Material Selection for Sustainable Electric Vehicle Chassis Design

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
dataset = pd.read_csv("Data.csv")
dataset
```

Out[2]:

	Std	ID	Material	Heat treatment	Su	Sy	A5	Bhn	E	G	mu	Ro	pH	Desc	HV
0	ANSI	D8894772B88F495093C43AF905AB6373	Steel SAE 1015	as-rolled	421	314	39.0	126.0	207000	79000	0.30	7860	NaN	NaN	NaN
1	ANSI	05982AC66F064F9EBC709E7A4164613A	Steel SAE 1015	normalized	424	324	37.0	121.0	207000	79000	0.30	7860	NaN	NaN	NaN
2	ANSI	356D6E63FF9A49A3AB23BF66BAC85DC3	Steel SAE 1015	annealed	386	284	37.0	111.0	207000	79000	0.30	7860	NaN	NaN	NaN
3	ANSI	1C758F8714AC4E0D9BD8D8AE1625AECD	Steel SAE 1020	as-rolled	448	331	36.0	143.0	207000	79000	0.30	7860	NaN	NaN	NaN
4	ANSI	DCE10036FC1946FC8C9108D598D116AD	Steel SAE 1020	normalized	441	346	35.8	131.0	207000	79000	0.30	7860	550.0	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1547	JIS	512A80EC21EA416BA2725B38BA8096EF	Nodular cast iron	NaN	600	370	NaN	NaN	169000	70000	0.20	7160	480.0	Nodular cast iron	210.0
1548	JIS	38526441BA8741CA979DBF870D0B8A9B	Nodular cast iron	NaN	700	420	NaN	NaN	169000	70000	0.20	7160	560.0	Nodular cast iron	230.0
1549	JIS	CAC03D7EB1AA45E68EFF92A2EF4C3D9B	Nodular cast iron	NaN	800	480	NaN	NaN	169000	70000	0.20	7160	600.0	Nodular cast iron	240.0
1550	JIS	45C82A36EC644F8BB6170A99ED819B62	Malleable cast iron	NaN	400	180	4.0	NaN	160000	64000	0.27	7160	300.0	Malleable cast iron	220.0
1551	JIS	BC74F870412F4DDBADDEF1063C1C079F	Malleable cast iron	NaN	500	260	4.0	NaN	160000	64000	0.27	7160	370.0	Malleable cast iron	230.0

1552 rows × 15 columns

In [3]:
```
d2 = pd.read_csv("Data.csv")

#Merge the columns - 'Std', 'Material' and 'Heat treatment' under a new column named - 'Material'
d2['Material'] = dataset[['Std', 'Material', 'Heat treatment']].fillna('').agg(' '.join, axis=1)

d2.drop(['Std', 'Heat treatment'], axis=1, inplace=True)
d2
```

Out[3]:

	ID	Material	Su	Sy	A5	Bhn	E	G	mu	Ro	pH	Desc	HV
0	D8894772B88F495093C43AF905AB6373	ANSI Steel SAE 1015 as-rolled	421	314	39.0	126.0	207000	79000	0.30	7860	NaN	NaN	NaN
1	05982AC66F064F9EBC709E7A4164613A	ANSI Steel SAE 1015 normalized	424	324	37.0	121.0	207000	79000	0.30	7860	NaN	NaN	NaN
2	356D6E63FF9A49A3AB23BF66BAC85DC3	ANSI Steel SAE 1015 annealed	386	284	37.0	111.0	207000	79000	0.30	7860	NaN	NaN	NaN
3	1C758F8714AC4E0D9BD8D8AE1625AECD	ANSI Steel SAE 1020 as-rolled	448	331	36.0	143.0	207000	79000	0.30	7860	NaN	NaN	NaN
4	DCE10036FC1946FC8C9108D598D116AD	ANSI Steel SAE 1020 normalized	441	346	35.8	131.0	207000	79000	0.30	7860	550.0	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...
1547	512A80EC21EA416BA2725B38BA8096EF	JIS Nodular cast iron	600	370	NaN	NaN	169000	70000	0.20	7160	480.0	Nodular cast iron	210.0
1548	38526441BA8741CA979DBF870D0B8A9B	JIS Nodular cast iron	700	420	NaN	NaN	169000	70000	0.20	7160	560.0	Nodular cast iron	230.0
1549	CAC03D7EB1AA45E68EFF92A2EF4C3D9B	JIS Nodular cast iron	800	480	NaN	NaN	169000	70000	0.20	7160	600.0	Nodular cast iron	240.0
1550	45C82A36EC644F8BB6170A99ED819B62	JIS Malleable cast iron	400	180	4.0	NaN	160000	64000	0.27	7160	300.0	Malleable cast iron	220.0
1551	BC74F870412F4DDBADDEF1063C1C079F	JIS Malleable cast iron	500	260	4.0	NaN	160000	64000	0.27	7160	370.0	Malleable cast iron	230.0

1552 rows × 13 columns

In [4]:
```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1552 entries, 0 to 1551
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Std             1552 non-null   object 
 1   ID              1552 non-null   object 
 2   Material        1552 non-null   object 
 3   Heat treatment  802 non-null    object 
 4   Su              1552 non-null   int64  
 5   Sy              1552 non-null   object 
 6   A5              1346 non-null   float64
 7   Bhn             463 non-null    float64
 8   E               1552 non-null   int64  
 9   G               1552 non-null   int64  
 10  mu              1552 non-null   float64
 11  Ro              1552 non-null   int64  
 12  pH              193 non-null    float64
 13  Desc            981 non-null    object 
 14  HV              165 non-null    float64
dtypes: float64(5), int64(4), object(6)
memory usage: 182.0+ KB
```

In [5]:
```
#Check for missing values
dataset.isnull().sum()
```

Out[5]:
```
Std                 0
ID                  0
Material            0
Heat treatment    750
Su                  0
Sy                  0
A5                206
Bhn              1089
E                   0
G                   0
mu                  0
Ro                  0
pH               1359
Desc              571
HV               1387
dtype: int64
```

In [6]:
```
# Drop columns with missing values: A5, Bhn, pH, Desc, HV
d2.drop(['A5', 'Bhn', 'pH', 'Desc', 'HV'], axis=1, inplace=True)
d2
```

Out[6]:

	ID	Material	Su	Sy	E	G	mu	Ro
0	D8894772B88F495093C43AF905AB6373	ANSI Steel SAE 1015 as-rolled	421	314	207000	79000	0.30	7860
1	05982AC66F064F9EBC709E7A4164613A	ANSI Steel SAE 1015 normalized	424	324	207000	79000	0.30	7860
2	356D6E63FF9A49A3AB23BF66BAC85DC3	ANSI Steel SAE 1015 annealed	386	284	207000	79000	0.30	7860
3	1C758F8714AC4E0D9BD8D8AE1625AECD	ANSI Steel SAE 1020 as-rolled	448	331	207000	79000	0.30	7860
4	DCE10036FC1946FC8C9108D598D116AD	ANSI Steel SAE 1020 normalized	441	346	207000	79000	0.30	7860
...	...	...	...	...	...	...	...	...
1547	512A80EC21EA416BA2725B38BA8096EF	JIS Nodular cast iron	600	370	169000	70000	0.20	7160
1548	38526441BA8741CA979DBF870D0B8A9B	JIS Nodular cast iron	700	420	169000	70000	0.20	7160
1549	CAC03D7EB1AA45E68EFF92A2EF4C3D9B	JIS Nodular cast iron	800	480	169000	70000	0.20	7160
1550	45C82A36EC644F8BB6170A99ED819B62	JIS Malleable cast iron	400	180	160000	64000	0.27	7160
1551	BC74F870412F4DDBADDEF1063C1C079F	JIS Malleable cast iron	500	260	160000	64000	0.27	7160

1552 rows × 8 columns

In [7]:
```
# Check for Non-Numeric Values
print(d2.dtypes)  # Check data types

