Step 1:

We implemented the extract_metadata function. This function is designed to process each filename and extract the Knesset Number (XX) and Protocol Type (ptm or ptv).

The **Knesset Number**, which represents the session of the Knesset, is extracted as the first part of the filename and converted into an integer for consistency and numerical operations. The **Protocol Type**, which indicates whether the file pertains to a plenary or a committee protocol, is extracted as the second part of the filename.

Then, we mapped the protocol type to a meaningful label using a dictionary:

"ptm" was mapped to "plenary" and "ptv" was mapped to "committee" as required.

If the protocol type does not match these predefined mappings, the function assigns the label "unknown" to ensure robustness when handling unexpected formats.

```python
def extract_metadata(file_name):
    # Remove file extension and split by underscores
    base_name = os.path.splitext(file_name)[0]
    parts = base_name.split("_")

    # Extract knesset number and protocol type
    knesset_number = int(parts[0])
    protocol_type = parts[1]

    # Map protocol type to a meaningful name
    protocol_type_mapping = {
        "ptm": "plenary",
        "ptv": "committee"
    }
    protocol_type = protocol_type_mapping.get(protocol_type, "unknown")
    return knesset_number, protocol_type
```

Step 2:

We implemented the extract_protocol_number function. This function processes the text of each document to locate and extract the protocol number in one of the following formats:

- **Numeric Format:** The number often appears after the text "פרוטוקול מס'".
- **Textual Format:** In Hebrew, the number may also be written as text after "הישיבה".

After reviewing most of the files, we observed the following:

In ptm files the protocol number is often written in Hebrew as text. So, to handle this, we manually reviewed all the files and created a mapping list that converts Hebrew textual numbers to their corresponding integers. This list is stored in the **hebrew_to_int** dictionary.

The function first tries to extract a numeric protocol number using regular expressions. If no number is found, it searches for Hebrew textual numbers and translates them using the predefined mapping. If neither approach succeeds, the function returns -1 as required.

Handling Multiple Protocol Numbers:

One issue we handled was the presence of multiple protocol numbers in a single string, like:

פרוטוקול מס׳ 141, 144, 145
מישיבת ועדת הכספים
יום חמישי, כ״ג בחשון התשע״ו (05 בנובמבר 2015), שעה 10:50

Initially, we considered returning all numbers as a list. However, as per the requirements, the function needed to return a single integer. To address this:

The function identifies the list of numbers. Then, It extracts and returns only the first number.

```python
def extract_protocol_number(text):
    # Match numerical protocol numbers (e.g., "45 'פרוטוקול מס")
    protocol_number_match = re.search(r'פרוטוקול מס\'\s*(\d+(?:,\s*\d+)*)', text)
    if protocol_number_match:
        numbers = protocol_number_match.group(1).split(",")  # Split numbers by comma
        for number in numbers:
            try:
                return int(number.strip())  # Return the first valid number
            except ValueError:
                pass
    # Match Hebrew text-based session numbers (e.g., "הישיבה המאה וחמישים ושש של הכנסת")
    hebrew_session_match = re.search(r'הישיבה\s+(.*?)\s+של\s+הכנסת', text)
    if hebrew_session_match:
        hebrew_number_text = hebrew_session_match.group(1).replace("-", " ")
        if hebrew_number_text in hebrew_to_int:
            return hebrew_to_int[hebrew_number_text]
    # If no valid number is found, return -1
    return -1
```

Step 3:

We observed that speaker names often appear followed by a colon (:) in the text. This consistent pattern was used to identify the names of speakers in the protocol. To validate this pattern, we conducted a **Ctrl+F** search for colons (:) within the protocol files and discovered an exceptional situation where a colon was present, but it did not indicate a speaker name.

To address this issue, we identified specific words that commonly appear before a colon but are not speaker names. These words were compiled into a list called **EXCEPTION_WORDS**. This list includes terms such as "משתתפים", "סדר היום", "מוזמנים", and other administrative or structural terms. By cross-referencing the text preceding colons with this list, we ensured that non-speaker cases were excluded from the analysis.

Once a valid speaker name was identified, all subsequent text was attributed to that speaker until a new speaker was detected. This approach allowed us to handle scenarios where titles, remarks, or other text fragments appear in the middle of the protocol. Such text fragments were treated as part of the last speaker's speech to maintain continuity unless a new speaker was explicitly identified.

How We Implemented this task?

Here is the explanation!

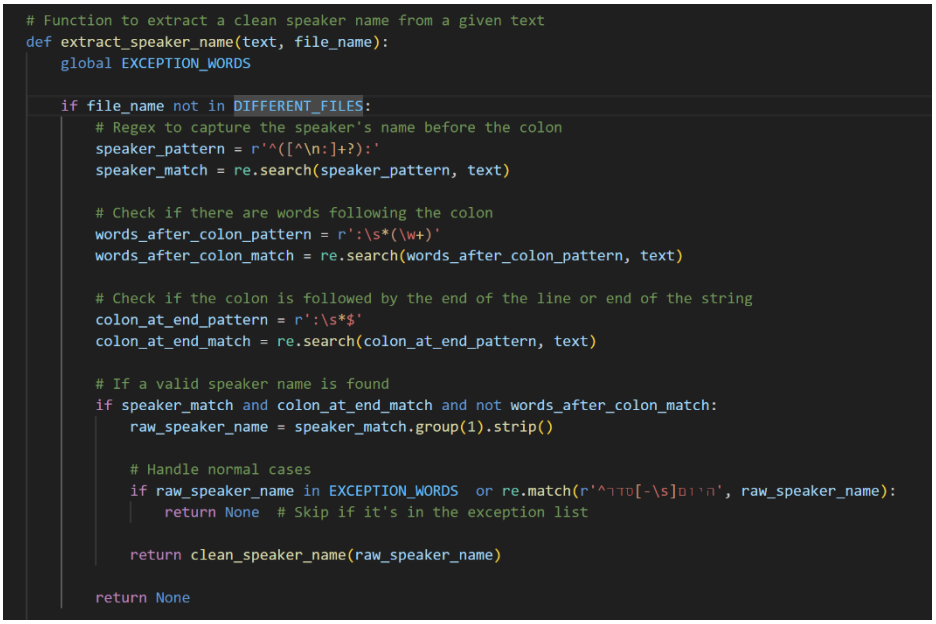
We used the Document class from the python-docx library to open and iterate through all paragraphs in the .docx protocol files.

```
doc = Document(doc_path)

for paragraph in doc.paragraphs:

        paragraph_text = paragraph.text.strip()
```

For each paragraph, we called the function `extract_speaker_name` to detect and validate the speaker's name based on a pattern (e.g., a colon : following the speaker's name). After identifying the potential name, we called the `clean_speaker_name` function to refine the name by removing unnecessary titles, prefixes, and suffixes. This ensured only meaningful speaker names were retained.



```python
# Function to extract a clean speaker name from a given text
def extract_speaker_name(text, file_name):
    global EXCEPTION_WORDS

    if file_name not in DIFFERENT_FILES:
        # Regex to capture the speaker's name before the colon
        speaker_pattern = r'^([^\n:]+?):'
        speaker_match = re.search(speaker_pattern, text)

        # Check if there are words following the colon
        words_after_colon_pattern = r':\s*(\w+)'
        words_after_colon_match = re.search(words_after_colon_pattern, text)

        # Check if the colon is followed by the end of the line or end of the string
        colon_at_end_pattern = r':\s*$'
        colon_at_end_match = re.search(colon_at_end_pattern, text)

        # If a valid speaker name is found
        if speaker_match and colon_at_end_match and not words_after_colon_match:
            raw_speaker_name = speaker_match.group(1).strip()

            # Handle normal cases
            if raw_speaker_name in EXCEPTION_WORDS  or re.match(r'^עדה[-\s]היום', raw_speaker_name):
                return None  # Skip if it's in the exception list

            return clean_speaker_name(raw_speaker_name)

        return None
```

* After running the code, we encountered exceptional files that did not follow the same structure as the others. These files required special handling. We define them at a **DIFFERENT_FILES** list.
the continue of this implementation is to deal with them.

We did our best to filter the names:

To address this, we created a JSON file containing all detected speaker names, ensuring each name appeared only once. This allowed us to manually verify and confirm that the names were properly cleaned and validated.

These are the popular words or patterns that we used to clean and filter the names:

```
# Remove certain patterns (titles, prefixes, and suffixes)
patterns_to_remove = [
    r'\s*סגנית מזכיר הכנסת\s+', r'\s*סגן מזכיר הכנסת\s+', r'\s*סגן מזכירת הכנסת\s+',
    r'\s*סגנית מזכירת הכנסת\s+', r'\s*מזכיר הכנסת\s+', r'\s*מזכירת הכנסת\s+',
    r'\s*סגנית שר במשרד ראש הממשלה\s+', r'\s*סגן שר במשרד ראש הממשלה\s+',
    r'\s*(סגן|סגנית)\s+(ה\w+\s+וה\w+)\s*',
    r'\s*(סגן|סגנית)\s+(ה\w+\s+)\s*', r'\s*סגן\s+', r'\s*סגנית\s+', r'(?:שרת?\s+ה\w+,\s+ה\w+\s+וה\w+\s+).*',
    r'\s*שר(?:ת)?\s+ה\w+\s+וה\w+\s*', r'\s*שר(?:ת)?\s+ל\w+\s+\w+\s+',
    r'\s*שר(?:ת)?\s+ל\w+\s+[^,]\s', r'\s*סגן שר(?:ת)?(?:\s+[^,])?,', r'\s*שר(?:ת)?(?:\s+[^,])?,'
]
```

```
# Remove specific predefined titles and phrases
phrases_to_remove = [
    '<< יור >>', '<< דובר >>', '(יש עתיד-תל"מ)', '<< אורח >>',
    '<< קבוע_במשך >>', '<< קריאה >>', '<', '>', 'מ"מ היו"ר', 'היו"ר','היו”ר','היו”ר ',
    'השר', 'לביטחון פנים', 'ראש הממשלה', 'הלאומיות', 'פרופ',
    ' מר ', 'ופיתוח הכפר', 'תשובת','יו"ר', 'לאיכות הסביבה', ' היו"ר', 'עו"ד', 'נצ"מ',
    'ד"ר'
]
```

```
# Global list of exception words
EXCEPTION_WORDS = [
   'משתתפים', 'מנהלות הוועדה', 'מנהל הוועדה', 'מוזמנים באמצעים מקוונים','משרד האוצר',
   'משתתפים באמצעים מקוונים', 'נכחו', 'מנהלת הוועדה', 'מסקנות הוועדה', 'משתתפים',
   'מוזמנים', 'סדר היום', 'חברי הוועדה', 'חברי הכנסת', 'ייעוץ משפטי',
   'מנהל/ת הוועדה', 'רצוני לשאול', 'קריאות', 'קריאה', 'האפשרות', 'ביום',
   'ברצוני לשאול', 'רישום פרלמנטרי','נוכחים','משרד המשפטים','יועץ משפטי',
   'סדר-היום','רשמת פרלמנטרית', 'קצרנית','מנהלי הוועדה','חברי כנסת','רשמה וערכה'
]
```

<u>Challenges in Handling Names Before and After Cleaning:</u>

One of the key challenges in handling names lies in ensuring consistency across different protocols. Variations in how a name is presented can make it difficult to determine if two references point to the same individual. For instance, someone might appear as "ראש הממשלה ביבי" in one protocol and just "ביבי" in another. Similarly, abbreviations like "ב.נתניהו" "בנימין נתניהו" or even a nickname like "ביבי" can all refer to the same person but may be treated as separate entities without proper cleaning.

Before Cleaning:
Without name cleaning, the data might include redundancies and inconsistencies due to titles, prefixes, or contextual variations. This creates challenges in identifying speakers and linking their speeches correctly, especially when individuals' roles change over time. For example, a person might be a minister one year and not the next, but references to their name with the title "שר" might persist in some contexts.

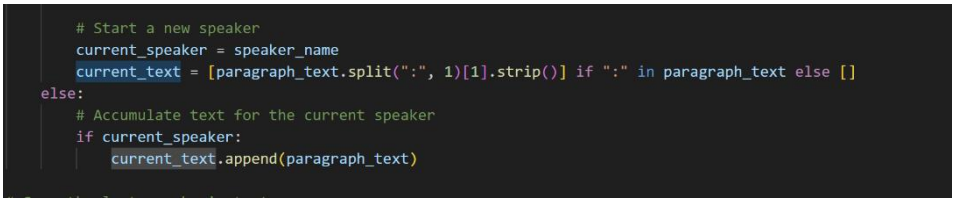
After Cleaning:
Although the cleaning process improves consistency, it introduces its own challenges. Over-aggressive cleaning might strip essential parts of the name or fail to recognize subtle variations. For example, while cleaning, names like "בנימין נתניהו" versus "נתניהו" or "ביבי" may still remain distinct unless additional techniques, such as advanced name-matching algorithms or metadata integration, are employed.

While name cleaning significantly enhances the reliability of the data by standardizing references, it is not a perfect solution.

After obtaining the filtered and validated speaker names, we returned to the paragraphs and processed them to associate the text with the corresponding speakers:

if a valid speaker name was found, the speaker was set as the current_speaker.

When a current_speaker was identified, The text following the colon (:) was stored as the speaker's initial speech. If subsequent paragraphs did not contain a new speaker, they were appended to the current_speaker's speech.
When a new speaker was detected, the previous speaker's text and metadata were saved in a structured format.



```
        # Start a new speaker
        current_speaker = speaker_name
        current_text = [paragraph_text.split(":", 1)[1].strip()] if ":" in paragraph_text else []
    else:
        # Accumulate text for the current speaker
        if current_speaker:
            current_text.append(paragraph_text)
```

Step 4:

To breaking down the text into phrases, We implemented the `break_text_into_phrases` function.

We define that this punctuation : {'.', '!', '?'} will be our separators.

This is because these punctuation indicate natural breaks in speech and are the primary markers for dividing the text.

Handling Special Cases:

1. Numbers and dates, such as "3.14" or "25.12.2024", often contain periods that should not be treated as sentence boundaries, to address this, our function checks the characters immediately before and after a separator. If both are digits, the separator is considered part of the number or date rather than a sentence break.

```
for i, symbol in enumerate(input_text):
    if symbol in separators:
        # Check if the separator is part of a number or date
        if i > 0 and i < len(input_text) - 1:
            prev_char = input_text[i - 1]
            next_char = input_text[i + 1]

            if prev_char.isdigit() and next_char.isdigit():
                buffer += symbol  # Treat as part of a number/date
                continue
```

2. To handle numbered lists or enumerations where the period should not split the text into separate sentences.

```
# Check if the period is part of a numbered list
if i > 0 and input_text[i - 1].isdigit() and input_text[i - 2] == ' ' and (i < len(input_text) - 1 and input_text[i + 1] == ' '):
    buffer += symbol  # Treat as part of the enumeration
    continue
```

The buffer variable accumulates characters until a sentence boundary is reached. Once a separator is detected ,the sentence is added to the phrases list, and the buffer is reset for the next phrase.

```
        # Add the sentence to the list
        buffer += symbol
        if buffer.strip():
            phrases.append(buffer.strip())
        buffer = ''
    else:
        buffer += symbol
```

Step 5:

To cleaning sentences and ensuring their validity as Hebrew-only sentences, We implemented the `clean_sentence` function.

The function:

1. Ensures sentences consist of valid Hebrew words.
2. Removes invalid patterns like dashes (---, --).
3. Strips English words using regex and cleans unnecessary whitespace.
4. Verifies sentences contain Hebrew characters; sentences without them are discarded.

This approach ensures only meaningful and properly formatted Hebrew sentences are retained.

Step 6:

We implemented a function to tokenize sentences by splitting them based on whitespace, ensuring each word or meaningful unit becomes a separate token. The function also ensures the text remains meaningful and handles specific details as required. Here's how it works:

- **Preserves Hebrew abbreviations:** Abbreviations such as "א.ב" are treated as single tokens to maintain their meaning.
- **Maintains dates and numeric values:** Dates (e.g., "12.11.2023") and numbers (e.g., "12.5") are handled as individual tokens to preserve their context.
- **Handles punctuation intelligently:** Punctuation marks (e.g., ".", ",", "!") are treated as separate tokens unless they are part of an abbreviation or numeric value.
- **Keeps Hebrew prefixes intact:** Prefixes like "מ-" and "ו-" remain attached to their words, ensuring the linguistic structure is retained.

This approach ensures that the tokenized output is consistent, linguistically meaningful, and avoids unnecessary segmentation, aligning directly with the requirements of the task.

Step 7:

We add a condition to check if len(tokens) >= 4, ensuring only sentences with 4 or more tokens are processed.

This ensures that only meaningful and complete sentences are included, avoiding fragments or short phrases that might lack context.

```
tokens = get_tokens(clean_sentence_text)  # Tokenize the cleaned sentence
if len(tokens) >= 4:  # Only add if there are 4 or more tokens
    data_list.append({
        "protocol_name": file_name,
        "knesset_number": knesset_number,
        "protocol_type": protocol_type,
        "protocol_number": protocol_number,
        "speaker_name": current_speaker,
        "sentence_text": " ".join(tokens)  # Combine tokens into a string
    })
```

Step 8:

We append all the data that are required at the data_list :

```
data_list.append({
    "protocol_name": file_name,
    "knesset_number": knesset_number,
    "protocol_type": protocol_type,
    "protocol_number": protocol_number,
    "speaker_name": current_speaker,
    "sentence_text": clean_sentence_text  # Cleaned sentence
})
```

To address this task of saving the extracted and processed data in a **JSONL format** (where each JSON object is written as a separate line in the file), We implemented the save_data_to_json function. And call it with our data_list

```
def save_data_to_json(data_list, output_path):
    with open(output_path, "w", encoding="utf-8") as json_file:
        for data in data_list:
            json_file.write(json.dumps(data, ensure_ascii=False) + "\n")
```