(1)Distributed computing Pseudo Codes:

## Pseudo Code for Distributed Model Training

### 1. Data Preparation

\* \*\*Load Geographic Data\*\*: Read geographic data from a URL into a GeoDataFrame.

\* \*\*Project Data\*\*: Convert the GeoDataFrame to Web Mercator projection (EPSG:3857).

\* \*\*Generate Labels\*\*: Create binary labels based on a population threshold (e.g., population > 10 million).

\* \*\*Extract Features\*\*:

\* Calculate the centroid coordinates (x, y) for each geographic entity.

\* Store x, y coordinates as features.

\* \*\*Split Data\*\*: Divide the data into training and testing sets (e.g., 80% training, 20% testing).

### 2. Teacher Model Training

\* \*\*Define Teacher Model\*\*:

\* Create a sequential neural network model.

\* Input Layer: Expects input of shape (2,) (x, y coordinates).

\* Hidden Layers:

\* Dense layer with 128 neurons and ReLU activation.

\* Batch Normalization layer.

\* Dense layer with 32 neurons and ReLU activation.

\* Output Layer: Dense layer with 1 neuron and sigmoid activation (for binary classification).

\* \*\*Compile Teacher Model\*\*:

\* Use the Adam optimizer.

\* Use binary cross-entropy loss function.

\* Track accuracy as a metric.

\* \*\*Train Teacher Model\*\*:

\* Train the model on the training data.

\* Use early stopping to prevent overfitting (monitor validation loss and restore best weights).

\* Set verbose to 0 to suppress training output.

\* \*\*Generate Soft Labels\*\*:

\* Use the trained teacher model to predict on the training data.

\* Apply temperature scaling to the logits (output of the teacher model) to produce soft labels.

\* Temperature Scaling: Divide the logits by a temperature value (e.g., 5.0) and apply the sigmoid function.

### 3. Student Model Training (Distributed with Dask)

\* \*\*Initialize Dask Client\*\*: Start a Dask client for distributed computation.

\* \*\*Define Objective Function\*\*:

\* \*\*Input\*\*: A set of hyperparameters (learning rate, batch size, alpha).

\* \*\*Process\*\*:

\* \*\*Define Student Model\*\*:

\* Create a sequential neural network model (smaller than the teacher model).

\* Input Layer: Expects input of shape (2,).

\* Hidden Layers:

\* Dense layer with 64 neurons and ReLU activation.

\* Batch Normalization layer.

\* Dropout layer (e.g., with a rate of 0.3).

\* Dense layer with 32 neurons, ReLU activation, and L2 regularization.

\* Output Layer: Dense layer with 1 neuron.

\* \*\*Compile Student Model:\*\* Use Adam optimizer with given learning rate.

\* \*\*Knowledge Distillation Training Loop\*\*:

\* For each epoch:

\* For each batch:

\* Randomly select a batch of data from the training set.

\* Calculate the loss:

\* `loss = alpha \* loss\_true + (1 - alpha) \* loss\_soft + beta \* regularization\_loss`

\* `loss\_true`: Binary cross-entropy loss between student predictions and true labels.

\* `loss\_soft`: Binary cross-entropy loss between student predictions and soft labels (from the teacher).

\* `regularization\_loss`: L2 regularization loss.

\* `alpha`: Weighting factor for true vs soft label loss

\* `beta`: Weighting factor for regularization loss.

\* Calculate gradients and apply them to the student model's trainable variables.

\* \*\*Evaluate Student Model\*\*: Evaluate the trained student model on the test set and return the loss.

\* \*\*Parallel Parameter Optimization\*\*:

\* Create a list of hyperparameter combinations.

\* Use `client.map` to distribute the objective function evaluation across the Dask cluster.

\* Gather the results from all parallel evaluations.

\* Determine the best hyperparameter set based on the lowest loss.

### 4. Final Student Model Training with Optimized Hyperparameters

\* \*\*Define and Compile Student Model\*\*: Same as in the objective function, but using the best hyperparameters found in the previous step.

\* \*\*Training Loop\*\*: Similar to the knowledge distillation loop in the objective function, but now training the final student model for a set number of epochs. Print the loss after each epoch.

\* \*\*Evaluate Final Student Model\*\*: Evaluate the final trained student model on the test set.

### 5. Evaluation and Visualization

\* \*\*Calculate Prediction Probabilities\*\*: Use the trained student model to predict probabilities on the test set.

\* \*\*Generate ROC Curve\*\*:

\* Calculate the False Positive Rate (FPR) and True Positive Rate (TPR) for different thresholds.

\* Calculate the Area Under the ROC Curve (AUC).

\* \*\*Plot ROC Curve\*\*:

\* Plot the ROC curve (TPR vs FPR).

\* Add a diagonal line (representing a random classifier).

\* Label the axes and add a title.

\* Display the AUC score in the legend.

\* Show the plot.

Routine Map:

## Routine Map: Distributed Model Training with Dask

### 1. Initialization

\* Import necessary libraries (geopandas, numpy, tensorflow, dask).

\* Start a Dask client.

### 2. Data Preparation

\* Load Geographic Data:

\* Read data from URL into GeoDataFrame.

\* Convert to Web Mercator projection (EPSG:3857).

\* Generate Labels:

\* Create binary labels based on population threshold.

\* Extract Features:

\* Calculate centroid coordinates (x, y).

\* Split Data:

\* Divide data into training and testing sets.

### 3. Teacher Model Training

\* Define Teacher Model:

\* Sequential model with input layer, hidden layers (Dense, BatchNormalization), and output layer (sigmoid).

\* Compile Teacher Model:

\* Use Adam optimizer, binary cross-entropy loss, and accuracy metric.

\* Train Teacher Model:

\* Fit the model to training data with early stopping.

\* Generate Soft Labels:

\* Predict on training data using the teacher model.

\* Apply temperature scaling to logits.

### 4. Student Model Training (Distributed with Dask)

\* Define Objective Function (Input: Hyperparameters):

\* Define Student Model: Smaller sequential model with Dense, BatchNormalization, Dropout, and L2 regularization layers.

\* Compile Student Model: Use Adam optimizer with given learning rate.

\* Knowledge Distillation Training Loop:

\* For each epoch:

\* For each batch:

\* Calculate loss: `loss = alpha \* loss\_true + (1 - alpha) \* loss\_soft + beta \* regularization\_loss`

\* Calculate gradients and apply to student model.

\* Evaluate Student Model: Evaluate on the test set and return loss.

\* Parallel Parameter Optimization:

\* Create list of hyperparameter combinations.

\* Use `client.map` to distribute objective function evaluation.

\* Gather results and determine the best hyperparameters.

### 5. Final Student Model Training

\* Define and Compile Student Model: Using the best hyperparameters.

\* Training Loop: Train the final student model using knowledge distillation.

\* Evaluate Final Student Model: Evaluate on the test set.

### 6. Evaluation and Visualization

\* Calculate Prediction Probabilities: Use trained student model.

\* Generate ROC Curve:

\* Calculate FPR, TPR, and AUC.

\* Plot ROC Curve.

(2) Surrogate computing Pseudo Codes:

// Install necessary libraries

INSTALL geopandas

INSTALL tensorflow

INSTALL scikit-optimize

// Import necessary modules

IMPORT geopandas

IMPORT numpy

IMPORT train\_test\_split

IMPORT gp\_minimize, Real, Integer, use\_named\_args from scikit-optimize

IMPORT tensorflow

IMPORT Dropout from tensorflow.keras.layers

// --------------------------------------------------------------------

// Data Loading and Preprocessing

// --------------------------------------------------------------------

// Load geospatial data from URL

gdf = LOAD\_GEOSPATIAL\_DATA("https://naciscdn.org/naturalearth/110m/cultural/ne\_110m\_admin\_0\_countries.zip")

// Extract centroid coordinates

FOR EACH entry IN gdf:

gdf['x'] = CENTROID\_X(entry.geometry)

gdf['y'] = CENTROID\_Y(entry.geometry)

END FOR

// Generate binary labels based on population estimate

FOR EACH entry IN gdf:

IF entry['POP\_EST'] > 1e7:

gdf['label'] = 1 // High population

ELSE:

gdf['label'] = 0 // Low population

END IF

END FOR

// Prepare features and labels

features = EXTRACT\_FEATURES(gdf, ['x', 'y'])

labels = EXTRACT\_LABELS(gdf, 'label')

// Split data into training and testing sets

(x\_train, x\_test, y\_train, y\_test) = SPLIT\_DATA(features, labels, test\_size=0.2, random\_state=42)

// --------------------------------------------------------------------

// Bayesian Optimization Setup

// --------------------------------------------------------------------

// Define the hyperparameter search space

space = {

'learning\_rate': Real(1e-5, 1e-1, prior='log-uniform'),

'batch\_size': Integer(10, 100),

'alpha': Real(0.0, 1.0)

}

// Generate random soft labels (PLACEHOLDER - Replace with actual soft label generation)

soft\_labels = GENERATE\_RANDOM\_ARRAY(length(x\_train)) // Values between 0 and 1

// --------------------------------------------------------------------

// Objective Function Definition

// --------------------------------------------------------------------

FUNCTION objective(learning\_rate, batch\_size, alpha):

// Create the student model (neural network)

student\_model = CREATE\_SEQUENTIAL\_MODEL([

Dense(64, activation='relu', input\_shape=(2,), l2\_regularization=0.01),

Dropout(0.2),

Dense(1) // Output layer

])

// Define the optimizer

optimizer = Adam(learning\_rate=learning\_rate)

// Compile the model

student\_model.compile(optimizer=optimizer)

// Training parameters

beta = 0.01

num\_epochs = 10

num\_batches = length(x\_train) / batch\_size

// Training loop

FOR epoch IN 1 to num\_epochs:

FOR i IN 1 to num\_batches:

// Select a random batch of data

batch\_indices = RANDOM\_CHOICE(length(x\_train), size=batch\_size, replace=False)

x\_batch = x\_train[batch\_indices]

y\_batch = y\_train[batch\_indices]

y\_soft\_batch = soft\_labels[batch\_indices]

// Calculate loss and gradients

WITH GradientTape() as tape:

logits = student\_model(x\_batch)

loss\_true = binary\_crossentropy(y\_batch, logits)

loss\_soft = binary\_crossentropy(y\_soft\_batch, logits)

regularization\_loss = SUM(student\_model.losses)

loss = alpha \* loss\_true + (1 - alpha) \* loss\_soft + beta \* regularization\_loss

// Apply gradients

grads = tape.gradient(loss, student\_model.trainable\_variables)

student\_model.optimizer.apply\_gradients(zip(grads, student\_model.trainable\_variables))

END FOR

END FOR

// Evaluate the model

student\_model.compile(optimizer=optimizer, loss='binary\_crossentropy', metrics=['accuracy'])

loss, accuracy = student\_model.evaluate(x\_test, y\_test)

RETURN loss

END FUNCTION

// --------------------------------------------------------------------

// Run Bayesian Optimization

// --------------------------------------------------------------------

// Perform Bayesian optimization

res = gp\_minimize(objective, space, n\_calls=50, random\_state=42)

// Get the best parameters

best\_params = res.x

// Print the best parameters

PRINT "Best Parameters:", best\_params

+----------------+

| Start |

+----------------+

|

v

+-------------------------------+

| Install & Import Libraries |

| - geopandas, numpy, |

| tensorflow, scikit-optimize|

+-------------------------------+

|

v

+-------------------------------+

| Load Geospatial Data |

| (from the provided URL) |

+-------------------------------+

|

v

+-------------------------------+

| Extract Centroid Coordinates |

| - Compute "x" and "y" |

| from geometry |

+-------------------------------+

|

v

+-------------------------------+

| Generate Binary Labels |

| (based on population value) |

+-------------------------------+

|

v

+-------------------------------+

| Prepare Features & Labels |

| - Features: [x, y] |

| - Labels: binary values |

+-------------------------------+

|

v

+-------------------------------+

| Split Data into Train & Test |

| (using train\_test\_split) |

+-------------------------------+

|

v

+-----------------------------------------+

| Define Hyperparameter Search Space |

| - learning\_rate (Real, log-uniform) |

| - batch\_size (Integer range) |

| - alpha (Real: weight for loss mixing) |

+-----------------------------------------+

|

v

+-------------------------------+

| Generate Placeholder Soft |

| Labels for Training |

+-------------------------------+

|

v

+----------------------------------------------+

| Define Objective Function (for gp\_minimize) |

| - Create Student Model (Dense + Dropout) |

| - Compile Model with Adam optimizer |

| - Training Loop: |

| • For each epoch and batch: |

| - Select random training batch |

| - Get true labels & soft labels |

| - Compute logits, losses: |

| • Binary crossentropy (true & soft)|

| • Regularization loss |

| - Calculate gradients |

| - Update weights using optimizer |

| - Evaluate model on test data and return loss |

+----------------------------------------------+

|

v

+-------------------------------+

| Run Bayesian Optimization |

| (gp\_minimize optimizes objective) |

+-------------------------------+

|

v

+-------------------------------+

| Retrieve & Print Best |

| Hyperparameters |

+-------------------------------+

|

v

+----------------+

| End |

(3) FOA Bayesian Optimization pseudo code:

BEGIN Main

// --- Import Libraries ---

IMPORT geopandas

IMPORT tensorflow and keras

IMPORT scikit-optimize (for hyperparameter tuning)

// --- Load and Prepare Geospatial Data ---

gdf ← load geospatial data from file

IF gdf’s Coordinate Reference System is geographic THEN

// Warning: Centroid calculations may be inaccurate.

// Consider re-projecting to a projected CRS.

ENDIF

gdf["x"] ← compute the x-coordinate of the centroid for each geometry in gdf

gdf["y"] ← compute the y-coordinate of the centroid for each geometry in gdf

// --- Define the Neural Network Model ---

FUNCTION define\_neural\_network(input\_shape):

model ← initialize a Sequential model

model.add( Input layer with shape = input\_shape )

model.add( Dense layer with 64 units and ReLU activation )

model.add( Dense layer with 32 units and ReLU activation )

model.add( Dense output layer with 1 unit and linear activation )

model.compile( optimizer = Adam, loss = Mean Squared Error )

RETURN model

END FUNCTION

// --- Set Up Hyperparameter Search ---

hyperparameter\_space ← {

learning\_rate : continuous value between 0.0001 and 0.1 (log-uniform distribution),

batch\_size : integer between 16 and 128,

layer1\_units : integer between 32 and 128,

layer2\_units : integer between 16 and 64

}

bayes\_search\_cv ← initialize BayesSearchCV with:

estimator = define\_neural\_network,

search\_space = hyperparameter\_space,

scoring = negative Mean Squared Error,

cross-validation= 3 folds,

verbosity = high

// --- Data Splitting and Model Training ---

(train\_data, train\_labels), (test\_data, test\_labels) ← split the dataset appropriately

best\_model ← bayes\_search\_cv.fit( train\_data, train\_labels )

// --- Evaluate and Save the Model ---

evaluation\_metric ← best\_model.evaluate( test\_data, test\_labels )

save best\_model for future use

END Main