

# PYTHON GUIDE PROTOCOL

January 16, 2026

Irene Baena Álvarez & Francisco Castillo Eslava

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.

## Contents

<b>1</b>	<b>PYTHON GUIDE PROTOCOL</b>	<b>2</b>
1.1	WEB SCRAPPING GUIDE FOR SESSION DIARIES . . . . .	2
1.1.1	DIARIES OF THE CONGRESS OF DEPUTIES: SPAIN 1931-1977 . . . . .	2
1.1.2	DIARIES OF THE CONGRESS OF DEPUTIES: SPAIN 1977 - Present . . . . .	5
1.1.3	DIARIES OF THE CONGRESS OF DEPUTIES: Czech Republic . . . . .	9
1.1.4	DIARIES OF THE EUROPEAN PARLIAMENT . . . . .	13
1.1.5	EXTRACTION OF JSON FILES . . . . .	16
1.1.6	READING JSON FILES . . . . .	18
1.2	CONSTRUCTION OF THE DATABASE . . . . .	20
1.2.1	DATABASE WITH JSON FILES . . . . .	20
1.2.2	DATABASE WITH PDF FILES . . . . .	24
1.3	FINAL DATABASES . . . . .	29
1.3.1	COVID DATABASE . . . . .	29
1.3.2	POWER OUTAGE DATABASE (“APAGÓN DE ABRIL”) . . . . .	37
1.3.3	POST-FRANCOISM DATABASE . . . . .	43
1.3.4	SECOND SPANISH REPUBLIC DATABASE . . . . .	52
1.3.5	FRANCOISM DATABASE . . . . .	63

# 1 PYTHON GUIDE PROTOCOL

## 1.1 WEB SCRAPING GUIDE FOR SESSION DIARIES

### 1.1.1 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/](https://app.congreso.es/est_sesiones/)), specifically from the historical parliamentary sessions section.

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

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```

[ ]: # 10. Close the browser
    driver.quit()

```

Finally, close the browser and terminate the script.

### 1.1.2 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.

```

[ ]: # 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*).

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```

[ ]: # 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.

```

[ ]: # 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.

```

[ ]: # 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.

```

[ ]: # 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_siguiete = WebDriverWait(driver, 10).until(
        EC.element_to_be_clickable((By.XPATH, '//a[@class="page-link btn_pag"
↪and text()=">"]'))
    )
    print(" Clic en 'Siguiete'")
    boton_siguiete.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 'Siguiete': {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.

```

[ ]: # 10. Close the browser
driver.quit()

```

### 1.1.3 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.

```

[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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 `sanitize_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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```

[ ]: # 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.

```

[ ]: # 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()`.

```

[ ]: # 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.

#### 1.1.4 DIARIES OF THE EUROPEAN PARLIAMENT

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```

[ ]: # 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.

```

[ ]: # 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.

```

[ ]: # 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.

```

[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

### 1.1.5 EXTRACTION OF JSON FILES

```

[ ]: # 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 ORCID IDs and affiliations, dataset name, publication date, and access URL.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

### 1.1.6 READING JSON FILES

```
[ ]: # 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`.

```
[ ]: # 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*.

```
[ ]: # 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', ...]

```
[ ]: # 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.

## 1.2 CONSTRUCTION OF THE DATABASE

### 1.2.1 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```

[ ]: # 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.

```

[ ]: # 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*).

```

[ ]: # 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”.

```

[ ]: # 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.

```

[ ]: # 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.

```

[ ]: # 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.

```
[ ]: # 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).

```
[ ]: # 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.

```
[ ]: # 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.

### 1.2.2 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.

```
[ ]: # 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.

```
[ ]: # 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 documentos correspondientes a las legislaturas queremos analizar.

```
[ ]: # 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.

```
[ ]: # 4. Other patterns for extracting metadata
orador_pattern = re.compile(r"(?:El señor|La señora)\s+([A-ZÁÉÍÓÚÑ]+(?:\s+[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).

```
[ ]: # 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').

```
[ ]: # 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)
    return match.group(1) if match else "Desconocido"
```

```

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.

```

[ ]: # 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
                except:
                    first_page_text = ""
                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.

```

[ ]: # 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.

```

[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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 Explícitas, 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()

```

## 1.3 FINAL DATABASES

### 1.3.1 COVID DATABASE

```

[ ]: # 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.

```

[ ]: # 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.

```

[ ]: 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.

```

[ ]: # 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).

```

[ ]: # 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.

```

[ ]: # 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.

```

[ ]: # 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”).

```

[ ]: # 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.

```
[ ]: # 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"engañososa(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+demonstró|no\s+es\s+cierto|no\s+
    ]

```

```

if any(re.search(p, context_lower) for p in refutacion_patrones):
    return "C"
if tiene_mentalidad_conspirativa(context):
    return "B"
mencion_teorias_patrones = [
    r"\b(nuevo\s+orden\s+mundial|plandemia|élite(s)?
↳|gran\s+reseteo|agenda\s+2030|microchip(s)?
↳|dictadura\s+sanitaria|teoría\s+conspirativa|teoría\s+de\s+la\s+conspiración|conspiración\s
↳\s+mundial(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+compr
]
if any(re.search(p, context_lower) for p in mencion_teorias_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.

```

[ ]: # 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", "Implicita"
    ])
    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.

```

[ ]: # 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.

```

[ ]: # 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.

### 1.3.2 POWER OUTAGE DATABASE (“APAGÓN DE ABRIL”)

```

[ ]: # 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).

```
[ ]: # 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.

```
[ ]: 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.

