cytoautocluster-2

November 14, 2024

[65]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np


df = pd.read_csv(r"C:\Users\Jai dabas\Downloads\Infosys internship\Levine_32dim.
 ↪csv")
df
```

```
C:\Users\Jai dabas\AppData\Local\Temp\ipykernel_25196\2301157716.py:7:
DtypeWarning: Columns (39) have mixed types. Specify dtype option on import or
set low_memory=False.
  df = pd.read_csv(r"C:\Users\Jai dabas\Downloads\Infosys
internship\Levine_32dim.csv")
```

[65]:

	Time	Cell_length	DNA1	DNA2	CD45RA	CD133	\
0	2693.0000	22	4.391057	4.617262	0.162691	-0.029585	
1	3736.0000	35	4.340481	4.816692	0.701348	-0.038280	
2	7015.0000	32	3.838727	4.386369	0.603568	-0.032216	
3	7099.0000	29	4.255805	4.830048	0.433747	-0.027611	
4	7700.0000	25	3.976909	4.506433	-0.008809	-0.030297	
…	…	…	…	…	…	…	
265621	707917.4375	60	6.733888	7.179924	1.901087	-0.054719	
265622	707951.4375	41	6.826629	7.133022	1.474081	-0.019174	
265623	708145.4375	45	6.787791	7.154027	0.116755	-0.056213	
265624	708398.4375	41	6.889866	7.141219	0.684921	-0.006264	
265625	708585.4375	39	6.865218	7.144353	0.288761	-0.011310	

	CD19	CD22	CD11b	CD4	…	CD117	CD49d	\
0	-0.006696	0.066388	-0.009184	0.363602	…	0.053050	0.853505	
1	-0.016654	0.074409	0.808031	-0.035424	…	0.089660	0.197818	
2	0.073855	-0.042977	-0.001881	-0.008781	…	0.046222	2.586670	
3	-0.017661	-0.044072	0.733698	-0.019066	…	0.066470	1.338669	
4	0.080423	0.495791	1.107627	0.552746	…	-0.006223	0.180924	
…	…	…	…	…	…	…	…	
265621	3.127012	2.389596	0.212047	0.003287	…	-0.043032	0.069388	
265622	-0.055620	-0.007261	0.063395	0.145304	…	-0.011105	0.533736	

```
265623 -0.008864 -0.035158 -0.041845  0.970120  …  0.143869  1.269464
265624 -0.026111 -0.030837 -0.034641  1.597189  …  0.087102 -0.055912
265625 -0.048786  0.073983 -0.031787  0.078800  … -0.047971  0.101955

          HLA-DR      CD64      CD41   Viability   file_number   event_number  \
0       1.664480 -0.005376 -0.001961    0.648429      3.627711            307
1       0.491592  0.144814  0.868014    0.561384      3.627711            545
2       1.308337 -0.010961 -0.010413    0.643337      3.627711           1726
3       0.140523 -0.013449 -0.026039   -0.026523      3.627711           1766
4       0.197332  0.076167 -0.040488    0.283287      3.627711           2031
…            …         …         …           …             …              …
265621  3.550516  0.147588 -0.043806    0.144479      3.669327         102685
265622  0.123758 -0.042495 -0.027971    0.236957      3.669327         102686
265623  0.047215 -0.008000 -0.025811   -0.003500      3.669327         102690
265624  0.501536  0.053884 -0.042602    0.107206      3.669327         102701
265625  6.200001  0.296877  0.192786    0.620872      3.669327         102706

         label   individual
0            1            1
1            1            1
2            1            1
3            1            1
4            1            1
…          …            …
265621     NaN            2
265622     NaN            2
265623     NaN            2
265624     NaN            2
265625     NaN            2

[265626 rows x 41 columns]
```

[66]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load the dataset (already with NaN values identified)
df = pd.read_csv(r"C:\Users\Jai dabas\Downloads\Infosys internship\Levine_32dim.
 ↪csv", na_values=['NA', 'N/A', '', 'unknown', ' '])

# Strip any leading/trailing whitespaces in the entire DataFrame
df = df.apply(lambda x: x.str.strip() if x.dtype == "object" else x)

# Calculate the number of null and non-null values per column
null_values_sum = df.isnull().sum()
nonnull_values_sum = df.notnull().sum()
```

```
# Create indices for the columns
indices = np.arange(len(df.columns))

# Set the height of bars (which controls width in horizontal bar charts)
bar_height = 0.5  # Changed from bar_width to bar_height

# Increase figure size for better readability
plt.figure(figsize=(10, 8))  # You can adjust the figure size

# Plot stacked bars for null and non-null values
plt.barh(indices, nonnull_values_sum, height=bar_height, color='green',␣
 ↪label='Non-Null')
plt.barh(indices, null_values_sum, height=bar_height, left=nonnull_values_sum,␣
 ↪color='red', label='Null')

# Add labels, title, and legend
plt.xlabel('Count')
plt.ylabel('Columns')
plt.title('Null and Non-Null Values per Column')

# Reduce the rotation angle for readability and adjust alignment
plt.yticks(indices, df.columns, fontsize=10)  # You can reduce or increase␣
 ↪fontsize as needed

plt.legend()

# Display the plot with tight layout to avoid overlapping
plt.tight_layout()
plt.show()
```

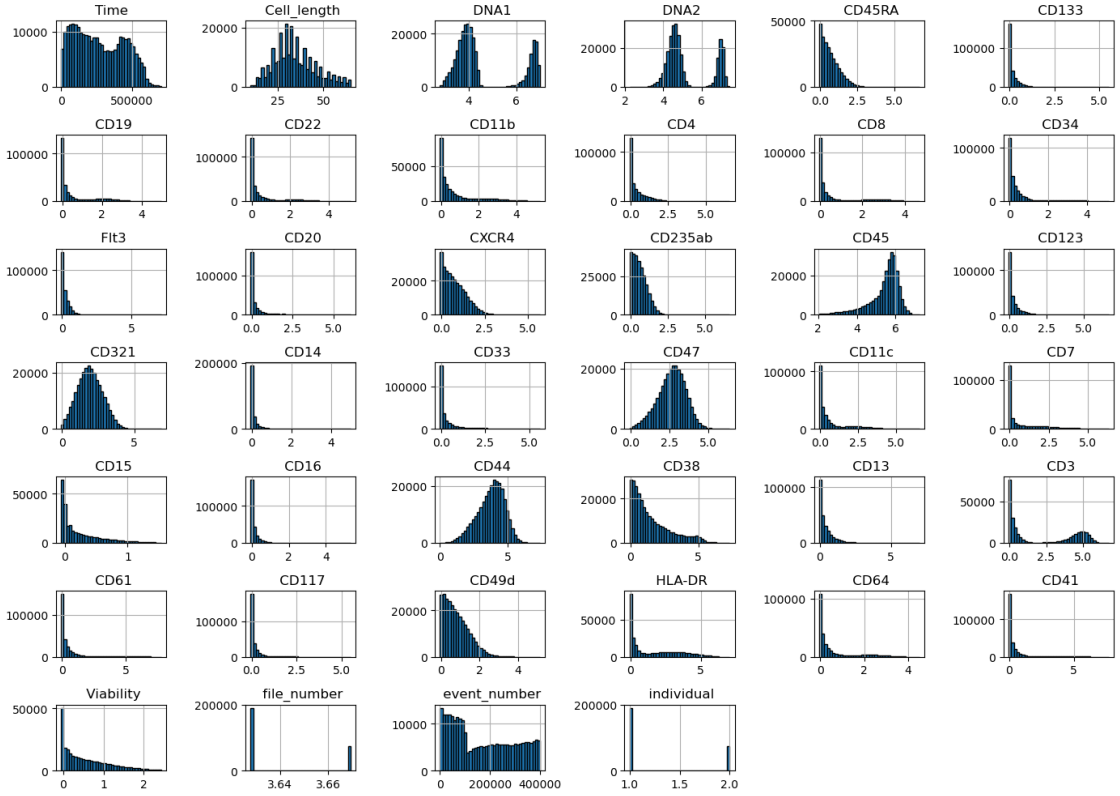
```
C:\Users\Jai dabas\AppData\Local\Temp\ipykernel_25196\626664001.py:6:
DtypeWarning: Columns (39) have mixed types. Specify dtype option on import or
set low_memory=False.
  df = pd.read_csv(r"C:\Users\Jai dabas\Downloads\Infosys
internship\Levine_32dim.csv", na_values=['NA', 'N/A', '', 'unknown', ' '])
```

[58]:
```
# Display the range (min, max) for each numerical feature
feature_ranges = df.describe().loc[['min', 'max']]
print(feature_ranges)
```

```
            Time   Cell_length      DNA1      DNA2    CD45RA     CD133  \
min       1.0000          10.0  2.786488  2.236450 -0.057305 -0.058081
max  708585.4375          65.0  7.001489  7.472308  6.691197  5.527494

         CD19      CD22      CD11b       CD4  …      CD61     CD117  \
min -0.058089 -0.057342 -0.058236 -0.057751  … -0.057642 -0.057668
max  4.990085  5.160477  5.260789  6.581762  …  7.748497  5.502125

        CD49d    HLA-DR      CD64      CD41   Viability   file_number  \
min -0.058064 -0.057974 -0.058199 -0.058244   -0.057979      3.627711
max  5.153438  7.052507  4.517843  7.718288    2.433031      3.669327

      event_number   individual
min            1.0          1.0
max       400112.0          2.0
```

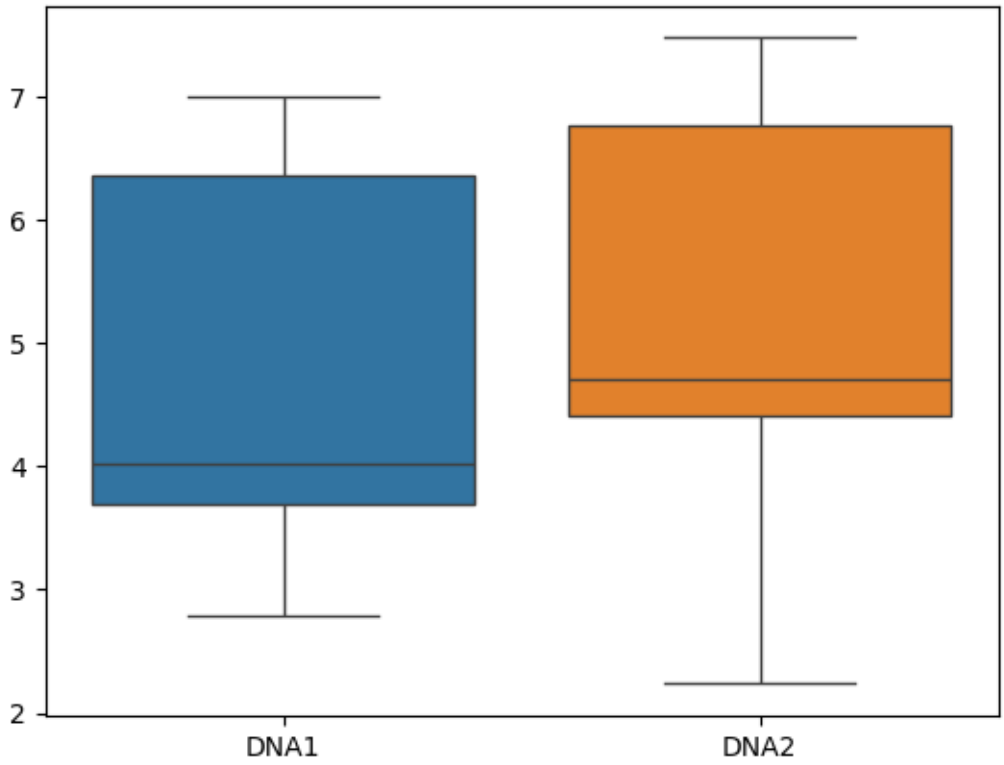
[2 rows x 40 columns]

[59]:
```
# Plot histograms for all numerical columns
df.hist(figsize=(14, 10), bins=38, edgecolor='black')
plt.tight_layout()
plt.show()
```

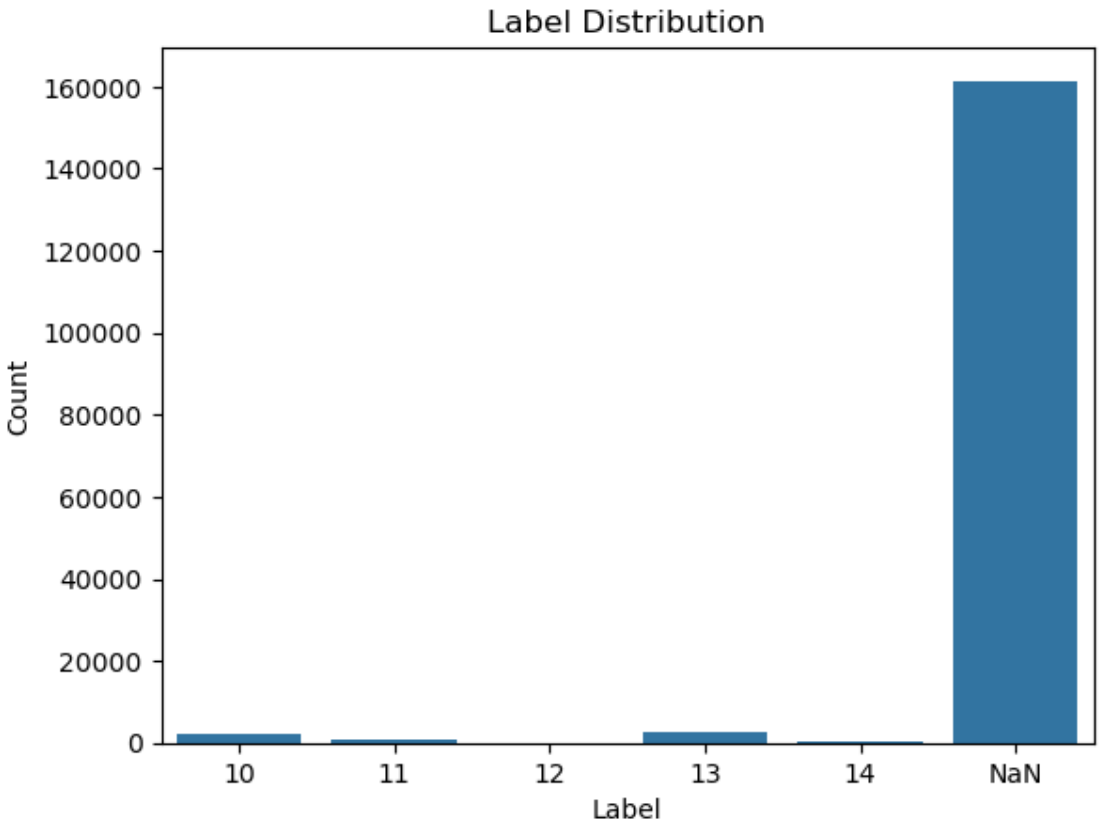


[67]:
```
df.columns = df.columns.str.strip()
```

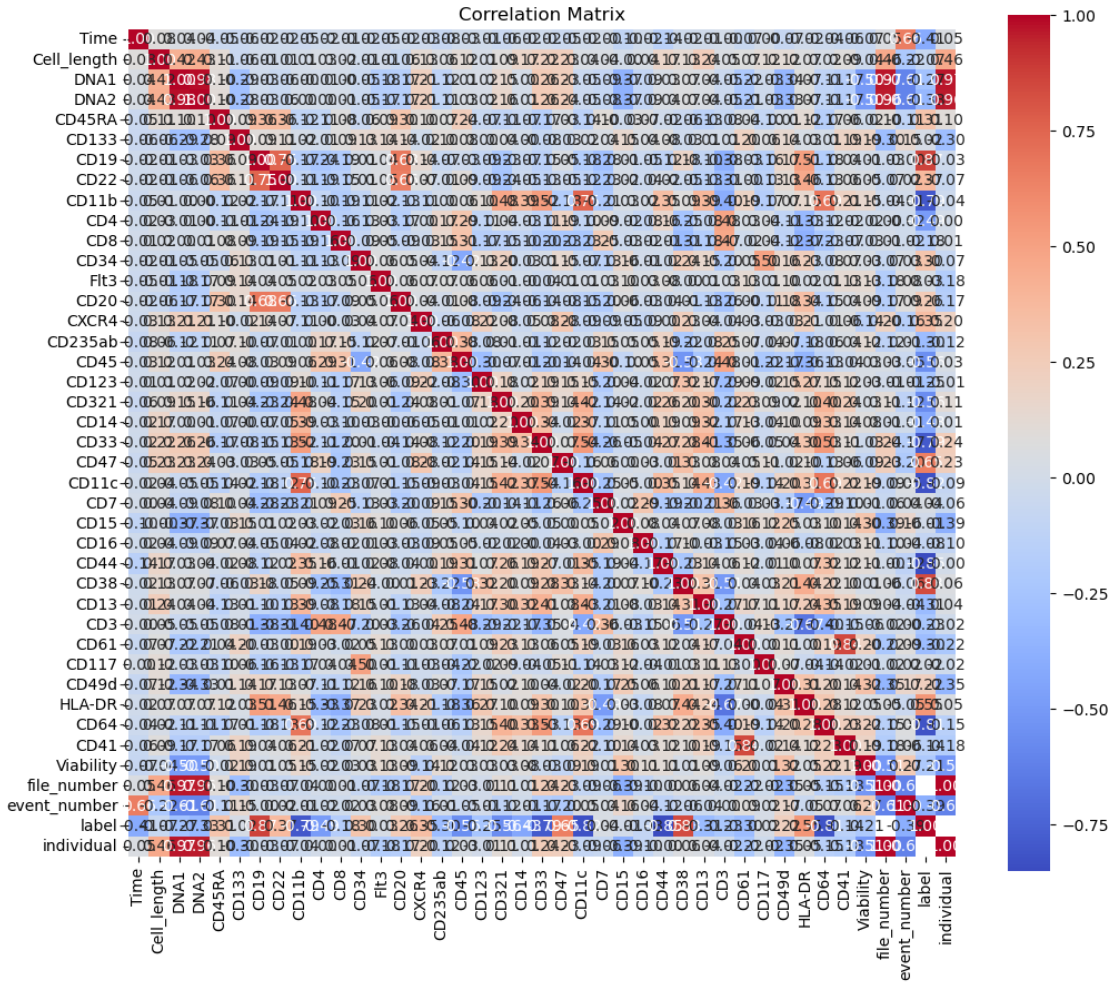
[68]:
```
import seaborn as sns

sns.boxplot(data=df[['DNA1', 'DNA2']])
plt.show()
```



