# INSTITUTO POLITECNICO NACIONAL

Escuela Superior de Cómputo

## PLab4.- Métodos de validación

Unidad de aprendizaje: Fundamentos de Inteligencia Artificial


Alumno:

Flores Lara Alberto

Profesor:
Catalán Salgado Edgar armando


Grupo:
4BV1

El código proporcionado es un programa que carga datos desde un archivo de texto plano, permite seleccionar columnas para realizar predicciones y ejecuta dos tipos de clasificadores: KNN y clasificador de distancia mínima. Además, se ofrece la opción de elegir entre tres métodos de validación: entrenamiento y prueba, validación cruzada K-fold y bootstrap. Estos métodos se utilizan para calcular la precisión y el error de los modelos.

## Funciones Utilizadas:

### Función para calcular la distancia euclidiana

distancia_euclidiana(point1, point2): Calcula la distancia euclidiana entre dos puntos en un espacio n-dimensional.

```
def distancia_euclidiana(point1, point2):
    distance = 0.0
    for i in range(len(point1)):
        distance += (point1[i] - point2[i]) ** 2
    return math.sqrt(distance)
```

### Función para calcular la distancia de Manhattan

distancia_manhattan(point1, point2): Calcula la distancia de Manhattan entre dos puntos en un espacio n-dimensional.

```
def distancia_manhattan(point1, point2):
    distance = 0.0
    for i in range(len(point1)):
        distance += (point1[i] - point2[i])
    return distance
```

### Función para cargar datos

cargardatos(archivo, delimitador): Carga los datos desde un archivo de texto plano utilizando pandas.

```
def cargardatos(archivo, delimitador):
    data=pd.read_csv(archivo, delimiter=delimitador)
    return data
```

### Método principal: main()

1. Solicita al usuario el nombre del archivo y el delimitador para cargar los datos.
2. Muestra información sobre el DataFrame cargado.
3. Permite al usuario seleccionar la columna a predecir y los límites para generar el vector de atributos.
4. Pregunta al usuario qué tipo de clasificador desea utilizar (KNN o distancia mínima).
5. Invoca la función correspondiente según la selección del usuario (clas_knn() o class_min()).

```
def main():
```

```
    archivo=input("Escriba el nombre del archivo de donde obtendremos la
informacion: ")
    delimitador=input("Seleccione cual es el signo delimitador del archivo:
")
    datos=cargardatos(archivo,delimitador)
    num_filas, num_columnas = datos.shape
    print(f"El DataFrame tiene {num_filas} patrones y {num_columnas}
atributos.")
    tipos_de_datos = datos.dtypes
    print(tipos_de_datos)
    #Seleccionamos atributos para nuestro vector
    z=(str(input("Escriba el nombre de la columna que quiere predecir: ")))
    x = datos.drop(z, axis=1).values
    y = np.array(datos[z])
    limite_inferior_1 = int(input(f"Seleccione el limite inferior (Valores
entre 0 y {num_columnas-2}) para generar el vector de attributos: "))
    limite_superior_1 = int(input(f"Ahora el limite superior (Valores entre
{limite_inferior_1} y {num_columnas-2}): "))
    matriz_patrones = x[:, limite_inferior_1:limite_superior_1+1]
    opc1=0
    while(opc1!=1 and opc1!=2):
        opc1=int(input("Ingrese el tipo de clasificador que desee usar:\n
1.Clasificador Knn 2.Clasificador distancia minima\n"))
        if(opc1==1):
            clas_knn(matriz_patrones,y)
        elif(opc1==2):
            class_min(matriz_patrones,y)
        else:
            print("Seleccione una opcion correcta")
```

**Función clas_knn(x, y)**

Clasificador KNN: crea una instancia del clasificador KNN y permite al usuario elegir el tipo de validación que desea utilizar (train-test, K-fold, o bootstrap). En función de la elección del usuario, realiza el entrenamiento y evaluación utilizando el método seleccionado.

```
clas_knn(x,y):
    # Clasificador KNN
    class ClasificadorKNN:
        def __init__(self, n_neighbors=1):
            self.n_neighbors = n_neighbors

        def fit(self, X, y):
            self.X_train = X
            self.y_train = y
```

```
        def predict(self, X):
            met_distancia=int(input("Ingrese el tipo de distancia que desee
usar:\n 1.Euclidiana 2.Manhattan\n"))
            y_pred = []
            for sample in X:
                distances = []
                for i, train_sample in enumerate(self.X_train):
                    if(met_distancia==1):
                        distance = distancia_euclidiana(sample,
train_sample)
                    else:
                        distance = distancia_manhattan(sample, train_sample)
                    distances.append((distance, self.y_train[i]))

                distances.sort(key=lambda x: x[0])
                neighbors = distances[:self.n_neighbors]
                neighbor_labels = [neighbor[1] for neighbor in neighbors]
                prediction = max(set(neighbor_labels),
key=neighbor_labels.count)
                y_pred.append(prediction)
            return y_pred

    # Pedir al usuario el número de vecinos a considerar
    n_neighbors_input = int(input("Introduce el número de vecinos a
considerar: "))

    # Crear una instancia del clasificador KNN con el número de vecinos
especificado
    knn_classifier = ClasificadorKNN(n_neighbors=n_neighbors_input)
    opc=0
    while(opc!=1 and opc!=2 and opc!=3):
        opc = int(input("Elija el tipo de validacion que desee usar: 1.Train
and test 2.K-fold cross-validation 3.Bootstrap\n"))
        if opc == 1:
            x_train, x_test, y_train, y_test = train_test(x, y)
            knn_classifier.fit(x_train, y_train)
            y_pred = knn_classifier.predict(x_test)
            accuracy = np.mean(y_pred == y_test) *100
            error = 100 - accuracy
            print(f"Porcentaje de precisión de la clasificación de distancia
mínima: {accuracy:.2f}%")
            print(f"Precisión de error en la clasificación de distancia
mínima: {error:.2f}%")

        elif opc==2:
```

```
            k = int(input("Ingrese la cantidad de grupos (K) para la
validación cruzada: "))
            kf = KFold(n_splits=k)

            accuracy_scores = []
            error_scores = []

            i=0 #Indice de experimentos

            for train_index, test_index in kf.split(x):
                x_train, x_test = x[train_index], x[test_index]
                y_train, y_test = y[train_index], y[test_index]
                knn_classifier.fit(x_train, y_train)
                y_pred = knn_classifier.predict(x_test)
                accuracy = np.mean(y_pred == y_test) * 100
                error = np.mean(y_pred != y_test) * 100

                accuracy_scores.append(accuracy)
                error_scores.append(error)

                i=i+1

                print(f"Porcentaje de precisión para el experimento {i}:
{accuracy:.2f}%")
                print(f"Porcentaje de error para el experimento {i}:
{error:.2f}%")

            avg_accuracy = mean(accuracy_scores)
            avg_error = mean(error_scores)
            std_accuracy = stdev(accuracy_scores)
            std_error = stdev(error_scores)

            print("\nResultados generales:")
