Schneider Electric Machine Learning Project

PRASAD JADHAV

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')

df = pd.read_csv('schneider_electric_dataset.csv')
pd.set_option('display.max_columns',20)
print(df.shape)
```

```
(100000, 20)
```

```
df.head()
```

```
   Product_ID     Category             Technical_Specifications  Sales
Region  \
0          1  Power Meter  {"voltage": "220V", "power": "50W"}   4851
North
1          2       Switch  {"voltage": "220V", "power": "50W"}    556
West
2          3       Switch  {"voltage": "110V", "power": "30W"}   4713
East
3          4        Relay  {"voltage": "220V", "power": "50W"}   4499
East
4          5       Switch  {"voltage": "110V", "power": "30W"}   3704
West

   Customer_Rating Fault_Flag  Maintenance_Records    Price  \
```

```
0           3.62        No                     5  2009.28
1           1.70        No                     2  1899.78
2           1.80       Yes                     0  2271.00
3           3.46        No                     2  1174.32
4           4.59        No                     7  3081.87

                  Date  Energy_Consumption  \
0  2020-01-01 00:00:00              468.20
1  2020-01-01 01:00:00              133.46
2  2020-01-01 02:00:00              414.33
3  2020-01-01 03:00:00              170.97
4  2020-01-01 04:00:00              115.83

                        Sensor_Readings  Warranty_Expiry  Lead_Time  \
0  {"temperature": 30, "pressure": 1.2}               24         11
1  {"temperature": 40, "pressure": 1.5}                9         15
2  {"temperature": 40, "pressure": 1.5}                7         10
3  {"temperature": 40, "pressure": 1.5}               26         18
4  {"temperature": 40, "pressure": 1.5}               30         19

   Supply_Chain_Risk  Demand_Variation Promotions_Applied
Competitor_Price  \
0               0.13             45.39                Yes
2990.56
1               0.40             47.75                Yes
1358.28
2               0.94             46.11                 No
2160.00
3               0.08             11.98                 No
2948.44
4               0.06             35.71                 No
4197.57

  Return_Flag User_Demographics
0         Yes        Enterprise
1         Yes        Individual
2          No    Small Business
3         Yes        Government
4         Yes        Individual
```

```
df.tail()
```

```
       Product_ID Category             Technical_Specifications  Sales
Region  \
99995       99996   Switch  {"voltage": "220V", "power": "50W"}   3311
West
99996       99997   Switch  {"voltage": "220V", "power": "50W"}   4178
East
99997       99998   Switch  {"voltage": "220V", "power": "50W"}   4403
West
```

```
99998       99999   Switch  {"voltage": "110V", "power": "30W"}   2154
West
99999      100000   Switch  {"voltage": "220V", "power": "50W"}   4157
North

       Customer_Rating Fault_Flag  Maintenance_Records    Price  \
99995             2.41        Yes                    6  4651.82
99996             4.82         No                    2  3206.15
99997             1.25        Yes                    9   280.02
99998             4.65         No                    7  4570.44
99999             1.58        Yes                    8  2879.91

                      Date  Energy_Consumption  \
99995  2031-05-29 11:00:00              445.16
99996  2031-05-29 12:00:00              292.51
99997  2031-05-29 13:00:00              300.55
99998  2031-05-29 14:00:00              279.96
99999  2031-05-29 15:00:00              382.38

                                Sensor_Readings  Warranty_Expiry
Lead_Time  \
99995  {"temperature": 30, "pressure": 1.2}                6
19
99996  {"temperature": 30, "pressure": 1.2}                3
11
99997  {"temperature": 30, "pressure": 1.2}                9
5
99998  {"temperature": 30, "pressure": 1.2}               16
2
99999  {"temperature": 40, "pressure": 1.5}                3
11

       Supply_Chain_Risk  Demand_Variation Promotions_Applied  \
99995               0.16             47.12                Yes
99996               0.70             32.99                Yes
99997               0.79              4.98                Yes
99998               0.23             23.90                Yes
99999               0.21             18.29                 No

       Competitor_Price Return_Flag User_Demographics
99995            543.99         Yes    Small Business
99996           4132.05         Yes        Enterprise
99997           2179.27          No        Government
99998           2823.15          No        Government
99999           1819.78         Yes        Enterprise
```

Dataset Columns Overview

- Product_ID: Unique identifier for each product.
- Category: Product category (e.g., circuit breakers, switches).

- Technical_Specifications: JSON field containing specs (voltage, power rating, etc.).
- Sales: Historical sales data (number of units sold).
- Region: Sales region.
- Customer_Rating: Average customer ratings.
- Fault_Flag: Indicates if a product had a fault (Yes/No).
- Maintenance_Records: Number of maintenance events.
- Price: Product price.
- Date: Timestamp of sales/maintenance data.
- Energy_Consumption: Energy consumed by the product (kWh).
- Sensor_Readings: Data from product sensors (temperature, pressure, etc.).
- Warranty_Expiry: Time remaining on the product warranty (in months).
- Lead_Time: Delivery lead time (in days).
- Supply_Chain_Risk: Risk factor based on supply chain conditions.
- Demand_Variation: Month-to-month variation in demand.
- Promotions_Applied: Whether a promotion was applied (Yes/No).
- Competitor_Price: Price of a similar product by competitors.
- Return_Flag: Indicates if the product was returned (Yes/No).
- User_Demographics: Information about the purchasing company or customer (size, industry).

Data Exploration

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 20 columns):
 #   Column                    Non-Null Count   Dtype
---  ------                    --------------   -----
 0   Product_ID                100000 non-null  int64
 1   Category                  100000 non-null  object
 2   Technical_Specifications  100000 non-null  object
 3   Sales                     100000 non-null  int64
 4   Region                    100000 non-null  object
 5   Customer_Rating           100000 non-null  float64
 6   Fault_Flag                100000 non-null  object
 7   Maintenance_Records       100000 non-null  int64
 8   Price                     100000 non-null  float64
 9   Date                      100000 non-null  object
 10  Energy_Consumption        100000 non-null  float64
 11  Sensor_Readings           100000 non-null  object
 12  Warranty_Expiry           100000 non-null  int64
 13  Lead_Time                 100000 non-null  int64
 14  Supply_Chain_Risk         100000 non-null  float64
 15  Demand_Variation          100000 non-null  float64
 16  Promotions_Applied        100000 non-null  object
 17  Competitor_Price          100000 non-null  float64
```

```
 18  Return_Flag                100000 non-null  object
 19  User_Demographics          100000 non-null  object
dtypes: float64(6), int64(5), object(9)
memory usage: 15.3+ MB
```

