Building a Statistical Language Model for Maltese

ICS2203 Statistical Language Processing

Miguel Baldacchino — May 2025

Contents

1 Introduction

This project implements a statistical language model from scratch using a subsection of the Maltese corpus, covering Unigram, Bigram, and Trigram models. Three variants: Vanilla, Laplace and UNK were developed to support sentence generation, probability estimation, and perplexity evaluation using modular and scalable code [1].

2 Corpus Overview and Extraction

The 'Korpus Malti v3.0' [2] was used in this project.

2.1 Corpus Analysis

The first task included reviewing the contents of the corpus and planning the extraction. The corpus is made up of hundreds of files, each containing hundreds of thousands of lines, containing data as shown in Figure 1, in **.vrt** format.

```
<p id="p_2">
<s id="s_2">
Lesti   ADJ lesti   null
l-  DEF l-  null
għaġina NOUN    għaġina għ-ġ-n
ħelwa   ADJ ħelu    ħ-l-w
.   X-PUN   .   null
</s>

</p>
```

Figure 1: Raw Corpus Data

The corpus contains verticalized text, with each word annotated with part-of-speech amongst other metadata.

2.2 Corpus Extraction

All extraction code is reproduceable and provided in ***extraction.py***

```
# takes file by file
def vrtParser(filepath):
    # list storing all sentences
    sentences = []

    with open(filepath, 'r', encoding='utf-8') as file:
        # currently in sentence flag
        in_sentence = False
        # list storing current sentence
        current_sentence = []


        # iterating every line in file
        for line in file:
            # start of sentence
            if line.startswith("<s"):
                # removes spaces
                line = tokenize(line)
                in_sentence = True
                current_sentence = []
            # end of sentence
            elif line.startswith("</s>"):
                if current_sentence:
                    sentences.append(current_sentence)
                in_sentence = False
            # sentence contents
            elif in_sentence and not line.startswith("<"):
                word = line.split('\t')[0]  # Extract first
                    column
                if word:  # Avoid empty strings
                    current_sentence.append(lowercase(word))
    return sentences
```

The function ***vrtParser()*** is responsible for extracting the raw data required from the.vrt files. It works by taking a filepath as parameter, opening the file within the path, iterating through every line until it is met with a `'<s'` indicating a start of what we need. This sets an 'in_sentence' flag as True and extracts the first word from each line until the end of the sentence is met via `'</s>'`.

```
def loadCorpus(folder_path, max_files=None):
    parsed = []
    for i, file_name in enumerate(os.listdir(folder_path)):
        if max_files and i >= max_files:
            break
        file_path = os.path.join(folder_path, file_name)
        if os.path.isfile(file_path):
            parsed.extend(vrtParser(file_path))
            print(f'Parsed {i}: {file_name}')
    return parsed
```

This function ***loadCorpus()*** takes folder's path and 'max_files' as parameters, and works by iterating through the files within a folder, and uses the previously mentioned ***vrtParser()*** to extract the data.

```
def splitCorpus(corpus, train_size = 0.8):
    random.shuffle(corpus)

    train_len = int(len(corpus) * train_size)

    train_sentences = corpus[:train_len]
    test_sentences = corpus[train_len:]

    return train_sentences, test_sentences
```

The ***splitCorpus()*** function takes the corpus, and 'train_size' as parameters, and works to split the corpus into a train and test split, by randomly shuffling the corpus and returning the split corpus'. This function will be used down the line.

Initial exploration included counting the number of sentences and tokens and inspecting sample outputs to ensure correct extraction.

2.3 Corpus Size and Files Used

For practical reasons and resource constraints, a subset of 300 files from the Maltese Corpus was used in the main experiment. This subset provided a representative sample, ensuring manageable processing times and memory usage.

3 Preprocessing

As per preprocessing, a few functions were required for our intended usage. All preprocessing was performed from scratch without the use of preprocessing libraries.

All code is modular in ***preprocessing.py***.

```
def lowercase(text):
    # lowercases every character
    return text.lower()
```

The ***lowercase()*** function converts all characters in an input string into lowercase.

```
def removePunctuation(text):
    # all punctuation characters
    punctuation_chars = {'!', '"', '#', '$', '%', '&', '(',
        ')', '*', '+', ',',
                         '.', '/', ':', ';', '<', '=', '>', '?',
                             '@', '[', '\\', ']', '^',
                         '_', '`', '{', '|', '}', '~'}
    # current word being worked on
    word = ''

    for char in text:
        # filters out punct chars
        if char not in punctuation_chars:
            word += char
    return word
```

The ***removePunctuation()*** function removes predefined punctuation from a given text, iterating over every character, preserving language-specific characters like the hyphen (-) and the apostrophe (') used in Maltese orthography in words like: il-, lill-, ta', ma' etc.

```
def tokenize(text):
    return text.split()
```

The ***tokenize()*** function splits a given string into a list of tokens, using whitespace as a delimiter using Python's in-built method.

```
# corpus must be parsed through vrt parser first
def flattenCorpus(corpus):
    flatten_corpus = []
    for sentence in corpus:
        flatten_corpus.extend(sentence)  # correct: appends
            words as whole tokens
    return flatten_corpus
```

The ***flattenCorpus()*** function takes a list of tokenized sentences and flattens it into a single string, used when building the n-gram frequency counts across the corpus.

```
def preprocess(sentence):
    cleaned_sentence = []
    sentence = [removePunctuation(word) for word in sentence
        if tokenize(word) != '' and removePunctuation(word)
       != '']
        # skips empty lines
    if sentence:
        sentence = ['<s>'] + sentence + ['</s>']
        cleaned_sentence.append(sentence)

    return cleaned_sentence
```

The ***preprocess()*** function processes a tokenized sentence by removing punctuation, discarding empty tokens, and adding `'<s>'` and `'</s>'` indicating start and end of sentences, ensuring proper context handling without crossing sentence boundaries. It returns a list containing the cleaned sentence structure.

