EDA Lab 1

André Plancha
andre.plancha@hotmail.com
Yana Zlatanova
yana-zlatanova-gold@gmail.com

MALMÖ UNIVERSITY
MA661E - VT25
April 20, 2025

1) Generate n =150, p = 6 normally distributed random variables that have high variance in some dimension and low variance in another dimension. Try PCA using both correlation and covariance matrices.

Following the example given, a dataset with 2 dimensions of high variance and a dataset with 4 dimensions of low variance were created and combined into one dataset with $n = 150, p = 6$.

a) Is the covariance matrix very informative?

The covariance matrix PCA is not very informative in our example because it reflects only the variance of the the first 2 dimensions with high-variance and it fails to show the structures in the remaining dimensions.

b) Which one would be better to use in this case?

In our case correlation matrix would be a better choice because it standardizes all variables to have equal variance before finding principal components.

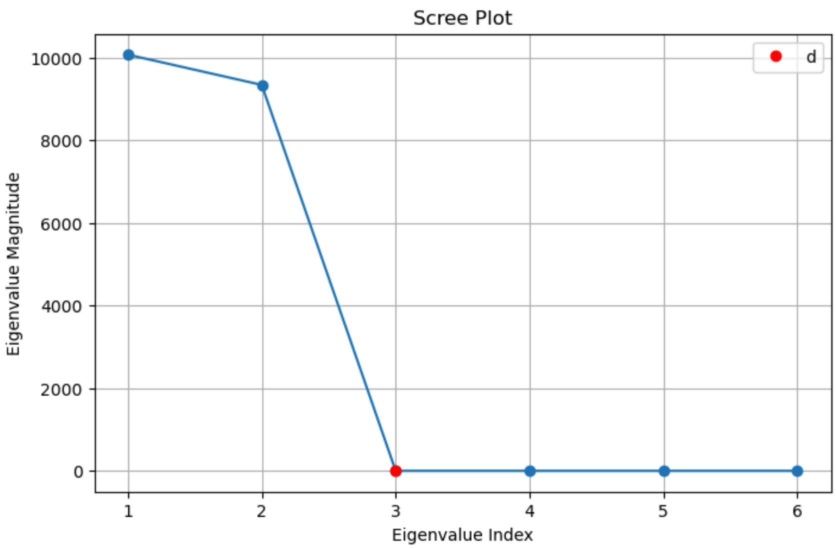

Figure 1: PCA using covariance matrix

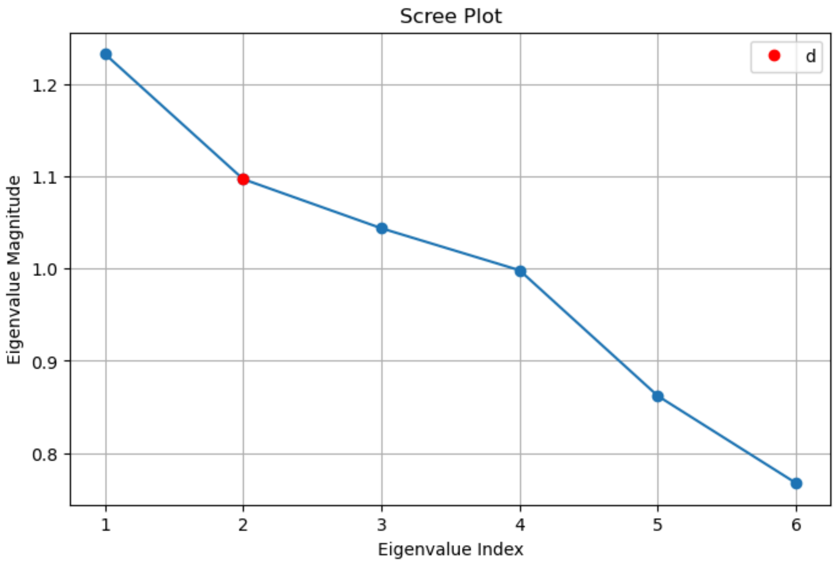

Figure 2: PCA using correlation matrix

2) For each case, apply Linear Discriminant Analysis (LDA) to the `Iris` dataset and visualize the data in one dimension, using the Kernel Density Estimate; and discuss how good the mapping are for each case.

case a): Apply LDA for only 2 classes at a time, i.e., [setosa, versicolor]; [versicolor, virginica]; [virginica, setosa].

The `Iris` dataset is composed of 150 samples of iris flowers with 4 numerical features that describe sepal length, sepal width, petal length and petal width. Each flower can be a setosa, a versicolor, or a virginica (target).

For this case, for each pair, we filtered out the other class and applied `sklearn.discriminant_analysis.LinearDiscriminantAnalysis(n_components=1)` to the remaining rows, using the class as the target. To calculate and plot Kernel Density Estimate, `plotnine.geom_density` was used, which by default uses a *Gaussian* kernel. The code for this process can be observed in Listing 3.

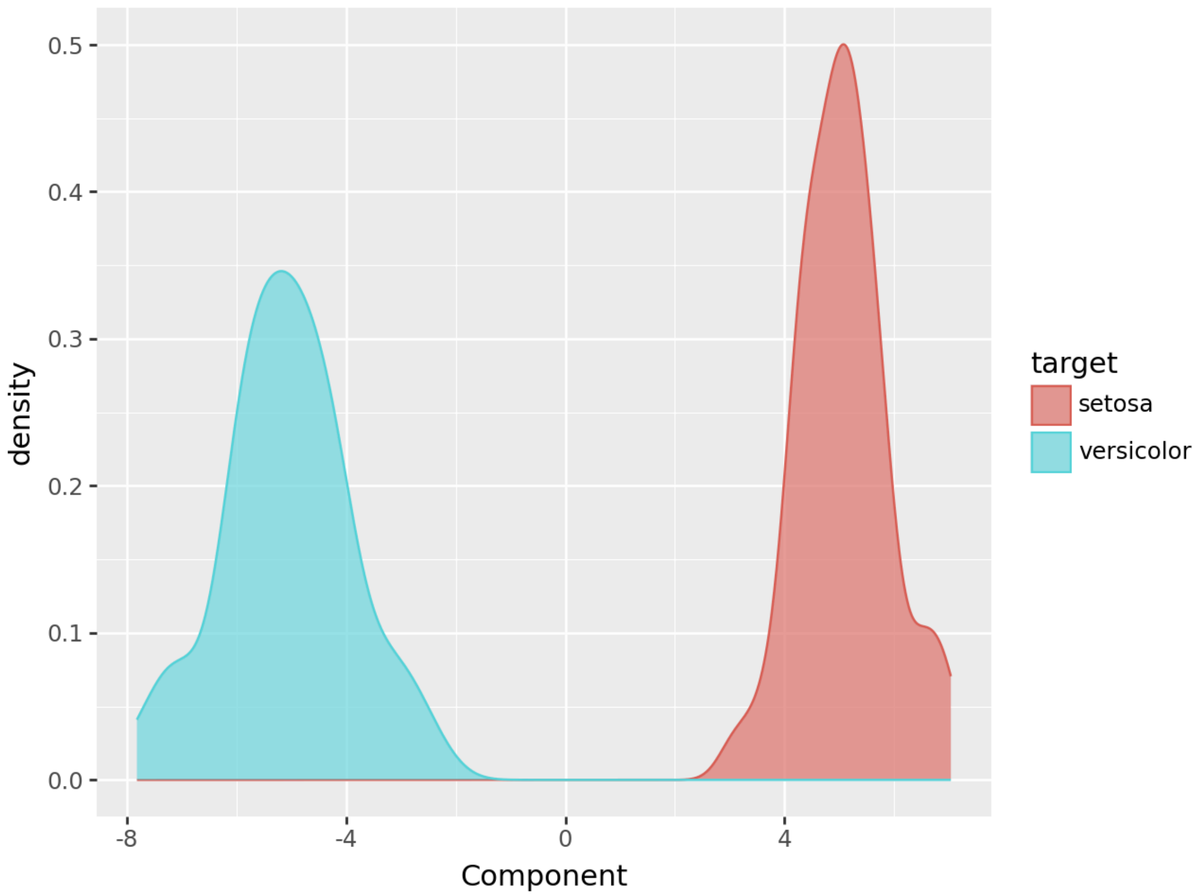

Figure 3: LDA without virginica

As observed in Figure 3, since there's no overlap between the mappings, the LDA transformation from the *setosa* and *versicolor* pair effectively found a direction where the two classes are linearly separable.

