AYDIN ADNAN MENDERES UNIVERSITY

ENGINEERING FACULTY
COMPUTER SCIENCE ENGINEERING DEPARTMENT

Comparison Between ML and DL methods in Signal Classification

CSE436 | Signal Processing and Machine Learning | Fall 2024/2025

Student Number | Name & Surname:

211805118 | Engin Halil YEDİRMEZ
211805078 | Ali Cemal GÜLMEZ
211805131 | Cemal KADIOĞLU

Lecturer:

Associate Prof. Dr. Ahmet Çağdaş SEÇKİN

Voice Command Dataset

```
import numpy as np
import os
import librosa
import time
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
import matplotlib.pyplot as plt
import seaborn as sns
```

Imported Libraries Explanation:

- numpy: Used for numerical computations and array operations.
- os: Handles file and directory operations, like listing or accessing paths.
- librosa: Provides functions for audio analysis and feature extraction (e.g., MFCC, chroma).
- time: Tracks the time elapsed during operations.
- imblearn.over_sampling.SMOTE: Synthetic Minority Over-sampling Technique to balance imbalanced datasets.
- sklearn.preprocessing.StandardScaler: Scales data to have zero mean and unit variance.
- sklearn.model_selection: Splits datasets and performs cross-validation.
- sklearn.metrics: Provides evaluation metrics for models like accuracy and confusion matrices.
- sklearn.ensemble: Contains ensemble learning models such as Random Forest and Gradient Boosting.
- sklearn.tree, neighbors, naive_bayes, svm, linear_model: Different machine learning classifiers.
- xgboost, lightgbm: Gradient boosting classifiers for better model performance.
- matplotlib.pyplot, seaborn: Used for plotting and data visualization.

```python
# Özellik çıkarma fonksiyonu (zenginleştirilmiş)
def extract_features_with_fixed_sampling_rate(data_path, exclude_class="inohom", sr=16000):
    features, labels = [], []
    for command_dir in os.listdir(data_path):
        if command_dir == exclude_class:
            continue
        command_path = os.path.join(data_path, command_dir)
        if os.path.isdir(command_path):
            for file in os.listdir(command_path):
                if file.endswith('.wav'):
                    file_path = os.path.join(command_path, file)

                    # Sabit sampling rate ile ses yükleme
                    y, _ = librosa.load(file_path, sr=sr, mono=True)

                    # Özellikler
                    mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=40)
                    chroma = librosa.feature.chroma_stft(y=y, sr=sr)
                    spectral_contrast = librosa.feature.spectral_contrast(y=y, sr=sr)
                    zero_crossing_rate = librosa.feature.zero_crossing_rate(y=y)
                    spectral_rolloff = librosa.feature.spectral_rolloff(y=y, sr=sr)
                    spectral_centroid = librosa.feature.spectral_centroid(y=y, sr=sr)
                    rms = librosa.feature.rms(y=y)
                    delta_mfccs = librosa.feature.delta(mfccs)
                    delta2_mfccs = librosa.feature.delta(mfccs, order=2)
                    onset_env = librosa.onset.onset_strength(y=y, sr=sr)

                    # Özellikleri birleştir
                    combined_features = np.hstack([
                        np.mean(mfccs.T, axis=0), np.std(mfccs.T, axis=0),
                        np.mean(chroma.T, axis=0), np.std(chroma.T, axis=0),
                        np.mean(spectral_contrast.T, axis=0), np.std(spectral_contrast.T, axis=0),
                        np.mean(zero_crossing_rate.T, axis=0),
                        np.mean(spectral_rolloff.T, axis=0), np.std(spectral_rolloff.T, axis=0),
                        np.mean(spectral_centroid.T, axis=0), np.std(spectral_centroid.T, axis=0),
                        np.mean(rms.T, axis=0), np.std(rms.T, axis=0),
                        np.mean(delta_mfccs.T, axis=0), np.std(delta_mfccs.T, axis=0),
                        np.mean(delta2_mfccs.T, axis=0), np.std(delta2_mfccs.T, axis=0),
                        np.mean(onset_env), np.std(onset_env)
                    ])
                    features.append(combined_features)
                    labels.append(command_dir)
    return np.array(features), np.array(labels)
```

This function extract_features_with_fixed_sampling_rate extracts audio features from .wav files in a specified directory structure and returns them as feature vectors with their corresponding class labels. The function first initializes empty lists for features and labels. It then iterates over the subdirectories in data_path, skipping any directory matching exclude_class. For each valid audio file, the function loads the audio using a fixed sampling rate (sr=16000) and converts stereo signals to mono if needed. It extracts various audio features, including MFCCs (Mel-frequency cepstral coefficients), chroma features (pitch class distribution), spectral contrast (difference between spectral peaks and valleys), zero-crossing rate (frequency of amplitude sign changes), spectral rolloff (frequency below which a percentage of the spectrum lies), spectral centroid (spectral "center of mass"), root mean square (RMS) energy, delta and delta-delta MFCC features (first and second-order temporal changes), and onset envelope (intensity change over time). For each feature type, the mean and standard deviation are computed to create a comprehensive feature vector. These vectors are combined using np.hstack() and appended to the features list. The corresponding class label, derived from the subdirectory name, is stored in the labels list. Finally, the function returns NumPy arrays for the extracted features and labels, which are well-suited for further use in machine learning models for audio classification tasks.

```python
# Özellikleri çıkar
print("Extracting features with fixed sampling rate (16000 Hz)...")
features, labels = extract_features_with_fixed_sampling_rate(data_path, sr=16000)


# Verileri ölçeklendir
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)

# SMOTE ile sınıf dengesini sağla
print("Applying SMOTE for class balance...")
smote = SMOTE(random_state=42)
features_balanced, labels_balanced = smote.fit_resample(features_scaled, labels)

# Eğitim ve test setlerini böl
X_train, X_test, y_train, y_test = train_test_split(features_balanced, labels_balanced, test_size=0.2, stratify=labels_balanced, random_state=42)
```

This code segment performs the following steps for audio data preprocessing and preparation for machine learning:

1. Feature Extraction: The call to extract_features_with_fixed_sampling_rate(data_path, sr=16000) extracts features from audio files at a fixed sampling rate of 16,000 Hz. The extracted features and their corresponding labels are assigned to features and labels, respectively.
2. Feature Scaling: The StandardScaler() is applied to normalize the features by removing the mean and scaling them to unit variance. This ensures that all features contribute equally to the learning process. The result is stored in features_scaled.
3. Class Imbalance Handling: To address potential class imbalance issues, the Synthetic Minority Oversampling Technique (SMOTE) is employed via smote.fit_resample(). SMOTE generates synthetic samples for underrepresented classes, creating a more balanced dataset. The balanced features and labels are stored in features_balanced and labels_balanced.
4. Train-Test Split: The data is divided into training and testing sets using train_test_split() with an 80-20 split. The stratify=labels_balanced parameter ensures that class distribution is preserved between the training and testing datasets. The random seed (random_state=42) guarantees reproducibility of the split. The resulting sets are stored in X_train, X_test, y_train, and y_test.

```python
# Modeller
models = {
    "Decision Tree": DecisionTreeClassifier(max_depth=30, class_weight='balanced',
                                            random_state=42),
    "Random Forest": RandomForestClassifier(n_estimators=500, max_depth=30, min_samples_split=5, class_weight='balanced', random_state=42),
    "K-Nearest Neighbors": KNeighborsClassifier(n_neighbors=5, weights='distance'),
    "Naive Bayes": GaussianNB(),
    "SVM": SVC(kernel='rbf', class_weight='balanced', random_state=42),
    "Logistic Regression": LogisticRegression(max_iter=1000, class_weight='balanced',
                                              random_state=42)
}
```

This part of the code defines a dictionary named models containing several machine learning classifiers for supervised learning tasks such as classification. Each key-value pair represents a model name and its corresponding instance initialized with specific parameters.

1. Decision Tree (DecisionTreeClassifier): This classifier is configured with a maximum tree depth of 30 to control complexity and avoid overfitting. The class_weight='balanced' parameter handles class imbalance by adjusting weights inversely proportional to class frequencies.
2. Random Forest (RandomForestClassifier): Composed of 500 decision trees (n_estimators=500), this ensemble model has a maximum depth of 30 and requires a minimum of 5 samples to split nodes (min_samples_split=5). Class balancing and reproducibility are ensured using class_weight='balanced' and random_state=42.
3. K-Nearest Neighbors (KNeighborsClassifier): This algorithm uses 5 neighbors (n_neighbors=5) and applies distance-based weighting (weights='distance') to give higher influence to closer neighbors during classification.
4. Naive Bayes (GaussianNB): This simple probabilistic classifier assumes Gaussian distribution for continuous features, often used for fast and efficient classification tasks.
5. Support Vector Machine (SVC): The kernel used here is the radial basis function (kernel='rbf') to handle nonlinear classification boundaries. Class imbalance is addressed with class_weight='balanced'.
6. Logistic Regression (LogisticRegression): A linear model with max_iter=1000 to ensure convergence during optimization. Class balancing and reproducibility are managed with class_weight='balanced' and random_state=42.

```python
# 10-katlı çapraz doğrulama
print("Performing stratified 10-fold cross-validation...")
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

best_model = None
best_accuracy = 0
for name, model in models.items():
    accuracies = []
    for train_index, val_index in skf.split(X_train, y_train):
        X_tr, X_val = X_train[train_index], X_train[val_index]
        y_tr, y_val = y_train[train_index], y_train[val_index]

        model.fit(X_tr, y_tr)
        predictions = model.predict(X_val)
        acc = accuracy_score(y_val, predictions)
        accuracies.append(acc)

    mean_accuracy = np.mean(accuracies)
    print(f"{name} Mean CV Accuracy: {mean_accuracy:.4f}")

    if mean_accuracy > best_accuracy:
        best_model = model
        best_accuracy = mean_accuracy

print("\nBest Model:", best_model)
```

Cross-Validation Initialization:
StratifiedKFold is used to maintain the class distribution across training and validation sets (n_splits=10 for 10 folds). Shuffling (shuffle=True) ensures randomness, and random_state=42 maintains reproducibility.
Model Evaluation Loop:
The loop iterates over the models dictionary. For each model:

- Data Splitting: The dataset is split into training (X_tr, y_tr) and validation (X_val, y_val) sets based on indices returned by skf.split(X_train, y_train).
- Training: The model is trained on the training subset using model.fit(X_tr, y_tr).
- Prediction & Accuracy Calculation: Predictions are made for the validation set, and their accuracy is computed using accuracy_score(y_val, predictions).
- Accuracy Storage: The accuracy for each fold is stored in the list accuracies.

