

**ASSIGNMENT - 2**

[**Google Colab**](https://colab.research.google.com/drive/1GcbhjWKmMqbFxqJsL9g5kNAv_ktFA0Kp?authuser=1#scrollTo=HicGUbJqUZ_E)

SAINZOLBOO ANUJIN – 1311002

## Problem 1 – Curse of Dimensionality

d-dimensional unit ball of radius ( *R* ) is defined:

* **Part (A)**

For unit ball,

**For :**

* *B*(2) represents a unit circle (disk) in 2D space
* All points such as that
* This is a filled circle with radius 1 centered at origin

**For :**

* *B*(3) represents a unit sphere (ball) in 3D space
* All points such as that
* This is a filled sphere with radius 1 centered at origin
* **Part (B) - Function to Generate (N) Random Points**

Uniform distribution steps:

1. Generate random directions
2. Normalize to unit sphere: Divide by L2 norm to get uniform directions
3. Scale by random radius: For uniform distribution in d-dimensional ball we need to multiply by where

def generate\_points(n, d):

#Step 1

points = np.random.normal(0, 1, (n, d))

#Step 2

norms = np.linalg.norm(points, axis=1, keepdims=True)

norms = np.where(norms == 0, 1, norms)

unit\_points = points / norms

# Step 3

rad = np.random.uniform(0, 1, (n, 1)) \*\* (1/d)

return unit\_points \* rad

* **Part (C) - Distance Calculation Function**

def freq\_plot(d\_dim, iterations=10000):

dist = []

for i in range(iterations):

# Generate two random points in d-dimensional unit ball

point1 = generate\_points(1, d\_dim)[0]

point2 = generate\_points(1, d\_dim)[0]

dist.append(np.linalg.norm(point1 - point2))

plt.figure()

plt.hist(dist, range=[0, math.sqrt(d\_dim)], bins=100, density=True, edgecolor='black')

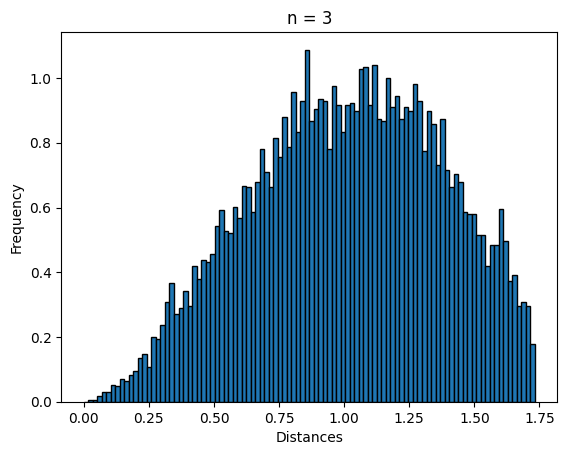
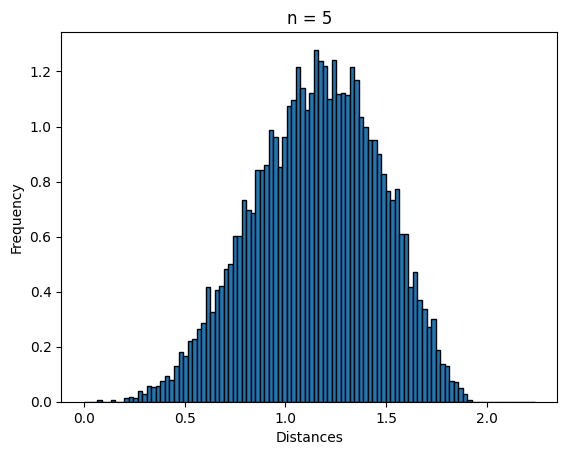
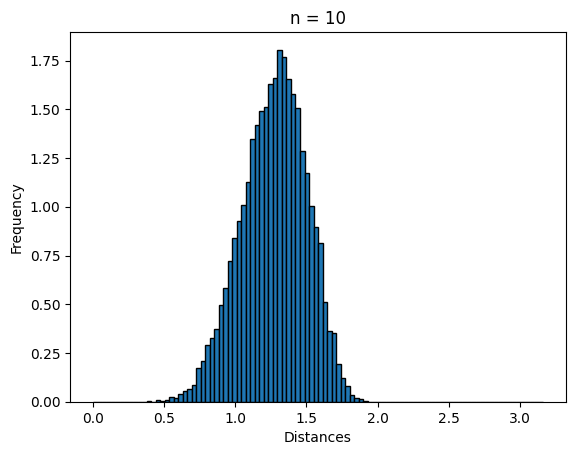
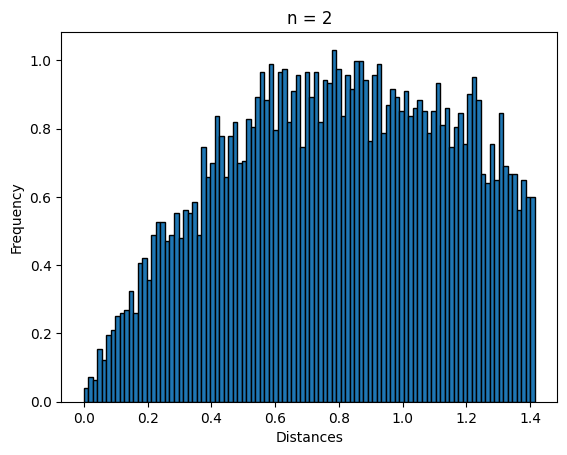
title = f'n = {d\_dim}'

