Assignment 5 - Natural Language Processing

February 20, 2024

1 Assignment 5 - Natural Language Processing

- Student 1 - Luca Modica
- Student 2 - Hugo Alves Henriques E Silva

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import math

sns.set_style()
%matplotlib inline
```

1.1 Reading data

[ ]:
```
from collections import Counter
import re

# Paths to the files
de_file_path = 'dat410_europarl/europarl-v7.de-en.lc.de'
en_de_file_path = 'dat410_europarl/europarl-v7.de-en.lc.en'
fr_file_path = 'dat410_europarl/europarl-v7.fr-en.lc.fr'
en_fr_file_path = 'dat410_europarl/europarl-v7.fr-en.lc.en'
sv_file_path = 'dat410_europarl/europarl-v7.sv-en.lc.sv'
en_sv_file_path = 'dat410_europarl/europarl-v7.sv-en.lc.en'
```

1.2 Warmup

[ ]:
```
def get_word_frequencies(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        text = file.read().lower()
        words = re.findall(r'\b\w+\b', text)
        word_freq = Counter(words)
    return word_freq
```

```
[ ]: # Get word frequencies for German-English pair
de_word_freq = get_word_frequencies(de_file_path)
en_de_word_freq = get_word_frequencies(en_de_file_path)

# Print the 10 most common words in German and English (German-English pair)
de_common_words = de_word_freq.most_common(10)
en_de_common_words = en_de_word_freq.most_common(10)

# Get word frequencies for French-English pair
fr_word_freq = get_word_frequencies(fr_file_path)
en_fr_word_freq = get_word_frequencies(en_fr_file_path)

# Get word frequencies for Swedish-English pair
sv_word_freq = get_word_frequencies(sv_file_path)
en_sv_word_freq = get_word_frequencies(en_sv_file_path)

# Print the 10 most common words in French, English (French-English pair),␣
 ↪Swedish, and English (Swedish-English pair)
fr_common_words = fr_word_freq.most_common(10)
en_fr_common_words = en_fr_word_freq.most_common(10)
sv_common_words = sv_word_freq.most_common(10)
en_sv_common_words = en_sv_word_freq.most_common(10)

print("Most common words in German:")
print(", ".join(
    [f"{word} ({count} occurrences)" for word, count in de_common_words]))

print("Most common words in English (German-English pair):")
print(", ".join(
    [f"{word} ({count} occurrences)" for word, count in en_de_common_words]))

print("Most common words in French:")
print(", ".join(
    [f"{word} ({count} occurrences)" for word, count in fr_common_words]))

print("Most common words in English (French-English pair):")
print(", ".join(
    [f"{word} ({count} occurrences)" for word, count in en_fr_common_words]))

print("Most common words in Swedish:")
print(", ".join(
    [f"{word} ({count} occurrences)" for word, count in sv_common_words]))

print("Most common words in English (Swedish-English pair):")
print(", ".join(
    [f"{word} ({count} occurrences)" for word, count in en_sv_common_words]))
```

```
[ ]: # Calculate the total word counts and the counts for 'speaker' and 'zebra'␣
      ↪across all English files
     total_words = sum(en_de_word_freq.values()) + sum(en_fr_word_freq.values()) +␣
      ↪sum(en_sv_word_freq.values())
     speaker_count = en_de_word_freq['speaker'] + en_fr_word_freq['speaker'] +␣
      ↪en_sv_word_freq['speaker']
     zebra_count = en_de_word_freq['zebra'] + en_fr_word_freq['zebra'] +␣
      ↪en_sv_word_freq['zebra']

     # Calculate probabilities
     prob_speaker = speaker_count / total_words
     prob_zebra = zebra_count / total_words

     print("Total words:", total_words)
     print("Speaker count:", speaker_count)
     print("Zebra count:", zebra_count)
     print("Probability of 'speaker':", prob_speaker)
     print("Probability of 'zebra':", prob_zebra)
```

1.3 Language modeling

```
[ ]: from nltk.tokenize import word_tokenize
     from nltk.util import bigrams

     # Function to tokenize corpus into bigrams with start and end tokens
     def create_bigrams(text):
         sentences = text.split('\n')
         bigram_list = []
         for sentence in sentences:
             tokens = ['<START>'] + word_tokenize(sentence)
             bigram_list.extend(list(bigrams(tokens)))
         return bigram_list

     # Read the English text files from all three pairs to create a single corpus
     corpus_de_en = open(en_de_file_path, 'r', encoding='utf-8').read()
     corpus_fr_en = open(en_fr_file_path, 'r', encoding='utf-8').read()
     corpus_sv_en = open(en_sv_file_path, 'r', encoding='utf-8').read()

     # Combine the corpora
     combined_corpus = '\n'.join([corpus_de_en, corpus_fr_en, corpus_sv_en])

     # Create bigrams from the combined corpus
     bigram_list = create_bigrams(combined_corpus)

     # Calculate bigram and unigram counts
```

```
unigram_counts = Counter([unigram for bigram in bigram_list for unigram in␣
↪bigram])
bigram_counts = Counter(bigram_list)

# Function to calculate bigram probabilities using MLE
def calculate_bigram_prob(bigram):
    return bigram_counts[bigram] / unigram_counts[bigram[0]]

# Test the function with an example bigram
example_bigram = ('<START>', 'the')
print("Probability of", example_bigram, ":",␣
↪calculate_bigram_prob(example_bigram))
example_bigram = ('the', 'zebra')
print("Probability of", example_bigram, ":",␣
↪calculate_bigram_prob(example_bigram))
```

[ ]:
```
def calculate_sentence_prob(sentence):
    sentence_bigram_list = create_bigrams(sentence)
    probability = 1
    for bigram in sentence_bigram_list:
        probability *= calculate_bigram_prob(bigram)
    return probability


print(
    f'Probability of "why are no-smoking areas not enforced ?":␣
↪{calculate_sentence_prob("why are no-smoking areas not enforced ?")}')
print(
    f'Probability of "the door is green": {calculate_sentence_prob("the door is␣
↪green")}')
print(
    f'Probability of "we pass": {calculate_sentence_prob("we pass")}')
```

When we encounter a word that did not appear in the training texts, this will result in a probability of zero for any bigram containing this word, making the probability of the entire sentence zero. This is a common issue in language modeling known as the zero-probability problem, and it can be handled using techniques like Laplace (add-one) smoothing.

If the sentence is very long, the probability of the sentence will tend to be very small due to the multiplication of probabilities, which can lead to underflow problems in computers. One way to handle this is by working with the log probabilities instead of the raw probabilities.

[ ]:
```
# Calculate the vocabulary size
vocabulary_size = len(unigram_counts)

# Function to calculate bigram probabilities using Laplace smoothing
def calculate_bigram_log_prob_with_laplace(bigram):
    numerator = bigram_counts[bigram] + 1
    denominator = unigram_counts[bigram[0]] + vocabulary_size
    return math.log(numerator) - math.log(denominator)


#calculate probability of a sentence
def calculate_sentence_prob_improved(sentence):
    tokens = ['<START>'] + word_tokenize(sentence.lower())
    probability = 0
    for i in range(len(tokens) - 1):
        bigram = (tokens[i], tokens[i + 1])
        probability += calculate_bigram_log_prob_with_laplace(bigram)
    return probability


print(
    f'Log probability of "why are no-smoking areas not enforced ?":␣
  ↪{calculate_sentence_prob_improved("why are no-smoking areas not enforced ?
  ↪")}')
print(
    f'Log probability of "the door is open":␣
  ↪{calculate_sentence_prob_improved("the door is open")}')
print(
    f'Log probability of "the door is green":␣
  ↪{calculate_sentence_prob_improved("the door is green")}')
print(
    f'Log probability of "we pass": {calculate_sentence_prob_improved("we␣
  ↪pass")}')
```

The more negative a log probability is, the less likely the sentence is.

1.4 Translation modeling

```
[ ]: import string

def tokenize_corpus(corpus, add_null=False):
    """Tokenize the input corpus (a list of sentences) into a list of lists of␣
  ↪tokens.
    Optionally add a NULL token at the beginning of each sentence."""
    clean_corpus = [sentence.translate(str.maketrans(
        '', '', string.punctuation)) for sentence in corpus]
    tokenized_corpus = [sentence.lower().split() for sentence in clean_corpus]
    if add_null:
        for sentence in tokenized_corpus:
            sentence.insert(0, "<NULL>")
    return tokenized_corpus
```

