CSCI 4140: Natural Language Processing

CSCI/DASC 6040: Computational Analysis of Natural Languages

Spring 2025

Homework 1 - N-gram models

Due Sunday, January 26, at 11:59 PM

The learning goals of this assignment are to:

- Understand how to compute language model probabilities using maximum likelihood estimation.
- Implement back-off.
- Have fun using a language model to probabilistically generate texts.

N-gram Language model

- For undergraduates: **100 pts + 10 extra credit pts**
- For graduates: **110 pts**

Preliminaries

In [ ]:
```
import random
from collections import *
import numpy as np
```

We'll start by loading the data. The WikiText language modeling dataset is a collection of tokens extracted from the set of verified Good and Featured articles on Wikipedia.

In [ ]:
```
data = {'test': '', 'train': '', 'valid': ''}

for data_split in data:
    fname = "wiki.{}.tokens".format(data_split)
    with open(fname, 'r') as f_wiki:
```

```
        data[data_split] = f_wiki.read().lower().split()

vocab = list(set(data['train']))
```

Now have a look at the data by running this cell.

In [ ]:
```
print('train : %s ...' % data['train'][:10])
print('dev : %s ...' % data['valid'][:10])
print('test : %s ...' % data['test'][:10])
print('first 10 words in vocab: %s' % vocab[:10])
```

Q1: Train N-gram language model (60 pts)

Complete the following `train_ngram_lm` function based on the following input/output specifications. If you've done it right, you should pass the tests in the cell below.

Input:

- **data**: the data object created in the cell above that holds the tokenized Wikitext data
- **order**: the order of the model (i.e., the "n" in "n-gram" model). If order=3, we compute $p(w_2 | w_0, w_1)$.

Output:

- **lm**: A dictionary where the key is the history and the value is a probability distribution over the next word computed using the maximum likelihood estimate from the training data. Importantly, this dictionary should include *backoff* probabilities as well; e.g., for order=4, we want to store $p(w_3 | w_0,w_1,w_2)$ as well as $p(w_3|w_1,w_2)$ and $p(w_3|w_2)$.

Each key should be a single string where the words that form the history have been concatenated using spaces. Given a key, its corresponding value should be a dictionary where each word type in the vocabulary is associated with its probability of appearing after the key. For example, the entry for the history 'w1 w2' should look like:

```
lm['w1 w2'] = {'w0': 0.001, 'w1' : 1e-6, 'w2' : 1e-6, 'w3': 0.003, ...}
```

In this example, we also want to store `lm['w2']` and `lm['']`, which contain the bigram and unigram distributions respectively.

Hint: You might find the **defaultdict** and **Counter** classes in the **collections** module to be helpful.

In [ ]:
```
def train_ngram_lm(data, order=3):
    """
        Train n-gram language model
    """

    # pad (order-1) special tokens to the left
    # for the first token in the text
    order -= 1
    data = ['<S>'] * order + data #
    lm = defaultdict(Counter)
    for i in range(len(data) - order):
        """
        IMPLEMENT ME!
        """
        #pass
```

In [ ]:
```
def test_ngram_lm():

    print('checking empty history ...')
    lm1 = train_ngram_lm(data['train'], order=1)
    assert '' in lm1, "empty history should be in the language model!"

    print('checking probability distributions ...')
    lm2 = train_ngram_lm(data['train'], order=2)
    sample = [sum(lm2[k].values()) for k in random.sample(list(lm2), 10)]
    assert all([a > 0.999 and a < 1.001 for a in sample]), "lm[history][word] should sum to 1!"

    print('checking lengths of histories ...')
    lm3 = train_ngram_lm(data['train'], order=3)
    assert len(set([len(k.split()) for k in list(lm3)])) == 3, "lm object should store histories of all sizes!"

    print('checking word distribution values ...')
    assert lm1['']['the'] < 0.064 and lm1['']['the'] > 0.062 and \
           lm2['the']['first'] < 0.017 and lm2['the']['first'] > 0.016 and \
           lm3['the first']['time'] < 0.106 and lm3['the first']['time'] > 0.105, \
           "values do not match!"

    print("Congratulations, you passed the ngram check!")

test_ngram_lm()
```

Q2: Generate text from n-gram language model (40 pts)

Complete the following `generate_text` function based on these input/output requirements:

Input:

- **lm**: the lm object is the dictionary you return from the **train_ngram_lm** function
- **vocab**: vocab is a list of unique word types in the training set, already computed for you during data loading.
- **context**: the input context string that you want to condition your language model on, should be a space-separated string of tokens
- **order**: order of your language model (i.e., "n" in the "n-gram" model)
- **num_tok**: number of tokens to be generated following the input context

Output:

- generated text, should be a space-separated string

Hint:

After getting the next-word distribution given history, try using **numpy.random.choice** to sample the next word from the distribution.

In [ ]:

```
# generate text
def generate_text(lm, vocab, context="he is the", order=3, num_tok=25):

    # The goal is to generate new words following the context
    # If context has more tokens than the order of lm,
    # generate text that follows the last (order-1) tokens of the context
    # and store it in the variable `history`
    order -= 1
    history = context.split()[-order:]
    # `out` is the list of tokens of context
    # you need to append the generated tokens to this list
    out = context.split()

    for i in range(num_tok):
        """
        IMPLEMENT ME!
        """
        #pass
```

Now try to generate some texts, generated by ngram language model with different orders.

In [ ]:
```
order = 1
generate_text(train_ngram_lm(data['train'], order=order), vocab, context='he is the', order=order)
```

In [ ]:
```
order = 2
generate_text(train_ngram_lm(data['train'], order=order), vocab, context='he is the', order=order)
```

In [ ]:
```
order = 3
generate_text(train_ngram_lm(data['train'], order=order), vocab, context='he is the', order=order)
```

In [ ]:
```
order = 4
generate_text(train_ngram_lm(data['train'], order=order), vocab, context='he is the', order=order)
```

Q3 : Evaluate the models (10 pts)

Now let's evaluate the models quantitively using the intrinsic metric **perplexity**.

Recall perplexity is the inverse probability of the test text $$\text{PP}(w_1, \dots, w_t) = P(w_1, \dots, w_t)^{-\frac{1}{T}}$$

For an n-gram model, perplexity is computed by $$\text{PP}(w_1, \dots, w_t) = \left[\prod_{t=1}^T P(w_t|w_{t-1},\ldots,w_{t-n+1})\right]^{-\frac{1}{T}}$$

To address the numerical issue (underflow), we usually compute $$\text{PP}(w_1, \dots, w_t) = \exp\left(-\frac{1}{T}\sum_i \log P(w_t|w_{t-1},\ldots,w_{t-n+1})\right)$$

Input:

- **lm**: the language model you trained (the object you returned from the `train_ngram_lm` function)
- **data**: test data
- **vocab**: the list of unique word types in the training set
- **order**: order of the lm

Output:

- the perplexity of test data

Hint:

- If the history is not in the **lm** object, back-off to (n-1) order history to check if it is in **lm**. If no history can be found, just use `1/|V|` where `|V|` is the size of vocabulary.

In [ ]:
```
from math import log, exp
def compute_perplexity(lm, data, vocab, order=3):

    # pad according to order
    order -= 1
    data = ['<S>'] * order + data
    log_sum = 0
    for i in range(len(data) - order):
        h, w = ' '.join(data[i: i+order]), data[i+order]
        """
        IMPLEMENT ME!
        # if h not in lm, back-off to n-1 gram and look up again
        """
        #pass
```

Let's evaluate the language model with different orders. You should see a decrease in perplexity as the order increases. As a reference, the perplexity of the unigram, bigram, trigram, and 4-gram language models should be around 795, 203, 141, and 130 respectively.

In [ ]:
```
for o in [1, 2, 3, 4]:
    lm = train_ngram_lm(data['train'], order=o)
    print('order {} ppl {}'.format(o, compute_perplexity(lm, data['test'], vocab, order=o)))
```