Sign in to chat with Claude, Gemini, ChatGPT, DeepSeek and Qwen for free after one minute. Please close this popup if you're not interested.
Sign in for free and unlock the full power of AI models and the ability to save your conversations. Please close this popup if you're not interested. Sign in for free and unlock all your conversations.

Unsupervised Algorithm And building machine learning Model

Data Processing And building machine learning Model

Steps of Preprocessing of Unsupervised Data

Step 1: Import necessary Library Step2: Read dataset Step3: Sanity Check Step 4: Exploratory Data Analysis Step5: Missing Value Treatments Step 6: Outliers Treatment Step 7: Scaling Step 8: Clustering Step 9: Dendrogram Step 10: Evaluation Step 11: Scatter Plot of Clusters Step 12: Conclusion Step 13: Cluster Step 14: Finishing of Data

this dataset available in the sklearn library

Step 1: Import Necessary Library

```
In [1]: import numpy as np
        import pandas as pd
        import seaborn as sns
        import matplotlib.pyplot as plt
        from sklearn.datasets import load_iris
        import warnings ## ignore warnings
        ## Ignore warnings
        warnings.filterwarnings('ignore')
```

```
In [2]: ## This is the iris dataset, the target variable represents the species (setosa, versicolor, virginica). Since clustering aims to find
        ## natural groups in the features (sepal length, sepal width, petal length, petal width), the species variable can be dropped before applying
        ## clustering.
```

Step 2: Read DataSet

```
In [3]: iris = load_iris ()
```

```
In [4]: ## Create a DataFrame using the features (excluding the target)
        df = pd.DataFrame(data=iris.data, columns=iris.feature_names)

        ## Optional: Add the target column for reference
        df['target'] = iris.target

        # Display basic information about the dataset
        print("Dataset shape:", df.shape)
        print("Columns:", df.columns)
        print("Missing values:")
        print(df.isnull().sum())
        print("\n")
        df.columns
```

```
Dataset shape: (150, 5)
Columns: Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
       'petal width (cm)', 'target'],
      dtype=object)
Missing values:
sepal length (cm)    0
sepal width (cm)     0
petal length (cm)    0
petal width (cm)     0
target               0
dtype: int64

Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
       'petal width (cm)', 'target'],
      dtype=object)
```

```
In [5]: df.columns
```

```
Out[5]: Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
       'petal width (cm)', 'target'],
      dtype=object)
```

```
In [6]: df.head()
```

```
Out[6]:   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  target
        0                5.1               3.5                1.4               0.2       0
        1                4.9               3.0                1.4               0.2       0
        2                4.7               3.2                1.3               0.2       0
        3                4.6               3.1                1.5               0.2       0
        4                5.0               3.6                1.4               0.2       0
```

```
In [7]: df.tail()
```

```
Out[7]:   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  target
        145                6.7               3.0                5.2               2.3       2
        146                6.3               2.5                5.0               1.9       2
        147                6.5               3.0                5.2               2.0       2
        148                6.2               3.4                5.4               2.3       2
        149                5.9               3.0                5.1               1.8       2
```

Note: we're only using the feature columns (sepal length, sepal width, petal length, and petal width) for the unsupervised learning. Keeping the target column because it is not removed for clustering.

Step 3: Sanity Check of Data

```
In [8]: df.info()
```

```
Out[8]: <class 'pd.DataFrame'>
        RangeIndex: 150 entries, 0 to 149
        Data columns (total 5 columns):
         #   Column             Non-Null Count  Dtype
        ---  ------             --------------  -----
         0   sepal length (cm)  150 non-null    float64
         1   sepal width (cm)   150 non-null    float64
         2   petal length (cm)  150 non-null    float64
         3   petal width (cm)   150 non-null    float64
         4   target             150 non-null    int64
        dtypes: float64(4), int64(1)
        memory usage: 6.0 KB
```

```
In [9]: df.describe()
```

```
Out[9]:   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  target
        count   150.000000        150.000000         150.000000        150.000000  150.0
        mean      5.843333          3.057333           3.758000          1.199333    1.0
        std       0.828066          0.435866           1.765298          0.762238    0.8
        min       4.300000          2.000000           1.000000          0.100000    0.0
        25%       5.100000          2.800000           1.600000          0.300000    0.0
        50%       5.800000          3.000000           4.350000          1.300000    1.0
        75%       6.400000          3.300000           5.100000          1.800000    2.0
        max       7.900000          4.400000           6.900000          2.500000    2.0
```

```
In [10]: ## Check for missing values
         print("Missing values:")
         df.isnull().sum()
```

```
Out[10]: Missing values:
         sepal length (cm)  0
         sepal width (cm)   0
         petal length (cm)  0
         petal width (cm)   0
         target             0
         dtype: int64
```

```
In [11]: ## Visualizing the distribution of missing values
         sns.heatmap(df.isnull())
```

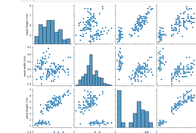
```
Out[11]: <Axes: >
```

No missing values.No missing values

```
In [12]: ## Optional: Save the cleaned data (if needed)
         df.to_csv('cleaned_iris.csv', index=False)
```

```
In [13]: ## Import necessary libraries for data exploration
         import matplotlib.pyplot as plt
         import seaborn as sns

         ## Set the style for better visualizations
         plt.style.use('seaborn')

         ## Create a figure with multiple subplots to see all distributions
         fig, axes = plt.subplots(2, 2, figsize=(10, 8))
         fig.suptitle('Distribution of Features', fontsize=16)

         ## Plot each feature
         features = ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
         for i, feature in enumerate(features):
             ax = axes[i//2, i%2]
             df[feature].hist(bins=30, ax=ax)
             ax.set_title(feature)
             ax.set_xlabel('')
             ax.set_ylabel('Frequency')

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

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.10/dist-packages/matplotlib/pyplot.py", line 2095, in hist
    return gca().hist(
  File "/usr/local/lib/python3.10/dist-packages/matplotlib/axes/_axes.py", line 6812, in hist
    raise ValueError(f"Unknown property {k!r}")
ValueError: Unknown property 'density'
```




```
In [14]: df.hist(figsize=(10,8))
```

```
Out[14]: array([[<Axes: title={'center': 'sepal length (cm)'}>,
                 <Axes: title={'center': 'sepal width (cm)'}>],
                [<Axes: title={'center': 'petal length (cm)'}>,
                 <Axes: title={'center': 'petal width (cm)'}>],
                [<Axes: title={'center': 'target'}>, <Axes: >]],
               dtype=object)
```

```
In [15]: ## Boxplot to understand the outliers
         df.boxplot(figsize=(10,6))
```

```
Out[15]: <Axes: >
```

```
In [16]: ## Optional: Pair plot to understand relationships
         ## This will show all the pairwise relationships between features
         sns.pairplot(df, hue='target')
```

```
Out[16]: <seaborn.axisgrid.PairGrid at 0x7f8a1c2d4e50>
```

Step 4: Exploratory Data Analysis

```
In [17]: ## Basic statistics for numerical columns
         df.describe()
```

```
Out[17]:   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  target
         count   150.000000        150.000000         150.000000        150.000000  150.0
         mean      5.843333          3.057333           3.758000          1.199333    1.0
         std       0.828066          0.435866           1.765298          0.762238    0.8
         min       4.300000          2.000000           1.000000          0.100000    0.0
         25%       5.100000          2.800000           1.600000          0.300000    0.0
         50%       5.800000          3.000000           4.350000          1.300000    1.0
         75%       6.400000          3.300000           5.100000          1.800000    2.0
         max       7.900000          4.400000           6.900000          2.500000    2.0
```

```
In [18]: ## Check for missing values
         print("Missing values:")
         df.isnull().sum()
```

```
Out[18]: Missing values:
         sepal length (cm)  0
         sepal width (cm)   0
         petal length (cm)  0
         petal width (cm)   0
         target             0
         dtype: int64
```

```
In [19]: ## Optional: Save the cleaned data (if needed)
         df.to_csv('cleaned_iris.csv', index=False)
```

```
In [20]: ## Create a more detailed distribution plot for each feature
         ## This will show both histogram and kernel density estimate
         fig, axes = plt.subplots(2, 2, figsize=(10, 8))
         fig.suptitle('Detailed Distribution of Features', fontsize=16)

         for i, feature in enumerate(features):
             ax = axes[i//2, i%2]
             sns.histplot(df[feature], kde=True, ax=ax)
             ax.set_title(feature)
             ax.set_xlabel('')
             ax.set_ylabel('Frequency')

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

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.10/dist-packages/matplotlib/pyplot.py", line 2095, in hist
    return gca().hist(
  File "/usr/local/lib/python3.10/dist-packages/matplotlib/axes/_axes.py", line 6812, in hist
    raise ValueError(f"Unknown property {k!r}")
ValueError: Unknown property 'density'
```

```
In [21]: ## Create a correlation matrix
         correlation = df.corr()
         print("Correlation matrix:")
         correlation
```

```
Out[21]: Correlation matrix:
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
sepal length (cm)	1.000000	-0.109369	0.871754	0.817954	0.782561
sepal width (cm)	-0.109369	1.000000	-0.420516	-0.356544	-0.426658
petal length (cm)	0.871754	-0.420516	1.000000	0.962757	0.949035
petal width (cm)	0.817954	-0.356544	0.962757	1.000000	0.956547
target	0.782561	-0.426658	0.949035	0.956547	1.000000

```
In [22]: ## Create a heatmap of the correlations
         plt.figure(figure=(10, 8))
         sns.heatmap(correlation, annot=True, cmap='coolwarm', center=0)
         plt.title('Correlation Heatmap')
         plt.show()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.10/dist-packages/matplotlib/figure.py", line 2457, in figure
    raise TypeeError("Cannot pass "
TypeeError: Cannot pass 'figure' keyword argument to figure().
```

5. Scatter Plot of Clusters (K-Means with PCA)

```
In [23]: from sklearn.decomposition import PCA
         from sklearn.cluster import KMeans

         # Apply PCA to reduce to 2 principal components
         pca = PCA(n_components=2)
         X_pca = pca.fit_transform(iris.data)

         # Apply KMeans clustering (3 clusters since Iris has 3 species)
         kmeans = KMeans(n_clusters=3, random_state=42)
         clusters = kmeans.fit_predict(X_pca)

         # Plot the clusters
         plt.figure(figsize=(10, 8))
         scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=clusters, cmap='viridis', s=50)
         plt.xlabel('Principal Component 1')
         plt.ylabel('Principal Component 2')
         plt.title('K-Means Clustering (PCA Reduced Iris Data)')
         plt.colorbar(scatter, label='Cluster')
         plt.grid(True)
         plt.show()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.10/dist-packages/matplotlib/figure.py", line 2457, in figure
    raise TypeeError("Cannot pass "
TypeeError: Cannot pass 'figure' keyword argument to figure().
```

Method outliers are removed from the column

In [24]: `df.describe()`

Out[24]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333	1.000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

In [25]: `df.info()`

Out[25]:
```
<class'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    int64
dtypes: float64(4), int64(1)
memory usage: 7.0 KB
```

Here is no need of encoding because no categorical data found.

5. Scaling

Scaling is a preprocessing technique used in data preparation for machine learning and statistical analysis. The main purpose of scaling is to adjust the range of the features in the dataset so that they contribute equally to the model's learning and prevent issues arising due to the different scales and units of measurement.

Why is Scaling Important?
Scaling is important for the following reasons:
1. Equal Contribution: Features with larger magnitudes can disproportionately influence the results of certain algorithms. Scaling ensures that all features contribute equally to the model.
2. Improved Performance: Many machine learning algorithms converge faster and perform better when the data is scaled. For instance, gradient descent-based algorithms (like linear regression, logistic regression, and neural networks) often require scaled data to converge efficiently.
3. Distance Calculations: Algorithms like k-nearest neighbors (KNN) and support vector machines (SVM) rely on distance calculations between data points. Scaling ensures that these calculations are not biased by the scale of the features.
4. Regularization: Some regularization techniques ensure that the data is normally distributed. Scaling can help to meet this assumption and improve the effectiveness of the regularization.

Common Scaling Techniques:
1. Standardization (Z-score Normalization):
Formula: $z = \frac{x - \mu}{\sigma}$
Range: Values the data is centered around the mean, typically ranging from -3 to 3.
Use Case: Suitable for algorithms that assume normally distributed data, such as neural networks.

2. Min-Max Scaling (Normalization):
Formula: $x' = \frac{x - x_{min}}{x_{max} - x_{min}}$
Range: Scales the data to a fixed range, typically [0, 1].
Use Case: Useful when the data has a bounded range of values, such as neural networks.

3. Robust Scaling:
Formula: $x' = \frac{x - \text{median}(x)}{\text{IQR}(x)}$
Range: Scales the data based on the median and interquartile range (IQR), which is more robust to outliers.
Use Case: Suitable for algorithms that are sensitive to outliers, such as decision trees.

4. Logarithmic Scaling:
Formula: $x' = \log(x)$
Range: Transforms the data to a logarithmic scale.
Use Case: Useful when the data has a skewed distribution or when the relationship between the features is multiplicative.

5. Unit Vector Scaling (Normalization):
Formula: $x' = \frac{x}{\|x\|}$
Range: Scales the data to have a unit norm (length of 1).
Use Case: Commonly used in text mining and natural language processing.

Implementation in Scikit-Learn:
Scikit-learn provides various scalers in the module `sklearn.preprocessing`. For example:
- `StandardScaler` for standardization.
- `MinMaxScaler` for min-max scaling.
- `RobustScaler` for robust scaling.
- `LogTransformer` for logarithmic scaling.
- `Normalizer` for unit vector scaling.

Choosing the Right Scaler:
The choice of scaler depends on the nature of the data and the requirements of the machine learning algorithm. For example, if the data is normally distributed, standardization might be suitable. If the data has outliers, robust scaling or min-max scaling might be more appropriate.

In [26]: `df`

Out[26]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

Encoding

In [27]: `cat_feed_indexlist=[]`

In [28]: `cat_indexlist=df.dtypes.type==object`

```
cat_indexlist.sum()
```

```
sepal length (cm)  False
sepal width (cm)   False
petal length (cm)  False
petal width (cm)   False
target             False
dtype: bool
```

Here is no need of encoding because no categorical data found.

6. Scaling

Scaling is a preprocessing technique used in data preparation for machine learning and statistical analysis. The main purpose of scaling is to adjust the range of the features in the dataset so that they contribute equally to the model's learning and prevent issues arising due to the different scales and units of measurement.

Why is Scaling Important?
Scaling is important for the following reasons:
1. Equal Contribution: Features with larger magnitudes can disproportionately influence the results of certain algorithms. Scaling ensures that all features contribute equally to the model.
2. Improved Performance: Many machine learning algorithms converge faster and perform better when the data is scaled. For instance, gradient descent-based algorithms (like linear regression, logistic regression, and neural networks) often require scaled data to converge efficiently.
3. Distance Calculations: Algorithms like k-nearest neighbors (KNN) and support vector machines (SVM) rely on distance calculations between data points. Scaling ensures that these calculations are not biased by the scale of the features.
4. Regularization: Some regularization techniques ensure that the data is normally distributed. Scaling can help to meet this assumption and improve the effectiveness of the regularization.

Common Scaling Techniques:
1. Standardization (Z-score Normalization):
Formula: $z = \frac{x - \mu}{\sigma}$
Range: Values the data is centered around the mean, typically ranging from -3 to 3.
Use Case: Suitable for algorithms that assume normally distributed data, such as neural networks.

2. Min-Max Scaling (Normalization):
Formula: $x' = \frac{x - x_{min}}{x_{max} - x_{min}}$
Range: Scales the data to a fixed range, typically [0, 1].
Use Case: Useful when the data has a bounded range of values, such as neural networks.

3. Robust Scaling:
Formula: $x' = \frac{x - \text{median}(x)}{\text{IQR}(x)}$
Range: Scales the data based on the median and interquartile range (IQR), which is more robust to outliers.
Use Case: Suitable for algorithms that are sensitive to outliers, such as decision trees.

4. Logarithmic Scaling:
Formula: $x' = \log(x)$
Range: Transforms the data to a logarithmic scale.
Use Case: Useful when the data has a skewed distribution or when the relationship between the features is multiplicative.

5. Unit Vector Scaling (Normalization):
Formula: $x' = \frac{x}{\|x\|}$
Range: Scales the data to have a unit norm (length of 1).
Use Case: Commonly used in text mining and natural language processing.

Implementation in Scikit-Learn:
Scikit-learn provides various scalers in the module `sklearn.preprocessing`. For example:
- `StandardScaler` for standardization.
- `MinMaxScaler` for min-max scaling.
- `RobustScaler` for robust scaling.
- `LogTransformer` for logarithmic scaling.
- `Normalizer` for unit vector scaling.

Choosing the Right Scaler:
The choice of scaler depends on the nature of the data and the requirements of the machine learning algorithm. For example, if the data is normally distributed, standardization might be suitable. If the data has outliers, robust scaling or min-max scaling might be more appropriate.

6. Feature Selection

```
import numpy as np, pandas as pd, matplotlib as mpl, matplotlib.pyplot as plt, seaborn as sns
%matplotlib inline
```

```
from sklearn.feature_selection import VarianceThreshold
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
```

```
# load the iris dataset
X = df.drop(['target'], axis=1)
y = df['target']X.shape, y.shape, np.unique(y), iris.feature_names
```

```
# Apply Variance Threshold
# This will remove features with low variance.
# By default, it removes features with zero variance.
sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
X_sel = sel.fit_transform(X)X_sel.shape
```

```
# Print retained features
print("Shape after feature selection:", X_sel.shape)
print(X_sel[:5])
```

```
Shape after feature selection: (150, 3)
[[5.1 1.4 0.2]
 [4.9 1.4 0.2]
 [4.7 1.3 0.2]
 [4.6 1.5 0.2]
 [5.0 1.4 0.2]]
```

```
# Apply PCA to reduce the dataset to 2 principal components
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)X_pca.shape
```

```
# Display the explained variance ratio for each principal component
print("Explained variance ratio:", pca.explained_variance_ratio_)
```

```
Explained variance ratio: [0.92461872 0.05306648]
```

```
# Apply t-SNE to reduce the dataset to 2 dimensions
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X)X_tsne.shape
```

```
# Plot the t-SNE reduced data
plt.figure(figure=None, dpi=None)
plt.scatter(X_tsne[:, 0], X_tsne[:, 1])
plt.xlabel('t-SNE Feature 1')
plt.ylabel('t-SNE Feature 2')
plt.show()
```

```
# Apply Recursive Feature Elimination (RFE) with a Random Forest classifier
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=100)
rfe = RFE(estimator=model, n_features_to_select=2)X_rfe = rfe.fit_transform(X, y)
```

```
# Print the selected features
print("Selected features after RFE:", X_rfe.shape)
print(X_rfe[:5])
```

```
Selected features after RFE: (150, 2)
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

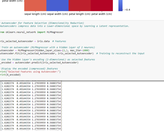
```
# Apply Principal Component Analysis (PCA)
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)X_pca.shape
```

```
# Print the explained variance ratio for each principal component
print("Explained variance ratio:", pca.explained_variance_ratio_)
```

```
Explained variance ratio: [0.92461872 0.05306648]
```

```
# Apply t-SNE to reduce the dataset to 2 components
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X)X_tsne.shape
```

```
# Plot the t-SNE reduced data
plt.figure(figure=None, dpi=None)
plt.scatter(X_tsne[:, 0], X_tsne[:, 1])
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.show()
```




```
# Apply Recursive Feature Elimination (RFE) with a Random Forest classifier
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=100)
rfe = RFE(estimator=model, n_features_to_select=2)X_rfe = rfe.fit_transform(X, y)
```

```
# Print the selected features
print("Selected features after RFE:", X_rfe.shape)
print(X_rfe[:5])
```

```
Selected features after RFE: (150, 2)
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding

```
# Apply Variance Thresholding
sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
X_sel = sel.fit_transform(X)
```

```
# Print the shape of the dataset after applying variance threshold
print("Shape after feature selection:", X_sel.shape)
```

```
Shape after feature selection: (150, 3)
```

```
# Print the selected features
print(X_sel[:5])
```

```
[[5.1 1.4 0.2]
 [4.9 1.4 0.2]
 [4.7 1.3 0.2]
 [4.6 1.5 0.2]
 [5.0 1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [29]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [30]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [31]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [32]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [33]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [34]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [35]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [36]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [37]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [38]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [39]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [40]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [41]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [42]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [43]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [44]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [45]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [46]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [47]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [48]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [49]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [50]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [51]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [52]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [53]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [54]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

```
Shape after variance thresholding: (150, 2)
```

```
# Print the first 5 rows of the reduced dataset
print("First 5 rows after variance thresholding:")
print(X_reduced[:5])
```

```
First 5 rows after variance thresholding:
[[1.4 0.2]
 [1.4 0.2]
 [1.3 0.2]
 [1.5 0.2]
 [1.4 0.2]]
```

Variance Thresholding is a simple yet effective method for feature selection. It transforms the dataset into a lower-dimensional space by capturing the directions (principal components) with the most variance.

In [55]: `import numpy as np`

```
# Calculate the variance of each feature
variances = np.var(X, axis=0)
print("Variances:", variances)
```

```
Variances: [0.68112222 0.18871289 3.09242489 0.57853156]
```

```
# Set a threshold for variance
threshold = 0.8
```

```
# Apply the variance threshold
mask = variances > threshold
X_reduced = X[:, mask]
print("Shape after variance thresholding:", X_reduced.shape)
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: