Viterbi Algorithm and N-gram models

Nan Hu

Mar. 7th, 2016

1 Introduction

This report consists of three sections. First, we look at word-level n-grams of English and implement a method to find the most likely third word given a two-word combination. Then, we look at character-level n-grams of English and implement a method to find the most likely third character given a two-character combination. "Characters" in this context mean all lower- and upper-case letters and the space character. Finally, we implement a method using the Viterbi algorithm to do spell checking for a given document.

An n-gram is a sequence of n continuous items from a given sequence of text. In this report, we focus on bigrams and trigrams, which are n-grams of size 2 and 3, respectively. Word-level n-grams mean n words from a text, and character-level n-grams mean n characters from a text. Please note that all the text used is taken from Project Gutenberg and that the text has been processed to only contain lower- and upper-case letters and whitespace. In other words, symbols such as "!" or "-" are not considered.

Acknowledgement: This report is based on a past AI assignment, which can be found here: https://web.archive.org/web/20150721045217/http://www.cs.dartmouth.edu/ devin/cs76/assignments/06viterbi/viterbi.html. The Perl script that processes the text is provided as part of the assignment.

2 Word-level N-grams

The question we want to investigate is this: given two words w_1 and w_2, what is the probability $P(w_3|w_1 \wedge w_2)$. Note that $P(a|bigram) = P(a \wedge bigram)/P(bigram)$. To this end, we want to construct frequencies of both bigrams and trigrams from given files. In order to have the ability to read in text from multiple files, and thus have a better model to predict words from, we choose to store words in a map of String-ArrayList¡String¿ pairs, where the key is the filename and the entry is the list of words in that file. For both bigrams and trigrams, we store a map of n-gram-frequency pairs and keep an actual count of frequencies instead of probabilities. Actual probabilities will be very small, so in order to not have instability, we calculate probabilities based on the recorded frequencies.

Each bigram and trigram is can be though of as a String array, but to make coding easier, we create a wrapper class, `NGram`, around each array. The class is very simple, with a constructor that takes a Sting array and a getter that returns the array being stored.

2.1 Document Parsing

In the process of reading in each file, we make sure that only words and spaces are being read in. Each file has already been removed of punctuation, but we also need to make sure we removed leading and trailing whitespace and newlines:

```
String line;
while(input.hasNextLine()) {
 line = input.nextLine();
 if (line.trim().equals("\n")||line.trim().equals("")||line.isEmpty()) {
  continue;
 }
 if (toLower==true) {
  line = line.toLowerCase();
 }
 String[] tokens = line.trim().split(" ");
 for (int i=0; i<tokens.length; i++) {
  if (tokens[i].equals(" ")||tokens[i].equals("\n")) {
   continue;
  }
  wordsInFile.add(tokens[i]);
 }
}
```

The method for getting bigrams and trigrams is fairly straightforward. For each file, and for all the words in that file, we look at every two (or three) contiguous words, create an `NGram` object from these words, and put them into the map of n-gram-frequencies pairs. In addition, we keep a counter of the total number of bigrams and trigrams present in the text, as we will need this later to compute probabilities. Here is the method for creating bigrams:

```
public void createBigram() {
 for (String file:words.keySet()) {
  ArrayList<String> wordsInFile = words.get(file);
  for (int i=0; i<wordsInFile.size()-1; i++) {
   String word1 = wordsInFile.get(i);
   String word2 = wordsInFile.get(i+1);
   String[] in = {word1, word2};
   NGram bigram = new NGram(in);
   if (!bigramFreq.containsKey(bigram)) {
    bigramFreq.put(bigram, 1);
   }
   else {
    Integer freq = bigramFreq.get(bigram);
    bigramFreq.put(bigram, freq+1);
   }
   sumBigram++;
  }
 }
}
```

For trigrams, the method is nearly identical, expect we want to iterate until the second to last word and get three words instead of two. Of course, the map that we should add to is the `trigramFreq` map, and we should increment `sumTrigram` instead.

2.2 Finding the Next Word

To find the third word, given two already, we need to first find the probability of occurrence of all possible third words. To find the probability of a given third word, we need to first find the probability of the bigram, which can be done by dividing the frequency of the bigram by the total number of bigrams. Then, we find the probability of the trigram (the bigram plus the given word), which is dividing the frequency of the trigram by the total number of bigrams. Finally, we find the conditional probability of the getting the trigram given the bigram, so we divide the trigram probability by the bigram probability. Of course, it is possible that a given bigram or trigram does not exist in our model, and if this is the case, we return a zero probability. Below is the code:

```java
public double findProb(String[] input) {
 double probInput = 0;
 double probBigram = 0;
 String[] in = {input[0],input[1]};
 NGram bigram = new NGram(in);
 NGram trigram = new NGram(input);
 if (!bigramFreq.containsKey(bigram)) {
  System.out.println("Bigram not found!");
  return 0.0;
 }
 else {
  probBigram = (double)bigramFreq.get(bigram)/sumBigram;
 }
 if (!trigramFreq.containsKey(trigram)) {
  System.out.println("Trigram not found!");
  return 0.0;
 }
 else {
  probInput = (double)trigramFreq.get(trigram)/sumTrigram;
 }
 return probInput/probBigram;
}
```

The second part is to find the third word that has the highest probability of occurrence. To start, we are given a bigram to predict from. If the bigram does not exist, we exit the method with no result. Otherwise, we iterate through all the trigrams, and for each trigram that contains the given bigram, we calculate its probability of occurrence using the method above. By "contains", we mean that the first two words of the trigram is the bigram. We keep track of the highest probability and also which trigram has that probability, and we return the trigram at the end. Here is the code:

```java
public NGram nextWord(NGram bigram) {
 if (!bigramFreq.containsKey(bigram)) {
  System.out.println("Given bigram does not exist!");
  return null;
 }
 int freq = 0;
 NGram result = null;
 for(NGram trigram:trigramFreq.keySet()) {
  if(bigram.getNgram()[0].equals(trigram.getNgram()[0])
   &&bigram.getNgram()[1].equals(trigram.getNgram()[1])) {
   if (trigramFreq.get(trigram)>freq) {
    result = trigram;
    freq = trigramFreq.get(result);
   }
  }
 }
 return result;
}
```

Note that the above method will always return the most likely word. It is possible to return words based on the probability distribution (i.e. if there are three possible third words for a given bigram and one is twice as likely as the others and we attempt to find the third word multiple times, then we can expect all three words to appear, with one appearing twice as often as the others), this does not make much sense.

Most users will want to find the most likely word, and in terms of speech recognition and natural language processing, only most likely words matter. However, it is not difficult to get words based on a probability distribution. The method to do this is included in the source code and is used in the driver code.

2.3 Is It English?

To test how well our predictor performs, we read in text from Locke's Second Treatise on Government and Kafka's Metamorphosis and create a list of 10 bigrams from which to find a third word. Here are our results:

- on the other
- next to it
- in the state
- this is the
- of his own
- of the society
- it was not
- because of his
- such a state
- if such a

We see that the phrases above do make sense in the context of English. For example, "next to it" and "it was not" are common phrases that we use daily. The multiple instances of the word "state" might be because we used a piece from Locke talking about government. If we instead used a text like Gulliver's Travels (which is included with the source code), we may expect to see words such as "Yahoo" or "Lilliput". The type of text that is read in does affect the kind of phrases that are predicted. Overall, we see that the predictor is very good.

We can also try to predict longer phrases, by construct a phrase word by word and using the most recent two words to predict the next. The result we get for a 50-word phrase is "there is commonly in the state of nature gave me no title to it also and they that fought on his side and a right to the goods of another sect. nay this power to which in time the measures we have no manner of governing his subjects as a subject". We see that this doesn't quite read like a sentence, although certain phrases within the larger phrase make sense, such as "we have no manner of governing his subjects".

3 Character-level N-grams

The question we investigate here is the same as the one we investigated in word-level n-grams, expect instead of words, we look at individual characters. To phrase the question again: given two characters c_1 and c_2, what is the probability $P(c_3|c_1 \wedge c_2)$. Again, note that $P(a|bigram) = P(a \wedge bigram)/P(bigram)$. The process of constructing character-level n-grams is very similar to the process for the word level, and we keep the same data structures. However, there are some key differences, which we will discuss as they come up. First is the matter of not all trigrams existing within the text we read in. The characters we consider are lower- and upper-case letters and space, so there are $53^3 = 148877$ total trigrams. To make sure that each trigram can be represented, we need to create all possible trigrams and add them to the map.

3.1 Document Parsing

As with parsing for words, we want to make sure that leading and trailing whitespace, as well as newlines, are removed. In addition, we do the added check of whether an individual character is actually in the alphabet or a space.

```
String w = "";
 while (input.hasNextLine()) {
  String line = input.nextLine();
  if (line.isEmpty()||line.trim().equals("")||line.trim().equals("\n")) {
    continue;
  }
  for (String word : line.trim().split("\\s+")) {
   w += word;
   w += "␣";
  }
 }
 for (int i = 0; i < w.length(); i++) {
  char c = w.charAt(i);
  int n = (int) c;
  if ((65<=n&&n<=90)||(97<=n&&n<=122)||n==32) {
    charsInFile.add(c);
  }
 }
```

3.2 Finding the Next Character

To find the third character, given two already, we need to first find the probability of occurrence of all possible third characters. Since we have created all trigrams in the constructor, we will have to look at 53 trigrams, each with a non-zero probability. To find the probability of a given third character we need to first find the probability of the bigram, which can be done by dividing the frequency of the bigram by the total number of bigrams. Even if the given bigram does not exist, we give the bigram a probability of one divided by the total number of bigrams. Then, we find the probability of the trigram (the bigram plus the given character), which is dividing the frequency of the trigram by the total number of bigrams. Finally, we find the conditional probability of the getting the trigram given the bigram, so we divide the trigram probability by the bigram probability. For character-level n-grams, we will not encounter instances of zero probabilities.

Since the methods for constructing bigram and trigram frequencies, finding probabilities, and finding the most likely next character are so similar to those used for word-level n-grams, we will not give the code here; rather, please refer to the source code.

3.3 Is It English?

To test our predictor, we read in text from Locke's Second Treatise on Government and Kafka's Metamorphosis, as before, and create a list of 10 bigrams from which to find a third character. Here are our results:

- are
- his
- mon
- for
- not
- bre
- tat
- ne
- whi
- wer

These results are pretty good and mostly make sense. Several of these trigrams, such as "are" and "not", are three-letter English words, so they are perfectly acceptable results. Others, such as "mon" and "bre", are parts of longer words. "mon" is the start of "monday", and "bre" can be the start of "breadth" or "bread". Even the one that doesn't make too much sense, which is "ne " (it ends in a space), is still appropriate. There are English words that end in "ne", so the inclusion of a space after that is still possible.

In addition, as with words, we can return characters following a probability distribution. However, as is the case with words, this form of returning characters does not have many uses. The method for doing so is included and may be used in the driver.

4 Spell Checking

In spell checking, we want to create two copies of a file, introduce errors to one copy, attempt to correct the errors using the Viterbi algorithm, and look at how good our spelling correction is. For spell checking, we clearly need to know the frequency and probability of character-level bigrams and trigrams, so we create an instance of the character-level n-gram class using an overloaded constructor that takes a list of characters (we only look at text from one file, and this list represents characters in that file) and creates the bigram and trigram frequency maps. To read in the text, we use the same method that we use for character-level n-grams, except this time we only keep the list of characters. To introduce errors, we iterate through all the characters, and for each character, we generate a random number between 0 and 1. If the random number is less than the specified error rate and the character is not a space, we change the character to a random other letter. Otherwise, the character is unchanged. The character is then added to the copy of the text that has spelling errors, thus giving us two copies, one good and one bad.

```java
public void inputErrors() {
 for (int i=0; i<chars.size(); i++) {
  double err = random.nextDouble();
  if (err<errRate&&chars.get(i)!=' ') {
   char rand = alphabet[random.nextInt(alphabet.length -1)];
   errChars.add(rand);
  }
  else {
   errChars.add(chars.get(i));
  }
 }
}
```

The next step is to use the Viterbi algorithm to do spelling correction. To do this, we first start with a map of prior probabilities, a probability associated with each character, all initially 0. We also need to create a list of backpointers, so we can backtrace the most likely sequence after we have computed probabilities. To compute probabilities at each step, we loop through all the characters in our copy of the text with errors, and for each character in the prior distribution and each character in the alphabet, we create a new trigram. We then get the probability of this trigram (this is the transition step) and then add to this a update value, which is based on the error rate. We then add this probability to the map of current probabilities. After this is done, we update the prior probabilities with the current ones and then go to the next step (which is the next letter in the bad copy). After the looping, we find the character at the last step with highest probability and do a backtrace from there. We also calculate the error to see if we have improved the spelling. Below is the code:

```java
public void viterbi() {
 HashMap<Character, Double> priors = new HashMap<Character, Double>();
 for (char character:alphabet) {
  priors.put(character, 0.0);
 }
 HashMap<Character, Double> probs = null;
 ArrayList<HashMap<Character, Character>> backpointers
 = new ArrayList<HashMap<Character, Character>>();
 for (int i=2; i<errChars.size(); i++) {
  HashMap<Character, Character> backpointer = new HashMap<Character, Character>();
```

```java
   for (Character prior:priors.keySet()) {
    for (char c:alphabet) {
     char[] trigram = {errChars.get(i-2), errChars.get(i-1), c};
     double transition = priors.get(prior) + Math.log(this.model.findProb(trigram));
     double update;
     if (c==errChars.get(i)) {
      update = 1 - this.errRate;
     }
     else {
      update = this.errRate;
     }
     double prob = transition + Math.log(update);
     if (!probs.containsKey(c)||prob > probs.get(c)) {
      probs.put(c, prob);
      backpointer.put(c, prior);
     }
    }
   }
   priors = probs;
   backpointers.add(backpointer);
  }
  double highest = Double.MAX_VALUE * -1;
  Character optimal = null;
  for (Character c:probs.keySet()) {
   if (probs.get(c)>highest) {
    highest = probs.get(c);
    optimal = c;
   }
  }
  ArrayList<Character> path = new ArrayList<Character>();
  path.add(optimal);
  for (int i = backpointers.size()-1; i>0; i--) {
   Character prev = backpointers.get(i).get(optimal);
   path.add(prev);
   optimal=prev;
  }
  path.add(errChars.get(1));
  path.add(errChars.get(0));
  Collections.reverse(path);
  for (Character c : path) {
   System.out.print(c);
  }
  System.out.println();
  int diffs = 0;
  for (int i=0; i<chars.size(); i++) {
   if (chars.get(i)!=path.get(i)) {
    diffs++;
   }
  }
  double rate = (double)diffs/chars.size();
  System.out.println(rate);
```

```
}
```

We test the implementation using an excerpt of text from John Locke. When we test with a 0.05 error rate, we find that after correction, the error is around 0.035. With a 0.1 error rate, we get a post-correction error of 0.094, which is not as good of an improvement. For a 0.2 error rate, we get 0.2227, which is decidedly worse. One possible explanation is that we are allowing whitespace characters to be changed in accordance to probability, even though we know that whitespace characters in the error copy are always whitespace. This is because when we input errors, we only do so if a character is not a space. If we have a way to make sure that whitespace characters are automatically added, unchanged, to the most likely sequence, we should be able to improve. We also notice that for larger files, the post-correction errors tend to be larger, possibly due to the same reason.

5 Further Discussion

5.1 Word and Character Prediction

We have seen that the predictions for the next word or character are pretty good and make good sense. However, we have also seen that trying to predict longer phrases and sentences results in something that doesn't make much sense. Clearly, this can be improved by providing more context, such as predicting based on more than just two words or characters. In addition, factoring in more attributes may help with prediction. Right now, the only consideration is probability. If we factor in features such as word meaning or part of speech, we could get better, more meaningful results.

5.2 Spell Checking

There is a glaring issue with the current method of spell checking, and that is what happens if two wrong characters appear in a row. Since we are predicting based on bigrams, if the bigram we are predicting from is entirely incorrect, it is very unlikely that we recover from this and get the correct third character. In addition, because trigram probabilities are generally very low and many trigrams end up having the same probability, it is difficult to pick the most likely one. Probabilities can be improved if we use data such as the average distribution of all trigrams in the English language. Currently, we are limited to the text that we read in, which may or may not be representative of English as a whole.

Another problem with our current method is that it is possible to change a correctly spelled word into an incorrectly spelled one. We are only looking at probability, and it is possible that a correctly spelled word does not use the most common combinations of trigrams, resulting in its being changed. If we first look at the error copy word by word and make note to ignore those that are spelled correctly, even if we still apply the same method to the remaining words, our improvement would be much better.

6 Conclusion

In this report, we construct bigrams and trigrams of words and characters in the English language and use the Viterbi algorithm to spell check. We see that using probabilisitic reasoning, we can predict words and characters with reasonable accuracy. We also see that the Viterbi spell checker performs well on shorter texts with higher frequencies of repeating trigrams. In the end, we outline further methods for improvement.