CS6999 Programming Assignment 1

Justin Kamerman 3335272

March 15, 2011

Assignment

1. Implement the Aho-Corasick string matching algorithm[1] and test its performance for:

   - 1000 blogs and 100 keywords
   - 2,000 blogs and 100 keywords
   - 4,000 blogs and 100 keywords
   - 8,000 blogs and 100 keywords
   - 16,000 blogs and 100 keywords
   - 32,000 blogs and 100 keywords

   Repeat the experiments for 200 and 400 keywords.

2. Build an inverted index for 10, 000 blogs. Use 10 keywords to query the index to:

   - Locate each keyword and retrieve their corresponding blogs.
   - Show the intersection of every two keywords. For example, retrieve the document ID of the blogs that keywords 1 and 2 have occurred together at least once. Similarly, for keywords 1 and 3, ..., keywords 2 and 3, keywords 2 and 4, ...
   - Similarly, show the intersection of every 3 keywords, 4 keywords, etc.

3. Do a comparative analysis (time to build the index and time to retrieve) of the two methods.

1 Aho-Corasick Algorithm

Implementation

The Aho-Corasick algorithm are implemented by a Java program. The only external dependency is on the Apache commons-cli library for parsing commend line options. To that end, the program is operated from the command line, taking options listed in table 1.

Option	Description
-d	Generate Graphviz DOT output.
-f arg	Path of data file
-i arg	Number of iterations to perform. Default is 10
-n arg	A comma-separated list of attribute names, matching the order in which they appear in the data file
-o arg	Number of folds to create in the training data during. Default is 5.
-h	Print help message

Table 1: Command line options

The program implements various mechanism to facilitate debugging. Throughout the code, log statements have been added using the Java logging framework. The logging output is controlled for individual classes via the *logging.properties* file which the program read on startup. In addition to logging, code was added to generate Graphviz DOT (described in [2]) output representing the decision tree created. This output can be rendered using Graphviz dot tool. An sample dot file for the mushroom.data data set is shown below. The associated image generated by dot in Figure 1.

```
digraph G {
0  [label="0 []", shape=circle];
0 -> 3 [label="s"];
0 -> 1 [label="h"];
1  [label="1 []", shape=circle];
1 -> 2 [label="e"];
1 -> 6 [label="i"];
1 -> 0 [color="red"];
6  [label="6 []", shape=circle];
6 -> 7 [label="s"];
6 -> 0 [color="red"];
7  [label="7 [his]", shape=circle];
7 -> 3 [color="red"];
2  [label="2 [he]", shape=circle];
2 -> 8 [label="r"];
2 -> 0 [color="red"];
8  [label="8 []", shape=circle];
8 -> 9 [label="s"];
```

```
8 -> 0 [color="red"];
9  [label="9 [hers]", shape=circle];
9 -> 3 [color="red"];
3  [label="3 []", shape=circle];
3 -> 4 [label="h"];
3 -> 0 [color="red"];
4  [label="4 []", shape=circle];
4 -> 5 [label="e"];
4 -> 1 [color="red"];
5  [label="5 [she, he]", shape=circle];
5 -> 2 [color="red"];
}
```

The implementation classes and their relationships are represented in a UML class diagram in Figure 2. Following is a brief description of each class:

- **AhoIndexMain**
- **AhoSearchMain**
- **AhoIndexer**
- **AhoSearcher**
- **ExecutionContext**
- **StateIterator**
- **StateMachine**
- **Data**
- **Document**
- **DocumentTask**
- **LogFormatter** is a helper class to format log messages.
- **Match**
- **Timer**

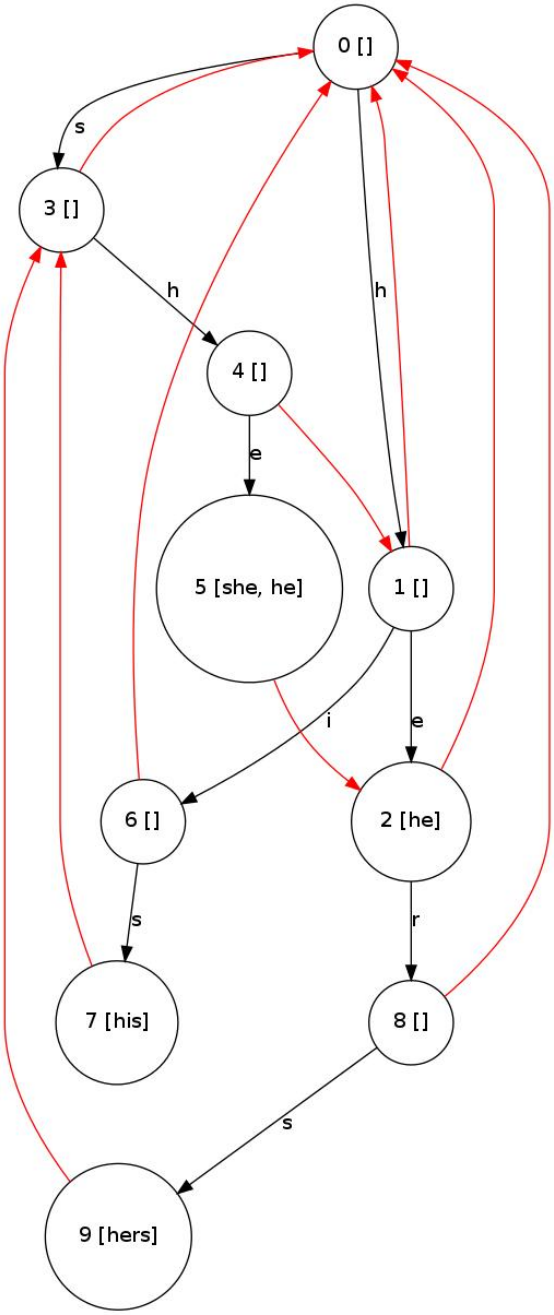

Figure 1: Aho-Corasick state machine visualization generated by DOT

Figure 2: Aho-Corasick Implementation UML class diagram

2 Inverted Index

Program expects data in CSV format with attributes occurring first and classification at the end of each line. All data files have been preprocessed to fit this format.

Implementation

Results

3 Algorithm Comparison

4 Results

The results of processing each data file using the NBTree implementation are summarized in Table 2. In general, learning time was significantly higher than both the ID3 and Naïve Bayes learning algorithms. Intuitively, this was expected because of the large number of Naïve Bayes classifiers that are created and evaluated during the utility calculation.

All tests were run on a personal computer with an AMD Athlon 64 2GHz Processor, 1GB RAM, running a 32 bit Linux 2.6.31 kernel. The Java Virtual Machine used was version 1.6.0-18.

Data Set	**Naïve Bayes**	**ID3**	**NBTree 5%**	**NBTree 3%**
car	0.841449	0.938377	0.882783	0.925797
ecoli	0.684776	0.384776	0.674328	0.668657
mushroom	0.952007	1.000000	0.951850	0.996921
letter-recognition	0.737670	0.760870	0.819500	
breast-cancer-wisconsin	0.972662	0.945899	0.972806	0.971223

Table 2: Comparison of Naïve Bayes, ID3, and NBTree accuracy over benchmark data sets. NBTree results for both 3% and 5% split utility thresholds are shown

Bibliography

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun.ACM*, vol. 18, no. 6, pp. 333–340, June 1975.

[2] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz - open source graph drawing tools," *Graph Drawing*, pp. 483–484, 2001.