ICS2203 - Assignment
Task 1
Language Modelling and Basic Spell Checking

Daniel Magro
484497M

December 2017

Contents

Project Task 1 **2**
1.1 Code Explanation - BuildModel Class 2
1.2 Code Explanation - Main Class 7
1.3 How to Run the Program 10
1.4 Source Code Listing - BuildModel Class 11
1.5 Source Code Listing - Main Class 17

Project Task 1

1.1 Code Explanation - BuildModel Class

The unigram, bigram and trigram are stored as ArrayLists of **'Ngram'** objects. 'ArrayLists' are defined in Java as arrays of 'infinite' size, which grow or shrink as needed (implemented as lists)
The Ngram class contains

- An Array of Strings named **n_gram** (of size n=1, 2 or 3; depending on whether it is a unigram, bigram or trigram respectively). which stores the words making up the n-gram.
- An Integer named **count** which stores the number of times the n-gram has appeared inside the corpus.
- A Double named **smoothedProbability** storing the Probability of that n-gram, with Laplace Smoothing.

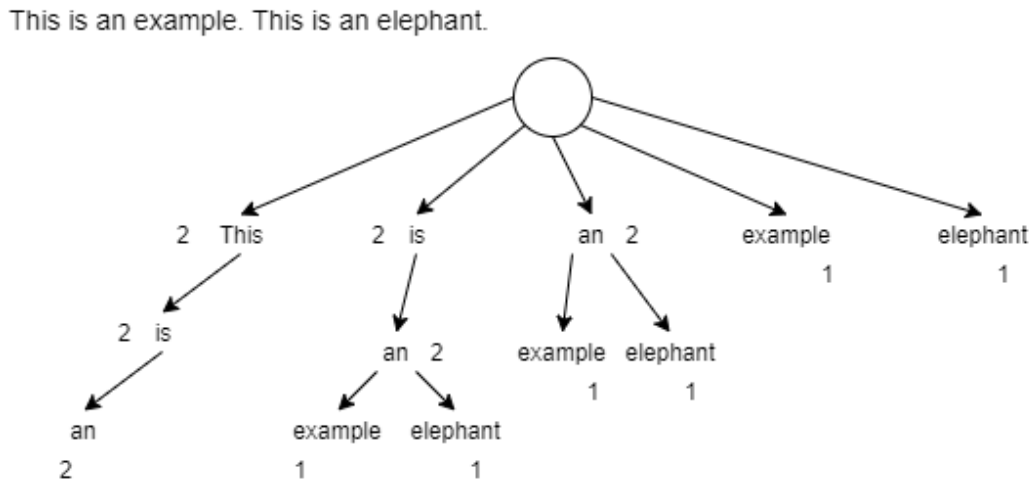
A more efficient way of storing the n-gram data would have been using a **trie**. With a larger corpus it is a much more memory efficient and more intuitive way of storing the n-grams. For example, the sentences "This is an example. This is an elephant." are represented using both methods on the next page.

Given the relatively small size of the corpus, the n-grams were stored as explained above, in ArrayLists since it is a simple yet effective enough implementation.

Table 1.1: The sentences "This is an example. This is an elephant." stored as ArrayLists of Type N-gram

n-gram[0]	n-gam[1]	n-gram[2]	Count
this			2
is			2
an			2
example			1
elephant			1
this	is		2
is	an		2
an	example		1
an	elephant		1
this	is	an	2
is	an	example	1
is	an	elephant	1

Figure 1.1: The sentences "This is an example. This is an elephant" stored as a Trie



When launched, the program starts executing the contents of the **main** method. First it creates 3 ArrayLists of type Ngram, one to store the **unigram** model, another to store the **bigram** model and a third to store the **trigram** model, and they are named as such.
Next, the **generateModels** method is called, and the 3 ArrayLists, unigram bigram and trigram, are passed as arguments, so that they can be referred to and filled from the generateModels method.

