# Multilingual Name Classifier using RNN and LSTM

In the following code, I have developed a language classification model using Recurrent Neural Network (RNN) and Long Short Term Memory (LSTM) to predict the origin language of the input names. I have fine-tuned the hyperparameters to identify the most optimal model for this language classification task.

## Import necessary libraries

```
import os
import unicodedata
import string
import torch
import torch.nn as nn
import torch.optim as optim
from io import open
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
```

## Lists all possible letters and the "End of String" marker. Loaded the data from the files and performed data pre-processing

```
# All letters and the "EOS" (End of String) marker
all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)


file_paths = [
    r'/content/drive/MyDrive/DSP Assignments/Arabic.txt',
    r'/content/drive/MyDrive/DSP Assignments/Chinese.txt',
    r'/content/drive/MyDrive/DSP Assignments/Czech.txt',
    r'/content/drive/MyDrive/DSP Assignments/English.txt',
    r'/content/drive/MyDrive/DSP Assignments/French.txt',
    r'/content/drive/MyDrive/DSP Assignments/German.txt',
    r'/content/drive/MyDrive/DSP Assignments/Greek.txt',
    r'/content/drive/MyDrive/DSP Assignments/Irish.txt',
    r'/content/drive/MyDrive/DSP Assignments/Italian.txt',
    r'/content/drive/MyDrive/DSP Assignments/Japanese.txt',
    r'/content/drive/MyDrive/DSP Assignments/Russian.txt',
    r'/content/drive/MyDrive/DSP Assignments/Spanish.txt',
]

# Utility function to turn a Unicode string to plain ASCII
def unicode_to_ascii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

# Load data
category_lines = {}
all_categories = []

for file_path in file_paths:
    # Extract category (language) from the file name
    category = os.path.splitext(os.path.basename(file_path))[0]
    all_categories.append(category)
    lines = open(file_path, encoding='utf-8').read().strip().split('\n')
    lines = [unicode_to_ascii(line) for line in lines]
    category_lines[category] = lines

# Print out some samples to check the data
for category in all_categories:
    print(category, ": ", category_lines[category][:5])
```

```
n_categories = len(all_categories) # Number of categories (languages)
```

```
Arabic :  ['Khoury', 'Nahas', 'Daher', 'Gerges', 'Nazari']
Chinese :  ['Ang', 'AuYong', 'Bai', 'Ban', 'Bao']
Czech :  ['Abl', 'Adsit', 'Ajdrna', 'Alt', 'Antonowitsch']
English :  ['Healy', 'Heath', 'Heathcote', 'Heather', 'Heatley']
French :  ['Abel', 'Abraham', 'Adam', 'Albert', 'Allard']
German :  ['Abbing', 'Abel', 'Abeln', 'Abt', 'Achilles']
Greek :  ['Adamidis', 'Adamou', 'Agelakos', 'Akrivopoulos', 'Alexandropoulos']
Irish :  ['Adam', 'Ahearn', 'Aodh', 'Aodha', 'Aonghuis']
Italian :  ['Abandonato', 'Abatangelo', 'Abatantuono', 'Abate', 'Abategiovanni']
Japanese :  ['Abe', 'Abukara', 'Adachi', 'Aida', 'Aihara']
Russian :  ['Ababko', 'Abaev', 'Abagyan', 'Abaidulin', 'Abaidullin']
Spanish :  ['Abana', 'Abano', 'Abarca', 'Abaroa', 'Abascal']
```

Mounted the Google Drive to access the files in it.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

In the below code, I am preparing data for training and validation and also creating a custom dataset for organizing the data.

name_to_tensor: Converts names to tensors through one-hot encoding. Tensors have dimensions with max_length for name length, a batch size of 1, and n_letters as the character count.

category_to_tensor: Maps categories to Long tensors by their index in all_categories.

NamesDataset: A custom dataset class for data organization. It facilitates training and validation data management. The **getitem** is used for accessing specific name-category pairs, transforming names with name_to_tensor and categories with category_to_tensor.

```
# Function to convert name and category to tensor
def name_to_tensor(name, max_length):
    tensor = torch.zeros(max_length, 1, n_letters)
    for li, letter in enumerate(name):
        letter_index = all_letters.find(letter)
        tensor[li][0][letter_index] = 1
    return tensor

def category_to_tensor(category):
    return torch.tensor([all_categories.index(category)], dtype=torch.long)

# Creating a custom dataset class for names
class NamesDataset(Dataset):
    def __init__(self, category_lines, all_categories, max_length):
        self.names = []
        self.categories = []
        self.max_length = max_length
        for category in all_categories:
            for name in category_lines[category]:
                self.names.append(name)
                self.categories.append(category)

    def __len__(self):
        return len(self.names)

    def __getitem__(self, index):
        name = self.names[index]
        category = self.categories[index]
        name_tensor = name_to_tensor(name, self.max_length)
        category_tensor = category_to_tensor(category)
        return name_tensor, category_tensor
```

This code segment divides the dataset into training and validation sets for each category. DataLoaders are established for easy batching and shuffling during model training.

```
# Split data into training and validation sets
train_data = {}
val_data = {}
for category in all_categories:
    train, val = train_test_split(category_lines[category], test_size=0.2, random_state=42)
```

```
    train_data[category] = train
    val_data[category] = val

# Get the maximum name length for padding
max_length = max(len(name) for category in all_categories for name in category_lines[category])


# Create datasets and dataloaders
train_dataset = NamesDataset(train_data, all_categories, max_length)
val_dataset = NamesDataset(val_data, all_categories, max_length)
train_dataloader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=16, shuffle=False)
```

▼ Defining a function to create and train the RNN Model

```
# Updated RNN model with an additional RNN layer
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        # Initializing hidden state
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)

        # Forward pass through RNN layers
        out, _ = self.rnn(x, h0)

        # Taking the output from the last time step and pass it through the fully connected layer
        out = self.fc(out[:, -1, :])

        # Apply softmax to get probabilities
        out = self.softmax(out)
        return out
```

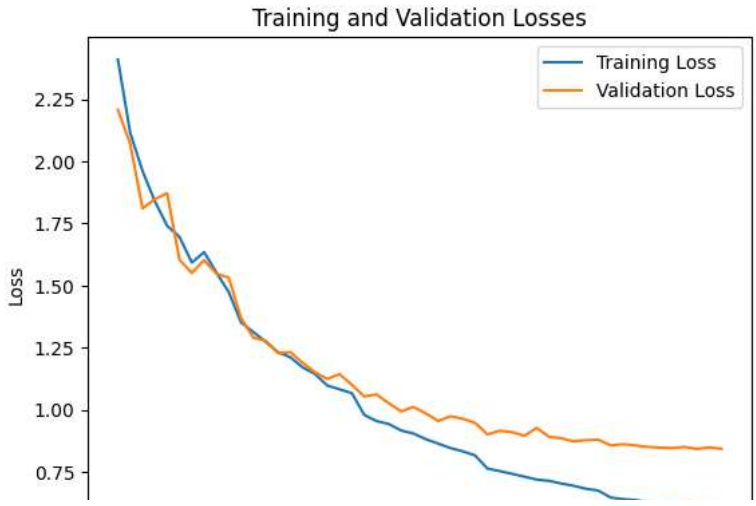
▼ Defining a function to create and train the LSTM Model

```
import torch.nn.functional as F

class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        # Initializing hidden state and cell state
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)

        # Forward pass through LSTM layers
        out, _ = self.lstm(x, (h0, c0))

        # Taking the output from the last time step and pass it through the fully connected layer
        out = self.fc(out[:, -1, :])

        # Apply softmax to get probabilities
        out = self.softmax(out)
        return out
```

▼ Training and Evaluation of the RNN Model with Varying Epochs

```
import matplotlib.pyplot as plt

# Hyperparameters
n_hidden = 128
n_categories = len(all_categories)
num_layers = 1
learning_rate = 0.001
num_epochs = 50

# Instantiate the model, loss function, and optimizer
model = RNN(n_letters, n_hidden, n_categories, num_layers)
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Learning rate scheduling
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5)

# Variables to keep track of losses for each epoch
training_losses = []
validation_losses = []

# Variable to keep track of the lowest validation loss
best_val_loss = float('inf')  # Initialize with infinity

# Training loop
for epoch in range(num_epochs):
    model.train()  # Set model to training mode
    total_loss = 0
    for names, categories in train_dataloader:
        optimizer.zero_grad()
        outputs = model(names.squeeze(2))
        loss = criterion(outputs, categories.squeeze(1))
        total_loss += loss.item()
        loss.backward()
        optimizer.step()

    train_loss = total_loss / len(train_dataloader)
    training_losses.append(train_loss)  # Store the training loss
    print(f'Epoch {epoch + 1}, Training Loss: {train_loss}')

    # Validation loop to compute the validation loss
    model.eval()  # Set model to evaluation mode
    val_loss = 0
    with torch.no_grad():
        for names, categories in val_dataloader:
            outputs = model(names.squeeze(2))
            loss = criterion(outputs, categories.squeeze(1))
            val_loss += loss.item()

    val_loss /= len(val_dataloader)
    validation_losses.append(val_loss)  # Store the validation loss
    print(f'Epoch {epoch + 1}, Validation Loss: {val_loss}')

    # Learning rate scheduling step
    scheduler.step()

    # Save the model if it has the lowest validation loss so far
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(model.state_dict(), '/content/drive/MyDrive/DSP Assignments/best_model_rnn.pth')
        print(f"Best model saved with validation loss: {best_val_loss}")

# Plotting the training and validation losses
plt.plot(training_losses, label='Training Loss')
plt.plot(validation_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Losses')
plt.legend()
plt.show()
```

```
Best model saved with validation loss: 0.8737509850882463
Epoch 39, Training Loss: 0.6833679706650487
Epoch 39, Validation Loss: 0.8784746358859109
Epoch 40, Training Loss: 0.6760504310992779
Epoch 40, Validation Loss: 0.8807698978102757
Epoch 41, Training Loss: 0.6480152975447577
Epoch 41, Validation Loss: 0.8576294232898198
Best model saved with validation loss: 0.8576294232898198
Epoch 42, Training Loss: 0.6414257443480497
Epoch 42, Validation Loss: 0.862179037464598
Epoch 43, Training Loss: 0.637498249194471
Epoch 43, Validation Loss: 0.8574141783279006
Best model saved with validation loss: 0.8574141783279006
Epoch 44, Training Loss: 0.6289341994877314
Epoch 44, Validation Loss: 0.8514583127967477
Best model saved with validation loss: 0.8514583127967477
Epoch 45, Training Loss: 0.6236085948867226
Epoch 45, Validation Loss: 0.8482455432977158
Best model saved with validation loss: 0.8482455432977158
Epoch 46, Training Loss: 0.6197923019157545
Epoch 46, Validation Loss: 0.8468765172360649
Best model saved with validation loss: 0.8468765172360649
Epoch 47, Training Loss: 0.6139410238763595
Epoch 47, Validation Loss: 0.8510167058053855
Epoch 48, Training Loss: 0.6098523489852768
Epoch 48, Validation Loss: 0.8433155926439475
Best model saved with validation loss: 0.8433155926439475
Epoch 49, Training Loss: 0.6034437125102022
Epoch 49, Validation Loss: 0.8493894725652026
Epoch 50, Training Loss: 0.601034028683178
Epoch 50, Validation Loss: 0.843938441473051
```




▼ Training and Evaluation of the LSTM Model with Varying Epochs

```
import matplotlib.pyplot as plt

# Instantiate the LSTM model
lstm_model = LSTMModel(n_letters, 128, n_categories, num_layers=1)

# Loss function and optimizer
criterion = nn.NLLLoss()
optimizer = optim.Adam(lstm_model.parameters(), lr=0.001)

# Learning rate scheduler
```

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5)  # Adjust parameters as needed

# Variables to keep track of losses for each epoch
training_losses = []
validation_losses = []

# Variable to keep track of the lowest validation loss
best_val_loss = float('inf')  # Initialize with infinity

# Training loop for the LSTM model
num_epochs = 50
for epoch in range(num_epochs):
    total_loss = 0
    for names, categories in train_dataloader:
        optimizer.zero_grad()
        outputs = lstm_model(names.squeeze(2))
        loss = criterion(outputs, categories.squeeze(1))
        total_loss += loss.item()
        loss.backward()
        optimizer.step()

    train_loss = total_loss / len(train_dataloader)
    training_losses.append(train_loss)  # Store the training loss
    print(f'Epoch {epoch+1}, Training Loss: {train_loss}')

    # Validation loop to compute the validation loss
    lstm_model.eval()  # Set model to evaluation mode
    val_loss = 0
    with torch.no_grad():
        for names, categories in val_dataloader:
            outputs = lstm_model(names.squeeze(2))
            loss = criterion(outputs, categories.squeeze(1))
            val_loss += loss.item()

    val_loss /= len(val_dataloader)
    validation_losses.append(val_loss)  # Store the validation loss
    print(f'Epoch {epoch+1}, Validation Loss: {val_loss}')

    # Save the model if it has the lowest validation loss so far
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(lstm_model.state_dict(), '/content/drive/MyDrive/DSP Assignments/best_model_lstm.pth')
        print(f"Best model saved with validation loss: {best_val_loss}")

    # Adjust the learning rate
    scheduler.step()

# Plotting the training and validation losses
plt.plot(training_losses, label='Training Loss')
plt.plot(validation_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Losses')
plt.legend()
plt.show()
```

```
Epoch 1, Training Loss: 2.2183951483323026
Epoch 1, Validation Loss: 1.9760991288265128
Best model saved with validation loss: 1.9760991288265128
Epoch 2, Training Loss: 1.8372774132236636
Epoch 2, Validation Loss: 1.6840819392252613
Best model saved with validation loss: 1.6840819392252613
Epoch 3, Training Loss: 1.6439362811556768
Epoch 3, Validation Loss: 1.5915077077308755
Best model saved with validation loss: 1.5915077077308755
Epoch 4, Training Loss: 1.5114751814177674
Epoch 4, Validation Loss: 1.4493497711111296
Best model saved with validation loss: 1.4493497711111296
Epoch 5, Training Loss: 1.3717666375556143
Epoch 5, Validation Loss: 1.3072026482036523
Best model saved with validation loss: 1.3072026482036523
Epoch 6, Training Loss: 1.2006840629342994
Epoch 6, Validation Loss: 1.1123454383498914
Best model saved with validation loss: 1.1123454383498914
Epoch 7, Training Loss: 1.0408309726423808
Epoch 7, Validation Loss: 1.0245954177703138
Best model saved with validation loss: 1.0245954177703138
Epoch 8, Training Loss: 0.9206360468295365
Epoch 8, Validation Loss: 0.962180615427929
Best model saved with validation loss: 0.962180615427929
Epoch 9, Training Loss: 0.8232106701618406
Epoch 9, Validation Loss: 0.8811055303190474
Best model saved with validation loss: 0.8811055303190474
Epoch 10, Training Loss: 0.7399045409659994
Epoch 10, Validation Loss: 0.9205237177980913
Epoch 11, Training Loss: 0.6107270195911643
Epoch 11, Validation Loss: 0.707168563592105
Best model saved with validation loss: 0.707168563592105
Epoch 12, Training Loss: 0.561624975023766
Epoch 12, Validation Loss: 0.691823558143533
Best model saved with validation loss: 0.691823558143533
Epoch 13, Training Loss: 0.5276340661797992
Epoch 13, Validation Loss: 0.6230765082411938
Best model saved with validation loss: 0.6230765082411938
Epoch 14, Training Loss: 0.4917329326145587
Epoch 14, Validation Loss: 0.6433236923316276
Epoch 15, Training Loss: 0.475773429190445
Epoch 15, Validation Loss: 0.6081832317333367
Best model saved with validation loss: 0.6081832317333367
Epoch 16, Training Loss: 0.44359317911173557
Epoch 16, Validation Loss: 0.5827451900029471
Best model saved with validation loss: 0.5827451900029471
Epoch 17, Training Loss: 0.42795665239561265
Epoch 17, Validation Loss: 0.5994665150547417
Epoch 18, Training Loss: 0.41662567894699315
Epoch 18, Validation Loss: 0.5588049829543174
Best model saved with validation loss: 0.5588049829543174
Epoch 19, Training Loss: 0.3877759960277756
Epoch 19, Validation Loss: 0.5694736048814144
Epoch 20, Training Loss: 0.3710787384317792
Epoch 20, Validation Loss: 0.5874382777911451
Epoch 21, Training Loss: 0.3178151330122581
Epoch 21, Validation Loss: 0.5402467442762425
```

Load the saved best RNN model and evaluate its performance on the validation set.

```
# Evaluation loop on the validation set
# Load the saved best model
best_model_rnn = RNN(n_letters, n_hidden, n_categories, num_layers)
best_model_rnn.load_state_dict(torch.load('/content/drive/MyDrive/DSP Assignments/best_model_rnn.pth'))
best_model_rnn.eval()
correct = 0
total = 0
with torch.no_grad():
    for names, categories in val_dataloader:
        outputs = model(names.squeeze(2))
        _, predicted = torch.max(outputs, 1)
        total += categories.size(0)
        correct += (predicted == categories.squeeze(1)).sum().item()

        # Print predicted probabilities for each name
        probabilities = torch.exp(outputs)
        for i in range(len(names)):
            # Removing padded characters by stopping when a padding character is encountered
            original_name = ''.join([all_letters[torch.argmax(letter)] for letter in names[i] if torch.max(letter).item() > 0])
            print(f"Name: {original_name}, "
                  f"Predicted: {all_categories[predicted[i].item()]}, "
                  f"Probabilities: {probabilities[i].numpy()}")

print(f'Validation Accuracy: {100 * correct / total}%')
```

```
 1.5776795e-03 5.9869701e-01]
Name: Castellanos, Predicted: Spanish, Probabilities: [6.3138671e-08 4.2463245e-07 6.4410470e-02 2.3743904e-05 4.0072352e-03
 6.6607841e-04 1.9397145e-03 1.8370821e-03 8.3684362e-02 3.0645730e-03
 1.0936427e-05 8.4035534e-01]
Name: Bermudez, Predicted: Spanish, Probabilities: [3.3501868e-07 4.7477727e-07 3.3915836e-02 1.5145315e-05 3.9180452e-03
 2.6215483e-03 7.0850994e-04 2.4154703e-03 8.0152497e-02 2.8091839e-03
 6.6399371e-06 8.7343627e-01]
Name: Jimenez, Predicted: Spanish, Probabilities: [3.1133924e-05 4.2931228e-05 2.2421066e-02 2.8576028e-01 1.1820189e-01
 6.9108494e-02 1.0191160e-02 8.2121994e-03 5.2381560e-02 6.3638831e-04
 1.2807538e-04 4.3288487e-01]
Name: Silva, Predicted: Spanish, Probabilities: [1.2900697e-05 2.5907147e-04 2.4842307e-02 2.5909403e-01 7.8108594e-02
 8.5072249e-02 3.8162111e-03 7.1305255e-03 3.7273146e-02 1.2759358e-03
 2.8374334e-04 5.0283128e-01]
Name: Arena, Predicted: Spanish, Probabilities: [1.0629519e-03 2.7396234e-05 4.9890839e-02 2.1264370e-04 1.6425973e-02
 3.4589797e-02 9.0497853e-03 4.3809484e-04 3.9766783e-01 3.2335583e-02
 2.1669623e-03 4.5613205e-01]
Name: Ramirez, Predicted: Spanish, Probabilities: [1.9262023e-05 1.4894761e-05 3.3023290e-02 8.9814581e-02 3.5349555e-02
 1.6957169e-02 8.6847078e-03 5.3263055e-03 9.3821131e-02 6.2202296e-04
 6.2813910e-05 7.1630424e-01]
Name: Suarez, Predicted: English, Probabilities: [1.6461263e-05 6.4683612e-03 8.2991838e-02 3.5405648e-01 5.1661387e-02
 1.2052701e-01 1.8487381e-03 2.0211808e-02 2.1674242e-02 6.9081672e-03
 1.1195714e-03 3.3251593e-01]
Name: Bermudez, Predicted: Spanish, Probabilities: [3.3501868e-07 4.7477727e-07 3.3915836e-02 1.5145315e-05 3.9180452e-03
 2.6215483e-03 7.0850994e-04 2.4154703e-03 8.0152497e-02 2.8091839e-03
 6.6399371e-06 8.7343627e-01]
Name: Gil, Predicted: Italian, Probabilities: [5.6405588e-06 3.6618036e-05 6.6002421e-03 1.6946450e-04 1.4086805e-01
 1.7032137e-03 1.4024838e-03 3.8699270e-03 5.7052463e-01 9.9320104e-04
 2.3166253e-06 2.7382413e-01]
Name: Gaspar, Predicted: Spanish, Probabilities: [4.8103578e-07 4.3518929e-07 9.2281297e-02 3.2144868e-05 3.6054368e-03
 1.6631623e-03 2.4231372e-03 1.3366434e-03 5.8382574e-02 8.2670674e-03
 3.1218064e-05 8.3197641e-01]
Name: Ortega, Predicted: Spanish, Probabilities: [1.0552412e-05 1.1045527e-04 7.4069373e-02 2.7490154e-01 1.6525267e-03
 5.9880838e-02 2.4336989e-03 1.9539094e-02 6.0579877e-02 1.7661442e-01
 4.5146488e-05 3.3016253e-01]
Name: Orellana, Predicted: English, Probabilities: [2.3606468e-05 9.4945455e-05 1.5650235e-01 3.3415040e-01 3.6111956e-03
 7.0724756e-02 2.7418204e-03 6.4075016e-03 6.2893912e-02 9.6514292e-02
 1.4181709e-04 2.6619339e-01]
Name: Vives, Predicted: Russian, Probabilities: [2.1539037e-05 3.1243977e-05 9.9189408e-02 9.6449739e-04 8.3534658e-02
 1.8098475e-02 6.3421777e-03 3.3871186e-04 1.2832905e-01 6.0555787e-04
 3.8960025e-01 2.7294439e-01]
Name: Moreno, Predicted: Spanish, Probabilities: [9.1406382e-06 4.0439303e-05 1.3430227e-01 4.0424820e-02 7.6861028e-03
 3.1782386e-03 1.9563276e-02 5.0060758e-03 1.2704913e-01 8.3987843e-03
 2.2552178e-04 6.5411615e-01]
Validation Accuracy: 75.04939316963026%
```

## Load the saved best LSTM model and evaluate its performance on the validation set.

```
# Evaluation loop on the validation set
# Load the saved best model
best_model_lstm = LSTMModel(n_letters, n_hidden, n_categories, num_layers)
best_model_lstm.load_state_dict(torch.load('/content/drive/MyDrive/DSP Assignments/best_model_lstm.pth'))
best_model_lstm.eval()
correct = 0
total = 0
with torch.no_grad():
    for names, categories in val_dataloader:
        outputs = lstm_model(names.squeeze(2))
        _, predicted = torch.max(outputs, 1)
        total += categories.size(0)
        correct += (predicted == categories.squeeze(1)).sum().item()

        # Print predicted probabilities for each name
        probabilities = torch.exp(outputs)
        for i in range(len(names)):
            # Removing padded characters by stopping when a padding character is encountered
            original_name = ''.join([all_letters[torch.argmax(letter)] for letter in names[i] if torch.max(letter).item() > 0])
            print(f"Name: {original_name}, "
                  f"Predicted: {all_categories[predicted[i].item()]}, "
                  f"Probabilities: {probabilities[i].numpy()}")

