NLP-HW-1

עבאס אסמאעיל – 214742025

סלאם קייס-327876116

Part 1:

1- To collect the necessary data we begin with reading the names of the files from the given directory and split it accordingly using this function:

```
def get_docx(folder_path):
    try:

        info =[]
        # go thru all the documants in the directory
        current_path =os.path.join(os.getcwd(),folder_path)

        for _, filename in enumerate(os.listdir(current_path)):
            if filename.endswith('docx'):

                attributes = filename.split('_')# Split the filename to get the attributes

                if attributes[1] == 'ptv':  # committee or plenary
                        type = 'committee'
                elif attributes[1] == 'ptm':
                        type = 'plenary'
                else:
                    type = '-1'

                text = Document(os.path.join(current_path,filename))

                info_item = {
                        'file_name': filename,
                        'knesset number': int(attributes[0]),
                        'type': type,
                        'text': text,
                        'file_number': attributes[2].replace('.docx', '')
                    }
                info.append(info_item)
        return info
    except Exception as e:
        print(f'Exception in get_docx: {e}')
```

2- This was the first somewhat challenging part of the task. After going through the files I noticed a pattern which is: the protocol number always comes after one of two specific words, we read every line until we find one of these two occurrences, if we cant find any we simply return -1.
These are the words:

```
target_words = ["הישיבה ה", "פרוטוקול מס'"]  # Search for the protocal number
```

We run this loop at the start of reading every file in main:

```
# Find the protocol number
for par in doc['text'].paragraphs:
    text = par.text.strip() # Remove leading and trailing spaces

    if text.startswith('<') or text.startswith('>'): # Sometimes the text starts with < and ends with >, its probably caused by the conversion
        text = text[1:-1]

    if target_words[0] in text:
        position = text.find(target_words[0])
        next_word = get_next_word(text, position + len(target_words[0]))
        #print(f"Found in doc {doc_num}: {text}, NEXT {next_word}") # Debugging
        protocol_number = next_word
        break

    if target_words[1] in text:
        position = text.find(target_words[1])
        next_word = get_next_word(text, position + len(target_words[1]))
        #print(f"Found in doc {doc_num}: {text}, NEXT {next_word}") # Debugging
        protocol_number = next_word
        break
```

If we find any of these words then we run the function to get the following word:

```
# Function to the number after we find "הישיבה ה" or "'פרוטוקול מס"
def get_next_word(text, position):
    # Find the start of the next word
    word_start = position


    while word_start < len(text) and text[word_start].isspace(): # Skip spaces
        word_start += 1

    # If reached the end of the text, return -1
    if word_start >= len(text):
        return '-1'

    # Find the end of the next word
    word_end = word_start
    while word_end < len(text) and not text[word_end].isspace():
        word_end += 1

    # Return the next continuous word
    return text[word_start:word_end]
```

Note that we return a string and we have to turn it into Int, if its already a number then we simply cast it to int:

```
if protocol_number != '-1':
    if protocol_number[-1] == ',' or protocol_number[-1] == '.': # Remove the last character if it's a comma or period
        protocol_number = protocol_number[:-1]
    protocol_int = fix_protocol(protocol_number)
    #print(f"Protocol number: {protocol_int}, string {protocol_number}")
else:
    protocol_int = -1
```

Using this function:

```
def fix_protocol(str):
    # Regular expressions for numbers only and letters only

    digit_pattern = r'^\d+$'
    first = {"עשרי" : 20, "עשר" : 10, "שלושים" : 30, "ארבעים": 40, "חמישים":50, "שישים":60, "שבעים":70, "שמונים":80, "תשעים":90}
    second = {"אח": 1, "שתי":2, "שלוש":3, "ארבע":4, "חמש":5, "שש":6, "שבע":7, "שמונה":8, "תשע":9}

    # Check if the input string matches the digit pattern
    if re.match(digit_pattern, str):
        return int(str)
    num = 0
    splits = str.split('-')
    for i in range(len(splits)):
        enter = True

        if splits[i] == '':
            continue
        if "מאה" in splits[i]:
            num+=100
            continue
        if "מאתי" in splits[i]:
            num+=200
            continue
        if "מאות" in splits[i]:
            num*=100
            continue

        for key in first:
            if key in splits[i]:
                num+=first[key]
                enter = False
                break

        if enter == False:
            continue
        for key in second:
            if key in splits[i]:
                num+=second[key]
    return num
```

We have an expression for the digits and we must check if its digits or not, if it is we simply cast it and return it.

Otherwise we continue We see that the protocol number has a maximum of 3 digits, we have 3 levels of priorities:
First we check 3 specific words in the 3 digits, then 2 digits and finally 1 digit numbers. It must be done in this order otherwise we might have bugs.

Another example of the importance of the order is how 20 is added before 10, since the word for 10 is in the word for 20.

3- Now we need to extract the names and the speech. In both file types we always have the names ending with ":" and the name is underlined. Our goal is to find all these occurrences, then every line after that that isn't underlined and ending with ":" we add it to the current speaker's text.
There are certain cases where the lines aren't a speech, we decided to give it to the current speaker as well.
This is the code we use to find the names:

```
index = text.find(":")
if index>=0 and index== len(text) -1 and is_underlined(par):
```

The underlined function checks if the text is underlined, or if the text style itself is underlined, or any of the base styles.

Some other sentences have a ":" at the end and are underlined, to deal with that issue we have a list of all of these specific cases since there are only a few of them:

```
common_pos = ["סדר-היום", "סדר היום", "נכחו", "חברי", "מנהל", "רישום", "משתתפים", "מוזמנים", "ייעוץ", "יועץ", "קצרנית",
```

We check if the name contains any of these, if they do then we skip this name and search for the next one, if not then we have successfully found a name.

Note that sometimes the text beings with the following tags:

```
tags = ["<< דובר >>", "<< נושא >>", "<< יור >>", "<< דובר_המשך >>", "<< אורח >>", "<< סיום >> " , "<< הפסקה >> " , "<< יור >>"]
```

Which are used to format the files, we remove them and remove anything else:

```
def remove_tags(text, tags):
    # Strip leading/trailing spaces
    try:
        cleaned_text = text.strip()
        for tag in tags:
            # Remove specific tag from start
            if cleaned_text.startswith(tag):
                cleaned_text = cleaned_text[len(tag):].strip()  # Remove the tag and strip spaces

            # Remove specific tag from end
            if cleaned_text.endswith(tag):
                cleaned_text = cleaned_text[:-len(tag)].strip()  # Remove the tag and strip spaces

        return cleaned_text
    except Exception as e:
        print(f'Exception in remove_tags: {e}')
```

After we find a name, we use a dictionary to save the text of every speaker, and a list to order all the speakers, this will be used later to save the data.

After we find a name, we clean it:

```python
def clean_name(name): # clean
    try:
        name = name.strip()
        comps = name.split(' ') #split name to words/comps
        clean_name = ""
        open_parentheses =False
        common_pos = ["ראש", "ממשלה", "יו\"ר", "שי" , "לאומי", "ערבית", "ועדת", "איכות", "פנים", "שר", "אוצר" , "משפטים"  , "\""  ,"אנרגיה"  , "ים

        for comp in comps:
            if comp == '':
                continue
            if any(pos in comp for pos in common_pos): # if the text contains any of the common positions then skip
                continue
            if '(' in comp: # This is a closing parentheses, backwards since its in hebrew
                open_parentheses = False
                continue
            if open_parentheses:
                continue
            if '\'' == comp[-1] and len(comp) <4: #then this a position
                clean_name = '' # if it is then remove what we collected before
                continue

            if ")" in comp: # This is an opening parentheses, backwads since its in hebrew
                if "(" in comp:# if we have closing parentheses then skip it
                    continue
                else:# else we must take the following comps until we see closing parentheses
                    open_parentheses = True

            elif comp == "-" or comp == '־' or comp == '~' or comp ==',':#if the name has a dash then take the first part
                break
            else:
                clean_name += comp+" "

        clean_name = clean_name.strip()
        if ',' in clean_name:
            return ''
        if clean_name != "" and clean_name.find(':')+1 == len(clean_name): # Remove the colon if it's the last character, check if name isnt empty
            clean_name = clean_name[:-1]

        return clean_name.strip()
    except Exception as e:
        print(f'Exception in clear_name: {e}')
```

The idea is to check for common words and positions and remove them, then remove the rest using the loop, everything in parentheses is removed, everything after a dash is removed, anything that is less than 4 letters and ends with ‘ is removed.

This may cause issues if the name is 3 or less letters and ends with ‘, or if the position comes before the name and is separated with a dash, or if the name is in between parentheses.

These cases are highly unlikely, and they are part of the challenge of language processing. The only way to fix every case is to generalize or rearrange the data in a specific format.

Another issue is if the name is used more than once differently for the same person, we don’t have a way of differentiating between a new speaker and a previous speaker, the solution for this is also either generalization or formatting, or saving a of all different names of the same person (practically impossible)

הערה:

ייתכן ותצטרכו להתמודד עם כותרות או טקסטים אחרים שמופיעים באמצע הפרוטוקול. טקסטים אלו אינם משויכים לאף דובר. תוכלו לבחור איך אתם מתמודדים עם טקסטים אלו: למשל, לצרף אותם כטקסט של הדובר האחרון, כטקסט של היו"ר או להתעלם מהם לחלוטין. כתבו בדו"ח את בחירתכם והסבירו.

About this comment, we decided that any text not attributed to a specific speaker would be assigned to the last speaker. We felt that this is good because unattributed text often pertains to the topic that discussed by the last speaker,Also this text might be impotant text in the discustion . Finally detecting text without speaker and delete it can be challenging, So we chose this method .

4- We defined the cases as

```
cases = [' - - -','- - -', ' - -', '- -', ' - - -','- - -', ' - -', '- -', ' - - -', '- - -', ' - -', '- -' ]
```

and for any text containing these cases, we deleted the entire text, This decision was based on the hw instructions , which indicated that any sentence ending with "---" might be truncated. We observed that all the defined cases followed this rule

.While splitting the paragraph we must keep everything that is in quotation marks together without splitting it, everything else is treated.

We split each paragraph into parts separated with spaces and iterate over them in a loop, if were in between quotations we save every part together until the end of the quote, otherwise if the current part ends with any mark, we split it there:

```
def split_paragrph(par):
    try:
        new_sentence = ''
        seperators = ':;!?!؟.'  # use this to check start and end of sentences
        qutoed = False

        cases = [' - - -','- - -', ' - -', '- -', ' - - -','- - -', ' - -', '- -', ' - - -', '- - -', ' - -', '- -' ]
        txt = par.text.strip()
        for case in cases:
            if case in txt:  # Check if the text contains any of the special cases
                txt = txt.replace(txt, '')  # Replace the special case with a space
                break

        par_parts = txt.split(' ')
        sentece_list = []
        for part in par_parts:
            if part == '':  # if empty then skip
                continue

            new_sentence += part + " "  # Collect sentence
            if '"' == part[0]:
            if '"' == part[0]:
                qutoed = True
            if '"' == part[-1] or (len(part) >= 2 and part[-2] == '"' and part[-1] in seperators):
                # if the second to last char is a " and last is a seperator, or if the last is a quote then we end the quote
                qutoed = False

            if part[-1] in seperators or (
                    len(part) >= 2 and part[-2] in seperators):  # If we reached the end of the sentence, save it
                if qutoed == False:  # if were still in quotes, we dont save yet
                    sentece_list.append(new_sentence.strip())
                    new_sentence = ''

        # if we start a quote but it didnt end, save the text
        if qutoed:
            sentece_list.append(new_sentence)
        return sentece_list
    except Exception as e:
        print(f'exception in split_paragrph: {e}')
```

The reason behind keeping the quoted text together is that it doesn't make sense to split the quote into parts, this may cause issues if we have a quote inside of a quote- highly unlikely.

Note that words like 4:50 or 12.5 aren't split.

We include a form of text cleaning that we will talk about it next.

5- Majority of this part was already done when we splitting the paragraphs, we already removed sentence that have one of the case (theses are the sentences that truncated).

```
cases = [' - - -','- - -', ' - -', '- -', ' - - -','- - -', ' - -', '- -', ' - - -', '- - -', ' - -','- -' ]
```

We also use this function to clean the text:

```
def clean_text(txt):
    try:
        if txt == '':
            return ''
        allowed = re.compile('[0-9א-ת"#$%&\'()*+,-./:;<=>?@[\\]_`{|}~־ ]+') # Allowed characters
        occurences = re.findall(allowed, txt) # Find all allowed characters
        if len(occurences) != 1: # If there are more than one occurence that means we have unwated characters in the text or in between the text
            return ''

        filtered_txt = occurences[0] # Get the first and only occurence, we must check if its actually hebrew or not
        heb_txt = False # Check if the text is in hebrew
        heb_letters = ['א', 'ב', 'ג', 'ד', 'ה', 'ו', 'ז', 'ח', 'ט', 'י', 'ך', 'כ', 'ל', 'ם', 'מ', 'ן', 'נ', 'ס', 'ע', 'ף', 'פ', 'ץ', 'צ', 'ק', 'ר', 'ש', 'ת']

        for letter in filtered_txt:
            if letter in heb_letters:
                heb_txt = True # This means that the text is in hebrew since theres one occurence
                break
        if heb_txt == False:
            return ''

        return txt
    except Exception as e:
        print(f'Exception in clean_text: {e}')
```

We find the occurrences of allowed characters (Hebrew letters and punctuation)
If we have one occurrence that means its either fully correct or fully incorrect, we check if it contains any Hebrew letter.

If there isn't only 1 occurrence then we know for sure that the text isn't correct and simply return an empty string, otherwise we return the text itself.

So we delete the sentences that have Eng or truncated

This is another form of cleaning:

```
if text.startswith('<') or text.startswith('>'): # Sometimes the text starts with < and ends with >, its probably caused by the conversion
    text = text[1:-1]
```

When we clean the text in the splitting function, we were able to eliminate hundreds of cases, around 30 cases were left untreated which can be caused by having a different ascii/character in the text that wasn't detected, this issue is difficult to fix since we have to find every character/case and add it to the code.

6+7-

Now for tokenization we being by filtering the array we get from splitting and cleaning by removing empty words. We use this function:

```
def tokenize(list_text):
    try:
        tokens = []
        punctuation = ['!', '"', '\'', '(', ')', ',', '-', '.', '/', ':', ';', '?', '[', '\\', ']', '_', '{', '}', '~']

        for text in list_text:
            words = text.split(' ') # split the text into words
            new_token =[]
            for j in range(len(words)):
                only_punctuation = True # Check if the word is only punctuation
                word = words[j]
                if word == '':
                    continue
                for i in range(len(word)):
                    if only_punctuation == False: # this is done so we save text like 3:00 without spearing the colon
                        break
                    if word[i] in punctuation:
                        new_token.append(word[i])
                        words[j] = words[j][1:] # Remove from the text
                    else:
                        only_punctuation = False

                punctuation_at_end = [] # save extra seperated punctuation marks at the end of the word
                for i in reversed(range(len(word))):
                    if word[i] in punctuation:
                        punctuation_at_end.append(word[i])
                        words[j] = words[j][:-1] # Remove from the text
                    else:
                        new_token.append(words[j])
                        new_token.extend(reversed(punctuation_at_end))
                        break

            if len(new_token) < 4:
                continue
            tokens.append(new_token)
        return tokens

    except Exception as e:
        print(f'Exception in tokenize: {e}')
```

We use this function to differentiate actual punctuation marks from marks used in words like 4:00 or Yitshak-Rabbin.

We begin reading each word on its own from the start, if we find a mark at the start or at the end then we know it's a punctuation mark and we add a new token and continue. Otherwise, the mark might be in the middle of the word (since we split based on spaces)

If we have less than 4 tokens then we don't save it. Finally, we combine all the tokens to a single string as requested:

```
                for token in all_tokens:
                    combine_tokens = ''
                    for word1 in token :
                        combine_tokens+=str(word1)+' '
                    speaker_text[prev_speaker].append(combine_tokens.strip())
```

And we save the text for the corresponding speaker in the dictionary.

8-

When were done going over all the paragraphs of a document, we save the data which we collected:

```
for speaker in speakers_order:
    for text in speaker_text[speaker]:
        jsonl_data.append({
            'protocol_name':protocol_name,
            'knesset_number':knesset_number,
            'protocol_type':protocol_type,
            'protocol_number':protocol_int,
            'speaker_name':speaker,
            'sentence_text':text
        })
```

We go over the list of all speakers we saved in order, and append each line of data for every text.

After we finish all the documents we save all the data into the jsonl file:

```
with open(output_path, 'w', encoding='utf-8') as jsonl_file:
    for data_item in jsonl_data: # change back
        # Convert the dictionary to json lines
        json_line = json.dumps(data_item, ensure_ascii=False)

        # Write the json line to the file
        jsonl_file.write(json_line + '\n')
```

Part 2:

1- We begin with reading the file and going over every word in the text and checking if its in Hebrew and not numbers and punctuation:

```
    output_path = sys.argv[2]

    # Read the JSONL file
    df = pd.read_json(jsonl_file, lines=True)

    # Count word frequencies
    frequency_dictionary = {}
    for row in df.itertuples():
        all_words = str(row[6]).split(" ")
        for word in all_words:
            if is_hebrew(word) == False:#must be hebrew word
                continue
            if word not in frequency_dictionary.keys(): # Add it to the dictionary
                frequency_dictionary[word]=1
            else:
                frequency_dictionary[word]+=1
```

We use this function to check if its in Hebrew:

```
def is_hebrew(word):
    try:
        if word == '':
            return False
        hebrew_letters = ['א', 'ב', 'ג', 'ד', 'ה', 'ו', 'ז', 'ח', 'ט', 'י', 'ך', 'כ', 'ל', 'ם', 'מ', 'ן', 'נ', 'ס', 'ע'
        for letter in word:
            if letter not in hebrew_letters:
                return False
        return True
    except Exception as e:
        print(f'Exception in is_hebrew: {e}')
```

We check if any item of the word isn't in the Hebrew letters.

Then we make a list for the rank and the frequency (log) and then sorting the frequency list for later use:

```
rank_list = [] # we sort the list
frequency_list = []
i=1
for word in frequency_dictionary.keys():
    frequency_list.append(np.log2(frequency_dictionary[word]))
    rank_list.append(np.log2(i))
    i+=1
frequency_list.sort(reverse= True)

# Sort the frequency dictionary by values in descending order
sorted_frequency = sorted(frequency_dictionary.items(), key=lambda x: x[1], reverse=True)
```

Then we get the graph like this:

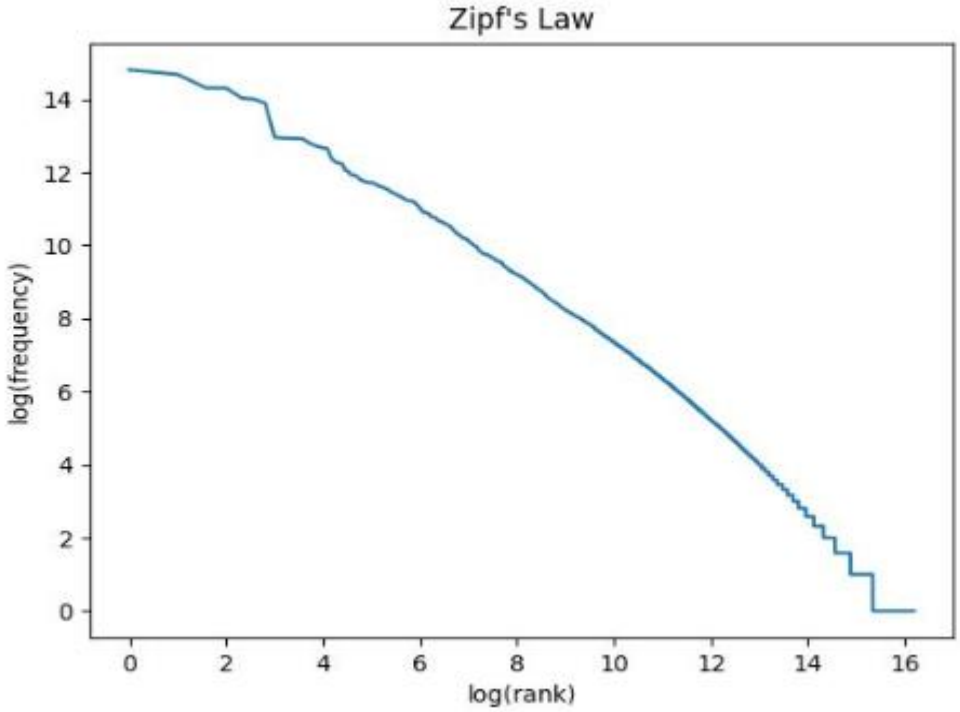
```
    # Plot the Zipf's law
    plt.plot(rank_list, frequency_list)
    plt.xlabel('log(rank)')
    plt.ylabel('log(frequency)')
    plt.title('Zipf\'s Law')
    plt.savefig(output_path)
```

2- This graph shows how often words appear compared to the rest of the words in the corpus. The frequency is the number of occurrences of a certain word while the rank is the order from the most frequent to the least frequent.

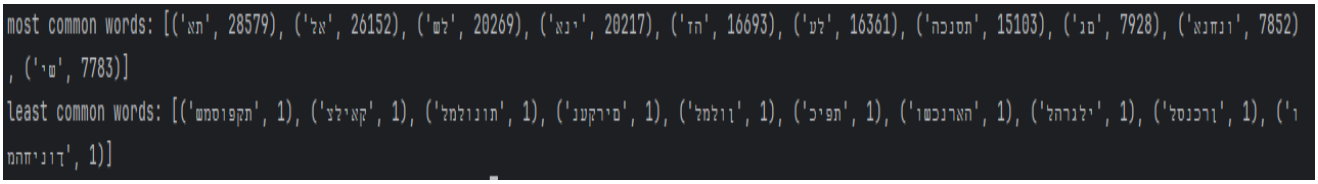
3- We got what we expected in the graph, based on Zipf's Law the frequency is inversely proportional to the rank. We don't get a perfect graph but that is expected. We can see that the graph is linear in the middle and nonlinear at the sides when we plot it with log on its values.

4- If we make it smaller then it will be less "smooth" since we don't have enough data to represent it. On the other hand, if we have more data then the line will be smoother for the same reason.

5-



6- We do that with this code:



(Note that the words are reversed when printed in hebrew)

Table of the most common words:

word	Frequency
את	28579
לא	26152
של	20269
אני	20217
זה	16693
על	16361
הכנסת	15103
גם	7928
אנחנו	7852
יש	7783

Table of the least common words

word	Frequency
שמסופקת	1
צליאק	1
למלונות	1
נעקרים	1
למליו	1
כיפת	1
ושכנראה	1
להרגל	1
לסנכרן	1
ומחינוך	1

We expect the most common words to be either associated with court, or very common in the language, we see "אנחנו" "הכנסת" which are heavily associated with court, "את" "לא" are common words in Hebrew.

The least common words are expected to be really long words that are rarely used in court and in Hebrew, we can see that in our example.