Experiment – 7

Handling Imbalanced Data: Explore techniques to address imbalanced datasets, such as oversampling, undersampling, or using synthetic data generation methods.

Date Set: *This dataset represents fictional customer data and includes the following columns:*

- ***CustomerID:*** *Unique identifier for each customer.*
- ***Age:*** *Age of the customer.*
- ***Tenure:*** *Number of years the customer has been with the company.*
- ***Products:*** *Number of products owned by the customer.*
- ***Purchased:*** *Target variable indicating whether the customer churned (Yes) or not (No).*

Objective:

Explore techniques to address imbalanced datasets.

Tasks:

1. Identify and visualize class imbalances in the dataset.
2. Experiment with oversampling, undersampling, and synthetic data generation methods.
3. Evaluate the impact of imbalanced data handling on model performance.

Handling Imbalanced Data is crucial in machine learning, especially in classification tasks where the number of instances in each class is disproportionate. Imbalanced datasets can lead to biased models that favor the majority class and perform poorly on the minority class. Therefore, it's essential to employ techniques to address this imbalance to improve model performance and accuracy.

Here are some common techniques for handling imbalanced datasets:

1. **Oversampling:** Oversampling involves increasing the number of instances in the minority class to balance the class distribution. This can be achieved by randomly duplicating instances from the minority class or by generating synthetic instances using techniques like Synthetic Minority Over-sampling Technique (SMOTE). Oversampling helps provide more training data for the minority class, which can improve the model's ability to learn from it.
2. **Undersampling:** Undersampling involves reducing the number of instances in the majority class to balance the class distribution. This can be done by randomly removing instances from the majority class until the class distribution is balanced. While undersampling is simpler and faster than oversampling, it may lead to loss of important information from the majority class.
3. **Synthetic Data Generation:** Synthetic data generation techniques, such as SMOTE, ADASYN (Adaptive Synthetic Sampling), and Borderline-SMOTE, create synthetic instances for the minority class by interpolating between existing instances. These techniques generate new instances by considering the feature space of existing minority class instances, which helps preserve the underlying characteristics of the data.
4. **Class Weighting:** Many machine learning algorithms allow for assigning different weights to classes during training. By assigning higher weights to the minority class

and lower weights to the majority class, the algorithm pays more attention to the minority class during training, effectively mitigating the imbalance.

5. **Ensemble Methods:** Ensemble methods, such as Random Forests and Gradient Boosting Machines, can be effective for imbalanced datasets. These methods combine multiple models to make predictions, and they can handle class imbalance by weighting the predictions from each model based on their performance on different classes.
6. **Cost-sensitive Learning:** Cost-sensitive learning involves modifying the algorithm's objective function to penalize misclassifications of the minority class more heavily than misclassifications of the majority class. This approach explicitly accounts for the imbalance in the class distribution and adjusts the model's learning process accordingly.

Python Program:

Importing the libraries

```
import pandas as pd
```

A. **import pandas as pd**:
   a. **pandas** is a data manipulation and analysis library for Python.
   b. **import pandas as pd** imports the **pandas** library and gives it an alias **pd** for convenience.
   c. The **pd** alias is commonly used to reference functions and objects from the **pandas** library. For example, **pd.DataFrame()** is used to create a DataFrame.

Importing the dataset

```
# Load the dataset
df = pd.read_csv(r'C:\Users\KUNAL
KUNDU\OneDrive\Desktop\GHRCEM\...\Exp-7\Experiment-7.csv')

# Display the first few rows of the dataset
print(df.head())
```

Note:

- **print(df.head())**: This line prints the first few rows of the DataFrame **df** using the **head()** function. By default, **head()** displays the first 5 rows of the DataFrame. This is useful for quickly inspecting the structure and content of the dataset.

Output:

```
   CustomerID  Age  Tenure  Products Purchased
0           1   35       5         1        No
1           2   42       8         3       Yes
2           3   28       3         2        No
3           4   55      15         1        No
4           5   39       7         2       Yes
```

Oversampling the minority class using Synthetic Minority Over-sampling Technique (SMOTE)

```
from imblearn.over_sampling import SMOTE

# Separate features and target variable
X = df.drop(columns=['Purchased'])
y = df['Purchased']

# Apply SMOTE for oversampling
smote = SMOTE()
X_resampled, y_resampled = smote.fit_resample(X, y)

# Display the class distribution after oversampling
print(y_resampled.value_counts())
```

Note:
This part of the code deals with oversampling the minority class using the Synthetic Minority Over-sampling Technique (SMOTE). Here's a detailed breakdown:

1. **from imblearn.over_sampling import SMOTE**: This line imports the SMOTE class from the imbalanced-learn library. SMOTE is a technique used to address class imbalance by generating synthetic samples from the minority class.
2. **X = df.drop(columns=['Purchased'])** and **y = df['Purchased']**: These lines separate the features (X) and the target variable (y) from the DataFrame **df**. The target variable here is **'Purchased'**, which indicates whether a customer has made a purchase or not.
3. **smote = SMOTE()**: This line initializes an instance of the SMOTE class.
4. **X_resampled, y_resampled = smote.fit_resample(X, y)**: This line applies the SMOTE technique to the dataset. The **fit_resample()** method is used to perform oversampling. It takes the features **X** and the target variable **y** as input and returns the resampled feature matrix **X_resampled** and the corresponding resampled target vector **y_resampled**.
5. **print(y_resampled.value_counts())**: This line prints the class distribution of the target variable **y_resampled** after oversampling. The **value_counts()** method counts the occurrences of each unique value in the target variable, which helps in evaluating whether the classes are balanced after oversampling.

Output:

```
No     11
Yes    11
Name: Purchased, dtype: int64
```

Undersampling the majority class using RandomUnderSampler

```
from imblearn.under_sampling import RandomUnderSampler

# Apply RandomUnderSampler for undersampling
undersampler = RandomUnderSampler()
X_resampled, y_resampled = undersampler.fit_resample(X, y)
```

```
# Display the class distribution after undersampling
print(y_resampled.value_counts())
```

Note:
This part of the code is focused on undersampling the majority class using the RandomUnderSampler technique. Here's a detailed breakdown:

1. **from imblearn.under_sampling import RandomUnderSampler**: This line imports the RandomUnderSampler class from the imbalanced-learn library. RandomUnderSampler is a technique used to randomly remove samples from the majority class to balance the class distribution.
2. **undersampler = RandomUnderSampler()**: Here, an instance of the RandomUnderSampler class is created. It will be used to perform undersampling on the dataset.
3. **X_resampled, y_resampled = undersampler.fit_resample(X, y)**: This line applies the RandomUnderSampler technique to the dataset. The **fit_resample()** method is called on the undersampler instance, with the features **X** and the target variable **y** as inputs. It returns the resampled feature matrix **X_resampled** and the corresponding resampled target vector **y_resampled**, where the majority class has been undersampled.
4. **print(y_resampled.value_counts())**: Finally, this line prints the class distribution of the resampled target variable **y_resampled**. The **value_counts()** method counts the occurrences of each unique value in the target variable, showing the class distribution after undersampling. This is important to verify whether the classes are now balanced or not.

Output:

```
No     9
Yes    9
Name: Purchased, dtype: int64
```

Generating synthetic samples using Synthetic Minority Over-sampling Technique (SMOTE) and Edited Nearest Neighbors (ENN)

```
from imblearn.combine import SMOTEENN

# Apply SMOTEENN for generating synthetic samples
smote_enn = SMOTEENN()
X_resampled, y_resampled = smote_enn.fit_resample(X, y)

# Display the class distribution after generating synthetic
samples
print(y_resampled.value_counts())
```

Note:
This part of the code focuses on generating synthetic samples using the SMOTEENN technique. Here's a detailed breakdown:

1. **from imblearn.combine import SMOTEENN**: This line imports the SMOTEENN class from the imbalanced-learn library. SMOTEENN is a combination of over-sampling using SMOTE and under-sampling using Edited Nearest Neighbors (ENN).

It generates synthetic samples for the minority class and removes noisy samples from both classes.

2. **smote_enn = SMOTEENN()**: Here, an instance of the SMOTEENN class is created. This instance will be used to apply the SMOTEENN technique to the dataset.
3. **X_resampled, y_resampled = smote_enn.fit_resample(X, y)**: This line applies the SMOTEENN technique to the dataset. The **fit_resample()** method is called on the **smote_enn** instance, with the features **X** and the target variable **y** as inputs. It returns the resampled feature matrix **X_resampled** and the corresponding resampled target vector **y_resampled**, where synthetic samples have been generated to address the class imbalance.
4. **print(y_resampled.value_counts())**: Finally, this line prints the class distribution of the resampled target variable **y_resampled**. The **value_counts()** method counts the occurrences of each unique value in the target variable, showing the class distribution after generating synthetic samples. This is important to verify whether the classes are now balanced or not after applying the SMOTEENN technique.

Output:

```
No     7
Yes    7
Name: Purchased, dtype: int64
```