Clasificación con regresión logística

Dr. Esteban Castillo Juarez

El objetivo principal del laboratorio es el de crear el segundo algoritmo de aprendizaje automático en Python: **Regresión logística**, que es una herramienta muy efectiva para clasificar elementos en función de valores numéricos o nominales.

Primero, se expone el enfoque general para desarrollar el clasificador. Luego, se examina la función sigmoidea, clave para realizar predicciones. A continuación, se detalla cómo implementar un proceso de optimización para identificar los coeficientes o pesos más adecuados para las características seleccionadas. Finalmente, se procede a entrenar y probar el modelo de regresión utilizando un conjunto de datos genérico

Elementos principales del código:

1. Razonamiento detrás de la regresión logística
2. Creación del conjunto de datos
3. Función sigmoidea
4. Entrenamiento de la regresión logística
   - Gradiente Estocástico (proceso de optimización)
   - Cálculo del gradiente estocástico
   - Función `random.uniform()`
   - Función de clasificación
5. Prueba de la regresión logística
   - Definición de Accuracy (Exactitud)
   - Implementación completa
6. Implementación en Scikit-learn
   - Parámetros básicos de la regresión logística
   - Uso de la regresión logística
   - Matriz de confusión
   - Definición de precisión y recuerdo
   - Clase positiva y negativa en métricas de clasificación
   - Precisión vs recuerdo

For more information see:

1. Regresión logística

2. Aprendizaje de maquina
3. Creacion de un notebook

1 Razonamiento detrás de la regresión logística

Cualquier proceso que intente encontrar relaciones entre variables se llama regresión.

La regresión logística es un método de análisis estadístico utilizado para predecir un resultado en función de observaciones previas de un conjunto de datos. Este es el segundo algoritmo de aprendizaje maquina a revisar.

En la regresión logística, se toman las características a analizar de una muestra y se multiplican por un peso optimizado (basado en la teoría de Ascenso de Gradiente Estocástico) para luego sumarlas. Este resultado se introducirá en una función que generará un número entre `0` y `1`. Cualquier valor superior a `0.5` se clasificará como `1`, y cualquier valor inferior a `0.5` se clasificará como `0`. También se puede pensar en la regresión logística como una estimación de probabilidad.

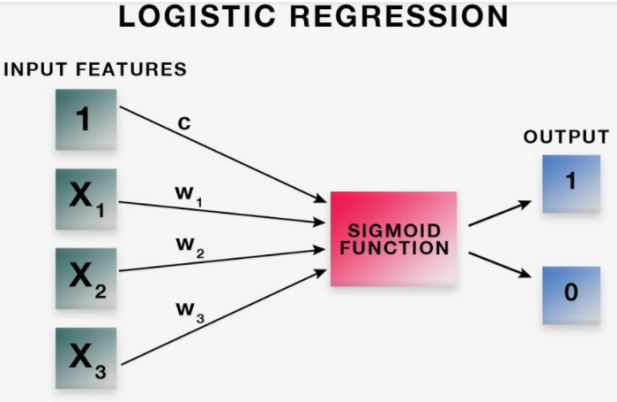
A diferencia de la regresión lineal, que se usa para predecir valores numéricos continuos, la regresión logística se utiliza para clasificar datos en categorías. Mientras que la regresión lineal se ajusta una línea para predecir un número exacto, la regresión logística usa una fórmula que da como resultado un valor entre `0` y `1` que se interpreta como la probabilidad de que un dato pertenezca a una categoría específica. Por lo tanto, en lugar de predecir un número, decide a qué grupo o clase pertenece una muestra.

En esta sección, implementaremos la regresión logística en Python. Primero, exploraremos el enfoque general en pseudocódigo y luego lo llevaremos a cabo en código.

```
ENTRENAMIENTO: usa el ascenso de gradiente estocástico para encontrar los mejores parámetros/pesos
Comienza con todos los pesos establecidos en 1.0 (un peso para cada característica)

Repite un número definido de iteraciones (k):
    Para muestra en el subconjunto de entrenamiento:
        Calcula el gradiente estocástico para esa muestra
        Actualiza el vector de pesos sumando alpha * gradiente
    Devuelve el vector de pesos actualizado

PRUEBA: Basado en los parámetros/pesos óptimos obtenidos durante el entrenamiento
Para cada muestra en el subconjunto de prueba:
    Multiplica cada característica de la muestra por los pesos optimizados
    Suma los valores obtenidos
    Aplica la función sigmoidea al resultado
    Si el valor de la sigmoidea es mayor que 0.5, clasifica el dato como 1
    En caso contrario, clasifica el dato como 0
```



2 Creación del conjuntos de datos

Para implementar la regresión logística, crearemos un conjunto de datos personalizado que constará de elementos de entrenamiento y prueba (regressionTraining.txt y regressionTest.txt).

Las muestras de entrenamiento y prueba tendrán cinco características numéricas y dos posibles etiquetas (`0/1`). En el entrenamiento, estas etiquetas se utilizarán para detectar/predictir una clase, mientras que en la prueba se determinará la etiqueta real de una muestra (ground-truth).

A continuación se muestran algunos ejemplos del conjunto de datos:

```
-0.02854153291856363,0.044758959136042296,-0.37135752335129735,0.04326731993794278,0.34304940766286374,0.0
-0.04965006680345901,0.10270686279660618,-0.2932616660984198,0.31284692613266146,0.14953861333713556,0.0
 0.1419322223158102,-0.28755686113903595,0.10227489508778553,-0.0005332650419625944,-0.12984433874755283,1.0
 0.25915697066631543,-0.27883078034220776,-0.02519712277187007,0.10216117879936266,0.3443395713114573,0.0
-0.35638949432303635,-0.05479798101845598,-0.00866667091394243,0.01818992581975883,-0.16911561396605582,1.0
```

[1]:
```
#Import a mathematical library
import numpy as np
#Import a file manipulation library
import codecs
#Google drive access library
from google.colab import drive

"""
We set up the Google Drive instance associated with
our email account within the institution
"""
drive.mount('/content/drive')

"""
We set the path where the file is located in Google Colab.
It is desirable to change it to use it with another account.
"""
base_path = "/content/drive/My Drive/ColabNotebooks/TC3006C-2024/algorithms/Notebook3/"



"""
Create a random dataset with 100 training samples and 30 test samples
The dataset is composed of five features and a classification label (1 or 0 - binary classification)
"""
print("Create training samples")

feature1=list(np.random.normal(0, 0.2, 100))
feature2=list(np.random.normal(0, 0.2, 100))
feature3=list(np.random.normal(0, 0.2, 100))
```

```
feature4=list(np.random.normal(0, 0.2, 100))
feature5=list(np.random.normal(0, 0.2, 100))
label=list(np.random.choice([1.0, 0.0], size=100, p=[0.5, 0.5]))

print("Save training samples into a file")
with codecs.open("regressionTraining.txt","w","UTF-8") as file:
    for f1,f2,f3,f4,f5,l in zip(feature1,feature2,feature3,feature4,feature5,label):
        file.write(str(f1)+","+str(f2)+","+str(f3)+","+str(f4)+","+str(f5)+","+str(l)+"\n")


print("Create test samples")

feature1=list(np.random.normal(0, 0.2, 50))
feature2=list(np.random.normal(0, 0.2, 50))
feature3=list(np.random.normal(0, 0.2, 50))
feature4=list(np.random.normal(0, 0.2, 50))
feature5=list(np.random.normal(0, 0.2, 50))
label=list(np.random.choice([1.0, 0.0], size=50, p=[0.5, 0.5]))

print("Save test samples into a file")
with codecs.open("regressionTest.txt","w","UTF-8") as file:
    for f1,f2,f3,f4,f5,l in zip(feature1,feature2,feature3,feature4,feature5,label):
        file.write(str(f1)+","+str(f2)+","+str(f3)+","+str(f4)+","+str(f5)+","+str(l)+"\n")
```

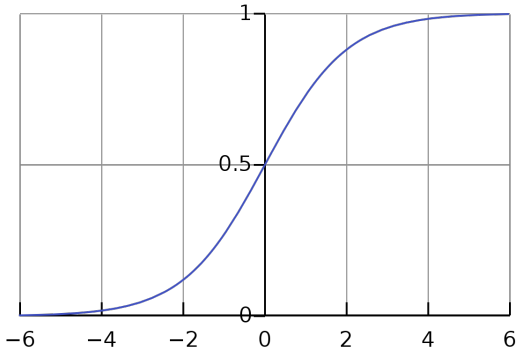
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
Create training samples
Save training samples into a file
Create test samples
Save test samples into a file
```

3 Función sigmoidea

Es deseable tener una ecuación que, al ingresar todas las características de una muestra, permita predecir la clase a la que pertenece. En el caso de una clasificación binaria, una de las mejores opciones es la **función sigmoidea**, la cual se representa mediante la siguiente ecuación:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Para valores crecientes de `z`, la sigmoidea se acercará a `1`, y para valores decrecientes de `z`, la sigmoidea se acercará a `0`. De manera gráfica, la sigmoidea se parece a una función escalonada.



La implementación de una función sigmoidea en Python es la siguiente:

[2]:
```python
#Import a mathematical library
import math
"""
Define a Python function that calculates the sigmoid function
Input: float
Output: float (value between 0 and 1)
"""
def sigmoid(z):
  return 1 / (1 + math.exp(-z))
```

4 Entrenamiento de la regresión logística

La entrada a la función sigmoidea será `z`, donde esta viene dada por lo siguiente:

$$z = w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n$$

De la ecuación anterior se puede ver que se tienen dos vectores de números que se multiplican elemento por elemento y se suman para obtener un solo número. El vector `x` representa los datos de entrada, y se busca encontrar los mejores coeficientes `w` para que el clasificador sea lo más efectivo posible. Para lograr esto, se considerará un enfoque de optimización estocástico (aleatorio).

4.1 Gradiente Estocástico (proceso de optimización)

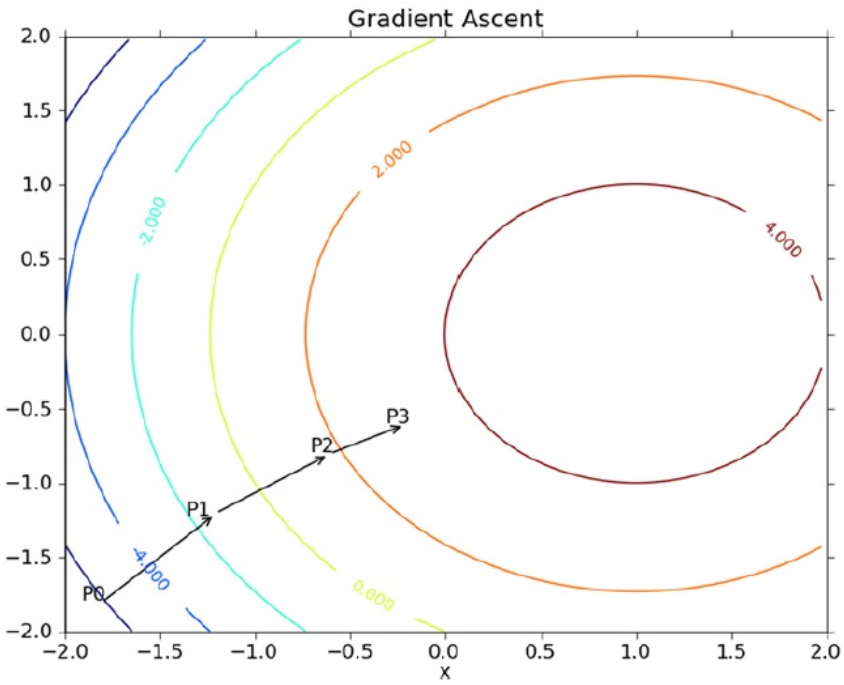
El gradiente estocástico es una técnica para encontrar la mejor solución a un problema. La idea es ajustar los valores de manera que se mejore continuamente el rendimiento de un modelo.

En este método, se ajustan los valores poco a poco en la dirección que promete una mejora rápida. Se repite este ajuste hasta que se cumple una meta, como un número específico de pasos o cuando el modelo ya está funcionando de manera aceptable.

Normalmente, se utiliza todo el conjunto de datos para ajustar los valores en cada paso. Esto funciona bien si hay pocas muestras, pero se vuelve muy costoso cuando hay muchas muestras y características asociadas. Por eso, se prefiere actualizar los valores usando solo una muestra a la vez, lo cual se llama ascenso de gradiente estocástico. Este método se conoce como aprendizaje en línea porque permite ajustar el clasificador poco a poco a medida que llegan nuevos datos, en lugar de hacerlo todo de una vez, como en el procesamiento por lotes.

4.2 Cálculo del gradiente estocástico

El algoritmo de gradiente estocástico avanza en la dirección que indica cómo mejorar en cada ubicación. Comenzando en la posición `P0`, se calcula cómo debería moverse para mejorar, y luego la función se desplaza a la siguiente ubicación, `P1`. En `P1`, se calcula nuevamente la dirección de mejora y la función se mueve a `P2`. Este proceso se repite hasta que se cumple una condición de parada. Este método siempre garantiza que se esté avanzando en la mejor dirección para lograr la mejora.



A continuación se muestra una implementación del gradiente estocástico en Python:

[3]:
```
import random

"""
Define a Python function that obtain the gradient
Input: list - sampleList
Input: list - weights
Output: float (gradient)

Note: For this function we use the sigmoid function for the first time.

"""

def gradient(sampleList, weights):
    sumElements=0.0
    for x,y in zip(sampleList,weights):
        #Multiply weights and feature elements and add all values
        sumElements=sumElements+(x*y)
    #Return the sigmoid of previous addition.
    return sigmoid(sumElements)


"""
Define a Python function that given a training list, labels, feature number and iterations, calculates the␣
↪stochastic
gradient ascent

Input: list - trainingLists
Input: list - trainingLabels
Input: int - featureNumber
Input: int - iterarions
Output: list (optimal weights)
"""

def stochasticGradientAscent(trainingLists, trainingLabels, featureNumber ,iterarions=150):
    #Get the number of training samples
    sampleNumber=len(trainingLists)
```

```
    #Create a list of N fatures (featureNumber) for saving optimal weights (1.0 as initial value)
    weights=[1.0] * featureNumber
    #Iterate a fixed number of times for getting optimal weights
    for x in range(iterarions):
        #Get the index number of training samples
        sampleIndex = list(range(sampleNumber))
        #For each training sample do the following
        for y in range(sampleNumber):

            """
            Alpha is the learning rate and controls how much the coefficients (and therefore the model)
            changes or learns each time it is updated.

            Alpha decreases as the number of iterations increases, but it never reaches 0
            """
            alpha=4/(1.0+x+y)+0.01
            #Randomly obtain an index of one of training samples
            """
            Here, you're randomly selecting each instance to use in updating the weights.
            This will reduce the small periodic variations that can be present if we analyze
            everything sequentially
            """
            randIndex = int(random.uniform(0,len(sampleIndex)))
            #Obtain the gradient from the current training sample and weights
            sampleGradient=gradient(trainingLists[randIndex],weights)
            #Check the error rate
            error=trainingLabels[randIndex]-sampleGradient
            """
            we are calculating the error between the actual class and the predicted class and
            then moving in the direction of that error (CURRENT TRAINING PROCESS)
            """
            temp=[]
            for index in range(featureNumber):
                temp.append(alpha*(error*trainingLists[randIndex][index]))

            for z in range(featureNumber):
                weights[z]= weights[z] + temp[z]

            del(sampleIndex[randIndex])
    return weights
```

4.3 Función `random.uniform()`

En la implementación del gradiente estocástico, la función `random.uniform(a, b)` genera un número decimal aleatorio en el rango `[a, b]`. Aquí se presentan algunos ejemplos para entender su uso:

[4]:
```
import random

#First example
print(random.uniform(20, 60))
#Second example
print(random.uniform(1.5, 1.9))
```

```
28.090573735401115
1.5960536131987966
```

4.4 Función de clasificación

Con la regresión logística, no es necesario hacer mucho para clasificar una instancia. Solo se debe calcular la función sigmoidea del vector de prueba multiplicado por los pesos optimizados anteriormente. Si el resultado de la función sigmoidea es mayor a `0.5`, la clase es `1`; de lo contrario, es `0`.

La implementación de la función de clasificación en Python es la siguiente:

[5]:
```
"""
Create a test function for the logistic regression
Input: list - testList
Input: list - weights
Output: int (0.0/1.0)

"""
def classifyList(testList, weights):
    sumElements=0
    #Multiply all features and optimized weights
    for x,y in zip(testList,weights):
        sumElements=sumElements+(x*y)
    #Obtain the sigmoid output which will tell us the class a test vector belongs
    probability = sigmoid(sumElements)
    if probability  > 0.5:
        return 1.0
    else:
        return 0.0
```

5 Prueba de la regresión logística

Ahora, se procederá a probar el clasificador actual con los datos de entrenamiento y prueba existentes.

5.1 Definición de Accuracy (Exactitud)

La exactitud es una métrica que indica el porcentaje de predicciones correctas en relación con el número total de muestras. Es la medida que usamos habitualmente en modelos de aprendizaje automático.

$$Accuracy = \frac{Number\ of\ Correct\ predictions}{Total\ number\ of\ predictions\ made}$$

Funciona bien si hay un número igual (o casi igual) de muestras pertenecientes a cada clase.

5.2 Implementación completa

La implementación real del proceso de entrenamiento y prueba en Python es la siguiente:

[6]:
```
#Import a file manipulation library
import codecs
training=[]
test=[]
trainingLabels=[]
testLabels=[]
```

```
#Number of repetitions for optimizing the weights (for the training process)
iterarions=100
#Number of features found in the dataset
featureNumber=5

#Load training and test data from dataset files

print("Load training samples")
with codecs.open("regressionTraining.txt","r","UTF-8") as file:
    for line in file:
        elements=(line.rstrip('\n')).split(",")
        training.append([float(elements[0]),
                         float(elements[1]),
                         float(elements[2]),
                         float(elements[3]),
                         float(elements[4])])
        trainingLabels.append(float(elements[5]))

print("Load test samples")
with codecs.open("regressionTest.txt","r","UTF-8") as file:
    for line in file:
        elements=(line.rstrip('\n')).split(",")
        test.append([float(elements[0]),
                         float(elements[1]),
                         float(elements[2]),
                         float(elements[3]),
                         float(elements[4])])
        testLabels.append(float(elements[5]))

#Apply the stochastic gradient ascent for finding the weights that maximize features

print("Apply the stochastic gradient ascent over training samples")

optimalWeights=stochasticGradientAscent(training, trainingLabels, featureNumber, iterarions)

print("Use the obtained weights over test samples for clasifying")

correctPredictions=0
TotalPredictions=0

#Check all test samples for predicting the real value and then check the overall accuracy of the model
for x,y in zip (test, testLabels):
    TotalPredictions+=1
    predicted=classifyList(x,optimalWeights)
    if predicted==y:
        correctPredictions+=1
    print("Predicted: "+str(predicted)+" realValue: "+str(y))

#Calculates model accuracy
print("Model accuracy: "+str(correctPredictions/TotalPredictions)+"%")
```

```
Load training samples
Load test samples
Apply the stochastic gradient ascent over training samples
Use the obtained weights over test samples for clasifying
Predicted: 1.0 realValue: 0.0
Predicted: 1.0 realValue: 0.0
Predicted: 1.0 realValue: 1.0
Predicted: 1.0 realValue: 0.0
Predicted: 0.0 realValue: 0.0
Predicted: 1.0 realValue: 1.0
Predicted: 0.0 realValue: 1.0
```

```
Predicted: 0.0 realValue: 1.0
Predicted: 1.0 realValue: 0.0
Predicted: 0.0 realValue: 0.0
Predicted: 0.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 0.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 0.0 realValue: 1.0
Predicted: 0.0 realValue: 0.0
Predicted: 0.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 0.0 realValue: 1.0
Predicted: 0.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 0.0 realValue: 1.0
Predicted: 0.0 realValue: 0.0
Predicted: 1.0 realValue: 1.0
Predicted: 0.0 realValue: 0.0
Predicted: 1.0 realValue: 1.0
Predicted: 1.0 realValue: 0.0
Predicted: 1.0 realValue: 1.0
Predicted: 1.0 realValue: 0.0
Predicted: 0.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 0.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 0.0 realValue: 1.0
Predicted: 1.0 realValue: 0.0
Predicted: 1.0 realValue: 0.0
Predicted: 1.0 realValue: 1.0
Predicted: 1.0 realValue: 0.0
Predicted: 0.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 0.0 realValue: 1.0
Predicted: 0.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Predicted: 1.0 realValue: 0.0
Predicted: 1.0 realValue: 1.0
Predicted: 1.0 realValue: 1.0
Model accuracy: 0.5%
```

6 Implementación en Scikit-learn

Con la implementación básica de Python hecha, ahora utilizaremos la librería favorita de personas asociadas a la ciencia o analítica de datos para comparar.

6.1 Parámetros básicos de la regresión logística

El algoritmo de regresión logística está disponible en scikit-learn a través de la clase `LogisticRegression`. La implementación de la regresión logística en esta biblioteca es intuitiva y flexible, permitiendo la configuración de parámetros como el método de optimización (por ejemplo, `liblinear`, `newton-cg`, `lbfgs`), la regularización (`C`, que controla la penalización) y el tipo de penalización (como `l1`, `l2`, o ninguna). Esta flexibilidad permite ajustar el modelo para adaptarse a diferentes tipos de datos y necesidades específicas.

De manera especifica, los parametros de la regresión logística son los siguientes:

1. `penalty`: Define cómo se aplica la penalización para evitar que el modelo se ajuste demasiado a los datos.
   (a) `l1`: Elimina algunas características que no son tan importantes, ayudando a simplificar el modelo.
   (b) `l2`: Ajusta todas las características de manera más equilibrada.
   (c) `none`: No se aplica penalización.
2. `C`: Controla cuánto se penaliza el modelo por ajustar demasiado los datos de entrenamiento.
   (a) `Un valor bajo`: Significa que se aplica más penalización. Esto hace que el modelo sea más simple y evite ajustarse demasiado a los datos de entrenamiento, lo que puede ayudar a que funcione mejor con datos nuevos. Sin embargo, si se penaliza demasiado, el modelo puede no aprender bien de los datos.
   (b) `Un valor alto`: Significa que se aplica menos penalización. Esto permite que el modelo ajuste los datos de entrenamiento de manera más precisa. Si se usa un valor muy alto, el modelo puede volverse demasiado complejo y ajustarse demasiado a los datos de entrenamiento, lo que podría hacer que no funcione tan bien con datos nuevos.
3. `solver`: Método utilizado para encontrar la mejor solución.
   (a) `liblinear`: Bueno para conjuntos de datos pequeños o cuando se usa la penalización L1.
   (b) `newton-cg`: Adecuado para problemas grandes y utiliza un enfoque más avanzado.
   (c) `lbfgs`: Un método común y eficiente para conjuntos de datos de tamaño mediano.
   (d) `saga`: Ideal para conjuntos de datos grandes y permite usar penalización L1 y L2.
4. `max_iter`: Número máximo de veces que el algoritmo intentará mejorar el modelo. Si el valor es muy bajo, el modelo puede no ajustarse bien; si es muy alto, el proceso puede tardar mucho.
5. `multi_class`: Maneja problemas con más de dos categorías.
   (a) `auto`: Elige automáticamente el mejor método según los datos.

(b) `ovr (One-vs-Rest)`: Compara cada categoría con todas las demás.

(c) `multinomial`: Trata todas las categorías directamente a la vez.

6. `class_weight`: Ajusta el peso de cada categoría para manejar desequilibrios en los datos.

   (a) `balanced`: Ajusta automáticamente el peso de cada categoría según su frecuencia.

   (b) `None`: No ajusta los pesos y trata todas las categorías por igual.

7. `fit_intercept`: Decide si se debe incluir un término adicional en el modelo para ajustar el resultado. Normalmente se mantiene en True.

8. `tol`: Define la precisión necesaria para que el algoritmo considere que ha terminado de ajustar el modelo. Un valor bajo significa más precisión.

6.2 Uso de la regresión logística

Para utilizar la regresión logística en scikit-learn, primero se debe importar la clase `LogisticRegression` y crear una instancia de la misma. Luego, el modelo se ajusta a los datos de entrenamiento mediante el método `fit()`. Una vez entrenado, el modelo puede hacer predicciones sobre datos nuevos utilizando el método `predict()`. Además, scikit-learn proporciona herramientas para evaluar el rendimiento del modelo, como la exactitud, la precisión o una matriz de confusión (que se detallará más adelante), lo que facilita la validación y ajuste del modelo. La implementación en scikit-learn es eficiente y flexible, adecuada para una amplia variedad de aplicaciones en aprendizaje automático.

A continuación, se muestra cómo usar la regresión logística en scikit-learn con el conjunto de datos utilizado en la implementación manual del algoritmo.

[7]:
```
# Load specific libraries
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import numpy as np
import codecs

# Load training and test data from dataset files
training = []
trainingLabels = []
test = []
testLabels = []

print("Load training samples")
with codecs.open(base_path + "regressionTraining.txt", "r", "UTF-8") as file:
    for line in file:
        elements = (line.rstrip('\n')).split(",")
        training.append([float(elements[0]), float(elements[1]),
                         float(elements[2]), float(elements[3]),
                         float(elements[4])])
        trainingLabels.append(elements[5])

print("Load test samples")
with codecs.open(base_path + "regressionTest.txt", "r", "UTF-8") as file:
    for line in file:
        elements = (line.rstrip('\n')).split(",")

        test.append([float(elements[0]), float(elements[1]),
                         float(elements[2]), float(elements[3]),
                         float(elements[4])])
        testLabels.append(elements[5])
```

```
# Apply the logistic regression approach over test samples using training data on scikit-learn

print("Apply the logistic regression approach over test samples on scikit-learn")

# Create the logistic regression classifier
logistic = LogisticRegression()

# Fit the model to the training data
logistic.fit(training, trainingLabels)

# Make predictions on the test data
predictions = logistic.predict(test)

# Print each prediction with the real label
print("Predictions vs. True Labels:")
for i in range(len(predictions)):
    print("Predicted: " + str(predictions[i]) + " Real Value: " + testLabels[i])

# Evaluate the model's performance
accuracy = accuracy_score(testLabels, predictions)
print("Model accuracy:", accuracy)
```

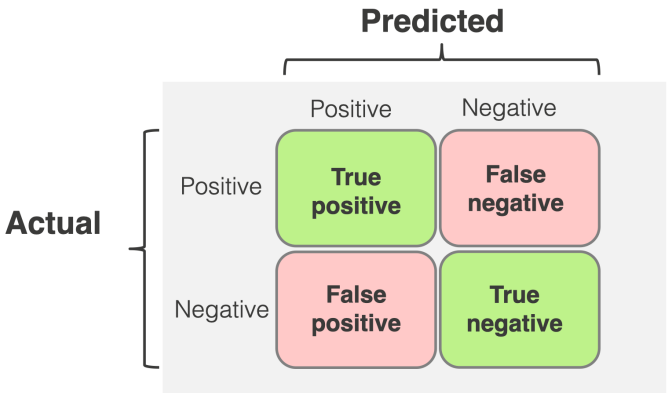
```
Load training samples
Load test samples
Apply the logistic regression approach over test samples on scikit-learn
Predictions vs. True Labels:
Predicted: 0.0 Real Value: 1.0
Predicted: 1.0 Real Value: 1.0
Predicted: 0.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Predicted: 0.0 Real Value: 1.0
Predicted: 0.0 Real Value: 0.0
Predicted: 0.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Predicted: 1.0 Real Value: 1.0
Predicted: 0.0 Real Value: 1.0
Predicted: 0.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Predicted: 0.0 Real Value: 1.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Predicted: 1.0 Real Value: 0.0
Predicted: 0.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Predicted: 0.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Predicted: 1.0 Real Value: 0.0
```

```
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Predicted: 1.0 Real Value: 0.0
Predicted: 0.0 Real Value: 1.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Predicted: 0.0 Real Value: 1.0
Predicted: 1.0 Real Value: 1.0
Predicted: 0.0 Real Value: 0.0
Predicted: 0.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Predicted: 1.0 Real Value: 1.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Predicted: 1.0 Real Value: 1.0
Predicted: 1.0 Real Value: 1.0
Predicted: 1.0 Real Value: 0.0
Predicted: 1.0 Real Value: 1.0
Model accuracy: 0.5
```

6.3 Matriz de confusión

Una matriz de confusión es una herramienta popular en el análisis de rendimiento de modelos de clasificación. Esta matriz es una tabla que permite visualizar el desempeño del modelo comparando las predicciones realizadas con los valores reales.

La matriz se estructura en cuatro categorías principales: verdaderos positivos (`TP`), verdaderos negativos (`TN`), falsos positivos (`FP`) y falsos negativos (`FN`). Estos valores ayudan a entender cuántas veces el modelo predijo correctamente o erróneamente una clase particular, proporcionando así una visión más completa de su precisión y errores.



Una matriz de confusión se compone de los siguientes cuatro elementos básicos:

1. Verdaderos Positivos (`TP`; True Positive): Son los casos en los que el modelo predijo correctamente la clase positiva. Es decir, el modelo dijo que algo pertenecía a una categoría específica, y en realidad sí pertenece a esa categoría.
2. Verdaderos Negativos (`TN`; True Negative): Son los casos en los que el modelo predijo correctamente la clase negativa. Aquí, el modelo indicó que algo no pertenecía a una categoría específica, y efectivamente, no pertenece a esa categoría.
3. Falsos Positivos (`FP`; False Positive): Son los casos en los que el modelo predijo incorrectamente la clase positiva. Esto ocurre cuando el modelo dice que algo pertenece a una categoría específica, pero en realidad no es así.
4. Falsos Negativos (`FN`; False Negative): Son los casos en los que el modelo predijo incorrectamente la clase negativa. Es decir, el modelo indicó que algo no pertenecía a una categoría específica, pero en realidad sí pertenece a esa categoría.

En el contexto de scikit-learn, la matriz de confusión puede generarse fácilmente utilizando la función `confusion_matrix` dentro del módulo `metrics`. Esta función requiere como parámetros las etiquetas reales y las predicciones del modelo, devolviendo una matriz que facilita la evaluación del desempeño del modelo en tareas de clasificación.

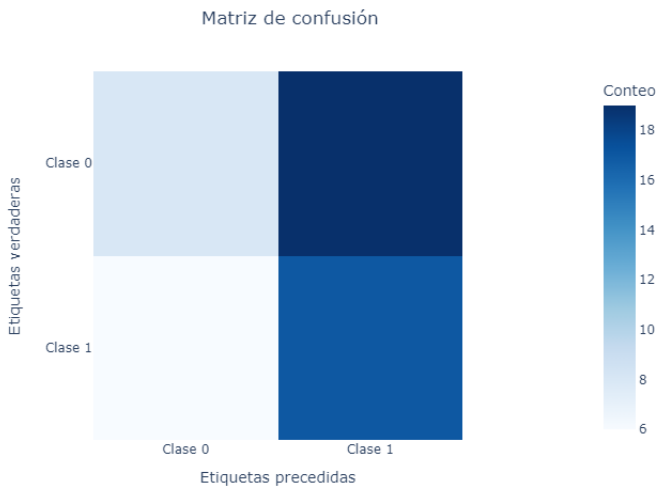
A continuación se muestra cómo obtener y visualizar la matriz de confusión en Python:

[8]:
```
# Import additional libraries
from sklearn.metrics import confusion_matrix
import plotly.express as px

# Generate the confusion matrix
cm = confusion_matrix(testLabels, predictions)

# Visualize the confusion matrix using Plotly Express
fig = px.imshow(cm,
                labels=dict(x="Etiquetas precedidas", y="Etiquetas verdaderas", color="Conteo"),
                x=["Clase 0", "Clase 1"],  # Replace with class names if necessary
                y=["Clase 0", "Clase 1"],
                 title="Matriz de confusión",
                color_continuous_scale="Blues")

fig.update_xaxes(side="bottom")
fig.show()
```

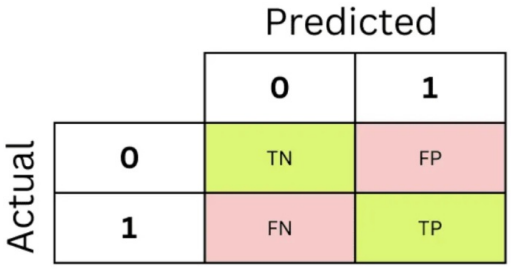


6.4 Definición de precisión y recuerdo

La precisión (precision) y el recuerdo (recall) son dos métricas fundamentales utilizadas para evaluar el desempeño de modelos de clasificación en aprendizaje automático. Estas métricas son particularmente útiles en situaciones donde las clases están desbalanceadas, es decir, cuando una clase es mucho más frecuente que otra.

La **precisión** mide la exactitud de las predicciones positivas del modelo. Se calcula como el número de verdaderos positivos (`TP`) dividido entre el total de instancias que fueron clasificadas como positivas (la suma de `TP` y falsos positivos, `FP`). En otras palabras, la precisión responde a la pregunta: "De todas las instancias que el modelo predijo como positivas, ¿cuántas son realmente positivas?". Un modelo con alta precisión comete pocos errores al clasificar instancias como positivas, lo que es crucial en contextos donde los falsos positivos tienen un alto costo, como en el diagnóstico médico.

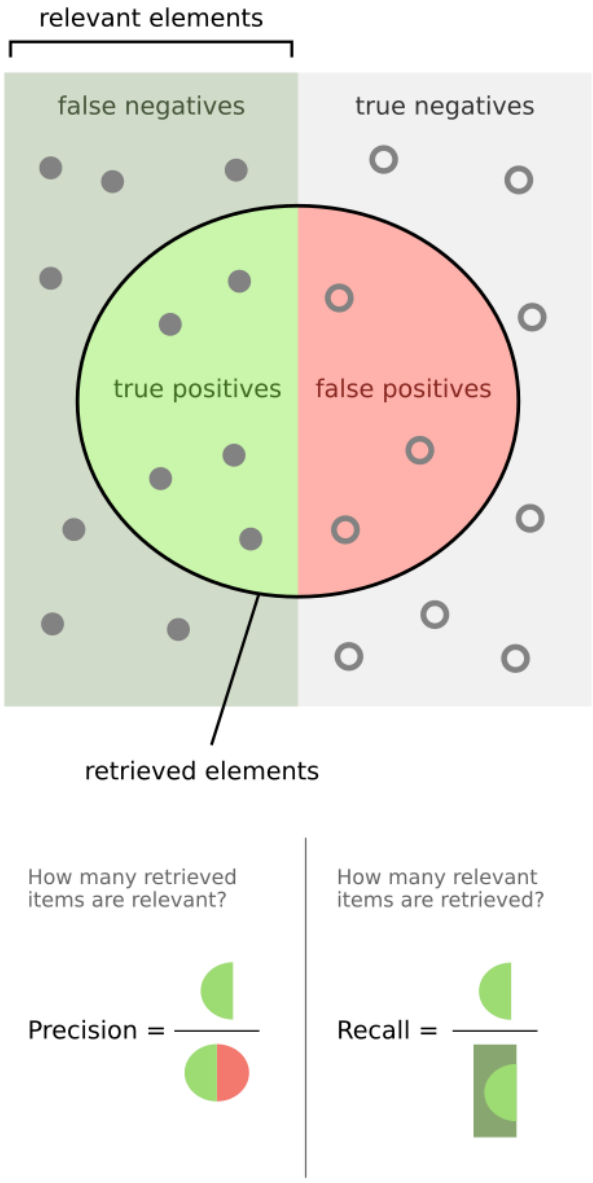
El **recuerdo**, por otro lado, mide la capacidad del modelo para identificar correctamente todas las instancias positivas. Se calcula como el número de verdaderos positivos (`TP`) dividido entre el total de instancias que son realmente positivas (la suma de `TP` y falsos negativos, `FN`). El recuerdo responde a la pregunta: "De todas las instancias que son realmente positivas, ¿cuántas fueron identificadas correctamente por el modelo?". Un modelo con alto recuerdo es eficaz para encontrar todas las instancias positivas, lo que es vital en escenarios donde es más importante no dejar pasar ninguna instancia positiva, como en la detección de fraudes.



$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

En scikit-learn, estas métricas pueden calcularse fácilmente utilizando las funciones `precision_score` y `recall_score` del módulo `metrics`. Estas funciones permiten evaluar el modelo después de realizar las predicciones, proporcionando un análisis detallado que puede ayudar a ajustar y mejorar el rendimiento del modelo, especialmente en problemas de clasificación donde es crucial balancear correctamente la precisión y el recuerdo.



A continuación se presenta como calcular la precisión y el recuerdo utilizando la librería de Scikit-learn en Python:

[24]:
```
from sklearn.metrics import precision_score, recall_score

# Calculate precision and recall for '1.0' as positive class
precision1 = precision_score(testLabels, predictions, pos_label='1.0')
recall1 = recall_score(testLabels, predictions, pos_label='1.0')

# Calculate precision and recall for '0.0' as positive class
precision0 = precision_score(testLabels, predictions, pos_label='0.0')
recall0 = recall_score(testLabels, predictions, pos_label='0.0')
```

```
# Print the results
print("When '1.0' is positive class:")
print("Precision:", precision1)
print("Recall:", recall1)

print("When '0.0' is positive class:")
print("Precision:", precision0)
print("Recall:", recall0)
```

```
When '1.0' is positive class:
Precision: 0.4722222222222222
Recall: 0.7391304347826086
When '0.0' is positive class:
Precision: 0.5714285714285714
Recall: 0.2962962962962963
```

6.5 Clase positiva y negativa en métricas de clasificación

En problemas de clasificación binaria, se tienen dos clases: una se denomina **positiva** y la otra **negativa**. La clase positiva es aquella que se desea identificar o para la cual se pretende medir el rendimiento del modelo, mientras que la clase negativa es la otra categoría. En un sistema de detección de correos electrónicos, por ejemplo, spam sería la clase positiva, ya que es el tipo de correo que se quiere identificar y filtrar. Por otro lado, no spam sería la clase negativa, ya que no es el foco principal del análisis y no requiere una atención especial en el proceso de clasificación.

El concepto de **clase positiva** es crucial para el cálculo de métricas de rendimiento como la precisión y el recuerdo. En la biblioteca scikit-learn, al calcular estas métricas, es necesario especificar cuál es la clase positiva mediante el parámetro `pos_label` en las funciones `precision_score` y `recall_score`. Este parámetro indica a las funciones cuál clase debe considerarse como la positiva durante los cálculos.

En el conjunto de datos que se esta usando para este laboratorio, si la clase de interés es `1`, se debe especificar `pos_label='1'` al usar las funciones de scikit-learn. Esto asegura que las métricas se calculen correctamente en relación con la clase positiva deseada.

[25]:
```
import plotly.express as px
import pandas as pd

# Create a DataFrame for the precision and recall metrics
metrics = pd.DataFrame({
    "Metric": ["Precision", "Recall"],
    "Value": [precision, recall]
})

metrics_df = pd.DataFrame({
    "Class": ["0.0", "0.0", "1.0", "1.0"],
    "Metric": ["Precision", "Recall", "Precision", "Recall"],
    "Value": [precision0, recall0, precision1, recall1]
```

```
})

# Create the grouped bar plot
fig = px.bar(metrics_df,
             x='Class',
             y='Value',
             color='Metric',
             barmode='group',
             title="Precision and Recall Comparison for Classes 0 and 1",
             labels={'Value': 'Score', 'Class': 'Class'},
             text_auto=True)

# Show the plot
fig.show()
```

6.6 Precisión vs recuerdo

Cuando un modelo de clasificación muestra una mayor precisión en comparación con su recuerdo, esto indica que el modelo es muy efectivo en identificar casos positivos correctamente entre las predicciones que realiza. En otras palabras, de todos los casos que el modelo clasifica como positivos, una alta proporción realmente lo es. Esto es beneficioso en situaciones donde es fundamental minimizar los falsos positivos, como en el diagnóstico de enfermedades graves. Por ejemplo, si un modelo tiene una alta precisión al identificar cáncer, significa que cuando el modelo dice que alguien tiene cáncer, es muy probable que realmente lo tenga, reduciendo el riesgo de diagnósticos erróneos. Sin embargo, una alta precisión a menudo puede venir a expensas de un menor recuerdo, lo que significa que algunos casos positivos verdaderos pueden no ser detectados. En contextos donde detectar todos los casos positivos es crucial, esta falta de recuerdo puede ser una desventaja significativa.

En contraste, cuando un modelo tiene un mayor recuerdo en comparación con su precisión, el modelo es eficiente en detectar la mayor parte de los casos positivos verdaderos entre todos los que deberían ser positivos. Esto significa que el modelo tiene una alta capacidad para identificar los

positivos, aunque esto pueda resultar en una mayor cantidad de falsos positivos. En aplicaciones donde es vital no pasar por alto ningún caso positivo, como en la detección de fraudes financieros, un alto recuerdo es crucial. Esto asegura que la mayoría de los fraudes sean identificados, aunque algunas transacciones legítimas puedan ser incorrectamente clasificadas como fraudulentas. Sin embargo, un alto recuerdo puede implicar una menor precisión, lo que puede llevar a una mayor cantidad de errores de clasificación, y esto debe ser considerado dependiendo de las implicaciones de estos errores en el contexto específico.