Building a Language Model - Part I

Downloading Pretrained Models

Google drive Folders with all 3 Pretrained file: https://drive.google.com/drive/folders/17r9wsbGnEg_Sr5S_LlSBstKekHCFXjR6?usp=drive_link

In [ ]:
```
import xml.etree.ElementTree as ET
import random
import json
import math
import os
```

Preprocessing

Read and parse the data

In [ ]:
```
def split_corpus(directory, test_ratio=0.2):
    file_list = []
    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith(".xml"):
                file_list.append(os.path.join(root, file))

    num_test_files = int(len(file_list) * test_ratio)
    test_files = random.sample(file_list, num_test_files)
    train_files = [file for file in file_list if file not in test_files]
    return train_files, test_files

def parse_xml(xml_file):
    tree = ET.parse(xml_file)
    root = tree.getroot()
    sentences = []
    for sentence in root.findall(".//s"):
        words = []
        words.append("<s>")
        for wtext in sentence.findall(".//w"):
            if wtext.text is not None and wtext.text.strip().lower() != "":
                words.append(wtext.text.strip().lower())
        words.append("</s>")
        sentences.append(words)
    return sentences

def split_list(lst, delimiter):
    result = []
    sublist = []
    for item in lst:
        if item == delimiter:
            if sublist:
                result.append(sublist)
                sublist = []
        else:
            sublist.append(item)
    if sublist:
        result.append(sublist)
    return result

root_dir = os.path.join("British National Corpus, Baby edition", "Texts")
```

How large is the corpus that you are working with? What splits did you use?

Data is randomly distributed, with 80% of files for training and 20% for testing. You'll find more details on the data below.

How much space did your data structures require once all models were learnt?

My Vanilla model take 163 218 Ko, my Laplace model take 163 218 Ko and my Unk model take 154 243 Ko, for a total of 490 679 Ko.

In [ ]:
```
train_files, test_files = split_corpus(root_dir)

print("Number of files in training set:", len(train_files))
print("Number of files in test set:", len(test_files))

train_sentences = sum([parse_xml(xml_file) for xml_file in train_files], [])
test_sentences = sum([parse_xml(xml_file) for xml_file in test_files], [])
train_words = [word for sentence in train_sentences for word in sentence]
test_words = [word for sentence in test_sentences for word in sentence]

print("Number of word in training set:" , len(train_words))
print("Number of word in test set:", len(test_words))
print("Number of words in all corpus:", len(train_words) + len(test_words))
```

```
print("Number of unique words in training set:", len(set(train_words)))
print("Number of unique words in test set:", len(set(test_words)))
print("Number of unique words in all corpus:", len(set(train_words + test_words)))
```

```
Number of files in training set: 146
Number of files in test set: 36
Number of word in training set: 3675462
Number of word in test set: 992251
Number of words in all corpus: 4667713
Number of unique words in training set: 77012
Number of unique words in test set: 33993
Number of unique words in all corpus: 85768
```

Vanilla Language Model

The Vanilla Language model is your regular language model.

In [ ]:
```
class VanillaLanguageModel:
    def __init__(self, sentences: list[list[str]] = None, words: list[str] = None, model_data: dict = None):
        self.gram: dict[int, dict[tuple, int]] = {
            1: {},
            2: {},
            3: {},
        }
        self.words: list[str] = None
        if sentences is not None:
            self.words: list[str] = [word for sentence in sentences for word in sentence]
        elif words is not None:
            self.words: list[str] = words
        elif model_data is not None:
            self.load_model(model_data)
        else:
            raise ValueError("Either sentences or words or gram must be provided.")
        if self.words:
            self.words_count: int = len(self.words)
            self.words_vocabulary_count: int = len(set(self.words))

    def _count_ngrams(self, words: str, ngram: int):
        grams = self._tokenize(words, ngram)
        gram_counts: dict = {}
        for gram in grams:
            if gram in gram_counts:
                gram_counts[gram] += 1
            else:
                gram_counts[gram] = 1
        return gram_counts

    def _tokenize(self, words: str, ngram: int):
        ngrams = [tuple(words[i:i+ngram]) for i in range(len(words)-ngram+1)]
        return ngrams

    def _calculate_probability(self, ngram: int, word: str, context: tuple, log_format: bool = False):
        numerator = self.gram[ngram].get(context + (word,), 0)
        if ngram - 1 >= 1:
            denominator = self.gram[ngram - 1].get(context, 0)
        else:
            denominator = self.words_count
        if numerator == 0:
            return 0
        if log_format:
            return math.log(numerator / denominator)
        else:
            return numerator / denominator

    def calculate_sentence_probability_log(self, tokens: list, ngram: int):
        n = len(tokens)
        probability: float = 0.0

        for i in range(ngram-1, n):
            if ngram == 1:
                probability += self._calculate_probability(1, tokens[i], tuple(), True)
            else:
                if i < ngram - 1:
                    _prob = self._calculate_probability(i+1, tokens[i], tuple(tokens[:i]), True)
                    probability += _prob
                else:
                    _prob = self._calculate_probability(ngram, tokens[i], tuple(tokens[i-ngram+1:i]), True)
                    probability += _prob
        return probability

    def calculate_sentence_probability(self, tokens: list, ngram: int):
        n = len(tokens)
        probability: float = 1.0

        for i in range(ngram-1, n):
```

```
            if ngram == 1:
                probability *= self._calculate_probability(1, tokens[i], tuple())
            else:
                if i < ngram - 1:
                    _prob = self._calculate_probability(i+1, tokens[i], tuple(tokens[:i]))
                    probability *= _prob
                else:
                    _prob = self._calculate_probability(ngram, tokens[i], tuple(tokens[i-ngram+1:i]))
                    probability *= _prob
        return probability

    def _check_ngram(self, ngram: int):
        if ngram in self.gram:
            return True
        else:
            raise ValueError(f"Invalid ngram. Choose from 1 to {len(self.gram)}.")

    def train(self, ngram: int = 3):
        for i in range(1, ngram+1):
            self.gram[i] = self._count_ngrams(self.words, i)

    def get_top_ngrams(self, ngram, top_n=10):
        self._check_ngram(ngram)

        ngram_counts: dict = self.gram[ngram]

        return sorted(ngram_counts.items(), key=lambda x: x[1], reverse=True)[:top_n]
```

```
    def get_ngam_probability(self, ngram: int):
        ngram_probabilities = {k: self._calculate_probability(ngram, k[-1], k[:-1]) for k in self.gram[ngram].keys()}
        return ngram_probabilities

    def get_top_ngram_probability(self, ngram: int, top_n=10):
        ngram_probabilities = self.get_ngam_probability(ngram)
        return sorted(ngram_probabilities.items(), key=lambda x: x[1], reverse=True)[:top_n]

    def calculate_sentence_perplexity(self, words: list, ngram: int):
        self._check_ngram(ngram)

        log_sentence_probability = self.calculate_sentence_probability_log(words, ngram)
        perplexity = math.exp(-log_sentence_probability / len(words))
        return perplexity

    def calculate_perplexity(self, sentences: list, ngram: int) -> float:
        perplexities = []
        for sentence in sentences:
            perplexity = self.calculate_sentence_perplexity(sentence, ngram)
            perplexities.append(perplexity)
        average_perplexity = sum(perplexities) / len(perplexities)
        return average_perplexity

    def predict_next_word(self, context: tuple, ngram: int = 2):
        next_word_probs = {}
        context = tuple(context[-ngram+1:])
        for ngram_tuple, count in self.gram[ngram].items():
            if ngram_tuple[:-1] == context:
                next_word = ngram_tuple[-1]
                next_word_prob = self._calculate_probability(ngram, next_word, context)
                next_word_probs[next_word] = next_word_prob
```

```
        print(sorted(next_word_probs.items(), key=lambda x: x[1], reverse=True)[:5])
        if not next_word_probs:
            raise ValueError("No words found for the given context.")

        predicted_word = max(next_word_probs, key=next_word_probs.get)
        return predicted_word

    def gram_to_json(self):
        converted_dict = {}
        for key, inner_dict in self.gram.items():
            converted_inner_list = []
            for inner_key, value in inner_dict.items():
                converted_inner_list.append({"key": inner_key, "value": value})
            converted_dict[key] = converted_inner_list
        return json.dumps(converted_dict)

    def json_to_gram(self, json_str):
        loaded_dict: json = json.loads(json_str)
        reconstructed_dict: dict = {}
        for key, inner_list in loaded_dict.items():
            reconstructed_inner_dict = {}
            for item in inner_list:
                inner_key = item["key"]
                value = item["value"]
                reconstructed_inner_dict[tuple(inner_key)] = int(value)
            reconstructed_dict[int(key)] = reconstructed_inner_dict
```

```
        return reconstructed_dict

    def save_model(self, filename: str):
        model_data = {
            'gram': self.gram_to_json(),
            'words_count': self.words_count,
            'words_vocabulary_count': self.words_vocabulary_count,
        }
        with open(filename, 'w') as f:
            f.write(json.dumps(model_data))

    def load_model(self, model_data):
        if isinstance(model_data, str):
            with open(model_data, 'r') as f:
                model_data = json.loads(f.read())
        self.gram = self.json_to_gram(model_data['gram'])
        self.words_count = model_data['words_count']
        self.words_vocabulary_count = model_data['words_vocabulary_count']
```

How much time does it take to build the language models?

It takes around 25.5 seconds to build the language model.

In [ ]:
```
vanilla_model = VanillaLanguageModel(train_sentences)
vanilla_model.train()
# vanilla_model.save_model("vanilla_model.json")

print("Top 10 unigrams:", vanilla_model.get_top_ngrams(1))
print("Top 10 unigram probabilities:", vanilla_model.get_top_ngram_probability(1))
print("---"*10)
print("Top 10 bigrams:", vanilla_model.get_top_ngrams(2))
print("Top 10 bigram probabilities:", vanilla_model.get_top_ngram_probability(2))
print("---"*10)
print("Top 10 trigrams:", vanilla_model.get_top_ngrams(3))
print("Top 10 trigram probabilities:", vanilla_model.get_top_ngram_probability(3))
```

```
Top 10 unigrams: [(('<s>',), 256046), (('</s>',), 256046), (('the',), 171034), (('of',), 83002), (('to',), 76265), (('and',), 73501), (('a',), 70417), (('in',), 55618), (('it',), 45928), (('i',), 42350)]
Top 10 unigram probabilities: [(('<s>',), 0.06966362323974509), (('</s>',), 0.06966362323974509), (('the',), 0.046534013955252425), (('of',), 0.02258273925835718), (('to',), 0.02074977240956375), (('and',), 0.019997758104967484), (('a',), 0.019158679915613333), (('in',), 0.015132247320200834), (('it',), 0.01249584405987601), (('i',), 0.011522360998426864)]
------------------------------
Top 10 bigrams: [(('</s>', '<s>'), 256045), (('of', 'the'), 19120), (('<s>', 'the'), 16636), (('<s>', 'i'), 16021), (('in', 'the'), 14909), (('<s>', 'it'), 9290), (('<s>', 'he'), 9012), (('to', 'the'), 7655), (('it', "'s"), 7614), (('<s>', 'yeah'), 7228)]
Top 10 bigram probabilities: [(('maya', 'deren'), 1.0), (('deren', 'says'), 1.0), (('recourse', 'to'), 1.0), (('underscores', 'a'), 1.0), (('incongruities', 'into'), 1.0), (('depiction', 'of'), 1.0), (('iconographic', 'programmes'), 1.0), (('enrique', 'tord'), 1.0), (('visitas', 'or'), 1.0), (('inspections', 'intended'), 1.0)]
------------------------------
Top 10 trigrams: [(('</s>', '<s>', 'the'), 16636), (('</s>', '<s>', 'i'), 16021), (('</s>', '<s>', 'it'), 9290), (('</s>', '<s>', 'he'), 9012), (('</s>', '<s>', 'yeah'), 7228), (('</s>', '<s>', 'oh'), 6800), (('it', '</s>', '<s>'), 6680), (('</s>', '<s>', 'you'), 5966), (('</s>', '<s>', 'and'), 5921), (('yeah', '</s>', '<s>'), 5761)]
Top 10 trigram probabilities: [(('image', '</s>', '<s>'), 1.0), (('guy', 'brett', '</s>'), 1.0), (('brett', '</s>', '<s>'), 1.0), (('images', 'matter', 'to'), 1.0), (('involuntary', 'as', 'the'), 1.0), (('response', 'itself', '</s>'), 1.0), (('itself', '</s>', '<s>'), 1.0), (('answer', 'should', 'have'), 1.0), (('some', 'objective', 'quality'), 1.0), (('objective', 'quality', 'about'), 1.0)]
```

Laplace Language Model

The Laplace Language model will take the Vanilla as its basis, but you will now include Laplace smoothing.

In [ ]:
```
class LaplaceLanguageModel(VanillaLanguageModel):
    def __init__(self, sentences: list[list[str]] = None, words: list[str] = None, model_data: dict = None):
        super().__init__(sentences, words, model_data)

    def _calculate_probability(self, ngram: int, word: str, context: tuple, log_format: bool = False):
        numerator = self.gram[ngram].get(context + (word,), 0) + 1
        if ngram - 1 >= 1:
            denominator = self.gram[ngram - 1].get(context, 0) + self.words_vocabulary_count
        else:
            denominator = self.words_count + self.words_vocabulary_count
        probability = numerator / denominator
        return math.log(probability) if log_format else probability
```

How much time does it take to build the language models?

It takes around 13.5 seconds to build the language model.

In [ ]:
```
laplace_model = LaplaceLanguageModel(train_sentences)
laplace_model.train()
# laplace_model.save_model("laplace_model.json")

print("Top 10 Unigrams with Laplace Smoothing:", laplace_model.get_top_ngrams(1))
print("Top 10 unigram with Laplace Smoothing probabilities:", laplace_model.get_top_ngram_probability(1))
print("---"*10)
```

```
print("Top 10 Bigrams with Laplace Smoothing:", laplace_model.get_top_ngrams(2))
print("Top 10 bigram with Laplace Smoothing probabilities:", laplace_model.get_top_ngram_probability(2))
print("---"*10)
print("Top 10 Trigrams with Laplace Smoothing:", laplace_model.get_top_ngrams(3))
print("Top 10 trigram with Laplace Smoothing probabilities:", laplace_model.get_top_ngram_probability(3))
```

```
Top 10 Unigrams with Laplace Smoothing: [(('<s>',), 256046), (('</s>',), 256046), (('the',), 171034), (('of',), 83002), (('t
o',), 76265), (('and',), 73501), (('a',), 70417), (('in',), 55618), (('it',), 45928), (('i',), 42350)]
Top 10 unigram with Laplace Smoothing probabilities: [(('<s>',), 0.06823418363458347), (('</s>',), 0.06823418363458347), (('th
e',), 0.045579263174108604), (('of',), 0.022119540335256153), (('to',), 0.020324191453425126), (('and',), 0.01958761073361201
3), (('a',), 0.018765752940593326), (('in',), 0.014821954795689457), (('it',), 0.012239658422683274), (('i',), 0.01128615414790
349)]
------------------------------
Top 10 Bigrams with Laplace Smoothing: [(('</s>', '<s>'), 256045), (('of', 'the'), 19120), (('<s>', 'the'), 16636), (('<s>',
'i'), 16021), (('in', 'the'), 14909), (('<s>', 'it'), 9290), (('<s>', 'he'), 9012), (('to', 'the'), 7655), (('it', "'s"), 761
4), (('<s>', 'yeah'), 7228)]
Top 10 bigram with Laplace Smoothing probabilities: [(('</s>', '<s>'), 0.7687730065033718), (('of', 'the'), 0.1194957941180146
7), (('in', 'the'), 0.11241800497624972), (('on', 'the'), 0.06944153106989102), (('do', "n't"), 0.06830675447667774), (('yeah',
'</s>'), 0.06698130754208129), (('it', "'s"), 0.0619407841223361), (('it', '</s>'), 0.0543435822352367), (('<s>', 'the'), 0.049
95226056722853), (('to', 'the'), 0.049948785532075914)]
------------------------------
Top 10 Trigrams with Laplace Smoothing: [(('</s>', '<s>', 'the'), 16636), (('</s>', '<s>', 'i'), 16021), (('</s>', '<s>', 'i
t'), 9290), (('</s>', '<s>', 'he'), 9012), (('</s>', '<s>', 'yeah'), 7228), (('</s>', '<s>', 'oh'), 6800), (('it', '</s>', '<s
>'), 6680), (('</s>', '<s>', 'you'), 5966), (('</s>', '<s>', 'and'), 5921), (('yeah', '</s>', '<s>'), 5761)]
Top 10 trigram with Laplace Smoothing probabilities: [(('it', '</s>', '<s>'), 0.07982841848683267), (('yeah', '</s>', '<s>'),
0.0696120715692315), (('<s>', 'yeah', '</s>'), 0.05538936372269706), (('</s>', '<s>', 'the'), 0.04995241054834458), (('</s>', '
<s>', 'i'), 0.04810587977433292), (('you', '</s>', '<s>'), 0.03981048563057166), (('mm', '</s>', '<s>'), 0.038156021282441985),
(('that', '</s>', '<s>'), 0.03649534581123011), (('<s>', 'it', "'s"), 0.03501656972028458), (('i', 'do', "n't"), 0.034263326013
17628)]
```

UNK Language Model

The UNK Language model will take the Vanilla as its basis, but now you will set all the words that have a count of 2 or less as <UNK> tokens. And then recalculate accordingly. And recalculate Laplace smoothing on this model.

In [ ]:
```
class UNKLanguageModel(VanillaLanguageModel):
    def __init__(self, sentences: list[list[str]] = None, words: list[str] = None, model_data: dict = None):
        super().__init__(sentences, words, model_data)
        self.unk_threshold = 2
        if self.words:
            self._replace_rare_words()

    # Set all the words that have a count of 2 or less as <UNK> tokens.
    def _replace_rare_words(self):
        word_counts = {}
        for word in self.words:
            if word in word_counts:
                word_counts[word] += 1
            else:
                word_counts[word] = 1
        for i, word in enumerate(self.words):
            if word_counts[word] <= self.unk_threshold:
                self.words[i] = '<UNK>'

    def calculate_sentence_probability_log(self, tokens: list, ngram: int):
        for i, token in enumerate(tokens):
            if token not in self.gram[1] or self.gram[1][token] <= 2:
                tokens[i] = '<UNK>'
        return super().calculate_sentence_probability_log(tokens, ngram)

    # Recalculate Laplace smoothing on this model
    def _calculate_probability(self, ngram: int, word: str, context: tuple, log_format: bool = False):
        numerator = self.gram[ngram].get(context + (word,), 0) + 1
        if ngram - 1 >= 1:
            denominator = self.gram[ngram - 1].get(context, 0) + self.words_vocabulary_count
        else:
            denominator = self.words_count + self.words_vocabulary_count
        probability = numerator / denominator
        return math.log(probability) if log_format else probability
```

How much time does it take to build the language models?

It takes around 14 seconds to build the language model.

In [ ]:
```
unk_model = UNKLanguageModel(train_sentences)
unk_model.train()
# unk_model.save_model("unk_model.json")

print("Top 10 Unigrams after replacing rare words with <UNK>:", unk_model.get_top_ngrams(1))
print("Top 10 unigram after replacing rare words with <UNK> probabilities:", unk_model.get_top_ngram_probability(1))
print("---"*10)
print("Top 10 Bigrams after replacing rare words with <UNK>:", unk_model.get_top_ngrams(2))
print("Top 10 bigram after replacing rare words with <UNK> probabilities:", unk_model.get_top_ngram_probability(2))
print("---"*10)
print("Top 10 Trigrams after replacing rare words with <UNK>:", unk_model.get_top_ngrams(3))
```

```
print("Top 10 trigram after replacing rare words with <UNK> probabilities:", unk_model.get_top_ngram_probability(3))
```

```
Top 10 Unigrams after replacing rare words with <UNK>: [(('<s>',), 256046), (('</s>',), 256046), (('the',), 171034), (('of',), 83002), (('to',), 76265), (('and',), 73501), (('a',), 70417), (('in',), 55618), (('<UNK>',), 53542), (('it',), 45928)]
Top 10 unigram after replacing rare words with <UNK> probabilities: [(('<s>',), 0.06823418363458347), (('</s>',), 0.06823418363458347), (('the',), 0.045579263174108604), (('of',), 0.022119540335256153), (('to',), 0.020324191453425126), (('and',), 0.019587610733612013), (('a',), 0.018765752940593326), (('in',), 0.014821954795689457), (('<UNK>',), 0.01426871978326832), (('it',), 0.012239658422683274)]
------------------------------
Top 10 Bigrams after replacing rare words with <UNK>: [(('</s>', '<s>'), 256045), (('of', 'the'), 19120), (('<s>', 'the'), 16636), (('<s>', 'i'), 16021), (('in', 'the'), 14909), (('<s>', 'it'), 9290), (('<s>', 'he'), 9012), (('to', 'the'), 7655), (('it', "'s"), 7614), (('<s>', 'yeah'), 7228)]
Top 10 bigram after replacing rare words with <UNK> probabilities: [(('</s>', '<s>'), 0.7687730065033718), (('of', 'the'), 0.1194957941801467), (('in', 'the'), 0.11241800497624972), (('on', 'the'), 0.06944153106989102), (('do', "n't"), 0.06830675447667774), (('yeah', '</s>'), 0.06698130754208129), (('it', "'s"), 0.0619407841223361), (('it', '</s>'), 0.0543435822352367), (('<UNK>', '</s>'), 0.05105167210502934), (('<s>', 'the'), 0.04995226056722853)]
------------------------------
Top 10 Trigrams after replacing rare words with <UNK>: [(('</s>', '<s>', 'the'), 16636), (('</s>', '<s>', 'i'), 16021), (('</s>', '<s>', 'it'), 9290), (('</s>', '<s>', 'he'), 9012), (('</s>', '<s>', 'yeah'), 7228), (('</s>', '<s>', 'oh'), 6800), (('it', '</s>', '<s>'), 6680), (('<UNK>', '</s>', '<s>'), 6664), (('</s>', '<s>', 'you'), 5966), (('</s>', '<s>', 'and'), 5921)]
Top 10 trigram after replacing rare words with <UNK> probabilities: [(('it', '</s>', '<s>'), 0.07982841848683267), (('<UNK>', '</s>', '<s>'), 0.07965246904727759), (('yeah', '</s>', '<s>'), 0.0696120715692315), (('<s>', 'yeah', '</s>'), 0.05538936372269706), (('</s>', '<s>', 'the'), 0.04995241054834458), (('</s>', '<s>', 'i'), 0.04810587977433292), (('you', '</s>', '<s>'), 0.03981048563057166), (('mm', '</s>', '<s>'), 0.038156021282441985), (('that', '</s>', '<s>'), 0.03649534581123011), (('<s>', 'it', "'s"), 0.03501656972028458)]
```

Interpolation

This function takes one of the above 3 flavours of language models and calculate the probability of a sentence using the following lambdas:

- trigram = 0.6
- bigram = 0.3
- unigram = 0.1

In [ ]:
```
def linear_interpolation(language_model: VanillaLanguageModel, sentences: list, get_perplexity: bool = False):
    trigram_lambda = 0.6
    bigram_lambda = 0.3
    unigram_lambda = 0.1

    prob = []
    for sentence in sentences:
        trigram_log_prob = language_model.calculate_sentence_probability_log(sentence, 3)
        bigram_log_prob = language_model.calculate_sentence_probability_log(sentence, 2)
        unigram_log_prob = language_model.calculate_sentence_probability_log(sentence, 1)

        max_log_prob = max(trigram_log_prob, bigram_log_prob, unigram_log_prob)
        log_total_prob = math.log(
            math.exp(trigram_log_prob - max_log_prob) * trigram_lambda +
            math.exp(bigram_log_prob - max_log_prob) * bigram_lambda +
            math.exp(unigram_log_prob - max_log_prob) * unigram_lambda
        ) + max_log_prob

        if get_perplexity:
            total_prob = math.exp(-log_total_prob / len(sentence))
        else:
            total_prob = math.exp(log_total_prob)
        prob.append(total_prob)

    log_total_prob = sum(prob) / len(prob)

    return log_total_prob
```

In [ ]:
```
print("Vanilla model linear interpolation: ", linear_interpolation(vanilla_model, [["<s>", "i", "know", "him"]]))
print("Unk model linear interpolation: ", linear_interpolation(unk_model, [["<s>", "i", "know", "him"]]))
print("Laplace model linear interpolation: ", linear_interpolation(laplace_model, [["<s>", "i", "know", "him"]]))
```

```
Vanilla model linear interpolation:  2.1393906897869986e-05
Unk model linear interpolation:  1.4971198217767322e-05
Laplace model linear interpolation:  1.789328817918789e-07
```

Perplexity Evaluation

Take the test corpus, iterate through the sentences and calculate the probabilities for each sentence with every model. You will then use this to calculate the Perplexity.

For the Perplexity calculation, I don't know why my laplace Perplexity is so big because in all my other evaluation calculations the model looks good but not for the Perplexity. I've tried a lot of tests and changes with no better result.

In [ ]:
```
vanilla_unigram_perplexity = vanilla_model.calculate_perplexity(test_sentences, ngram=1)
vanilla_bigram_perplexity = vanilla_model.calculate_perplexity(test_sentences, ngram=2)
vanilla_trigram_perplexity = vanilla_model.calculate_perplexity(test_sentences, ngram=3)
vanilla_linear_interpolation = linear_interpolation(vanilla_model, test_sentences, get_perplexity=True)
```

```
laplace_unigram_perplexity = laplace_model.calculate_perplexity(test_sentences, ngram=1)
laplace_bigram_perplexity = laplace_model.calculate_perplexity(test_sentences, ngram=2)
laplace_trigram_perplexity = laplace_model.calculate_perplexity(test_sentences, ngram=3)
laplace_linear_interpolation = linear_interpolation(laplace_model, test_sentences, get_perplexity=True)

unk_unigram_perplexity = unk_model.calculate_perplexity(test_sentences, ngram=1)
unk_bigram_perplexity = unk_model.calculate_perplexity(test_sentences, ngram=2)
unk_trigram_perplexity = unk_model.calculate_perplexity(test_sentences, ngram=3)
unk_linear_interpolation = linear_interpolation(unk_model, test_sentences, get_perplexity=True)

print("|            | Unigram | Bigram | Trigram | Linear Interpolation |")
print("|------------|---------|--------|---------|----------------------|")
print(f"| Vanilla    | {vanilla_unigram_perplexity:.2f}   | {vanilla_bigram_perplexity:.2f}  | {vanilla_trigram_perplexity:.2f
print(f"| Laplace    | {laplace_unigram_perplexity:.2f}   | {laplace_bigram_perplexity:.2f}  | {laplace_trigram_perplexity:.2f
print(f"| UNK        | {unk_unigram_perplexity:.2f}   | {unk_bigram_perplexity:.2f}  | {unk_trigram_perplexity:.2f}  | {unk_li
```

```
|            | Unigram | Bigram | Trigram | Linear Interpolation |
|------------|---------|--------|---------|----------------------|
| Vanilla    | 1.00   | 1.00  | 1.00  | 1.00 |
| Laplace    | 3829982.00   | 23224.53  | 11553.88  | 11878.47 |
| UNK        | 77.47   | 29.10  | 88.75  | 30.94 |
```

In [ ]:
```
def remove_start_end_tokens(sentence: str):
    return sentence.replace("<s>", "\n").replace("</s>", "").strip().capitalize()

def generate_sentence(language_model: VanillaLanguageModel, starting_sentence: str, ngram: int = 3):
    sentence = starting_sentence.split()
    while sentence[-1] != '</s>':
        next_word = language_model.predict_next_word(tuple(sentence[-(ngram):]), ngram)
        sentence.append(next_word)
    return remove_start_end_tokens(' '.join(sentence))
```

Load model from file

Vanilla Language Model

In [ ]:
```
vanilla_model = VanillaLanguageModel(model_data="vanilla_model.json")
```

Laplace Language Model

In [ ]:
```
laplace_model = LaplaceLanguageModel(model_data="laplace_model.json")
```

UNK Language Model

In [ ]:
```
unk_model = UNKLanguageModel(model_data="unk_model.json")
```

Generation Evaluation

This function that allows the user to select one of the Language Models (Vanilla, Laplace, UNK) and input a phrase. The function will use this phrase to Generate the rest of the sentence until it encounters the end of sentence token (i.e. the LM says stop!)

Take a test input and generate a sentence using the different models.

I send to the input "I love you like my" and the output is:

```
[('</s>', 0.1), ('mum', 0.06666666666666667), ('other', 0.03333333333333333), ('leg', 0.03333333333333333),
('browny', 0.03333333333333333)]
Vanilla model generated sentence:  I love you like my
------------------------------
[('</s>', 5.570797877526009e-05), ('mum', 4.1780984081445065e-05), ('other', 2.7853989387630044e-05),
('leg', 2.7853989387630044e-05), ('browny', 2.7853989387630044e-05)]
Laplace model generated sentence:  I love you like my
------------------------------
[('</s>', 5.570797877526009e-05), ('mum', 4.1780984081445065e-05), ('<UNK>', 4.1780984081445065e-05),
('other', 2.7853989387630044e-05), ('leg', 2.7853989387630044e-05)]
UNK model generated sentence:  I love you like my
```

I don't know why the highest probability is the token "</s>" after "my", but if we look at the words the probability follows, we can see some good results.

In [ ]:
```
selected_model = input("Select a language model (Vanilla, Laplace, UNK or All): ").strip().lower()

model: VanillaLanguageModel
if selected_model == "vanilla":
    model = vanilla_model
elif selected_model == "laplace":
    model = laplace_model
elif selected_model == "unk":
```

```
    model = unk_model
elif selected_model == "all":
    model = None
else:
    raise ValueError("Invalid model selection.")

starting_phrase = input("Enter a starting phrase: ")
if model is None:
    for key, value in {
        "Vanilla": vanilla_model,
        "Laplace": laplace_model,
        "UNK": unk_model
        }.items():
        generated_sentence = generate_sentence(value, f"<s> {starting_phrase}", ngram=3)
        print(f"{key} model generated sentence: ", generated_sentence)
        print("---"*10)
else:
    generated_sentence = generate_sentence(model, f"<s> {starting_phrase}", ngram=3)
    print("Generated sentence: ", generated_sentence)
```

```
[('</s>', 0.1), ('mum', 0.06666666666666667), ('other', 0.03333333333333333), ('leg', 0.03333333333333333), ('browny', 0.033333333333333333)]
Vanilla model generated sentence:  I love you like my
------------------------------
[('</s>', 5.570797877526009e-05), ('mum', 4.1780984081445065e-05), ('other', 2.7853989387630044e-05), ('leg', 2.7853989387630044e-05), ('browny', 2.7853989387630044e-05)]
Laplace model generated sentence:  I love you like my
------------------------------
[('</s>', 5.570797877526009e-05), ('mum', 4.1780984081445065e-05), ('<UNK>', 4.1780984081445065e-05), ('other', 2.7853989387630044e-05), ('leg', 2.7853989387630044e-05)]
UNK model generated sentence:  I love you like my
------------------------------
```

Take a test sentence and output its probability using all the different models.

I've run a few tests, which you can see below, and the results are good for all models. We can see that if the vanilla model doesn't know a word combination, the probability is 0. But for the laplace and UNK models, the probability isn't 0 because of smoothing.

In [ ]:
```
def sen_probability(sentence: list, ngram: int = 3):
    key: str
    value: VanillaLanguageModel
    for key, value in {
        "Vanilla": vanilla_model,
        "Laplace": laplace_model,
        "UNK": unk_model
    }.items():
        probability = value.calculate_sentence_probability(sentence, ngram)
        print(f"{key} model probability for the sentence '{sentence}': ", probability)
sentence = ["<s>"] + "Tell me who".split() + ["</s>"]
sen_probability(sentence, 2)
sen_probability(["<s>", "i", "love", "you", "</s>"])
sen_probability(["<s>", "i", "love", "you", "like", "</s>"])
```

```
Vanilla model probability for the sentence '['<s>', 'Tell', 'me', 'who', '</s>']':  0.0
Laplace model probability for the sentence '['<s>', 'Tell', 'me', 'who', '</s>']':  1.614067697045491e-17
UNK model probability for the sentence '['<s>', 'Tell', 'me', 'who', '</s>']':  1.614067697045491e-17
Vanilla model probability for the sentence '['<s>', 'i', 'love', 'you', '</s>']':  0.0003703859137006264
Laplace model probability for the sentence '['<s>', 'i', 'love', 'you', '</s>']':  1.145100587095341e-10
UNK model probability for the sentence '['<s>', 'i', 'love', 'you', '</s>']':  1.145100587095341e-10
Vanilla model probability for the sentence '['<s>', 'i', 'love', 'you', 'like', '</s>']':  0.0
Laplace model probability for the sentence '['<s>', 'i', 'love', 'you', 'like', '</s>']':  4.2323546626946686e-15
UNK model probability for the sentence '['<s>', 'i', 'love', 'you', 'like', '</s>']':  4.2323546626946686e-15
```

Testing

The testing part should detail how you tested the different components throughout your project development, what results were obtained and what fixes, if any, were required.

One of the big problems I encountered was that calculating the probabilities of a long sentence in the test data set generated a "math error". So I tried to solve this problem, first by looking through the course slides to see if there was anything I was missing, and after a little while I found something about the log, so I searched the Internet for more information, and after a while I came across this article https://web.stanford.edu/~jurafsky/slp3/3.pdf, I read it and tried to understand it, after a while I did a lot of tests and changed my way of calculating the probability of a sentence. I tried using the logarithm of probability and got better results.

Another problem i encountered was that the perplexity of the laplace model was very high, so I tried to solve this problem by changing the way I calculated the perplexity, but I didn't get a better result. I tried to understand why this was happening, but I couldn't find the answer.