VINCENT MICHAEL SUTANTO - 2111434
vincent.michael_sutanto.vg6@is.naist.jp

Q. Before an era of deep learning and pre-trained language model, traditional language models were applied in many fields of NLP. In this first warm-up assignment, we will learn how traditional language models work. Your missions are to create

- Bi-gram model

- Tri-gram model

- n-gram model (n>=3)

- Use the smoothing method.

- Kneser-Ney smoothing

- Evaluated your models with Entropy

- Write a report or summary about this assignment

- Write a short explaining about your program and show us your code

- Demonstrate your program by showing scores on correct and incorrect sentences.

Training file: https://github.com/neubig/nlptutorial/blob/master/data/wiki-en-train.word

Calculate Entropy on: https://github.com/neubig/nlptutorial/blob/master/data/wiki-en-test.word

N-Gram Model Overview

For this traditional language task, I created an n-gram language model with 3 smoothing techniques, namely laplace smoothing, add-k smoothing, and kneser-ney smoothing. Preprocessing is not done because the dataset has been preprocessed properly. The N-gram language model is created as a python class. The functions that I implemented in the class are as follows: (snippets of images can be seen at the end of the report, and the complete code can be seen in the notebook)

1. 1. The tokenize function is used to tokenize the sentence into a token.
2. 2. The initiateMatrix function is used to initialize a zero matrix which will later be used to store the probabilities of n-grams
3. 3. The endMarker function is used to add a marker tag to the end of the sentence "</S>"
4. 4. The nGramCounter function is used to count the number of occurrences of an n-gram from the dataset
5. 5. The entropy function is used to calculate the entropy of the dataset. This function works by calculating the entropy of each sentence and finally returns the average entropy of all sentences
6. 6. The fit function is the heart of the nGramModel class. This function calls the previously described functions. First, the frequency of occurrence of each n-gram is recorded in the matrix. Then, the switch case will determine what smoothing process will be used in smoothing probability.

Evaluation on Test Data

I did the entropy calculation for n = 1,2,3 and used all smoothing techniques for each n. The results are as follows:

N	Smoothing	Entropy
1	No	4.825769440410396
	Laplace	4.825769440410396
	Add-5	4.825769440410396
2	No	2.6230528091478513
	Laplace	0.7593649354450983
	Add-5	0.3284907998484869
	Kneser-ney	2.429837788943742
3	No	0.8718609481693802
	Laplace	0.03855510563005131
	Add-5	0.020822585357205863
	Kneser-ney	0.8718609481693802

From the result, it can be seen that the Laplace and Add-k decreased the entropy. Theoretically, the Kneser-ney should produce a better entropy score than the Laplace and Add-k. However, in my experiment, the result said otherwise. The Kneser-ney algorithm is also very memory-intensive because it has to store probability data from the previous n values.

Also, the bigger the N is, the lower the entropy. This means that by using a large value of n, the resulting model will have more understandable meaning. However, the drawback of using a large n is that it takes up more memory, and in my case, n = 4 already crashes the runtime.

Functions

```
def tokenize(self, sentence):
  return sentence.split()

def initiateMatrix(self, xAxis, yAxis):
  self._countMatrix.append([np.zeros((len(xAxis), len(yAxis)), dtype=float)])

def endMarker(self, sentence):
  return sentence + str(" </S>")

def nGramCounter(self, pandasSeriesOfSentence, n):
  words = (pandasSeriesOfSentence.str.split(' ').explode())
  tempGram = (words)
  for i in range(n-1):
    nextWord = words.groupby(level=0).shift(i*-1-1)
    tempGram = (tempGram + " " + nextWord)

  tempGram.dropna()
  return tempGram.value_counts()[:]
```

```
def entropy(self, testData):
  listEntropy = []
  xContext = None
  yGram = None

  self._testData = testData.copy().apply(self.endMarker)
  for gram in range(self._n):
    if gram != 0: self._testData = '<S> ' + self._testData

  for sentence in self._testData:
    key = []
    sumProb = 0
    tokenized = sentence.split()

    if gram > 0:
      for i in range(len(tokenized)):
        key.append(tokenized[i])
        if len(key) == gram + 1:
          xgram = " ".join(key[:gram])
          ygram = " ".join(key[:])

          if xgram in self._nmin1gramCounterIndexer[gram][0] and ygram in self._ngramCounterIndexer[gram][0]:
            proba = self._countMatrix[gram][0][self._nmin1gramCounterIndexer[gram][0].index(xgram)][self._ngramCounterIndexer[gram][0].index(ygram)]
            sumProb = sumProb + proba * math.log2(proba)
          else:
            proba = 0.75/len(self._nmin1gramCounterIndexer[gram][0])
            sumProb = sumProb + proba * math.log2(proba)
          key.pop(0)
    else:
      for i in range(len(tokenized)):
        if tokenized[i] in self._countMatrix[0][0]:
          proba = self._countMatrix[0][0][tokenized[i]]
          sumProb = sumProb + proba * math.log2(proba)
        else:
          proba = 0.75/len(self._countMatrix[0][0])
          sumProb = sumProb + proba * math.log2(proba)

  listEntropy.append(-sumProb)

return statistics.mean(listEntropy)
```

```
def fit(self, x, n, smoothingType = 'non', addk = 0):
  self._n = n
  self._data = x.copy().swifter.apply(self.endMarker)
  self._smoothingType  = smoothingType
  self._addk = addk

  for gram in range(n):
    if gram == 0:
      self._ngramCounter.append([self.nGramCounter(self._data, gram+1)])
      self._nmin1gramCounter.append([])
      self._nmin1gramCounterIndexer.append([])
      self._ngramCounterIndexer.append([])
      self._countMatrix.append([self._ngramCounter[gram][0] / len(self._ngramCounter[gram][0])])
    else:
      self._data = '<S> ' + self._data
      self._nmin1gramCounter.append([self.nGramCounter(self._data, gram)])
      self._ngramCounter.append([self.nGramCounter(self._data, gram+1)])
      self._nmin1gramCounterIndexer.append([self._nmin1gramCounter[gram][0].index.to_list()])
      self._ngramCounterIndexer.append([self._ngramCounter[gram][0].index.to_list()])
      self.initiateMatrix(self._nmin1gramCounter[gram][0], self._ngramCounter[gram][0])

      for sentence in self._data:
        key = []

        tokenized = sentence.split()
        for i in range(len(tokenized)):
          key.append(tokenized[i])
          if len(key) == gram + 1:
            xgram = " ".join(key[:gram])
            ygram = " ".join(key[:])
            self._countMatrix[gram][0][self._nmin1gramCounterIndexer[gram][0].index(xgram)][self._ngramCounterIndexer[gram][0].index(ygram)] += 1
            key.pop(0)
```

```
    if self._smoothingType == 'non':
      for i in range(len(self._countMatrix[gram][0])):
        self._countMatrix[gram][0][i] = self._countMatrix[gram][0][i] / self._nmin1gramCounter[gram][0][i]
    elif self._smoothingType == 'laplace':
      for i in range(len(self._countMatrix[gram][0])):
        self._countMatrix[gram][0][i] = (self._countMatrix[gram][0][i] + 1) / (self._nmin1gramCounter[gram][0][i] + len(self._nmin1gramCounter[gram][0]))
    elif self._smoothingType == 'add-k':
      for i in range(len(self._countMatrix[gram][0])):
        self._countMatrix[gram][0][i] = (self._countMatrix[gram][0][i] + self._addk) / (self._nmin1gramCounter[gram][0][i] + (self._addk * len(self._nmin1gramCounter[gram][0])))
```

```
if self._smoothingType == 'kneser-ney':
  for i in tqdm(range(len(self._countMatrix[self._n-1][0]))): #->74
    for j in range(len(self._countMatrix[self._n-1][0][i])): #->79
      context = self._ngramCounterIndexer[self._n-1][0][j]
      string = self._nmin1gramCounterIndexer[self._n-1][0][i]
      if context.startswith(string):
        pContinuationNominator = 0
        pContinuationDenominator = 0
        d = 1
        backStep = self._n
        tokenContext = context.replace(string, '')
        #firstTerm = max(0, ) /

        while pContinuationNominator == 0:
          if backStep != 0:
            backStep = backStep - 1
            if backStep != 0:
              pContinuationNominator = [item.endswith(tokenContext) for item in self._ngramCounterIndexer[backStep][0]].count(True)
              pContinuationDenominator = len(self._ngramCounterIndexer[backStep][0])
          else:
            pContinuationNominator = 1
            pContinuationDenominator = len(self._ngramCounter[0][0])

        if backStep != self._n-1: d = 0.75

        firstTerm = max(0, (self._countMatrix[self._n-1][0][i][j] - d))/ self._nmin1gramCounter[self._n-1][0][i]
        B = [item.startswith(string) for item in self._ngramCounterIndexer[self._n-1][0]]
        A = self._ngramCounter[self._n-1][0]
        lambdaLaterExpression =  sum(a for a,b in zip(A, B) if b)
        lamd = (d / self._nmin1gramCounter[self._n-1][0][i]) * lambdaLaterExpression
        pContinuation = pContinuationNominator / pContinuationDenominator
        self._countMatrix[self._n-1][0][i][j] = firstTerm + lamd * pContinuation
```