A Workbench for Finding Structure in Texts

Andrei Mikheev
HCRC, Language Technology Group,
University of Edinburgh,
2 Buccleuch Place, Edinburgh EH8 9LW, UK.
Andrei.Mikheev@ed.ac.uk

Steven Finch
Thomson Technical Labs,
1375 Piccard Drive, Suite 250,
Rockville Maryland, 20850
sfinch@thomtech.com

Abstract

In this paper we report on a set of computational tools with (n)SGML pipeline data flow for uncovering internal structure in natural language texts. The main idea behind the workbench is the independence of the text representation and text analysis phases. At the representation phase the text is converted from a sequence of characters to features of interest by means of the annotation tools. At the analysis phase those features are used by statistics gathering and inference tools for finding significant correlations in the texts. The analysis tools are independent of particular assumptions about the nature of the feature-set and work on the abstract level of feature-elements represented as SGML items.

1 Introduction

There is increasing agreement that progress in various areas of language engineering needs large collections of unconstrained language material. Such corpora are emerging and are proving to be important research tools in areas such as lexicography, text understanding and information extraction, spoken language understanding, the evaluation of parsers, the construction of large-scale lexica, etc.

The key idea of corpus oriented language analysis is to collect frequencies of "interesting" events and then run statistical inferences on the basis of those frequencies. For instance, one might be interested in frequencies of co-occurences of a word with other words and phrases (collocations) (Smadja, 1993), or one might be interested in inducing word-classes from the text by collecting frequencies of the left and right context words for a word in focus (Finch&Chater, 1993). Thus, the building blocks of the "interesting" events might be words, their morpho-syntactic properties (e.g. part-of-speech, suffix, etc.), phrases or their sub-phrases (e.g. head-noun of a noun group), etc. The "interesting" events usually also specify the relation between those building blocks such as "the two words should occur next to each other or in the same sentence". In this paper we describe a workbench for uncovering that kind of internal structure in natural language texts.

2 Data Level Integration

The underlying idea behind our workbench is data level integration of abstract data processing tools by means of structured streams. The idea of using an open set of modular tools with stream input/output (IO) is akin to the philosophy behind UNIX. This allows for localization of specific data processing or manipulation tasks so we can use different combinations of the same tools in a pipeline for fulfilling different tasks. Our architecture, however, imposes an additional constraint on the IO streams: they should have a common *syntactic* format which is realized as SGML markup (Goldfarb, 1990). A detailed comparison of this SGML-oriented architecture with more traditional data-base oriented architectures can be found in (McKelvie et al., 1997).

As a markup device an SGML element has a label (L), a pre-specified set of attributes (attr) and can have character data:

```
<L attr=val .. attr=val>character data</L>
```

SGML elements can also include other elements thus producing tree-like structures. For instance, a document can comprise sections which consist of a title and a body-text and the body-text can consist of sentences each of which has its number stated as an attribute, has its contents as character data and can include other marked elements such as pre-tokenized phrases and dates as shown in Figure 1. Such structures are described in *Document Type Definition* (DTD) files which are used to check whether an SGML document is syntactically correct, i.e. whether its SGML elements have only pre-specified attributes and include only the right kinds of other SGML elements. So, for instance, if in the document shown in Figure 1 we had a header element (H) under an S element – this would be detected as a violation of the defined structure.

An important property of SGML is that defining a rigorous syntactic format does not set any assumptions on the semantics of the data and it is up to a tool to assign a specific interpretation to a particular SGML item or its attributes. Thus a tool in our architecture is a piece of software which uses an SGML-handling Application Programmer Interface (API) for all its data access to corpora and performs some useful task, whether exploiting markup which has previously been added by other tools, or itself adding new markup to the stream(s) and without destroying the previously added markup. This approach allows us to remain entirely within the SGML paradigm for corpus markup while allowing us to be very general in designing our tools, each of which can be used for many purposes. Furthermore, through the ability to pipe data through processes, the UNIX operating system itself provides the natural "glue" for integrating data-level applications together.

