RC22122 (W0107-020) July 20, 2001
Computer Science

IBM Research Report

Construction of an OO Framework for Text Mining

James W. Cooper, Edward C. So, Christian L. Cesar,
Robert L. Mack
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Construction of an OO Framework for Text Mining

James W. Cooper, Edward C. So, Christian L. Cesar, and Robert L. Mack

IBM T J Watson Research Center, PO Box 704, Yorktown Heights, NY 10598

jwcnmr@watson.ibm.com

Abstract

We describe how we constructed a Java class library for text mining and information retrieval. The system consists of Facades around a database, a search engine and a text mining tool. We discuss the design of the object models we used for each of these elements and how they evolved as different databases and search engines became available. Then we discuss how we needed to evolve the system further in work with our customer. Finally we discuss the eventual fate of the system after the customer adopted the final version of the code, showing what we learned from the experience.

Introduction

We began building Java applications for text mining and information retrieval in 1995, shortly after Java was released. We reported on the first application interface in 1997, discussing the advantages of Prompted Query Refinement [1] to help users focus their queries. The initial system was built using a Unix-based internal research level search engine and a set of binary tables for term lookups. The communication between the Java applet and the server was provided using a low-level socket interface.

We then embarked on a more ambitious project, working with the Giga Information group, to index their consultant reports and provide a more sophisticated way for their user to search and review these reports. This system consisted of a Windows NT server, an IBM product search engine and a database of the terms and relations discovered by our TALENT [2] text mining tools. In this initial modest-sized collection, we used a Microsoft Access database and wrote a Java RMI server to provide information to a Java applet client [3]. In a follow-on project, we extended this design further to include major sentences in each document, which could be used to construct a variable length document summary [4].

While this system provided a fairly compelling demonstration of capabilities, it was not scalable to large numbers of users or larger amounts of data. Further, the construction of the data indexes and the construction of the client and server were all disparate processes, written in different styles and languages. For example, the Access database was constructed from the text files produced by the TALENT text mining tools using a Visual Basic program. Further, even though the client and server were both written in Java, they evolved incrementally and did not make effective use of good OO concepts that would make it easy for new user to pick up this technology and use it in their own work. Thus, we began the design and development of a set of Java class libraries containing objects for manipulating all of the information involved in text mining, search, and query refinement. This report describes the evolution of this library, pitfalls in its construction, how the code was eventually transferred to a product division, and how that division received and used it. While the report is not a cautionary tale, we did find that practices most common in the research context are often viewed quite differently in the product context.

Objectives of the System

When the system was completed, the objective was to provide a toolkit that make it easy for a reasonably competent Java programmer with only minimal information retrieval experience to write code to index and text mine a document collection and produce a server which could support text queries and display relations between terms in the collection. A typical query page and results is shown below in Figure 1.

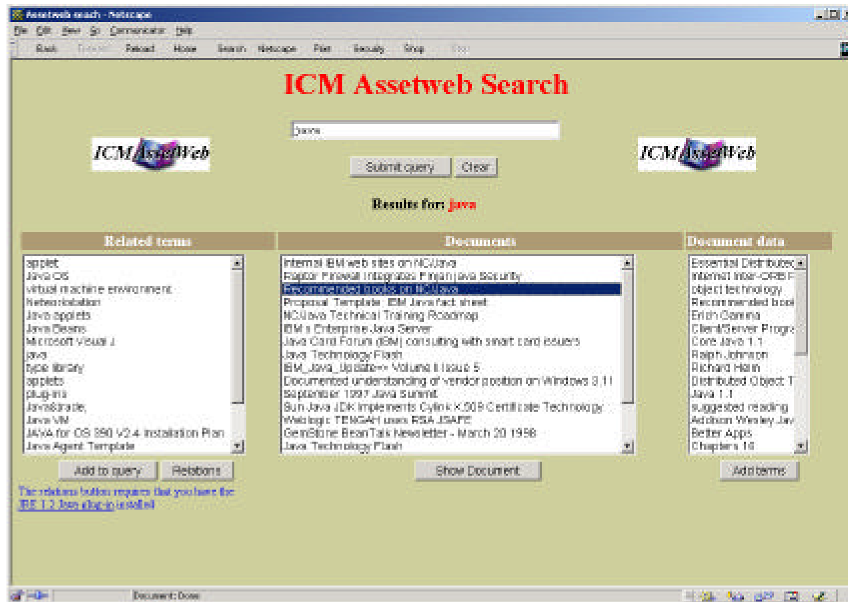

Figure 1 - A query page containing the results of searching. Shown is the Context Thesaurus (left), the document index (middle). and the major terms found in the selected document (right).

We can also mine and represent query and represent relations between terms as illustrated in Figure 2. Here we used a Java applet which communicates with the server using SOAP RPC methods.

Figure 2 – Terms related to Design Patterns in the IBM Global Services consultant report collection.

Discussion of our text mining software tools

The TALENT (Text and Language Engineering Tools) project in our text mining group consists of a suite of programs for recognizing major terms in a document and aggregating these terms into a collection vocabulary. The Textract program recognizes technical terms, proper names, organizations, and places. It also determines named relations between terms using English language patterns such as appositives, parentheticals and comma-separated lists, and unnamed relations based on proximity [5].

Finally, it produces a concordance of all the sentences in the collection that contain these major, salient terms. This concordance is then used in conjunction with a search engine to create a context thesaurus index of terms that co-occur near the term you specify. This is related to the Phrase Finder work of Xu and Croft [6]. We show a block diagram of these elements in Figure 3.

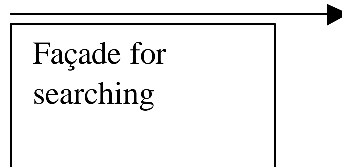

Figure 3 –Block diagram of the elements in the text mining system.

Objects used in indexing documents

We began by defining the major objects in our system. When applying our text mining tools, we typically started with a collection of HTML documents and indexed them using a search engine to produce a searchable free text index. Then, we ran the Textract program and associated TALENT tools on this collection to produce a collection-wide vocabulary, a vocabulary of each document, a set of named and unnamed relations between terms and a second search index of terms related by proximity which we have called the Context Thesaurus.

The major data objects we needed to contain in programming objects were then a terms table, a document table, a document-terms correlation table, a relations table, a relation names table, a document free-text index and a Context Thesaurus free-text index. Many of these objects map well into a relational database, as we found in the original VB-Access design, but here we wanted to use Java classes throughout. The objects and methods in Java for accessing and manipulating databases are a bit complex and low level for the needs of this project, and accordingly we designed a Façade [7] around the database classes to derive a simpler interface, as we described earlier [8]. In essence, we reduced the java.sql package classes to ones called AbstractDatabase, KSSResultSet, SQLQuery and PreparedQuery. We also added an AbstractDatabaseTable class to be used in creating tables other than the ones we required for this system.

Since we wanted to be able to test this design on small systems, but still be able to scale it later, we derived two complete packages from the classes in the above Database package, one for Access and one for IBM DB2. While the underlying JDBC methods were probably going to be the same for both databases, we wanted to allow for customization for each database system and for others that might be developed later. In the derived packages we created a class for each major database table:

- Database
- Documents
- SQLQuery
- KSSResultSet
- Terms
- TermDocs
- TermRelations
- RelationNames

The class hierarchy is shown in Figure 4.

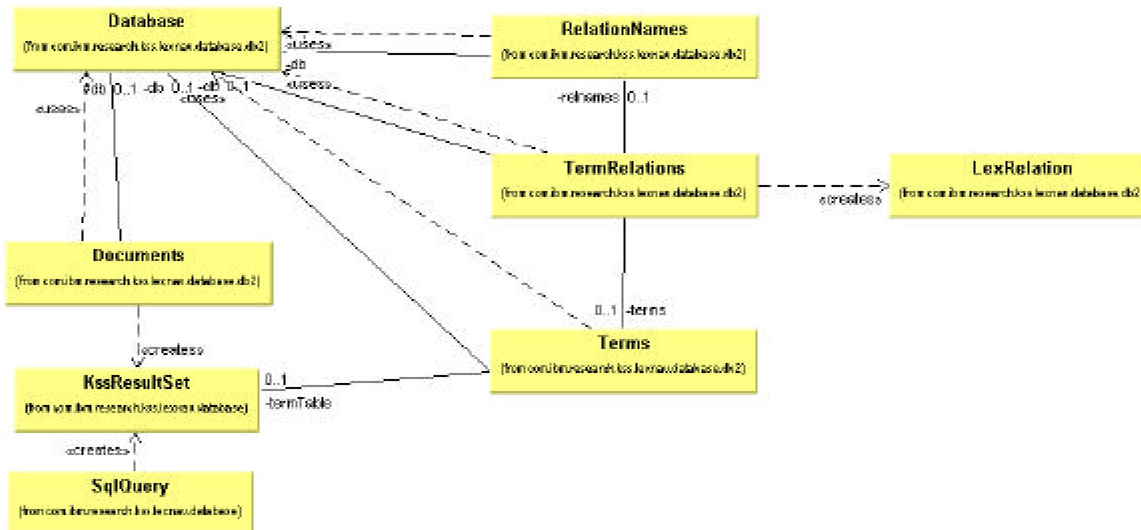

Figure 4 – The Database class hierarchy

The actual columns of these tables could then be varied for each database depending on customization requirements. This turned out to be crucial as we soon discovered.

Search Engines

A search engine is a program that builds an "inverted index" of the text in a collection of documents, and provides methods for querying the content of those documents, returning the title and/or URL of those documents, perhaps along with the locations of the search terms in those documents. It also ranks the returned documents in order of decreasing strength of the document's match to the query. There are any number of commercial and research-grade search engines available. We describe here our work with several IBM internal and IBM product search engine systems.

Our initial work was with an internal research search engine called Guru, which was available only on AIX systems. As we began to move to a more migratable design, we began using the IBM NetQuestion product search engine, and then later a derived system called TSE. We designed the search engine objects in a similar fashion, with a basic set of abstract classes in a parent package and derived concrete classes in a package beneath them. The implementation of the interface to the TSE search system involved connection to a search server, so the concrete classes needed to reflect this. Later, we adapted the GTR (Global Text Retrieval) search engine from the IBM Yamato Laboratory to fit in this same framework, by building a JNI interface to support the Java class structure. The GTR system does not involve direct connection to a server, so its implementation of the class structure is a bit simpler. It, too, is a Façade around the functions provided by the engine itself. The GTR classes amounted to

- GTRIndex (a document index)
- GTRQuery (a free text or Boolean query)
- GTRConthes (a special index for co-occurring terms)
- GTRDocument (the title, URL and rank of each result)
- GTRCalls (the JNI interface to GTR)

The implementation of these classes for the GTR search engine is shown in Figure 5.

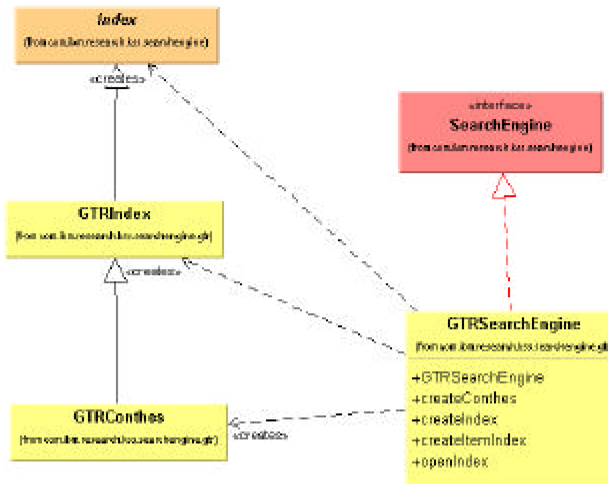

Figure 5 – The search engine classes.

System Overview

The final system design consisted of a set of packages organizing the text mining, search and database capabilities into logical units. Since we named the system "KSS" for Knowledge System Server, this is reflected in the package names:

- kss.documents –represents document collections
- kss.lexnav –abstract classes for terms and relations
  - kss.lexnav.database – database representation of terms, docs and relations
    - kss.lexnav.database.db2
    - kss.lexnav.database.basic
- kss.searchengine –abstract classes for search engines
  - kss.searchengine.tse
  - kss.searchengine.gtr
- kss.textract –classes for text mining

We were then able to use these packages to create a program to run Textract, index the documents and load the database, all written in about 200 lines of Java.

The main use of such a system, however, is to provide a way to query the data we indexed and produce a list of suggested terms to refine such queries. Because of difficulties in using Java RMI through firewalls, we designed our current system using JavaServer pages that generate pure HTML output. Such a query page was shown in Figure 1, where the results of the query "java" are shown for a set of IBM consultant reports

The JavaServer pages that generate the output interact with a server-side Java bean, which in turn connects to the search indexes and database using the same "kss" classes we used above. The complete server bean is less than 600 lines of Java code.

Evolution of the Database classes

The system we described above is complete and could be used without further changes, but as we begin looking at larger collections, some scaling issues evolved. Our original indexing program loaded the database programmatically a line at a time from the output text files

generated by Textract. While this worked well for the single-user Access database, it is far from the best way to load data into DB2 or other large-scale database systems. Instead, DB2 provides a low-level table loading function that rapidly loads huge amounts of data into tables and performs all the consistency checking at the end of the load process. To speed the data loading process, we wrote a DB2 TableLoader class that connects to this low-level function using JNI.

As we began moving to the newer GTR search engine, we also found that it indexed documents only by numeric key, and relied on the user to correlate that key with the document's title and URL. We thus wrote code into the GTR indexing C-program to produce a DB2 table load file that included these document keys. Thus, the DB2 Documents table became the lookup system for the search system results.

Delivery of the System

At this point we had a complete working system, tuned for use with DB2 and the latest internal search engine technology, and we began making it available to other researchers. As we completed the toolkit, it was adopted by a sister group as part of a project to construct a knowledge portal of marketing information. This group had already been working on the problem for some months and had developed their own object model, based on an eventual migration of the server system to Enterprise Java Beans. They adopted our toolkit by wrapping our classes in their own classes, thus making something like a Façade of adapters.

As a phase of that project completed, they delivered the KSS package along with the remainder of the portal code to our partners in IBM Global Services. Here is where we had our first surprising learning experience,

We designed the database schema for KSS to keep data in 3NF (Third Normal Form). For example, the terms table consisted of each term and its rank, along with an integer key, and the documents table consisted of the integer document key along with the document title and URL. Therefore using 3NF assumptions, our TermDocuments table was just a table of document keys and term keys. In the same fashion, the TermRelations table consisted of the two term keys and a key to a RelationName table entry. When the software engineers in our customer's group ran this code, they found that for millions of terms, looking up the relations required two database joins per row, and thus was too expensive for a production system. They suggested that we abandon the foreign key system we had developed and keep copies of the terms in each of the tables. For large systems, this greatly improved the lookup of term relations, and since we stored the terms as variable length strings (VARCHARs), it had minimal impact on the size of the database tables.

System Evolution

The interaction with our customer for this code amounted to integration of our work in stages. In each case there was convergence, followed by divergence, and then convergence at the next delivery point. We delivered basic functionality before the KSS classes were fully realized and then delivered more text analysis functions in further releases. In each case, we delivered a complete object-oriented Java class-based solution, which the customer group had to integrate with their existing infrastructure. In fact, we discovered that their usual integration mode was to take our database schema and queries and use our lowest level JNI classes to implement our functions without ever instantiating the object system we constructed. They adopted our interface

to DB2 and to the GTR engine and used our SQL, but did not use very much else from the Java jar file we delivered. This was because in the first phase, they were primarily interested in search functions and did not need to use our object model. Since our KSS library had a layered architecture, they were able to use its lowest level functions.

In addition, while our system assumed that documents would be indexed in large in-memory blocks of multiple documents that we called *virdocs*, each containing a concatenation of multiple documents, and that only the indexes and URLs were stored in a database Documents table, the customer's model was one in which each actual document was stored as an object in the database itself. This led to our developing a few more JNI calls to the base GTR indexing functions to index single documents from memory buffers instead of from a large file of concatenated documents.

It also developed that the production system used a somewhat different table schema than the table schema used during indexing. This was because their production system consisted of two subsystems: a backend indexing system and an end-user runtime system. These had different data models and database schema. For example, our initial Documents table consisted of a document key, a URL and a title. Their table contained the actual document and a different unique identifier. Between these two, we interposed a key translation table. Then after indexing was complete, a simple SQL statement combined these keys back into a column in the DB2 table containing the documents. This was much faster than keeping the translation table separate, but still allowed the translation table to be loaded rapidly during indexing.

In our second delivery, we provided all of the named and unnamed term relations and the context thesaurus, which they could really only use by adopting our class structure and much of our database schema. Consequently, they paid a price at this stage in converting their existing code to work with this model.

Further Work

Following delivery of our second version of the client-server system, we added the ability to exchange TermRelation objects between a simple client applet and an RPC server using the SOAP protocol instead of RMI. This has the advantage of being very lightweight and since it uses XML over HTTP it can be transmitted through firewalls without difficulty. The applet plot of term relations for a subject of great interest is shown in Figure 2.

Further Customer Developments

The customer continued evolution of our code after our final delivery to meet new requirements. Many of the documents they had to index originated in Lotus Notes, where there can be any number of named regions that it might be desirable to index and search on. Conveniently for this purpose, region search is indeed a feature of the GTR engine, although we had not brought it to the surface using our JNI library or included it directly in our object model. Rather than extending our object model to handle this field indexing and searching, they went directly to the base JNI C-language code and created their own functions for this indexing.

In a further development, they also are investigating integrating this search code into a DB2 UDF (User-Defined Function) that could be called by extensions from SQL. This effectively ends their use of the object system we designed.

Conclusions

We designed a powerful, high-level set of Java classes for text mining and indexing HTML documents. It has had a great deal of use within our research community, and has a number of happy users. Without this object design, and the accompanying productivity gains it gave us in constructing the system, it could never have been built or tested.

The strategy used by our sister group in adopting our KSS library toolkit is typical of what happens in cross departmental development projects. The encapsulation of our code in their object model avoids the problems inherent in understanding the classes sufficiently to modify or subclass them, and certainly is more efficient than starting from scratch.

On the other hand, the software engineers in the final customer group were driven by real and perceived performance requirements, an object learning curve, and the need to make our additions interoperate with a system that was already in production. Further, the engineers had no particular need to preserve our object design or any other object design. They were neither concerned with overall architecture nor with the need to be able to derive and change the system using our convenient set of objects. Instead, they took the most useful methods, SQL statements and JNI interfaces from our code and used them to construct the system they have currently deployed. They did, of course also have a simple object model, but it was far less comprehensive than the one we had developed.

Some of the later changes and divergence were driven by requirements that we probably could not have foreseen, particularly since the KSS object system was designed and implemented before our work with this customer commenced.

It is rewarding to see a system we designed and developed deployed in a useful real world application, but at least a surprise, if not a shock, to realize that a group of programmers may elect to use your code in far different ways than you had intended. While this was a highly successful system, its evolution was somewhat analogous to the evolution you see in other kinds of offspring, who may be influenced by your original model, but take it in their own direction.

References

[1] J W Cooper and R J Byrd, "Lexical Navigation: Visually Prompted Query Expansion and Refinement," ACM Digital Libraries, 1997.

[2] Byrd, R.J. and Ravin, Y. "Identifying and Extracting Relations in Text," *Proceeedings of NLDB 99*, Klagenfurt, Austria.

[3] J W Cooper and R J Byrd, "OBIWAN – The One Button Interface for Prompted Query Refinement.," presented at HICSS-31, Kona, HI, January, 1998.

[4] J W Cooper and Mary Neff, "Active markup and Summarization,: presented at HICSS-32, Maui, HI, January, 1999.

[5] Byrd, R.J. and Ravin, Y. Identifying and Extracting Relations in Text. *Proceeedings of NLDB 99*, Klagenfurt, Austria.

[6] Xu, Jinxi and Croft, W. Bruce. "Query Expansion Using Local and Global Document Analysis," *Proceedings of the 19th Annual ACM-SIGIR Conference*, 1996, pp. 4-11.

[7] E. Gamma, R. helms, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object Software,* Addison-Wesley, 1995.

[8] J W Cooper, *Java Design Patterns: A Tutorial*, Addison-Wesley, 2000.