```python
# import some basic Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

https://archive.ics.uci.edu/dataset/464/superconductivty+data

```python
# loading the dataset
from google.colab import files
file = files.upload()
df = pd.read_csv('train.csv')
df.head()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving train.csv to train (1).csv
```

	number_of_elements	mean_atomic_mass	wtd_mean_atomic_mass	gmean_atomic_mass	wtd_gmean_atomic_mass	entropy_atomic_mass	wtd_entrop
0	4	88.944468	57.862692	66.361592	36.116612	1.181795	
1	5	92.729214	58.518416	73.132787	36.396602	1.449309	
2	4	88.944468	57.885242	66.361592	36.122509	1.181795	
3	4	88.944468	57.873967	66.361592	36.119560	1.181795	
4	4	88.944468	57.840143	66.361592	36.110716	1.181795	

5 rows × 82 columns

Data Cleaning & Preprocessing

```python
# Checking basic info and check for missing values
print("Dataset Info:")
df.info()
print("\nMissing Values:")
print(df.isnull().sum())
```

```
 15  entropy_fie                      21263 non-null  float64
 16  wtd_entropy_fie                  21263 non-null  float64
 17  range_fie                        21263 non-null  float64
 18  wtd_range_fie                    21263 non-null  float64
 19  std_fie                          21263 non-null  float64
 20  wtd_std_fie                      21263 non-null  float64
 21  mean_atomic_radius               21263 non-null  float64
 22  wtd_mean_atomic_radius           21263 non-null  float64
 23  gmean_atomic_radius              21263 non-null  float64
 24  wtd_gmean_atomic_radius          21263 non-null  float64
 25  entropy_atomic_radius            21263 non-null  float64
 26  wtd_entropy_atomic_radius        21263 non-null  float64
 27  range_atomic_radius              21263 non-null  int64
 28  wtd_range_atomic_radius          21263 non-null  float64
 50  wtd_std_ElectronAffinity         21263 non-null  float64
 51  mean_FusionHeat                  21263 non-null  float64
 52  wtd_mean_FusionHeat              21263 non-null  float64
 53  gmean_FusionHeat                 21263 non-null  float64
 54  wtd_gmean_FusionHeat             21263 non-null  float64
 55  entropy_FusionHeat               21263 non-null  float64
 56  wtd_entropy_FusionHeat           21263 non-null  float64
 57  range_FusionHeat                 21263 non-null  float64
 58  wtd_range_FusionHeat             21263 non-null  float64
 59  std_FusionHeat                   21263 non-null  float64
 60  wtd_std_FusionHeat               21263 non-null  float64
 61  mean_ThermalConductivity         21263 non-null  float64
 62  wtd_mean_ThermalConductivity     21263 non-null  float64
 63  gmean_ThermalConductivity        21263 non-null  float64
 64  wtd_gmean_ThermalConductivity    21263 non-null  float64
 65  entropy_ThermalConductivity      21263 non-null  float64
 66  wtd_entropy_ThermalConductivity  21263 non-null  float64
 67  range_ThermalConductivity        21263 non-null  float64
 68  wtd_range_ThermalConductivity    21263 non-null  float64
 69  std_ThermalConductivity          21263 non-null  float64
 70  wtd_std_ThermalConductivity      21263 non-null  float64
 71  mean_Valence                     21263 non-null  float64
 72  wtd_mean_Valence                 21263 non-null  float64
 73  gmean_Valence                    21263 non-null  float64
```

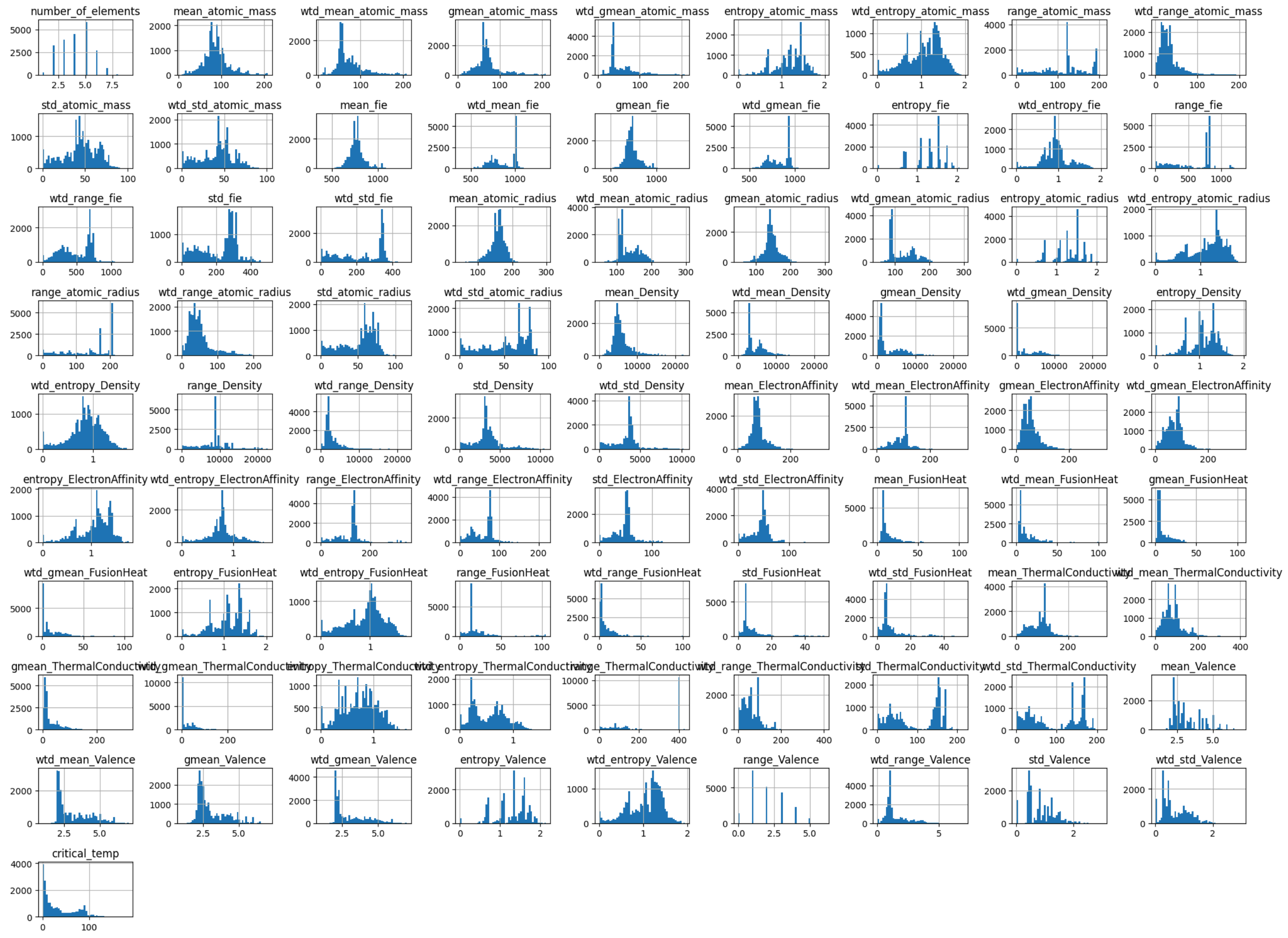
```
# Summary statistics
print("\nSummary Statistics:")
print(df.describe())
```

```
Summary Statistics:
       number_of_elements  mean_atomic_mass  wtd_mean_atomic_mass  \
count        21263.000000      21263.000000          21263.000000   
mean             4.115224         87.557631             72.988310   
std              1.439295         29.676497             33.490406   
min              1.000000          6.941000              6.423452   
25%              3.000000         72.458076             52.143839   
50%              4.000000         84.922750             60.696571   
75%              5.000000        100.404410             86.103540   
max              9.000000        208.980400            208.980400   

       gmean_atomic_mass  wtd_gmean_atomic_mass  entropy_atomic_mass  \
count       21263.000000           21263.000000         21263.000000   
mean           71.290627              58.539916             1.165608   
std            31.030272              36.651067             0.364930   
min             5.320573               1.960849             0.000000   
25%            58.041225              35.248990             0.966676   
50%            66.361592              39.918385             1.199541   
75%            78.116681              73.113234             1.444537   
max           208.980400             208.980400             1.983797   

       wtd_entropy_atomic_mass  range_atomic_mass  wtd_range_atomic_mass  \
count             21263.000000       21263.000000           21263.000000   
mean                  1.063884         115.601251              33.225218   
std                   0.401423          54.626887              26.967752   
min                   0.000000           0.000000               0.000000   
25%                   0.775363          78.512902              16.824174   
50%                   1.146783         122.906070              26.636008   
75%                   1.359418         154.119320              38.356908   
max                   1.958203         207.972460             205.589910   

       std_atomic_mass  ...  wtd_mean_Valence  gmean_Valence  \
count     21263.000000  ...      21263.000000   21263.000000   
mean         44.391893  ...          3.153127       3.056536   
std          20.035430  ...          1.191249       1.046257   
min           0.000000  ...          1.000000       1.000000   
25%          32.890369  ...          2.116732       2.279705   
50%          45.123500  ...          2.618182       2.615321   
75%          59.322812  ...          4.026201       3.727919   
max         101.019700  ...          7.000000       7.000000   

       wtd_gmean_Valence  entropy_Valence  wtd_entropy_Valence  range_Valence  \
count       21263.000000     21263.000000         21263.000000   21263.000000   
mean            3.055885         1.295682             1.052841       2.041010   
std             1.174815         0.393155             0.380291       1.242345   
min             1.000000         0.000000             0.000000       0.000000   
25%             2.091251         1.060857             0.775678       1.000000   
50%             2.434057         1.368922             1.166532       2.000000   
75%             3.914868         1.589027             1.330801       3.000000   
max             7.000000         2.141963             1.949739       6.000000   

       wtd_range_Valence   std_Valence  wtd_std_Valence  critical_temp  
count       21263.000000  21263.000000     21263.000000   21263.000000  
mean            1.483007      0.839342         0.673987      34.421219  
std             0.978176      0.484676         0.455580      34.254362  
```

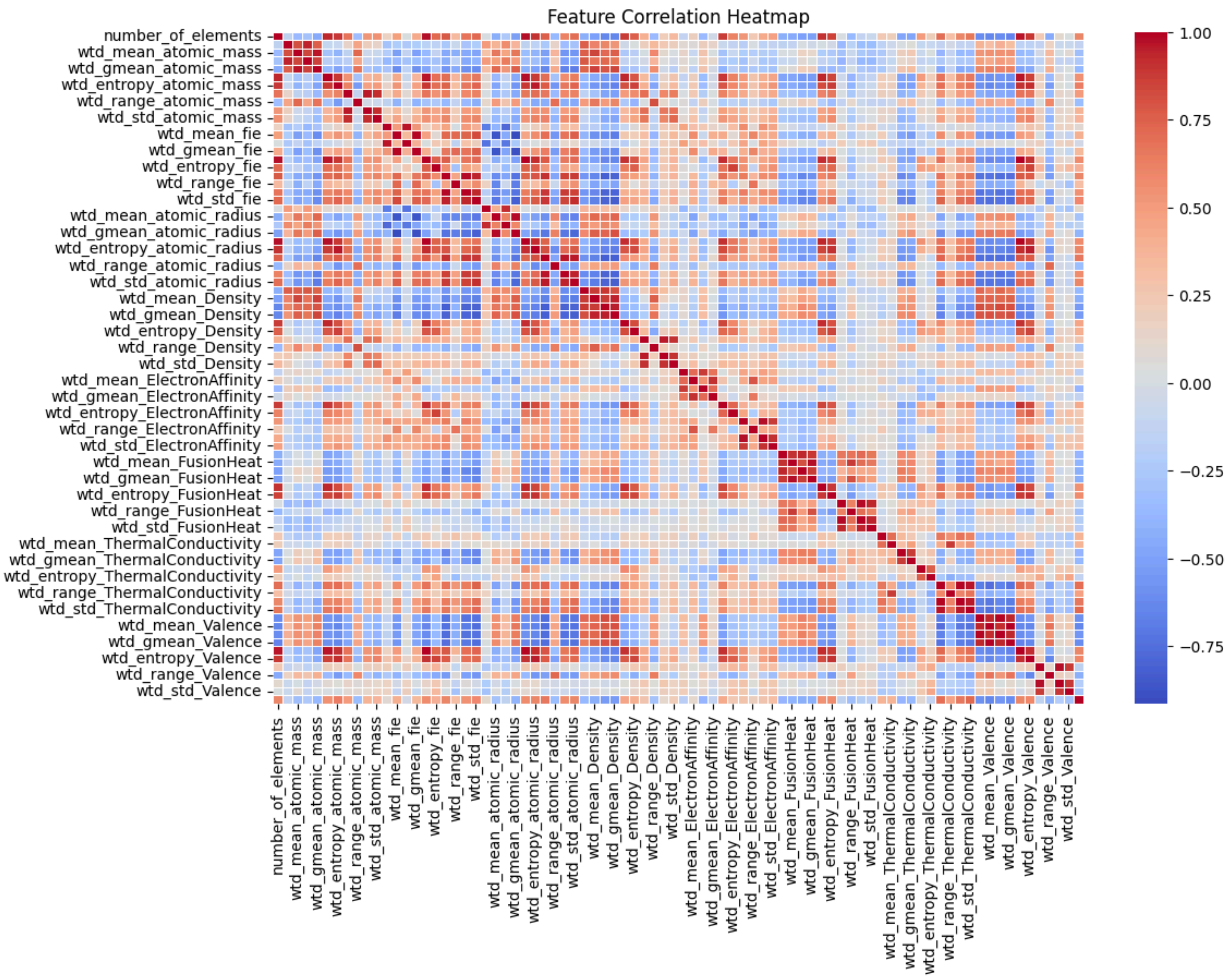
```
min              0.000000      0.000000         0.000000        0.000210
25%              0.921454      0.451754         0.306892        5.365000
```

