Utils - SVD, PCA, Jacobi from Scratch

Bhanu Prasanna Koppolu

Table of contents

> **Note**
>
> Full implementation available at GitHub - ML from Scratch

Jacobi Eigenvalue Method

Eigen Values and Eigen Vectors

A non-zero column vector $\mathbf{v}$ is called the **eigen vector** of a Matrix A with the **eigen value** λ, if:

$$A\mathbf{v} = \lambda\mathbf{v}$$

Matrix Types

- **Dense matrix**: A type of matrix where most of elements are non-zero
- **Sparse matrix**: A type of matrix where most of the elements are zero

The Jacobi Method

The Jacobi method iteratively applies rotations to zero out off-diagonal elements until the matrix becomes diagonal.

Algorithm Steps

1. Find the largest off-diagonal element (for ij where $i \neq j$)
2. Compute rotation angle θ
3. Apply rotation to matrix A
4. Update eigenvector matrix V
5. Repeat until convergence

Finding Theta

$$\theta = \begin{cases} \frac{\pi}{4} \cdot \text{sign}(A_{ij}) & \text{if } A_{ii} = A_{jj} \\ \frac{1}{2} \arctan\left(\frac{2A_{ij}}{A_{ii} - A_{jj}}\right) & \text{otherwise} \end{cases}$$

Rotation Matrix

Place the rotation matrix at coordinates (i, j):

$$S = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Then: $B = S^T A S$

```
def largest_non_diagonal_element(A):
    max = -1 * np.inf
    cord = (0, 0)

    for i in range(A.shape[0]):
        for j in range(A.shape[1]):
            if i != j:
                if np.abs(A[i][j]) > max:
                    max = np.abs(A[i][j])
                    cord = (i, j)

    return max, cord

def find_theta(A, i, j):
    if A[i, i] == A[j, j]:
        return (np.pi / 4) * np.sign(A[i, j])
    return 0.5 * np.arctan((2 * A[i][j]) / (A[i][i] - A[j][j]))

def apply_rotation_to_sym(A, i, j, theta):
    S = np.identity(A.shape[0])

    S[i][i] = np.cos(theta)
    S[i][j] = -1 * np.sin(theta)
    S[j][i] = np.sin(theta)
    S[j][j] = np.cos(theta)

    B = S.T @ A @ S

    return B, S

def jacobi_eigenvalue(A, max_iter=1000, tol=1e-10):
    n = A.shape[0]
    A = A.copy()
```

```
    V = np.eye(n)

    for iteration in range(max_iter):
        max_val, (p, q) = largest_non_diagonal_element(A)

        if np.abs(max_val) < tol:
            print(f"Converged in {iteration} iterations")
            break

        theta = find_theta(A, p, q)
        A, S = apply_rotation_to_sym(A, p, q, theta)
        V = V @ S

    eigenvalues = np.diag(A)
    return eigenvalues, V
```

Singular Value Decomposition (SVD)

What is SVD?

SVD decomposes a matrix A into three matrices:

$$A = U\Sigma V^T$$

Where: - U is $m \times m$ (left singular vectors) - Σ is $m \times n$ diagonal matrix (singular values) - V is $n \times n$ (right singular vectors)

Traditional Method

1. Find U: Compute AA^T, then solve $(AA^T - \lambda I) = 0$ to find eigenvalues, then find eigenvectors
2. Find V: Compute A^TA, then solve similarly
3. Build Σ: Diagonal matrix with $\sqrt{\lambda_i}$

Important Detail

For U, column vectors are computed as:

$$U_i = \frac{1}{\sigma_i}(AV_i)$$

where V_i is the i-th column vector of V.

```
eig_a_at = np.linalg.eig(A @ A.T)  # Represents U
eig_at_a = np.linalg.eig(A.T @ A)  # Represents V

idx_u = np.argsort(eig_a_at.eigenvalues)[::-1]
idx_v = np.argsort(eig_at_a.eigenvalues)[::-1]

sorted_eigenvalues_u = eig_a_at.eigenvalues[idx_u]
```

```
sorted_eigenvectors_v = eig_at_a.eigenvectors[:, idx_v]

sigma_mat = np.eye(m, n)
for i in range(min(m, n)):
    sigma_mat[i][i] = sorted_eigenvalues_u[i] ** 0.5

V = sorted_eigenvectors_v
U = np.zeros_like(sorted_eigenvectors_u)
for i in range(m):
    U[:, i] = (A @ V[:, i]) / sigma_mat[i, i]
```

Image Compression with Rank-k Approximation

Why it Works

Full SVD:

$$A = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + ... + \sigma_r u_r v_r^T$$

Rank-k approximation:

$$A_k = \sigma_1 u_1 v_1^T + ... + \sigma_k u_k v_k^T$$

Why this works:

1. Singular values are sorted descending: $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq ...$
2. The first few singular values are much larger than later ones
3. They capture the most important patterns in the data
4. Later terms add tiny details

For Images

- Keep just $k = 50$ singular values instead of 1000
- Storage: $(m \times k + k + k \times n)$ instead of $(m \times n)$
- $A_k = U[:, :k] \cdot \Sigma[:k, :k] \cdot V[:k, :]^T$
- Compression ratio: huge for large images

```
from sklearn.datasets import load_sample_image
img = load_sample_image('flower.jpg')
gray_img = np.mean(img, axis=2).astype(np.uint8)

U, sigma, V = np.linalg.svd(gray_img)
```

```
k = 50  # Number of singular values to keep
sigma_k = np.diag(sigma[:k])
A_k = U[:, :k] @ sigma_k @ V[:k, :]
A_k_uint8 = np.clip(A_k, 0, 255).astype(np.uint8)
```

Principal Component Analysis (PCA)

The Difference Between SVD and PCA

- **SVD**: Finds the best rank-k approximation that minimizes reconstruction error
- **PCA**: Finds k directions that maximize captured variance

The PCA core idea is to find directions where the **variance (spread) is largest**.

Algorithm Steps

1. **Calculate mean**: For each column (feature), find the mean
2. **Center the data**: Subtract mean from each feature to get mean of 0
3. **Compute Covariance matrix**: Find how features spread and relate
4. **Find eigenvalues and eigenvectors** of covariance matrix
5. **Order eigenvectors** by eigenvalues (descending)
6. **Select top k eigenvectors**
7. **Transform**: $Z = X_{\text{centered}} \cdot V_k$

Why Center the Data?

- Subtracting ensures no single feature dominates due to its scale
- Makes all features contribute equally
- We don't care about position (where the data is), only about its shape/spread

```
def pca(X, n_components=5):
    if n_components > X.shape[1]:
        print(f"PCA Not possible: n_components must be less than {X.shape[1]}.")
        return np.array([])

    X = X.copy()

    # Step 1: Compute mean
    X_mean = X.mean(axis=0)
```

```
    # Step 2: Center the data
    X = (X - X_mean)

    # Step 3: Covariance matrix
    X_cov = np.cov(X.T)

    # Step 4: Eigen decomposition
    eig_values, eig_vectors = jacobi_eigenvalue(X_cov)

    # Step 5: Sort by eigenvalues
    idx_eigval = np.argsort(eig_values)[::-1]
    eig_values = eig_values[idx_eigval]
    eig_vectors = eig_vectors[:, idx_eigval]

    # Step 6: Select top k eigenvectors
    V_k = eig_vectors[:, :n_components]

    # Step 7: Transform
    Z = X @ V_k

    return Z
```

Note: The number of components k cannot be larger than the number of eigenvectors (number of features).