Understanding K-Means Clustering in Python

With Code Examples

save for later

Introduction to K-Means Clustering

K-Means clustering is an unsupervised machine learning algorithm used for partitioning a dataset into K distinct, non-overlapping subgroups (clusters) of observations. Each observation belongs to the cluster with the nearest mean (centroid). This algorithm is widely used for data exploration, customer segmentation, and image compression.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Generate sample data
np.random.seed(42)
X = np.random.rand(100, 2)

# Visualize the data
plt.scatter(X[:, 0], X[:, 1])
plt.title("Sample Data")
plt.show()
```

Swipe next →

follow for more

The K-Means Algorithm

The K-Means algorithm iteratively assigns data points to clusters and updates cluster centroids. It starts by randomly initializing K centroids, then alternates between assigning points to the nearest centroid and recalculating centroids based on the mean of assigned points. This process continues until convergence or a maximum number of iterations is reached.

```python
def kmeans(X, k, max_iters=100):
    centroids = X[np.random.choice(X.shape[0], k, replace=False)]

    for _ in range(max_iters):
        # Assign points to nearest centroid
        distances = np.sqrt(((X - centroids[:, np.newaxis])**2).sum(axis=2))
        labels = np.argmin(distances, axis=0)

        # Update centroids
        new_centroids = np.array([X[labels == i].mean(axis=0) for i in range(k)])

        # Check for convergence
        if np.all(centroids == new_centroids):
            break

        centroids = new_centroids

    return labels, centroids
```

Swipe next →

save for later

Choosing the Number of Clusters (K)

Selecting the optimal number of clusters is crucial for effective K-Means clustering. The elbow method is a common approach, which involves plotting the within-cluster sum of squares (WCSS) against the number of clusters. The "elbow" in the plot indicates a good choice for K.

```
from sklearn.cluster import KMeans

wcss = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, init='k-means++', random_state=42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)

plt.plot(range(1, 11), wcss)
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```

Swipe next →

save for later

Implementing K-Means with Scikit-learn

Scikit-learn provides an efficient implementation of the K-Means algorithm. We'll use it to cluster our sample data and visualize the results.

```
from sklearn.cluster import KMeans

# Create and fit the K-Means model
kmeans = KMeans(n_clusters=3, random_state=42)
labels = kmeans.fit_predict(X)

# Visualize the clusters
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='x', s=200, linewidths=3, color='r')
plt.title('K-Means Clustering')
plt.show()
```

Swipe next →

follow for more

Evaluating Cluster Quality

Silhouette analysis is a method to evaluate the quality of clusters. It measures how similar an object is to its own cluster compared to other clusters. The silhouette score ranges from -1 to 1, where higher values indicate better-defined clusters.

```
from sklearn.metrics import silhouette_score

# Calculate silhouette score
silhouette_avg = silhouette_score(X, labels)
print(f"The average silhouette score is: {silhouette_avg}")

# Visualize silhouette plot
from sklearn.metrics import silhouette_samples

silhouette_values = silhouette_samples(X, labels)
y_lower = 10
for i in range(3):
    ith_cluster_silhouette_values = silhouette_values[labels == i]
    ith_cluster_silhouette_values.sort()

    size_cluster_i = ith_cluster_silhouette_values.shape[0]
    y_upper = y_lower + size_cluster_i

    plt.fill_betweenx(np.arange(y_lower, y_upper),
                      0, ith_cluster_silhouette_values,
                      alpha=0.7)
    y_lower = y_upper + 10

plt.title("Silhouette Plot")
plt.xlabel("Silhouette coefficient values")
plt.ylabel("Cluster label")
plt.show()
```

Swipe next →

save for later

Handling Categorical Data

K-Means typically works with numerical data, but we can use techniques like one-hot encoding to handle categorical variables. Here's an example using pandas and scikit-learn:

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

# Sample data with mixed types
data = pd.DataFrame({
    'Age': [25, 30, 35, 40, 45],
    'Income': [50000, 60000, 70000, 80000, 90000],
    'Education': ['Bachelor', 'Master', 'PhD', 'Bachelor', 'Master']
})

# Create a ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', 'passthrough', ['Age', 'Income']),
        ('cat', OneHotEncoder(drop='first'), ['Education'])
    ])

# Fit and transform the data
X_encoded = preprocessor.fit_transform(data)

# Now you can use X_encoded with K-Means
kmeans = KMeans(n_clusters=2, random_state=42)
labels = kmeans.fit_predict(X_encoded)
```

Swipe next →

save for later

Dealing with High-Dimensional Data

K-Means can struggle with high-dimensional data due to the "curse of dimensionality". Dimensionality reduction techniques like Principal Component Analysis (PCA) can help mitigate this issue.

```
from sklearn.decomposition import PCA

# Generate high-dimensional data
X_high_dim = np.random.rand(100, 50)

# Apply PCA
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X_high_dim)

# Perform K-Means on reduced data
kmeans = KMeans(n_clusters=3, random_state=42)
labels = kmeans.fit_predict(X_reduced)

# Visualize results
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=labels, cmap='viridis')
plt.title('K-Means on PCA-reduced Data')
plt.show()
```

Swipe next →

Real-life Example: Customer Segmentation

K-Means clustering is widely used in marketing for customer segmentation. Let's segment customers based on their annual income and spending score.

```
import pandas as pd

# Load sample customer data
data = pd.read_csv('Mall_Customers.csv')
X = data[['Annual Income (k$)', 'Spending Score (1-100)']].values

# Perform K-Means clustering
kmeans = KMeans(n_clusters=5, random_state=42)
labels = kmeans.fit_predict(X)

# Visualize the segments
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='x', s=200, linewidths=3, color='r')
plt.title('Customer Segments')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.show()
```

Swipe next →

save for later

Real-life Example: Image Compression

K-Means can be used for image compression by reducing the number of colors in an image. Each cluster centroid represents a color in the compressed image.

```
from sklearn.cluster import KMeans
from PIL import Image
import numpy as np

# Load and preprocess the image
image = Image.open('sample_image.jpg')
image_array = np.array(image).reshape(-1, 3)

# Perform K-Means clustering
kmeans = KMeans(n_clusters=16, random_state=42)
labels = kmeans.fit_predict(image_array)

# Create the compressed image
compressed_image = kmeans.cluster_centers_[labels].reshape(image.size[1],
image.size[0], 3)
compressed_image = Image.fromarray(compressed_image.astype('uint8'))

# Display original and compressed images
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
ax1.imshow(image)
ax1.set_title('Original Image')
ax2.imshow(compressed_image)
ax2.set_title('Compressed Image (16 colors)')
plt.show()
```

Swipe next →

Limitations of K-Means

K-Means has several limitations: it assumes spherical clusters of similar size, is sensitive to initial centroid placement, and may converge to local optima. It also struggles with outliers and non-linear separable data.

```
# Generate non-spherical data
from sklearn.datasets import make_moons

X, _ = make_moons(n_samples=200, noise=0.05, random_state=42)

# Apply K-Means
kmeans = KMeans(n_clusters=2, random_state=42)
labels = kmeans.fit_predict(X)

# Visualize results
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.title('K-Means Limitation: Non-spherical Clusters')
plt.show()
```

Swipe next →

save for later

Addressing K-Means Limitations

To address some limitations of K-Means, we can use variations like K-Means++ for better initialization, or alternative algorithms like DBSCAN for non-spherical clusters.

```
from sklearn.cluster import KMeans, DBSCAN

# K-Means++
kmeans_plus = KMeans(n_clusters=2, init='k-means++', random_state=42)
labels_plus = kmeans_plus.fit_predict(X)

# DBSCAN
dbscan = DBSCAN(eps=0.3, min_samples=5)
labels_dbscan = dbscan.fit_predict(X)

# Visualize results
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
ax1.scatter(X[:, 0], X[:, 1], c=labels_plus, cmap='viridis')
ax1.set_title('K-Means++')
ax2.scatter(X[:, 0], X[:, 1], c=labels_dbscan, cmap='viridis')
ax2.set_title('DBSCAN')
plt.show()
```

Swipe next →

Optimizing K-Means Performance

For large datasets, we can use mini-batch K-Means to improve performance. This variant uses mini-batches to reduce computation time while still attempting to optimize the same objective function.

```python
from sklearn.cluster import MiniBatchKMeans
import time

# Generate a large dataset
X_large = np.random.rand(100000, 2)

# Standard K-Means
start_time = time.time()
kmeans = KMeans(n_clusters=5, random_state=42)
kmeans.fit(X_large)
print(f"Standard K-Means time: {time.time() - start_time:.2f} seconds")

# Mini-batch K-Means
start_time = time.time()
mbkmeans = MiniBatchKMeans(n_clusters=5, random_state=42)
mbkmeans.fit(X_large)
print(f"Mini-batch K-Means time: {time.time() - start_time:.2f} seconds")
```

Swipe next →

save for later

Interpreting K-Means Results

After performing K-Means clustering, it's crucial to interpret the results. We can examine cluster centroids and visualize data distribution within clusters to gain insights.

```
# Assuming we have performed K-Means on our data
kmeans = KMeans(n_clusters=3, random_state=42)
labels = kmeans.fit_predict(X)

# Examine cluster centroids
print("Cluster Centroids:")
print(kmeans.cluster_centers_)

# Visualize data distribution within clusters
for i in range(3):
    cluster_data = X[labels == i]
    plt.scatter(cluster_data[:, 0], cluster_data[:, 1], label=f'Cluster {i}')

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='x', s=200, linewidths=3, color='r', label='Centroids')
plt.legend()
plt.title('Data Distribution within Clusters')
plt.show()
```

Swipe next →

Additional Resources

For those interested in diving deeper into K-Means clustering and its applications, here are some valuable resources:

1. Arthur, D., & Vassilvitskii, S. (2007). k-means++: The Advantages of Careful Seeding. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (pp. 1027-1035). Society for Industrial and Applied Mathematics. ArXiv: https://arxiv.org/abs/2209.07373
2. Sculley, D. (2010). Web-scale k-means clustering. In Proceedings of the 19th international conference on World wide web (pp. 1177-1178). ArXiv: https://arxiv.org/abs/1012.0694
3. Jain, A. K. (2010). Data clustering: 50 years beyond K-means. Pattern recognition letters, 31(8), 651-666. ArXiv: https://arxiv.org/abs/1101.0901

Data scientist
&ML Engineer

Follow For More Data
Science Content