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

```
Product_ID                    0
Category                      0
Technical_Specifications      0
Sales                         0
Region                        0
Customer_Rating               0
Fault_Flag                    0
Maintenance_Records           0
Price                         0
Date                          0
Energy_Consumption            0
Sensor_Readings               0
Warranty_Expiry               0
Lead_Time                     0
Supply_Chain_Risk             0
Demand_Variation              0
Promotions_Applied            0
Competitor_Price              0
Return_Flag                   0
User_Demographics             0
dtype: int64
```

```
df.duplicated().sum()
```

```
0
```

```
df.describe()
```

```
          Product_ID          Sales  Customer_Rating
Maintenance_Records  \
count  100000.000000  100000.000000    100000.000000
100000.000000
mean    50000.500000    2519.835300         3.008512
4.500480
std     28867.657797    1430.183332         1.153912
2.880612
min         1.000000      50.000000         1.000000
0.000000
25%     25000.750000    1280.000000         2.010000
2.000000
50%     50000.500000    2514.000000         3.010000
5.000000
75%     75000.250000    3766.000000         4.010000
7.000000
```

```
max    100000.000000    4999.000000         5.000000
9.000000

                  Price  Energy_Consumption  Warranty_Expiry
Lead_Time  \
count  100000.000000       100000.000000    100000.000000
100000.000000
mean     2525.309220          254.743850        17.969350
9.991390
std      1427.492425          141.423348        10.084338
5.478388
min        50.010000           10.000000         1.000000
1.000000
25%      1289.540000          132.150000         9.000000
5.000000
50%      2527.080000          255.325000        18.000000
10.000000
75%      3760.700000          376.580000        27.000000
15.000000
max      4999.870000          500.000000        35.000000
19.000000

       Supply_Chain_Risk  Demand_Variation  Competitor_Price
count      100000.000000     100000.000000     100000.000000
mean            0.499296         25.093246       2558.876172
std             0.288830         14.402074       1416.896620
min             0.000000          0.000000        100.000000
25%             0.250000         12.640000       1332.490000
50%             0.500000         25.140000       2560.285000
75%             0.750000         37.592500       3789.760000
max             1.000000         50.000000       4999.900000
```

```python
num_cols = [x for x in df.columns if df[x].dtypes != 'float64']

for col in num_cols:
    print(f"Value counts for column '{col}':")
    print(df[col].value_counts())
    print("\n" + "_"*40 + "\n")
```

```
Value counts for column 'Product_ID':
Product_ID
1         1
66651     1
66673     1
66672     1
66671     1
         ..
33332     1
33331     1
33330     1
```

```
33329     1
100000    1
Name: count, Length: 100000, dtype: int64

________________________________________

Value counts for column 'Category':
Category
Circuit Breaker    20166
Switch             20024
Busway             20006
Power Meter        19945
Relay              19859
Name: count, dtype: int64

________________________________________

Value counts for column 'Technical_Specifications':
Technical_Specifications
{"voltage": "220V", "power": "50W"}    50074
{"voltage": "110V", "power": "30W"}    49926
Name: count, dtype: int64

________________________________________

Value counts for column 'Sales':
Sales
3333    39
1087    37
1437    37
4053    36
3482    35
        ..
3437     8
3549     8
3773     7
1925     7
1113     6
Name: count, Length: 4950, dtype: int64

________________________________________

Value counts for column 'Region':
Region
East     25177
South    25049
North    24887
West     24887
Name: count, dtype: int64
```

```
________________________________________

Value counts for column 'Fault_Flag':
Fault_Flag
Yes    50217
No     49783
Name: count, dtype: int64

________________________________________

Value counts for column 'Maintenance_Records':
Maintenance_Records
0    10146
8    10103
2    10071
9    10071
4     9988
1     9972
5     9968
6     9960
7     9939
3     9782
Name: count, dtype: int64

________________________________________

Value counts for column 'Date':
Date
2020-01-01 00:00:00    1
2027-08-09 02:00:00    1
2027-08-10 00:00:00    1
2027-08-09 23:00:00    1
2027-08-09 22:00:00    1
                      ..
2023-10-20 19:00:00    1
2023-10-20 18:00:00    1
2023-10-20 17:00:00    1
2023-10-20 16:00:00    1
2031-05-29 15:00:00    1
Name: count, Length: 100000, dtype: int64

________________________________________

Value counts for column 'Sensor_Readings':
Sensor_Readings
{"temperature": 40, "pressure": 1.5}    50084
{"temperature": 30, "pressure": 1.2}    49916
Name: count, dtype: int64

________________________________________
```

```
Value counts for column 'Warranty_Expiry':
Warranty_Expiry
13    3026
6     2971
9     2956
26    2953
24    2940
28    2911
18    2904
33    2899
19    2893
21    2882
10    2870
1     2862
4     2861
30    2858
35    2855
32    2853
25    2853
14    2849
5     2847
16    2846
3     2846
7     2841
12    2832
23    2830
22    2826
8     2825
15    2823
17    2818
27    2815
2     2813
29    2794
20    2787
11    2767
34    2758
31    2736
Name: count, dtype: int64

________________________________________

Value counts for column 'Lead_Time':
Lead_Time
8     5428
1     5386
12    5368
5     5337
2     5318
17    5316
```

```
9     5263
14    5260
15    5258
6     5256
11    5255
19    5250
16    5242
13    5235
4     5217
10    5208
18    5184
3     5117
7     5102
Name: count, dtype: int64

________________________________________

Value counts for column 'Promotions_Applied':
Promotions_Applied
Yes    50261
No     49739
Name: count, dtype: int64

________________________________________

Value counts for column 'Return_Flag':
Return_Flag
Yes    50103
No     49897
Name: count, dtype: int64

________________________________________

Value counts for column 'User_Demographics':
User_Demographics
Individual        25117
Government        25067
Enterprise        24994
Small Business    24822
Name: count, dtype: int64

________________________________________
```

```
cat_cols = [col for col in df.columns if df[col].dtype == 'object' or
df[col].dtype.name == 'category']

for col in cat_cols:
    print(f"Value counts for column '{col}':")
```

```
    print(df[col].value_counts())
    print("\n" + "_"*40 + "\n")
```

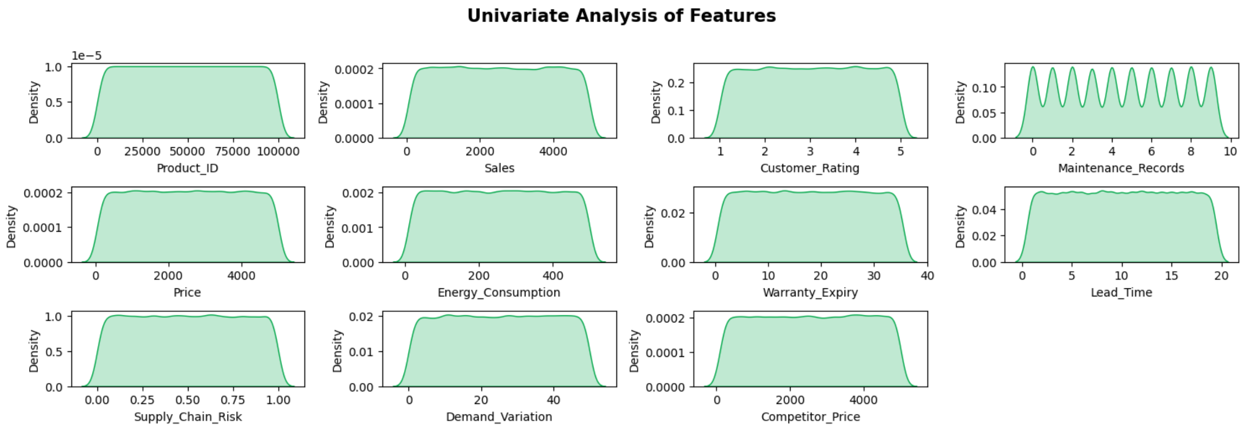
```
Value counts for column 'Category':
Category
Circuit Breaker    20166
Switch             20024
Busway             20006
Power Meter        19945
Relay              19859
Name: count, dtype: int64

________________________________________

Value counts for column 'Technical_Specifications':
Technical_Specifications
{"voltage": "220V", "power": "50W"}    50074
{"voltage": "110V", "power": "30W"}    49926
Name: count, dtype: int64

________________________________________

Value counts for column 'Region':
Region
East     25177
South    25049
North    24887
West     24887
Name: count, dtype: int64

________________________________________

Value counts for column 'Fault_Flag':
Fault_Flag
Yes    50217
No     49783
Name: count, dtype: int64

________________________________________

Value counts for column 'Date':
Date
2020-01-01 00:00:00    1
2027-08-09 02:00:00    1
2027-08-10 00:00:00    1
2027-08-09 23:00:00    1
2027-08-09 22:00:00    1
                      ..
2023-10-20 19:00:00    1
2023-10-20 18:00:00    1
```

```
2023-10-20 17:00:00    1
2023-10-20 16:00:00    1
2031-05-29 15:00:00    1
Name: count, Length: 100000, dtype: int64

________________________________________

Value counts for column 'Sensor_Readings':
Sensor_Readings
{"temperature": 40, "pressure": 1.5}    50084
{"temperature": 30, "pressure": 1.2}    49916
Name: count, dtype: int64

________________________________________

Value counts for column 'Promotions_Applied':
Promotions_Applied
Yes    50261
No     49739
Name: count, dtype: int64

________________________________________

Value counts for column 'Return_Flag':
Return_Flag
Yes    50103
No     49897
Name: count, dtype: int64

________________________________________

Value counts for column 'User_Demographics':
User_Demographics
Individual        25117
Government        25067
Enterprise        24994
Small Business    24822
Name: count, dtype: int64

________________________________________
```

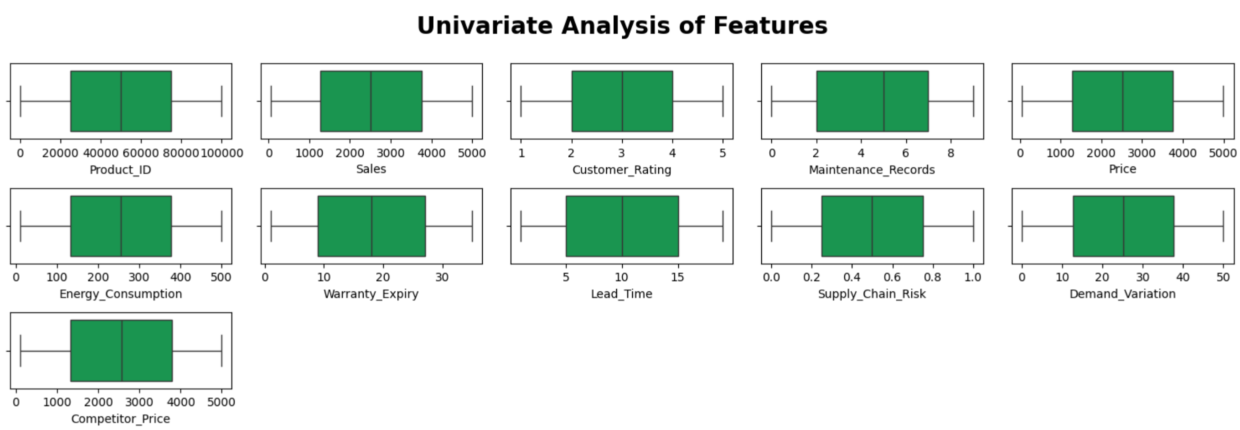
EDA

```
num_features = df.select_dtypes(include = ['int64',
'float64']).dtypes.index

plt.figure(figsize=(15,15))
plt.suptitle('Univariate Analysis of
Features',fontweight='bold',fontsize=15,y=1)
```

```python
for i in range(0,len(num_features)):
    plt.subplot(10,4,i+1)
    sns.kdeplot(x=df[num_features[i]],shade=True,color='#06a94d')
    plt.tight_layout()
```




```python
plt.figure(figsize = (15,15))
plt.suptitle('Univariate Analysis of 
Features',fontweight='bold',fontsize=20,y=1)

for i in range(0,len(num_features)):
    plt.subplot(10,5,i+1)
    sns.boxplot(data=df,x=num_features[i],color='#06a94d')
    plt.xlabel(num_features[i])
    plt.tight_layout()
```




```python
cat_features = df.select_dtypes(include='object').dtypes.index

plt.figure(figsize=(15,15))
plt.suptitle('Univariate Analysis of 
Features',fontweight='bold',fontsize=15,y=1)

for i in range(0,len(cat_features)):
```

```
    plt.subplot(10,4,i+1)
    sns.countplot(x=df[cat_features[i]],color='#06a94d')
    plt.tight_layout()
```

Feature Preprocessing

```
# Extract the values for Voltage and Power
df['Voltage'] = df['Technical_Specifications'].apply(lambda x: eval(x)
['voltage'].replace('V', ''))
df['Power'] = df['Technical_Specifications'].apply(lambda x: eval(x)
['power'].replace('W', ''))

# Extract temperature and pressure from the Sensor_Readings column
df['Temperature'] = df['Sensor_Readings'].apply(lambda x: eval(x)
['temperature'])
df['Pressure'] = df['Sensor_Readings'].apply(lambda x: eval(x)
['pressure'])

# Define the mapping function
def map_category(value):
    mapping = {
        "Circuit Breaker": "4",
        "Switch": "3",
        "Busway": "2",
        "Power Meter": "1",
        "Relay": "0"
    }
    return mapping.get(value, "Unknown")  # Default to "Unknown" if
value not in mapping


# Apply the mapping function to the Category column
df['Category'] = df['Category'].apply(map_category)

# Define the mapping function
def map_fault_flag(value):
    mapping = {
        "Yes": 1,
        "No": 0
    }
    return mapping.get(value, -1)  # Default to -1 if value not in
mapping

# Apply the mapping function to the Fault_Flag column
df['Fault_Flag'] = df['Fault_Flag'].apply(map_fault_flag)

# Define the mapping function
def map_return_flag(value):
    mapping = {
        "Yes": 1,
```

```
        "No": 0
    }
    return mapping.get(value, -1)  # Default to -1 if value not in
mapping

# Apply the mapping function to the Return_Flag column
df['Return_Flag'] = df['Return_Flag'].apply(map_return_flag)

# Define the mapping function
def map_user_demographics(value):
    mapping = {
        "Individual": 0,
        "Government": 1,
        "Enterprise": 2,
        "Small Business":3
    }
    return mapping.get(value, -1)

df['User_Demographics'] =
df['User_Demographics'].apply(map_user_demographics)

# Example Machine Learning Use Cases:
```

Product Fault Detection (Classification Problem)

- Objective: Predict whether a product will have a fault or not (Fault_Flag column).

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

Fault_Flag
1    50217
0    49783
Name: count, dtype: int64

# Preprocessing
X = df[['Category', 'Maintenance_Records', 'Energy_Consumption',
'Warranty_Expiry', 'Voltage', 'Power']]
y = df['Fault_Flag'] # .apply(lambda x: 1 if x == 'Yes' else 0)  #
Binary classification

# Imbalanced Data Handling
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_res,y_res = smote.fit_resample(X,y)

X_res,y_res = SMOTE().fit_resample(X,y)

from sklearn.model_selection import train_test_split

# Split into train and test sets
```

```python
X_train, X_test, y_train, y_test = train_test_split(X_res, y_res,
test_size=0.20, random_state=42)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

from sklearn.metrics import accuracy_score, classification_report

def get_best_model_report(models, X_train, y_train, X_test, y_test):
    """
    Train models, evaluate their accuracy, and display the
classification report for the best model.

    Args:
    models (dict): A dictionary with model names as keys and model
objects as values.
    X_train, y_train: Training features and labels.
    X_test, y_test: Testing features and labels.
    """
    best_model_name = None
    best_model = None
    best_accuracy = 0
    y_best_pred = None

    # Train, predict, and evaluate each model
    for model_name, model in models.items():
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)
        print(f'Accuracy {model_name}: {accuracy:.4f}')

        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_model_name = model_name
            best_model = model
            y_best_pred = y_pred

    print(f"\nBest Model: {best_model_name} with Accuracy:
{best_accuracy:.4f}\n")
    print("Classification Report for Best Model:\n")
    print(classification_report(y_test, y_best_pred))

# Example usage
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
```

```python
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier

# Define models
models = {
    "LogisticRegression": LogisticRegression(n),
    "KNeighborsClassifier": KNeighborsClassifier(n),
    "SVC": SVC(),
    "DecisionTreeClassifier": DecisionTreeClassifier(),
    "RandomForestClassifier": RandomForestClassifier(),
    "GradientBoostingClassifier": GradientBoostingClassifier()
}

# Call the function
get_best_model_report(models, X_train, y_train, X_test, y_test)
```

```
Accuracy LogisticRegression: 0.4949
Accuracy KNeighborsClassifier: 0.5012
Accuracy SVC: 0.4963
Accuracy DecisionTreeClassifier: 0.4965
Accuracy RandomForestClassifier: 0.4954
Accuracy GradientBoostingClassifier: 0.5003

Best Model: KNeighborsClassifier with Accuracy: 0.5012

Classification Report for Best Model:

              precision    recall  f1-score   support

           0       0.50      0.50      0.50     10079
           1       0.50      0.50      0.50     10008

    accuracy                           0.50     20087
   macro avg       0.50      0.50      0.50     20087
weighted avg       0.50      0.50      0.50     20087
```

Sales Demand Forecasting (Regression Problem)

- Objective: Predict the sales of products based on historical data.

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Define the mapping function
def map_region(value):
    mapping = {
        "East": 3,
        "South": 2,
        "North": 1,
        "West":0
```

```python
    }
    return mapping.get(value, -1)

df['Region'] = df['Region'].apply(map_region)

# Preprocessing
X = df[['Customer_Rating', 'Region', 'Energy_Consumption',
'Demand_Variation', 'Supply_Chain_Risk', 'Lead_Time', 'Price']]
y = df['Sales']

# Split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Model training
regressor = RandomForestRegressor()
regressor.fit(X_train, y_train)
```

```
RandomForestRegressor()
```

```python
# Predictions
y_pred = regressor.predict(X_test)

# Evaluation
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
```

```
Mean Squared Error: 2108473.0428792466
```

```python
import xgboost as xgb
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, r2_score

# XGBoost model instance
xgb_model = xgb.XGBRegressor(objective='reg:squarederror',
random_state=42)

# Hyperparameter tuning with GridSearchCV
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.05, 0.1],
    'subsample': [0.8, 0.9, 1.0],
    'colsample_bytree': [0.8, 0.9, 1.0]
}

grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid,
scoring='r2', cv=3, verbose=1, n_jobs=-1)
grid_search.fit(X_train, y_train)

# Best parameters
best_params = grid_search.best_params_
```

```
print("Best parameters:", best_params)

# Train the model with the best parameters
best_xgb_model = xgb.XGBRegressor(**best_params,
objective='reg:squarederror', random_state=42)
best_xgb_model.fit(X_train, y_train)

# Evaluate on training and testing sets
y_train_pred = best_xgb_model.predict(X_train)
y_test_pred = best_xgb_model.predict(X_test)

# Calculate performance metrics
train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))

train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

print(f"Training RMSE: {train_rmse}, R2: {train_r2}")
print(f"Testing RMSE: {test_rmse}, R2: {test_r2}")

# Check for underfitting and overfitting
if train_r2 > test_r2 + 0.1:
    print("The model might be overfitting.")
elif test_r2 > train_r2 + 0.1:
    print("The model might be underfitting.")
else:
    print("The model is performing well.")

Fitting 3 folds for each of 243 candidates, totalling 729 fits
Best parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.01,
'max_depth': 3, 'n_estimators': 100, 'subsample': 1.0}
Training RMSE: 1428.672709539614, R2: 0.0010957717895507812
Testing RMSE: 1431.829846463929, R2: 3.427267074584961e-05
The model is performing well.
```

Predict Maintenance Needs (Classification Problem)

- Objective: Predict when maintenance will be required based on usage and sensor data.

```
from sklearn.ensemble import GradientBoostingClassifier
from xgboost import XGBClassifier

# Preprocessing
X = df[['Energy_Consumption', 'Supply_Chain_Risk', 'Warranty_Expiry',
'Temperature', 'Pressure']]
y = df['Maintenance_Records'].apply(lambda x: 1 if x > 0 else 0)  #
Binary classification (maintenance or no maintenance)
```

```
# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Model training
gb_model = GradientBoostingClassifier()
gb_model.fit(X_train, y_train)
```

```
GradientBoostingClassifier()
```

```
# Predictions
y_pred = gb_model.predict(X_test)

# Evaluation
print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.8993666666666666
              precision    recall  f1-score   support

           0       0.00      0.00      0.00      3018
           1       0.90      1.00      0.95     26982

    accuracy                           0.90     30000
   macro avg       0.45      0.50      0.47     30000
weighted avg       0.81      0.90      0.85     30000
```

```
# Predict Maintenance Needs (Classification or Regression Problem)
# Objective: Predict when maintenance will be required based on usage
and sensor data.
# Model: Classification models for predicting maintenance or
regression models for predicting the time to next maintenance.

# Classification Problem
# The goal is to predict whether a product will require maintenance
(Maintenance_Need_Flag = 1) or not (Maintenance_Need_Flag = 0).

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report

# Feature selection and preprocessing
X = df[['Energy_Consumption', 'Supply_Chain_Risk', 'Warranty_Expiry',
'Temperature', 'Pressure']]
y = df['Maintenance_Records'].apply(lambda x: 1 if x > 0 else 0)  #
Binary classification (maintenance or no maintenance)

# Split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

```python
# Model training
clf_model = RandomForestClassifier(random_state=42)
clf_model.fit(X_train, y_train)
```

```
RandomForestClassifier(random_state=42)
```

```python
# Predictions
y_pred = clf_model.predict(X_test)

# Evaluation
print(f"Classification Accuracy: {accuracy_score(y_test, y_pred)}")
print(classification_report(y_test, y_pred))
```

```
Classification Accuracy: 0.8873
              precision    recall  f1-score   support

           0       0.10      0.01      0.03      3018
           1       0.90      0.98      0.94     26982

    accuracy                           0.89     30000
   macro avg       0.50      0.50      0.48     30000
weighted avg       0.82      0.89      0.85     30000
```

```python
# Regression Problem
# The goal is to predict the time remaining before maintenance is
required (in days or hours).

from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error

# Normalize columns for calculation
df['Energy_Consumption_Norm'] = df['Energy_Consumption'] /
df['Energy_Consumption'].max()
df['Temperature_Norm'] = df['Temperature'] / df['Temperature'].max()
df['Pressure_Norm'] = df['Pressure'] / df['Pressure'].max()
# Define maintenance lead time (in days)
# The lower the combined normalized score, the shorter the maintenance
lead time.
df['Maintenance_Lead_Time'] = 100 - (
    40 * df['Energy_Consumption_Norm'] +
    30 * df['Temperature_Norm'] +
    20 * df['Pressure_Norm']
)

# Add random noise to simulate variability
df['Maintenance_Lead_Time'] = df['Maintenance_Lead_Time'] +
np.random.normal(0, 5, len(df))

# Clip values to ensure no negative lead times
df['Maintenance_Lead_Time'] =
```

```python
df['Maintenance_Lead_Time'].clip(lower=1)

# Drop temporary normalized columns
df.drop(columns=['Energy_Consumption_Norm', 'Temperature_Norm',
                 'Pressure_Norm'], inplace=True)

# Feature selection and preprocessing
X = df[['Maintenance_Records', 'Energy_Consumption',
'Supply_Chain_Risk', 'Warranty_Expiry', 'Temperature', 'Pressure']]
y = df['Maintenance_Lead_Time']  # Numeric column indicating time to
maintenance

# Split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Model training
reg_model = RandomForestRegressor(random_state=42)
reg_model.fit(X_train, y_train)
```

```
RandomForestRegressor(random_state=42)
```

```python
# Predictions
y_pred = reg_model.predict(X_test)

# Evaluation
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"Mean Absolute Error: {mae}")
```

```
Mean Squared Error: 27.291767909916082
Mean Absolute Error: 4.170469035139971
```

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, r2_score

# Random Forest model instance
rf_model = RandomForestRegressor(random_state=42)

# Hyperparameter tuning with GridSearchCV
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2']
}

grid_search = GridSearchCV(estimator=rf_model, param_grid=param_grid,
```

```
scoring='r2', cv=3, verbose=1, n_jobs=-1)
grid_search.fit(X_train, y_train)

# Best parameters
best_params = grid_search.best_params_
print("Best parameters:", best_params)

# Train the model with the best parameters
best_rf_model = RandomForestRegressor(**best_params, random_state=42)
best_rf_model.fit(X_train, y_train)

# Evaluate on training and testing sets
y_train_pred = best_rf_model.predict(X_train)
y_test_pred = best_rf_model.predict(X_test)

# Calculate performance metrics
train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))

train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

print(f"Training RMSE: {train_rmse:.4f}, R2: {train_r2:.4f}")
print(f"Testing RMSE: {test_rmse:.4f}, R2: {test_r2:.4f}")

# Check for underfitting and overfitting
if train_r2 > test_r2 + 0.1:
    print("The model might be overfitting.")
elif test_r2 > train_r2 + 0.1:
    print("The model might be underfitting.")
else:
    print("The model is performing well.")
```

```
Fitting 3 folds for each of 324 candidates, totalling 972 fits
Best parameters: {'max_depth': 20, 'max_features': 'sqrt',
'min_samples_leaf': 4, 'min_samples_split': 10, 'n_estimators': 150}
Training RMSE: 4.0193, R2: 0.9135
Testing RMSE: 5.0704, R2: 0.8613
The model is performing well.
```

Product Return Prediction (Classification Problem)

- Objective: Predict whether a product will be returned (`Return_Flag`).

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

```
Return_Flag
1    50103
0    49897
Name: count, dtype: int64
```

```python
# Preprocessing
X = df[['Customer_Rating', 'Price', 'Sales', 'Warranty_Expiry',
'Category', 'User_Demographics']]
y = df['Return_Flag'] # .apply(lambda x: 1 if x == 'Yes' else 0)

# Imbalanced Data Handling
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split

smote = SMOTE(random_state=42)
X_res,y_res = smote.fit_resample(X,y)

X_res,y_res = SMOTE().fit_resample(X,y)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_res, y_res,
test_size=0.20, random_state=42)

# Model training
model = RandomForestClassifier()
model.fit(X_train, y_train)
```

```
RandomForestClassifier()
```

```python
# Predictions
y_pred = model.predict(X_test)

# Evaluation
print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.5045903602434887
              precision    recall  f1-score   support

           0       0.50      0.53      0.52      9966
           1       0.51      0.48      0.49     10076

    accuracy                           0.50     20042
   macro avg       0.50      0.50      0.50     20042
weighted avg       0.50      0.50      0.50     20042
```

```python
import pickle

pickle.dump(model,open('product_return_predictor.pkl','wb'))

# model = pickle.load(open('product_return_predictor.pkl','rb'))

# Assuming `model` is already trained and loaded

print("Product Return Prediction")
```

```python
# Input data
customer_rating = float(input("Enter the Customer Rating: "))
price = float(input("Enter the Price: "))
sales = int(input("Enter the Sales: "))
warranty_expiry = int(input("Enter the Warranty Expiry: "))
category = int(input("Enter the Category: "))
user_demographics = int(input("Enter the User Demographics: "))

# Create a feature array
input_point = np.array([[customer_rating, price, sales,
warranty_expiry, category, user_demographics]])

# Make a prediction
prediction = model.predict(input_point)

# Check the prediction and print the result
if prediction[0] == 1:
    print("Product Return")
else:  # No need to check for `prediction[0] == 0` since it's implied
    print("Product Not Returned")
```

```
Product Return Prediction
Enter the Customer Rating: 1.80
Enter the Price: 2271.00
Enter the Sales: 4713
Enter the Warranty Expiry: 7
Enter the Category: 3
Enter the User Demographics: 3
Product Not Returned
```

```python
# X['Category'] = X['Category'].astype('int')

# Model training
x_model = XGBClassifier()
x_model.fit(X_train, y_train)

# Predictions
x_pred = model.predict(X_test)

# Evaluation
print(f"Accuracy: {accuracy_score(y_test, x_pred)}")
print(classification_report(y_test, x_pred))
```

```
Accuracy: 0.5040914080431095
              precision    recall  f1-score   support

           0       0.50      0.53      0.52      9966
           1       0.51      0.48      0.49     10076

    accuracy                           0.50     20042
   macro avg       0.50      0.50      0.50     20042
```

```
weighted avg       0.50      0.50      0.50     20042


# Code for XGBoost Classification with Hyperparameter Tuning

import xgboost as xgb
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# XGBoost classifier instance
xgb_clf = xgb.XGBClassifier(objective='binary:logistic',
use_label_encoder=False, eval_metric='logloss', random_state=42)

# Hyperparameter tuning with GridSearchCV
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.05, 0.1],
    'subsample': [0.8, 0.9, 1.0],
    'colsample_bytree': [0.8, 0.9, 1.0],
    'gamma': [0, 0.1, 0.2],
    'reg_alpha': [0, 0.01, 0.1],
    'reg_lambda': [1, 1.5, 2]
}

grid_search = GridSearchCV(estimator=xgb_clf, param_grid=param_grid,
scoring='accuracy', error_score='raise', cv=3, verbose=1, n_jobs=-1)
grid_search.fit(X_train, y_train)

# Best parameters
best_params = grid_search.best_params_
print("Best parameters:", best_params)

# Train the model with the best parameters
best_xgb_model = xgb.XGBClassifier(**best_params,
objective='binary:logistic', use_label_encoder=False,
eval_metric='logloss', random_state=42, n_jobs=1)
best_xgb_model.fit(X_train, y_train)

# Evaluate on training and testing sets
y_train_pred = best_xgb_model.predict(X_train)
y_test_pred = best_xgb_model.predict(X_test)

# Calculate performance metrics
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

print(f"Training Accuracy: {train_accuracy:.4f}")
print(f"Testing Accuracy: {test_accuracy:.4f}")
```

```
# Display classification report and confusion matrix
print("\nClassification Report:\n", classification_report(y_test,
y_test_pred))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_test_pred))

# Check for overfitting or underfitting
if train_accuracy > test_accuracy + 0.1:
    print("The model might be overfitting.")
elif test_accuracy > train_accuracy + 0.1:
    print("The model might be underfitting.")
else:
    print("The model is performing well.")
```

Predicting Warranty Expiry Risk (Regression)

- Predict the time remaining on the warranty `(Warranty_Expiry)`.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score

# Features and target
X = df[['Maintenance_Records', 'Voltage', 'Power', 'Temperature',
'Pressure']] # Maintenance_Lead_Time
y = df['Warranty_Expiry'] # .apply(lambda x: 1 if x == 'Yes' else 0)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Model training
regressor = RandomForestRegressor(random_state=42)
regressor.fit(X_train, y_train)

# Predictions and evaluation
y_pred = regressor.predict(X_test)
print("Mean Absolute Error:", mean_absolute_error(y_test, y_pred))
print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
print("R2 Score:", r2_score(y_test, y_pred))
```

```
Mean Absolute Error: 8.731453395728098
Mean Squared Error: 101.68509626570433
R2 Score: -0.0004927865631072503
```

Collaborative Filtering (User-Product Recommendation)

- Recommend products to users based on historical sales and customer ratings.

```
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.preprocessing import MinMaxScaler
```

```
# Load dataset
df_2 = pd.read_csv("schneider_electric_dataset.csv")

# Create a user-product matrix
user_product_matrix = df_2.pivot_table(
    index='Region', columns='Product_ID', values='Customer_Rating',
aggfunc='mean'
).fillna(0)

# Compute similarity
user_similarity = cosine_similarity(user_product_matrix)
user_similarity_df = pd.DataFrame(user_similarity,
index=user_product_matrix.index, columns=user_product_matrix.index)

# Recommend products for a specific user
def recommend_products(user, top_n=5):
    similar_users =
user_similarity_df[user].sort_values(ascending=False)[1:top_n + 1]
    recommended_products =
df_2[df_2['Region'].isin(similar_users.index)]['Product_ID'].unique()
    return recommended_products

# Example recommendation for a user
print(recommend_products('North', top_n=5))

[    2     3     4 ... 99997 99998 99999]
```

Content-Based Filtering (Product Similarity Recommendation)

- Recommend similar products based on Technical_Specifications, Category, and Sensor_Readings.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neighbors import NearestNeighbors

# Combine features for product similarity
df_2['Combined_Features'] = df_2['Category'] + " " +
df_2['Technical_Specifications'] + " " + df_2['Sensor_Readings']

# TF-IDF Vectorizer
tfidf = TfidfVectorizer(stop_words='english')
tfidf_matrix = tfidf.fit_transform(df_2['Combined_Features'])

# Nearest Neighbors model
nn = NearestNeighbors(metric='cosine', algorithm='brute',
n_neighbors=6)
nn.fit(tfidf_matrix)

NearestNeighbors(algorithm='brute', metric='cosine', n_neighbors=6)
```

```python
# Product recommendation function
def get_recommendations(product_id, top_n=5):
    product_idx = df_2[df_2['Product_ID'] == product_id].index[0]
    distances, indices = nn.kneighbors(tfidf_matrix[product_idx],
n_neighbors=top_n + 1)
    product_indices = indices.flatten()[1:]  # Exclude the first one
(itself)
    return df_2['Product_ID'].iloc[product_indices].tolist()

# Example recommendation
print(get_recommendations(227, top_n=5))
```

```
[79689, 79719, 46712, 46716, 79738]
```

Region-Based Recommendation

- Recommend top-selling products in a specific Region.

```python
# Get top products by region
def recommend_top_products(region, top_n=5):
    regional_sales = df_2[df_2['Region'] ==
region].groupby('Product_ID')['Sales'].sum()
    top_products =
regional_sales.sort_values(ascending=False).head(top_n).index.tolist()
    return top_products

# Example for North Region
print(recommend_top_products('East', top_n=5))
```

```
[31433, 97295, 95349, 88942, 12037]
```

Price-Sensitive Recommendation

- Recommend products within a similar price range.

```python
# Recommend products in a price range
def recommend_by_price(product_id, price_range=100):
    product_price = df_2[df_2['Product_ID'] == product_id]
['Price'].values[0]
    recommended_products = df[
        (df_2['Price'] >= product_price - price_range) &
(df_2['Price'] <= product_price + price_range)
    ]['Product_ID'].tolist()
    return recommended_products

# Example recommendation
print(recommend_by_price(727, price_range=5))
```

```
[62, 607, 648, 727, 2212, 3080, 4357, 4518, 5243, 5589, 5690, 5863,
6288, 6447, 6736, 6868, 7404, 7594, 8127, 8584, 9223, 9312, 10096,
10606, 10976, 11092, 11279, 11387, 11588, 12540, 13201, 13674, 13787,
```

```
14022, 14077, 14516, 14573, 15363, 15562, 15580, 16061, 16132, 17202,
17219, 17875, 18260, 18717, 18904, 19535, 19739, 19771, 19867, 19983,
20464, 20572, 22235, 22244, 22309, 23242, 23746, 24056, 26336, 26499,
26777, 28004, 28602, 28806, 29270, 29349, 29904, 30033, 30922, 31044,
32630, 33722, 33983, 34236, 34432, 35398, 35623, 36697, 36993, 37289,
38195, 38360, 38417, 38544, 38554, 38628, 38949, 39222, 39288, 39383,
39549, 40104, 40224, 40722, 41476, 41892, 43090, 43333, 44375, 44812,
45587, 48165, 51023, 51205, 51290, 51423, 51658, 51868, 52199, 54147,
54870, 55439, 55675, 55751, 55977, 56134, 56154, 56779, 56986, 57636,
58520, 58746, 58748, 58971, 59439, 59771, 60495, 60751, 60819, 61597,
62397, 62426, 63429, 63795, 63937, 64227, 65080, 65621, 66605, 67187,
68621, 69167, 69593, 69673, 69895, 70788, 70863, 72609, 73307, 74435,
74663, 75782, 76808, 78202, 79026, 79914, 80025, 80193, 80620, 80662,
81069, 81430, 81914, 82166, 82324, 83826, 83871, 84049, 84483, 84491,
84849, 84966, 85158, 85212, 85681, 87251, 87272, 87677, 89491, 90478,
90678, 91693, 92265, 92415, 93204, 93823, 94669, 94997, 97111, 97536,
97772, 98454, 98559, 98663, 98730, 99788]
```

Customer Behavior Recommendation

- Recommend products based on customer demographics (User_Demographics).

```
# Recommend products for a specific user demographic
def recommend_by_demographics(demographic, top_n=5):
    demo_products = df_2[df_2['User_Demographics'] ==
demographic].groupby('Product_ID')['Sales'].sum()
    top_products =
demo_products.sort_values(ascending=False).head(top_n).index.tolist()
    return top_products

# Example for Small Business demographic
print(recommend_by_demographics('Small Business', top_n=5))

[95349, 94738, 37180, 77266, 97295]
```

Competitor Price-Based Recommendation

- Recommend products that are competitively priced.

```
# Recommend products with competitive pricing
def recommend_competitive_products(product_id, price_diff=50):
    product_price = df_2[df_2['Product_ID'] == product_id]
['Price'].values[0]
    recommended_products = df[
        abs(df_2['Price'] - df_2['Competitor_Price']) <= price_diff
    ]['Product_ID'].tolist()
    return recommended_products

# Example recommendation
print(recommend_competitive_products(227, price_diff=5))
```

```
[234, 932, 1207, 1350, 2966, 3302, 3750, 4259, 4804, 5237, 5329, 5784,
5858, 6473, 6716, 6939, 7124, 8708, 9528, 9793, 10058, 10606, 10898,
12142, 12298, 12643, 14023, 14530, 14713, 15945, 16253, 16360, 16491,
16972, 17402, 18726, 19179, 20267, 20353, 21165, 21334, 22279, 23514,
23541, 23841, 24511, 24645, 24920, 25159, 26317, 26323, 26688, 27304,
28634, 29036, 29521, 30407, 32719, 33141, 33556, 33925, 34587, 35366,
35389, 35441, 36426, 37039, 37366, 37448, 37723, 38161, 38261, 38325,
38513, 39562, 39880, 40327, 40601, 40650, 40943, 42055, 42458, 42615,
43271, 43332, 43586, 43662, 44913, 45106, 46848, 47350, 47513, 48043,
48327, 48374, 48687, 49203, 49284, 49519, 50000, 50082, 50105, 50175,
50235, 51021, 51091, 51627, 51953, 52107, 52755, 52838, 53037, 53115,
53269, 53544, 53948, 54134, 54539, 54571, 54913, 55484, 56014, 58071,
58108, 58153, 60111, 60244, 60394, 60875, 60919, 61214, 62059, 62282,
63112, 63795, 64259, 65501, 67218, 67281, 68200, 69109, 69563, 70219,
71636, 71680, 71842, 73305, 73328, 73332, 74249, 75527, 75825, 76198,
76866, 77416, 79182, 79775, 79890, 79921, 80509, 81360, 81589, 82168,
83166, 83258, 83710, 83768, 84844, 84907, 85053, 85418, 85615, 85768,
86047, 86430, 86703, 86982, 87142, 87320, 87439, 87506, 88364, 88852,
88864, 89000, 89323, 89791, 90340, 90962, 91514, 91576, 92093, 92313,
93217, 93781, 94039, 94668, 95304, 96653, 96690, 97545, 97731, 98142,
99121, 99229, 99458]
```

Energy Efficiency Recommendation

- Recommend products with lower energy consumption.

```python
# Recommend energy-efficient products
def recommend_energy_efficient(top_n=5):
    efficient_products = 
df_2.sort_values(by='Energy_Consumption').head(top_n)
['Product_ID'].tolist()
    return efficient_products

# Example recommendation
print(recommend_energy_efficient(top_n=5))
```

```
[7607, 68749, 24660, 61109, 82260]
```

```python
# More Advance Working Sonn..!
# Notebook Project By : PRASAD JADHAV (ML-ENG)
# LinkedIn: linkedin.com/in/prasadmjadhav2 | Github: 
github.com/prasadmjadhav2 | Mail: prasadmjadhav6161@gmail.com
```

Thank You!