Bank Marketing Prediction System

Areeb Khemani, Chandrakanth Talakokkula, Shijie Zhou

December 12, 2023

Abstract

This project leverages the Bank-Marketing data set sourced from the UCI Machine Learning Repository, employing advanced machine learning techniques to extract valuable insights. The data set comprises records of marketing campaigns conducted by a bank, with the primary objective of predicting client subscriptions to term deposits based on historical data.

The initial phase of this project involves Exploratory Data Analysis (EDA), a critical step in understanding and preparing the data for modelling. This includes handling missing values and performing data pre-processing to ensure the data set's integrity. Furthermore, feature engineering is conducted to enhance the data set's predictive power, which includes addressing categorical variables and addressing outliers. To facilitate a comprehensive understanding of the data, a variety of data visualization techniques are also employed.

Subsequently, the project delves into constructing predictive models for term deposit subscriptions. A range of machine learning algorithms, including but not limited to logistic regression, decision trees, random forests, and gradient boosting, will be implemented. These models will undergo rigorous training and evaluation.

In summary, this project underscores the pivotal role of data-driven strategies in elevating marketing initiatives within the banking sector. Through the adept application of machine learning techniques, this study will provide actionable insights that can be harnessed for designing more effective marketing campaigns, ultimately leading to heightened client engagement and improved business outcomes.

Explanation: The data is related to marketing campaigns done by a banking institution. Our goal is to predict whether the client will subscribe to the term deposit or not. Our data set has two feature types: Categorical, Integer. Our data set has 16 features which are: Age, job, marital, education, default, balance, housing, loan, contact, day of week, month, duration, campaign, pdays, previous, poutcome. Each of them has their own description:

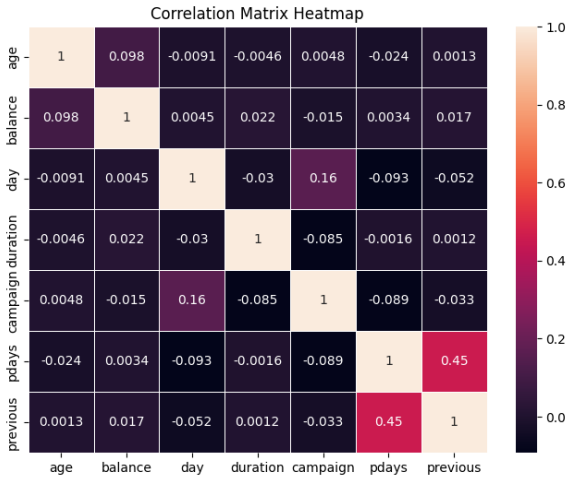
- Age: Age,
- Job: type of job
- Marital: marital status
- education: education background
- default: has credit in default?
- balance: average yearly balance
- housing: has housing loan?
- loan: has personal loan?
- contact: contact communication type
- dayOfWeek: last contact day of the week
- month: last contact month of year
- duration: last contact duration, in seconds
- campaign: number of contact performed during this campaign and for this client
- pdays: number of days that passed by after the client was last contacted from a previous campaign

- previous: number of contacts performed before this campaign and for this client
- poutcome: outcome of the previous marketing campaign
- y: has the client subscribed a term deposit

1 Analysis

We used Pearson Correlation as our first metric to check if our numerical features have any sort of correlation among them. We plotted a heat map to show the correlation matrix and it shows that our features aren't correlated to each other.

As shown in the heatmap, the values are all close to zero. Meaning There is no systematic linear pattern between the variables.

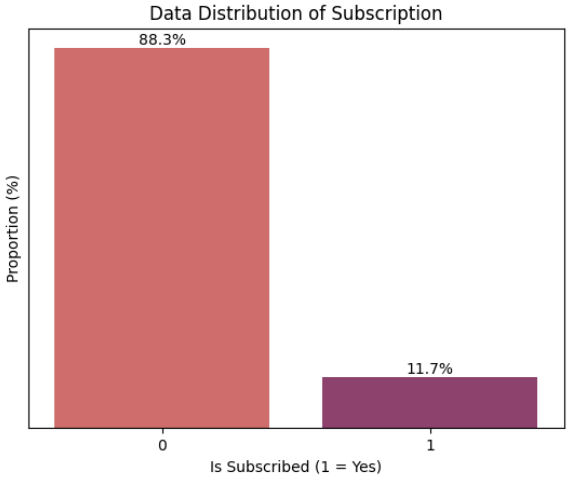


The second metrics we used is that we found the variance of the numerical features. This results show that the age and pdays attribute has variance closer to 1. it tells us that the data points are closer to its mean. While the other numerical attributes have a higher variance, showing that the data points are widely spread out and we could have some outliers in them.

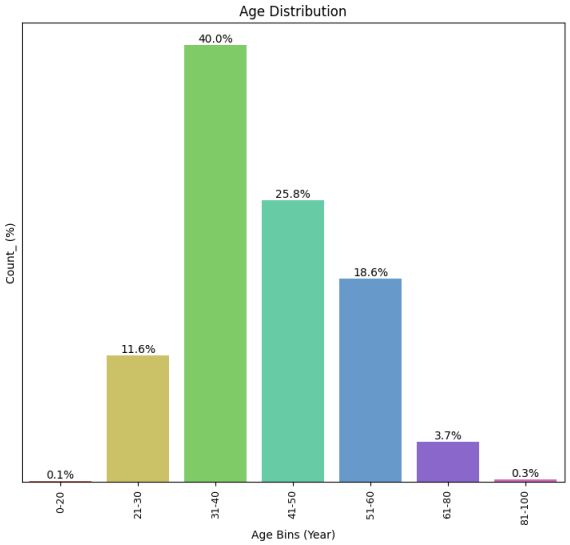
In this analysis, we have managed to calculate the range and mean for the age, balance, day, duration, campaign, pdays, previous and the mode for the job, marital, education, ,default, housing, loan, contact, month and poutcome attribute, it gave us a rough picture of our dataset on what range, mean and mode it will look like overall. It also analysis the categorical attributes as well, identifying the most common categories for "job, martial, education, default, housing, loan, contact, day, month, and poutcome". Afterall, this analysis explained what attributes we are aiming for dive in and it gives us more detailed and more comprehensive views on how is our dataset looks like, providing us a better understanding of the dataset so that we can devote to do the next part job which is visualizing with tables and charts.

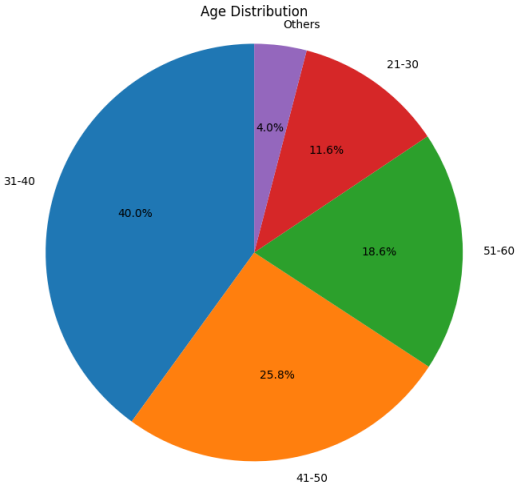
2 Visualisation

First we plotted our target label into a bar graph to look at the target data in our dataset. The graph shows that we have a high imbalance in our data and we will have to take care of it by using undersampling or oversampling.

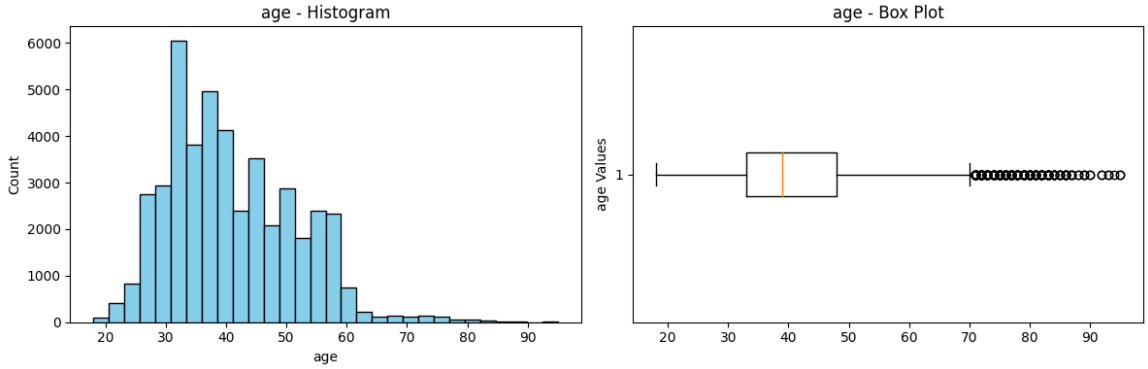


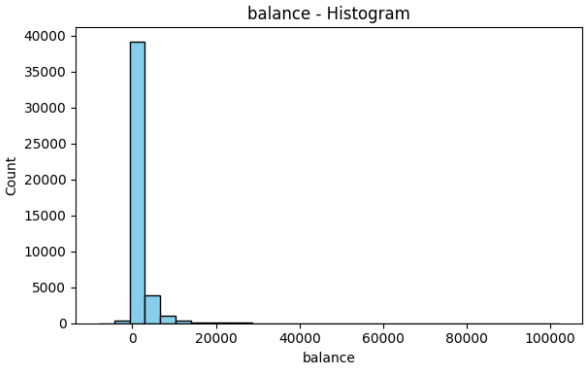
We plotted a pie chart and a bar graph to look at the distribution of the age of the customers. It shows that most of our customers are between the age of 30-40 followed by customers between the age of 41-50.

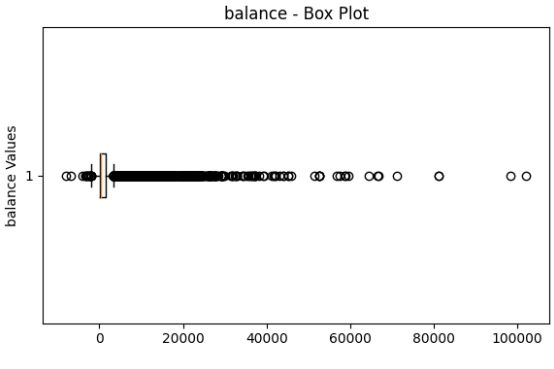


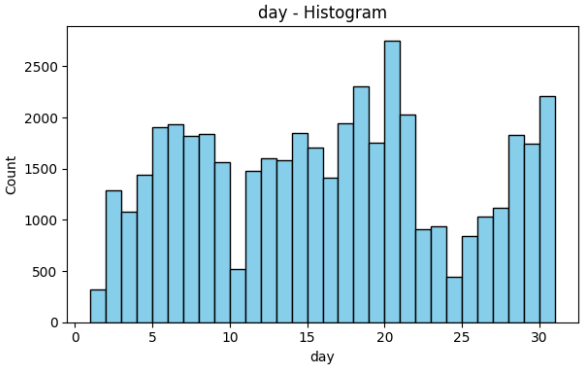


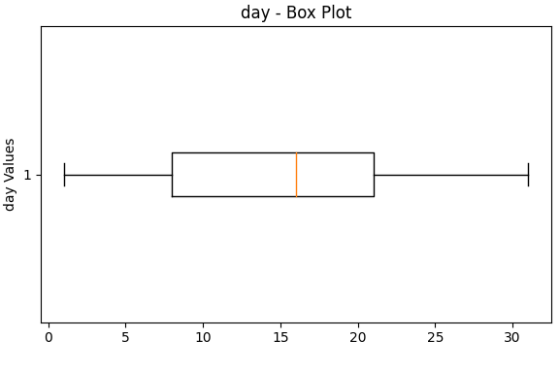
We used a Box and Whisker plot for our numerical attributes to check for outliers and the plot shows that our attributes have outliers in them and we need to take care of it before proceeding towards the model training.

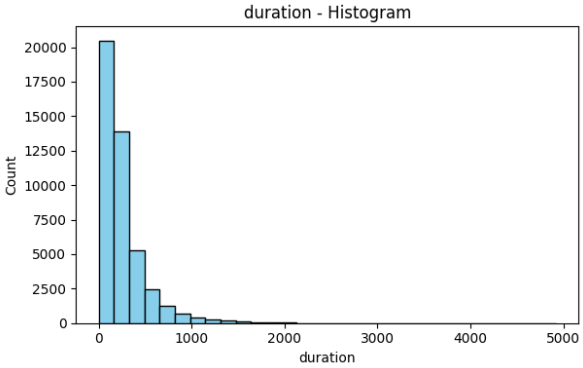


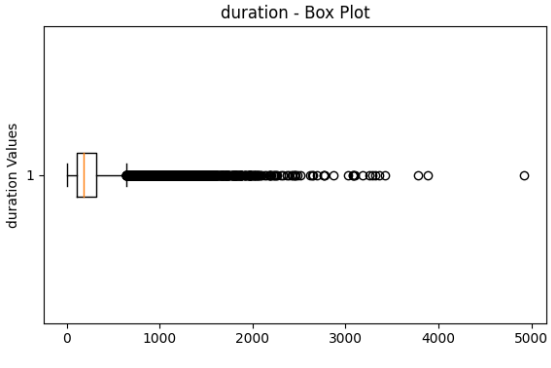

balance - Histogram
Count
balance

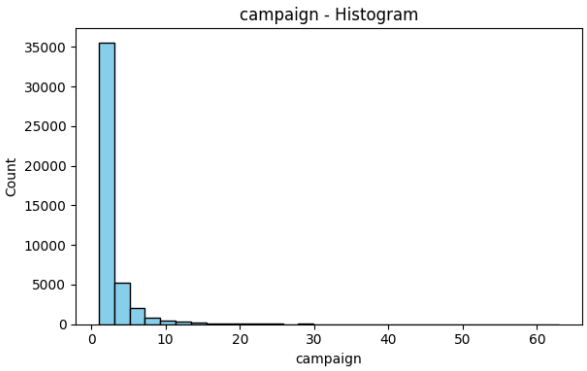

balance - Box Plot
balance Values

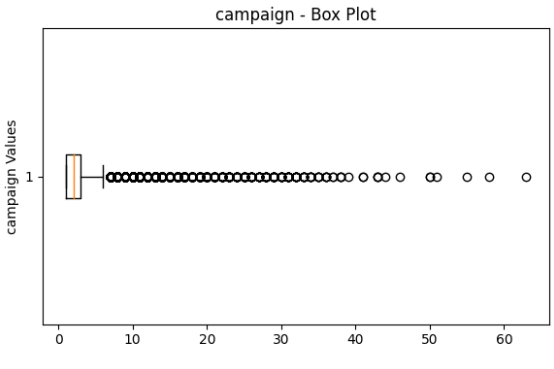

day - Histogram
Count
day

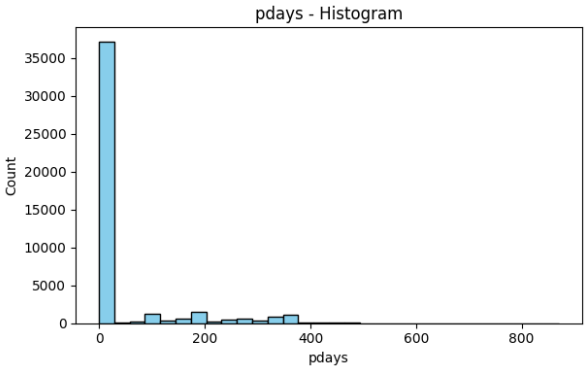

day - Box Plot
day Values

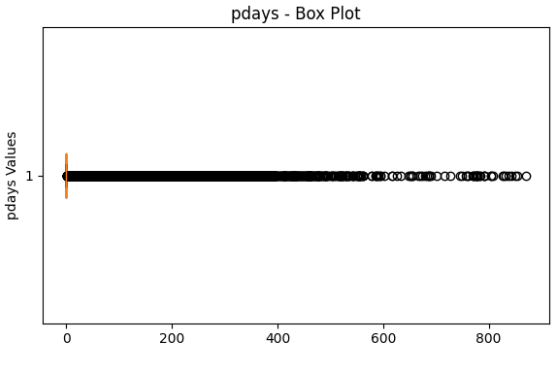

duration - Histogram
Count
duration

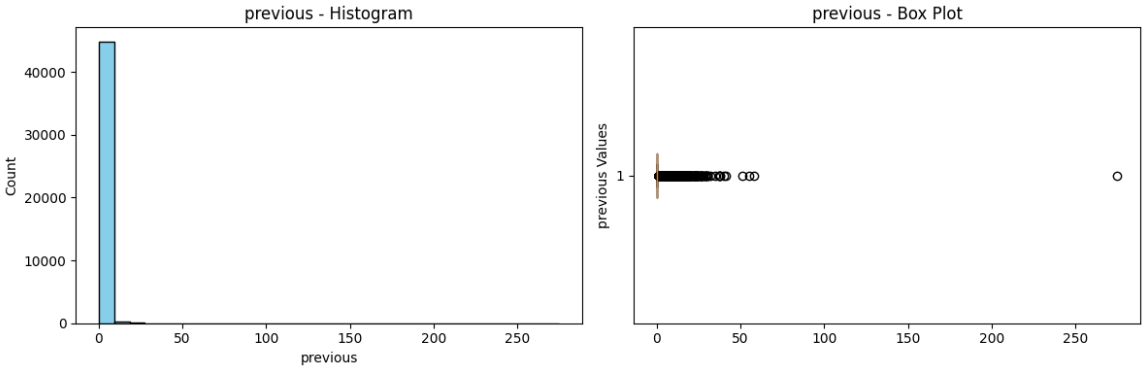

duration - Box Plot
duration Values


campaign - Histogram
Count
campaign

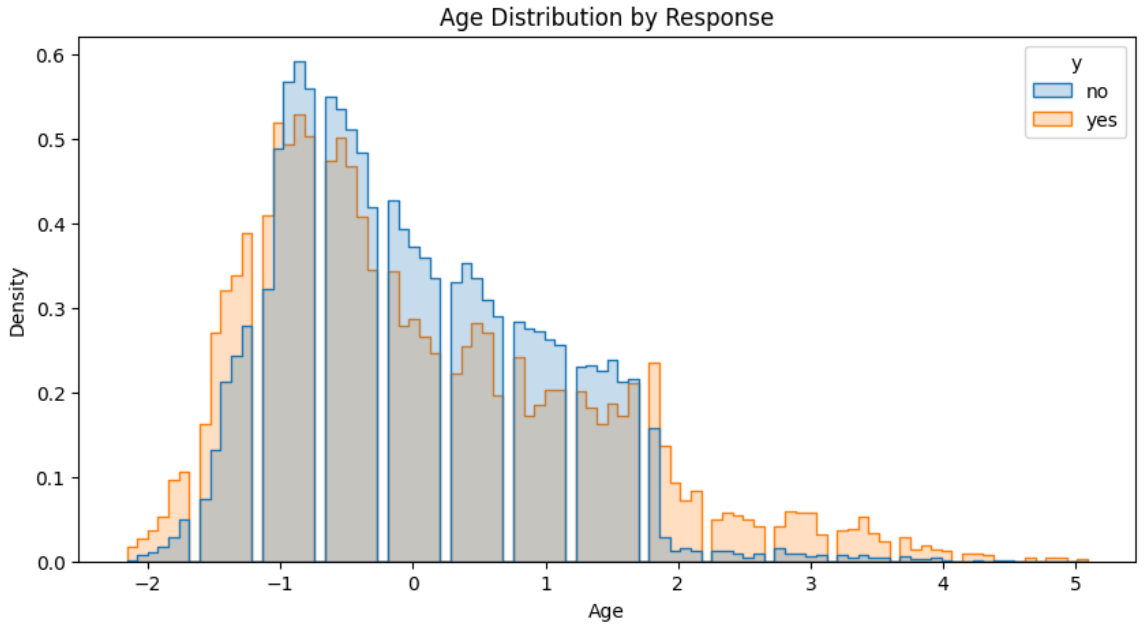

campaign - Box Plot
campaign Values


pdays - Histogram
Count
pdays

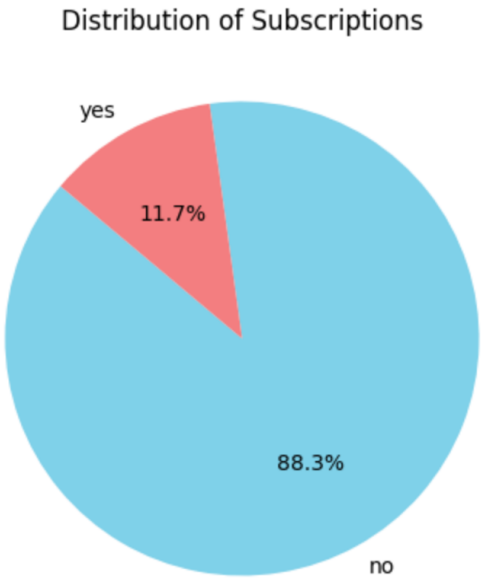

pdays - Box Plot
pdays Values



We used a density plot to show the density of clients with respect to age who have subscribed to term deposit. In this plot it shows, clients who are in the age between 30 and 59 have mixed response towards subscribing a term deposit. Clients who are aged greater than 60 are more likely to subscribe to the term deposit.



We used a pie chart to understand our target variable `'is_subscribed'`. This visualization says that 11.7 percent of clients subscribed to term deposit and 88.3 percent of clients didn't subscribe to term deposit. This also tells that our dataset is highly imbalanced which indicates that we have to do either undersampling or oversampling.



3 Task

- **Looking for Deeper Null Values**

  As part of our task, we have chosen to exclude attributes with a higher number of unknown values. The snippet below reveals that there are more unknown values in the "contact" and "poutcome" attributes. Consequently, we have decided to exclude these attributes.

```
#Looking deeper for null values
missing_values = {
    "job": (data["job"] == "unknown").sum(),
    "education": (data["education"] == "unknown").sum(),
    "contact": (data["contact"] == "unknown").sum(),
    "poutcome": (data["poutcome"] == "unknown").sum()
}

for column, missing_count in missing_values.items():
    print(f"{column}: {missing_count} missing values")
```

✓ 0.0s

```
job: 288 missing values
education: 1857 missing values
contact: 13020 missing values
poutcome: 36959 missing values
```

- **Outlier Detection**

  Outliers refer to data points that deviate significantly from the overall pattern of a dataset. These aberrant observations can distort the distribution of data and typically arise from inconsistencies in data entry or erroneous measurements. Recognizing the impact outliers can have on model performance, it becomes crucial to identify and eliminate them to ensure the trained model generalizes effectively to a valid range of test inputs[1].

  To achieve this, various statistical techniques are commonly employed for outlier detection and removal. These methods aid in maintaining the integrity of the dataset, allowing machine learning models to better capture the underlying patterns and relationships within the data. The process of identifying and handling outliers is a critical step in enhancing the reliability and robustness of predictive models, contributing to their overall effectiveness in real-world applications[1].

  Why outliers should be detected?

Within the machine learning pipeline, the crucial stage of data cleaning and preprocessing plays a pivotal role in enhancing our comprehension of the dataset. This phase involves addressing missing values, identifying outliers, and other critical tasks[1].

Outliers, being notably distinct values, whether exceptionally low or high, can significantly impact the outcomes of statistical analyses applied to the dataset. Their presence may lead to less effective and less reliable models. However, addressing outliers necessitates domain expertise, and it is essential not to apply outlier detection techniques blindly, but rather with a deep understanding of the data distribution and the specific use case[1].

Consider a dataset of house prices as an illustrative example. Detecting outliers, such as a few houses priced at 1.5 million, might be straightforward, but it becomes nuanced when there is a considerable number of houses priced at $1.5 million or above. Such instances may not necessarily be outliers but could indicate a legitimate trend in rising house prices, underscoring the importance of domain knowledge[1].

The ultimate objective of outlier detection is to eliminate truly anomalous points, enabling the construction of a model that performs optimally on new, unseen test data. To achieve this, we will delve into various techniques designed to effectively identify outliers in the dataset[1].

Detecting Outliers using Z-Score

Now, let's delve into the concept of the z-score. In the context of a normal distribution characterized by a mean (μ) and standard deviation (σ), the z-score for a given value (x) within the dataset is mathematically expressed as[1]:

$$z = \frac{(x - \mu)}{\sigma}$$

This formulation reveals that when x equals the mean (μ), the z-score is 0. As x deviates from the mean by ± 1, ± 2, or ± 3 standard deviations, the corresponding z-score takes on the values of ± 1, ± 2, or ± 3, respectively[1].

It's noteworthy that this z-score technique aligns with the earlier approach involving standard deviation-based scores. The transformation maintains a consistent mapping: data points below the lower limit ($\mu - 3\sigma$) now correspond to z-scores below -3, while points above the upper limit ($\mu + 3\sigma$) map to z-scores exceeding 3. This establishes a standardized scale of [-3, 3] for the range $[lower\_limit, upper\_limit]$[1].

Applying this z-score technique to our dataset facilitates a robust analysis, enabling us to identify and interpret the relative positions of data points within the distribution. This method proves particularly valuable for outlier detection and assessing the variability of data.[1]

Detecting Outliers using Interquartile Range (IQR)

In the field of statistics, the interquartile range (IQR) functions as a crucial metric that gauges the spread of a given dataset by quantifying the disparity between its first quartile ($Q1$) and third quartile ($Q3$). $Q1$, often denoted as the 25% quartile or $q25$, indicates the threshold below which 25% of the dataset's data points are situated. Conversely, $Q3$, known as the 75% quartile or $q75$, designates the point beneath which 75% of the dataset's values are positioned. Using these established notations, the IQR is computed as the difference between $q75$ and $q25$[1].

This statistical measure proves invaluable as it encapsulates the central 50% of the dataset, effectively sidestepping the influence of outliers or extreme values. The IQR's focus on the middle portion of the data distribution offers a robust assessment of variability, providing insights into the dataset's overall shape. Additionally, the IQR facilitates the identification of potential data skewness or asymmetry, offering a nuanced perspective beyond standard measures like the mean and standard deviation. By emphasizing quartiles, the IQR enables a more nuanced understanding of the distribution's central tendency and dispersion, laying the groundwork for comprehensive statistical analyses[1].

Sample code for detecting Outliers using Interquartile Range (IQR)

Import the necessary modules

```
import numpy as np
import pandas as pd
import seaborn as sns
```

Define a function generate_scores() that returns 200 samples of normal distribution which contains student scores. This function should be called and output is stored in another variable.

```
def generate_scores(mean=60,std_dev=12,num_samples=200):
    np.random.seed(27)
    scores = np.random.normal(loc=mean,scale=std_dev,size=num_samples)
    scores = np.round(scores, decimals=0)
    return scores
scores_data = generate_scores()
```

Load the data into Pandas Dataframe

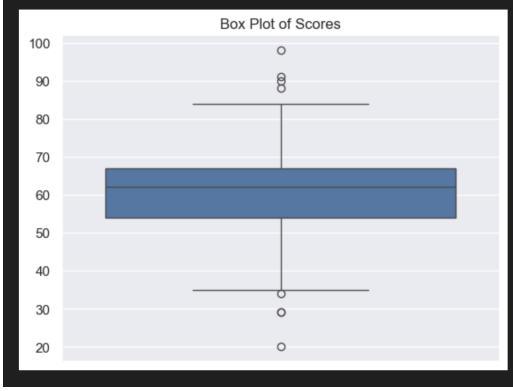
```
df_scores = pd.DataFrame(scores_data,columns=['score'])
0.0s
```

In the realm of exploratory data analysis, the utilization of a box plot, also known as a box and whisker plot, proves to be an effective means of visually inspecting and comprehending the underlying distribution of a dataset, particularly in relation to the identification of outliers. The distinctive feature of a box plot lies in its ability to graphically represent the central tendency, spread, and potential outliers within a dataset[1].

The visualization is constructed through the use of a rectangular box, or "box," which spans the interquartile range (IQR) of the data, encompassing the first quartile (Q1) to the third quartile (Q3). Extending from the box are "whiskers" that delineate the range of typical data points, and points beyond these whiskers are flagged as potential outliers. This graphical representation allows for a nuanced examination of the data's dispersion and facilitates the identification of data points that deviate significantly from the norm[1].

In Python, specifically using Seaborn, the boxplot function serves as a powerful tool for generating box plots. Leveraging this function provides a straightforward and insightful way to visually assess the distribution of a dataset and promptly detect any anomalies or outliers that may impact subsequent analyses[1].

```
sns.boxplot(data=scores_data).set(title="Box Plot of Scores")
1.3s
[Text(0.5, 1.0, 'Box Plot of Scores')]
```



Implement describe() on dataframe df_scores

```
df_scores.describe()
```

✓ 0.0s

	score	z_score
count	200.000000	2.000000e+02
mean	61.005000	-2.131628e-16
std	11.854434	1.000000e+00
min	20.000000	-3.459043e+00
25%	54.000000	-5.909181e-01
50%	62.000000	8.393484e-02
75%	67.000000	5.057179e-01
max	98.000000	3.120773e+00

In the process outlined above, we extract the first quartile (Q1) and third quartile (Q3) values from the computed quartile results, representing the 25% and 75% points in the dataset, respectively. This extraction allows us to calculate the Interquartile Range (IQR), a robust statistical measure that quantifies the spread of the middle 50% of the data distribution[1].

With the IQR in hand, we proceed to establish lower and upper limits for filtering the dataset. The lower limit is set at Q1 minus a certain factor times the IQR, and the upper limit is set at Q3 plus a corresponding factor times the IQR. These limits serve as criteria for identifying potential outliers beyond the typical range of the dataset[1].

Implementing this approach enables us to systematically filter the dataframe (df_scores) based on these calculated limits, retaining data points within a statistically reasonable range while identifying and potentially excluding outliers that fall outside this range. This method, grounded in quartile values and the IQR, provides a robust means of outlier detection that is resilient to the influence of extreme values and not reliant on assumptions about the data distribution[1].

```
IQR = 67-54
lower_limit = 54 - 1.5*IQR
upper_limit = 67 + 1.5*IQR
print(upper_limit)
print(lower_limit)
```

✓ 0.0s

```
86.5
34.5
```

Following the preceding analysis, the subsequent stage involves refining the dataframe, df_scores, through a filtering process aimed at retaining records within a specified permissible range. This range is determined by applying calculated lower and upper limits derived from the quartile values and the Interquartile Range (IQR). These limits act as stringent criteria, allowing us to systematically identify and retain data points falling within the statistically acceptable range[1].

By employing this filtering mechanism, the dataset is strategically refined, preserving records that align with the central 50% of the distribution and excluding those that exhibit significant deviation from the norm. This step is essential for mitigating the influence of potential outliers and fostering a more accurate representation of the dataset for subsequent analyses. Moreover, it ensures that the retained records adhere to a statistically reasonable range, enhancing the robustness of the dataset for further exploration and interpretation[1].

```
df_scores_filtered = df_scores[(df_scores['score']>lower_limit) & (df_scores['score']<upper_limit)]
print(df_scores_filtered)
```

✓ 0.0s

```
     score   z_score
0     75.0  1.180571
1     56.0 -0.422205
2     67.0  0.505718
3     65.0  0.337005
4     63.0  0.168291
..     ...       ...
194   42.0 -1.603198
195   76.0  1.264928
196   67.0  0.505718
197   74.0  1.096214
199   53.0 -0.675275

[192 rows x 2 columns]
```

As observed in the results, eight data points are identified as outliers through this method, leading to a filtered dataframe consisting of 192 records[1].

It is not mandatory to employ the describe method exclusively for quartile identification; an alternative approach involves utilizing the percentile() function in NumPy. This function accepts two essential parameters: 'a', representing an array or dataframe, and 'q', a list specifying the desired quartiles. The subsequent code cell illustrates the practical application of the percentile function to compute both the first and third quartiles. This versatile function provides a more flexible and customizable means of quartile computation, allowing for efficient and targeted analysis of dataset distributions without solely relying on the describe method[1].

```
q25, q75 = np.percentile(a = df_scores,q=[25,75])
IQR = q75-q25
print(IQR)
```

✓ 0.0s

```
13.0
```

In our analytical endeavor, we determined the outlier percentages for numerical attributes through the application of the Interquartile Range (IQR) method. Specifically, we designated the upper quartile (Q3) and lower quartile (Q1) as 75 and 25, respectively. The provided code snippet in the following screenshot encapsulates the procedure employed to compute these outlier percentages utilizing the IQR method. During this analysis, we carefully set the quartile boundaries to capture the central 50% of the data distribution, allowing us to identify and quantify the presence of outliers beyond this range. This methodological approach is instrumental in understanding the statistical characteristics of the dataset and provides valuable insights into the extent of variations in the numerical attributes under consideration. The specified quartile values, coupled with the IQR method, contribute to a robust outlier detection technique, laying the foundation for informed decision-making in subsequent analytical tasks.

```
# Checking for outliers
# Converting object types to 'category' for efficiency where appropriate
categorical_columns = data.select_dtypes(include=['object']).columns
data[categorical_columns] = data[categorical_columns].astype('category')

# Checking for outliers in numerical columns
# We use the IQR method to detect outliers
numerical_columns = data.select_dtypes(include=['int64', 'float64']).columns
Q1 = data[numerical_columns].quantile(0.25)
Q3 = data[numerical_columns].quantile(0.75)
IQR = Q3 - Q1
outliers = ((data[numerical_columns] < (Q1 - 1.5 * IQR)) | (data[numerical_columns] > (Q3 + 1.5 * IQR))).sum()

outliers_percentage = (outliers / len(data)) * 100
print("Outlier Percentages: ")
outliers_percentage
```

✓ 0.1s Python

```
Outlier Percentages:

age          1.077171
balance     10.459844
day          0.000000
duration     7.155338
campaign     6.777112
pdays       18.263255
previous    18.263255
dtype: float64
```

- **What is Scikit-learn?**

Scikit-learn, also known as sklearn, is a free and open-source library for Python that focuses on machine learning and data modeling. It offers a variety of tools for tasks like classification,

regression, and clustering. Inside, you'll find powerful algorithms like support vector machines, random forests, gradient boosting, k-means, and DBSCAN. Sklearn is built to work seamlessly with other Python libraries such as NumPy and SciPy[2].

Debuting in 2010, Scikit-learn has become a major player in the Python machine learning world. It's packed with a range of algorithms for data modeling and machine learning, providing a consistent interface in Python. This library supports a standardized and straightforward way to work with different models. For instance, Scikit-learn adopts a user-friendly fit/predict approach for its classification algorithms[2].

Scikit-learn works seamlessly with various Python libraries like matplotlib, plotly, NumPy, Pandas, SciPy, and more. You can easily use NumPy arrays and Pandas dataframes directly with Scikit-learn's algorithms[2].

This library offers a wide range of supervised and unsupervised learning algorithms, addressing key areas such as[2]:

- **Classification:** Determining the category to which an object belongs.
- **Regression:** Predicting a continuous-valued attribute linked to an object.
- **Clustering:** Automatically grouping similar objects into sets, featuring models like k-means.
- **Dimensionality Reduction:** Decreasing the number of attributes in data for summarization, visualization, and feature selection. This includes models like Principal Component Analysis (PCA).
- **Model Selection:** Comparing, validating, and selecting parameters and models.
- **Pre-processing:** Extracting features and normalizing data, including defining attributes in image and text data.

- **Normalization**

In our task, we incorporated the `StandardScaler` for the purpose of normalization, employing the Z-Score Normalization process as part of our strategy to handle outliers in the dataset[3][4].

The `StandardScaler` is a method commonly used in machine learning and statistics to normalize features by transforming them to have a mean of 0 and a standard deviation of 1. It is particularly effective in mitigating the impact of outliers and ensuring that variables are on a consistent scale[3][4].

The Z-Score Normalization, as implemented by the `StandardScaler`, involves subtracting the mean (μ) of the dataset from each data point and dividing the result by the standard deviation (σ). Mathematically, this is expressed as[3][4]:

$$Z - Score\ Normalization : z = \frac{(x - \mu)}{\sigma}$$

Where:

- x is the original value,
- μ is the mean of the dataset,
- σ is the standard deviation of the dataset.

By applying this normalization technique, we ensure that the features are standardized and follow a normal distribution. This is beneficial for machine learning models as it helps in achieving better convergence during training and avoids certain features disproportionately influencing the model due to differences in scale[3][4].

In the context of handling outliers, Z-Score Normalization is robust as it centers the data around its mean, making extreme values less impactful. Therefore, by incorporating the `StandardScaler` with Z-Score Normalization, our task is better equipped to handle outliers and promote a more stable and effective analysis[3][4].

Below screenshot which contains code snippet tells the implementation of normalization on our dataset.

```
# Normalising the data
# Using Z-score normalization to normalize the data.
import numpy as np
from sklearn.preprocessing import StandardScaler

# Handling missing values: leaving 'unknown' as is in 'job', 'education', 'contact', 'poutcome'

# Handling outliers: Applying log transformation to certain columns
columns_to_transform = ['balance', 'duration', 'campaign', 'pdays', 'previous']
data[columns_to_transform] = np.log1p(data[columns_to_transform])

# Normalization/Standardization: Standardizing numerical columns
scaler = StandardScaler()

# Correcting pdays values: adding 1 to all entries before applying the log transformation
data['pdays'] = np.log1p(data['pdays'] + 1)

# Reattempting standardization of numerical columns
data[numerical_columns] = scaler.fit_transform(data[numerical_columns])

# Displaying the transformed data again
transformed_data_head = data.head()
transformed_data_head
```

	age	job	marital	education	default	balance	housing	loan	day	month	duration	campaign	pdays	previous	y
0	1.606965	management	married	tertiary	no	0.602969	yes	no	-1.298476	may	0.460485	-1.00556	NaN	-0.449689	no
1	0.288529	technician	single	secondary	no	-0.513252	yes	no	-1.298476	may	-0.073740	-1.00556	NaN	-0.449689	no
2	-0.747384	entrepreneur	married	secondary	no	-1.732461	yes	yes	-1.298476	may	-0.813329	-1.00556	NaN	-0.449689	no
3	0.571051	blue-collar	married	unknown	no	0.534632	yes	no	-1.298476	may	-0.598974	-1.00556	NaN	-0.449689	no
4	-0.747384	unknown	single	unknown	no	-2.085889	no	no	-1.298476	may	0.196395	-1.00556	NaN	-0.449689	no

- **Label Encoding**

  Label Encoding is a method used to transform categorical columns into numerical format, making them suitable for machine learning models that exclusively operate on numerical data. This conversion is a crucial pre-processing step in machine-learning projects[5].

  In simpler terms, when we have categories like "red," "blue," and "green," label encoding assigns a unique numerical label to each category, such as 0, 1, and 2. This numeric representation allows machine learning models to understand and work with the data effectively[5].

  Imagine training a machine learning model to predict car prices, and one of the features is "car color." Since models prefer numbers, label encoding helps convert colors like "black" or "white" into corresponding numerical values[5].

  So, label encoding acts as a bridge, translating categorical information into a language that machine learning algorithms comprehend, ultimately enhancing the model's ability to make accurate predictions[5].

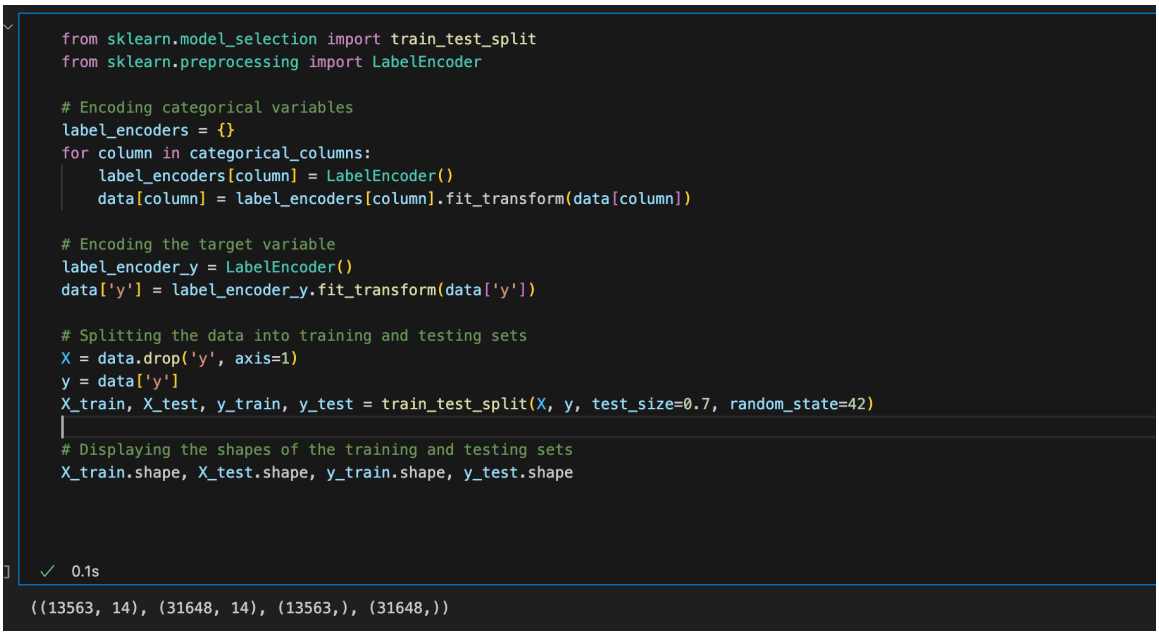
- **Spliting the dataset into training and testing**

  The `train_test_split()` method is a tool we use to divide our dataset into two parts: one for training our model and one for testing it. Initially, we break our data into features (`X`) and labels (`y`). The resulting split creates sets known as `X_train`, `X_test`, `y_train`, and `y_test`. When training our model, we utilize `X_train` and `y_train`, and for testing the model's predictions, we use `X_test` and `y_test`. It's common to check and adjust the size of these sets, with a recommendation to keep the training set larger than the testing set[6].

  In simple terms, the training set is the data our model learns from, while the testing set is used to assess how well the model predicts outcomes. Additionally, there's a concept of a validation set, a subset of the training set used to fine-tune the model's performance[6].

  Underfitting occurs when a model hasn't effectively learned the relationship between input and output variables, leading to high error rates on both training and unseen data. On the other hand, overfitting happens when a model memorizes the training data but struggles to perform well on new, unseen data, losing its generalization ability[6].

  Below screenshot shows how have we done the label encoding and split the testing and training datasets



```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Encoding categorical variables
label_encoders = {}
for column in categorical_columns:
    label_encoders[column] = LabelEncoder()
    data[column] = label_encoders[column].fit_transform(data[column])

# Encoding the target variable
label_encoder_y = LabelEncoder()
data['y'] = label_encoder_y.fit_transform(data['y'])

# Splitting the data into training and testing sets
X = data.drop('y', axis=1)
y = data['y']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.7, random_state=42)

# Displaying the shapes of the training and testing sets
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
✓ 0.1s
((13563, 14), (31648, 14), (13563,), (31648,))
```

- **Sampling**

Managing data is a crucial aspect of data science, and when dealing with datasets, encountering imbalances is common. To address this, we employ two methods known as under-sampling and over-sampling, collectively referred to as resampling. Imagine having a dataset for testing and training, but using imbalanced data may lead to imperfect accuracy results[7].
Oversampling and undersampling are techniques used in data analysis to adjust the class distribution within a dataset. These terms are not only relevant in statistical sampling and survey design methodology but also play a significant role in machine learning. Resampling methods aim to either add or remove examples from the training dataset to modify the class distribution. This adjustment helps achieve a more balanced class distribution, allowing standard machine learning classification algorithms to be applied successfully to the transformed datasets[7].

  - **Under Sampling**

Under-sampling is a technique employed to address imbalanced datasets by retaining all data from the minority class while reducing the size of the majority class. It's among the various methods data scientists use to extract more accurate insights from initially imbalanced datasets[7].

This technique involves removing samples from the majority class, either with or without replacement. On the other hand, oversampling entails enhancing the training data by adding multiple copies of some minority class samples. Under-sampling or oversampling is often applied to overcome imbalances in datasets[7].

Consider a scenario where we have a dataset, X, with 100 rows and another dataset, y, with only 10 rows. If we opt for under-sampling, the process randomly selects 10 rows from X, and the accuracy is determined based on these 10 rows from X and 10 rows from y, totaling 20 rows in the evaluation[7].

- **Data Imputation**

Data imputation is a strategy to preserve most of a dataset's information by substituting missing data with alternative values. This approach is essential as removing data each time it's missing would be impractical and significantly shrink the dataset, potentially introducing bias and hindering analysis[8].

Imputation becomes necessary due to the problems posed by missing data:

  - Distortion of Dataset: Extensive missing data can distort variable distributions, altering the relative importance of different categories.
  - Compatibility Issues with ML Libraries: Many machine learning libraries, like SkLearn, may encounter errors without automated handling of missing data.
  - Impact on Final Model: Missing data can introduce bias, influencing the analysis of the final model.
  - Desire to Preserve the Entire Dataset: In cases where every piece of data is crucial, discarding any portion, especially when the dataset isn't large, can significantly affect the final model.

Having grasped the concept and significance of data imputation, let's delve into various techniques, including:

  - Next or Previous Value
  - K Nearest Neighbors
  - Maximum or Minimum Value
  - Missing Value Prediction
  - Most Frequent Value
  - (Rounded) Mean or Moving Average or Median Value
  - Fixed Value

- **Random Forest Classifier**
  Random Forest stands out as a well-known machine learning algorithm within the realm of supervised learning. Its application extends to both Classification and Regression tasks in ML. The foundation of this algorithm lies in ensemble learning, a technique that merges multiple classifiers to address intricate problems and enhance overall model performance[9].

  In simple terms, a Random Forest classifier comprises several decision trees, each trained on different subsets of the provided dataset. Its approach involves taking the average of these trees' predictions to enhance the overall predictive accuracy. Unlike relying on a single decision tree, a random forest aggregates predictions from each tree and, based on majority votes, determines the final output[9].

  The effectiveness of a random forest grows with an increased number of trees in the forest, yielding higher accuracy while concurrently preventing the risk of overfitting. This makes it a robust choice for achieving accurate and well-generalized predictions in machine learning scenarios[9].

  - **Accuracy:** The primary metric commonly used to evaluate models is Accuracy, which signifies the proportion of correct predictions relative to all predictions made by the model. In simpler terms, Accuracy provides an overall measure of how well a model performs in correctly predicting outcomes across the entire dataset[10].

    $$accuracy = \frac{true\ positives + true\ negatives}{true\ positives + true\ negatives + false\ negatives + false\ positives}$$

  - **Precision:** Precision is an assessment of the accuracy of positive predictions, specifically measuring how many of them are correct (true positives). In simpler terms, it evaluates the model's ability to accurately identify instances it labels as positive[10].

    $$precision = \frac{true\ positives}{true\ positives + false\ positives}$$

  - **Recall:** Recall is a metric that gauges the proportion of correctly identified positive cases by the classifier, considering all the positive cases present in the dataset. In simpler terms, Recall assesses the model's ability to capture and correctly classify instances belonging to the positive class among all the actual positive cases[10].

    $$recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

  - **F1 Score:** The F1-Score is a comprehensive metric that integrates both precision and recall, often referred to as their harmonic mean. The harmonic mean is an alternative method for calculating an "average" of values, particularly considered more appropriate for ratios, as seen in precision and recall. In simple terms, the F1-Score provides a balanced assessment by combining the strengths of precision and recall, ensuring a well-rounded evaluation of a model's performance[10].

    $$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Below Screenshot shows the snippet of code where we have handled data impuation and classification using Random Forest Classifier.

```python
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from imblearn.under_sampling import RandomUnderSampler

# Split the data into training and testing sets
#X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.7, random_state=42, stratify=y)

# Imputing missing values in X_train and X_test
imputer = SimpleImputer(strategy='median')  # or choose 'mean', 'most_frequent'
X_train_imputed = imputer.fit_transform(X_train)
X_test_imputed = imputer.transform(X_test)

# Applying Random Under Sampler to balance the dataset
rus = RandomUnderSampler(random_state=42)
X_resampled, y_resampled = rus.fit_resample(X_train_imputed, y_train)

# Training the Random Forest Classifier
rf_classifier = RandomForestClassifier(random_state=42)
rf_classifier.fit(X_resampled, y_resampled)

# Predicting on the test set
y_pred_rf = rf_classifier.predict(X_test_imputed)

# Evaluating the model
classification_rep = classification_report(y_test, y_pred_rf)
print(classification_rep)
```

```
              precision    recall  f1-score   support

           0       0.98      0.81      0.89     27935
           1       0.37      0.86      0.52      3713

    accuracy                           0.82     31648
   macro avg       0.68      0.83      0.70     31648
weighted avg       0.91      0.82      0.84     31648
```

Classification results using Random Forest has given accuracy of 0.82

- **Gradient Boosting Classifier**

  Gradient boosting is a remarkable technique renowned for its swift and accurate predictions, especially when dealing with extensive and intricate datasets. It has demonstrated superior performance in various domains, from Kaggle competitions to addressing real-world business challenges. In the realm of machine learning, errors are pivotal considerations, with bias error and variance error being the primary types. The Gradient Boost algorithm serves a crucial role in mitigating the bias error of a model, contributing to enhanced overall performance and reliability[11].

  Classification results using Gradient Boosting has given accuracy of 0.83

```python
from sklearn.ensemble import GradientBoostingClassifier

# Training the Gradient Boosting Classifier
gb_classifier = GradientBoostingClassifier(random_state=42)
gb_classifier.fit(X_resampled, y_resampled)

# Predicting on the test set
y_pred_gb = gb_classifier.predict(X_test_imputed)

# Evaluating the model
classification_rep = classification_report(y_test, y_pred_gb)
print(classification_rep)
```

✓ 1.2s

```
              precision    recall  f1-score   support

           0       0.98      0.82      0.89     27935
           1       0.39      0.85      0.54      3713

    accuracy                           0.83     31648
   macro avg       0.68      0.84      0.71     31648
weighted avg       0.91      0.83      0.85     31648
```

- Without Under Sampling
  - Random Forest Classifier:

Below is the classification report for the same task without using any sampling techniques.

```python
# Initialize the Random Forest Classifier
rf_classifier = RandomForestClassifier(random_state=42)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred_rf = rf_classifier.predict(X_test)

# Calculate accuracy
accuracy_rf = accuracy_score(y_test, y_pred_rf)

# Generate a classification report
classification_rep_rf = classification_report(y_test, y_pred_rf)

# Print the accuracy and classification report
print(f"Accuracy: {accuracy_rf}")
print("Without Undersampling Report \n",classification_rep_rf)
```

```
Accuracy: 0.8980978260869565
Without Undersampling Report
              precision    recall  f1-score   support

           0       0.92      0.97      0.94     27935
           1       0.61      0.36      0.45      3713

    accuracy                           0.90     31648
   macro avg       0.77      0.66      0.70     31648
weighted avg       0.88      0.90      0.89     31648
```

Gradient Boosting Classifier:

Below is the classification report for the same task without using any sampling techniques.

```
# Initialize the Gradient Boosting Classifier
gb_classifier = GradientBoostingClassifier(random_state=42)

# Train the classifier on the training data
gb_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred_gb = gb_classifier.predict(X_test)

# Calculate accuracy
accuracy_gb = accuracy_score(y_test, y_pred_gb)

# Generate a classification report
classification_rep_gb = classification_report(y_test, y_pred_gb)

# Output the accuracy and classification report
print("Accuracy: ", accuracy_gb)
print(classification_rep_gb)
```

```
Accuracy:  0.89737108190091
              precision    recall  f1-score   support

           0       0.92      0.97      0.94     27935
           1       0.60      0.37      0.46      3713

    accuracy                           0.90     31648
   macro avg       0.76      0.67      0.70     31648
weighted avg       0.88      0.90      0.89     31648
```

4 Conclusion

As its quite evident from the classification reports that both the classifiers have almost the same accuracy.. But accuracy isn't everything. When we closely look at the precision scores. it is quite evident that the models with under sampling approaches are much better at predicting the customers who will subscribe to the term deposit.

Without using under sampling techniques, the results are more precise but it may miss positive cases. The model with under sampling has a higher recall for the positive class, meaning it is better at identifying all potential subscribers.

In conclusion, the model with sampling techniques applied to it is more balanced and it is better for identifying positive cases.

References

[1] C, Bala Priya. "How to Detect Outliers in Machine Learning – 4 Methods for Outlier Detection." freeCodeCamp.Org, freeCodeCamp.org, 11 July 2022, Available:www.freecodecamp.org/news/how-to-detect-outliers-in-machine-learning

[2] "What Is Sklearn?" Domino Data Lab, domino.ai/data-science-dictionary/sklearn. Accessed 12 Dec. 2023.

[3] Zach. "Z-Score Normalization: Definition & Examples." Statology, August 12, 2021. Available:https://www.statology.org/z-score-normalization/

[4] Upadhyay, Amit. "Standardscaler and Normalization with Code and Graph." Medium, July 23, 2020. Available:https://medium.com/analytics-vidhya/standardscaler-and-normalization-with-code-and-graph-ba220025c054

[5] "Label Encod ing in Python." GeeksforGeeks, April 18, 2023. Available:https://www.geeksforgeeks.org/ml-label-encoding-of-datasets-in-python/

[6] "How to Split the Dataset with Scikit-Learn's Train_test_split() Function." GeeksforGeeks, June 29, 2022. Available:https://www.geeksforgeeks.org/how-to-split-the-dataset-with-scikit-learns-train$_t est_s plit - function/$

[7] rmadhu2131. "Under Sampling." Numpy Ninja, November 29, 2022. Available:https://www.numpyninja.com/post/under-sampling

[8] Simplilearn. "Introduction to Data Imputation: Simplilearn." Simplilearn.com, August 16, 2023. Available:https://www.simplilearn.com/data-imputation-article: :text=Data

[9] "Machine Learning Random Forest Algorithm - Javatpoint." www.javatpoint.com. Accessed December 12, 2023. Available:https://www.javatpoint.com/machine-learning-random-forest-algorithm

[10] Kanstrén, Teemu. "A Look at Precision, Recall, and F1-Score." Medium, August 4, 2023. Available:https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec

[11] Saini, Anshul. "Gradient Boosting Algorithm: A Complete Guide for Beginners." Analytics Vidhya, August 2, 2023. Available:https://www.analyticsvidhya.com/blog/2021/09/gradient-boosting-algorithm-a-complete-guide-for-beginners/

red