```
import pickle
import nltk


def ngram_model(filename):
    with open(filename, 'rb') as file:
        unigrams, bigrams = pickle.load(file)
    return unigrams, bigrams

# Load pickled language models
english_unigrams, english_bigrams = ngram_model('english_lang_model.pkl')
french_unigrams, french_bigrams = ngram_model('french_lang_model.pkl')
italian_unigrams, italian_bigrams = ngram_model('italian_lang_model.pkl')



# Combines keys from all language models and count unique unigrams
unigrams_count = len(set(english_unigrams.keys()) | set(french_unigrams.keys()) | set(italian_unigrams.keys()))


def sentence_probability(sentence, unigrams, bigrams, vocab_size):
    """
    Calculate the probability of a sentence based on unigrams and bigrams.

    Args:
    - sentence (str): The input sentence.
    - unigrams (dict): Dictionary containing unigram frequencies.
    - bigrams (dict): Dictionary containing bigram frequencies.
    - vocab_size (int): Size of the vocabulary.

    Returns:
    - probability (float): Probability of the sentence.
    """
    # Tokenize the input sentence into words
    tokens = nltk.word_tokenize(sentence)

    # Generate bigrams from the tokenized words
    sentence_bigrams = list(nltk.bigrams(tokens))

    # Initialize probability with 1.0
    probability = 1.0

    # Calculate probability based on bigrams
    for bigram in sentence_bigrams:
        # Convert the bigram tuple into a string
        bigram_str = ' '.join(bigram)

        # Get the count of the bigram from the bigrams dictionary
        bigram_count = bigrams.get(bigram_str, 0)

        # Get the count of the first word in the bigram from the unigrams dictionary
        word_count = unigrams.get(bigram[0], 0)

        # Update the probability using add-1 smoothing
        probability *= (bigram_count + 1) / (word_count + vocab_size)

    return probability
```

```
def predict_language(sentence, vocab_size):
    """
    Predict the language of a sentence based on probabilities.

    Args:

def evaluate_language_identification(test_file_path, solution_file_path, output_file_path):
    # Opens test file, solution file, and output file
    with open(test_file_path, 'r') as test_file, open(solution_file_path, 'r') as solution_file, open(output_file_path, 'w') as
        test_lines = test_file.readlines()
        solution_lines = solution_file.readlines()

        # Performs language identification and evaluation
        correct_lines = 0
        incorrect_lines = []

        for i, test_line in enumerate(test_lines):
            # Predicts language for each line in the test data
            predicted_lang = predict_language(test_line, unigrams_count)

            # Extracts the actual language from the solution file
            actual_language = solution_lines[i].split(maxsplit=1)[1].strip()

            # Compares predicted and actual languages
            if predicted_lang == actual_language:
                correct_lines += 1
            else:
                incorrect_lines.append(i + 1)

            # Writes predicted language to the output file
            output_file.write(predicted_lang + '\n')

        # Calculates accuracy
        accuracy = (correct_lines / len(test_lines)) * 100

        # Output results
        print("Classified Lines Accuracy:", accuracy, "%")
        print("Incorrectly classified lines:", incorrect_lines)

# Calls the function to evaluate language identification and write predictions
evaluate_language_identification('LangId.test.txt', 'LangId.sol.txt', 'wordLangId.out.txt')
```

```
Classified Lines Accuracy: 96.0 %
Incorrectly classified lines: [24, 44, 92, 111, 162, 185, 187, 191, 247, 271, 277, 279]
```