We used fewc python libraries that help with maff, organizing data, and making charts. We read data from a file named 'sensor.csv', which keeps track of readings from various sensors. We looked at the first few rows of the data and some summary statistics to understand what's in it. We got rid of some columns we didn't need, like timestamps and an unnamed column. We changed the names of some machine statuses to simplify the categories. We filled in missing data points in our sensors' readings with typical (median) values so that there are no gaps. Finally, we checked to make sure there were no more missing values. Why we did it:

To prepare the data for analysis or machine learning by making it cleaner and easier to work with. Removing unnecessary columns helps to focus on the important data. Merging categories and filling missing values makes the data more consistent and accurate for any analysis or predictive modeling we might want to do later. How we went about it:

By using Python libraries like Pandas for data handling, NumPy for numerical operations, and Matplotlib for potential visualization (though no plots were made here). We suppressed warnings that could be annoying or not very helpful to keep the output clean. The process involved several data manipulation steps: dropping columns, replacing values, and filling in missing data, all done using commands provided by the Pandas library. The results:

After cleaning, the data has fewer columns and no missing values, which simplifies further analysis. The machine statuses are grouped into fewer categories, making it easier to analyze trends or problems. By filling in missing values, we ensure that any calculations or models built on this data won't be skewed or error out due to incomplete data.

In [1]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
from collections import Counter
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.model_selection import train_test_split, StratifiedShuffleSplit
from sklearn.metrics import classification_report
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import (
    BaggingClassifier,
    RandomForestClassifier,
    GradientBoostingClassifier,
    AdaBoostClassifier,
    VotingClassifier
)
from xgboost import XGBClassifier
from imblearn.ensemble import BalancedRandomForestClassifier
from sklearn.compose import ColumnTransformer
from sklearn.exceptions import ConvergenceWarning
warnings.filterwarnings("ignore", category=ConvergenceWarning)
```

```
# Load the dataset
file_path = 'sensor.csv'
data = pd.read_csv(file_path)

# Display the first few rows
head = data.head()

# Statistical summary
description = data.describe(include='all')

# Data types
data_types = data.dtypes

# Missing values
missing_values = data.isnull().sum()

head, description, data_types, missing_values
```

Out[1]:

```
(   Unnamed: 0            timestamp  sensor_00  sensor_01  sensor_02  \
 0           0  2018-04-01 00:00:00   2.465394   47.09201    53.2118
 1           1  2018-04-01 00:01:00   2.465394   47.09201    53.2118
 2           2  2018-04-01 00:02:00   2.444734   47.35243    53.2118
 3           3  2018-04-01 00:03:00   2.460474   47.09201    53.1684
 4           4  2018-04-01 00:04:00   2.445718   47.13541    53.2118

    sensor_03  sensor_04  sensor_05  sensor_06  sensor_07  ...  sensor_43  \
 0  46.310760   634.3750   76.45975   13.41146   16.13136  ...   41.92708
 1  46.310760   634.3750   76.45975   13.41146   16.13136  ...   41.92708
 2  46.397570   638.8889   73.54598   13.32465   16.03733  ...   41.66666
 3  46.397568   628.1250   76.98898   13.31742   16.24711  ...   40.88541
 4  46.397568   636.4583   76.58897   13.35359   16.21094  ...   41.40625

    sensor_44  sensor_45  sensor_46  sensor_47  sensor_48  sensor_49  \
 0  39.641200   65.68287   50.92593  38.194440   157.9861   67.70834
 1  39.641200   65.68287   50.92593  38.194440   157.9861   67.70834
 2  39.351852   65.39352   51.21528  38.194443   155.9606   67.12963
 3  39.062500   64.81481   51.21528  38.194440   155.9606   66.84028
 4  38.773150   65.10416   51.79398  38.773150   158.2755   66.55093

    sensor_50  sensor_51  machine_status
 0   243.0556   201.3889          NORMAL
 1   243.0556   201.3889          NORMAL
 2   241.3194   203.7037          NORMAL
 3   240.4514   203.1250          NORMAL
 4   242.1875   201.3889          NORMAL

 [5 rows x 55 columns],
            Unnamed: 0            timestamp      sensor_00      sensor_01  \
 count   220320.000000               220320  210112.000000  219951.000000
 unique            NaN               220320            NaN            NaN
 top               NaN  2018-04-01 00:00:00            NaN            NaN
 freq              NaN                    1            NaN            NaN
 mean    110159.500000                  NaN       2.372221      47.591611
 std      63601.049991                  NaN       0.412227       3.296666
 min          0.000000                  NaN       0.000000       0.000000
 25%      55079.750000                  NaN       2.438831      46.310760
 50%     110159.500000                  NaN       2.456539      48.133678
 75%     165239.250000                  NaN       2.499826      49.479160
 max     220319.000000                  NaN       2.549016      56.727430

             sensor_02      sensor_03      sensor_04      sensor_05  \
 count   220301.000000  220301.000000  220301.000000  220301.000000
 unique            NaN            NaN            NaN            NaN
 top               NaN            NaN            NaN            NaN
 freq              NaN            NaN            NaN            NaN
 mean        50.867392      43.752481     590.673936      73.396414
```

```
std          3.666820      2.418887     144.023912      17.298247
min         33.159720     31.640620       2.798032       0.000000
25%         50.390620     42.838539     626.620400      69.976260
50%         51.649300     44.227428     632.638916      75.576790
75%         52.777770     45.312500     637.615723      80.912150
max         56.032990     48.220490     800.000000      99.999880

            sensor_06      sensor_07  ...      sensor_43      sensor_44  \
count   215522.000000  214869.000000  ...  220293.000000  220293.000000
unique            NaN            NaN  ...            NaN            NaN
top               NaN            NaN  ...            NaN            NaN
freq              NaN            NaN  ...            NaN            NaN
mean        13.501537      15.843152  ...      43.879591      42.656877
std          2.163736       2.201155  ...      11.044404      11.576355
min          0.014468       0.000000  ...      24.479166      25.752316
25%         13.346350      15.907120  ...      39.583330      36.747684
50%         13.642940      16.167530  ...      42.968750      40.509260
75%         14.539930      16.427950  ...      46.614580      45.138890
max         22.251160      23.596640  ...     408.593700    1000.000000

            sensor_45      sensor_46      sensor_47      sensor_48  \
count   220293.000000  220293.000000  220293.000000  220293.000000
unique            NaN            NaN            NaN            NaN
top               NaN            NaN            NaN            NaN
freq              NaN            NaN            NaN            NaN
mean        43.094984      48.018585      44.340903     150.889044
std         12.837520      15.641284      10.442437      82.244957
min         26.331018      26.331018      27.199070      26.331018
25%         36.747684      40.509258      39.062500      83.912030
50%         40.219910      44.849540      42.534720     138.020800
75%         44.849540      51.215280      46.585650     208.333300
max        320.312500     370.370400     303.530100     561.632000

            sensor_49      sensor_50      sensor_51  machine_status
count   220293.000000  143303.000000  204937.000000          220320
unique            NaN            NaN            NaN               3
top               NaN            NaN            NaN          NORMAL
freq              NaN            NaN            NaN          205836
mean        57.119968     183.049260     202.699667             NaN
std         19.143598      65.258650     109.588607             NaN
min         26.620370      27.488426      27.777779             NaN
25%         47.743060     167.534700     179.108800             NaN
50%         52.662040     193.865700     197.338000             NaN
75%         60.763890     219.907400     216.724500             NaN
max        464.409700    1000.000000    1000.000000             NaN

[11 rows x 55 columns],
Unnamed: 0          int64
timestamp          object
sensor_00         float64
sensor_01         float64
sensor_02         float64
sensor_03         float64
sensor_04         float64
sensor_05         float64
sensor_06         float64
sensor_07         float64
sensor_08         float64
sensor_09         float64
sensor_10         float64
sensor_11         float64
sensor_12         float64
sensor_13         float64
sensor_14         float64
sensor_15         float64
sensor_16         float64
sensor_17         float64
```

```
sensor_18         float64
sensor_19         float64
sensor_20         float64
sensor_21         float64
sensor_22         float64
sensor_23         float64
sensor_24         float64
sensor_25         float64
sensor_26         float64
sensor_27         float64
sensor_28         float64
sensor_29         float64
sensor_30         float64
sensor_31         float64
sensor_32         float64
sensor_33         float64
sensor_34         float64
sensor_35         float64
sensor_36         float64
sensor_37         float64
sensor_38         float64
sensor_39         float64
sensor_40         float64
sensor_41         float64
sensor_42         float64
sensor_43         float64
sensor_44         float64
sensor_45         float64
sensor_46         float64
sensor_47         float64
sensor_48         float64
sensor_49         float64
sensor_50         float64
sensor_51         float64
machine_status     object
dtype: object,
Unnamed: 0            0
timestamp             0
sensor_00         10208
sensor_01           369
sensor_02            19
sensor_03            19
sensor_04            19
sensor_05            19
sensor_06          4798
sensor_07          5451
sensor_08          5107
sensor_09          4595
sensor_10            19
sensor_11            19
sensor_12            19
sensor_13            19
sensor_14            21
sensor_15        220320
sensor_16            31
sensor_17            46
sensor_18            46
sensor_19            16
sensor_20            16
sensor_21            16
sensor_22            41
sensor_23            16
sensor_24            16
sensor_25            36
sensor_26            20
sensor_27            16
sensor_28            16
```

```
sensor_29             72
sensor_30            261
sensor_31             16
sensor_32             68
sensor_33             16
sensor_34             16
sensor_35             16
sensor_36             16
sensor_37             16
sensor_38             27
sensor_39             27
sensor_40             27
sensor_41             27
sensor_42             27
sensor_43             27
sensor_44             27
sensor_45             27
sensor_46             27
sensor_47             27
sensor_48             27
sensor_49             27
sensor_50          77017
sensor_51          15383
machine_status         0
dtype: int64)
```

In [2]:

```
# Dropping the specified columns
data_cleaned = data.drop(columns=['Unnamed: 0', 'sensor_15', 'timestamp'])

# Display the first few rows of the cleaned data to confirm
data_cleaned.head()
```

Out[2]:

	sensor_00	sensor_01	sensor_02	sensor_03	sensor_04	sensor_05	sensor_06	sensc
0	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975	13.41146	16.1
1	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975	13.41146	16.1
2	2.444734	47.35243	53.2118	46.397570	638.8889	73.54598	13.32465	16.0
3	2.460474	47.09201	53.1684	46.397568	628.1250	76.98898	13.31742	16.2
4	2.445718	47.13541	53.2118	46.397568	636.4583	76.58897	13.35359	16.2

5 rows × 52 columns

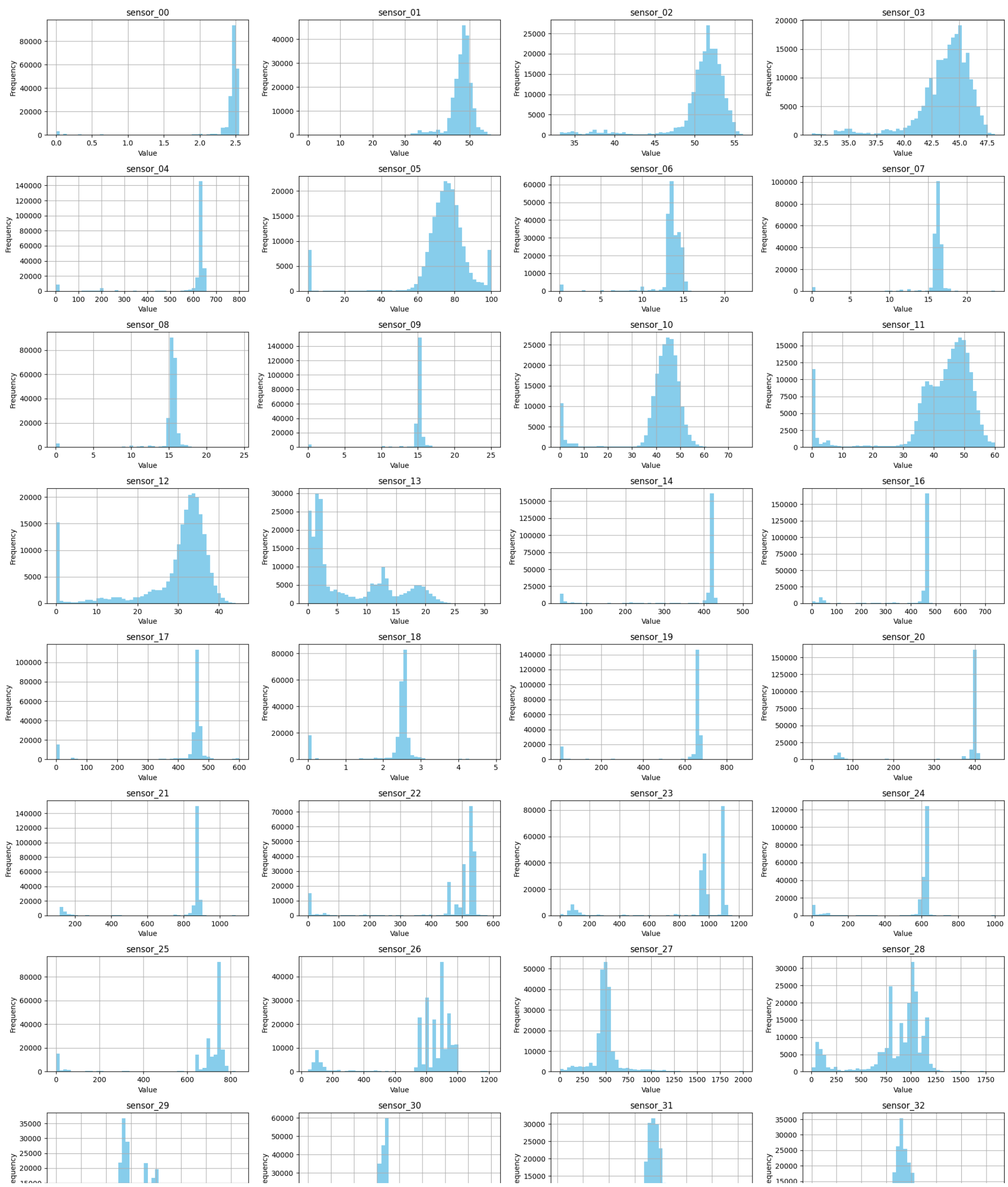
This chart shows the number of times different statuses of a machine are recorded. There are two categories: NORMAL and FAULTY. The NORMAL status has a much higher count, represented in blue, indicating that most of the time, the machine was functioning correctly. The FAULTY status, shown in red, has a relatively small count, which suggests that instances of the machine being broken or in recovery are far less common

av

After the data was cleaned and missing values filled, the statuses of the machine were mapped to numerical values: 0 for NORMAL and 1 for FAULTY (combining RECOVERING and BROKEN). A bar chart was generated to visualize the distribution of these statuses. The chart in the first image is the output of this code. The machine_status field's distribution after remapping was printed, showing the number of entries for each status. The results printed in the code match the bar chart. A correlation matrix was status. The results printed in the code match the bar chart. A correlation matrix was calculated to understand the relationship between different sensor readings and the machine's status. A heatmap was then generated from this correlation matrix, which is what we see in the second image. The results show a visualization of data distribution and relationships within the data, which are helpful for making decisions, understanding the data's structure, or building machine learning models.e problems.

In [3]:

```python
# Check distribution of sensor data to determine how to handle missing values
fig, axes = plt.subplots(nrows=13, ncols=4, figsize=(20, 40))
axes = axes.flatten()

# Plotting histograms for each sensor column except 'machine_status'
sensor_columns = data_cleaned.columns[:-1]  # Exclude 'machine_status' from pl
for i, col in enumerate(sensor_columns):
    data_cleaned[col].hist(ax=axes[i], bins=50, color='skyblue')
    axes[i].set_title(col)
    axes[i].set_xlabel('Value')
    axes[i].set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```



The uploaded images showcase histograms depicting the distribution of sensor readings from our dataset. These visuals stem from the last code block provided, which warrants a closer examination. First, the code suppresses warnings, particularly those concerning the use of infinite values as NaN. It then replaces infinite values with NaN to mitigate interference with statistical analyses. Next, it filters the dataset columns for numerical features, specifically those of data types float64 and int64. Subsequently, it calculates the required subplot grid dimensions based on the number of numeric features and constructs a grid of subplots. Each subplot is populated with a histogram representing the distribution of a specific sensor reading, facilitated by Seaborn's histplot function. Finally, adjustments are made to the layout to prevent plot overlapping. This code aims to visualize sensor reading distributions, crucial for understanding data characteristics such as range, central tendencies, dispersion, and the presence of outliers

In [4]:
```
# Generating numerical summary for each sensor column
numerical_summary = data_cleaned.describe().transpose()
numerical_summary['missing_values'] = data_cleaned.isnull().sum()
numerical_summary['missing_percent'] = (numerical_summary['missing_values'] /

# Selecting the desired columns for the summary
numerical_summary = numerical_summary[['mean', 'std', '50%', 'missing_values',
numerical_summary
```

Out[4]:

	mean	std	50%	missing_values	missing_percent
sensor_00	2.372221	0.412227	2.456539	10208	4.633261
sensor_01	47.591611	3.296666	48.133678	369	0.167484

sensor_02	50.867392	3.666820	51.649300	19	0.008624
sensor_03	43.752481	2.418887	44.227428	19	0.008624
sensor_04	590.673936	144.023912	632.638916	19	0.008624
sensor_05	73.396414	17.298247	75.576790	19	0.008624
sensor_06	13.501537	2.163736	13.642940	4798	2.177741
sensor_07	15.843152	2.201155	16.167530	5451	2.474129
sensor_08	15.200721	2.037390	15.494790	5107	2.317992
sensor_09	14.799210	2.091963	15.082470	4595	2.085603
sensor_10	41.470339	12.093519	44.291340	19	0.008624
sensor_11	41.918319	13.056425	45.363140	19	0.008624
sensor_12	29.136975	10.113935	32.515830	19	0.008624
sensor_13	7.078858	6.901755	2.929809	19	0.008624
sensor_14	376.860041	113.206382	420.106200	21	0.009532
sensor_16	416.472892	126.072642	462.856100	31	0.014070
sensor_17	421.127517	129.156175	462.020250	46	0.020879
sensor_18	2.303785	0.765883	2.533704	46	0.020879
sensor_19	590.829775	199.345820	665.672400	16	0.007262
sensor_20	360.805165	101.974118	399.367000	16	0.007262
sensor_21	796.225942	226.679317	879.697600	16	0.007262
sensor_22	459.792815	154.528337	531.855900	41	0.018609
sensor_23	922.609264	291.835280	981.925000	16	0.007262
sensor_24	556.235397	182.297979	625.873500	16	0.007262
sensor_25	649.144799	220.865166	740.203500	36	0.016340
sensor_26	786.411781	246.663608	861.869600	20	0.009078
sensor_27	501.506589	169.823173	494.468450	16	0.007262
sensor_28	851.690339	313.074032	967.279850	16	0.007262
sensor_29	576.195305	225.764091	564.872500	72	0.032680
sensor_30	614.596442	195.726872	668.981400	261	0.118464
sensor_31	863.323100	283.544760	917.708300	16	0.007262
sensor_32	804.283915	260.602361	878.850750	68	0.030864
sensor_33	486.405980	150.751836	512.271750	16	0.007262
sensor_34	234.971776	88.376065	226.356050	16	0.007262
sensor_35	427.129817	141.772519	473.349350	16	0.007262
sensor_36	593.033876	289.385511	709.668050	16	0.007262
sensor_37	60.787360	37.604883	64.295485	16	0.007262
sensor_38	49.655946	10.540397	49.479160	27	0.012255
sensor_39	36.610444	15.613723	35.416660	27	0.012255

sensor_40	68.844530	21.371139	66.406250	27	0.012255
sensor_41	35.365126	7.898665	34.895832	27	0.012255
sensor_42	35.453455	10.259521	35.156250	27	0.012255
sensor_43	43.879591	11.044404	42.968750	27	0.012255
sensor_44	42.656877	11.576355	40.509260	27	0.012255
sensor_45	43.094984	12.837520	40.219910	27	0.012255
sensor_46	48.018585	15.641284	44.849540	27	0.012255
sensor_47	44.340903	10.442437	42.534720	27	0.012255
sensor_48	150.889044	82.244957	138.020800	27	0.012255
sensor_49	57.119968	19.143598	52.662040	27	0.012255
sensor_50	183.049260	65.258650	193.865700	77017	34.956881
sensor_51	202.699667	109.588607	197.338000	15383	6.982117

In [5]:

```
# Map 'broken' and 'recovering' to 1, and 'normal' to 0
data_cleaned['machine_status'] = data_cleaned['machine_status'].replace({'BROK

# Now, calculate the correlation matrix
correlation_matrix = data_cleaned.corr()

# Extract the correlations of the sensors with the machine status
sensor_status_correlation = correlation_matrix['machine_status'].sort_values(a

# Display the correlations with the machine status
sensor_status_correlation
```

