

Assignment 3: Principle Component Analysis and Power Iteration

1 Instructions

In this assignment, you will implement two fundamental algorithms from linear algebra and machine learning: **Power Iteration** and **Principal Component Analysis (PCA)**. The goal is to deepen your understanding of their mechanics, not just by implementing them correctly, but by identifying, diagnosing, and fixing common bugs.

For each of the two algorithms, you will produce a sequence of **three presentation slides** that walks through a buggy implementation, explains the errors, and presents the corrected version. This exercise emphasizes the importance of connecting mathematical theory to practical code.

2 Pseudo Algorithms & Mathematical Reference

2.1 Power Iteration

The **Power Iteration** method is an algorithm for finding the dominant eigenvector of a matrix—that is, the eigenvector corresponding to the eigenvalue with the largest absolute value. It's an iterative approach that is conceptually simple and relies on repeated matrix-vector multiplication. When applied to the covariance matrix of a dataset, Power Iteration

provides a way to find the first principal component, which is the direction of maximum variance in the data. This makes it a building block for understanding more complex dimensionality reduction techniques like PCA. The pseudo algorithm is given below.

Algorithm 1 Power Iteration

Input: A symmetric matrix $A \in \mathbb{R}^{d \times d}$, an initial unit vector v_0 , a maximum number of iterations T , and a tolerance $\varepsilon > 0$.

Output: A unit vector \hat{v} approximating the dominant eigenvector and its corresponding eigenvalue $\hat{\lambda}$.

```
1:  $v \leftarrow v_0 / \|v_0\|_2$ 
2: for  $t = 1$  to  $T$  do
3:    $w \leftarrow Av$ 
4:    $v_{\text{new}} \leftarrow w / \|w\|_2$ 
5:   if  $\|v_{\text{new}} - v\|_2 < \varepsilon$  then
6:     break
7:   end if
8:    $v \leftarrow v_{\text{new}}$ 
9: end for
10:  $\hat{v} \leftarrow v$ 
11:  $\hat{\lambda} \leftarrow \hat{v}^\top A \hat{v}$ 
12: return  $(\hat{v}, \hat{\lambda})$ 
```

Note: To find the first principal component of a centered data matrix X , you would apply this algorithm to the sample covariance matrix $S = \frac{1}{n-1} X^\top X$.

2.2 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a cornerstone of dimensionality reduction. It transforms a dataset into a new coordinate system where the axes (principal components) are ordered by the amount of variance they capture. This allows you to summarize the data using a smaller number of dimensions while retaining as much information as possible. The full PCA can be computed efficiently using either the Singular Value Decomposition (SVD) of the data matrix or the Eigendecomposition (EVD) of its covariance matrix. The pseudo algorithm is given below.

Algorithm 2 Principal Component Analysis (PCA)

Input: A data matrix $X \in \mathbb{R}^{n \times d}$ (where rows are samples), the number of components k , and a boolean $\text{UseSVD} \in \{\text{True}, \text{False}\}$ indicating whether to use SVD or eigendecomposition.

Output: The top k principal directions $V_k \in \mathbb{R}^{d \times k}$, the projected data (scores) $Z \in \mathbb{R}^{n \times k}$, and a diagonal matrix of the top k variances $\Lambda_k \in \mathbb{R}^{k \times k}$.

```
1:  $\mu \leftarrow \frac{1}{n} \sum_{i=1}^n x_i$ 
2:  $X_c \leftarrow X - \mathbf{1}\mu^\top$ 
3: if UseSVD then
4:    $X_c = U\Sigma V^\top$ 
5:    $V_k \leftarrow V_{[:,1:k]}$ 
6:    $\Lambda_k \leftarrow \Sigma_{1:k,1:k}^2 / (n - 1)$ 
7: else
8:    $S \leftarrow \frac{1}{n-1} X_c^\top X_c$ 
9:    $S = V\Lambda V^\top$ 
10:   $V_k \leftarrow V_{[:,1:k]}$ 
11:   $\Lambda_k \leftarrow \Lambda_{1:k,1:k}$ 
12: end if
13:  $Z \leftarrow X_c V_k$ 
14: return ( $V_k, Z, \Lambda_k$ )
```

3 Content Requirements

3.1 Slide Organization and Content

You will submit a total of **6 slides**: a three-slide sequence for Power Iteration, followed by a three-slide sequence for PCA. Details of each slide will be given below. Submit the following **6 slides**:

1. **Power Iteration — Code (no hints)**
2. **Power Iteration — Bug(s) & Explanation**

3. **Power Iteration — Fixed & Explanation**
4. **PCA — Code (no hints)**
5. **PCA — Bug(s) & Explanation**
6. **PCA — Fixed & Explanation**

3.1.1 Slide 1: Code (No Hints)

Present a Python implementation of the algorithm. This code must contain **at least one, and up to three, subtle bugs**. Do not include any comments or hints that reveal the errors.

3.1.2 Slide 2: Bug(s) & Explanation

Use a two-column layout (50% / 50%).

