Dimensionality Reduction

Dimensionality reduction is a process used to reduce the number of input variables in a dataset, while retaining as much relevant information as possible. It transforms high-dimensional data into a lower-dimensional space, making it more manageable and interpretable without losing important patterns or relationships.

Why Do We Need Dimensionality Reduction?

1. **Curse of Dimensionality**: When dealing with high-dimensional data, certain algorithms (like k-NN, clustering, or regression models) can become less effective because distances between data points become harder to measure accurately. As the number of dimensions increases, the volume of the data space increases exponentially, making the data points sparse. This is known as the "curse of dimensionality."

2. **Computational Efficiency**: High-dimensional datasets require more computational power for both processing and storage. By reducing the number of dimensions, we reduce the computational costs associated with training machine learning models, making algorithms faster.

3. **Overfitting Prevention**: High-dimensional data often contain irrelevant or noisy features that do not contribute meaningfully to the prediction task. These features can lead to overfitting, where the model becomes too specific to the training data and fails to generalize to new data. Reducing dimensions helps mitigate this risk.

4. **Visualization**: Visualization of high-dimensional data is difficult. By reducing dimensions to two or three, it becomes easier to visualize and understand the data, revealing underlying patterns or clusters.

Motivation Behind Dimensionality Reduction

1. **Data Simplification**: Often, not all features in a dataset are equally important. Many features might be redundant or highly correlated. Dimensionality reduction helps simplify data by removing such redundancies and preserving only the essential features, which improves the interpretability of models.

2. **Improved Model Performance**: With fewer dimensions, models may become more robust and generalizable. Simplified models can also improve accuracy, as irrelevant or noisy features are removed from consideration, focusing on the most informative aspects of the data.

3. **Easier Data Storage and Transmission**: Lower-dimensional data is smaller in size, which reduces storage requirements and makes data transmission faster and easier, particularly when working with large datasets or streaming data.

Common Dimensionality Reduction Techniques

1. **Principal Component Analysis (PCA)**: PCA transforms the data into new features called "principal components," which are linear combinations of the original features. These components are chosen to maximize variance, preserving the most information in the data while reducing dimensions.

2. **Linear Discriminant Analysis (LDA)**: LDA is a supervised dimensionality reduction technique that maximizes the separation between different classes in the data by finding the linear discriminants.

3. **t-SNE (t-Distributed Stochastic Neighbor Embedding)**: t-SNE is a nonlinear dimensionality reduction technique that visualizes high-dimensional data by mapping it to a lower-dimensional space (usually 2D or 3D), while preserving the local relationships between data points.

4. **Autoencoders**: In deep learning, autoencoders are a type of neural network used to learn compressed representations of input data, often used for dimensionality reduction in complex, nonlinear datasets.

Conclusion

Dimensionality reduction helps manage the challenges of high-dimensional data by reducing noise, improving computation efficiency, preventing overfitting, and facilitating data visualization. It is a crucial step when working with large datasets, ensuring models remain accurate and interpretable.

Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a dimensionality reduction technique used to transform high-dimensional data into a lower-dimensional space while preserving as much variance (information) as possible. It is commonly used for data compression, noise reduction, and as a preprocessing step for machine learning algorithms.

The core idea of PCA is to identify the directions (principal components) in which the data varies the most and project the data onto these new directions. These directions are orthogonal (perpendicular) to each other and capture the maximum variance in the dataset.

How PCA Works (Step-by-Step)

1. **Standardization**: Since PCA is affected by the scale of the features, it's important to standardize the dataset (mean = 0, variance = 1).

2. **Covariance Matrix Calculation**: The covariance matrix of the data is calculated to understand how the variables are related to each other.

3. **Eigenvalues and Eigenvectors**: The covariance matrix is decomposed into its **eigenvectors** and **eigenvalues**. Eigenvectors represent the directions of the new feature space, and eigenvalues represent the magnitude of variance along these directions.

4. **Principal Components**: The eigenvectors corresponding to the largest eigenvalues are selected as the principal components. These are the new axes onto which the data is projected.

5. **Projection**: The data is projected onto the selected principal components to obtain a lower-dimensional representation while retaining the maximum amount of variance.

