Mohamad Arvin Fadriansyah
2006596996

1) Pada tahap 1.1 disebutkan bahwa terdapat dua macam skenario yang perlu dilakukan, yakni lowercasing (uncased) serta no lowercasing (cased). Bagaimana performa kedua model, baik unigram maupun bigram dari segi perplexitynya? Model mana yang secara perplexity lebih baik?

Model bigram dibandingkan unigram karena memiliki perplexity yang lebih rendah karena mempertimbangkan dari konteks sebelumnya.

Model Uncased memiliki perplexity yang lebih rendah karena mengurangi variasi dan penggunaan huruf kapital sehingga corpus menjadi lebih seragam.

Huruf kapital tidak banyak membawa informasi yang penting sehingga loweringcase tidak adakan mengurangi peforma prediksi sehingga perplexity cenderung lebih rendah.

3) Proses apa yang Anda lakukan saat implementasi tokenization menggunakan library Aksara2 ?

```
"""
- Fungsionalitas method di bawah ini adalah melakukan pembersihan kalimat dari simbol newline ('\n').
- Di samping itu, method diharapkan dapat melakukan trimming whitespace pada awal dan akhir kalimat.
- Output format yang diharapkan berupa list of strings.
"""

def split_sentences(self, data: list[str]) -> list[str]:  1 usage  new *
    # TODO: Implement based on the given description
    cleaned_sentences = []
    for sentence in data:
        sentence = sentence.replace( _old: '\n',  _new: '').strip()
        cleaned_sentences.append(sentence)

    return cleaned_sentences
```

Parameter:

Fungsi menerima dua parameter: self (merujuk pada instance dari kelas) dan data, yang merupakan daftar string (list[str]), dimana setiap elemen di dalamnya adalah sebuah kalimat.

Inisialisasi List Kosong:

- Sebuah list kosong bernama cleaned_sentences diinisialisasi untuk menyimpan kalimat-kalimat yang telah dibersihkan.

Iterasi pada Kalimat:

- Fungsi kemudian melakukan iterasi pada setiap kalimat di dalam data.

Membersihkan Kalimat:

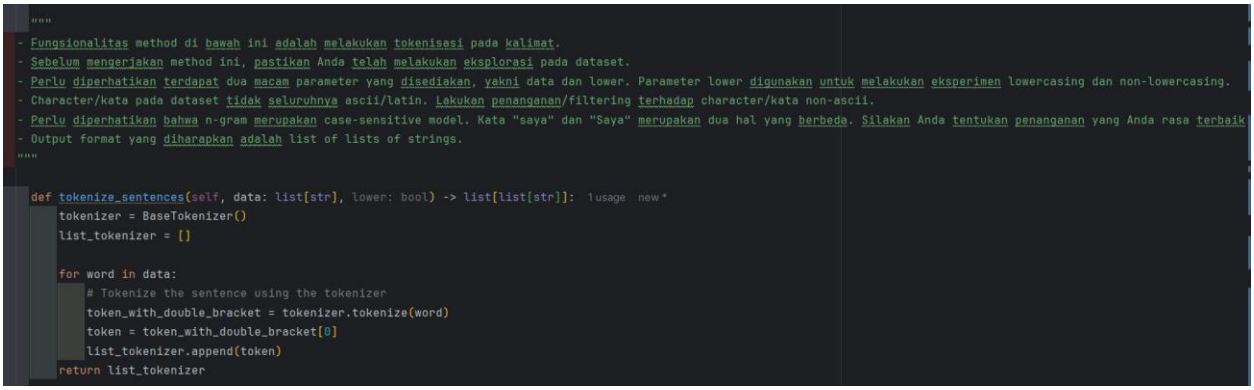
- Untuk setiap kalimat, fungsi menggunakan method replace() untuk menghilangkan semua simbol newline (\n) dan menggantinya dengan string kosong. Setelah itu, fungsi menggunakan method strip() untuk menghapus semua spasi di awal dan akhir kalimat.

Menambahkan Kalimat Bersih ke List:

- Kalimat yang sudah dibersihkan dari newline dan spasi tambahan tersebut kemudian ditambahkan ke dalam list cleaned_sentences.

