**Advanced Text Analytics Project**

**Part 1: Language Modeling**

1. **Write a code from the scratch that learns unigram and bigram models on the training data as Python dictionaries. Report the perplexity of your unigram and bigram models on the both training data and test data.**

We used brown corpus data set. There are around 57340 sentences in the data set. We have split the data into three parts: training, validation, and testing sets. The training data has

40138, validation data has 5734 and test data has 11466 sentences.

Code:

A screenshot of a computer

Description automatically generated

Preprocessing step: in this step we defined a function called “pre\_process\_data”. the function iterates through each sentence in the dataset and removes punctuation marks, empty strings, converts all words to lowercase and finally appends <s> at the beginning and ending of each sentence to ensure each sentence is clear.

Code:

A computer screen shot of a computer code

Description automatically generated

Vocab function: we defined a function called “vocab” to extract the unique words from the training dataset. In order to do this, we had to remove “<s>” from the words, convert the unique words into list and then add the “<s>” back to the list and calculate the vocabulary size.

Code:

A screenshot of a computer program

Description automatically generated

Calculate the frequency of the unique words: we defined a function called “unique\_word\_func” to find the frequency of the unique words.

Code:

A computer screen shot of a computer code

Description automatically generated

Unigram model: we build a unigram model based on total tokens and total word frequencies. As shown in the code below, the ‘unigram\_model\_prob’ function calculates the probabilities of individual words (unigrams) by dividing their frequency in the dataset, represented by “word\_freq”, by the total number of tokens in the dataset.

Code:

A screenshot of a computer program

Description automatically generated

Bigram model: we calculated bigram frequencies and probabilities within the dataset. First, the “bigram\_model\_freq” function processes a list of sentences, iterates through each sentence, and captures the frequencies of bigrams. These frequencies are stored in a dictionary called “bigram\_frequencies”.

The “bigram\_model \_prob” function calculates the probabilities of each bigram by considering their frequencies, unique word frequencies, and an optional denominator, allowing for add-lambda smoothing if specified. The resulting dictionary, “bigram\_probabilities”, contains the probability values associated with each observed bigram.

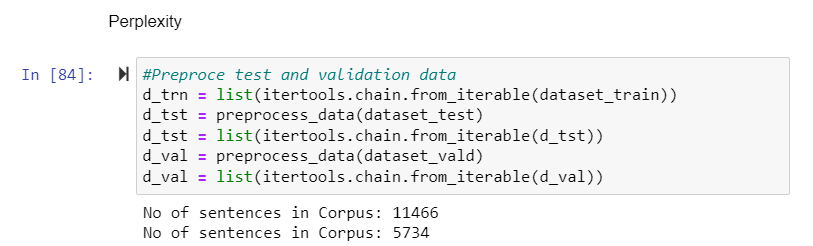
Codes:

A screenshot of a computer program

Description automatically generated

Perplexity calculation: in order to evaluate how good our model is performing we find the perplexity value. Before calculating the perplexity, we preprocess test and validation data.

Code:



We then defined a function called “perplexity” which takes two main inputs(ngrams, model)

Code:

A screenshot of a computer program

Description automatically generated

Then the perplexity values are computed for unigram model on the training, test and validation dataset

Code:

A screenshot of a computer program

Description automatically generated

And finally, we calculated the perplexity of bigram model for training, test and validation data.

Code:

A screenshot of a computer code

Description automatically generated

Unigram Perplexity on training set: 730.130991189557

Unigram Perplexity on test set: 377.645417328154

Bigram Perplexity on training set: 49.87132625761988

Bigram Perplexity on test set: 11.836979146604865

Conclusion : The unigram model appears to have high perplexity values. The bigram model shows significantly lower perplexity values compared to the unigram model, indicating that it performs better in terms of modeling language dependencies. therefore, Perplexity of upper n-gram models is lowest as they use the context. Lower the perplexity, better the model.

1. **Implement add-λ smoothing method. With varying λ values. Draw a curve that measures your perplexity change over different λ values on the developing data.**

We defined a function called “add\_smoothing” that serves the purpose of providing smoothed frequencies. It takes two main inputs: “l”, which represents the smoothing parameter, and “freq”, which is a dictionary containing frequencies of certain events or items. The result is a dictionary of smoothed frequencies, which is often used in statistical models to address issues like zero-frequency events and improve the robustness of probability estimations.

Code:

A screenshot of a computer program

Description automatically generated

Then we calculated perplexity for different lambda values. We used lambda values such as ( 0.001, 0.01, 0.1, 1, 1.5 ,2 ,5 , 10) using add-lambda smoothing for both unigram and bigram models. The ‘df’ table shown below illustrates the lambda values and perplexity values.

Code:

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

According to the ‘df’ table, perplexity values increase as lambda values increase. Therefore we decided to use lambda value of 0.001 for our models because lower perplexity indicates that the model is better. We then graphed the lambda and perplexity values for both unigram and bigram models.

A screenshot of a computer

Description automatically generated

A screen shot of a graph

Description automatically generated

**(c) Pick the best λ value(s) and train again your unigram and bigram models on training data + developing data. Report new perplexity of your unigram and bigram models on the test data.**

We used lambda value of 0.001. We first combined the training and validation data, preprocessing the combined data, and calculating the vocabulary size.

A screenshot of a computer code

Description automatically generated

Then we used “unique\_word\_func” function that we defined earlier to calculate the unigram and bigram frequencies of the new combined dataset.

Code:

A screenshot of a computer code

Description automatically generated

After calculating the frequencies of the new dataset, we implemented add-λ smoothing for both unigram and bigram models, calculating the smoothed probabilities, and then calculating the perplexity of the models on the validation dataset to evaluate the performance.

Code:

A screenshot of a computer program

Description automatically generated

1. **Generate random sentences based on the unigram and bigram language models from part (c). Report 5 sentences per model by sampling words from each model continuously until meeting the stop symbol ⟨/s⟩.**

First, we create a function called “context\_counter”to get the context and counter for ngram models.

A screenshot of a computer program

Description automatically generated

We then defined a function called “generate\_random\_token”, to randomly select a word based on the context derived from training data. It generates a random number 'r' and maps it to the probabilities associated with the potential words to be chosen. Depending on the value of 'n' the function selects the next word.

Code:

A screenshot of a computer program

Description automatically generated

The following code shows a function called “generate\_random\_text”, that produces random text using an n-gram model with a specified probability distribution. It initializes a context queue and an empty result list, then iterates to generate tokens based on the context and probability distribution until an end token is encountered. The generated text is returned as a string.

Code:

A screenshot of a computer program

Description automatically generated

After defining the function, 5 sentences was generated for unigram model

A screenshot of a computer

Description automatically generated

The following code generates random text using a Unigram model. The “next” function selects a word based on random probability from the provided dictionary, and the “generate\_sent\_uni” function builds a sentence by choosing words until an end or start token is encountered. Five generated sentence is returned as a string.

A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

(**e) Choose at least one additional extension to implement. The available options are trigram, Good-Turing smoothing, interpolation method, and creative handling of unknown words. Verify quantitative** **improvement by measuring 1) the perplexity on test data; and qualitative improvement by retrying 2) the random sentence generation in part (d).**

We have used the trigram model as an extension. For the trigram model, trigram frequencies are computed by iterating through a collection of sentences, counting the occurrences of each trigram in the dataset. This wasachieved using the “trigram\_model\_freq” function. The function takes a list of tokenized sentences as input and returns a dictionary where each trigram is a key, and its corresponding frequency represents how often that trigram appears in the dataset.

**A screenshot of a computer program

Description automatically generated**

Then, we utilize the “trigram\_model\_prob” function to compute the trigram probabilities. This function takes the trigram and bigram frequencies as inputs. By applying this function, we can obtain the probabilities for each trigram in the dataset. After defining the function, we calculated the perplexity of trigram model on training and test data in order to evaluate how good our model is.

A screenshot of a computer program

Description automatically generated

Then we performed add-lambda smoothing on trigram model and calculated the perplexity values

A screenshot of a computer program

Description automatically generated

In the following graph, we have visually represented the impact of different lambda values on the perplexity of our trigram model. Based on the graph lambda value of 0.001 has the lowest perplexity.

A graph on a computer screen

Description automatically generated

After identifying the best lambda value. We calculated the perplecity of tigram model using lambda smoothing.

A screenshot of a computer program

Description automatically generated

And finally, we generated five random texts using trigram model.

A screenshot of a computer program

Description automatically generated