James P O Gaskell 01/19/2023

CSC-483 Report: Project1 ‘Ngrams’

James Gaskell
ID: 2677886

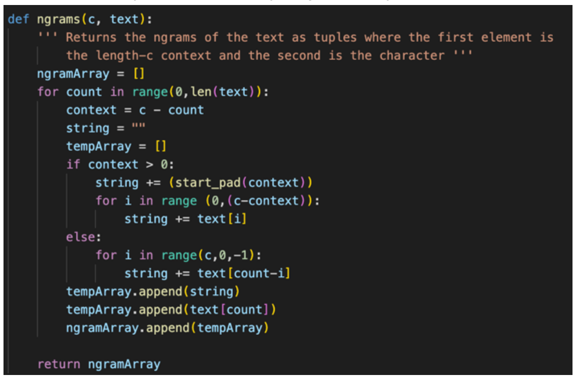
Abstract:

The impetus of this project is to produce a natural language model that can use given test corpuses to both write novel scripts and to identify the language of other given texts. The specific corpuses I will use to test this model and prove its funcitonality are the works of Shakespeare – which I will use to produce an output of N chcracters in the same style as the given text – and lists of cities which will use a predictive function to supply me the most likely country and the accuracy of the preditions.

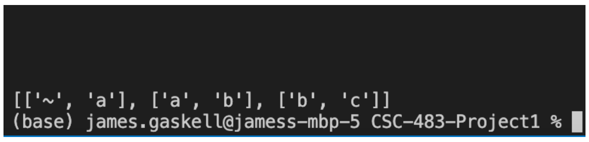
Evidence:

Ngrams()

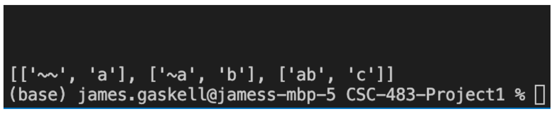
The first aspect of the project is pictured below.



```python
def ngrams(c, text):
    ''' Returns the ngrams of the text as tuples where the first element is
        the length-c context and the second is the character '''
    ngramArray = []
    for count in range(0,len(text)):
        context = c - count
        string = ""
        tempArray = []
        if context > 0:
            string += (start_pad(context))
            for i in range (0,(c-context)):
                string += text[i]
        else:
            for i in range(c,0,-1):
                string += text[count-i]
        tempArray.append(string)
        tempArray.append(text[count])
        ngramArray.append(tempArray)

    return ngramArray
```



```
[['~', 'a'], ['a', 'b'], ['b', 'c']]
(base) james.gaskell@jamess-mbp-5 CSC-483-Project1 %
```

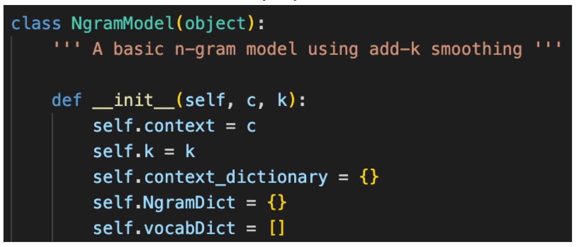


```
[['~~', 'a'], ['~a', 'b'], ['ab', 'c']]
(base) james.gaskell@jamess-mbp-5 CSC-483-Project1 %
```

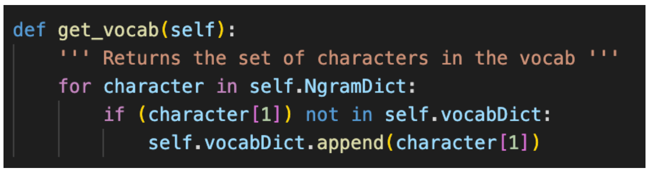
The ngrams() function takes perameters c and text – c being the number of characters for the context of the model and the text being the corpus for which I must find the ngrams of order c. The function returns the ngrams as an array; since I will iterate through this list later on to collect like terms I chose this data structure as it is parsed easily. The output of a simple test is also shown for ngrams(1,’abc’) and ngrams (2,’abc’)

Basic Model

The initialiser called each time a new object NgramModel is intantiated is shown below. I decided to use dictionaries for my Ngrams and contexts. My decision to save the contexts and their counts seperately was decidedly a good decision as whilst my Ngram models take slightly longer to populate, the functions that run on the models are much quicker later on. A vocab dictionary is also initialised and is later populated with the characters/tokens from the corpus



```python
class NgramModel(object):
    ''' A basic n-gram model using add-k smoothing '''

    def __init__(self, c, k):
        self.context = c
        self.k = k
        self.context_dictionary = {}
        self.NgramDict = {}
        self.vocabDict = []
```

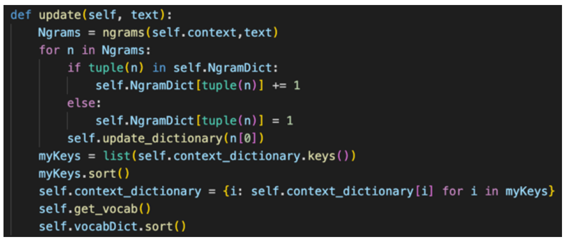


```python
def get_vocab(self):
    ''' Returns the set of characters in the vocab '''
    for character in self.NgramDict:
        if (character[1]) not in self.vocabDict:
            self.vocabDict.append(character[1])
```

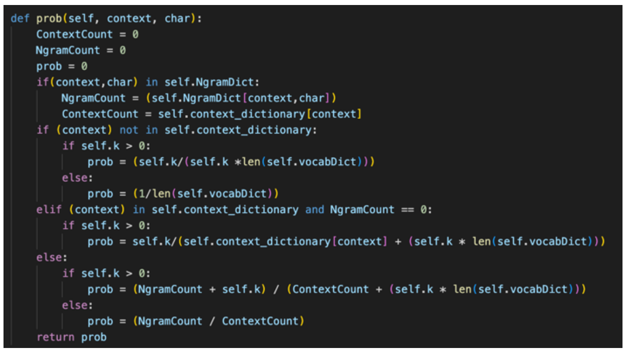
The get_vocab() method is also pictured. This method returns the set of characters to the vocab array.

Update() and Prob()

Update() and Prob() are pictured below



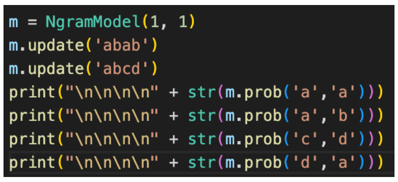
```
def update(self, text):
    Ngrams = ngrams(self.context,text)
    for n in Ngrams:
        if tuple(n) in self.NgramDict:
            self.NgramDict[tuple(n)] += 1
        else:
            self.NgramDict[tuple(n)] = 1
        self.update_dictionary(n[0])
    myKeys = list(self.context_dictionary.keys())
    myKeys.sort()
    self.context_dictionary = {i: self.context_dictionary[i] for i in myKeys}
    self.get_vocab()
    self.vocabDict.sort()
```



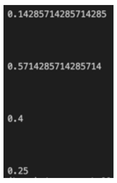
```
def prob(self, context, char):
    ContextCount = 0
    NgramCount = 0
    prob = 0
    if(context,char) in self.NgramDict:
        NgramCount = (self.NgramDict[context,char])
        ContextCount = self.context_dictionary[context]
    if (context) not in self.context_dictionary:
        if self.k > 0:
            prob = (self.k/(self.k *len(self.vocabDict)))
        else:
            prob = (1/len(self.vocabDict))
    elif (context) in self.context_dictionary and NgramCount == 0:
        if self.k > 0:
            prob = self.k/(self.context_dictionary[context] + (self.k * len(self.vocabDict)))
    else:
        if self.k > 0:
            prob = (NgramCount + self.k) / (ContextCount + (self.k * len(self.vocabDict)))
        else:
            prob = (NgramCount / ContextCount)
    return prob
```

Update() is used to update the model with new texts to convert into Ngrams and store in the dictionary. The method systematicllay checks id the new ngram is in the dictionary; if so it adds 1 to its count otherwise it adds it as a novel key with count 1. The Prob method above is adjusted for add-k smoothing. The method takes context and char as perameters and is used to find the probability of the character occuring after a given context. In some cases the full ngram or the context may not be saved within the dictionary in which case, initially 1/len(vocab) is used to prevent 0 probabilities.

Later in the project I added 'add-k' smoothing to add a fraction of 1 to each of the probabilities. This makes the n-gram probabilities not probabilities as such but rather representative figures to show how likely the character is given a context. The formula I used was k/k*len(vocab) and this figure was added to all n-grams preventing 0 probabilities and not disadvantaging other n-grams as much as would be the case if this was only added to novel contexts. Some tests are shown below.



```
m = NgramModel(1, 1)
m.update('abab')
m.update('abcd')
print("\n\n\n\n" + str(m.prob('a','a')))
print("\n\n\n\n" + str(m.prob('a','b')))
print("\n\n\n\n" + str(m.prob('c','d')))
print("\n\n\n\n" + str(m.prob('d','a')))
```

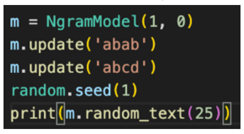


```
0.14285714285714285


0.5714285714285714


0.4


0.25
```

The model in this case allocates probabilities to (a,a) and (d,a) which would have otherwise returned 0 probabilities.

