```
# ============================================================
# 2.2.2–2.2.4 ANN Model for Predicting Endemic Species Richness
# ============================================================

# Required Libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.inspection import permutation_importance
import tensorflow as tf
from tensorflow.keras import models, layers, optimizers, callbacks
import matplotlib.pyplot as plt
import seaborn as sns

# ========================================
# Load dataset
# ========================================
# Example: replace with your dataset
# df = pd.read_csv("endemic_species_data.csv")

# Example placeholder structure
# df = pd.DataFrame({
#     "temperature": np.random.rand(200),
#     "rainfall": np.random.rand(200),
#     "elevation": np.random.rand(200),
```

```
#     "soil_type": np.random.choice(["A", "B", "C"], 200),
#     "species_richness": np.random.rand(200) * 100
# })

# Identify predictors and response
predictor_vars = df.drop(columns=["species_richness"])
response_var = df["species_richness"]

# Identify categorical and numeric columns
categorical_cols = predictor_vars.select_dtypes(include=["object", "category"]).columns.tolist()
numeric_cols = predictor_vars.select_dtypes(exclude=["object", "category"]).columns.tolist()

# ========================================
# Preprocessing: Standardization and Encoding
# ========================================
preprocessor = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(), numeric_cols),
        ("cat", OneHotEncoder(drop="first"), categorical_cols)
    ]
)

# ========================================
# Split dataset: Training (80%) / Testing (20%)
# ========================================
X_train, X_test, y_train, y_test = train_test_split(
    predictor_vars, response_var, test_size=0.2, random_state=42
)
```

```
# ========================================
# Define model-building function (for grid search)
# ========================================
def build_model(hidden_neurons=4, learning_rate=0.001, l2_reg=0.0):
    model = models.Sequential([
        layers.Input(shape=(input_dim,)),
        layers.Dense(hidden_neurons, activation="relu", kernel_regularizer=tf.keras.regularizers.l2(l2_reg)),
        layers.Dense(1, activation="linear")
    ])
    optimizer = optimizers.Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss="mse")
    return model

# ========================================
# Pipeline with preprocessing and model
# ========================================
# Fit preprocessor to obtain transformed shape
X_train_proc = preprocessor.fit_transform(X_train)
input_dim = X_train_proc.shape[1]

# ========================================
# Grid Search for Hyperparameter Tuning (5-fold CV)
# ========================================
param_grid = {
    "hidden_neurons": [4, 8, 16],
    "learning_rate": [0.001, 0.005],
    "l2_reg": [0.0, 0.001, 0.01]
}
```

```
best_score = np.inf
best_params = None

kf = KFold(n_splits=5, shuffle=True, random_state=42)

for hn in param_grid["hidden_neurons"]:
    for lr in param_grid["learning_rate"]:
        for reg in param_grid["l2_reg"]:
            fold_mse = []
            for train_idx, val_idx in kf.split(X_train_proc):
                X_tr, X_val = X_train_proc[train_idx], X_train_proc[val_idx]
                y_tr, y_val = y_train.values[train_idx], y_train.values[val_idx]

                model = build_model(hidden_neurons=hn, learning_rate=lr, l2_reg=reg)
                es = callbacks.EarlyStopping(monitor='val_loss', patience=15, min_delta=0.001,
restore_best_weights=True)
                model.fit(X_tr, y_tr, validation_data=(X_val, y_val),
                          epochs=200, batch_size=16, verbose=0, callbacks=[es])
                y_pred_val = model.predict(X_val, verbose=0)
                fold_mse.append(mean_squared_error(y_val, y_pred_val))
            mean_mse = np.mean(fold_mse)
            if mean_mse < best_score:
                best_score = mean_mse
                best_params = {"hidden_neurons": hn, "learning_rate": lr, "l2_reg": reg}

print("Best hyperparameters:", best_params)

# ========================================
# Train final model with best parameters
```

```
# ========================================

final_model = build_model(**best_params)

es_final = callbacks.EarlyStopping(monitor='val_loss', patience=20, min_delta=0.001,
restore_best_weights=True)


history = final_model.fit(

   X_train_proc, y_train,

   validation_split=0.2,

   epochs=500,

   batch_size=16,

   callbacks=[es_final],

   verbose=1

)


# ========================================

# Model Evaluation on Testing Data

# ========================================

X_test_proc = preprocessor.transform(X_test)

y_pred = final_model.predict(X_test_proc).flatten()


rmse = np.sqrt(mean_squared_error(y_test, y_pred))

mae = mean_absolute_error(y_test, y_pred)

r2 = r2_score(y_test, y_pred)


print(f"Test RMSE: {rmse:.3f}")

print(f"Test MAE:  {mae:.3f}")

print(f"Test R²:   {r2:.3f}")


# ========================================
```

```
# Permutation Importance
# ========================================
def model_predict(X):
    return final_model.predict(preprocessor.transform(X)).flatten()

perm_results = permutation_importance(
    estimator=None, X=X_test, y=y_test, scoring="neg_mean_squared_error",
    n_repeats=10, random_state=42, n_jobs=-1,
    predictor=model_predict
)

importance = pd.DataFrame({
    "Variable": predictor_vars.columns,
    "Importance": np.abs(perm_results["importances_mean"])
}).sort_values(by="Importance", ascending=False)

print(importance)

# ========================================
# Plot Model Outputs vs Observed Data
# ========================================
sns.set(style="whitegrid")

# Predicted vs Observed
plt.figure(figsize=(8,6))
sns.regplot(x=y_test, y=y_pred, ci=95, scatter_kws={"alpha":0.6})
plt.xlabel("Observed Species Richness")
plt.ylabel("Predicted Species Richness (ANN)")
plt.title("ANN Predictions vs Observed Data")
```

```
plt.show()


# Residuals vs Observed
residuals = y_test - y_pred
plt.figure(figsize=(8,6))
sns.regplot(x=y_test, y=residuals, ci=95, scatter_kws={"alpha":0.6})
plt.axhline(0, color='red', linestyle='--')
plt.xlabel("Observed Species Richness")
plt.ylabel("Residuals (Observed - Predicted)")
plt.title("Residuals vs Observed Species Richness")
plt.show()
```