Portfolio Assessment 2 Report

Personal Information

- **Name**: Dang Khoa Le
- **Student ID**: 103844421
- **Studio Class**: Studio 1-7 (Tuesday 6.30PM – 8.30PM)

Aim

The aim of this task is to demonstrate a systematic approach to developing a machine learning model. The focus is on data pre-processing, class labelling, data separation, feature selection, training, and model evaluation. We will use acceleration data collected from body-worn sensors to distinguish between two types of meat processing activities: boning and slicing.

About the Dataset

This dataset (Boning.csv and Slicing.csv from ampc2.zip) contains acceleration data collected using 17 body worn sensors in different body positions. From 17 sensors we get 22 body position values in x,y,z acceleration, so we have total 66 columns. There are 2 data files that contains acceleration values, and each belong to a type of meat processing activity (boning and slicing). Indicating class value for these 2 datasets with 0 - boning, 1 – slicing. Each row contains data from 1 frame and length of each frame is 1 second. That's mean 60 frames contains 1 minute data.

Objective

The objective of this task is to develop a classification model that can accurately distinguish between the activities of boning and slicing. The task involves extracting relevant features, creating composite columns, computing statistical features, and training multiple machine learning models. Finally, we will evaluate these models using different techniques, including hyperparameter tuning, feature selection, and dimensionality reduction and conclude the best model from evaluation.

Step 1: Data Collection

- **Assigned Columns:**
  - **Column Set 1:** Right Shoulder (x, y, z)
  - **Column Set 2:** Left Shoulder (x, y, z)

```
# Step 1: Data collection
# Student ID ending with 1 (103844421)
# => Column set 1: Right Shoulder (x,y,z)
# => Column set 2: Left Shoulder (x,y,z)
folder_path = 'ampc2'
# Get csv files
boning_df = pd.read_csv(os.path.join(folder_path, 'Boning.csv'))
slicing_df = pd.read_csv(os.path.join(folder_path, 'Slicing.csv'))
# Apply Class as 0 (Boning) and 1 (Slicing)
boning_df['Class'] = 0
slicing_df['Class'] = 1
boning_df.to_csv('updated_Boning.csv', index=False)
slicing_df.to_csv('updated_Slicing.csv', index=False)
# Load the updated files
updated_boning_df = pd.read_csv('updated_Boning.csv')
updated_slicing_df = pd.read_csv('updated_Slicing.csv')
# Filtrate column and combine the CSV files
combined_df = pd.concat([updated_boning_df, updated_slicing_df], ignore_index=True)
# Add the Frame column
# Modified to increment per each row, to avoid duplicated frame id when coming 2 file
combined_df['Frame'] = range(len(combined_df))
keeping_features =  [
    'Right Shoulder x',
    'Right Shoulder y',
    'Right Shoulder z',
    'Left Shoulder x',
    'Left Shoulder y',
    'Left Shoulder z',
]
# Save the combined dataset to CSV file
combined_df[['Frame'] + keeping_features + ['Class']].to_csv('combined_data.csv', index=False)
```

- **Data Extraction:**
  - The data was extracted from the two provided files (Boning.csv and Slicing.csv).
  - Add the Class column to the 2 CSV files, with the value of 0 for Boning.csv and 1 for Slicing.csv, which indicating the 2 data activities' value from original dataset when combined. These updates will be saved as 'updated_Boning.csv' and 'updated_Slicing.csv'.
  - Set the new Frame value to the new combined dataset as incremented integer value, (started from 0). This ensure the data's id will not be repeated.
  - The columns corresponding to the assigned sets ('Right Shoulder x', 'Right Shoulder y', 'Right Shoulder z' and 'Left Shoulder x', 'Left Shoulder y', 'Left Shoulder z') were extracted and combined (from the 2 new updated datasets) with the Frame and Class columns defined, resulting in an initial dataset with 8 columns.

Extracted and combined dataset saved as 'combined_data.csv'

Step 2: Creating Composite Columns

- **Composite Data Points:** Set the equation to calculate
  - **Root Mean Square (RMS):** for Right and Left Shoulder 'x and y', 'y and z', 'z and x', and 'x, y, and z' sets combined. Using the RMS equation of $\sqrt{(a^2 + b^2)}/2$ (for a combined of 2 entries) and $\sqrt{(a^2 + b^2 + c^2)}/3$ (for a combined of 3 entries).
  - **Roll:** Calculated as $180 * \frac{\text{atan2(accelY,}\sqrt{\text{accelX^2+accelZ^2)}}}{\pi}$
  - **Pitch:** Calculated as $180 * \frac{\text{atan2(accelX,}\sqrt{\text{accelY^2+accelZ^2)}}}{\pi}$

- **Final Dataset:**

The dataset was expanded to include 12 composite columns for each column set, resulting in a final dataset with 20 columns (with 8 from combined_data.csv).

Composite dataset saved as 'composite_data.csv'

```python
# Step 2: Create composite columns
# Load file
combined_df = pd.read_csv('combined_data.csv')
# Functions to calculate composite data points
# Calculate root mean square, given there could be 2 or 3 entry values (a,b,c)
# Define function to separate these 2 scenarios to calculate the root mean square
def root_mean_square(a, b, c=None):
    if c is None: # Case 2 entries
        return np.sqrt((a**2 + b**2) / 2)        # √(a^2+b^2)/2
    else:         # Case 3 entries
        return np.sqrt((a**2 + b**2 + c**2) / 3) # √(a^2+b^2+c^2)/3
def roll(accelY, accelX, accelZ): # 180 * atan2(accelY, sqrt(accelX*accelX + accelZ*accelZ))/PI
    return 180 * np.arctan2(accelY, np.sqrt(accelX**2 + accelZ**2)) / np.pi
def pitch(accelX, accelY, accelZ):
    return 180 * np.arctan2(accelX, np.sqrt(accelY**2 + accelZ**2)) / np.pi
# Call functions to calculate composite data points for Column Set 1 (Right Shoulder)
combined_df['Right Shoulder RMS xy'] = root_mean_square(combined_df['Right Shoulder x'], combined_df['Right Shoulder y'])
combined_df['Right Shoulder RMS yz'] = root_mean_square(combined_df['Right Shoulder y'], combined_df['Right Shoulder z'])
combined_df['Right Shoulder RMS zx'] = root_mean_square(combined_df['Right Shoulder z'], combined_df['Right Shoulder x'])
combined_df['Right Shoulder RMS xyz'] = root_mean_square(combined_df['Right Shoulder x'], combined_df['Right Shoulder y'], combined_df['Right Shoulder z'])
combined_df['Right Shoulder Roll'] = roll(combined_df['Right Shoulder y'], combined_df['Right Shoulder x'], combined_df['Right Shoulder z'])
combined_df['Right Shoulder Pitch'] = pitch(combined_df['Right Shoulder x'], combined_df['Right Shoulder y'], combined_df['Right Shoulder z'])
# Call functions to calculate composite data points for Column Set 2 (Left Shoulder)
combined_df['Left Shoulder RMS xy'] = root_mean_square(combined_df['Left Shoulder x'], combined_df['Left Shoulder y'])
combined_df['Left Shoulder RMS yz'] = root_mean_square(combined_df['Left Shoulder y'], combined_df['Left Shoulder z'])
combined_df['Left Shoulder RMS zx'] = root_mean_square(combined_df['Left Shoulder z'], combined_df['Left Shoulder x'])
combined_df['Left Shoulder RMS xyz'] = root_mean_square(combined_df['Left Shoulder x'], combined_df['Left Shoulder y'], combined_df['Left Shoulder z'])
combined_df['Left Shoulder Roll'] = roll(combined_df['Left Shoulder y'], combined_df['Left Shoulder x'], combined_df['Left Shoulder z'])
combined_df['Left Shoulder Pitch'] = pitch(combined_df['Left Shoulder x'], combined_df['Left Shoulder y'], combined_df['Left Shoulder z'])
# Restructure columns order
final_columns = [
    'Frame',
    'Right Shoulder x', 'Right Shoulder y', 'Right Shoulder z',
    'Left Shoulder x', 'Left Shoulder y', 'Left Shoulder z',
    'Right Shoulder RMS xy', 'Right Shoulder RMS yz', 'Right Shoulder RMS zx', 'Right Shoulder RMS xyz', 'Right Shoulder Roll', 'Right Shoulder Pitch',
    'Left Shoulder RMS xy', 'Left Shoulder RMS yz', 'Left Shoulder RMS zx', 'Left Shoulder RMS xyz', 'Left Shoulder Roll', 'Left Shoulder Pitch',
    'Class'
]
# Save the updated composite dataset to CSV file
combined_df[final_columns].to_csv('composite_data.csv', index=False)
```

Step 3: Data Pre-processing and Feature Computation

- **Statistical Features:**
  - The following statistical features were computed per minute (60 frames) for each of the 18 columns:
    1. Mean // using np.mean()
    2. Standard Deviation // using np.mean()
    3. Min // using np.min()
    4. Max // using np.max()
    5. Area Under the Curve (AUC) // using np.trapz()
    6. Peaks (Number of Peaks) // using len (frequencies) and find_peaks()

```python
# Step 3: Data pre-processing and Feature computation
# Load file
composite_df = pd.read_csv('composite_data.csv')
# Set columns to compute statistics (2-19)
columns_to_compute = composite_df.columns[1:19]
# Function to compute features for a single window
def compute_statistical_features(window):
    features = {} # Initialize an empty array to store computed features
    features['Frame'] = window['Frame'].iloc[0] # Include Frame as a feature
    for col in columns_to_compute:
        data = window[col].values # Extract the values of current column for the window
        # Compute statistical features
        features[f'{col}_mean'] = np.mean(data)             # Mean
        features[f'{col}_std'] = np.std(data)               # Standard deviation
        features[f'{col}_min'] = np.min(data)               # Min value
        features[f'{col}_max'] = np.max(data)               # Max value
        features[f'{col}_auc'] = np.trapz(data)             # Area under the curve (AUC)
        features[f'{col}_peaks'] = len(find_peaks(data)[0]) # Number of peaks
    features['Class'] = window['Class'].iloc[0] # Include Class as a feature
    return features
# Create an empty list to hold the results
statistical_features = []
# Process the DataFrame in windows of 60 frames
window_size = 60
for i in range(0, len(composite_df), window_size):
    # Extract the window of data
    window = composite_df.iloc[i:i+window_size]
    # Compute the features for this window
    features = compute_statistical_features(window)
    # Append the result to the list
    statistical_features.append(features)
# Convert the list of dictionaries to a DataFrame
stat_features_df = pd.DataFrame(statistical_features)
# Save the resulting DataFrame to a CSV file
stat_features_df.to_csv('statistical_features.csv', index=False)
```

- **Final Dataset:**

This step resulted in a dataset with 108 features (18 columns of extracted and composite sets multiplied by 6 features defined).
Set the window to 60 (frames per minute) and process data.
Statistical features dataset saved as 'statistical_features.csv'

Step 4: Model Training and Evaluation

1. Algorithm of Data Training and Summary Tables

Data Preparation and Train-Test Split:

- **Load Data:** The data is loaded from statistical_features.csv. The Class column is separated as the target feature (y), and the remaining columns are used as features (X).
- **Train-Test Split (70/30):** The data is split into training and testing sets using train_test_split() with a 70/30 split. This creates the X_train, X_test, y_train, and y_test sets.

```
# Step 4: Training
# Load file
stat_features_df = pd.read_csv('statistical_features.csv')
# Separate features and target
X = stat_features_df.drop(columns=['Class'])
y = stat_features_df['Class']
# 1) Train-Test Split (70/30)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

SVM Models:

1. **SVM with Original Features:**
   - **Train the Model:** The SVM model is trained using the original features without any tuning.
   - **Evaluate the Model:** The model is evaluated on the test data, and 10-fold cross-validation is performed to calculate the average accuracy.

```
# 2) 10-fold cross validation
# Train SVM with default settings
svm = SVC()
svm.fit(X_train, y_train)
# Evaluate on test data
train_accuracy = svm.score(X_train, y_train)
test_accuracy = svm.score(X_test, y_test)
# Cross-validation
cv_scores = cross_val_score(svm, X, y, cv=10)
cv_accuracy = cv_scores.mean()
# Store results
results = {
    'Model': 'SVM (Original features)',
    'Train-test split': f'{test_accuracy:.2%}',
    'Cross validation': f'{cv_accuracy:.2%}'
}
```

2. **SVM with Hyperparameter Tuning:**
   - **Grid Search:** GridSearchCV is used to find the best hyperparameters (C, gamma, and kernel) for the SVM model.
   - **Train and Evaluate:** The model with the best hyperparameters is trained, and both the test accuracy and cross-validation accuracy are calculated.

```python
# 3) 1 and 2 with hyper parameter tuning
# Hyperparameter tuning with GridSearchCV
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [1, 0.1, 0.01, 0.001],
    'kernel': ['rbf', 'linear']
}
grid = GridSearchCV(SVC(), param_grid, refit=True, cv=10)
grid.fit(X_train, y_train)
# Best parameters
best_params = grid.best_params_
# Evaluate on test data
train_accuracy = grid.score(X_train, y_train)
test_accuracy = grid.score(X_test, y_test)
# Cross-validation
cv_scores = cross_val_score(grid.best_estimator_, X, y, cv=10)
cv_accuracy = cv_scores.mean()
# Store results (2 dp)
results_hyper = {
    'Model': 'SVM (With hyperparameter tuning)',
    'Train-test split': f'{test_accuracy:.2%}',
    'Cross validation': f'{cv_accuracy:.2%}'
}
```

3. **SVM with Feature Selection and Hyperparameter Tuning:**
   - **Feature Selection:** SelectKBest is used to select the top 10 features.
   - **Grid Search:** The model is tuned using the same grid search approach.
   - **Train and Evaluate:** The model is trained with the selected features and evaluated.

```python
# 4) 1 and 2 with hyper parameter tuning and 10 best features
# Feature selection with SelectKBest
selector = SelectKBest(f_classif, k=10)
X_train_kbest = selector.fit_transform(X_train, y_train)
X_test_kbest = selector.transform(X_test)
# Train with hyperparameter tuning
grid.fit(X_train_kbest, y_train)
# Evaluate on test data
train_accuracy = grid.score(X_train_kbest, y_train)
test_accuracy = grid.score(X_test_kbest, y_test)
# Cross-validation
cv_scores = cross_val_score(grid.best_estimator_, selector.transform(X), y, cv=10)
cv_accuracy = cv_scores.mean()
# Store results
results_feature_selection = {
    'Model': 'SVM (With feature selection and hyperparameter tuning)',
    'Train-test split': f'{test_accuracy:.2%}',
    'Cross validation': f'{cv_accuracy:.2%}'
}
```

4. **SVM with PCA and Hyperparameter Tuning:**
   - **PCA:** Principal Component Analysis (PCA) is used to reduce the data to the top 10 principal components.
   - **Grid Search:** The model is tuned using the same grid search approach.
   - **Train and Evaluate:** The model is trained with the PCA-transformed features and evaluated.

```python
# 5) 1 and 2 with hyper parameter tuning and 10 principal components
# PCA for dimensionality reduction
pca = PCA(n_components=10)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
# Train with hyperparameter tuning
grid.fit(X_train_pca, y_train)
# Evaluate on test data
train_accuracy = grid.score(X_train_pca, y_train)
test_accuracy = grid.score(X_test_pca, y_test)
# Cross-validation
cv_scores = cross_val_score(grid.best_estimator_, pca.transform(X), y, cv=10)
cv_accuracy = cv_scores.mean()
# Store results
results_pca = {
    'Model': 'SVM (With PCA and hyperparameter tuning)',
    'Train-test split': f'{test_accuracy:.2%}',
    'Cross validation': f'{cv_accuracy:.2%}'
}
```

Other Classifiers:

1. **SGDClassifier:**
   - **Train and Evaluate:** The model is trained and evaluated on the test set, and 10-fold cross-validation is performed.
2. **RandomForestClassifier:**

- **Train and Evaluate:** The model is trained and evaluated similarly to the SGD classifier.

3. **MLPClassifier:**
   - **Train and Evaluate:** The model is trained and evaluated similarly to the previous classifiers.

```
# 7) Train SGD, RandomForest, and MLP Classifiers
# Train SGDClassifier
sgd = SGDClassifier(max_iter=1000, tol=1e-3)
sgd.fit(X_train, y_train)
sgd_test_accuracy = sgd.score(X_test, y_test)
sgd_cv_accuracy = cross_val_score(sgd, X, y, cv=10).mean()
# Train RandomForestClassifier
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
rf_test_accuracy = rf.score(X_test, y_test)
rf_cv_accuracy = cross_val_score(rf, X, y, cv=10).mean()
# Train MLPClassifier
mlp = MLPClassifier(max_iter=1000)
mlp.fit(X_train, y_train)
mlp_test_accuracy = mlp.score(X_test, y_test)
mlp_cv_accuracy = cross_val_score(mlp, X, y, cv=10).mean()
```

Summary Tables:

- **SVM Models Summary Table:** The results of the SVM models (with different configurations) are stored in a DataFrame and printed.

```
# 6) Summary table for SVM Models
# Create a DataFrame for the summary table
summary_svm = pd.DataFrame([results, results_hyper, results_feature_selection, results_pca])
print("Summary table for SVM Models")
print(summary_svm)
```

- **All Models Summary Table:** The results of the SVM, SGD, RandomForest, and MLP models are stored in a DataFrame and printed.

```
# 8) Summary table for all Models
# Create a summary table for all models
summary_all_models = pd.DataFrame({
    'Model': ['SVM', 'SGD', 'RandomForest', 'MLP'],
    'Train-test split': [
        results['Train-test split'],
        f'{sgd_test_accuracy:.2%}',
        f'{rf_test_accuracy:.2%}',
        f'{mlp_test_accuracy:.2%}'
    ],
    'Cross validation': [
        results['Cross validation'],
        f'{sgd_cv_accuracy:.2%}',
        f'{rf_cv_accuracy:.2%}',
        f'{mlp_cv_accuracy:.2%}'
    ]
})
print("Summary table for all Models")
print(summary_all_models)
```

2. Understand the data

SVM Models:

1. **Original Features (No Tuning):**
   - Train-Test Split (70/30): The accuracy on the test set was 100.00%.
   - 10-Fold Cross-Validation: The average accuracy across folds was 94.93%.
2. **With Hyperparameter Tuning:**
   - Best Parameters: C = 1, gamma = 0.1, kernel = rbf
   - Train-Test Split (70/30): The accuracy on the test set improved to 99.72%.
   - 10-Fold Cross-Validation: The average accuracy across folds was 95.01%.
3. **With Feature Selection and Hyperparameter Tuning:**
   - Feature Selection: Top 10 features selected using SelectKBest.
   - Train-Test Split (70/30): The accuracy on the test set remained at 99.72%.
   - 10-Fold Cross-Validation: The average accuracy across folds was 95.01%.
4. **With PCA and Hyperparameter Tuning:**
   - PCA Components: Top 10 principal components selected.
   - Train-Test Split (70/30): The accuracy on the test set remained at 99.72%.
   - 10-Fold Cross-Validation: The average accuracy across folds was 95.01%.

Other Classifiers:

1. **SGD Classifier:**
   - Train-Test Split (70/30): The accuracy on the test set was 63.16%.
   - 10-Fold Cross-Validation: The average accuracy across folds was 64.85%.
2. **RandomForest Classifier:**
   - Train-Test Split (70/30): The accuracy on the test set was 97.51%.
   - 10-Fold Cross-Validation: The average accuracy across folds was 95.43%.
3. **MLP Classifier:**
   - Train-Test Split (70/30): The accuracy on the test set was 89.20%.
   - 10-Fold Cross-Validation: The average accuracy across folds was 90.35%.

Summary Table for SVM Models

Model	Train-test split	Cross validation
SVM (Original features)	100.00%	94.93%
SVM (With hyperparameter tuning)	99.72%	95.01%
SVM (With feature selection and hyperparameter tuning)	99.72%	95.01%
SVM (With PCA and hyperparameter tuning)	99.72%	95.01%

Summary Table for All Models

```
(sklearn-env) khoale@khoas-MacBook-Pro-2 A2 % python3 a2.py
Summary table for SVM Models
                                                Model Train-test split Cross validation
0                           SVM (Original features)          100.00%           94.93%
1                    SVM (With hyperparameter tuning)           99.72%           95.01%
2  SVM (With feature selection and hyperparameter...           99.72%           95.01%
3            SVM (With PCA and hyperparameter tuning)           99.72%           95.01%
--------------------------------------------------------------------------------
Summary table for all Models
          Model Train-test split Cross validation
0           SVM          100.00%           94.93%
1           SGD           63.16%           64.85%
2  RandomForest           97.51%           95.43%
3           MLP           89.20%           90.35%
```

Fig. Terminal output for the 2 tables

Model	Train-test split	Cross validation
SVM	100.00%	94.93%
SGD	63.16%	64.85%
RandomForest	97.51%	95.43%
MLP	89.20%	90.35%

Step 5: Model Selection

1) Which SVM model will be the best for your problem?

Based on the results from the previous steps, the best SVM model for this problem is:

- SVM with Hyperparameter Tuning (also with Feature Selection and PCA):
  - Train-Test Split Accuracy: 99.72%
  - Cross-Validation Accuracy: 95.01%

Reasoning:

- The SVM model with hyperparameter tuning consistently provided high accuracy on both the train-test split and cross-validation datasets.
- While feature selection and PCA did not significantly improve the performance beyond the hyperparameter-tuned SVM, the tuned model's performance was highly consistent and robust across all configurations.
- Therefore, the hyperparameter-tuned SVM (with or without additional feature selection or PCA) is the best SVM model for this problem.

2) Which ML model will be the best for your problem?

Based on the results, the best overall ML model for this problem is:

- RandomForest Classifier:
  - Train-Test Split Accuracy: 97.51%
  - Cross-Validation Accuracy: 95.43%

Reasoning:

- Although the SVM models showed slightly higher train-test split accuracy (especially the hyperparameter-tuned SVM), the RandomForest classifier outperformed the SVM models in cross-validation accuracy.

- Cross-validation accuracy is a more reliable indicator of a model's generalization ability because it evaluates the model's performance across multiple folds of the data, reducing the risk of overfitting to a specific dataset.
- The RandomForest classifier also showed strong performance across both evaluation metrics, making it a more versatile and robust choice for this problem.
- Additionally, RandomForest classifiers are less sensitive to feature scaling and hyperparameter tuning compared to SVMs, making them easier to work with in practice.

Therefore, the RandomForest classifier is selected as the best overall machine learning model for this problem, considering its strong and consistent performance across different evaluation criteria.

Studio 3 Activity 6 (Summary Table for SVM Models)

Model	Train-test split	Cross validation
SVM (Original features)	91.86%	91.79%
SVM (With hyperparameter tuning)	89.19%	89.24%
SVM (With feature selection and hyperparameter tuning)	92.49%	92.60%
SVM (With PCA and hyperparameter tuning)	88.59%	89.53%

Key Observations:

1. **SVM with Original Features:**
   - The SVM model with the original features performs well, achieving a high accuracy of 91.86% on the train-test split and 91.79% on cross-validation. This indicates that the original features are effective for this classification task, providing a strong baseline performance.
2. **SVM with Hyperparameter Tuning:**
   - Surprisingly, after applying hyperparameter tuning, the performance slightly decreases, with 89.19% accuracy on the train-test split and 89.24% on cross-validation. This could suggest that the hyperparameter tuning led to overfitting or that the chosen hyperparameters did not improve the model's generalization as expected.
   - It's possible that the hyperparameter grid search might have explored suboptimal regions of the parameter space, or the original configuration was already close to optimal.
3. **SVM with Feature Selection and Hyperparameter Tuning:**
   - This configuration outperforms all others, achieving the highest accuracy of 92.49% on the train-test split and 92.60% on cross-validation. This suggests that selecting the top features and tuning the model together helps the SVM focus on the most informative parts of the data, improving both performance and generalization.
   - The improvement compared to the original model indicates that the additional complexity introduced by irrelevant features was reduced, leading to a more accurate model.
4. **SVM with PCA and Hyperparameter Tuning:**
   - This configuration shows the lowest performance among all, with 88.59% accuracy on the train-test split and 89.53% on cross-validation. While PCA is useful for reducing dimensionality, it might have discarded some important features in this context, leading to a drop in performance.

- The slight increase in cross-validation accuracy compared to the train-test split suggests that PCA helped prevent overfitting to some extent, but it still couldn't compete with feature selection.

Studio 3 Activity 7 (Summary Table for All Models)

Model	Train-test split	Cross validation
SVM	91.86%	91.79%
SGD	85.24%	89.13%
RandomForest	92.43%	92.54%
MLP	89.65%	89.70%

Key Observations:

1. **SVM:**
   - The SVM model with feature selection and hyperparameter tuning achieved 91.86% on the train-test split and 91.79% on cross-validation. While performing well, it's slightly outperformed by the RandomForest model in both metrics.
2. **SGD:**
   - The SGD classifier showed the lowest performance in the train-test split (85.24%) but improved significantly in cross-validation (89.13%). This indicates that while the model may struggle with the initial split, it generalizes better with cross-validation.
3. **RandomForest:**
   - RandomForest outperformed all models, achieving the highest accuracy on both the train-test split (92.43%) and cross-validation (92.54%). This suggests that RandomForest is the most robust model in this comparison, likely due to its ensemble nature, which reduces overfitting.
4. **MLP:**
   - The MLP classifier performed better than SGD but lower than SVM and RandomForest, with 89.65% on the train-test split and 89.70% on cross-validation. This indicates a relatively stable performance, but it falls short compared to RandomForest and SVM.

Appendix:

Source codes and CSV files can be found from this Google Drive link (access allowed):

https://drive.google.com/drive/folders/19fd1F3pCENdyV8Zlnt1sUgCqmG2etv0F?usp=sharing

In the Python script 'a2.py' notice that all programs from the 4 activities is placed within this single file, by which it is best to commend-in (hashtag '#' the part that you do not wish to run).

The file is separated by 4 sections (4 steps), suggestively, only run 1 step section at the time while disable the rest to understand the program the best.