Pengembalian List:

- Setelah iterasi selesai, list cleaned_sentences yang berisi kalimat-kalimat bersih dikembalikan sebagai output.



```
"""
- Fungsionalitas method di bawah ini adalah melakukan tokenisasi pada kalimat.
- Sebelum mengerjakan method ini, pastikan Anda telah melakukan eksplorasi pada dataset.
- Perlu diperhatikan terdapat dua macam parameter yang disediakan, yakni data dan lower. Parameter lower digunakan untuk melakukan eksperimen lowercasing dan non-lowercasing.
- Character/kata pada dataset tidak seluruhnya ascii/latin. Lakukan penanganan/filtering terhadap character/kata non-ascii.
- Perlu diperhatikan bahwa n-gram merupakan case-sensitive model. Kata "saya" dan "Saya" merupakan dua hal yang berbeda. Silakan Anda tentukan penanganan yang Anda rasa terbaik
- Output format yang diharapkan adalah list of lists of strings.
"""

def tokenize_sentences(self, data: list[str], lower: bool) -> list[list[str]]:  1 usage  new *
    tokenizer = BaseTokenizer()
    list_tokenizer = []

    for word in data:
        # Tokenize the sentence using the tokenizer
        token_with_double_bracket = tokenizer.tokenize(word)
        token = token_with_double_bracket[0]
        list_tokenizer.append(token)
    return list_tokenizer
```

Parameter:

- self: merujuk pada instance dari kelas.
- data: merupakan daftar string (list[str]), yaitu kalimat-kalimat yang akan ditokenisasi.
- lower: boolean untuk menentukan apakah teks akan diubah menjadi huruf kecil atau tidak, yang memungkinkan eksperimen antara lowercasing dan non-lowercasing.

Inisialisasi Tokenizer:

- Sebuah instance dari BaseTokenizer dibuat, yang nantinya akan digunakan untuk proses tokenisasi setiap kalimat atau kata dalam data.

Inisialisasi List Kosong:

- list_tokenizer diinisialisasi sebagai list kosong untuk menampung hasil tokenisasi.

Iterasi pada Data:

- Fungsi melakukan iterasi pada setiap kalimat (atau kata) di dalam data.

Proses Tokenisasi:

- Setiap kata atau kalimat ditokenisasi menggunakan tokenize() dari BaseTokenizer.
- Setelah tokenisasi, token yang dihasilkan berupa list, dan token yang berada di dalam double bracket (list dalam list) diakses melalui index 0, kemudian disimpan ke list_tokenizer.

Pengembalian Hasil:

- Setelah semua kata atau kalimat selesai diproses, list_tokenizer yang berisi token hasil tokenisasi dikembalikan sebagai output.

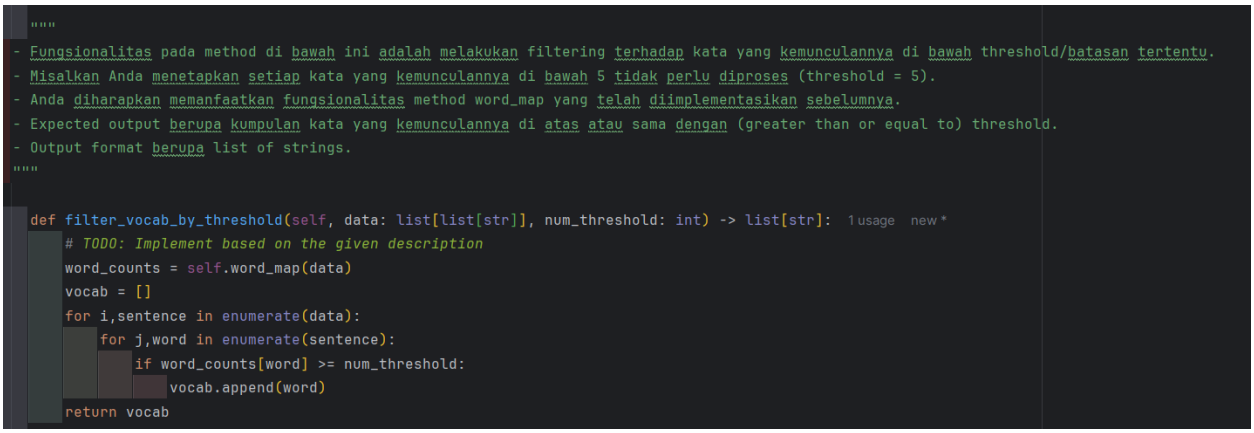
```
"""
- Fungsionalitas pada method di bawah ini adalah menghitung kemunculan kata di dalam corpus.
- Output format yang diharapkan berupa dictionary dengan pasangan key berupa kata/token dan value berupa jumlah kemunculan kata/token tsb.
"""

def word_map(self, data: list[list[str]]) -> dict:  1 usage  new *
    # TODO: Implement based on the given description
    word_map: dict[str, int] = {}
    for sentence in data:
        for word in sentence:
            if word not in word_map:
                word_map[word] = 0
            else:
                word_map[word] += 1
    return word_map
```

1. Metode ini menerima parameter data yang merupakan list of lists of strings, di mana setiap list bagian dalam mewakili sebuah kalimat dan setiap string di dalamnya mewakili sebuah kata.
2. Sebuah dictionary kosong word_map diinisialisasi untuk menyimpan hasil perhitungan frekuensi kata.
3. Metode ini menggunakan nested loops untuk mengiterasi setiap kalimat dalam data dan kemudian setiap kata dalam kalimat tersebut.
4. Untuk setiap kata yang ditemui:
    - Jika kata tersebut belum ada dalam word_map, ditambahkan ke word_map dengan nilai awal 0.
    - Jika kata sudah ada dalam word_map, nilai frekuensinya ditambah 1.
5. Setelah semua kata dihitung, metode mengembalikan word_map.

Hasil akhirnya adalah sebuah dictionary di mana setiap key adalah kata unik yang ditemukan dalam corpus, dan value-nya adalah jumlah kemunculan kata tersebut di seluruh corpus.

Mohamad Arvin Fadriansyah
2006596996

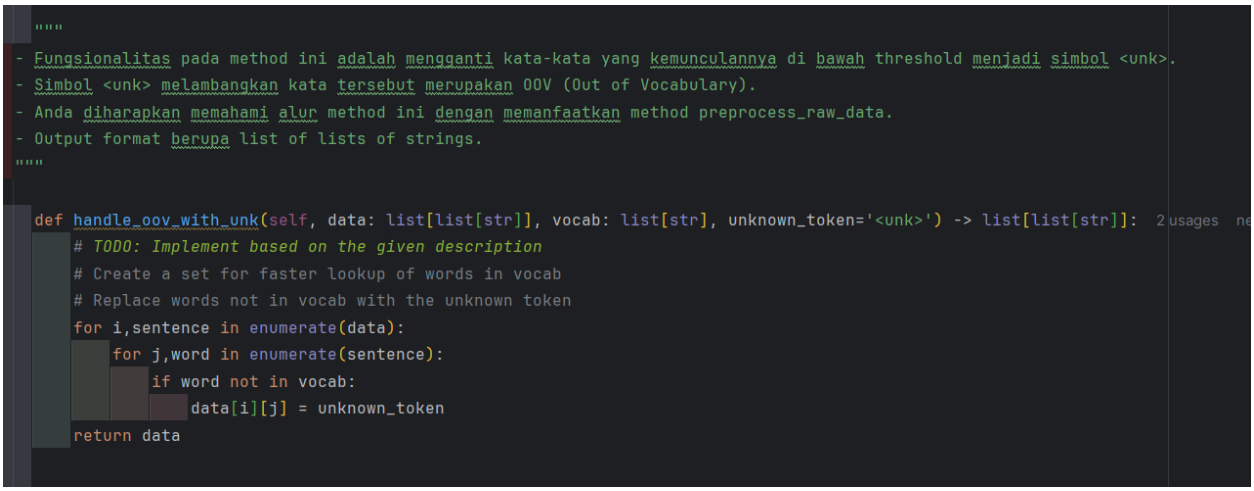


```
"""
- Fungsionalitas pada method di bawah ini adalah melakukan filtering terhadap kata yang kemunculannya di bawah threshold/batasan tertentu.
- Misalkan Anda menetapkan setiap kata yang kemunculannya di bawah 5 tidak perlu diproses (threshold = 5).
- Anda diharapkan memanfaatkan fungsionalitas method word_map yang telah diimplementasikan sebelumnya.
- Expected output berupa kumpulan kata yang kemunculannya di atas atau sama dengan (greater than or equal to) threshold.
- Output format berupa list of strings.
"""

def filter_vocab_by_threshold(self, data: list[list[str]], num_threshold: int) -> list[str]:  1 usage  new *
    # TODO: Implement based on the given description
    word_counts = self.word_map(data)
    vocab = []
    for i,sentence in enumerate(data):
        for j,word in enumerate(sentence):
            if word_counts[word] >= num_threshold:
                vocab.append(word)
    return vocab
```

1. Metode ini menerima dua parameter: data (list of lists of strings, mewakili kumpulan kalimat) dan num_threshold (integer, batas minimum frekuensi kata).
2. Pertama, metode memanggil self.word_map(data) untuk menghitung frekuensi setiap kata dalam dataset. Hasilnya disimpan dalam word_counts.
3. Sebuah list kosong vocab diinisialisasi untuk menyimpan kata-kata yang lolos filter.
4. Metode menggunakan nested loops dengan enumerate() untuk mengiterasi setiap kalimat dan kata dalam dataset.
5. Untuk setiap kata, metode memeriksa apakah frekuensinya (dari word_counts) sama dengan atau melebihi num_threshold.
6. Jika kata memenuhi syarat threshold, kata tersebut ditambahkan ke dalam vocab.
7. Setelah semua kata diperiksa, metode mengembalikan vocab.

Hasil akhirnya adalah sebuah list yang berisi kata-kata unik yang muncul setidaknya sebanyak num_threshold kali dalam dataset. Metode ini berguna untuk mengurangi vocabulary size dengan menghilangkan kata-kata yang jarang muncul, yang sering kali dapat membantu dalam berbagai tugas pemrosesan bahasa alami dengan mengurangi noise dan meningkatkan efisiensi.



```
"""
- Fungsionalitas pada method ini adalah mengganti kata-kata yang kemunculannya di bawah threshold menjadi simbol <unk>.
- Simbol <unk> melambangkan kata tersebut merupakan OOV (Out of Vocabulary).
- Anda diharapkan memahami alur method ini dengan memanfaatkan method preprocess_raw_data.
- Output format berupa list of lists of strings.
"""

def handle_oov_with_unk(self, data: list[list[str]], vocab: list[str], unknown_token='<unk>') -> list[list[str]]:  2 usages  ne
    # TODO: Implement based on the given description
    # Create a set for faster lookup of words in vocab
    # Replace words not in vocab with the unknown token
    for i,sentence in enumerate(data):
        for j,word in enumerate(sentence):
            if word not in vocab:
                data[i][j] = unknown_token
    return data
```

Mohamad Arvin Fadriansyah
2006596996

Kode yang ditampilkan bertujuan untuk menangani kata-kata yang tidak ada dalam kosakata (Out of Vocabulary, OOV) dengan menggantinya menggunakan simbol <unk>. Berikut adalah penjelasan langkah-langkah yang dilakukan dalam kode:

1. Fungsi handle_oov_with_unk menerima 3 parameter:
    - data: Sebuah list of lists yang berisi kalimat-kalimat dalam bentuk list kata-kata.
    - vocab: Sebuah list yang berisi kata-kata yang ada di kosakata.
    - unknown_token: Simbol yang digunakan untuk menggantikan kata-kata OOV, secara default diset sebagai <unk>.
2. Proses yang dilakukan:
    - Dua loop for digunakan:
        - Loop pertama, for i, sentence in enumerate(data) untuk mengiterasi setiap kalimat dalam data.
        - Loop kedua, for j, word in enumerate(sentence) untuk mengiterasi setiap kata dalam kalimat tersebut.
    - Jika sebuah kata tidak ada dalam vocab, kata tersebut diganti dengan unknown_token (default <unk>).
3. Pengembalian hasil:
    - Kode akan mengembalikan data yang telah dimodifikasi di mana kata-kata OOV telah diganti.

2) Laporkan kalimat yang telah Anda hasilkan pada tahap 1.2 untuk model unigram dan bigram. Bagaimana pendapat Anda terkait masing-masing kalimat tersebut? Anda dipersilakan meninjau dari sisi tata bahasa, keselarasan makna antar kata, dsb.

```python
def generate_n_grams(self, data: list[list[str]], n: int, start_token: str = '<s>', end_token='</s>') -> dict:
    # TODO: Implement based on the given description
    grams_count = {}
    for i,sentence in enumerate(data):
        sentence.insert( _index: 0,start_token)
        sentence.append(end_token)
        data[i] = sentence
    for sentence in data:
        first_index = 0
        last_index = n
        while last_index <= len(sentence):
            gram = tuple(sentence[first_index:last_index])

            grams_count[gram] = grams_count.get(gram, 0) + 1
            first_index += 1
            last_index += 1
    return grams_count
```

Fungsi generate_n_grams menerima parameter:

- data: List yang berisi kalimat-kalimat dalam bentuk list kata.
- n: Ukuran dari n-gram (misalnya, 2 untuk bigram, 3 untuk trigram, dll.).
- start_token dan end_token: Token khusus yang ditambahkan ke awal dan akhir setiap kalimat, defaultnya '<s>' dan '</s>'.

Langkah-langkah dalam kode:

- Pertama, dictionary grams_count digunakan untuk menyimpan n-gram dan menghitung kemunculannya.
- Loop pertama, for i, sentence in enumerate(data):
    - start_token disisipkan di awal kalimat.
    - end_token ditambahkan di akhir kalimat.
    - Kalimat yang telah dimodifikasi ini kemudian dimasukkan kembali ke data.
- Loop kedua mengiterasi kembali setiap kalimat:
    - Dua variabel first_index dan last_index digunakan untuk menentukan jendela n-gram dari kalimat.
    - Loop while digunakan untuk mengekstraksi n-gram dari potongan kalimat dengan ukuran n (dari first_index sampai last_index).
    - N-gram yang dihasilkan kemudian dihitung dengan menggunakan grams_count.

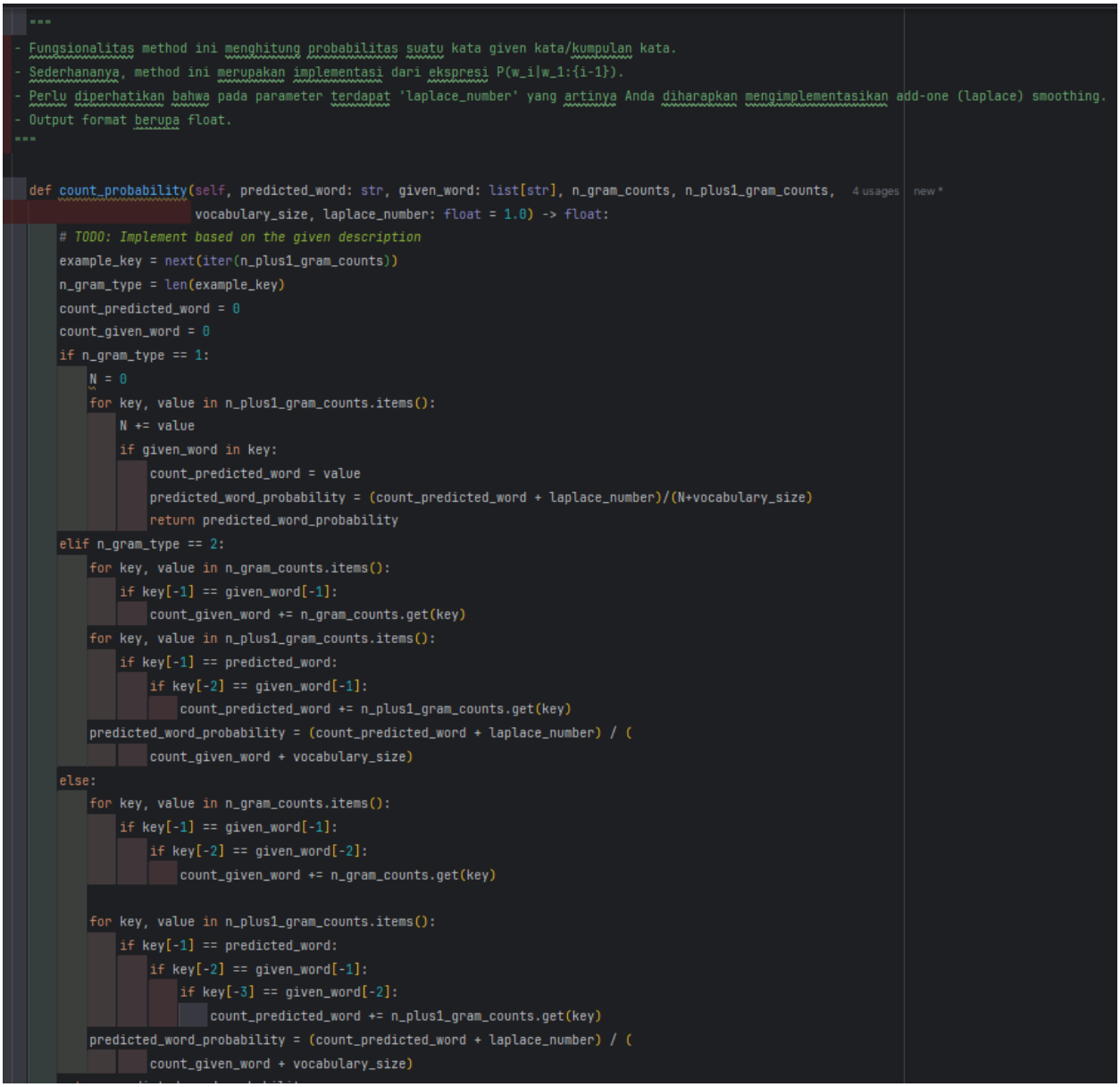
Proses pergeseran:

- Setelah n-gram pertama dihasilkan, indeks awal (first_index) dan akhir (last_index) masing-masing ditingkatkan sebanyak 1 sehingga proses ini dapat menghasilkan semua n-gram dalam kalimat.

Pengembalian hasil:

- Dictionary grams_count yang berisi n-gram sebagai kunci dan jumlah kemunculannya sebagai nilai dikembalikan sebagai hasil fungsi.

Mohamad Arvin Fadriansyah
2006596996



```
"""
- Fungsionalitas method ini menghitung probabilitas suatu kata given kata/kumpulan kata.
- Sederhananya, method ini merupakan implementasi dari ekspresi P(w_i|w_1:{i-1}).
- Perlu diperhatikan bahwa pada parameter terdapat 'laplace_number' yang artinya Anda diharapkan mengimplementasikan add-one (laplace) smoothing.
- Output format berupa float.
"""


def count_probability(self, predicted_word: str, given_word: list[str], n_gram_counts, n_plus1_gram_counts,  4 usages  new *
                      vocabulary_size, laplace_number: float = 1.0) -> float:
    # TODO: Implement based on the given description
    example_key = next(iter(n_plus1_gram_counts))
    n_gram_type = len(example_key)
    count_predicted_word = 0
    count_given_word = 0
    if n_gram_type == 1:
        N = 0
        for key, value in n_plus1_gram_counts.items():
            N += value
            if given_word in key:
                count_predicted_word = value
                predicted_word_probability = (count_predicted_word + laplace_number)/(N+vocabulary_size)
                return predicted_word_probability
    elif n_gram_type == 2:
        for key, value in n_gram_counts.items():
            if key[-1] == given_word[-1]:
                count_given_word += n_gram_counts.get(key)
        for key, value in n_plus1_gram_counts.items():
            if key[-1] == predicted_word:
                if key[-2] == given_word[-1]:
                    count_predicted_word += n_plus1_gram_counts.get(key)
        predicted_word_probability = (count_predicted_word + laplace_number) / (
                count_given_word + vocabulary_size)
    else:
        for key, value in n_gram_counts.items():
            if key[-1] == given_word[-1]:
                if key[-2] == given_word[-2]:
                    count_given_word += n_gram_counts.get(key)

        for key, value in n_plus1_gram_counts.items():
            if key[-1] == predicted_word:
                if key[-2] == given_word[-1]:
                    if key[-3] == given_word[-2]:
                        count_predicted_word += n_plus1_gram_counts.get(key)
        predicted_word_probability = (count_predicted_word + laplace_number) / (
                count_given_word + vocabulary_size)
```

Fungsi count_probability menerima parameter:

- predicted_word: Kata yang probabilitasnya ingin dihitung.
- given_word: Sebuah list kata yang memberikan konteks untuk kata yang diprediksi.
- n_gram_counts: Dictionary yang berisi hitungan n-gram (n-kata) dari korpus.
- n_plus1_gram_counts: Dictionary untuk hitungan (n+1)-gram.
- vocabulary_size: Ukuran dari kosakata yang digunakan.
- laplace_number: Parameter untuk add-one smoothing Laplace (default adalah 1.0).

- Inisialisasi:
    - n_gram_type dihitung berdasarkan panjang dari kunci n-gram, yang membantu menentukan apakah sedang bekerja dengan unigram, bigram, atau trigram.

    - count_predicted_word dan count_given_word diinisialisasi untuk menghitung kemunculan kata yang diprediksi dan konteksnya.
- Jika n_gram_type = 1 (unigram):
    - Loop pertama menghitung total kemunculan semua unigram (N), lalu memeriksa jika kata yang diberikan (given_word) ada di unigram.
    - Probabilitas kata yang diprediksi dihitung dengan membagi kemunculan kata yang diprediksi dengan total kemunculan semua unigram (N), ditambah Laplace smoothing.
- Jika n_gram_type = 2 (bigram):
    - Dua loop digunakan:
        1. Loop pertama menghitung kemunculan dari given_word sebagai bigram, jika cocok dengan kata yang diberikan.
        2. Loop kedua menghitung kemunculan dari kata yang diprediksi bersama-sama dengan kata yang diberikan sebagai bigram.
    - Probabilitas kata yang diprediksi dihitung menggunakan kemunculan n+1-gram yang terkait.
- Jika n_gram_type > 2 (trigram atau lebih):
    - Sama seperti bigram, tetapi dilakukan lebih banyak pengecekan berdasarkan urutan kata dalam trigram.

```
"""
- Silakan Anda menggunakan method ini untuk bermain-main/menguji segala kemungkinan sentence/word generation berdasarkan method count_probability yang telah Anda bangun.
"""

def probabilities_for_all_vocab(self, given_word: list[str], n_gram_counts, n_plus1_gram_counts, vocabulary,  3 usages  new *
                                end_token='</s>', unknown_token='<unk>', laplace_number=1.0):
    example_key = next(iter(n_plus1_gram_counts))  # Get the first key from n_gram_counts
    n_gram_type = len(example_key)
    vocabulary = vocabulary + [end_token, unknown_token]
    vocab_size = len(vocabulary)
    probs = dict()
    if n_gram_type == 1:
        for word in given_word:
            prob = self.count_probability(word, word, n_gram_counts, n_plus1_gram_counts, vocab_size,
                                          laplace_number=laplace_number)
            probs[word] = prob
    else:
        for word in vocabulary:
            prob = self.count_probability(word, given_word, n_gram_counts, n_plus1_gram_counts, vocab_size,
                                          laplace_number=laplace_number)
            probs[word] = prob
    return probs
```

Parameter Input:

- given_word: Daftar kata yang diberikan sebagai input.
- n_gram_counts, n_plus1_gram_counts: Data frekuensi kemunculan n-gram dan n+1-gram yang digunakan untuk menghitung probabilitas.
- vocabulary: Kumpulan kata yang akan digunakan untuk menghitung probabilitas.
- laplace_number: Parameter smoothing Laplace (default 1.0) untuk mencegah probabilitas nol.

Mohamad Arvin Fadriansyah
2006596996

Persiapan Awal:

- end_token dan unknown_token ditambahkan sebagai token spesial untuk kata akhir dan kata yang tidak dikenal.
- Dari n_plus1_gram_counts, contoh pasangan n-gram diambil untuk menentukan panjang tipe n-gram (n_gram_type).
- Ukuran kosakata dihitung, termasuk token akhir dan token yang tidak dikenal.
- Sebuah dictionary (probs) disiapkan untuk menyimpan hasil probabilitas.

Penghitungan Probabilitas:

- Jika n_gram_type adalah 1 (berarti model unigram), maka perulangan dilakukan pada given_word untuk menghitung probabilitas setiap kata dengan memanggil fungsi count_probability (metode yang sudah dibangun sebelumnya).
- Jika n_gram_type lebih dari 1, perulangan dilakukan pada seluruh kata dalam kosakata (vocabulary), dan probabilitas dihitung untuk setiap kata berdasarkan frekuensi n-gram dan n+1-gram.

Pengembalian Nilai:

- Setelah semua probabilitas dihitung, hasilnya disimpan dalam dictionary probs, yang kemudian dikembalikan sebagai output fungsi.

```
    """
    - Fungsionalitas pada method ini adalah untuk mengevaluasi n-gram model Anda menggunakan metrik perplexity.
    """

    def count_perplexity(self, sentence, n_gram_counts, n_plus1_gram_counts, vocab_size, vocab, start_token='<s>',  4 usages  new *
                         end_token='</s>', laplace_number=1.0):
        # TODO: Implement based on the given description
        sentence.insert(0, start_token)
        sentence.append(end_token)
        example_key = next(iter(n_plus1_gram_counts))
        n_gram_type = len(example_key)
        log_probability = 0.0
        probs = dict()
        if n_gram_type == 1:
            for word in sentence:
                prob = self.count_probability(word, word, n_gram_counts, n_plus1_gram_counts, vocab_size,
                                              laplace_number=laplace_number)
                probs[word] = prob
        else:
            for word in vocab:
                prob = self.count_probability(word, sentence, n_gram_counts, n_plus1_gram_counts, vocab_size,
                                              laplace_number=laplace_number)
                probs[word] = prob
        for key, value in probs.items():
            log_probability += math.log(value)
        perplexity = math.exp(-log_probability / len(sentence))
        return perplexity
```

Mohamad Arvin Fadriansyah
2006596996

1. Parameter Input:
    - sentence: Kalimat yang diberikan untuk evaluasi.
    - n_gram_counts, n_plus1_gram_counts: Data frekuensi n-gram dan n+1-gram.
    - vocab_size: Ukuran kosakata.
    - vocab: Daftar kosakata yang digunakan dalam model.
    - laplace_number: Nilai smoothing Laplace, dengan default 1.0.
    - start_token dan end_token: Token khusus untuk menandai awal dan akhir kalimat.
2. Langkah Awal:
    - Kalimat diberikan token awal (start_token) di awal dan token akhir (end_token) di akhir.
    - Tipe n-gram ditentukan berdasarkan contoh pasangan n-gram dari n_plus1_gram_counts.
    - Variabel log_probability disiapkan untuk menghitung total logaritma probabilitas kalimat.
    - Sebuah dictionary probs disiapkan untuk menyimpan probabilitas kata-kata.
3. Penghitungan Probabilitas:
    - Jika tipe n-gram adalah 1 (unigram), maka probabilitas dihitung untuk setiap kata dalam kalimat menggunakan metode count_probability. Probabilitas setiap kata disimpan di probs.
    - Jika tipe n-gram lebih dari 1, probabilitas dihitung untuk setiap kata dalam kosakata (vocab) menggunakan data n-gram dan n+1-gram.
4. Menghitung Logaritma Probabilitas Total:
    - Setelah semua probabilitas dihitung, logaritma dari masing-masing probabilitas dijumlahkan ke variabel log_probability.
5. Menghitung Perplexity:
    - Perplexity dihitung menggunakan rumus eksponensial dari negatif logaritma probabilitas total dibagi dengan panjang kalimat. Ini menghasilkan nilai perplexity yang mengindikasikan seberapa baik model dapat memprediksi kata-kata dalam kalimat tersebut.
6. Mengembalikan Nilai Perplexity:
    - Nilai perplexity kemudian dikembalikan sebagai hasil dari fungsi.

Secara keseluruhan, fungsi ini digunakan untuk mengukur seberapa baik model n-gram memprediksi kata-kata dalam kalimat, dan mengembalikan nilai perplexity sebagai indikator kualitas model.