Iswar Mahapatro
(2023201047)

IIIT HYD

Assignment 1 report
CS7.401
Introduction To NLP

Dataset and Preprocessing:-

The dataset consists of 2 corpuses :-Pride and Prejudice corpus (1,24,970 words) and Ulysses James Joyce(2,68,117 words).
(a) On “Pride and Prejudice” corpus:
i. LM 1: Tokenization + 3-gram LM + Good-Turing Smoothing
ii. LM 2: Tokenization + 3-gram LM + Linear Interpolation
(b) On “Ulysses” corpus:
i. LM 3: Tokenization + 3-gram LM + Good-Turing Smoothing
ii. LM 4: Tokenization + 3-gram LM + Linear Interpolation

Preprocessing steps considered:-

1. My Language model is on sentences. Several sentences are treated separately for training and testing purposes.

2. Any token during training is part of the model’s vocabulary including (*<BOS>* , *<EOS>*).

This functionality is implemented in *tokenizer.py*

read data: reads data from specified path and performs preprocessing to return list of list of sentences.

preprocess: tokenizes the text into sentences and adds sentence delimiters(tags).

N-Gram Language Models:-

These are simple models without any smoothing techniques and record counts for tokens seen after a particular context. To handle the start of a sentence multiple *<BOS> tags* are prep-ended (n-1 times) and an *<EOS>* tag is appended at the end of each sentence.. During training, any token seen is considered a part of the vocabulary hence counts for the unknown token are zero during training. These simple models cannot handle unseen tokens at inference time and assign zero probability (infinite perplexity) to such events.

An N-gram model computes the following:

$$P(w_n|w_{n-N+1} \ldots w_{n-1}) = \frac{C(wn-N+1 \ldots wn)}{C(w_{n-N+1} \ldots w_{n-1})}$$

These models generate sentences iteratively by sampling words from this distribution according to the given context (updated in each iteration).

Interpolation Smoothing:-

This model uses an ensemble of N-gram models for modeling the probability of a word given its context. For eg for a Trigram model interpolated with simpler bigram and unigram models the model is as follows

$$P\,(w_n|w_{n-2}w_{n-1}) = \lambda_1 \frac{C(w_{n-2}w_{n-1}w_n)}{C(w_{n-2}w_{n-1})} + \lambda_2 \frac{C(w_{n-1}w_n)}{C(w_{n-1})} + \lambda_3 \frac{C(w_n)}{C(\epsilon)}$$

This can be extended to a general n-gram model, where we interpolate n models n-gram, (n-1)-gram.We must ensure that the interpolation weights sum to 1 to ensure that the model returns probabilities.

To find these interpolation weights we do a Grid Search in the hyperparameter space using the validation set.

LM model	**Interpolation Weights(lambda1,lambda2,lambda3)**	**Test Perplexity**	**Train perplexity**
LM-2	(0.36684379338407774,0.417723841894458,0.21543236472146424)	**365.40**	362.78
LM-4	(0.46981661288890203,0.335280927148397,0.19490245996270092)	174652.36	148380.95

To prevent overflow conditions while calculating perplexity I am taking the maximum of actual probabilities of sentences and 1e-50 as lower than this probability doesn’t make any sense and its equivalent to 0 probability.The high perplexities in LM-4 is mostly due to high data sparsity.

This functionality is implemented in *language_model.py*

Simple Good Turing Smoothing:-

Let's say we have an N-gram Language model, where given a context with (n-1) tokens we want to estimate the count of a word. Let N_c be the number of words with count c (i.e. frequency of frequency).

Good Turing smoothing shifts probability mass from higher frequency items to lower frequency items (including those tokens with zero frequency) in the following manner:

For items with zero frequency the new estimate will be: $P_0 = N_1/N$, for items which had count c the new count will be $c^* = (c + 1)^{Nc+1/Nc}$.

Several steps are done before this basic smoothing:

N_c (not smooth for higher values of c) is smoothed out to Z_c, which averages many non zero N_c values with zero N_c values. First order the non zero N_r values by increasing value of r then for each non zero N_r we average with zero N_r's that surround it: Let q, r, t be successive indices of non-zero values, We replace N_r with $Z_r = {}^{2}{}_{t-}{}^{N}{}_{q}r$ (averaging with all z~~ero~~ values from q to t). Now we have Z_r, a smoothed out version of N_r. We have estimated N_r with the density of N_r.

