Demo 3 - CS 156

Shervan Shahparnia & Sharon Le

Step 1 - Feature Engineering

Import Dependencies

In [4]:
```
# === Data Handling ===
import pandas as pd                      # For reading CSVs and manipulating DataFra
import numpy as np                      # For efficient numerical computations

# === Time-Domain Statistical Features ===
from scipy.stats import skew, kurtosis  # For computing skewness and kurtosis of si

# === Frequency-Domain Feature Extraction ===
from scipy.fftpack import fft           # Fast Fourier Transform (FFT) for frequenc
from scipy.signal import welch, find_peaks  # Welch's method for power spectral den

# === Cross-Validation Tools ===
from sklearn.model_selection import KFold, LeaveOneGroupOut
# KFold: 10-fold cross-validation
# LeaveOneGroupOut: cross-validation leaving one subject (group) out

# === Preprocessing ===
from sklearn.preprocessing import StandardScaler  # For scaling features to standar

# === Evaluation Metrics ===
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
# Accuracy, F1, Precision, Recall: metrics used to evaluate classifier performance

# === Machine Learning Models ===
from sklearn.ensemble import RandomForestClassifier  # Random Forest - tree-based e
from sklearn.neighbors import KNeighborsClassifier   # KNN - classification based o
from sklearn.tree import DecisionTreeClassifier      # Decision Tree - simple inter
from sklearn.svm import SVC                          # Support Vector Classifier -

# === Signal Processing Helpers ===
from scipy.signal import correlate, find_peaks       # Cross-correlation and peak d
```

Combine the 22 subjects into a single dataset with a unique 'ID' column to identify them

In [5]:
```
# Path to the folder containing the 22 CSV files (update with actual path)
path = r"C:\Users\sherv\Desktop\SP25\CS156-main\demo3\harth_data"  # Modify the pat

# Using glob to find all CSV files in the directory
```

```
import glob
all_files = glob.glob(path + "/*.csv")

# List to hold dataframes
df_list = []

# Initialize the id value
current_id = 1

# Loop through each file and read it into a dataframe
for file in all_files:
    # Read the current CSV file
    df = pd.read_csv(file)

    # Add the 'id' column to store the subject identifier (current_id)
    df['id'] = current_id

    # Append the dataframe to the list
    df_list.append(df)

    # Increment the ID for the next file
    current_id += 1
# Concatenate all dataframes into a single DataFrame
combined_df = pd.concat(df_list, ignore_index=True)
combined_df = combined_df.drop(columns=['index'], errors='ignore')

# Display the first few rows of the combined dataset
combined_df.head()

# Define the path where the new CSV file will be saved (you can modify this path)
output_path = r"C:\Users\sherv\Desktop\SP25\CS156-main\demo2\new_data"

# Save the combined DataFrame to a CSV file
combined_df.to_csv(output_path, index=False)
combined_df = combined_df.loc[:, ~combined_df.columns.str.contains('^Unnamed')]


# Confirm that the file has been saved
print(f"Combined dataset saved to {output_path}")
```

```
Combined dataset saved to C:\Users\sherv\Desktop\SP25\CS156-main\demo2\new_data
```

Load Combined Dataset

In [6]:
```
# Load combined dataset
df = pd.read_csv(r"C:\Users\sherv\Desktop\SP25\CS156-main\demo3\new_data")
```

Define Sensor Axes and Setup Feature Storage

In [7]:
```
# All signal columns (from your HARTH dataset)
axes = ["back_x", "back_y", "back_z", "thigh_x", "thigh_y", "thigh_z"]

# Store per-subject feature results
subject_stats = []
```

Define Time-Domain Feature Function

In [8]:
```
# Define time-domain feature computation
def compute_features(signal):
    features = {
        "mean": np.mean(signal),                                    # Average value
        "std": np.std(signal),                                      # Standard deviation
        "var": np.var(signal),                                      # Variance
        "min": np.min(signal),                                      # Minimum value
        "max": np.max(signal),                                      # Maximum value
        "range": np.ptp(signal),                                    # Range (max - min)
        "median": np.median(signal),                                # Median value
        "iqr": np.percentile(signal, 75) - np.percentile(signal, 25),  # Interquart
        "rms": np.sqrt(np.mean(np.square(signal))),                 # Root Mean Square
        "zcr": ((signal[:-1] * signal[1:]) < 0).sum(),               # Zero-crossing rate
        "skew": skew(signal),                                       # Asymmetry of distri
        "kurtosis": kurtosis(signal),                               # Tailedness of distr
        "energy": np.sum(np.square(signal)),                        # Sum of squared valu
        "autocorr": np.corrcoef(signal[:-1], signal[1:])[0, 1] if len(signal) > 1 e
        "peak_count": len(find_peaks(signal)[0])                    # Number of signal pe
    }
    return features
```

Loop Over Subjects and Extract Features

In [9]:
```
# Loop through each subject
for subject_id in df["id"].unique():
    row = {"id": subject_id}
    sub_df = df[df["id"] == subject_id]

    # Compute Signal Magnitude Area (SMA) for back and thigh
    back_sma = np.mean(np.abs(sub_df["back_x"]) + np.abs(sub_df["back_y"]) + np.abs
    thigh_sma = np.mean(np.abs(sub_df["thigh_x"]) + np.abs(sub_df["thigh_y"]) + np.
    row["back_sma"] = back_sma
    row["thigh_sma"] = thigh_sma

    # Compute time-domain stats for each axis
    for axis in axes:
        stats = compute_features(sub_df[axis].values)
        for stat_name, value in stats.items():
            row[f"{axis}_{stat_name}"] = value

    subject_stats.append(row)
```

Convert List of Feature Dicts to DataFrame

In [10]:
```
# Convert to DataFrame
subject_stats_df = pd.DataFrame(subject_stats)
```

Compute Mean of Each Feature Across Subjects

In [11]:
```
# Compute overall stats (averaged over all subjects)
overall_stats = subject_stats_df.mean(numeric_only=True).to_frame(name="Overall_Mea
overall_stats.columns = ["Feature", "Overall_Mean"]
```