            print(f"Porcentaje de precisión promedio: {avg_accuracy:.2f}% ±
{std_accuracy:.2f}")
            print(f"Porcentaje de error promedio: {avg_error:.2f}% ±
{std_error:.2f}")


        elif opc==3:
            k = int(input("Ingrese la cantidad de experimentos (K) para el
bootstrap: "))
            muestras_entrenamiento = int(input("Ingrese la cantidad de
muestras en el conjunto de entrenamiento: "))
```

```
            muestras_prueba = int(input("Ingrese la cantidad de muestras en
el conjunto de prueba: "))

            accuracy_scores = []
            error_scores = []
            class_accuracy_scores = {}
            class_error_scores = {}

            for i in range(k):
                # Muestreo bootstrap para crear conjuntos de entrenamiento y
prueba
                train_indices = np.random.choice(range(len(x)),
size=muestras_entrenamiento, replace=True)
                test_indices = np.random.choice(range(len(x)),
size=muestras_prueba, replace=True)

                x_train, x_test = x[train_indices], x[test_indices]
                y_train, y_test = y[train_indices], y[test_indices]

                knn_classifier.fit(x_train, y_train)
                y_pred = knn_classifier.predict(x_test)

                # Calcular la precisión y el error para cada grupo
                accuracy = np.mean(y_pred == y_test) * 100
                error = np.mean(y_pred != y_test) * 100

                accuracy_scores.append(accuracy)
                error_scores.append(error)

                # Calcular precisión y error por clase
                unique_classes = np.unique(y_test)
                for cls in unique_classes:
                    cls_indices = np.where(y_test == cls)[0]
                    cls_pred = np.array(y_pred)[cls_indices]  # Filtrar
predicciones para la clase actual
                    cls_accuracy = np.mean(cls_pred == cls) * 100
                    cls_error = 100 - cls_accuracy

                    if cls not in class_accuracy_scores:
                        class_accuracy_scores[cls] = []
                    if cls not in class_error_scores:
                        class_error_scores[cls] = []

                    class_accuracy_scores[cls].append(cls_accuracy)
                    class_error_scores[cls].append(cls_error)
```

```
                print(f"Porcentaje de precisión para el experimento {i+1}:
{accuracy:.2f}%")
                print(f"Porcentaje de error para el experimento {i+1}:
{error:.2f}%")

            avg_accuracy = mean(accuracy_scores)
            avg_error = mean(error_scores)
            std_accuracy = stdev(accuracy_scores)
            std_error = stdev(error_scores)

            print("\nResultados generales:")
            print(f"Porcentaje de precisión promedio: {avg_accuracy:.2f}% ±
{std_accuracy:.2f}")
            print(f"Porcentaje de error promedio: {avg_error:.2f}% ±
{std_error:.2f}")

            # Calcular el promedio y la desviación estándar de precisión y
error por clase
            for cls, cls_acc_scores in class_accuracy_scores.items():
                cls_avg_acc = mean(cls_acc_scores)
                cls_std_acc = stdev(cls_acc_scores)
                cls_avg_err = mean(class_error_scores[cls])
                cls_std_err = stdev(class_error_scores[cls])

                print(f"\nResultados para la clase {cls}:")
                print(f"Porcentaje de precisión promedio: {cls_avg_acc:.2f}%
± {cls_std_acc:.2f}")
                print(f"Porcentaje de error promedio: {cls_avg_err:.2f}% ±
{cls_std_err:.2f}")
        else:
                print("Seleccione una opcion correcta")
```