```
def initialize_translation_prob(corpus_english, corpus_foreign):
    """Initialize translation probabilities with a lower probability for NULL.
↪"""

    word_correspondence = {}

    for sentence_e, sentence_f in zip(corpus_english, corpus_foreign):
        for word_e in sentence_e:
            if word_e not in word_correspondence:
                word_correspondence[word_e] = []
            for word_f in sentence_f:
                if word_f not in word_correspondence[word_e]:
                    word_correspondence[word_e] += [word_f]


    translation_prob = {}
    null_prob = 0.00001


    for word_e in word_correspondence:
        for word_f in word_correspondence[word_e]:
            if word_f == "<NULL>":
               translation_prob[(word_e, word_f)] = null_prob
            else:
                translation_prob[(word_e, word_f)] = (1 - null_prob) /␣
↪(len(word_correspondence[word_e]) - 1)
            #translation_prob[(word_e, word_f)] = 1 /␣
↪len(word_correspondence[word_e])

    return translation_prob




print(tokenize_corpus(["The dog runs", "The cat sleeps"]))
print(initialize_translation_prob(tokenize_corpus(["The dog runs", "The cat␣
↪sleeps", "I am"]), tokenize_corpus(["Le chien court", "Le chat dort", "Je␣
↪suis"], add_null=True)))
```

[ ]:
```
from collections import defaultdict

def ibm_model_1(corpus_english, corpus_foreign, iterations=10):
    corpus_foreign_tokens = tokenize_corpus(corpus_foreign, add_null=True)  #␣
↪foreign language corpus
    corpus_english_tokens = tokenize_corpus(corpus_english)  # English corpus,␣
↪with null tokens
```

```
    # Initialize translation probabilities uniformly
    translation_prob = initialize_translation_prob(corpus_english_tokens,␣
↪corpus_foreign_tokens)

    for iteration in range(iterations):
        count_ef = defaultdict(float)
        total_e = defaultdict(float)

        # E-step: Expectation
        for sentence_e, sentence_f in zip(corpus_english_tokens,␣
↪corpus_foreign_tokens):
            for word_f in sentence_f:
                s_total_word_e = sum(translation_prob[(word_e, word_f)] for␣
↪word_e in sentence_e)
                for word_e in sentence_e:
                    delta = translation_prob[(word_e, word_f)] / s_total_word_e

                    # Update counts
                    count_ef[(word_e, word_f)] += delta
                    total_e[word_e] += delta

        # M-step: Maximization
        for (word_e, word_f), count in count_ef.items():
            translation_prob[(word_e, word_f)] = count / total_e[word_e]


        # normalize probabilities
        new_dict = {}
        for key, value in translation_prob.items():
            if key[0] not in new_dict:
                new_dict[key[0]] = value
            else:
                new_dict[key[0]] += value

        for key, value in translation_prob.items():
            translation_prob[key] = value / new_dict[key[0]]

    print(translation_prob)
    return translation_prob
```

[ ]:
```
# Example usage (using dummy data):
corpus1 = ["the house", "the book", "a big house"]
corpus2 = ["das haus", "das buch", "ein großes haus"]  # Assuming German for␣
↪demonstration
```

```
print(initialize_translation_prob(tokenize_corpus(corpus1),␣
↪tokenize_corpus(corpus2, add_null=True)))

translation_prob = ibm_model_1(corpus1, corpus2, iterations=100)

# Find translations for a specific word (e.g., "house")
translations_for_word = {pair[1]: prob for pair, prob in translation_prob.
↪items() if pair[0] == "house"}
# Sort translations by probability
sorted_translations = sorted(translations_for_word.items(), key=lambda item:␣
↪item[1], reverse=True)

# Print top N translations
print("Top translations for 'house':")
summm = 0
for foreign_word, prob in sorted_translations[:10]:
    print(f"{foreign_word}: {prob}")
    summm += prob
print(summm)
```

test with swede

```
[ ]: #Write code that implements the estimation algorithm for IBM model 1.
# Then print, for either Swedish, German, or French, the 10 words that
#the English word european is most likely to be translated into, according
#to your estimate. It can be interesting to look at this list of 10 words and
#see how it changes during the EM iterations.

#reduce the size of corpus_de_en and corpus_sv_en
sv_en = open(en_sv_file_path, 'r', encoding='utf-8').read()
sv_en = sv_en.split("\n")
corpus_en = sv_en[:]

sv = open(sv_file_path, 'r', encoding='utf-8').read()
sv = sv.split("\n")
corpus_sv = sv[:]

# Estimate translation probabilities
# translation_prob_sv = ibm_model_1(corpus_en, corpus_sv, iterations=5)
```

```
[ ]: #reduce the size of corpus_de_en and corpus_sv_en
fr_en = open(en_fr_file_path, 'r', encoding='utf-8').read()
fr_en = fr_en.split("\n")
corpus_en = fr_en[:]

fr = open(fr_file_path, 'r', encoding='utf-8').read()
fr = fr.split("\n")
```

```
corpus_fr = fr[:]

# Estimate translation probabilities
translation_prob_fr = ibm_model_1(
    corpus_en, corpus_fr, iterations=3)
```

[ ]:
```
# Find translations for a specific word (e.g., "european")
translations_for_word = {pair[1]: prob for pair,
                         prob in translation_prob_fr.items() if pair[0] ==␣
 ↪"european"}
# Sort the top 10 translations by probability

suma = 0
for i in translations_for_word:
    suma = suma + translations_for_word[i]
french_translations = sorted(
    translations_for_word.items(), key=lambda item: item[1], reverse=True)[:10]

print("Top translations for 'european' in French:")
for foreign_word, prob in french_translations:
    print(f"{foreign_word}: {prob}")
```

test with french

[ ]:
```
# Find translations for a specific word (e.g., "european")
translations_for_word = {pair[1]: prob for pair,
                         prob in translation_prob.items() if pair[0] ==␣
 ↪"european"}
# Sort the top 10 translations by probability

suma = 0
for i in translations_for_word:
    suma = suma + translations_for_word[i]
swedish_translations = sorted(
    translations_for_word.items(), key=lambda item: item[1], reverse=True)[:10]

print("Top translations for 'european' in Swedish:")
for foreign_word, prob in swedish_translations:
    print(f"{foreign_word}: {prob}")
```

1.5 Decoding

[ ]:
```
import heapq


def get_top_n_word_translations(foreign_sentence, translation_prob, n):
    word_translations = {}
    for word in foreign_sentence:
        #get top 5 translations for each word
        translations_for_word = {
            pair[0]: prob for pair, prob
            in translation_prob.items() if pair[1] == word}

        # Sort translations by probability
        sorted_translations = sorted(
            translations_for_word.items(),
            key=lambda item: item[1], reverse=True)[:n]
        word_translations[word] = sorted_translations
        print(f'word list for {word}: {word_translations[word]}')
        print('\n')

    return word_translations

def translate_sentence_approx(sentence, translation_prob,
                              n_words=5, beam_width=5):
    """
    Translate one sentence from a foreign language to English.
    In the algorithm, we will keep the top n word translations for
    each word in the sentence. Moreover, we will use beam search to
    keep the most likely translations for the whole sentence.
    """

    beam = [(0, [])]  # (log_prob, sequence)
    top_n_word_translations = get_top_n_word_translations(
        sentence, translation_prob, n_words)

    # Iterate over each word in the foreign sentence
    for word in sentence:
        # Get the top translations for the current foreign word
        if word in top_n_word_translations:
            top_translations = top_n_word_translations[word]
        else:
            continue

        next_beam = []

        # Expand each sequence in the beam with
        # each translation of the current foreign word
        for log_prob, seq in beam:
            for (translation, translation_prob) in top_translations:
                new_seq = seq + [translation]

                # Update the log probability
                new_log_prob = log_prob + math.log(translation_prob)
                next_beam.append((new_log_prob, new_seq))

        # Keep only the top `beam_width` sequences
        beam = heapq.nlargest(beam_width, next_beam, key=lambda x: x[0])

    prob, highest_prob_sentence = (0, '') if len(beam) == 0 else max(beam,␣
  ↪key=lambda x: x[0])
    return prob, " ".join(highest_prob_sentence)


french_sentence = "je suis européenne".split()
prob_translated_sentennce, translated_sentence = translate_sentence_approx(
    sentence=french_sentence,  translation_prob=translation_prob_fr,
    n_words=5)
print(f"French sentence: {' '.join(french_sentence)}")
print(
    f"Translated sentence: {translated_sentence} (log probability:␣
  ↪{prob_translated_sentennce})")
```