



**Instituto Politécnico Nacional
Escuela Superior de Cómputo**

**Ingeniería en Inteligencia Artificial
Machine Learning**

Semestre: 2025-1

Grupo: 5BM1

Práctica 9

Fecha de entrega: 19 de noviembre de 2024



Laboratorio 9: SMOTE y Perceptrón Simple

Profesor: M. en C. Andrés Floriano García

Integrantes:

Juan Manuel Alvarado Sandoval
Alexander Iain Crombie Esquinca
Herrera Saavedra Jorge Luis
Quiñones Mayorga Rodrigo

Contents

1	Introducción	3
1.1	Perceptrón Simple y Funciones de Activación	3
2	Metodología	5
2.1	Preparación del entorno	5
2.2	Aplicación de clasificadores	5
2.3	Validaciones	5
2.4	Análisis de resultados	5
3	Resultados	6
3.1	Preparación	6
3.2	Aplicación de clasificadores y validación	7
3.3	Análisis de Resultados	13
4	Conclusiones	14
5	Enlace al Repositorio	14
6	Bibliografía	15

1 Introducción

En esta práctica se aborda el uso del método **SMOTE** (Synthetic Minority Oversampling Technique) para el balanceo de clases en conjuntos de datos desbalanceados. Además, se explora el desempeño de clasificadores como el *Euclidiano* y el *1NN* antes y después de aplicar SMOTE, utilizando validaciones *Hold-Out* y *10-Fold Cross-Validation*. Se aplicarán estos conceptos en el conjunto de datos **Glass**.

1.1 Perceptrón Simple y Funciones de Activación

El **Perceptrón Simple** es un modelo de red neuronal desarrollado por Frank Rosenblatt en 1958. Es una unidad de procesamiento que utiliza un conjunto de entradas, ponderadas con valores específicos, para producir una salida binaria basada en una función de activación.

Estructura del Perceptrón Simple

La estructura básica del Perceptrón Simple incluye:

- **Entradas:** Representan las características del problema a resolver.
- **Pesos:** Son los valores que multiplican a las entradas para determinar su importancia relativa.
- **Sesgo:** Un término adicional que permite ajustar la salida del modelo.
- **Salida:** Calculada utilizando una función de activación sobre la combinación lineal de entradas y pesos.

El modelo se entrena ajustando los pesos y el sesgo para minimizar los errores en las predicciones.

Funciones de Activación

Las funciones de activación son esenciales para decidir si una neurona debe activarse o no. En el caso del Perceptrón Simple, se utiliza típicamente la función **umbral**:

$$f(x) = \begin{cases} 1 & \text{si } x \geq 0, \\ 0 & \text{si } x < 0. \end{cases}$$

Otras funciones de activación comunes incluyen:

- **Sigmoide:**

$$f(x) = \frac{1}{1 + e^{-x}}$$

- **ReLU (Rectified Linear Unit):**

$$f(x) = \max(0, x)$$

Cada función tiene aplicaciones específicas según el problema y los requerimientos del modelo.




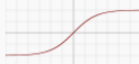





Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Figure 1: Algunas funciones de activación.

2 Metodología

Para realizar esta práctica, se siguieron los siguientes pasos:

2.1 Preparación del entorno

1. Implementación del método SMOTE según lo descrito en el artículo:
<https://www.analyticsvidhya.com/blog/2020/10/overcoming-class-imbalance-using-smote-te/>
2. Carga y preprocesamiento del conjunto de datos **Glass**.

2.2 Aplicación de clasificadores

1. Aplicación del clasificador *Euclidiano*.
2. Aplicación del clasificador *1NN*.
3. Evaluación del desempeño antes y después de aplicar el método SMOTE.

2.3 Validaciones

1. Uso de *Hold-Out* para dividir los datos en conjuntos de entrenamiento y prueba.
2. Uso de *10-Fold Cross-Validation* para evaluar el desempeño de los clasificadores.

2.4 Análisis de resultados

1. Comparación de métricas de desempeño antes y después de aplicar SMOTE.
2. Discusión sobre las mejoras obtenidas y su impacto en la clasificación.

3 Resultados

3.1 Preparación

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe
0	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.0
1	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.0
2	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0.0	0.0
3	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.0
4	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.0

Figure 2: Primeras filas de Glass Dataset.

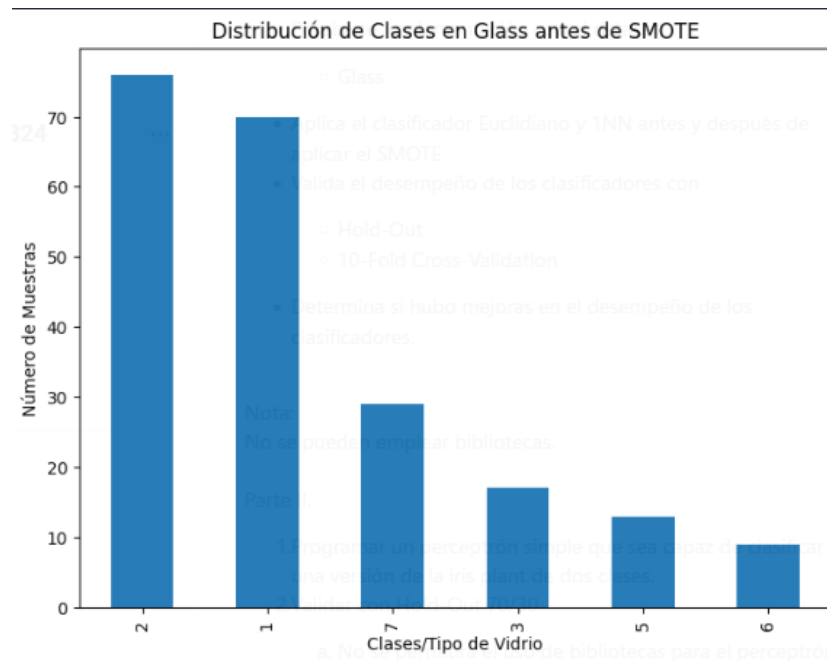


Figure 3: Distribución de clases antes de aplicar SMOTE.

3.2 Aplicación de clasificadores y validación

A continuación, se aplicaron los clasificadores *1NN* y el modelo *Euclidiano* al conjunto de datos **Glass** sin aplicar el método SMOTE, manteniendo el desbalance de clases. Posteriormente, se repitió el proceso tras aplicar el método SMOTE para observar las diferencias en el rendimiento.

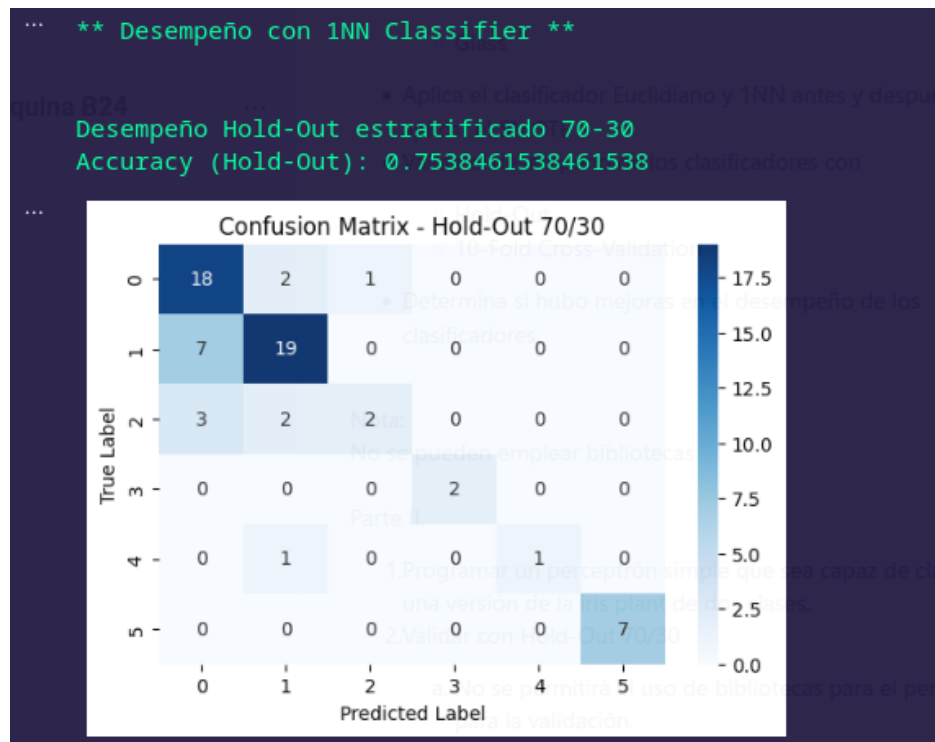


Figure 4: Matriz de confusión y accuracy (Hold Out) para 1NN sin SMOTE.

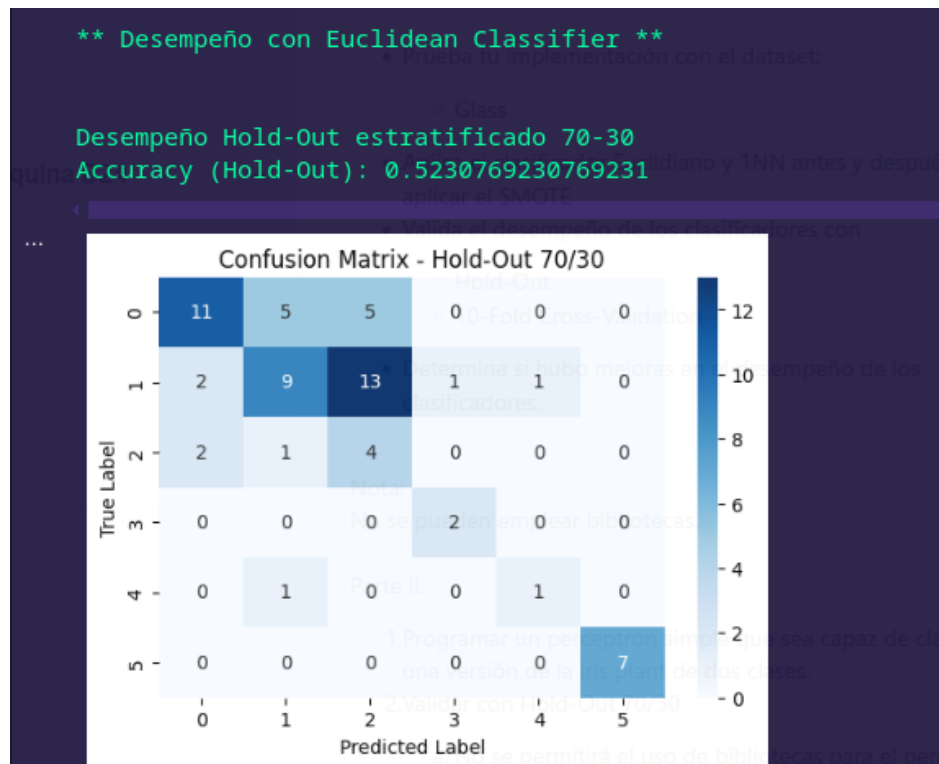


Figure 5: Matriz de confusión y accuracy (Hold Out) para Euclidiano sin SMOTE.

```

Fold 1 Accuracy: 0.7272727272727273
Fold 2 Accuracy: 0.5909090909090909
Fold 3 Accuracy: 0.5454545454545454
Fold 4 Accuracy: 0.8636363636363636
Fold 5 Accuracy: 0.8571428571428571
Fold 6 Accuracy: 0.8095238095238095
Fold 7 Accuracy: 0.7142857142857143
Fold 8 Accuracy: 0.8095238095238095
Fold 9 Accuracy: 0.5714285714285714
Fold 10 Accuracy: 0.5714285714285714
Average Accuracy (Stratified 10-Fold): 0.7060606060606060

```

Figure 6: 10 Fold-Cross Validation para 1NN sin SMOTE.


```

Fold 1 Accuracy: 0.6363636363636364
Fold 2 Accuracy: 0.5909090909090909
Fold 3 Accuracy: 0.5
Fold 4 Accuracy: 0.3181818181818182
Fold 5 Accuracy: 0.5714285714285714
Fold 6 Accuracy: 0.42857142857142855
Fold 7 Accuracy: 0.6190476190476191
Fold 8 Accuracy: 0.47619047619047616
Fold 9 Accuracy: 0.47619047619047616
Fold 10 Accuracy: 0.38095238095238093
Average Accuracy (Stratified 10-Fold): 0.4997835497835498

```

Figure 7: 10 Fold-Cross Validation para Euclidiano sin SMOTE

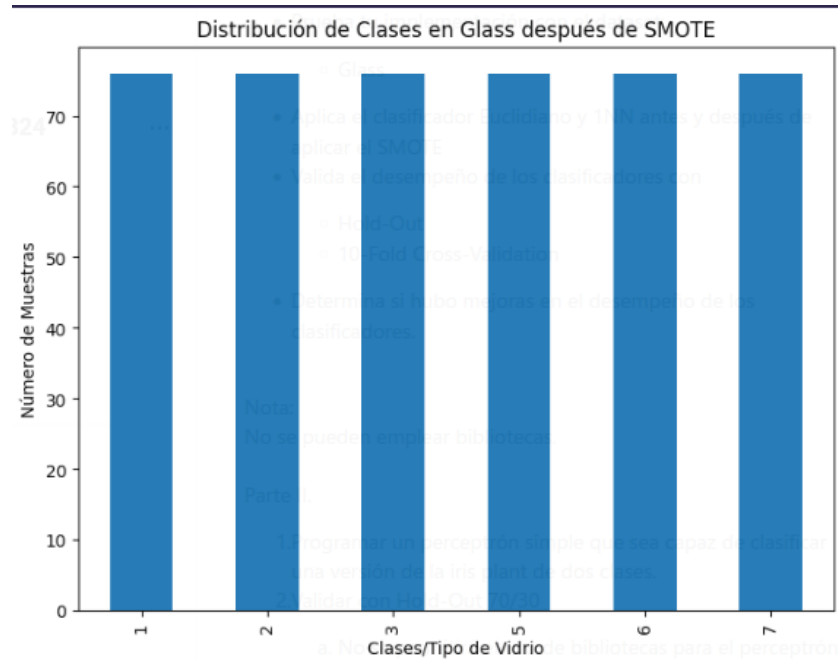


Figure 8: Distribución de clases después de aplicar SMOTE.

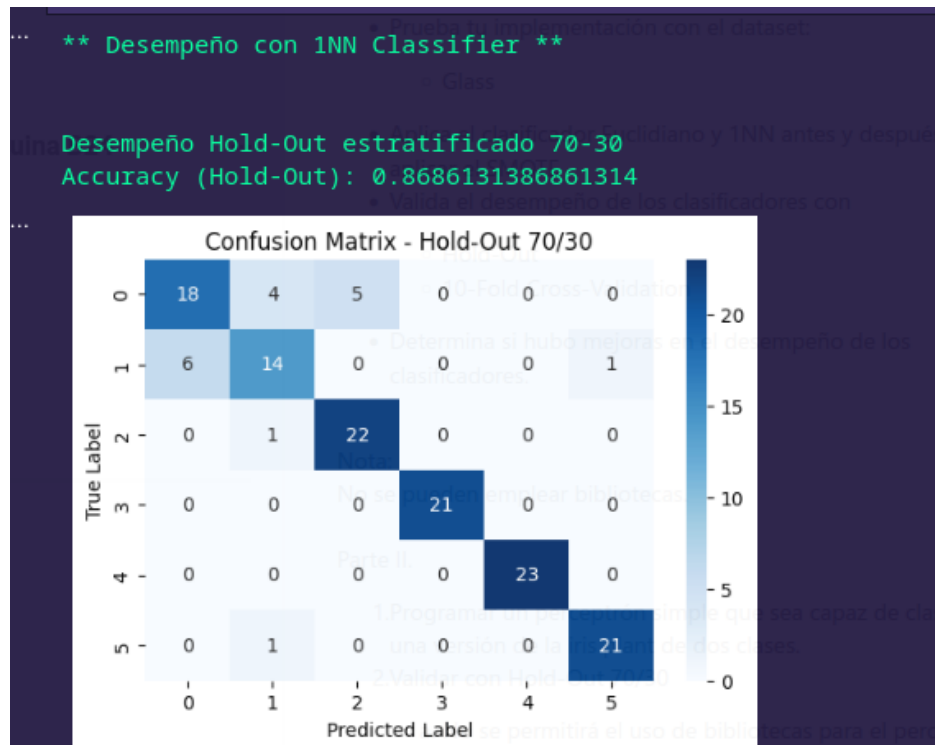


Figure 9: Distribución de clases antes de aplicar SMOTE. Matriz de confusión y accuracy (Hold Out) para 1NN con SMOTE.

```

...na
Fold 1 Accuracy: 0.9130434782608695
Fold 2 Accuracy: 0.8913043478260869
Fold 3 Accuracy: 0.8695652173913043
Fold 4 Accuracy: 0.9347826086956522
Fold 5 Accuracy: 0.8478260869565217
Fold 6 Accuracy: 0.9565217391304348
Fold 7 Accuracy: 0.8888888888888888
Fold 8 Accuracy: 0.8666666666666667
Fold 9 Accuracy: 0.9111111111111111
Fold 10 Accuracy: 0.9111111111111111
Average Accuracy (Stratified 10-Fold): 0.8990821256038648

```

Figure 10: 10 Fold-Cross Validation para 1NN con SMOTE

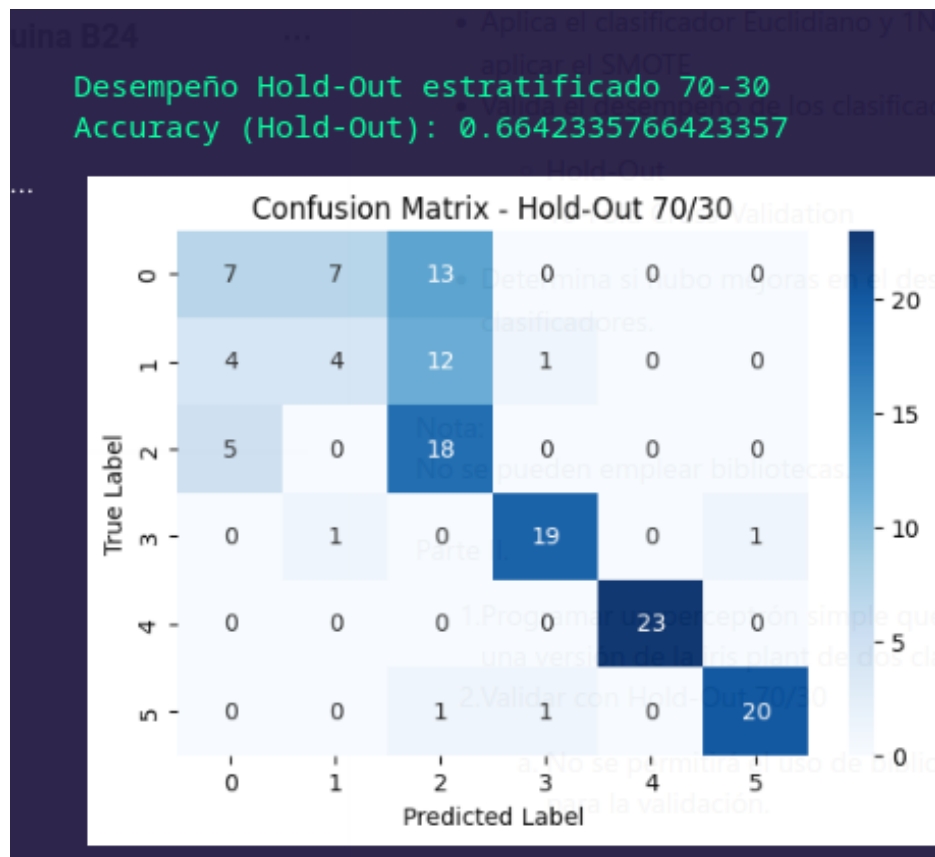


Figure 11: Matriz de confusión y accuracy (Hold Out) para Euclidiano con SMOTE

```

Fold 1 Accuracy: 0.717391304347826
Fold 2 Accuracy: 0.717391304347826
Fold 3 Accuracy: 0.6521739130434783
Fold 4 Accuracy: 0.717391304347826
Fold 5 Accuracy: 0.6739130434782609
Fold 6 Accuracy: 0.717391304347826
Fold 7 Accuracy: 0.7111111111111111
Fold 8 Accuracy: 0.6666666666666666
Fold 9 Accuracy: 0.7555555555555555
Fold 10 Accuracy: 0.6222222222222222
Average Accuracy (Stratified 10-Fold): 0.6951207729468599

```

Figure 12: 10 Fold-Cross Validation para Euclidiano con SMOTE.

```

class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, epochs=10):
        self.weights = np.random.uniform(-1, 1, input_size) # inicializar pesos aleatoriamente
        self.bias = random.uniform(-1, 1) # inicializar el sesgo aleatoriamente
        self.learning_rate = learning_rate
        self.epochs = epochs

    def activation(self, x):
        # Función de activación (paso binario)
        return 1 if x >= 0 else -1

    def train(self, X, y):
        # Entrenamiento del perceptrón
        for epoch in range(self.epochs):
            for i in range(len(X)):
                x = X[i]
                y_true = y[i]
                # Calcular la salida del perceptrón
                y_pred = self.activation(np.dot(self.weights, x) + self.bias) #  $W_i \cdot X_i + b_i$ 
                # Ajuste de pesos y sesgo si hay error
                if y_true != y_pred:
                    self.weights += self.learning_rate * (y_true - y_pred) * x
                    self.bias += self.learning_rate * (y_true - y_pred)

    def predict(self, x):
        # Predicción para una entrada
        return self.activation(np.dot(self.weights, x) + self.bias)

```

Figure 13: Implementación de la clase para aplicar perceptrón simple.

```

# Dividir en conjuntos de entrenamiento y prueba (70/30)
split_index = int(len(X) * 0.7)
X_train, X_test = X[:split_index], X[split_index:]
y_train, y_test = y[:split_index], y[split_index:]

# Crear y entrenar el modelo
perceptron = Perceptron(input_size=X.shape[1], learning_rate=0.1, epochs=100)
perceptron.train(X_train, y_train)

# Validación en el conjunto de prueba
correct_predictions = 0
for i in range(len(X_test)):
    prediction = perceptron.predict(X_test[i])
    if prediction == y_test[i]:
        correct_predictions += 1

# Calcular y mostrar la accuracy
accuracy = correct_predictions / len(X_test)
print("Hold Out Accuracy perceptron:", accuracy)

```

Hold Out Accuracy perceptron: 1.0

Figure 14: Accuracy perceptrón binario (Hold Out 70-30).

3.3 Análisis de Resultados

Se puede observar que aumentó el rendimiento tanto en *1NN* como en el modelo *Euclidiano*.

Para el modelo *1NN*, antes de aplicar SMOTE, se alcanzó un *accuracy* de 0.75 y de 0.706 para *Hold-Out* y el promedio de *10-Fold Cross-Validation* respectivamente. En contraste, tras aplicar SMOTE, se observaron resultados de *accuracy* de **0.86** y **0.89** respectivamente.

Para el clasificador *Euclidiano*, antes de aplicar SMOTE, se alcanzó un *accuracy* de 0.52 y de 0.49 para *Hold-Out* y el promedio de *10-Fold Cross-Validation* respectivamente. En contraste, tras aplicar SMOTE, se observaron resultados de *accuracy* de **0.664** y **0.695** respectivamente.

Además, al aplicar el Perceptrón Simple al conjunto de datos **Iris**, se obtuvo un *accuracy* perfecto del **100%** (1.0) para clasificar las clases **Setosa** y **Virginica**. Esto indica que el Perceptrón Simple es altamente efectivo en problemas linealmente separables, como lo son estas dos clases en el conjunto *Iris*. Este rendimiento ideal resalta que, para problemas de este tipo, un modelo simple puede ofrecer soluciones rápidas y precisas sin requerir algoritmos más complejos.

Lo anterior prueba que SMOTE es una técnica esencial para reducir el impacto del desbalance de clases mediante *oversampling* y mejorar el rendimiento de los clasificadores, sin perder la esencia de la información original del conjunto de datos. Adicionalmente, el desempeño sobresaliente del Perceptrón Simple en problemas linealmente separables subraya su importancia como una herramienta básica y eficiente para ciertos escenarios.

4 Conclusiones

En esta práctica, se exploraron dos enfoques fundamentales para mejorar el rendimiento de los clasificadores: la técnica SMOTE para abordar el desbalance de clases y el Perceptrón Simple como un modelo base para clasificación.

Se concluye que:

- SMOTE demostró ser una técnica eficiente para mejorar el *accuracy* de clasificadores como *1NN* y el modelo *Euclidiano*, particularmente en contextos con desbalance de clases severo.
- El Perceptrón Simple alcanzó un rendimiento perfecto en la clasificación de las clases **Setosa** y **Virginica** en el conjunto de datos *Iris*, destacando su utilidad en problemas linealmente separables.
- Las métricas de validación, como *Hold-Out* y *10-Fold Cross-Validation*, son esenciales para medir de forma objetiva las mejoras obtenidas tras aplicar métodos como SMOTE.
- La combinación de técnicas de preprocesamiento de datos (como SMOTE) con modelos adecuados puede aumentar significativamente la calidad de las predicciones y la robustez del sistema.

5 Enlace al Repositorio

[Haz click para seguir el enlace](#)

6 Bibliografía

1. Analytics Vidhya. (2020). *Overcoming Class Imbalance Using SMOTE Techniques*. Recuperado de: <https://www.analyticsvidhya.com/blog/2020/10/overcoming-class-imbalance-using-smote-techniques/>