Display and Save Results of Feature Engineering

In [12]:
```
# Show the first few rows of each table in Jupyter Notebook
print(":bar_chart: Per-Subject Feature Statistics:")
display(subject_stats_df.head())

print("\n:bar_chart: Overall Feature Averages:")
display(overall_stats.head())

# Optionally save them as CSV files for Excel or Google Sheets
subject_stats_df.to_csv("subject_feature_statistics.csv", index=False)
overall_stats.to_csv("overall_feature_averages.csv", index=False)
```

```
:bar_chart: Per-Subject Feature Statistics:
```

	id	back_sma	thigh_sma	back_x_mean	back_x_std	back_x_var	back_x_min	back_x_max
0	1	1.326687	1.351477	-0.802201	0.238346	0.056809	-3.542889	0.952109
1	2	1.379790	1.319124	-0.920351	0.130877	0.017129	-3.066853	0.873471
2	3	1.391304	1.965097	-0.944405	0.199062	0.039626	-5.238408	0.858455
3	4	1.198823	1.450655	-1.019898	0.186093	0.034631	-2.365137	-0.309204
4	5	1.191779	1.340398	-0.915515	0.242905	0.059003	-3.810360	0.511867

5 rows × 93 columns

```
:bar_chart: Overall Feature Averages:
```

	Feature	Overall_Mean
0	id	11.000000
1	back_sma	1.359009
2	thigh_sma	1.647008
3	back_x_mean	-0.890122
4	back_x_std	0.358888

Time & Frequency-Domain Feature Functions for Signal Windows

In [13]:
```
def extract_time_features(window):
    return {
        "mean": np.mean(window),                                    # Average value
        "std": np.std(window),                                      # Standard devi
```

```
        "var": np.var(window),                                          # Variance
        "min": np.min(window),                                          # Minimum value
        "max": np.max(window),                                          # Maximum value
        "range": np.ptp(window),                                        # Peak-to-peak
        "median": np.median(window),                                    # Median value
        "iqr": np.percentile(window, 75) - np.percentile(window, 25),  # Interquart
        "rms": np.sqrt(np.mean(np.square(window))),                     # Root Mean Squ
        "zcr": ((window[:-1] * window[1:]) < 0).sum(),                  # Zero-crossing
        "skew": skew(window),                                           # Measure of as
        "kurtosis": kurtosis(window),                                   # Tailedness or
        "energy": np.sum(np.square(window)),                            # Total energy
        "peak_count": len(find_peaks(window)[0])                        # Number of pea
    }
```

In [14]:

```
def extract_freq_features(window, sampling_rate=50):
    fft_vals = fft(window)                                              # Compute FFT (
    fft_mag = np.abs(fft_vals)[:len(window)//2]                         # Take magnitua
    freqs = np.fft.fftfreq(len(window), d=1/sampling_rate)[:len(window)//2]  # Freq

    spectral_centroid = np.sum(freqs * fft_mag) / np.sum(fft_mag)  # Weighted avera
    spectral_entropy = -np.sum((fft_mag/np.sum(fft_mag)) * np.log2(fft_mag/np.sum(f
    spectral_energy = np.sum(fft_mag ** 2)                              # Total power i
    dominant_freq = freqs[np.argmax(fft_mag)]                          # Frequency with
    freq_variance = np.var(fft_mag)                                    # Variance in fr
    spectral_flatness = np.exp(np.mean(np.log(fft_mag + 1e-10))) / (np.mean(fft_mag
    bandwidth = np.max(freqs) - np.min(freqs)                          # Spread of freq
    psd = welch(window, fs=sampling_rate, nperseg=len(window))[1]  # Power Spectral

    return {
        "spectral_centroid": spectral_centroid,
        "spectral_entropy": spectral_entropy,
        "spectral_energy": spectral_energy,
        "dominant_freq": dominant_freq,
        "freq_variance": freq_variance,
        "spectral_flatness": spectral_flatness,
        "bandwidth": bandwidth,
        "psd_mean": np.mean(psd)
    }
```

In [15]:

```
def extract_features(df, window_size=100, step_size=50):
    X, y, groups = [], [], []  # X = features, y = labels, groups = subject IDs

    # Loop over each subject in the dataset
    for subject_id in df["id"].unique():
        sub_df = df[df["id"] == subject_id]  # Get data for the current subject

        # Apply a sliding window over the subject's data
        for i in range(0, len(sub_df) - window_size, step_size):
            window = sub_df.iloc[i:i+window_size]  # Get the i-th window of data

            # Get the most frequent activity label in the window
            label = window["label"].mode()[0]

            combined_features = {}
```

```
            # Loop through each motion axis and extract features
            for axis in ["back_x", "back_y", "back_z", "thigh_x", "thigh_y", "thigh
                td = extract_time_features(window[axis].values)  # Time-domain feat
                fd = extract_freq_features(window[axis].values)  # Frequency-domain

                # Combine features with axis-specific prefixes
                combined_features.update({f"{axis}_{k}": v for k, v in {**td, **fd}

            # Append the features, label, and subject ID
            X.append(combined_features)
            y.append(label)
            groups.append(subject_id)

    # Return feature matrix, labels, and group identifiers as numpy/pandas
    return pd.DataFrame(X), np.array(y), np.array(groups)
```

Step 2 - Model Development

Sample Data & Extract Features

In [16]:
```
# Load the full dataset
df = pd.read_csv(r"C:\Users\sherv\Desktop\SP25\CS156-main\demo3\new_data")

# Group by 'id' and sample 50,000 rows per subject without replacement
df_sampled = df.groupby("id", group_keys=False).apply(lambda x: x.sample(n=50000, r

# Shuffle the entire sampled DataFrame
df_sampled = df_sampled.sample(frac=1, random_state=42).reset_index(drop=True)

# Extract time + frequency domain features using sliding windows
X, y, groups = extract_features(df_sampled)
```

Inspect Extracted Features

In [17]:
```
# Check shape of feature matrix and data types of each column
print(X.shape)      # Rows = number of windows, Cols = number of extracted features
print(X.dtypes)     # Verify all features are numeric and properly structured
```

```
(20958, 132)
back_x_mean                 float64
back_x_std                  float64
back_x_var                  float64
back_x_min                  float64
back_x_max                  float64
                             ...
thigh_z_dominant_freq       float64
thigh_z_freq_variance       float64
thigh_z_spectral_flatness   float64
thigh_z_bandwidth           float64
thigh_z_psd_mean            float64
Length: 132, dtype: object
```

Originally checked the shape of the feature matrix because there were no parameters for random forest to work with, so to clarify I checked. Then realized that the sampled dataframe had ended up empty due to improper filtering of the data.

Feature Selection Using Random Forest Importance

In [18]:
```
# Train a Random Forest model on the full extracted feature set
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X, y)  # Fit the model to your features and labels

# Get feature importance scores and convert to a Pandas Series
importances = pd.Series(rf.feature_importances_, index=X.columns)

# Select the top 30 most important features based on those scores
top_features = importances.nlargest(30).index

# Filter the feature matrix to keep only the top 30 selected features
X_selected = X[top_features]
```

Define Classification Models for Comparison

In [19]:
```
# Define a dictionary of classification models to compare
models = {
    "Random Forest": RandomForestClassifier(n_estimators=100),  # Tree-based ensemb
    "KNN": KNeighborsClassifier(n_neighbors=5),                 # K-Nearest Neighbo
    "Decision Tree": DecisionTreeClassifier(),                  # Single tree model
    "SVM": SVC()                                                # Support Vector Ma
}
```

Step 3 - Evaluation

10-Fold Cross-Validation for Model Evaluation

In [20]:
```
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score

print("\n--- 10-Fold Cross-Validation Results ---")

# Set up 10-fold cross-validation
kf = KFold(n_splits=10, shuffle=True, random_state=42)

# Loop through each model
for name, model in models.items():
    acc, f1, prec, rec = [], [], [], []  # Store metrics for each fold

    # Perform 10-fold splitting
    for train_idx, test_idx in kf.split(X_selected):
        model.fit(X_selected.iloc[train_idx], y[train_idx])             # Train on
        preds = model.predict(X_selected.iloc[test_idx])               # Predict on
```

```
        # Calculate evaluation metrics
        acc.append(accuracy_score(y[test_idx], preds))
        f1.append(f1_score(y[test_idx], preds, average='weighted', zero_division=0)
        prec.append(precision_score(y[test_idx], preds, average='weighted', zero_di
        rec.append(recall_score(y[test_idx], preds, average='weighted', zero_divisi

    # Print average metrics across the 10 folds
    print(f"{name} - Acc: {np.mean(acc):.4f}, F1: {np.mean(f1):.4f}, Prec: {np.mean
```

```
--- 10-Fold Cross-Validation Results ---
Random Forest - Acc: 0.9914, F1: 0.9912, Prec: 0.9915, Rec: 0.9914
KNN - Acc: 0.9646, F1: 0.9645, Prec: 0.9647, Rec: 0.9646
Decision Tree - Acc: 0.9871, F1: 0.9870, Prec: 0.9871, Rec: 0.9871
SVM - Acc: 0.8663, F1: 0.8270, Prec: 0.8016, Rec: 0.8663
```

Leave-One-Subject-Out (LOSO) Cross-Validation

In [21]:
```
print("\n--- Leave-One-Subject-Out Results ---")

# Use Leave-One-Group-Out cross-validation (subject ID as group)
logo = LeaveOneGroupOut()

# Loop through each classification model
for name, model in models.items():
    acc, f1, prec, rec = [], [], [], []  # Store metrics per subject

    # Leave one subject (group) out each round
    for train_idx, test_idx in logo.split(X_selected, y, groups):
        model.fit(X_selected.iloc[train_idx], y[train_idx])        # Train on all b
        preds = model.predict(X_selected.iloc[test_idx])           # Test on the he

        # Compute evaluation metrics for this round
        acc.append(accuracy_score(y[test_idx], preds))
        f1.append(f1_score(y[test_idx], preds, average='weighted', zero_division=0)
        prec.append(precision_score(y[test_idx], preds, average='weighted', zero_di
        rec.append(recall_score(y[test_idx], preds, average='weighted', zero_divisi

    # Print average scores across all subjects
    print(f"{name} - Acc: {np.mean(acc):.4f}, F1: {np.mean(f1):.4f}, Prec: {np.mean
```

```
--- Leave-One-Subject-Out Results ---
Random Forest - Acc: 0.7975, F1: 0.7994, Prec: 0.8882, Rec: 0.7975
KNN - Acc: 0.8174, F1: 0.8486, Prec: 0.9375, Rec: 0.8174
Decision Tree - Acc: 0.8132, F1: 0.8234, Prec: 0.9359, Rec: 0.8132
SVM - Acc: 0.7453, F1: 0.7613, Prec: 0.8406, Rec: 0.7453
```

Combined 10-Fold and LOSO Evaluation Results

In [22]:
```
# Store evaluation results
results = []

# Loop through both cross-validation strategies
for eval_name, splitter in [("10-Fold CV", KFold(n_splits=10, shuffle=True, random_
                            ("LOSO", LeaveOneGroupOut())]:
```

```
    # Loop through each model
    for name, model in models.items():
        acc, f1, prec, rec = [], [], [], []

        # Perform split: use subject groupings only for LOSO
        for train_idx, test_idx in splitter.split(X_selected, y, groups if eval_nam
            model.fit(X_selected.iloc[train_idx], y[train_idx])         # Train moa
            preds = model.predict(X_selected.iloc[test_idx])           # Predict on

            # Collect evaluation metrics
            acc.append(accuracy_score(y[test_idx], preds))
            f1.append(f1_score(y[test_idx], preds, average='weighted', zero_divisio
            prec.append(precision_score(y[test_idx], preds, average='weighted', zer
            rec.append(recall_score(y[test_idx], preds, average='weighted', zero_di

        # Store the average performance for this model and evaluation type
        results.append({
            "Model": name,
            "Evaluation": eval_name,
            "Accuracy": np.mean(acc),
            "F1 Score": np.mean(f1),
            "Precision": np.mean(prec),
            "Recall": np.mean(rec)
        })

# Create results DataFrame
results_df = pd.DataFrame(results)
print(results_df)
```

```
           Model  Evaluation  Accuracy  F1 Score  Precision    Recall
0  Random Forest  10-Fold CV  0.991364  0.991238   0.991504  0.991364
1            KNN  10-Fold CV  0.964644  0.964494   0.964664  0.964644
2  Decision Tree  10-Fold CV  0.987022  0.986996   0.987023  0.987022
3            SVM  10-Fold CV  0.866304  0.827003   0.801638  0.866304
4  Random Forest        LOSO  0.822932  0.828555   0.840683  0.822932
5            KNN        LOSO  0.817397  0.848573   0.937484  0.817397
6  Decision Tree        LOSO  0.813246  0.822702   0.888358  0.813246
7            SVM        LOSO  0.745252  0.761341   0.840604  0.745252
```

Model Performance Summary Random Forest

- Best overall performer
- ~99% in 10-Fold CV, ~82% in LOSO
- Strong generalization to new subjects

KNN

- High precision (~94%) in LOSO
- Best F1 score in LOSO
- Slightly weaker than RF but still strong

Decision Tree

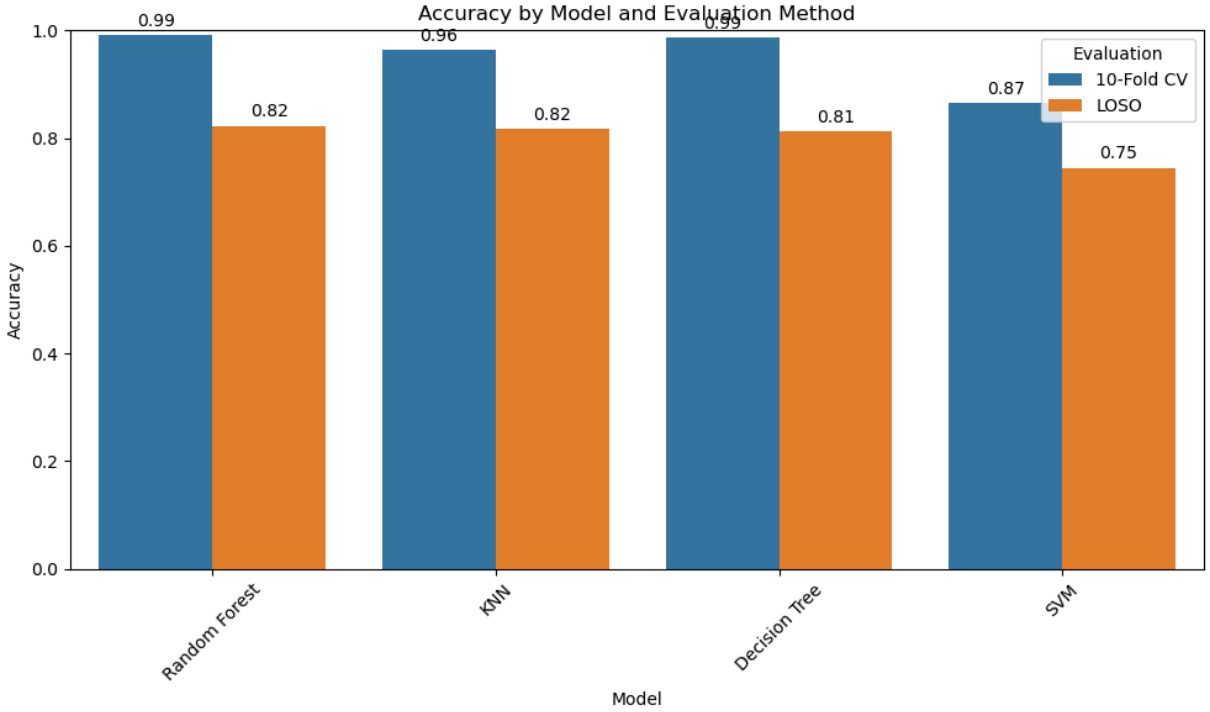
- Similar to RF in 10-Fold CV (~98.7%)
- ~81% in LOSO
- High precision, stable but less robust than RF

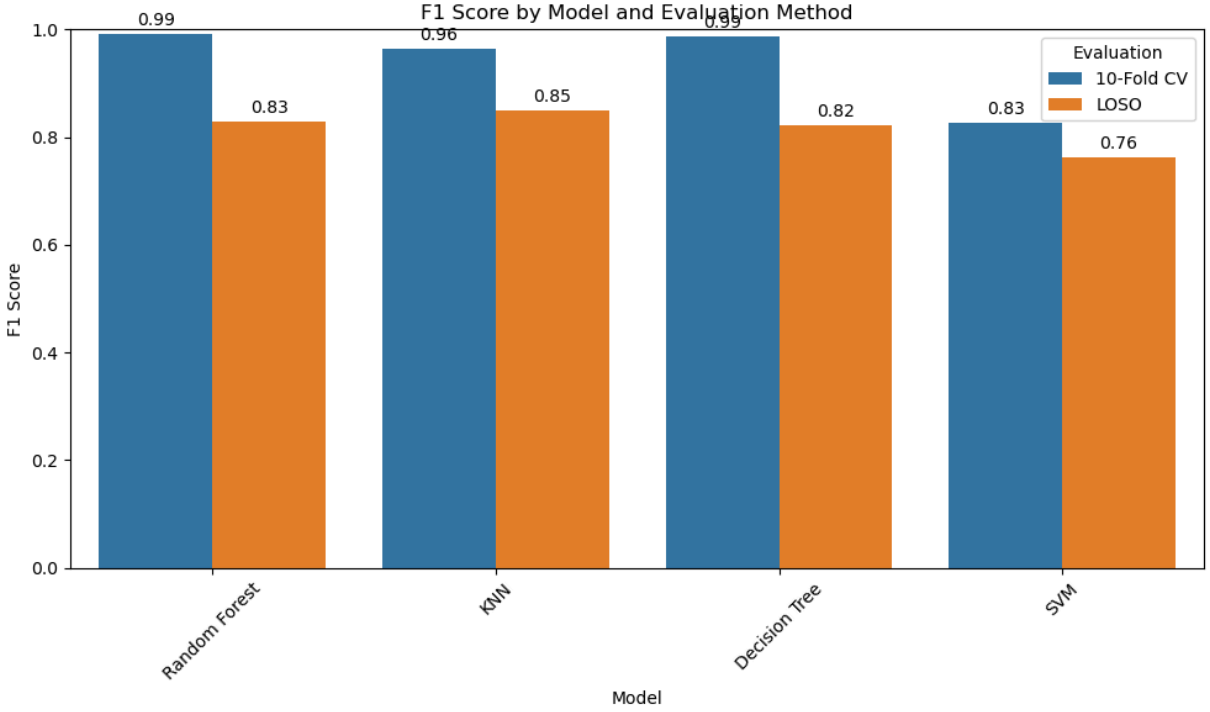
SVM

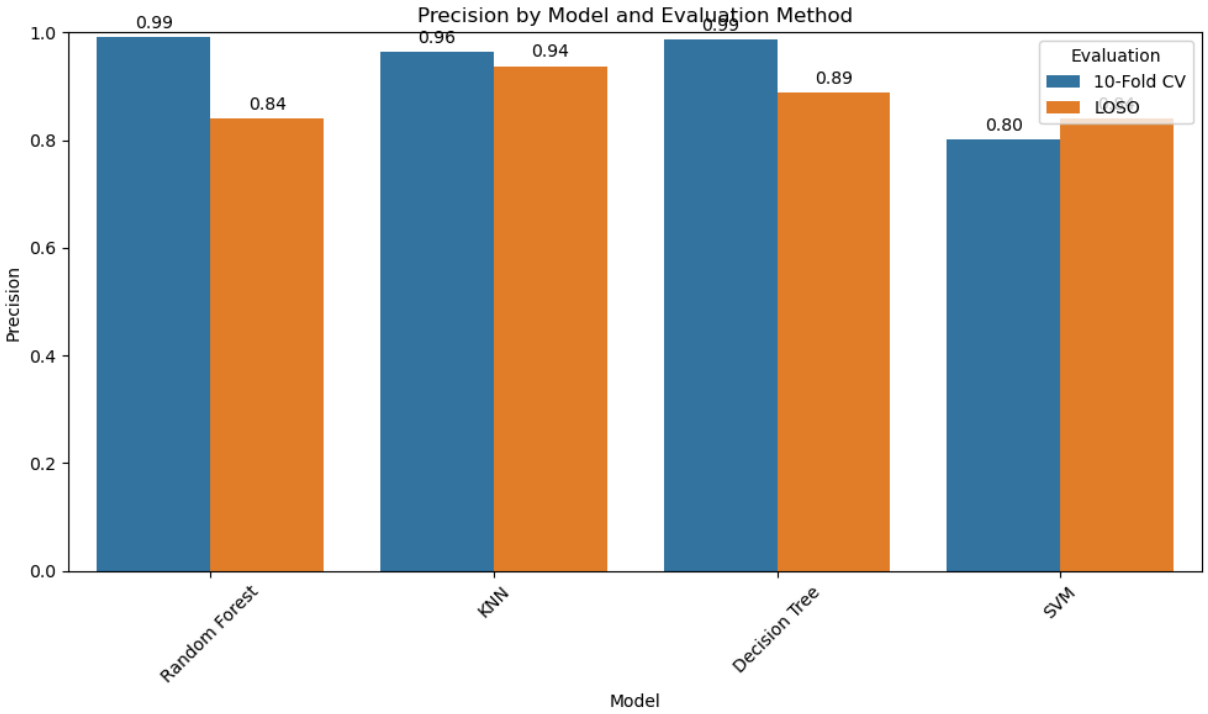
- Weakest performer overall
- ~87% in 10-Fold, ~74.5% in LOSO
- Struggles to generalize across subjects

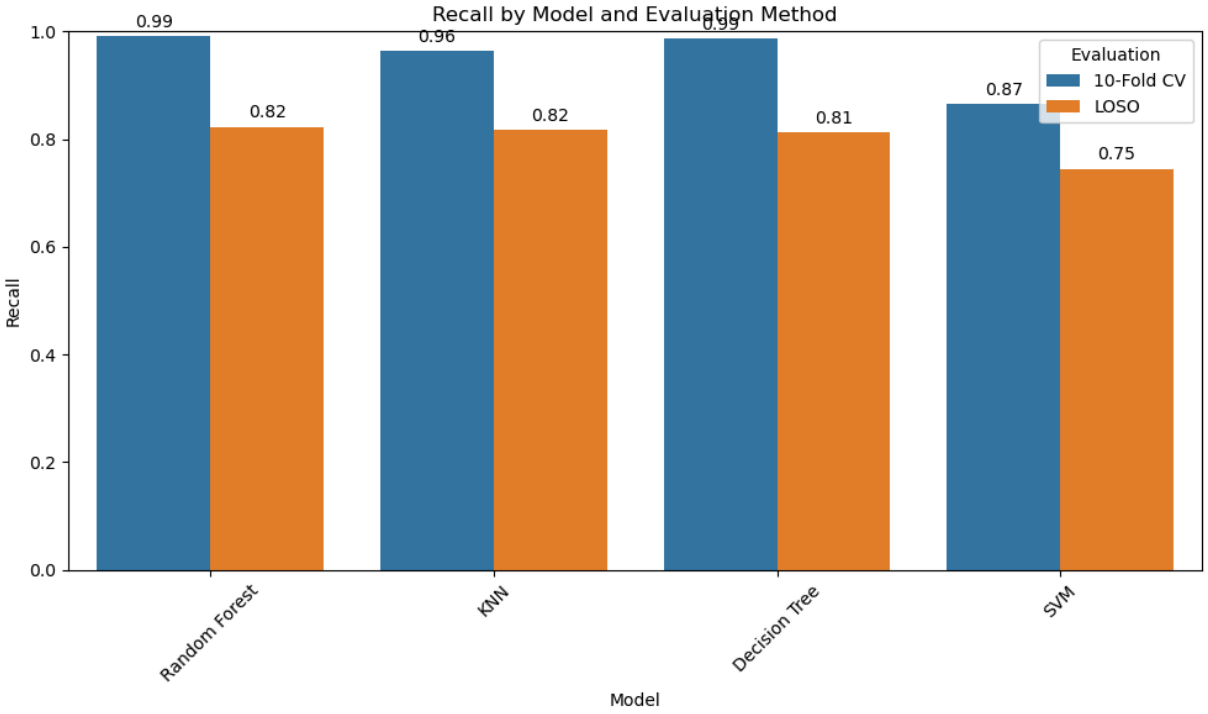
Bar Plots of Evaluation Metrics for Each Model

In [23]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Metrics to plot
metrics = ["Accuracy", "F1 Score", "Precision", "Recall"]

# Generate a bar plot for each metric
for metric in metrics:
    plt.figure(figsize=(10, 6))

    # Create grouped bar plot by model and evaluation type
    ax = sns.barplot(data=results_df, x="Model", y=metric, hue="Evaluation")

    # Title and axis formatting
    plt.title(f"{metric} by Model and Evaluation Method")
    plt.ylabel(metric)
    plt.ylim(0, 1)
    plt.xticks(rotation=45)
    plt.legend(title="Evaluation")

    # Add percentage labels on each bar
    for container in ax.containers:
        ax.bar_label(container, fmt="%.2f", label_type="edge", padding=3)

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


Accuracy by Model and Evaluation Method
Evaluation
10-Fold CV
LOSO
Accuracy
Model
Random Forest
KNN
Decision Tree
SVM
0.99
0.82
0.96
0.82
0.99
0.81
0.87
0.75


F1 Score by Model and Evaluation Method
Evaluation
10-Fold CV
LOSO
F1 Score
Model
Random Forest
KNN
Decision Tree
SVM
0.99
0.83
0.96
0.85
0.99
0.82
0.83
0.76

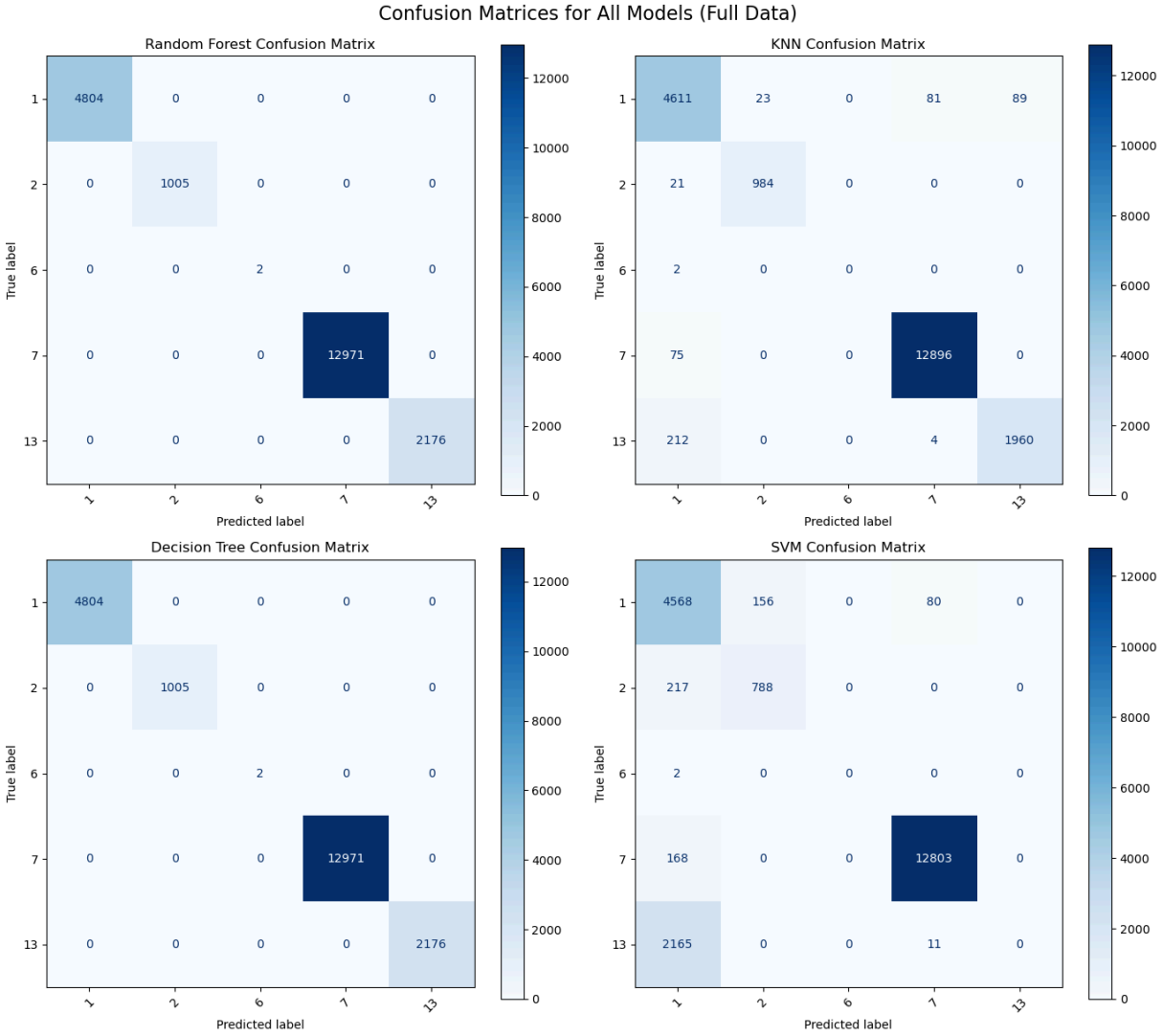




Confusion Matrices for All Models (Full Data)

In [25]:
```
from sklearn.metrics import ConfusionMatrixDisplay
# Set up a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(14, 12))
axes = axes.flatten()  # Flatten to access each subplot by index

# Loop through each model and plot its confusion matrix
for ax, (name, model) in zip(axes, models.items()):
    model.fit(X_selected, y)                # Train on all available data
    preds = model.predict(X_selected)       # Predict on the same data (for visualiz
```

```
    # Display the confusion matrix on its respective subplot
    disp = ConfusionMatrixDisplay.from_predictions(y, preds, ax=ax, cmap='Blues', x
    ax.set_title(f"{name} Confusion Matrix")

# Layout and overall figure formatting
plt.tight_layout()
plt.suptitle("Confusion Matrices for All Models (Full Data)", fontsize=16, y=1.02)
plt.show()
```

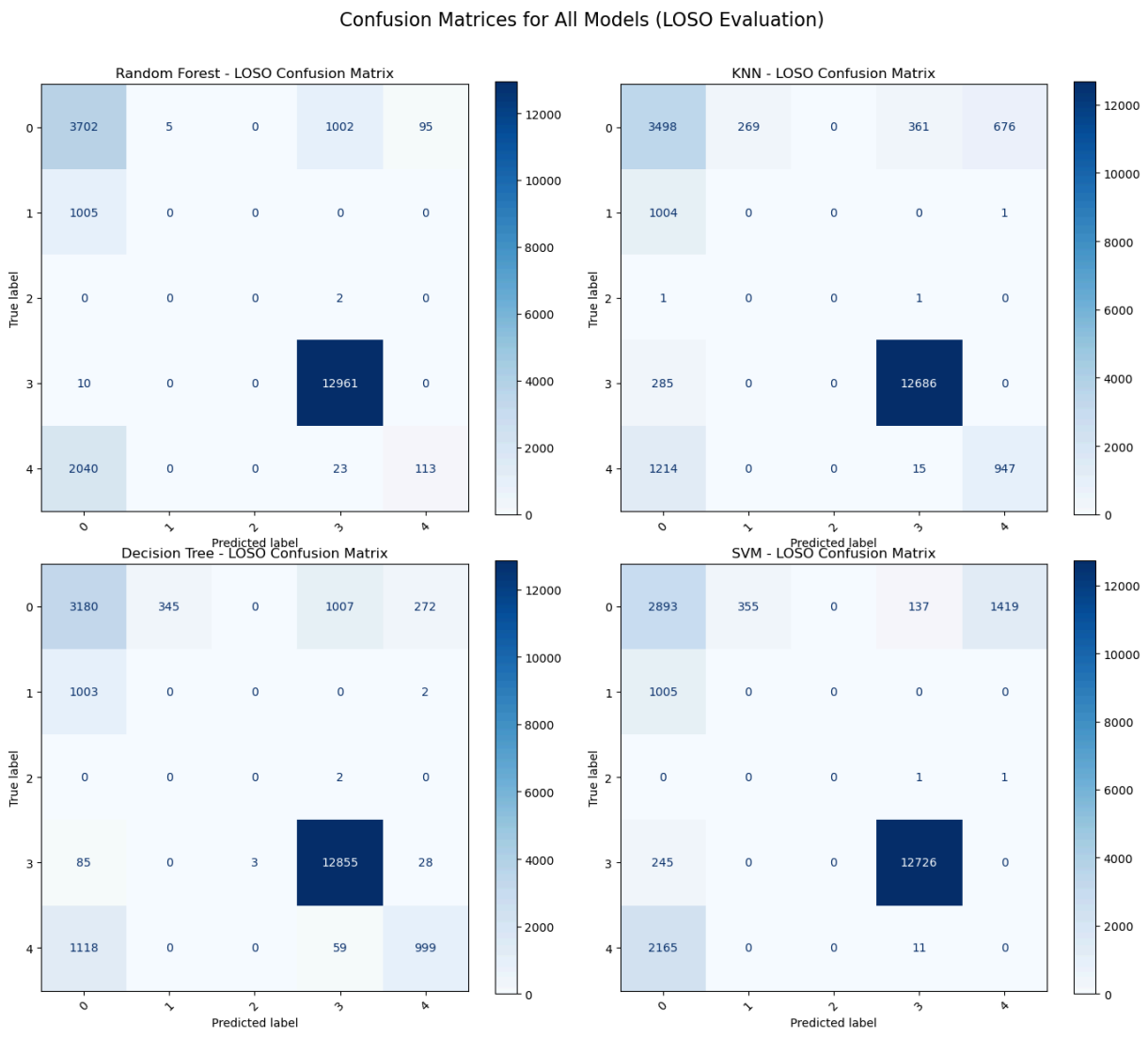


LOSO Confusion Matrices for All Models

In [26]:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Create a 2x2 grid of subplots (one for each model)
fig, axes = plt.subplots(2, 2, figsize=(14, 12))
axes = axes.flatten()

logo = LeaveOneGroupOut()

# Loop through each model and generate its LOSO confusion matrix
for ax, (name, model) in zip(axes, models.items()):
```

```
    y_true_all = []
    y_pred_all = []

    # Collect predictions across all LOSO folds
    for train_idx, test_idx in logo.split(X_selected, y, groups):
        model.fit(X_selected.iloc[train_idx], y[train_idx])    # Train on all subje
        preds = model.predict(X_selected.iloc[test_idx])       # Predict on the hel
        y_true_all.extend(y[test_idx])                         # Save true labels
        y_pred_all.extend(preds)                               # Save predictions

    # Compute full confusion matrix after all folds
    cm = confusion_matrix(y_true_all, y_pred_all)

    # Plot confusion matrix in respective subplot
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot(ax=ax, cmap="Blues", xticks_rotation=45)
    ax.set_title(f"{name} - LOSO Confusion Matrix")

# Add overall title and spacing
plt.suptitle("Confusion Matrices for All Models (LOSO Evaluation)", fontsize=16, y=
plt.tight_layout()
plt.show()
```



The confusion matrices illustrate the stark difference in model performance between 10-Fold Cross-Validation and Leave-One-Subject-Out (LOSO) evaluation. In 10-Fold CV, the models perform exceptionally well, showing near-perfect accuracy and clear class separability—this is because training and testing data are randomly split, allowing the model to learn patterns from all subjects. However, when evaluated under LOSO, where the model must predict on a subject it has never seen during training, performance drops significantly. The confusion matrices become more scattered, indicating that the models struggle to generalize across different individuals. This suggests that while the features work well for seen subjects, more robust or personalized methods may be needed for deployment in real-world, subject-independent scenarios.

10-Fold CV:

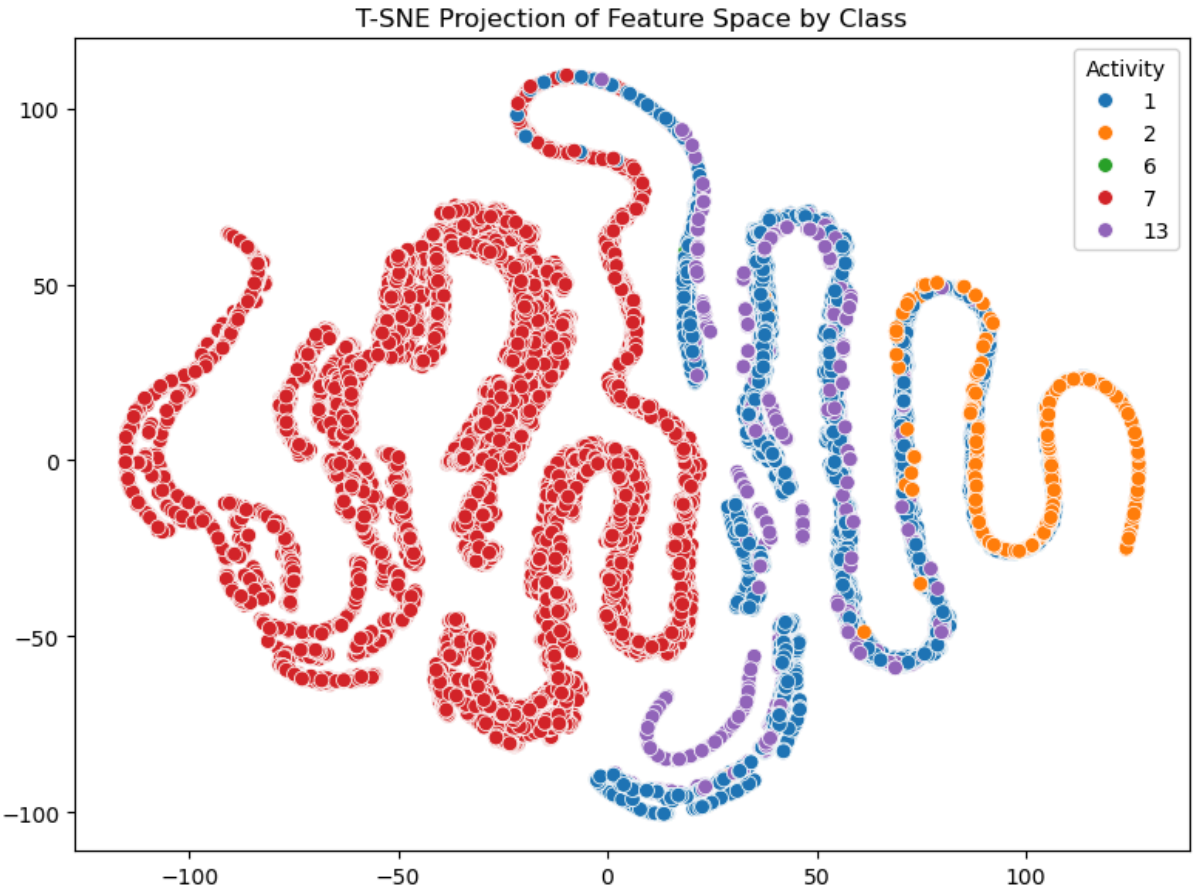
- Very high accuracy (up to ~99%)
- Minimal misclassifications (strong diagonal in confusion matrices)
- Easier task due to shared subjects in train/test sets
- Good for quick model comparisons and tuning

LOSO:

- Realistic generalization setting (~74–82% accuracy)
- Increased confusion between similar activity classes
- Highlights model limitations when facing unseen users
- Essential for evaluating real-world deployment readiness

Visualizing Feature Space with T-SNE

In [27]:
```
from sklearn.manifold import TSNE

# Project high-dimensional feature space to 2D using T-SNE
X_vis = TSNE(n_components=2, perplexity=40, random_state=42).fit_transform(X_select

# Create a scatter plot of the 2D projection, colored by class label
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_vis[:, 0], y=X_vis[:, 1], hue=y, palette='tab10', s=50)

# Plot formatting
plt.title("T-SNE Projection of Feature Space by Class")
plt.legend(title='Activity')
plt.tight_layout()
plt.show()
```



This T-SNE (T-SNE, also called T-distributed Stochastic Neighbor Embedding (t-SNE), is commonly used in machine learning to see patterns, clusters, or separability in feature-rich data) projection shows how our extracted features separate different activity classes. Activities like '2' and '13' form tight, distinct clusters, indicating strong feature separability. In contrast, others show some overlap, helping us identify which activities may need more refined features or model tuning.