COMPGS04/M024: Tools and Environments

Coursework 2 (Group Coursework)

Authors:
Jihyun Han
Elliott Omosheye
Sachin Pande
Luke Richardson
Yasaman Sepanj

March 22, 2014

1 Introduction

1.1 Sub-section

2 Paper 1

2.1 Sub-section

3 Paper 2

3.1 Sub-section

4 Paper 3

4.1 Sub-section

5 Latent Semantic Indexing

Manning, Raghavan and Schütze. Introduction to Information Retrieval: 18 Matrix decompositions & latent semantic indexing. Cambridge University Press. 2008.

5.1 Overview of technique

Latent Semantic Indexing (LSI) makes use of a term-document matrix, this is simply an MxN matrix, where columns represent the number of documents in the collection and rows represent the terms. Calculation can then be carried out over this matrix to compute the LSI.

In the case of our tool, there would be two columns, for the pair of documents. The terms would be tokens, produced by some simple whitespace parser or potentially more complex grammar.

5.1.1 Low-rank approximation and computing similarity

LSI first requires an approximation of the term-document matrix into lower-dimensional space using singular value decomposition, an extension of symmetric diagonal decomposition. Symmetric diagonal decomposition is a technique in which a square matrix can be factored into the product of matrices derived from its eigenvectors.

This approximation is required as even medium sized term-document matrices can contain tens of thousands of terms, so this is required for LSI to be scalable.

The mathematics of low-rank approximation is too complex to cover in a short summary, but the end result is a reduction of the size of the term-document matrix, while still maintaining differences and similarities between documents close to that of the original matrix.

Once the low-rank approximation has been computed, the similarity between two documents can be calculated using cosine similarity.

5.2 Performance

Dumais 1993 and Dumais 1995 conducted experiments on TREC documents and tasks. They managed to achieve results with precision at or above the TREC median, achieving a top score on 20% of topics. Therefore we can conclude that the accuracy of LIS is, in general, good.

The performance of LSI is noted to be poor in terms of computational complexity. It is noted that "there have been no successful experiments with over one million documents". Although this is something of concern for some, for us this would be a tolerable limitation, considering we are only dealing with two input files.

LSI is noted to "work best in applications where there is little overlap between queries and documents". For calculating document similarity, ideally we would have more linear performance and be similarly good at working with considerable overlap.

LSI is a vector space model and as such treats documents as a 'bag of words'. This could either prove to improve similarity matching or degrade it, as source code can be sufficiently reordered to produce a program with distinct behaviour, but LSI will still produce results that show a close match. Conversely if someone was to rearrange code to attempt to disguise plagiarism, LSI matching would be unaffected.

Finally there are two other limitations of LSI related to its vector space representation, its inability to cope with synonymy and polysemy. Synonymy refers to two distinct words having the same meaning and polysemy one word being having different meanings dependent on use. This would be an issue for programming language code, where certain words, such as access modifiers and Java keywords are used frequently throughout text.

5.3 Conclusion

Overall LSI seems to be a good similarity matching algorithm. However, for our particular use case, matching Java source code, the weaknesses due to it being a vector space model are likely to put it at a disadvantage to other models which take into account word ordering.

6 Paper 5

6.1 Sub-section

7 Tool requirements

7.1 Overview

The following list of requirements are formed from the defined set of tool requirements in the coursework brief and based on our algorithm choice, JPLAG plagiarism detection.

7.2 Requirements

1. The tool will compute a similarity measurement for two input Java source code files using an implementation of the JPLAG plagiarism detection algorithm.
2. The tool must work via command line and should not have a graphical interface.
3. Input taken should be the names of two files as command lines arguments.
4. The output should be a similarity measure to STDOUT as a percentage.
5. The tool will output additional results from running the plagiarism detection algorithm
6. Input Java source files will be parsed using a Java parser library, ANTLR.
7. The tool will be written in Java (version 6) to meet the requirements for the parser library.
8. The tool will not invoke the JPLAG binary or any other similarity testing tool.
9. The output of similarity measurement by plagiarism detection algorithm will complete in reasonable time for expected inputs. Expected inputs are generally Java source code files and specifically under testing TOH.java, variants of TOH.java and arbitrary dissimilar test inputs.
10. The tool will have an automated testing mechanism, using the JUnit testing framework to carry out pairwise comparison.
11. The tool should work with all platforms the JVM works on that support ANTLR. The tool will be tested and run on Linux and Windows platforms.

8 UML Diagrams of architecture

8.1 Sub-section

9 Tool implementation

9.1 Sub-section

10 Testing

10.1 Sub-section

11 Results of pairwise comparison

11.1 Sub-section

12 Evaluation of results and project review

12.1 Sub-section

13 Responsibilites

13.1