Applied Probability Homework - 60 points total

This file needs the accompanying file train.tsv.

Part 1: Bayesian Tomatoes (25 points)

In this part of the assignment, you'll implement the final part of a Naive Bayes classifier that performs sentiment analysis on sentences from movie reviews. Upload and read the train.tsv file that contains sentences and phrases that have been rated 0 to 4 for sentiment ranging from very negative to very positive. We'll only be working with the full sentences.

In [1]:
```
import nltk
nltk.download('punkt')  # Data for tokenization
```

```
[nltk_data] Downloading package punkt to /Users/yunzheyu/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[1]: True

In [2]:
```
with open('train.tsv', 'r') as textfile:
  ratings_data = textfile.read()
```

In [17]:
```
from nltk.tokenize import word_tokenize

def tokenize(sentence):
    """ Returns list of tokens (strings) from the sentence.

    Sets to lowercase and runs NLTK tokenizer.

    Args:
        sentence (string):  the string to tokenize
    """
    return [t.lower() for t in word_tokenize(sentence)]

class ModelInfo:
    """ Contains all counts from the data necessary to do Naive Bayes or bab

    Attributes:
        word_counts (List[Dict[string,int]]):  counts of tokens, indexed by
        sentiment_counts (List[int]):  counts of sentences with each sentime
        total_words (List[int]):  counts of words in each sentiment
        total_examples (int):  total sentence count
        bigram_counts (List[Dict[string, int]]):  counts of bigrams (two-wor
                for each sentiment, indexed by string 'word1_word2'
    """
```

```
    def __init__(self):
        self.word_counts = [{}, {}, {}, {}, {}]
        self.sentiment_counts = [0, 0, 0, 0, 0]
        self.total_words = [0, 0, 0, 0, 0]
        self.total_examples = 0
        self.bigram_counts = [{}, {}, {}, {}, {}]
        self.trigram_counts = [{}, {}, {}, {}, {}]


    def update_word_counts(self, sentence, sentiment):
        """ Consume a sentence and update all counts.

        To "tokenize" the sentence we'll make use of NLTK, a widely-used Pyt
        processing (NLP) library.  This will handle otherwise onerous tasks
        from their attached words.  (Unless the periods are decimal points .
        than you might think.)  The result of tokenization is a list of indi
        words or their equivalent.

        Args:
            sentence (string):  The example sentence.
            sentiment (int):  The sentiment label.
        """

        # Get the relevant dicts for the sentiment
        s_word_counts = self.word_counts[sentiment]
        tokens = tokenize(sentence)
        for i in range(len(tokens)):
            self.total_words[sentiment] += 1
            s_word_counts[tokens[i]] = s_word_counts.get(tokens[i], 0) + 1
            if i < len(tokens) - 1:
                bigram = tokens[i] + '_' + tokens[i+1]
                self.bigram_counts[sentiment][bigram] = self.bigram_counts[s
            if i < len(tokens) - 2:
                trigram = tokens[i] + '_' + tokens[i+1] + '_' + tokens[i+2]
                self.trigram_counts[sentiment][trigram] = self.trigram_count

FIRST_SENTENCE_NUM = 1

def get_models(ratings_data):
    """Returns a model_info object, consuming a string for examples."""
    next_fresh = FIRST_SENTENCE_NUM
    info = ModelInfo()
    for line in ratings_data.splitlines():
        if line.startswith("---"):
            return info
        fields = line.split("\t")
        try:
            sentence_num = int(fields[1])
            if sentence_num <= next_fresh:
                continue
            next_fresh += 1
            sentiment = int(fields[3])
            info.sentiment_counts[sentiment] += 1
            info.total_examples += 1
            info.update_word_counts(fields[2], sentiment)
        except ValueError:
```

```
            # Some kind of bad input?  Unlikely with our provided data
            continue
    return info

model_info = get_models(ratings_data)
```

(1, 20 points) Complete naive_bayes_classify(), below. It should take a ModelInfo object and use the counts stored therein to give the most likely class according to a Naive Bayes calculation, and the log likelihood of that class. For priors on the sentiment, use the actual frequencies with which each sentiment is used. Notice that there are 5 different classes to compare. Use the OUT_OF_VOCAB_PROB constant for any tokens that haven't been seen for a particular sentiment in the data.

In [4]:
```
import math
CLASSES = 5
OUT_OF_VOCAB_PROB = 0.0000000001

""" naive_bayes_classify:  takes a ModelInfo containing all counts necessary
    and a String to be classified.  Returns a number indicating sentiment ar
    of that sentiment (two comma-separated return values).
"""
def naive_bayes_classify(info, sentence):
    """ Use a Naive Bayes model to return sentence's most likely classificat

    Args:
        info (ModelInfo):  a ModelInfo containing the counts from the traini
        sentence (string):  the test sentence to classify

    Returns:
        int for the best sentiment
        float for the best log probability (unscaled, just log(prior * produ
    """
      # Tokenize the sentence
    tokens = tokenize(sentence)

    # Initialize variables to store the best class and its log probability
    best_class = None
    best_log_prob = -float('inf')

    # Iterate over each class (sentiment)
    for c in range(CLASSES):
        # Calculate the log prior: log(P(c)) = log(sentiment_counts[c] / tot
        log_prior = math.log(info.sentiment_counts[c] / info.total_examples)

        # Initialize the log likelihood: log(P(word|c))
        log_likelihood = 0

        # Calculate the log likelihood for each word in the sentence
        for word in tokens:
            word_count = info.word_counts[c].get(word, 0)
            if word_count == 0:
                # Use OUT_OF_VOCAB_PROB for words not in the vocabulary
                log_likelihood += math.log(OUT_OF_VOCAB_PROB)
            else:
```

```
                # Calculate the probability: P(word|c) = count(word, c) / to
                prob_word_given_class = word_count / info.total_words[c]
                log_likelihood += math.log(prob_word_given_class)

        # Calculate the total log probability for the class: log(P(c)) + sum
        total_log_prob = log_prior + log_likelihood

        # Update the best class and its probability if this class is better
        if total_log_prob > best_log_prob:
            best_class = c
            best_log_prob = total_log_prob

    return best_class, best_log_prob
```

In [5]:
```
# Tests
print(naive_bayes_classify(model_info, "I hate this movie")) # Should returr
```

```
(0, -25.947997071867018)
```

In [6]:
```
print(naive_bayes_classify(model_info, "A joyous romp"))    # Should return
```

```
(4, -22.9049498861873)
```

In [7]:
```
print(naive_bayes_classify(model_info, "notaword")) # Should return 3, -24.3
```

```
(3, -24.32724312507062)
```

2, 5 points) Naive Bayes is sometimes called a "linear" classifier; let's explore why. Suppose I have two classes C_1 and C_2 and two observable features A and B; each feature can take on 3 values. Feature A's conditional distribution is [1/2, 1/4, 1/4] for class 1, [1/4, 1/4, 1/2] for class 2. Feature B's conditional distribution is [1/4, 1/2, 1/4] for class 1, [1/2, 1/4, 1/4] for class 2. The two classes each have a prior of 1/2. Given boolean values (i.e. valued 0 or 1) a_0, a_1, a_2 and b_0, b_1, b_2 to represent the observations of feature A and B, derive an equation for each class that gives the log likelihood of that class given the observations. (Use base 2 for your logs to make the equations simpler.) Then, use these log likelihoods to come up with a linear inequality (a weighted sum of $a_0, \ldots, b_2$ that is compared to a constant) that decides whether an example belongs to class 1.

Answer

Naive Bayes Formula for Log Likelihood

For a class (C), the log likelihood given observations is:

$$\log P(C|\text{observations}) = \log P(C) + \sum \log P(\text{feature value}|C)$$

Since we are using base 2 for logs, and the priors for each class are $(\frac{1}{2})$, the log prior $(\log P(C))$ for both classes is $(\log_2(\frac{1}{2}) = -1)$.

Feature A and B Conditional Distributions

- Feature A for (C_1) : $(\left[\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\right])$, and for (C_2) : $(\left[\frac{1}{4}, \frac{1}{4}, \frac{1}{2}\right])$
- Feature B for (C_1) : $(\left[\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\right])$, and for (C_2) : $(\left[\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\right])$

Log Likelihood Equations

For class (C_1):

$$\log P(C_1|a_0, a_1, a_2, b_0, b_1, b_2) = -1 + (a_0 \cdot \log_2(\frac{1}{2}) + a_1 \cdot \log_2(\frac{1}{4}) + a_2 \cdot \log_2(\frac{1}{4})) + ($$

For class (C_2):

$$\log P(C_2|a_0, a_1, a_2, b_0, b_1, b_2) = -1 + (a_0 \cdot \log_2(\frac{1}{4}) + a_1 \cdot \log_2(\frac{1}{4}) + a_2 \cdot \log_2(\frac{1}{2})) + ($$

Part 2: Bigram and Trigram Beam Search Babblers (35 points)

Even complex neural networks like ChatGPT can use search to think a little bit ahead, and choose options that will work out better later on down the line. Beam search is commonly used for this purpose.

The steps below refer to *bigrams*, which are sequences of two words (like "I see"), and *trigrams*, which are sequences of 3 words ("I see you"). A bigram model of language tracks the likelihood of one word following another (P("see" | "I")). A trigram model tracks the likelihood of one word following two other words (P("you" | "I see")). Both were highly influential models of language in natural language processing, but running them in the "forward direction" for production reveals their limitations. (ChatGPT effectively conditions on *all* previous words in the prompt and response so far.)

1 (10 points): Create a function "babble" that takes as input a starting word *start*, a sentiment number (0-4), a ModelInfo object, and a number of additional words to babble, n. Start with the starting word, and for n steps, add a space to that string followed by the single most likely word in terms of conditional probability $P(word_i|word_{i-1}, sentiment)$, which you can compute from the ModelInfo. Return the string you've created. (Note that bigram counts are stored with key 'word1_word2'.)

In [8]:
```
def babble(start, sentiment, model_info, n):
    current_word = start.lower()
    sentence = start

    for _ in range(n):
        next_word = None
        max_count = -1  # Initialize with -1 to ensure any count is consider
        bigram_prefix = current_word + '_'
```

```
        # Find the most likely next word
        for bigram, count in model_info.bigram_counts[sentiment].items():
            if bigram.startswith(bigram_prefix):
                # Only consider the bigram if it's strictly more common
                if count > max_count:
                    max_count = count
                    next_word = bigram[len(bigram_prefix):]

        if next_word:
            sentence += ' ' + next_word
            current_word = next_word
        else:
            break

    return sentence
babble('acting', 0, model_info, 10) # Expect "acting , and the movie . ' anc
```

Out[8]: "acting , and the movie . ' and the movie ."

In [9]:
```
babble('acting', 4, model_info, 10) # Expect "acting , and the film . ' . '
```

Out[9]: "acting , and the film . ' . ' . '"

2 (15 points): Now, write a similar function beam_babble(start, sentiment, info, n, k) that performs beam search with beam size k from start word "start" out to n steps, and returns the most likely sequence. This should still use the conditional probabilities $P(word_i|word_{i-1}, sentiment)$ that you used for the previous babbler.

The solution made use of the following helper functions: make_new_seq(old_words, new_word, old_prob, bigram_prob), which took the old_words and old_prob from an existing sequence and updated it with the new word and bigram probability; bigram_prob(info, sentiment, prev_word, word), which used the word at the end of the sequence and the new word to calculate $P(word_i|word_{i-1}, sentiment)$; and top_k(seqs, k), which returned the top k sequences of a list of (wordlist, prob) entries representing sequences. You can have top_k's code, and the tests for the others are left here if you want them.

In [4]:
```
import heapq
import numpy as np
# TODO def beam_babble(start, sentiment, info, n, k):

# Optional TODO def make_new_seq(old_words, new_word, old_prob, bigram_prob)

# Optional TODO def bigram_prob(info, sentiment, prev_word, word):

def beam_babble(start, sentiment, info, n, k):
    """Generate text using beam search with a beam size of k, with error har
    start_word = start.lower()
    initial_seq = ([start_word], 1)  # Starting with the initial word and pr
    beam = [initial_seq]

    for _ in range(n):
```

```
        all_candidates = []

        for seq in beam:
            prev_word = seq[0][-1]  # Last word in the current sequence
            expansions_found = False
            words_list = list(info.word_counts[sentiment].keys())

            # Check if words_list is not empty before proceeding
            if words_list:
                for word in words_list:
                    bigram = prev_word + '_' + word
                    if bigram in info.bigram_counts[sentiment]:
                        new_prob = bigram_prob(info, sentiment, prev_word, w
                        if new_prob > 1e-5:  # Consider only probable bigram
                            new_seq = make_new_seq(seq[0], word, seq[1], new
                            all_candidates.append(new_seq)
                            expansions_found = True