3.1 Code Structure and Modularity

Multiple choices were made to ensure modularity within this code structure. Each preprocessing step has its own dedicated set of functions., and no class dependencies present offered easier testing.

3.2 Challenges and Solutions

Hyphen Handling was an issue at first, since definite articles were losing the hyphen (-). This was solved by removing the hyphen from the hardcoded set of punctation being removed by ***removePunctuation()***.

Empty Tokens were showing up due to the tokenized punctuation which were being emptied by the ***removePunctuation()***. This was mitigated in ***preprocessing()*** via:

```
if tokenize(word) != '' and removePunctuation(word) != ''
```

4 N-Grams

As described in [3], "N-grams are contiguous sequences of 'n' items, typically words in the context of NLP. These items can be characters, words, or even syllables, depending on the granularity desired. The value of 'n' determines the order of the N-gram."

In this assignment, I implemented a manual N-gram counter from scratch without libraries, alongside a library-based version using Python's Counter, and compared their execution time and memory usage to evaluate efficiency and scalability.

The theory and notation used for n-grams in this report also follows the lecture notes [1].

4.1 Manual N-Gram

```
def buildNgrams(tokenized_sentence):
    unigram_count = {}
    bigram_count = {}
    trigram_count = {}

    for i in range(len(tokenized_sentence)):
        # Unigram
        unigram = tokenized_sentence[i]
        unigram_count[unigram] = unigram_count.get(unigram,
            0) + 1

        # Bigram (no crossing sentence boundaries)
        if i >= 1:
            if tokenized_sentence[i-1] != '</s>':
                bigram = (tokenized_sentence[i-1],
                    tokenized_sentence[i])
                bigram_count[bigram] = bigram_count.get(
                    bigram, 0) + 1

        # Trigram (no crossing sentence boundaries)
        if i >= 2:
            if (tokenized_sentence[i-2] != '</s>' and
                tokenized_sentence[i-1] != '</s>'):
                trigram = (tokenized_sentence[i-2],
                    tokenized_sentence[i-1],
                    tokenized_sentence[i])
                trigram_count[trigram] = trigram_count.get(
                    trigram, 0) + 1

    return {
        'Unigrams': unigram_count,
        'Bigrams': bigram_count,
        'Trigrams': trigram_count
```

```
    }
```

The ***buildNgrams()*** function constructs the frequency counts of unigrams, bigrams, and trigrams from a tokenized flattened corpus of sentences, ensuring n-grams only accept `</s>` only where valid. It iterates through each token and conditionally builds the bigrams and trigrams, with conditions in place to only do it on a per-sentence basis.

This design preserves syntactic coherence of the language model, via respecting the sentence structure. Resulting frequency dictionaries will later be used for estimating the probabilities in all three language models.

4.2 Library N-Gram

```
def buildLibraryNGrams(tokenized_sentence):
    unigrams = Counter(tokenized_sentence)

    # Bigram: skip if either token is </s>
    bigrams = Counter(
        (tokenized_sentence[i], tokenized_sentence[i+1])
        for i in range(len(tokenized_sentence)-1)
        if tokenized_sentence[i] != '</s>'
    )

    # Trigram: skip if any token is </s>
    trigrams = Counter(
        (tokenized_sentence[i], tokenized_sentence[i+1],
            tokenized_sentence[i+2])
        for i in range(len(tokenized_sentence)-2)
        if tokenized_sentence[i] != '</s>' and
           tokenized_sentence[i+1] != '</s>'
    )

    return {
        'Unigrams': dict(unigrams),
        'Bigrams': dict(bigrams),
        'Trigrams': dict(trigrams)
    }
```

The ***buildLibraryNGrams()*** function uses Python's Counter from the collections module. This is to determine the modules efficiency at computing unigrams, bigrams and trigrams' frequency counts. It uses a similar cross-sentence boundary check like the manual version, excluding sequences with an out-of-place `</s>`. This function should prove itself a more performance-optimized alternative due to the efficient use of the Counter. Speed and scalability will be compared to the manual implementation of the function

4.3 Performance Comparison

To evaluate the efficiency of the n-gram construction methods, both the manual and library-based (using `Counter`) implementations were benchmarked on the same corpus subset. The manual approach completed n-gram extraction in **82 seconds**, while the library-based implementation required **153 seconds**, as can be seen in Figure 2.

```
========== N-GRAM TIMING COMPARISON ==========
Manual N-Gram Time : 82.9834 seconds
Library N-Gram Time: 153.4361 seconds
Speedup (Library / Manual): 1.85x
```

Figure 2: N-Gram Time Comparison

The manual n-gram construction method proved to be approximately **1.85 times faster** than the library-based approach in this scenario. This is an unexpected result, as the `Counter` class from Python's `collections` module is typically considered highly optimized for frequency counting.

Although we might expect that Python's `Counter` would be faster because it is a built-in tool, in this case the manual method turned out to be much faster. This is because the library method has to do extra work, like checking conditions in every step of the list comprehensions, which slows it down. On the other hand, the manual method does everything in one loop and checks for sentence boundaries as it goes, so it is more direct and efficient for this particular problem.

