Assignment #2
Discovering Affixes Automatically

Thuong-Hai Pham

January 28, 2017

Abstract

1 TrieNode

1.1 Fields

Field Detail

nodeChar

```
private char nodeChar
```

endToken

```
private boolean endToken
```

tokenCount

```
private int tokenCount
```

ChildNodes

```
java.util.ArrayList<TrieNode> ChildNodes
```

Although we have three different kinds of node in our algorithm, these nodes are implemented in only one single class because the attributes in this class and the way we construct our algorithm help distinguishing an instance into these three kinds.

First, every node instance is created (equal and) to be “regular node”. Then, by having the attribute “endToken” set to true or false, it will be considered as “end token node” or still “regular node”, respectively.

After introducing “endToken” attribute, the only problem is how to mark “root node” from those two. If we look closely to the diagram, there is only one root node which has no parent, i.e. no node points to this root node as its child. Hence, we will have to handle this node from outside the tree, e.g. in “main class” or in “Trie Dictionary” (which will be implemented later). Therefore, there is no need to mark one node is root or not.

1.2 Methods

Our straightforward approach to manage the child nodes is to create an ArrayList (supported by Java) of TrieNode to handle child nodes of a specific node.

```
        ArrayList<TrieNode> ChildNodes = new ArrayList<>();
```

Listing 1: Declaration of ChildNodes

Whenever we need to add a new child node (e.g. add new entry), we first check whether it exists then initiate the new one, add it in the list.

```
        TrieNode t = this.getChildNode(input.charAt(0));
        if (t == null) {
                t = new TrieNode(input.charAt(0));
                this.ChildNodes.add(t);
        }
```

Listing 2: Add new child node

Method Detail

getNodeChar

```
private char getNodeChar()
```

getNodeChar

Returns:

`nodeChar attribute`

getChildNode

```
public TrieNode getChildNode(char c)
```

getAllChildNodes

```
public java.util.List<TrieNode> getAllChildNodes()
```

getAllChildNodes

Returns:

`all elements in ChildNodes attribute (wrapper in case ChildNodes is a map)`

addEntry

```
boolean addEntry(java.lang.String input,
                 int count)
```

addEntry adds the input string to Trie from this current node, created new child node if needed

Parameters:

`input` - `string`

`count` - `occurrence of input string`

Returns:

`true if succeeded`

addChildNode

```
public void addChildNode(java.lang.Character c,
                         TrieNode t)
```

addChildNode add new node to be child of this current node, to be overwritten in other ChildNodes structure

Parameters:

`c - child's nodeChar`

`t - the new (already initiated) node`

hasEntry

```
boolean hasEntry(java.lang.String input)
```

hasEntry checks if this word exists in the branch starting from this node

Parameters:

`input - input string`

Returns:

`true if word exists in this TrieDictionary`

getTokenCount

```
int getTokenCount(java.lang.String input)
```

getTokenCount gets token count of the input string return 0 if the input string does not exist in the branch starting from this node

Parameters:

`input - input string`

Returns:

`token count`

2 TrieNodeAdv

However, the simple and straightforward approach above may face a disadvantage in which our algorithm has to search throughout all of elements of ChildNodes to find out the queried child. Hence, we implemented an advanced version of our TrieNode by inheriting TrieNode with modification of its ChildNodes attribute.

In this version, ChildNodes is implemented with a Java HashMap. This data structure helps to reduce the complexity of child searching from O(n) (n is number of elements in ChildNodes) down to approximation of O(logn) by searching and hash function.

Field Detail

ChildNodes

```
java.util.Map<java.lang.Character,TrieNode> ChildNodes
```

3 Trie Dictionary

Our TrieDictionary serves as a wrapper or entry point for the whole Trie itself, which basically holds the root node as mentioned in section 1.

3.1 Fields

Field Detail

ESP

```
private final double ESP
```

rootNode

```
private TrieNode rootNode
```

3.2 Methods

Method Detail

AddWord

```
boolean AddWord(java.lang.String word,
                int count)
```

addWord adds a new word and its occurrence to this dictionary

Parameters:

`word` - input word

`count` - occurrence

Returns:

true if succeeded

hasEntry

```
boolean hasEntry(java.lang.String word)
```

hasEntry checks if this TrieDictionary contains the input word

Parameters:

`word` - input

Returns:

true if word exists in this TrieDictionary

findAffixes

```
void findAffixes(java.lang.String word,
                 java.util.Map<java.lang.String,java.lang.Integer> result,
                 boolean verbose)
```

findAffixes tries to cut the word down in 2 chunks and applies tests as described in our lecture

Parameters:

`word` - input word

`result` - a map String -> Integer, which includes an affix and its score

`verbose` - print log or not

fromFile

```
java.util.Set<java.lang.String> fromFile(java.io.File file,
                                          boolean prefix)
                                   throws java.io.IOException
```

fromFile reads all lines from a file object then construct our Trie

Parameters:

`file` - input file, which is a file object

`prefix` - for the function to decide reverse each string or not

Returns:

a set of words in our database (i.e. the first entry in each line) for later use

Throws:

`java.io.IOException` - when file not found or error during reading file

getTokenCount

```
private int getTokenCount(java.lang.String word)
```

getTokenCount

Parameters:

`word` - the input string

Returns:

the number of tokens start with 'word' (i.e. number of tokens have 'word' as prefix)

In this class, we also implemented the affixes finding method to discover affixes in the whole Trie by examining the three criteria.

This also justifies the need of constant variable ESP (epsilon) which helps us to do the approximate comparison of "approximate to 1" and "much less than 1".

```java
void findAffixes(String word, Map<String, Integer> result,
        boolean verbose) {
        ...
        boolean test1 = this.hasEntry(alphaA);
        boolean test2 = Math.abs((float) freqAlphaA /
        freqAlpha - 1) < ESP;
        boolean test3 = (float) freqAlphaAB /
        freqAlphaA < 1 - ESP;
        ...
}
```

Listing 3: Affixes criteria

4 Main Class

Main class is home for our program entry point, and also equipped with a utility string reversing function. In Main, we also implemented a test as described in our assignment instruction to make sure the implementation basic functions work well before running with large datasets.

Method Detail

main

```
public static void main(java.lang.String[] args)
```

myReverse

```
private static java.lang.String myReverse(java.lang.String st)
```

myReverse

Parameters:

`st` - input string

Returns:

the reversed string

mainProd

```
private static void mainProd(java.lang.String inputFileName,
                             java.lang.String outputFileName,
                             boolean prefix)
```

mainProd which does the affix discovery

Parameters:

`inputFileName` - string

`outputFileName` - string

`prefix` - discover the prefix (if true) or suffix (false)

mainTest1

```
private static void mainTest1()
```

this is purely for testing purpose (instead of writing unit test)

5 Research question

5.1 Computational perspective

As discussed in section 2, TrieNodeAdv is implemented with HashMap from Java in an attempt to prevent linear search throughout ChildNodes and to reduce our complexity for that part from O(n) to O(logn). By running those two on our real dataset English.txt, we get the following result. (running time is measured in second (S))

```
TrieNode
        Constructing tree from file: PT4.978S
        Discovering affixes: PT10.413S

TrieNodeAdv
        Constructing tree from file: PT4.495S
        Discovering affixes: PT10.007S
```

Listing 4: Speed testing output TrieNode and TrieNodeAdv

It is clear that TrieNodeAdv is faster than its parent class. However, this improvement is subtle due to one possible explanation. The number of child nodes is varied, and does not always reach the limit of around 30. Therefore, the real n in average is much less than 30, which does not leverage O(n)-to-O(logn) improvement.

5.2 Linguistic perspective

As we can see in the output files, the algorithm worked pretty well when having discovered these top (sorted by scores) prefixes and suffixes:

Prefixes	Suffixes
un 48609	s 521319
re 27857	ly 60223
non 12638	ing 43367
in 12162	ed 41247
dis 11215	ness 19834
sub 11149	ers 15311
de 10921	es 13903
bio 8965	ism 12712
micro 8893	ally 12689
get 8750	al 10715
pre 8367	ist 8997
over 7909	er 7911

However, there exists an issue in which "-ers" shows up among its building-block real suffixes "-er" and "-s". This problem can be solved by prunning [Keshava and Pitler, 2006], in which we scan through our affixes list again. While scanning, if there exists a morpheme that is a concatenation of other two morphemes with higher scores, the unfortunate morpheme should be removed from our list.

One more problem in our algorithm is that some affixes are penalised so badly that they can not make themself to the final list. For example, the suffix of "-en" in "lengthen 1007", "strengthen 23200", "shorten 2621", "harden 1822" should be listed, yet not because it is penalised by its existence in "heaven 47060", "garden 176285" and similar nouns. It is obvious that this issue and its similar situations can be solved by distinguishing the Part-of-speech (POS) of both words and their segmented morphemes.

In the two problems above, solution for the first one still keeps the unsupervised nature of our algorithm. On the other hand, the second one, which requires an additional information, turns our algorithm into semi-supervised.

References

[Keshava and Pitler, 2006] Keshava, S. and Pitler, E. (2006). A simpler, intuitive approach to morpheme induction. In *Proceedings of 2nd Pascal Challenges Workshop*, pages 31–35.