L-Università ta' Malta

ICS2203

NLP: Language Model

B.Sc. in IT(Hons.) (Artificial Intelligence)

Zakkarija Micallef (0466500L)

Building the Language Model

Pre-processing

The British National English Corpus was chosen for this implementation of the Language Model. The corpus was made up of multiple XML files. Initially a random small subset of all the XML files were randomly selected to build the language model from.

The XML were read from using Java's DOM Parser [1]. Each file was either added to a Training String Arraylist or a Test Arraylist. At 25/75 split was chosen between the training and test set respectively. This split was chosen as it is the default in many python libraries [2]. Each Arraylist entry is a single word value, while "EOS" tokens were added between each sentence and "EOF" tokens were added at the end of every file. Theses ArrayLists were finally printed to separate text files.

In order to test the model, the computed language model would have to be loaded from the file's created. Instead of using txt files to store the N-grams and lexicon, the JSON file format was chosen as it was easier to read from and would not require as much string manipulation. On top of that, it also has the possibility of being used in a web application. To create and read from the JSON file format a Java plugin, JSON Simple was used [3].

Building N-gram Models

A N-gram object is made up 3 parts. The actual N-gram which is a String Array which hold the words such as [Hello, there, mate]. An integer "count" which stores the frequency of the N-gram and finally a double probability value.

The entire Training Arraylist is looped and every unique word is added to it, along with its frequency. Every word is passed onto a valid method which confirms that it is not blank or an "EOS" or "EOF" token.

The probabilities of the three Arraylists are then calculated using the Maximum Likelihood estimate.

Unigram Probability: Count (Unigram)/ Vocabulary Size

Bigram Probability: Count(Bigram)/Count(Unigram)

$$P(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

Trigram Probability: Count (Trigram) / Count (Bigram)

The Vanilla Arraylists are then cloned to the other variant ArrayLists.

Vanilla Version Arraylists: Unigrams, Bigrams, Trigrams

Laplace Version Arraylists: UnigramsLP, BigramsLP, TrigramsLP

UNK Version Arraylists: UnigramsUNK, BigramsUNK, TrigramsUNK

The Laplace Arraylists are then traversed and the count of each N-gram was incremented. The Probability was also recomputed using the Laplace Probability methods.

$$P(w_i|w_{i-1}) = \frac{count(w_{i-1}w_i)+1}{count(w_{i-1})+V}$$

For the UNK version, a new Training Arraylist had to be created, trainingUNK. It is a direct copy of the training ArrayList, but instead every word that only had a count of 1 in the vanilla unigram ArrayList is changed to a UNK Token. The Probabilities are then recalculated using the exact same method used for the Vanilla N-grams.

Linear Interpolation

There are further Arraylists dedicated to the linear interpolation of each of the 3 flavours. The trigram ArrayList of the given flavour li looped and each trigram's probability is stored. The probability of the bigrams is then searched with the first and a second word of the trigram. Same is done with the unigrams. The probabilities are finally multiplied by the given weights and added up to get the final probability for that trigram. This method is applied to all the flavours of N-grams.

Testing the Language Model

Text Generation

The generateSequence() method accepts 3 parameters. A String Arraylist which contains the initial text written by the user. The flavour & type chosen (e.g. Laplace bigram). The user is also requested to enter the number of words to generate. Initially the ArrayLists that will be used are set using the setngram() method.

If the text entered is only one word, and the trigram model is selected, a message pops up informing the user that either the unigram or bigram model will be used for the initial word generation.

The first word is taken and is passed onto a method (searchBigrams()) which will go through the Bigrams Arraylist and return another small ArrayList with all the bigrams that have an identical first word. If the arraylist is not empty than it is sent to a roulletteWheel() method which uses a roulette wheel possibility in order to choose the appropriate next word. If no bigrams with that word has been found, then it will choose a word using the rouletteWheel() method using the unigram model.

If the user has entered more than one word for example ("We are now in"), by the Markov Assumption, the method will only take into consideration the last two words. The last two word, in this case “Now” (word n-2) and “In” (word n-1), will be sent off to a (searchTrigrams()) which is exactly like the searchBigrams() method but for trigrams. If no trigrams are found it will then use the Bigram Model. If the user has chosen to use the bigram model in the beginning, it will initially start with the bigram model and won’t attempt to search for the trigrams. The same is applied to the unigram, if the user selects to only use the Laplace unigram, it will only attempt the unigram model.

Sentence Probability

This is done by using the chain Rule applied to the joint probability of words in a sentence.[4]

Initially, the selected N-gram flavour is selected and the user enters his sentence. The Sentence Probability is calculated using the same formulas used in the N-gram probability calculation. The sentence is traversed and each N-gram in the sentence is sent to a method which calculates its probability. The cumulative sum off all the probabilities of N-grams in the sentence is its probability.

The *N*-gram model assumes a generative model in which the next word generated depends only on the preceding n-1 words. Using Bayes' law, we get that the probability of a sentence is

$$P(w_1 \cdots w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1w_2)\cdots P(w_N|w_1 \ldots w_{N-1})P(w_{N+1}|w_2 \ldots w_N) \\ \cdots P(w_n|w_{n-N+1} \ldots w_{n-1}).$$

$$P(w_1 w_2 \ldots w_n) = \prod_i P(w_i \mid w_1 w_2 \ldots w_{i-1})$$

Future Improvements

A drawback of this implementation is that during the sequence/text generation, the program asks the user how many words to generate. If while building the language model, the N-grams took into account the EOS (end of sentence) tokens instead of ignoring them, the text prediction could keep on predicting words until it encounters an EOS token, which would signal the end of that particular sentence.

Folder Structure

All the XML files from the British Corpus are stored in a 1 file deep folder called “corpus”. All the JSON files used by the application are located in the JSON Folder. It has the training, test, training set with UNK tokens, three linear interpolation files for each of the flavours and 3 folders with their respective N-grams. The SampleModelOutput folder is identical to the JSON Folder but stores a txt version for easier referencing.

How to Run

First extract the zip folder. A run.bat file is included which will run the jar file in a command prompt. It is vital that the JSON folder is in the same directory as the JAR File since it has to load the data from there.

References

[1] L. Gupta, "Java Read XML - Java DOM Parser Example - HowToDoInJava", *HowToDoInJava*, 2020. [Online]. Available: https://howtodoinjava.com/xml/read-xml-dom-parser-example/.[Accessed: 22- Mar- 2020].

[2] "sklearn.model_selection.train_test_split — scikit-learn 0.22.2 documentation", *Scikit-learn.org*, 2020. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html. [Accessed: 22- Mar- 2020].

[3] *Code.google.com*, 2020. [Online]. Available: https://code.google.com/archive/p/json-simple/. [Accessed: 22- Mar- 2020].

[4] D. Jurafsky, *Web.stanford.edu*, 2020. [Online]. Available: https://web.stanford.edu/class/cs124/lec/languagemodeling.pdf. [Accessed: 22- Mar- 2020].