CS 5740 Project 1: Language Modeling

Philip Su (ps845) and Kelvin Jin (kkj9)

INTRODUCTION

For the first project, a collection of n-gram-based language models were implemented. Throughout the entire project, Philip and Kelvin split the work equally, often peer programming at times.

PART 1 - UNSMOOTHED N-GRAMS

Data Structures

For part 1, several data structures were implemented to compute unsmoothed unigrams and bigrams for an arbitrary text corpus. The data structures are introduced below.

Single token strings were stored in a **Token** object inunigrams. Beyond containing the string value of the token, this object also had a frequency field that denoted how many times the token was seen in the corpus, as well as another, internal field to assist in random word generation.

```
public static class Token {
    public String value;
    public int frequency;
    public int cumulativeFrequency;

    /**
     * Constructs a new instance of the Token class.
     *
     * @param value               The string value associated with this token.
     * @param frequency           The frequency at which this token appears in the corpus.
     * @param cumulativeFrequency The sum of all token frequencies to the left of this token
     *                            in the corpus's list of tokens.
     */
    public Token(String value, int frequency, int cumulativeFrequency) {
        this.value = value;
        this.frequency = frequency;
        this.cumulativeFrequency = cumulativeFrequency;
    }
}
```

Figure 1: Token Class

The **UnigramModel** object, shown in Figure 2, contains 2 data structures: a list of tokens *tokenFrequencies* and a map *tokenIndexLookupTable* which allows O(1) lookup into the list of tokens.

```
/**
 * A collection of tokens.
 * <p>
 * This class is implemented by maintaining a list of tokens, as well as a mapping from a token
 * string to its index in the list.
 */
class UnigramModel extends NgramModel {
    private List<Token> tokenFrequencies = new ArrayList<>();
    private Map<String, Integer> tokenIndexLookupTable = new HashMap<>();
```

Figure 2: Unigram Class

To represent bigrams, we created **BigramModel** objects (Figure 3), which simply contained maps from String objects to UnigramModel objects. The idea behind this was that the set of tokens that followed some token *k* could be represented the same way a unigram model was represented.

```
class BigramModel extends NgramModel {
    Map<String, UnigramModel> bigramTokenCollectionMap = new HashMap<>();
```

Figure 3: ***Initial Bigram Model, later replaced with generalized N-gram model***

Finally, the **Corpus** object represents the entire corpus trained from data. The corpus was created by providing the path to the directory containing the books. By having internal unigram and bigram fields, the corpus has methods that calculate the probability of an n-gram as well as methods to generate random sentences.

Methodology

After examining several libraries for extracting tokens from a corpus (such as NLTK), it was decided to use a custom token extraction method utilizing I/O functions and regular expressions for finer control over the token extraction process. Specifically, Strings were split on any regular expression not matching: [a-zA-Z0-0' -]. Words with hyphens and apostrophes were not split to prevent contraction and hyphenated words to be erroneously separated (e.g. "don't" would have resulted in "don" and "t"). The process involved iterating through the entire corpus, and string every token as unigrams as well as every pair of tokens as bigrams.

PART 2 - RANDOM SENTENCE GENERATION

```
public String getWord(final TokenList previousTokens, final double p) {
    double prob = p;
    if (totalCount == 0 || prob < 0.0 || prob >= 1.0) {
        return "";
    }
    // Multiply p to scale it to the range [0.0, totalCount).
    prob *= totalCount;
    // Utilize binary search to find the word associated with the range in which p falls.
    // This range is determined by the cumulative frequency field in each token object.
    int left = 0;
    int right = tokenFrequencies.size() - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        Token element = tokenFrequencies.get(mid);
        if (prob >= element.cumulativeFrequency && prob < element.cumulativeFrequency + element.frequency) {
            return element.value;
        } else if (prob < element.cumulativeFrequency) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    // Should never hit this point
    assert(false);
    return "";
}
```

Figure 4: Getting a random word involves binary search

In order to generate random sentences based on a corpus set, the function *getWord()* was implemented within the Corpus object. The method took 2 parameters: the length of the sentence to generate and the N-gram model to use to generate the sentence. The implementation is shown in Figure 4. Each word in a unigram was given a range between 0 and 1, and a random word was generated by taking a random number and looking up the word with that range. Some examples of 20-word sentences from different genres, generated by both the Unigram and Bigram model, are shown below:

Model	Genre	Sentence
Unigram	children	the. he first their i scrambled looking, pull re-established knocking, the at be he over corn colonies
Unigram	crime	of kept had by came collection himself to forests", window, got the www you, sad was
Unigram	history	in qui the hundred sertorius the to immense his," queen italy a will on. it, it
Bigram	children	, long already said to dig into the best bone, to whistle sounded, and nine o'clock; there
Bigram	crime	had brought something else he was subsiding in which flat and found how sensitive instrument upon us!" from
Bigram	history	of public speeches of this time her throne of mead such enormous number of christian countries and on wheels up

Table 1: Random sentences generates by Unigram and Bigram models

PART 3 - SMOOTHING AND UNKNOWN WORDS

In the second half of the project, smoothing methods were added. Smoothing was necessary for several reasons, one of which is that it allowed for a better estimator than the naive, un-smoothed probabilities for rare events (in this case, rare tokens in the training set, but not necessarily rare in a test set). Without smoothing, the predicted probability of a word sequence in a test set or the most likely next word given previous words resulted in zero-valued probabilities, which posed a problem when calculating perplexity (taking the negative log of zero resulted in infinity). Good-Turing smoothing alleviated the problem by moving probability mass from previously seen events to the probability of rare events.

The Good-Turing smoothing implementation is shown in Figure 5. Tokens with frequencies up to a certain cutoff were smoothed (tokens with frequencies less than 5). Because the smoothing algorithm required knowledge of the number of tokens that appeared at a given frequency as well as the number of tokens that appear at a 1 higher frequency. a *frequencyCountMap* was implemented that stored mapping of frequency to counts, allowing for efficient O(1) calculation.

```java
/**
 * Returns the frequency with which a list of tokens appears in the corpus, with
 * the given smoothing option.
 * @param tokens The tokens to find.
 * @return The frequency of the given list of tokens.
 */
public double getTokenFrequency(final TokenList tokens, final SmoothOptions smoothOptions) {
    int tokenFrequency = getUnsmoothedTokenFrequency(tokens);
    if (tokenFrequency >= smoothOptions.getCutoff()) {
        // Use unsmoothed
        return tokenFrequency;
    } else {
        // Use smoothed
        if (frequencyCountMap.get(tokenFrequency + 1) == null) {
            return tokenFrequency;
        }
        int frequencyCount = frequencyCountMap.get(tokenFrequency);
        int oneLargerFrequencyCount = frequencyCountMap.get(tokenFrequency + 1);
        return (tokenFrequency + 1) * (oneLargerFrequencyCount / (double)frequencyCount);
    }
}
```

Figure 5: ***Good-Turing smoothing calculation***

The token frequency was adjusted with the *SmoothOption* parameter. The value that was returned for a token with frequency f was: (f + 1) * (count of tokens with frequency f+1 / count of tokens with frequency f).

This number was then divided by the total count to obtain the probability.

Additional measures were taken to use a trained corpus on a test corpus. More often than not, there were a number of words from the test set that the model had not seen before. The naive solution would be to use a probability of 0, but this led to skewed results, such as the probability being 0 for a word sequence containing an unknown word, or the predicted next word in an n-gram model given the previous n-1 words cannot be the unseen word.

During training, the Corpus assumed a fixed vocabulary (e.g. all words seen at least 5 times). Any other tokens were replaced with a default unknown token *<unk>*. When the Corpus encountered an unseen word during test, the corpus treated the unseen word with the probability of the *<unk>* token. Additionally, the implementation allows for a flexible vocabulary because the rare word frequency is easily tunable. This methodology allowed randomness in the model.

PART 4 - PERPLEXITY

Perplexity is a metric used to evaluate the effectiveness of the model. The definition of perplexity used was the logarithmic version to avoid underflow.

$$\exp \frac{1}{N} \sum_{i}^{N} -\log P(w_i | w_{i-1}, \ldots, w_{i-n+1})$$

The perplexity calculation is shown in Figure 6. The method first created a test corpus with the method *createCorpusWithTestSet().* The method first parsed the test corpus with the method *createCorpusWithTestSet()* in which unknown words are absorbed into the corpus, with probabilities and counts updated. The “unknown” n-gram is then generated (this is for arbitrary N-gram models used in our extension). Then, every n-gram was extracted from the test

corpus where the *anomaly* score, the negative log of the probability of each token (in the case of unigrams) or each pair of tokens (in the case of bigrams) in the training corpus, was multiplied by the number of times it appeared in the test corpus. The sum of all the values are added together, divided by the total count of tokens, and finally the natural exponential is raised to this value to determine *perplexity*.

```
public double calculatePerplexityFromModel(final Corpus testCorpus, final int n) throws IOException {
    if (testCorpus != null) {
        double sum = 0.0;
        // Run through test set to find unknown tokens
        final NgramModel trainNgramModel = this.ngramModels.get(n);
        final NgramModel testNgramModel = testCorpus.ngramModels.get(n);
        final Iterator<TokenList> iterator = testNgramModel.getIterator();
        while (iterator.hasNext()) {
            final TokenList tokenList = iterator.next();
            // Compute the contribution to the sum
            int numOccurrences = testNgramModel.getUnsmoothedTokenFrequency(tokenList);
            double anomalyScore = Utils.getAnomalyScore(trainNgramModel.getProbability(tokenList, SmoothOptions.DEFAULT));
            sum += numOccurrences * anomalyScore;
        }
        sum /= testNgramModel.getTotalCount();
        sum = Math.exp(sum);
        return sum;
    }
    throw new IOException("I/O issue");
}
```

Figure 6: Perplexity calculation

Lower perplexity values indicated that the training set was a tighter fit to the test data and the corpus predicted the sequence of tokens in the test data with higher probability. Perplexity values are shown in Table 2, calculated on different unknown thresholds - the number of counts a token must have to be in the closed vocabulary of the corpus. Increasing the unknown threshold resulted in a smaller vocabulary which resulted in lower perplexity values. This is because the number of *<unk>* tokens increased, allowing for more "randomness" and the language model was less perplexed with unseen words.

Genre of Train	Genre of Test	Perplexity (threshold = 1)	Perplexity (threshold = 3)	Perplexity (threshold = 5)
children	children	100918.55	57451.58	42760.23
children	**crime**	**60908.20**	**37307.01**	**27327.14**
children	history	81032.89	45313.50	31124.26
crime	children	74190.66	40075.76	28951.84
crime	**crime**	**42728.81**	**26657.90**	**20456.87**
crime	history	66140.64	34587.29	23399.56
history	children	79591.32	28428.42	16517.39
history	crime	63183.06	24562.77	14105.20
history	**history**	**55351.94**	**23014.72**	**14032.10**

Table 2: Perplexity values on different genres with different unknown thresholds

PART 5 - GENRE CLASSIFICATION

A simple genre classification method was implemented. Given a test corpus, the perplexity was calculated with every genre in the training set, and the genre that resulted in the lowest perplexity was used to label the test corpus. As shown in Table 2, crime and history books were classified correctly, but children books were misclassified.

There may be different reasons for the misclassification. First of all, the number of tokens for each genre in the training set was not uniform. Additionally, there was a a significant amount of noise in the data. Possible improvements include the following:

- Larger data sets - in any language modeling application, more data can always improve results.
- Exclude stopwords - Common words such as “the”, “is”, “he/she” are usually not genre specific and should be ignored when creating a language modelling system for genre recognition.
- External features - Using non-textual features such as when the book was written, who the book was written by, what company published the book may give insight to the genre of the book.

PART 6 - EXTENSION: GENERAL N-GRAM MODELS

In this part, the concept behind the relationship between unigrams and bigrams was recursively generalized to all n-grams. With the UnigramModel as our base case, n-grams were represented as maps of strings to (n-1)-gram models.

The experiments done in the previous parts of the project were performed again with the generic n-gram model. The random sentences for trigrams and 4-grams are shown below:

Model	Genre	Sentence
Trigram	children	cornell a voice of happy boys and girls[illustration: he after now? that'll be a nobleman and
Trigram	crime	cried guerchard."" yes; i make out that attempted escape and give you the story to the
Trigram	history	and of a single motive, the queen's service at balmoral, and thus both the project gutenberg literary archive
4-gram	children	felt that even he, old and weary, on one occasion the combat raged, the men, their
4-gram	crime	and put the book on the night of thursday, august 7th."" here?"" the
4-gram	history	and at malaca(_ malaga_), the rights of the provinces; quitting their government without leave

Table 3: Random sentences for various genres generated with Trigrams and 4-grams **sentences**

The sentences seem more sensible as the value of N increases, but at a certain point the became actual excerpts from the training corpus.

The perplexity values using the higher trigram and 4-gram models are shown below:

Genre of Train	Genre of Test	Trigram Model	4-Gram Model (10^8)
children	children	7584839.15	3.09
children	**crime**	**3092396.59**	**1.36**
children	history	3916636.59	1.162
crime	children	3918935.79	1.53
crime	**crime**	**2110354.05**	**0.87**
crime	history	2754496.83	1.04
history	children	2833950.41	0.99
history	Crime	1790033.82	0.83
history	**history**	**1682076.43**	**0.76**

Table 3: Perplexity values on different genres using with a threshold of 5 for trigram and 4-gram models

The results show that the perplexity increases as N increases because the corpus looks for N consecutive tokens that match its training data.