- **Left Column:** Display the same buggy code from Slide 1. Mark each error clearly with a comment, e.g., `# Bug 1`, `# Bug 2`.
- **Right Column:** For each numbered bug, provide a concise explanation. Describe what the bug is, why it is incorrect (referencing the math or logic of the algorithm), and how it might manifest (e.g., incorrect output, slow convergence, shape mismatch).

3.1.3 Slide 3: Fixed & Explanation

Use a two-column layout (50% / 50%).

- **Left Column:** Present the fully corrected code. Mark each fix with comments like `# Fix for Bug 1`.
- **Right Column:** Explain why each correction is the right fix, connecting it back to the mathematical definition. Conclude with a brief statement on why understanding this particular concept or fix is important for using the algorithm correctly in practice.

4 Format

Write your slides in Quarto RevealJS markdown format. All code should be in fenced code blocks (via ````python`). Use markdown subsection headers (`##`) to separate slides. **You should include a Quarto YAML header for RevealJS for this assignment so that we can render it.** An example template header for RevealJS is given in the example below.

Character Limit: 9000 (Your submission will be truncated after this limit.)

5 Example Slides — Matrix Multiplication (Implement AB)

Implement simple matrix multiplication AB .

```
---
format:
  revealjs:
    width: 1600
    height: 900
    scrollable: true
---

## Matrix Multiplication Code with Bug(s)

```python
import numpy as np
def matrix_multiply(A,B):
 # Intentionally no comments or hints on this slide
 C = A * B
 return C
```
---
```

```

## Matrix Multiplication Bug(s) & Explanation

:::: {.columns}
::: {.column width="50%"}
### Code
```python
import numpy as np
def matrix_multiply(A,B):
 C = A * B # Bug 1
 return C
```
:::

:::

::: {.column width="50%"}
### Bug 1: Incorrect Operator

- **What it is**: The code uses the `*` operator, which performs element-wise multiplication in NumPy, not the standard matrix product.
- **Why it's wrong**: The mathematical definition of a matrix product is  $(AB)_{ij} = \sum_k A_{ik} B_{kj}$ . The `*` operator computes  $C_{ij} = A_{ij} \cdot B_{ij}$ , which is a different operation (the Hadamard product).
- **Manifestation**: The code runs without error because the matrix shapes are compatible, but the resulting matrix `C` is mathematically incorrect.

:::
::::

## Fixed Matrix Multiplication Code & Explanation

:::: {.columns}
::: {.column width="50%"}
### Code
```python
import numpy as np

```

```

def matrix_multiply(A,B):
 C = A @ B # Fix for Bug 1
 return C
'''
'''

:::
::: {.column width="50%"}
Fix 1: Use the `@` Operator

- **Why it's correct**: The `@` operator was introduced in Python 3.5 specifically for
matrix multiplication. It correctly implements the $C_{ij} = \sum_k A_{ik} B_{kj}$ definition,
producing the expected linear algebraic result.

- **Importance**: Distinguishing between element-wise operations and matrix operations
is fundamental in scientific computing. Using the wrong operator can lead to silent,
hard-to-debug errors that produce plausible but incorrect results.

:::
::::

```

---

## 6 Rubric

Each of the four criteria below will be weighted equally (25% each).

---

<b>Criterion</b>	<b>Excellent (5)</b>	<b>Good (4)</b>	<b>Satisfactory (3)</b>	<b>Needs Improvement (2)</b>	<b>Unsatisfactory (1)</b>
<b>Final Correctness</b>	Implementations are fully correct, align with pseudo-algorithms, and produce numerically consistent results.	Mostly correct, with minor issues or unhandled edge cases.	Largely correct but with significant omissions or vague reasoning.	Major logical or mathematical errors remain in the final code.	Code is non-functional or does not implement the algorithms.
<b>Bug Identification</b>	Clearly identifies 1-3 distinct and plausible bugs; provides a clear explanation of their cause and impact.	Identifies bugs, but explanations of cause or impact are partial or slightly unclear.	Bugs identified, but explanations are vague or miss the core issue.	Bugs are trivial (e.g., typos), unclear, or poorly explained.	Fails to identify meaningful bugs or explanations are incorrect.
<b>Mathematical Grounding</b>	Explanations are consistently and accurately tied to mathematical concepts (e.g., normalization, centering, SVD).	Explanations make frequent reference to math, but with some minor gaps or inaccuracies.	Occasional math references are present, but explanations are mostly verbal.	Minimal or incorrect connection made to the underlying mathematics.	No meaningful connection to mathematical concepts is made.
<b>Bug Diversity &amp; Plausibility</b>	Bugs are realistic, non-trivial, and cover different potential failure modes (e.g., logic, math, preprocessing).	Bugs are plausible but may lack diversity (e.g., multiple bugs are of a similar type).	Bugs lack diversity or are overly simple and obvious.	Bugs feel artificial, redundant, or are not plausible real-world errors.	No meaningful or plausible bugs are included.

---