# Check if there are any non-numeric values
print(f"Unique values in {'Sy'}:")
print(d2['Sy'].unique())
```

```
ID          object
Material    object
Su           int64
Sy          object
E            int64
G            int64
mu         float64
Ro           int64
dtype: object
Unique values in Sy:
['314' '324' '284' '331' '346' '295' '359' '317' '345' '341' '648' '414'
 '374' '353' '593' '427' '365' '724' '483' '421' '372' '779' '586' '524'
 '376' '979' '572' '500' '379' '827' '305' '303' '279' '316' '319' '396'
 '938' '405' '1213' '400' '627' '558' '436' '1593' '600' '422' '758'
 '1662' '360' '1462' '655' '417' '1641' '734' '1724' '464' '425' '862'
 '472' '1675' '366' '484' '1407' '1517' '293' '529' '357' '1731' '531'
 '276' '1793' '615' '412' '1689' '385' '429' '1503' '1669' '688' '386'
 '1551' '607' '415' '1655' '579' '486' '2048' '571' '440' '517' '931'
 '965' '255' '241' '207' '1000' '1034' '1344' '310' '1069' '448' '1896'
 '138' '172' '538' '455' '552' '62' '248' '469' '69' '117' '110' '97' '90'
 '83' '159' '124' '193' '290' '262' '165' '221' '145' '152' '179' '186'
 '234' '41' '48' '76' '103' '131' '228' '269' '338' '200' '28' '34' '296'
 '393' '352' '55' '214' '407' '283' '434' '490' '503' '462' '230' '225'
 '380' '280 max' '240 max' '210 max' '265' '285' '325' '355' '300' '340'
 '320' '350' '430' '460' '520' '590' '630' '660' '680' '270' '650' '770'
 '700' '900' '1050' '390' '585' '450' '240' '190' '180' '280' '150' '250'
 '370' '420' '480' '260' '375' '250 max' '195' '470' '275' '410' '1300'
 '670' '665' '1270' '1180' '750' '760' '690' '850' '735' '1030' '785'
 '550' '800' '205' '210' '580' '140' '215' '245' '1100' '510' '570' '740'
 '685' '925' '635' '226' '373' '235' '315' '1470' '1320' '540' '1080'
 '1210' '343' '637' '1373' '687' '588' '686' '794' '1175' '1275' '981'
 '880' '495' '560' '175' '100' '160' '120' '130' '335' '196' '220' '330'
 '950' '1250' '835' '1200' '885' '1400' '216' '294' '334' '392' '491'
 '589' '441' '638' '1079' '1177' '1472' '1422' '1325' '50' '75' '105'
 '125' '170' '225 max' '155' '1350' '620' '715' '815' '1125' '720' '810'
 '185' '640' '177' '539']
```

The column 'Sy' has entries like "280 max", etc. It has to be cleaned.

In [8]:
```
# Remove non-numeric characters and convert to int
d2["Sy"] = d2["Sy"].astype(str).str.extract(r'(\d+)').astype(float).astype(np.int64)
d2["Sy"]
```

Out[8]:
```
0       314
1       324
2       284
3       331
4       346
       ...
1547    370
1548    420
1549    480
1550    180
1551    260
Name: Sy, Length: 1552, dtype: int64
```

8 columns (attributes) in the dataset:

1. ID - Unique Identification code for the Material

2. Material - Consists of Standard (Std), Material name and Heat treatment method.

3. Su - Ultimate Tensile Strength in Mpa (Megapascal)

4. Sy - Yield Strength in Mpa (Megapascal)

5. E - Elastic Modulus in Mpa (Megapascal)

6. G - Shear Modulus in Mpa (Megapascal)

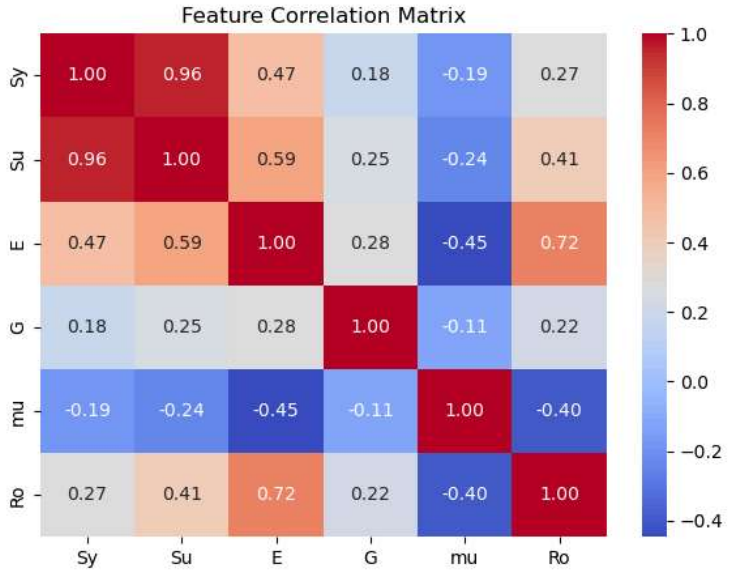
7. mu - Poisson's Ratio in mu (Units of Length)

8. Ro - Density in Kg/m3

Considering it as a Regression problem : Predicting a performance score for material suitability.

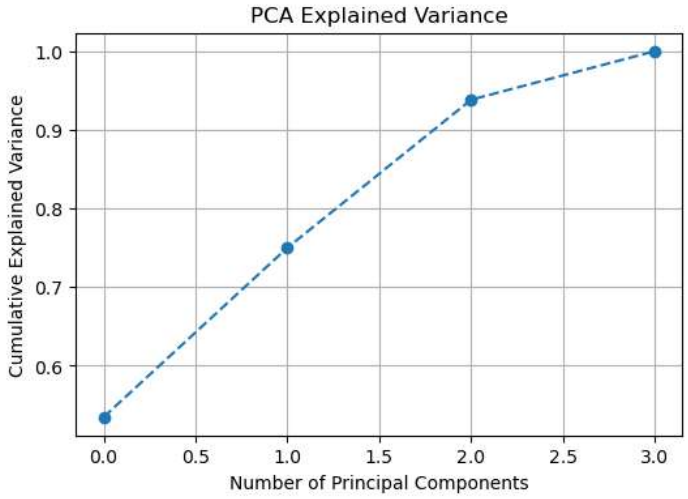
Since there is no specific target variable (performance score), use Principal Component Analysis (PCA) to generate a performance score.

In [9]:
```
# Select numerical features.
features = ["Sy", "Su", "E", "G", "mu", "Ro"]
X = d2[features]

# Standardize Data
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Visualize correlation matrix
plt.figure(figsize=(7, 5))
sns.heatmap(pd.DataFrame(X_scaled, columns=features).corr(), annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Feature Correlation Matrix")
plt.show()
```



Since, 'Sy' and 'Su' are highly correlated features. Remove one of them, apply PCA and compute explained variance:

In [10]:
```
# Principal Component Analysis (PCA)
from sklearn.decomposition import PCA

# Select numerical features. 'Su' is removed, 'mu' is not used bcz it has lower variance compared to other features.
features2 = ["Sy", "E", "G", "Ro"]
X = d2[features2]

# Standardize Data
from sklearn.preprocessing import StandardScaler, MinMaxScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)


# Apply PCA
pca = PCA()
X_pca = pca.fit_transform(X_scaled)

# Explained variance ratio
explained_variance = pca.explained_variance_ratio_

# Plot the explained variance
plt.figure(figsize=(6, 4))
plt.plot(np.cumsum(explained_variance), marker="o", linestyle="--")
plt.xlabel("Number of Principal Components")
plt.ylabel("Cumulative Explained Variance")
plt.title("PCA Explained Variance")
plt.grid()
plt.show()