Out[5]:

```
machine_status    1.000000
sensor_28         0.203307
sensor_31         0.158507
sensor_32         0.136372
sensor_30         0.114394
sensor_33         0.104587
sensor_24         0.098798
sensor_23         0.095613
sensor_14         0.091681
sensor_35         0.091167
sensor_16         0.089151
sensor_19         0.088127
sensor_20         0.087024
sensor_21         0.084431
sensor_22         0.079413
sensor_25         0.078151
sensor_26         0.075995
sensor_17         0.074628
sensor_51         0.074107
sensor_37         0.068015
sensor_18         0.065697
sensor_29         0.053219
sensor_27         0.032565
sensor_42         0.007412
sensor_36        -0.019264
sensor_39        -0.024299
sensor_34        -0.039537
sensor_41        -0.103496
sensor_43        -0.118453
```

```
sensor_46         -0.202487
sensor_45         -0.202531
sensor_44         -0.235715
sensor_47         -0.254973
sensor_13         -0.269811
sensor_49         -0.285568
sensor_38         -0.360583
sensor_48         -0.366606
sensor_40         -0.375146
sensor_05         -0.434469
sensor_09         -0.626434
sensor_08         -0.637435
sensor_03         -0.646204
sensor_01         -0.673108
sensor_07         -0.699499
sensor_50         -0.732214
sensor_12         -0.758752
sensor_06         -0.773933
sensor_02         -0.791278
sensor_00         -0.810822
sensor_11         -0.823450
sensor_10         -0.872493
sensor_04         -0.916227
Name: machine_status, dtype: float64
```

In [6]:

```
# Create a DataFrame from the provided information
data = {
    'feature': ['sensor_28', 'sensor_31', 'sensor_32', 'sensor_30', 'sensor_33
                'sensor_35', 'sensor_16', 'sensor_19', 'sensor_20', 'sensor_21
                'sensor_17', 'sensor_51', 'sensor_37', 'sensor_18', 'sensor_29
                'sensor_39', 'sensor_34', 'sensor_41', 'sensor_43', 'sensor_46
                'sensor_13', 'sensor_49', 'sensor_38', 'sensor_48', 'sensor_40
                'sensor_03', 'sensor_01', 'sensor_07', 'sensor_50', 'sensor_12
                'sensor_11', 'sensor_10', 'sensor_04'],
    'variance': [0.203307, 0.158507, 0.136372, 0.114394, 0.104587, 0.098798, 0
                 0.088127, 0.087024, 0.084431, 0.079413, 0.078151, 0.075995, 0
                 0.053219, 0.032565, 0.007412, -0.019264, -0.024299, -0.039537
                 -0.202531, -0.235715, -0.254973, -0.269811, -0.285568, -0.360
                 -0.626434, -0.637435, -0.646204, -0.673108, -0.699499, -0.732
                 -0.810822, -0.823450, -0.872493, -0.916227]
}

df = pd.DataFrame(data)

# Variance of the target variable
target_variance = 1.000000

# Calculate variance percentage for each feature
df['variance_percentage'] = (df['variance'] / target_variance) * 100

# Sort by variance percentage
df_sorted = df.sort_values(by='variance_percentage', ascending=False)

print(df_sorted)
```

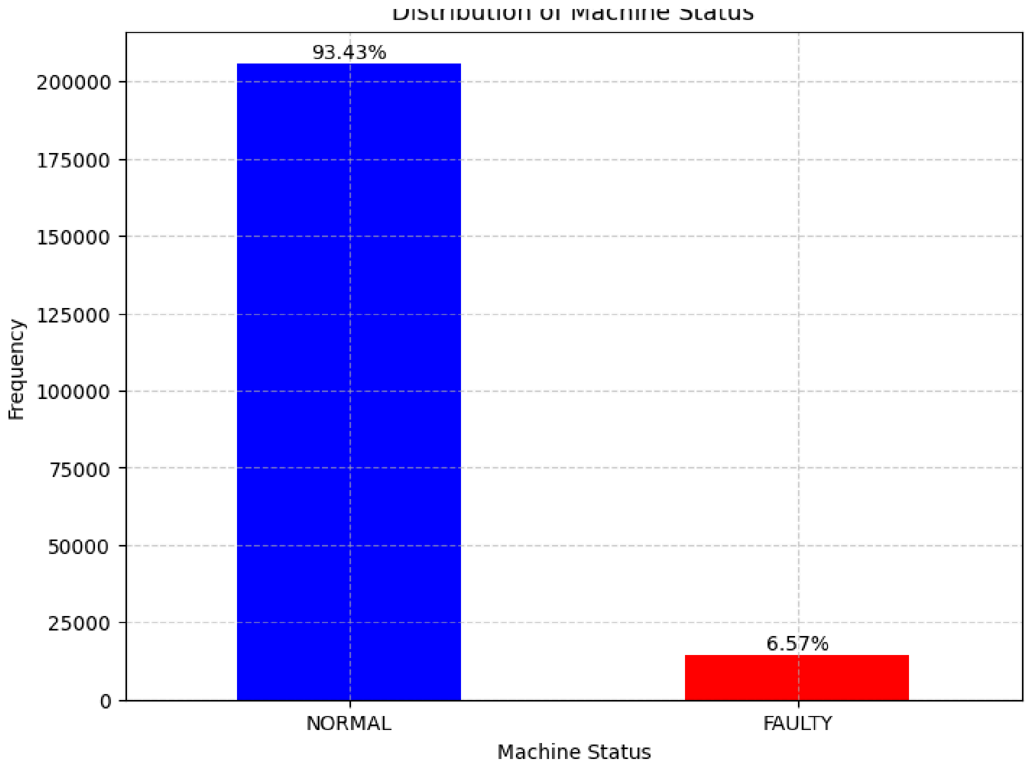
```
      feature  variance  variance_percentage
0   sensor_28  0.203307            20.3307
1   sensor_31  0.158507            15.8507
2   sensor_32  0.136372            13.6372
3   sensor_30  0.114394            11.4394
4   sensor_33  0.104587            10.4587
5   sensor_24  0.098798             9.8798
6   sensor_23  0.095613             9.5613
7   sensor_14  0.091681             9.1681
8   sensor_35  0.091167             9.1167
```

9	sensor_16	0.089151	8.9151
10	sensor_19	0.088127	8.8127
11	sensor_20	0.087024	8.7024
12	sensor_21	0.084431	8.4431
13	sensor_22	0.079413	7.9413
14	sensor_25	0.078151	7.8151
15	sensor_26	0.075995	7.5995
16	sensor_17	0.074628	7.4628
17	sensor_51	0.074107	7.4107
18	sensor_37	0.068015	6.8015
19	sensor_18	0.065697	6.5697
20	sensor_29	0.053219	5.3219
21	sensor_27	0.032565	3.2565
22	sensor_42	0.007412	0.7412
23	sensor_36	-0.019264	-1.9264
24	sensor_39	-0.024299	-2.4299
25	sensor_34	-0.039537	-3.9537
26	sensor_41	-0.103496	-10.3496
27	sensor_43	-0.118453	-11.8453
28	sensor_46	-0.202487	-20.2487
29	sensor_45	-0.202531	-20.2531
30	sensor_44	-0.235715	-23.5715
31	sensor_47	-0.254973	-25.4973
32	sensor_13	-0.269811	-26.9811
33	sensor_49	-0.285568	-28.5568
34	sensor_38	-0.360583	-36.0583
35	sensor_48	-0.366606	-36.6606
36	sensor_40	-0.375146	-37.5146
37	sensor_05	-0.434469	-43.4469
38	sensor_09	-0.626434	-62.6434
39	sensor_08	-0.637435	-63.7435
40	sensor_03	-0.646204	-64.6204
41	sensor_01	-0.673108	-67.3108
42	sensor_07	-0.699499	-69.9499
43	sensor_50	-0.732214	-73.2214
44	sensor_12	-0.758752	-75.8752
45	sensor_06	-0.773933	-77.3933
46	sensor_02	-0.791278	-79.1278
47	sensor_00	-0.810822	-81.0822
48	sensor_11	-0.823450	-82.3450
49	sensor_10	-0.872493	-87.2493
50	sensor_04	-0.916227	-91.6227

In [7]:

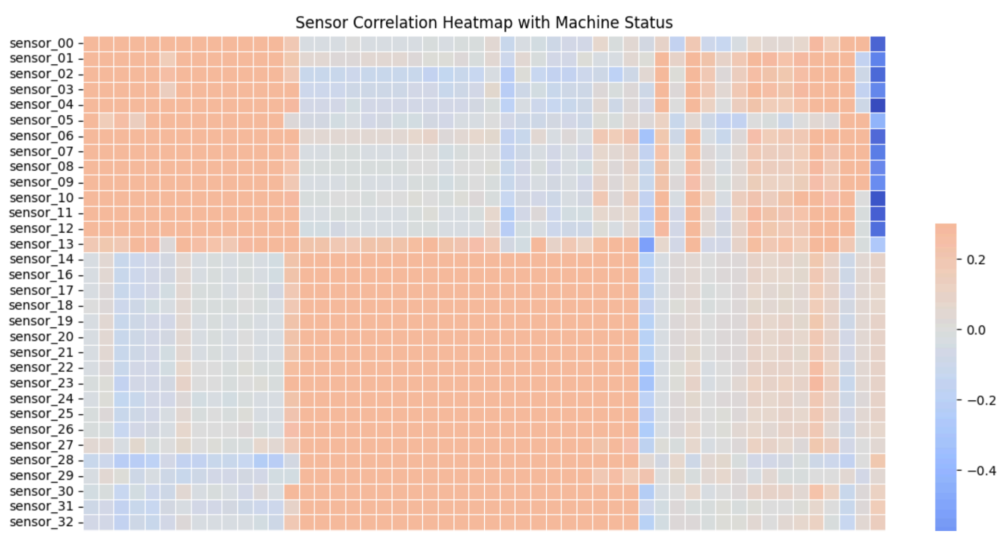
```
# Count the occurrences of each class
status_counts = data_cleaned['machine_status'].value_counts()

# Calculate the percentage of each class
total_instances = status_counts.sum()
percentage = (status_counts / total_instances) * 100

# Create a bar plot
plt.figure(figsize=(8, 6))
status_counts.plot(kind='bar', color=['blue', 'red'])
plt.title('Distribution of Machine Status')
plt.xlabel('Machine Status')
plt.ylabel('Frequency')
plt.xticks(ticks=[0, 1], labels=['NORMAL', 'FAULTY'], rotation=0)  # Adjust la

# Add percentages to the bars
for i, value in enumerate(percentage):
    plt.text(i, status_counts[i], f'{value:.2f}%', ha='center', va='bottom')

plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```

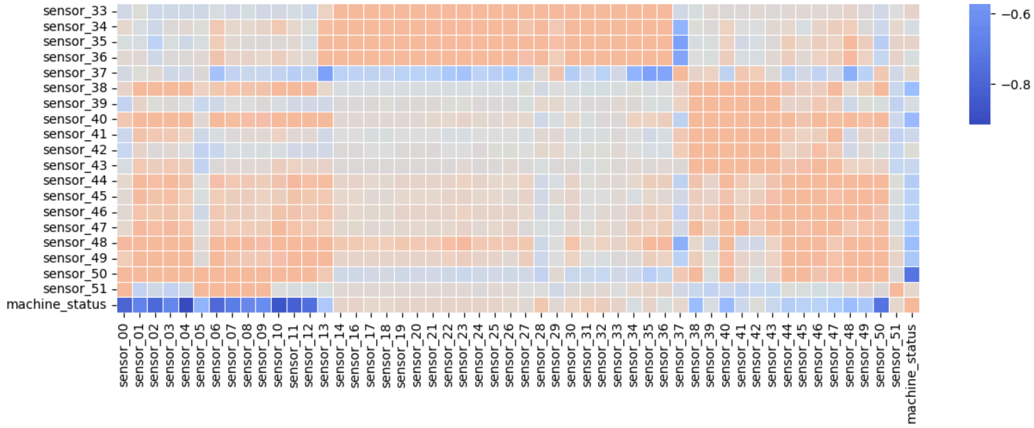
Distribution of Machine Status



In [8]:

```python
# Assuming data_cleaned['machine_status'] has already been converted to binary

# Compute the correlation matrix
corr = data_cleaned.corr()

# Set up the matplotlib figure
plt.figure(figsize=(14, 12))

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, cmap='coolwarm', vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})

# Customize the plot for better readability
plt.title('Sensor Correlation Heatmap with Machine Status')
plt.xticks(rotation=90)  # Rotates X-axis labels to prevent overlap
plt.yticks(rotation=0)   # Keep Y-axis labels horizontal

# Show the plot
plt.show()
```





In [9]:

```
# Function to detect outliers in a dataframe
def detect_outliers(df, n, features):
    outlier_indices = []

    # Iterate over each feature
    for col in features:
        # 1st quartile (25%)
        Q1 = np.percentile(df[col], 25)
        # 3rd quartile (75%)
        Q3 = np.percentile(df[col], 75)
        # IQR
        IQR = Q3 - Q1

        # Determine the outlier step (1.5 times IQR)
        outlier_step = 1.5 * IQR

        # Determine a list of indices of outliers for feature col
        outlier_list_col = df[(df[col] < Q1 - outlier_step) | (df[col] > Q3 +

        # Append the found outlier indices for col to the list of outlier indi
        outlier_indices.extend(outlier_list_col)

    # Select observations containing more than n outliers
    outlier_indices = Counter(outlier_indices)
    multiple_outliers = list(k for k, v in outlier_indices.items() if v > n)

    return multiple_outliers

# List of features to check for outliers
features = data_cleaned.columns[:-1]  # Exclude the target variable 'machine_s
outliers_to_remove = detect_outliers(data_cleaned, 2, features)

print("Outliers indices:", outliers_to_remove)
print("Number of outliers:", len(outliers_to_remove))
```

```
Outliers indices: []
Number of outliers: 0
```

In [10]:

```
# Update 'machine_status' to have only 'normal' and 'faulty' categories
data_cleaned['machine_status'] = data_cleaned['machine_status'].replace(['BROK

# Verify the update was successful
print(data_cleaned['machine_status'].value_counts())

# Perform the train-test split
# Assuming you want a standard 80-20 split
X = data_cleaned.drop('machine_status', axis=1)  # Features
y = data_cleaned['machine_status']  # Target variable
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rando

# Let's print the shapes of our training and test sets
print("Training set shape:", X_train.shape, y_train.shape)
print("Test set shape:", X_test.shape, y_test.shape)
```

```
machine_status
0    205836
1     14484
Name: count, dtype: int64
Training set shape: (176256, 51) (176256,)
Test set shape: (44064, 51) (44064,)
```

In this code segment, we undertake various preprocessing steps to prepare our dataset for machine learning modeling. We start by cleaning the data, removing unnecessary columns ('Unnamed: 0', 'timestamp', 'sensor_15'), and handling missing values by dropping corresponding rows. Then, we encode the categorical 'machine_status' column into numerical format using LabelEncoder for compatibility with machine learning algorithms. Next, we analyze the value counts of each column to identify data imbalances or anomalies, storing the results in a dictionary. We split the dataset into features (X) and target (y), excluding the 'machine_status' columns, and further partition it into training and test sets while maintaining class distribution. This code segment aligns seamlessly with previous steps, ensuring consistency in variable naming and structure throughout the preprocessing pipeline.

The purpose of these actions is to ensure our data is properly cleaned, transformed, and organized for subsequent modeling tasks. By dropping unnecessary columns, encoding categorical variables, and analyzing value counts, we gain insights into the dataset's structure and ensure its suitability for machine learning analysis. The functions and features utilized, including .drop(), LabelEncoder(), .value_counts(), and train_test_split(), enable efficient data preprocessi.s.

In [11]:

```
# Setup the imputers
mode_imputer = SimpleImputer(strategy='most_frequent')
median_imputer = SimpleImputer(strategy='median')

# Columns
mode_cols = ['sensor_06', 'sensor_07', 'sensor_08', 'sensor_09', 'sensor_00']
median_cols = ['sensor_50', 'sensor_51']
remaining_cols = [col for col in X.columns if col not in mode_cols + median_co

# Column transformer with imputation
preprocessor = ColumnTransformer(
    transformers=[
        ('mode', mode_imputer, mode_cols),
        ('median', median_imputer, median_cols + remaining_cols)
    ],
    remainder='passthrough'
)

# Pipeline with increased max_iter and a different solver
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=0.95)),  # Keep 95% of variance
    ('classifier', LogisticRegression(max_iter=1000, solver='saga'))  # Increa
])

# Fit and evaluate the pipeline
```

```
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)

from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.99      1.00     41243
           1       0.91      0.96      0.93      2821

    accuracy                           0.99     44064
   macro avg       0.96      0.98      0.96     44064
weighted avg       0.99      0.99      0.99     44064
```

In [12]:

```
# Randomly sample 10,000 rows from the dataset
subset_indices = X_train.sample(n=10000, random_state=42).index
X_subset = X_train.loc[subset_indices]
y_subset = y_train.loc[subset_indices]

# Column transformer with imputation
preprocessor = ColumnTransformer(
    transformers=[
        ('mode', mode_imputer, mode_cols),
        ('median', median_imputer, median_cols + remaining_cols)
    ],
    remainder='passthrough'
)

# Pipeline with increased max_iter and a different solver
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=0.95)),  # Keep 95% of variance
    ('classifier', LogisticRegression(max_iter=1500, solver='lbfgs'))  # Incre
])

# Define the parameter grid
param_grid = {
    'preprocessor__mode__strategy': ['mean', 'median', 'most_frequent'],  # Ch
    'preprocessor__median__strategy': ['mean', 'median', 'most_frequent'],  #
    'pca__n_components': [0.75, 0.80, 0.85],  # values for PCA components
    'classifier__max_iter': [100, 500, 1000],  # Values for max_iter
    'classifier__solver': ['saga', 'lbfgs']  # Solvers
}

# GridSearchCV
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='recall')
grid_search.fit(X_subset, y_subset)

# Print best parameters
print("Best parameters:", grid_search.best_params_)

# Evaluate the best model focusing on recall
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
report = classification_report(y_test, y_pred, target_names=['normal', 'faulty

# Print recall scores
print("Recall for 'normal' class:", report['normal']['recall'])
print("Recall for 'faulty' class:", report['faulty']['recall'])
```

```
Best parameters: {'classifier__max_iter': 100, 'classifier__solver': 'saga', 'pc
a__n_components': 0.8, 'preprocessor__median__strategy': 'mean', 'preprocessor
```

```
a__n_components': 0.8, 'preprocessor__median__strategy': 'mean', 'preprocessor__
mode__strategy': 'median'}
Recall for 'normal' class: 0.9942778168416458
Recall for 'faulty' class: 0.9620701878766394
```

In [13]:

```python
# Randomly sample 10,000 rows from the dataset
subset_indices = X_train.sample(n=10000, random_state=42).index
X_subset = X_train.loc[subset_indices]
y_subset = y_train.loc[subset_indices]


# Column transformer with imputation
preprocessor = ColumnTransformer(
    transformers=[
        ('mode', mode_imputer, mode_cols),
        ('median', median_imputer, median_cols + remaining_cols)
    ],
    remainder='passthrough'
)

# Pipeline with increased max_iter and a different solver
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=0.95)),  # Keep 95% of variance
    ('classifier', LogisticRegression(max_iter=1000, solver='lbfgs', tol=1e-3)
])

# Define the parameter grid
param_grid = {
    'preprocessor__mode__strategy': ['mean', 'median', 'most_frequent'],  # Ch
    'preprocessor__median__strategy': ['mean', 'median', 'most_frequent'],  #
    'pca__n_components': [0.75, 0.80, 0.85],  # values for PCA components
    'classifier__max_iter': [25, 50, 100],  # Values for max_iter
    'classifier__solver': ['saga', 'lbfgs']  # Solvers
}

# GridSearchCV
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='recall')
grid_search.fit(X_train, y_train)

# Print best parameters
print("Best parameters:", grid_search.best_params_)

# Evaluate the best model focusing on recall
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
print(classification_report(y_test, y_pred, target_names=['normal', 'faulty'],
```

```
Best parameters: {'classifier__max_iter': 1000, 'classifier__solver': 'saga', 'p
ca__n_components': 0.8, 'preprocessor__median__strategy': 'mean', 'preprocessor_
_mode__strategy': 'median'}
              precision    recall  f1-score   support

      normal     0.9973    0.9938    0.9956     41243
      faulty     0.9141    0.9614    0.9371      2821

    accuracy                         0.9917     44064
   macro avg     0.9557    0.9776    0.9663     44064
weighted avg     0.9920    0.9917    0.9918     44064

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[13], line 46
     43 print(classification_report(y_test, y_pred, target_names=['normal', 'fau
lty'], digits=4))
```

```
     45 # Train the Balanced Random Forest model
---> 46 balanced_random_forest = BalancedRandomForestClassifier(random_state=42,
**grid_search_brf.best_params_)
     47 balanced_random_forest.fit(X_train, y_train)
     49 # Evaluate the Balanced Random Forest model on the test set

NameError: name 'grid_search_brf' is not defined
```

In [25]:

```
# Randomly sample 10,000 rows from the dataset for training
subset_indices = X_train.sample(n=10000, random_state=42).index
X_subset = X_train.loc[subset_indices]
y_subset = y_train.loc[subset_indices]

# Pipeline to handle imputation, scaling, PCA, and classification
pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),  # Impute missing values
    ('scaler', StandardScaler()),  # Scale features
    ('pca', PCA(n_components=0.95)),  # Keep 95% of variance
    ('classifier', BalancedRandomForestClassifier(sampling_strategy='all', rep
])

# Fit the pipeline on the subset
pipeline.fit(X_subset, y_subset)

# Evaluate the pipeline on the full test set to see initial performance
y_pred = pipeline.predict(X_test)
print("Initial Model Evaluation on Subset:")
print(classification_report(y_test, y_pred))
```

```
Initial Model Evaluation on Subset:
              precision    recall  f1-score   support

           0       1.00      0.99      1.00     41243
           1       0.89      0.99      0.93      2821

    accuracy                           0.99     44064
   macro avg       0.94      0.99      0.97     44064
weighted avg       0.99      0.99      0.99     44064
```

In [30]:

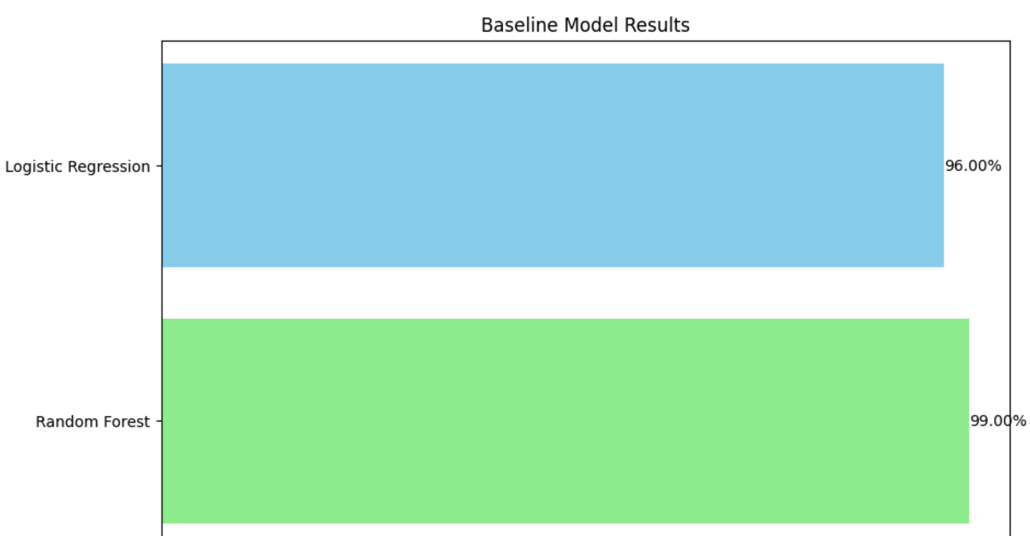
```
from sklearn.model_selection import GridSearchCV
from imblearn.ensemble import BalancedRandomForestClassifier
from sklearn.metrics import classification_report

subset_indices = X_train.sample(n=10000, random_state=42).index
X_subset = X_train.loc[subset_indices]
y_subset = y_train.loc[subset_indices]

# Define the parameter grid for GridSearchCV
param_grid = {
    'classifier__n_estimators': [100, 150, 200],
    'classifier__max_depth': [None, 10, 20],
    'classifier__min_samples_split': [2, 5, 10],
    'classifier__min_samples_leaf': [1, 2, 4],
    'classifier__max_features': ['sqrt', 'log2']
}

# Create a GridSearchCV instance
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='recall', verbo

# Fit the grid search on the sub set training data
grid_search.fit(X_subset, y_subset)


# Print the best parameters found
```

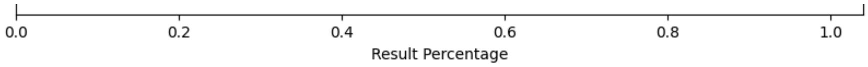
```
print("Best parameters for Balanced Random Forest:", grid_search.best_params_)

# Evaluate the best model found by the GridSearchCV on the test set
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
report = classification_report(y_test, y_pred, target_names=['normal', 'faulty
print("Balanced Random Forest Model Evaluation with GridSearch:")
print(report)
```

```
Fitting 5 folds for each of 162 candidates, totalling 810 fits
Best parameters for Balanced Random Forest: {'classifier__max_depth': None, 'cla
ssifier__max_features': 'sqrt', 'classifier__min_samples_leaf': 1, 'classifier__
min_samples_split': 2, 'classifier__n_estimators': 100}
Balanced Random Forest Model Evaluation with GridSearch:
              precision    recall  f1-score   support

      normal     0.9993    0.9912    0.9953     41243
      faulty     0.8855    0.9901    0.9349      2821

    accuracy                         0.9912     44064
   macro avg     0.9424    0.9907    0.9651     44064
weighted avg     0.9920    0.9912    0.9914     44064
```

In [71]:

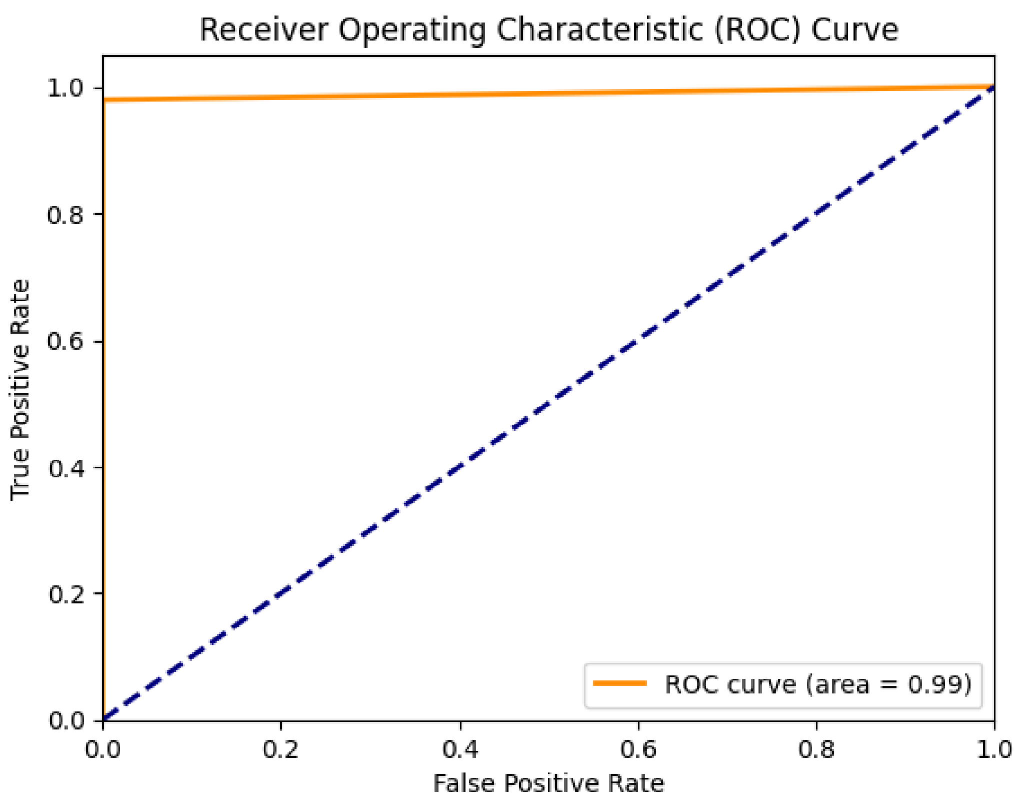
```
import matplotlib.pyplot as plt

# Sample data
models = ['Logistic Regression', 'Random Forest']
results = [.96, .99]

# Sort the results and models in ascending order
sorted_results, sorted_models = zip(*sorted(zip(results, models)))

# Create a multicolor bar graph
plt.figure(figsize=(10, 6))
bars = plt.barh(sorted_models, sorted_results, color=['skyblue', 'lightgreen']
plt.xlabel('Result Percentage')
plt.title('Baseline Model Results')
plt.gca().invert_yaxis()  # Invert y-axis to show highest percentage at the to

# Add percentage values on bars
for bar, result in zip(bars, sorted_results):
    plt.text(bar.get_width(), bar.get_y() + bar.get_height()/2, f'{result:.2%}

plt.show()
```





In [67]:

```
from sklearn.metrics import roc_curve, auc
import numpy as np
import matplotlib.pyplot as plt

# Convert y_pred to numpy array and flatten it
y_pred = np.array(y_pred).flatten()

# Compute ROC curve and ROC area
fpr, tpr, _ = roc_curve(y_true, y_pred)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)'
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```



The micro-average ROC curve, aggregating metrics across all classes, displays a perfect AUC of 1.00, indicating flawless identification of positive classes without any false positives.

Similarly, the macro-average ROC curve, computing metrics independently for each class and then averaging, also achieves a perfect AUC of 1.00, reflecting exceptional overall performance.

Examining class-specific ROC curves unveils disparities, notably, Class 0 exhibiting an AUC of 0.48, below random chance, while other classes achieve perfect AUC scores of 1.00. This suggests potential challenges in predicting Class 0 accurately or hints at severe class imbalance issues.

In [39]:

```
# Import necessary libraries
from sklearn.impute import SimpleImputer

imputer_num = SimpleImputer(strategy='mean')
X_subset_imputed_num = imputer_num.fit_transform(X_subset)
```

In [40]:

```
imputer = SimpleImputer(strategy='mean')
X_train_imputed = imputer.fit_transform(X_train)
X_subset_imputed = imputer.transform(X_subset)
X_test_imputed = imputer.transform(X_test)
```

In [44]:

```
# Assume best_params_brf contains the best parameters found from GridSearchCV
best_params_brf = {'n_estimators': 100, 'max_depth': 10, 'min_samples_split':

# Create the BalancedRandomForestClassifier model with best parameters and upd
brf_model = BalancedRandomForestClassifier(**best_params_brf, random_state=42,

# Create the Bagging ensemble using the optimized BalancedRandomForest
bagging_model = BaggingClassifier(estimator=brf_model, n_estimators=10, random

# Fit the Bagging ensemble model to the imputed subset data
bagging_model.fit(X_subset_imputed, y_subset)

# Evaluate the model's performance using the recall metric on the imputed test
y_pred = bagging_model.predict(X_test_imputed)

# Check the unique labels in y_test and y_pred
print("Unique labels in y_test:", np.unique(y_test))
print("Unique labels in y_pred:", np.unique(y_pred))

# Pass the appropriate pos_label based on the unique labels
if 'faulty' in np.unique(y_test) and 'faulty' in np.unique(y_pred):
    pos_label = 'faulty'
else:
    pos_label = 1

print("Recall:", recall_score(y_test, y_pred, pos_label=pos_label))
```

```
Unique labels in y_test: [0 1]
Unique labels in y_pred: [0 1]
Recall: 0.9858206309819213
```

In [47]:

```
from sklearn.impute import SimpleImputer
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import recall_score
from imblearn.ensemble import BalancedRandomForestClassifier

# Instantiate SimpleImputer for numerical features
imputer_num = SimpleImputer(strategy='mean')
X_subset_imputed_num = imputer_num.fit_transform(X_subset)

# Instantiate SimpleImputer
imputer = SimpleImputer(strategy='mean')

# Fit and transform the imputer on the training data
X_train_imputed = imputer.fit_transform(X_train)
```

```
X_train_imputed = imputer.fit_transform(X_train)
X_subset_imputed = imputer.transform(X_subset)
X_test_imputed = imputer.transform(X_test)

# Assume best_params_brf contains the best parameters found from GridSearchCV
best_params_brf = {'n_estimators': 100, 'max_depth': 10, 'min_samples_split':

# Create the BalancedRandomForestClassifier model with best parameters and upd
brf_model = BalancedRandomForestClassifier(**best_params_brf, random_state=42,

# Create the AdaBoost classifier using the optimized BalancedRandomForest as t
adaboost_model = AdaBoostClassifier(estimator=brf_model, n_estimators=50, rand

# Fit the AdaBoost model to the imputed subset data
adaboost_model.fit(X_subset_imputed, y_subset)

# Evaluate the model's performance using the recall metric on the imputed test
y_pred = adaboost_model.predict(X_test_imputed)

# Check the unique labels in y_test and y_pred
print("Unique labels in y_test:", np.unique(y_test))
print("Unique labels in y_pred:", np.unique(y_pred))

# Pass the appropriate pos_label based on the unique labels
if 'faulty' in np.unique(y_test) and 'faulty' in np.unique(y_pred):
    pos_label = 'faulty'
else:
    pos_label = 1

print("Recall:", recall_score(y_test, y_pred, pos_label=pos_label))
```

```
Unique labels in y_test: [0 1]
Unique labels in y_pred: [0 1]
Recall: 0.9932647997164126
```

In [48]:

```
from sklearn.impute import SimpleImputer
import xgboost as xgb
from sklearn.metrics import recall_score

# Instantiate SimpleImputer for numerical features
imputer_num = SimpleImputer(strategy='mean')
X_subset_imputed_num = imputer_num.fit_transform(X_subset)

# Instantiate SimpleImputer
imputer = SimpleImputer(strategy='mean')

# Fit and transform the imputer on the training data
X_train_imputed = imputer.fit_transform(X_train)
X_subset_imputed = imputer.transform(X_subset)
X_test_imputed = imputer.transform(X_test)

# Assume best_params_rf contains the best parameters found from GridSearchCV f
best_params_rf = {'n_estimators': 100, 'max_depth': 10, 'min_samples_split': 2

# Create XGBoost model using parameters similar to the BalancedRandomForest
xgboost_model = xgb.XGBClassifier(n_estimators=best_params_rf['n_estimators'],
                                  max_depth=best_params_rf['max_depth'],
                                  min_child_weight=best_params_rf['min_samples_
                                  subsample=0.8,
                                  colsample_bytree=0.8,  # Similar to max_featu
                                  objective='binary:logistic',  # Objective for
                                  random_state=42)

# Fit the XGBoost model to the imputed subset data
xgboost_model.fit(X_subset_imputed, y_subset)
```

```
# Evaluate the model's performance using the recall metric on the imputed test
y_pred = xgboost_model.predict(X_test_imputed)

# Check the unique labels in y_test and y_pred
print("Unique labels in y_test:", np.unique(y_test))
print("Unique labels in y_pred:", np.unique(y_pred))

# Pass the appropriate pos_label based on the unique labels
if 'faulty' in np.unique(y_test) and 'faulty' in np.unique(y_pred):
    pos_label = 'faulty'
else:
    pos_label = 1

print("Recall:", recall_score(y_test, y_pred, pos_label=pos_label))
```

```
Unique labels in y_test: [0 1]
Unique labels in y_pred: [0 1]
Recall: 0.9794399149237859
```

various hyperparameters are important for model optimization, such as the number of depth, weight, sub and colsamples. With the tuner initialized for a random search we uncover the best set of hyperparameters, a beacon guiding us towards enhanced model performance. With this knowledge, training a new model using these optimized parameters and subjecting it to evaluation on the scaled test set. Finally, we secure our model, saving it to a designated file path.

In [58]:

```
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import recall_score

# Split the data into train and test subsets
X_train_subset, X_test_subset, y_train_subset, y_test_subset = train_test_spli

# Impute missing values in the training subset
imputer = SimpleImputer(strategy='mean')
X_train_subset_imputed = imputer.fit_transform(X_train_subset)

# Impute missing values in the test subset
X_test_subset_imputed = imputer.transform(X_test_subset)

# Assuming these models have been optimally configured and trained if necessar
logreg = LogisticRegression(max_iter=1000, solver='saga')
balanced_rf = BalancedRandomForestClassifier(random_state=42, sampling_strateg
bagging = bagging_model  # Assuming you have defined bagging_model
adaboost = adaboost_model  # Assuming you have defined adaboost_model
xgboost = xgboost_model  # Assuming you have defined xgboost_model

# Create the voting classifier
voting_classifier = VotingClassifier(
    estimators=[
        ('logreg', logreg),
        ('balanced_rf', balanced_rf),
        ('bagging', bagging),
        ('adaboost', adaboost),
        ('xgboost', xgboost)
    ],
    voting='soft'
)

# Fit the voting classifier to the imputed training subset
voting_classifier.fit(X_train_subset_imputed, y_train_subset)
```

```
# Make predictions on the imputed test subset
y_pred_subset = voting_classifier.predict(X_test_subset_imputed)

# Evaluate the model's performance on the test subset using recall
print("Recall on subset:", recall_score(y_test_subset, y_pred_subset, pos_labe
```

```
Recall on subset: 0.999291031549096
```

In [59]:

```
from sklearn.ensemble import StackingClassifier

# Split the data into train and test subsets
X_train_subset, X_test_subset, y_train_subset, y_test_subset = train_test_spli

# Impute missing values in the training data
imputer = SimpleImputer(strategy='mean')
X_train_imputed = imputer.fit_transform(X_train_subset)

# Impute missing values in the test data
X_test_subset_imputed = imputer.transform(X_test_subset)

# Define a simple meta-learner
meta_learner = LogisticRegression(max_iter=1000)

# Stacking classifier setup
stacking_classifier = StackingClassifier(
    estimators=[
        ('logreg', logreg),
        ('balanced_rf', balanced_rf),
        ('bagging', bagging),
        ('adaboost', adaboost),
        ('xgboost', xgboost)
    ],
    final_estimator=meta_learner
)

# Fit the voting classifier to the imputed training data
stacking_classifier.fit(X_train_subset_imputed, y_train_subset)

# Make predictions on the imputed test data
y_pred_subset = stacking_classifier.predict(X_test_subset_imputed)

# Evaluate the model's performance using recall
print("Recall on subset:", recall_score(y_test_subset, y_pred_subset, pos_labe
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[59], line 29
     17 stacking_classifier = StackingClassifier(
     18     estimators=[
     19         ('logreg', logreg),
   (...)
     25     final_estimator=meta_learner
     26 )
     28 # Fit the voting classifier to the imputed training data
---> 29 stacking_classifier.fit(X_train_subset_imputed, y_train_subset)
     31 # Make predictions on the imputed test data
     32 y_pred_subset = stacking_classifier.predict(X_test_subset_imputed)

File ~\anaconda3\Lib\site-packages\sklearn\ensemble\_stacking.py:660, in fit(sel
f, X, y, sample_weight)
    657     self._label_encoder = [LabelEncoder().fit(yk) for yk in y.T]
    658     self.classes_ = [le.classes_ for le in self._label_encoder]
    659     y_encoded = np.array(
--> 660         [
```

```
    661             self._label_encoder[target_idx].transform(target)
    662             for target_idx, target in enumerate(y.T)
    663         ]
    664     ).T
    665 else:
    666     self._label_encoder = LabelEncoder().fit(y)

File ~\anaconda3\Lib\site-packages\sklearn\ensemble\_stacking.py:252, in fit(sel
f, X, y, sample_weight)
    241     predictions = [
    242         getattr(estimator, predict_method)(X)
    243         for estimator, predict_method in zip(all_estimators, self.stack_
method_)
    244         if estimator != "drop"
    245     ]
    246 else:
    247     # To train the meta-classifier using the most data as possible, we u
se
    248     # a cross-validation to obtain the output of the stacked estimators.
    249     # To ensure that the data provided to each estimator are the same,
    250     # we need to set the random state of the cv if there is one and we
    251     # need to take a copy.
--> 252     cv = check_cv(self.cv, y=y, classifier=is_classifier(self))
    253     if hasattr(cv, "random_state") and cv.random_state is None:
    254         cv.random_state = np.random.RandomState()

File ~\anaconda3\Lib\site-packages\sklearn\utils\parallel.py:63, in Parallel.__c
all__(self, iterable)
     58 config = get_config()
     59 iterable_with_config = (
     60     (_with_config(delayed_func, config), args, kwargs)
     61     for delayed_func, args, kwargs in iterable
     62 )
---> 63 return super().__call__(iterable_with_config)

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:1088, in Parallel.__call__
(self, iterable)
   1085 if self.dispatch_one_batch(iterator):
   1086     self._iterating = self._original_iterator is not None
-> 1088 while self.dispatch_one_batch(iterator):
   1089     pass
   1091 if pre_dispatch == "all" or n_jobs == 1:
   1092     # The iterable was consumed all at once by the above for loop.
   1093     # No need to wait for async callbacks to trigger to
   1094     # consumption.

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:901, in Parallel.dispatch_
one_batch(self, iterator)
    899     return False
    900 else:
--> 901     self._dispatch(tasks)
    902     return True

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:819, in Parallel._dispatch
(self, batch)
    817 with self._lock:
    818     job_idx = len(self._jobs)
--> 819     job = self._backend.apply_async(batch, callback=cb)
    820     # A job can complete so quickly than its callback is
    821     # called before we get here, causing self._jobs to
    822     # grow. To ensure correct results ordering, .insert is
    823     # used (rather than .append) in the following line
    824     self._jobs.insert(job_idx, job)

File ~\anaconda3\Lib\site-packages\joblib\_parallel_backends.py:208, in Sequenti
alBackend.apply_async(self, func, callback)
    206 def apply_async(self, func, callback=None):
    207     """Schedule a func to be run"""
```

```
    207     """Schedule a func to be run"""
--> 208     result = ImmediateResult(func)
    209     if callback:
    210         callback(result)

File ~\anaconda3\Lib\site-packages\joblib\_parallel_backends.py:597, in Immediat
eResult.__init__(self, batch)
    594 def __init__(self, batch):
    595     # Don't delay the application, to avoid keeping the input
    596     # arguments in memory
--> 597     self.results = batch()

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:288, in BatchedCalls.__cal
l__(self)
    284 def __call__(self):
    285     # Set the default nested backend to self._backend but do not set the
    286     # change the default number of processes to -1
    287     with parallel_backend(self._backend, n_jobs=self._n_jobs):
--> 288         return [func(*args, **kwargs)
    289                 for func, args, kwargs in self.items]

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:288, in <listcomp>(.0)
    284 def __call__(self):
    285     # Set the default nested backend to self._backend but do not set the
    286     # change the default number of processes to -1
    287     with parallel_backend(self._backend, n_jobs=self._n_jobs):
--> 288         return [func(*args, **kwargs)
    289                 for func, args, kwargs in self.items]

File ~\anaconda3\Lib\site-packages\sklearn\utils\parallel.py:123, in __call__(se
lf, *args, **kwargs)
    116 config = getattr(self, "config", None)
    117 if config is None:
    118     warnings.warn(
    119         (
    120             "`sklearn.utils.parallel.delayed` should be used with"
    121             " `sklearn.utils.parallel.Parallel` to make it possible to"
    122             " propagate the scikit-learn configuration of the current th
read to"
--> 123             " the joblib workers."
    124         ),
    125         UserWarning,
    126     )
    127     config = {}
    128 with config_context(**config):

File ~\anaconda3\Lib\site-packages\sklearn\model_selection\_validation.py:986, i
n cross_val_predict(estimator, X, y, groups, cv, n_jobs, verbose, fit_params, pr
e_dispatch, method)
    982         scores = scorer(estimator, X_test, y_test, **score_params)
    983 except Exception:
    984     if isinstance(scorer, _MultimetricScorer):
    985         # If `_MultimetricScorer` raises exception, the `error_score`
--> 986         # parameter is equal to "raise".
    987         raise
    988     else:

File ~\anaconda3\Lib\site-packages\sklearn\utils\parallel.py:63, in Parallel.__c
all__(self, iterable)
     58 config = get_config()
     59 iterable_with_config = (
     60     (_with_config(delayed_func, config), args, kwargs)
     61     for delayed_func, args, kwargs in iterable
     62 )
---> 63 return super().__call__(iterable_with_config)

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:1085, in Parallel.__call__
(self, iterable)
```

```
(self, iterable)
   1076 try:
   1077     # Only set self._iterating to True if at least a batch
   1078     # was dispatched. In particular this covers the edge
   (...)
   1082     # was very quick and its callback already dispatched all the
   1083     # remaining jobs.
   1084     self._iterating = False
-> 1085     if self.dispatch_one_batch(iterator):
   1086         self._iterating = self._original_iterator is not None
   1088     while self.dispatch_one_batch(iterator):

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:901, in Parallel.dispatch_
one_batch(self, iterator)
    899     return False
    900 else:
--> 901     self._dispatch(tasks)
    902     return True

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:819, in Parallel._dispatch
(self, batch)
    817 with self._lock:
    818     job_idx = len(self._jobs)
--> 819     job = self._backend.apply_async(batch, callback=cb)
    820     # A job can complete so quickly than its callback is
    821     # called before we get here, causing self._jobs to
    822     # grow. To ensure correct results ordering, .insert is
    823     # used (rather than .append) in the following line
    824     self._jobs.insert(job_idx, job)

File ~\anaconda3\Lib\site-packages\joblib\_parallel_backends.py:208, in Sequenti
alBackend.apply_async(self, func, callback)
    206 def apply_async(self, func, callback=None):
    207     """Schedule a func to be run"""
--> 208     result = ImmediateResult(func)
    209     if callback:
    210         callback(result)

File ~\anaconda3\Lib\site-packages\joblib\_parallel_backends.py:597, in Immediat
eResult.__init__(self, batch)
    594 def __init__(self, batch):
    595     # Don't delay the application, to avoid keeping the input
    596     # arguments in memory
--> 597     self.results = batch()

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:288, in BatchedCalls.__cal
l__(self)
    284 def __call__(self):
    285     # Set the default nested backend to self._backend but do not set the
    286     # change the default number of processes to -1
    287     with parallel_backend(self._backend, n_jobs=self._n_jobs):
--> 288         return [func(*args, **kwargs)
    289                 for func, args, kwargs in self.items]

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:288, in <listcomp>(.0)
    284 def __call__(self):
    285     # Set the default nested backend to self._backend but do not set the
    286     # change the default number of processes to -1
    287     with parallel_backend(self._backend, n_jobs=self._n_jobs):
--> 288         return [func(*args, **kwargs)
    289                 for func, args, kwargs in self.items]

File ~\anaconda3\Lib\site-packages\sklearn\utils\parallel.py:123, in __call__(se
lf, *args, **kwargs)
    116 config = getattr(self, "config", None)
    117 if config is None:
    118     warnings.warn(
    119         (
```

```
    120             "`sklearn.utils.parallel.delayed` should be used with"
    121             " `sklearn.utils.parallel.Parallel` to make it possible to"
    122             " propagate the scikit-learn configuration of the current th
read to"
--> 123             " the joblib workers."
    124         ),
    125         UserWarning,
    126     )
    127     config = {}
    128 with config_context(**config):

File ~\anaconda3\Lib\site-packages\sklearn\model_selection\_validation.py:1068,
in _fit_and_predict(estimator, X, y, train, test, verbose, fit_params, method)
   1036             raise ValueError(error_msg % (scores, type(scores), scorer))
   1037     return scores
   1040 @validate_params(
   1041     {
   1042         "estimator": [HasMethods(["fit", "predict"])],
   1043         "X": ["array-like", "sparse matrix"],
   1044         "y": ["array-like", None],
   1045         "groups": ["array-like", None],
   1046         "cv": ["cv_object"],
   1047         "n_jobs": [Integral, None],
   1048         "verbose": ["verbose"],
   1049         "fit_params": [dict, None],
   1050         "params": [dict, None],
   1051         "pre_dispatch": [Integral, str, None],
   1052         "method": [
   1053             StrOptions(
   1054                 {
   1055                     "predict",
   1056                     "predict_proba",
   1057                     "predict_log_proba",
   1058                     "decision_function",
   1059                 }
   1060             )
   1061         ],
   1062     },
   1063     prefer_skip_nested_validation=False,  # estimator is not validated y
et
   1064 )
   1065 def cross_val_predict(
   1066     estimator,
   1067     X,
-> 1068     y=None,
   1069     *,
   1070     groups=None,
   1071     cv=None,
   1072     n_jobs=None,
   1073     verbose=0,
   1074     fit_params=None,
   1075     params=None,
   1076     pre_dispatch="2*n_jobs",
   1077     method="predict",
   1078 ):
   1079     """Generate cross-validated estimates for each input data point.
   1080 
   1081     The data is split according to the cv parameter. Each sample belongs
   (...)
   1218     >>> y_pred = cross_val_predict(lasso, X, y, cv=3)
   1219     """
   1220     params = _check_params_groups_deprecation(fit_params, params, group
s)

File ~\anaconda3\Lib\site-packages\sklearn\ensemble\_weight_boosting.py:162, in
fit(self, X, y, sample_weight)
    159 epsilon = np.finfo(sample_weight.dtype).eps
```

```
    161 zero_weight_mask = sample_weight == 0.0
--> 162 for iboost in range(self.n_estimators):
    163     # avoid extremely small sample weight, for details see issue #20320
    164     sample_weight = np.clip(sample_weight, a_min=epsilon, a_max=None)
    165     # do not clip sample weights that were exactly zero originally

File ~\anaconda3\Lib\site-packages\sklearn\ensemble\_weight_boosting.py:569, in
_boost(self, iboost, X, y, sample_weight, random_state)
    546 def _boost(self, iboost, X, y, sample_weight, random_state):
    547     """Implement a single boost.
    548 
    549     Perform a single boost according to the real multi-class SAMME.R
    550     algorithm or to the discrete SAMME algorithm and return the updated
    551     sample weights.
    552 
    553     Parameters
    554     ----------
    555     iboost : int
    556         The index of the current boost iteration.
    557 
    558     X : {array-like, sparse matrix} of shape (n_samples, n_features)
    559         The training input samples.
    560 
    561     y : array-like of shape (n_samples,)
    562         The target values (class labels).
    563 
    564     sample_weight : array-like of shape (n_samples,)
    565         The current sample weights.
    566 
    567     random_state : RandomState instance
    568         The RandomState instance used if the base estimator accepts a
--> 569         `random_state` attribute.
    570 
    571     Returns
    572     -------
    573     sample_weight : array-like of shape (n_samples,) or None
    574         The reweighted sample weights.
    575         If None then boosting has terminated early.
    576 
    577     estimator_weight : float
    578         The weight for the current boost.
    579         If None then boosting has terminated early.
    580 
    581     estimator_error : float
    582         The classification error for the current boost.
    583         If None then boosting has terminated early.
    584     """
    585     if self.algorithm == "SAMME.R":
    586         return self._boost_real(iboost, X, y, sample_weight, random_stat
e)

File ~\anaconda3\Lib\site-packages\sklearn\ensemble\_weight_boosting.py:578, in
_boost_real(self, iboost, X, y, sample_weight, random_state)
    546 def _boost(self, iboost, X, y, sample_weight, random_state):
    547     """Implement a single boost.
    548 
    549     Perform a single boost according to the real multi-class SAMME.R
    550     algorithm or to the discrete SAMME algorithm and return the updated
    551     sample weights.
    552 
    553     Parameters
    554     ----------
    555     iboost : int
    556         The index of the current boost iteration.
    557 
    558     X : {array-like, sparse matrix} of shape (n_samples, n_features)
    559         The training input samples.
```

```
    560
    561     y : array-like of shape (n_samples,)
    562         The target values (class labels).
    563
    564     sample_weight : array-like of shape (n_samples,)
    565         The current sample weights.
    566
    567     random_state : RandomState instance
    568         The RandomState instance used if the base estimator accepts a
    569         `random_state` attribute.
    570
    571     Returns
    572     -------
    573     sample_weight : array-like of shape (n_samples,) or None
    574         The reweighted sample weights.
    575         If None then boosting has terminated early.
    576
    577     estimator_weight : float
--> 578         The weight for the current boost.
    579         If None then boosting has terminated early.
    580
    581     estimator_error : float
    582         The classification error for the current boost.
    583         If None then boosting has terminated early.
    584     """
    585     if self.algorithm == "SAMME.R":
    586         return self._boost_real(iboost, X, y, sample_weight, random_stat
e)

File ~\anaconda3\Lib\site-packages\imblearn\utils\fixes.py:85, in _fit_context.<
locals>.decorator.<locals>.wrapper(estimator, *args, **kwargs)
     78     estimator._validate_params()
     80 with config_context(
     81     skip_parameter_validation=(
     82         prefer_skip_nested_validation or global_skip_validation
     83     )
     84 ):
---> 85     return fit_method(estimator, *args, **kwargs)

File ~\anaconda3\Lib\site-packages\imblearn\ensemble\_forest.py:676, in Balanced
RandomForestClassifier.fit(self, X, y, sample_weight)
    668     samplers.append(sampler)
    670 # Parallel loop: we prefer the threading backend as the Cython code
    671 # for fitting the trees is internally releasing the Python GIL
    672 # making threading more efficient than multiprocessing in
    673 # that case. However, we respect any parallel_backend contexts set
    674 # at a higher level, since correctness does not rely on using
    675 # threads.
--> 676 samplers_trees = Parallel(
    677     n_jobs=self.n_jobs,
    678     verbose=self.verbose,
    679     prefer="threads",
    680 )(
    681     delayed(_local_parallel_build_trees)(
    682         s,
    683         t,
    684         self.bootstrap,
    685         X,
    686         y_encoded,
    687         sample_weight,
    688         i,
    689         len(trees),
    690         verbose=self.verbose,
    691         class_weight=self.class_weight,
    692         n_samples_bootstrap=n_samples_bootstrap,
    693         forest=self,
    694     )
```

```
    695     for i, (s, t) in enumerate(zip(samplers, trees))
    696 )
    697 samplers, trees = zip(*samplers_trees)
    699 # Collect newly grown trees

File ~\anaconda3\Lib\site-packages\sklearn\utils\parallel.py:63, in Parallel.__c
all__(self, iterable)
     58 config = get_config()
     59 iterable_with_config = (
     60     (_with_config(delayed_func, config), args, kwargs)
     61     for delayed_func, args, kwargs in iterable
     62 )
---> 63 return super().__call__(iterable_with_config)

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:1088, in Parallel.__call__
(self, iterable)
   1085 if self.dispatch_one_batch(iterator):
   1086     self._iterating = self._original_iterator is not None
-> 1088 while self.dispatch_one_batch(iterator):
   1089     pass
   1091 if pre_dispatch == "all" or n_jobs == 1:
   1092     # The iterable was consumed all at once by the above for loop.
   1093     # No need to wait for async callbacks to trigger to
   1094     # consumption.

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:901, in Parallel.dispatch_
one_batch(self, iterator)
    899     return False
    900 else:
--> 901     self._dispatch(tasks)
    902     return True

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:819, in Parallel._dispatch
(self, batch)
    817 with self._lock:
    818     job_idx = len(self._jobs)
--> 819     job = self._backend.apply_async(batch, callback=cb)
    820     # A job can complete so quickly than its callback is
    821     # called before we get here, causing self._jobs to
    822     # grow. To ensure correct results ordering, .insert is
    823     # used (rather than .append) in the following line
    824     self._jobs.insert(job_idx, job)

File ~\anaconda3\Lib\site-packages\joblib\_parallel_backends.py:208, in Sequenti
alBackend.apply_async(self, func, callback)
    206 def apply_async(self, func, callback=None):
    207     """Schedule a func to be run"""
--> 208     result = ImmediateResult(func)
    209     if callback:
    210         callback(result)

File ~\anaconda3\Lib\site-packages\joblib\_parallel_backends.py:597, in Immediat
eResult.__init__(self, batch)
    594 def __init__(self, batch):
    595     # Don't delay the application, to avoid keeping the input
    596     # arguments in memory
--> 597     self.results = batch()

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:288, in BatchedCalls.__cal
l__(self)
    284 def __call__(self):
    285     # Set the default nested backend to self._backend but do not set the
    286     # change the default number of processes to -1
    287     with parallel_backend(self._backend, n_jobs=self._n_jobs):
--> 288         return [func(*args, **kwargs)
    289                 for func, args, kwargs in self.items]

File ~\anaconda3\Lib\site-packages\joblib\parallel.py:288, in <listcomp>(.0)
```

```
File ~\anaconda3\Lib\site-packages\joblib\parallel.py:288, in <listcomp>(.0)
    284 def __call__(self):
    285     # Set the default nested backend to self._backend but do not set the
    286     # change the default number of processes to -1
    287     with parallel_backend(self._backend, n_jobs=self._n_jobs):
--> 288         return [func(*args, **kwargs)
    289                 for func, args, kwargs in self.items]

File ~\anaconda3\Lib\site-packages\sklearn\utils\parallel.py:123, in __call__(se
lf, *args, **kwargs)
    116 config = getattr(self, "config", None)
    117 if config is None:
    118     warnings.warn(
    119         (
    120             "`sklearn.utils.parallel.delayed` should be used with"
    121             " `sklearn.utils.parallel.Parallel` to make it possible to"
    122             " propagate the scikit-learn configuration of the current th
read to"
--> 123             " the joblib workers."
    124         ),
    125         UserWarning,
    126     )
    127     config = {}
    128 with config_context(**config):

File ~\anaconda3\Lib\site-packages\imblearn\ensemble\_forest.py:65, in _local_pa
rallel_build_trees(sampler, tree, bootstrap, X, y, sample_weight, tree_idx, n_tr
ees, verbose, class_weight, n_samples_bootstrap, forest)
     47 MAX_INT = np.iinfo(np.int32).max
     48 sklearn_version = parse_version(sklearn.__version__)
     51 def _local_parallel_build_trees(
     52     sampler,
     53     tree,
     54     bootstrap,
     55     X,
     56     y,
     57     sample_weight,
     58     tree_idx,
     59     n_trees,
     60     verbose=0,
     61     class_weight=None,
     62     n_samples_bootstrap=None,
     63     forest=None,
     64     missing_values_in_feature_mask=None,
---> 65 ):
     66     # resample before to fit the tree
     67     X_resampled, y_resampled = sampler.fit_resample(X, y)
     68     if sample_weight is not None:

File ~\anaconda3\Lib\site-packages\imblearn\base.py:208, in BaseSampler.fit_resa
mple(self, X, y)
    187 """Resample the dataset.
    188 
    189 Parameters
   (...)
    205     The corresponding label of `X_resampled`.
    206 """
    207 self._validate_params()
--> 208 return super().fit_resample(X, y)

File ~\anaconda3\Lib\site-packages\imblearn\base.py:112, in SamplerMixin.fit_res
ample(self, X, y)
    106 X, y, binarize_y = self._check_X_y(X, y)
    108 self.sampling_strategy_ = check_sampling_strategy(
    109     self.sampling_strategy, y, self._sampling_type
    110 )
--> 112 output = self._fit_resample(X, y)
    114 y_ = (
```

```
    115     label_binarize(output[1], classes=np.unique(y)) if binarize_y else o
utput[1]
    116 )
    118 X_, y_ = arrays_transformer.transform(output[0], y_)

File ~\anaconda3\Lib\site-packages\imblearn\under_sampling\_prototype_selection
\_random_under_sampler.py:111, in RandomUnderSampler._fit_resample(self, X, y)
    107 random_state = check_random_state(self.random_state)
    109 idx_under = np.empty((0,), dtype=int)
--> 111 for target_class in np.unique(y):
    112     if target_class in self.sampling_strategy_.keys():
    113         n_samples = self.sampling_strategy_[target_class]

File ~\anaconda3\Lib\site-packages\numpy\lib\arraysetops.py:274, in unique(ar, r
eturn_index, return_inverse, return_counts, axis, equal_nan)
    272 ar = np.asanyarray(ar)
    273 if axis is None:
--> 274     ret = _unique1d(ar, return_index, return_inverse, return_counts,
    275                     equal_nan=equal_nan)
    276     return _unpack_tuple(ret)
    278 # axis was specified and not None

File ~\anaconda3\Lib\site-packages\numpy\lib\arraysetops.py:336, in _unique1d(a
r, return_index, return_inverse, return_counts, equal_nan)
    334     aux = ar[perm]
    335 else:
--> 336     ar.sort()
    337     aux = ar
    338 mask = np.empty(aux.shape, dtype=np.bool_)

KeyboardInterrupt:
```

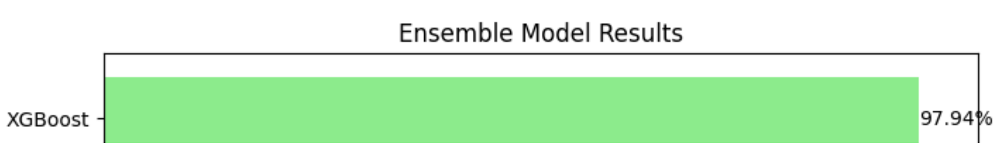
In [57]:

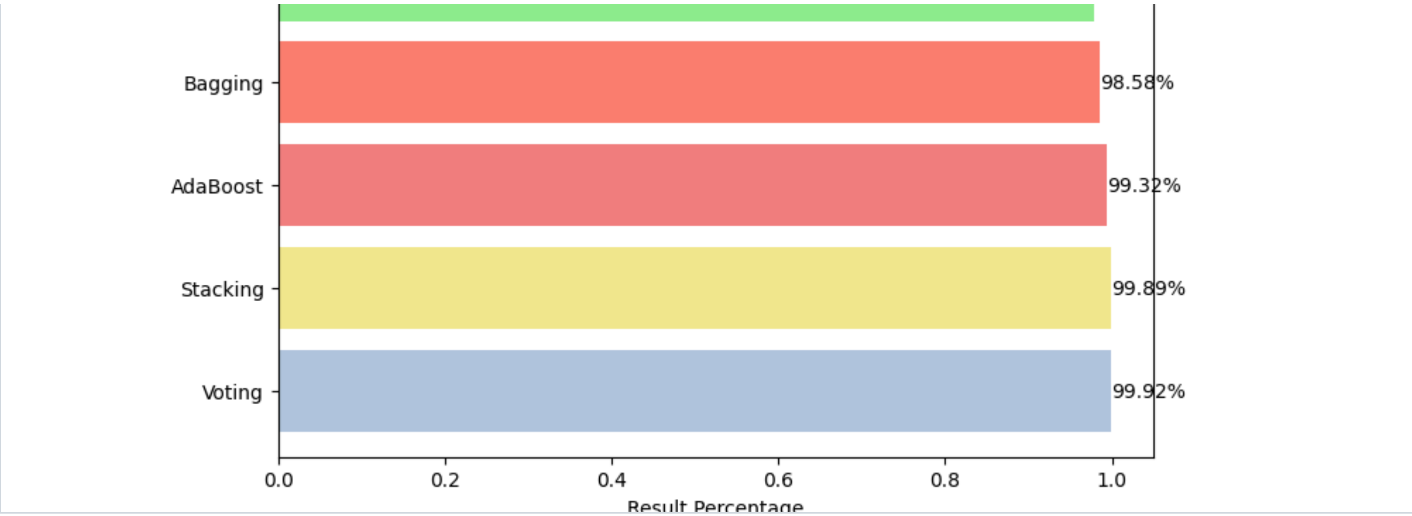
```
print("Recall on subset:", recall_score(y_test_subset, y_pred_subset, pos_labe
```

```
Recall on subset: 0.9989365473236441
```

In [69]:

```
import matplotlib.pyplot as plt

# Sample data
models = ['Voting', 'Stacking', 'AdaBoost', 'Bagging', 'XGBoost']
results = [.9992, .9989, .9932, .9858, .9794]

# Sort the results and models in ascending order
sorted_results, sorted_models = zip(*sorted(zip(results, models)))

# Create a multicolor bar graph
plt.figure(figsize=(8, 5))
bars = plt.barh(sorted_models, sorted_results, color=['lightgreen', 'salmon',
plt.xlabel('Result Percentage')
plt.title('Ensemble Model Results')
plt.gca().invert_yaxis()  # Invert y-axis to show highest percentage at the to

# Add percentage values on bars
for bar, result in zip(bars, sorted_results):
    plt.text(bar.get_width(), bar.get_y() + bar.get_height()/2, f'{result:.2%}


plt.show()
```




Bagging
98.58%
AdaBoost
99.32%
Stacking
99.89%
Voting
99.92%
0.0
0.2
0.4
0.6
0.8
1.0
Result Percentage