Data Analysis Report: Missing Values and Imbalanced Datasets

This report covers two important topics in data analysis: missing values in datasets and techniques for handling imbalanced datasets. We will explore examples using the Titanic dataset and synthetic datasets.

1. Missing Values in Datasets

Missing values occur in datasets when some information is not stored for a variable. There are three main mechanisms:

1.1 Missing Completely at Random (MCAR)

Missing completely at random (MCAR) is a type of missing data mechanism in which the probability of a value being missing is unrelated to both the observed data and the missing data. In other words, if the data is MCAR, the missing values are randomly distributed throughout the dataset, and there is no systematic reason for why they are missing.For example, in a survey about the prevalence of a certain disease, the missing data might be MCAR if the survey participants with missing values for certain questions were selected randomly and their missing responses are not related to their disease status or any other variables measured in the survey.

1.2 Missing at Random (MAR)

Missing at Random (MAR) is a type of missing data mechanism in which the probability of a value being missing depends only on the observed data, but not on the missing data itself. In other words, if the data is MAR, the missing values are systematically related to the observed data, but not to the missing data. Here are a few examples of missing at random:Income data: Suppose you are collecting income data from a group of people, but some participants choose not to report their income. If the decision to report or not report income is related to the participant's age or gender, but not to their income level, then the data is missing at random.Medical data: Suppose you are collecting medical data on patients, including their blood pressure, but some patients do not report their blood pressure. If the patients who do not report their blood pressure are more likely to be younger or have healthier lifestyles, but the missingness is not related to their actual blood pressure values, then the data is missing at random.

1.3 Missing Not at Random (MNAR)

It is a type of missing data mechanism where the probability of missing values depends on the value of the missing data itself. In other words, if the data is MNAR, the missingness is not random and is dependent on unobserved or unmeasured factors that are associated with the missing values.For example, suppose you are collecting data on the income and job satisfaction of employees in a company. If employees who are less satisfied with their jobs are more likely to refuse to report their income, then the data is not missing at random. In this case, the missingness is dependent on job satisfaction, which is not directly observed or measured.

2. Example with Titanic Dataset

We will use the Titanic dataset to demonstrate how to handle missing values.

Code Snippet: Load Dataset

```
import seaborn as sns
df = sns.load_dataset('titanic')
df.head()
```

This code snippet loads the Titanic dataset using the seaborn library and displays the first few rows.

Code Snippet: Check for Missing Values

```
df.isnull().sum()
```

This code checks for missing values in each column of the dataset.

Code Snippet: Delete Rows with Missing Values

```
df.dropna().shape
```

This code snippet deletes rows with any missing values and shows the new shape of the dataset.

3. Imputation Techniques

We can handle missing values using various imputation techniques:

3.1 Mean Value Imputation

```
df['age_mean'] = df['age'].fillna(df['age'].mean())
```

This code fills missing values in the 'age' column with the mean age.

3.2 Median Value Imputation

```
df['age_median'] = df['age'].fillna(df['age'].median())
```

This code fills missing values in the 'age' column with the median age, which is useful in the presence of outliers.

3.3 Mode Imputation for Categorical Values

```
df['embarked'].unique()

mode_value = df[df['embarked'].notna()]['embarked'].mode()[0]

df['embarked_mode'] = df['embarked'].fillna(mode_value)
```

This code fills missing values in the 'embarked' column with the mode (most frequent value).

4. Handling Imbalanced Datasets

Imbalanced datasets can lead to biased models. We will explore techniques for handling such datasets, specifically focusing on upsampling and downsampling methods.

4.1 Creating an Imbalanced Dataset

We start by creating a synthetic dataset with two classes, where one class is significantly larger than the other.

Code Snippet: Create Imbalanced Dataset

```
import numpy as np
import pandas as pd

# Set the random seed for reproductivity
np.random.seed(123)

# Create a dataframe with two classes
n_samples = 1000
class_0_ratio = 0.9
n_class_0 = int(n_samples * class_0_ratio)
n_class_1 = n_samples - n_class_0

class_0 = pd.DataFrame({
    'feature_1' : np.random.normal(loc=0, scale=1, size=n_class_0),
    'feature_2' : np.random.normal(loc=0, scale=1, size=n_class_0),
    'target' : [0] * n_class_0
})

class_1 = pd.DataFrame({
    'feature_1': np.random.normal(loc=2, scale=1, size=n_class_1),
    'feature_2': np.random.normal(loc=2, scale=1, size=n_class_1),
    'target': [1] * n_class_1
})

df = pd.concat([class_0, class_1]).reset_index(drop=True)
```

In this code, we create a dataset with 1000 samples, where 900 belong to class 0 and 100 belong to class 1. The features are generated using a normal distribution.

4.2 Checking Class Distribution

We can check the distribution of the classes in our dataset.

Code Snippet: Check Class Distribution

```
df['target'].value_counts()
```

This code snippet shows the distribution of the target classes in the dataset, confirming the imbalance.

4.3 Upsampling

Upsampling involves increasing the number of instances in the minority class to balance the dataset.

Code Snippet: Upsampling

```
from sklearn.utils import resample

df_minority = df[df['target'] == 1]
df_majority = df[df['target'] == 0]

df_minority_upsampled = resample(df_minority, replace=True,
                                 n_samples=len(df_majority),
                                 random_state=42)

df_upsampled = pd.concat([df_majority, df_minority_upsampled])
df_upsampled['target'].value_counts()
```

In this code, we use the `resample` function from `sklearn.utils` to upsample the minority class. We then concatenate the upsampled minority class with the majority class to create a balanced dataset.

4.4 Downsampling

Downsampling involves reducing the number of instances in the majority class to balance the dataset.

Code Snippet: Downsampling

```
df_majority_downsampled = resample(df_majority, replace=False,
                                   n_samples=len(df_minority),
                                   random_state=42)

df_downsampled = pd.concat([df_minority, df_majority_downsampled])
df_downsampled['target'].value_counts()
```

This code snippet demonstrates how to downsample the majority class. We again use the `resample` function, but this time we set `replace=False` to ensure we are not sampling with replacement.

5. Conclusion

Handling missing values and imbalanced datasets is crucial for effective data analysis and model training. Understanding the mechanisms behind missing data and applying appropriate techniques for imbalanced datasets can significantly improve the quality of the analysis.

Glossary

Missing Values: Data entries that are not recorded or are absent in a dataset.

Imbalanced Dataset: A dataset where the classes are not represented equally, leading to potential bias in model training.

Upsampling: A technique used to increase the number of instances in the minority class to achieve a balanced dataset.

Downsampling: A method that reduces the number of instances in the majority class to balance the dataset.

Synthetic Dataset: A dataset created artificially, often used for testing and validation purposes.

Resampling: The process of randomly selecting samples from a dataset to create a new dataset, which can be done with or without replacement.

Class Distribution: The proportion of different classes within a dataset, which is crucial for understanding the balance of the dataset.

SMOTE (Synthetic Minority Over-sampling Technique)

This report covers the SMOTE technique, which is used in machine learning to address imbalanced datasets. SMOTE generates synthetic instances of the minority class by interpolating between existing instances.

1. Generating a Sample Dataset

We will first create a synthetic dataset using the `make_classification` function from the `sklearn.datasets` module.

Code Snippet: Generate Sample Dataset

```
from sklearn.datasets import make_classification
X, Y = make_classification(n_samples=1000, n_redundant=0, n_features=2,
                           n_clusters_per_class=1, weights=[0.90],
                           random_state=12)
import pandas as pd
df1 = pd.DataFrame(X, columns=['f1', 'f2'])
df2 = pd.DataFrame(Y, columns=['target'])
final_df = pd.concat([df1, df2], axis=1)
final_df.head()
```

In this code, we generate a dataset with 1000 samples, where 90% belong to one class (majority) and 10% to another (minority). The features are stored in a DataFrame, and we can visualize the first few rows using `head()`.

2. Visualizing the Dataset

Next, we visualize the dataset to understand the class distribution.

Code Snippet: Visualize Dataset

```
import matplotlib.pyplot as plt
```

```
plt.scatter(final_df['f1'], final_df['f2'], c=final_df['target'])
plt.title('Original Dataset Visualization')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

The scatter plot above shows the distribution of the two classes in the feature space. The majority class (0) is represented by one color, while the minority class (1) is represented by another.

3. Applying SMOTE

To address the class imbalance, we will apply the SMOTE technique using the `imblearn` library.

Code Snippet: Apply SMOTE

```
!pip install imblearn
from imblearn.over_sampling import SMOTE

# Transform the dataset
oversample = SMOTE()
X, Y = oversample.fit_resample(final_df[['f1', 'f2']], final_df['target'])
X.shape, Y.shape
```

In this code, we first install the `imblearn` library if it is not already installed. Then, we apply SMOTE to the dataset, which generates synthetic samples for the minority class, resulting in a balanced dataset with equal instances of both classes.

4. Visualizing the Oversampled Dataset

After applying SMOTE, we can visualize the new dataset to confirm that the classes are now balanced.

Code Snippet: Visualize Oversampled Dataset

```
df1 = pd.DataFrame(X, columns=['f1', 'f2'])
df2 = pd.DataFrame(Y, columns=['target'])
oversample_df = pd.concat([df1, df2], axis=1)
plt.scatter(oversample_df['f1'], oversample_df['f2'], c=oversample_df['target'])
plt.title('Oversampled Dataset Visualization')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

The scatter plot above illustrates the oversampled dataset, where both classes are now represented equally. This balance is crucial for training machine learning models effectively.

Conclusion

In this report, we explored the SMOTE technique for handling imbalanced datasets. We generated a synthetic dataset, visualized it, applied SMOTE to balance the classes, and visualized the results. This technique is essential for improving the performance of machine learning models on imbalanced data.

SMOTE Glossary

SMOTE: Synthetic Minority Over-sampling Technique, a method to generate synthetic samples for the minority class in imbalanced datasets.

Synthetic Sample: An artificially created instance that is generated based on existing instances in the minority class.

Interpolation: A method used in SMOTE to create new samples by combining features of existing samples.

Minority Class: The class in a dataset that has fewer instances compared to the majority class.

Majority Class: The class in a dataset that has more instances compared to the minority class.

Five Number Summary and Box Plot Report

This report covers the five-number summary and box plots for a given dataset of marks.

Five Number Summary

```
import numpy as np
lst_marks = [45, 32, 56, 75, 89, 54, 32, 89, 90, 87, 67, 54, 45, 98, 99, 67, 74]
minimum, Q1, median, Q3, maximum = np.quantile(lst_marks, [0, 0.25, 0.50, 0.75, 1.0])
IQR = Q3 - Q1
lower_fence = Q1 - 1.5 * IQR
higher_fence = Q3 + 1.5 * IQR
```

The five-number summary consists of the minimum, first quartile (Q1), median, third quartile (Q3), and maximum values of the dataset. These statistics provide a quick overview of the distribution of the data.

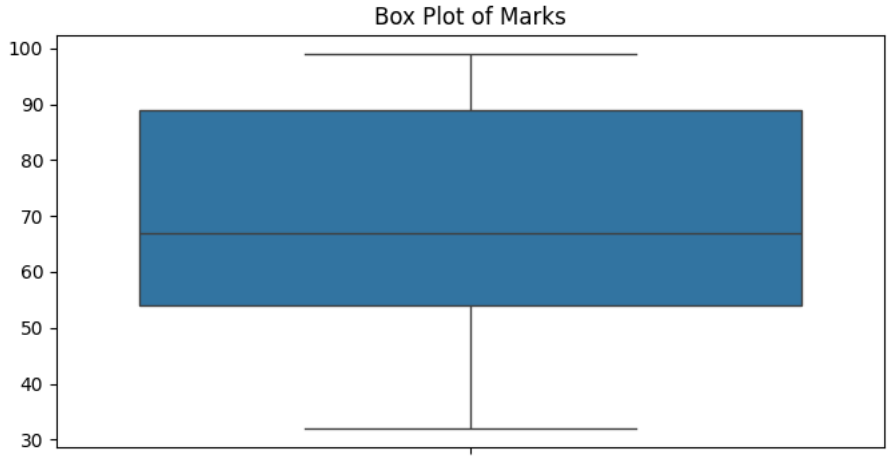
The five-number summary for the dataset is as follows:
Minimum: 32.0
Q1: 54.0
Median: 67.0
Q3: 89.0
Maximum: 99.0

Box Plot

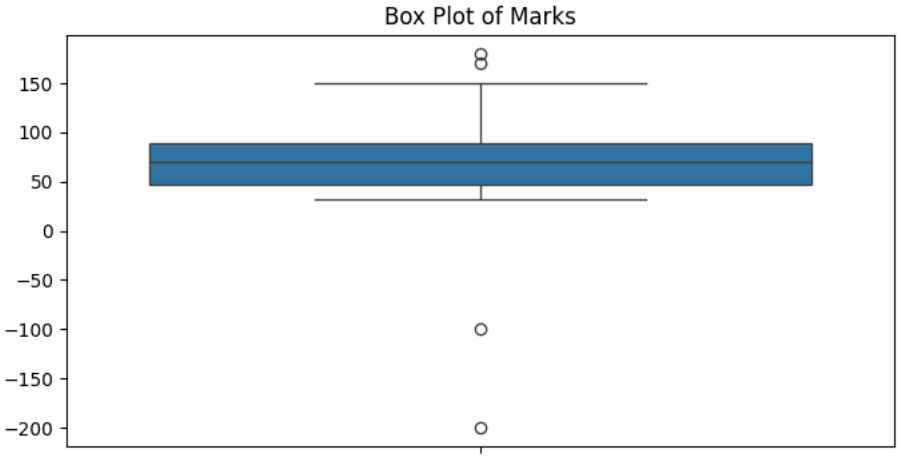
```
import seaborn as sns
lst_marks_with_outliers = [-100, -200, 45, 32, 56, 75, 89, 54, 32, 89, 90,
                           87, 67, 54, 45, 98, 99, 67, 74, 150, 170, 180]
sns.boxplot(lst_marks_with_outliers)
```

A box plot visually represents the five-number summary of the dataset. It displays the minimum, Q1, median, Q3, and maximum values, along with any potential outliers. The box represents the interquartile range (IQR), and the line inside the box indicates the median.

Box Plot of Marks (without outliers):



Box Plot of Marks (with outliers):



Conclusion

This report provided an overview of the five-number summary and box plots for a dataset of marks. These tools are essential for understanding the distribution and identifying outliers in the data.

Glossary

1. Five Number Summary: A descriptive statistic that provides information about a dataset's minimum, first quartile (Q1), median, third quartile (Q3), and maximum.
2. Box Plot: A graphical representation of the five-number summary that displays the distribution of data based on a five-number summary.
3. Outlier: A data point that differs significantly from other observations in the dataset.
4. Interquartile Range (IQR): A measure of statistical dispersion, calculated as the difference between the third quartile (Q3) and the first quartile (Q1).

Data Encoding Techniques in Machine Learning

Introduction

Data encoding is a crucial step in preparing categorical data for machine learning algorithms. This report covers various encoding techniques including One Hot Encoding, Label Encoding, Ordinal Encoding, and Target Guided Ordinal Encoding.

One Hot Encoding

One Hot Encoding is a technique used to convert categorical variables into a format that can be provided to ML algorithms to do a better job in prediction.

Code Snippet:

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

# Create a simple dataframe
df = pd.DataFrame({
    'color' : ['red', 'blue', 'green', 'green', 'red', 'blue']
})

# Create an instance of OneHotEncoder
encoder = OneHotEncoder()

# Perform fit and transform
encoded = encoder.fit_transform(df[['color']]).toarray()

# Create DataFrame from the encoded array
encoder_df = pd.DataFrame(encoded, columns=encoder.get_feature_names_out())

# Concatenate the original DataFrame with the encoded DataFrame
result_df = pd.concat([df, encoder_df], axis=1)
```

In this code, we create a DataFrame with a categorical variable 'color'. We then instantiate the OneHotEncoder and fit it to the 'color' column, transforming it into a binary array. Next, we create a new DataFrame from the encoded array and concatenate it with the original DataFrame.

Example with Seaborn Dataset

We can also apply One Hot Encoding to more complex datasets, such as the 'tips' dataset from Seaborn.

Code Snippet:

```
import seaborn as sns

# Load the tips dataset
df = sns.load_dataset('tips')

# Apply One Hot Encoding to the 'sex' column
encoder = OneHotEncoder()
encoded = encoder.fit_transform(df[['sex']]).toarray()
```

```
# Create DataFrame from the encoded array
encoder_df = pd.DataFrame(encoded, columns=encoder.get_feature_names_out())

# Concatenate the original DataFrame with the encoded DataFrame
result_df = pd.concat([df, encoder_df], axis=1)
```

In this example, we load the 'tips' dataset and apply One Hot Encoding to the 'sex' column, which converts the categorical values into a binary format suitable for machine learning. We then create a new DataFrame from the encoded array and concatenate it with the original DataFrame.

Conclusion

Data encoding is essential for preparing categorical data for machine learning. One Hot Encoding is one of the most commonly used techniques, allowing algorithms to interpret categorical variables effectively.

Glossary

One Hot Encoding: A technique to convert categorical variables into a binary matrix, where each category is represented as a binary vector.

Categorical Variable: A variable that can take on one of a limited, and usually fixed, number of possible values, assigning each individual or other unit of observation to a particular group.

DataFrame: A two-dimensional, size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns) provided by the pandas library.

Machine Learning: A field of artificial intelligence that uses statistical techniques to give computer systems the ability to 'learn' from data, improving their performance on a specific task over time.

Encoder: A component that transforms categorical data into a numerical format that can be used by machine learning algorithms.

Categorical Data Encoding Techniques

This report covers two important techniques for encoding categorical data: Label Encoding and Ordinal Encoding. These techniques are essential for preprocessing data for machine learning models.

1. Label Encoding

Label encoding is a technique used to convert categorical variables into numerical values. Each category is assigned a unique integer based on its order. Below is an example of label encoding for a categorical variable 'color'.

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Create a sample dataframe
df = pd.DataFrame({
    'color' : ['red', 'blue', 'green', 'green', 'red', 'blue']
})

# Initialize LabelEncoder
lbl_encoder = LabelEncoder()

# Fit and transform the data
encoded_labels = lbl_encoder.fit_transform(df['color'])
print(encoded_labels)

# Transforming individual labels
print(lbl_encoder.transform(['red']))
print(lbl_encoder.transform(['green']))
print(lbl_encoder.transform(['blue']))
```

In the above code, we first create a DataFrame with a 'color' column. We then initialize the LabelEncoder and use it to fit and transform the 'color' column. The output is an array of encoded labels corresponding to each color. We also demonstrate how to transform individual labels.

2. Ordinal Encoding

Ordinal encoding is used for categorical variables that have a clear order. Each category is assigned a numerical value based on its rank. Below is an example of ordinal encoding for a categorical variable 'size'.

```
from sklearn.preprocessing import OrdinalEncoder

# Create a sample dataframe with an ordinal variable
df = pd.DataFrame({
    'size' : ['small', 'medium', 'large', 'medium', 'small', 'large']
})

# Initialize OrdinalEncoder with specified categories
encoder = OrdinalEncoder(categories=[['small', 'medium', 'large']])
```

```
# Fit and transform the data
encoded_sizes = encoder.fit_transform(df[['size']])
print(encoded_sizes)

# Transforming individual labels
print(encoder.transform([['medium']]))
```

In this code, we create a DataFrame with a 'size' column. We then initialize the OrdinalEncoder with the specified order of categories. The fit_transform method is used to encode the sizes into numerical values based on their order. We also demonstrate how to transform individual labels.

Conclusion

Both label encoding and ordinal encoding are crucial techniques for converting categorical data into a numerical format that can be used in machine learning algorithms. Understanding when to use each method is essential for effective data preprocessing.

Glossary

1. **Label Encoding**: A technique for converting categorical variables into numerical values by assigning a unique integer to each category.

2. **Ordinal Encoding**: A method for encoding categorical variables that have a clear order by assigning numerical values based on their rank.

3. **Categorical Data**: Data that can be divided into specific groups or categories.

4. **Numerical Data**: Data that represents quantities and can be measured.

5. **DataFrame**: A two-dimensional, size -mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns) in pandas.

6. **Machine Learning**: A field of artificial intelligence that uses statistical techniques to give computer systems the ability to learn from data.

Target Guided Ordinal Encoding

This report covers the concept of Target Guided Ordinal Encoding, a technique used to encode categorical variables based on their relationship with the target variable. This encoding technique is particularly useful when dealing with categorical variables that have a large number of unique categories.

1. Sample DataFrame Creation

```
import pandas as pd

# create a sample dataframe with a categorical variable and a target variable
df = pd.DataFrame({
    'city': ['New York', 'London', 'Paris', 'Tokyo', 'New York', 'Paris'],
    'price': [200, 150, 300, 250, 180, 320]
})
df
```

In this section, we create a sample DataFrame with a categorical variable 'city' and a target variable 'price'. This DataFrame will be used to demonstrate the encoding technique.

2. Mean Price Calculation

```
mean_price = df.groupby('city')['price'].mean().to_dict()
mean_price
```

Here, we calculate the mean price for each city using the groupby method and convert it to a dictionary.

3. Encoding the Categorical Variable

```
df['city_encoded'] = df['city'].map(mean_price)
df[['price', 'city_encoded']]
```

In this step, we replace each category in the 'city' column with its corresponding mean price, creating a new column 'city_encoded'. This establishes a monotonic relationship between the categorical variable and the target variable.

4. Example with Seaborn Dataset

```
import seaborn as sns
df = sns.load_dataset('tips')
mean_bill = df.groupby('time')['total_bill'].mean().to_dict()
df['time_encoded'] = df['time'].map(mean_bill)
df[['time', 'time_encoded']]
```

In this section, we use the 'tips' dataset from the seaborn library to demonstrate the encoding technique. We calculate the mean total bill for each time of day and create a new encoded column.

Conclusion

Target Guided Ordinal Encoding is a powerful technique for encoding categorical variables, especially when dealing with high cardinality. By establishing a relationship between the categorical variable and the target variable, we can improve the predictive power of our machine learning models.

Glossary

1. **Target Variable**: The variable that we are trying to predict or explain in a model.

2. **Categorical Variable**: A variable that can take on one of a limited, and usually fixed, number of possible values, assigning each individual or other unit of observation to a particular group or nominal category.

3. **Mean**: The average value of a set of numbers, calculated by dividing the sum of the values by the number of values.

4. **Monotonic Relationship**: A relationship that is either entirely non-increasing or non-decreasing, meaning that as one variable increases, the other variable either only increases or only decreases.

5. **Encoding**: The process of converting categorical data into a numerical format that can be provided to machine learning algorithms.