            # If no expansions are found, keep the current sequence (prevent
            if not expansions_found and len(seq[0]) < n:
                # Add a low probability continuation to keep the sequence go
                if words_list:
                    dummy_next_word = words_list[0]
                    new_seq = make_new_seq(seq[0], dummy_next_word, seq[1],
                    all_candidates.append(new_seq)

        # Keep only the top k sequences
        beam = top_k(all_candidates, k)

    # Return the highest probability sequence along with its probability
    return max(beam, key=lambda x: x[1]) if beam else ([], 0)

def top_k(seqs, k):
    seqs.sort(key = lambda x : x[1], reverse=True)
    return seqs[0:k]

def bigram_prob(info, sentiment, prev_word, word):
    bigram = f"{prev_word}_{word}"
    bigram_count = info.bigram_counts[sentiment].get(bigram, 0)
    prev_word_count = info.word_counts[sentiment].get(prev_word, 0)

    # If the bigram has been seen, calculate its probability normally
    if bigram_count > 0:
        return bigram_count / prev_word_count
    # If the bigram has not been seen, return a small fixed probability
    else:
        return 1e-7


def make_new_seq(old_words, new_word, old_prob, bigram_prob):
    """Update a sequence with a new word and probability"""
    new_seq = old_words + [new_word]
    new_prob = old_prob * bigram_prob
    return new_seq, new_prob
```

In [11]:
```
# Testing
seqs = [(['hello', 'there'], 0.5), (['well', 'goodbye'], 0.2), (['hello', 'h
print(top_k(seqs, 2)) # Should be [(['hello', 'there'], 0.5), (['hello', 'he
```

```
[(['hello', 'there'], 0.5), (['hello', 'hello'], 0.3)]
```

In [12]:
```
model_info = ModelInfo()
model_info.update_word_counts('hello hello goodbye goodbye', 1)
print(model_info.bigram_counts)
print(bigram_prob(model_info, 1, 'hello', 'hello')) # Expect 0.5
print(bigram_prob(model_info, 1, 'hello', 'goodbye')) # Expect 0.5
print(bigram_prob(model_info, 1, 'goodbye', 'goodbye')) # Expect 0.5 (transi
print(bigram_prob(model_info, 1, 'goodbye', 'hello')) # Not seen, expect tir
```

```
[{}, {'hello_hello': 1, 'hello_goodbye': 1, 'goodbye_goodbye': 1}, {}, {},
{}]
0.5
0.5
0.5
1e-07
```

In [13]:
```
print(make_new_seq(['hello', 'hello'], 'goodbye', 0.2, 0.1))
```

```
(['hello', 'hello', 'goodbye'], 0.020000000000000004)
```

In [18]:
```
"""
Expect
(['acting',
  'talents',
  'wasting',
  'away',
  'inside',
  'unnecessary',
  'prologue',
  ',',
  'but',
  'it',
  "'s"],
 5.982521449703316e-07)
"""
beam_babble('acting', 0, model_info, 10, 20)
```

Out[18]:
```
(['acting',
  'talents',
  'wasting',
  'away',
  'inside',
  'unnecessary',
  'prologue',
  ',',
  'but',
  'it',
  "'s"],
 5.982521449703316e-07)
```

In [182…
```
# Check if 'acting' is in the vocabulary for sentiment 0
print("'acting' in vocabulary for sentiment 0:", 'acting' in model_info.word
```

```python
# Optionally, print some of the vocabulary for sentiment 0 to inspect it
print("Some words in vocabulary for sentiment 0:", list(model_info.word_cour
```

```
'acting' in vocabulary for sentiment 0: True
Some words in vocabulary for sentiment 0: ['hampered', '--', 'no', ',', 'pa
ralyzed', 'by', 'a', 'self-indulgent', 'script', '...']
```

In [183…

```python
"""
Expect
(['acting',
  'bond',
  ';',
  'i',
  "'ve",
  'ever',
  'seen',
  'before',
  'swooping',
  'aerial',
  'shots'],
 1.3664786392059117e-07)
"""
beam_babble("acting", 4, model_info, 10, 20)
```

Out[183]:

