REPORT FOR SENTENCE AUTOCOMPLETION USING PROBABILITY

This code is an implementation of a language model using n-grams. Specifically, it is using unigrams, bigrams, trigrams, and quadgrams to build a model of a given corpus. It then provides a function that suggests the possible next word given an input sentence using the probabilities calculated from the n-grams.

The code first downloads several text files from Project Gutenberg and stores them in a variable called **l**. It then tokenizes the text by splitting on whitespace and stores the lowercase version of each token in **lower_case_corpus**. It also creates a set of unique words in **lower_case_corpus** and stores it in **vocab**.

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

```
CORPUS EXAMPLE: ['\ufeffthe', 'project', 'gutenberg', 'ebook', 'of', 'crime', 'and', 'punishment,', 'by', 'fyodor', 'dostoevsky', 'this', 'ebook', 'i
s', 'for', 'the', 'use', 'of', 'anyone', 'anywhere', 'in', 'the', 'united', 'states', 'and', 'most', 'other', 'parts', 'of', 'the']


VOCAB EXAMPLE: ['moody', 'starting.', "tortur'd", 'addressing,', 'what!”', 'legislating!”', 'confidence', 'bar’ls.”', 'proved', 'lace,']
```

```
print('Total words in Corpus: ' + str(len(lower_case_corpus)))
print('Vocab of the Corpus: ' + str(len(vocab)))
```

```
Total words in Corpus: 549241
Vocab of the Corpus: 46819
```

Next, it calculates the counts for each n-gram in the corpus and stores them in **unigram_counts**, **bigram_counts**, **trigram_counts**, and **quadgram_counts**.

```python
unigram_counts={}
bigram_counts = {}
trigram_counts = {}
quadgram_counts={}

for i in range(len(lower_case_corpus)):
      unigram= (lower_case_corpus[i])
      if unigram in unigram_counts.keys():
         unigram_counts[unigram] += 1
      else:
         unigram_counts[unigram] = 1
for i in range(len(lower_case_corpus) - 1):
      bigram = (lower_case_corpus[i], lower_case_corpus[i+1])
      if bigram in bigram_counts.keys():
         bigram_counts[bigram] += 1
      else:
         bigram_counts[bigram] = 1
for i in range(len(lower_case_corpus) - 2):
      trigram = (lower_case_corpus[i], lower_case_corpus[i+1], lower_case_corpus[i+2])
      if trigram in trigram_counts.keys():
         trigram_counts[trigram] += 1
      else:
         trigram_counts[trigram] = 1
for i in range(len(lower_case_corpus) - 3):
      quadgram = (lower_case_corpus[i], lower_case_corpus[i+1], lower_case_corpus[i+2], lower_case_corpus[i+3])
      if quadgram in quadgram_counts.keys():
         quadgram_counts[quadgram] += 1
      else:
         quadgram_counts[quadgram] = 1



print("Example, count for unigram ('the') is: " + str(unigram_counts[('the')]))
print("Example, count for bigram ('the', 'king') is: " + str(bigram_counts[('the', 'king')]))
print("Example, count for trigram ('the', 'king' ,'of') is: " + str(trigram_counts[('the', 'king','of')]))
```

```
Example, count for unigram ('the') is: 26098
Example, count for bigram ('the', 'king') is: 47
Example, count for trigram ('the', 'king' ,'of') is: 15
```

Finally, it defines a function called **suggest_next_word** that takes an input sentence, the n-gram counts, and the vocabulary as inputs. It then uses the probabilities calculated from the n-grams to suggest the possible next words that could follow the input sentence. The function first considers the last trigram of the input sentence and calculates the probability of each word in the vocabulary given that trigram. If there are no occurrences of the trigram in the corpus, it tries with the last bigram, and if that fails, with the last unigram. The function returns a dictionary of possible next words along with their probabilities.

```python
# Function takes sentence as input and suggests possible words that comes after the sentence
def suggest_next_word(input_, bigram_counts, trigram_counts, vocab):
    # Consider the last bigram of sentence
    tokenized_input = word_tokenize(input_.lower())
    last_trigram= tokenized_input[-3:]
    last_bigram = tokenized_input[-2:]
    last_unigram= tokenized_input[-1:]

    # Calculating probability for each word in vocab
    vocab_probabilities = {}
    for vocab_word in vocab:
        test_quadgram = (last_trigram[0], last_trigram[1], last_trigram[2], vocab_word)
        test_trigram = (last_trigram[0], last_trigram[1], last_trigram[2])

        test_quadgram_count = quadgram_counts.get(test_quadgram, 0)
        test_trigram_count = trigram_counts.get(test_trigram, 0)

        if(test_trigram_count!=0):
           probability = test_quadgram_count / test_trigram_count
           vocab_probabilities[vocab_word] = probability

        else:

            test_trigram = (last_bigram[0], last_bigram[1], vocab_word)
            test_bigram = (last_bigram[0], last_bigram[1])

            test_trigram_count = trigram_counts.get(test_trigram, 0)
            test_bigram_count = bigram_counts.get(test_bigram, 0)

            if(test_bigram_count!=0):
                probability = test_trigram_count / test_bigram_count
                vocab_probabilities[vocab_word] = probability
            else:

                test_bigram = (last_unigram[0], vocab_word)
                test_unigram = (last_unigram[0])

                test_bigram_count = bigram_counts.get(test_bigram, 0)
                test_unigram_count = unigram_counts.get(test_unigram, 0)

                if(test_unigram_count!=0):
                  probability = test_bigram_count / test_unigram_count
                  vocab_probabilities[vocab_word] = probability
                else:
                  vocab_probabilities[vocab_word] = 0




    # Sorting the vocab probability in descending order to get top probable words
    top_suggestions = sorted(vocab_probabilities.items(), key=lambda x: x[1], reverse=True)[:1]
    return top_suggestions[0][0]
```

```python
suggest_next_word('I am the king',bigram_counts, trigram_counts,  vocab)
```

```
[('of', 0.3191489361702128)]
```

```python
suggest_next_word('I am the king of', bigram_counts, trigram_counts, vocab)
```

```
[('the', 0.6)]
```

```python
suggest_next_word('I am the king of france ', bigram_counts, trigram_counts, vocab)
```

```
[('and', 0.6666666666666666)]
```