Principal Component Analysis (PCA) in Python

With Code Examples

save for later

Introduction to Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a powerful dimensionality reduction technique used in data analysis and machine learning. It helps to identify patterns in high-dimensional data by transforming it into a new coordinate system where the axes represent the directions of maximum variance.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Generate sample data
np.random.seed(42)
X = np.random.randn(100, 2)
X = np.dot(X, [[2, 1], [1, 2]])  # Introduce correlation

# Perform PCA
pca = PCA()
X_pca = pca.fit_transform(X)

# Plot original data and principal components
plt.scatter(X[:, 0], X[:, 1], alpha=0.7)
for i, (comp, var) in enumerate(zip(pca.components_, pca.explained_variance_)):
    comp = comp * var  # Scale component by its variance explanation power
    plt.arrow(0, 0, comp[0], comp[1], color=f'C{i+1}', alpha=0.8, width=0.05)
    plt.text(comp[0], comp[1], f'PC{i+1}', color=f'C{i+1}')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('PCA: Original Data and Principal Components')
plt.axis('equal')
plt.show()
```

Swipe next ⟶

follow for more

Mathematical Foundation of PCA

PCA is based on the concept of eigenvectors and eigenvalues. It finds the directions (eigenvectors) in which the data varies the most, and these directions become the principal components. The amount of variance explained by each component is given by its corresponding eigenvalue.

```python
import numpy as np

# Generate sample data
np.random.seed(42)
X = np.random.randn(100, 3)

# Calculate covariance matrix
cov_matrix = np.cov(X.T)

# Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# Sort eigenvectors by decreasing eigenvalues
idx = eigenvalues.argsort()[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx]

print("Covariance Matrix:")
print(cov_matrix)
print("\nEigenvalues:")
print(eigenvalues)
print("\nEigenvectors:")
print(eigenvectors)
```

Swipe next ⟶

save for later

Implementing PCA from Scratch

Let's implement PCA step by step to understand its inner workings. We'll create a simple dataset, center it, compute the covariance matrix, and then find the principal components.

```python
import numpy as np

def pca_from_scratch(X, n_components):
    # Center the data
    X_centered = X - np.mean(X, axis=0)

    # Compute covariance matrix
    cov_matrix = np.cov(X_centered.T)

    # Compute eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

    # Sort eigenvectors by decreasing eigenvalues
    idx = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    # Select top n_components
    top_eigenvectors = eigenvectors[:, :n_components]

    # Project data onto principal components
    X_pca = X_centered.dot(top_eigenvectors)

    return X_pca, top_eigenvectors, eigenvalues

# Generate sample data
np.random.seed(42)
X = np.random.randn(100, 5)

# Apply PCA
X_pca, components, explained_variance = pca_from_scratch(X, n_components=2)

print("Transformed data shape:", X_pca.shape)
print("Principal components shape:", components.shape)
print("Explained variance:", explained_variance[:2])
```

Swipe next ⟶

Variance Explained and Choosing the Number of Components

One crucial aspect of PCA is determining how many principal components to retain. This decision is often based on the cumulative explained variance ratio, which tells us how much of the total variance in the data is captured by a given number of components.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Generate sample data
np.random.seed(42)
X = np.random.randn(100, 10)

# Perform PCA
pca = PCA()
pca.fit(X)

# Calculate cumulative explained variance ratio
cumulative_variance_ratio = np.cumsum(pca.explained_variance_ratio_)

# Plot cumulative explained variance ratio
plt.plot(range(1, len(cumulative_variance_ratio) + 1), cumulative_variance_ratio,
'bo-')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance Ratio')
plt.title('Explained Variance vs. Number of Components')
plt.grid(True)
plt.show()

# Find number of components for 95% variance explained
n_components_95 = np.argmax(cumulative_variance_ratio >= 0.95) + 1
print(f"Number of components for 95% variance explained: {n_components_95}")
```

Swipe next ⟶

save for later

PCA for Dimensionality Reduction

One of the primary applications of PCA is dimensionality reduction. By projecting high-dimensional data onto a lower-dimensional subspace, we can reduce the complexity of our dataset while retaining most of its important information.

```python
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Load the digits dataset
digits = load_digits()
X, y = digits.data, digits.target

# Perform PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Plot the results
plt.figure(figsize=(10, 8))
scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis', alpha=0.7)
plt.colorbar(scatter)
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.title('Digits Dataset Projected onto First Two Principal Components')
plt.show()

print(f"Original data shape: {X.shape}")
print(f"Reduced data shape: {X_pca.shape}")
```

Swipe next ⟶

follow for more

PCA for Data Visualization

PCA is an excellent tool for visualizing high-dimensional data in two or three dimensions. This can help us identify patterns, clusters, or outliers that might not be apparent in the original high-dimensional space.

```python
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Load the iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Perform PCA
pca = PCA(n_components=3)
X_pca = pca.fit_transform(X)

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

scatter = ax.scatter(X_pca[:, 0], X_pca[:, 1], X_pca[:, 2], c=y, cmap='viridis',
alpha=0.7)
ax.set_xlabel('First Principal Component')
ax.set_ylabel('Second Principal Component')
ax.set_zlabel('Third Principal Component')
plt.title('Iris Dataset Projected onto First Three Principal Components')
plt.colorbar(scatter)
plt.show()
```

Swipe next ⟶

save for later

PCA for Noise Reduction

PCA can be used for noise reduction in data by assuming that the principal components with the smallest variances correspond to noise. By reconstructing the data using only the top principal components, we can potentially remove some of the noise.

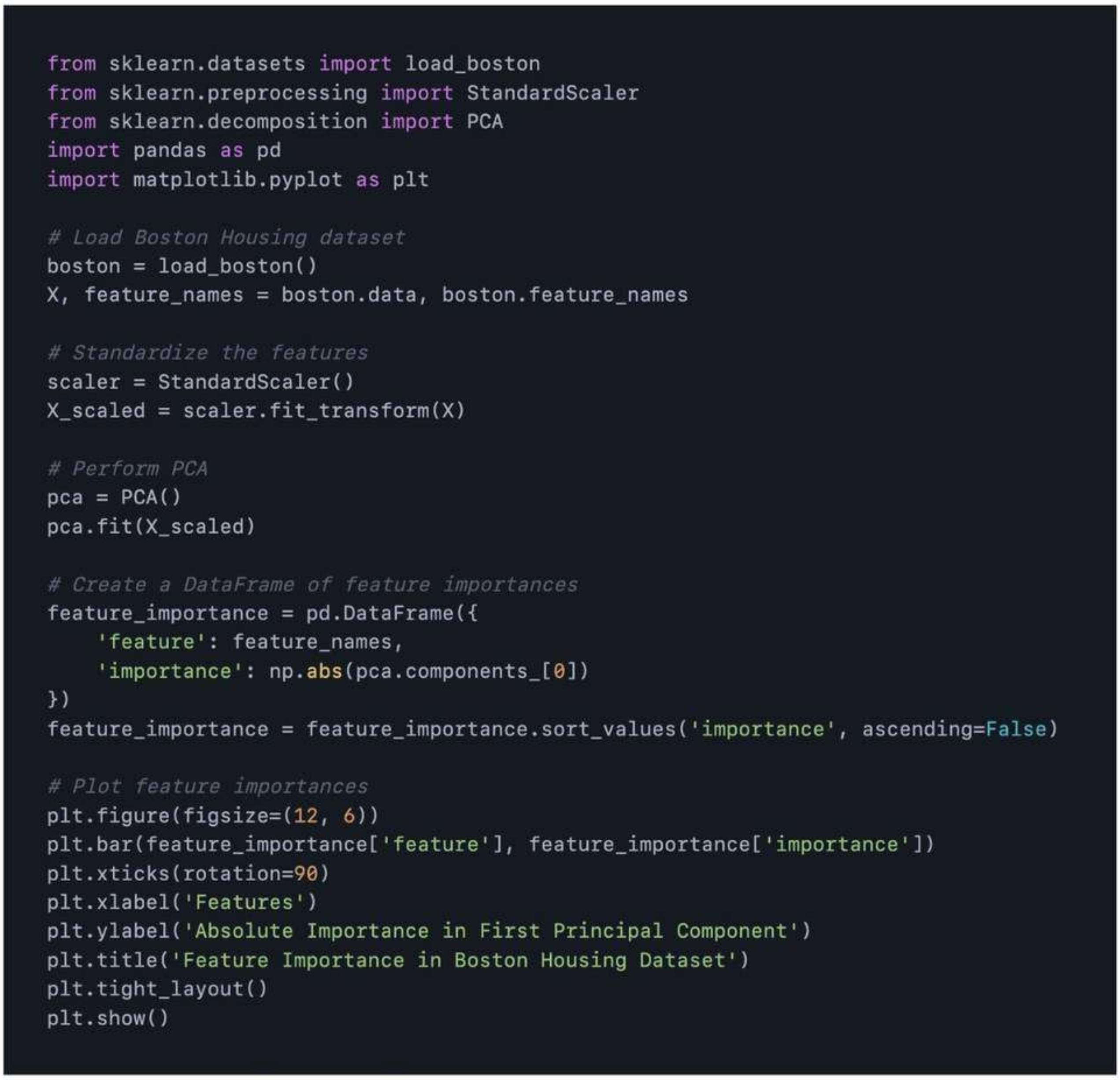
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Generate noisy sinusoidal data
np.random.seed(42)
t = np.linspace(0, 10, 1000)
x = np.sin(t) + 0.1 * np.random.randn(1000)

# Reshape data for PCA
X = np.column_stack((t, x))

# Perform PCA
pca = PCA(n_components=1)
X_pca = pca.fit_transform(X)
X_reconstructed = pca.inverse_transform(X_pca)

# Plot results
plt.figure(figsize=(12, 4))
plt.plot(t, x, 'b.', alpha=0.3, label='Noisy data')
plt.plot(X_reconstructed[:, 0], X_reconstructed[:, 1], 'r-', label='PCA
reconstruction')
plt.legend()
plt.title('PCA for Noise Reduction')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.show()
```

Swipe next ⟶

follow for more

PCA and Feature Importance

PCA can help us understand which original features contribute most to the principal components. This information can be valuable for feature selection and interpretation of our data.



```python
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import pandas as pd
import matplotlib.pyplot as plt

# Load Boston Housing dataset
boston = load_boston()
X, feature_names = boston.data, boston.feature_names

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Perform PCA
pca = PCA()
pca.fit(X_scaled)

# Create a DataFrame of feature importances
feature_importance = pd.DataFrame({
    'feature': feature_names,
    'importance': np.abs(pca.components_[0])
})
feature_importance = feature_importance.sort_values('importance', ascending=False)

# Plot feature importances
plt.figure(figsize=(12, 6))
plt.bar(feature_importance['feature'], feature_importance['importance'])
plt.xticks(rotation=90)
plt.xlabel('Features')
plt.ylabel('Absolute Importance in First Principal Component')
plt.title('Feature Importance in Boston Housing Dataset')
plt.tight_layout()
plt.show()
```

Swipe next ⟶

PCA and Outlier Detection

PCA can be used to detect outliers in multivariate data. By projecting the data onto the principal components and examining the reconstruction error, we can identify points that don't fit well with the overall structure of the data.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.covariance import EllipticEnvelope

# Generate sample data with outliers
np.random.seed(42)
X = np.random.randn(100, 2)
X = np.dot(X, [[2, 1], [1, 2]])
outliers = np.random.uniform(low=-10, high=10, size=(5, 2))
X = np.vstack([X, outliers])

# Perform PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Calculate reconstruction error
X_reconstructed = pca.inverse_transform(X_pca)
reconstruction_error = np.sum((X - X_reconstructed) ** 2, axis=1)

# Use Elliptic Envelope for comparison
ee = EllipticEnvelope(contamination=0.1, random_state=42)
outlier_labels = ee.fit_predict(X)

# Plot results
plt.figure(figsize=(12, 5))
plt.subplot(121)
plt.scatter(X[:, 0], X[:, 1], c=reconstruction_error, cmap='viridis')
plt.colorbar(label='Reconstruction Error')
plt.title('PCA Reconstruction Error')

plt.subplot(122)
plt.scatter(X[:, 0], X[:, 1], c=outlier_labels, cmap='viridis')
plt.title('Elliptic Envelope Outlier Detection')

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

Swipe next ⟶

follow for more

PCA for Time Series Analysis

PCA can be applied to time series data to identify underlying patterns or trends. This technique is particularly useful when dealing with multiple related time series.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Generate sample time series data
np.random.seed(42)
dates = pd.date_range(start='2023-01-01', end='2023-12-31', freq='D')
trend = np.linspace(0, 10, len(dates))
seasonality = 5 * np.sin(2 * np.pi * np.arange(len(dates)) / 365)
noise = np.random.randn(len(dates))

ts1 = trend + seasonality + noise
ts2 = 0.5 * trend + 2 * seasonality + noise
ts3 = -0.3 * trend + 0.5 * seasonality + noise

# Combine time series into a DataFrame
df = pd.DataFrame({'TS1': ts1, 'TS2': ts2, 'TS3': ts3}, index=dates)

# Perform PCA
pca = PCA()
components = pca.fit_transform(df)

# Plot results
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10))

df.plot(ax=ax1)
ax1.set_title('Original Time Series')

pd.DataFrame(components, index=dates, columns=['PC1', 'PC2', 'PC3']).plot(ax=ax2)
ax2.set_title('Principal Components')

plt.tight_layout()
plt.show()

print("Explained variance ratio:", pca.explained_variance_ratio_)
```

Swipe next ⟶

PCA for Anomaly Detection in Sensor Data

save for later

PCA can be used to detect anomalies in multivariate sensor data by identifying data points that deviate significantly from the principal components.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Generate synthetic sensor data
np.random.seed(42)
n_samples = 1000
n_sensors = 5

normal_data = np.random.randn(n_samples, n_sensors)
anomalies = np.random.uniform(low=-10, high=10, size=(10, n_sensors))
data = np.vstack([normal_data, anomalies])

# Perform PCA
pca = PCA(n_components=2)
pca_result = pca.fit_transform(data)

# Calculate reconstruction error
reconstructed = pca.inverse_transform(pca_result)
mse = np.mean(np.square(data - reconstructed), axis=1)

# Plot results
plt.figure(figsize=(12, 6))
plt.scatter(pca_result[:, 0], pca_result[:, 1], c=mse, cmap='viridis')
plt.colorbar(label='Reconstruction Error')
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.title('PCA for Anomaly Detection in Sensor Data')
plt.tight_layout()
plt.show()

# Identify potential anomalies
threshold = np.percentile(mse, 99)
anomalies = np.where(mse > threshold)[0]
print(f"Potential anomalies: {anomalies}")
```

Swipe next ⟶

follow for more

PCA in Image Recognition: Eigenfaces

PCA is used in facial recognition systems to create a set of eigenfaces, which are the principal components of a set of face images. These eigenfaces can be used to represent and recognize faces efficiently.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_lfw_people
from sklearn.decomposition import PCA

# Load face dataset
faces = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
X = faces.data
y = faces.target

# Perform PCA
n_components = 150
pca = PCA(n_components=n_components, whiten=True).fit(X)

# Plot the first few eigenfaces
fig, axes = plt.subplots(3, 5, figsize=(15, 9),
                         subplot_kw={'xticks':[], 'yticks':[]})
for i, ax in enumerate(axes.flat):
    ax.imshow(pca.components_[i].reshape(faces.images[0].shape),
              cmap='gray')
    ax.set_title(f'Eigenface {i+1}')
plt.tight_layout()
plt.show()

# Print explained variance ratio
print("Cumulative explained variance ratio:",
      np.sum(pca.explained_variance_ratio_))
```

Swipe next ⟶

save for later

Additional Resources

For those interested in diving deeper into Principal Component Analysis and its applications, here are some valuable resources:

1. ArXiv paper: "A Tutorial on Principal Component Analysis" by Jonathon Shlens URL: https://arxiv.org/abs/1404.1100
2. ArXiv paper: "Principal Component Analysis: A Review and Recent Developments" by Hervé Abdi and Lynne J. Williams URL: https://arxiv.org/abs/1404.1100
3. Scikit-learn documentation on PCA: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html
4. Coursera course: "Machine Learning" by Andrew Ng, which covers PCA in depth
5. Book: "Pattern Recognition and Machine Learning" by Christopher M. Bishop, which provides a thorough mathematical treatment of PCA

Data scientist
&ML Engineer

Follow For More Data Science Content