TRINITY COLLEGE DUBLIN

MCS DISSERTATION

Narrative Entity & Relation Alignment: Structuralist Analysis In Natural Language Processing

Author:
NEIL HYLAND
Student No. 11511677

Supervisor:
DR. CARL VOGEL

A dissertation submitted in fulfillment of the requirements for the degree: ***Integrated Masters In Computer Science*** *at the*

School Of Computer Science & Statistics

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Trinity Term (May 2016)

Declaration Of Authorship

I, Neil Hyland, declare that the following dissertation ("Narrative Entity & Relation Alignment: Structuralist Analysis In Natural Language Processing") except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signed:

Date:

TRINITY COLLEGE DUBLIN

Abstract

Computer Science Department
School Of Computer Science & Statistics

Integrated Masters In Computer Science

Narrative Entity & Relation Alignment: Structuralist Analysis In Natural Language Processing

by Neil Hyland

"Structuralist Analysis" is a linguistic approach to understanding text, pioneered by the French anthropologist Claude Levi-Strauss in the 1960s. It aims to represent the structure of a text (e.g. a narrative or story) in a formal manner: a table of data points describing major events in the order they occurred (the rows) and their shared themes (the columns).

The research undertaken for this work was to devise and implement a possible set of algorithms or procedures to perform structuralist analysis computationally, with the desired output being matrices describing the structure of narrative Elements (events occurring in the text) and Entities (people/places/things etc...). This was achieved with a four-stage process: where Elements were generated from the text (1), then processed using the K-Means statistical clustering algorithm to approximate the aforementioned thematic grouping (2), Entities were generated from mentions in the text (3), and then all Element and Entity data is combined into an output form of a two-dimensional matrix in the style of Lévi-Strauss' original table approach, and a three-dimensional spatial matrix representing the position and alignment of Entities relative to the Elements they were found in (4).

The findings of this work can be summarised as follows: computationally implementing structuralist analysis is feasible, and with further development could yield a new and different approach to modelling text data for the purposes of comparison, aggregation, summarisation or other use.

Acknowledgements

I would like the give special thanks to my project supervisor: Dr. Carl Vogel, for providing the dissertation topic and supporting my investigation of it.

Contents

List Of Figures

List Of Abbreviations

NLP	**N**atural **L**anguage **P**rocessing
SA	**S**tructuralist **A**nalysis
NER	**N**amed **E**ntity **R**ecogniser (**R**ecognition)
CRA	**C**o-**R**eference **A**nalysis
POS	**P**art-**O**f-**S**peech
API	**A**pplication **P**rogramming **I**nterface
IDE	**I**ntegrated **D**evelopment **E**nvironment
ML	**M**achine **L**earning
UI	**U**ser **I**nterface
GUI	**G**raphical **U**ser **I**nterface

Dedicated to my parents, who saw fit to entertain the prospect of me spending a fifth year in college, chasing a slightly more prestigious qualification than a Bachelors Degree…

Chapter 1

"Summary & Justification"

This chapter outlines the problem domain of "structuralist analysis" pertaining to natural language text processing and the goals of using such an approach when analysing text.

1.1 Introduction

This dissertation intends to explore usage of the linguistic/anthropological technique of *structuralist analysis* (abbreviation: "SA") as a computational model for representing the underlying structure of processed text (i.e. the "narrative" or "story flow" of a piece of text; the arrangement of events and corresponding entities).

This notion (SA) has existed as a form of scholarly analysis of myth, folklore and prose since the 1960s, and has historically been performed by hand to define the resultant structure of a text. Attempting to model some way of performing SA computationally is a non-trivial problem when dealing with the complexity of human language and the myriad of possible forms of narrative structure.

The following sections describe the field of SA in detail, providing a background in the analytical technique: what it is, how it's performed traditionally, what's the use of automating such an approach, and (if SA proves to be feasible computationally) what can such an approach be used for in the wider world of computer science and industry?

It is important to note that – as mentioned above – SA is an open, non-trivial problem. There may not exist a satisfiable solution to performing SA today, but the work this dissertation represents should be considered a step forward in attempting it.

1.2 Problem Domain

The Oxford Dictionary Of Philosophy defines *structuralism* as "the belief that the phenomena of human life are not intelligible unless except through their interrelations. These relations constitute a structure, and behind local variations in the surface phenomena there are constant laws of abstract culture." **[Blackburn, 2008]**

Structuralism as a concept exists in many forms across different academic disciplines, from psychology to sociology and anthropology. The single common facet of *structuralism* across these fields is the focus on understanding the fundamental components of a "structure" – be it social, literary or even biological – and their interrelations. This presents the uninformed reader with a somewhat daunting plethora of possible interpretations/definitions of *structuralism* given by different disciplines.

To avoid confusion, the concept of *structuralism* explored in this dissertation belongs to the field(s) of linguistics and anthropology. That is, *structuralism* refers to understanding the "structure" of human language (and by extension, human culture).

Furthermore, the primary source for this definition of *structuralism* is the French anthropologist Claude Lévi-Strauss, whose seminal work *Structural Anthropology* forms the basis for the practice of "structuralist analysis" (SA) in this dissertation. Lévi-Strauss' techniques for SA will be explored in Section 1.4, and a literature review of his work is presented in the next chapter: Section 2.1.

For all intents and purposes, *structuralism* and its application SA represent the problem domain of this dissertation.

1.3 Structuralism In Linguistics

Linguistic *structuralism* concerns itself with the sentence as its fundamental unit of discourse – as opposed to traditional linguistics which examines the elements that comprise words and their arrangement in said sentences. The idea behind this approach is that, fundamentally, a story (or other text-based work) is a collection of sentences upon which some form of narrative structure is imposed.

"Structuralist analysis" of language is based on linguist Ferdinand de Saussure's study of the underlying system of language (the "langue") as opposed to the use of language (the "parole") **[de Saussure and Riedlinger, 1983]**. The most important idea of de Saussure's work is that words cannot be understood outside of their context. Structuralists then, generally view the aforementioned system of language in terms of sentences, only dipping back into sub-sentence analysis to extract important concepts and entities within the given text.

In this way, *structuralism* is used to examine the context of a body of text (a story/essay/report etc...) and aims to achieve greater understanding through both analysing the structure (the system or "langue", to use de Saussure's earlier notion) of the text, and also perform comparisons with different versions of the same text, perhaps even extending it to compare similarities/differences between wholly different texts.

1.4 Structuralist Analysis

The anthropologist Claude Lévi-Strauss developed the idea of *structuralism* from de Saussre's work, himself even coining the term. He turned his study to the mythologies and folklore of the world, and sought to understand the "common language" of myth, by attempting to compare myths from one or more cultures for similar patterns or recurring concepts. In this way, myth or story is considered as a collective whole, rather than fixating on narrow elements within a single tale.

The linguistic nature of his analysis led him to extend the classical units of language, the original...

- "Phonemes" (simple sounds, e.g. letters).
- "Morphemes" (basic elements of grammar).
- "Lexemes/Sememes" (words).

Were complemented with a new linguistic construct, the...

- "Mytheme" (basic element of myth/story/narrative).

This was conceived by Lévi-Strauss as a derivation from the existing units (*Phonemes, Morphemes, Lexemes*) **[Lévi-Strauss, 1963]** so that now individual grammatical units

compose words, said words compose sentences, said sentences compose *Mythemes*, and these *Mythemes* describe the structure of the narrative.

These *Mythemes* are arranged by sequence and similarity into a matrix structure, ordered horizontally and vertically, respectively, upon which one can see the potential for performing operations over several of such matrices (e.g. comparison, aggregation etc...). Below is an example of the matrix-based/tabular *Mytheme* layout using the story of *Oedipus Rex*...

FIGURE 1.1: *Mytheme* example by Lévi-Strauss.

Cadmos seeks his sister Europa, ravished by Zeus			
		Cadmos kills the dragon	
	The Spartoi kill one another		
			Labdacos (Laios' father) = *lame* (?)
	Oedipus kills his father, Laios		Laios (Oedipus' father) = *left-sided* (?)
		Oedipus kills the Sphinx	
			Oedipus = *swollen-foot* (?)
Oedipus marries his mother, Jocasta			
	Eteocles kills his brother, Polynices		
Antigone buries her brother, Polynices, despite prohibition			

"We thus find ourselves confronted with four vertical columns, each of which includes several relations belonging to the same bundle. Were we to tell the myth, we would disregard the columns and read the rows from left to right and from top to bottom. But if we want to understand the myth, then we will have to disregard one half of the diachronic dimension (top-to-bottom) and read from left-to-right, column after column, each one being considered as a unit." **[Lévi-Strauss, 1963]**

To simplify:

- The output of such a process (SA) can be treated as a table or matrix of narrative elements (in this case *Mythemes*).
- The narrative elements are ordered vertically by order of appearance in the text, and horizontally by general "theme" or category (see in Figure 1.1 the grouping of "killing" events together as an example).
- As can also be seen in Figure 1.1, each row comprises a single element (*Mytheme*). Do however note that the rightmost column is not part of the table/matrix – merely some notes on the text.

The sequential ordering of each row can be generated by iterating over the sentences in order of when they appear during *Mytheme* construction (i.e. the chronological order). The similarity ordering ("thematic" grouping) of the columns is a more complex problem for a computational system however, as in the example given the categories are derived from academic study of the myth.

The main hurdle of the dissertation is how to produce analytical output comparable to a scholar's study of the same corpus (body of text). Lévi-Strauss' examples given in his books are the results of careful – more importantly: human – thinking. To simulate or approximate this study is not a straightforward solution.

1.5 Aims

With Lévi-Strauss' process in mind, the aims of dissertation can be summarised as follows...

- Investigate the possibility of performing SA computationally.
- Develop an approach, process or set of algorithms to enable a practical implementation of SA.
- Attempt to implement the described process using existing language processing tools and other computational techniques.
- Evaluate the attempt(s) made and discuss the potential for future work.

1.6 Potential Uses

Structuralism deals with understanding and reasoning about the underlying structure of text (be it myth/story/essay etc...). Imbuing language processing tools with the ability to compute and reason about the structure of text on a more abstract level not only improves formal analysis, but has a number of more general uses in text aggregation and comparison systems.

There are a number of potential areas that such a novel (to computer science and natural language processing) technique might benefit. At the core of all of these applications is the potential for structural comparison, i.e. algorithms for comparing and contrasting the data generated from SA. Should SA prove to be feasible, it is entirely possible future research could yield such comparison systems. Below are some examples, along with relevant justifications...

RECOMMENDER SYSTEMS:

Using Lévi-Strauss' SA techniques to model the structure of processed text can potentially be used to compute the structural similarity of different texts. Implementing such a system in – say – an e-book marketplace could provide recommendations or suggestions on structurally similar stories, comparing texts not just on tagged metadata like theme or genre.

AGGREGATION SYSTEMS:

Modelling the structure of analysed text(s) can be used in aggregation systems to better identify similarities between texts. Gathering structurally similar articles on a topic for a person who prefers to read text written with a certain (structural) layout is one example application.

SUMMARISATION SYSTEMS:

In a similar manner to aggregation, document summarisation techniques could also benefit from SA-generated data. One may use the structural data as a framework or guide to help generate automated summaries or abridgements of text.

1.7 Report Structure

The remainder of this dissertation write-up is structured as follows: the second chapter provides a brief literature review of the main texts behind *structuralism* and structuralist analysis. The same chapter also details the available natural language processing tools, what is required from them for conducting this dissertation research, how were they compared/contrasted, and a more detailed explanation of the final tool(s) selected.

The third chapter describes in detail the SA approach developed from a theoretical perspective (no real code or implementation details, only high-level processes). The fourth chapter describes relevant parts of the practical implementation of the SA approach from the third chapter.

Finally, the fifth chapter provides some results and discussion thereof. The work done is evaluated against the aims specified in chapter one. Additionally, problems encountered, limitations of the approach/design, and potential future work is explored before concluding.

Chapter 2

"State Of The Art"

This chapter provides an overview of the key texts of *structuralism* pertaining to Lévi-Strauss' analytical process, the requirements initially derived from the problem domain for this dissertation, and the process of examining and selecting suitable tools/libraries to use in satisfying the aforementioned requirements.

2.1 Background Research

In order to acquire a working knowledge of *structuralism* and related linguistic analyses, several major books in the field were sourced. Some to provide the theoretical grounding in *structuralism*, others to aid in developing a practical application of story/narrative analysis. The following section provides a brief background on these texts...

2.1.1 *Structural Anthropology*

Lévi-Strauss' primary text on structuralism, *Structural Anthropology* covers a variety of topics relating to language, culture, art and myth – documenting the perspective of SA ("structuralist analysis") on each of those. The section of this book most relevant to the dissertation project is Part 3: where mythology and religion are subjected to his analyses **[Lévi-Strauss, 1963]**.

The basic framework of his approach (discussed in Section 1.4) is detailed in said part(s) – the *Mytheme*, organising *Mythemes* into matrices, and deriving the structure of the underlying mythic/narrative language from that. Thus this book formed the primary area of background research, as Lévi-Strauss' entire process is described within.

2.1.2 *The Raw & The Cooked*

This book by Lévi-Strauss applies his structuralist ideas to a case study of Amerindian mythology and folklore, mainly the relationship between cultural concepts of food and language **[Lévi-Strauss, 1969]**. In this book he applies de Saussure's earlier techniques of structural linguistics to his analysis of the tribal stories. The study of these myths provide a good example of the process of structuralist analysis (SA).

2.1.3 *Introduction To The Structuralist Analysis Of Narrative*

The post-structuralist Roland Barthes' essay on narrative analysis brings together a number of linguists' work on understanding story and presents a combined process for approaching narrative **[Barthes, 1975]**.

Although Barthes' work dates from a much later period in linguistics (one where *structuralism* has largely been superseded) his retrospective of previous forms of analysis lend additional context to the *structuralism* of Lévi-Strauss.

2.1.4 *Modelling Propp & Lévi-Strauss In A Meta-Symbolic Simulation System*

A paper published by the University Of Wisconsin in 1974 details attempts to transfer some of the ideas of Lévi-Strauss and the linguist Vladimir Propp to an abstract system for performing computation and logical reasoning **[Klein et al., 1974]**.

This paper is the closest previous work to the concept of implementing Lévi-Strauss' analyses in computational systems – though aimed at generating myths from a set of linguistic rules instead of applying linguistic analyses to myths. That being said however, it does provide extra insight into the *structuralism* of Lévi-Strauss and how such analysed myths and legends are reduced to abstract rule sets and basic structural concepts.

2.2 Available Tools

In order to build a system for performing SA, existing natural language processing (abbreviation: "NLP") tools needed to be researched and evaluated, as the information required for performing SA is generated through existing NLP processes.

Name:	*Stanford CoreNLP*	*Apache OpenNLP*	*GATE*	*Apache UIMA*	*NLTK*
License:	GPL	ASF 2.0	LGPL	ASF 2.0	ASF 2.0
Programming Language(s):	Java	Java	Java	Java/C++	Python

Each of these tools/libraries were considered for the project. In the end, CoreNLP was chosen as the best tool to build the analyser system atop (evaluation and final selection is discussed later in Section 2.4).

2.2.1 *Stanford CoreNLP*

The Stanford CoreNLP toolkit is a suite of Java-based linguistic analysis tools designed to be easily configurable with a range of possible functionality. The toolkit primarily operates on plain text – though with additional parser setup XML-formatted data is readable.

"[CoreNLP] can take raw human language text input and give the base forms of words, their parts of speech, whether they are names of companies, people, etc..., normalize and interpret dates, times, and numeric quantities, mark up the structure of sentences in terms of phrases or word dependencies, and indicate which noun phrases refer to the same entities... Its analyses provide the foundational building blocks for higher-level and domain-specific text understanding applications." **[Stanford, 2016]**

CoreNLP is provided as a set of Java libraries (.JAR files) representing the natural language processor along with the models each algorithm was trained with. It's usage is possibly the simplest of all the examined tools: create a pipeline object, configure it with desired tasks, then execute over a set of input data (text). Further details on CoreNLP are provided in Section 2.5.

2.2.2 *Apache OpenNLP*

The Apache NLP toolkit is a competitor to CoreNLP known as OpenNLP, similarly distributed as a Java library. "It supports the most common NLP tasks, such as tokenisation, sentence segmentation, part-of-speech tagging, named entity extraction,

chunking, parsing, and co-reference resolution." **[Apache, 2011–2014]**

OpenNLP execution is slightly more cumbersome than CoreNLP: each task requires a model (algorithm) object to be initialised and passed the input data. This is a far less "out-of-the-box" approach compared to CoreNLP, which executes all desired NLP tasks in one pass, but nonetheless flexible.

2.2.3 *GATE*

GATE ("General Architecture for Text Engineering") is a collective framework of language processing tools available in a number of different forms. GATE is provided as an IDE (integrated development environment), a collaborative web app, a cloud platform, a Java library ("GATE Embedded") and even an abstract model for describing "robust and maintainable services" (i.e. language processing architecture) **[Cunningham et al., 2011]**.

The version of GATE most applicable to the dissertation topic is the GATE Embedded Java library **[GATE, 1995-2015]**, which provides a wider range of features than CoreNLP or OpenNLP. As well as CoreNLP and OpenNLP's functionality, some of the more advanced GATE features include...

- Specialised Data Structure Modelling.
- Visualisation.
- Machine Learning Framework Integration.
- Gazetteer tools (looking-up real-world definitions of places or concepts for additional context during analysis).

GATE actually implements OpenNLP internally, so when considering OpenNLP the decision was made to instead evaluate the combined OpenNLP/GATE suite against the other options.

2.2.4 *Apache UIMA*

The UIMA project ("Unstructured Information Management Architecture") is another open-source offering from Apache for language processing. Unlike CoreNLP or OpenNLP (which are single libraries one can plug into a Java program) UIMA

provides a general framework for the management and execution of unstructured (read: linguistic) information processing **[Apache, 2006–2015]**.

