In [ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from imblearn.over_sampling import SMOTE
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report
from sklearn.ensemble import HistGradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
import tensorflow as tf
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization, LeakyReLU, LSTM
from tensorflow.keras.models import Sequential
from tensorflow.keras.regularizers import l2, l2
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam
from statsmodels.graphics.tsaplots import plot_acf
```

Source:

https://www.kaggle.com/datasets/datasetengineer/logistics-vehicle-maintenance-history-dataset

Data Dictionary:

- Features:
- Vehicle_ID: Unique identifier for each vehicle.
- Make_and_Model: The manufacturer and model of the vehicle.
- Year_of_Manufacture: The year the vehicle was manufactured.
- Vehicle_Type: Type of vehicle (e.g., Truck, Van).
- Usage_Hours: Total hours the vehicle has been in operation.
- Route_Info: Description of the type of routes the vehicle typically takes (e.g., Rural, Urban).
- Load_Capacity: The maximum load the vehicle can carry.
- Actual_Load: The actual load carried during operations.
- Last_Maintenance_Date: The date of the vehicle's last maintenance activity.
- Maintenance_Type: The type of maintenance performed (e.g., Oil Change, Tire Rotation).
- Maintenance_Cost: The cost associated with the last maintenance performed.
- Engine_Temperature: Temperature of the engine during operation.
- Tire_Pressure: Pressure of the tires in PSI.
- Fuel_Consumption: Fuel consumption in gallons.
- Battery_Status: Current condition of the vehicle's battery.

- Vibration_Levels: Measured vibration levels of the vehicle.
- Oil_Quality: Quality of the engine oil, rated as Good, Fair, or Poor.
- Brake_Condition: Condition of the vehicle's brakes.
- Failure_History: Indicates whether the vehicle has a history of failures (1 = Yes, 0 = No).
- Anomalies_Detected: Number of anomalies detected during monitoring.
- Predictive_Score: A score indicating the likelihood of maintenance needs based on predictive analytics.
- Maintenance_Required: Indicates if maintenance is required (1 = Yes, 0 = No).
- Weather_Conditions: Weather conditions during the vehicle's operation (e.g., Clear, Rainy).
- Road_Conditions: Type of road conditions experienced (e.g., Highway, Urban).
- Delivery_Times: Average delivery times for the vehicle.
- Downtime_Maintenance: Time spent on maintenance activities.
- Impact_on_Efficiency: Metric indicating how maintenance activities affect operational efficiency.
- Use Cases:
  - This dataset is designed for various applications, including: Developing predictive maintenance models using machine learning and deep learning techniques. Analyzing vehicle performance under different environmental conditions. Enhancing fleet management strategies through IoT data integration.

EDA

In [2]:
```
df = pd.read_csv('logistics_dataset_with_maintenance_required.csv')
df.tail()
```

Out[2]:

	Vehicle_ID	Make_and_Model	Year_of_Manufacture	Vehicle_Type	Usage_Hours	Route_Info	Load_Capacity	Actual_Load	Last_Maintenance_Date	Maintenance_Type	...	Brake_Condition	Failure_History	Anomalies_Detected	Predic
91995	91996	Chevy Silverado	2022	Van	293	Urban	12.446365	14.460276	2023-02-14	Oil Change	...	Fair	0	1	
91996	91997	Ford F-150	2006	Truck	1445	Highway	82.281140	78.013688	2023-02-16	Tire Rotation	...	Good	0	0	
91997	91998	Tesla Semi	2020	Van	831	Rural	27.510624	21.631656	2023-04-18	Engine Overhaul	...	Poor	0	1	
91998	91999	Tesla Semi	2022	Truck	1326	Highway	4.439415	4.511761	2024-05-14	Tire Rotation	...	Poor	0	0	
91999	92000	Ford F-150	2020	Van	4334	Urban	4.103816	4.170903	2023-05-04	Tire Rotation	...	Good	0	0	

5 rows × 27 columns

In [3]:
```
df.nunique()
```

Out[3]:

	0
Vehicle_ID	92000
Make_and_Model	4
Year_of_Manufacture	18
Vehicle_Type	2
Usage_Hours	11975
Route_Info	3
Load_Capacity	92000
Actual_Load	92000
Last_Maintenance_Date	547
Maintenance_Type	3
Maintenance_Cost	92000
Engine_Temperature	1
Tire_Pressure	33079
Fuel_Consumption	43004
Battery_Status	4993
Vibration_Levels	92000
Oil_Quality	89907
Brake_Condition	3
Failure_History	2
Anomalies_Detected	2
Predictive_Score	92000
Maintenance_Required	2
Weather_Conditions	4
Road_Conditions	3
Delivery_Times	63595
Downtime_Maintenance	36761
Impact_on_Efficiency	71225

dtype: int64

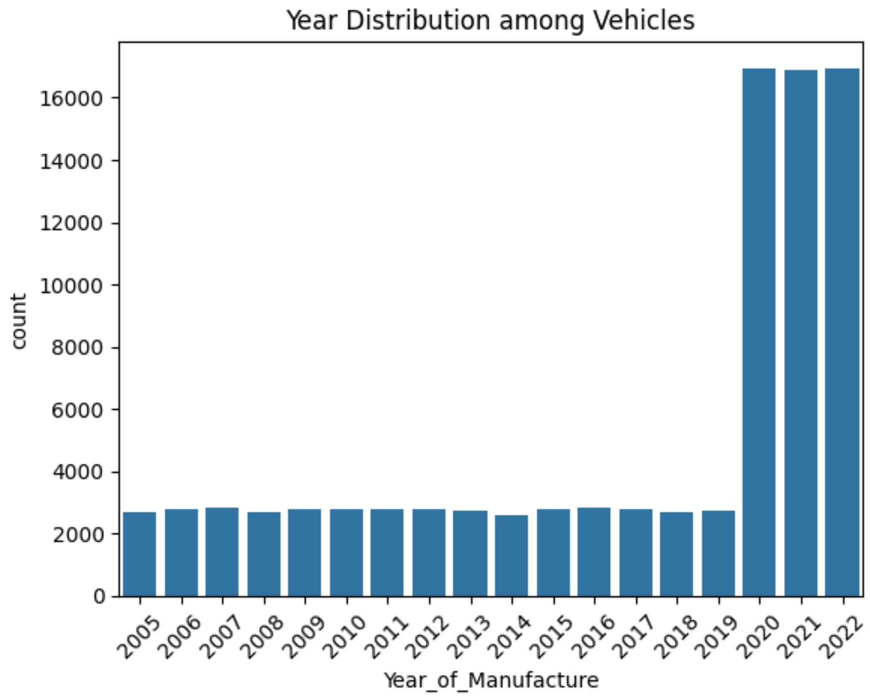
In [4]:

```
df.drop(columns = 'Vehicle_ID', axis = 1, inplace = True)
```

In [5]:
```
df.info()
```

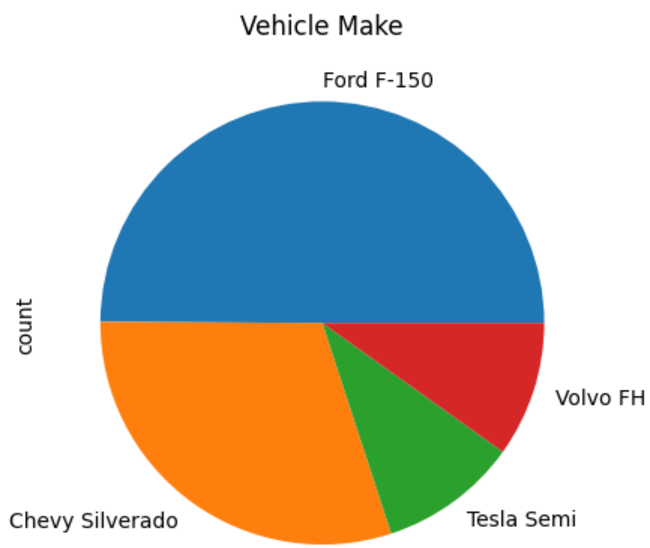
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 92000 entries, 0 to 91999
Data columns (total 26 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   Make_and_Model         92000 non-null  object 
 1   Year_of_Manufacture    92000 non-null  int64  
 2   Vehicle_Type           92000 non-null  object 
 3   Usage_Hours            92000 non-null  int64  
 4   Route_Info             92000 non-null  object 
 5   Load_Capacity          92000 non-null  float64
 6   Actual_Load            92000 non-null  float64
 7   Last_Maintenance_Date  92000 non-null  object 
 8   Maintenance_Type       92000 non-null  object 
 9   Maintenance_Cost       92000 non-null  float64
 10  Engine_Temperature     92000 non-null  float64
 11  Tire_Pressure          92000 non-null  float64
 12  Fuel_Consumption       92000 non-null  float64
 13  Battery_Status         92000 non-null  float64
 14  Vibration_Levels       92000 non-null  float64
 15  Oil_Quality            92000 non-null  float64
 16  Brake_Condition        92000 non-null  object 
 17  Failure_History        92000 non-null  int64  
 18  Anomalies_Detected     92000 non-null  int64  
 19  Predictive_Score       92000 non-null  float64
 20  Maintenance_Required   92000 non-null  int64  
 21  Weather_Conditions     92000 non-null  object 
 22  Road_Conditions        92000 non-null  object 
 23  Delivery_Times         92000 non-null  float64
 24  Downtime_Maintenance   92000 non-null  float64
 25  Impact_on_Efficiency   92000 non-null  float64
dtypes: float64(13), int64(5), object(8)
memory usage: 18.2+ MB
```

In [6]:
```
sns.countplot(data= df, x = 'Year_of_Manufacture')
plt.title('Year Distribution among Vehicles')
plt.xticks(rotation = 45)
plt.show()
```



In [7]:
```
df['Make_and_Model'].value_counts().plot(kind='pie')
plt.title('Vehicle Make')
```

Out[7]:
```
Text(0.5, 1.0, 'Vehicle Make')
```



In [8]:
```
df.groupby('Make_and_Model')['Year_of_Manufacture'].mean()
```

Out[8]:

Make_and_Model	Year_of_Manufacture
Chevy Silverado	2016.929703
Ford F-150	2017.001960
Tesla Semi	2016.900474
Volvo FH	2016.987016

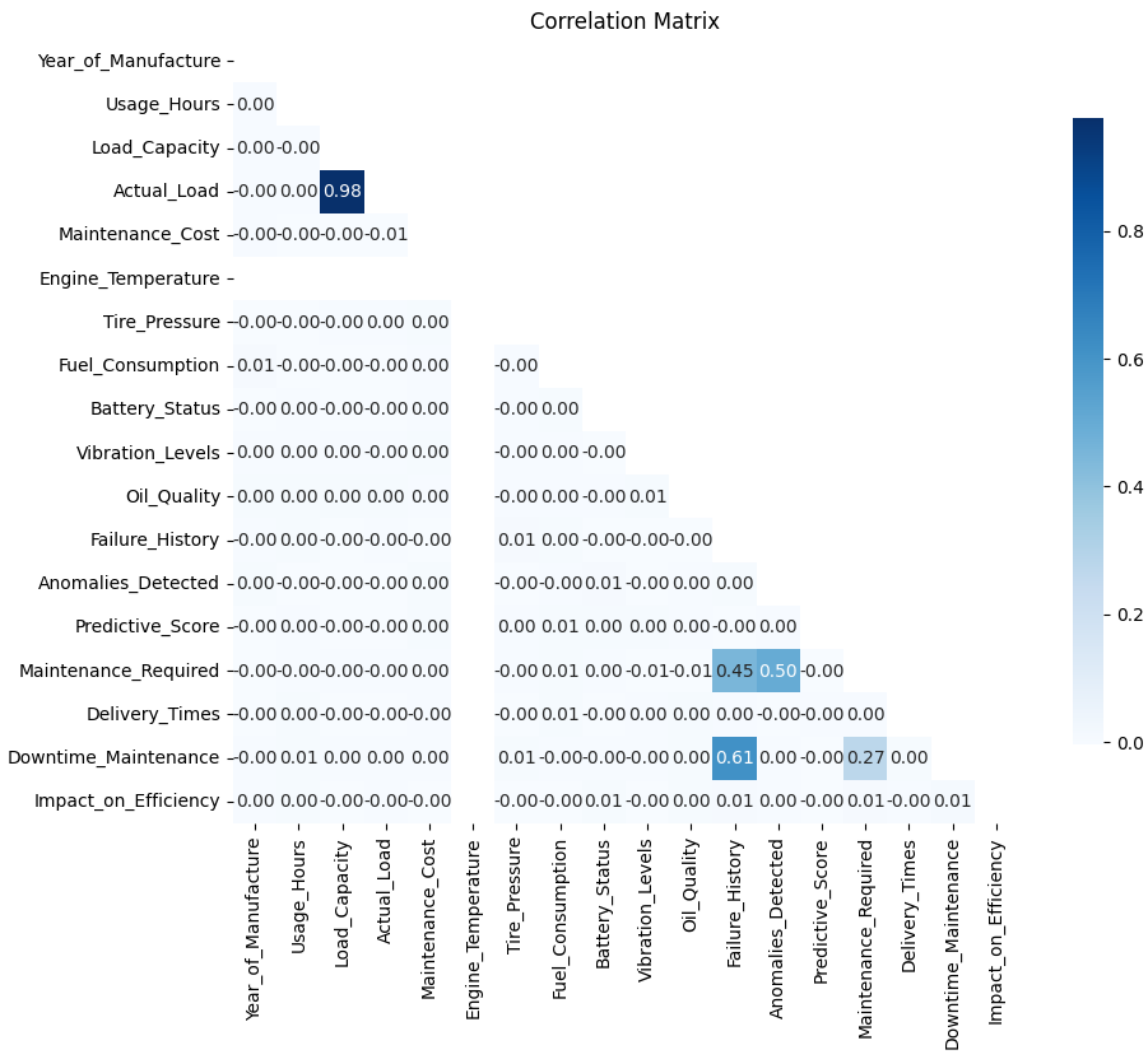
dtype: float64

In [9]:
```
df['Last_Maintenance_Date'] = pd.to_datetime(df['Last_Maintenance_Date'])
df['Last_Maintenance_Week'] = df['Last_Maintenance_Date'].dt.weekday
```

Correlation Matrix

In [10]:
```
num = []
for i in df.columns:
    if (df[i].dtype == 'float64' or df[i].dtype == 'int64'):
        num.append(i)
numerical =  df[num]

corr = numerical.corr()
mask = np.triu(np.ones_like(corr, dtype=bool))
```

```
plt.figure(figsize=(11, 8))
sns.heatmap(corr,mask=mask, cmap='Blues', annot=True, fmt=".2f", square=True, cbar_kws={"shrink": 0.8})
plt.title('Correlation Matrix')
plt.show()
```



Observation: There is a high correlation between failure history, anomalies detected and maintenance required as expected. Therefore it is important to address the correlation between these features to avoid redunday

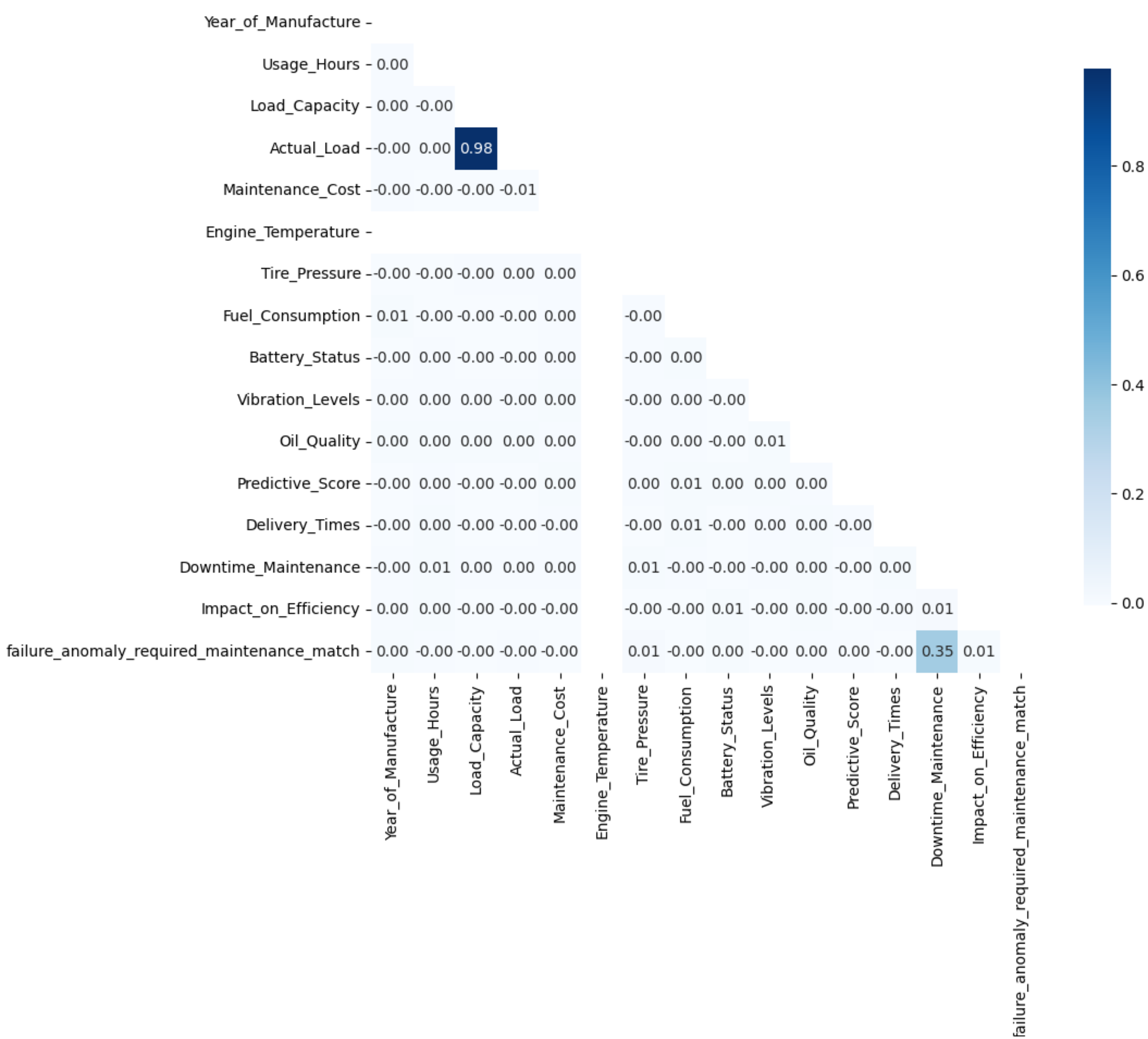
In [11]:
```
# Matching features with' failure history' and 'detected anomalies' into a single feature
df['failure_anomaly_required_maintenance_match'] = ((df['Failure_History'] == 1) & (df['Anomalies_Detected'] == 1) & (df['Maintenance_Required'])).astype('int64')
df.drop(columns=['Failure_History','Anomalies_Detected','Maintenance_Required'], axis= 1, inplace= True)
df.head()
```

Out[11]:

	Make_and_Model	Year_of_Manufacture	Vehicle_Type	Usage_Hours	Route_Info	Load_Capacity	Actual_Load	Last_Maintenance_Date	Maintenance_Type	Maintenance_Cost	...	Oil_Quality	Brake_Condition	Predictive_Score	Weather_Co
0	Ford F-150	2022	Truck	530	Rural	7.534549	9.004247	2023-04-09	Oil Change	110.165442	...	80.393803	Good	0.171873	
1	Volvo FH	2015	Van	10679	Rural	7.671728	6.111785	2023-07-20	Tire Rotation	265.898087	...	91.302461	Fair	0.246670	
2	Chevy Silverado	2022	Van	4181	Rural	2.901159	3.006055	2023-03-17	Oil Change	412.483470	...	70.109021	Good	0.455236	
3	Chevy Silverado	2011	Truck	2974	Urban	15.893347	18.825290	2024-05-01	Tire Rotation	444.110857	...	74.932225	Good	0.060208	
4	Ford F-150	2014	Van	2539	Rural	60.668320	65.605463	2023-11-15	Tire Rotation	478.841922	...	86.357250	Good	0.264929	

5 rows × 25 columns

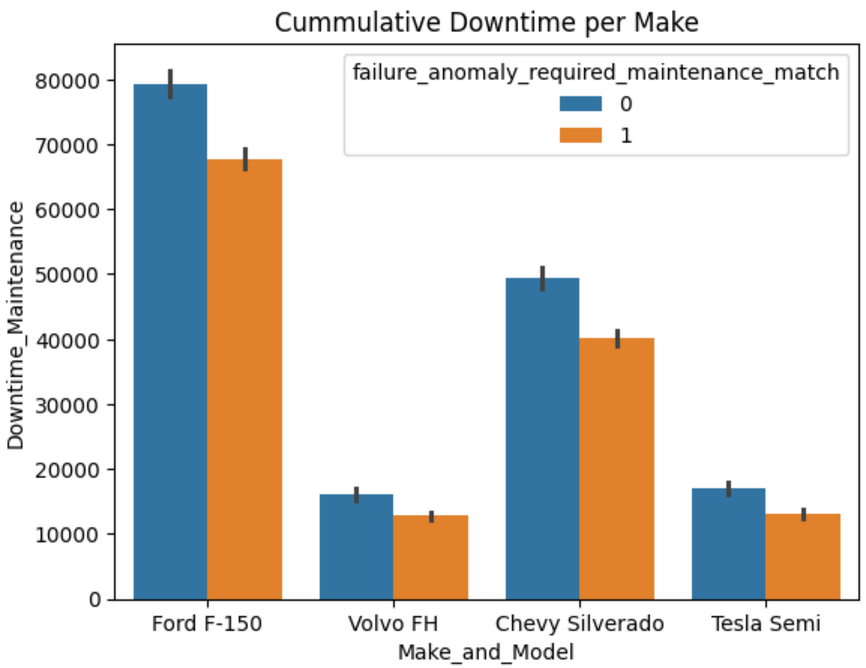
In [ ]:
```
num = []
for i in df.columns:
    if (df[i].dtype == 'float64' or df[i].dtype == 'int64'):
        num.append(i)
numerical =  df[num]

corr = numerical.corr()
mask = np.triu(np.ones_like(corr, dtype=bool))
plt.figure(figsize=(10, 8))
sns.heatmap(corr, mask=mask, cmap='Blues', annot=True, fmt=".2f", square=True, cbar_kws={"shrink": 0.8})
plt.show()
```



Year_of_Manufacture
Usage_Hours
Load_Capacity
Actual_Load
Maintenance_Cost
Engine_Temperature
Tire_Pressure
Fuel_Consumption
Battery_Status
Vibration_Levels
Oil_Quality
Predictive_Score
Delivery_Times
Downtime_Maintenance
Impact_on_Efficiency
failure_anomaly_required_maintenance_match
0.98
0.35
0.0
0.2
0.4
0.6
0.8

Downtime

In [13]:
```python
sns.barplot(data = df, x = 'Make_and_Model', y = 'Downtime_Maintenance', estimator= 'sum', hue= 'failure_anomaly_required_maintenance_match')
plt.title('Cummulative Downtime per Make')
plt.show()
```



Time Series

In [14]:
```python
df['Last_Maintenance_Date']
```

Out[14]:

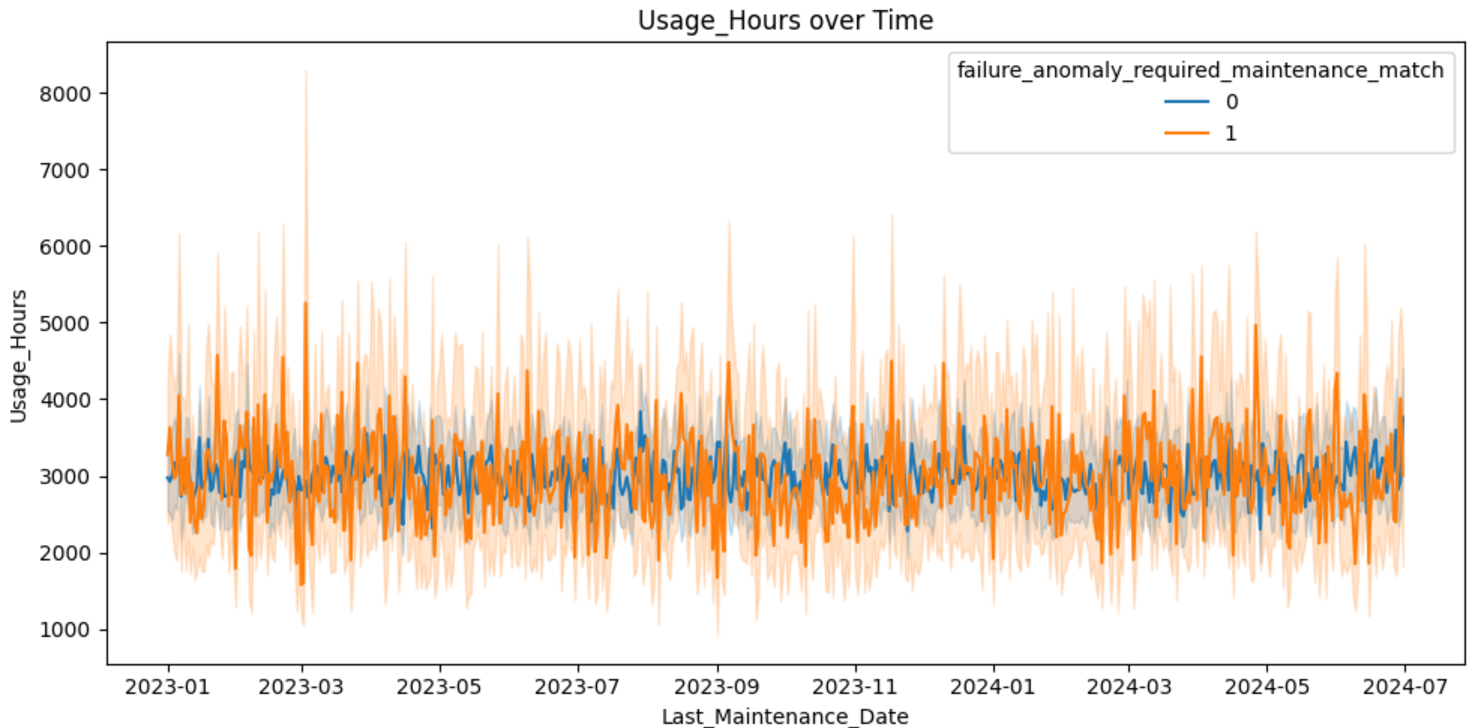
	Last_Maintenance_Date
0	2023-04-09
1	2023-07-20
2	2023-03-17
3	2024-05-01
4	2023-11-15
...	...
91995	2023-02-14
91996	2023-02-16
91997	2023-04-18
91998	2024-05-14
91999	2023-05-04

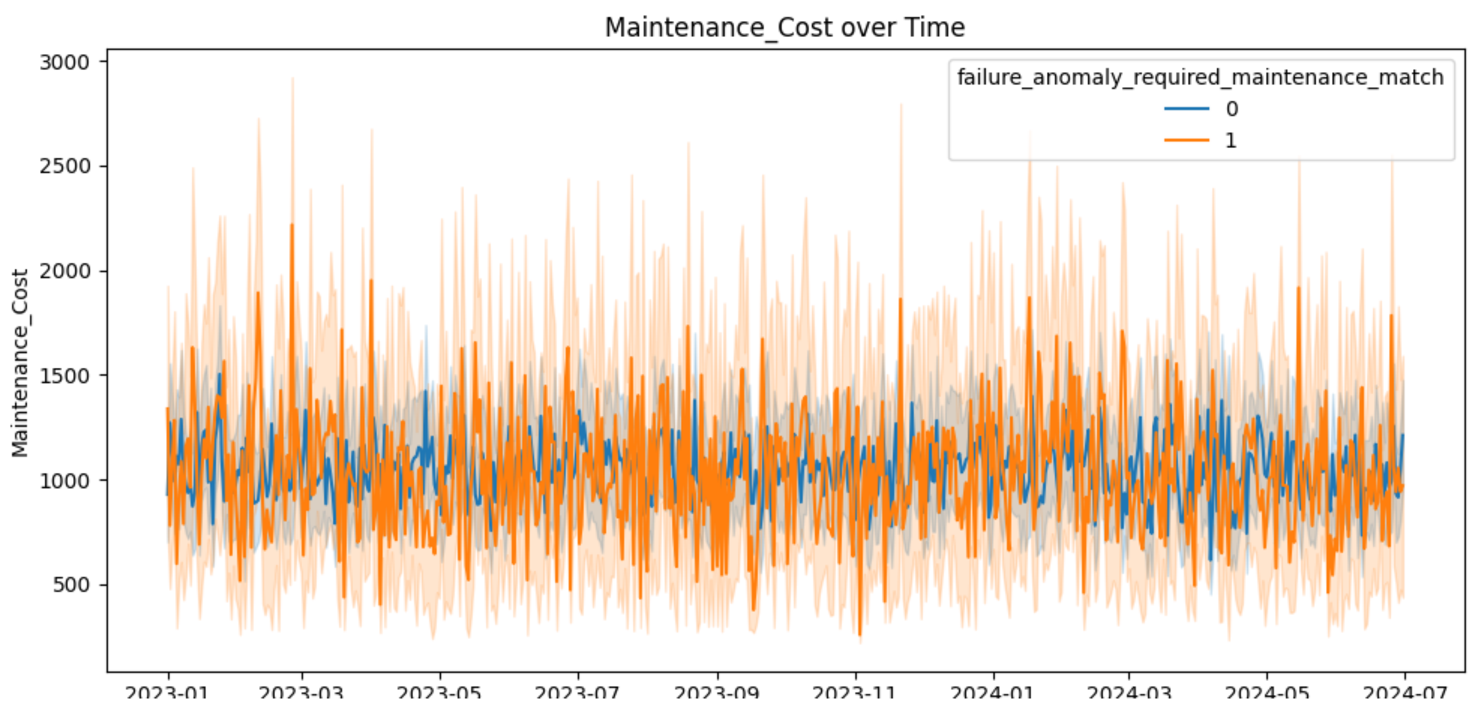
92000 rows × 1 columns

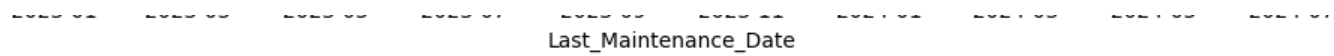
dtype: datetime64[ns]

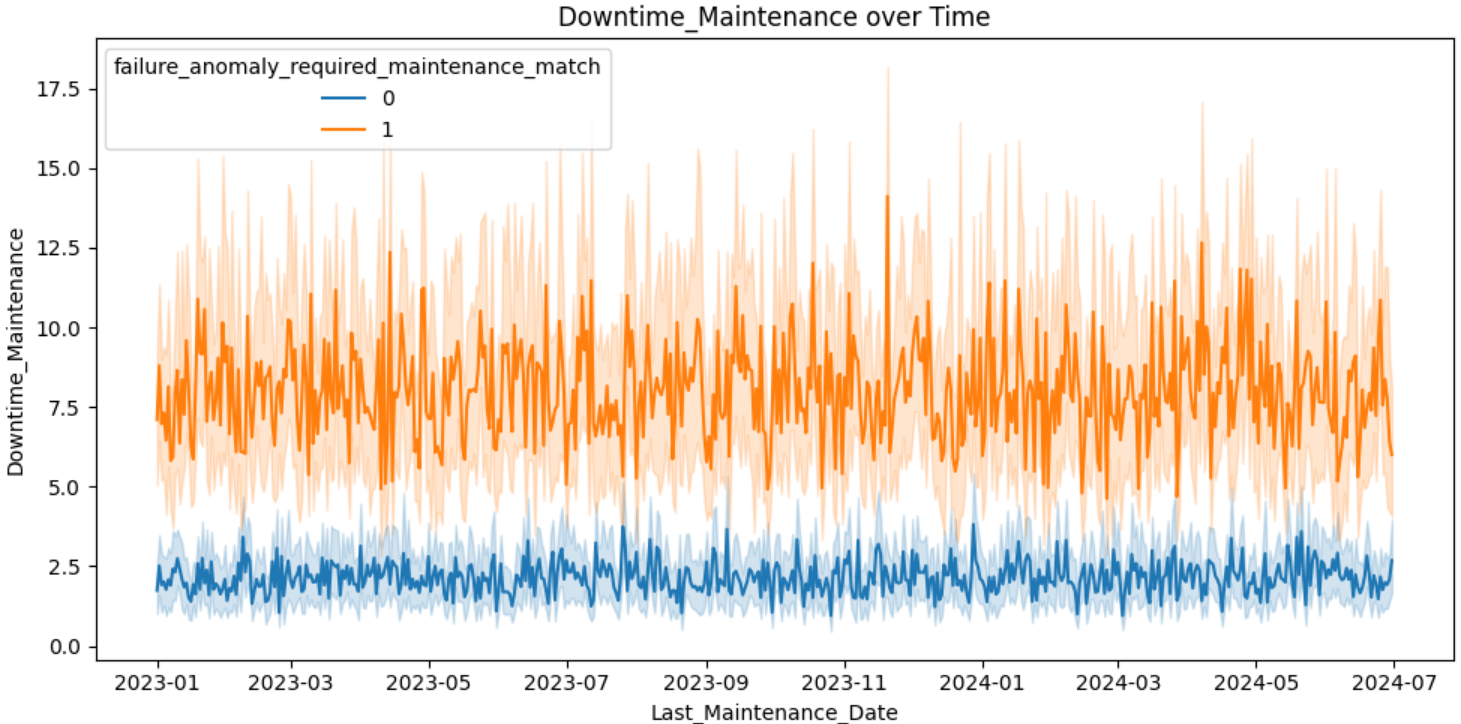
In [15]:

```
columns = ['Usage_Hours', 'Maintenance_Cost', 'Downtime_Maintenance']

fig, axes = plt.subplots(len(columns), 1, figsize=(10, 15))
for ax, col in zip(axes, columns):
    sns.lineplot(data=df, x='Last_Maintenance_Date', y=col, hue='failure_anomaly_required_maintenance_match', ax=ax, estimator= 'mean')
    ax.set_title(f'{col} over Time')
plt.tight_layout()
plt.show()
```


Usage_Hours over Time
failure_anomaly_required_maintenance_match
0
1
8000
7000
6000
5000
4000
3000
2000
1000
Usage_Hours
2023-01
2023-03
2023-05
2023-07
2023-09
2023-11
2024-01
2024-03
2024-05
2024-07
Last_Maintenance_Date

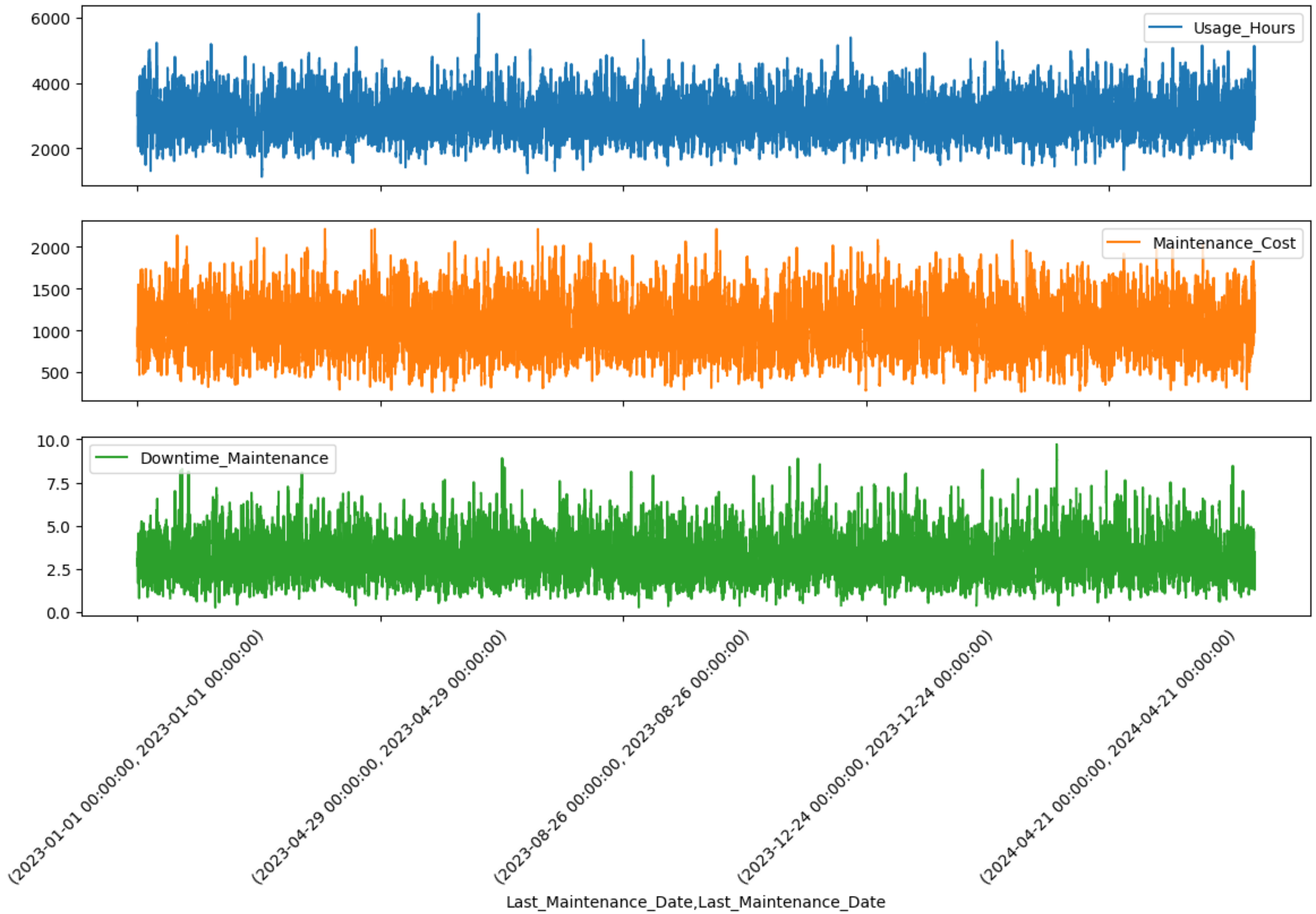

Maintenance_Cost over Time
failure_anomaly_required_maintenance_match
0
1
3000
2500
2000
1500
1000
500
Maintenance_Cost
2023-01
2023-03
2023-05
2023-07
2023-09
2023-11
2024-01
2024-03
2024-05
2024-07





In [16]:
```
df.set_index('Last_Maintenance_Date', inplace=True)
```

In [17]:
```
df.groupby('Last_Maintenance_Date')[['Usage_Hours', 'Maintenance_Cost', 'Downtime_Maintenance']].rolling(30).mean().plot(kind='line', subplots= True, figsize=(14,7))

plt.xticks(rotation = 45)
plt.show();
```

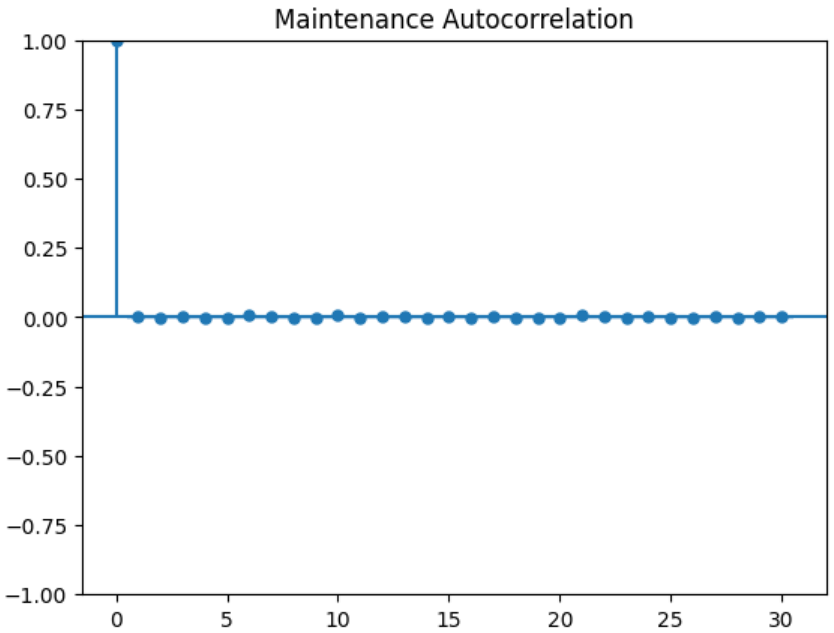


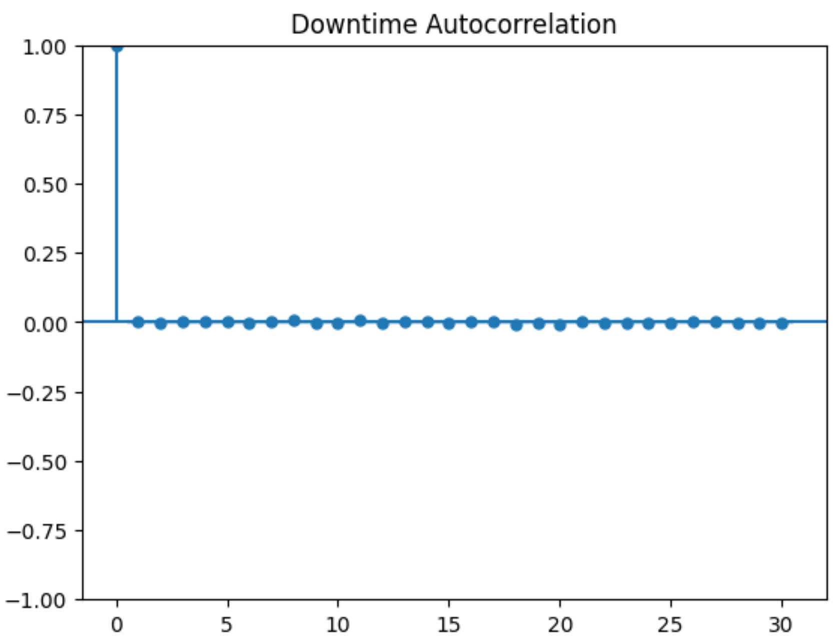
Autocorrelation

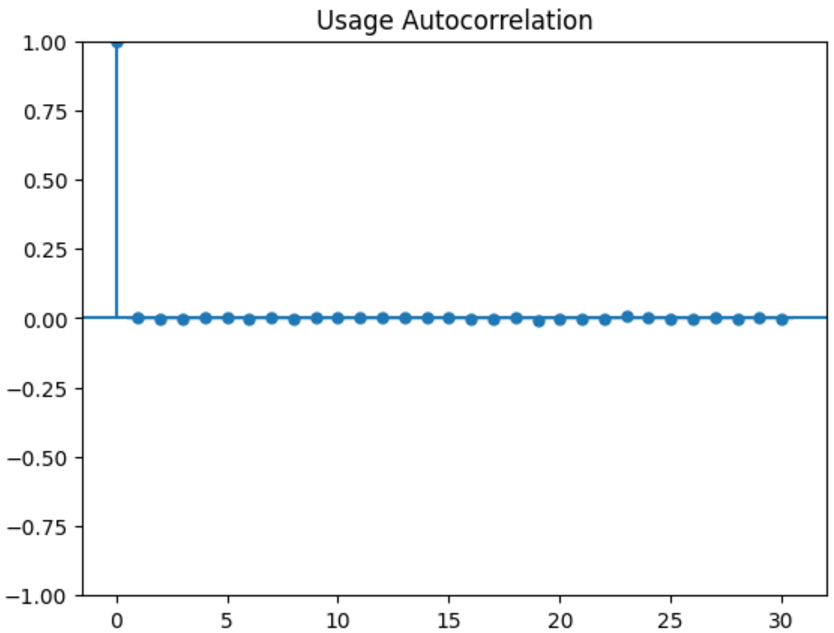
In [ ]:

```
plot_acf(df['Maintenance_Cost'], lags=30)
plt.title('Maintenance Autocorrelation')
plt.show();
plot_acf(df['Downtime_Maintenance'], lags=30)
plt.title('Downtime Autocorrelation')
```

```
plt.show()
plot_acf(df['Usage_Hours'], lags=30)
plt.title('Usage Autocorrelation')
plt.show()
```







Observation: Based on the lack of autocorrelation after the first lag, with all subsequent points close to zero, we can conclude that there is no significant temporal relationship in the features selected. The absence of autocorrelation may stem from the structure of the data, where each observation is unique, making it difficult to identify consistent patterns or trends for prediction. This lack of regularity limits the potential for models to leverage temporal dependencies effectively. This leads us to modify future time-series predictions. In this case we will use LSTM to predict the number of maintenance units we might expect during a certain time period

Capacity

Observation: We will take the capcity columns and we will combine them into a single categorical feature.

In [20]:
```
df.reset_index('Last_Maintenance_Date',inplace= True)
```

In [21]:
```
df['over_capacity/under_capacity'] = df['Load_Capacity'] - df['Actual_Load']
df.drop(columns=['Load_Capacity', 'Actual_Load'], axis = 1, inplace = True)
```

In [22]:
```
df['over_capacity/under_capacity'] = df['over_capacity/under_capacity'].apply(lambda x: 'under_capacity' if x < 0 else ('over_capacity' if x > 0 else 'at_capacity'))
```

In [23]:
```
df.groupby('over_capacity/under_capacity')['Downtime_Maintenance'].mean()
```

Out[23]:

over_capacity/under_capacity	Downtime_Maintenance
over_capacity	3.195530
under_capacity	3.232643

dtype: float64

In [24]:
```
df.groupby('over_capacity/under_capacity')['Maintenance_Cost'].mean()
```

Out[24]:

over_capacity/under_capacity	Maintenance_Cost
over_capacity	1051.564421
under_capacity	1030.029468

dtype: float64

In [25]:
```
df.groupby('over_capacity/under_capacity')['failure_anomaly_required_maintenance_match'].value_counts()
```

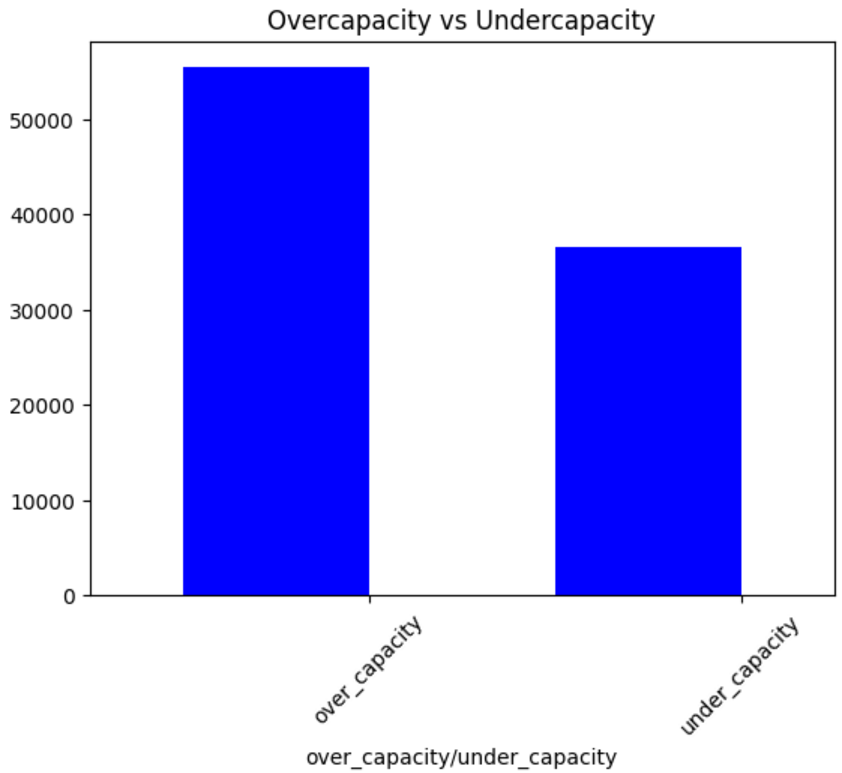
Out[25]:

over_capacity/under_capacity	failure_anomaly_required_maintenance_match	count
over_capacity	0	45422
	1	10010
under_capacity	0	29853
	1	6715

dtype: int64

In [26]:
```
df['over_capacity/under_capacity'].value_counts().plot(kind='bar', color='b', position=1)
plt.title('Overcapacity vs Undercapacity')
plt.xticks(rotation = 45)
```

Out[26]:
```
(array([0, 1]), [Text(0, 0, 'over_capacity'), Text(1, 0, 'under_capacity')])
```

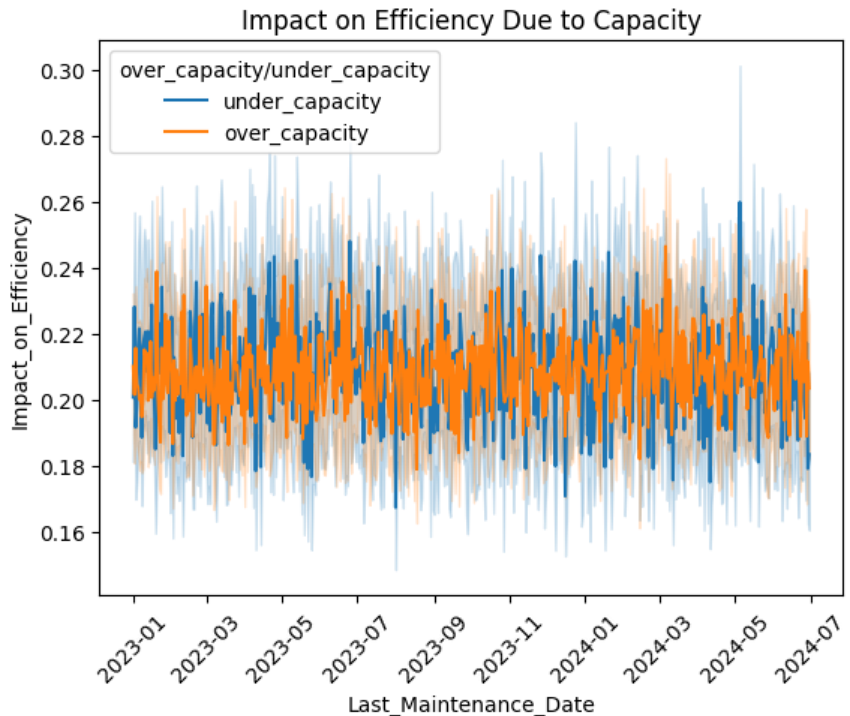


In [27]:
```
over = df[df['over_capacity/under_capacity'] == 'over_capacity']
under = df[df['over_capacity/under_capacity'] == 'under_capacity']

over_tot = round(over['Usage_Hours'].sum() / df['Usage_Hours'].sum() * 100,2)
under_tot = round(under['Usage_Hours'].sum() / df['Usage_Hours'].sum() * 100,2)

print('Percentage of Total Usage Hours of Vehicles:\n')
print('Over-capacity Usage Hours:',over_tot)
print('Under Capacity Usage Hours:',under_tot)
```

```
Percentage of Total Usage Hours of Vehicles:

Over-capacity Usage Hours: 60.18
Under Capacity Usage Hours: 39.82
```

Efficiency

In [28]:
```
sns.lineplot(data = df, x = 'Last_Maintenance_Date', y = 'Impact_on_Efficiency', hue ='over_capacity/under_capacity')
plt.xticks(rotation = 45)
plt.title('Impact on Efficiency Due to Capacity')
plt.show();
```



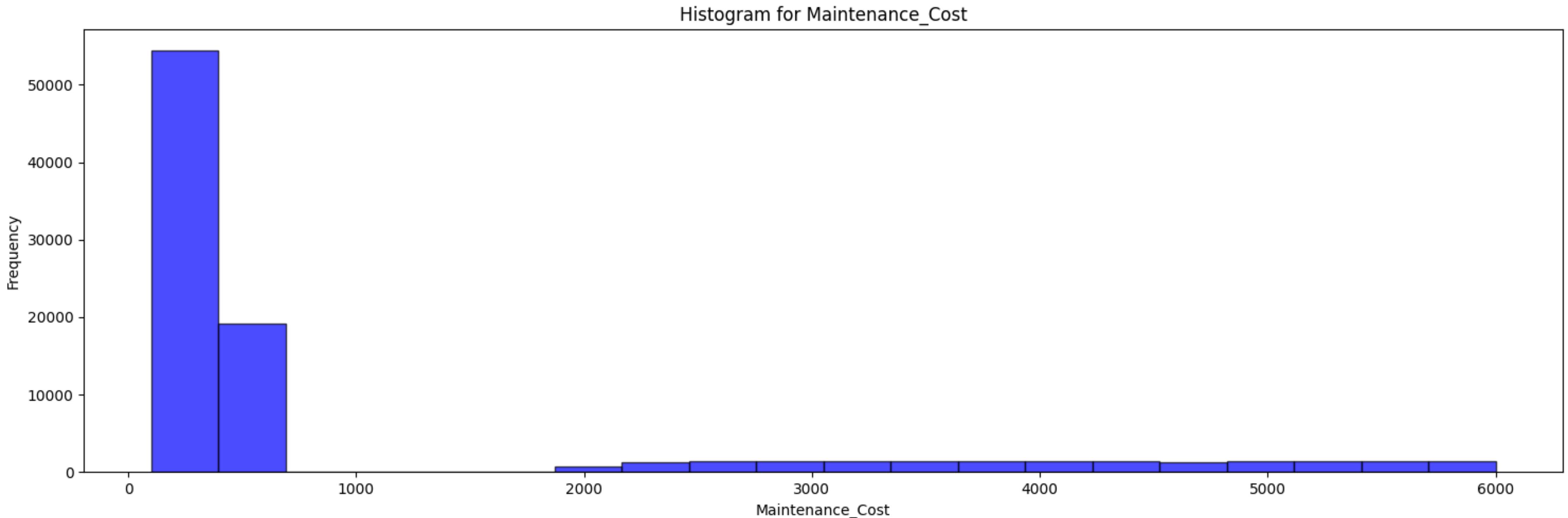
In [29]:
```
df.describe()
```

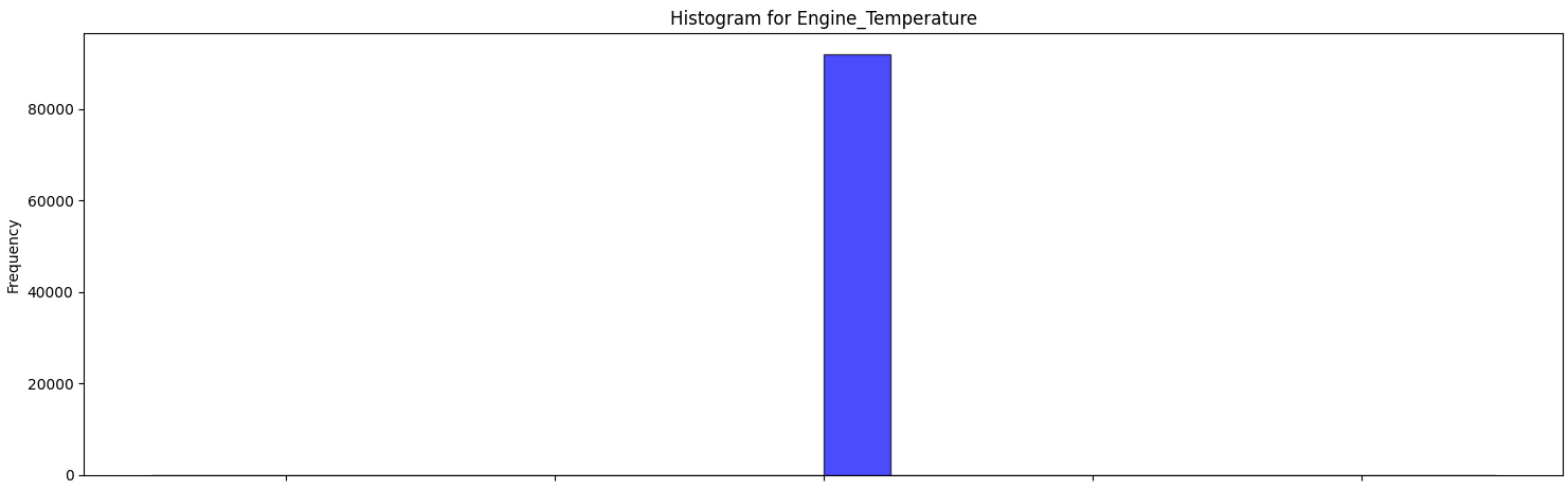
Out[29]:

	Last_Maintenance_Date	Year_of_Manufacture	Usage_Hours	Maintenance_Cost	Engine_Temperature	Tire_Pressure	Fuel_Consumption	Battery_Status	Vibration_Levels	Oil_Quality	Predictive_Score	Delivery_Times	Downtime_Mai
count	92000	92000.000000	92000.000000	92000.000000	92000.0	92000.000000	92000.000000	92000.000000	92000.000000	92000.000000	92000.000000	92000.000000	920(
mean	2023-10-01 01:44:23.999999744	2016.968478	2989.550913	1043.004745	120.0	32.570643	10.657493	45.669862	3.977629	79.930316	0.166754	99.283161	
min	2023-01-01 00:00:00	2005.000000	0.000000	100.002837	120.0	20.000000	5.000000	45.000000	0.000370	38.303330	0.000161	30.000000	
25%	2023-05-17 00:00:00	2013.000000	856.000000	225.213756	120.0	20.000000	5.000000	45.000000	1.135643	73.319542	0.087777	30.000000	
50%	2023-10-01 00:00:00	2020.000000	2070.000000	348.722087	120.0	24.516540	8.349716	45.000000	2.760726	80.013201	0.147868	69.617815	
75%	2024-02-15 00:00:00	2021.000000	4146.000000	474.925612	120.0	48.810813	16.678173	45.000000	5.498541	86.750897	0.227352	139.084008	
max	2024-06-30 00:00:00	2022.000000	36392.000000	5999.905095	120.0	55.000000	20.000000	50.000000	45.475464	100.000000	0.746450	300.000000	(
std	NaN	5.359597	2992.083426	1575.109426	0.0	14.483096	5.979493	1.634766	4.003637	9.794350	0.103435	79.708201	

