Actividad: Laboratorio 9: SMOTE y Perceptrón Simple

Alumno:

Miguel Angel Ocampo

Gerardo Martinez Ayala

Maestro:

Andres Garcia Floriano

Explicación del código.

Este código es un ejemplo educativo diseñado para explorar técnicas de clasificación, manejo de datos desbalanceados y métodos de evaluación en problemas de Machine Learning. Se centra en tres componentes principales: la carga y preprocesamiento de datos, el desarrollo de algoritmos básicos de clasificación y la implementación de técnicas de evaluación como la validación Hold-Out y la validación cruzada.

En primer lugar, el código incluye una función llamada load_datasets, cuya finalidad es cargar conjuntos de datos y prepararlos para ser utilizados en los experimentos. Se trabaja con dos datasets: el conjunto de datos Iris, conocido en la comunidad de Machine Learning, y el dataset Glass, que se descarga desde OpenML. En el caso de Iris, se filtran las clases Setosa y Virginica, simplificando el problema a una tarea de clasificación binaria. Esto se logra identificando las instancias de las dos clases y extrayéndolas del conjunto de datos original. Para el dataset Glass, las etiquetas categóricas se convierten en valores numéricos únicos mediante un mapeo, asegurando que el conjunto de datos esté listo para ser procesado por algoritmos que requieren entradas numéricas.

Un aspecto destacado del código es la implementación manual de la técnica SMOTE (Synthetic Minority Oversampling Technique). Este método se utiliza para abordar el problema de clases desbalanceadas, que puede afectar negativamente el rendimiento de los clasificadores al sesgar el aprendizaje hacia la clase mayoritaria. SMOTE genera ejemplos sintéticos para la clase minoritaria mediante la interpolación entre puntos cercanos en el espacio de características. En este caso, se seleccionan vecinos aleatorios dentro de la clase minoritaria, y se generan nuevas muestras calculando combinaciones lineales aleatorias de las diferencias entre estos vecinos y las instancias originales. Finalmente, estas muestras se agregan al conjunto de datos, aumentando el tamaño de la clase minoritaria y mejorando el balance general de los datos.

El código también implementa un clasificador basado en la distancia euclidiana, una técnica sencilla que predice las etiquetas de las muestras de prueba basándose en su vecino más cercano dentro del conjunto de entrenamiento. Este enfoque utiliza la norma euclidiana para calcular distancias entre puntos en el espacio de características, seleccionando la etiqueta de la instancia más cercana como la predicción final. Aunque este método no es sofisticado, es útil para comprender cómo los clasificadores simples pueden funcionar en datos pequeños o estructurados.

Para evaluar los modelos, el código utiliza dos enfoques comunes: validación Hold-Out y validación cruzada. La validación Hold-Out divide los datos en conjuntos de entrenamiento y prueba en una proporción específica, típicamente 70/30. Este método proporciona una evaluación directa de los modelos en datos no vistos, aunque puede ser sensible a cómo se dividen los datos. Por otro lado, la validación cruzada divide los datos en k particiones o "folds" y entrena el modelo k veces, utilizando un fold diferente como conjunto de prueba en cada iteración. Este enfoque reduce la varianza asociada a una

sola partición, proporcionando una evaluación más robusta y representativa del rendimiento del modelo.

El código luego aborda un problema de clasificación utilizando un perceptrón simple, un algoritmo clásico en Machine Learning. Este modelo ajusta pesos para cada característica durante el entrenamiento mediante un proceso iterativo. En cada iteración, el perceptrón calcula una salida lineal combinando las características con los pesos y ajusta los pesos en función de los errores de predicción. Este algoritmo es particularmente adecuado para problemas de clasificación binaria linealmente separables. Además, las características se normalizan antes del entrenamiento, una práctica que mejora la estabilidad numérica y el rendimiento general del modelo.

El flujo principal del código demuestra cómo aplicar estas técnicas en situaciones prácticas. Para el dataset Glass, se identifica la clase minoritaria y se aplica SMOTE para equilibrar las clases. Luego, se evalúa el rendimiento del clasificador Euclidiano utilizando validación Hold-Out y cruzada, tanto antes como después de aplicar SMOTE, para analizar el impacto de esta técnica en la precisión del modelo. Para el dataset Iris, se implementa el perceptrón para clasificar las clases Setosa y Virginica, demostrando cómo un modelo sencillo puede lograr buenos resultados en un problema de clasificación binaria bien definido. La precisión del perceptrón se evalúa en un conjunto de prueba separado utilizando validación Hold-Out, lo que permite comparar su rendimiento con otras técnicas.

En resumen, este código combina técnicas fundamentales de preprocesamiento, implementación de clasificadores simples y métodos rigurosos de evaluación. Sirve como una introducción práctica a conceptos esenciales de Machine Learning, especialmente en el contexto de conjuntos de datos desbalanceados y problemas de clasificación binaria. Además, destaca la importancia de manejar el desbalance de clases, seleccionar métodos de evaluación apropiados y aplicar técnicas básicas para obtener insights iniciales sobre problemas más complejos. Es un excelente punto de partida para desarrollar una comprensión más profunda de los desafíos y enfoques en la construcción de modelos de aprendizaje automático.

Explicación de los resultados

1. Hold-Out Accuracy (Original Data): 0.7077

Este resultado refleja la precisión obtenida utilizando el clasificador basado en distancia euclidiana sobre los datos originales sin aplicar SMOTE. Una precisión del 70.77% indica que el modelo predijo correctamente aproximadamente el 71% de las muestras en el conjunto de prueba. Dado que el conjunto de datos es desbalanceado, es posible que el modelo esté sesgado hacia la clase mayoritaria, lo que limita su capacidad para clasificar correctamente las instancias de la clase minoritaria.

2. Hold-Out Accuracy (After SMOTE): 0.6716

Después de aplicar SMOTE, la precisión disminuyó ligeramente al 67.16%. Esto ocurre porque SMOTE introduce ejemplos sintéticos en el conjunto de datos, equilibrando las clases, pero también puede generar ruido en los datos. Aunque el modelo ahora tiene acceso a más instancias de la clase minoritaria, el clasificador euclidiano, que es sensible a la distribución de los datos, puede verse afectado por la variabilidad de los ejemplos sintéticos. Es importante destacar que esta disminución en precisión podría ser compensada por un mejor rendimiento en métricas específicas para la clase minoritaria, como la sensibilidad o la F1-Score.

3. 10-Fold CV Accuracy (Original Data): 0.7286

La validación cruzada con 10 particiones sobre los datos originales dio una precisión promedio del 72.86%. Este resultado es ligeramente superior al obtenido con Hold-Out en los datos originales, lo que refleja la robustez de la validación cruzada al promediar los resultados de múltiples particiones y reducir la varianza asociada a una sola división de datos. Sin embargo, al igual que en el caso anterior, el modelo probablemente esté sesgado hacia la clase mayoritaria.

4. 10-Fold CV Accuracy (After SMOTE): 0.7409

Después de aplicar SMOTE, la validación cruzada muestra un incremento en la precisión promedio al 74.09%. Esto sugiere que equilibrar las clases mediante SMOTE ayudó al modelo a mejorar su rendimiento general, posiblemente al permitirle identificar mejor las instancias de la clase minoritaria durante el entrenamiento. Este aumento, aunque modesto, es indicativo de que SMOTE puede ser beneficioso cuando se utiliza junto con métodos de evaluación robustos como la validación cruzada.

5. Hold-Out Accuracy (Perceptrón): 1.00

El perceptrón logró una precisión perfecta del 100% en el conjunto de prueba para el problema de clasificación binaria entre Setosa y Virginica en el dataset Iris. Este resultado es notable, pero también debe interpretarse con cautela. La tarea específica (clasificación de Setosa y Virginica) es conocida por ser linealmente separable en el espacio de características del dataset Iris, lo que permite al perceptrón, un clasificador lineal, separar perfectamente las dos clases. Además, dado que se utilizó una validación Hold-Out, el resultado puede estar influenciado por una división afortunada de los datos; una evaluación más robusta, como la validación cruzada, podría proporcionar una medida más confiable del rendimiento.

Conclusiones Generales

1. Impacto de SMOTE:
    - En el dataset Glass, SMOTE equilibró las clases y mejoró ligeramente la precisión promedio en validación cruzada, aunque redujo la precisión en Hold-Out. Esto resalta la importancia de evaluar métricas adicionales (por ejemplo, sensibilidad y F1-Score) para capturar los beneficios reales de equilibrar las clases.
2. Validez de los Resultados:
    - La validación cruzada muestra resultados más confiables al reducir la varianza causada por una única división de datos. Es evidente que SMOTE tiene un impacto positivo en este contexto.
3. Perceptrón en el Dataset Iris:
    - La precisión perfecta del perceptrón en el dataset Iris es esperada debido a la naturaleza linealmente separable del problema. Este resultado resalta la utilidad del perceptrón en problemas de este tipo, pero también su limitación cuando las clases no son separables linealmente.
4. Lecciones Aprendidas:
    - Los clasificadores simples como el basado en distancia euclidiana o el perceptrón pueden ser efectivos en escenarios específicos, pero su desempeño depende en gran medida de la distribución de los datos y de la naturaleza del problema.
    - El uso de técnicas como SMOTE y validación cruzada ayuda a mitigar problemas asociados con clases desbalanceadas y divisiones de datos no representativas.

Evidencias.

```python
# Parte I - Implementación de SMOTE y Clasificadores

import numpy as np
import random
from sklearn.datasets import load_iris, fetch_openml

# Función para cargar datasets automáticamente
def load_datasets():
    datasets = {}

    # Cargar Iris (clases Setosa y Virginica únicamente)
    iris = load_iris()
    iris_X = iris['data']
    iris_y = iris['target']

    # Filtrar solo Setosa y Virginica
    filter_indices = np.where((iris_y == 0) | (iris_y == 1))
    datasets['Iris'] = (iris_X[filter_indices], iris_y[filter_indices])

    # Cargar Glass desde OpenML
    glass = fetch_openml(name="glass", version=1, as_frame=False)
    glass_X = glass.data
    glass_y = glass.target
```

```
    # Convertir etiquetas a valores numéricos únicos
    unique_labels = np.unique(glass_y)
    label_to_numeric = {label: idx for idx, label in enumerate(unique_labels)}
    numeric_glass_y = np.array([label_to_numeric[label] for label in glass_y])

    datasets['Glass'] = (glass_X, numeric_glass_y)

    return datasets

# SMOTE
def smote(X, y, minority_class, k=5):
    X_minority = X[y == minority_class]
    n_samples, n_features = X_minority.shape
    new_samples = []

    for _ in range(len(X_minority)):
        i = random.randint(0, n_samples - 1)
        neighbors = [random.randint(0, n_samples - 1) for _ in range(k)]
        neighbor = X_minority[random.choice(neighbors)]

        diff = neighbor - X_minority[i]
        new_sample = X_minority[i] + random.random() * diff
        new_samples.append(new_sample)

    new_samples = np.array(new_samples)
    X_augmented = np.vstack((X, new_samples))
    y_augmented = np.hstack((y, np.full(len(new_samples), minority_class)))
```

```python
    return X_augmented, y_augmented

# Clasificador Euclidiano
def euclidean_classifier(X_train, y_train, X_test):
    predictions = []
    for x in X_test:
        distances = np.linalg.norm(X_train - x, axis=1)
        nearest_index = np.argmin(distances)
        predictions.append(y_train[nearest_index])
    return np.array(predictions)

# Validación Hold-Out
def hold_out_validation(X, y, test_ratio=0.3):
    n_samples = X.shape[0]
    indices = np.arange(n_samples)
    np.random.shuffle(indices)
    split_point = int(n_samples * (1 - test_ratio))
    train_indices, test_indices = indices[:split_point], indices[split_point:]
    return X[train_indices], y[train_indices], X[test_indices], y[test_indices]

# Validación Cruzada
def k_fold_cross_validation(X, y, k=10):
    n_samples = len(y)
    fold_size = n_samples // k
    indices = np.arange(n_samples)
    np.random.shuffle(indices)
    folds = [indices[i * fold_size:(i + 1) * fold_size] for i in range(k)]
```

```
    scores = []
    for i in range(k):
        test_indices = folds[i]
        train_indices = np.hstack([folds[j] for j in range(k) if j != i])

        X_train, y_train = X[train_indices], y[train_indices]
        X_test, y_test = X[test_indices], y[test_indices]

        predictions = euclidean_classifier(X_train, y_train, X_test)
        accuracy = np.mean(predictions == y_test)
        scores.append(accuracy)

    return np.mean(scores)

# Cargar los datasets
datasets = load_datasets()

# Trabajar con el dataset Glass
X, y = datasets['Glass']

# Identificar clase minoritaria
unique_classes, class_counts = np.unique(y, return_counts=True)
minority_class = unique_classes[np.argmin(class_counts)]

# Validación Hold-Out
X_train, y_train, X_test, y_test = hold_out_validation(X, y)
print("Hold-Out Accuracy (Original Data):", np.mean(euclidean_classifier(X_train, y_train, X_test) == y_test))
```

```
# Validación Hold-Out
X_train, y_train, X_test, y_test = hold_out_validation(X, y)
print("Hold-Out Accuracy (Original Data):", np.mean(euclidean_classifier(X_train, y_train, X_test) == y_test))

# Aplicar SMOTE
X_smote, y_smote = smote(X, y, minority_class)

X_train_smote, y_train_smote, X_test_smote, y_test_smote = hold_out_validation(X_smote, y_smote)
print("Hold-Out Accuracy (After SMOTE):", np.mean(euclidean_classifier(X_train_smote, y_train_smote, X_test_smote) == y_test_smote))

# Validación Cruzada
print("10-Fold CV Accuracy (Original Data):", k_fold_cross_validation(X, y))
print("10-Fold CV Accuracy (After SMOTE):", k_fold_cross_validation(X_smote, y_smote))
```

```python
# Parte II - Implementación del Perceptrón Simple

# Cargar dataset Iris (Setosa y Virginica)
def load_iris_binary():
    # Datos manuales para Setosa y Virginica
    from sklearn.datasets import load_iris
    iris = load_iris()
    X, y = iris.data, iris.target

    # Filtrar solo Setosa (0) y Virginica (1)
    filter_indices = (y == 0) | (y == 1)
    return X[filter_indices], y[filter_indices]

# Perceptrón Simple
def perceptron(X, y, epochs=1000, lr=0.01):
    n_samples, n_features = X.shape
    weights = np.zeros(n_features)  # Inicializar pesos
    bias = 0                        # Inicializar sesgo

    for epoch in range(epochs):
        for i in range(n_samples):
            # Salida lineal
            linear_output = np.dot(X[i], weights) + bias
            # Predicción
            y_pred = 1 if linear_output > 0 else 0
            # Actualización de pesos y sesgo
```

```
            update = lr * (y[i] - y_pred)
            weights += update * X[i]
            bias += update
    return weights, bias

# Predicción con el Perceptrón
def perceptron_predict(X, weights, bias):
    linear_output = np.dot(X, weights) + bias
    return (linear_output > 0).astype(int)

# Validación Hold-Out 70/30
def hold_out_validation(X, y, test_ratio=0.3):
    n_samples = X.shape[0]
    indices = np.arange(n_samples)
    np.random.shuffle(indices)
    split_point = int(n_samples * (1 - test_ratio))
    train_indices, test_indices = indices[:split_point], indices[split_point:]
    return X[train_indices], y[train_indices], X[test_indices], y[test_indices]

# Programa principal
# Cargar datos
X, y = load_iris_binary()

# Normalización de características (opcional pero recomendado)
X = (X - X.mean(axis=0)) / X.std(axis=0)
```

```
# Dividir datos en entrenamiento y prueba (70/30)
X_train, y_train, X_test, y_test = hold_out_validation(X, y)

# Entrenar el perceptrón
weights, bias = perceptron(X_train, y_train)

# Evaluar el modelo en los datos de prueba
predictions = perceptron_predict(X_test, weights, bias)

# Calcular precisión
accuracy = np.mean(predictions == y_test)
print(f"Hold-Out Accuracy (Perceptrón): {accuracy:.2f}")
```

```
Hold-Out Accuracy (Original Data): 0.7076923076923077
Hold-Out Accuracy (After SMOTE): 0.6716417910447762
10-Fold CV Accuracy (Original Data): 0.7285714285714285
10-Fold CV Accuracy (After SMOTE): 0.740909090909091
```

```
Hold-Out Accuracy (Perceptrón): 1.00
```