```
[ ]: # 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).

```
[ ]: # 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.

```

[ ]: # 4. Patterns of mention
apagon_patterns = [
    r"\bapag[oo]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[oo]n de suministro\b",
    r"\bdesconexi[oo]n el[eé]ctrica\b",
    r"\bfalta de energ[ií]a\b",
    r"\bfalla de luz\b",
    r"\bcorte de luz\b",
    r"\bapag[oo]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[oo]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[oo]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.

```

[ ]: # 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”.

```

[ ]: # 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”.

```

[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

```
[ ]: # 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.

### 1.3.3 POST-FRANCOISM DATABASE

```
[ ]: # 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*).

```
[ ]: # 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.

```
[ ]: 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.

```
[ ]: # 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).

```
[ ]: # 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.

```

[ ]: # 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]?|conspiración[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.

```

[ ]: # 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”.

```

[ ]: # 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.

```

[ ]: # 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+demonstró|no\s+es\s+cierto|no\s+
]
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+sanitaria|teoría\s+conspirativa|teoría\s+de\s+la\s+conspiración|conspiración\s
↳\s+mundial(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+compr
]
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.

```

[ ]: # 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_expression(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_implicit
    ])

    # 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.

```

[ ]: # 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).

```

[ ]: # 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.

### 1.3.4 SECOND SPANISH REPUBLIC DATABASE

```
[ ]: # 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*).

```
[ ]: # 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.

```
[ ]: ## 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.

```

[ ]: # 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",
]

```

```

[ ]: List of routes where the PDFs are located

```

```

[ ]: # 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)", # yyyyymmdd
    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]: # yyyyymmdd
            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 yyyyymmdd
    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.

```

[ ]: # 4. Reading 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.

```

[ ]: # 5. List of patterns of conspiracy theories
GUION_GUARD = r"(?![-\u2010-\u2015])" # -, -, , -, -

conspiracy_patterns = [
    rf"{GUION_GUARD}\bjud[ii](?:o|a|os|as)\b", # Judío/a/os/as
    rf"{GUION_GUARD}\bjuda[ii](?:smo|co)\b", # Judaísmo / Judaico
    rf"{GUION_GUARD}\bcontubernio(?:s)?\b", # Contubernio(s)
    rf"{GUION_GUARD}\b(?:complot|compl[oó])\b", # Complot / Complotó

```

```

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.

```

[ ]: # 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

```

[ ]: # 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"engañoso(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
        ↵
↪r"\b(no\s+hay\s+(pruebas|evidencia|fundamento)|nunca\s+se\s+demostr[oó]|no\s+es\s+cierto|no
    ]
    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_teoría_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_teoría_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[á]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_expression`: determines whether the mention is Explicit, Implicit or Not Applicable according to the markers present.

```

[ ]: # 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_expression(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.

```
[ ]: # 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).

```
[ ]: # 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

### 1.3.5 FRANCOISM DATABASE

```
[ ]: # 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.

```
[ ]: 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.

```

[ ]: 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.

```
[ ]: ## 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.

```
[ ]: # 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).

```
[ ]: # 3. Reading and preprocessing 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.

```
[ ]: 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.

```

[ ]: 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.

```
[ ]: 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*.

```
[ ]: ## Robust pre-processing of extracted text
LIGATURAS = {
    " ": "fi", " ": "fl", " ": "ff", " ": "ffi", " ": "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[\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-zÁÉÍÓÚŃáéíóúñ])\r?\n([a-záéíóúñ])", 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[\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-zÁÉÍÓÚÑáéíóúñ])\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.

```

[ ]: 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-zÁÉÍÓÚáéíóúñ]+)\s+(\d{1,2})\s+de\s+([A-Za-zÁÉÍÓÚáéíóúñ]+)\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_ddmmyyy(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.

```

[ ]: 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-ZÁÉÍÓÚÛÑ][A-ZÁÉÍÓÚÛÑ' \-]{1,80})
    [ \t]*:\s*$"
)

EPOCA_TITULARES = re.compile(
    r"(?im)^[ \t]*(?:SU[ \t]+EXCELENCIA|S\.[ \t]*E\.[ \t]+PRESIDENCIA|EL[ \t]+
    ↪[ \t]+PRESIDENTE|LA[ \t]+PRESIDENCIA|
    r"EL[ \t]+SEÑOR[ \t]+PRESIDENTE|JEFE[ \t]+DEL[ \t]+ESTADO|EXCMO\.[ \t]+EXCMA\.[ \t]+
    ↪[A-ZÁÉÍÓÚÛÑ' \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.

```

[ ]: # 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.)

```

[ ]: 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", "admission", "transmisi
    ↳
↳"omision", "agresion", "depression", "opresion", "supresion", "suspension", "impresion", "version",
    ↳
↳"perversion", "conversion", "inversion", "diversion", "aversion", "excursion", "provision", "revis
    ↳
↳"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-zÁÉÍÓÚÑáéíóúñ]", "",
↳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-zÁÉÍÓÚÑ]", "", 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').

```

[ ]: # 5. Clasificación
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+
    r"|"
    ↪
↪r"\b(no\s+hay\s+(pruebas|evidencia|fundamento)|nunca\s+se\s+demostr[oó]|no\s+es\s+cierto|no
    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"engañososa(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.

```

[ ]: # 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-zÁÉÍÓÚÛÑáéíóúûñ]+", 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.

```

[ ]: # 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).

```
[ ]: # 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_expression(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_implicit
    ])
    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.

```

[ ]: # 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.

```

[ ]: # 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.