The **generateModels** method, as the name implies, generates the Vanilla Language Models for the given corpus.
First the directory containing all the text files that make up the corpus is indicated to the program.
Now the method loops through all the text files that make up the corpus, and does the following to it:
An array of Strings **"textFile"** is created which will store all the lines of the current text file being processed. The array textFile is then filled by the **readInput** method, which takes the path of a text file as a parameter, and returns each line of the text file (part of the corpus) as an element of the textFile array (whereby each line of the text file is stored as an element, of type string, inside the textFile array).
The method then loops through each element of the textFile array (starting from i=3, since the first 3 lines are not text but meta information, "<" and ">" tags), and does the following to it:
First 3 Strings are declared; **word_i**, **word_ip1** and **word_ip2**, which will store the current word, next word, and the word after that, respectively.
Next we check if the first letter of the string currently being considered is a "<", and if so, the loop moves onto the next line. If not, then the current line contains a word, and thus: the current array element is split up by tabulations into a new string array, the first element of that array is considered, converted to lower case, and stored inside **word_i**.
Once the current word is converted to lower case and stored in word_i, it is checked whether that word is already included inside our unigram model, using the **checkExistence** method, which takes the unigram model and the current word (as a single element array) as parameters. (The checkExistence method will be explained below)
If it is found that **word_i** is already in the unigram, its count is incremented by one. If not (if it is a new word that does not already exist in our model): First an object of type Ngram is created/declared, named **ug** with n=1 (which represents one element of the unigram model), then the first element of the Array of Strings (of size 1) is set to **word_i** and the count is set to **1**. Finally the object ug is added to the ArrayList unigram.

Next, the following line of the text file (textFile[i+1]) is processed very similarly to the current line. First it is checked whether the first character is a <, if it is, textFile[i+1] does not contain a word, and thus the construction of the bigram and trigram are skipped, and the next iteration is tried. If not, the current line is split by tabulations, the word is converted to lower case and stored in **word_ip1**. Then, the checkExistence method is called, this time with the bigram model and an array of Strings containing word_i and word_ip1 as parameters. If the bigram already exists in the language model, its count is simply incremented, if not, an object of type Ngram is created, named **bg** with n=2, its Array of Strings is set to [word_i, word_ip1], its count is set to 1 and added to the ArrayList of bigrams.

Finally, the line 2 steps ahead of the current in the text file (textFile[i+2]) is processed as explained before; First it is checked whether the first character is a <, and if so the construction of the trigram is skipped (since it does not contain a word), and the next iteration is tried. If not, the current line is split by tabulations, the word is converted to lower case and stored in **word_ip2**. Then, the checkExistence method is called, this time with the trigram model and an array of Strings containing word_i, word_ip1 and word_ip2 as parameters. If the trigram already exists in the language model, its count is simply incremented, if not it is created and its count is set to 1 and added to the ArrayList of trigrams.

The above process is repeated for all the lines inside the text file, and all the text files inside the corpus

When the generateModels method is done and has created the unigram, bigram and trigram each with a count of occurrences, their smoothed probability needs to be calculated. For this, 3 methods were created, **smoothUnigram**, **smoothBigram** and **smoothTrigram**

The **smoothUnigram** method takes the unigram model as a parameter. This method calculates the Laplace Smoothed probability of each unigram using the equation: $P(w_i) = \frac{count(w_i)+1}{N+V}$. where N is the total number of words in the corpus (sum of all counts of all unigrams) and V is the total number of unique words/unigrams in the corpus.
The smoothUnigram method first loops through all the unigrams to calculate the total count of all unigrams (N).
Then, vocabSize (V) is set to the total amount of unique unigrams in the model (unigram.size()).

Another loop iterates through all the unigrams, calculating their respective probability using the formula stated above.

The **smoothBigram** method takes the unigram and bigram models as parameters. This method calculates the Laplace Smoothed probability of each bigram using the equation: $P(w_i|w_{i-1}) = \frac{count(w_{i-1}w_i)+1}{count(w_{i-1})+V}$. where V is the total number of unique words/unigrams in the corpus.
The smoothBigram method first sets vocabSize (V) to the total amount of unique unigrams in the model (unigram.size()).
Another loop then iterates through all the bigrams. First a nested loop goes through the unigram model to find the count of w_{n-1}. Once that value is found, the smoothed probability of the bigram can be calculated using the formula stated above.

The **smoothTrigram** method takes the unigram, bigram and trigram models as parameters. This method calculates the Laplace Smoothed probability of each trigram using the equation: $P(w_i|w_{i-2}w_{i-1}) = \frac{count(w_{i-2}w_{i-1}w_i)+1}{count(w_{i-2}w_{i-1})+V}$. where V is the total number of unique words/unigrams in the corpus.
The smoothTrigram method first sets vocabSize (V) to the total amount of unique unigrams in the model (unigram.size()).
Another loop then iterates through all the trigrams. First a nested loop goes through the bigram model to find the count of $w_{n-2}w_{i-1}$. Once that value is found, the smoothed probability of the trigram can be calculated using the formula stated above.

Once all of the smoothed probabilities have been calculated, the **saveModels** method is called, which simply saves the unigram, bigram and trigram as objects, so that they can be loaded and used when required (for example from the Main class).