Mean Accuracy Calculation:
The average accuracy across all folds (mean_accuracy) is computed using np.mean(accuracies). This score represents the model's cross-validation performance.
Model Selection:
If a model achieves better accuracy than the current best (best_accuracy), it is stored in best_model, and best_accuracy is updated.
Output:
The mean cross-validation accuracy for each model is printed. After all iterations, the best-performing model is identified and displayed.

```python
# Test seti üzerinde değerlendirme
start_time = time.time()
final_predictions = best_model.predict(X_test)
test_time = time.time() - start_time

print(f"\nTest Accuracy: {accuracy_score(y_test, final_predictions):.4f}")
print(f"Test Time: {test_time:.4f} seconds")
print("Classification Report:\n", classification_report(y_test, final_predictions))
```

Prediction Timing:
start_time = time.time() records the start time to measure how long the model takes to make predictions on the test set.
After best_model.predict(X_test), the difference between the current and recorded time (test_time) provides the total prediction time.
Model Prediction:
final_predictions = best_model.predict(X_test) generates predicted class labels for the test set (X_test).
Test Accuracy Calculation:
The function accuracy_score(y_test, final_predictions) computes the proportion of correctly predicted labels.
Test Results Display:
The test accuracy and prediction time are printed in formatted outputs (.4f specifies four decimal points).
Detailed Classification Report:
classification_report(y_test, final_predictions) generates a detailed report that includes precision, recall, F1-score, and support for each class.
Key Metrics in the Classification Report:

- Precision: The proportion of true positive predictions out of all predicted positives.
- Recall: The proportion of true positive predictions out of actual positive samples.
- F1-Score: A harmonic mean of precision and recall, useful for imbalanced datasets.
- Support: The number of true instances for each class in the test set.

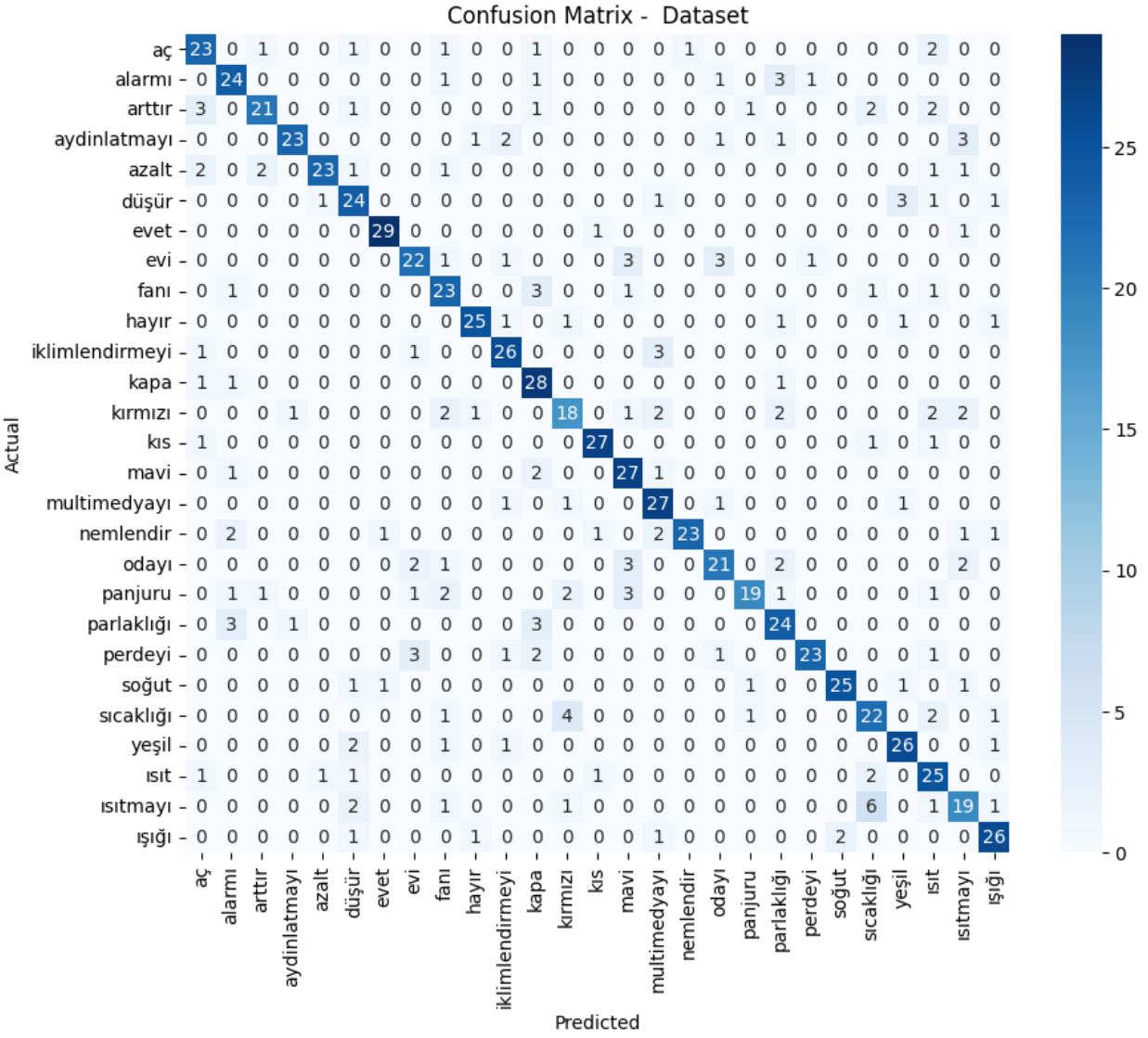
```python
# Karışıklık matrisi
conf_matrix = confusion_matrix(y_test, final_predictions)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=np.unique(y_test), yticklabels=np.unique(y_test))
plt.title("Confusion Matrix -  Dataset")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

Confusion Matrix Calculation:
conf_matrix = confusion_matrix(y_test, final_predictions) creates a matrix where rows represent the true classes, and columns represent the predicted classes.
Diagonal elements indicate the number of correctly predicted samples for each class.
Off-diagonal elements show misclassifications.
Figure Setup:
plt.figure(figsize=(10, 8)) initializes the plot size to ensure the matrix is visually clear.
Heatmap Creation:
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues") displays the matrix using Seaborn's heatmap function:
annot=True: Shows the numeric values in each cell.
fmt="d": Formats these values as integers.
cmap="Blues": Uses a blue color gradient to make high values visually distinct.
Axis Labels and Title:
The x-axis (xticklabels) and y-axis (yticklabels) are labeled using unique classes from y_test. Titles and labels are added for clarity.
Plot Display:
plt.show() renders the plot.

Results

```
/usr/local/lib/python3.11/dist-packages/dask/dataframe/__init__.py:42: FutureWarning:
Dask dataframe query planning is disabled because dask-expr is not installed.

You can install it with `pip install dask[dataframe]` or `conda install dask`.
This will raise in a future version.

  warnings.warn(msg, FutureWarning)
Extracting features with fixed sampling rate (16000 Hz)...
Applying SMOTE for class balance...
Performing stratified 10-fold cross-validation...
Decision Tree Mean CV Accuracy: 0.3617
Random Forest Mean CV Accuracy: 0.7754
K-Nearest Neighbors Mean CV Accuracy: 0.5475
Naive Bayes Mean CV Accuracy: 0.4101
SVM Mean CV Accuracy: 0.7601
Logistic Regression Mean CV Accuracy: 0.7748

Best Model: RandomForestClassifier(class_weight='balanced', max_depth=30,
                       min_samples_split=5, n_estimators=500, random_state=42)

Test Accuracy: 0.7728
Test Time: 0.1894 seconds
Classification Report:
                  precision    recall  f1-score   support

              aç       0.72      0.77      0.74        30
          alarmı       0.73      0.77      0.75        31
          arttır       0.84      0.68      0.75        31
     aydinlatmayı       0.92      0.74      0.82        31
           azalt       0.92      0.74      0.82        31
         düşür       0.71      0.77      0.74        31
            evet       0.94      0.94      0.94        31
...
        accuracy                           0.77       832
       macro avg       0.78      0.77      0.77       832
    weighted avg       0.78      0.77      0.77       832
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...


Confusion Matrix - Dataset
Actual
Predicted
aç
alarmı
arttır
aydinlatmayı
azalt
düşür
evet
evi
fanı
hayır
iklimlendirmeyi
kapa
kırmızı
kıs
mavi
multimedyayı
nemlendir
odayı
panjuru
parlaklığı
perdeyi
soğut
sıcaklığı
yeşil
ısıt
ısıtmayı
ışığı
0
5
10
15
20
25

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import LSTM, Bidirectional, Dropout, Dense
from tensorflow.keras.models import Sequential

print("Preparing data for LSTM...")
X_train, X_test, y_train, y_test = train_test_split(
    features_balanced, labels_balanced, test_size=0.2, stratify=labels_balanced, random_state=42
)



# SMOTE ile sınıf dengesini sağla
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Veriyi tekrar ölçeklendir
scaler = StandardScaler()
X_train_smote_scaled = scaler.fit_transform(X_train_smote)
X_test_scaled = scaler.transform(X_test)

def pad_or_crop_features(features, target_size):
    current_size = features.shape[1]
    if current_size < target_size:
        # Sıfırlarla doldur
        pad_width = target_size - current_size
        features = np.pad(features, ((0, 0), (0, pad_width)), mode='constant')
    elif current_size > target_size:
        # Fazlalıkları kırp
        features = features[:, :target_size]
    return features

# 100 özellik boyutuna getir
X_train = pad_or_crop_features(X_train, target_size=200)
X_test = pad_or_crop_features(X_test, target_size=200)



# LSTM için 3D şekline dönüştür
X_train_lstm = X_train.reshape(X_train.shape[0], 20, 10) #change
X_test_lstm = X_test.reshape(X_test.shape[0], 20, 10) #change
```

In this code, the dataset is first split into training and testing sets using train_test_split(), with an 80-20% split while maintaining class distribution using stratified sampling. SMOTE (Synthetic Minority Over-sampling Technique) is then applied to the training set to balance class distribution. The SMOTE() function generates synthetic samples for the minority class to match the number of samples in the majority class. Afterward, the features are scaled using StandardScaler, which standardizes the feature values to have zero mean and unit variance. The scaled training and testing sets are then prepared for the next steps. The function pad_or_crop_features() is used to ensure all feature vectors are the same size. If the current feature size is smaller than the target size (200 in this case), the data is padded with zeros, and if it's larger, excess features are truncated. Lastly, the reshaped data is converted into 3D arrays suitable for input into an LSTM model. Specifically, X_train_lstm and X_test_lstm are reshaped to the shape (samples, timesteps, features), where samples is the number of data points, timesteps is 20 (an arbitrary choice for the sequence length), and features is 10, representing the number of features per timestep. This reshaping is crucial for the LSTM to process sequential data.

In this part of the code, the class labels are first converted into categorical format using

tf.keras.utils.to_categorical(). A label_map dictionary is created to assign each class label a unique integer index. The labels for both the training and testing sets (y_train and y_test) are then transformed into categorical form using this mapping.

```
# Sınıfları kategorik hale getir
label_map = {label: idx for idx, label in enumerate(np.unique(y_train))}
y_train_cat = tf.keras.utils.to_categorical([label_map[label] for label in y_train])
y_test_cat = tf.keras.utils.to_categorical([label_map[label] for label in y_test])



# Model 2: LSTM
print("Building LSTM model...")
lstm_model = Sequential([
    Bidirectional(LSTM(256, return_sequences=True), input_shape=(10, 10)),
    Bidirectional(LSTM(256)),
    Dropout(0.5),
    Dense(512, activation='relu'),
    Dense(len(label_map), activation='softmax')
])

lstm_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
lstm_model.summary()

# Train LSTM model
lstm_history = lstm_model.fit(X_train_lstm, y_train_cat, validation_split=0.2, epochs=50, batch_size=16)

# Save LSTM model
lstm_model.save("lstm_model.h5")

# Test LSTM model
start_time = time.time()
lstm_predictions = lstm_model.predict(X_test_lstm)
lstm_test_time = time.time() - start_time
lstm_pred_classes = np.argmax(lstm_predictions, axis=1)
y_test_classes = [label_map[label] for label in y_test]
# LSTM evaluation
lstm_accuracy = accuracy_score(y_test_classes, lstm_pred_classes)
print(f"\nLSTM Test Accuracy: {lstm_accuracy:.4f}")
print(f"LSTM Test Time: {lstm_test_time:.4f} seconds")
print("LSTM Classification Report:\n", classification_report(y_test_classes, lstm_pred_classes))

# Plot LSTM confusion matrix
conf_matrix_lstm = confusion_matrix(y_test_classes, lstm_pred_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix_lstm, annot=True, fmt="d", cmap="Blues", xticklabels=label_map.keys(), yticklabels=label_map.keys())
plt.title("Confusion Matrix - LSTM")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

The next step builds a Bidirectional Long Short-Term Memory (LSTM) model using TensorFlow/Keras. The LSTM model consists of two bidirectional LSTM layers with 256 units each, which allows the model to capture information from both past and future time steps in a sequence. The first LSTM layer is configured with return_sequences=True, which ensures that the output from each time step is returned for the next layer. A Dropout layer with a 50% rate is added to reduce overfitting by randomly setting some of the input values to zero during training. The model then passes through a Dense layer with 512 units and ReLU activation, followed by the final output layer with softmax activation, which outputs a probability distribution across all classes. The model is compiled using the Adam optimizer and categorical cross-entropy loss, suitable for multi-class classification.

The model is trained for 50 epochs with a batch size of 16, and 20% of the data is used as a validation set during training. After training, the model is saved to a file named lstm_model.h5.

Once trained, the model's performance is evaluated on the test set. The predictions are made using lstm_model.predict() and the accuracy is calculated by comparing the predicted class labels with the true labels (y_test_classes). A classification report is printed, which provides detailed metrics such as precision, recall, and F1 score for each class.

Finally, a confusion matrix is plotted using Seaborn's heatmap() to visualize the model's classification performance, with the true labels on the y-axis and the predicted labels on the x-axis. This matrix helps to identify any misclassifications in the model's predictions.

Results

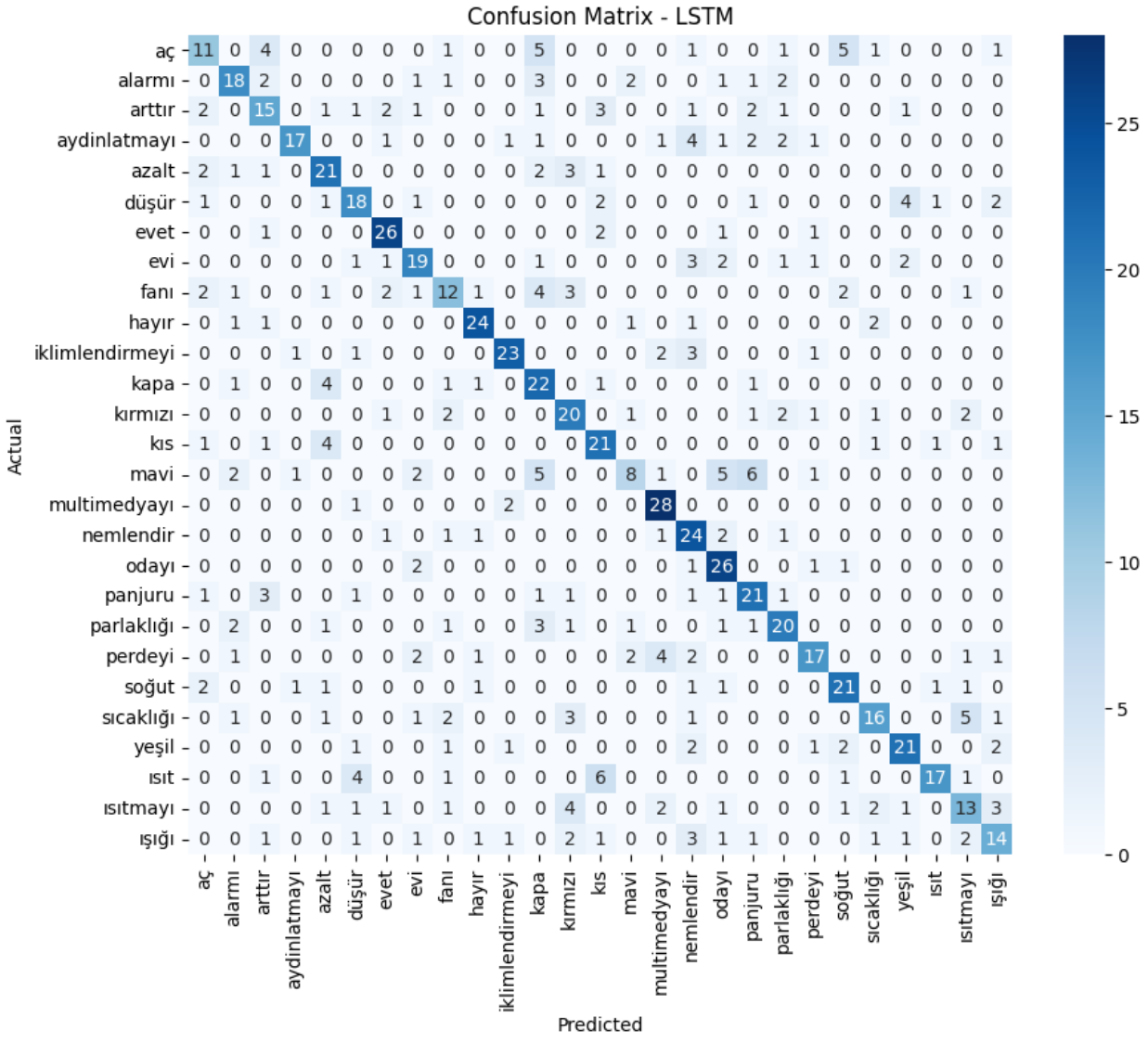
```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
bidirectional (Bidirectional)	(None, 10, 512)	546,816
bidirectional_1 (Bidirectional)	(None, 512)	1,574,912
dropout (Dropout)	(None, 512)	0
dense (Dense)	(None, 512)	262,656
dense_1 (Dense)	(None, 27)	13,851

```
 Total params: 2,398,235 (9.15 MB)

 Trainable params: 2,398,235 (9.15 MB)

 Non-trainable params: 0 (0.00 B)

Epoch 1/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 84s 401ms/step - accuracy: 0.0984 - loss: 3.1174 - val_accuracy: 0.2102 - val_loss: 2.6489
Epoch 2/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 78s 379ms/step - accuracy: 0.2381 - loss: 2.4502 - val_accuracy: 0.3273 - val_loss: 2.2115
Epoch 3/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 61s 365ms/step - accuracy: 0.4060 - loss: 1.9168 - val_accuracy: 0.3453 - val_loss: 2.0841
Epoch 4/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 83s 373ms/step - accuracy: 0.5188 - loss: 1.5944 - val_accuracy: 0.4595 - val_loss: 1.8170
Epoch 5/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 63s 378ms/step - accuracy: 0.6056 - loss: 1.2594 - val_accuracy: 0.5030 - val_loss: 1.6756
Epoch 6/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 83s 386ms/step - accuracy: 0.6899 - loss: 0.9473 - val_accuracy: 0.5105 - val_loss: 1.6551
Epoch 7/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 62s 369ms/step - accuracy: 0.7698 - loss: 0.6999 - val_accuracy: 0.5180 - val_loss: 1.7420
Epoch 8/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 61s 363ms/step - accuracy: 0.8144 - loss: 0.5405 - val_accuracy: 0.5495 - val_loss: 1.6979
Epoch 9/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 86s 386ms/step - accuracy: 0.8766 - loss: 0.3714 - val_accuracy: 0.5405 - val_loss: 1.8663
Epoch 10/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 80s 377ms/step - accuracy: 0.9127 - loss: 0.2684 - val_accuracy: 0.5631 - val_loss: 1.9237
Epoch 11/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 64s 381ms/step - accuracy: 0.9109 - loss: 0.2506 - val_accuracy: 0.5360 - val_loss: 1.9852
Epoch 12/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 84s 392ms/step - accuracy: 0.9323 - loss: 0.2037 - val_accuracy: 0.5375 - val_loss: 2.1605
Epoch 13/50
...
Epoch 49/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 62s 369ms/step - accuracy: 0.9766 - loss: 0.0636 - val_accuracy: 0.5931 - val_loss: 2.7530
Epoch 50/50
167/167 ━━━━━━━━━━━━━━━━━━━━ 61s 365ms/step - accuracy: 0.9739 - loss: 0.0831 - val_accuracy: 0.6111 - val_loss: 2.8379
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is consi
26/26 ━━━━━━━━━━━━━━━━━━━━ 3s 88ms/step
```

```
LSTM Test Accuracy: 0.6166
LSTM Test Time: 5.1744 seconds
LSTM Classification Report:
              precision    recall  f1-score   support

           0       0.50      0.37      0.42        30
           1       0.64      0.58      0.61        31
           2       0.50      0.48      0.49        31
           3       0.85      0.55      0.67        31
           4       0.58      0.68      0.63        31
           5       0.60      0.58      0.59        31
           6       0.74      0.84      0.79        31
           7       0.61      0.61      0.61        31
           8       0.50      0.40      0.44        30
           9       0.80      0.80      0.80        30
          10       0.82      0.74      0.78        31
          11       0.46      0.71      0.56        31
          12       0.54      0.65      0.59        31
          13       0.57      0.70      0.63        30
          14       0.53      0.26      0.35        31
          15       0.72      0.90      0.80        31
          16       0.50      0.77      0.61        31
          17       0.60      0.84      0.70        31
...
    accuracy                           0.62       832
   macro avg       0.63      0.62      0.61       832
weighted avg       0.63      0.62      0.61       832
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...



```python
import os
import numpy as np
import librosa
import librosa.display
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Conv2D, MaxPooling2D, Flatten, Dense, LSTM, Dropout,
                                     BatchNormalization, Bidirectional)
import time

# Define paths and constants
data_path = '/content/drive/MyDrive/KAYIT_DOSYALARI-20250117T090128Z-001/ KAYIT_DOSYALARI'
SAMPLING_RATE = 16000
CLASSES_TO_EXCLUDE = ["inohom"]
TARGET_CLASSES = 27

# Function to extract MFCCs
def extract_features(file_path, n_mfcc=40, max_pad_len=100):
    y, sr = librosa.load(file_path, sr=SAMPLING_RATE)
    mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=n_mfcc)
    if mfccs.shape[1] < max_pad_len:
        pad_width = max_pad_len - mfccs.shape[1]
        mfccs = np.pad(mfccs, pad_width=((0, 0), (0, pad_width)), mode='constant')
    else:
        mfccs = mfccs[:, :max_pad_len]
    return mfccs
```

This code part begins by importing the necessary libraries, such as os, numpy, librosa, matplotlib, and seaborn, among others. It defines a few constants and the path to the dataset directory (data_path). The SAMPLING_RATE is set to 16,000, which is a common value for audio signal processing, and CLASSES_TO_EXCLUDE is specified to filter out any undesired class labels (in this case, the class inohom). The TARGET_CLASSES is set to 27, which likely represents the number of distinct sound classes in the dataset.

The function extract_features() is defined to extract Mel-Frequency Cepstral Coefficients (MFCCs) from audio files. MFCCs are a commonly used feature for audio classification tasks because they capture the timbral aspects of an audio signal, which are relevant for speech and music recognition. Inside the function, an audio file is loaded using librosa.load(), and the MFCCs are computed using librosa.feature.mfcc(). The shape of the MFCC matrix is checked, and if it is smaller than the specified max_pad_len (100), the matrix is padded with zeros to make it consistent in size. If the MFCC matrix exceeds the target size, it is truncated to the first max_pad_len columns. This ensures that all audio files yield a consistent feature shape, which is crucial for feeding the data into machine learning models.

```python
# Load and preprocess data
def load_data(data_path):
    features, labels = [], []
    for root, _, files in os.walk(data_path):
        for file in files:
            if file.endswith(".wav"):
                label = os.path.basename(root)
                if label not in CLASSES_TO_EXCLUDE:
                    file_path = os.path.join(root, file)
                    mfcc = extract_features(file_path)
                    features.append(mfcc)
                    labels.append(label)
    return np.array(features), np.array(labels)

# Load data
print("Loading data...")
X, y = load_data(data_path)

# Encode labels
le = LabelEncoder()
y_encoded = le.fit_transform(y)
y_categorical = tf.keras.utils.to_categorical(y_encoded)

# Split data stratified
X_train, X_test, y_train, y_test = train_test_split(
    X, y_categorical, test_size=0.2, stratify=y_encoded, random_state=42
)

# Reshape for CNN and LSTM
X_train_cnn = X_train[..., np.newaxis]
X_test_cnn = X_test[..., np.newaxis]
```

This section of the code handles the loading and preprocessing of the audio data. The load_data() function traverses the directory specified by data_path, looking for .wav files. For each .wav file found, it retrieves the label from the parent directory's name (which presumably corresponds to the class of the sound in the file). The function then calls the previously defined extract_features() function to extract MFCCs from the audio file and appends the features and corresponding labels to the features and labels lists, respectively.
After loading the data, the labels (y) are encoded using LabelEncoder, which converts the string labels into numerical format. These encoded labels are then transformed into categorical format using tf.keras.utils.to_categorical, which is appropriate for multi-class classification tasks in neural networks.
Next, the dataset is split into training and testing sets using train_test_split with stratification, ensuring that the class distribution is preserved in both sets. The test_size=0.2 argument means that 20% of the data will be reserved for testing, while 80% is used for training.
Lastly, the input data (X_train and X_test) is reshaped for compatibility with both Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) models. The [..., np.newaxis] syntax adds an extra dimension to the features to represent the channels (required for CNNs). This is done for both the training and testing data. The reshaping ensures that the data can be processed by models that expect a 3D input (for example, in the form (samples, height, width, channels) for CNNs or (samples, time_steps, features) for LSTMs).

```python
# Model 1: 2D CNN
print("Building 2D CNN model...")
cnn_model = Sequential([
    Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=(40, 100, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    BatchNormalization(),
    Conv2D(128, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.3),
    Dense(TARGET_CLASSES, activation='softmax')
])

cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
cnn_model.summary()

# Train CNN model
cnn_history = cnn_model.fit(X_train_cnn, y_train, validation_split=0.2, epochs=20, batch_size=32)

# Save CNN model
cnn_model.save("cnn_model_voice.h5")

# Test CNN model
print("Testing CNN model...")
start_time = time.time()
cnn_predictions = cnn_model.predict(X_test_cnn)
cnn_test_time = time.time() - start_time
cnn_pred_classes = np.argmax(cnn_predictions, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

# CNN evaluation
cnn_accuracy = accuracy_score(y_test_classes, cnn_pred_classes)
print(f"\nCNN Test Accuracy: {cnn_accuracy:.4f}")
print(f"CNN Test Time: {cnn_test_time:.4f} seconds")
print("CNN Classification Report:\n", classification_report(y_test_classes, cnn_pred_classes))

# Plot CNN confusion matrix
conf_matrix_cnn = confusion_matrix(y_test_classes, cnn_pred_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix_cnn, annot=True, fmt="d", cmap="Blues", xticklabels=le.classes_, yticklabels=le.classes_)
plt.title("Confusion Matrix - CNN")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

In this part of the code, a 2D Convolutional Neural Network (CNN) model is being built and trained for the classification task. Here's a breakdown of the process:

Building the CNN Model:

A Sequential model is used to build the CNN, which consists of layers stacked on top of each other.

The model starts with a 2D convolutional layer (Conv2D) with 64 filters, each with a kernel size of (3, 3). The activation function used is ReLU (relu). The input_shape is set to (40, 100, 1), which corresponds to the MFCC features with 40 coefficients and 100 time frames, with one channel (since the features are grayscale).

The first convolutional layer is followed by a MaxPooling2D layer with a pool size of (2, 2), which reduces the spatial dimensions (height and width) of the feature maps.

A BatchNormalization layer is added to normalize the activations of the previous layer to speed up training and make the model more stable.

Another convolutional layer (Conv2D) is added with 128 filters and a (3, 3) kernel, followed by another MaxPooling2D layer.

After the convolutional and pooling layers, a Flatten layer is used to flatten the output from the previous layers into a one-dimensional vector.

A dense layer with 256 units and ReLU activation is added next, followed by a Dropout layer (with a dropout rate of 0.3) to prevent overfitting by randomly setting some of the input units to 0 during training.

Finally, the output layer consists of a dense layer with a number of units equal to TARGET_CLASSES (which is 27 in this case) and a softmax activation function. This ensures that the output represents the probabilities for each class.

Compiling the Model:

The model is compiled using the Adam optimizer, categorical cross-entropy as the loss function (since this is a multi-class classification problem), and accuracy as the evaluation metric.

Training the CNN Model:

The model is trained using the fit function, where the training data (X_train_cnn and y_train) is used for training. The validation data is obtained by splitting the training data further (20% validation). The model is trained for 20 epochs with a batch size of 32.

Saving the Model:

After training, the model is saved as cnn_model_voice.h5 for future use.
Testing the CNN Model:
The model is tested using the test data (X_test_cnn and y_test). The time taken for testing is recorded. Predictions are made on the test set, and the predicted class labels are obtained by applying np.argmax to the predicted probabilities.
The accuracy of the model on the test set is calculated using accuracy_score, and the classification report is printed, which includes precision, recall, and F1-score for each class.
Confusion Matrix:
A confusion matrix is created to visualize the performance of the model. The confusion matrix is plotted using seaborn to show how well the model is performing in terms of true positive, true negative, false positive, and false negative predictions across all classes.

Results

```
Loading data...
Building 2D CNN model...
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 38, 98, 64)	640
max_pooling2d (MaxPooling2D)	(None, 19, 49, 64)	0
batch_normalization (BatchNormalization)	(None, 19, 49, 64)	256
conv2d_1 (Conv2D)	(None, 17, 47, 128)	73,856
max_pooling2d_1 (MaxPooling2D)	(None, 8, 23, 128)	0
flatten (Flatten)	(None, 23552)	0
dense_2 (Dense)	(None, 256)	6,029,568
dropout_1 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 27)	6,939

Total params: 6,111,259 (23.31 MB)

Trainable params: 6,111,131 (23.31 MB)

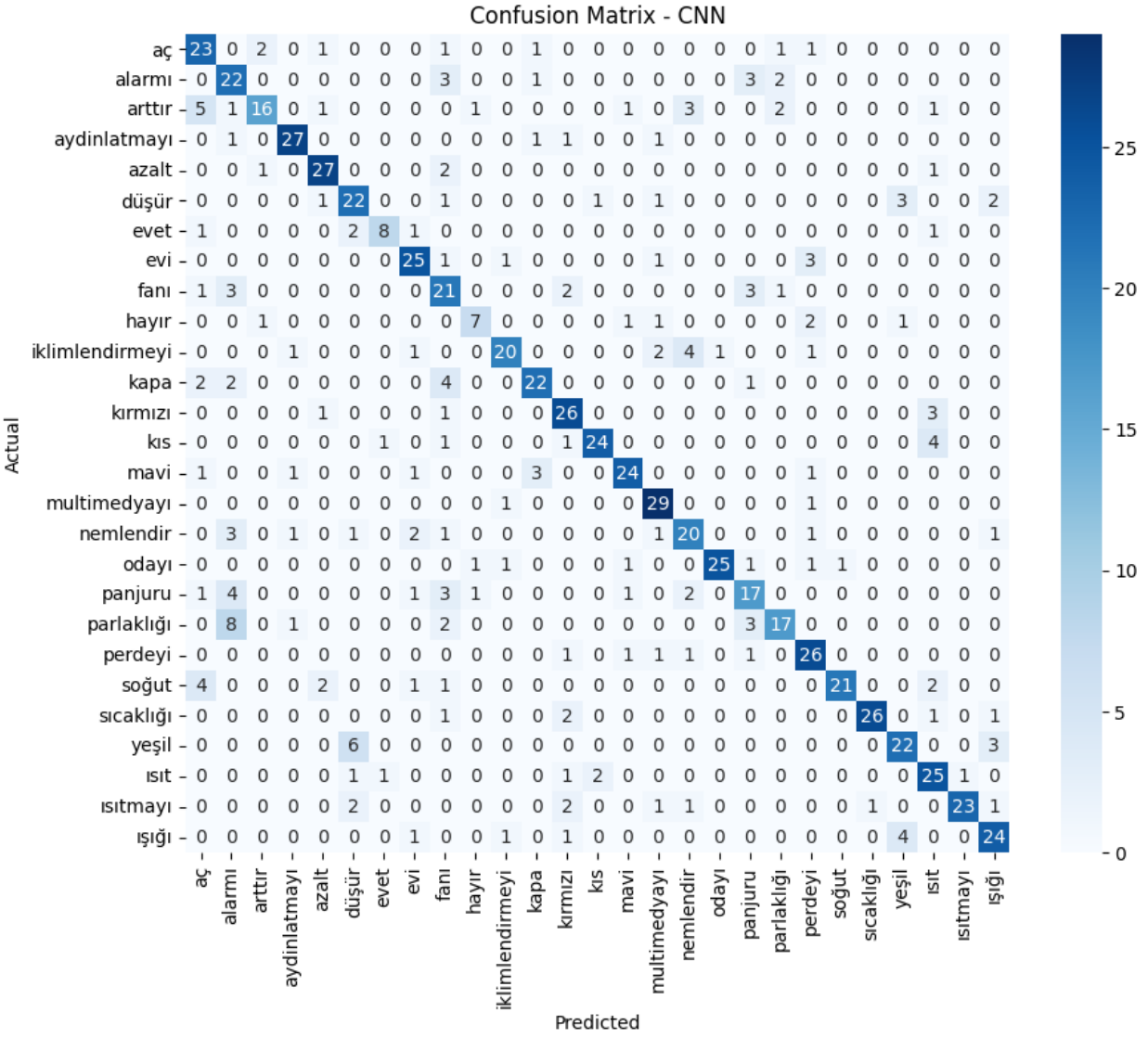
Non-trainable params: 128 (512.00 B)

```
Epoch 1/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 52s 620ms/step - accuracy: 0.0536 - loss: 4.4461 - val_accuracy: 0.0799 - val_loss: 3.2463
Epoch 2/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 77s 567ms/step - accuracy: 0.1068 - loss: 3.0559 - val_accuracy: 0.1897 - val_loss: 2.7546
Epoch 3/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 83s 575ms/step - accuracy: 0.2454 - loss: 2.5374 - val_accuracy: 0.4091 - val_loss: 2.1008
Epoch 4/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 81s 565ms/step - accuracy: 0.3536 - loss: 2.1022 - val_accuracy: 0.4671 - val_loss: 1.6917
Epoch 5/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 83s 583ms/step - accuracy: 0.5060 - loss: 1.6416 - val_accuracy: 0.5737 - val_loss: 1.4917
Epoch 6/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 82s 582ms/step - accuracy: 0.5627 - loss: 1.3826 - val_accuracy: 0.6489 - val_loss: 1.2442
Epoch 7/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 81s 569ms/step - accuracy: 0.6110 - loss: 1.1782 - val_accuracy: 0.6536 - val_loss: 1.1261
Epoch 8/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 81s 548ms/step - accuracy: 0.6808 - loss: 0.9403 - val_accuracy: 0.6787 - val_loss: 1.0367
Epoch 9/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 46s 571ms/step - accuracy: 0.7109 - loss: 0.8604 - val_accuracy: 0.6865 - val_loss: 1.0763
Epoch 10/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 49s 617ms/step - accuracy: 0.7577 - loss: 0.7327 - val_accuracy: 0.7179 - val_loss: 0.9799
Epoch 11/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 78s 565ms/step - accuracy: 0.7635 - loss: 0.6814 - val_accuracy: 0.7210 - val_loss: 0.9557
Epoch 12/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 83s 583ms/step - accuracy: 0.8014 - loss: 0.6231 - val_accuracy: 0.7226 - val_loss: 0.9846
Epoch 13/20
...
Epoch 19/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 42s 523ms/step - accuracy: 0.8462 - loss: 0.4338 - val_accuracy: 0.7461 - val_loss: 0.9993
Epoch 20/20
80/80 ━━━━━━━━━━━━━━━━━━━━ 43s 540ms/step - accuracy: 0.8506 - loss: 0.4003 - val_accuracy: 0.7680 - val_loss: 0.9296
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We
Testing CNN model...
25/25 ━━━━━━━━━━━━━━━━━━━━ 3s 108ms/step

CNN Test Accuracy: 0.7381
CNN Test Time: 5.1883 seconds
CNN Classification Report:
              precision    recall  f1-score   support

           0       0.61      0.77      0.68        30
           1       0.50      0.71      0.59        31
           2       0.80      0.52      0.63        31
           3       0.87      0.87      0.87        31
           4       0.82      0.87      0.84        31
           5       0.65      0.71      0.68        31
           6       0.80      0.62      0.70        13
           7       0.76      0.81      0.78        31
           8       0.49      0.68      0.57        31
           9       0.70      0.54      0.61        13
          10       0.83      0.67      0.74        30
          11       0.79      0.71      0.75        31
          12       0.70      0.84      0.76        31
          13       0.89      0.77      0.83        31
          14       0.83      0.77      0.80        31
          15       0.76      0.94      0.84        31
          16       0.65      0.65      0.65        31
...
    accuracy                           0.74       798
   macro avg       0.76      0.73      0.74       798
weighted avg       0.76      0.74      0.74       798
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...


Confusion Matrix - CNN
Actual
Predicted
aç
alarmı
arttır
aydinlatmayı
azalt
düşür
evet
evi
fanı
hayır
iklimlendirmeyi
kapa
kırmızı
kıs
mavi
multimedyayı
nemlendir
odayı
panjuru
parlaklığı
perdeyi
soğut
sıcaklığı
yeşil
ısıt
ısıtmayı
ışığı
0
5
10
15
20
25

Predictive Maintenance Dataset

```python
import numpy as np
import pandas as pd
import os
import time
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Bidirectional, Input, Conv2D, MaxPooling2D, BatchNormalization, Flatten
from tensorflow.keras.utils import to_categorical

# Define dataset path
data_path = "/content/propeller_dataset/output"

# Feature Extraction for Sensor Data
def extract_sensor_features(df):
    return np.hstack([
        df.mean(axis=0),
        df.std(axis=0),
        df.min(axis=0),
        df.max(axis=0),
    ])
```

The provided code begins by importing essential libraries for data manipulation, machine learning, and deep learning tasks. Libraries such as numpy, pandas, sklearn, and imblearn are used for handling and processing data, performing machine learning operations, and dealing with imbalanced classes. Deep learning models are built using tensorflow.keras, while matplotlib and seaborn are employed for visualizing results, particularly confusion matrices. The dataset path is specified as data_path, which is the location of the propeller dataset.

The core of the code lies in the feature extraction function extract_sensor_features(df). This function takes a DataFrame df as input, representing sensor data collected over time, and calculates four key statistical features for each sensor: the mean, standard deviation, minimum, and maximum. These features summarize the data in terms of central tendencies (mean), variation (standard deviation), and boundaries (min and max), providing a concise summary of the sensor readings. These extracted features are then combined into a single feature vector for each sample by horizontally stacking them (np.hstack). This step enables the transformation of raw sensor data into a more structured and informative format that can be used for further machine learning or deep learning tasks.

Overall, the function serves as a critical preprocessing step, converting raw sensor data into meaningful features that capture both the distribution and the range of the sensor values, which are essential for the classification or prediction tasks ahead.

```python
sensor_data = []
sensor_labels = []
speed_ranges = []
for file in os.listdir(data_path):
    file_path = os.path.join(data_path, file)
    df = pd.read_csv(file_path)
    features = extract_sensor_features(df.iloc[:, 1:-1])
    label = df.iloc[0, -1]
    speed_range = df.iloc[0, 0]  # Assuming speed range is in the first column
    sensor_data.append(features)
    sensor_labels.append(label)
    speed_ranges.append(speed_range)

sensor_data = np.array(sensor_data)
sensor_labels = np.array(sensor_labels)
speed_ranges = np.array(speed_ranges)

# Encoding sensor labels
label_encoder_sensor = LabelEncoder()
sensor_labels_encoded = label_encoder_sensor.fit_transform(sensor_labels)
label_map = {i: label for i, label in enumerate(label_encoder_sensor.classes_)}

# Encoding speed range labels
label_encoder_speed = LabelEncoder()
speed_ranges_encoded = label_encoder_speed.fit_transform(speed_ranges)

# Split dataset
X_train, X_test, y_train, y_test, speed_train, speed_test = train_test_split(
    sensor_data, sensor_labels_encoded, speed_ranges_encoded, test_size=0.2, stratify=sensor_labels_encoded, random_state=42
)

# Apply SMOTE to balance the dataset
smote = SMOTE(random_state=42)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)

# Normalize data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_balanced)
X_test_scaled = scaler.transform(X_test)
```

The provided code processes sensor data by reading CSV files from a specified directory (data_path). For each file, the data is loaded into a DataFrame, and a feature extraction function (extract_sensor_features) is applied to extract key statistical features (mean, standard deviation, minimum, and maximum) from the sensor readings. These extracted features are stored in sensor_data, while the corresponding sensor labels and speed ranges are stored in sensor_labels and speed_ranges, respectively. The speed range is assumed to be located in the first column of each CSV file, while the sensor label is stored in the last column.
After the data is collected, the labels are encoded using LabelEncoder from sklearn. This step transforms the sensor labels into numerical values that can be used for machine learning tasks. A separate encoding process is applied to the speed range labels. The dataset is then split into training and test sets using train_test_split, with a stratified split to ensure that the class distribution is maintained in both sets. Additionally, SMOTE (Synthetic Minority Over-sampling Technique) is applied to balance the class distribution in the training set by generating synthetic examples for the minority classes.
To ensure that the features are on a similar scale, the data is normalized using StandardScaler. The training data is scaled using the fit_transform method, while the test data is scaled using the transform method, ensuring that both sets are standardized with the same parameters. This prepares the data for further machine learning or deep learning model training.

```python
# Define models
models = {
    "Decision Tree": DecisionTreeClassifier(max_depth=30, class_weight='balanced', random_state=42),
    "Random Forest": RandomForestClassifier(n_estimators=500, max_depth=30, min_samples_split=5, class_weight='balanced', random_state=42),
    "K-Nearest Neighbors": KNeighborsClassifier(n_neighbors=5, weights='distance'),
    "Naive Bayes": GaussianNB(),
    "SVM": SVC(kernel='rbf', class_weight='balanced', random_state=42),
    "Logistic Regression": LogisticRegression(max_iter=1000, class_weight='balanced', random_state=42)
}

# Perform 10-fold cross-validation
print("Performing stratified 10-fold cross-validation...")
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

best_model = None
best_accuracy = 0
for name, model in models.items():
    accuracies = []
    for train_index, val_index in skf.split(X_train_scaled, y_train_balanced):
        X_tr, X_val = X_train_scaled[train_index], X_train_scaled[val_index]
        y_tr, y_val = y_train_balanced[train_index], y_train_balanced[val_index]

        model.fit(X_tr, y_tr)
        predictions = model.predict(X_val)
        acc = accuracy_score(y_val, predictions)
        accuracies.append(acc)

    mean_accuracy = np.mean(accuracies)
    print(f"{name} Mean CV Accuracy: {mean_accuracy:.4f}")

    if mean_accuracy > best_accuracy:
        best_model = model
        best_accuracy = mean_accuracy

print("\nBest Model:", best_model)
```

The code defines a set of machine learning models for classification, including Decision Tree, Random Forest, K-Nearest Neighbors, Naive Bayes, SVM, and Logistic Regression. Each model is configured with specific hyperparameters, such as maximum depth for Decision Tree and Random Forest, or the number of neighbors for K-Nearest Neighbors. The models are then evaluated using 10-fold stratified cross-validation to ensure that each model's performance is assessed across different splits of the training data.

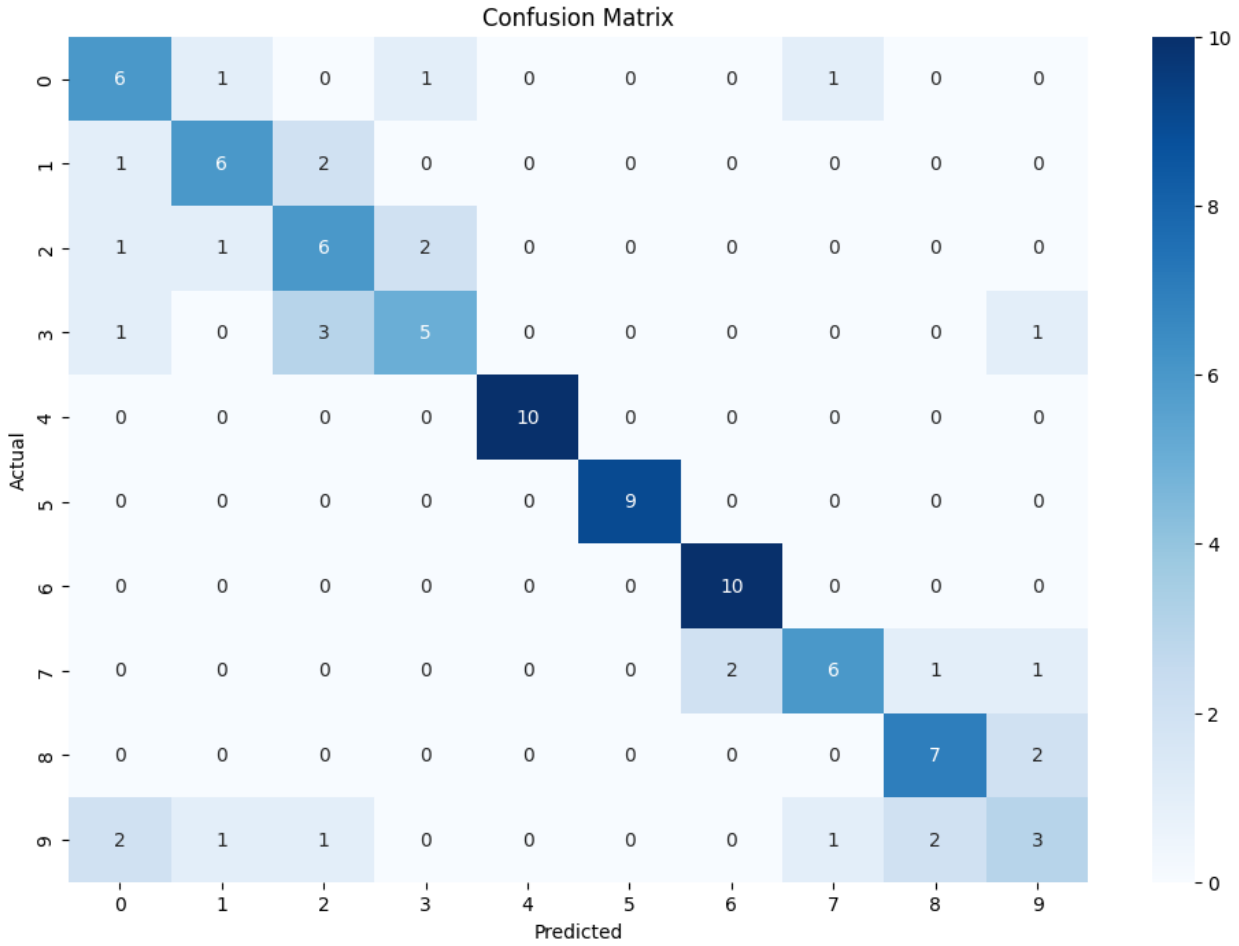
In the cross-validation process, the dataset is divided into 10 folds, and for each fold, a model is trained on 90% of the data and tested on the remaining 10%. This process is repeated for all 10 folds, and the accuracy for each fold is computed. The mean accuracy across all folds is calculated for each model. The model with the highest mean accuracy is selected as the best model. Finally, the best-performing model is printed along with its accuracy, allowing for comparison of different models' performance.

```python
# Test set evaluation
start_time = time.time()
final_predictions = best_model.predict(X_test_scaled)
test_time = time.time() - start_time

print(f"\nTest Accuracy: {accuracy_score(y_test, final_predictions):.4f}")
print(f"Test Time: {test_time:.4f} seconds")
print("Classification Report:\n", classification_report(y_test, final_predictions))

# Generate confusion matrix
conf_matrix = confusion_matrix(y_test, final_predictions)
plt.figure(figsize=(12, 8))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

In this section of the code, the best model selected during the 10-fold cross-validation is used to make predictions on the test set. The time taken for the model to make predictions is recorded. Then, the accuracy of the model on the test set is computed using the accuracy_score function, and the time it took to perform the prediction is printed.
The classification report is also generated, which provides more detailed performance metrics, such as precision, recall, and F1-score, for each class. This allows for a better understanding of how well the model performs on each of the classes.
Additionally, a confusion matrix is generated to visualize the performance of the model. The confusion matrix shows the true positives, false positives, true negatives, and false negatives for each class. A heatmap is used to display the confusion matrix for easy interpretation. This step helps in understanding where the model is making errors and which classes are being misclassified.

Results

```
Performing stratified 10-fold cross-validation...
Decision Tree Mean CV Accuracy: 0.5051
Random Forest Mean CV Accuracy: 0.6564
K-Nearest Neighbors Mean CV Accuracy: 0.4744
Naive Bayes Mean CV Accuracy: 0.1949
SVM Mean CV Accuracy: 0.3231
Logistic Regression Mean CV Accuracy: 0.4462

Best Model: RandomForestClassifier(class_weight='balanced', max_depth=30,
                       min_samples_split=5, n_estimators=500, random_state=42)

Test Accuracy: 0.7083
Test Time: 0.0377 seconds
Classification Report:
              precision    recall  f1-score   support

           0       0.55      0.67      0.60         9
           1       0.67      0.67      0.67         9
           2       0.50      0.60      0.55        10
           3       0.62      0.50      0.56        10
           4       1.00      1.00      1.00        10
           5       1.00      1.00      1.00         9
           6       0.83      1.00      0.91        10
           7       0.75      0.60      0.67        10
           8       0.70      0.78      0.74         9
...
    accuracy                           0.71        96
   macro avg       0.70      0.71      0.70        96
weighted avg       0.70      0.71      0.70        96

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```




```python
# Reshape for CNN
X_train_cnn = X_train_scaled.reshape(X_train_scaled.shape[0], 8, 4, 1)
X_test_cnn = X_test_scaled.reshape(X_test_scaled.shape[0], 8, 4, 1)

y_train_cat = to_categorical(y_train_balanced, num_classes=len(label_map))
y_test_cat = to_categorical(y_test, num_classes=len(label_map))

# Build 2D CNN Model
cnn_model = Sequential([
    Conv2D(64, kernel_size=(2,2), activation='relu', input_shape=(8, 4, 1)),
    MaxPooling2D(pool_size=(2,1)),  # Adjust pooling to preserve width
    BatchNormalization(),

    Conv2D(128, kernel_size=(2,2), activation='relu'),
    MaxPooling2D(pool_size=(2,1)),  # Prevent too much shrinking

    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.3),
    Dense(len(label_map), activation='softmax')
])

cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
cnn_model.summary()

# Train CNN model
cnn_model.fit(X_train_cnn, y_train_cat, validation_split=0.2, epochs=100, batch_size=32)
cnn_model.save("cnn_model.h5")

# Test CNN model
start_time = time.time()
cnn_predictions = cnn_model.predict(X_test_cnn)
cnn_test_time = time.time() - start_time
cnn_pred_classes = np.argmax(cnn_predictions, axis=1)

print(f"\nCNN Test Accuracy: {accuracy_score(y_test, cnn_pred_classes):.4f}")
print(f"CNN Test Time: {cnn_test_time:.4f} seconds")
print("CNN Classification Report:\n", classification_report(y_test, cnn_pred_classes))
```

In this section, the data is reshaped to fit the requirements of the 2D Convolutional Neural Network (CNN). The training and testing data are reshaped into a format suitable for CNN input, where each sample is structured as a 2D array with a single channel. The labels are also one-hot encoded using

to_categorical to convert them into categorical format, making them compatible with the model's output layer.
Next, a CNN model is constructed using a series of convolutional layers (Conv2D) followed by max-pooling layers (MaxPooling2D) to extract spatial features from the input data. Batch normalization is applied to normalize the activations in each layer, and dropout is used to prevent overfitting. After flattening the output from the convolutional layers, dense layers are used to perform classification, with the final output layer using a softmax activation function to output the predicted class probabilities.
The model is compiled with the Adam optimizer and categorical crossentropy loss function, and the training begins with a validation split of 20%. After training the model for 100 epochs, the CNN model is saved for future use.
Once the model is trained, it is evaluated on the test set. The time taken for predictions is measured, and the accuracy score is computed. Additionally, a classification report is generated to provide metrics such as precision, recall, and F1-score for each class. This helps in evaluating the model's performance in more detail. The final output includes the test accuracy, test time, and the classification report.

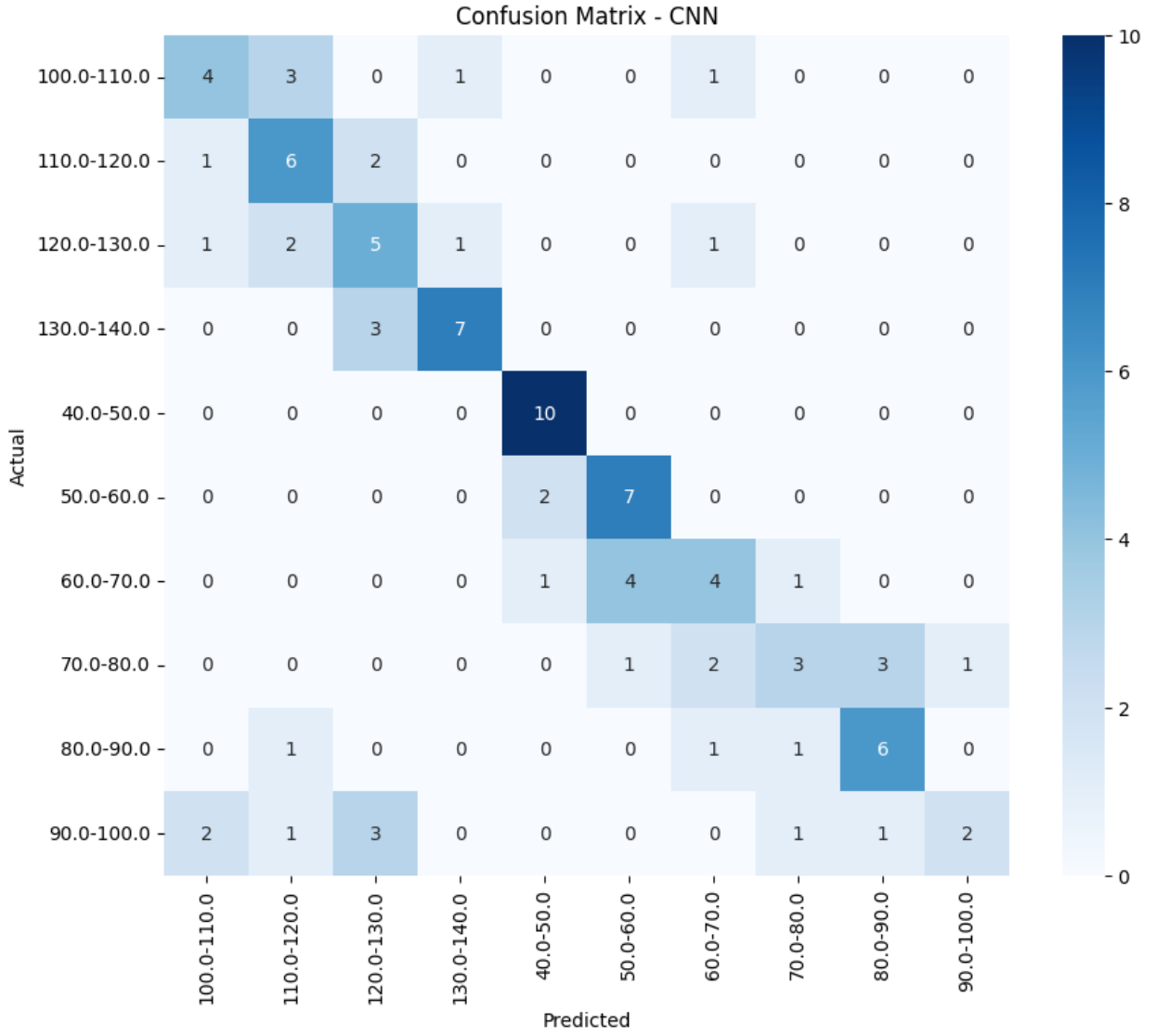
Results

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 7, 3, 64)	320
max_pooling2d (MaxPooling2D)	(None, 3, 3, 64)	0
batch_normalization (BatchNormalization)	(None, 3, 3, 64)	256
conv2d_1 (Conv2D)	(None, 2, 2, 128)	32,896
max_pooling2d_1 (MaxPooling2D)	(None, 1, 2, 128)	0
flatten (Flatten)	(None, 256)	0
dense (Dense)	(None, 256)	65,792
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2,570

```
Total params: 101,834 (397.79 KB)

Trainable params: 101,706 (397.29 KB)

Non-trainable params: 128 (512.00 B)

Epoch 1/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 4s 46ms/step - accuracy: 0.1286 - loss: 2.4358 - val_accuracy: 0.2051 - val_loss: 2.2976
Epoch 2/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.3821 - loss: 1.8810 - val_accuracy: 0.2949 - val_loss: 2.2455
Epoch 3/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.4616 - loss: 1.4739 - val_accuracy: 0.4231 - val_loss: 2.1881
Epoch 4/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.5144 - loss: 1.4335 - val_accuracy: 0.3974 - val_loss: 2.1563
Epoch 5/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.4355 - loss: 1.4133 - val_accuracy: 0.4744 - val_loss: 2.0996
Epoch 6/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.5787 - loss: 1.2145 - val_accuracy: 0.3974 - val_loss: 2.0852
Epoch 7/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.5722 - loss: 1.1813 - val_accuracy: 0.4487 - val_loss: 2.0491
Epoch 8/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.5794 - loss: 1.0618 - val_accuracy: 0.4744 - val_loss: 2.0221
Epoch 9/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.5752 - loss: 1.1031 - val_accuracy: 0.4744 - val_loss: 1.9752
Epoch 10/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.6170 - loss: 1.0149 - val_accuracy: 0.5513 - val_loss: 1.9300
Epoch 11/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.6899 - loss: 0.9130 - val_accuracy: 0.4872 - val_loss: 1.9011
Epoch 12/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.6322 - loss: 0.9678 - val_accuracy: 0.5769 - val_loss: 1.8649
Epoch 13/100
...
Epoch 99/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.9814 - loss: 0.0863 - val_accuracy: 0.6026 - val_loss: 2.7761
Epoch 100/100
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.9682 - loss: 0.0934 - val_accuracy: 0.6026 - val_loss: 2.5836
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend usi
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
```

```
CNN Test Accuracy: 0.5625
CNN Test Time: 0.3876 seconds
CNN Classification Report:
              precision    recall  f1-score   support

           0       0.50      0.44      0.47         9
           1       0.46      0.67      0.55         9
           2       0.38      0.50      0.43        10
           3       0.78      0.70      0.74        10
           4       0.77      1.00      0.87        10
           5       0.58      0.78      0.67         9
           6       0.44      0.40      0.42        10
           7       0.50      0.30      0.38        10
           8       0.60      0.67      0.63         9
           9       0.67      0.20      0.31        10

    accuracy                           0.56        96
   macro avg       0.57      0.57      0.55        96
weighted avg       0.57      0.56      0.54        96
```



```python
# LSTM Model
print("Building LSTM model...")
# Reshape for LSTM
time_steps = 8  # Adjust based on data
n_features = X_train_scaled.shape[1] // time_steps

X_train_lstm = X_train_scaled.reshape(X_train_scaled.shape[0], time_steps, n_features)
X_test_lstm = X_test_scaled.reshape(X_test_scaled.shape[0], time_steps, n_features)

# Convert Labels to Categorical
y_train_cat = to_categorical(y_train_balanced, num_classes=len(label_map))
y_test_cat = to_categorical(y_test, num_classes=len(label_map))

# Define Fixed LSTM Model
lstm_model = Sequential([
    Input(shape=(time_steps, n_features)),  # ✅ Fixed: Explicit input shape
    Bidirectional(LSTM(256, return_sequences=True)),
    Bidirectional(LSTM(256)),
    Dropout(0.5),
    Dense(512, activation='relu'),
    Dense(len(label_map), activation='softmax')
])

lstm_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
lstm_model.summary()

# Train LSTM Model
lstm_history = lstm_model.fit(X_train_lstm, y_train_cat, validation_split=0.2, epochs=100, batch_size=16)

# Save Model
lstm_model.save("lstm_model.h5")
```

In this part of the process, an LSTM model is built for sequential data analysis. First, the training and testing data are reshaped to fit the LSTM's input requirements. The data is structured into sequences, where each sequence consists of a specific number of time steps (8 in this case) and corresponding features. The number of features is determined by dividing the total number of features by the time steps. This reshaped data is then ready for input into the LSTM model.

The labels are converted into categorical format using to_categorical, which is necessary for the model to perform classification with a softmax output layer. The LSTM model itself is defined with two bidirectional LSTM layers, each with 256 units. These bidirectional layers process the data in both forward and backward directions, allowing the model to capture dependencies in the data more effectively. A dropout layer with a rate of 0.5 is included to help prevent overfitting.

The model is compiled with the Adam optimizer and categorical crossentropy loss function, suitable for multi-class classification. The model's architecture is displayed, summarizing the layers and the number of parameters.

The model is then trained on the reshaped data for 100 epochs with a batch size of 16, using a validation split of 20%. After training, the model is saved for future use. This LSTM model is designed to capture temporal dependencies in the data, which is particularly useful for time-series or sequential datasets like sensor readings or audio features.

```python
# Test LSTM Model
start_time = time.time()
lstm_predictions = lstm_model.predict(X_test_lstm)
lstm_test_time = time.time() - start_time
lstm_pred_classes = np.argmax(lstm_predictions, axis=1)

# Evaluate LSTM Model
lstm_accuracy = accuracy_score(y_test, lstm_pred_classes)
print(f"\nLSTM Test Accuracy: {lstm_accuracy:.4f}")
print(f"LSTM Test Time: {lstm_test_time:.4f} seconds")
print("LSTM Classification Report:\n", classification_report(y_test, lstm_pred_classes))

# Plot LSTM Confusion Matrix
conf_matrix_lstm = confusion_matrix(y_test, lstm_pred_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix_lstm, annot=True, fmt="d", cmap="Blues",
            xticklabels=label_map.values(), yticklabels=label_map.values())
plt.title("Confusion Matrix - LSTM")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

In this final step, the performance of the trained LSTM model is evaluated on the test set. The test predictions are made by passing the test data (X_test_lstm) through the model. The time taken for predictions is measured, and the accuracy of the model is calculated by comparing the predicted classes with the actual test labels. The accuracy score is printed along with the time it took to make the predictions.
Additionally, the classification report is generated, which provides a detailed evaluation of the model's performance across all classes, including precision, recall, and F1-score. Finally, a confusion matrix is plotted using seaborn to visually assess how well the model performed in terms of correctly and incorrectly classified samples. The heatmap of the confusion matrix helps identify which classes the model struggled with the most. This gives further insights into where the model could potentially be improved.

Results

Layer (type)	Output Shape	Param #
bidirectional (Bidirectional)	(None, 8, 512)	534,528
bidirectional_1 (Bidirectional)	(None, 512)	1,574,912
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 512)	262,656
dense_3 (Dense)	(None, 10)	5,130

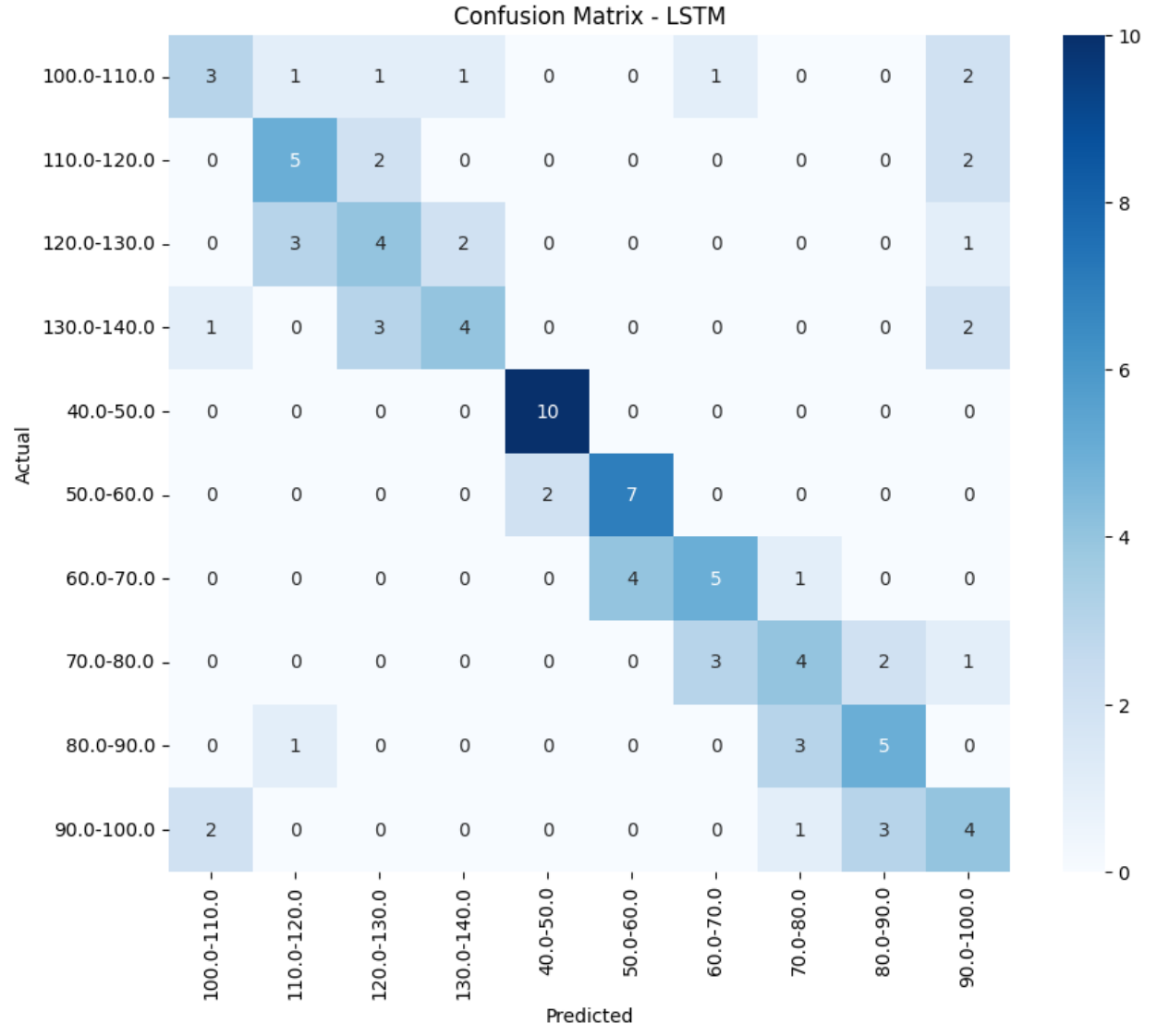
Total params: 2,377,226 (9.07 MB)

Trainable params: 2,377,226 (9.07 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 12s 253ms/step - accuracy: 0.1169 - loss: 2.2565 - val_accuracy: 0.1026 - val_loss: 2.0976
Epoch 2/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 136ms/step - accuracy: 0.2610 - loss: 1.9635 - val_accuracy: 0.1538 - val_loss: 1.9183
Epoch 3/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 131ms/step - accuracy: 0.2799 - loss: 1.8283 - val_accuracy: 0.3333 - val_loss: 1.8277
Epoch 4/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 135ms/step - accuracy: 0.3207 - loss: 1.6655 - val_accuracy: 0.2436 - val_loss: 1.7649
Epoch 5/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 6s 166ms/step - accuracy: 0.3708 - loss: 1.5791 - val_accuracy: 0.3974 - val_loss: 1.7368
Epoch 6/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 135ms/step - accuracy: 0.3141 - loss: 1.6555 - val_accuracy: 0.3205 - val_loss: 1.8322
Epoch 7/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 5s 139ms/step - accuracy: 0.3897 - loss: 1.5194 - val_accuracy: 0.5256 - val_loss: 1.6248
Epoch 8/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 6s 168ms/step - accuracy: 0.4648 - loss: 1.3577 - val_accuracy: 0.3974 - val_loss: 1.7481
Epoch 9/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 138ms/step - accuracy: 0.4024 - loss: 1.4630 - val_accuracy: 0.3846 - val_loss: 1.7211
Epoch 10/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 6s 160ms/step - accuracy: 0.4346 - loss: 1.3533 - val_accuracy: 0.3974 - val_loss: 1.5876
Epoch 11/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 5s 146ms/step - accuracy: 0.4422 - loss: 1.3232 - val_accuracy: 0.3974 - val_loss: 1.7986
Epoch 12/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 5s 137ms/step - accuracy: 0.4827 - loss: 1.3157 - val_accuracy: 0.3846 - val_loss: 1.8581
Epoch 13/100
...
Epoch 99/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 5s 137ms/step - accuracy: 0.9519 - loss: 0.1549 - val_accuracy: 0.5128 - val_loss: 2.9064
Epoch 100/100
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 141ms/step - accuracy: 0.9387 - loss: 0.1995 - val_accuracy: 0.5256 - val_loss: 3.0321
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We
3/3 ━━━━━━━━━━━━━━━━━━━━ 1s 44ms/step
```

```
LSTM Test Accuracy: 0.5312
LSTM Test Time: 1.3183 seconds
LSTM Classification Report:
              precision    recall  f1-score   support

           0       0.50      0.33      0.40         9
           1       0.50      0.56      0.53         9
           2       0.40      0.40      0.40        10
           3       0.57      0.40      0.47        10
           4       0.83      1.00      0.91        10
           5       0.64      0.78      0.70         9
           6       0.56      0.50      0.53        10
           7       0.44      0.40      0.42        10
           8       0.50      0.56      0.53         9
           9       0.33      0.40      0.36        10

    accuracy                           0.53        96
   macro avg       0.53      0.53      0.52        96
weighted avg       0.53      0.53      0.52        96
```



REFERENCES

[1] Lecture Notes

[2] Google searching

[3] Artificial Intelligence Services