Stuttering Detection Using CNN Model: Approach and Techniques

This document provides a detailed explanation of the steps to generate prediction labels for stuttering, normal speech, or noise detection from audio files. It covers the process of audio preprocessing, feature extraction, normalization, model structure, training, and evaluation.

1. Steps to Generate Prediction Labels

- 1. **Audio Preprocessing and Feature Extraction**
- 2. **Normalization of Features**
- 3. **Prepare Data for Model Input**
- 4. **Prediction Using Pre-trained Model**
- 5. **Weighted Majority Voting for Final Label**

Step 1: Audio Preprocessing and Feature Extraction

The first step is to preprocess the audio and extract relevant features that can be used for classification. This involves loading the audio file, segmenting it into smaller chunks, and adding noise for data augmentation.

Features extracted include MFCCs, MFCC Delta, MFCC Delta-Delta, Spectral Centroid, and STFT, which together form a feature vector for each audio segment.

Step 2: Normalization of Features

Normalization ensures that all features have the same scale, avoiding bias in the model. Techniques include Standardization (mean=0, variance=1) and MinMax scaling.

Step 3: Prepare Data for Model Input

Segmentation features are stored in CSV files, ensuring they can be processed effectively. Short audio files are handled separately to avoid data loss.

Step 4: Prediction Using Pre-trained Model

The trained model is loaded, and the reshaped features are used for prediction. The model outputs probabilities for each class (stuttering, normal, noise).

Step 5: Weighted Majority Voting for Final Label

Using weighted majority voting, final predictions are determined based on the accumulated weights of each class. This ensures robust decision-making for long audio files.

2. CNN Model Structure and Influencing Factors

The Convolutional Neural Network (CNN) model for stuttering detection is composed of multiple convolutional layers, batch normalization, and max-pooling layers, followed by dense layers for classification.

Key factors influencing model performance include:

- 1. **Number of Filters and Kernel Size**: Determines how features are learned from audio signals.
- 2. **Batch Normalization**: Helps in stabilizing learning and faster convergence.
- 3. **Dropout Layers**: Prevents overfitting by randomly ignoring some neurons during training.
- 4. **Pooling Layers**: Reduce the dimensionality of features, focusing on essential information.
- 5. **Regularization**: L2 regularization is used to penalize large weights, promoting generalization.

- **Dropout (0.4):** Increased rate to ensure model does not overfit complex features.

4. **Flatten Layer:**
   - Converts the 2D feature maps into a 1D vector for the fully connected layers.

5. **Dense Layers:**
   - **First Dense Layer (128 units):** Applies non-linear transformations to learn complex relationships, uses L2 regularization to penalize large weights.
   - **Dropout (0.5):** High dropout rate for regularization.
   - **Second Dense Layer (64 units):** Additional non-linear transformations with regularization.
   - **Dropout (0.3):** Provides extra regularization.

6. **Output Layer:**
   - **Dense Layer (3 units):** Outputs three probabilities corresponding to the classes (stuttering, normal speech, noise). Uses softmax activation for multi-class classification.

```
# Split the training data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.15,
random_state=123, stratify=y_train)

# Convert labels to categorical format (one-hot encoding)
y_train = tf.keras.utils.to_categorical(y_train, num_classes=3)
y_val = tf.keras.utils.to_categorical(y_val, num_classes=3)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=3)

# Reshape the data to 3D (samples, time steps, features) for CNN input
X_train = np.expand_dims(X_train, axis=2)
X_val = np.expand_dims(X_val, axis=2)
X_test = np.expand_dims(X_test, axis=2)

# Build the improved CNN model
def build_improved_cnn_model():
    model = Sequential()

    # First convolutional block
    model.add(Conv1D(filters=64, kernel_size=7, activation='relu', padding='same',
input_shape=(X_train.shape[1], 1)))
    model.add(BatchNormalization())
    model.add(MaxPooling1D(pool_size=2, padding='same'))
    model.add(Dropout(0.2))

    # Second convolutional block
    model.add(Conv1D(filters=128, kernel_size=6, activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling1D(pool_size=2, padding='same'))
    model.add(Dropout(0.3))

    # Third convolutional block
    model.add(Conv1D(filters=256, kernel_size=3, activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling1D(pool_size=2, padding='same'))
    model.add(Dropout(0.4))

    # Flatten for dense layers
    model.add(Flatten())

    # Dense layers
    model.add(Dense(128, activation='relu',
```

```
kernel_regularizer=tf.keras.regularizers.l2(0.01)))
  model.add(Dropout(0.5))
  model.add(Dense(64, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01)))
  model.add(Dropout(0.3))

  # Output layer
  model.add(Dense(3, activation='softmax'))

  # Compile the model
  model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),
          loss='categorical_crossentropy', metrics=['accuracy'])
  model.summary()
  return model

# Build the model
model = build_improved_cnn_model()
```

Data Processing for Prediction Code

```
# Load test data for predictions
test = pd.read_csv('/content/drive/MyDrive/Trail_1/n40seg1over0.5-test.csv')
true_labels = test['label']
X_test = test.drop(columns=['label'])

# Reshape the test data
X_test = np.expand_dims(X_test, axis=2)

# Predict on the test data
predictions = model.predict(X_test, steps=int(np.ceil(X_test.shape[0] / 1.0)))

# Convert predicted probabilities to class labels
predicted_labels = np.argmax(predictions, axis=1)

# Save predicted labels in the test dataframe
test['predicted_label'] = predicted_labels

# Save the predictions to a CSV file
test.to_csv('Test_Predictions.csv', index=False)
```

Model Training & Evaluation Code

```
# Define callbacks
```

```
early_stop = EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True)
model_checkpoint = ModelCheckpoint('/content/drive/MyDrive/model.keras',
monitor='val_loss', save_best_only=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5)

# Train the model
history = model.fit(X_train, y_train,
                    batch_size=64,
                    epochs=400,
                    validation_data=(X_val, y_val),
                    callbacks=[early_stop, model_checkpoint, reduce_lr],
                    verbose=1)

# Evaluate on the test data
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}")

# Plot loss curves
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Curves')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend()
plt.show()

# Plot accuracy curves
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy Curves')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend()
plt.show()
```