Linear Algebra and Optimization Review

Thai Dinh

November 10, 2018

About me

- B.Sc., Summa cum Laude, University of Oklahoma
- M.Sc., University of Wisconsin-Madison
- Publication venues: Mathematical Programming, Nature Energy
- Academic father: Stephen Wright (Numerical Optimization author)
- Academic grandfather: George Nemhauser (John von Neumann Theory Prize winner)
- Just goes to show how an apple can fall **VERY** far from its tree

Outline

Matrices and vectors

- Matrix: Rectangular array of numbers

$$m1 = \begin{bmatrix} 23 & 402 \\ 69 & 221 \\ 118 & 0 \end{bmatrix}$$

- Dimension of matrix: number of rows x number of columns
- Vector: n x 1 matrix

$$v1 = \begin{bmatrix} 149 \\ 92 \\ 313 \end{bmatrix}$$

Matrices and vectors

Python snippet

```
import numpy as np
m1 = np.array([[23,402],[69,221],[118,0]])
print("m1={}".format(m1))
print(m1.shape)
v1 = np.array([149,92,313])
print("v1={}".format(v1))
```

Matrix addition

$$\begin{bmatrix} 23 & 402 \\ 69 & 221 \\ 118 & 0 \end{bmatrix} + \begin{bmatrix} 93 & 21 \\ 223 & 11 \\ 123 & 6 \end{bmatrix} = \begin{bmatrix} 116 & 423 \\ 292 & 232 \\ 241 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 23 & 402 \\ 69 & 221 \\ 118 & 0 \end{bmatrix} + \begin{bmatrix} 93 & 21 \\ 223 & 11 \end{bmatrix} = ???$$

Matrix addition

Python snippet

```
import numpy as np
m1 = np.array([[23,402],[69,221],[118,0]])
m2 = np.array([[93,21],[223,11],[123,6]])
print("m1+m2={}".format(m1+m2))
```

Scalar multiplication

$$3x\begin{bmatrix} 23 & 402 \\ 69 & 221 \\ 118 & 0 \end{bmatrix} = \begin{bmatrix} 69 & 1206 \\ 207 & 663 \\ 354 & 0 \end{bmatrix}$$

Python snippet

```
import numpy as np
m1 = np.array([[23,402],[69,221],[118,0]])
print("3*m1={}".format(3*m1))
```

Vector-vector multiplication

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1x4 + 2x5 + 3x6 \end{bmatrix} = 32$$

Matrix-vector multiplication

- Multiply each row of the matrix with the vector $\rightarrow$ an element of the resulting vector

$$\begin{bmatrix} 1 & 3 \\ 4 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \end{bmatrix} = \begin{bmatrix} 16 \\ 4 \\ 7 \end{bmatrix}$$

- An m x n matrix multiplied by an n x 1 vector is an m x 1 vector

Matrix-matrix multiplication

- Multiply the left hand side matrix with each column of the right hand side matrix $\rightarrow$ a column of the resulting matrix

$$\begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 0 & 1 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} 11 & 10 \\ 9 & 14 \end{bmatrix}$$

- An m x n matrix multiplied by an n x p matrix is an m x p matrix

Matrix-matrix multiplication

Python snippet

```
import numpy as np
m1 = np.array([[1,3,2],[4,0,1]])
m2 = np.array([[1,3],[0,1],[5,2]])
assert(m1.shape[1]==m2.shape[0])
m3 = np.dot(m1,m2)
print("m1*m2={}".format(m3))
print((m1.shape[0],m2.shape[1])==m3.shape)
```

Matrix multiplication properties

- Matrix multiplication is associative: $(AB)C = A(BC)$
- Matrix multiplication is distributive: $A(B + C) = AB + AC$
- Matrix Multiplication is NOT commutative in general, that is $AB \neq BA$

Determinant

- Determinant of **square** matrix A is denoted $det(A)$ or $|A|$
- In case of a 2 x 2 matrix

$$|A| = \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

- In case of an n x n matrix with $n > 2$...

Python snippet

```
import numpy as np
m1 = np.array([[23,42,79],[69,6,21],[8,0,9]])
print(np.linalg.det(m1))
```

Rank

Linearly independent

A set of vectors $x_1, x_2, ..., x_n$ is said to be linearly independent if no vector can be represented as a linear combination of the remaining vectors

- Rank of a matrix A is the size of the **largest** subset of columns of A that constitute a linearly independent set
- For an n x m matrix A, $rank(A) = rank(A^T)$ and $rank(A) \leq min(m, n)$. If $rank(A) = min(m, n)$, then A is **full rank**

Identity matrix

- Denoted I or I_{nxn}
- Examples of identity matrices

$$|I_{2x2}| = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- For any matrix A, $AI = IA = A$

Inverse

- Not all numbers have an inverse (e.g., 0)
- If a square matrix A has an inverse (denoted A^{-1}): $AA^{-1} = A^{-1}A = I$
- Matrices that don't have an inverse are **singular** or **degenerate**
- A matrix is singular iff its determinant is 0
- How to invert a matrix http://www.macs.hw.ac.uk/~markl/teaching/Inverses.pdf

Python snippet

```
import numpy as np
m1 = np.array([[0,5],[.5,0]])
print("inv(m1)={}".format(np.linalg.inv(m1)))
```