We then fit a linear regression to the log-log plot between Z_r and r. This plot is then used for computing the smoothed counts. For small values of c, we N_c is as it is without smoothing. For larger values of c we read N_c from the log-log regression line. If there are too few frequencies for a given context that is there are too few N_c values for a given context then we resort to normal add-k smoothing for that context.

LM model	**Test Perplexity**	**Train perplexity**
LM-1	743.30	770.37
LM-3	**1203.006**	1265.59

To prevent overflow conditions while calculating perplexity, I am taking the maximum of actual probabilities of sentences and 1e-50 as lower than this probability doesn't make any sense and its equivalent to 0 probability.

This functionality is implemented in *language_model.py*

Best Model:-

The Best Model after all experiments for "Pride and Prejudice" corpus is **LM-2,** and for "Ulysses" corpus is **LM-3.**

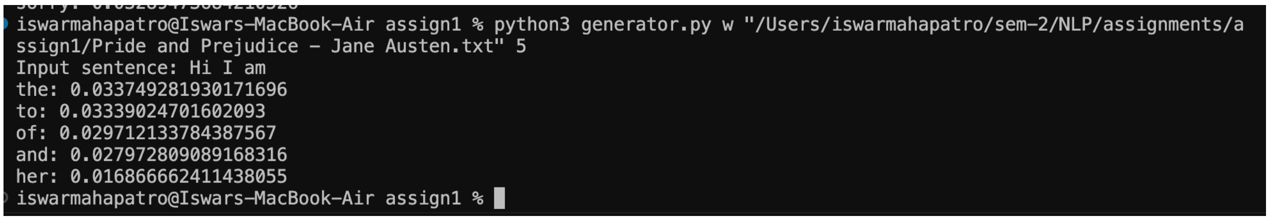
Generation:-

In generation there are 3 cases:-
Case 'w': generation without smoothing techniques for different values of N.
Case 'i': generation with Interpolation smoothing technique.
Case 'g': generation with Good-turing smoothing technique.

This functionality is implemented in *generator.py.*

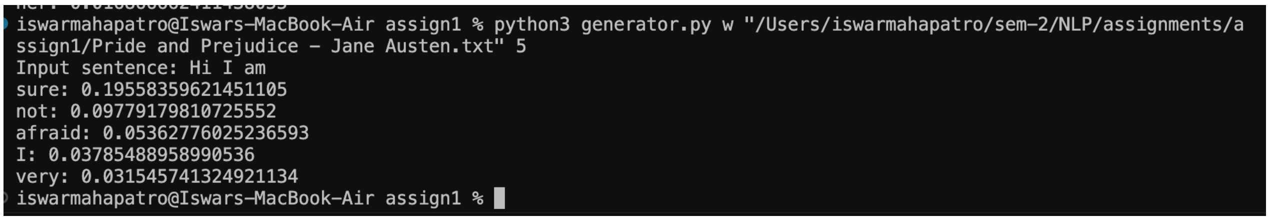
Case ‘w’:

Model(N=1):-



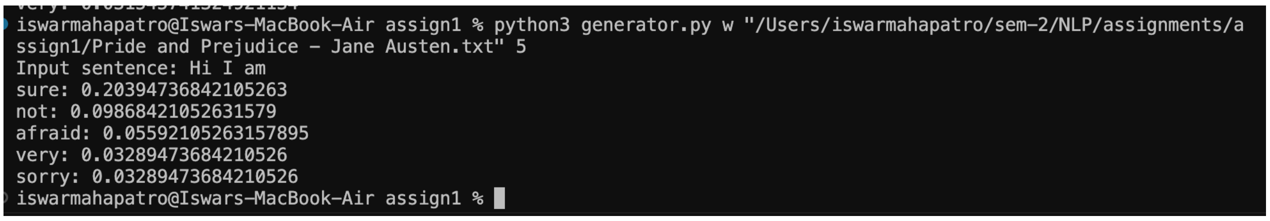
```
iswarmahapatro@Iswars-MacBook-Air assign1 % python3 generator.py w "/Users/iswarmahapatro/sem-2/NLP/assignments/a
ssign1/Pride and Prejudice - Jane Austen.txt" 5
Input sentence: Hi I am
the: 0.033749281930171696
to: 0.03339024701602093
of: 0.029712133784387567
and: 0.027972809089168316
her: 0.016866662411438055
iswarmahapatro@Iswars-MacBook-Air assign1 %
```

Model(N=2):-



```
iswarmahapatro@Iswars-MacBook-Air assign1 % python3 generator.py w "/Users/iswarmahapatro/sem-2/NLP/assignments/a
ssign1/Pride and Prejudice - Jane Austen.txt" 5
Input sentence: Hi I am
sure: 0.19558359621451105
not: 0.09779179810725552
afraid: 0.05362776025236593
I: 0.03785488958990536
very: 0.031545741324921134
iswarmahapatro@Iswars-MacBook-Air assign1 %
```

Model(N=3):-

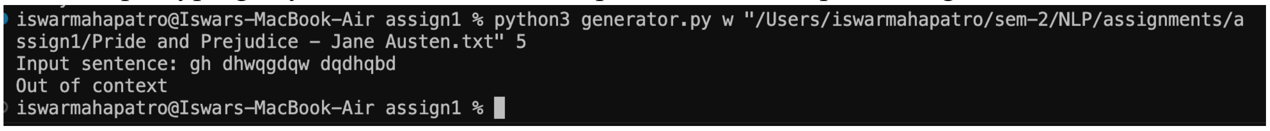


```
iswarmahapatro@Iswars-MacBook-Air assign1 % python3 generator.py w "/Users/iswarmahapatro/sem-2/NLP/assignments/a
ssign1/Pride and Prejudice - Jane Austen.txt" 5
Input sentence: Hi I am
sure: 0.20394736842105263
not: 0.09868421052631579
afraid: 0.05592105263157895
very: 0.03289473684210526
sorry: 0.03289473684210526
iswarmahapatro@Iswars-MacBook-Air assign1 %
```

As we can see, the N-gram model with N=3 performs better and predicts the next word more fluently in comparison to N=1,and N=2.

For out of context data sentence input given, then I am printing it as *out of context* since there is no use of predicting irrelevant words for which the sentence doesn’t make any sense.

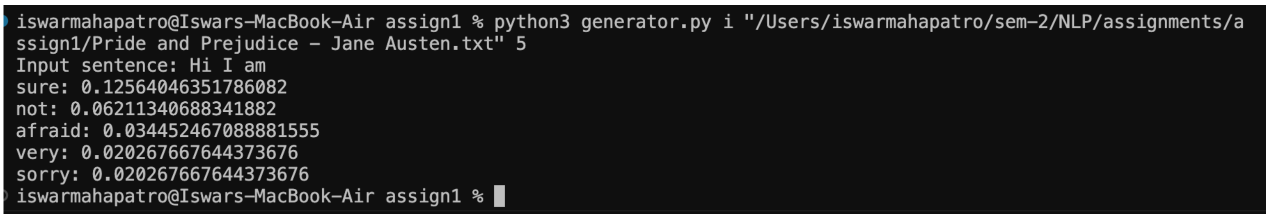
For example typing any random sentence as input like in below pic we’ll get out of context:-



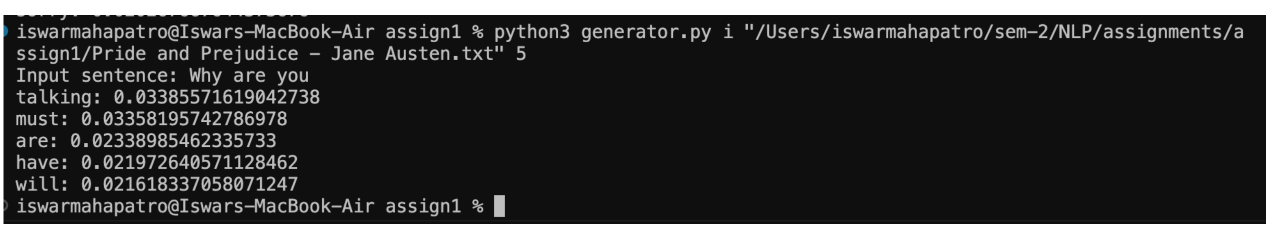
```
iswarmahapatro@Iswars-MacBook-Air assign1 % python3 generator.py w "/Users/iswarmahapatro/sem-2/NLP/assignments/a
ssign1/Pride and Prejudice - Jane Austen.txt" 5
Input sentence: gh dhwqgdqw dqdhqbd
Out of context
iswarmahapatro@Iswars-MacBook-Air assign1 %
```

NOTE:- I have handled similarly the same for ‘g’ and ‘i’ cases as well,i.e, if any random sentences given as input then it will show out of context and it will basically predict only relevant words.

Case ‘i’:

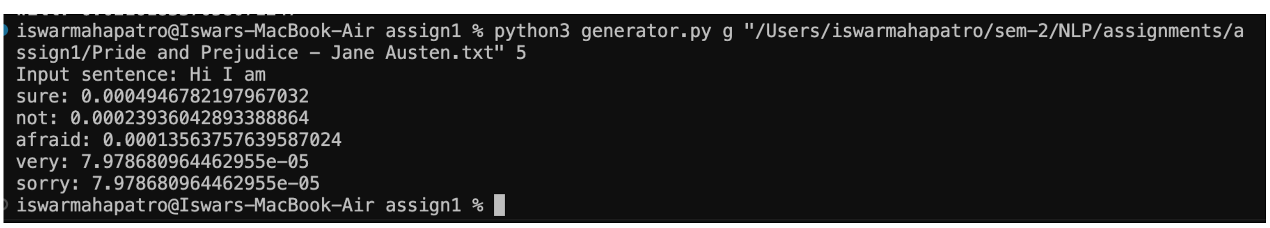


```
iswarmahapatro@Iswars-MacBook-Air assign1 % python3 generator.py i "/Users/iswarmahapatro/sem-2/NLP/assignments/a
ssign1/Pride and Prejudice - Jane Austen.txt" 5
Input sentence: Hi I am
sure: 0.12564046351786082
not: 0.06211340688341882
afraid: 0.034452467088881555
very: 0.020267667644373676
sorry: 0.020267667644373676
iswarmahapatro@Iswars-MacBook-Air assign1 %
```

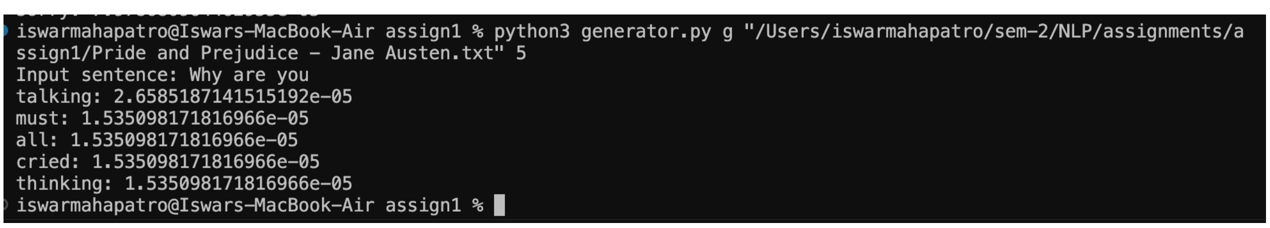


```
iswarmahapatro@Iswars-MacBook-Air assign1 % python3 generator.py i "/Users/iswarmahapatro/sem-2/NLP/assignments/a
ssign1/Pride and Prejudice - Jane Austen.txt" 5
Input sentence: Why are you
talking: 0.03385571619042738
must: 0.03358195742786978
are: 0.02338985462335733
have: 0.021972640571128462
will: 0.021618337058071247
iswarmahapatro@Iswars-MacBook-Air assign1 %
```

Case ‘g’:



```
iswarmahapatro@Iswars-MacBook-Air assign1 % python3 generator.py g "/Users/iswarmahapatro/sem-2/NLP/assignments/a
ssign1/Pride and Prejudice - Jane Austen.txt" 5
Input sentence: Hi I am
sure: 0.0004946782197967032
not: 0.00023936042893388864
afraid: 0.00013563757639587024
very: 7.978680964462955e-05
sorry: 7.978680964462955e-05
iswarmahapatro@Iswars-MacBook-Air assign1 %
```



```
iswarmahapatro@Iswars-MacBook-Air assign1 % python3 generator.py g "/Users/iswarmahapatro/sem-2/NLP/assignments/a
ssign1/Pride and Prejudice - Jane Austen.txt" 5
Input sentence: Why are you
talking: 2.6585187141515192e-05
must: 1.535098171816966e-05
all: 1.535098171816966e-05
cried: 1.535098171816966e-05
thinking: 1.535098171816966e-05
iswarmahapatro@Iswars-MacBook-Air assign1 %
```