UIMA is divided into three elements: "Frameworks", "Components" and "Infrastructure". *Frameworks* comprise UIMA's Java and C++ language implementation and their interoperability, in addition to the high-level architecture of the UIMA system. *Components* encompass the individual processing tools like annotators or inference systems. Finally, *Infrastructure* gathers together to management systems and server platforms for execution **[Apache, 2008]**.

As UIMA is more a gel for binding existing toolkits together into a single platform, for instance OpenNLP and GATE are compatible with UIMA workflows. Whether or not such an architectural system is useful given the scope of the dissertation project is discussed in Section 2.4.

2.2.5 *NLTK*

NLTK ("Natural Language Tool-Kit") is a Python-based NLP library designed for research and education in linguistics, cognition, artificial intelligence and machine learning "with a suite of text processing libraries for classification, tokenisation, stemming, tagging, parsing, and semantic reasoning, [and] wrappers for [other] industrial-strength NLP libraries." **[Bird, Klein, and Loper, 2009]** NTLK was considered as an alternate to the primarily Java-based CoreNLP and OpenNLP/GATE.

2.3 Expected Requirements For System

The overall goal: to design some form of SA ("structuralist analysis") system within the time constraints of the dissertation period, gave rise to a number of expectations that were then formalised as general requirements.

The choice of library to develop the SA system with introduced the need for design requirements that each potential tool had to be evaluated against. These were not hard, technical criteria – more a set of rules of thumb to avoid spending too much time pursuing the wrong approach or learning the wrong functionality in the chosen library.

Thus, some key areas that needed to be examined were specified:

- Ease-Of-Use (i.e. is the tool easy to configure, execute and retrieve data from?).
- Complexity (i.e. is the tool understandable, is there room for extension or modification?).
- Functionality (i.e. does the tool offer the necessary features that are expected to factor into the process of SA?).

Each of these areas acted as a guiding category for evaluating the third-party libraries mentioned in Section 2.2 (CoreNLP, OpenNLP, NLTK etc...). Whichever tool satisfied enough of each category would be chosen as the base for the practical implementation.

A second consideration arose regarding the specifics of the SA approach: what data would be necessary, what algorithms would be needed, what platform and/or programming language best supported this?

From the onset, it was known that any forms of NLP related to the following information would be necessary:

- Sentence Annotation (splitting sequences of tokens into sentences).
- Part-Of-Speech & Quotation Annotation ("POS", recognising quotes or passages of spoken dialogue).
- Lemma Annotation (generating canonical meanings for each word/token).

- Named-Entity Recognition ("NER", identifying individuals, places, concepts by name in the text).
- Co-Reference Analysis ("CRA", identifying multiple expressions referring to the same "thing").
- Mention Analysis (using the NER or CRA processes to build a list of "mentions" i.e. mapping references to entities).

The data types produced from these procedures comprise the rough set of inputs that a SA system would (conceivably) be required to operate over. Whether any of the third-party tools could supply all the desired data is discussed in the next section (2.4).

Additionally, the programming language used to develop the SA implementation must be considered. Not just a syntactic preference – the availability of extra third-party libraries pertaining to algorithms, data structures or just plain old usability improvements is an important concern. As both Java (for CoreNLP, OpenNLP and GATE) and Python (for NLTK) have plenty of open-source libraries for statistics, machine learning, algorithms and specialised data structures, this doesn't present as big a problem when compared to other, less-rich programming language environments.

As has been mentioned earlier, these initial design requirements are not to be taken as a final, strict set of criteria. They instead should be seen as a loose set of guidelines for choosing the right tool(s) for achieving the aims of this dissertation set out in Section 1.5.

2.4 Tool Selection

Each of the mentioned third-party libraries was compared based on the three given categories in Section 2.3: *Ease-Of-Use*, *Complexity*, and *Functionality*. The decision was made to go with Stanford's offering (CoreNLP) at the end. The reasons for doing so are elaborated in this section, but first: a write-up of the pros and cons of each tool...

2.4.1 *Ease-Of-Use*

ADVANTAGES (CORENLP):

- Easy to setup & configure (simple pipeline options to enable/disable features).
- Information can be retrieved per-document or per-sentence (maps easily to the concept of *Mytheme* generation).

DISDVANTAGES (CORENLP):

- If running with many features enabled, performance and memory consumption is excessive.

ADVANTAGES (OPENNLP/GATE):

- Easy-to-modify standalone classes per parser/analyser structure.
- Most of the same functionality of CoreNLP, with similar concepts behind them.
- GATE provides an IDE to learn about functionality as well as embedded Java library.

DISADVANTAGES (OPENNLP/GATE):

- Cannot easily access base OpenNLP components when using the GATE tools, hampering flexibility.
- Limited documentation for OpenNLP on some of the features listed as an expected requirement (Section 2.3).

ADVANTAGES (NLTK):

- Simple API for performing analyses.
- Python more flexible than Java when programming.

DISADVANTAGES (NLTK):

- Less familiarity with Python brings a greater learning curve to usage.

2.4.2 *Complexity*

ADVANTAGES (CORENLP):

- Lots of parsing & analytical steps performed in a single function call.
- Can be treated as more of a "black box", i.e. provide input, expect output with no need to worry about the internals.
- Consistent, system-wide format for output data.

DISDVANTAGES (CORENLP):

- Any problem encountered with features not easily solvable without deep code inspection & refactoring.
- Somewhat counter-intuitive (at first) method for retrieving data.

ADVANTAGES (OPENNLP/GATE):

- Functionality implemented in a layered system, different analytical processes running atop one another (easy to understand).
- Extensive documentation for GATE processes (not OpenNLP however).

DISADVANTAGES (OPENNLP/GATE):

- Harder to implement some analyses in OpenNLP (requires special configuration to target specific algorithm unless using GATE).
- GATE systems more complex to configure & execute, output data also not uniformly organised.

ADVANTAGES (NLTK):

- Accompanying book for learning the system & providing examples.
- Python API simple to read and try out.

DISADVANTAGES (NLTK):

- Dealing with the Python language's idiosyncrasies may affect implementation if unfamiliar.

2.4.3 *Functionality*

ADVANTAGES (CORENLP):

- Satisfies all of the initial expected requirements (Section 2.3).

DISDVANTAGES (CORENLP):

- Lacks the greater variety of functionality expressed by GATE (although may not be very useful).

ADVANTAGES (OPENNLP/GATE):

- Satisfies most (if not all) of the initial expected requirements.

DISADVANTAGES (OPENNLP/GATE):

- Additional GATE tools unusable as library (e.g. GATE Embedded provides no GUI or visualisation system), reducing its distinctiveness from CoreNLP.
- Have to manually create, configure & execute components for performing different analyses.

ADVANTAGES (NLTK):

- Satisfies most of the initial expected requirements (similar to OpenNLP/GATE).

DISADVANTAGES (NLTK):

- Have to manually create, configure & execute components for performing different analyses (similar problem to OpenNLP/GATE).

2.4.4 *Final Choice*

All three main third-party libraries (CoreNLP, OpenNLP/GATE and NLTK) proved to be able to support many features needed for implementing SA. This meant that personal preference for programming language, method of utilisation and documentation factored into the selection as much as any technical concerns. The toolkit finally chosen was Stanford's CoreNLP suite, as it provided all of the necessary functionality in the form most suitable to the existing skill set of the author.

INTEGRATING UIMA?

The other tool examined: Apache UIMA, was considered as a framework for managing any/all subsystems used to perform SA. It was apparent that in a more formally-designed scenario, using UIMA as a management system for a SA tool would be beneficial. However, as the nature of this dissertation is an open investigation into applying SA computationally, such a formal design could not be conceived of at the onset. As such, UIMA was left out of the selection.

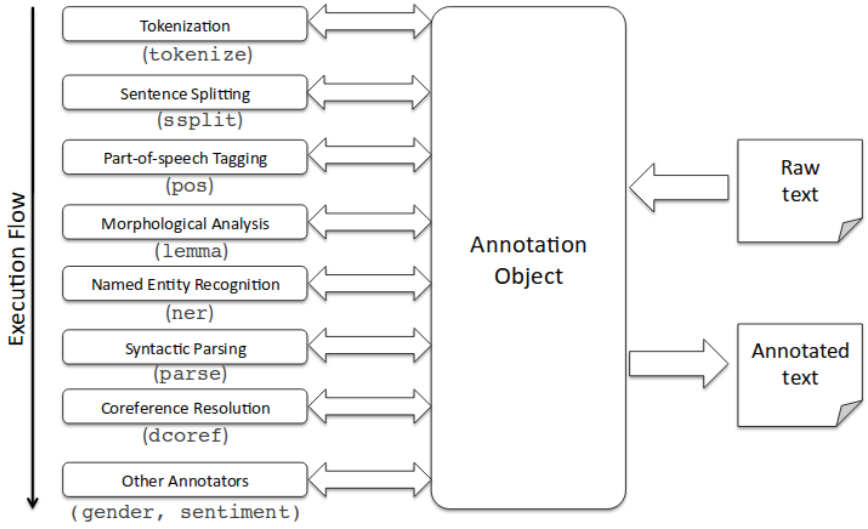
2.5 Stanford CoreNLP (In Detail)

CoreNLP provides a set of specialised text annotators through a pipeline API. That is: a singular object representing the NLP system that is configured once with the desired annotator options, then executed over text data (extracted from file or other sources). Each of these annotators are described below **[Manning et al., 2014]**:

- "tokenize" (divides the text into a sequence of tokens).
- "cleanxml" (removes any XML tags from input text).
- "ssplit" (splits & groups the token sequences into a sequence of sentences).
- "truecase" (determines the correct case information for text).
- "pos" (labels tokens with their "Part-Of-Speech").
- "lemma" (generated the base – canonical – forms of tokens).
- "gender" (adds possible gender flags to names).
- "ner" (recognises named entities in the text, e.g. nouns).
- "regexner" (implements named-entity recognition over tokens with regular expressions).
- "parse" and "depparse" (performs syntactic analysis and dependency parsing).
- "sentiment" (sentiment analysis of text).
- "dcoref" and "hcoref" (implements mention detection and co-reference analysis).

Each of these annotators can be enabled/disabled depending on use case.

FIGURE 2.1: CoreNLP architecture **[Manning et al., 2014]**.



From the available annotators, "Part-Of-Speech" (POS), "Named-Entity Recognition" (NER), "Lemma", Sentence ("ssplit") and "Co-Reference Analysis" (CRA) are the most important to this dissertation.

Chapter 3

"The Approach"

This chapter details the theoretical portion of the dissertation – that is, the algorithmic and procedural approach to creating a SA system using current NLP tools.

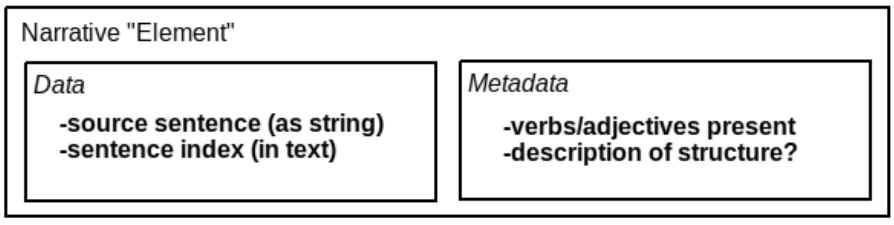
3.1 Overview

In developing this approach to SA, two key concepts were defined, representing the resultant object/data types created by the analysis. The follow subsections explain these concepts, and provide a high-level overview of the SA process...

3.1.1 *Narrative Elements*

An "Element" in this system represents a sentence-level narrative event or event(s). A more low-level analogue to the *Mytheme* of Lévi-Strauss' process – where – instead of a small number of abstracted narrative points, a larger number of these *Elements* are used to represent the story structure.

FIGURE 3.1: Initial idea for narrative *Element*.

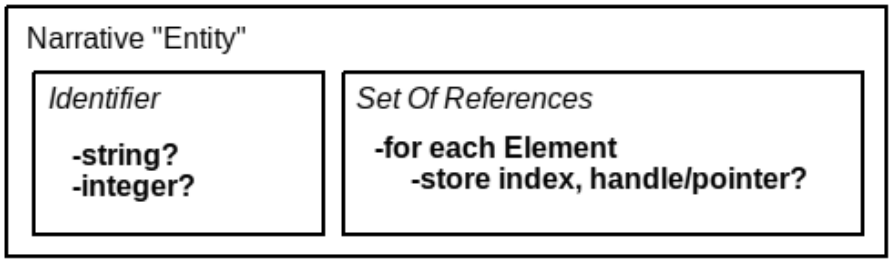


From the onset, it was deemed necessary to reduce in complexity and sophistication the original *Mytheme* concept, with the potential for aggregating them into larger representational objects if the approach proved feasible. Section 4.4 goes into further detail on the make-up of *Elements* and their data.

3.1.2 *Narrative Entities*

An "Entity" (as per the name) represents some person, place or thing involved in the events of the text. *Entities* are the counterpart to narrative *Elements* in the text, as *Entities* are arranged in relation to the *Elements* (read: sentences/events) they are part of.

FIGURE 3.2: Initial idea for narrative *Entity*.



Initially, two NLP techniques were considered to provide input to *Entity* creation: "Named-Entity Recognition" (NER) and "Co-Reference Analysis" (CRA). The pros and cons of both approaches are discussed in Section 3.4, along with implementation details of *Entity* creation in Section 4.4.

3.1.3 *High-Level Process Description*

The approach to Lévi-Strauss' analysis may be broken into several phases...

- "Pre-Process" (apply existing NLP tools to corpus).
- "Analysis" (consume data from *Pre-Process* phase and perform SA – several sub-steps involved).
  - Extract information from NLP output.
  - Build datasets of co-references, mentions or named-entities (narrative *Entities*).
  - Select useful sentences for *Element* creation.
  - Construct narrative *Element* dataset.
  - Build matrix of narrative *Element/Entities* from extracted datasets.
  - Repeat for all corpora.
- "Post-Process" (collect and tidy the information generated in the *Analysis* phase).

As mentioned earlier in Section 1.4, the final output of this process is a multi-dimensional (matrix) representation of the structural elements of the text. This is not the sole possible output however, and other forms of result data are discussed in Section 4.6.

The *Pre-Process* phase primarily consists of feeding input data from an existing tool (CoreNLP) into the *Analysis* and *Post-Process* phases. As such, one can assume the data used as input in the processes described in subsequent sections of this chapter as readily available and ready for analysis.

3.2 Element Extraction

The creation (or "extraction") of narrative *Elements* from the pre-processed corpus is described with the following diagram...

FIGURE 3.3: *Element* extraction process flow.

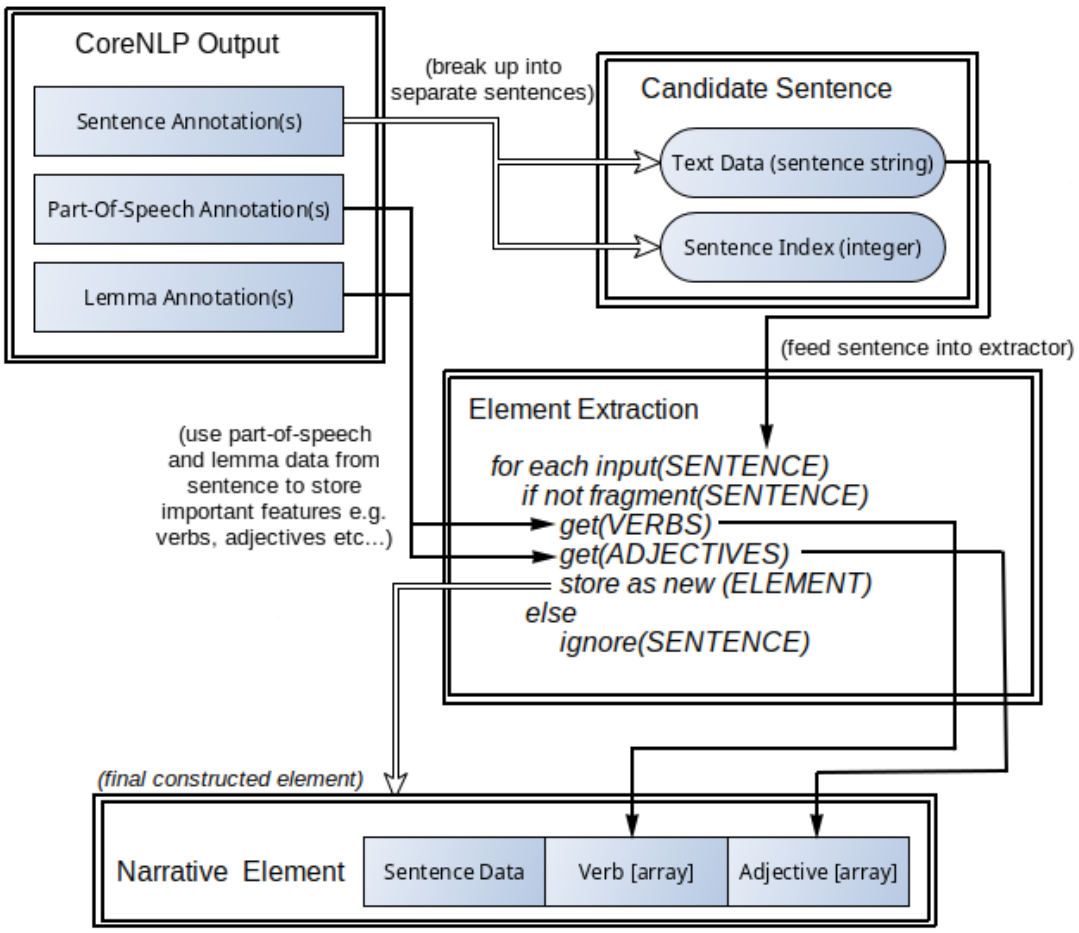
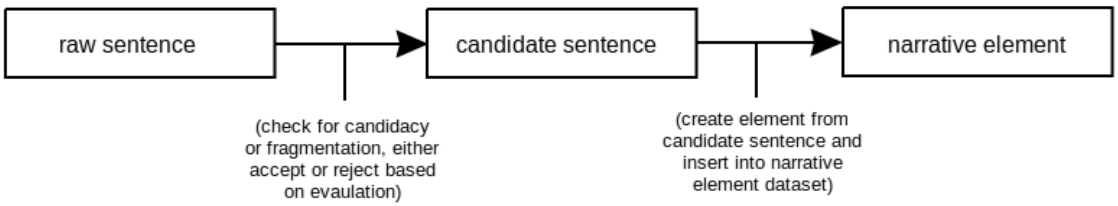


Figure 3.3 shows the high-level approach to extracting text *Elements* from CoreNLP's output annotations. The relevant annotations are:

- *Sentence* (treats the corpus as a set of sentences – not a single body of text – for the purpose of iteration).
- *Part-Of-Speech* ("POS": identifies important words in sentences, namely verbs and adjectives, which indicate the presence of important narrative features).
- *Lemma* (provides the canonical – i.e. tense-less – version of important words identified by the POS annotations).

Elements are constructed from so-called "candidate" sentences, i.e. those which contain possibly important word types (such as verbs or adjectives). A sentence is ignored (or rejected) if it's merely a "fragment", i.e. containing no verbs, adjectives or other indicators of narrative events/actions. The POS annotations **[Toutanova et al., 2003]** are used to check if such words exist or not, and thus determine whether each sentence is a *candidate* or a *fragment*.

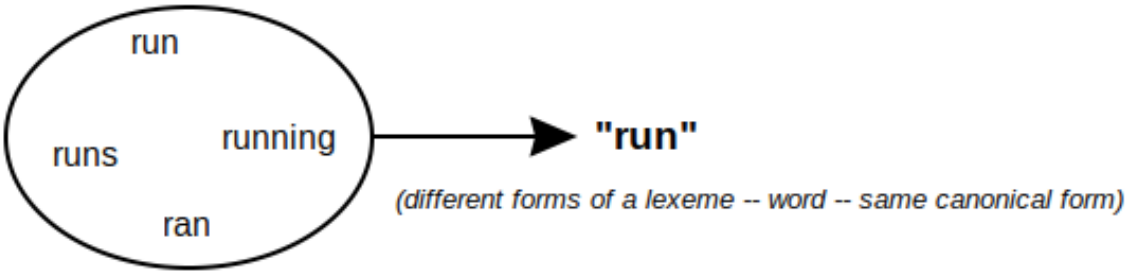
FIGURE 3.4: *Element* creation from sentence (stages).



Once processed, a *candidate* sentence is used to create a new narrative *Element*. The idea behind this hinges on the fact that the structure of a narrative (or similar piece of text) is defined by the events and actions contained within. So *Elements* are created from sentences where action occurs to advance the narrative, better simplified as: where verbs are present, or "where things happen".

The index of the *candidate* sentence used to create an *Element* is also noted, as this is a useful value for ordering the resultant data. As per Lévi-Strauss' description of SA, our lower-level *Mytheme*-analogues (the narrative *Elements*) are laid out in ascending order, mirroring the order of events in the text. The second part of Lévi-Strauss' approach – grouping of mechanically- or thematically-related *Elements*, which accompanies sequential ordering – is explained in Section 3.3.

FIGURE 3.5: Using lemmas to generalise important words.



Finally, it is important to note that the tenses of the "important" words are disregarded. Using the *Lemma* annotation produced by CoreNLP, the dictionary form of each word can be stored, enabling a degree of uniformity across *Element* data. This is used in later steps once *Elements* start to be compared based on verbs (and sometimes adjectives) contained.

3.3 Element Grouping

The grouping (or "clustering") of narrative *Elements* generated from the previous step is described with the following diagram…

FIGURE 3.6: *Element* grouping process flow.

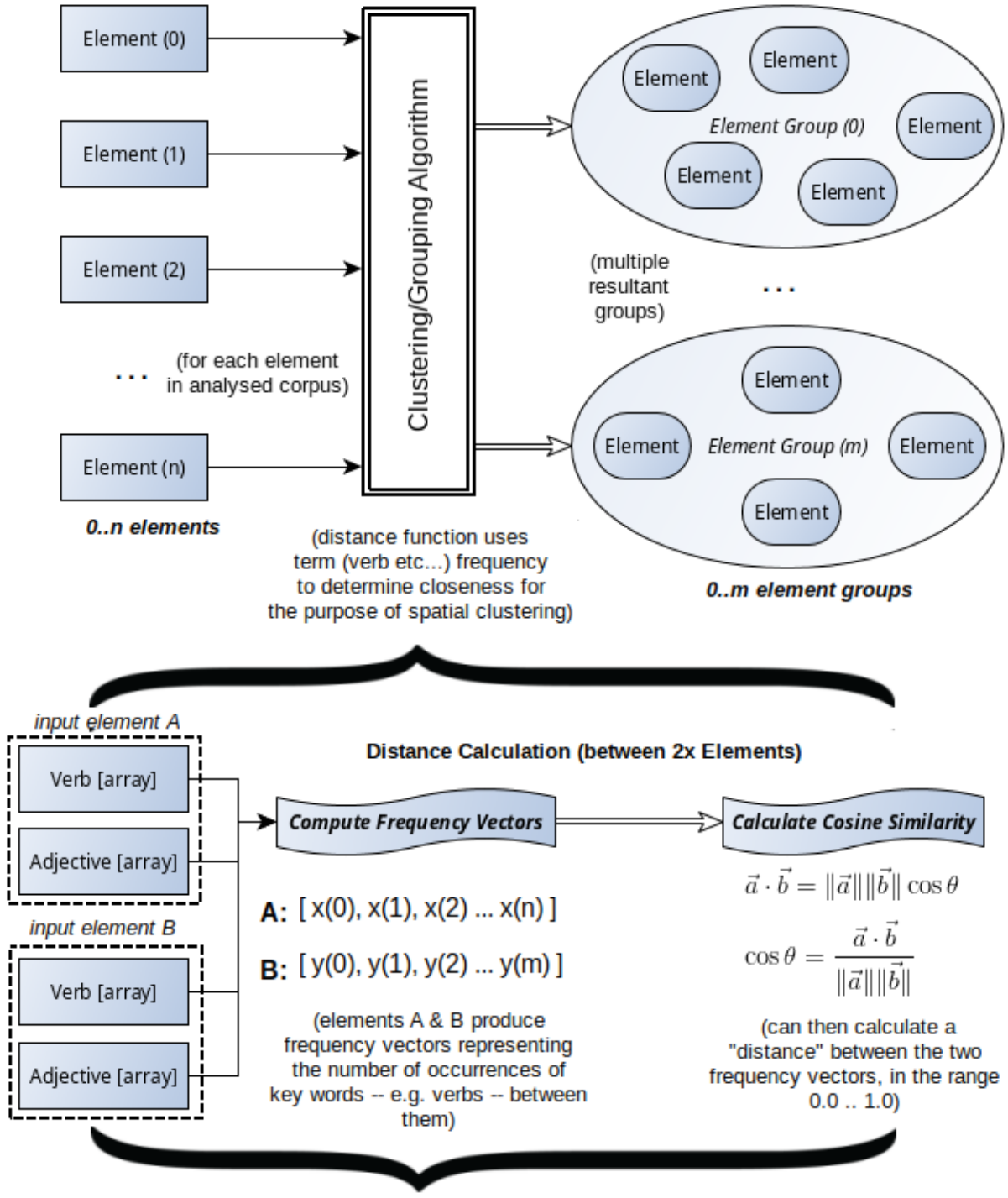


Figure 3.6 shows the use of statistical clustering to generated "groups" of *Elements*. In this case a set of **n** *Elements* are clustered into a superset of **m** "Element Groups". The following sections explain the process in more detail...

3.3.1 *K-Means Clustering With Cosine Similarity*

The process of grouping is achieved through the use of statistical clustering algorithm(s) over the *Element* data. K-Means was chosen as the desired algorithm due to its ubiquity in statistical computing. Applying K-Means clustering to the *Elements* generated from the previous stage creates a rough approximation of the grouping of like *Mythemes* in SA, albeit on a much lower (simpler) level. From these clusters the columns of the resultant matrix/table can be derived.

K-Means **[Macqueen, 1967]** is an unsupervised learning algorithm that takes an input dataset, specified target number of clusters (to assign data points to), and a target number of iterations. A simple example of its execution is presented below:

1. Place **k** points into the space represented by the data point being clustered. These are the initial group centroids.
2. Assign each data point to the group with the closest (shortest calculated distance) centroid.
3. When all data points are assigned to the **k** centroids, recalculate their positions.
4. Repeat steps 2 & 3 until the centroids are no longer moving. This provides a set of partitioned data where each data point is assigned to one (and only one) centroid – from which groups can be labelled.

Typically, K-Means is used to cluster spatial data (points, vectors, etc...). However, it is not restricted to that particular type. Figure 3.6 shows a different approach, where – when two *Elements* are being compared for likeness – the stored verb (and adjective) data is used to create "frequency vectors".

These *frequency vectors* represent the number of occurrences of the aforementioned "important" words (verbs/adjectives) between the two *Element* word sets. Instead of using a Euclidean distance calculation between spatial vectors as the comparison system between *Elements* in K-Means, the cosine similarity between the *frequency vectors* is computed. This allows the clustering of non-spatial data.

"Cosine similarity of two vectors is computed by dividing the dot product of the two vectors by the product of their magnitudes. The cosine of the angle between the vectors ends up being a good indicator of similarity because at the closest the two vectors could be – 0 degrees apart – the cosine function returns its maximum value of 1. It's worth noting that because we are calculating similarities and not distances, the optimisation objective in this function is not to minimize the cost function, or error, but rather to maximize the similarity function." **[Zong, 2013]**

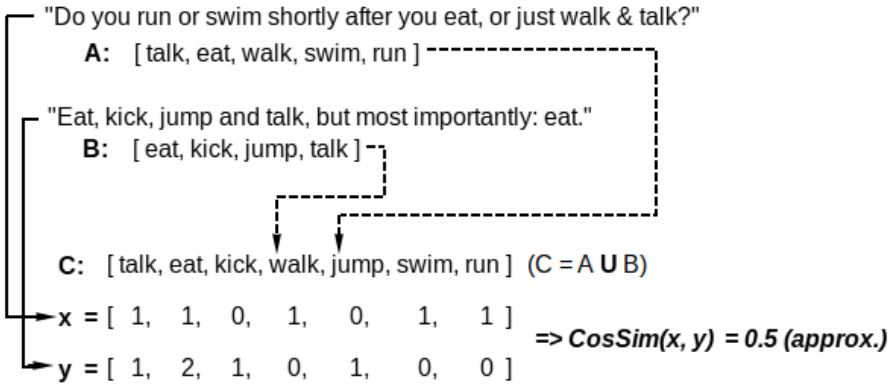
Use of cosine similarity in K-Means has historically been applied alongside *tf-idf* weighting (otherwise known as "term-frequency inverse-document-frequency"), which produces multidimensional data from the entire text for document clustering **[Balabantaray, Sarma, and Jha, 2015]**. The approach detailed in this dissertation discards this notion, instead performing the similarity computation between the *frequency vectors* derived from narrative *Elements* during the K-Means process.

3.3.2 *Computing Similarity With Verb Frequencies*

The first – naïve – approach to similarity calculation takes only the verb sets from the two input *Elements*:

1. Retrieve set of verbs **A** from *Element* **a**, where $\mathbf{A} \neq \emptyset$.
2. Retrieve set of verbs **B** from *Element* **b**, where $\mathbf{B} \neq \emptyset$.
3. Create set **C**, where $\mathbf{C} = \mathbf{A} \cup \mathbf{B}$.
4. **C** must not be $\emptyset$, if it is $\Rightarrow$ return 0 (i.e. no similarity).
5. Create vectors **x** and **y**, where [length(**x**) = length(**y**) = size(**C**)]. These are the so-called *frequency vectors* of the *Element* verb sets.
6. Count frequencies of verbs in set **A**, storing results in corresponding offsets of vector **x**.
7. Count frequencies of verbs in set **B**, storing results in corresponding offsets of vector **y**.
8. Compute cosine similarity between **x** and **y**.
9. Repeat (steps 1-to-8) for all remaining distance calculations in K-Means.

FIGURE 3.7: *Frequency vector* & cosine similarity example.



3.3.3 *Computing Similarity With Weighted Verb/Adjective Frequencies*

An improvement over the initial similarity calculation, this algorithm incorporates the adjective sets from the input *Elements* in addition to the verb sets:

1. Compute cosine similarity between verb *frequency vectors* as per steps 1-to-8 of previous algorithm, i.e. Intermediate result $\mathbf{r_1} = \text{CosSim}(\mathbf{A_x}, \mathbf{B_y})$. Where **x** and **y** are the *frequency vectors* derived from **A** and **B** respectively.

2. Compute the cosine similarity between adjective *frequency vectors* in a similar manner to the previous step, i.e. Intermediate result $\mathbf{r_2} = \text{CosSim}(\mathbf{A_i}, \mathbf{B_j})$. Where **i** and **j** are the *frequency vectors* derived from **A** and **B** respectively.

3. Apply pre-defined weights to intermediate results: $\mathbf{r_1} = \mathbf{r_1} \times \mathbf{w_1}$ and $\mathbf{r_2} = \mathbf{r_2} \times \mathbf{w_2}$. Where $\mathbf{w_1}$ and $\mathbf{w_2}$ are the weights applied to verb and adjective similarities respectively.

4. Sum the resultant similarity results: Result $\mathbf{r} = \mathbf{r_1} + \mathbf{r_2}$.

5. Normalise the result back into the range 0.0 <=> 1.0, in this case: Final Result $\mathbf{R} = \mathbf{r} \div 2.0$.

6. Use this result (**R**) as the signifier of similarity.

Combining weighted similarity values allows one to bring some additional contextual information into the similarity calculation, instead of the raw verb frequencies being the only indicator of similarity.

3.4 Entity (Named/Mention) Extraction

Two approaches were considered for the narrative *Entity* extraction phase, listed below:

- Using Named-Entity Recognition (NER) to identify information for *Entity* creation.
- Using Co-Reference Analysis (CRA) to identify information for *Entity* creation.

Both could potentially satisfy basing the *Entity* creation system upon. In the end however, CRA was chosen as the definitive source of *Entity* data, the reason(s) discussed in the following sections...

3.4.1 *Narrative Entity Creation With Named Entities*

A NER system produces a set of "Named Entities", representing tangible people, places, concepts or (other) things present in the text **[Finkel, Grenager, and Manning, 2005]**. The NER system used by CoreNLP provides such "Named Entities" as a set of names matched with identifiers (describing what type of "thing" it could be, e.g. person, number, city etc...).

Thus it's possible to use the extracted data from the NER system to generate the narrative *Entities* needed by the SA process:

1. Get "name-identifier" object from NER system.
2. Create new *Entity* with the object's name.
3. For all sentences in the text, match the name with the *Entity* to produce a list of locations the *Entity* existed as part of.
4. Repeat steps 1-to-3 for all NER-recognised objects.

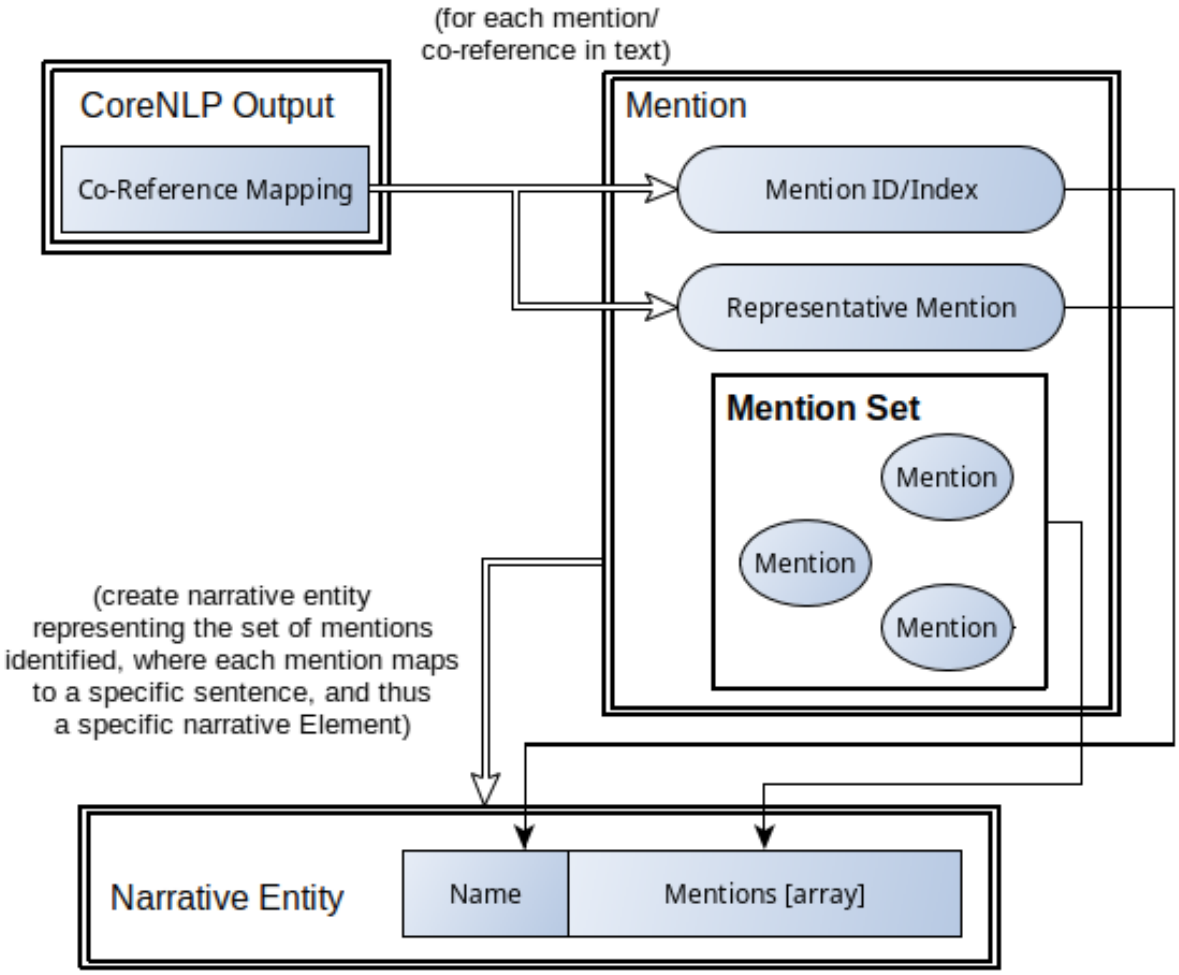
The goal behind finding out where each of the *Entities* are located is to be able to map out the *Element/Entity* data in a multi-dimensional form (see Section 3.5 and Section 3.6).

A problem arises when the text lacks sufficient proper nouns. The NER system fails to recognise otherwise completely valid potential *Entities* in the text. This approach is therefore less capable than a co-referential system.

3.4.2 *Narrative Entity Creation With Co-References (Mentions)*

As an alternative to the use of the NER system, CRA produces a set – not of "Named Entities", but of "Mentions" – representing objects or things referenced in multiple places throughout the text **[Lee et al., 2011]**. As such, CRA is preferable to NER for the SA process.

FIGURE 3.8: *Mention* (co-reference) extraction & *Entity* creation.



Each *Mention* generated by CoreNLP comprises an integer key alongside its internal set of references (or "mentions") to sentences it was found in. Also included is the "representative" mention, i.e. the particular mention that the system reckons best describes the particular object/thing **[Raghunathan et al., 2010]**.

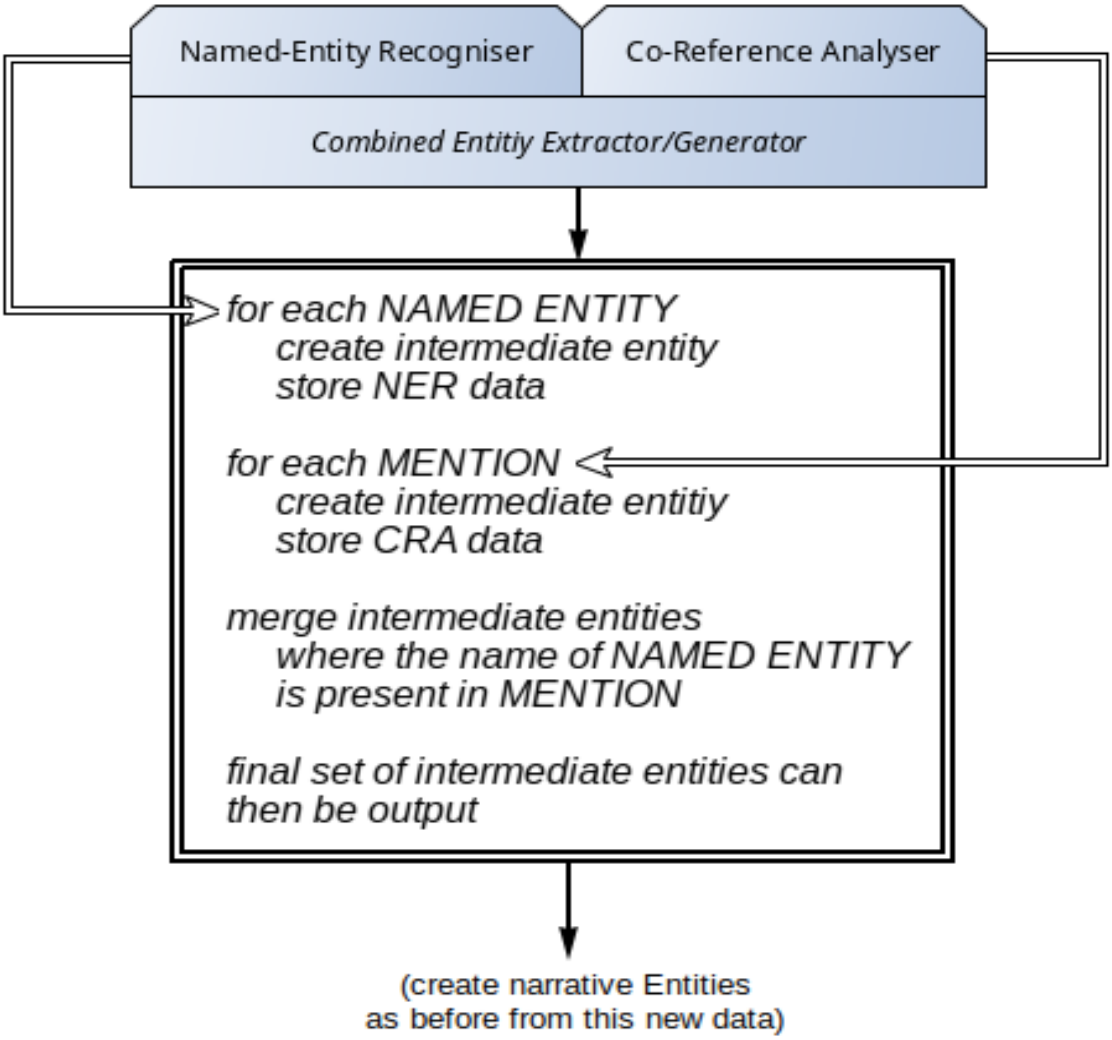
When creating a new narrative *Entity*, either the integer key or representative mention is suitable for uniquely identifying said *Entity*. This approach is superior to using a NER system, but still has some pitfalls in that on occasion mentions will

not be correctly identified or inadvertently duplicated if there exists a significant enough difference in the phrasing.

3.4.3 *Merging The Named Entity/Co-Reference Approaches*

A potential third way explored in the process of evaluating NER and CRA as *Entity* generators is the possibility of combining the two approaches, in an attempt to improve the robustness of the SA system.

FIGURE 3.9: Combined NER and CRA system for creating narrative *Entities*.



Although this combined approach was not implemented for this project, it remains a potential improvement to the SA system. See also Section 5.3 for a discussion of future work.

3.5 Matrix Creation (Post-Processing On The Datasets)

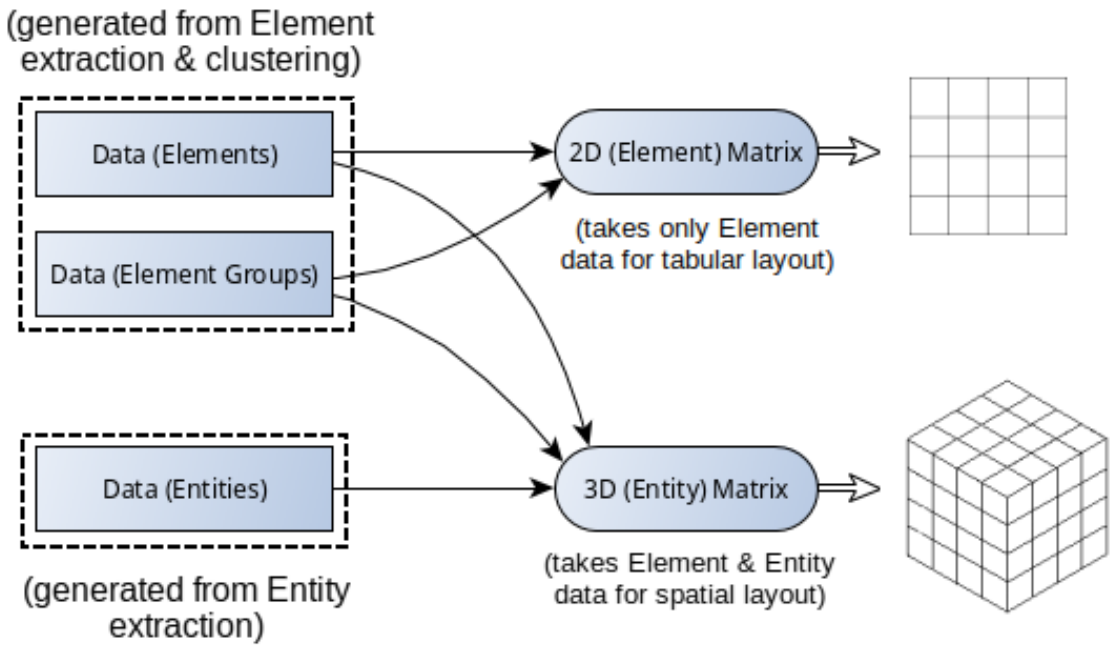
The resultant datasets of *Entities* and grouped *Elements* can be represented with several matrix structures, both mimicking the tabular style of Lévi-Strauss' SA examples, and modelling the structural relations between narrative *Entities* in the analysed text.

Two matrices can be created by passing over the generated *Element/Entity* data:

- 2D "Element Matrix" (lower-level analogue to the *Mytheme* matrix of Lévi-Strauss).
- 3D "Entity Matrix" (spatial representation of the *Entities* present in the text relative to the *Elements* they were found in).

These matrices form the output of the SA system, i.e. the final result data that can then be written to file, or used to build other analytical systems on top of. Section 3.7 discusses the initial groundwork for a structural comparison system using the matrix output.

FIGURE 3.10: Result datasets to matrices.



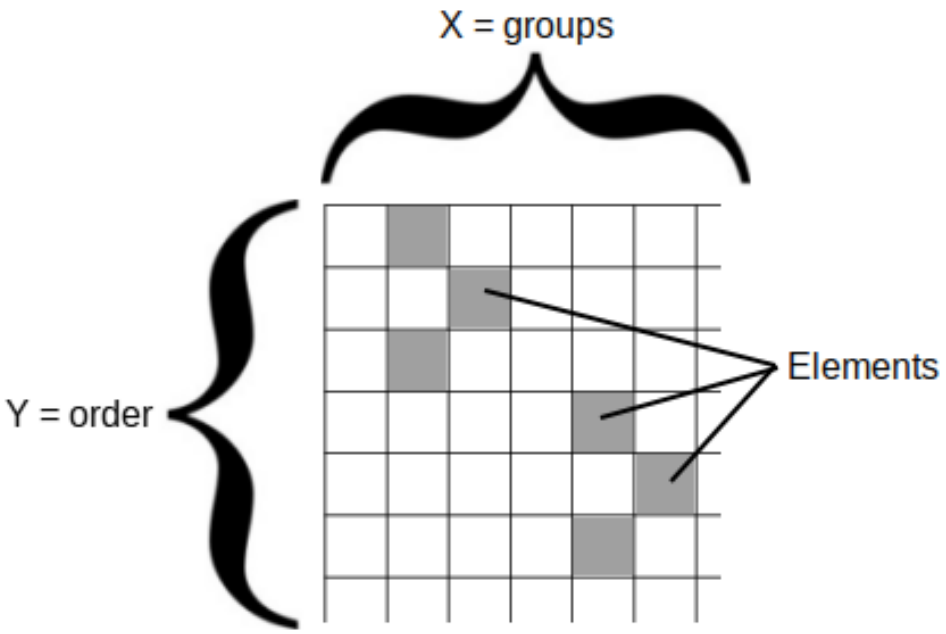
3.6 Matrix Organisation

The organisation of the two result matrices (*Element* and *Entity*) created by the SA system are detailed in this section, with some notes on their structure and data used...

3.6.1 *2D (Element) Matrix*

The 2D *Element Matrix* can be treated as a table for the sake of simplicity. As the narrative *Element* objects are considered a more rudimentary version of the *Mytheme* component of SA, this "table" acts as the rudimentary *Mytheme* matrix, describing the order and structure of events occurring throughout the text(s).

FIGURE 3.11: Mock-up of 2D *Element Matrix*.



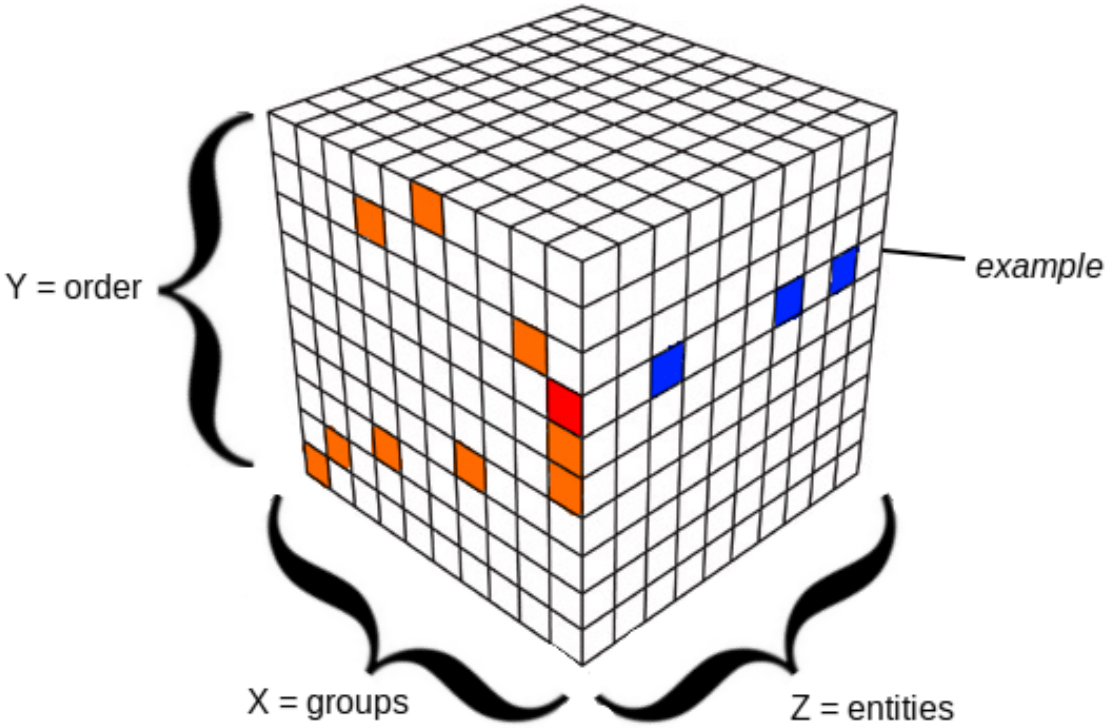
As can be seen in Figure 3.11, the X- and Y-axes mirror the row and column format of Lévi-Strauss' examples – where the columns represent the grouping together of associated *Elements* based on the clustering information generated, and the rows are laid out in ascending order of event position in the text. Each row – in keeping with Lévi-Strauss' approach – contains only one *Element*, and one can re-tell the basic sequence of events in the text by iterating over the matrix, row-by-row.

As there exists only one *Element* per row, this table can be treated as a sparse matrix for convenience of storage.

3.6.2 *3D (Entity) Matrix*

As opposed to the *Element Matrix*, this multidimensional spatial matrix presents the *Entity* information generated by the SA system, a representation not part of the original process devised by Lévi-Strauss, but useful for modelling the presence, relationships and identity between narrative *Entities* and the *Elements* (read: sentences/text events) they're found in.

FIGURE 3.12: Mock-up of 3D *Entity Matrix*.



The above mock visualisation of the *Entity Matrix* (Figure 3.12) shows an example of the way *Entities* are positioned relative to *Elements*. One can see that both the X- and Y-axes are the same as the 2D *Element Matrix*, extended with the Z-axis as indicators of *Entity* presence.

The colour-coding is as follows: red/orange faces mark the *Elements* corresponding to the previous matrix type, while blue faces mark whether or not one or more *Entities* existed in the sentence(s) the *Element(s)* were derived from. Each "column" along the Z-axis matches a distinct *Entity* generated by the SA system, as such this

axis can extend quite far back depending on the size of the text and volume of identified *Entities*.

The 3D row/slice labelled "example" (highlighted with a red *Element* and blue *Entities*) is given as an example of how each *Entity* is either flagged as "present" or "not present" per *Element*. Do note that the position of the *Entity* data points are fixed to the (X, Y) co-ordinates of the *Element* they're associated with, while the Z co-ordinate is (as mentioned earlier) based on the particular *Entity*.

Again, as with the 2D *Element Matrix*, this spatial arrangement is easily reducible to a sparse data structure. Another point to note is the exact nature of the Z-axis *Entitiy* range; as mentioned in Section 3.4, *Entities* can be uniquely identified either by a representative "mention" or integer key roughly denoting at what point the co-reference(s) used to generated the *Entity* were deemed to be co-references by the CRA system. Using this integer key (which usually starts at 0, and increments by 1 per each new co-reference chain extracted) as the Z-axis value is a simple way to convert the *Entity* dataset to part of the *Entity Matrix*.

3.7 Comparison Investigation

One of the potential uses of the result matrices generated from the SA process is for performing comparisons between analysed texts, to ascertain structural similarity or identify differences. Although such a system was not implemented for this dissertation, some initial investigation was conducted into its feasibility, with a couple of possible ideas noted in this section. See also Section 5.3 for a discussion on future work.

A comparison stage can be considered a potential "fifth" phase of the SA system, coming after *Element Extraction*, *Element Clustering*, *Entity Extraction* and *Matrix Creation/Organisation*. During the course of the dissertation research there was a small amount of time devoted to investigating potential approaches for performing structural comparisons between the matrix data, though only the 2D *Element Matrix* was examined as the 3D *Entity Matrix* presented a far more complex problem.

Comparing the *Element Matrices* of two texts presents the following problems:

- No guarantee of common *Elements* or *Element Groups* between the texts.

- If there are sufficiently similar *Element Groups*, the order of the *Element* columns will not be the same.
- Does each column get compared "in a vacuum" or does the data of multiple columns factor into each other for the purpose of comparing them?

These problems were examined and some initial ideas were put forward to resolve them, mainly involving reducing the scope of the comparison system to perform simple scoring between two columns, one from each text's matrices, attempting to reduce the complexity of such a process.

As such, two steps were explored near the end of the dissertation research...

1. "Column Matching" (attempting to identify closely-linked columns, i.e. those containing similar "themes" or important words).
2. "Column Scoring" (attempting to produce a similarity score for compared columns, which could possibly be aggregated across them all for a final result).

To re-iterate: the concept of a structural comparison system was only roughly laid out before time constraints forced it to be left out of the practical implementation (and thus, the main part of this dissertation). The ideas developed so far are mentioned below...

3.7.1 *Column Matching*

For two input matrices (i.e. the *Element Matrices* from the two texts being compared), a cache of all the important words (i.e. verbs, adjectives etc...) is made alongside *Elements* during the extraction phase. This per-column cache is used to compute similarity between columns using a similar approach to the *Element Clustering* distance function: where several *frequency vectors* of important words are passed to a cosine similarity function.

The results of these "column similarity" calculations are then used to order each set of column-to-column mappings descending from highest similarity value to lowest. Picking from this list of column-to-column mappings one can create a rudimentary basis for comparing said columns.

3.7.2 *Column Scoring*

A less-developed idea for the comparison system, the concept of "scoring" each column for structural similarity was briefly considered with the following steps:

1. Using column mappings from previous (*Column Matching*) stage, compare individual columns between the two *Element Matrices*.
2. Ignore other columns an focus on just the structural similarities between those in questions. This is to reduce the complexity of performing such a comparison.
3. For each *Element* in the columns, compute similarity (possibly with existing method of cosine similarity) and add to a running score.
4. Running score is of "positive" (above a certain similarity threshold – representing likeness) and "negative" (below a certain similarity threshold – representing difference) values.
5. Running scores (*positive* and *negative*) are combined in a – as of now unspecified – manner that produces a final score on the similarity between the columns.
6. Repeat steps 2-to-4 for all outstanding column mappings.

The potential algorithm(s) for this stage were still being investigated but no approach was settled on before the end. It therefore remains only a theoretical proposal.

3.8 Evaluation

This chapter has provided an overview of the theory-side of the dissertation, describing the high-level details of the structuralist analysis (SA) approach devised. The next chapter goes into detail on the practical implementation (what hurdles were encountered, how were certain systems actually implemented etc...). How the implementation and its results were evaluated against the theoretical process just described is an important matter to discuss before entering the next chapter.

To summarise, the four key stages of the SA system that were laid out have been implemented (see the upcoming Section 4.4 and Section 4.5):

- *Element Extraction* (3.2).

- *Element Clustering* (3.3).
- *Entity Extraction* (3.4).
- *Matrix Creation/Organisation* (3.5 & 3.6).

The *Comparison System* (3.7) stage and some of the improvements to the *Entity Extraction* stage (3.4) remain in an unfinished state. These incomplete designs have less bearing on the overall satisfaction of the problem of *structuralist analysis* described in the introduction (1.4), therefore they can be placed to one side when considering the success of this initial research into applying the techniques of Lévi-Strauss computationally.

A test case of structurally analysing text is carried out in Section 5.1 of the final chapter, this provides a set of output data (the matrices mentioned earlier) that can be contrasted with the original concept of Lévi-Strauss' *Mytheme Matrix*, to help gauge how close this work got to mirroring the scholarly application of SA.

Chapter 4

"Notes On The Implementation"

This chapter details the practical portion of the dissertation – that is, the programming, development and experimentation conducted during the implementation of a SA system using current NLP tools.

4.1 Architecture

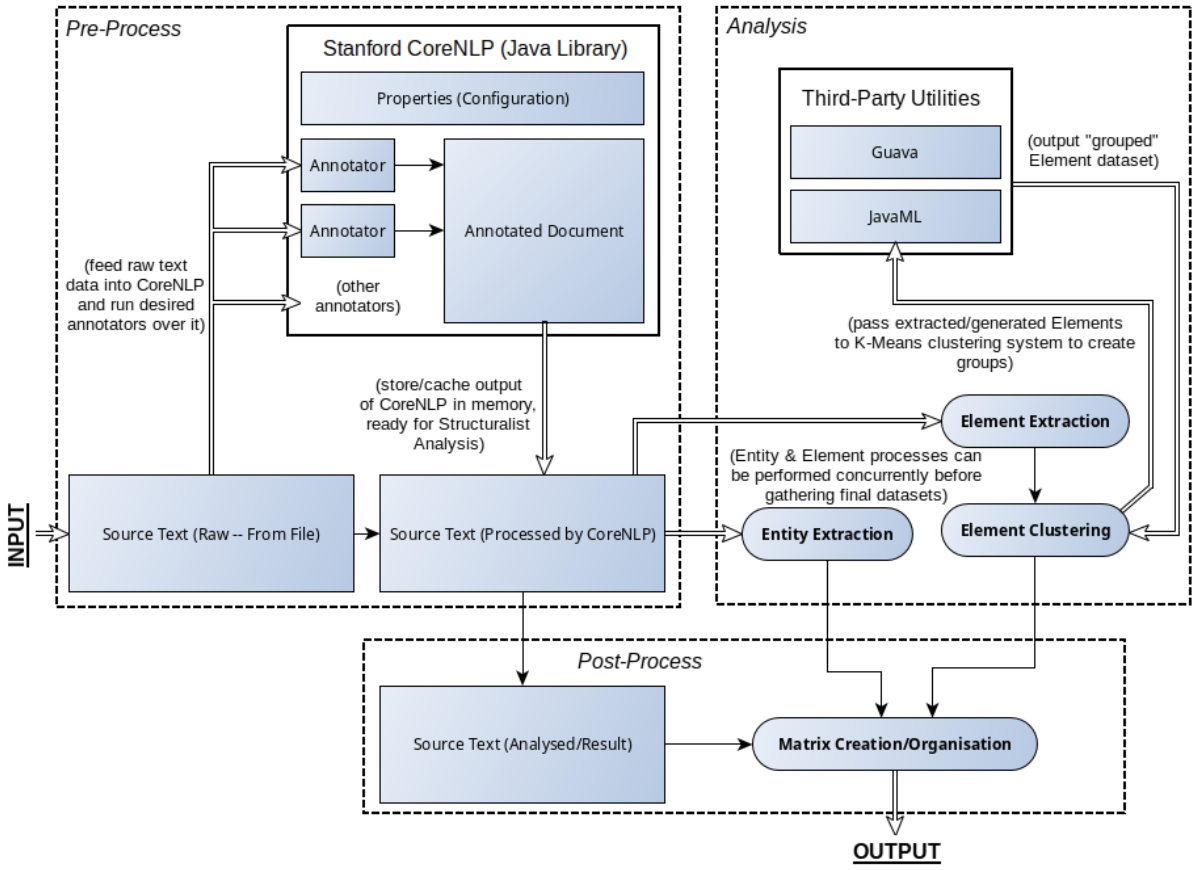
The architecture of the practical implementation roughly mirrors the *Pre-Process*, *Analysis* and *Post-Process* phases described in Section 3.1.

To summarise:

- *Pre-Process*.
  - Text loading.
  - CoreNLP execution over loaded text.
- *Analysis*
  - **Element Extraction**.
  - **Element Clustering**.
  - **Entity Extraction**.
- *Post-Process*
  - **Matrix Creation/Organisation**.
  - Final output (e.g. to file).

Figure 4.1 on the next page shows the overall design of the system, the components of which will be discussed over the remaining sections of this chapter.

FIGURE 4.1: System architecture overview.



From the above diagram we can see the overall flow of data operations in the SA system. Input texts are passed into the *Pre-Process* components and processed by CoreNLP. This intermediate data is then transferred over to the *Analysis* components which perform the actual process of *structuralist analysis* on the data. Finally, the resultant datasets are combined into the output matrices by the *Post-Process* components.

As has been mentioned, the entire system is built using the Java programming language, as CoreNLP and some of the other third-party libraries chosen are Java-based. There are several build systems available for Java that provide easy incorporation of third-party libraries (Maven and Gradle for instance), so combining all of these components was not an issue.

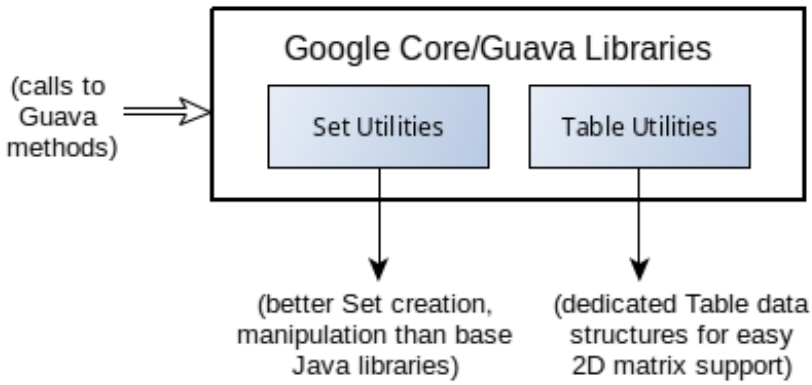
4.2 Utility Framework & Supporting Libraries

To support the execution of CoreNLP and the SA system, a small set of helper tools/classes were developed in Java to provide some user interactivity (e.g. a simple UI), and to glue together the different third-party libraries and their data formats. Two of the main libraries (apart from CoreNLP) used as a part of this were JavaML and Guava.

The JavaML ("Java Machine Learning") library was used to provide the statistical algorithms necessary for the project (clustering in particular). JavaML provides a common set of machine learning algorithms with a degree of customisability (important for this implementation), allowing users to modify and extend its functionality. **[Abeel, Peer, and Saeys, 2009]**. The modified K-Means implementation used for the dissertation project is detailed further in Section 4.5.

Also, to improve upon the current Java standard library, the Google core Java libraries (known as "Guava") were used. Guava provides additional functionality for commonly-used features like collections, concurrency, caching, string processing and more **[Bourrillion and Levy, 2015]**. Guava was used to provide better versions of the standard Java sets, and also provides support for tabular data structures – and their associated functions – instead of just plain multi-dimensional arrays.

FIGURE 4.2: Guava features used in system.



Both of these supporting libraries were incorporated into the components of the *Analysis* phase.

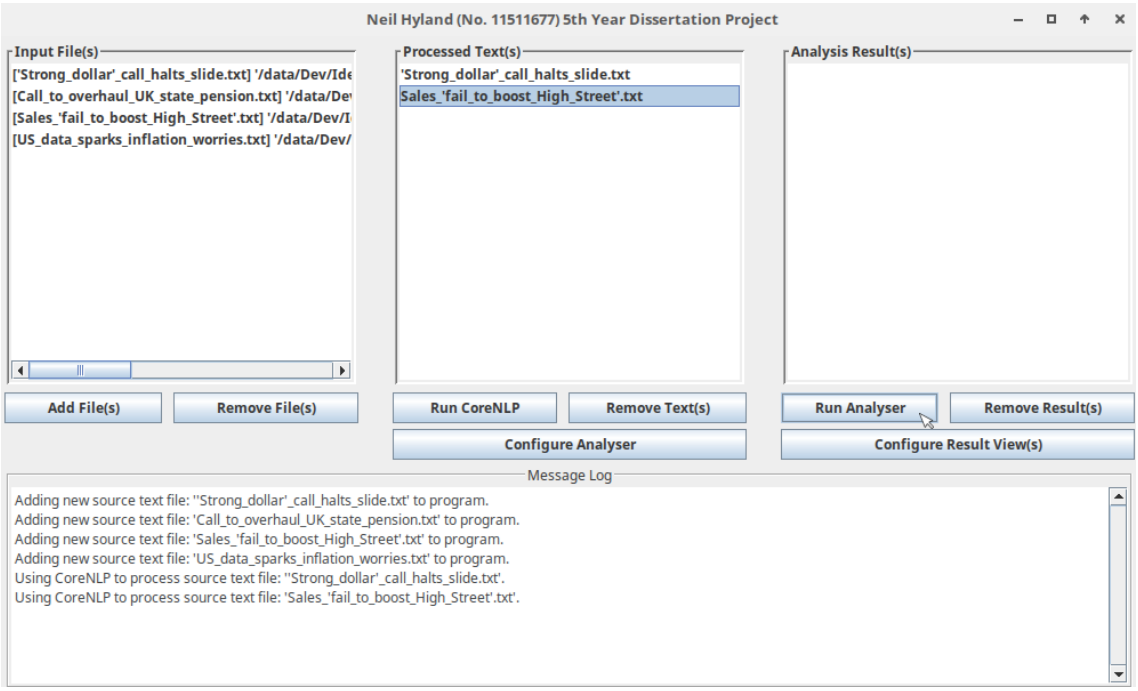
Part of the utility framework to manage the execution of CoreNLP and the SA processes involved the breaking-up of the actual input text data into several (potentially cache-able) objects. The set of corpora (or source texts) was structured as follows...

- "Raw" (input directly from text file(s), not processed or analysed at all).
- "Processed" (i.e. CoreNLP-generated annotations from raw text).
- "Analysed" (data produced by the SA process).

This was primarily to allow the retention of the CoreNLP-generated data for texts already processed, as re-running the entire CoreNLP pipeline on the same texts every time consumed a huge amount of time as CoreNLP executed each of its annotation algorithms over the text – the effect was even worse when using a longer text as input. In this way, one could select the intermediate CoreNLP data for use in different analyses, without wasting time on unnecessary execution.

To support this, a simple graphical user interface (abbreviation: GUI) was implemented in Java using the default Swing toolkit. The threefold split of each text (*Raw/Processed/Analysed*) can be seen in Figure 4.3.

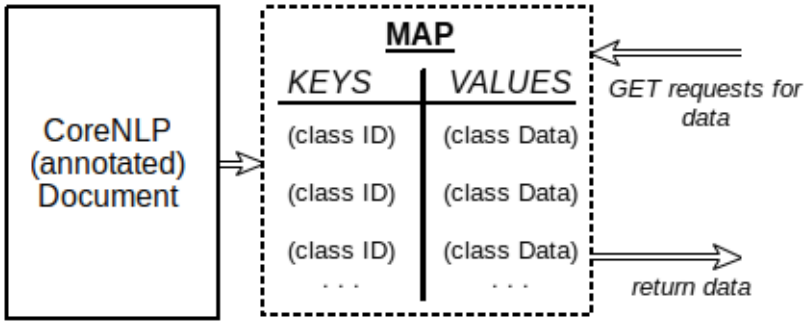
FIGURE 4.3: Java GUI front-end screenshot.



4.3 CoreNLP Data Extraction

The data produced by the CoreNLP annotators needed to be retrieved and handled. As can be seen from Figure 4.4 the annotated text data is returned as a hash map (or similar variant), where the keys are literally the unique definition for the Java class(es) required and the data is the actual class, which can also comprise sub-maps of further key-value pairs.

FIGURE 4.4: CoreNLP data layout.



These annotation classes all present the same interface, allowing the same kind of queries be passed to CoreNLP for all manner of information. Nested objects produced by the CoreNLP maps can be used to order the data from document-wide, sentence-wide to sub-sentence-wide, depending on the nature of the annotation.

Per-sentence and sub-sentence annotated data extraction code is shown below...

```
/*
 * CODE SNIPPET:
 * extractSentenceAnnotations(Annotation corenlp_document)
 */
for (CoreMap sentence : corenlp_document.get(CoreAnnotations.
    SentencesAnnotation.class))
{
    // sentence annotation example:
    int index = sentence.get(CoreAnnotations.SentenceIndexAnnotation.
        class);

    // iterate over tokens (words) example:
    for (CoreLabel token : sentence.get(CoreAnnotations.TokensAnnotation.
        class))
    {
        // part-of-speech example:
```

```
        String pos_tag = token.get(CoreAnnotations.PartOfSpeechAnnotation
            .class);

        if (pos_tag.contains("VB"))
        {
            // is a verb
        }
        else if (pos_tag.contains("AD"))
        {
            // is an adjective
        }
        else
        {
            /* etc... */
        }

        // lemma example:
        String lemma = token.get(CoreAnnotations.LemmaAnnotation.class).
            toLowerCase(Locale.ENGLISH);

        /* do other stuff with individual tokens */
    }

    /* do other stuff with sentence */
}
```

In a similar manner, document-wide information (i.e. that which is gathered across the entire text – such as co-reference/mention data) can be accessed thusly...

```
/*
 * CODE SNIPPET:
 * extractMentions(Annotation corenlp_document)
 */
Map<Integer, CorefChain> coreferences = corenlp_document.get(
    CorefCoreAnnotations.CorefChainAnnotation.class);

for (Map.Entry<Integer, CorefChain> coreference : coreferences.entrySet()
    )
{
    int index = coreference.getKey();
    CorefChain coref = coreference.getValue();

    // or all mentions in co-reference chain:
```

```
    for (CorefChain.CorefMention mention : coref.
        getMentionsInTextualOrder())
    {
        /* do stuff with each mention */
    }

    // get representative mention:
    CorefChain.CorefMention repr_mention = coref.getRepresentativeMention
        ();
    String repr_mention_str = repr_mention.mentionSpan;

    /* do other stuff with co-reference chains */
}
```

4.4 Element, Entity & Mention Extraction

Narrative *Element* Java implementation...

```
/*
 * CODE SNIPPET:
 * Narrative Element (class layout)
 */
public class TextElement
{
    // "important" word lists (possible exemplars for similarity
       comparison):
    private List<String> m_verbs,
                         m_adjectives,
                         m_nouns;

    // original (raw) string from text:
    private String m_original_sentence;

    // index of sentence (in text) for element ordering purposes:
    private int m_sentence_index;

    /* methods: getters, setters etc... */
}
```

Element objects are created at the sentence level, so the example CoreNLP data extraction shown in Section 4.3 can be used to retrieve and store the requisite data:

```
/*
 * CODE SNIPPET:
 * TextElement => createElementFromSentence(CoreMap sentence)
 */
{
    m_verbs = new ArrayList<String>();
    m_adjectives = new ArrayList<String>();
    m_nouns = new ArrayList<String>();

    m_original_str = sentence.get(CoreAnnotations.TextAnnotation.class).
       replace("\n", "␣");

    m_sentence_index = sentence.get(CoreAnnotations.
       SentenceIndexAnnotation.class);

    for (CoreLabel token : sentence.get(CoreAnnotations.TokensAnnotation.
       class))
    {
```

```
            String pos_type = token.get(CoreAnnotations.
                PartOfSpeechAnnotation.class);
            String lemma = token.get(CoreAnnotations.LemmaAnnotation.class).
                toLowerCase(Locale.ENGLISH);

            if (pos_type.contains("NN"))
            {
                m_nouns.add(lemma);
            }
            else if (pos_type.contains("VB"))
            {
                m_verbs.add(lemma);
            }
            else if (pos_type.contains("AD"))
            {
                m_adjectives.add(lemma);
            }
        }
    }
```

Narrative *Entity* Java implementation...

```
/*
 * CODE SNIPPET:
 * Co-Reference/Mention (Entity data handler from CoreNLP)
 */
public class TextMention
{
    // set of sentence indices mentions were found in:
    private Set<Integer> m_sentence_indices;

    // set of sentences covering all possible permutations of the mention
        :
    private Set<String> m_variant_strs;

    // index from CoreNLP mention chain:
    private int m_coref_index;

    // name (i.e. representative mention):
    private String m_name;

    /* methods: getters, setters etc... */
}
```

These *Entity* objects are created by retrieving the requisite information from CoreNLP (as seen in the co-reference extraction example in Section 4.3).

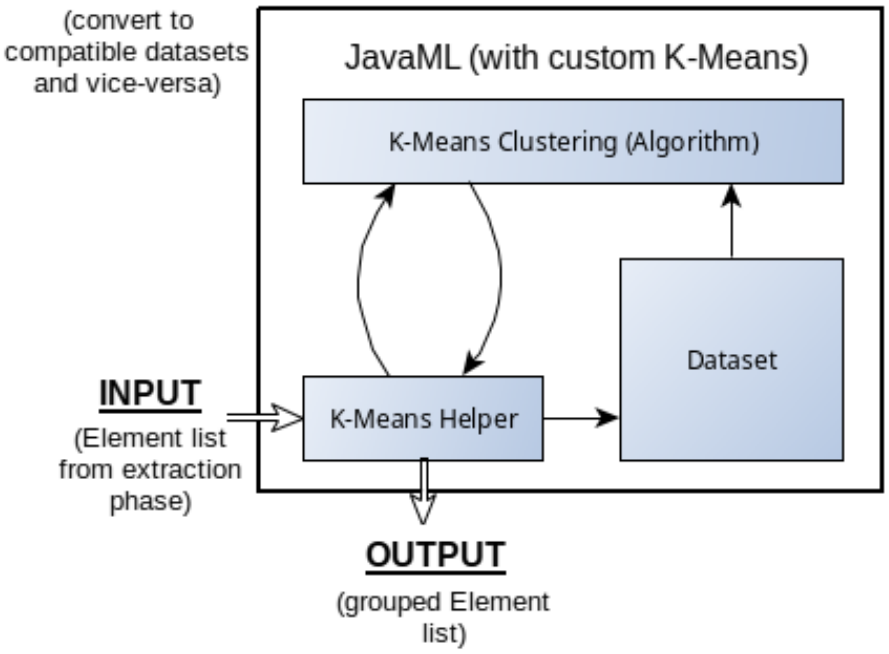
```
/*
 * CODE SNIPPET:
 * TextMention => createMentionFromCoRefChain(int coref_index, CorefChain
     coref)
 */
{
    m_coref_index = coref_index;
    m_sentence_indices = new HashSet<Integer>();
    m_variant_strs = new HashSet<String>();

    for (CorefChain.CorefMention mention : coref.
        getMentionsInTextualOrder())
    {
        // subtract '1' from sentence number to match sentence index in
            other classes:
        m_sentence_indices.add(mention.sentNum - 1);
        m_variant_strs.add(mention.mentionSpan);
    }

    // retrieve & store representative mention:
    CorefChain.CorefMention representative_mention = coref.
        getRepresentativeMention();
    m_name = representative_mention.mentionSpan;
}

...

/*
 * CODE SNIPPET:
 * TextMention => generateFromText(Annotation corenlp_doc)
 *
 * Creates & calls 'createMentionFromCoRefChain()' for each CoreNLP-
     identified co-reference/mention chain.
 */
{
    mentions = new ArrayList<TextMention>();
    Map<Integer, CorefChain> coreferences = corenlp_doc.get(
        CorefCoreAnnotations.CorefChainAnnotation.class);

    for (Map.Entry<Integer, CorefChain> coreference : coreferences.
        entrySet())
    {
        mentions.add(new TextMention(coreference.getKey(), coreference.
            getValue()));
```

```
    }

    /* store mention list as member data */
}
```

4.5 Clustering

The K-Means algorithms provided by JavaML allows users to customise the distance and comparison functions by inheriting from a certain class. This allows the *frequency vector* cosine similarity computation to replace the traditional spatial approach.

FIGURE 4.5: K-Means implementation.




```
/**
 * CODE SNIPPET:
 *
 * Helper class for KMeans clustering of TextElements.
 * Uses cosine similarity for distance measuring in algorithm.
 */
public class KMeansHelper implements DistanceMeasure
{
    // Constant member data:
    private final double m_min_val,
```

```
                    m_max_val;

// List of narrative elements from
private List<TextElement> m_elements;

// Configuration settings:
private boolean m_use_adjectives;
private double m_verb_weight,
               m_adjective_weight;

/**
 * Utility function to create a JavaML-compatible dataset for running
     the KMeans algorithm.
 * JavaML instances correspond to the list index of each TextElement,
     which is resolved in
 * the distance measurement method.
 *
 * @return JavaML dataset of instances corresponding to TextElements.
 */
public Dataset createCompatibleData()
{
    if (m_elements != null)
    {
        Dataset data = new DefaultDataset();

        for (int i = 0; i < m_elements.size(); ++i)
        {
            double[] new_attrs = new double[1];
            new_attrs[0] = (double)i;

            Instance new_inst = new DenseInstance(new_attrs);
            data.add(new_inst);
        }

        return data;
    }
    else
    {
        System.err.println("Cannot create JavaML dataset, no
            TextElements to process!");
        return null;
    }
}
```

```
/*
 * Method to convert list of datasets (each representing a cluster)
    to TextElement arrays/lists.
 */
public List<TextElementGroup> convertClustersToGroups(Dataset[]
   kmeans_data)
{
    List<TextElementGroup> groups = new ArrayList<TextElementGroup>()
       ;
    int cluster_no = 0;

    for (Dataset data : kmeans_data)
    {
        TextElementGroup new_group = new TextElementGroup(cluster_no)
           ;

        for (Instance inst : data)
        {
            int element_index = (int)inst.value(0);
            new_group.addTextElement(m_elements.get(element_index));
        }

        groups.add(new_group);
        ++cluster_no;
    }

    return groups;
}

/**
 * Custom distance function.
 */
public double measure(Instance x, Instance y)
{
    if (x.noAttributes() != y.noAttributes() ||
        x.noAttributes() != 1 || y.noAttributes() != 1)
    {
        System.err.println("Cannot␣compute␣similarity,␣differently-
           sized␣instances␣passed␣to␣function!");
        return 0.0;
    }

    // Get elements from instances:
    TextElement elem_x = m_elements.get((int)x.value(0)),
```

```
                elem_y = m_elements.get((int)y.value(0));

        double result = 0.0;

        // Perform similarity computation:
        if (m_use_adjectives)
        {
            result = CosineSimilarity.computePairOfLists(elem_x.
                getVerbList(),
                                                         elem_y.
                                                             getVerbList()
                                                             ,
                                                         elem_x.
                                                             getAdjectiveList
                                                             (),
                                                         elem_y.
                                                             getAdjectiveList
                                                             (),
                                                         m_verb_weight,
                                                         m_adjective_weight
                                                             );
        }
        else
        {
            result = CosineSimilarity.computeLists(elem_x.getVerbList(),
                                                   elem_y.getVerbList());
        }

        return result;
    }

    /**
     * Comparison between computed similarity values.
     */
    public boolean compare(double x, double y)
    {
        // Highest similarity value is '1', so first element should be
            larger than second:
        return (Double.compare(x, y) > 0);
    }

    /* other member methods: getters, setters etc... */
}
```

The cosine similarity calculations themselves are performed by a utility class shown below...

```
/**
 * CODE SNIPPET:
 *
 * Utility class for calculating cosine similarities.
 */
public class CosineSimilarity
{
    /**
     * Computes the cosine similarity between two vectors/arrays of
        floating-point values.
     *
     * NOTE: If lengths are different, does not compute (returns '0').
     *
     * @param vec1 First input vector/array.
     * @param vec2 Second input vector/array.
     *
     * @return Value of cosine similarity.
     */
    public static double compute(final double[] vec1, final double[] vec2
        )
    {
        if (vec1.length != vec2.length)
        {
            return 0.0;
        }

        double dot_prod = 0.0,
               norm1 = 0.0,
               norm2 = 0.0;

        for (int i = 0; i < vec1.length; ++i)
        {
            dot_prod += vec1[i] * vec2[i];
            norm1 += Math.pow(vec1[i], 2);
            norm2 += Math.pow(vec2[i], 2);
        }

        return (dot_prod / (Math.sqrt(norm1) * Math.sqrt(norm2)));
    }

    /**
```

```
     * Compute the cosine similarity between two string lists (i.e.
        finding matching words).
     *
     * NOTE: If lengths are different, does not compute (returns '0').
     *
     * @param str1 First input list.
     * @param str2 Second input list.
     *
     * @return Value of cosine similarity.
     */
    public static double computeLists(final List<String> str1, final List
       <String> str2)
    {
        // Guava set creation utilities:
        Set all_strs = new ImmutableSet.Builder<String>().addAll(str1)
                                                         .addAll(str2).
                                                            build();

        double[] count1 = new double[all_strs.size()],
                 count2 = new double[all_strs.size()];

        int i = 0;

        for (Object str : all_strs)
        {
            count1[i] = (double)Collections.frequency(str1, str);
            count2[i] = (double)Collections.frequency(str2, str);

            ++i;
        }

        return compute(count1, count2);
    }

    /**
     * Compute the cosine similarity between two pairs of string lists
        with weights.
     *
     * NOTE (1): If lengths are different, does not compute (returns '0')
        .
     *
     * NOTE (2): Weight values are clamped in the range ('0.0' => '1.0').
     *
     * @param str1a First input list of first pair.
```

```
     * @param str1b First input list of second pair.
     * @param str2a Second input list of first pair.
     * @param str2b Second input list of second pair.
     * @param weight1 Value to modify match count of first pair of input
       lists.
     * @param weight2 Value to modify match count of second pair of input
       lists.
     *
     * @return Value of cosine similarity.
     */
    public static double computePairOfLists(final List<String> str1a,
                                            final List<String> str1b,
                                            final List<String> str2a,
                                            final List<String> str2b,
                                            final double weight1,
                                            final double weight2)
    {
        double similarity1 = computeLists(str1a, str1b),
               similarity2 = computeLists(str2a, str2b),
               final_similarity = 0.0,
               weight1 = Math.max(0.0, Math.min(1.0, weight1)),
               weight2 = Math.max(0.0, Math.min(1.0, weight2));

        final_similarity += (similarity1 * weight1);
        final_similarity += (similarity2 * weight2);
        final_similarity /= 2.0;

        return final_similarity;
    }
}
```

Below is an example of the setup & execution of the K-Means clustering algorithm for grouping narrative *Elements*...

```
/*
 * CODE SNIPPET:
 * Example of TextElement clustering using customised K-Means.
 */
{
    List<TextElement> text_elements = // <= elements generated from
       CoreNLP document annotations

    KMeansHelper kmeans_helper = new KMeansHelper(text_elements, false);
```

```
    // K-Means with 10 target clusters, 100 iterations, and the custom
       distance utility class:
    KMeans kmeans_algo = new KMeans(10, 100, kmeans_helper);

    Dataset[] kmeans_result = kmeans_algo.cluster(kmeans_helper.
       createCompatibleData());
    List<TextElementGroup> text_element_groups = kmeans_helper.
       convertClustersToGroups(kmeans_result);

    /* do stuff with grouped elements */
}
```

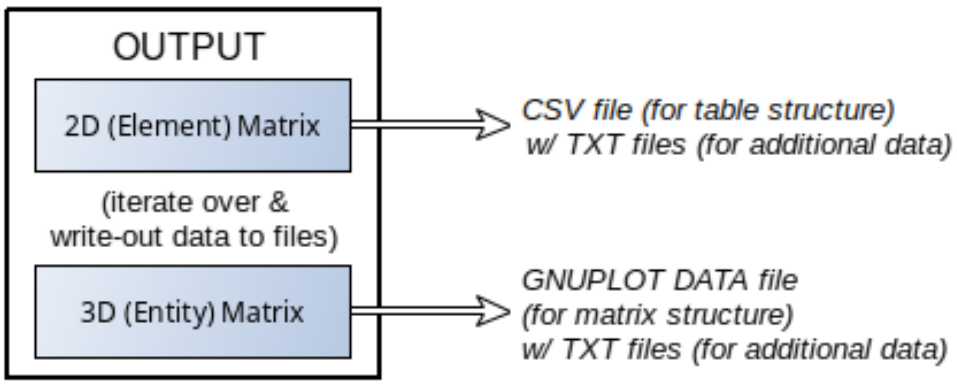
Given both the list of grouped *Elements* and *Entities*, one can then re-iterate over them and produce the 2D and 3D output matrices. The following section elaborates on the output process.

4.6 Matrices & Output/Presentation

The output from the *Entity Extraction* and *Element Clustering* stages of the SA system can be transformed into matrices simply enough by iterating over the generated data. For *Element Matrices* this is using the index of the group each *Element* is assigned to against the sentence index of the *Element* (data produced by the stages outline in Section 4.4 and Section 4.5).

For *Entity Matrices*, the same indices for *Element* groups and sentence order is retained, with the remaining axis derived from the set/list of *Entities* identified – where the maximum Z-value is the size of the set, and each point along the Z-axis indicates the presence of a specific *Entity*.

FIGURE 4.6: Output of matrix data to file(s).



Serialising these matrices to file is done into either a .CSV file (for *Element Matrices*) or a GNUplot .DAT file (for *Entity Matrices*). Both of these files serialise only the indices of their respective data, where the remaining information corresponding to each *Element* or *Entity* can be written out to other secondary files (.XML or .TXT for example).

The reason for this is that specialised data formates like .CSV files cannot easily store all of the requisite object data in a neat way. So – in order to write out as much data as possible – extra files in a less-rigid format are used.

For the purposes of visualisation, *Element Matrix* .CSV files can be loaded into any spreadsheet program (Microsoft Excel, LibreOffice Calc etc...) and viewed from there. For the *Entity Matrix* however, GNUplot-compatible files were output in order to create 3D visualisations of the data. GNUplot is a cross-platform command-line tool for performing data visualisation **[Williams and Kelley, 2015]**. It was used in this implementation due to the ease of reading in plain text-formatted numeric data as input.

Section 5.1 shows examples of such visualisations and some of the raw data used.

4.7 System Review

A number of important implementation details were laid out in this chapter, including code snippets from the system for key pieces of functionality. This implementation covered the four phases of the SA process defined in the third chapter, and overall each of these areas were implemented relatively successfully (as per the stated design goals at least). A quick summary and review of the implementation of the key phases makes up the remainder of this section...

ELEMENT EXTRACTION & CLUSTERING:

The narrative *Elements* – as mentioned throughout this write-up – are a lower-level version of the *Mytheme* as defined by Lévi-Strauss. Thus the major deviation from the academic approach to SA is the greater number of sentence-based *Elements* representing the event structure of the text.

The other area under consideration is the nature of the clustering achieved using statistical methods, which – while allowing a general, content-agnostic grouping of data – does not exactly match the scholarly creation of thematic categories. As such the exists a difficulty in examining common aspects of multiple texts due to the unpredictable types of the resultant groups.

Overall however, the implementation does present a useful process for generating structural data from text – even when looking at how well Lévi-Strauss' techniques were or weren't performed. The code snippets seen in Section 4.4 show us that it's not too complex a problem to expand on, some ideas on which are described in the next chapter.

ENTITY EXTRACTION:

As the narrative *Entity* concept is an extension to the original techniques espoused by Lévi-Strauss, its harder for us to judge its suitability as part of implementing SA. That being said, the use of *Entity* data to extend the dimensions of the output matrices into a model of *Entity/Element* "relations" provides additional data to use for future work.

MATRIX CREATION/ORGANISATION:

As mentioned earlier, because the *Elements* constitute a modified "simpler" version of the *Mythemes*, the *Element Matrix* acts as a "simpler" version of the *Mytheme* matrix/table. This is not particularly limiting as again, further improvements to *Element Extraction* could allow for more complex data structures covering a wider range of information from the text. Eventually bringing the computational process of SA closer to its manual academic equivalent.

The *Entity Matrix* meanwhile, acts as an extension to the *Element Matrix* for aligning narrative *Entities*. As it's a new conception not previously laid out by Lévi-Strauss there isn't much to evaluate against the original aims.

Chapter 5

"Outcome & Reflection"

This chapter wraps up the work carried out over the dissertation and discusses the resultant system(s) implemented, the limitations of said system(s), as well as any outstanding investigations and possible future improvements to implementing SA.

5.1 Results

This section displays a sample use case of SA data output, using one of the Grimm's fairytales "The Wolf & The Fox" as input (in keeping with the myth/folklore theme of the Lévi-Strauss' original works). The text for this is public domain and can be sourced from: http://www.sacred-texts.com/neu/grimm/ht28.htm. Other example datasets are listed in the Appendices (A and B).

FIGURE 5.1: Raw *Element* data examples.

```
Element [Sentence: 3]
'That suited the wolf, and they went thither, and the fox stole the little lamb,
took it to the wolf, and went away.'
[suit, go, steal, take, go]
[]
[wolf, fox, lamb, wolf]

Element [Sentence: 5]
'As, however, he did it so awkwardly, the mother of the little lamb heard him,
and began to cry out terribly, and to bleat so that the farmer came running there.'
[do, hear, begin, cry, bleat, come, run]
[]
[mother, lamb, farmer]

Element [Sentence: 20]
'Said the wolf, "I will go when thou dost, that thou mayest help me if I
am not able to get away."'
[say, go, help, be, get]
[]
[wolf]

Element [Sentence: 30]
'The wolf wanted to follow him, but he had made himself so fat with eating
that he could no longer get through, but stuck fast.'
[want, follow, have, make, eat, get, stick]
[]
[wolf]
```

Figure 5.1 provides a look at some of the raw data generated for *Element* objects in the system. Shown is the original source sentence the *Elements* are made from, and a couple of "important" word sets identified per-*Element*. Each of the verbs extracted are shown as their lemmas (canonical forms), and one can see the usefulness of lemma annotation – e.g. *Element* 20 in Figure 5.1 has archaic English terms being correctly reduced to lemmas compatible with the more modern terms in the rest of the text.

FIGURE 5.2: 2D *Element Matrix* example (indices only).

Group (0)	Group (1)	Group (2)	Group (3)	Group (4)
	0			
	1			
	2			
	3			
	4			
5				
	6			
7				
				8
	9			
	10			
			11	
	12			
13				
14				
15				
16				
				17
	18			
	19			
	20			
		21		
	22			
			23	
	24			
				25
	26			
	27			
28				
	29			
30				
	31			

Figure 5.2 shows the 2D *Element Matrix* represented as a table of *Element* indices.

This was generated by configuring the K-Means clustering algorithm with 5 target clusters as the desired number of result *Element* groups.

This table was created from the .CSV output from the SA system, cleaned-up in a spreadsheet program – adding some colouring to help highlight the *Element* groupings. The table's similarity to the structure of the *Mytheme* matrix example in Section 1.4 can be considered a rough approximation, as mentioned before the narrative *Elements* constitute a lower-level data structure than Lévi-Strauss' *Mythemes*.

FIGURE 5.3: 2D *Element Matrix* example (verbs only).

Group (0)	Group (1)	Group (2)	Group (3)	Group (4)
"have?" "say?" "want?" "hear?"	"go?" "be?" "eat?" "get?"	"show?" "reach?"	"slip?"	"reply?"
	have/wish/be/compel/do/be/have/be			
	chance/be/go/say/get/eat/eat			
	answer/know/be/fetch			
	suit/go/steal/take/go			
	devour/be/satisfy/want/go/get			
do/hear/begin/cry/bleat/come/run				
	find/beat/go/limp/howl			
have/mislead/say/want/fetch/surprise/have/beat				
				reply
	go/say/get/eat/eat			
	answer/know/be/bake/get			
			go/slip/peep/sniff/discover/be/draw/carry	
	be/eat/say/go			
swallow/say/make/want/go/tear/break				
make/come/see/call/hurry/beat/hold/howl/get				
hast/mislead				
cry/catch/tan				
				reply
	be/say/get/eat/eat			
	answer/know/have/be/kill/be/lie/get			
	say/go/help/be/get			
		be/say/show/reach		
	be/attack/think/be/need/leave			
			like/look/run/have/come/try/be/slip	
	say/tell/run/jump			
				see/be/come/reply
	do/eat			
	say/leave/be			
have/hear/jump/come				
	see/be/bind			
want/follow/have/make/eat/get/stick				
	come/strike/bound/be			

This table (Figure 5.3) shows the same *Elements* as Figure 5.2, but the data displayed are the sets of verbs used to cluster the *Elements*. The topmost – grey – row (after the group names) lists the prevalent verbs, i.e. the best indicators of commonality between the *Elements* of the same group.

One can see in both instances that the *Element* clustering is lopsided (i.e. groups 1 and 2 have the majority of the narrative *Elements*) due to over-abundance of simple verbs like "be". This is discussed further in Section 5.2.

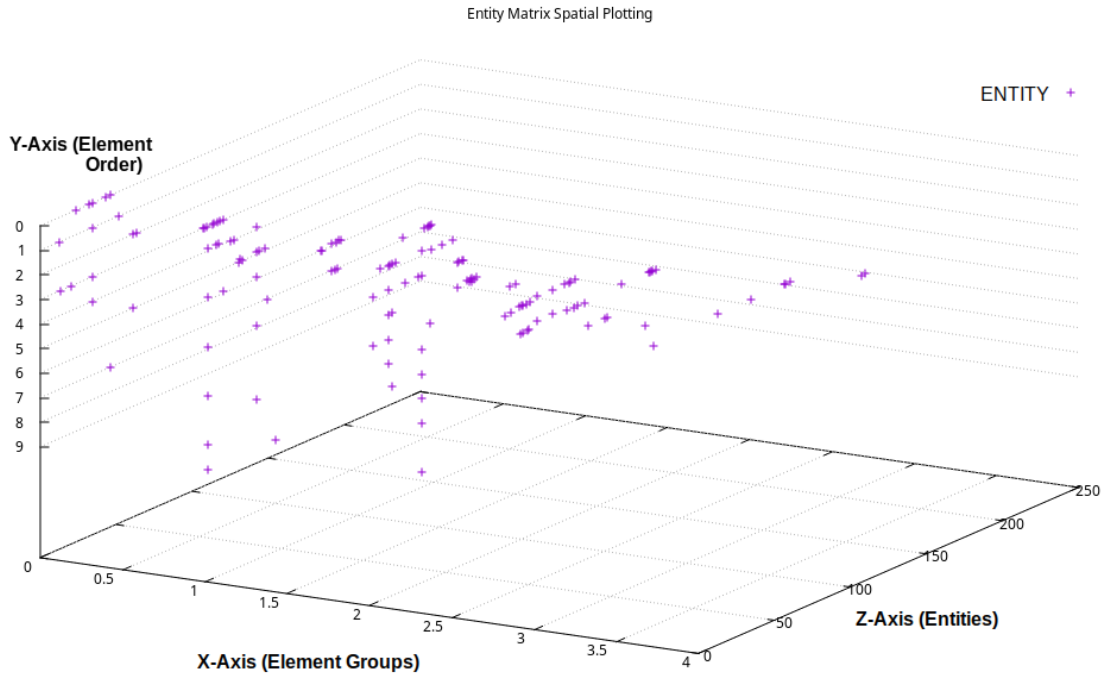
FIGURE 5.4: Raw *Entity* data examples.

```
Mention (Entity): 2
'the fox with him'
[The fox, his, I, the fox, the fox with him, he, him]
In Sentences: [0, 2, 3, 21, 23, 10, 11, 12, 29, 31]

Mention (Entity): 130
'the fox replied , `` Why art thou such a glutton'
[they, we, the fox replied , `` Why art thou such a glutton]
In Sentences: [17, 18, 19]

Mention (Entity): 148
'a barrel in the cellar'
[the barrel, a barrel in the cellar]
In Sentences: [19, 27]

Mention (Entity): 46
'the farmer'
[the farmer , who had heard the noise of the fox 's jumping, the farmer]
In Sentences: [5, 28, 31]
```

The above example of raw *Entity* data (Figure 5.4) shows the use of CRA (Co-Reference Analysis) to produce mentions shared across several sentences. Each mention (potential *Entity*) is shown with a representative mention, list of variants (also containing the representative mention) and the sentences each one was found in.

The effectiveness of the CRA system in CoreNLP can be seen by looking at the diverse range of mentions correctly identified, . Though one can see (in *Entities* 2 and 130) that sometimes the CRA incorrectly separates what should be one mention into several due to inescapable statistical error in the algorithm.

FIGURE 5.5: 3D *Entity Matrix* example (plotted with GNUplot).



The 3D *Entity Matrix* plot shown in Figure 5.5 provides a visual representation of the *Entity* alignment within the structure of the text. As with the *Element Matrix* before it, one can see the lopsidedness of the clustering algorithm (a limitation explored in the next section). Despite this, the 3D plot helps explain the structure of the *Entities* and their mentions in *Elements* of the text.

AIMS ACHIEVED:

As set out in Section 1.5 and reviewed in Section 4.7, the aims of this dissertation were to try and re-create Lévi-Strauss' process of *structuralist analysis* (SA) computationally. As has already been discussed, the *Element Matrix* output does come close to something resembling SA, while the *Entity Matrix* output lends additional relational information to the previous data.

Both of these have promise in working towards more completely analysing the structure of text, and could be modified and extended to increase both the range of information available to process and output, as well as improve the accuracy and robustness of the system. The overall evaluation of the attempt then, is that while there needs to be much work done to improve such a system, the process of SA is feasible as a model of understanding text computationally.

5.2 Problems Encountered

Some of the problems encountered in the practical implementation of SA are noted here...

CLUSTERING ISSUES:

In Section 5.1: Figure 5.2 and Figure 5.3 exhibit an asymmetrical or "lopsided" clustering of narrative *Elements*. This occurs when commonly-used verbs in sentences take precedence over the more unique verbs when performing *Element* similarity computation using cosine similarity.

The *frequency vectors* cannot understand how relevant a particular verb is to the sentence an *Element* is created from. Therefore high-frequency but unimportant verbs can skew the clustering. There is no clear way to resolve this in the current system, only expanding the scope of the similarity computation to other types of data can fix it.

DEPTH OF INFORMATION (DETAIL):

This is more a limitation of the solution that a straight-up problem, but the lack of deeper (read: more comprehensive) language data being used hampered attempts at modelling higher-level linguistic constructs and producing better *Element* comparisons. The best example of this is the narrative *Element* objects, lacking a lot of the complexity of Lévi-Strauss' *Mytheme* notion.

Time constraints, both in the upskilling on CoreNLP and changes/reviews made to the approach as problems arose, slowed progress on the foundational aspects of SA, preventing more work being done on exploring better analyses. If a longer period of time was allocated to investigation this problem would surely be mitigated.

COMPLEXITY:

One of the more vague problems during the dissertation research was the increasing complexity of finding working solutions to parts of SA as the project progressed. The obvious example of this is the proposed comparison system, which

would require an entire research project on its own to complete – as it contains a number of very open problems with no clear solution.

5.3 Future Work

Some of the proposed systems (such as the ideas for comparing matrices in Section 3.7) were not implemented due to time constraints and complexity of investigation. Mentioned in this section are some of the areas for performing future work...

COMPARISON SYSTEM:

The initial concepts for the SA-based comparison system have been mentioned in Section 3.7, namely the *Column Matching* and *Column Scoring* stages. Further stages or refinements to the initial stages are necessary before attempting to realise such a system.

CULLING DATA:

One of the possible optimisations to the approach described in this report is the culling (i.e. removal) of datums viewed as too minor to contribute to the analysis. An immediate example would be the rejection of the large volumes of simpler verb/adjective data that negatively affects the clustering of *Elements*.

In order to do so, some kind of user-defined heuristic could be used to determine importance of particular words, allowing the *frequency vectors* be narrowed in scope and – hopefully – broadened in uniqueness, enabling better "thematic" results in the grouping of *Elements*.

AGGREGATING *Elements*:

As has been mentioned throughout the report, the narrative *Elements* defined in this implementation act as a simplified analogue to the *Mythemes* of Lévi-Strauss. In order to bring the current narrative *Elements* closer to the notion of the *Mytheme*, a system for aggregating or "collapsing" *Elements* into larger objects is required.

EXTENDING *Elements* & *Entities*:

There is far more to be done to extend the notion of the narrative *Elements* and *Entities*, first and foremost the use of additional data provided by CoreNLP. An example of this which may prove useful is the use of sentence-level dependency parsing, which builds a graph of each sentence and establishes which parts (i.e. sub-strings) depend on others. Comparing and computing similarity based on this information could be improve the overall accuracy of the system, and lend new forms of interaction between *Elements* and dependent *Entities*.

As was also mentiond in Section 3.4, combining the CRA and NER systems provided by CoreNLP into a more robust *Entity* extraction system would be of great benefit.

The above propositions for future work are but a small number of potential improvements and new developments in implementing SA computationally.

5.4 Conclusion

The research presented in this dissertation write-up attempted to broadly approximate the SA process developed by Claude Lévi-Strauss. Section 1.5 set out the general goals of this project; with Section 3.8, Section 4.7, and Section 5.1 evaluating the approach designed and implemented during this research.

The overall conclusion is that the initial work done holds promise. Applying *Structuralist Analysis* (SA) using current natural language processing (NLP) tools has been shown to be feasible, at least in some rudimentary form. It is the expectation of this author that further development and expansion of the notion will provide an alternative method of understanding text in computational systems.

Appendix A

"Example Texts"

In addition to the example given in Section 5.1, a few extra examples are included in this report. This appendix lists the examples provided in Appendix B, along with the relevant links to sources...

1) Grimm Fairytales (Rapunzel):

Another Grimm fairytail in keeping with the mythic/folkloric theme of the origins of SA, this one longer and more widely-known. E-Book version can be sourced from Project Gutenberg (https://www.gutenberg.org) and raw HTML text version available at http://www.sacred-texts.com/neu/grimm/ht06.htm.

2) Entertainment News Articles (BBC):

Several randomly-chosen articles from the BBC ("British Broadcasting Corporation") archives related to entertainment news are provided to showcase the use of the SA system on traditionally non-mythic/narrative texts:

- "Duran-Duran Show Set For US T.V."
- "Pupils To Get Anti-Piracy Lessons"
- "U2 To Play At Grammy Awards Show"

Sources for these articles are the BBC "Full Text" corpus available here: http://mlg.ucd.ie/datasets/bbc.html **[Greene and Cunningham, 2006]**.

Appendix B

"Example Output"

1) Rapunzel:

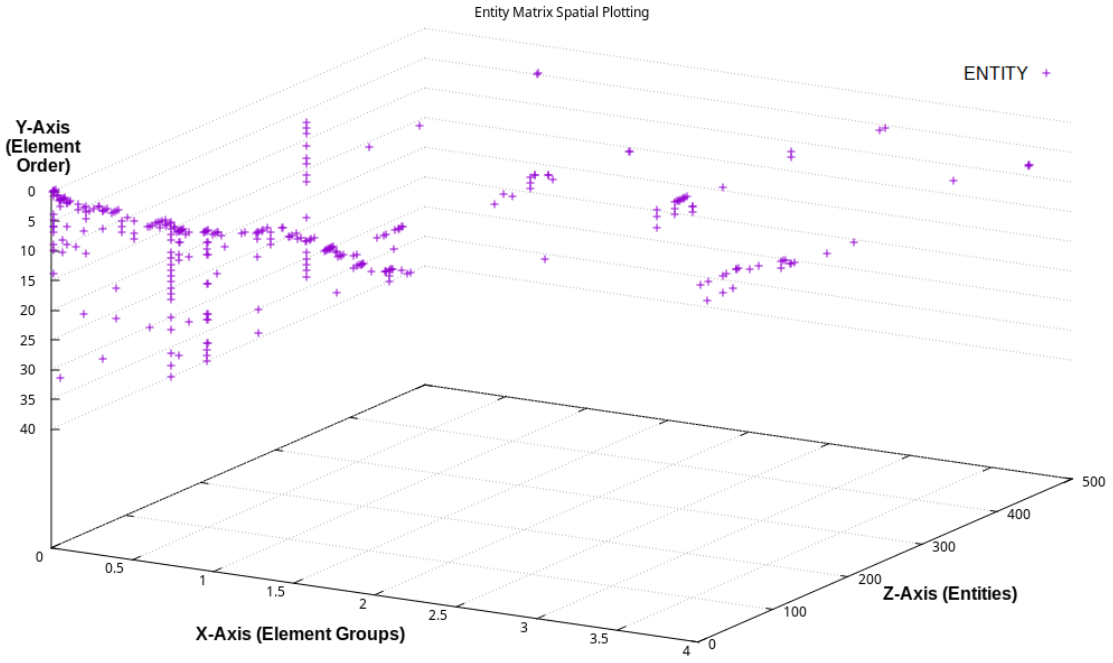
Group (0)	Group (1)	Group (2)	Group (3)	Group (4)
0				
1				
2				
3				
		4		
5				
6				
				7
8				
9				
				10
11				
12				
13				
14				
15				
				16
				17
18				
19				
20				
21				
22				
			23	
24				
		25		
26				
27				
		28		
		29		
			30	
31				
			32	
33				
34				
35				
36				
37				
38				
39				
		40		
41				
42				
43				
44				
45				
46				
47				
48				
49				
	50			
				51
52				
			53	
54				
55				

Group (0)	Group (1)
be/have/wish	
hope/be/grant	
have/be/see/be	
be/surround/dare/go/belong/have/be/dread	
increase/know/get/pine/look	
be/alarm/ask/aileth	
love/think/let/die/bring/let/cost	
clamber/clutch/take	
like/long	
be/have/descend	
let/have/clamber/be/see	
dare/say/descend/steal	
suffer	
allow/be/soften/say/be/sayest/allow/take/wilt/make/give/bring/be/treat/care	
consent/be/bring/appear/give/take	
grow	
be/shut/lay/have/be	
want/go/place/cry/let	
come/pass/ride/go	
be/pass/let/resound	
want/climb/look/be/be/find	
be/mount/try/say/begin/grow/go/cry/let	
be/frighten/have/behold/come/begin/talk/tell/have/be/stir/have/let/have/have/be/force/see	
lose/ask/take/see/be/think/love/do/say/lay	
say/go/do/know/get	
bring/weave/be/descend/take	
agree/come/come	
remark/say/tell/happen/be/draw/be	
think/have/separate/hast/deceive	
clutch/wrap/round/seize/snap/be/cut/lay	
be/take/have/live	
cast/fasten/have/cut/come/cry/let/let	
ascend/do/find/gaze	
cry/wouldst/fetch/sit/have/get/scratch	
be/lose/see	
be/leap	
	escape/fall/pierce
roam/come/have/give/live	
wet/grow/see	
lead/be/receive/live/contend	

...

Group (2)	Group (3)	Group (4)
be/stand/look/see/be/plant/look/long/have/eat		
		reply/get/be/eat/die/
		make/eat/
		answer/let/take/make/do/
		see/feel/have/die/have/get/eat/
	have/spin/hear/unfasten/wind/fall/climb	
hear/be/stand/listen		
ride/have/touch/go/listen		
be/stand/see/come/hear/cry/let		
	let/climb	
	fall/climb	
cry/do/hear/say		
		wander/eat/do/
	hear/seem/go/approach/know/fall/weep	

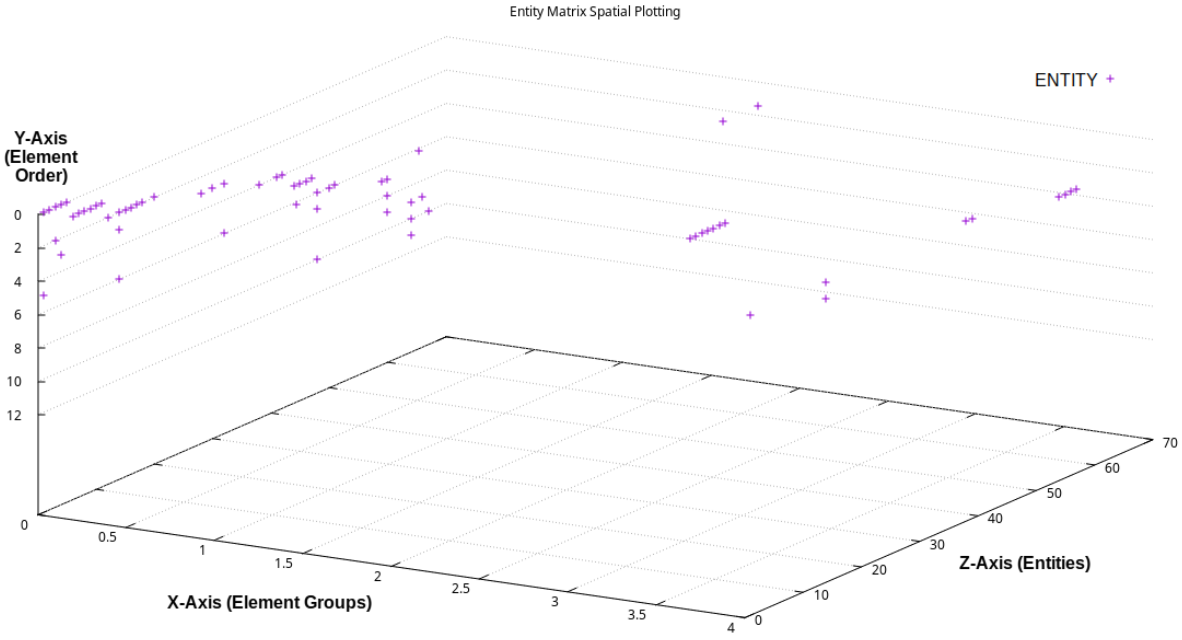
...


Entity Matrix Spatial Plotting
ENTITY
Y-Axis (Element Order)
0
5
10
15
20
25
30
35
40
X-Axis (Element Groups)
0
0.5
1
1.5
2
2.5
3
3.5
4
Z-Axis (Entities)
0
100
200
300
400
500

2a) DURAN-DURAN SHOW SET FOR US T.V.:

Group (0)	Group (1)	Group (2)	Group (3)	Group (4)
0				
1				
2				
			3	
4				
	5			
				6
7				
8				
9				
				10
11				
12				
		13		
14				
15				
16				

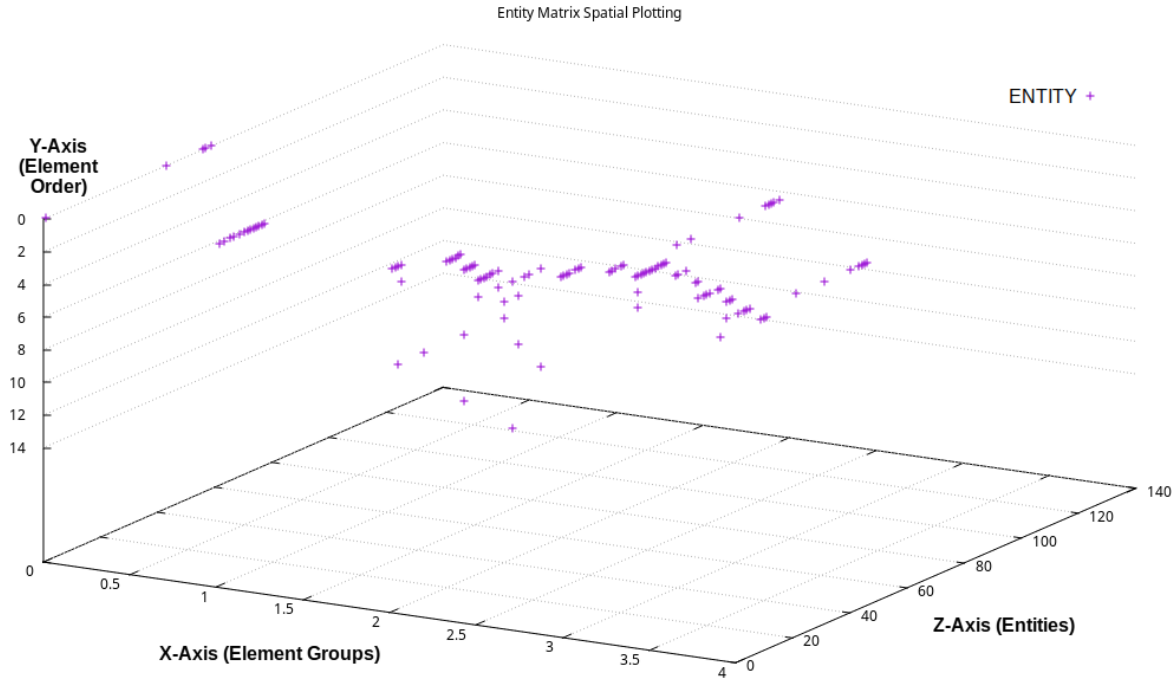
Group (0)	**Group (1)**	**Group (2)**	**Group (3)**	**Group (4)**
be/appear/include/				
air/feature/				
seem/say/				
			release/	
tour/				
	start/play/do/try/make/			
				want/make/say/
be/				
be/promote/happen/				
say/be/try/update/				
				want/lose/try/try/connect/say/
say/be/get/				
be/get/				
		argue/		
be/inspire/				
do/say/				
be/				


Entity Matrix Spatial Plotting
ENTITY
Y-Axis (Element Order)
0
2
4
6
8
10
12
0
0.5
1
1.5
2
2.5
3
3.5
4
X-Axis (Element Groups)
0
10
20
30
40
50
60
70
Z-Axis (Entities)

2b) PUPILS TO GET ANTI-PIRACY LESSONS:

Group (0)	Group (1)	Group (2)	Group (3)	Group (4)
		0		
	1			
		2		
		3		
		4		
				5
		6		
7				
		8		
			9	
		10		
		11		
		12		
		13		
		14		
		15		
		16		
		17		

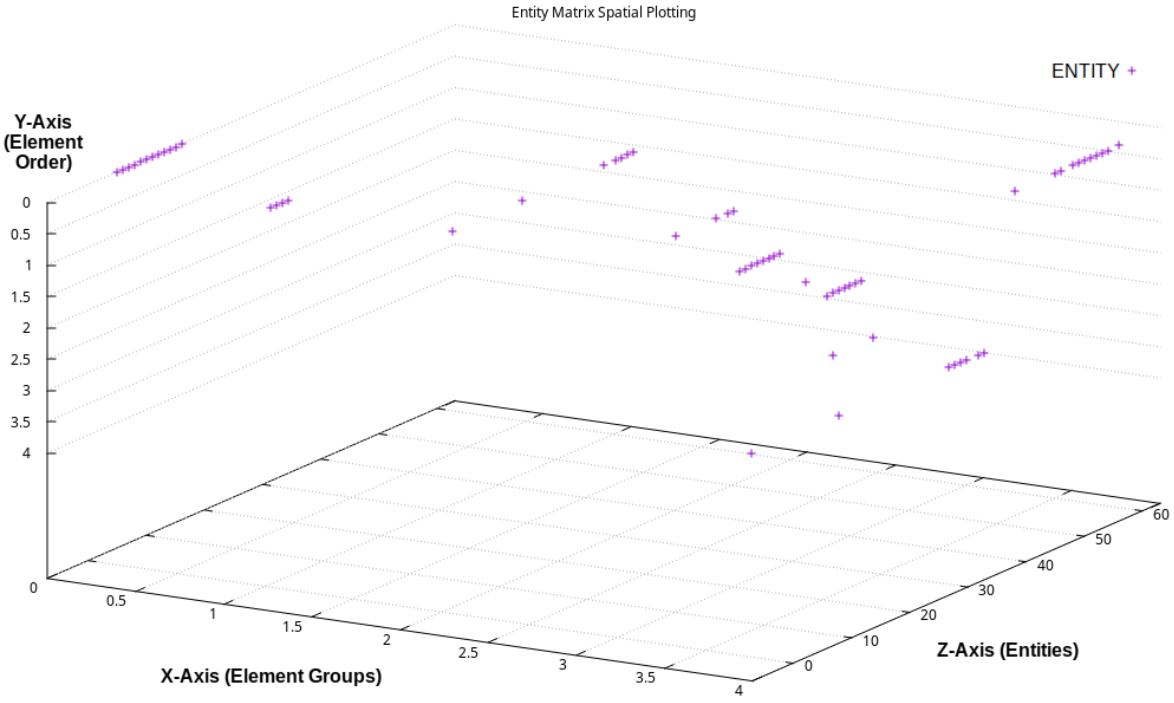
Group (0)	**Group (1)**	**Group (2)**	**Group (3)**	**Group (4)**
		be/be/teach/		
	aim/introduce/include/download/protect/			
		include/swap/cost/have/be/blame/		
		be/form/represent/work/put/		
		have/work/include/have/throw/		
				say/educate/protect/plan/
		launch/say/think/be/receive/		
give/work/help/				
		have/be/request/be/aim/give/		
			teach/raise/	
		tell/have/be/cry/help/educate/		
		be/be/be/back/		
		say/think/be/gain/work/be/		
		wish/have/be/give/be/		
		say/believe/be/		
		be/know/turn/		
		have/be/be/face/		
		be/allow/create/distribute/		


Entity Matrix Spatial Plotting
ENTITY
Y-Axis (Element Order)
0
2
4
6
8
10
12
14
0
0.5
1
1.5
2
2.5
3
3.5
4
X-Axis (Element Groups)
0
20
40
60
80
100
120
140
Z-Axis (Entities)

2c) U2 TO PLAY AT GRAMMY AWARDS SHOW:

Group (0)	Group (1)	Group (2)	Group (3)	Group (4)
	0			
1				
			2	
			3	
		4		
			5	
			6	
				7
			8	

Group (0)	Group (1)	Group (2)	Group (3)	Group (4)
	be/play/have/say/			
play/include/				
			be/nominate/include/	
			have/be/dominate/be/	
		present/		
			announce/be/host/	
			be/hold/	
				have/dismantle/top/
			be/dominate/be/undertake/	



Bibliography

Abeel, Thomas, Yves Van de Peer, and Yvan Saeys (2009). "Java-ML: A Machine Learning Library". In: *J. Mach. Learn. Res.* 10, pp. 931–934. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1577069.1577103.

Apache (2006–2015). *Apache UIMA*. Apache Software Foundation. URL: https://uima.apache.org/ (visited on 11/12/2015).

— (2008). *Unstructured Information Management Architecture (UIMA), Version 1.0*. working draft 5. OASIS UIMA Technical Committee.

— (2011–2014). *Apache OpenNLP Developer Documentation*. OpenNLP Development Community. URL: https://opennlp.apache.org/documentation/1.6.0/manual/opennlp.html (visited on 11/10/2015).

Balabantaray, Rakesh Chandra, Chandrali Sarma, and Monica Jha (2015). "Document Clustering Using K-Means & K-Medoids". In: *CoRR* abs/1502.07938. URL: http://arxiv.org/abs/1502.07938.

Barthes, Roland (1975). "New Literary History (Vol. 6, No. 2, On Narrative & Narratives)". In: The Johns Hopkins Univeristy Press. Chap. (*Introduction To The Structural Analysis Of Narrative*).

Bird, Steven, Ewan Klein, and Edward Loper (2009). *Natural Language Processing With Python*. O'Reilly Media.

Blackburn, Simon (2008). *Oxford Dictionary Of Philosophy*. 2nd (revised). Oxford University Press.

Bourrillion, Kevin and Jared Levy (2015). *Guava: Google Core Libraries For Java*. Google. URL: https://github.com/google/guava (visited on 04/01/2016).

Cunningham, Hamish et al. (2011). *Text Processing With GATE (Version 6)*. The University of Sheffield, Department of Computer Science.

de Saussure, Ferdinand and Albert Riedlinger (1983). *Course In General Linguistics*. (reprint). Open Court Publishing.

Finkel, Jenny Rose, Trond Grenager, and Christopher Manning (2005). "Incorporating Non-local Information Into Information Extraction Systems By Gibbs Sampling". In: *Proceedings Of The 43rd Annual Meeting On Association For Computational Linguistics*. ACL '05. Ann Arbor, Michigan: Association For Computational

Linguistics, pp. 363–370. DOI: 10.3115/1219840.1219885. URL: http://dx.doi.org/10.3115/1219840.1219885.

GATE (1995-2015). *GATE: A Full-Lifecycle Open-Source Solution For Text Processing*. The University of Sheffield, Department of Computer Science. URL: https://gate.ac.uk/sale/tao/splitch7.html (visited on 11/11/2015).

Greene, Derek and Pádraig Cunningham (2006). "Practical Solutions To The Problem Of Diagonal Dominance In Kernel Document Clustering". In: *Proc. 23rd International Conference On Machine learning (ICML'06)*. ACM Press, pp. 377–384.

Klein, Sheldon et al. (1974). *Modelling Propp & Levi-Strauss In A Meta-Symbolic Simulation System*. Tech. rep. University Of Wisconsin, Computer Science & Linguistic Department.

Lee, Heeyoung et al. (2011). "Stanford's Multi-pass Sieve Coreference Resolution System At The CoNLL-2011 Shared Task". In: *Proceedings Of The Fifteenth Conference On Computational Natural Language Learning: Shared Task*. CONLL Shared Task '11. Portland, Oregon: Association For Computational Linguistics, pp. 28–34. ISBN: 9781937284084. URL: http://dl.acm.org/citation.cfm?id=2132936.2132938.

Lévi-Strauss, Claude (1963). *Structural Anthropology*. (reprint). Basic Books.

— (1969). *The Raw & The Cooked*. (reprint). Harper & Row, Publishers Inc. & Jonathan Cape Ltd.

Macqueen, J.B. (1967). "Some Methods For Classification & Analysis Of Multivariate Observations". In: *In 5th Berkeley Symposium On Mathematical Statistics & Probability*, pp. 281–297.

Manning, Christopher D. et al. (2014). "The Stanford CoreNLP Natural Language Processing Toolkit". In: *Association For Computational Linguistics (ACL) System Demonstrations*, pp. 55–60. URL: http://www.aclweb.org/anthology/P/P14/P14-5010.

Raghunathan, Karthik et al. (2010). "A Multi-pass Sieve For Coreference Resolution". In: *Proceedings Of The 2010 Conference On Empirical Methods In Natural Language Processing*. EMNLP '10. Cambridge, Massachusetts: Association For Computational Linguistics, pp. 492–501. URL: http://dl.acm.org/citation.cfm?id=1870658.1870706.

Stanford (2016). *Stanford CoreNLP Repository*. Stanford NLP Group. URL: https://github.com/stanfordnlp/CoreNLP (visited on 11/29/2015).

Toutanova, Kristina et al. (2003). "Feature-Rich Part-Of-Speech Tagging With A Cyclic Dependency Network". In: *Proceedings Of The 2003 Conference Of The North American Chapter Of The Association For Computational Linguistics On Human Language*

Technology - Volume 1. NAACL '03. Edmonton, Canada: Association For Computational Linguistics, pp. 173–180. DOI: 10.3115/1073445.1073478. URL: http://dx.doi.org/10.3115/1073445.1073478.

Williams, Thomas and Colin Kelley (2015). *Gnuplot 5: An Interactive Plotting Program*. http://gnuplot.sourceforge.net/.

Zong, Jonathan (2013). *K-Means Clustering With tf-idf Weights*. URL: http://jonathanzong.com/blog/2013/02/02/k-means-clustering-with-tfidf-weights (visited on 12/30/2015).