CSCI 4140: Natural Language Processing

CSCI/DASC 6040: Computational Analysis of Natural Languages

Spring 2024

Homework 1 - N-gram models

Due Sunday, January 21, at 11:59 PM

Do not redistribute without the instructor's written permission.

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.
- Compare word-level langauage models and character-level language models.

Part 1: N-gram Language model (60 pts)

Preliminaries

```
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.

```
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.

```
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])
```

```
    train : ['=', 'valkyria', 'chronicles', 'iii', '=', 'senjō', 'no', 'valkyria', '3', ':'] ...
    dev : ['=', 'homarus', 'gammarus', '=', 'homarus', 'gammarus', ',', 'known', 'as', 'the'] ...
    test : ['=', 'robert', '<unk>', '=', 'robert', '<unk>', 'is', 'an', 'english', 'film'] ...
    first 10 words in vocab: ['noble', 'effigies', 'kovacs', '272', 'balista', '攻殻機動隊', '2e', 'creeks', 'laverton', 'refute']
```

Q1.1: Train N-gram language model (25 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.

```
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):
        history = ' '.join(data[i:i+order])
        current_word = data[i+order]

        # Update language model
        for j in range(order):
            partial_history = ' '.join(data[i+j:i+order])
            lm[partial_history][current_word] += 1

        # Update unigram model
        lm[''][current_word] += 1

    # Convert counts to probabilities
    for history, word_counts in lm.items():
        total_count = sum(word_counts.values())
        probabilities = {word: count / total_count for word, count in word_counts.items()}
        lm[history] = probabilities

    return lm


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()
```

```
checking empty history ...
checking probability distributions ...
checking lengths of histories ...
checking word distribution values ...
Congratulations, you passed the ngram check!
```

Q1.2: Generate text from n-gram language model (10 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.

```
# 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):
        # Get the history as a string
        history_str = ' '.join(history)

        # Check if the history is in the language model
        if history_str in lm:
            # Get the next-word distribution given the history
            next_word_dist = lm[history_str]

            # Sample the next word using numpy.random.choice
            next_word = np.random.choice(list(next_word_dist.keys()), p=list(next_word_dist.values()))

            # Append the next word to the output
            out.append(next_word)

            # Update the history for the next iteration
            history = out[-order:]
        else:
            # If history is not in the language model, break the loop
            break

    return ' '.join(out)
```

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

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

```
'he is the'
```

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

```
'he is the headshrinkers match around 5 million households . in wales : what if they attempted to
moments hide their first weekend but if he then lost'
```

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

```
'he is the main factors preventing the transmission of information of the most important diviniti
es of the gedarite arabs   " beat of song in choreographed moves hacked'
```

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

```
'he is the only god , creator of the armenian alphabet . it was the first organelle to be discove
red   and two younger children ( one around'
```

Q1.3 : Evaluate the models (25 pts)

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

Recall perplexity is the inverse probability of the test text

$$\mathrm{PP}(w_1, \dots, w_t) = P(w_1, \dots, w_t)^{-\frac{1}{T}}$$

For an n-gram model, perplexity is computed by

$$\mathrm{PP}(w_1, \dots, w_t) = \left[\prod_{t=1}^{T} P(w_t | w_{t-1}, \dots, w_{t-n+1})\right]^{-\frac{1}{T}}$$

To address the numerical issue (underflow), we usually compute

$$\text{PP}(w_1, \ldots, w_t) = \exp\left(-\frac{1}{T}\sum_i \log P(w_t|w_{t-1}, \ldots, w_{t-n+1})\right)$$

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.

```
from collections import Counter, defaultdict
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
    V = len(vocab)

    for i in range(len(data) - order):
        h, w = ' '.join(data[i: i+order]), data[i+order]

        # If history h is not in lm, back-off to (n-1) gram and look up again
        while h not in lm and order > 1:
            order -= 1
            h = ' '.join(data[i: i+order])

        # If no history can be found, use 1/|V|
        if h not in lm:
            log_sum += -log(1/V)
        else:
            # Look up probability in the language model
            log_sum += -log(lm[h].get(w, 1/V))

    # Compute perplexity
    perplexity = exp(log_sum / len(data))
    return perplexity
```

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.

```
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)))
```

```
 order 1 ppl 652.7888243015223
 order 2 ppl 289.18308911565356
 order 3 ppl 289.21504974686553
 order 4 ppl 289.2197909653991
```

Part 2: Character-level N-gram language model (50 points)

In the lecture, language modeling was defined as the task of predicting the next word in a sequence given the previous words. In this part of the assignment, we will focus on the related problem of predicting the next character or word in a sequence given the previous characters.

Preliminaries

We have to modify how we load the data, by splitting it into characters rather than words. Also, we'll use both upper case and lower case letters.

```
data = {'test': '', 'train': '', 'valid': ''}

for data_split in data:
    fname = "wiki.{}.tokens".format(data_split)
    data[data_split] = list(open(fname, 'r', encoding = 'utf-8').read())

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

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

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

```
train : [' ', '\n', ' ', '=', ' ', 'V', 'a', 'l', 'k', 'y'] ...
dev : [' ', '\n', ' ', '=', ' ', 'H', 'o', 'm', 'a', 'r'] ...
test : [' ', '\n', ' ', '=', ' ', 'R', 'o', 'b', 'e', 'r'] ...
first 10 characters in vocab: ['ê', '₹', '±', '-', '£', 'ä', '.', 'X', 'ç', 'キ']
```

Q2.1: Train N-gram language model (20 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 $p(c_2|c_0, c_1)$.

Output:

- **lm**: A dictionary where the key is the history and the value is a probability distribution over the next character 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 $p(c_3|c_0, c_1, c_2)$ as well as $p(c_3|c_1, c_2)$ and $p(c_3|c_2)$.

Each key should be a single string where the characters that form the history have been concatenated. Given a key, its corresponding value should be a dictionary where each character in the vocabulary is associated with its probability of appearing after the key. For example, the entry for the history 'c1c2' should look like:

```
lm['c1c2'] = {'c0': 0.001, 'c1' : 1e-6, 'c2' : 1e-6, 'c3': 0.003, ...}
```

In this example, we also want to store `lm['c2']` 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.

```
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 = ['~'] * order + data #
    lm = defaultdict(Counter)
    for i in range(len(data) - order):
        history = ''.join(data[i: i+order])
        lm[history][data[i+order]] += 1

    # Convert counts to probabilities
    for history, counts in lm.items():
        total_count = sum(counts.values())
        lm[history] = {char: count / total_count for char, count in counts.items()}

    return lm
```

```
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][character] should sum to 1!"

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

    print('checking character distribution values ...')
    assert lm1['']['t'] < 0.062 and lm1['']['t'] > 0.060 and \
           lm2['t']['h'] < 0.297 and lm2['t']['h'] > 0.296 and \
           lm3['th']['e'] < 0.694 and lm3['th']['e'] > 0.693, \
           "values do not match!"

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

test_ngram_lm()
```

```
checking empty history ...
checking probability distributions ...
checking lengths of histories ...
checking character distribution values ...
Congratulations, you passed the ngram check!
```

Q2.2: Generate text from n-gram language model (10 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 characters 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 string
- **order**: order of your language model (i.e., "n" in the "n-gram" model)
- **num_tok**: number of characters to be generated following the input context

Output:

- generated text, should be a sequence of characters

Hint:

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

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

    # The goal is to generate new characters 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 = list(context)[-order:]
    # `out` is the list of tokens of context
    # you need to append the generated tokens to this list
    out = list(context)

    for i in range(num_tok):
  # Get the history as a string
        history_str = ''.join(history)

        # Check if the history is in the language model
        if history_str in lm:
            # Get the next-character distribution given the history
            next_char_dist = lm[history_str]

            # Sample the next character using numpy.random.choice
            next_char = np.random.choice(list(next_char_dist.keys()), p=list(next_char_dist.values()))

            # Append the next character to the output
            out.append(next_char)

            # Update the history for the next iteration
            history = out[-order:]
        else:
            # If history is not in the language model, break the loop
            break

    return ''.join(out)
```

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

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

```
'he is the'
```

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

```
'he is thed sth be lernieas 20 ) ct'
```

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

```
'he is thernits the fire Val bee fa'
```

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

```
'he is the Piazz , Prespecial bes ,'
```

Q2.3 : Evaluate the models (20 pts)

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

Recall perplexity is the inverse probability of the test text

$$\mathrm{PP}(w_1,\ldots,w_t) = P(w_1,\ldots,w_t)^{-\frac{1}{T}}$$

For an n-gram model, perplexity is computed by

$$\mathrm{PP}(w_1,\ldots,w_t) = \left[\prod_{t=1}^{T} P(w_t|w_{t-1},\ldots,w_{t-n+1})\right]^{-\frac{1}{T}}$$

To address the numerical issue (underflow), we usually compute

$$\mathrm{PP}(w_1,\ldots,w_t) = \exp\left(-\frac{1}{T}\sum_i \log P(w_t|w_{t-1},\ldots,w_{t-n+1})\right)$$

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 characters 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.

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

    # pad according to order
    order -= 1
```