Paper SAS195-2014

Processing and Storing Sparse Data in SAS® Using SAS® Text Miner Procedures

Zheng Zhao, Russell Albright, and James Cox
SAS Institute Inc.

ABSTRACT

Sparse data sets are common in applications of text and data mining, social network analysis, and recommendation systems. In SAS® software, sparse data sets are usually stored in the coordinate list (COO) transactional format. Two major drawbacks are associated with this sparse data representation: First, most SAS procedures are designed to handle dense data and cannot consume data that are stored transactionally. In that case, the options for analysis are significantly limited. Second, a sparse data set in transactional format is hard to store and process in distributed systems. Most techniques require that all transactions for a particular object be kept together; this assumption is violated when the transactions of that object are distributed to different nodes of the grid. This paper presents some different ideas about how to package all transactions of an object into a single row. Approaches include storing the sparse matrix densely, doing variable selection, doing variable extraction, and compressing the transactions into a few text variables by using Base64 encoding. These simple but effective techniques enable you to store and process your sparse data in better ways. This paper demonstrates how to use SAS® Text Miner procedures to process sparse data sets and generate output data sets that are easy to store and can be readily processed by traditional SAS modeling procedures. The output of the system can be safely stored and distributed in any grid environment.

INTRODUCTION

In recent years, there has been a growing interest in techniques that can analyze sparse data. A data set is sparse when most of its elements are missing.[1] For example, say a company that has 100,000 products and 10,000 customers executes one million sales transactions in 2012. The total number of combinations of products and customers is one billion, which means that the transaction table contains only 0.1% of all possible combinations. Sparse data exist widely in the analytics world. In text mining, social network analysis, and recommendation systems, data are usually sparse. For example, the data used in the Netflix challenge[2] contain 103,297,638 ratings from 480,189 users for 17,770 films. The data density (the ratio of all actual user ratings to all possible user ratings) is about 1.21%. Another example is the 20 Newsgroups data,[3] which are widely used in text mining research as benchmark data. This data set contains 18,846 documents, and the dictionary size is 61,188. When this data set is represented using the bag-of-words model, the document-by-term table has 18,846 rows that correspond to documents and 61,188 columns that correspond to terms. Only 2,435,219 nonzero elements appear in the document-by-term table. This represents a data density of about 0.21%. Sparse data pose major challenges for both data storage and processing in statistical modeling. Most elements in the design matrix of a sparse data set are zeroes. Therefore, it is inefficient to store the full design matrix to perform modeling. For example, assuming that double-precision floating-point format is used to store numbers, storing the full design matrix of the Netflix challenge data requires 63.6 GB of storage space. In contrast, storing only the nonzero elements requires only 0.77 GB of storage space. Similarly, storing the full design matrix of the 20 Newsgroups data requires 8.6 GB of storage space, whereas storing only the nonzero elements requires only 0.018 GB of storage space. These examples show the huge savings in storage space when you store only the nonzero elements of a sparse design matrix. To this end, various sparse matrix representations have been designed to store a sparse matrix in transactional format (where each row represents one cell of the matrix). Representative sparse matrix formats include the list of lists (LIL) format, the coordinate list (COO) format, the compressed sparse row (CSR) format, and the compressed sparse column (CSC) format.

In big-data analysis, storing sparse data and distributing them across a distributed computing environment present extra challenges. For handling large-scale problems, SAS high-performance analytical procedures perform distributed computing in a grid environment. Data that are used by these procedures are either predistributed onto a distributed database on a grid or distributed to the grid when the procedures are launched for computation. In many database systems, these data are distributed in a round-robin style, and there is no guarantee that transactions associated with a particular object will be on the same node in the grid. Let an object correspond to a row in the design matrix, and let

[1] For text, the missing elements are assumed to represent zero.
[2] The Netflix challenge was an open competition for the best algorithm for predicting users' ratings for films. http://en.wikipedia.org/wiki/Netflix_Prize
[3] The 20 Newsgroups data set is a collection of newsgroup documents, which are partitioned (approximately) evenly across 20 newsgroups. http://qwone.com/~jason/20Newsgroups/

the transactions of the object correspond to the elements in the row. Most distributed matrix operations require that elements from the same row of the design matrix be distributed on the same node of the grid system. When data are not distributed in this way, computation usually generates meaningless results.

To address these issues, this paper describes a number of ways to compact the transactional format of a sparse matrix to guarantee that all the transactions of an object are stored on a single node. Your choice of how to represent your sparse data depends on whether you want the full data or a reduced representation for modeling. In some cases, you want to use all the data to build a model. This paper describes an approach that can compress all the transactions of an object into a few text variables by using Base64 encoding. The resulting data can then be distributed to the grid nodes without concerns about keeping transactions together. However, this approach requires that a decoding be performed before modeling when the data are read in. In many cases, it is preferable to reduce the dimensionality of the data. This paper describes two techniques, variable selection (where a small subset of the original variables are retained) and variable extraction (where a small set of new composite variables are created), that can lead to a dramatic reduction in the dimensionality of the problem being analyzed. Speed and accuracy for those two approaches are also compared for a real-world data set.

DATA STRUCTURES FOR STORING SPARSE DATA

Sparse data have the special characteristic of low density. Various data structures have been designed to store a sparse matrix more efficiently by storing only the nonzero elements. The most common sparse matrix formats include the list of lists (LIL) format, the coordinate list (COO) format, the compressed sparse row (CSR) format, and the compressed sparse column (CSC) format. In general, these formats fall into two categories:

- formats that enable data to be updated efficiently, including the LIL format and the COO format
- formats that support efficient matrix operations, including the CSR format and the CSC format

The sparse format in the first category is usually used to construct a matrix. After a matrix is constructed, it is usually converted to a format in the second category, such as the CSR format or the CSC format, in order to support more efficient matrix operations. The following sections briefly introduce different sparse matrix formats and discuss their advantages and disadvantages. The following sparse matrix, ***A***, is used as an example in these sections to show how different sparse matrix formats represent it in different ways:

$$\boldsymbol{A} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{2} \\ \mathbf{0} & \mathbf{3} & \mathbf{0} \\ \mathbf{5} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