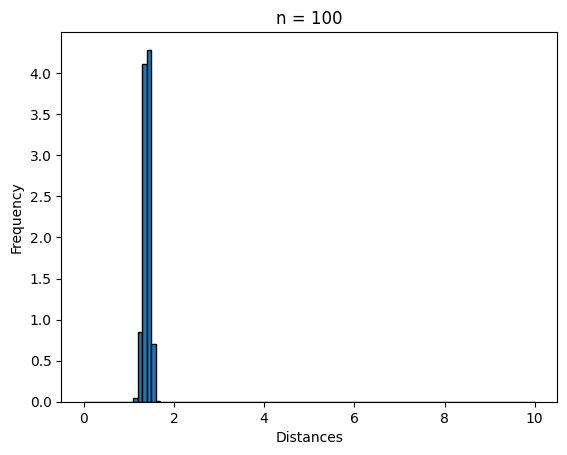
plt.gca().set(title=title, xlabel='Distances', ylabel='Frequency')

plt.show()

return dist

distances\_2d = freq\_plot(2)

* ******Part (D) – Results for Different Dimensions**

****

* **Part (E) – Conclusion**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dimension | Mean Distance | Standard Deviation | Max | Min |
| 2 | 0.9102 | 0.4249 | 1.9786 | 0.0109 |
| 3 | 1.0314 | 0.3764 | 1.9450 | 0.0485 |
| 5 | 1.1560 | 0.3115 | 1.9326 | 0.1782 |
| 10 | 1.2713 | 0.2259 | 1.8990 | 0.2912 |
| 100 | 1.3989 | 0.0704 | 1.6417 | 1.0988 |

**Key observations:**

1. **Increasing Mean Distance**: As dimension increases, mean distance between random points increases significantly
2. **Decreasing Relative Variation**: The concentration ratio (std/mean) decreases dramatically with dimension
3. **Distribution Shape Changes**:
   * Low dimensions (): Broader, more variable distributions
   * High dimensions (): Narrow, concentrated distributions around mean
   * Very high dimensions (): Extremely narrow distribution around mean

**Mathematical Explanation:**

In high-dimensional space, most volume is concentrated near the boundary of unit ball. Which means:

1. Most random points lie near the surface ()
2. Distances between points become more similar

The curse of dimensionality manifests as loss of meaningful distance-based similarity measures, fundamentally challenging distance-based algorithms.

## Problem 2 – Principal Component Analysis (PCA)

* **Part (A)**

Proof of is minimized when

**Step 1:** Expand the squared norm

**Step 2:** Replace *P*

Since :

**Step 3:**

To minimize , we need to maximize .

The minimum is achieved when *P* projects *x* onto the subspace that captures maximum variance. This occurs when columns of *W* are the first *k* principal components (eigenvectors) of the covariance matrix.

Therefore, is minimized when , where *W* contains the *k* PC.

* **Part (B) - PCA Implementation**

Given matrix:

**Step 1:** Centered Matrix

* Feature means: ,
* Centered matrix

X

**Step 2:** Covariance Matrix

**Step 3:** Eigen Decomposition

For 2x2 symmetric matrix the eigen values are:

Thus,

**Step 4:** Eigen Vector

**Step 5:** Centered data to 1D

Notes: Code implementation included in Jupyter Notebook.

## Problem 3 – Analytical Solution of Least Squares

* **Part (A) – Least Squares Solution**

Linear model:

Normal equation:

X = np.array([[0.05], [0.18], [0.31], [0.42], [0.50]])

y = np.array([0.12, 0.22, 0.35, 0.38, 0.49])

XTX = np.dot(X.T, X)

print("X^T X =", XTX)

XTX\_inv = np.linalg.inv(XTX)

print("(X^T X)^(-1) =", XTX\_inv)

XTy = np.dot(X.T, y)

print("X^T y =", XTy)

theta = np.dot(XTX\_inv, XTy)

print("θ =", theta)

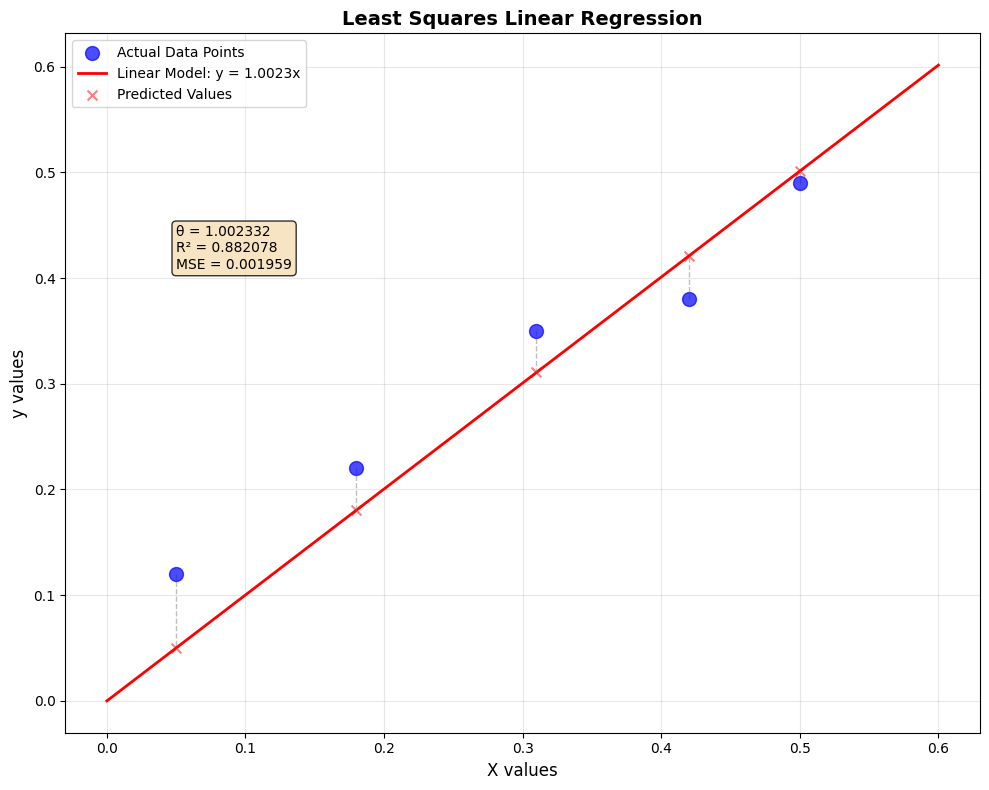
y\_pred = X.flatten() \* theta[0]

residuals = y - y\_pred

mse = np.mean(residuals\*\*2)

r\_squared = 1 - np.sum(residuals\*\*2) / np.sum((y - np.mean(y))\*\*2)

