**Read Me**

This code is designed to analyze and visualize data related to errors, failures, and machines in a Predictive Maintenance (PdM) project. It fetches data from CSV files using Google Colab, a popular Python development environment, which can be used on Windows, Mac, and Google Colab.

**Libraries Used**

* pandas: for data manipulation and analysis.
* numpy: for numerical computing.
* seaborn: for statistical data visualization.
* matplotlib: for creating plots and charts.
* sklearn: for machine learning tasks such as data preprocessing.
* warnings: for suppressing warnings.

**Data Files**

The code reads data from the following CSV files:

1. PdM\_errors.csv: Contains information about errors occurred in machines, including error IDs, dates, and times.
2. PdM\_failures.csv: Contains information about component failures, including failure types, dates, and times.
3. PdM\_machines.csv: Contains information about machines, including machine IDs and model types.
4. PdM\_maint.csv: Contains information about maintenance activities performed on machines, including dates and times.
5. PdM\_telemetry.csv: Contains telemetry data from machines, including voltage, rotation, and pressure readings.

**Data Analysis and Visualization**

The code performs various data analysis and visualization tasks on the fetched data, including:

**Error Data Analysis**

* Checking for missing values in the error data.
* Extracting date and time components from the datetime column.
* Visualizing the frequency of different error IDs.
* Visualizing the distribution of errors by year, month, and day.

**Failure Data Analysis**

* Formatting the date and time columns in the failure data.
* Visualizing the frequency of different component failures.
* Visualizing the distribution of failures by year, month, and day.

**Machines Data Analysis**

* Checking the content of the machines data.

**Usage**

1. Make sure you have the necessary libraries installed. If not, you can install them using pip or conda.
2. Load the code in a Python environment such as Google Colab, Jupyter Notebook, or any other Python IDE.
3. Upload the required CSV files (PdM\_errors.csv, PdM\_failures.csv, PdM\_machines.csv, PdM\_maint.csv, PdM\_telemetry.csv) to your environment or update the file paths in the code accordingly.
4. Run the code to perform data analysis and visualization tasks.
5. Explore the generated plots and charts to gain insights from the data.Top of Form

# Dealing with Null Values

After merging all the dataframes into one master dataframe named **base\_data**, we checked for null values in the data using **base\_data.isnull().sum()**. We found that **error\_id**, **comp**, and **failure** columns have a significant number of missing values.

We also noticed that the distribution of data based on machine ID is not equal after merging, which is contrary to what we observed in individual dataframes before merging. This could be due to the application of the Central Limit Theorem, which states that the distribution of sample means approximates a normal distribution as the sample size gets larger, regardless of the population's distribution. However, since we have missing values in the merged data, it could have affected the distribution.

To address the issue of missing values, we have several options:

1. Dropping Rows: We can choose to drop rows with missing values using the **dropna()** method. However, this may result in a significant loss of data and may not be the best approach, especially if the missing values are not randomly distributed.
2. Filling with Default/Placeholder Values: We can fill the missing values with default or placeholder values using the **fillna()** method. This approach should be carefully chosen, as it may introduce bias or misinformation in the data.
3. Filling with Statistical Measures: We can fill the missing values with statistical measures such as mean, median, or mode of the respective columns using the **fillna()** method. This can help to retain the integrity of the data to some extent.
4. Using Machine Learning Techniques: We can use machine learning techniques, such as regression or imputation models, to predict and fill the missing values based on other available data. This approach may require more computational resources and expertise but can result in a more accurate imputation of missing values.

The choice of approach for dealing with missing values depends on the nature of the data and the specific requirements of the analysis or model development. It is important to carefully consider the potential impact of handling missing values on the integrity and validity of the results. In our case, we need to carefully evaluate the options and choose the most appropriate approach to handle the missing values in the **error\_id**, **comp**, and **failure** columns of the **base\_data** master dataframe.

# Data Analysis and Visualization

Once the missing values are handled appropriately, we can proceed with data analysis and visualization to gain insights from the data. Some potential areas for analysis and visualization include:

1. Exploratory Data Analysis (EDA): We can perform EDA on the cleaned data to understand the distribution, trends, and patterns in the data. This can involve generating summary statistics, creating visualizations such as histograms, box plots, and scatter plots, and exploring relationships between variables.
2. Time Series Analysis: Since the data is time-stamped, we can perform time series analysis to understand the temporal patterns and trends in the data. This can involve analyzing trends, seasonality, and autocorrelation in the data, and building forecasting models.
3. Machine Failure Analysis: We can analyze the data related to machine failures, such as the frequency, duration, and causes of failures, to identify patterns and potential factors contributing to machine failures. This can help in developing strategies for preventive maintenance and reducing downtime.
4. Maintenance Analysis: We can analyze the data related to machine maintenance, such as maintenance events, schedules, and costs, to understand the effectiveness of maintenance strategies and optimize maintenance processes.

# Preprocessing the Data

The code provided performs several preprocessing steps on the dataset before using it for machine learning model training. These steps include:

1. Copying Categorical Columns: The code creates a copy of all the columns in the dataset that have object (categorical) data type and stores it in a variable called **catdf**.
2. Dropping Categorical Columns: The code drops the categorical columns from the original dataset **base\_data** using the **drop** function with **axis=1** to specify columns and **inplace=True** to apply the changes in place.
3. Creating Dummy Variables for Categorical Columns: The code uses the **pd.get\_dummies** function to create dummy variables for the categorical columns in **catdf** and adds them to the original dataset **base\_data**. This step converts the categorical variables into numerical representations that can be used as input features in a machine learning model.
4. Data Visualization: The code uses the **sns** (Seaborn) library to visualize the data. It creates box plots and bar charts to analyze the relationship between failure, date, and other variables in the dataset.

# Balancing the Data

To handle imbalanced data, the code uses the Synthetic Minority Over-sampling Technique (SMOTE) to oversample the minority class (label '1' for failure) and create a balanced dataset. The **SMOTE** function from the **imblearn** library is used with a sampling strategy of 0.04, which means that the minority class will be oversampled to have approximately 4% of the total samples.

# Scaling and Splitting the Data

The code further preprocesses the data by scaling the input features using the Min-Max Scaling technique. It splits the dataset into train and test sets using the **train\_test\_split** function from the **sklearn.model\_selection** module. The train set contains 80% of the data, and the test set contains 20% of the data. The **MinMaxScaler** is used to scale the features in both the original dataset and the oversampled dataset.

The scaled train and test sets are stored in variables **X\_train**, **X\_test**, **Y\_train**, and **Y\_test** for the original dataset, and **X\_train\_sam**, **X\_test\_sam**, **Y\_train\_sam**, and **Y\_test\_sam** for the oversampled dataset.

# Saving the Min-Max Scaler

The code saves the trained **MinMaxScaler** object into a pickle file named **MinMaxScaler.pkl** using the **dump** function from the **pickle** module. This scaler can be later used to scale new data during the prediction phase of the machine learning model.

**Random Forest Classifier with Min-Max Scaling**

This code snippet showcases the implementation of a Random Forest Classifier for a dataset after preprocessing and oversampling using the SMOTE technique.

**Preprocessing the Data**

The code starts with preprocessing the data, which includes the following steps:

1. Creating a copy of all object data type columns using **select\_dtypes([object]).copy()**.
2. Dropping the categorical values from the master data using **drop()** function.
3. Creating dummy variables for categorical columns using **pd.get\_dummies()** function.
4. Shuffling the data using **shuffle()** function.
5. Splitting the data into input features (X) and output labels (y).

**Balancing the Data**

The imbalanced nature of the dataset is addressed using the SMOTE (Synthetic Minority Over-sampling Technique) method. The code uses the **SMOTE** class from **imblearn** library to oversample the minority class (**failure**) with a specified sampling strategy of 0.04, resulting in a balanced dataset.

**Scaling and Splitting the Data**

The data is then scaled using Min-Max scaling technique using **MinMaxScaler()** from **sklearn.preprocessing** library. The scaled data is split into training and testing sets using **train\_test\_split()** function from **sklearn.model\_selection** with a 80-20 split ratio.

**Random Forest Classifier**

The Random Forest Classifier is implemented using **RandomForestClassifier()** from **sklearn.ensemble** library. The classifier is trained on the oversampled training data and predictions are made on the testing data. Accuracy and classification report (including precision, recall, and F1-score) are computed using **accuracy\_score()** and **classification\_report()** functions from **sklearn.metrics**.

**Model Evaluation**

Confusion matrix is plotted using **ConfusionMatrixDisplay** from **sklearn.metrics.plot\_confusion\_matrix** to visually evaluate the model's performance. The trained Random Forest Classifier model is saved using **joblib** library for future use.

Overall, Random Forest Classifier with Min-Max scaling and SMOTE oversampling technique is implemented to address the class imbalance issue and achieve better accuracy, precision, and recall on the given dataset.

**Contributing**

If you wish to contribute to this project, you can fork the repository, make your changes, and submit a pull request. Your contributions are welcome and will be reviewed by the project maintainers.

**License**

This project is released under the [MIT License](https://chat.openai.com/chat/LICENSE), which allows you to use, modify, and distribute the code for both commercial and non-commercial purposes. However, you are required to include the original license and copyright information in your derivative works.