PCA Example

Let's consider a simple example where we have a dataset with two features, `X1` and `X2`.

```
plaintext
X1  X2
2   4
3   5
4   6
5   7
6   8
```

Step 1: Standardization

Standardize the data so that both features have a mean of 0 and variance of 1.

```
plaintext
Standardized data:
X1'  X2'
-1.41  -1.41
-0.71  -0.71
0.00   0.00
0.71   0.71
1.41   1.41
```

Step 2: Covariance Matrix

Next, we calculate the covariance matrix of the standardized data to understand how the variables are related.

```
plaintext
Covariance Matrix:
[1  1]
[1  1]
```

Step 3: Eigenvectors and Eigenvalues

We compute the eigenvalues and eigenvectors of the covariance matrix. These tell us the directions and the amount of variance in those directions.

```
plaintext
Eigenvalues: 2, 0
Eigenvectors: [0.71, 0.71], [-0.71, 0.71]
```

Here, the first eigenvector `[0.71, 0.71]` corresponds to the larger eigenvalue of `2`, meaning this is the principal component that captures the most variance in the data.

Step 4: Projection

Finally, we project the data onto the new axis (the principal component).

```
plaintext
Projected Data:
PC1
-1.99
-0.99
0.00
0.99
1.99
```

The data has now been reduced to a single dimension (`PC1`), retaining most of the variance.

Eigenvalues and Eigenvectors in PCA

- **Eigenvalues**: Eigenvalues tell us how much variance there is along each principal component. A higher eigenvalue indicates that the principal component captures more variance in the data.
- **Eigenvectors**: Eigenvectors represent the directions (or axes) of the principal components. They are orthogonal to each other and point in the direction of maximum variance.

In PCA, the eigenvector with the largest eigenvalue is the first principal component, and it explains the most variance in the data. The eigenvector with the second largest eigenvalue is the second principal component, and so on.

Interpretation of PCA

- **Dimensionality Reduction**: PCA allows us to reduce the number of dimensions in the dataset by selecting only the top `k` principal components that capture most of the variance. For example, in a 3D dataset, you might reduce it to 2D by selecting the two components with the highest eigenvalues.

- **Variance Explained**: The proportion of the total variance explained by each principal component can be computed as the ratio of the corresponding

eigenvalue to the sum of all eigenvalues. This is often visualized using a **scree plot**, which shows the explained variance for each component.

Applications of PCA

1. **Data Compression**: PCA is used to reduce the size of datasets while preserving important patterns and structures.
2. **Noise Reduction**: By keeping only the principal components that capture significant variance, PCA helps filter out noise in the data.
3. **Visualization**: PCA is often used for visualizing high-dimensional data in two or three dimensions.

Conclusion

PCA is a powerful tool for reducing the dimensionality of large datasets while retaining essential information. It helps alleviate the "curse of dimensionality" and makes data easier to visualize and work with in machine learning models. The eigenvectors and eigenvalues computed during PCA provide the principal components and their importance in explaining the variance in the data.

In [1]:
```
import numpy as np
import pandas as pd
```

In [2]:
```
data = np.array([[3, 7],
                 [-4, -6],
                 [1, -1],
                 [7, 8],
                 [-4, -1],
                 [-3, -7]])
```

In [3]:
```
dataframe = pd.DataFrame(data, columns=['x1', 'x2'])
```

Standard Normal Form: If it contains mean as 0 and standard deviation as 1

In [4]:
```
dataframe
```

Out[4]:

	x1	x2
0	3	7
1	-4	-6
2	1	-1
3	7	8
4	-4	-1
5	-3	-7

In [5]:
```
dataframe.describe()
```

Out[5]:

	x1	x2
count	6.000000	6.000000
mean	0.000000	0.000000
std	4.472136	6.324555
min	-4.000000	-7.000000
25%	-3.750000	-4.750000
50%	-1.000000	-1.000000
75%	2.500000	5.000000
max	7.000000	8.000000

In [6]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
dataScaled = scaler.fit_transform(dataframe)
```

In [7]:
```
type(dataScaled)
```

Out[7]:
```
numpy.ndarray
```

In [8]:
```
dataframe = pd.DataFrame(dataScaled, columns=['x1', 'x2'])
```

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

Out[9]:

	x1	x2
count	6.000000e+00	6.000000
mean	1.850372e-17	0.000000
std	1.095445e+00	1.095445
min	-9.797959e-01	-1.212436
25%	-9.185587e-01	-0.822724
50%	-2.449490e-01	-0.173205
75%	6.123724e-01	0.866025
max	1.714643e+00	1.385641

1. To check whether the data is in standard normal form or not. As our data is in standard normal form, we are skipping that step of coverting our data to standard normal form.

2. Covariance Matrix between the two features that we have (x1, x2)

In [10]:
```
## Approach 1
c1 = dataframe.x1
c2 = dataframe.x2
np.cov(c1, c2)
```

Out[10]:
```
array([[1.2       , 1.06066017],
       [1.06066017, 1.2       ]])
```

In [11]:
```
dataframe.shape
```

Out[11]:
```
(6, 2)
```

In [12]:
```
## Approach 2
covarianceMatrix = dataframe.T @ dataframe / 5
covarianceMatrix
```

Out[12]:

	x1	x2
x1	1.20000	1.06066
x2	1.06066	1.20000

In [13]:
```
## Approach 3
np.sum(c1 * c2)/5
```

Out[13]:
```
1.0606601717798212
```

3. Evaluate the eigen values and eigen vectors of the above coorelation matrix

In [14]:
```
eigenValues, eigenVectors = np.linalg.eig(covarianceMatrix)
```

In [15]:
```
## eigenValues = # features
## eigenValue represents the strength of the information given by the eig
eigenValues
```

Out[15]:
```
array([2.26066017, 0.13933983])
```

In [16]:
```
eigenVectors
```

Out[16]:
```
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
```

In [17]:
```
eigenVectors[:, 0]
```

Out[17]:
```
array([0.70710678, 0.70710678])
```

In [18]:
```
dataframe
```

Out[18]:

	x1	x2
0	0.734847	1.212436
1	-0.979796	-1.039230
2	0.244949	-0.173205
3	1.714643	1.385641
4	-0.979796	-0.173205
5	-0.734847	-1.212436

In [19]:
```
## PC1 - Contains the maximum information of the original two features th
PC1 = dataframe @ eigenVectors[:, 0]
```

In [20]:
```
PC1
```

Out[20]:
```
0    1.376937
1   -1.427667
2    0.050731
3    2.192231
4   -0.815295
5   -1.376937
dtype: float64
```

In [21]:
```
## PC2
PC2 = dataframe @ eigenVectors[:, 1]
PC2
```

Out[21]:
```
0    0.337706
1   -0.042027
2   -0.295680
3   -0.232640
4    0.570346
5   -0.337706
dtype: float64
```

In [22]:
```
dataframe
```

Out[22]:

	x1	x2
0	0.734847	1.212436
1	-0.979796	-1.039230
2	0.244949	-0.173205
3	1.714643	1.385641
4	-0.979796	-0.173205
5	-0.734847	-1.212436

Applications of Dimensionality reduction

techniques, including Principal Component Analysis (PCA), t-SNE, and others, are widely used in real-world applications to simplify complex, high-dimensional datasets while preserving key information. These techniques make data analysis more efficient, improve model performance, and reduce computational costs. Below are some real-time applications of dimensionality reduction:

1. Image Compression

- **Application**: Dimensionality reduction techniques like PCA are used to reduce the size of image files by representing them with fewer components without losing significant detail.

- **Example**: JPEG compression uses PCA to reduce the dimensionality of color images, allowing for faster storage and transmission.

2. Facial Recognition

- **Application**: In facial recognition systems, dimensionality reduction is used to reduce the complexity of the high-dimensional image data while preserving the unique features necessary to distinguish between faces.
- **Example**: Eigenfaces and Fisherfaces are popular methods based on PCA that reduce image data to a lower-dimensional space for faster and more accurate face recognition.

3. Data Visualization

- **Application**: In cases where data has many dimensions (features), it is hard to visualize and analyze. Dimensionality reduction techniques help to project the data into 2D or 3D space, making it easier to explore and interpret.
- **Example**: t-SNE and PCA are frequently used in exploratory data analysis (EDA) to visualize complex datasets like genomic data, sensor data, or high-dimensional machine learning features.

4. Text Data Analysis and NLP

- **Application**: Natural Language Processing (NLP) applications often deal with very high-dimensional data, such as the bag-of-words or TF-IDF representations of text. Dimensionality reduction helps in reducing the complexity of such data while retaining essential semantic information.
- **Example**: In topic modeling, techniques like Latent Semantic Analysis (LSA) use dimensionality reduction to find patterns and relationships in the text corpus by reducing the number of features (words) to key topics.

5. Recommender Systems

- **Application**: Recommender systems rely on user-item interaction data, which can be extremely high-dimensional. Dimensionality reduction helps in reducing the number of latent features used to represent user preferences and item characteristics.
- **Example**: Matrix factorization techniques, like Singular Value Decomposition (SVD), are commonly used in collaborative filtering to predict user preferences based on lower-dimensional latent factors.

6. Genome Data Analysis

- **Application**: Genomic data contains thousands of features representing different genes. Dimensionality reduction is crucial for analyzing this data efficiently by reducing the number of features while preserving key biological information.

- **Example**: PCA and t-SNE are used in genomics to identify patterns in gene expression data, classify diseases, and discover subgroups of patients with similar genetic profiles.

7. Anomaly Detection

- **Application**: In anomaly detection, reducing the dimensionality of the data can help highlight key features that distinguish normal from abnormal behavior.
- **Example**: PCA is used in network security to reduce the dimensionality of network traffic data to identify abnormal patterns that may indicate cyber threats.

8. Sensor Data Analysis

- **Application**: IoT devices and sensors generate high-dimensional data streams, making it difficult to analyze in real-time. Dimensionality reduction techniques simplify the data while retaining important signals.
- **Example**: PCA is used in predictive maintenance systems to reduce the dimensionality of sensor data and detect equipment failures or operational anomalies in industrial environments.

9. Medical Imaging

- **Application**: In medical imaging, high-resolution images like MRIs and CT scans generate massive amounts of data. Dimensionality reduction helps in compressing these images and extracting essential features for diagnosis.
- **Example**: PCA and autoencoders are used to reduce the dimensionality of MRI or CT scan images, making them easier to process and analyze for disease detection, segmentation, or 3D reconstruction.

10. Speech Recognition

- **Application**: Speech data is high-dimensional and complex, especially when dealing with raw audio signals. Dimensionality reduction helps in extracting important features for speech recognition systems.
- **Example**: Techniques like PCA or Mel-frequency cepstral coefficients (MFCCs) are used to reduce the dimensionality of audio data, making it more manageable for speech-to-text algorithms.

11. Financial Market Analysis

- **Application**: Financial data often includes many correlated variables like stock prices, interest rates, and economic indicators. Dimensionality reduction is used to simplify this data and find patterns or trends.
- **Example**: PCA is used to reduce the number of correlated variables in financial data to create models that predict market movements, manage risks, or optimize portfolios.

12. **Customer Segmentation**

- **Application**: In marketing and customer relationship management, dimensionality reduction is used to group customers based on behaviors, preferences, or demographics, allowing businesses to create targeted marketing strategies.
- **Example**: PCA can reduce customer data features like purchase history and demographics into a few key components, making it easier to segment customers for personalized offers.

13. **Drug Discovery**

- **Application**: Dimensionality reduction is crucial in drug discovery, where datasets often consist of thousands of chemical compounds with numerous features.
- **Example**: Techniques like PCA and t-SNE are used to reduce the number of variables in chemical compound datasets, making it easier to identify promising candidates for drug development.

14. **Robotics**

- **Application**: Robots often operate in complex environments, generating high-dimensional sensor data. Dimensionality reduction helps robots navigate, recognize objects, and make decisions more efficiently.
- **Example**: Dimensionality reduction techniques are used in robotics for object recognition, reducing the complexity of sensor data inputs and improving decision-making algorithms.

Why Dimensionality Reduction Is Needed:

- **Improves Performance**: High-dimensional data can lead to overfitting in machine learning models. Reducing dimensionality simplifies the model, making it more generalizable.
- **Computational Efficiency**: High-dimensional data requires significant computational power. Reducing dimensions speeds up model training and prediction.
- **Removes Noise**: Dimensionality reduction helps filter out irrelevant features, making it easier to identify underlying patterns in the data.
- **Storage and Transmission**: In applications like image compression or sensor data analysis, dimensionality reduction reduces the data's size, making it easier to store and transmit.

In summary, dimensionality reduction is a critical step in making high-dimensional data more manageable, interpretable, and computationally feasible for real-world applications across various industries.

Implementation of Dimensionality reduction

Using SKlearn

In [24]:
```
import pandas as pd
import numpy as np
```

In [25]:
```
data = pd.read_csv('./diabetes.csv')
data.head()
```

Out[25]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPed
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	

1. Check the Standard Normal Form

In [26]:
```
## descriptive statistics of a given data
data.describe()
```

Out[26]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.0
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.9
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.8
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.3
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.0
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.6
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.1

In [27]:
```
## input features only
data = data.drop(columns=['Outcome'], axis=1)
```

In [28]:
```
data.head()
```

Out[28]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPed
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	

In [29]:
```
## convert the data into standard scaler form
## mean = 0 and standard deviation = 1
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
dataScaled = scaler.fit_transform(data)
```

In [30]:
```
dataScaled = pd.DataFrame(dataScaled)
dataScaled.head()
```

Out[30]:

	0	1	2	3	4	5	6	
0	0.639947	0.848324	0.149641	0.907270	-0.692891	0.204013	0.468492	
1	-0.844885	-1.123396	-0.160546	0.530902	-0.692891	-0.684422	-0.365061	-
2	1.233880	1.943724	-0.263941	-1.288212	-0.692891	-1.103255	0.604397	-
3	-0.844885	-0.998208	-0.160546	0.154533	0.123302	-0.494043	-0.920763	-
4	-1.141852	0.504055	-1.504687	0.907270	0.765836	1.409746	5.484909	-

In [31]:
```
dataScaled.describe()
```

Out[31]:

	0	1	2	3	4
count	7.680000e+02	7.680000e+02	7.680000e+02	7.680000e+02	7.680000e+02
mean	-6.476301e-17	-9.251859e-18	1.503427e-17	1.006140e-16	-3.006854e-17
std	1.000652e+00	1.000652e+00	1.000652e+00	1.000652e+00	1.000652e+00
min	-1.141852e+00	-3.783654e+00	-3.572597e+00	-1.288212e+00	-6.928906e-01
25%	-8.448851e-01	-6.852363e-01	-3.673367e-01	-1.288212e+00	-6.928906e-01
50%	-2.509521e-01	-1.218877e-01	1.496408e-01	1.545332e-01	-4.280622e-01
75%	6.399473e-01	6.057709e-01	5.632228e-01	7.190857e-01	4.120079e-01
max	3.906578e+00	2.444478e+00	2.734528e+00	4.921866e+00	6.652839e+00

In [32]:
```
dataScaled.shape
```

Out[32]:
```
(768, 8)
```

2. Covariance Matrix

In [33]:
```
covarianceMatrix = dataScaled.T @ dataScaled / 767
covarianceMatrix
```

Out[33]:

	0	1	2	3	4	5	6	
0	1.001304	0.129627	0.141466	-0.081778	-0.073630	0.017706	-0.033566	0.5
1	0.129627	1.001304	0.152789	0.057403	0.331789	0.221359	0.137516	0.2
2	0.141466	0.152789	1.001304	0.207641	0.089049	0.282173	0.041319	0.2
3	-0.081778	0.057403	0.207641	1.001304	0.437352	0.393085	0.184167	-0
4	-0.073630	0.331789	0.089049	0.437352	1.001304	0.198117	0.185312	-0.(
5	0.017706	0.221359	0.282173	0.393085	0.198117	1.001304	0.140830	0.(
6	-0.033566	0.137516	0.041319	0.184167	0.185312	0.140830	1.001304	0.(
7	0.545051	0.263858	0.239840	-0.114119	-0.042218	0.036289	0.033605	1.(

3. Evaluate the EigenValues and EigenVectors using the Covariance Matrix

In [34]:
```
eigenValues, eigenVectors = np.linalg.eig(covarianceMatrix)
```

In [35]:
```
## eigenValues = number of features
eigenValues
```

Out[35]:
```
array([2.09711056, 1.73346726, 0.42036353, 0.40498938, 0.68351839,
       0.76333832, 0.87667054, 1.03097228])
```

In [36]:
```
eigenVectors
```

Out[36]:
```
array([[-0.1284321 , -0.59378583, -0.58879003,  0.11784098, -0.19359817,
         0.47560573, -0.08069115,  0.01308692],
       [-0.39308257, -0.17402908, -0.06015291,  0.45035526, -0.09416176,
        -0.46632804,  0.40432871, -0.46792282],
       [-0.36000261, -0.18389207, -0.19211793, -0.01129554,  0.6341159 ,
        -0.32795306, -0.05598649,  0.53549442],
       [-0.43982428,  0.33196534,  0.28221253,  0.5662838 , -0.00958944,
         0.48786206, -0.03797608,  0.2376738 ],
       [-0.43502617,  0.25078106, -0.13200992, -0.54862138,  0.27065061,
         0.34693481,  0.34994376, -0.33670893],
       [-0.45194134,  0.1009598 , -0.03536644, -0.34151764, -0.68537218,
        -0.25320376, -0.05364595,  0.36186463],
       [-0.27061144,  0.122069  , -0.08609107, -0.00825873,  0.08578409,
        -0.11981049, -0.8336801 , -0.43318905],
       [-0.19802707, -0.62058853,  0.71208542, -0.21166198,  0.03335717,
         0.10928996, -0.0712006 , -0.07524755]])
```

In [37]:
```
## PC1 data
PC1_data = dataScaled @ eigenVectors[:, 0]
PC1_data
```

Out[37]:
```
0     -1.068503
1      1.121683
2      0.396477
3      1.115781
4     -2.359334
         ...
763   -1.562085
764    0.100405
765    0.283475
766    1.060324
767    0.839892
Length: 768, dtype: float64
```

In [38]:
```
## PC2 data
PC2_data = dataScaled @ eigenVectors[:, 1]
PC2_data
```

Out[38]:
```
0     -1.234895
1      0.733852
2     -1.595876
3      1.271241
4      2.184819
         ...
763   -1.923150
764    0.614181
765   -0.097065
766   -0.837062
767    1.151755
Length: 768, dtype: float64
```

In [39]:
```
## PC3 data
PC3_data = dataScaled @ eigenVectors[:, 7]
PC3_data
```

Out[39]:
```
0     -0.095930
1      0.712938
2     -1.760678
3      0.663729
4     -2.963107
         ...
763    0.867408
764    0.764353
765    0.077192
766   -0.425030
767    1.009178
Length: 768, dtype: float64
```

PCA Implementation via the sklearn library

In [40]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=3)
pca.fit_transform(dataScaled)
```

Out[40]:
```
array([[ 1.06850273,  1.23489499, -0.09592984],
       [-1.12168331, -0.73385167,  0.71293816],
       [-0.39647671,  1.59587594, -1.76067844],
       ...,
       [-0.28347525,  0.09706503,  0.07719194],
       [-1.06032431,  0.83706234, -0.42503045],
       [-0.83989172, -1.15175485,  1.00917817]])
```

In [41]:
```
from sklearn.decomposition import PCA
pca = PCA()
principalComponent = pca.fit_transform(dataScaled)
```

In [42]:
```
pca.explained_variance_ratio_
```

Out[42]:
```
array([0.26179749, 0.21640127, 0.12870373, 0.10944113, 0.09529305,
       0.08532855, 0.05247702, 0.05055776])
```

`pca.explained_variance_ratio_` is an attribute of the Principal Component Analysis (PCA) object in Python's `scikit-learn` library. It provides a measure of how much variance in the data is captured by each principal component after applying PCA. Specifically, it returns an array that represents the proportion (or percentage) of the total variance explained by each of the selected principal components.

Key Points:

- **Variance**: In the context of PCA, variance refers to how spread out the data is along the principal component axes.
- **Principal Components**: These are the new axes (or directions) in the data that maximize the variance after transforming the original features.
- **Explained Variance Ratio**: Each principal component explains a certain amount of the variance in the dataset. `pca.explained_variance_ratio_` tells you how much of the total variance is explained by each component.

Example:

Suppose you perform PCA on a dataset with 5 features, and you select 2 principal components. After running the PCA, you get the following result:

`pca.explained_variance_ratio_`
The output might look something like this:

`array([0.6, 0.3])`
This tells us:

- The first principal component explains 60% of the variance in the dataset.
- The second principal component explains 30% of the variance.

Together, the two components explain 90% of the total variance in the data.

Usefulness:

- **Dimensionality Reduction**: `explained_variance_ratio_` helps you decide how many components to keep by showing the variance explained by each. For example, you might decide to keep enough components to explain 95% of the variance.
- **Model Interpretation**: It helps to understand the importance of each principal component and how well they capture the underlying structure of the data.

In summary, `pca.explained_variance_ratio_` tells you how much of the variability in your data is captured by each of the principal components and helps in determining the optimal number of components to retain.

In [43]:
```
import matplotlib.pyplot as plt
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel("Dimensions")
plt.ylabel("Explained Variance Ratio")
plt.savefig("ScreenPlot.png")
plt.show()
```