* *MSE*: 0.001959
* : 0.8821
* *RMSE*: 0.044265
* **Part (B) – Least Squares Solution**



**Key observations:**

* Strong linear relationship with
* Small residuals indicating good fit
* Slope ≈ 1.002 suggests almost 1:1 relationship

## Problem 4 – Multivariate Gradient Descent

* **Part (A) – Least Squares Solution**

We have one training sample with feature vector

Loss function:

Gradients:

**Code implementation:**

X = np.array([4, 9, 5])

theta = np.array([-0.8, 1.0, 0.3])

b = 0.5

y = 2

alpha = 0.001

def compute\_prediction(X, theta, b):

return np.dot(X.T, theta) + b

def compute\_loss(y\_pred, y):

return 0.5 \* (y\_pred - y)\*\*2

def compute\_gradients(X, y\_pred, y):

grad\_theta = (y\_pred - y) \* X

grad\_b = (y\_pred - y)

return grad\_theta, grad\_b

iterations\_data = []

current\_theta = theta

current\_b = b

for iteration in range(4):

y\_pred = compute\_prediction(X, current\_theta, current\_b)

loss = compute\_loss(y\_pred, y)

iteration\_data = {

'iteration': iteration,

'theta': current\_theta.copy(),

'b': current\_b,

'y\_pred': y\_pred,

'loss': loss

}

iterations\_data.append(iteration\_data)

if iteration < 3:

grad\_theta, grad\_b = compute\_gradients(X, y\_pred, y)

# Update parameters

current\_theta = current\_theta - alpha \* grad\_theta

current\_b = current\_b - alpha \* grad\_b

final\_data = iterations\_data[-1]

**Verification:** Loss decreasing

|  |  |  |
| --- | --- | --- |
| Iteration | Loss value | Change |
| 0 | 16.820 | - |
| 1 | 12.936 | -3.883 |
| 2 | 9.9500 | -2.986 |
| 3 | 7.652 | -2.297 |

Loss function consistently decreases, confirming correct implementation.

* **Part (B) – Learning Rate () Analysis**

A **small learning rate** is typically used to ensure **stable and precise convergence** in gradient descent.

**Why small learning rates are preferred:**

* **Stability:** Small steps prevent overshooting the minimum, avoiding oscillations or divergence.
* **Precision:** Allows smoother, more accurate updates near the optimal point.
* **Mathematical reasoning:** Gradient-based updates rely on small-step assumptions (e.g., Taylor expansion), which hold true only when the learning rate is small.

**Drawbacks of very small learning rates:**

* **Slow convergence:** Requires many iterations, increasing training time and computational cost.
* **Plateau issues:** May get stuck in flat regions or saddle points, giving the illusion of convergence.
* **Practical limits:** In real-time or resource-limited settings, extremely small steps are inefficient.

## Problem 5 – Weighted Ridge Regression

* **Part (A) – Diagonal Matrix W**

Find a diagonal matrix *W* such that:

Orginal Loss Function:

Step 1: Expand the desired form

Let be the residual vector:

Step 2: For Diagonal matrix )

Step 3: Compare with original:

Step 4: For equality for

**Final answer:**

* **Part (B) – Closed Form Solution**

Derive closed form solution for using

Step 1: Expand the Loss function

Since W is symmetric and using scalar properties:

Step 2: Compute Gradient

Using matrix calculus rules:

* , where
* , where is symmetric

Therefore:

Step 3: Set gradient to zero

Step 4: Solve for

* Assuming is invertible:

**Final Answer:**

Where )

Interpretation:

1. This is the weighted least squares estimator
2. When all , reduces to ordinary least squares:
3. Larger weights give more influence to the i-th observation
4. Matrix must be invertible for solution to exist

Comparison:

* Original Loss Function:
* Weighted Loss Function: