Distributed Parse Mining

Scott A. Waterman, PhD
Microsoft Live Search/Powerset
475 Brannan St.
San Francisco, USA
waterman@acm.org

Abstract

We describe the design and implementation of a system for data exploration over dependency parses and derived semantic representations in a large-scale NLP-based search system at powerset.com. Because of the distributed nature of the document repository and the processing infrastructure, and also the complex representations of the corpus data, standard text analysis tools such as grep or awk or language modeling toolkits are not applicable. This paper explores the challenges of extracting statistical information and of building language models in such a distributed NLP environment, and introduces a corpus analysis system, *Oceanography*, that simplifies the writing of analysis code and transparently takes advantage of existing distributed processing infrastructure.

1 Introduction

In computational linguistics we deal with large corpora and vast amounts of data from which we would like to extract useful information. The size of the text resources, derived linguistic analyses, and the complexity of their representations is often a stumbling block on the way to understanding the statistical and linguistic behavior within the corpus. Simple software tools suffice for small or simple analysis problems, or for building models of easily represented relations. However, as the size of data, the intricacy of relations to be analyzed, and the complexity of the representation grow, so too does the technical difficulty of conducting the analysis.

Software is our given means of escape from this escalation of complexity. However, as "computational linguists," we often find ourselves spending more time and attention building software to perform the required *computations* than we do on understanding the linguistics.

Even once a suitable set of NLP tools (e.g. taggers, chunkers, parsers, etc.) has been chosen, analysis software, in the CL world, often consists of "throw away" scripts. Small, *ad hoc* programs are often the norm, often with no assurance (via strict design or testing) of correctness or completeness.

1.1 Oceanography

Our goal is to ensure that analysis is not so problematic. Powerset is a group within the Microsoft Live Search team focused on using semantic NLP to improve web search. We face many problems with the scale and integration of our NLP components, and are approaching solving them by applying sound software design and abstraction principles to corpus processing. By generalizing tools to fit the processing environment, and the nature of the problems at hand, we enable flexible processing which scales with the size of the platform and the data.

The *Oceanography* software environment is designed to address two important needs in large corpus analysis. The first is to simplify the actual programming of analysis code to reduce development time and increase reliability, and the second is to use the available distributed computing resources to reduce running time and provide for rapid testing and experimental turnaraound.

Proceedings of the NAACL HLT Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing, pages 56–64,
Boulder, Colorado, June 2009.

1.2 Linguistic and Diagnostic data analysis

There are two separate kinds of analysis we want to support over this processed corpus. The first is linguistic modeling. In order to achieve the best semantic interpretation of each source document, we seek to understand the linguistic behavior within the corpus. Probabilistic parsing, entity extraction, sense disambiguation, and argument role assignment are all informed by structured, statistical models of word behavior within the corpus. Some models can be built from simple tokenized text, while other models need to incorporate parse dependencies or real-word knowledge of entities. Some of these tasks are exploratory and underspecified (e.g. selectional restrictions), while others, such as name tagging, have a well-developed literature and a number of almost standard methodologies.

The second kind of analysis is aimed at characterizing and improving system behavior. For example, distributions of POS-tags or preposition attachments can serve as regression indicators for parser performance. In order to perform error analysis, we need to selectively sample various types of label assignments or parse structures. So summarization and sampling from the various intermediate NL analyses are very important processes to support.

2 Generalizing Text Mining

We have found that most of these analysis and data modeling tasks share certain higher order steps that allow us to generalize them all into a single programming framework. All involve identifying some phenomena in one of the NLP outputs, representing it in some characteristic form, and then summing or comparing distributions. These general steps apply to many corpus tasks, including building n-gram data, learning sentence breaks, identifying selectional preferences, or building role mappings for verb nominalizations.

The Oceanography system generalizes these steps into a declarative language for stating the selection of data, and the form of output, in a way that avoids repetitive and error prone boilerplate code for file traversal, regular expression matching, and statistics programming. By matching a declarative syntax to the general analysis steps, these common functions can be relegated to library code, or wrapped into the executable in the compilation step. The less time spent in describing a task, or in coding and debugging the implementation, the more time and attention can be spent in understanding the results and modeling the linguistic processes that underly the data.

This sort of abstraction away from the details of file representation, storage architecture, and processing model fits a general trend toward data mining, or *text mining* (Feldman and Dagan, 1995). In data mining or KDD systems (Fayyad et al., 1996), the goal is to separate the tasks of creative analysis and theorizing from the mundane aspects of traversing the data collection and computing statistics. These are much the same goals emphasized by Tukey (1977) – exploration of the data and interactions in order to understand which hypotheses, and which models of interaction, would be fruitful to explore. For our needs in analyzing collections of text, parses, and semantic representations, we have achieved a very practical step toward these goals.

2.1 Matching process to conception

We have found four steps that map very closely to our conception of the data analysis problem, which at the same time are easily translated to implementations that can be run on both small local data sets and on very large distributed corpora.

1. **Pattern matching –** find the interesting phenomena among the mass of data, by declaring a set of desired properties to be met. In Oceanography, these are matched per-sentence.

2. **Transformation –** rewrite the raw occurrence data to identify the interesting part, and isolate it from the background

3. **Aggregation –** group together instances of the same kind

4. **Statistics –** compute statistics for counts, relative frequency, conditional distributions, distributional comparisons, etc.

In the following sections we describe the nature of each step in more detail, map these steps to a declarative data analysis language, give some motivating examples, and describe how these steps are typically

accomplished in an exploratory setting for NLP investigations.

Later, in section 4, we describe how the steps are mapped to processing operations within the NLP pipeline architecture. Following that, we give examples of how this framework maps to specific problems, of both the exploratory and the diagnostic type.

2.2 Pattern Matching

The first step is to identify the specific phenomena of interest within the source data. If the data is a complex structure, it is helpful to express the patterns in a logical representation of the structure, rather than matching the representation directly.

Pattern matching in Oceanography for dependency parse structures is handled using a domain specific language (DSL) built explicitly for pattern-based manipulation of parse trees and semantic representations generated by the XLE linguistic components (Crouch et al., 2006). This *Transfer* language (Crouch, 2006) is normally used in the regular linguistic processing pipeline to incrementally rewrite LFG dependency parses into a role-labeled semantic representations (*semreps*) of entities, events, and relations in the text. Transfer matches pattern rules to a current set of parse dependencies or semantic facts, and writes alternate expressions derived from the matched components. Variables in these expressions are bound via Prolog-style unification (Huet, 1975).

For example, in figure 1, the first expression `word(···)` will match word forms in a parse that are `'verb'`s, and bind `%VerbSk` variable to a unique occurrence id and `%VerbWord` to the verb lemma. The second pattern finds the node in the dependency graph that fills the `ob` (object) role for that verb, and extracts its lemmas. (The `%%`'s are placeholder variables in the pattern, needed to match the arity of the expression.) Below, in the same figure, is a representation of the verb and object from a parse of the phrase "determined the structure". On matching these facts, the `VerbWord` and `ObjLemma` variables would be bound to the strings `determine` and `structure`.

In a simpler environment, with more basic textual representations, this pattern matching step would be written with regular expressions, for example using the familiar `grep` command. The balance provided by `grep` between the simplicity of its operational model (a transform from stdin to stdout) and the expressiveness of the regular expressions allows `grep` to be a workhorse for data analysis over text.

However, except for simple cases such as word cooccurrence models, the typical need in deep linguistic analysis is not well served by regular expressions over strings. Anyone in the NLP field who has written regular expressions to match, say, part-of-speech labeled text knows the difficulties of having a pattern language which differs from the logical structures being matched. Another typical solution is to write a short program in a scripting language (e.g. perl, python, SNOBOL) which combines regular expressions to provide a simple structure parser. Tgrep (Pito, 1993) is a one such program which extends this regular expression notion to patterns over trees, and can output subtrees matching those expressions, but only provided they are represented as text in the LDC TreeBank format.

2.3 Transformation

Once the items of the pattern have been identified in their original context, it is often necessary to isolate them from that context, and remove the extraneous, irrelevant information. For instance, if one is doing a simple word count, the tokenized words of text must be separated from any annotation and counted independently. For more complicated tasks, such as finding a verb's distribution of occurrence with direct objects, the verb and object need to be isolated from the remainder of the parse tree, perhaps as the simple tuple *(verb, object)*, or in a more complex structure, with additional dependent information.

In our case, we express the transformed output of each pattern match with an expression built from the unification variables bound to the match. In figure 2, we construct a `vo_pair` of *(verb, object)*. This new construct is simply added to the collection of facts and representations already present. All other pre-existing facts in the NL analysis of the sentence also remain in context, potentially available for aggregation and counting.

```
==> vo_pair(%VerbWord, %ObjLemma).
```

Figure 2: Transforming the matched pattern

```
word(%VerbSk, %VerbWord, verb, verb, %%, %%, %%, %% ),
in_context(%%, role(hier(ob, %%), %VerbSk, %ObjLemma:%%))
```

```
word(determine:n(41,3),determine,verb,verb, ....)
in_context(t,role(hier(ob,[[ob,root],..]),
          determine:n(82,3),structure:n(91,3))))
```

Figure 1: Pattern matching using Transfer

In shallower text mining, this might be accomplished using regex matching in a perl program. Another common approach is to use command-line text tools such as `awk` or `sed`. `Awk` (Aho et al., 1977) is designed especially for text mining, but is limited to plain text files, on single machines, and doesn't extend easily to structured graph representations or distributed processing. (But see, e.g. Sawzall (Pike et al., 2005) for a scalable awk-like language.)

2.4 Aggregation

The aggregation step collects the extracted instances and groups them by type and by key. Rather than have the matched, transformed results simply dumped out in some enormous file or database in their order of occurrence in the data set (as one would get e.g. from `grep`), it is quite useful even in the simplest of cases to aggregate all similar output items. This condenses the mass of data selected, and allows one to see the extent and diversity of the items that are found by the patterns. This simple counting is often enough for diagnostic tasks, and sometimes for exploratory tasks when a statistical judgement is not yet desired. The aggregation key might be, for various kinds of extraction: the head noun of an NN-compound, or the error type for parse errors, or the controlling verb of a relative clause.

In Oceanography, we require a declaration of the data that will be aggregated, in order to separate it from the remainder which will be discarded. These declarations take the form of familiar static type declarations, in the style of C++ or Java. Figure 3 shows the simple declaration for our `vo_pair` type, where both fields are declared as strings. These named fields also provide a handle to refer to structure members in later statements.

In the command line text world, aggregation might be accomplished by using the unix pipeline

```
vo_pair :: {
    verb::String, object::String }
```

Figure 3: Declaring aggregation types

command `sort | uniq -c` , to organize the output by the appropriate key. If using a small program to do this kind of analysis, one would use a dictionary or hash-table and sorting routines to organize the data before output.

2.5 Statistics

With the matched and extracted data, one can build up a statistical picture of the data and its interrelations. In our practice, and in the computational NLP literature, we have found a few fundamental statistical operations that are frequently used to make sense of the corpus data. Primary among these are simple class counts: the number of occurrences of a given phenomena. For instance, the count of part-of-speech tags, or of head nouns with adjective modifiers, or the counts of *(verb,object)* pairs. These counts can be computed easily by summing the occurrences in the aggregated groups.

Other statistics are more complicated, requiring combinations of the simple counts and sums — normalizing distributions by the total occurrence counts, for instance, as in the conditional occurrence of a part-of-speech label relative to the frequency of the token. Estimation of log-likelihood ratios or Pearson's Chi-square test for pairwise correlation also falls in this category. These kinds of computations are used heavily for building classifiers and for diagnostic purposes.

Higher order functions of the counts are also interesting, in which various distributions compared. These include computing KL distance between conditional distributions for similarity measurements,

clustering over similarity, and building predictive or classification models for model corpus behavior.

3 Data Parallel Document Processing at Powerset

To simplify the processing of large web document collections, and flexibly include new processing modules, we have built a single consistent processing architecture for the natural language document pipeline, which allows us to process millions of documents and handle terabytes of analysis data effectively. *Coral* is the name of the distributed, document-parallel NLP pipeline at Powerset. Coral provides both a process and a data management framework in order to smoothly execute the multi-step linguistic analysis of all content indexed for Powerset's search.

Coral controls a multi-step pipeline for deep linguistic processing of documents indexed for search. A partial list of the steps every web document undergoes includes: HTML destructuring, sentence breaking, name tagging, parsing, semantic interpretation, anaphora resolution, and indexing. The pipeline is similar to the UIMA (Ferrucci and Lally, 2004) architecture in that each step adds intermediate data — tagged spans, dependency trees, coreferent expressions — that can be used in subsequent steps. Each step adds a different kind of data to the set, with its own labels and meanings. The output of all these steps is a daunting amount of information, all of which is valuable for understanding the linguistic relations within the text, and also the behavior and effectiveness of the NLP pipeline.

Documents are processed in a data-parallel fashion. Multiple documents are processed independently, across multiple processes on multiple compute nodes within a clustered environment. The document processing model is sequential, with multiple steps run in a fixed sequence for each document in the index. All processing for a single document is typically performed on a single compute node. The steps of the pipeline communicate through intermediate data writen to the local filesystem in between steps, where each step is free to consume data produced earlier. Output from the stages is checkpointed to backing storage at various points along the way, and the final index fragments are merged at the end.

This kind of data-parallel process lends itself well to a map/reduce programming infrastructure (Dean and Ghemawat, 2004). Map/reduce divides processing into two classes: data-parallel 'map' operations, and commutative 'reduce' operations, in which all map output aggregated under a particular key is processed together. In map/reduce terms, the entire linguistic processing runs as a sequence of 'map' steps (there is no inter-document communication), with a final 'reduce' step to collect index fragments and construct a unified search index. Coral uses the open-source hadoop implementation of map/reduce (Cutting,) as the central task control and distribution mechanism for assigning NLP pipeline jobs to documents in the input data, and it has full control of the map/reduce processing layer.

3.1 Difficulties for data mining in Coral

All of the intermediate processing output of the pipeline, the name tags, parses, semantic representations, etc., are are retained by this complex process. Unfortunately, they are retained in an unfriendly format: small document-addressed chunks scattered across a large distributed filesystem, on hundreds of machines. There is no operational way to collect these chunks in any single file, or to traverse them efficiently from any single point. Traditional scripting techniques, even if scalable to the terabytes of data, are not applicable to the distributed organization of the underlying data.

3.2 Re-using processing infrastructure for mining

However, we can re-use the same Coral process and data management for the problems of data analysis. The breakdown of parse-mining steps presented earlier, in addition to providing a coherent model for data analysis, also maps very cleanly to the distributed map/reduce computational model. By translating the four steps of any analysis into corresponding map/reduce operations across the linguistic pipeline data, we can efficiently translate the corpus analytics to an arbitrarily large data setting. Further, because we can rely on the Coral process and data management infrastructure to handle the data movement and traversal, we allow the researcher or language engineer to concentrate on specifying the

patterns and relations to be investigated, rather than burdening them with complex yet necessary details.

4 Oceanography - a compiled data mining language

Oceanography has a compiler that transforms short analysis programs into multiple map/reduce steps that will operate over the corpus of text and deep linguistic analyses. These multiple sub-operations are then farmed out through the distributed cluster environment, managed by the Coral job controller. The data flow and dependencies between these jobs are compiled to a Coral-specific job control language.

An oceanography program (cf. figure 4) is a single-file description of the data analysis task. It contains specifications for each of the four operations: pattern matching, transformation, aggregation, and statistics. The program style is declarative – there are no instructions for iterating over files, summing distributions, or parsing the dependency graph representations.

We find that this matches our intuitions and conception of the parse mining task. A statement of the end-product of the analysis is natural: e.g. find the conditional distribution of object head nouns for verbs, or symbolically $p(obj|verb)$. The style of the oceanography program matches this well, where the statistics statement such as

```
dist triple.object cond on triple.verb
```

states the desired output, and the preceding pattern match and type declarations serve as definitions to specify precisely what is meant by the variable names.

In the following sections, we will follow the steps of the Oceanography program in the listing in figure 4. The example analysis presented is a simple one – to find all verbs with both subject and object roles, i.e. triples of *(subject, object, verb)*, and report some counts and relative frequencies of verbs, subjects, and objects.

4.1 Step 1: Pattern Matching

The pattern matching rules are similar to those presented above in sec. 2.2. The first line matches a verb term, and the next two lines require the presence of terms in both the subject (`role(hier(sb, %%))`) and object `role(hier(ob, %%))` roles. Following the explicit pattern expression, we add negative checks to ensure that neither the subject or object are PRO elements, which have no surface form.

4.2 Transformation

The transformation expressed in figure 4 is almost trivial. We capture the verb-subject-object triple in a simple three place predicate. Recall that the values of the triple:

```
(%VerbWord, %SubjLemma, %ObjLemma)
```

are bound by unification to the terms matched in the pattern, above.

Although we have only one pattern and one matching transformation in this example, we are not in general limited in the number of patterns or output expressions we might use. Multiple transforms, from multiple patterns, can be used.

During compilation, these Transfer rules are compiled into a binary object module, then distributed at runtime to the compute nodes where they will be executed in the proper sequence by the Coral job controller. Output from the transformation step, and between all the steps, is encoded as a set of hierarchically structured objects using JSON (Crockford, 2006). Because JSON provides a simple structural encoding with named fields, and many programming environments can handle the JSON format, it provides a flexible and self-describing interchange format between steps in the Oceanography runtime.

4.3 Aggregation

The third section of the Oceanography program declares the types of objects to be aggregated following the transform step. The type declarations in this section serve two purposes. First, they specify exactly what types of data from the matching/transformation phase should be carried forward. Recall that all of the source data is available for processing, but we are likely only interested in a small portion of it. Secondly, the declarations serve as type hints to the compiler so that operations and data storage are performed correctly in the later phases (e.g. adding strings vs. integers).

4.4 Statistics

The simplest statistic we can compute is the count of a type that has been aggregated. For example,

```
## Step 1: pattern matching
rules {
    word(%VerbSk, %VerbWord, verb, verb, %%Pos, %%SentNum, %%Context, %%LexicalInfo ),
    in_context(%%, role(hier(sb, %%), %VerbSk, %SubjLemma:%%)),
    in_context(%%, role(hier(ob, %%), %VerbSk, %ObjLemma:%%)),
    { \+memberchk( %SubjLemma, [group_object, null_pro, agent_pro]),
      \+memberchk( %ObjLemma, [group_object, null_pro, agent_pro])  }
## Step 2: Transformation
    ==> triple(%VerbWord, %SubjLemma, %ObjLemma).
}
## Step 3: Aggregation
triple :: {
    verb :: String,
    subject :: String,
    object :: String
}
## Step 4: Statistics
count triple
count triple.verb
count triple.verb, triple.subject
dist triple.object cond on triple.verb
```

Figure 4: A complete Oceanography program

`count triple.verb`

will result in occurrence counts of each verb seen in the parses. We can combine primitive types into tuples, in order to count n-grams (which are not necessarily adjacent), e.g.

`count triple.verb, triple.subject`

to give occurrence counts for all (verb,subject) pairs.

The `dist X cond on Y` statement is used to produce the conditional distribution $p(x|y)$. The map/reduce framework collates all occurrences with a given value y_i to a single reduce function, which sums the conditional counts of x, and normalizes by the total.

Other statistics require multiple map/reduce operations. Computing the probability for the verb unigrams requires knowing the total number of occurrences, which, in this kind of data-parallel processing architecture, is not available until the output of all occurrence counts is known. So, a `prob triple.verb` statistic must implicitly compute `count triple.verb`, sum all occurrences, and normalize across the set. For a good type-driven analysis of information flow during various stages of a map/reduce computations, see Lämmel (2008).

4.5 Output

Output is given two forms. For ease of interpretations, human-readable tab delimited files are written, in which each record is preceded by the type, as given in the argument to the statistics declaration. To simplify later offline computation, the record can also be written out in a JSON encoded structure with named fields corresponding to the type.

5 Development and testing in Oceanography

Rapid turnaround and testing in exploratory corpus analytics is essential to understanding the nature of the data, and the performance and behavior of one's program. Because the tools on which Oceanography is built are modular, we can compile an analysis program for a local, single machine target as easily as we can for a cluster of arbitrarily many compute nodes. The resulting compiled programs differ somewhat in the ways they traverse the data, and in the control structures for the Coral processing steps. However, it was an important design requirement that we could compile and test using small data on a single machine as easily as on a muti-terabyte corpus on a distributed cluster.

The same source program is compiled for either single machine or cluster execution. The user must specify a different type of store location for input and output data, depending on environment. Compilation is done using a command line program, which takes as input the Oceanography program, and produces a set of executable outputs, corresponding to the tasks in the map/reduce process. These can also be run immediately in the single machine setting, with results going to stdout.

5.1 Some sample tasks

Although these tools have been available at Powerset only a few months, we have already used them to great advantage in diagnostic and linguistic analysis tasks. Diagnostically, it is important to understand the failure modes of the various linguistic pipeline components. For instance, the morphological analysis component of the XLE parser will on occasion encounter tokens it cannot analyze. Hand-examining a few hundred parses (which starts to exceed the mental fatigue threshold), one can find numerous examples. But one has no idea of the relative frequency of any given type of error, or their combined effect on the parse output. Oceanography enables a very simple single pattern match rule to be used to find the frequency distribution of unknown tokens over 100M sentences as easily as 100, and the grammar engineers can use this information to prioritize their effort. Other diagnostics on the parse, such as the frequency of certain rare grammatical constructs (e.g. reduced relatives), or the prevalence of unparseable fragments, or relative frequencies of transitive v. intransitive use, are immensely important for understanding the nature of the corpus and the behavior of the parser.

The S-V-O triples used as an example also have practical import. By identifying the most common verb expressions, we can, just as in a keyword stop list, eliminate or downweight some of the less meaningful relations in our semantic index. For example, in the Wikipedia corpus, one of the most common S-V-O triples comes from the phrase "this article needs references."

We are also beginning a series of lexical semantic studies, looking at selectional preferences and their dependence on surface form. Correspondence between prepositional adjunct roles and other surface realizations is also an active area. Additionally, Oceanography is being used to analyze feature data from the parses in order to experiment with an unsupervised word sense disambiguation project.

6 Conclusion

We have presented a methodology for understanding a certain class of linguistic data analysis problems, which identifies the steps of pattern matching, data transformation, aggregation, and statistics. We have also presented a programming system, Oceanography, which by following this breakdown simplifies the programming of these tasks while at the same time enabling us to take advantage of existing large scale distributed processing infrastructure.

Acknowledgments

I would like to thank Jim Firby, creator of the Coral document processing pipeline at Powerset, and Dick Crouch, creator of the XLE Transfer system, for their foundational work which makes these present developments possible.

References

Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan. 1977. awk.

D. Crockford. 2006. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July.

Richard S. Crouch, Mary Dalrymple, Ronald M. Kaplan, Tracy Holloway King, John Maxwell, and P. Newman. 2006. XLE documentation.

Richard S. Crouch. 2006. Packed rewriting for mapping text to semantics and KR.

Doug Cutting. Apache Hadoop Project. http://hadoop.apache.org/.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, Berkeley, CA, USA. USENIX Association.

Usama M. Fayyad, David Haussler, and Paul E. Stolorz. 1996. KDD for Science Data Analysis: Issues and Examples. In *KDD*, pages 50–56.

Ronen Feldman and Ido Dagan. 1995. Knowledge Discovery in Textual Databases (KDT). In *KDD*, pages 112–117.

David Ferrucci and Adam Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4):327–348.

Grard P. Huet. 1975. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci*, 1:27.

Ralf Lämmel. 2008. Google's MapReduce programming model - Revisited. *Sci. Comput. Program.*, 70(1):1–30.

Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298.

Richard Pito. 1993. Tgrep.

John Wilder Tukey. 1977. *Exploratory Data Analysis*. Addison-Wesley, New York.