Machine Learning Homework 1 Solution

Junjia Zhang

January 2026

1 Simpson's Paradox

1.1 Winning Probability per Machine

$$\mathbb{P}_{\text{you},M1} = \frac{40}{40+60} = \frac{2}{5}$$
$$\mathbb{P}_{\text{friend},M1} = \frac{30}{30+70} = \frac{3}{10}$$
$$\mathbb{P}_{\text{you},M2} = \frac{210}{210+830} = \frac{21}{104}$$
$$\mathbb{P}_{\text{friend},M2} = \frac{14}{14+70} = \frac{1}{6}$$
$$\mathbb{P}_{\text{you},M1} > \mathbb{P}_{\text{you},M1}$$
$$\mathbb{P}_{\text{you},M2} > \mathbb{P}_{\text{you},M2}$$

You are more likely to win on both machines.

1.2 Overall Winning Probability

$$\mathbb{P}_{\text{you}} = \frac{40+210}{40+60+210+830} = \frac{25}{114}$$
$$\mathbb{P}_{\text{friend}} = \frac{30+14}{30+70+14+70} = \frac{11}{46}$$
$$\mathbb{P}_{\text{you}} < \mathbb{P}_{\text{friend}}$$

Your friend is more likely to win overall.

1.3 Mathematical explanation

$$\mathbb{P}_{\text{you}} = \frac{40+60}{40+60+210+830}\mathbb{P}_{\text{you},M1} + \frac{210+830}{40+60+210+830}\mathbb{P}_{\text{you},M2} = \frac{5}{57}\mathbb{P}_{\text{you},M1} + \frac{52}{57}\mathbb{P}_{\text{you},M2}$$
$$\mathbb{P}_{\text{friend}} = \frac{30+70}{30+70+14+70}\mathbb{P}_{\text{you},M1} + \frac{14+70}{30+70+14+70}\mathbb{P}_{\text{friend},M2} = \frac{25}{46}\mathbb{P}_{\text{friend},M1} + \frac{21}{46}\mathbb{P}_{\text{friend},M2}$$

Although both $\mathbb{P}_{\text{you},M1}$ and $\mathbb{P}_{\text{you},M2}$ are larger, $\mathbb{P}_{\text{you}}$ relies too much on the probability of winning on the second machine, which is too small compared to winning on the first.

2 Matrix as Operations

2.1

$a_1 \to b_1$: $\begin{bmatrix} 2 & a \\ 0 & b \end{bmatrix}$

$a_2 \to b_2$: $\begin{bmatrix} c & 3 \\ d & 2 \end{bmatrix}$

where a, b, c, d are arbitrary numbers.
By combining these two matrices together, we have the following:

$$\begin{bmatrix} 2 & 3 \\ 0 & 2 \end{bmatrix}$$

that satisfy the conditions.

2.2

According to the description:

$$V = \begin{bmatrix} \frac{\sqrt{5}}{5} & \frac{2\sqrt{5}}{5} \\ -\frac{2\sqrt{5}}{5} & \frac{\sqrt{5}}{5} \end{bmatrix}$$
$$\Sigma = \begin{bmatrix} 4 & 0 \\ 0 & 1 \end{bmatrix}$$
$$U = \begin{bmatrix} \frac{2\sqrt{5}}{5} & -\frac{\sqrt{5}}{5} \\ \frac{\sqrt{5}}{5} & \frac{2\sqrt{5}}{5} \end{bmatrix}$$

As a result, $U\Sigma V = U \begin{bmatrix} \frac{4\sqrt{5}}{5} & \frac{8\sqrt{5}}{5} \\ -\frac{2\sqrt{5}}{5} & \frac{\sqrt{5}}{5} \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 0 & 2 \end{bmatrix}$

The result matrix is also W.

2.3

$$W^T W = \begin{bmatrix} 4 & 6 \\ 6 & 13 \end{bmatrix}$$

To compute the eigenvector and the eigenvalue of $W^T W$

$$(W^T W - \lambda I)\vec{x} = \vec{0}$$

$$\begin{bmatrix} 4-\lambda & 6 \\ 6 & 13-\lambda \end{bmatrix} \vec{x} = \vec{0}$$

$$det(\begin{bmatrix} 4-\lambda & 6 \\ 6 & 13-\lambda \end{bmatrix}) = 0$$

$$\lambda^2 - 17\lambda + 16 = 0$$

$\implies \lambda = 16, \vec{x} = (t, 2t)$ or $\lambda = 1, \vec{x} = (-2t, t)$, where $t \in \mathbb{R}, t \neq 0$

After transformation W, the unit circle becomes an ellipse.

What I find is that the square of the semi-axis(length: 1 and 4) of the ellipse is equal to the eigenvalues that we get.

2.4

$$det(W) = 2 \times 2 - 0 \times 3 = 4$$

The area of the transformed shape $S' = \pi ab = \pi \times 1 \times 4 = 4\pi$
So, the relationship is $S' = det(W) \times S = 4\pi$

Because the overall effects of the matrices A, B and AB are the same, the scaling effects on the area of $det(AB)$ and $det(A)det(B)$ could also be seen as equal.

3 Some Practices

3.1

Because $var(X^3) \geq 0$, $var(X^3) = \mathbb{E}((X^3 - \mathbb{E}(X^3))^2) = \mathbb{E}((X^3)^2) - \mathbb{E}(X^3)^2 \geq 0$.
As a result, $\mathbb{E}((X^3)^2) \leq \mathbb{E}(X^3)^2$.

3.2

Apply Jensen's Inequality, when $f(x)$ is a convex function, $\mathbb{E}(f(X)) \geq f(\mathbb{E}(X))$.
Because $f(Y) = Y^2, f(Y)'' = 2 > 0$, it is a convex function, for $Y = X^3$, $\mathbb{E}((X^3)^2) \leq \mathbb{E}(X^3)^2$.

3.3

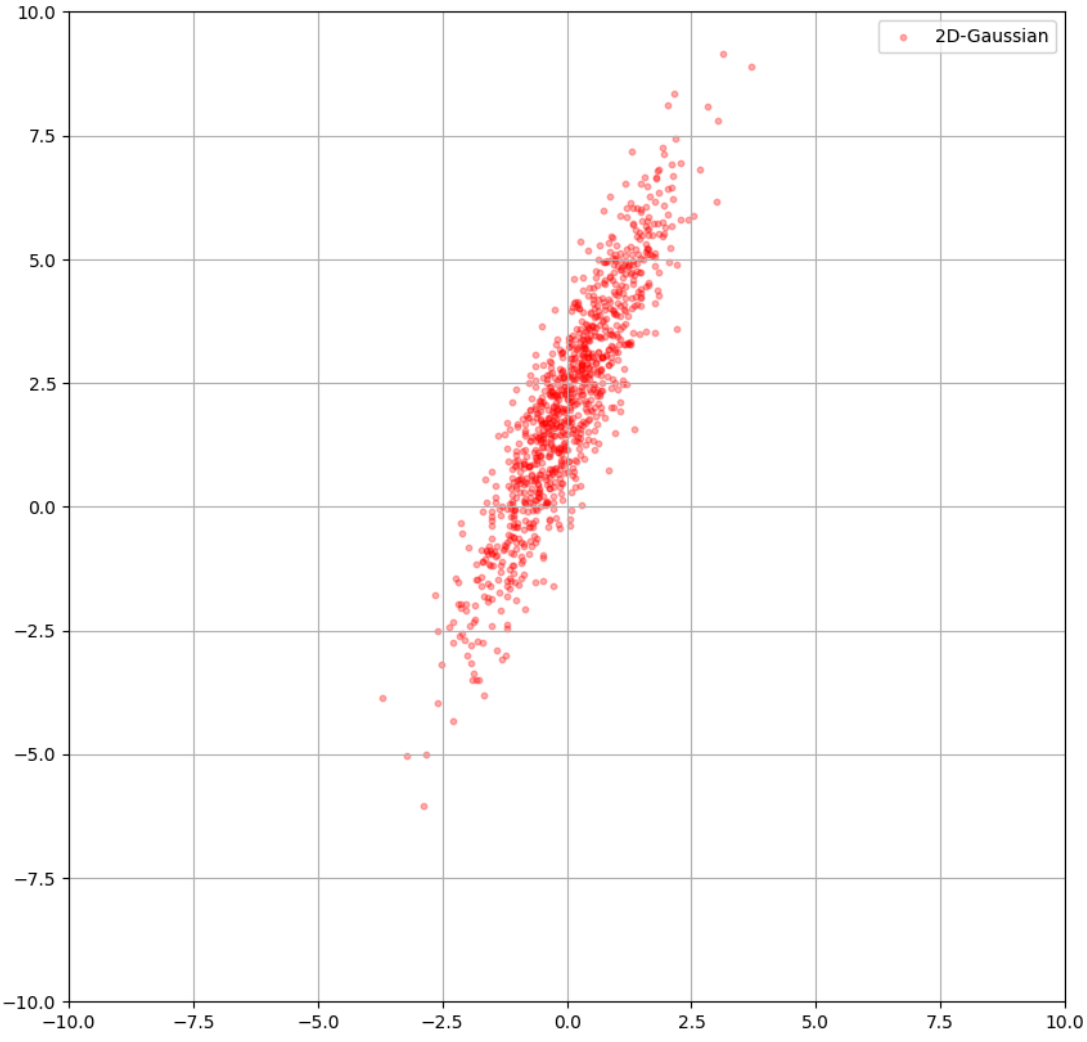
According to the definition of PSD matrices, for any nonzero vector z:

$$
\begin{aligned}
&z^T A z \geq 0, z^T B z \geq 0 \\
&\lambda z^T A z \geq 0, (1-\lambda) z^T B z \geq 0 \\
&z^T (\lambda A) z \geq 0, z^T [(1-\lambda) B] z \geq 0 \\
&z^T (\lambda A) z + z^T [(1-\lambda) B] z \geq 0 \\
&z^T [(\lambda A) + (1-\lambda) B] z \geq 0
\end{aligned}
$$

As a result, $\lambda A + (1-\lambda)B$ is also a PSD matrix.

4 Density Estimation of Multivariate Gaussian

In [9]:
```
import numpy as np
import matplotlib.pyplot as plt

# This is similar to the code from lecture 1
# to sample from a 2D Gaussian Distribution
mean = [0,2]
cov = [[1, 2], [2, 5]]
# Fix random seed to get consistent results
np.random.seed(1024)
X = np.random.multivariate_normal(mean, cov, 1000)
fig, ax = plt.subplots(figsize=(10, 10))
# c='r', dot color is red
# s=10.0, dot size is 10
# alpha=0.3, dot opacity is 0.3
ax.scatter(X[:,0], X[:,1], c='r', s=10.0, alpha=0.3, label="2D-Gaussian")
ax.grid()
ax.legend(loc = 0)
# Set x/y axis limits
ax.set_xlim([-10, 10])
ax.set_ylim([-10, 10])
plt.show()
```

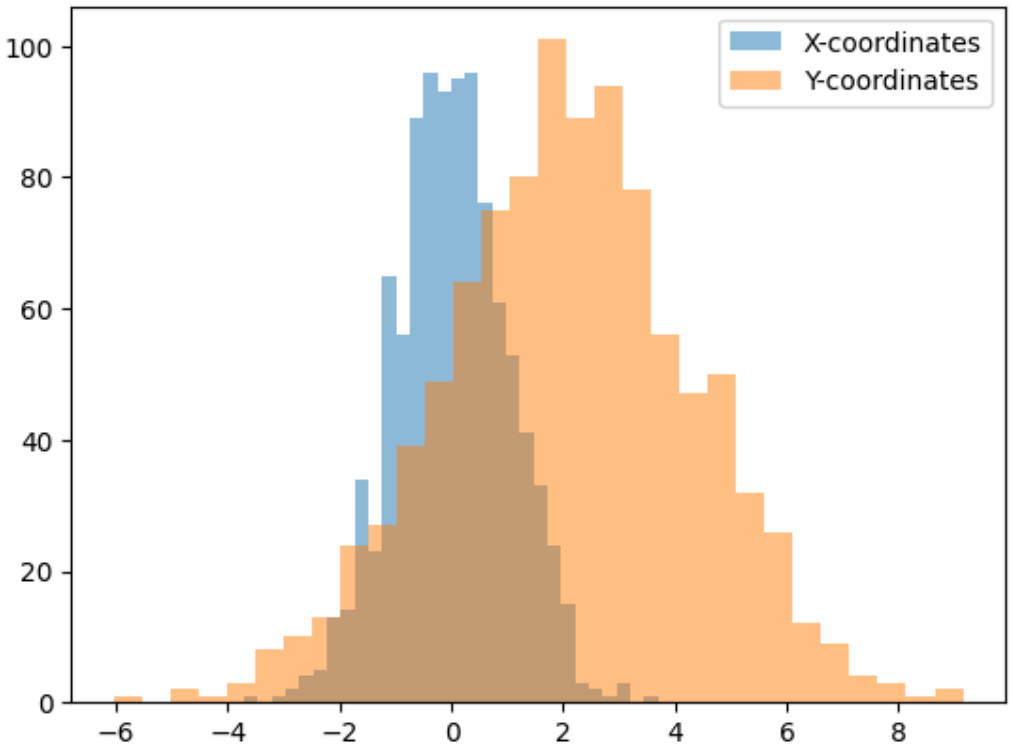


In [10]:
```
# TODO: Report the estimate mean and covariance
# of the sampled points
mean_est = np.mean(X, axis = 0)
cov_est = np.cov(X.T)
print("Estimated mean:", mean_est)
print("Estimated covariance:", cov_est)
```

```
Estimated mean: [0.01909265 2.06052385]
Estimated covariance: [[1.03589238 2.06244165]
 [2.06244165 5.08839176]]
```

In [11]:
```
# TODO: Plot the histogram for the x-coordinates of X
# and y-coordinates of X respectively.
# You can use the plt.hist() function
plt.hist(X[:,0], bins=30, alpha=0.5, label='X-coordinates')
plt.hist(X[:,1], bins=30, alpha=0.5, label='Y-coordinates')
plt.legend(loc='upper right')
```

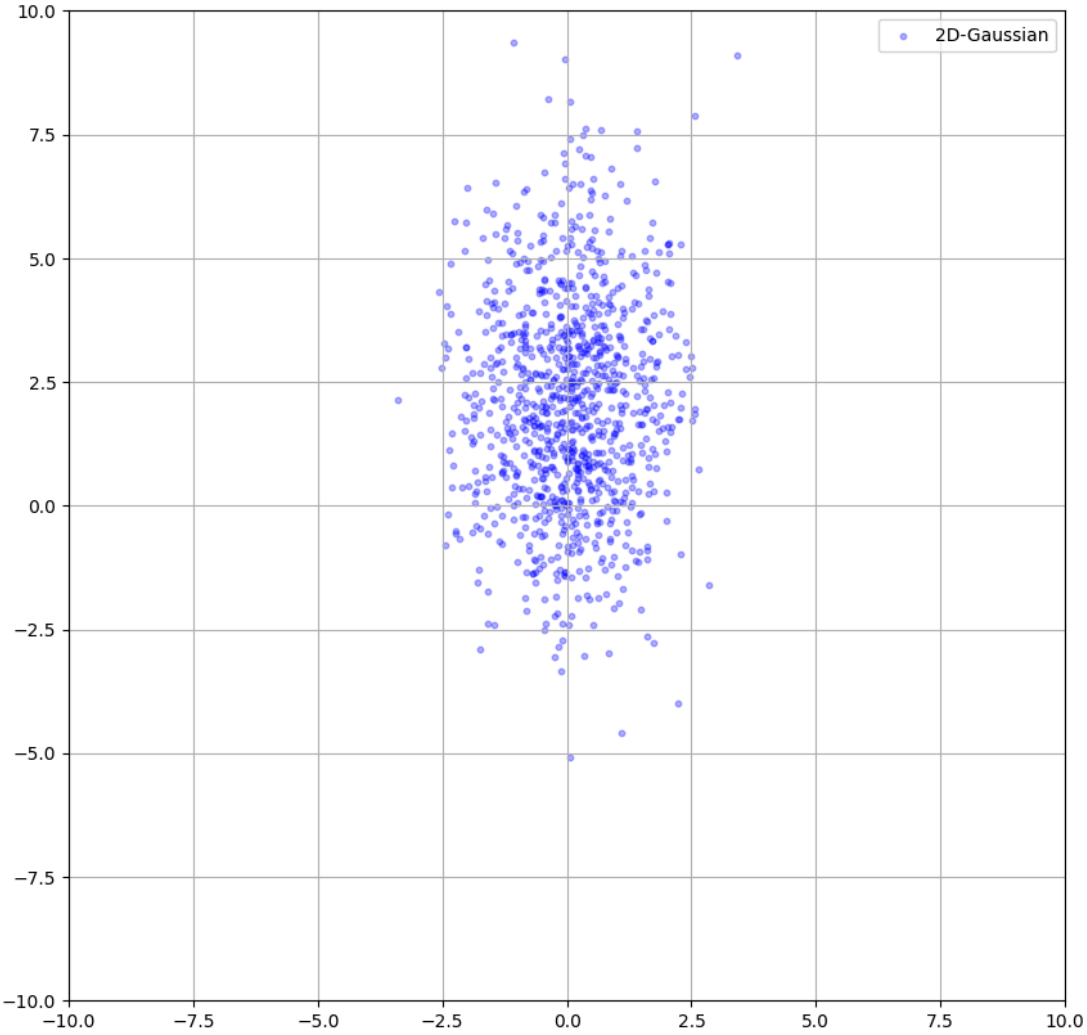
Out[11]:
```
<matplotlib.legend.Legend at 0x20803a29450>
```



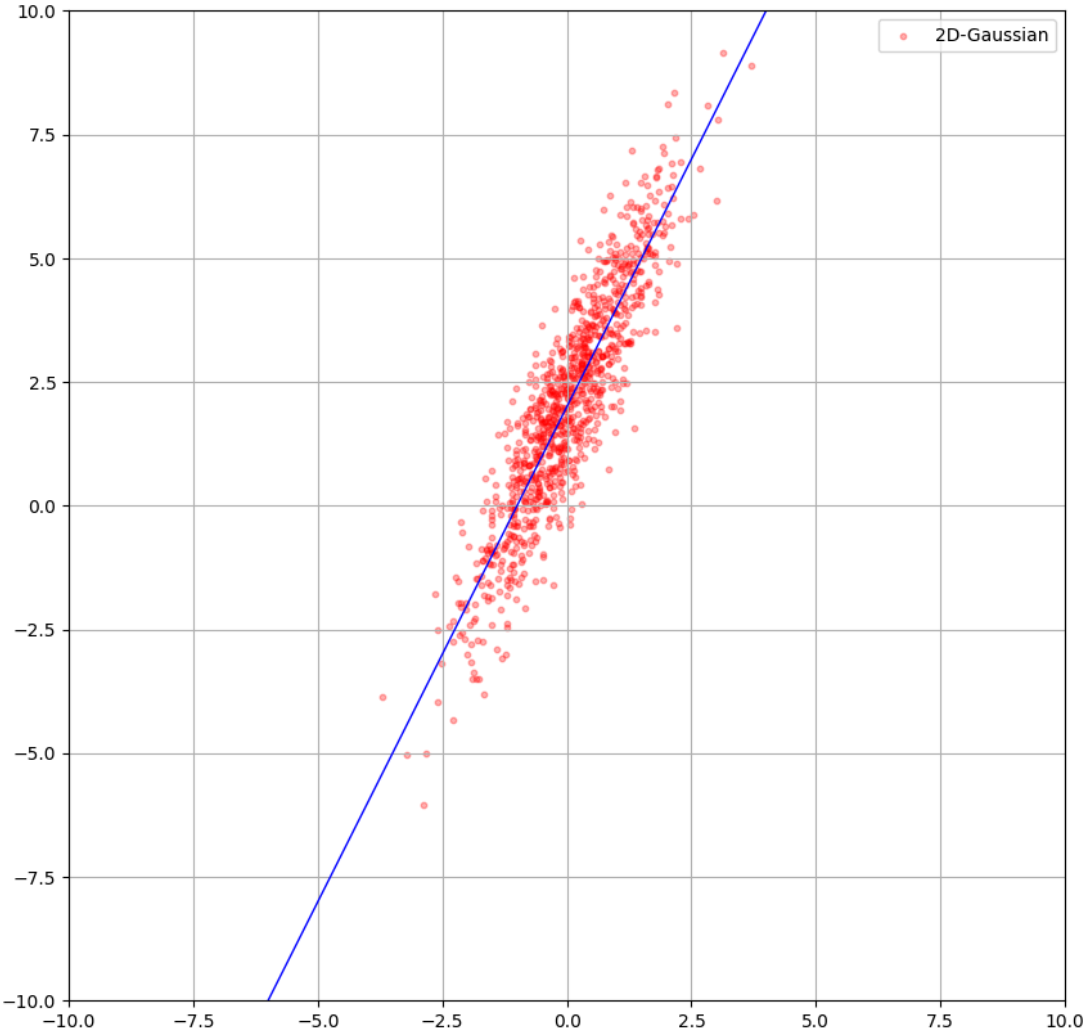
In [ ]:
```
# TODO: Are the x-coordinates of X samples from
# some Gaussian distribution?
# If so, estimate the mean and variance.
# Do the same for the y-coordinates.

# Answer:
# Yes, both x-coordinates and y-coordinates of X
# are samples from Gaussian distributions.
# The estimated mean and variance for x-coordinates:
xmean = np.mean(X[:,0])
xvar = np.var(X[:,0])
print("x-coordinates mean:", xmean)
print("x-coordinates variance:", xvar)
# The estimated mean and variance for y-coordinates:
ymean = np.mean(X[:,1])
```

```
yvar = np.var(X[:,1])
print("y-coordinates mean:", ymean)
print("y-coordinates variance:", yvar)


# TODO: Generate a new 2D scatter plot of 1000 points,
# such that the x-coordinates(y-coordinates) of all the points
# are samples from a 1D Gaussian distribution
# using the estimated mean and variance based on the x-coordinates(y-coordinates
Y = np.array([np.random.normal(xmean, xvar**0.5, 1000),
              np.random.normal(ymean, yvar**0.5, 1000)]).T
fig, ax = plt.subplots(figsize=(10, 10))
ax.scatter(Y[:,0], Y[:,1], c='b', s=10.0, alpha=0.3, label="2D-Gaussian")
ax.grid()
ax.legend(loc = 0)
ax.set_xlim([-10, 10])
ax.set_ylim([-10, 10])
plt.show()

#  Why the plot looks different from the first one?
#  The plot looks different because the original 2D Gaussian distribution has co
#  The new plot generated from independent 1D Gaussian distributions so the cova
#  resulting in a different scatter pattern.
```

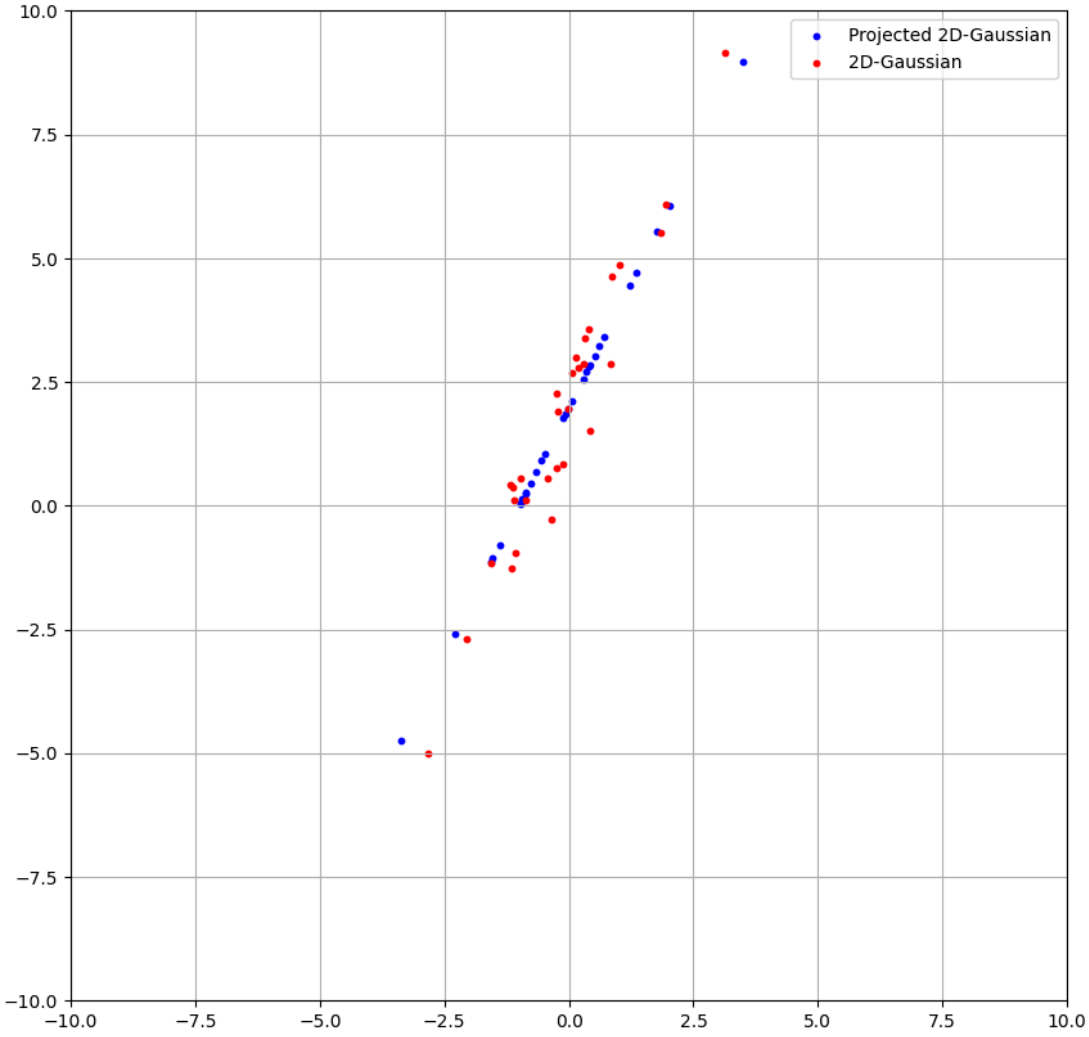
```
x-coordinates mean: 0.01909265145905166
x-coordinates variance: 1.0348564864741474
y-coordinates mean: 2.0605238541899022
y-coordinates variance: 5.083303371449156
```



In [13]:
```python
# Back to the original X
fig, ax = plt.subplots(figsize=(10, 10))
# c='r', dot color is red
# s=10.0, dot size is 10
# alpha=0.3, dot opacity is 0.3
ax.scatter(X[:,0], X[:,1], c='r', s=10.0, alpha=0.3, label="2D-Gaussian")
ax.grid()
ax.legend(loc = 0)
ax.set_xlim([-10, 10])
ax.set_ylim([-10, 10])

# TODO: Plot a line segment with x = [-10, 10]
# and y = 2x + 2 onto the 2D-Gaussian plot.
# The np.linspace() function may be helpful.
x = np.linspace(-10, 10, 1000)
y = 2 * x + 2
ax.plot(x, y, color='blue', linewidth=1)
plt.show()
```

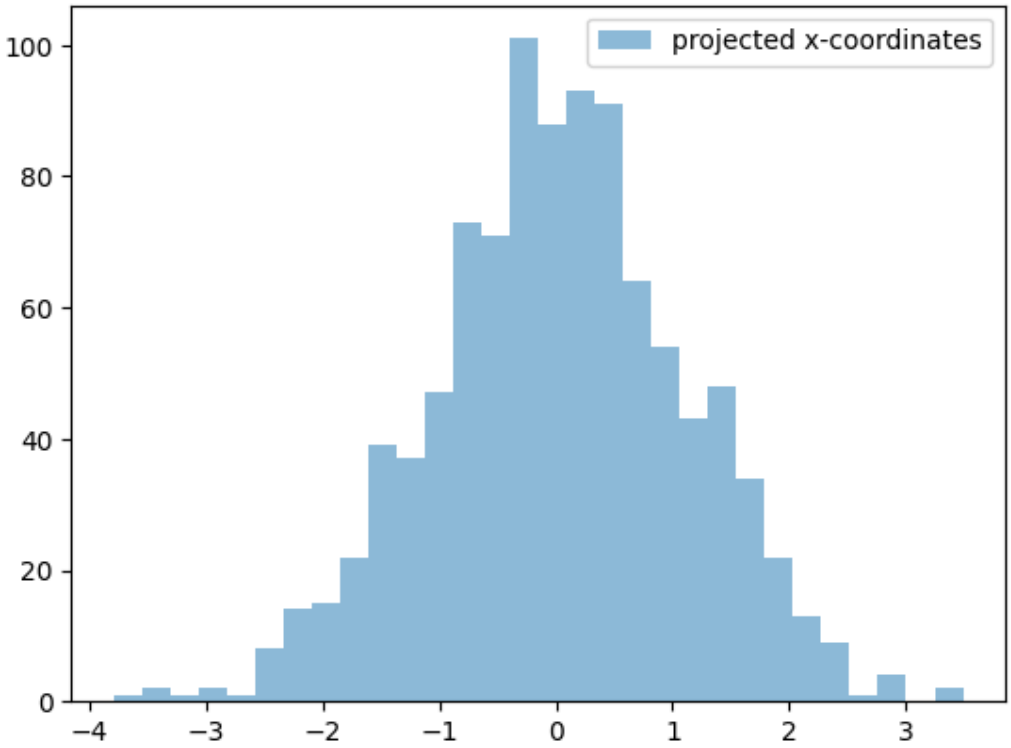


In [14]:
```python
# TODO: Project X onto line y = 2x + 2
# and plot the projected points on the 2D space.
# You need to remove this line and assign the projected points to X_proj.
X_proj = np.zeros((1000,2))
for i, (m, n) in enumerate(X):
    proj_x = (m + 2*n - 4) / 5
    proj_y = (2*m + 4*n + 2) / 5
    X_proj[i] = [proj_x, proj_y]

# Here we only plot 30 points to check the correctness
fig, ax = plt.subplots(figsize=(10, 10))
ax.scatter(X_proj[:30,0], X_proj[:30,1], c='b', s=10.0, alpha=1.0, label="Projec
ax.scatter(X[:30,0], X[:30,1], c='r', s=10.0, alpha=1.0, label="2D-Gaussian")
ax.grid()
ax.legend(loc = 0)
ax.set_xlim([-10, 10])
ax.set_ylim([-10, 10])
# You can also add the line from the previous plot for verification.
plt.show()
```



In [15]:
```
# TODO: Draw the histogram of the x-coordinates
# of the projected points.
# Are the x-coordinates of the projected points
# samples from some Gaussian distribution?
# If so, estimate the mean and variance.
plt.hist(X_proj[:,0], bins=30, alpha=0.5, label='projected x-coordinates')
plt.legend(loc='upper right')
# Answer:
# Yes, the x-coordinates of the projected points
# are samples from a Gaussian distribution.
# The estimated mean and variance:
proj_xmean = np.mean(X_proj[:,0])
proj_xvar = np.var(X_proj[:,0])
print("projected x-coordinates mean:", proj_xmean)
print("projected x-coordinates variance:", proj_xvar)
```

```
projected x-coordinates mean: 0.02802807196777122
projected x-coordinates variance: 1.1843834728271618
```



5 Programming Problem: Matrix/Tensor Transpose

In [16]:
```
import numpy as np

T = np.arange(144).reshape(2, 3, 4, 6)
```

In [17]:
```
def patchify(Q, Hp, Wp):
    B, C, H, W = Q.shape
    reshaped_Q = Q.reshape(B, C, int(H/Hp), Hp, int(W/Wp), Wp)
    reshaped_Q = np.transpose(reshaped_Q, (0, 1, 2, 4, 3, 5))
    reshaped_Q = reshaped_Q.reshape(B, C, int(H/Hp * (W/Wp)), Hp, Wp)
    reshaped_Q = np.transpose(reshaped_Q, (0, 2, 1, 3, 4))
    print(reshaped_Q.shape)
    return reshaped_Q
print("patchified result of the tensor is:", patchify(T, 2, 3))
```

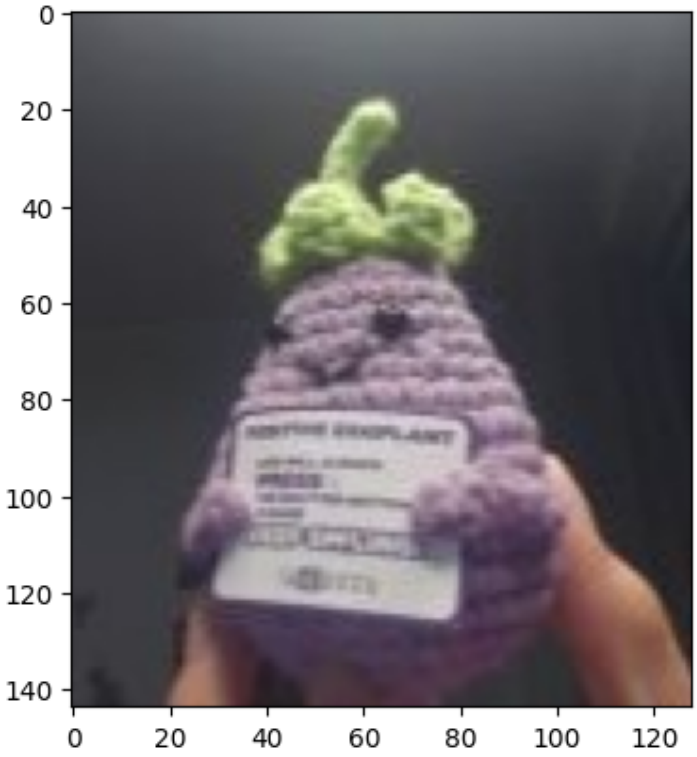
```
(2, 4, 3, 2, 3)
patchified result of the tensor is: [[[[[  0   1   2]
    [  6   7   8]]

   [[ 24  25  26]
    [ 30  31  32]]

   [[ 48  49  50]
    [ 54  55  56]]]


  [[[  3   4   5]
    [  9  10  11]]

   [[ 27  28  29]
    [ 33  34  35]]

   [[ 51  52  53]
    [ 57  58  59]]]


  [[[ 12  13  14]
    [ 18  19  20]]

   [[ 36  37  38]
    [ 42  43  44]]

   [[ 60  61  62]
    [ 66  67  68]]]


  [[[ 15  16  17]
    [ 21  22  23]]

   [[ 39  40  41]
    [ 45  46  47]]

   [[ 63  64  65]
    [ 69  70  71]]]]



 [[[[ 72  73  74]
    [ 78  79  80]]

   [[ 96  97  98]
    [102 103 104]]

   [[120 121 122]
    [126 127 128]]]


  [[[ 75  76  77]
    [ 81  82  83]]

   [[ 99 100 101]
    [105 106 107]]

   [[123 124 125]
    [129 130 131]]]
```

```
[[[ 84  85  86]
  [ 90  91  92]]

 [[108 109 110]
  [114 115 116]]

 [[132 133 134]
  [138 139 140]]]


[[[ 87  88  89]
  [ 93  94  95]]

 [[111 112 113]
  [117 118 119]]

 [[135 136 137]
  [141 142 143]]]]]
```

In [18]:
```
import numpy as np
import torch
from PIL import Image
import matplotlib.pyplot as plt

exaimg = np.array(Image.open('image.jpg'))

plt.imshow(exaimg)

exaimg = exaimg.transpose(2, 0, 1)

processed_img = torch.from_numpy(exaimg).unsqueeze(0)
patched_img = patchify(processed_img, 12, 16)
patched_img = patched_img.reshape(96, 3, 12, 16)
patched_img.dim()
```

```
torch.Size([1, 96, 3, 12, 16])
```

Out[18]: 4



In [19]:
```
import matplotlib.pyplot as plt
from torchvision.utils import make_grid

input_tensor = patched_img.squeeze().float().contiguous()
if input_tensor.max() > 1.0:
    input_tensor /= 255.0

grid_img = make_grid(input_tensor, nrow=8, padding=1, pad_value=1.0)

grid_np = grid_img.permute(1, 2, 0).cpu().numpy()

plt.figure(figsize=(25, 15))
plt.imshow(grid_np, interpolation='nearest')
plt.axis('off')
plt.show()
```

6 K-Means clustering

In [20]:
```
# Processing data
import pandas as pd
import numpy as np

load_data = np.array(pd.read_csv('clust_data.csv'))
```

In [121…
```
import numpy as np

class Kmeans:
    def __init__(self, max_iter, nstart, limit):
        self.max_iter = max_iter
        self.nstart = nstart
        self.limit = limit
        self.centroids = None
        self.labels = None
```

```
        self.inertia = float('inf')

    def run(self, X, k): # The controlling part, adopting 'fit' for multiple tim
        best_loss = float('inf')
        best_centroids = None
        best_labels = None

        for _ in range(self.nstart):
            self.fit(X, k)

            if self.inertia < best_loss: # Getting the best inertia among all th
                best_loss = self.inertia
                best_centroids = self.centroids.copy()
                best_labels = self.labels.copy()

        self.centroids = best_centroids
        self.labels = best_labels
        self.inertia = best_loss

        print(f"Best Loss: {self.inertia:.4f}")

    def fit(self, X, k): # Applying the iteration to try to minimize inertia.
        indices = np.random.choice(len(X), k, replace=False)
        self.centroids = X[indices]

        for _ in range(self.max_iter):
            self.labels = self._assign_clusters(X)
            new_centroids = self._update_centroids(X, k)
            if np.all(np.abs(self.centroids - new_centroids) < self.limit):
                break
            self.centroids = new_centroids

        selected_centroids = self.centroids[self.labels]
        self.inertia = np.sum((X - selected_centroids) ** 2)

    def _assign_clusters(self, X): # Setting indices that aligning data points t
        distances = np.linalg.norm(X[:, np.newaxis, :] - self.centroids, axis=2)
        return np.argmin(distances, axis=1)

    def _update_centroids(self, X, k): # Updating centroids to better ones.
        new_centroids = np.zeros((k, X.shape[1]))
        for i in range(k):
            points_in_cluster = X[self.labels == i]
            if len(points_in_cluster) > 0:
                new_centroids[i] = points_in_cluster.mean(axis=0)
            else:
                new_centroids[i] = self.centroids[i]
        return new_centroids

    def get_clusters_sizes(self): # Getting the sizes of each cluster.
        sizes = []
        for i in range(len(self.centroids)):
            sizes.append(int(np.sum(self.labels == i)))
        return sizes
```

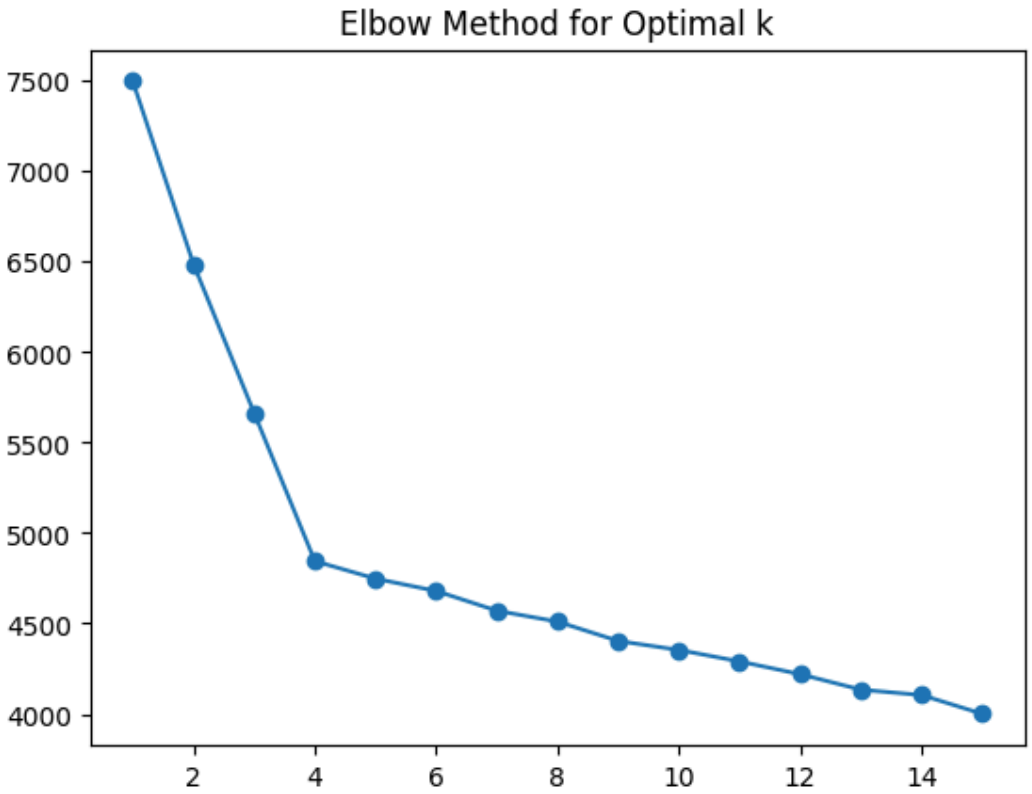
In [130…

```
# Find the 'elbow'.
from matplotlib import pyplot as plt
model = Kmeans(max_iter=80, nstart=10, limit=1e-4)
inertias = []
for k in range(1, 16):
```

```
    model.run(load_data, k)
    inertias.append(model.inertia)

plt.plot(range(1, 16), inertias, marker='o')
plt.title('Elbow Method for Optimal k')
```

```
Best Loss: 7497.2370
Best Loss: 6487.1965
Best Loss: 5663.6576
Best Loss: 4844.9258
Best Loss: 4747.9769
Best Loss: 4679.8718
Best Loss: 4570.6867
Best Loss: 4510.5054
Best Loss: 4403.5025
Best Loss: 4353.8619
Best Loss: 4290.4149
Best Loss: 4219.6531
Best Loss: 4134.2927
Best Loss: 4104.6065
Best Loss: 4000.6219
```

Out[130… Text(0.5, 1.0, 'Elbow Method for Optimal k')



It is obvious that 4 is the elbow point and the best k to adopt.

In [131…
```
# Running K-means with k=4 and getting cluster sizes.
model.run(load_data, 4)
model.get_clusters_sizes()
```

```
Best Loss: 4844.9258
```

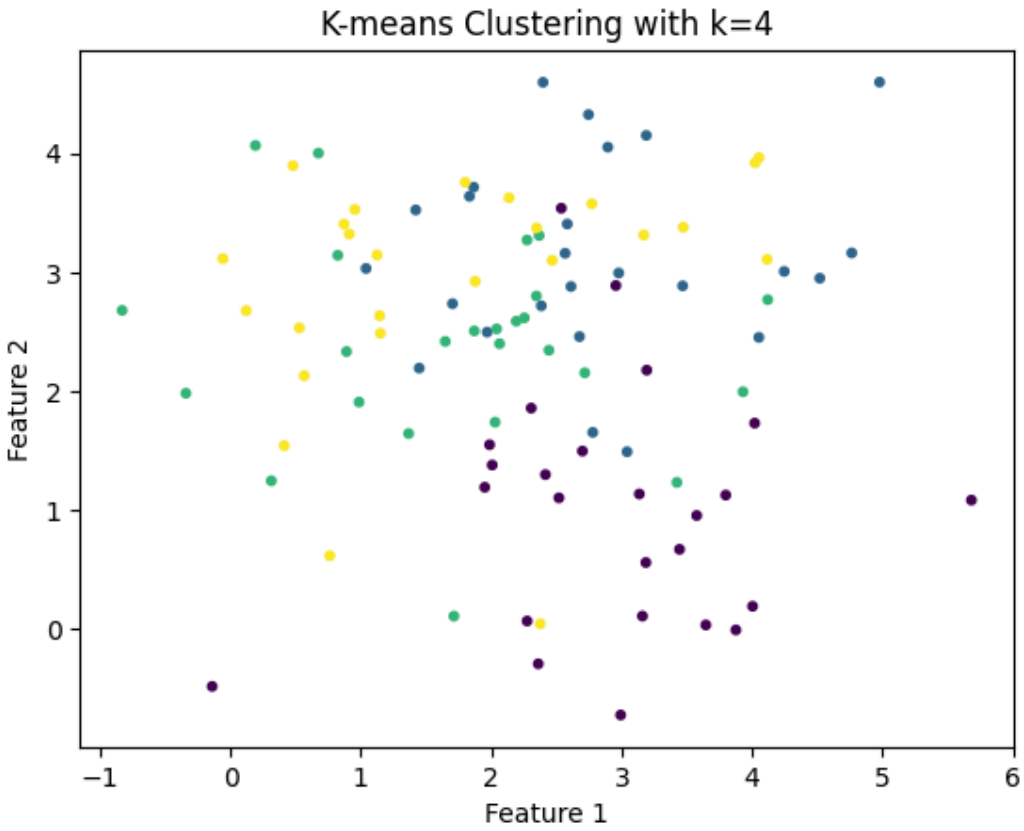
Out[131… [25, 25, 25, 25]

In [132…

```
# Plotting the clustered data with k=4.
plt.scatter(load_data[:, 0], load_data[:, 1], c=model.labels, cmap='viridis', s=
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('K-means Clustering with k=4')
```

Out[132…

```
Text(0.5, 1.0, 'K-means Clustering with k=4')
```



The clustering result seems not very precise, because there are too many features, only 2 features cannot reflect the whole situation. Maybe there should be more k centroids.