

CodeRead: A multiplatform coding engine for text-based data

Andrew J. Perrin
Department of Sociology
University of California, Berkeley
216 Oxford Hills Drive
Chapel Hill, North Carolina 27514
`aperrin@socrates.berkeley.edu`

Presented at the
American Sociological Association annual meeting
Washington, D.C., August 16, 2000

Abstract

Most social science research uses data that originate, in one form or another, as written or spoken text. Quantitative researchers code these data very strictly, categorizing answers to questions into fixed groups. By contrast, qualitative researchers typically code free-form text by marking it up according to a set of ideas about the nature and context of the text.

This paper presents CodeRead, a computer-based system for systematically coding large quantities of free-form textual data. The system's principal innovation is its ability to generate coding rules from a pre-coded sample of text. This capacity allows for the analysis of much longer textual data than was previously practical. It also insures that the rules used for coding such data are specific and uniformly applied. The paper uses interviews with former members of the Namibian independence movement to illustrate the capabilities and limitations of the system.

Most social research uses data that, at one point or another, are in the form of written text. This includes both traditionally “quantitative” data usually coded from structured or semi-structured interviews as well as “qualitative” data coming from a variety of sources including interviews, focus groups, media analysis, and participant observation. Each of these traditions uses different techniques to distill the information contained in linguistic interactions into analytic categories. Generally, survey-based quantitative research codes answers according to pre-determined categories, while qualitative research seeks to identify themes and ideas that recur in the text.

This paper describes the CodeRead system, a computer analysis package I developed that seeks to facilitate the reliable, specific coding of textual data without requiring a set of *a priori* coding rules. It aims, thereby, to bridge the classic qualitative-quantitative divide by facilitating researchers’ use of unstructured, text-based data in a way that allows for systematic, reproducible analysis. This paper outlines the purpose of the system as well as the benefits it offers potential users.

The Namibia Study

As an example, the paper presents key elements of an analysis of a series of fourteen interviews carried out in 1993 with members of the Namibian independence movement. The interviews were part of an earlier research project, and cover a range of topics regarding the country’s then-recent shift from colony status to independence. There were sixteen in-depth interviews in all, carried out with a variety of elite and grassroots Namibians soon after independence. I discarded two of the interviews before the analysis. One, with prime minister Hage Geingob, was discarded because the tape recorder malfunctioned, leaving no reliable transcript. The other, with a leading conservative politician and businessman, Dirk Mudge, was out of place since it was the only interview not conducted with a participant in the independence movement.¹

I hand-coded a small subset of the data, a randomly selected sample of about 15% of the text of each of four interviews: those with Gwen Lister, the editor-in-chief of *The Namibian*, a pro-independence newspaper; Nathaniel Maxhuilili, an outspoken member of parliament from the South-West Africa People’s Organization (SWAPO), the main independence organization; Libertine Amathila, the Minister of Local Government and Housing; and Theo-Ben Gurirab, the Minister of Foreign Affairs. I did not code the interview sample fully; rather, I coded for three concepts:

1. Positive comments about post-independence changes. Coded as `change_positive`

¹See Perrin (1994, 1995) for other analysis of these interviews. Those interested in using these interviews for further analysis may use them—as well as a large collection of copies of *The Namibian* through the Bancroft Library at the University of California, Berkeley, or by contacting me.

2. Negative comments about post-independence changes including comments such as “we have not come far enough” and others that suggest independence will still bring better days. Coded as `change_negative`.
3. Explanations of interviewees’ personal experiences in the movement. Coded as `experience`.

1 Text as Data

A variety of qualitative approaches to social research use unstructured text as data. These include studies using media reports (e.g., Franzosi 1995; Tilly 1995), speeches, focus groups (Cerulo, 1998; Gamson, 1992), in-depth interviews, and multiple types of text (e.g., Wagner-Pacifici, 1994). Most approaches to analyzing such qualitative data involve a coding scheme in which the researcher painstakingly studies the text, using some technology (be it glue and scissors, index cards, highlighter pens, or qualitative data analysis software) to mark portions of the text as representing one or more of the themes salient in the study.

Researchers who use such methods prefer them to structured, survey-style studies because they allow deeper, more extended portrayals of research subjects’ ideas and because they avoid the problem of pre-determining the categories into which respondents’ answers will be divided. However, these methodologies suffer from significant drawbacks as well. Some of these drawbacks are technical: the process of reading and coding text is so time-consuming that studies using qualitative techniques tend to be relatively small, raising questions about their pertinence to broader theoretical questions. Other concerns are more substantive; for example, readers must trust that researchers’ coding is correct and consistent, since the studies’ conclusions rest entirely on the author’s perception of the meaning of the text. It is difficult, if not impossible, to establish measures of the reliability and validity of qualitative codes.

Nevertheless, free-form text is a tremendously promising form of data. Contemporary societies produce huge amounts of such text in the course of everyday life; newspapers, magazines, books, speeches, advertising copy, and legal and political proceedings are just a few of the institutional environments from which text emanates. The growth of the internet makes huge and increasing amounts of this text available for little cost. Furthermore, techniques such as focus groups and interviews yield free-form text that can be analyzed in much the same way. These texts contain far richer, more detailed insights into their subjects than the highly structured quantitative data available.

CodeRead offers a means for preserving these advantages of free-form textual data while addressing some of the disadvantages. CodeRead uses an open design and a series of algorithms to “learn” patterns of text coded by a researcher, apply these patterns to more text, and produce a variety of summary output for use in analyzing the data. It seeks thereby to approximate the researcher’s coding patterns, and its open design allows users to design custom procedures

to perform different analyses from those that are built in.

2 Computers in Textual Analysis

CodeRead uses previous work in three traditions: computer content analysis (Stone, 1966); computational linguistics (Manning and Schütze, 1999); and qualitative data analysis. It is not entirely within any of these traditions; rather, it derives functionality from each.

The computer content analysis tradition goes back to Philip Stone’s 1966 work on the General Inquirer, a computer program that applies “dictionaries” of codes to texts and produced summary output. The General Inquirer—now online at <http://inquirer.wjh.harvard.edu>—relies on a researcher’s predetermined codes to work. That is, researchers provide the program with a dictionary containing a list of words and how they should be coded. The General Inquirer then codes the text(s) accordingly. Also, the General Inquirer codes entirely on the basis of single words; for example, the Harvard dictionary used by the Inquirer’s online version codes “guns” as, among other things, *negative*—an association that is more appropriately tested by the context of the word’s use than assumed *a priori*.

At the time the General Inquirer was originally developed, the computer time necessary was so expensive that it remained impractical to use the system to code and analyze large amounts of text. Also, the requirement that a dictionary be created before the analysis was undertaken meant that coding rules had to be developed outside the text itself rather than through an inductive process based on a researcher’s own coding practices.

The General Inquirer is able, to a significant extent, to disambiguate word senses; users may therefore find it useful to use the system to attempt to distinguish among the different ways a given word may be used before coding it using CodeRead. For example, the word *over* can have multiple meanings in English. These meanings are established by the context in which the word is used. In the phrase “over the bridge,” *over* has an entirely different meaning than in the phrase “this meeting is over.” The General Inquirer implements the algorithms published in Kelly and Stone (1975) to distinguish among these “word senses” within texts.²

The field of computational linguistics is the most technically developed of the three fields that relate to CodeRead. Computational linguistics seeks to represent complex linguistic structures in systematic ways and, thereby, to enable computers to “speak” in idiomatic ways and to understand human language. Carbonell’s (1981) work, in particular, tried to model the structure of ideological belief systems. However, the emphasis in most computational linguistics is on modeling increasingly complex structures. Most research does not deal with

²Users of CodeRead may find it useful to first disambiguate their texts using the General Inquirer or another word-sense disambiguator. A limited amount of this capacity is available on the General Inquirer’s web site.

summarizing or coding large amounts of text or with methods for estimating the meanings of parts of such text.

The field of natural language processing (see Manning and Schütze (1999)) is the part of computational linguistics most closely related to CodeRead. Natural language processing uses computers to analyze the structure of languages: the frequency of word uses, keywords, expressions, and other word patterns. As a technical part of the field of linguistics, however, its focus remains on understanding the uses of parts of speech such as word collocations (combinations of words that are common in languages) and senses (multiple meanings of the same word).

Finally, there are numerous commercial programs currently available for use in analyzing qualitative data. These include NUD*IST, WinMax, HyperResearch, Atlas-TI, FolioViews, and AskSam (Dohan and Sanchez-Jankowski, 1998; Fielding and Lee, 1998). These mostly graphically-oriented programs allow qualitative researchers to mark parts of text with one or another code, then see similarly-coded text arranged by code. With some such software, users can also test hypotheses about the relationships among codes and attach demographic information about the speakers to develop ideas about the texts. Most software in this field, though, concentrates on allowing users to hand-code data and to view the coded data in various ways. There is no attempt to develop systematic ways of coding the texts, nor is there much concentration on presenting summary or statistical information about large quantities of textual data.

