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

---

[146]:
```
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

[147]:
```
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

[148]:
```
# Function to read a file and return word frequencies
def get_word_frequencies(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        text = file.read().lower()  # Ensure all text is lowercase
        words = re.findall(r'\b\w+\b', text)  # Extract words
        word_freq = Counter(words)  # Count word frequencies
```

```
    return word_freq
```

[149]:
```
# 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]))
```

```
Most common words in German:
die (10521 occurrences), der (9374 occurrences), und (7028 occurrences), in
(4175 occurrences), zu (3169 occurrences), den (2976 occurrences), wir (2863
occurrences), daß (2738 occurrences), ich (2670 occurrences), das (2669
occurrences)
Most common words in English (German-English pair):
the (19853 occurrences), of (9633 occurrences), to (9069 occurrences), and (7307
occurrences), in (6278 occurrences), is (4478 occurrences), that (4441
occurrences), a (4438 occurrences), we (3372 occurrences), this (3362
occurrences)
Most common words in French:
apos (16729 occurrences), de (14528 occurrences), la (9746 occurrences), et
(6620 occurrences), l (6536 occurrences), le (6177 occurrences), à (5588
occurrences), les (5587 occurrences), des (5232 occurrences), que (4797
occurrences)
Most common words in English (French-English pair):
the (19627 occurrences), of (9534 occurrences), to (8992 occurrences), and (7214
occurrences), in (6197 occurrences), is (4453 occurrences), that (4421
occurrences), a (4388 occurrences), we (3341 occurrences), this (3332
occurrences)
Most common words in Swedish:
att (9181 occurrences), och (7038 occurrences), i (5954 occurrences), det (5687
occurrences), som (5028 occurrences), för (4959 occurrences), av (4013
occurrences), är (3840 occurrences), en (3724 occurrences), vi (3211
occurrences)
Most common words in English (Swedish-English pair):
the (19327 occurrences), of (9344 occurrences), to (8814 occurrences), and (6949
occurrences), in (6124 occurrences), is (4400 occurrences), that (4357
occurrences), a (4271 occurrences), we (3223 occurrences), this (3222
occurrences)
```

[150]:
```
# 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)
```

```
Total words: 784458
Speaker count: 33
Zebra count: 0
Probability of 'speaker': 4.206726172720528e-05
Probability of 'zebra': 0.0
```

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")}')
```

```
Probability of "why are no-smoking areas not enforced ?": 7.157743716417319e-18
Probability of "the door is green": 0.0
Probability of "we pass": 2.0531400966183576e-05
```

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")}')
```

```
Log probability of "why are no-smoking areas not enforced ?": -57.44415862256026
Log probability of "the door is open": -29.796067530994655
Log probability of "the door is green": -31.741977680049967
Log probability of "we pass": -11.416178508839266
```

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)))
```

```
[['the', 'dog', 'runs'], ['the', 'cat', 'sleeps']]
{('the', '<NULL>'): 1e-05, ('the', 'le'): 0.199998, ('the', 'chien'): 0.199998,
('the', 'court'): 0.199998, ('the', 'chat'): 0.199998, ('the', 'dort'):
0.199998, ('dog', '<NULL>'): 1e-05, ('dog', 'le'): 0.33333, ('dog', 'chien'):
0.33333, ('dog', 'court'): 0.33333, ('runs', '<NULL>'): 1e-05, ('runs', 'le'):
0.33333, ('runs', 'chien'): 0.33333, ('runs', 'court'): 0.33333, ('cat',
'<NULL>'): 1e-05, ('cat', 'le'): 0.33333, ('cat', 'chat'): 0.33333, ('cat',
'dort'): 0.33333, ('sleeps', '<NULL>'): 1e-05, ('sleeps', 'le'): 0.33333,
('sleeps', 'chat'): 0.33333, ('sleeps', 'dort'): 0.33333, ('i', '<NULL>'):
```

```
1e-05, ('i', 'je'): 0.499995, ('i', 'suis'): 0.499995, ('am', '<NULL>'): 1e-05,
('am', 'je'): 0.499995, ('am', 'suis'): 0.499995}
```

[ ]:
```python
from collections import defaultdict

def ibm_model_1(corpus_english, corpus_foreign, iterations=10):
    # Assuming tokenize_corpus adds a "null" token to the beginning of each␣
 ↪English sentence
    # and splits sentences into lists of words.
    corpus_foreign_tokens = tokenize_corpus(corpus_foreign, add_null=True)  #␣
 ↪foreign language corpus
    corpus_english_tokens = tokenize_corpus(corpus_english)  # English corpus,␣
 ↪with null word

    # 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 each word in the english sentence
            for word_f in sentence_f:
                # Compute normalization factor for the word2
                s_total_word_e = sum(translation_prob[(word_e, word_f)] for␣
 ↪word_e in sentence_e)
                # For each word in the foreign sentence
                for word_e in sentence_e:
                    # Calculate delta, which is the proportion of the alignment␣
 ↪probability of the word2 to the word1
                    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]]

    return translation_prob
```

[ ]:
```
from collections import defaultdict


def ibm_model_1_optimized(corpus_english, corpus_foreign, iterations=10):
    corpus_foreign_tokens = tokenize_corpus(corpus_foreign, add_null=True)
    corpus_english_tokens = tokenize_corpus(corpus_english)

    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):
            # Optimization: Use local variables to reduce global lookups
            # Convert to a list once per sentence pair, if not already a list
            sentence_e_local = list(sentence_e)
            get_translation_prob = translation_prob.get  # Local function␣
↪reference

            # For each word in the foreign sentence
            for word_f in sentence_f:
                # Optimization: Cache computed probabilities and use list␣
↪comprehension for sum
                word_probs = [get_translation_prob(
                    (word_e, word_f), 0.0) for word_e in sentence_e_local]
                s_total_word_e = sum(word_probs)

                # Optimization: Use enumerate to iterate over both words and␣
↪cached probabilities
                for idx, word_e in enumerate(sentence_e_local):
                    delta = word_probs[idx] / s_total_word_e
                    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]

        # Optimization: Normalize probabilities more efficiently
        for word_e in total_e:
            normalization_factor = 0
            for sentence_f in corpus_foreign_tokens:
                for word_f in sentence_f:
                    normalization_factor += translation_prob.get(
                        (word_e, word_f), 0.0)
            for sentence_f in corpus_foreign_tokens:
                for word_f in sentence_f:
                    pair = (word_e, word_f)
                    if pair in translation_prob:  # Check to avoid creating␣
 ↪zero entries
                        translation_prob[pair] /= normalization_factor

    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_optimized(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)
```

```
{('the', '<NULL>'): 1e-05, ('the', 'das'): 0.33333, ('the', 'haus'): 0.33333,
```

```
('the', 'buch'): 0.33333, ('house', '<NULL>'): 1e-05, ('house', 'das'):
0.2499975, ('house', 'haus'): 0.2499975, ('house', 'ein'): 0.2499975, ('house',
'großes'): 0.2499975, ('book', '<NULL>'): 1e-05, ('book', 'das'): 0.499995,
('book', 'buch'): 0.499995, ('a', '<NULL>'): 1e-05, ('a', 'ein'): 0.33333, ('a',
'großes'): 0.33333, ('a', 'haus'): 0.33333, ('big', '<NULL>'): 1e-05, ('big',
'ein'): 0.33333, ('big', 'großes'): 0.33333, ('big', 'haus'): 0.33333}
Top translations for 'house':
haus: 0.13227183579011997
<NULL>: 0.02466562698421499
das: 5.373602007709076e-21
ein: 1.953105363653598e-30
großes: 1.953105363653598e-30
0.15693746277433496
```

Need to be very careful with the NULL probability. It needs to be way lower than the other probabilities, otherwise the model will always choose the NULL translation, because NULL will be present in every sentence. Increasing the number of iterations will eventually make null the most probable translation for every word.

test with swede

[ ]:
```python
#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)
```

[ ]:
```python
#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=30)
```

[160]:
```
# 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}")
```

```
Top translations for 'european' in French:
européenne: 0.4793358348474479
européen: 0.2891000250623562
apos: 0.09885042396056481
l: 0.0869579716619051
de: 0.029416858572761516
au: 0.005985469766092662
le: 0.0037956171957049777
<NULL>: 0.0024198789253493296
en: 0.000951969306135209
d: 0.0007054494484298159
```

test with french

[159]:
```
# 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}")
```

```
Top translations for 'european' in Swedish:
```

1.5 Decoding

[ ]:
```python
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})")
```

```
word list for je: [('i', 0.6624285821735717), ('rossa', 0.27582155226175936),
('am', 0.23761432290720727), ('like', 0.22266832182209578), ('please',
0.20905968207564107)]


word list for suis: [('am', 0.40911230518245667), ('galicia',
0.2430798650512864), ('contacted', 0.20017739757920394), ('abstained',
0.1884958628197376), ('campaigning', 0.1835540759633998)]


word list for européenne: [('european', 0.4793358348474479), ('reluctance',
0.17700219884502422), ('construction', 0.16067392674243278), ('periodical',
0.15517872071109737), ('harsh', 0.1506930590885051)]


French sentence: je suis européenne
Translated sentence: i am european (log probability: -2.0409619139049204)
```