Dataset and Preprocessing

The dataset consists of 7 Harry Potter Books. Books 1-5 are considered for training, Book 6 is used as the validation set for hyperparameter tuning and Book 7 is used as the unseen test set for reporting final results. Preprocessing steps considered:

1. Page Demarcations are removed, these are used in *txt* files to separate pages. E.g *Page — 2 Harry Potter and the Deathly Hallows - J.K. Rowling*
2. All sentences are lowercased.
3. New Line characters are removed which were used for indentation purposes and are replaced by spaces
4. Punctuations are removed.
5. I have used *word_tokenize* and *sent_tokenize* from the *Natural Language Tool Kit* to divide the text into sentences and sentences into corresponding tokens. Each sentence is then prepended with the start of sentence token ($\langle S \rangle$) and is appended with the end of sentence token ($\langle / S \rangle$).
6. My Language model is on sentences. Several sentences are treated separately for training and testing purposes.
7. Any token during training is part of the model's vocabulary including ($\langle S \rangle$, $\langle /S \rangle$ and $\langle UNK \rangle$). Any other token is out of the vocabulary for the model and hence treated as $\langle UNK \rangle$.

This functionality is implemented in *preprocess.py*
read_data: reads data from specified path and performs preprocessing to return text string
preprocess: tokenizes the text into sentences and adds sentence delimiters

N-Gram Language Models

These are simple models without any smoothing and record counts for tokens seen after a particular context. To handle the start of a sentence multiple $\langle S \rangle$ are prep-ended (n-1 times). 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:

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

These models generate sentences iteratively by sampling words from this distribution according to the given context (updated in each iteration).
Some samples generated by different N-gram models (restricted to certain number of tokens):

1. Unigram Language Model (n=1)
   (a) *when in ron onto woman at said the crashed*
   (b) *of and his face his goal never knickerbockers harry have*
   (c) *you been the they to otherwise comment the a and*

2. Bigram Language Model (n=2)
   (a) *wheres hagrid was not practiced disarming charm work he looked*
   (b) *honestly i dont like this was clicking noise in the*
   (c) *would attract far end of gratitude for release harry and*
3. Trigram Language Model (n=3)
   (a) *but somehow they still had his eyes narrowed dangerously again*
   (b) *his heart plummeted had she met that potter might resort*
   (c) *mum an dad were as undursleyish as it had flashed*
4. 4-Gram Language Model
   (a) *three two one with a roar of noise greeted them mainly cheers because ravenclaw and hufflepuff were anxious to see*
   (b) *its not just you its dumbledore too she believes the daily prophet kept letting off a highpitched whistle as steam*
   (c) *unicorn blood has strengthened me these past weeks*
5. 5-Gram Language Model
   (a) *harry inhaled the familiar smell and felt his spirits soar*
   (b) *letters from school said mr. weasley passing harry and ron identical envelopes of yellowish parchment addressed in green ink*
   (c) *you know where she comes from you must know to whom she is reporting*

As we can see more complex models generate more meaningful sentences because they use larger contexts to generate words. But they tend to overfit to the training data and copy sentences from the training set.
All these models assign infinite perplexity to the test and validation sets because of out-of-vocabulary words on which the model is not trained and assigns to them zero probability.
Sample Command To run Code: *python n_gram.py -n 2 –generate True –generate_cnt 3 –smoothing True*

Add-k Smoothing

To account for zero counts in the train set of words, not in vocabulary (unknown token) we use smoothing to shift probability mass from frequent tokens to less frequent tokens.

$$P(w_n|w_{n-N+1}\dots w_{n-1}) = \frac{C(w_{n-N+1}\dots w_n)+k}{C(w_{n-N+1}\dots w_{n-1})+k|V|}$$

k is a hyperparameter that is tuned using the validation set (linear search). Tuning curves for different N-gram models are as follows:

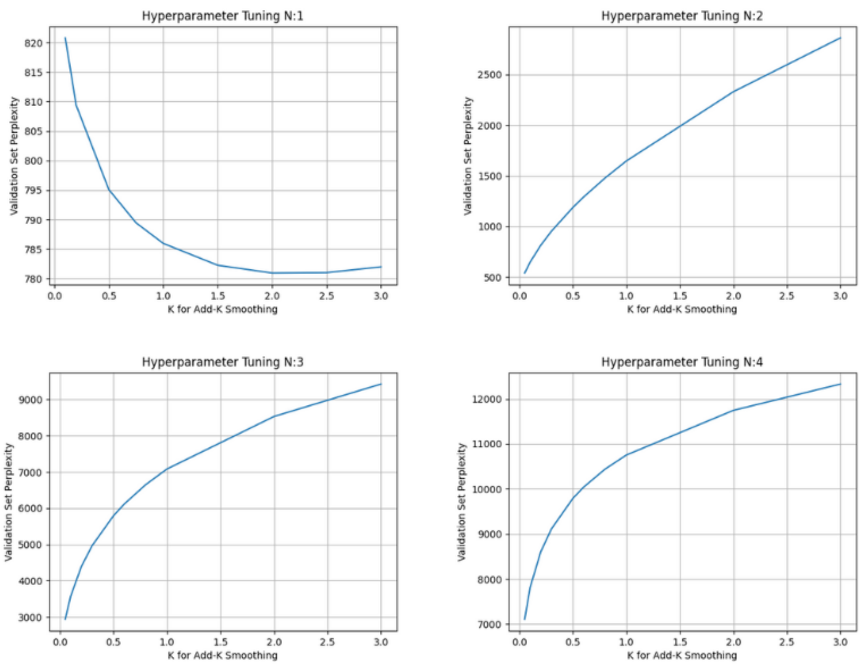

Abbildung 1: Hyperparameter Tuning for Add-k Smoothing

The results for the tuned models on the test set are as follows:

N	k from Valdiation	Test Perplexity
1	2	776.31
2	**0.05**	**594.51**
3	0.05	3174.54
4	0.05	7398.74

Sample Command To run Code:
python n_gram.py -n 2 –generate True –generate_cnt 3 –smoothing True –val True –plot True

Stupid Backoff

Stupid Backoff falls back to using shorter context to estimate probabilities for a given word when its count following larger context is zero. There is also a penalty multiplied when backing off to shorter contexts. Stupid Backoff scores are given according to the following:

$$S(w_i|w_{i-N+1}\dots w_{i-1}) \begin{cases} \frac{C(w_{i-N+1}\dots w_i)}{C(w_{i-N+1}\dots w_{i-1})}, & \text{if} C(w_{i-N+1}\dots w_i) > 0 \\ \alpha S(w_i|w_{i-N+2}\dots w_{i-1}), & \text{otherwise} \end{cases}$$

Note this model returns raw scores and not probabilities. I normalize these scores to ensure that they sum to one for any given context, i.e normalizing factor k is computed as follows:

$$\sum_{v\in V} kS(v|w_{i-N+1}\dots w_{i-1}) = 1$$

Computing this normalizing constant is particularly slow. In my analysis I have fixed α to be 0.4. I have also added add-k smoothing (k=0.05) for unigram scores.

N	Test Perplexity
1	831.83
2	**322.50**
3	366.05
4	630.72

Sample Command to run Code:
python backoff.py -n 3

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 ..... 1-gram. We must ensure that the interpolation weights sum to 1 to ensure that the model returns probabilities.

$$\sum_{i=1}^{N} \lambda_i = 1$$

To find these interpolation weights we do a Grid Search in the hyperparameter space using the validation set. We also use slight smoothing to handle out of vocabulary words in the validation/test set.

N	**Interpolation Weights from Validation (lower-order to higher order)**	**Test Perplexity**
1	(1.0,)	819.73
2	**(0.375, 0.625)**	**453.20**
3	(0.4, 0.5, 0.1)	467.20
4	(0.42, 0.42, 0.08, 0.08)	488.40

Sample Command to run Code:

python interpolation.py -n 2 –val True –smoothing True

Kneser Ney Smoothing

This method combines absolute discounting, backoff and continuation counts in one framework. Probabilies are estimated as follows:

$$P_{KN}(w_i|w_{i-N+1}\dots w_{i-1}) = \frac{max(c_{KN}(w_{i-N+1:i}) - d, 0)}{\sum_{v\in V} c_{KN}(w_{i-N+1:i-1}v)} + \lambda(w_{i-N+1:i-1})P_{KN}(w_i|w_{i-N+2:i-1})$$

Note this is a recursive formula. c_{KN} is defined as follows:

$$c_{KN}(.) \begin{cases} count(.), & \text{if we are estimating highest order n-gram} \\ continuation\ count(.) & \text{otherwise} \end{cases}$$

Where continuation count of a string is the number of unique single word contexts of that string. The Base case for the recursion is as follows:

$$P_{KN}(w) = \frac{c_{KN}(w)}{\sum_{v\in V} c_{KN}(v)} + \lambda(\epsilon)\frac{1}{|V|}$$

In all these cases λ has to be set to ensure that probabilities sum to one.

$$\sum_{v\in V} P_{KN}(v|w_{i-N+1:i-1}) = 1 \Rightarrow$$

$$\sum_{v\in V} \frac{max(c_{KN}(w_{i-N+1:i-1}v) - d, 0)}{\sum_{v\in V} c_{KN}(w_{i-N+1:i-1}v)} + \sum_{v\in V} \lambda(w_{i-N+1:i-1})P_{KN}(v|w_{i-N+2:i-1}) = 1 \Rightarrow$$

$$\sum_{v\in V} \frac{max(c_{KN}(w_{i-N+1:i-1}v) - d, 0)}{\sum_{v\in V} c_{KN}(w_{i-N+1:i-1}v)} + \lambda(w_{i-N+1:i-1}) = 1 \Rightarrow$$

$$\lambda(w_{i-N+1:i-1}) = \frac{d\ |\{w|c_{KN}(w_{i-N+1:i-1}w) > 0\}|}{\sum_{v\in V} c_{KN}(w_{i-N+1:i-1}v)}$$

This gives us all steps for the recursive implementations. I have used d=0.75 for all my models. The results for different N-gram models is as follows:

N	Test Perplexity
1	785.94
2	**276.29**
3	308.681
4	460.53

Sample Command to run Code:
python kneser-ney.py -n 2

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), thus the following:

$$\sum_{c=1}^{\infty} cN_c = N$$

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 $c^* = (c+1)\frac{N_{c+1}}{N_c}$.
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 $Z_r = \frac{2N_r}{t-q}$ (averaging with all zero 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.

N	Test Perplexity
1	656.01
2	**349.33**
3	3307.83

Sample Command to run Code:
python good-turing.py -n 1

Best Model

The Best Model amongst all experiments is the **Bigram Model (n=2) with Kneser Ney Smoothing.**