```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from google.colab import drive
drive.mount('/content/drive')

# Correct file path to your file in Google Drive
file_path = '/content/drive/MyDrive/dataset/data.csv'

# Read the file using the correct variable
df = pd.read_csv(file_path)
new_df =
df.drop(['Event','Time','file_number','event_number','individual'],axi
s=1)
display(df)
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
```

```
{"type":"dataframe","variable_name":"df"}
```

```
print("Basic Structure of the Data:")
display(df)
```

```
Basic Structure of the Data:
```

```
{"type":"dataframe","variable_name":"df"}
```

```
print("\nData Information:")
display(df.info())
```

```
Data Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 265627 entries, 0 to 265626
Data columns (total 42 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   Event         265627 non-null  int64
 1   Time          265627 non-null  float64
 2   Cell_length   265627 non-null  int64
 3   DNA1          265627 non-null  float64
 4   DNA2          265627 non-null  float64
 5   CD45RA        265627 non-null  float64
 6   CD133         265627 non-null  float64
 7   CD19          265627 non-null  float64
 8   CD22          265627 non-null  float64
 9   CD11b         265627 non-null  float64
```

```
 10  CD4             265627 non-null  float64
 11  CD8             265627 non-null  float64
 12  CD34            265627 non-null  float64
 13  Flt3            265627 non-null  float64
 14  CD20            265627 non-null  float64
 15  CXCR4           265627 non-null  float64
 16  CD235ab         265627 non-null  float64
 17  CD45            265627 non-null  float64
 18  CD123           265627 non-null  float64
 19  CD321           265627 non-null  float64
 20  CD14            265627 non-null  float64
 21  CD33            265627 non-null  float64
 22  CD47            265627 non-null  float64
 23  CD11c           265627 non-null  float64
 24  CD7             265627 non-null  float64
 25  CD15            265627 non-null  float64
 26  CD16            265627 non-null  float64
 27  CD44            265627 non-null  float64
 28  CD38            265627 non-null  float64
 29  CD13            265627 non-null  float64
 30  CD3             265627 non-null  float64
 31  CD61            265627 non-null  float64
 32  CD117           265627 non-null  float64
 33  CD49d           265627 non-null  float64
 34  HLA-DR          265627 non-null  float64
 35  CD64            265627 non-null  float64
 36  CD41            265627 non-null  float64
 37  Viability       265627 non-null  float64
 38  file_number     265627 non-null  float64
 39  event_number    265627 non-null  int64
 40  label           104184 non-null  float64
 41  individual      265627 non-null  int64
dtypes: float64(38), int64(4)
memory usage: 85.1 MB

None
```

```
print("\nMissing Values:")
display(df.isnull().sum())
```

```
Missing Values:

Event                 0
Time                  0
Cell_length           0
DNA1                  0
DNA2                  0
CD45RA                0
CD133                 0
```

```
CD19                  0
CD22                  0
CD11b                 0
CD4                   0
CD8                   0
CD34                  0
Flt3                  0
CD20                  0
CXCR4                 0
CD235ab               0
CD45                  0
CD123                 0
CD321                 0
CD14                  0
CD33                  0
CD47                  0
CD11c                 0
CD7                   0
CD15                  0
CD16                  0
CD44                  0
CD38                  0
CD13                  0
CD3                   0
CD61                  0
CD117                 0
CD49d                 0
HLA-DR                0
CD64                  0
CD41                  0
Viability             0
file_number           0
event_number          0
label            161443
individual            0
dtype: int64
```

```
print("\nMissing Values:")
missing_values = df.isnull().sum()
missing_percentage = (missing_values / len(df)) * 100
missing_df = pd.DataFrame({'Missing Values': missing_values,
'Percentage': missing_percentage})
display(missing_df[missing_df['Missing Values'] > 0])
```

```
Missing Values:
```


```
{"summary":"{\n  \"name\": \"display(missing_df[missing_df['Missing
Values'] > 0])\",\n  \"rows\": 1,\n  \"fields\": [\n    {\n
\"column\": \"Missing Values\",\n      \"properties\": {\n
```

\"dtype\": \"number\",\n   \"std\": null,\n   \"min\": 161443,\n   \"max\": 161443,\n   \"num_unique_values\": 1,\n \"samples\": [\n   161443\n   ],\n \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Percentage\",\n \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": null,\n   \"min\": 60.778083553253246,\n   \"max\": 60.778083553253246,\n   \"num_unique_values\": 1,\n \"samples\": [\n   60.778083553253246\n   ],\n \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe"}

```
df.nunique()
```

Event	265627
Time	147364
Cell_length	56
DNA1	158903
DNA2	157946
CD45RA	161079
CD133	161181
CD19	161229
CD22	161247
CD11b	161236
CD4	161258
CD8	161213
CD34	161214
Flt3	161204
CD20	161224
CXCR4	161121
CD235ab	161039
CD45	158665
CD123	161241
CD321	160654
CD14	161193
CD33	161233
CD47	160534
CD11c	161251
CD7	161257
CD15	161194
CD16	161202
CD44	160151
CD38	161143
CD13	161232
CD3	160856
CD61	161241
CD117	161212
CD49d	161040
HLA-DR	161214
CD64	161238

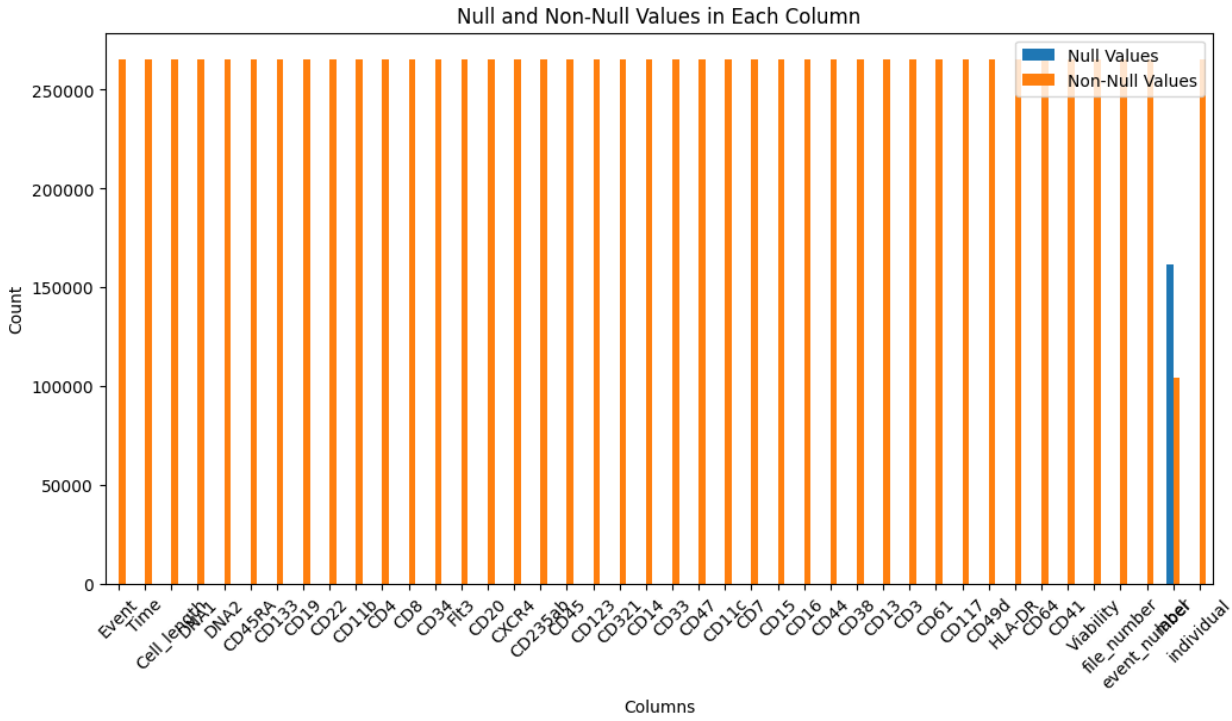
```
CD41            161232
Viability       161149
file_number          2
event_number    151613
label               14
individual           2
dtype: int64
```

```
print("\nDescriptive Statistics:")
display(df.describe())
```

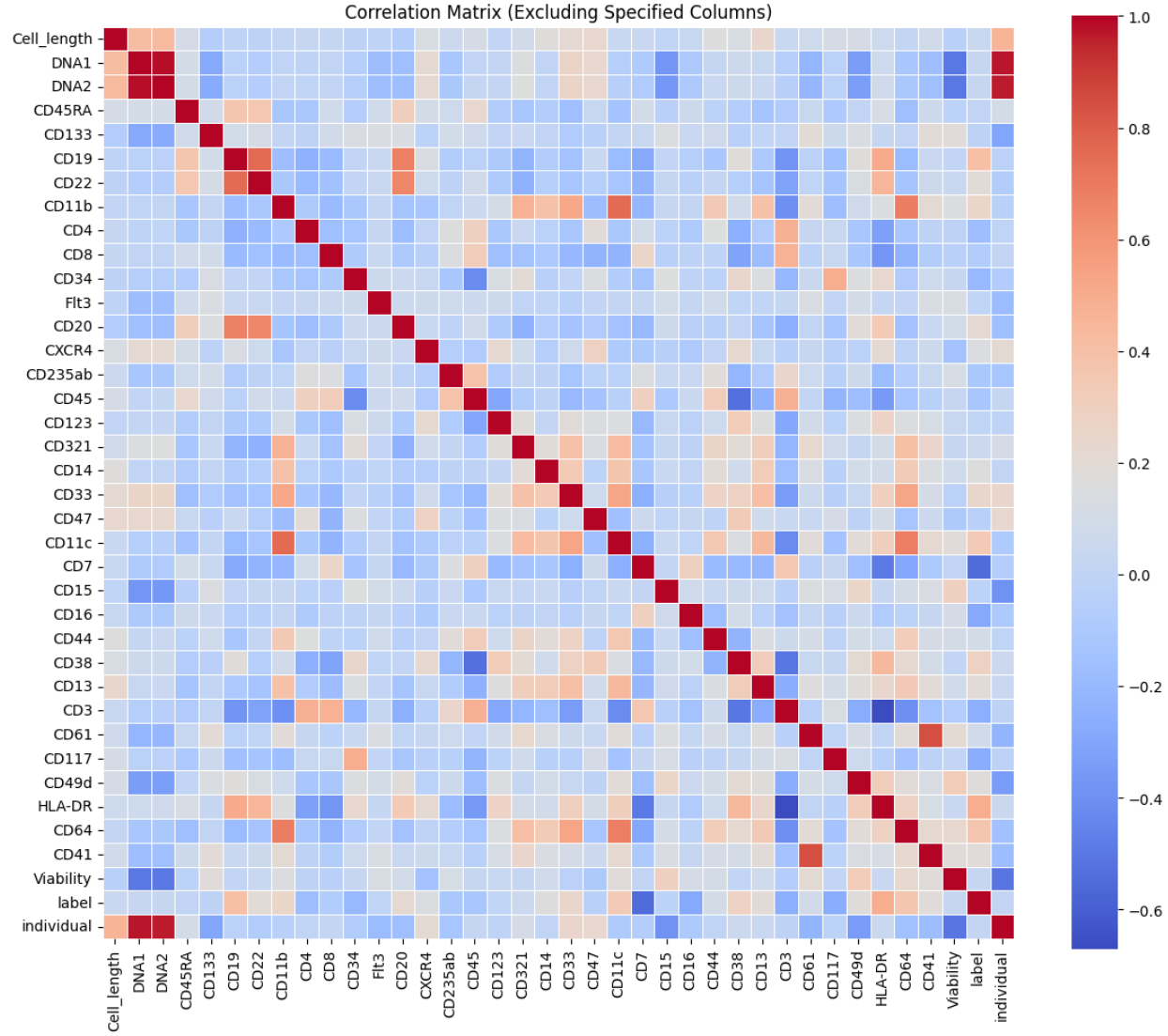
```
Descriptive Statistics:
```

```
{"type":"dataframe"}
```

```
null_counts = df.isnull().sum()
non_null_counts = df.notnull().sum()


plot_data = pd.DataFrame({
    'Null Values': null_counts,
    'Non-Null Values': non_null_counts
})

plot_data.plot(kind='bar', figsize=(12, 6))
plt.title('Null and Non-Null Values in Each Column')
plt.xlabel('Columns')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.legend(loc='upper right')
plt.show()
```




```
print("\nCorrelation Matrix:")
columns_to_exclude = ['Event', 'Time', 'file_number', 'event_number']
filtered_df = df.drop(columns=columns_to_exclude)

correlation_matrix = filtered_df.corr()

plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm',
square=True, cbar=True, fmt='', linewidths=0.5)
plt.title('Correlation Matrix (Excluding Specified Columns)')
plt.tight_layout()
plt.show()


Correlation Matrix:
```

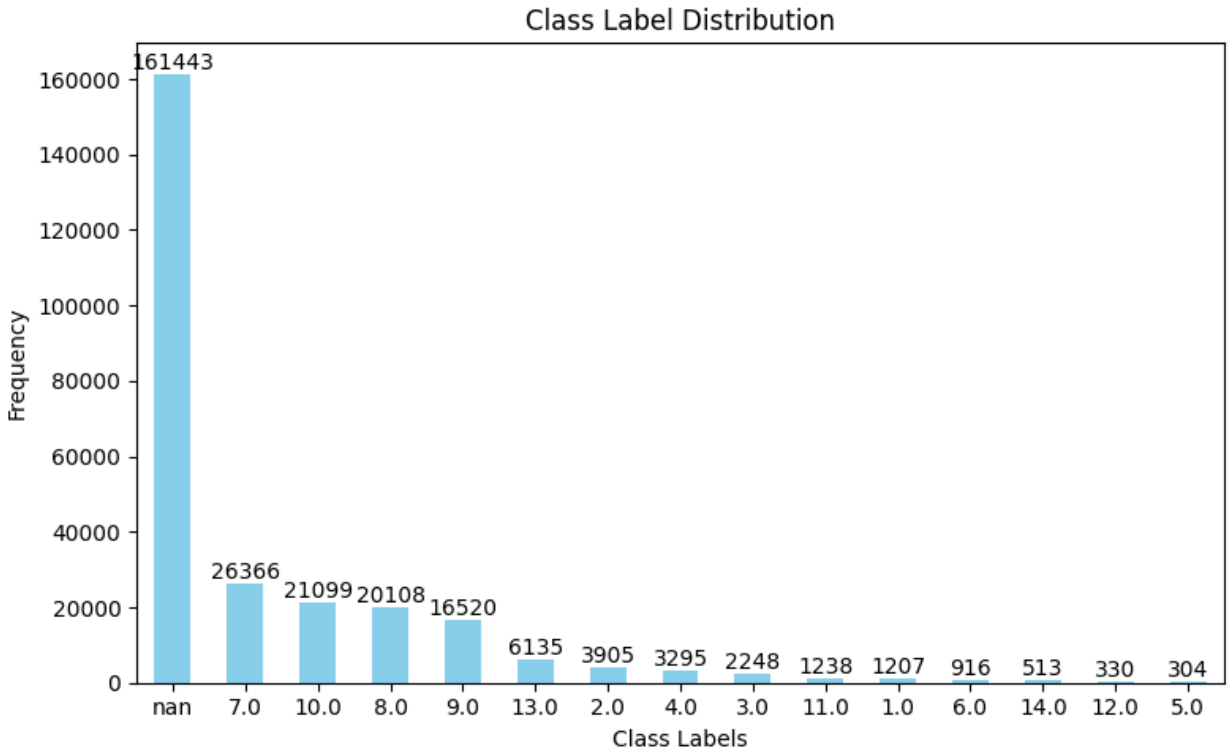



```
label_distribution = df['label'].value_counts(dropna=False)
print("Class Label Distribution:")
print(label_distribution)
```

```
Class Label Distribution:
label
NaN     161443
7.0      26366
10.0     21099
8.0      20108
9.0      16520
13.0      6135
2.0       3905
4.0       3295
3.0       2248
11.0      1238
```

```
1.0       1207
6.0        916
14.0       513
12.0       330
5.0        304
Name: count, dtype: int64
```

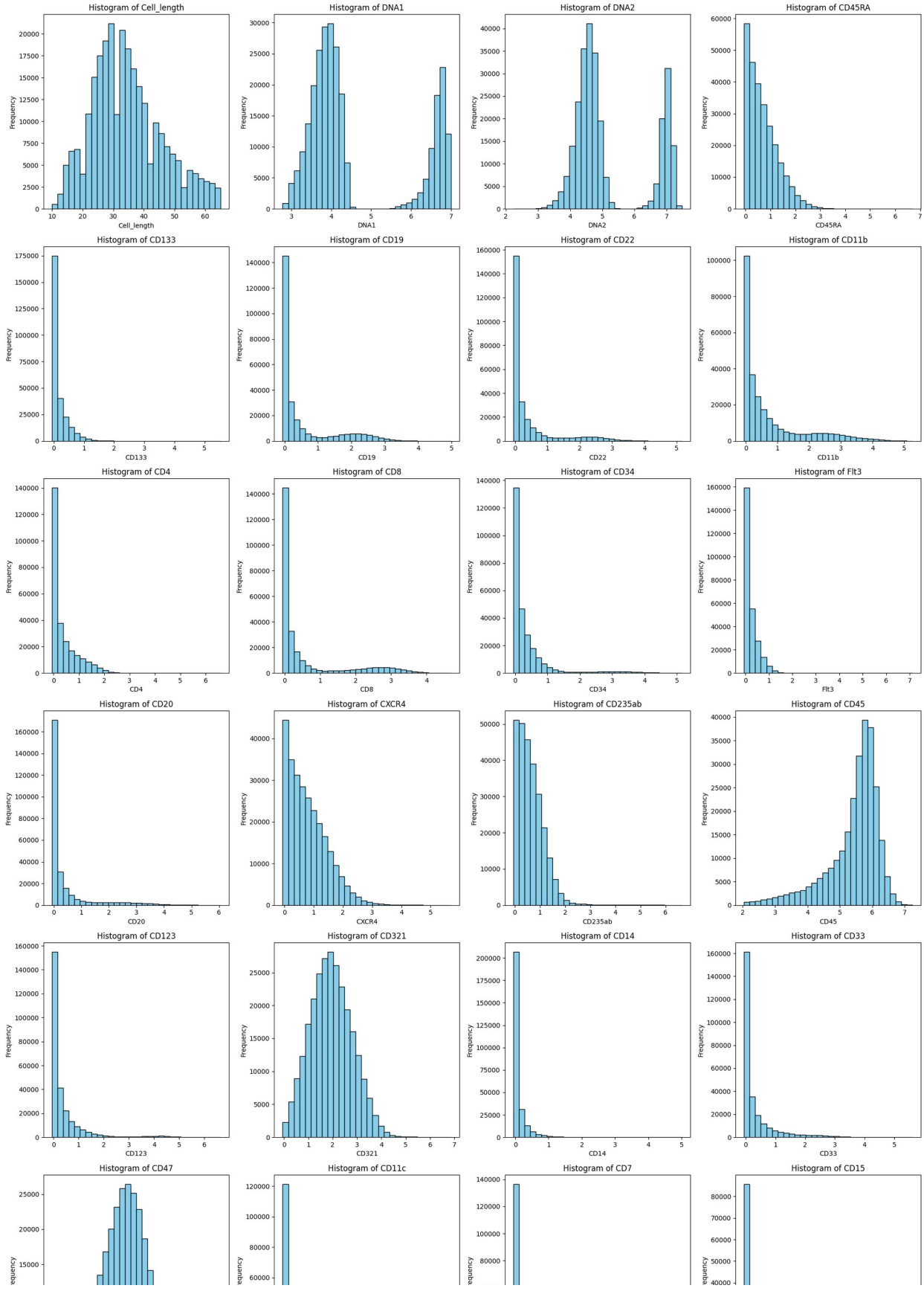
```
label_distribution = df['label'].value_counts(dropna=False)

plt.figure(figsize=(8, 5))
bars = label_distribution.plot(kind='bar', color='skyblue')
plt.title('Class Label Distribution')
plt.xlabel('Class Labels')
plt.ylabel('Frequency')
plt.xticks(rotation=0)

for bar in bars.patches:
    bars.annotate(bar.get_height(),
                  (bar.get_x() + bar.get_width() / 2,
bar.get_height()),
                  ha='center',
                  va='bottom')

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



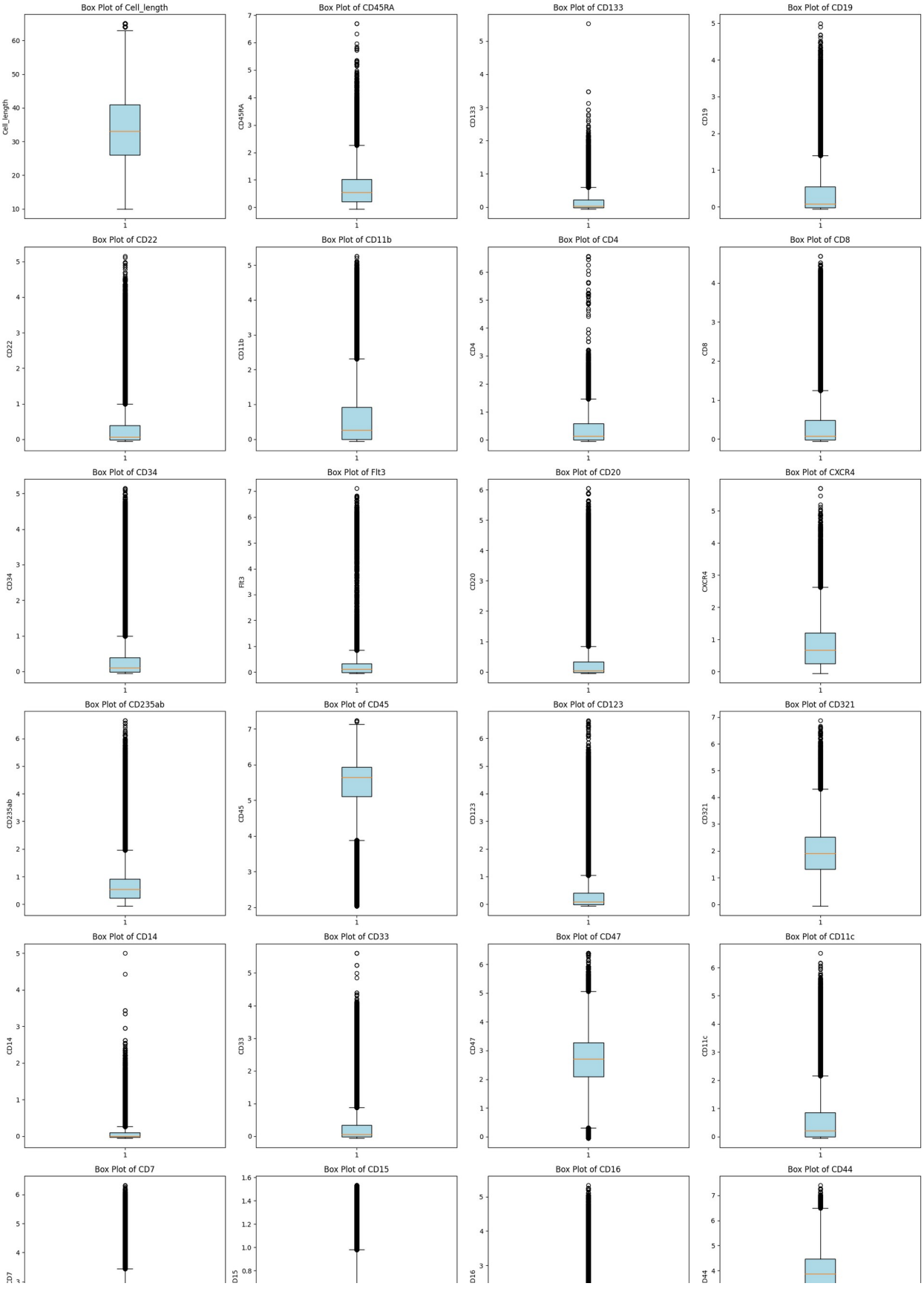
```
corr_matrix = df.corr()

corr_pairs = corr_matrix.unstack().sort_values(ascending=False)

corr_pairs = corr_pairs[corr_pairs.index.get_level_values(0) !=
corr_pairs.index.get_level_values(1)]

top_5_corr_pairs = corr_pairs.head(10)

print("Top 10 correlated column pairs:")
for (col1, col2), corr_value in top_5_corr_pairs.items():
    print(f"{col1} and {col2}: Correlation = {corr_value:.2f}")
```

```
Top 10 correlated column pairs:
individual and file_number: Correlation = 1.00
file_number and individual: Correlation = 1.00
DNA1 and DNA2: Correlation = 0.98
DNA2 and DNA1: Correlation = 0.98
DNA1 and individual: Correlation = 0.97
individual and DNA1: Correlation = 0.97
file_number and DNA1: Correlation = 0.97
DNA1 and file_number: Correlation = 0.97
DNA2 and individual: Correlation = 0.96
individual and DNA2: Correlation = 0.96
```

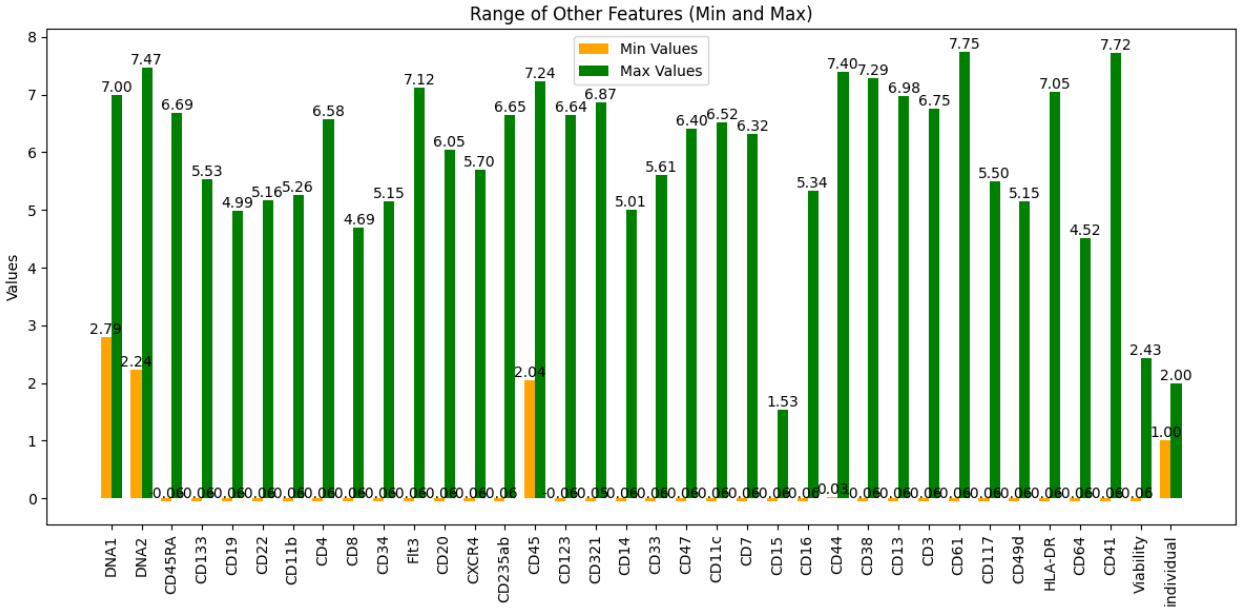
```
columns_to_exclude = ['Event', 'Time', 'file_number', 'event_number',
'individual']
columns_to_plot = [col for col in df.columns if col not in
columns_to_exclude]

num_cols = 4
num_rows = (len(columns_to_plot) + num_cols - 1) // num_cols

plt.figure(figsize=(20, num_rows * 5))

for i, column in enumerate(columns_to_plot):
    plt.subplot(num_rows, num_cols, i + 1)
    plt.hist(df[column], bins=30, color='skyblue', edgecolor='black')
    plt.title(f'Histogram of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')

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


Histogram of Cell_length
Cell_length
Frequency
Histogram of DNA1
DNA1
Frequency
Histogram of DNA2
DNA2
Frequency
Histogram of CD45RA
CD45RA
Frequency
Histogram of CD133
CD133
Frequency
Histogram of CD19
CD19
Frequency
Histogram of CD22
CD22
Frequency
Histogram of CD11b
CD11b
Frequency
Histogram of CD4
CD4
Frequency
Histogram of CD8
CD8
Frequency
Histogram of CD34
CD34
Frequency
Histogram of Flt3
Flt3
Frequency
Histogram of CD20
CD20
Frequency
Histogram of CXCR4
CXCR4
Frequency
Histogram of CD235ab
CD235ab
Frequency
Histogram of CD45
CD45
Frequency
Histogram of CD123
CD123
Frequency
Histogram of CD321
CD321
Frequency
Histogram of CD14
CD14
Frequency
Histogram of CD33
CD33
Frequency
Histogram of CD47
Histogram of CD11c
Histogram of CD7
Histogram of CD15

```
columns_to_exclude = ['Event', 'DNA1', 'DNA2', 'Time', 'file_number',
'event_number', 'Viability', 'label']
columns_to_plot = [col for col in df.columns if col not in
columns_to_exclude]

num_cols = 4
num_rows = (len(columns_to_plot) + num_cols - 1) // num_cols

plt.figure(figsize=(20, num_rows * 5))

for i, column in enumerate(columns_to_plot):
    plt.subplot(num_rows, num_cols, i + 1)
    plt.boxplot(df[column], patch_artist=True,
boxprops=dict(facecolor='lightblue'))
    plt.title(f'Box Plot of {column}')
    plt.ylabel(column)

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


Box Plot of Cell_length
Box Plot of CD45RA
Box Plot of CD133
Box Plot of CD19
Box Plot of CD22
Box Plot of CD11b
Box Plot of CD4
Box Plot of CD8
Box Plot of CD34
Box Plot of Flt3
Box Plot of CD20
Box Plot of CXCR4
Box Plot of CD235ab
Box Plot of CD45
Box Plot of CD123
Box Plot of CD321
Box Plot of CD14
Box Plot of CD33
Box Plot of CD47
Box Plot of CD11c
Box Plot of CD7
Box Plot of CD15
Box Plot of CD16
Box Plot of CD44

```
high_value_columns = ['Time', 'Event', 'event_number', 'label',
'Cell_length', 'file_number']
other_columns = [col for col in df.columns if col not in
high_value_columns]

min_values_other = df[other_columns].min()
max_values_other = df[other_columns].max()

fig, ax = plt.subplots(figsize=(12, 6))

x_other = np.arange(len(other_columns))
width = 0.35

bars_min_other = ax.bar(x_other - width/2, min_values_other, width,
label='Min Values', color='orange')
bars_max_other = ax.bar(x_other + width/2, max_values_other, width,
label='Max Values', color='green')

ax.set_ylabel('Values')
ax.set_title('Range of Other Features (Min and Max)')
ax.set_xticks(x_other)
ax.set_xticklabels(other_columns, rotation=90)
ax.legend()

for bar in bars_min_other:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}',
ha='center', va='bottom')

for bar in bars_max_other:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.2f}',
ha='center', va='bottom')

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



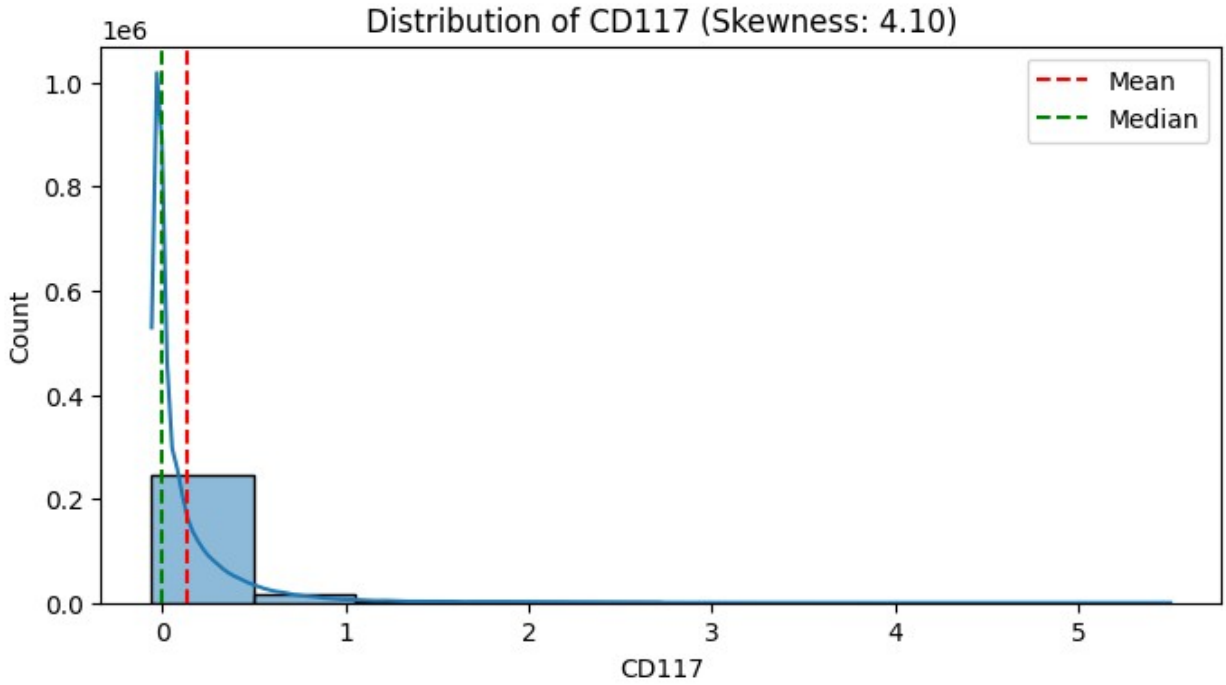

```
print("Skewness")
from scipy.stats import skew
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Exclude specific columns from the skewness calculation
columns_to_exclude = ['Event', 'Time', 'file_number', 'event_number',
'individual']
numerical_columns =
df.select_dtypes(include=['number']).columns.difference(columns_to_exc
lude)

# Calculate skewness for the specified numerical columns
skewness = df[numerical_columns].apply(skew)

# Function to categorize skewness
def categorize_skewness(value):
    if value > 0.5:
        return 'Right-skewed'
    elif value < -0.5:
        return 'Left-skewed'
    else:
        return 'Approximately symmetrical'

# Apply the categorization
skewness_category = skewness.apply(categorize_skewness)

# Display skewness and its categorization
skewness_df = pd.DataFrame({'Skewness': skewness, 'Category':
skewness_category})
```

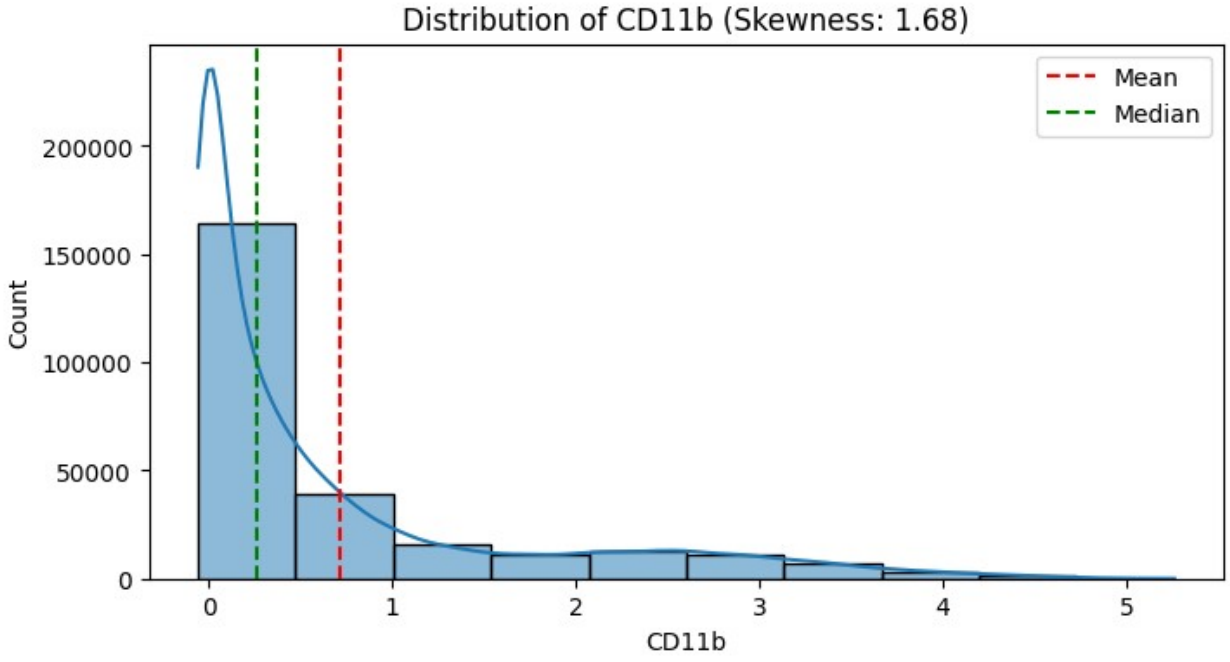
```
print(skewness_df)

# Plot histograms for each numerical column, excluding specified
columns
for col in numerical_columns:
    plt.figure(figsize=(8, 4))
    sns.histplot(df[col], bins=10, kde=True)
    plt.title(f'Distribution of {col} (Skewness:
{skewness[col]:.2f})')
    plt.axvline(df[col].mean(), color='red', linestyle='--',
label='Mean')
    plt.axvline(df[col].median(), color='green', linestyle='--',
label='Median')
    plt.legend()
    plt.show()
```

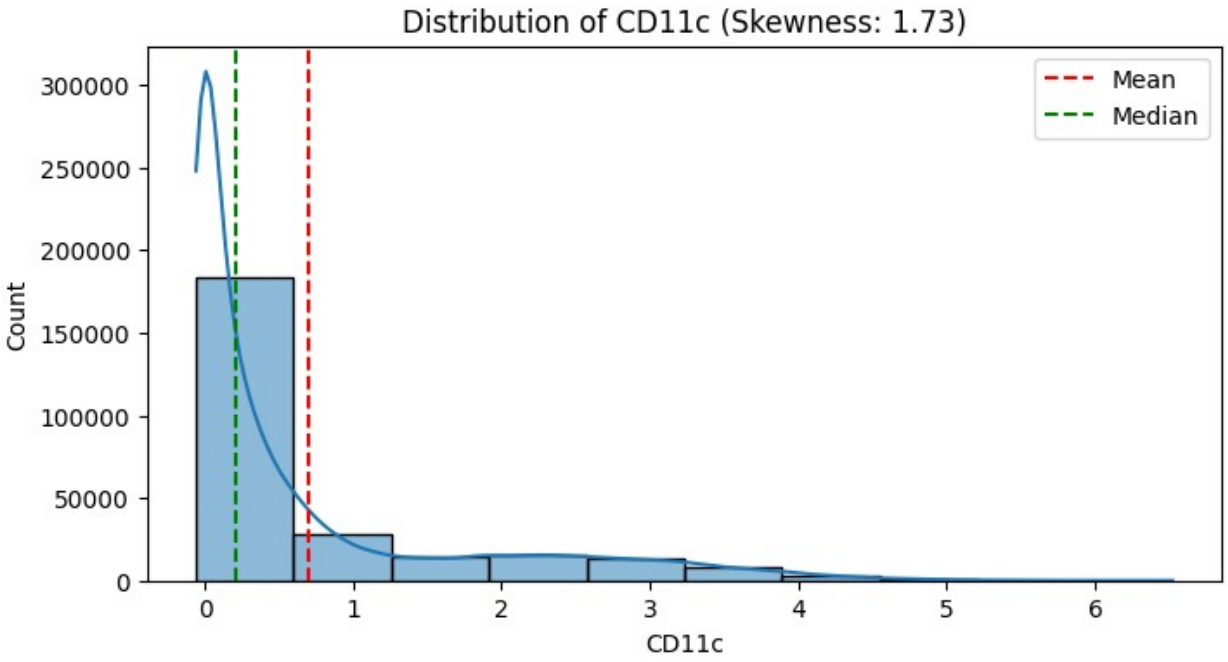
Skewness

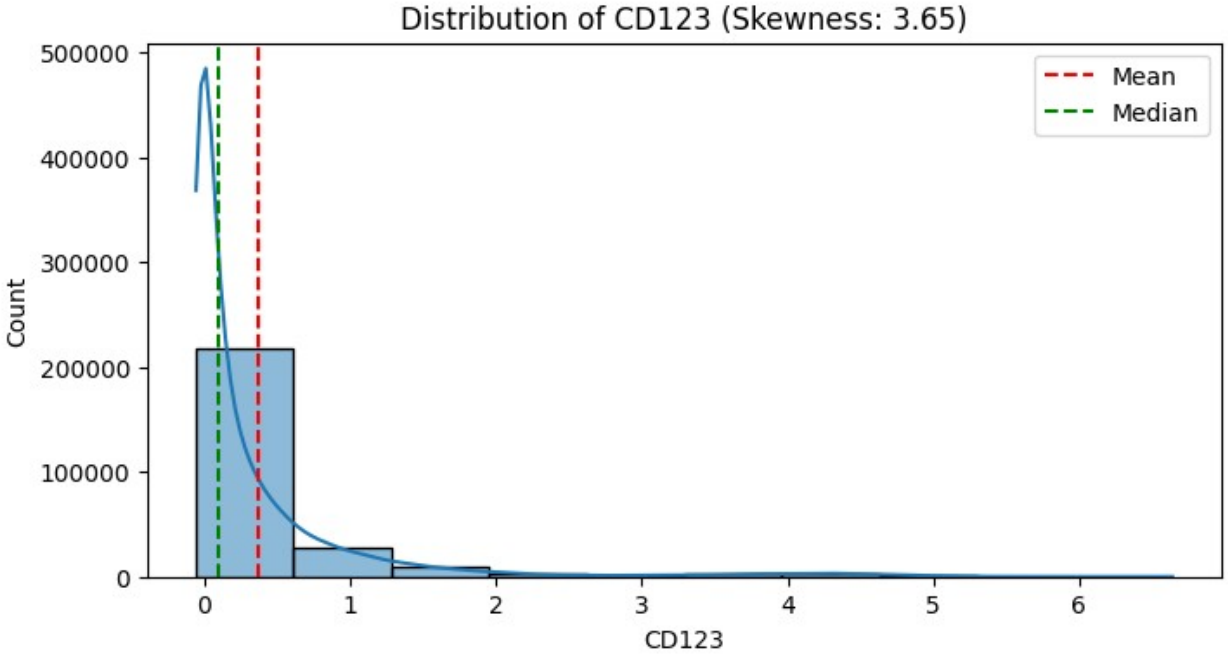
	Skewness	Category
CD117	4.097508	Right-skewed
CD11b	1.679089	Right-skewed
CD11c	1.733888	Right-skewed
CD123	3.648890	Right-skewed
CD13	2.234311	Right-skewed
CD133	2.141953	Right-skewed
CD14	3.609006	Right-skewed
CD15	1.445147	Right-skewed
CD16	5.733203	Right-skewed
CD19	1.682609	Right-skewed
CD20	2.754699	Right-skewed
CD22	2.283181	Right-skewed
CD235ab	2.001479	Right-skewed
CD3	0.342239	Approximately symmetrical
CD321	0.247097	Approximately symmetrical
CD33	2.724977	Right-skewed
CD34	3.492437	Right-skewed
CD38	1.141482	Right-skewed
CD4	1.622044	Right-skewed
CD41	5.366314	Right-skewed
CD44	-0.431589	Approximately symmetrical
CD45	-1.484824	Left-skewed
CD45RA	1.191595	Right-skewed
CD47	-0.250323	Approximately symmetrical
CD49d	0.856805	Right-skewed
CD61	4.894707	Right-skewed
CD64	1.743733	Right-skewed
CD7	1.606528	Right-skewed
CD8	1.775713	Right-skewed
CXCR4	0.955342	Right-skewed
Cell_length	0.527832	Right-skewed
DNA1	0.845010	Right-skewed

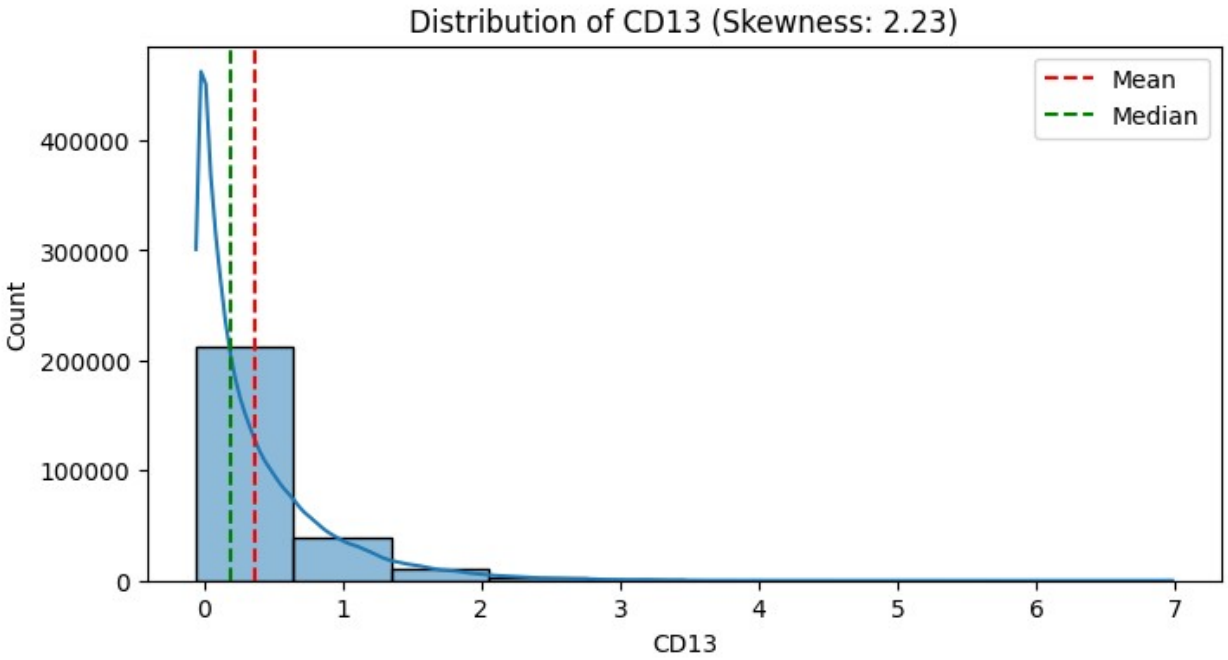
```
DNA2          0.779167                Right-skewed
Flt3          7.098151                Right-skewed
HLA-DR        0.795359                Right-skewed
Viability     0.985417                Right-skewed
label              NaN  Approximately symmetrical
```