The API methodology is very widely used in the software industry to integrate software components to form finished applications, often making use of some *glue* environment to stick the pieces together (e.g. tcl/tk, Visual Basic, Delphi, etc.). However, we choose to integrate our applications at the data level. Rather than define a set of functions which can be called to perform tasks, we define a set of representations for how information which is typically produced by the tasks is actually represented. For natural language processing, there are many advantages to the data level integration approach. Let us take the practical example of a tokenizer. Rather than provide a set of functions which take strings and return sets of tokens, we define a tokenizer to be something which takes in a SGML stream and returns a SGML stream which has been marked up for tokens. Firstly, there is no direct tie to the process which actually performed the markup; provided a tokenizer adds a markup around what it tokenizes, it doesn't matter whether it is written in C or LISP, or whether it is based on a FSA or a neural net. Some tokenization can even be done by hand, and any downline application which uses tokenization is completely functionally isolated from the processes used to perform the tokenization. Secondly, each part of the process has a well-defined image at the data level, and a data-level semantics. Thus a tokenizer as part of a complex task has its own semantics, and furthermore its own image in the data.

3 Queries and Views

SGML markup (Goldfarb, 1990) represents a document in terms of embedded elements akin to a file structure with directories, subdirectories and files. Thus in the example in Figure 1, the document comprises a header and a body text and these might require different strategies for processing[1]. The SGML-handling API in our workbench is realized by means of the LT NSL library (Thompson et al., 1996) which can handle even the most complex document structures (DTDs). It allows a tool to read, change or add attribute values and character data to SGML elements and address a particular element in a normalized[2] SGML (NSGML) stream using its partial description by means of *nsl-queries*. Consider the sample text shown in Figure 1. Given that text and its markup, we can refer to the second sentence under a BODY element which is under a DOC element: `/DOC/BODY/S[n=2]`. This will sequentially give us the second sentences in all BODYs. If we want to address only the sentence under the first BODY we can specify that in the query: `/DOC/BODY[0]/S[n=2]`. We can use wildcards in the queries. For instance, the query `.*/S` says "give me all sentences anywhere in the document" and the wildcard ".*" means " at any level of embedding". Thus we can directly specify which parts of the stream we want to process and which to skip.

Using nsl-queries we can access required SGML elements in a document. These elements can have, however, quite complex internal structures which we might want to represent in a number of different ways according to a task at hand. For instance, if we want to count words in the corpus and these words are marked with their parts of speech and base forms such as
`<W pos=VBD l=look>looked</W>`
we should be able to specify to a counting program which fields of the element it should consider. We might be interested in counting only word-tokens themselves and in this case two word-tokens "looked" will be counted as one regardless whether they were past verbs or participles. Using the same markup we can specify that the "pos" attribute should be considered as well, or we can count just parts-of-speech or lemmas. A special *view pattern* provides such information for a counting tool. A view pattern consists of names of the attributes to consider with the symbol # representing the character data field of the element. For instance:

- {#} – this view pattern specifies that only the character data field of the element should be considered ("looked");
- {#}{pos} – this view pattern says that the

[1]For instance, unlike common texts, headers often have capitalized words which are not proper nouns.

[2]There are a number of constraints on SGML markup in its normalized version.

```
<DOC>
  <H>This is a Title</H>
  <BODY>
   <S n=1>This is the first sentence with character data only</S>
   <S n=2>There can be  sub-elements such as <W pos=NN>noun groups</W> inside sentences.</S>
 </BODY>
 <H>This is another Title</H>
  <BODY>
   <S n=1>This is the first sentence of the second section</S>
   <S n=2>Here is a marked date <D d=1 m=11 y=1996>1st October 1996</D> in this sentence.</S>
 </BODY>
</DOC>
```

Figure 1: SGML marked text.