The **checkExistince** method takes an Ngram model and an array of Strings as parameters. This method checks through each element of the given model to find if the array of strings already exists within it. If so the global variable 'index' is set to 'i' and the method returns 'true'. If not, the method returns false.

1.2 Code Explanation - Main Class

The Main class will be the interface between the user and the language model. Through this class the user can choose which function, generate text or wrong word, he would like to run, and with which input.

Firstly, the program creates 3 global constants, ArrayLists of type Ngram, which will store the unigram, bigram and trigram.
A string named **word_i** is also created which serves as a global variable for the result of the generateText method.
An integer named **wwi** was created which will store the index of the word in the array chosen to be incorrectly spelt.

The **main method** first loads the unigram, bigram and trigram from disk into the their respective global variables, using the **loadModels** method.
Then, a simple menu is printed which asks the user to select either the **Generate Text** or the **Wrong Word** function.
If number **1 (Generate Text)** is chosen, the user is asked to input a sentence. This sentence is then converted into lower case, split into individual words by whitespaces and stored inside the **text** string array. Then the **generateText** method is called and the **text** array is passed as an argument.
If number **2 (Wrong Word)** is chosen, as in option 1, the user is asked to input a sentence, which is converted into lower case, split into individual words by whitespaces and stored inside the **text** string array. Then the **wrongWord** method is called and the **text** array is passed as an argument.
If number **3 (Terminate Program)** is chosen, the program terminates.

The **generateText** method receives the array of words inputted by the user as an argument.
Note: this method is making use of the **Markov Assumption**, since every time it is given a sentence of any length, it only considers the last 2 words of that sentence.
First, the methods sets the strings **word_im2** and **word_im1** to the last 2 words of the sentence (last 2 elements inside the text array). A new ArrayList of Ngrams is declared and called **trigramMatches**. trigramMatches is the set to the return of the **findTrigramMatches** method.
The **findTrigramMatches** method takes word_im2 and word_im1 as arguments, and looks through all the trigrams in the language model to find all the trigrams that start with word_im2 and word_im1 (i.e. are of the form

[word_im2, word_im1, x]) and returns all the trigrams that match as an ArrayList of Ngrams to the generateText method.
If the trigramMatches array is not of size 0, i.e. matches were found, the method **probabilisticChoice** is called and the ArrayList of matches is passed as an argument. *Note:* The program backs off from the trigram model to the bigram model or bigram model to the unigram model only when **no (0)** matches are found, alternatively, back off can be programmed to take place when say less than 3 matches are found, or when the total probability of all matches found is less than a certain cut-off point.

The **probabilisticChoice** method first calculates the sum of laplace smoothed probabilities of the matching n-grams and stores it in totalProbability. Then it divides the probability of each individual n-gram by the total probability (so as to make them out of 1). After that, a random real number between 0 and 1 is generated, and an ngram is chosen based on its cumulative probability. The index of that n-gram is returned as the method's output.
Finally, **word_i** is set to the third string (ngram[2]) of the i^{th} trigram match.

If the ArrayList of trigramMatches would have been empty (i.e. no trigram matches were found), the same procedure would have been followed, however the program backs off and uses bigrams, and of course only the last word in the sentence, word_im1, and instead the method **findBigramMatches** is used, which is almost identical to the **findTrigramMatches** however is applied to bigrams rather than trigrams.
Again, if both the trigram and bigram models returned no matches, then the program backs off to the unigram model instead.
If the user only entered 1 word, then the same procedure would take place, however starting from the bigram, and if need be backing off to the unigram.
If no input is entered, the program notifies the user that no input was given and starts from the main menu again.

The **wrongWord** method also receives the array of words inputted by the user as an argument.
First, an array of doubles named **word_probability** and an array of integers **word_match_count** are declared, of the same size as the length of the input sentence. word_probability stores the probability of each word, and word_match_count stores the number of times each word was matched with an n-gram (This count is stored to remove any 'unfair' bias the words on the edges of the sentence might have, since they will be matched less often, eg. once in the trigram, whereas words in the middle will get matched 3 times

by the trigram).
First the input sentence is matched by trigrams, meaning that every 3 consecutive words are matched to a trigram from the language model. The matched trigram's probability is then added to each word's (all 3) current probability.
The same process is repeated for every 2 consecutive words, which are matched to bigrams, and the probability of the bigram is added to the 2 matched words.
The process is finally repeated for unigrams, where every word is matched with a unigram from the language model and the unigram's probability is added to the word's probability.
After the matching process is done, the word_match_count array is initialised, based upon the length of the sentence and the position of each word in the sentence.
Each element of the word_probability array is then divided by its respective count in the word_match_count array.
Finally the word with the lowest probability is found and its index is set to the global variable **wwi**. Then, text[wwi] is printed as the most probably misspelt word.

