Sparse data with Scipy

In the rest of the lecture notes, the word “matrix” will be used a lot. If you are not comfortable with it, you can just think of it as a two-dimensional array. When such an object contains a very small amount of information compared to its size (a matrix with mostly zeros or a dataframe with a lot of NaN values), the data is called sparse.

1. Why sparse data ?

Sparse data is present in a lot of domains:

- Search engines: a graph of the web is sparse, so using ranking algorithms like PageRank to rank websites requires calculations involving sparse data.
- Natural Language Processing: working on a set of text documents and a vocabulary requires building sparse graphs. The computations required use highly sparse vectors and matrices.
- Digital image processing: when the processed images have a lot of black pixels.
- Studying communication networks: communication networks or social networks are often sparse (lots of nodes but few links). Using community detection algorithms or other tools requires computations with sparse matrices (since a network/graph can be represented by its adjacency matrix).

First, sparse data should be stored in an efficient way. Indeed, we do not want to use a large amount of memory for storing a lot of zeros.

Second, computations with sparse data can be a problem if we use regular dense two-dimensional arrays in Python. Indeed, multiplying two sparse matrices will involve multiplying and adding a lot of zeros, which are useless operations that take time.

2. How is it stored and handled?

The library SciPy provides a sparse matrices module. It provides several types that allow to store spare data in a memory efficient way, but also implements basic arithmetic operations so that computations with sparse matrices focus on “useful” operations.

We will talk about the three most important sparse matrix formats provided by SciPy. To choose the right format for the right application, it will be important to consider what kind of operations we want to do with the matrix: * accessing data often * entering new data often * vector matrix multiplication * ...

2.1 The coordinate format: COO

A COO matrix is a sparse matrix in the coordinate format. The coordinates of non null values and the values are stored. It takes the shape of three arrays:

- an array *i* of row indices
- an array *j* of column indices
- an array *data* of data

The value at position *(i[k], j[k])* in our matrix is *data[k]*.

Advantages:

- easy to construct
- facilitates fast conversion among sparse formats
- very fast conversion to and from CSR/CSC formats

Disadvantages:

- does not support direct arithmetic operations
- no efficient slicing

Usage suggestion: Use it to construct a sparse matrix and directly convert it to one of the two formats we will present next. It will then allow fast operations.

When initializing a COO matrix, one can specify the type of the values that will be stored in the matrix:

```
>>> import numpy as np
>>> from scipy.sparse import coo_matrix
>>> # Here we directly convert the matrix to an numpy array only to visualize the result
>>> coo_matrix((3, 4), dtype=np.int64).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])


>>> row  = np.array([0, 3, 1, 0])
>>> col  = np.array([0, 3, 1, 2])
>>> data = np.array([4, 5, 7, 9])
>>> print(coo_matrix((data, (row, col)), shape=(4, 4)))
(0, 0)      4
(3, 3)      5
(1, 1)      7
(0, 2)      9
```

2.2 The compressed sparse column format: CSC

A CSC matrix is a compressed sparse column matrix. The row indices for each column are stored in an array (*indices*) and the values are stored in a second array (*data*). A third array (*indptr*) is used to specify from which index to which index of the two previous arrays each row is. More precisely, the indices in *indices[indptr[i]:indptr[i+1]]* give the row indices where the data in *data[indptr[i]:indptr[i+1]]* is. So for each column *i*, we are given the row indices and the values indices. Notice that the values of a single column are stored consecutively in *data* but not at all elements of the same row. That is why this format is preferred if calculations impose column accesses and if we want to access slices of columns quickly.

Advantages:

- efficient arithmetic operations CSC + CSC, CSC * CSC, etc.
- efficient column slicing
- fast matrix vector products (CSR may be faster)

Disadvantages:

- slow row slicing operations (consider CSR)

Such a sparce matrix can be initialized from a dense matrix (numpy two-dimensional array or numpy matrix) or from another sparse matrix, whatever its format. It can also be initialized in the same way as a COO matrix, or by directly giving the three needed arrays

```
>>> from scipy.sparse import csc_matrix

>>> csc_matrix((3, 4), dtype=np.int8).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)


>>> row = np.array([0, 2, 2, 0, 1, 2])
>>> col = np.array([0, 0, 1, 2, 2, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> csc_matrix((data, (row, col)), shape=(3, 3)).toarray()
array([[1, 0, 4],
       [0, 0, 5],
       [2, 3, 6]])


>>> indptr = np.array([0, 2, 3, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> csc_matrix((data, indices, indptr), shape=(3, 3)).toarray()
array([[1, 0, 4],
       [0, 0, 5],
       [2, 3, 6]])

>>> dense_matrix = np.array([[1, 0, 4],[0, 0, 5],[2, 3, 6]])
>>> print(csc_matrix(dense_matrix))
(0, 0)      1
(2, 0)      2
(2, 1)      3
(0, 2)      4
(1, 2)      5
(2, 2)      6
```

Check the documentation for the list of all available operations and functions.

2.2 The compressed sparse row format: CSR

A CSR matrix is a compressed sparse row matrix. The format is the same as the CSC format but here the *indices* array stores columns indices and the *indptr* array gives ranges of *indices* and *data* arrays that correspond to a specific row (instead of a specific column).

This format is used exactly in the same way as CSC but the computation against vectors is faster.