In [5]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [6]:
```
df = pd.read_csv("e11.csv")
```

#Data Preprocessing - By visualizing the raw data we identified coulumns with too many missing values (either NaN or #REF or #NA), at least more than half, and drop those columns. There columns are c189, c190, c199, c202, c204, c206, c207 - c223, c231, c232, c233, c234, c226

Data Preprocessing

In [7]:
```
columns_to_drop = ['c1', 'c188', 'c189', 'c190', 'c199', 'c202', 'c204', 'c206', 'c207', 'c208', 'c209', 'c210', 'c211', 'c212', 'c213', 'c214', 'c215', 'c216',
                   'c217', 'c218', 'c219', 'c220', 'c221', 'c222', 'c223', 'c231', 'c232', 'c233', 'c234', 'c226', 'c229']

df = df.drop(columns=columns_to_drop)
```

Next we take care of the cells which have some non numeric values like and convert them all to numeric so that they appear similar as missing values, so that its eaier to impute later.

In [8]:
```
df1 = df.apply(pd.to_numeric,errors='coerce')

```

Imputing NaN values using mean

In [9]:
```
df1 = df1.fillna(df1.mean())
```

In [10]:
```
df1.to_csv('df1.csv')
```

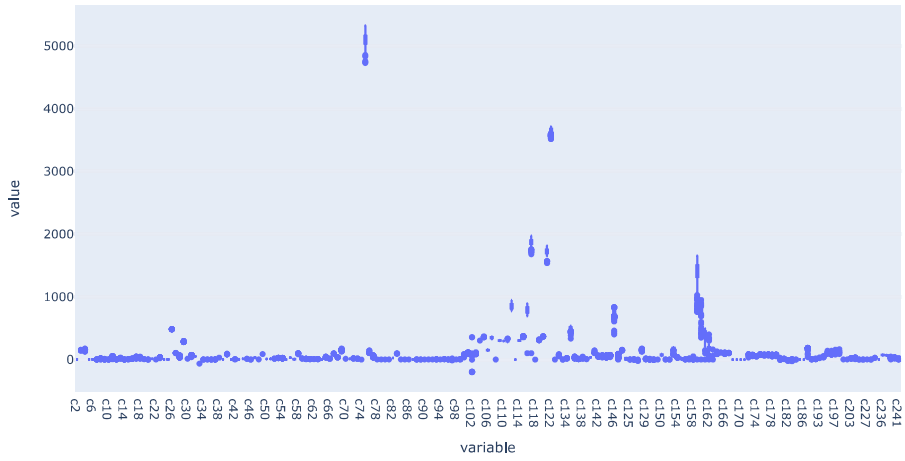
In [11]:
```
pip install plotly
```

```
Requirement already satisfied: plotly in c:\users\parwa\appdata\local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages\python311
\site-packages (5.17.0)
Requirement already satisfied: tenacity>=6.2.0 in c:\users\parwa\appdata\local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages\p
ython311\site-packages (from plotly) (8.2.3)
Requirement already satisfied: packaging in c:\users\parwa\appdata\local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages\python3
11\site-packages (from plotly) (23.1)
Note: you may need to restart the kernel to use updated packages.


[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: C:\Users\parwa\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\python.exe -m pip install --upgrade pip
```

In [12]:
```
import plotly.express as px
```

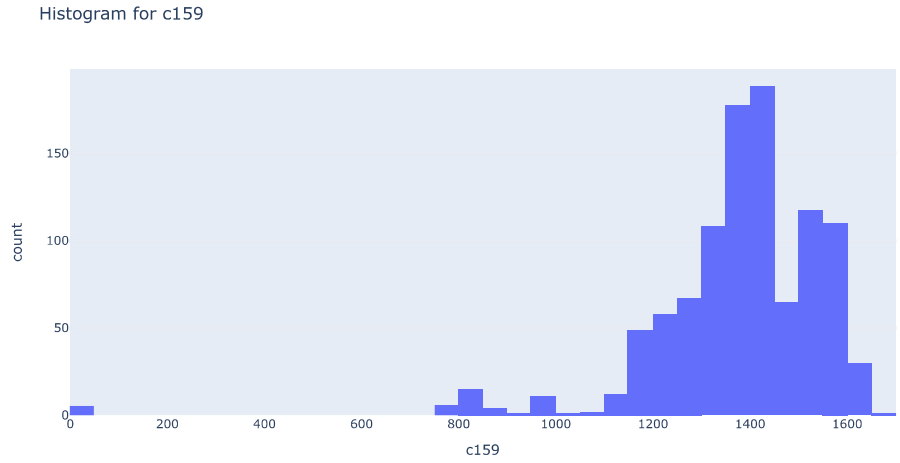
Using plotly to generate interactive box plots for all variables to visualise outliers

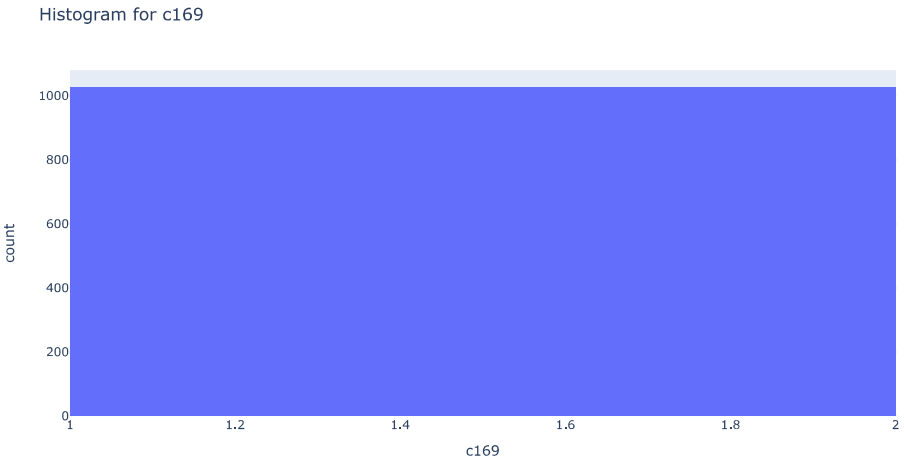
In [13]:
```
import warnings
warnings.filterwarnings("ignore")

fig = px.box(df1)

fig.show()
```



Visualising outliers for all variables using histogram

In [14]:
```
columns = df1.columns

# We created histograms for each column using a loop
'''for column in columns:
    histogram = px.histogram(df1, x=column, title=f'Histogram for {column}')
    histogram.show()'''
#But we are showing for some example of constant and non-constant columns
#only to not make this file full of histogram images
example_columns = ['c159','c169']
for column in example_columns:
    histogram = px.histogram(df1, x=column, title=f'Histogram for {column}')
    histogram.show()
```





After reviewing the histograms, we see there are few more vairables which need to be dropped as they have the same value across all observations and will not have any effect on model output. Note: None of these columns fall in the ambit of controllable parameters that is: c26, c27, c28, c29, c30, c31, c32, c33, c39, c139, c142, c143, c155, c156, c157, c158, c160, c161, c162, c163

In [15]:
```
constant_columns = ['c2', 'c82','c110', 'c168', 'c169', 'c170', 'c171']

df1 = df1.drop(columns=constant_columns)
```

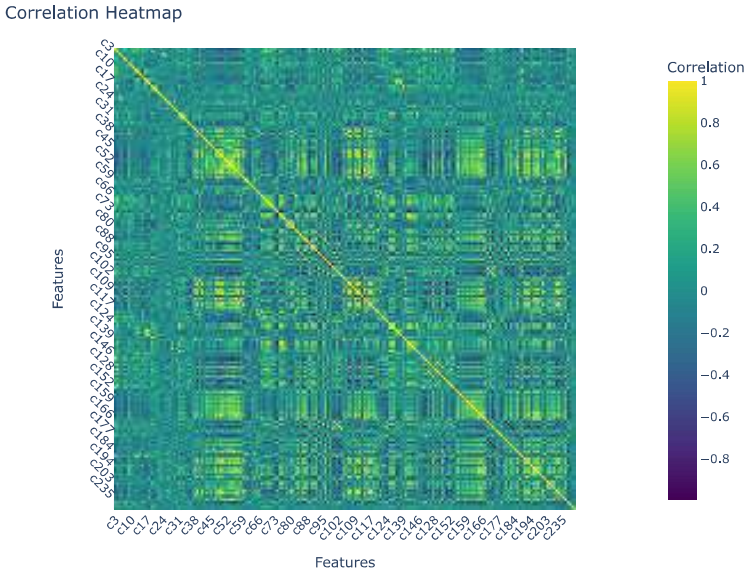
Defining and replacing outliers

In [16]:
```
def replace_outliers_with_mean(column):
    q1 = column.quantile(0.25)
    q3 = column.quantile(0.75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr

    # Replace outliers with the median value
    column = column.apply(lambda x: column.mean() if x < lower_bound or x > upper_bound else x)
    return column

# Iterate through each column in the DataFrame
for column in df1.columns:
    # Exclude target columns from outlier replacement
    if column not in ['c51', 'c52', 'c53', 'c54', 'c241']:
        # Replace outliers with the median value for each column
        df1[column] = replace_outliers_with_mean(df1[column])
```

Heat Map for visualizing correlations between variabels

In [17]:

```
correlation_matrix = df1.corr()

# Create a heatmap using Plotly Express
fig = px.imshow(correlation_matrix,
                labels=dict(x="Features", y="Features", color="Correlation"),
                x=correlation_matrix.columns,
                y=correlation_matrix.columns,
                color_continuous_scale="Viridis")

# Customize the layout
fig.update_layout(title="Correlation Heatmap",
                  xaxis=dict(tickangle=-45),
                  yaxis=dict(tickangle=45),
                  width=800, height=600)

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

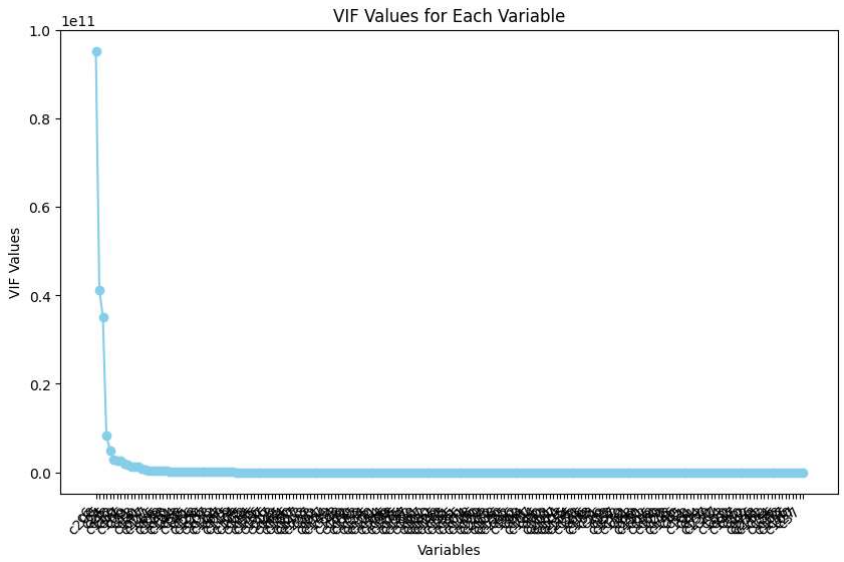


Now we check for multicolinearity using VIF

In [ ]:

```
from statsmodels.stats.outliers_influence import variance_inflation_factor

# Assuming df1 is your original DataFrame
# Define controllable_and_target_parameters
controllable_and_target_parameters = ['c26', 'c27', 'c28', 'c29', 'c30', 'c31', 'c32',
                                      'c33', 'c39', 'c139', 'c142', 'c143', 'c155', 'c156',
                                      'c157', 'c158', 'c160', 'c161', 'c162', 'c163', 'c51',
                                      'c52', 'c53', 'c54', 'c241']

# Calculate VIF for each variable in df1
vif_data = pd.DataFrame()
vif_data["Variable"] = df1.columns
vif_data["VIF"] = [variance_inflation_factor(df1.values, i) for i in range(df1.shape[1])]

# Display the VIF values
print("VIF Results:")
print(vif_data)

# Sort VIF values
vif_results_sorted = vif_data.sort_values(by='VIF', ascending=False)

# Plot the VIF values
plt.figure(figsize=(10, 6))
plt.plot(vif_results_sorted['Variable'], vif_results_sorted['VIF'], marker='o', linestyle='-', color='skyblue')
plt.xlabel('Variables')
plt.ylabel('VIF Values')
plt.title('VIF Values for Each Variable')
plt.xticks(rotation=45, ha='right')
plt.show()

# Set a threshold for high VIF
high_vif_threshold = 10
high_vif_features = vif_data[vif_data['VIF'] > high_vif_threshold]['Variable']

# Drop features with high VIF that are not in controllable_and_target_parameters
high_vif_features_filtered = high_vif_features[~high_vif_features.isin(controllable_and_target_parameters)]

# Create a new DataFrame df2 by dropping features with high VIF
df2 = df1.drop(columns=high_vif_features_filtered)

# Display the new DataFrame df2
print("\nDataFrame df2 with High VIF Features Dropped:")
print(df2)

```

```
VIF Results:
    Variable           VIF
0         c3  1.880862e+02
1         c4  1.356468e+07
2         c5  3.518368e+10
3         c6  9.532941e+10
4         c7  6.919079e+00
..       ...           ...
197     c236  1.008484e+02
198     c237  2.937045e+02
199     c238  5.514066e+00
200     c239  7.109210e+00
201     c241  5.449952e+00

[202 rows x 2 columns]
```



```
DataFrame df2 with High VIF Features Dropped:
            c7         c8       c10        c11        c12       c14       c20  \
0     2.264965  18.978886  0.719497  58.340911  10.824996  0.002228  7.120964
1     2.260011  20.188791  0.702348  58.340911  10.824996  0.002228  7.793413
2     2.085850  20.188791  0.656799  58.340911  10.824996  0.002228  7.289157
3     2.258069  20.188791  0.662394  58.340911   9.744850  0.011701  7.958076
4     2.252392  20.188791  0.689549  56.226017  10.457647  0.011701  8.757605
...        ...        ...       ...        ...        ...       ...       ...
1020  2.150980  19.448113  0.577761  57.962816  11.216774  0.002229  7.606814
1021  2.145125  19.449504  0.585410  57.752414  11.321437  0.002229  8.093012
1022  2.149443  19.411187  0.586067  57.625140  11.288059  0.002229  7.956216
1023  2.144927  19.426058  0.583313  57.386525  11.327831  0.002229  7.985720
1024  2.144609  19.457788  0.587881  57.289982  11.310388  0.002229  8.146586

            c21       c22         c26  ...      c158        c160        c161  \
0      9.257515  2.743170  493.796764  ...  23.75039  773.400976  370.726829
1      9.218110  2.596314  493.661889  ...  23.75039  773.400976  370.726829
2      9.599612  2.557701  495.644947  ...  23.75039  773.400976  370.726829
3      9.436385  2.897314  494.354041  ...  23.75039  773.400976  370.726829
4      9.954739  2.917772  492.051373  ...  23.75039  773.400976  370.726829
...         ...       ...         ...  ...       ...         ...         ...
1020  10.317679  6.734358  497.999661  ...  22.20000  800.000000  370.000000
1021  10.316755  6.646485  497.139686  ...  22.60000  800.000000  400.000000
1022  10.436322  6.575772  497.557435  ...  22.60000  800.000000  370.000000
1023  10.552277  6.480572  497.669483  ...  22.70000  800.000000  350.000000
1024  10.610720  6.097910  498.180745  ...  22.30000  800.000000  200.000000

            c162   c163       c176       c177       c238       c239      c241
0     150.000000   50.0  80.273187  83.780942  39.628026  42.811412  2.184083
1     150.000000   50.0  80.272263  83.780942  33.338342  42.915315  2.233879
2     150.000000   50.0  79.999906  83.780942  36.442830  42.981814  2.088296
3     150.000000   50.0  79.903734  83.780942  38.454676  43.168207  2.089270
4     150.000000   50.0  79.806441  83.780942  40.601255  43.445943  2.096676
...          ...    ...        ...        ...        ...        ...       ...
1020  160.000000   80.0  80.039638  83.752424  39.780049  38.551735  1.957681
1021  200.000000   55.0  80.019832  83.776712  38.035531  40.965092  1.986429
1022  200.000000   55.0  80.015796  83.790451  37.499035  41.278392  2.006031
1023  100.000000   50.0  80.024588  83.795744  37.503065  41.388672  2.020471
1024  167.482927  100.0  80.022341  83.796978  37.933377  41.791693  2.026799

[1025 rows x 53 columns]
```

In [ ]:
```
print("Features with High VIF:")
print(high_vif_features.tolist())
```

```
Features with High VIF:
['c3', 'c4', 'c5', 'c6', 'c9', 'c13', 'c15', 'c16', 'c17', 'c18', 'c19', 'c23', 'c24', 'c25', 'c26', 'c28', 'c29', 'c31', 'c32', 'c33', 'c38', 'c39', 'c40', 'c41', 'c
43', 'c46', 'c47', 'c48', 'c49', 'c50', 'c51', 'c52', 'c53', 'c54', 'c55', 'c56', 'c57', 'c58', 'c59', 'c60', 'c61', 'c62', 'c64', 'c65', 'c66', 'c67', 'c68', 'c69',
'c70', 'c71', 'c74', 'c75', 'c76', 'c77', 'c78', 'c79', 'c80', 'c81', 'c83', 'c84', 'c86', 'c87', 'c88', 'c89', 'c90', 'c91', 'c92', 'c93', 'c94', 'c95', 'c96', 'c9
7', 'c98', 'c99', 'c100', 'c101', 'c102', 'c103', 'c104', 'c105', 'c106', 'c107', 'c108', 'c109', 'c111', 'c112', 'c113', 'c114', 'c115', 'c116', 'c117', 'c118', 'c11
9', 'c120', 'c121', 'c122', 'c123', 'c124', 'c134', 'c135', 'c136', 'c138', 'c139', 'c140', 'c141', 'c142', 'c143', 'c144', 'c145', 'c148', 'c149', 'c125', 'c126', 'c
127', 'c128', 'c129', 'c130', 'c131', 'c132', 'c150', 'c151', 'c152', 'c153', 'c154', 'c155', 'c157', 'c158', 'c159', 'c164', 'c165', 'c166', 'c167', 'c172', 'c173',
'c174', 'c175', 'c178', 'c179', 'c180', 'c181', 'c182', 'c183', 'c184', 'c185', 'c186', 'c187', 'c191', 'c192', 'c193', 'c194', 'c195', 'c196', 'c197', 'c198', 'c20
0', 'c201', 'c203', 'c205', 'c224', 'c225', 'c227', 'c228', 'c230', 'c235', 'c236', 'c237']
```

Next we convert the 4 target variables to categorical - We use feature binning as a method and tranform these variables

Creating New variable 'overall_risk' which takes categorical values of SAFE, MODERATE, HIGH and CRITICAL depending on values in c51 to c54. For example if any value in all four columns is above 20 that data point will take value 'CRITICAL' in the overall risk column.

In [ ]:
```
import pandas as pd


# Create the 'overall_risk' column based on the maximum value across the target columns
df2['overall_risk'] = df2[['c51', 'c52', 'c53', 'c54']].max(axis=1)

# Map numerical values to risk categories
df2['overall_risk'] = pd.cut(df2['overall_risk'], bins=[-float('inf'), 5, 10, 20, float('inf')],
                             labels=['SAFE', 'MODERATE', 'HIGH', 'CRITICAL'], right=False)

# Print unique values for the 'overall_risk' column
unique_values = df2['overall_risk'].unique()
print(f"Unique values for overall_risk: {unique_values}")
```

```
Unique values for overall_risk: ['MODERATE', 'HIGH', 'CRITICAL']
Categories (4, object): ['SAFE' < 'MODERATE' < 'HIGH' < 'CRITICAL']
```

Label encoding Target Variable 'overall_risk' with class mapping

In [ ]:
```
from sklearn.preprocessing import LabelEncoder
selected_columns = ['overall_risk']

# Initialize the LabelEncoder for each selected column
label_encoders = {}
for column in selected_columns:
    label_encoders[column] = LabelEncoder()
    df1[column] = label_encoders[column].fit_transform(df2[column])
```

Importing necessary libraries

In [ ]:
```
from sklearn.model_selection import train_test_split, cross_val_score, KFold, GridSearchCV
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import MinMaxScaler
```

Task 1: Implementing ML to predict when vibrations reach 'High' or 'Critical' Values

Creating X and y, label encoding y, splitting data into train and test

In [ ]:
```

X_initial = df1.drop(['c51', 'c52', 'c53', 'c54', 'overall_risk'], axis=1)
y = df2['overall_risk']

# Split the data into training and testing sets for the initial model
X_train_initial, X_test_initial, y_train, y_test = train_test_split(X_initial, y, test_size=0.2, random_state=42)
```

Since the data has many variables acorss various ranges we will normalize the data using Min-Max Scaling

In [ ]:
```
scaler = MinMaxScaler()
X_train_initial_scaled = scaler.fit_transform(X_train_initial)
X_test_initial_scaled = scaler.transform(X_test_initial)
```

We then train our data on a simple Random Forest Classifier before dropping any features which may have multicolinearity.

In [ ]:
```
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix

lr = LogisticRegression(max_iter=1000, solver='liblinear')

lr.fit(X_train_initial, y_train)

# Initial model evaluation
y_pred_initial_lr = lr.predict(X_test_initial)
accuracy_initial = accuracy_score(y_test, y_pred_initial_lr)
print(f"Initial Model Accuracy: {accuracy_initial}")

# Confusion matrix
conf_matrix_initial_lr = confusion_matrix(y_test, y_pred_initial_lr)

# Print the confusion matrix
print("Confusion Matrix:")
print(conf_matrix_initial_lr)

# Print classification report for additional metrics
print("Classification Report:")
print(classification_report(y_test, y_pred_initial_lr))

```

```
Initial Model Accuracy: 0.9414634146341463
Confusion Matrix:
[[19  0  0]
 [ 1 88  5]
 [ 0  6 86]]
Classification Report:
              precision    recall  f1-score   support

    CRITICAL       0.95      1.00      0.97        19
        HIGH       0.94      0.94      0.94        94
    MODERATE       0.95      0.93      0.94        92

    accuracy                           0.94       205
   macro avg       0.94      0.96      0.95       205
weighted avg       0.94      0.94      0.94       205
```

Retraining mlr model with limited features using df2 which has no feature with high vif

In [ ]:
```
X_filtered = df2.drop(['c51', 'c52', 'c53', 'c54', 'overall_risk'], axis=1)

# Split the data into training and testing sets for the initial model
X_train_filtered, X_test_filtered = train_test_split(X_filtered, test_size=0.2, random_state=42)

scaler = MinMaxScaler()
X_train_filtered_scaled = scaler.fit_transform(X_train_filtered)
X_test_filtered_scaled = scaler.transform(X_test_filtered)

lr.fit(X_train_filtered, y_train)

# Initial model evaluation
y_pred_filtered_lr = lr.predict(X_test_filtered)
accuracy_filtered = accuracy_score(y_test, y_pred_filtered_lr)
print(f"New Model Accuracy: {accuracy_filtered}")

# Confusion matrix
conf_matrix_filtered_lr = confusion_matrix(y_test, y_pred_filtered_lr)

# Print the confusion matrix
print("Confusion Matrix:")
print(conf_matrix_filtered_lr)

# Print classification report for additional metrics
print("Classification Report:")
print(classification_report(y_test, y_pred_filtered_lr))

```

```
New Model Accuracy: 0.9121951219512195
Confusion Matrix:
[[19  0  0]
 [ 1 80 13]
 [ 0  4 88]]
Classification Report:
              precision    recall  f1-score   support

    CRITICAL       0.95      1.00      0.97        19
        HIGH       0.95      0.85      0.90        94
    MODERATE       0.87      0.96      0.91        92

    accuracy                           0.91       205
   macro avg       0.92      0.94      0.93       205
weighted avg       0.92      0.91      0.91       205
```

Alternative Method: There are some machine learnign models which take care of the multicolinearity of the features by using certain methods like regularization and can give good results with the right hyper parameter tuning even without identifying things like VIF. We demonstrate the use of one such ML model which Support Vector Classifier

In [ ]:
```
from sklearn.svm import SVC
param_grid = {
    'C': [0.001, 0.01, 0.1],  # Regularization parameter
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],  # Kernel type
    'gamma': ['scale', 'auto', 0.1, 0.01, 0.001],  # Kernel coefficient
}


# Create an SVM classifier
svm = SVC()


# Define the number of folds
num_folds = 10

# Create a cross-validation object (KFold)
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)

# Create a GridSearchCV object with the SVM model and parameter grid
grid_search = GridSearchCV(svm, param_grid, cv=kf)

# Fit the GridSearchCV object on your data
grid_search.fit(X_train_initial_scaled, y_train)

# Print the best parameters and the corresponding mean cross-validated score
print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

# Get the best parameters and the corresponding accuracy
best_params = grid_search.best_params_
best_accuracy = grid_search.best_score_

print("Best Parameters:", best_params)
print("Best Accuracy:", best_accuracy)

# Evaluate the model on the test data using the best parameters
best_svm= grid_search.best_estimator_
test_accuracy_svm = best_svm.score(X_test_initial_scaled, y_test)
print("Test Accuracy SVM:", test_accuracy_svm)
```

```
Best Parameters: {'C': 0.1, 'gamma': 'scale', 'kernel': 'poly'}
Best Score: 0.9597560975609755
Best Parameters: {'C': 0.1, 'gamma': 'scale', 'kernel': 'poly'}
Best Accuracy: 0.9597560975609755
Test Accuracy SVM: 0.975609756097561
```

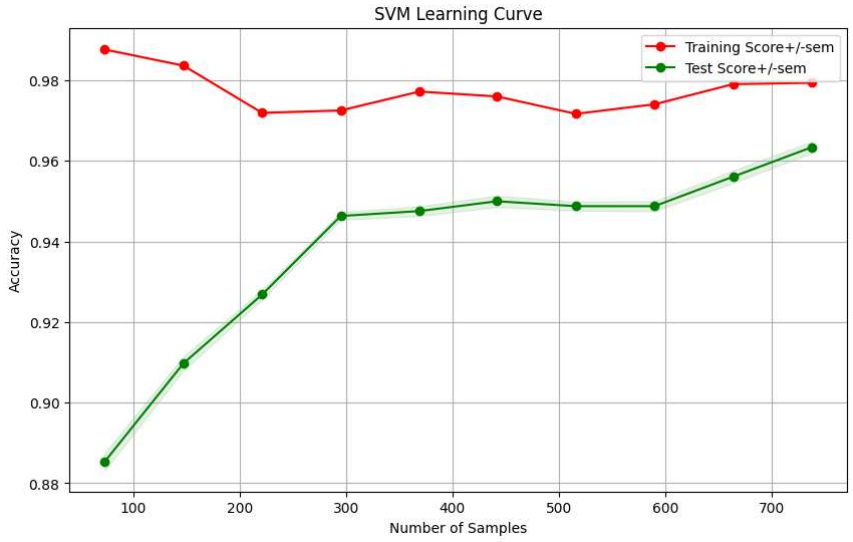
In [ ]:
```
from sklearn.metrics import confusion_matrix

y_pred_svm = best_svm.predict(X_test_initial_scaled)

# Confusion matrix
conf_matrix_svm = confusion_matrix(y_test, y_pred_svm)

# Print the confusion matrix
print("Confusion Matrix SVM:")
print(conf_matrix_svm)

# Print classification report for additional metrics
print("Classification Report SVM:")
print(classification_report(y_test, y_pred_svm))
```

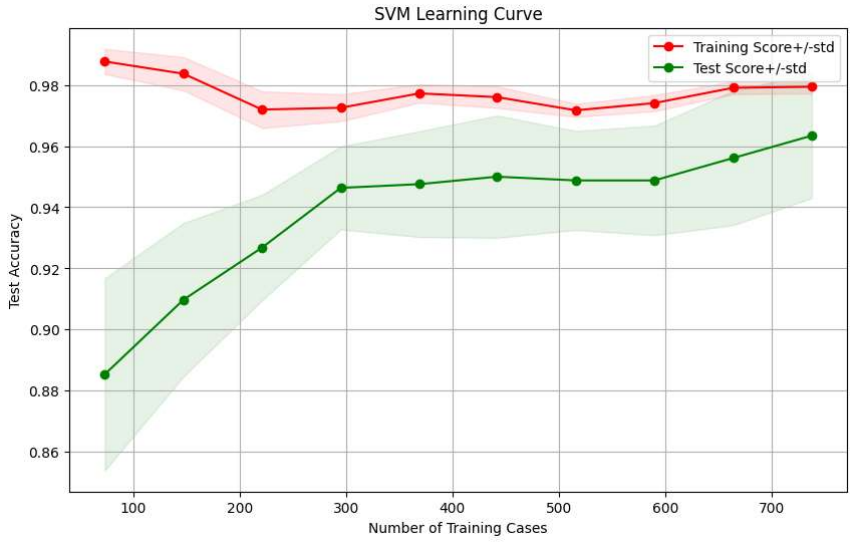
```
Confusion Matrix SVM:
[[19  0  0]
 [ 1 91  2]
 [ 0  2 90]]
Classification Report SVM:
              precision    recall  f1-score   support

    CRITICAL       0.95      1.00      0.97        19
        HIGH       0.98      0.97      0.97        94
    MODERATE       0.98      0.98      0.98        92

    accuracy                           0.98       205
   macro avg       0.97      0.98      0.98       205
weighted avg       0.98      0.98      0.98       205
```

As we can see SVM beats the metrics of the base model we used which is logistic regression. We also go on to plot the learnign curves for SVM

In [ ]:

```
from sklearn.model_selection import learning_curve

# Generate a learning curve
train_sizes, train_scores, test_scores = learning_curve(
    best_svm, X_train_initial_scaled, y_train, cv=10, scoring='accuracy', train_sizes=np.linspace(0.1, 1.0, 10))

# Calculate mean and standard deviation of training and test scores
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)


test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
train_sem = train_scores_std/np.sqrt(X_train_initial_scaled.shape[0])
test_sem = test_scores_std/np.sqrt(X_test_initial_scaled.shape[0])
# Plot the learning curve
plt.figure(figsize=(10, 6))
plt.title('SVM Learning Curve')
plt.xlabel('Number of Samples')
plt.ylabel('Accuracy')
plt.grid()

plt.fill_between(train_sizes, train_scores_mean - train_sem, train_scores_mean + train_sem, alpha=0.1, color="r")
plt.fill_between(train_sizes, test_scores_mean - test_sem, test_scores_mean + test_sem, alpha=0.1, color="g")

plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training Score+/-sem")
plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Test Score+/-sem")

plt.legend(loc="best")
plt.show()

plt.figure(figsize=(10, 6))
plt.title('SVM Learning Curve')
plt.xlabel('Number of Training Cases')
plt.ylabel('Test Accuracy')
plt.grid()

plt.fill_between(train_sizes, train_scores_mean - train_scores_std, train_scores_mean + train_scores_std, alpha=0.1, color="r")
plt.fill_between(train_sizes, test_scores_mean - test_scores_std, test_scores_mean + test_scores_std, alpha=0.1, color="g")

plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training Score+/-std")
plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Test Score+/-std")

plt.legend(loc="best")
plt.show()
```



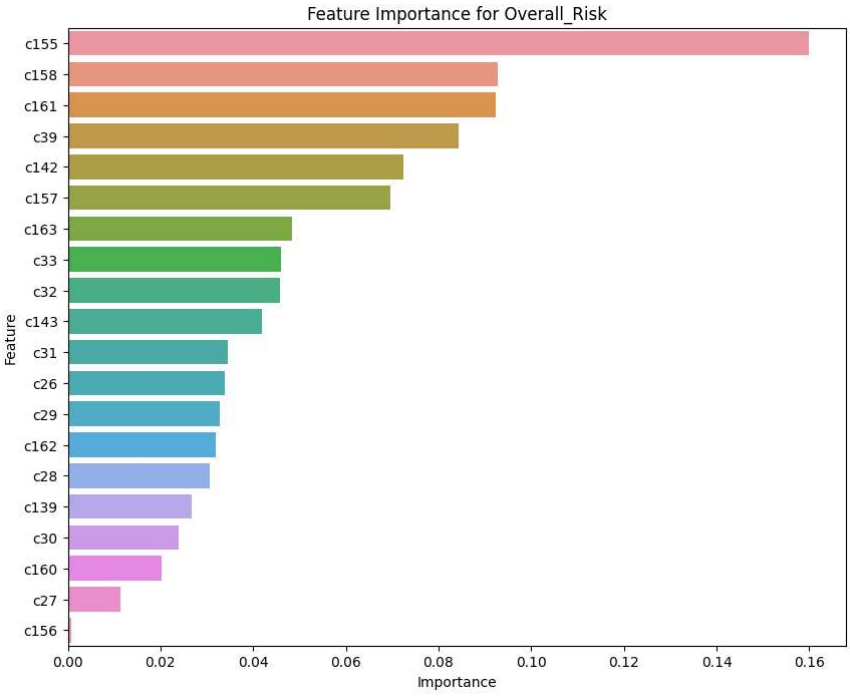


Next task is to create ML models to predict and control these vibrations using only the 'controllable parameters'. It is proposed to create an automated vibration control and reduction system that gets activated when the vibrationsnreach high and critical levels. One of the goals of this model should be to create a list of the most important parameters to change to reduce vibrations. This list should be in descending order of importance. This can be done by **two approaches**, in the **first we will use a ML model which gives us inbuilt feature importance such as Radnom Forest**. In the second method we use a **model agnostic method - SHapeley Additive exPlanations (SHAP values) as a method to identify class wise feature importance**.

In [ ]:
```
controllable_parameters = ['c26', 'c27', 'c28', 'c29', 'c30', 'c31', 'c32',
'c33', 'c39', 'c139', 'c142', 'c143', 'c155', 'c156', 'c157', 'c158', 'c160', 'c161', 'c162', 'c163']
```

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns

controllable_parameters = ['c26', 'c27', 'c28', 'c29', 'c30', 'c31', 'c32', 'c33', 'c39', 'c139', 'c142', 'c143', 'c155', 'c156', 'c157', 'c158', 'c160', 'c161', '
X_controllable = df1[controllable_parameters]
y = df1['overall_risk']

# Split the data into training and testing sets
X_train_rf, X_test_rf, y_train_rf, y_test_rf = train_test_split(X_controllable, y, test_size=0.2, random_state=42)

X_train_rf_scaled = scaler.fit_transform(X_train_rf)
X_test_rf_scaled = scaler.transform(X_test_rf)

# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the classifier
rf_classifier.fit(X_train_rf_scaled, y_train)

# Predictions on the test set
y_pred_rf = rf_classifier.predict(X_test_rf_scaled)

# Evaluate the model
print("Accuracy RF:", accuracy_score(y_test, y_pred_rf))
print("Classification Report RF:")
print(classification_report(y_test, y_pred_rf))

# Feature importance
feature_importance = rf_classifier.feature_importances_

# Create a DataFrame to visualize feature importance
feature_importance_df = pd.DataFrame({'Feature': controllable_parameters, 'Importance': feature_importance})
feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=False)

# Plot feature importance
plt.figure(figsize=(10, 8))
sns.barplot(x='Importance', y='Feature', data=feature_importance_df)
plt.title('Feature Importance for Overall_Risk')
plt.show()

```

```
Accuracy RF: 0.9560975609756097
Classification Report RF:
              precision    recall  f1-score   support

    CRITICAL       0.95      1.00      0.97        19
        HIGH       0.95      0.96      0.95        94
    MODERATE       0.97      0.95      0.96        92

    accuracy                           0.96       205
   macro avg       0.95      0.97      0.96       205
weighted avg       0.96      0.96      0.96       205
```



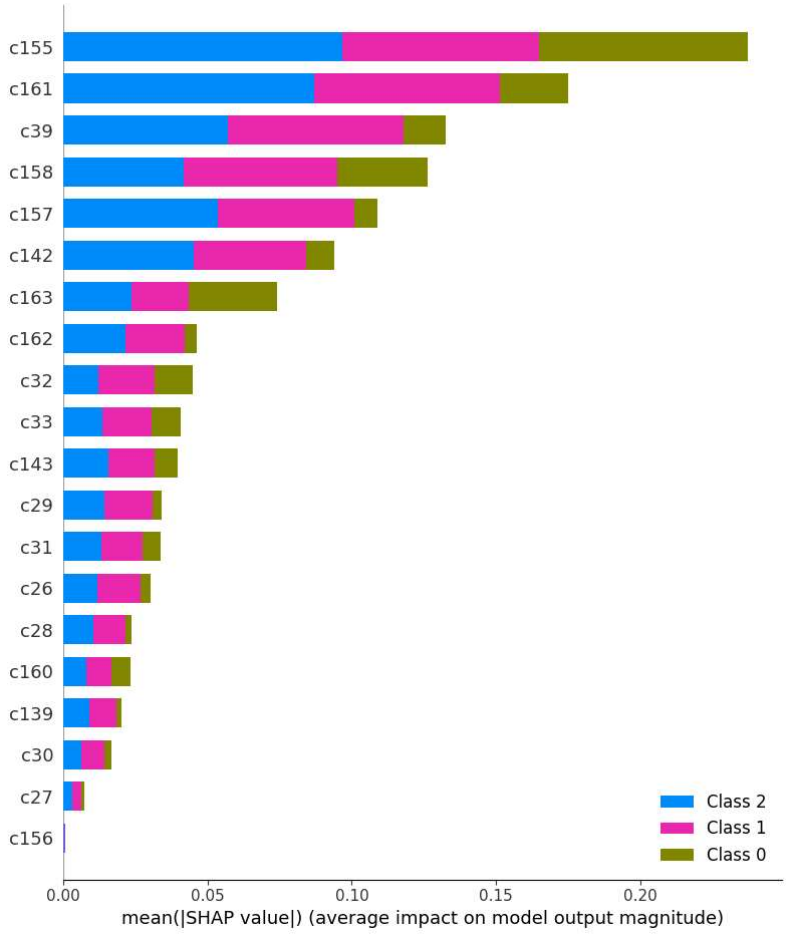
In [ ]:
```
feature_importance_df.to_csv('feature_importance_vibrations.csv')
```

In [ ]:
```
pip install shap
```

```
Requirement already satisfied: shap in /usr/local/lib/python3.10/dist-packages (0.43.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from shap) (1.23.5)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from shap) (1.11.3)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (from shap) (1.2.2)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from shap) (1.5.3)
Requirement already satisfied: tqdm>=4.27.0 in /usr/local/lib/python3.10/dist-packages (from shap) (4.66.1)
Requirement already satisfied: packaging>20.9 in /usr/local/lib/python3.10/dist-packages (from shap) (23.2)
Requirement already satisfied: slicer==0.0.7 in /usr/local/lib/python3.10/dist-packages (from shap) (0.0.7)
Requirement already satisfied: numba in /usr/local/lib/python3.10/dist-packages (from shap) (0.58.1)
Requirement already satisfied: cloudpickle in /usr/local/lib/python3.10/dist-packages (from shap) (2.2.1)
Requirement already satisfied: llvmlite<0.42,>=0.41.0dev0 in /usr/local/lib/python3.10/dist-packages (from numba->shap) (0.41.1)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas->shap) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->shap) (2023.3.post1)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn->shap) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn->shap) (3.2.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pandas->shap) (1.16.0)
```

Implementing SHAP

In [ ]:
```
import tensorflow as tf

import shap
explainer = shap.TreeExplainer(rf_classifier)

# Calculate SHAP values for the test set
shap_values = explainer.shap_values(X_test_rf_scaled)

# Plot the summary plot
shap.summary_plot(shap_values, X_test_rf_scaled, feature_names=controllable_parameters, plot_type='bar')
```



Task 2: C241 as Target Variable

In [ ]:
```
columns_to_drop = ['c241', 'overall_risk']
X_c241 = df2.drop(columns=columns_to_drop)


#Matrics of depenent variable
y_c241 = df2['c241']
```

Task 2, part 1: ML prediction model to understand which parameters (operating + controllable) significantly contribute to the 'specific energy'

In [ ]:
```
X_c241_normalized = scaler.fit_transform(X_c241)

# Convert the result back to a DataFrame (optional)
X_c241_normalized_df = pd.DataFrame(X_c241_normalized, columns=X_c241.columns)
```

Implementing mlr model

In [ ]:
```
import statsmodels.api as sm
X_var_c241 = sm.add_constant(X_c241_normalized_df)
```

In [ ]:
```
1
2 mlr_model = sm.OLS(y_c241,X_var_c241).fit()
3 mlr_model.summary()
```

Out[34]:

OLS Regression Results

Dep. Variable:	c241	**R-squared:**	0.258
Model:	OLS	**Adj. R-squared:**	0.218
Method:	Least Squares	**F-statistic:**	6.494
Date:	Sun, 12 Nov 2023	**Prob (F-statistic):**	4.70e-36
Time:	10:45:45	**Log-Likelihood:**	-981.61
No. Observations:	1025	**AIC:**	2069.
Df Residuals:	972	**BIC:**	2331.
Df Model:	52		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	4.0180	0.487	8.242	0.000	3.061	4.975

In [ ]:
```
1 #Creating a dataframe of p values
2 df_pval= pd.DataFrame(mlr_model.pvalues)
3 df_pval.columns = ['p_val']
4 df_pval.head()
```

Out[35]:

	p_val
const	5.431615e-16
c7	4.415969e-04
c8	7.276615e-01
c10	6.162898e-01
c11	5.170322e-01

Determining Features with significant p value

In [ ]:
```
1 #Identifying which columns have pval>0.05
2 condition = df_pval['p_val']<0.05
3 df_imp = df_pval[condition]
4 df_imp= df_imp.sort_values(by= 'p_val', ascending = True)
```

In [ ]:
```
1 df_imp
```

Out[37]:

	p_val
const	5.431615e-16
c155	3.699908e-14
c39	1.810322e-09
c22	6.204579e-05
c143	1.191433e-04
c139	3.987530e-04
c7	4.415969e-04
c42	8.409049e-04
c137	9.382014e-04
c72	2.067982e-03
c26	2.498036e-03
c147	4.441144e-03
c21	5.471427e-03
c14	1.016282e-02
c157	1.216262e-02
c158	1.468552e-02
c73	2.464412e-02
c20	2.534467e-02
c28	3.119151e-02

In [ ]:
```
1 df_imp.to_csv('Pval_sort.csv')
```

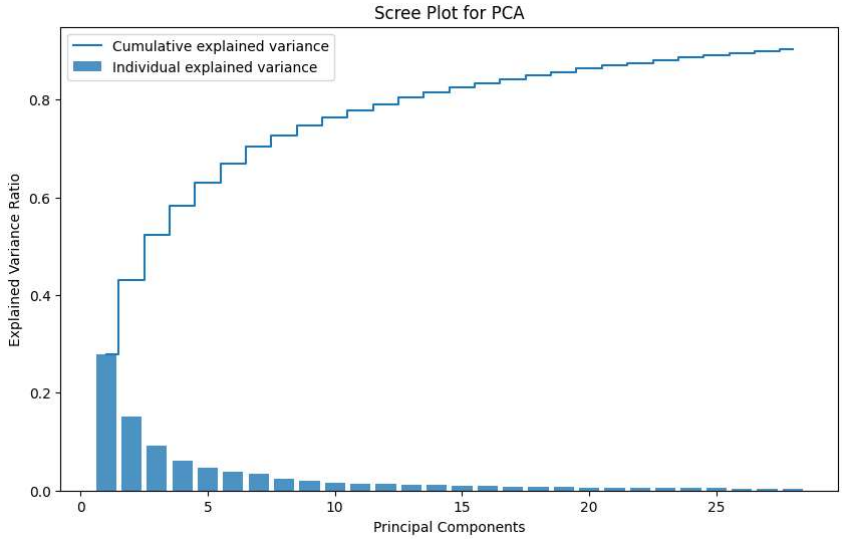
Task 2, part 2: find out the minimum number of ‘independent’ variables that can be used to ‘only predict – not control’ the specific energy consumption

Attempt 1 - Use PCA We decided to use PCA on the original dataset which ha features with High VIF also as PCA itself will take care of any multicolinearity

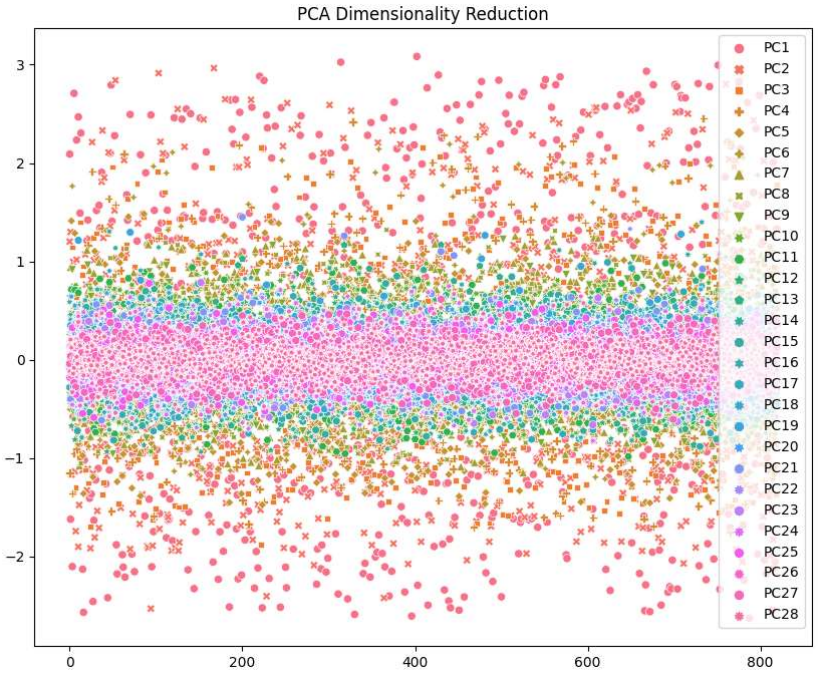
In [ ]:

```
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score

columns_to_drop = ['c241', 'overall_risk']
X_c241_new = df1.drop(columns=columns_to_drop)

#Matrics of depenent variable
y_c241 = df1['c241']


X_train_241, X_test_241, y_train_241, y_test_241 = train_test_split(X_c241_new, y_c241, test_size=0.2, random_state=42)

X_train_241_scaled = scaler.fit_transform(X_train_241)
X_test_241_scaled = scaler.transform(X_test_241)

# Apply PCA only on the training data and determine the number of components
desired_variance = 0.9  # Set the desired explained variance
pca = PCA(n_components=desired_variance)
X_train_pca = pca.fit_transform(X_train_241_scaled)
X_test_pca = pca.transform(X_test_241_scaled)

# Get the number of components explaining the desired variance
num_components = pca.n_components_
print(f"Number of components to explain {desired_variance * 100}% of the variance: {num_components}")

explained_variance_ratio = pca.explained_variance_ratio_

# Cumulative explained variance
cumulative_variance = explained_variance_ratio.cumsum()

# Plot the scree plot
plt.figure(figsize=(10, 6))
plt.bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio, alpha=0.8, align='center', label='Individual explained variance')
plt.step(range(1, len(explained_variance_ratio) + 1), cumulative_variance, where='mid', label='Cumulative explained variance')
plt.xlabel('Principal Components')
plt.ylabel('Explained Variance Ratio')
plt.title('Scree Plot for PCA')
plt.legend()
plt.show()

```

```
Number of components to explain 90.0% of the variance: 28
```



In [ ]:
```
X_train_pca_df = pd.DataFrame(X_train_pca, columns=[f'PC{i+1}' for i in range(num_components)])

# Plot the PCA results using Seaborn
plt.figure(figsize=(10, 8))
sns.scatterplot(data=X_train_pca_df)
plt.title('PCA Dimensionality Reduction')
plt.legend(loc='best')
plt.show()
```



ML using only Principal Components

In [ ]:
```
from sklearn.ensemble import RandomForestRegressor
model_rf = RandomForestRegressor()
model_rf.fit(X_train_pca, y_train_241)

# Make predictions on the training and test sets
y_train_pca_pred_rf = model_rf.predict(X_train_pca)
y_test_pca_pred_rf = model_rf.predict(X_test_pca)

# Evaluate the model
mse_train_pca_rf = mean_squared_error(y_train_241, y_train_pca_pred_rf)
r2_train_pca_rf = r2_score(y_train_241, y_train_pca_pred_rf)

mse_test_pca_rf = mean_squared_error(y_test_241, y_test_pca_pred_rf)
r2_test_pca_rf = r2_score(y_test_241, y_test_pca_pred_rf)

# Print the results
print("Training MSE PCA RF:", mse_train_pca_rf)
print("Training R2 PCA RF:", r2_train_pca_rf)
print("\nTesting MSE PCA RF:", mse_test_pca_rf)
print("Testing R2 PCA RF:", r2_test_pca_rf)
```

```
Training MSE PCA RF: 0.092787369045453
Training R2 PCA RF: 0.8606748310407734

Testing MSE PCA RF: 0.0019393442486813042
Testing R2 PCA RF: 0.8471603710508758
```

Hence we determine the minimum number of independent features to predict c241 is 28. Using PCA we can ensure that these are orthogonal features that is they are independent of each other, and since we require to only predict and not control it is not essential to determine exactly which features, so loosing explainability with PCA is also not a problem