5 Vanilla Language Model

The ***VanillaLanguageModel*** class serves as the foundation of the language modeling system, constructing probabilistic models from raw n-gram frequency counts, as explained in the course notes [1].

```
class VanillaLanguageModel:
    def __init__(self, ngrams, compute_probs=True):
        self.unigram_freqs = ngrams['Unigrams']
        self.bigram_freqs = ngrams['Bigrams']
        self.trigram_freqs = ngrams['Trigrams']

        if compute_probs:
            self.total_count = sum(self.unigram_freqs.values
                ())
            self.unigram_probs = self.computeUnigramProbs()
            self.bigram_probs = self.computeBigramProbs()
            self.trigram_probs = self.computeTrigramProbs()

    def computeUnigramProbs(self):
        return {word: count / self.total_count for word,
            count in self.unigram_freqs.items()}

    def computeBigramProbs(self):
        probs = {}
        for (w1, w2), count in self.bigram_freqs.items():
            if w1 not in probs:
                probs[w1] = {}
            w1_count = self.unigram_freqs.get(w1, 0)
            probs[w1][w2] = count / w1_count
        return probs

    def computeTrigramProbs(self):
        probs = {}
        for (w1, w2, w3), count in self.trigram_freqs.items
            ():
            context = (w1, w2)
            if context not in probs:
                probs[context] = {}
            bigram_count = self.bigram_freqs.get((w1, w2),
                0)
            probs[context][w3] = count / bigram_count
        return probs
```

Upon initialization, the class extracts and stores the unigram, bigram, and trigram frequency data. If 'compute_probs' flag is set to True, this computes the corresponding probability distributions using maximum likelihood estimation (MLE).

The ***computeUnigramProbs()*** normalizes each unigram's count over the total word count, estimating its probability in the corpus.

The ***computeBigramProbs()*** method normalizes by the count of the first word, i.e. $P(w_2 \mid w_1)$.

The ***computeTrigramProbs()*** method normalizes by the count of the preceding bigram context i.e. $P(w_3 \mid w_1, w_2)$

This form of hierarchical structure ensures that probability estimations are context-aware. This modular approach separates probability logic by n-gram level, making it easy to test.

5.1 Sentence Generation

```
    def wordChosen(self, probabilities_dict):
        words = list(probabilities_dict.keys())
        probabilities = list(probabilities_dict.values())

        if not words:
            return '</s>'
        return random.choices(words, weights=probabilities,
            k=1)[0]

    def generateSentence(self, max_length=15, input_string='
        '):

        if isinstance(input_string, str):
            input_tokens = tokenize(input_string)
            sentence = []
            for token in input_tokens:
                token = lowercase(token)
                token = removePunctuation(token)
                if token:
                    sentence.append(token)
            input_tokens = sentence
        else:
            # Assume already list of tokens, but still
                process for safety
            input_tokens = [removePunctuation(lowercase(
                token)) for token in input_string if
                removePunctuation(lowercase(token)) != '']

        sentence = ['<s>'] + input_tokens
        attempts = 0
        while len(sentence) < max_length and attempts < 100:
             # loop limit
            attempts += 1

            if len(sentence) < 2:
                context = sentence[-1]
                dict = self.bigram_probs.get(context, {})
            else:
```

```
                trigram_context = (sentence[-2], sentence
                    [-1])
                dict = self.trigram_probs.get(
                    trigram_context, {})
                if not dict:
                    # fallback to bigram using the last word
                    dict = self.bigram_probs.get(sentence
                        [-1], {})

            next_word = self.wordChosen(dict)

            if next_word.endswith('-'):
                continue
            if not next_word:
                break
            if next_word == '</s>':
                break
            sentence.append(next_word)

        return sentence[1:]
```

The ***generateSentence()*** constructs a sentence by probabilistically selecting the next word, based on trigram or bigram contexts, starting from an input string, or an input set of tokens. If an **empty string** is provided, the model defaults to generating a sentence from the start token <s>, allowing for unsupervised sentence creation. An attempt limiter is also present to prevent infinite loops during generation.

It proritizes trigram context for accuracy, but a ***Graceful Fallback Mechanism*** was added to fallback to bigram probabilities when trigram is unavailable, using the ***wordChosen()*** method.

Sentence Length was also added to constrain sentence length to a desired token count e.g. 15.

Sometimes runs would loop infinitely when attempting sentence generation, so an ***Infinite Loop Prevention*** system was added via an attempt counter, to prevent infinite loops from occuring.

5.2 Sentence Probability

```
def Sen_Probability(self, sentence):
        if isinstance(sentence, str):
            sentence = tokenize(sentence)
            sentence = [lowercase(token) for token in
                sentence]

        # add start/end tokens if needed
        if sentence[0] != '<s>':
            sentence = ['<s>'] + sentence
        if sentence[-1] != '</s>':
```

```
            sentence = sentence + ['</s>']

        prob = 1.0

        for i in range(len(sentence)):
            w1 = sentence[i - 2] if i >= 2 else None
            w2 = sentence[i - 1] if i >= 1 else None
            w3 = sentence[i]

            # try trigram first
            if w1 and w2:
                context = (w1, w2)
                prob_dict = self.trigram_probs.get(context,
                    {})
                p = prob_dict.get(w3, None)
                if p is not None:
                    prob *= p
                    continue

            # fallback to bigram
            if w2:
                prob_dict = self.bigram_probs.get(w2, {})
                p = prob_dict.get(w3, None)
                if p is not None:
                    prob *= p
                    continue

            # if both missing, fallback to very small value
            prob *= 1e-20

        return prob
```

This ***Sen_Probability()*** method computes overall probability of a given input sentence, passed via parameter. It tokenizes and normalizes input, if provided as string, with `</s>` and `<s>` checks. If not found, the method takes care of adding them.

The method is built upon the same structure as ***sentenceGeneration()***, since it is calculating the probability of the given sentence being generated. This means, prioritizing trigram probabilities on each word based on context, and graceful fallback to bigram probabilities if trigram is unavailable.

In cases where neither is found, a small fallback probability is used, to prevent multiplying by 0.

In the case the input is a string, the function takes care of tokenizing the input.

In the case where the input is void of proper sentence boundaries, the code takes care of adding them too.

