Lecture 4

Arrays and Matrices

While linear algebra is not technically a requirement for our course, it is at the heart of much of machine learning, particularly regression. In MTH 325, we did a crash course in basic matrix and vector operations. Here we will look at implementing those same operations in R. We have seen some of them already in our optimization algorithms, and in the next lecture we'll be working with the normal equation to solve regression problems.

What is a Matrix?
A matrix is a two-dimensional array of numbers, with rows and columns. In R, a matrix is a special kind of array that allows mathematical operations.

```
# Create a matrix with numbers from 1 to 6
A <- matrix(1:6, nrow = 2, ncol = 3)
print(A)

# Creating a 3x3 matrix
B <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3)
print(B)
```

Addition and Subtraction of Matrices

```
# Creating two matrices
A <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
B <- matrix(c(5, 6, 7, 8), nrow = 2, ncol = 2)

# Addition
C <- A + B
print(C)

# Subtraction
D <- A - B
print(D)
```

Matrix Multiplication
There are two different operations in R for matrix multiplication. One multiplies element-wise, and one is true matrix multiplication using the rules of linear algebra. Some algorithms will use one or the other operation. You need to pay attention to which is intended. In regression, we use true matrix multiplication, but some neural network models use element-wise.

```
# Matrix multiplication (element-wise)
E <- A * B
print(E)

# Matrix multiplication (true matrix multiplication)
F <- A %*% B
print(F)
```

Transposing a Matrix
The transpose is an operation we'll encounter in SVD (singular value decomposition) and in the normal equation.

```
# Transposing a matrix
G <- t(A)
print(G)
```

Finding Eigenvalues and Eigenvectors
Eigenvalues and eigenvectors are essential in understanding the structure of a matrix and have applications in PCA, dimensionality reduction, etc. The basic idea is that matrices scale inputs linearly in particular characteristic directions. The directional vectors on which this happens are the eigenvectors, and the amount of scaling in that direction are the eigenvalues. Some eigenpairs are complex and produce rotations rather than directional scaling, but in most of the examples we will consider in statistics have real eigenpairs.

```
# Calculate eigenvalues and eigenvectors
eigen_B <- eigen(B)
print(eigen_B$values)    # Eigenvalues
print(eigen_B$vectors)   # Eigenvectors
```

Matrix Decomposition
Matrix decomposition methods aim to separate matrices that may be difficult to perform operations on into more matrices that may make operations easier to compute. This technique can be especially important for large matrices where the number of computations can be extremely large. We'll look at a couple of examples here that are frequently discussed in a linear algebra course, and then work our way up to singular value decomposition, which has applications in machine learning.

LU Decomposition:
LU decomposition decomposes a matrix into a lower triangular matrix (L) and an upper triangular matrix (U). To find the decomposition by hand typically involves row-reduction.

```
install.packages("Matrix")  # Install if not already installed
library(Matrix)
# Create a non-singular matrix
B <- matrix(c(2, 1, 1,
              4, -6, 0,
              -2, 7, 2), nrow = 3, byrow = TRUE)

# Perform LU Decomposition
LU_decomp <- lu(B)

# Extract the L (lower triangular) and U (upper triangular) matrices
L <- expand(lu(B))$L
U <- expand(lu(B))$U
P <- expand(lu(B))$P  # The permutation matrix

# Display the matrices
print("Matrix L:")
print(L)
```

```
print("Matrix U:")
print(U)

print("Permutation Matrix P:")
print(P)
```

This process will not work for singular matrices.

QR Decomposition:
This method decomposes a matrix into Q, an orthonormal matrix (with he property that $Q^T = Q^{-1}$), and R, a triangular matrix.

```
# QR Decomposition
QR_decomp <- qr(B)
Q <- qr.Q(QR_decomp)
R <- qr.R(QR_decomp)

print(Q)
print(R)
```

Singular Value Decomposition:
Singular Value Decomposition (SVD) is a powerful matrix factorization technique used in various applications, including dimensionality reduction, noise reduction, and more. SVD decomposes a matrix AAA into three matrices: U, Σ, and V^T.

Singular Value Decomposition (SVD) Overview:
Given a matrix A of dimensions $m \times n$, SVD decomposes A as: $A = U\Sigma V^T$
U is an $m \times m$ orthogonal matrix. Σ is an $m \times n$ diagonal matrix with non-negative real numbers on the diagonal, known as singular values. V^T is the transpose of an $n \times n$ orthogonal matrix V.

```
# Create a 4x3 matrix
A <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), nrow = 4, byrow = TRUE)
print("Matrix A:")
print(A)

# Perform Singular Value Decomposition
svd_A <- svd(A)

# U matrix
U <- svd_A$u
print("Matrix U:")
print(U)

# Sigma (Diagonal) matrix
Sigma <- diag(svd_A$d)
print("Matrix Sigma:")
print(Sigma)

# V^T matrix
```

```
V_T <- svd_A$v
print("Matrix V^T:")
print(t(V_T))

# Reconstruct A using U, Sigma, and V^T
A_reconstructed <- U %*% Sigma %*% t(V_T)
print("Reconstructed Matrix A:")
print(A_reconstructed)
```

- U Matrix: This matrix contains the left singular vectors of A.
- Sigma Matrix: This diagonal matrix contains the singular values of A. These values are non-negative and sorted in descending order.
- V Matrix: This matrix contains the right singular vectors of A.

When we multiply these matrices together, we should recover the original matrix A. In practice, due to numerical precision, the reconstructed matrix might have very small differences from the original, but it should be nearly identical.

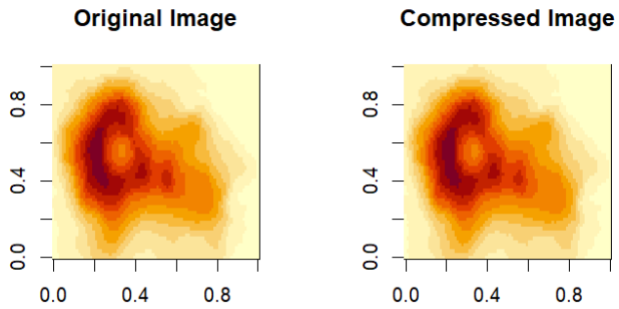
Applications of SVD:

- Dimensionality Reduction: By keeping only the top k singular values and corresponding vectors, we can approximate the original matrix with reduced dimensions, which is useful in data compression and noise reduction.
- Principal Component Analysis (PCA): SVD is closely related to PCA, where it is used to find the principal components of the data.
- Latent Semantic Analysis (LSA): In natural language processing, SVD is used for semantic analysis of textual data.

Practical Example:
SVD is a method of doing image compression.

```
# Using the 'volcano' dataset for image compression example
image_matrix <- as.matrix(volcano)

# Perform SVD
svd_image <- svd(image_matrix)

# Retain only the top 20 singular values for compression
k <- 20
U_k <- svd_image$u[, 1:k]
Sigma_k <- diag(svd_image$d[1:k])
V_k_T <- svd_image$v[, 1:k]

# Reconstruct the compressed image
compressed_image <- U_k %*% Sigma_k %*% t(V_k_T)

# Plot the original and compressed images
par(mfrow = c(1, 2))
```

```
image(volcano, main = "Original Image")
image(compressed_image, main = "Compressed Image")
```



This code snippet compresses the `volcano` dataset, which is a matrix representing topographic information, by retaining only the top 20 singular values. The resulting compressed image is a lower-dimensional approximation of the original.

SVD is a powerful tool with a variety of applications in data science and machine learning. Understanding how to perform and interpret SVD in R is essential for advanced data analysis, particularly in dimensionality reduction, data compression, and understanding the structure of matrices.

We'll work with the normal equation a bit more in the next lecture, but let's consider a brief introduction here.

The Normal Equation:
The Normal Equation is used to find the best-fit parameters (theta) for linear regression without relying on iterative methods like Gradient Descent. It's given by:

$$\theta = (X^T X)^{-1} X^T y$$

We discussed this method of finding regression coefficients in MTH 325. You can review your notes for a more detailed review of this equation. But, for this course, we want to perform the operations in R rather than by hand.

```
# Simulate some data
set.seed(987)
x <- matrix(rnorm(100), nrow = 50, ncol = 2)
y <- 1 + x[,1] * 2 + x[,2] * 3 + rnorm(50, 0, 0.1)

# Add a column of 1s for the intercept
X <- cbind(1, x)

# Compute theta using the Normal Equation
theta <- solve(t(X) %*% X) %*% t(X) %*% y
print(theta)
```

Resources:

1. https://www.geeksforgeeks.org/operations-on-matrices-in-r/#
2. http://www.philender.com/courses/multivariate/notes/matr.html
3. https://r-coder.com/matrix-operations-r/
4. https://stats.oarc.ucla.edu/r/seminars/r-matrix-operations/
5. https://www.math.ucla.edu/~yanovsky/Teaching/Math151B/handouts/GramSchmidt.pdf