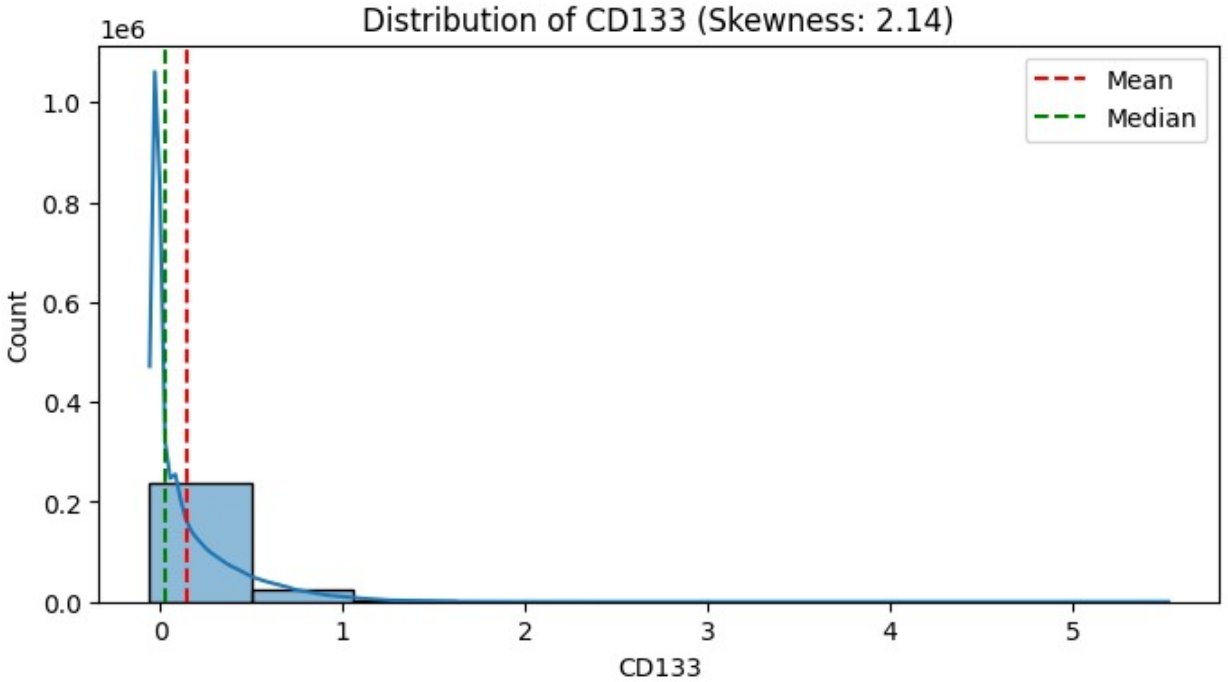


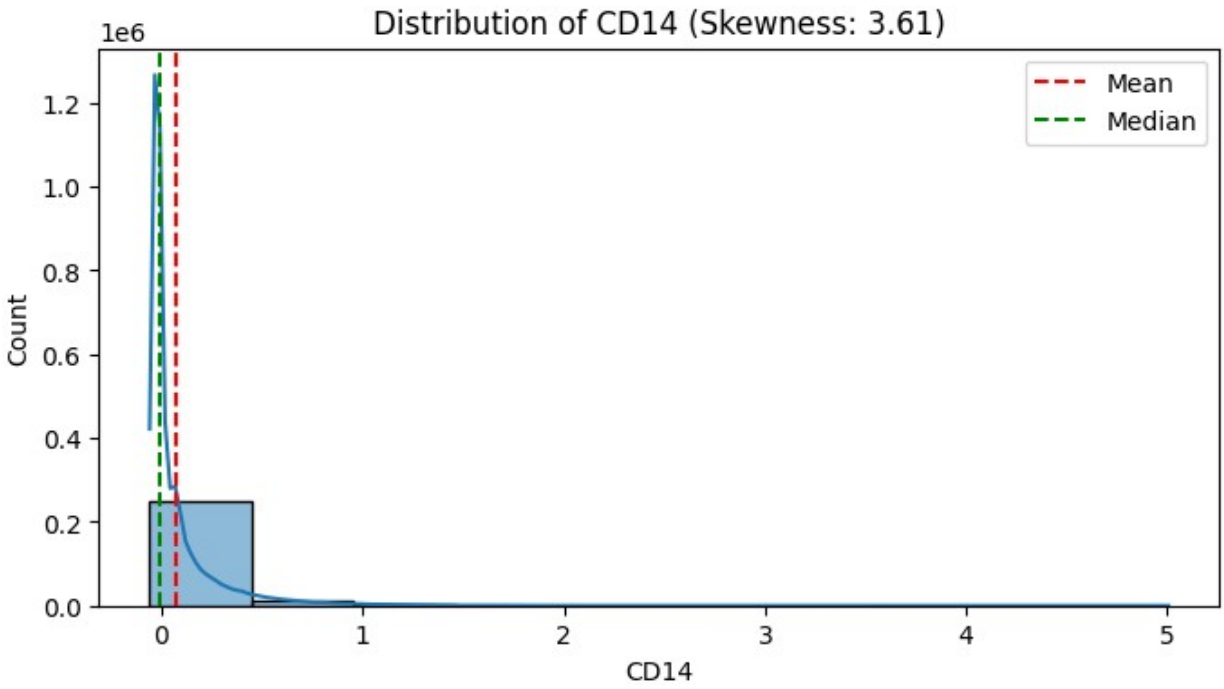


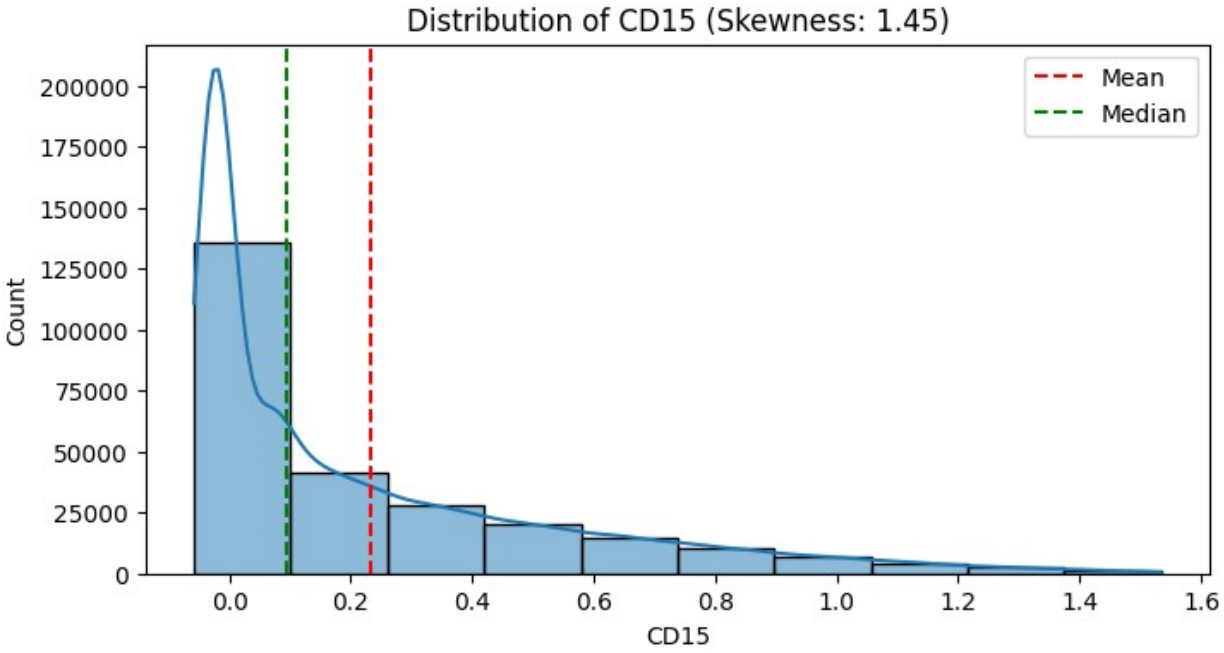

Distribution of CD11c (Skewness: 1.73)
Mean
Median
Count
CD11c
0
50000
100000
150000
200000
250000
300000
0
1
2
3
4
5
6

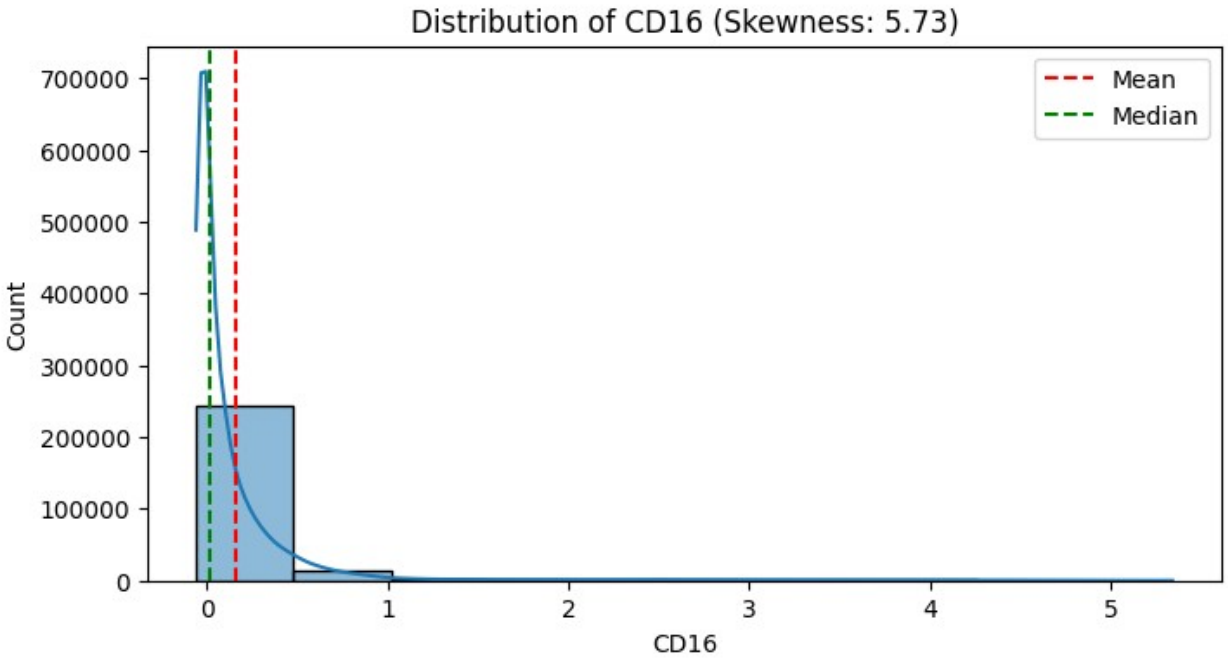

Distribution of CD123 (Skewness: 3.65)
Mean
Median
Count
CD123
0
100000
200000
300000
400000
500000
0
1
2
3
4
5
6

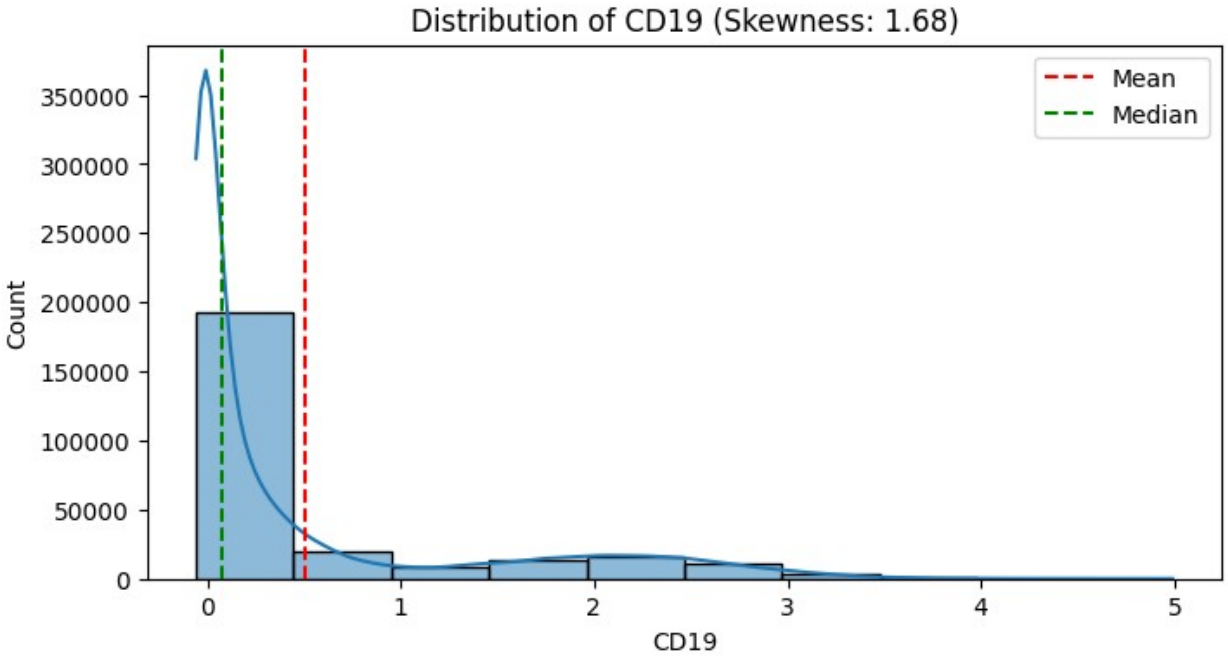

Distribution of CD13 (Skewness: 2.23)
Mean
Median
Count
CD13
0
100000
200000
300000
400000
0
1
2
3
4
5
6
7

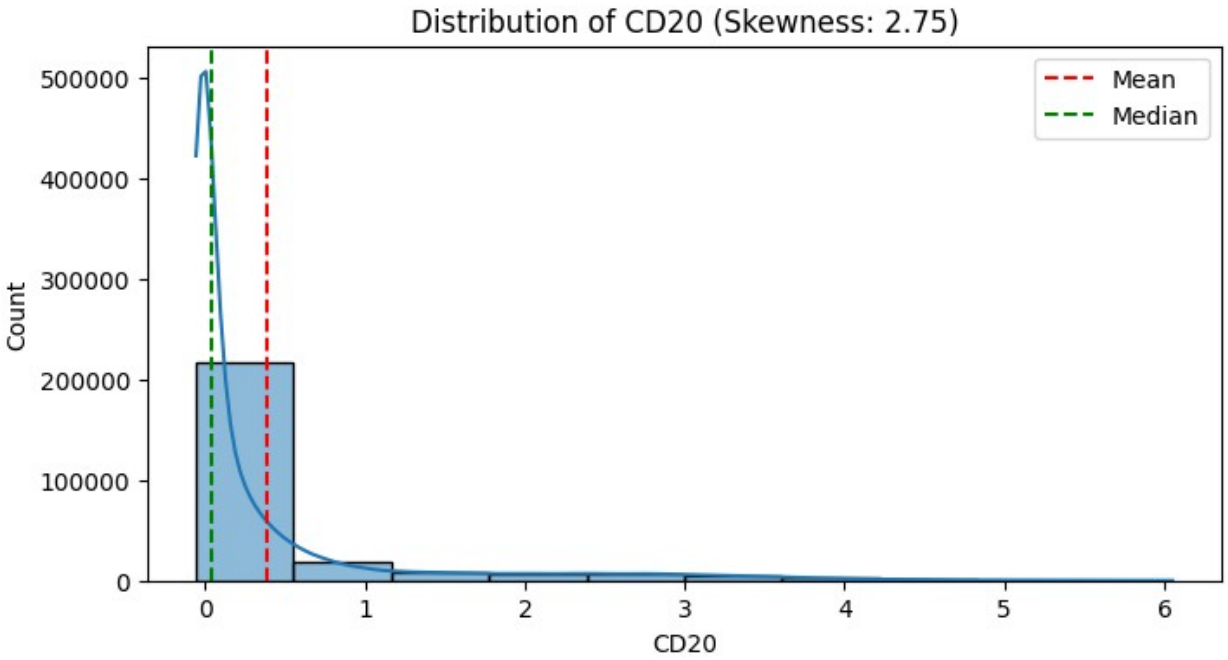

Distribution of CD133 (Skewness: 2.14)
1e6
Mean
Median
Count
CD133
0.0
0.2
0.4
0.6
0.8
1.0
0
1
2
3
4
5

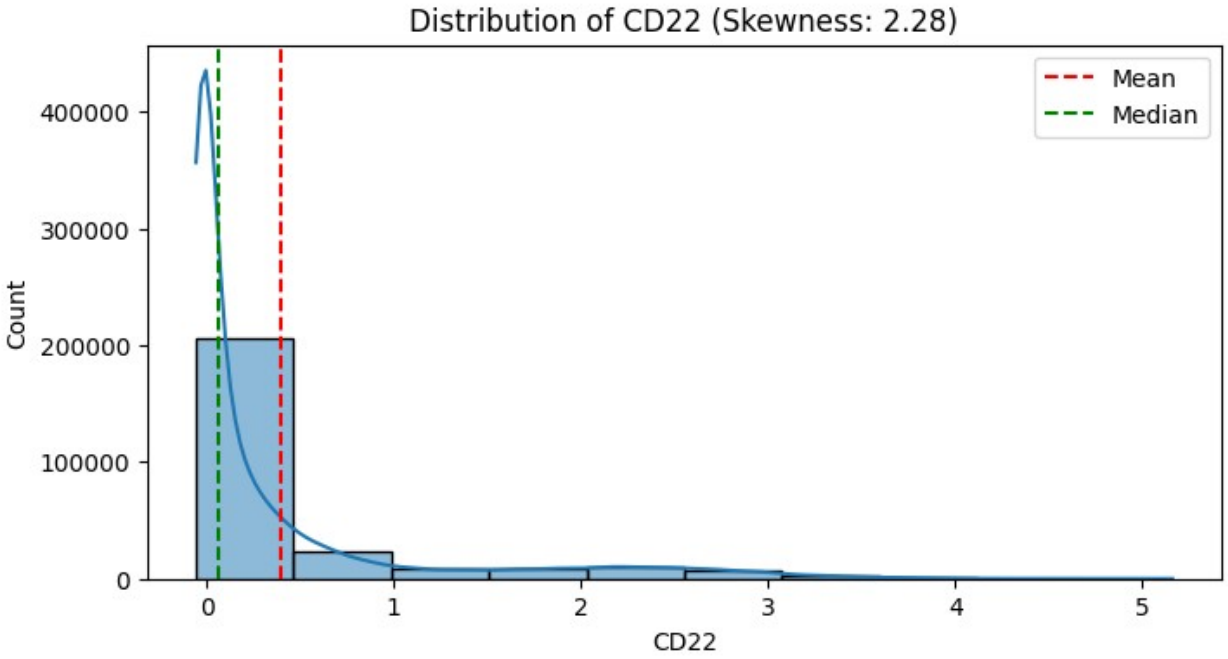

Distribution of CD14 (Skewness: 3.61)
1e6
Mean
Median
Count
CD14
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
1
2
3
4
5

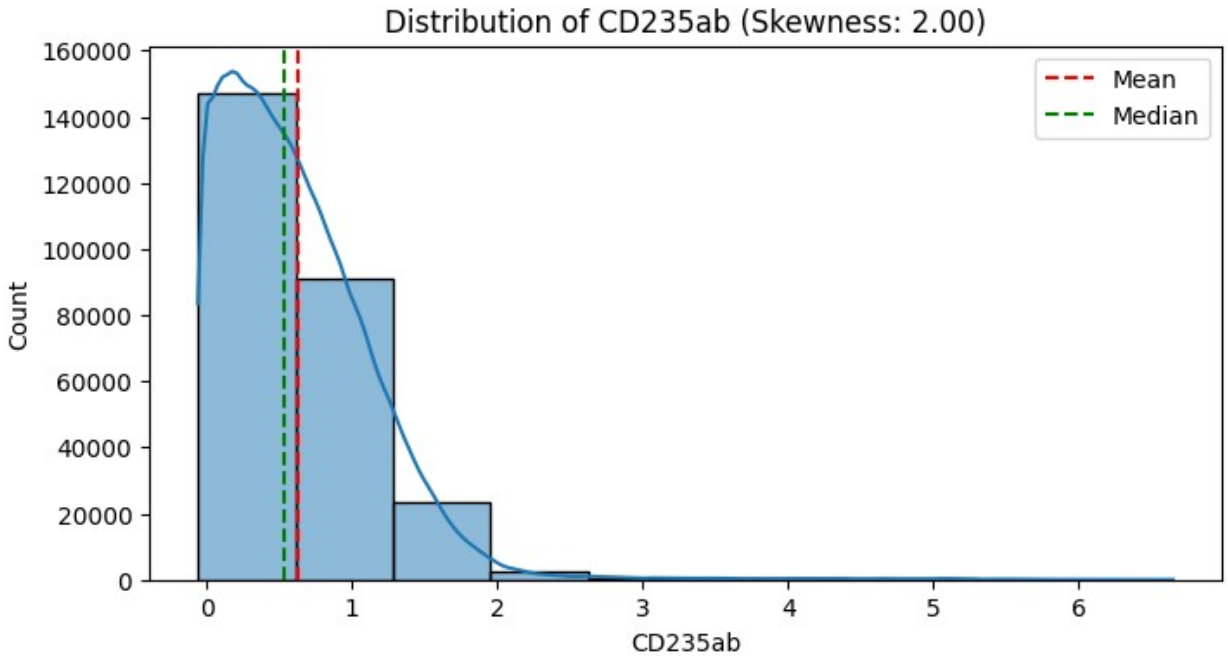

Distribution of CD15 (Skewness: 1.45)
Mean
Median
Count
CD15
0
25000
50000
75000
100000
125000
150000
175000
200000
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

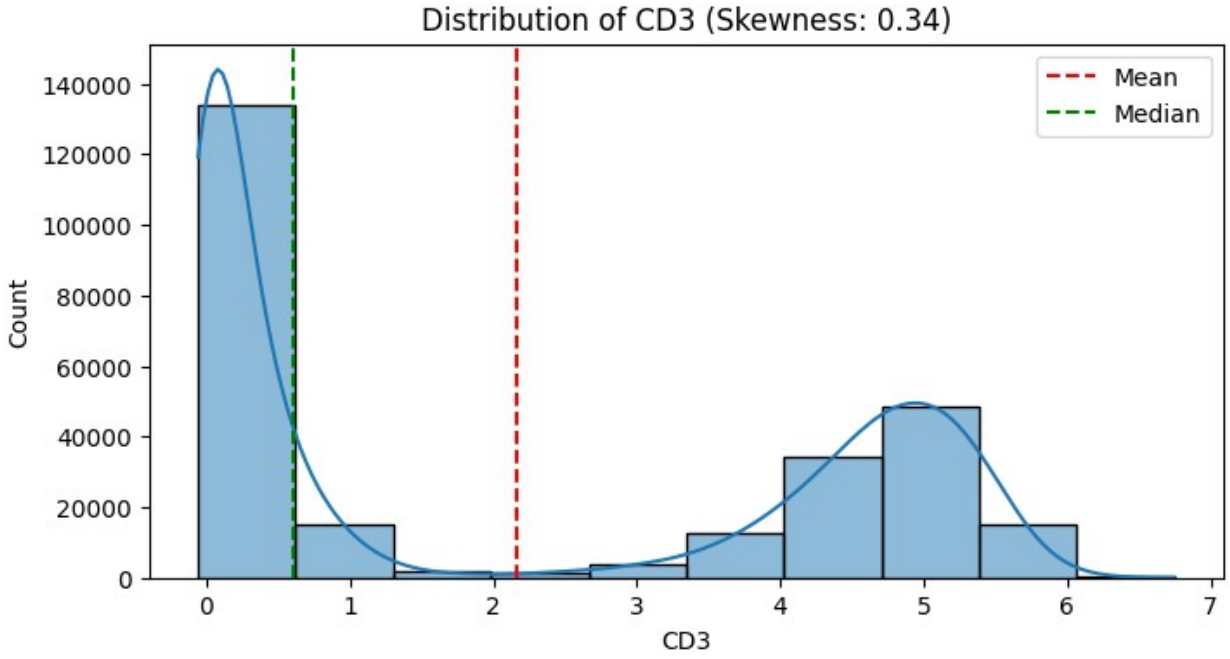

Distribution of CD16 (Skewness: 5.73)
Mean
Median
Count
CD16
0
100000
200000
300000
400000
500000
600000
700000
0
1
2
3
4
5

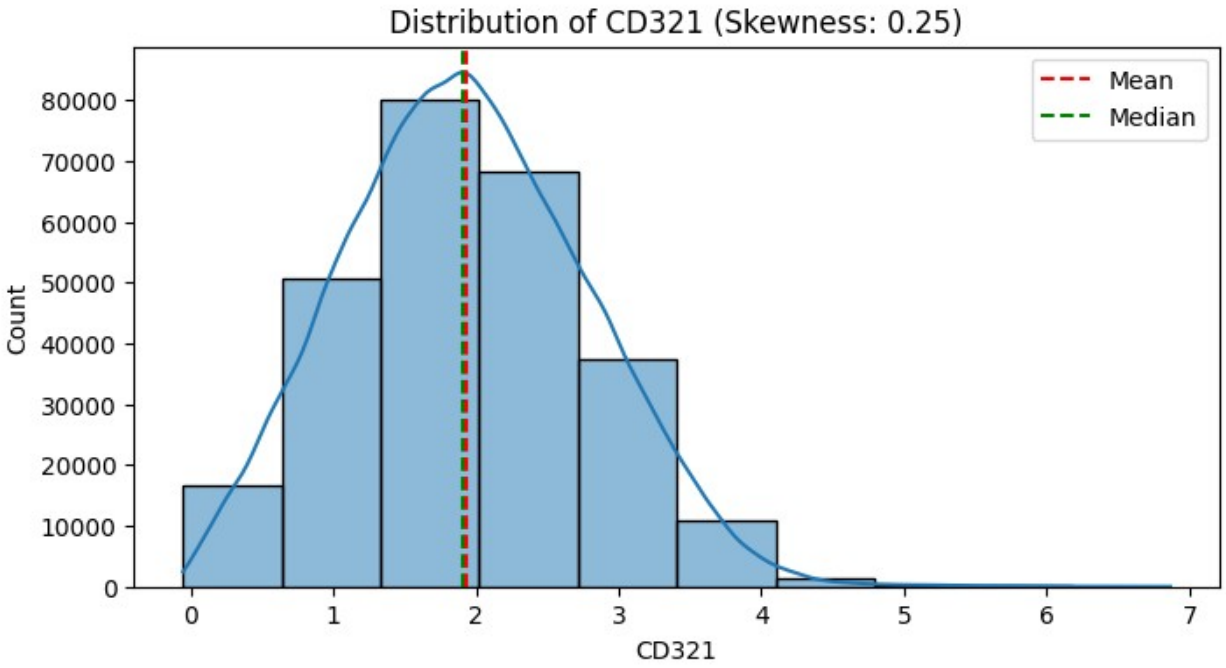

Distribution of CD19 (Skewness: 1.68)
Mean
Median
Count
CD19
0
50000
100000
150000
200000
250000
300000
350000
0
1
2
3
4
5

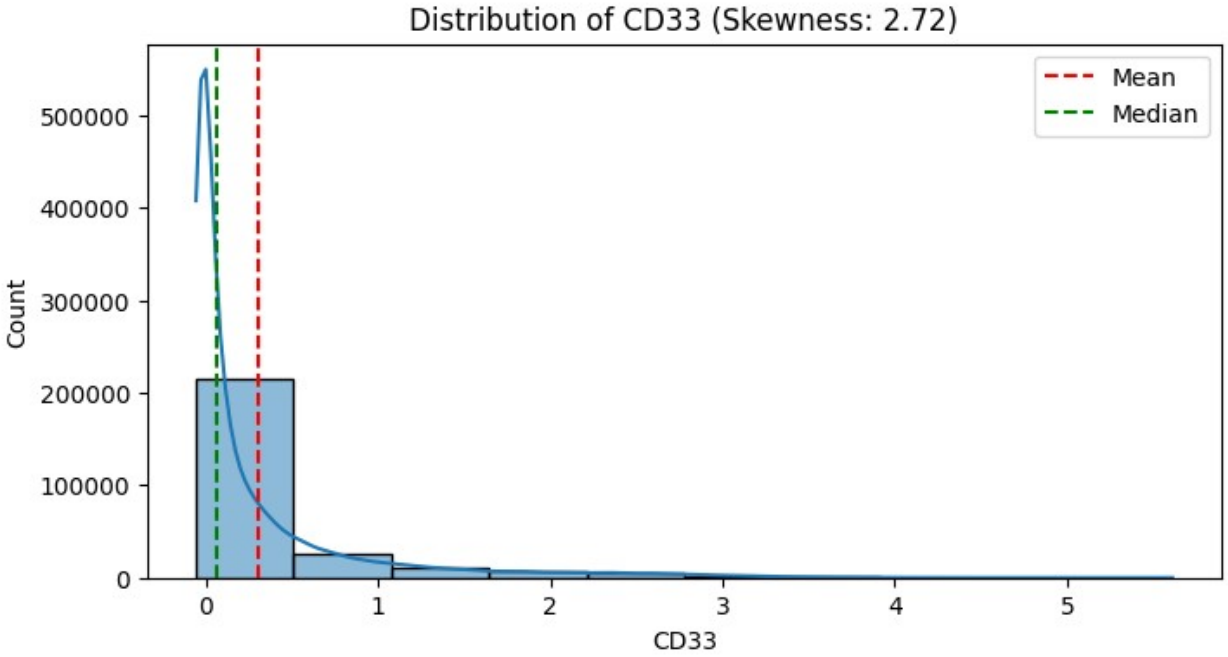

Distribution of CD20 (Skewness: 2.75)
Mean
Median
Count
CD20
0
100000
200000
300000
400000
500000
0
1
2
3
4
5
6

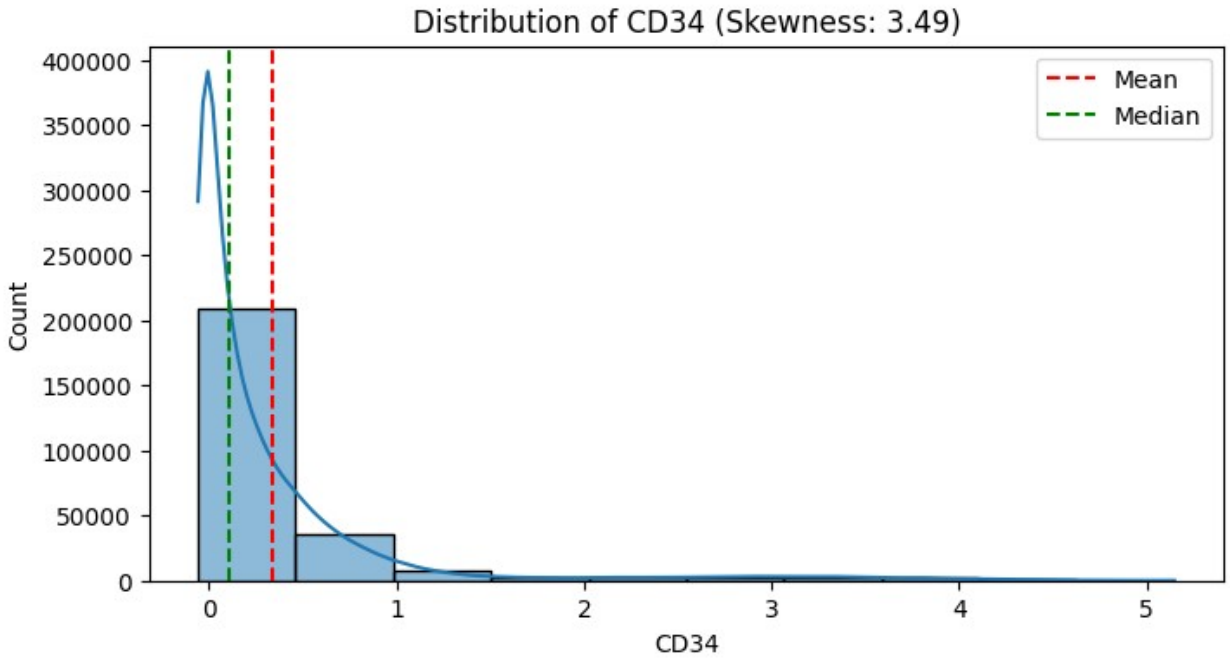

Distribution of CD22 (Skewness: 2.28)
Mean
Median
Count
CD22
0
100000
200000
300000
400000
0
1
2
3
4
5

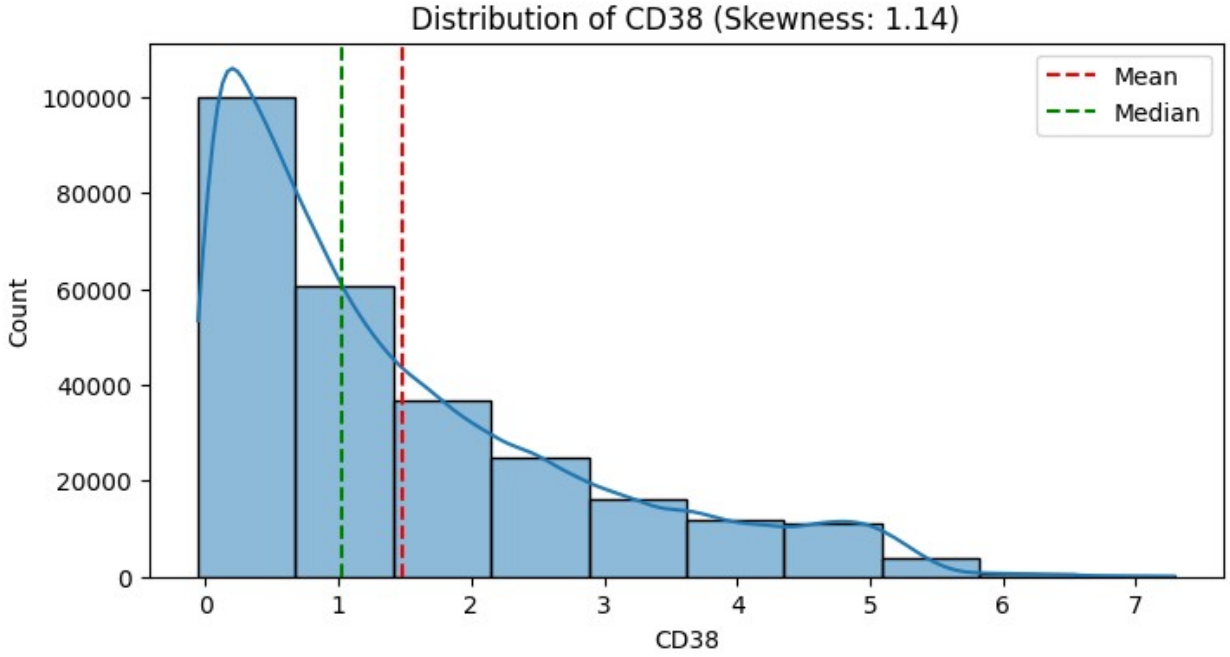

Distribution of CD235ab (Skewness: 2.00)
Mean
Median
Count
CD235ab
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6

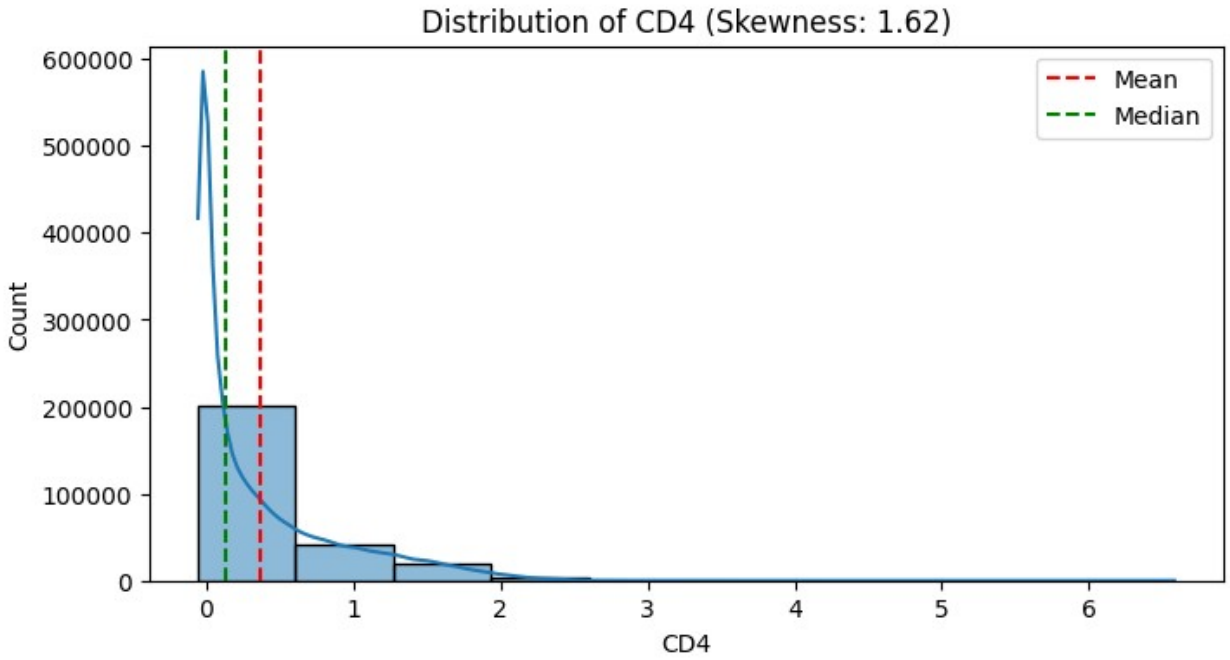

Distribution of CD3 (Skewness: 0.34)
Mean
Median
Count
CD3
0
20000
40000
60000
80000
100000
120000
140000
0
1
2
3
4
5
6
7

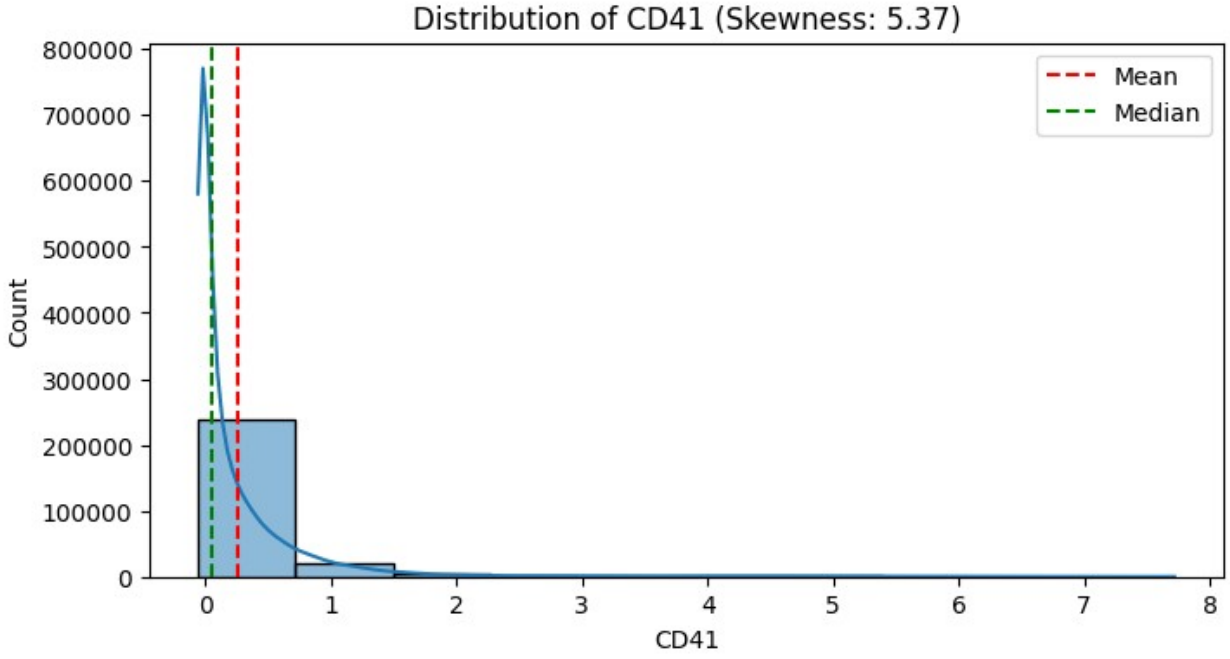

Distribution of CD321 (Skewness: 0.25)
Mean
Median
Count
CD321
0
10000
20000
30000
40000
50000
60000
70000
80000
0
1
2
3
4
5
6
7

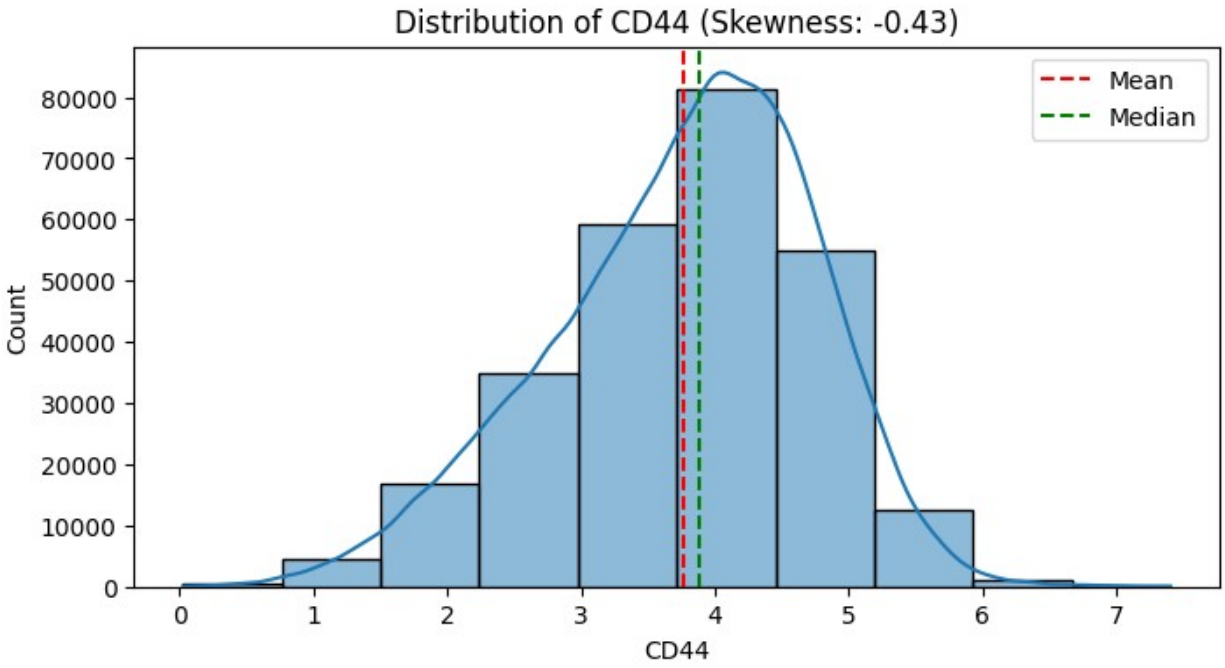

Distribution of CD33 (Skewness: 2.72)
Mean
Median
Count
CD33
0
100000
200000
300000
400000
500000
0
1
2
3
4
5

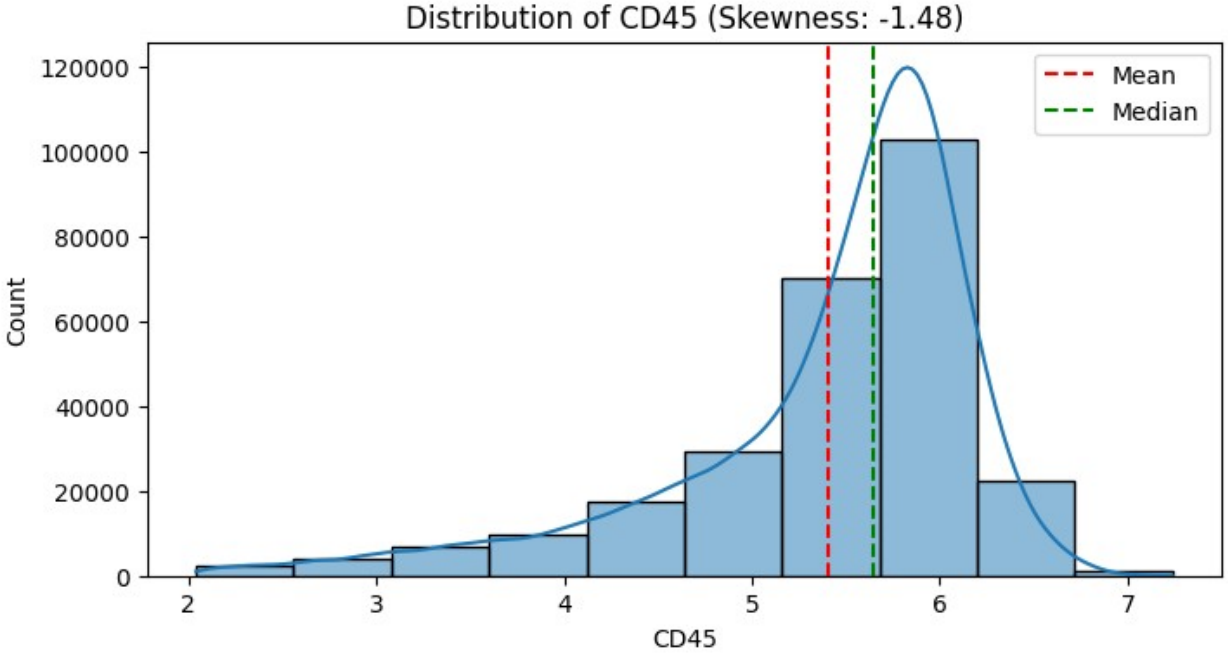

Distribution of CD34 (Skewness: 3.49)
Mean
Median
Count
CD34
400000
350000
300000
250000
200000
150000
100000
50000
0
0
1
2
3
4
5

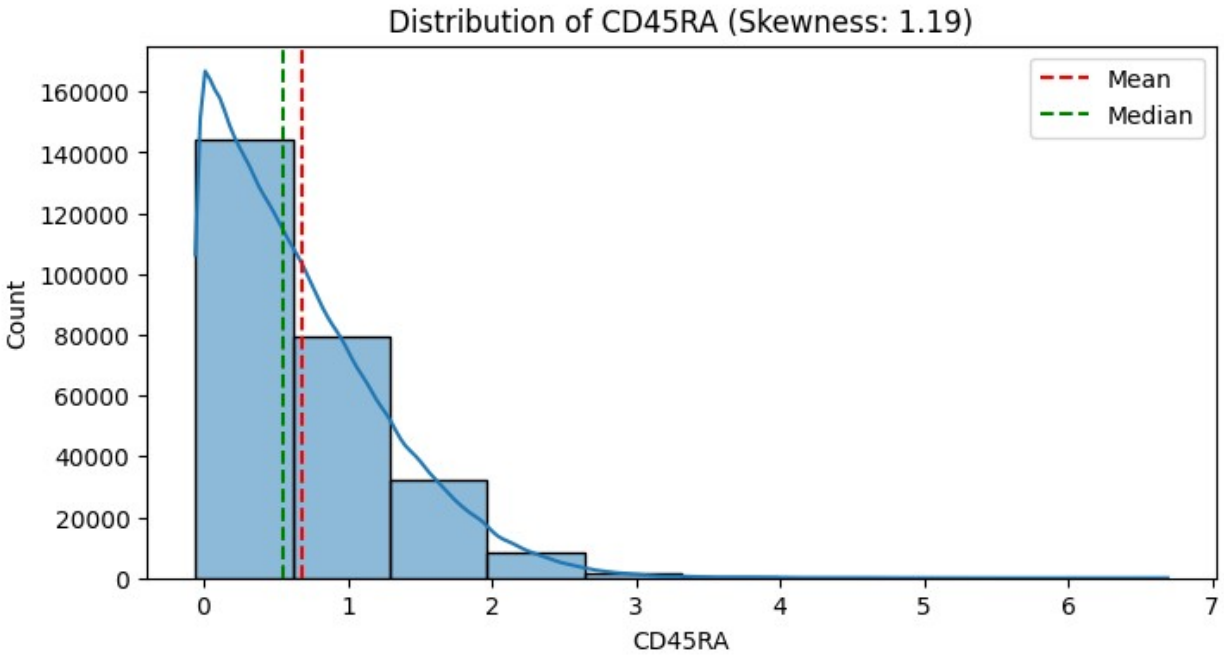

Distribution of CD38 (Skewness: 1.14)
Mean
Median
Count
CD38
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7

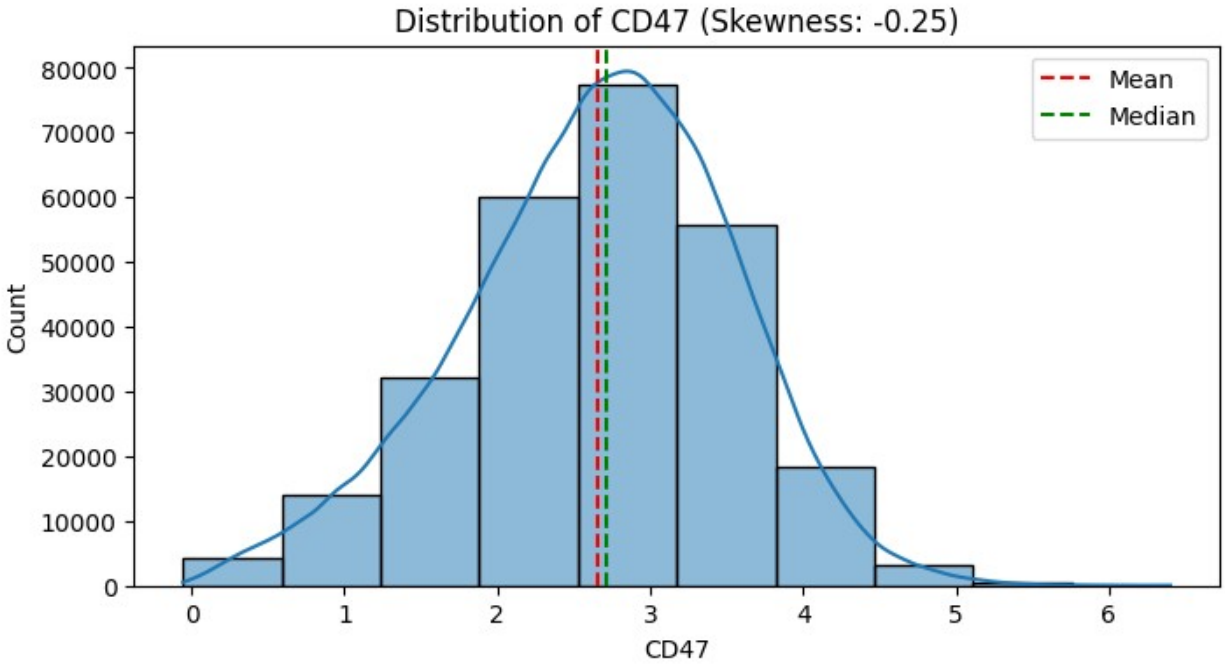

Distribution of CD4 (Skewness: 1.62)
Mean
Median
Count
CD4
600000
500000
400000
300000
200000
100000
0
0
1
2
3
4
5
6

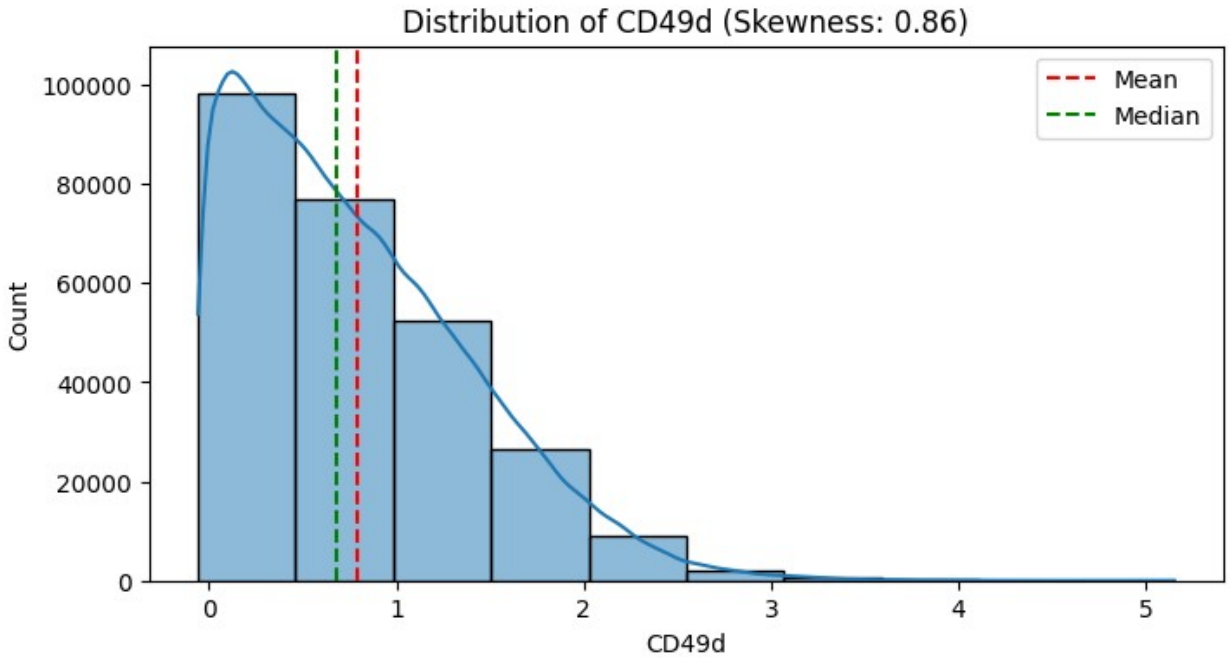

Distribution of CD41 (Skewness: 5.37)
Mean
Median
Count
CD41
800000
700000
600000
500000
400000
300000
200000
100000
0
0
1
2
3
4
5
6
7
8

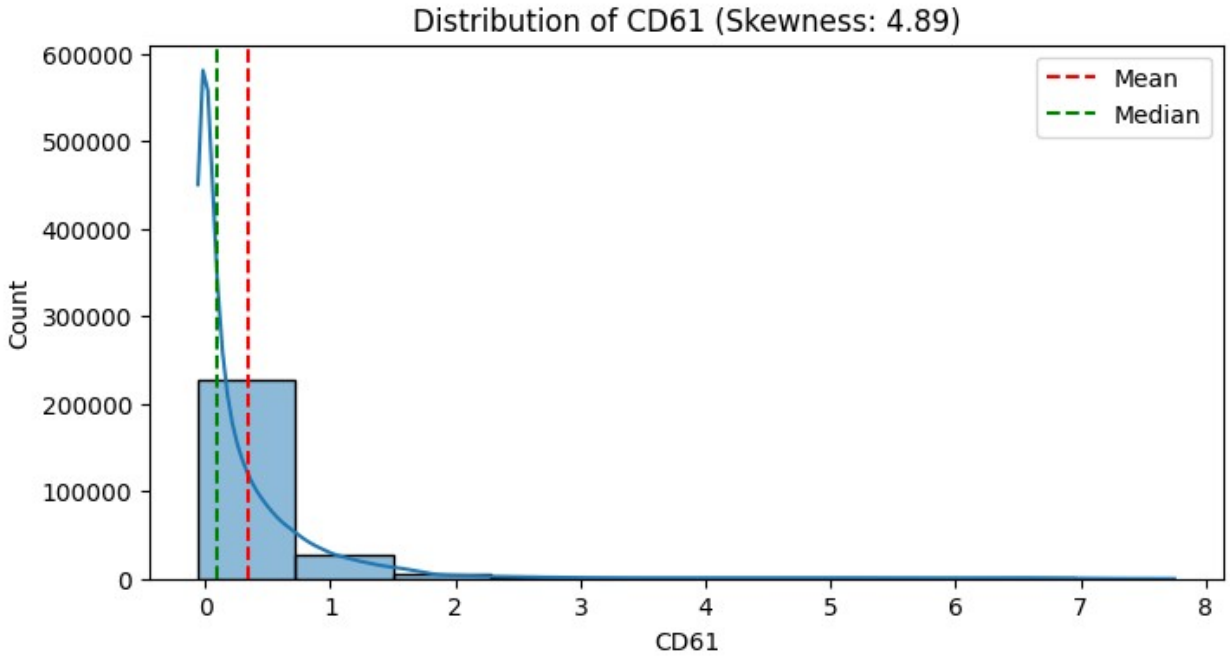

Distribution of CD44 (Skewness: -0.43)
Mean
Median
Count
CD44
0
10000
20000
30000
40000
50000
60000
70000
80000
0
1
2
3
4
5
6
7

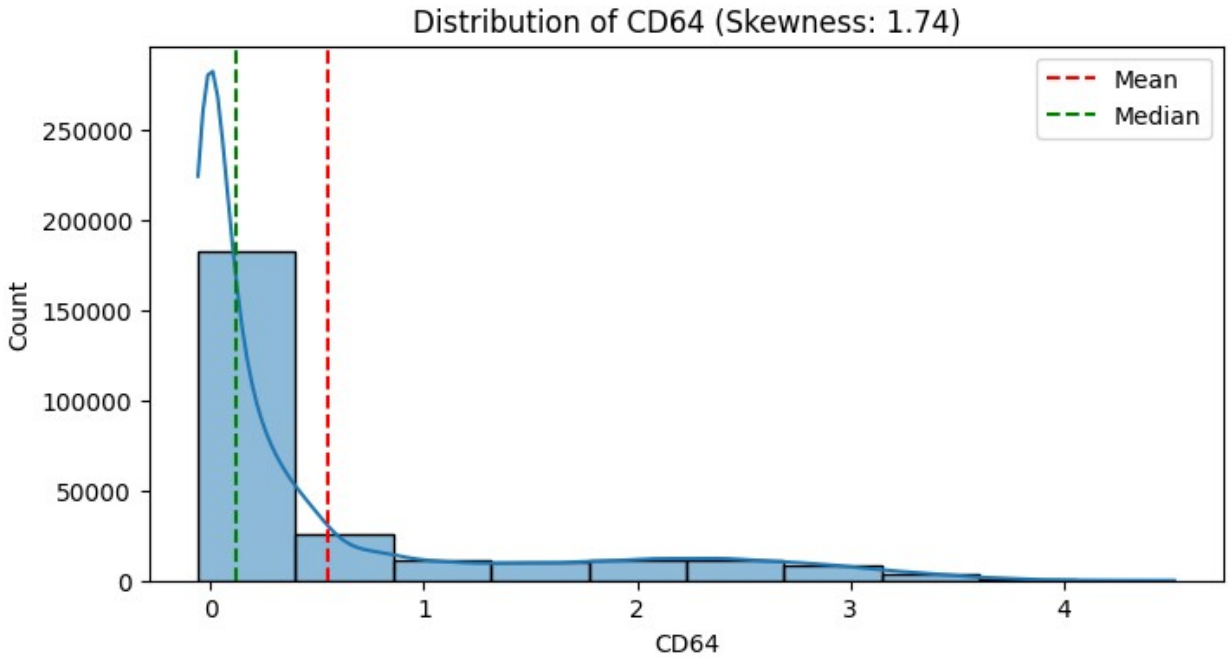

Distribution of CD45 (Skewness: -1.48)
Mean
Median
Count
CD45
0
20000
40000
60000
80000
100000
120000
2
3
4
5
6
7

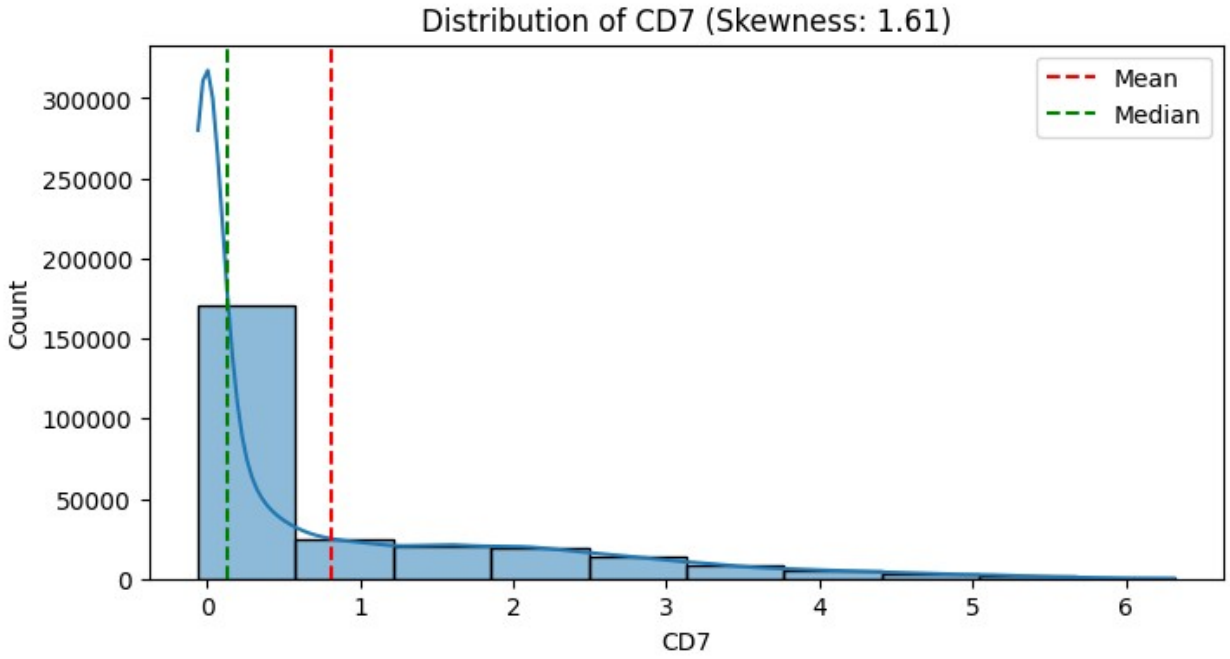

Distribution of CD45RA (Skewness: 1.19)
Mean
Median
Count
CD45RA
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6
7

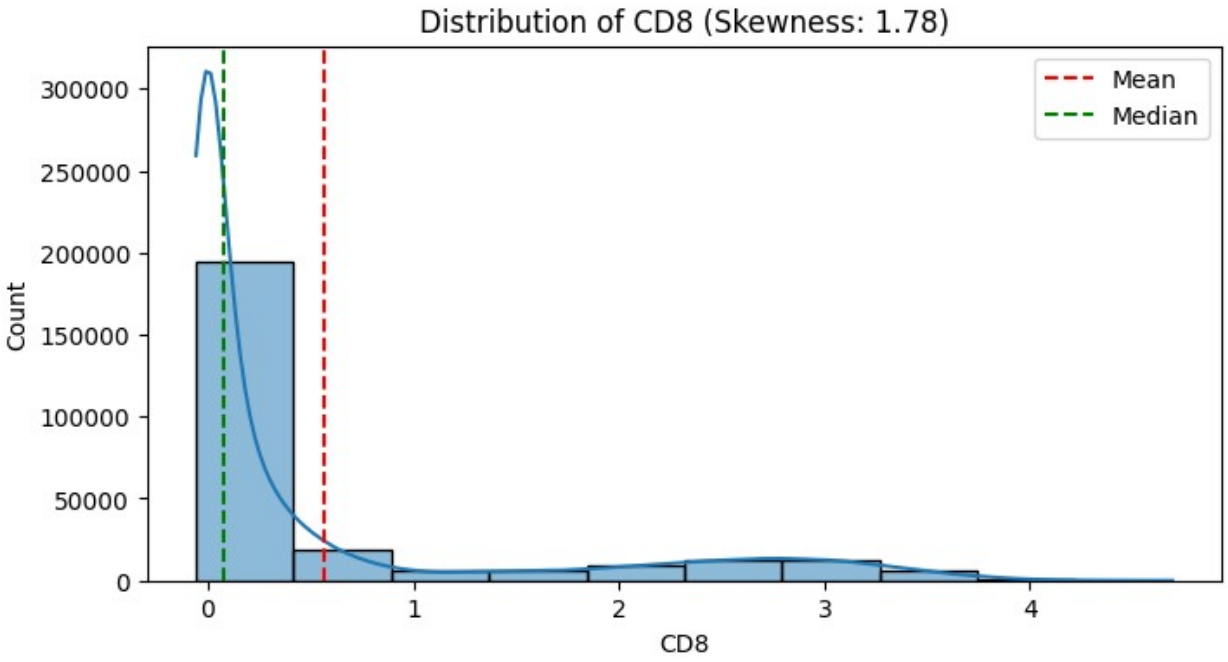

Distribution of CD47 (Skewness: -0.25)
Mean
Median
Count
CD47
0
10000
20000
30000
40000
50000
60000
70000
80000
0
1
2
3
4
5
6

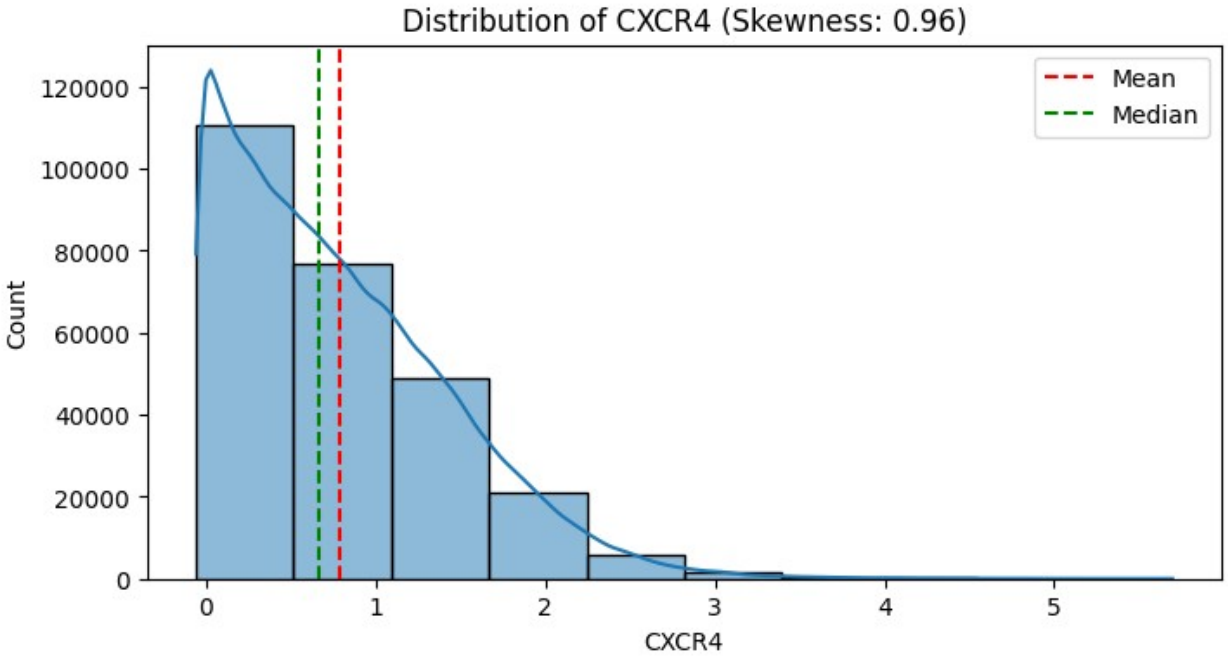

Distribution of CD49d (Skewness: 0.86)
Mean
Median
Count
CD49d
0
20000
40000
60000
80000
100000
0
1
2
3
4
5

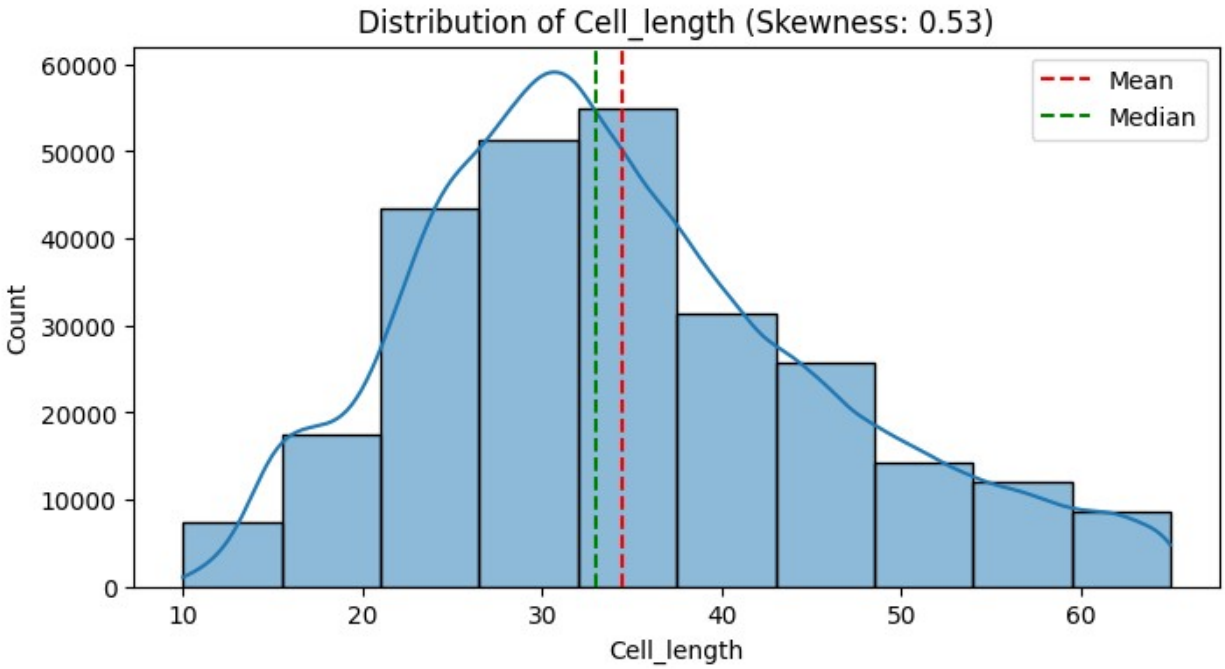

Distribution of CD61 (Skewness: 4.89)
Mean
Median
Count
CD61
0
100000
200000
300000
400000
500000
600000
0
1
2
3
4
5
6
7
8

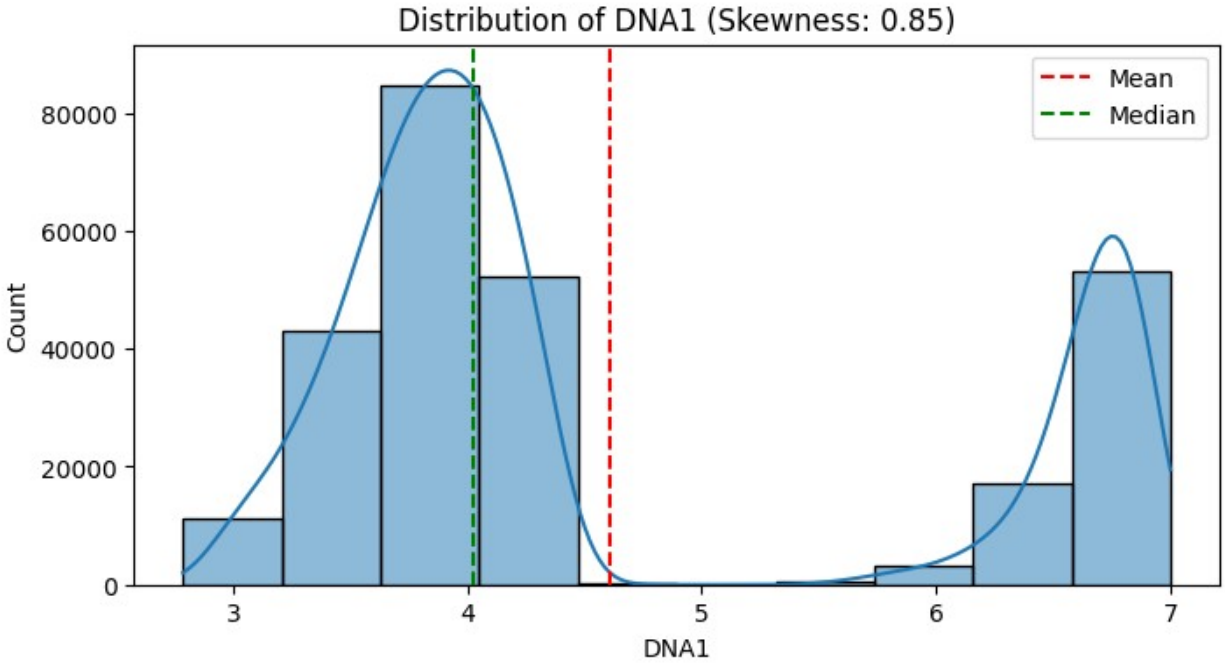

Distribution of CD64 (Skewness: 1.74)
Mean
Median
Count
250000
200000
150000
100000
50000
0
0
1
2
3
4
CD64

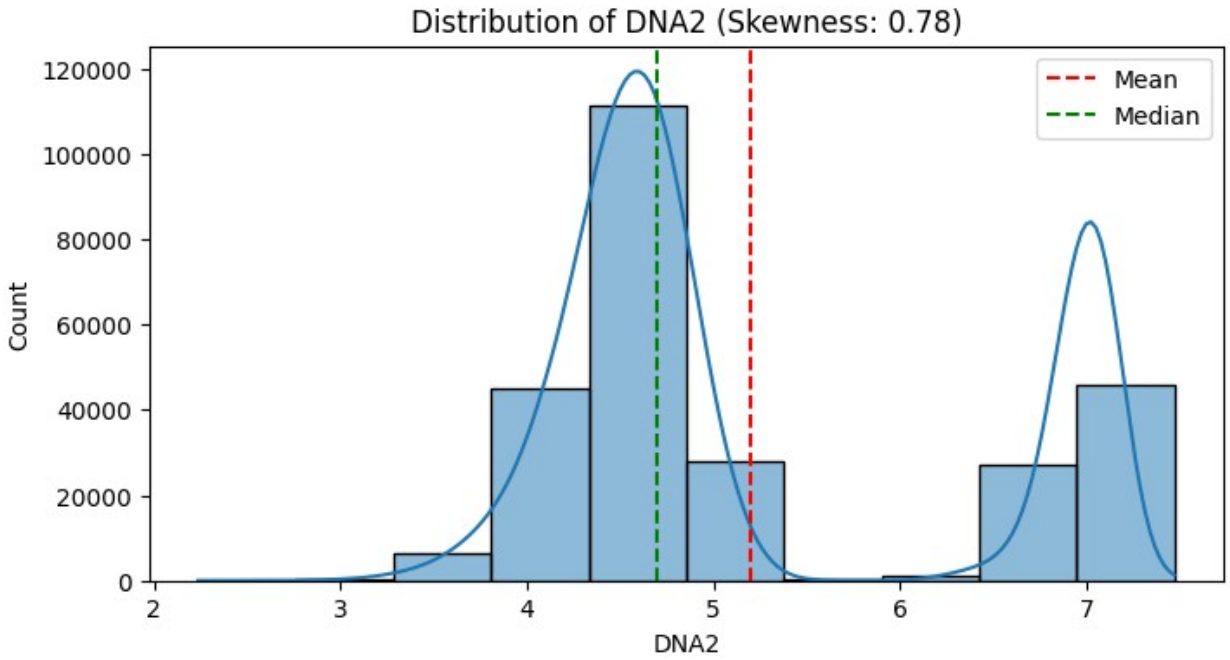

Distribution of CD7 (Skewness: 1.61)
Mean
Median
Count
300000
250000
200000
150000
100000
50000
0
0
1
2
3
4
5
6
CD7

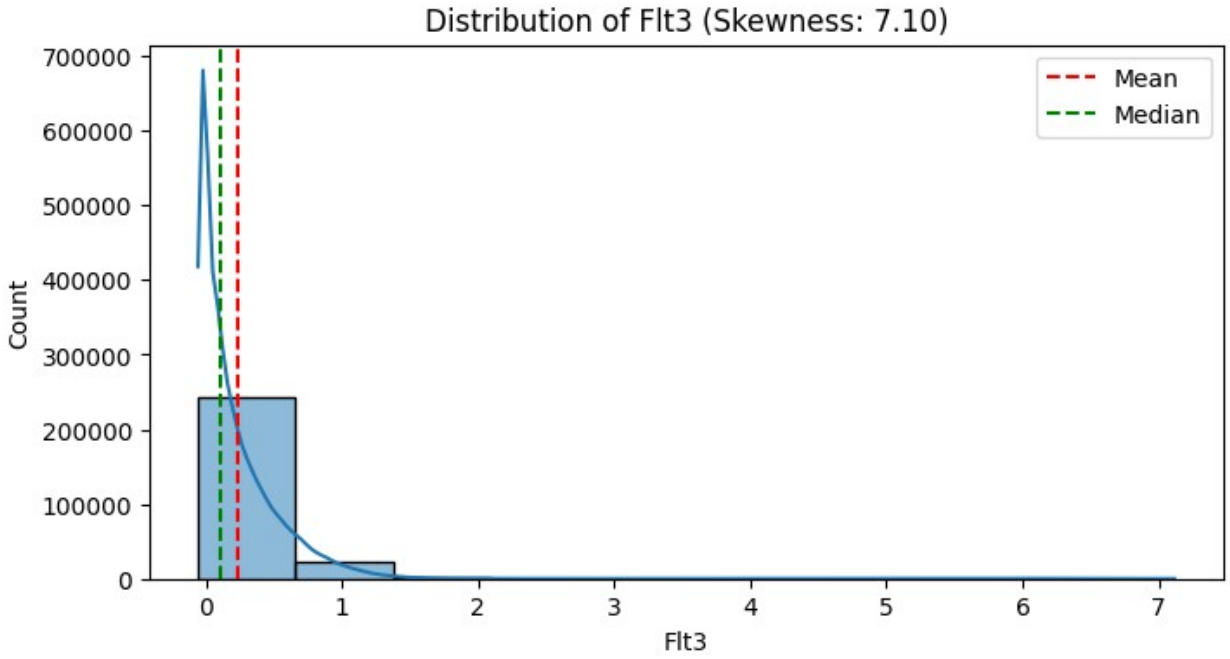

Distribution of CD8 (Skewness: 1.78)
Mean
Median
Count
CD8
300000
250000
200000
150000
100000
50000
0
0
1
2
3
4

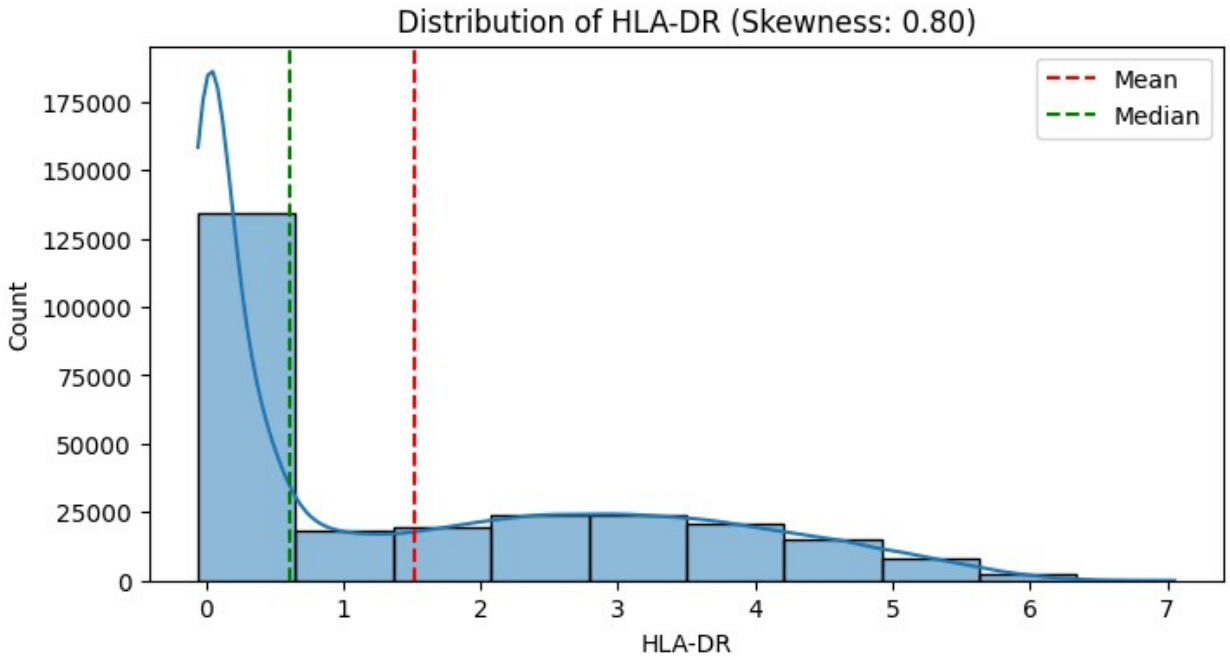

Distribution of CXCR4 (Skewness: 0.96)
Mean
Median
Count
CXCR4
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5

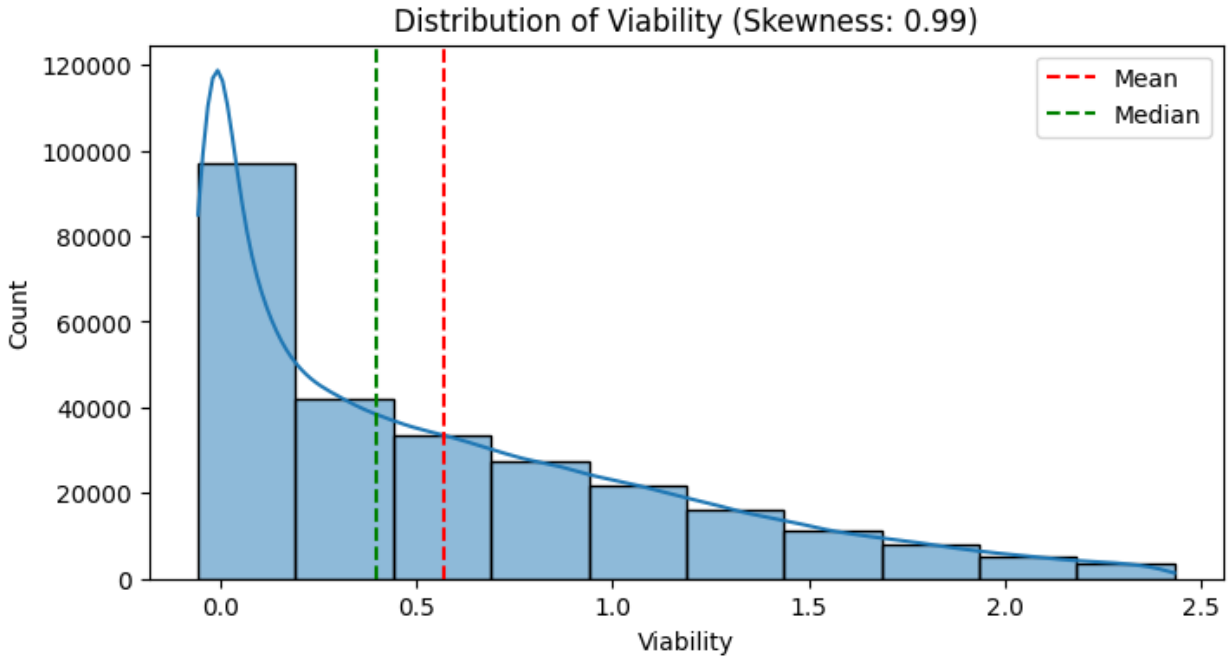

Distribution of Cell_length (Skewness: 0.53)
Mean
Median
Count
Cell_length
0
10000
20000
30000
40000
50000
60000
10
20
30
40
50
60

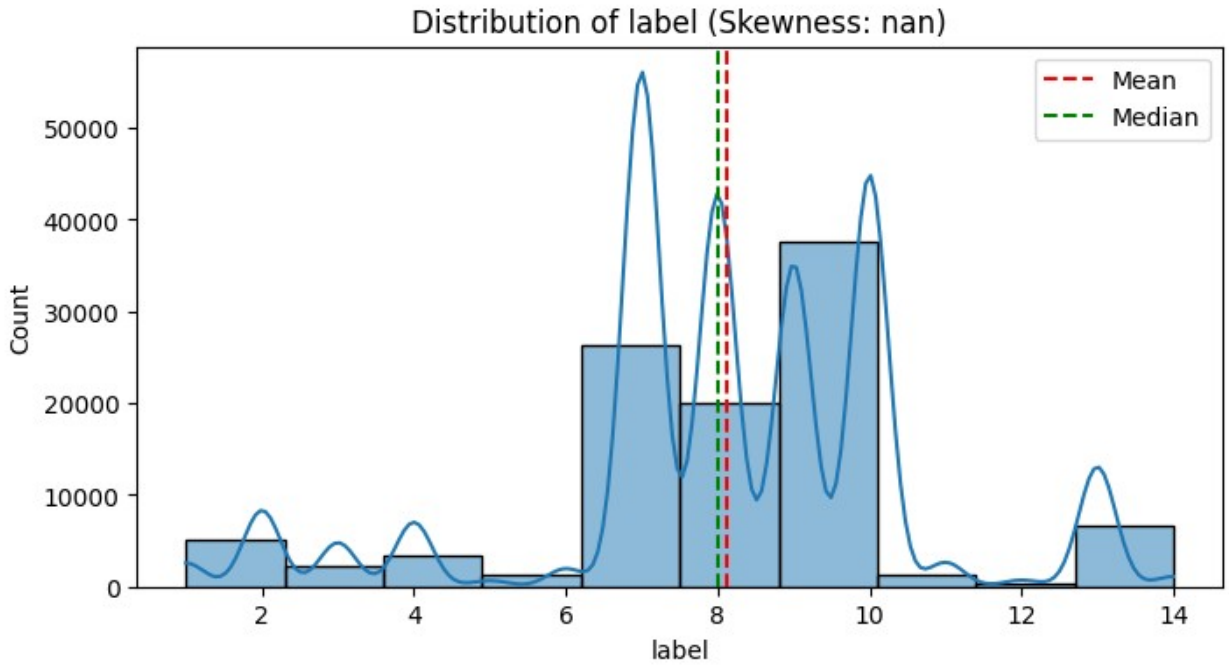

Distribution of DNA1 (Skewness: 0.85)
Mean
Median
Count
DNA1
0
20000
40000
60000
80000
3
4
5
6
7


Distribution of DNA2 (Skewness: 0.78)
Mean
Median
Count
DNA2
0
20000
40000
60000
80000
100000
120000
2
3
4
5
6
7


Distribution of Flt3 (Skewness: 7.10)
Mean
Median
Count
Flt3
0
100000
200000
300000
400000
500000
600000
700000
0
1
2
3
4
5
6
7


Distribution of HLA-DR (Skewness: 0.80)
Mean
Median
Count
HLA-DR
0
25000
50000
75000
100000
125000
150000
175000
0
1
2
3
4
5
6
7


Distribution of Viability (Skewness: 0.99)
Mean
Median
Count
Viability
0
20000
40000
60000
80000
100000
120000
0.0
0.5
1.0
1.5
2.0
2.5



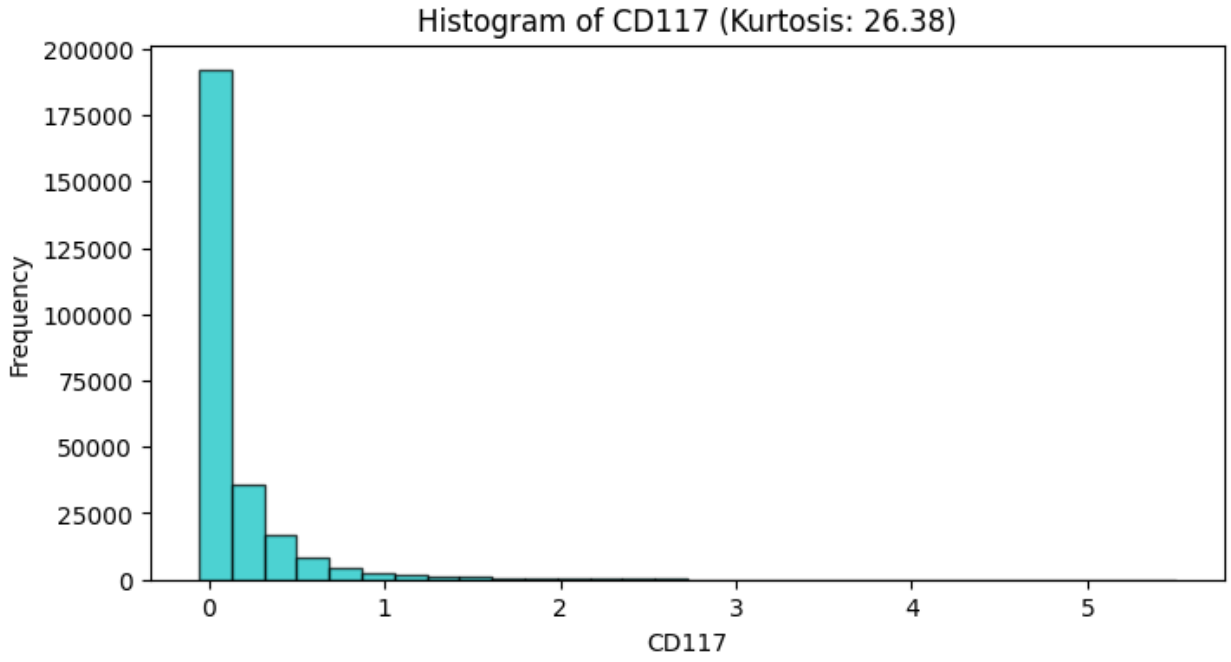

```
print('Kurtosis')
from scipy.stats import kurtosis
import matplotlib.pyplot as plt
import pandas as pd

# Exclude specific columns from the kurtosis calculation
columns_to_exclude = ['Event', 'Time', 'file_number', 'event_number',
'individual']
numerical_columns =
df.select_dtypes(include=['number']).columns.difference(columns_to_exc
lude)

# Calculate kurtosis for the specified numerical columns
kurtosis_values = df[numerical_columns].apply(kurtosis, fisher=False)
# Fisher=False gives Pearson kurtosis (normal kurtosis = 3)

# Create a DataFrame with kurtosis values
kurtosis_df = pd.DataFrame({'Column': numerical_columns, 'Kurtosis':
kurtosis_values})

# Categorize the kurtosis values (Leptokurtic, Mesokurtic,
Platykurtic)
def categorize_kurtosis(value):
    if value > 3:
        return 'Leptokurtic (heavy tails)'
    elif value < 3:
        return 'Platykurtic (light tails)'
    else:
        return 'Mesokurtic (normal tails)'
```

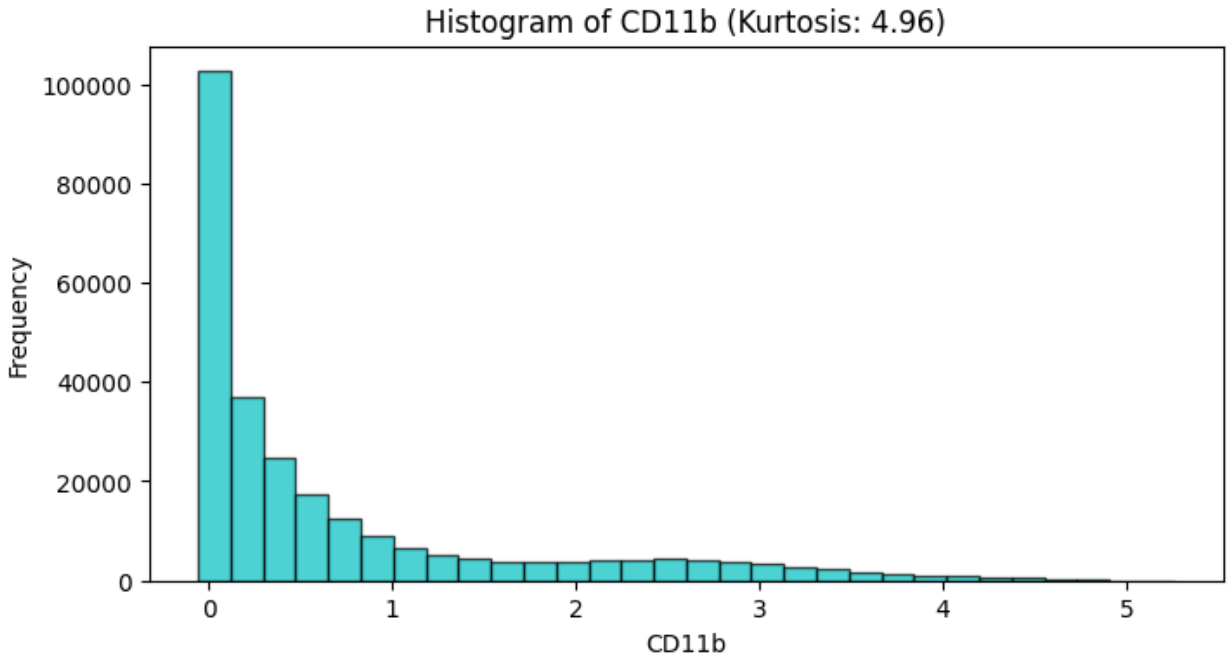
```
kurtosis_df['Category'] =
kurtosis_df['Kurtosis'].apply(categorize_kurtosis)

# Print the kurtosis values and their categories
print(kurtosis_df)

# Plot histogram for each numerical column, excluding specified
columns
for column in numerical_columns:
    plt.figure(figsize=(8, 4))
    plt.hist(df[column].dropna(), bins=30, color='c',
edgecolor='black', alpha=0.7)
    plt.title(f'Histogram of {column} (Kurtosis:
{kurtosis_df.loc[kurtosis_df["Column"] == column,
"Kurtosis"].values[0]:.2f})')
    plt.xlabel(column)
    plt.ylabel('Frequency')
    plt.grid(False)  # Remove grid lines
    plt.show()
```

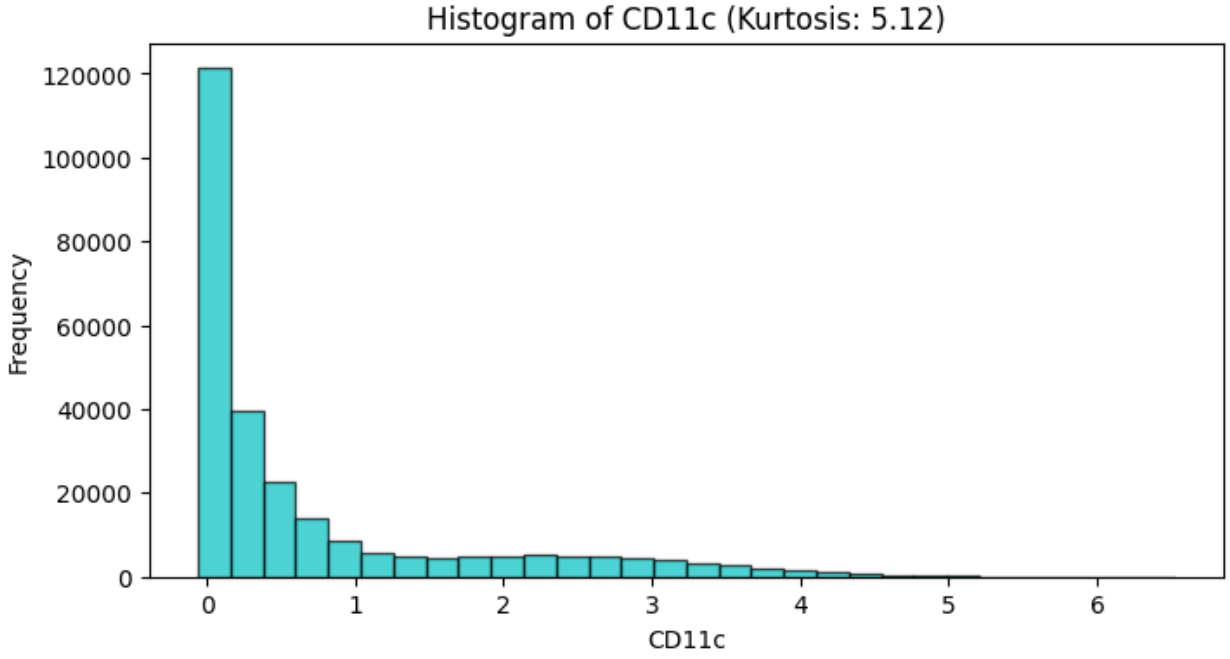
Kurtosis

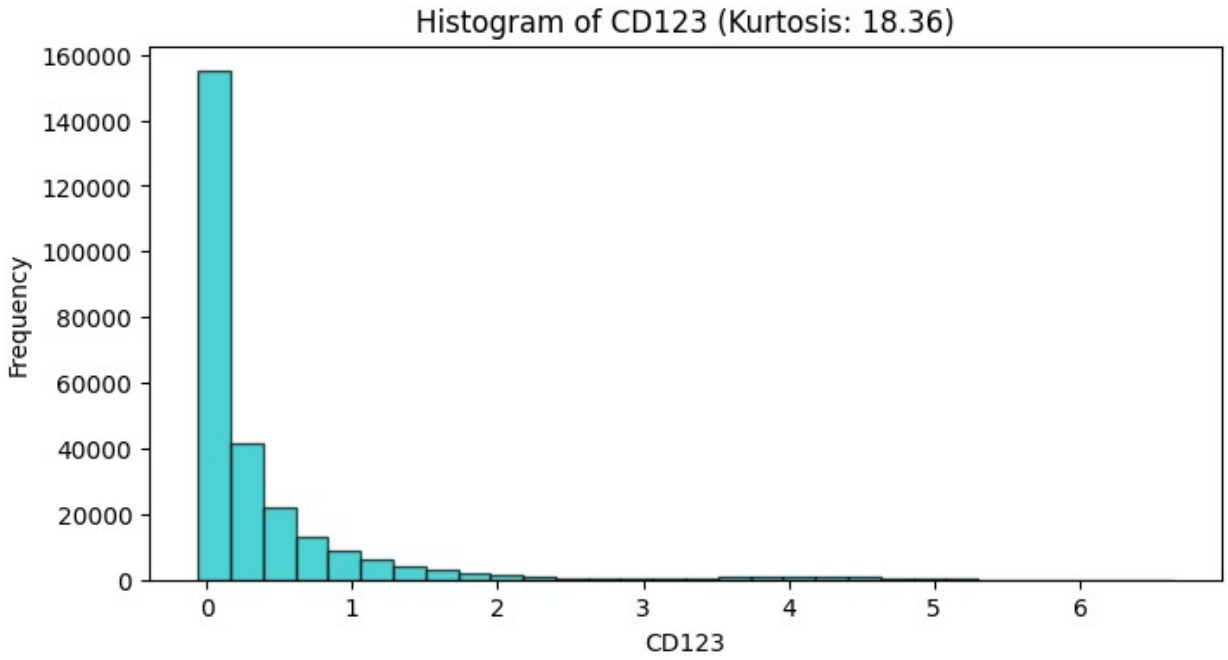
	Column	Kurtosis	Category
CD117	CD117	26.375108	Leptokurtic (heavy tails)
CD11b	CD11b	4.964495	Leptokurtic (heavy tails)
CD11c	CD11c	5.117156	Leptokurtic (heavy tails)
CD123	CD123	18.361217	Leptokurtic (heavy tails)
CD13	CD13	10.637564	Leptokurtic (heavy tails)
CD133	CD133	9.190066	Leptokurtic (heavy tails)
CD14	CD14	23.062535	Leptokurtic (heavy tails)
CD15	CD15	4.504387	Leptokurtic (heavy tails)
CD16	CD16	42.287749	Leptokurtic (heavy tails)
CD19	CD19	4.590887	Leptokurtic (heavy tails)
CD20	CD20	10.435449	Leptokurtic (heavy tails)
CD22	CD22	7.500223	Leptokurtic (heavy tails)
CD235ab	CD235ab	13.440586	Leptokurtic (heavy tails)
CD3	CD3	1.264612	Platykurtic (light tails)
CD321	CD321	2.914593	Platykurtic (light tails)
CD33	CD33	10.967536	Leptokurtic (heavy tails)
CD34	CD34	16.596416	Leptokurtic (heavy tails)
CD38	CD38	3.521190	Leptokurtic (heavy tails)
CD4	CD4	5.844261	Leptokurtic (heavy tails)
CD41	CD41	41.521113	Leptokurtic (heavy tails)
CD44	CD44	2.918792	Platykurtic (light tails)
CD45	CD45	5.246770	Leptokurtic (heavy tails)
CD45RA	CD45RA	4.964272	Leptokurtic (heavy tails)
CD47	CD47	2.943834	Platykurtic (light tails)
CD49d	CD49d	3.468119	Leptokurtic (heavy tails)
CD61	CD61	34.878020	Leptokurtic (heavy tails)
CD64	CD64	4.910631	Leptokurtic (heavy tails)

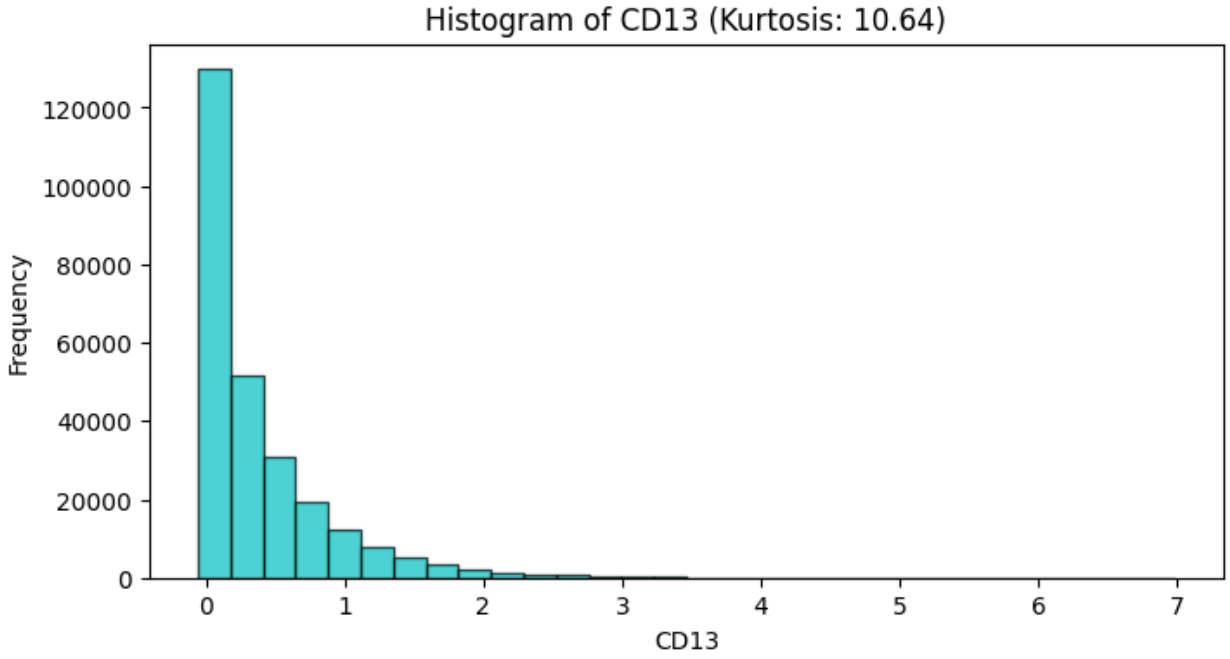
```
CD7                   CD7   4.885115  Leptokurtic (heavy tails)
CD8                   CD8   4.745776  Leptokurtic (heavy tails)
CXCR4               CXCR4   3.936307  Leptokurtic (heavy tails)
Cell_length  Cell_length   2.834033  Platykurtic (light tails)
DNA1                 DNA1   1.994037  Platykurtic (light tails)
DNA2                 DNA2   1.975021  Platykurtic (light tails)
Flt3                 Flt3  85.583534  Leptokurtic (heavy tails)
HLA-DR            HLA-DR   2.309924  Platykurtic (light tails)
Viability      Viability   3.156935  Leptokurtic (heavy tails)
label              label        NaN  Mesokurtic (normal tails)
```

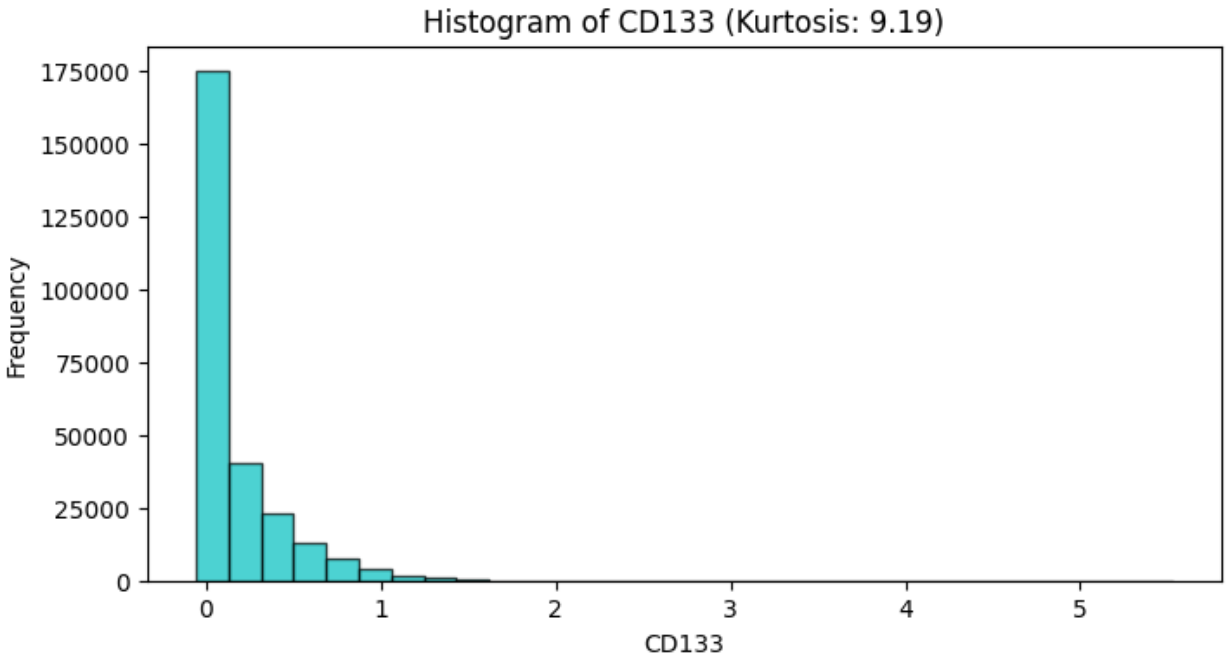


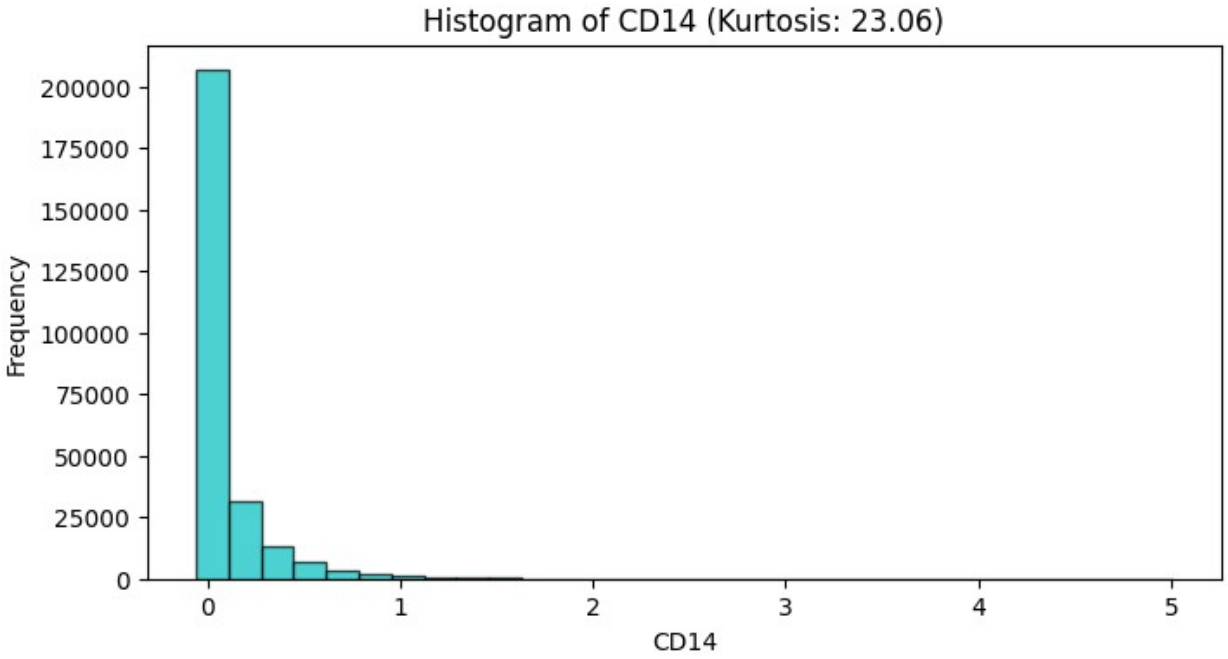

Histogram of CD11b (Kurtosis: 4.96)
Frequency
CD11b
0
20000
40000
60000
80000
100000
0
1
2
3
4
5

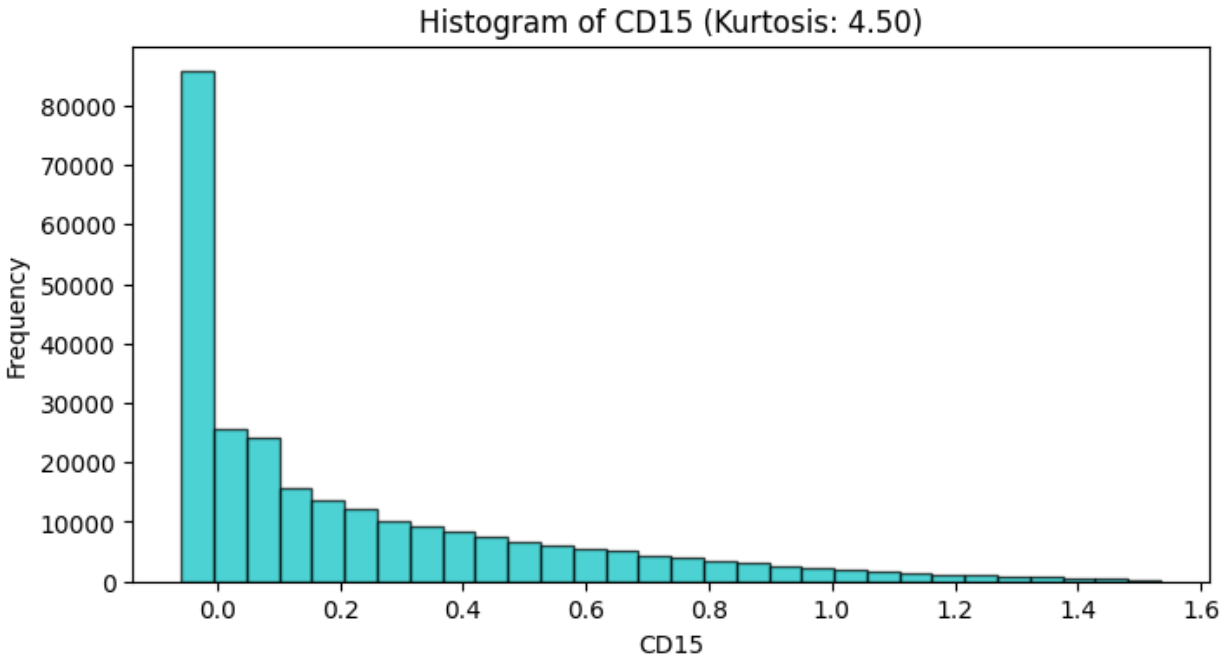

Histogram of CD11c (Kurtosis: 5.12)
Frequency
CD11c
0
20000
40000
60000
80000
100000
120000
0
1
2
3
4
5
6

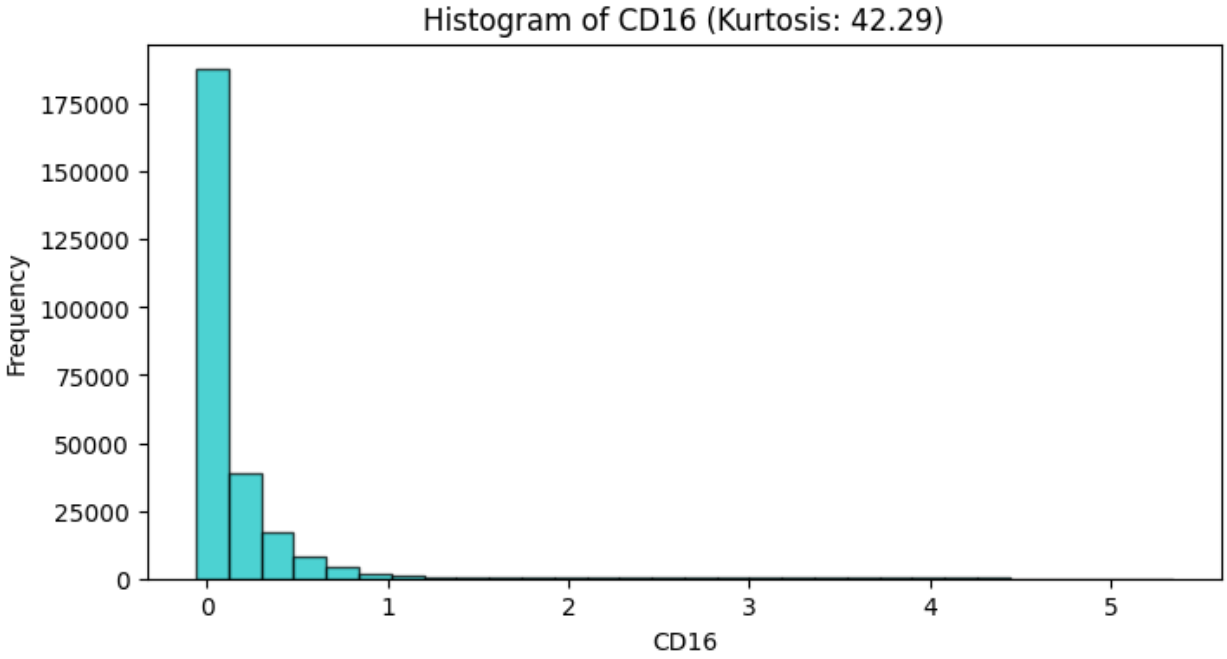

Histogram of CD123 (Kurtosis: 18.36)
Frequency
CD123
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6

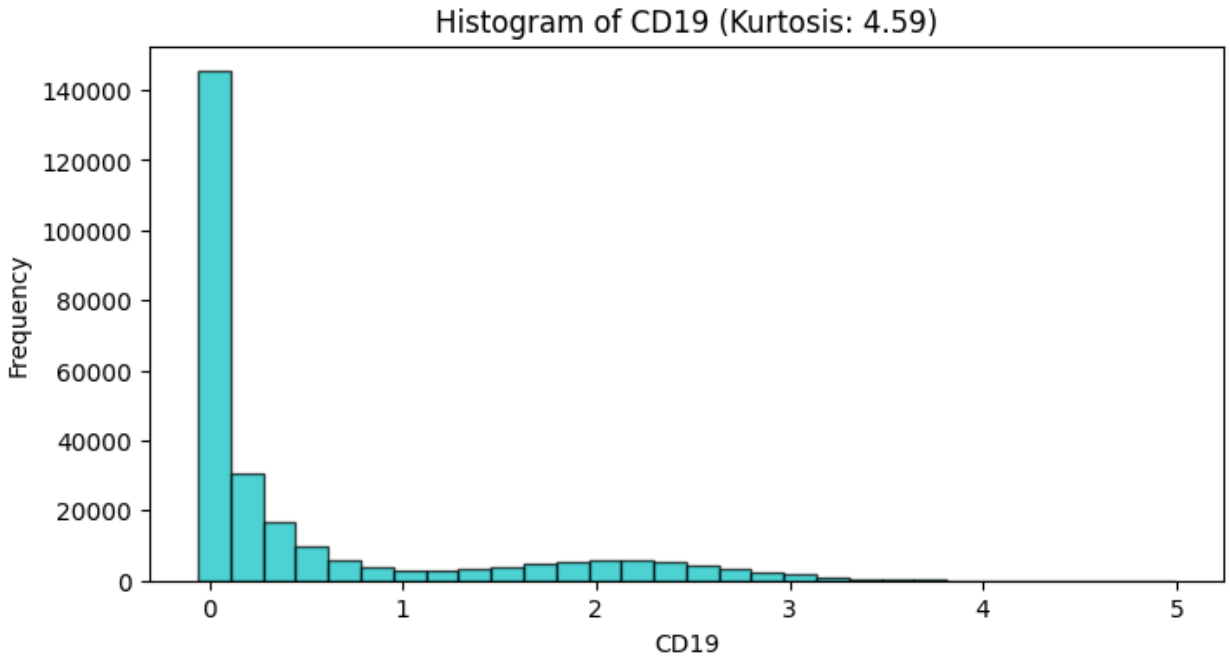

Histogram of CD13 (Kurtosis: 10.64)
Frequency
CD13
0
20000
40000
60000
80000
100000
120000
0
1
2
3
4
5
6
7

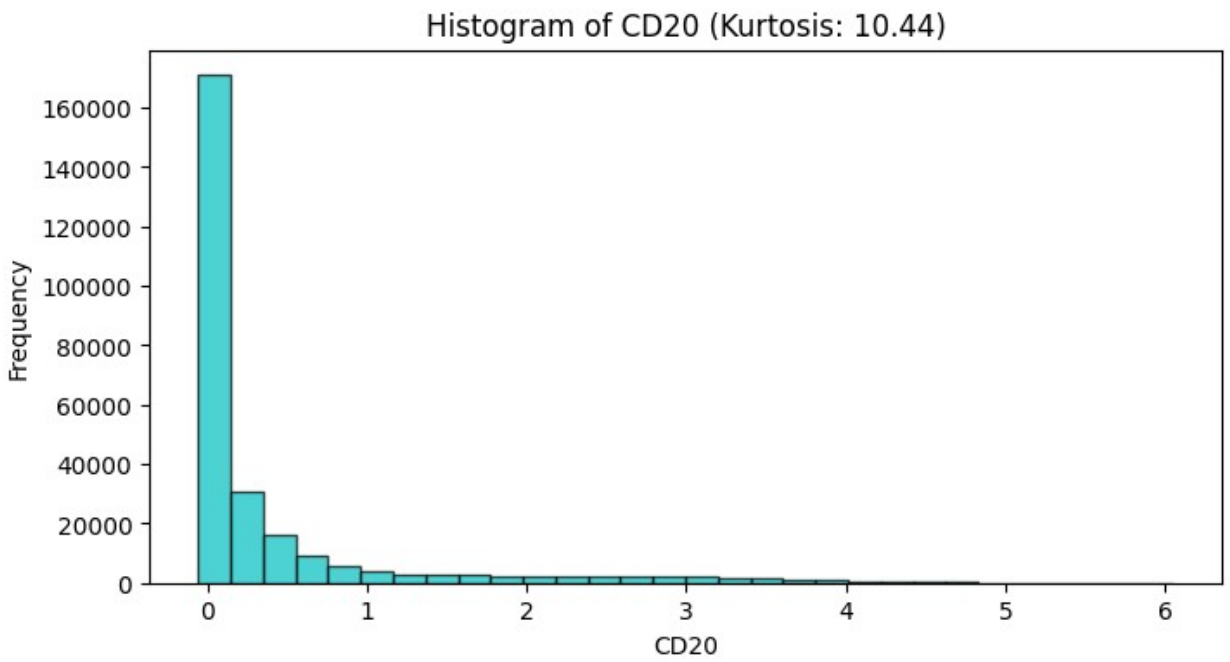

Histogram of CD133 (Kurtosis: 9.19)
Frequency
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5
CD133

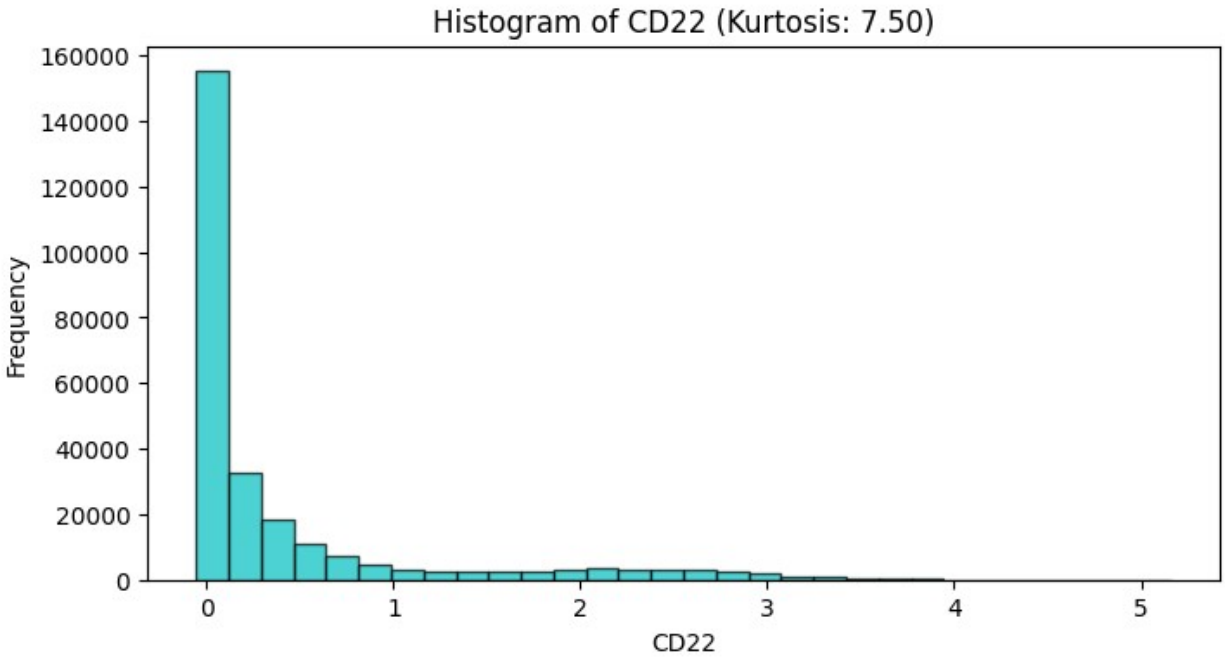

Histogram of CD14 (Kurtosis: 23.06)
Frequency
200000
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5
CD14

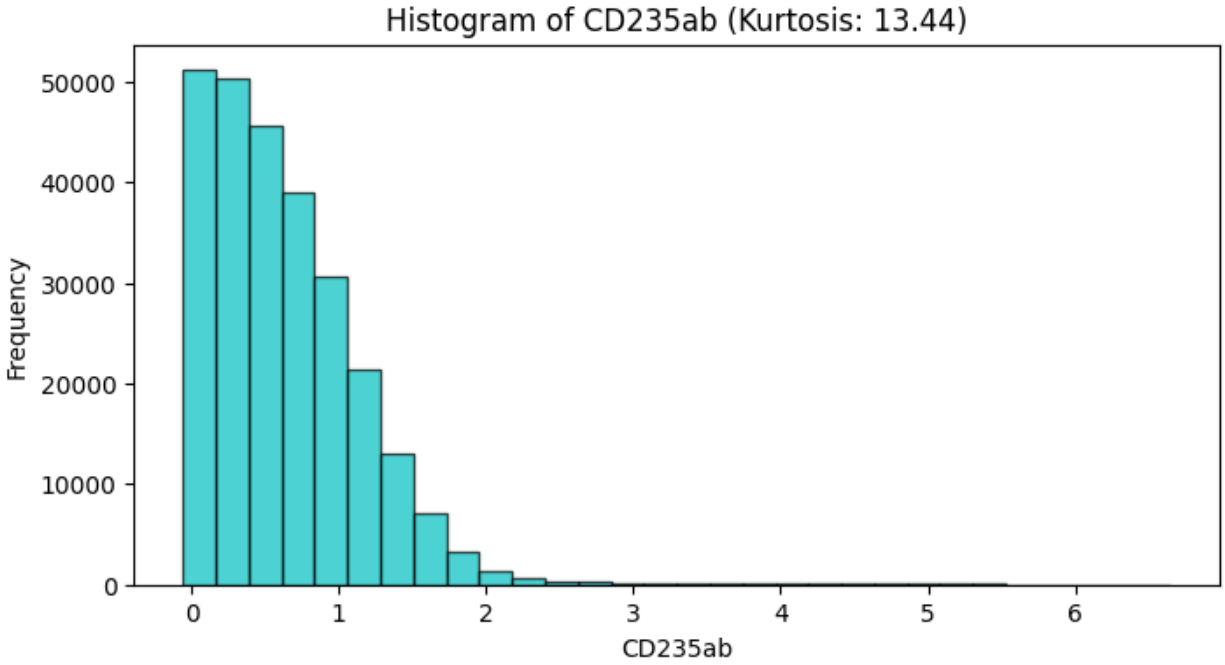

Histogram of CD15 (Kurtosis: 4.50)
Frequency
80000
70000
60000
50000
40000
30000
20000
10000
0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
CD15

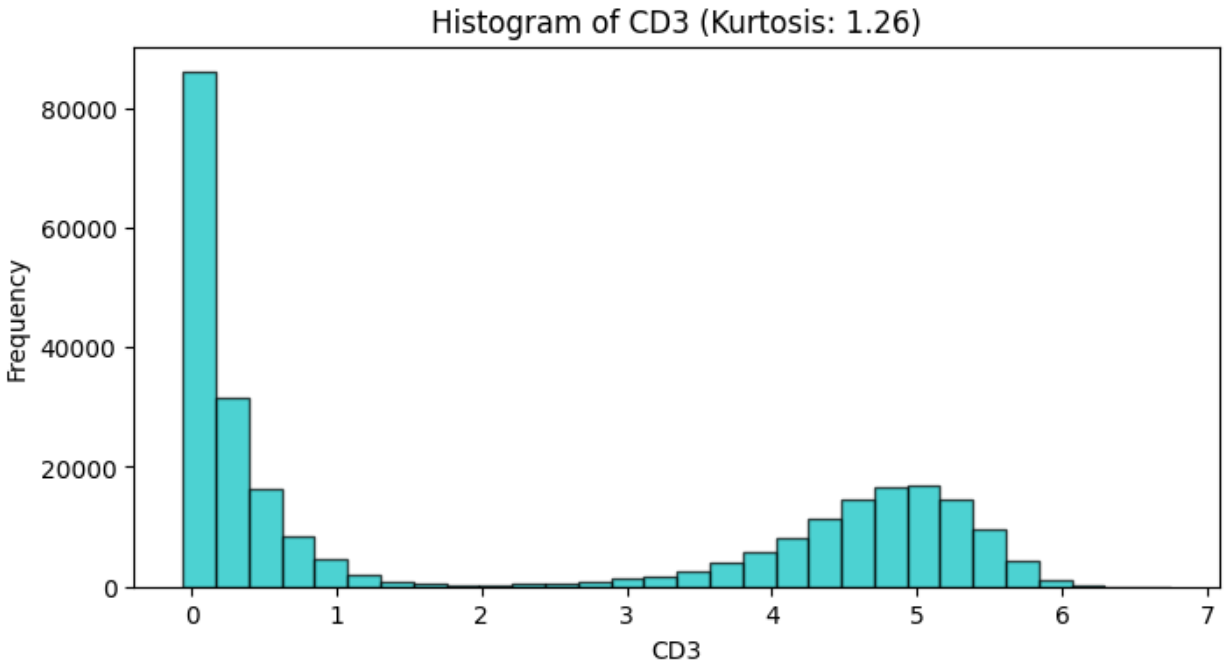

Histogram of CD16 (Kurtosis: 42.29)
Frequency
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5
CD16

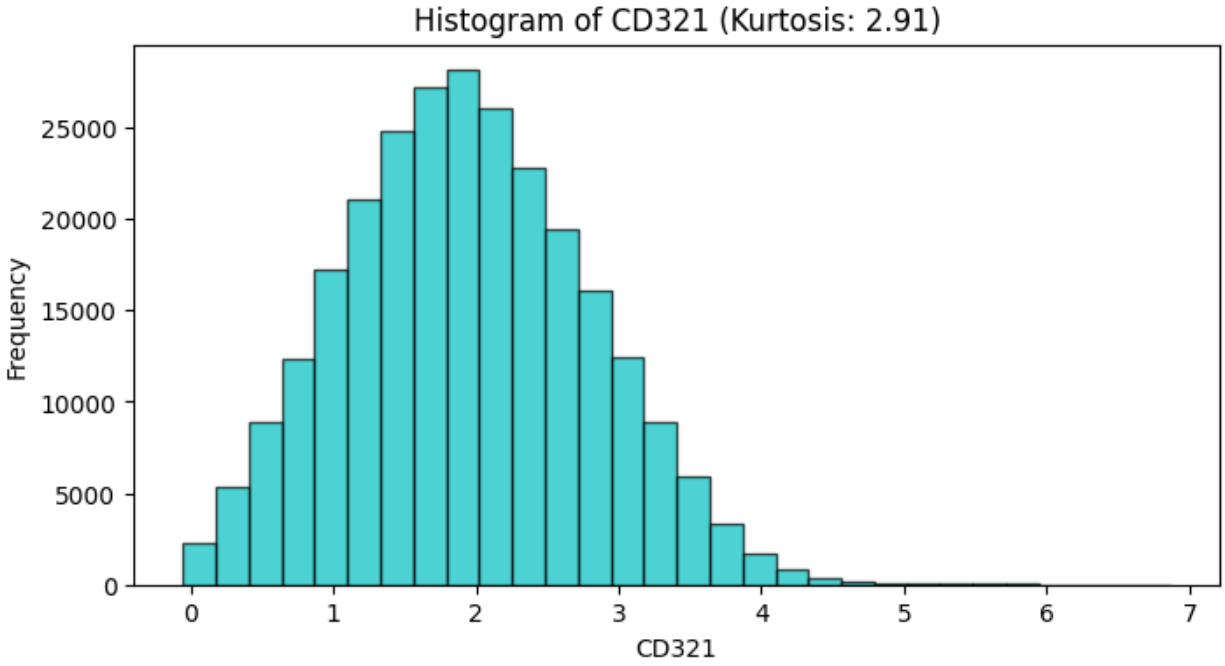

Histogram of CD19 (Kurtosis: 4.59)
Frequency
CD19
0
20000
40000
60000
80000
100000
120000
140000
0
1
2
3
4
5

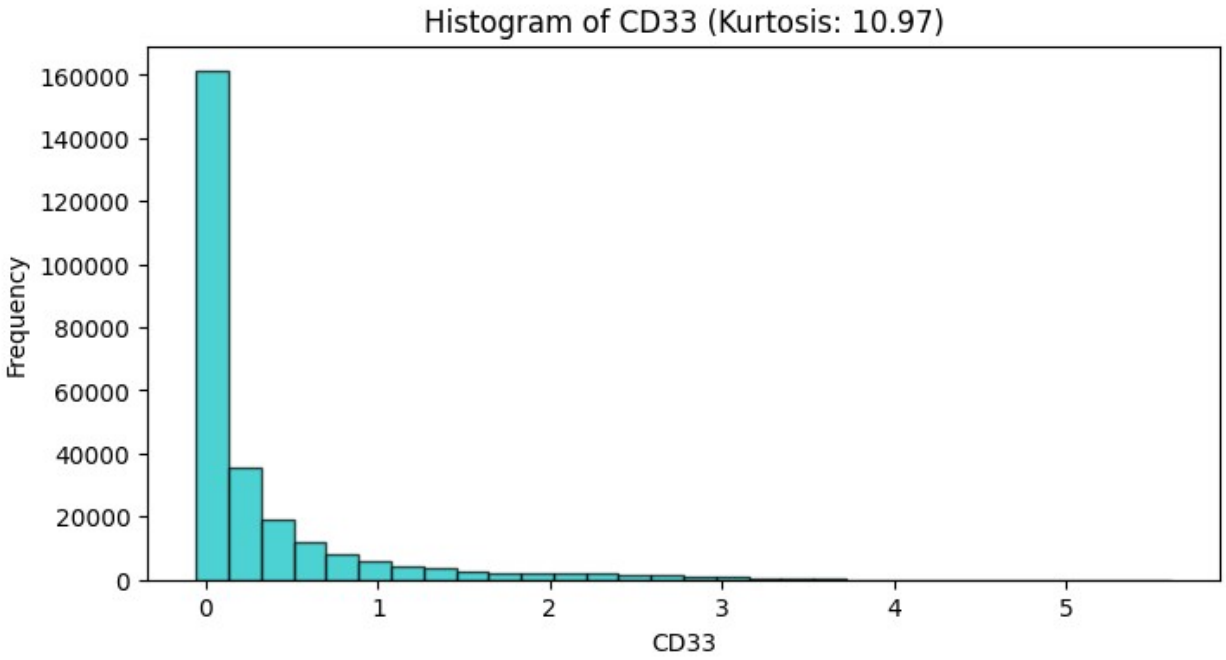

Histogram of CD20 (Kurtosis: 10.44)
Frequency
CD20
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6

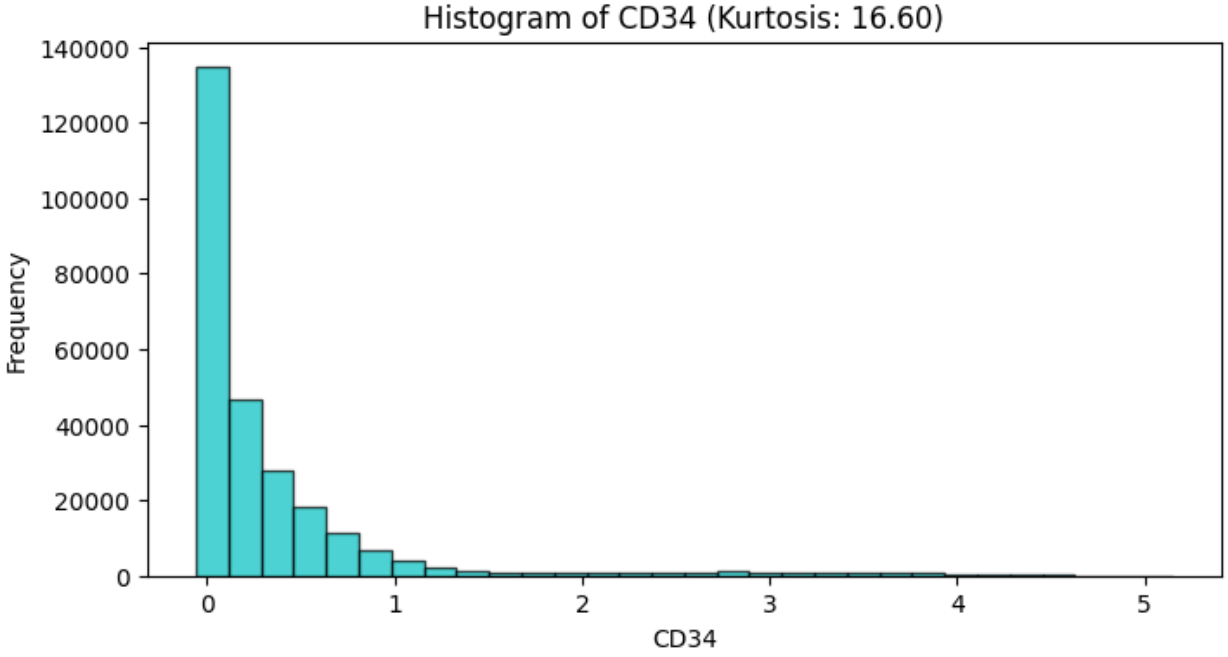

Histogram of CD22 (Kurtosis: 7.50)
Frequency
CD22
160000
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5

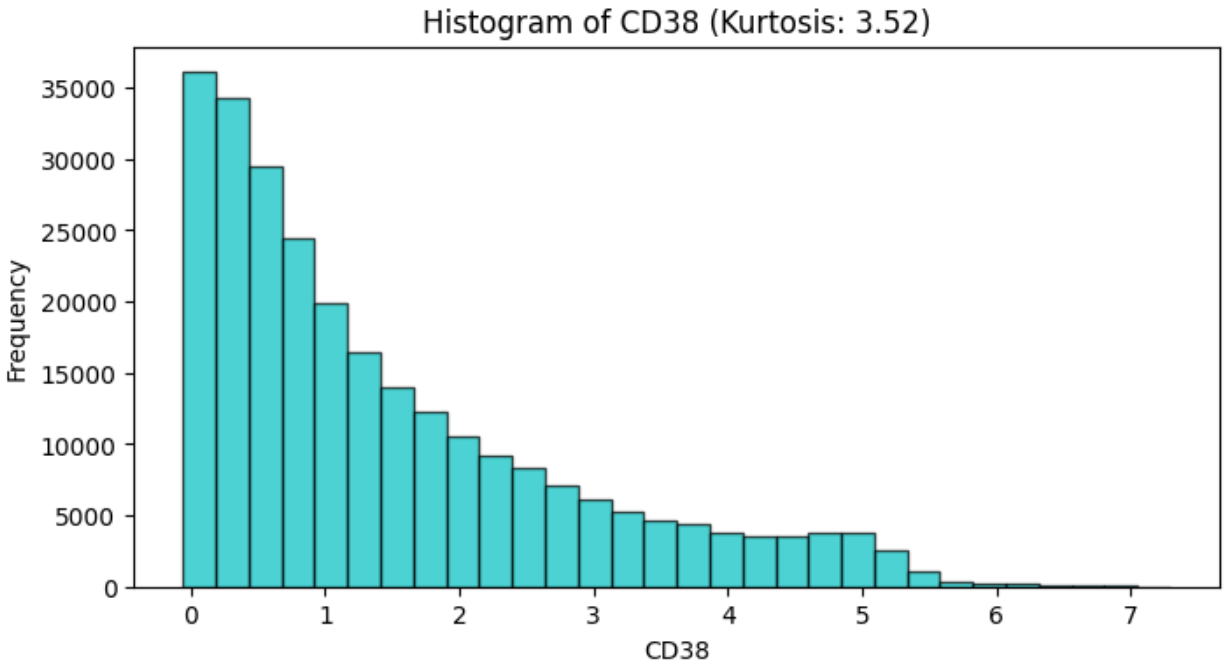

Histogram of CD235ab (Kurtosis: 13.44)
Frequency
CD235ab
50000
40000
30000
20000
10000
0
0
1
2
3
4
5
6

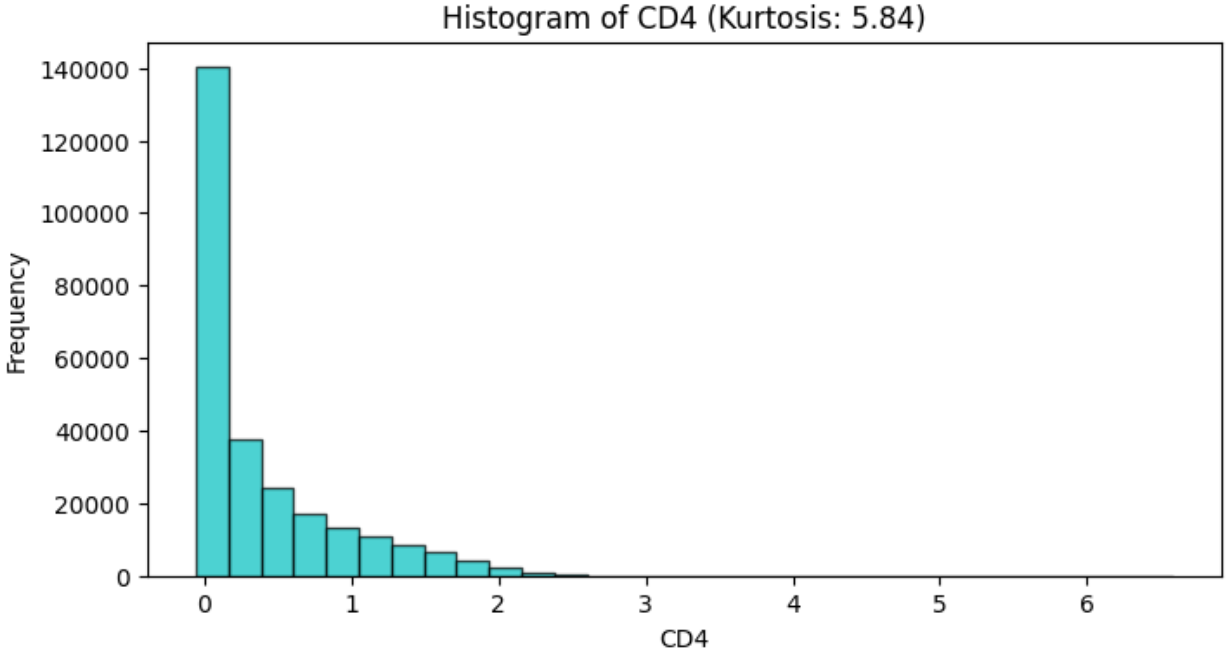

Histogram of CD3 (Kurtosis: 1.26)
Frequency
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7
CD3

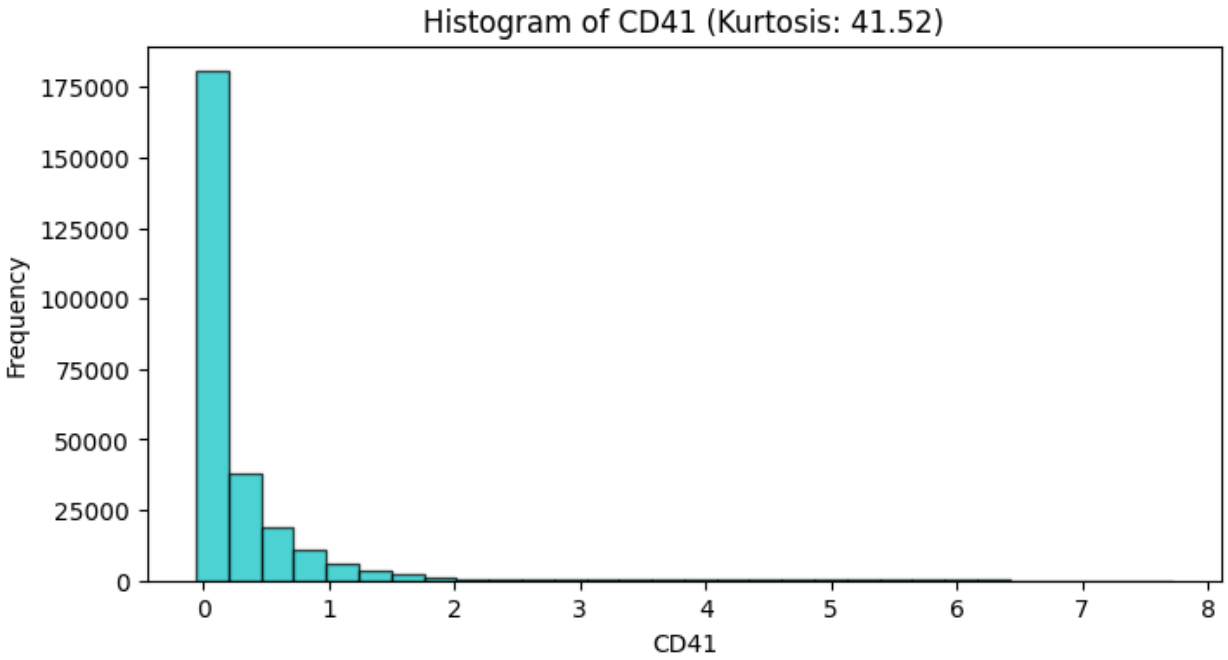

Histogram of CD321 (Kurtosis: 2.91)
Frequency
25000
20000
15000
10000
5000
0
0
1
2
3
4
5
6
7
CD321

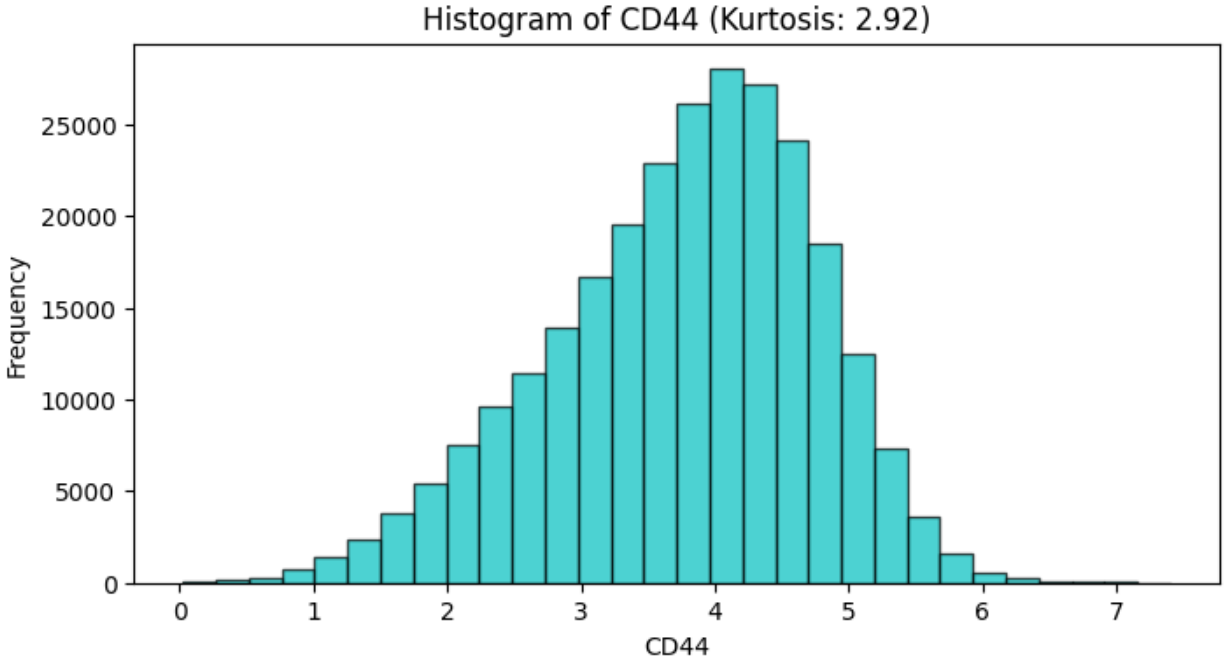

Histogram of CD33 (Kurtosis: 10.97)
Frequency
CD33
160000
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5

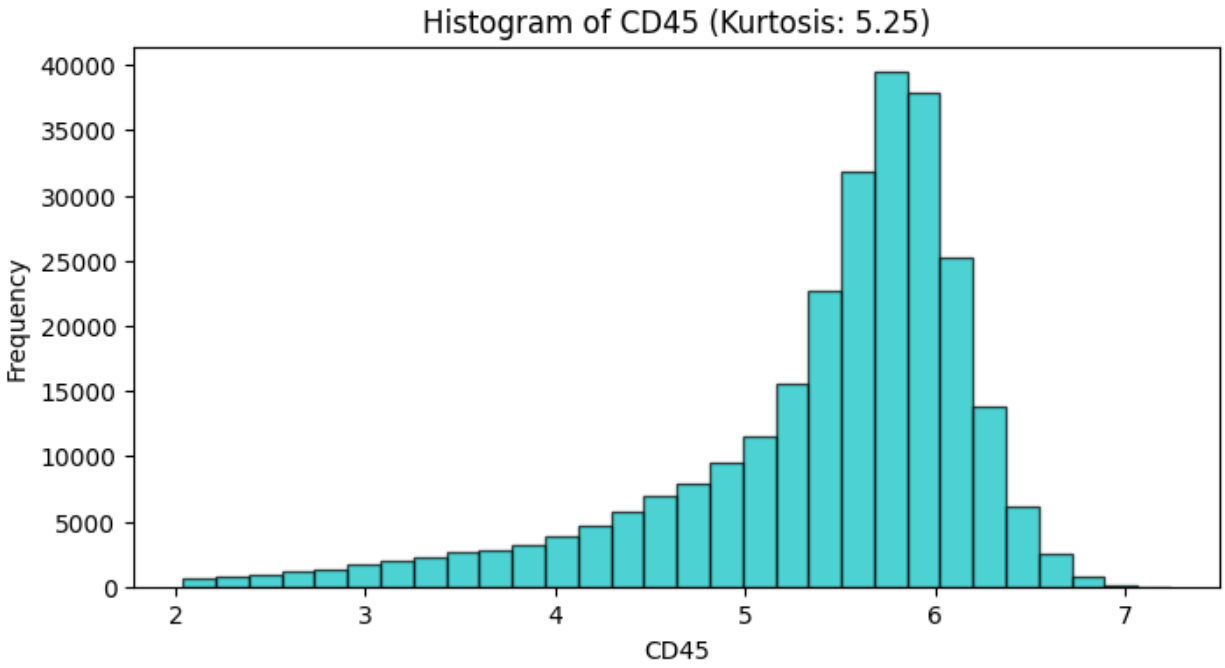

Histogram of CD34 (Kurtosis: 16.60)
Frequency
CD34
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5

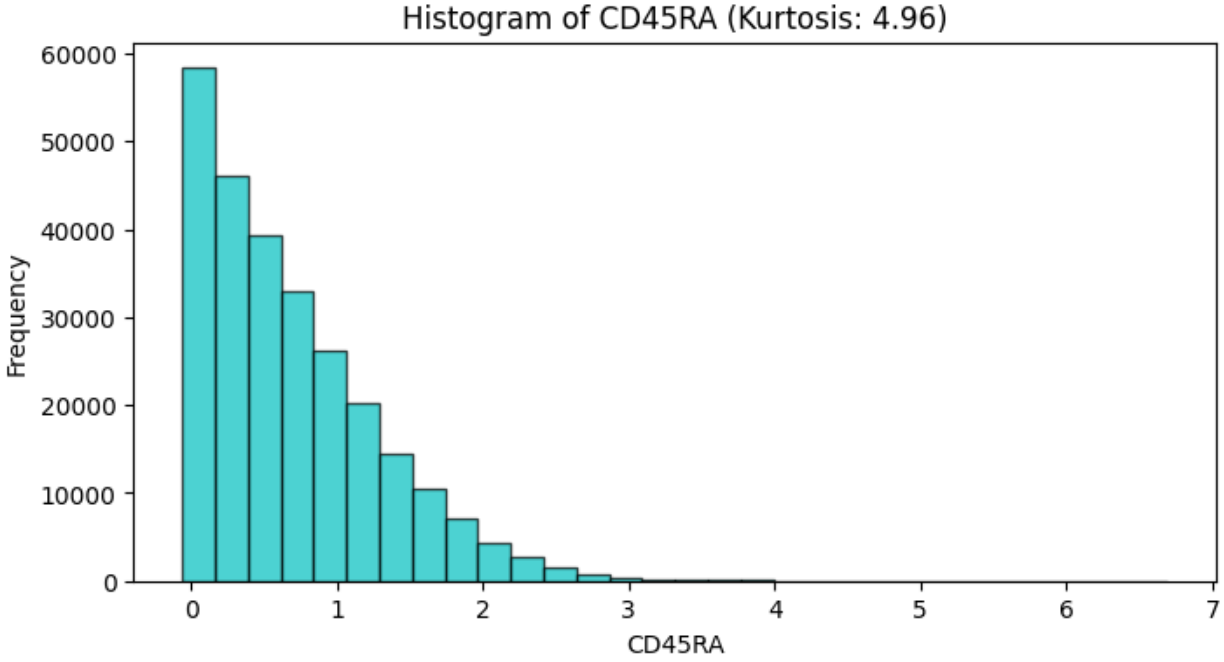

Histogram of CD38 (Kurtosis: 3.52)
Frequency
CD38
0
5000
10000
15000
20000
25000
30000
35000
0
1
2
3
4
5
6
7

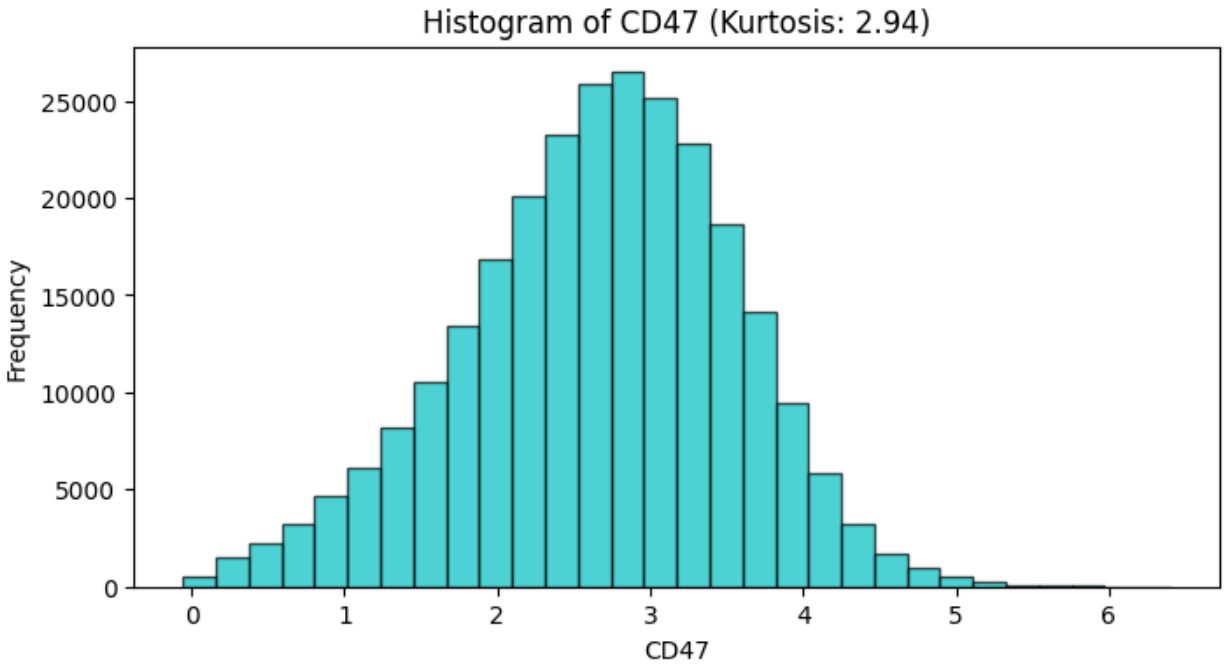

Histogram of CD4 (Kurtosis: 5.84)
Frequency
CD4
0
20000
40000
60000
80000
100000
120000
140000
0
1
2
3
4
5
6

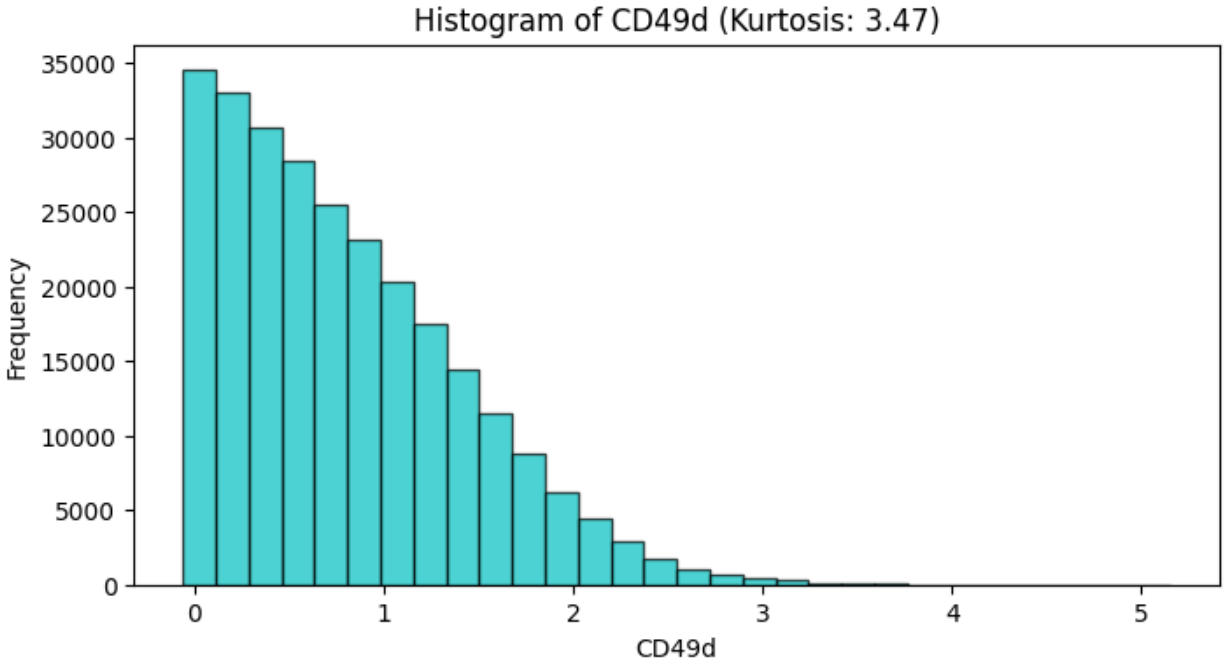

Histogram of CD41 (Kurtosis: 41.52)
Frequency
CD41
0
25000
50000
75000
100000
125000
150000
175000
0
1
2
3
4
5
6
7
8

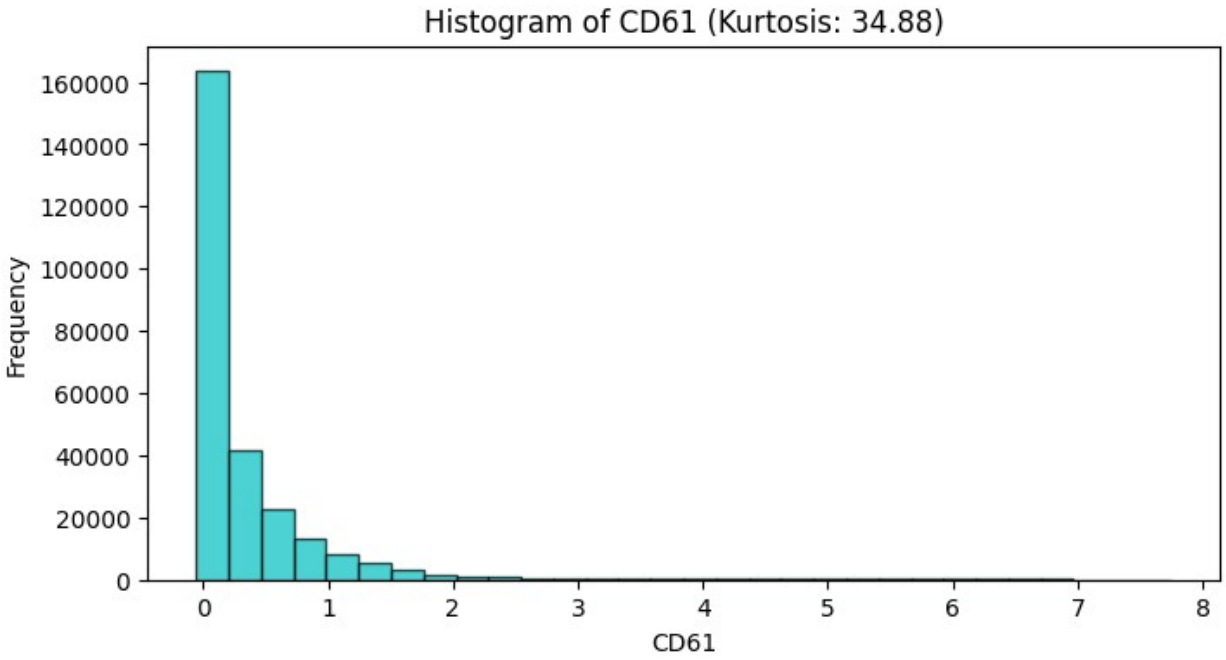

Histogram of CD44 (Kurtosis: 2.92)
Frequency
CD44
0
5000
10000
15000
20000
25000
0
1
2
3
4
5
6
7

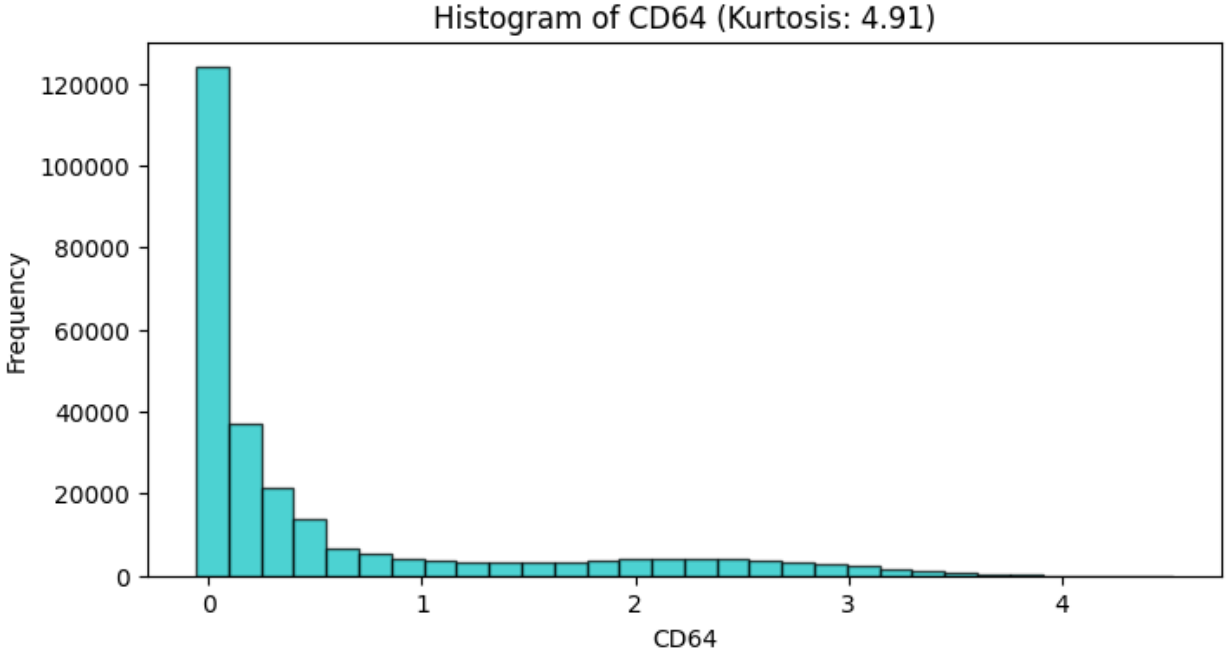

Histogram of CD45 (Kurtosis: 5.25)
Frequency
CD45
0
5000
10000
15000
20000
25000
30000
35000
40000
2
3
4
5
6
7

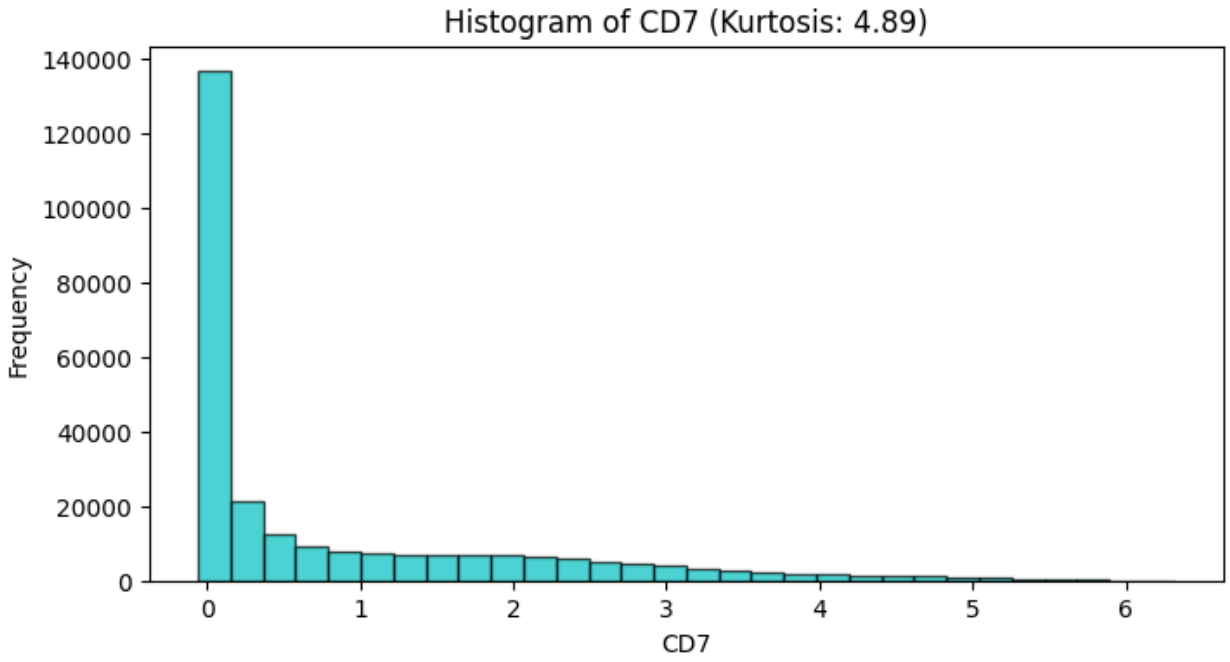

Histogram of CD45RA (Kurtosis: 4.96)
Frequency
CD45RA
0
10000
20000
30000
40000
50000
60000
0
1
2
3
4
5
6
7

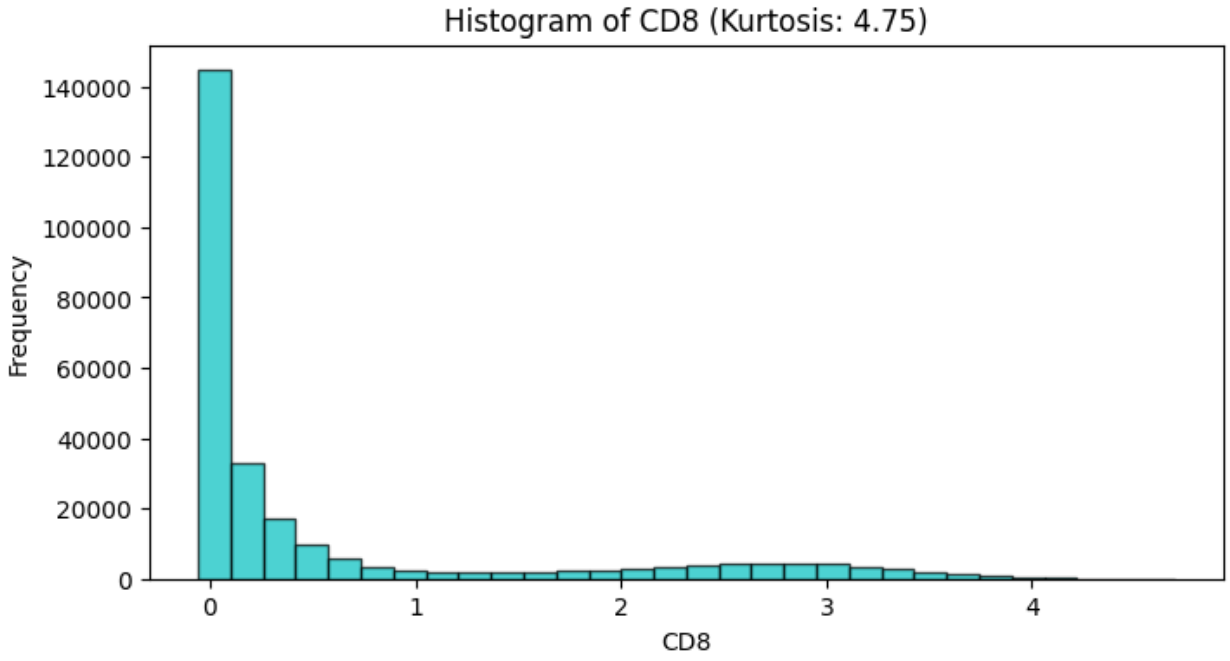

Histogram of CD47 (Kurtosis: 2.94)
Frequency
CD47
0
5000
10000
15000
20000
25000
0
1
2
3
4
5
6

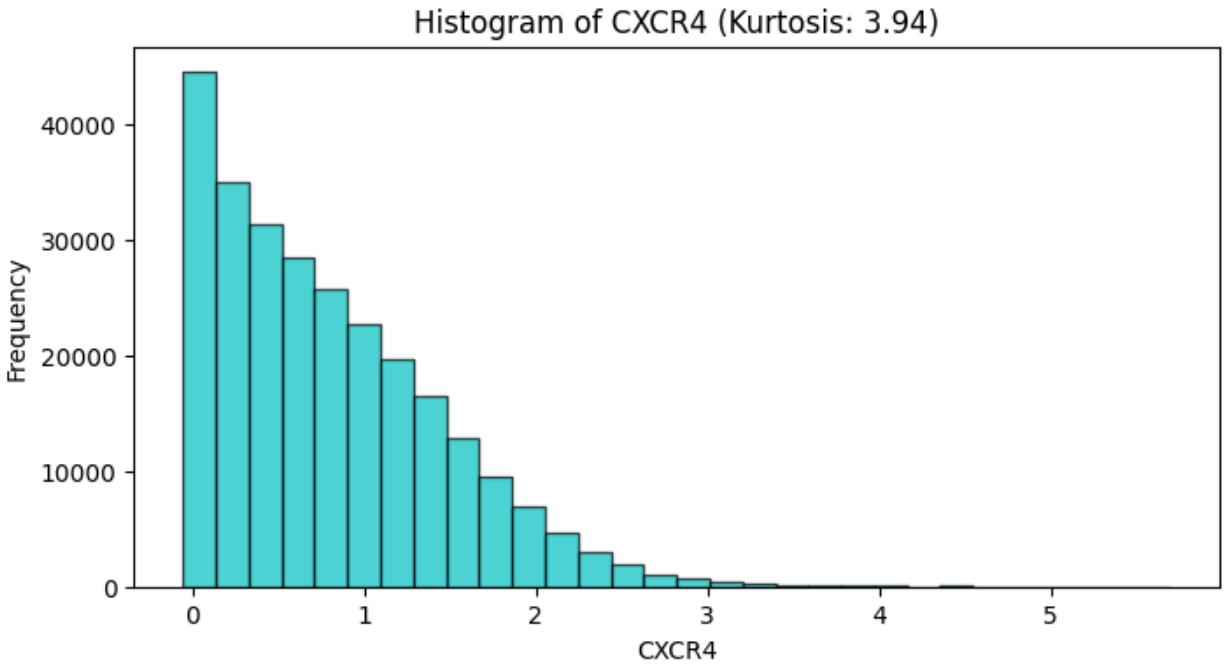

Histogram of CD49d (Kurtosis: 3.47)
Frequency
CD49d
0
5000
10000
15000
20000
25000
30000
35000
0
1
2
3
4
5

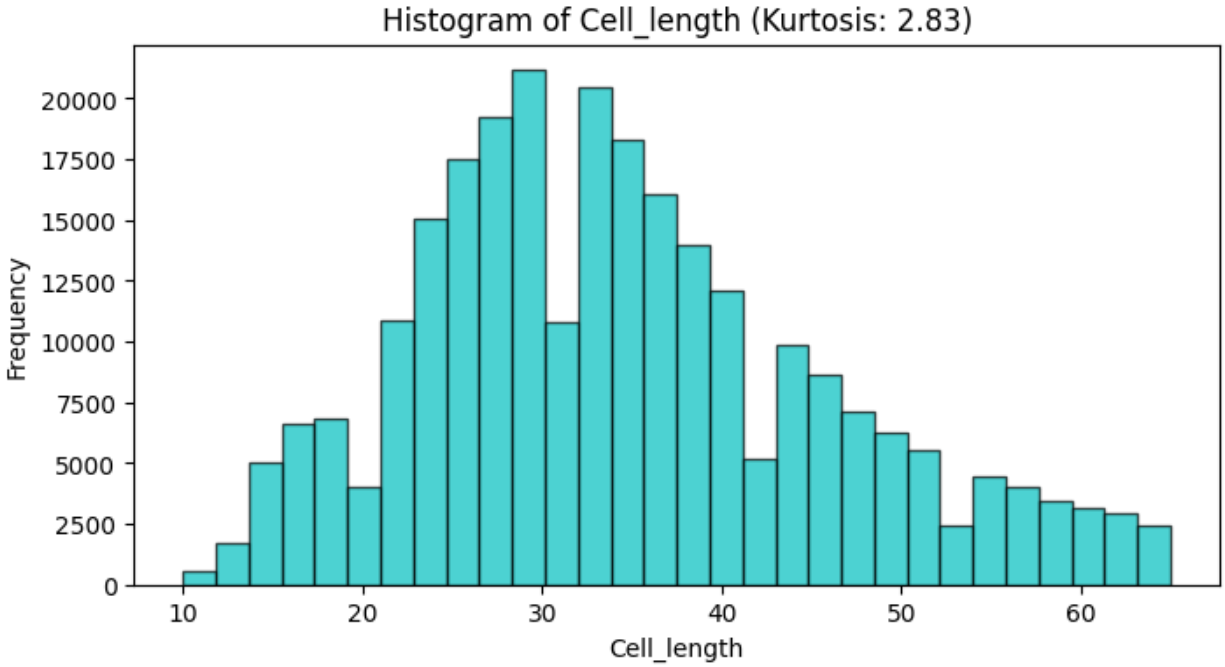

Histogram of CD61 (Kurtosis: 34.88)
Frequency
CD61
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6
7
8

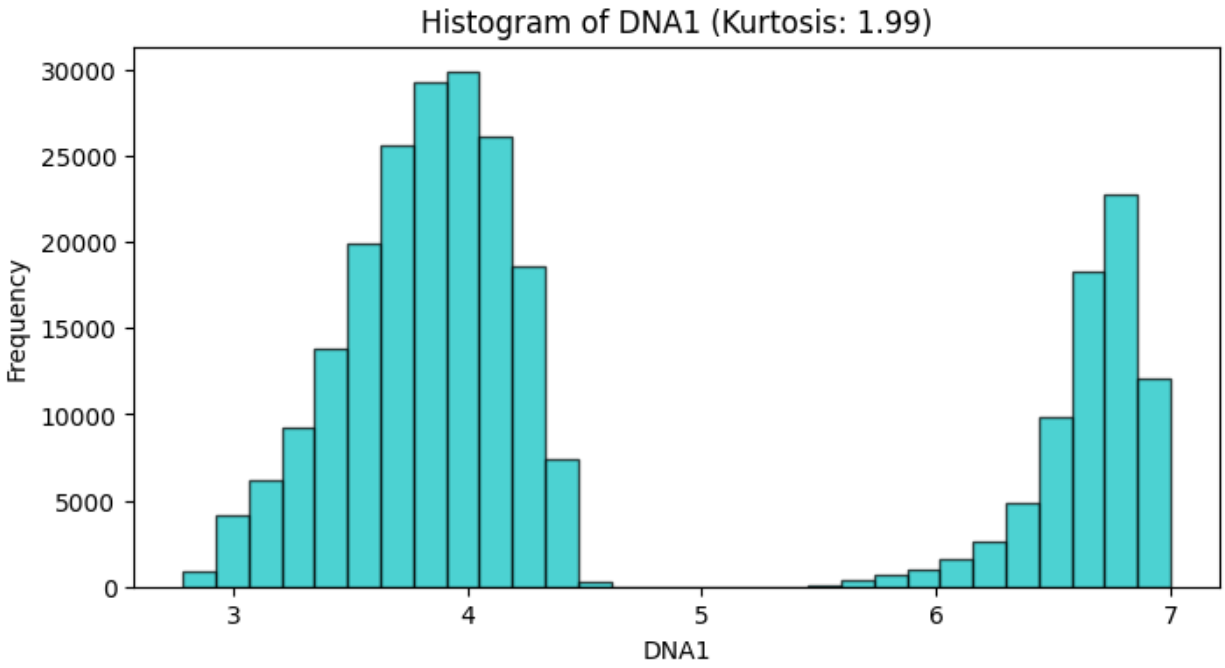

Histogram of CD64 (Kurtosis: 4.91)
Frequency
CD64
0
20000
40000
60000
80000
100000
120000
0
1
2
3
4

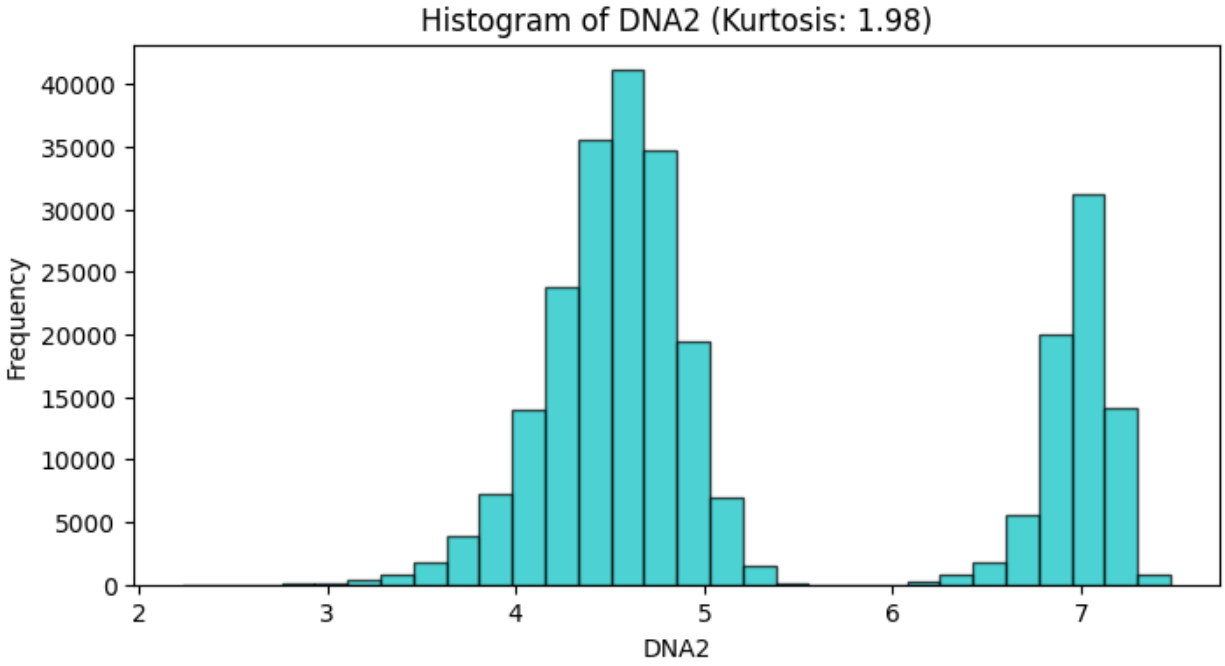

Histogram of CD7 (Kurtosis: 4.89)
Frequency
CD7
0
20000
40000
60000
80000
100000
120000
140000
0
1
2
3
4
5
6

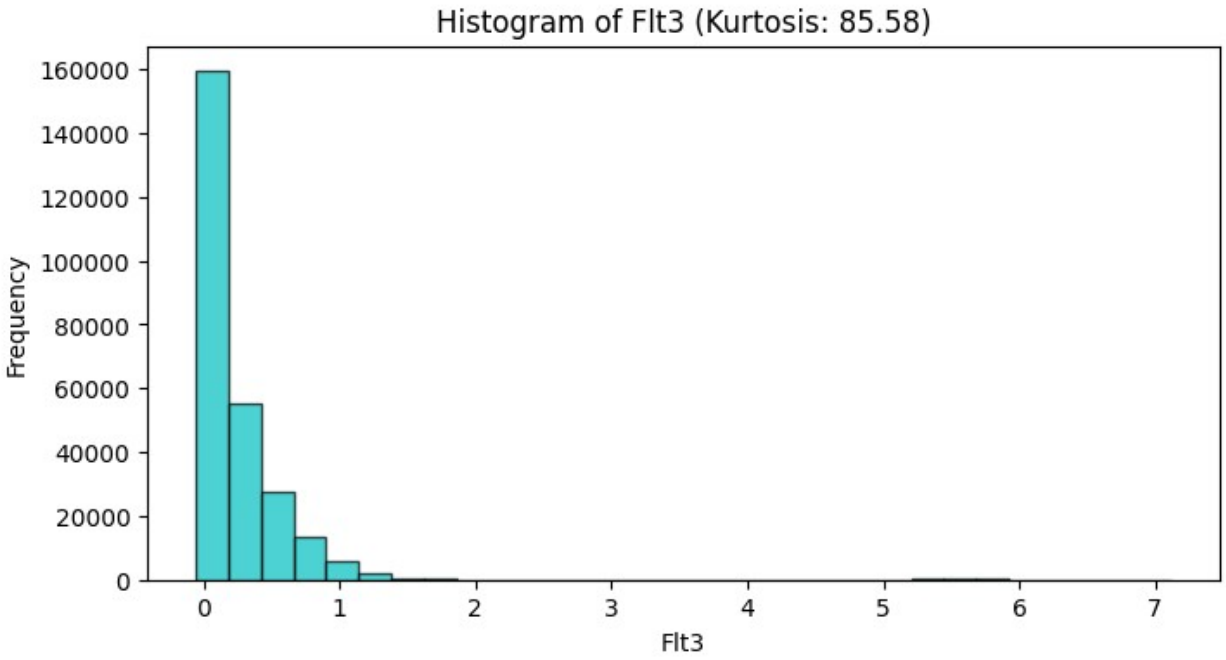

Histogram of CD8 (Kurtosis: 4.75)
Frequency
CD8
0
20000
40000
60000
80000
100000
120000
140000
0
1
2
3
4

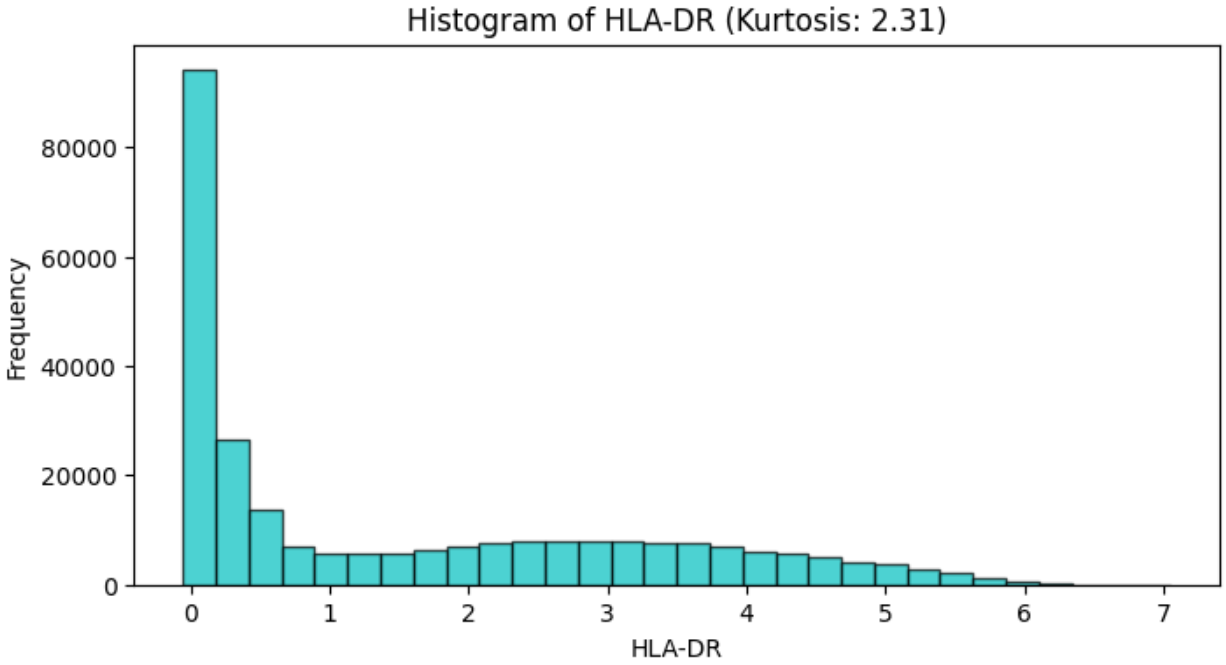

Histogram of CXCR4 (Kurtosis: 3.94)
Frequency
CXCR4
0
10000
20000
30000
40000
0
1
2
3
4
5

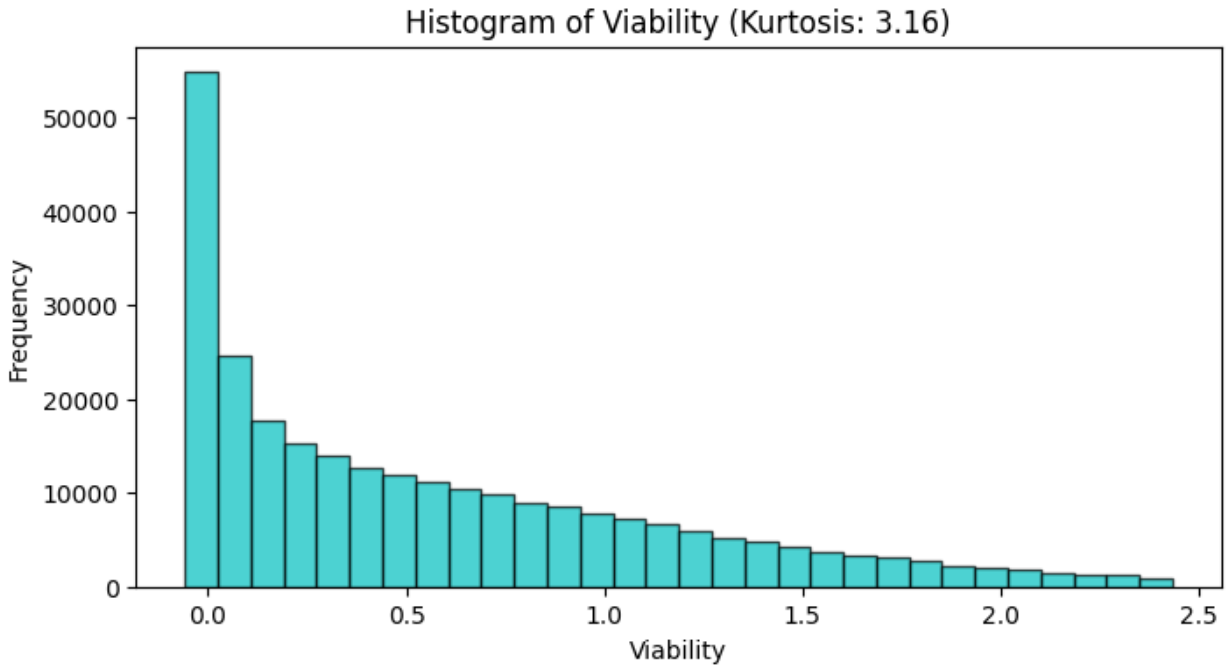

Histogram of Cell_length (Kurtosis: 2.83)
Frequency
Cell_length
0
2500
5000
7500
10000
12500
15000
17500
20000
10
20
30
40
50
60

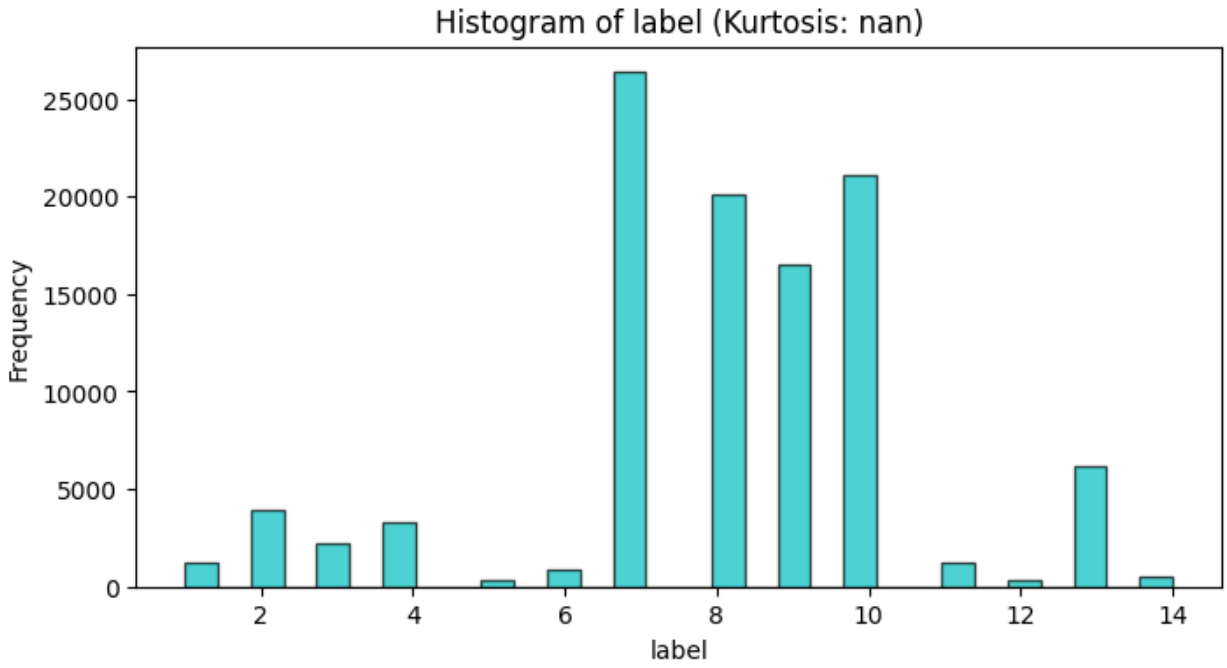

Histogram of DNA1 (Kurtosis: 1.99)
Frequency
DNA1
0
5000
10000
15000
20000
25000
30000
3
4
5
6
7


Histogram of DNA2 (Kurtosis: 1.98)
Frequency
DNA2
0
5000
10000
15000
20000
25000
30000
35000
40000
2
3
4
5
6
7


Histogram of Flt3 (Kurtosis: 85.58)
Frequency
Flt3
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6
7


Histogram of HLA-DR (Kurtosis: 2.31)
Frequency
HLA-DR
0
20000
40000
60000
80000
0
1
2
3
4
5
6
7

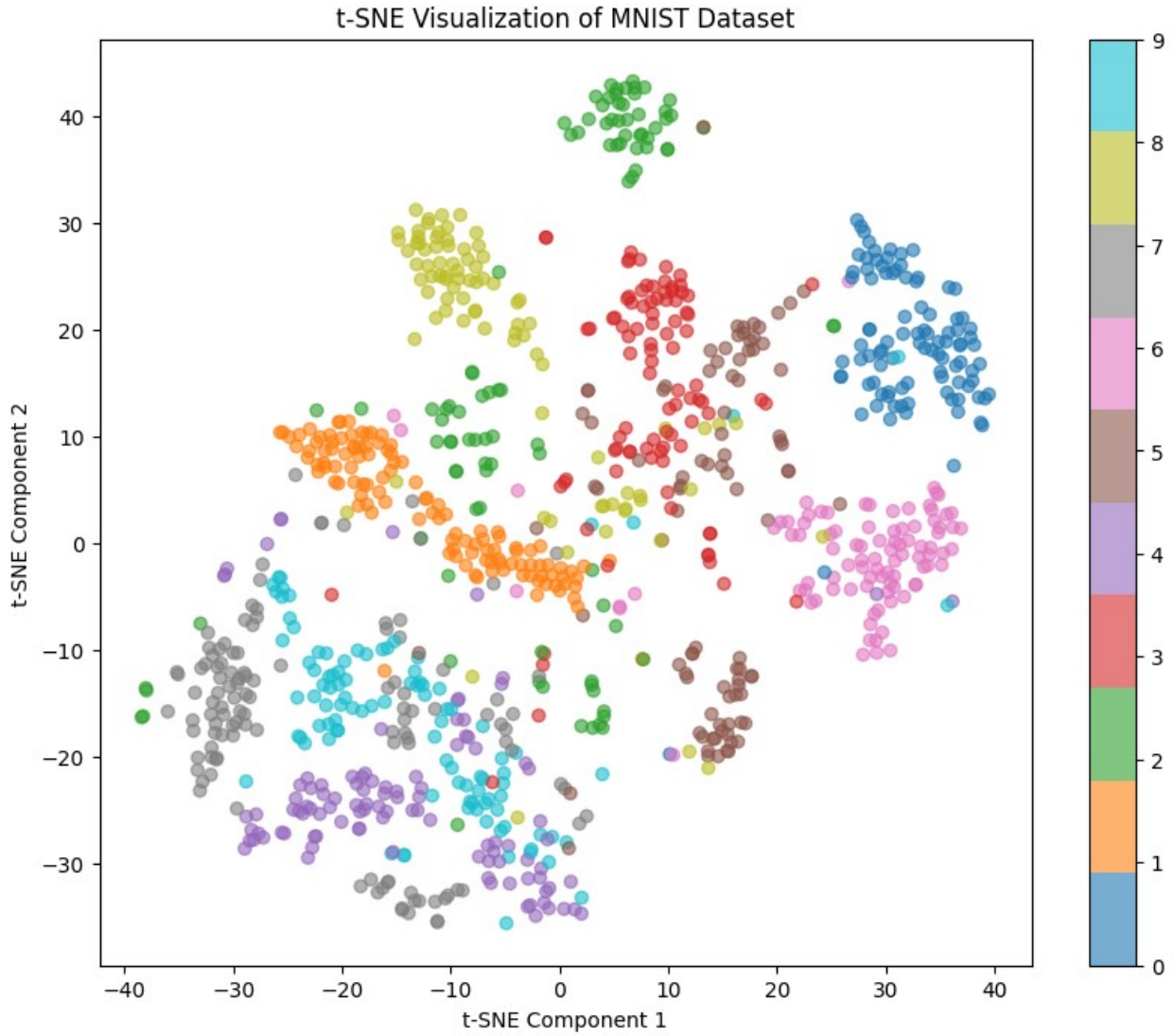





```
import tensorflow as tf
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import numpy as np

(train_images, train_labels), (test_images, test_labels) =
tf.keras.datasets.mnist.load_data()
train_images = train_images.astype('float32') / 255.0
```

```
test_images = test_images.astype('float32') / 255.0

n_samples = 1000
train_images_flat = train_images[:n_samples].reshape(n_samples, -1)
import tensorflow as tf
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import numpy as np

(train_images, train_labels), (test_images, test_labels) =
tf.keras.datasets.mnist.load_data()
train_images = train_images.astype('float32') / 255.0
test_images = test_images.astype('float32') / 255.0

n_samples = 1000
train_images_flat = train_images[:n_samples].reshape(n_samples, -1)
train_labels_subset = train_labels[:n_samples]

tsne = TSNE(n_components=2, random_state=42, perplexity=30)
tsne_results = tsne.fit_transform(train_images_flat)

train_images_tsne = tsne.fit_transform(train_images_flat)

# plot the results
plt.figure(figsize=(10, 8))
scatter = plt.scatter(train_images_tsne[:, 0], train_images_tsne[:,
1], c=train_labels_subset, cmap='tab10', alpha=0.6)
plt.colorbar(scatter, ticks=range(10))
plt.title('t-SNE Visualization of MNIST Dataset')
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.show()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```




```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

cleaned_new_df = df.dropna(subset=['label'])

numeric_columns =
cleaned_new_df.select_dtypes(include=[np.number]).columns.tolist()
features = [col for col in numeric_columns if col != 'label']

# Prepare the data
X = cleaned_new_df[features].values

imputer = SimpleImputer(strategy='mean')
```

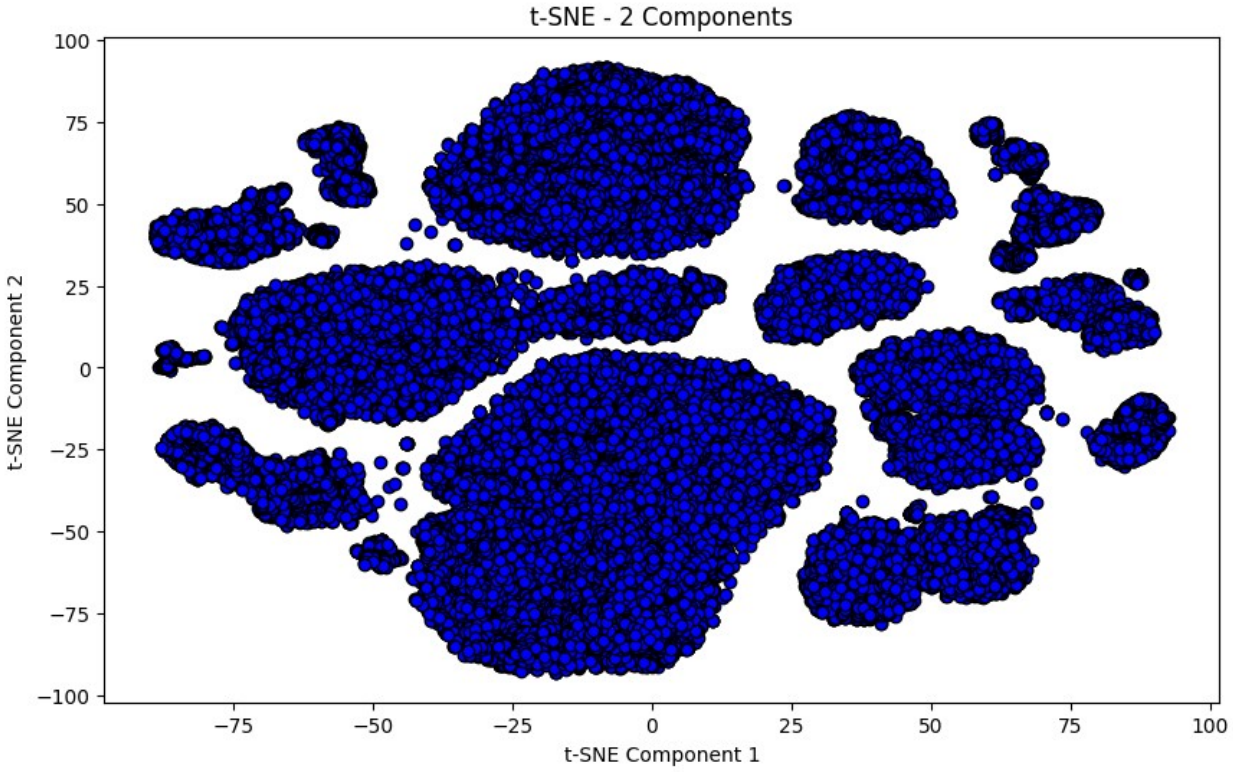
```
X_imputed = imputer.fit_transform(X)

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

# Initialize t-SNE
tsne = TSNE(n_components=2, random_state=42, perplexity=30,
n_iter=1000, verbose=1)

X_tsne = tsne.fit_transform(X_scaled)

cleaned_new_df['tsne-2d-one'] = X_tsne[:, 0]
cleaned_new_df['tsne-2d-two'] = X_tsne[:, 1]

# Plot the results
plt.figure(figsize=(10, 6))
plt.scatter(cleaned_new_df['tsne-2d-one'], cleaned_new_df['tsne-2d-
two'], c='blue', edgecolor='k')
plt.title('t-SNE - 2 Components')
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/manifold/
_t_sne.py:1162: FutureWarning: 'n_iter' was renamed to 'max_iter' in
version 1.5 and will be removed in 1.7.
  warnings.warn(

[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 104184 samples in 0.017s...
[t-SNE] Computed neighbors for 104184 samples in 79.391s...
[t-SNE] Computed conditional probabilities for sample 1000 / 104184
[t-SNE] Computed conditional probabilities for sample 2000 / 104184
[t-SNE] Computed conditional probabilities for sample 3000 / 104184
[t-SNE] Computed conditional probabilities for sample 4000 / 104184
[t-SNE] Computed conditional probabilities for sample 5000 / 104184
[t-SNE] Computed conditional probabilities for sample 6000 / 104184
[t-SNE] Computed conditional probabilities for sample 7000 / 104184
[t-SNE] Computed conditional probabilities for sample 8000 / 104184
[t-SNE] Computed conditional probabilities for sample 9000 / 104184
[t-SNE] Computed conditional probabilities for sample 10000 / 104184
[t-SNE] Computed conditional probabilities for sample 11000 / 104184
[t-SNE] Computed conditional probabilities for sample 12000 / 104184
[t-SNE] Computed conditional probabilities for sample 13000 / 104184
[t-SNE] Computed conditional probabilities for sample 14000 / 104184
[t-SNE] Computed conditional probabilities for sample 15000 / 104184
[t-SNE] Computed conditional probabilities for sample 16000 / 104184
[t-SNE] Computed conditional probabilities for sample 17000 / 104184
[t-SNE] Computed conditional probabilities for sample 18000 / 104184
```

```
[t-SNE] Computed conditional probabilities for sample 19000 / 104184
[t-SNE] Computed conditional probabilities for sample 20000 / 104184
[t-SNE] Computed conditional probabilities for sample 21000 / 104184
[t-SNE] Computed conditional probabilities for sample 22000 / 104184
[t-SNE] Computed conditional probabilities for sample 23000 / 104184
[t-SNE] Computed conditional probabilities for sample 24000 / 104184
[t-SNE] Computed conditional probabilities for sample 25000 / 104184
[t-SNE] Computed conditional probabilities for sample 26000 / 104184
[t-SNE] Computed conditional probabilities for sample 27000 / 104184
[t-SNE] Computed conditional probabilities for sample 28000 / 104184
[t-SNE] Computed conditional probabilities for sample 29000 / 104184
[t-SNE] Computed conditional probabilities for sample 30000 / 104184
[t-SNE] Computed conditional probabilities for sample 31000 / 104184
[t-SNE] Computed conditional probabilities for sample 32000 / 104184
[t-SNE] Computed conditional probabilities for sample 33000 / 104184
[t-SNE] Computed conditional probabilities for sample 34000 / 104184
[t-SNE] Computed conditional probabilities for sample 35000 / 104184
[t-SNE] Computed conditional probabilities for sample 36000 / 104184
[t-SNE] Computed conditional probabilities for sample 37000 / 104184
[t-SNE] Computed conditional probabilities for sample 38000 / 104184
[t-SNE] Computed conditional probabilities for sample 39000 / 104184
[t-SNE] Computed conditional probabilities for sample 40000 / 104184
[t-SNE] Computed conditional probabilities for sample 41000 / 104184
[t-SNE] Computed conditional probabilities for sample 42000 / 104184
[t-SNE] Computed conditional probabilities for sample 43000 / 104184
[t-SNE] Computed conditional probabilities for sample 44000 / 104184
[t-SNE] Computed conditional probabilities for sample 45000 / 104184
[t-SNE] Computed conditional probabilities for sample 46000 / 104184
[t-SNE] Computed conditional probabilities for sample 47000 / 104184
[t-SNE] Computed conditional probabilities for sample 48000 / 104184
[t-SNE] Computed conditional probabilities for sample 49000 / 104184
[t-SNE] Computed conditional probabilities for sample 50000 / 104184
[t-SNE] Computed conditional probabilities for sample 51000 / 104184
[t-SNE] Computed conditional probabilities for sample 52000 / 104184
[t-SNE] Computed conditional probabilities for sample 53000 / 104184
[t-SNE] Computed conditional probabilities for sample 54000 / 104184
[t-SNE] Computed conditional probabilities for sample 55000 / 104184
[t-SNE] Computed conditional probabilities for sample 56000 / 104184
[t-SNE] Computed conditional probabilities for sample 57000 / 104184
[t-SNE] Computed conditional probabilities for sample 58000 / 104184
[t-SNE] Computed conditional probabilities for sample 59000 / 104184
[t-SNE] Computed conditional probabilities for sample 60000 / 104184
[t-SNE] Computed conditional probabilities for sample 61000 / 104184
[t-SNE] Computed conditional probabilities for sample 62000 / 104184
[t-SNE] Computed conditional probabilities for sample 63000 / 104184
[t-SNE] Computed conditional probabilities for sample 64000 / 104184
[t-SNE] Computed conditional probabilities for sample 65000 / 104184
[t-SNE] Computed conditional probabilities for sample 66000 / 104184
[t-SNE] Computed conditional probabilities for sample 67000 / 104184
```

```
[t-SNE] Computed conditional probabilities for sample 68000 / 104184
[t-SNE] Computed conditional probabilities for sample 69000 / 104184
[t-SNE] Computed conditional probabilities for sample 70000 / 104184
[t-SNE] Computed conditional probabilities for sample 71000 / 104184
[t-SNE] Computed conditional probabilities for sample 72000 / 104184
[t-SNE] Computed conditional probabilities for sample 73000 / 104184
[t-SNE] Computed conditional probabilities for sample 74000 / 104184
[t-SNE] Computed conditional probabilities for sample 75000 / 104184
[t-SNE] Computed conditional probabilities for sample 76000 / 104184
[t-SNE] Computed conditional probabilities for sample 77000 / 104184
[t-SNE] Computed conditional probabilities for sample 78000 / 104184
[t-SNE] Computed conditional probabilities for sample 79000 / 104184
[t-SNE] Computed conditional probabilities for sample 80000 / 104184
[t-SNE] Computed conditional probabilities for sample 81000 / 104184
[t-SNE] Computed conditional probabilities for sample 82000 / 104184
[t-SNE] Computed conditional probabilities for sample 83000 / 104184
[t-SNE] Computed conditional probabilities for sample 84000 / 104184
[t-SNE] Computed conditional probabilities for sample 85000 / 104184
[t-SNE] Computed conditional probabilities for sample 86000 / 104184
[t-SNE] Computed conditional probabilities for sample 87000 / 104184
[t-SNE] Computed conditional probabilities for sample 88000 / 104184
[t-SNE] Computed conditional probabilities for sample 89000 / 104184
[t-SNE] Computed conditional probabilities for sample 90000 / 104184
[t-SNE] Computed conditional probabilities for sample 91000 / 104184
[t-SNE] Computed conditional probabilities for sample 92000 / 104184
[t-SNE] Computed conditional probabilities for sample 93000 / 104184
[t-SNE] Computed conditional probabilities for sample 94000 / 104184
[t-SNE] Computed conditional probabilities for sample 95000 / 104184
[t-SNE] Computed conditional probabilities for sample 96000 / 104184
[t-SNE] Computed conditional probabilities for sample 97000 / 104184
[t-SNE] Computed conditional probabilities for sample 98000 / 104184
[t-SNE] Computed conditional probabilities for sample 99000 / 104184
[t-SNE] Computed conditional probabilities for sample 100000 / 104184
[t-SNE] Computed conditional probabilities for sample 101000 / 104184
[t-SNE] Computed conditional probabilities for sample 102000 / 104184
[t-SNE] Computed conditional probabilities for sample 103000 / 104184
[t-SNE] Computed conditional probabilities for sample 104000 / 104184
[t-SNE] Computed conditional probabilities for sample 104184 / 104184
[t-SNE] Mean sigma: 1.068890
[t-SNE] KL divergence after 250 iterations with early exaggeration:
95.806900
[t-SNE] KL divergence after 1000 iterations: 3.487447

<ipython-input-22-98cf05c93f85>:27: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
```

```
  cleaned_new_df['tsne-2d-one'] = X_tsne[:, 0]
<ipython-input-22-98cf05c93f85>:28: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  cleaned_new_df['tsne-2d-two'] = X_tsne[:, 1]
```




```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

# Assuming 'new_df' is your dataset, and the 'label' column contains
class labels
cleaned_new_df = df.dropna(subset=['label'])

# Select numerical features for t-SNE
numeric_columns =
cleaned_new_df.select_dtypes(include=[np.number]).columns.tolist()
features = [col for col in numeric_columns if col != 'label']
```

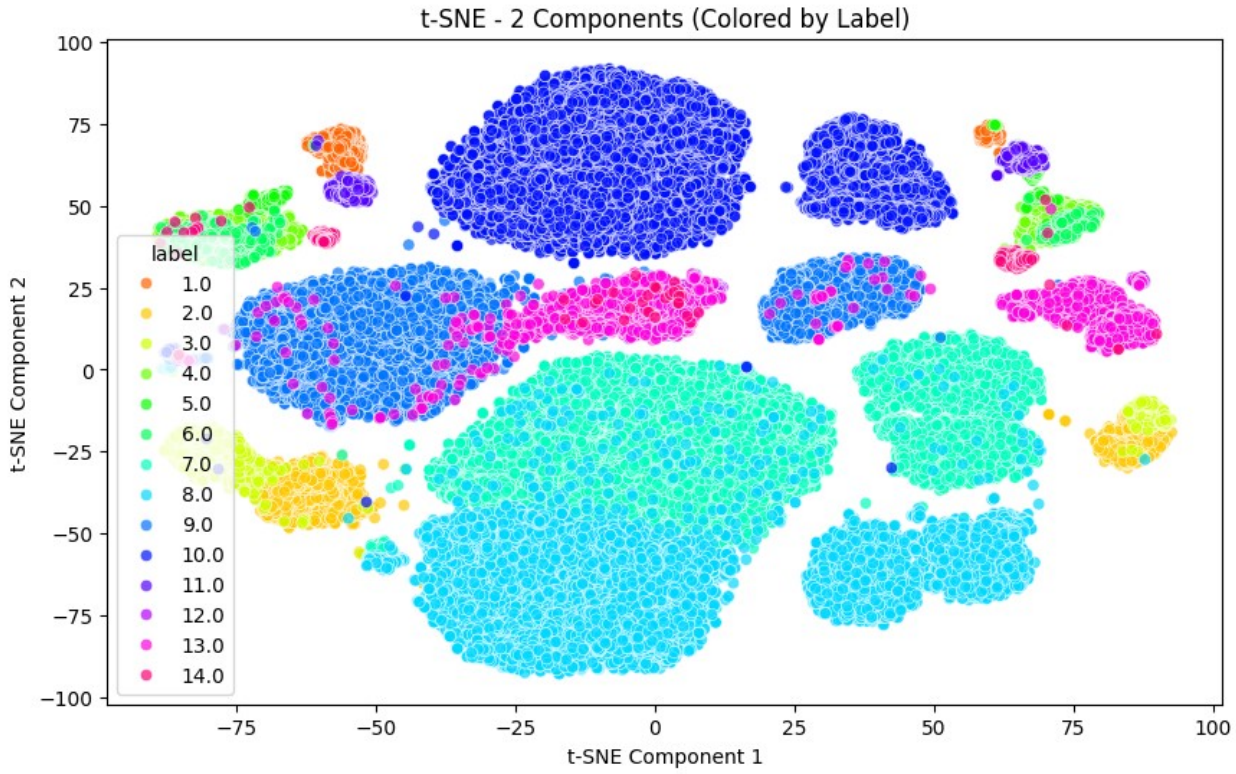
```
# Prepare the data
X = cleaned_new_df[features].values

# Handle missing values (if any are left)
imputer = SimpleImputer(strategy='mean')
X_imputed = imputer.fit_transform(X)

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

# Initialize t-SNE
tsne = TSNE(n_components=2, random_state=42, perplexity=30,
n_iter=1000, verbose=1)

# Fit and transform the data
X_tsne = tsne.fit_transform(X_scaled)

# Add t-SNE results to the DataFrame
cleaned_new_df['tsne-2d-one'] = X_tsne[:, 0]
cleaned_new_df['tsne-2d-two'] = X_tsne[:, 1]

# Plot the t-SNE results with different colors for each label
plt.figure(figsize=(10, 6))
sns.scatterplot(
    x='tsne-2d-one',
    y='tsne-2d-two',
    hue='label',  # Use the 'label' column for color encoding
    palette=sns.color_palette("hsv",
len(cleaned_new_df['label'].unique())),  # Color palette for labels
    data=cleaned_new_df,
    legend='full',
    alpha=0.7
)

plt.title('t-SNE - 2 Components (Colored by Label)')
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/manifold/
_t_sne.py:1162: FutureWarning: 'n_iter' was renamed to 'max_iter' in
version 1.5 and will be removed in 1.7.
  warnings.warn(

[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 104184 samples in 0.018s...
[t-SNE] Computed neighbors for 104184 samples in 80.865s...
[t-SNE] Computed conditional probabilities for sample 1000 / 104184
```

```
[t-SNE] Computed conditional probabilities for sample 2000 / 104184
[t-SNE] Computed conditional probabilities for sample 3000 / 104184
[t-SNE] Computed conditional probabilities for sample 4000 / 104184
[t-SNE] Computed conditional probabilities for sample 5000 / 104184
[t-SNE] Computed conditional probabilities for sample 6000 / 104184
[t-SNE] Computed conditional probabilities for sample 7000 / 104184
[t-SNE] Computed conditional probabilities for sample 8000 / 104184
[t-SNE] Computed conditional probabilities for sample 9000 / 104184
[t-SNE] Computed conditional probabilities for sample 10000 / 104184
[t-SNE] Computed conditional probabilities for sample 11000 / 104184
[t-SNE] Computed conditional probabilities for sample 12000 / 104184
[t-SNE] Computed conditional probabilities for sample 13000 / 104184
[t-SNE] Computed conditional probabilities for sample 14000 / 104184
[t-SNE] Computed conditional probabilities for sample 15000 / 104184
[t-SNE] Computed conditional probabilities for sample 16000 / 104184
[t-SNE] Computed conditional probabilities for sample 17000 / 104184
[t-SNE] Computed conditional probabilities for sample 18000 / 104184
[t-SNE] Computed conditional probabilities for sample 19000 / 104184
[t-SNE] Computed conditional probabilities for sample 20000 / 104184
[t-SNE] Computed conditional probabilities for sample 21000 / 104184
[t-SNE] Computed conditional probabilities for sample 22000 / 104184
[t-SNE] Computed conditional probabilities for sample 23000 / 104184
[t-SNE] Computed conditional probabilities for sample 24000 / 104184
[t-SNE] Computed conditional probabilities for sample 25000 / 104184
[t-SNE] Computed conditional probabilities for sample 26000 / 104184
[t-SNE] Computed conditional probabilities for sample 27000 / 104184
[t-SNE] Computed conditional probabilities for sample 28000 / 104184
[t-SNE] Computed conditional probabilities for sample 29000 / 104184
[t-SNE] Computed conditional probabilities for sample 30000 / 104184
[t-SNE] Computed conditional probabilities for sample 31000 / 104184
[t-SNE] Computed conditional probabilities for sample 32000 / 104184
[t-SNE] Computed conditional probabilities for sample 33000 / 104184
[t-SNE] Computed conditional probabilities for sample 34000 / 104184
[t-SNE] Computed conditional probabilities for sample 35000 / 104184
[t-SNE] Computed conditional probabilities for sample 36000 / 104184
[t-SNE] Computed conditional probabilities for sample 37000 / 104184
[t-SNE] Computed conditional probabilities for sample 38000 / 104184
[t-SNE] Computed conditional probabilities for sample 39000 / 104184
[t-SNE] Computed conditional probabilities for sample 40000 / 104184
[t-SNE] Computed conditional probabilities for sample 41000 / 104184
[t-SNE] Computed conditional probabilities for sample 42000 / 104184
[t-SNE] Computed conditional probabilities for sample 43000 / 104184
[t-SNE] Computed conditional probabilities for sample 44000 / 104184
[t-SNE] Computed conditional probabilities for sample 45000 / 104184
[t-SNE] Computed conditional probabilities for sample 46000 / 104184
[t-SNE] Computed conditional probabilities for sample 47000 / 104184
[t-SNE] Computed conditional probabilities for sample 48000 / 104184
[t-SNE] Computed conditional probabilities for sample 49000 / 104184
[t-SNE] Computed conditional probabilities for sample 50000 / 104184
```

```
[t-SNE] Computed conditional probabilities for sample 51000 / 104184
[t-SNE] Computed conditional probabilities for sample 52000 / 104184
[t-SNE] Computed conditional probabilities for sample 53000 / 104184
[t-SNE] Computed conditional probabilities for sample 54000 / 104184
[t-SNE] Computed conditional probabilities for sample 55000 / 104184
[t-SNE] Computed conditional probabilities for sample 56000 / 104184
[t-SNE] Computed conditional probabilities for sample 57000 / 104184
[t-SNE] Computed conditional probabilities for sample 58000 / 104184
[t-SNE] Computed conditional probabilities for sample 59000 / 104184
[t-SNE] Computed conditional probabilities for sample 60000 / 104184
[t-SNE] Computed conditional probabilities for sample 61000 / 104184
[t-SNE] Computed conditional probabilities for sample 62000 / 104184
[t-SNE] Computed conditional probabilities for sample 63000 / 104184
[t-SNE] Computed conditional probabilities for sample 64000 / 104184
[t-SNE] Computed conditional probabilities for sample 65000 / 104184
[t-SNE] Computed conditional probabilities for sample 66000 / 104184
[t-SNE] Computed conditional probabilities for sample 67000 / 104184
[t-SNE] Computed conditional probabilities for sample 68000 / 104184
[t-SNE] Computed conditional probabilities for sample 69000 / 104184
[t-SNE] Computed conditional probabilities for sample 70000 / 104184
[t-SNE] Computed conditional probabilities for sample 71000 / 104184
[t-SNE] Computed conditional probabilities for sample 72000 / 104184
[t-SNE] Computed conditional probabilities for sample 73000 / 104184
[t-SNE] Computed conditional probabilities for sample 74000 / 104184
[t-SNE] Computed conditional probabilities for sample 75000 / 104184
[t-SNE] Computed conditional probabilities for sample 76000 / 104184
[t-SNE] Computed conditional probabilities for sample 77000 / 104184
[t-SNE] Computed conditional probabilities for sample 78000 / 104184
[t-SNE] Computed conditional probabilities for sample 79000 / 104184
[t-SNE] Computed conditional probabilities for sample 80000 / 104184
[t-SNE] Computed conditional probabilities for sample 81000 / 104184
[t-SNE] Computed conditional probabilities for sample 82000 / 104184
[t-SNE] Computed conditional probabilities for sample 83000 / 104184
[t-SNE] Computed conditional probabilities for sample 84000 / 104184
[t-SNE] Computed conditional probabilities for sample 85000 / 104184
[t-SNE] Computed conditional probabilities for sample 86000 / 104184
[t-SNE] Computed conditional probabilities for sample 87000 / 104184
[t-SNE] Computed conditional probabilities for sample 88000 / 104184
[t-SNE] Computed conditional probabilities for sample 89000 / 104184
[t-SNE] Computed conditional probabilities for sample 90000 / 104184
[t-SNE] Computed conditional probabilities for sample 91000 / 104184
[t-SNE] Computed conditional probabilities for sample 92000 / 104184
[t-SNE] Computed conditional probabilities for sample 93000 / 104184
[t-SNE] Computed conditional probabilities for sample 94000 / 104184
[t-SNE] Computed conditional probabilities for sample 95000 / 104184
[t-SNE] Computed conditional probabilities for sample 96000 / 104184
[t-SNE] Computed conditional probabilities for sample 97000 / 104184
[t-SNE] Computed conditional probabilities for sample 98000 / 104184
[t-SNE] Computed conditional probabilities for sample 99000 / 104184
```

```
[t-SNE] Computed conditional probabilities for sample 100000 / 104184
[t-SNE] Computed conditional probabilities for sample 101000 / 104184
[t-SNE] Computed conditional probabilities for sample 102000 / 104184
[t-SNE] Computed conditional probabilities for sample 103000 / 104184
[t-SNE] Computed conditional probabilities for sample 104000 / 104184
[t-SNE] Computed conditional probabilities for sample 104184 / 104184
[t-SNE] Mean sigma: 1.068890
[t-SNE] KL divergence after 250 iterations with early exaggeration:
95.806900
[t-SNE] KL divergence after 1000 iterations: 3.487447

<ipython-input-23-2618c6de12d9>:33: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  cleaned_new_df['tsne-2d-one'] = X_tsne[:, 0]
<ipython-input-23-2618c6de12d9>:34: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  cleaned_new_df['tsne-2d-two'] = X_tsne[:, 1]
```




```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from mpl_toolkits.mplot3d import Axes3D  # For 3D plotting

new_df =
df.drop(['Event','Time','file_number','event_number','individual'],axi
s=1)
# Assuming 'new_df' is your dataset, and the 'label' column contains
class labels
cleaned_new_df = new_df.dropna(subset=['label'])

# Select numerical features for t-SNE
numeric_columns =
cleaned_new_df.select_dtypes(include=[np.number]).columns.tolist()
features = [col for col in numeric_columns if col != 'label']

# Prepare the data
X = cleaned_new_df[features].values

# Handle missing values (if any are left)
imputer = SimpleImputer(strategy='mean')
X_imputed = imputer.fit_transform(X)
```

```python
# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_imputed)

# Initialize t-SNE with 3 components
tsne = TSNE(n_components=3, random_state=42, perplexity=30,
n_iter=1000, verbose=1)

# Fit and transform the data
X_tsne = tsne.fit_transform(X_scaled)

# Add t-SNE results to the DataFrame
cleaned_new_df['tsne-3d-one'] = X_tsne[:, 0]
cleaned_new_df['tsne-3d-two'] = X_tsne[:, 1]
cleaned_new_df['tsne-3d-three'] = X_tsne[:, 2]

# Plot the 3D t-SNE results with different colors for each label
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# 3D scatter plot
scatter = ax.scatter(
    cleaned_new_df['tsne-3d-one'],
    cleaned_new_df['tsne-3d-two'],
    cleaned_new_df['tsne-3d-three'],
    c=cleaned_new_df['label'],  # Color points based on 'label'
    cmap='hsv',  # Color map for the labels
    edgecolor='k',
    alpha=0.7
)

# Add axis labels and title
ax.set_title('t-SNE - 3 Components')
ax.set_xlabel('t-SNE Component 1')
ax.set_ylabel('t-SNE Component 2')
ax.set_zlabel('t-SNE Component 3')

# Show color bar
plt.colorbar(scatter)
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/manifold/
_t_sne.py:1162: FutureWarning: 'n_iter' was renamed to 'max_iter' in
version 1.5 and will be removed in 1.7.
  warnings.warn(

[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 104184 samples in 0.016s...
[t-SNE] Computed neighbors for 104184 samples in 94.516s...
```

```
[t-SNE] Computed conditional probabilities for sample 1000 / 104184
[t-SNE] Computed conditional probabilities for sample 2000 / 104184
[t-SNE] Computed conditional probabilities for sample 3000 / 104184
[t-SNE] Computed conditional probabilities for sample 4000 / 104184
[t-SNE] Computed conditional probabilities for sample 5000 / 104184
[t-SNE] Computed conditional probabilities for sample 6000 / 104184
[t-SNE] Computed conditional probabilities for sample 7000 / 104184
[t-SNE] Computed conditional probabilities for sample 8000 / 104184
[t-SNE] Computed conditional probabilities for sample 9000 / 104184
[t-SNE] Computed conditional probabilities for sample 10000 / 104184
[t-SNE] Computed conditional probabilities for sample 11000 / 104184
[t-SNE] Computed conditional probabilities for sample 12000 / 104184
[t-SNE] Computed conditional probabilities for sample 13000 / 104184
[t-SNE] Computed conditional probabilities for sample 14000 / 104184
[t-SNE] Computed conditional probabilities for sample 15000 / 104184
[t-SNE] Computed conditional probabilities for sample 16000 / 104184
[t-SNE] Computed conditional probabilities for sample 17000 / 104184
[t-SNE] Computed conditional probabilities for sample 18000 / 104184
[t-SNE] Computed conditional probabilities for sample 19000 / 104184
[t-SNE] Computed conditional probabilities for sample 20000 / 104184
[t-SNE] Computed conditional probabilities for sample 21000 / 104184
[t-SNE] Computed conditional probabilities for sample 22000 / 104184
[t-SNE] Computed conditional probabilities for sample 23000 / 104184
[t-SNE] Computed conditional probabilities for sample 24000 / 104184
[t-SNE] Computed conditional probabilities for sample 25000 / 104184
[t-SNE] Computed conditional probabilities for sample 26000 / 104184
[t-SNE] Computed conditional probabilities for sample 27000 / 104184
[t-SNE] Computed conditional probabilities for sample 28000 / 104184
[t-SNE] Computed conditional probabilities for sample 29000 / 104184
[t-SNE] Computed conditional probabilities for sample 30000 / 104184
[t-SNE] Computed conditional probabilities for sample 31000 / 104184
[t-SNE] Computed conditional probabilities for sample 32000 / 104184
[t-SNE] Computed conditional probabilities for sample 33000 / 104184
[t-SNE] Computed conditional probabilities for sample 34000 / 104184
[t-SNE] Computed conditional probabilities for sample 35000 / 104184
[t-SNE] Computed conditional probabilities for sample 36000 / 104184
[t-SNE] Computed conditional probabilities for sample 37000 / 104184
[t-SNE] Computed conditional probabilities for sample 38000 / 104184
[t-SNE] Computed conditional probabilities for sample 39000 / 104184
[t-SNE] Computed conditional probabilities for sample 40000 / 104184
[t-SNE] Computed conditional probabilities for sample 41000 / 104184
[t-SNE] Computed conditional probabilities for sample 42000 / 104184
[t-SNE] Computed conditional probabilities for sample 43000 / 104184
[t-SNE] Computed conditional probabilities for sample 44000 / 104184
[t-SNE] Computed conditional probabilities for sample 45000 / 104184
[t-SNE] Computed conditional probabilities for sample 46000 / 104184
[t-SNE] Computed conditional probabilities for sample 47000 / 104184
[t-SNE] Computed conditional probabilities for sample 48000 / 104184
[t-SNE] Computed conditional probabilities for sample 49000 / 104184
```

```
[t-SNE] Computed conditional probabilities for sample 50000 / 104184
[t-SNE] Computed conditional probabilities for sample 51000 / 104184
[t-SNE] Computed conditional probabilities for sample 52000 / 104184
[t-SNE] Computed conditional probabilities for sample 53000 / 104184
[t-SNE] Computed conditional probabilities for sample 54000 / 104184
[t-SNE] Computed conditional probabilities for sample 55000 / 104184
[t-SNE] Computed conditional probabilities for sample 56000 / 104184
[t-SNE] Computed conditional probabilities for sample 57000 / 104184
[t-SNE] Computed conditional probabilities for sample 58000 / 104184
[t-SNE] Computed conditional probabilities for sample 59000 / 104184
[t-SNE] Computed conditional probabilities for sample 60000 / 104184
[t-SNE] Computed conditional probabilities for sample 61000 / 104184
[t-SNE] Computed conditional probabilities for sample 62000 / 104184
[t-SNE] Computed conditional probabilities for sample 63000 / 104184
[t-SNE] Computed conditional probabilities for sample 64000 / 104184
[t-SNE] Computed conditional probabilities for sample 65000 / 104184
[t-SNE] Computed conditional probabilities for sample 66000 / 104184
[t-SNE] Computed conditional probabilities for sample 67000 / 104184
[t-SNE] Computed conditional probabilities for sample 68000 / 104184
[t-SNE] Computed conditional probabilities for sample 69000 / 104184
[t-SNE] Computed conditional probabilities for sample 70000 / 104184
[t-SNE] Computed conditional probabilities for sample 71000 / 104184
[t-SNE] Computed conditional probabilities for sample 72000 / 104184
[t-SNE] Computed conditional probabilities for sample 73000 / 104184
[t-SNE] Computed conditional probabilities for sample 74000 / 104184
[t-SNE] Computed conditional probabilities for sample 75000 / 104184
[t-SNE] Computed conditional probabilities for sample 76000 / 104184
[t-SNE] Computed conditional probabilities for sample 77000 / 104184
[t-SNE] Computed conditional probabilities for sample 78000 / 104184
[t-SNE] Computed conditional probabilities for sample 79000 / 104184
[t-SNE] Computed conditional probabilities for sample 80000 / 104184
[t-SNE] Computed conditional probabilities for sample 81000 / 104184
[t-SNE] Computed conditional probabilities for sample 82000 / 104184
[t-SNE] Computed conditional probabilities for sample 83000 / 104184
[t-SNE] Computed conditional probabilities for sample 84000 / 104184
[t-SNE] Computed conditional probabilities for sample 85000 / 104184
[t-SNE] Computed conditional probabilities for sample 86000 / 104184
[t-SNE] Computed conditional probabilities for sample 87000 / 104184
[t-SNE] Computed conditional probabilities for sample 88000 / 104184
[t-SNE] Computed conditional probabilities for sample 89000 / 104184
[t-SNE] Computed conditional probabilities for sample 90000 / 104184
[t-SNE] Computed conditional probabilities for sample 91000 / 104184
[t-SNE] Computed conditional probabilities for sample 92000 / 104184
[t-SNE] Computed conditional probabilities for sample 93000 / 104184
[t-SNE] Computed conditional probabilities for sample 94000 / 104184
[t-SNE] Computed conditional probabilities for sample 95000 / 104184
[t-SNE] Computed conditional probabilities for sample 96000 / 104184
[t-SNE] Computed conditional probabilities for sample 97000 / 104184
[t-SNE] Computed conditional probabilities for sample 98000 / 104184
```

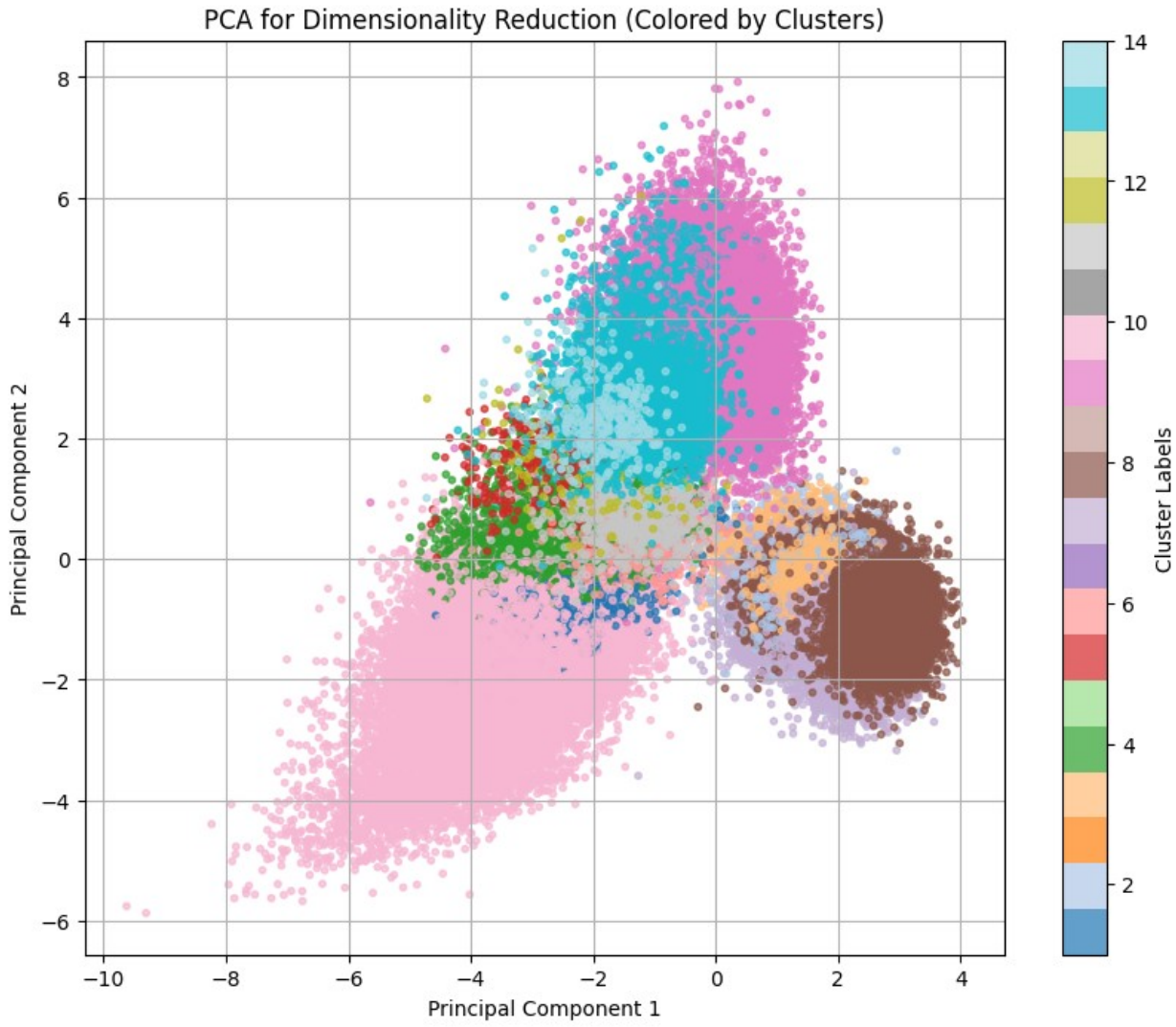
```
[t-SNE] Computed conditional probabilities for sample 99000 / 104184
[t-SNE] Computed conditional probabilities for sample 100000 / 104184
[t-SNE] Computed conditional probabilities for sample 101000 / 104184
[t-SNE] Computed conditional probabilities for sample 102000 / 104184
[t-SNE] Computed conditional probabilities for sample 103000 / 104184
[t-SNE] Computed conditional probabilities for sample 104000 / 104184
[t-SNE] Computed conditional probabilities for sample 104184 / 104184
[t-SNE] Mean sigma: 1.002555
[t-SNE] KL divergence after 250 iterations with early exaggeration:
95.079407
```

2D plot for PCA

```python
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import numpy as np
print(df.columns)

columns_to_drop = ['Event', 'Time', 'Cell_length', 'file_number',
'event_number', 'label', 'individual']

# Drop the columns, ensuring errors='ignore' to avoid KeyError if a
column is not found
filtered_df = df.drop(columns=columns_to_drop, errors='ignore')
labels = df['label']  # Assuming you have a 'label' column indicating
clusters

imputer = SimpleImputer(strategy='mean')
filled_df = imputer.fit_transform(filtered_df)

scaler = StandardScaler()
scaled_data = scaler.fit_transform(filled_df)

# Perform PCA
pca = PCA(n_components=4)  # Set to 4 components for demonstration
pca_transformed = pca.fit_transform(scaled_data)

explained_variance = pca.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance)
std_dev = np.sqrt(pca.explained_variance_)

# Create a summary table for PCA results
pca_summary = pd.DataFrame({
    'Standard deviation': std_dev,
    'Proportion of Variance': explained_variance,
    'Cumulative Proportion': cumulative_variance
}, index=[f'PC{i+1}' for i in range(len(std_dev))])
```

```
print(pca_summary)

# Scatter plot for the first two components, with different colors for
different clusters
plt.figure(figsize=(10, 8))
scatter = plt.scatter(pca_transformed[:, 0], pca_transformed[:, 1],
c=labels, cmap='tab20', s=10, alpha=0.7)
plt.colorbar(scatter, label='Cluster Labels')
plt.title('PCA for Dimensionality Reduction (Colored by Clusters)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.grid(True)
plt.show()
```

```
Index(['Event', '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')
     Standard deviation  Proportion of Variance  Cumulative Proportion
PC1            2.327669                0.154801               0.154801
PC2            1.957437                0.109473               0.264273
PC3            1.877982                0.100766               0.365039
PC4            1.606712                0.073758               0.438797
```



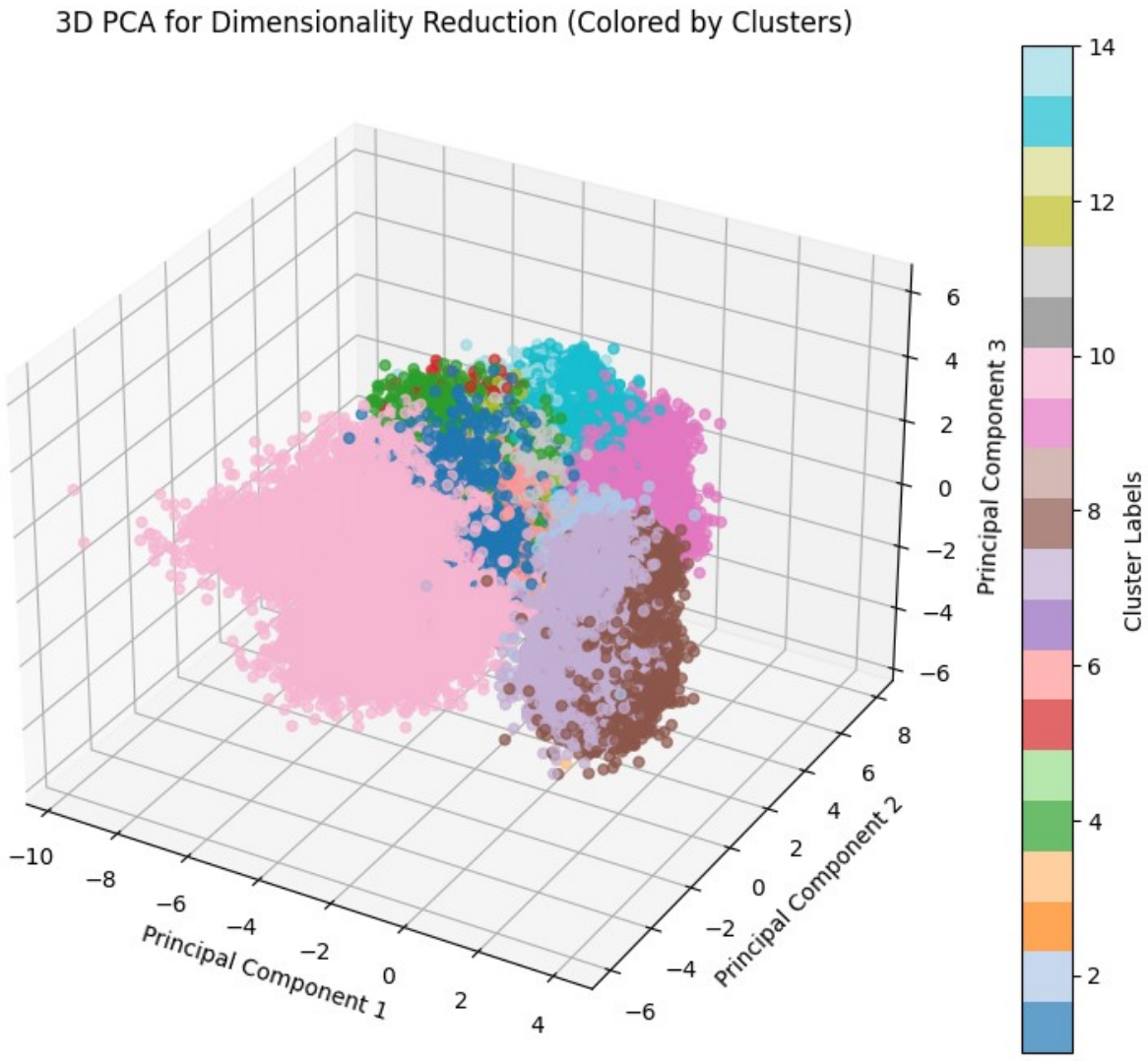
3D plot for PCA

```
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

# Assuming df is already loaded and 'label' column exists for cluster
labeling

# Get actual column names from your DataFrame
actual_columns = df.columns.tolist()
print(f"Actual columns in DataFrame: {actual_columns}")
```

```
# Define columns to drop, ensuring they are present in the DataFrame
columns_to_drop = ['Event', 'Time', 'Cell_length', 'file_number',
'event_number', 'label', 'individual']
# Replace with actual column names if different, e.g.,
# columns_to_drop = ['event', 'time', 'cell_length', 'file_number',
'event_number', 'label', 'individual']

# Filter out columns that are not in the DataFrame
columns_to_drop = [col for col in columns_to_drop if col in
actual_columns]

# Drop the columns, errors='ignore' is no longer needed since we've
filtered
filtered_df = df.drop(columns=columns_to_drop)
labels = df['label']

imputer = SimpleImputer(strategy='mean')
filled_df = imputer.fit_transform(filtered_df)

scaler = StandardScaler()
scaled_data = scaler.fit_transform(filled_df)

# Perform PCA for 4 components
pca = PCA(n_components=4)
pca_transformed = pca.fit_transform(scaled_data)

explained_variance = pca.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance)
std_dev = np.sqrt(pca.explained_variance_)

# Create a summary table for PCA results
pca_summary = pd.DataFrame({
    'Standard deviation': std_dev,
    'Proportion of Variance': explained_variance,
    'Cumulative Proportion': cumulative_variance
}, index=[f'PC{i+1}' for i in range(len(std_dev))])

print(pca_summary)
# 3D scatter plot for the first three components, with different
colors for different clusters
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

scatter = ax.scatter(pca_transformed[:, 0], pca_transformed[:, 1],
pca_transformed[:, 2],
                     c=labels, cmap='tab20', s=20, alpha=0.7)

# Add labels and colorbar
ax.set_title('3D PCA for Dimensionality Reduction (Colored by
Clusters)')
```

```
ax.set_xlabel('Principal Component 1')
ax.set_ylabel('Principal Component 2')
ax.set_zlabel('Principal Component 3')
fig.colorbar(scatter, ax=ax, label='Cluster Labels')

plt.show()
```

```
Actual columns in DataFrame: ['Event', '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']
     Standard deviation  Proportion of Variance  Cumulative Proportion
PC1            2.327669                0.154801               0.154801
PC2            1.957437                0.109473               0.264273
PC3            1.877982                0.100766               0.365039
PC4            1.606712                0.073758               0.438797
```

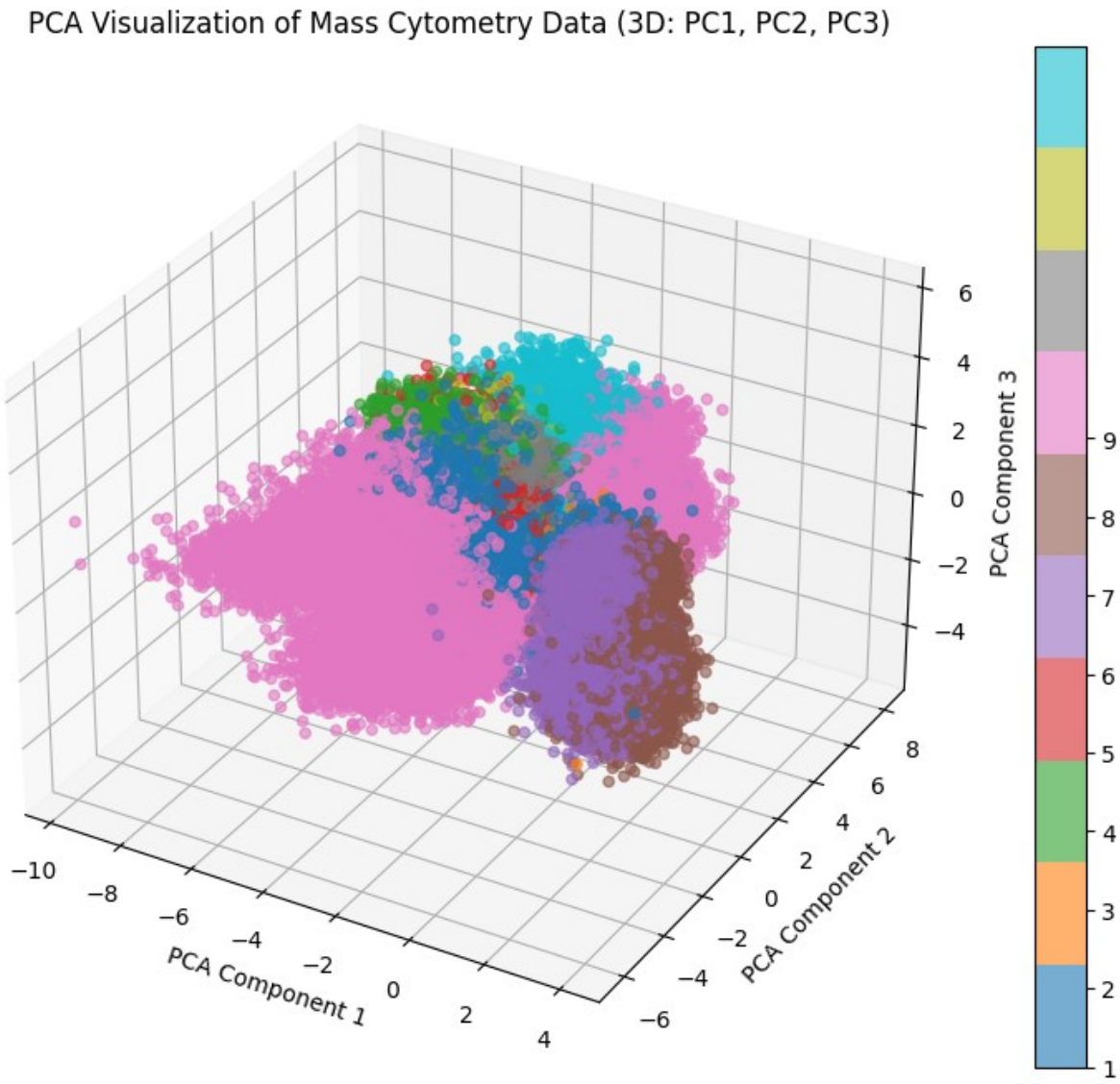



```
from sklearn.preprocessing import StandardScaler
import pandas as pd

# Example data (replace this with your mass cytometry dataset)
# Assuming 'features' is a DataFrame containing your mass cytometry
features
features = pd.DataFrame(df)

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit the scaler to the data and transform it
scaled_data = scaler.fit_transform(features)

# Convert back to a DataFrame if needed for further use
scaled_features = pd.DataFrame(scaled_data, columns=features.columns)
```

```
# Now you can use 'scaled_features' for t-SNE or other analyses



import pandas as pd
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D  # For 3D plotting

# Columns to exclude from the analysis
exclude_columns = ['Event', 'Time', 'file_number', 'event_number',
'label', 'individual']

# Dropping the columns to exclude
data_filtered = df.drop(exclude_columns, axis=1)

# Scaling the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data_filtered)

# Apply PCA with 4 components
pca = PCA(n_components=4)
pca_results = pca.fit_transform(scaled_data)

# Adding PCA results to the original dataframe
df['PCA1'] = pca_results[:, 0]
df['PCA2'] = pca_results[:, 1]
df['PCA3'] = pca_results[:, 2]
df['PCA4'] = pca_results[:, 3]

# Print the PCA results (Standard deviation, Proportion of variance,
and Cumulative Proportion)
explained_variance = pca.explained_variance_ratio_
cumulative_variance = explained_variance.cumsum()
standard_deviation = pca.singular_values_ / np.sqrt(len(data_filtered)
- 1)

print(f"Standard deviation: {standard_deviation}")
print(f"Proportion of Variance: {explained_variance}")
print(f"Cumulative Proportion: {cumulative_variance}")

# Plotting the PCA results in 3D (PC1, PC2, PC3)
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# 3D scatter plot
```

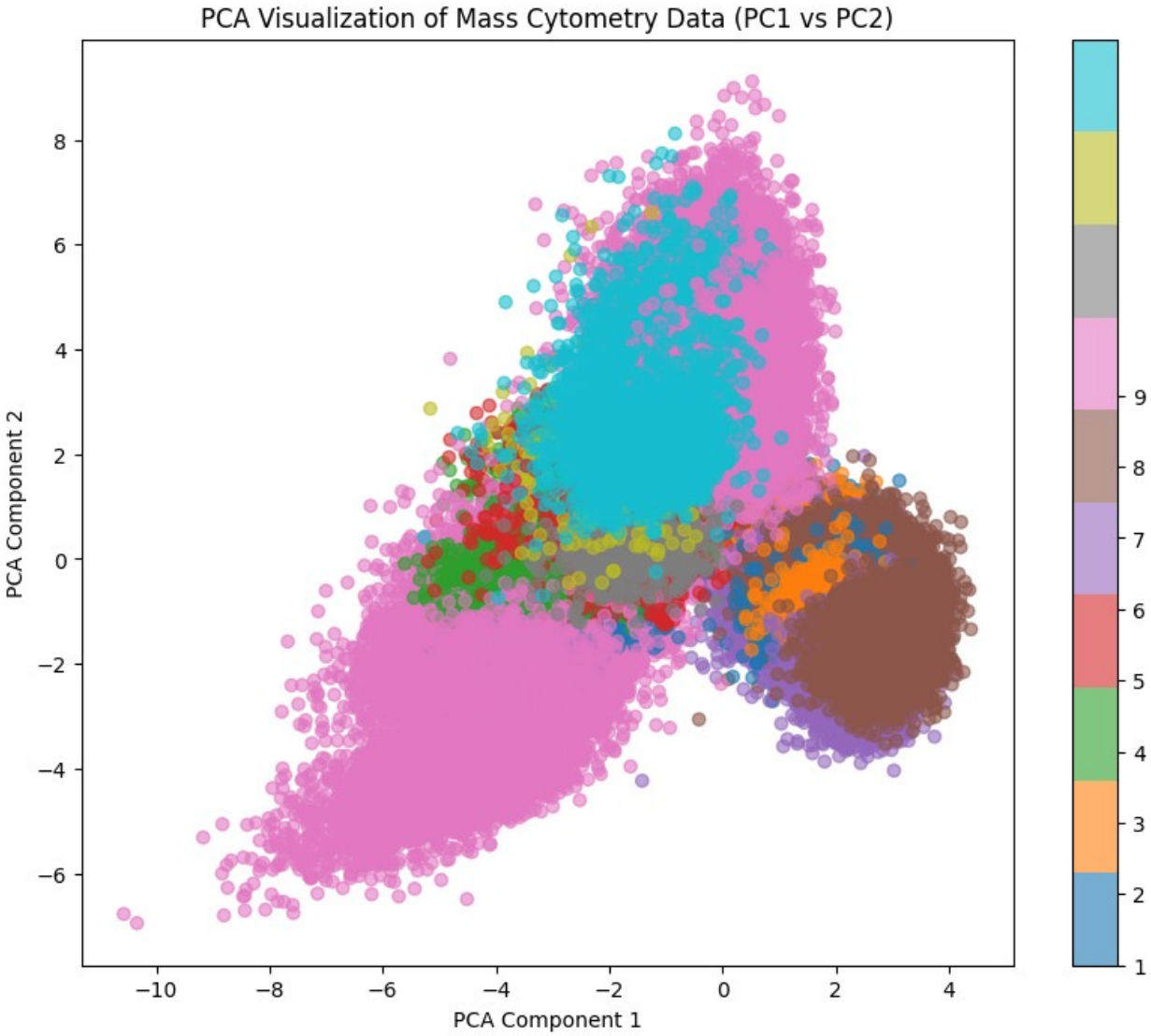
```
scatter = ax.scatter(df['PCA1'], df['PCA2'], df['PCA3'],
c=df['label'], cmap='tab10', alpha=0.6)

# Add labels and title
ax.set_title('PCA Visualization of Mass Cytometry Data (3D: PC1, PC2,
PC3)')
ax.set_xlabel('PCA Component 1')
ax.set_ylabel('PCA Component 2')
ax.set_zlabel('PCA Component 3')

# Adding color bar
plt.colorbar(scatter, ticks=range(10))

plt.show()
```

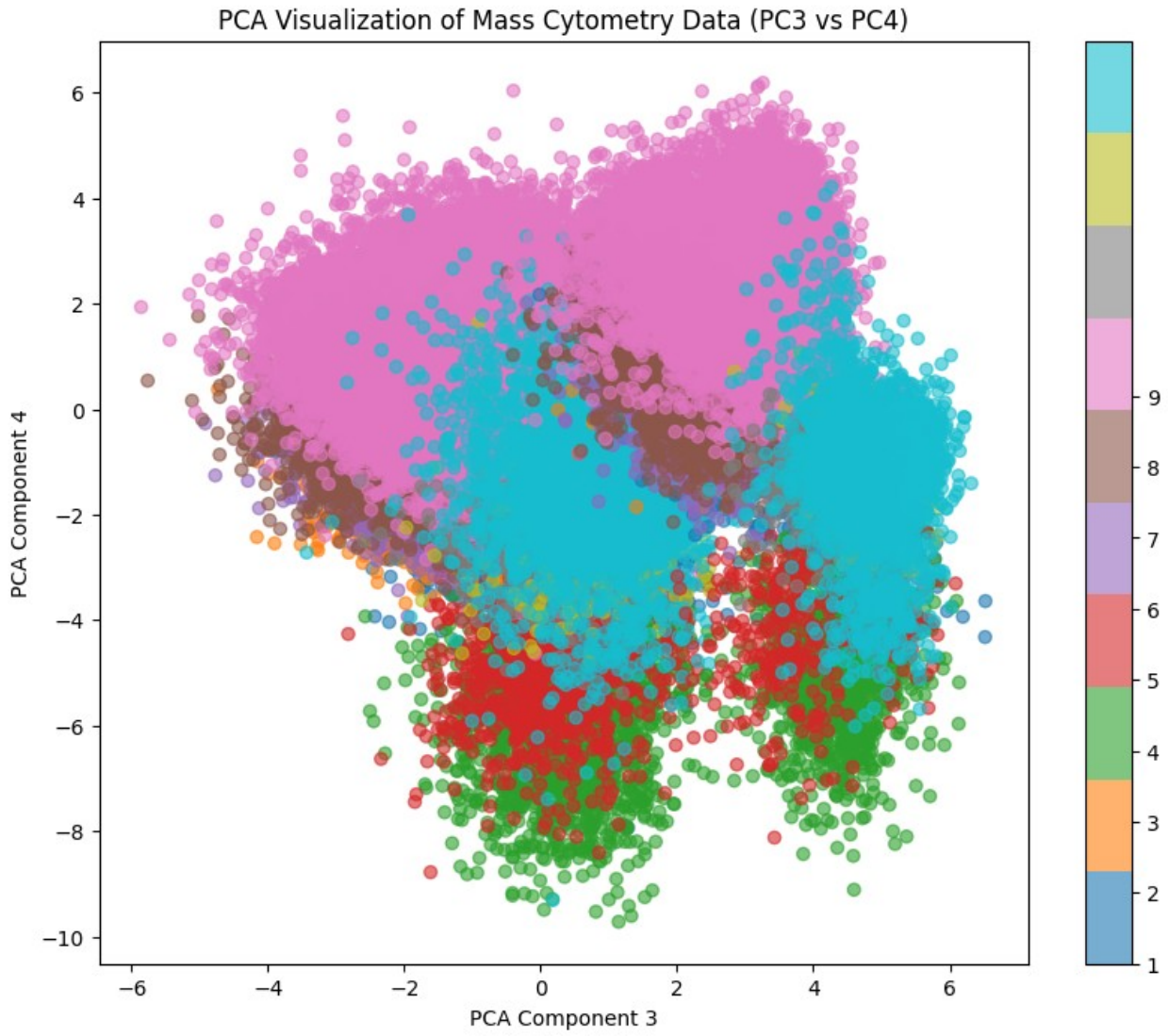
```
Standard deviation: [2.33033605 1.96234686 1.89526477 1.61107568]
Proportion of Variance: [0.15084571 0.10696641 0.09977819 0.07209875]
Cumulative Proportion: [0.15084571 0.25781212 0.35759032 0.42968907]
```




```
import pandas as pd
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Columns to exclude from the analysis
exclude_columns = ['Event', 'Time', 'file_number', 'event_number',
'label', 'individual']

# Dropping the columns to exclude
df_filtered = df.drop(exclude_columns, axis=1)

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

```
scaled_data = scaler.fit_transform(df_filtered)

# Apply PCA with 4 components
pca = PCA(n_components=4)
pca_results = pca.fit_transform(scaled_data)

# Adding PCA results to the original dataframe
df['PCA1'] = pca_results[:, 0]
df['PCA2'] = pca_results[:, 1]
df['PCA3'] = pca_results[:, 2]
df['PCA4'] = pca_results[:, 3]

# Print the PCA results (Standard deviation, Proportion of variance,
and Cumulative Proportion)
explained_variance = pca.explained_variance_ratio_
cumulative_variance = explained_variance.cumsum()
standard_deviation = pca.singular_values_ / np.sqrt(len(df_filtered) -
1)

print(f"Standard deviation: {standard_deviation}")
print(f"Proportion of Variance: {explained_variance}")
print(f"Cumulative Proportion: {cumulative_variance}")

# Plotting the PCA results for the first two principal components
plt.figure(figsize=(10, 8))
scatter = plt.scatter(df['PCA1'], df['PCA2'], c=df['label'],
cmap='tab10', alpha=0.6)
plt.colorbar(scatter, ticks=range(10))
plt.title('PCA Visualization of Mass Cytometry Data (PC1 vs PC2)')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.show()

# Optionally plot PC3 vs PC4
plt.figure(figsize=(10, 8))
scatter = plt.scatter(df['PCA3'], df['PCA4'], c=df['label'],
cmap='tab10', alpha=0.6)
plt.colorbar(scatter, ticks=range(10))
plt.title('PCA Visualization of Mass Cytometry Data (PC3 vs PC4)')
plt.xlabel('PCA Component 3')
plt.ylabel('PCA Component 4')
plt.show()

Standard deviation: [2.53583711 2.20245521 2.14290277 1.89619847]
Proportion of Variance: [0.16076114 0.12126977 0.11480038 0.08988888]
Cumulative Proportion: [0.16076114 0.28203091 0.39683128 0.48672016]
```

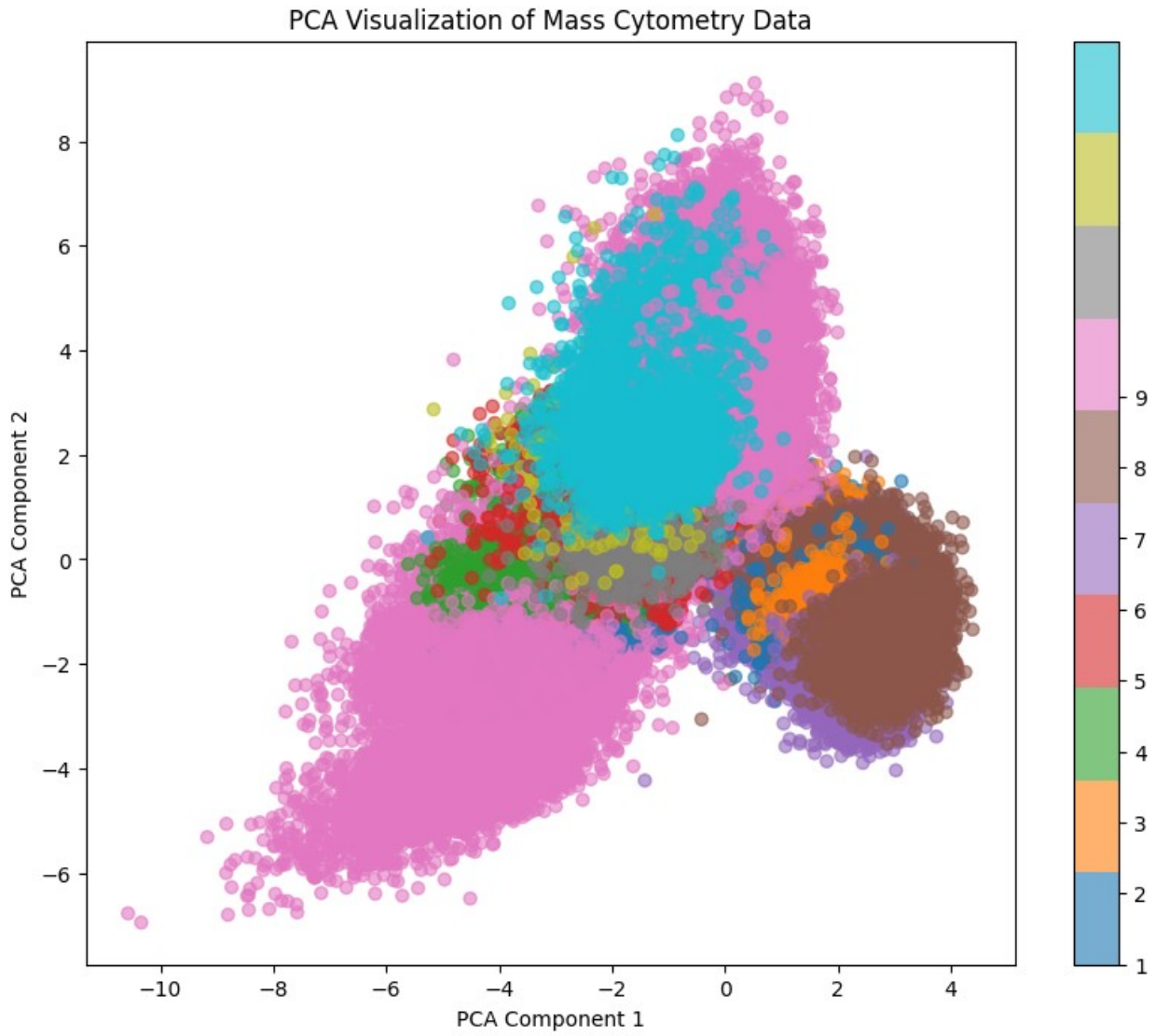

PCA Visualization of Mass Cytometry Data (PC1 vs PC2)
PCA Component 2
PCA Component 1
8
6
4
2
0
−2
−4
−6
−10
−8
−6
−4
−2
0
2
4
9
8
7
6
5
4
3
2
1




```
import pandas as pd
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt


print(('PCA for Dimensionality Reduction'))
# Columns to exclude from the analysis
exclude_columns = ['Event', 'Time', 'file_number', 'event_number',
'label', 'individual']

# Dropping the columns to exclude
df_filtered = df.drop(exclude_columns, axis=1)

# Scaling the data
```

```
scaler = StandardScaler()
scaled_data = scaler.fit_transform(df_filtered)

# Apply PCA
pca = PCA(n_components=2)
pca_results = pca.fit_transform(scaled_data)

# Adding PCA results to the original dataframe
df['PCA1'] = pca_results[:, 0]
df['PCA2'] = pca_results[:, 1]

# Plotting the PCA results
plt.figure(figsize=(10, 8))
scatter = plt.scatter(df['PCA1'], df['PCA2'], c=df['label'],
cmap='tab10', alpha=0.6)
plt.colorbar(scatter, ticks=range(10))
plt.title('PCA Visualization of Mass Cytometry Data')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.show()
```

PCA for Dimensionality Reduction




```
import sklearn
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

print(df.columns) # Print available column names

# Replace with actual column names from the printed output
# Carefully check spelling and case sensitivity
columns_to_remove = ['Time', 'Cell_length', 'file_number',
'event_number', 'label', 'individual']

# Use a list comprehension to filter out columns not present in the
DataFrame
valid_columns_to_remove = [col for col in columns_to_remove if col in
```

```
df.columns]

columns_to_remover = df.drop(columns=valid_columns_to_remove)

len(columns_to_remover.columns)
```

```
Index(['Event', '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')
```

```
36
```

```
import tensorflow as tf
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import numpy as np

# Load and preprocess MNIST data
(train_images, train_labels), (test_images, test_labels) =
tf.keras.datasets.mnist.load_data()
train_images = train_images.astype('float32') / 255.0
test_images = test_images.astype('float32') / 255.0

# Select a subset of data for t-SNE
n_samples = 1000
train_images_flat = train_images[:n_samples].reshape(n_samples, -1)
train_labels_subset = train_labels[:n_samples] # Defining
train_labels_subset here

features = columns_to_remover
# Instead of accessing df['label'], use the original labels before
dropping columns
labels = train_labels_subset
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

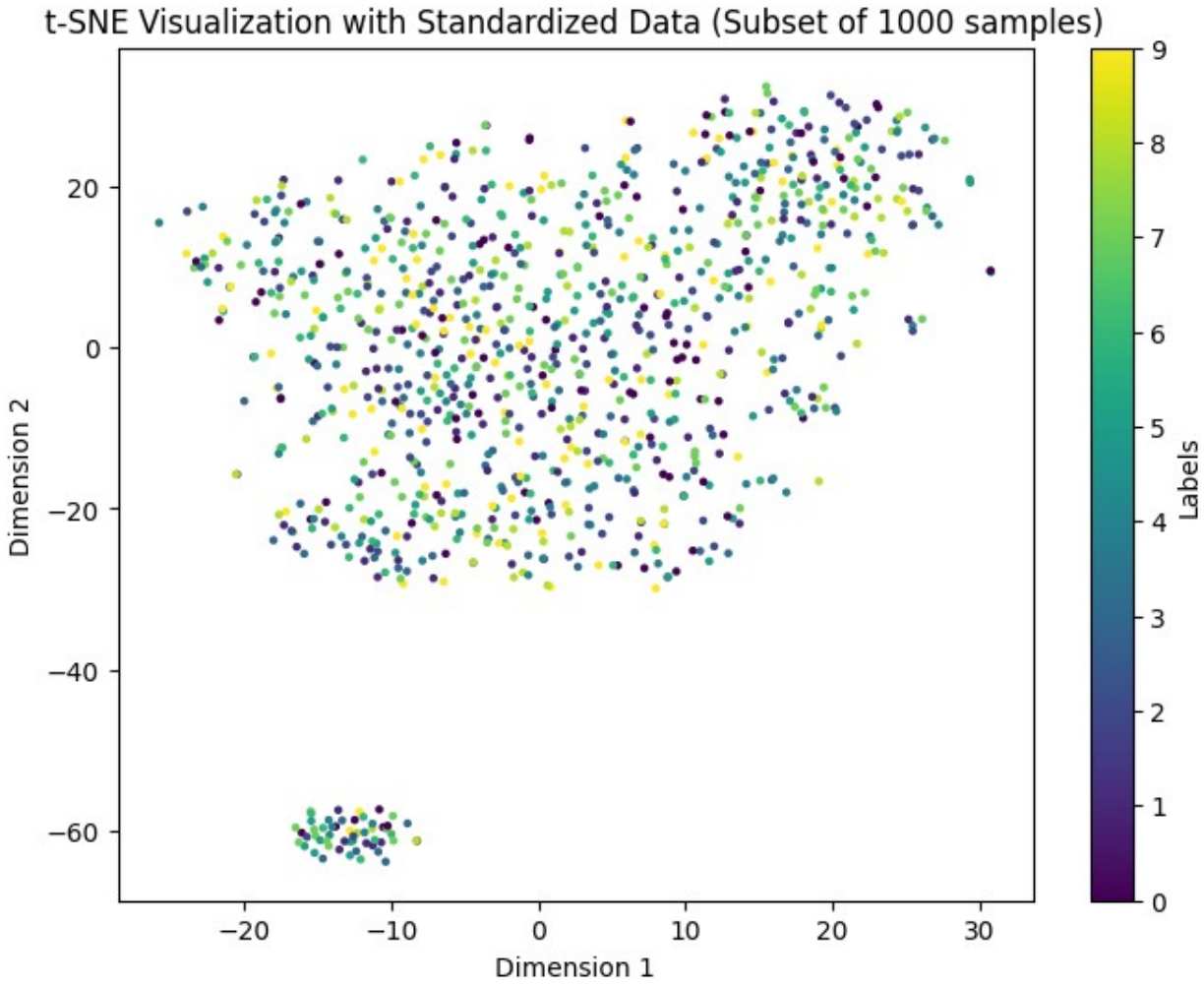
t-SNE plot for 1000 samples

```
import numpy as np
import pandas as pd
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer  # Import SimpleImputer

# Convert all columns to numeric, coercing errors to NaN
features = features.apply(pd.to_numeric, errors='coerce')

# Impute missing values using SimpleImputer
imputer = SimpleImputer(strategy='mean')  # Replace 'mean' with your
desired strategy
features_imputed = imputer.fit_transform(features)

scaler = StandardScaler()
features_standardized = scaler.fit_transform(features_imputed)

# Updated: n_samples should match the size of labels
n_samples = min(len(features_standardized), len(labels))  # Use the
minimum size

# Generate subset_indices within the bounds of both features and
labels
subset_indices = np.random.choice(n_samples, n_samples, replace=False)

subset_features = features_standardized[subset_indices]
subset_labels = labels[subset_indices]

tsne = TSNE(n_components=2, random_state=42, perplexity=30,
n_iter=1000, verbose=1)
tsne_result = tsne.fit_transform(subset_features)

plt.figure(figsize=(8, 6))
scatter = plt.scatter(tsne_result[:, 0], tsne_result[:, 1],
c=subset_labels, cmap='viridis', s=5)
plt.title('t-SNE Visualization with Standardized Data (Subset of {}
samples)'.format(n_samples))
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')

plt.colorbar(scatter, label='Labels')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/manifold/
_t_sne.py:1162: FutureWarning: 'n_iter' was renamed to 'max_iter' in
version 1.5 and will be removed in 1.7.
  warnings.warn(

[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 1000 samples in 0.001s...
[t-SNE] Computed neighbors for 1000 samples in 0.034s...
[t-SNE] Computed conditional probabilities for sample 1000 / 1000
[t-SNE] Mean sigma: 1.314912
[t-SNE] KL divergence after 250 iterations with early exaggeration:
```

```
67.653954
[t-SNE] KL divergence after 1000 iterations: 1.567945
```



-------***AUTOENDOER***----------

Binary Mask

```
import numpy as np
import pandas as pd
# Set a random seed for reproducibility
np.random.seed(42)
# Create a sample DataFrame called 'demodata' for demonstration
demodata = pd.DataFrame({
 'column1': [5, 12, 18, 7],
 'column2': [10, 20, 15, 30],
 'column3': [25, 35, 40, 45]
 })
# Define the probability of masking (e.g., 0.3 means a 30% chance each
eleme
```

```
p_m = 0.3
# Convert 'demodata' to a NumPy array for masking
data_array = demodata.values
# Generate a binary mask based on the probability, where 1 = not
masked, 0 =
mask = np.random.binomial(1, 1 - p_m, data_array.shape) # Reverse
probabili
# Convert to a DataFrame for easier analysis
binary_mask_df = pd.DataFrame(mask, columns=demodata.columns)
print("Original DataFrame:\n", demodata)
print("\nBinary Mask DataFrame:\n", binary_mask_df)
```

```
Original DataFrame:
    column1  column2  column3
0         5       10       25
1        12       20       35
2        18       15       40
3         7       30       45

Binary Mask DataFrame:
    column1  column2  column3
0         1        0        0
1         1        1        1
2         1        0        1
3         0        1        0
```

Randomly Shuffle

```
import numpy as np
import pandas as pd
# Create a sample DataFrame called 'demodata' for demonstration
demodata = pd.DataFrame({
 'column1': [5, 12, 18, 7],
 'column2': [10, 20, 15, 30],
 'column3': [25, 35, 40, 45]
})
# Shuffle each column in the DataFrame independently
shuffled_demodata = demodata.apply(lambda col:
np.random.permutation(col))
print("Original DataFrame:\n", demodata)
print("\nShuffled DataFrame:\n", shuffled_demodata)
```

```
Original DataFrame:
    column1  column2  column3
0         5       10       25
1        12       20       35
2        18       15       40
3         7       30       45

Shuffled DataFrame:
```

```
    column1  column2  column3
0         5       30       25
1        18       15       40
2        12       10       35
3         7       20       45
```

Corrupted DataFrame

Formula = (x.values * (1 - m) + x_shuffled.values * m)

```
import numpy as np
import pandas as pd
# Create a sample DataFrame called 'x' (original data)
x = pd.DataFrame({
 'column1': [5, 12, 18, 7],
 'column2': [10, 20, 15, 30],
 'column3': [25, 35, 40, 45]
})
# Define the probability of masking (e.g., 0.3 means a 30% chance each
eleme
p_m = 0.3
# Generate a binary mask matrix 'm'
m = np.random.binomial(1, 1 - p_m, x.shape)
binary_mask_df = pd.DataFrame(m, columns=x.columns)
# Shuffle each column in 'x' independently to create 'x_shuffled'
x_shuffled = x.apply(lambda col: np.random.permutation(col))
# Calculate the corrupted DataFrame 'x_corrupted' using the formula
x_corrupted_array = x.values * (1 - m) + x_shuffled.values * m
x_corrupted = pd.DataFrame(x_corrupted_array, columns=x.columns)
# Display results
print("Original DataFrame (x):\n", x)
print("\nBinary Mask DataFrame (m):\n", binary_mask_df)
print("\nShuffled DataFrame (x_shuffled):\n", x_shuffled)
print("\nCorrupted DataFrame (x_corrupted):\n", x_corrupted)
```

```
Original DataFrame (x):
    column1  column2  column3
0         5       10       25
1        12       20       35
2        18       15       40
3         7       30       45

Binary Mask DataFrame (m):
    column1  column2  column3
0         1        1        1
1         1        1        0
2         1        1        1
3         1        0        1

Shuffled DataFrame (x_shuffled):
```

```
    column1  column2  column3
0        12       20       40
1         7       15       45
2        18       30       25
3         5       10       35

Corrupted DataFrame (x_corrupted):
    column1  column2  column3
0        12       20       40
1         7       15       35
2        18       30       25
3         5       30       35
```

Applying Binary Mask, Shuffled Output and Corrupted DataFrame on Original Data

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Load the dataset
data = df

# Exclude the specified columns

data_filtered = data.drop(columns=new_df)

# Set the probability of masking
p_m = 0.3

# Generate a binary mask matrix 'm'
m = np.random.binomial(1, 1 - p_m, data_filtered.shape)
binary_mask_df = pd.DataFrame(m, columns=data_filtered.columns)

# Shuffle each column in 'data_filtered' independently to create
'data_shuffled'
data_shuffled = data_filtered.apply(lambda col:
np.random.permutation(col))

# Calculate the corrupted DataFrame 'data_corrupted' using the formula
data_corrupted_array = data_filtered.values * (1 - m) +
data_shuffled.values * m
data_corrupted = pd.DataFrame(data_corrupted_array,
columns=data_filtered.columns)

# Display results
print("Binary Mask DataFrame (m):\n", binary_mask_df)
print("\nShuffled DataFrame (data_shuffled):\n", data_shuffled)
print("\nCorrupted DataFrame (data_corrupted):\n", data_corrupted)
```

```
Binary Mask DataFrame (m):
         Event  Time  file_number  event_number  individual
0            1     1            1             1           1
1            1     1            0             1           1
2            1     0            1             1           1
3            1     0            1             1           0
4            0     1            1             1           0
...        ...   ...          ...           ...         ...
265622       0     0            0             1           1
265623       1     1            1             1           0
265624       0     1            1             1           1
265625       1     1            1             1           1
265626       1     1            1             1           0

[265627 rows x 5 columns]

Shuffled DataFrame (data_shuffled):
           Event       Time  file_number  event_number  individual
0       193588  225626.00     3.627711        321215           1
1       149022  553631.00     3.669327         51538           1
2       196668  575854.44     3.627711        311248           1
3       131479  308878.00     3.627711         60445           2
4        27957  519915.00     3.627711         34692           1
...        ...         ...          ...           ...         ...
265622  230887  180450.00     3.627711        246388           1
265623  217233  211312.00     3.669327        324495           2
265624   22371  101370.00     3.627711         51597           1
265625  112878  509820.44     3.669327        152233           1
265626   59081  291265.00     3.627711        247506           1

[265627 rows x 5 columns]

Corrupted DataFrame (data_corrupted):
             Event       Time  file_number  event_number  individual
0       193588.0  225626.00     3.627711      321215.0         1.0
1       149022.0  553631.00     3.627711       51538.0         1.0
2       196668.0    7015.00     3.627711      311248.0         1.0
3       131479.0    7099.00     3.627711       60445.0         1.0
4            5.0  519915.00     3.627711       34692.0         1.0
...          ...         ...          ...           ...         ...
265622  265623.0  707951.44     3.669327      246388.0         1.0
265623  217233.0  211312.00     3.669327      324495.0         2.0
265624  265625.0  101370.00     3.627711       51597.0         1.0
265625  112878.0  509820.44     3.669327      152233.0         1.0
265626   59081.0  291265.00     3.627711      247506.0         2.0

[265627 rows x 5 columns]
```

New Mask

Formula = (mask_new = 1 * (data_filtered != data_corrupted))

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Load the dataset
data = df

# Exclude the specified columns

# data_filtered = data.drop(columns=new_df)
data_filtered = df

# Set the probability of masking
p_m = 0.3

# Generate a binary mask matrix 'm' (changes every run)
m = np.random.binomial(1, 1 - p_m, data_filtered.shape)
binary_mask_df = pd.DataFrame(m, columns=data_filtered.columns)

# Shuffle each column in 'data_filtered' independently to create
'data_shuffled'
data_shuffled = data_filtered.apply(lambda col:
np.random.permutation(col))

# Calculate the corrupted DataFrame 'data_corrupted' using the formula
data_corrupted_array = data_filtered.values * (1 - m) +
data_shuffled.values * m
data_corrupted = pd.DataFrame(data_corrupted_array,
columns=data_filtered.columns)

# Generate mask_new to indicate differences between original and
corrupted data
mask_new = 1 * (data_filtered != data_corrupted)

# Print only the new mask matrix
print("New Mask Matrix (mask_new):\n", mask_new)
```

```
New Mask Matrix (mask_new):
         Event  Time  Cell_length  DNA1  DNA2  CD45RA  CD133  CD19
CD22  \
0            0     0            1     1     0       0      1     1
1
1            0     1            0     0     0       1      0     1
1
2            1     1            0     1     0       1      1     1
1
3            1     1            1     1     1       1      0     0
1
4            1     1            1     1     1       1      1     1
```

```
0
...       ...   ...          ...   ...   ...     ...    ...   ...   ..
.
265622      0     1            0     0     0       1      1     1
1
265623      1     1            0     1     1       1      0     0
1
265624      0     0            0     0     0       0      1     1
0
265625      0     1            0     1     1       1      1     0
1
265626      1     0            1     1     0       1      1     1
0

        CD11b  ...  CD117  CD49d  HLA-DR  CD64  CD41  Viability
file_number  \
0           1  ...      0      1       0     1     0          1
0
1           0  ...      1      1       1     1     0          0
1
2           0  ...      1      0       1     1     1          0
0
3           1  ...      1      1       1     0     0          1
0
4           1  ...      0      1       0     1     0          1
1
...       ...  ...    ...    ...     ...   ...   ...        ...
...
265622      0  ...      1      1       0     1     1          1
1
265623      1  ...      1      1       1     0     1          0
1
265624      1  ...      0      1       1     0     1          1
1
265625      1  ...      1      1       1     1     1          1
0
265626      1  ...      0      1       1     0     0          0
1

        event_number  label  individual
0                  1      1           1
1                  1      0           0
2                  1      1           0
3                  0      1           0
4                  0      1           0
...              ...    ...         ...
265622             0      1           1
265623             0      1           1
265624             1      1           0
```

```
265625              0      1           0
265626              1      1           0

[265627 rows x 42 columns]
```

Split features and labels for unlabeled data

```
import numpy as np
import pandas as pd
# Define the target column used for labeling
label_column = 'label'
# Separate labeled and unlabeled data using label_df
label_df = df[df[label_column].notnull()] # labeled data
unlabeled_df = df[df[label_column].isnull()] # unlabeled data
# Split features and labels for labeled data
x_labeled = label_df.drop(columns=[label_column])
y_labeled = label_df[label_column]
# Split features and labels for unlabeled data
x_unlabeled = unlabeled_df.drop(columns=[label_column])
y_unlabeled = unlabeled_df[label_column]
# Display results
print("Labeled Features (x_labeled):\n", x_labeled)
print("\nLabeled Labels (y_labeled):\n", y_labeled)
print("\nUnlabeled Features (x_unlabeled):\n", x_unlabeled)
print("\nUnlabeled Labels (y_unlabeled):\n", y_unlabeled)
```

```
Labeled Features (x_labeled):
          Event       Time  Cell_length      DNA1      DNA2    CD45RA
\
0             1    2693.00           22  4.391057  4.617262  0.162691

1             2    3736.00           35  4.340481  4.816692  0.701349

2             3    7015.00           32  3.838727  4.386369  0.603568

3             4    7099.00           29  4.255806  4.830048  0.433747

4             5    7700.00           25  3.976909  4.506433 -0.008809

...         ...         ...          ...       ...       ...       ...

104179   104180  641812.44           58  6.827981  7.249403 -0.000106

104180   104181  653387.44           55  6.683204  7.166172  0.692668

104181   104182  671024.44           40  6.911546  7.152603 -0.036795

104182   104183  680006.44           48  6.700332  7.100771  0.308817

104183   104184  687494.44           64  6.559460  7.080928  0.519572
```

```
           CD133      CD19      CD22     CD11b  ...      CD61
CD117  \
0      -0.029585 -0.006696  0.066388 -0.009184  ... -0.002936
0.053050
1      -0.038280 -0.016654  0.074409  0.808031  ...  1.258437
0.089660
2      -0.032216  0.073855 -0.042977 -0.001881  ...  0.257137
0.046222
3      -0.027611 -0.017661 -0.044072  0.733698  ... -0.041140
0.066470
4      -0.030297  0.080423  0.495791  1.107627  ...  0.168609 -
0.006223
...          ...       ...       ...       ...  ...       ...       ..
.
104179 -0.030641  1.432347 -0.044946 -0.016534  ...  0.188846 -
0.002144
104180 -0.037335  1.639063  0.286325 -0.036985  ... -0.029213 -
0.031301
104181 -0.014477  1.637975 -0.021794 -0.020169  ... -0.015220 -
0.034755
104182  0.075762  1.455129  0.042576 -0.049737  ... -0.016644 -
0.047522
104183  0.097257  1.346523  0.279473 -0.021585  ... -0.051973 -
0.017015

           CD49d    HLA-DR      CD64      CD41  Viability  file_number
\
0       0.853505  1.664480 -0.005376 -0.001961   0.648429     3.627711

1       0.197818  0.491592  0.144814  0.868014   0.561384     3.627711

2       2.586670  1.308337 -0.010961 -0.010413   0.643337     3.627711

3       1.338669  0.140523 -0.013449 -0.026039  -0.026523     3.627711

4       0.180924  0.197332  0.076167 -0.040488   0.283287     3.627711

...          ...       ...       ...       ...        ...          ...

104179  1.115652  2.373524 -0.004620 -0.051592   0.157816     3.669327

104180  1.653418  4.367032  0.062683  0.158656   0.025255     3.669327

104181  1.083173  3.541526  0.110382  0.108349  -0.043739     3.669327

104182  0.432565  3.882030  0.058852  0.185295   0.204898     3.669327

104183  0.263008  4.332834 -0.017214  0.130106   0.023135     3.669327
```

```
        event_number  individual
0                307           1
1                545           1
2               1726           1
3               1766           1
4               2031           1
...              ...         ...
104179        100344           2
104180        100892           2
104181        101558           2
104182        101842           2
104183        102112           2

[104184 rows x 41 columns]

Labeled Labels (y_labeled):
 0          1.0
1           1.0
2           1.0
3           1.0
4           1.0
           ...
104179     14.0
104180     14.0
104181     14.0
104182     14.0
104183     14.0
Name: label, Length: 104184, dtype: float64

Unlabeled Features (x_unlabeled):
          Event       Time  Cell_length      DNA1      DNA2    CD45RA
\
104184  104185      40.00           25  4.203073  4.837565  0.095543

104185  104186     176.00           34  4.042991  4.808275  0.035310

104186  104187     189.00           37  4.233125  4.922201  0.415954

104187  104188     193.00           26  3.997143  4.685426 -0.038565

104188  104189     204.00           20  4.115830  4.893428  0.177246

...        ...        ...          ...       ...       ...       ...

265622  265623  707951.44           41  6.826629  7.133022  1.474081

265623  265624  708145.44           45  6.787791  7.154026  0.116755

265624  265625  708398.44           41  6.889866  7.141219  0.684921
```

```
265625  265626  708585.44           39  6.865218  7.144353  0.288761

265626  265627  709122.44           41  6.887820  7.127359  0.360753


           CD133      CD19      CD22     CD11b  ...      CD61
CD117  \
104184 -0.027206  0.172384 -0.001950  0.505713  ...  3.029787 -
0.010093
104185 -0.013869 -0.043922 -0.001871  0.180261  ... -0.017628
0.346248
104186  0.412757  0.431715 -0.025619  0.491190  ...  0.000544
0.691393
104187  0.125894  0.191383 -0.026497  0.342190  ... -0.012887
0.033096
104188  0.171916  0.028568 -0.029751  2.480689  ... -0.015719 -
0.043689
...          ...       ...       ...       ...  ...       ...       ..
.
265622 -0.019174 -0.055620 -0.007261  0.063395  ...  0.861068 -
0.011105
265623 -0.056213 -0.008864 -0.035158 -0.041845  ...  0.565170
0.143869
265624 -0.006264 -0.026111 -0.030837 -0.034641  ... -0.008680
0.087102
265625 -0.011310 -0.048786  0.073983 -0.031787  ... -0.029347 -
0.047971
265626  0.128604 -0.006934  0.109846  3.864711  ... -0.023831
0.080195

           CD49d    HLA-DR      CD64      CD41  Viability  file_number
\
104184  0.387121  2.859639  2.709532  1.208795   0.102978     3.627711

104185  0.089940 -0.017702  0.045091 -0.022009   0.092770     3.627711

104186  2.996583  5.812406  1.713608  0.479122   1.888485     3.627711

104187 -0.029722 -0.031126 -0.020739 -0.014693   0.067437     3.627711

104188  0.027586  2.543139  3.323810 -0.002918   0.109243     3.627711

...          ...       ...       ...       ...        ...          ...

265622  0.533736  0.123758 -0.042495 -0.027971   0.236957     3.669327

265623  1.269464  0.047215 -0.008000 -0.025811  -0.003500     3.669327

265624 -0.055912  0.501536  0.053884 -0.042602   0.107206     3.669327
```

```
265625  0.101955  6.200001  0.296877  0.192786   0.620872     3.669327

265626  0.037962  3.675123 -0.000878 -0.052526   0.310466     3.669327


        event_number  individual
104184             1           1
104185             6           1
104186             7           1
104187             8           1
104188             9           1
...              ...         ...
265622        102686           2
265623        102690           2
265624        102701           2
265625        102706           2
265626        102720           2

[161443 rows x 41 columns]

Unlabeled Labels (y_unlabeled):
 104184   NaN
104185   NaN
104186   NaN
104187   NaN
104188   NaN
          ..
265622   NaN
265623   NaN
265624   NaN
265625   NaN
265626   NaN
Name: label, Length: 161443, dtype: float64
```

Split labelled dataset into x_test,x_train and y_test and y_train . train = 70% and test = 30%

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

# Define the target column used for labeling
label_column = 'label'

# Assuming df is defined, separate labeled data
label_df = df[df[label_column].notnull()]

# Split features and labels for labeled data
x_labeled = label_df.drop(columns=[label_column])
y_labeled = label_df[label_column]
```

```python
# Split labeled data into training and testing sets (70%-30% split)
x_train, x_test, y_train, y_test = train_test_split(x_labeled,
y_labeled, test_size=0.3, random_state=42)

# Display results
print("Training Features (x_train):\n", x_train)
print("\nTesting Features (x_test):\n", x_test)
print("\nTraining Labels (y_train):\n", y_train)
print("\nTesting Labels (y_test):\n", y_test)
```

```
Training Features (x_train):
          Event       Time  Cell_length      DNA1      DNA2    CD45RA
\
64113    64114  401196.00           25  3.899656  4.594272  0.976652

82744    82745  502826.44           31  6.592998  6.901888  0.431481

24294    24295  488377.00           41  3.543583  4.467671  0.377192

7820      7821  225689.00           38  4.305227  4.881685  0.199351

43295    43296  153333.00           26  4.159271  4.861015  0.831285

...        ...        ...          ...       ...       ...       ...

54886    54887   93991.00           15  4.074604  4.747052  0.431805

76820    76821   46189.00           33  6.584427  6.882117  0.640424

103694  103695  574005.44           43  6.719895  7.080995  0.306443

860        861  516979.00           26  3.886782  4.886936  0.060176

15795    15796  225860.00           25  3.523293  4.289820  0.646288


           CD133      CD19      CD22     CD11b  ...      CD61
CD117  \
64113   0.302811  0.154761 -0.011676  3.180236  ...  0.051464 -
0.003680
82744  -0.052898 -0.037690 -0.029715 -0.040846  ... -0.036430
0.021689
24294   0.219081  0.245478  0.193328  0.075123  ...  1.003383
0.406137
7820    0.100678 -0.025812 -0.002898  1.437247  ... -0.007282
1.421540
43295   0.191518  2.002712  3.387782  0.179219  ... -0.040754
0.060944
...          ...       ...       ...       ...  ...       ...       ..
.
54886   0.228761 -0.011434 -0.017082  1.379518  ... -0.029607 -
```

```
0.039425
76820  -0.044057 -0.013737 -0.030704 -0.009781  ... -0.038000
0.190509
103694 -0.026339  2.074008  0.052549  0.167479  ...  0.054690
0.011329
860     0.233401 -0.020592 -0.007786  1.090780  ... -0.001868 -
0.046200
15795  -0.028126  0.184879  0.214664  0.224471  ...  0.089666
0.343049

           CD49d    HLA-DR      CD64      CD41  Viability  file_number
\
64113   1.260410  0.700093  2.355886  0.125409   0.840205     3.627711

82744   0.034946 -0.055651 -0.023248 -0.054842  -0.009329     3.669327

24294   1.928676 -0.046849  0.229309  0.937020   1.231347     3.627711

7820    1.443145  2.461705  0.528679  0.072205   0.892480     3.627711

43295   1.294561  3.085858 -0.014128  0.479256   2.269233     3.627711

...          ...       ...       ...       ...        ...          ...

54886   0.036619  2.424191  1.080756 -0.014481   0.190138     3.627711

76820   0.204920 -0.004600  0.135288 -0.042874  -0.023160     3.669327

103694  0.267845  4.060155  0.123218  0.006991  -0.026324     3.669327

860     1.016980  0.000744 -0.030356 -0.033473   0.371143     3.627711

15795   0.784416  0.064465  0.088172 -0.013586   0.153918     3.627711


        event_number  individual
64113         318320           1
82744          80934           2
24294         366690           1
7820          203131           1
43295         152117           1
...              ...         ...
54886          96894           1
76820           8563           2
103694         94148           2
860           378748           1
15795         203230           1

[72928 rows x 41 columns]

Testing Features (x_test):
```

```
        Event      Time  Cell_length      DNA1      DNA2    CD45RA
CD133  \
60544  60545  278003.0           49  3.618797  4.144135  0.198186
0.000282
50673  50674  490341.0           27  3.660988  4.497041  1.272625
0.129642
50682  50683  490912.0           23  3.854865  4.663734  1.527763
0.151383
1761    1762  170466.0           17  3.716473  4.465312  0.375236 -
0.037150
98760  98761  423490.0           32  6.826030  7.007709  0.223441 -
0.048813
...      ...       ...          ...       ...       ...       ...
...
20510  20511  370777.0           63  3.260559  3.934633  0.448954
0.219533
11540  11541   99635.0           37  3.204839  3.422136  0.088893
0.359100
30042  30043  145367.0           57  3.351777  4.185945  1.148632
0.383412
40569  40570   45221.0           50  4.010990  4.529642  1.211406
1.121462
93618  93619  289293.0           37  6.732461  6.913152  1.734362
0.126751

           CD19      CD22     CD11b  ...      CD61     CD117     CD49d
\
60544  0.253703 -0.018972  2.665005  ...  0.307357  0.208639  2.039954

50673  3.054480  2.493220  0.189975  ...  0.084448  0.033192  0.004637

50682  2.361353  2.281009  0.528589  ... -0.041903 -0.026017  0.109363

1761  -0.035385  0.127904  0.415204  ... -0.001024 -0.017034  0.023385

98760 -0.018816 -0.045954  4.067125  ... -0.029816 -0.046020  0.140410

...         ...       ...       ...  ...       ...       ...       ...

20510  0.105799  0.093621 -0.006647  ...  0.599577  0.376384  2.196247

11540 -0.001227  0.128556  0.008345  ...  0.908547  0.001992  0.464461

30042 -0.037390  0.229479  0.005238  ...  0.596622  0.055177  0.761682

40569  1.185200  0.905587  0.254603  ...  0.120182 -0.007947  1.649371

93618  1.406384  1.672294  0.082506  ... -0.033528 -0.011614  0.134475
```

```
         HLA-DR      CD64      CD41  Viability  file_number
event_number  \
60544  2.847283  2.798986  1.090235   1.005784     3.627711
237532
50673  4.488360  0.866820 -0.002174   0.917810     3.627711
367731
50682  2.328828 -0.008223 -0.018680   1.091297     3.627711
367970
1761   0.120367  0.472159 -0.014919   0.620643     3.627711
164637
98760  0.735830  1.011186 -0.044875   0.149759     3.669327
62492
...         ...       ...       ...        ...          ...
...
20510  0.342656  0.235691  0.128557   1.251073     3.627711
298390
11540 -0.011717  0.331829  0.804992   1.791590     3.627711
103618
30042  0.194395  0.496897  1.122718   0.614461     3.627711
146117
40569  3.598308  0.521024  0.592218   1.099637     3.627711
37211
93618  1.677873  0.355002 -0.013528  -0.017024     3.669327
56333

       individual
60544           1
50673           1
50682           1
1761            1
98760           2
...           ...
20510           1
11540           1
30042           1
40569           1
93618           2

[31256 rows x 41 columns]

Training Labels (y_train):
 64113     10.0
82744      7.0
24294      7.0
7820       6.0
43295      9.0
          ...
54886     10.0
76820      7.0
103694    13.0
```

```
860        1.0
15795      7.0
Name: label, Length: 72928, dtype: float64

Testing Labels (y_test):
 60544    10.0
50673     9.0
50682     9.0
1761      2.0
98760    10.0
         ...
20510     7.0
11540     7.0
30042     8.0
40569     9.0
93618     9.0
Name: label, Length: 31256, dtype: float64
```

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

def corrupt_data(df, corruption_prob=0.5):
    corrupted_df = df.copy()
    for col in df.columns:
        # Create a mask for the column
        mask = np.random.rand(len(df)) < corruption_prob
        # Shuffle column values
        shuffled_values = np.random.permutation(df[col].values)
        # Apply corruption using the mask and shuffled values
        corrupted_df.loc[mask, col] = shuffled_values[mask]

    return corrupted_df

# Load the Levine_32 dataset
df_levine = df

# Apply the corrupt_data function to the Levine_32 dataset
df_levine_corrupted = corrupt_data(df_levine, corruption_prob=0.3)

# Generate the mask matrix
mask_levine = 1 * (df_levine != df_levine_corrupted)

# Get the indices of labeled (corrupted) and unlabeled (uncorrupted)
data
labeled_indices_levine = mask_levine.any(axis=1)
unlabeled_indices_levine = ~labeled_indices_levine

# Create the labeled and unlabeled datasets for Levine_32
x_labeled_levine = df_levine[labeled_indices_levine]
```

```
y_labeled_levine = mask_levine[labeled_indices_levine]
x_unlabeled_levine = df_levine[unlabeled_indices_levine]
y_unlabeled_levine = mask_levine[unlabeled_indices_levine]

# Split labeled data into train and test sets (70/30 ratio)
x_train_levine, x_test_levine, y_train_levine, y_test_levine =
train_test_split(
    x_labeled_levine, y_labeled_levine, test_size=0.3, random_state=42
)

# Print the shapes of the datasets to verify
print("Shape of x_train_levine:", x_train_levine.shape)
print("Shape of y_train_levine:", y_train_levine.shape)
print("Shape of x_test_levine:", x_test_levine.shape)
print("Shape of y_test_levine:", y_test_levine.shape)
```

```
Shape of x_train_levine: (2, 3)
Shape of y_train_levine: (2, 3)
Shape of x_test_levine: (2, 3)
Shape of y_test_levine: (2, 3)
```

```
import pandas as pd
import numpy as np

label_column = 'label'

# Check if the label column exists
if label_column in new_df.columns:
    # Split the data based on the presence of labels
    labeled_data = new_df.dropna(subset=[label_column])
    unlabeled_data = new_df[new_df[label_column].isna()]

    # Optional: Drop the label column from the unlabeled data (as they
are not needed)
    unlabeled_data = unlabeled_data.drop(columns=[label_column])

    print("Labeled Data:")
    print(labeled_data.head())
    print("\nUnlabeled Data:")
    print(unlabeled_data.head())
else:
    print(f"Label column '{label_column}' not found in the dataset.")
```

```
Labeled Data:
   Cell_length      DNA1      DNA2    CD45RA     CD133      CD19
CD22  \
0           22  4.391057  4.617262  0.162691 -0.029585 -0.006696
0.066388
1           35  4.340481  4.816692  0.701349 -0.038280 -0.016654
0.074409
2           32  3.838727  4.386369  0.603568 -0.032216  0.073855 -
```

```
0.042977
3           29  4.255806  4.830048  0.433747 -0.027611 -0.017661 -
0.044072
4           25  3.976909  4.506433 -0.008809 -0.030297  0.080423
0.495791

      CD11b       CD4       CD8  ...      CD13       CD3      CD61
CD117  \
0 -0.009184  0.363602  0.520195  ...  0.038552 -0.032596 -0.002936
0.053050
1  0.808031 -0.035424 -0.010551  ...  1.457326 -0.043466  1.258437
0.089660
2 -0.001881 -0.008781 -0.005632  ...  0.213583  0.320792  0.257137
0.046222
3  0.733698 -0.019066  0.056109  ...  0.514349  0.060443 -0.041140
0.066470
4  1.107627  0.552746  0.031310  ...  0.585712  0.137186  0.168609 -
0.006223

      CD49d    HLA-DR      CD64      CD41  Viability  label
0  0.853505  1.664480 -0.005376 -0.001961   0.648429    1.0
1  0.197818  0.491592  0.144814  0.868014   0.561384    1.0
2  2.586670  1.308337 -0.010961 -0.010413   0.643337    1.0
3  1.338669  0.140523 -0.013449 -0.026039  -0.026523    1.0
4  0.180924  0.197332  0.076167 -0.040488   0.283287    1.0

[5 rows x 37 columns]

Unlabeled Data:
        Cell_length      DNA1      DNA2    CD45RA     CD133      CD19
\
104184           25  4.203073  4.837565  0.095543 -0.027206  0.172384

104185           34  4.042991  4.808275  0.035310 -0.013869 -0.043922

104186           37  4.233125  4.922201  0.415954  0.412757  0.431715

104187           26  3.997143  4.685426 -0.038565  0.125894  0.191383

104188           20  4.115830  4.893428  0.177246  0.171916  0.028568


            CD22     CD11b       CD4       CD8  ...      CD38
CD13  \
104184 -0.001950  0.505713 -0.033282 -0.030555  ...  1.853700
0.082662
104185 -0.001871  0.180261  0.985182 -0.006863  ...  0.768658
0.015103
104186 -0.025619  0.491190  0.485538  0.058734  ...  3.485305
0.535455
```

```
104187 -0.026497  0.342190  1.284395  0.062419  ...  0.309538 -
0.027833
104188 -0.029751  2.480689  0.214722  0.052903  ...  0.844086
0.114819

             CD3      CD61     CD117     CD49d    HLA-DR      CD64
CD41  \
104184  0.048850  3.029787 -0.010093  0.387121  2.859639  2.709532
1.208795
104185  5.084088 -0.017628  0.346248  0.089940 -0.017702  0.045091 -
0.022009
104186  0.221659  0.000544  0.691393  2.996583  5.812406  1.713608
0.479122
104187  4.991485 -0.012887  0.033096 -0.029722 -0.031126 -0.020739 -
0.014693
104188 -0.032199 -0.015719 -0.043689  0.027586  2.543139  3.323810 -
0.002918

        Viability
104184   0.102978
104185   0.092770
104186   1.888485
104187   0.067437
104188   0.109243

[5 rows x 36 columns]
```

```python
from sklearn.model_selection import train_test_split

label_column = 'label'

# Separate features (X) and labels (y) from the labeled data
x = labeled_data.drop(columns=[label_column])
y = labeled_data[label_column]

# Split the labeled data into 70% training and 30% test sets
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.3, random_state=42)

print("Training features (X_train):")
print(x_train.head())
print("\nTest features (X_test):")
print(x_test.head())
print("\nTraining labels (y_train):")
print(y_train.head())
print("\nTest labels (y_test):")
print(y_test.head())
print("\nShape of X_train:", x_train.shape)
print("Shape of X_test:", x_test.shape)
print("Shape of y_train:", y_train.shape)
```

```
print("Shape of y_test:", y_test.shape)
print("shape of X", x.shape)
print("shape of y", y.shape)

Training features (X_train):
       Cell_length      DNA1      DNA2    CD45RA     CD133
CD19  \
64113           25  3.899656  4.594272  0.976652  0.302811  0.154761

82744           31  6.592998  6.901888  0.431481 -0.052898 -0.037690

24294           41  3.543583  4.467671  0.377192  0.219081  0.245478

7820            38  4.305227  4.881685  0.199351  0.100678 -0.025812

43295           26  4.159271  4.861015  0.831285  0.191518  2.002712


           CD22     CD11b       CD4       CD8  ...      CD38      CD13
\
64113 -0.011676  3.180236  1.465950  0.086209  ...  1.563844  0.480488

82744 -0.029715 -0.040846  0.914311  0.022305  ...  1.232765  0.100678

24294  0.193328  0.075123  0.936352 -0.044813  ...  0.486930  0.046766

7820  -0.002898  1.437247 -0.013400 -0.001012  ...  1.250272  0.731957

43295  3.387782  0.179219  0.115231 -0.010963  ...  2.883403  0.345273


             CD3      CD61     CD117     CD49d    HLA-DR      CD64
CD41  \
64113  0.017010  0.051464 -0.003680  1.260410  0.700093  2.355886
0.125409
82744  5.722406 -0.036430  0.021689  0.034946 -0.055651 -0.023248 -
0.054842
24294  4.061728  1.003383  0.406137  1.928676 -0.046849  0.229309
0.937020
7820   0.245939 -0.007282  1.421540  1.443145  2.461705  0.528679
0.072205
43295  0.226596 -0.040754  0.060944  1.294561  3.085858 -0.014128
0.479256

       Viability
64113   0.840205
82744  -0.009329
24294   1.231347
7820    0.892480
43295   2.269233
```

```
[5 rows x 36 columns]

Test features (X_test):
       Cell_length      DNA1      DNA2    CD45RA     CD133
CD19  \
60544           49  3.618797  4.144135  0.198186  0.000282  0.253703

50673           27  3.660988  4.497041  1.272625  0.129642  3.054480

50682           23  3.854865  4.663734  1.527763  0.151383  2.361353

1761            17  3.716473  4.465312  0.375236 -0.037150 -0.035385

98760           32  6.826030  7.007709  0.223441 -0.048813 -0.018816


           CD22     CD11b       CD4       CD8  ...      CD38      CD13
\
60544 -0.018972  2.665005  0.079150 -0.002045  ...  2.479135  1.419488

50673  2.493220  0.189975 -0.024412  0.186744  ...  2.212054 -0.020246

50682  2.281009  0.528589 -0.014516 -0.002732  ...  0.787080 -0.010742

1761   0.127904  0.415204  0.226788  2.802413  ...  0.042091 -0.018271

98760 -0.045954  4.067125  0.004401 -0.012083  ...  1.382377  0.154702


             CD3      CD61     CD117     CD49d    HLA-DR      CD64
CD41  \
60544  0.643676  0.307357  0.208639  2.039954  2.847283  2.798986
1.090235
50673  0.054290  0.084448  0.033192  0.004637  4.488360  0.866820 -
0.002174
50682  0.068448 -0.041903 -0.026017  0.109363  2.328828 -0.008223 -
0.018680
1761  -0.039628 -0.001024 -0.017034  0.023385  0.120367  0.472159 -
0.014919
98760  0.250393 -0.029816 -0.046020  0.140410  0.735830  1.011186 -
0.044875

       Viability
60544   1.005784
50673   0.917810
50682   1.091297
1761    0.620643
98760   0.149759

[5 rows x 36 columns]
```

```
Training labels (y_train):
64113    10.0
82744     7.0
24294     7.0
7820      6.0
43295     9.0
Name: label, dtype: float64

Test labels (y_test):
60544    10.0
50673     9.0
50682     9.0
1761      2.0
98760    10.0
Name: label, dtype: float64

Shape of X_train: (72928, 36)
Shape of X_test: (31256, 36)
Shape of y_train: (72928,)
Shape of y_test: (31256,)
shape of X (104184, 36)
shape of y (104184,)
```

Logistic regression and Xgboost

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Load the dataset

# Define the target column used for labeling
label_column = 'label'

# Separate labeled data
label_df = df[df[label_column].notnull()]

# Split features and labels for labeled data
x_labeled = label_df.drop(columns=[label_column])
y_labeled = label_df[label_column]

# Encode labels if necessary (e.g., for non-numeric labels)
label_encoder = LabelEncoder()
y_labeled = label_encoder.fit_transform(y_labeled)

# Split labeled data into training and testing sets (70%-30% split)
x_train, x_test, y_train, y_test = train_test_split(x_labeled,
```

```
y_labeled, test_size=0.3, random_state=42)

# Scale features for Logistic Regression and XGBoost
scaler = StandardScaler()
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

# Logistic Regression Model with increased max_iter and scaled data
logistic_model = LogisticRegression(max_iter=2000)
logistic_model.fit(x_train_scaled, y_train)
y_test_hat_logistic = logistic_model.predict_proba(x_test_scaled)

# XGBoost Model (using scaled data)
xgb_model = XGBClassifier(eval_metric='mlogloss')
xgb_model.fit(x_train_scaled, y_train)  # Use scaled data for training
y_test_hat_xgb = xgb_model.predict_proba(x_test_scaled)  # Use scaled
test data

# Display the predicted probabilities for Logistic Regression and
XGBoost
print("Logistic Regression Predicted Probabilities:\n",
y_test_hat_logistic)
print("\nXGBoost Predicted Probabilities:\n", y_test_hat_xgb)
```

```
Logistic Regression Predicted Probabilities:
 [[8.71532014e-11 7.15777512e-16 5.86413813e-12 ... 3.71679134e-10
  2.57615664e-08 2.92952487e-08]
 [3.89684389e-14 5.45734530e-13 1.08967360e-13 ... 9.25120234e-10
  3.73081989e-05 6.16167181e-09]
 [1.84225273e-11 5.90550517e-10 1.12355888e-11 ... 1.59113271e-11
  1.62867176e-05 6.66716711e-10]
 ...
 [5.05774251e-10 1.66206209e-05 5.93223532e-09 ... 3.54583853e-09
  3.57005965e-11 9.95221511e-09]
 [6.94828818e-11 7.05048242e-10 2.51660858e-10 ... 1.19779551e-11
  1.34185138e-06 2.54193721e-06]
 [3.32647156e-10 1.39746319e-06 5.84840147e-10 ... 1.09125624e-13
  1.93385444e-05 1.73712999e-10]]

XGBoost Predicted Probabilities:
 [[8.8804103e-07 8.2875789e-07 5.7034327e-07 ... 1.1134865e-06
  7.0003387e-07 9.7590839e-07]
 [7.1397778e-07 7.8185877e-07 5.3256036e-07 ... 1.1057809e-06
  6.3680345e-06 1.2659862e-06]
 [8.2513844e-07 9.2658485e-07 6.4014063e-07 ... 9.4035636e-07
  2.3140719e-06 1.1265030e-06]
 ...
 [4.7537603e-07 1.6399291e-06 4.6528021e-07 ... 5.1612403e-07
  4.0387292e-07 4.1018055e-07]
 [3.5595222e-06 3.8425301e-06 3.3242245e-06 ... 4.7351091e-06
```

```
  7.2303219e-06 2.1757294e-05]
 [1.9763070e-06 1.7193395e-06 1.8571778e-06 ... 2.2942536e-06
  3.7291477e-06 2.6302241e-06]]
```

Logistic Regression Log Loss && XGBoost Log Loss

```
from sklearn.metrics import log_loss
# Calculate log loss for Logistic Regression
logistic_loss = log_loss(y_test, y_test_hat_logistic)
print("Logistic Regression Log Loss:", logistic_loss)

# Calculate log loss for XGBoost
xgb_loss = log_loss(y_test, y_test_hat_xgb)
print("XGBoost Log Loss:", xgb_loss)
```

```
Logistic Regression Log Loss: 0.007401566935815308
XGBoost Log Loss: 0.0014473331046161019
```

ENCODER MODEL

```
from keras.layers import Input, Dense
from keras.models import Model
import numpy as np

# Define binary mask generation function
def binary_mask(p_m, x):
    """
    Generate a binary mask with the same shape as x, where each
element is set to 0 with probability p_m.
    """
    return np.random.binomial(1, 1 - p_m, x.shape)

# Define corruption function
def corruption(mask, x):
    """
    Apply the mask to x, setting masked elements to 0 to simulate
corrupted input.
    """
    return x * mask

# Define the self-supervised function
def self_supervised(x_unlabeled, p_m, alpha, parameters):
    # Extract batch_size and epochs from parameters
    epochs = parameters['epochs']
    batch_size = parameters['batch_size']
    _, dimension = x_unlabeled.shape
```

```
    # Input layer
    input_layer = Input(shape=(dimension,))

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

    # Output 1 --> mask estimation
    output1 = Dense(dimension, activation='sigmoid',
name='mask_estimation')(h)

    # Output 2 --> feature estimation
    output2 = Dense(dimension, activation='sigmoid',
name='feature_estimation')(h)

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

    # Compile the model with custom loss weights as floats
    model.compile(
        optimizer="rmsprop",
        loss={'mask_estimation': "binary_crossentropy",
'feature_estimation': 'mean_squared_error'},
        loss_weights={'mask_estimation': 1.0, 'feature_estimation':
float(alpha)}
    )

    # Generate corruption binary mask
    corruption_binary_mask = binary_mask(p_m, x_unlabeled)

    # Corrupt the unlabeled data
    x_unlabeled_corrupted = corruption(corruption_binary_mask,
x_unlabeled)

    # Train the model with corrupted data
    history = model.fit(
        x_unlabeled_corrupted,
        {'mask_estimation': corruption_binary_mask,
'feature_estimation': x_unlabeled},
        epochs=epochs,
        batch_size=batch_size,
        verbose=1
    )

    return model, history

# Example usage
# Define parameters
parameters = {
    'epochs': 10,
    'batch_size': 32
```

```
}
alpha = 0.5  # Weight for feature estimation loss
p_m = 0.3  # Masking probability

# Generate some example data for x_unlabeled
x_unlabeled = np.random.rand(100, 37)  # 100 samples with 37 features

# Run self-supervised training
model, history = self_supervised(x_unlabeled, p_m, alpha, parameters)

# Print the model summary and training history
model.summary()
print("Training history:", history.history)
```

```
Epoch 1/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 5s 426ms/step - loss: 0.7569
Epoch 2/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 0.7432
Epoch 3/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.7330
Epoch 4/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.7264
Epoch 5/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.7228
Epoch 6/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.7172
Epoch 7/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.7104
Epoch 8/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.7040
Epoch 9/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.7007
Epoch 10/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.6964

Model: "functional_1"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape                           ┃                        Param # ┃
┃ Connected to                    ┃                                        ┃                                ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│ input_layer_1                   │ (None, 37)                             │                              0 │
│ -                               │                                        │                                │
│ (InputLayer)                    │                                        │                                │
├─────────────────────────────────┼────────────────────────────────────────┼────────────────────────────────┤
│ dense_1 (Dense)                 │ (None, 37)                             │                          1,406 │
```

```
│ input_layer_1[0][0]    │
├────────────────────────┼──────────────────────────┼──────────────────
┼────────────────────────┤
│ mask_estimation (Dense)   │ (None, 37)               │           1,406
│ dense_1[0][0]          │
├────────────────────────┼──────────────────────────┼──────────────────
┼────────────────────────┤
│ feature_estimation        │ (None, 37)               │           1,406
│ dense_1[0][0]          │
│ (Dense)                   │                          │
│                        │
└────────────────────────┴──────────────────────────┴──────────────────
┴────────────────────────┘

 Total params: 8,438 (32.96 KB)

 Trainable params: 4,218 (16.48 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 4,220 (16.49 KB)

Training history: {'loss': [0.7551195621490479, 0.7412986755371094,
0.7332472801208496, 0.7270447611808777, 0.7216742634773254,
0.7160345315933228, 0.7109150886535645, 0.7057392001152039,
0.700700581073761, 0.695417582988739]}

from keras.layers import Input, Dense
from keras.models import Model
import numpy as np
from sklearn.preprocessing import StandardScaler

# Define binary mask generation function
def binary_mask(p_m, x):
    """
    Generate a binary mask with the same shape as x, where each
element is set to 0 with probability p_m.
    """
    return np.random.binomial(1, 1 - p_m, x.shape)

# Define corruption function
def corruption(mask, x):
    """
    Apply the mask to x, setting masked elements to 0 to simulate
corrupted input.
    """
    return x * mask

# Define the self-supervised function
def self_supervised(x_unlabeled, p_m, alpha, parameters):
    # Extract batch_size and epochs from parameters
```

```
    epochs = parameters['epochs']
    batch_size = parameters['batch_size']
    learning_rate = parameters['learning_rate']
    hidden_units = parameters['hidden']
    _, dimension = x_unlabeled.shape

    # Input layer
    input_layer = Input(shape=(dimension,))

    # Encoder layer
    h = Dense(hidden_units, activation='relu')(input_layer)  # Use
hidden_units for the number of neurons

    # Output 1 --> mask estimation
    output1 = Dense(dimension, activation='sigmoid',
name='mask_estimation')(h)

    # Output 2 --> feature estimation
    output2 = Dense(dimension, activation='sigmoid',
name='feature_estimation')(h)

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

    # Compile the model with custom loss weights as floats
    model.compile(
        optimizer="rmsprop",  # Using the optimizer defined by the
user
        loss={'mask_estimation': "binary_crossentropy",
'feature_estimation': 'mean_squared_error'},
        loss_weights={'mask_estimation': 1.0, 'feature_estimation':
float(alpha)}
    )

    # Generate corruption binary mask
    corruption_binary_mask = binary_mask(p_m, x_unlabeled)

    # Corrupt the unlabeled data
    x_unlabeled_corrupted = corruption(corruption_binary_mask,
x_unlabeled)

    # Training the model with corrupted data
    history = model.fit(
        x_unlabeled_corrupted,
        {'mask_estimation': corruption_binary_mask,
'feature_estimation': x_unlabeled},
        epochs=epochs,
        batch_size=batch_size,
        verbose=1
    )
```

```
    # Extract encoder part of the model
    name_of_layer = model.layers[1].name
    layer_output = model.get_layer(name_of_layer).output
    encoder = Model(inputs=model.input, outputs=layer_output)

    return encoder, model, history


# Example usage
# Example raw data (replace with your actual data)
x_unlabeled = np.random.rand(100, 37)  # 100 samples, 37 features
(replace with your data)

# Scale the data using StandardScaler
scaler = StandardScaler()
x_unlabeled_scaled = scaler.fit_transform(x_unlabeled)  # Scaled data

# Masking probability, alpha value, and parameters
p_m = 0.3  # Masking probability
alpha = 2.0  # Weight for feature estimation loss

# Parameters for training
parameters = {
    'batch_size': 128,         # Batch size for training
    'epochs': 50,              # Number of epochs for training
    'learning_rate': 0.001,    # Learning rate for the optimizer
    'hidden': 64               # Number of neurons in the hidden layer
}

# Run self-supervised training
encoder, model, history = self_supervised(x_unlabeled_scaled, p_m,
alpha, parameters)

# Print the model summary
model.summary()

# Optionally, you can also print the training history
print("Training history:", history.history)
```

```
Epoch 1/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 3s 3s/step - loss: 3.3035
Epoch 2/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 68ms/step - loss: 3.2532
Epoch 3/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 173ms/step - loss: 3.2173
Epoch 4/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step - loss: 3.1879
Epoch 5/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 72ms/step - loss: 3.1622
```

```
Epoch 6/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step - loss: 3.1391
Epoch 7/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - loss: 3.1178
Epoch 8/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 89ms/step - loss: 3.0980
Epoch 9/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 144ms/step - loss: 3.0793
Epoch 10/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step - loss: 3.0615
Epoch 11/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - loss: 3.0445
Epoch 12/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 72ms/step - loss: 3.0282
Epoch 13/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 134ms/step - loss: 3.0124
Epoch 14/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step - loss: 2.9971
Epoch 15/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 87ms/step - loss: 2.9823
Epoch 16/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 149ms/step - loss: 2.9680
Epoch 17/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step - loss: 2.9539
Epoch 18/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 91ms/step - loss: 2.9403
Epoch 19/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 118ms/step - loss: 2.9269
Epoch 20/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step - loss: 2.9138
Epoch 21/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 145ms/step - loss: 2.9010
Epoch 22/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 121ms/step - loss: 2.8884
Epoch 23/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 106ms/step - loss: 2.8761
Epoch 24/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 93ms/step - loss: 2.8641
Epoch 25/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 145ms/step - loss: 2.8523
Epoch 26/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 147ms/step - loss: 2.8407
Epoch 27/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 147ms/step - loss: 2.8292
Epoch 28/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 106ms/step - loss: 2.8180
Epoch 29/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - loss: 2.8069
Epoch 30/50
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step - loss: 2.7961
Epoch 31/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 133ms/step - loss: 2.7855
Epoch 32/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 142ms/step - loss: 2.7750
Epoch 33/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 237ms/step - loss: 2.7647
Epoch 34/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 309ms/step - loss: 2.7546
Epoch 35/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 145ms/step - loss: 2.7447
Epoch 36/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 155ms/step - loss: 2.7350
Epoch 37/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step - loss: 2.7254
Epoch 38/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - loss: 2.7160
Epoch 39/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 2.7067
Epoch 40/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 2.6976
Epoch 41/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.6886
Epoch 42/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 2.6798
Epoch 43/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 2.6711
Epoch 44/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step - loss: 2.6626
Epoch 45/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 2.6542
Epoch 46/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 2.6459
Epoch 47/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - loss: 2.6378
Epoch 48/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - loss: 2.6298
Epoch 49/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 2.6219
Epoch 50/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 2.6141

Model: "functional_4"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┃ Connected to                    ┃                        ┃               ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
```

```
┃ input_layer_3                ┃ (None, 37)                ┃                0
-                          ┃
(InputLayer)               ┃                           ┃
                           ┃
━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━
━━━━━━━━━━━━━━━━━━━━━━━━━━┩
dense_3 (Dense)              ┃ (None, 64)                ┃            2,432
input_layer_3[0][0]        ┃
━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━
━━━━━━━━━━━━━━━━━━━━━━━━━━┩
mask_estimation (Dense)      ┃ (None, 37)                ┃            2,405
dense_3[0][0]              ┃
━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━
━━━━━━━━━━━━━━━━━━━━━━━━━━┩
feature_estimation           ┃ (None, 37)                ┃            2,405
dense_3[0][0]              ┃
(Dense)                    ┃                           ┃
                           ┃
━━━━━━━━━━━━━━━━━━━━━━━━━━━┻━━━━━━━━━━━━━━━━━━━━━━━━━━━┻━━━━━━━━━━━━━━━━━━
━━━━━━━━━━━━━━━━━━━━━━━━━━┛

 Total params: 14,486 (56.59 KB)

 Trainable params: 7,242 (28.29 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 7,244 (28.30 KB)

Training history: {'loss': [3.3034558296203613, 3.253218173980713,
3.217334508895874, 3.1878931522369385, 3.1622400283813477,
3.1391210556030273, 3.1178383827209473, 3.0979909896850586,
3.0793159008026123, 3.0615434646606445, 3.044511318206787,
3.0281620025634766, 3.0124001502990723, 2.9971306324005127,
2.982325553894043, 2.967958450317383, 2.953948736190796,
2.9402780532836914, 2.926877737045288, 2.9137725830078125,
2.9009621143341064, 2.888425350189209, 2.8761484622955322,
2.8641092777252197, 2.8522937297821045, 2.840651750564575,
2.829228401184082, 2.817988395690918, 2.806945562362671,
2.7961041927337646, 2.785466194152832, 2.7749996185302734,
2.764725685119629, 2.7546205520629883, 2.744699478149414,
2.734952449798584, 2.7253730297088623, 2.7159605026245117,
2.706690788269043, 2.697579860687256, 2.6886072158813477,
2.6797866821289062, 2.671107292175293, 2.662569999694824,
2.6541800498962402, 2.645921230316162, 2.6377735137939453,
2.629751205444336, 2.6218695640563965, 2.614124059677124]}

from keras.layers import Input, Dense
from keras.models import Model
import numpy as np
```

```
def binary_mask(p_m, data):
    """Generates a binary mask with probability p_m."""
    return np.random.binomial(1, 1 - p_m, data.shape)

def corruption(mask, data):
    num_samples, num_features = data.shape
    shuffled_data = np.zeros([num_samples, num_features])

    for feature_idx in range(num_features):
        shuffled_indices = np.random.permutation(num_samples)
        shuffled_data[:, feature_idx] = data[shuffled_indices,
feature_idx]

    data_corrupted = data * (1 - mask) + shuffled_data * mask
    mask_new = (data != data_corrupted).astype(int)

    return mask_new, data_corrupted

def self_supervised(x_unlabeled, p_m, alpha, parameters):
    epochs = parameters['epochs']
    batch_size = parameters['batch_size']
    _, dimension = x_unlabeled.shape

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

    output1 = Dense(int(dimension), activation='sigmoid',
name='mask_estimation')(h)
    output2 = Dense(int(dimension), activation='sigmoid',
name='feature_estimation')(h)

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

    # Compile model with appropriate loss functions and weights
    model.compile(
        optimizer="rmsprop",
        loss={'mask_estimation': 'binary_crossentropy',
'feature_estimation': 'mean_squared_error'},
        loss_weights={'mask_estimation': 1.0, 'feature_estimation':
float(alpha)}  # Corrected to use float
    )

    # Generate corrupted input and mask labels
    corruption_binary_mask = binary_mask(p_m, x_unlabeled)
    x_unlabeled_corrupted,mask_label =
corruption(corruption_binary_mask, x_unlabeled)

    assert x_unlabeled_corrupted.shape == mask_label.shape
```

```
    # Train model
    model.fit(x_unlabeled_corrupted, {'mask_estimation': mask_label,
'feature_estimation': x_unlabeled},
              epochs=epochs, batch_size=batch_size)

    # Display model summary (this will print the model's parameters)
    model.summary()

    # Define encoder
    name_of_layer = model.layers[1].name
    layer_output = model.get_layer(name_of_layer).output
    encoder = Model(inputs=model.input, outputs=layer_output)

    return encoder


    import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

# Load the dataset
data = df

# Exclude specified columns
exclude_columns = ['Event', 'Time', 'Cell_length', 'file_number',
'event_number', 'label', 'individual']
data_filtered = data.drop(columns=exclude_columns)

# 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)

Epoch 1/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - loss: 2.0901
Epoch 2/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 7s 1ms/step - loss: 1.9985
Epoch 3/50
```

```
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - loss: 1.9923
Epoch 4/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 1.9911
Epoch 5/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 1.9628
Epoch 6/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 1.9646
Epoch 7/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 2.0023
Epoch 8/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 1.9350
Epoch 9/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 1.9250
Epoch 10/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 1.7993
Epoch 11/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 1.8982
Epoch 12/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 1.7882
Epoch 13/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - loss: 1.6235
Epoch 14/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 1.7309
Epoch 15/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 1.8710
Epoch 16/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 1.8308
Epoch 17/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 1.3090
Epoch 18/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 1.2085
Epoch 19/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 1.5766
Epoch 20/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 1.4759
Epoch 21/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 1.3987
Epoch 22/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - loss: 1.4454
Epoch 23/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 1.2118
Epoch 24/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 1.0863
Epoch 25/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 1.4461
Epoch 26/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.3247
Epoch 27/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 0.9816
```

```
Epoch 28/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: -0.3328
Epoch 29/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - loss: 0.5657
Epoch 30/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 0.7519
Epoch 31/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - loss: 0.4376
Epoch 32/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: -0.6964
Epoch 33/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: 0.7095
Epoch 34/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - loss: -0.3430
Epoch 35/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 0.3017
Epoch 36/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: -1.7692
Epoch 37/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: -0.8509
Epoch 38/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: -1.5674
Epoch 39/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: 0.0469
Epoch 40/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: -2.3817
Epoch 41/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: -1.6022
Epoch 42/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: -1.8190
Epoch 43/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: -1.5599
Epoch 44/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: -1.2138
Epoch 45/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - loss: -0.1428
Epoch 46/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: -0.5129
Epoch 47/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: -4.0247
Epoch 48/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - loss: -4.0278
Epoch 49/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - loss: -4.2943
Epoch 50/50
2076/2076 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - loss: -2.3350

Model: "functional"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape                           ┃                        Param # ┃
┃ Connected to                    ┃                                        ┃                                ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│ input_layer (InputLayer)        │ (None, 35)                             │                              0 │
│ -                               │                                        │                                │
├─────────────────────────────────┼────────────────────────────────────────┼────────────────────────────────┤
│ dense (Dense)                   │ (None, 35)                             │                          1,260 │
│ input_layer[0][0]               │                                        │                                │
├─────────────────────────────────┼────────────────────────────────────────┼────────────────────────────────┤
│ mask_estimation (Dense)         │ (None, 35)                             │                          1,260 │
│ dense[0][0]                     │                                        │                                │
├─────────────────────────────────┼────────────────────────────────────────┼────────────────────────────────┤
│ feature_estimation              │ (None, 35)                             │                          1,260 │
│ dense[0][0]                     │                                        │                                │
│ (Dense)                         │                                        │                                │
└─────────────────────────────────┴────────────────────────────────────────┴────────────────────────────────┘
```

```
 Total params: 7,562 (29.54 KB)

 Trainable params: 3,780 (14.77 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 3,782 (14.78 KB)
```

```
encoder_path = "/content/encoder_model.keras"
encoder_model.save(encoder_path)

from keras.models import load_model
encoder_model = load_model(encoder_path)

from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier
from sklearn.metrics import log_loss

# Step 1: Encode the training and testing data using the encoder model
X_train_scaled_encoded = encoder_model.predict(X_train_scaled)
X_test_scaled_encoded = encoder_model.predict(X_test_scaled)

# Step 2: Train and evaluate Logistic Regression on the encoded data
logistic_model = LogisticRegression(max_iter=2000)
logistic_model.fit(X_train_scaled_encoded, y_train)
y_encoded_logistic =
```

```
logistic_model.predict_proba(X_test_scaled_encoded)

# Step 3: Compute and print log loss for Logistic Regression
print("Logistic Regression Log Loss:", log_loss(y_test,
y_encoded_logistic))

# Step 4: Train and evaluate XGBoost on the encoded data
xgb_model = XGBClassifier(eval_metric='mlogloss')
xgb_model.fit(X_train_scaled_encoded, y_train)
y_encoded_xgb = xgb_model.predict_proba(X_test_scaled_encoded)

# Step 5: Compute and print log loss for XGBoost
print("XGBoost Log Loss:", log_loss(y_test, y_encoded_xgb))
```

```
2279/2279 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step
977/977 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step
Logistic Regression Log Loss: 0.8070695382083957
XGBoost Log Loss: 0.8596009027566897
```