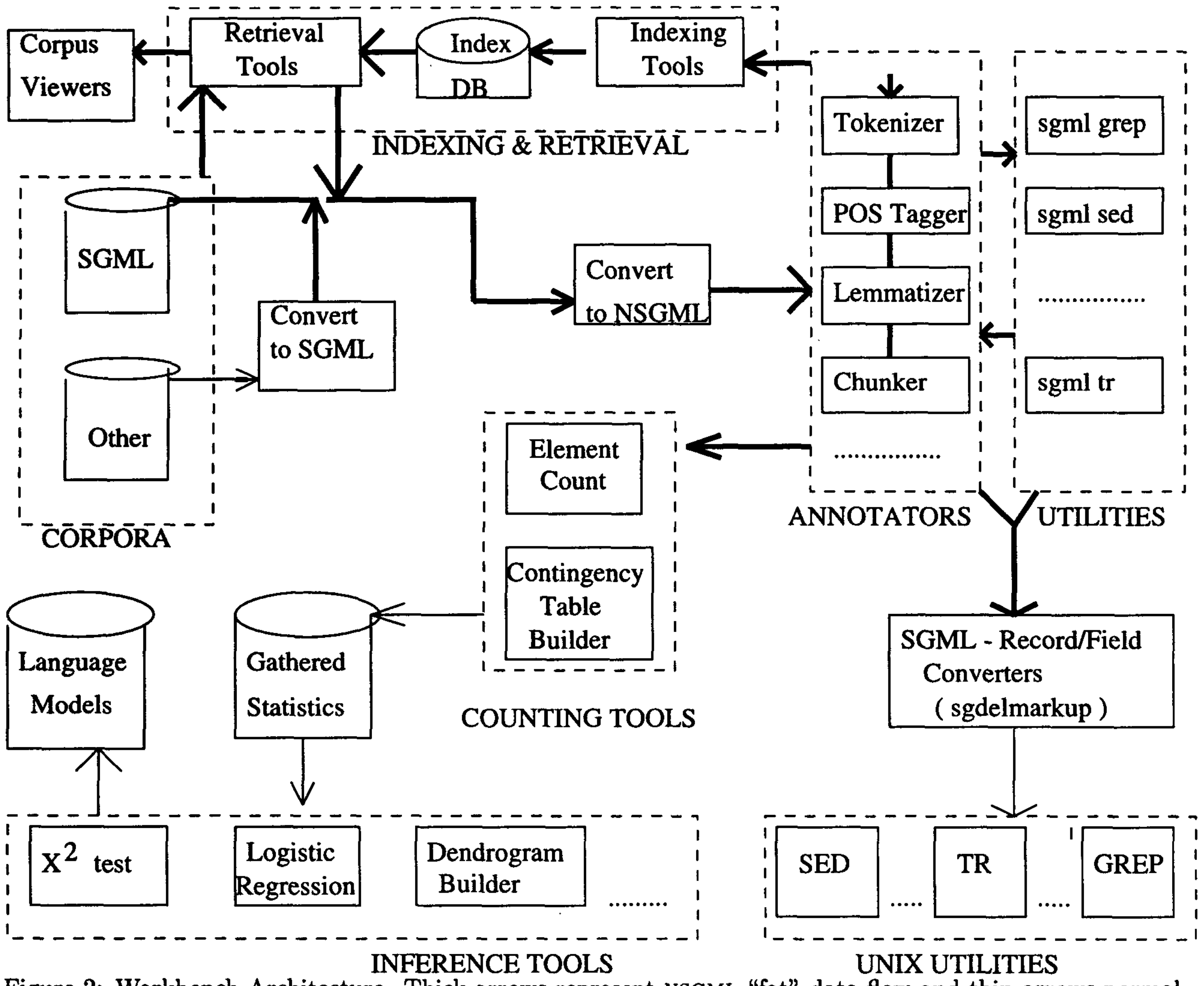

Figure 2: Workbench Architecture. Thick arrows represent NSGML "fat" data flow and thin arrows normal (record/field) data flow.

character data and the value of the "pos" attribute should be considered ("looked/VBD");

- {l} – this view pattern says that only the lemmas will be counted ("look");

4 The Workbench

Using the idea of data level integration the workbench described in this paper promotes the idea of independence of the text representation and the text analysis phases. At the representation phase the text is converted from a sequence of characters to features of interest by means of the annotation tools. At the analysis phases those features are used by the tools such as statistics gathering and inference tools for finding significant correlations in the texts. The analysis tools are independent of particular assumptions about the nature of the feature-set and work on the abstract level of feature-elements which are represented as SGML items. Figure 2 shows the main modules and data flow between them.

At the first phase documents are represented in an SGML format and then converted to the normalized SGML (NSGML) markup. Unfortunately there is no general way to convert free text into SGML since it is not trivial to recognize the layout of a text; however, there already is a large body of SGML-marked texts such as, for instance, the British National Corpus. The widely used on WWW format – HTML – is based on SGML and requires only a limited amount of efforts to be converted to strict SGML. Other markup formats such as LATEX can be relatively easily converted to SGML using publicly available utilities. In many cases one can write a perl script to convert a text in a known layout, for example, Penn Treebank into SGML. In the simplest case one can put all the text of a document as character data under, for instance, a `DOC` element. Such conversions are relatively easy to implement and they can be done "on the fly" (i.e. in a pipe), thus without the need to keep versions of the same corpus in different formats. The conversion from arbitrary SGML to NSGML is well defined and is done by a special tool (`nsgml`) "on the fly".

The NSGML stream is then sent to the *annotation tools* which convert the sequence of characters in specified by the nsl-queries parts of the stream into SGML elements. At the annotation phase the tools mark up the text elements and their features: words, words with their part-of-speech, syntactic groups, pairs of frequently co-occuring words, sentence length or any other features to be modelled.

The annotated text can be used by other tools which rely on the existence of marked features of interest in the text. For instance, the statistic gathering tools employ standard algorithms for counting frequencies of the events and are not aware of the nature of these events. They work with SGML elements which represent features we want to account for in the text. So these tools are called with the specification of which SGML elements to consider, and what should be the relation between those elements. Thus the same tools can count words and noun-groups, collect contingency tables for a pair of words in the same sentence or for a pair of sentences in the same or different documents. For instance, for automatic alignment we might be interested in finding frequently co-occuring words in two sentences, one of which is in English and the other one in French. Then the collected statistics are used with the standard tools for statistical inferences to produce desirable language models. The important point here is that neither statistics gathering nor inference tools are aware of the nature of the statistics – they work with abstract data (SGML elements) and the semantics of the statistical experiments is controlled at the annotation phase where we enrich texts with the features to model.

4.1 Text Annotation Phase

At the text annotation phase the text as a sequence of characters is converted into a set of SGML elements which will later be used by other tools. These elements can be words with their features, phrases, combinations of words, length of sentences, etc. The set of annotation tools is completely open and the only requirement to the integration of a new tool is that it should be able to work with NSGML and pass through the information it is not supposed to change. An annotation tool takes a specification (nsl-query) of which part of the stream to annotate and all other parts of the stream are passed through without modifications. Here is the standard set of the annotators provided by our workbench:

`sgtoken` - *the tokenizer* (marks word boundaries). Tokenization is at the base of many NLP applications allowing the jump from the level of characters to the level of words. `sgtoken` is built on a deterministic FSA library similar to the Xerox FSA library, and as such provides the ability to define tokens as regular expressions. It also provides the ability to define a priority amongst competing token types so one doesn't need to ensure that the token types defined are entirely distinct. However of greatest interest to us is how it interfaces with the NSGML stream. The arguments to `sgtoken` include a specification of the FSA to use for tokenization (there are several pre-defined ones, or the user can define their own in a perl-like regular expression syntax), an nsl-query which syntactically specifies the part of the source stream to process, and specification of the markup to add. The output of the process is an NSGML data stream which contains all the data of the input stream (in the same order as it appears in the input stream) together with additional markup which tokenizes those parts of the input stream which are specified by the nsl-query parameter. A call

```
sgtoken -q /DOC/BODY/S <== what to tokenize
        -s     <== use standard tokenizer
        -m W      <== markup tokens as W
```

can produce an output like:

```
<W>books</W><W>,</W> <W>for instance</W>
```

This gives rise to a simple data-level semantics — "everything inside a <W> element is a token added by **sgtoken**". However, this semantics is flexible. For example, if W markup is used for some other purpose, this markup might be changed to T markup.

`ltpos` – *a* POS-*tagger* (assigns a single part-of-speech (POS) to a word according to the context it was used). This is an HMM tagger akin to that described in (Kupiec, 1992). It receives a tokenized NSGML stream and instructions on what is the markup for words (word element label), where to apply tagging (nsl-query), and how to output the assigned information (attribute to assign). For instance, we might want to tag only the body-text of a document, and if the tokenizer marked up words as W elements we specify this to the tagger, together with the attribute that is to stand for the part-of-speech in the W element:

```
ltpos -q /DOC/BODY/.*/W <== path to words
      -m pos <== attribute to set with tag
      resource <== resources spec. file
```

This call will produce, for instance,

```
<W pos=NNS>books</W><W pos=CM>,</W>
<W pos=NNS>pens</W>
```

Here the "pos" attributes of the word elements "W" are set by the tagger to POS-tags: NNS – plural noun and CM - -comma. We can combine results produced by different taggers in different attributes which is useful for their evaluation.

`ltlem` – *the lemmatizer* (finds the base form for a word). The lemmatizer takes a stream with word elements together with their part-of-speech tags and further enriches the elements assigning a pre-specified attribute with lemmas, such as:

```
<W pos=NNS l=book>books</W><W pos=CM>,</W>
<W pos=NNS l=pen>pens</W>
```

`ltchunk` - syntactic *chunker* which determines the structural boundaries for syntactic groups such as noun groups and verb groups as, for instance:

```
 <NG>
   <W pos=DT>the</W>
   <W pos=JJ>good</W>
   <W pos=NNS l=man>men</W>
 </NG>
```

The chunker leaves all previously added information in the text and creates a structural element which includes the words of the chunk. The chunker itself is a combination of a finite state transducer over SGML elements with a grammar for syntactic groups similar in spirit to that of Fidditch (Hindle, 1983). This grammar can employ all the fields of the SGML elements. For instance a rule can say:"If there is an element of type "W" with character data "book" and the "pos" attribute set to "NN" followed by zero or more elements of type "W" with the "pos" attributes set to "NN" – create an "NG" element and put this sequence under it. The transducer itself is application independent – it rewrites the SGML streams according to a grammar stated in terms of SGML elements. It was, for instance, applied for the conversion of SGML markup into the LaTex markup.

The presented annotation tools are quite fast: the whole pipeline annotates at a speed of 1500-2000 words per second. Thus we never store the results of the annotation on disk, but annotate in the pipe shaping the annotation to the task in hand. Although the tools presented are deterministic and always produce a single annotation there is a way to incorporate multiple analyses into SGML structures using the *hyperlinks* described in (McKelvie et al., 1997). This initial set of annotation tools serves a wide range of tasks but one can imagine, for instance, a tool which adds the suffix of a word as its attribute `<W s=ed>looked</W>` or noun group length in words or characters `<NG wl=3 cl=10><W>...</NG>`. Another point to mention here is that it is quite easy to integrate another tagger or chunker or other tools as long as they obey the NSGML input/output conventions of the workbench or can be encased in a suitable "wrapper".

4.2 Counting Tools

After the annotation tools have been used to convert the text from a sequence of characters into a sequence of SGML elements, the counting tools can count different phenomena in a uniform way. Like the annotation tools, the counting tools take a specification of which part of the document to count things from (nsl-query) and they also take a specification of the *view* of an SGML element, i.e. which parts of those elements to consider for comparison.

The element counting program sgcount counts the number of occurrences of certain SGML elements in pre-specified fields of the stream according to a certain view. For instance, the call

```
sgcount -q /DOC/H/.*/W  -v {#}{pos}
```

will count frequencies of words occurring only in the titles of a document at any level of embedding (.*) and considering their character fields and the part-of-speech information. The call

```
sgcount -q /DOC/BODY/.*/W  -v {pos}
```

will produce the distribution of parts of speech in the BODY fields of documents.

To count joint events such as co-occurences of a word with other words, there is a tool for building contingency tables – sgcontin. A contingency table[3] records the number of times a joint event happened and the number of times a corresponding single event happened. For instance, if we are interested in the association between some two words, in the contingency table we will collect the frequency when these two words were seen together and when only one of them was seen. For instance, the call:

```
sgcontin -q  /DOC/BODY/W   -v  {#}
         -q2 /DOC/H/W  -v2 {#}
```

will give us a table of the associations between words in the body text of a document with the words in the title. Here, the program takes the query and the view for each element of a joint event. We can also specify the relative position of elements to each other. For instance, the call

```
sgcontin -q  /DOC/BODY/W    -v  {#}
         -q2 {q}/W[-1]      -v2 {#}
```

will build a contingency for a word with words to the left of it.

Both sgcount and sgcontin are extremely fast – they can process a million word corpus in a few minutes, so it is cheap to collect statistics of different sorts from the corpus.

4.3 The Inference Tools

The statistics gathered at the counting phase can be used for different kinds of statistical inferences. For instance, using mutual information (MI) or X^2 test on a contingency table we can find "sticky" pairs of items. Thus if we collected a contingency table of words in titles vs. words in body-texts we can infer which words in a title strongly imply certain words in a body text.

Using a contingency table for collecting left and right contexts of words we can run tests on similarity of the context vectors such as Spearman Rank Correlation Coefficient, Manhattan Metric, Euclidean Metric, Divergence, etc, to see which two words have the most similar context vectors and thus behave similarly in the corpus. Such similarity would imply closeness of those words and the words can be clustered in one class. The *dendrogram* tool then can be used to represent clustered words in a hierarchical classification.

There is a wide body of publicly available statistical software (StatXact, Cytel Software, etc) which can be used with the collected statistics for performing different sorts of statistical inferences and one can chose the test and package according to the task.

4.4 Indexing/Retrieval Tools

For fast access to particular locations in an SGML corpus we can index the text by using features of interest. Again, as in the case with the statistical tools, the indexing tools work on the level of abstract SGML elements and take as arguments which element should be indexed and what are the indexing units. For instance, we can index documents by word-tokens in their sentences, or by sentence length, or we can index sentences themselves by their tokens, or by tokens together with their parts-of-speech or by other features (marked by the annotation tools). Then we can instantly retrieve only those documents or sentences which possess the set of features specified by the user or another tool.

If, for instance, we index sentences by their words we can collect the collocations for a particular word or a set of words in seconds. The mkindex program takes an annotated NSGML stream and indexes elements specified by an nsl-query by their sub-elements specified by another nsl-query:

```
mkindex
   -dq  .*/BODY/S <== index all S in BODY
   -iq  .*/W  <= by Ws in these Ss
   -v   {#} <= using only character data of W
```

The "v" specifies the view of the indexing units. Such call will produce a table of indexing units (words in our case) with references (sequential numbers) to the indexed elements (sentences) they were found in. For instance:

```
book 23 78 96 584
```

says that word "book" was found in the sentences 23 78 96 and 584.

Next we have to relate (hook) the indexed elements (sentences) to the locations on disk. Note here that for the indexing itself we used sentences with annotations (they included W elements) but for hooking of these sentences to their absolute locations the annotation is not needed if we want to retrieve sentences as character data. The call:

```
MakeSGMLHook  -dq .*/BODY/S   filename.sgm
```

finds all sentences in the BODY elements of the file and stores their locations:

```
0 12344
1 33444
```

So sentence 0, for instance, starts from the offset 12344 in the file. To retrieve a sentence with a certain word (or set of words) we look up in which sentences this word was found and then look up the locations of those sentences in the file.

[3]Here we will talk only about two-way contingency tables but our tools can build n-way tables.

In some cases when the corpus to index already has a required annotation there is no need for our annotation tools. An example of such case is the British National Corpus (BNC). The BNC itself is distributed in SGML format with annotation, thus we used the indexing tools directly after the pipeline conversion into NSGML.