Use of Randomness - Shakespeare

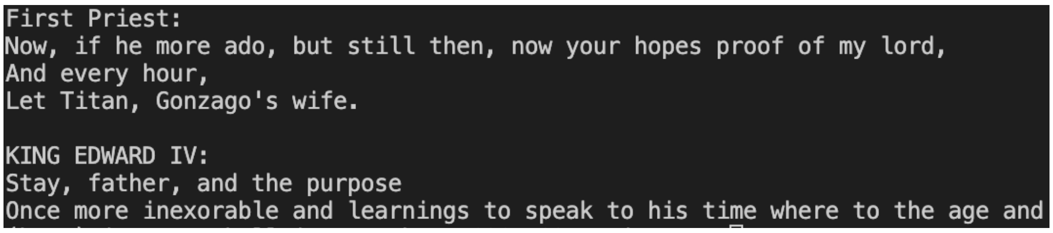
The model uses an element of randomness to generate characters from the model. Random.random() is used to generate an r value between 1 and 0. Methods random_text() and random_char() are evidenced below. The random chacracter function cycles through the dictionary and cumulates the probabilities of chcracters unctil a character is found that, when added to the total cumulative probability, takes the number above r. This chcracter is taken by random_text and added to the string to be returned. An interesting part of implementing this method was updating the context after each character is added since it must be used to find the next character. I achieved this by using string[-self.context:] to take only the last 'c' characters in each run through the random character loop. A targeted test is shown below.



```
m = NgramModel(1, 0)
m.update('abab')
m.update('abcd')
random.seed(1)
print(m.random_text(25))
```

The test produced 'abcdbabcdabababcdddabcdba' as output in line with the expected behaviour.

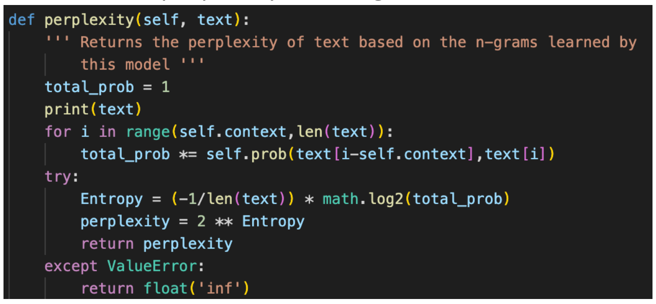
Upon training the data with the Shakespeare corpus it was evident that each of the outputted scripts all had something in common; starting with the character 'F'. Using higher n-gram models this common starting string evolved to the word first. This can be explained by the test corpus starting with the word First – the n-grams within the model containing the start pad (using a tri-gram as an example) '~~F' , '~Fi and 'fir' cause the random character to return the characters in this specific order due to their high probabilities when using start padded context compared to the rest of the dictionary. An example of a six-gram output based on the shakespeare test file is shown below.



```
First Priest:
Now, if he more ado, but still then, now your hopes proof of my lord,
And every hour,
Let Titan, Gonzago's wife.

KING EDWARD IV:
Stay, father, and the purpose
Once more inexorable and learnings to speak to his time where to the age and
```

Perplexity, Smoothing and Interpolation

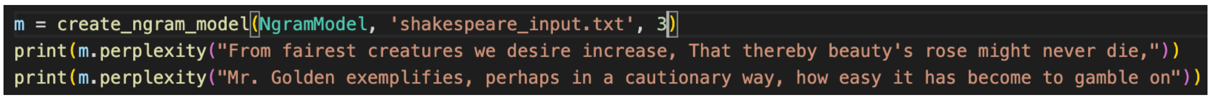
Perplexity is a measure of ow surprised a model is to be presented with a string. My model to calculate the perplexity of strings is shown below.



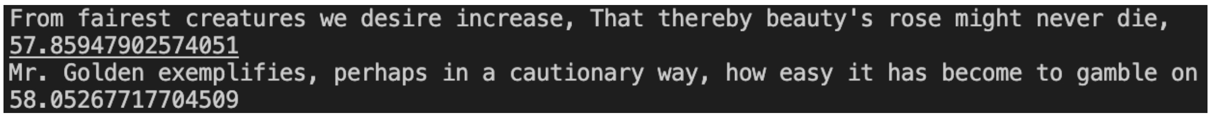
```python
def perplexity(self, text):
    ''' Returns the perplexity of text based on the n-grams learned by
        this model '''
    total_prob = 1
    print(text)
    for i in range(self.context,len(text)):
        total_prob *= self.prob(text[i-self.context],text[i])
    try:
        Entropy = (-1/len(text)) * math.log2(total_prob)
        perplexity = 2 ** Entropy
        return perplexity
    except ValueError:
        return float('inf')
```

First the entropy is calculated the formula for entropy is: -1/N * $\log_2$(w1,w2…….wn)
The perplexity is then calculated as 2^{Entropy}

The perplexity algorithm works well to show that an inputted text is not the same type as the training corpus and so I tested my perplexity function on both a Shakespeare string and a New York Times string using a model trained on Shakespeare. My test and results are shown below.



```python
m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 3)
print(m.perplexity("From fairest creatures we desire increase, That thereby beauty's rose might never die,"))
print(m.perplexity("Mr. Golden exemplifies, perhaps in a cautionary way, how easy it has become to gamble on"))
```



```
From fairest creatures we desire increase, That thereby beauty's rose might never die,
57.85947902574051
Mr. Golden exemplifies, perhaps in a cautionary way, how easy it has become to gamble on
58.05267717704509
```

The perplexity is higher for the Inputted New York times article – most likely because of the difference in writing style of the two validation corpuses.

Apologies for not getting further – I ran out of time