Part 1: Building Language Models

Firstly, we loaded the data using the load_jsonl function.

After loading the data, we filtered the sentences based on the protocol type (e.g., **committee** and **plenary**) using the get_sentences_by_type function.

Then to prepare the data for training the Trigram model, we added start tokens (<s_0> and <s_1>) at the beginning of each sentence using the add_start_tokens function.

After that, We implemented the **Trigram Language Model** using the Trigram_LM class. This involved the following steps:

- Calculating **unigram counts**: The frequency of each individual token in the corpus.
- Calculating **bigram counts**: The frequency of consecutive token pairs.
- Calculating **trigram counts**: The frequency of token triplets.
- Storing the vocabulary size and total token count for smoothing purposes.

Function: **calculate_prob_of_sentence**

Within the Trigram_LM class, we implemented the calculate_prob_of_sentence function. This function computes the log probability of a given sentence using:

- **Unigram probabilities**.
- **Bigram probabilities**.
- **Trigram probabilities**. To handle unseen words and ensure non-zero probabilities, we applied **Laplace smoothing**. Probabilities were combined using **linear interpolation** with predefined weights (λ1, λ2, λ3).
  Higher weight was given to the trigram because it captures the most context-specific information. Bigrams and unigrams received lower but non-zero weights to handle sparsity and provide a fallback.
  We decided to choose : $\lambda_1=0.1$, $\lambda_2=0.2$, $\lambda_3=0.6$ -> $\lambda_1+ \lambda_2+ \lambda_3 = 1$

Function: **generate_next_token**

Within the Trigram_LM class, we implemented the generate_next_token function.

This function was implemented to predict the most likely next token given a two-token context:

- Calculates trigram, bigram, and unigram probabilities using MLE.
- Applies linear interpolation to combine these probabilities.

- Returns the token with the highest log probability and its corresponding probability value.
  In cases where no valid token was found, the model defaults to the most frequent unigram as a fallback mechanism.

Part 2: Collocations

We implemented the get_k_n_t_collocations function within the Trigram_LM class, which operates by extracting n-grams, filtering them based on a threshold t, and ranking the top k n-grams using **Frequency** or **TF-IDF** scoring methods.

This function supports two ranking methods:

1. Frequency-Based Ranking
   - The remaining n-grams are sorted in descending order of their occurrence counts.
   - The top **k** most frequent n-grams are returned

2. TF-IDF-Based Ranking
   To rank the n-grams using TF-IDF, the function calls calculate_tfidf, which calculates the scores based on the TF-IDF formula provided in the task.

   This function works as follows:

   - It computes the **TF** of each n-gram, which measures how often the n-gram appears in a sentence relative to all n-grams in that sentence.
   - It calculates the **DF** to determine how many sentences contain the n-gram.
   - Using TF and DF, the function computes the **TF-IDF score** for each n-gram, assigning higher importance to n-grams that are frequent in individual sentences but not common across all sentences.

As required, we applied a **threshold t** to filter out n-grams that appear fewer than t times:

```
# frequency threshold `t`
filtered_ngrams = {ngram: count for ngram, count in ngram_counts.items() if count >= t}
```

At the end, the top k n-grams with the highest scores are printed to the knesset_collocations.txt file as requested.

Part 3: Applying the Language Models

Step 1:

We implemented the mask_tokens_in_sentences function. This function processes a list of sentences and replaces a specified percentage of tokens x in each sentence with the special placeholder token [*].

The function operates as follows:

1. **Calculate Number of Tokens to Mask**
    - For each sentence, the number of tokens to be masked is determined by multiplying the length of the sentence by x.
    - If the result is a decimal, it is rounded up or down based on standard rounding rules.
    - At least **one token** is always masked, even if x is very small.
2. **Random Selection of Tokens**
    - Random indices are selected within the sentence to determine which tokens will be replaced.
    - The random.sample method ensures no duplicate indices are chosen.
3. **Token Replacement**
    - The selected tokens are replaced with the placeholder [*] in a copy of the sentence to avoid modifying the original input.
4. **Store Masked Sentences**
    - The function returns a list of masked sentences, where each sentence includes the same structure as the original but with the masked tokens replaced.

Step 2:

We implemented the sample_and_save_sentences function. This function processes a list of sentences, filters them, and saves both the original and masked versions to files.

this function operates as follows:

1. **Sentence Filtering**
   - The function selects only sentences that contain **at least 5 tokens**.
2. **Sentence Sampling**
   - From the filtered sentences, the function randomly samples num_samples sentences.
3. **Token Masking**
   - For each sampled sentence, mask_tokens_in_sentences is called to replace **x%** of the tokens with the placeholder [*].
4. **Saving Results to Files**
   - The original sentences are saved to a file named original_sampled_sents.txt
   - The masked sentences are saved to a file named asked_sampled_sents.txt as the order of sentences in the in the original file.