Transpose

Let A be an n x m matrix and let $B = A^T$. Then B is an m x n matrix and $B_{ij} = A_{ji}$

Properties

For any matrices X and Y

- $(X^T)^T = X$
- $(XY)^T = Y^T X^T$
- $(X + Y)^T = X^T + Y^T$

Eigenvalues and eigenvectors

For a square matrix A, λ is an eigenvalue and x is the coresponding eigenvector if $Ax = \lambda x$

How to find eigenvalues and eigenvectors

- For an eigenvalue λ of a matrix A, $(A - \lambda I)$ must not be singular (why?)
- Solve $det(A - \lambda I) = 0$ for λ
- Once you have an eigenvalue λ, solve $(\lambda I - A)x = 0$ to find the coresponding eigenvector x

Eigenvalues and eigenvectors

Python snippet

```
import numpy as np
m1 = np.diag(((1,2,3)))
w,v=np.linalg.eig(m1)
print("eigenvalues:{}".format(w))
print("eigenvectors:{}".format(v))
```

Quadratic forms

For an n x n matrix A and a vector x, the scalar value $x^T Ax$ is refered to as quadratic form

$$x^T Ax = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j$$

Positive definite and positive semidefinite

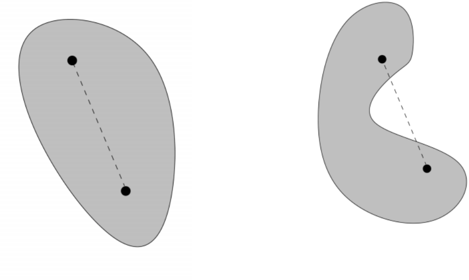
- A matrix A is said to be **positive definite** if for all non-zero vectors x, $x^T Ax > 0$
- A matrix A is said to be **positive semidefinite** if for all non-zero vectors x, $x^T Ax \geq 0$

Outline

1. Linear Algebra
   - Matrices and vectors
   - Addition and scalar multiplication
   - Matrix multiplication
   - Determinant, rank, inverse, and transpose
   - Eigenvalues and eigenvectors
   - Positive definite/semidefinite matrices
2. Optimization
   - Convex sets and convex functions
   - Matrix calculus
   - Unconstrained convex optimization

Convex sets

A set X is **convex** if $\forall x, y \in X, \lambda x + (1-\lambda)y \in X$ for any $\lambda \in [0,1]$



(a) convex set (b) non-convex set

Convex functions

A function f is **convex** if its domain (denoted $D(f)$) is a convex set and if, for all $x, y \in D(f)$ and $\lambda \in [0, 1]$,

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y))$$

Gradient and hessian

Gradient

$$(\nabla f(x))_i = \frac{\delta f(x)}{\delta x_i}$$

Hessian

$$(\nabla^2 f(x))_{ij} = \frac{\delta^2 f(x)}{\delta x_i \delta x_j}$$

Common forms of derivatives

$$\frac{\delta(x^T a)}{\delta x} = \frac{\delta(a^T x)}{\delta x} = a$$

$$\frac{\delta(x^T A x)}{\delta x} = (A + A^T)x$$

Taylor's theorem

- Suppose f is a continuously differentiable function and p is any vector,

$$f(x+p) = f(x) + \nabla f(x+tp)^T p,$$

for some $t \in (0,1)$.

- Moreover, if f is twice continuously differentiable,

$$f(x+p) = f(x) + \nabla f(x)^T p + \frac{1}{2} p^T \nabla^2 f(x+tp) p,$$

for some $t \in (0,1)$

Local/global solutions

- A point x^* is a global minimizer if $f(x^*) \leq f(x)$ for all x
- A point x^* is a local minimizer if there is a neighborhood N of x^* such that $f(x^*) \leq f(x)$ for $x \in N$
- A point x^* is an isolated local minimizer if there is a neighborhood N of x^* such that x^* is the only local minimizer in N
- A point x^* is a strict local minimizer if there is a neighborhood N of x^* such that $f(x^*) < f(x)$ for all $x \in N$ and $x \neq x^*$

Necessary and sufficient conditions

- First-order necessary conditions: If x^* is a local minimizer and f is continuously differentiable in an open neighborhood of x^*, then $\nabla f(x^*) = 0$
- Second-order necessary conditions: If x^* is a local minimizer of f and $\nabla^2 f$ exists and is continuous in an open neighborhood of x^*, then $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is positive semidefinite
- Second-order sufficient conditions: Suppose $\nabla^2 f$ is continuous in an open neighborhood of x^* and that $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is positive definite, then x^* is a strict local minimizer of f
- Bonus: When f is convex, any local minimizer x^* is a global minimizer of f

Line search methods

Overview: Algorithm chooses a direction and searches along this direction from the current iterate for a new iterate with a lower function value

Updates: $x_{k+1} = x_k + \alpha_k p_k$

Search directions:

- Descent directions: $p_k^T \nabla f_k < 0$
- Widely chosen: $p_k = -B_k^{-1} \nabla f_k$
- $B_k = I \rightarrow$ Gradient Descent Method
- $B_k \approx \nabla^2 f(x_k) \rightarrow$ Quasi-Newton Methods