# Print explained variance for each component
for i, var in enumerate(explained_variance):
    print(f"Principal Component {i+1}: {var:.4f}")
```




```
Principal Component 1: 0.5347
Principal Component 2: 0.2153
Principal Component 3: 0.1882
Principal Component 4: 0.0618
```

From the above curve, the first 3 principal components should be enough to capture most of the information.

Use the first 3 principal components for your performance score generation:

In [11]:
```
# Select numerical features. 'mu' is not used bcz it has lower variance compared to other features.
features3 = ["Sy", "E", "G", "Ro"]
X = d2[features3]

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # Standardizing data

# Apply PCA
pca = PCA(n_components=3)  # Keep first 3 components
X_pca = pca.fit_transform(X_scaled)

# Get explained variance ratio (importance of each PC)
explained_variance = pca.explained_variance_ratio_

# Compute the performance score as a weighted sum
performance_score = np.dot(X_pca, explained_variance)

# Normalize the score (optional)
scaler = MinMaxScaler()
performance_score_normalized = scaler.fit_transform(performance_score.reshape(-1, 1))

# Add score to the dataframe
d2["Performance Score"] = performance_score_normalized

# Check dataset
d2
```

Out[11]:

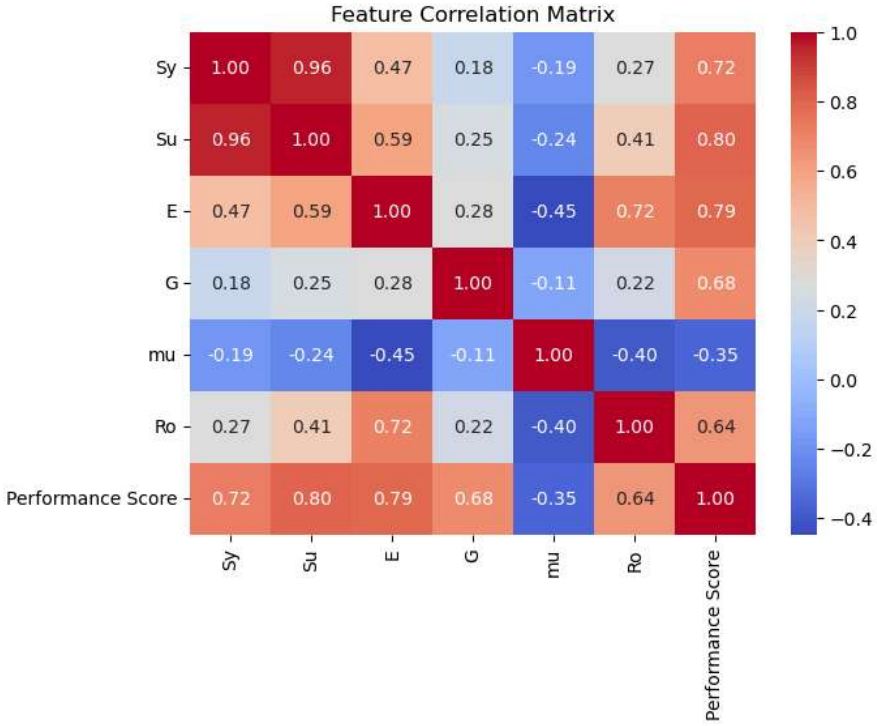
	ID	Material	Su	Sy	E	G	mu	Ro	Performance Score
0	D8894772B88F495093C43AF905AB6373	ANSI Steel SAE 1015 as-rolled	421	314	207000	79000	0.30	7860	0.278873
1	05982AC66F064F9EBC709E7A4164613A	ANSI Steel SAE 1015 normalized	424	324	207000	79000	0.30	7860	0.281068
2	356D6E63FF9A49A3AB23BF66BAC85DC3	ANSI Steel SAE 1015 annealed	386	284	207000	79000	0.30	7860	0.272290
3	1C758F8714AC4E0D9BD8D8AE1625AECD	ANSI Steel SAE 1020 as-rolled	448	331	207000	79000	0.30	7860	0.282604
4	DCE10036FC1946FC8C9108D598D116AD	ANSI Steel SAE 1020 normalized	441	346	207000	79000	0.30	7860	0.285896
...	...	...	...	...	...	...	...	...	...
1547	512A80EC21EA416BA2725B38BA8096EF	JIS Nodular cast iron	600	370	169000	70000	0.20	7160	0.245236
1548	38526441BA8741CA979DBF870D0B8A9B	JIS Nodular cast iron	700	420	169000	70000	0.20	7160	0.256208
1549	CAC03D7EB1AA45E68EFF92A2EF4C3D9B	JIS Nodular cast iron	800	480	169000	70000	0.20	7160	0.269375
1550	45C82A36EC644F8BB6170A99ED819B62	JIS Malleable cast iron	400	180	160000	64000	0.27	7160	0.192752
1551	BC74F870412F4DDBADDEF1063C1C079F	JIS Malleable cast iron	500	260	160000	64000	0.27	7160	0.210308

1552 rows × 9 columns

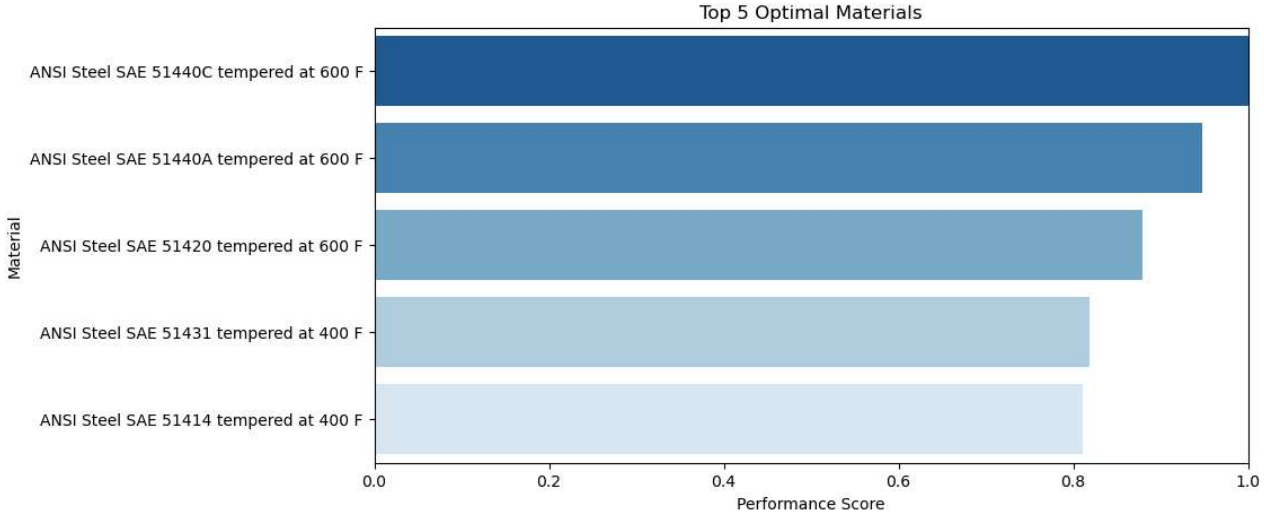
Summary of above steps:

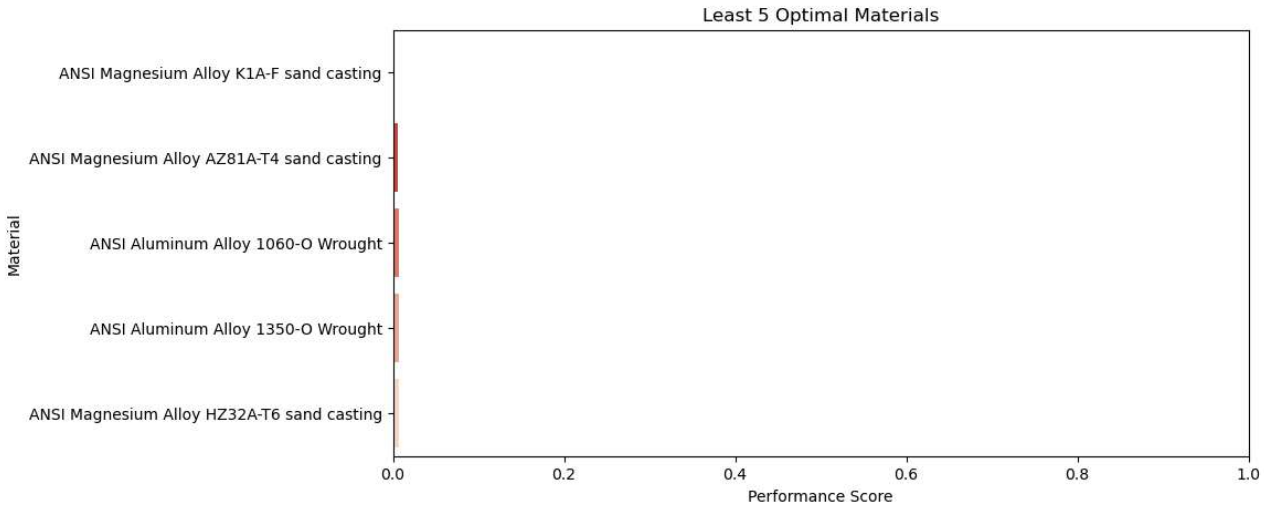
1. Check correlation matrix, and remove one of the highly correlated features.
2. Standardize the feature data.
3. Apply PCA and selects the first 3 components.
4. Compute Performance score and normalize the score for better interpretability.
5. Add the performance score to the dataset.

In [12]:
```
# Correlation matrix:
# Select numerical features.
features4 = ["Sy", "Su", "E", "G", "mu", "Ro", "Performance Score"]
X = d2[features4]

# Standardize Data
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Visualize correlation matrix
plt.figure(figsize=(7, 5))
sns.heatmap(pd.DataFrame(X_scaled, columns=features4).corr(), annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Feature Correlation Matrix")
plt.show()
```



In [13]:
```
# Create recommended_materials DataFrame (Top 5 & Bottom 5)
top_5 = d2.nlargest(5, "Performance Score")
bottom_5 = d2.nsmallest(5, "Performance Score")
recommended_materials = pd.concat([top_5, bottom_5]).reset_index(drop=True)

# Visualization Function
def plot_optimal_materials(data):
    """
    Visualizes the highest and least optimal materials based on performance score.
    """
    top_materials = data.nlargest(5, 'Performance Score')
    least_materials = data.nsmallest(5, 'Performance Score')

    # Top Performing Materials
    plt.figure(figsize=(10, 5))
    sns.barplot(x=top_materials['Performance Score'], y=top_materials['Material'], palette='Blues_r')
    plt.xlabel('Performance Score')
    plt.ylabel('Material')
    plt.title('Top 5 Optimal Materials')
    plt.xlim([0, 1])
    plt.show()

    # Least Performing Materials
    plt.figure(figsize=(10, 5))
    sns.barplot(x=least_materials['Performance Score'], y=least_materials['Material'], palette='Reds_r')
    plt.xlabel('Performance Score')
    plt.ylabel('Material')
    plt.title('Least 5 Optimal Materials')
    plt.xlim([0, 1])
    plt.show()

# Call the function
plot_optimal_materials(recommended_materials)
```





In [14]:
```
def heatmap_materials(data):
    """
    Heatmap showing key properties of top and least optimal materials
    """
    plt.figure(figsize=(8, 5))
    sns.heatmap(data.set_index("Material")[["Sy", "E", "G", "Ro", "Performance Score"]], cmap="coolwarm", annot=True, fmt=".2f")
    plt.title("Material Properties Heatmap")
    plt.show()

# Call function
heatmap_materials(recommended_materials)
```

In [15]:
```
from sklearn.model_selection import train_test_split


# Select features and target variable
X1 = d2[["Sy", "Su", "E", "G", "mu", "Ro"]]
y1 = d2["Performance Score"]

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X1)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y1, test_size=0.2, random_state=42)
```

1. LINEAR REGRESSION

In [16]:
```
from sklearn.linear_model import LinearRegression

# Initialize and train the model
LR_model = LinearRegression()
LR_model.fit(X_train, y_train)
```

Out[16]:
```
LinearRegression()
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [17]:
```
from sklearn.metrics import mean_squared_error, r2_score

# Predictions
y_train_pred_LR = LR_model.predict(X_train)
y_test_pred_LR = LR_model.predict(X_test)

# Compute metrics
mse_LR = mean_squared_error(y_test, y_test_pred_LR)
r2_train_LR = r2_score(y_train, y_train_pred_LR)
r2_test_LR = r2_score(y_test, y_test_pred_LR)

print(f"Mean Squared Error (Linear regression): {mse_LR}")
print(f"Training R² Score (Linear regression): {r2_train_LR:.4f}")
print(f"Test R² Score (Linear regression): {r2_test_LR:.4f}")
```

```
Mean Squared Error (Linear regression): 1.5318264663782207e-32
Training R² Score (Linear regression): 1.0000
Test R² Score (Linear regression): 1.0000
```

With an MSE close to zero and an R^2 of 1.0, the model could indicate overfitting. Dataset might be perfectly linear.

2. DECISION TREE REGRESSOR

In [18]:
```
from sklearn.tree import DecisionTreeRegressor

# Initialize Decision Tree Regressor
dt_regressor = DecisionTreeRegressor(random_state=42)

# Train the model
dt_regressor.fit(X_train, y_train)

# Predictions
y_train_pred_DT = dt_regressor.predict(X_train)
y_test_pred_DT = dt_regressor.predict(X_test)

# Evaluate performance
mse_DT = mean_squared_error(y_test, y_test_pred_DT)
r2_train_DT = r2_score(y_train, y_train_pred_DT)
r2_test_DT = r2_score(y_test, y_test_pred_DT)

print(f"Mean Squared Error (Decision Tree Regression): {mse_DT}")
print(f"Training R² Score (Decision Tree Regression): {r2_train_DT:.4f}")
print(f"Test R² Score (Decision Tree Regression): {r2_test_DT:.4f}")
```

```
Mean Squared Error (Decision Tree Regression): 1.6670244014297756e-05
Training R² Score (Decision Tree Regression): 1.0000
Test R² Score (Decision Tree Regression): 0.9991
```

The Decision Tree model has risk of Overfitting, since it perfectly fits training data but slightly drops in test performance.

3. RANDOM FOREST REGRESSION

In [19]:
```
from sklearn.ensemble import RandomForestRegressor

# Initialize and train the model
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
rf_regressor.fit(X_train, y_train)

# Predictions
y_train_pred_RF = rf_regressor.predict(X_train)
y_test_pred_RF = rf_regressor.predict(X_test)

# Evaluate performance
mse_RF = mean_squared_error(y_test, y_test_pred_RF)
r2_train_RF = r2_score(y_train, y_train_pred_RF)
r2_test_RF = r2_score(y_test, y_test_pred_RF)


# Print results
print(f"Mean Squared Error (Random Forest Regression): {mse_RF}")
print(f"Training R² Score (Random Forest Regression): {r2_train_RF:.4f}")
print(f"Test R² Score (Random Forest Regression): {r2_test_RF:.4f}")
```

```
Mean Squared Error (Random Forest Regression): 1.4685456414299405e-05
Training R² Score (Random Forest Regression): 0.9998
Test R² Score (Random Forest Regression): 0.9992
```

Random Forest provides a more stable and generalizable performance compared to Decision Tree.

4. XGBOOST REGRESSION

In [20]:
```
!pip install xgboost
```

```
Requirement already satisfied: xgboost in c:\users\admin\anaconda3\lib\site-packages (3.0.0)
Requirement already satisfied: numpy in c:\users\admin\anaconda3\lib\site-packages (from xgboost) (1.24.3)
Requirement already satisfied: scipy in c:\users\admin\anaconda3\lib\site-packages (from xgboost) (1.10.1)
```

In [21]:
```
from xgboost import XGBRegressor

# Initialize the XGBoost regressor
xgb_reg = XGBRegressor(objective="reg:squarederror", n_estimators=100, learning_rate=0.1, max_depth=5, random_state=42)

# Train the model
xgb_reg.fit(X_train, y_train)

# Predictions
y_train_pred_XG = xgb_reg.predict(X_train)
y_test_pred_XG = xgb_reg.predict(X_test)

# Performance Metrics
mse_XG = mean_squared_error(y_test, y_test_pred_XG)
r2_train_XG = r2_score(y_train, y_train_pred_XG)
r2_test_XG = r2_score(y_test, y_test_pred_XG)

print(f"Mean Squared Error (Xgboost): {mse_XG}")
print(f"Training R² Score (Xgboost): {r2_train_XG:.4f}")
print(f"Test R² Score (Xgboost): {r2_test_XG:.4f}")
```

```
Mean Squared Error (Xgboost): 5.333095041508803e-06
Training R² Score (Xgboost): 0.9999
Test R² Score (Xgboost): 0.9997
```

XGBoost gives the best performance with the lowest error and highest R² scores.

Next, fine-tune hyperparameters for XGBoost to optimize:

In [22]:
```
from xgboost import XGBRegressor
from sklearn.model_selection import GridSearchCV

# Define parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],   # Number of trees
    'max_depth': [3, 4, 7],           # Maximum depth of trees
    'learning_rate': [0.01, 0.1, 0.2], # Step size shrinkage
    'subsample': [0.8, 1.0],          # Fraction of samples per tree
    'colsample_bytree': [0.8, 1.0]    # Fraction of features per tree
}

# Initialize XGBoost regressor
xgb = XGBRegressor(objective='reg:squarederror', random_state=42)

# Perform Grid Search with Cross-Validation (CV=5)
grid_search = GridSearchCV(estimator=xgb, param_grid=param_grid,
                           cv=5, scoring='r2', n_jobs=-1, verbose=2)

# Fit on training data
grid_search.fit(X_train, y_train)

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

# Use the best model
best_xgb = grid_search.best_estimator_

# Make predictions
y_test_pred_XGB = best_xgb.predict(X_test)

# Evaluate model
mse_XGB = mean_squared_error(y_test, y_test_pred_XGB)
r2_train_XGB = best_xgb.score(X_train, y_train)
r2_test_XGB = best_xgb.score(X_test, y_test)

print(f"Mean Squared Error (Xgboost): {mse_XGB}")
print(f"Training R² Score (Xgboost): {r2_train_XGB:.4f}")
print(f"Test R² Score  (Xgboost): {r2_test_XGB:.4f}")
```

```
Fitting 5 folds for each of 108 candidates, totalling 540 fits
Best Parameters: {'colsample_bytree': 1.0, 'learning_rate': 0.2, 'max_depth': 4, 'n_estimators': 200, 'subsample': 0.8}
Mean Squared Error (Xgboost): 6.118589484138114e-06
Training R² Score (Xgboost): 0.9999
Test R² Score  (Xgboost): 0.9997
```

XGboost is the most accurate model with the lowest error and highest test R².

5. GRADIENT BOOSTING REGRESSION

In [23]:
```
from sklearn.ensemble import GradientBoostingRegressor

# Initialize the model with default parameters
gbr = GradientBoostingRegressor(random_state=42)

# Train the model
gbr.fit(X_train, y_train)

# Predictions
y_train_pred_GB = gbr.predict(X_train)
y_test_pred_GB = gbr.predict(X_test)

mse_GB = mean_squared_error(y_test, y_test_pred_GB)
train_r2_GB = r2_score(y_train, y_train_pred_GB)
test_r2_GB = r2_score(y_test, y_test_pred_GB)

print(f"Mean Squared Error (Gradient Boosting): {mse_GB}")
print(f"Training R² Score (Gradient Boosting): {train_r2_GB:.4f}")
print(f"Test R² Score (Gradient Boosting): {test_r2_GB:.4f}")
```

```
Mean Squared Error (Gradient Boosting): 1.538745248397711e-05
Training R² Score (Gradient Boosting): 0.9995
Test R² Score (Gradient Boosting): 0.9992
```

GBR is performing well, but slightly less than XGBoost, which has a higher R².

6. ARTIFICIAL NEURAL NETWORK (ANN)

In [26]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

In [39]:
```
# Define ANN model
model = Sequential()
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))  # Regression output

# Compile the model
model.compile(optimizer='adam', loss='mse')

# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=10, verbose=0)

# Predict on training and testing sets
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Evaluate performance
mse_ann = mean_squared_error(y_test, y_test_pred)
train_r2_ann = r2_score(y_train, y_train_pred)
test_r2_ann = r2_score(y_test, y_test_pred)

# Print results
print("ANN Regression Results:")
print(f"Mean Sqaured Error (ANN): {mse_ann:.4e}")
print(f"Training R² Score (ANN): {train_r2_ann:.4f}")
print(f"Test R² Score (ANN): {test_r2_ann:.4f}")
```

```
C:\Users\Admin\AppData\Roaming\Python\Python311\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `i
nput_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first l
ayer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

39/39 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
ANN Regression Results:
Mean Sqaured Error (ANN): 4.4331e-06
Training R² Score (ANN): 0.9998
Test R² Score (ANN): 0.9998
```

Early stopping

In [45]:
```
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_squared_error, r2_score

# Define ANN model
model_ann = Sequential()
model_ann .add(Dense(64, input_dim=X_train.shape[1], activation='relu'))
model_ann .add(Dense(32, activation='relu'))
model_ann .add(Dense(1))

# Compile the model
model_ann .compile(optimizer='adam', loss='mse')

# Train the model with EarlyStopping and capture history
early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
history = model_ann .fit(X_train, y_train, validation_split=0.2,
                    epochs=200, batch_size=32, callbacks=[early_stop], verbose=1)

# Plot training and validation loss
plt.figure(figsize=(6, 4))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss', linestyle='--')
plt.title('Training vs. Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('MSE Loss')
plt.legend()
#plt.grid(True)
plt.tight_layout()
plt.show()

# Predict on training and testing sets
y_train_pred = model_ann .predict(X_train)
y_test_pred = model_ann .predict(X_test)

# Evaluate performance
mse_ann = mean_squared_error(y_test, y_test_pred)
train_r2_ann = r2_score(y_train, y_train_pred)
test_r2_ann = r2_score(y_test, y_test_pred)

# Print results
print("ANN Regression Results:")
print(f"Mean Sqaured Error (ANN): {mse_ann:.4e}")
print(f"Training R² Score (ANN): {train_r2_ann:.4f}")
print(f"Test R² Score (ANN): {test_r2_ann:.4f}")
```

```
C:\Users\Admin\AppData\Roaming\Python\Python311\site-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)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Epoch 1/200
31/31 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 0.0720 - val_loss: 0.0047
Epoch 2/200
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0031 - val_loss: 8.7687e-04
Epoch 3/200
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.9284e-04 - val_loss: 3.7459e-04
Epoch 4/200
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 3.9766e-04 - val_loss: 2.7693e-04
Epoch 5/200
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 2.2221e-04 - val_loss: 2.1215e-04
Epoch 6/200
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.8413e-04 - val_loss: 1.7482e-04
Epoch 7/200
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.6744e-04 - val_loss: 1.5915e-04
Epoch 8/200
```

In [46]:
```
import pandas as pd
from prettytable import PrettyTable

# Store model performance
model_performance = {
    "Linear Regression": {"MSE": mse_LR, "Train R²": r2_train_LR, "Test R²": r2_test_LR},
    "Decision Tree": {"MSE": mse_DT, "Train R²": r2_train_DT, "Test R²": r2_test_DT},
    "Random Forest": {"MSE": mse_RF, "Train R²": r2_train_RF, "Test R²": r2_test_RF},
    "XGBoost": {"MSE": mse_XGB, "Train R²": r2_train_XGB, "Test R²": r2_test_XGB},
    "Gradient Boosting": {"MSE": mse_GB, "Train R²": train_r2_GB, "Test R²": test_r2_GB},
    "Artificial Neural Network": {"MSE": mse_ann, "Train R²": train_r2_ann, "Test R²": test_r2_ann}
}

# Convert to DataFrame
performance_df = pd.DataFrame.from_dict(model_performance, orient="index").reset_index()
performance_df.columns = ["Model", "MSE", "Train R²", "Test R²"]

# PrettyTable for better display
table = PrettyTable()
table.field_names = ["Model", "MSE", " Train R²", " Test R²"]

for _, row in performance_df.iterrows():
    table.add_row([row["Model"], f"{row['MSE']:.4e}", f"{row['Train R²']:.4f}", f"{row['Test R²']:.4f}"])

print(table)
```

```
+---------------------------+------------+-----------+----------+
|           Model           |    MSE     |  Train R² |  Test R² |
+---------------------------+------------+-----------+----------+
|     Linear Regression     | 1.5318e-32 |   1.0000  |  1.0000  |
|       Decision Tree       | 1.6670e-05 |   1.0000  |  0.9991  |
|       Random Forest       | 1.4685e-05 |   0.9998  |  0.9992  |
|          XGBoost          | 6.1186e-06 |   0.9999  |  0.9997  |
|     Gradient Boosting     | 1.5387e-05 |   0.9995  |  0.9992  |
| Artificial Neural Network | 3.3370e-05 |   0.9987  |  0.9982  |
+---------------------------+------------+-----------+----------+
```

Best Model

XGBoost is the best model as it has lowest MSE and highest generalization capability.

Gradient Boosting is other alternative. However, it is slightly less accurate than XGBoost.

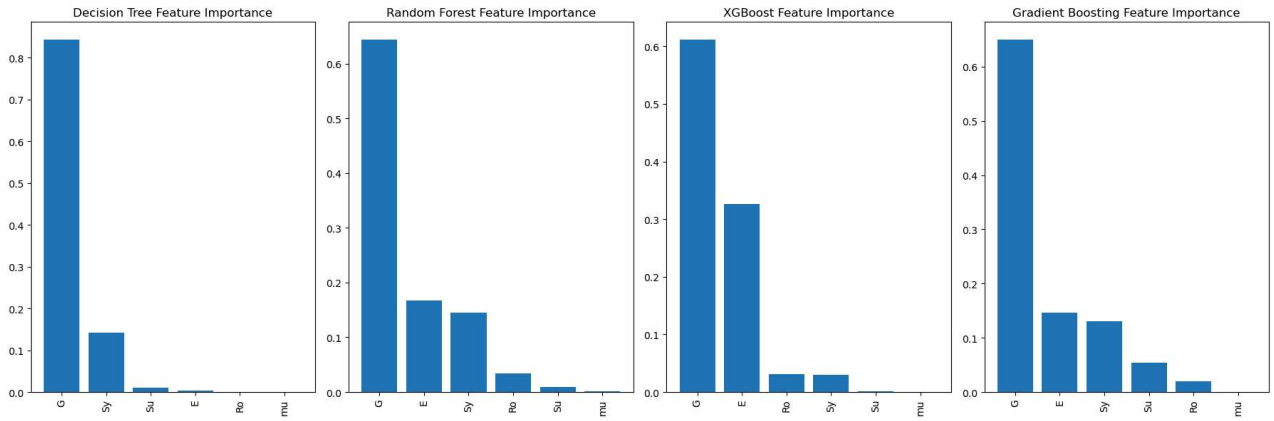
Artificial Neural Network achieves solid performance, while not outperforming XGBoost.

Random Forest and Decision Tree model memorizes the training data well but generalizes slightly less effectively to new data (overfitting).

Linear Regression is suspiciously too good to be true. Suggests overfitting or that the dataset might be perfectly linear.

Feature Importance Analysis

In [50]:
```
import matplotlib.pyplot as plt
import numpy as np

feature_names = ["Sy", "Su", "E", "G", "mu", "Ro"]

models = {
    "Decision Tree": dt_regressor,
    "Random Forest": rf_regressor,
    "XGBoost": best_xgb,
    "Gradient Boosting": gbr
}

plt.figure(figsize=(18, 6))

for i, (name, model) in enumerate(models.items()):
    plt.subplot(1, 4, i + 1)
    feature_importances = model.feature_importances_
    indices = np.argsort(feature_importances)[::-1]
    plt.bar(range(len(feature_importances)), feature_importances[indices])
    plt.xticks(range(len(feature_importances)), np.array(feature_names)[indices], rotation=90)
    plt.title(f"{name} Feature Importance")

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



ENSEMBLE MODEL

STEPS:

1. Add a binary column (YES/NO) to the dataset.
2. Combine the prediction results from 5 models
3. Perform Majority / Average / Maximum voting to combine results, Output is performance score
4. Give this output as input to the Decision tree classifier, which classifies materials as Yes/No.

In [30]:
```
# Create another column - YES/No

# Predict performance scores for the full feature dataset X
y_all_pred_LR = LR_model.predict(X_scaled)
y_all_pred_DT = dt_regressor.predict(X_scaled)
y_all_pred_RF = rf_regressor.predict(X_scaled)
y_all_pred_XGB = best_xgb.predict(X_scaled)
y_all_pred_GB = gbr.predict(X_scaled)

# Stack predictions into a 2D array and compute average
all_predictions = np.vstack([
    y_all_pred_LR,
    y_all_pred_DT,
    y_all_pred_RF,
    y_all_pred_XGB,
    y_all_pred_GB
])
avg_score_full = np.mean(all_predictions, axis=0)

# Generate YES/NO labels based on domain-specific threshold
threshold = 0.5
labels_full = np.where(avg_score_full > threshold, "YES", "NO")

# Create a new DataFrame with added columns
d3 = d2.copy()
#d3['Avg_Performance_Score'] = avg_score_full
d3['Material_Selection'] = labels_full
```

In [32]:
```
d3.head()
```

Out[32]:

	ID	Material	Su	Sy	E	G	mu	Ro	Performance Score	Material_Selection
0	D8894772B88F495093C43AF905AB6373	ANSI Steel SAE 1015 as-rolled	421	314	207000	79000	0.3	7860	0.278873	NO
1	05982AC66F064F9EBC709E7A4164613A	ANSI Steel SAE 1015 normalized	424	324	207000	79000	0.3	7860	0.281068	NO
2	356D6E63FF9A49A3AB23BF66BAC85DC3	ANSI Steel SAE 1015 annealed	386	284	207000	79000	0.3	7860	0.272290	NO
3	1C758F8714AC4E0D9BD8D8AE1625AECD	ANSI Steel SAE 1020 as-rolled	448	331	207000	79000	0.3	7860	0.282604	NO
4	DCE10036FC1946FC8C9108D598D116AD	ANSI Steel SAE 1020 normalized	441	346	207000	79000	0.3	7860	0.285896	NO

In [33]:
```
# Step 0: Prepare data
X_e = d3.drop(columns=['ID', 'Material', 'Performance Score', 'Material_Selection'])  # Features
y_e = d3['Performance Score']  # Regression target
labels = (y_e > 0.5).astype(int)  # YES=1, NO=0 for classification


# Step 1: Train-Test Split
X_train_e, X_test_e, y_train_e, y_test_e, y_train_labels_e, y_test_labels_e = train_test_split(
    X_e, y_e, labels, test_size=0.2, random_state=42)
```

Ensemble model : Majority Voting

In [55]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report


#  Train 5 Regression Models
lr_e = LinearRegression().fit(X_train_e, y_train_e)
dt_e = DecisionTreeRegressor().fit(X_train_e, y_train_e)
rf_e = RandomForestRegressor().fit(X_train_e, y_train_e)
gb_e = GradientBoostingRegressor().fit(X_train_e, y_train_e)
xgb_e = XGBRegressor().fit(X_train_e, y_train_e)

#  Predict performance scores on test data
preds_test = np.vstack([
    lr_e.predict(X_test_e),
    dt_e.predict(X_test_e),
    rf_e.predict(X_test_e),
    gb_e.predict(X_test_e),
    xgb_e.predict(X_test_e)
])

avg_pred_test = np.mean(preds_test, axis=0)  # Final regressor output (Performance Score)

#  Majority voting for YES/NO classification
binary_preds = (preds_test > 0.5).astype(int)
majority_vote = (np.mean(binary_preds, axis=0) > 0.5).astype(int)

#  Train classifier on avg predicted performance scores from training set
preds_train = np.vstack([
    lr_e.predict(X_train_e),
    dt_e.predict(X_train_e),
    rf_e.predict(X_train_e),
    gb_e.predict(X_train_e),
    xgb_e.predict(X_train_e)
])
avg_pred_train = np.mean(preds_train, axis=0).reshape(-1, 1)

clf = DecisionTreeClassifier().fit(avg_pred_train, y_train_labels_e)

#  Classify test materials using average predicted score
final_class_labels = clf.predict(avg_pred_test.reshape(-1, 1))

#  Evaluate
mse_en_maj = mean_squared_error(y_test_e, avg_pred_test)
train_r2_en_maj = r2_score(y_train_e, avg_pred_train)
test_r2_en_maj = r2_score(y_test_e, avg_pred_test)
print("=== Ensemble Model Performance (Majority Voting) ===")
print(f"Mean Sqaured Error: {mse_en_maj:.4e}")
print(f"Training R² Score: {train_r2_en_maj:.4f}")
print(f"Test R² Score (ANN): {test_r2_en_maj:.4f}")


print("\n=== Decision Tree Classification Report (Majority Voting) ===")
print(classification_report(y_test_labels_e, final_class_labels, target_names=["NO", "YES"]))
```

```
=== Ensemble Model Performance (Majority Voting) ===
Mean Sqaured Error: 4.2040e-06
Training R² Score: 1.0000
Test R² Score (ANN): 0.9998

=== Decision Tree Classification Report (Majority Voting) ===
              precision    recall  f1-score   support

          NO       1.00      1.00      1.00       299
         YES       1.00      1.00      1.00        12

    accuracy                           1.00       311
   macro avg       1.00      1.00      1.00       311
weighted avg       1.00      1.00      1.00       311
```

Ensemble model : Average Voting

In [58]:
```
#  Train 5 Regression Models
lr_e = LinearRegression().fit(X_train_e, y_train_e)
dt_e = DecisionTreeRegressor().fit(X_train_e, y_train_e)
rf_e = RandomForestRegressor().fit(X_train_e, y_train_e)
gb_e = GradientBoostingRegressor().fit(X_train_e, y_train_e)
xgb_e = XGBRegressor().fit(X_train_e, y_train_e)

#  Predict performance scores on test data
preds_test = np.vstack([
    lr_e.predict(X_test_e),
    dt_e.predict(X_test_e),
    rf_e.predict(X_test_e),
    gb_e.predict(X_test_e),
    xgb_e.predict(X_test_e)
])
avg_pred_test = np.mean(preds_test, axis=0)  # Final regressor output (Performance Score)

#  Average Voting for YES/NO classification
average_vote = (avg_pred_test > 0.5).astype(int)

#  Train classifier on avg predicted performance scores from training set
preds_train = np.vstack([
    lr_e.predict(X_train_e),
    dt_e.predict(X_train_e),
    rf_e.predict(X_train_e),
    gb_e.predict(X_train_e),
    xgb_e.predict(X_train_e)
])
avg_pred_train = np.mean(preds_train, axis=0).reshape(-1, 1)

clf = DecisionTreeClassifier().fit(avg_pred_train, y_train_labels_e)

#  Classify test materials using average predicted score
final_class_labels = clf.predict(avg_pred_test.reshape(-1, 1))

# Evaluate
mse_en_avg = mean_squared_error(y_test_e, avg_pred_test)
train_r2_en_avg = r2_score(y_train_e, avg_pred_train)
test_r2_en_avg = r2_score(y_test_e, avg_pred_test)

print("=== Ensemble Model Performance (Average Voting) ===")
print(f"Mean Squared Error: {mse_en_avg:.4e}")
print(f"Training R² Score: {train_r2_en_avg:.4f}")
print(f"Test R² Score: {test_r2_en_avg:.4f}")

print("\n=== Decision Tree Classification Report (Average Voting) ===")
print(classification_report(y_test_labels_e, final_class_labels, target_names=["NO", "YES"]))
```

```
=== Ensemble Model Performance (Average Voting) ===
Mean Squared Error: 4.7200e-06
Training R² Score: 1.0000
Test R² Score: 0.9997

=== Decision Tree Classification Report (Average Voting) ===
              precision    recall  f1-score   support

          NO       1.00      1.00      1.00       299
         YES       1.00      1.00      1.00        12

    accuracy                           1.00       311
   macro avg       1.00      1.00      1.00       311
weighted avg       1.00      1.00      1.00       311
```

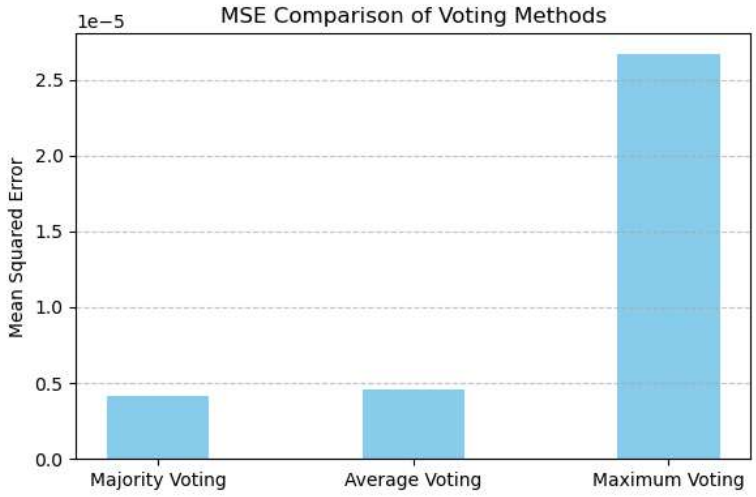
Ensemble model : Maximum Voting

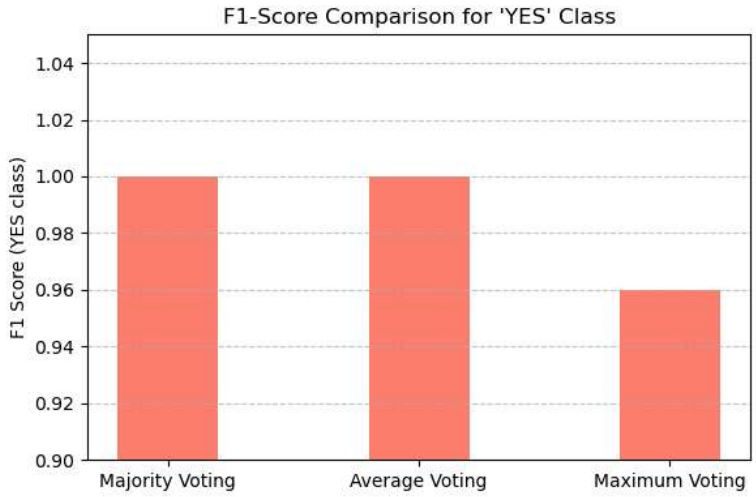
In [67]:
```
# Train 5 Regression Models
lr_e = LinearRegression().fit(X_train_e, y_train_e)
dt_e = DecisionTreeRegressor().fit(X_train_e, y_train_e)
rf_e = RandomForestRegressor().fit(X_train_e, y_train_e)
gb_e = GradientBoostingRegressor().fit(X_train_e, y_train_e)
xgb_e = XGBRegressor().fit(X_train_e, y_train_e)

#  Predict performance scores on test data
preds_test = np.vstack([
    lr_e.predict(X_test_e),
    dt_e.predict(X_test_e),
    rf_e.predict(X_test_e),
    gb_e.predict(X_test_e),
    xgb_e.predict(X_test_e)
])
max_pred_test = np.max(preds_test, axis=0)  # Final regressor output (Maximum Voting Score)

# Maximum Voting for YES/NO classification
max_vote = (max_pred_test > 0.5).astype(int)

#  Train classifier on max predicted performance scores from training set
preds_train = np.vstack([
    lr_e.predict(X_train_e),
    dt_e.predict(X_train_e),
    rf_e.predict(X_train_e),
    gb_e.predict(X_train_e),
    xgb_e.predict(X_train_e)
])
max_pred_train = np.max(preds_train, axis=0).reshape(-1, 1)

clf = DecisionTreeClassifier().fit(max_pred_train, y_train_labels_e)

#  Classify test materials using max predicted score
final_class_labels = clf.predict(max_pred_test.reshape(-1, 1))

#  Evaluate
mse_en_max = mean_squared_error(y_test_e, max_pred_test)
train_r2_en_max = r2_score(y_train_e, max_pred_train)
test_r2_en_max = r2_score(y_test_e, max_pred_test)

print("=== Ensemble Model Performance (Maximum Voting) ===")
print(f"Mean Squared Error: {mse_en_max:.4e}")
print(f"Training R² Score: {train_r2_en_max:.4f}")
print(f"Test R² Score: {test_r2_en_max:.4f}")

print("\n=== Decision Tree Classification Report (Maximum Voting) ===")
print(classification_report(y_test_labels_e, final_class_labels, target_names=["NO", "YES"]))
```

```
=== Ensemble Model Performance (Maximum Voting) ===
Mean Squared Error: 2.8353e-05
Training R² Score: 0.9996
Test R² Score: 0.9985

=== Decision Tree Classification Report (Maximum Voting) ===
              precision    recall  f1-score   support

          NO       1.00      1.00      1.00       299
         YES       0.92      1.00      0.96        12

    accuracy                           1.00       311
   macro avg       0.96      1.00      0.98       311
weighted avg       1.00      1.00      1.00       311
```

Analysis of different voting strategies

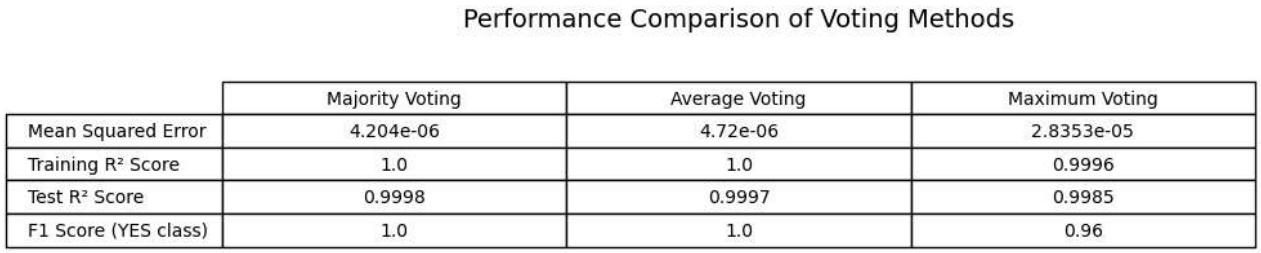
In [37]:
```
# Data
methods = ['Majority Voting', 'Average Voting', 'Maximum Voting']
mse_values = [mse_en_maj, mse_en_avg, mse_en_max]
f1_yes_scores = [1.00, 1.00, 0.96]

x = np.arange(len(methods))
width = 0.4

# Plot MSE
plt.figure(figsize=(6, 4))
plt.bar(x, mse_values, width, color='skyblue')
plt.xticks(x, methods, rotation=0)
plt.ylabel("Mean Squared Error")
plt.title("MSE Comparison of Voting Methods")
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# Plot F1-Score for "YES" class
plt.figure(figsize=(6, 4))
plt.bar(x, f1_yes_scores, width, color='salmon')
plt.xticks(x, methods, rotation=0)
plt.ylim(0.9, 1.05)
plt.ylabel("F1 Score (YES class)")
plt.title("F1-Score Comparison for 'YES' Class")
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```





In [68]:
```
# Define the performance data
data = {
    "Metric": [
        "Mean Squared Error",
        "Training R² Score",
        "Test R² Score",
        "F1 Score (YES class)"
    ],
    "Majority Voting": [4.2040e-06, 1.0000, 0.9998, 1.00],
    "Average Voting": [4.7200e-06, 1.0000, 0.9997, 1.00],
    "Maximum Voting": [2.8353e-05, 0.9996, 0.9985, 0.96],
}

# Create DataFrame
df = pd.DataFrame(data)

# Set Metric as index
df.set_index("Metric", inplace=True)

# Display the table as text
print(df)

# Optional: Visualize the table using matplotlib
fig, ax = plt.subplots(figsize=(10, 2.5))
ax.axis('off')
table = ax.table(cellText=df.values,
                 rowLabels=df.index,
                 colLabels=df.columns,
                 loc='center',
                 cellLoc='center')
table.scale(1, 1.5)
table.auto_set_font_size(False)
table.set_fontsize(10)
plt.title("Performance Comparison of Voting Methods", fontsize=14)
plt.tight_layout()
plt.show()
```

```
                    Majority Voting  Average Voting  Maximum Voting
Metric
Mean Squared Error         0.000004        0.000005        0.000028
Training R² Score          1.000000        1.000000        0.999600
Test R² Score              0.999800        0.999700        0.998500
F1 Score (YES class)       1.000000        1.000000        0.960000
```



	Majority Voting	Average Voting	Maximum Voting
Mean Squared Error	4.204e-06	4.72e-06	2.8353e-05
Training R² Score	1.0	1.0	0.9996
Test R² Score	0.9998	0.9997	0.9985
F1 Score (YES class)	1.0	1.0	0.96

Summary:

1. Majority and Average Voting both yield perfect metrics
2. Maximum Voting slightly drops in regression performance and classification precision for the YES class, suggesting a higher false positive rate.
3. Based on interpretability and use case:
   - If the goal is classification and robustness, Majority Voting is a safe and interpretable choice.
   - If goal is to use regression output directly or valuing finer prediction nuance, Average Voting is slightly more informative.

In [ ]: