Announcements

HW1

- Due Tue 2/8
- Keeping an eye on Yelp changes

HW2

- Plan: Out Wed 2/9, Due Tue 2/22

Plan

Wrap up Visualization

- ~~SQLite examples~~
- ~~DB joins~~

Matrices, vectors, and linear algebra

15-388/688 - Practical Data Science: Matrices, vectors, and linear algebra

Pat Virtue
Carnegie Mellon University
Spring 2022

Slide credits: CMU AI, Zico Kolter

Outline

Matrices and vectors

Basics of linear algebra

Libraries for matrices and vectors

Sparse matrices

Outline

Matrices and vectors

Basics of linear algebra

Libraries for matrices and vectors

Sparse matrices

Vectors

A vector is a 1D array of values

We use the notation $x \in \mathbb{R}^n$ to denote that x is an n-dimensional vector with real-valued entries

$\vec{x}$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

We use the notation x_i to denote the *i*th entry of x

By default, we consider vectors to represent *column* vectors, if we want to consider a row vector, we use the notation x^T

Matrices

A matrix is a 2D array of values

We use the notation $A \in \mathbb{R}^{m \times n}$ to denote a real-valued matrix with m rows and n columns

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{bmatrix}$$

We use A_{ij} to denote the entry in row i and column j

Use the notation $A_{i:}$ to refer to row i, $A_{:j}$ to refer to column j (sometimes we'll use other notation, but we will define before doing so)

$A_{i:}$ $A_{i:}$ $A_{:j}$

Matrices and linear algebra

Matrices are:

1. The "obvious" way to store tabular data (particularly numerical entries, though categorical data can be encoded too) in an efficient manner

2. The foundation of linear algebra, how we write down and operate upon (multi-variate) systems of linear equations

Understanding both these perspectives is critical for virtually all data science analysis algorithms

Matrices as tabular data

Given the "Grades" table from our relation data lecture

Person ID	HW1 Grade	HW2 Grade
5	100	80
6	60	80
100	100	100

Natural to represent this data (ignoring primary key) as a matrix

$$A \in \mathbb{R}^{3\times 2} = \begin{bmatrix} 100 & 80 \\ 60 & 80 \\ 100 & 100 \end{bmatrix}$$

Row and column ordering

Matrices can be laid out in memory by row or by column

$$A = \begin{bmatrix} 100 & 80 \\ 60 & 80 \\ 100 & 100 \end{bmatrix}$$

Row major ordering: $100, 80, 60, 80, 100, 100$

Column major ordering: $100, 60, 100, 80, 80, 100$

Row major ordering is default for C 2D arrays (and default for Numpy), column major is default for FORTRAN (since a lot of numerical methods are written in FORTRAN, also the standard for most numerical code)

Higher dimensional matrices

From a data storage standpoint, it is easy to generalize 1D vector and 2D matrices to higher dimensional ND storage

"Higher dimensional matrices" are called *tensors*

There is also an extension or linear algebra to tensors, but be aware: most tensor use cases you see are *not* really talking about true tensors in the linear algebra sense

1D array	2D array	ND array 3D+ array
vector	matrix	tensor

Outline

Matrices and vectors

Basics of linear algebra

Libraries for matrices and vectors

Sparse matrices

Systems of linear equations

Matrices and vectors also provide a way to express and analyze systems of linear equations

Consider two linear equations, two unknowns

$$
\begin{aligned}
4x_1 - 5x_2 &= -13 \\
-2x_1 + 3x_2 &= 9
\end{aligned}
$$

Ax $\quad\quad$ b

We can write this using matrix notation as

$$Ax = b$$

$$A = \begin{bmatrix} 4 & -5 \\ -2 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} -13 \\ 9 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Basic matrix operations

For $A, B \in \mathbb{R}^{m \times n}$, matrix addition/subtraction is just the elementwise addition or subtraction of entries

$$C \in \mathbb{R}^{m \times n} = A + B \iff C_{ij} = A_{ij} + B_{ij}$$

$$\boxed{C} = \boxed{A} + \boxed{B}$$

For $A \in \mathbb{R}^{m \times n}$, transpose is an operator that "flips" rows and columns

$$C \in \mathbb{R}^{n \times m} = A^T \iff C_{ji} = A_{ij}$$

For $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$ matrix multiplication is defined as

$$C \in \mathbb{R}^{m \times p} = AB \iff C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

- Matrix multiplication is associative, distributive, *but not commutative*

$$A(BC) = (AB)C \qquad A(B + C) = AB + AC \qquad (AB \neq BA)$$

Basic matrix operations

For $A, B \in \mathbb{R}^{m \times n}$, matrix addition/subtraction is just the elementwise addition or subtraction of entries

$$C \in \mathbb{R}^{m \times n} = A + B \iff C_{ij} = A_{ij} + B_{ij}$$

For $A \in \mathbb{R}^{m \times n}$, transpose is an operator that "flips" rows and columns

$$C \in \mathbb{R}^{n \times m} = A^T \iff C_{ji} = A_{ij}$$

For $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$ matrix multiplication is defined as

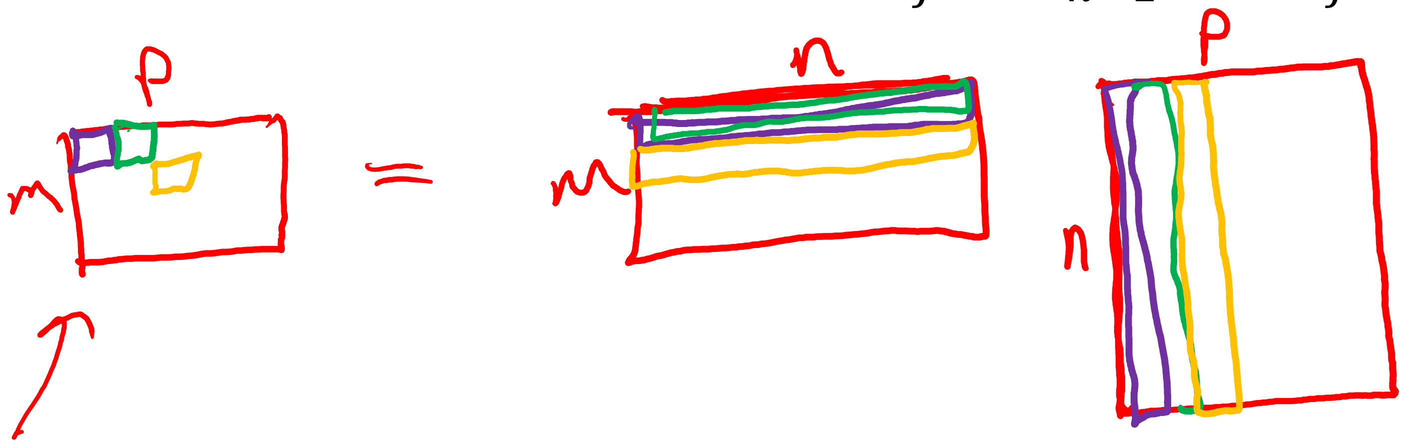
$$C \in \mathbb{R}^{m \times p} = AB \iff C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

- Matrix multiplication is associative, distributive, *but not commutative*

$$A(BC) = (AB)C \qquad A(B + C) = AB + AC \qquad (AB \neq BA)\ A)$$

Basic matrix operations

Matrix multiplication $\quad C = AB \iff C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$



$[m \times p] = [\underline{m} \times n][n \times \underline{p}]$

$$\begin{bmatrix} 1 & & & 0 & 0 \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ 0 & 0 & & & 1 \end{bmatrix}$$

Matrix inverse

The identity matrix $I_n \in \mathbb{R}^{n \times n}$ is a square matrix with ones on diagonal and zeros elsewhere, has property that for $A \in \mathbb{R}^{m \times n}$

$$AI = IA = A \text{ (for different sized } I\text{)}$$

For a square matrix $A \in \mathbb{R}^{n \times n}$, matrix inverse $A^{-1} \in \mathbb{R}^{n \times n}$ is the matrix such that

$$AA^{-1} = I = A^{-1}A$$

Recall our previous system of linear equations $Ax = b$, solution is easily written using the inverse

$$\underbrace{A^{-1}A}x = A^{-1}b \qquad x = A^{-1}b$$

Inverse need not exist for all matrices (conditions on linear independence of rows/columns of A), we will consider such possibilities later

Some miscellaneous definitions/properties

Transpose of matrix multiplication, $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}$

$$(AB)^T = B^T A^T$$

$(\quad)^T$

Inverse of product, $A \in \mathbb{R}^{n\times n}, B \in \mathbb{R}^{n\times n}$ *both square and invertible*

$$(AB)^{-1} = B^{-1}A^{-1}$$

Inner product: for $x, y \in \mathbb{R}^n$, special case of matrix multiplication

outer

$x y^T$

$$x^T y \in \mathbb{R} = \sum_{i=1}^{n} x_i y_i$$

$[n \times 1][1 \times n] = [n \times n]$

$x^T \quad y$

$[1 \times n][n \times 1] \to 1$

Vector norms: for $x \in \mathbb{R}^n$, we use $\|x\|_2$ to denote Euclidean norm

$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ $\quad \|x\|_2 = \sqrt{x_1^2 + x_2^2}$

$$\|x\|_2 = \left(x^T x\right)^{\frac{1}{2}}$$

N=M

Poll 1: Valid linear algebra expressions

N x N   N x M   M x N   N   M > N

Assume $A \in \mathbb{R}^{n\times n}, B \in \mathbb{R}^{n\times m}, C \in \mathbb{R}^{m\times n}, x \in \mathbb{R}^{n}$ with $m > n$. Which of the following are valid linear algebra expressions?

✗ 1. $A + B$ N x N + N x M

✓ 2. $A + BC$ N x N + [N x M][M x N]

✗ 3. $(AB)^{-1}$ [N x N][N x M]

✓ 4. $(ABC)^{-1}$ [M x N]

✗ 5. CBx [M x N][N x M][N x 1]

✗ 6. $Ax + Cx$ [N x N][N x 1] + [M x N][N x 1]

[N x 1] [M x 1]

x^T

UV^T

Outline

Matrices and vectors

Basics of linear algebra

Libraries for matrices and vectors

Sparse matrices

Software for linear algebra

Linear algebra computations underlie virtually *all* machine learning and statistical algorithms

There have been *massive* efforts to write extremely fast linear algebra code: don't try to write it yourself!

Example: matrix multiply, for large matrices, specialized code will be ~10x faster than this "obvious" algorithm

```
void matmul(double **A, double **B, double **C, int m, int n, int p) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            C[i][j] = 0.0;
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Numpy

In Python, the standard library for matrices, vectors, and linear algebra is Numpy

Numpy provides *both* a framework for storing tabular data as multidimensional arrays *and* linear algebra routines

Important note: numpy ndarrays are multi-dimensional arrays, *not* matrices and vectors (there are just routines that support them acting like matrices or vectors)

Specialized libraries

BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage) provide general interfaces for basic matrix multiplication (BLAS) and fancier linear algebra methods (LAPACK)

Highly optimized version of these libraries: ATLAS, OpenBLAS, Intel MKL

Anaconda typically uses a reasonably optimized version of Numpy that uses one of these libraries on the back end, but you should check

```
import numpy as np
print(np.__config__.show()) # print information on underlying libraries
```

Creating Numpy arrays

Creating 1D and 2D arrays in Numpy

```
b = np.array([-13,9])              # 1D array construction
A = np.array([[4,-5], [-2,3]])     # 2D array contruction

b = np.ones(4)             # 1D array of ones
b = np.zeros(4)            # 1D array of zeros
b = np.random.randn(4)     # 1D array of random normal entries

A = np.ones((5,4))         # 2D array of all ones
A = np.zeros((5,4))        # 2D array of zeros
A = np.random.randn(5,4)   # 2D array with random normal entries

I = np.eye(5)                 # 2D identity matrix (2D array)
D = np.diag(np.random(5))     # 2D diagonal matrix (2D array)
```

Indexing into Numpy arrays

Arrays can be indexed by integers (to access specific element, row), or by slices, integer arrays, or Boolean arrays (to return subset of array)

```
A[0,0]    # select single entry
A[0,:]    # select entire column
A[0:3,1] # slice indexing

# integer indexing
idx_int = np.array([0,1,2])
A[idx_int,3]

# boolean indexing
idx_bool = np.array([True, True, True, False, False])
A[idx_bool,3]

# fancy indexing on two dimensions
idx_bool2 = np.array([True, False, True, True])
A[idx_bool, idx_bool2]      # not what you want
A[idx_bool,:][:,idx_bool2] # what you want
```

Basic operations on arrays

Arrays can be added/subtracted, multiply/divided, and transposed, but these are *not* the same as matrix operations

```
A = np.random.randn(5,4)
B = np.random.randn(5,4)
x = np.random.randn(4)
y = np.random.randn(5)

A+B          # matrix addition
A-B          # matrix subtraction

A*B          # ELEMENTWISE multiplication
A/B          # ELEMENTWISE division
A*x          # multiply columns by x
A*y[:,None]  # multiply rows by y (look this one up)

A.T          # transpose (just changes row/column ordering)
x.T          # does nothing (can't transpose 1D array)
```

Basic matrix operations

Matrix multiplication done using the .dot() function or @ operator, special meaning for multiplying 1D-1D, 1D-2D, 2D-1D, 2D-2D arrays