1.3 How to Run the Program

1) Place the corpus you would like to build a language model for in the folder **C:\NLP\Full MLRS Corpus**.
2) Run the BuildModel class so that the language models are built and stored in **C:\NLP\Language Models**. Alternatively, you can copy the 3 files found inside the **Language Models Outputs** folder and place them in **C:\NLP\Language Models**
3) Run the Main class, which loads the language models from disk.
4) The program outputs a basic menu, choose 1 to use the **Generate Text** function, 2 for the **Wrong Word** function and 3 to terminate the program.
5) After choosing options 1 or 2, please type in your input sentence and press enter.

1.4 Source Code Listing - BuildModel Class

```
1  import java.io.*;
2  import java.nio.file.Files;
3  import java.nio.file.Paths;
4  import java.util.ArrayList;
5  import java.util.Arrays;
6
7  public class BuildModel {
8
9      public static void main(String[] args) throws IOException
       {
10         /* Declaring the Data Structures which will store the
       N-Grams.
11          * Each N-gram is stored as an Arraylist of 'Ngram'
       objects. The 'Ngram' object contains an array of strings
12          * (size 1 for unigram, 2 for bigram, 3 for trigram)
       and an integer storing count, i.e. the number of times it
       occurs
13          * inside the corpus. It also contains a field which
       stores the smoothed probability.
14          */
15         ArrayList<Ngram> unigram = new ArrayList<>();
16         ArrayList<Ngram> bigram = new ArrayList<>();
17         ArrayList<Ngram> trigram = new ArrayList<>();
18
19         // Constructing the unigram, bigram and trigram for
       the given corpus and storing in 'unigram', 'bigram' and '
       trigram'
20         System.out.println("Generating Models...");
21         generateModels(unigram, bigram, trigram);
22
23         // Performing Laplace Smoothing on each of the n-gram
       models
24         smoothUnigram(unigram);
25         System.out.println("Unigram Laplace Smoothed
       Probability has been calculated");
26         smoothBigram(unigram, bigram);
27         System.out.println("Bigram Laplace Smoothed
       Probability has been calculated");
28         smoothTrigram(unigram, bigram, trigram);
29         System.out.println("Trigram Laplace Smoothed
       Probability has been calculated");
30
31         // saving unigram, bigram and trigram as objects
32         saveModels(unigram, bigram, trigram);
33         System.out.println("Models have been saved as objects
       to disk");
34     }
```

```
35
36
37      // File IO method, retrieves all the lines in the text
    file and stores each line
38      // as an element in an array of Strings
39      private static String[] readInput(String inputFile)
    throws IOException {
40          String[] data = Files.readAllLines(Paths.get(
    inputFile)).toArray(new String[]{});
41          return data;
42      }
43
44      // index is a global variable that will store in which
    element an already exisitng N-gram's count is stored (used
     by checkExistence method)
45      private static int index;
46      // this variable tracks the progress of the model
    generation (files completed)
47      private static int file_count = 0;
48
49      private static void generateModels(ArrayList<Ngram>
    unigram, ArrayList<Ngram> bigram, ArrayList<Ngram> trigram
    ) throws IOException {
50          File dir = new File("C:\\NLP\\Full MLRS Corpus");
51          File[] directoryListing = dir.listFiles();
52
53          // iterating through all the text files that make up
    the corpus
54          for (File child : directoryListing) {
55              String textFile[] = readInput(child.
    getAbsolutePath());
56
57              // starting from i=3 since first 3 lines are not
    part of the corpus
58              for (int i = 3; i < textFile.length; i++) {
59
60                  String word_i, word_ip1, word_ip2;
61
62                  // checking if the first symbol is a '<', and
     if so, skipping that line
63                  if (textFile[i].charAt(0) == '<'){
64                      continue;
65                  } else {
66                      // the current word (word_i) is the first
     word of the current line of the text file
67                      // First the current line is considered (
    textFile[i])
68                      // Then the line is split by tabulations
    "\t" and stored in an array ( .split("\t") )
```

```
69                  // Next the first element of that array
    is considered ( [0] )
70                  // Finally the first word is converted
    into lowercase and stored inside word_i ( .toLowerCase() )
     in lowercase
71                  word_i = textFile[i].split("\t")[0].
    toLowerCase();
72
73                  // if the word is already in our model,
    we simply increment its count
74                  if (checkExistence(unigram, new String
    []{word_i})){
75                      unigram.get(index).count++;
76                  } else {    // if it is not already in
    our model, it is created
77                      Ngram ug = new Ngram(1);
78                      ug.n_gram[0] = word_i;
79                      ug.count = 1;
80                      unigram.add(ug);
81                  }
82              }
83
84              if (textFile[i+1].charAt(0) == '<'){
85                  // do nothing (no words follow the
    current so a bigram or trigram cannot be constructed)
86                  continue;
87              } else {
88                  // Same as word_i but textFile[i+1] is
    used to consider the word on the next line
89                  word_ip1 = textFile[i + 1].split("\t")
    [0].toLowerCase();
90
91                  if (checkExistence(bigram, new String []{
    word_i, word_ip1})){
92                      bigram.get(index).count++;
93                  } else {
94                      Ngram bg = new Ngram(2);
95                      bg.n_gram[0] = word_i;
96                      bg.n_gram[1] = word_ip1;
97                      bg.count = 1;
98
99                      bigram.add(bg);
100                 }
101             }
102
103             if (textFile[i+2].charAt(0) == '<'){
104                 // do nothing (no words follow the
    current so a trigram cannot be built)
105                 continue;
```

```
106                 } else {
107                     // Same as word_i but textFile[i+2] is
    used to consider the word 2 lines ahead
108                     word_ip2 = textFile[i+2].split("\t")[0].
    toLowerCase();
109 
110                     if (checkExistence(trigram, new String
    []{word_i, word_ip1, word_ip2})){
111                         trigram.get(index).count++;
112                     } else {
113                         Ngram tg = new Ngram(3);
114                         tg.n_gram[0] = word_i;
115                         tg.n_gram[1] = word_ip1;
116                         tg.n_gram[2] = word_ip2;
117                         tg.count = 1;
118 
119                         trigram.add(tg);
120                     }
121                 }
122             }
123             file_count++;
124             System.out.println("file " + file_count + "/" +
    directoryListing.length + " done");
125         }
126     }
127 
128     // Method to check if a word is already in the language
    model, and if so returning its index
129     private static boolean checkExistence(ArrayList<Ngram>
    model, String[] words){
130         for (int i = 0; i < model.size(); i++) {
131             // comparing 2 arrays to check if their size and
    contents are identical
132             if (Arrays.equals(model.get(i).n_gram, words)){
133                 index = i;
134                 return true;
135             }
136         }
137         return false;
138     }
139 
140     // Class which will hold the n_grams (n=1 unigram, n=2
    bigram, n=3 trigram)
141     public static class Ngram implements Serializable{
142         final String[] n_gram;
143         int count;
144         double smoothedProbability;
145 
146         public Ngram(final int n) {
```

```
147             n_gram = new String[n];
148         }
149     }
150 
151     private static void smoothUnigram(ArrayList<Ngram>
    unigram){
152         // looping through all unigrams to find the total
    count
153         int totalCount = 0;
154         for (int i = 0; i < unigram.size(); i++) {
155             totalCount += unigram.get(i).count;
156         }
157 
158         int vocabSize = unigram.size();
159 
160         for (int i = 0; i < unigram.size(); i++) {
161             // (count(Wn) + 1) / (totalcount(all Wn) + V)
162             unigram.get(i).smoothedProbability = (double) (
    unigram.get(i).count + 1)/(totalCount + vocabSize);
163         }
164     }
165 
166     private static void smoothBigram(ArrayList<Ngram> unigram
    , ArrayList<Ngram> bigram){
167         int vocabSize = unigram.size();
168 
169         for (int i = 0; i < bigram.size(); i++) {
170             // Finding the count of w_n-1
171             int count_wordNminus1 = 0;
172             for (int j = 0; j < unigram.size(); j++) {
173                 if (bigram.get(i).n_gram[0].equals(unigram.
    get(j).n_gram[0])){
174                     count_wordNminus1 = unigram.get(j).count;
175                 }
176             }
177             // (count(Wn-1,Wn) + 1) / (count(Wn-1) + V)
178             bigram.get(i).smoothedProbability = (double) (
    bigram.get(i).count + 1)/(count_wordNminus1 + vocabSize);
179         }
180     }
181 
182     private static void smoothTrigram(ArrayList<Ngram>
    unigram, ArrayList<Ngram> bigram, ArrayList<Ngram> trigram
    ){
183         int vocabSize = unigram.size();
184 
185         for (int i = 0; i < trigram.size(); i++) {
186             // Finding the count of w_n-2,w_n-1
187             int count_wordNminus2and1 = 0;
```

```
188             for (int j = 0; j < bigram.size(); j++) {
189                 if (Arrays.equals(new String[]{trigram.get(i)
    .n_gram[0], trigram.get(i).n_gram[1]}, bigram.get(j).
    n_gram)){
190                     count_wordNminus2and1 = bigram.get(j).
    count;
191                 }
192             }
193             // (count(Wn-2,Wn-1,Wn) + 1) / (count(Wn-2,Wn-1)
    + V)
194             trigram.get(i).smoothedProbability = (double) (
    trigram.get(i).count + 1)/(count_wordNminus2and1 +
    vocabSize);
195         }
196     }
197
198
199     // Method to Save objects to Disk
200     private static void saveModels(ArrayList<Ngram> unigram,
    ArrayList<Ngram> bigram, ArrayList<Ngram> trigram) throws
    IOException {
201         String fileName;
202         FileOutputStream fos;
203         ObjectOutputStream oos;
204
205         fileName = "C:\\NLP\\Language Models\\unigram";
206         fos = new FileOutputStream(fileName);
207         oos = new ObjectOutputStream(fos);
208         oos.writeObject(unigram);
209
210         fileName= "C:\\NLP\\Language Models\\bigram";
211         fos = new FileOutputStream(fileName);
212         oos = new ObjectOutputStream(fos);
213         oos.writeObject(bigram);
214
215         fileName= "C:\\NLP\\Language Models\\trigram";
216         fos = new FileOutputStream(fileName);
217         oos = new ObjectOutputStream(fos);
218         oos.writeObject(trigram);
219
220         fos.close();
221         oos.close();
222     }
223 }
```

1.5 Source Code Listing - Main Class

```
1  import java.io.FileInputStream;
2  import java.io.IOException;
3  import java.io.ObjectInputStream;
4  import java.util.ArrayList;
5  import java.util.Scanner;
6
7  public class Main {
8
9      // String which stores the result of the generateText
       method
10     private static String word_i;
11     // Integer which stores the result of the wrongWord
       method
12     private static int wwi;
13
14     // Declaring all the language models as global variables
15     private static ArrayList<BuildModel.Ngram> unigram = new
       ArrayList<>();
16     private static ArrayList<BuildModel.Ngram> bigram = new
       ArrayList<>();
17     private static ArrayList<BuildModel.Ngram> trigram = new
       ArrayList<>();
18
19     public static void main(String[] args) throws IOException
       , ClassNotFoundException {
20         // Loading the Language Model
21         loadModels();
22
23         // Asking the user which function he wishes to
       perform (Main Menu)
24         Scanner sc = new Scanner(System.in);
25         String choice = "0";
26         do {
27             System.out.println("Press \"1\". Generate Text");
28             System.out.println("Press \"2\". Wrong Word");
29             System.out.println("Press \"3\". Terminate
       Program");
30             choice = sc.nextLine();
31             switch (Integer.parseInt(choice)) {
32                 case 1: {
33                     System.out.println("Please enter the text
        you would like the program to continue.");
34                     String input = sc.nextLine();
35                     if (input.length() == 0){
36                         System.out.println("No input was
       received, please try again");
37                         continue;
```

```
38                      }
39                      String lowerCaseInput = input.toLowerCase
    ();
40                      String[] text = lowerCaseInput.split(" ")
    ;
41
42                      generateText(text);
43                      System.out.println(input + " + " + word_i
    );
44
45                      break;
46                  }
47                  case 2: {
48                      System.out.println("Please enter the text
     you would like the program to detect the most likely
    error in.");
49                      String input = sc.nextLine();
50                      if (input.length() == 0){
51                          System.out.println("No input was
    received, please try again");
52                          continue;
53                      }
54                      String lowerCaseInput = input.toLowerCase
    ();
55                      String[] text = lowerCaseInput.split(" ")
    ;
56                      if (text.length < 3){
57                          System.out.println("Please enter 3 or
     more words for the program to detect the most likely
    misspelt word");
58                          continue;
59                      }
60
61                      wrongWord(text);
62                      System.out.println("Most likely misspelt
    word is: " + text[wwi]);
63
64                      break;
65                  }
66              }
67          } while (Integer.parseInt(choice) != 3);
68          System.out.println("Program is terminating");
69      }
70
71
72
73      private static void generateText(String text[]){
74          // finding the penultimate (word_i-2) and last words(
    word_i-1) in the entered sentence (programm will generate
```

```
    word_i)
75          String word_im2, word_im1;
76
77          if (text.length == 1){   // If the user has only
    entered one word
78              word_im1 = text[0];
79
80              ArrayList<BuildModel.Ngram> bigramMatches = new
    ArrayList<>();
81              bigramMatches = findBigramMatches(word_im1);
82
83              // If no matching bigrams are found, back off to
    unigram model
84              if (bigramMatches.size() == 0){
85                  System.out.println("Using the Unigram model")
    ;
86                  word_i = unigram.get( probabilisticChoice(
    unigram) ).n_gram[0];
87              } else {
88                  // instead of choosing the matching bigram
    with the highets probability, it is chosen
    probabilistically
89                  // using the probabilisticChoice method
90                  /*
91                  int highestProbability = 0;
92                  for (int i = 1; i < bigramMatches.size(); i
    ++) {
93                      if (bigramMatches.get(i).
    smoothedProbability > bigramMatches.get(highestProbability
    ).smoothedProbability){
94                          highestProbability = i;
95                      }
96                  }
97                  */
98                  System.out.println("Using the Bigram model");
99                  word_i = bigramMatches.get(
    probabilisticChoice(bigramMatches) ).n_gram[1];
100             }
101         } else {                        // If the user has
    entered 2 or more words
102             word_im2 = text[text.length - 2];
103             word_im1 = text[text.length - 1];
104
105             // finding all trigrams which start with the last
     2 words of the entered sentence
106             ArrayList<BuildModel.Ngram> trigramMatches = new
    ArrayList<>();
107             trigramMatches = findTrigramMatches(word_im2,
    word_im1);
```

```
108
109             // if no matching trigrams are found (back off)
110             ArrayList<BuildModel.Ngram> bigramMatches = new
    ArrayList<>();
111             if (trigramMatches.size() == 0){
112                 bigramMatches = findBigramMatches(word_im1);
113             } else {
114                 System.out.println("Using the Trigram model")
    ;
115                 word_i = trigramMatches.get(
    probabilisticChoice(trigramMatches) ).n_gram[2];
116             }
117
118             // if no matching trigrams or bigrams are found,
    back off to unigram model
119             if (trigramMatches.size() == 0 && bigramMatches.
    size() == 0){
120                 System.out.println("Using the Unigram model")
    ;
121                 word_i = unigram.get( probabilisticChoice(
    unigram) ).n_gram[0];
122             } else if (trigramMatches.size() == 0 &&
    bigramMatches.size() != 0){
123                 System.out.println("Using the Bigram model");
124                 word_i = bigramMatches.get(
    probabilisticChoice(bigramMatches) ).n_gram[1];
125             }
126         }
127     }
128
129     // Returns an ArrayList of trigrams, for which words_im2
    and word_im1 are the last words in the input sentence
130     private static ArrayList<BuildModel.Ngram>
    findTrigramMatches(String word_im2, String word_im1){
131         ArrayList<BuildModel.Ngram> trigramMatches = new
    ArrayList<>();
132
133         for (int i = 0; i < trigram.size(); i++) {
134             if (trigram.get(i).n_gram[0].equals(word_im2) &&
    trigram.get(i).n_gram[1].equals(word_im1)){
135                 trigramMatches.add(trigram.get(i));
136             }
137         }
138
139         return trigramMatches;
140     }
141
142     // Returns an ArrayList of bigrams, for which word_im1 is
     the last word in the input sentence
```

```
143     private static ArrayList<BuildModel.Ngram>
    findBigramMatches(String word_im1){
144         ArrayList<BuildModel.Ngram> bigramMatches = new
    ArrayList<>();
145 
146         for (int i = 0; i < bigram.size(); i++) {
147             if (bigram.get(i).n_gram[0].equals(word_im1)){
148                 bigramMatches.add(bigram.get(i));
149             }
150         }
151 
152         return bigramMatches;
153     }
154 
155     // Method which chooses an ngram based on its smoothed
    probability
156     private static int probabilisticChoice(ArrayList<
    BuildModel.Ngram> ngram){
157         // finding the total probability of all the elements
    in the given ngram
158         double totalProbability = 0.0;
159         for (int i = 0; i < ngram.size(); i++) {
160             totalProbability += ngram.get(i).
    smoothedProbability;
161         }
162 
163         // divide all the probabilities by the
    totalProbability so as to make them out of 1 (x% of 100%)
164         for (int i = 0; i < ngram.size(); i++) {
165             ngram.get(i).smoothedProbability /=
    totalProbability;
166         }
167 
168         // choosing an ngram based on its probability
169         double p = Math.random();
170         double cumulativeProbability = 0.0;
171         int i;
172         for (i = 0; i < ngram.size(); i++) {
173             cumulativeProbability += ngram.get(i).
    smoothedProbability;
174             if (cumulativeProbability > p) {
175                 break;
176             }
177         }
178         return i;
179     }
180 
181     private static void wrongWord(String text[]){
182         // array containing the probability of each
```

```
individual word in the sentence
183         double word_probability[] = new double[text.length];
184         // array containing the number of times each word was
 matched with an n-gram
185         int word_match_count[] = new int[text.length];
186 
187         // Spell Checking using Trigrams
188         for (int i = 0; i < text.length - 2; i++) {
189             for (int j = 0; j < trigram.size(); j++) {
190                 if (trigram.get(j).n_gram[0].equals(text[i])
    && trigram.get(j).n_gram[1].equals(text[i+1]) && trigram.
    get(j).n_gram[2].equals(text[i+2])){
191                     word_probability[i] += trigram.get(j).
    smoothedProbability;
192                     word_probability[i+1] += trigram.get(j).
    smoothedProbability;
193                     word_probability[i+2] += trigram.get(j).
    smoothedProbability;
194                     break;
195                 }
196             }
197         }
198 
199         // Spell Checking using Bigrams
200         for (int i = 0; i < text.length-1; i++) {
201             for (int j = 0; j < bigram.size(); j++) {
202                 if (bigram.get(j).n_gram[0].equals(text[i])
    && bigram.get(j).n_gram[1].equals(text[i+1])){
203                     word_probability[i] += bigram.get(j).
    smoothedProbability;
204                     word_probability[i+1] += bigram.get(j).
    smoothedProbability;
205                     break;
206                 }
207             }
208         }
209 
210         // Spell Checking using Unigrams
211         for (int i = 0; i < text.length; i++) {
212             for (int j = 0; j < unigram.size(); j++) {
213                 if (text[i].equals(unigram.get(j).n_gram[0]))
    {
214                     word_probability[i] += unigram.get(j).
    smoothedProbability;
215                     break;
216                 }
217             }
218         }
219 
```

```
220         // Initialising the values of the word_match_count
    array
221         for (int i = 0; i < word_match_count.length; i++) {
222             // the first and last word are matched 3 times
223             if (i == 0 || i == word_match_count.length - 1) {
224                 word_match_count[i] = 3;
225                 continue;
226             }
227             // if the sentence is 3 words long
228             if (word_match_count.length == 3) {
229                 // the second word is matched 4 times
230                 word_match_count[1] = 4;
231             } else {    // if the sentence is of any other
    length
232                 // the second and penultimate word are
    matched 5 times
233                 if (i == 1 || i == word_match_count.length -
    2) {
234                     word_match_count[i] = 5;
235                 } else {    // all other words are matched 6
    times
236                     word_match_count[i] = 6;
237                 }
238             }
239         }
240
241         // normalising the word probabilities, by dividing
    each word's probability by its word match count
242         for (int i = 0; i < word_probability.length; i++) {
243             word_probability[i] /= word_match_count[i];
244         }
245
246         // finding the word with the lowest probability
247         wwi = 0;
248         for (int i = 1; i < word_probability.length; i++) {
249             if (word_probability[i] < word_probability[wwi]){
250                 wwi = i;
251             }
252         }
253     }
254
255     private static void loadModels() throws IOException,
    ClassNotFoundException {
256         String fileName;
257         FileInputStream fis;
258         ObjectInputStream ois;
259
260         fileName= "C:\\NLP\\Language Models\\unigram";
261         fis = new FileInputStream(fileName);
```

```
262             ois = new ObjectInputStream(fis);
263             unigram = (ArrayList<BuildModel.Ngram>) ois.
        readObject();
264 
265             fileName= "C:\\NLP\\Language Models\\bigram";
266             fis = new FileInputStream(fileName);
267             ois = new ObjectInputStream(fis);
268             bigram = (ArrayList<BuildModel.Ngram>) ois.readObject
        ();
269 
270             fileName= "C:\\NLP\\Language Models\\trigram";
271             fis = new FileInputStream(fileName);
272             ois = new ObjectInputStream(fis);
273             trigram = (ArrayList<BuildModel.Ngram>) ois.
        readObject();
274 
275             fis.close();
276             ois.close();
277         }
278 }
```