Solutions — Practical Session (M1 Computer Science)

Linear Algebra for Machine Learning — Python Implementations & Timings

Youssef SALMAN

Academic Year 2025–2026

General Notes

- Set random seeds for reproducibility when comparing timings.
- For time measurement, use either `time.time()` (wall time) or `timeit`.
- For numerical comparisons, use `np.allclose` with tolerances (e.g., `rtol=1e-6, atol=1e-8`).

Q1 — Determinant (Laplace Expansion) vs `numpy.linalg.det`

Reminder. Laplace expansion along the first row:

$$\det(A) = \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(M_{1j})$$

where M_{1j} is the minor of A (remove row 1 and column j). This grows very fast (about $O(n!)$).

Solution Code

```
import numpy as np, time

np.random.seed(0)

def minor_matrix(A, i, j):
    # remove row i and column j
    return np.delete(np.delete(A, i, axis=0), j, axis=1)

def det_laplace(A):
    A = np.asarray(A, dtype=float)
    n, m = A.shape
    assert n == m, "Matrix must be square."
    if n == 0:
```

```
        return 1.0
    if n == 1:
        return A[0, 0]
    if n == 2:
        return A[0,0]*A[1,1] - A[0,1]*A[1,0]
    # Expand along first row
    s = 0.0
    for j in range(n):
        cofactor = ((-1)**(0+j)) * A[0, j] * det_laplace(minor_matrix(A, 0, j))
        s += cofactor
    return s

def time_det(n, trials=3):
    t1 = t2 = 0.0
    max_err = 0.0
    for _ in range(trials):
        A = np.random.randn(n, n)
        t0 = time.time(); d1 = det_laplace(A); t1 += (time.time() - t0)
        t0 = time.time(); d2 = np.linalg.det(A); t2 += (time.time() - t0)
        max_err = max(max_err, abs(d1 - d2))
    speedup = (t1/t2) if t2 > 0 else float("inf")
    return t1/trials, t2/trials, max_err, speedup

for n in [3,5,7,10]:
    t_lap, t_np, max_err, ratio = time_det(n, trials=3)
    print(f"n={n} | Laplace {t_lap:.4f}s | NumPy {t_np:.6f}s | max|Delta|={
        max_err:.3e} | speedup={ratio:.1f}x")
```

Discussion

- Laplace expansion becomes intractable for $n \gtrsim 10$.
- `numpy.linalg.det` uses LU factorization ($\approx O(n^3)$), faster and more stable.
- Validate with `np.allclose(d1, d2, rtol=1e-6, atol=1e-8)`.

Q2 — Power Iteration for Dominant Eigenvalue vs `numpy.linalg.eig`

Reminder. Power Iteration:

1. Start with random b_0.
2. Iterate $b_{k+1} = \dfrac{Ab_k}{\|Ab_k\|}$.
3. Rayleigh quotient $\lambda_k = \dfrac{b_k^T A b_k}{b_k^T b_k}$ approximates the top eigenvalue.

Solution Code

```
import numpy as np, time
np.random.seed(1)

def power_iteration(A, max_iter=1000, tol=1e-8):
    A = np.asarray(A, dtype=float)
    n = A.shape[0]
    b = np.random.randn(n)
    b = b / np.linalg.norm(b)
    lam_old = 0.0
    for _ in range(max_iter):
        Ab = A @ b
        norm = np.linalg.norm(Ab)
        if norm == 0:
            return 0.0, b
        b = Ab / norm
        lam = b @ (A @ b) # Rayleigh quotient
        if abs(lam - lam_old) < tol:
            break
        lam_old = lam
    return lam, b

def time_power(n, trials=3):
    t1 = t2 = 0.0
    max_err = 0.0
    for _ in range(trials):
        X = np.random.randn(n, n)
        A = (X + X.T)/2 # symmetric
        t0 = time.time(); lam, v = power_iteration(A); t1 += (time.time() - t0)
        t0 = time.time(); w, V = np.linalg.eig(A); t2 += (time.time() - t0)
        lam_np = np.max(np.real(w))
        max_err = max(max_err, abs(lam - lam_np))
    return t1/trials, t2/trials, max_err

for n in [50, 100, 200]:
    t_pi, t_np, max_err = time_power(n)
    print(f"n={n} | PowerIter {t_pi:.4f}s | NumPy eig {t_np:.4f}s | max|
        Delta_lambda|={max_err:.3e}")
```

Discussion

- Converges to the largest-magnitude eigenvalue if unique and dominant; rate depends on the spectral gap.
- `numpy.linalg.eig` returns the full spectrum; power iteration is attractive if you only need the top pair for large n.

Q3 — First Singular Vector via Power Iteration vs `numpy.linalg.svd`

Reminder. Compute the top left singular vector u_1 as the dominant eigenvector of AA^T; then $v_1 = A^T u_1 / \|A^T u_1\|$ and $\sigma_1 = \|A^T u_1\|$.

Solution Code

```
import numpy as np, time
np.random.seed(2)

def top_singular_vector(A, max_iter=1000, tol=1e-8):
    # Power iteration on A A^T to get u1, then v1 = A^T u1 / ||A^T u1||
    A = np.asarray(A, dtype=float)
    m, n = A.shape
    u = np.random.randn(m)
    u = u / np.linalg.norm(u)
    lam_old = 0.0
    for _ in range(max_iter):
        u_new = A @ (A.T @ u)
        norm = np.linalg.norm(u_new)
        if norm == 0:
            break
        u = u_new / norm
        lam = u @ (A @ (A.T @ u))
        if abs(lam - lam_old) < tol:
            break
        lam_old = lam
    v = A.T @ u
    sv = np.linalg.norm(v)
    if sv > 0:
        v = v / sv
    return u, v, sv # left vec, right vec, top singular value (approx)

def time_svd(m, n, trials=3):
    t1 = t2 = 0.0
    max_err = 0.0
    for _ in range(trials):
        A = np.random.randn(m, n)
        t0 = time.time(); u, v, s1 = top_singular_vector(A); t1 += (time.time()
            - t0)
        t0 = time.time(); U, S, VT = np.linalg.svd(A, full_matrices=False); t2
           += (time.time() - t0)
        max_err = max(max_err, abs(s1 - S[0]))
    return t1/trials, t2/trials, max_err

for shape in [(100,100),(200,100),(100,200)]:
    t_top, t_svd, max_err = time_svd(*shape)
    print(f"{shape} | Power-Top-SV {t_top:.4f}s | NumPy SVD {t_svd:.4f}s | max|
       Delta_sigma1|={max_err:.3e}")
```

Discussion

- Power iteration yields only the top singular triplet (u_1, σ_1, v_1).
- `np.linalg.svd` returns the full spectrum and is typically more accurate; may be slower if you only need σ_1 on huge matrices.

Q4 — PCA: Manual vs `sklearn.decomposition.PCA`

Reminder. Manual PCA:

1. Center data: $X_c = X - \bar{X}$,
2. Covariance: $C = \frac{1}{n-1} X_c^T X_c$,
3. Eigen-decomposition of C,
4. Project onto top k eigenvectors.

Solution Code

```
import numpy as np, time
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA

digits = load_digits()
X = digits.data # shape (n_samples, n_features)

def pca_manual(X, k):
    # Center, covariance, eigen-decomposition, sort, project
    X = np.asarray(X, dtype=float)
    Xc = X - X.mean(axis=0, keepdims=True)
    C = (Xc.T @ Xc) / (Xc.shape[0]-1)
    w, V = np.linalg.eigh(C) # symmetric
    idx = np.argsort(w)[::-1]
    w, V = w[idx], V[:, idx]
    Wk, Vk = w[:k], V[:, :k]
    Z = Xc @ Vk
    return Z, Vk, Wk

k = 20

t0 = time.time()
Zm, Vm, Wm = pca_manual(X, k)
t_manual = time.time() - t0

t0 = time.time()
pca = PCA(n_components=k, svd_solver='full').fit(X)
Zsk = pca.transform(X)
t_sklearn = time.time() - t0

# Compare subspaces via principal angles: SVD of Vm.T * components.T
```

```
U, S, VT = np.linalg.svd(Vm.T @ pca.components_[:k].T, full_matrices=False)
subspace_cosines = S # in [0,1]; closer to 1 => better alignment

print(f"Manual PCA time: {t_manual:.4f}s")
print(f"sklearn PCA time: {t_sklearn:.4f}s")
print("Subspace cosines (principal angles):", subspace_cosines)
```

Discussion

- Manual PCA uses eigen-decomposition of covariance ($O(d^3)$ for d features).
- `sklearn` uses optimized SVD; typically faster/more stable, and can use randomized solvers.
- Compare timings and subspace alignment (cosines close to 1).

Bonus — Cosine Similarity: Manual vs `sklearn`

Reminder. For vectors x, y: $\cos(\theta) = \dfrac{x \cdot y}{\|x\| \, \|y\|}$.

Solution Code

```
import numpy as np, time
from sklearn.metrics.pairwise import cosine_similarity

np.random.seed(3)
X = np.random.randn(100000, 300).astype(np.float32)

def cosine_sim_manual(A, B):
    # Returns matrix of cosine similarities between rows of A and B
    A = A.astype(np.float64); B = B.astype(np.float64)
    An = A / (np.linalg.norm(A, axis=1, keepdims=True) + 1e-12)
    Bn = B / (np.linalg.norm(B, axis=1, keepdims=True) + 1e-12)
    return An @ Bn.T

# Smaller test for timing due to memory constraints
A = X[:5000]
B = X[5000:10000]

t0 = time.time()
M1 = cosine_sim_manual(A, B)
t_manual = time.time() - t0

t0 = time.time()
M2 = cosine_similarity(A, B)
t_sklearn = time.time() - t0

print(f"Manual cosine: {t_manual:.3f}s | sklearn: {t_sklearn:.3f}s")
print("max|Delta| =", np.max(np.abs(M1 - M2)))
```

Discussion

- Manual implementation (normalize then matrix multiply) matches `cosine_similarity`.
- `sklearn` uses efficient vectorization and is often faster; for very large N, compute in blocks to save RAM.