Función class_min(x, y)
Clasificador de distancia mínima: Similar al KNN, permite al usuario elegir el tipo de validación que desea utilizar y realiza el entrenamiento y la evaluación según la elección.

```
def class_min(x, y):
    class ClasificadorDistanciaMinima:
        def fit(self, X, y):
            self.X_train = X
            self.y_train = y

        def predict(self, X):
```

```
            met_distancia = int(input("Ingrese el tipo de distancia que
desee usar:\n 1.Euclidiana 2.Manhattan\n"))
            y_pred = []
            for sample in X:
                min_distance = float('inf')
                nearest_label = None
                for i, train_sample in enumerate(self.X_train):
                    if met_distancia == 1:
                        distance = distancia_euclidiana(sample,
train_sample)
                    else:
                        distance = distancia_manhattan(sample, train_sample)
                    if distance < min_distance:
                        min_distance = distance
                        nearest_label = self.y_train[i]
                y_pred.append(nearest_label)
            return y_pred

    min_distance = ClasificadorDistanciaMinima()

    opc=0
    while(opc!=1 and opc!=2 and opc!=3):
        opc = int(input("Elija el tipo de validacion que desee usar: 1.Train
and test 2.K-fold cross-validation 3.Bootstrap\n"))
        if opc == 1:
            x_train, x_test, y_train, y_test = train_test(x, y)
            min_distance.fit(x_train, y_train)
            y_pred = min_distance.predict(x_test)
            accuracy = np.mean(y_pred == y_test) *100
            error = 100 - accuracy
            print(f"Porcentaje de precisión de la clasificación de distancia
mínima: {accuracy:.2f}%")
            print(f"Precisión de error en la clasificación de distancia
mínima: {error:.2f}%")

        elif opc==2:
            k = int(input("Ingrese la cantidad de grupos (K) para la
validación cruzada: "))
            kf = KFold(n_splits=k)

            accuracy_scores = []
            error_scores = []

            i = 0  #Indice de experimentos
```

```
            for train_index, test_index in kf.split(x):
                x_train, x_test = x[train_index], x[test_index]
                y_train, y_test = y[train_index], y[test_index]
                min_distance.fit(x_train, y_train)
                y_pred = min_distance.predict(x_test)
                accuracy = np.mean(y_pred == y_test) * 100
                error = np.mean(y_pred != y_test) * 100

                accuracy_scores.append(accuracy)
                error_scores.append(error)

                i=i+1
                print(f"Porcentaje de precisión para el experimento {i}:
{accuracy:.2f}%")
                print(f"Porcentaje de error para el experimento {i}:
{error:.2f}%")

            avg_accuracy = mean(accuracy_scores)
            avg_error = mean(error_scores)
            std_accuracy = stdev(accuracy_scores)
            std_error = stdev(error_scores)

            print("\nResultados generales:")
            print(f"Porcentaje de precisión promedio: {avg_accuracy:.2f}% ±
{std_accuracy:.2f}")
            print(f"Porcentaje de error promedio: {avg_error:.2f}% ±
{std_error:.2f}")


        elif opc==3:
            k = int(input("Ingrese la cantidad de experimentos (K) para el
bootstrap: "))
            muestras_entrenamiento = int(input("Ingrese la cantidad de
muestras en el conjunto de entrenamiento: "))
            muestras_prueba = int(input("Ingrese la cantidad de muestras en
el conjunto de prueba: "))

            accuracy_scores = []
            error_scores = []
            class_accuracy_scores = {}
            class_error_scores = {}

            for i in range(k):
                # Muestreo bootstrap para crear conjuntos de entrenamiento y
prueba
```

```
                train_indices = np.random.choice(range(len(x)),
size=muestras_entrenamiento, replace=True)
                test_indices = np.random.choice(range(len(x)),
size=muestras_prueba, replace=True)

                x_train, x_test = x[train_indices], x[test_indices]
                y_train, y_test = y[train_indices], y[test_indices]

                min_distance.fit(x_train, y_train)
                y_pred = min_distance.predict(x_test)

                # Calcular la precisión y el error para cada grupo
                accuracy = np.mean(y_pred == y_test) * 100
                error = np.mean(y_pred != y_test) * 100

                accuracy_scores.append(accuracy)
                error_scores.append(error)

                # Calcular precisión y error por clase
                unique_classes = np.unique(y_test)
                for cls in unique_classes:
                    cls_indices = np.where(y_test == cls)[0]
                    cls_pred = np.array(y_pred)[cls_indices]  # Filtrar
predicciones para la clase actual
                    cls_accuracy = np.mean(cls_pred == cls) * 100
                    cls_error = 100 - cls_accuracy

                    if cls not in class_accuracy_scores:
                        class_accuracy_scores[cls] = []
                    if cls not in class_error_scores:
                        class_error_scores[cls] = []

                    class_accuracy_scores[cls].append(cls_accuracy)
                    class_error_scores[cls].append(cls_error)

                print(f"Porcentaje de precisión para el experimento {i+1}:
{accuracy:.2f}%")
                print(f"Porcentaje de error para el experimento {i+1}:
{error:.2f}%")

            avg_accuracy = mean(accuracy_scores)
            avg_error = mean(error_scores)
            std_accuracy = stdev(accuracy_scores)
            std_error = stdev(error_scores)
```

```
            print("\nResultados generales:")
            print(f"Porcentaje de precisión promedio: {avg_accuracy:.2f}% ±
{std_accuracy:.2f}")
            print(f"Porcentaje de error promedio: {avg_error:.2f}% ±
{std_error:.2f}")

            # Calcular el promedio y la desviación estándar de precisión y
error por clase
            for cls, cls_acc_scores in class_accuracy_scores.items():
                cls_avg_acc = mean(cls_acc_scores)
                cls_std_acc = stdev(cls_acc_scores)
                cls_avg_err = mean(class_error_scores[cls])
                cls_std_err = stdev(class_error_scores[cls])

                print(f"\nResultados para la clase {cls}:")
                print(f"Porcentaje de precisión promedio: {cls_avg_acc:.2f}%
± {cls_std_acc:.2f}")
                print(f"Porcentaje de error promedio: {cls_avg_err:.2f}% ±
{cls_std_err:.2f}")
        else:
                print("Seleccione una opcion correcta")
```

Función train_test(x, y)
Divide los datos en conjuntos de entrenamiento y prueba, permitiendo al usuario especificar el tamaño del conjunto de entrenamiento.

```
def train_test(x,y):
    porcentaje_entrenamiento = float(input("Ingrese el porcentaje de
muestras para el conjunto de entrenamiento (0-1): "))
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=1 -
porcentaje_entrenamiento)
    return x_train, x_test, y_train, y_test
```

Las diferencias entre las aplicaciones de los métodos de validación realmente no varían en el caso de utilizar el clasificador de distancia mínima o el de vecinos knn. Por lo que describiremos de forma detallada que es lo que realiza cada método de validación dentro del código:

**Entrenamiento y Prueba:**
En este método, se divide el conjunto de datos en conjuntos de entrenamiento y prueba. A continuación, se entrena el modelo de clasificación seleccionado (ya sea KNN o clasificador de distancia mínima) con el conjunto de entrenamiento y se evalúa su rendimiento con el conjunto de prueba.

El flujo del código para este método es el siguiente:

1. Se pregunta al usuario qué porcentaje de muestras se utilizará para el conjunto de entrenamiento.
2. Se emplea train_test_split de scikit-learn para dividir los datos en conjuntos de entrenamiento y prueba.
3. El modelo se entrena con los datos de entrenamiento (fit).
4. Se realizan predicciones con el conjunto de prueba (predict).
5. Se calculan la precisión y el error comparando las predicciones con las etiquetas reales.

```
x_train, x_test, y_train, y_test = train_test(x, y)
            min_distance.fit(x_train, y_train)
            y_pred = min_distance.predict(x_test)
            accuracy = np.mean(y_pred == y_test) *100
            error = 100 - accuracy
            print(f"Porcentaje de precisión de la clasificación de distancia
mínima: {accuracy:.2f}%")
            print(f"Precisión de error en la clasificación de distancia
mínima: {error:.2f}%")
```

**K-Fold Cross-Validation:**
En este método, se divide el conjunto de datos en K grupos (folds) para realizar validación cruzada. El modelo se entrena y evalúa K veces, cada vez utilizando un grupo diferente como conjunto de prueba y el resto como conjunto de entrenamiento.

El flujo del código para este método es el siguiente:

1. Se utiliza KFold de scikit-learn para dividir los datos en K grupos.
2. Se realiza un bucle sobre los grupos generados por KFold.
3. En cada iteración, se entrena el modelo con el conjunto de entrenamiento actual y se evalúa con el conjunto de prueba.
4. Se calculan la precisión y el error para cada iteración y se imprimen.

```
k = int(input("Ingrese la cantidad de grupos (K) para la validación cruzada:
"))
            kf = KFold(n_splits=k)

            accuracy_scores = []
            error_scores = []

            i = 0  #Indice de experimentos

            for train_index, test_index in kf.split(x):
                x_train, x_test = x[train_index], x[test_index]
                y_train, y_test = y[train_index], y[test_index]
```

```
                min_distance.fit(x_train, y_train)
                y_pred = min_distance.predict(x_test)
                accuracy = np.mean(y_pred == y_test) * 100
                error = np.mean(y_pred != y_test) * 100

                accuracy_scores.append(accuracy)
                error_scores.append(error)

                i=i+1
                print(f"Porcentaje de precisión para el experimento {i}:
{accuracy:.2f}%")
                print(f"Porcentaje de error para el experimento {i}:
{error:.2f}%")

            avg_accuracy = mean(accuracy_scores)
            avg_error = mean(error_scores)
            std_accuracy = stdev(accuracy_scores)
            std_error = stdev(error_scores)

            print("\nResultados generales:")
            print(f"Porcentaje de precisión promedio: {avg_accuracy:.2f}% ±
{std_accuracy:.2f}")
            print(f"Porcentaje de error promedio: {avg_error:.2f}% ±
{std_error:.2f}")
```

**Bootstrap:**
Este método utiliza el muestreo bootstrap para generar múltiples conjuntos de entrenamiento y prueba. Se entrena y evalúa el modelo K veces con estos conjuntos.

El flujo del código para este método es el siguiente:

1. Se solicita al usuario la cantidad de experimentos (K), la cantidad de muestras en el conjunto de entrenamiento y en el conjunto de prueba.
2. Se ejecuta un bucle para realizar K experimentos.
3. En cada experimento, se crea un conjunto de entrenamiento y prueba utilizando muestreo bootstrap.
4. Se entrena el modelo con el conjunto de entrenamiento y se evalúa con el conjunto de prueba.
5. Se calculan la precisión y el error para cada experimento y para cada clase individual.
6. Se imprimen los resultados detallados de cada experimento y los resultados generales.

```
# Se solicita al usuario la cantidad de experimentos, la cantidad de
muestras para entrenamiento y prueba
```

```
k = int(input("Ingrese la cantidad de experimentos (K) para el bootstrap:
"))
muestras_entrenamiento = int(input("Ingrese la cantidad de muestras en el
conjunto de entrenamiento: "))
muestras_prueba = int(input("Ingrese la cantidad de muestras en el conjunto
de prueba: "))

# Listas para almacenar resultados generales y por clase
accuracy_scores = []
error_scores = []
class_accuracy_scores = {}
class_error_scores = {}

# Bucle para realizar K experimentos de bootstrap
for i in range(k):
    # Muestreo bootstrap para crear conjuntos de entrenamiento y prueba
    train_indices = np.random.choice(range(len(x)),
size=muestras_entrenamiento, replace=True)
    test_indices = np.random.choice(range(len(x)), size=muestras_prueba,
replace=True)

    x_train, x_test = x[train_indices], x[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    # Entrenar el clasificador con el conjunto de entrenamiento y predecir
el conjunto de prueba
    min_distance.fit(x_train, y_train)
    y_pred = min_distance.predict(x_test)

    # Calcular precisión y error para el experimento actual
    accuracy = np.mean(y_pred == y_test) * 100
    error = np.mean(y_pred != y_test) * 100

    # Almacenar precisión y error en las listas correspondientes
    accuracy_scores.append(accuracy)
    error_scores.append(error)

    # Calcular precisión y error por clase
    unique_classes = np.unique(y_test)
    for cls in unique_classes:
        cls_indices = np.where(y_test == cls)[0]
        cls_pred = np.array(y_pred)[cls_indices]  # Filtrar predicciones
para la clase actual
        cls_accuracy = np.mean(cls_pred == cls) * 100
        cls_error = 100 - cls_accuracy
```

```
        # Almacenar precisión y error por clase en diccionarios separados
        if cls not in class_accuracy_scores:
            class_accuracy_scores[cls] = []
        if cls not in class_error_scores:
            class_error_scores[cls] = []

        class_accuracy_scores[cls].append(cls_accuracy)
        class_error_scores[cls].append(cls_error)

    # Imprimir resultados del experimento actual
    print(f"Porcentaje de precisión para el experimento {i+1}:
{accuracy:.2f}%")
    print(f"Porcentaje de error para el experimento {i+1}: {error:.2f}%")

# Calcular el promedio y la desviación estándar de precisión y error para
todos los experimentos
avg_accuracy = mean(accuracy_scores)
avg_error = mean(error_scores)
std_accuracy = stdev(accuracy_scores)
std_error = stdev(error_scores)

# Imprimir resultados generales
print("\nResultados generales:")
print(f"Porcentaje de precisión promedio: {avg_accuracy:.2f}% ±
{std_accuracy:.2f}")
print(f"Porcentaje de error promedio: {avg_error:.2f}% ± {std_error:.2f}")

# Calcular el promedio y la desviación estándar de precisión y error por
clase
for cls, cls_acc_scores in class_accuracy_scores.items():
    cls_avg_acc = mean(cls_acc_scores)
    cls_std_acc = stdev(cls_acc_scores)
    cls_avg_err = mean(class_error_scores[cls])
    cls_std_err = stdev(class_error_scores[cls])

    # Imprimir resultados por clase
    print(f"\nResultados para la clase {cls}:")
    print(f"Porcentaje de precisión promedio: {cls_avg_acc:.2f}% ±
{cls_std_acc:.2f}")
    print(f"Porcentaje de error promedio: {cls_avg_err:.2f}% ±
{cls_std_err:.2f}")
```