Summer Semester 2020 SNLP Assignment 6

Name: Awantee Deshpande
Id: 2581348
Email: s8awdesh@stud.uni-saarland.de

Name: Lakshmi Rajendra Bashyam
Id: 2581455
Email: s8laraje@stud.uni-saarland.de

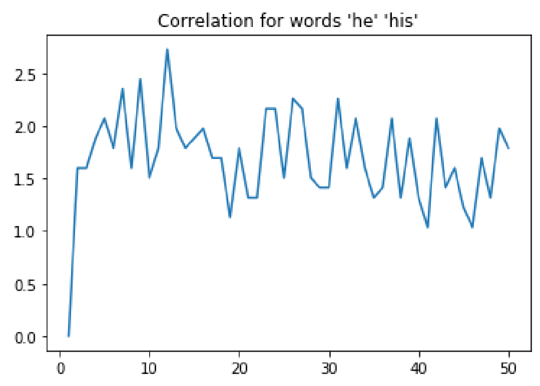
Long range dependencies

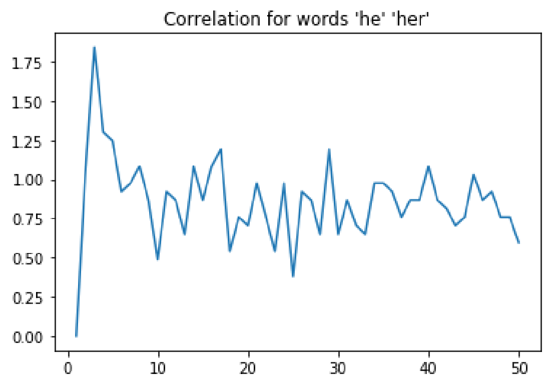
1) Linear interpolation

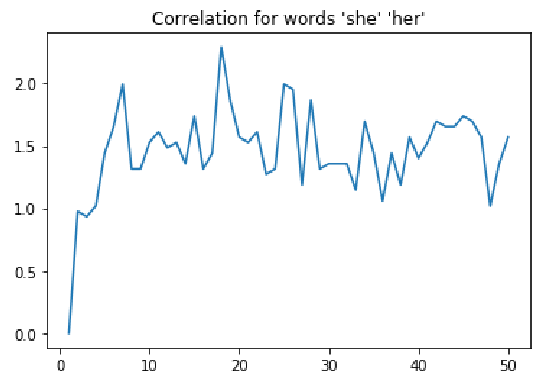
The correlation between words w1 and w2 separated by distance D is given by

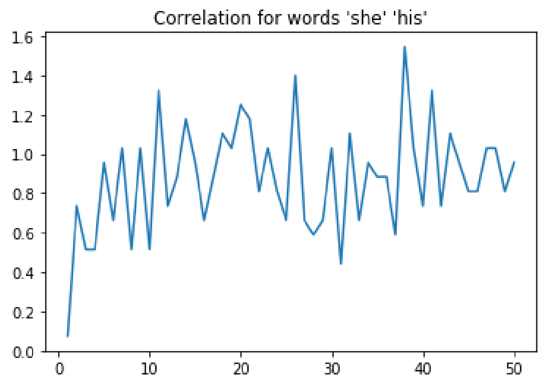
$$Corr_D(w_1, w_2) = \frac{P_D(w_1, w_2)}{P(w_1)P(w_2)}$$

After preprocessing the file, we run the correlation function for the given word pairs ("he","his"), ("he","her"), ("she","her"), ("she","his") with D varying from 1 to 50. We get the following plots:

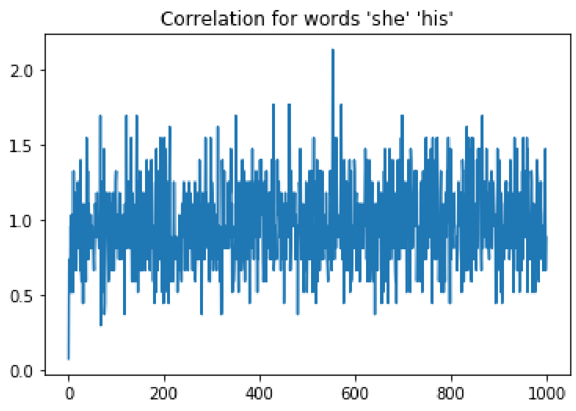






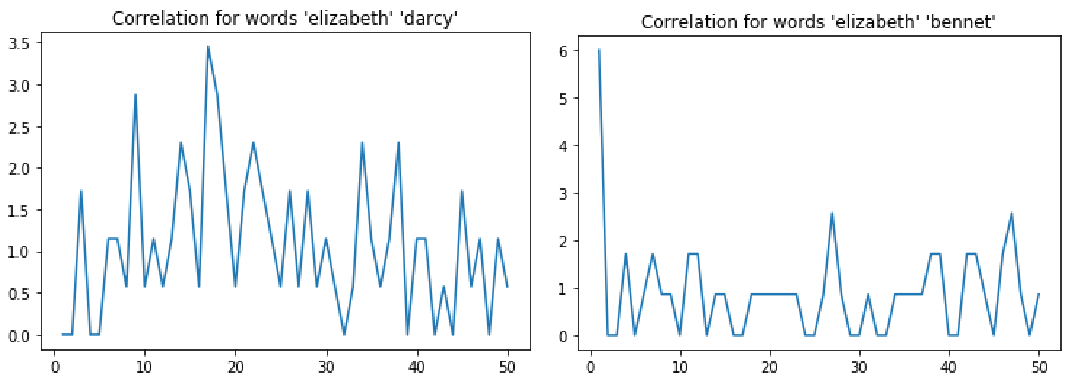


Analysis: Words like ‘he’, she, ‘his’, ‘her’ are very common pronouns used in the English language and not really (or necessarily) related to each other over a distance D. Hence, the correlation plots for these word pairs are very erratic. Just for the sake of comparison, we take the plot of (‘she’, ‘his’) over a distance D = 1000.



(‘she’, ‘his’, D=1000 : still very erratic)

At the same time, we take two (possibly) related words (‘elizabeth’, ‘darcy’) and (‘elizabeth’,’bennet’) over the corpus. We get the following plots.



(Much higher correlation values: Highest for ‘elizabeth’-’bennet’ at distance 1 as expected)

Thus, it makes sense that correlation of related words is high over shorter distances, and the dependency would ideally decrease as the distance becomes really high. This case doesn’t hold true for the above pronouns because they cannot be coreferenced to the previous one over a long distance D.

Language Models

1) Linear interpolation

Linear interpolation for bigrams is calculated using

$$P_I(w_2|w_1) = \lambda_1 P(w_2) + \lambda_2 P(w_2|w_1)$$

In our code, the function find_interpolated_prob finds the interpolated probabilities of bigrams. We calculate the corresponding bigram model with $\lambda_{1,2}$ = 0.5 and α = 0.3.

Bigram perplexity without interpolation = 839.486578185118

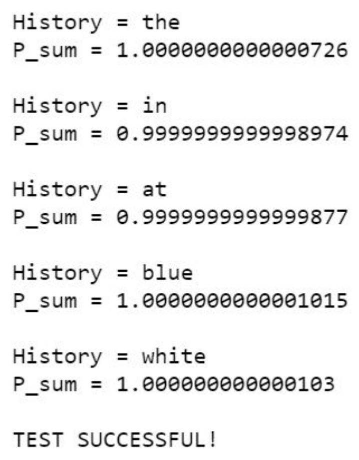
Bigram perplexity with interpolation = 419.2059754075171

Analysis: In simple linear interpolation, we combine different order n-grams by linearly interpolating all the models. We estimate the bigram probabilities using unigram and bigram probabilities weighted by a lambda term.

Because interpolation assigns higher weights to more context and better matches, higher weighted probabilities contribute more information. Hence, one can observe a significant decrease in perplexity.

2) Absolute discounting
The file AbsoluteDiscounting.py contains the code for this part of the exercise. The code is explained in the README file. Using the obtained language model, we get the following results:

a) Tests successfully passed!

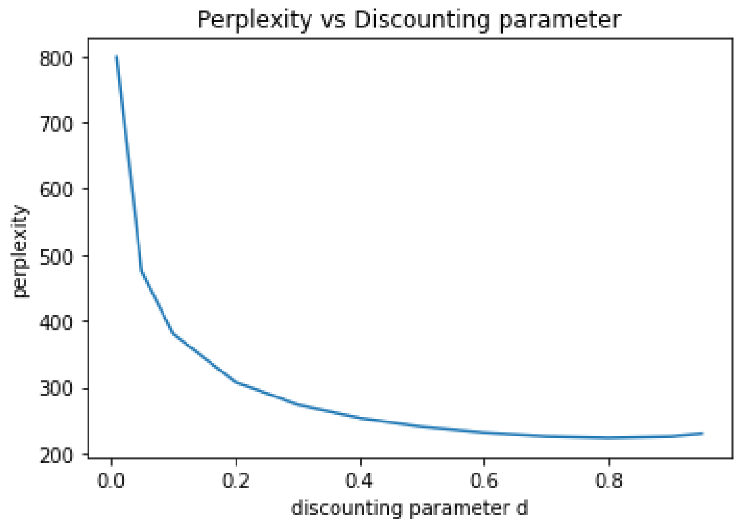


```
History = the
P_sum = 1.0000000000000726

History = in
P_sum = 0.9999999999998974

History = at
P_sum = 0.9999999999999877

History = blue
P_sum = 1.0000000000001015

History = white
P_sum = 1.000000000000103

TEST SUCCESSFUL!
```

b)
Bigram perplexity with discounting parameter d (0.7) = 225.61367810627084

c) For d in range (0.1, 0.9) we obtain the following results:
(Just to see the change we take some extra values beyond the range)
D = [0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95]
Perplexity values: [798.6057580555289, 474.9334192032063, 381.1715962975732, 308.0260557618603, 273.796822928229, 253.36470031424003, 240.00337909465202, 231.11271976255702, 225.61367810627084, 223.33858440160583, 225.4702135331811, 229.9155427867341]

Plot:



Analysis:
The obtained plot is as expected. We observe a dip in the perplexity values initially, and then they start increasing from d = 0.9.
The lowest perplexity is obtained at d = 0.8 (223.33858440160583).
This value is much lower than that of the Lidstone smoothed perplexity and Linear interpolation perplexities. This is because the model backs off to the unigram probabilities in case of unseen bigrams. For OOV unigrams, it simply backs of to a factor of the unigram probability over the entire vocabulary. It leaves the higher probability counts virtually intact.

3) Kneser-Ney Smoothing
The code for this is in the file Kneser_Ney.py. We obtain the following results after the implementation:

a)

	w = “longbourn”	w = “pleasure”
N(w)	71	71
$N_{1+}(\bullet w)$	13	28
$-\log_2 P_{LID}(w)$	10.4249215583	10.424921558
$-\log_2 P_{KN}(w)$	11.774185177	10.66726997

Analysis: The number of occurrences for both ‘longbourn’ and ‘pleasure’ is same. Lidstone smoothing assigns the same probability to both the words. But, the KN probability for ‘longbourn’ is less than that of ‘pleasure’. This is because ‘longbourn’ has a smaller context than ‘pleasure’. Hence, Kneser-Ney smoothing assigns a higher probability to the word with more context. That is, the negative logarithm (essentially the information content) is higher for ‘longbourn’ than it is for ‘pleasure’.

b)
The idea behind Kneser-Ney smoothing is based on discounted smoothing, with the extra characteristic of handling lower order unigrams differently. A standard unigram models assigns higher probabilities to unigrams with high frequencies. KN smoothing considers the *context* a unigram occurs in.

$$P_{KN}(w_3) = \frac{N_{1+}(\bullet w_3)}{\sum_{v \in V} N_{1+}(\bullet v)} = \frac{N_{1+}(\bullet w_3)}{N_{1+}(\bullet\bullet)}$$

If the unigram is not in the corpus, the probability backs of to the uniform distribution as usual. Thus, instead of just seeing how often a unigram u occurs in a corpus (which other backing off language models do), Kneser Ney smoothing will consider how often u occurs *in continuation* to some word.