In [30]:
```
col = ['Maintenance_Cost',      'Engine_Temperature',   'Tire_Pressure','Fuel_Consumption',     'Battery_Status',
                'Vibration_Levels',      'Oil_Quality',  'Predictive_Score',     'Delivery_Times',
                'Downtime_Maintenance', 'Impact_on_Efficiency']

plt.figure(figsize=(15, 5 * len(col)))
```

```python
for i, column in enumerate(col):
    plt.subplot(len(col), 1, i + 1)
    plt.hist(df[column], bins=20, color='blue', alpha=0.7, edgecolor='black')
    plt.title(f'Histogram for {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')

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

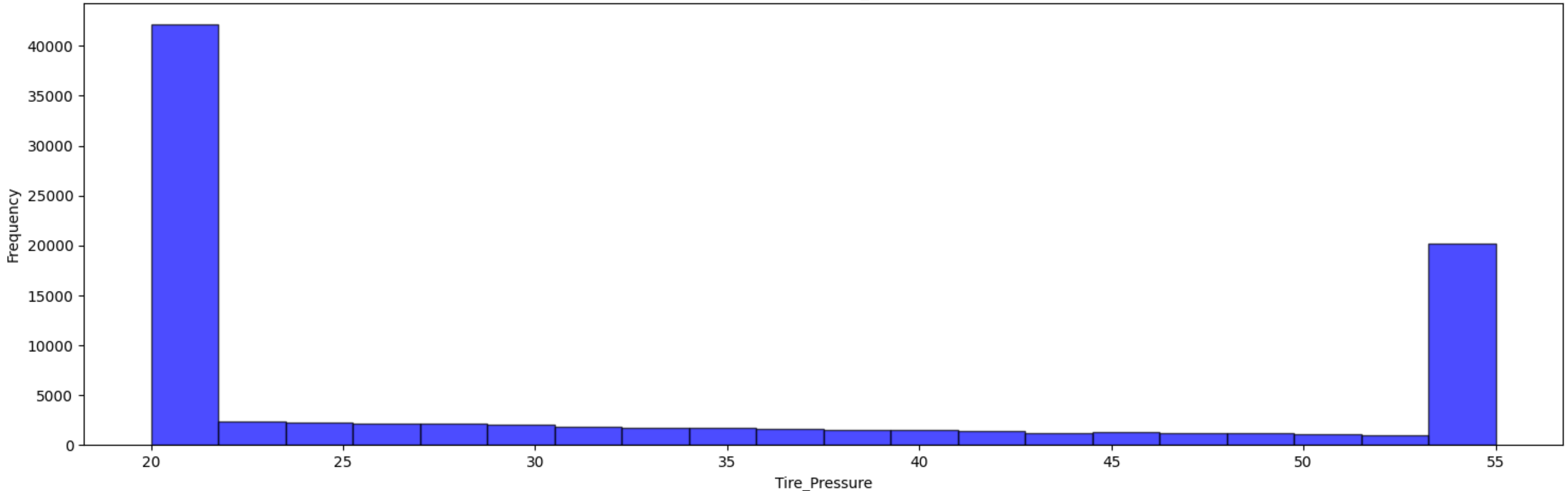

Histogram for Maintenance_Cost
Frequency
0
10000
20000
30000
40000
50000
0
1000
2000
3000
4000
5000
6000
Maintenance_Cost


Histogram for Engine_Temperature
Frequency
0
20000
40000
60000
80000

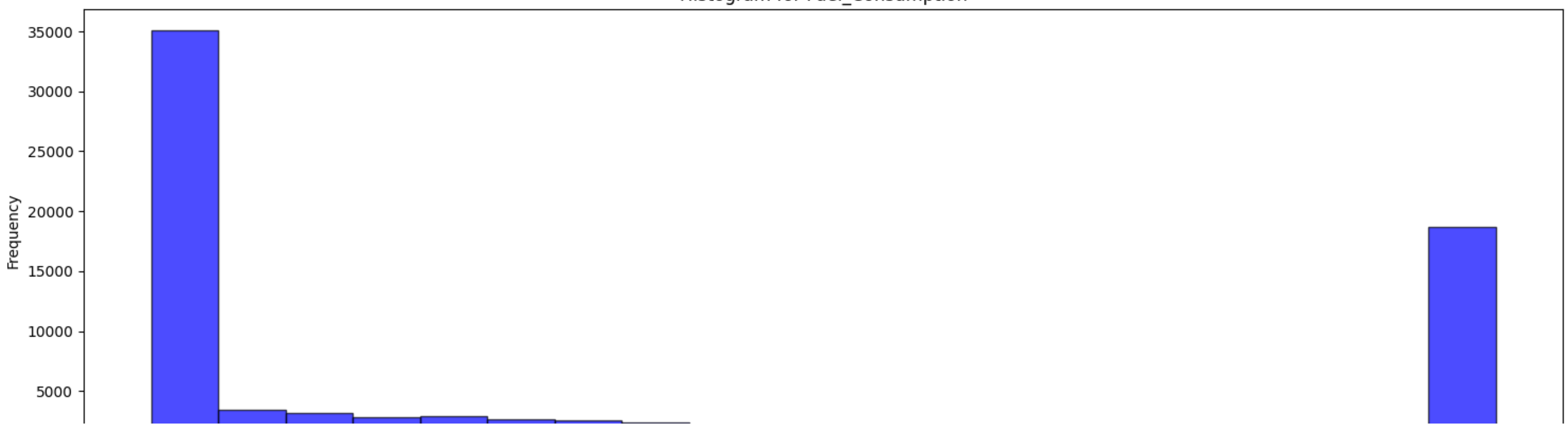
119.6 119.8 120.0 120.2 120.4

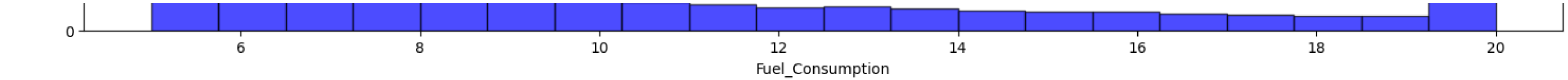
Engine_Temperature

Histogram for Tire_Pressure

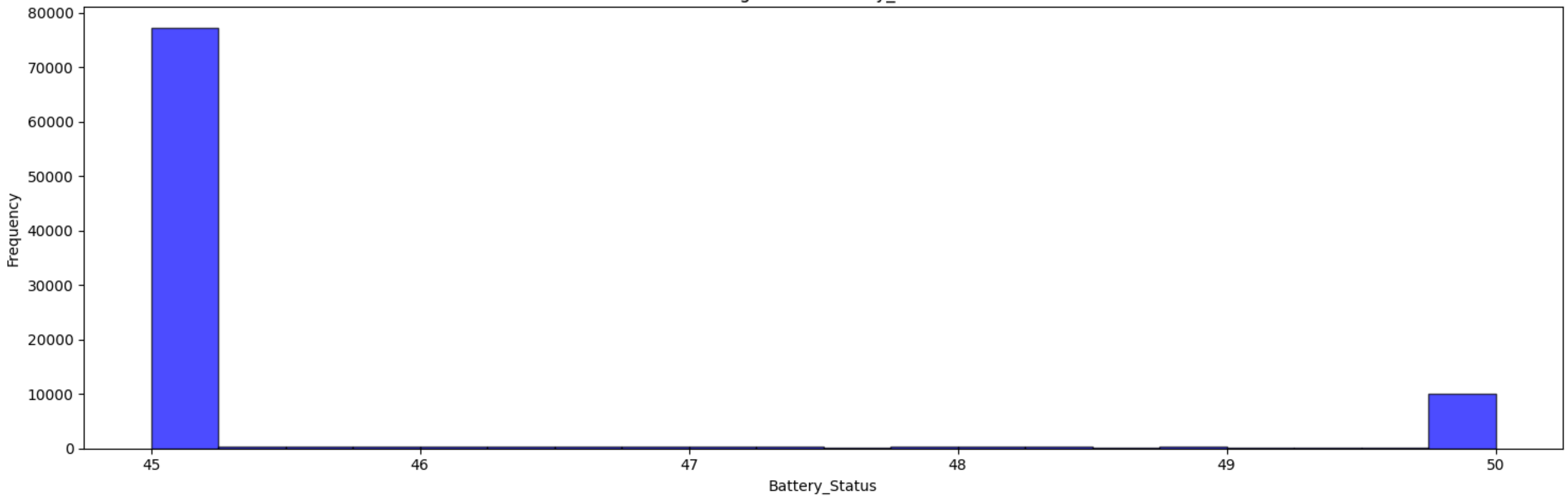


Histogram for Fuel_Consumption

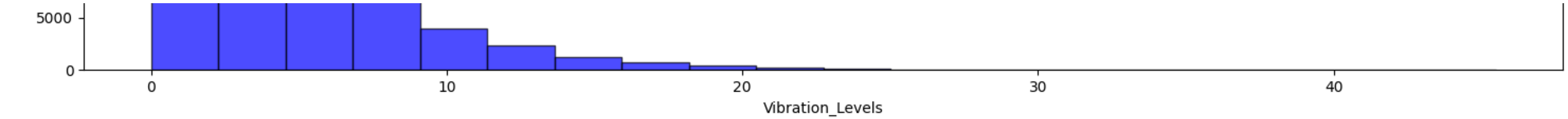




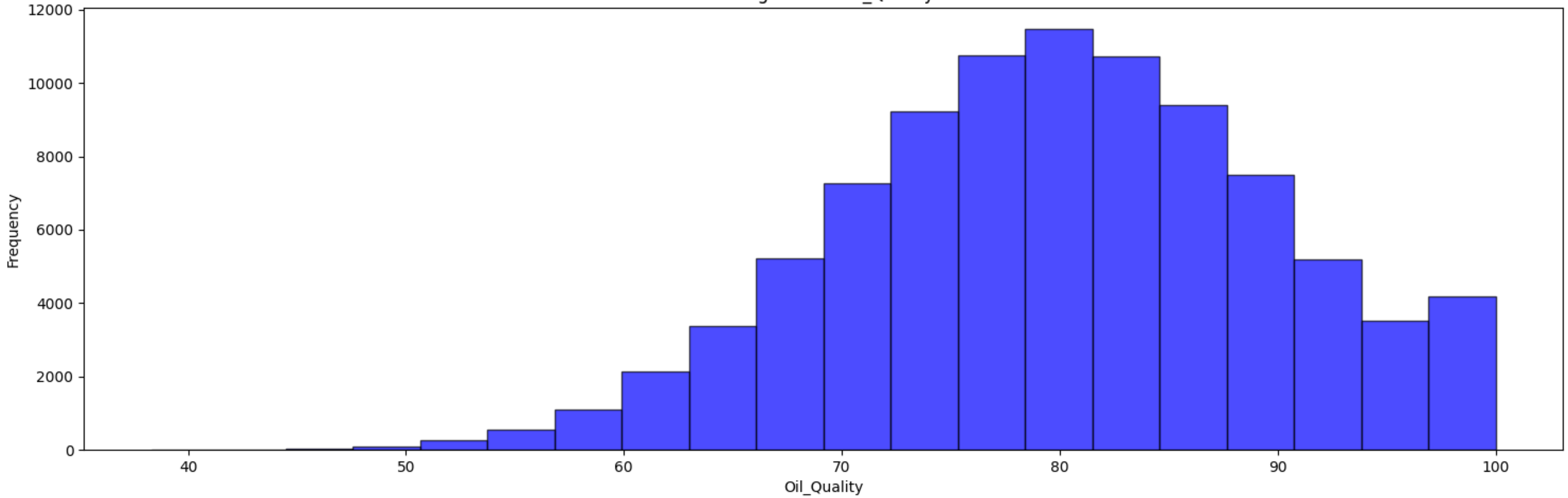
Histogram for Battery_Status



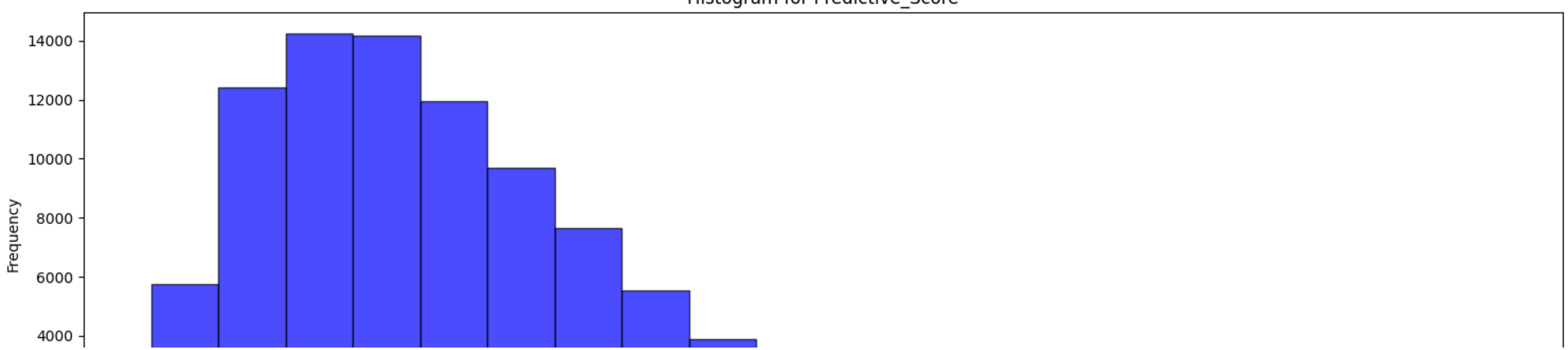
Histogram for Vibration_Levels

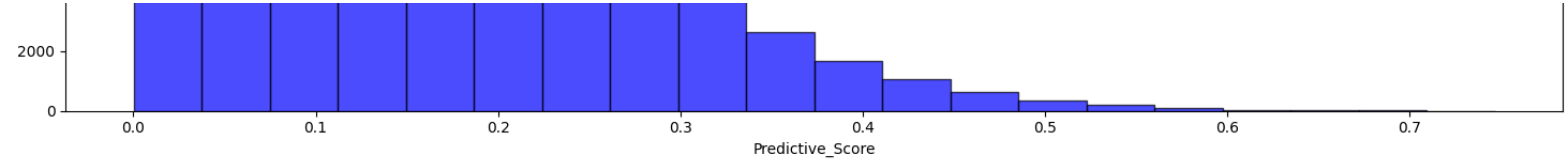


Histogram for Oil_Quality

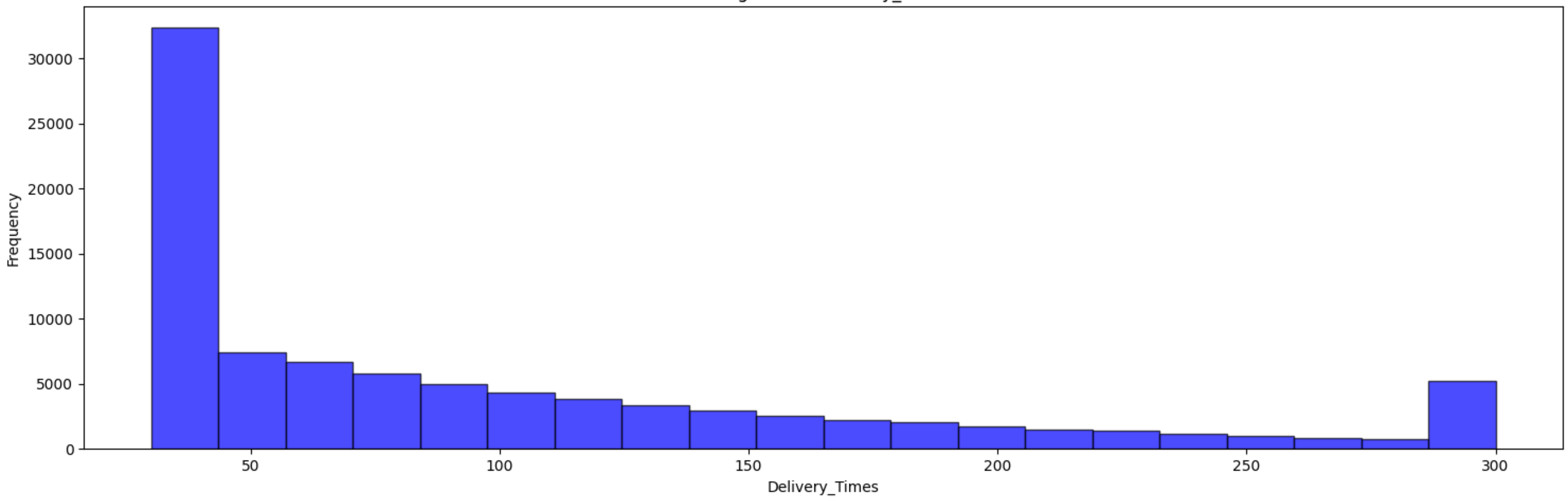


Histogram for Predictive_Score

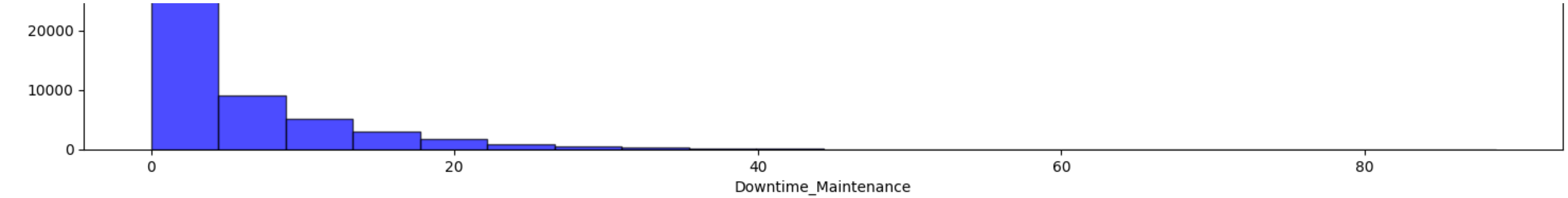


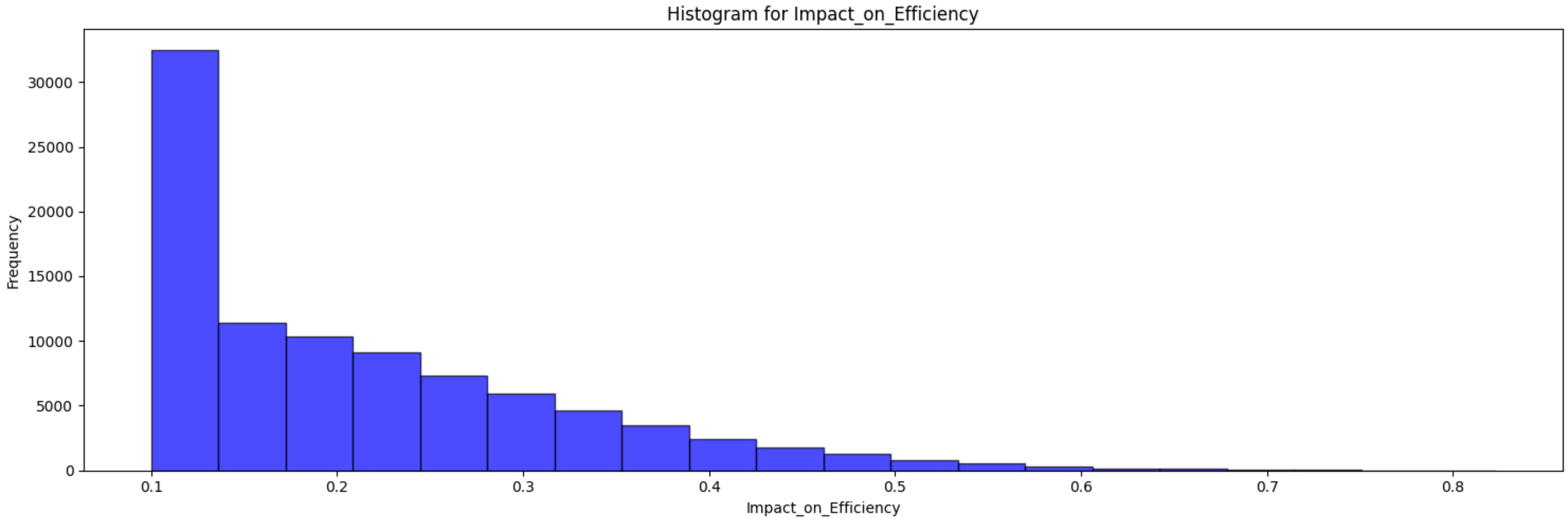


Histogram for Delivery_Times



Histogram for Downtime_Maintenance





Maintenance Services

In [31]:
```
df.groupby('Maintenance_Type')['Maintenance_Cost'].mean()
```

Out[31]:

	Maintenance_Cost
Maintenance_Type	
Engine Overhaul	4005.358675
Oil Change	299.064409
Tire Rotation	299.938745

dtype: float64

In [32]:
```
df[['Road_Conditions','Route_Info']].value_counts()
```

Out[32]:

		count
Road_Conditions	**Route_Info**	
Highway	**Highway**	18644
Rural	**Highway**	16031
Urban	**Highway**	11377
Highway	**Rural**	9118
	Urban	9099
Rural	**Rural**	8138
	Urban	8125
Urban	**Rural**	5781
	Urban	5687

dtype: int64

In [33]:
```
df['road_route_combo'] = df['Road_Conditions'] + ' - ' + df['Route_Info']
df.drop(columns=['Road_Conditions', 'Route_Info'], axis= 1, inplace= True)
```

In [34]:
```
df['road_route_combo'].value_counts()
```

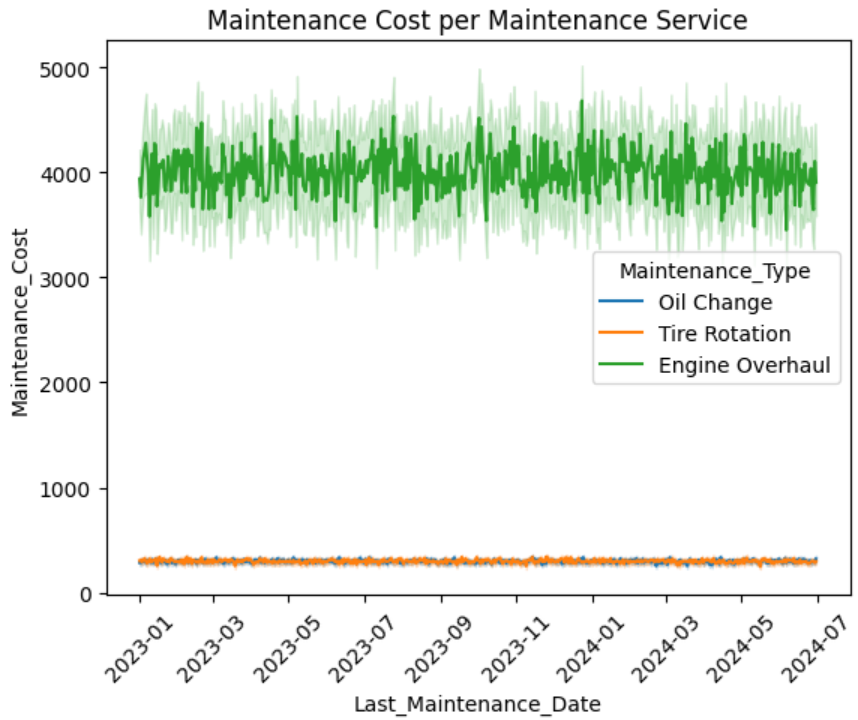
Out[34]:

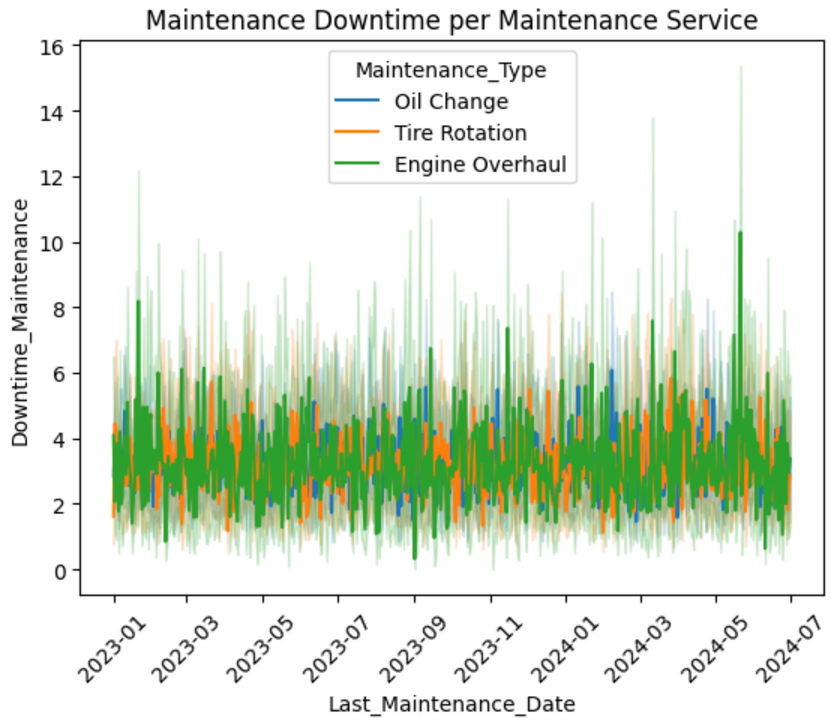
road_route_combo	count
Highway - Highway	18644
Rural - Highway	16031
Urban - Highway	11377
Highway - Rural	9118
Highway - Urban	9099
Rural - Rural	8138
Rural - Urban	8125
Urban - Rural	5781
Urban - Urban	5687

dtype: int64

In [35]:

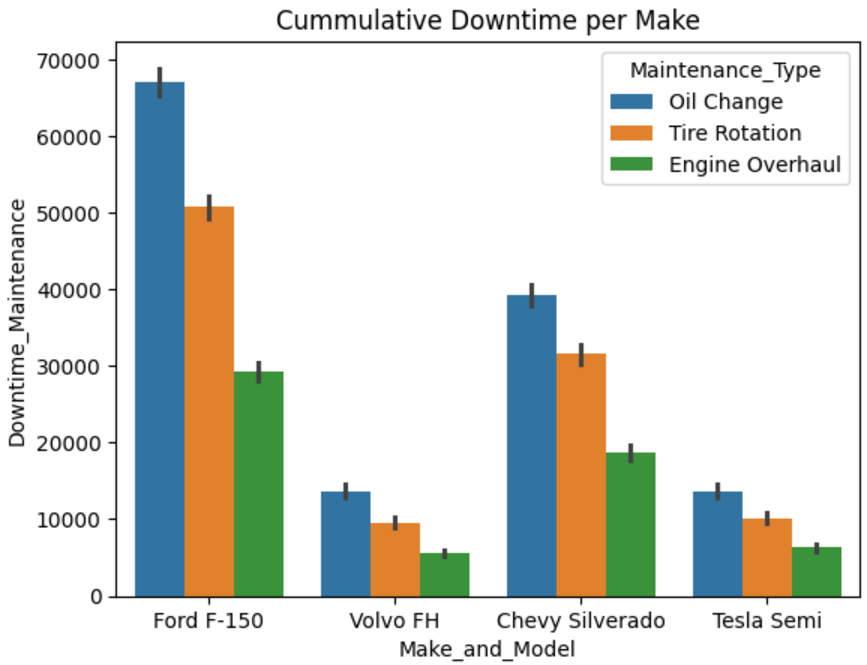
```
sns.lineplot(data = df, x = 'Last_Maintenance_Date', y = 'Maintenance_Cost', hue = 'Maintenance_Type')
plt.xticks(rotation = 45)
plt.title('Maintenance Cost per Maintenance Service')
plt.show();

sns.lineplot(data= df,  x = 'Last_Maintenance_Date', y = 'Downtime_Maintenance',  hue = 'Maintenance_Type' )
plt.title('Maintenance Downtime per Maintenance Service')
plt.xticks(rotation = 45)
plt.show();
```


Maintenance Cost per Maintenance Service
Maintenance_Cost
5000
4000
3000
2000
1000
0
Maintenance_Type
Oil Change
Tire Rotation
Engine Overhaul
2023-01
2023-03
2023-05
2023-07
2023-09
2023-11
2024-01
2024-03
2024-05
2024-07
Last_Maintenance_Date



In [36]:
```
df.drop(columns=['Engine_Temperature','Battery_Status'], axis= 1, inplace= True)
```

In [37]:
```
sns.barplot(data = df, x = 'Make_and_Model', y = 'Downtime_Maintenance', hue = 'Maintenance_Type', estimator= 'sum')
plt.title('Cummulative Downtime per Make')
plt.show()
```



In [38]:
```
df.groupby('Make_and_Model')['Maintenance_Type'].value_counts(ascending= False) / len(df) * 100
```

Out[38]:

Make_and_Model	Maintenance_Type	count
Chevy Silverado	Oil Change	13.471739
	Tire Rotation	10.575000
	Engine Overhaul	6.073913
Ford F-150	Oil Change	22.543478
	Tire Rotation	17.335870
	Engine Overhaul	10.030435
Tesla Semi	Oil Change	4.548913
	Tire Rotation	3.502174
	Engine Overhaul	2.040217
Volvo FH	Oil Change	4.531522
	Tire Rotation	3.427174
	Engine Overhaul	1.919565

dtype: float64

In [39]:
```
df['Weather_Conditions'].value_counts() / len(df)
```

Out[39]:

Weather_Conditions	count
Clear	0.798717
Rainy	0.151370
Snowy	0.030163
Windy	0.019750

dtype: float64

Dummy Variables for Remaining Categorical Variables

In [40]:
```
df_dummies = pd.get_dummies(df[['Maintenance_Type', 'Vehicle_Type', 'road_route_combo','Make_and_Model','Weather_Conditions','over_capacity/under_capacity']],
                             prefix=['Maintenance_Type', 'Vehicle_Type', 'road_route_combo','Make_and_Model','Weather_Conditions','over_capacity/under_capacity'],drop_first= True, dtype= int)

df = df.drop(['Maintenance_Type', 'Vehicle_Type', 'road_route_combo','Brake_Condition','Make_and_Model','Weather_Conditions','over_capacity/under_capacity'], axis=1)

df = pd.concat([df, df_dummies], axis=1)
```

Target Variable Adjustment:

- Using 50% Threshold for predictive score if maintenance will be needed. For example if a unit has a 50% proabbality score of requiring maintenance then it will assign a binary value of 1 else 0

In [41]:
```
df['maintenance_required'] = df['Predictive_Score'].apply(lambda x: 1 if x < 0.5 else 0)
```

In [42]:
```
df['maintenance_required_maintenance'] = (df['failure_anomaly_required_maintenance_match'] ==1) & (df['maintenance_required'] ==1).astype(int)
```

In [43]:
```
df.drop(columns= 'Predictive_Score', axis= 1, inplace =True)
df.drop(columns=['failure_anomaly_required_maintenance_match', 'maintenance_required'], axis= 1, inplace= True)
```

In [44]:
```
df['maintenance_required_maintenance'] = df['maintenance_required_maintenance'].astype(int)
```

In [45]:
```
df2 = df.copy()
```

In [46]:
```
df.drop(columns=['Last_Maintenance_Date'], axis= 1, inplace= True)
```

In [48]:
```
df.dropna(inplace= True)
df.isnull().sum().sum()
```

Out[48]:
```
0
```

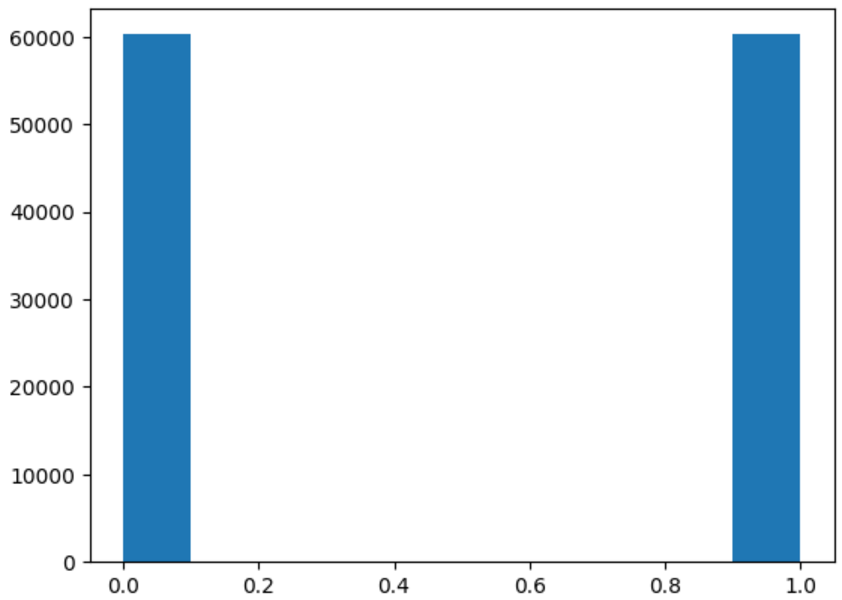
Machine Learning

In [ ]:
```
X = df.drop('maintenance_required_maintenance', axis= 1)
y = df['maintenance_required_maintenance']

smote = SMOTE()
model = LogisticRegression()
scaler = MinMaxScaler()
reduction = PCA(n_components=10)

x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.20, shuffle=True, random_state=42)

x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

x_balance, y_balance = smote.fit_resample(x_train_scaled,y_train)

x_pca = reduction.fit_transform(x_balance)
x_test_pca = reduction.transform(x_test_scaled)
```

Logistics Regression

In [50]:
```
plt.hist(y_balance)
```

Out[50]:
```
(array([60258.,     0.,     0.,     0.,     0.,     0.,     0.,     0.,
            0., 60258.]),
 array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
 <BarContainer object of 10 artists>)
```



In [51]:
```
param_grid = {
    'penalty': ['l1', 'l2'],
    'C': np.logspace(-4, 4, 20),
    'solver': ['liblinear', 'saga'],
    'max_iter': [100, 200, 500]
}

model = LogisticRegression()

random_search = RandomizedSearchCV(
    estimator=model,
    param_distributions=param_grid,
    n_iter=50,
    cv=5,
    scoring='accuracy',
    verbose=1,
    random_state=42,
    n_jobs=-1
)

random_search.fit(x_balance, y_balance)

print('Best Hyperparameters:', random_search.best_params_)
print('Best Accuracy:', random_search.best_score_)

best_model = random_search.best_estimator_
```

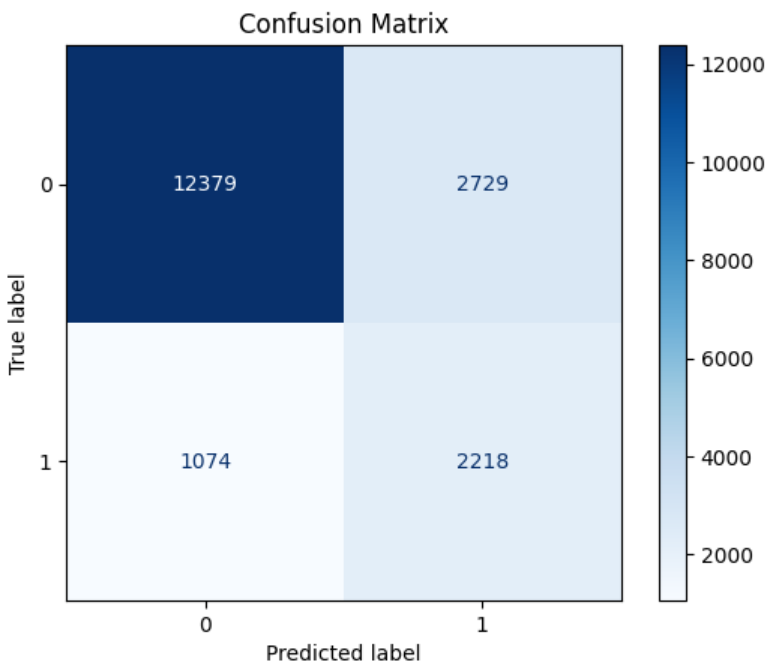
```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best Hyperparameters: {'solver': 'liblinear', 'penalty': 'l1', 'max_iter': 100, 'C': 0.0006951927961775605}
Best Accuracy: 0.866275040039677
```

In [52]:
```
y_hat = best_model.predict_proba(x_test_scaled)
predicted_classes = np.argmax(y_hat, axis=1)
```

In [53]:
```
print(classification_report(predicted_classes, y_test))
```

```
              precision    recall  f1-score   support

           0       0.82      0.92      0.87     13453
           1       0.67      0.45      0.54      4947

    accuracy                           0.79     18400
   macro avg       0.75      0.68      0.70     18400
weighted avg       0.78      0.79      0.78     18400
```

In [54]:
```
cm = confusion_matrix(y_test, predicted_classes)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)

disp.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.show()
```



XGBoost

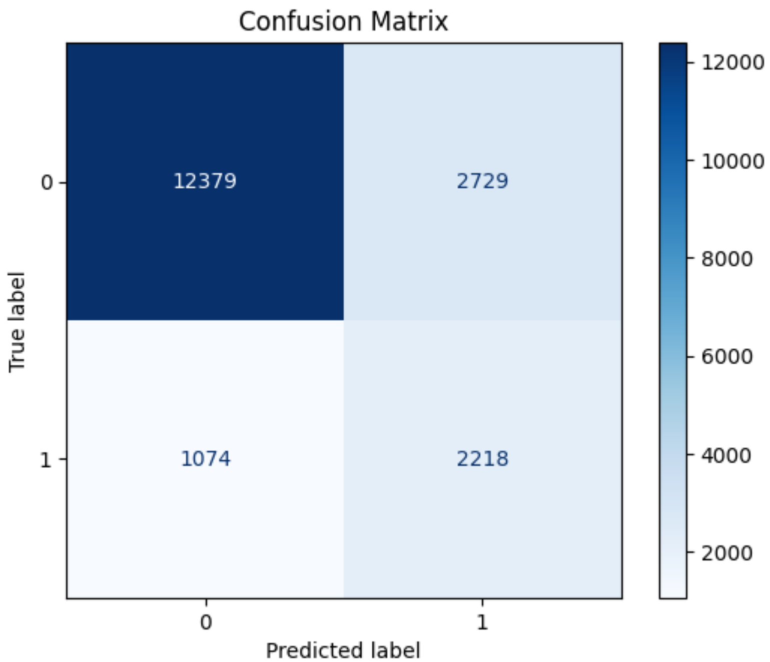
In [ ]:
```
xgb = XGBClassifier()
```

```
xgb.fit(x_balance, y_balance)
y_pred_clf = xgb.predict(x_test_scaled)


accuracy = accuracy_score(y_pred_clf, y_test)
print(f'Accuracy: {round(accuracy,2)}%')
```

```
Accuracy: 0.8%
```

In [56]:
```
print(classification_report(predicted_classes, y_test))


cm = confusion_matrix(y_test, predicted_classes)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.show()
```

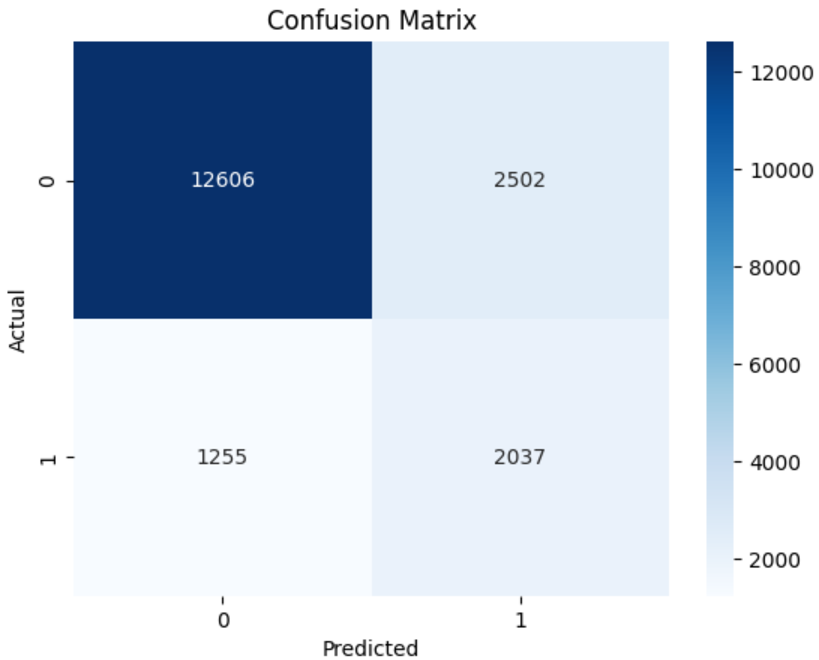
```
              precision    recall  f1-score   support

           0       0.82      0.92      0.87     13453
           1       0.67      0.45      0.54      4947

    accuracy                           0.79     18400
   macro avg       0.75      0.68      0.70     18400
weighted avg       0.78      0.79      0.78     18400
```



Random Forest

In [ ]:
```
rf = RandomForestClassifier()

param_grid = {
    'n_estimators': [50, 100, 200, 500],
    'max_depth': [None, 10, 20, 30, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': [1,10,100],
    'bootstrap': [True, False],
    'criterion': ['gini', 'entropy', 'log_loss'],
}

random_search = RandomizedSearchCV(estimator=rf, param_distributions=param_grid,
                                   cv=2, scoring='accuracy', verbose=1, n_jobs=-1)

random_search.fit(x_balance, y_balance)


y_pred_clf = random_search.predict(x_test_scaled)
accuracy = accuracy_score(y_test, y_pred_clf)

print('Best Hyperparameters:', random_search.best_params_)
print('Best Accuracy:', random_search.best_score_)

cm = confusion_matrix(y_test, y_pred_clf)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix")
plt.ylabel("Actual")
plt.xlabel("Predicted")
plt.show()

print(classification_report(y_test, y_pred_clf))
```

```
Fitting 2 folds for each of 10 candidates, totalling 20 fits
Best Hyperparameters: {'n_estimators': 200, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features': 1, 'max_depth': 20, 'criterion': 'gini', 'bootstrap': False}
Best Accuracy: 0.8690962195891001
```

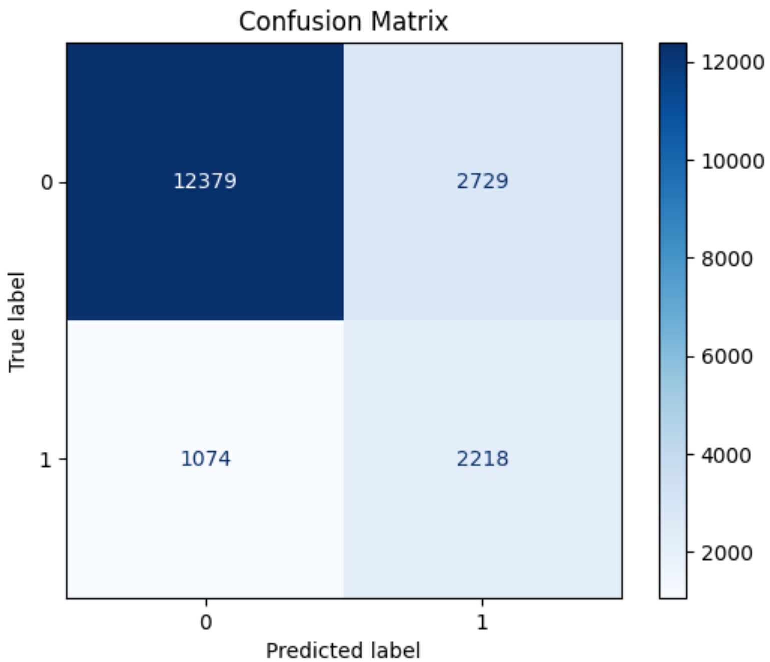



```
              precision    recall  f1-score   support

           0       0.91      0.83      0.87     15108
           1       0.45      0.62      0.52      3292

    accuracy                           0.80     18400
   macro avg       0.68      0.73      0.70     18400
weighted avg       0.83      0.80      0.81     18400
```

In [58]:

```
accuracy = accuracy_score(y_pred_clf, y_test)
print(classification_report(predicted_classes, y_test))
print(f"Accuracy: {accuracy}%")


cm = confusion_matrix(y_test, predicted_classes)

disp = ConfusionMatrixDisplay(confusion_matrix=cm)

disp.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.show()
```

```
              precision    recall  f1-score   support

           0       0.82      0.92      0.87     13453
           1       0.67      0.45      0.54      4947

    accuracy                           0.79     18400
   macro avg       0.75      0.68      0.70     18400
weighted avg       0.78      0.79      0.78     18400

Accuracy: 0.7958152173913043%
```

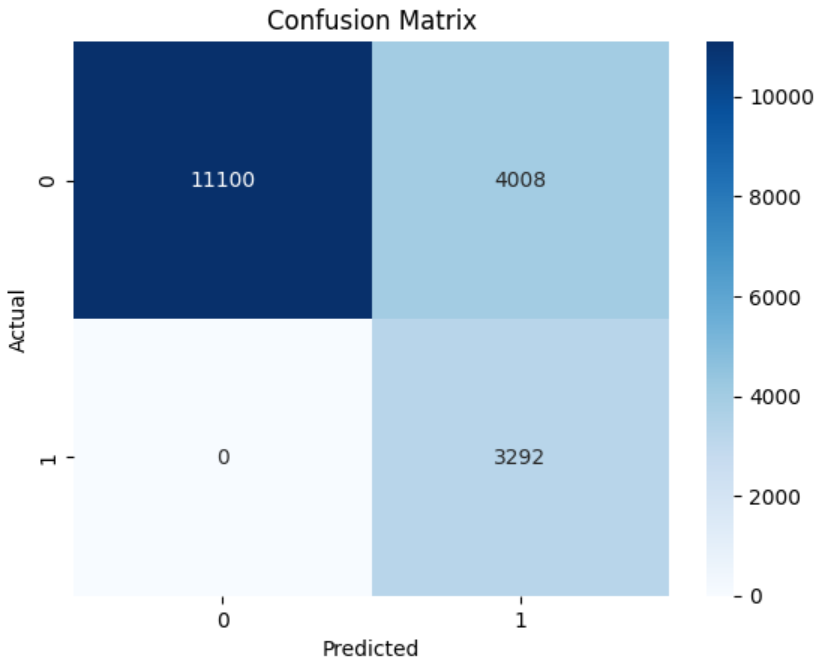


Histgradient Boost

In [59]:
```python
hist = HistGradientBoostingClassifier(random_state=42)

param_grid = {
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'max_iter': [100, 200, 500],
    'max_leaf_nodes': [10, 20, 31, 50],
    'max_depth': [None, 3, 5, 10],
    'min_samples_leaf': [1, 5, 10, 20],
    'l2_regularization': [0.0, 1.0, 10.0],
    'max_bins': [2, 255],
}

random_search = RandomizedSearchCV(estimator=hist,param_distributions=param_grid, n_iter=50, verbose=1,
                                   cv=3,
                                   scoring='accuracy',
```

```python
                                    n_jobs=-1,
                                    random_state=42)


random_search.fit(x_balance, y_balance)
y_pred_clf = random_search.predict(x_test_scaled)


print('Best Hyperparameters:', random_search.best_params_)
print('Best Accuracy:', random_search.best_score_)

accuracy = accuracy_score(y_pred_clf, y_test)
print(f'Accuracy: {round(accuracy,2)}%')


print(classification_report(y_test, y_pred_clf))


cm = confusion_matrix(y_test, y_pred_clf)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix")
plt.ylabel("Actual")
plt.xlabel("Predicted")
plt.show()
```

```
Fitting 3 folds for each of 50 candidates, totalling 150 fits
Best Hyperparameters: {'min_samples_leaf': 1, 'max_leaf_nodes': 50, 'max_iter': 100, 'max_depth': 3, 'max_bins': 255, 'learning_rate': 0.01, 'l2_regularization': 0.0}
Best Accuracy: 0.8602592186929536
Accuracy: 0.78%
              precision    recall  f1-score   support

           0       1.00      0.73      0.85     15108
           1       0.45      1.00      0.62      3292

    accuracy                           0.78     18400
   macro avg       0.73      0.87      0.73     18400
weighted avg       0.90      0.78      0.81     18400
```



Neural Nets

NN Classifier Model 1

In [ ]:
```
point = .001

NN_CLASF = Sequential([
    Dense(3050, input_shape=(x_balance.shape[1],), kernel_regularizer=l2(point)),
    LeakyReLU(alpha = .1),
    BatchNormalization(),
    Dropout(0.50),


    Dense(1050,  kernel_regularizer=l2(point)),
    LeakyReLU(alpha = .1),
    BatchNormalization(),
    Dropout(0.50),

    Dense(525, kernel_regularizer=l2(point)),
    LeakyReLU(alpha = .1),
    BatchNormalization(),
    Dropout(0.50),

    Dense(250, kernel_regularizer=l2(point)),
    LeakyReLU(alpha = .1),
```

```
    BatchNormalization(),
    Dropout(0.50),

    Dense(125, kernel_regularizer=l2(point)),
    LeakyReLU(alpha = .1),
    BatchNormalization(),
    Dropout(0.50),

    Dense(50, kernel_regularizer=l2(point)),
    LeakyReLU(alpha = .1),
    BatchNormalization(),
    Dropout(0.50),

    Dense(1, activation='sigmoid')
])

early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

lr_scheduler = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.9,
    patience=3
)

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

NN_CLASF.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])

NN_CLASF.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` objec
t as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
/usr/local/lib/python3.11/dist-packages/keras/src/layers/activations/leaky_relu.py:41: UserWarning: Argument `alpha` is deprecated. Use `negative_slope` instead.
  warnings.warn(
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 3050)	91,500
leaky_re_lu (LeakyReLU)	(None, 3050)	0
batch_normalization (BatchNormalization)	(None, 3050)	12,200
dropout (Dropout)	(None, 3050)	0
dense_1 (Dense)	(None, 1050)	3,203,550
leaky_re_lu_1 (LeakyReLU)	(None, 1050)	0
batch_normalization_1 (BatchNormalization)	(None, 1050)	4,200
dropout_1 (Dropout)	(None, 1050)	0
dense_2 (Dense)	(None, 525)	551,775
leaky_re_lu_2 (LeakyReLU)	(None, 525)	0
batch_normalization_2 (BatchNormalization)	(None, 525)	2,100
dropout_2 (Dropout)	(None, 525)	0
dense_3 (Dense)	(None, 250)	131,500
leaky_re_lu_3 (LeakyReLU)	(None, 250)	0
batch_normalization_3 (BatchNormalization)	(None, 250)	1,000
dropout_3 (Dropout)	(None, 250)	0
dense_4 (Dense)	(None, 125)	31,375
leaky_re_lu_4 (LeakyReLU)	(None, 125)	0
batch_normalization_4 (BatchNormalization)	(None, 125)	500
dropout_4 (Dropout)	(None, 125)	0
dense_5 (Dense)	(None, 50)	6,300
leaky_re_lu_5 (LeakyReLU)	(None, 50)	0
batch_normalization_5 (BatchNormalization)	(None, 50)	200
dropout_5 (Dropout)	(None, 50)	0
dense_6 (Dense)	(None, 1)	51

Total params: 4,036,251 (15.40 MB)

Trainable params: 4,026,151 (15.36 MB)

Non-trainable params: 10,100 (39.45 KB)

In [61]:
```
history = NN_CLASF.fit(x_balance, y_balance, epochs=100, batch_size=300, validation_data=(x_test_scaled, y_test),
                       callbacks = [early_stopping, lr_scheduler])
```

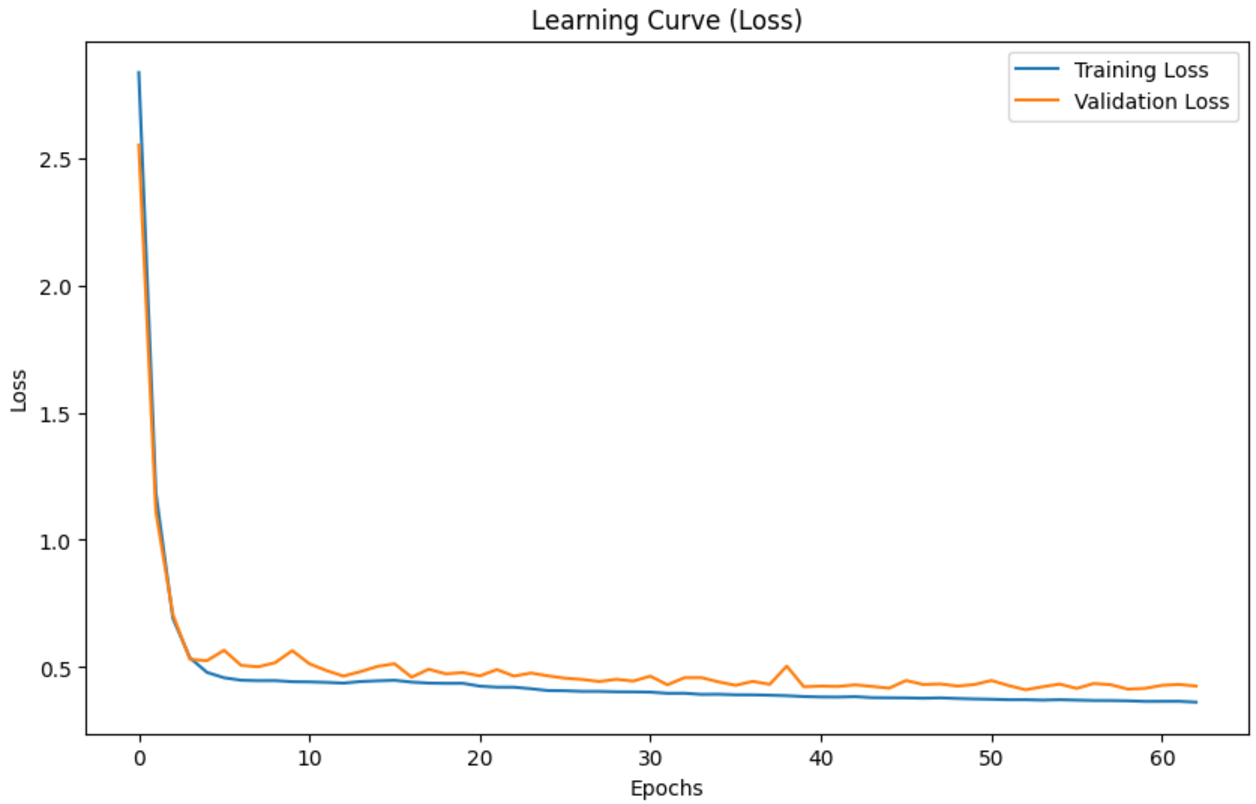
```
Epoch 1/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 29s 62ms/step - accuracy: 0.5161 - loss: 3.4194 - val_accuracy: 0.1789 - val_loss: 2.5496 - learning_rate: 0.0010
Epoch 2/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 63ms/step - accuracy: 0.7991 - loss: 1.4119 - val_accuracy: 0.6128 - val_loss: 1.1110 - learning_rate: 0.0010
Epoch 3/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 61ms/step - accuracy: 0.8312 - loss: 0.7590 - val_accuracy: 0.7525 - val_loss: 0.7057 - learning_rate: 0.0010
Epoch 4/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 60ms/step - accuracy: 0.8422 - loss: 0.5573 - val_accuracy: 0.7843 - val_loss: 0.5310 - learning_rate: 0.0010
Epoch 5/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 60ms/step - accuracy: 0.8482 - loss: 0.4872 - val_accuracy: 0.7828 - val_loss: 0.5259 - learning_rate: 0.0010
Epoch 6/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 59ms/step - accuracy: 0.8507 - loss: 0.4624 - val_accuracy: 0.7451 - val_loss: 0.5667 - learning_rate: 0.0010
Epoch 7/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 61ms/step - accuracy: 0.8542 - loss: 0.4474 - val_accuracy: 0.7848 - val_loss: 0.5074 - learning_rate: 0.0010
Epoch 8/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 62ms/step - accuracy: 0.8515 - loss: 0.4532 - val_accuracy: 0.7853 - val_loss: 0.5018 - learning_rate: 0.0010
Epoch 9/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 62ms/step - accuracy: 0.8537 - loss: 0.4474 - val_accuracy: 0.7831 - val_loss: 0.5180 - learning_rate: 0.0010
Epoch 10/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 26s 64ms/step - accuracy: 0.8541 - loss: 0.4442 - val_accuracy: 0.7471 - val_loss: 0.5653 - learning_rate: 0.0010
Epoch 11/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 62ms/step - accuracy: 0.8556 - loss: 0.4422 - val_accuracy: 0.7841 - val_loss: 0.5138 - learning_rate: 0.0010
Epoch 12/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 61ms/step - accuracy: 0.8549 - loss: 0.4442 - val_accuracy: 0.7847 - val_loss: 0.4870 - learning_rate: 9.0000e-04
Epoch 13/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 61ms/step - accuracy: 0.8558 - loss: 0.4374 - val_accuracy: 0.7837 - val_loss: 0.4653 - learning_rate: 9.0000e-04
Epoch 14/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 59ms/step - accuracy: 0.8553 - loss: 0.4446 - val_accuracy: 0.7832 - val_loss: 0.4827 - learning_rate: 9.0000e-04
Epoch 15/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 59ms/step - accuracy: 0.8570 - loss: 0.4437 - val_accuracy: 0.7840 - val_loss: 0.5034 - learning_rate: 9.0000e-04
Epoch 16/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 58ms/step - accuracy: 0.8555 - loss: 0.4492 - val_accuracy: 0.7795 - val_loss: 0.5137 - learning_rate: 9.0000e-04
Epoch 17/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 60ms/step - accuracy: 0.8578 - loss: 0.4434 - val_accuracy: 0.7835 - val_loss: 0.4611 - learning_rate: 8.1000e-04
Epoch 18/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 61ms/step - accuracy: 0.8596 - loss: 0.4342 - val_accuracy: 0.7834 - val_loss: 0.4923 - learning_rate: 8.1000e-04
Epoch 19/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 59ms/step - accuracy: 0.8577 - loss: 0.4356 - val_accuracy: 0.7829 - val_loss: 0.4741 - learning_rate: 8.1000e-04
Epoch 20/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 61ms/step - accuracy: 0.8584 - loss: 0.4389 - val_accuracy: 0.7842 - val_loss: 0.4790 - learning_rate: 8.1000e-04
Epoch 21/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 63ms/step - accuracy: 0.8576 - loss: 0.4299 - val_accuracy: 0.7833 - val_loss: 0.4661 - learning_rate: 7.2900e-04
Epoch 22/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 62ms/step - accuracy: 0.8603 - loss: 0.4173 - val_accuracy: 0.7835 - val_loss: 0.4909 - learning_rate: 7.2900e-04
Epoch 23/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 60ms/step - accuracy: 0.8572 - loss: 0.4254 - val_accuracy: 0.7833 - val_loss: 0.4653 - learning_rate: 7.2900e-04
Epoch 24/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 62ms/step - accuracy: 0.8578 - loss: 0.4190 - val_accuracy: 0.7835 - val_loss: 0.4773 - learning_rate: 6.5610e-04
Epoch 25/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 60ms/step - accuracy: 0.8603 - loss: 0.4077 - val_accuracy: 0.7818 - val_loss: 0.4660 - learning_rate: 6.5610e-04
Epoch 26/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 60ms/step - accuracy: 0.8574 - loss: 0.4104 - val_accuracy: 0.7835 - val_loss: 0.4570 - learning_rate: 6.5610e-04
Epoch 27/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 25s 61ms/step - accuracy: 0.8612 - loss: 0.4027 - val_accuracy: 0.7829 - val_loss: 0.4521 - learning_rate: 6.5610e-04
Epoch 28/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 28s 71ms/step - accuracy: 0.8606 - loss: 0.4053 - val_accuracy: 0.7804 - val_loss: 0.4440 - learning_rate: 6.5610e-04
```

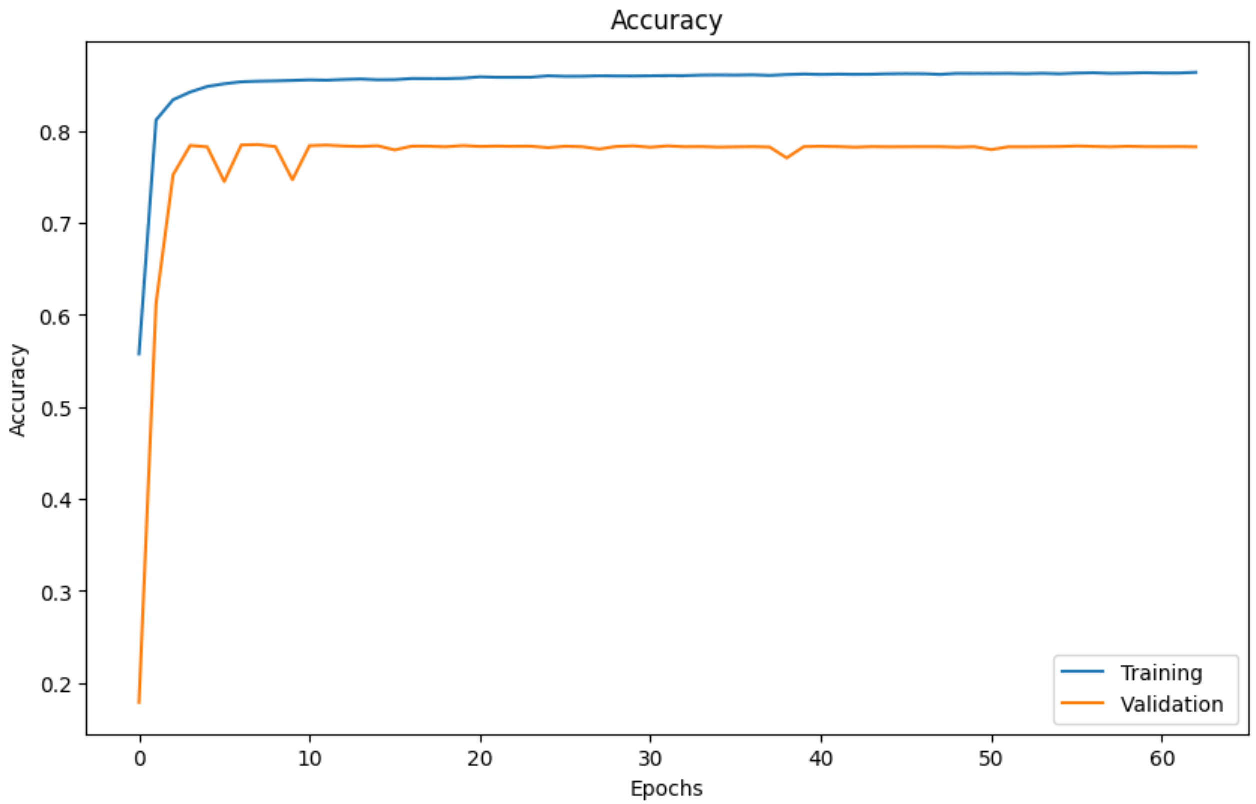
```
Epoch 29/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 29s 71ms/step - accuracy: 0.8595 - loss: 0.4035 - val_accuracy: 0.7832 - val_loss: 0.4525 - learning_rate: 6.5610e-04
Epoch 30/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 28s 69ms/step - accuracy: 0.8587 - loss: 0.4047 - val_accuracy: 0.7839 - val_loss: 0.4458 - learning_rate: 6.5610e-04
Epoch 31/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 27s 67ms/step - accuracy: 0.8603 - loss: 0.4027 - val_accuracy: 0.7823 - val_loss: 0.4650 - learning_rate: 6.5610e-04
Epoch 32/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 27s 68ms/step - accuracy: 0.8588 - loss: 0.4019 - val_accuracy: 0.7839 - val_loss: 0.4310 - learning_rate: 5.9049e-04
Epoch 33/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 27s 66ms/step - accuracy: 0.8587 - loss: 0.4010 - val_accuracy: 0.7829 - val_loss: 0.4591 - learning_rate: 5.9049e-04
Epoch 34/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 28s 69ms/step - accuracy: 0.8615 - loss: 0.3915 - val_accuracy: 0.7831 - val_loss: 0.4591 - learning_rate: 5.9049e-04
Epoch 35/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 28s 69ms/step - accuracy: 0.8597 - loss: 0.3973 - val_accuracy: 0.7824 - val_loss: 0.4425 - learning_rate: 5.9049e-04
Epoch 36/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 28s 70ms/step - accuracy: 0.8623 - loss: 0.3907 - val_accuracy: 0.7828 - val_loss: 0.4291 - learning_rate: 5.3144e-04
Epoch 37/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 28s 70ms/step - accuracy: 0.8606 - loss: 0.3931 - val_accuracy: 0.7830 - val_loss: 0.4444 - learning_rate: 5.3144e-04
Epoch 38/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 28s 70ms/step - accuracy: 0.8580 - loss: 0.3935 - val_accuracy: 0.7826 - val_loss: 0.4330 - learning_rate: 5.3144e-04
Epoch 39/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 29s 71ms/step - accuracy: 0.8622 - loss: 0.3858 - val_accuracy: 0.7708 - val_loss: 0.5045 - learning_rate: 5.3144e-04
Epoch 40/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 29s 72ms/step - accuracy: 0.8612 - loss: 0.3885 - val_accuracy: 0.7830 - val_loss: 0.4233 - learning_rate: 4.7830e-04
Epoch 41/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 27s 67ms/step - accuracy: 0.8610 - loss: 0.3851 - val_accuracy: 0.7834 - val_loss: 0.4259 - learning_rate: 4.7830e-04
Epoch 42/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 59ms/step - accuracy: 0.8612 - loss: 0.3841 - val_accuracy: 0.7831 - val_loss: 0.4245 - learning_rate: 4.7830e-04
Epoch 43/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 58ms/step - accuracy: 0.8625 - loss: 0.3819 - val_accuracy: 0.7824 - val_loss: 0.4309 - learning_rate: 4.7830e-04
Epoch 44/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 60ms/step - accuracy: 0.8615 - loss: 0.3821 - val_accuracy: 0.7830 - val_loss: 0.4247 - learning_rate: 4.3047e-04
Epoch 45/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 59ms/step - accuracy: 0.8609 - loss: 0.3816 - val_accuracy: 0.7828 - val_loss: 0.4180 - learning_rate: 4.3047e-04
Epoch 46/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 59ms/step - accuracy: 0.8616 - loss: 0.3806 - val_accuracy: 0.7829 - val_loss: 0.4480 - learning_rate: 4.3047e-04
Epoch 47/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 60ms/step - accuracy: 0.8619 - loss: 0.3789 - val_accuracy: 0.7830 - val_loss: 0.4323 - learning_rate: 4.3047e-04
Epoch 48/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 57ms/step - accuracy: 0.8611 - loss: 0.3799 - val_accuracy: 0.7830 - val_loss: 0.4344 - learning_rate: 4.3047e-04
Epoch 49/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 60ms/step - accuracy: 0.8631 - loss: 0.3757 - val_accuracy: 0.7824 - val_loss: 0.4262 - learning_rate: 3.8742e-04
Epoch 50/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 60ms/step - accuracy: 0.8633 - loss: 0.3753 - val_accuracy: 0.7830 - val_loss: 0.4320 - learning_rate: 3.8742e-04
Epoch 51/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 58ms/step - accuracy: 0.8617 - loss: 0.3758 - val_accuracy: 0.7798 - val_loss: 0.4480 - learning_rate: 3.8742e-04
Epoch 52/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 57ms/step - accuracy: 0.8617 - loss: 0.3745 - val_accuracy: 0.7828 - val_loss: 0.4283 - learning_rate: 3.4868e-04
Epoch 53/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 56ms/step - accuracy: 0.8639 - loss: 0.3701 - val_accuracy: 0.7828 - val_loss: 0.4116 - learning_rate: 3.4868e-04
Epoch 54/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 56ms/step - accuracy: 0.8629 - loss: 0.3702 - val_accuracy: 0.7830 - val_loss: 0.4234 - learning_rate: 3.4868e-04
Epoch 55/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 56ms/step - accuracy: 0.8633 - loss: 0.3718 - val_accuracy: 0.7832 - val_loss: 0.4338 - learning_rate: 3.4868e-04
Epoch 56/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 57ms/step - accuracy: 0.8621 - loss: 0.3737 - val_accuracy: 0.7837 - val_loss: 0.4169 - learning_rate: 3.4868e-04
```

```
Epoch 57/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 57ms/step - accuracy: 0.8632 - loss: 0.3707 - val_accuracy: 0.7833 - val_loss: 0.4361 - learning_rate: 3.1381e-04
Epoch 58/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 59ms/step - accuracy: 0.8625 - loss: 0.3683 - val_accuracy: 0.7828 - val_loss: 0.4314 - learning_rate: 3.1381e-04
Epoch 59/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 57ms/step - accuracy: 0.8631 - loss: 0.3672 - val_accuracy: 0.7834 - val_loss: 0.4142 - learning_rate: 3.1381e-04
Epoch 60/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 57ms/step - accuracy: 0.8621 - loss: 0.3689 - val_accuracy: 0.7830 - val_loss: 0.4170 - learning_rate: 2.8243e-04
Epoch 61/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 58ms/step - accuracy: 0.8633 - loss: 0.3643 - val_accuracy: 0.7830 - val_loss: 0.4290 - learning_rate: 2.8243e-04
Epoch 62/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 23s 58ms/step - accuracy: 0.8634 - loss: 0.3659 - val_accuracy: 0.7831 - val_loss: 0.4325 - learning_rate: 2.8243e-04
Epoch 63/100
402/402 ━━━━━━━━━━━━━━━━━━━━ 24s 59ms/step - accuracy: 0.8653 - loss: 0.3618 - val_accuracy: 0.7828 - val_loss: 0.4261 - learning_rate: 2.5419e-04
```

In [62]:

```
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')

plt.title('Learning Curve (Loss)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Training')
plt.plot(history.history['val_accuracy'], label='Validation ')

plt.title('Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```


Learning Curve (Loss)
Training Loss
Validation Loss
Loss
Epochs
0.5
1.0
1.5
2.0
2.5
0
10
20
30
40
50
60

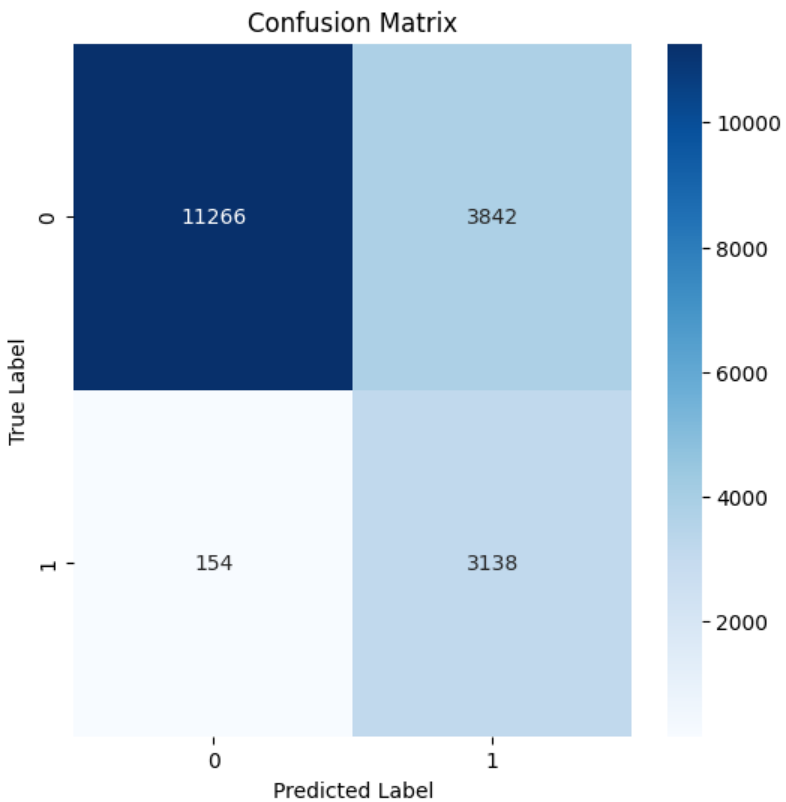


In [ ]:
```
y_pred_prob = NN_CLASF.predict(x_test_scaled)
y_pred = (y_pred_prob > 0.5).astype(int)

conf_matrix = confusion_matrix(y_test, y_pred)
print(accuracy_score(y_test, y_pred))
```

```
575/575 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step
0.7828260869565218
```

In [ ]:
```
print(classification_report(y_test, y_pred))

plt.figure(figsize=(6,6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues")

plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()
```

```
              precision    recall  f1-score   support

           0       0.99      0.75      0.85     15108
           1       0.45      0.95      0.61      3292

    accuracy                           0.78     18400
   macro avg       0.72      0.85      0.73     18400
weighted avg       0.89      0.78      0.81     18400
```



LSTM Time Series

In [65]:
```
df2.isnull().sum().sum()
```

Out[65]:
```
0
```

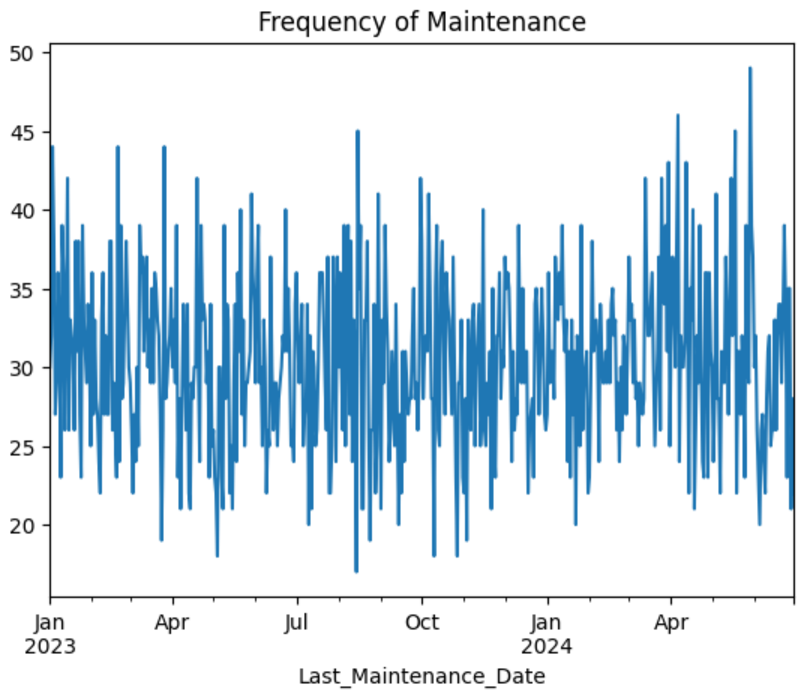
In [66]:
```
df2['maintenance_required_maintenance'].value_counts()
```

Out[66]:

	count
maintenance_required_maintenance	
0	75366
1	16634

dtype: int64

In [67]:
```
df2.groupby('Last_Maintenance_Date')['maintenance_required_maintenance'].sum().plot(kind = 'line')
plt.title('Frequency of Maintenance')
plt.show()
```



In [68]:
```
df2 = df2.sort_values('Last_Maintenance_Date')
df2.head()
```

Out[68]:

	Last_Maintenance_Date	Year_of_Manufacture	Usage_Hours	Maintenance_Cost	Tire_Pressure	Fuel_Consumption	Vibration_Levels	Oil_Quality	Delivery_Times	Downtime_Maintenance	...	road_route_combo_Urban - Rural	road_route_comt
57153	2023-01-01	2016	1054	196.525794	20.000000	10.745958	4.903479	61.013368	30.000000	0.0	...	0	
88871	2023-01-01	2020	2108	383.516409	55.000000	6.163432	0.625396	94.132347	117.444974	0.0	...	0	
83756	2023-01-01	2012	3507	257.008075	20.000000	7.035849	0.922788	67.580466	139.058633	0.0	...	0	
78985	2023-01-01	2012	1311	162.213739	55.000000	20.000000	5.623135	71.618660	183.387737	0.0	...	0	
43463	2023-01-01	2021	6526	292.257096	39.479994	10.324075	0.603263	78.747421	46.110788	0.0	...	0	

5 rows × 31 columns

In [69]:

```
df2['timestamp'] = pd.to_datetime(df2['Last_Maintenance_Date'])
df2['unix_timestamp'] = df2['timestamp'].astype(int) // 10**9
```

In [70]:

```
maintenance_count = pd.DataFrame(df2.groupby('unix_timestamp')['maintenance_required_maintenance'].sum())
maintenance_count.reset_index(inplace= True)
maintenance_count.rename(columns= {'maintenance_required_maintenance': 'maintenance_count'}, inplace= True)
maintenance_count
```

Out[70]:

	unix_timestamp	maintenance_count
0	1672531200	30
1	1672617600	32
2	1672704000	44
3	1672790400	38
4	1672876800	27
...	...	...
542	1719360000	35
543	1719446400	35
544	1719532800	21
545	1719619200	28
546	1719705600	22

547 rows × 2 columns

In [71]:

```
maintenance_count['scaled_count'] = scaler.fit_transform(maintenance_count[['maintenance_count']])
```

In [72]:

```
maintenance_count
```

Out[72]:

	unix_timestamp	maintenance_count	scaled_count
0	1672531200	30	0.40625
1	1672617600	32	0.46875
2	1672704000	44	0.84375
3	1672790400	38	0.65625
4	1672876800	27	0.31250
...	...	...	...
542	1719360000	35	0.56250
543	1719446400	35	0.56250
544	1719532800	21	0.12500
545	1719619200	28	0.34375
546	1719705600	22	0.15625

547 rows × 3 columns

In [ ]:

```
maintenance_count['timestamp'] = pd.to_datetime(maintenance_count['unix_timestamp'], unit='s')
data = maintenance_count['scaled_count'].values.reshape(-1, 1)

seq_length = 30
def create_sequences(data, seq_length):
    sequences, labels = [], []
    for i in range(len(data) - seq_length):
        sequences.append(data[i:i + seq_length])
        labels.append(data[i + seq_length])
    return np.array(sequences), np.array(labels)

X, y = create_sequences(data, seq_length)

train_size = int(len(X) * 0.7)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

model = Sequential([
    LSTM(560, return_sequences=True, input_shape=(seq_length, 1)),
    BatchNormalization(),
    Dropout(0.30),
    LSTM(230),
    BatchNormalization(),
    Dropout(0.30),
    Dense(1)
])

model.summary()

# ****Credit to OpenAI Chat GPT for helping develop the sequence code and Module 4 from class in Time Series for LSTM*****
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
lstm_2 (LSTM)	(None, 30, 560)	1,258,880
batch_normalization_8 (BatchNormalization)	(None, 30, 560)	2,240
dropout_8 (Dropout)	(None, 30, 560)	0
lstm_3 (LSTM)	(None, 230)	727,720
batch_normalization_9 (BatchNormalization)	(None, 230)	920
dropout_9 (Dropout)	(None, 230)	0
dense_8 (Dense)	(None, 1)	231

Total params: 1,989,991 (7.59 MB)

Trainable params: 1,988,411 (7.59 MB)

Non-trainable params: 1,580 (6.17 KB)

In [80]:
```
learning_rate = 0.0005
model.compile(optimizer=Adam(learning_rate=learning_rate), loss='mse')
history = model.fit(X_train, y_train, epochs=100, batch_size=20, validation_data=(X_test, y_test))
```

```
Epoch 1/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 5s 129ms/step - loss: 3.2674 - val_loss: 0.1157
Epoch 2/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 115ms/step - loss: 1.2899 - val_loss: 0.0635
Epoch 3/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 116ms/step - loss: 0.8249 - val_loss: 0.0389
Epoch 4/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 118ms/step - loss: 1.1242 - val_loss: 0.0323
Epoch 5/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 114ms/step - loss: 0.8598 - val_loss: 0.0371
Epoch 6/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 114ms/step - loss: 0.8363 - val_loss: 0.0318
Epoch 7/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 111ms/step - loss: 0.6780 - val_loss: 0.0312
Epoch 8/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 112ms/step - loss: 0.6858 - val_loss: 0.0292
Epoch 9/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 111ms/step - loss: 0.9120 - val_loss: 0.0294
Epoch 10/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 114ms/step - loss: 0.5654 - val_loss: 0.0341
Epoch 11/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 116ms/step - loss: 0.7642 - val_loss: 0.0317
Epoch 12/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 114ms/step - loss: 0.6557 - val_loss: 0.0297
Epoch 13/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 113ms/step - loss: 0.5629 - val_loss: 0.0309
Epoch 14/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 113ms/step - loss: 0.4927 - val_loss: 0.0291
Epoch 15/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 117ms/step - loss: 0.4072 - val_loss: 0.0296
Epoch 16/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 121ms/step - loss: 0.3067 - val_loss: 0.0310
Epoch 17/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 120ms/step - loss: 0.3316 - val_loss: 0.0304
Epoch 18/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 122ms/step - loss: 0.2772 - val_loss: 0.0305
Epoch 19/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.2350 - val_loss: 0.0292
Epoch 20/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.2356 - val_loss: 0.0307
Epoch 21/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 129ms/step - loss: 0.1792 - val_loss: 0.0375
Epoch 22/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 125ms/step - loss: 0.1990 - val_loss: 0.0371
Epoch 23/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 127ms/step - loss: 0.1433 - val_loss: 0.0296
Epoch 24/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 125ms/step - loss: 0.1653 - val_loss: 0.0292
Epoch 25/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 127ms/step - loss: 0.1223 - val_loss: 0.0449
Epoch 26/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 127ms/step - loss: 0.0986 - val_loss: 0.0446
Epoch 27/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 124ms/step - loss: 0.0980 - val_loss: 0.0316
Epoch 28/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.0978 - val_loss: 0.0361
```

```
Epoch 29/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 123ms/step - loss: 0.0839 - val_loss: 0.0330
Epoch 30/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 128ms/step - loss: 0.0845 - val_loss: 0.0390
Epoch 31/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.0735 - val_loss: 0.0308
Epoch 32/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 122ms/step - loss: 0.0684 - val_loss: 0.0311
Epoch 33/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 120ms/step - loss: 0.0729 - val_loss: 0.0550
Epoch 34/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 124ms/step - loss: 0.0555 - val_loss: 0.0604
Epoch 35/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 123ms/step - loss: 0.0851 - val_loss: 0.0376
Epoch 36/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 130ms/step - loss: 0.0593 - val_loss: 0.0354
Epoch 37/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 125ms/step - loss: 0.0577 - val_loss: 0.0412
Epoch 38/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 125ms/step - loss: 0.0669 - val_loss: 0.0620
Epoch 39/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 122ms/step - loss: 0.0682 - val_loss: 0.0422
Epoch 40/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 118ms/step - loss: 0.0652 - val_loss: 0.0354
Epoch 41/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 117ms/step - loss: 0.0473 - val_loss: 0.0441
Epoch 42/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 120ms/step - loss: 0.0560 - val_loss: 0.0311
Epoch 43/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 125ms/step - loss: 0.0470 - val_loss: 0.0365
Epoch 44/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 129ms/step - loss: 0.0441 - val_loss: 0.0407
Epoch 45/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 128ms/step - loss: 0.0455 - val_loss: 0.0892
Epoch 46/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 128ms/step - loss: 0.0481 - val_loss: 0.6179
Epoch 47/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 129ms/step - loss: 0.0439 - val_loss: 0.2515
Epoch 48/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.0437 - val_loss: 0.3752
Epoch 49/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.0477 - val_loss: 0.0315
Epoch 50/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 125ms/step - loss: 0.0422 - val_loss: 0.1183
Epoch 51/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 123ms/step - loss: 0.0401 - val_loss: 0.0351
Epoch 52/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 124ms/step - loss: 0.0389 - val_loss: 0.0669
Epoch 53/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 120ms/step - loss: 0.0414 - val_loss: 0.0360
Epoch 54/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 127ms/step - loss: 0.0327 - val_loss: 0.0345
Epoch 55/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 123ms/step - loss: 0.0351 - val_loss: 0.0377
Epoch 56/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.0325 - val_loss: 0.0360
```

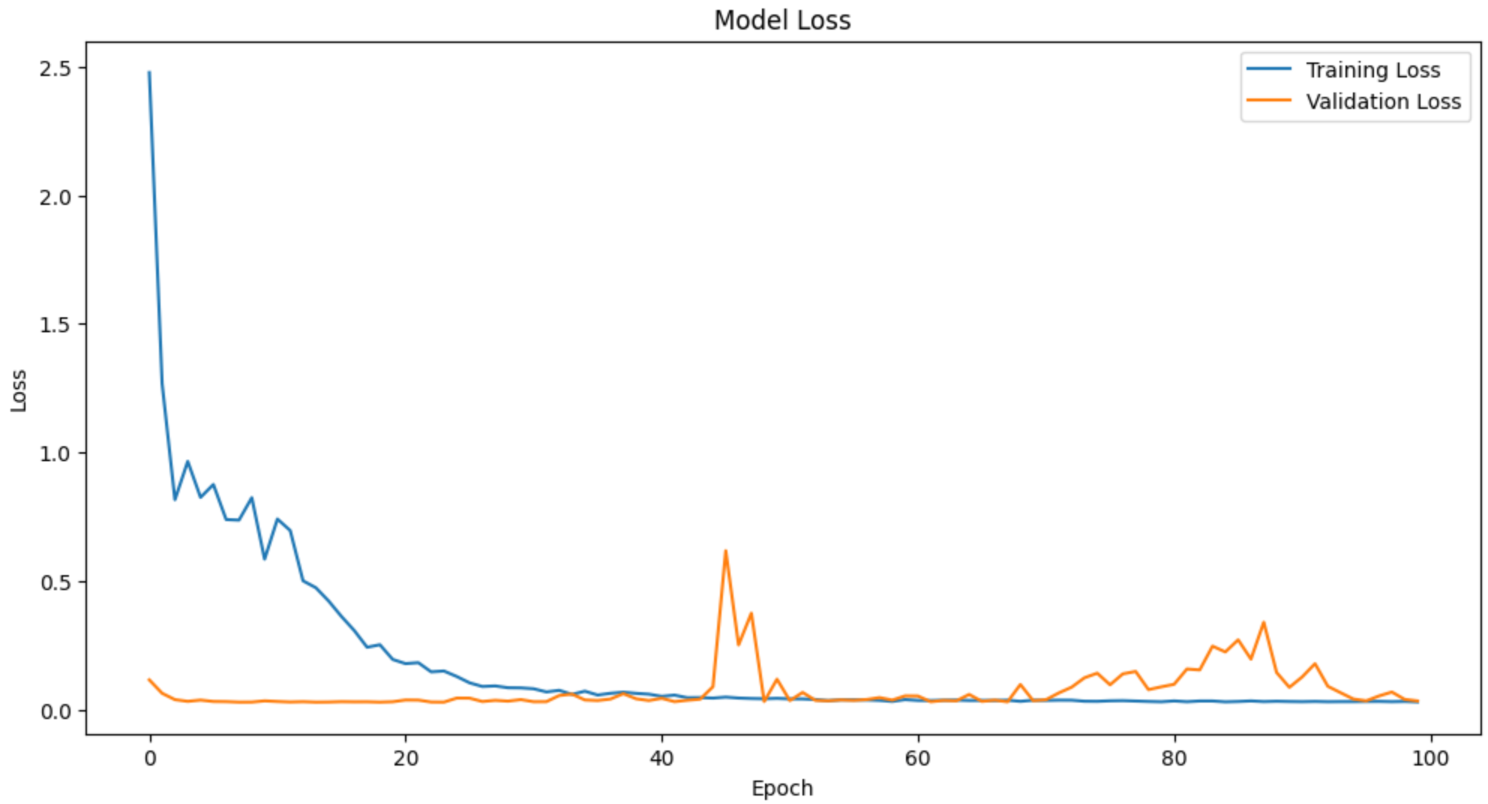
```
Epoch 57/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 120ms/step - loss: 0.0382 - val_loss: 0.0393
Epoch 58/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 120ms/step - loss: 0.0369 - val_loss: 0.0468
Epoch 59/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 123ms/step - loss: 0.0320 - val_loss: 0.0371
Epoch 60/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 125ms/step - loss: 0.0372 - val_loss: 0.0534
Epoch 61/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 3s 133ms/step - loss: 0.0393 - val_loss: 0.0529
Epoch 62/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 129ms/step - loss: 0.0330 - val_loss: 0.0308
Epoch 63/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 130ms/step - loss: 0.0368 - val_loss: 0.0360
Epoch 64/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 130ms/step - loss: 0.0347 - val_loss: 0.0342
Epoch 65/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 130ms/step - loss: 0.0347 - val_loss: 0.0591
Epoch 66/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 124ms/step - loss: 0.0377 - val_loss: 0.0321
Epoch 67/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 121ms/step - loss: 0.0359 - val_loss: 0.0372
Epoch 68/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.0379 - val_loss: 0.0305
Epoch 69/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.0305 - val_loss: 0.0978
Epoch 70/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 118ms/step - loss: 0.0386 - val_loss: 0.0348
Epoch 71/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 118ms/step - loss: 0.0325 - val_loss: 0.0387
Epoch 72/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 120ms/step - loss: 0.0355 - val_loss: 0.0649
Epoch 73/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 122ms/step - loss: 0.0396 - val_loss: 0.0873
Epoch 74/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.0319 - val_loss: 0.1238
Epoch 75/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 124ms/step - loss: 0.0280 - val_loss: 0.1418
Epoch 76/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 123ms/step - loss: 0.0310 - val_loss: 0.0965
Epoch 77/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 121ms/step - loss: 0.0373 - val_loss: 0.1391
Epoch 78/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 121ms/step - loss: 0.0340 - val_loss: 0.1485
Epoch 79/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.0319 - val_loss: 0.0778
Epoch 80/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 125ms/step - loss: 0.0299 - val_loss: 0.0887
Epoch 81/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 119ms/step - loss: 0.0320 - val_loss: 0.0985
Epoch 82/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 116ms/step - loss: 0.0314 - val_loss: 0.1579
Epoch 83/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 117ms/step - loss: 0.0352 - val_loss: 0.1544
Epoch 84/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 122ms/step - loss: 0.0344 - val_loss: 0.2469
```

```
Epoch 85/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 121ms/step - loss: 0.0308 - val_loss: 0.2243
Epoch 86/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 125ms/step - loss: 0.0306 - val_loss: 0.2715
Epoch 87/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 123ms/step - loss: 0.0292 - val_loss: 0.1964
Epoch 88/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 120ms/step - loss: 0.0310 - val_loss: 0.3399
Epoch 89/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 118ms/step - loss: 0.0329 - val_loss: 0.1445
Epoch 90/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 120ms/step - loss: 0.0309 - val_loss: 0.0867
Epoch 91/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 123ms/step - loss: 0.0313 - val_loss: 0.1273
Epoch 92/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 124ms/step - loss: 0.0314 - val_loss: 0.1787
Epoch 93/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 121ms/step - loss: 0.0314 - val_loss: 0.0908
Epoch 94/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 121ms/step - loss: 0.0296 - val_loss: 0.0657
Epoch 95/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 129ms/step - loss: 0.0327 - val_loss: 0.0411
Epoch 96/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 3s 133ms/step - loss: 0.0301 - val_loss: 0.0342
Epoch 97/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 131ms/step - loss: 0.0285 - val_loss: 0.0533
Epoch 98/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 126ms/step - loss: 0.0306 - val_loss: 0.0684
Epoch 99/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 123ms/step - loss: 0.0314 - val_loss: 0.0397
Epoch 100/100
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 125ms/step - loss: 0.0301 - val_loss: 0.0338
```

In [81]:
```
y_pred = model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

print(f"MAE: {mae:.4f}")
print(f"MSE: {mse:.4f}")
print(f"RMSE: {rmse:.4f}")
```

```
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 110ms/step
MAE: 0.1451
MSE: 0.0338
RMSE: 0.1838
```

In [ ]:
```
plt.figure(figsize=(12, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

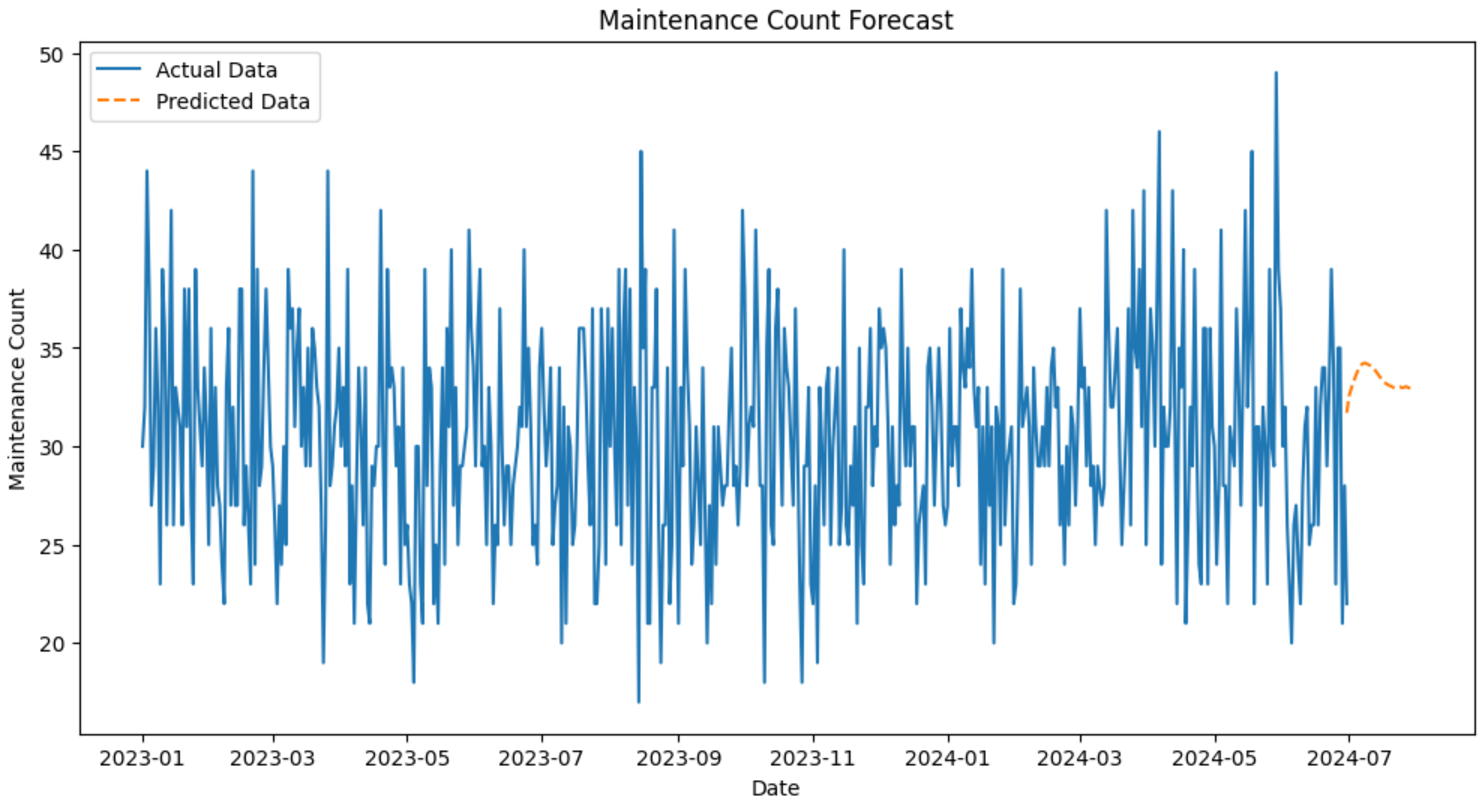


Forecast

In [ ]:
```
future_steps = 30
last_sequence = data[-seq_length:]
predictions = []

for _ in range(future_steps):
    seq = last_sequence[-seq_length:].reshape(1, seq_length, 1)
    pred = model.predict(seq)
    predictions.append(pred[0, 0])
    last_sequence = np.append(last_sequence, pred)[-seq_length:]

predictions = scaler.inverse_transform(np.array(predictions).reshape(-1, 1)).flatten()

last_timestamp = maintenance_count['timestamp'].iloc[-1]
future_dates = pd.date_range(start=maintenance_count['timestamp'].max(), periods=len(predictions), freq='D')

plt.figure(figsize=(12, 6))
plt.plot(maintenance_count['timestamp'], maintenance_count['maintenance_count'], label='Actual Data')
plt.plot(future_dates, predictions, label='Predicted Data', linestyle='dashed')
plt.title('Maintenance Count Forecast')
plt.xlabel('Date')
```

```
plt.ylabel('Maintenance Count')
plt.legend()
plt.show()
print(predictions)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
```




```
[31.7099   32.55255  32.925026 33.254402 33.574997 33.836937 34.051483
 34.19092  34.237267 34.202    34.146843 34.076782 33.991913 33.847572
 33.692383 33.52971  33.42052  33.27041  33.17199  33.10827  33.063965
 32.99848  32.97404  33.010216 33.028435 32.976273 33.00226  33.039146
 32.9794   32.95014 ]
```

In [ ]:

```
min_length = min(len(predictions), len(future_dates))
predictions = predictions[:min_length]
future_dates = future_dates[:min_length]

future_df = pd.DataFrame({"timestamp": future_dates, "predicted_maintenance": predictions})

future_df.to_csv("future_maintenance_predictions.csv", index=False)
print("CSV file saved successfully!")
```

```
CSV file saved successfully!
```

In [78]: