Instituto Politécnico Nacional

Escuela Superior de Computo

Métodos validación PL1

Inteligencia Artificial

Profesor:

Catalán Salgado Edgar Armando

Alumnos:

Briones Cruz Juan Carlos

Espinosa Vergara David Daniel

Hernández Reyes Julio Cesar

Vargas Velez Angel Isaac

Fecha:

12 Enero 2024

METODOS DE VALIDACION

Desarrolla la funcionalidad solicitada

REQUISITOS GENERALES

1. El sistema debe permitir cargar la base de datos con la cual se trabajara
2. Debe permitir especificar los atributos(columnas) a utilizar para construir el vector de entrada
3. Debe permitir especificar los atributos (columnas) a utilizar para construir el vector de sallida Y

Aquí se cumplen los tres puntos solicitados, el primero al poder ingresar cualquier base de datos a la práctica, en nuestro caso se usó el archivo Iris.data, que contiene los datos de las flores de iris vistas en el curso. Y después como se puede ver en la ejecución de la función especificar atributos, uno como usuario puede habilitar como deshabilitar las columnas o atributos que se deseen o en todo caso especificar las muestras con las que trabajar como atributos.

```python
if __name__ == "__main__":

    # Lee el archivo CSV
    input_data, output_data, column_names = add_dataset('iris.data')
    # Obtener clases
    clases, subsets_por_clase = obtener_clases(input_data, output_data)

    nuevodf, X, Y= especificaratributos(input_data, output_data, subsets_por_clase)
```

```
erParcial/Practica4/Practica4.py
(150, 4)
Tenemos las siguitente cantidad de clases(columnas) 4
Para habilitar 1, deshabilitar 0
Habilite los atributos (columas) que desea tener en su vector de entrada. Ej. 0, 0, 1, 0:
1,1,1,1
Tenemos la siguiente cantidad de muestras 150
Use la notacion de segmentación, Ej. [X:N], [:N], [X:], , etc
Habilite los atributos (filas) que desea tener en su vector de salida.
0:149
Conjunto de datos creado:

[[5.1 3.5 1.4 0.2 'Iris-setosa']
 [4.9 3.0 1.4 0.2 'Iris-setosa']
 [nan 3.2 1.3 0.2 'Iris-setosa']
 [4.6 3.1 1.5 0.2 'Iris-setosa']
 [5.0 3.6 1.4 0.2 'Iris-setosa']
 [5.4 3.9 1.7 0.4 'Iris-setosa']
 [4.6 3.4 1.4 0.3 'Iris-setosa']
 [5.0 3.4 1.5 0.2 'Iris-setosa']
 [4.4 2.9 1.4 0.2 'Iris-setosa']
 [4.9 3.1 1.5 0.1 'Iris-setosa']
 [5.4 3.7 1.5 0.2 'Iris-setosa']
 [4.8 3.4 1.6 0.2 'Iris-setosa']
 [4.8 3.0 1.4 0.1 'Iris-setosa']
 [4.3 3.0 1.1 0.1 'Iris-setosa']
 [5.8 4.0 1.2 0.2 'Iris-setosa']
 [5.7 4.4 1.5 0.4 'Iris-setosa']
 [5.4 3.9 1.3 0.4 'Iris-setosa']
 [5.1 3.5 1.4 0.3 'Iris-setosa']
 [5.7 3.8 1.7 0.3 'Iris-setosa']
```

TRAIN AND TEST

Implementa el metodo de validacion train and test, considerando que:

1. Debe permitir especificar el porcentaje de muestras para el aprendizaje
2. Debe de calcular el porcentaje de eficiencia y error

Como se puede ver se permite especificar la cantidad de muestras para el aprendizaje, y en el ejemplo se indico que se usara un 80 porciento para aprendizaje, dejando un 20 para pruebas. Y de la misma forma se calcula el porcentaje de eficiencia y error usando la función de nuestro clasificador relacionada con train_and_test:

```
# Crea una instancia de la clase CustomClassifierP3
classifier = CustomClassifierP4(input_size=X.shape[1], output_size=len(np.unique(Y)))

# Agrega los datos al clasificador
for input_vector, output_value in zip(X, Y):
    classifier.add_data_point(input_vector, output_value)

# Train and Test
train_percentage = input("Cual es el porcentaje para las muestras para el aprendizaje (0.0 - 1.0): ")
train_percentage = float(train_percentage)
#train_percentage = 0.8
efficiency, error = classifier.train_and_test(train_percentage)
print(f"\nEficiencia de Train and Test: {efficiency * 100:.2f}%")
print(f"Error de Train and Test: {error * 100:.2f}%")
```

```
Cual es el porcentaje para las muestras para el aprendizaje (0.0 - 1.0): 0.8
Usando 120 muestras para entrenamiento y 30 muestras para validación.

Eficiencia de Train and Test: 96.67%
Error de Train and Test: 3.33%
```

K FOLD CROSS VALIDATION

Implementa el metodo de validacion K fold cross validation, considerando que:

1. Debe permitir especificar la cantidad de grupos (K)
2. Debe calcular el porcentaje de eficiencia y error para cada grupo
3. Debe calcular el porcentaje de eficiencia y error generales y su respectiva desviacion estandar

Se puede especificar la cantidad de grupos para realizar las pruebas y de esa forma poder hacer el calculo de los porcentajes de eficiencia y error para cada grupo, en el ejemplo se uso k=5, de esta forma se crearon 5 grupos como se ve en la ejecución y después a estos se les calcula los porcentajes de eficiencia y error generales así como su respectiva deviación estándar:

```python
# K-Fold Cross Validation
cantidaddegruposk = input("Cual es la cantidad de grupos (k): ")
cantidaddegruposk = int(cantidaddegruposk)
#cantidaddegruposk = 5
test_percentage = 1.0 - train_percentage

kfold_accuracies, kfold_errors = classifier.kfold_cross_validation(cantidaddegruposk, train_percentage, test_percentage)

# Imprimir resultados generales
kfold_accuracy = np.mean(kfold_accuracies)
kfold_error = np.mean(kfold_errors)
std_kfold_accuracy = np.std(kfold_accuracies)
std_kfold_error = np.std(kfold_errors)

# Imprimir resultados para cada grupo
for i, (accuracy, error) in enumerate(zip(kfold_accuracies, kfold_errors), start=1):
    print(f"\nResultados para el Grupo {i}:")
    print(f"Eficiencia: {accuracy * 100:.2f}%")
    print(f"Error: {error * 100:.2f}%")

print(f"\nResultados Generales de K-Fold Cross Validation:")
print(f"Eficiencia de K-Fold: {kfold_accuracy * 100:.2f}%")
print(f"Error de K-Fold: {kfold_error * 100:.2f}%")
print(f"Desviación Estandar de la Eficiencia: {std_kfold_accuracy * 100:.2f}%")
print(f"Desviación Estandar del Error: {std_kfold_error * 100:.2f}%")
```

```
Cual es la cantidad de grupos (k): 5

Resultados para el Grupo 1:
Eficiencia: 100.00%
Error: 0.00%

Resultados para el Grupo 2:
Eficiencia: 100.00%
Error: 0.00%

Resultados para el Grupo 3:
Eficiencia: 90.00%
Error: 10.00%

Resultados para el Grupo 4:
Eficiencia: 93.33%
Error: 6.67%

Resultados para el Grupo 5:
Eficiencia: 100.00%
Error: 0.00%

Resultados Generales de K-Fold Cross Validation:
Eficiencia de K-Fold: 96.67%
Error de K-Fold: 3.33%
Desviación Estandar de la Eficiencia: 4.22%
Desviación Estandar del Error: 4.22%
```

BOTSTRAP

Implementa el metodo bootstrap, considerando que:

1. Debe permitir especificar la cantidad de experimentos(K)
2. Debe permitir especificar la cantidad de muestras en el conjunto de aprendizaje
3. Debe permitir especificar la cantidad de muestras en el conjunto de prueba
4. Debe calcular el porcentaje de eficiencia y error para en cada grupo y para cada clase
5. Debe calcular el porcentaje de eficiencia y error generales y su respectiva

Aquí se permite especificar la cantidad de experimentos(k), en el ejemplo se usaron 10 pruebas, se sigue manteniendo la cantidad de muestras en el conjunto de aprendizaje anterior pues si se requiere cambiar se tiene que correr de nuevo. Después se toma la cantidad de muestras en el conjunto de prueba como el mismo valor antes seleccionado. Teniendo esto se procede a calcular el porcentaje de eficiencia y error para cada grupo y para cada clase. Al final se obtienen o se calculan los porcentajes de eficiencia y error generales así como su respectiva desviación estándar:

```
# Bootstrap
num_iterations = input("Cual es el cantidad de experimentos: ")
num_iterations = int(num_iterations)

bootstrap_accuracies, bootstrap_errors = classifier.bootstrap(num_iterations, train_percentage, test_percentage)

# Imprimir resultados por grupo
for i in range(len(bootstrap_accuracies)):
    efficiency = bootstrap_accuracies[i]
    error = bootstrap_errors[i]
    print(f"\nResultados para el Grupo {i + 1}:")
    print(f"Eficiencia de Bootstrap: {efficiency * 100:.2f}%")
    print(f"Error de Bootstrap: {error * 100:.2f}%")

# Calcular y imprimir resultados generales
mean_accuracy = np.mean(bootstrap_accuracies)
mean_error = np.mean(bootstrap_errors)
std_accuracy = np.std(bootstrap_accuracies)
std_error = np.std(bootstrap_errors)

print(f"\nResultados Generales de Bootstrap:")
print(f"Eficiencia de Bootstrap: {mean_accuracy * 100:.2f}%")
print(f"Error de Bootstrap: {mean_error * 100:.2f}%")
print(f"Desviación Estandar de la Eficiencia: {std_accuracy * 100:.2f}%")
print(f"Desviación Estandar del Error: {std_error * 100:.2f}%")
```

```
Cual es el cantidad de experimentos: 10

Resultados para el Grupo 1:
Eficiencia de Bootstrap: 100.00%
Error de Bootstrap: 0.00%

Resultados para el Grupo 2:
Eficiencia de Bootstrap: 100.00%
Error de Bootstrap: 0.00%

Resultados para el Grupo 3:
Eficiencia de Bootstrap: 100.00%
Error de Bootstrap: 0.00%

Resultados para el Grupo 5:
Eficiencia de Bootstrap: 93.10%
Error de Bootstrap: 6.90%

Resultados para el Grupo 6:
Eficiencia de Bootstrap: 89.66%
Error de Bootstrap: 10.34%

Resultados para el Grupo 7:
Eficiencia de Bootstrap: 96.55%
Error de Bootstrap: 3.45%

Resultados para el Grupo 8:
Eficiencia de Bootstrap: 96.55%
Error de Bootstrap: 3.45%

Resultados para el Grupo 9:
Eficiencia de Bootstrap: 100.00%
Error de Bootstrap: 0.00%

Resultados para el Grupo 10:
Eficiencia de Bootstrap: 100.00%
```

```
Resultados Generales de Bootstrap:
Eficiencia de Bootstrap: 97.59%
Error de Bootstrap: 2.41%
Desviación Estandar de la Eficiencia: 3.47%
Desviación Estandar del Error: 3.47%
```

CODIGO COMPLETO:

A continuación se muestra el código completo de la practica:

```
import numpy as np
import pandas as pd


class CustomClassifierP4:
    def __init__(self, input_size, output_size):
        # Inicialización de la clase con el tamaño de entrada y salida
        self.input_size = input_size
        self.output_size = output_size
        # Inicialización de matrices para almacenar datos de entrada y salida
        self.input_data = np.array([]).reshape(0, input_size)
        self.output_data = np.array([])

    def add_data_point(self, input_vector, output_value):
        # Agrega un punto de datos al conjunto de entrenamiento
        self.input_data = np.vstack([self.input_data, input_vector])
        self.output_data = np.append(self.output_data, output_value)

    def euclidean_distance(self, x1, x2):
        # Calcula la distancia euclidiana entre dos vectores
        return np.sqrt(np.sum((x1 - x2)**2))

    def manhattan_distance(self, x1, x2):
        # Calcula la distancia de Manhattan entre dos vectores
        return np.sum(np.abs(x1 - x2))

    def train_knn(self, k):
        # Establece el valor de k para el algoritmo k-NN
        self.k = k

    def predict_knn(self, input_vector):
        # Predice la clase utilizando el algoritmo k-NN
        distances = [self.euclidean_distance(input_vector, x) for x in self.input_data]
        indices = np.argsort(distances)[:self.k]
        k_nearest_labels = self.output_data[indices]
        unique_labels, counts = np.unique(k_nearest_labels, return_counts=True)
        return unique_labels[np.argmax(counts)]

    def train_min_distance(self):
        # Calcular los promedios de los datos de entrenamiento para Mínima Distancia por
clase
        unique_classes = np.unique(self.output_data)
        self.class_means = {label: np.mean(self.input_data[self.output_data == label],
axis=0) for label in unique_classes}

    def predict_min_distance(self, input_vector):
        # Predice la clase utilizando el algoritmo de Mínima Distancia
```

```
        distances = {label: self.euclidean_distance(input_vector, mean) for label, mean in
self.class_means.items()}
        min_class = min(distances, key=distances.get)
        return min_class

    def train_and_test(self, train_percentage):
        # Dividir el conjunto de datos en entrenamiento y prueba
        num_samples = len(self.input_data)
        num_train_samples = int(train_percentage * num_samples)
        test_input = self.input_data[num_train_samples:]
        test_output = self.output_data[num_train_samples:]

        print(f"Usando {num_train_samples} muestras para entrenamiento y {num_samples -
num_train_samples} muestras para validación.")

        # Entrenar el clasificador con el conjunto de entrenamiento
        self.train_knn(k=3)  # Puedes ajustar el valor de k según sea necesario
        self.train_min_distance()

        # Realizar predicciones en el conjunto de prueba
        correct_predictions = 0
        for i in range(len(test_input)):
            input_vector = test_input[i]
            true_output = test_output[i]

            # Hacer predicciones con ambos algoritmos y comparar con la verdad
            knn_prediction = self.predict_knn(input_vector)
            min_distance_prediction = self.predict_min_distance(input_vector)

            # Contar la predicción correcta si al menos uno de los algoritmos acierta
            if knn_prediction == true_output or min_distance_prediction == true_output:
                correct_predictions += 1

        # Calcular la eficiencia y el error
        efficiency = correct_predictions / len(test_input)
        error = 1 - efficiency

        return efficiency, error

    def bootstrap(self, num_iterations, train_percentage, test_percentage):
        accuracies = []
        errors = []

        if train_percentage + test_percentage > 1.0:
            raise ValueError("La suma de los porcentajes de entrenamiento y prueba no puede
ser mayor que 1.")

        total_samples = len(self.input_data)
        train_samples = int(train_percentage * total_samples)
        test_samples = int(test_percentage * total_samples)
```

```
        for iteration in range(num_iterations):
            # Muestreo con reemplazo para crear un nuevo conjunto de entrenamiento y prueba
            train_indices = np.random.choice(total_samples, size=train_samples, replace=True)
            test_indices = np.random.choice(total_samples, size=test_samples, replace=True)

            bootstrap_train_input = self.input_data[train_indices]
            bootstrap_train_output = self.output_data[train_indices]
            bootstrap_test_input = self.input_data[test_indices]
            bootstrap_test_output = self.output_data[test_indices]

            # Entrenar el clasificador con el conjunto de entrenamiento bootstrap
            self.train_knn(k=5)  # Puedes ajustar el valor de k según sea necesario
            self.train_min_distance()

            # Realizar predicciones en el conjunto de prueba bootstrap
            correct_predictions = 0
            for i in range(len(bootstrap_test_input)):
                input_vector = bootstrap_test_input[i]
                true_output = bootstrap_test_output[i]

                # Hacer predicciones con ambos algoritmos y comparar con la verdad
                knn_prediction = self.predict_knn(input_vector)
                min_distance_prediction = self.predict_min_distance(input_vector)

                # Contar la predicción correcta si al menos uno de los algoritmos acierta
                if knn_prediction == true_output or min_distance_prediction == true_output:
                    correct_predictions += 1

            # Calcular la eficiencia y el error y agregarlos a las listas
            efficiency = correct_predictions / len(bootstrap_test_input)
            error = 1 - efficiency
            accuracies.append(efficiency)
            errors.append(error)

        return accuracies, errors

    def kfold_cross_validation(self, k, train_percentage, test_percentage):
        accuracies = []
        errors = []

        if train_percentage + test_percentage > 1.0:
            raise ValueError("La suma de los porcentajes de entrenamiento y prueba no puede
ser mayor que 1.")

        num_samples = len(self.input_data)
        fold_size = int(num_samples / k)
        train_samples = int(train_percentage * num_samples)
        test_samples = int(test_percentage * num_samples)
```

```
        for i in range(k):
            # Índices para el conjunto de prueba
            start_test = i * fold_size
            end_test = (i + 1) * fold_size if i != k - 1 else num_samples
            test_indices = np.arange(start_test, end_test)

            # Índices para el conjunto de entrenamiento
            train_indices = np.concatenate([np.arange(0, start_test), np.arange(end_test,
num_samples)])

            train_input = self.input_data[train_indices]
            train_output = self.output_data[train_indices]
            test_input = self.input_data[test_indices]
            test_output = self.output_data[test_indices]

            # Entrenar el clasificador con el conjunto de entrenamiento k-fold
            self.train_knn(k=5)  # Puedes ajustar el valor de k según sea necesario
            self.train_min_distance()

            # Realizar predicciones en el conjunto de prueba k-fold
            correct_predictions = 0
            for i in range(len(test_input)):
                input_vector = test_input[i]
                true_output = test_output[i]

                # Hacer predicciones con ambos algoritmos y comparar con la verdad
                knn_prediction = self.predict_knn(input_vector)
                min_distance_prediction = self.predict_min_distance(input_vector)

                # Contar la predicción correcta si al menos uno de los algoritmos acierta
                if knn_prediction == true_output or min_distance_prediction == true_output:
                    correct_predictions += 1

            # Calcular la eficiencia y agregarla a la lista
            efficiency = correct_predictions / len(test_input)
            accuracies.append(efficiency)

            # Calcular el error y agregarlo a la lista
            error = 1 - efficiency
            errors.append(error)

        return accuracies, errors

def add_dataset(path):
    try:
        # Leemos los datos desde el archivo CSV informando la dirección
        data = pd.read_csv(path)
        # Obtenemos los nombres de las columnas
        column_names = data.columns.tolist()
        # Se convierte el dataframe a numpy-array los datos a un NumPy array
```

```
        data_array = data.to_numpy()
        # separamos el dataset en las columnas de entrada y salida (x, y)
        input_data = data_array[:, :-1].astype(float)
        output_data = data_array[:, -1]

        return input_data, output_data, column_names
    except Exception as e:
        print(f"Error al cargar el conjunto de datos: {e}")
        return None, None, None
def obtener_clases(x, y):
    ## Obtenemos la clase con base en sus etiquetas [y]
    clases_unicas = np.unique(y)

    ## Fragmentamos las clases en arrays diferentes 1 x clase
    subsets = [x[y == cls] for cls in clases_unicas]
    return clases_unicas, subsets

def especificaratributos(x, y, subsets_por_clase):
    new_x = np.concatenate(subsets_por_clase)
    new_y = y

    print(new_x.shape)

    print(f"Tenemos las siguitente cantidad de clases(columnas) {new_x.shape[1]}")
    print(f"Para habilitar 1, deshabilitar 0")
    input_size = (input("Habilite los atributos (columas) que desea tener en su vector de
entrada. Ej. 0, 0, 1, 0: \n"))

    # Solicita al usuario los tamaños de salida
    print(f"Tenemos la siguiente cantidad de muestras {new_x.shape[0]}")
    print(f"Use la notacion de segmentación, Ej. [X:N], [:N], [X:], , etc")
    output_size = (input("Habilite los atributos (filas) que desea tener en su vector de
salida.\n"))

    habilitar = input_size.split(',')
    habilitar = [int(splits) for splits in habilitar]
    counter = 0
    to_delete = []
    for split in habilitar:
        if split == 0:
            to_delete.append(counter)
        counter = counter + 1
    #print(to_delete)

    new_xMod = np.delete(new_x, to_delete, axis=1)
    #print(new_xMod)


    filas_seleccionadas = output_size.split(":")
    filas_seleccionadas = [int(splits) for splits in filas_seleccionadas]
```

```
    output_dataMod = y[filas_seleccionadas[0]: filas_seleccionadas[1]+1]
    new_xMod= new_xMod[filas_seleccionadas[0]: filas_seleccionadas[1]+1]
    #print(new_xMod)

    print("Conjunto de datos creado: \n")
    nuevodf = np.column_stack((new_xMod, output_dataMod))
    print(nuevodf)
    return nuevodf, new_xMod, output_dataMod

#############################################################################################
##############
if __name__ == "__main__":

    # Lee el archivo CSV
    input_data, output_data, column_names = add_dataset('iris.data')
    # Obtener clases
    clases, subsets_por_clase = obtener_clases(input_data, output_data)

    nuevodf, X, Y= especificaratributos(input_data, output_data, subsets_por_clase)

    # Crea una instancia de la clase CustomClassifierP3
    classifier = CustomClassifierP4(input_size=X.shape[1], output_size=len(np.unique(Y)))

    # Agrega los datos al clasificador
    for input_vector, output_value in zip(X, Y):
        classifier.add_data_point(input_vector, output_value)

    # Train and Test
    train_percentage = input("Cual es el porcentaje para las muestras para el aprendizaje
(0.0 - 1.0): ")
    train_percentage = float(train_percentage)
    #train_percentage = 0.8
    efficiency, error = classifier.train_and_test(train_percentage)
    print(f"\nEficiencia de Train and Test: {efficiency * 100:.2f}%")
    print(f"Error de Train and Test: {error * 100:.2f}%")

    # K-Fold Cross Validation
    cantidaddegruposk = input("Cual es la cantidad de grupos (k): ")
    cantidaddegruposk = int(cantidaddegruposk)
    #cantidaddegruposk = 5
    test_percentage = 1.0 - train_percentage

    kfold_accuracies, kfold_errors = classifier.kfold_cross_validation(cantidaddegruposk,
train_percentage, test_percentage)

    # Imprimir resultados generales
    kfold_accuracy = np.mean(kfold_accuracies)
    kfold_error = np.mean(kfold_errors)
    std_kfold_accuracy = np.std(kfold_accuracies)
```

```
    std_kfold_error = np.std(kfold_errors)

    # Imprimir resultados para cada grupo
    for i, (accuracy, error) in enumerate(zip(kfold_accuracies, kfold_errors), start=1):
        print(f"\nResultados para el Grupo {i}:")
        print(f"Eficiencia: {accuracy * 100:.2f}%")
        print(f"Error: {error * 100:.2f}%")

    print(f"\nResultados Generales de K-Fold Cross Validation:")
    print(f"Eficiencia de K-Fold: {kfold_accuracy * 100:.2f}%")
    print(f"Error de K-Fold: {kfold_error * 100:.2f}%")
    print(f"Desviación Estandar de la Eficiencia: {std_kfold_accuracy * 100:.2f}%")
    print(f"Desviación Estandar del Error: {std_kfold_error * 100:.2f}%")


    # Bootstrap
    num_iterations = input("Cual es el cantidad de experimentos: ")
    num_iterations = int(num_iterations)

    bootstrap_accuracies, bootstrap_errors = classifier.bootstrap(num_iterations,
train_percentage, test_percentage)

    # Imprimir resultados por grupo
    for i in range(len(bootstrap_accuracies)):
        efficiency = bootstrap_accuracies[i]
        error = bootstrap_errors[i]
        print(f"\nResultados para el Grupo {i + 1}:")
        print(f"Eficiencia de Bootstrap: {efficiency * 100:.2f}%")
        print(f"Error de Bootstrap: {error * 100:.2f}%")

    # Calcular y imprimir resultados generales
    mean_accuracy = np.mean(bootstrap_accuracies)
    mean_error = np.mean(bootstrap_errors)
    std_accuracy = np.std(bootstrap_accuracies)
    std_error = np.std(bootstrap_errors)

    print(f"\nResultados Generales de Bootstrap:")
    print(f"Eficiencia de Bootstrap: {mean_accuracy * 100:.2f}%")
    print(f"Error de Bootstrap: {mean_error * 100:.2f}%")
    print(f"Desviación Estandar de la Eficiencia: {std_accuracy * 100:.2f}%")
    print(f"Desviación Estandar del Error: {std_error * 100:.2f}%")
```