```
A = np.random.randn(5,4)
C = np.random.randn(4,3)
x = np.random.randn(4)
y = np.random.randn(5)
z = np.random.randn(4)

A @ C        # matrix-matrix multiply (returns 2D array)
A @ x        # matrix-vector multiply (returns 1D array)
x @ z        # inner product (scalar)

A.T @ y      # matrix-vector multiply
y.T @ A      # same as above
y @ A        # same as above
#A @ y       # would throw error
```

There is also an np.matrix class … don't use it

Solving linear systems

$Ax = b$

$\rightarrow \quad x = A^{-1}b$

Methods for inverting a matrix, solving linear systems

```
b = np.array([-13,9])
A = np.array([[4,-5], [-2,3]])

np.linalg.inv(A)        # explicitly form inverse
np.linalg.solve(A, b)  # A^(-1)*b, more efficient and numerically stable
```

Important, always prefer to *solve a linear system* over directly forming the inverse and multiplying (more stable and cheaper computationally)

Details: solution methods use a factorization (e.g., LU factorization), which is cheaper than forming inverse

Complexity of operations

Assume $A, B \in \mathbb{R}^{n \times n}, x, y \in \mathbb{R}^n$

Matrix-matrix product AB: $O(n^3)$

Matrix-vector product Ax: $O(n^2)$

Vector-vector inner product $x^T y$: $O(n)$

Matrix inverse/solve: A^{-1}, $A^{-1}y$: $O(n^3)$

Important: Be careful about order of operations, $(AB)x = A(Bx)$ but the left one is $O(n^3)$ right is $O(n^2)$

Outline

Matrices and vectors

Basics of linear algebra

Libraries for matrices and vectors

Sparse matrices

Sparse matrices

Many matrices are sparse (contain mostly zero entries, with only a few non-zero entries)

Examples: matrices formed by real-world graphs, document-word count matrices (more on both of these later)

Storing all these zeros in a standard matrix format can be a huge waste of computation and memory

Sparse matrix libraries provide an efficient means for handling these sparse matrices, storing and operating only on non-zero entries

Coordinate format

There are several different ways of storing sparse matrices, each optimized for different operations

Coordinate (COO) format: store each entry as a tuple

$$(\text{row\_index}, \text{col\_index}, \text{value})$$

$$A = \begin{bmatrix} 0 & 0 & 3 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{bmatrix} \qquad \begin{aligned} \text{data} &= [2\ 4\ 1\ 3\ 1\ 1] \\ \text{row\_indices} &= [1\ 3\ 2\ 0\ 3\ 1] \\ \text{col\_indices} &= [0\ 0\ 1\ 2\ 2\ 3] \end{aligned}$$

Important: these could be placed in any order

A good format for constructing sparse matrices

Compressed sparse column format

Compressed sparse column (CSC) format

$$A = \begin{array}{c} \begin{array}{cccc} 0 & 2 & 3 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 3 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

data = [2 4 1 3 1 1]

row_indices = [1 3 2 0 3 1]

~~col_indices = [0 0 1 2 2 3]~~

col_indices = [0 2 3 5 6]

Ordering is important (always column-major ordering)

Faster for matrix multiplication, easier to access individual columns

Very bad for modifying a matrix, to add one entry need to shift all data

Compressed sparse column format

Compressed sparse column (CSC) format

$$A = \begin{bmatrix} 0 & 0 & 3 & 0 \\ 2 & 0 & 0 & \cancel{1}\,0 \\ 0 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{bmatrix}$$

$$\text{data} = [2\ 4\ 1\ 3\ 1\ \cancel{1}]$$
$$\text{row\_indices} = [1\ 3\ 2\ 0\ 3\ \cancel{1}]$$
$$\cancel{\text{col\_indices} = [0\ 0\ 1\ 2\ 2\ 3]}$$

$$\text{col\_indices} = [0\ 2\ 3\ 5\ \overset{5}{\cancel{6}}]$$

Ordering is important (always column-major ordering)

Faster for matrix multiplication, easier to access individual columns

Very bad for modifying a matrix, to add one entry need to shift all data

Sparse matrix libraries

Need specialized libraries for handling matrix operations (multiplication/solving equations) for sparse matrices

General rule of thumb (very adhoc): if your data is 80% sparse or more, it's probably worthwhile to use sparse matrices for multiplication, if it's 95% sparse or more, probably worthwhile for solving linear systems)

The `scipy.sparse` module provides routines for constructing sparse matrices in different formats, converting between them, and matrix operations

```
import scipy.sparse as sp
A = sp.coo_matrix((data, (row_idx, col_idx)), size)
B = A.tocsc()
C = A.todense()
```