Sparse matrices are used early in data ingestion development

Once a well-trained machine learning model has been deployed, the data ingestion pipeline for that model will also be deployed. That pipeline will consist of a collection of tools and systems used to fetch, transform, and feed data to the machine learning system in production.

However, that pipeline cannot be finalized during the development of the machine learning model it feeds. Finalizing the process of data ingestion *before* models have been run and your hypotheses about the business use case have been tested often leads to lots of re-work. Early experiments almost always fail and you should be careful about investing large amounts of time in building a data ingestion pipeline until there is enough accumulated evidence that a deployed model will help the business.

Instead of building a complete data ingestion pipeline, data scientists will often use sparse matrices during the development and testing of a machine learning model. Sparse matrices are used to represent complex sets of data (e.g., word counts) in a way that reduces the use of computer memory and processing time.

There are Python libraries available in the **SciPy** package to work with sparse matrices. The code block below imports this library as well as NumPy for calculations.

```
1  import numpy as np
2  from scipy import sparse
```

Sparse matrices offer a middle-ground between a comprehensive data warehouse solution with extensive test coverage and a directory of text files and database dumps. Sparse matrices do not work for all data types, but in situations where they are an appropriate technology you can leverage them even under load in production. Lets use an example to see how this process might play out.

A sparse matrix is one in which most of the values are zero. If the number of zero-valued elements divided by the size of the matrix is greater than 0.5 then it is consider *sparse*.

```
1  A = np.random.randint(0,2,100000).reshape(100,1000)
2  sparcity = 1.0 - (np.count_nonzero(A) / A.size)
3  print(round(sparcity,4))
4
```

```
1  0.5038
```

Very large matrices require significant amounts of memory. If we make a matrix of counts for a document or a book where the features are all known English words, the chances are high that your personal machine does not have enough memory to represent it as a dense matrix. Sparse matrices have the additional advantage of getting around time-complexity issues that arise with operations on large dense matrices.

WARNING: Many of the common functions like np.dot do not work on sparse matrices. See the scipy.sparse docs to learn about the specific functions for matrix products.

Some of the common applications of sparse matrices are:

- word counts with a large vocabulary
- recommender systems
- large networks

There are different types of sparse matrix representations in Python available through SciPy. The most commonly used are:

coo_matrix

sparse matrix built from the COOrdinates and values of the non-zero entries.

```
A = np.random.poisson(0.3, (10,100))
B = sparse.coo_matrix(A)
C = B.todense()

print("A",type(A),A.shape,"\n"
      "B",type(B),B.shape,"\n"
      "C",type(C),C.shape,"\n")

```

```
A <class 'numpy.ndarray'> (10, 100)
B <class 'scipy.sparse.coo.coo_matrix'> (10, 100)
C <class 'numpy.matrix'> (10, 100)

```

csc_matrix

When there are repeated entries in the rows or cols, we can remove the redundancy by indicating the location of the first occurrence of a value and its increment instead of the full coordinates. When the repeats occur in colums we use a CSC format.

```
A = np.random.poisson(0.3, (10,100))
B = sparse.csc_matrix(A)
```

csr_matrix

Like the CSC format there is a CSR format to account for data that repeat along the rows

```
A = np.random.poisson(0.3, (10,100))
B = sparse.csr_matrix(A)
```

Because the coordinate format is easier to create, it is common to create it first then cast to another more efficient format. Let us first show how to create a matrix from coordinates:

```
rows = [0,1,2,8]
cols = [1,0,4,8]
vals = [1,2,1,4]

A = sparse.coo_matrix((vals, (rows, cols)))
print(B.todense())

```

```
[[0 1 0 0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 4]]

```

Then to cast it to a CSR matrix

```
B = A.tocsr()
```

Because this introduction to sparse matrices is applied to data ingestion we would need to be able to:

1. concatenate matrices (e.g., add a new user to a recommender matrix)
2. read and write the matrices to and from disk

```
## matrix merge example
C = sparse.csr_matrix(np.array([0,1,0,0,2,0,0,0,1]).reshape(1,9))
print(B.shape,C.shape)
D = sparse.vstack([B,C])
print(D.todense())

```

```
[[0 1 0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 4]
 [0 1 0 0 2 0 0 0 1]]
```

```
## read and write
file_name = "sparse_matrix.npz"
sparse.save_npz(file_name, D)
E = sparse.load_npz(file_name)
print(E.shape)
```

```
(10, 9)
```

As you can see the syntax is very similar to NumPy. Additionally, sklearn's train_test_split is scipy.sparse matrices aware so you can call it directly.

There are other tools that walk the line between polished and prototype when it comes to data ingestion.

- Hive is a powerful alternative
- MongoDB when the data have a reasonable amount of structure
- IBM's data refinery
- IBM's InfoSphere DataStage

Additional resources

- Breaking the 80/20 rule: How data catalogs transform data scientists productivity
- Data preprocessing in detail
- Automating low-level tasks for data scientists