Lab2_1_language_modeling_to_student_2024

January 15, 2025

1 Language Modeling using Ngram

In this Exercise, we are going to create a bigram language model and its variation. We will build one model for each of the following type and calculate their perplexity: - Unigram Model - Bigram Model - Bigram Model with Laplace smoothing - Bigram Model with Interpolation - Bigram Model with Kneser-ney Interpolation

We will also use NLTK which is a natural language processing library for python to make our lives easier.

[1]:
```
# # #download corpus
# # !wget --no-check-certificate https://github.com/ekapolc/nlp_2019/raw/master/
 ↪HW4/BEST2010.zip
# # !unzip BEST2010.zip
# !curl -L -o BEST2010.zip https://github.com/ekapolc/nlp_2019/raw/master/HW4/
 ↪BEST2010.zip
# !unzip -o BEST2010.zip
```

[2]:
```
# !wget https://www.dropbox.com/s/jajdlqnp5h0ywvo/tokenized_wiki_sample.csv
# !curl -L -o tokenized_wiki_sample.csv https://www.dropbox.com/s/
 ↪jajdlqnp5h0ywvo/tokenized_wiki_sample.csv
```

[3]:
```
#First we import necessary library such as math, nltk, bigram, and collections.
import math
import nltk
import io
import random
from random import shuffle
from nltk import bigrams, trigrams
from collections import Counter, defaultdict
random.seed(999)
```

BEST2010 is a free Thai NLP dataset by NECTEC usually used as a standard benchmark for various NLP tasks including language modeling. It is separated into 4 domains including article, encyclopedia, news, and novel. The data is already tokenized using '|' as a separator.

For example,

| | | | | | . . | | | | . . | | | | | | | | | | | | |

```
[4]: total_word_count = 0
     best2010 = []
     with open('BEST2010/news.txt','r',encoding='utf-8') as f:
       for i,line in enumerate(f):
         line=line.strip()[:-1] #remove the trailing |
         total_word_count += len(line.split("|"))
         best2010.append(line)
```

```
[5]: #For simplicity, we assumes that each line is a sentence.
     print (f'Total sentences in BEST2010 news dataset :\t{len(best2010)}')
     print (f'Total word counts in BEST2010 news dataset :\t{total_word_count}')
```

```
Total sentences in BEST2010 news dataset :      30969
Total word counts in BEST2010 news dataset :    1660190
```

We separate the input into 2 sets, train and test data with 70:30 ratio

```
[6]: sentences = best2010
     # The data is separated to train and test set with 70:30 ratio.
     train = sentences[:int(len(sentences)*0.7)]
     test = sentences[int(len(sentences)*0.7):]

     #Training data
     train_word_count =0
     for line in train:
         for word in line.split('|'):
             train_word_count+=1
     print ('Total sentences in BEST2010 news training dataset :\t'+ str(len(train)))
     print ('Total word counts in BEST2010 news training dataset :\t'+␣
      ↪str(train_word_count))
```

```
Total sentences in BEST2010 news training dataset :     21678
Total word counts in BEST2010 news training dataset :   1042797
```

Here we load the data from Wikipedia which is also already tokenized. It will be used for answering questions in MyCourseville.

```
[7]: import pandas as pd
     wiki_data = pd.read_csv("tokenized_wiki_sample.csv")
```

1.1 Data Preprocessing

Before training any language models, the first step we always do is process the data into the format suited for the LM.

For this exercise, we will use NLTK to help process our data.

```
[8]: from nltk.lm.preprocessing import pad_both_ends, flatten
     from nltk.lm.vocabulary import Vocabulary
     from nltk import ngrams
```

We begin by "tokenizing" our training set. Note that the data is already tokenized so we can just split it.

[9]:
```
tokenized_train = [["<s>"] + t.split("|") + ["</s>"] for t in train] #␣
  ↪"tokenize" each sentence
```

Next we create a vocabulary with the `Vocabulary` class from NLTK. It accepts a list of tokens so we flatten our sentences into one long sentence first.

[10]:
```
flat_tokens = list(flatten(tokenized_train)) #join all sentences into one long␣
  ↪sentence
vocab = Vocabulary(flat_tokens, unk_cutoff=3) #Words with frequency **below** 3␣
  ↪(not exactly 3) will not be considered in our vocab and will be converted to␣
  ↪<UNK>.
```

Then we replace low frequency words and pad each sentence with <s> in the front and </s> in the back of each sentence.

Now *each* sentence is going to look something like this: ["<s>", "hello", "my", "name", "is", "<UNK>", "</s>"]

[11]:
```
tokenized_train = [[token if token in vocab else "<UNK>" for token in sentence]␣
  ↪for sentence in tokenized_train]
padded_tokenized_train = [list(pad_both_ends(sentence, n=2)) for sentence in␣
  ↪tokenized_train]
```

Finally, we do the same for the test set and the wiki dataset.

[12]:
```
tokenized_test = [t.split("|") for t in test]
tokenized_test = [[token if token in vocab else "<UNK>" for token in sentence]␣
  ↪for sentence in tokenized_test]
padded_tokenized_test = [list(pad_both_ends(sentence, n=2)) for sentence in␣
  ↪tokenized_test]

tokenized_wiki_test = [t.split("|") for t in wiki_data['tokenized'].tolist()]
tokenized_wiki_test = [[token if token in vocab else "<UNK>" for token in␣
  ↪sentence] for sentence in tokenized_wiki_test]
padded_tokenized_wiki_test = [list(pad_both_ends(sentence, n=2)) for sentence␣
  ↪in tokenized_wiki_test]
```

2 Unigram

In this section, we will demonstrate how to build a unigram language model **Important note:** **<s>** = sentence start symbol **</s>** = sentence end symbol

3 VERY IMPORTANT:

- In this notebook, we will *not* default the unknown token probability to `1/len(vocab)` but instead will treat it as a normal word and let the model learn its probability so that we can compare our results to NLTK.

- **Also make sure that the code in this notebook can be executed without any problem. If we find that you used NLTK to answer questions in MyCourseVille and did not finish the assignment, you will receive a grade of 0 for this assignment.**

[13]:
```
class UnigramModel():
  def __init__(self, data, vocab):
    self.unigram_count = defaultdict(lambda: 0.0)
    self.word_count = 0
    self.vocab = vocab
    for sentence in data:
        for w in sentence: #[(word1, ), (word2, ), (word3, )...]
          w = w[0]
          if w in self.vocab:
            self.unigram_count[w] +=1.0
          else:
            self.unigram_count["<UNK>"] += 1.0
          self.word_count+=1

  def __getitem__(self, w):
    w = w[0]  #[(word1, ), (word2, ), (word3, )...]
    if w in self.vocab:
      return self.unigram_count[w]/(self.word_count)
    else:
      return self.unigram_count["<UNK>"]/(self.word_count)
```

[14]:
```
train_unigrams = [list(ngrams(sent, n=1)) for sent in padded_tokenized_train]␣
 ↪#creating the unigrams by setting n=1
model = UnigramModel(train_unigrams, vocab)
```

[15]:
```
def getLnValue(x):
      return math.log(x)
```

[16]:
```
#problability of '   '
print(getLnValue(model['   ']))

#for example, problability of '      ' which is an unknown word is equal to
print(getLnValue(model['      ']))

#problability of '   ' ' ' ' ' '   ' ' ' ' '
prob = getLnValue(model['   '])+getLnValue(model[' '])+␣
 ↪getLnValue(model[' '])+getLnValue(model['   '])+getLnValue(model[' '])+getLnValue(model[' '
 ↪s>'])
print ('Problability of a sentence', math.exp(prob))
```

```
-3.991273499731109
-3.991273499731109
Problability of a sentence 1.408776035744038e-16
```

4 Perplexity

In order to compare language model we need to calculate perplexity. In this task you should write a perplexity calculation code for the unigram model. The result perplexity should be around 406.89 and 376.86 on train and test data.

4.1 TODO #1 Calculate perplexity

[17]:
```
def getLnValue(x):
    return math.log(x) if x>0 else float('-inf')

def calculate_unigram_sentence_ln_prob(sentence, model):
    ln_prob = 0.0
    for word in sentence:
        ln_prob += getLnValue(model[word])
    return ln_prob

def perplexity(test,model):
    ln_prob_sum = 0.0
    word_count = 0
    for sentence in test:
        ln_prob_sum += calculate_unigram_sentence_ln_prob(sentence, model)
        word_count += len(sentence)
    avg_ln_prob = ln_prob_sum/word_count
    return math.exp(-avg_ln_prob)
```

[18]:
```
test_unigrams = [list(ngrams(sent, n=1)) for sent in padded_tokenized_test]
```

[19]:
```
print(perplexity(train_unigrams,model))
print(perplexity(test_unigrams,model))
```

```
406.8950820766048
376.86063648570286
```

4.2 Q1 MCV

Calculate the perplexity of the model on the wiki test set and answer in MyCourseVille

[20]:
```
wiki_test_unigrams = [list(ngrams(sent, n=1)) for sent in␣
 ↪padded_tokenized_wiki_test]
```

[21]:
```
print(perplexity([list(flatten(wiki_test_unigrams))], model))
```

```
498.25056811239347
```

5 Bigram

Next, you will create a better language model than a unigram (which is not much to compare with). But first, it is very tedious to count every pair of words that occur in our corpus by ourselves. Lucky for us, nltk provides us a simple library which will simplify the process.

[22]:
```python
#example of nltk usage for bigram
sentence = 'I always search google for an answer .'
padded_sentence = list(pad_both_ends(sentence.split(), n=2))

print('This is how nltk generate bigram.')
for w1,w2 in bigrams(padded_sentence):
    print(w1,w2)
print('\n<s> and </s> are used as a start and end of sentence symbol.␣
 ↪respectively.')
```

```
This is how nltk generate bigram.
<s> I
I always
always search
search google
google for
for an
an answer
answer .
. </s>

<s> and </s> are used as a start and end of sentence symbol. respectively.
```

Now, you should be able to implement a bigram model by yourself. Also, you must create a new perplexity calculation for bigram. The result perplexity should be around 50.21 and inf on train and test data.

5.1 TODO #3 Write Bigram Model

[23]:
```python
class BigramModel():
  def __init__(self, data, vocab):
    """
    Args:
        data (list of list of tuples): list of sentences, a sentence is a list␣
 ↪of tuples (w_i, w_{i+1})
    """
    self.bigram_count = defaultdict(lambda: 0)
    self.unigram_count = defaultdict(lambda: 0)
    self.word_count = 0
    self.vocab = vocab
    for sentence in data: # data = list of sentences(list of tuples(w1, w2))
      for w1, w2 in sentence: # sentence = [(w1, w2), (w2, w3), (w3, w4), ... ,␣
 ↪(w_f-1, w_f)]
        self.bigram_count[(w1, w2)] += 1
        self.unigram_count[w1] += 1
        self.word_count += 1
      # count last word
      self.unigram_count[sentence[-1][-1]] += 1
      self.word_count += 1

  def __getitem__(self, bigram):
    """Probability of bigram.
    Args:
        bigram (tuple): (w1, w2)
    Returns:
        P(w2|w1) = count(w1, w2) / count(w1)
    """
    w1, w2 = bigram
    return self.bigram_count[(w1, w2)] / self.unigram_count[w1]
```

5.2 TODO #4 Write Perplexity for Bigram Model

Sum perplexity score at a sentence level, instead of word level

[24]:
```
def calculate_bigram_sentence_ln_prob(bigram_sentence, model):
    ln_prob = 0.0
    for w1, w2 in bigram_sentence:
        prob = model[(w1, w2)]
        ln_prob += math.log(prob) if prob>0 else float('-inf')
    return ln_prob

def perplexity(bigram_data, model):
    ln_prob_sum = 0.0
    n = 0
    for sentence in bigram_data:
        ln_prob_sum = calculate_bigram_sentence_ln_prob(sentence, model)
        n += len(sentence)
    avg_ln_prob = ln_prob_sum/n
    # return 2**(-avg_ln_prob)
    return math.exp(-avg_ln_prob)
```

[25]:
```
train_bigrams = [list(ngrams(sent, n=2)) for sent in padded_tokenized_train]
test_bigrams = [list(ngrams(sent, n=2)) for sent in padded_tokenized_test]
```

[26]:
```
bigram_model_scratch = BigramModel(train_bigrams, vocab)
```

[27]:
```
print(perplexity([list(flatten(train_bigrams))], bigram_model_scratch))
print(perplexity([list(flatten(test_bigrams))[:17]], bigram_model_scratch))
print(perplexity([list(flatten(test_bigrams))], bigram_model_scratch))
```

```
50.21343110065736
24.977802535470772
inf
```

5.3 Q2 MCV

```
[28]: wiki_test_bigrams = [list(ngrams(sent, n=2)) for sent in␣
       ↪padded_tokenized_wiki_test]
```

```
[29]: print(perplexity([list(flatten(wiki_test_bigrams))],bigram_model_scratch))
```

```
inf
```

6 Smoothing

Usually any ngram models have a sparsity problem, which means it does not have every possible ngram of words in the dataset. Smoothing techniques can alleviate this problem. In this section, you will implement three basic smoothing methods laplace smoothing, interpolation for bigram, and Knesey-Ney smoothing.

6.1 TODO #5 write Bigram with Laplace smoothing (Add-One Smoothing)

The result perplexity on training and testing should be:

```
307.29, 364.17 for Laplace smoothing
```

```
[30]: class BigramWithLaplaceSmoothing():

        def __init__(self, data, vocab):
          self.bigram_count = defaultdict(lambda: 0)
          self.unigram_count = defaultdict(lambda: 0)
          self.vocab = vocab
          self.word_count = 0
          for sentence in data: # data = list of sentences(list of tuples(w1, w2))
            for w1, w2 in sentence: # sentence = [(w1, w2), (w2, w3), (w3, w4), ... ,␣
      ↪(w_f-1, w_f)]
              self.bigram_count[(w1, w2)] += 1
              self.unigram_count[w1] += 1
              self.word_count += 1
            # count last word
            self.unigram_count[sentence[-1][-1]] += 1
            self.word_count += 1

        def __getitem__(self, bigram):
          """Probability of bigram.
          Args:
              bigram (tuple): (w1, w2)
          Returns:
              P(w2|w1) = count(w1, w2)+1 / count(w1)+V
          """
          w1, w2 = bigram
          numerator = self.bigram_count[(w1, w2)] + 1
```

```
    denominator = self.unigram_count[w1] + len(self.vocab)
    return numerator / denominator


model = BigramWithLaplaceSmoothing(train_bigrams, vocab)
print(perplexity([list(flatten(train_bigrams))],model),␣
 ↪f"(={round(perplexity([list(flatten(train_bigrams))],model), 2)})")
print(perplexity([list(flatten(test_bigrams))], model),␣
 ↪f"(={round(perplexity([list(flatten(test_bigrams))], model), 2)})")
```

```
307.2932191431376 (=307.29)
364.17463606907467 (=364.17)
```

6.2 Q3 MCV

[31]:
```
print(perplexity([list(flatten(wiki_test_bigrams))],model))
```

```
738.5456651453641
```

6.3 TODO #6 Write Bigram with Interpolation

Set the lambda value as 0.7 for bigram, 0.25 for unigram, and 0.05 for unknown word.

The result perplexity on training and testing should be:

```
62.44, 103.99 for Interpolation
```

[32]:
```
class BigramWithInterpolation():

  def __init__(self, data, vocab, l = 0.7):
    self.unigram_count = defaultdict(lambda: 0.0)
    self.bigram_count = defaultdict(lambda: 0.0)
    self.total_word_count = 0
    self.vocab = vocab
    self.l = l #l for lambda
    for sentence in data:
        for w1, w2 in sentence:
            self.bigram_count[(w1,w2)] += 1.0
            self.unigram_count[w1] += 1.0
            self.total_word_count += 1

        #account of the last word of each sentence
        self.unigram_count[w2] += 1.0
        self.total_word_count += 1

  def __getitem__(self, bigram):
      w1, w2 = bigram
      unigram_prob = self.unigram_count[w2]/self.total_word_count
```

```
        bigram_prob = self.bigram_count[(w1,w2)]/self.unigram_count[w1] if self.
↪unigram_count[w1]>0 else 0

        return 0.7*bigram_prob + 0.25*unigram_prob + 0.05*(1/len(self.vocab))

model = BigramWithInterpolation(train_bigrams, vocab)
print(perplexity([list(flatten(train_bigrams))],model),␣
↪f"(={round(perplexity([list(flatten(train_bigrams))],model), 2)})")
print(perplexity([list(flatten(test_bigrams))], model),␣
↪f"(={round(perplexity([list(flatten(test_bigrams))], model), 2)})")
```

```
62.44269181334268 (=62.44)
103.99017321534633 (=103.99)
```

6.4 Q4 MCV

[33]:
```
print(perplexity([list(flatten(wiki_test_bigrams))],model))
```

```
255.71779470477514
```

6.5 Language modeling on multiple domains

Sometimes, we do not have enough data to create a language model for a new domain. In that case, we can improvised by combining several models to improve result on the new domain.

In this exercise you will try to merge two language models from news and article domains to create a language model for the encyclopedia domain.

[34]:
```
# create encyclopeida data (test data)
encyclo_data=[]
with open('BEST2010/encyclopedia.txt','r',encoding='utf-8') as f:
    for i,line in enumerate(f):
        encyclo_data.append(line.strip()[:-1])
```

(news) First, you should try to calculate perplexity of your bigram with interpolation on encyclopedia data. The perplexity should be around 240.75

[35]:
```
def generate_bigrams(data, vocab):
    """
    Generate bigrams.
    Args:
        data (list of str): List of sentences with '|'.
        vocab (Vocabulary): NLTK Vocabulary object.
    Returns:
        List[List[Tuple[str, str]]]: List of bigrams for each sentence.
    """
    tokenized_sentences = [sentence.split("|") for sentence in data]
    tokenized_sentences = [
        [token if (token in vocab) else "<UNK>" for token in sentence]
        for sentence in tokenized_sentences
    ]
    padded_tokenized_sentences = [
        list(pad_both_ends(sentence, n=2)) for sentence in tokenized_sentences
    ]
    bigrams = [list(ngrams(sent, n=2)) for sent in padded_tokenized_sentences]

    return bigrams


encyclopedia_bigrams = generate_bigrams(encyclo_data, vocab)
```

[36]:
```
# 1) news only on "encyclopedia"
print(perplexity([list(flatten(encyclopedia_bigrams))], model))
```

```
240.74578402349226
```

6.6 TODO #7 - Langauge Modelling on Multiple Domains

Combine news and article datasets to create another bigram model and evaluate it on the encyclopedia data.

(article) For your information, a bigram model with interpolation using article data to test on encyclopedia data has a perplexity of 218.57

[37]:
```
def generate_vocabulary(data):
    """
    Generate a vocabulary from tokenized data.
    Args:
        data (list of str): List of sentences with '|'.
    Returns:
        Vocabulary: NLTK Vocabulary object with unknown word handling.
    """
    tokenized_data = [["<s>"] + sentence.split("|") + ["</s>"] for sentence in␣
 ↪data]
    flat_tokens = list(flatten(tokenized_data))
    vocab = Vocabulary(flat_tokens, unk_cutoff=3)
    return vocab
```

[38]:
```
# 2) article only on "encyclopedia"
best2010_article=[]
with open('BEST2010/article.txt','r',encoding='utf-8') as f:
    for i,line in enumerate(f):
        best2010_article.append(line.strip()[:-1])

combined_total_word_count = 0
for line in best2010_article:
    combined_total_word_count += len(line.split('|'))
```

```
article_vocab = generate_vocabulary(best2010_article)
article_bigrams = generate_bigrams(best2010_article, article_vocab)

article_model = BigramWithInterpolation(article_bigrams, article_vocab)
```

[39]:
```
encyclopedia_bigrams = generate_bigrams(encyclo_data, article_vocab)
print(
    "Perplexity of the bigram model using article data with interpolation␣
 ↪smoothing on encyclopedia test data",
    perplexity([list(flatten(encyclopedia_bigrams))], article_model),
)
```

```
Perplexity of the bigram model using article data with interpolation smoothing
on encyclopedia test data 218.57479345888848
```

[40]:
```
# 3) train on news + article, test on "encyclopedia"
best2010_article_and_news = best2010_article.copy()
with open("BEST2010/news.txt", "r", encoding="utf-8") as f:
    for i, line in enumerate(f):
        best2010_article_and_news.append(line.strip()[:-1])

combined_vocab = generate_vocabulary(best2010_article_and_news)
combined_bigrams = generate_bigrams(best2010_article_and_news, combined_vocab)

combined_model = BigramWithInterpolation(combined_bigrams, combined_vocab)

encyclopedia_bigrams = generate_bigrams(encyclo_data, combined_vocab)
print(
    "Perplexity of the combined Bigram model with interpolation smoothing on␣
 ↪encyclopedia test data",
    perplexity([list(flatten(encyclopedia_bigrams))], combined_model),
)
```

```
Perplexity of the combined Bigram model with interpolation smoothing on
encyclopedia test data 242.88025282580364
```

6.7 TODO #8 - Kneser-ney on "News"

Implement Bigram Knerser-ney LM. The result perplexity should be around 58.18, 93.84 on train and test data. Be careful not to mix up vocab from the above section!

[41]:
```
class BigramKneserNey:
    def __init__(self, data, vocab, discount=0.75):
        self.unigram_counts = defaultdict(lambda: 0.0)
        self.bigram_counts = defaultdict(lambda: 0.0)
        self.vocab = vocab
        self.discount = discount
        self.start_with_unique_counts = defaultdict(set) # f(w_i-1) = |{w :␣
↪c(w_i-1, w)}| # Dict[w_i-1] = set(w)
        self.end_with_unique_counts = defaultdict(set) # f(w) = |{w_i-1 :␣
↪c(w_i-1, w)}| # Dict[w] = set(w_i-1)

        # Count unigrams, bigrams, and unique continuations
        for sentence in data:
            for w1, w2 in sentence:
                self.unigram_counts[w1] += 1
                self.bigram_counts[(w1, w2)] += 1

                self.start_with_unique_counts[w1].add(w2)
                self.end_with_unique_counts[w2].add(w1)
            # count last word in sentence
            self.unigram_counts[w2] += 1

        self.total_unique_bigrams = len(self.bigram_counts)

    def lambda_(self, word) -> float:
        normalized_discount = self.discount / self.unigram_counts[word]
        return normalized_discount * len(self.start_with_unique_counts[word])

    def continuation_probability(self, word) -> float:
        return len(self.end_with_unique_counts[word]) / self.
↪total_unique_bigrams

    def __getitem__(self, bigram) -> float:
        """
        Return: (float)
            P_KN(w2|w1) = max{C(w1, w2) - d, 0} / C(w1) + lambda(w1) *␣
↪P_continuation(w2)
        """
        w1, w2 = bigram
        discounted_bigram_count = max(self.bigram_counts[(w1, w2)] - self.
↪discount, 0)
        return (discounted_bigram_count / self.unigram_counts[w1]) + self.
↪lambda_(w1) * self.continuation_probability(w2)


model = BigramKneserNey(train_bigrams, vocab)
print(perplexity([list(flatten(train_bigrams))],model))
print(perplexity([list(flatten(train_bigrams))[:1000]],model))
print(perplexity([list(flatten(test_bigrams))[:1000]], model))
print(perplexity([list(flatten(test_bigrams))], model))
```

```
58.18312117005813
46.16427141273723
88.87482261840823
93.8399459324311
```

6.8 Q5 MCV

[42]:
```
print(perplexity([list(flatten(wiki_test_bigrams))],model))
```

```
268.6766593898691
```