```
# Plot histograms of features
plt.figure(figsize=(12, 8))
df.hist(bins=50, figsize=(20, 15))
plt.tight_layout()
plt.show()
```

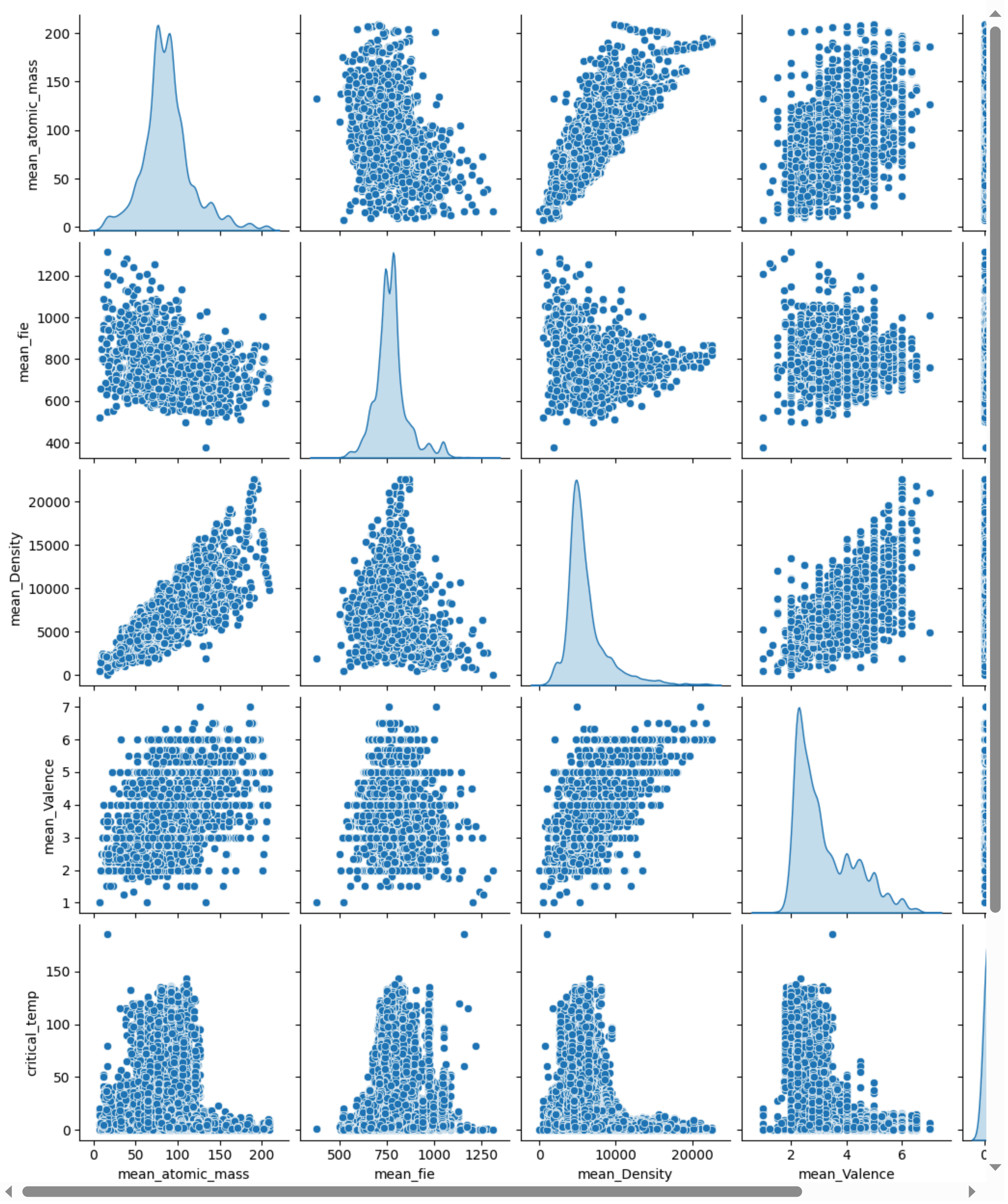
```
<Figure size 1200x800 with 0 Axes>
```



```
# Correlation matrix heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(df.corr(), cmap='coolwarm', annot=False, linewidths=0.5)
plt.title("Feature Correlation Heatmap")
plt.show()
```



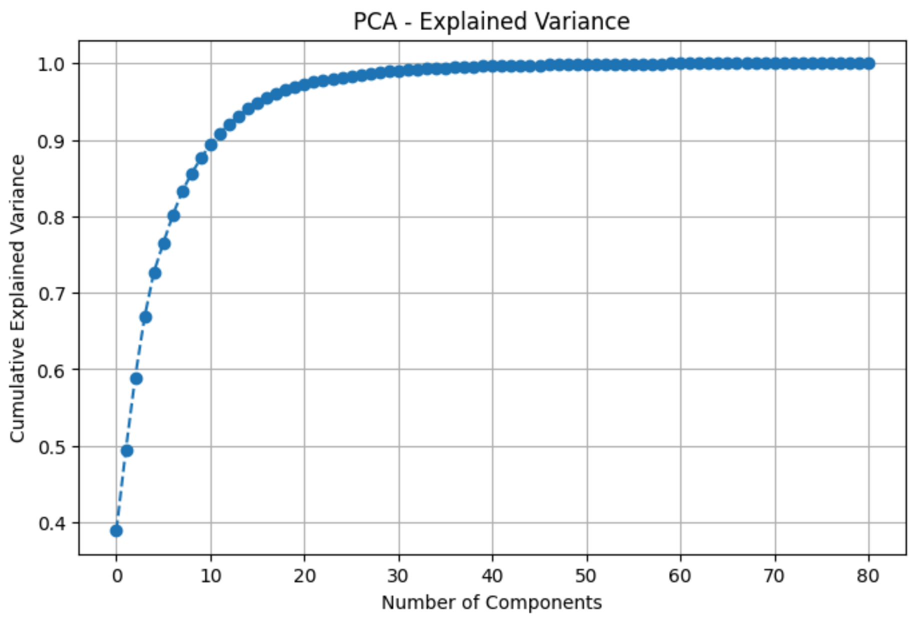
```
# Pair plot of selected key features
selected_features = ['mean_atomic_mass', 'mean_fie', 'mean_Density', 'mean_Valence', 'critical_temp']
sns.pairplot(df[selected_features], diag_kind='kde')
plt.show()
```



Principle Component Analysis

```
# Drop the target variable (critical_temp) since PCA is unsupervised
X = df.drop(columns=['critical_temp'])


# Standardizing the data
scaler = StandardScaler()
```

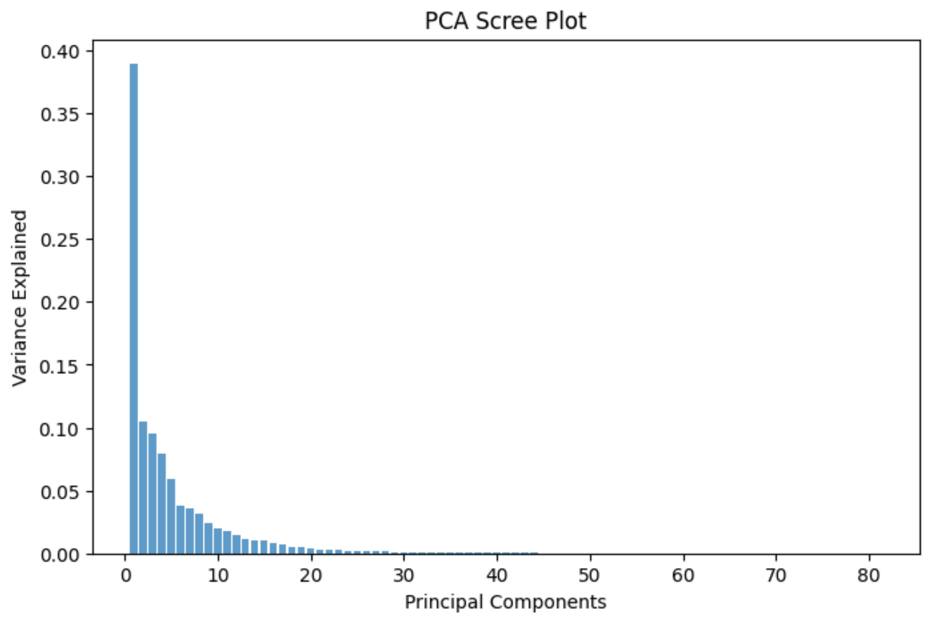
```
X_scaled = scaler.fit_transform(X)

# Convert back to DataFrame for better readability
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)


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

# Explained variance ratio
explained_variance = pca.explained_variance_ratio_

# Plot cumulative explained variance
plt.figure(figsize=(8, 5))
plt.plot(np.cumsum(explained_variance), marker='o', linestyle='--')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('PCA - Explained Variance')
plt.grid()
plt.show()
```



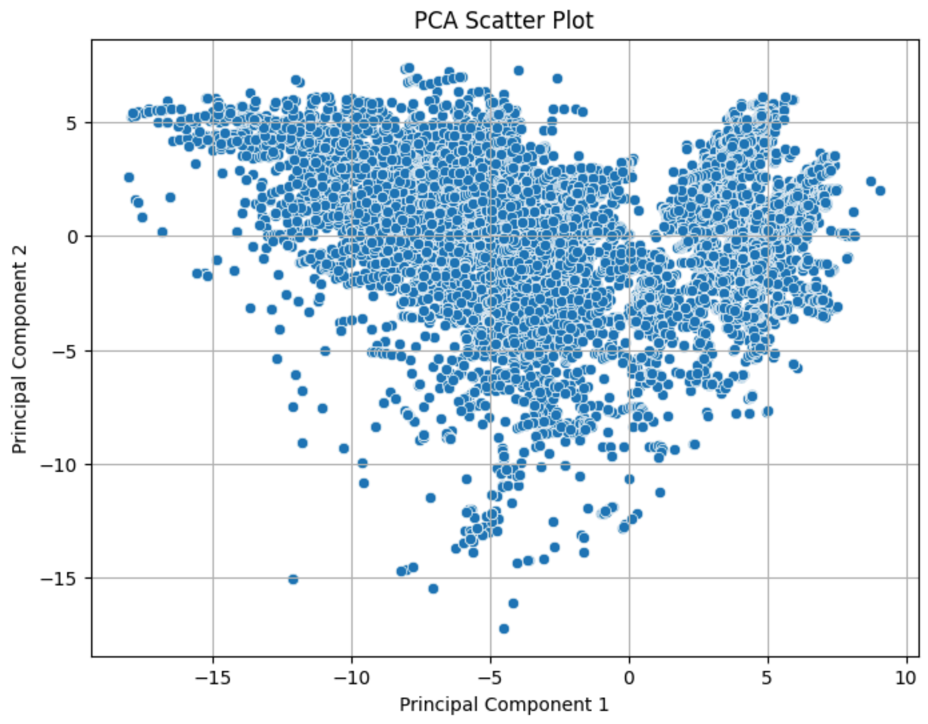
```
# Select number of components (e.g., 95% variance)
num_components = np.argmax(np.cumsum(explained_variance) >= 0.95) + 1
print(f"Optimal number of components: {num_components}")

# Apply PCA with selected components
pca_final = PCA(n_components=num_components)
X_pca_final = pca_final.fit_transform(X_scaled)

# Convert PCA results into a DataFrame
X_pca_df = pd.DataFrame(X_pca_final, columns=[f'PC{i+1}' for i in range(num_components)])
```

```
Optimal number of components: 17
```

```
plt.figure(figsize=(8, 5))
plt.bar(range(1, len(explained_variance) + 1), explained_variance, alpha=0.7)
plt.xlabel('Principal Components')
plt.ylabel('Variance Explained')
plt.title('PCA Scree Plot')
plt.show()
```

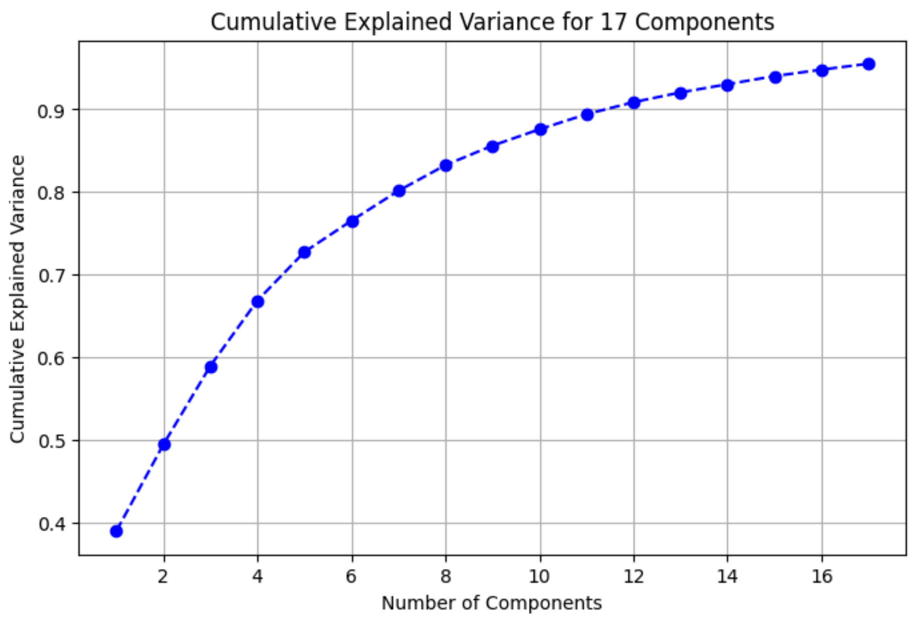


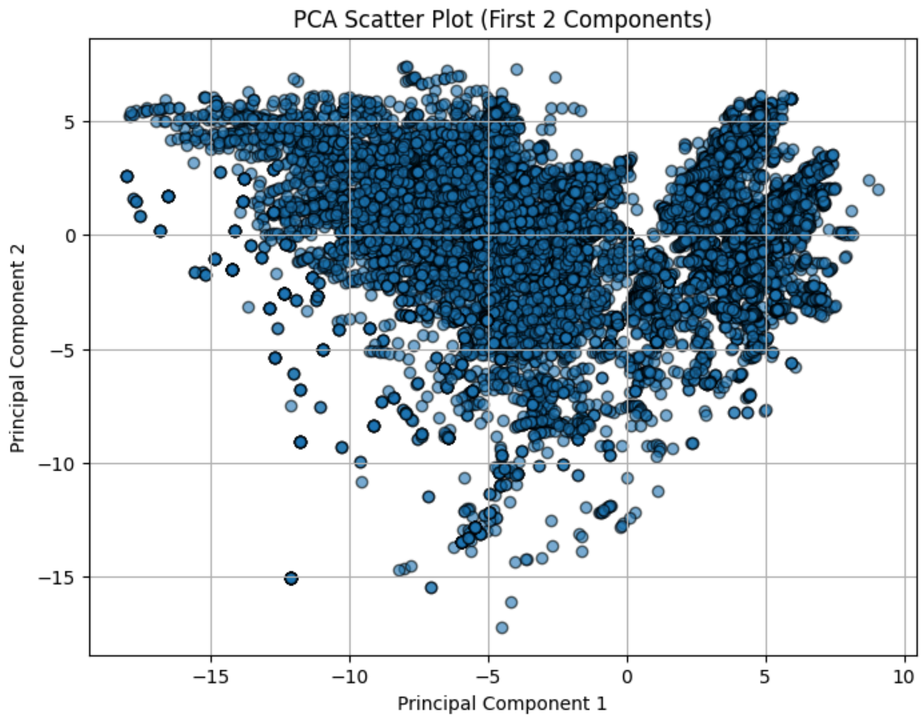
```
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_pca_df['PC1'], y=X_pca_df['PC2'])
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA Scatter Plot')
plt.grid()
plt.show()
```



```
# Drop the target variable (critical_temp)
X = df.drop(columns=['critical_temp'])

# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA with optimal 17 components
pca = PCA(n_components=17)
X_pca = pca.fit_transform(X_scaled)
```

```
# Explained variance ratio for 17 components
explained_variance = pca.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance)

# Plot cumulative explained variance
plt.figure(figsize=(8, 5))
plt.plot(range(1, 18), cumulative_variance, marker='o', linestyle='--', color='b')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Cumulative Explained Variance for 17 Components')
plt.grid()
plt.show()

# Scatter plot for first two principal components
plt.figure(figsize=(8, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], alpha=0.6, edgecolors='k')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA Scatter Plot (First 2 Components)')
plt.grid()
plt.show()
```





Linear Regression Model vs Random Forest for Model Performance

```python
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Step 1: Define target variable and features
y = df['critical_temp']
X = df.drop(columns=['critical_temp'])

# Step 2: Apply PCA to features (X)
# (Assuming PCA has already been applied to X, resulting in X_pca)

# Step 3: Split the PCA-transformed data
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=42)

# Step 4: Train your model (e.g., Linear Regression)


# Step 2: Train a Linear Regression Model
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Step 3: Make Predictions
y_pred = lin_reg.predict(X_test)

# Step 4: Evaluate Model Performance
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Linear Regression Model Performance:")
print(f"📉 Mean Squared Error (MSE): {mse:.2f}")
print(f"📊 R² Score: {r2:.4f}")

# Step 5: Try a More Advanced Model - Random Forest
rf_reg = RandomForestRegressor(n_estimators=100, random_state=42)
rf_reg.fit(X_train, y_train)

# Predictions
y_pred_rf = rf_reg.predict(X_test)

# Evaluate
mse_rf = mean_squared_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)

print(f"\nRandom Forest Model Performance:")
print(f"📉 Mean Squared Error (MSE): {mse_rf:.2f}")
print(f"📊 R² Score: {r2_rf:.4f}")
```

```
Linear Regression Model Performance:
📉 Mean Squared Error (MSE): 446.48
📊 R² Score: 0.6121

Random Forest Model Performance:
📉 Mean Squared Error (MSE): 95.39
📊 R² Score: 0.9171
```

Summary of the Analysis:

- The Random Forest model significantly outperforms the Linear Regression model, with a lower MSE and a much higher R² score.
- This suggests that Random Forest is better at capturing complex patterns and relationships in the data.
- The Linear Regression model, while decent, is less effective in predicting the target variable, as it doesn't capture as much of the variance compared to Random Forest.

```python
from sklearn.model_selection import RandomizedSearchCV

# Define the Random Forest model
rf_model = RandomForestRegressor(random_state=42)
```

```
# Define the parameter distribution for randomized search
param_dist = {
    'n_estimators': [100, 200, 300, 400],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10]
}

# Perform randomized search with n_iter set to a smaller value to speed up
random_search = RandomizedSearchCV(rf_model, param_distributions=param_dist, n_iter=10, cv=3, n_jobs=-1, random_state=42)
random_search.fit(X_train, y_train)

# Get the best model and print the results
print("Best Parameters:", random_search.best_params_)
rf_best = random_search.best_estimator_

# Evaluate the best model
y_pred_rf = rf_best.predict(X_test)
mse_rf = mean_squared_error(y_test, y_pred_rf)
r2_rf = r2_score(y_test, y_pred_rf)

print(f"Random Forest Test MSE: {mse_rf}, R²: {r2_rf}")
```

```
Best Parameters: {'n_estimators': 400, 'min_samples_split': 2, 'max_depth': 30}
Random Forest Test MSE: 94.31938522021353, R²: 0.9180601278578859
```

The results of the RandomizedSearchCV indicate that the best hyperparameters for your Random Forest model are:

- n_estimators: 400 (number of trees in the forest)
- min_samples_split: 2 (the minimum number of samples required to split an internal node)
- max_depth: 30 (the maximum depth of the trees) Model Performance:
- Test MSE (Mean Squared Error): 94.32
- R^2 (R-squared): 0.9181

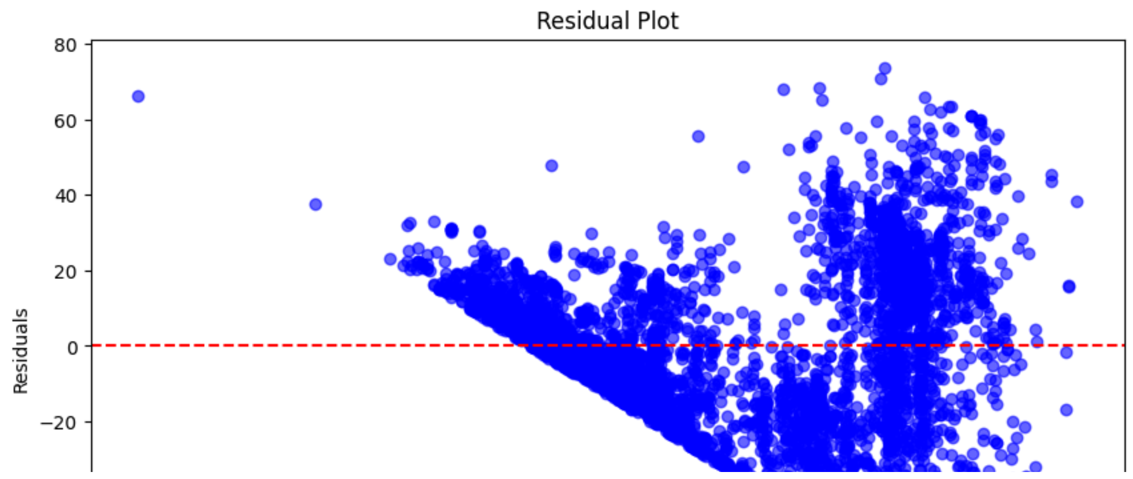
Interpretation of the Results:

- MSE (94.32): A relatively low MSE suggests that the model's predictions are fairly close to the actual values. Since the MSE is based on squared errors, a lower MSE means better model performance.
- R^2 (0.9181): This value indicates that the model explains approximately 91.81% of the variance in the critical temperature of superconductors. This is a strong result, as an R^2 score close to 1 means the model is doing an excellent job of fitting the data.

Residual Plot

- This plot helps you understand how well your model's predictions match the actual values. Ideally, the residuals should be scattered randomly around zero, which suggests no patterns are left to be captured by the model.

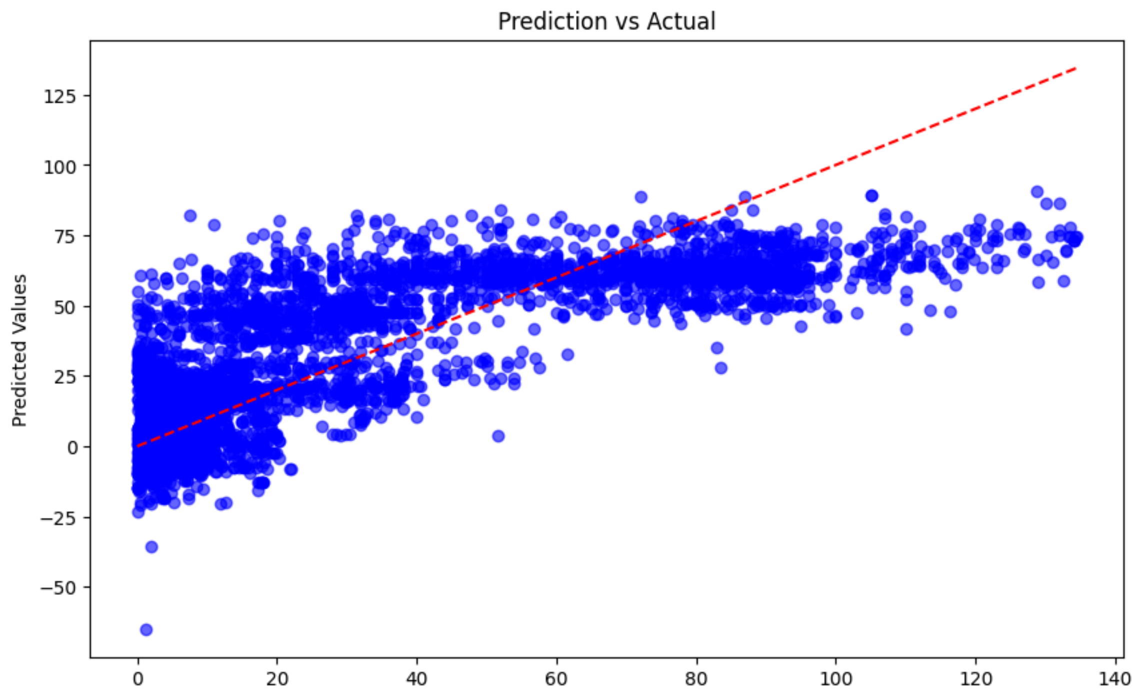
```
import matplotlib.pyplot as plt

# Calculate residuals
residuals = y_test - y_pred

# Plotting Residuals
plt.figure(figsize=(10,6))
plt.scatter(y_pred, residuals, color='blue', alpha=0.6)
plt.axhline(y=0, color='red', linestyle='--')  # Line at 0 to show residuals centered
plt.title('Residual Plot')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.show()
```



Prediction vs. Actual Plot

- This plot compares the predicted values with the actual values. A diagonal line represents perfect predictions (i.e., predicted = actual), and the points should ideally be close to this line.

```
plt.figure(figsize=(10,6))
plt.scatter(y_test, y_pred, color='blue', alpha=0.6)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linestyle='--')  # Diagonal line for perfect predictions
plt.title('Prediction vs Actual')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.show()
```


Prediction vs Actual
Predicted Values