Documentation

May 6, 2024

1 Language Models Documentation

1.1 Project Structure

The project is split into 3 directories:

- data - contains the british corpus and the training and test data sets.
- documentation - contains any files used for testing purposes or related to the documentation
- src - contains the source code of the project.

Within src there is a file for the abstract base class of a language model, a file for each language model class, the main file *main.py*, and a file containing all the functions created used to manipulate the data sets, *dataset_functions.py*. There is also a folder containing all the n_gram counts for each model and the corpus counts in JSON format.

1.2 Part 1 - Generating Counts

Initially for the first part of this project I will be generating counts for the corpus as a whole and will not be splitting the data set, however, this process is designed in a way that it can extract counts from any xml file of similar structure. Thus, later in the project when I am training the models on a training set I was able to reuse the same functions. These corpus counts can be found under *src/n_grams/corpus*.

In my implementation I have opted to store the generated counts in JSON files so that this expensive computational section can be skipped in testing. I do this by checking if the JSON files exist, if they do I simply load the counts from the json into a dictionary and if they dont I create them. Due to this choice I am storing the counts in dictionaries, with the n_gram in a string as a key and the count as the value. Later on when I calculate and generate probabilities I will key the dictionary with tuples containing each token of the ngram rather than 1 string.

Generating the corpus counts is done using the following 4 functions:

- generate_corpus_counts()
- traverse_tree(node, number_of_words, counts)
- handle_sentence(sentence_node, number_of_words, counts)
- retrieve_text(node)

1.2.1 generate_corpus_counts()

This function starts by parsing either a single or multiple *.xml* files which contain the training data/corpus for this project. Once the root of each *.xml* file has been found, the function iterates over a loop three times, generating a frequency count for each n_gram *(unigram, bigram, trigram)*. This is achieved by calling the traverse_tree() function for each of the roots children.

Once the frequency count dictionary has been created, it is stored in a json file so that in the future this process doesn't need to be repeated everytime the language model is to be used.

1.2.2 traverse_tree(node, number_of_words, counts)

This function takes 3 parameters:

- node (xml.etree.ElementTree.Element): The current node in the XML tree.
- number_of_words (int): The number of words in the n-grams to be counted.
- counts (collections.defaultdict): A dictionary to store the n-gram counts.

This function is a recursive depth first function that traverses the tree until it finds either a *teiHeader* tag or an *s* tag. Due to the structure of the *.xml* files in the corpus, the first child of the root (*teiHeader*) contains information about the text and its origins. This is irrelevant to our frequency counts so we simply ignore it and keep searching the tree until an *s* tag is found. Once found the handle_sentence() function is called.

1.2.3 handle_sentence(sentence_node, number_of_words, counts)

This function takes 3 parameters:

- sentence_node (xml.etree.ElementTree.Element): The current sentence node in the XML tree.
- number_of_words (int): The size of the ngram counts to be extracted.
- counts (collections.defaultdict): A dictionary in which to store the n-gram counts.

The function calls retrieve_text() and recieves the sentence in string format, processes each sentence by stripping it and adding opening and closing sentence tokens, and then computes and updates n-gram frequencies stored in the counts variable. It will only update n-gram frequencies if the sentence contained words, if no words were present then it moves onto the next sentence.

1.2.4 retrieve_text(node)

This function takes one parameter:

- node (xml.etree.ElementTree.Element): The current node in the XML tree.

It returns a str, which represents the concatenated text from the node and its descendants. It does this by recursively checking all of the nodes children for word tags and ignores punctuation in the process.

[ ]:
```python
import os
import json
import xml.etree.ElementTree as ET
from collections import defaultdict

directories = ['aca', 'dem', 'fic', 'news']
BASE_PATH = '../data/corpus/Texts/'
```

```python
def generate_corpus_counts():
    for number_of_words in range(1, 4):
        n_gram_counts = defaultdict(int)

        for directory in directories:
            dir_path = os.path.join(BASE_PATH, directory)
            for file in os.listdir(dir_path):
                if file.endswith('.xml'):
                    file_path = os.path.join(dir_path, file)
                    tree = ET.parse(file_path)
                    root = tree.getroot()
                    for child in root:
                        if child.tag != 'teiHeader':
                            traverse_tree(child, number_of_words, n_gram_counts)

        with open(f'n_grams/corpus/{number_of_words}_gram_counts.json',
                  'w', encoding='utf-8') as fp:
            json.dump(n_gram_counts, fp, indent=4)

def traverse_tree(node, number_of_words, counts):
    """ Recursively traverses the XML tree to find sentences and process their
    text for n-gram frequency calculation.

    Parameters:
    node (xml.etree.ElementTree.Element): The current node in the XML tree.
    number_of_words (int): The number of words in the n-grams to be counted.
    counts (collections.defaultdict): A dictionary to store the n-gram counts.

    Returns:
        None
    """
    for child in node:
        if child.tag == 's':
            handle_sentence(child, number_of_words, counts)
        else:
            traverse_tree(child, number_of_words, counts)

def handle_sentence(sentence_node, number_of_words, counts):
    """ Processes each sentence to compute and update n-gram frequencies.

    Parameters:
    sentence_node (xml.etree.ElementTree.Element): The current sentence node in␣
  ↪the XML tree.
    number_of_words (int): The number of words in the n-grams to be counted.
    counts (collections.defaultdict): A dictionary to store the n-gram counts.

    Returns:
        None
    """
    text = retrieve_text(sentence_node)
    if text.strip() != "":
        text = ("<s> " * number_of_words) + text + (" </s>")
        words = text.split()
        for index in range(len(words) - number_of_words + 1):
            if number_of_words == 1:
                n_gram = words[index]
            else:
                n_gram = " ".join(words[index:index + number_of_words])
            counts[n_gram] += 1

def retrieve_text(node):
    """ Extracts and concatenates text from XML nodes, adding start and end
    markers to each sentence.

    Parameters:
    node (xml.etree.ElementTree.Element): The current node in the XML tree.

    Returns:
    str: The concatenated text from the node and its descendants.
    """
    text = ""
    for child in node:
        if child.tag == 'w':
            if child.text:
                text += child.text.lower()
        elif child.tag == 'c':
            text += " "
        else:
            if len(node) > 0:
                for grandchild in child:
                    text += retrieve_text(grandchild)
    return text
```

1.2.5 Stages of Implementation

Obviously it was not a good idea to start with trying to process the whole corpus all at once. Thus, I initially started by extracting 1 sentence node from a custom made xml file *documentation/test_1.xml*. Then I moved onto extracting the sentences from 1 file from the corpus *test_2.xml*. Finally, when both of these tests were passed I moved onto processing the whole corpus

1.3 Testing

Before extending the initial *single file* implementation, the json files were checked to verify that the generated counts were correct.

Finally once the implemenation was finished, the performance of this section was tested using 2 different metrics. 1. Time taken to create each n_gram 2. CPU and RAM usage

The extremely long time taken to generate the counts of the whole corpus is the reason behind the decision to generate these counts once and then store them for future use. Another reason for this is that the counts will not change, once the data set has been split the counts for each model will be consistent. The probabilities will also be consistent, however, for performance reasons we want the probabilities to be keyed using tuples since accessing the tokens is much easier this way.

The results for time and computer resources usage are below.

1.3.1 Time Taken

[ ]:
```
import time

def timed_single_file_frequency_counts(path):
    tree = ET.parse(path)
    root = tree.getroot()

    # Create frequency counts for unigrams, bigrams, and trigrams and save to a␣
 ↪json file
    for number_of_words in range(1, 4):
        start_time = time.time()

        n_gram_counts = defaultdict(int)
        traverse_tree(root, number_of_words, n_gram_counts)

        end_time = time.time()
        print(f"Time to process {number_of_words}_gram: {end_time - start_time}␣
 ↪seconds")

def timed_corpus_frequency_counts():
    directories = ['aca', 'dem', 'fic', 'news']
    BASE_PATH = '../data/corpus/Texts/'

    # Create frequency counts for unigrams, bigrams, and trigrams
    for number_of_words in range(1, 4):
        start_time = time.time()
        n_gram_counts = defaultdict(int)

        for directory in directories:
            dir_path = os.path.join(BASE_PATH, directory)
            for file in os.listdir(dir_path):
                if file.endswith('.xml'):
                    file_path = os.path.join(dir_path, file)
                    tree = ET.parse(file_path)
                    root = tree.getroot()
                    for child in root:
                        if child.tag != 'teiHeader':
                            traverse_tree(child, number_of_words, n_gram_counts)

        end_time = time.time()
        print(f"Time to process {number_of_words}_gram: {end_time - start_time}␣
 ↪seconds")

print("Test 1:")
timed_single_file_frequency_counts("test_1.xml")
print()

print("Test 2:")
timed_single_file_frequency_counts("test_2.xml")
print()

print("Test 3:")
timed_corpus_frequency_counts()
print()
```

1.3.2 Computer Resources

```
[ ]: import psutil
     import os

     def single_file_frequency_counts(path):
         tree = ET.parse(path)
         root = tree.getroot()

         # Create frequency counts for unigrams, bigrams, and trigrams
         for number_of_words in range(1, 4):
             n_gram_counts = defaultdict(int)
             for child in root:
                 traverse_tree(child, number_of_words, n_gram_counts)

     def corpus_frequency_counts():
         test_dir_path = "test_3/"
         xml_files = [f for f in os.listdir(test_dir_path) if f.endswith('.xml')]

             # Create frequency counts for unigrams, bigrams, and trigrams
         for number_of_words in range(1, 4):
             n_gram_counts = defaultdict(int)

             for xml_file in xml_files:
                 path = os.path.join(test_dir_path, xml_file)
                 tree = ET.parse(path)
                 root = tree.getroot()
                 for child in root:
```

```python
                traverse_tree(child, number_of_words, n_gram_counts)

def test_cpu_usage_single_file(path):
    process = psutil.Process(os.getpid())
    start_cpu = process.cpu_percent()

    single_file_frequency_counts(path)

    end_cpu = process.cpu_percent()

    cpu_usage = end_cpu - start_cpu

    print(f"CPU Usage: {cpu_usage}%")

def test_ram_usage_single_file(path):
    process = psutil.Process(os.getpid())
    start_ram = process.memory_info().rss / 1024 / 1024

    single_file_frequency_counts(path)

    end_ram = process.memory_info().rss / 1024 / 1024

    ram_usage = end_ram - start_ram

    print(f"RAM Usage: {ram_usage} MB")

def test_cpu_usage_corpus():
    process = psutil.Process(os.getpid())
    start_cpu = process.cpu_percent()

    corpus_frequency_counts()

    end_cpu = process.cpu_percent()

    cpu_usage = end_cpu - start_cpu

    print(f"CPU Usage: {cpu_usage}%")

def test_ram_usage_corpus():
    process = psutil.Process(os.getpid())
    start_ram = process.memory_info().rss / 1024 / 1024

    corpus_frequency_counts()

    end_ram = process.memory_info().rss / 1024 / 1024

    ram_usage = end_ram - start_ram
    print(f"RAM Usage: {ram_usage} MB")

print("One lines test:")
test_cpu_usage_single_file("test_1.xml")
test_ram_usage_single_file("test_1.xml")

print("Single file test:")
test_cpu_usage_single_file("test_2.xml")
test_ram_usage_single_file("test_2.xml")

print("Corpus Test:")
test_cpu_usage_corpus()
test_ram_usage_corpus()
```

1.4 Training and Testing Datasets

I am using the british corpus which consists of a total of 332963 sentence tags. For my project, since in my linear interpolation function the lambdas are preset and dont need to be trained, I am opting to simply split my data set into a training and test set. The split I opted for is 80|20.

Ive also opted to split my data set randomly. Since for this project we mostly dont care about continuity and context of a phrase, I chose to randomly select 80% of the sentences from each file.

The splitting of the data set is handled using 2 functions: - splitting_datasets() - split_and_append_elements(s_elements, training_set, test_set):

1.4.1 splitting_datasets()

This function simply loops through all the files of the corpus, extracts all the sentences from it, and calls split_and_append_elements().

1.4.2 split_and_append_elements(s_elements, training_set, test_set)

It takes 3 parameters:

- s_elements - list of sentence elements
- trainin_set - array representing the training set
- testing_set - array representing the testing set

This function shuffles the initial sentences randomly, and then splits them based on the desired percentages, appending the split lists to the respective global lists.

[ ]:
```
import random

def splitting_datasets():
    """
    Splits the corpus into train and test sets and saves them as XML files.

    Returns:
        None
    """
    train_file_path = '../data/training_set.xml'
    test_file_path = '../data/test_set.xml'
    # Splitting the corpus into train, validation, and test sets if not already␣
 ↪created
    train = []
    test = []

    for directory in directories:
        dir_path = os.path.join(BASE_PATH, directory)
        for file in os.listdir(dir_path):
            if file.endswith('.xml'):
                file_path = os.path.join(dir_path, file)
                tree = ET.parse(file_path)
                root = tree.getroot()
                sentences = list(root.findall('.//s'))
                split_and_append_elements(sentences, train, test)

    if os.path.exists(train_file_path):
        os.remove(train_file_path)
    if os.path.exists(test_file_path):
        os.remove(test_file_path)
    write_xml_from_elements(train, train_file_path)
    write_xml_from_elements(test, test_file_path)

def split_and_append_elements(s_elements, training_set, test_set):
    """
    Splits the given list of elements into train, test, and test sets,
    and appends them to the respective global lists.

    Args:
        s_elements (list): The list of elements to be split.
        training_set (list): The list to append the training set elements to.
        test_set (list): The list to append the test set elements to.
        test_set (list): The list to append the test set elements to.

    Returns:
        None
    """
    total_elements = len(s_elements)
    train_size = int(total_elements * 0.8)

    random.shuffle(s_elements)
    train_elements = s_elements[:train_size]
    test_elements = s_elements[train_size:]
    training_set.extend(train_elements)
    test_set.extend(test_elements)

def write_xml_from_elements(elements, path):
    root = ET.Element('root')
    root.extend(elements)
    tree = ET.ElementTree(root)
    tree.write(path)
```

1.5 Language Models

The 3 language models of my project are all built upon a language model abstract base class. This ABC implements the backbone of all of the models. The most important functions implemented are:

- __init__(self)
- _get_counts(self)
- _generate_counts(self)
- _linear_interpolation(self, trigram)
- text_generator(self, sentence, choice)
- _get_probable_tokens(self, context, choice)
- uni_sentence_probability(self, words)
- bi_sentence_probability(self, words)
- tri_sentence_probability(self, words)
- sentence_probability(self, words)

The language models are then left to simply define how they calculate the probabilities from the counts by implementing the following abstract methods:

- _default_uni_value(self)
- _get_bigram_probability(self, bigram)
- _get_trigram_probability(self, trigram)
- _generate_unigram_probs(self)
- _generate_bigram_probs(self)
- _generate_trigram_probs(self)

1.5.1 __init__(self)

The init method simply defines 6 default dictionaries, one for each ngram count, and one for each ngram probabilities. Then it proceeds to populate these dictionaries by calling _get_counts(), _generate_unigram_probs(), _generate_bigram_probs() and _generate_trigram_probs().

1.5.2 _get_counts(self)

This is method checks if the ngram counts have already been created for the vanilla/laplace model and loads them if so, otherwise it calls self._generate_counts()

1.5.3 __generate__counts(self)

This method iterates over a range of word counts (1 to 3) and generates n-gram counts based on the sentences in the training_set.xml file. The n-gram counts are then saved to separate JSON files for each word count. It does this using the handle_sentence() method discussed in part 1.

1.5.4 __linear__interpolation(self, trigram)

This method returns the probability of a trigram using linear interpolation. It does this by calling the abstract methods implemented by the models themselves.

1.5.5 text__generator(self, sentence, choice)

This section was inspired by the first link in the references section, however, it is still my own work [1]. This method handles the text generation of the model, it accepts a starting phrase and a choice that determines whether the user wants to use unigram, bigram, trigram or linear interpolation to determine the next word. The method first makes sure that the sentence supplied is converted to a list of words and adds a start tag to the beginning of it. Then it jumps to the last 2 tokens and generates a tuple for the context. Next it gets the most probable tokens by calling self.__get__probable__tokens(). Then it selects the next token based on its normalised probability of being chosen. The context tuple is updated and the loop continues until either no word is generated, and end tag is generated or over 100 words are generated.

1.5.6 __get__probable__tokens(self, context, choice)

This method creates a dictionary of all the possible next tokens and their probabilities of being selected. It does this based on the choice selected by the user. In the case that linear interpolation is selected, if the a trigram exists that starts with the context supplied as an argument, than the probability of the token is calculated using the linear interpolation method.

1.5.7 uni__sentence__probability(self, words), bi__sentence__probability(self, words) and tri__sentence__probability(self, words)

These methods return the probability of a sentence using the following formula:

$$P(S) = P(W1)P(W1|CONTEXT)P(W2|CONTEXT)...P(Wn|CONTEXT)$$

Where context is determined by the size of the ngram being used.

1.5.8 sentence__probability(self, words)

This method returns the probability of a sentence using the following formula:

$$P(S) = P(W1)P(W2)P(W3)...P(Wn-1)P(Wn)$$

Where the probability of a word is found using linear interpolation

[ ]:
```
"""
Implements an abstract base class for language models
```

```python
"""
import string
import random
import xml.etree.ElementTree as ET
from collections import defaultdict
import json
import os
from abc import ABC, abstractmethod
import numpy as np

class LanguageModel(ABC):
    def __init__(self):
        self.uni_count = defaultdict(int)
        self.bi_count = defaultdict(int)
        self.tri_count = defaultdict(int)
        self.uni_probabilities = defaultdict(self._default_uni_value)
        self.bi_probabilities = defaultdict(float)
        self.tri_probabilities = defaultdict(float)

        self._get_counts()
        self._generate_unigram_probs()
        self._generate_bigram_probs()
        self._generate_trigram_probs()

    def __str__(self):
        ret_str =  (f"uni_count has {len(self.uni_count.keys())} tokens\n"
                + f"bi_count has {len(self.bi_count.keys())} tokens\n"
                + f"tri_count has {len(self.tri_count.keys())} tokens\n")
        return ret_str

    @abstractmethod
    def _default_uni_value(self):
        pass

    def _get_counts(self):
        if not (os.path.exists('n_grams/vanilla_laplace/1_gram_counts.json')
                and os.path.exists('n_grams/vanilla_laplace/2_gram_counts.json')
                and os.path.exists('n_grams/vanilla_laplace/3_gram_counts.
↪json')):
            self._generate_counts()

        with open("n_grams/vanilla_laplace/1_gram_counts.json", 'r',␣
↪encoding='utf-8') as fp:
            self.uni_count = json.load(fp)
        with open("n_grams/vanilla_laplace/2_gram_counts.json", 'r',␣
↪encoding='utf-8') as fp:
            self.bi_count = json.load(fp)
        with open("n_grams/vanilla_laplace/3_gram_counts.json", 'r',␣
↪encoding='utf-8') as fp:
            self.tri_count = json.load(fp)

    def _generate_counts(self):
        for number_of_words in range(1, 4):
            n_gram_counts = defaultdict(int)
            tree = ET.parse('../data/training_set.xml')
            root = tree.getroot()
            for child in root:
                handle_sentence(child, number_of_words, n_gram_counts)

            with open(f'n_grams/vanilla_laplace/{number_of_words}_gram_counts.
↪json',
                      'w', encoding='utf-8') as fp:
                json.dump(n_gram_counts, fp, indent=4)

    @abstractmethod
    def _generate_unigram_probs(self):
        pass

    @abstractmethod
    def _generate_bigram_probs(self):
        pass

    @abstractmethod
    def _generate_trigram_probs(self):
        pass

    def _remove_punctuation(self, text):
        punctuation = string.punctuation.replace("'", "")
        text_without_punctuation = text.translate(str.maketrans("", "",␣
↪punctuation))

        return text_without_punctuation

    @abstractmethod
    def _get_bigram_probability(self, bigram):
        pass

    @abstractmethod
    def _get_trigram_probability(self, trigram):
        pass

    def _linear_interpolation(self, trigram):
        uni_prob = 0.1 * self.uni_probabilities[trigram[-1]]
        bi_prob = 0.3 * self._get_bigram_probability(trigram[-2:])
        tri_prob = 0.6 * self._get_trigram_probability(trigram)
        return uni_prob + bi_prob + tri_prob

    def text_generator(self, sentence, choice):
        words = sentence
        if not isinstance(words, list):
            words = self._remove_punctuation(words)
            words = words.lower().split()

        words.insert(0, "<s>")
        if len(words) > 1:
            context = tuple(words[-2:])
            loop_prevention_counter = 0

            while context[-1] not in ["</s>", ""] and loop_prevention_counter <␣
↪100:
                token_probabilities = self._get_probable_tokens(context, choice)

                if not token_probabilities:
                    break

                if token_probabilities["<s>"] != 0:
                    del token_probabilities["<s>"]

                # semi-random selection of next word based on normalised␣
↪probability
                prob_sum = sum(token_probabilities.values())
                random_dec = random.random()
                running_sum = 0
                for token, probability in sorted(token_probabilities.items(),
                                                 key=lambda item: item[1]):
                    running_sum += np.divide(probability, prob_sum)
                    if running_sum > random_dec:
                        word = token
                        break

                words.append(word)
                context = (context[-1], word)
                loop_prevention_counter += 1

        if words[-1] != "</s>":
            words.append("</s>")

        print(" ".join(words))

    def _get_probable_tokens(self, context, choice):
        token_probabilities = defaultdict(float)
        if choice == '1':
            for key, value in self.uni_probabilities.items():
                token_probabilities[key] = value
        elif choice == '2':
            for key, value in self.bi_probabilities.items():
                if key[0] == context[-1]:
                    token_probabilities[key[-1]] = value
        elif choice == '3':
            for key, value in self.tri_probabilities.items():
                if key[:2] == context:
                    token_probabilities[key[-1]] = value
        elif choice == '4':
            for key in self.tri_probabilities:
                if key[0:2] == context:
                    token_probabilities[key[-1]] = self.
↪_linear_interpolation(key)

        return token_probabilities

    def uni_sentence_probability(self, words):
        if not isinstance(words, list):
            words = self._remove_punctuation(words.lower())
            words = words.split()

        sentence_probability = 1
        for unigram in words:
            prob = self.uni_probabilities[unigram]
            if prob == 0:
                prob = self.uni_probabilities["<UNK>"]
            sentence_probability *= prob

        return sentence_probability

    def bi_sentence_probability(self, words):
        if not isinstance(words, list):
            words = self._remove_punctuation(words.lower())
            words = ["<s>"] + words.split() + ["</s>"]

        sentence_probability = 1
        for index in range(len(words) - 2):
            bigram = tuple(words[index : index+2])
            prob = self._get_bigram_probability(bigram)
            sentence_probability *= prob

        return sentence_probability
    def tri_sentence_probability(self, words):
        if not isinstance(words, list):
            words = self._remove_punctuation(words.lower())
            words = ["<s>", "<s>"] + words.split() + ["</s>"]

        sentence_probability = 1
        for index in range(len(words) - 3):
            trigram = tuple(words[index : index+3])
            prob = self._get_trigram_probability(trigram)
            sentence_probability *= prob

        return sentence_probability

    def sentence_probability(self, words):
        if not isinstance(words, list):
            words = self._remove_punctuation(words.lower())
            words = ["<s>", "<s>"] + words.split() + ["</s>"]

        sentence_probability = 1
        for index in range(len(words) - 3):
            trigram = tuple(words[index : index+3])
            prob = self._linear_interpolation(trigram)
            sentence_probability *= prob

        return sentence_probability
```

1.6 Vanilla LM

This class extends the LM ABC and implements the abstract methods in their simplest forms.

1.6.1 __default__uni__value(self)

This method simply returns 0.0, the default value for the default dictionary.

1.6.2 __generate__unigram__probs(self)

This method simply calculates the probability of each token by dividing that token by the total token count.

1.6.3 __generate__bigram__probs(self)

This method calculates the probability of a bigram by dividing the count of that bigram over the count of the 1st word in the bigram

1.6.4 __generate__trigram__probs(self)

This method calculates the probability of a trigram by dividing the count of that trigram over the count of the 1st 2 words in the trigram

1.6.5 __get__bigram__probability(self, bigram) and __get__trigram__probability(self, trigram)

These methods simply access the probability dictionaries and return the value of the key.

```
[ ]: from language_model_abc import LanguageModel

class VanillaLM(LanguageModel):
    def _default_uni_value(self):
        return 0.0

    def _generate_unigram_probs(self):
        total_tokens = float(sum(self.uni_count.values()))
        for key in self.uni_count:
            self.uni_probabilities[key] = self.uni_count[key] / total_tokens

    def _generate_bigram_probs(self):
        for key in self.bi_count:
            words = tuple(key.split())
            self.bi_probabilities[words] = self.bi_count[key] / self.
 ↪uni_count[words[0]]

    def _generate_trigram_probs(self):
        for key in self.tri_count:
            words = tuple(key.split())
            bi_gram_key = words[0] + " " + words[1]
            self.tri_probabilities[words] = self.tri_count[key] / self.
 ↪bi_count[bi_gram_key]

    def _get_bigram_probability(self, bigram):
        return self.bi_probabilities[bigram]

    def _get_trigram_probability(self, trigram):
        return self.tri_probabilities[trigram]
```

1.7 Laplace LM

This class extends the LM ABC but has slightly more complicated implementations of the abstract methods. It also has a small adjustment to the methods that calculate the probability of a sentence. In my project a sentence I have treated a sentence as the contents of a s tag in the xml files. Thus, some sentences are extremely long. This combined with the fact that now we are accounting for unkown words by returning an extremely low probability for them, means that sometimes the probability of a sentence will be too small to be represented in python. When this happens I am simply returning the minimum float that python can represent instead. I opted for this method because if only 1 a sentence has 0 probability it will cause the perplexity of that model to be infinity.

1.7.1 __default__uni__value(self)

This method is used as the default value for the default dictionary o the unigram probabilities dictionary. It simply returns

$$\frac{1}{TotalTokens + V}$$

where V is the amount of unique tokens in the training data set.

1.7.2 __generate__unigram__probs(self), __generate__bigram__probs(self) and __generate__trigram__probs(self)

These methods populate the probability dictionaries using the calculation explained in the lecture notes.

1.7.3 __get__bigram__probability(self, bigram) and __get__trigram__probability(self, trigram)

These methods either return the probability of the ngram, or calculate the probability of an unseen word using the formula explained in the lecture notes.

[ ]:
```
import sys
from language_model_abc import LanguageModel

class LaplaceLM(LanguageModel):
    def _default_uni_value(self):
        return float(1 / sum(self.uni_count.values()) + len(self.uni_count))

    def _generate_unigram_probs(self):
        total_tokens = float(sum(self.uni_count.values()))
        for key in self.uni_count:
            self.uni_probabilities[key] = ((self.uni_count[key] + 1)
                                            / (total_tokens + len(self.
 ↪uni_count)))

    def _generate_bigram_probs(self):
        for key in self.bi_count:
            words = tuple(key.split())
            self.bi_probabilities[words] = ((self.bi_count[key] + 1)
                                             / (self.uni_count[words[0]] +␣
 ↪len(self.uni_count)))

    def _generate_trigram_probs(self):
        for key in self.tri_count:
            words = tuple(key.split())
            bi_gram_key = words[0] + " " + words[1]
            self.tri_probabilities[words] = ((self.tri_count[key] + 1)
                                              / (self.bi_count[bi_gram_key] +␣
 ↪len(self.uni_count)))
```

```
    def _get_bigram_probability(self, bigram):
        return self.bi_probabilities.get(bigram,
                                         1 / (self.uni_count.get(bigram[0], 1)
                                              + len(self.uni_count)))

    def _get_trigram_probability(self, trigram):
        return self.tri_probabilities.get(trigram,
                                          1 / (self.bi_count.get(trigram[:2], 1)
                                               + len(self.uni_count)))

    def uni_sentence_probability(self, words):
        return max(super().uni_sentence_probability(words), sys.float_info.min)

    def bi_sentence_probability(self, words):
        return max(super().bi_sentence_probability(words), sys.float_info.min)

    def tri_sentence_probability(self, words):
        return max(super().tri_sentence_probability(words), sys.float_info.min)

    def sentence_probability(self, words):
        return max(super().sentence_probability(words), sys.float_info.min)
```

1.8 UNK LM

This LM also prevents a sentence probability from being 0 and it implements _default_uni_value(self) in the same way as the Laplace LM. It has its own versions of _get_counts(self) and _generate_counts(self) since the token counts will be different for this model. This model also has another attribute vocabulary which is a set of all the unique words that it has seen.

All of the abstract methods of this method are implemented in the same way as the Laplace LM. This is as the abstract methods are all related to probability calculations.

I also implemented a new version of the handle_sentence function that accepts a dictionary of words that are unknown as a parameter. Then when generating the counts any word that is in the unkown tokens dictionary gets set to *UNK*.

1.8.1 __generate_counts(self)

The method is slightly different to the initial implementation of the LM ABC. First it generates the uni gram counts of the traning set, then it adds any token with 2 or less occurences to an unknown tokens dictionary. Finally it generates the ngram counts as normal but replacing all of the unkown tokens with *UNK*.

1.8.2 text_generator(self, sentence, choice), uni_sentence_probability(self, words), bi_sentence_probability(self, words), tri_sentence_probability(self, words) and sentence_probability(self, words)

These methods are the same as the ABC's implementation, however, the sentences need to preprocessed to change any words that are not in the models vocabulary to UNK.

[ ]:
```python
import xml.etree.ElementTree as ET
from collections import defaultdict
import json
import os
import sys
from vanilla import VanillaLM
from dataset_functions import handle_sentence, handle_sentence_unk

def handle_sentence_unk(sentence_node, number_of_words, counts, unknown_tokens):
    text = retrieve_text(sentence_node)
    if text.strip() != "":
        text = ("<s> " * number_of_words) + text + (" </s>")
        words = text.split()
        for index in range(len(words) - number_of_words + 1):
            if words[index] in unknown_tokens:
                words[index] = "<UNK>"
            if number_of_words == 1:
                n_gram = words[index]
            else:
                n_gram = " ".join(words[index:index + number_of_words])
            counts[n_gram] += 1

class UnkLM(VanillaLM):
    def __init__(self):
        super().__init__()
        self.vocabulary = set(self.uni_count)

    def _defualt_uni_value(self):
        return float(1 / sum(self.uni_count.values()) + len(self.uni_count))

    def _get_counts(self):
        if not (os.path.exists('n_grams/unk/1_gram_counts.json')
                and os.path.exists('n_grams/unk/2_gram_counts.json')
                and os.path.exists('n_grams/unk/3_gram_counts.json')):
            self._generate_counts()

        with open("n_grams/unk/1_gram_counts.json", 'r', encoding='utf-8') as␣
 ↪fp:
            self.uni_count = json.load(fp)
        with open("n_grams/unk/2_gram_counts.json", 'r', encoding='utf-8') as␣
 ↪fp:
```

```
            self.bi_count = json.load(fp)
        with open("n_grams/unk/3_gram_counts.json", 'r', encoding='utf-8') as␣
↪fp:
            self.tri_count = json.load(fp)

    def _generate_counts(self):
        n_gram_counts = defaultdict(int)
        tree = ET.parse('../data/training_set.xml')
        root = tree.getroot()
        for child in root:
            handle_sentence(child, 1, n_gram_counts)

        unknown_tokens = {key for key, count in n_gram_counts.items() if count␣
↪<= 2}

        # Generate real counts:
        for number_of_words in range(1, 4):
            n_gram_counts = defaultdict(int)
            for child in root:
                handle_sentence_unk(child, number_of_words, n_gram_counts,␣
↪unknown_tokens)

            with open(f'n_grams/unk/{number_of_words}_gram_counts.json',
                    'w', encoding='utf-8') as fp:
                json.dump(n_gram_counts, fp, indent=4)

        print(self.uni_count["<UNK>"])

    def _generate_unigram_probs(self):
        total_tokens = float(sum(self.uni_count.values()))
        for key in self.uni_count:
            self.uni_probabilities[key] = ((self.uni_count[key] + 1)
                                            / (total_tokens + len(self.
↪uni_count)))

    def _generate_bigram_probs(self):
        for key in self.bi_count:
            words = tuple(key.split())
            self.bi_probabilities[words] = ((self.bi_count[key] + 1)
                                             / (self.uni_count[words[0]] +␣
↪len(self.uni_count)))

    def _generate_trigram_probs(self):
        for key in self.tri_count:
            words = tuple(key.split())
            bi_gram_key = words[0] + " " + words[1]
            self.tri_probabilities[words] = ((self.tri_count[key] + 1)
                                            / (self.bi_count[bi_gram_key] +␣
↪len(self.uni_count)))

    def _get_bigram_probability(self, bigram):
        return self.bi_probabilities.get(bigram,
                                         1 / (self.uni_count.get(bigram[0], 1)
                                              + len(self.uni_count)))

    def _get_trigram_probability(self, trigram):
        return self.tri_probabilities.get(trigram,
                                          1 / (self.bi_count.get(trigram[:2], 1)
                                               + len(self.uni_count)))

    def text_generator(self, sentence, choice):
        sentence = self._remove_punctuation(sentence)
        sentence = sentence.lower().split()
        for index, word in enumerate(sentence):
            if word not in self.vocabulary:
                sentence[index] = "<UNK>"
        return super().text_generator(sentence, choice)

    def uni_sentence_probability(self, words):
        words = self._remove_punctuation(words.lower())
        words = words.split()
        for index, word in enumerate(words):
            if word not in self.vocabulary:
                words[index] = "<UNK>"
        return max(super().uni_sentence_probability(words), sys.float_info.min)

    def bi_sentence_probability(self, words):
        words = self._remove_punctuation(words.lower())
        words = ["<s>"] + words.split() + ["</s>"]
        for index, word in enumerate(words):
            if word not in self.vocabulary:
                words[index] = "<UNK>"
        return max(super().bi_sentence_probability(words), sys.float_info.min)

    def tri_sentence_probability(self, words):
        words = self._remove_punctuation(words.lower())
        words = ["<s>", "<s>"] + words.split() + ["</s>"]
        for index, word in enumerate(words):
            if word not in self.vocabulary:
                words[index] = "<UNK>"
        return max(super().tri_sentence_probability(words), sys.float_info.min)

    def sentence_probability(self, words):
        words = self._remove_punctuation(words.lower())
        words = ["<s>", "<s>"] + words.split() + ["</s>"]
        for index, word in enumerate(words):
            if word not in self.vocabulary:
                words[index] = "<UNK>"
        return max(super().sentence_probability(words), sys.float_info.min)
```

1.9 LM Training Time and Space Requirements

To test how much time was needed to train the models 2 tests were done, the first was when the word counts were not already created and then with them created. The results of training the models and their space requirements are below:

1.9.1 Time

No counts ready: Vanilla: 27.697623014450073 seconds Laplace: 27.66264796257019 seconds UNK: 18.40277075767517 seconds

Counts ready: Vanilla: 3.051021099090576 Seconds Laplace: 3.132711887359619 Seconds Unk: 2.9299330711364746 Seconds

1.9.2 Space

To calculate the space each model used I created the method below for the LM ABC.

Vanilla: 744142138 bytes Laplace: 744142138 bytes UNK: 729590527 bytes

1.9.3 Findings

From my findings it is evident that the trigram model for vanilla and unknown models is very expensive to calculate. Through further investigation I found that this is due to all the single and double trigram occurences that occur that the UNK model doesnt have to deal with. Also I was initially quite surprised with the space demands, but it makes sense when considering the size of the training data and the fact that I am storing it 6 times.

[ ]:
```
import sys

def calculate_space_needed(self):
    size = sys.getsizeof(self.uni_count)
    for key, value in self.uni_count.items():
        size += sys.getsizeof(key)
        size += sys.getsizeof(value)

    size += sys.getsizeof(self.bi_count)
    for key, value in self.bi_count.items():
        size += sys.getsizeof(key)
        size += sys.getsizeof(value)

    size += sys.getsizeof(self.tri_count)
    for key, value in self.tri_count.items():
        size += sys.getsizeof(key)
```

```
            size += sys.getsizeof(value)

        size += sys.getsizeof(self.uni_probabilities)
        for key, value in self.uni_probabilities.items():
            size += sys.getsizeof(key)
            size += sys.getsizeof(value)

        size += sys.getsizeof(self.bi_probabilities)
        for key, value in self.bi_probabilities.items():
            size += sys.getsizeof(key)
            size += sys.getsizeof(value)

        size += sys.getsizeof(self.tri_probabilities)
        for key, value in self.tri_probabilities.items():
            size += sys.getsizeof(key)
            size += sys.getsizeof(value)

        print(size)

    def calculate_space_needed_unk(self):
        size = sys.getsizeof(self.uni_count)
        for key, value in self.uni_count.items():
            size += sys.getsizeof(key)
            size += sys.getsizeof(value)

        size += sys.getsizeof(self.bi_count)
        for key, value in self.bi_count.items():
            size += sys.getsizeof(key)
            size += sys.getsizeof(value)

        size += sys.getsizeof(self.tri_count)
        for key, value in self.tri_count.items():
            size += sys.getsizeof(key)
            size += sys.getsizeof(value)

        size += sys.getsizeof(self.uni_probabilities)
        for key, value in self.uni_probabilities.items():
            size += sys.getsizeof(key)
            size += sys.getsizeof(value)

        size += sys.getsizeof(self.bi_probabilities)
        for key, value in self.bi_probabilities.items():
            size += sys.getsizeof(key)
            size += sys.getsizeof(value)

        size += sys.getsizeof(self.tri_probabilities)
        for key, value in self.tri_probabilities.items():
```

```
        size += sys.getsizeof(key)
        size += sys.getsizeof(value)

    size += sys.getsizeof(self.vocabulary)
    for key in self.vocabulary:
        size += sys.getsizeof(key)

    print(size)
```

1.10 Perplexities

I calculated perplexity of the data set by averaging the perplexities of each sentence. This was calculated using the following formula:

$$PP(S) = \frac{1}{P(S)}^{\frac{1}{N}}$$

where N is the number of words in the sentence

	Unigram	**Bigram**	**Trigram**	**Linear Interpolation**
Vanilla	INFINITY	INFINITY	INFINITY	INFINITY
Laplace	8354.61	9615.34	22976.01	5268.14
UNK	5603.24	5965.04	10951.60	3515.70

The infinity values obtained are due to the vanilla lm encountering unknown words. When this happens the the perplexity approaches infinity due the division by zero. The other models perplexity values are a bit skewed due to the fact that for some sentences I am not returning their true probability, as this is smaller than python can represent using a float.

1.11 Sentence Probabilities

For this section the first test sentence I used is this:

“ocean environmental management has applied for a 30,000-tonne plant at seal sands teesside and is appealing against the local authority’s refusal to allow it”

bosnian refugees assured of asylum

Here is a table with the sentence probabilities:

	Unigram	**Bigram**	**Trigram**	**Linear Interpolation**
Vanilla	0.0	0.0	0.0	0.0
Laplace	4.9653571e-77	6.8572746e-99	8.8640346e-116	5.2145152e-86
UNK	1.31012857e-83	6.44653e-91	3.321860e-107	4.20396477e-88

I also tried it with this sentence to try and gain some values from the Vanilla lm.

“bosnian refugees assured of asylum”

	Unigram	Bigram	Trigram	Linear Interpolation
Vanilla	4.3879832e-23	0.0	0.0	6.770046e-26
Laplace	4.68944600e-23	5.1685384e-25	3.3177554e-25	4.0163971e-23
UNK	4.99982e-23	1.09758518e-23	1.16178394e-23	4.9647802e-22

The results are consistent, The more specific the model gets in terms of context the lower the probability of a sentence occuring. This is due simply to the massive increase in the amount of combinations of words that can occur.

1.12 Text generation

The phrase I selected initially to prompt the models with is the first 3 words of a sentence in my test set:

“what is the”

1.12.1 Vanilla

- Uni: what is the this monogram manager hence that be the telephoned can’t presentable jones more bridgeborough skin had up eleven from to joint-owner to he known breaking
- Bi: what is the agency ownership can be left
- Tri: what is the subject
- Linear Interpolation: what is the most private of all those years ago

1.12.2 Laplace

- Uni: what is the doreen created forms obesity familiar spaces east they times else essential worry
- Bi: what is the user is sold out four bedrooms
- Tri: what is the boss of coach giant national express closed at a big house
- Linear Interpolation: what is the value the right

1.12.3 UNK

- Uni: what is the these mine current an management what the give managed
- Bi: what is the equation 18 sartre how you can give consumers have been denied the middle aged golden girl at 7.45pm
- Tri: what is the probable outcome
- Linear Interpolation: what is the

1.12.4 Observations

Here are a few observations of my results:

1. Unigrams as expected make no sense.
2. Unigrams are also the longest sentences as expected.

3. As the ngram size increases, the sequences of words produced seem to make more sense, it gets noticeably better at each step.
4. Obviously since the predicted word is random, the above obvservations are not true 100% of the time.

1.13 Final Project

When running the main file of my project, after the models have trained the user is prompted whether they would like to generate text, or calculate the probability of a sentence. Next they are prompted to choose their model. In the case that they choose text generation, they are also prompted to decide whether they would like to generate based on unigram, bigram, trigram or linear interpolation.

I opted for this approach since I found it more interesting to be able to compare each level of each model.

1.14 Testing

Throughout the development of my code I tested each section rigorously. Any large issue that I encountered such as sentence probabilities being too small has already been documented and explianed.

1.15 References

1. Lecture notes - https://www.um.edu.mt/vle/pluginfile.php/1356879/mod_resource/content/1/Lecture%20S
%20Language%20Modelling.pdf
2. Inspiration for text generator section - https://github.com/aduroy/n-gram-generator/tree/master
3. sentence probability research - https://cs.stackexchange.com/questions/47502/computing-probability-of-sentence-using-n-grams#:~:text=The%20N%2Dgram%20model%20assumes,N%2B1%7Cw2%
4. perplexity research - https://towardsdatascience.com/perplexity-of-language-models-revisited-6b9b4cf46792

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Saul Vassallo	[illegible]
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

ICS2203	Language Models Assignment
Course Code	Title of work submitted

5/5/2024

Date