fg-smote

June 5, 2024

1 FG_SMOTE ALGORITHM

Algorithm contains

```
1. Levenshtein Distance Calculation:
     Calculates the distance between two strings.

2. Gaussian SMOTE:
     Generates synthetic samples for the minority class using Gaussian noise.

3. Fuzzy-Based Gaussian SMOTE:
     Filters synthetic samples based on Levenshtein distance.
```

1.0.1 Implementation of Levenshtein Distance Calculation

The Levenshtein distance is a metric for measuring the difference between two sequences.

[1]:

```python
import numpy as np

def levenshtein_distance(s1, s2):
    len_s1, len_s2 = len(s1), len(s2)
    matrix = np.zeros((len_s1 + 1, len_s2 + 1))

    for i in range(len_s1 + 1):
        matrix[i][0] = i
    for j in range(len_s2 + 1):
        matrix[0][j] = j

    for i in range(1, len_s1 + 1):
        for j in range(1, len_s2 + 1):
            if s1[i - 1] == s2[j - 1]:
                cost = 0
            else:
                cost = 1
            matrix[i][j] = min(matrix[i - 1][j] + 1,      # Deletion
                               matrix[i][j - 1] + 1,      # Insertion
                               matrix[i - 1][j - 1] + cost) # Substitution
    return matrix[len_s1][len_s2]
```

1.0.2 Implementation of Gaussian SMOTE

Gaussian SMOTE is used to generate synthetic samples in the feature space and After generating synthetic samples using Gaussian SMOTE, the fuzzy-based filtering step aims to refine the quality of these sample.

[12]:

```
from sklearn.neighbors import NearestNeighbors
from sklearn.utils import resample
import numpy as np

def gaussian_smote(X, y, minority_class, N=100, k=5):
    # Extract the minority class samples
    X_minority = X[y == minority_class]

    # Fit the Nearest Neighbors model
    nn = NearestNeighbors(n_neighbors=k)
    nn.fit(X_minority)

    # Generate synthetic samples
    synthetic_samples = []
    for _ in range(N):
        # Randomly select a minority class sample
        sample_idx = np.random.randint(0, len(X_minority))
        sample = X_minority[sample_idx]

        # Find its nearest neighbors
        neighbors = nn.kneighbors([sample], return_distance=False)[0]

        # Randomly select one of the neighbors
        neighbor_idx = np.random.choice(neighbors)
        neighbor = X_minority[neighbor_idx]

        # Generate a synthetic sample
        diff = neighbor - sample
        synthetic_sample = sample + diff * np.random.rand() + np.random.
 ↪normal(0, 0.1, size=sample.shape)
        synthetic_samples.append(synthetic_sample)

    return np.array(synthetic_samples)

def fuzzy_based_gaussian_smote(X, y, minority_class, threshold=0.5, N=100, k=5):
    # Step 1: Generate synthetic samples using Gaussian SMOTE
    synthetic_samples = gaussian_smote(X, y, minority_class, N, k)

    # Combine original and synthetic samples
    X_combined = np.vstack((X, synthetic_samples))
    y_combined = np.hstack((y, [minority_class] * len(synthetic_samples)))
    # Step 2-5: Apply Levenshtein distance to filter samples
    filtered_samples = []
    for sample in synthetic_samples:
        # Find nearest neighbor in the original minority class samples
        distances = [levenshtein_distance(sample, x) for x in X[y ==␣
 ↪minority_class]]
        nearest_distance = min(distances)

        # Filter based on threshold
        if nearest_distance > threshold:
            filtered_samples.append(sample)

        X_res = np.vstack([X, np.array(filtered_samples)])
        y_res = np.hstack([y, np.full(len(filtered_samples), minority_class)])

    return X_res, y_res


# Usage example
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1000, n_features=20, n_informative=2,␣
 ↪n_redundant=10,
                           n_clusters_per_class=1, weights=[0.9, 0.1],␣
 ↪flip_y=0, random_state=42)

minority_class = 1
synthetic_samples = fuzzy_based_gaussian_smote(X, y, minority_class)

# Define the total number of synthetic samples to generate
# Apply FG-SMOTE
minority_class = 1
X_res, y_res = fuzzy_based_gaussian_smote(X, y, minority_class=1 , N=10)   #(e.
 ↪g., make minority class 1:1 balanced)

minority_count = np.sum(y == minority_class)
print("Number of minority class samples in original dataset:", minority_count)

print("Original dataset shape:", X.shape, y.shape)
print("Resampled dataset shape:", X_res.shape, y_res.shape)

print(f"Number of synthetic samples generated: {len(X_res) - len(X)}")
print("Number of samples in minority class after resampling:", np.sum(y_res ==␣
 ↪minority_class))
```

```
Number of minority class samples in original dataset: 100
Original dataset shape: (1000, 20) (1000,)
Resampled dataset shape: (1010, 20) (1010,)
```

```
Number of synthetic samples generated: 10
Number of samples in minority class after resampling: 110
```