print(f'Validation Accuracy: {100 * correct / total}%')
```

```
 1.8071429e-07 9.9101744e-01]
Name: Pena, Predicted: Spanish, Probabilities: [3.1434746e-07 1.1577340e-05 8.7195765e-03 8.4323036e-03 5.1972656e-05
 1.4205391e-03 6.7929723e-06 2.8109204e-04 2.7766163e-02 4.4696582e-03
 1.5455499e-06 9.4883847e-01]
Name: Vela, Predicted: Czech, Probabilities: [8.7292591e-09 8.1129554e-07 9.5137745e-01 9.1392743e-05 6.5271422e-05
 7.1311818e-04 1.2718217e-05 7.7017455e-04 3.3428200e-02 7.7433238e-04
 1.0063148e-06 1.2765487e-02]
Name: Castellanos, Predicted: Spanish, Probabilities: [1.1380449e-09 1.1639445e-09 1.3295936e-05 3.5489370e-06 3.2410874e-06
 4.2018193e-05 3.2150354e-05 3.0285444e-05 1.5606883e-03 2.9802933e-08
 8.9681646e-07 9.9831390e-01]
Name: Bermudez, Predicted: Spanish, Probabilities: [3.6449665e-06 3.5161541e-08 1.3989478e-05 4.9325026e-05 2.9115854e-06
 9.0427062e-04 4.3581687e-05 3.6994457e-05 1.0743596e-03 1.7742188e-06
 1.6737170e-06 9.9786735e-01]
Name: Jimenez, Predicted: Spanish, Probabilities: [3.1255419e-07 3.6673461e-07 2.2480523e-04 4.6231658e-03 1.2372350e-06
 6.5367547e-04 2.5389600e-03 3.9095034e-05 7.1730989e-04 1.9739343e-06
 2.2364420e-05 9.9117672e-01]
Name: Silva, Predicted: Spanish, Probabilities: [1.3509341e-06 4.7943968e-07 4.2602620e-03 3.6088340e-02 5.1741381e-06
 3.1935758e-04 1.0388025e-04 3.3048200e-04 6.4965211e-02 2.4153262e-05
 1.0517811e-05 8.9389080e-01]
Name: Arena, Predicted: Italian, Probabilities: [3.3302761e-06 2.2106502e-05 4.9217533e-02 4.0555923e-04 1.4755325e-04
 2.7383175e-03 1.5143950e-04 6.6031287e-03 8.8227499e-01 9.3032718e-03
 2.4863957e-06 4.9130257e-02]
Name: Ramirez, Predicted: Spanish, Probabilities: [6.6823205e-08 3.3844142e-08 8.4071098e-06 4.1456000e-04 1.1138561e-06
 2.1888861e-04 3.6734005e-04 5.2729843e-06 2.8558070e-04 2.6891232e-07
 1.4472745e-06 9.9869710e-01]
Name: Suarez, Predicted: Spanish, Probabilities: [1.3923653e-06 3.9344222e-08 2.0478032e-05 7.8170717e-04 3.2031503e-06
 6.5788126e-04 1.9238349e-04 2.5458141e-05 3.9372305e-04 1.3328798e-06
 1.4431300e-06 9.9792093e-01]
Name: Bermudez, Predicted: Spanish, Probabilities: [3.6449665e-06 3.5161541e-08 1.3989478e-05 4.9325026e-05 2.9115854e-06
 9.0427062e-04 4.3581687e-05 3.6994457e-05 1.0743596e-03 1.7742188e-06
 1.6737170e-06 9.9786735e-01]
Name: Gil, Predicted: Spanish, Probabilities: [1.6960710e-06 1.0269053e-02 9.1921454e-03 1.8264985e-02 5.9507798e-02
 3.0525040e-02 8.9459854e-06 3.0580931e-03 8.8593438e-03 5.3537856e-03
 1.9249177e-04 8.5476667e-01]
Name: Gaspar, Predicted: Spanish, Probabilities: [1.5841675e-04 7.5512885e-09 3.0906784e-05 1.3421429e-04 2.8283788e-05
 2.9840791e-03 6.4008444e-04 1.2010250e-05 2.0363356e-04 1.0089146e-07
 4.7297413e-05 9.9576098e-01]
Name: Ortega, Predicted: Spanish, Probabilities: [1.4709050e-07 4.2029915e-06 2.0219220e-03 4.2039022e-02 1.2401126e-04
 3.8903658e-04 1.2363621e-04 4.8563280e-03 8.8466793e-02 1.9116821e-02
 5.6373079e-07 8.4285754e-01]
Name: Orellana, Predicted: Italian, Probabilities: [2.12650477e-07 1.06855914e-07 4.17485790e-05 5.56892774e-05
 1.49021498e-05 1.60666093e-06 3.25402711e-04 1.90569146e-03
 9.96927798e-01 4.44884972e-05 2.50461483e-07 6.82185928e-04]
Name: Vives, Predicted: Spanish, Probabilities: [6.1133514e-06 7.7271415e-07 1.2362119e-03 6.0786129e-05 1.0483388e-02
 1.1222675e-03 2.5936756e-02 6.1381156e-06 1.3601736e-02 2.3670389e-06
 3.7342307e-01 5.7412040e-01]
Name: Moreno, Predicted: Spanish, Probabilities: [4.2088055e-07 8.6887823e-08 1.1621195e-04 5.4631161e-04 5.4337975e-06
 1.8510765e-04 4.7074014e-04 1.3695537e-03 4.6266060e-02 2.2508613e-05
 1.6071398e-07 9.5101744e-01]
Validation Accuracy: 87.52469658481513%
```

▼ Predicting the language category for a name using the best RNN model and providing probabilities for each category.

```
def predict_name_RNN(name, model):
    model.eval()  # Set the model to evaluation mode

    # Convert the name to tensor
    name_tensor = name_to_tensor(name, max_length).unsqueeze(0)  # Add batch dimension

    # Get the model output probabilities
    with torch.no_grad():
        output = model(name_tensor.squeeze(2))
        probabilities = torch.exp(output)  # Convert log probabilities to probabilities

    # Print the probabilities for each category
    for i, category in enumerate(all_categories):
        print(f"{category}: {probabilities[0][i].item()*100:.2f}%")

    # Print the category with the highest probability
    _, predicted = torch.max(output, 1)
    print(f"Predicted category: {all_categories[predicted.item()]}")

# Test with the name
predict_name_RNN('meloni', best_model_rnn)
```

```
Arabic: 0.04%
Chinese: 0.00%
Czech: 4.43%
English: 0.05%
```

```
French: 1.43%
German: 0.17%
Greek: 89.13%
Irish: 0.00%
Italian: 3.57%
Japanese: 0.77%
Russian: 0.01%
Spanish: 0.41%
Predicted category: Greek
```

▼ Predicting the language category for a name using the best LSTM model and providing probabilities for each category.

```
# Test with the name using the LSTM model
def predict_name_with_lstm(name, model):
    model.eval()
    name_tensor = name_to_tensor(name, max_length).unsqueeze(0)
    with torch.no_grad():
        output = model(name_tensor.squeeze(2))
        probabilities = torch.exp(output)

    for i, category in enumerate(all_categories):
        print(f"{category}: {probabilities[0][i].item() * 100:.2f}%")

    _, predicted = torch.max(output, 1)
    print(f"Predicted category: {all_categories[predicted.item()]}")

  # Test with the name
predict_name_with_lstm('meloni', best_model_lstm)
```

```
Arabic: 0.00%
Chinese: 0.00%
Czech: 0.00%
English: 0.00%
French: 0.01%
German: 0.00%
Greek: 0.14%
Irish: 0.03%
Italian: 99.72%
Japanese: 0.03%
Russian: 0.00%
Spanish: 0.06%
Predicted category: Italian
```

▼ Assessing correct and incorrect predictions for each language category in the validation set for RNN

```
import matplotlib.pyplot as plt
import numpy as np

# Evaluation loop on the validation set for the RNN model
best_model_rnn.eval()
correct_counts = {category: 0 for category in all_categories}
incorrect_counts = {category: 0 for category in all_categories}

with torch.no_grad():
    for names, categories in val_dataloader:
        outputs = best_model_rnn(names.squeeze(2))
        _, predicted = torch.max(outputs, 1)

        for i in range(len(names)):
            true_category = all_categories[categories[i].item()]
            predicted_category = all_categories[predicted[i].item()]

            if true_category == predicted_category:
                correct_counts[true_category] += 1
            else:
                incorrect_counts[true_category] += 1

categories = list(all_categories)
correct_values = [correct_counts[category] for category in categories]
incorrect_values = [incorrect_counts[category] for category in categories]

x = np.arange(len(categories))
width = 0.35

fig, ax = plt.subplots()
```