6 Laplace Language Model

```
class LaplaceLanguageModel(VanillaLanguageModel):
    def __init__(self, ngrams):
        # inherited raw counts, will override probs of
            vanilla
        super().__init__(ngrams, compute_probs=False)

        self.N = sum(self.unigram_freqs.values())
        self.V = len(self.unigram_freqs)

        self.unigram_probs = self.computeUnigramProbsLaplace
            ()
        self.bigram_probs = self.computeBigramProbsLaplace()
        self.trigram_probs = self.computeTrigramProbsLaplace
            ()

    def computeUnigramProbsLaplace(self):
        denom = self.N + self.V
        probabilities = {}
        for w, count in self.unigram_freqs.items():
            probabilities[w] = (count + 1) / denom
        return probabilities

    def computeBigramProbsLaplace(self):
        probs = {}
        for (w1, w2), count in self.bigram_freqs.items():
            if w1 not in probs:
                probs[w1] = {}
            denom = self.unigram_freqs.get(w1, 0) + self.V
            probs[w1][w2] = (count + 1) / denom
        return probs

    def computeTrigramProbsLaplace(self):
        probs = {}
        for (w1, w2, w3), count in self.trigram_freqs.items
            ():
            context = (w1, w2)
            if context not in probs:
                probs[context] = {}
            denom = self.bigram_freqs.get(context, 0) + self
                .V
            probs[context][w3] = (count + 1) / denom
        return probs
```

The ***LaplaceLanguageModel*** class extends the ***VanillaLanguageModel*** by implementing Laplace smoothing, which handles zero frequency issues in unseen n-grams [1].

It overrides probability computation methods, whilst maintaining the raw frequency counts as before, obtained via inheritance. Each ***compute...Laplace()***

method adjusts the probability by adding 1 to the count of each n-gram and normalizing using the vocabulary size V.

Model robustness is increased using this smoothing technique.

6.1 Sentence Generation and Probability

Sentence Generation and Sentence Probability methods are directly inherited from the VanillaLanguageModel with no modifications required. This is because the core logic for traversing and predicting sequences remains valid, with only underlying probability values being updated, using Laplace Smoothing.

7 UNK Language Model

This language model also works with Vanilla as its basis, but now all words with a count of 2 or less will be replaced with an <UNK> token.

```
class UNKLanguageModel(LaplaceLanguageModel):
    def __init__(self, ngrams):
        super().__init__(ngrams)
        self.vocab = set(self.unigram_freqs.keys())  #
            define known vocab
```

The ***UNKLanguageModel*** extends the ***LaplaceLangaugeModel*** incorporating the explicit handling of unknown words [1].

```
class UNKLanguageModel(LaplaceLanguageModel):
    def __init__(self, ngrams):
        super().__init__(ngrams)
        self.vocab = set(self.unigram_freqs.keys())  #
            define known vocab
```

7.1 Handling Rare Words with <UNK>

To improve generalization and reduce the impact of low-frequency words, rare tokens will be replaced with the <UNK> token as previously said. This helps the language model handle unseen / infrequent words better.

```
def wordCounter(corpus):
    word_count = {}
    for word in corpus:
        if word in word_count:
            word_count[word] += 1
        else:
            word_count[word] = 1
    return word_count

def replaceRareWords(corpus, word_count):
    updated_corpus = []
    for sentence in corpus:
        new_sentence = []
        for word in sentence:
            if word_count.get(word, 0) <= 2:
                new_sentence.append('<UNK>')
            else:
                new_sentence.append(word)
        updated_corpus.append(new_sentence)
    return updated_corpus
```

The ***wordCounter()*** function calculates raw word frequencies across the corpus.

The ***replaceRareWords()*** function looks through each sentence and replaces rare words appearing twice or fewer times with <UNK>.

7.2 Sentence Generation

```
def generateSentence(self, max_length=15, input_string=''):
        if isinstance(input_string, str):
            input_tokens = tokenize(input_string)
            sentence = []
            for token in input_tokens:
                token = lowercase(token)
                token = removePunctuation(token)
                if token:
                    sentence.append(token)
            input_tokens = sentence
        else:
            input_tokens = [removePunctuation(lowercase(
                token)) for token in input_string if
                removePunctuation(lowercase(token)) != '']


        # replace unseen tokens with <UNK>
        input_tokens = [w if w in self.vocab else '<UNK>'
            for w in input_tokens]

        sentence = ['<s>'] + input_tokens
        attempts = 0
        while len(sentence) < max_length and attempts < 100:
            attempts += 1

            if len(sentence) < 2:
                context = sentence[-1]
                dict = self.bigram_probs.get(context, {})
            else:
                trigram_context = (sentence[-2], sentence
                    [-1])
                dict = self.trigram_probs.get(
                    trigram_context, {})
                if not dict:
                    dict = self.bigram_probs.get(sentence
                        [-1], {})

            next_word = self.wordChosen(dict)

            if next_word.endswith('-'):
                continue
            if not next_word:
                break
            if next_word == '</s>':
                break
            sentence.append(next_word)

        return sentence[1:]
```

This overridden ***generateSentence()*** enhances ssentence generation by explicitly replacing any unknown input tokens with the `<UNK>` token before processing. As before the input sentence is first preprocessed, but now also replacing unseen words with `<UNK>` (unseen words ar this stage satisfy the count ¡= 2 rule). The rest of the logic mirrors the base implementation, using trigram or bigram contexts to probabilistically select the next word, with graceful handling of unseen sequences, tolerance for an empty input string and attempt limiter to prevent infinite loops during generation.

7.3 Sentence Probability

```
def Sen_Probability(self, sentence):
        if isinstance(sentence, str):
            sentence = tokenize(sentence)
            sentence = [lowercase(token) for token in
                sentence]

        sentence = [w if w in self.vocab else '<UNK>' for w
            in sentence]

        # add start/end tokens if needed
        if sentence[0] != '<s>':
            sentence = ['<s>'] + sentence
        if sentence[-1] != '</s>':
            sentence = sentence + ['</s>']

        prob = 1.0

        for i in range(len(sentence)):
            w1 = sentence[i - 2] if i >= 2 else None
            w2 = sentence[i - 1] if i >= 1 else None
            w3 = sentence[i]

            # try trigram first
            if w1 and w2:
                context = (w1, w2)
                prob_dict = self.trigram_probs.get(context,
                    {})
                p = prob_dict.get(w3, None)
                if p is not None:
                    prob *= p
                    continue

            # fallback to bigram
            if w2:
                prob_dict = self.bigram_probs.get(w2, {})
                p = prob_dict.get(w3, None)
                if p is not None:
                    prob *= p
```

```
                    continue

                # if both missing, fallback to very small value
                prob *= 1e-20

        return prob
```

Implementation remained the same as prior models, with the addition of replacing unseen words with <UNK> in the input sentence. Since the corpus has already been extracted, preprocess to replace words with a count of ¡= 2 with <UNK>, then any words not found within the unigram_freqs.keys() will also satisfy this rule.

8 Linear Interpolation

```
    def linearInterpolation(self, sentence, l1=0.1, l2=0.3,
        l3=0.6, preprocessed=False):
        if not sentence:
            return 1e-20, [], [], [], 0
        if not preprocessed:
            # lowercase and tokenize first
            if isinstance(sentence, str):
                sentence = tokenize(lowercase(sentence))
            else:
                # assume already list of tokens; lowercase
                    each
                sentence = [lowercase(w) for w in sentence]
            cleaned = preprocess(sentence)
            if not cleaned:
                return 1e-20, [], [], [], 0
            # take the first (and only) cleaned sentence for
                scoring
            sentence = cleaned[0]
        # For UNK Version, add line below
        #sentence = [w if w in self.vocab else '<UNK>' for w
            in sentence]
        total_prob = 1.0
        uni_p = []
        bi_p = []
        tri_p = []
        addedS = 0
        if sentence[0] != '<s>':
            sentence = ['<s>'] + sentence
            addedS += 1
        if sentence[-1] != '</s>':
            sentence = sentence + ['</s>']
            addedS += 1
        for i in range(len(sentence)):
            w1 = sentence[i-2] if i >= 2 else None
            w2 = sentence[i-1] if i >= 1 else None
            w3 = sentence[i]

            uni = self.unigram_probs.get(w3, 1e-20)
            bi = self.bigram_probs.get(w2, {}).get(w3, 1e
                -20) if w2 else 0
            tri = self.trigram_probs.get((w1, w2), {}).get(
                w3, 1e-20) if w1 and w2 else 0

            uni_p.append(uni)
            bi_p.append(bi)
            tri_p.append(tri)
```

```
            total_prob *= (l1 * uni) + (l2 * bi) + (l3 * tri
                )

        return total_prob, uni_p, bi_p, tri_p, addedS
```

The ***linearInterpolation()*** method within every model calculates the probability of a sentence using a weighted combination of unigram, bigram and trigram probabilities. Ensuring sentence boundaries are present, iterating through each word through the given sentence to collect the n-gram probabilities, substituting a very small value of 1e-20 if not found (i.e. unseen), to avoid any *= 0.

It is sure to first preprocess the given sentence accordingly using previously written preprocessing functions. The modularity of the preprocessing functions allows them to be used virtually all throughout the assignment.

At each position, the weighted sum of each n-gram probability is multiplied by the total probability.

Returned are the linear interpolation, n-gram probabilities (which will be used when calculating perplexity) and a value from 0 to 2 for whether we added sentence boundaries or not, used in perplexity also.

```
Available models:
  1. Vanilla
  2. Laplace
  3. UNK
Select a model by number: 1

Operations:
  1. Generate Sentence
  2. Sentence Probability
  3. Linear Interpolation
  4. Perplexity (on test set)
  5. Exit
Select an operation by number: 3
Enter your sentence: il- bniedem ġie maħluq minn Alla, fuq xbieha tiegħu.

Linear Interpolation Probability: 2.109023795477284e-20
```

Figure 3: Vanilla Model Linear Interpolation

The outcome shows how the Vanilla Model assigned a much higher probability to the same sentence, when compared to Laplace and UNK.

This is due to the fact that Vanilla model uses actual counts from the training data, not penalizing unseen or uncommon words.

This is in contrast to Laplace's smoothing, which adds 1 to all counts, even unseen ones. This results in the spreading of the probability across the board and hence, lower probabilities for any specific sentence.

The UNK model lowers the probability if it encounters any unknown or uncommon words, since it replaces them with `<UNK>`.

Overall, this shows how smoothing and unknown/uncommon word handling makes the model more robust to unseen data, but significantly lowers probability for any given sentence.

```
Available models:
  1. Vanilla
  2. Laplace
  3. UNK
Select a model by number: 2

Operations:
  1. Generate Sentence
  2. Sentence Probability
  3. Linear Interpolation
  4. Perplexity (on test set)
  5. Exit
Select an operation by number: 3
Enter your sentence: il- bniedem ġie maħluq minn Alla, fuq xbieha tiegħu.

Linear Interpolation Probability: 5.214934082265585e-36
```

Figure 4: Laplace Model Linear Interpolation

```
Available models:
  1. Vanilla
  2. Laplace
  3. UNK
Select a model by number: 3

Operations:
  1. Generate Sentence
  2. Sentence Probability
  3. Linear Interpolation
  4. Perplexity (on test set)
  5. Exit
Select an operation by number: 3
Enter your sentence: il- bniedem ġie maħluq minn Alla, fuq xbieha tiegħu.

Linear Interpolation Probability: 5.861710003720246e-34
```

Figure 5: UNK Model Linear Interpolation

.

9 Evaluation

9.1 Perplexity Evaluation

```
    def perplexity(self, corpus):
        prob_unigram = 0.0
        total_uni = 0
        prob_bigram = 0.0
        total_bi = 0
        prob_trigram = 0.0
        total_tri = 0
        prob_interpolation = 0.0
        total_inter = 0

        for sentence in corpus:
            prob, uni, bi, tri, s = self.linearInterpolation
                (sentence)
            for probab in uni:
                prob_unigram += math.log(probab if probab >
                    0 else 1e-20)
            for probab in bi:
                prob_bigram += math.log(probab if probab > 0
                     else 1e-20)
            for probab in tri:
                prob_trigram += math.log(probab if probab >
                    0 else 1e-20)
            prob_interpolation += math.log(prob if prob > 0
                else 1e-20)
            total_uni += len(uni)
            total_bi += len(bi)
            total_tri += len(tri)
            total_inter += len(sentence) + s

        log_probs = [prob_unigram, prob_bigram, prob_trigram
            , prob_interpolation]
        total_words = [total_uni, total_bi, total_tri,
            total_inter]

        perplexity = []
        for i in range(4):
            if total_words[i] == 0:
                perplexity.append(float('inf'))
            else:
                perplexity.append(math.exp(-log_probs[i]/
                    total_words[i]))
        return {
            'Unigram': perplexity[0],
            'Bigram': perplexity[1],
```

```
            'Trigram': perplexity[2],
            'Linear Interpolation': perplexity[3]
        }
```

This function above computes the perplexity of a corpus for unigram, bigram, trigram and linear interpolation models, by summing the log probabilities of each sentence, then averaging them per word, exponentiating the negative average and returning as a measure of how well each of the corpus' predict the given corpus.

```
Perplexity Results:
Model | Unigram | Bigram | Trigram | Interpolation | Time (s) | RAM (MB)
-----------------------------------------------------------------------------------------
Vanilla | 2397.564545315026 | 178373.29975100825 | 23854712173.69995 | 362.461126745682 | 411.18   | -1763.21
Laplace | 2415.151091876907 | 2873827.2538388977 | 1029640560893.1051 | 4786.409562944435 | 1016.19  | -183.56
UNK | 1446.2203223350107 | 373187.21409527026 | 193880875710.21014 | 2227.384151075251 | 343.78   | 1497.74
```

Figure 6: Perplexity of all Models on the Test Corpus

The perplexity results on the 20% test corpus provide insight on the significant differences across the models and n-gram orders.

For unigrams, all models achieve a relatively low perplexity, with UNK performing best at 1446. This indicates better handling of the rare and unseen words.

As we move up in n-gram order, the perplexity increases rapidly, especially for the trigram models. This comes from the strictness of longer contexts.

The Laplace model gave the highest perplexity results across the board, with its trigram results soaring at the 13-figure mark, when compared to UNK's 12-figure mark. UNK consistently produces lower scores for bigrams, trigrams and interpolation, clearly showing that the rare word replacement improved robustnesss and predective performance.

The lowest interpolation scoress for perplexity (362 for Vanilla, 4786 for Laplace, 2227 for UNK) further highlights that by combining the n-gram probabilities via interpolation, a more stable model is offered, mitigating weaknesses from higher-order models.

9.2 Generation Evaluation

As per previous explanation, the sentence generation in each model can handle both an input string and an empty input string.

Example 1: Empty Input

To illustrate the behaviour of the different language models, the below Figures present the sentences generated when using an empty string as input.

```
Enter your sentence:

Generated sentence:
['din', 'hija', 'gidba', 'sfaċċata']
```

Figure 7: Vanilla Model: Empty Input

```
Enter your sentence:

Generated sentence:
['provi', 'u', 'jaghmlu', 'rapport', 'hemmhekk']
```

Figure 8: Laplace Model: Empty Input

```
Enter your sentence:

Generated sentence:
['munita', 'ohra', 'dik', "ta'", 'san', 'ġużepp', "ta'", 'ħal', 'għaxaq', 'biżbud', 'u', 'ġwann', 'għalkemm', 'hu']
```

Figure 9: UNK Model: Empty Input

These outputs show that while Vanilla and Laplace produce more coherent sentences, UNK produces outputs that are less fluent, yet longer.

Example 2: Partial Input

The below outputs come from the given partial sentence 'id- dinja jehtieġ li nżommuha nadifa u'

```
Enter your sentence: id- dinja jeħtieġ li nżommuha nadifa u

Generated sentence:
['id-', 'dinja', 'jeħtieġ', 'li', 'nżommuha', 'nadifa', 'u', 'li', 'dejjem', 'għandhom', 'fuqhom', 'responsabbiltajiet', 'ferm', 'ikbar']
```

Figure 10: Vanilla Model: Partial Input

```
Enter your sentence: id- dinja jeħtieġ li nżommuha nadifa u

Generated sentence:
['id-', 'dinja', 'jeħtieġ', 'li', 'nżommuha', 'nadifa', 'u', 'poġġiha', 'fuq', 'nar', 'baxx', 'u', 'abużiv', 'u']
```

Figure 11: Laplace Model: Partial Input

```
Enter your sentence: id- dinja jeħtieġ li nżommuha nadifa u

Generated sentence:
['id-', 'dinja', 'jeħtieġ', 'li', 'nżommuha', 'nadifa', 'u', 'timplimenta', 'strateġiji', "ta'", 'tagħlim', 'li', 'tgħix', "ma'"]
```

Figure 12: UNK Model: Partial Input

Each model produced a continuation which is grammatically plausible and contextually related. Different levels of coherence are present, but are to be expected from a Statistical Language Model.

Vanilla and Laplace models extend the prompt longer with sometimes repetitive endings, whilst UNK model introduces new concepts which may not be linked with the original context.

This shows how n-gram models way of generating sentences is based on the preceding context of the word, and how the choice of model and smoothing affects output.

It is also important to note the method used to choose the next word during sentence generation. Instead of always picking the word with the highest probability, the model selects the next word using weighted random sampling based on the probability distribution. This approach encourages more varied and natural sentence generation by avoiding overly predictable, greedy outputs.

10 Modularity and Efficiency

The codebase has been structured with modularity in mind, dividing key functionalities such as preprocessing, n-gram extraction, model construction and evaluation into separate, reusable components.

Each major task is handled using its own function or class, making testing easier and maintaining/extending any part with ease.

Performance considerations were also taken, ensuring that the n-gram extraction and probability can scale to larger datasets without significant slowdowns.

10.1 Probability Evaluation

The test sentence 'id- dinja tibqa ddur, anke jekk ma tridx' was used to test each of the 3 models.

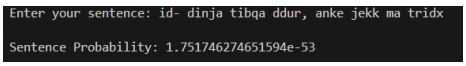

Figure 13: Vanilla Model: Sentence Probability

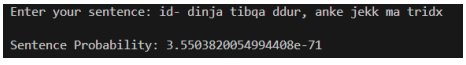

Figure 14: Laplace Model: Sentence Probability

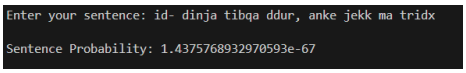

Figure 15: UNK Model: Sentence Probability

The Figures above depict the sentence probability output of the test sentence, for each model: Vanilla, Laplace and UNK.

These values are so small, since they are the joint probabilities of the entire sentence, under each model. It works by multiplying many small n-gram probabilities together (mostly trigrams, fallback to bigram otherwise) resulting in an exponentially small output as sentence length increases.

Vanilla has highest probability, suggesting some of the trigram/bigram sequences were likely seen in training. However, it is sensitive to unseen n-grams, resulting in even lower probability if found to be smaller.

Laplace has the lowest probability. This is due to the +1 to every count, which dilutes the probability mass across all n-grams. As a result, sequences might get lower probabilities than in Vanilla.

UNK sits in between. This intermediate probability suggests it handled the unkowns better than Laplace but not as confidently as Vanilla, for this particular sentence.

11 Resource Usage and Scalability

Resource usage was monitored throughout the development and evaluation processes, with special attention to memory consumption and execution times.

12 Testing and Validation

Many different tests and validations were made for this project, to ensure proper outputs. **Examples of Tests Performed:**

- **Preprocessing Verification:** Tested the `removePunctuation` and `tokenize` functions on sentences with mixed punctuation and casing to ensure correct tokenization and punctuation removal, including special Maltese characters and retained hyphens/apostrophes.
- **N-Gram Count Accuracy:** Compared manual and library-based n-gram extraction on a small, known sentence to verify that unigram, bigram, and trigram counts matched expectations and did not cross sentence boundaries.
- **Sentence Probability Consistency:** Calculated sentence probabilities for a fixed example sentence across all models to confirm that probabilities were nonzero and reflected the correct model behaviour (Vanilla, Laplace, and UNK).

- **Sentence Generation:** Ran the `generateSentence` function with both empty and partial input strings, ensuring the output was fluent, started with the given prompt, and did not loop indefinitely.
- **UNK Token Handling:** Checked that rare and unseen words were correctly replaced by `<UNK>` tokens in the UNK model and that this affected generation and probability calculation as expected.

Additional testing included the following:

- **Manual validation**: Probabilities, n-gram extractions, and sentence processing outputs were verified using a small, controlled testing corpus. Outputs were manually calculated and compared to ensure consistency and correctness in implementation.
- Further testing considerations and outcomes were discussed throughout the document in context with each section's functionality.

13 Discussion and Analysis

13.1 Corpus Statistics

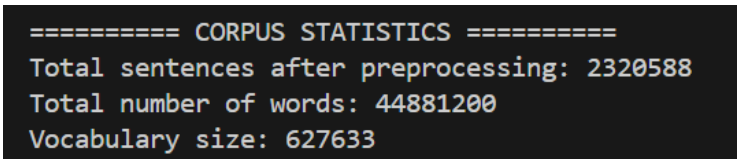

Figure 16: Sentence Count, Word/Token Count, Vocabulary Size

The processed subset of the Maltese corpus used for training and evaluation consisted of 300 files, containing approximately **2,320,588** sentences and **44,881,200** total tokens (words). After preprocessing, the final vocabulary size was reduced to **627,633**, with rare words replaced with the `<UNK>` token for the UNK model.

This data was then split into training and testing, and an 80/20 split seemed sufficient as it provides enough data for the model to train on, and sufficient enough data for the models to be tested on without any bias that emerges form small testing data.

13.2 Time, Space, and Data Usage

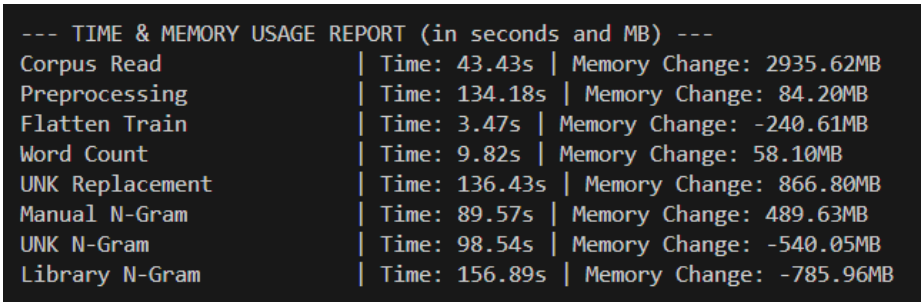

Figure 17: Corpus Execution Time and Memory

Resource monitoring was performed at every major processing step, tracking execution time and memory cosnuption. As can be shown in Figure 14, reading the full set of extracted filed, consumed the most memory at almost 3GB and took 43s. Preprocessing and UNK replacements were also memory-intensive but managable, with each step taking about 2 minutes and less than 1GB of additional memory.

N-Gram construction was tested for both manual and library-based. Manual was completed in 90s with a memory change of 490MB, whilst library was slower

```
N-Gram Timing Comparison:
Manual  N-Gram Time: 89.5671 seconds
Library N-Gram Time: 156.8942 seconds
Speedup (Library / Manual): 1.75x
```

Figure 18: N-Gram Execution Time and Memory

at 157s and reduced memory usage since it has more frequent object discarding. Yet, manual method proved to be more time-efficient.

```
Building Vanilla Model...
Saved vanilla_model.pkl
Vanilla model saved. Time: 656.58s | Memory: -2084.85MB

Building Laplace Model...
Saved laplace_model.pkl
Laplace model saved. Time: 846.29s | Memory: -1151.97MB

Building UNK Model...
Saved unk_model.pkl
UNK model saved. Time: 419.65s | Memory: 1537.74MB
```

Figure 19: Models Execution Time and Memory

Models took the longest to process, ranging from around 400-850s. The Vanilla model was built in 657s with a net memory reduction of 2GB. The Laplace model, which now includes the additional smoothing took longest to construct at 846s and a memory decrease of 1.1GB. The UNK Model was built most efficiently, completing in 420s and 1.5GB of additional memory, likely due to the `<UNK>` token handling.

The n-gram, model and preprocessing outputs took up a total of 3.96GB.

13.3 Additional Features

- **Resource monitoring:** I tracked execution time and memory usage for each major part (like preprocessing, n-gram counting, and model building) to see how the code scales.
- **Manual vs Library n-gram comparison:** I benchmarked both my manual and the Python `Counter`-based n-gram counting functions for speed and memory, and showed the difference using plots.
- **Weighted random sentence generation:** Instead of always picking the highest probability, I used weighted random sampling to make sentence outputs less repetitive and more varied.
- **Robust input handling:** My functions work even if the input is empty, partial, or contains unknown words, and they automatically add sentence start/end tokens if needed.

- **Infinite loop prevention:** I added a limit to how many times the generation loop can run so the code doesn't get stuck generating endless sentences.
- **Context fallback:** When generating or scoring sentences, the code tries trigrams first, then falls back to bigrams or unigrams if necessary.
- **Maltese language support:** Preprocessing keeps Maltese-specific punctuation (like hyphens and apostrophes), which is important for the corpus.

14 AI Usage and Prompts

As per the Generative AI usage policy, ChatGPT [4] was used to assist with debugging, improving modularity, verifying test logic, resolving implementation errors, and providing insight on technical questions encountered during development. Below are selected prompts and responses that contributed to the design and implementation of this project.

Prompt 1: Extremely Small Probability

Prompt:

```
probability:  8.765709257836653e-233
Is this too small of a probability?
```

Response:

Yes — that is an extremely small probability, but it is expected when computing joint probabilities over full sentences in language models.

You are multiplying many small values (e.g., trigram or bigram probabilities), and even moderately sized sentences lead to exponentially small products. For example, multiplying 14 probabilities averaging 0.01 gives approximately 10^{-28}. Trigram values are often smaller, so values like 10^{-233} are not unusual. This is not a bug — just an artifact of probability scaling.

Prompt 2: AttributeError from Unigram Probabilities

Prompt:

```
AttributeError:  'int' object has no attribute 'get'
Line:  total_prob *= l1 * self.unigram_probs.get(word, 0.000001)
```

Response:

This error occurs because `self.unigram_probs` was assigned as:

```
self.unigram_probs = sum(self.unigram_freqs.values())
```

This makes it an `int` (the total count of unigrams), not a dictionary. You cannot call `.get()` on an integer.

To fix this, construct `self.unigram_probs` as a proper probability dictionary:

```
 self.unigram_total = sum(self.unigram_freqs.values())
self.unigram_probs = {word:  count / self.unigram_total
    for word, count in self.unigram_freqs.items()}
```

Prompt 3: Sentence Generation Quality

Prompt:

`Is the generateSentence using the model well?`

Response:

Yes — your `generateSentence` function is well designed for a vanilla n-gram model. It:

- Prioritizes trigram probabilities.
- Falls back to bigram probabilities when necessary.
- Terminates sentences on `</s>` properly.
- Skips tokens with extremely low probability and avoids infinite loops.

This is a smart and practical implementation for a basic model.

Prompt 4: Is the Vanilla Model Incorrect?

Prompt:

`Am I doing the basic model wrong then?  Or is it OK?`

Response:

Your VanillaLanguageModel is functionally correct, but has typical limitations. Here's a breakdown:

Strengths:

- Builds unigram, bigram, and trigram probability tables.
- Supports sentence generation using fallback logic.
- Multiplies n-gram probabilities to compute sentence likelihoods.

Limitations:

- No smoothing — unseen n-grams get zero probability.
- No handling for unknown words or vocabulary limits.

So the model is not "wrong," just basic — and correctly implements the unsmoothed version.

Prompt 5: AttributeError in Laplace Model

Prompt:

```
AttributeError:  'list' object has no attribute 'lower'
Line:  token = lowercase(input_tokens)
```

Response:

The error occurs because `input_tokens` is a list. You're trying to apply `lower()` to the entire list, which fails.

Instead, apply `lower()` to each string element individually inside a loop or comprehension. Also, double-check that you're not accidentally appending the string `'token'` instead of the variable content.

15 Links to Models

Link for Downloading Models, Preprocessed Corpus Etc. can be found here: https://drive.google.com/drive/folders/1H-G-pAftTsE8vEAvC0P2g5CLPmF71TkD?usp=sharing

For testing, use the testingEverything.py file, since it loads the models once and you may test any feature of any model on loop.

References

[1] C. Borg, "Ics2203 statistical natural language processing: Lecture notes," Virtual Learning Environment, University of Malta, 2025, accessed: May 2025. [Online]. Available: https://vle.um.edu.mt/

[2] A. Gatt and S. Čéplö, "Digital corpora and other electronic resources for maltese," in *Proc. Corpus Linguistics Conference*. UCREL, Lancaster University, 2013, pp. 96–97.

[3] A. Jain, "N-grams in nlp," Medium [Online], Feb. 2024, available: https://medium.com/@abhishekjainindore24/n-grams-in-nlp-a7c05c1aff12, Accessed on: May 15, 2025.

[4] OpenAI. (2024) Gpt-4 technical report. Accessed: May 2025. [Online]. Available: https://openai.com/research/gpt-4