The development of microchip-based personal computers and the extraordinary reduction in the price of computer power suggest that computers could be used to standardize and carry out the coding and analysis of very large textual datasets. Computer power that, when *The General Inquirer* was originally written, would have required specialized computer rooms, card punch machines, and enormous financing is now available in desktop computers. Similarly, the prices of memory and disk storage have declined so dramatically as to make the storage and manipulation of all but the largest texts a serious possibility.

3 Functions and Algorithm

CodeRead is a three-stage textual analysis tool, currently available free of charge to interested users. It allows a user to code, systematically and efficiently, a potentially very large textual dataset³ without having to develop an *a priori* set of coding rules. It does this by “learning” a set of coding rules based on a researcher’s own coding of a sample of text, then applying these (or other) coding rules to the rest of the text.

The three stages a researcher would typically go through in analyzing a dataset using CodeRead are:

³My computer, a relatively low-end Pentium 150 with 96MB of RAM, has no problem coping with projects of over 450,000 words, or around 2,000 pages of typed, double-spaced text. Perl has no built-in limits on the size of a data structure, so there is no theoretical limit to how big a CodeRead project can be.

Table 1: CodeRead Terminology

Project	One or more text files, with or without added coding, that can be used by CodeRead
code	A label attached to a portion of text that identifies something of substantive interest about the text
Textchunk	A contiguous subset of a Project containing some of the words in the Project
Codechunk	A Textchunk with a code attached
Pseudochunk	A collection of two or more Textchunks that are not contiguous within the project. Internally, CodeRead is able to carry out most of the operations available on Textchunks on Pseudochunks as well.
Pattern file	A list of ‘recipes’ CodeRead can both generate and use to attach codes to a Project or Textchunk

1. Feed a portion of the text, coded by hand (see below for details) into the program, instructing CodeRead to generate a set of coding rules from the user’s hand coding (called a *pattern file*);
2. Feed the entire text into the program along with a pattern file to generate a systematically, reliably coded textual dataset; and
3. Feed such a coded textual dataset into the program in order to obtain summary information about the data.

This list, of course, is limited to parts of the process that involve the CodeRead program. It does not include other time-consuming issues including transcribing text or otherwise making it suitable for analysis or editing the pattern file generated by CodeRead before applying it to the project. The remainder of this paper illustrates the process of using CodeRead.

3.1 Preparing text files for use in CodeRead

Before CodeRead can be useful, the data must be in a format it can read. CodeRead works on plain-text files only; it cannot operate on files saved in formats such as Microsoft Word or HTML.⁴ A collection of one or more text files, with or without hand coding, is called a **CodeRead Project** (see Table 1). Several files may be strung together using CodeRead’s `include` command. Inserted into a project file in the form `{include=<filename>}`, the file referenced in `<filename>` is simply put in place. The Namibian interviews are stored in fourteen separate files, one for each interview; the project file is simply a list of `include` statements (see Figure 1).

⁴Note, though, that there are useful conversion filters available to make this less of a problem.

Figure 1: The Namibia interviews project file

```
{include=AMATHILA.doc.txt}
{include=ASHIPALA.doc.txt}
{include=ELAGO.doc.txt}
{include=GURIRAB.doc.txt}
{include=HMBAKO.doc.txt}
{include=IMMANUEL.doc.txt}
{include=ISAACKS.doc.txt}
{include=KAZOMBAU.doc.txt}
{include=LISTER.doc.txt}
{include=MAXHUILI.doc.txt}
{include=NDAITWAH.doc.txt}
{include=NMBAKO.doc.txt}
{include=SHIVOLO.doc.txt}
{include=UIRAB.doc.txt}
```

3.1.1 Strategies for Developing Code-Sets

In general, coding for abstract comments, opinions, etc., makes CodeRead less reliable. The most successful coding strategy is to decide what question the codes will answer. For example, coding the **experience** chunks, CodeRead is being asked to distinguish between parts of the text where interviewees were using personal experiences to underscore points. It would be much harder, for example, to ask CodeRead to determine whether an interviewee approved or disapproved of a policy, or whether she were gainfully employed. This is because CodeRead can only use the existence of words and combinations of words to predict codes.⁵

One way of generating a useful code set is to ask what logical frame a speaker or writer is using to back up a point. It is a relatively trivial task to code the specific position the speaker or writer takes; coding for the rhetorical strategy used allows for avoiding the problem of negating words and offers the possibility of interesting insights into how speakers and writers understand the issue under investigation.

⁵An instructive example of one limitation of the approach would be asking the system to determine from a transcript whether two speakers approved or disapproved of Bill Clinton. Consider the two answers: (1) “I don’t know... Bill Clinton’s a jerk” and (2) “I don’t know if Bill Clinton’s a jerk!”. The word patterns in the two are almost exactly the same. Properly handling negating words like *don’t* is one of the unsolved problems in Natural Language Processing.

3.2 Generating Coding Rules from Coded Text

The first step in creating a CodeRead project is to code portions of a text by hand. The amount of text to be coded at this stage is left up to the researcher; it is entirely possible, however, to return to the project multiple times, refining the coding technique using feedback from the system. Such a technique allows users to determine at what point the patterns the computer has ‘learned’ are sufficiently similar to the by-hand coding to perform the rest of the analysis automatically. This original coding is done in exactly the same way the researcher would code traditional qualitative data, and the codes are entered according to the marking scheme discussed below in the technical section of this paper.

The user then enters this coded text into CodeRead and instructs the program to generate a series of patterns for each code. The patterns CodeRead generates are in the form of *regular expressions* (see Friedl, 1997) and *word combinations*. By convention, regular expressions are bounded by forward slashes (/); I include their schematic representations here for reference. CodeRead will try to generate patterns in each of the following forms:

- containing a particular ordered, contiguous set of one or more words (a phrase), e.g., code all instances of ‘enough’ as `change_negative`, represented as `/enough/`, or all instances of ‘progress’ as `change_positive`, represented as `/progress/`.
- containing a particular ordered combination of m words within an n -word block, e.g., code all text areas containing ‘not’ followed within one word by ‘yet’ as `change_negative`, represented as `/not(?:\s+\w+){0,1}\s+yet/`.
- a particular non-ordered, semi-contiguous set of words, e.g., code all text areas containing the words ‘i’ and ‘went’ within two paragraphs of one another as `experience`, represented as `/i/&&/went/`.⁶

CodeRead is capable of using *any* properly formed regular expression to code text; however, in its current form, these three types of patterns are the only ones it can ‘learn’ from user-coded text.

3.2.1 Representing Coded Text

For all its operations, CodeRead uses a subset of the Standard Graphics Markup Language (SGML; see International Standards Organization (1986)) to represent codes inserted into text. Coding tags are enclosed in angle-brackets (<>), which means these cannot appear elsewhere in the text. Coded regions of text begin with a begin-code marker consisting of a code name enclosed in angle-brackets, and end with the same code name, enclosed in angle brackets and preceded by a slash (/). Code names may be any length, but must contain no

⁶The slashes are optional in a combination type recipe, but the words must be separated by two ampersands (&&), and there cannot be any spaces within the pattern. This rule may be relaxed in future versions.

spaces, line breaks, backslashes (\), braces ({}), punctuation marks (., ,, !) or hashes (#), since these are interpreted by the compiler. In addition, the equals sign (=) has a special meaning within a coding tag, so it cannot be part of a code name. Codes may overlap or even be coterminous; there is no limit on the number of codes that can be linked to any single part of the text.

Code names that begin with a colon (:) will be ignored by the Stage 1 pattern-recognition engine. This is useful for entering objective codes such as information about a speaker, the circumstances of an interview or focus group, or other information that is not based on the content of the text itself.

In addition to codes, CodeRead uses braces ({}) to mark *processor directives*. The only processor directive currently available is the `{include=}` directive described above. In future versions, directives will be available to direct the system on how to interpret codes, words, and punctuation marks. Future versions of the software will allow users to specify other parameters as well. For example, users will be able to determine how much text surrounding a matching pattern should be included when applying patterns and how simple or complicated the word combinations for which CodeRead searches should be.

When reading any file—whether coded fully, partially, or not at all—CodeRead generates a large, ordered list (represented internally as an array) of all the words in the text. The words are all represented in lower case to facilitate comparisons, and punctuation and whitespace are removed. This simple data structure, called the *words array*, is the basis for all CodeRead analysis. The entire project is, at baseline, only an ordered collection of words.

In addition to the words array, CodeRead generates a series of pointers to structure the text. Each pointer consists of a *textchunk* structure, which is simply a combination of a pointer to the words array along with numbers corresponding to the area's starting and ending point in the words array. In addition, some types of textchunks contain codes or other indicators of how the chunk of text to which they point is to be interpreted. These textchunks are called *codechunks*. All CodeRead datasets contain at least two sets of pointers: the *paragraphs* array, which splits the text based on paragraph markers (generally, two newline characters together), and the *sentences* array, which splits it based on punctuation marks. The sentences array contains references to the type of punctuation that closed the sentence: 'regular' for those punctuated with a period, 'question' for those punctuated with a question mark, and 'exclamation' for those punctuated with an exclamation mark. Future versions of the program will allow users to specify processor directives to change this association.

For texts with any codes in them, CodeRead also generates two more indexes. These are the *codechunks* array and the *codes* array. The codechunks array contains an ordered list of parts of the text that have been coded and the code they have been assigned; the codes array does the reverse, containing a list of coded text areas for each code name.

3.2.2 The Pattern File

The output from this stage is a list of coding rules, each with two or three scores. The first score (entitled *recall*) corresponds to the proportion of text areas coded with the relevant code that match the pattern. The second score (*precision*) is the proportion of text areas *not* coded with the relevant code that do *not* match the pattern. Each of these scores is between 0 and 1. Finally, F allows the user to specify a degree of tradeoff between precision and recall (the constant for the tradeoff is called α). F is calculated as:

$$F = \frac{1}{\alpha \frac{1}{P} + (1 - \alpha) \frac{1}{R}}$$

where P is precision, R is recall, and α is the tradeoff between the two.⁷ If α is not specified by a user, F is not recorded in the pattern file. When generating a pattern file, a user can specify minimum scores on any or all of these dimensions to instruct CodeRead to discard candidate patterns that are unsuitable for use in coding.

The file CodeRead produces, then, contains (among others) the following lines. The file it produces is called a *pattern file* and is in a format that can be read by Stage 2 of the process:⁸

```
/progress/ change_positive 1 0.3333333333333333
/not(?:\s+\w+){0,1}\s+yet/ change_negative 1 0.5
/i/&&/went/ experience 0.4 1
```

This process is rather slow, as the processing is intensive. On my home computer (a 150 mhz Pentium with 96MB of memory, running Windows NT 4.0), generating patterns for a subset of approximately 15% of four of the 14 interviews took just under four hours to complete.⁹

3.3 Applying coding rules to text

For the second stage, the user produces a list of patterns in a pattern file produced by Stage 1, by hand, or both and uses the list and the entire text to be coded as input. CodeRead applies the patterns to the text, marking sections of text with the appropriate codes. Users may write regular expressions to match patterns they want coded according to a custom pattern; CodeRead will apply any properly-formed regular expression, regardless of its origin. For example, a

⁷See Manning and Schütze (1999, p. 268ff) for more on these measures.

⁸Regular expressions are sometimes represented in parentheses with a series of option characters, as in (?-xism:). In some circumstances, the pattern file contains that format instead of the // format.

⁹Better processing speed, more memory, and a better operating system do matter; the same task, run on the same computer under Linux, took only 1 hour and 18 minutes; on a two-processor Sun Ultra Enterprise 450 running Solaris, it took only 18 minutes.

user could note that the pattern: *the word i, followed within ten words by one of the words say, went, or surprised* would have precision and recall scores of 1.000 in a sample. Even though CodeRead’s pattern-recognition engine could not generate this pattern automatically, the user could enter the pattern in a pattern file: `/i(?:\s+\w+){0,10}\s+(?:say|went|surprised)/` `experience` and CodeRead would mark areas that matched the pattern as ‘experience.’ Note that it is unnecessary to generate precision and recall scores for user-defined patterns; they are not needed to apply patterns to a text. They are only provided by CodeRead to assist users in choosing among patterns to be applied.

3.3.1 Paring Down the Pattern List

One of the limitations of the pattern-generating engine is that it produces too many reasonable patterns. Even when specifying moderately restrictive limits on precision, recall, and F , the engine may generate many options, including patterns that are subsets of one another. For example, the Namibia interviews gave rise to hundreds of patterns for the three codes I used. Two of the codes generated for the `<change_negative>` code were:

```
# Contains the pattern yet:
(?-xism:yet) change_negative 1 0.5
# not followed within 1 words by yet:
(?-xism:not(?:\ s+\ w+){0,1}\ s+yet) change_negative 1 0.5
```

The two patterns have the same scores (1.0 on precision, meaning it picked up no false positives, and 0.5 on recall, meaning half the `<change_negative>` codechunks were identified with the patterns). The first is clearly a superset of the second, since any area containing `not` followed within one word by `yet` must also contain the pattern `yet`.

In these situations, it is necessary for the researcher to use substantive knowledge about the data to decide whether the more or the less restrictive pattern makes the most sense. Therefore, before applying patterns generated in step one, researchers should edit and condense the pattern file.

After culling the list of available patterns, I selected the following fifteen to use:

```
# Recipes for code change\_negative:
# Contains the pattern enough:
(?-xism:enough) change\_negative 1 0.5
# not followed within 1 words by yet:
(?-xism:not(?:\s+\w+){0,1}\s+yet) change\_negative 1 0.5
# The words ya, it’s, not, and enough:
/ya&&/it’s&&/not&&/enough/ change\_negative 1 0.5
# The words lot, still, and done:
```

```

/lot/%%/still/%%/done/          change\_negative 1      0.5

# Recipes for code change\_positive:
# Contains the pattern progress:
(?-xism:progress)      change\_positive 1      0.3333333333333333
# The words great and progress:
/great/%%/progress/    change\_positive 1      0.3333333333333333
# it's followed within 1 words by fantastic:
(?-xism:it's(?:\s+\w+){0,1}\s+fantastic)
      change\_positive 1      0.3333333333333333
# The words scared and arrived:
/scared/%%/arrived/    change\_positive 1      0.3333333333333333
# Contains the pattern improving:
(?-xism:improving)     change\_positive 1      0.3333333333333333

# Recipes for code experience:
# Contains the pattern my:
(?-xism:my)      experience      0.6666666666666667      1
# The words i and say:
/i/%%/say/      experience      0.4      1
# The words i and went:
/i/%%/went/      experience      0.4      1
# The words i, very, and surprised:
/i/%%/very/%%/surprised/      experience      1      0.75
# The words think, myself, and the:
/think/%%/myself/%%/the/      experience      0.75      0.75
# The words activities and i:
/activities/%%/i/      experience      0.4      1

```

When applying a pattern file, CodeRead checks the text in overlapping two-paragraph blocks. That is, it first checks paragraphs 1 and 2 together; then 2 and 3; then 3 and 4; and so on. This approach insures that patterns that could span a paragraph boundary are not missed. It applies the least restrictive (“greediest”) match possible; for example, consider the text:

```

Her children took part in violence, but it was not the violence
that killed them.

```

Applying the pattern above, the match is ambiguous; the pattern matches both children took part in violence and children took part in violence, but it was not the violence. CodeRead resolves this situation in favor of the more inclusive match; in this case, it codes children took part in violence, but it was not the violence:

Her `<parenting>` children took part in violence, but it was not
the violence `</parenting>` that killed them.

The output of Stage 2 is a fully-coded text. The text is in a form that CodeRead can then read for the following stage; all codes are marked with angle brackets, and punctuation is preserved. The Project can be saved in a human-readable format (using angle-brackets to denote codes) or in a faster, machine-readable format.

Based on the 33 codechunks originally coded with the three codes I used for the demonstration (`change_negative`, `change_positive`, and `experience`), CodeRead coded an additional 301 chunks: 52 for `change_negative`, 20 for `change_positive`, and 229 for `experience`. For example, the chunk below—an excerpt from an interview with Nairo Mbako, a former guerilla fighter who was unemployed at the time of the interview, was appropriately coded `experience` by the system.

A: Well, I used to run away. We run away. I was lucky, let me just say I was lucky they didn't caught me. When we went with the organization, SWAPO, to Rehoboth, is where they arrest me for the first time. And they brought us here to jail in Windhoek Central Prison, and they start interrogating us, who sent you, who sent you to Rehoboth and so on, and we couldn't reveal who sent us or what we are going to do. And then they start beating us but they didn't put down a charge, set a charge what have we done. Then they released. After so hard struggle then I decided that no, I have to fled the country. It was in 1977. In there, in Angola I went for military training, joined the PLAN combatants, then I engaged myself in the front. So I stayed there, maybe at the front for 1 year, and then I start driving.

3.4 Obtaining summary information about the data

Once a fully-coded project is available (whether coded by hand or using Stages 1 and/or 2), CodeRead can produce a variety of summary information about it.¹⁰ CodeRead generates an internal representation of the dataset (see below for a complete description) and uses this representation to produce output as requested.

Some forms of output consist solely of sorting text according to code. For example, CodeRead can produce a listing of all areas coded as a particular code, or a “key word in context” (KWIC)-style listing that presents coded text within

¹⁰This is the least-developed section of CodeRead for the moment; I will be writing more ways of analyzing the data in the future.

its context such as the following, showing a short `change_negative` codechunk with 20 lines of context:

```
<CODED>
  <change\_negative>
    <justice>
      what we are seeing now are really killing hearts at
      the end of the day .
    </change\_negative>
  </justice>
Q: In what sense ?

A:
  <change\_negative>
    hard to swallow these changes, you know .

  </change\_negative>
Q:
  <experience>
    Why ?

A:
  <experience>
    Because, I mean, why most of the things are not
    taking place now .
  <change\_negative>
    <justice>
      We cannot talk anymore .

  </CODED>
  </experience>
  </experience>
  </change\_negative>
  </justice>
```

Other output options include a listing of all active codes in the document along with their frequencies and information about how much of the text is coded with one or more codes.

More advanced options are available as well. CodeRead will produce a correspondence matrix suitable for performing a correspondence analysis in other statistical software such as SPSS, S-Plus, Stata, or R (see Krinsky, 1998; Clausen, 1998; Greenacre and Blasius, 1994). It will also perform a *proximity analysis*. This is a score for any two codes corresponding to the mean distance between occurrences of the first code and the nearest occurrence of the second. A κ score

(Kelly and Stone, 1975, page 41) is also available; this measures the degree of ‘agreement’ between two codes, effectively measuring how much text is coded with both of the two codes.

Because of the modular structure of the algorithm, any analysis that can be done on an entire project can also be done on any portion of the project. Thus users can analyze parts of projects without re-coding each individual part.

In addition to these built-in analysis modes, CodeRead provides a well-documented data structure containing the dataset in a flexible, internally-represented form. This allows users to write functions in Perl to access and format the data in whatever way they require. CodeRead is therefore not restricted to performing a few built-in functions; it can be expanded relatively easily as requirements and uses for the system change.

4 The software

CodeRead is written entirely in Perl, a free language that runs on most modern computers, including Unix/Linux, Windows, and Macintosh. It is generally available under the GPL and its source code is also available. That means researchers can use it without any additional cost, and they can modify it to suit their needs. Perl is particularly adept at text manipulation using regular expressions, so much of the functionality that powers CodeRead is, in fact, built into Perl itself. Perl is also a fully-functioning programming language, so technically skilled users can build on the CodeRead model to perform a variety of different analyses on textual datasets. CodeRead provides access to project data at numerous different levels, so researchers can use it as a set of tools to carry out almost any sort of linguistic analysis.¹¹

CodeRead works by applying a ‘brute-force’ algorithm. That is, it does not seek to use artificial intelligence or other predictive techniques to anticipate the likelihood of patterns in text. Rather, it simply counts the occurrences of words and phrases and ‘tries’ each of these candidates until it encounters one that matches. This is not an especially efficient way of searching text, but it guarantees that all possibilities will be considered. It is particularly inefficient during Stage 1, when the system must try large numbers of patterns gleaned from each coded area. Nevertheless, the amount of computing power available for social science work has grown so quickly that these techniques are quite reasonable, even for large datasets. Even a few hours’ running time (likely on an older computer) to generate the pattern file is manageable.

CodeRead does not offer the user-friendly, point-and-click interface most qualitative data analysis software aims for. It is possible to use CodeRead as an interactive tool, but it is designed mainly to be used in ‘batch’ mode: users prepare a project, a set of instructions, and (if required) a pattern file, then start the system. The output is designed to be interpreted directly, although some

¹¹One particularly interesting idea might be using it in conjunction with the *Lingua:Wordnet* modules, which provide a connection to a full English dictionary, complete with word senses and parts of speech. See Brian (2000) for more information.

output could be used as input to a statistical package, for example to perform a correspondence analysis.

The system is also designed mainly to allow for reductive analyses of texts. Although it can be useful for traditional qualitative data analysis tasks such as compiling all text coded in a certain way or displaying coded text in context, the main innovation of CodeRead is its ability to compare, count, and graph the occurrences of codes within a text.

5 Technical Information

CodeRead is implemented as a top-level Perl package (`CodeRead.pm`), which contains most of the documentation but little code, plus five lower-level packages: `Project.pm`, `Textchunk.pm`, `Codechunk.pm`, `Pseudochunk.pm`, and `Utils.pm`. The first four of these lower-level packages define the object classes for which they are named. `Utils.pm` provides a collection of helpful utilities used in various places in the application. Most of the data structure used in the system is contained in `Project.pm`. `Project.pm` defines the internal representation of CodeRead datasets.

CodeRead can be obtained from its official web site, <http://demog.berkeley.edu/~aperrin/CodeRead>. Although the six packages (`CodeRead.pm`, `Project.pm`, `Textchunk.pm`, `Codechunk.pm`, `Pseudochunk.pm`, and `Utils.pm`) are distributed as separate files, no one of them is likely to be useful without the other two. Since `CodeRead.pm` calls the other two, it should never be necessary to use the others explicitly. For further technical documentation, see the `CodeRead.pm` file, which contains standardized POD (Perl's 'plain old documentation' format). Although there is relatively little documentation at this point, more complete user documentation will be available soon. For more information on using Perl, see Wall (1996). For specific information on using object-oriented Perl such as CodeRead, see Srinivasan (1997) or type `perldoc perltoot` on a computer with a recent installation of Perl.

6 Version 2

There are several features I expect to add to CodeRead in the future; while I have no specific plans for when these will happen, some of the ideas on the list are:

- A more user-friendly front end Users must currently learn some (relatively small) amount of Perl in order to use CodeRead at all. Eventually, there will be a tool for using at least some of CodeRead's tools without learning the programming language.
- Continuous-variable codes The current setup allows only binary variables: a chunk of text is coded with a particular code or it is not. It would be useful to be able to code continuous variables as well; for example, coding

a chunk of text as `<age=45>` would allow for analyses that considered the age of speakers in a transcript.

- Additional customizability At the moment, decisions such as punctuation, the character separating paragraphs, the size of the window of text that CodeRead considers at a time, and more, are built into the program. Via the processor directives (instructions contained in braces in a project), users should be able to change that.
- More patterns, better rules I will be adding more types of patterns CodeRead can recognize in coded text, as well as more user control over which patterns CodeRead automatically discards as being of little use in predicting codes. This could include the use of more sophisticated Natural Language Processing concepts such as Hidden Markov Chains. (Manning and Schütze, 1999; Charniak, 1993)

7 Conclusions

CodeRead uses the impressive (and increasing) power of modern computers to facilitate the often time-consuming and expensive task of coding free-form textual data. It promises to bridge the traditional divide between qualitative and quantitative sociological analysis by allowing systematic, reproducible research using non-structured, textual data. Qualitative researchers will find it valuable to expand the amount of data they can consider in a study, as well as to allow for reproducible, systematic analysis. Quantitative researchers will appreciate the ability to expand beyond structured questions by carrying out quantitative studies of free-form text.

References

- Brian, D. (2000, Summer). *Lingua::wordnet*. *The Perl Journal* 5(2), 40–48.
- Carbonell, J. G. (1981). Subjective understanding: Computer modeling of belief systems.
- Cerulo, K. A. (1998). *Deciphering Violence: The Cognitive Structure of Right and Wrong*. New York: Routledge.
- Charniak, E. (1993). *Statistical Language Learning*. Cambridge: MIT Press.
- Clausen, S.-E. (1998). *Applied Correspondence Analysis: An Introduction*. Beverly Hills: Sage.
- Dohan, D. and M. Sanchez-Jankowski (1998). Using computers to analyze ethnographic field data: Theoretical and practical considerations. *Annual Review of Sociology* (24), 477–498.

- Fielding, N. G. and R. M. Lee (1998). *Computer Analysis and Qualitative Research*. London: Sage Books.
- Franzosi, R. (1995). *The Puzzle of Strikes*. Cambridge: Cambridge University Press.
- Friedl, J. E. F. (1997). *Mastering Regular Expressions* (2nd ed.). Cambridge: O'Reilly.
- Gamson, W. A. (1992). *Talking Politics*. Cambridge: Cambridge University Press.
- Greenacre, M. and J. Blasius (Eds.) (1994). *Correspondence Analysis in the Social Sciences*. London: Academic Press.
- International Standards Organization (1986). International standard 8879: Information processing - text and office systems - standard generalized markup language (sgml). Technical report, Geneva.
- Kelly, E. F. and P. J. Stone (1975). *Computer Recognition of English Word Senses*. Amsterdam: North-Holland Publishing Company.
- Krinsky, J. D. (1998). Organizing the organizing of the unorganized: Opposition to workfare in new york city and the recombination of contentious repertoires.
- Manning, C. D. and H. Schütze (1999). *Foundations of Statistical Natural Language Processing*. Cambridge: MIT Press.
- Perrin, A. J. (1994, May). Come the revolution: Movement politics and namibian independence. Unpublished Honors Thesis, Swarthmore College.
- Perrin, A. J. (1995, April). Election fetishism: Perceptions of southern african democratization. New York African Studies Association.
- Srinivasan, S. (1997). *Advanced Perl Programming* (1st ed.). Cambridge: O'Reilly.
- Stone, P. J. (1966). *The general inquirer: a computer approach to content analysis*. Cambridge: MIT Press.
- Tilly, C. (1995). *Popular contention in Great Britain, 1758-1834*. Cambridge: Harvard University Press.
- Wagner-Pacifici, R. (1994). *Discourse & Destruction: The City of Philadelphia vs MOVE*. Chicago: University of Chicago Press.
- Wall, L. (1996). *Programming Perl*. Sebastopol, CA: O'Reilly & Associates.