Step 3:

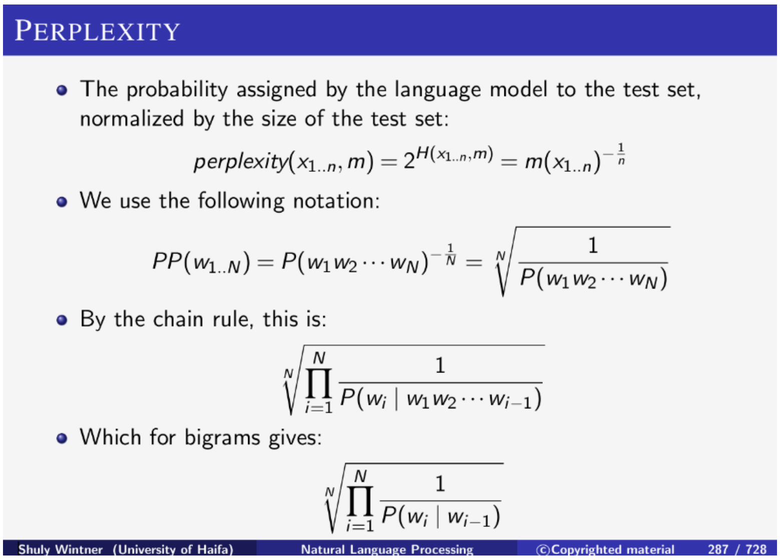
We implemented the restore_and_evaluate_sentences function.

this function operates as follows:

1. **Load Original and Masked Sentences**
   - The function reads the **original sentences** and the corresponding **masked sentences** from the input files.
2. **Restore Missing Tokens**
   - For each masked sentence, tokens marked as [*] are restored using the **plenary model**.
   - The restoration process works by predicting the masked token based on its **context**:
     - w_{k-2} (two tokens before the mask)
     - w_{k-1} (one token before the mask)
   - If no prior context exists, special tokens <s_0> and <s_1> are used.
3. **Track Generated Tokens**
   - The restored tokens are collected and stored in a comma-separated list.
4. **Evaluate the Restored Sentences**
   - The probabilities of the restored sentences are calculated using both:
     - The **plenary model** (trained on plenary corpus).
     - The **committee model** (trained on committee corpus).
5. **Save the Results**
   - The results for each sentence are written to the sampled_sents_results.txt file.

Step 4:

We implemented the calculate_perplexity function to measure how well the plenary language model predicts the masked tokens. Perplexity serves as a quantitative measure of the model's uncertainty when restoring the tokens, where a lower perplexity indicates better predictions.



We used this definition of perplexity to evaluate the performance of the plenary language model. Specifically:

1. **By the Chain Rule**:
The probability of a sentence can be broken down into smaller components, where each token's probability depends on its preceding tokens. This allows us to approximate the overall probability of a sentence using n-grams.

2. **Simplification for Bigrams**:
For simplicity, we considered each token's probability based on its immediate previous token. This approach makes it computationally feasible to evaluate perplexity.

3. **Perplexity Calculation**:
Perplexity is calculated as a measure of the model's uncertainty in predicting the masked tokens. It considers the average probability of the restored tokens within their context:

   - Higher probabilities mean the model is more confident in its predictions.
   - Lower perplexity scores indicate better performance because the model is less "surprised" by the restored tokens.

4. **Interpolation of Probabilities**:
To make the predictions more robust, we combined three probabilities:

   - **Unigram probability**: Based on the frequency of a single token.

- **Bigram probability**: Based on the frequency of a token given the previous token.
- **Trigram probability**: Based on the frequency of a token given the two previous tokens.

We assigned weights to each of these components:

- 10% to unigram, 30% to bigram, and 60% to trigram probabilities.

5. **Output**:

- The average perplexity was computed for all sentences, considering only the masked positions.
- The final result was saved to the file **perplexity_result.txt**.
- The average perplexity for the restored sentences is **17,243.48**.

The average perplexity for the restored sentences, with the current lambda values ($\lambda1 = 0.1, \lambda2 = 0.3, \lambda3 = 0.6$), is **17,243.48**.

If the lambda values are modified, the perplexity score changes to reflect the new weighting of unigram, bigram, and trigram probabilities.