[69]:
```
sns.countplot(data=df, x='label')
plt.title('Label Distribution')
plt.xlabel('Label')
plt.ylabel('Count')
plt.show()
```



[13]:
```
correlation_matrix = df.corr()

# Create a heatmap for the correlation matrix
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, fmt='.2f', cmap='coolwarm',␣
 ↪square=True)
plt.title('Correlation Matrix')
plt.show()
```

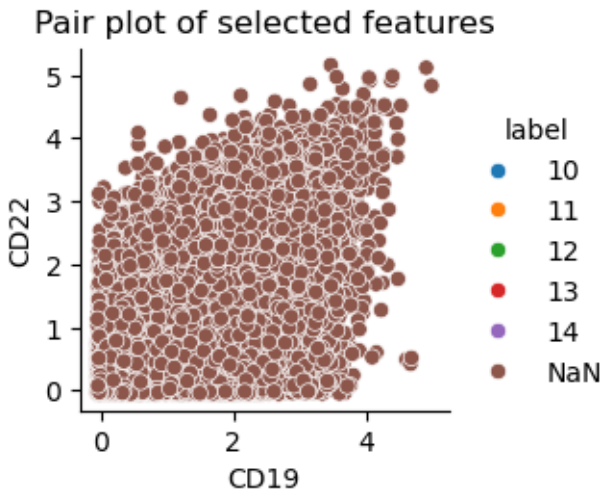


[35]:
```
# Remove specified columns from the DataFrame
columns_to_remove = ['Viability', 'file_number', 'event_number']
df.drop(columns=columns_to_remove, inplace=True)

# Display the updated DataFrame to verify the removal
print("Updated DataFrame:")
print(df.head())
```

```
Updated DataFrame:
     Time  Cell_length      DNA1      DNA2    CD45RA    CD133      CD19  \
0  2693.0           22  4.391057  4.617262  0.162691 -0.029585 -0.006696
1  3736.0           35  4.340481  4.816692  0.701348 -0.038280 -0.016654
2  7015.0           32  3.838727  4.386369  0.603568 -0.032216  0.073855
3  7099.0           29  4.255805  4.830048  0.433747 -0.027611 -0.017661
4  7700.0           25  3.976909  4.506433 -0.008809 -0.030297  0.080423

       CD22      CD11b       CD4  …      CD13       CD3      CD61     CD117  \
```

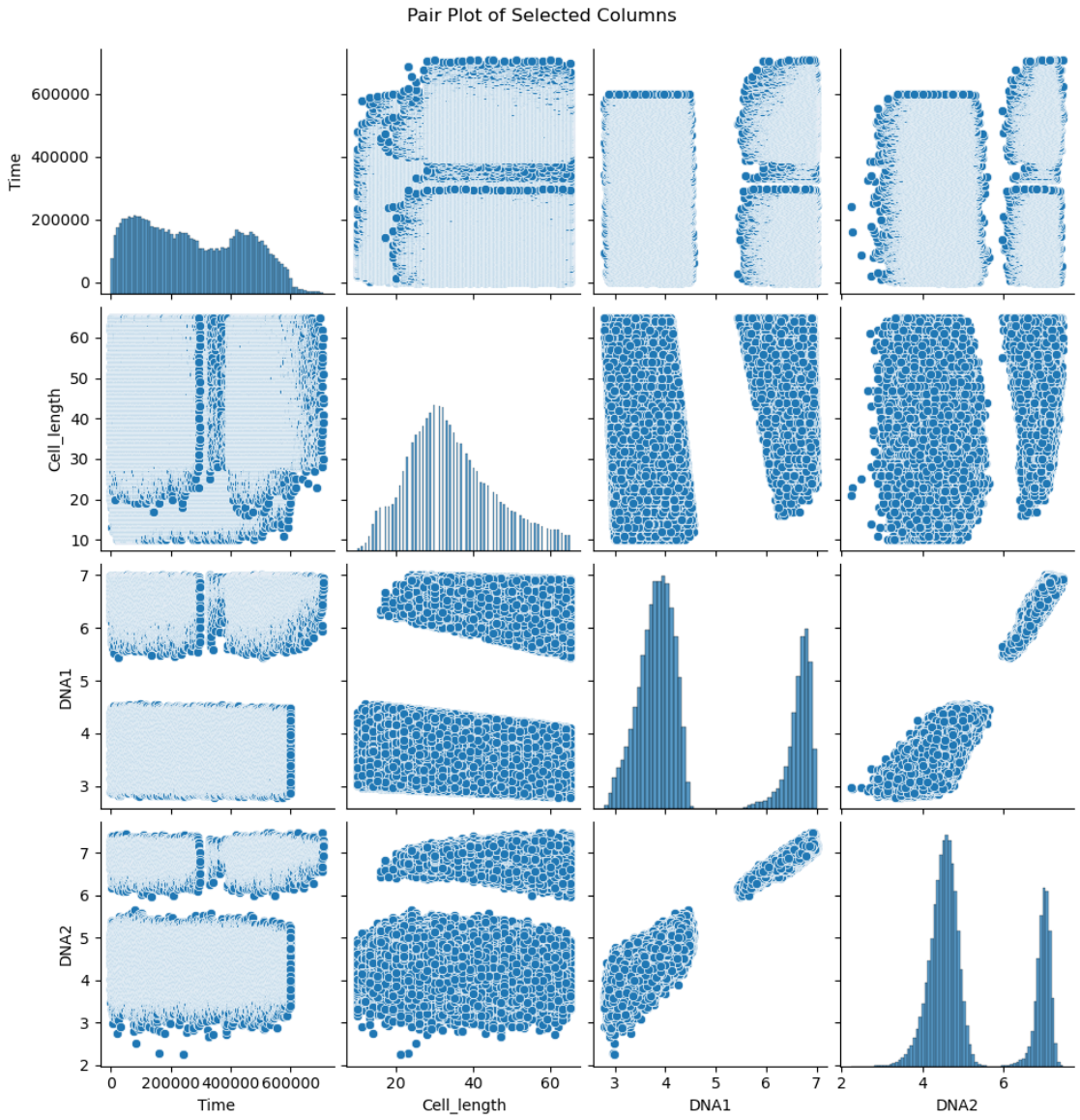
```
0  0.066388 -0.009184  0.363602  …  0.038552 -0.032596 -0.002936  0.053050
1  0.074409  0.808031 -0.035424  …  1.457326 -0.043466  1.258437  0.089660
2 -0.042977 -0.001881 -0.008781  …  0.213583  0.320792  0.257137  0.046222
3 -0.044072  0.733698 -0.019066  …  0.514349  0.060443 -0.041140  0.066470
4  0.495791  1.107627  0.552746  …  0.585712  0.137186  0.168609 -0.006223

      CD49d    HLA-DR      CD64      CD41  label  individual
0  0.853505  1.664480 -0.005376 -0.001961    NaN           1
1  0.197818  0.491592  0.144814  0.868014    NaN           1
2  2.586670  1.308337 -0.010961 -0.010413    NaN           1
3  1.338669  0.140523 -0.013449 -0.026039    NaN           1
4  0.180924  0.197332  0.076167 -0.040488    NaN           1

[5 rows x 38 columns]
```

[36]:
```
import seaborn as sns
sns.pairplot(df,hue='label',x_vars=['CD19'],y_vars=['CD22'])
plt.title('Pair plot of selected features')
plt.show()
```



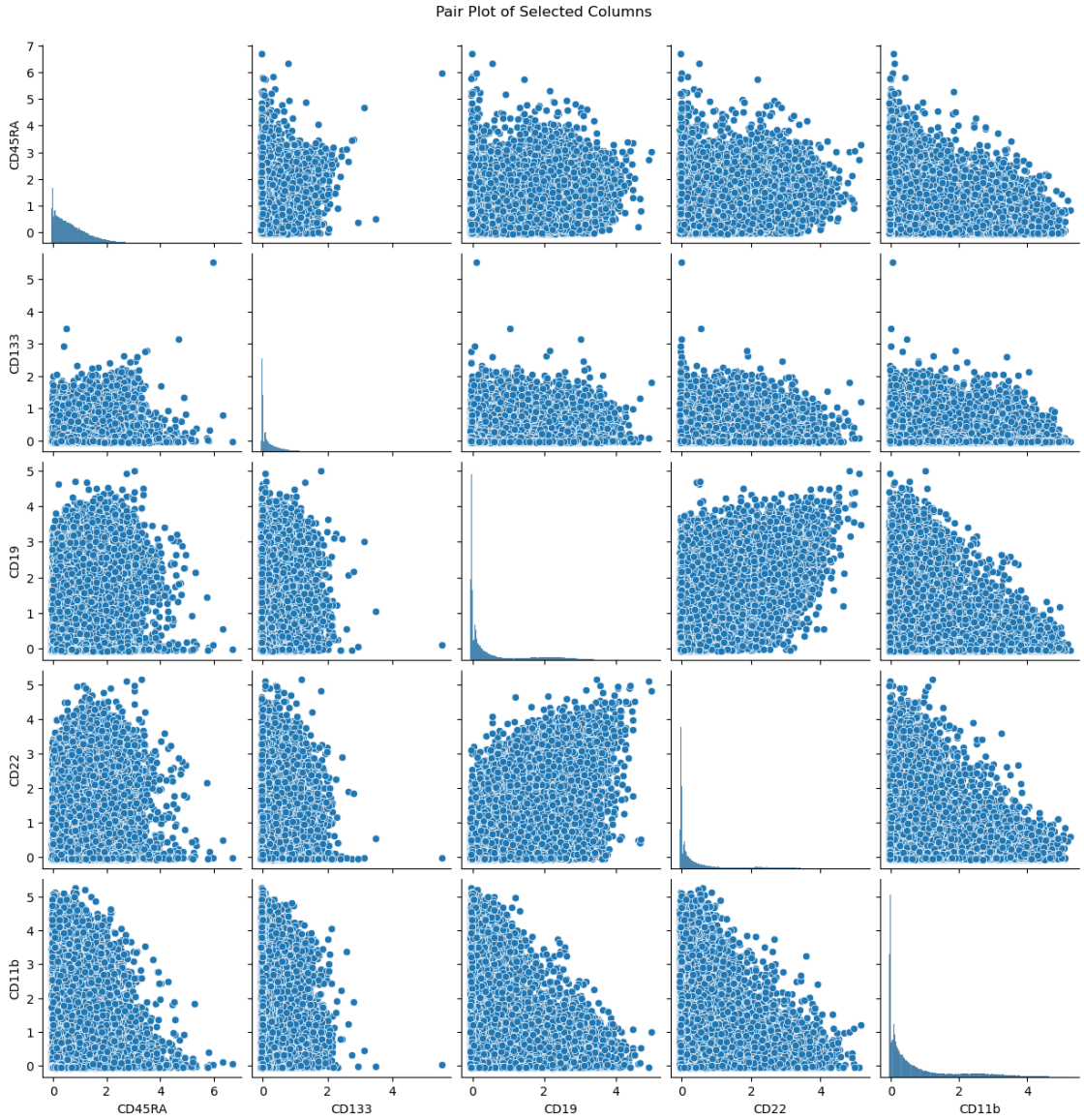
[38]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Specify the columns for the pair plot
pairplot_columns = [
    'Time', 'Cell_length', 'DNA1', 'DNA2'
]

# Create a pair plot for the specified columns
sns.pairplot(df[pairplot_columns])
```

```
plt.suptitle('Pair Plot of Selected Columns', y=1.02)
plt.show()
```



[37]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Specify the columns for the pair plot
pairplot_columns = [
     'CD45RA', 'CD133',
    'CD19', 'CD22', 'CD11b',
]
```

```
# Create a pair plot for the specified columns
sns.pairplot(df[pairplot_columns])
plt.suptitle('Pair Plot of Selected Columns', y=1.02)
plt.show()
```



[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Specify the columns for the pair plot
pairplot_columns = [
     'CD4', 'CD117', 'CD49d',
    'HLA-DR', 'CD64', 'CD41'
```

```
]

# Create a pair plot for the specified columns
sns.pairplot(df[pairplot_columns])
plt.suptitle('Pair Plot of Selected Columns', y=1.02)
plt.show()
```

[ ]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Specify the columns for skewness calculation
skewness_columns = [
    'Time', 'Cell_length', 'DNA1', 'DNA2', 'CD45RA', 'CD133',
    'CD19', 'CD22', 'CD11b', 'CD4', 'CD117', 'CD49d',
    'HLA-DR', 'CD64', 'CD41'
]

# Calculate skewness for the specified columns
skewness_values = df[skewness_columns].skew()

# Print the skewness values
print("Skewness values for the selected columns:")
print(skewness_values)

# Plot the skewness values as a bar chart
plt.figure(figsize=(12, 6))
sns.barplot(x=skewness_values.index, y=skewness_values.values)
plt.axhline(0, color='red', linestyle='--', linewidth=1)  # Line at y=0 for␣
 ↪reference
plt.title('Skewness of Selected Columns')
plt.xlabel('Columns')
plt.ylabel('Skewness')
plt.xticks(rotation=45)
plt.show()
```

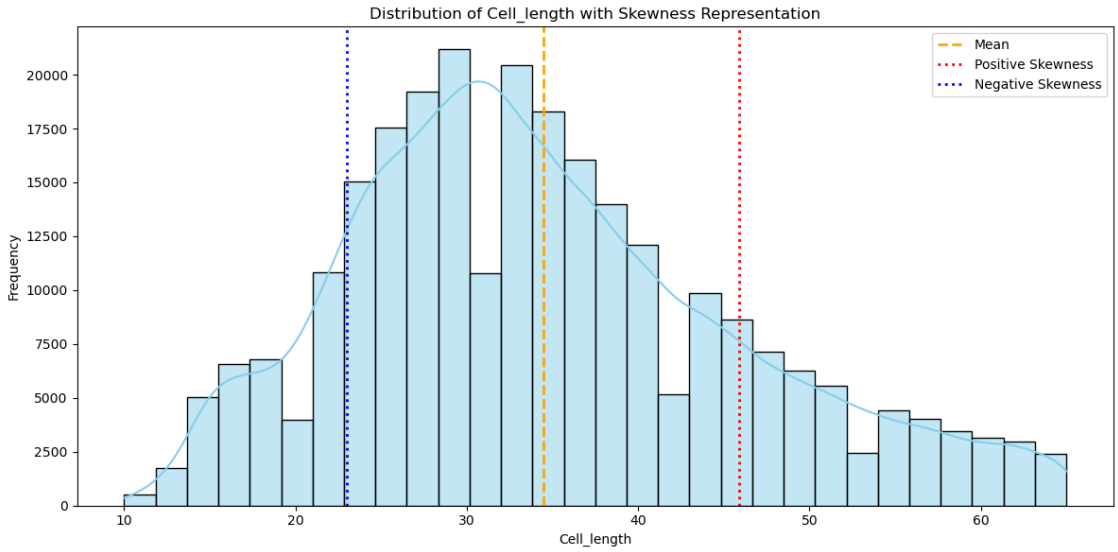
[ ]:
```
import pandas as pd
import matplotlib.pyplot as plt

# Define the specific columns to analyze
specific_columns = [
    'Time', 'Cell_length', 'DNA1', 'DNA2', 'CD45RA', 'CD133',
    'CD19', 'CD22', 'CD11b', 'CD4', 'CD117', 'CD49d',
    'HLA-DR', 'CD64', 'CD41'
]

# Filter the DataFrame to include only the specified columns
```

```
filtered_df = df[specific_columns]

# Calculate kurtosis for the specific columns
kurtosis_values = filtered_df.kurtosis()

# Display kurtosis values
print("Kurtosis values for specified columns:")
print(kurtosis_values)

# Plot the kurtosis values
plt.figure(figsize=(12, 6))
kurtosis_values.plot(kind='bar', color='skyblue')
plt.title('Kurtosis of Specified Columns')
plt.xlabel('Columns')
plt.ylabel('Kurtosis')
plt.axhline(0, color='red', linewidth=1, linestyle='--')  # Line at y=0 for␣
 ↪reference
plt.xticks(rotation=45)  # Rotate x-axis labels for better visibility
plt.tight_layout()  # Adjust layout to prevent clipping of labels
plt.show()
```

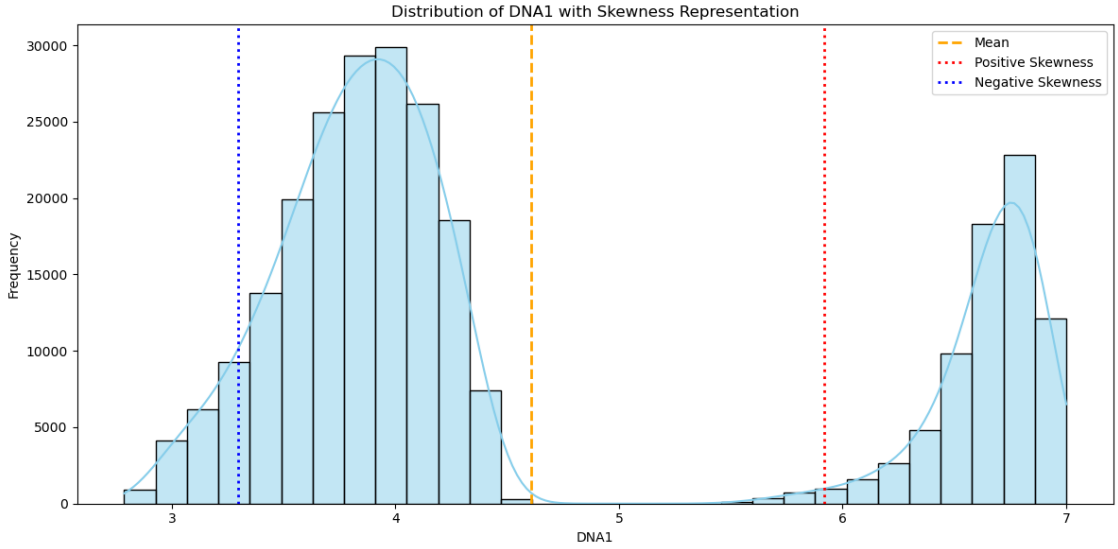
[39]:
```
# List of columns to analyze
columns_to_analyze = [
    'Time', 'Cell_length', 'DNA1', 'DNA2', 'CD45RA', 'CD133',
    'CD19', 'CD22', 'CD11b', 'CD4', 'CD117', 'CD49d',
    'HLA-DR', 'CD64', 'CD41'
]

# Create a function to plot skewness
def plot_skewness(column):
    # Calculate skewness for the column
    skewness = df[column].skew()

    # Create a histogram for the column
    plt.figure(figsize=(12, 6))
    sns.histplot(df[column], bins=30, kde=True, color='skyblue')

    # Add vertical line for mean
    plt.axvline(df[column].mean(), color='orange', linestyle='dashed',␣
 ↪linewidth=2, label='Mean')

    # Add vertical lines to represent skewness
    if skewness > 0:
        plt.axvline(df[column].mean() + 1 * df[column].std(), color='red',␣
 ↪linestyle='dotted', linewidth=2, label='Positive Skewness')
        plt.axvline(df[column].mean() - 1 * df[column].std(), color='blue',␣
 ↪linestyle='dotted', linewidth=2, label='Negative Skewness')
```

```
    else:
        plt.axvline(df[column].mean() - 1 * df[column].std(), color='red',␣
↪linestyle='dotted', linewidth=2, label='Negative Skewness')
        plt.axvline(df[column].mean() + 1 * df[column].std(), color='blue',␣
↪linestyle='dotted', linewidth=2, label='Positive Skewness')

    # Set title and labels
    plt.title(f'Distribution of {column} with Skewness Representation')
    plt.xlabel(column)
    plt.ylabel('Frequency')

    # Show legend
    plt.legend()

    # Show plot
    plt.tight_layout()
    plt.show()

    # Print skewness value
    print(f"Skewness of {column}: {skewness:.4f}")

# Loop through the columns and plot skewness for each
for col in columns_to_analyze:
    plot_skewness(col)
```

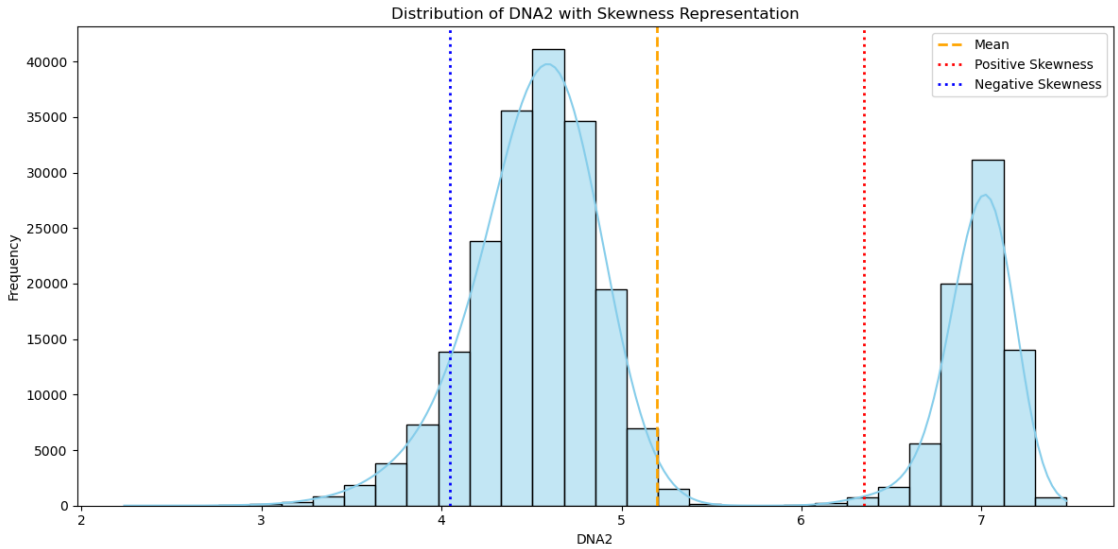

```
Skewness of Time: 0.2199
```



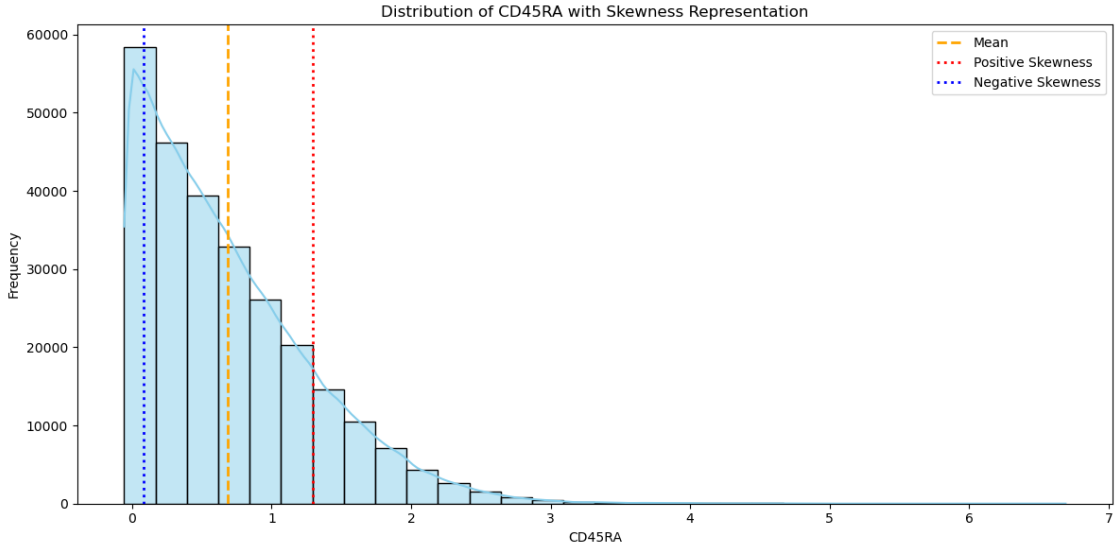

```
Skewness of Cell_length: 0.5278
```



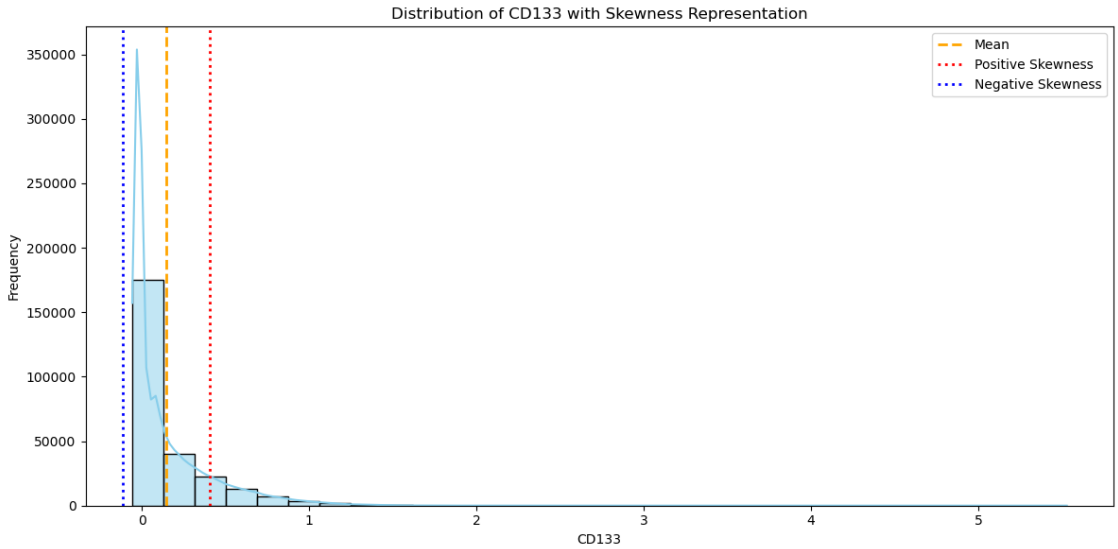

```
Skewness of DNA1: 0.8450
```



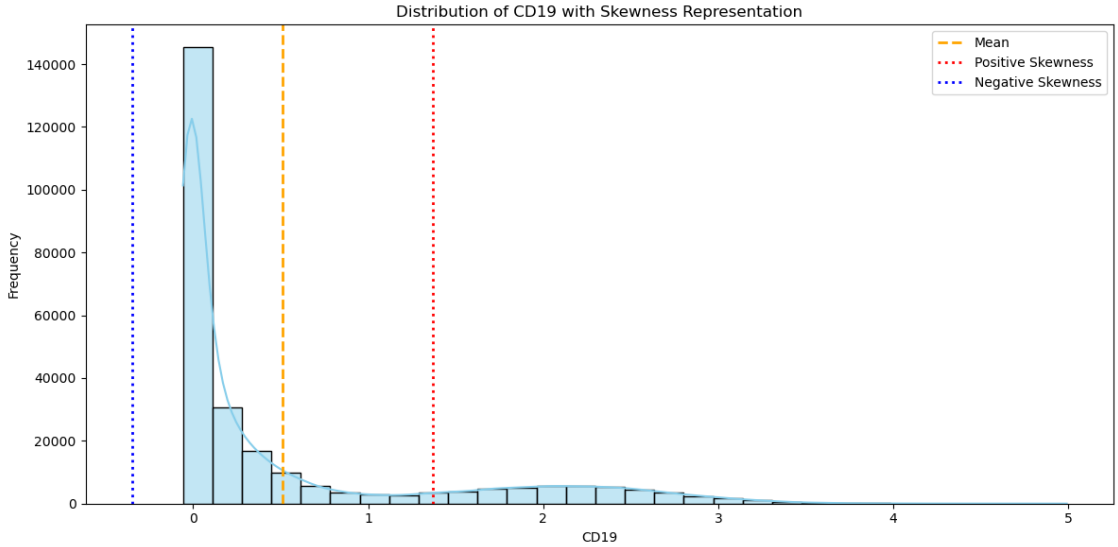
Skewness of DNA2: 0.7792



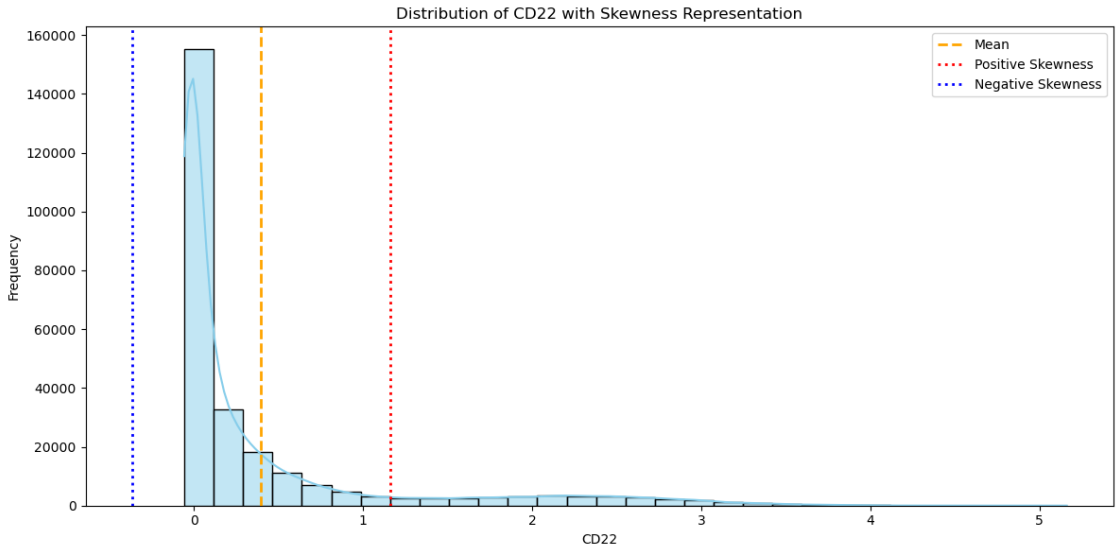
Skewness of CD45RA: 1.1916



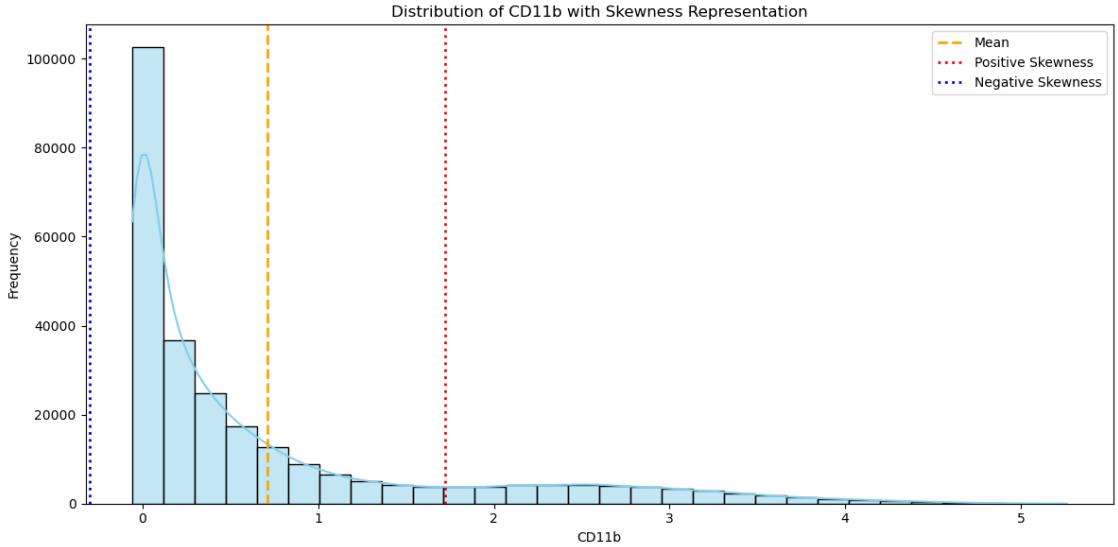
Skewness of CD133: 2.1420



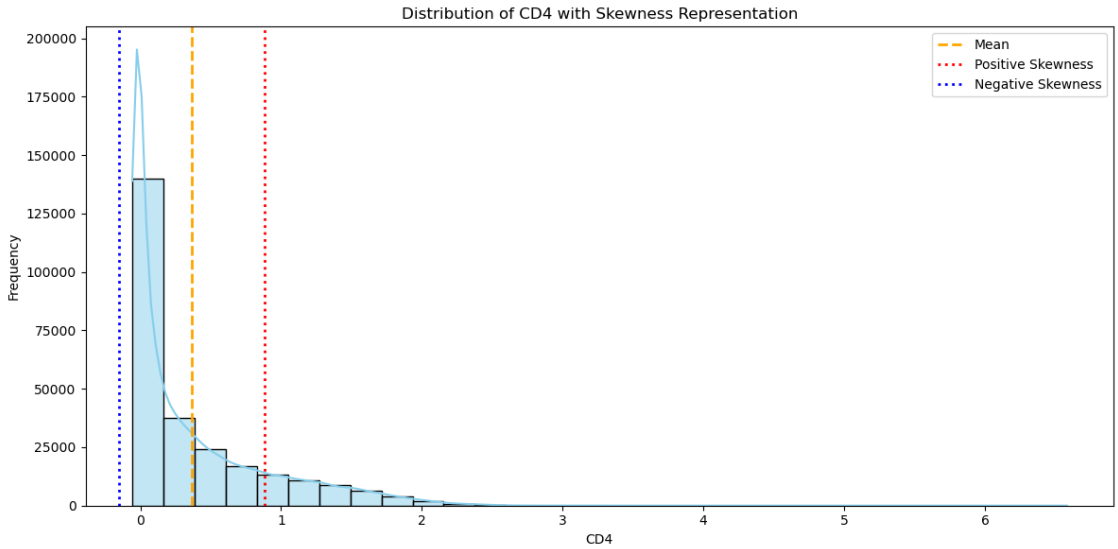
Skewness of CD19: 1.6826



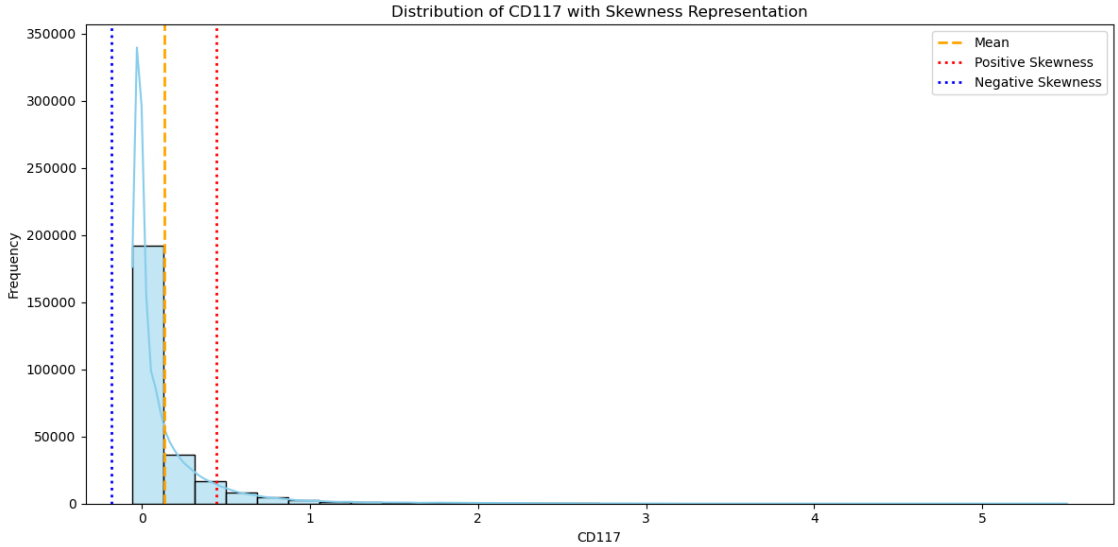
Skewness of CD22: 2.2832



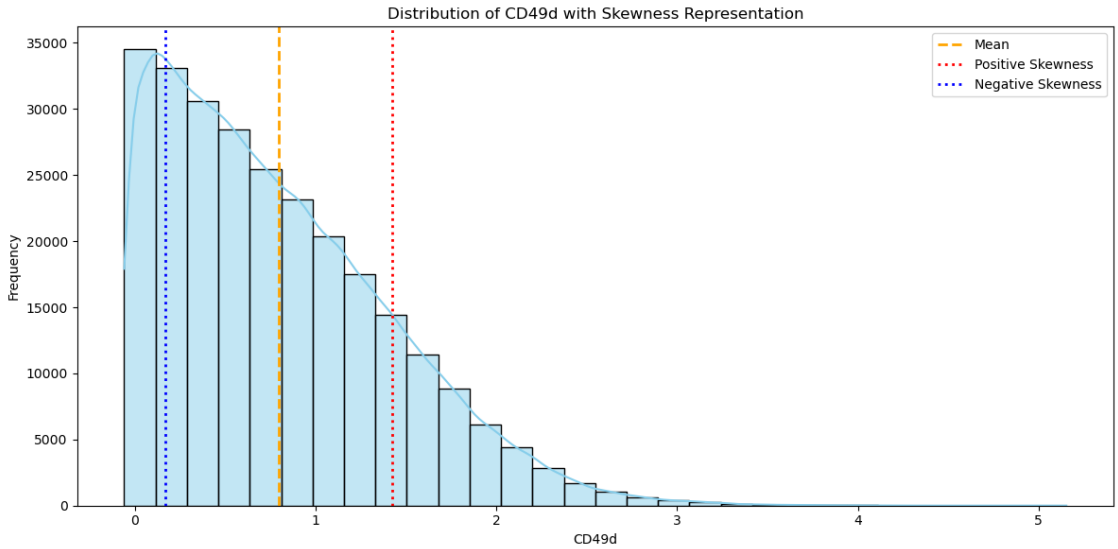
Skewness of CD11b: 1.6791



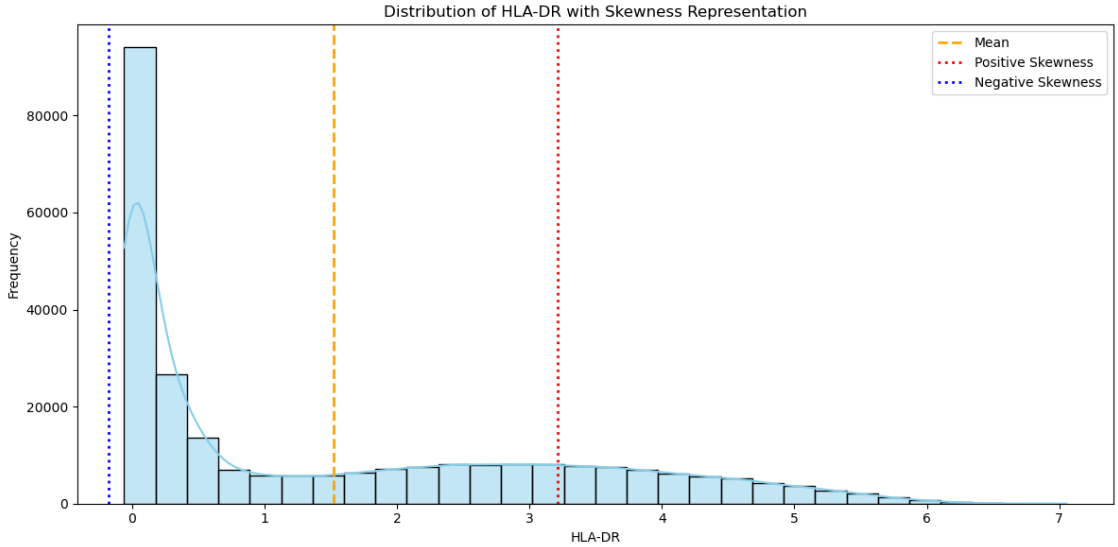
Skewness of CD4: 1.6221



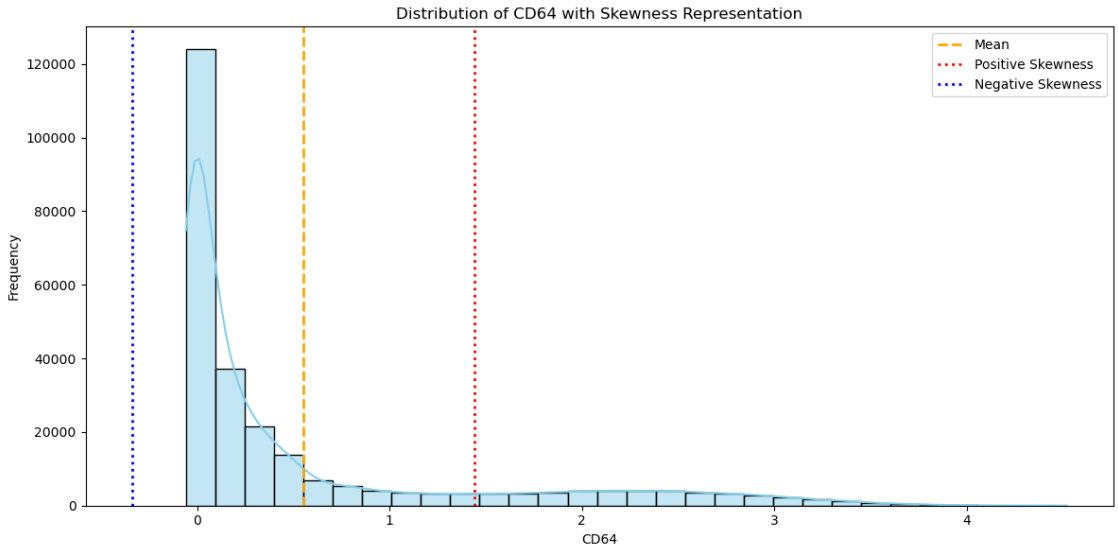
Skewness of CD117: 4.0975



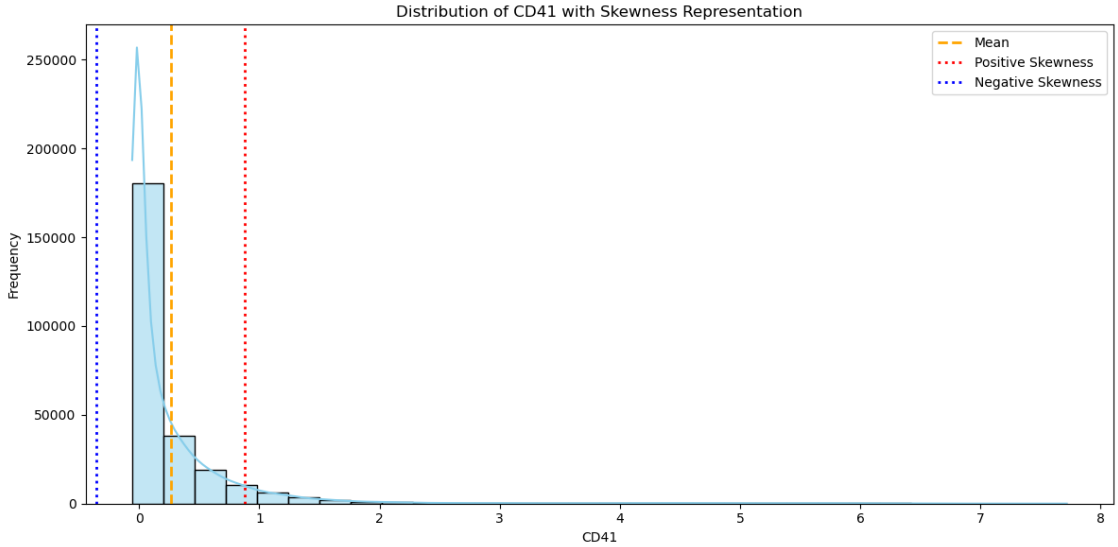
Skewness of CD49d: 0.8568



Skewness of HLA-DR: 0.7954



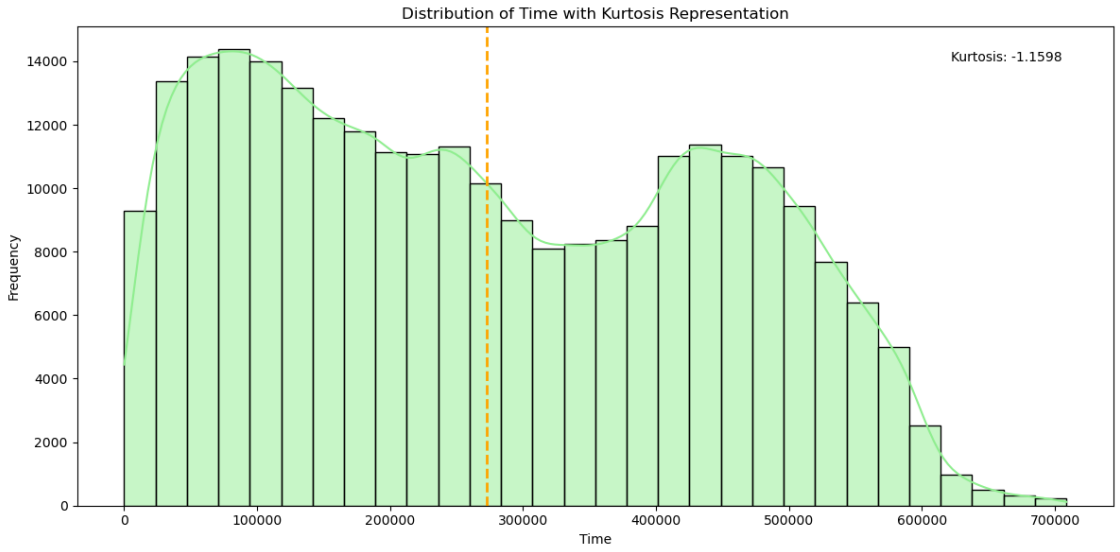
Skewness of CD64: 1.7437



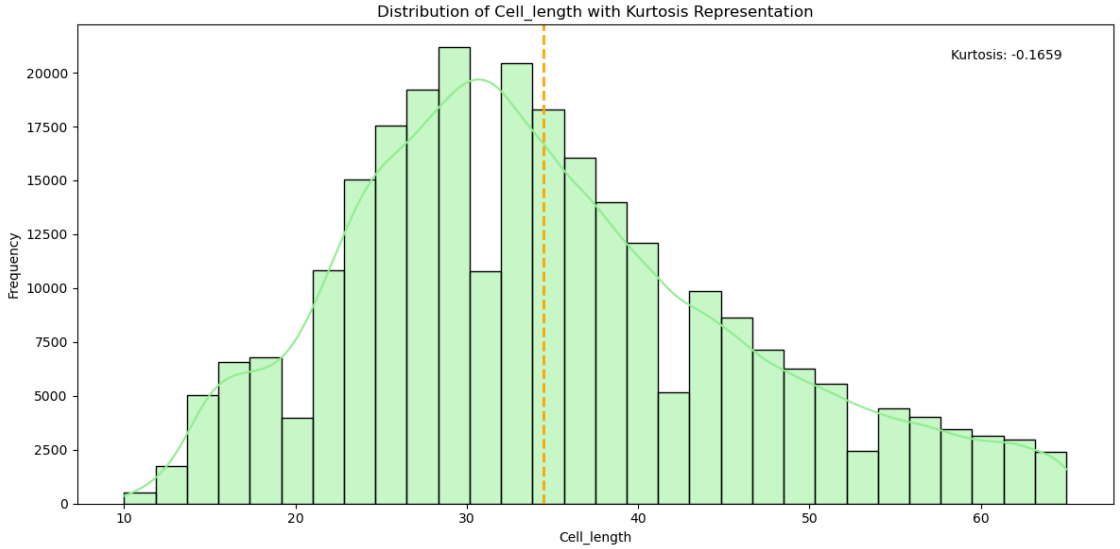
Skewness of CD41: 5.3663

[40]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# List of columns to analyze
columns_to_analyze = [
```

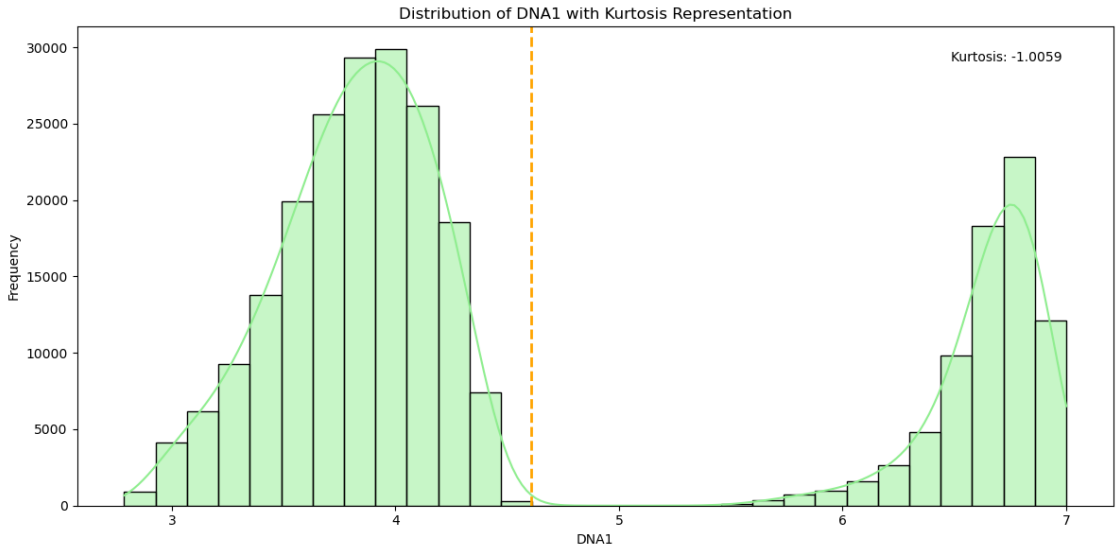
```
    'Time', 'Cell_length', 'DNA1', 'DNA2', 'CD45RA', 'CD133',
    'CD19', 'CD22', 'CD11b', 'CD4', 'CD117', 'CD49d',
    'HLA-DR', 'CD64', 'CD41'
]

# Create a function to plot kurtosis
def plot_kurtosis(column):
    # Calculate kurtosis for the column
    kurtosis = df[column].kurtosis()

    # Create a histogram for the column
    plt.figure(figsize=(12, 6))
    sns.histplot(df[column], bins=30, kde=True, color='lightgreen')

    # Add vertical line for mean
    plt.axvline(df[column].mean(), color='orange', linestyle='dashed',␣
 ↪linewidth=2, label='Mean')

    # Set title and labels
    plt.title(f'Distribution of {column} with Kurtosis Representation')
    plt.xlabel(column)
    plt.ylabel('Frequency')

    # Show kurtosis value on the plot
    plt.text(0.95, 0.95, f'Kurtosis: {kurtosis:.4f}',␣
 ↪horizontalalignment='right',
             verticalalignment='top', transform=plt.gca().transAxes)

    # Show plot
    plt.tight_layout()
    plt.show()

    # Print kurtosis value
    print(f"Kurtosis of {column}: {kurtosis:.4f}")

# Loop through the columns and plot kurtosis for each
for col in columns_to_analyze:
    plot_kurtosis(col)
```



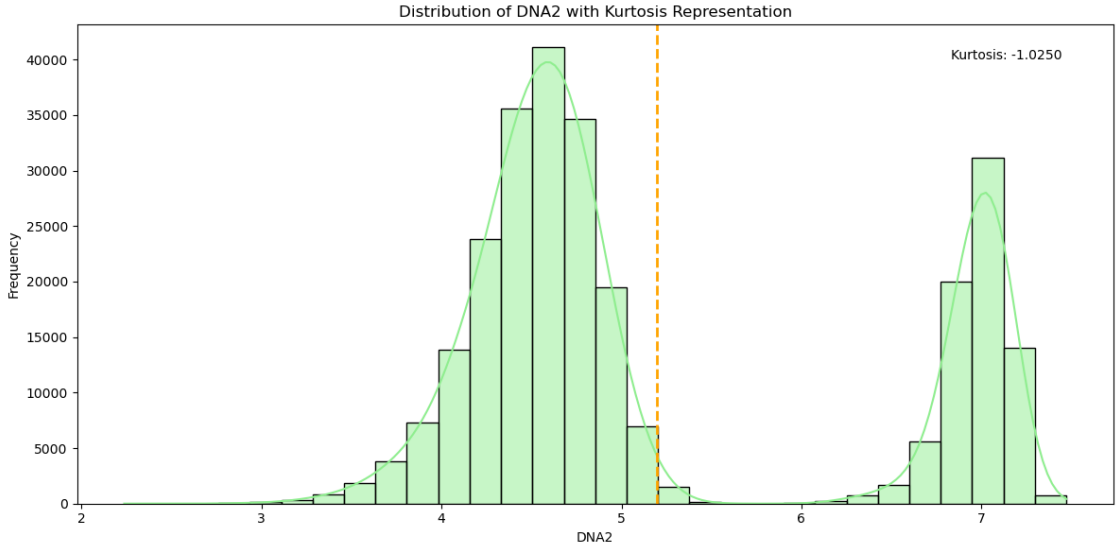
Kurtosis of Time: -1.1598



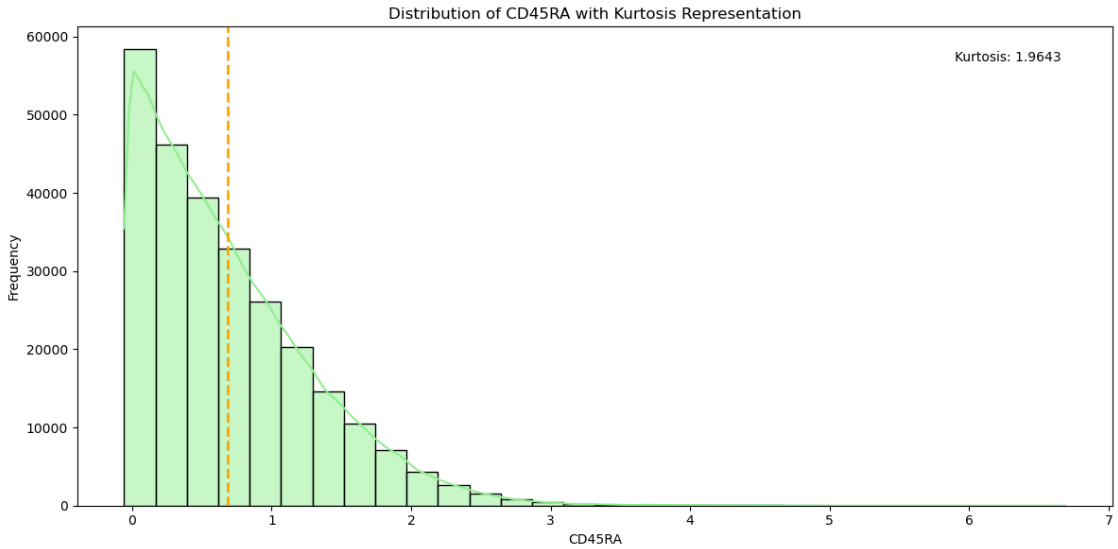
Kurtosis of Cell_length: -0.1659



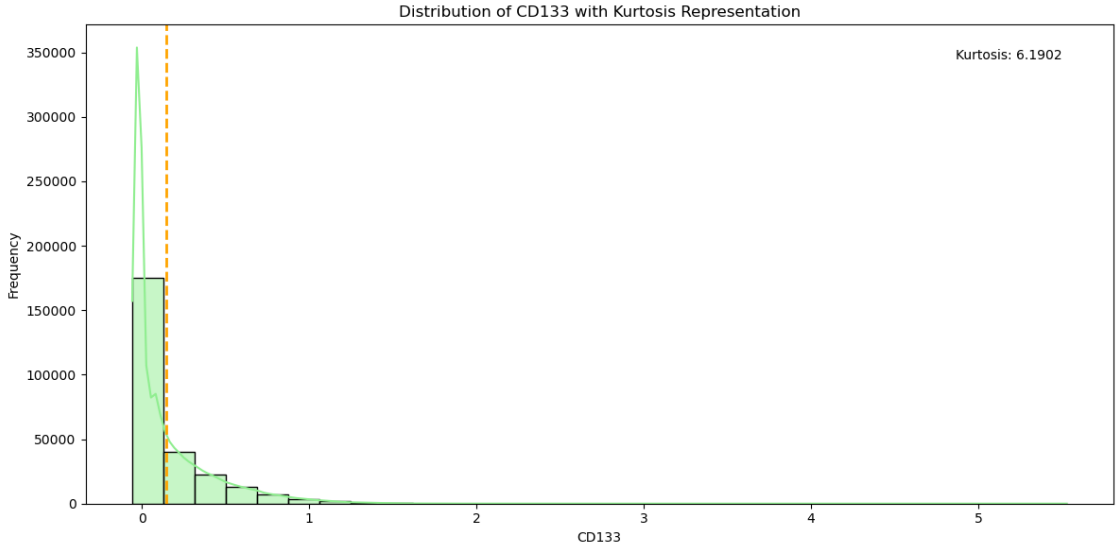
Kurtosis of DNA1: -1.0059



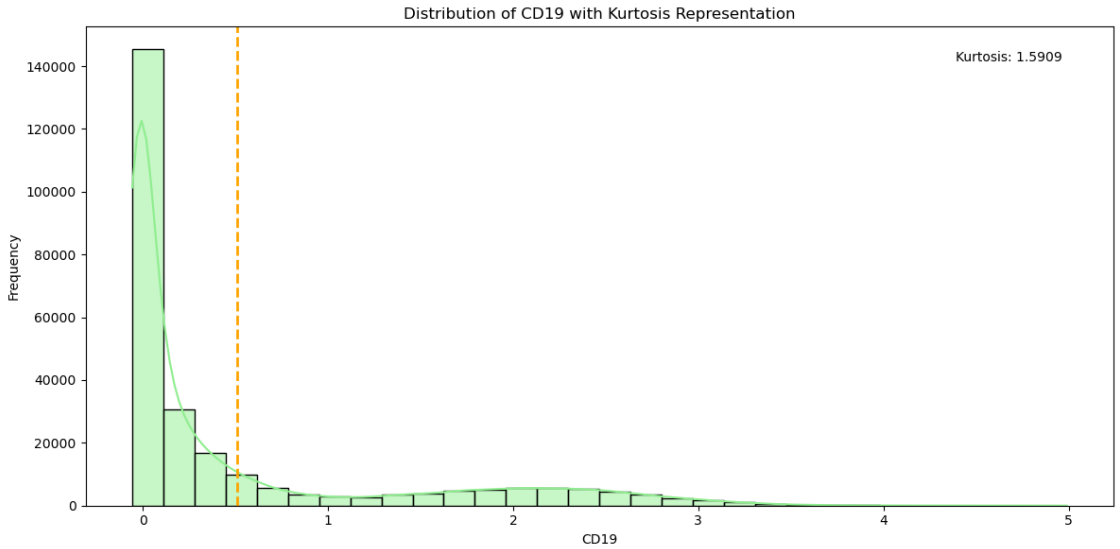
Kurtosis of DNA2: -1.0250



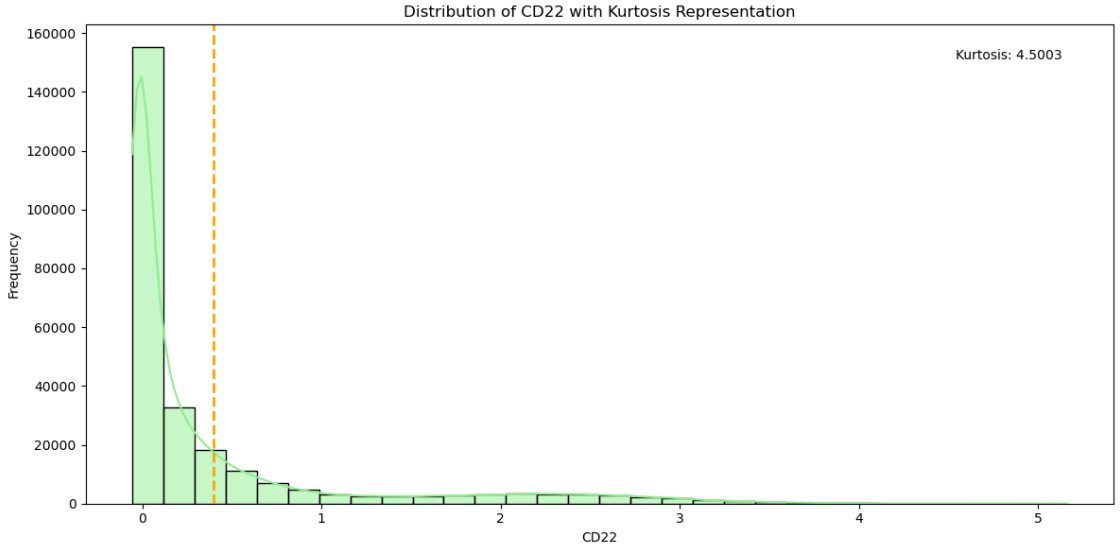
Kurtosis of CD45RA: 1.9643



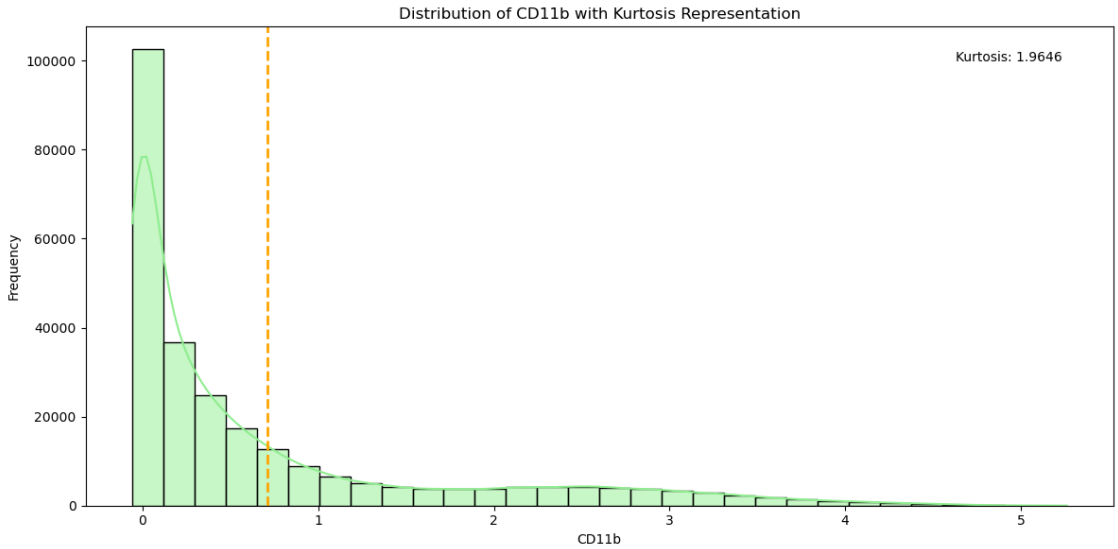
Kurtosis of CD133: 6.1902



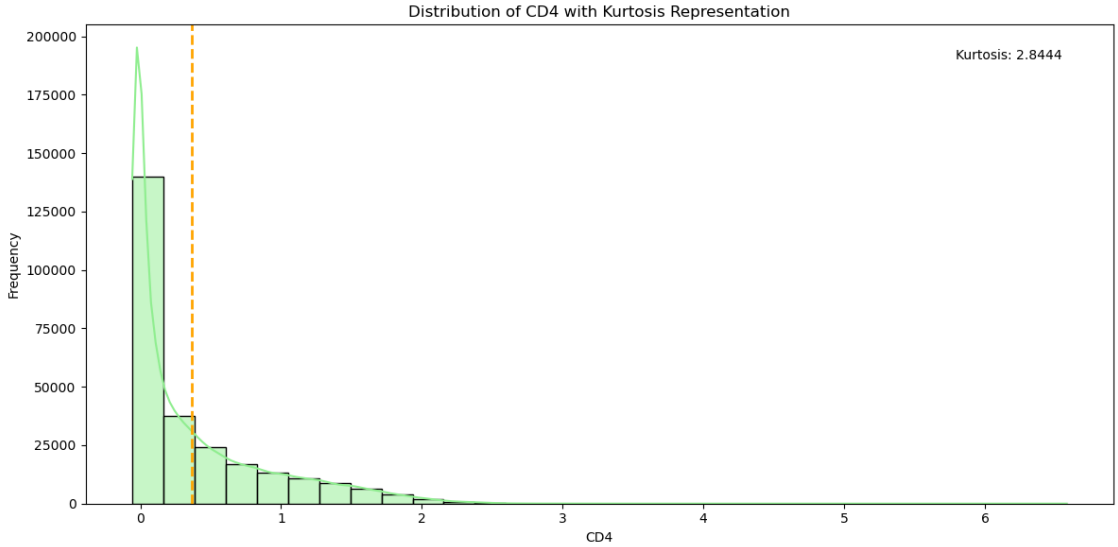
Kurtosis of CD19: 1.5909



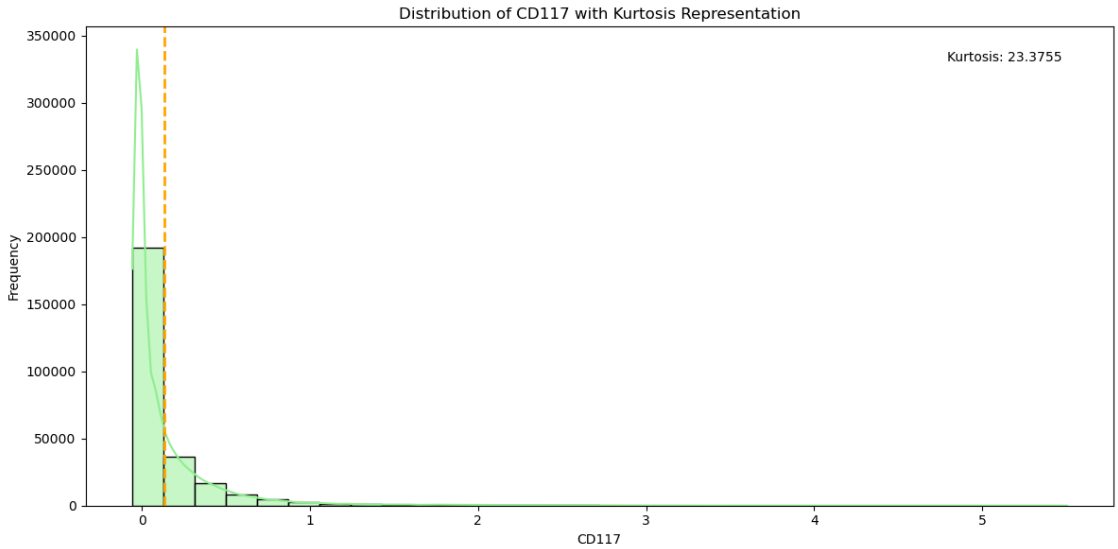
Kurtosis of CD22: 4.5003



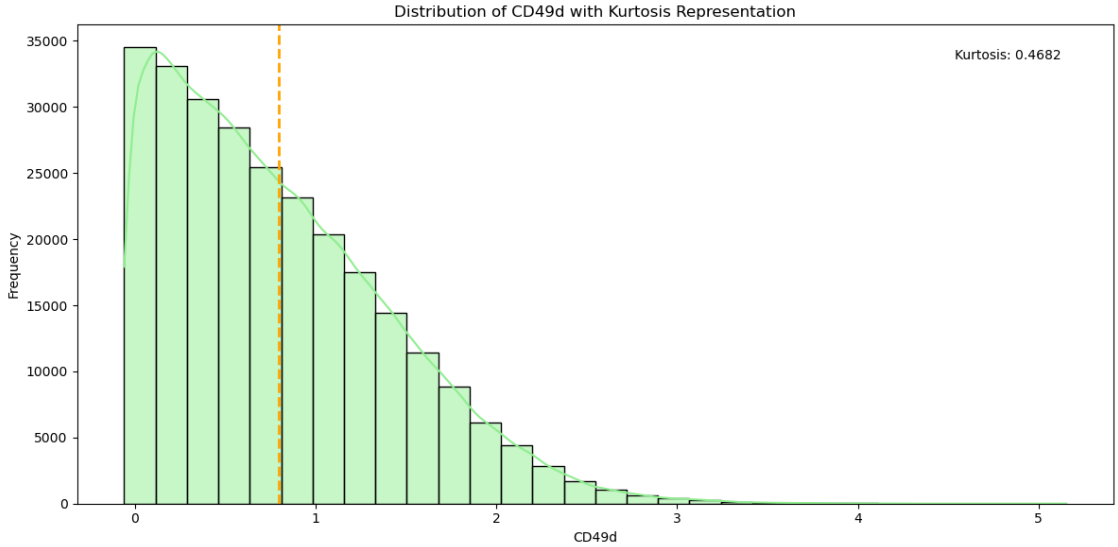
Kurtosis of CD11b: 1.9646



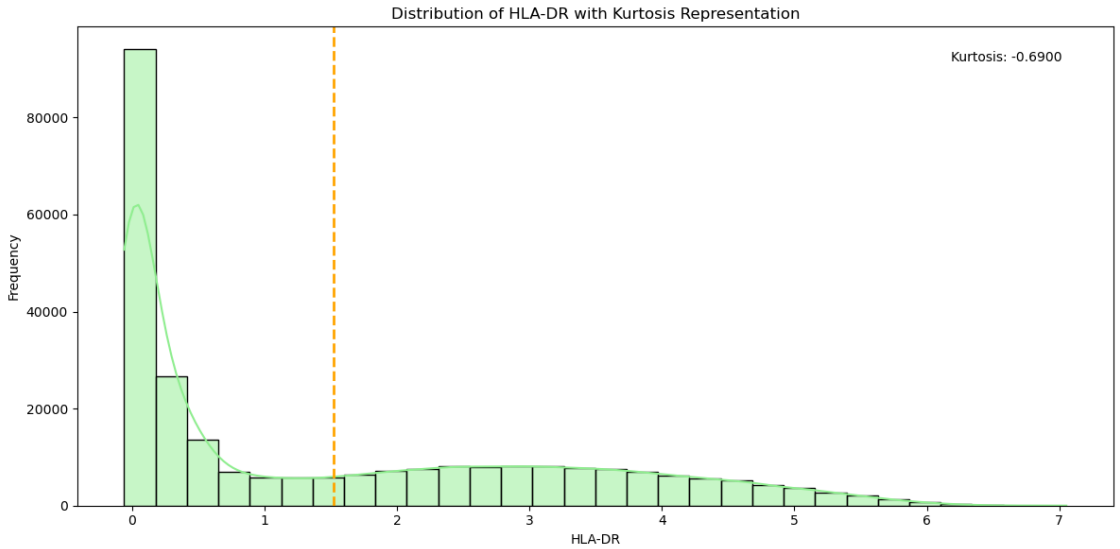
Kurtosis of CD4: 2.8444



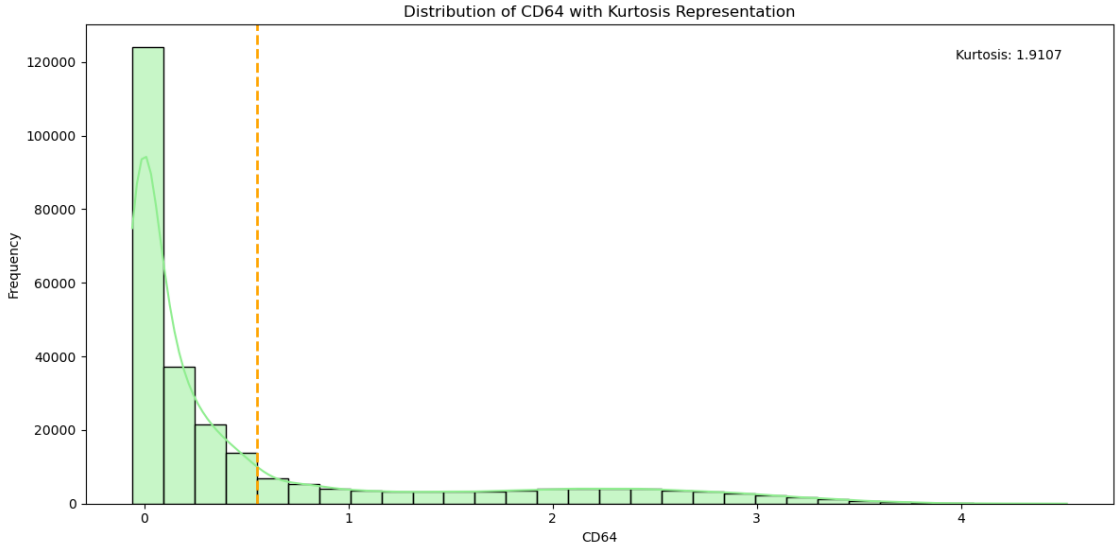
Kurtosis of CD117: 23.3755



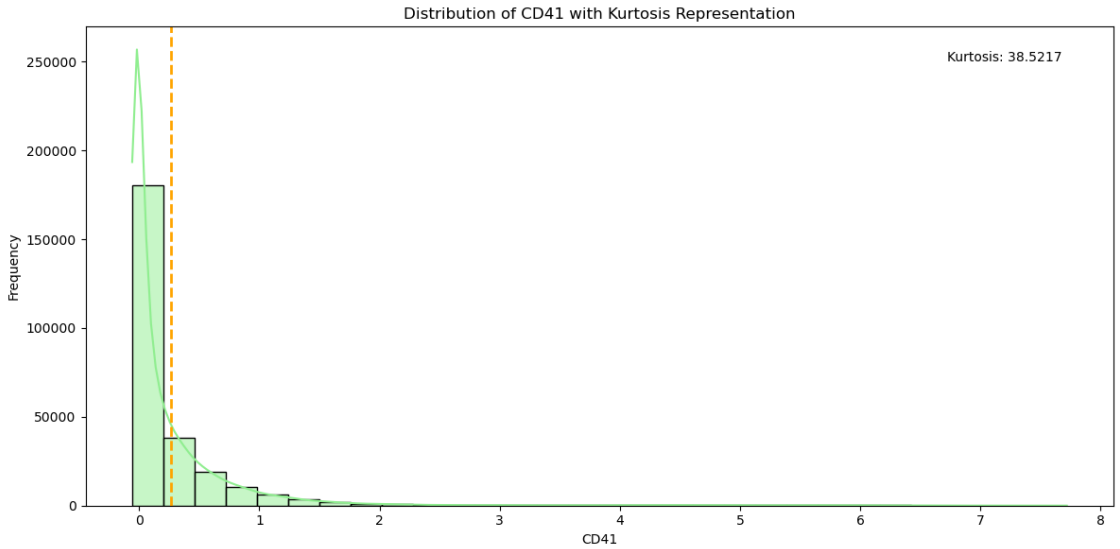
Kurtosis of CD49d: 0.4682



Kurtosis of HLA-DR: -0.6900



Kurtosis of CD64: 1.9107




```
Kurtosis of CD41: 38.5217
```

[38]:
```
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
```

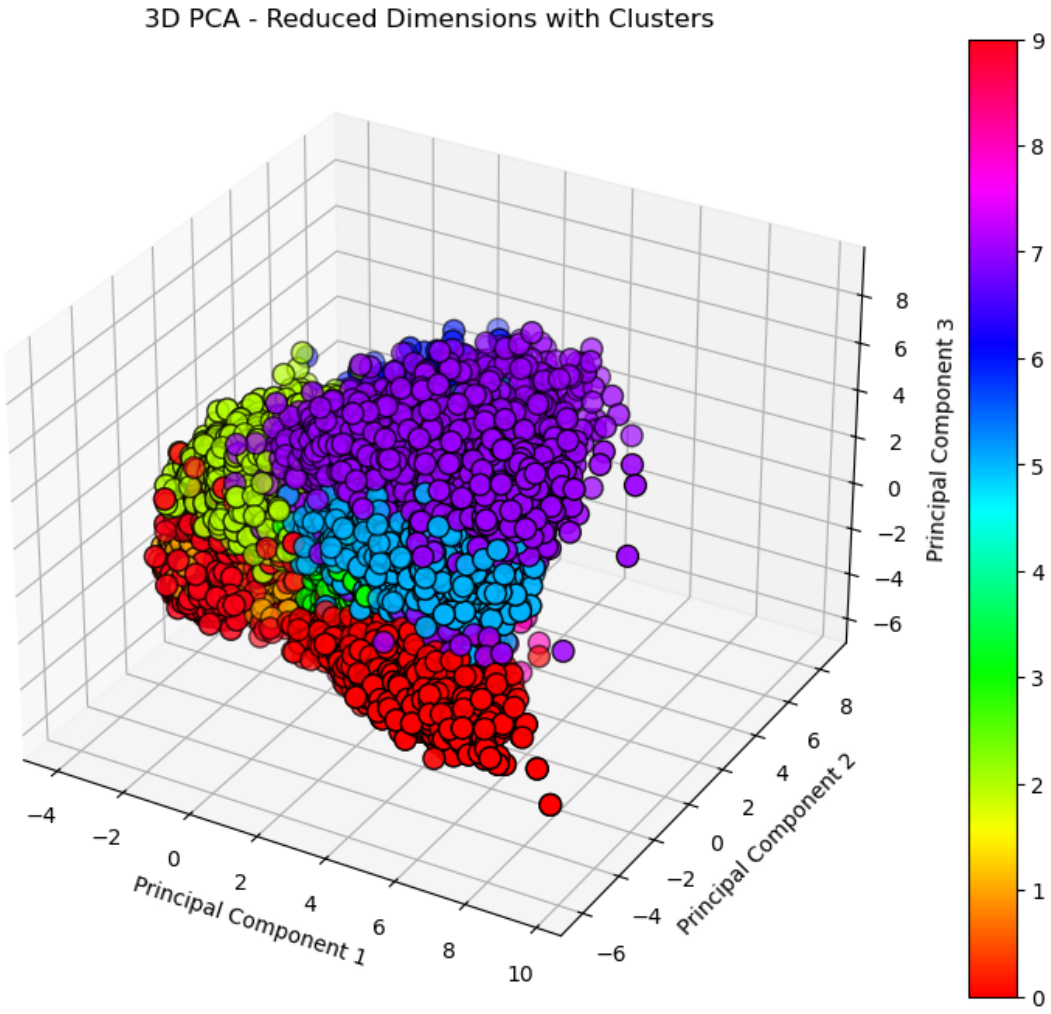
[39]:
```
# Step 1: Clean the column names by stripping any extra spaces
data = pd.read_csv(r"C:\Users\Jai dabas\Downloads\Infosys␣
 ↪internship\Levine_32dim.csv")
data.columns = data.columns.str.strip()

columns_to_remove = ['Event', 'Time', 'Cell_length', 'file_number',␣
 ↪'event_number', 'label', 'individual']
columns_to_remove = [col for col in columns_to_remove if col in data.columns]
data_cleaned = data.drop(columns=columns_to_remove)


scaler = StandardScaler()
X_scaled = scaler.fit_transform(data_cleaned)
print("Standardized Data Shape:", X_scaled.shape)
```

```
C:\Users\Jai dabas\AppData\Local\Temp\ipykernel_25196\1280669671.py:2:
DtypeWarning: Columns (39) have mixed types. Specify dtype option on import or
set low_memory=False.
  data = pd.read_csv(r"C:\Users\Jai dabas\Downloads\Infosys
internship\Levine_32dim.csv")

Standardized Data Shape: (265626, 35)
```

[ ]:
```
from sklearn.cluster import KMeans
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
imputer = SimpleImputer(strategy='mean')
data_cleaned_imputed = imputer.fit_transform(data_cleaned)

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

# Apply PCA to reduce dimensions to 2
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
n_clusters = 6  # You can change this based on your needs
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
clusters = kmeans.fit_predict(X_scaled)
custom_palette = sns.color_palette("hsv", n_colors=n_clusters)

# Visualize the PCA result with cluster coloring
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_pca[:, 1], y=X_pca[:, 0], hue=clusters,␣
 ↪palette=custom_palette, legend='full')
plt.title('PCA - Reduced Dimensions with Clusters')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Cluster')
plt.show()

print('Explained variance ratio:', pca.explained_variance_ratio_)
```

[42]:
```
from sklearn.cluster import KMeans
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
from sklearn.decomposition import PCA

# Impute missing data
imputer = SimpleImputer(strategy='mean')
data_cleaned_imputed = imputer.fit_transform(data_cleaned)

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

```
X_scaled = scaler.fit_transform(data_cleaned_imputed)

# Apply PCA to reduce dimensions to 3 for 3D plotting
pca = PCA(n_components=3)
X_pca = pca.fit_transform(X_scaled)

# Perform KMeans clustering
n_clusters = 10  # Define the number of clusters
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
clusters = kmeans.fit_predict(X_scaled)

# Generate a color palette
palette = sns.color_palette("hsv", n_clusters)

# Create 3D plot
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(111, projection='3d')

# Plot each point with the corresponding cluster color
scatter = ax.scatter(X_pca[:, 0], X_pca[:, 1], X_pca[:, 2], c=clusters,␣
 ↪cmap='hsv', s=100, edgecolor="k")

# Labeling the axes
ax.set_xlabel('Principal Component 1')
ax.set_ylabel('Principal Component 2')
ax.set_zlabel('Principal Component 3')

# Add a color bar
plt.colorbar(scatter)
plt.title('3D PCA - Reduced Dimensions with Clusters')
plt.show()

print('Explained variance ratio:', pca.explained_variance_ratio_)
```




```
Explained variance ratio: [0.15480196 0.10947208 0.10076616]
```

[ ]:
```python
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Assuming data_cleaned is your preprocessed data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(data_cleaned)

# Apply PCA
pca = PCA(n_components=4)
pca.fit(X_scaled)

# Calculate Standard deviation (square root of eigenvalues)
```

```
std_dev = np.sqrt(pca.explained_variance_)

# Proportion of variance
prop_var = pca.explained_variance_ratio_

# Cumulative proportion
cum_var = np.cumsum(prop_var)

# Create a DataFrame for better visualization
pca_table = pd.DataFrame({
    'Standard Deviation': std_dev,
    'Proportion of Variance': prop_var,
    'Cumulative Proportion': cum_var
}, index=[f'PC{i+1}' for i in range(4)])

# Display the table
print(pca_table)
```

pip install –upgrade matplotlib seaborn

[ ]:
```
plt.savefig('tsne_clusters_plot.png')
```

[ ]:
```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.manifold import TSNE
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import seaborn as sns

# Load the data
data = pd.read_csv(r"C:\Users\Jai dabas\Downloads\Infosys␣
 ↪internship\Levine_32dim.csv")
data.columns = data.columns.str.strip()
columns_to_remove = ['Event', 'Time', 'Cell_length', 'file_number',␣
 ↪'event_number', 'label', 'individual']
columns_to_remove = [col for col in columns_to_remove if col in data.columns]
data_cleaned = data.drop(columns=columns_to_remove)

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

# Apply KMeans to find clusters
n_clusters = 10  # Adjust as needed
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
clusters = kmeans.fit_predict(X_scaled)
```

```
# Run t-SNE
tsne = TSNE(n_components=2, perplexity=30, n_iter=1000, random_state=42,␣
 ↪verbose=1)
X_tsne = tsne.fit_transform(X_scaled)

# Create a color palette
palette = sns.color_palette("hsv", n_clusters)

# Plot the t-SNE result with colors
plt.figure(figsize=(10, 8))
sns.scatterplot(x=X_tsne[:, 0], y=X_tsne[:, 1], hue=clusters, palette=palette,␣
 ↪s=100, edgecolor='k', alpha=0.7)

# Customize the plot
plt.title('t-SNE Visualization with Cluster Colors')
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.legend(title='Clusters', loc='best')
plt.grid(True)
plt.show()
```

[ ]:
```
import pandas as pd
import numpy as np

np.random.seed(42)
demo_data = pd.DataFrame({
    'A': [5, 11, 18, 8],
    'B': [10, 40, 15, 30],
    'C': [9, 25, 35, 20]
})

p_m = 0.5

data_array = demo_data.values
mask = np.random.binomial(1, p_m, data_array.shape)
print("Generated Mask (1 represents masked values):\n", mask)

masked_data = np.where(mask == 1, np.nan, data_array)
masked_demo_data = pd.DataFrame(masked_data, columns=demo_data.columns)

print("\nOriginal DataFrame:\n", demo_data)
print("\nMasked DataFrame:\n", masked_demo_data)
```

[ ]:
```
import numpy as np

# Example dataset (e.g., 5 rows, 10 columns)
data = np.random.rand(5, 10)
```

```
# Define the probability of keeping a value (e.g., 70% chance to retain data)
retain_prob = 0.7

# Generate a binary mask for each row independently
mask = np.random.binomial(1, retain_prob, data.shape)

# Apply the binary mask to the dataset
corrupted_data = data * mask

print("Original Data:\n", data)
print("Binary Mask:\n", mask)
print("Corrupted Data:\n", corrupted_data)
```

[ ]:
```
import pandas as pd
import numpy as np

data = {
    'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 40, 50],
    'C': [10.5, 20.5, 30.5, 40.5, 50.5]
}

df = pd.DataFrame(data)

# Shuffle each column independently
shuffled_df = df.apply(lambda x: np.random.permutation(x))

# Generate a random mask DataFrame with values between 0 and 1
mask = pd.DataFrame(np.random.rand(*df.shape), columns=df.columns)

# Calculate the corrupted DataFrame using the formula
corrupted_df = df * (1 - mask) + shuffled_df * mask

print("Original DataFrame:")
print(df)
print("\nShuffled DataFrame:")
print(shuffled_df)
print("\nMask DataFrame:")
print(mask)
print("\nCorrupted DataFrame:")
print(corrupted_df)
```

[40]:
```
import pandas as pd
import numpy as np
df = pd.read_csv(r"C:\Users\Jai dabas\Downloads\Infosys internship\Levine_32dim.
 ↪csv")  # Adjust the file path and file type as needed
```

```python
# Select only numeric columns
numeric_df = df.select_dtypes(include=[np.number])

# Step 1: Shuffle each column independently
shuffled_df = numeric_df.apply(lambda x: np.random.permutation(x))

# Step 2: Generate a random mask with values between 0 and 1
mask = pd.DataFrame(np.random.rand(*numeric_df.shape), columns=numeric_df.
 ↪columns)

# Step 3: Apply the corruption formula
corrupted_df = numeric_df * (1 - mask) + shuffled_df * mask

# Display the results
print("Original Numeric DataFrame:")
print(numeric_df.head())  # Show only the first few rows for readability
print("\nShuffled Numeric DataFrame:")
print(shuffled_df.head())
print("\nMask DataFrame:")
print(mask.head())
print("\nCorrupted Numeric DataFrame:")
print(corrupted_df.head())
corrupted_df.to_csv('corrupted_numeric_dataset.csv', index=False)
```

```
C:\Users\Jai dabas\AppData\Local\Temp\ipykernel_25196\1286124188.py:3:
DtypeWarning: Columns (39) have mixed types. Specify dtype option on import or
set low_memory=False.
  df = pd.read_csv(r"C:\Users\Jai dabas\Downloads\Infosys
internship\Levine_32dim.csv")  # Adjust the file path and file type as needed
```

```
Original Numeric DataFrame:
     Time   Cell_length      DNA1      DNA2    CD45RA     CD133      CD19  \
0  2693.0            22  4.391057  4.617262  0.162691 -0.029585 -0.006696
1  3736.0            35  4.340481  4.816692  0.701348 -0.038280 -0.016654
2  7015.0            32  3.838727  4.386369  0.603568 -0.032216  0.073855
3  7099.0            29  4.255805  4.830048  0.433747 -0.027611 -0.017661
4  7700.0            25  3.976909  4.506433 -0.008809 -0.030297  0.080423

       CD22     CD11b       CD4  …      CD61     CD117     CD49d    HLA-DR  \
0  0.066388 -0.009184  0.363602  … -0.002936  0.053050  0.853505  1.664480
1  0.074409  0.808031 -0.035424  …  1.258437  0.089660  0.197818  0.491592
2 -0.042977 -0.001881 -0.008781  …  0.257137  0.046222  2.586670  1.308337
3 -0.044072  0.733698 -0.019066  … -0.041140  0.066470  1.338669  0.140523
4  0.495791  1.107627  0.552746  …  0.168609 -0.006223  0.180924  0.197332

       CD64      CD41   Viability   file_number   event_number   individual
0 -0.005376 -0.001961    0.648429      3.627711            307            1
```

```
1  0.144814  0.868014    0.561384      3.627711            545            1
2 -0.010961 -0.010413    0.643337      3.627711           1726            1
3 -0.013449 -0.026039   -0.026523      3.627711           1766            1
4  0.076167 -0.040488    0.283287      3.627711           2031            1

[5 rows x 40 columns]

Shuffled Numeric DataFrame:
          Time   Cell_length      DNA1      DNA2    CD45RA     CD133  \
0  150065.0000            27  3.872726  3.278782  1.153302  0.413941
1   57068.0000            32  3.329390  4.955018  0.669912  0.081947
2  287278.0000            22  6.936321  3.818933  1.169744  0.117483
3  408242.0000            22  6.911945  4.650151  1.177265 -0.011076
4  499726.4375            23  6.901727  4.676143  0.055722  0.838414

       CD19      CD22     CD11b       CD4  …      CD61     CD117     CD49d  \
0 -0.008341 -0.022674  1.511375  0.219111  … -0.004308 -0.001466 -0.043889
1  0.057064 -0.002600  1.233898  0.372674  … -0.045614  0.077214  1.993805
2 -0.021608  2.872243  0.251158 -0.016674  … -0.020379 -0.021998  0.086764
3 -0.027496 -0.008136 -0.002881  0.297840  … -0.045165 -0.035611  1.219935
4  0.080570 -0.034493  0.673453 -0.025444  …  0.702945 -0.034191  0.837690

     HLA-DR      CD64      CD41   Viability   file_number   event_number  \
0  3.731541 -0.034454  0.132572   -0.019558      3.669327          90103
1 -0.041743 -0.050831 -0.029592   -0.048923      3.627711          77215
2  0.433949  1.274004  0.412577    1.634716      3.669327         342865
3  1.181792 -0.032243  0.357148    1.451420      3.627711         240561
4  3.009833 -0.008601 -0.026546    1.439570      3.627711         397168

    individual
0            1
1            1
2            1
3            1
4            2

[5 rows x 40 columns]

Mask DataFrame:
       Time   Cell_length      DNA1      DNA2    CD45RA     CD133      CD19  \
0  0.250639      0.717870  0.555088  0.411862  0.497742  0.148547  0.338929
1  0.220524      0.448071  0.045725  0.473360  0.749536  0.460303  0.386199
2  0.049648      0.933175  0.172848  0.564109  0.131319  0.788018  0.382131
3  0.565956      0.536815  0.122272  0.420779  0.314761  0.441657  0.712650
4  0.965625      0.593721  0.767042  0.157009  0.180260  0.359619  0.631838

       CD22     CD11b       CD4  …      CD61     CD117     CD49d    HLA-DR  \
0  0.711690  0.760815  0.973849  …  0.830683  0.805864  0.159053  0.586723
```

```
1  0.411152  0.275604  0.058986  …  0.803614  0.988365  0.524996  0.096883
2  0.818270  0.752344  0.148562  …  0.058239  0.746764  0.191950  0.437251
3  0.919429  0.200641  0.430224  …  0.737709  0.058119  0.655392  0.475511
4  0.038624  0.039200  0.107538  …  0.785861  0.697905  0.034928  0.173792

       CD64      CD41   Viability   file_number   event_number   individual
0  0.554882  0.056935    0.571307      0.665265       0.587498     0.312692
1  0.788269  0.130728    0.014944      0.699356       0.500931     0.838540
2  0.628569  0.784879    0.897966      0.402913       0.711522     0.473156
3  0.283197  0.517404    0.315310      0.803888       0.735358     0.350874
4  0.340547  0.451477    0.139617      0.493333       0.016806     0.463909

[5 rows x 40 columns]

Corrupted Numeric DataFrame:
            Time   Cell_length      DNA1      DNA2    CD45RA     CD133  \
0   39630.112556     25.589349  4.103338  4.065993  0.655760  0.036300
1   15496.978500     33.655787  4.294249  4.882170  0.677786  0.017061
2   20929.474630     22.668249  4.374139  4.066273  0.677918  0.085749
3  234128.238737     25.242297  4.580577  4.754351  0.667778 -0.020308
4  482812.892851     23.812558  6.220368  4.533079  0.002823  0.282108

       CD19      CD22     CD11b       CD4  …      CD61     CD117     CD49d  \
0 -0.007254  0.003004  1.147680  0.222889  … -0.004076  0.009118  0.710772
1  0.011816  0.042747  0.925402 -0.011352  …  0.210484  0.077359  1.140705
2  0.037376  2.342460  0.188491 -0.009953  …  0.240975 -0.004722  2.106812
3 -0.024670 -0.011032  0.585910  0.117274  … -0.044110  0.060537  1.260852
4  0.080516  0.475310  1.090607  0.490568  …  0.588523 -0.025742  0.203864

     HLA-DR      CD64      CD41   Viability   file_number   event_number  \
0  2.877272 -0.021511  0.005699    0.266803      3.655396   53061.926446
1  0.439921 -0.009407  0.750672    0.552263      3.627711   38951.366932
2  0.926010  0.796729  0.321583    1.533562      3.644478  244454.070917
3  0.635658 -0.018771  0.172223    0.439487      3.627711  177365.739030
4  0.686121  0.047300 -0.034193    0.444724      3.627711    8671.730602

    individual
0     1.000000
1     1.000000
2     1.000000
3     1.000000
4     1.463909

[5 rows x 40 columns]
```

[53]:
```
print(df.columns)
```

```
Index(['Time', 'Cell_length', 'DNA1', 'DNA2', 'CD45RA', 'CD133', 'CD19',
```

```
       'CD22', 'CD11b', 'CD4', 'CD8', 'CD34', 'Flt3', 'CD20', 'CXCR4',
       'CD235ab', 'CD45', 'CD123', 'CD321', 'CD14', 'CD33', 'CD47', 'CD11c',
       'CD7', 'CD15', 'CD16', 'CD44', 'CD38', 'CD13', 'CD3', 'CD61', 'CD117',
       'CD49d', 'HLA-DR', 'CD64', 'CD41', 'Viability', 'file_number',
       'event_number', 'label', 'individual'],
      dtype='object')
```

[70]:
```
import pandas as pd

# Assuming 'df' is your original DataFrame with a 'label' column

# Separate rows with and without labels
labeled_data = df[df['label'].notna()]
unlabeled_data = df[df['label'].isna()]

# Split labeled_data into x_label and y_label
x_labeled = labeled_data.drop(columns='label')  # Features without the 'label' column
y_labeled = labeled_data['label']               # The 'label' column values

# Split unlabeled_data into x_unlabeled and y_unlabeled
# For unlabeled data, y_unlabeled can simply be set to None or filled with NaN
x_unlabeled_scaled = unlabeled_data.drop(columns='label')
#y_unlabeled = unlabeled_data['label']  # This will contain NaN or None for all rows

# Display results
print("Labeled Data (x_label):\n", x_labeled.head())
print("\nLabels (y_label):\n", y_labeled.head())
print("\nUnlabeled Data (x_unlabeled):\n", x_unlabeled_scaled.head())
#print("\nUnlabeled Labels (y_unlabeled - should be NaN or None):\n", y_unlabeled.head())
```

```
Labeled Data (x_label):
            Time  Cell_length      DNA1      DNA2    CD45RA     CD133  \
98304  258774.0           31  6.521691  6.683959 -0.043242  0.053053
98305  259076.0           46  6.910923  7.214163  0.938538 -0.015475
98306  259120.0           47  6.427468  6.925929  0.384571 -0.004175
98307  259187.0           52  6.634492  7.006770 -0.041910 -0.045223
98308  259346.0           55  6.734909  7.092107  0.365834 -0.019823

           CD19      CD22     CD11b       CD4  …       CD61     CD117  \
98304 -0.011619  0.128668  3.456165  0.123497  …  0.226301 -0.012806
98305 -0.026055 -0.028350  3.360608  0.779725  …  0.458380 -0.012684
98306  0.030729 -0.018638  4.246710  0.287215  … -0.020765  0.284424
98307  0.295282 -0.023605  3.158714  0.008212  …  0.117218 -0.006635
98308  0.219947 -0.034787  2.038478  0.280221  …  0.729796  0.461026
```

```
          CD49d    HLA-DR      CD64      CD41  Viability  file_number  \
98304  0.793253  2.654707  1.558488 -0.024085  -0.011326     3.669327
98305  0.397168  2.088919  1.035073  0.276884  -0.000675     3.669327
98306  1.074889  2.415125  2.213952  0.684035  -0.036371     3.669327
98307  0.609407  2.000651  2.555156  0.109426   0.199961     3.669327
98308  0.362782  1.839421  2.080078  0.360985   0.137301     3.669327

       event_number  individual
98304         52800           2
98305         52856           2
98306         52862           2
98307         52881           2
98308         52917           2

[5 rows x 40 columns]

Labels (y_label):
 98304    10
98305    10
98306    10
98307    10
98308    10
Name: label, dtype: object

Unlabeled Data (x_unlabeled):
      Time  Cell_length      DNA1      DNA2    CD45RA     CD133      CD19  \
0  2693.0           22  4.391057  4.617262  0.162691 -0.029585 -0.006696
1  3736.0           35  4.340481  4.816692  0.701348 -0.038280 -0.016654
2  7015.0           32  3.838727  4.386369  0.603568 -0.032216  0.073855
3  7099.0           29  4.255805  4.830048  0.433747 -0.027611 -0.017661
4  7700.0           25  3.976909  4.506433 -0.008809 -0.030297  0.080423

       CD22     CD11b       CD4  …      CD61     CD117     CD49d    HLA-DR  \
0  0.066388 -0.009184  0.363602  … -0.002936  0.053050  0.853505  1.664480
1  0.074409  0.808031 -0.035424  …  1.258437  0.089660  0.197818  0.491592
2 -0.042977 -0.001881 -0.008781  …  0.257137  0.046222  2.586670  1.308337
3 -0.044072  0.733698 -0.019066  … -0.041140  0.066470  1.338669  0.140523
4  0.495791  1.107627  0.552746  …  0.168609 -0.006223  0.180924  0.197332

       CD64      CD41  Viability  file_number  event_number  individual
0 -0.005376 -0.001961   0.648429     3.627711           307           1
1  0.144814  0.868014   0.561384     3.627711           545           1
2 -0.010961 -0.010413   0.643337     3.627711          1726           1
3 -0.013449 -0.026039  -0.026523     3.627711          1766           1
4  0.076167 -0.040488   0.283287     3.627711          2031           1

[5 rows x 40 columns]
```

[71]:
```
print(x_unlabeled_scaled.shape)
```

```
(98304, 40)
```

[45]:
```
print(x_labeled.shape)
```

```
(265626, 40)
```

[80]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Assuming labeled_data contains your labeled dataset and
# 'label' is the column name for the labels

# Separate features (X) and labels (y)
X = labeled_data.drop(columns='label')
y = labeled_data['label']

# Split the data with 70% training and 30% testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)

# Display the shapes of the resulting splits
print("Training set (X_train):", X_train.shape)
print("Training labels (y_train):", y_train.shape)
print("Test set (X_test):", X_test.shape)
print("Test labels (y_test):", y_test.shape)
```

```
Training set (X_train): (117125, 40)
Training labels (y_train): (117125,)
Test set (X_test): (50197, 40)
Test labels (y_test): (50197,)
```

[81]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
import numpy as np

# Initialize the Logistic Regression model
model = LogisticRegression()

# Train the model with the training data
model.fit(X_train, y_train)

# Predict probabilities on the test data
y_pred_proba = model.predict_proba(X_test)
```

```
# Calculate the log loss (cross-entropy loss) using y_test and predicted␣
 ↪probabilities
loss = log_loss(y_test, y_pred_proba)

# Output the predicted probabilities and the log loss
print("Predicted Probabilities on X_test:\n", y_pred_proba)
print("\nLog Loss on test data:", loss)
```

```
C:\Users\Jai dabas\anaconda3\Lib\site-
packages\sklearn\linear_model\_logistic.py:469: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

```
Predicted Probabilities on X_test:
 [[1.16631347e-35 1.23913197e-42 9.37341035e-63 2.52502013e-25
  6.48779259e-63 1.00000000e+00]
 [6.27009959e-23 2.70780190e-27 6.78963871e-40 2.49428553e-16
  3.43555185e-40 1.00000000e+00]
 [1.81234180e-01 2.36472381e-02 8.01629252e-05 1.17564979e-01
  1.34113300e-02 6.64062110e-01]
 …
 [2.39637900e-15 3.45000062e-18 2.13887496e-26 5.49829215e-11
  1.10191109e-26 1.00000000e+00]
 [8.70121903e-03 1.66965082e-03 1.53645608e-05 2.52327758e-02
  1.17217284e-04 9.64263772e-01]
 [4.99906580e-02 1.90982724e-02 1.24035180e-03 9.53647479e-02
  3.90402222e-03 8.30401948e-01]]

Log Loss on test data: 0.21989920595385007
```

[85]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
import numpy as np

# Define the logistic regression function
def Logistic(X_train, y_train, X_test, y_test):

    # Check and reshape y_train if needed
    if len(y_train.shape) > 1:
        y_train = y_train.ravel()
```

```
    # Define and fit the logistic regression model
    model = LogisticRegression(random_state=42, max_iter=500)
    model.fit(X_train, y_train)

    # Predict probabilities on x_test
    y_test_hat = model.predict_proba(X_test)

    # Calculate log loss using the true labels (y_test) and predicted␣
 ↪probabilities (y_test_hat)
    loss = log_loss(y_test, y_test_hat)

    return y_test_hat, loss


# Assuming x_train, y_train, x_test, and y_test are already defined
# Uncomment and define them as needed
y_test_probabilities, loss_value = Logistic(X_train, y_train, X_test, y_test)

# Display results
print("Predicted probabilities:\n", y_test_probabilities)
print("\nLog Loss:", loss_value)
```

```
C:\Users\Jai dabas\anaconda3\Lib\site-
packages\sklearn\linear_model\_logistic.py:469: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

```
Predicted probabilities:
 [[7.58405763e-21 8.81038122e-28 9.16289659e-64 7.36982718e-09
  5.80826483e-15 9.99999993e-01]
 [2.08707686e-14 1.15467511e-18 1.90711769e-41 1.03102321e-06
  1.94654476e-10 9.99998969e-01]
 [8.96177290e-02 1.85538062e-02 2.28843988e-03 5.28453456e-02
  5.77050458e-03 8.30924175e-01]
 …
 [6.78620705e-10 1.17646663e-12 1.24232752e-27 8.33145843e-05
  3.32002454e-07 9.99916353e-01]
 [1.40318290e-03 4.60922336e-04 3.26648100e-06 5.54339985e-03
  1.42174158e-03 9.91167487e-01]
 [1.10816873e-02 6.08694591e-03 3.31984562e-04 2.55557700e-02
  1.23008895e-02 9.44642723e-01]]
```

```
Log Loss: 0.16439477854983603
```

[88]:
```python
from xgboost import XGBClassifier
from sklearn.metrics import log_loss
import numpy as np

def XGBoostClassifier(x_train, y_train, x_test, y_test):

    # Check and reshape y_train if needed
    if len(y_train.shape) > 1:
        y_train = y_train.ravel()

    # Define and fit the XGBoost model
    model = XGBClassifier(use_label_encoder=False, eval_metric='logloss',␣
 ↪random_state=42)
    model.fit(x_train, y_train)

    # Predict probabilities on x_test
    y_test_hat = model.predict_proba(x_test)

    # Calculate log loss
    loss = log_loss(y_test, y_test_hat)

    return y_test_hat, loss


# Get the predicted probabilities and calculate log loss
#y_test_probabilities, loss_value = XGBoostClassifier(x_train, y_train, x_test,␣
 ↪y_test)

print("Predicted probabilities:\n", y_test_probabilities)
print("\nLog Loss:", loss_value)
```

```
Predicted probabilities:
 [[7.58405763e-21 8.81038122e-28 9.16289659e-64 7.36982718e-09
  5.80826483e-15 9.99999993e-01]
 [2.08707686e-14 1.15467511e-18 1.90711769e-41 1.03102321e-06
  1.94654476e-10 9.99998969e-01]
 [8.96177290e-02 1.85538062e-02 2.28843988e-03 5.28453456e-02
  5.77050458e-03 8.30924175e-01]
 …
 [6.78620705e-10 1.17646663e-12 1.24232752e-27 8.33145843e-05
  3.32002454e-07 9.99916353e-01]
 [1.40318290e-03 4.60922336e-04 3.26648100e-06 5.54339985e-03
  1.42174158e-03 9.91167487e-01]
 [1.10816873e-02 6.08694591e-03 3.31984562e-04 2.55557700e-02
  1.23008895e-02 9.44642723e-01]]
```

```
Log Loss: 0.16439477854983603
```

[101]:
```python
import numpy as np
import pandas as pd
from keras.layers import Input, Dense
from keras.models import Model
import tensorflow as tf
from keras.optimizers import Adam
from keras import models  # Import models from keras

def Binary_mask(p_m, data):
  return  np.random.binomial(1, p_m, data.shape)

def x_corrupted_df(corruption_binary_mask, data):
  # Convert data to DataFrame if it's not already
  if not isinstance(data, pd.DataFrame):
    data = pd.DataFrame(data)
  #Use data to derive the shuffled dataframe
  df_shuffled = data.apply(lambda x: x.sample(frac=1).reset_index(drop=True))
  # Apply the mask to corrupt the data
  x_corrupted = data.values * (1 - corruption_binary_mask) + df_shuffled.values␣
 ↪* corruption_binary_mask

  mask_new = 1 * (data.values != x_corrupted)

# Convert the corrupted data back to a DataFrame
  x_corrupted_df = pd.DataFrame(x_corrupted, columns=data.columns)

  return x_corrupted_df,mask_new

def self_supervised(x_unlabeled, p_m, alpha, parameters):

  epochs = parameters['epochs']
  batch_size = parameters['batch_size']
  _,dimension = x_unlabeled.shape

  # Generate the corrupted data and mask first
  corruption_binary_mask = Binary_mask(p_m, x_unlabeled)

  x_unlabeled_corrupted, mask_new = x_corrupted_df( corruption_binary_mask␣
 ↪,x_unlabeled)

  input_layer = Input(shape=(dimension,))
  #encoder model
  h = Dense(int(dimension), activation='relu')(input_layer)
```

```
 #output1 ---> mask estimationm
 output1 = Dense(int(dimension), activation ='sigmoid', name␣
↪='mask_estimation')(h)
 # iutput2 --->feature estimation
 output2 = Dense(int(dimension), activation='sigmoid', name␣
↪='feature_estimation')(h)

 # input --->output1
 #
 # --------->output2

 model = Model(inputs=input_layer, outputs=[output1, output2])

 # Change the loss_weights to use floats
 model.compile(optimizer='rmsprop', loss={'mask_estimation':
↪'binary_crossentropy','feature_estimation':'mean_squared_error'},
               loss_weights = {'mask_estimation': 1.0, 'feature_estimation':␣
↪alpha})  # Use 1.0 instead of 1


 model.fit(
       x_unlabeled_corrupted,
       {
           'mask_estimation': mask_new,
           'feature_estimation': x_unlabeled
       },
       epochs=epochs,
       batch_size=batch_size
   )
 layer_name = model.layers[1].name
 layer_output = model.get_layer(layer_name).output

 # Assuming model.layers[1] is the desired layer
 encoder = models.Model(inputs=model.input, outputs=model.layers[1].output)

 return encoder
```

[102]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

# Exclude specified columns
exclude_columns = ['Event', 'Time', 'Cell_length', 'file_number',␣
 ↪'event_number', 'label', 'individual']

# Check if all columns in exclude_columns are present in the DataFrame
missing_columns = [col for col in exclude_columns if col not in df.columns]
```

```
# Print the missing columns, if any
if missing_columns:
    print(f"Warning: The following columns are not found in the DataFrame:␣
 ↪{missing_columns}")

# Remove missing columns from exclude_columns
exclude_columns = [col for col in exclude_columns if col in df.columns]

data_filtered = df.drop(columns=exclude_columns)

# Convert all columns to numeric, coercing errors to NaN
for col in data_filtered.columns:
    data_filtered[col] = pd.to_numeric(data_filtered[col], errors='coerce')

# Impute or drop NaN values strategically
# Option 1: Impute with mean/median
# for col in data_filtered.columns:
#     data_filtered[col] = data_filtered[col].fillna(data_filtered[col].mean())

# Option 2: Drop only rows where all values are NaN
# data_filtered = data_filtered.dropna(how='all')

# Option 3: (If a specific column causes most NaNs, and you can drop it):
# data_filtered = data_filtered.drop(columns=['problematic_column']) #Replace␣
 ↪problematic_column
# data_filtered = data_filtered.dropna()


# Standardize the data
scaler = StandardScaler()
x_unlabeled_scaled = scaler.fit_transform(data_filtered)  # Now␣
 ↪x_unlabeled_scaled is defined

# Define other parameters
p_m = 0.3
alpha = 2.0
parameters = {
    'batch_size': 128,
    'epochs': 50,
}

# Run the self_supervised function with the scaled data
encoder_model = self_supervised(x_unlabeled_scaled, p_m, alpha, parameters)
```

```
Warning: The following columns are not found in the DataFrame: ['Event']
Epoch 1/50
```

```
2076/2076            6s 2ms/step -
feature_estimation_loss: 0.6294 - loss: 2.2709 - mask_estimation_loss: 1.6415
Epoch 2/50
2076/2076            4s 2ms/step -
feature_estimation_loss: 0.6087 - loss: 1.9766 - mask_estimation_loss: 1.3679
Epoch 3/50
2076/2076            4s 2ms/step -
feature_estimation_loss: 0.6076 - loss: 1.9675 - mask_estimation_loss: 1.3599
Epoch 4/50
2076/2076            4s 2ms/step -
feature_estimation_loss: 0.6066 - loss: 1.9599 - mask_estimation_loss: 1.3533
Epoch 5/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6061 - loss: 1.9516 - mask_estimation_loss: 1.3455
Epoch 6/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6053 - loss: 1.9519 - mask_estimation_loss: 1.3467
Epoch 7/50
2076/2076            10s 5ms/step -
feature_estimation_loss: 0.6049 - loss: 1.9515 - mask_estimation_loss: 1.3466
Epoch 8/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6048 - loss: 1.9512 - mask_estimation_loss: 1.3464
Epoch 9/50
2076/2076            6s 2ms/step -
feature_estimation_loss: 0.6043 - loss: 1.9474 - mask_estimation_loss: 1.3431
Epoch 10/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6042 - loss: 1.9513 - mask_estimation_loss: 1.3472
Epoch 11/50
2076/2076            5s 3ms/step -
feature_estimation_loss: 0.6038 - loss: 1.9483 - mask_estimation_loss: 1.3445
Epoch 12/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6036 - loss: 1.9467 - mask_estimation_loss: 1.3431
Epoch 13/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6035 - loss: 1.9518 - mask_estimation_loss: 1.3483
Epoch 14/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6035 - loss: 1.9490 - mask_estimation_loss: 1.3455
Epoch 15/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6033 - loss: 1.9482 - mask_estimation_loss: 1.3449
Epoch 16/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6028 - loss: 1.9483 - mask_estimation_loss: 1.3455
Epoch 17/50
```

```
2076/2076            6s 3ms/step -
feature_estimation_loss: 0.6026 - loss: 1.9447 - mask_estimation_loss: 1.3422
Epoch 18/50
2076/2076            6s 3ms/step -
feature_estimation_loss: 0.6026 - loss: 1.9476 - mask_estimation_loss: 1.3450
Epoch 19/50
2076/2076            7s 4ms/step -
feature_estimation_loss: 0.6021 - loss: 1.9468 - mask_estimation_loss: 1.3447
Epoch 20/50
2076/2076            9s 4ms/step -
feature_estimation_loss: 0.6020 - loss: 1.9438 - mask_estimation_loss: 1.3418
Epoch 21/50
2076/2076            8s 4ms/step -
feature_estimation_loss: 0.6018 - loss: 1.9481 - mask_estimation_loss: 1.3463
Epoch 22/50
2076/2076            7s 3ms/step -
feature_estimation_loss: 0.6019 - loss: 1.9443 - mask_estimation_loss: 1.3424
Epoch 23/50
2076/2076            6s 3ms/step -
feature_estimation_loss: 0.6018 - loss: 1.9469 - mask_estimation_loss: 1.3451
Epoch 24/50
2076/2076            6s 3ms/step -
feature_estimation_loss: 0.6012 - loss: 1.9461 - mask_estimation_loss: 1.3448
Epoch 25/50
2076/2076            5s 3ms/step -
feature_estimation_loss: 0.6015 - loss: 1.9435 - mask_estimation_loss: 1.3420
Epoch 26/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6013 - loss: 1.9436 - mask_estimation_loss: 1.3423
Epoch 27/50
2076/2076            5s 3ms/step -
feature_estimation_loss: 0.6011 - loss: 1.9402 - mask_estimation_loss: 1.3391
Epoch 28/50
2076/2076            5s 3ms/step -
feature_estimation_loss: 0.6012 - loss: 1.9466 - mask_estimation_loss: 1.3454
Epoch 29/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6010 - loss: 1.9450 - mask_estimation_loss: 1.3441
Epoch 30/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6009 - loss: 1.9433 - mask_estimation_loss: 1.3424
Epoch 31/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6007 - loss: 1.9451 - mask_estimation_loss: 1.3444
Epoch 32/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6010 - loss: 1.9452 - mask_estimation_loss: 1.3441
Epoch 33/50
```

```
2076/2076            6s 3ms/step -
feature_estimation_loss: 0.6009 - loss: 1.9436 - mask_estimation_loss: 1.3427
Epoch 34/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6010 - loss: 1.9423 - mask_estimation_loss: 1.3413
Epoch 35/50
2076/2076            5s 3ms/step -
feature_estimation_loss: 0.6009 - loss: 1.9419 - mask_estimation_loss: 1.3410
Epoch 36/50
2076/2076            5s 3ms/step -
feature_estimation_loss: 0.6008 - loss: 1.9397 - mask_estimation_loss: 1.3389
Epoch 37/50
2076/2076            6s 3ms/step -
feature_estimation_loss: 0.6010 - loss: 1.9413 - mask_estimation_loss: 1.3403
Epoch 38/50
2076/2076            7s 3ms/step -
feature_estimation_loss: 0.6010 - loss: 1.9427 - mask_estimation_loss: 1.3417
Epoch 39/50
2076/2076            7s 4ms/step -
feature_estimation_loss: 0.6007 - loss: 1.9386 - mask_estimation_loss: 1.3379
Epoch 40/50
2076/2076            6s 3ms/step -
feature_estimation_loss: 0.6009 - loss: 1.9411 - mask_estimation_loss: 1.3402
Epoch 41/50
2076/2076            6s 3ms/step -
feature_estimation_loss: 0.6009 - loss: 1.9409 - mask_estimation_loss: 1.3400
Epoch 42/50
2076/2076            6s 3ms/step -
feature_estimation_loss: 0.6007 - loss: 1.9400 - mask_estimation_loss: 1.3393
Epoch 43/50
2076/2076            6s 3ms/step -
feature_estimation_loss: 0.6010 - loss: 1.9406 - mask_estimation_loss: 1.3396
Epoch 44/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6006 - loss: 1.9438 - mask_estimation_loss: 1.3432
Epoch 45/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6006 - loss: 1.9402 - mask_estimation_loss: 1.3396
Epoch 46/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6007 - loss: 1.9422 - mask_estimation_loss: 1.3415
Epoch 47/50
2076/2076            5s 2ms/step -
feature_estimation_loss: 0.6007 - loss: 1.9429 - mask_estimation_loss: 1.3422
Epoch 48/50
2076/2076            5s 3ms/step -
feature_estimation_loss: 0.6006 - loss: 1.9421 - mask_estimation_loss: 1.3415
Epoch 49/50
```

```
2076/2076             5s 2ms/step -
feature_estimation_loss: 0.6009 - loss: 1.9416 - mask_estimation_loss: 1.3407
Epoch 50/50
2076/2076             5s 2ms/step -
feature_estimation_loss: 0.6007 - loss: 1.9400 - mask_estimation_loss: 1.3393
```

[105]:
```python
import numpy as np
from keras.layers import Input, Dense
from keras.models import Model
from keras.optimizers import Adam

# Function to create the self-supervised model
def self_supervised(x_unlabeled, p_m, alpha, parameters):
    # Extract batch_size and epochs from parameters
    epochs = parameters['epochs']
    batch_size = parameters['batch_size']

    # Get the dimension of the input data
    _, dimension = x_unlabeled.shape

    # Model creation: Defining an encoder (autoencoder structure)
    input_layer = Input(shape=(dimension,))

    # Encoder network
    h = Dense(int(dimension), activation='relu')(input_layer)

    # Output 1: Mask estimation
    output1 = Dense(int(dimension), activation='sigmoid',␣
 ↪name='mask_estimation')(h)

    # Output 2: Feature estimation
    output2 = Dense(int(dimension), activation='sigmoid',␣
 ↪name='feature_estimation')(h)

    # Define the model with inputs and outputs
    model = Model(inputs=input_layer, outputs=[output1, output2])

    # Compile the model (using Adam optimizer and mean squared error for both␣
 ↪outputs)
    model.compile(optimizer=Adam(), loss='mean_squared_error')

    return model
```

[108]:
```python
x_unlabeled_scaled = np.random.rand(1000, 64)  # Example input data with 1000␣
 ↪samples and 64 features

# Set parameters for the model
```

```
p_m = 0.3
alpha = 2.0
parameters = {'batch_size': 128, 'epochs': 50}

# Create the model
encoder = self_supervised(x_unlabeled_scaled, p_m, alpha, parameters)

# Display the summary of the model, which will include parameter counts
encoder.summary()

# Optionally, you can also extract the total number of parameters␣
 ↪programmatically
total_params = encoder.count_params()

# Calculate trainable and non-trainable parameters
trainable_params = np.sum([np.prod(v.shape) for v in encoder.trainable_weights])
non_trainable_params = np.sum([np.prod(v.shape) for v in encoder.
 ↪non_trainable_weights])

# Print the parameter details
print(f"Total params: {total_params} ({total_params / 1024:.2f} KB)")
print(f"Trainable params: {trainable_params} ({trainable_params / 1024:.2f}␣
 ↪KB)")
print(f"Non-trainable params: {non_trainable_params} ({non_trainable_params /␣
 ↪1024:.2f} B)")

# Display optimizer params (the optimizer parameters are usually the variables␣
 ↪in the optimizer, such as momentum, etc.)
optimizer_params = np.sum([np.prod(v.shape) for v in encoder.optimizer.
 ↪variables])
print(f"Optimizer params: {optimizer_params} ({optimizer_params / 1024:.2f}␣
 ↪KB)")
```

```
Model: "functional_4"
```

Layer (type)	Output Shape	Param #	␣ ↪Connected to
input_layer_3 (InputLayer)	(None, 64)	0	- ␣ ↪
dense_3 (Dense)	(None, 64)	4,160	␣ ↪input_layer_3[0][0]

```
 mask_estimation (Dense)         (None, 64)                    4,160 ␣
↪dense_3[0][0]

 feature_estimation (Dense)      (None, 64)                    4,160 ␣
↪dense_3[0][0]


 Total params: 12,480 (48.75 KB)


 Trainable params: 12,480 (48.75 KB)


 Non-trainable params: 0 (0.00 B)
```

```
Total params: 12480 (12.19 KB)
Trainable params: 12480 (12.19 KB)
Non-trainable params: 0.0 (0.00 B)
Optimizer params: 2.0 (0.00 KB)
```

[115]:
```
import os

file_path = "content/encoder.keras"

# Create the 'content' directory if it doesn't exist
os.makedirs(os.path.dirname(file_path), exist_ok=True)

encoder.save(file_path)
```

[116]:
```
encoder_path ="content/encoder.keras"
encoder.save(encoder_path)
```

[117]:
```
from keras.models import load_model

encoder = load_model(encoder_path)
```

```
C:\Users\Jai dabas\anaconda3\Lib\site-
packages\keras\src\saving\saving_lib.py:719: UserWarning: Skipping variable
loading for optimizer 'adam', because it has 14 variables whereas the saved
optimizer has 2 variables.
  saveable.load_own_variables(weights_store.get(inner_path))
```

[1]:
```
import tensorflow as tf
print(tf.__version__)
from tensorflow.keras.layers import Input, Dense
```

```
2.16.2
```

[145]:
```
from keras import layers, models

# Example: Change the input shape to match 37 features (modify this based on␣
 ↪your model's architecture)
input_layer = layers.Input(shape=(37,))  # Adjusted to 37 features instead of 64
# Add the rest of your model layers
# model = models.Sequential([...])

# Re-compile the model and retrain it if necessary.
```

[147]:
```
import numpy as np
from keras.preprocessing.sequence import pad_sequences

# Step 1: Clean the data to keep only numeric columns (drop any non-numeric␣
 ↪ones)
X_train_cleaned_numeric = X_train_cleaned.select_dtypes(include=[np.number])
X_test_cleaned_numeric = X_test_cleaned.select_dtypes(include=[np.number])

# If necessary, explicitly drop unwanted columns (such as 'event_number',␣
 ↪'individual', 'file_number')
# These columns should be explicitly removed if they're present in the dataset
X_train_cleaned_numeric = X_train_cleaned_numeric.drop(columns=['event_number',␣
 ↪'individual', 'file_number'], errors='ignore')
X_test_cleaned_numeric = X_test_cleaned_numeric.drop(columns=['event_number',␣
 ↪'individual', 'file_number'], errors='ignore')

# Step 2: Pad the data to match the model's expected input shape (64 features)
# Pad the sequences with zeros to ensure shape (None, 64)
X_train_padded = pad_sequences(X_train_cleaned_numeric, maxlen=64,␣
 ↪padding='post', dtype='float32')
X_test_padded = pad_sequences(X_test_cleaned_numeric, maxlen=64,␣
 ↪padding='post', dtype='float32')

# Verify the padded shapes
print("Shape of X_train:", X_train_padded.shape)
print("Shape of X_test:", X_test_padded.shape)

# Step 3: Use the encoder model to generate encoded representations of the␣
 ↪cleaned and padded data
X_train_encoded = encoder_model.predict(X_train_padded)
X_test_encoded = encoder_model.predict(X_test_padded)

# Step 4: Proceed with Logistic Regression (assuming y_train and y_test are␣
 ↪available)
log_reg = LogisticRegression(max_iter=1000)
log_reg.fit(X_train_encoded, y_train)
```

```
# Get predicted probabilities on the test set
y_test_probabilities = log_reg.predict_proba(X_test_encoded)

# Calculate log loss
loss_value = log_loss(y_test, y_test_probabilities)

# Print results
print("Predicted probabilities:\n", y_test_probabilities)
print("\nLog Loss:", loss_value)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[147], line 15
     11 X_test_cleaned_numeric = X_test_cleaned_numeric.
 ↪drop(columns=['event_number', 'individual', 'file_number'], errors='ignore')
     13 # Step 2: Pad the data to match the model's expected input shape (64␣
 ↪features)
     14 # Pad the sequences with zeros to ensure shape (None, 64)
---> 15 X_train_padded = pad_sequences(X_train_cleaned_numeric, maxlen=64,␣
 ↪padding='post', dtype='float32')
     16 X_test_padded = pad_sequences(X_test_cleaned_numeric, maxlen=64,␣
 ↪padding='post', dtype='float32')
     18 # Verify the padded shapes

File ~\anaconda3\Lib\site-packages\keras\src\utils\sequence_utils.py:125, in␣
 ↪pad_sequences(sequences, maxlen, dtype, padding, truncating, value)
    122     raise ValueError(f'Truncating type "{truncating}" not understood')
    124 # check `trunc` has expected shape
--> 125 trunc = np.asarray(trunc, dtype=dtype)
    126 if trunc.shape[1:] != sample_shape:
    127     raise ValueError(
    128         f"Shape of sample {trunc.shape[1:]} of sequence at "
    129         f"position {idx} is different from expected shape "
    130         f"{sample_shape}"
    131     )

ValueError: could not convert string to float: 'Time'
```

[149]:
```
import tensorflow as tf
from tensorflow.keras import layers, models

def model(input_dimension, hidden_dimension, label_dimension, activation=tf.nn.
 ↪relu):
    """
    Define the model architecture with two hidden layers and softmax output.
```

```
    Args:
    - input_dimension (int or tuple): The number of input features (or shape of␣
↪the input).
    - hidden_dimension (int): The number of neurons in each hidden layer.
    - label_dimension (int): The number of output labels.
    - activation (function): The activation function to use for hidden layers␣
↪(default is ReLU).

    Returns:
    - model (tf.keras.Model): The compiled neural network model.
    """
    # Define the input layer with specified shape and name
    inputs = tf.keras.Input(shape=(input_dimension,), name='model_input')

    # First hidden layer
    x = layers.Dense(hidden_dimension, activation=activation,␣
↪name='model_dense_layer_1')(inputs)

    # Second hidden layer
    x = layers.Dense(hidden_dimension, activation=activation,␣
↪name='model_dense_layer_2')(x)

    # Logit output layer (without activation)
    y_logit = layers.Dense(label_dimension, activation=None,␣
↪name='model_logit_output')(x)

    # Final softmax activation layer for the actual prediction
    y = layers.Activation('softmax', name='model_output')(y_logit)

    # Create the model with input and output layers
    model = models.Model(inputs=inputs, outputs=[y_logit, y],␣
↪name="custom_model")

    # Compile the model with an optimizer, loss, and metrics
    model.compile(optimizer='adam', loss='categorical_crossentropy',␣
↪metrics=['accuracy'])

    return model
```

[150]:
```
def train(feature_batch, label_batch, unlabeled_feature_batch, model, beta,␣
↪supv_loss_fn, optimizer):
    """
    Train the model on the batch data with both labeled and unlabeled data.

    Args:
```

```
    - feature_batch (tensor): A batch of feature data (labeled).
    - label_batch (tensor): A batch of labels (labeled).
    - unlabeled_feature_batch (tensor): A batch of unlabeled feature data.
    - model (tf.keras.Model): The model to train.
    - beta (float): Regularization term for the semi-supervised loss.
    - supv_loss_fn (function): The supervised loss function.
    - optimizer (tf.optimizers): Optimizer used for training.

    Returns:
    - total_loss (float): The total loss during training.
    """
    with tf.GradientTape() as tape:
        # Forward pass for labeled data
        y_logit, y = model(feature_batch, training=True)  # Get outputs for␣
↪labeled data
        y_loss = supv_loss_fn(label_batch, y_logit)  # Calculate supervised␣
↪loss function for labeled data

        # Forward pass for unlabeled data
        unlabeled_y_logit, unlabeled_y = model(unlabeled_feature_batch,␣
↪training=True)  # Get outputs for unlabeled data

        # Loss function for unlabeled data - reduce variance
        unlabeled_y_loss = tf.reduce_mean(tf.nn.moments(unlabeled_y_logit,␣
↪axes=0)[1])  # Penalize variance of outputs

        # Combine supervised loss and unsupervised loss
        total_loss = y_loss + beta * unlabeled_y_loss  # Loss formula: beta is␣
↪a hyperparameter

    # Calculate gradients and apply the optimizer
    grads = tape.gradient(total_loss, model.trainable_weights)  # Calculate␣
↪gradients
    optimizer.apply_gradients(zip(grads, model.trainable_weights))  # Apply␣
↪gradients to update weights

    return total_loss
```

[140]:

[ ]: