is no evidence of a conspiracy theory
    return "No procede"
```

Support functions for:

- *extract_presidente*: attempts to identify who is chairing the session based on the text.
- *extract_orador*: detects who is speaking just before the mention found.
- *extract_context*: extracts the text fragment around the word found and highlights the quote in bold.
- *tiene_mentalidad_conspirativa*: identifies language that suggests conspiratorial thinking (such as 'they are hiding it from us,' 'they are deceiving us,' etc.).
- *classify_mention*: classifies each mention as A (theory), B (mindset), C (refutation) or N (none).
- *determinar_expresion*: indicates whether the mention is Explicit, Implicit or Not applicable, depending on how the conspiratorial idea is expressed.

In [ ]:

```
# 8. Processing documents
def process_documents(data, regex_patterns):
    results = []
    print(" Procesando documentos para menciones de teorías de conspiración...")
    for idx, item in enumerate(data):
        text_original = item.get("texto", "")
        text = text_original.lower()
        fecha = item.get("fecha", "Desconocida")
        legislatura = item.get("legislatura", "Desconocida")
        doc_id = item.get("id", "Sin ID")
        doc_id_str = str(doc_id)

        numero_sesion_match = re.search(r"(?:sesión\s*)?(?:n[úu]m(?:\.|ero)?)[\s:]*?(\d+)", text,
re.IGNORECASE)
        numero_sesion = numero_sesion_match.group(1) if numero_sesion_match else "Desconocido"

        documento_match = re.search(r"\b(?:BOCG[_-]|DS(?:CD|CG)?[_-])[A-Z0-9_-]+", text)
        documento = documento_match.group() if documento_match else item.get("cve", "Desconocido")

        fecha_formateada = format_fecha(fecha)
        presidente = extract_presidente(text)

        print(f" Documento {idx + 1}/{len(data)}")
        ya_detectados = set()

        for grupo_index, pattern in enumerate(regex_patterns, start=1):
            for match in pattern.finditer(text):
                contexto = extract_context(text, match)
                texto_hasta_mencion = text_original[:match.start()]
                orador_nombre = extract_orador(texto_hasta_mencion, presidente)
                clave = f"{orador_nombre}_{doc_id_str}"
                if clave in ya_detectados:
                    continue
                ya_detectados.add(clave)
                partido, ideologia = buscar_partido_ideologia(orador_nombre)
                clasificacion = classify_mention(contexto)
                dic_conspiracion = "Sí" if clasificacion in ["A", "B", "C"] else "No"
                grupo_texto = f"Grupo {grupo_index}"
                modo_exp_implic = determinar_expresion(contexto, clasificacion) if dic_conspiracion == "Sí"
else "No procede"

                results.append([
                    doc_id_str, fecha_formateada, legislatura, documento, numero_sesion, orador_nombre,
                    partido, ideologia, grupo_texto, match.group(0), contexto, clasificacion,
```

```
dic_conspiracion, modo_exp_implic
                ])

        # Second pass for conspiracy mentality in full text
        orador_general = extract_orador(text_original, presidente)
        clave_general = f"{orador_general}_{doc_id_str}"
        if clave_general not in ya_detectados:
            clasificacion_general = classify_mention(text_original)
            if clasificacion_general == "B":
                partido, ideologia = buscar_partido_ideologia(orador_general)
                results.append([
                    doc_id_str, fecha_formateada, legislatura, documento, numero_sesion, orador_general,
                    partido, ideologia, "Intervención completa", "", text_original[:1000],
clasificacion_general, "Sí", "Implícita"
                ])
                ya_detectados.add(clave_general)

    print(f" Se encontraron {len(results)} menciones de teorías de conspiración.")
    return results
```

It processes each document and searches for patterns, extracts context, identifies the speaker + party/ideology, classifies the mention, and aggregates results. It also adds a check for conspiracy mentality throughout the intervention.

In [ ]:

```
# 9. Generate CSV
detected_mentions = process_documents(data, regex_patterns)
output_folder = r"D:\TRABAJO\RESULTADOS"
os.makedirs(output_folder, exist_ok=True)
output_file = os.path.join(output_folder, "conspiraciones.csv")
columnas = ["ID", "Fecha", "Legislatura", "Documento", "Numero de Sesion", "Orador", "Partido", "Ideologia",
"Grupo", "Cita", "Contexto", "Clasificacion", "Si/No Conspiracion", "Explicita/Implicita"]
df_resultados = pd.DataFrame(detected_mentions, columns=columnas)
df_resultados.to_csv(output_file, sep=';', quoting=csv.QUOTE_ALL, index=False, encoding='utf-8')
print(f" Archivo guardado: {output_file}")
```

Convert the results into a DataFrame and export them to conspiraciones.csv (separator ;, quotation marks, UTF-8).

In [ ]:

```
# 10. Show results
df = pd.read_csv(output_file, delimiter=';')
print(df.head())
```

Load the generated CSV and display the first few rows for quick verification.

SECOND SPANISH REPUBLIC DATABASE

In [ ]:

```
# 1. Importing libraries
import os
import re
import csv
import pandas as pd
from datetime import datetime
import unicodedata

# To read PDFs
from pdfminer.high_level import extract_text
```

Libraries for managing folders/files (*os*, *csv*), searching with regex (*re*), tabulating data (*pandas*), dates (*datetime*), removing accents (*unicodedata*) and extracting text from PDFs (*pdfminer.six*).

In [ ]:

```
# Load base of Members of Parliament (MPs) with politic party and ideology
diputados_df = pd.read_excel(r"D:\TRABAJO\DIPUTADOS\diputados_2REP.xlsx")

def quitar_tildes(texto):
    return ''.join(
```

```
        c for c in unicodedata.normalize('NFD', str(texto))
        if unicodedata.category(c) != 'Mn'
    )
```

Load the Excel file with MPs and define a text normaliser that removes accents to compare names reliably.

In [ ]:

```
## Normalise column names to avoid accents and capital letters
def _norm_col(s):
    return quitar_tildes(str(s)).strip().upper()

colmap = {_norm_col(c): c for c in diputados_df.columns}

def _find_col(*contains_candidates):
    for norm_name, orig in colmap.items():
        for frag in contains_candidates:
            if frag in norm_name:
                return orig
    return None

## Locate columns even if they contain accents or variations.
nombre_col    = _find_col("NOMBRE")
partido_col   = _find_col("PARTIDO")
ideologia_col = _find_col("IDEOLOGIA", "IDEOLOGÍA", "IDEOLO")  # IDEOLOGÍA/IDEOLOGIA/IDEOLO...

if not nombre_col:
    raise ValueError(f"No encuentro la columna de nombres. Columnas disponibles:
{list(diputados_df.columns)}")

## Standardised key by row and we build a dictionary {NOMBRE_NORMALIZADO: (partido, ideología)}
diputados_df["_NOMBRE_NORMALIZADO"] = (
    diputados_df[nombre_col]
    .astype(str)
    .map(lambda x: quitar_tildes(x).upper().strip())
)

_mapa_diputados = {}
for _, row in diputados_df.iterrows():
    k = row["_NOMBRE_NORMALIZADO"]
    partido = str(row[partido_col]) if partido_col and (partido_col in row) and pd.notna(row[partido_col])
else "No identificado"
    ideol   = str(row[ideologia_col]) if ideologia_col and (ideologia_col in row) and
pd.notna(row[ideologia_col]) else "No especificada"
    _mapa_diputados[k] = (partido, ideol)

def buscar_partido_ideologia(nombre):
    if not isinstance(nombre, str) or not nombre.strip():
        return "No identificado", "No especificada"
    clave = quitar_tildes(nombre).upper().strip()
    return _mapa_diputados.get(clave, ("No identificado", "No especificada"))
```

It performs robust column mapping even if capital letters/accents change, validates that the name column exists, and creates a quick search dictionary by normalised name → (politic party, ideology) and a function to consult it.

In [ ]:

```
# 2. Folders with PDFs to process
pdf_dirs = [
    r"D:\TRABAJO\DIARIOS\31-77\2REP\36",
    r"D:\TRABAJO\DIARIOS\31-77\2REP\33-35",
    r"D:\TRABAJO\DIARIOS\31-77\2REP\31-33",
]
```

In [ ]:

```
List of routes where the PDFs are located
```

In [ ]:

```
# 3. Functions to read PDFs
## Extract text
def extract_text_from_pdf(file_path):
    try:
        return extract_text(file_path)
    except Exception as e:
```

```
        print(f" No se pudo extraer texto de: {file_path} ({e})")
        return ""

## Date patterns (text and numeric in file name)
meses = {
    "enero": "01", "febrero": "02", "marzo": "03", "abril": "04", "mayo": "05",
    "junio": "06", "julio": "07", "agosto": "08", "septiembre": "09", "setiembre": "09",
    "octubre": "10", "noviembre": "11", "diciembre": "12"
}
fecha_texto_pattern = re.compile(r"([A-Za-zÁÉÍÓÚáéíóúñÑ]+)\s+(\d{1,2})\s+de\s+([A-Za-
zÁÉÍÓÚáéíóúñÑ]+)\s+de\s+(\d{4})")

## Possible numerical dates in file names: 20240131, 31-01-2024, 31_01_2024, 31.01.2024, 2024-01-31, etc.
fecha_num_patterns = [
    re.compile(r"(?<!\d)(\d{4})(\d{2})(\d{2})(?!\d)"),            # yyyymmdd
    re.compile(r"(?<!\d)(\d{2})[-_.](\d{2})[-_.](\d{4})(?!\d)"),  # dd-mm-yyyy
    re.compile(r"(?<!\d)(\d{4})[-_.](\d{2})[-_.](\d{2})(?!\d)"),  # yyyy-mm-dd
]

def normaliza_ddmmyyyy(d, m, y):
    try:
        return datetime(int(y), int(m), int(d)).strftime("%d/%m/%Y")
    except Exception:
        return "Desconocida"

def intenta_fecha_desde_texto(txt):
    m = fecha_texto_pattern.search(txt)
    if not m:
        return None

    dia = m.group(2).zfill(2)
    mes_texto = quitar_tildes(m.group(3).lower())
    mes_num = meses.get(mes_texto)
    anio = m.group(4)
    if mes_num:
        return normaliza_ddmmyyyy(dia, mes_num, anio)
    return None

def intenta_fecha_desde_nombre(nombre):
    base = os.path.basename(nombre)
    for pat in fecha_num_patterns:
        m = pat.search(base)
        if m:
            if pat is fecha_num_patterns[0]:  # yyyymmdd
                y, mm, d = m.group(1), m.group(2), m.group(3)
                return normaliza_ddmmyyyy(d, mm, y)
            elif pat is fecha_num_patterns[1]:  # dd-mm-yyyy
                d, mm, y = m.group(1), m.group(2), m.group(3)
                return normaliza_ddmmyyyy(d, mm, y)
            else:  # yyyy-mm-dd
                y, mm, d = m.group(1), m.group(2), m.group(3)
                return normaliza_ddmmyyyy(d, mm, y)
    return None

def get_legislatura_from_path(path_str):
    try:
        last_dir = os.path.basename(os.path.dirname(path_str))
        if not last_dir:
            last_dir = os.path.basename(os.path.dirname(os.path.dirname(path_str)))
        return last_dir if last_dir else "Desconocida"
    except Exception:
        return "Desconocida"

def format_fecha(fecha):
    # Try dd/mm/yyyy
    try:
        parsed = datetime.strptime(fecha, "%d/%m/%Y")
        return parsed.strftime("%d/%m/%Y")
    except Exception:
        pass
    # Try yyyymmdd
    try:
        parsed_date = datetime.strptime(fecha, "%Y%m%d")
        return parsed_date.strftime("%d/%m/%Y")
    except Exception:
        return fecha
```

Functions to extract text from PDFs, detect dates both in the text and in the file name, normalise dates from different formats, and obtain the legislative period from the PDF path.

In [ ]:

```
# 4. Reeding PDFs from folders
def read_pdfs_from_dirs(dirs):
    all_data = []
    print(" Inicio de la lectura de archivos PDF...")
    for dir_path in dirs:
        if not os.path.isdir(dir_path):
            print(f" Carpeta no encontrada: {dir_path}")
            continue
        for root, _, files in os.walk(dir_path):
            for file in files:
                if not file.lower().endswith(".pdf"):
                    continue
                file_path = os.path.join(root, file)
                legislatura = get_legislatura_from_path(file_path)
                print(f" Procesando PDF: {file_path} (Legislatura {legislatura})")

                texto = extract_text_from_pdf(file_path)

                # Date: first try from text; if not, from the file name
                fecha = intenta_fecha_desde_texto(texto) or intenta_fecha_desde_nombre(file_path) or
"Desconocida"
                fecha = format_fecha(fecha)

                # ID: use file name whitout extension
                doc_id = os.path.splitext(os.path.basename(file_path))[0]

                # Document/CVE: if there is a BOCG/DS key in the text itself, use it; if not, use the base
name
                documento_match = re.search(r"\b(?:BOCG[_-]|DS(?:CD|CG)?[_-])[A-Z0-9_-]+", texto,
re.IGNORECASE)
                cve = documento_match.group() if documento_match else doc_id

                all_data.append({
                    "texto": texto,
                    "fecha": fecha,
                    "legislatura": legislatura,
                    "id": doc_id,
                    "cve": cve
                })
    print(f" Lectura de archivos completada. Total de documentos: {len(all_data)}")
    return all_data

data = read_pdfs_from_dirs(pdf_dirs)
```

Browse through the folders, extract text from each PDF, deduce date, legislature, ID, and CVE.

In [ ]:

```
# 5. List of patterns of conspiracy theories
GUION_GUARD = r"(?<![-\u2010-\u2015])"  # -, ‐, ‑, ‒, –

conspiracy_patterns = [
    rf"{GUION_GUARD}\bjud[ií](?:o|a|os|as)\b",          # Judío/a/os/as
    rf"{GUION_GUARD}\bjuda[ií](?:smo|co)\b",            # Judaísmo / Judaico
    rf"{GUION_GUARD}\bcontubernio(?:s)?\b",             # Contubernio(s)
    rf"{GUION_GUARD}\b(?:complot|compl[oó])\b",         # Complot / Compló
    rf"{GUION_GUARD}\bconjura(?:s)?\b",                 # Conjura(s)
    rf"{GUION_GUARD}\bconspiraci[oó]n(?:es)?\b",        # Conspiración(es)
    rf"{GUION_GUARD}\bmas[oó]n(?:es)?\b",               # Masón / Masones
    rf"{GUION_GUARD}\bmasoner[ií]a\b",                  # Masonería
    rf"{GUION_GUARD}\bmas[oó]nic[oa]s?\b",              # Masónico/a/os/as
    rf"{GUION_GUARD}\bsanedr[ií]n\b",                   # Sanedrín
    rf"{GUION_GUARD}\bsi[oó]n\b",                       # Sión
    rf"{GUION_GUARD}\brabino(?:s)?\b",                  # Rabino(s)
]
regex_patterns = [re.compile(p, re.IGNORECASE | re.UNICODE) for p in conspiracy_patterns]

## Additional filter for hyphen with line break or spaces
HYPHENS = "-\u2010\u2011\u2012\u2013\u2014\u2015"
def preceded_by_hyphen(text, idx):
```

```
    j = idx - 1
    while j >= 0 and text[j].isspace():
        j -= 1
    return j >= 0 and text[j] in HYPHENS
```

Define search patterns and avoid errors due to line breaks.

In [ ]:

```
# 6. Auxiliary regular expressions
fecha_pattern = re.compile(r"[a-zA-ZÁÉÍÓÚáéíóúñÑ]+ (\d{1,2}) de ([a-zA-ZÁÉÍÓÚáéíóúñÑ]+) de (\d{4})")
```

Regex to identify dates written in different ways

In [ ]:

```
# 7. Auxiliary functions
def extract_presidente(text):
    patron_presidente = r"PRESIDENCIA DEL(?: EXCMO\.| EXCMA\.)?(?: SR\.| SRA\.)?(?: D\.| DÑA\.)?(?: DOÑA|
DON)?\s+([A-ZÁÉÍÓÚÑÜ\s]+)"
    match = re.search(patron_presidente, text.upper())
    if match:
        nombre = match.group(1).strip()
        nombre = re.sub(r"\s+", " ", nombre)
        return nombre.title()
    return None

def extract_orador(texto_antes_mencion, presidente=None):
    posibles = re.findall(r"([A-ZÁÉÍÓÚÑÜ\s]{2,}):", texto_antes_mencion)
    candidatos = []
    for p in posibles:
        palabras = p.strip().split()
        palabras_validas = [w for w in palabras if len(w) > 1]
        if len(palabras_validas) >= 2:
            candidato = " ".join(palabras_validas).title()
            candidatos.append(candidato)
    if candidatos:
        return candidatos[-1]
    elif presidente:
        return presidente
    else:
        return "Desconocido"

# Context before and after, by 'words', keeping the original text
def extract_context(text_original, match, pre_words=40, post_words=40):
    start_pos = match.start()
    end_pos = match.end()

    def take_last_tokens(s, n):
        tokens = re.findall(r"\S+\s*", s, flags=re.UNICODE)
        return "".join(tokens[-n:])

    def take_first_tokens(s, n):
        tokens = re.findall(r"\S+\s*", s, flags=re.UNICODE)
        return "".join(tokens[:n])

    before = text_original[:start_pos]
    middle = text_original[start_pos:end_pos]
    after  = text_original[end_pos:]

    left  = take_last_tokens(before, pre_words)
    right = take_first_tokens(after, post_words)

    left_trimmed  = left.lstrip()
    right_trimmed = right.rstrip()

    left_prefix  = "…" if len(before) > len(left) else ""
    right_suffix = "…" if len(after)  > len(right) else ""

    highlighted = f"**{middle}**"
    contexto = f"{left_prefix}{left_trimmed}{highlighted}{right_trimmed}{right_suffix}"
    return contexto.strip()

def tiene_mentalidad_conspirativa(texto):
    texto = texto.lower()
```

```
    patrones_mentales = [
        r"ocultan?", r"manipulan?", r"esconden?", r"controlan?", r"engañan?",
        r"mentira(s)?", r"engaño(s)?", r"verdad escondida", r"no se dice todo",
        r"intereses oscuros", r"poder oculto", r"información oculta", r"no nos cuentan",
        r"sospechoso(s)?", r"falta de transparencia", r"intereses que no se revelan",
        r"engañosa(s)?", r"engaños?", r"manipulación", r"manipulado(s)?", r"secretismo"
    ]
    for patron in patrones_mentales:
        if re.search(patron, texto):
            return True
    return False

def classify_mention(context):
    context_lower = context.lower()

    # 1) Explicit refutation
    refutacion_patrones = [

r"\b(bulo(s)?|mentira(s)?|falsedad(es)?|desinformación|es\s+falso|es\s+mentira|no\s+es\s+verdad|ha\s+sido\s+desmentid[oa]|carece\s+de\s+fundamento|infundad[oa]|no\s+tiene\s+base|ha\s+sido\s+refutad[oa]|teor[ií]a\s+sin\s+base|conspiraci[oó]n\s+infundada)\b",

r"\b(no\s+hay\s+(pruebas|evidencia|fundamento)|nunca\s+se\s+demostr[oó]|no\s+es\s+cierto|no\s+es\s+real|es\s+una\s+invenci[oó]n)\b"
    ]
    if any(re.search(p, context_lower) for p in refutacion_patrones):
        return "C"

    # 2) Suggestive language
    if tiene_mentalidad_conspirativa(context):
        return "B"

    # 3) Direct mention
    mencion_teoria_patrones = [
        rf"{GUION_GUARD}\bjud[ií](?:o|a|os|as)\b",
        rf"{GUION_GUARD}\bjuda[ií](?:smo|co)\b",
        rf"{GUION_GUARD}\bcontubernio(?:s)?\b",
        rf"{GUION_GUARD}\b(?:complot|compl[oó])\b",
        rf"{GUION_GUARD}\bconjura(?:s)?\b",
        rf"{GUION_GUARD}\bconspiraci[oó]n(?:es)?\b",
        rf"{GUION_GUARD}\bmas[oó]n(?:es)?\b",
        rf"{GUION_GUARD}\bmasoner[ií]a\b",
        rf"{GUION_GUARD}\bmas[oó]nic[oa]s?\b",
        rf"{GUION_GUARD}\bsanedr[ií]n\b",
        rf"{GUION_GUARD}\bsi[oó]n\b",
        rf"{GUION_GUARD}\brabino(?:s)?\b"
    ]
    if any(re.search(p, context_lower) for p in mencion_teoria_patrones):
        return "A"

    return "N"


def determinar_expresion(contexto, clasificacion):
    ctx = contexto.lower()

    marcadores_conspirativos = [
        rf"{GUION_GUARD}\bcontubernio(?:s)?\b",
        rf"{GUION_GUARD}\b(?:complot|compl[oó])\b",
        rf"{GUION_GUARD}\bconjura(?:s)?\b",
        rf"{GUION_GUARD}\bconspiraci[oó]n(?:es)?\b",
        rf"{GUION_GUARD}\bmasoner[ií]a\b",
        rf"{GUION_GUARD}\bmas[oó]n(?:es)?\b",
        rf"{GUION_GUARD}\bmas[oó]nic[oa]s?\b",
        rf"{GUION_GUARD}\bsanedr[ií]n\b",
        rf"{GUION_GUARD}\bsi[oó]n\b",
    ]
    marcadores_identitarios = [
        rf"{GUION_GUARD}\bjud[ií](?:o|a|os|as)\b",
        rf"{GUION_GUARD}\bjuda[ií](?:smo|co)\b",
        rf"{GUION_GUARD}\brabino(?:s)?\b",
    ]
    patrones_implicitos = [
        r"\bnos\s+mienten\b", r"\bno\s+nos\s+cuentan\b", r"\bintereses\s+ocultos\b",
        r"\bfalta\s+de\s+transparencia\b", r"\bno\s+es\s+toda\s+la\s+verdad\b",
        r"\bmanipulaci[oó]n\b", r"\bengaño\b", r"\bsecretismo\b",
        r"\binformaci[oó]n\s+que\s+no\s+se\s+dice\b", r"\bnos\s+han\s+ocultado\b",
```

```
        r"\bsospechamos\s+que\b", r"\bparece\s+que\s+hay\s+algo\s+detr[aá]s\b",
        r"\bdecisiones\s+que\s+no\s+se\s+explican\b", r"\bverdad\s+escondida\b",
        r"\bcontrolan\s+la\s+narrativa\b", r"\bpoder\s+en\s+la\s+sombra\b"
    ]

    if clasificacion == "C":
        if any(re.search(p, ctx) for p in marcadores_conspirativos + marcadores_identitarios):
            return "Explícita"
        else:
            return "Implícita"

    if any(re.search(p, ctx) for p in marcadores_conspirativos):
        return "Explícita"

    if (any(re.search(p, ctx) for p in marcadores_identitarios)
        and any(re.search(p, ctx) for p in patrones_implicitos)):
        return "Implícita"

    if clasificacion in ["A", "B", "C"] and any(re.search(p, ctx) for p in marcadores_identitarios):
        return "Implícita"

    if clasificacion in ["A", "B", "C"]:
        return "Implícita"

    return "No procede"
```

Support functions for:

- extract_presidente: attempts to identify who is chairing the session.
- extract_orador: detects who is speaking just before the mention.
- extract_context: returns an extract with ~N words before/after and highlights the match in bold.
- tiene_mentalidad_conspirativa: detects suggestive language of a conspiratorial nature.
- classify_mention: classifies the context as C (refutation) > B (mindset) > A (direct mention) > N (not applicable).
- determinar_expresion: determines whether the mention is Explicit, Implicit or Not Applicable according to the markers present.

In [ ]:

```
# 8. Process documents
def process_documents(data, regex_patterns):
    results = []
    print(" Procesando documentos para menciones de teorías de conspiración...")
    for idx, item in enumerate(data):
        text_original = item.get("texto", "")
        text_lower = text_original.lower()
        fecha = item.get("fecha", "Desconocida")
        legislatura = item.get("legislatura", "Desconocida")
        doc_id = item.get("id", "Sin ID")
        doc_id_str = str(doc_id)

        numero_sesion_match = re.search(r"(?:sesión\s*)?(?:n[úu]m(?:\.|ero)?)[\s:]*?(\d+)", text_lower,
re.IGNORECASE)
        numero_sesion = numero_sesion_match.group(1) if numero_sesion_match else "Desconocido"

        documento_match = re.search(r"\b(?:BOCG[_-]|DS(?:CD|CG)?[_-])[A-Z0-9_-]+", text_lower)
        documento = documento_match.group() if documento_match else item.get("cve", "Desconocido")

        fecha_formateada = format_fecha(fecha)
        presidente = extract_presidente(text_lower)

        print(f" Documento {idx + 1}/{len(data)}")
        ya_detectados = set()

        for grupo_index, pattern in enumerate(regex_patterns, start=1):
            for match in pattern.finditer(text_original):
                if preceded_by_hyphen(text_original, match.start()):
                    continue

                contexto = extract_context(text_original, match)

                texto_hasta_mencion = text_original[:match.start()]
                orador_nombre = extract_orador(texto_hasta_mencion, presidente)
                clave = f"{orador_nombre}_{doc_id_str}"
                if clave in ya_detectados:
                    continue
```

```
                ya_detectados.add(clave)
                partido, ideologia = buscar_partido_ideologia(orador_nombre)
                clasificacion = classify_mention(contexto)
                dic_conspiracion = "Sí" if clasificacion in ["A", "B", "C"] else "No"
                grupo_texto = f"Grupo {grupo_index}"
                modo_exp_implic = determinar_expresion(contexto, clasificacion) if dic_conspiracion == "Sí"
else "No procede"

                results.append([
                    doc_id_str, fecha_formateada, legislatura, documento, numero_sesion, orador_nombre,
                    partido, ideologia, grupo_texto, text_original[match.start():match.end()], contexto,
clasificacion, dic_conspiracion, modo_exp_implic
                ])

        # Second pass for conspiracy mentality in full text
        orador_general = extract_orador(text_original, presidente)
        clave_general = f"{orador_general}_{doc_id_str}"
        if clave_general not in ya_detectados:
            clasificacion_general = classify_mention(text_original)
            if clasificacion_general == "B":
                partido, ideologia = buscar_partido_ideologia(orador_general)
                results.append([
                    doc_id_str, fecha_formateada, legislatura, documento, numero_sesion, orador_general,
                    partido, ideologia, "Intervención completa", "", text_original[:1000],
clasificacion_general, "Sí", "Implícita"
                ])
                ya_detectados.add(clave_general)

    print(f" Se encontraron {len(results)} menciones de teorías de conspiración.")
    return results
```

Analyze each document, search for patterns, extract context, identify speakers, classify, and add rows of results.

In [ ]:

```
# 9. Generate CSV
detected_mentions = process_documents(data, regex_patterns)
output_folder = r"D:\TRABAJO\RESULTADOS"
os.makedirs(output_folder, exist_ok=True)
output_file = os.path.join(output_folder, "conspiraciones.csv")
columnas = ["ID", "Fecha", "Legislatura", "Documento", "Numero de Sesion", "Orador", "Partido", "Ideologia",
"Grupo", "Cita", "Contexto", "Clasificacion", "Si/No Conspiracion", "Explicita/Implicita"]
df_resultados = pd.DataFrame(detected_mentions, columns=columnas)
df_resultados.to_csv(output_file, sep=';', quoting=csv.QUOTE_ALL, index=False, encoding='utf-8')
print(f" Archivo guardado: {output_file}")
```

Convert the results to DataFrame and export them to conspiracies.csv (separator ;, quotation marks, UTF-8).

In [ ]:

```
# 10. Show results
try:
    df = pd.read_csv(output_file, delimiter=';')
    print(df.head())
except Exception as e:
    print(f" No se pudo leer el CSV generado para vista previa: {e}")
```

Display a preview of the CSV

FRANCOISM DATABASE

In [ ]:

```
# 1. Importing libraries
import os
import re
import sys
import csv
import pandas as pd
from datetime import datetime
import unicodedata
import tempfile
```

```
import subprocess
import shutil

# To read PDFs
from pdfminer.high_level import extract_text
```

Import the necessary libraries to handle files, text, dates, and PDFs, which have already been mentioned above. With *pdfminer*, text is extracted from PDFs, while *shutil* and *subprocess* allow external tools such as *ocrmypdf* to be run.

In [ ]:

```
def _lazy_import_ocr_deps():
    try:
        from pdf2image import convert_from_path  # type: ignore
        import pytesseract  # type: ignore
        return convert_from_path, pytesseract
    except Exception as e:
        raise
```

Load the OCR dependencies (*pdf2image*, *pytesseract*) only if they are needed, so as not to slow down the script.

In [ ]:

```
def _resolve_poppler_path():

    # 1) conda
    conda_prefix = os.getenv("CONDA_PREFIX")
    if conda_prefix:
        cand = os.path.join(conda_prefix, "Library", "bin")
        if os.path.isdir(cand):
            pdfinfo = os.path.join(cand, "pdfinfo.exe")
            pdftoppm = os.path.join(cand, "pdftoppm.exe")
            if os.path.exists(pdfinfo) and os.path.exists(pdftoppm):
                return {"poppler_path": cand, "has_pdfinfo": True, "has_pdftoppm": True}

    # 2) POPPLER_PATH environment variable
    env = os.getenv("POPPLER_PATH")
    if env and os.path.isdir(env):
        pdfinfo = os.path.join(env, "pdfinfo.exe")
        pdftoppm = os.path.join(env, "pdftoppm.exe")
        return {
            "poppler_path": env,
            "has_pdfinfo": os.path.exists(pdfinfo),
            "has_pdftoppm": os.path.exists(pdftoppm),
        }

    # 3) System PATH
    pdfinfo_path = shutil.which("pdfinfo")
    pdftoppm_path = shutil.which("pdftoppm")
    folder = None
    for p in (pdfinfo_path, pdftoppm_path):
        if p:
            folder = os.path.dirname(p); break
    return {
        "poppler_path": folder,
        "has_pdfinfo": bool(pdfinfo_path),
        "has_pdftoppm": bool(pdftoppm_path),
    }
```

Search for the location of *Poppler* (library required for *pdf2image*) in the conda environment, system variables, or your computer's PATH.

In [ ]:

```
## Performance/robustness settings
OCR_CACHE_DIR = r"D:\TRABAJO\OCR_CACHE"
os.makedirs(OCR_CACHE_DIR, exist_ok=True)

## Always use OCR strength (robust for old scans)
OCR_ALWAYS_FORCE = True  # True = always use ocrmypdf with "force ocr"
```

Create an OCR cache folder to avoid repeated scans and set OCR to always be applied (force-ocr), ideal for old scanned PDFs.

In [ ]:

```
# 2. Folders with PDFs
pdf_dirs = [
    r"D:\TRABAJO\DIARIOS\31-77\67-71"
]
```

List of folders containing the PDFs to be analyzed (in this case, those relating to Franco's dictatorship).

In [ ]:

```
# 3. Reading and precessing PDF
def _cached_ocr_path(in_pdf: str, force: bool) -> str:
    base = os.path.basename(in_pdf)
    name, _ = os.path.splitext(base)
    suf = "_force" if force else "_auto"
    return os.path.join(OCR_CACHE_DIR, f"{name}{suf}.pdf")
```

Generates the name of the cached OCR file based on the original PDF and whether or not OCR was forced.

In [ ]:

```
def _run_ocrmypdf(in_pdf: str, out_pdf: str, lang: str = "spa", jobs: int = 0, force: bool = False) -> None:
    if jobs <= 0:
        jobs = max(2, (os.cpu_count() or 4))

    # Detect 'unpaper'
    has_unpaper = shutil.which("unpaper") is not None

    cmd = [
        sys.executable, "-m", "ocrmypdf",
        "-l", lang,
        "--jobs", str(jobs),
        "--output-type", "pdf",
        "--rotate-pages", "--deskew",
    ]
    if has_unpaper:
        cmd.append("--clean")
    if force:
        cmd.append("--force-ocr")

    cmd += [in_pdf, out_pdf]

    try:
        subprocess.run(cmd, check=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    except subprocess.CalledProcessError as e:
        print(" ocrmypdf (intento 1) falló")
        print("   CMD:", " ".join(cmd))
        if not has_unpaper:
            print("   Nota: '--clean' no se usó porque no se encontró 'unpaper' en PATH.")
        stderr = e.stderr.decode("utf-8", errors="ignore") if e.stderr else ""
        if stderr.strip():
            print("   STDERR:\n", stderr)
        raise
```

Run ocrmypdf from Python to apply OCR, clean up, straighten pages, and enhance text in scanned PDFs.

In [ ]:

```
def _ocr_pages_with_tesseract(file_path: str, dpi: int = 400, lang: str = "spa") -> str:
    try:
        convert_from_path, pytesseract = _lazy_import_ocr_deps()
    except Exception as e:
        print(f" OCR por páginas no disponible (faltan dependencias): {e}")
        return ""
    try:
        info = _resolve_poppler_path()
        kwargs = {"dpi": dpi}
        if info["poppler_path"]:
            kwargs["poppler_path"] = info["poppler_path"]
        else:
            print(" Poppler no localizado. Instala poppler o define POPPLER_PATH.")
        if not (info["has_pdfinfo"] and info["has_pdftoppm"]):
            print(f" Aviso Poppler: pdfinfo={info['has_pdfinfo']} pdftoppm={info['has_pdftoppm']} "
                  f"({info['poppler_path'] or 'PATH no resuelto'})")
```

```
        pages = convert_from_path(file_path, **kwargs)
        texts = []
        for img in pages:
            txt = pytesseract.image_to_string(img, lang=lang)
            texts.append(txt)
        return "\n\n".join(texts)
    except Exception as e:
        print(f" OCR (pytesseract/pdf2image) falló en {file_path}: {e}")
        return ""
```

Alternative to ocrmypdf: converts each page of the PDF to an image and uses Tesseract OCR to extract text. Serves as a backup method if the main OCR fails.

In [ ]:

```
def extract_text_from_pdf(file_path):
    cached = _cached_ocr_path(file_path, force=OCR_ALWAYS_FORCE)
    if os.path.exists(cached):
        try:
            return extract_text(cached) or ""
        except Exception:
            pass

    # 1) OCR with ocrmypdf
    try:
        with tempfile.TemporaryDirectory() as td:
            tmp_out = os.path.join(td, "ocr.pdf")
            _run_ocrmypdf(file_path, tmp_out, lang="spa", jobs=0, force=OCR_ALWAYS_FORCE)
            try:
                os.replace(tmp_out, cached)  # mueve a caché
            except Exception:
                shutil.copyfile(tmp_out, cached)
        return extract_text(cached) or ""
    except Exception as e:
        print(f" OCR (ocrmypdf) falló en {file_path}: {e}")

    # 2) Fallback: pytesseract by pages
    txt = _ocr_pages_with_tesseract(file_path, dpi=400, lang="spa")
    if txt:
        return txt

    # 3) Last network: pdfminer direct
    try:
        return extract_text(file_path) or ""
    except Exception as e:
        print(f" pdfminer falló en {file_path}: {e}")
        return ""
```

General function to extract text from a PDF. Sequence: use cache → try *ocrmypdf* → try *Tesseract* → if all else fails, *pdfminer*.

In [ ]:

```
## Robust pre-processing of extracted text
LIGATURAS = {
    "ﬁ": "fi", "ﬂ": "fl", "ﬀ": "ff", "ﬃ": "ffi", "ﬄ": "ffl",
    "“": '"', "”": '"', "’": "'", "‘": "'", "–": "-", "—": "-"
}

CABECERAS_PIES_RE = re.compile(
    r"(?:^\s*DIARIO\s+DE\s+SESIONES.*?$)|"
    r"(?:^\s*CONGRESO\s+DE\s+LOS\s+DIPUTADOS.*?$)|"
    r"(?:^\s*SENADO.*?$)|"
    r"(?:^\s*SESI[ÓO]N.*?$)|"
    r"(?:^\s*SUMARIO.*?$)|"
    r"(?:^\s*P[ÁA]GINA\s+\d+.*?$)",
    flags=re.MULTILINE
)

def fix_ligaduras_y_espacios(t: str) -> str:
    # 1) Replaces ligatures and soft hyphen
    for k, v in LIGATURAS.items():
        t = t.replace(k, v)
    t = t.replace("\u00AD", "")  # soft hyphen

    # 2) Remove lines consisting ONLY of dots/spaces
```

```
    t = re.sub(r"(?m)^[\s\.\·]{2,}$", "", t)

    # 3) Synthesize groups of dots into "..." or remove them if there are only dots in the row.
    t = re.sub(r"(?:\.\s*){3,}", "…", t)
    t = re.sub(r"(?m)^\s*[…\.]+\s*$", "", t)

    # 4) Join words separated by hyphen + line break (all variants)
    t = re.sub(r"(\w{2,})\s*[-\u2010-\u2015]\s*\r?\n\s*(\w{2,})", r"\1\2", t)

    # 5) Quita guiones de salto duro "- \n"
    t = re.sub(r"(\w{2,})\s*-\s*\r?\n\s*(\w{2,})", r"\1\2", t)

    # 6) Join "soft" breaks: end of line without punctuation + lowercase letter
    t = re.sub(r"([A-Za-zÁÉÍÓÚÜÑáéíóúüñ])\r?\n([a-záéíóúüñ])", r"\1 \2", t)

    # 7) Typical OCR case: "comi sion" o "comi-   sion" => "comision"
    t = re.sub(r"(?i)\bcomi(?:\s|[-\u2010-\u2015]|\u00AD){1,3}si[oó]n\b", "comision", t)

    # 8) Normalize spaces and line breaks
    t = re.sub(r"[ \t]+", " ", t)
    t = re.sub(r"\n{3,}", "\n\n", t)
    return t

def limpiar_cabeceras_pies(t: str) -> str:
    return CABECERAS_PIES_RE.sub("", t, count=0)

def clean_context_noise(s: str) -> str:
    # remove lines consisting only of dots/… that have survived
    s = re.sub(r"(?m)^[\s\.\·…]{2,}$", "", s)

    # collapse sequences such as ". . ."
    s = re.sub(r"(?:\.\s*){3,}", "…", s)
    s = re.sub(r"…{2,}", "…", s)
    s = re.sub(r"[ \t]{2,}", " ", s)
    s = re.sub(r"\n{3,}", "\n\n", s)

    # fix cuts "sión \n de" -> "sión de"
    s = re.sub(r"([A-Za-zÁÉÍÓÚÜÑáéíóúüñ])\s*\n\s+(de|del|la|las|los|y)\b", r"\1 \2", s, flags=re.IGNORECASE)
    return s.strip()

def looks_like_garbage(s: str, min_letters_ratio: float = 0.35) -> bool:
    s2 = re.sub(r"\s+", "", s)
    if not s2:
        return True
    letters = sum(ch.isalpha() for ch in s2)
    return letters < min_letters_ratio * len(s2)
```

Functions to clean up OCR text: remove ligatures, hyphens, and line breaks; delete headers/footers; and discard noisy or wordless fragments.

In [ ]:

```
def quitar_tildes(texto):
    return ''.join(
        c for c in unicodedata.normalize('NFD', str(texto))
        if unicodedata.category(c) != 'Mn'
    )

meses = {
    "enero": "01", "febrero": "02", "marzo": "03", "abril": "04", "mayo": "05",
    "junio": "06", "julio": "07", "agosto": "08", "septiembre": "09", "setiembre": "09",
    "octubre": "10", "noviembre": "11", "diciembre": "12"
}

fecha_texto_pattern = re.compile(
    r"([A-Za-zÁÉÍÓÚáéíóúñÑ]+)\s+(\d{1,2})\s+de\s+([A-Za-zÁÉÍÓÚáéíóúñÑ]+)\s+de\s+(\d{4})"
)

fecha_num_patterns = [
    re.compile(r"(?<!\d)(\d{4})(\d{2})(\d{2})(?!\d)"),
    re.compile(r"(?<!\d)(\d{2})[-_.](\d{2})[-_.](\d{4})(?!\d)"),
    re.compile(r"(?<!\d)(\d{4})[-_.](\d{2})[-_.](\d{2})(?!\d)"),
]

def normaliza_ddmmyyyy(d, m, y):
    try:
```

```
        return datetime(int(y), int(m), int(d)).strftime("%d/%m/%Y")
    except Exception:
        return "Desconocida"

def intenta_fecha_desde_texto(txt):
    m = fecha_texto_pattern.search(txt)
    if not m:
        return None
    dia = m.group(2).zfill(2)
    mes_texto = quitar_tildes(m.group(3).lower())
    mes_num = meses.get(mes_texto)
    anio = m.group(4)
    if mes_num:
        return normaliza_ddmmyyyy(dia, mes_num, anio)
    return None

def intenta_fecha_desde_nombre(nombre):
    base = os.path.basename(nombre)
    for i, pat in enumerate(fecha_num_patterns):
        m = pat.search(base)
        if not m:
            continue
        if i == 0:
            y, mm, d = m.group(1), m.group(2), m.group(3)
            return normaliza_ddmmyyyy(d, mm, y)
        elif i == 1:
            d, mm, y = m.group(1), m.group(2), m.group(3)
            return normaliza_ddmmyyyy(d, mm, y)
        else:
            y, mm, d = m.group(1), m.group(2), m.group(3)
            return normaliza_ddmmyyyy(d, mm, y)
    return None

def get_legislatura_from_path(path_str):
    try:
        last_dir = os.path.basename(os.path.dirname(path_str))
        if not last_dir:
            last_dir = os.path.basename(os.path.dirname(os.path.dirname(path_str)))
        return last_dir if last_dir else "Desconocida"
    except Exception:
        return "Desconocida"

def format_fecha(fecha):
    try:
        parsed = datetime.strptime(fecha, "%d/%m/%Y")
        return parsed.strftime("%d/%m/%Y")
    except Exception:
        pass
    try:
        parsed_date = datetime.strptime(fecha, "%Y%m%d")
        return parsed_date.strftime("%d/%m/%Y")
    except Exception:
        return fecha
```

These functions normalise text and dates to compare similarities.

In [ ]:

```
SPEAKER_HEADER_RE = re.compile(
    r"""(?imx)^[ \t]*(?:EL|LA)?[ \t]*
    (
        (?:S[ \t]*R\.?)
        |
        (?:S[ \t]*R[ \t]*A\.?)
        |
        (?:SE[ \t]*Ñ?[ \t]*O[ \t]*R\.?)
        |
        (?:SE[ \t]*Ñ?[ \t]*O[ \t]*R[ \t]*A\.?)
    )
    [ \t]+
    ([A-ZÁÉÍÓÚÜÑ][A-ZÁÉÍÓÚÜÑ' \-]{1,80})
    [ \t]*:\s*$"""
)

EPOCA_TITULARES = re.compile(
    r"(?im)^[ \t]*(?:SU[ \t]+EXCELENCIA|S\.?[ \t]*E\.?|PRESIDENCIA|EL[ \t]+PRESIDENTE|LA[ \t]+PRESIDENCIA|"
```

```
    r"EL[ \t]+SEÑOR[ \t]+PRESIDENTE|JEFE[ \t]+DEL[ \t]+ESTADO|EXCMO\.?|EXCMA\.?)[A-ZÁÉÍÓÚÜÑ' \t\.]*:\s*$"
)

def _normalize_sr_label(s: str) -> str:
    s_clean = quitar_tildes(re.sub(r"\s+", "", s)).lower().rstrip(".")
    if s_clean in ("sr", "s"):
        return "Sr."
    if s_clean in ("sra", "srta"):
        return "Sra."
    if s_clean == "senor":
        return "Señor"
    if s_clean == "senora":
        return "Señora"
    return s.strip().title()


def _find_prev_speaker_header(texto: str, pos_ini: int, max_chars_back=20000, max_lines_back=120):
    floor = max(0, pos_ini - max_chars_back)
    line_break = pos_ini
    lines_seen = 0

    while lines_seen < max_lines_back:
        lb = texto.rfind("\n", floor, line_break)
        if lb == -1:
            lb = floor
        start, end = (floor if lb == floor else lb + 1), line_break
        raw = texto[start:end].rstrip("\r\n")
        stripped = raw.strip()

        if stripped in (":", ":"):
            prev_lb = texto.rfind("\n", floor, start - 1)
            prev_start = (floor if prev_lb == -1 else prev_lb + 1)
            prev_line = texto[prev_start:start]

            m = SPEAKER_HEADER_RE.search(prev_line)
            if m:
                label_sr = _normalize_sr_label(m.group(1))
                apes = re.sub(r"\s{2,}", " ", m.group(2).strip())
                return f"{label_sr} {apes}", end, True

            if EPOCA_TITULARES.search(prev_line):
                label = re.sub(r"\s{2,}", " ", prev_line.strip())
                label = quitar_tildes(label).upper()
                if label.endswith(":"):
                    label = label[:-1]
                return label, end, True

        m = SPEAKER_HEADER_RE.search(raw)
        if m:
            label_sr = _normalize_sr_label(m.group(1))
            apes = re.sub(r"\s{2,}", " ", m.group(2).strip())
            return f"{label_sr} {apes}", end, True

        if EPOCA_TITULARES.search(raw):
            label = re.sub(r"\s{2,}", " ", stripped)
            label = quitar_tildes(label).upper()
            if label.endswith(":"):
                label = label[:-1]
            return label, end, True

        lines_seen += 1
        if lb == floor:
            break
        line_break = lb

    para_sep = texto.rfind("\n\n", floor, pos_ini)
    start_fallback = (para_sep + 2) if para_sep != -1 else max(0, pos_ini - 1200)
    return "DESCONOCIDO", start_fallback, False
```

Detect who is speaking (speaker) and search for headings such as “Sr. Pérez:” o “El Presidente:“, and normalise the labels “Sr.“, “Sra.“, etc.

In [ ]:

```
# 4. Search patterns
GUION_GUARD = r"(?<![-\u2010-\u2015])"  # -, -, –, -, —
```

```
CONSP_BASE = [
    rf"{GUION_GUARD}\bjud[ií](?:o|a|os|as)\b",
    rf"{GUION_GUARD}\bjuda[ií](?:smo|co)\b",
    rf"{GUION_GUARD}\bcontubernio(?:s)?\b",
    rf"{GUION_GUARD}\b(?:complot|compl[oó])\b",
    rf"{GUION_GUARD}\bconjura(?:s)?\b",
    rf"{GUION_GUARD}\bconspiraci[oó]n(?:es)?\b",
    rf"{GUION_GUARD}\bmas[oó]n(?:es)?\b",
    rf"{GUION_GUARD}\bmasoner[ií]a\b",
    rf"{GUION_GUARD}\bmas[oó]nic[oa]s?\b",
    rf"{GUION_GUARD}\bsanedr[ií]n\b",
    rf"{GUION_GUARD}\bsi[oó]n\b",
    rf"{GUION_GUARD}\brabino(?:s)?\b",
]
CONSP_IDENTITARIOS = [
    rf"{GUION_GUARD}\bjud[ií](?:o|a|os|as)\b",
    rf"{GUION_GUARD}\bjuda[ií](?:smo|co)\b",
    rf"{GUION_GUARD}\brabino(?:s)?\b",
]
CONSP_MARCADORES = [
    rf"{GUION_GUARD}\bcontubernio(?:s)?\b",
    rf"{GUION_GUARD}\b(?:complot|compl[oó])\b",
    rf"{GUION_GUARD}\bconjura(?:s)?\b",
    rf"{GUION_GUARD}\bconspiraci[oó]n(?:es)?\b",
    rf"{GUION_GUARD}\bmasoner[ií]a\b",
    rf"{GUION_GUARD}\bmas[oó]n(?:es)?\b",
    rf"{GUION_GUARD}\bmas[oó]nic[oa]s?\b",
    rf"{GUION_GUARD}\bsanedr[ií]n\b",
    rf"{GUION_GUARD}\bsi[oó]n\b",
]
```

Define word patterns related to conspiracy theories about Francoism (judíos, masones, conjura, etc.)

In [ ]:

```
REGEX_CONSP_BASE = [re.compile(p, re.IGNORECASE | re.UNICODE) for p in CONSP_BASE]
REGEX_IDENTITARIOS = [re.compile(p, re.IGNORECASE | re.UNICODE) for p in CONSP_IDENTITARIOS]
REGEX_MARCADORES = [re.compile(p, re.IGNORECASE | re.UNICODE) for p in CONSP_MARCADORES]

regex_patterns = REGEX_CONSP_BASE

HYPHENS = "-\u2010\u2011\u2012\u2013\u2014\u2015"

def preceded_by_hyphen(text, idx):
    j = idx - 1
    while j >= 0 and text[j].isspace():
        j -= 1
    return j >= 0 and text[j] in HYPHENS

COMMON_SION_WORDS = {
    "comision","sesion","expresion","presion","mision","discusion","invasion","vision",
    "cohesion","concesion","decision","ocasion","prevision","precision","profesion","posesion",
    "expansion","explosion","conclusion","confusion","ilusion","dimision","admision","transmision",
    "omision","agresion","depresion","opresion","supresion","suspension","impresion","version",
    "perversion","conversion","inversion","diversion","aversion","excursion","provision","revision",
    "incursion","intrusion","erosion","corrosion","fision","fusion","adhesion","cesion"
}

def looks_like_broken_spanish_sion(text, start, end, max_back=20):
    token = quitar_tildes(text[start:end]).lower()
    if token != "sion":
        return False
    window = text[max(0, start - max_back):start]
    tail_letters = re.sub(r"[^A-Za-zÁÉÍÓÚÜÑáéíóúüñ]", "", quitar_tildes(window)).lower()
    if not tail_letters:
        return False
    for k in range(2, min(10, len(tail_letters)) + 1):
        candidate = tail_letters[-k:] + "sion"
        if candidate in COMMON_SION_WORDS:
            return True
    return False

def is_all_uppercase_word(text):
    txt = quitar_tildes(text)
    txt = re.sub(r"[^A-Za-zÁÉÍÓÚÜÑ]", "", txt)
    return len(txt) > 1 and txt.isupper()
```

Compile all the above patterns into regular expressions ready for searching in texts and establish auxiliary functions that avoid broken words (‘comi-sion’).

In [ ]:

```
# 5. Clasification
REFUTACION_RE = re.compile(

r"\b(bulo(s)?|mentira(s)?|falsedad(es)?|desinformación|es\s+falso|es\s+mentira|no\s+es\s+verdad|ha\s+sido\s+desmentid[oa]|carece\s+de\s+fundamento|infundad[oa]|no\s+tiene\s+base|ha\s+sido\s+refutad[oa]|teor[ií]a\s+sin\s+base|conspiraci[oó]n\s+infundada)\b"
    r"|"

r"\b(no\s+hay\s+(pruebas|evidencia|fundamento)|nunca\s+se\s+demostr[oó]|no\s+es\s+cierto|no\s+es\s+real|es\s+una\s+invenci[oó]n)\b",
    re.IGNORECASE
)

def tiene_mentalidad_conspirativa(texto):
    texto = texto.lower()
    patrones_mentales = [
        r"ocultan?", r"manipulan?", r"esconden?", r"controlan?", r"engañan?",
        r"mentira(s)?", r"engaño(s)?", r"verdad escondida", r"no se dice todo",
        r"intereses oscuros", r"poder oculto", r"información oculta", r"no nos cuentan",
        r"sospechoso(s)?", r"falta de transparencia", r"intereses que no se revelan",
        r"engañosa(s)?", r"engaños?", r"manipulación", r"manipulado(s)?", r"secretismo"
    ]
    for patron in patrones_mentales:
        if re.search(patron, texto):
            return True
    return False

def classify_mention(context):
    context_lower = context.lower()
    if REFUTACION_RE.search(context_lower):
        return "C"
    if any(p.search(context_lower) for p in REGEX_CONSP_BASE):
        return "A"
    if tiene_mentalidad_conspirativa(context_lower):
        return "B"
    return "N"


def determinar_expresion(contexto, clasificacion):
    ctx = contexto.lower()
    patrones_implicitos = [
        r"\bnos\s+mienten\b", r"\bno\s+nos\s+cuentan\b", r"\bintereses\s+ocultos\b",
        r"\bfalta\s+de\s+transparencia\b", r"\bno\s+es\s+toda\s+la\s+verdad\b",
        r"\bmanipulaci[oó]n\b", r"\bengaño\b", r"\bsecretismo\b",
        r"\binformaci[oó]n\s+que\s+no\s+se\s+dice\b", r"\bnos\s+han\s+ocultado\b",
        r"\bsospechamos\s+que\b", r"\bparece\s+que\s+hay\s+algo\s+detr[aá]s\b",
        r"\bdecisiones\s+que\s+no\s+se\s+explican\b", r"\bverdad\s+escondida\b",
        r"\bcontrolan\s+la\s+narrativa\b", r"\bpoder\s+en\s+la\s+sombras?\b"
    ]

    if clasificacion == "C":
        if any(p.search(ctx) for p in REGEX_MARCADORES + REGEX_IDENTITARIOS):
            return "Explícita"
        else:
            return "Implícita"
    if any(p.search(ctx) for p in REGEX_MARCADORES):
        return "Explícita"
    if (any(p.search(ctx) for p in REGEX_IDENTITARIOS)
        and any(re.search(p, ctx) for p in patrones_implicitos)):
        return "Implícita"
    if clasificacion in ["A", "B", "C"] and any(p.search(ctx) for p in REGEX_IDENTITARIOS):
        return "Implícita"
    if clasificacion in ["A", "B", "C"]:
        return "Implícita"
    return "No procede"
```

Classify the mentions detected:

- A: direct mention
- B: suggestive or conspiratorial language

- C: refutation
- N: not applicable

Also define whether the expression was explicit or implicit.

In [ ]:

```
# 6. Extract context

def _is_letter(ch: str) -> bool:
    return quitar_tildes(ch).isalpha()


def _adjust_to_word_start(texto: str, idx: int) -> int:
    n = len(texto)
    if idx <= 0 or idx >= n:
        return max(0, min(idx, n))
    i = idx
    if _is_letter(texto[i-1]) and _is_letter(texto[i]):
        while i > 0 and _is_letter(texto[i-1]):
            i -= 1
    while i < n and not _is_letter(texto[i]) and texto[i] != "\n":
        i += 1
    return i


def _token_spans(texto: str, start: int, end: int):
    for m in re.finditer(r"[A-Za-zÁÉÍÓÚÜÑáéíóúüñ]+", texto[start:end]):
        yield (start + m.start(), start + m.end())

def backtrack_turn_and_build_fragment(
    texto: str,
    match,
    pre_words: int = 7000,
    post_words: int = 5000
):
    n = len(texto)
    pos_ini, pos_fin = match.start(), match.end()
    token = texto[pos_ini:pos_fin]

    orador_label, block_start, has_header = _find_prev_speaker_header(texto, pos_ini, max_chars_back=20000,
max_lines_back=120)
    if not has_header:
        para_sep = texto.rfind("\n\n", 0, pos_ini)
        rough_start = (para_sep + 2) if para_sep != -1 else max(0, pos_ini - 1500)
        block_start = _adjust_to_word_start(texto, rough_start)

    next_limit = len(texto)
    seek_from = pos_fin
    probe_idx = texto.find("\n", seek_from)
    while probe_idx != -1:
        line_start = texto.rfind("\n", 0, probe_idx)
        line_start = 0 if line_start == -1 else line_start + 1
        line_txt = texto[line_start:probe_idx].strip()

        if line_txt in (":", ":"):
            prev_start = texto.rfind("\n", 0, line_start - 1)
            prev_start = 0 if prev_start == -1 else prev_start + 1
            prev_txt = texto[prev_start:line_start].strip()
            if SPEAKER_HEADER_RE.search(prev_txt) or EPOCA_TITULARES.search(prev_txt):
                next_limit = probe_idx + 1
                break
        elif SPEAKER_HEADER_RE.search(line_txt) or EPOCA_TITULARES.search(line_txt):
            next_limit = probe_idx + 1
            break

        probe_idx = texto.find("\n", probe_idx + 1)

    para_end = texto.find("\n\n", pos_fin, next_limit)
    if para_end != -1:
        para_end += 1
        next_limit = min(next_limit, para_end)

    blk_start = block_start
    blk_end = next_limit
    spans = list(_token_spans(texto, blk_start, blk_end))
    if not spans:
```

```
        frag = clean_context_noise(texto[blk_start:blk_end].strip())
        frag, _ = re.subn(rf"(?i){re.escape(token)}", lambda m: f"**{m.group(0)}**", frag, count=1)
        return orador_label, frag, has_header

    mid_idx = None
    for i, (a, b) in enumerate(spans):
        if a <= pos_ini < b or pos_ini < a:
            mid_idx = i
            break
    if mid_idx is None:
        mid_idx = len(spans) - 1

    start_idx = max(0, mid_idx - pre_words)
    end_idx = min(len(spans) - 1, mid_idx + post_words)
    win_start = spans[start_idx][0]
    win_end = spans[end_idx][1]

    para_start_within_block = texto.rfind("\n\n", blk_start, pos_ini)
    if para_start_within_block != -1:
        para_start_within_block = para_start_within_block + 2
        if para_start_within_block < win_start:
            win_start = para_start_within_block

    while win_start > blk_start and texto[win_start - 1] not in "\n ":
        win_start -= 1
    while win_end < blk_end and texto[win_end:win_end + 1] not in "\n ":
        win_end += 1

    frag = texto[win_start:win_end]

    rel_ini = pos_ini - win_start
    rel_fin = rel_ini + (pos_fin - pos_ini)
    highlighted = False
    if 0 <= rel_ini <= len(frag) and 0 <= rel_fin <= len(frag):
        frag = frag[:rel_ini] + "**" + frag[rel_ini:rel_fin] + "**" + frag[rel_fin:]
        highlighted = True

    frag = clean_context_noise(frag.strip())

    if (not highlighted) or (re.search(re.escape(token), frag, flags=re.IGNORECASE) is None):
        pat = re.compile(rf"(?i)\b{re.escape(token)}\b")
        if not pat.search(frag):
            pat = re.compile(rf"(?i){re.escape(token)}")
        frag, _ = pat.subn(lambda m: f"**{m.group(0)}**", frag, count=1)

    return orador_label, frag, has_header


def prefija_apellidos_en_contexto(orador_label: str, cuerpo: str, has_header: bool) -> str:
    if has_header:
        cuerpo_sin = re.sub(r"^[^:\n]{2,}:\s*", "", cuerpo.strip())
        return f"{orador_label}: {cuerpo_sin}"
    else:
        return cuerpo.strip()
```

Functions that reconstruct the context of the fragment where the keyword appears: they search for the beginning and end of the speech, highlight the quote in bold, and add the name of the speaker.

In [ ]:

```
# 7. Reading PDFs
DOC_CVE_RE = re.compile(r"\b(?:BOCG[_-]|DS(?:CD|CG)?[_-])[A-Z0-9_-]+", re.IGNORECASE)
NUM_SESION_RE = re.compile(r"(?:sesión\s*)?(?:n[úu]m(?:\.|ero)?)[\s:]*?(\d+)\s*\.?", re.IGNORECASE)


def read_pdfs_from_dirs(dirs):
    all_data = []
    print(" Inicio de la lectura de archivos PDF...")
    for dir_path in dirs:
        if not os.path.isdir(dir_path):
            print(f" Carpeta no encontrada: {dir_path}")
            continue
        for root, _, files in os.walk(dir_path):
            for file in files:
                if not file.lower().endswith(".pdf"):
                    continue
```

```
                file_path = os.path.join(root, file)
                legislatura = get_legislatura_from_path(file_path)
                print(f" Procesando PDF: {file_path} (Legislatura {legislatura})")

                texto = extract_text_from_pdf(file_path)
                texto = fix_ligaduras_y_espacios(texto)
                texto = limpiar_cabeceras_pies(texto)

                fecha = intenta_fecha_desde_texto(texto) or intenta_fecha_desde_nombre(file_path) or
"Desconocida"
                fecha = format_fecha(fecha)

                doc_id = os.path.splitext(os.path.basename(file_path))[0]

                documento_match = DOC_CVE_RE.search(texto)
                cve = documento_match.group() if documento_match else doc_id

                all_data.append({
                    "texto": texto,
                    "fecha": fecha,
                    "legislatura": legislatura,
                    "id": doc_id,
                    "cve": cve
                })
    print(f" Lectura de archivos completada. Total de documentos: {len(all_data)}")
    return all_data


data = read_pdfs_from_dirs(pdf_dirs)
```

Read PDFs from folders, apply OCR and cleaning, detect date, document (CVE) and legislature, and save everything in a list (data).

In [ ]:

```
# 8. Process documents

def process_documents(data, regex_patterns):
    results = []
    print(" Procesando documentos para menciones…")
    for idx, item in enumerate(data):
        text_original = item.get("texto", "")
        text_lower = text_original.lower()
        fecha = item.get("fecha", "Desconocida")
        legislatura = item.get("legislatura", "Desconocida")
        doc_id = item.get("id", "Sin ID")
        doc_id_str = str(doc_id)

        numero_sesion_match = NUM_SESION_RE.search(text_lower)
        numero_sesion = numero_sesion_match.group(1) if numero_sesion_match else "Desconocido"

        documento = item.get("cve", "Desconocido")

        fecha_formateada = format_fecha(fecha)

        print(f" Documento {idx + 1}/{len(data)}")
        ya_detectados = set()
        hits_doc = 0

        for grupo_index, pattern in enumerate(regex_patterns, start=1):
            for match in pattern.finditer(text_original):
                if preceded_by_hyphen(text_original, match.start()):
                    continue

                token = text_original[match.start():match.end()]

                if quitar_tildes(token).lower() == "sion" and looks_like_broken_spanish_sion(text_original,
match.start(), match.end()):
                    continue

                if is_all_uppercase_word(token):
                    continue

                orador_label, cuerpo, has_header = backtrack_turn_and_build_fragment(text_original, match)
                if looks_like_garbage(cuerpo):
                    continue
```

```
                contexto = prefija_apellidos_en_contexto(orador_label, cuerpo, has_header)

                clave = f"{orador_label}_{doc_id_str}_{match.start()}_{match.end()}"
                if clave in ya_detectados:
                    continue
                ya_detectados.add(clave)

                clasificacion = classify_mention(contexto)
                dic_conspiracion = "Sí" if clasificacion in ["A", "B", "C"] else "No"
                grupo_texto = f"Grupo {grupo_index}"
                modo_exp_implic = (
                    determinar_expresion(contexto, clasificacion)
                    if dic_conspiracion == "Sí" else "No procede"
                )

                results.append([
                    doc_id_str, fecha_formateada, legislatura, documento, numero_sesion,
                    (orador_label if has_header else "DESCONOCIDO"),
                    grupo_texto,
                    token, contexto,
                    clasificacion, dic_conspiracion, modo_exp_implic
                ])
                hits_doc += 1

        print(f"   → Menciones en este documento: {hits_doc}")
    print(" Total de menciones detectadas:", len(results))
    return results

detected_mentions = process_documents(data, regex_patterns)
```

Analyse the texts by detecting conspiratorial references, extract the context, identify the speaker, and classify each case.

In [ ]:

```
# 9. Generate CSV
output_folder = r"D:\TRABAJO\RESULTADOS"
os.makedirs(output_folder, exist_ok=True)
output_file = os.path.join(output_folder, "franquismo.csv")

columnas = [
    "ID", "Fecha", "Legislatura", "Documento", "Numero de Sesion",
    "Orador", "Grupo", "Cita", "Contexto",
    "Clasificacion", "Si/No Conspiracion", "Explicita/Implicita"
]

df_resultados = pd.DataFrame(detected_mentions, columns=columnas)
df_resultados.to_csv(output_file, sep=';', quoting=csv.QUOTE_ALL, index=False, encoding='utf-8')
print(f" Archivo guardado: {output_file}")
```

Convert the results into a *pandas* DataFrame and save them in a CSV file (franquismo.csv) with all the metadata.

In [ ]:

```
# 10. Showw results
try:
    df = pd.read_csv(output_file, delimiter=';', dtype=str)
    print(df.head())
except Exception as e:
    print(f" No se pudo leer el CSV generado para vista previa: {e}")
```

Displays the first few rows of the final CSV on screen as a preview.

AUSTRIAN DATABASE

In [ ]:

```
# TO BE UPDATED
```

GERMAN DATABASE

In [ ]:

```
# TO BE UPDATED
```

CZECH DATABASE

In [ ]:

```
# 1. Importing the required libraries
import os
import re
import csv
import pandas as pd
import unicodedata
import pdfplumber
import pytesseract
from pdf2image import convert_from_path
```

The script imports the libraries required to process the Czech parliamentary records. os is used to manage files, folders and paths; re provides regular expressions for text matching and extraction; csv is used for exporting the final dataset; and pandas is used to manage the parliamentary database and construct the final dataframe. unicodedata is used for text normalization. pdfplumber extracts text from PDF files, while pdf2image and pytesseract provide an OCR-based alternative when the PDF does not contain sufficiently extractable text.

In [ ]:

```
# 2. Loading the Czech MP database
diputados_df = pd.read_excel(
    r"D:\TRABAJO\DIPUTADOS\MPS_RCH.xlsx",
    sheet_name="RCH"
)

diputados_df.columns = diputados_df.columns.str.strip()
```

The script loads the Czech parliamentary database from an Excel file, specifically using the RCH worksheet. The column names are then stripped of unnecessary leading and trailing spaces to facilitate subsequent matching operations.

In [ ]:

```
# 3. Defining the text-normalization function
def normalizar_texto(texto):

    texto = str(texto).upper()

    reemplazos = {
        "Ä": "AE",
        "Ö": "OE",
        "Ü": "UE",
        "ß": "SS",
        "ß": "SS"
    }

    for original, nuevo in reemplazos.items():
        texto = texto.replace(original, nuevo)

    texto = ''.join(
        c for c in unicodedata.normalize('NFD', texto)
        if unicodedata.category(c) != 'Mn'
    )

    return texto.strip()
```

A text-normalization function is defined to standardize names and other textual information before matching. The function converts text to uppercase, replaces German special characters with standardized forms, removes Unicode diacritical marks and eliminates unnecessary surrounding spaces. This creates a standardized representation of names that can be compared more consistently between the parliamentary records and the external MP database.

In [ ]:

```
# 4. Defining the function for removing titles
def quitar_titulos(nombre):

    nombre = str(nombre)
```

```
    titulos = [
        "PROF. DR. H.C.",
        "PROF. DR.",
        "DR. H.C.",
        "PROF.",
        "PROFESSOR",
        "DR.",
        "DR.IN",
        "MAG.A",
        "MAG.",
        "MMAG.A",
        "MMAG.",
        "DIPL.-ING.",
        "DI",
        "ING.",
        "BA",
        "MA",
        "MBA",
        "BSC",
        "MSC",
        "LLM"
    ]

    for titulo in titulos:

        nombre = re.sub(
            re.escape(titulo),
            "",
            nombre,
            flags=re.IGNORECASE
        )

    return re.sub(r"\s+", " ", nombre).strip(" ,")
```

The script defines a second normalization function specifically for removing academic and professional titles from names. The list includes titles such as Prof., Dr., Mag., Dipl.-Ing., Ing. and several academic degree abbreviations. Each title is removed using regular expressions, after which repeated spaces and unnecessary punctuation are cleaned. This provides an additional standardized version of names without titles.

In [ ]:

```
# 5. Normalizing the MP database
diputados_df["LEGISLATURE_NORM"] = (
    diputados_df["LEGISLATURE"]
    .fillna("")
    .apply(normalizar_texto)
)

diputados_df["FULL_NAME_NORM"] = (
    diputados_df["FULL_NAME"]
    .fillna("")
    .apply(normalizar_texto)
)

diputados_df["FULL_NAME_NORM_SIN_TITULO"] = (
    diputados_df["FULL_NAME_NORM"]
    .apply(quitar_titulos)
    .str.replace(r"\s+", " ", regex=True)
    .str.strip()
)
```

Three normalized variables are created in the MP database: · LEGISLATURE_NORM, containing the normalized legislature; · FULL_NAME_NORM, containing the normalized full name; · FULL_NAME_NORM_SIN_TITULO, containing the normalized full name after removing academic and professional titles. The last variable provides an alternative matching strategy for cases where the parliamentary record and the database differ only because one version contains titles.

In [ ]:

```
# 6. Searching for the corresponding MP
def buscar_diputado(full_name, legislatura):

    full_name_norm = normalizar_texto(
        limpiar_nombre_orador(full_name)
```

```
    )

    legislatura_norm = normalizar_texto(
        legislatura
    )
```

The buscar_diputado() function attempts to identify the MP corresponding to a speaker found in the parliamentary record. The speaker's full name is first cleaned and normalized, and the legislature is normalized as well.

In [ ]:

```
# 7. First matching attempt: full name and legislature
candidatos = diputados_df[
    (diputados_df["FULL_NAME_NORM"] == full_name_norm) &
    (diputados_df["LEGISLATURE_NORM"] == legislatura_norm)
]
```

The first matching strategy searches for an exact match between the normalized full name and the normalized legislature. This restricts the identification to MPs belonging to the appropriate parliamentary period.

In [ ]:

```
# 8. Second matching attempt: ignoring titles
if candidatos.empty:

    full_name_sin_titulo = quitar_titulos(full_name_norm)

    full_name_sin_titulo = re.sub(
        r"\s+",
        " ",
        full_name_sin_titulo
    ).strip()

    candidatos = diputados_df[
        (diputados_df["FULL_NAME_NORM_SIN_TITULO"] == full_name_sin_titulo) &
        (diputados_df["LEGISLATURE_NORM"] == legislatura_norm)
    ]
```

If no exact match is found, the script performs a second search after removing academic and professional titles. This allows the speaker to be identified even when titles are present in one source but not the other.

In [ ]:

```
# 9. Handling unidentified speakers
if candidatos.empty:
    return (
        "No identificado",
        "No especificada"
    )
```

If neither matching strategy produces a candidate, the function records the speaker as unidentified and indicates that the political party and ideology are not specified.

In [ ]:

```
# 10. Retrieving party and ideology
fila = candidatos.iloc[0]

return (
    fila["POLITICAL PARTY"],
    fila["IDEOLOGY"]
)
```

When a candidate is identified, the script retrieves the corresponding political party and ideological classification from the Czech MP database. Thus, the speaker identified in the parliamentary document is enriched with information from the external database.

In [ ]:

```
# 11. Defining the input and output folders
folder_path = r"D:\TRABAJO\DIARIOS\RCH"
```

```
output_folder = r"D:\TRABAJO\RESULTADOS\RCH"
```

The script defines the folder containing the Czech parliamentary records and the folder where the resulting analytical datasets will be stored. The input directory contains the parliamentary documents organized into folders, while the output directory is used for the CSV files generated by the analysis.

In [ ]:

```
# 12. Defining the Czech keyword dictionary
KEYWORDS_CZECH = {
       "proimigračn",

        "migrant",

        "uprchlí",

        "koloniz",

        "záměn národ",

        "výměn národ",

        "nahrazen národ",

        "vystřídan národ"

    ]
}
```

The script establishes the Czech keyword classification framework through the KEYWORDS_CZECH dictionary. Unlike the Austrian version, this dictionary contains two theoretical categories:

- secret_elites
- great_replacement The keywords are represented largely as stems so that different morphological forms of Czech words can be detected.

In [ ]:

```
# 13. Converting keywords into regular expressions
# ========================================
# x COMPILAR REGEX x
# ========================================


def convertir_a_regex(termino):


    palabras = termino.split()


    regex = []


    for palabra in palabras:


        if palabra.endswith("*"):


            palabra = re.escape(palabra[:-1])


            regex.append(fr"{palabra}\w*")


        else:


            regex.append(re.escape(palabra))


    patron = r"\b" + r"\s+".join(regex) + r"\w*\b"
```

```
    return re.compile(
        patron,
        flags=re.IGNORECASE
    )
```

This function converts each keyword into a regular-expression pattern that can be searched in the parliamentary text. The keyword is first divided into separate words. If a word ends with *, the asterisk is treated as a wildcard, allowing the expression to detect additional word characters after the specified stem. The individual components are then combined while allowing flexible whitespace between words. Finally, the complete expression is compiled with case-insensitive matching. This makes the keyword search more flexible than a simple exact-string search and allows the procedure to capture variations of the stems included in the keyword dictionary.

In [ ]:

```
# 14. Compiling individual keyword lists
def compilar_lista(lista):

    return [
        convertir_a_regex(x)
        for x in lista
    ]
```

The compilar_lista() function applies the regular-expression conversion procedure to every keyword contained in a category. The result is a list of compiled regular-expression patterns ready to be searched against the parliamentary text.

In [ ]:

```
# 15. Compiling the keyword dictionary
def compilar_keywords(diccionario):

    resultado = {}

    for teoria, lista in diccionario.items():

        resultado[teoria] = compilar_lista(lista)

    return resultado
```

This function applies compilar_lista() to every theoretical category in the Czech keyword dictionary. For each theory, the original keyword list is transformed into a list of compiled regular-expression patterns. The resulting dictionary maintains the same theoretical structure as KEYWORDS_CZECH, but now contains searchable regular expressions instead of raw keyword strings.

In [ ]:

```
# 16. Creating the compiled Czech keyword dictionary
REGEX_CZECH = compilar_keywords(KEYWORDS_CZECH)
```

This line applies the previous compilation function to the complete Czech keyword dictionary. REGEX_CZECH therefore contains the two theoretical categories, secret_elites and great_replacement, with their keywords transformed into compiled regular-expression patterns. This object is subsequently used by the functions responsible for detecting theoretical references in parliamentary interventions.

In [ ]:

```
# 17. Summarizing detected keywords
def resumir_keywords(keywords):

    return ", ".join(sorted(set(keywords)))
```

This function converts the detected keyword matches into a single summarized string. set(keywords) removes duplicate matches, sorted() orders the remaining values, and “, “.join() combines them into one comma-separated field. This format allows the detected keywords to be stored together in the final analytical dataset without unnecessarily repeating identical matches.

In [ ]:

```
# 18. Analyzing a theory
def analizar_teoria(texto, nombre_teoria):

    encontrados = []

    for patron in REGEX_CZECH[nombre_teoria]:

        for match in patron.finditer(texto):

            encontrados.append(match.group(0))

    return {

        "cumple": len(encontrados) > 0,

        "keywords": encontrados,

        "n_keywords": len(set(encontrados))

    }
```

This function searches an intervention for all regular-expression patterns associated with one specific theory. Each pattern is searched throughout the text using finditer(), and every detected match is added to encontrados. The function then returns three elements. cumple indicates whether the intervention meets the criterion for that theory. In this Czech procedure, the criterion is simply whether at least one keyword has been detected keywords contains the detected keyword matches, while n_keywords records the number of unique detected keyword forms. Therefore, the Czech classification rule is based on the presence of at least one keyword associated with the theory.

In [ ]:

```
# 19. Analyzing an intervention
def analizar_intervencion(texto):

    resultados = {}

    for teoria in REGEX_CZECH.keys():

        resultados[teoria] = analizar_teoria(

            texto,

            teoria

        )

    return resultados
```

This function applies the theoretical keyword analysis to a complete parliamentary intervention. It iterates through every theory contained in REGEX_CZECH and calls analizar_teoria() for each one. The results are stored in a dictionary where each theory has its own classification result. Consequently, each intervention is evaluated independently for secret_elites and great_replacement.

In [ ]:

```
# 20. Cleaning the extracted text

def limpiar_texto(texto):

    texto = re.sub(r"-\s+", "", texto)

    texto = re.sub(r"[ \t]+", " ", texto)

    texto = re.sub(r"\n{3,}", "\n\n", texto)

    return texto.strip()
```

This function performs the basic cleaning of the text extracted from the parliamentary PDFs. It removes hyphenation caused by line breaks, reduces repeated spaces and tabs to a single space, and reduces excessive sequences of line breaks. Finally, leading and trailing whitespace is removed. The purpose is to create a cleaner text representation before the subsequent speaker identification, intervention segmentation and keyword detection stages.

In [ ]:

```
# 21. Extracting text from PDFs
# ========================================
# EXTRAER TEXTO PDF
# ========================================


def extraer_texto_pdf(pdf_path):

    texto_completo = ""


    try:

        with pdfplumber.open(pdf_path) as pdf:

            for page in pdf.pages:

                texto = page.extract_text()

                if texto:

                    texto_completo += texto + "\n"


    except Exception as e:

        print(f"⚠ Error leyendo PDF: {e}")


    if len(texto_completo.strip()) < 100:

        print(f"🧠 OCR necesario: {os.path.basename(pdf_path)}")


        try:

            paginas = convert_from_path(pdf_path)

            texto_ocr = ""


            for pagina in paginas:

                texto_ocr += pytesseract.image_to_string(

                    pagina,

                    lang="ces"

                )


            texto_completo = texto_ocr


        except Exception as e:
```

```
            print(f"❌ Error OCR: {e}")

    return limpiar_texto(texto_completo)
```

This function extracts the text from each parliamentary PDF. It first attempts direct extraction with pdfplumber, processing all pages and concatenating the available text. If the extracted text contains fewer than 100 characters, the function considers the extraction insufficient and activates an OCR procedure. The PDF pages are converted into images and processed using pytesseract with the Czech language model (lang="ces"). The resulting OCR text replaces the insufficient direct extraction. Finally, the extracted text is passed through limpiar_texto() and returned. Thus, the function provides a fallback OCR mechanism for PDFs whose text cannot be adequately extracted directly.

In [ ]:

```
# 22. Reading PDFs from the subfolders
# ========================================
# LEER PDFs EN SUBCARPETAS
# ========================================

def read_pdf_files(folder_path):

    print("ENTRA EN read_pdf_files")

    all_data = []

    pdf_files = []

    for root, dirs, files in os.walk(folder_path):

        # PDFs de ESTA carpeta

        archivos_pdf = sorted([

            file for file in files

            if file.lower().endswith(".pdf")

        ])

        ##################################

        # SOLO PARA PRUEBAS:
#########################################################

        #archivos_pdf = archivos_pdf[:5]

        ##################################

        for file in archivos_pdf:

            pdf_files.append(os.path.join(root, file))

    print(f"Se van a procesar {len(pdf_files)} PDFs")

    for idx, pdf_path in enumerate(pdf_files):

        file = os.path.basename(pdf_path)

        legislatura = os.path.basename(os.path.dirname(pdf_path))

        print(f"Procesando {idx + 1}/{len(pdf_files)} → {legislatura}/{file}")

        texto = extraer_texto_pdf(pdf_path)
```

```
        all_data.append({
            "id": file,
            "texto": texto,
            "pdf_origen": file,
            "ruta_pdf": pdf_path,
            "legislatura": legislatura
        })

    return all_data
```

This function constructs the parliamentary corpus from the PDFs stored in the selected folder and its subfolders. It recursively searches the directory structure with os.walk() and identifies files ending in .pdf. Each PDF path is stored in pdf_files. The commented testing line allows the researcher to limit the number of PDFs processed when necessary. For every PDF, the function identifies its filename and derives the legislature from the name of its parent directory. The PDF is then passed to extraer_texto_pdf() for text extraction. Each document is stored as a dictionary containing its ID, extracted text, source filename, complete path and legislature. The complete collection of documents is returned as all_data.

In [ ]:

```
# 23. Cleaning the speaker name
# ========================================
# LIMPIAR NOMBRE DEL ORADOR
# ========================================

def limpiar_nombre_orador(nombre):

    nombre = str(nombre)

    nombre = re.sub(r"\s+", " ", nombre).strip()

    # Eliminar cargos parlamentarios checos
    nombre = re.sub(
        r"^(Poslanec|Poslankyně|"
        r"Předseda PSP|Předsedkyně PSP|"
        r"Místopředseda PSP|Místopředsedkyně PSP|"
        r"Předseda vlády ČR|Předsedkyně vlády ČR|"
        r"Místopředseda vlády|Místopředsedkyně vlády|"
        r"Ministr|Ministryně|"
        r"Senátor|Senátorka)\s+",
        "",
        nombre,
        flags=re.IGNORECASE
    )

    # Eliminar ministerios o cargos adicionales
    nombre = re.sub(
        r"^(ministr|ministryně).*?ČR\s+",
```

```
        "",
        nombre,
        flags=re.IGNORECASE
    )

    nombre = re.sub(r"\s+", " ", nombre).strip(" ,")

    return nombre
```

This function cleans the speaker information extracted from the parliamentary documents. It first standardizes whitespace and then removes Czech parliamentary titles and positions that can appear before the person's name, including parliamentary, governmental and ministerial positions. A second regular-expression operation removes additional ministerial descriptions containing ČR. The function then performs a final whitespace and punctuation cleanup and returns the resulting speaker name. This cleaned name is subsequently used to identify the corresponding MP in the reference database.

In [ ]:

```
# 24. Identifying speaker markers
PATRON_ORADOR = re.compile(
    r"""(?mx)

^(
(?:Pan\ poslanec|
Paní\ poslankyně|
Poslanec|
Poslankyně|
Předsedající|
Předseda\ PSP|
Předsedkyně\ PSP|
Místopředseda\ PSP|
Místopředsedkyně\ PSP|
Předseda\ vlády|
Předsedkyně\ vlády|
Místopředseda\ vlády|
Místopředsedkyně\ vlády|
Ministr|
Ministryně)

[^\n:]{0,200}
):

"""
)
```

This block defines the regular expression used to detect parliamentary speaker markers. The pattern identifies lines beginning

with Czech forms of address or institutional positions such as Pan poslanec, Paní poslankyně, Poslanec, Poslankyně, parliamentary chair positions, government positions and ministers. After the identified position, the expression allows up to 200 characters before a colon. The resulting compiled pattern is stored in PATRON_ORADOR and is subsequently used to locate speakers throughout each parliamentary document.

In [ ]:

```
# 25. Removing the cover page and table of contents

def eliminar_portada_indice(texto):


    patron = re.compile(

        r"Zahájení schůze Poslanecké sněmovny",

        flags=re.IGNORECASE

    )


    m = patron.search(texto)


    if m:

        return texto[m.start():]


    return texto
```

This function removes the introductory material that appears before the beginning of the parliamentary sitting. It searches for the phrase Zahájení schůze Poslanecké sněmovny, which is used as the point marking the start of the parliamentary proceedings. If the phrase is found, everything before that position is discarded and the text from that point onward is retained. If the phrase cannot be found, the original text is returned unchanged.

In [ ]:

```
# 26. Obtaining the speakers in a document
def obtener_oradores_documento(texto):


    oradores = []


    for m in PATRON_ORADOR.finditer(texto):


        oradores.append({


            "inicio": m.start(),


            "nombre": limpiar_nombre_orador(

                m.group(1)

            ),


            "partido_diario": ""


        })


    return oradores
```

This function searches the complete document for all occurrences of the previously defined speaker pattern. For every match, it records the position where the speaker marker begins and cleans the identified speaker name using limpiar_nombre_orador(). It also creates the partido_diario field, which is left empty in this script. The resulting list contains the speaker markers and their positions, which are necessary for subsequently segmenting the document into individual interventions.

In [ ]:

```
# 27. Segmenting the interventions
def obtener_intervenciones(texto):

    oradores = obtener_oradores_documento(texto)

    intervenciones = []

    for i, orador in enumerate(oradores):

        inicio = orador["inicio"]

        if i + 1 < len(oradores):
            fin = oradores[i + 1]["inicio"]
        else:
            fin = len(texto)

        intervencion = texto[inicio:fin].strip()

        intervenciones.append({

            "texto": intervencion,

            "orador": orador

        })

    return intervenciones
```

This function divides the parliamentary document into individual interventions. It first obtains all speaker markers and their positions. For each speaker, the beginning of the intervention is the position of that speaker's marker. The end is defined as the position where the next speaker begins. For the last speaker, the intervention extends to the end of the document. Each resulting segment is stored together with the corresponding speaker information. This establishes the individual parliamentary intervention as the unit of textual analysis used by the subsequent keyword-classification procedure.

In [ ]:

```
# 28. Extracting the date
# ========================================
# EXTRAER FECHA
# ========================================

def extraer_fecha(texto):

    patrones_fecha = [

        r"(\d{1,2}\.\d{1,2}\.\d{4})",

r"(\d{1,2}\.\s*(ledna|února|března|dubna|května|června|července|srpna|září|října|listopadu|prosince)\s*(\d{4})
)"

    ]

    for patron in patrones_fecha:
```

```
        match = re.search(
            patron,
            texto,
            flags=re.IGNORECASE
        )

        if match:
            return match.group(0)

    return "Fecha desconocida"
```

This function extracts the date of the parliamentary document. It searches for two possible formats: a numerical date such as 15.3.2022, and a date written with the Czech name of the month. The patterns are checked sequentially. When a date is found, the complete matched expression is returned. If neither pattern produces a result, the function returns Fecha desconocida. The extracted date is later incorporated into the analytical observations generated from the document.

In [ ]:

```
# 29. Extracting the session number
# ========================================
# EXTRAER NÚMERO SESIÓN
# ========================================

def extraer_numero_sesion(texto):

    patron = r"o\s+(\d+)\.\s*schůzi"

    match = re.search(
        patron,
        texto,
        flags=re.IGNORECASE
    )

    if match:
        return match.group(1)

    patron2 = r"(\d+)\.\s*schůze"

    match2 = re.search(
        patron2,
        texto,
        flags=re.IGNORECASE
    )

    if match2:
        return match2.group(1)

    return "Desconocido"
```

This function extracts the number of the parliamentary sitting. It first searches for the formulation represented by o X. schůzi. If this is not found, it searches for the alternative formulation X. schůze. When either pattern matches, the numerical session identifier is returned. If no session number is detected, the function returns Desconocido.

In [ ]:

```
# 30. Completing speaker information
# ========================================
# OBTENER INFORMACIÓN DEL ORADOR
# ========================================


def completar_info_orador(orador, legislatura):


    partido, ideologia = buscar_diputado(

        orador["nombre"],

        legislatura

    )


    return {


        "nombre": orador["nombre"],

        "partido_diario": orador["partido_diario"],

        "partido": partido,

        "ideologia": ideologia


    }
```

This function enriches the detected speaker with information from the MPs' reference database. The speaker's cleaned name and the relevant legislature are passed to buscar_diputado(). The resulting party and ideology are combined with the original speaker information. The returned dictionary therefore contains the speaker's name, the diary-party field, the database-derived political party and the database-derived ideology.

In [ ]:

```
# 31. Processing an individual intervention
# ========================================
# x PROCESAR DOCUMENTOS
# ========================================


def procesar_intervencion(

    intervencion,

    info_orador,

    doc_id,

    pdf_origen,

    legislatura,

    ruta_pdf,

    fecha,

    numero_sesion

):


    resultados = analizar_intervencion(intervencion)
```

```
    filas = []

    for teoria, resultado in resultados.items():

        if not resultado["cumple"]:
            continue

        filas.append([

            doc_id,
            pdf_origen,
            legislatura,
            ruta_pdf,
            fecha,
            numero_sesion,

            info_orador["nombre"],
            info_orador["partido_diario"],
            info_orador["partido"],
            info_orador["ideologia"],

            teoria,

            resultado["n_keywords"],

            resumir_keywords(resultado["keywords"]),

            intervencion

        ])

    return filas
```

This function transforms a classified intervention into the rows that will form the final analytical dataset. It first applies analizar_intervencion() to the intervention. For every theory, the function checks whether the intervention satisfies the classification criterion. If cumple is false, that theory is not included. For every positive classification, a complete analytical row is created containing the document information, legislature, date, session number, speaker, party, ideology, theory, number of detected keywords, detected keywords and complete intervention text. Thus, this block connects the keyword classification with the construction of the final analytical observations.

In [ ]:

```
# 32. Processing all documents
def process_documents(data):

    print("ENTRA EN process_documents (NUEVO)")

    results = []

    print("Procesando documentos...")
```

```
for i, item in enumerate(data, start=1):

    print(f"📄 Procesando documento {i}/{len(data)}: {item['id']}")

    text_original = item["texto"]
    text_original = eliminar_portada_indice(text_original)

    doc_id = item["id"]
    pdf_origen = item["pdf_origen"]
    legislatura = item["legislatura"]
    ruta_pdf = item["ruta_pdf"]

    fecha = extraer_fecha(text_original)
    numero_sesion = extraer_numero_sesion(text_original)

    intervenciones = obtener_intervenciones(text_original)
    print(f"Intervenciones encontradas: {len(intervenciones)}")

    for intervencion in intervenciones:

        texto = intervencion["texto"]

        orador = intervencion["orador"]

        info_orador = completar_info_orador(
            orador,
            legislatura
        )

        filas = procesar_intervencion(

            texto,
            info_orador,
            doc_id,
            pdf_origen,
            legislatura,
            ruta_pdf,
            fecha,
            numero_sesion

        )

        results.extend(filas)

print(f"Total menciones encontradas: {len(results)}")
```

```
    return results
```

This is the main function coordinating the complete processing of the parliamentary corpus. For each document, it removes the preliminary material, retrieves the document metadata, extracts the date and session number, and identifies the individual interventions. Each intervention is then linked to its speaker, and the speaker is matched to the MPs' database to obtain party and ideology. The intervention is subsequently classified according to the two theoretical keyword dictionaries. The resulting rows are added to results. After all documents and interventions have been processed, the function returns the complete collection of detected analytical observations.

In [ ]:

```
# 33. Starting the execution
# ========================================
# EJECUTAR
# ========================================


print("LLEGA AL EJECUTAR")
```

This block marks the beginning of the execution section of the script. The print() command is a progress indicator showing that the script has reached the point at which the previously defined functions are actually executed.

In [ ]:

```
# 34. Selecting the parliamentary period and running the analysis

carpeta = "2021-2025"



ruta_carpeta = os.path.join(

    folder_path,

    carpeta

)



print(f"PROCESANDO CARPETA {carpeta}")

print(f"Carpeta: {ruta_carpeta}")



data = read_pdf_files(ruta_carpeta)



# PARA PRUEBAS: 1 PDF DE CADA CARPETA

#data = read_pdf_files(folder_path)



detected_mentions = process_documents(data)
```

This block specifies the parliamentary period to be processed and launches the complete analytical workflow. The selected folder is 2021-2025. The complete path is then constructed by combining this folder with the previously defined root directory. read_pdf_files() loads and processes the PDFs contained in that parliamentary period. The commented line provides an alternative testing configuration. Finally, process_documents(data) processes the complete corpus and stores the resulting analytical observations in detected_mentions.

In [ ]:

```
# 35. Creating the output directory

os.makedirs(output_folder, exist_ok=True)
```

This block creates the directory in which the final analytical dataset will be saved. If the directory already exists, exist_ok=True prevents the command from producing an error.

In [ ]:

```
# 36. Defining the output file
output_file = os.path.join(

    output_folder,

    #f"menciones_prueba.csv"

    f"menciones_{carpeta}.csv"

)
```

This block constructs the final output path and filename.

In [ ]:

```
# 37. Defining the final analytical variables
columnas = [

    "ID",

    "PDF_Source",

    "Legislature",

    "PDF_Path",

    "Date",

    "Session_Number",

    "Speaker",

    "Party_Diary",

    "Party",

    "Ideology",

    "Theory",

    "N_Keywords",

    "Keywords",

    "Intervention"

]
```

This block defines the complete structure of the final analytical dataset. The variables combine document-level information, parliamentary metadata, speaker characteristics and the results of the theoretical keyword classification. The final dataset therefore records the source document, legislature, date, session, speaker, party, ideology, detected theory, number of detected keywords, the keywords themselves and the complete intervention.

In [ ]:

```
# 38. Creating the final DataFrame
df_resultados = pd.DataFrame(

    detected_mentions,

    columns=columnas

)
```

In [ ]:

```
This block converts the analytical observations stored in detected_mentions into a pandas DataFrame.
The columnas list establishes the names and order of the variables.
At this stage, the final analytical dataset exists as the DataFrame df_resultados.
```

In [ ]:

```
# 39. Exporting the final analytical dataset
df_resultados.to_csv(
```

```
    output_file,

    sep=";",

    quoting=csv.QUOTE_ALL,

    index=False,

    encoding="utf-8-sig"

)
```

This block writes the final DataFrame to the CSV file. The semicolon is used as the column separator, all fields are quoted, and the DataFrame index is excluded from the export. The file is saved using utf-8-sig, which preserves the Czech characters contained in the dataset and improves compatibility with spreadsheet applications. This is the point at which the final analytical dataset is physically saved to disk.

In [ ]:

```
# 40. Confirming the export
print(f"📁 Archivo guardado: {output_file}")
```

It provides a simple execution message showing the path stored in output_file

In [ ]:

```
# 41. Reloading the exported CSV
df = pd.read_csv(

    output_file,

    delimiter=";"

)
```

This block reads the newly generated CSV file back into pandas. The delimiter is explicitly set to ;, corresponding to the separator used during export. This provides a basic verification that the generated file can be reopened correctly.

In [ ]:

```
# 42. Checking the resulting dataset
print(df.head())
```

This line displays the first rows of the reloaded dataset. It allows the researcher to perform a quick visual check that the output file contains the expected records and variables.

In [ ]:

```
# 43. Confirming completion of the legislature
#print(f"✅ Prueba finalizada.\n")

print(f"✅ Legislatura {carpeta} finalizada.\n")
```

The first line is commented out and therefore not executed. The active line confirms that the parliamentary period stored in carpeta has finished processing. With the current configuration, this corresponds to the 2021-2025 parliamentary period.

In [ ]:

```
# 44. Deleting temporary objects
del data

del detected_mentions

del df_resultados

del df
```

This final block removes the temporary objects created during processing from the current Python session. data contains the processed corpus, detected_mentions contains the analytical observations, df_resultados contains the final DataFrame, and df contains the DataFrame reloaded from the exported CSV. These commands only remove the objects from memory. They do

not delete the original parliamentary PDFs or the final CSV file. This marks the end of the script.

SLOVAKIAN DATABASE

In [ ]:

```
# 1. Importing the required libraries

import os

import re

import csv

import pandas as pd

import unicodedata

from docx import Document
```

This block imports all the libraries required by the script. os is used to work with folders, file paths and the directory structure containing the parliamentary documents. re provides regular expressions, which are used throughout the script for text cleaning, speaker identification, date extraction and keyword detection. csv is used when exporting the final analytical dataset, particularly for controlling the quotation of CSV fields. pandas is used to load the MPs' reference database, manipulate the data and create the final analytical DataFrame. unicodedata is used to normalize textual characters. Finally, Document from python-docx is used to read and extract the content of the Slovak parliamentary DOCX documents

In [ ]:

```
# 2. Loading the MPs' reference database

diputados_df = pd.read_excel(

    r"D:\TRABAJO\DIPUTADOS\MPS_RCH.xlsx",

    sheet_name="SLV"

)


diputados_df.columns = diputados_df.columns.str.strip()
```

This block loads the MPs' reference database from the Excel file. The script uses the SLV worksheet, corresponding to the Slovak parliamentary dataset. The database is loaded into the pandas DataFrame diputados_df. The column names are then stripped of leading and trailing whitespace to ensure that they can be referenced consistently later in the script. This database is subsequently used to associate detected parliamentary speakers with their political party and ideological classification.

In [ ]:

```
# 3. Defining the text normalization function

def normalizar_texto(texto):


    texto = str(texto).upper()


    texto = ''.join(

        c for c in unicodedata.normalize('NFD', texto)

        if unicodedata.category(c) != 'Mn'

    )


    return texto.strip()
```

This function standardizes text so that names and other textual information can be compared more consistently. First, the input is converted into a string and transformed to uppercase. The text is then decomposed using Unicode normalization (NFD). Characters classified as combining marks are removed, which eliminates diacritics. For example, accented characters can therefore be converted into their non-accented equivalents. Finally, surrounding whitespace is removed. This normalized representation is used when matching speakers against the MPs' reference database.

In [ ]:

```
# 4. Removing academic and professional titles
def quitar_titulos(nombre):


    nombre = str(nombre)


    titulos = [
        "PROF. DR. H.C.",
        "PROF. DR.",
        "DR. H.C.",
        "PROF.",
        "PROFESSOR",
        "DR.",
        "DR.IN",
        "MAG.A",
        "MAG.",
        "MMAG.A",
        "MMAG.",
        "DIPL.-ING.",
        "DI",
        "ING.",
        "BA",
        "MA",
        "MBA",
        "BSC",
        "MSC",
        "LLM"
    ]


    for titulo in titulos:
        nombre = re.sub(
            re.escape(titulo),
            "",
            nombre,
            flags=re.IGNORECASE
        )
```

```
    return re.sub(r"\s+", " ", nombre).strip(" ,")
```

This function removes academic and professional titles that may appear as part of an MP's full name. The list includes titles such as PROF., DR., MAG., DIPL.-ING., MBA, MSC and LLM. Each title is removed using a case-insensitive regular-expression substitution. After removing the titles, repeated spaces are reduced and unnecessary spaces or commas at the beginning or end of the name are removed. This creates a standardized version of the name without titles, which is used as a fallback when matching speakers to the MPs' database.

In [ ]:

```
# 5. Normalizing the MPs' reference database
# ==========================================
# NORMALIZAR BASE DE DIPUTADOS
# ==========================================



diputados_df["LEGISLATURE_NORM"] = (

    diputados_df["LEGISLATURE"]

    .fillna("")

    .apply(normalizar_texto)

)




diputados_df["FULL_NAME_NORM"] = (

    diputados_df["FULL_NAME"]

    .fillna("")

    .apply(normalizar_texto)

)



diputados_df["FULL_NAME_NORM_SIN_TITULO"] = (

    diputados_df["FULL_NAME_NORM"]

    .apply(quitar_titulos)

    .str.replace(r"\s+", " ", regex=True)

    .str.strip()

)
```

This block creates normalized versions of the key fields in the MPs' database. LEGISLATURE_NORM contains the normalized legislature information. FULL_NAME_NORM contains the normalized full name of each MP. FULL_NAME_NORM_SIN_TITULO removes academic and professional titles from the normalized full name. Missing values are first replaced with empty strings. These additional normalized columns make it possible to perform exact matching while also providing a second matching strategy that ignores titles.

In [ ]:

```
# 6. Searching for an MP in the reference database
# ================================================
# BUSCAR DIPUTADO EN LA BASE
# ================================================



def buscar_diputado(full_name, legislatura):
```

```
full_name_norm = normalizar_texto(
    limpiar_nombre_orador(full_name)
)

legislatura_norm = normalizar_texto(
    legislatura
)



candidatos = diputados_df[
    (diputados_df["FULL_NAME_NORM"] == full_name_norm) &
    (diputados_df["LEGISLATURE_NORM"] == legislatura_norm)
]

# Segundo intento: ignorando títulos
if candidatos.empty:

    full_name_sin_titulo = quitar_titulos(full_name_norm)
    full_name_sin_titulo = re.sub(
        r"\s+",
        " ",
        full_name_sin_titulo
    ).strip()

    candidatos = diputados_df[
        (diputados_df["FULL_NAME_NORM_SIN_TITULO"] == full_name_sin_titulo) &
        (diputados_df["LEGISLATURE_NORM"] == legislatura_norm)
    ]

if candidatos.empty:
    return (
        "No identificado",
        "No especificada"
    )

fila = candidatos.iloc[0]
```

```
    return (
        fila["POLITICAL PARTY"],
        fila["IDEOLOGY"]
    )
```

This function identifies the MP corresponding to a detected parliamentary speaker. First, the speaker's name is normalized using normalizar_texto() after being cleaned with limpiar_nombre_orador(). The legislature is also normalized. The first matching attempt requires both:

- an exact normalized full-name match;
- an exact normalized legislature match. If no candidate is found, the function performs a second matching attempt using the version of the name with professional and academic titles removed.

In [ ]:

```
# 7. Defining the corpus and output folders

folder_path = r"D:\TRABAJO\DIARIOS\SLV"

output_folder = r"D:\TRABAJO\RESULTADOS\SLV"
```

This block defines the two main filesystem locations used by the script. folder_path identifies the root directory containing the Slovak parliamentary DOCX documents. output_folder identifies the directory where the final analytical CSV files will be stored. The script subsequently combines these paths with the selected parliamentary period.

In [ ]:

```
# 8. Defining the Slovak keyword dictionary
# ========================================
# SLVAK KEYWORDS
# ========================================


KEYWORDS_SLOVAK = {


     "secret_elites": [


        "elit",
        "establishment",
        "establišment",
        "stranick klik",
        "kartel",
        "centr moc",
        "poskok",
        "posluhovač",
        "sluh",
        "stafáž",
        "podvol",
        "globalis",
        "zákulis",
        "riad zvonku",
        "globalistic siet",
```

```
        "superštát",
        "cudz záuj",
        "zahranič záuj",
        "serviln",
        "iluminát",
        "soroš",
        "bábkov vlád",
        "skryt",
        "spiknut",
        "sprisahan",
        "konšpira",
        "nov svet rád",
        "deep state",
        "hlbok štát",
        "hlbin štát",
        "tajn spoločens",
        "nevolen",
        "nitk",
        "tieň vlád",
        "paralel štruktúr"

    ],

    "great_replacement": [

        "migrac",
        "migran",
        "migrač",

        "demograf výmen",

        "veľk výmen",
        "veľk nahrad",
        "veľk vystriedan",

        "chirurg z dovoz",

        "moslim",
```

```
    "hostitel zem",
    "hostitel štát",
    "hostitel krajin",

    "soroš",

    "výmen civilizac",

    "islam",

    "genocíd európan",
    "genocíd beloch",

    "prisťahoval",

    "zánik národ",
    "zánik kultúr",
    "zánik civilizác",
    "zánik eúrop",

    "ohrozen identit",
    "ohrozen kultur",
    "ohrozen hodnot",

    "nahrad populác",
    "nahrad obyvateľ",

    "biel genocíd",
    "biel populác",
    "biel ras",
    "biel obyvateľ",

    "multikultur zlyhan",
    "multikultur experiment",

    "zničen civilizác",

    "likvidác národ",
```

```
        "likvidác identit",
        "likvidác kultúr",
        "likvidác civiliz",

        "proimigračn",

        "utečen",

        "koloniz",

        "zámen národ",
        "výmen národ",
        "nahrad národ",
        "vystriedan národ"
    ]
```

This block defines the complete Slovak keyword dictionary used to operationalize the two theoretical concepts. The first category, secret_elites, contains terms associated with elites, establishments, hidden power structures, external control, conspiracies, secret organizations and unelected authorities. The second category, great_replacement, contains terms associated with migration, demographic replacement, Islam, population replacement, cultural or civilizational decline and related formulations. The two theoretical categories remain separate so that each parliamentary intervention can subsequently be evaluated independently against each concept.

In [ ]:

```
# 9. Converting keywords into regular expressions
def convertir_a_regex(termino):

    palabras = termino.split()

    regex = []

    for palabra in palabras:

        if palabra.endswith("*"):

            palabra = re.escape(palabra[:-1])

            regex.append(fr"{palabra}\w*")

        else:

            regex.append(re.escape(palabra))
```

```
    patron = r"\b" + r"\s+".join(regex) + r"\w*\b"


    return re.compile(
        patron,
        flags=re.IGNORECASE
    )
```

This function converts each keyword into a regular-expression pattern. The keyword is first divided into separate words. When a word ends with *, the asterisk is treated as a wildcard and the corresponding stem is allowed to be followed by additional word characters. The individual components are then combined with flexible whitespace between them. Finally, the complete pattern is compiled using case-insensitive matching. This allows the keyword detection procedure to identify textual variations rather than relying exclusively on exact string matches.

In [ ]:

```
# 10. Compiling individual keyword lists
def compilar_lista(lista):


    return [
        convertir_a_regex(x)
        for x in lista
    ]
```

The compilar_lista() function applies convertir_a_regex() to every keyword contained in a theoretical category. The result is a list of compiled regular-expression patterns that can subsequently be searched against parliamentary text.

In [ ]:

```
# 11. Compiling the complete keyword dictionary
def compilar_keywords(diccionario):


    resultado = {}


    for teoria, lista in diccionario.items():


        resultado[teoria] = compilar_lista(lista)


    return resultado
```

This function applies the previous compilation procedure to every theoretical category in the keyword dictionary. For each theory, the original keyword list is transformed into a list of compiled regular-expression patterns. The resulting dictionary preserves the theoretical structure of the original keyword dictionary while making all keywords directly searchable.

In [ ]:

```
# 12. Creating the compiled keyword dictionary
REGEX = compilar_keywords(KEYWORDS_SLOVAK)
```

This line compiles the complete Slovak keyword dictionary. It is subsequently used by the functions that classify parliamentary interventions according to the two theoretical concepts.

In [ ]:

```
# 13. Summarizing detected keywords
def resumir_keywords(keywords):

    return ", ".join(sorted(set(keywords)))
```

This function converts the detected keyword matches into a single comma-separated string. set() removes duplicate matches, sorted() orders the remaining values and join() combines them into a single field. This allows the final analytical dataset to retain the detected keywords in a compact format.

In [ ]:

```
# 14. Analyzing a specific theory
def analizar_teoria(texto, nombre_teoria):

    encontrados = []

    for patron in REGEX[nombre_teoria]:

        for match in patron.finditer(texto):

            encontrados.append(match.group(0))

    return {

        "cumple": len(encontrados) > 0,

        "keywords": encontrados,

        "n_keywords": len(set(encontrados))

    }
```

This function searches an intervention for all keyword patterns associated with one specific theory. Each compiled regular expression belonging to the selected theory is searched throughout the intervention. Every detected match is added to encontrados. The function then returns three elements. cumple indicates whether the intervention satisfies the criterion for the theory.

In [ ]:

```
# 15. Analyzing a complete intervention
def analizar_intervencion(texto):

    resultados = {}

    for teoria in REGEX.keys():

        resultados[teoria] = analizar_teoria(

            texto,
```

```
            teoria
        )

    return resultados
```

This function applies the theoretical keyword analysis to a complete parliamentary intervention. It iterates through all theories contained in REGEX and applies analizar_teoria() independently to each one. The resulting classifications are stored in a dictionary. Consequently, each intervention is evaluated separately for secret_elites and great_replacement.

In [ ]:

```
# 16. Cleaning the extracted text
# ========================================
# LIMPIEZA TEXTO
# ========================================


def limpiar_texto(texto):
    texto = re.sub(r"-\s+", "", texto)
    texto = re.sub(r"[ \t]+", " ", texto)
    texto = re.sub(r"\n{3,}", "\n\n", texto)
    return texto.strip()
```

This function performs the basic cleaning of the text extracted from the parliamentary documents. It removes hyphenation produced by line breaks, reduces repeated spaces and tabs, and reduces excessive sequences of line breaks. Finally, leading and trailing whitespace is removed. The resulting text is used in the subsequent speaker identification, intervention segmentation and keyword detection stages.

In [ ]:

```
# 17. Extracting text from DOCX documents
def extraer_texto_docx(docx_path):

    doc = Document(docx_path)

    texto = []

    for p in doc.paragraphs:
        texto.append(p.text)

    for table in doc.tables:
        for row in table.rows:
            texto.append(
                " ".join(cell.text for cell in row.cells)
            )

    return limpiar_texto("\n".join(texto))
```

This function extracts the textual content of a DOCX parliamentary document. The document is opened using Document() from the python-docx library. The function first extracts the text from all paragraphs. It then processes all tables, extracting the text from each cell and joining the cells belonging to each row. The paragraph and table content are combined into a single text string, with line breaks separating the different elements. Finally, the resulting text is passed through limpiar_texto(). Therefore, this extraction procedure captures both ordinary paragraph text and text contained in tables within the DOCX documents.

In [ ]:

```
# 18. Reading DOCX documents from subfolders

def read_doc_files(folder_path):


    all_data = []

    doc_files = []


    for root, dirs, files in os.walk(folder_path):


        # Docxs de ESTA carpeta

        archivos_doc = sorted([

            file for file in files

            if file.lower().endswith(".docx")

        ])


        ###################################

        # SOLO PARA PRUEBAS:
#####################################################

        #archivos_doc = archivos_doc[:1]

        ###################################


        for file in archivos_doc:

            doc_files.append(os.path.join(root, file))


    print(f"Se van a procesar {len(doc_files)} documentos")


    for idx, doc_path in enumerate(doc_files):


        file = os.path.basename(doc_path)

        legislatura = os.path.basename(os.path.dirname(doc_path))


        print(f"Procesando {idx + 1}/{len(doc_files)} → {legislatura}/{file}")


        texto = extraer_texto_docx(doc_path)


        all_data.append({
```

```
            "id": file,
            "texto": texto,
            "doc_origen": file,
            "ruta_docx": doc_path,
            "legislatura": legislatura
        })

    return all_data
```

This function constructs the parliamentary corpus from the DOCX files stored in the selected directory and its subfolders. It recursively searches the folder structure using os.walk() and identifies files whose names end in .docx. The documents are stored in doc_files. The commented testing line can be used to restrict processing to a single document. For each document, the function identifies the filename and derives the legislature from the name of the parent folder. The document is then passed to extraer_texto_docx().

In [ ]:

```
# 19. Cleaning the speaker name
# ========================================
# LIMPIAR NOMBRE DEL ORADOR
# ========================================

def limpiar_nombre_orador(nombre):

    nombre = str(nombre).strip()

    partes = [x.strip() for x in nombre.split(",")]

    if len(partes) >= 2:
        return f"{partes[0]}, {partes[1]}"

    return nombre
```

This function standardizes the speaker name extracted from the parliamentary document. The name is first converted to a string and stripped of surrounding whitespace. It is then divided into parts using commas. If two or more parts are present, the function retains the first two and reconstructs them as a standardized part, part format. If fewer than two parts are present, the original cleaned name is returned. This standardized name is later used to match the speaker against the MPs' reference database.

In [ ]:

```
# 20. Identifying speaker markers
PATRON_ORADOR = re.compile(
    r"""(?mx)

^(
[A-ZÁÄČĎÉÍĹĽÑÓÔŔŠŤÚÝŽ][^,\n]+,
\s\*
[^,\n]+,
[^\n]+
```

```
)


"""

)
```

This block defines the regular-expression pattern used to identify parliamentary speaker markers in the Slovak documents. The expression searches for a specific structure beginning at the start of a line. It requires an initial uppercase character from the specified character set, followed by text and commas in the structure represented by the expression. The pattern is compiled with multiline and extended regular-expression modes through (?mx). The resulting PATRON_ORADOR object is later used to locate every speaker marker in the document.

In [ ]:

```
# 21. Removing the cover page and index

def eliminar_portada_indice(texto):


    patron = re.compile(

        r"Národnej rady Slovenskej republiky",

        flags=re.IGNORECASE

    )


    m = patron.search(texto)


    if m:

        return texto[m.start():]


    return texto
```

This function removes the preliminary material appearing before the relevant parliamentary proceedings.

In [ ]:

```
# 22. Obtaining the speakers from a document
def obtener_oradores_documento(texto):


    oradores = []


    for m in PATRON_ORADOR.finditer(texto):


        oradores.append({


            "inicio": m.start(),


            "nombre": limpiar_nombre_orador(

                m.group(1)

            ),
```

```
            "partido_diario": ""

        })

    return oradores
```

This function searches the complete document for all occurrences of the speaker pattern. For each match, it records the position at which the speaker marker begins. The speaker name is extracted from the match and cleaned with limpiar_nombre_orador(). A partido_diario field is also created, although it is left empty in this script. The resulting list provides the speaker positions and names required for intervention segmentation.

In [ ]:

```
# 23. Segmenting parliamentary interventions
def obtener_intervenciones(texto):

    oradores = obtener_oradores_documento(texto)

    intervenciones = []

    for i, orador in enumerate(oradores):

        inicio = orador["inicio"]

        if i + 1 < len(oradores):
            fin = oradores[i + 1]["inicio"]
        else:
            fin = len(texto)

        intervencion = texto[inicio:fin].strip()

        intervenciones.append({

            "texto": intervencion,

            "orador": orador

        })

    return intervenciones
```

This function divides the parliamentary document into individual interventions. It first obtains the detected speakers and their positions. For each speaker, the beginning of the intervention is the position of that speaker's marker. The end of the

intervention is defined as the position where the next speaker begins. For the final speaker, the intervention continues until the end of the document. Each segment is stored together with its corresponding speaker. This establishes the individual parliamentary intervention as the textual unit of analysis for the subsequent keyword-classification procedure.

In [ ]:

```
# 24. Extracting the date
# ========================================
# EXTRAER FECHA
# ========================================


def extraer_fecha(texto):


    patrones_fecha = [


        r"(\d{1,2}\.\d{1,2}\.\d{4})",



r"(\d{1,2}\.\s*(januára|februára|marca|apríla|mája|júna|júla|augusta|septembra|októbra|novembra|decembra)\s*(\
d{4}))"


    ]


    for patron in patrones_fecha:


        match = re.search(
            patron,
            texto,
            flags=re.IGNORECASE
        )


        if match:
            return match.group(0)


    return "Fecha desconocida"
```

This function extracts the date from the parliamentary document. If no date is detected, the function returns Fecha desconocida.

In [ ]:

```
# 25. Extracting the session number
# ========================================
# EXTRAER NÚMERO SESIÓN
# ========================================


def extraer_numero_sesion(texto):


    patron = r"(\d+)\.\s*schôdze"
```

```
    match = re.search(
        patron,
        texto,
        flags=re.IGNORECASE
    )

    if match:
        return match.group(1)

    return "Desconocido"
```

This function extracts the number of the parliamentary session. The regular expression searches for a number followed by the Slovak term schôdze. If a match is found, the numerical component is returned. If no session number is identified, the function returns Desconocido.

In [ ]:

```
# 26. Determining the legislature from the date
# ==============================================
# OBTENER INFORMACIÓN DEL ORADOR
# ==============================================

from datetime import datetime

def obtener_legislatura_desde_fecha(fecha):

    meses = {

        "januára": 1,

        "februára": 2,

        "marca": 3,

        "apríla": 4,

        "mája": 5,

        "júna": 6,

        "júla": 7,
```

```
        "augusta": 8,

        "septembra": 9,

        "októbra": 10,

        "novembra": 11,

        "decembra": 12

    }

    m = re.match(r"(\d{1,2})\.\s*([^\s]+)\s*(\d{4})", fecha)

    if not m:
        return ""

    dia = int(m.group(1))
    mes_texto = m.group(2).lower()
    mes_texto = mes_texto.rstrip(".")
    anio = int(m.group(3))

    if mes_texto.isdigit():
        mes = int(mes_texto)
    else:
        mes = meses[mes_texto]

    fecha = datetime(anio, mes, dia)

    if datetime(2016, 3, 23) <= fecha <= datetime(2020, 2, 29):
        return "7"

    elif datetime(2020, 3, 1) <= fecha <= datetime(2023, 9, 30):
        return "8"

    elif datetime(2023, 10, 1) <= fecha <= datetime(2025, 12, 31):
        return "9"

    return ""
```

This block determines the Slovak parliamentary legislature associated with a document based on its date. First, the dictionary meses maps Slovak month names to their numerical values. The date is then parsed into day, month and year. If the date cannot be parsed, the function returns an empty string. The month can be represented either numerically or through its Slovak name. The resulting date is converted into a datetime object.

In [ ]:

```
# 27. Completing speaker information
def completar_info_orador(orador, fecha):


    legislatura = obtener_legislatura_desde_fecha(fecha)


    partido, ideologia = buscar_diputado(

        orador["nombre"],

        legislatura

    )


    return {


        "nombre": orador["nombre"],


        "partido_diario": orador["partido_diario"],


        "partido": partido,


        "ideologia": ideologia


    }
```

This function enriches the detected speaker with information from the MPs' reference database. The legislature is first determined from the document date through obtener_legislatura_desde_fecha(). The speaker's name and the resulting legislature are then passed to buscar_diputado().

In [ ]:

```
# 28. Processing an individual intervention
# ========================================
# x PROCESAR DOCUMENTOS
# ========================================


def procesar_intervencion(


    intervencion,


    info_orador,


    doc_id,
```

```
    doc_origen,
    legislatura,
    ruta_docx,
    fecha,
    numero_sesion
):
    resultados = analizar_intervencion(intervencion)
    filas = []
    for teoria, resultado in resultados.items():
        if not resultado["cumple"]:
            continue
        filas.append([
            doc_id,
            doc_origen,
            legislatura,
            ruta_docx,
            fecha,
            numero_sesion,
            info_orador["nombre"],
            info_orador["partido_diario"],
```

```
            info_orador["partido"],

            info_orador["ideologia"],

            teoria,

            resultado["n_keywords"],

            resumir_keywords(resultado["keywords"]),

            intervencion

        ])

    return filas
```

This function converts a classified parliamentary intervention into the rows that will form the final analytical dataset. First, the intervention is analyzed using analizar_intervencion(). The function then examines the result for each theoretical category. If the intervention does not satisfy the keyword criterion for a theory, that theory is skipped.

In [ ]:

```
# 29. Processing the complete corpus
def process_documents(data):

    results = []

    print("Procesando documentos...")

    for i, item in enumerate(data, start=1):

        print(f"📄 Procesando documento {i}/{len(data)}: {item['id']}")

        text_original = item["texto"]
        text_original = eliminar_portada_indice(text_original)

        doc_id = item["id"]
        doc_origen = item["doc_origen"]
        legislatura = item["legislatura"]
        ruta_docx = item["ruta_docx"]
```

```
        fecha = extraer_fecha(text_original)
        numero_sesion = extraer_numero_sesion(text_original)

        intervenciones = obtener_intervenciones(text_original)
        print(f"Intervenciones encontradas: {len(intervenciones)}")

        for intervencion in intervenciones:

            texto = intervencion["texto"]

            orador = intervencion["orador"]

            info_orador = completar_info_orador(

                orador,

                fecha

            )

            filas = procesar_intervencion(

                texto,

                info_orador,

                doc_id,

                doc_origen,

                legislatura,

                ruta_docx,

                fecha,

                numero_sesion

            )
```

```
            results.extend(filas)

    print(f"Total menciones encontradas: {len(results)}")

    return results
```

This function coordinates the complete processing pipeline.

In [ ]:

```
# 30. Starting the execution section

print("LLEGA AL EJECUTAR")
```

This block marks the beginning of the execution stage of the script. The print() statement is a progress indicator confirming that the script has reached the point where the previously defined functions are executed.

In [ ]:

```
# 31. Selecting the parliamentary period

carpeta = "2016-2020"


ruta_carpeta = os.path.join(

    folder_path,

    carpeta

)


print(f"PROCESANDO CARPETA {carpeta}")

print(f"Carpeta: {ruta_carpeta}")


data = read_doc_files(ruta_carpeta)


# PARA PRUEBAS: 1 DOC DE CADA CARPETA

#data = read_doc_files(folder_path)


detected_mentions = process_documents(data)
```

This block selects the parliamentary period to be processed and launches the analysis.The script combines the general Slovak corpus path with this folder to construct the complete path. read_doc_files() then loads the DOCX documents contained in that period. The commented line provides an alternative testing configuration using the broader root directory. Finally, process_documents(data) executes the complete corpus-processing and classification pipeline. The resulting analytical observations are stored in detected_mentions.

In [ ]:

```
# 32. Creating the output directory
os.makedirs(output_folder, exist_ok=True)
```

In [ ]:

```
This block creates the directory in which the final analytical CSV will be stored.
```

```
The exist_ok=True parameter means that the command does not generate an error if the directory already exists.
```

In [ ]:

```
# 33. Defining the output file
output_file = os.path.join(

    output_folder,

    #f"menciones_prueba.csv"

    f"menciones_{carpeta}.csv"

)
```

This block constructs the path and filename of the final analytical CSV. The active filename is generated dynamically using the value of carpeta.

In [ ]:

```
# 34. Defining the final analytical variables
columnas = [

    "ID",

    "DOC_Source",

    "Legislature",

    "DOC_Path",

    "Date",

    "Session_Number",

    "Speaker",

    "Party_Diary",

    "Party",

    "Ideology",

    "Theory",

    "N_Keywords",

    "Keywords",

    "Intervention"

]
```

This block defines the structure of the final analytical dataset. The 14 variables combine document-level metadata, parliamentary information, speaker characteristics and the results of the theoretical classification.

In [ ]:

```
# 35. Creating the final DataFrame
df_resultados = pd.DataFrame(

    detected_mentions,

    columns=columnas

)
```

This block converts the analytical observations stored in detected_mentions into a pandas DataFrame. The columnas list defines the names and order of the variables. At this point, the final analytical dataset exists in memory as df_resultados.

In [ ]:

```
# 36. Exporting the final analytical dataset
df_resultados.to_csv(
```

```
    output_file,
    sep=";",
    quoting=csv.QUOTE_ALL,
    index=False,
    encoding="utf-8-sig"
)
```

This block writes the final analytical DataFrame to the CSV file.

In [ ]:

```
# 37. Confirming that the file was saved
print(f"📁 Archivo guardado: {output_file}")
```

This line prints the location of the newly created analytical CSV. It provides a simple confirmation that the export operation has completed.

In [ ]:

```
# 38. Reloading the exported dataset
df = pd.read_csv(
    output_file,
    delimiter=";"
)
```

This block reads the newly exported CSV back into pandas. The delimiter is explicitly set to ;, corresponding to the separator used during export. This provides a basic verification that the generated CSV can be reopened using the expected format.

In [ ]:

```
# 39. Checking the first rows of the dataset
print(df.head())
```

This line displays the first rows of the reloaded dataset. It provides a quick visual check of the resulting analytical records and allows the researcher to verify that the output contains the expected structure.

In [ ]:

```
# 40. Confirming completion of the parliamentary period
#print(f"✅ Prueba finalizada.\n")

print(f"✅ Legislatura {carpeta} finalizada.\n")
```

The first line is commented out and therefore is not executed. The active line confirms that the parliamentary period stored in carpeta has been fully processed. With the current configuration, this corresponds to the 2016-2020 period.

In [ ]:

```
# 41. Deleting temporary objects
del data

del detected_mentions

del df_resultados

del df
```

This final block removes the temporary objects created during the analysis from the current Python session.