```python
# -*- coding: utf-8 -*-
"""
Predictive analysis of naval incidents in the USA, 2002 - 2015:
Functions for model performance comparison

@author: "Oscar Anton"
@date: "2024"
@license: "CC BY-NC-ND 4.0 DEED"
@version: "0.9"
"""

"""
Example:
import sys
sys.path.append('myCustomFunctions.py')
import myCustomFunctions as own

own.model_metrics(nb_MA_train, X_test, y_test, styled=True)
"""

# %% LIBRARIES

# System environment
import os

# Data general management
import numpy as np
import pandas as pd

# Visualization
import seaborn as sns
import matplotlib.pyplot as plt

# Model management
from sklearn.model_selection import learning_curve
from sklearn.preprocessing import LabelEncoder, label_binarize

# Model metrics
from sklearn.metrics import (accuracy_score, mean_squared_error, r2_score, f1_score,
                             mean_absolute_error, roc_auc_score, roc_curve, auc,
                             cohen_kappa_score, confusion_matrix, recall_score, precision_score)


# %% GENERAL VARIABLES
#  Available CPU cores for multiprocessing (training models)
n_jobs = os.cpu_count() - 1

# Label encoder
label_encoder = LabelEncoder()


# %% SKLEARN: PERFORMANCE FOR BINOMIAL CLASSIFICATION MODELS (5.1)

# Function: Table with main metrics data
def model_metrics(model, X, y):
    # Predictions (absolute)
    y_pred = model.predict(X)

    # Calculate main metrics
    roc_auc = round(roc_auc_score(y, y_pred), 4)
    accuracy = round(accuracy_score(y, y_pred), 4)
```

```python
    kappa = round(cohen_kappa_score(y, y_pred), 4)
    rmse = round(mean_squared_error(y, y_pred), 4)
    mae = round(mean_absolute_error(y, y_pred), 4)
    r2 = round(r2_score(y, y_pred), 4)
    f1 = round(f1_score(y, y_pred), 4)

    # Sensitivity And Specificity
    tn, fp, fn, tp = confusion_matrix(y, y_pred).ravel()
    sensitivity = round(tp / (tp + fn), 4)
    specificity = round(tn / (tn + fp), 4)

    # Build multiindex table
    metrics_df = pd.DataFrame([['ROC AUC:', roc_auc], ['Accuracy:', accuracy], ['Kappa:', kappa],
                               ['RMSE:', rmse], ['MAE:', mae], ['R2:', r2], ['F1:', f1], [' ', ' '],
                               ['Sensitivity:', sensitivity], ['Specificity:', specificity]],
                              columns=pd.MultiIndex.from_product([[model.__class__.__name__], ['Metric', 'Value']]))

    return metrics_df.style.hide()


# Function: Table with Confusion Matrix data
def confusion_matrix_table(model, X, y):
    # Predictions (absolute)
    y_pred = model.predict(X)

    # Confusion matrix
    tn, fp, fn, tp = confusion_matrix(y, y_pred).ravel()

    # Dataframe creation
    df = pd.DataFrame([[tp, fn], [fp, tn]],
                      index=pd.Index(['1', '0'], name='Actual Label:'),
                      columns=pd.MultiIndex.from_product([[model.__class__.__name__],['1', '0']],
                                                         names=['Model:', 'Predicted:']))

    # Dataframe style
    styled_df = df.style.set_table_styles([
        {'selector': 'th.col_heading', 'props': 'text-align: center;'},
        {'selector': 'td', 'props': 'text-align: center;'},
    ], overwrite=False)

    return styled_df


# Function: Plot ROC Curve
def plot_roc_curve(model, X, y):
    # Predicted probabilities
    y_score = model.predict_proba(X)

    # Calculate ROC for each class
    fpr, tpr, _ = roc_curve(y, y_score[:, 1])

    # Calculate AUC (Area Under Curve)
    roc_auc = auc(fpr, tpr)

    # Plot ROC Curve
    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = %0.2f)' % roc_auc)
```

```python
    plt.plot([0, 1], [0, 1], color='black', lw=1, linestyle='dotted')
    plt.xlim([0, 1])
    plt.ylim([0, 1.01])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC Curve for {model.__class__.__name__}')
    plt.legend(loc="lower right")
    plt.show()


# Function to plot learning curves
def plot_learning_curve(model, X, y, cv, train_sizes=np.linspace(.1, 1.0, 5)):
    plt.figure()
    plt.title(f"Learning Curve of {model}")
    plt.xlabel("Training examples")
    plt.ylabel("Score")

    train_sizes, train_scores, test_scores = learning_curve(
        model, X, y, cv=cv, train_sizes=train_sizes, n_jobs=n_jobs)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.grid()

    plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1, color="r")
    plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="g")
    plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Training score")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Cross-validation score")

    return plt


# %% -------------------------------------------------------------------------------------------------------


# %% SKLEARN: PERFORMANCE FOR MULTILEVEL CLASSIFICATION MODELS (5.2)

# Function: Table with main metrics data
def ma_model_metrics(model, X, y, styled=False):
    # Predictions (absolute)
    y_pred = model.predict(X)

    # Data binarize for auc calculation
    y_bin = label_binarize(y, classes=np.unique(y))
    y_pred_bin = label_binarize(y_pred, classes=np.unique(y))

    # Calculate main metrics
    roc_auc = round(roc_auc_score(y_bin, y_pred_bin), 4)
    accuracy = round(accuracy_score(y, y_pred), 4)
    kappa = round(cohen_kappa_score(y, y_pred), 4)
    rmse = round(mean_squared_error(y, y_pred), 4)
    mae = round(mean_absolute_error(y, y_pred), 4)
    r2 = round(r2_score(y, y_pred), 4)
    f1 = round(f1_score(y, y_pred, average='macro'), 4)

```

```python
    # Build multiindex table
    df = pd.DataFrame([['ROC AUC:', roc_auc], ['Accuracy:', accuracy], ['Kappa:', kappa],
                       ['RMSE:', rmse], ['MAE:', mae], ['R2:', r2], ['F1:', f1]],
                      columns=('metric', 'value'))

    if styled:
        title = f'{model.__class__.__name__} Training'
        df.columns = pd.MultiIndex.from_tuples([(title, col) for col in df.columns])
        return df.style.hide()
    else:
        return df


# Function: Table for recall & precision, sensitivity & specificity
def sens_spec(model, X, y, styled=False):
    # Predictions (absolute)
    y_pred = model.predict(X)

    # Recall & Precision values
    recall = round(recall_score(y, y_pred, average='macro'), 4)
    precision = round(precision_score(y, y_pred, average='macro'), 4)

    # Confusion matrix
    conf_matrix = confusion_matrix(y, y_pred)

    # List compression for sens & spec values calculation
    sensitivity, specificity = zip(*[(round(recall_score(y, y_pred, labels=[i], average='macro'), 4),
                                      round(conf_matrix[i, i] / sum(conf_matrix[:, i]), 4))
                                     for i in range(len(conf_matrix))])

    # Labels and indexes
    column_labels = label_encoder.inverse_transform(model.classes_)
    index_1 = ['Recall:', 'Precision:']
    index_2 = [recall, precision]
    index_ = [' - ', ' - ']
    index_3 = ['Sensitivity:', 'Specificity:']

    # Build multiindex table
    df = pd.DataFrame([sensitivity, specificity])
    df.columns = column_labels
    df.index = [index_1, index_2, index_, index_3]

    # Dataframe style
    if styled:
        title = f'{model.__class__.__name__} Model'
        df.columns = pd.MultiIndex.from_tuples([(title, col) for col in df.columns])
        return df.style.set_table_styles([{'selector': 'th.col_heading',
                                           'props': 'text-align: center;'}], overwrite=False)
    else:
        return df


# Function: Table with Confusion Matrix
def ma_confusion_matrix_table(model, X, y):
    # Predictions (absolute)
    y_pred = model.predict(X)

    # Get labels decoding target variable
    labels = label_encoder.inverse_transform(model.classes_)

```

```python
    # Build table
    df = pd.DataFrame(confusion_matrix(y, y_pred),
                      columns=pd.MultiIndex.from_product([[f'{model.__class__.__name__}: Confusion Matrix'], labels]),
                      index=labels)

    # Dataframe style
    styled_df = df.style.set_table_styles([
        {'selector': 'th.col_heading', 'props': 'text-align: center;'},
        {'selector': 'td', 'props': 'text-align: center;'},
    ], overwrite=False)

    return styled_df


# Function: Plot Multiclass ROC Curve
def roc_curve_plot(model, X, y):
    # Predictions (absolute)
    y_pred = model.predict(X)

    # Data binarize for auc calculation
    y_bin = label_binarize(y , classes=model.classes_)
    y_pred_bin = label_binarize(y_pred , classes=model.classes_)

    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in model.classes_:
        fpr[i], tpr[i], _ = roc_curve(y_bin[:, i], y_pred_bin[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(y_bin.ravel(), y_pred_bin.ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

    # Plot ROC curve for each class
    plt.figure()
    colors = sns.color_palette("hls", 5)
    for i, color in zip(model.classes_, colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=2,
                label='ROC of class {0} (AUC = {1:0.2f})'.format(i, roc_auc[i]))

    # Plot micro-average ROC curve
    plt.plot(fpr["micro"], tpr["micro"], color='grey', lw=1, linestyle='dashed',
            label='micro-average ROC (AUC = {0:0.2f})'.format(roc_auc["micro"]))

    plt.plot([0, 1], [0, 1], color='black', lw=1, linestyle='dotted')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.01])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC (Multiclass) for {model.__class__.__name__}')
    plt.legend(loc="lower right", fontsize="8")
    plt.show()


# Function: Feature importances
def sklearn_feature_importances(model, plot=False):
    importances = pd.DataFrame({'variable_name': model.feature_names_in_,
                                'value': model.feature_importances_}).sort_values(by='value', ascending=False)
```

```python
    # Plot horizontal bars if enabled in the call, otherwise return values
    if plot:
        plt.figure(figsize=(10, 7))
        plt.barh(importances['variable_name'], importances['value'], color='#00bfc4')
        plt.title(f"Feature importances of {model.__class__.__name__} model")
        plt.xlabel('Relative Feature importance')
        plt.gca().invert_yaxis()
        plt.show()
    else:
        return importances


# %% KERAS: PERFORMANCE FOR MULTILEVEL CLASSIFICATION MODELS (5.2)

# Function: Plot loss / accuracy train evolution
def keras_train_plot(data):
    # Train process visualization
    df_train = pd.DataFrame(data)
    # df_train['epochs']=history.epoch
    df_train['epochs'] = list(range(0, len(data['accuracy'])))

    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 6))

    fig.suptitle('Train process', fontsize=12)

    ax1.plot(df_train['epochs'], df_train['accuracy'], label='train_accuracy')
    ax1.plot(df_train['epochs'], df_train['val_accuracy'], label='val_accuracy')

    ax2.plot(df_train['epochs'], df_train['loss'], label='train_loss')
    ax2.plot(df_train['epochs'], df_train['val_loss'], label='val_loss')

    ax1.legend(loc='best')
    ax2.legend(loc='best')
    plt.show()


# Function: Table with main metrics data
def keras_model_metrics(model, X, y_ohe, styled=False):
    y_pred_ohe = pd.DataFrame(model.predict(X))
    y_pred_serie = y_pred_ohe.idxmax(axis=1)
    y_serie = pd.Series(label_encoder.fit_transform(y_ohe.idxmax(axis=1)))

    roc_auc = round(roc_auc_score(y_ohe, y_pred_ohe), 4)
    accuracy = round(accuracy_score(y_serie, y_pred_serie), 4)
    kappa = round(cohen_kappa_score(y_serie, y_pred_serie), 4)
    rmse = round(mean_squared_error(y_ohe, y_pred_ohe), 4)
    mae = round(mean_absolute_error(y_ohe, y_pred_ohe), 4)
    r2 = round(r2_score(y_ohe, y_pred_ohe), 4)
    f1 = round(f1_score(y_serie, y_pred_serie, average='macro'), 4)

    # Build multiindex table
    df = pd.DataFrame([['ROC AUC:', roc_auc], ['Accuracy:', accuracy], ['Kappa:', kappa],
                       ['RMSE:', rmse], ['MAE:', mae], ['R2:', r2], ['F1:', f1]],
                      columns=('metric', 'value'))

    if styled:
        title = f'{model.__class__.__name__} Training'
        df.columns = pd.MultiIndex.from_tuples([(title, col) for col in df.columns])
        return df.style.hide()
    else:
        return df

```

```python

# Function: Table for recall & precision, sensitivity & specificity
def keras_sens_spec(model, X, y, styled=False):
    # Predictions (absolute)
    y_pred_ohe = pd.DataFrame(model.predict(X))
    y_pred_serie = y_pred_ohe.idxmax(axis=1)
    y_serie = pd.Series(label_encoder.fit_transform(y.idxmax(axis=1)))

    # Recall & Precision values
    recall = round(recall_score(y_serie, y_pred_serie, average='macro'), 4)
    precision = round(precision_score(y_serie, y_pred_serie, average='macro'), 4)

    # Confusion matrix
    conf_matrix = confusion_matrix(y_serie, y_pred_serie)

    # List compression for sens & spec values calculation
    sensitivity, specificity = zip(*[(round(recall_score(y_serie, y_pred_serie, labels=[i], average='macro'), 4),
                                      round(conf_matrix[i, i] / sum(conf_matrix[:, i]), 4))
                                      for i in range(len(conf_matrix))])

    # Labels and indexes
    column_labels = y.columns
    index_1 = ['Recall:', 'Precision:']
    index_2 = [recall, precision]
    index_ = [' - ', ' - ']
    index_3 = ['Sensitivity:', 'Specificity:']

    # Build multiindex table
    df = pd.DataFrame([sensitivity, specificity])
    df.columns = column_labels
    df.index = [index_1, index_2, index_, index_3]

    # Dataframe style
    if styled:
        title = f'{model.__class__.__name__} Model'
        df.columns = pd.MultiIndex.from_tuples([(title, col) for col in df.columns])
        return df.style.set_table_styles([{'selector': 'th.col_heading',
                                           'props': 'text-align: center;'}], overwrite=False)
    else:
        return df


# Function: Table with Confusion Matrix
def keras_confusion_matrix_table(model, X, y):
    # Predictions (max)
    y_pred_max = np.argmax(model.predict(X), axis=1)
    y_max = np.argmax(y, axis=1)

    # Build table
    df = pd.DataFrame(confusion_matrix(y_max, y_pred_max),
                              columns=pd.MultiIndex.from_product([[f'{model.name}: Confusion Matrix'], y.columns]),
                              index=y.columns)

    # Dataframe style
    styled_df = df.style.set_table_styles([
        {'selector': 'th.col_heading', 'props': 'text-align: center;'},
        {'selector': 'td', 'props': 'text-align: center;'},
    ], overwrite=False)

```

```python
    return styled_df


# Function: Plot Multiclass ROC Curve
def keras_roc_curve_plot(model, X, y):
    # Predictions (max)
    y_pred = pd.DataFrame(model.predict(X))
    y_pred.columns = label_encoder.inverse_transform(y_pred.columns)

    # Compute ROC curve and ROC area for each class
    fpr, tpr, roc_auc = {}, {}, {}
    for i in y.columns:
        fpr[i], tpr[i], _ = roc_curve(y[i], y_pred[i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(y.values.ravel(), y_pred.values.ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

    # Plot ROC curve for each class
    plt.figure()
    colors = sns.color_palette("hls", 5)
    for i, color in zip(y.columns, colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=2,
                label='ROC of class {0} (AUC = {1:0.2f})'.format(i, roc_auc[i]))

    # Plot micro-average ROC curve
    plt.plot(fpr["micro"], tpr["micro"], color='grey', lw=1, linestyle='dashed',
            label='micro-average ROC (AUC = {0:0.2f})'.format(roc_auc["micro"]))

    plt.plot([0, 1], [0, 1], color='black', lw=1, linestyle='dotted')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.01])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC (Multiclass) for {model.name}')
    plt.legend(loc="lower right", fontsize="8")
    plt.show()


# Function: Plot feature importances
def keras_sec_importances(model, X, plot=False):
    # Calculate input layer weights
    weights = model.layers[0].get_weights()[0]

    # Create dataframe for weights and variable names
    importances = pd.DataFrame({'variable_name': X.columns,
                                'value': np.mean(np.abs(weights), axis=1)}).sort_values(by='value', ascending=False)

    # Plot horizontal bars if enabled in the call, otherwise return values
    if plot:
        plt.figure(figsize=(10, 7))
        plt.barh(importances['variable_name'], importances['value'], color='#00bfc4')
        plt.title(f"Feature importances in Keras {model.__class__.__name__} model")
        plt.xlabel('Relative Feature importance (based on first layer weights)')
        plt.gca().invert_yaxis()
        plt.show()
    else:
        return importances
```

```
# Function: Plot feature importances
def keras_func_importances(model, plot = False):
    weight_data = []
    # Considering all entrance dense layers have a name like corresponding variable name
    # Iterate through dense layers with a particular name, obtaining their weights
    for layer in model.layers:
        if 'Dense' in layer.__class__.__name__ and 'dense_' not in layer.name:
            layer_name = layer.name
            weights = np.mean(np.abs(layer.get_weights()[0]), axis=1)
            for i, weight in enumerate(weights):
                weight_data.append([f"{layer_name}_{i}", weight])

    # Build dataframe
    importances = pd.DataFrame(weight_data, columns=['variable_name', 'value']).sort_values(by='value', ascending=False)

    # Plot horizontal bars if enabled in the call, otherwise return values
    if plot:
        plt.figure(figsize=(10, 7))
        plt.barh(importances['variable_name'], importances['value'], color='#00bfc4')
        plt.title(f"Feature importances in Keras {model.__class__.__name__} model")
        plt.xlabel('Relative Feature importance (based on first layer weights)')
        plt.gca().invert_yaxis()
        plt.show()
    else:
        return importances



# %% H2O: PERFORMANCE FOR MULTILEVEL CLASSIFICATION MODELS (5.2)

# Function: Table with main metrics data
def h2o_model_metrics(h2o_model, h2o_test, styled=False):
    h2o_predict = pd.Series(label_encoder.fit_transform(h2o_model.predict(h2o_test)['predict'].as_data_frame()))
    h2o_y = pd.Series(label_encoder.fit_transform(h2o_test['y'].as_data_frame()))

    h2o_y_bin = label_binarize(h2o_y, classes=np.unique(h2o_y))
    h2o_predict_bin = label_binarize(h2o_predict, classes=np.unique(h2o_y))

    # Calculate main metrics
    roc_auc = round(roc_auc_score(h2o_y_bin, h2o_predict_bin), 4)
    accuracy = round(accuracy_score(h2o_y_bin, h2o_predict_bin), 4)
    kappa = round(cohen_kappa_score(h2o_y, h2o_predict), 4)
    rmse = round(mean_squared_error(h2o_y_bin, h2o_predict_bin), 4)
    mae = round(mean_absolute_error(h2o_y_bin, h2o_predict_bin), 4)
    r2 = round(r2_score(h2o_y_bin, h2o_predict_bin), 4)
    f1 = round(f1_score(h2o_y_bin, h2o_predict_bin, average='macro'), 4)

    # Build multiindex table
    df = pd.DataFrame([['ROC AUC:', roc_auc], ['Accuracy:', accuracy], ['Kappa:', kappa],
                       ['RMSE:', rmse], ['MAE:', mae], ['R2:', r2], ['F1:', f1]],
                       columns=('metric', 'value'))

    if styled:
        title = f'{h2o_model.key} Training'
        df.columns = pd.MultiIndex.from_tuples([(title, col) for col in df.columns])
        return df.style.hide()
    else:
        return df
```

```python
# Function: Table for recall & precision, sensitivity & specificity
def h2o_sens_spec(h2o_model, h2o_test, styled=False):
    # Predictions
    h2o_pred = pd.Series(label_encoder.fit_transform(h2o_model.predict(h2o_test)['predict'].as_data_frame()))
    h2o_y = pd.Series(label_encoder.fit_transform(h2o_test['y'].as_data_frame()))

    # Recall & Precision values
    recall = round(recall_score(h2o_y, h2o_pred, average='macro'), 4)
    precision = round(precision_score(h2o_y, h2o_pred, average='macro'), 4)

    # Confusion matrix
    conf_matrix = confusion_matrix(h2o_y, h2o_pred)

    # List compression for sens & spec values calculation
    sensitivity, specificity = zip(*[(round(recall_score(h2o_y, h2o_pred, labels=[i], average='macro'), 4),
                                    round(conf_matrix[i, i] / sum(conf_matrix[:, i]), 4))
                                    for i in range(len(conf_matrix))])

    # Labels and indexes
    column_labels = np.unique(h2o_test['y'].as_data_frame())
    index_1 = ['Recall:', 'Precision:']
    index_2 = [recall, precision]
    index_ = [' - ', ' - ']
    index_3 = ['Sensitivity:', 'Specificity:']

    # Build multiindex table
    df = pd.DataFrame([sensitivity, specificity])
    df.columns = column_labels
    df.index = [index_1, index_2, index_, index_3]

    # Dataframe style
    if styled:
        title = f'{h2o_model.key} Model'
        df.columns = pd.MultiIndex.from_tuples([(title, col) for col in df.columns])
        return df.style.set_table_styles([{'selector': 'th.col_heading',
                                           'props': 'text-align: center;'}], overwrite=False)
    else:
        return df


```