Jacob Hansen

Professor Kanemoto

CPSC-39

18 May 2023

# n-Gram Language Model

## Summary

For this final project I wrote an implementation of an n-Gram language model. This model is capable of taking a String as a prompt, and generating a continuation on that prompt for as many characters as the user desires. This model is both useful and entertaining. n-Gram models are used in the real world for tasks such as spell checking, autocompletion, language translation, and many other natural language processing tasks. However, the application that is perhaps most interesting is in computational biology, where n-Grams can be used to perform biological sequence analysis on DNA. This specific implementation is also entertaining because it comes up with very wild sentences that can be funny or interesting.

**n-Gram Process**

1. Load the dataset.
2. Get the unique **vocabulary** (*1*) and **ngrams** (*n+1*) as Sets by indexing over the dataset.
3. Generate **contexts** (*n*), and **targets** (*1*) in two parallel ArrayLists by slicing **ngrams**
4. Tokenize the targets into **targetTokens** so they can be **embedded** as an integer number representing their vocabulary index.
5. Generate the 2d array **embeddings** by summing targetTokens for each context index, and then normalize the sums as probabilities by dividing all elements by the sum of each row.
6. Take an **input** string of *n* letters and predict the next letter using the **embeddings** matrix as a look-up table. By getting the row of **embeddings** whose index is the same as the index of the **input** in **contexts**, we get an array of probabilities corresponding to the **vocabulary** indices. The index of that array of probabilities whose value is the most (max algorithm) will be equivalent to the vocabulary index of the predicted next character.
7. Append the generated character to the input string, remove the first character of the input string (maintain length *n*), then repeat step 6 with the new input string until the number of characters the user wants to generate is reached. This could be implemented recursively, I chose not to for the project for more clear code.

**Algorithms**

1. **Embedding Generation**

for each index **i** in **targetTokens**:

    **embeddings** [ **i** ] [ token at **i** in **targetTokens** ] += 1

for each **row** of index **i** in **embeddings**:

    for each **value** in **row**:

        **rowSums** [ **i** ] += **value**

for each **element** in **embeddings** at row **i**:

    **element** = **element** / **rowSums** [ **i** ]

This algorithm has Big O complexity of O(n). The input is a 2d array of doubles, which has *x* elements. Row and column wise, the number of elements *x* = *i* * *j*, where *i* is the number of rows, and *j* is the number of columns. We loop over all elements three times, which yields $\Theta(3n + k)$, simplifying to O(n).

It made the most sense to use a normal double array data structure to conserve memory, ensure speed, and because the size of **embeddings** is fixed. See the next page for screencap.

```java
// Sum into embeddings
double[][] embeddingSum = new double[size][vSize];
for (int i = 0; i < size; i++) {
          embeddingSum[i][targetTokens.get(i)] += 1;
}
double[][] embeddings = likelihood(embeddingSum);
// Calculates the likelihood of each matrix index based on row sums, index/sum
   public static double[][] likelihood(double[][] a) {
       double[][] ret = new double[a.length][a[0].length];
       double[] rowSums = new double[a.length];

       // Calculate row sums
       int i = 0;
       for (double[] row : a) {
           for (double d : row) {
               rowSums[i] += d;
           }
           i++;
       }

       // Calculate likelihood
       for (i = 0; i < a.length; i++) {
           for (int j = 0; j < a[0].length; j++) {
               // [i][j] becomes the likelihood based on the row sum
               ret[i][j] = a[i][j] / rowSums[i];
           }
       }

       return ret;
   }
```

2. **Parsing nGrams from Dataset**

let **ngrams** be a Set of Strings

for **i** in ( **dataset** length - n ):

    add **dataset** substring( **i**, **i** + **n**) to **ngrams**

For this algorithm, it made the most sense to use the Set data structure to retain the uniqueness of each element. When the program generates the **embedding** matrix, it is of size **vocabulary** * **ngrams**. Only storing unique **ngrams** saves memory and also allows the lookup to operate smoothly and efficiently because there are no duplicate rows in the embeddings table.

This algorithm has time complexity O(n), because we loop over every character in the input String **dataset**. See the next page for screencaps.

```java
// Returns a set of ngrams of size n from a string
   public static Set<String> getNGrams(String s, int n) {
       Set<String> ngrams = new HashSet<>();
       for (int i = 0; i < s.length() - n + 1; i++) {
           ngrams.add(s.substring(i, i + n));
       }

       return ngrams;
   }
public static void main(String[] args) {
       // Only get unique ngrams
       Set<String> ngrams = new HashSet<>();
       Set<String> vocabSet = new HashSet<>();
       List<String> vocab = new ArrayList<>();
       List<String> contexts;
       List<String> targets;
       String text =
loadStringFromFile("src/main/java/com/jackrabbit/data/OpenWebText/xsmall.txt");
       text = text.replaceAll("\n", " "); // clean up newlines
       int n = 7;

       // Get our vocabulary from the input text
       // Reuse our ngram function to get unique characters
       vocabSet = getNGrams(text, 1);
       for (String s : vocabSet) {
           vocab.add(s);
       }
       int vSize = vocab.size();

       // Get our ngrams from the input text
       // Split into contexts and targets
       // Targets will be the last character ngrams
       // Context is all preceeding characters
       ngrams = getNGrams(text, n + 1);
       int size = ngrams.size();
       contexts = splitNGrams(ngrams, n, "contexts");
       targets = splitNGrams(ngrams, n, "targets");
       List<Integer> targetTokens = new ArrayList<>();
       for (String s : targets) {
           targetTokens.add(vocab.indexOf(s));
       }
// main() continues……
```

3. **Prediction**

```
let max = 0
let probabilities = embeddings [ index of input in contexts ]    // an array holding the
for p in probabilities:                                          // probabilities for the next
        if p > max:                                              // character
                max = p
                output = character whose probability is p

return output
```

This algorithm executes a simple max algorithm like we have talked about in class. The intricacy comes from getting the array **probabilities** from the **input** string. Then the **output** is the character in the **vocabulary**, whose index is the same as the index in the **probability** array.

In the worst case, the algorithm has to examine every element of **probabilities** before finding the maximum, so its Big O complexity is O(n).

On the next page, another almost identical implementation uses a pseudo-random variable called **bias** to inject some randomness into the character prediction, intending to randomize the character prediction just a little bit.

```java
// Makes a prediction with the embedding matrix
   // Naive prediction using max probability
   public static String naivePredict(String in, double[][] embed, List<String> ctxt,
List<String> vocab) {
       String ret = "";
       if (!ctxt.contains(in)) {
           System.out.println("Unknown input string");
           return ret;
       }

       // Naive prediction using max probability
       double max = 0;
       double[] probabilities = embed[ctxt.indexOf(in)];
       // Find the max
       for (int i = 0; i < probabilities.length; i++) {
           if (probabilities[i] > max) {
               max = probabilities[i];
               ret = vocab.get(i);
           }
       }
       return ret;
   }
// Predicts based on a random threshold
   // Threshold is a random number between 0 and 1, biased by the bias parameter
   public static String predict(String in, double[][] embed, List<String> ctxt,
List<String> vocab, double bias) {
       if (!ctxt.contains(in)) {
           System.out.println("Unknown input string");
           return "";
       }
       double threshold = biasedRandom(bias);
       double accumulator = 0.0;
       double[] probabilities = embed[ctxt.indexOf(in)];
       for (int i = 0; i < probabilities.length; i++) {
           accumulator += probabilities[i];
           if (threshold < accumulator) {
               return vocab.get(i);
           }
       }
       // In case of rounding error
       return vocab.get(probabilities.length - 1);
   }
```

**Future Changes and Problems**

The main issue with this implementation is that the algorithm is making predictions on a letter-by-letter basis. In order to make coherent sentences, the value of *n* (how many letters to look at to predict the next one) needs to be very high. Like the letter permutation problem we looked at in class, as the length of our permutations grows, the amount of permutations increases exponentially. Likewise, for this algorithm, as *n* grows, the number of **contexts** grows exponentially. This problem is made even worse by the fact that the **embeddings** table is of size **contexts.size()** * **vocabulary.size()**, so we are multiplying an exponentially increasing function by a very large number, making the problem much bigger.

I found that for my implementation, selecting any value of *n* greater than 7 would cause a memory overflow. To improve performance, I reduced the size of the dataset by about an order of magnitude which solved the overflow. Even so, the demo in my video presentation uses *n*=7, a dataset with ~6,000 lines of text, and still when I examined the amount of memory the JVM was using, it was around 10Gb.

A better fix for this problem, and thus what I would do if I had a do-over, is to use a more nuanced tokenization technique. My implementation tokenized each letter simply as it's ASCII code, but there are much better alternatives. Using an already existing algorithm such as Byte Pair Encoding, or Word2Vec, the n-Gram algorithm would be looking at full words, or even sentences to predict the next word or sentence. This would drastically reduce the value of *n* necessary to produce coherent sentences. Not only would the context be full words, but the vocabulary would be full words as well. For the same cost, a good tokenization algorithm could help encode a much better representation of language into the embedding matrix.