4.5 Utilities and Filters

One of the attractive features of data-level integration is the availability of a number of very flexible data manipulation and filtering utilities. For the record/field data format of the UNIX environment, such utilities include **grep, sed, count, awk, sort, tr** and so on. These utilities allow flexible and selective data-manipulation applications to be built by "piping" data through several utility processes. SGML is a more powerful data representation language than the simple record/field format assumed by many of these UNIX utilities, and consequently new utilities which exploit the additional power of the SGML representation format are required. We shall briefly describe the **sggrep** and **sgdelmarkup** utilities.

sggrep is an NSGML-aware version of **grep**. This utility selects parts of the NSGML stream according to whether or not a regular expression appears in character data at a specific place. As arguments, it takes two queries, the first (context query) tells it what parts of the stream to select or reject, and the second (content query) tells it where to look for the regular expression (within the context of the first). Optional flags tell the utility whether to include or omit parts of the stream falling outside the scope of the first query, or whether to reverse the polarity of the decision to include or exclude (the "-v" flag). For instance, the call:

```
sggrep -qx .*/SECT[t=SUBSECTION] <== context
       -qt .*/TITLE          <== content
       miocar.+[ ]+inf       <== regular expr.
```

will produce a stream of those SECT elements whose attribute "t" has the value SUBSECTION and which contain somewhere an element called TITLE with contiguous characters matched by the regular expression "miocar.+[ ]+inf" in the character data field.

sgdelmarkup is a utility which converts SGML elements into the record field format adopted by UNIX so the information can be further processed by the standard UNIX utilities such as **perl, awk, sed, tr**, etc. This tool takes an nsl-query as the specification which elements to convert and the view to the elements as the specification how to convert them. For instance, the call:

```
sgdelmarkup -q .*/W[pos=NN ] -v "{#} {l}"
```

will convert all nouns into "word lemma" format:

```
<W l=book pos=NN>books</W>  ==> books book
```

As an example of the combined functionality we can first extract from an NSGML stream elements of interest by means of **sggrep**, then convert them into the record-field format using **sgdelmarkup** and then sort them using the standard UNIX utility **sort**.

5 Putting it all together

Here we present a simple example of extracting and clustering the terminology from a medical corpus using the tools from the workbench. The PDS is a corpus of short Patient Discharge Summaries written by a doctor to another doctor. Usually such a letter comprises a structured header and a few paragraphs describing the admission, investigations, treatments and the discharge of the patient. We easily converted thes texts into SGML format with the structure PDS–HEADER–BODY writing a few lines of perl script. We did not keep a separate SGML version of the corpus and converted it "on the fly". Then we applied the annotation as described in section 4.1 thus marking up words with their lemmas and parts-of-speech and noun/verb group boundaries. Putting in this annotation is computationally cheap and we did it in the pipe rather than storing the annotated text on disk:

```
cat *.pds | nawk -f pds2sgml.awk | nsgml  |
 sgtoken -q ".*/BODY" -m W   |
  ltpos -q ".*/BODY/.*/W" -m pos |
   ltlem -q ".*/BODY/.*/W" -m l    |
    ltchunk -q ".*/BODY/S -m NG ng-gram |
      ltchunk -q ".*/BODY/S -m VG vg-gram
```

We annotated only the body-text of the summaries and as the result of the annotation phase we obtained sentence elements "S" with noun-group ("NG"), verb-group ("VG") and word ("W") elements inside, such as:

```
<S n=1>      -- sentence  N 1
  <NG>       -- start of noun group
    <W pos=DT>This</W>
    <W pos=CD>70</W>
    <W pos=NN>year</W>
    <W pos=JJ>old</W>
    <W pos=NN>man</W>
  </NG>        -- end of noun group
  <VG>         -- start of verb group
    <W pos=BED l=be>was</W>
    <W pos=VBN l=admit>admitted</W>
  </VG>        -- end of verb group
  <W pos=IN>from</W>
  ..........     -- other phrases and words
  <W pos=SENT>.</W>
</S>      -- end of sentence 1
```

Following (Justeson&Katz, 1995) we extracted terminological multi-word phrases as frequent multi-word noun groups. We collected all noun-groups with their frequencies of occurrences by running:

```
sgdelmarkup -q ".*/NG/W" -v {#} |
  sgcount -q /PDS/BODY/S/NG -v {#}
```

The sgdelmarkup call substituted all W elements in noun groups with their character data:

```
<NG><W pos=DT>the</W> <W pos=NN>man</W></NG>
===>
<NG>the man</NG>
```

and sgcount counted all NGs considering only their character fields. Here are the most frequent noun groups found in the corpus:

463	cardiac catheterisation
207	Happy Valley Hospital
144	ischaemic heart disease
114	Consultant Cardiologist
111	Isosorbide Mononitrate
108	the right coronary artery

Then we clustered the extracted terms by their left and right contexts. Using the sgcontin tool we collected frequencies of the two words on the left and two words on the right as the context. We also imposed a constraint that if there is a group rather than a word, we take only the head word: the last word in a noun group or the last verb in a verb group. As the result of such clustering we obtained four distinctive clusters: body-parts ("the right coronary artery"), patient-conditions ("70% severe stenosis"), treatments ("coronary bypass operation") and investigations ("cardiac catheterisation"). Unlike simple word-level clustering this clustering revealed some interesting details about the terms. For instance, the terms "left coronary artery" and "right coronary artery" were clustered together whereas "occluded coronary artery" was clustered with "occlusion" and "stenosis" thus uncovering the fact that it is of the "patient-condition" type rather than of the "body-part". A more detailed description of the process for uncovering corpus regularities can be found in (Mikheev&Finch, 1995).

6 Conclusion

In this paper we outlined a workbench for investigating corpus regularities. The important concept of the workbench is the uniform representation of corpus data by using SGML markup at the corpus annotation phase. This allows us to use the same statistics gathering and inference tools with different annotations for modelling on different features. The workbench is completely open to the integration of new tools but imposes SGML requirements on their input/output interface. The pipeline architecture of our workbench is not particularly suited for nice GUIs but there are publicly available visual pipeline builders which can be used with our tools. The tools described in this paper and some other tools are available by contacting the authors. Most of the tools are implemented in C/C++ under the UNIX environment and now we are porting them to the NT platform since it supports the pipes which are essential to our architecture.

7 Acknowledgements

This work was carried out in the HCRC Language Technology Group, partly with support from the UK EPSRC project "Text Tokenisation Tool". The HCRC is a UK ESRC funded institution.

References

S. Finch and N. Chater 1993. "Learning Syntactic Categories: a statistical approach." In M.R.Oaksford & G.D.A.Brown (eds) *Neurodynamics and Psychology*, pp. 295-322. London: Harcourt Brace&Co.

C.F. Goldfarb 1990. "The SGML Handbook." Oxford: Clarendon Press.

D. Hindle 1983. "User manual for Fidditch." Naval Research Laboratory Technical Memorandum #7590-142

J.S. Justeson and S.M. Katz 1995. "Technical terminology: some linguistic properties and an algorithm for identification in text." In *Journal for Natural Language Engineering* vol 1(1). pp.9–27. Cambridge University Press.

J. Kupiec 1992. "Robust Part-of-Speech Tagging Using a Hidden Markov Model." In *Computer Speech and Language*. pp.225-241. Academic Press Limited.

D. McKelvie, C. Brew and H. Thompson 1997. "Using SGML as a Basis for Data-Intensive NLP." In *Proceedings of the 5th Applied Natural Language Processing Conference (ANLP'97)*, Washington D.C., USA. ACL

A. Mikheev and S. Finch 1995. "Towards a Workbench for Acquisition of Domain Knowledge from Natural Language." In *Proceedings of the 7th Conference of the European Chapter of the Association for Computational Linguistics (EACL'95)*. Dublin. pp.194–201. ACL (CMP-LG 9604026)

F. Smadja 1993. "Retrieving Collocations from Text: Xtract." In *Computational Linguistics*, vol 19(2), pp. 143-177. ACL

H. Thompson, D. McKelvie and S. Finch 1996. "The Normalised SGML Library LT NSL version 1.4.6." *Technical Report*, Language Technology Group, University of Edinburgh. http://www.ltg.ed.ac.uk/software/nsl