LIST OF LISTS (LIL) FORMAT

The LIL format stores one list per row. In each list, an entry stores a column index and the value of the corresponding element. To support quick lookup, the entries in the list are usually sorted by their column index. Matrix ***A*** has the following representation in this format:

$$\boldsymbol{A} = \begin{pmatrix} (\{0,1\},\{2,2\}) \\ (\{1,3\}) \\ (\{0,5\}) \end{pmatrix}$$

The LIL format is good for incremental matrix construction. For example, it is easy to add new rows and new columns to the matrix by expanding the lists. However, storing this data structure in a relational database can be very inefficient because the number of the elements in each row of the matrix usually vary dramatically. Also, when the maximum number of nonzero elements in a row of a sparse matrix exceeds the number of columns that a database system supports, it becomes impossible to store the matrix in a database table because the column number of the data table needs to match the maximum number of the nonzero elements in a row of the matrix. Many database systems limit the maximum number of columns in a data table to a few thousand.

COORDINATE LIST (COO) FORMAT

The COO format stores a list of tuples. Each tuple's entries consist of the row index, the column index, and the value of the corresponding matrix element. To support quick lookup, the tuples are usually grouped by row index and are sorted by column index. Matrix ***A*** has the following representation in this format:

$$\boldsymbol{A} = \begin{pmatrix} \{0,0,1\} \\ \{0,2,2\} \\ \{1,1,3\} \\ \{2,0,5\} \end{pmatrix}$$

The COO format is also good for incremental matrix construction. For example, it is easy to add new rows and new columns to the matrix by inserting more tuples in the list. Storing this data structure in a relational database is straightforward because the database table needs to have only three columns and the number of rows is equal to the total nonzero elements in the matrix.

COMPRESSED SPARSE ROW (CSR) FORMAT

The CSR format uses three arrays, **`Val`**, **`Col_Ind`**, and **`Row_Ptr`**, to store a sparse matrix. Assume that a sparse matrix has *t* nonzero elements. Then **`Val`** is the value array that contains the *t* nonzero elements of the sparse matrix, **`Col_Ind`** is the column index array of length *t* that contains the column indices of the elements in the **`Val`** array, and **`Row_Ptr`** is the row index array that contains the indices of the **`Col_Ind`** array. The *i*th element of the **`Row_Ptr`** array indicates the start of the *i*th row of the sparse matrix. In the CSR format, each row of the sparse matrix corresponds to only one element in the **`Row_Ptr`** array. This contrasts to the COO format, in which each row index is stored in multiple tuples. Therefore, the CSR format provides a more economical way for storing a sparse matrix. Matrix ***A*** has the following representation in this format:

$$\boldsymbol{A} = \begin{pmatrix} \textbf{Val} & = \{1,2,3,5\} \\ \textbf{Col\_Ind} & = \{0,2,1,0\} \\ \textbf{Row\_Ptr} & = \{0,2,3\} \end{pmatrix}$$

The CSR format is efficient for arithmetic operations that are applied to matrix elements and for row slicing and matrix-vector products. However, it could be expensive to insert new columns in the matrix or to delete existing columns. Therefore, this format is more suitable for supporting efficient matrix computations. As with the LIL format, storing the data structure that is used in the CSR format in a relational database table can be very inefficient.

COMPRESSED SPARSE COLUMN (CSC) FORMAT

The CSC format is similar to the CSR format except that values are grouped by columns. The CSC format also uses three arrays, **`Val`**, **`Row_Ind`**, and **`Col_Ptr`**, to store a sparse matrix. Assume that a sparse matrix has *t* nonzero elements. Then **`Val`** is the value array that contains the *t* nonzero elements of the sparse matrix, **`Row_Ind`** is the row index array of length *t* that contains the row indices of the elements in the **`Val`** array, and **`Col_Ptr`** is the column index array that contains the indices of the **`Row_Ind`** array. The *i*th element of the **`Col_Ptr`** array indicates the start of the *i*th column of the sparse matrix. In the CSC format, each column of the sparse matrix corresponds to only one element in the **`Col_Ptr`** array. Therefore, the CSC format provides a more economical way for storing sparse matrix than the COO format provides. Matrix ***A*** has the following representation in this format:

$$\boldsymbol{A} = \begin{pmatrix} \textbf{Val} & = \{1,5,3,2\} \\ \textbf{Row\_Ind} & = \{0,2,1,0\} \\ \textbf{Col\_Ptr} & = \{0,2,3\} \end{pmatrix}$$

As with the CSR format, the CSC format is efficient for arithmetic operations that are applied to matrix elements and for column slicing and matrix-vector products. However, it could be expensive to insert new rows in the matrix or delete existing rows. Therefore, this format is more suitable for supporting efficient matrix computations. Also, storing the data structure that is used in the CSC format in a relational database table can be very inefficient.

HANDLING SPARSE DATA IN SAS

A SAS data set resembles a data table in a relational database. As analyzed in the preceding section, among the four sparse formats, only the COO format is suitable for updating a sparse matrix in a relational database table. Therefore, the COO format is usually used in SAS for storing sparse matrices. Procedures that can produce or consume a sparse matrix in COO format include the ASSOC, HPTMINE, HPTMSCORE, NETFLOW, OPTGRAPH, RECOMMEND, RULEGEN, SEQUENCE, SPSVD, TGPARSE, and TMUTIL procedures.

Table 1 shows the document-by-term table that is created by the HPTMINE procedure. The _TERMNUM_ column contains the ID of terms (the column indices), the _DOCUMENT_ column contains the ID of documents (the row indices), and the _COUNT_ column contains the number of times that a term appears in a document. The elements in the _COUNT_ column are the elements in the sparse matrix that is presented by the document-by-term table.

Table 1. The Document-by-Term Table Created by the HPTMINE Procedure

DOCUMENT	_TERMNUM_	_COUNT_
1	4	1
1	10	2
2	1	1
2	5	5
3	4	3

CHALLENGES FROM BIG-DATA ANALYSIS

Big-data analysis poses new challenges for handling sparse data in a distributed computing environment. For example, although the COO format is effective for storing a sparse data set in a SAS data file, it could cause problems when the data are distributed to the grid. More specifically, to ensure the validity of the outcomes, most distributed matrix operations require that the elements from the same row of the design matrix be distributed to the same node of the grid system. These elements usually are also required to be grouped together when they are delivered to the operations. However, when a sparse data set is stored in the COO format, this assumption can be easily broken when the data are distributed to a grid. For example, Figure 1 shows how data integrity is lost, when rows are distributed to the grid in a round-robin fashion. Similar problems exist when a sparse data set in the COO format is predistributed and stored in a distributed database system. To avoid these problems, you need to use additional options for distributing the data set.

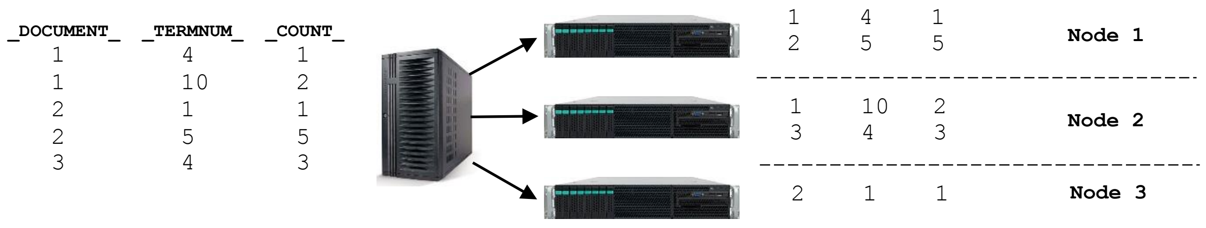


Figure 1. Broken Data Integrity When Data in the COO Format Are Distributed in a Round-Robin Fashion

When you use the COO format to store your sparse data in a local SAS data set and you want to run a SAS high-performance analytical procedure on the grid, you can use the DISTBY statement to specify the distribution key for distributing your data. The following example uses the DISTBY statement in the HPTMINE procedure:

```
option set=GRIDHOST      ="&your_host";
option set=GRIDINSTALLLOC="&your_grid_instloc";

proc hptmine data   =  docs_by_terms;
svd     k           =  10
        row         = _termnum_
        col         = _document_
        entry       = _count_
        outdocpro   =  docpro
        res         =  med;
distby  document_;
performance nodes   =  2;
run;
```

The DISTBY statement specifies that the _DOCUMENT_ column is the distribution key for distributing data to the grid. When distributed to the grid, all the rows that have the same _DOCUMENT_ value are distributed to the same node of the grid. These rows are also grouped together when they are delivered to the HPTMINE procedure.

A sparse data set in the COO format can also be loaded to the grid and stored in a distributed database system, if the system supports a distribution key. The following example shows how to use the SAS DATA step and the HPDS2 procedure to distribute sparse data in the COO format to a grid that runs a Teradata database system:

```
option set=GRIDHOST      ="&your_host";
option set=GRIDINSTALLLOC="&your_grid_instloc";
option set=GRIDDATASERVER="&your_data_server";

libname TDLib teradata
        server  =&your_data_server
        user    =&usr
        password=&pwd
        database=&db;

/* load data using DATA steps */
data TDLib.docs_by_terms (dbcreate_table_opts='PRIMARY INDEX (_document_)');
set docs_by_terms;
run;

/* load data using the HPDS2 procedure */
proc hpds2 data= docs_by_terms out=TDLib.docs_by_terms_hpds2(dbcreate_ta-
ble_opts='PRIMARY INDEX (_document_)');

  data DS2GTF.out;
     method run();
        set DS2GTF.in;
        output;
     end;
  enddata;
run;
```

The DBCREATE_TABLE_OPTS= option in the DATA step specifies the distribution key. For the Teradata database system, the value of this option is 'PRIMARY INDEX (_document_)'. It tells the Teradata system to use the _DOCUMENT_ column for distributing the data. This ensures that all rows that have the same _DOCUMENT_ value are distributed to the same node. You can use similar code to load your data to a grid system that runs a Greenplum database system by specifying DBCREATE_TABLE_OPTS='DISTRIBUTED BY (_document_)'. For the SASHDAT and the SASIOLA engine, you can specify the distribution key by using the PARTITION option as follows:

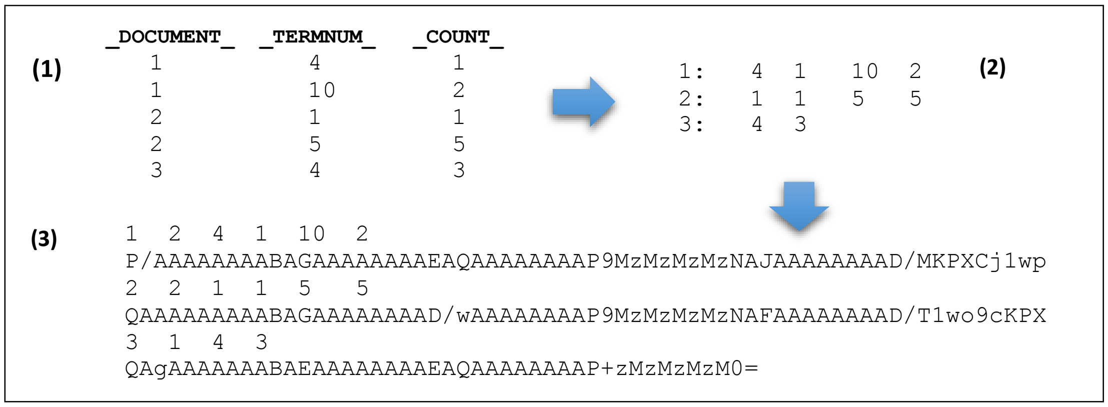
```
data lasr.docs_by_terms (partition=(_document_));
  set docs_by_terms;
run;
```

Not all distributed database systems can distribute data by using a distribution key. In these cases, procedures needs to reshuffle the data after the data are loaded into memory to enforce integrity. However, this is usually very expensive.

THE BASE64 ENCODED SPARSE ROW (BESR) FORMAT

Starting with SAS Text Miner 13.2, high-performance text mining procedures support a new sparse data format, called the Base64-encoded sparse row (BESR) format. This format encodes each row of the sparse design matrix as a string and stores the string in one or a few text variables in a data table. This sparse format enables you to store your sparse design matrix in any relational database. Because each row of the sparse matrix is stored in one row of the data table, distributing data to the grid does not lead to any integrity issue. This format also enables you to jointly store your sparse design matrix along with additional information about its rows. For example, in text mining, each row of the document-by-term matrix can have its BESR representation, the original document, and its author as three individual columns in the data table. In recommendation systems, each row of the user-item-ratings table can have its BESR representation and the corresponding user's profile stored together as two individual columns in a data table.

Figure 2 illustrates how a row of a sparse matrix is converted to a string by using the BESR format.



```
     _DOCUMENT_  _TERMNUM_   _COUNT_
(1)      1           4          1
         1           10         2          1:   4  1   10  2    (2)
         2           1          1          2:   1  1   5   5
         2           5          5          3:   4  3
         3           4          3

(3)    1  2  4  1  10  2
       P/AAAAAAAABAGAAAAAAAAEAQAAAAAAAAP9MzMzMzMzNAJAAAAAAAAD/MKPXCj1wp
       2  2  1  1  5   5
       QAAAAAAAAABAGAAAAAAAAD/wAAAAAAAAP9MzMzMzMzNAFAAAAAAAAD/T1wo9cKPX
       3  1  4  3
       QAgAAAAAAABAEAAAAAAAAEAQAAAAAAAAP+zMzMzMzM0=
```

Figure 2. Base64-Encoded Sparse Row Format Converts Each Row of a Sparse Matrix to a String

Given a sparse design matrix, the BESR format first forms an array for each row of the sparse matrix. Each array starts with its row index, which is followed by the column index and value pairs. Before encoding, some meta-information is added to each array after the row index. In this example, the meta-information is the number of column and value pairs, which are equal to the number of nonzero elements in each row of the sparse design matrix. To make the BESR representation platform-independent, the array is then converted to an array that contains doubles in the IEEE 754 format. The resulting double arrays are finally encoded using the Base64 encoding.

The BESR format has the following advantages:

- Strings are easy to store in SAS and database systems:
  - In SAS, the maximum length for text variable is 32 KB, which can be used to store more than 3,000 doubles. And in database systems, the maximum length is usually much longer than 32 KB.
  - If the text string is too large, you can use several text variables to store it.
  - SAS and most database systems can compress text variables.
- The BESR format is independent of platform:
  - Doubles in the array are converted from their native format to the portable IEEE 754 format before they are encoded to a string by using the Base64 encoding.
  - Base64 encoding uses 64 characters that are common to all encoding systems. Therefore, there are no encoding compatibility issues when data are transported and stored in different systems.
  - All 64 characters appear on a regular keyboard; thus they can be easily viewed.
- There is no data integrity issue when data are distributed in a grid environment:
  - Elements from the same sparse row are always grouped together.
  - No reshuffling is required.
- The BESR format facilitates joint storage:
  - You can store a sparse design matrix along with the meta-information about its rows in one data table.

The following example uses the HPTMINE procedure to process a text data set and stores the sparse document-by-term matrix in the BESR format. It then reads in the stored document-by-term matrix and applies singular value decomposition in SVD-only mode.

```
option set=GRIDHOST      ="&your_host";
option set=GRIDINSTALLLOC="&your_grid_instloc";

 proc hptmine data=&your_text spfmt=BESR;
     doc_id &doc_id; var &text_var;
     parse termwgt=mi cellwgt=log stop=sashelp.engstop
           outterms=keys outparent=docs_by_terms_besr (compress=yes);
     performance nodes=4 details;
```

```
run;

proc hptmine data=docs_by_terms_besr spfmt=BESR;
    doc_id &doc_id; var sparse_row:;
    svd k=10 res=med outdocpro=docpro_svdonly;
    performance nodes=2 details;
run;
```

In the first PROC HPTMINE call, the SPFMT= option specifies the sparse format for storing the sparse document-by-term matrix that is generated by the HPTMINE procedure. In the `docs_by_terms_besr` data set that is generated by the procedure, each Base64-encoded sparse row is stored in one or a few text variables whose names include the prefix SPARSE_ROW_. In the second HPTMINE call, the `docs_by_terms_besr` data set is delivered to the procedure as input. The VAR statement defines the variables that are used to store the Base64-encoded sparse rows. In contrast to the example on page 4, in which the input document-by-term matrix is in the COO format, the DISTBY statement is not used. This is because the BESR format can guarantee the data integrity on the grid.

ANALYZING SPARSE DATA IN SAS

Most SAS statistical modeling procedures are designed to process a dense design matrix. Therefore, they cannot handle a data set that is stored in a sparse matrix format. If you have a sparse data set, you can process it and make it suitable for standard SAS procedures to analyze in several ways:

1. Create a full design matrix: You can create a full design matrix from a sparse design matrix and deliver the full design matrix to the procedures. If the full matrix contains too many columns, you will face problems for both storing and analyzing the data because each column of the design matrix corresponds to a variable.
2. Variable selection: You can reduce the dimensionality (the number of variables) of the original sparse design matrix by selecting a subset of the original columns. This approach selects a subset of the original variable. Its output is still a full sparse matrix, but the matrix has many fewer columns than the original matrix.
3. Variable extraction: You can generate a lower-dimension representation of the original design matrix by using a factorization procedure, such as the SPSVD or HPTMINE procedure. This approach generates a small set of new variables by linearly combining all the original variables. Its output is a dense matrix, which has many fewer columns than the original matrix.

Compared to the variable selection approach, the variable extraction approach usually outputs a dense matrix that has fewer columns. At the same time, it can usually preserve more information from the original design matrix, which leads to better performance of your modeling procedures. However, sometimes model interpretability can also be very important. For example, you might want to know the impact of the original variables to your model. In this case, you must use the variable selection approach, because the output of the variable extraction approach does not contain any original variables.

ROLLING UP A SPARSE DESIGN MATRIX IN COO FORMAT

When your sparse design matrix does not contain many columns, you can use the following SAS macro to roll up your matrix stored in the COO format to a full matrix:

```
%MACRO coo2full( infile=, rowid=, colid=, val=, oufile=, keys= );
    proc sql noprint;
    create table c2f_tbl_a as select distinct &colid from &infile;
    create table &keys as select &colid, monotonic() as c2f_key from c2f_tbl_a;
    create table new_&infile as select a.&rowid, b.c2f_key as &colid, a.&val
        from &infile a, &keys b where a.&colid=b.&colid;
    select count(&colid) into :c2f_var_ncols separated by '' from &keys;
    quit;

    data &oufile (compress = yes);
        set new_&infile;
        by  &rowid;
        array cols {&c2f_var_ncols} col1-col&c2f_var_ncols;
        retain cols;
        do i= 1 to &c2f_var_ncols;
            if i=&colid then cols{i}= &val;
        end;
        keep &rowid col1-col&c2f_var_ncols;
        if last.&rowid then do;
            output;
            do j = 1 to &c2f_var_ncols; cols{j}= 0; end;
        end;
    run;
%MEND coo2full;
```

This macro takes five inputs. The INFILE macro variable specifies the input data set that stores your sparse data in the COO format. For this data, the ROWID, COLID, and VAL macro variables specify the variables that contain the row indices, column indices, and values, respectively. The OUFILE macro variable specifies the output data set that will be used to store the full design matrix. The KEYS macro variable specifies the output data set that will be used to store the mapping information between the old and the new column indices.

In the input data column, indices might not start from 1 or be numbered sequentially. Therefore, the macro uses the SQL procedure to identify all distinct column indices, renumber them sequentially, and uses the new column indices to represent the data. The macro also generates the data set that is specified in the KEYS macro variable to record the mapping information between the old and new column indices. Given the new representation of your data, the COO2FULL macro uses DATA step code to create a full design matrix, which is stored in the data set that is specified by the OUFILE macro variable. Because most elements in this matrix are zero, the COMPRESS=YES option in the DATA step is specified to compress the data. Compressing can effectively reduce the size of your data file.

VARIABLE SELECTION BEFORE ROLLING UP

When your sparse data in the COO format contain too many columns (variables), it becomes impractical for you to roll them up to create a full design matrix. First, the size of the full matrix can be prohibitively large. Second, your modeling procedure might also face the “curse-of-dimensionality” problem. One way to address these problems is to select a subset of the original variables and generate sparse data in the COO format that contain only the selected variables. A full design matrix can then be created by rolling up this dimensionality-reduced data. This is a variable selection approach. Variable selection can effective improve the efficiency and interpretability of your statistical models. Assume that your sparse data are in the COO format and you have already weighted the variables in your data according to their utility for modeling. The following SAS macro shows how you can use the weights of the variables to filter your data and generate a dimensionality-reduced data set in the COO format:

```
%MACRO coo_filter( infile=, rowid=, colid=, keys=, key_var=,
                   weight_var=, oufile=, ncols= );
    proc sql outobs=&ncols;
        create table top_&keys as
            select * from &keys
                order by &weight_var desc;
    quit;
    proc sql;
        create table &oufile as
            select a.* from &infile a, top_&keys b
                where a.&colid=b.&key_var order by a.&rowid;
    quit;
%MEND coo_filter;
```

Let ***A*** be your design matrix. This macro takes eight inputs. The INFILE macro variable specifies the input data set that stores ***A*** in the COO format. For this data set, the ROWID, COLID, and VAL macro variables specify the variables that contain the row indices, column indices, and values of ***A***, respectively. Because each column of ***A*** corresponds to a variable of your data, the column indices of ***A*** are the variable indices. The KEYS macro variable specifies the input data set that contains information about the variables in your data. For these data, the KEY_VAR macro variable specifies the variable that contains the variable indices, and the WEIGHT_VAR macro variable specifies the variable that contains the variable weights that measure the relevance of the variables for modeling your data. The OUFILE macro variable specifies the output data set that will be used to store the filtered design matrix in the COO format, and the NCOLS macro variable specifies how many variables to select.

In the COO_FILTER macro, the first SQL procedure call sorts the data set that is specified by the KEYS macro variable in descending order according to the weights of the variables and selects the top NCOLS variables. The second SQL procedure call filters the input data by using the selected variables and outputs the data to the data file whose name is specified by the OUFILE macro variable. In a supervised learning context, you can estimate the relevance of the variables in your data by measuring their correlation to the target variable. And in an unsupervised learning context, you can estimate the relevance of the variables by measuring the variance they contribute to the data. For example, in the HPTMINE procedure, you can use the TERMWGT option in the PARSE statement to weight terms for supervised learning by specifying TERMWFT=MI or for unsupervised learning by specifying TERMWGT=ENTROPY. The following example shows how you can use the TERMWFT=MI option to weight the variables by using mutual information. The WEIGHT variable stores the weights of the variables in the KEYS data set.

```
proc hptmine data=textdata;
    doc_id id_var; var text_var; target target_var;
    parse termwgt=mi cellwgt=log outterms=keys outparent=out_coo;
run;
```

GENERATING A DENSE DESIGN MATRIX BY USING SINGULAR VALUE DECOMPOSITION

You can also use the variable extraction approach to generate a dense matrix by linearly combining the variables in the original data. A popular variable extraction technique is the truncated singular value decomposition (SVD). This technique can effectively transform a high-dimensional sparse matrix into a low-dimensional dense matrix. The truncated SVD is a matrix factorization that enables you to approximate a sparse matrix, $\boldsymbol{A}$, by using the product of three factors: $\boldsymbol{D} \approx \boldsymbol{U\Sigma V}^T$. Assume that matrix $\boldsymbol{A}$ contains n rows and m columns, which correspond to the n sample and m features of your data, respectively. $\boldsymbol{U}$ is an $n \times k$ matrix, $\boldsymbol{V}$ is an $m \times k$ matrix, and $\boldsymbol{\Sigma}$ is a $k \times k$ diagonal matrix. The $\boldsymbol{U}$ matrix contains the left singular vectors of the $\boldsymbol{A}$ matrix. Each row of the $\boldsymbol{U}$ matrix corresponds to a sample from the data, and each of its columns corresponds to a newly extracted variable. Usually, $k \ll m$. The diagonal elements of $\boldsymbol{\Sigma}$ indicate the importance of the extracted variables. Let the *i*th column of the $\boldsymbol{U}$ matrix be $\boldsymbol{u}$. The k-dimensional projection of the *i*th sample is computed by $\boldsymbol{u\Sigma}/\|\boldsymbol{u\Sigma}\|_2$. Here $\|\cdot\|_2$ computes the Euclidian norm of a vector.

Figure 3 shows some interesting properties of the left-singular vectors of the $\boldsymbol{A}$ matrix.

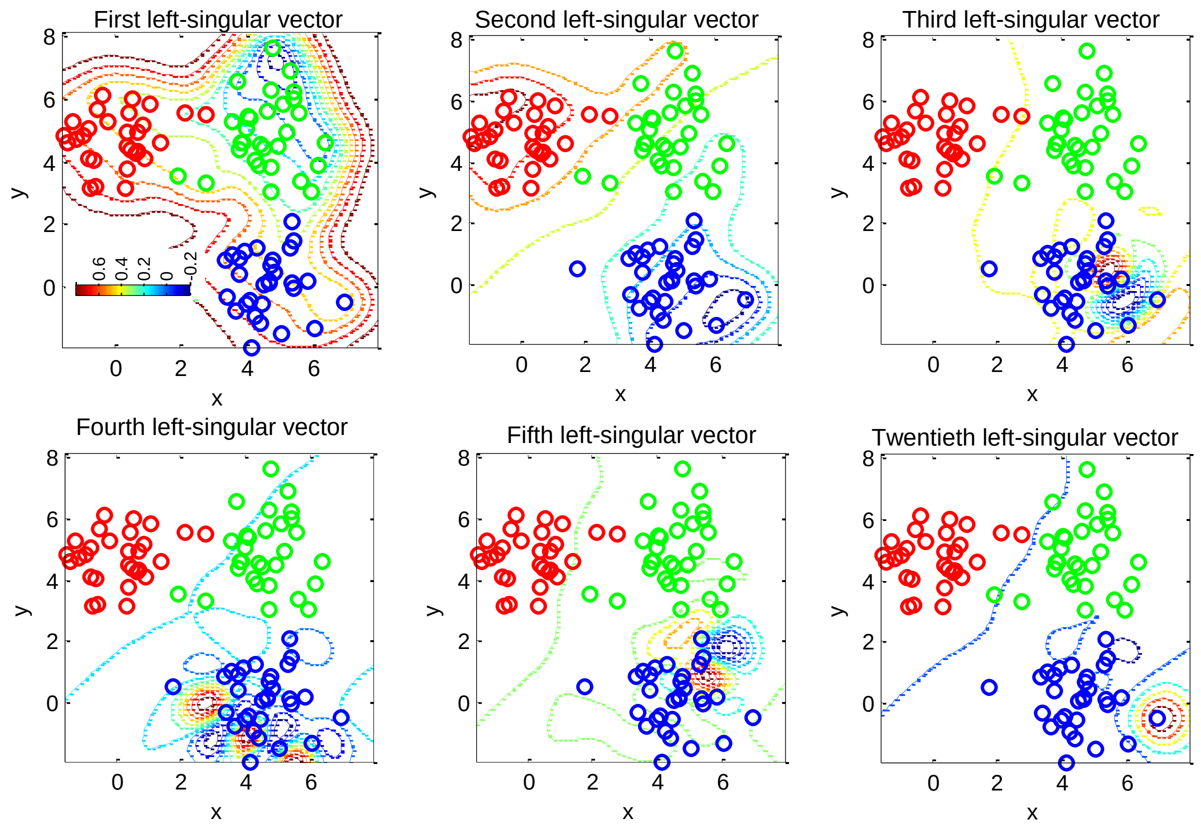


Figure 3. Contours of Various Left-Singular Vectors of the $\boldsymbol{A}$ Matrix

Figure 3 plots the contours of six left-singular vectors of the $\boldsymbol{A}$ matrix. The $\boldsymbol{A}$ matrix represents a data set that contains 90 samples, which are drawn from a mixture of three 2-dimensional (2D) Gaussian distributions. Because n = 90, each left-singular vector contains 90 elements, and each element assigns a value to one of the 90 samples. In each plot, only the points that correspond to the 90 samples have values assigned to them. Therefore, for any point in the 2D space that has no value, its value is computed by averaging the values of the nearby points that correspond to samples. In the averaging process, the values of its neighbors are also weighted by their distance from the point. For the contour lines, the closer the color is to blue, the smaller the value. Figure 3 shows that the first and second left-singular vectors capture the cluster structure of the data by assigning small values to samples from the green group and blue group, respectively. They also assign similar values to samples that are close to each other in the original space. The remaining left-singular vectors capture the subtle structures of the data, which might be created by noise. The example shows that the similarity among samples is preserved when the leading left-singular vectors are used to represent samples. Analyzing the similarity among samples is essential for many statistical modeling algorithms, such as regression, support vector machine, and neural network.

Given a sparse matrix that is stored in the COO format, you can use the SPSVD or the HPTMINE procedure to perform singular value decomposition. The following SAS code shows how you can use the two procedures to compute SVD and generate a low-dimensional dense design matrix. For the SPSVD procedure, this dense matrix is stored in the SAS data set that is specified in the DOCPRO= option in the OUTPUT statement. For the HPTMINE procedure, the dense matrix is stored in the SAS data set that is specified in the OUTDOCPRO= option in the SVD statement. Compared to the SPSVD procedure, the HPTMINE procedure also supports multithreading and distributed computing. Therefore, PROC HPTMINE is more efficient and can handle problems with a much larger scale.

```
proc spsvd data=spsvdIn
    k=&ncol res=med;
    row &row_var; col &col_var; entry &val_var;
    output docpro=docpro;
run;

proc hptmine data=mat_coo;
svd  k=&ncol res=med
     row=&row_var col=&col_var entry=&val_var
     outdocpro=docpro;
run;
```

PERFORMANCE OBSERVATIONS

You can expect to see considerable performance improvements when you use the variable extraction approach instead of the variable selection approach for generating a low-dimensional full design matrix for your sparse data. The following example takes a text data and generates a bag-of-words representation. The result is a very sparse document-by-term matrix in the COO format. Both the variable selection and the variable extraction approaches are used to generate a full design matrix from the document-by-term matrix. The resulting full design matrices are then delivered to five different predictive models to compare the performance that these models can achieve when different approaches are used to generate the full design matrices.

Evaluation Data and Experiment Setup

The text data for this evaluation contain 1,329 newsgroup posts from two newsgroups: hockey and baseball. These posts are processed by the HPTMINE procedure to generate a document-by-term matrix in the COO format. The resulting matrix contains 1,329 row indices and 9,394 column indices. Each row index corresponds to a sample, and each column index corresponds to a term. Both the variable selection approach and the variable extraction approach are used generate a lower-dimensional full design matrix from the document-by-term matrix that is in the COO format.

For the variable selection approach, the relevance weight of the *i*th term is computed as $w_i = \sqrt{\#docs_i} \times mi_\text{i}$. In this equation, w_i is the relevance weight of the *i*th term, $\#docs_i$ is the number of documents that the *i*th term has appeared in, and mi_i is the mutual information between the *i*th term and the target. In order for a term to obtain high relevance weight when this equation is used, it needs to be highly correlated to the target variable. At the same time, it should not be a rare term that appears only in a few documents. To preserve a reasonable amount of information, 500 terms that have the highest relevance weight are used to create the full design matrix.

For the variable extraction approach, the HPTMINE procedure is used to compute the top 50 singular vectors for the document-by-term matrix that is in the COO format. A dense design matrix that has 50 columns is then computed based on the resulting singular vectors by using the projection formulation for SVD.

To facilitate data partitioning for model comparison, the design matrices that are generated by the feature selection approach and the feature extraction approach are joined by their document indices to form a larger data set that contains 1,329 rows and 552 columns. One column is for the target variable, one is for the document index, 500 are from the data that are generated by the variable selection approach, and the remaining 50 are from the data that are generated by the variable extraction approach. You can find the SAS code for processing the text data and generating the final full design matrix in the Appendix.

The experiment is conducted in the SAS® Enterprise Miner™. Figure 4 shows the Enterprise Miner diagram for comparing the variable selection approach to the variable extraction approach. In this example, five modeling nodes are used: the HP Forest node, the Regression node, the Decision Tree node, the Neural Network node, and the HP SVM node. The Data Partition node is used to partition the data to the training and validation sets. Modeling nodes are first trained by using the training set, and then their performance is evaluated by using the validation set. Each modeling node has two instances in the diagram. For one instance, the columns are generated by the variable selection approach; for the other instance, the columns are generated by the variable extraction approach. For example, the Regression node has two instances in the diagram. The instance whose columns are generated by the variable selection approach is named "Regression – Term 500," and the instance whose columns are generated by the variable extraction approach is named "Regression – SVD 50." The Model Comparison node is used to compare the modeling nodes.

