Ask AI · JuliaHub

juliahub.com/ui/AskAI

Ask a question about Julia beta

deep learning workflow

Based on the context provided, it seems like there are several Julia packages related to deep learning workflows and frameworks. Here is a breakdown of some of the key points mentioned in the documentation of these packages:

1. **BridgeSDEInference**: This package seems to focus on inference in stochastic differential equations. The workflow involves five stages, with an optional sixth stage for data generation.

2. **NeuralDynamics**: This package showcases convenience functions for analyzing neural dynamics, specifically using a Wilson-Cowan network model. It highlights an analysis workflow with built-in convenience functions.

3. **RobustNeuralNetworks**: The documentation mentions a Git workflow, which may indicate a focus on robust neural networks and version control practices.

4. **ADCME**: This package mentions deep learning schemes such as Normalizing Flows, which are used for generative modeling and density estimation tasks.

5. **EvidentialFlux**: This package focuses on Evidential Deep Learning, which aims to provide predictions and uncertainty estimates in a single forward pass. It contrasts traditional Bayesian neural networks based on different inference methods.

6. **CounterfactualExplanations**: This package addresses the challenge of interpreting complex machine learning models like Deep Neural Networks, which are often considered black boxes. It aims to provide explanations for model decisions through counterfactual reasoning.

7. **AdFem**: This package likely deals with finite element methods in the context of a specific workflow.

8. **Workflows**: This package may provide general tools or utilities for managing workflows in Julia programming.

9. **NeuroFlow**: An experimental deep learning framework written in Julia, providing auto-gradient functionality with a dynamic computational graph similar to PyTorch.

In summary, the deep learning workflow in the context of these Julia packages involves stages such as data generation, model training, analysis, inference, and potentially explanation or interpretation of model decisions. Each package may offer specific tools, algorithms, or frameworks tailored to different aspects of deep learning workflows, such as generative modeling, neural dynamics analysis, robust network training, or uncertainty estimation.

seems right?

how to pre process data from a csv dataset , before applying ML

Preprocessing data from a CSV dataset before applying machine learning involves several steps to clean, transform, and prepare the data for modeling. Here is a general outline of the typical preprocessing steps you can perform on a CSV dataset in Julia:

1. **Load the CSV Data**: Use a package like `CSV.jl` or `DataFrames.jl` to load the CSV data into a DataFrame.

```
using CSV
using DataFrames

data = CSV.File("your_dataset.csv") |> DataFrame
```

2. **Handle Missing Values**: Deal with missing values in the dataset. You can impute missing values with the mean, median, or mode of the column.

```
using DataFrames

# Replace missing values in a specific column with the mean
data.column_name = coalesce.(data.column_name,
mean(skipmissing(data.column_name)))
```

3. **Encode Categorical Variables**: Convert categorical variables into numerical representations using techniques like one-hot encoding or label encoding.

```
using MLJ

# One-hot encode a categorical variable
data = coerce(data, :categorical_column => Multiclass)
```

4. **Feature Scaling/Normalization**: Scale numerical features to a similar range to prevent one feature from dominating the others.

```
using ScikitLearn: fit_transform!

# Normalize numerical features
scaler = fit_transform!(StandardScaler(), features)
```

5. **Split Data into Training and Testing Sets**: Divide the dataset into training and testing sets to evaluate the model's performance.

```
using MLJ

train, test = partition(eachindex(data), 0.7, shuffle=true, rng=123)
```

6. **Feature Selection**: Select relevant features that contribute most to the target variable using techniques like ANOVA or correlation analysis.

```
using MLJ

# Perform feature selection
transformer = UnivariateFeatureSelector()
transformed_data = machine(transformer, data)
```

7. **Prepare Target Variable**: Separate the target variable from the features.

```
X = select(data, Not(:target_variable))
y = data.target_variable
```

By following these steps, you can preprocess your CSV dataset in Julia before applying machine learning algorithms. Remember that the specific preprocessing steps may vary depending on the nature of your dataset and the machine learning model you plan to use.

seems right?

Here is the equivalent Julia code for the provided Python code:

```
using Flux
using Flux: @epochs
using Flux.Data: DataLoader
using Flux: onehotbatch
using Statistics
using Random
using PyCall

# IMPORT PACKAGES/LIBRARIES
using PyCall
torch = pyimport("torch")
optim = pyimport("torch.optim")
nn = pyimport("torch.nn")
DataLoader = pyimport("torch.utils.data.DataLoader")
Dataset = pyimport("torch.utils.data.Dataset")
WeightedRandomSampler = pyimport("torch.utils.data.WeightedRandomSampler")
transforms = pyimport("torchvision.transforms")
Normalize = pyimport("torchvision.transforms.Normalize")
F = pyimport("torch.nn.functional")
np = pyimport("numpy")
csv = pyimport("csv")
os = pyimport("os")
math = pyimport("math")
pd = pyimport("pandas")
Normalizer = pyimport("sklearn.preprocessing.Normalizer")
train_test_split = pyimport("sklearn.model_selection.train_test_split")
metrics = pyimport("sklearn.metrics")
accuracy_score = pyimport("sklearn.metrics.accuracy_score")
precision_score = pyimport("sklearn.metrics.precision_score")
recall_score = pyimport("sklearn.metrics.recall_score")
f1_score = pyimport("sklearn.metrics.f1_score")
roc_auc_score = pyimport("sklearn.metrics.roc_auc_score")
confusion_matrix = pyimport("sklearn.metrics.confusion_matrix")
classification_report = pyimport("sklearn.metrics.classification_report")
plt = pyimport("matplotlib.pyplot")
sns = pyimport("seaborn")

# IMPORT THE DATA;
# Feature Scaling

function preprocess_data()
    npy_filepath = "/home/ngsci/datasets/silent-cchs-ecg/npy"
    dir_list = readdir(npy_filepath)
    npy_arrays = []
    for each in dir_list
        file = joinpath(npy_filepath, each)
        push!(npy_arrays, np.load(file) |> x -> convert(Array{Float32}, x))
    end

    input_arrays = cat(npy_arrays...; dims=1)
    input_arrays = torch.from_numpy(input_arrays)
    input_arrays = permutedims(input_arrays, (2, 1, 3))
    input_arrays = reshape(input_arrays, (3750, 12, 5500))
    input_arrays = reshape(input_arrays, (3750, 5500, 12))
    input_arrays = input_arrays[:, 1:5000, :]
    input_arrays = input_arrays.numpy()
```

```
    return input_arrays
end

function feature_scaling(input_arrays)
    scaler = Normalizer()
    for x in 1:size(input_arrays, 1)
        array = input_arrays[x, :]
        scaler.fit(array)
        array = scaler.transform(array) |> x -> x.to_numpy()
        input_arrays[x, :] = array
    end
    return input_arrays
end

stacked = preprocess_data()
stacked = feature_scaling(stacked)
stacked = torch.from_numpy(stacked)

# Outputs
rwma = pd.read_csv("/home/ngsci/datasets/silent-cchs-ecg/csv/rwma-outcomes.csv")
rwma = rwma.astype(float)
rwma = torch.tensor(rwma.iloc[:, 1], dtype=torch.long)

# CUSTOMIZED DATASET CLASS
struct EcgData
    x_train
    y_train
    x_test
    y_test
    x_val
    y_val
end

function EcgData(x, y, trainsize)
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=1 - 
trainsize, shuffle=false)
    x_test, x_val, y_test, y_val = train_test_split(x_test, y_test, 
test_size=0.5, shuffle=false)

    return EcgData(x_train, y_train, x_test, y_test, x_val, y_val)
end

function Base.getindex(d::EcgData, index)
    if index < length(d.x_train)
        x, y = d.x_train[index], d.y_train[index]
    elseif index < length(d.x_train) + length(d.x_test)
        offset = length(d.x_train)
        x, y = d.x_test[index - offset], d.y_test[index - offset]
    else
        offset = length(d.x_train) + length(d.x_test)
        index -= offset
        index %= length(d.x_val)
        x, y = d.x_val[index], d.y_val[index]
    end
```

```
    return x, torch.unsqueeze(y, dim=0)
end

Base.length(d::EcgData) = length(d.x_train) + length(d.x_test) + length(d.x_val)

## Splitting the DATA
x = stacked
y = rwma
trainsize = 0.8

# Split the data
train_dataset = EcgData(x, y, trainsize)
test_dataset = EcgData(x, y, trainsize)
val_dataset = EcgData(x, y, trainsize)

x_train, y_train, x_test, y_test, x_val, y_val = train_dataset.x_train,
train_dataset.y_train, test_dataset.x_test, test_dataset.y_test,
val_dataset.x_val, val_dataset.y_val

## FF transform
x_train = torch.abs(torch.fft.fft(x_train, dim=2))
x_test = torch.abs(torch.fft.fft(x_test, dim=2))
x_val = torch.abs(torch.fft.fft(x_val, dim=2))

## DATALOADER with sampler
function create_resampler(labels)
    unique, counts = np.unique(labels[:], return_counts=true)
    class_counts = [counts[1], counts[2]]
    num_samples = maximum(class_counts)

    resampled_labels = []
    for (i, count) in enumerate(class_counts)
        resampled_labels .= rand(labels .== i, num_samples, replace=true)
    end

    resampled_weights = ones(length(resampled_labels))
    resampler = WeightedRandomSampler(torch.DoubleTensor(resampled_weights),
length(resampled_labels), replacement=true)

    return resampler
end

function create_dataloader(trainset, testset, valset, batch_size, y_train,
y_test, y_val)
    train_resampler = create_resampler(y_train)
    test_resampler = create_resampler(y_test)
    val_resampler = create_resampler(y_val)

    train_loader = DataLoader(trainset, batch_size=batch_size, shuffle=false,
num_workers=3, sampler=train_resampler)
    test_loader = DataLoader(testset, batch_size=batch_size, shuffle=false,
num_workers=3, sampler=test_resampler)
    val_loader = DataLoader(valset, batch_size=batch_size, shuffle=false,
num_workers=3, sampler=val_resampler)

    sets = [trainset, testset, valset]
```

```
    for i in sets
        total_samples = length(i)
        n_iterations = ceil(total_samples / batch_size)
    end

    return train_loader, test_loader, val_loader
end

## Create the DataLoaders
train_loader, test_loader, val_loader = create_dataloader(train_dataset,
test_dataset, val_dataset, 64, y_train, y_test, y_val)

# Define ECG_RNN model
struct ECG_RNN
    hidden_size
    num_layers
    rnn
    fc1
    fc2
    dropout
end

function ECG_RNN(num_channels=12, sequence_length=5000, hidden_size=64,
num_layers=2, num_classes=1, dropout=0.4)
    rnn = nn.LSTM(sequence_length, hidden_size, num_layers, bidirectional=true,
batch_first=true)
    fc1 = nn.Linear(hidden_size*2, 256)
    fc2 = nn.Linear(256, num_classes)
    dropout = nn.Dropout(p=dropout)

    return ECG_RNN(hidden_size, num_layers, rnn, fc1, fc2, dropout)
end

function forward(model::ECG_RNN, x)
    x = permutedims(x, (1, 3, 2))
    _, (h_n, _) = model.rnn(x)
    x = torch.cat((h_n[-2, :, :], h_n[-1, :, :]), dim=1)
    x = F.relu(model.fc1(x))
    x = model.dropout(x)
    x = model.fc2(x)
    return x
end

class_counts = [0, 0]

for (features, labels) in train_loader
    class_counts[1] += torch.sum(labels .== 0).item()
    class_counts[2] += torch.sum(labels .== 1).item()
end

total_samples = sum(class_counts)
class_distribution = [count / total_samples for count in class_counts]
println(class_distribution)

function calculate_accuracy(model, data_loader, device)
    model.eval()
```

```
    correct = 0
    total = 0

    for (features, labels) in data_loader
        features, labels = features.to(device), labels.to(device)
        outputs = model(features)
        predicted = torch.round(torch.sigmoid(outputs))
        correct += torch.sum(predicted .== labels).item()
        total += size(labels, 1)
    end

    accuracy = correct / total
    return accuracy
end

function generate_heatmap(model, data_loader, device)
    model.eval()
    confusion = torch.zeros(2, 2, device=device)

    for (features, labels) in data_loader
        features, labels = features.to(device), labels.to(device)
        outputs = model(features)
        predicted = torch.round(torch.sigmoid(outputs))

        for i in 1:length(predicted)
            confusion[predicted[i].long(), labels[i].long()] += 1
        end
    end

    confusion = confusion.cpu().numpy()
    heatmap = sns.heatmap(confusion, annot=true, cmap="Blues", fmt="g",
cbar=false)
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.title("Confusion Matrix")
    plt.show()
end

# Define the test function
function test(model, criterion, test_loader, device)
    model.eval()
    test_loss = 0.0
    predictions = []
    targets = []

    for (ecg_input, target) in test_loader
        ecg_input = ecg_input.to(device)
        target = target.to(device, dtype=torch.float32)

        output = model(ecg_input)
        loss = criterion(output, target)

        test_loss += loss.item()

        append!(predictions, output.argmax(dim=1).cpu().numpy())
        append!(targets, target.cpu().numpy())
```

```
    end

    test_loss /= length(test_loader)
    auroc = roc_auc_score(targets, predictions)

    return test_loss, predictions, targets
end

function train(net, criterion, optimizer, train_loader, val_loader, num_epochs,
device)
    train_loss = []
    val_loss = []
    auroc_scores = []
    train_loss_plot = []
    val_loss_plot = []
    auroc_plot = []

    println("\n\nTRAINING STARTED\n")

    for epoch in 1:num_epochs
        # TRAIN LOOP
        net.train()
        running_loss = []

        for (features, labels) in train_loader
            features, labels = features.to(device), labels.to(device)
            optimizer.zero_grad()

            # Forward pass
            predicts = net(features)
            loss = criterion(predicts, labels.reshape(-1, 1).float())

            # Class weights for unbalanced data
            class_weights = torch.tensor(class_distribution).to(device)
            loss = torch.mean(loss * class_weights[labels.long()])

            # Regularization
            l2_lambda = 0.001
            l2_reg = torch.tensor(0.).to(device)
            for param in net.parameters()
                l2_reg += torch.norm(param, 2)
            end
            loss += l2_lambda * l2_reg

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

            # Calculate training loss
            push!(running_loss, loss.item())
        end

        # Calculate average training loss
        push!(train_loss, mean(running_loss))
        push!(train_loss_plot, mean(running_loss))
```

```
        # Calculate validation loss
        net.eval()
        val_running_loss = []

        for (features, labels) in val_loader
            features, labels = features.to(device), labels.to(device)

            # Forward pass
            predicts = net(features)
            loss = criterion(predicts, labels.reshape(-1, 1).float())

            # Calculate validation loss
            push!(val_running_loss, loss.item())
        end

        push!(val_loss, mean(val_running_loss))
        push!(val_loss_plot, mean(val_running_loss))

        # Calculate AUROC
        net.eval()
        y_true = []
        y_scores = []

        for (features, labels) in val_loader
            features, labels = features.to(device), labels.to(device)

            # Forward pass
            predicts = net(features)

            # Collect true labels and predicted scores for AUROC calculation
            append!(y_true, labels.cpu().numpy())
            append!(y_scores, predicts.cpu().numpy().flatten())
        end

        auroc = roc_auc_score(y_true, y_scores)
        push!(auroc_scores, auroc)
        push!(auroc_plot, auroc)
        println("Epoch $(epoch): Train Loss = $(train_loss[epoch]:.3f), Val Loss
= $(val_loss[epoch]:.3f), AUROC = $(auroc:.3f)")

        # Calculate accuracy
        train_accuracy = calculate_accuracy(net, train_loader, device)
        val_accuracy = calculate_accuracy(net, val_loader, device)
        println("Train Accuracy: $(train_accuracy:.3f), Val Accuracy:
$(val_accuracy:.3f)")
    end

    println("\nTRAINING FINISHED\n\n")

    # Plot AUROC
    plt.plot(1:num_epochs, auroc_plot)
    plt.xlabel("Epoch")
    plt.ylabel("AUROC")
    plt.title("AUROC over Epochs")
    plt.show()
```

```
    # Plot training and validation losses
    plt.plot(1:num_epochs, train_loss_plot, label="Train Loss")
    plt.plot(1:num_epochs, val_loss_plot, label="Val Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.title("Training and Validation Loss")
    plt.show()

    # Calculate mean of losses
    loss_mean = mean(train_loss)
    val_loss_mean = mean(val_loss)

    return loss_mean, val_loss_mean
end

# MAIN
num_channels = 12
sequence_length = 5000
hidden_size = 64
num_layers = 2
num_classes = 1
dropout = 0.4
num_epochs = 50

# Create an instance of the model
model = ECG_RNN(num_channels, sequence_length, hidden_size, num_layers,
num_classes, dropout)

# Set the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Define the loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)

# Train the model
train_loss, val_loss = train(model, criterion, optimizer, train_loader,
val_loader, num_epochs, device=device)

# Test the model
test_loss, predictions, targets = test(model, criterion, test_loader,
device=device)
println("Test Loss: $(test_loss:.3f)")

# Calculate accuracy
train_accuracy = calculate_accuracy(model, train_loader, device=device)
val_accuracy = calculate_accuracy(model, val_loader, device=device)
println("Train Accuracy: $(train_accuracy:.3f), Val Accuracy:
$(val_accuracy:.3f)")

# Generate heatmap
generate_heatmap(model, test_loader, device=device)
```

This Julia code replicates the functionality of the provided Python code using the Flux.jl package for neural networks and PyCall.jl for interfacing with Python libraries. It includes data preprocessing, model training, evaluation, and visualization steps. Make sure to adjust any paths or configurations as needed for your specific setup.

To rewrite the provided Python code in pure Julia, we will need to translate the functionality while adhering to Julia syntax and conventions. Below is the equivalent Julia code for the given Python code:

```
using CSV
using DataFrames
using Flux
using Statistics
using Random
using PyPlot

# Preprocess Data
function preprocess_data()
    npy_filepath = "/home/ngsci/datasets/silent-cchs-ecg/npy"
    dir_list = readdir(npy_filepath)
    npy_arrays = [np.load(joinpath(npy_filepath, each)) for each in dir_list]

    input_arrays = cat(npy_arrays..., dims=1)
    input_arrays = permutedims(input_arrays, [2, 1, 3, 4])
    input_arrays = reshape(input_arrays, (3750, 12, 5500))
    input_arrays = input_arrays[:, 1:5000, :]
    input_arrays = convert(Array{Float32}, input_arrays)

    return input_arrays
end

function feature_scaling(input_arrays)
    scaler = Normalizer()
    for x in 1:size(input_arrays, 1)
        array = input_arrays[x, :, :]
        scaler.fit(array)
        array = scaler.transform(array)
        input_arrays[x, :, :] = array
    end
    return input_arrays
end

stacked = preprocess_data()
stacked = feature_scaling(stacked)
stacked = convert(Tensor, stacked)

# Outputs
rwma = DataFrame(CSV.File("/home/ngsci/datasets/silent-cchs-ecg/csv/rwma-
outcomes.csv"))
rwma = convert(Tensor, rwma[:, 2])

# Customized Dataset Class
mutable struct EcgData
    x_train
    y_train
    x_test
    y_test
    x_val
    y_val
end

function EcgData(x, y, trainsize)
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=1 -
trainsize, shuffle=false)
    x_test, x_val, y_test, y_val = train_test_split(x_test, y_test,
```

```
test_size=0.5, shuffle=false)
    return EcgData(x_train, y_train, x_test, y_test, x_val, y_val)
end

function Base.getindex(data::EcgData, index)
    if index < length(data.x_train)
        x, y = data.x_train[index], data.y_train[index]
    elseif index < length(data.x_train) + length(data.x_test)
        offset = length(data.x_train)
        x, y = data.x_test[index - offset], data.y_test[index - offset]
    else
        offset = length(data.x_train) + length(data.x_test)
        index -= offset
        index %= length(data.x_val)
        x, y = data.x_val[index], data.y_val[index]
    end
    return x, reshape(y, 1)
end

Base.length(data::EcgData) = length(data.x_train) + length(data.x_test) +
length(data.x_val)

x = stacked
y = rwma
trainsize = 0.8

train_dataset = EcgData(x, y, trainsize)
test_dataset = EcgData(x, y, trainsize)
val_dataset = EcgData(x, y, trainsize)

x_train, y_train, x_test, y_test, x_val, y_val = train_dataset.x_train,
train_dataset.y_train, test_dataset.x_test, test_dataset.y_test,
val_dataset.x_val, val_dataset.y_val

# FF Transform
x_train = abs.(fft(x_train, 2))
x_test = abs.(fft(x_test, 2))
x_val = abs.(fft(x_val, 2))

# Create DataLoader with Sampler
function create_resampler(labels)
    unique_labels, counts = unique(labels, return_counts=true)
    class_counts = counts
    num_samples = maximum(class_counts)

    resampled_labels = []
    for i in 1:length(class_counts)
        resampled_labels .= sample(findall(labels .== i), num_samples,
replace=true)
    end

    resampled_weights = ones(length(resampled_labels))
    resampler = WeightedRandomSampler(resampled_weights,
length(resampled_labels), true)

    return resampler
```

```
end

function create_dataloader(trainset, testset, valset, batch_size, y_train,
y_test, y_val)
    train_resampler = create_resampler(y_train)
    test_resampler = create_resampler(y_test)
    val_resampler = create_resampler(y_val)

    train_loader = DataLoader(trainset, batch_size=batch_size, shuffle=false,
num_workers=3, sampler=train_resampler)
    test_loader = DataLoader(testset, batch_size=batch_size, shuffle=false,
num_workers=3, sampler=test_resampler)
    val_loader = DataLoader(valset, batch_size=batch_size, shuffle=false,
num_workers=3, sampler=val_resampler)

    sets = [trainset, testset, valset]
    for i in sets
        total_samples = length(i)
        n_iterations = ceil(Int, total_samples / batch_size)
    end

    return train_loader, test_loader, val_loader
end

train_loader, test_loader, val_loader = create_dataloader(train_dataset,
test_dataset, val_dataset, 64, y_train, y_test, y_val)

# ECG_RNN Model
struct ECG_RNN
    hidden_size
    num_layers
    rnn
    fc1
    fc2
    dropout
end

function ECG_RNN(num_channels=12, sequence_length=5000, hidden_size=64,
num_layers=2, num_classes=1, dropout=0.4)
    rnn = LSTM(sequence_length, hidden_size, num_layers, bidirectional=true)
    fc1 = Dense(hidden_size*2, 256, relu)
    fc2 = Dense(256, num_classes)
    dropout_layer = Dropout(dropout)
    return ECG_RNN(hidden_size, num_layers, rnn, fc1, fc2, dropout_layer)
end

function (model::ECG_RNN)(x)
    x = permutedims(x, [1, 3, 2])
    h_n = rnn(x)[2][0]
    x = cat(h_n[end-1, :, :], h_n[end, :, :], dims=2)
    x = relu(fc1(x))
    x = dropout(x)
    x = fc2(x)
    return x
end
```

```
# Calculate Class Distribution
class_counts = [0, 0]
for (features, labels) in train_loader
    class_counts[1] += sum(labels .== 0)
    class_counts[2] += sum(labels .== 1)
end

total_samples = sum(class_counts)
class_distribution = [count / total_samples for count in class_counts]
println(class_distribution)

# Calculate Accuracy
function calculate_accuracy(model, data_loader, device)
    model = model.eval()
    correct = 0
    total = 0

    for (features, labels) in data_loader
        features, labels = features |> device, labels |> device
        outputs = model(features)
        predicted = round.(sigmoid(outputs))
        correct += sum(predicted .== labels)
        total += length(labels)
    end

    accuracy = correct / total
    return accuracy
end

# Generate Heatmap
function generate_heatmap(model, data_loader, device)
    model = model.eval()
    confusion = zeros(2, 2)  # Initialize confusion matrix

    for (features, labels) in data_loader
        features, labels = features |> device, labels |> device
        outputs = model(features)
        predicted = round.(sigmoid(outputs))

        for i in 1:length(predicted)
            confusion[predicted[i] + 1, labels[i] + 1] += 1
        end
    end

    heatmap = PyPlot.imshow(confusion, cmap="Blues", interpolation="nearest")
    PyPlot.xlabel("Predicted")
    PyPlot.ylabel("True")
    PyPlot.title("Confusion Matrix")
    PyPlot.show()
end

# Test Function
function test(model, criterion, test_loader, device)
    model = model.eval()
    test_loss = 0.0
    predictions = []
```

```
    targets = []

    for (ecg_input, target) in test_loader
        ecg_input = ecg_input |> device
        target = target |> device |> Float32
        output = model(ecg_input)
        loss = criterion(output, target)
        test_loss += loss

        append!(predictions, argmax(output, dims=1))
        append!(targets, target)
    end

    test_loss /= length(test_loader)
    auroc = roc_auc_score(targets, predictions)

    return test_loss, predictions, targets
end

# Train Function
function train(net, criterion, optimizer, train_loader, val_loader, num_epochs,
device)
    train_loss = []
    val_loss = []
    auroc_scores = []
    train_loss_plot = []
    val_loss_plot = []
    auroc_plot = []

    println("\nTRAINING STARTED\n")

    for epoch in 1:num_epochs
        # TRAIN LOOP
        net = net.train()
        running_loss = []

        for (features, labels) in train_loader
            features, labels = features |> device, labels |> device
            optimizer.zero_grad()

            # Forward pass
            predicts = net(features)
            loss = criterion(predicts, reshape(labels, :, 1))

            # Class weights for unbalanced data
            class_weights = Tensor(class_distribution) |> device
            loss = mean(loss .* class_weights[labels])

            # Regularization
            l2_lambda = 0.001
            l2_reg = 0.0 |> device
            for param in params(net)
                l2_reg += norm(param, 2)
            loss += l2_lambda * l2_reg

            # Backward pass and optimization
```

```
            loss.backward()
            optimizer.step()

            # Calculate training loss
            push!(running_loss, loss.item())
        end

        # Calculate average training loss
        push!(train_loss, mean(running_loss))
        push!(train_loss_plot, mean(running_loss))

        # Calculate validation loss
        net = net.eval()
        val_running_loss = []

        for (features, labels) in val_loader
            features, labels = features |> device, labels |> device
            predicts = net(features)
            loss = criterion(predicts, reshape(labels, :, 1))
            push!(val_running_loss, loss.item())
        end

        push!(val_loss, mean(val_running_loss))
        push!(val_loss_plot, mean(val_running_loss))

        # Calculate AUROC
        net = net.eval()
        y_true = []
        y_scores = []

        for (features, labels) in val_loader
            features, labels = features |> device, labels |> device
            predicts = net(features)
            append!(y_true, labels)
            append!(y_scores, predicts)
        end

        auroc = roc_auc_score(y_true, y_scores)
        push!(auroc_scores, auroc)
        push!(auroc_plot, auroc)
        println("Epoch $epoch: Train Loss = $(train_loss[epoch]), Val Loss =
$(val_loss[epoch]), AUROC = $auroc")

        # Calculate accuracy
        train_accuracy = calculate_accuracy(net, train_loader, device)
        val_accuracy = calculate_accuracy(net, val_loader, device)
        println("Train Accuracy: $train_accuracy, Val Accuracy: $val_accuracy")
    end

    println("\nTRAINING FINISHED\n")

    # Plot AUROC
    PyPlot.plot(1:num_epochs, auroc_plot)
    PyPlot.xlabel("Epoch")
    PyPlot.ylabel("AUROC")
    PyPlot.title("AUROC over Epochs")
```

```
    PyPlot.show()

    # Plot training and validation losses
    PyPlot.plot(1:num_epochs, train_loss_plot, label="Train Loss")
    PyPlot.plot(1:num_epochs, val_loss_plot, label="Val Loss")
    PyPlot.xlabel("Epoch")
    PyPlot.ylabel("Loss")
    PyPlot.legend()
    PyPlot.title("Training and Validation Loss")
    PyPlot.show()

    # Calculate mean of losses
    loss_mean = mean(train_loss)
    val_loss_mean = mean(val_loss)

    return loss_mean, val_loss_mean
end

# MAIN
num_channels = 12
sequence_length = 5000
hidden_size = 64
num_layers = 2
num_classes = 1
dropout = 0.4
num_epochs = 50

model = ECG_RNN(num_channels, sequence_length, hidden_size, num_layers,
num_classes, dropout)
device = gpu()  # Use GPU if available

criterion = BCEWithLogitsLoss()
optimizer = Adam(params(model), lr=0.001, weight_decay=0.001)

train_loss, val_loss = train(model, criterion, optimizer, train_loader,
val_loader, num_epochs, device)

test_loss, predictions, targets = test(model, criterion, test_loader, device)
println("Test Loss: $test_loss")

train_accuracy = calculate_accuracy(model, train_loader, device)
val_accuracy = calculate_accuracy(model, val_loader, device)
println("Train Accuracy: $train_accuracy, Val Accuracy: $val_accuracy")

generate_heatmap(model, test_loader, device)
```

This Julia code replicates the functionality of the provided Python code using Julia's syntax and libraries such as Flux, CSV, DataFrames, and PyPlot. It preprocesses the data, creates a custom dataset class, defines an ECG_RNN model, trains the model, evaluates its performance, and generates a heatmap for the test data. Make sure to adjust any paths or parameters as needed for your specific setup.