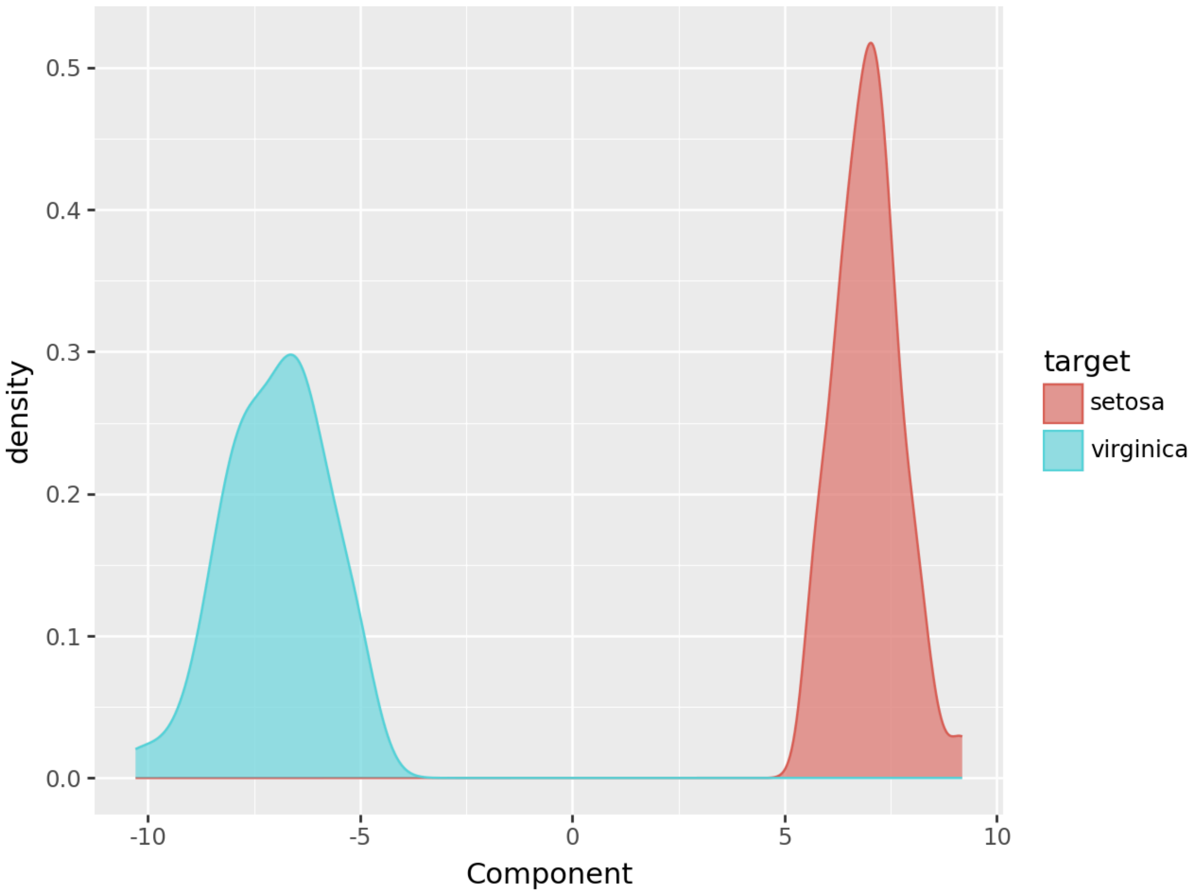

Figure 4: LDA without versicolor

Similar to the previous pair, the *setosa* and *virginica* pair also shows no overlap between the mappings, so the LDA transformation effectively found a direction where the two classes are linearly separable as well.

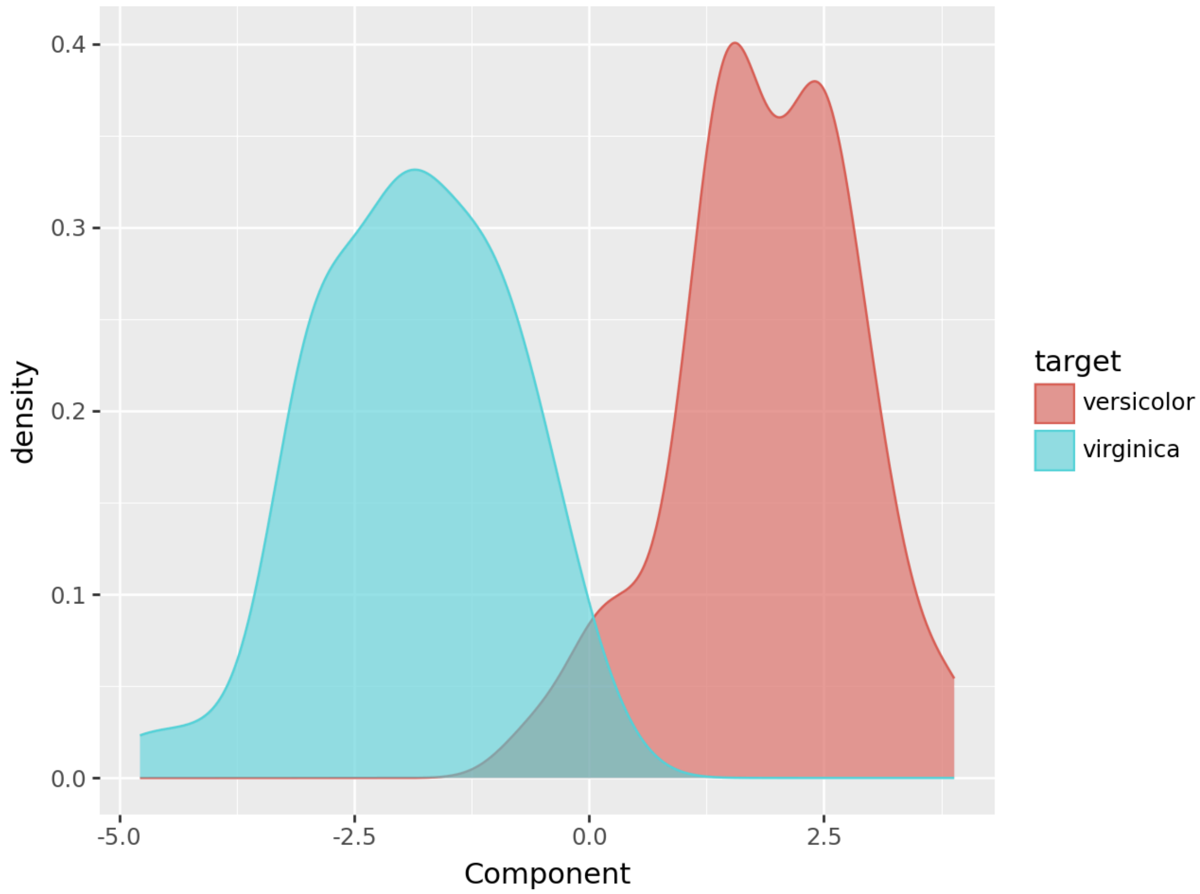

Figure 5: LDA without setosa

However, since there's some overlap in the transformation from the *versicolor* and *virginica* pair, as observed in Figure 5 we can conclude that the LDA couldn't find a direction as effective as the other ones; with that being said, the overlap is quite small, so it can be argued that it's still a good mapping.

case b): Instead of classifying based on species, consider two broad classes—Sepals and Petals. Apply LDA to distinguish between sepal-based features ([sepal length, sepal width]) and petal-based features ([petal length, petal width]).

For this case, we decided on concatenating the sepal width/length and petal width/length to make a `Width/Lenght` column, and make the target `Sepal_or_Petal` describe if the row was produced from the Sepal or Petal part. A sample of this transformed dataset can be seen on Figure 6.

	# Width	# Length	Sepal_Or_Petal
146	2.5	6.3	Sepal
147	3.0	6.5	Sepal
148	3.4	6.2	Sepal
149	3.0	5.9	Sepal
150	0.2	1.4	Petal
151	0.2	1.4	Petal
152	0.2	1.3	Petal
153	0.2	1.5	Petal
154	0.2	1.4	Petal
155	0.4	1.7	Petal
156	0.3	1.4	Petal
157	0.2	1.5	Petal

Figure 6: Sepal or Petal dataset sample

After this, a similar process from **2) a)** was done to for the LDA transformation and Kernel Density Estimation. The code for this one can be observed in Listing 4.

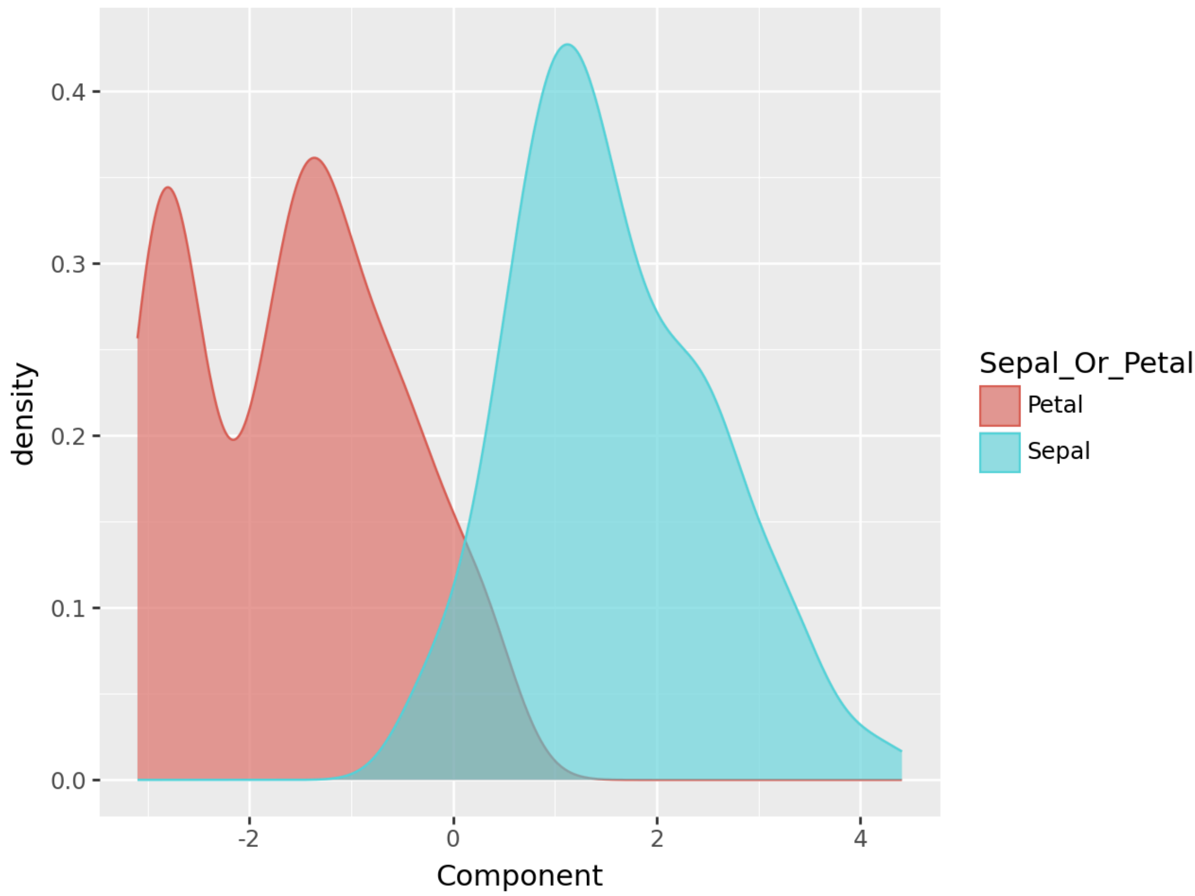

Figure 7: LDA of Petals and Sepals

As observed in Figure 7, there's only a small overlap between the mappings, however which suggests that the direction found by the LDA is somewhat effective.

3) Factory analysis

a) Repeat example 2.5 for dataset stockreturns.

For this analysis, we worked with the stockreturns dataset containing 10 columns (companies) and 100 rows (trading days, assumed). We first standardized the data and then applied factor analysis to extract three factors. We implemented the analysis both with varimax rotation and without rotation for comparison.

The 2D factor loading plots below visualize how each company relates to the first two factors, revealing which companies tend to move together and which respond differently to the factors.

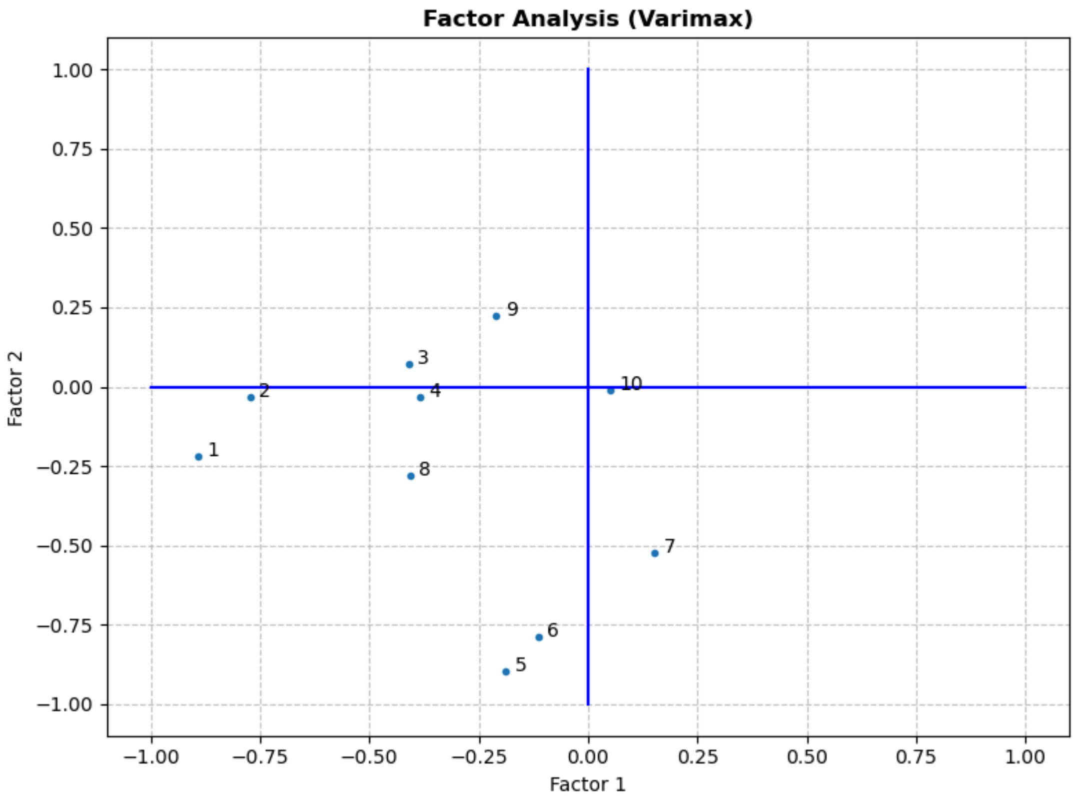

Figure 8: Factor loading plot with varimax rotation

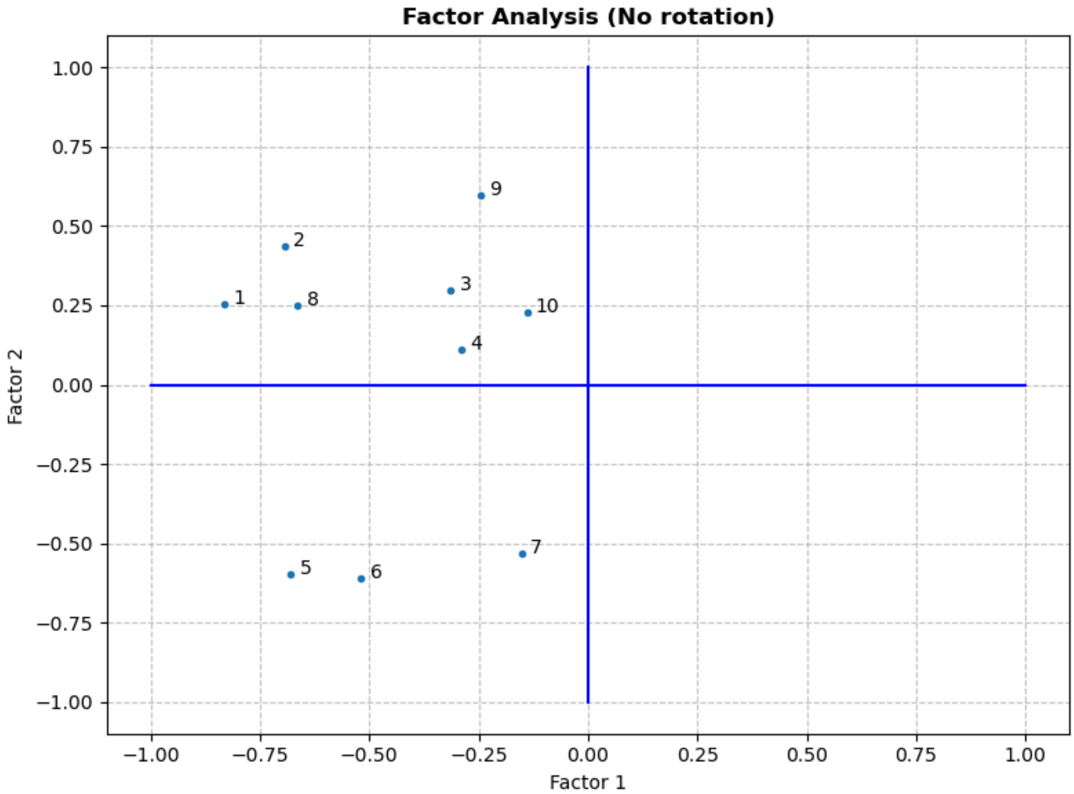

Figure 9: Factor loading plot with no rotation

If we compare the two figures, it's evident that rotation creates a more differentiated structure. The un-rotated solution primarily separates companies along Factor 2, while the rotated solution identifies more distinct groupings that better reflect underlying relationships.

The figure below shows a few plots that display the factor scores from the factor analysis with rotation, where each point represents a single trading day positioned according to its scores on each factor. These plots reveal varied market behavior across trading days with no distinct clustering patterns.

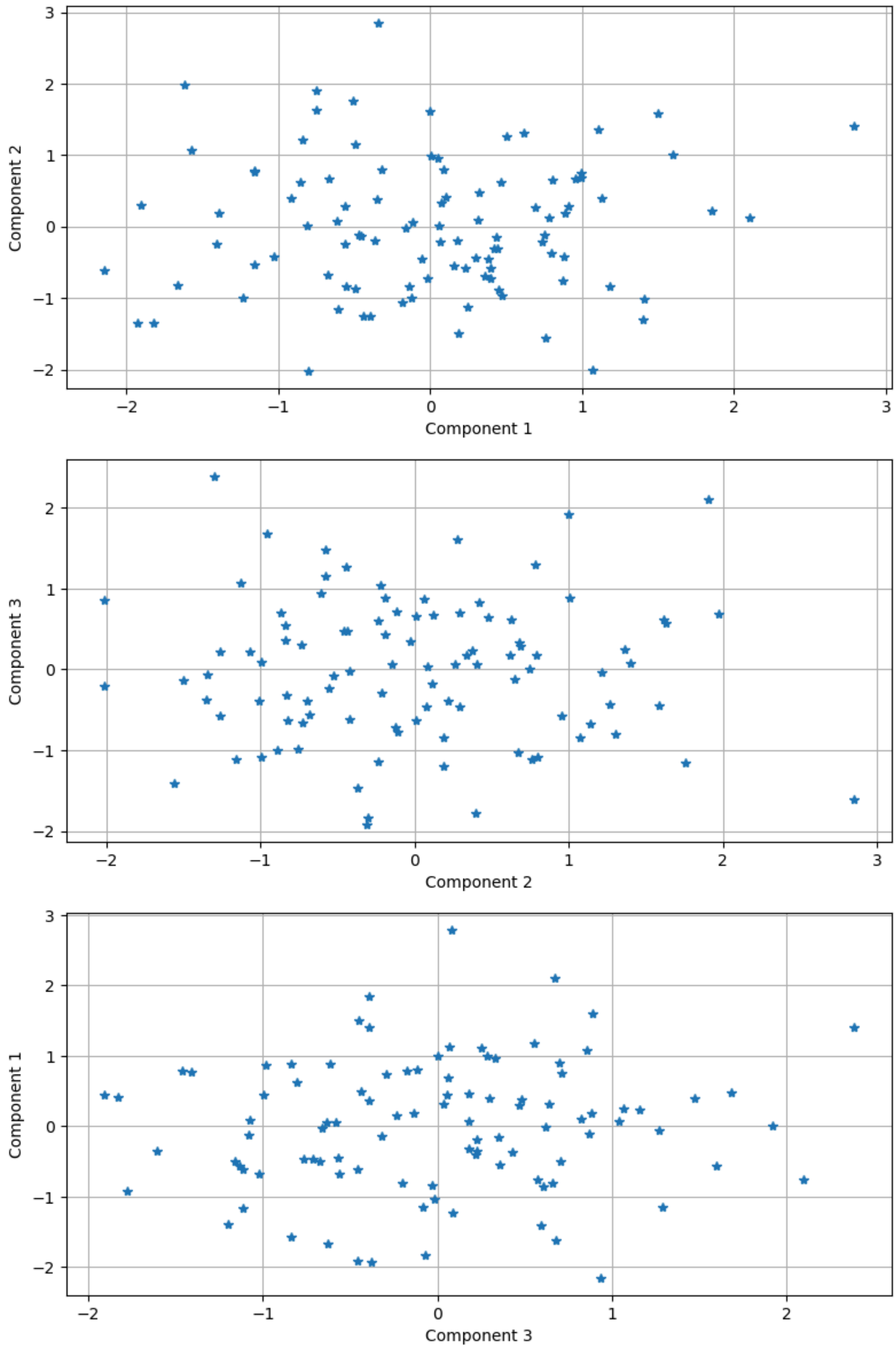

Figure 10: Factor scores from the factor analysis with rotation

b) Carry out a factor analysis for your data for 10 companies over 20 days.

Based on Figure 11, SEB and Nordea bank are positioned close together, indicating they have similar factor loadings and patterns of correlation with the underlying factors.

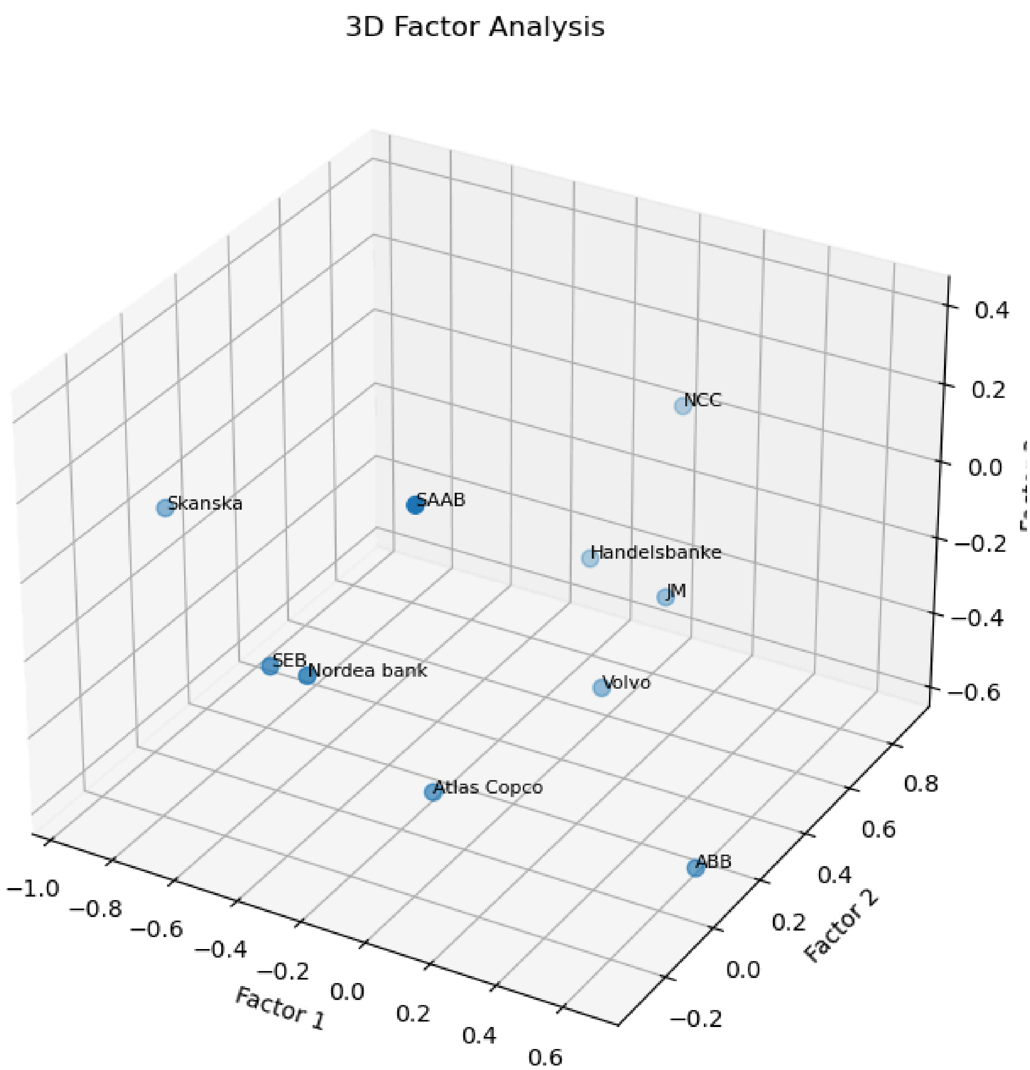

Figure 11: Factor loading plot with rotation

Figure 12 illustrates a broad distribution of points across all three plots, confirming that each component captures variation in the data. However, the absence of clear linear patterns between any pair of components suggests that the components are relatively independent.

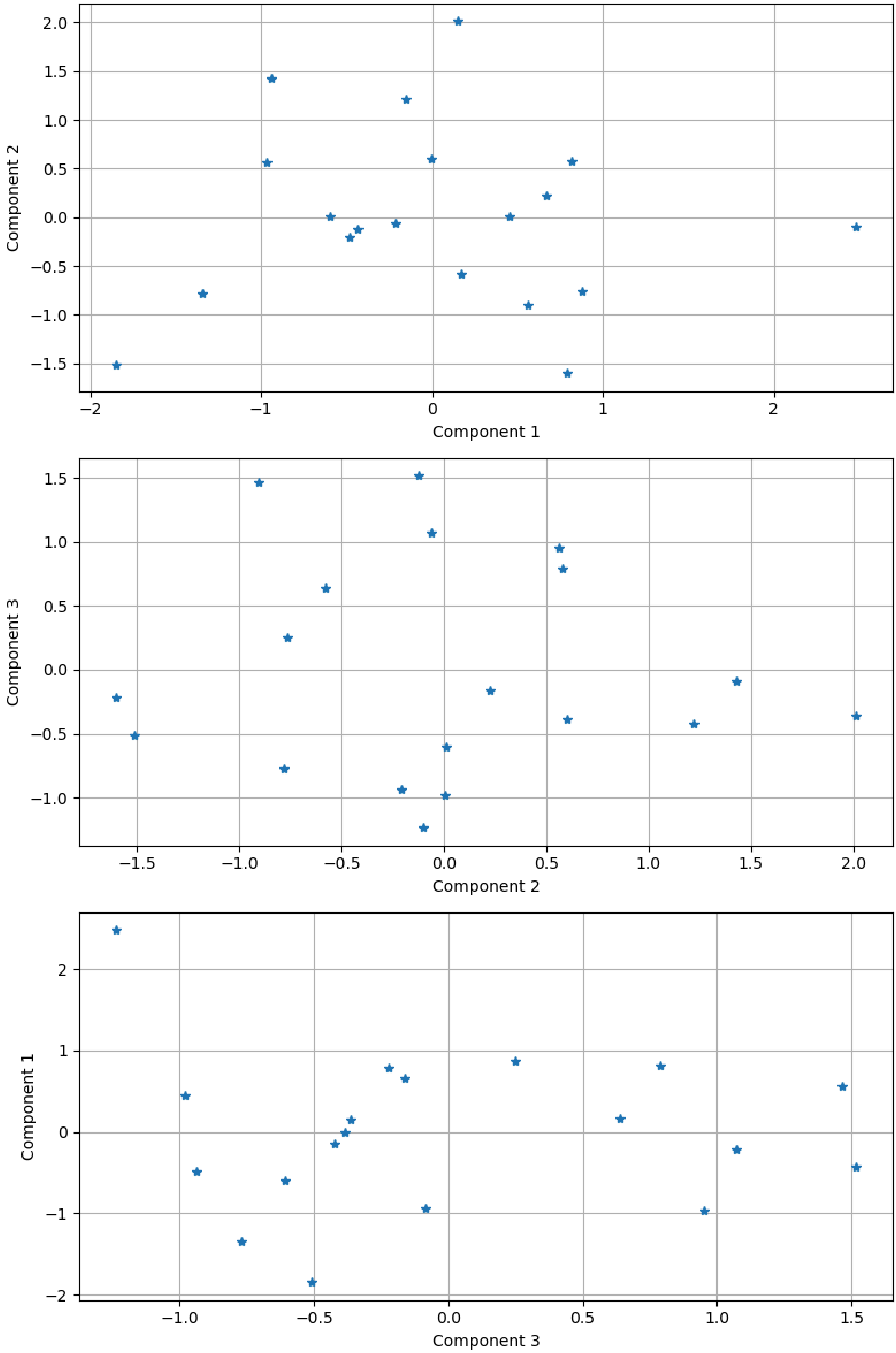

Figure 12: Factor scores from the factor analysis with rotation

4) Using the following curve:

$x_1 = \frac{t \cos t}{1+t^2}, x_2 = \frac{t \sin t}{1+t^2}, x_3 = t, -2\pi \le t \le 2\pi,$

a) Estimate the intrinsic dimensionality using the Pettis, Bailey, Jain, and Dubes algorithm (available in `idpettis.m`)

The curve was generated using the code listed in Listing 9 and the curve can be visualized in Figure 13.

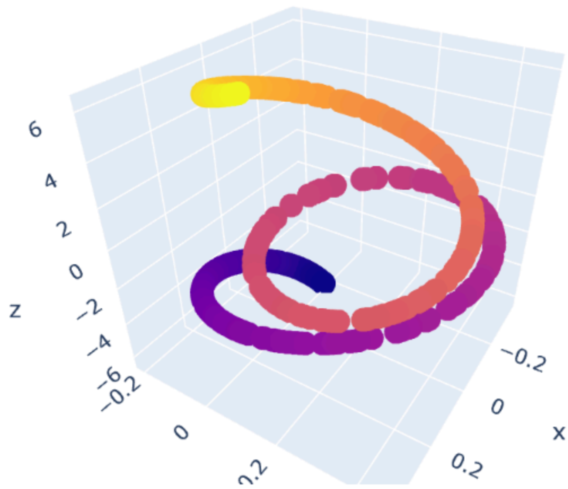

Figure 13: Curve generated

To calculate the intrinsic dimensionality estimate, `pyEDAkit.IntrinsicDimensionality.id_pettis` was used, resulting in

$$\text{IDE} \approx 1.119$$

b) Study the intrinsic dimensionality when introducing noise of various sizes to the curve.

For this exercise, to introduce noise of various sizes, we decided to, while generating the points from the curve, add a random noise to the point (x, y, z) generated each iteration, so the new noisy point would be equal to

$$(x + u_1 * \sigma, y + u_2 * \sigma, z + u_3 * \sigma), u_1, u_2, u_3 \sim U(0, 1)$$

Figure 14: Curves with noise added

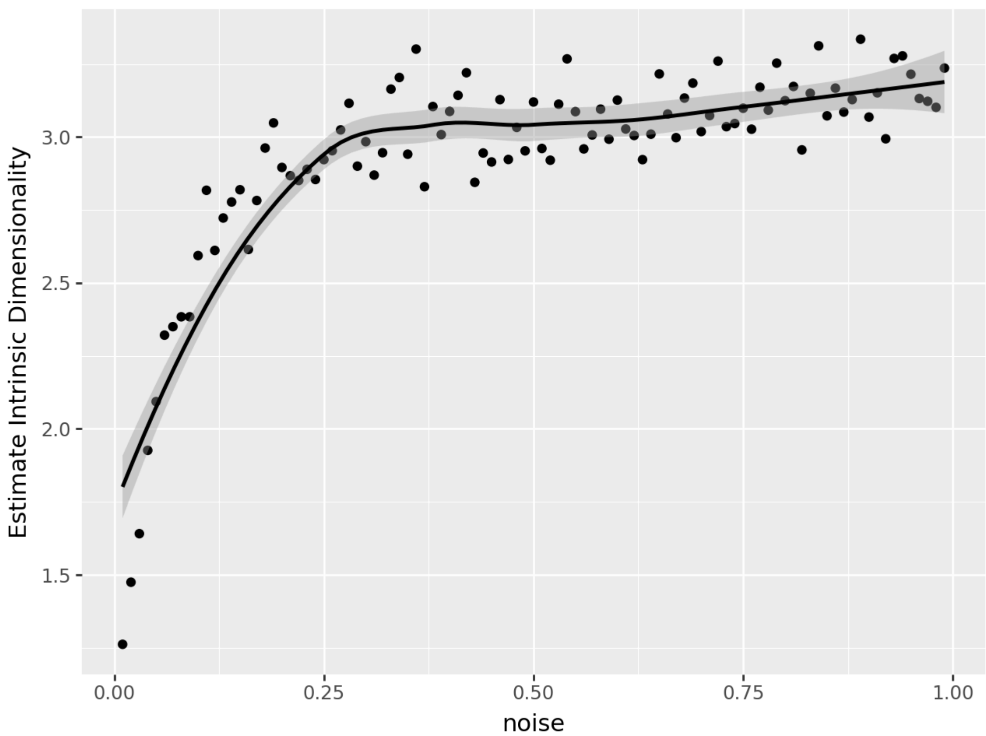

Figure 15: Intrinsic Dimensionality over noise size

We can see from Figure 14 that, as expected, a bigger level of noise will make the curve less recognizable, to the point that calling it a curve is even questionable. From Figure 15, we can see that the intrinsic dimensionality estimate goes up as noise increases, exponentially increasing until around IED $= 3$, which makes sense since adding a noise factor can be even argued as adding another intrinsic dimension to our curve. The IED increase seems to stabilize around this estimate, around a noise of $\sigma \approx 0.25$.

c) Is there any threshold number of noise size for the intrinsic dimensionality estimate?

From Figure 15, it seems that the the noise size seem to not affect the IDE as much around $\sigma = 0.25$, seemingly being exponentially increasing before that. To investigate the existence of such threshold, even smaller numbers of noise were calculated.

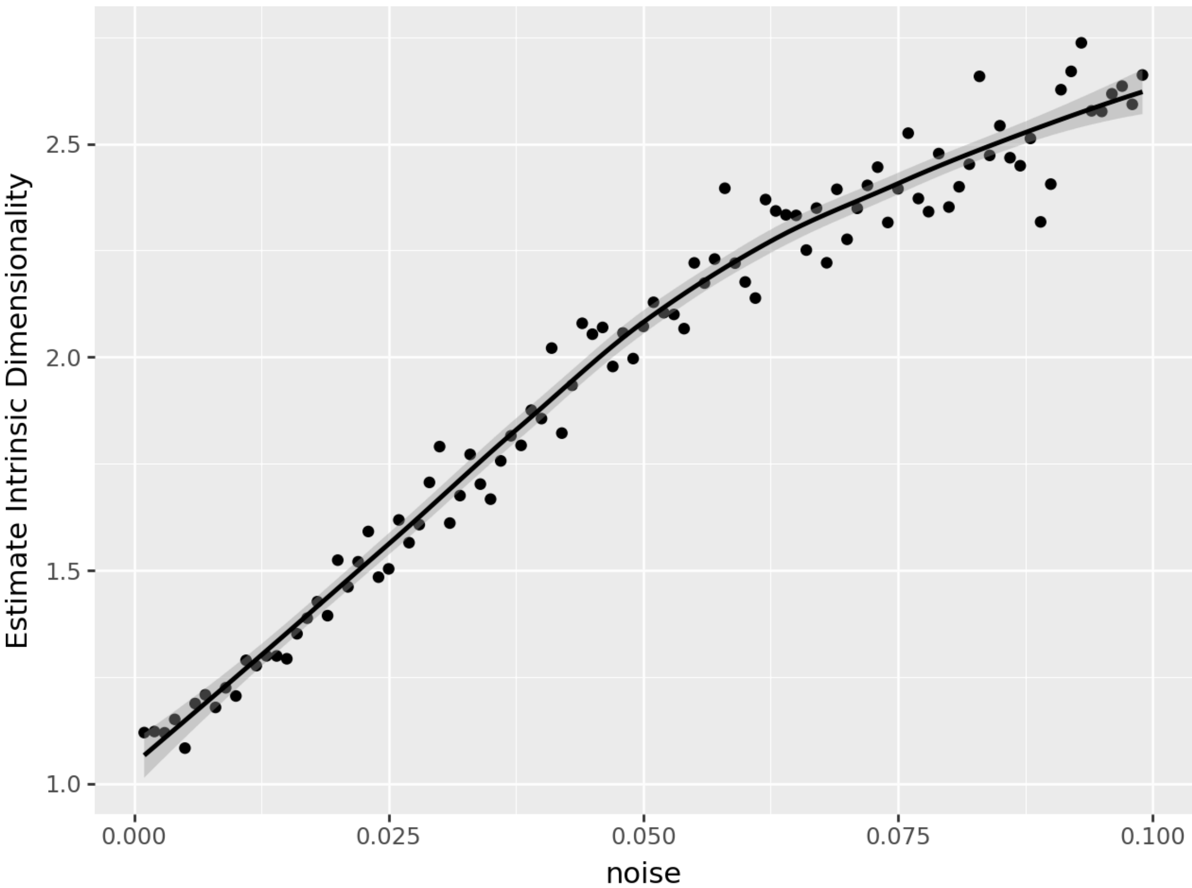

Figure 16: Intrinsic dimensionality estimate over even smaller noise sizes

Using even smaller noise sizes, we can see that the correlation is still present, without any threshold of noise before the increase starts. From this, we can conclude that, although there's a threshold for the IDE to stop increasing, there's no threshold before it starts increasing: the smallest amount of noise will always increase our IDE, making it a reliable metric for noise.

5)

a) Apply the Singular value decomposition (SVD) to dataset Leukemia, choose a proper lower dim k via elbow in the plot of singular values, then plot the dimension reduced data in both 2-dim and 3-dim (in case your k is at least three).

With the elbow method we identified $k = 6$ as the optimal number of components for dimensionality reduction, as shown on Figure 16. Using this information, we then performed Singular Value Decomposition (SVD) and visualized the reduced-dimension data in both 2D and 3D, as shown in Figure 17 and 18.

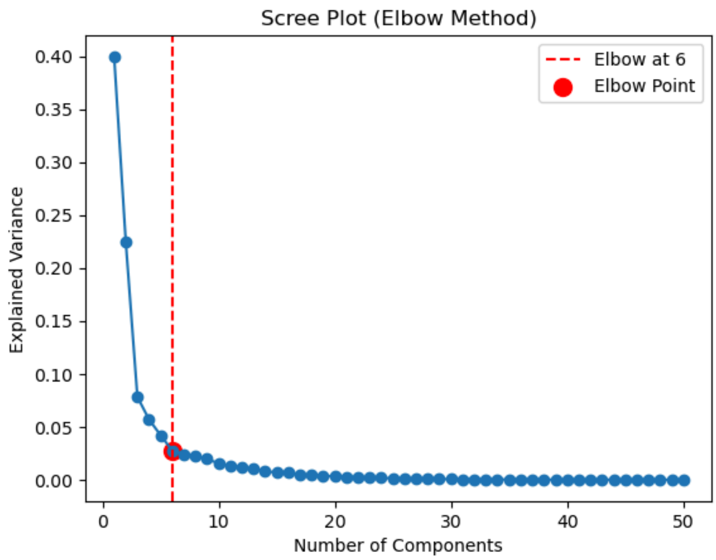

Figure 17: Elbow method.

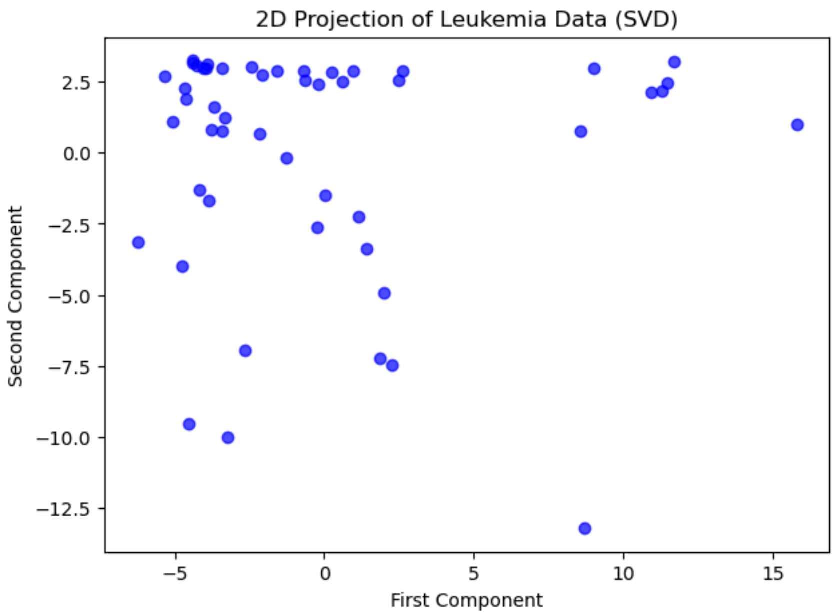

Figure 18: Singular Value Decomposition 2D visualization.

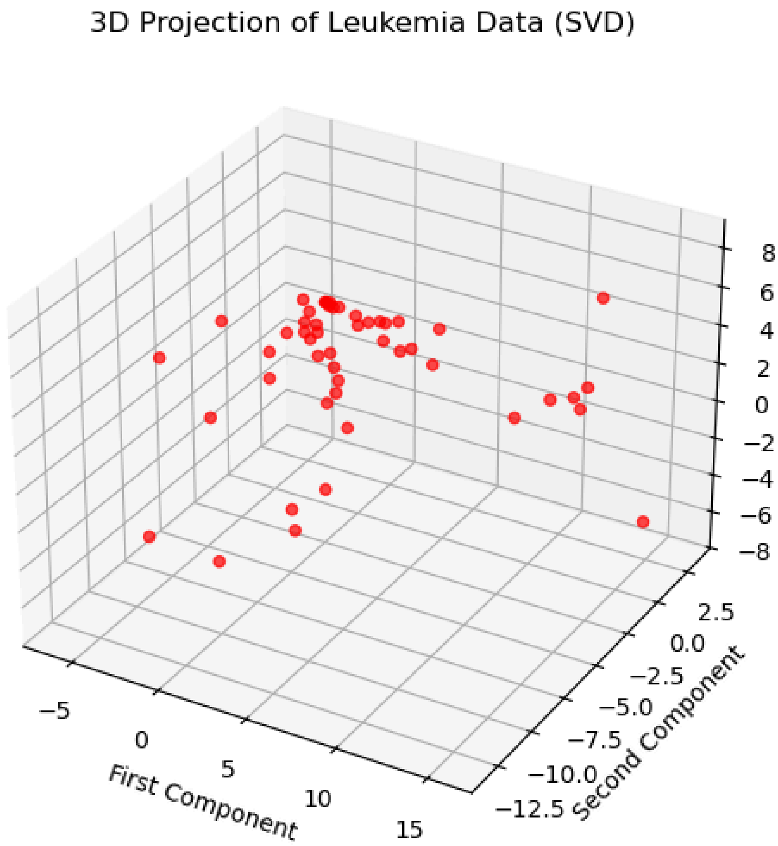

Figure 19: Singular Value Decomposition 3D visualization.

b) Try PCA (covariance) and compare the results from these two different methods.

From the figures below the results of Singular Value Decomposition (SVD) and Principal Component Analysis (PCA) applied to the same leukemia dataset can be compared. Both methods reveal similar distribution patterns with a significant difference in the range of the second component, where for SVD it spans from -13 to 3, and for PCA from -3 to 13. Additionally, the SVD results show a higher concentration of points in the upper half of the plot, while PCA points clustered mostly in the lower half. However, the overall shape and relative positioning of the points somehow appears consistent, indicating that both methods capture similar underlying patterns in the data.

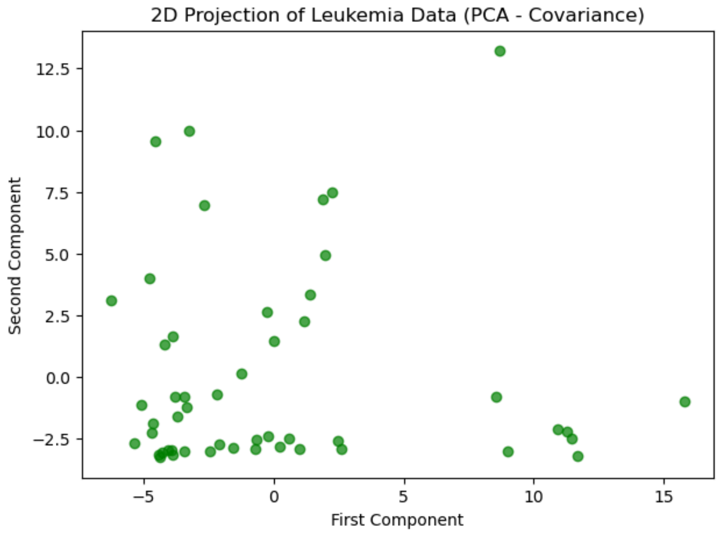

Figure 20: Singular Value Decomposition 3D visualization.

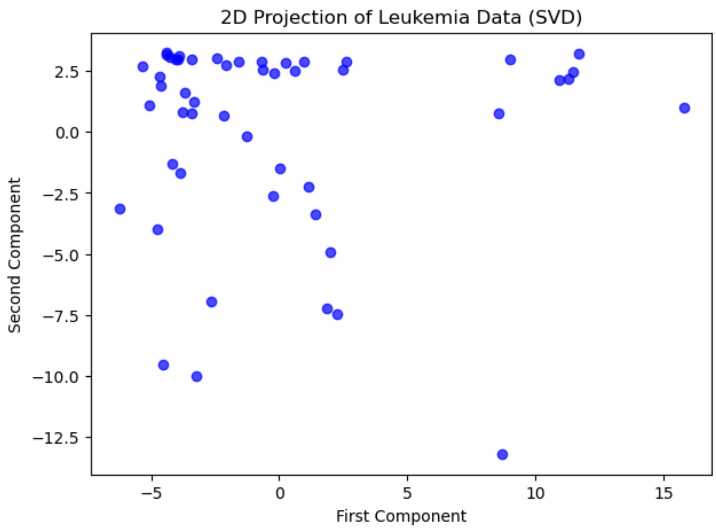

Figure 21: Singular Value Decomposition 2D visualization.

6) a) Using `genLDdata.m`, calculate the global and local intrinsic dimensionalities, using `MLE` and `CorrDim`, and compare the results. Use a neighborhood size of $k = 100$.

`genLDdata` generates random data points from the surface of a sphere, from a cube, and from 4 horizontal or vertical line segments coming out of the sphere. The global intrinsic dimensionality using MLE of the generated data was ≈ 1.52, and using *CorrDim* was ≈ 1.00. The local intrinsic dimensionality counts are presented in Table 1, and a scatter plot of these are available on Figure 22 and Figure 23.

Table 1: Local Intrinsic Dimensionality Counts

Dimension	Count (MLE)	Count (CorrDim)
1	2798 (46.63%)	2749 (45.82%)
2	1922 (32.03%)	2170 (36.17%)
3	1257 (20.95%)	1080 (18.00%)
4	23 (0.38%)	1 (0.02%)

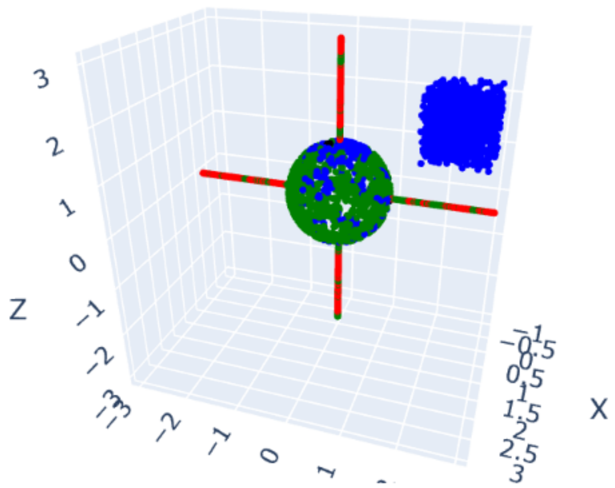

Figure 22: Local Intrinsic Dimensionality Scatter plot with `MLE`

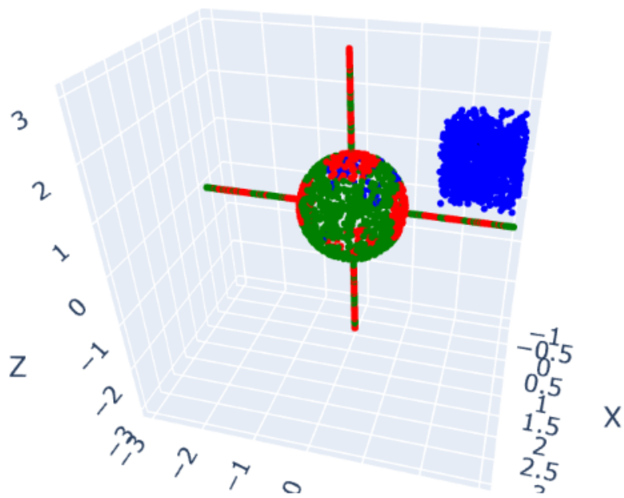

Figure 23: Local Intrinsic Dimensionality Scatter plot with `CorrDim`

From the figures, we can see that each shape, excluding the sphere surface, has a distinct local intrinsic dimensionality that matches our geometric intuition: the cube completely displays a local intrinsic dimensionality of 3 with both algorithms and the segments mostly displays a local intrinsic dimensionality of 1; however, since the sphere surface is a more complex surface, it seems the different algorithms classify it differently and not consistently: the output from the *MLE* presents the local intrinsic dimensionality of the sphere to be between 2 and 3, while the one with *CorrDim* presents it to be between 1 and 2 (the 1 points are closer to the connection between the segments and the sphere). Lastly, it seems the *MLE* could classify the segments more consistently than *CorrDim*.

b) Modify `genLDdata.m` so that:

- Instead of the sphere, we have the surface of an ellipsoid of $\frac{x_1^2}{4^2} + \frac{x_2^2}{5^2} + \frac{x_3^2}{6^2} = 1$,
- The line segments are replaced by the curve $x_1 = \frac{t \cos t^2}{1+t^2}, x_2 = \frac{t \sin t^2}{1+t^2}, x_3 = t, -2\pi \leq t \leq 2\pi$
- A line segment connecting a point from the ellipsoid and a point from the cube is added

Using this modified method, study the global and local intrinsic dimensionalities using `PackingNumbers`

The modified function can be found on Listing 17 and its output can be visualized on Figure 24, along with the output of the local intrinsic dimensionality with *PackingNumbers*. The global intrinsic dimensionality was ≈ 0.69.

Table 2: Local Intrinsic Dimensionality Counts

Dimension	Count (Packing Numbers)
1	2934 (73.350%)
2	679 (19.975%)
3	387 (9.675%)

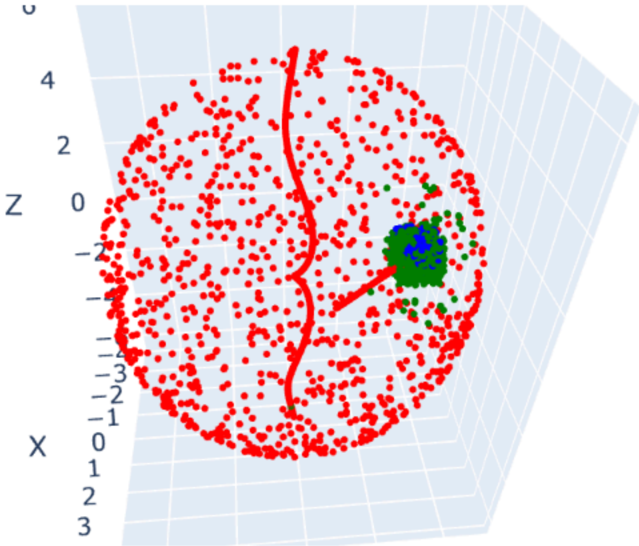

Figure 24: Local Intrinsic Dimensionality with *PackingNumbers*

We can see that the algorithm could not effectively find the expected local intrinsic dimensionality for any of our shapes, besides the connecting line between the cube and the sphere.

Appendix

1)

```python
import numpy as np
import matplotlib.pyplot as plt
import pyEDAkit as eda_lin

np.random.seed(42)
x1 = np.random.randn(150, 2) * 100 # 2 dimensions with high variance
x2 = np.random.randn(150, 4) # 4 dimensions with low variance
X = np.hstack((x1, x2))

print("\nPCA using covariance matrix")
PCA_cov = eda_lin.PCA(X, d=3, covariance = True, plot=True)

print("\nPCA using correlation matrix")
PCA_corr = eda_lin.PCA(X, d=2, covariance = False,  plot=True)
```

Listing 1: PCA using covariance vs correlation matrix for task *1*

```python
def PCA(X, d, covariance = True, plot = False):
    X_mean = X.mean(axis=0)
    X = X - X_mean
    S = None
    if covariance:
```

```
        S = np.cov(X, rowvar=False)
    else:
        S = np.corrcoef(X, rowvar=False)
    eigen_values, eigen_vectors = np.linalg.eig(S)
    sorted_index = np.argsort(eigen_values)[::-1]
    sorted_eigenvalue = eigen_values[sorted_index]
    sorted_eigenvectors = eigen_vectors[:, sorted_index]

    Z = (X @ sorted_eigenvectors)[:, :d]

    if plot:
        plt.figure(figsize=(8, 5))
        plt.plot(range(1, len(sorted_eigenvalue) + 1), sorted_eigenvalue,
        marker='o', linestyle='-')
        plt.plot(d, sorted_eigenvalue[d - 1], 'ro', label = 'd')
        plt.title('Scree Plot')
        plt.xlabel('Eigenvalue Index')
        plt.ylabel('Eigenvalue Magnitude')
        plt.legend()
        plt.grid(True)
        # scatter matrix of Z
        sns.pairplot(pd.DataFrame(Z, columns=[f"PC{i+1}" for i in
        range(d)]), diag_kind='kde')
        plt.legend()
        plt.show()

    return Z
```

Listing 2: PCA method for task *1*

2)

a)

```python
from plotnine import *
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as
LDA

plots = []
lda = LDA(n_components=1)
for target in iris_target_names:
  # filter out target
  iris_filtered = iris[iris["target"] != target]
  Z = lda.fit_transform(iris_filtered.drop(columns="target"),
  iris_filtered["target"])
```

```
  plot = (
    ggplot(
      pd.DataFrame({"Component": Z.flatten(), "target": iris_filtered["target"]}),
      aes(x = 'Component', color = 'target', fill = "target")
    )
    + geom_density(alpha = 0.6)
  )
  plots.append(plot)
plots[0]
plots[1]
plots[2]
```

Listing 3: LDA for case *a*

b)

```python
import numpy as np

data_other_way = pd.DataFrame(
  {
    "Width": np.concatenate([iris["sepal width (cm)"], iris["petal width (cm)"]]),
    "Length": np.concatenate([iris["sepal length (cm)"], iris["petal length (cm)"]]),
    "Sepal_Or_Petal": np.concatenate([np.repeat("Sepal", len(iris)), np.repeat("Petal", len(iris))])
  }
)
Z = lda.fit_transform(data_other_way.drop(columns="Sepal_Or_Petal"), data_other_way["Sepal_Or_Petal"])
(
  ggplot(
    pd.DataFrame({"Component": Z.flatten(), "Sepal_Or_Petal": data_other_way["Sepal_Or_Petal"]}),
    aes(x = 'Component', color = 'Sepal_Or_Petal', fill = "Sepal_Or_Petal")
  )
  + geom_density(alpha = 0.6)
)
```

Listing 4: LDA for case *b*

3)

a)

```python
stocks_df = pd.read_excel('../data/stockreturns.xlsx', names=list(range(1,11)))
stocks = stocks_df.values  # convert to numpy array

scaler = StandardScaler()
stocks_scaled = scaler.fit_transform(stocks)

# FA with rotation (default)
fa_default = FactorAnalysis(n_components=3, rotation='varimax')
fa_default.fit(stocks_scaled)
loadings_default = fa_default.components_.T

# FA NO rotation
fa_none = FactorAnalysis(n_components=3, rotation=None)
fa_none.fit(stocks_scaled)
loadings_none = fa_none.components_.T
```

Listing 5: Factor Analysis with and without rotation

```python
lab = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10'] # lables
t = np.linspace(-1, 1, 20) # reference line vector

def plot_factor_loadings(loadings, title):
    plt.figure(figsize=(8, 6))
    plt.plot(loadings[:, 0], loadings[:, 1], '.')
    plt.plot(t, np.zeros_like(t), 'b')  # horizontal line
    plt.plot(np.zeros_like(t), t, 'b')  # vertical line

    # add labels
    for i, txt in enumerate(lab):
        plt.annotate(txt, (loadings[i, 0] + 0.02, loadings[i, 1]))

    plt.xlabel('Factor 1')
    plt.ylabel('Factor 2')
    plt.title(title, fontweight='bold')
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.tight_layout()

plot_factor_loadings(loadings_default, 'Factor Analysis (Varimax)')
plot_factor_loadings(loadings_none, 'Factor Analysis (No rotation)')
```

Listing 6: Plot factor loadings method

```python
fa_scores = fa_default.transform(stocks_scaled)
```

```python
fig, axes = plt.subplots(3, 1, figsize=(8, 12))

axes[0].plot(fa_scores[:, 0], fa_scores[:, 1], '*')
axes[0].set_xlabel('Component 1')
axes[0].set_ylabel('Component 2')
axes[0].grid(True)

axes[1].plot(fa_scores[:, 1], fa_scores[:, 2], '*')
axes[1].set_xlabel('Component 2')
axes[1].set_ylabel('Component 3')
axes[1].grid(True)

axes[2].plot(fa_scores[:, 2], fa_scores[:, 0], '*')
axes[2].set_xlabel('Component 3')
axes[2].set_ylabel('Component 1')
axes[2].grid(True)

plt.tight_layout()
plt.show()
```

Listing 7: Factor scores for Factor Analysis with rotation

b)

Python

```python
data = pd.read_csv('../data/my_stock_returns.csv')

my_stocks_scaled = StandardScaler().fit_transform(data.values)
stock_names = data.columns.tolist()

# FA with varimax rotation
fa = FactorAnalysis(n_components=3, rotation='varimax')
fa.fit(my_stocks_scaled)
loadings = fa.components_.T

# Create 3D plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Plot points
ax.scatter(loadings[:, 0], loadings[:, 1], loadings[:, 2], s=50)
for i in range(len(loadings)):
    ax.text(loadings[i, 0], loadings[i, 1], loadings[i, 2],
    stock_names[i], size=8)

```

```python
for spine in ['xy', 'xz', 'yz']:
    ax.grid(True, ls='--', alpha=0.3, which='major', zorder=0)

ax.set_xlabel('Factor 1')
ax.set_ylabel('Factor 2')
ax.set_zlabel('Factor 3')
ax.set_title('3D Factor Analysis loadings plot')

# subplots for component visualization
fa_scores = fa.transform(my_stocks_scaled)
fig, axes = plt.subplots(3, 1, figsize=(8, 12))

axes[0].plot(fa_scores[:, 0], fa_scores[:, 1], '*')
axes[0].set_xlabel('Component 1')
axes[0].set_ylabel('Component 2')
axes[0].grid(True)

axes[1].plot(fa_scores[:, 1], fa_scores[:, 2], '*')
axes[1].set_xlabel('Component 2')
axes[1].set_ylabel('Component 3')
axes[1].grid(True)

axes[2].plot(fa_scores[:, 2], fa_scores[:, 0], '*')
axes[2].set_xlabel('Component 3')
axes[2].set_ylabel('Component 1')
axes[2].grid(True)

plt.tight_layout()
plt.show()
```

Listing 8: Factor analysis of collected stock data.

4)

a)

```python
import numpy as np
import pandas as pd
import plotly.express as px
def generate_random_numbers_from_these_functions(n = 500, lamb = 0, seed = None):
  if seed:
    np.random.seed(seed)
  theta = np.random.uniform(-2*np.pi, 2*np.pi, n)
```

```python
    x = (theta*np.cos(theta))/(1+theta**2) + lamb*np.random.uniform(0, 1, n)
    y = (theta*np.sin(theta))/(1+theta**2) + lamb*np.random.uniform(0, 1, n)
    z = theta + lamb*np.random.uniform(0, 1, n)
    df = pd.DataFrame({'x':x, 'y':y, 'z':z})
    return df

df = generate_random_numbers_from_these_functions(seed = 1)

fig = px.scatter_3d(df, x='x', y='y', z='z', color='z')
fig.show()
```

Listing 9: generating and plotting curve

```python
from pyEDAkit.IntrinsicDimensionality import id_pettis
id_pettis(df)
```

Listing 10: Estimate of intrinsic dimensionality

b)

```python
noised_df_point01 = generate_random_numbers_from_these_functions(lamb = 0.01, seed = 1)
noised_df_point05 = generate_random_numbers_from_these_functions(lamb = 0.05, seed = 1)
noised_df_point1 = generate_random_numbers_from_these_functions(lamb = 0.1, seed = 1)
noised_df_point5 = generate_random_numbers_from_these_functions(lamb = 0.5, seed = 1)
noised_df_point1 = generate_random_numbers_from_these_functions(lamb = 1, seed = 1)

import plotly.graph_objects as go
from plotly.subplots import make_subplots

fig = make_subplots(
    rows=1,
    cols=5,
    specs=[
        [
            {"type": "scatter3d"},
            {"type": "scatter3d"},
            {"type": "scatter3d"},
            {"type": "scatter3d"},
            {"type": "scatter3d"},
```

```
        ]
    ],
    subplot_titles=(
        "lamb = 0.01",
        "lamb = 0.05",
        "lamb = 0.1",
        "lamb = 0.5",
        "lamb = 1",
    ),
)
fig.add_trace(
    go.Scatter3d(
        x=noised_df_point01["x"],
        y=noised_df_point01["y"],
        z=noised_df_point01["z"],
        mode="markers",
        marker=dict(
            size=2, color=noised_df_point01["z"], colorscale="Viridis"
        ),
    ),
    row=1,
    col=1,
)
fig.add_trace(
    go.Scatter3d(
        x=noised_df_point05["x"],
        y=noised_df_point05["y"],
        z=noised_df_point05["z"],
        mode="markers",
        marker=dict(
            size=2, color=noised_df_point05["z"], colorscale="Viridis"
        ),
    ),
    row=1,
    col=2,
)
fig.add_trace(
    go.Scatter3d(
        x=noised_df_point1["x"],
        y=noised_df_point1["y"],
        z=noised_df_point1["z"],
        mode="markers",
```

```
        marker=dict(size=2, color=noised_df_point1["z"], colorscale="Viridis"),
    ),
    row=1,
    col=3,
)
fig.add_trace(
    go.Scatter3d(
        x=noised_df_point5["x"],
        y=noised_df_point5["y"],
        z=noised_df_point5["z"],
        mode="markers",
        marker=dict(size=2, color=noised_df_point5["z"], colorscale="Viridis"),
    ),
    row=1,
    col=4,
)
fig.add_trace(
    go.Scatter3d(
        x=noised_df_point1["x"],
        y=noised_df_point1["y"],
        z=noised_df_point1["z"],
        mode="markers",
        marker=dict(size=2, color=noised_df_point1["z"], colorscale="Viridis"),
    ),
    row=1,
    col=5,
)

fig.update_layout(title_text="3D Scatter Plots with Different Noise Levels")
fig.show()

# make from 0.1 to 2 noised dfs and calculate their eid
lambdas = np.array(range(1, 100, 1))*0.01
eids = []
np.random.seed(1)
for lamb in lambdas:
  noised_df = generate_random_numbers_from_these_functions(lamb = lamb)
  eid = id_pettis(noised_df)
  eids.append(eid)
```

```python
noise_df = pd.DataFrame({'sigma': lambdas, 'eid': eids})
# plot noise_df
from plotnine import *
(
  ggplot(noise_df, aes(x='sigma', y='eid'))
  + geom_point()
  + geom_smooth(method = "loess")
  + labs(x='noise', y='Estimate Intrinsic Dimensionality')
)
```

Listing 11: Noise generated and plotted

c)

```python
lambdas = np.array(range(1, 100, 1))*0.001
eids = []
np.random.seed(1)
for lamb in lambdas:
  noised_df = generate_random_numbers_from_these_functions(lamb = lamb)
  eid = id_pettis(noised_df)
  eids.append(eid)
noise_df = pd.DataFrame({'sigma': lambdas, 'eid': eids})
# plot noise_df
from plotnine import *
(
  ggplot(noise_df, aes(x='sigma', y='eid'))
  + geom_point()
  + geom_smooth(method = "loess")
  + labs(x='noise', y='Estimate Intrinsic Dimensionality')
)
```

5)

a)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import StandardScaler
from kneed import KneeLocator

data = loadmat('../data/leukemia.mat')
X = data["leukemia"]
print("Dataset shape:", X.shape)
```

```python

X_scaled = StandardScaler().fit_transform(X)

U, S, Vt = np.linalg.svd(X_scaled, full_matrices=False)

explained_variance = S**2 / np.sum(S**2)  # S contains singular values

plt.plot(range(1, len(S) + 1), explained_variance, marker='o', linestyle='-')
plt.xlabel("Number of Components")
plt.ylabel("Explained Variance")
plt.title("Scree Plot (Elbow Method)")

# find k
knee_locator = KneeLocator(range(1, len(S) + 1), explained_variance, curve="convex", direction="decreasing")
elbow_point = knee_locator.knee

# highlight elbow point
plt.axvline(x=elbow_point, color='r', linestyle='--', label=f'Elbow at {elbow_point}')
plt.scatter(elbow_point, explained_variance[elbow_point - 1], color='red', s=100, label='Elbow Point')
plt.legend()
plt.show()

k = 6 # reduce data to k=6
X_reduced_svd = U[:, :k] @ np.diag(S[:k])

# 2D plot
plt.figure(figsize=(7, 5))
plt.scatter(X_reduced_svd[:, 0], X_reduced_svd[:, 1], c='b', alpha=0.7)
plt.xlabel('First Component')
plt.ylabel('Second Component')
plt.title('2D Projection of Leukemia Data (SVD)')
plt.show()

# 3D plot
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_reduced_svd[:, 0], X_reduced_svd[:, 1], X_reduced_svd[:, 2], c='r', alpha=0.7)
ax.set_xlabel('First Component')
```

```python
ax.set_ylabel('Second Component')
ax.set_zlabel('Third Component')
ax.set_title('3D Projection of Leukemia Data (SVD)')
plt.show()
```

Listing 12: Singular value decomposition on leukemia dataset.

b)

```python
import pyEDAkit as eda_lin

X_reduced_pca_cov = eda_lin.PCA(X_scaled, d=6, covariance=True, plot=False)

# 2D plot
plt.figure(figsize=(7, 5))
plt.scatter(X_reduced_pca_cov[:, 0], X_reduced_pca_cov[:, 1], c='g', alpha=0.7)
plt.xlabel('First Component')
plt.ylabel('Second Component')
plt.title('2D Projection of Leukemia Data (PCA - Covariance)')
plt.show()
```

Listing 13: Singular value decomposition on leukemia dataset.

6)

```python
import numpy as np
import pyEDAkit as kit

def genLDdata(seed: int | None = 1):
  if seed is not None:
    np.random.seed(seed)
  # Sample from the surface of a sphere
  X1, X2, X3 = np.random.randn(3, 1000)
  lambda_ = np.sqrt(X1**2 + X2**2 + X3**2)
  X1, X2, X3 = X1 / lambda_, X2 / lambda_, X3 / lambda_
  X = np.column_stack((X1, X2, X3))

  # Sample from a cube
  X1, X2, X3 = np.random.rand(3, 1000) + 2
  XX = np.column_stack((X1, X2, X3))

  # Sample from lines attached to a sphere
  L1 = np.column_stack((np.zeros(1000), np.zeros(1000), 2 * np.random.rand(1000) + 1))
```

```python
    L2 = np.column_stack((np.zeros(1000), np.zeros(1000), -2 * np.random.rand(1000) - 1))
    L3 = np.column_stack((np.zeros(1000), 2 * np.random.rand(1000) + 1, np.zeros(1000)))
    L4 = np.column_stack((np.zeros(1000), -2 * np.random.rand(1000) - 1, np.zeros(1000)))

    A = np.vstack((X, XX, L1, L2, L3, L4))
    return A
A = genLDdata()
print(kit.IntrinsicDimensionality.MLE(A))
```

Listing 14: `genLDdata` function in python

Python

```python
import numpy as np
import pandas as pd
import scipy.spatial.distance as dist

def compute_local_intrinsic_dimensionality(A, method, k=100):
  # Compute pairwise Euclidean distances
  Ad = dist.squareform(dist.pdist(A))

  # Get the dimensions of A
  nr, nc = A.shape
  Ldim = np.zeros(nr)

  # Sort distances and get indices
  Ads = np.sort(Ad, axis=1)
  J = np.argsort(Ad, axis=1)

  # Compute local intrinsic dimensionality
  for m in range(nr):
    Ldim[m] = method(A[J[m, :k], :])

  # Adjust local dimensions
  Ldim[Ldim > 3] = 4
  Ldim = np.ceil(Ldim).astype(int)

  # Tabulate results
  unique, counts = np.unique(Ldim, return_counts=True)
  percentages = (counts / nr) * 100
  tabulation_df = pd.DataFrame({'Dimension': unique, 'Count': counts, 'Percentage': np.round(percentages, 3)})

```

```
  return Ldim, tabulation_df
```

Listing 15: Local intrinsic calculation function (generated from *Example 2.10*)

Python

```python
import plotly.graph_objects as go
# Scatter plot with color map
colors = {1: 'red', 2: 'green', 3: 'blue', 4: 'black'}
fig = go.Figure()
labels = [1, 2, 3, 4]
for label in labels:
    indices = np.where(Ldim == label)[0]
    if len(indices) > 0:
        fig.add_trace(go.Scatter3d(
            x=A[indices, 0], y=A[indices, 1], z=A[indices, 2],
            mode='markers',
            marker=dict(color=colors[label], size=2),
            name=f"Dim {label}"
        ))

fig.update_layout(scene=dict(xaxis_title='X', yaxis_title='Y',
zaxis_title='Z'), title="Intrinsic Dimensionality Scatterplot with [x]")
fig.show()
```

Listing 16: Local Intrinsic Dimensionality scatter plot

Python

```python
def genLDdata_mod(seed: int | None = 1):
  if seed is not None:
    np.random.seed(seed)

  # Sample from the surface of an ellipsoid
  X1, X2, X3 = np.random.randn(3, 1000)
  lambda_ = np.sqrt((X1/4)**2 + (X2/5)**2 + (X3/6)**2)
  X1, X2, X3 = X1 / lambda_, X2 / lambda_, X3 / lambda_
  X = np.column_stack((X1, X2, X3))

  # Sample from a cube
  X1, X2, X3 = np.random.rand(3, 1000) + 2
  XX = np.column_stack((X1, X2, X3))

  # Sample of the curve
  t = np.random.uniform(-2*np.pi, 2*np.pi, 1000)
  X1 = (t * np.cos(t))/(1 + t**2)
  X2 = (t * np.sin(t))/(1 + t**2)
  X3 = t
```

```python
    X_curve = np.column_stack((X1, X2, X3))

    # Sample from segment connecting ellipsoid and the cube
    point1 = X[np.random.randint(len(X))]
    point2 = XX[np.random.randint(len(XX))]
    t = np.linspace(0, 1, 1000)
    # linear interpolation
    X1 = point1[0] + t * (point2[0] - point1[0])
    X2 = point1[1] + t * (point2[1] - point1[1])
    X3 = point1[2] + t * (point2[2] - point1[2])
    X_segment = np.column_stack((X1, X2, X3))

    A = np.vstack((X, XX, X_curve, X_segment))
    return A
```

Listing 17: `genLDdata` modified

```python
A = genLDdata()

print("global intrinsic dimensionality (MLE):", kit.IntrinsicDimensionality.MLE(A))
Ldim, tabulation_df = compute_local_intrinsic_dimensionality(A, kit.IntrinsicDimensionality.MLE)
print(tabulation_df) # Ldim used for the plotting

print("global intrinsic dimensionality (CorrDim):", kit.IntrinsicDimensionality.corr_dim(A))
Ldim, tabulation_df = compute_local_intrinsic_dimensionality(A, kit.IntrinsicDimensionality.corr_dim)
print(tabulation_df)

A = genLDdata_mod()
print("global intrinsic dimensionality (PackingNumbers):", kit.IntrinsicDimensionality.packing_numbers(A))
Ldim, tabulation_df = compute_local_intrinsic_dimensionality(A, kit.IntrinsicDimensionality.packing_numbers)
```

Listing 18: Function calls for 6)