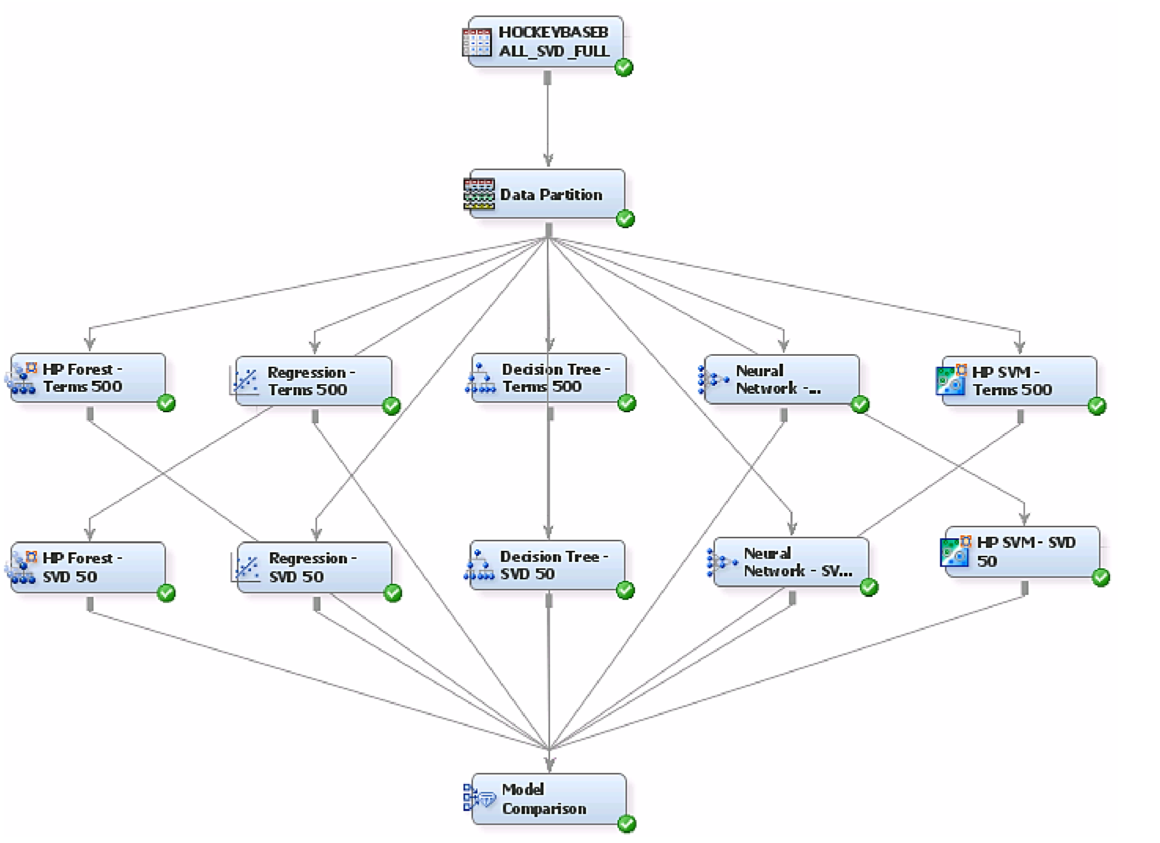


Figure 4. Comparing the Variable Selection Approach to the Variable Extraction Approach

Performance Observations

Table 2 compares the performance of different predictive models when they use the full design matrices that are generated by the variable selection approach and the variable extraction approach. The results show that modeling nodes can usually achieve a better performance by using the design matrix that is generated by the variable extraction approach. For example, the best accuracy rate that is achieved by modeling nodes when they use the design matrix generated by the variable selection approach is 0.955. However, this accuracy is still lower than the worst accuracy (0.975) that is achieved by the nodes when they use the design matrix generated by the variable extraction approach.

Table 2. Performance of the Predictive Models

Algorithms	Accuracy		Roc Index		Gini Coefficient	
	SVD 50	TERM 500	SVD 50	TERM 500	SVD 50	TERM 500
Neural Network	0.990	0.952	0.999	0.989	0.999	0.978
HP SVM	0.990	0.955	1.000	0.995	1.000	0.989
HP Forest	0.980	0.877	0.998	0.976	0.996	0.951
Decision Tree	0.977	0.852	0.975	0.866	0.951	0.732
Regression	0.975	0.952	0.990	0.990	0.980	0.980

Table 3 compares the run times of different predictive models when they use the full design matrices that are generated by the variable selection approach and the variable extraction approach. The results are obtained on a Windows server with two Intel Xeon E5-2667 CPUs[4] and 128 GB of memory. The results show that predictive models usually finish their tasks in less time when they use the design matrix that is generated by the variable extraction approach. This is reasonable, because the design matrix that is generated by the variable extraction approach contains considerably fewer columns than the one generated by the variable selection approach.

Table 3. Running Time of Predictive Models

Data	Neural Network	HP SVM	HP Forest	Decision Tree	Regression
SVD 50	8.418	8.451	9.276	8.146	8.671
TERM 500	9.797	10.841	13.613	14.161	9.449

`

The results presented in Table 2 and Table 3 show that when modeling nodes use the design matrix generated by the variable extraction approach, they can usually generate more accurate results in less time. However, the variable selection approach is still useful when model interpretability is important. For example, you need to use the original variables to build your model so that you can understand their impact on your predictive modeling process.

CONCLUSION

Sparse data sets pose challenges when SAS users need to store and process them in a distributed computing environment. This paper introduces the major sparse formats that are used by SAS procedures for handling sparse matrices. The new Base64-encoded sparse row format, supported in the HPTMINE procedure, provides a powerful way for SAS users to handle their large-scale sparse data in a grid environment. This paper also presents a few simple but very effective techniques that help you process your sparse data and generate outputs that are suitable for analysis by traditional SAS modeling procedures.

APPENDIX

The following code creates the data that are used in the paper:

```
%include "coo2full.sas";
%include "coo_filter.sas";

proc hptmine data=hockeybaseball;
    doc_id _document_; var text; target rec_baseball;
    parse termwgt=mi cellwgt=log stop=sashelp.engstop
          outterms=keys outparent=out_coo;
    svd k=50 outdocpro=out_svd;
    performance details;
run;

%let ncols = 500;

data parent_keys;
set keys; if parent=.; w=sqrt(numdocs)*weight;
run;

%coo_filter( infile=out_coo, rowid=_document_, colid=_termnum_,
             keys=parent_keys, key_var=key, weight_var=w,
             oufile=out_coo_filtered, ncols=&ncols );

%coo2full( infile=out_coo_filtered, rowid=_document_,colid=_termnum_,
           val=_count_, oufile=out_full, keys=key_map);

proc sql;
```

[4] Each Intel Xeon E5-2667 CPU has six cores and 15M cache, and its maximum turbo frequency is 3.5 GHz.

```
create table hockeybaseball_svd_full (compress=yes) as
    select a.*, b.*,c.rec_baseball
        from out_full a, hockeybaseball c,
             out_svd (rename=(col1-col50=svd1-svd50)) b
        where a._document_=b._document_ and b._document_= c._document_;
quit;
```

ACKNOWLEDGMENTS

The authors would like to thank Saratendu Sethi for his suggestions and Anne Baxter for her editorial contributions.

RECOMMENDED READING

Albright, R. 2004. *Taming Text with the SVD.* White paper. Cary, NC: SAS Institute Inc. Available at *ftp.sas.com/techsup/download/EMiner/TamingTextwiththeSVD.pdf*.

Pissanetzky, S. 1984. *Sparse Matrix Technology*. London: Academic Press.

The Base16, Base32, and Base64 Data Encodings. IETF. October 2006. RFC 4648. Retrieved March 18, 2010.

Information Technology -- Microprocessor-Systems – Floating-Point Arithmetic. ISO/IEC/IEEE 60559-2011. International Organization for Standardization.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Zheng Zhao	Russell Albright	James Cox
SAS Institute Inc.	SAS Institute Inc.	SAS Institute Inc.
SAS Campus Drive	SAS Campus Drive	SAS Campus Drive
Cary, NC, 27513	Cary, NC, 27513.	Cary, NC, 27513
Zheng.Zhao@sas.com	Russell.Albright@sas.com	James.Cox@sas.com