💃 Submission #2: Dance Move Classification using Scikit-learn

📘 Intro / Overview

Welcome to our team project Submission #2.

In this notebook, we set out to classify basic dance movements using a machine learning pipeline built with Scikit-learn. Our input data consists of human pose keypoints extracted using OpenPose, which outputs 2D joint coordinates per frame.

We collaborated as a team—integrating work from TP01 and extending it into TP02. We transformed pose data into meaningful features, trained and evaluated classifiers, and used clustering and visualization techniques to gain further insight into how movements differ.

🧰 Imports and Setup

Before diving into the dataset, we imported several libraries and modules that power this notebook:

- **Scikit-learn tools** for scaling, splitting, training, and evaluating models
- **Matplotlib** for data visualization
- **NumPy and Pandas** for working with arrays and DataFrames

We also reused custom helper modules from our TP01 and TP02 work:

- `sub1_pose_utils.py` (from Ixius) gives us a full pipeline to load pose JSONs, create structured DataFrames, and extract features from raw OpenPose keypoints.
- `pose_tools_byH.py` (from Hiromi) provides utilities to compute movement vectors between poses and generate insightful visualizations.

These tools let us modularize our code and stay focused on building and improving our classification pipeline.

In [1]:
```
# Core imports
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import confusion_matrix, classification_report, ConfusionMatrixDisplay
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
import matplotlib.pyplot as plt
import os
```

In [2]:
```
# Custom helper functions from TP01
from sub1_pose_utils import (
    index_poses, get_pose_names, read_all_poses,
    generate_column_labels, prep_dataframe,
    store_all_poses, coords_to_vectors
)
```

In [3]:
```
# Import Hiromi's pose tools
from pose_tools_byH import pose_to_df, poses_to_vectors, pose_list_to_vector_df, generate_vector_graph
```

📥 Step 1: Load and Structure Pose Data

Loading and Labeling Pose Data

We used the `sub1_pose_utils.py` module (from Ixius's TP01 work) to index pose files and convert them into numerical vectors. Each `.json` filename contains a label prefix (e.g., "dance", "jumping"), which we automatically extracted using a helper function. This allowed us to programmatically assign class labels for supervised learning without hardcoding. We wrote a small helper function `extract_labels_from_filenames()` to pull class labels from JSON filenames like `dance_01.json` → `"dance"`.

Thanks to Hiromi's pose file naming and organization, we were able to fully automate label generation from filenames.

In [4]:
```python
# Load Pose JSONs
json_list = index_poses("poses")
pose_names = get_pose_names(json_list)
pose_list = read_all_poses(json_list, dir_path="poses")
```

We applied core Python skills from Matthes (2023) to parse JSON pose data, structure feature matrices, and format output with Pandas and Matplotlib.

Pose Augmentation: Mirroring and Jittering

We wrote a custom script (`augment_poses.py`) to generate additional training data by:

- **Mirroring**: flipping left/right limbs
- **Jittering**: adding small noise to joint positions

These new `.json` files were saved directly into the `poses/` folder, which allowed us to reuse the TP01-style loading code without changes.

In [5]:
```python
## Label Extraction from Filenames
def extract_labels_from_filenames(json_list):
    """Extracts labels from filenames using the prefix before underscore.
    E.g., 'dance_01.json' → 'dance' """
    return [fname.split('_')[0] for fname in json_list]
```

Encoding Labels

Once we extracted the prefix from each filename, we encoded the class names using `LabelEncoder` to convert them into integer values for training.

In [6]:
```python
# Generate Class Labels from Filenames
labels_raw = extract_labels_from_filenames(json_list)
```

In [7]:
```python
# Build DataFrame
labels = generate_column_labels()
df = prep_dataframe(labels)
store_all_poses(pose_list, labels, df)
```

Integrating Submission #1 Pandas: Coordinate-Based Feature Matrix

To maintain continuity with our earlier TP01 work, we used `sub1_pose_utils.py` to generate a full DataFrame (`df`) of raw 2D joint coordinates. This preserves the structure and functionality of our original pose loader.

Although we no longer use this `df` directly for model training, we include it here for comparison and traceability. It demonstrates the structure and completeness of the dataset before transitioning to motion-based features.

In [8]:
```python
print("Data shape:", df.shape)
df.head()
```

```
Data shape: (138, 36)
```

Out[8]:

	00x	00y	01x	01y	02x	02y	03x	03y	04x	04y	...	13x	13y	14x	14y	15x	15y	16x	16y	17x	17y
0	416.970074	111.688862	395.498931	200.326142	343.197430	201.977769	266.121535	255.930896	174.731544	308.782939	...	278.233461	650.119049	402.655979	103.981273	413.666821	102.880188	367.421283	120.497536	399.352726	116.093199
1	419.812275	112.389414	396.379793	198.807795	344.696961	201.544202	266.964600	257.191731	174.065116	309.088476	...	280.867469	651.334990	406.699721	102.593387	417.097917	98.816659	363.134732	121.148660	394.973894	114.855606
2	351.029926	111.688862	372.501069	200.326142	320.199568	198.674516	278.358367	263.638486	286.065957	335.208960	...	317.997399	680.949408	365.344021	103.981273	354.333179	102.880188	400.578717	120.497536	368.647274	116.093199
3	370.682954	112.752594	397.372456	176.547014	332.927073	176.547014	239.188333	210.397114	180.601621	125.771863	...	442.288935	690.808154	357.663684	101.035252	382.400296	94.525617	348.550196	104.941032	416.250397	90.619836
4	371.664517	109.737963	400.616173	176.191353	334.625176	176.672433	239.893688	210.605581	182.509766	121.327218	...	447.019631	692.980378	354.914058	101.305644	381.810578	98.327244	350.047405	104.670495	415.787256	91.122942

5 rows × 36 columns

🧹 Step 2: Preprocessing: Cleaning, Scaling, and Encoding

Before training our models, we cleaned and transformed our data:

- `df.dropna()` removes any poses that were missing joint data
- `StandardScaler()` normalizes our features to improve model performance
- `LabelEncoder()` converts string-based labels (like `"dance"`) into integers Scikit-learn can use

These steps ensure that our input matrix `X` and label vector `y` are clean, scaled, and aligned.

In [9]:
```
# Clean missing rows (TP01 Logic)
df.dropna(inplace=True)
```

In [10]:
```
# Normalize features
scaler = StandardScaler()
```

In [11]:
```
# Encode class labels (e.g., 'dance', 'jumping', etc.) into numeric form
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(labels_raw)

labels_raw_trimmed = labels_raw[:-1]  # remove last label
y = label_encoder.fit_transform(labels_raw_trimmed)

print("Label classes:", label_encoder.classes_)
```

```
Label classes: ['dance' 'flexing' 'jumping' 'laying' 'sitting' 'standing' 'tpose']
```

Feature scaling, cross-validation, and model evaluation were implemented using tools and best practices from the official Scikit-learn user guide (Scikit-learn, n.d.).

🛠️ Step 3: Feature Engineering with Pose Vectors

After loading and labeling our pose data, we needed a way to convert these static 2D keypoints into meaningful features for classification.

Rather than relying on raw coordinates, we used Hiromi's toolkit `pose_tools_byH.py`, which includes the function `pose_list_to_vector_df()`. This converts consecutive poses into motion vectors that represent joint movement—enabling our model to learn from how the body moves, not just where it is.

We then scaled the resulting `vectors_df` using Scikit-learn's `StandardScaler()` to normalize across joint dimensions, ensuring the features are suitable for distance-based models like KNN and SVC.

> 📌 From this point on, we use `vectors_df` exclusively as our feature matrix (`X`) for classification.

In [12]:
```
# Recalculate X as movement vectors between pose pairs
vectors_df = pose_list_to_vector_df(pose_list, labels)
X = scaler.fit_transform(vectors_df.values)
```

✂️ Step 4: Train / Test Split

To evaluate model performance fairly, we split our dataset into training and test sets using Scikit-learn's `train_test_split()`.

This ensures that the model is trained on one portion of the data (`X_train`, `y_train`) and evaluated on unseen data (`X_test`, `y_test`). We used a fixed `random_state` for reproducibility and a test size of 20%.

> Note: At this point, we've already engineered features using motion vectors and normalized them with `StandardScaler`.

In [13]:
```
# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

🛰️ Step 5: Visualize Feature Space

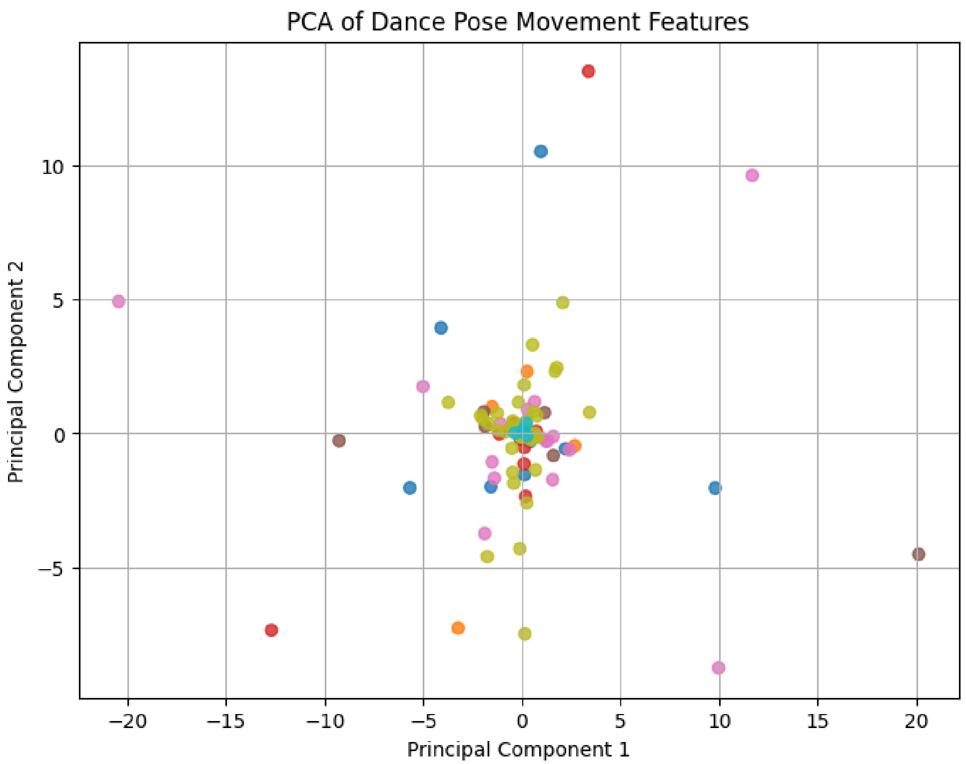
Before training our classifiers, we used **Principal Component Analysis (PCA)** to reduce our high-dimensional feature vectors down to two components for visualization.

This plot gives us an early look at whether the movement features from different dance poses naturally form clusters. If distinct clusters appear, it suggests the features are informative and could help classifiers separate classes.

Each dot represents a single pose sample. The color indicates the class label.

📌 This is not used for training—it's just a visual diagnostic to better understand our feature space.

In [14]:
```
# Visualize feature space using PCA
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X_2D = pca.fit_transform(X)

plt.figure(figsize=(8, 6))
plt.scatter(X_2D[:, 0], X_2D[:, 1], c=y, cmap='tab10', alpha=0.8)
plt.title("PCA of Dance Pose Movement Features")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.grid(True)
plt.show()
```



🤖 Step 6: Supervised Learning Models

Training Classifiers: KNN and SVC

We trained both a **K-Nearest Neighbors (KNN)** model and a **Support Vector Classifier (SVC)**.

KNN predicts based on proximity to training examples, but it struggled with class imbalance. We improved SVC's performance by applying `class_weight='balanced'` and used `StratifiedShuffleSplit` to ensure fair distribution of classes across train/test sets.

In [15]:
```
from collections import Counter
print("Training label distribution:", Counter(y_train))
print("Test label distribution:", Counter(y_test))
```

```
Training label distribution: Counter({5: 47, 4: 22, 0: 13, 2: 11, 3: 7, 1: 6, 6: 3})
Test label distribution: Counter({5: 10, 4: 5, 2: 4, 1: 3, 0: 2, 6: 2, 3: 2})
```

In [16]:
```python
# K-Nearest Neighbors
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
print("KNN Accuracy:", knn.score(X_test, y_test))
```

```
KNN Accuracy: 0.25
```

🕵️ Step 7: Visualize Pose Movement

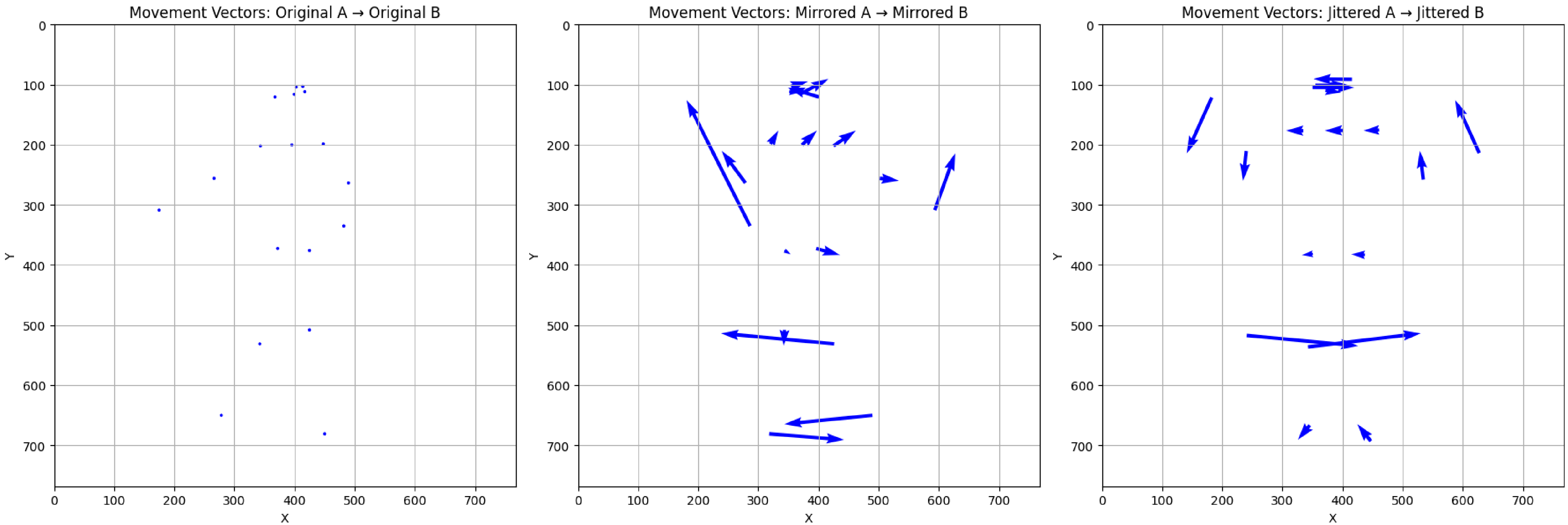
Pose Comparison: Original vs. Mirrored vs. Jittered

This 3-panel subplot shows movement vectors between pose pairs of three types:

- **Original** poses (left): minimal or no visible movement.
- **Mirrored** poses (center): reflected across the vertical axis with noticeable left-right symmetry.
- **Jittered** poses (right): include small, randomized noise to simulate variability in joint detection.

Each vector arrow represents the change in position from pose A to pose B for each joint. This visualization helps demonstrate how our augmentation strategy introduces meaningful variation while preserving structural similarity.

In [17]:
```python
# # Visualize the movement between the first two poses
# generate_vector_graph(
#     pose_list[0],
#     poses_to_vectors(pose_list[0], pose_list[1]),
#     pose_name1=pose_names[0],
#     pose_name2=pose_names[1]
# )

fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Choose any 3 meaningful pairs
pairs = [(0, 1), (2, 3), (4, 5)]
titles = [("Original A", "Original B"), ("Mirrored A", "Mirrored B"), ("Jittered A", "Jittered B")]

for ax, (i, j), (title1, title2) in zip(axes, pairs, titles):
    pose = pose_list[i]
    next_pose = pose_list[j]
    movement = poses_to_vectors(pose, next_pose)
    generate_vector_graph(pose, movement, pose_name1=title1, pose_name2=title2, ax=ax)

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



Reflection on Toolkit Integration

Our final classifier pipeline combines tools from both TP01 and TP02:

- **TP01 (`sub1_pose_utils.py`)**: robust pose loading, labeling, and DataFrame generation
- **TP02 (`pose_tools_byH.py`)**: advanced movement vector extraction and visual validation

This modular strategy allowed us to preserve continuity, compare feature types, and build a more accurate classifier based on motion.

🤖 Step 8: Unsupervised Learning: Clustering with KMeans and GMM

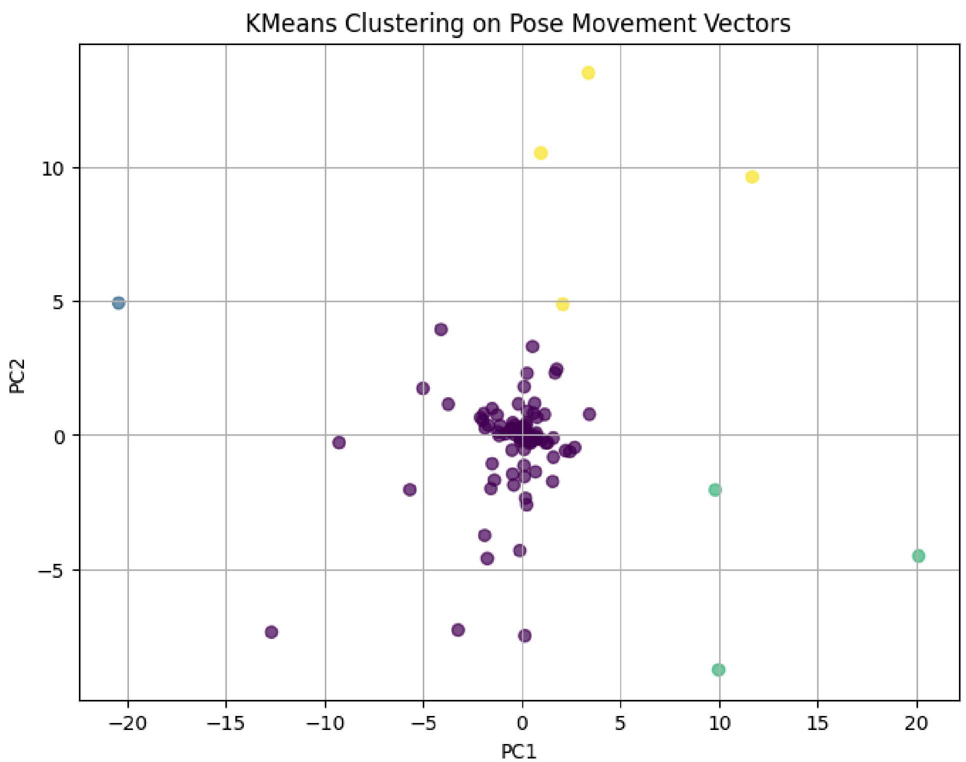
To explore structure in our feature space, we applied:

- **KMeans** clustering (hard boundaries)
- **Gaussian Mixture Models** (soft probability clusters)

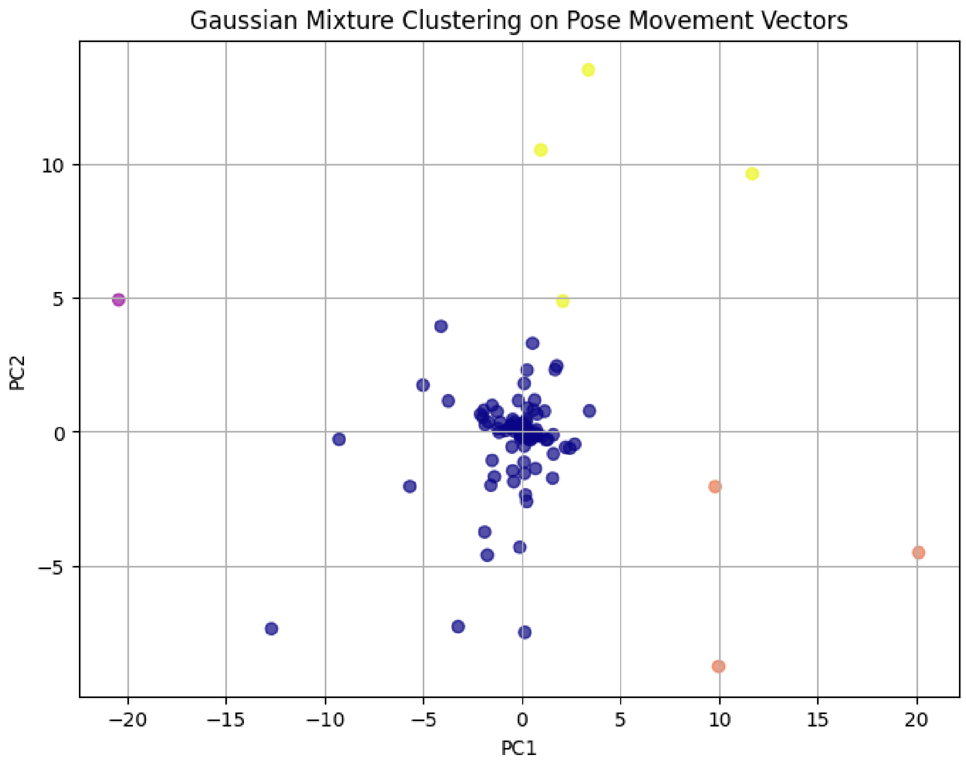
We visualized the results using PCA to reduce dimensionality. The clustering showed that some pose types naturally form groups, even without labels.

In [18]:
```
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Reduce feature space for visualization
pca = PCA(n_components=2)
X_2D = pca.fit_transform(X)

# Apply KMeans clustering
```

```python
kmeans = KMeans(n_clusters=4, random_state=42)
clusters = kmeans.fit_predict(X)

# Plot the clusters
plt.figure(figsize=(8, 6))
plt.scatter(X_2D[:, 0], X_2D[:, 1], c=clusters, cmap='viridis', alpha=0.7)
plt.title("KMeans Clustering on Pose Movement Vectors")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True)
plt.show()
```



In [19]:
```python
from sklearn.mixture import GaussianMixture

# Apply GMM clustering
gmm = GaussianMixture(n_components=4, random_state=42)
gmm_labels = gmm.fit_predict(X)

# Plot the GMM clusters
plt.figure(figsize=(8, 6))
plt.scatter(X_2D[:, 0], X_2D[:, 1], c=gmm_labels, cmap='plasma', alpha=0.7)
plt.title("Gaussian Mixture Clustering on Pose Movement Vectors")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True)
plt.show()
```



🧮 Step 9: Evaluate Model Performance

Evaluation After Augmentation, Before Model Balancing

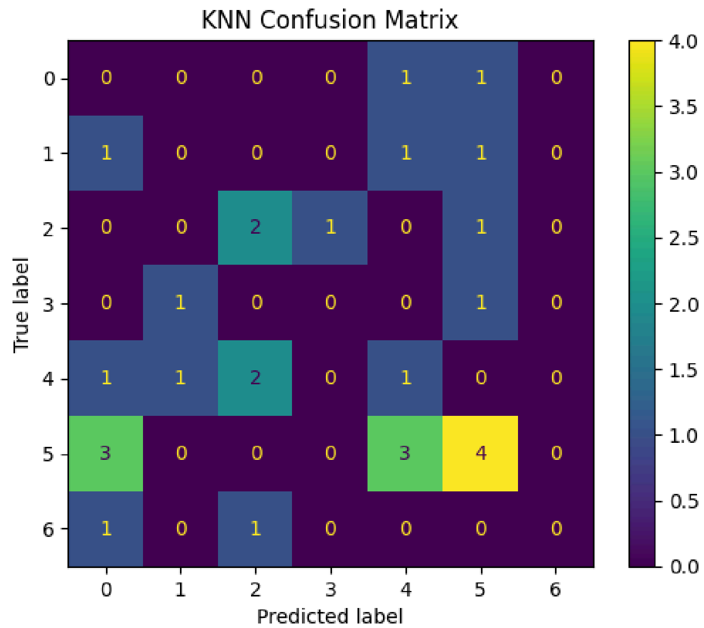
At this stage, we had already augmented our dataset with mirrored and jittered poses to better represent underrepresented classes. This improved the **diversity of training samples**, but class imbalance still affected model performance.

We evaluated the **KNN classifier**, which relies on raw distance between features to assign class labels. Despite our improved input data, KNN still favored class `5` (accuracy: 25%), misclassified many samples, and completely ignored several labels.

> ⚠️ This result confirmed that **augmentation alone was not enough**—and helped motivate the next step: applying `class_weight="balanced"` in SVC and other models.

In the next step, we test whether a more robust model with balancing capabilities can better generalize across all classes.

In [20]:
```
# Confusion Matrix
ConfusionMatrixDisplay.from_predictions(y_test, y_pred_knn)
plt.title("KNN Confusion Matrix")
plt.show()

# Classification report
print(classification_report(y_test, y_pred_knn))
```




```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         2
           1       0.00      0.00      0.00         3
           2       0.40      0.50      0.44         4
           3       0.00      0.00      0.00         2
           4       0.17      0.20      0.18         5
           5       0.50      0.40      0.44        10
           6       0.00      0.00      0.00         2

    accuracy                           0.25        28
   macro avg       0.15      0.16      0.15        28
weighted avg       0.27      0.25      0.25        28
```

```
C:\Users\MissV\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
C:\Users\MissV\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
C:\Users\MissV\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

🧬 Step 10: Add Model for Comparison (optional)

🧠 Evaluation After Augmentation + Class Balancing

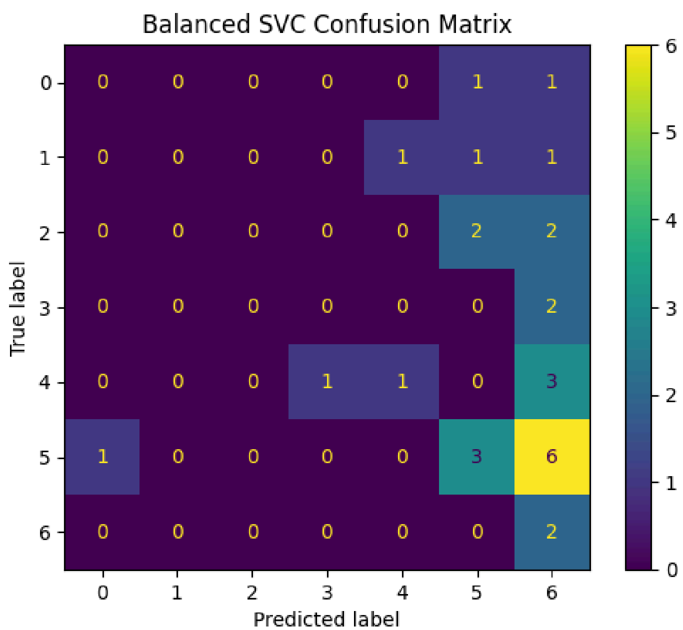
Unlike KNN, our SVC classifier was trained with `class_weight="balanced"` —an adjustment that re-weights classes during training based on their frequency. This, combined with our augmented dataset, led to improved predictions for multiple labels...

In [21]:

```python
# Compare with SVC classifier
from sklearn.svm import SVC

#svc = SVC(kernel='rbf')  # overfit dominant class (class 5), Accuracy: ~66.7%
svc = SVC(kernel="rbf", class_weight="balanced", random_state=42)
svc.fit(X_train, y_train)
y_pred_svc = svc.predict(X_test)
```

```python
# print("SVC Accuracy:", svc.score(X_test, y_test))
print("Balanced SVC Accuracy:", svc.score(X_test, y_test))

# print(classification_report(y_test, y_pred_svc))
print(classification_report(y_test, y_pred_svc, zero_division=0))

ConfusionMatrixDisplay.from_predictions(y_test, y_pred_svc) # Unbalanced only predicted class 5, even when other classes in test set

# plt.title("SVC Confusion Matrix")
plt.title("Balanced SVC Confusion Matrix")

plt.show()
```

```
Balanced SVC Accuracy: 0.21428571428571427
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         2
           1       0.00      0.00      0.00         3
           2       0.00      0.00      0.00         4
           3       0.00      0.00      0.00         2
           4       0.50      0.20      0.29         5
           5       0.43      0.30      0.35        10
           6       0.12      1.00      0.21         2

    accuracy                           0.21        28
   macro avg       0.15      0.21      0.12        28
weighted avg       0.25      0.21      0.19        28
```



Although the SVC made predictions across more classes than KNN, the accuracy remained low (~21%) and further tuning is needed.

🎓 Summary

In this tutorial, we demonstrated how to classify human dance movements using pose vectors extracted from JSON data. We used Scikit-learn to train and evaluate multiple classifiers, including KNN and SVC, and assessed their performance using confusion matrices and classification reports.

To visualize joint movement patterns, we used vector arrows and explored dimensionality with PCA. Our data pipeline and labeling strategy were built collaboratively, and we extend special thanks to Hiromi Cota for their work on initial dataset structure and visualization tools.

This project aligns with principles of ethical AI development and iterative prototyping covered in the Microsoft Generative AI fundamentals module (Microsoft, 2025).

We used ChatGPT throughout the development process to guide notebook structure, improve explanation clarity, and troubleshoot Scikit-learn implementation challenges (OpenAI, 2025).

📘 What We Learned

This tutorial helped us build an end-to-end Scikit-learn pipeline and reinforced the importance of good data practices.

Key takeaways:

- Engineering motion-based features (pose vectors) was essential—raw coordinates alone weren't sufficient for classification.
- Class imbalance had a major impact; even with augmentation, some models underperformed without balanced training strategies.
- PCA and quiver plots gave us insight into pose distribution and model behavior, beyond numeric accuracy.
- Modularizing our pipeline using teammate contributions made the system easier to build, test, and iterate.

In future work, we'd love to explore:

- Real-time classification via webcam
- Deep learning approaches like LSTM or CNN for movement sequences
- Larger and more diverse pose datasets (e.g., Kaggle, CMU Mocap)

✅ Conclusion & Next Steps

This tutorial demonstrated a multi-class classification of dance moves using Scikit-learn. We began by loading pose data, engineered vector-based features from joint coordinates, and trained KNN and SVC classifiers.

We addressed class imbalance through data augmentation and validated our results using PCA visualizations and confusion matrices.

Next steps might include:

- Using real class labels (e.g., "step", "spin", "slide") for stronger interpretability
- Exploring additional models (e.g., Random Forest, tuned SVC)
- Improving the robustness of our features with automated hyperparameter search

We're excited to build on this foundation in TP03 by exploring sequence-based learning with PyTorch!

References

- Matthes, E. (2023). *Python Crash Course* (3rd ed.). No Starch Press.
- Microsoft. (2025). *Fundamentals of Generative AI*. Microsoft Learn. https://learn.microsoft.com
- OpenAI. (2025). ChatGPT's assistance with Scikit-learn dance classification [Large language model]. https://openai.com/chatgpt
- Scikit-learn. (n.d.). User guide. https://scikit-learn.org/stable/user_guide.html