```
([' acting',
  'bond',
  ';',
  'i',
  "'ve",
  'ever',
  'seen',
  'before',
  'swooping',
  'aerial',
  'shots'],
 1.3664786392059117e-07)
```

3 (10 points): If the results still seem a little lackluster, it's because we're only conditioning on the previous word - it has no memory of what it was just saying. We can do a little better at the expense of plagiarizing the text more often - by using "trigrams" to calculate $P(w_n|w_{n-2}, w_{n-1}, sentiment)$. Rewrite your beam search code below to use trigrams. The counts of how often triplets of words appear - $w_i \wedge w_{i+1} \wedge w_{i+2}$ - have already been stored in the ModelInfo object. A trigram_prob() function similar to the previous problem step's bigram_prob() has some tests left here for you.

In [18]:

```python
def trigram_prob(info, sentiment, prev_word1, prev_word2, word):
    trigram_key = f'{prev_word1}_{prev_word2}_{word}'
    trigram_count = info.trigram_counts[sentiment].get(trigram_key, 0)

    # This should be the count of just 'prev_word1_prev_word2' bigram, not t
    prev_bigram_key = f'{prev_word1}_{prev_word2}'
    prev_bigram_count = info.bigram_counts[sentiment].get(prev_bigram_key, 0
```

```
    if trigram_count == 0:
        return 1e-7  # A tiny probability for unseen trigrams
    else:
        return trigram_count / prev_bigram_count
```

In [19]:
```
def beam_babble_trigram(start, start2, sentiment, info, n, k):
    initial_seq = ([start, start2], 1.0)  # Starting sequence with two words
    beam = [initial_seq]

    for _ in range(n - 2):  # Generate n-2 more words
        all_candidates = []

        for seq, prob in beam:
            prev_word1, prev_word2 = seq[-2], seq[-1]  # Last two words in t
            expansions_found = False

            for word in info.word_counts[sentiment].keys():
                new_prob = trigram_prob(info, sentiment, prev_word1, prev_wo
                if new_prob > 0:  # Only consider valid expansions
                    new_seq = seq + [word]
                    all_candidates.append((new_seq, new_prob * prob))
                    expansions_found = True

            # If no valid expansions for this sequence, it's carried over wi
            if not expansions_found:
                all_candidates.append((seq, prob))

        # Sort candidates by their probability and keep only the top k seque
        beam = sorted(all_candidates, key=lambda x: x[1], reverse=True)[:k]

    # Choose the sequence with the highest probability
    best_seq, best_prob = max(beam, key=lambda x: x[1]) if beam else ([], 0)

    # Normalize the final probability by the length of the sequence for fair
    normalized_prob = best_prob / len(best_seq) if best_seq else 0
    return best_seq, normalized_prob
```

In [7]:
```
model_info = ModelInfo()
model_info.update_word_counts('hello hello hello goodbye goodbye goodbye goo
print(model_info.trigram_counts)
print(trigram_prob(model_info, 1, 'hello', 'hello', 'goodbye')) # Expect 0.5
print(trigram_prob(model_info, 1, 'goodbye', 'goodbye', 'hello')) # Expect 0
```

```
[{}, {'hello_hello_hello': 1, 'hello_hello_goodbye': 1, 'hello_goodbye_good
bye': 1, 'goodbye_goodbye_goodbye': 2, 'goodbye_goodbye_hello': 1}, {}, {},
{}]
0.5
0.3333333333333333
```

In [20]:
```
beam_babble_trigram('the', 'acting', 0, model_info, n=10, k=20)
# Expect (['the','acting','is', 'amateurish', ',', 'quasi-improvised', 'acti
```

Out[20]:
```
(['the',
  'acting',
  'is',
  'amateurish',
  ',',
  'quasi-improvised',
  'acting',
  'exercise',
  'shot',
  'on'],
 0.016666666666666666)
```

In [53]:
```
beam_babble_trigram('the', 'acting', 4, model_info, n=10, k=20)
```

Out[53]:
```
(['the',
  'acting',
  'in',
  'pauline',
  'and',
  'paulette',
  'is',
  'good',
  'all',
  'round'],
 0.1)
```

Compare to this line in the original train.tsv: "The acting in Pauline And Paulette is good all round, but what really sets the film apart is Debrauwer's refusal to push the easy emotional buttons." We didn't intend for this copying to happen, but it just so happened that this word sequence was rare.

When you're done, use "File->Download .ipynb" and upload your .ipynb file to Blackboard, along with a PDF version (File->Print->Save as PDF) of your assignment.