Data imputation using Machine Learning

Potential utilities of data imputation in financial and economic context:

1. **Bias Prevention in Financial Analysis:** data imputation can mitigate bias introduced by incomplete data, ensuring that financial analyses and forecasting models reflect a more accurate market representation. This aids analysts in avoiding decisions based on partial data sets that could otherwise lead to erroneous conclusions.

2. **Improving Data Quality for Risk Assessment:** data imputation helps to fill gaps in financial records, providing a more solid foundation for risk assessment. Having a complete data set is critical for accurately analyzing credit and market risk, reducing uncertainties and enhancing confidence in risk estimates.

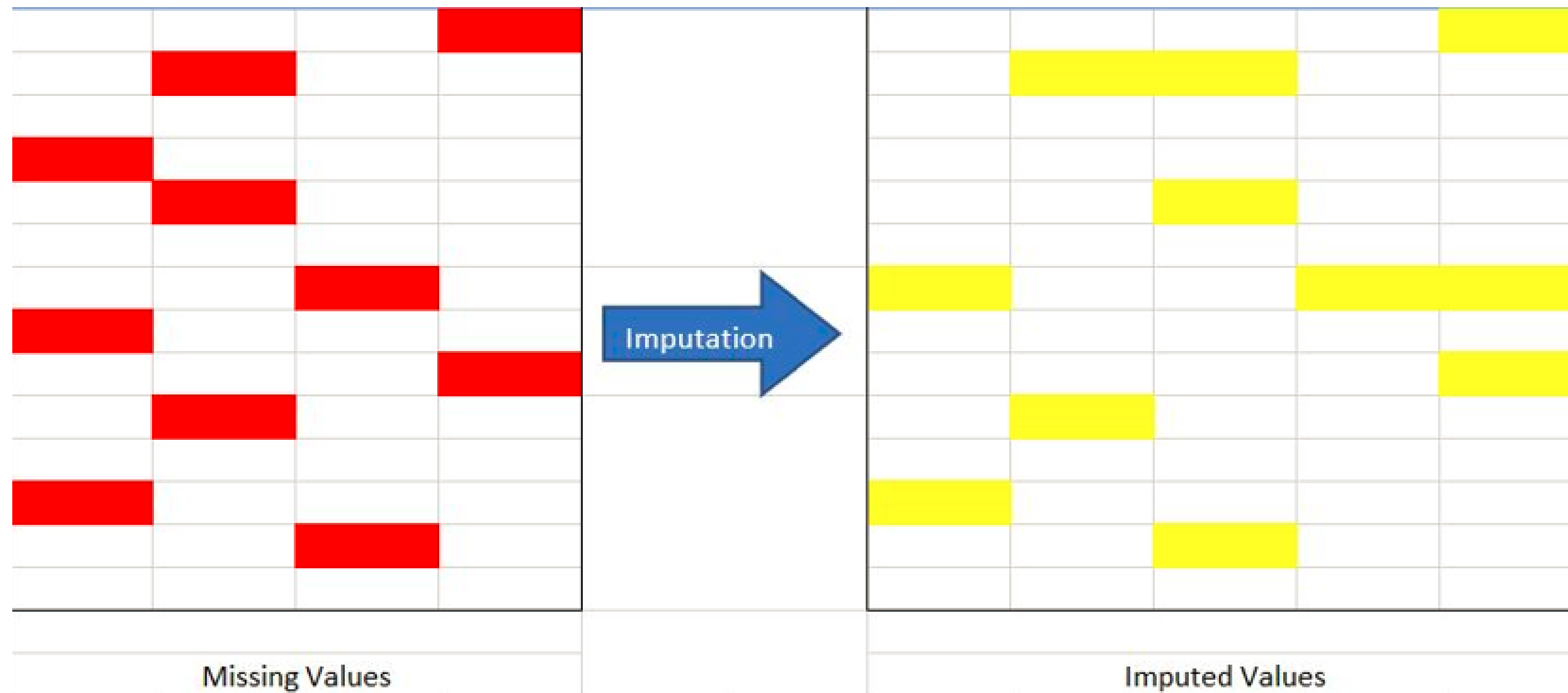
3. **Optimizing Investment Strategies:** complete and more accurate data are crucial for identifying reliable market patterns. Data imputation allows investors to formulate asset allocation and market timing strategies based on a comprehensive view of historical performances, potentially increasing returns and minimizing losses.

4. **Support in M&A Operations:** in mergers and acquisitions, accurate data imputation provides a complete view of the financial health of the companies involved. This is vital for conducting thorough due diligence, influencing the valuation and the final decision to proceed with an M&A transaction.

5. **Asset and Portfolio Valuation:** data imputation ensures that all relevant information is considered, allowing for more precise valuation and optimal risk management.

6. **Applications in Economics and Financial Research:** data imputation is invaluable for economic and financial research, enabling analysts to fully leverage large historical data sets. This approach improves the quality of research and the formulation of policies based on complete historical data.

7. **Artificial Intelligence:** more accurate and complete data are fundamental for training machine learning and deep learning models in financial forecasting and algorithmic trading. Data imputation enhances forecast accuracy and the quality of decision-making algorithms, leading to more informed and sophisticated investment decisions.



Why use Machine Learning instead of traditional approaches for data imputation?

In the field of financial analysis, data is the basis for the entire investment decision-making process. This project introduces three machine learning techniques—MICEFOREST, Bayesian Ridge and Linear Regression—to the critical task of data imputation. These methods surpass traditional approaches with enhanced accuracy and an ability to maintain the complex relationships inherent in financial data. Here, I will outline their benefits and demonstrate their application in refining the quality of financial datasets for robust analytical outcomes.

Some of the benefits of using Machine Learning models for data imputation:

- **Greater Accuracy:** machine learning models can handle complex relationships between variables much better than simple imputation methods. For instance, MICEFOREST uses random forest algorithms to model each feature with missing data as a function of other features, resulting in more accurate estimates.
- **Handling Non-Linearity:** algorithms like Bayesian Ridge are particularly effective at modeling non-linear relationships and interactions between variables, which simpler methods cannot capture.

- **Incorporating Uncertainty:** unlike mean or median imputation, machine learning methods can incorporate uncertainty into the estimation of missing data. This is especially true for methods like MICEFOREST, which generates multiple imputed datasets to reflect the possible variability in the data.
- **Adaptability:** machine learning models can be trained on one subset of data and applied to another, making them adaptable and scalable across large datasets.
- **Bias Prevention:** by using forecasting models for imputation, there is a reduced risk of introducing bias into the data. Linear regression, for example, can preserve the linear relationship between variables better than imputation based on central tendency statistics.
- **Efficiency Across Data Types:** machine learning models are flexible and can be effectively used with both categorical and continuous data, unlike simpler methods that might be suitable for only one type of data.
- **Feature Interaction:** advanced machine learning algorithms consider complex interactions between features that might be missed with simpler imputation methods.
- **Automation and Reproducibility:** once trained, machine learning models can be easily automated and used to impute missing data in a reproducible manner with little human intervention.

Explanation of work:

In this project, we will delve into a practical example of data imputation, applying it to financial ratios of European companies in the financial sector from 2019 to 2022. We will address instances where certain data points are missing for selected companies, employing machine learning techniques to forecast and fill these gaps. Notably, a preliminary phase of Exploratory Data Analysis (EDA) is essential. This foundational step will enable us to gain a deeper understanding of our dataset, identify any outliers and perform necessary preprocessing to minimize further anomalies during the forecasting phase. Our EDA will also provide valuable insights that will guide the imputation process and help ensure that the data used in subsequent analyses is as robust and informative as possible.

Project workflow:

1. **Initial Data Cleaning:** identify and replace initial anomalies values to clean the dataset.
2. **Visual Analysis:** generate plots to visualize data distribution and understand the underlying trends.
3. **Isolation Forest ML model with default threshold to detect outlier (detailed model explanation will be provided in its respective section):**

4. **Graphical Outlier Representation:**

    - a. Scatter plot of the outliers identified by the isolation forest
    - b. Bubble graph for a simple and immediate visualization of the most concentrated outlier areas
    - c. Dataframe showing only the identified outliers

5. **Isolation Forest ML model with custom threshold to 0.03**

    - a. Scatter plot of the outliers identified by the custom threshold

6. **Data Distribution Reassessment:** utilize box plots post-outlier detection to reassess the data distribution.

7. **Outlier Treatment:** substitute identified outliers with NaN to prepare for the imputation process.
8. **Data Preparation (Econding):** conduct the encoding process to transform categorical data into a machine-readable format.
9. **Imputation Model Testing:** experiment with various Data Imputation ML models to determine the most effective approach.
10. **Imputation Application:** select and apply the chosen model to impute missing values in the dataset.
11. **Post-Imputation Processing (Decoding):** decode the data to revert any encoding done before the imputation, ensuring data is in its original format for interpretation and analysis.

Note:

It is important to note that this model represents my own design development effort and it was achieved through the use of open source tools. While showcasing my personal methodology, this notebook is not intended as an authoritative guidance.

If you're interested on models ready to use or view this work in python, you can find the material at my GitHub (https://github.com/filcode/dataimputation)

SetUp

Importing libraries

```
import requests
import json
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from bs4 import BeautifulSoup
from io import StringIO
import time
import numpy as np
import threading
import klib # custom library (pip install klib) # EDA analysis
```

```
# Make sure pandas version is over or same of 2.1.0
print(pd.__version__)
```

```
2.1.0
```

```
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
pd.options.display.float_format = '{:.2f}'.format
```

Dataframe used

- It's available to anyone (just run the cell below to get it)
- It's about Ratio Table statements. Especially, it collects all listed European companies about "Financial Services" sector which start by "A" and "B" letters. (Source used: MarketWatch)

```
# Get data directly from google drive
import pandas as pd
def read_gdrive_to_df(file_id):
    """
    Read an Excel file from Google Drive using its file ID and return a DataFrame.
    """
    base_url = f'https://drive.google.com/file/d/{file_id}/view?usp=sharing'
```

```python
    download_url = 'https://drive.google.com/uc?id=' + base_url.split('/')[-2]
    return pd.read_excel(download_url, index_col=0)

# File IDs
rt_y_link = '1BeBmlIfq3COpaIidDUm9YYZ_CDiGu-8C' # file code Ratio table
agg_countries_link = '16CUezI0NP4wikyRY_AnKO9p-SNgNpnnX' # file code countries grouping

# Read files using the function
rt_y = read_gdrive_to_df(rt_y_link) # ratio table df with NaN values
aggregated_countries = read_gdrive_to_df(agg_countries_link) # table with countries and values weight for each one
```

```python
# Check columns type
rt_y.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1046 entries, 567 to 957
Data columns (total 18 columns):
 #   Column                     Non-Null Count  Dtype  
---  ------                     --------------  -----  
 0   Sector                     1046 non-null   object 
 1   Super Sector               1046 non-null   object 
 2   Industry                   1046 non-null   object 
 3   Country                    1046 non-null   object 
 4   Exchange                   1046 non-null   object 
 5   Ticker                     1046 non-null   object 
 6   Company Name               1046 non-null   object 
 7   Currency Sign              1046 non-null   object 
 8   Year                       1046 non-null   int64  
 9   Liabilities to Assets      1033 non-null   float64
 10  Debt to Liabilities        936 non-null    float64
 11  Debt to Equity             941 non-null    float64
 12  InterestExp to EBIT        925 non-null    float64
 13  Return On Equity           1037 non-null   float64
 14  Total_Shareholders_Equity  1037 non-null   float64
 15  Operating Margin           1016 non-null   float64
 16  Capex to EBIT              835 non-null    float64
 17  D&A aprox to EBITDA        984 non-null    float64
dtypes: float64(9), int64(1), object(8)
memory usage: 155.3+ KB
```

```python
# Convert to 'category'
# -- 'category' type will help up in future models
def convert_to_category(df, last_col='Year'):
```

```
    cols = df.columns.tolist()

    # Find the index of the specified column
    index = cols.index(last_col)

    # Select columns from start to specified column (included)
    cols_to_convert = cols[:index+1]

    # Convert these columns to 'category' type
    for col in cols_to_convert:
        df[col] = df[col].astype('category')

    return df

rt_y = convert_to_category(rt_y, last_col='Year')
```

---

1. ***Initial Data Cleaning:*** identify and replace initial anomalies values to clean the dataset

(anomalies are identified with the value -9.99)

a. Count -9.99 values in dataset

```
def count_value(df, value):
    total_count = (df == value).sum().sum()

    column_counts = (df == value).sum()
    column_totals = df.count()

    column_percentages = (column_counts / column_totals) * 100

    return total_count, column_percentages

# Call udf
total_count, column_percentages = count_value(rt_y, -9.99)
```

```
total_count
```

```
15
```

```python
# Distribution of outliers on individual columns
column_percentages
```

```
Sector                      0.00
Super Sector                0.00
Industry                    0.00
Country                     0.00
Exchange                    0.00
Ticker                      0.00
Company Name                0.00
Currency Sign               0.00
Year                        0.00
Liabilities to Assets       0.00
Debt to Liabilities         0.00
Debt to Equity              1.59
InterestExp to EBIT         0.00
Return On Equity            0.00
Total_Shareholders_Equity   0.00
Operating Margin            0.00
Capex to EBIT               0.00
D&A aprox to EBITDA         0.00
dtype: float64
```

b. Replace -9.99 with NaN

```python
def replace_with_nan(df, value_to_replace):
    '''
    -9.99 will be replace with NaN
    '''
    return df.replace(value_to_replace, np.nan)

rt_y1 = replace_with_nan(rt_y, -9.99)
```

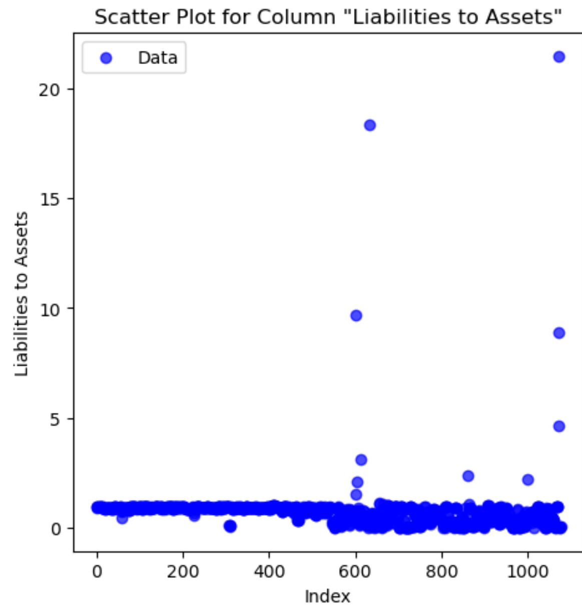
```python
# Check if values replaced
total_count, column_percentages = count_value(rt_y1, -9.99)
total_count
```

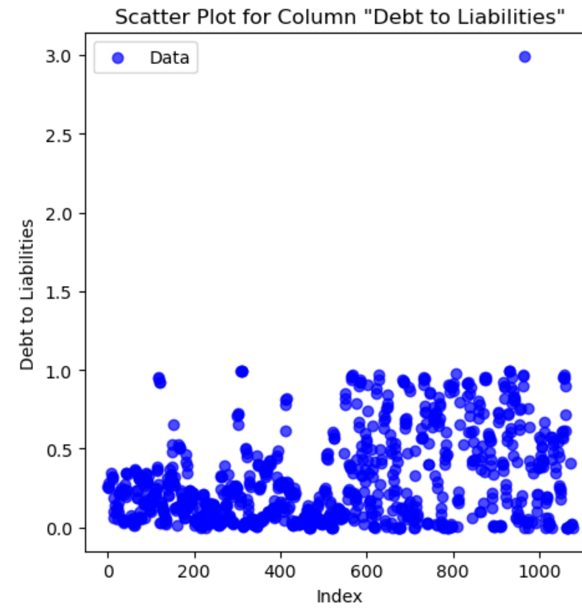
```
0
```

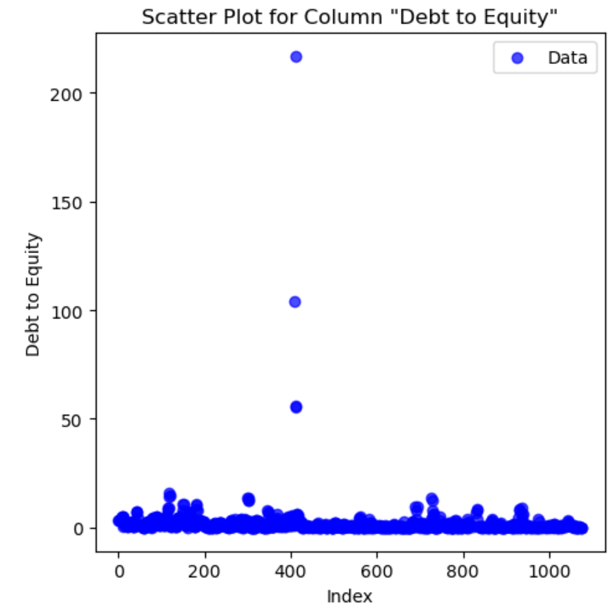
2. ***Visual Analysis:*** generate plots to visualize data distribution and understand the underlying trends

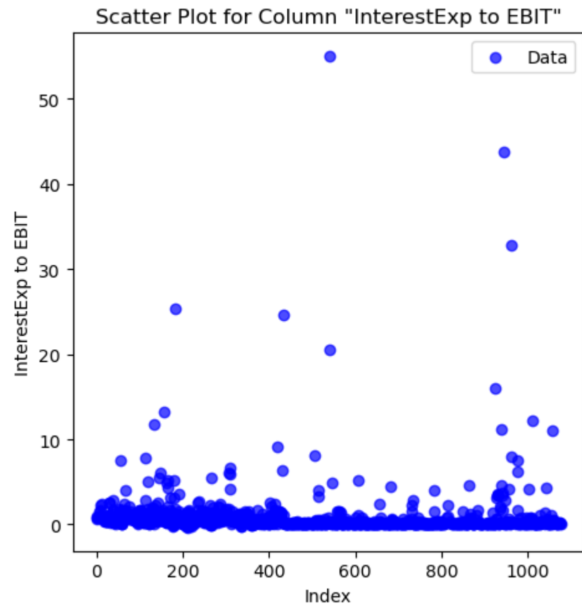
```
import numpy as np
import matplotlib.pyplot as plt

def plot_data_distribution(df, exclude_columns=[], plots_per_row=2):
    df_numeric = df.select_dtypes(include=[np.number])

    # Remove the specified columns from the analysis
    df_numeric = df_numeric.drop(columns=exclude_columns, errors='ignore')

    # Extract all columns without the extension "_outlier"
    original_columns = [col for col in df_numeric.columns if not col.endswith('_outlier')]

    # Calculate the number of rows needed
    num_rows = int(np.ceil(len(original_columns) / plots_per_row))

    # Initialize the subplots
    fig, axes = plt.subplots(num_rows, plots_per_row, figsize=(15, 5 * num_rows))

    if num_rows == 1:
        axes = np.expand_dims(axes, axis=0)

    for idx, column in enumerate(original_columns):
        row = idx // plots_per_row
        col = idx % plots_per_row

        # Get the values of the original column
        original_values = df[column]

        # Plot the values
        axes[row, col].scatter(original_values.index, original_values, color='b', label='Data', alpha=0.7)

        # Add title and legend
        axes[row, col].set_title(f'Scatter Plot for Column "{column}"')
        axes[row, col].set_xlabel('Index')
        axes[row, col].set_ylabel(column)
        axes[row, col].legend()

    # Remove any empty plots
```

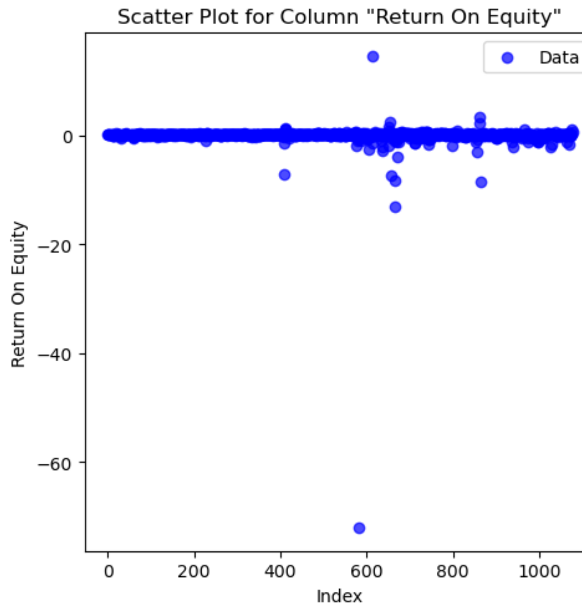
```
    for idx in range(len(original_columns), num_rows * plots_per_row):
        fig.delaxes(axes.flatten()[idx])

    plt.tight_layout()
    plt.show()

# Excluded 'Total_Shareholders_Equity' because it contains just -1 / 1 values
plot_data_distribution(rt_y1, exclude_columns=['Total_Shareholders_Equity'], plots_per_row=3)
```

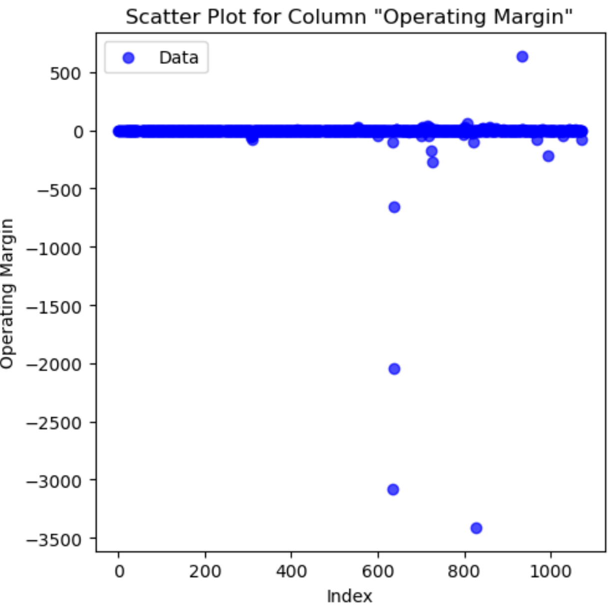

Scatter Plot for Column "Liabilities to Assets"
Data
Liabilities to Assets
Index
0
5
10
15
20
0
200
400
600
800
1000

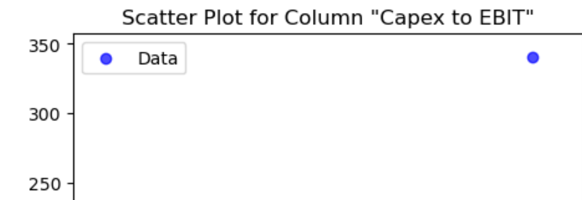

Scatter Plot for Column "Debt to Liabilities"
Data
Debt to Liabilities
Index
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
200
400
600
800
1000

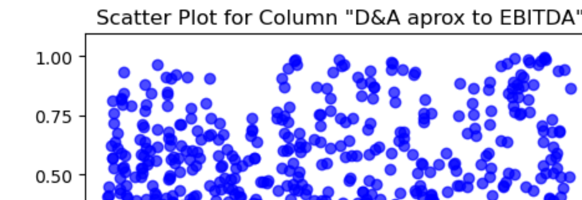

Scatter Plot for Column "Debt to Equity"
Data
Debt to Equity
Index
0
50
100
150
200
0
200
400
600
800
1000

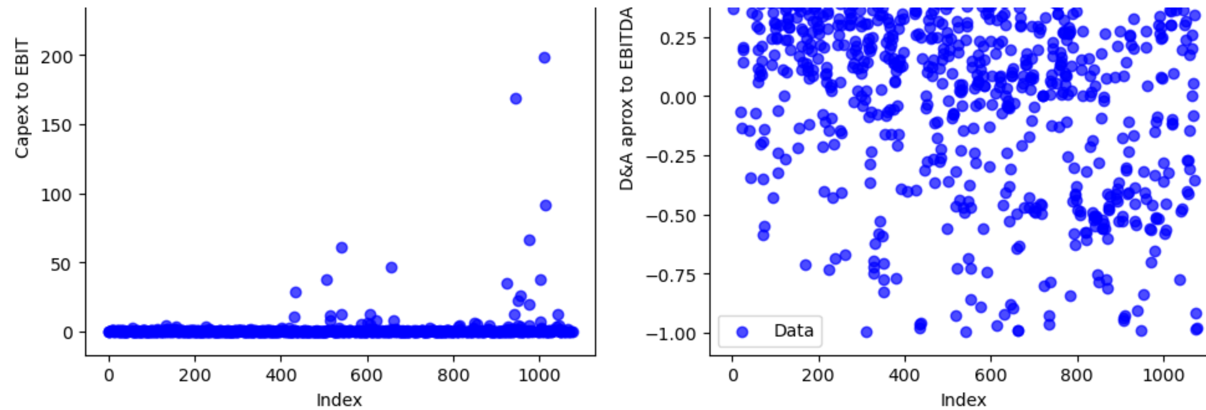

Scatter Plot for Column "InterestExp to EBIT"
Data
InterestExp to EBIT
Index
0
10
20
30
40
50
0
200
400
600
800
1000


Scatter Plot for Column "Return On Equity"
Data
Return On Equity
Index
0
−20
−40
−60
0
200
400
600
800
1000

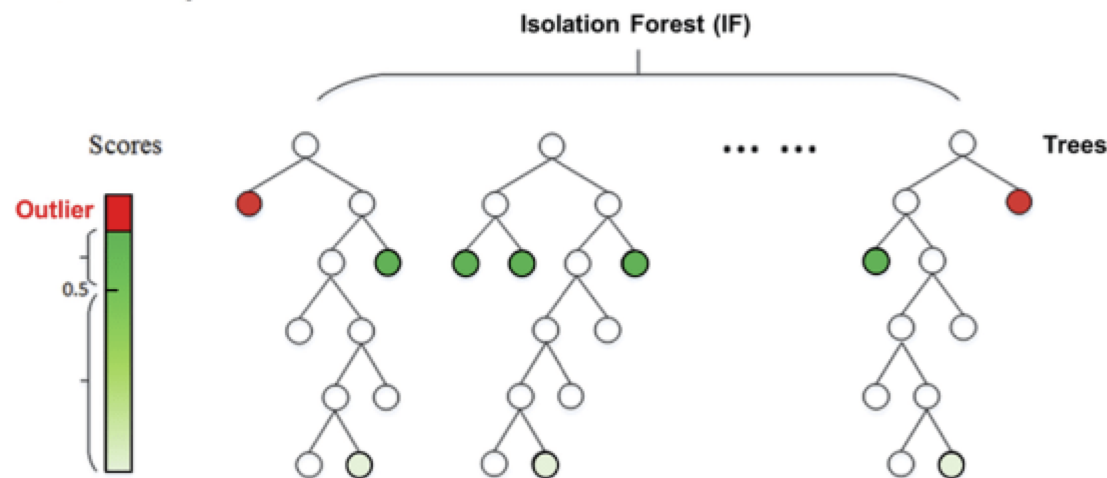

Scatter Plot for Column "Operating Margin"
Data
Operating Margin
Index
500
0
−500
−1000
−1500
−2000
−2500
−3000
−3500
0
200
400
600
800
1000


Scatter Plot for Column "Capex to EBIT"
Data
350
300
250


Scatter Plot for Column "D&A aprox to EBITDA"
1.00
0.75
0.50



Outliers detection with ***Isolation Forest*** Machine learning model



A brief introduction to Isolation Forest and how it works

Isolation Forest is an **unsupervised learning algorithm** that identifies outliers by isolating anomalous observations and not requiring a specific data distribution. The technique is based on the idea that outliers are easier to isolate than normal data points. Instead of measuring distance or density as in other methods, Isolation Forest literally **"isolates" outliers by repeatedly cutting through the data space**.

Steps the model follows for identifying outliers:

1. **Random Selection:** the algorithm randomly selects a feature and then chooses a split value randomly between the maximum and minimum values of that feature.
2. **Partitioning:** using the split value, the data is partitioned into two subsets. This partitioning process is recursively repeated, resulting in a tree structure called an "isolation tree."
3. **Isolation:** outliers will be isolated faster than normal data points because they are "few and different". This means they will have shorter paths in their respective isolation trees.
4. **Forest Construction:** a 'forest' is constructed by creating many isolation trees.
5. **Scoring:** each instance in the dataset is given an anomaly score based on the average length of the paths in the various trees. A shorter path equates to a higher anomaly score, indicating that the instance is more likely to be an outlier.
6. **Evaluation:** after computing scores for all instances, an anomaly threshold is established, above which data points are classified as outliers.
7. **Results:** at the end of the process, outliers are either flagged or removed from the dataset, or they can be further analyzed to understand their anomalous nature.

UDF's will be used following

a.Isolation Forest execution: this UDF is implemented and finalized to identify anomalies

```
from sklearn.ensemble import IsolationForest

def isolation_forest(df,
                     exclude_columns=None,
                     group_column=None,
                     replace_nan_with_zero=False,
                     replace_values_with_zero=None,
                     threshold="auto"):

    if replace_values_with_zero is None:
        replace_values_with_zero = []
```

```
    # Convert categorical columns to string columns
    for col in df.select_dtypes(include=['category']).columns:
        df[col] = df[col].astype('str')

    # Replace specified values and NaN with 0, if requested
    if replace_nan_with_zero:
        df = df.fillna(0)

    for value in replace_values_with_zero:
        df.replace(value, 0, inplace=True)

    if exclude_columns is None:
        exclude_columns = []

    # Select numeric columns, excluding specified columns
    df_numeric = df.select_dtypes(include=[np.number]).drop(columns=exclude_columns)

    # Copy of the original dataframe
    df_total_col = df.copy()

    # Number of trees in Isolation Forest
    n_estimators = 100

    # DataFrame to save the forecasted columns
    outliers_IsoFor = pd.DataFrame(index=df.index)

    # Create an Isolation Forest object
    isolation_forest = IsolationForest(n_estimators=n_estimators, contamination=threshold, random_state=1991)

    if group_column:
        for group_value, group in df.groupby(group_column):
            for column in df_numeric.columns:
                X = group[column].values.reshape(-1, 1)
                outlier_predictions = isolation_forest.fit_predict(X)

                # Save forecasts in the output DataFrame
                outliers_IsoFor.loc[group.index, column + '_outlier'] = outlier_predictions
    else:
        for column in df_numeric.columns:
            X = df[column].values.reshape(-1, 1)
            outlier_predictions = isolation_forest.fit_predict(X)

            # Save forecasts in the output DataFrame
            outliers_IsoFor[column + '_outlier'] = outlier_predictions
```

```
    # Add the expected columns to the original dataframe
    df_iso_for = pd.concat([df_total_col, outliers_IsoFor], axis=1)

    return df_iso_for
```

```
# df_isofor_outliers
```

```
time: 0 ns (started: 2023-09-08 10:17:11 +02:00)
```

b. Dataset view: this UDF will show for each column if values are -1 (outlier) and 1 (not outlier)

```
def iso_for_alternate_columns(df, exclude_columns=[]):
    df_category = df.select_dtypes(exclude=[np.number])
    df_numeric = df.select_dtypes(include=[np.number])

    # Remove specified columns from analysis.
    df_numeric = df_numeric.drop(columns=exclude_columns, errors='ignore')
    df_category = df_category.drop(columns=exclude_columns, errors='ignore')

    # Round column values with the extension '_outlier' to 0 decimal places
    for col in df_numeric.columns:
        if col.endswith('_outlier'):
            df_numeric[col] = df_numeric[col]

    # Create a blank list to store alternate columns
    alternating_columns = []

    # Extract all columns without the extension "_outlier"
    original_columns = [col for col in df_numeric.columns if not col.endswith('_outlier')]

    # Cycle through the original columns
    for column in original_columns:
        # Aggiungi la colonna originale
        alternating_columns.append(column)

        # Add the corresponding "_outlier" column only if it exists
        outlier_column = column + '_outlier'
        if outlier_column in df.columns:
            alternating_columns.append(outlier_column)

    # Create a new DataFrame using the desired column order
    alternating_df = df[alternating_columns]
```

```
    df_iso_for = pd.concat([df_category, alternating_df], axis=1)

    def style_outliers(data):
        styles = ['font-weight: bold' if (val == -1 and '_outlier' in col) else '' for val, col in zip(data, data.index)]
        return styles

    # Apply the styling function to the entire DataFrame
    df_iso_for_styled = df_iso_for.style.apply(style_outliers, axis=1)
    display(df_iso_for_styled)

    return df_iso_for
```

```
pd.options.display.float_format = '{:.2f}'.format
# df_iso_for = iso_for_alternate_columns(df_isofor_outliers, exclude_columns=['Total_Shareholders_Equity'])
```

---

3. ***Isolation Forest ML with default threshold to detect outliers***

```
df_isofor_outliers = isolation_forest(rt_y1,
                     exclude_columns='Total_Shareholders_Equity',
                     group_column=None,
                     replace_nan_with_zero=True,
                     replace_values_with_zero=None,
                     threshold="auto")
```

4. ***Graphical Outliers Representation***

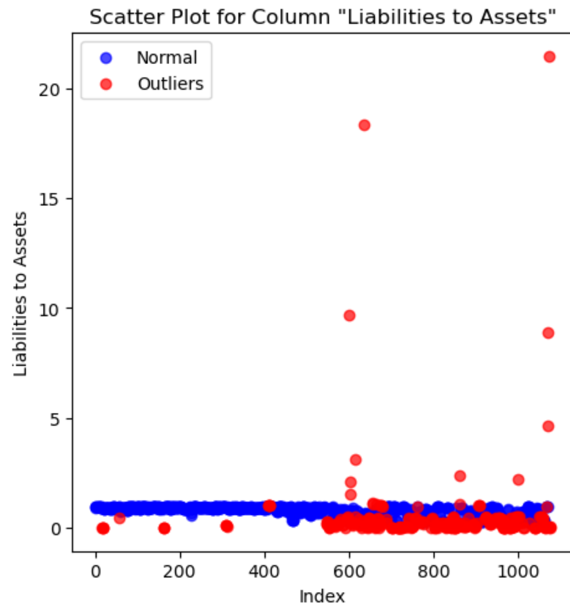
a. ***Scatter plot of the outliers identified by the isolation forest***

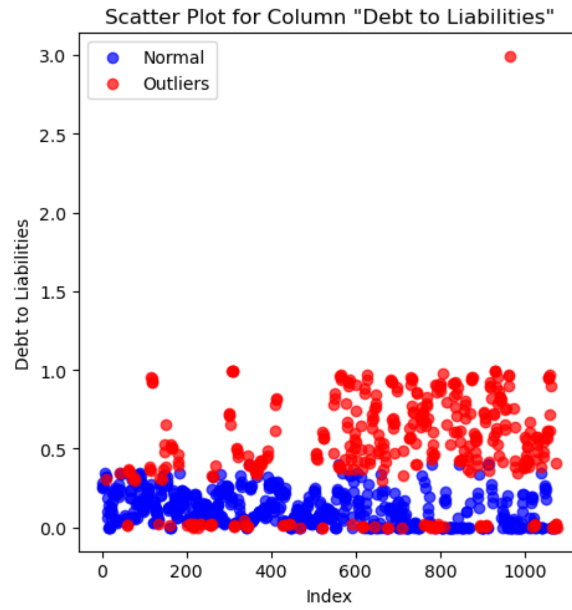
(identified outliers are highlighted in red)

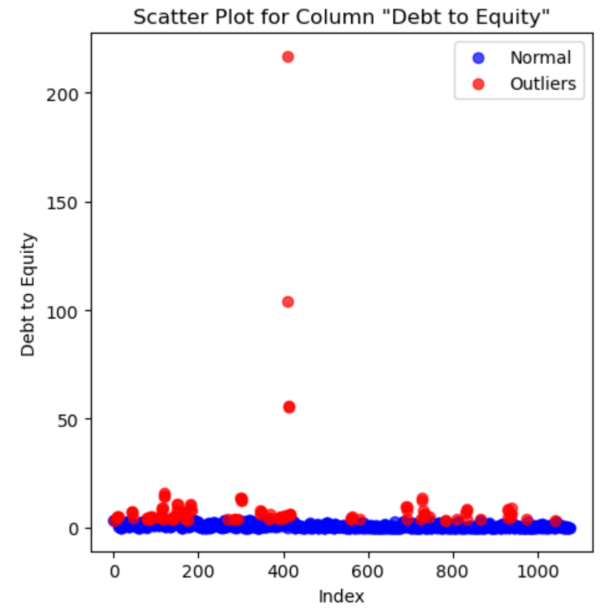
```
import matplotlib.pyplot as plt
import numpy as np

# Graph data distribution and outliers highlighted
def plot_outliers_scatter(df, exclude_columns=[], plots_per_row=2):
    df_numeric = df.select_dtypes(include=[np.number])
```

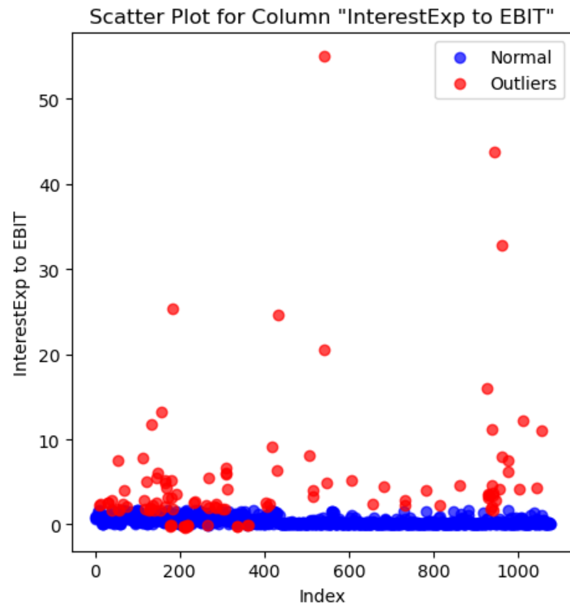
```
# Remove the specified columns from the analysis
df_numeric = df_numeric.drop(columns=exclude_columns, errors='ignore')

# Extract all columns without the extension "_outlier"
original_columns = [col for col in df_numeric.columns if not col.endswith('_outlier')]

# Calculate the number of rows needed
num_rows = int(np.ceil(len(original_columns) / plots_per_row))

# Initialize the subplots
fig, axes = plt.subplots(num_rows, plots_per_row, figsize=(15, 5 * num_rows))

if num_rows == 1:
    axes = np.expand_dims(axes, axis=0)

for idx, column in enumerate(original_columns):
    row = idx // plots_per_row
    col = idx % plots_per_row

    # Check if the corresponding "_outlier" column exists
    outlier_column = column + '_outlier'
    if outlier_column in df.columns:

        #  Get original column and outlier column values
        original_values = df[column]
        outlier_values = df[outlier_column]

        # Mask for values and outliers
        mask_normal = outlier_values == 1
        mask_outliers = outlier_values == -1

        # Plot normal values
        axes[row, col].scatter(original_values[mask_normal].index, original_values[mask_normal], color='b', label='Normal', a

        # Plots outliers
        axes[row, col].scatter(original_values[mask_outliers].index, original_values[mask_outliers], color='r', label='Outlie

        # Add title and legend
        axes[row, col].set_title(f'Scatter Plot for Column "{column}"')
        axes[row, col].set_xlabel('Index')
        axes[row, col].set_ylabel(column)
        axes[row, col].legend()
```

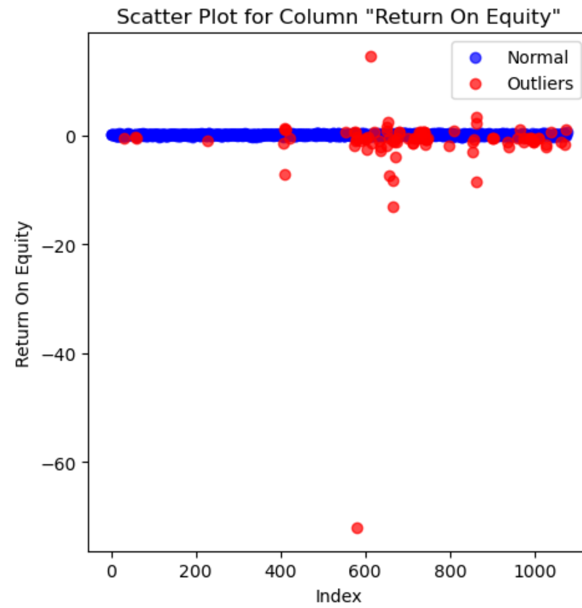
```
    # Remove empty plots
    for idx in range(len(original_columns), num_rows * plots_per_row):
        fig.delaxes(axes.flatten()[idx])

    plt.tight_layout()
    plt.show()

# Call
plot_outliers_scatter(df_isofor_outliers, exclude_columns=[], plots_per_row=3)
```

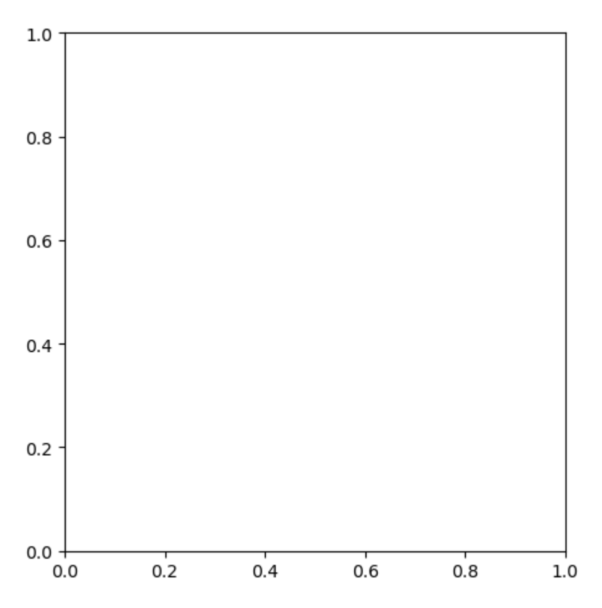

Scatter Plot for Column "Liabilities to Assets"
Normal
Outliers
Liabilities to Assets
Index

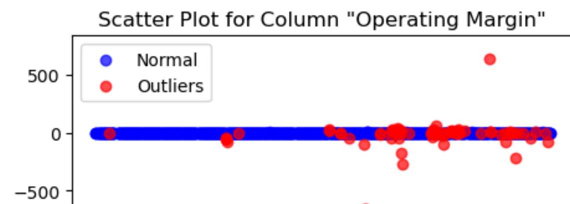

Scatter Plot for Column "Debt to Liabilities"
Normal
Outliers
Debt to Liabilities
Index

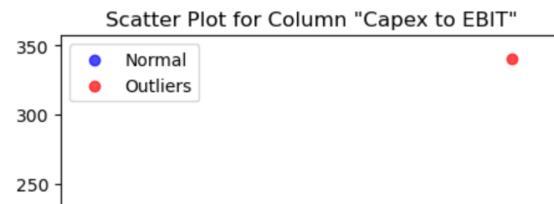

Scatter Plot for Column "Debt to Equity"
Normal
Outliers
Debt to Equity
Index

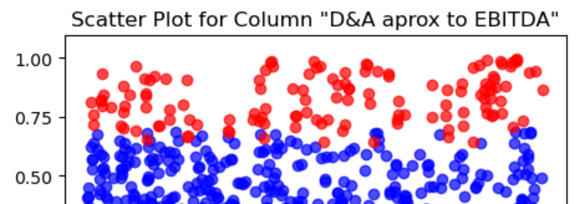

Scatter Plot for Column "InterestExp to EBIT"
Normal
Outliers
InterestExp to EBIT
Index

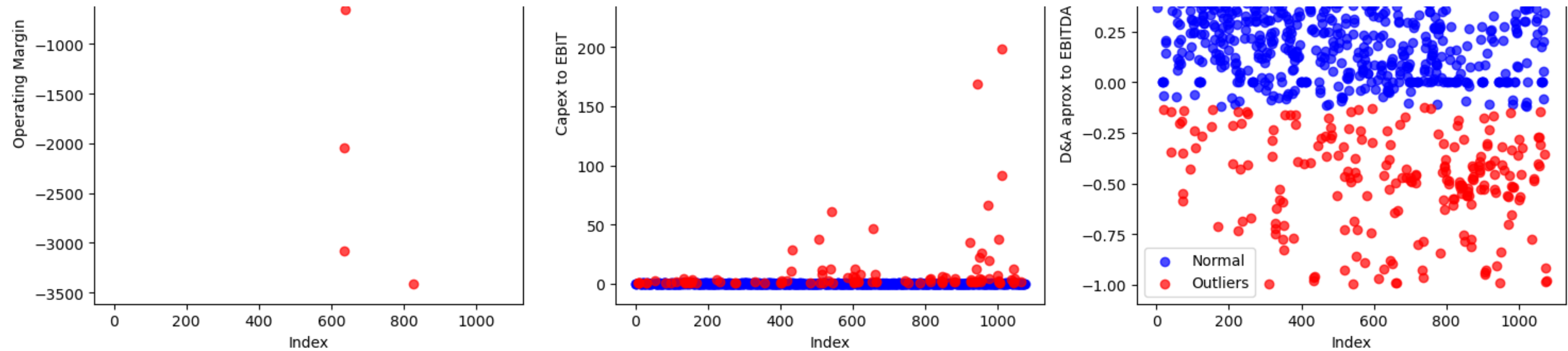

Scatter Plot for Column "Return On Equity"
Normal
Outliers
Return On Equity
Index




Scatter Plot for Column "Operating Margin"
Normal
Outliers


Scatter Plot for Column "Capex to EBIT"
Normal
Outliers


Scatter Plot for Column "D&A aprox to EBITDA"

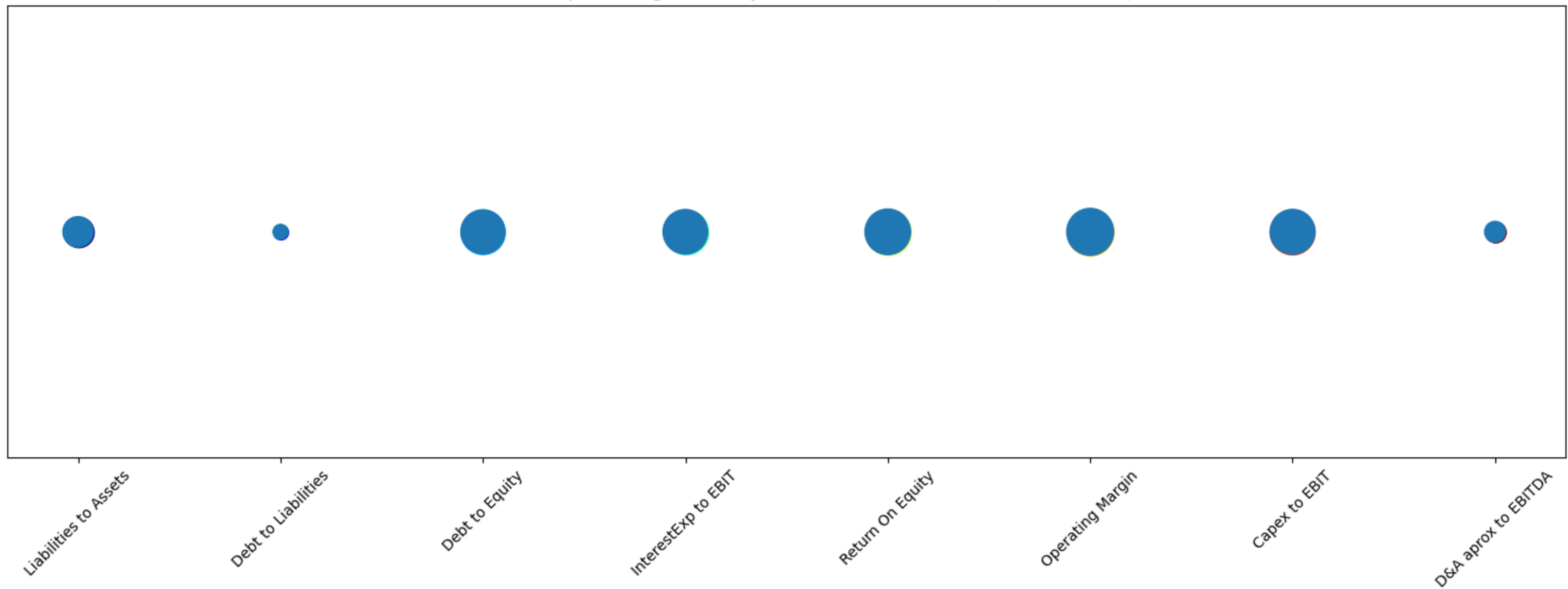


b. ***Bubble graph for a simple and immediate visualization of the most concentrated outlier areas***

```
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

def outliers_df_outliers_presence(df, exclude_columns=[]):
    df_numeric = df.select_dtypes(include=[np.number])

    # Remove the specified columns from the analysis
    df_numeric = df_numeric.drop(columns=exclude_columns, errors='ignore')

    # Calculate the size of bubbles
    nan_count = df_numeric.isna().sum()
    not_nan_count = df_numeric.count()
    total_count = nan_count + not_nan_count
    bubble_sizes = total_count - not_nan_count

    # --- TABLE
    print('Percentage outliers found for each column')
    fraction_outliers = not_nan_count/total_count
    display(fraction_outliers.round(2))

    # Show the overall average
    mean_outliers = fraction_outliers.mean()
    print(f'Average outliers found: {mean_outliers:.2f}')

    # --- PLOT
```

```
    # Resize bubble sizes in a specific range (e.g. 100-1000)
    scaler = MinMaxScaler(feature_range=(100, 1000))
    bubble_sizes_scaled = scaler.fit_transform(bubble_sizes.values.reshape(-1, 1))

    # Create a Bubble Chart
    plt.figure(figsize=(15, 6))
    colors = plt.cm.jet(np.linspace(0, 1, len(bubble_sizes))) # Color map

    for idx, size in enumerate(bubble_sizes_scaled):
        plt.scatter(idx, 1, s=size, color=colors[idx], label=bubble_sizes.index[idx]) # Bubble with unique color

    # Create graph
    plt.scatter(range(len(bubble_sizes)), [1] * len(bubble_sizes), s=bubble_sizes_scaled)
    plt.xticks(range(len(bubble_sizes)), bubble_sizes.index, rotation=45) Labels for the x axis
    plt.yticks([]) # Removes ticks on the y-axis
    plt.title('Bubble Plot Representing how many outliers has each column (total - not nan)')
    plt.tight_layout() # Ensures Labels are not cut
    plt.show()

# Call
outliers_df_outliers_presence(df_outlier, exclude_columns=[])
```

```
Percentage outliers found for each column
Liabilities to Assets   0.26
Debt to Liabilities     0.37
Debt to Equity          0.11
InterestExp to EBIT     0.10
Return On Equity        0.09
Operating Margin        0.07
Capex to EBIT           0.10
D&A aprox to EBITDA     0.34
dtype: float64
Average outliers found: 0.18
```



c. ***Dataframe showing only the identified outliers***

```
def outliers_df_iso_for(df, exclude_columns=None):
    if exclude_columns is None:
        exclude_columns = []

    outlier_columns = [col for col in df.columns if col.endswith('_outlier') and col not in exclude_columns]

    # Loop through outlier column indexes
    for col in outlier_columns:
        original_col_idx = df.columns.get_loc(col) - 1 # Index to previous column
        original_col = df.columns[original_col_idx] # Name of previous column

        if original_col not in exclude_columns:
            mask = df[col] == 1 # Create a mask where the values in the outlier column are 1
            df.loc[mask, original_col] = np.nan # Replace the corresponding values in the original column with Nan

    # Copy the dataframe and remove columns with the extension "_outlier"
    df_outlier = df.copy()
```

```
    df_outlier = df_outlier.drop(columns=outlier_columns)

    return df_outlier

df_outlier = outliers_df_iso_for(df_iso_for, exclude_columns=[])
```

```
df_outlier.head()
```

	Sector	Super Sector	Industry	Country	Exchange	Ticker	Company Name	Currency Sign	Year	Liabilities to Assets	Debt to Liabilities	Debt to Equity	InterestExp to EBIT	Return On Equity	C
567	Financial Services	Investment Advisors	Investing/Securities	France	XPAR	ABCA	ABC Arbitrage S.A.	€	2019	0.04	NaN	NaN	NaN	NaN	
568	Financial Services	Investment Advisors	Investing/Securities	France	XPAR	ABCA	ABC Arbitrage S.A.	€	2020	0.11	NaN	NaN	NaN	NaN	
569	Financial Services	Investment Advisors	Investing/Securities	France	XPAR	ABCA	ABC Arbitrage S.A.	€	2021	0.14	NaN	NaN	NaN	NaN	
570	Financial Services	Investment Advisors	Investing/Securities	France	XPAR	ABCA	ABC Arbitrage S.A.	€	2022	0.14	NaN	NaN	NaN	NaN	
571	Financial Services	Investment Advisors	Investing/Securities	Norway	XOSL	ABG	ABG Sundal Collier Holding ASA	kr	2019	NaN	NaN	NaN	NaN	NaN	

5. ***Isolation Forest ML model with custom threshold to 0.03***

```
# NaN values sum count
to_start_col = rt_y1.columns.get_loc('Year')
rt_y1.iloc[:, to_start_col+1:].isna().sum()/len(rt_y1)
```

```
Liabilities to Assets       0.01
Debt to Liabilities         0.11
Debt to Equity              0.11
InterestExp to EBIT         0.12
Return On Equity            0.01
Total_Shareholders_Equity   0.01
Operating Margin            0.03
Capex to EBIT               0.20
D&A aprox to EBITDA         0.06
dtype: float64
```

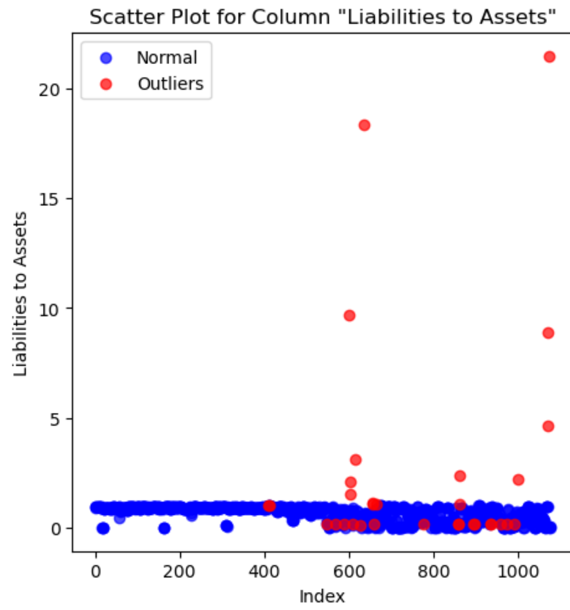
```
df_isofor_outliers1 = isolation_forest(rt_y1,
                     exclude_columns='Total_Shareholders_Equity',
                     group_column=None,
                     replace_nan_with_zero=True,
                     replace_values_with_zero=None,
                     threshold=0.03) # reset
```

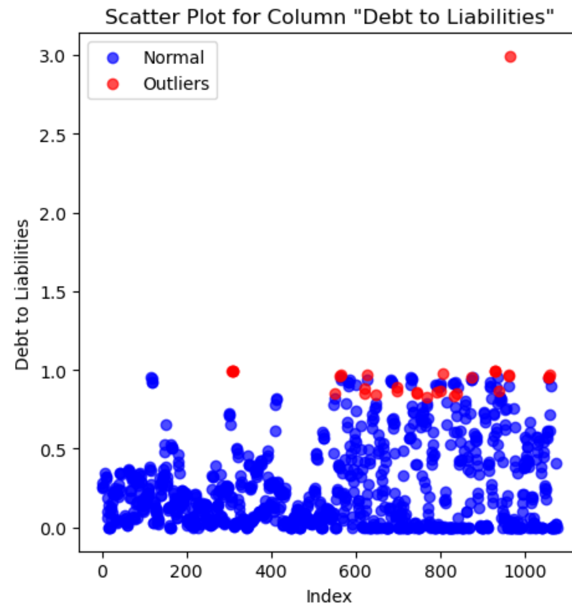
```
# data modelling (column and columns + '_outlier')
# df_iso_for1 = iso_for_alternate_columns(df_isofor_outliers1, exclude_columns=['Total_Shareholders_Equity'])
```

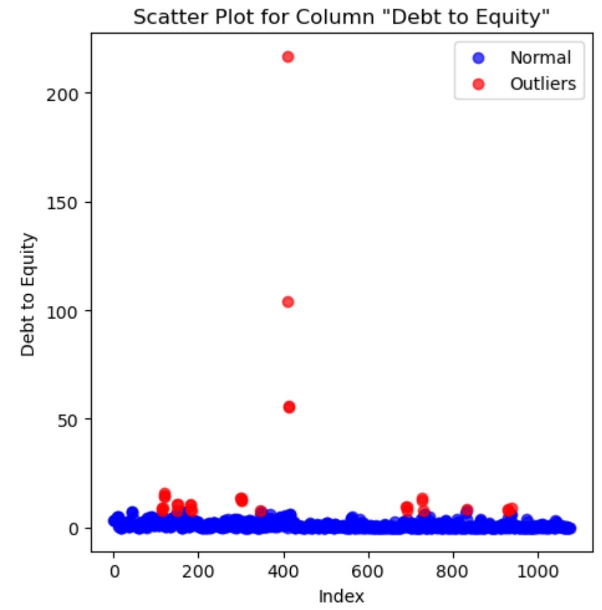
a. ***Scatter plot of the outliers identified by the custom threshold***

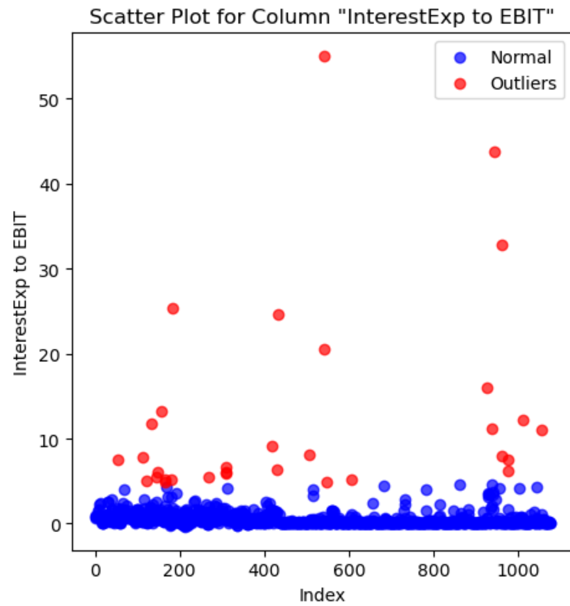
(identified outliers are highlighted in red)

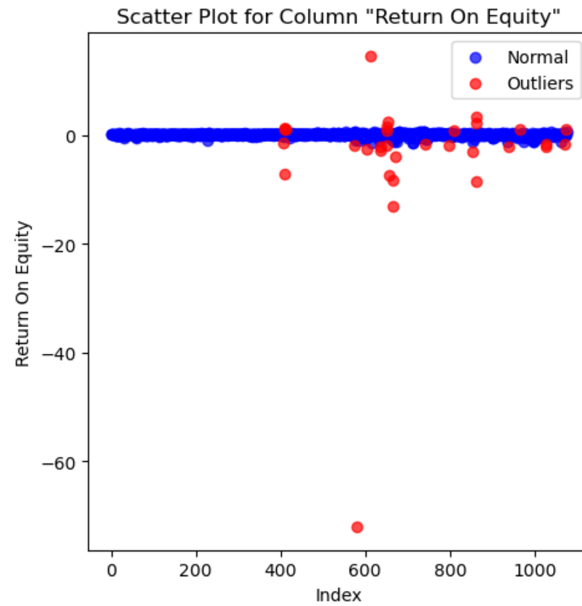
```
plot_outliers_scatter(df_isofor_outliers1, exclude_columns=[], plots_per_row=3)
```

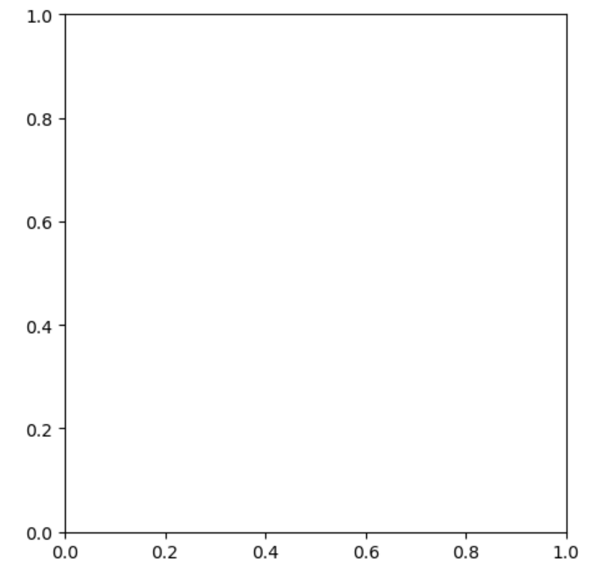

Scatter Plot for Column "Liabilities to Assets"
Normal
Outliers
Liabilities to Assets
Index

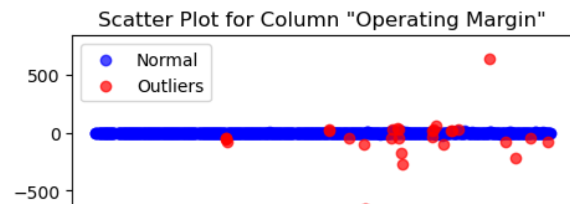

Scatter Plot for Column "Debt to Liabilities"
Normal
Outliers
Debt to Liabilities
Index

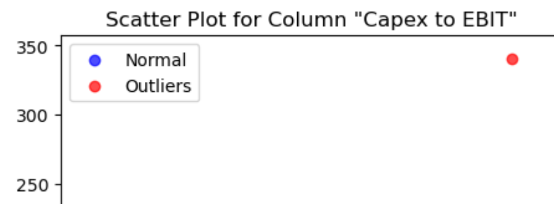

Scatter Plot for Column "Debt to Equity"
Normal
Outliers
Debt to Equity
Index

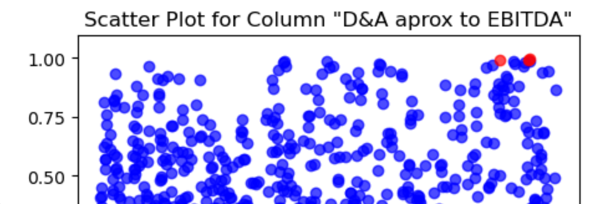

Scatter Plot for Column "InterestExp to EBIT"
Normal
Outliers
InterestExp to EBIT
Index

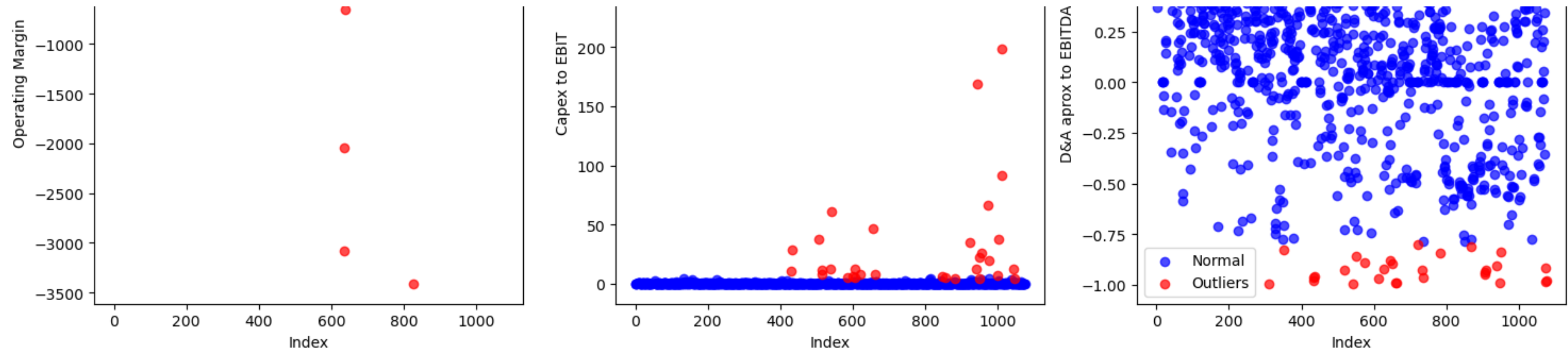

Scatter Plot for Column "Return On Equity"
Normal
Outliers
Return On Equity
Index




Scatter Plot for Column "Operating Margin"
Normal
Outliers


Scatter Plot for Column "Capex to EBIT"
Normal
Outliers


Scatter Plot for Column "D&A aprox to EBITDA"




```
# Leave just values and replace others with nan
df_outlier1 = outliers_df_iso_for(df_iso_for1, exclude_columns=['Total_Shareholders_Equity'])
```

6. ***<u>Data Distribution Reassessment:</u>*** utilize box plots post-outliers detection to reassess the data distribution

<u>Explanation of following UDF:</u>

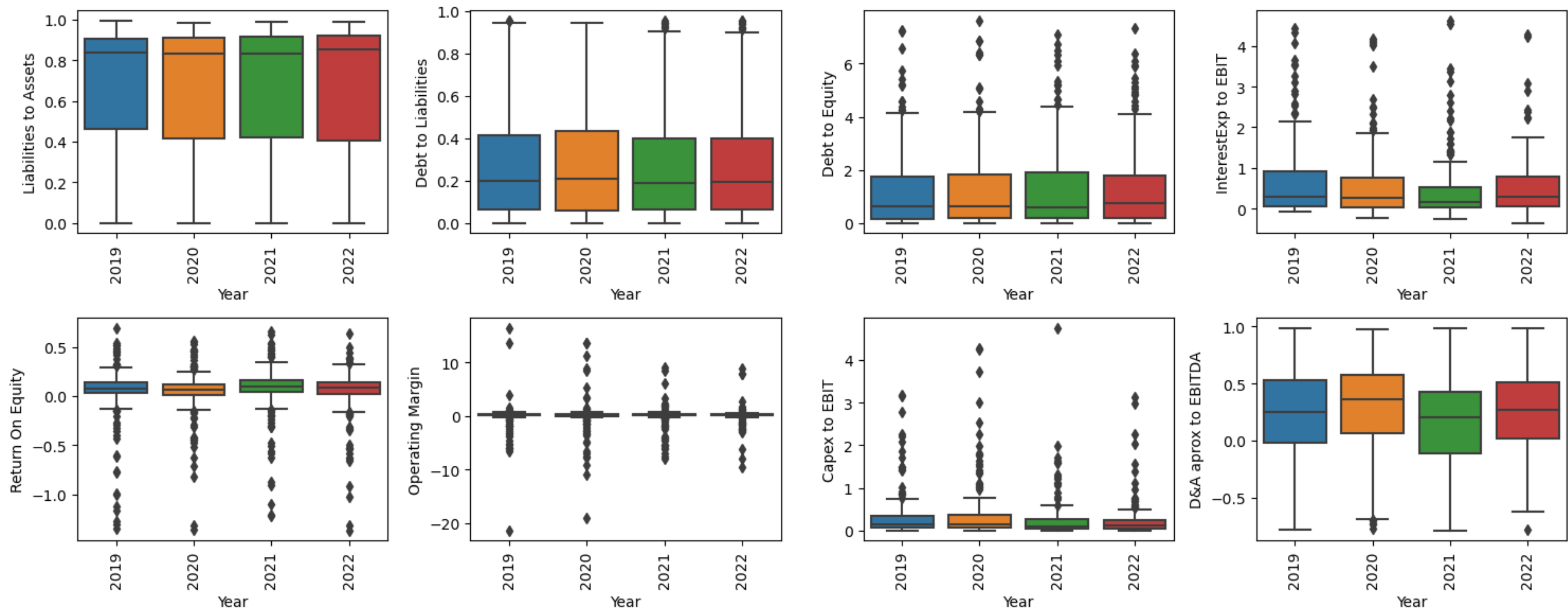
1. ***DataFrame Copy Creation:*** function starts by creating a copy of the original DataFrame to avoid modifying the original data.
2. ***Handling the Time Column:*** time column (specified as time_column) is converted to a category, which can be useful for sorting the data during plotting.
3. ***User Selection:*** user is prompted to choose whether they want to view the data with or without outliers ('Outliers' or 'No Outliers').
4. ***Removing Outliers:*** if the user chooses 'No Outliers', the function proceeds to remove the outliers. This is done using a mask that identifies non-outlier values (False) in numerical columns and retains them, while replacing outlier values with NaN.
5. ***Selecting Columns:*** all numerical columns except for time_column and those specified in exclude_columns are selected. These columns will be used for the boxplots.
6. ***Creating Boxplots:*** for each selected numerical column, the function creates a boxplot that shows the distribution of values over time.
7. ***Handling Missing Data:*** if a column contains no values, the function prints a warning message and does not display the corresponding boxplot.

8. ***Display Layout:*** the boxplots are arranged in a grid of subplots determined by the number of columns (ncols) and the calculated number of rows (nrows). The layout is adjusted to prevent overlap, and each boxplot displays the data relative to the time column.

```
import math
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

def Boxplot_DataDistribution(time_column, df, df_outliers, exclude_columns=[]):

    df_copy = df.copy()
    df_copy[time_column] = df_copy[time_column].astype('category')

    # Option plotting
    selection = None
    while selection not in ['Outliers', 'No Outliers']:
        selection = input('Outliers or No Outliers: ')
        if selection not in ['Outliers', 'No Outliers']:
            print('Try again with options available')

    # Remove outliers for 'No Outliers'
    if selection == 'No Outliers':
        mask = df_outliers.select_dtypes(include=[np.number]).notna()
        for col in mask.columns:
            if col in df_copy.columns and col != time_column and col not in exclude_columns:
                df_copy[col] = df_copy[col].where(mask[col] == False)


    selected_columns = df_copy.select_dtypes(include=[np.number]).drop(columns=[time_column] + exclude_columns, errors='ignore').

    # Select all columns except "year"
    n = len(selected_columns)
    ncols = 4
    nrows = math.ceil(n / ncols)  # Calculate the number of rows needed

    fig, axs = plt.subplots(nrows, ncols, figsize=(15, 3*nrows))  # Increase vertical size based on number of rows

    for i, ax in enumerate(axs.flat):
        if i < n:
            x_data = df_copy[time_column]
            y_data = df_copy[selected_columns[i]]
```

```
            if x_data.dropna().shape[0] == 0:
                print(f"x_data in column {selected_columns[i]} contains no values")
            if y_data.dropna().shape[0] == 0:
                print(f"y_data in column {selected_columns[i]} contains no values")

            if x_data.dropna().shape[0] > 0 and y_data.dropna().shape[0] > 0:
                sns.boxplot(x=x_data, y=y_data, ax=ax)
                ax.set_xticklabels(ax.get_xticklabels(), rotation=90)
            else:
                ax.set_visible(False)

    plt.tight_layout()  # Adjust layout to avoid overlap
    plt.show()

    return

# Select dataframe which include: Outliers or No Outliers
Boxplot_DataDistribution('Year', rt_y1, df_outlier1, exclude_columns=['Total_Shareholders_Equity'])
```

```
Outliers or No Outliers: No Outliers
```



7. ***Outliers Treatment:*** substitute identified outliers with NaN to prepare for the imputation process

```
# make copy of dataframe
rt_y2 = rt_y1.copy()
```

```
# dataframe passed
to_start_col = rt_y2.columns.get_loc('Year')
rt_y2.iloc[:, to_start_col+1:].isna().sum()/len(rt_y2)
```

```
Liabilities to Assets       0.01
Debt to Liabilities         0.11
Debt to Equity              0.11
InterestExp to EBIT         0.12
Return On Equity            0.01
Total_Shareholders_Equity   0.01
Operating Margin            0.03
Capex to EBIT               0.20
D&A aprox to EBITDA         0.06
dtype: float64
```

```
def replace_values_with_nan(main_df, outliers_df, exclude_columns=[], include_category=False):

    # Get 'category' column list
    category_columns = main_df.select_dtypes(include=['category']).columns.tolist()

    if not include_category:
        exclude_columns = list(set(exclude_columns + category_columns))

    # Get the list of numeric columns
    numeric_columns = main_df.select_dtypes(include=[np.number]).columns.tolist()

    for col in numeric_columns:
        if col not in exclude_columns:
            # Converte la Colonna al tipo di Dati float64
            main_df[col] = main_df[col].astype('float64')
            main_df[col] = main_df[col].where(outliers_df[col].isna(), np.nan)

    return main_df

rt_y2 = replace_values_with_nan(rt_y2, df_outlier1, exclude_columns=['Total_Shareholders_Equity'], include_category=False)
```

```
# percetage of nan + outliers replaces with nan
to_start_col = rt_y2.columns.get_loc('Year')
rt_y2.iloc[:, to_start_col+1:].isna().sum()/len(rt_y2)
```

```
Liabilities to Assets       0.04
Debt to Liabilities         0.14
Debt to Equity              0.15
InterestExp to EBIT         0.15
Return On Equity            0.04
Total_Shareholders_Equity   0.01
Operating Margin            0.06
Capex to EBIT               0.23
D&A aprox to EBITDA         0.09
dtype: float64
```

Check NaN distribution in the dataset

```
import numpy as np
import pandas as pd
from IPython.display import display

def analyze_nan_values(df, column_filter=None, threshold=0.3):
    """
    Analyze Nan values in DataFrame.

    :param df: The DataFrame to analyze
    :param column_filter: The column name to filter through in step b. If None, all numeric columns are parsed. (default=None)
    :param Threshold: The percentage threshold of Nan values to trigger the alert. (default=0.3)
    """

    # Step a)
    print('ddc meaning: data, columns, countries\n')
    print('a): Are NaN over specified percentage of total data?')
    total_values = [df.select_dtypes(include=[np.number]).size]
    total_nan_values = [df.isna().sum().sum()]
    percent_nan_values = [total_nan_values[0] / total_values[0] * 100]
    threshold_over = ['Yes' if total_nan_values[0] / total_values[0] > threshold else 'No']
    df_a = pd.DataFrame({'Total values': total_values, 'Total NaN values': total_nan_values, '% NaN values': [f'{x:.0f}%' for x i
    df_data = display(df_a)

    # Step b)
    print('\nb) Are NaN over specified percentage in some columns?')
    if column_filter:
        numeric_columns = [column_filter]
    else:
        numeric_columns = df.select_dtypes(include=[np.number]).columns
```

```
    col_names = []
    total_values_b = []
    total_nan_values_b = []
    percent_nan_values_b = []
    threshold_over_b = []
    for col in numeric_columns:
        col_names.append(col)
        total_values_col = df[col].size
        total_values_b.append(total_values_col)
        nan_values = df[col].isna().sum()
        total_nan_values_b.append(nan_values)
        percent_nan_values_b.append(nan_values / total_values_col * 100)
        threshold_over_b.append('Yes' if nan_values / total_values_col > threshold else 'No')

    df_b = pd.DataFrame({'Column name': col_names, 'Total values': total_values_b, 'Total NaN values': total_nan_values_b, '% NaN
    df_columns = display(df_b)

# Use function:
analyze_nan_values(rt_y2, column_filter=None, threshold=0.3)
```

```
ddc meaning: data, columns, countries

a): Are NaN over specified percentage of total data?
```

	Total values	Total NaN values	% NaN values	Threshold over
0	9414	941	10%	No

```
b) Are NaN over specified percentage in some columns?
```

	Column name	Total values	Total NaN values	% NaN values	Threshold over
0	Liabilities to Assets	1046	45	4%	No
1	Debt to Liabilities	1046	142	14%	No
2	Debt to Equity	1046	152	15%	No
3	InterestExp to EBIT	1046	153	15%	No
4	Return On Equity	1046	41	4%	No
5	Total_Shareholders_Equity	1046	9	1%	No
6	Operating Margin	1046	62	6%	No
7	Capex to EBIT	1046	243	23%	No
8	D&A aprox to EBITDA	1046	94	9%	No

8. ***Data Preparation (Econding):*** conduct the encoding process to transform categorical data into a machine-readable format

```
# rt_y copied
rt_y3 = rt_y2.copy()
```

a. Count unique values inside each subsector

```
# Check potentially subgroups which have enough to forecast
rt_y3['Industry'].value_counts()
```

```
Industry
Investing/Securities    520
Banking/Credit          418
Insurance               108
Name: count, dtype: int64
```

b. Column selection for encoding

Objective: a data preprocessing procedure that isolates specific columns in a DataFrame for subsequent encoding. Its purpose is to prepare the dataset for machine learning models by selecting only the relevant columns that require transformation from categorical to numerical data.

Columns selected: Industry, Country, Year

```
def filter_columns(df, keep_columns, start_column):
    # Create a list of columns to keep based on the specified start point
    columns_from_start = df.columns[df.columns.get_loc(start_column):].tolist()

    # We merge the two column lists
    final_columns = keep_columns + columns_from_start

    # Filter the DataFrame to keep only the desired columns
    return df[final_columns]
```

```
rt_y3 = filter_columns(rt_y3, ['Industry', 'Country'], 'Year')
rt_y3 = rt_y3.sort_values(by='Year', ascending=True)
```

```
# rt_y3.head()
```

c. Econding

- **Industry:** One-Hot Encoding
- **Country:** implemented a custom macroeconomic analysis to assign each country to a specific values. This values reflects the economic conditions of the country where a company is headquartered. (You can custom by yourself this process)
- **Year:** Ordinal Encoding

Upload 'aggregated_countries' df. It will be used to replace Country variables columns

```
def read_gdrive_to_df(file_id):
    """
    Read an Excel file from Google Drive using its file ID and return a DataFrame.
    """
    base_url = f'https://drive.google.com/file/d/{file_id}/view?usp=sharing'
    download_url = 'https://drive.google.com/uc?id=' + base_url.split('/')[-2]
    return pd.read_excel(download_url, index_col=0)

# File IDs
aggregated_countries_link = '1t-lU_WfxN2ddB-yLKq2em3A6HGiQcuCj'
```

```
# Read files using the function
aggregated_countries = read_gdrive_to_df(aggregated_countries_link) # mean distance from year for each country
```

1. Industry: One-Hot Encoding

```
from sklearn.preprocessing import OneHotEncoder

def one_hot_encode(df, column_name, prefix=None):

    if prefix is None:
        prefix = column_name

    df = pd.concat([df, pd.get_dummies(df[column_name], prefix=prefix)], axis=1)
    df.drop(column_name, axis=1, inplace=True)

    # List of one-hot columns created
    one_hot_columns = [col for col in df.columns if prefix in col and col != column_name]

    # Replace true and false with 0 and 1
    for col in one_hot_columns:
        df[col] = df[col].astype(int)

    return df

rt_y3 = one_hot_encode(rt_y3, 'Industry', 'Industry')
```

2. Countries: mapping by values showed in table below (!! this approach can be change leaving just countries instead of values)

```
# Table showing values assign to each country based by my own macroeconomic analysis
aggregated_countries.sort_values(by='TotPtsQle', ascending=False)
```

	TotPtsQle	Number	Countries
16	29	1	Germany
15	26	1	Belgium
14	25	1	Netherlands
13	23	1	Switzerland
12	19	3	Austria, Denmark, Luxembourg
11	18	1	Malta
10	17	2	Cyprus, Sweden
9	16	4	Greece, Ireland, Italy, Portugal
8	15	2	Czechia, Poland
7	14	4	France, Hungary, Norway, Spain
6	12	1	Iceland
5	11	2	Lithuania, Romania
4	10	1	Latvia
3	9	1	Slovakia
2	8	1	Slovenia
1	7	3	Bulgaria, Croatia, Estonia
0	6	1	Finland

```
country_mapping = {}

# Populating of the mapping dictionary
for index, row in aggregated_countries.iterrows():
    countries = row['Countries'].split(", ")
    for country in countries:
        country_mapping[country] = row['TotPtsQle']

# Map values in the rt_y1 'Country' column using the dictionary
rt_y3['Country'] = rt_y3['Country'].map(country_mapping)
```

3. Year: Ordinal Encoding

```
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder

# If they are not, you can specify the order with the 'categories' parameter of the OrdinalEncoder
def ordinal_encode(df, column_name):

    oe = OrdinalEncoder(categories=[sorted(df[column_name].unique())])
    df[column_name] = oe.fit_transform(df[[column_name]]).astype(int)

    return df, oe

rt_y3, oe = ordinal_encode(rt_y3, 'Year')
```

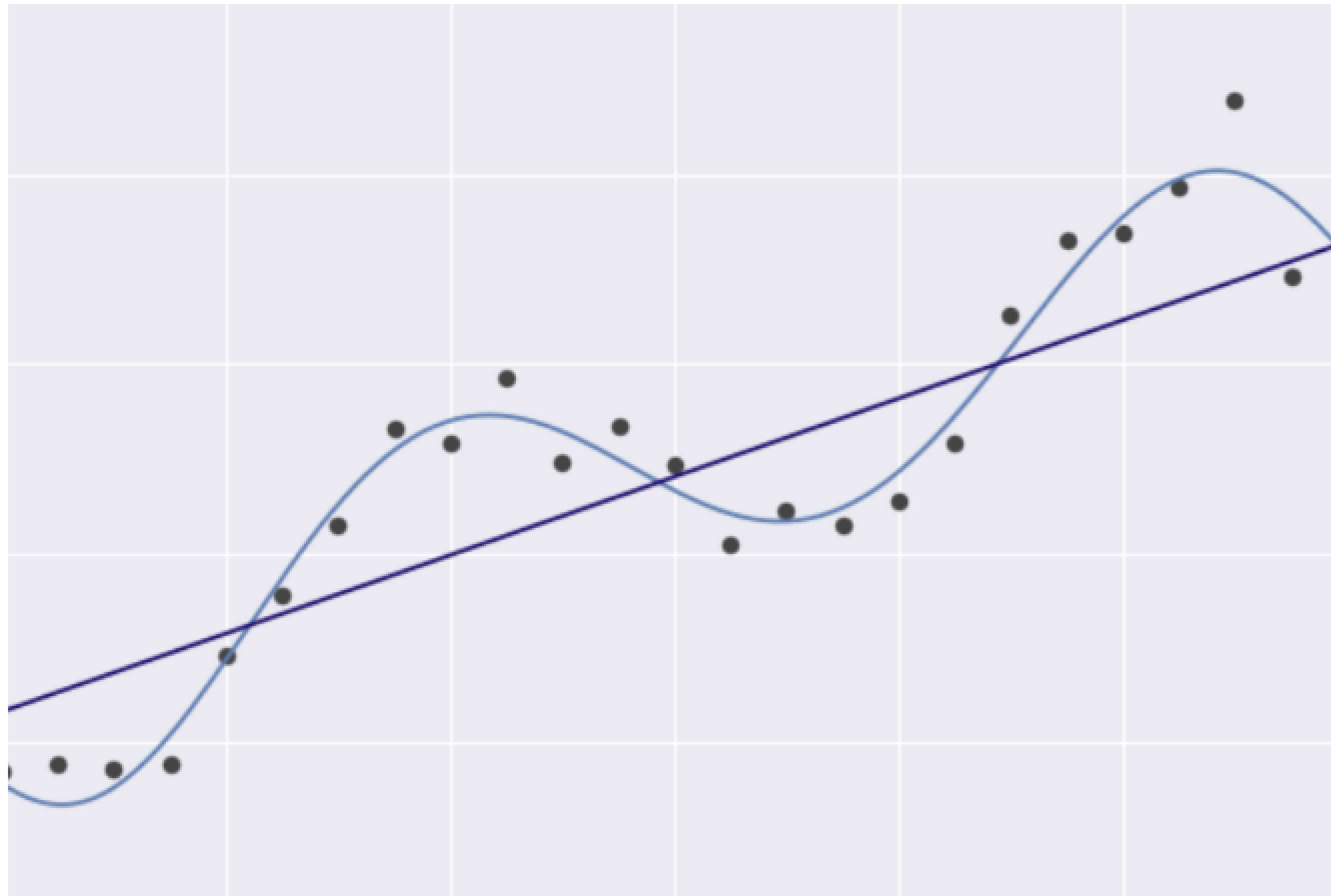
Order columns (One Hot col first)

```
# List cols one hot
one_hot_columns = [col for col in rt_y3.columns if "Industry_" in col]

# Column order: first one-hot columns, then the others
new_column_order = one_hot_columns + [col for col in rt_y3.columns if col not in one_hot_columns]

# Rearrange the DataFrame columns in the new order
rt_y3 = rt_y3[new_column_order]
```

```
# Check df encoded
rt_y3.head()
```

	Industry_Banking/Credit	Industry_Insurance	Industry_Investing/Securities	Country	Year	Liabilities to Assets	Debt to Liabilities	Debt to Equity	InterestExp to EBIT	Return On Equity	Total_Sh
567	0	0	1	14	0	0.04	0.35	0.02	0.00	0.13	
78	1	0	0	16	0	0.93	0.32	4.13	0.59	0.07	
873	0	0	1	17	0	0.04	0.94	0.04	0.00	0.27	
426	1	0	0	14	0	0.85	0.04	0.24	1.24	0.49	
881	0	0	1	17	0	0.47	0.60	0.52	1.15	0.01	

9. ***Imputation Models Testing***



Models will be tested:

MICEFOREST: Multiple Imputation by Chained Equations (MICE) Forest is an advanced imputation method that utilizes random forests. It works iteratively, creating multiple imputations for missing values by modeling each feature with missing data as a function of other features. It's particularly effective because it can handle non-linear relationships and interactions between variables.

BayesianRidge: applies the Bayesian approach to linear regression, meaning that it treats the regression parameters as random variables and estimates a probability distribution for them rather than single point estimates. This model is useful for imputation because it can incorporate the uncertainty in its forecasts, providing a range of possible values rather than a single estimate.

LinearRegression fundamental statistical approach that models the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data. For imputation, it forecasts missing values based on linear relationships identified in the data, assuming that the relationship between the variables is linear.

```
# copy df
rt_y4 = rt_y3.copy()
```

MICEFOREST

```
# Install libraries need
pip install miceforest
pip install miceforest --upgrade
pip install git+https://github.com/AnotherSamWilson/miceforest.git # (if miceforest previous install does not work)
pip install dask --upgrade
pip install lightgbm --upgrade
```

```
time: 0 ns (started: 2023-09-02 23:00:01 +02:00)
```

```
# Miceforest (MICE)
import miceforest as mf
from miceforest import ImputationKernel
```

UDF steps explanation

1. **Imputation Setup:** fraction of the known data is artificially set to NaN to simulate missing values for testing the MICE model's imputation accuracy.
2. **MICE Model Creation:** ImputationKernel from the MICEFOREST library is created, which is then tuned and run to impute missing values in the data.
3. **Parameter Tuning:** function optionally tunes the parameters of the MICE algorithm to optimize performance.
4. **Imputation Execution:** MICE model imputes missing values and the resulting complete dataset is retrieved.
5. **Accuracy Assessment:** function calculates the Root Mean Squared Error (RMSE) between the true values and the imputed ones to measure accuracy.
6. **Error Reporting:** it computes the percentage error relative to the range of the data, providing a normalized indicator of the imputation error.

7. **Result Comparison:** DataFrame is created displaying the original values against the imputed ones for a direct comparison.

```
from sklearn.metrics import mean_squared_error

def mice_imputation_test(df, col_to_test = 'X', cols_to_category=['A', 'B', 'C']):

    '''
    1) Select dataframe with NaN values
    2) All NaN expected predicted will be compared with real
    '''

    global imputed_df, comparison_df, percentage_error_mice, rmse_mice, kds

    # If columns are not category yet, convert them
    df_mf = df.copy()
    df_mf[cols_to_category] = df_mf[cols_to_category].astype('category')

    # Create a copy of the DataFrame and select a fraction of the known data to be imputed
    df_mf = df.dropna(subset=[col_to_test]).copy()
    true_values = df_mf.sample(frac=0.1)[col_to_test]
    df_mf.loc[true_values.index, col_to_test] = np.nan

    # --- MICE model creating
    kds = mf.ImputationKernel(df_mf, random_state=42)

    # --- Tuning parameters
    optimal_parameters, losses = kds.tune_parameters(dataset=0, optimization_steps=20)
    kds.mice(1, variable_parameters=optimal_parameters)
    # print(optimal_parameters)
    # ---

    # --- Manual setting
    # n_estimators, are number of tree. So, define it considering how mauch data your df contains
    # kds.mice(iterations=20, n_estimators=10, device='gpu') # verbose=2

    # --- Complete the imputed data
    imputed_df = kds.complete_data()

    # Recovers the imputed values
    imputed_values = imputed_df.loc[true_values.index, col_to_test]

    # --- Calculate the RMSE
```

```python
    rmse_mice = np.sqrt(mean_squared_error(true_values, imputed_values))

    # Calculate the percentage error
    range_of_data = df[col_to_test].max() - df[col_to_test].min()
    percentage_error_mice = (rmse_mice / range_of_data) * 100

    print(f'RMSE percentage_error (min,max): {percentage_error_mice:.2f}')
    print(f'RMSE: {rmse_mice:.2f}')

    # --- Create a DataFrame with real and imputed values
    comparison_df = pd.DataFrame({
        'True Values': true_values,
        'Imputed Values': ['{:.2f}'.format(value) for value in imputed_values]
    })

    display(comparison_df)

    return
```

```python
mice_imputation_test(rt_y4, col_to_test = 'Return On Equity',
                 cols_to_category=['Industry_Banking/Credit', 'Industry_Insurance',
                                   'Industry_Investing/Securities', 'Country', 'Year'])
```

```python
# Plot compare the distribution by data and data with imputation
# kds.plot_imputed_distributions(wspace=0.3,hspace=0.5)
```

time: 16 ms (started: 2023-09-03 13:07:05 +02:00)

Skelearn models: BayesianRidge and LinearRegression

```python
#- sklearn imputation libraries
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import BayesianRidge
```

UDF steps explanation

1. **Validate Estimator Choice:** function checks whether the chosen estimator is either 'BayesianRidge' or 'LinearRegression'.

2. **Grouping Data (if applicable):** if a grouping_col is specified, the function groups the DataFrame by this column; otherwise, it treats the entire DataFrame as a single group.

3. **Impute Within Groups:** for each group (or the entire DataFrame if no grouping is specified), the function performs the following:

- Temporarily removes a random subset of known values from the column designated for imputation to create a test set within the data.
- Determines the numerical columns that will be included in the imputation process.
- Applies the IterativeImputer with the specified estimator to the numerical columns, fitting the model and transforming the data to impute missing values.
- Re-integrates the imputed values back into the group.
- Reassemble Imputed DataFrame: If the data was grouped, the individual groups with imputed values are recombined into a single DataFrame. If no grouping was specified, the entire DataFrame with imputed values is prepared for output.

1. **Create Comparison Data:** comparison DataFrame is constructed containing the original and imputed values, allowing for a direct comparison to assess the imputation accuracy.

2. **Evaluate Model Performance:** function calculates the RMSE between the known true values and the imputed values, providing a metric of imputation accuracy. It also calculates the percentage error relative to the range of the data for normalization.

3. **Print Summary Statistics and Error Metrics:** outputs the mean of the true values, the mean of the imputed values, the percentage error, and the RMSE to give an overview of the imputation performance.

4. **Display Results:** comparison DataFrame is displayed, showing the true and imputed values side by side for inspection.

```
def iterative_imputation_sklearn(df, estimator_choice='BayesianRidge', col_to_test = 'X', grouping_col = 'A'):

    '''
    Impute missing values using iterative imputer.
    - estimator_choice: Either 'BayesianRidge' or 'LinearRegression'

    Returns:
    imputed_df, comparison_df, rmse
    '''

    global imputed_df, comparison_df, percentage_error_sklearn, rmse_sklearn

    # Check estimator choice
```

```
    if estimator_choice not in ['BayesianRidge', 'LinearRegression']:
        raise ValueError("estimator_choice must be either 'BayesianRidge' or 'LinearRegression'")

    # Select column to check

    true_values_list = []
    imputed_values_list = []
    categorical_values_list = []

    # If grouping_col is specified, group by that column, otherwise use the entire dataframe as a single group.
    if grouping_col:
        groups = df.groupby(grouping_col)
    else:
        groups = [df] # Wrap the dataframe in a list to make it compatible with the rest of the code

    # Model used
    print('BayesianRidge model')

    # Define a function to impute missing values within each group
    def impute_group(group):
        subset_group = group.dropna(subset=[col_to_test]).copy()
        true_values = subset_group.sample(frac=0.1)[col_to_test]
        subset_group.loc[true_values.index, col_to_test] = np.nan
        numerical_columns = subset_group.select_dtypes(include=[np.number]).columns

        # -- model decided to use in udf setting
        if estimator_choice == 'BayesianRidge':
            imp = IterativeImputer(estimator=BayesianRidge(), max_iter=30, random_state=0)
        else:
            imp = IterativeImputer(estimator=LinearRegression(), max_iter=30, random_state=0)

        subset_group[numerical_columns] = imp.fit_transform(subset_group[numerical_columns])

        categorical_columns = subset_group.select_dtypes(exclude=[np.number]).columns
        # --

        for idx in true_values.index:
            true_values_list.append(true_values[idx])
            imputed_values_list.append(subset_group.loc[idx, col_to_test])
            categorical_values_list.append(group.loc[idx, categorical_columns])

        return subset_group

    # Apply the imputation function
```

```
    if grouping_col:
        imputed_df = groups.apply(impute_group).reset_index(drop=True)
    else:
        imputed_df = impute_group(df)

    # -- df comparing values
    comparison_df = pd.DataFrame({
        **pd.DataFrame(categorical_values_list),
        'True Values': true_values_list,
        'Imputed Values': ['{:.2f}'.format(value) for value in imputed_values_list]
    })

    # -- Evaluate model
    rmse_sklearn = np.sqrt(mean_squared_error(true_values_list, imputed_values_list))
    range_of_data = df[col_to_test].max() - df[col_to_test].min()
    percentage_error_sklearn = (rmse_sklearn / range_of_data) * 100

    print(f'MEAN true values: {np.mean(true_values_list):.2f}')
    print(f'MEAN imputed values: {np.mean(imputed_values_list):.2f}')
    print(f'RMSE percentage_error (min,max): {percentage_error_sklearn:.2f}')
    print(f'RMSE: {rmse_sklearn:.2f}')

    display(comparison_df)

    return
```

BayesianRidge

```
# iterative_imputation_sklearn(rt_y4, estimator_choice='BayesianRidge', col_to_test = 'Return On Equity', grouping_col = None)
```

```
# Name variable
percentage_error_sklearn_Bayesian = percentage_error_sklearn
rmse_sklearn_Bayesian = rmse_sklearn
```

LinearRegression *(before running it, run once again the model)*

```
# iterative_imputation_sklearn(rt_y1, estimator_choice='LinearRegression', col_to_test = 'Return On Equity', grouping_col = None)
```

```
# Name variable
percentage_error_sklearn_Linear = percentage_error_sklearn
```

```
rmse_sklearn_Linear = rmse_sklearn
```

Evaluation of the models tested

```
# Print models RMSE percentage_error (min,max) and RMSE

print('All imputation model tested results\n')
print('MICE')
print(f'RMSE percentage_error (min,max): {percentage_error_mice:.2f}')
print(f'RMSE: {rmse_mice:.2f}')
print('='*15)

print('BayesianRidge')
print(f"RMSE percentage_error (min,max): {percentage_error_sklearn_Bayesian:.2f}")
print(f'RMSE: {rmse_sklearn_Bayesian:.2f}')
print('='*15)

print('LinearRegression')
print(f"RMSE percentage_error (min,max): {percentage_error_sklearn_Linear:.2f}")
print(f'RMSE: {rmse_sklearn_Linear:.2f}')

print('\nNOTE: random seed is not implemented. So, outputs can change running from scratch the model. Make decision considering a
```

```
All imputation model tested results

MICE
RMSE percentage_error (min,max): 8.81
RMSE: 0.18
===============
BayesianRidge
RMSE percentage_error (min,max): 6.38
RMSE: 0.13
===============
LinearRegression
RMSE percentage_error (min,max): 1.26
RMSE: 1.09

NOTE: random seed is not implemented. So, outputs can change running from scratch the model. Make decision considering also some
random tests.
```

10. ***Imputation Application:*** select and apply the chosen model to impute missing values in the dataset

```
# copy previous df
rt_y4_imputed = rt_y4.copy()
```

a. NaN total values before imputation

```
rt_y4_imputed.isna().sum().sum()
```

941

b. Imputation using model selected: BayesianRidge

```
# Model definition to imputing NaN values
def iterative_imputation_sklearn_all_df(df, estimator_choice='BayesianRidge', col_to_test = 'X', grouping_col = 'Y', highlight_im

    global imputed_df, comparison_df, percentage_error_sklearn, rmse_sklearn, imputed_coordinates, coordinates_df, imputed_values

    if estimator_choice not in ['BayesianRidge', 'LinearRegression']:
        raise ValueError("estimator_choice must be either 'BayesianRidge' or 'LinearRegression'")

    true_values_list = []
    imputed_values_list = []
    categorical_values_list = []

    if grouping_col:
        groups = df.groupby(grouping_col)
    else:
        groups = [df] # Wrap the dataframe in a list to make it compatible with the rest of the code

    print(f'Model used: {estimator_choice}\n')

    # Initialize the list and collect NaN coordinates
    imputed_coordinates = []
    for col in df.columns:
        if df[col].isna().any():
            for index in df[df[col].isna()].index:
                imputed_coordinates.append((col, index))
```

```python
    # Create an empty dictionary to store coordinates
    coordinates_dict = {}

    for col, index in imputed_coordinates:
        if col not in coordinates_dict:
            coordinates_dict[col] = []
        coordinates_dict[col].append(index)

    # Convert dictionary to a DataFrame
    coordinates_df = pd.DataFrame.from_dict(coordinates_dict, orient='index').transpose()

    # Fill in the Missing Values with 9.99 caused by mismatched length values imputed on columns
    coordinates_df.fillna(9.99, inplace=True)

    def impute_group(group):
        subset_group = group.dropna(subset=[col_to_test]).copy()

        if subset_group.empty:
            return group

        true_values = subset_group.sample(frac=0.1)[col_to_test]
        true_values_list.extend(true_values)

        numerical_columns = group.select_dtypes(include=[np.number]).columns

        if estimator_choice == 'BayesianRidge':
            imp = IterativeImputer(estimator=BayesianRidge(), max_iter=30, random_state=0)
        else:
            imp = IterativeImputer(estimator=LinearRegression(), max_iter=30, random_state=0)

        group[numerical_columns] = imp.fit_transform(group[numerical_columns])

        imputed_values_for_true = group.loc[true_values.index, col_to_test]
        imputed_values_list.extend(imputed_values_for_true)

        return group

    # Apply the imputation function
    if grouping_col:
        imputed_df = groups.apply(impute_group).reset_index(drop=True)
    else:
        imputed_df = impute_group(df)

    # -- df comparing values
```

```
    comparison_df = pd.DataFrame({
        **pd.DataFrame(categorical_values_list),
#         'True Values': true_values_list,
        'Imputed Values': ['{:.2f}'.format(value) for value in imputed_values_list]
    })

    # -- Evaluate model
    if not true_values_list or not imputed_values_list:
        print("One or both lists are empty!")
        return

    rmse_sklearn = np.sqrt(mean_squared_error(true_values_list, imputed_values_list))
    range_of_data = df[col_to_test].max() - df[col_to_test].min()
    percentage_error_sklearn = (rmse_sklearn / range_of_data) * 100

    print('To print dataframe with index imputed variable is: "coordinates_df" (values 9.99 are just to make df equal rows)')

    display(comparison_df)

    if highlight_imputed:
        def highlight_cells(data):
            # Create a default 'no highlight' style for all cells
            styles = pd.DataFrame('', index=data.index, columns=data.columns)

            # For each coordinate in the imputed_coordinates, set the background color
            for col, row in imputed_coordinates:
                styles.at[row, col] = 'background-color: yellow'
            return styles

        display(imputed_df.style.apply(highlight_cells, axis=None))

    return
```

```
# BayesianRidge
iterative_imputation_sklearn_all_df(rt_y4_imputed, estimator_choice='BayesianRidge', col_to_test = 'Return On Equity', grouping_c
```

```
# Dataframe explained for each column index where NaN has been replaces
# coordinates_df.sort_values(by=['Liabilities to Assets'], ascending=True)
```

```
time: 0 ns (started: 2023-09-08 20:13:31 +02:00)
```

Sum tot NaN after imputation

```
rt_y4_imputed.isna().sum().sum()
```

```
0
```

11. ***Post-Imputation Processing (Decoding):*** decode the data to revert any encoding done before the imputation, ensuring data is in its original format for interpretation and analysis

```
# Copy imputed df
rt_y1_pred = rt_y4_imputed.copy()
```

1. Decoding of 'Industry' column

```
def get_industry_name(row):
    for col in one_hot_columns:
        if row[col] == 1:
            return col.replace('Industry_', '')
    return None

# Apply the function to DataFrame to create the new 'industry_name' column
rt_y1_pred['Industry'] = rt_y1_pred.apply(get_industry_name, axis=1)

# Now remove the one-hot columns
rt_y1_pred.drop(one_hot_columns, axis=1, inplace=True)
```

```
# Set industry col in first position
cols = [col for col in rt_y1_pred.columns if col != 'Industry']

# Place 'industry_name' as first column
cols = ['Industry'] + cols

# Reordina il DataFrame
rt_y1_pred = rt_y1_pred[cols]
```

2. Decoding of 'Year' column

```
rt_y1_pred['Year'] = oe.inverse_transform(rt_y1_pred[['Year']])[:, 0]
```

```
# Sort values
rt_y1_pred = rt_y1_pred.sort_values(by='Year', ascending=True)
```

)

```
# Trasform to datatime adding month and year (are just rapresentative)
rt_y1_pred['Year'] = pd.to_datetime(rt_y1_pred['Year'].astype(str) + '-12-31')
```

3. Decoding of 'Country' column

```
rt_y1_pred['Country'] = rt_y3.loc[rt_y1_pred.index, 'Country']
```

Dataframe imputed (a few lines of demonstration)

```
rt_y1_pred.head(20)
```

	Industry	Country	Year	Liabilities to Assets	Debt to Liabilities	Debt to Equity	InterestExp to EBIT	Return On Equity	Total_Shareholders_Equity	Operating Margin	Capex to EBIT	D&A aprox to EBITDA
567	Investing/Securities	14	2019-12-31	0.04	0.35	0.02	0.00	0.13	1.00	0.49	0.30	0.09
623	Investing/Securities	14	2019-12-31	0.35	0.16	0.09	0.00	0.19	1.00	0.25	0.04	0.46
386	Banking/Credit	29	2019-12-31	0.88	0.17	1.27	0.59	0.11	1.00	0.30	0.21	0.28
1001	Investing/Securities	14	2019-12-31	0.52	0.39	0.45	0.43	-0.00	1.00	-0.06	0.21	-0.58
374	Banking/Credit	14	2019-12-31	0.83	0.42	2.00	0.51	0.13	1.00	0.58	0.01	-0.05
1005	Investing/Securities	6	2019-12-31	0.36	0.51	0.29	0.13	0.10	1.00	0.30	0.13	-0.57
38	Banking/Credit	29	2019-12-31	0.86	0.21	1.30	2.82	0.10	1.00	0.17	0.18	0.58
1015	Investing/Securities	19	2019-12-31	0.67	0.16	0.32	0.02	0.15	1.00	0.26	0.15	0.37
538	Insurance	16	2019-12-31	0.91	0.06	0.58	0.80	0.02	1.00	0.14	0.51	0.62
454	Insurance	26	2019-12-31	0.88	0.06	0.45	0.12	0.09	1.00	0.07	0.23	0.13
502	Insurance	29	2019-12-31	0.90	0.23	2.01	0.09	0.22	1.00	0.20	0.06	0.17
42	Banking/Credit	6	2019-12-31	0.95	0.35	7.20	0.27	0.10	1.00	0.20	0.20	0.45
434	Insurance	16	2019-12-31	0.95	0.04	0.70	0.38	0.06	1.00	0.05	0.11	0.31
601	Investing/Securities	15	2019-12-31	0.27	0.90	1.48	0.09	-0.77	-1.00	-0.35	0.45	0.09

	Industry	Country	Year	Liabilities to Assets	Debt to Liabilities	Debt to Equity	InterestExp to EBIT	Return On Equity	Total_Shareholders_Equity	Operating Margin	Capex to EBIT	D&A aprox to EBITDA
549	Investing/Securities	15	2019-12-31	0.18	0.78	0.18	0.37	-0.04	1.00	-2.73	0.31	-0.38
346	Banking/Credit	19	2019-12-31	0.95	0.46	2.98	1.16	0.05	1.00	0.22	0.22	0.64
642	Investing/Securities	16	2019-12-31	0.11	0.02	0.00	0.00	0.20	1.00	1.51	0.12	-0.38
358	Banking/Credit	19	2019-12-31	0.89	0.05	0.40	0.07	0.09	1.00	0.24	0.87	0.33
668	Investing/Securities	16	2019-12-31	0.37	0.23	0.13	0.02	0.25	1.00	0.31	0.29	0.30
1066	Investing/Securities	17	2019-12-31	0.97	0.00	0.07	0.10	0.11	1.00	0.21	1.48	0.26

Thank you, Filippo Maria Mariani