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

import warnings
warnings.filterwarnings('ignore')

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

```
Mounted at /content/drive
```

Load the CSV File

```
import pandas as pd


df = pd.read_csv('/content/drive/MyDrive/data/Levine_32dim.csv')
df.head()
```

```
<ipython-input-2-6ad238ab5ce1>:4: DtypeWarning: Columns (39) have
mixed types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv('/content/drive/MyDrive/data/Levine_32dim.csv')
```


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


This line gives the datatypes of all the columns

```
df.dtypes
```

```
Time              float64
 Cell_length        int64
 DNA1              float64
 DNA2              float64
 CD45RA            float64
 CD133             float64
 CD19              float64
 CD22              float64
 CD11b             float64
 CD4               float64
 CD8               float64
 CD34              float64
 Flt3              float64
 CD20              float64
 CXCR4             float64
 CD235ab           float64
 CD45              float64
 CD123             float64
```

```
 CD321            float64
 CD14             float64
 CD33             float64
 CD47             float64
 CD11c            float64
 CD7              float64
 CD15             float64
 CD16             float64
 CD44             float64
 CD38             float64
 CD13             float64
 CD3              float64
 CD61             float64
 CD117            float64
 CD49d            float64
 HLA-DR           float64
 CD64             float64
 CD41             float64
 Viability        float64
 file_number      float64
 event_number       int64
 label             object
 individual         int64
                   object
dtype: object
```

```
df.tail(5)
```

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

This line retrieves the column names of the DataFrame df

```
df.columns
```

```
Index(['Time', ' Cell_length', ' DNA1', ' DNA2', ' CD45RA', ' CD133',
' CD19',
       ' CD22', ' CD11b', ' CD4', ' CD8', ' CD34', ' Flt3', ' CD20', '
CXCR4',
       ' CD235ab', ' CD45', ' CD123', ' CD321', ' CD14', ' CD33', '
CD47',
       ' CD11c', ' CD7', ' CD15', ' CD16', ' CD44', ' CD38', ' CD13',
' CD3',
       ' CD61', ' CD117', ' CD49d', ' HLA-DR', ' CD64', ' CD41', '
Viability',
       ' file_number', ' event_number', ' label', ' individual', ' '],
      dtype='object')
```

This line gives the size of the dataframe df

```
df.size
```

```
11156292
```

```
df.shape
```

```
(265626, 42)
```

This checks the number of duplicate rows

```
# Find duplicate rows
duplicate_rows_levin_df = df[df.duplicated()]

# Print the number of duplicate rows
print("Number of duplicate rows: ", duplicate_rows_levin_df.shape[0])
```

```
Number of duplicate rows:  0
```

```
df.count()      # Used to count the number of rows
```

```
Time              265626
 Cell_length      265626
 DNA1             265626
 DNA2             265626
 CD45RA           265626
 CD133            265626
 CD19             265626
 CD22             265626
 CD11b            265626
 CD4              265626
 CD8              265626
 CD34             265626
 Flt3             265626
 CD20             265626
 CXCR4            265626
 CD235ab          265626
 CD45             265626
 CD123            265626
 CD321            265626
 CD14             265626
 CD33             265626
 CD47             265626
 CD11c            265626
 CD7              265626
 CD15             265626
 CD16             265626
 CD44             265626
 CD38             265626
 CD13             265626
 CD3              265626
```

```
 CD61             265626
 CD117            265626
 CD49d            265626
 HLA-DR           265626
 CD64             265626
 CD41             265626
 Viability        265626
 file_number      265626
 event_number     265626
 label            265626
 individual       265626
                  265626
dtype: int64
```

```
print(df.isnull().sum())
```

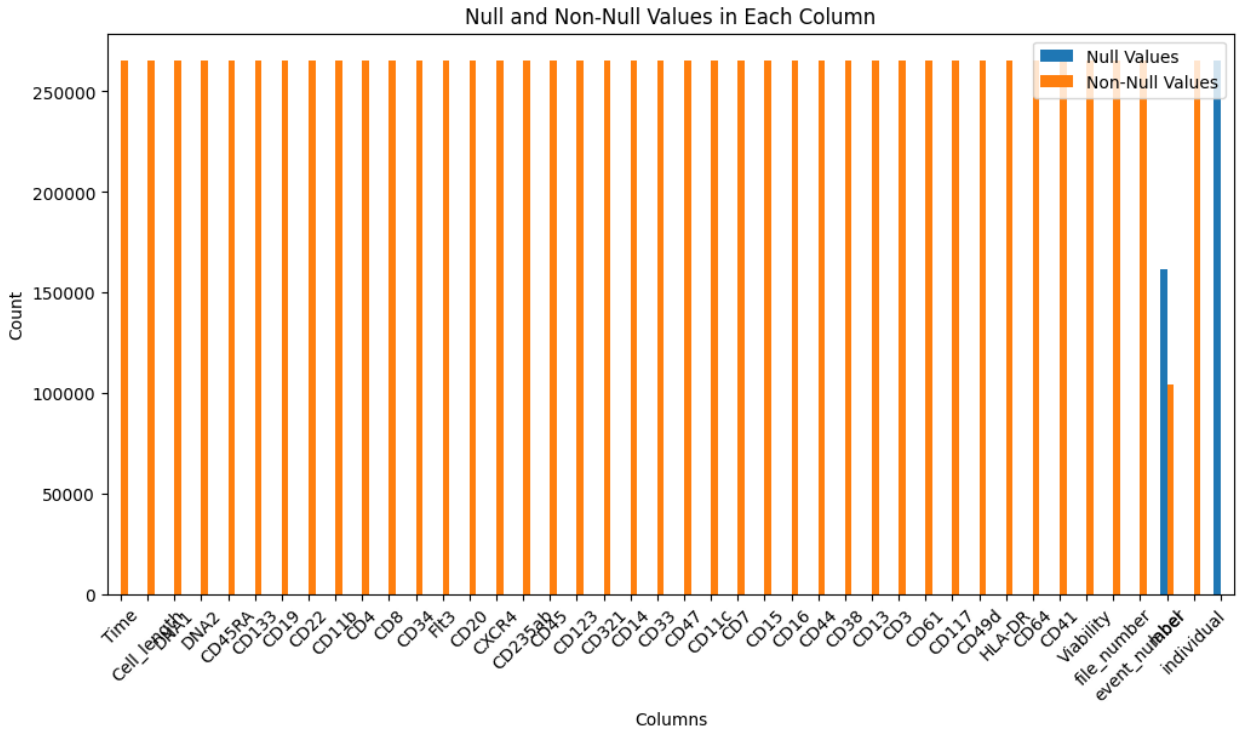
```
Time              0
 Cell_length      0
 DNA1             0
 DNA2             0
 CD45RA           0
 CD133            0
 CD19             0
 CD22             0
 CD11b            0
 CD4              0
 CD8              0
 CD34             0
 Flt3             0
 CD20             0
 CXCR4            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           0
 individual      0
                 0
dtype: int64

df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 265626 entries, 0 to 265625
Data columns (total 42 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   Time            265626 non-null  float64
 1    Cell_length    265626 non-null  int64
 2    DNA1           265626 non-null  float64
 3    DNA2           265626 non-null  float64
 4    CD45RA         265626 non-null  float64
 5    CD133          265626 non-null  float64
 6    CD19           265626 non-null  float64
 7    CD22           265626 non-null  float64
 8    CD11b          265626 non-null  float64
 9    CD4            265626 non-null  float64
 10   CD8            265626 non-null  float64
 11   CD34           265626 non-null  float64
 12   Flt3           265626 non-null  float64
 13   CD20           265626 non-null  float64
 14   CXCR4          265626 non-null  float64
 15   CD235ab        265626 non-null  float64
 16   CD45           265626 non-null  float64
 17   CD123          265626 non-null  float64
 18   CD321          265626 non-null  float64
 19   CD14           265626 non-null  float64
 20   CD33           265626 non-null  float64
 21   CD47           265626 non-null  float64
 22   CD11c          265626 non-null  float64
 23   CD7            265626 non-null  float64
 24   CD15           265626 non-null  float64
 25   CD16           265626 non-null  float64
 26   CD44           265626 non-null  float64
 27   CD38           265626 non-null  float64
 28   CD13           265626 non-null  float64
 29   CD3            265626 non-null  float64
 30   CD61           265626 non-null  float64
 31   CD117          265626 non-null  float64
 32   CD49d          265626 non-null  float64
```

```
 33   HLA-DR         265626 non-null  float64
 34   CD64           265626 non-null  float64
 35   CD41           265626 non-null  float64
 36   Viability      265626 non-null  float64
 37   file_number    265626 non-null  float64
 38   event_number   265626 non-null  int64
 39   label          265626 non-null  object
 40   individual     265626 non-null  int64
 41                  265626 non-null  object
dtypes: float64(37), int64(3), object(2)
memory usage: 85.1+ MB
```

The code calculates and plots the number of null and non-null values for each column in the DataFrame.

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

# Create a DataFrame for plotting
plot_data = pd.DataFrame({
    'Null Values': null_counts,
    'Non-Null Values': non_null_counts
})

# Plotting the null and non-null values
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()
```




```
cat_columns=df.select_dtypes(include='object').columns
num_columns=df.select_dtypes(exclude='object').columns

cat_columns
```

```
Index([' label', ' '], dtype='object')
```

Showing the columns before removing any columns

```
num_columns
```

```
Index(['Time', ' Cell_length', ' DNA1', ' DNA2', ' CD45RA', ' CD133',
' CD19',
       ' CD22', ' CD11b', ' CD4', ' CD8', ' CD34', ' Flt3', ' CD20', '
CXCR4',
       ' CD235ab', ' CD45', ' CD123', ' CD321', ' CD14', ' CD33', '
CD47',
       ' CD11c', ' CD7', ' CD15', ' CD16', ' CD44', ' CD38', ' CD13',
' CD3',
       ' CD61', ' CD117', ' CD49d', ' HLA-DR', ' CD64', ' CD41', '
Viability',
       ' file_number', ' event_number', ' individual'],
      dtype='object')
```

Removing three columns namely file_number, Time, event_number

```
columns_to_remove = ['file_number', 'Time', 'event_number']
df = df.drop(columns=columns_to_remove, errors='ignore')
print(df.head())
```

```
   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  0.701348 -0.038280 -0.016654
0.074409
2           32  3.838727  4.386369  0.603568 -0.032216  0.073855 -
0.042977
3           29  4.255805  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  ...     CD49d    HLA-DR      CD64
CD41  \
0 -0.009184  0.363602  0.520195  ...  0.853505  1.664480 -0.005376 -
0.001961
1  0.808031 -0.035424 -0.010551  ...  0.197818  0.491592  0.144814
0.868014
2 -0.001881 -0.008781 -0.005632  ...  2.586670  1.308337 -0.010961 -
0.010413
3  0.733698 -0.019066  0.056109  ...  1.338669  0.140523 -0.013449 -
0.026039
4  1.107627  0.552746  0.031310  ...  0.180924  0.197332  0.076167 -
0.040488

   Viability  file_number  event_number  label  individual
0   0.648429     3.627711           307    1.0           1  0.0
1   0.561384     3.627711           545    1.0           1  0.0
2   0.643337     3.627711          1726    1.0           1  0.0
3  -0.026523     3.627711          1766    1.0           1  0.0
4   0.283287     3.627711          2031    1.0           1  0.0

[5 rows x 41 columns]
```

This shows all the columns present after dropping few irrevalent columns

```
num_columns
```

```
Index(['Time', ' Cell_length', ' DNA1', ' DNA2', ' CD45RA', ' CD133',
' CD19',
       ' CD22', ' CD11b', ' CD4', ' CD8', ' CD34', ' Flt3', ' CD20', '
CXCR4',
       ' CD235ab', ' CD45', ' CD123', ' CD321', ' CD14', ' CD33', '
CD47',
       ' CD11c', ' CD7', ' CD15', ' CD16', ' CD44', ' CD38', ' CD13',
```

```
' CD3',
       ' CD61', ' CD117', ' CD49d', ' HLA-DR', ' CD64', ' CD41', '
Viability',
       ' file_number', ' event_number', ' individual'],
      dtype='object')

# Drop any unnamed columns that are usually irrelevant
df_cleaned = df.loc[:, ~df.columns.str.contains('^Unnamed')]

# Display the cleaned dataset
df_cleaned.head()

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

This code cleans the column names which have leading /tralling spaces

```
# Clean the column names: strip any leading/trailing spaces and
replace any special characters
df.columns = df.columns.str.strip().str.replace(' ',
'_').str.replace('(', '').str.replace(')', '')

# Now you can check if the column 'Cell_length' exists and proceed
with your operations
if 'Cell_length' in df.columns:
    print("Column 'Cell_length' is now accessible.")

# Display cleaned column names to verify
print(df.columns)

Column 'Cell_length' is now accessible.
Index(['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')
```

To detect the outliers using z-score method

```
# Convert non-numeric values to NaN
df = df.apply(pd.to_numeric, errors='coerce')

# Calculate Z-scores
z_scores = np.abs((df - df.mean()) / df.std())
```

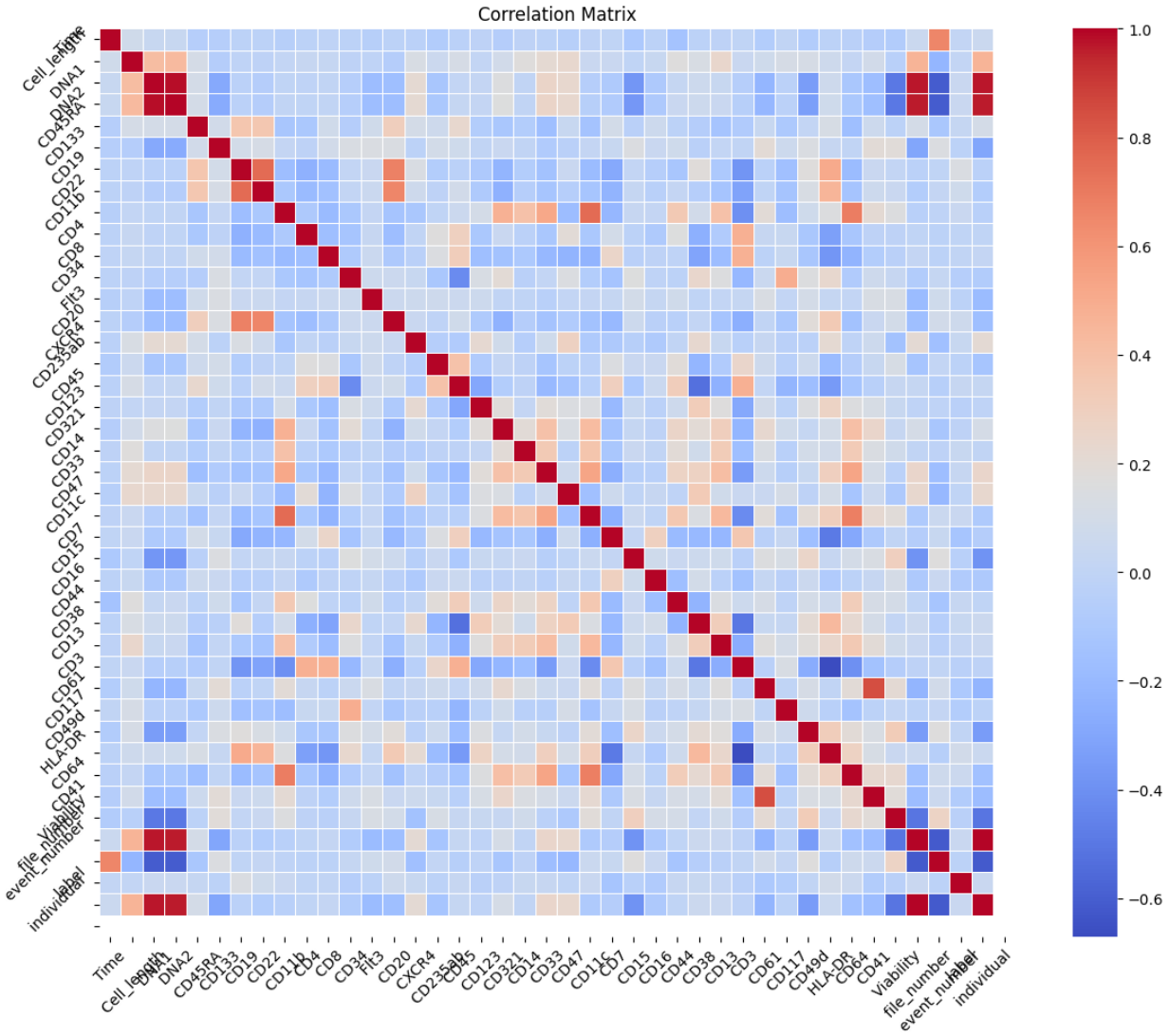
```python
# Set a threshold for Z-scores
threshold = 3

# Identify outliers
outliers = df[(z_scores > threshold).any(axis=1)]

print("Outliers detected using Z-score method:")
print(outliers)
```

```
Outliers detected using Z-score method:
                Time  Cell_length      DNA1      DNA2    CD45RA
CD133  \
0         2693.0000           22  4.391057  4.617262  0.162691 -
0.029585
1         3736.0000           35  4.340481  4.816692  0.701348 -
0.038280
2         7015.0000           32  3.838727  4.386369  0.603568 -
0.032216
3         7099.0000           29  4.255805  4.830048  0.433747 -
0.027611
4         7700.0000           25  3.976909  4.506433 -0.008809 -
0.030297
...             ...          ...       ...       ...       ...
..
265616  706794.4375           60  6.931340  7.472308  0.654410
0.090968
265619  707701.4375           51  6.907256  7.294560  0.065163 -
0.054869
265620  707902.4375           44  6.934081  7.210765  0.864620
0.013602
265621  707917.4375           60  6.733888  7.179924  1.901087 -
0.054719
265625  708585.4375           39  6.865218  7.144353  0.288761 -
0.011310

            CD19      CD22     CD11b       CD4  ...     CD49d    HLA-
DR  \
0      -0.006696  0.066388 -0.009184  0.363602  ...  0.853505
1.664480
1      -0.016654  0.074409  0.808031 -0.035424  ...  0.197818
0.491592
2       0.073855 -0.042977 -0.001881 -0.008781  ...  2.586670
1.308337
3      -0.017661 -0.044072  0.733698 -0.019066  ...  1.338669
0.140523
4       0.080423  0.495791  1.107627  0.552746  ...  0.180924
0.197332
...          ...       ...       ...       ...  ...       ...
..
.
```

```
265616 -0.004919  0.157188  2.901444 -0.030128  ...  1.002225
2.943578
265619 -0.041996  0.006614  3.506185  0.246416  ...  0.301530
0.651898
265620 -0.054534 -0.035866 -0.054300  2.089668  ...  0.593445
0.322045
265621  3.127012  2.389596  0.212047  0.003287  ...  0.069388
3.550516
265625 -0.048786  0.073983 -0.031787  0.078800  ...  0.101955
6.200001

             CD64      CD41  Viability  file_number  event_number
label  \
0      -0.005376 -0.001961   0.648429     3.627711           307
1.0
1       0.144814  0.868014   0.561384     3.627711           545
1.0
2      -0.010961 -0.010413   0.643337     3.627711          1726
1.0
3      -0.013449 -0.026039  -0.026523     3.627711          1766
1.0
4       0.076167 -0.040488   0.283287     3.627711          2031
1.0
...          ...       ...        ...          ...           ...    ..
.
265616  2.650225  0.266800  -0.011408     3.669327        102653
NaN
265619  2.210022  0.418056  -0.051843     3.669327        102679
NaN
265620 -0.056968 -0.036302  -0.044796     3.669327        102684
NaN
265621  0.147588 -0.043806   0.144479     3.669327        102685
NaN
265625  0.296877  0.192786   0.620872     3.669327        102706
NaN

        individual
0                1 NaN
1                1 NaN
2                1 NaN
3                1 NaN
4                1 NaN
...            ...  ..
265616           2 NaN
265619           2 NaN
265620           2 NaN
265621           2 NaN
265625           2 NaN

[92237 rows x 42 columns]
```

This code calculates and visualizes the correlation matrix for numeric columns using a heatmap

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
filepath='/content/drive/MyDrive/data/Levine_32dim.csv'
df=pd.read_csv(filepath)
df = df.apply(pd.to_numeric, errors='coerce')

df = df.fillna(0)
correlation_matrix=df.corr()

plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix,annot=False,cmap='coolwarm',fmt='.2f',l
inewidths=0.5)
plt.title('Correlation Matrix')
plt.xticks(rotation=45)
plt.yticks(rotation=45)
plt.tight_layout()
plt.show()
```



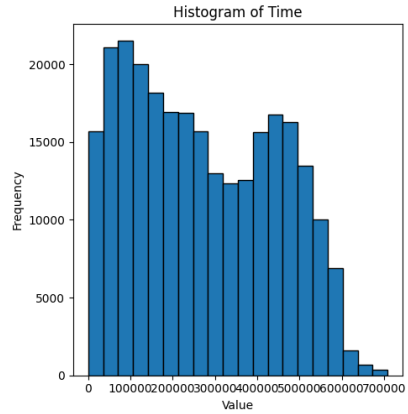

```
corr = df.corr()
corr.head()
```

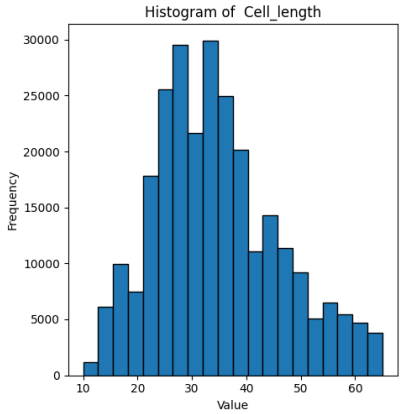
{"type":"dataframe","variable_name":"corr"}

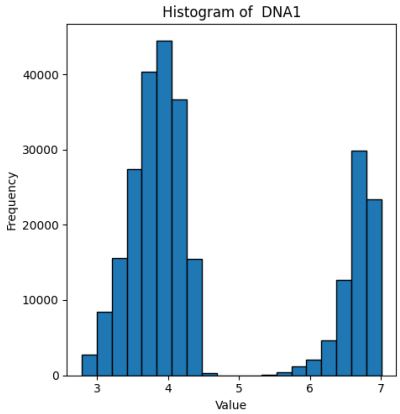
This code plots the histogram for all the values

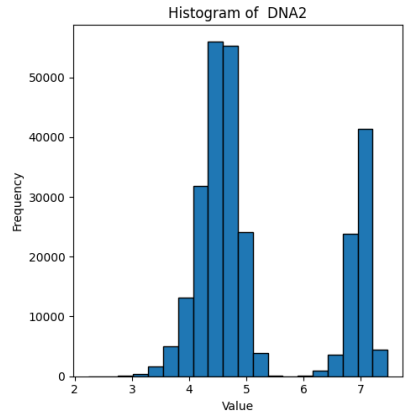
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

filepath='/content/drive/MyDrive/data/Levine_32dim.csv'
df=pd.read_csv(filepath)

# Convert all columns to numeric, replacing errors with NaN
```

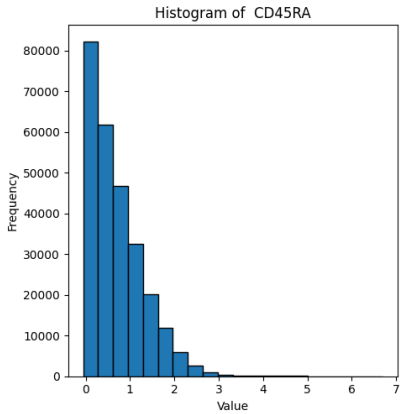
```
df = df.apply(pd.to_numeric, errors='coerce')

# Replace any remaining empty strings or NaN with 0
df = df.fillna(0)

# Create subplots for each column
num_cols = len(df.columns)
num_rows = (num_cols + 2) // 3  # Calculate number of rows for
subplots

fig, axes = plt.subplots(num_rows, 3, figsize=(15, 5 * num_rows))  #
Adjust figsize as needed
axes = axes.flatten()  # Flatten the axes array for easier iteration

for i, col in enumerate(df.columns):
    axes[i].hist(df[col], bins=20, edgecolor='black')  # Adjust bins
as needed
    axes[i].set_title(f'Histogram of {col}')
    axes[i].set_xlabel('Value')
    axes[i].set_ylabel('Frequency')

# Hide any unused subplots
for j in range(i + 1, len(axes)):
    axes[j].axis('off')

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

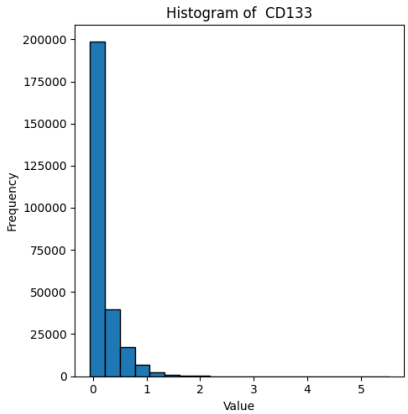

Histogram of Time
Frequency
Value

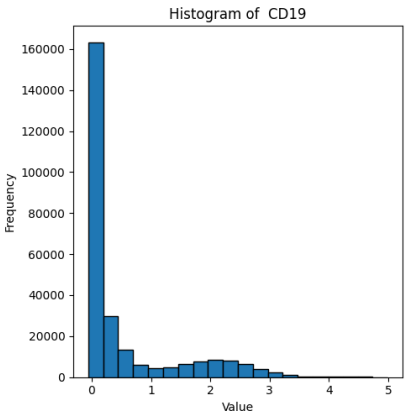

Histogram of Cell_length
Frequency
Value

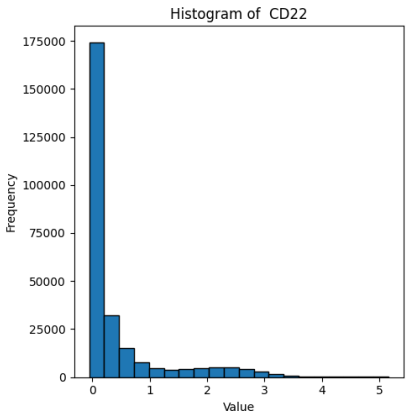

Histogram of DNA1
Frequency
Value

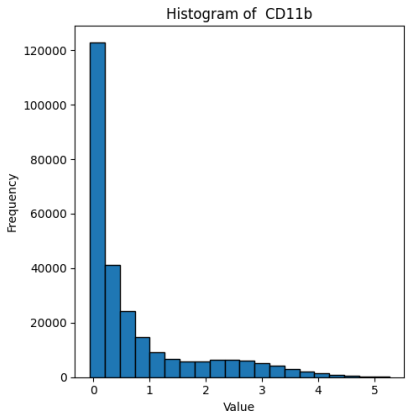

Histogram of DNA2
Frequency
Value

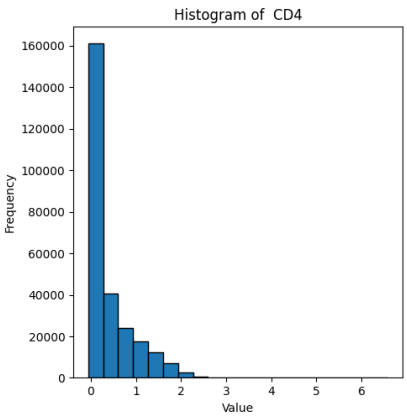

Histogram of CD45RA
Frequency
Value

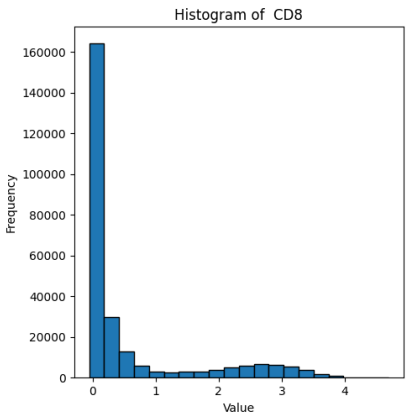

Histogram of CD133
Frequency
Value

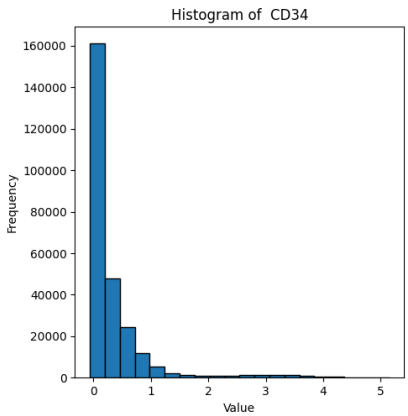

Histogram of CD19
Frequency
Value


Histogram of CD22
Frequency
Value


Histogram of CD11b
Frequency
Value


Histogram of CD4
Frequency
Value


Histogram of CD8
Frequency
Value


Histogram of CD34
Frequency
Value

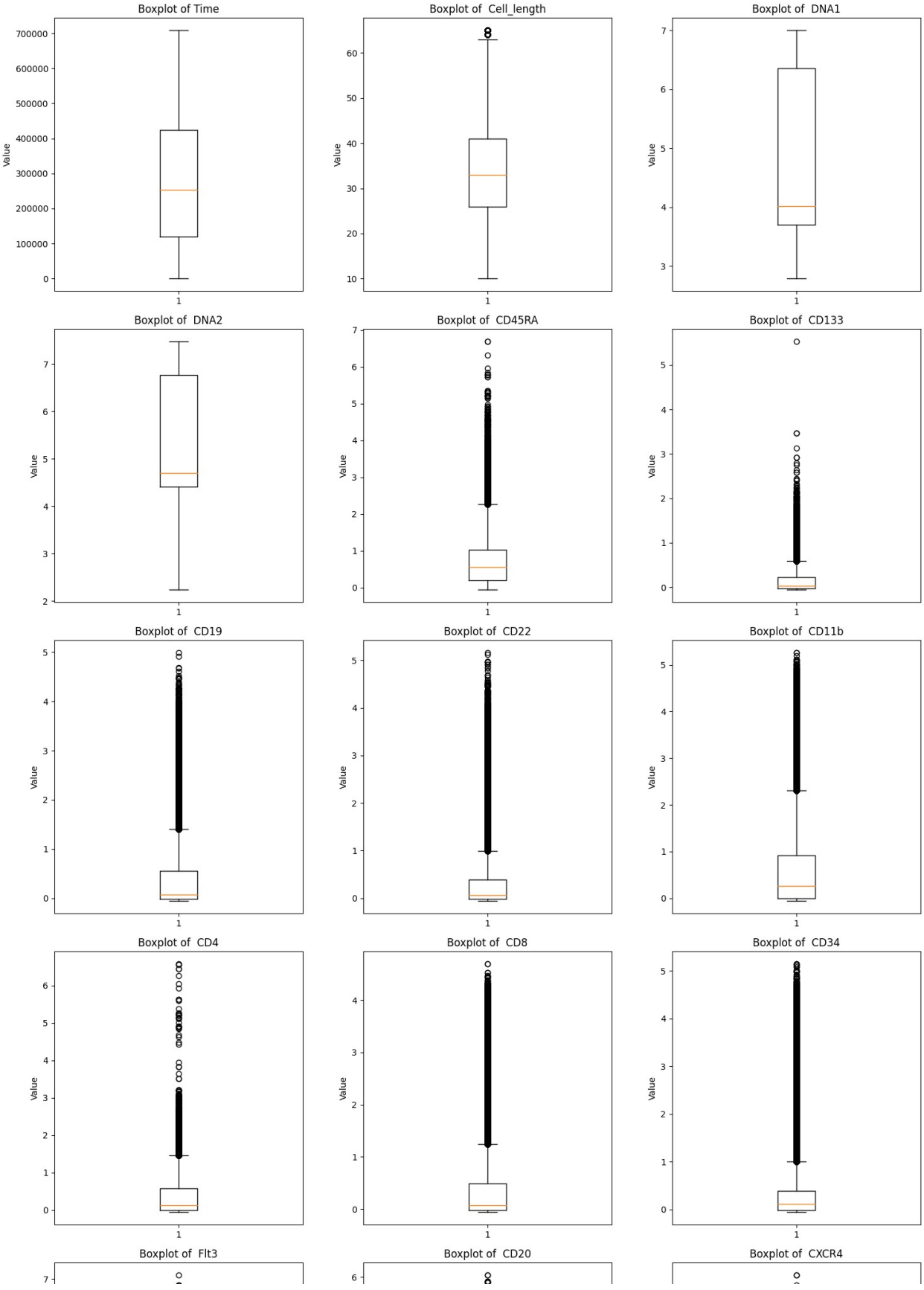

Histogram of Flt3


Histogram of CD20


Histogram of CXCR4

This code plots the box plot for all the values (after dropping certain irrelevant columns).

```
# Convert all columns to numeric, replacing errors with NaN
df = df.apply(pd.to_numeric, errors='coerce')

# Replace any remaining empty strings or NaN with 0
df = df.fillna(0)

# Create subplots for each column
num_cols = len(df.columns)
num_rows = (num_cols + 2) // 3  # Calculate number of rows for
subplots

fig, axes = plt.subplots(num_rows, 3, figsize=(15, 5 * num_rows))  #
Adjust figsize as needed
axes = axes.flatten()  # Flatten the axes array for easier iteration

for i, col in enumerate(df.columns):
    axes[i].boxplot(df[col])
    axes[i].set_title(f'Boxplot of {col}')
    axes[i].set_ylabel('Value')

# Hide any unused subplots
for j in range(i + 1, len(axes)):
    axes[j].axis('off')

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


Boxplot of Time
Boxplot of Cell_length
Boxplot of DNA1
Boxplot of DNA2
Boxplot of CD45RA
Boxplot of CD133
Boxplot of CD19
Boxplot of CD22
Boxplot of CD11b
Boxplot of CD4
Boxplot of CD8
Boxplot of CD34
Boxplot of Flt3
Boxplot of CD20
Boxplot of CXCR4
Value
1

This code prints the range after removing 'file_number', 'Event', 'Time', 'event_number'

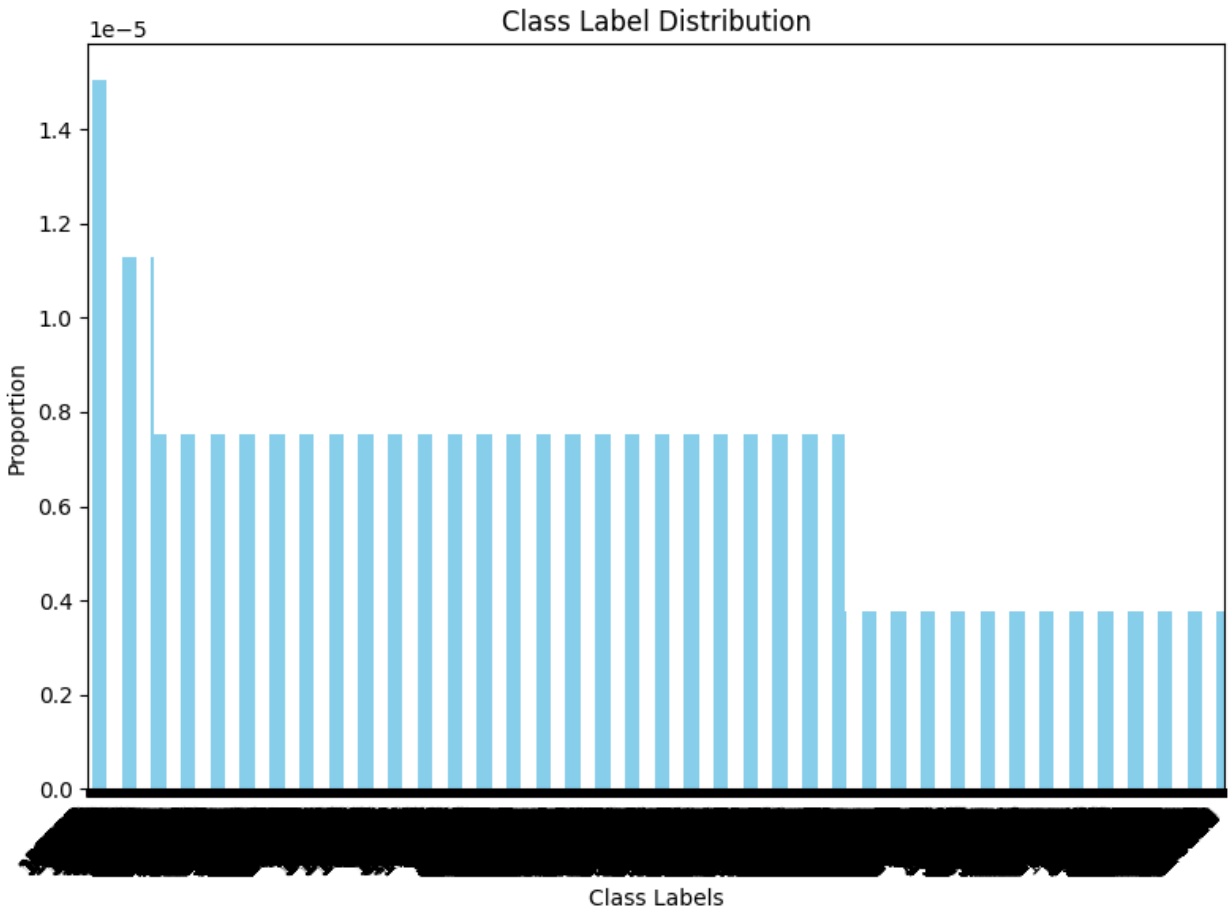
```
ranges = df.max()-df.min()

range_df=pd.DataFrame({'Min': df.min(), 'Max': df.max(), 'Range':
ranges})

print(range_df)
```

```
                      Min             Max          Range
Time             1.000000  708585.437500  708584.437500
 Cell_length    10.000000      65.000000      55.000000
 DNA1            2.786488       7.001489       4.215002
 DNA2            2.236450       7.472308       5.235858
 CD45RA         -0.057305       6.691197       6.748502
 CD133          -0.058081       5.527494       5.585575
 CD19           -0.058089       4.990085       5.048174
 CD22           -0.057342       5.160477       5.217819
 CD11b          -0.058236       5.260789       5.319025
 CD4            -0.057751       6.581762       6.639513
 CD8            -0.058003       4.693694       4.751697
 CD34           -0.058008       5.147996       5.206005
 Flt3           -0.057884       7.117323       7.175207
 CD20           -0.058132       6.051411       6.109544
 CXCR4          -0.057042       5.696674       5.753717
 CD235ab        -0.057612       6.646699       6.704311
 CD45            2.040243       7.238076       5.197833
 CD123          -0.058003       6.640626       6.698630
 CD321          -0.053552       6.867388       6.920940
 CD14           -0.057954       5.006121       5.064075
 CD33           -0.058079       5.612469       5.670548
 CD47           -0.055087       6.402488       6.457576
 CD11c          -0.058053       6.520939       6.578992
 CD7            -0.058162       6.319219       6.377381
 CD15           -0.058077       1.534151       1.592227
 CD16           -0.057780       5.338305       5.396085
 CD44            0.026061       7.404564       7.378503
 CD38           -0.057194       7.293085       7.350279
 CD13           -0.057728       6.981187       7.038915
 CD3            -0.058241       6.748363       6.806603
 CD61           -0.057642       7.748497       7.806139
 CD117          -0.057668       5.502125       5.559793
 CD49d          -0.058064       5.153438       5.211502
 HLA-DR         -0.057974       7.052507       7.110481
 CD64           -0.058199       4.517843       4.576042
 CD41           -0.058244       7.718288       7.776532
 Viability      -0.057979       2.433031       2.491010
 file_number     3.627711       3.669327       0.041616
 event_number    1.000000  400112.000000  400111.000000
 label           0.000000      14.000000      14.000000
```

```
individual     1.000000       2.000000       1.000000
               0.000000       0.000000       0.000000
```

Class Label Distribution

```
target_variable = 'event_number'  # Replace with your actual target
variable column name

# Get the class distribution
class_distribution = df[target_variable].value_counts(normalize=True)

# Plotting the distribution
plt.figure(figsize=(8, 6))
class_distribution.plot(kind='bar', color='skyblue')
plt.title('Class Label Distribution')
plt.xlabel('Class Labels')
plt.ylabel('Proportion')
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels for better
readability
plt.tight_layout()
plt.show()

# Print the class distribution as a table
print("Class Distribution:\n", class_distribution)
```



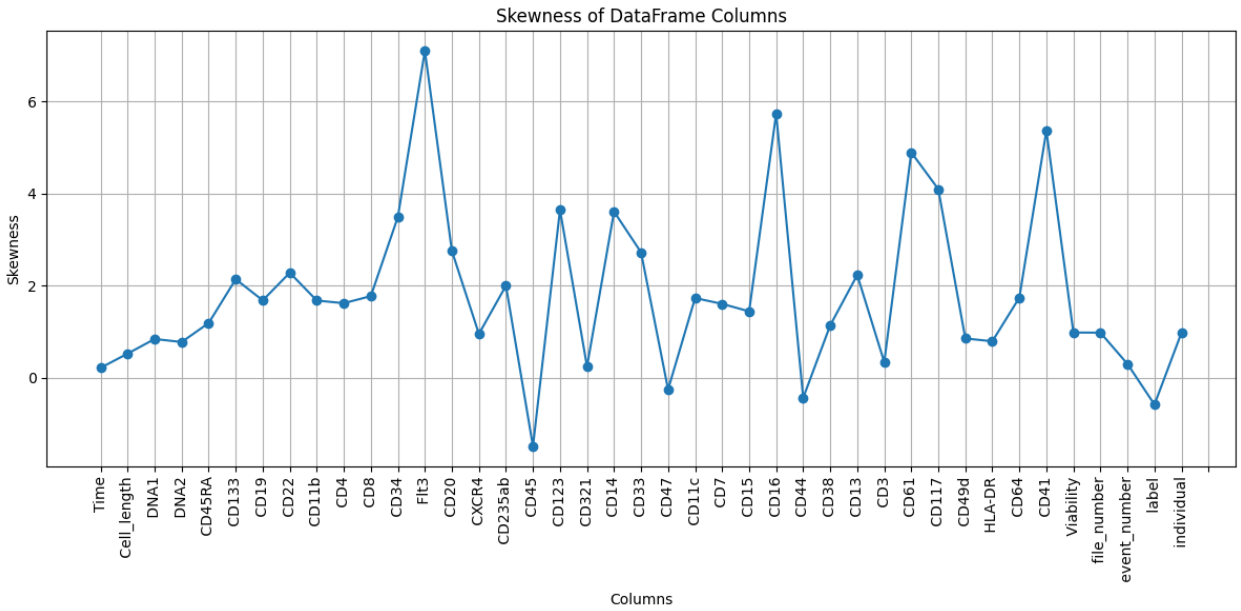

```
Class Distribution:
 event_number
307       0.000015
9207      0.000015
10882     0.000015
10753     0.000015
10710     0.000015
             ...
197271    0.000004
197279    0.000004
197297    0.000004
197299    0.000004
102686    0.000004
Name: proportion, Length: 151612, dtype: float64
```

This code plots the skewness at each point using a Line Graph

```
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import skew
```

```python
# 1. Data Type Conversion (Important!)
for col in df.columns:
    df[col] = pd.to_numeric(df[col], errors='coerce')

# 2. Calculate Skewness
skewness_values = df.apply(lambda x: skew(x, nan_policy='omit'))

# 3. Create DataFrame for Plotting
skewness_df = pd.DataFrame({'Column': df.columns, 'Skewness':
skewness_values})

# 4. Plot Line Graph
plt.figure(figsize=(12, 6))  # Adjust figure size as needed
plt.plot(skewness_df['Column'], skewness_df['Skewness'], marker='o',
linestyle='-')
plt.title('Skewness of DataFrame Columns')
plt.xlabel('Columns')
plt.ylabel('Skewness')
plt.xticks(rotation=90)  # Rotate x-axis labels for better readability
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/scipy/stats/_stats_py.py:1177:
RuntimeWarning: Mean of empty slice.
  mean = a.mean(axis, keepdims=True)
/usr/local/lib/python3.10/dist-packages/numpy/core/_methods.py:121:
RuntimeWarning: invalid value encountered in divide
  ret = um.true_divide(
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:3504
: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/usr/local/lib/python3.10/dist-packages/numpy/core/_methods.py:129:
RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```



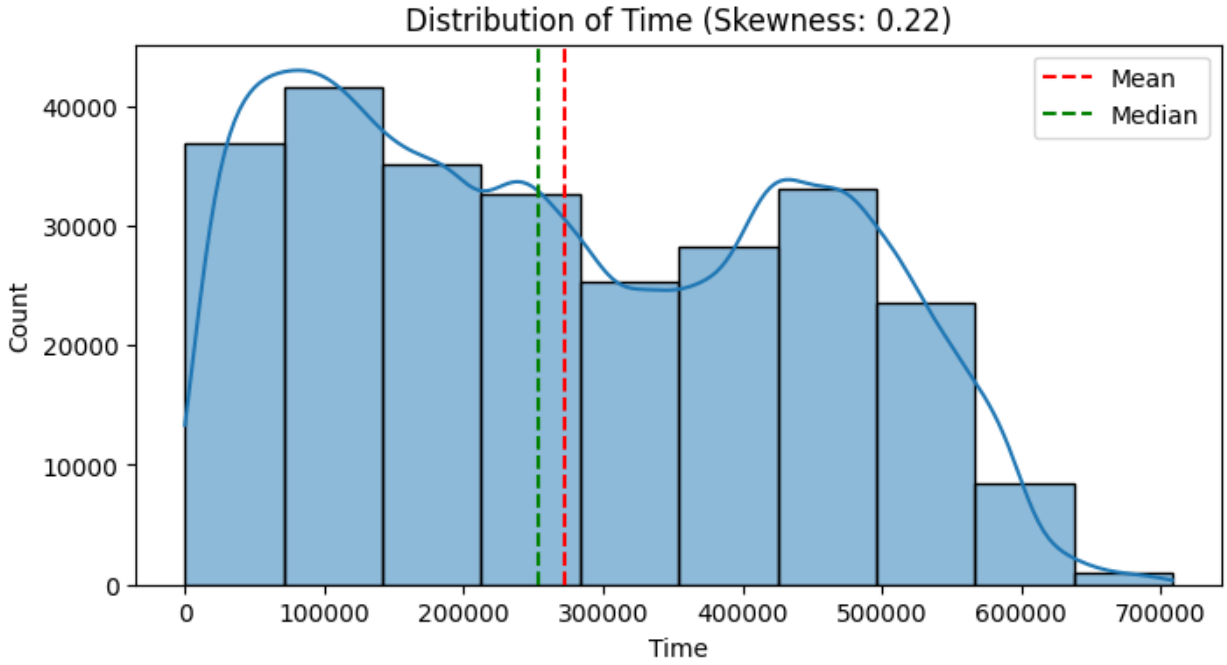
This code plots the skewness at each point using Histograms

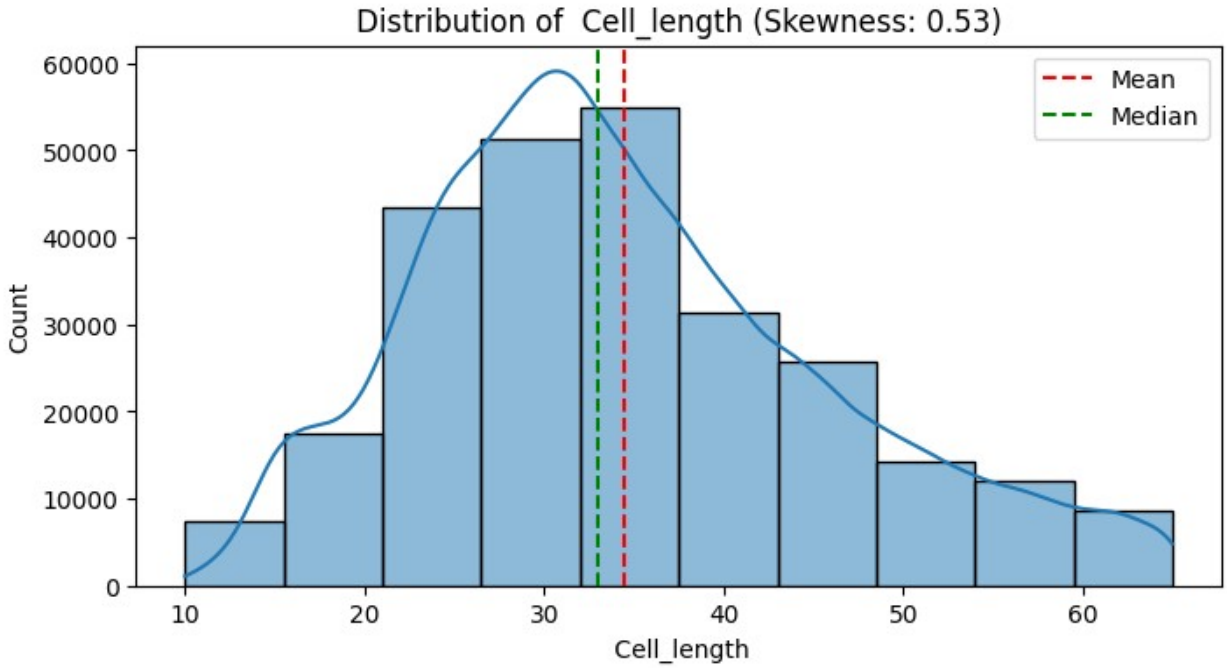
```
import pandas as pd
from scipy.stats import skew
import numpy as np  # Import NumPy with the alias 'np'
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import skew

# Select only numerical columns for skewness calculation
numerical_cols = df.select_dtypes(include=np.number).columns
skewness = df[numerical_cols].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
```

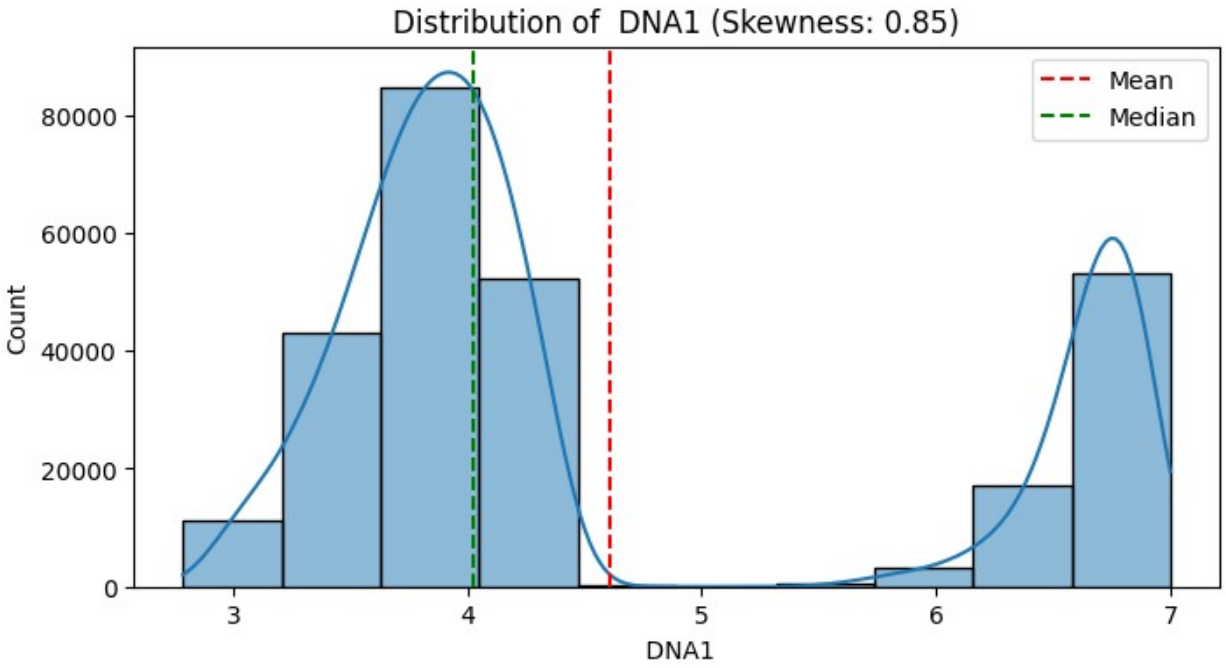
```
for col in numerical_cols: # Iterate through numerical columns only
    plt.figure(figsize=(8, 4))
    sns.histplot(df[col], bins=10, kde=True)
    plt.title(f'Distribution of {col} (Skewness:
{skewness[col]:.2f})')
    plt.axvline(df[col].mean(), color='red', linestyle='--',
label='Mean')
    plt.axvline(df[col].median(),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

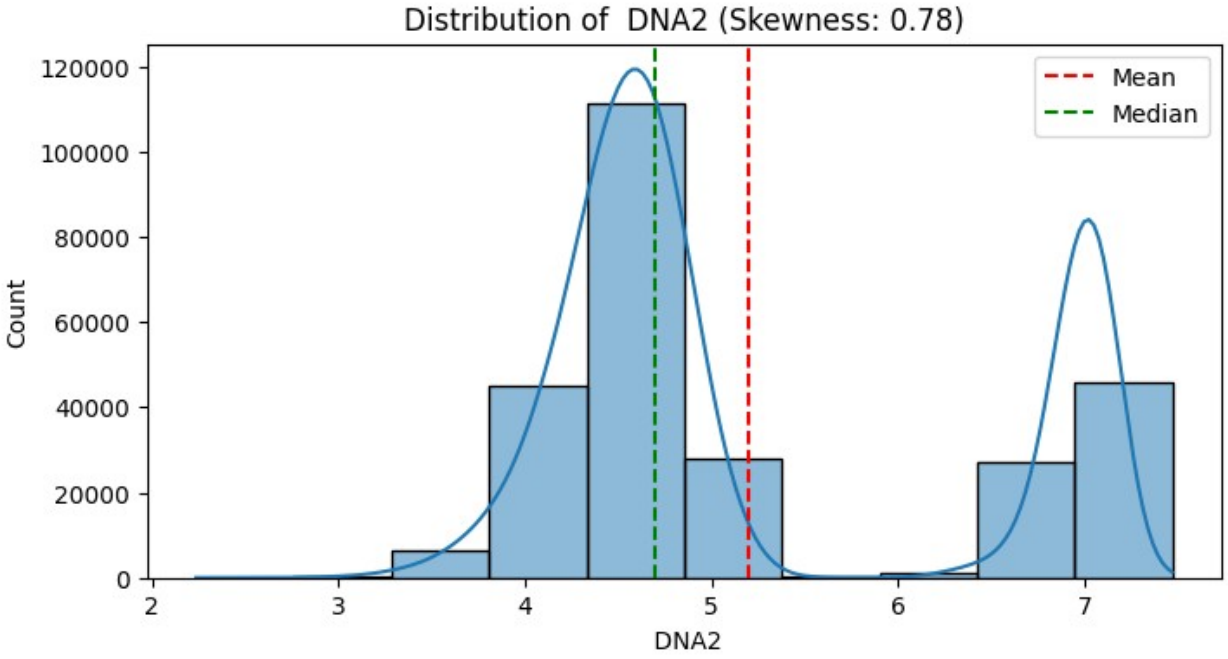
	Skewness	Category
Time	0.219929	Approximately symmetrical
Cell_length	0.527837	Right-skewed
DNA1	0.845023	Right-skewed
DNA2	0.779179	Right-skewed
CD45RA	1.191589	Right-skewed
CD133	2.141948	Right-skewed
CD19	1.682603	Right-skewed
CD22	2.283175	Right-skewed
CD11b	1.679099	Right-skewed
CD4	1.622054	Right-skewed
CD8	1.775707	Right-skewed
CD34	3.492430	Right-skewed
Flt3	7.098147	Right-skewed
CD20	2.754693	Right-skewed
CXCR4	0.955336	Right-skewed
CD235ab	2.001474	Right-skewed
CD45	-1.484825	Left-skewed
CD123	3.648884	Right-skewed
CD321	0.247103	Approximately symmetrical
CD14	3.609043	Right-skewed
CD33	2.724976	Right-skewed
CD47	-0.250331	Approximately symmetrical
CD11c	1.733879	Right-skewed
CD7	1.606522	Right-skewed
CD15	1.445142	Right-skewed
CD16	5.733193	Right-skewed
CD44	-0.431588	Approximately symmetrical
CD38	1.141479	Right-skewed
CD13	2.233992	Right-skewed
CD3	0.342233	Approximately symmetrical
CD61	4.894700	Right-skewed
CD117	4.097499	Right-skewed
CD49d	0.856803	Right-skewed
HLA-DR	0.795372	Right-skewed
CD64	1.743727	Right-skewed
CD41	5.366306	Right-skewed
Viability	0.985411	Right-skewed

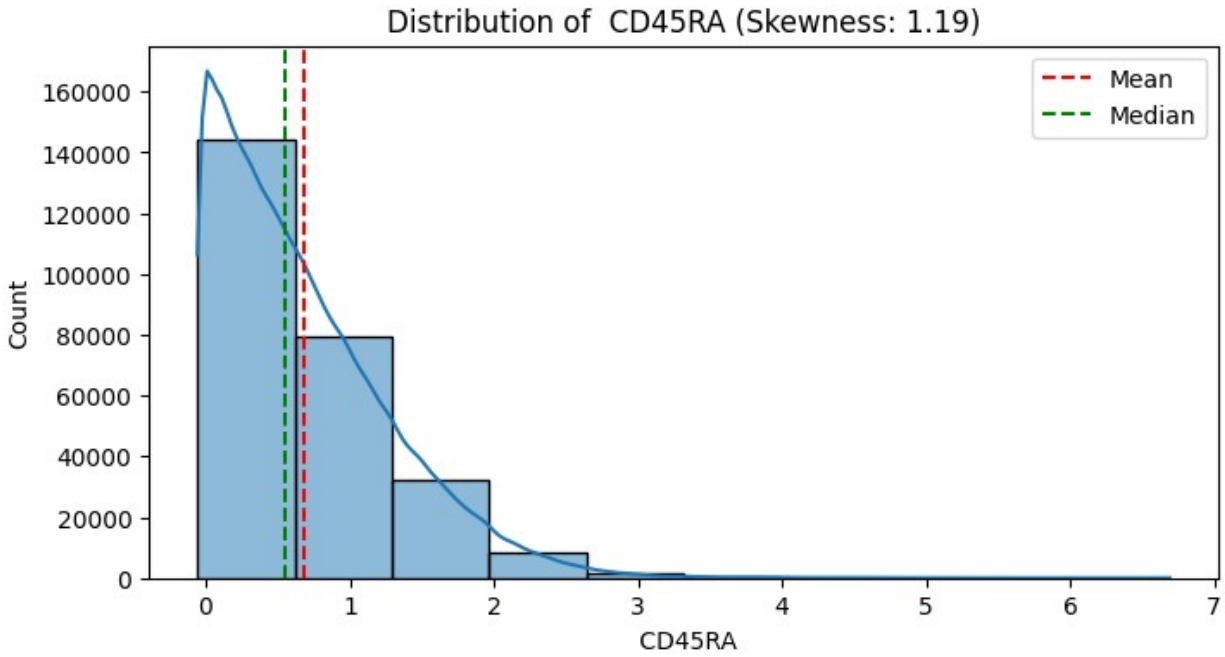
```
file_number   0.982045                Right-skewed
event_number  0.304111  Approximately symmetrical
individual    0.982045                Right-skewed
```

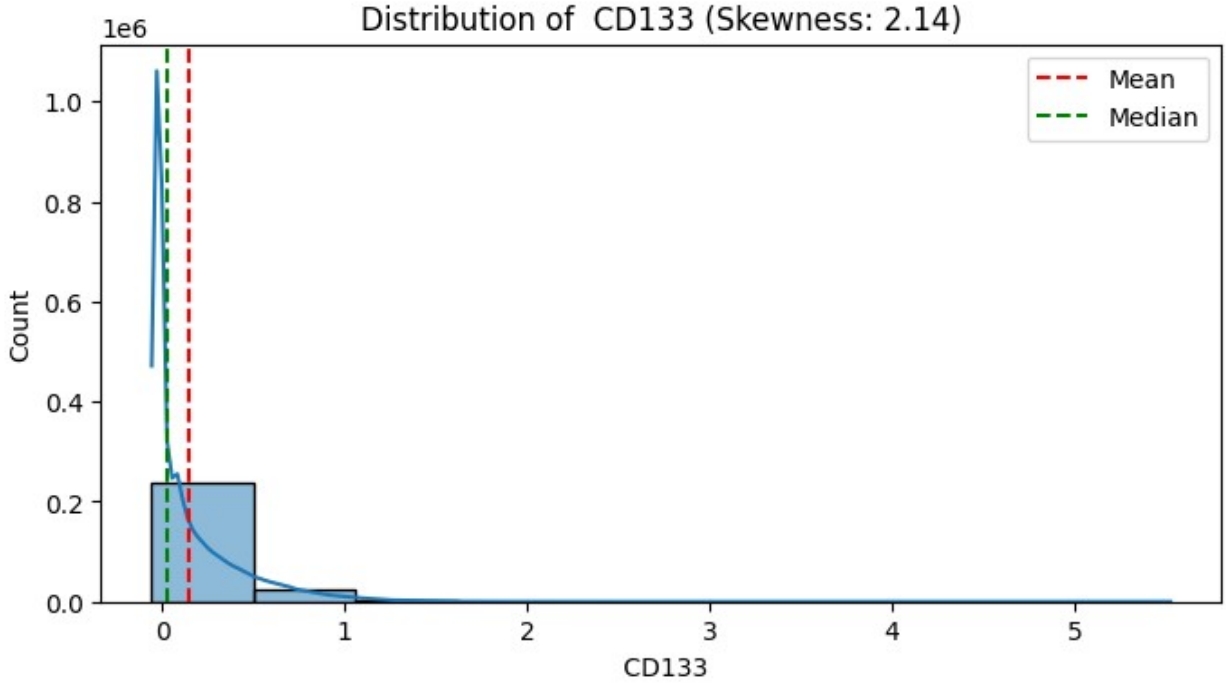


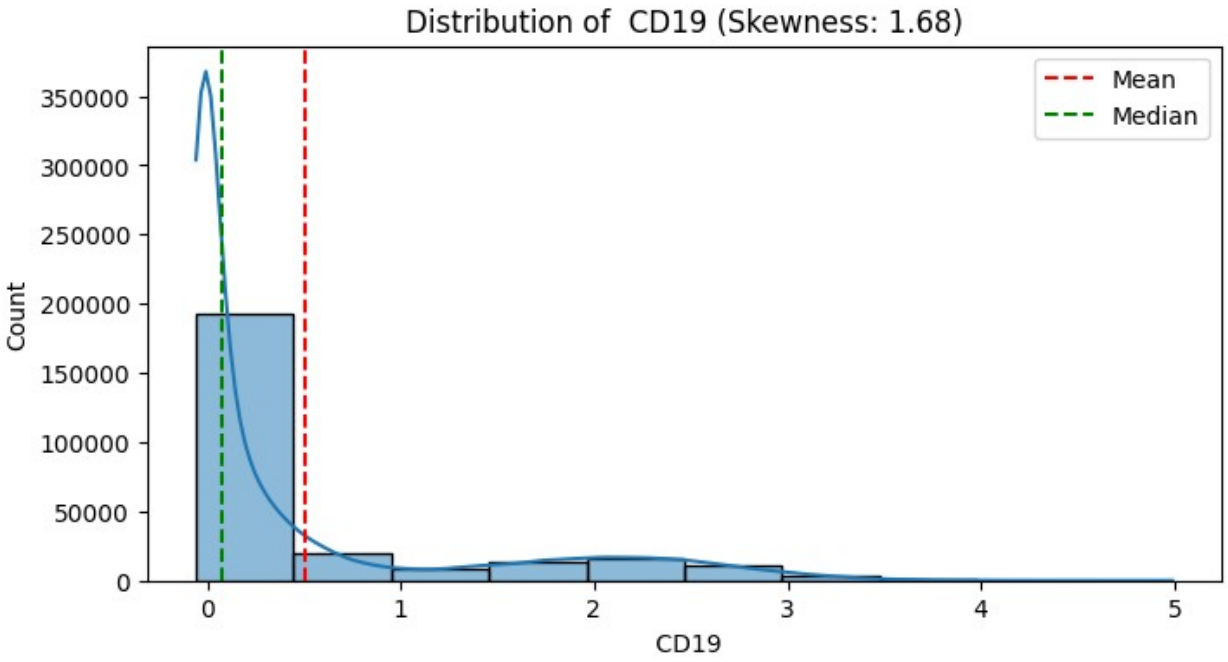


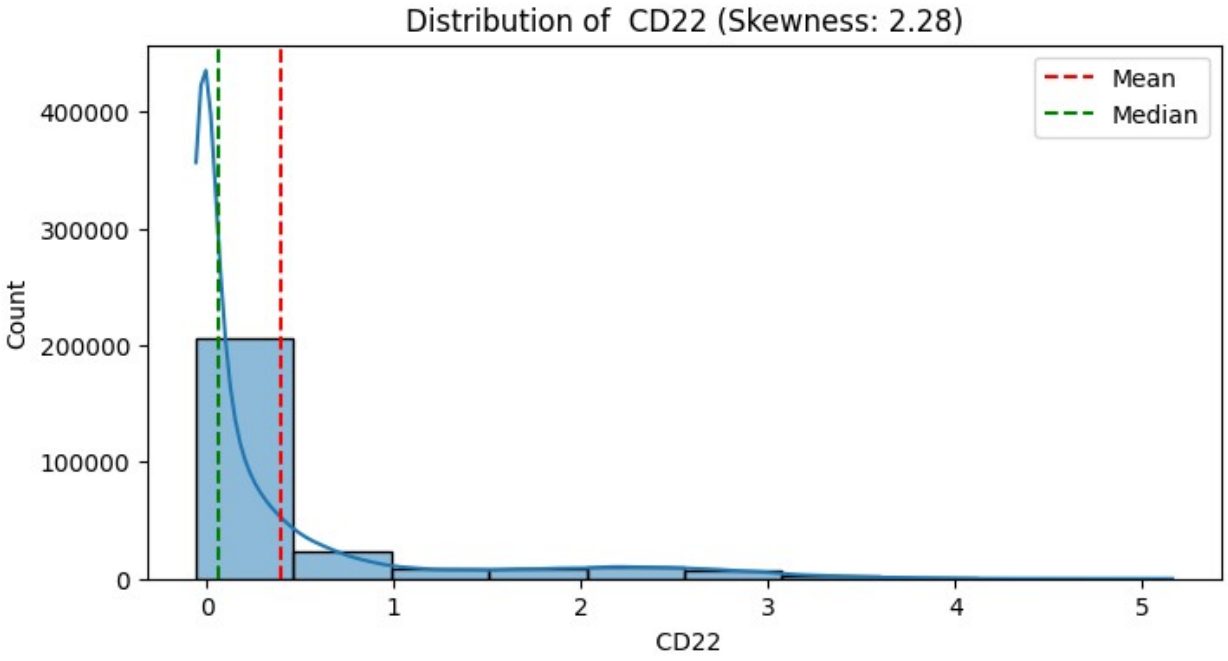

Distribution of DNA1 (Skewness: 0.85)
Mean
Median
Count
DNA1
0
20000
40000
60000
80000
3
4
5
6
7

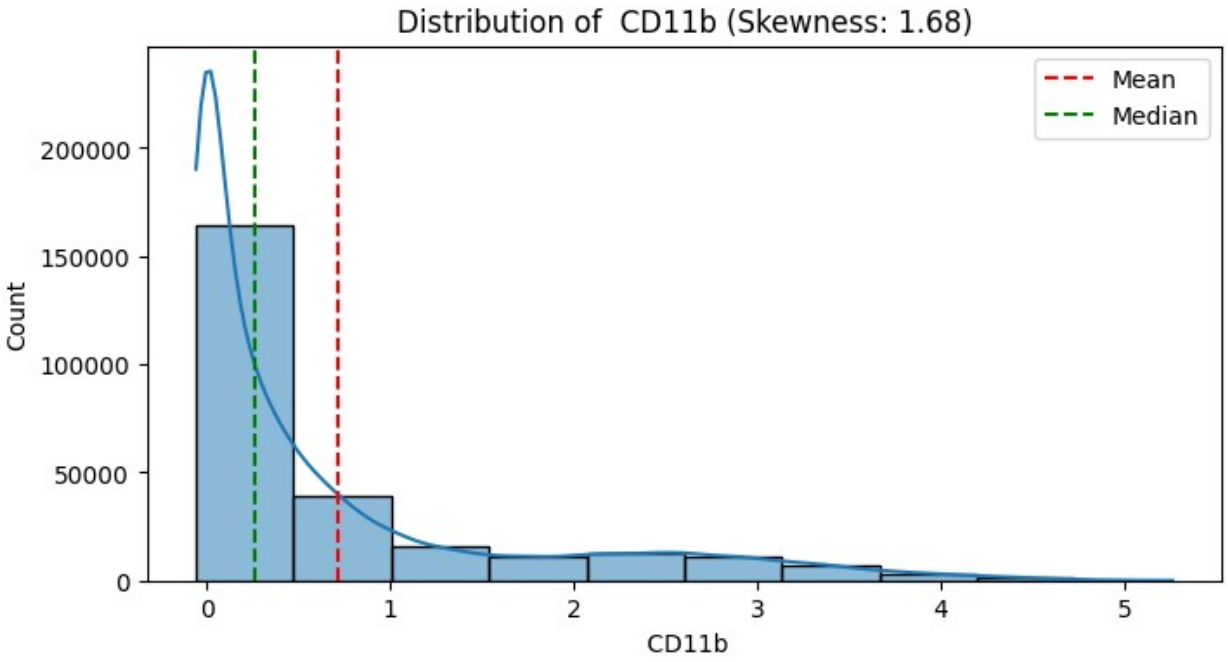

Distribution of DNA2 (Skewness: 0.78)
Mean
Median
Count
DNA2
0
20000
40000
60000
80000
100000
120000
2
3
4
5
6
7

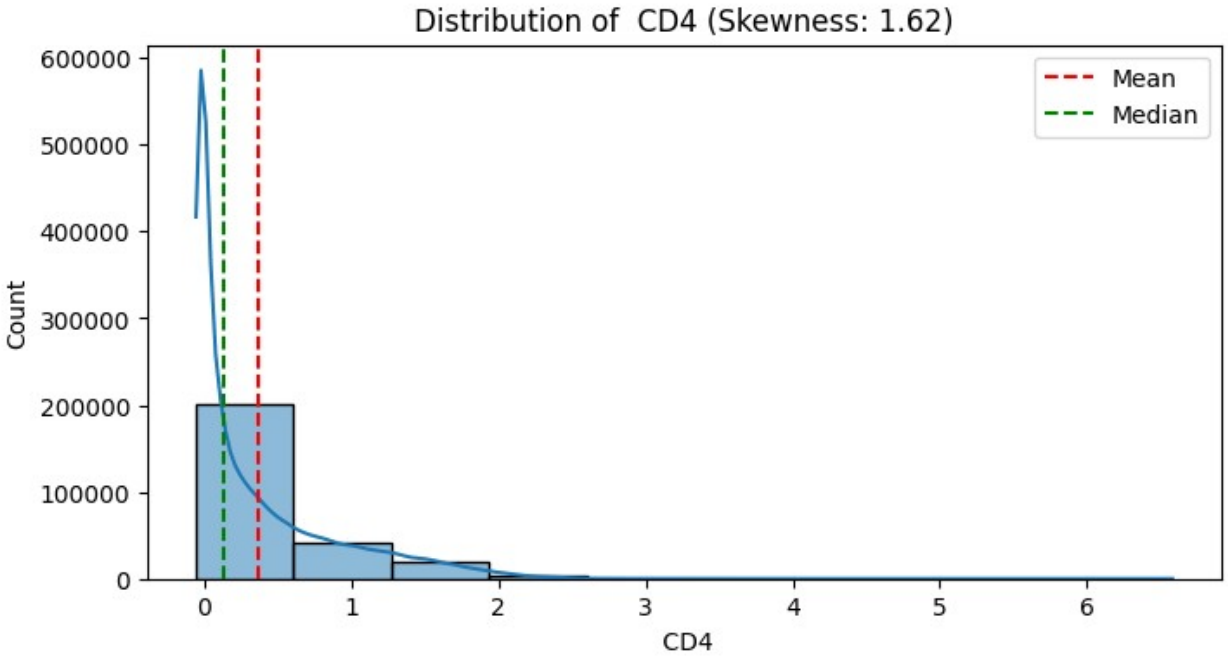

Distribution of CD45RA (Skewness: 1.19)
Mean
Median
Count
CD45RA
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6
7

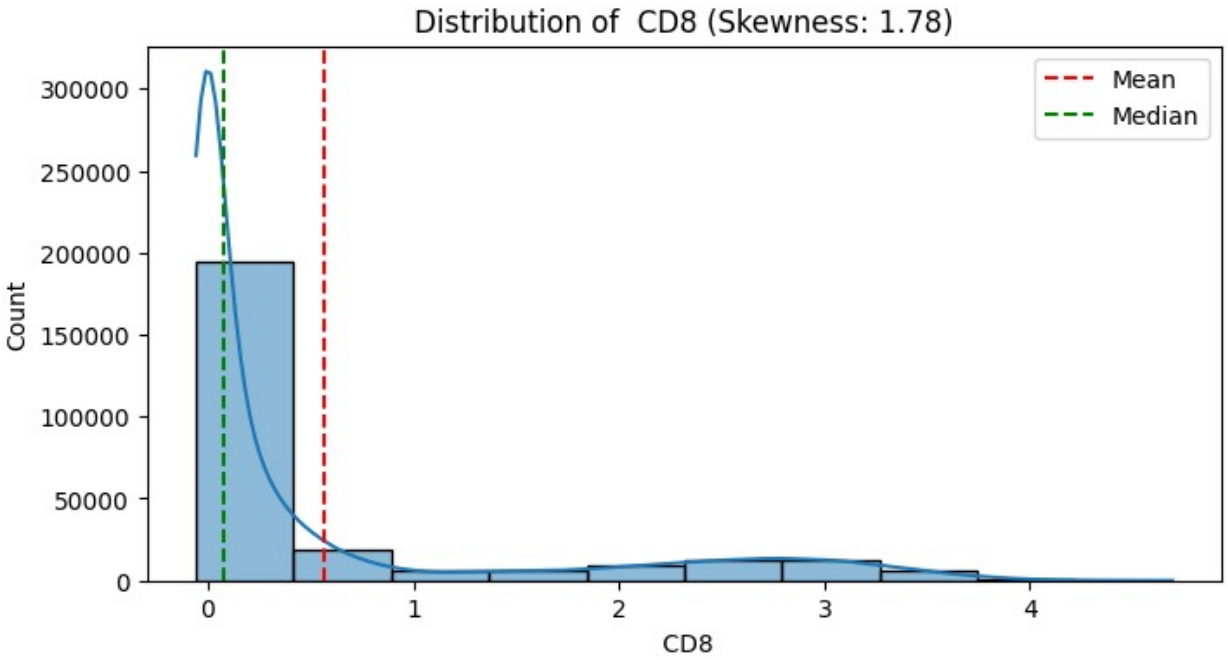

Distribution of CD133 (Skewness: 2.14)
1e6
Mean
Median
Count
CD133
0.0
0.2
0.4
0.6
0.8
1.0
0
1
2
3
4
5

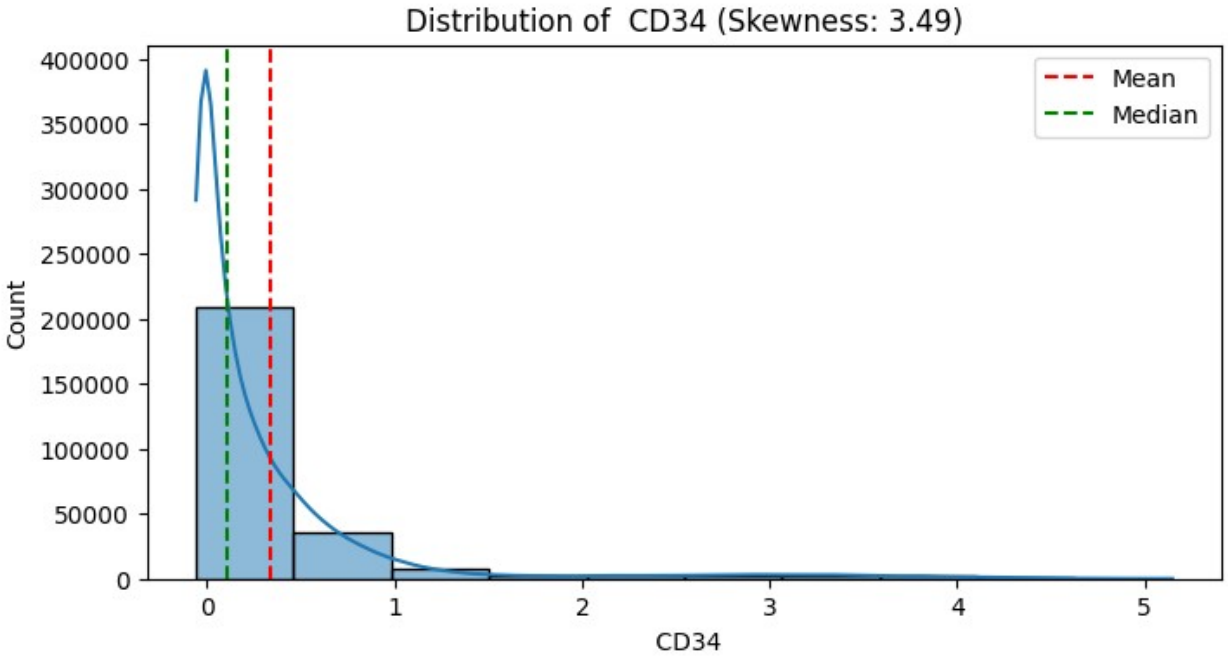

Distribution of CD19 (Skewness: 1.68)
Mean
Median
Count
CD19
0
50000
100000
150000
200000
250000
300000
350000
0
1
2
3
4
5

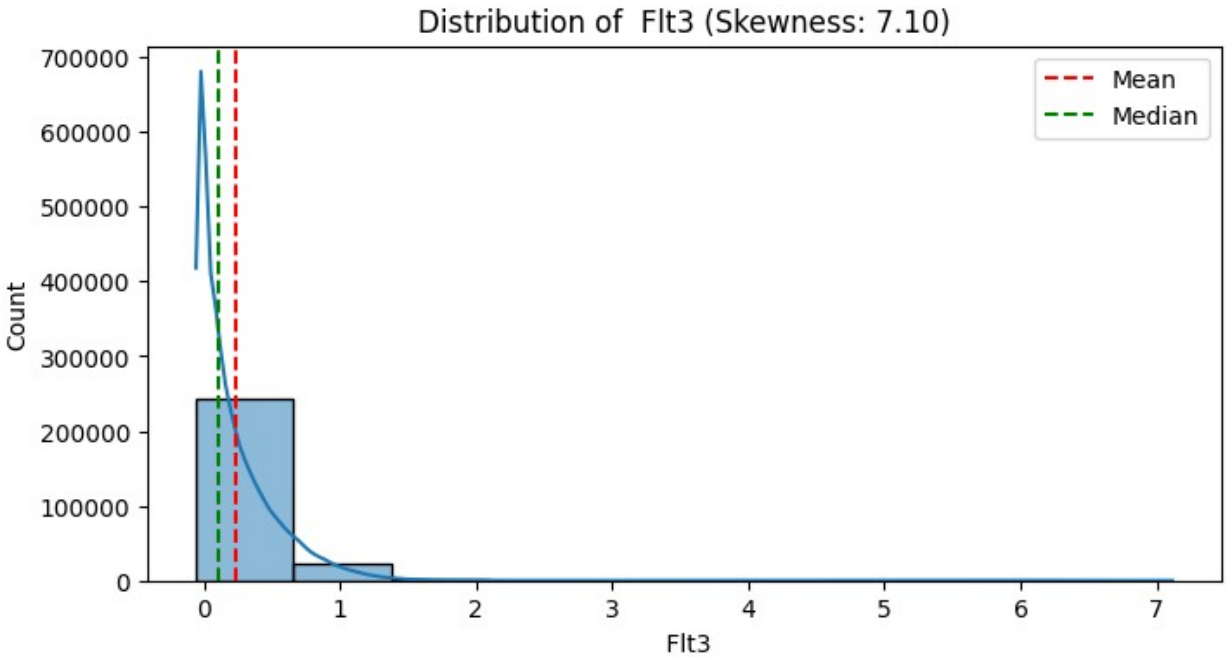

Distribution of CD22 (Skewness: 2.28)
Mean
Median
Count
CD22
0
100000
200000
300000
400000
0
1
2
3
4
5

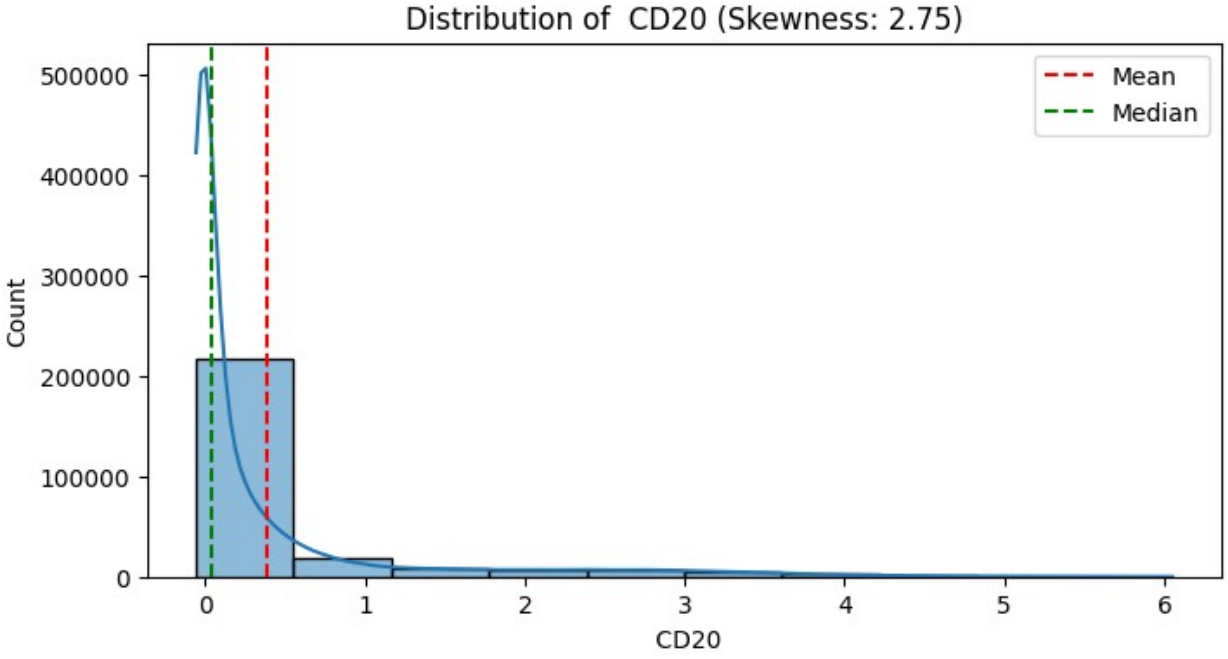

Distribution of CD11b (Skewness: 1.68)
Mean
Median
Count
CD11b
0
50000
100000
150000
200000
0
1
2
3
4
5

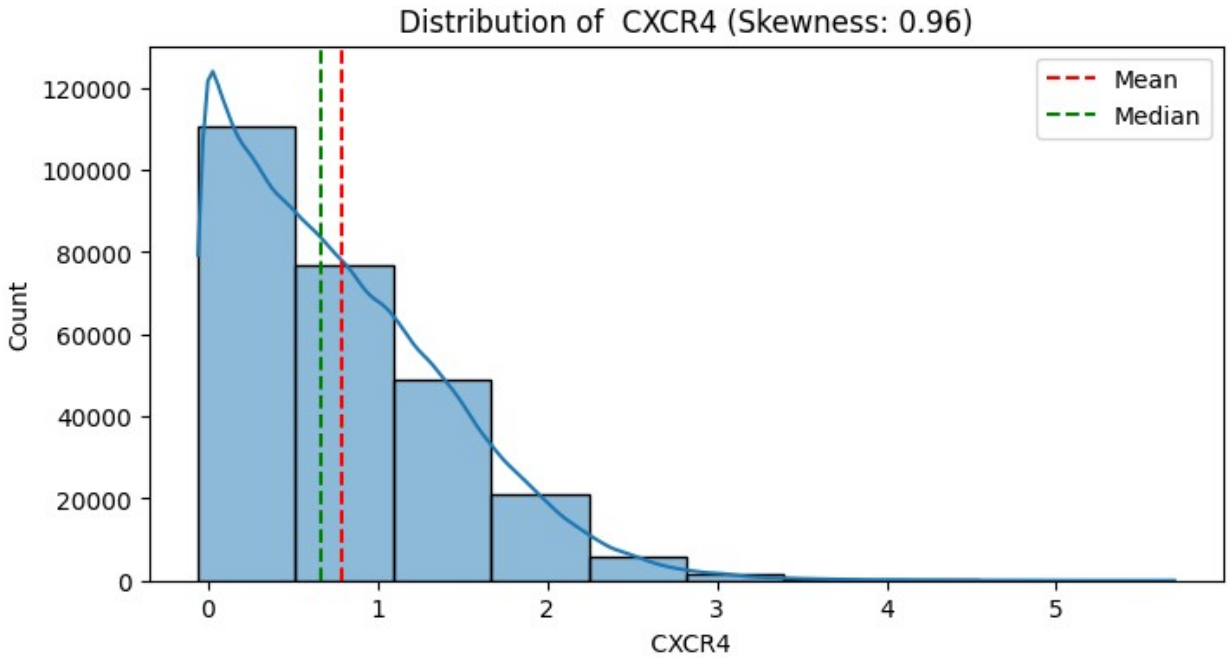

Distribution of CD4 (Skewness: 1.62)
Mean
Median
Count
CD4
0
100000
200000
300000
400000
500000
600000
0
1
2
3
4
5
6

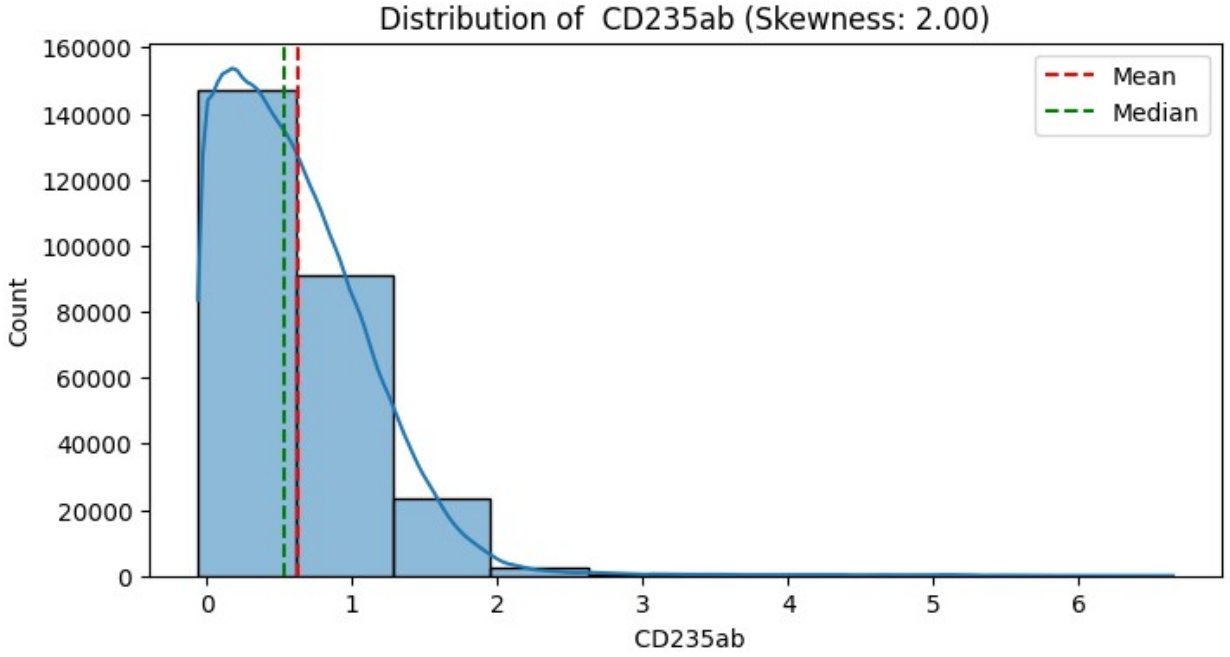

Distribution of CD8 (Skewness: 1.78)
Mean
Median
Count
CD8
0
50000
100000
150000
200000
250000
300000
0
1
2
3
4

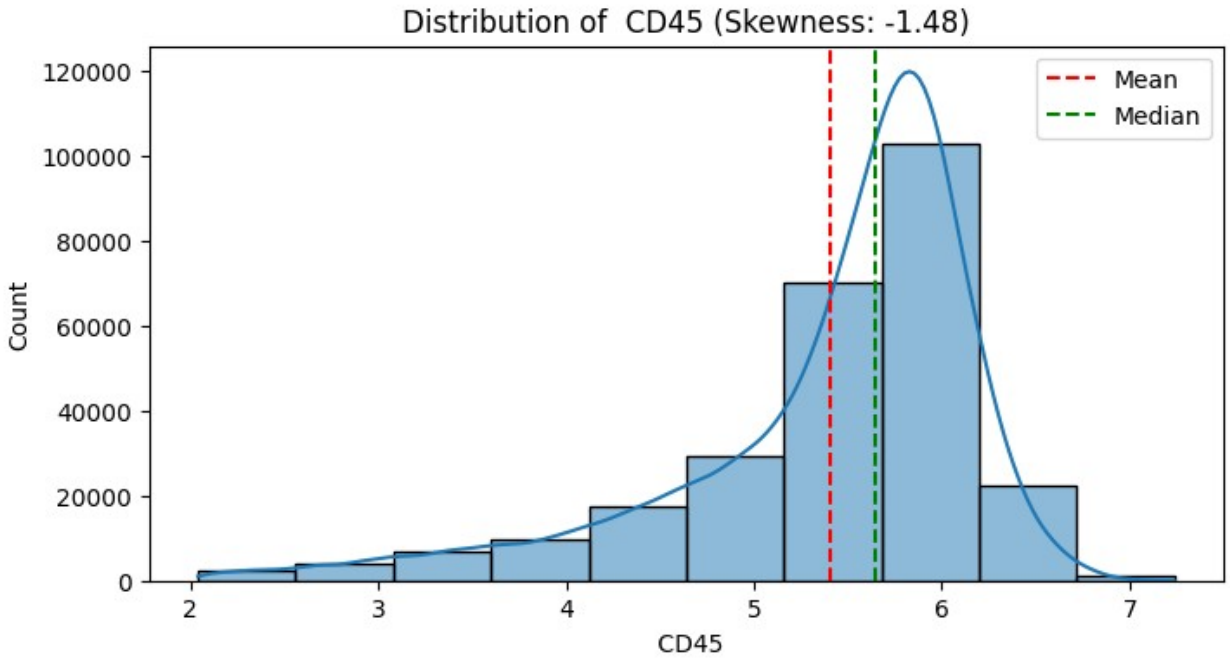

Distribution of CD34 (Skewness: 3.49)
Mean
Median
Count
CD34
0
50000
100000
150000
200000
250000
300000
350000
400000
0
1
2
3
4
5

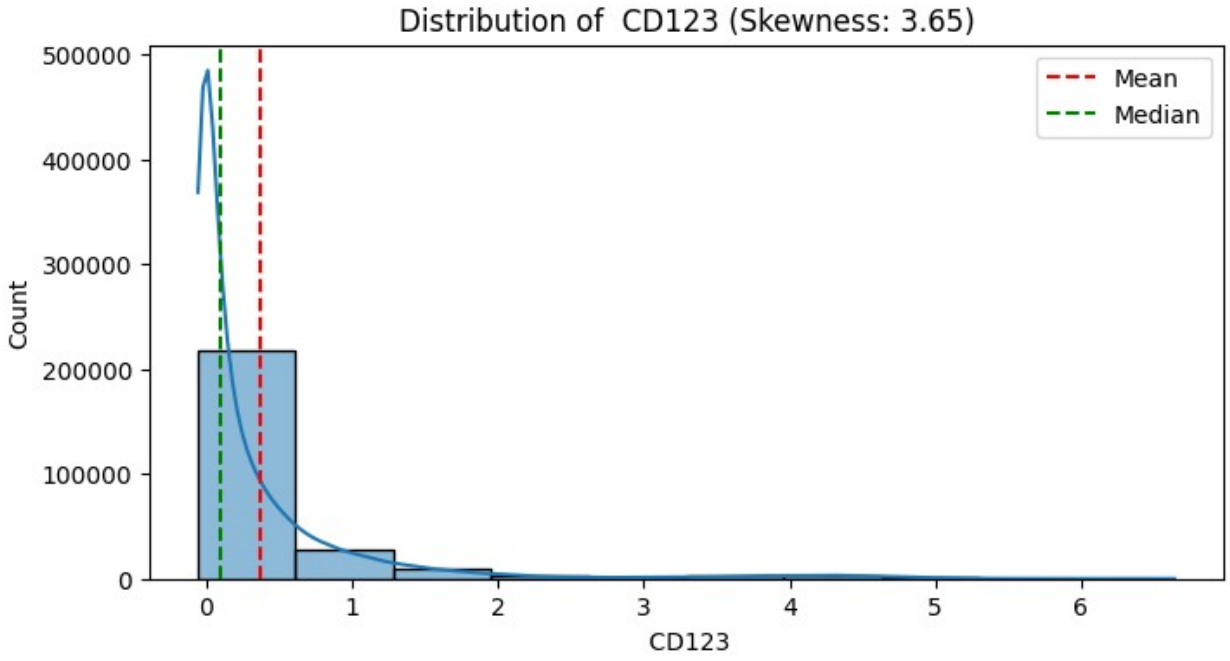

Distribution of Flt3 (Skewness: 7.10)
Mean
Median
Count
Flt3
700000
600000
500000
400000
300000
200000
100000
0
0
1
2
3
4
5
6
7

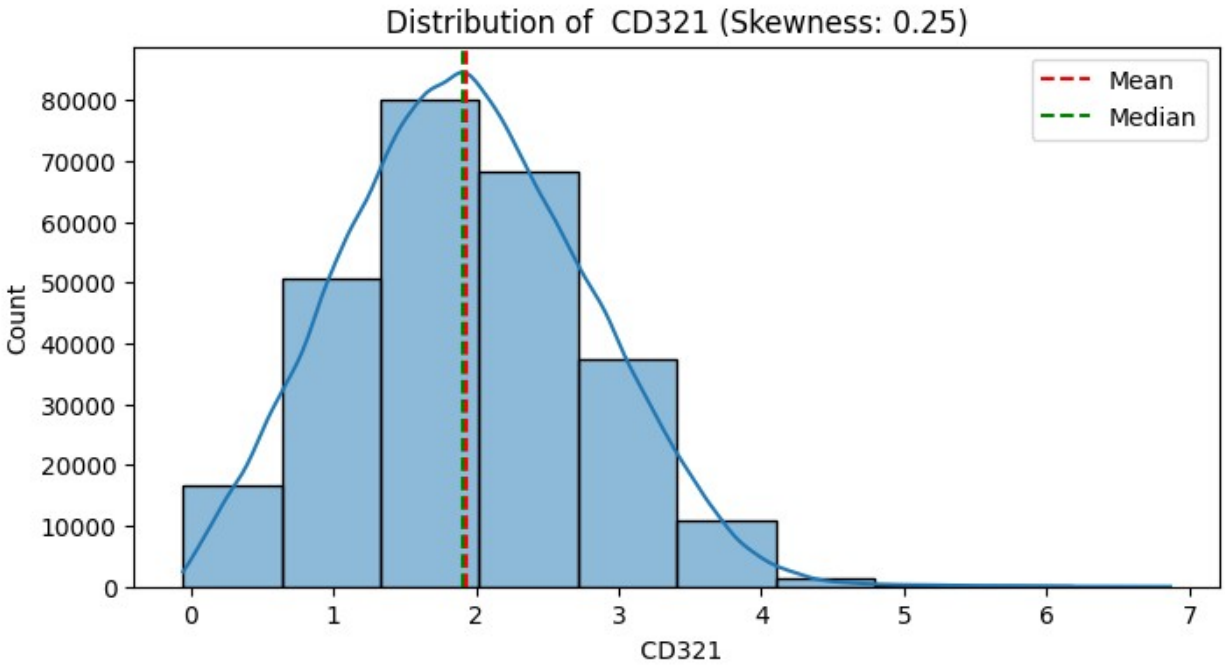

Distribution of CD20 (Skewness: 2.75)
Mean
Median
Count
CD20
500000
400000
300000
200000
100000
0
0
1
2
3
4
5
6

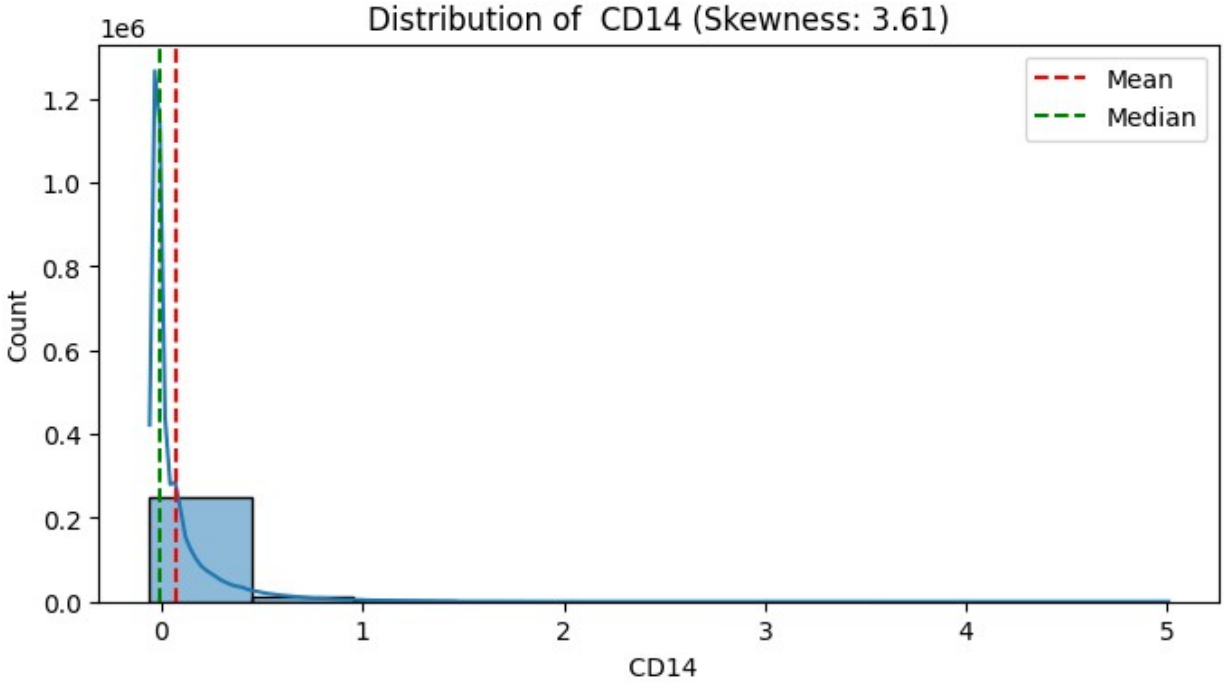

Distribution of CXCR4 (Skewness: 0.96)
Mean
Median
Count
CXCR4
0
20000
40000
60000
80000
100000
120000
0
1
2
3
4
5

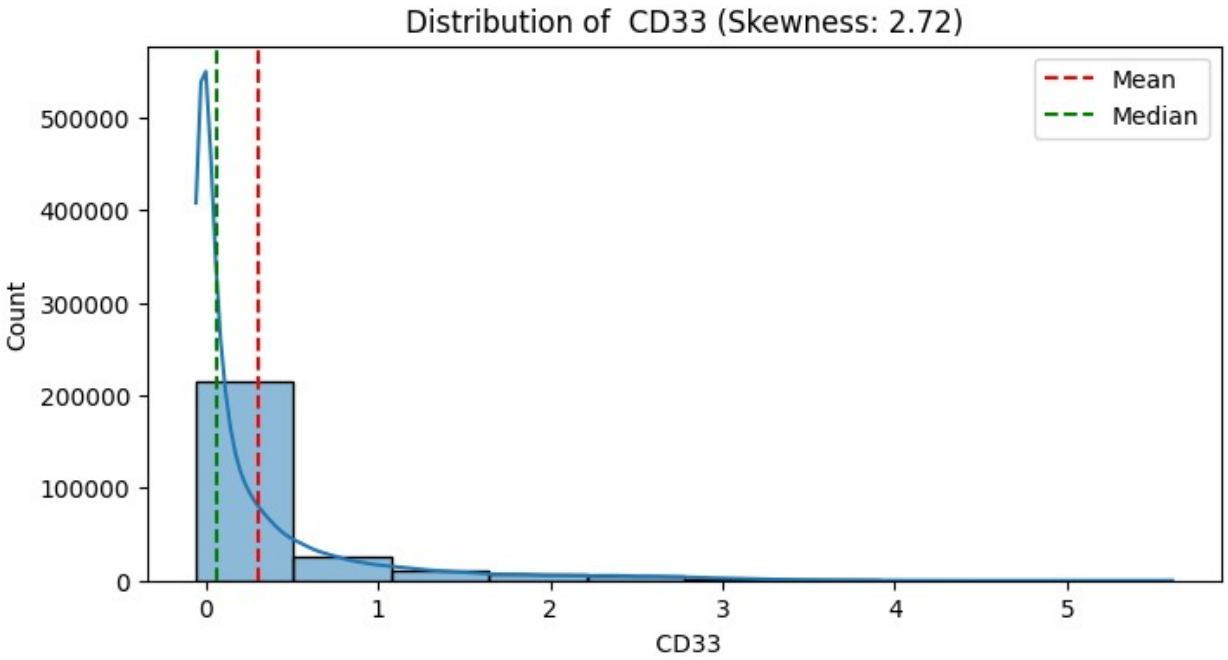

Distribution of CD235ab (Skewness: 2.00)
Mean
Median
Count
CD235ab
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6

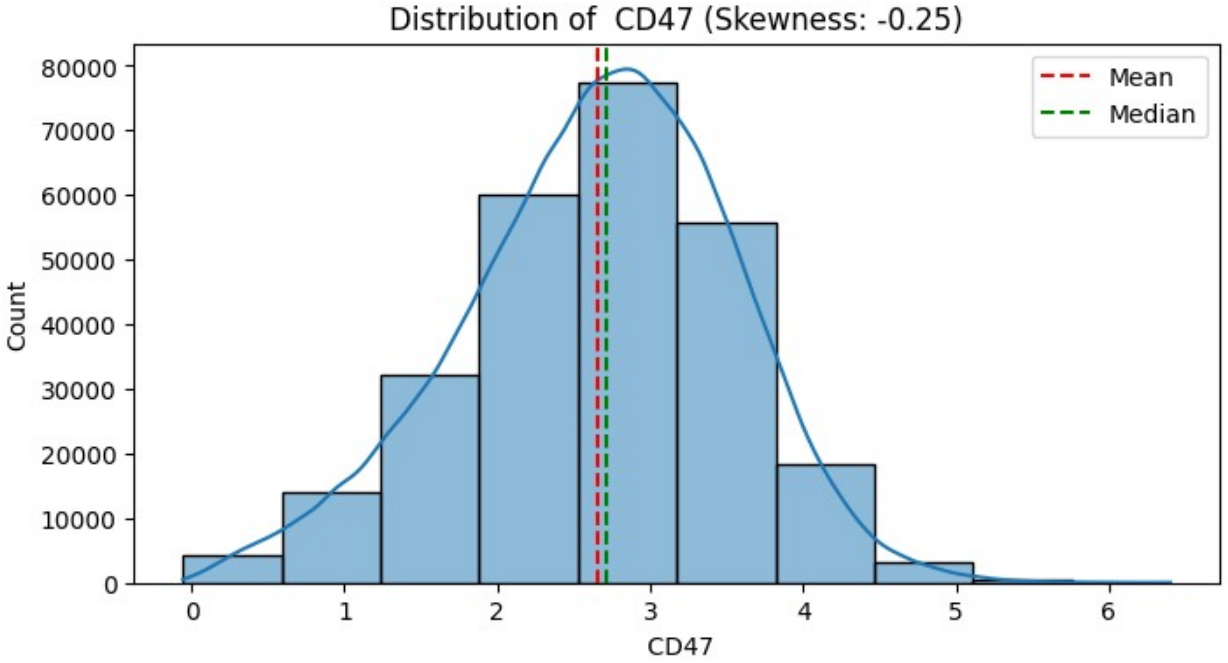

Distribution of CD45 (Skewness: -1.48)
Mean
Median
Count
CD45
0
20000
40000
60000
80000
100000
120000
2
3
4
5
6
7

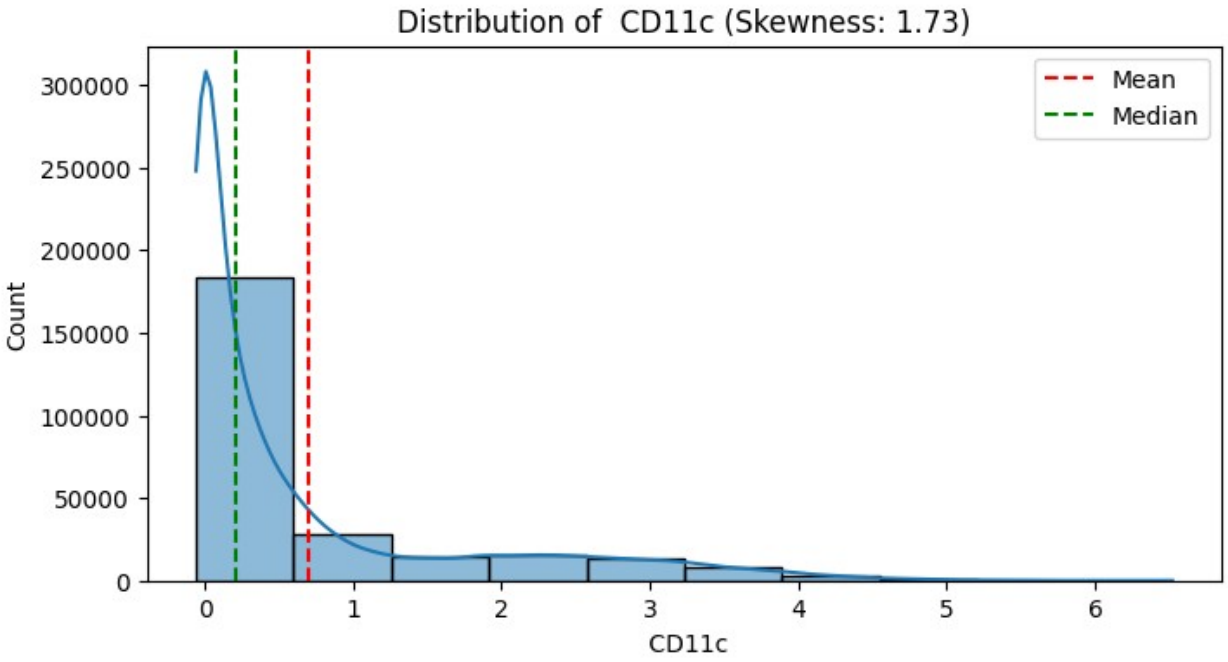

Distribution of CD123 (Skewness: 3.65)
Mean
Median
Count
CD123
0
100000
200000
300000
400000
500000
0
1
2
3
4
5
6

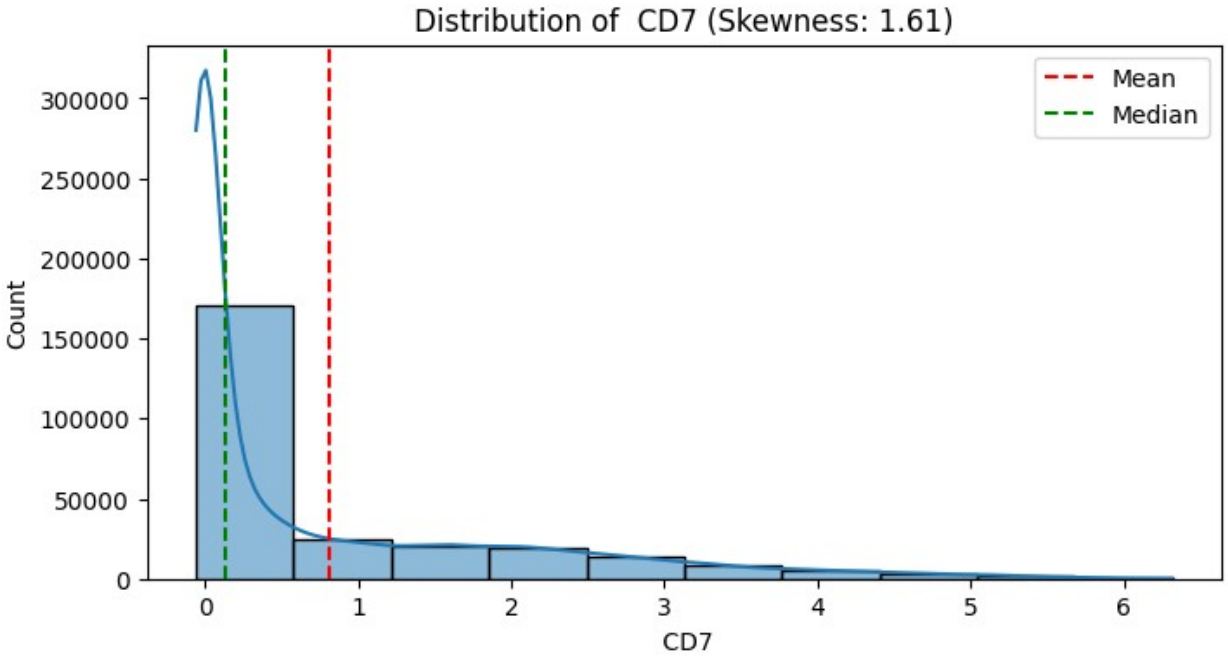

Distribution of CD321 (Skewness: 0.25)
Mean
Median
Count
CD321
0
10000
20000
30000
40000
50000
60000
70000
80000
0
1
2
3
4
5
6
7

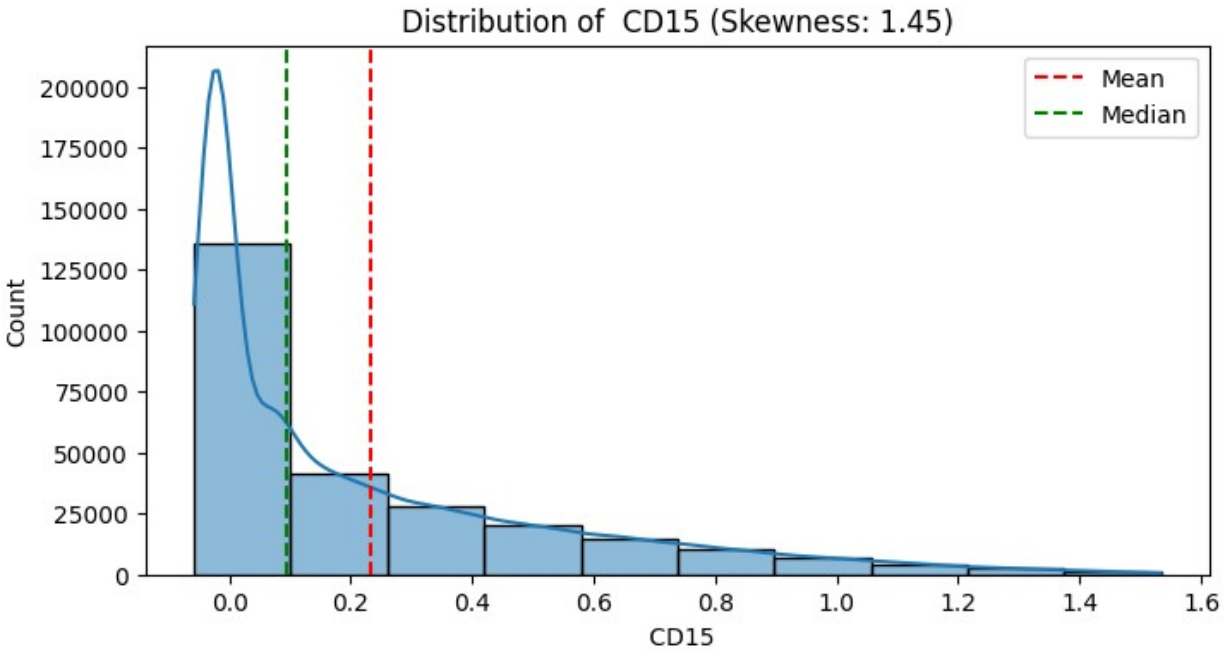

Distribution of CD14 (Skewness: 3.61)
1e6
Mean
Median
Count
CD14
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
1
2
3
4
5

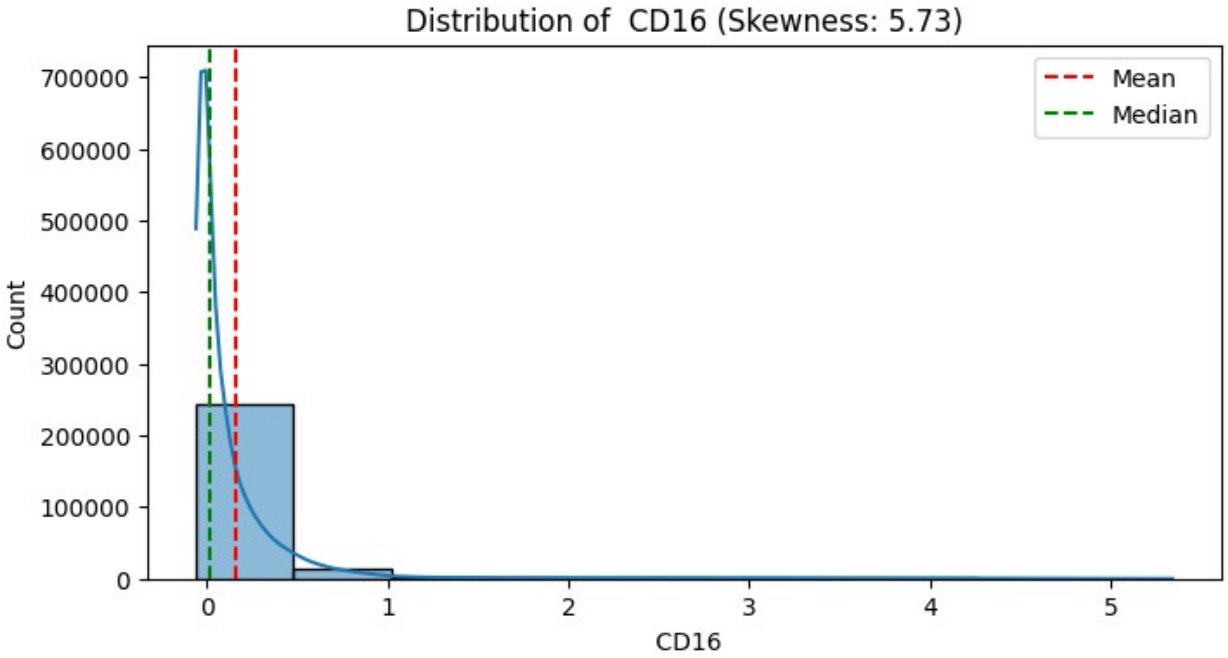

Distribution of CD33 (Skewness: 2.72)
Mean
Median
Count
CD33
0
100000
200000
300000
400000
500000
0
1
2
3
4
5

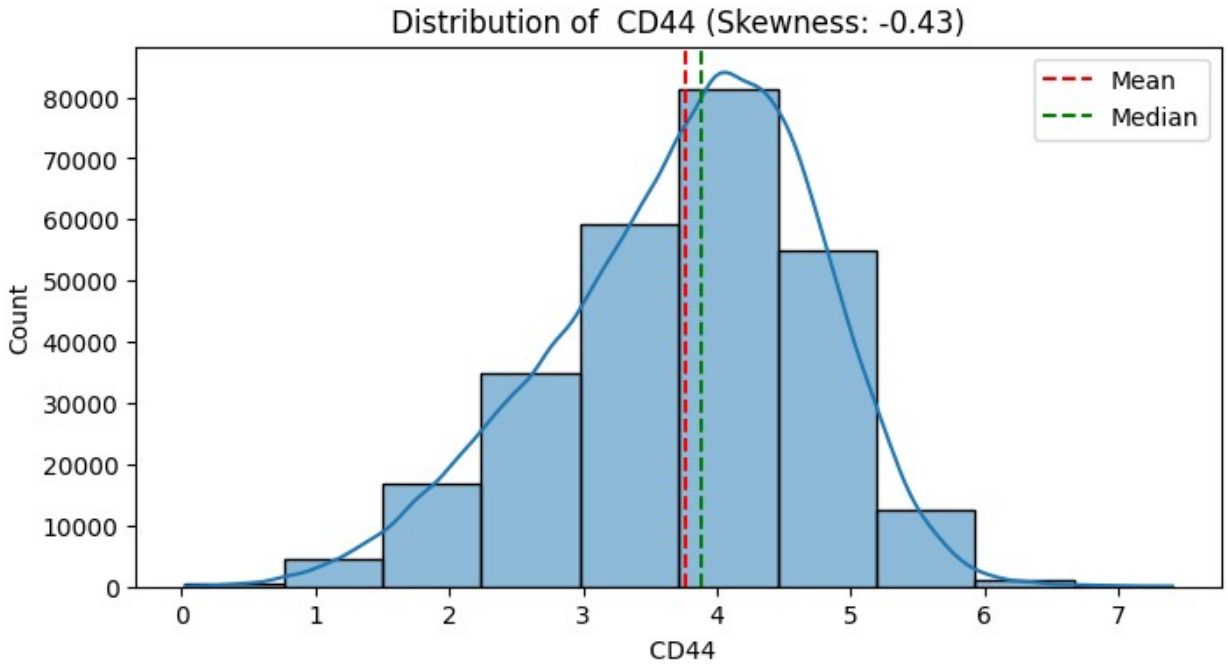

Distribution of CD47 (Skewness: -0.25)
Mean
Median
Count
CD47
0
10000
20000
30000
40000
50000
60000
70000
80000
0
1
2
3
4
5
6

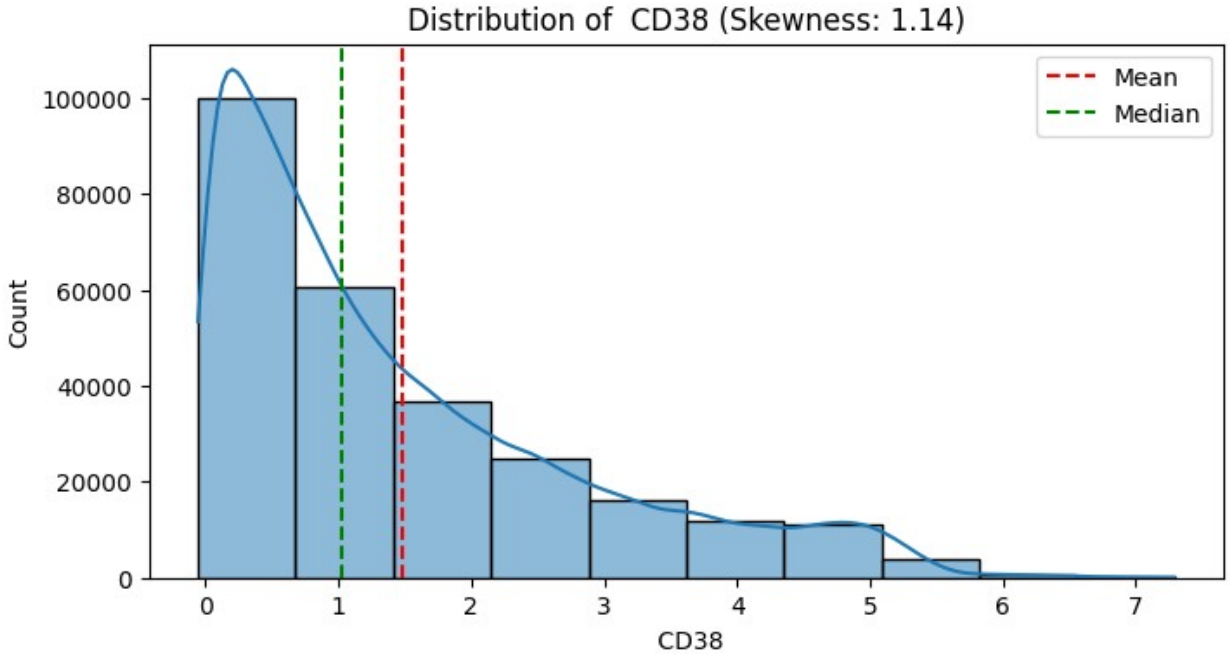

Distribution of CD11c (Skewness: 1.73)
Mean
Median
Count
CD11c
0
50000
100000
150000
200000
250000
300000
0
1
2
3
4
5
6

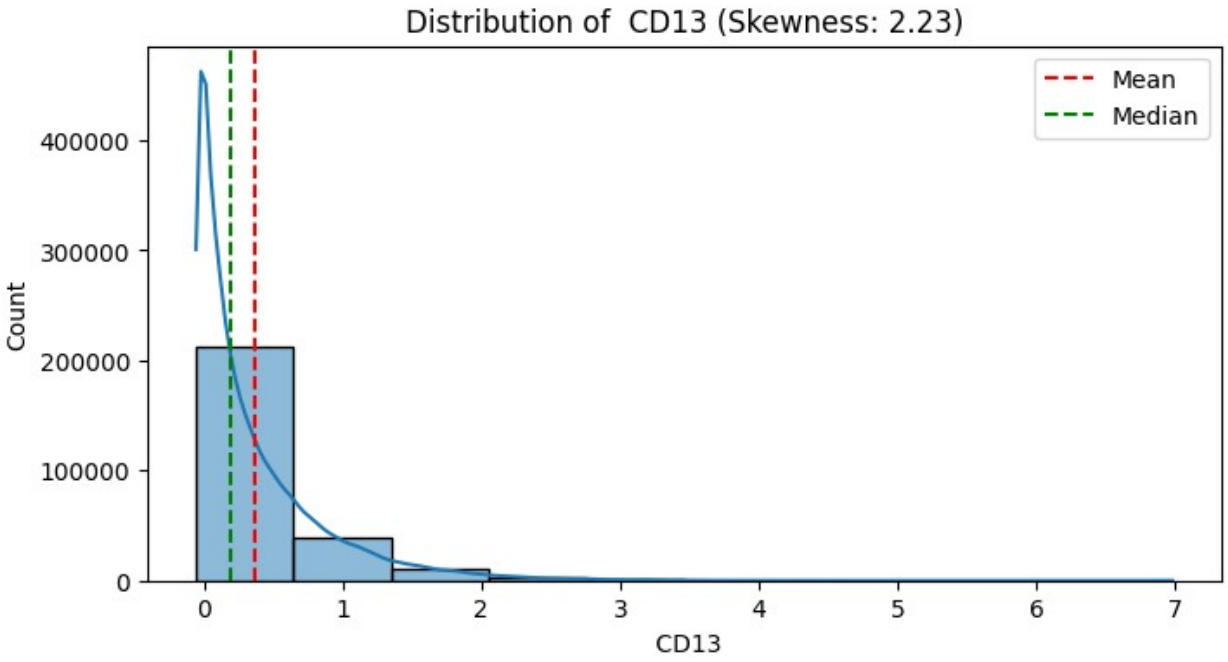

Distribution of CD7 (Skewness: 1.61)
Mean
Median
Count
CD7
0
50000
100000
150000
200000
250000
300000
0
1
2
3
4
5
6

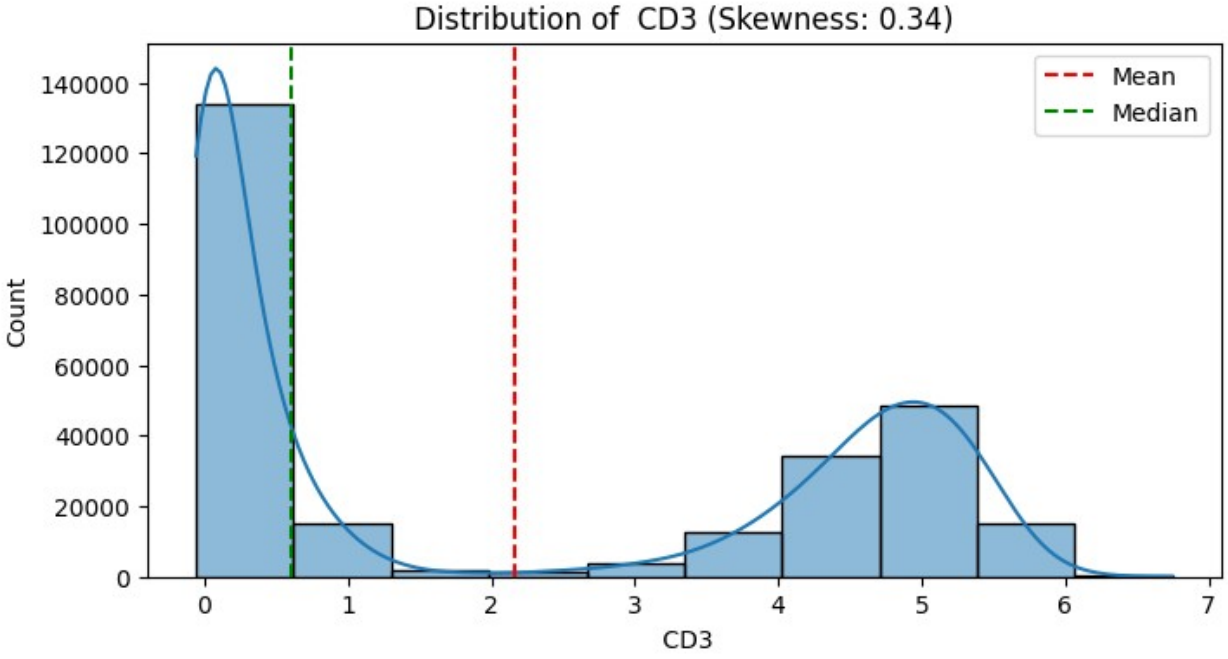

Distribution of CD15 (Skewness: 1.45)
Mean
Median
Count
CD15
0
25000
50000
75000
100000
125000
150000
175000
200000
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

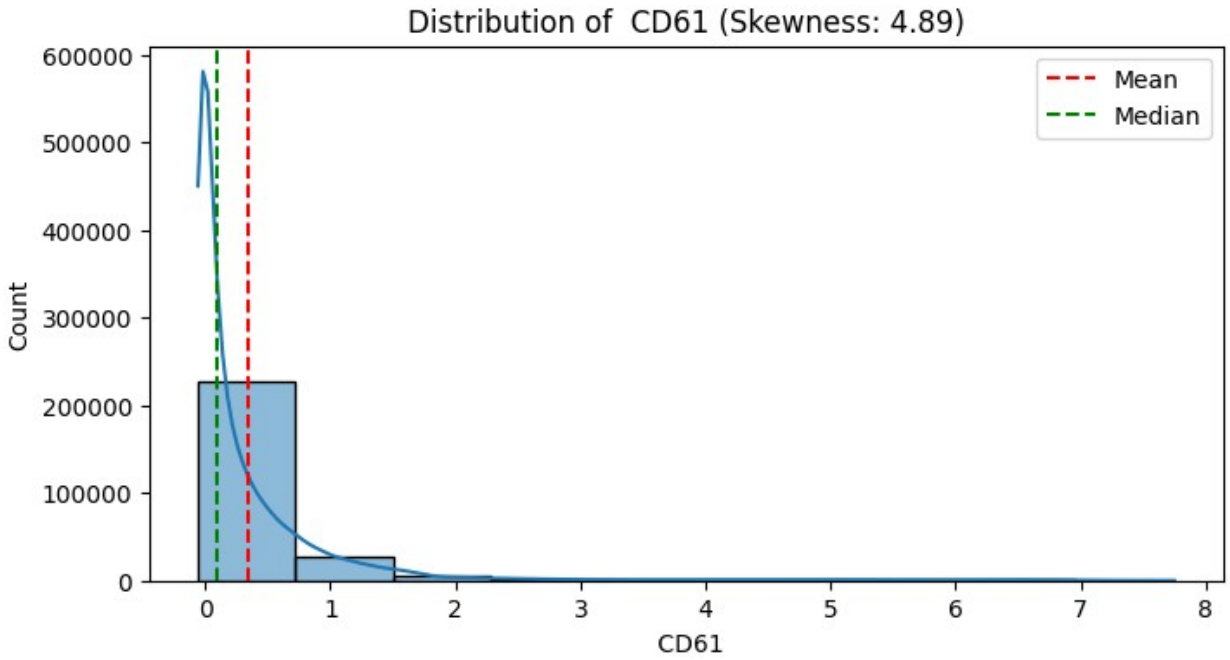

Distribution of CD16 (Skewness: 5.73)
Mean
Median
Count
CD16
0
100000
200000
300000
400000
500000
600000
700000
0
1
2
3
4
5

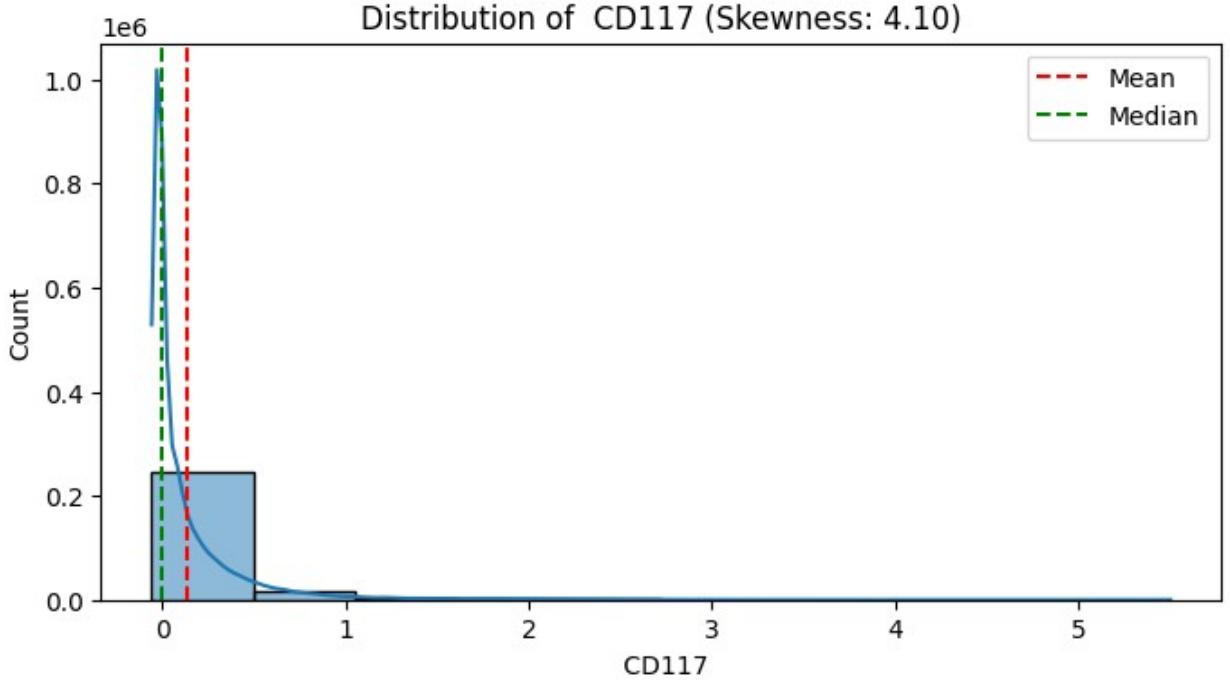

Distribution of CD44 (Skewness: -0.43)
Mean
Median
Count
CD44
0
10000
20000
30000
40000
50000
60000
70000
80000
0
1
2
3
4
5
6
7

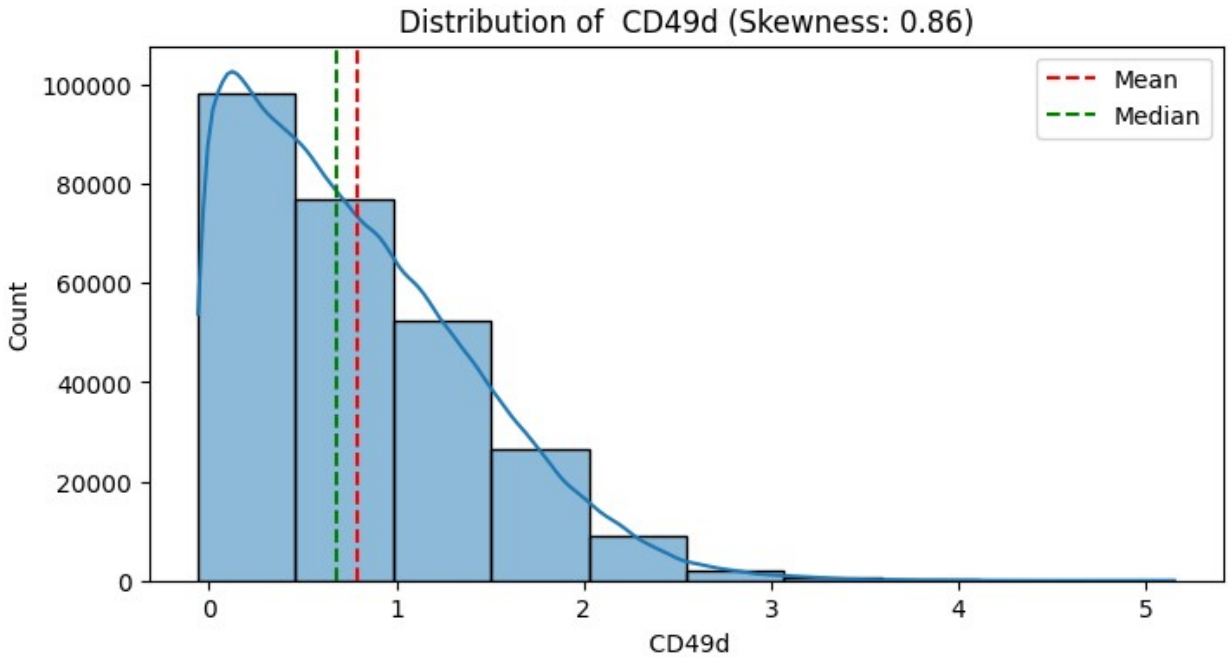

Distribution of CD38 (Skewness: 1.14)
Mean
Median
Count
CD38
0
20000
40000
60000
80000
100000
0
1
2
3
4
5
6
7

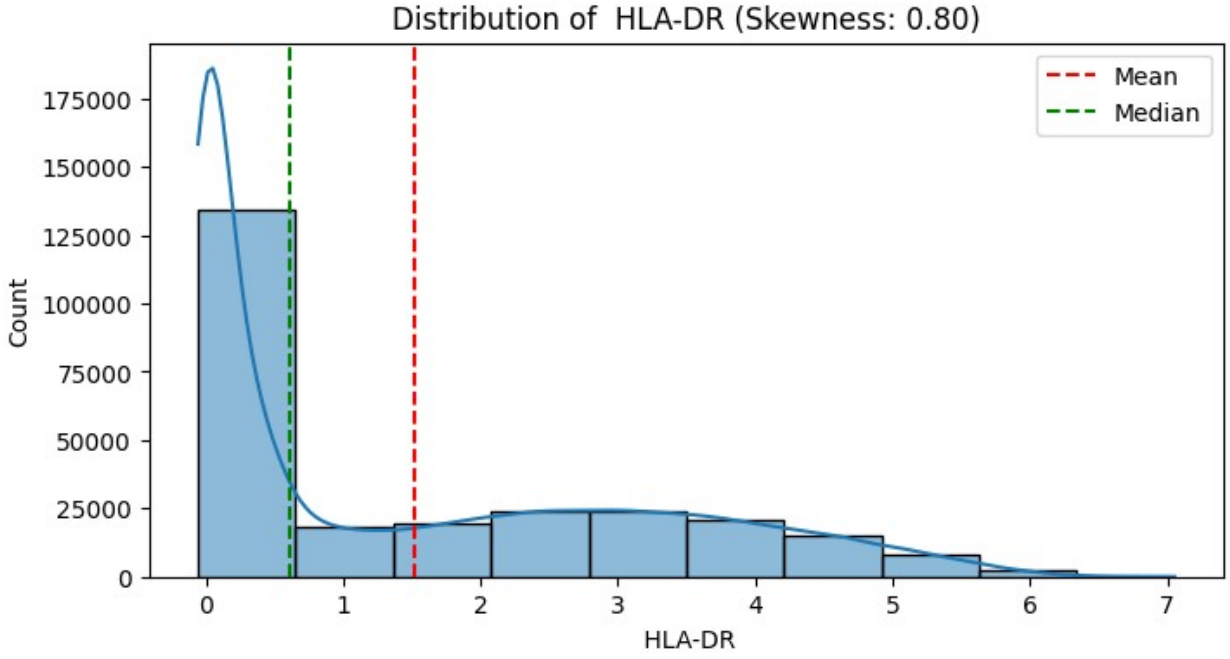

Distribution of CD13 (Skewness: 2.23)
Mean
Median
Count
CD13
0
100000
200000
300000
400000
0
1
2
3
4
5
6
7

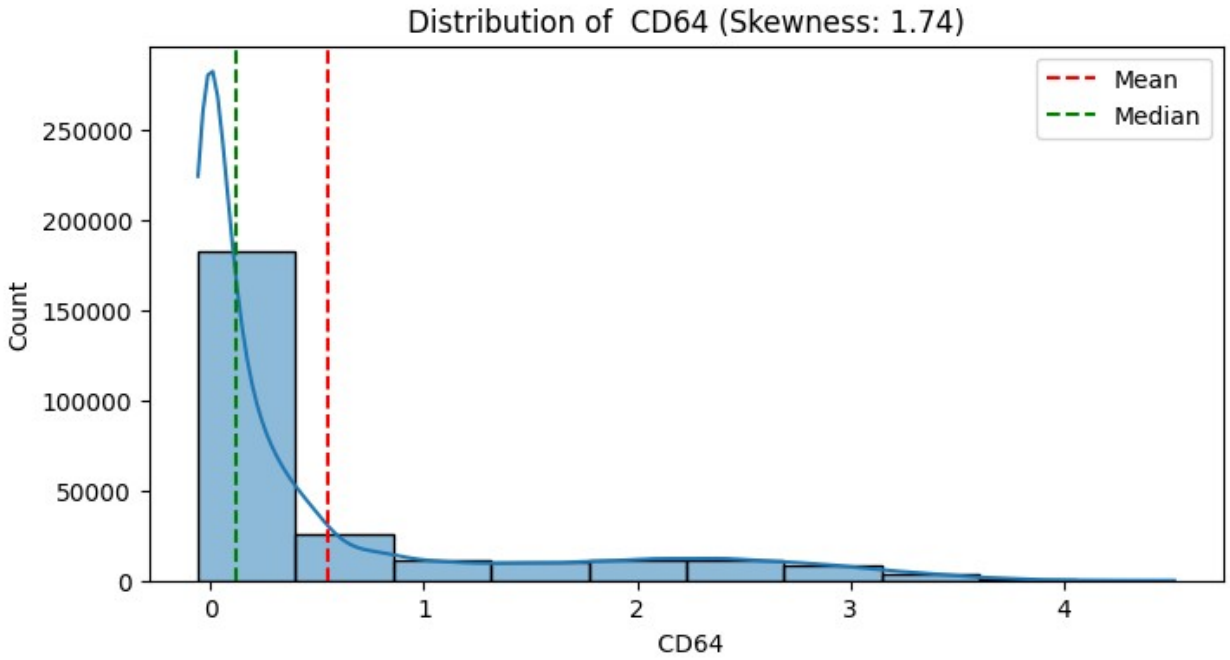

Distribution of CD3 (Skewness: 0.34)
Mean
Median
Count
CD3
0
20000
40000
60000
80000
100000
120000
140000
0
1
2
3
4
5
6
7

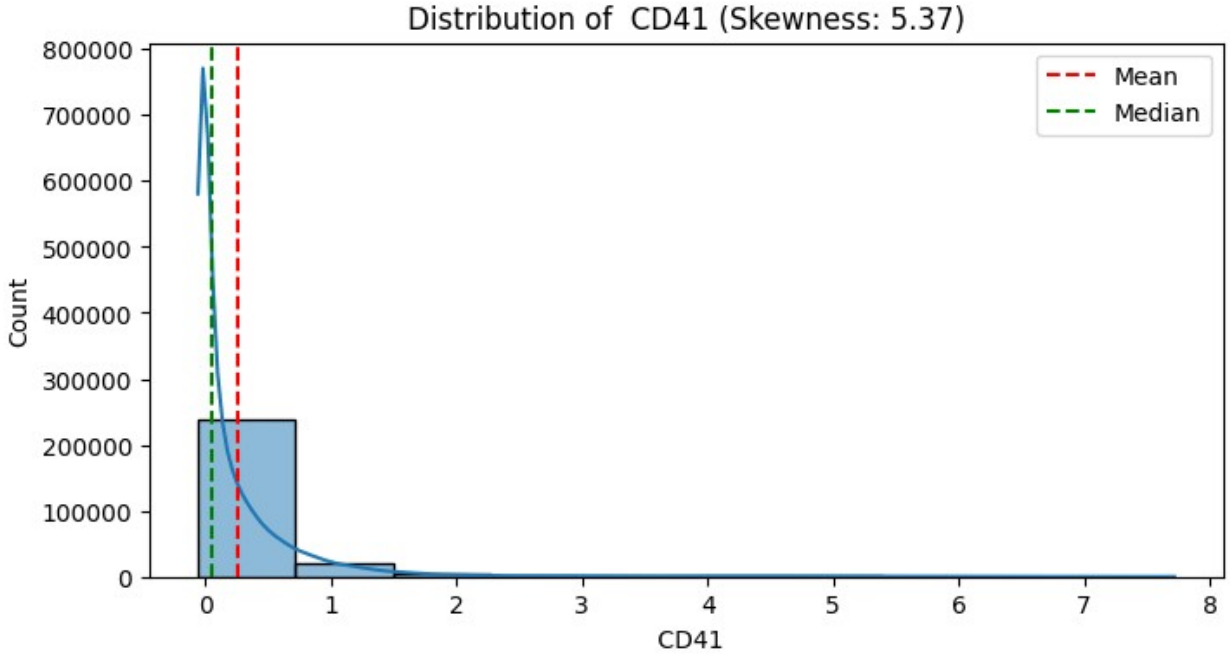

Distribution of CD61 (Skewness: 4.89)
Mean
Median
Count
CD61
600000
500000
400000
300000
200000
100000
0
0
1
2
3
4
5
6
7
8

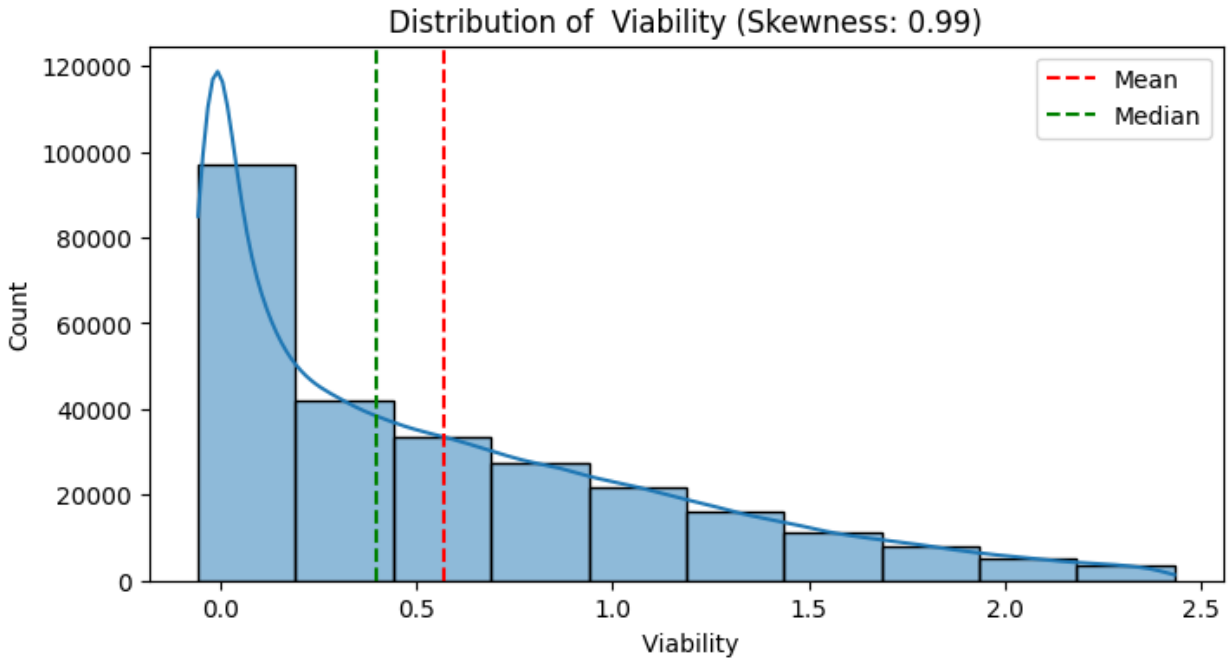

Distribution of CD117 (Skewness: 4.10)
1e6
Mean
Median
Count
CD117
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5

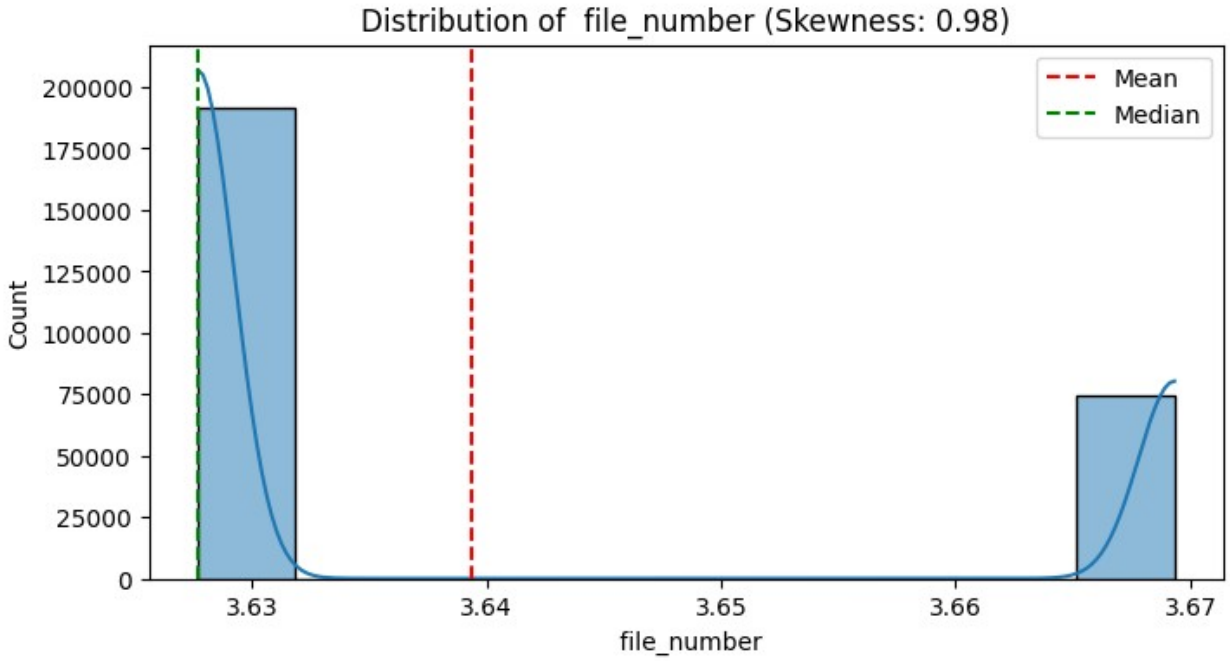

Distribution of CD49d (Skewness: 0.86)
Mean
Median
Count
CD49d
0
20000
40000
60000
80000
100000
0
1
2
3
4
5

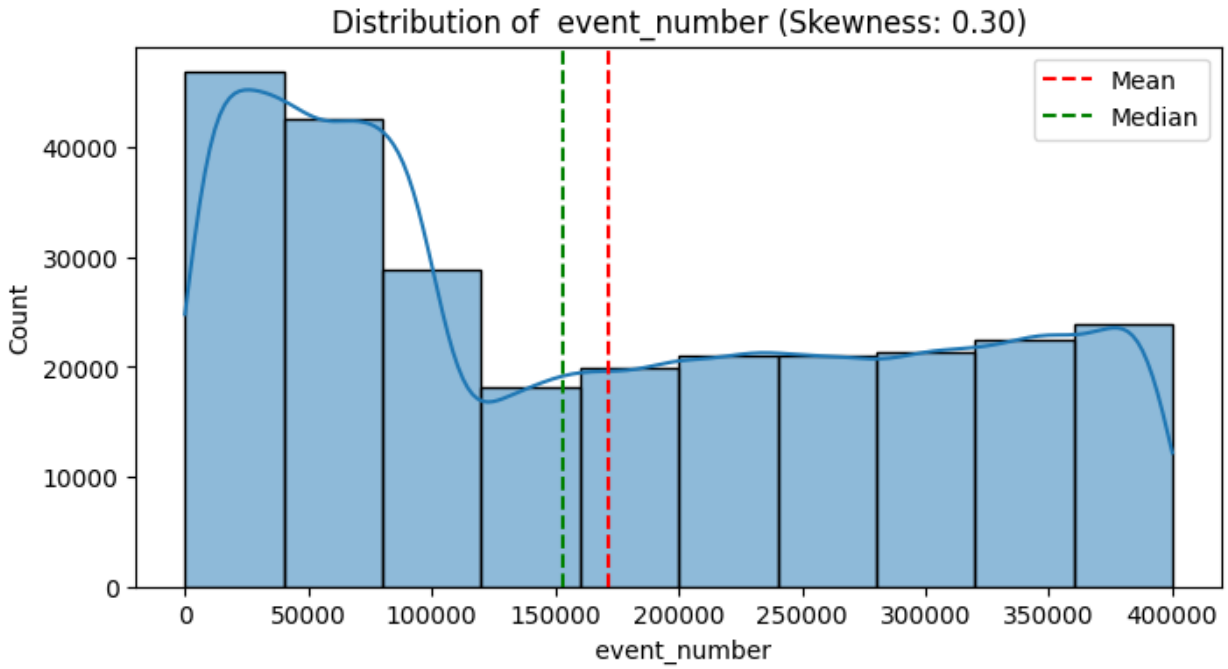

Distribution of HLA-DR (Skewness: 0.80)
Mean
Median
Count
HLA-DR
0
25000
50000
75000
100000
125000
150000
175000
0
1
2
3
4
5
6
7

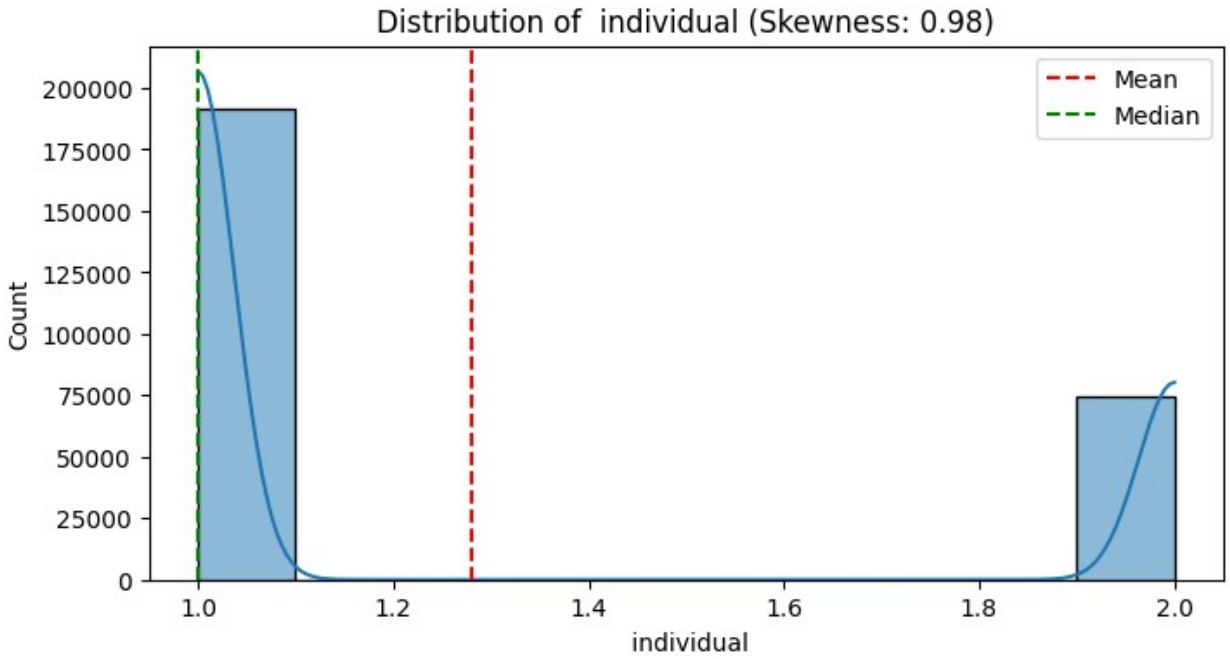

Distribution of CD64 (Skewness: 1.74)
Mean
Median
Count
CD64
0
50000
100000
150000
200000
250000
0
1
2
3
4


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


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


Distribution of file_number (Skewness: 0.98)
Mean
Median
Count
file_number
0
25000
50000
75000
100000
125000
150000
175000
200000
3.63
3.64
3.65
3.66
3.67



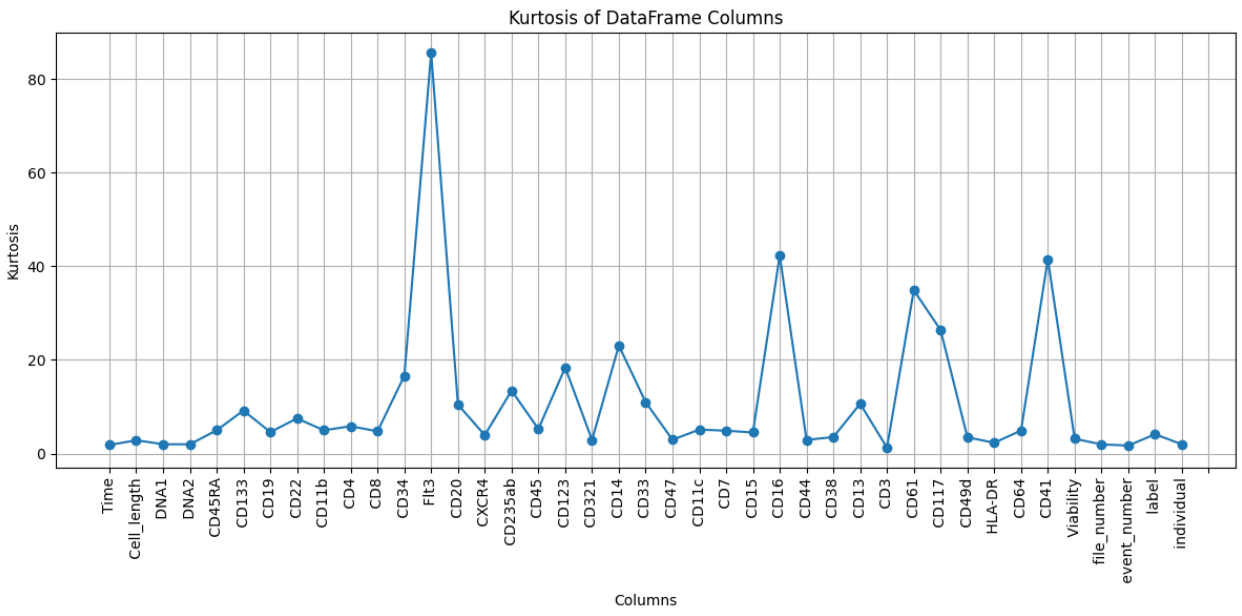


This code plots the Kurtosis at each point using a Line Graph

```
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import kurtosis

# Assuming df is your DataFrame
```

```
# 1. Data Type Conversion (Important!)
for col in df.columns:
    df[col] = pd.to_numeric(df[col], errors='coerce')

# 2. Calculate Kurtosis
kurtosis_values = df.apply(lambda x: kurtosis(x, nan_policy='omit',
fisher=False))
# fisher=False for Pearson kurtosis (normal kurtosis = 3)

# 3. Create DataFrame for Plotting
kurtosis_df = pd.DataFrame({'Column': df.columns, 'Kurtosis':
kurtosis_values})

# 4. Plot Line Graph
plt.figure(figsize=(12, 6))  # Adjust figure size as needed
plt.plot(kurtosis_df['Column'], kurtosis_df['Kurtosis'], marker='o',
linestyle='-')
plt.title('Kurtosis of DataFrame Columns')
plt.xlabel('Columns')
plt.ylabel('Kurtosis')
plt.xticks(rotation=90)  # Rotate x-axis labels for better readability
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/scipy/stats/_stats_py.py:1287:
RuntimeWarning: Mean of empty slice.
  mean = a.mean(axis, keepdims=True)
```



Prints the kurtosis values for each column

```python
import pandas as pd

# Assuming your data is in a CSV file named 'your_data.csv'
df = pd.read_csv('/content/drive/MyDrive/data/Levine_32dim.csv')

# Replace or remove spaces (or empty strings) in the dataframe with
NaN
df = df.replace(' ', pd.NA)

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

kurtosis = df.kurtosis()

# Print the kurtosis values for each column
print(kurtosis)
```

```
<ipython-input-6-065aee63191c>:4: DtypeWarning: Columns (39) have
mixed types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv('/content/drive/MyDrive/data/Levine_32dim.csv')
```

```
Time            -1.159846
 Cell_length    -0.165947
 DNA1           -1.005934
 DNA2           -1.024951
 CD45RA          1.964313
 CD133           6.190169
 CD19            1.590919
 CD22            4.500297
 CD11b           1.964622
 CD4             2.844363
 CD8             1.745808
 CD34           13.596631
 Flt3           82.584914
 CD20            7.435572
 CXCR4           0.936337
 CD235ab        10.440753
 CD45            2.246835
 CD123          15.361473
 CD321          -0.085361
 CD14           20.063229
 CD33            7.967686
 CD47           -0.056128
 CD11c           2.117172
 CD7             1.885155
 CD15            1.504423
 CD16           39.288365
 CD44           -0.081178
 CD38            0.521215
 CD13            7.635673
```

```
CD3              -1.735402
CD61             31.878541
CD117            23.375467
CD49d             0.468152
HLA-DR           -0.690042
CD64              1.910668
CD41             38.521744
Viability         0.156947
file_number      -1.035585
event_number     -1.293824
label             1.125359
individual       -1.035585
                       NaN
dtype: float64
```

This prints the kurtosis valus for each column using Histograms and also mentions which are the haevy tails and which are the light tails

```
import pandas as pd
from scipy.stats import kurtosis
import matplotlib.pyplot as plt

# Assuming df is your DataFrame

# Convert all columns to numeric, coerce errors to NaN
# IMPORTANT: This line ensures all columns are numeric before
calculating kurtosis
for col in df.columns:
    df[col] = pd.to_numeric(df[col], errors='coerce')

# Calculate kurtosis for each column
kurtosis_values = df.apply(kurtosis, fisher=False, nan_policy='omit')
# Fisher=False gives Pearson kurtosis (normal kurtosis = 3)
# nan_policy='omit' is added here to ignore NaN values in kurtosis
calculation


# Create a DataFrame with kurtosis values
kurtosis_df = pd.DataFrame({'Column': df.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)# 4.1 Histogram for each column
for column in df.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)
from scipy.stats import kurtosis
# Calculate kurtosis for each column
kurtosis_values = df.apply(kurtosis, fisher=False)  # Fisher=False
gives Pearson kurtosis (normal kurtosis = 3)

# Create a DataFrame with kurtosis values
kurtosis_df = pd.DataFrame({'Column': df.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)# 4.1 Histogram for each column
for column in df.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(True)
    plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/scipy/stats/_stats_py.py:1287:
RuntimeWarning: Mean of empty slice.
  mean = a.mean(axis, keepdims=True)
/usr/local/lib/python3.10/dist-packages/numpy/core/_methods.py:121:
RuntimeWarning: invalid value encountered in divide
  ret = um.true_divide(
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:3504
: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/usr/local/lib/python3.10/dist-packages/numpy/core/_methods.py:129:
RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

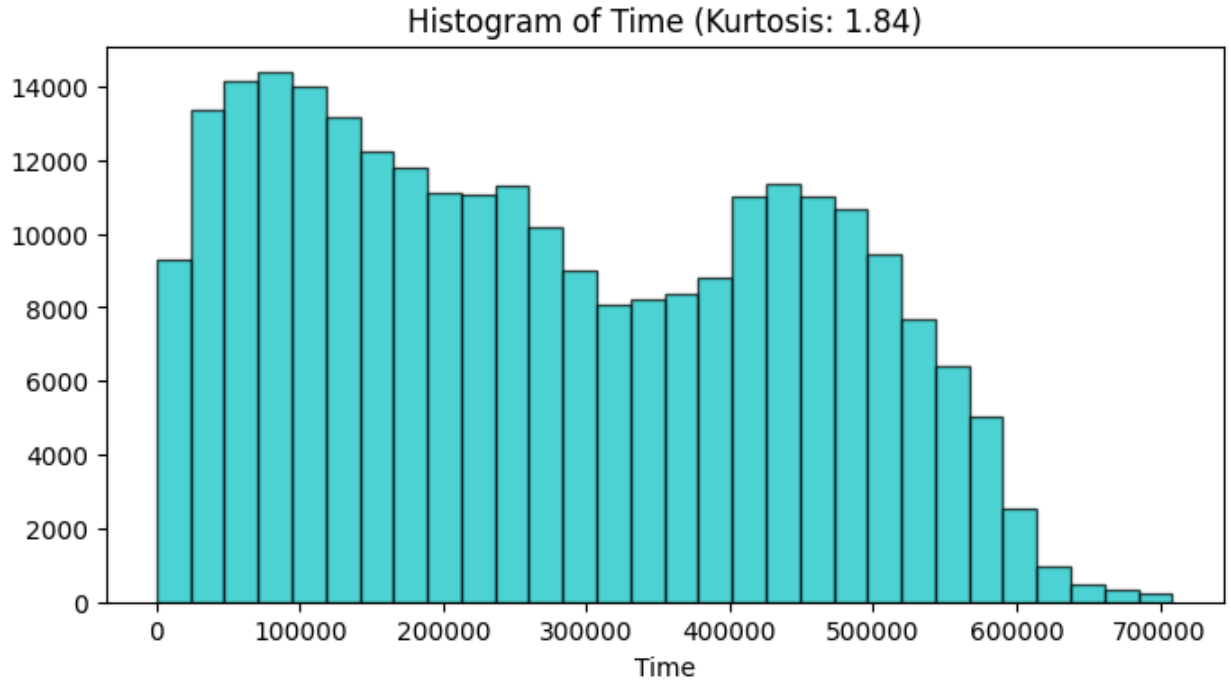
	Column	Kurtosis	Category
Time	Time	1.840154	Platykurtic (light tails)
Cell_length	Cell_length	2.834033	Platykurtic (light tails)
DNA1	DNA1	1.994062	Platykurtic (light tails)
DNA2	DNA2	1.975046	Platykurtic (light tails)
CD45RA	CD45RA	4.964254	Leptokurtic (heavy tails)
CD133	CD133	9.190030	Leptokurtic (heavy tails)
CD19	CD19	4.590867	Leptokurtic (heavy tails)
CD22	CD22	7.500190	Leptokurtic (heavy tails)
CD11b	CD11b	4.964563	Leptokurtic (heavy tails)
CD4	CD4	5.844286	Leptokurtic (heavy tails)
CD8	CD8	4.745753	Leptokurtic (heavy tails)
CD34	CD34	16.596353	Leptokurtic (heavy tails)
Flt3	Flt3	85.583337	Leptokurtic (heavy tails)
CD20	CD20	10.435410	Leptokurtic (heavy tails)
CXCR4	CXCR4	3.936297	Leptokurtic (heavy tails)
CD235ab	CD235ab	13.440534	Leptokurtic (heavy tails)
CD45	CD45	5.246770	Leptokurtic (heavy tails)
CD123	CD123	18.361161	Leptokurtic (heavy tails)
CD321	CD321	2.914618	Platykurtic (light tails)
CD14	CD14	23.062829	Leptokurtic (heavy tails)
CD33	CD33	10.967514	Leptokurtic (heavy tails)
CD47	CD47	2.943851	Platykurtic (light tails)
CD11c	CD11c	5.117109	Leptokurtic (heavy tails)
CD7	CD7	4.885097	Leptokurtic (heavy tails)
CD15	CD15	4.504372	Leptokurtic (heavy tails)
CD16	CD16	42.287603	Leptokurtic (heavy tails)
CD44	CD44	2.918801	Platykurtic (light tails)
CD38	CD38	3.521183	Leptokurtic (heavy tails)
CD13	CD13	10.635506	Leptokurtic (heavy tails)
CD3	CD3	1.264608	Platykurtic (light tails)
CD61	CD61	34.877918	Leptokurtic (heavy tails)
CD117	CD117	26.375004	Leptokurtic (heavy tails)
CD49d	CD49d	3.468121	Leptokurtic (heavy tails)
HLA-DR	HLA-DR	2.309949	Platykurtic (light tails)
CD64	CD64	4.910610	Leptokurtic (heavy tails)
CD41	CD41	41.520997	Leptokurtic (heavy tails)

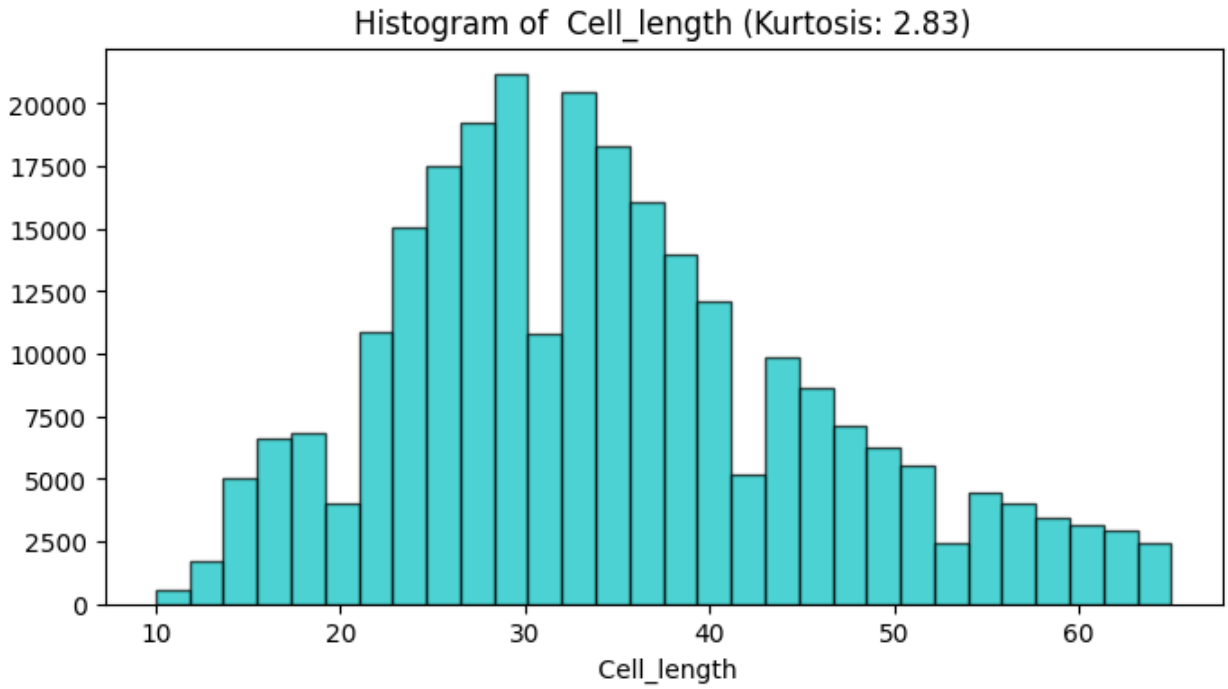
```
 Viability         Viability   3.156921  Leptokurtic (heavy tails)
 file_number     file_number   1.964411  Platykurtic (light tails)
 event_number   event_number   1.706178  Platykurtic (light tails)
 label                 label   4.125248  Leptokurtic (heavy tails)
 individual       individual   1.964411  Platykurtic (light tails)
                                    NaN  Mesokurtic (normal tails)
```

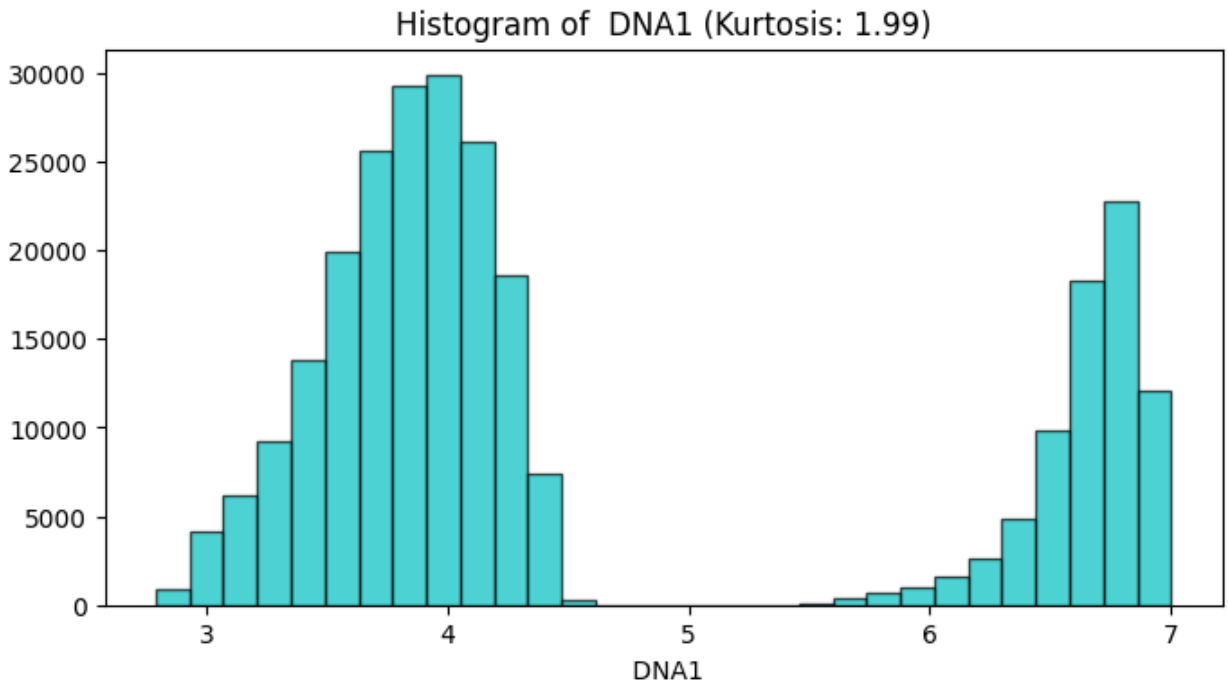
```
<ipython-input-9-4d592d6fdc55>:34: RuntimeWarning: More than 20
figures have been opened. Figures created through the pyplot interface
(`matplotlib.pyplot.figure`) are retained until explicitly closed and
may consume too much memory. (To control this warning, see the rcParam
`figure.max_open_warning`). Consider using
`matplotlib.pyplot.close()`.
  plt.figure(figsize=(8, 4))
```

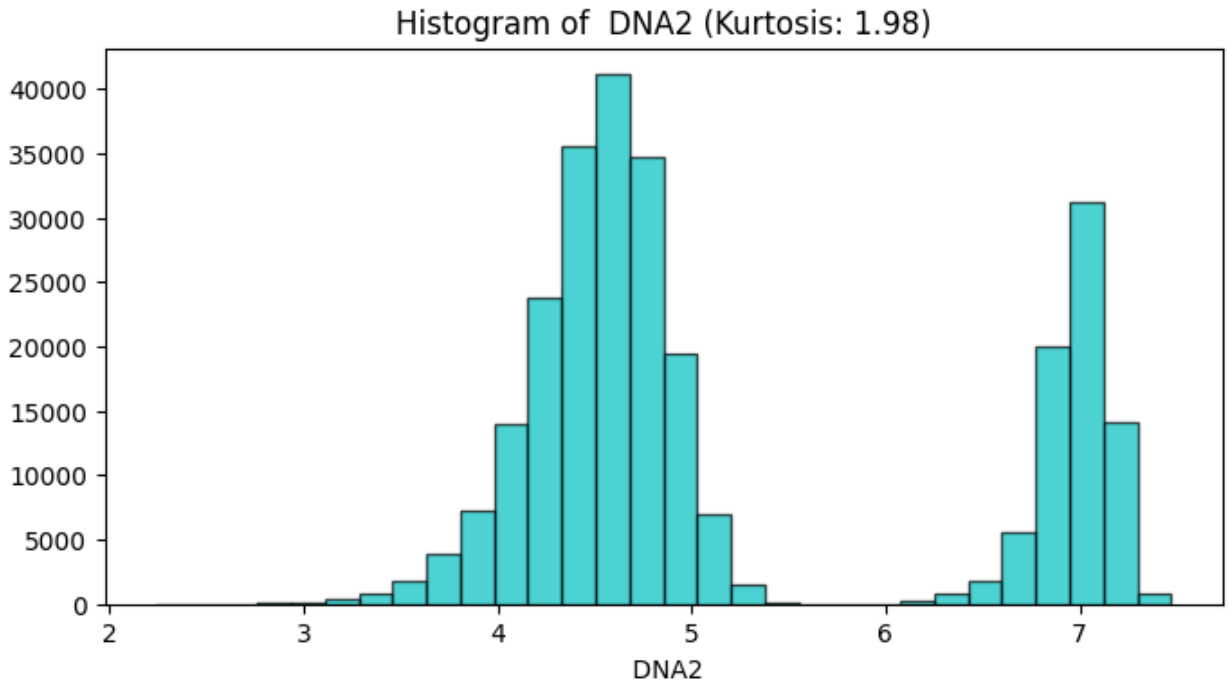
```
                      Column   Kurtosis                     Category
Time                    Time   1.840154  Platykurtic (light tails)
 Cell_length     Cell_length   2.834033  Platykurtic (light tails)
 DNA1                   DNA1   1.994062  Platykurtic (light tails)
 DNA2                   DNA2   1.975046  Platykurtic (light tails)
 CD45RA               CD45RA   4.964254  Leptokurtic (heavy tails)
 CD133                CD133   9.190030  Leptokurtic (heavy tails)
 CD19                  CD19   4.590867  Leptokurtic (heavy tails)
 CD22                  CD22   7.500190  Leptokurtic (heavy tails)
 CD11b                CD11b   4.964563  Leptokurtic (heavy tails)
 CD4                    CD4   5.844286  Leptokurtic (heavy tails)
 CD8                    CD8   4.745753  Leptokurtic (heavy tails)
 CD34                  CD34  16.596353  Leptokurtic (heavy tails)
 Flt3                  Flt3  85.583337  Leptokurtic (heavy tails)
 CD20                  CD20  10.435410  Leptokurtic (heavy tails)
 CXCR4                CXCR4   3.936297  Leptokurtic (heavy tails)
 CD235ab            CD235ab  13.440534  Leptokurtic (heavy tails)
 CD45                  CD45   5.246770  Leptokurtic (heavy tails)
 CD123                CD123  18.361161  Leptokurtic (heavy tails)
 CD321                CD321   2.914618  Platykurtic (light tails)
 CD14                  CD14  23.062829  Leptokurtic (heavy tails)
 CD33                  CD33  10.967514  Leptokurtic (heavy tails)
 CD47                  CD47   2.943851  Platykurtic (light tails)
 CD11c                CD11c   5.117109  Leptokurtic (heavy tails)
 CD7                    CD7   4.885097  Leptokurtic (heavy tails)
 CD15                  CD15   4.504372  Leptokurtic (heavy tails)
 CD16                  CD16  42.287603  Leptokurtic (heavy tails)
 CD44                  CD44   2.918801  Platykurtic (light tails)
 CD38                  CD38   3.521183  Leptokurtic (heavy tails)
 CD13                  CD13  10.635506  Leptokurtic (heavy tails)
 CD3                    CD3   1.264608  Platykurtic (light tails)
 CD61                  CD61  34.877918  Leptokurtic (heavy tails)
 CD117                CD117  26.375004  Leptokurtic (heavy tails)
 CD49d                CD49d   3.468121  Leptokurtic (heavy tails)
 HLA-DR              HLA-DR   2.309949  Platykurtic (light tails)
```

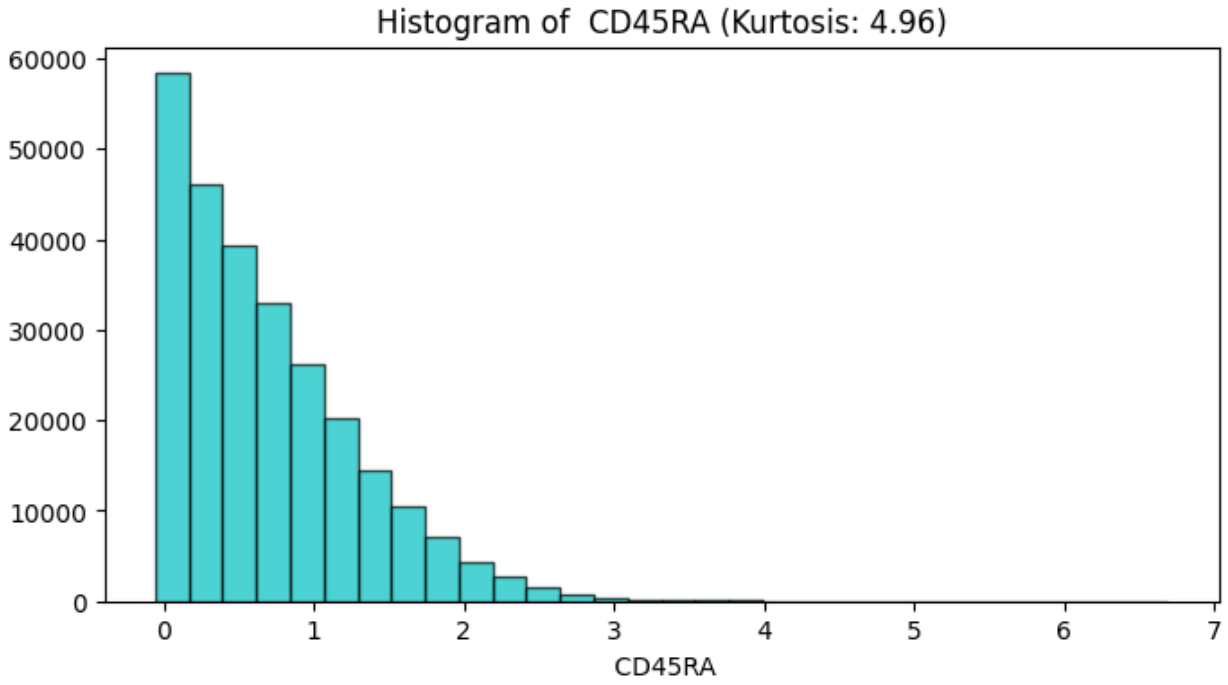
```
CD64                   CD64   4.910610  Leptokurtic (heavy tails)
CD41                   CD41  41.520997  Leptokurtic (heavy tails)
Viability           Viability   3.156921  Leptokurtic (heavy tails)
file_number      file_number   1.964411  Platykurtic (light tails)
event_number    event_number   1.706178  Platykurtic (light tails)
label                  label        NaN  Mesokurtic (normal tails)
individual        individual   1.964411  Platykurtic (light tails)
                                    NaN  Mesokurtic (normal tails)
```

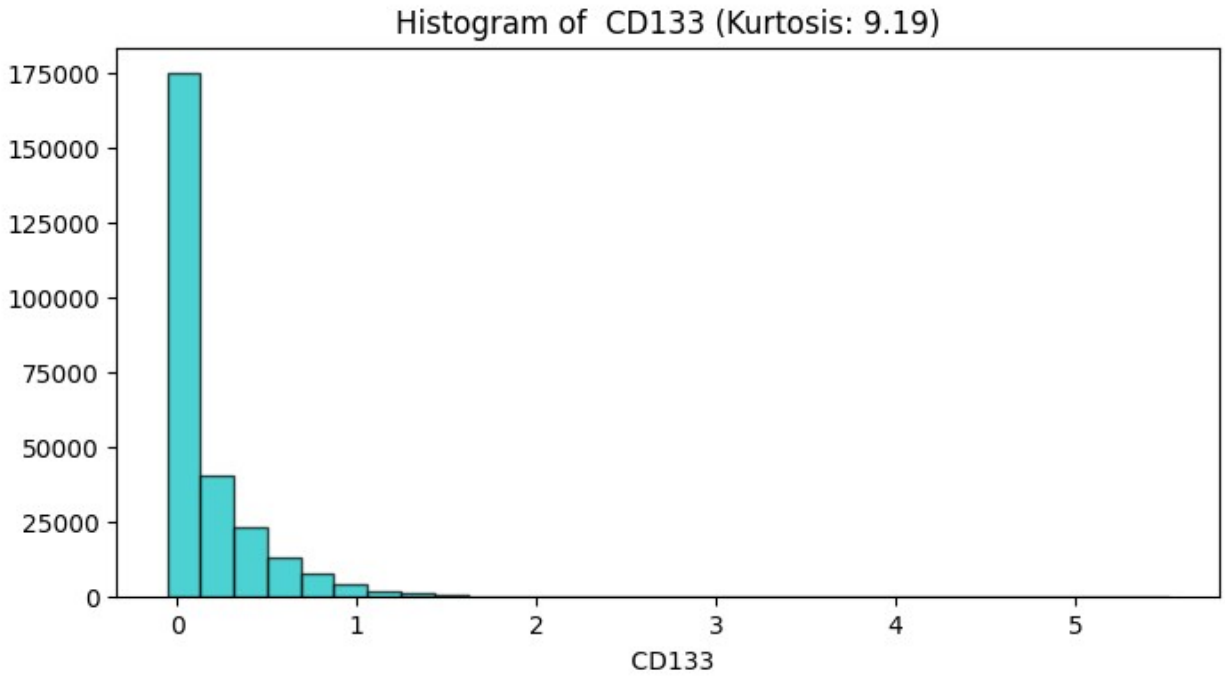


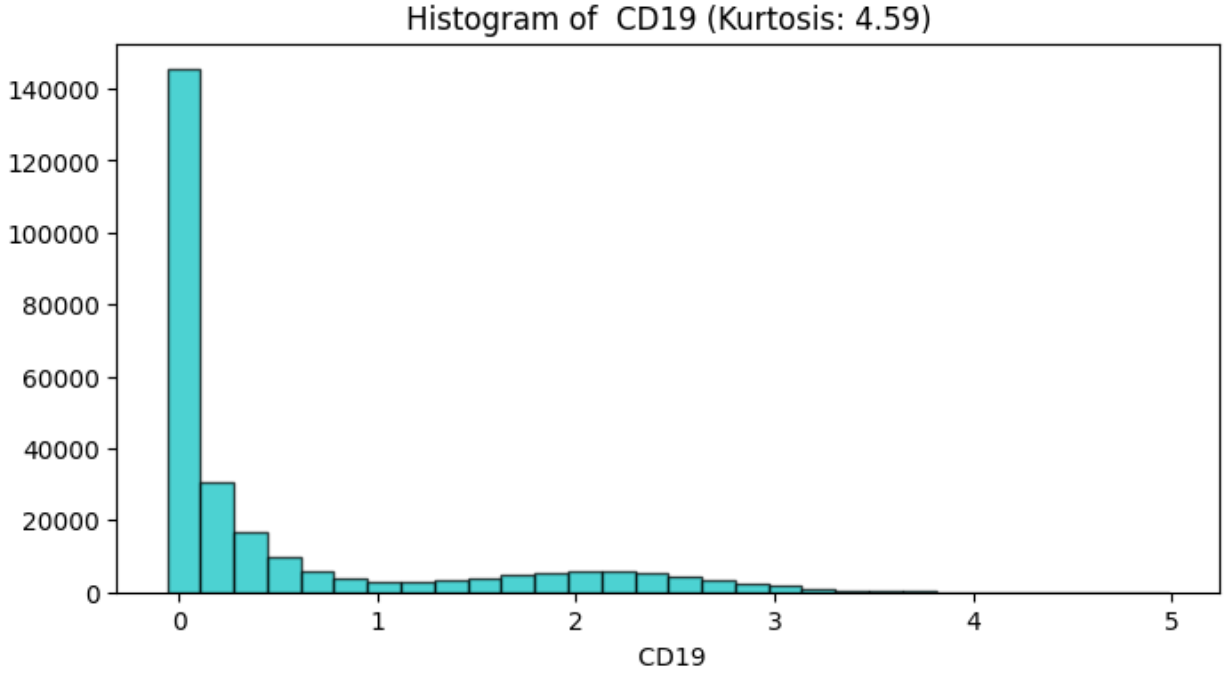

Histogram of Cell_length (Kurtosis: 2.83)
20000
17500
15000
12500
10000
7500
5000
2500
0
10
20
30
40
50
60
Cell_length

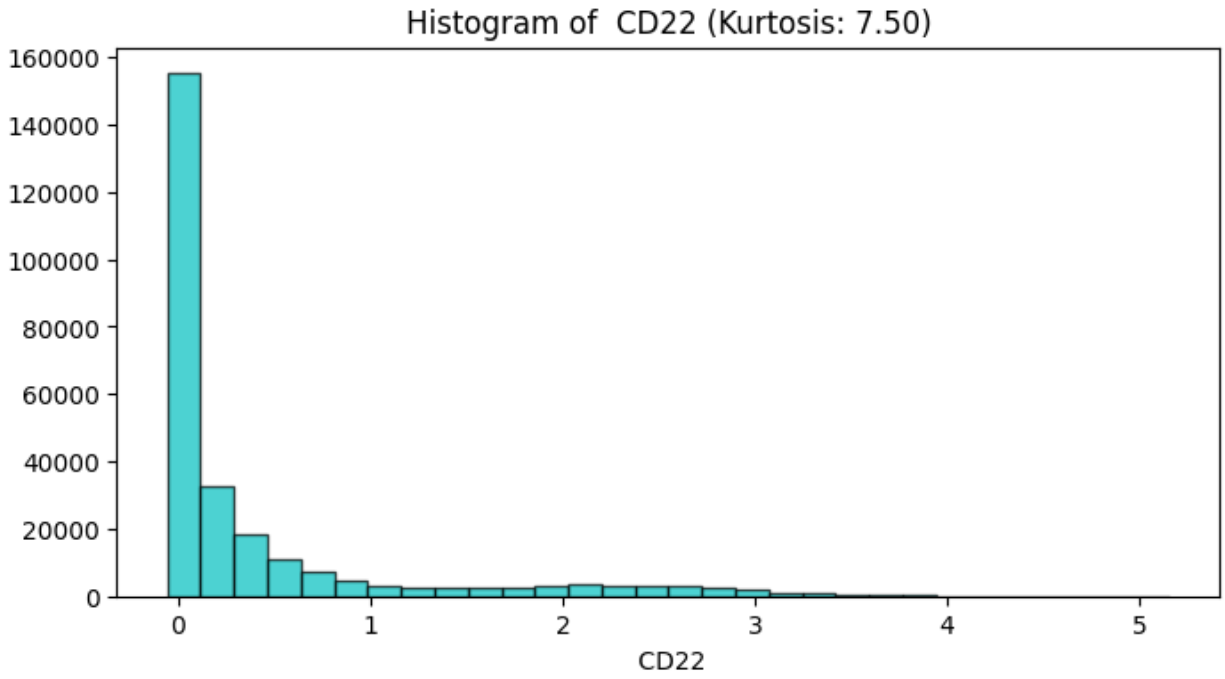

Histogram of DNA1 (Kurtosis: 1.99)
30000
25000
20000
15000
10000
5000
0
3
4
5
6
7
DNA1

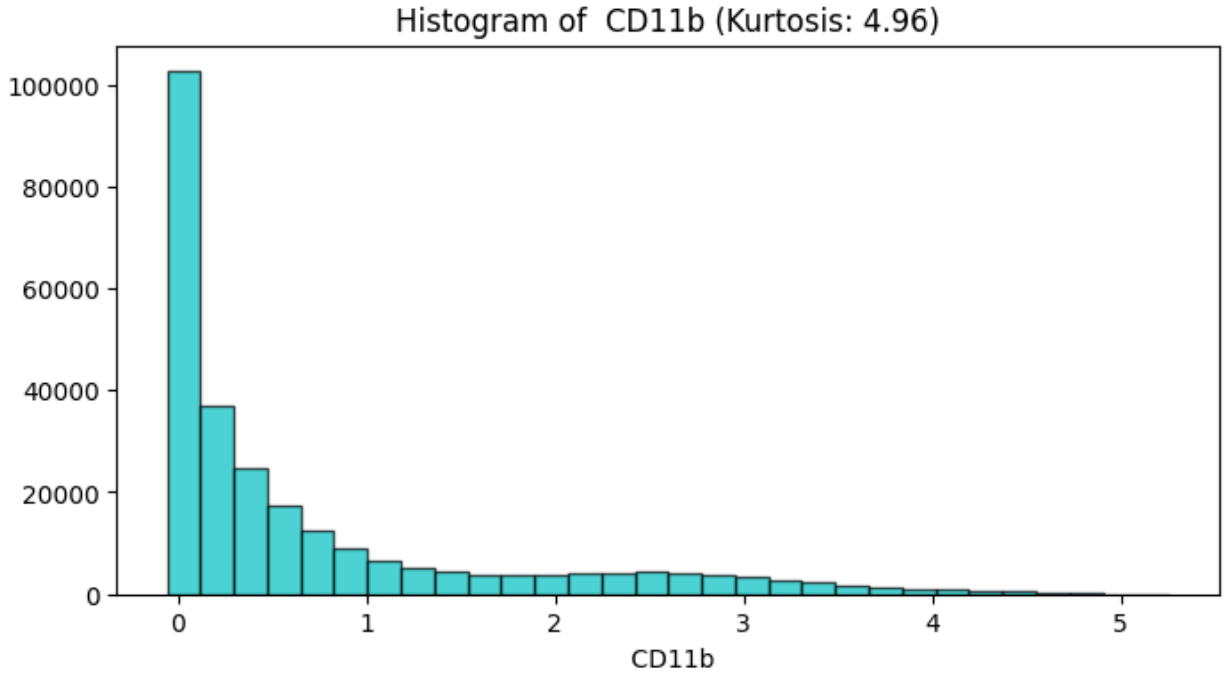

Histogram of DNA2 (Kurtosis: 1.98)
40000
35000
30000
25000
20000
15000
10000
5000
0
2
3
4
5
6
7
DNA2

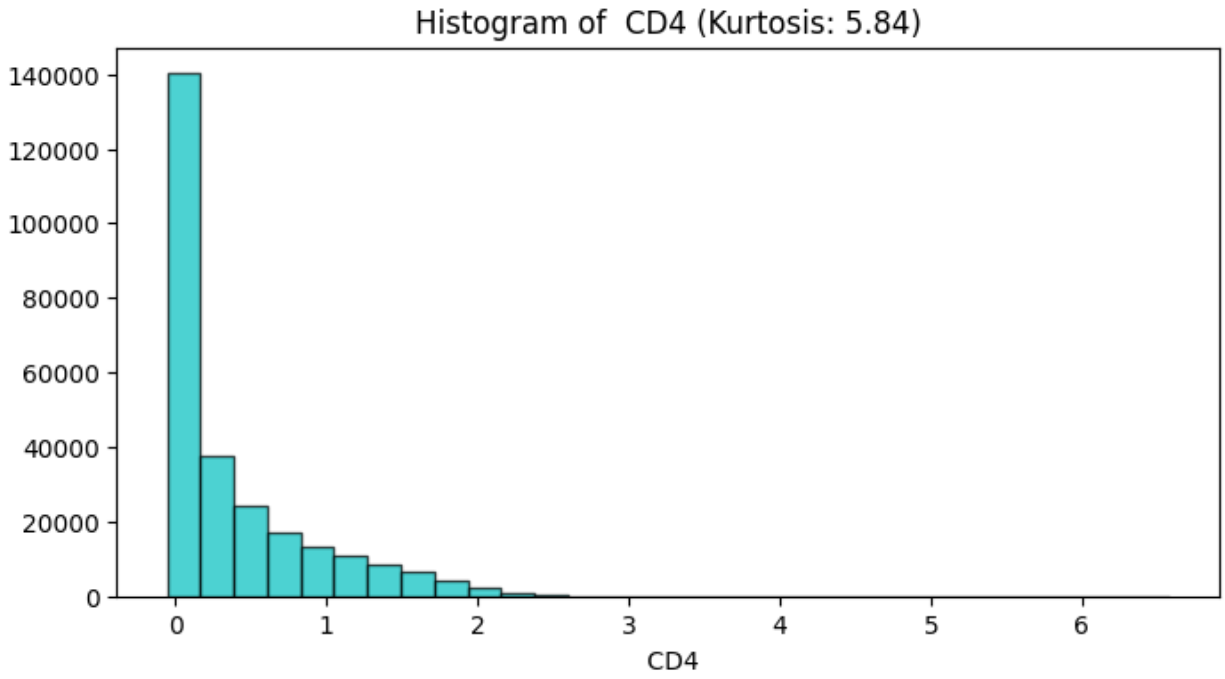

Histogram of CD45RA (Kurtosis: 4.96)
60000
50000
40000
30000
20000
10000
0
0
1
2
3
4
5
6
7
CD45RA

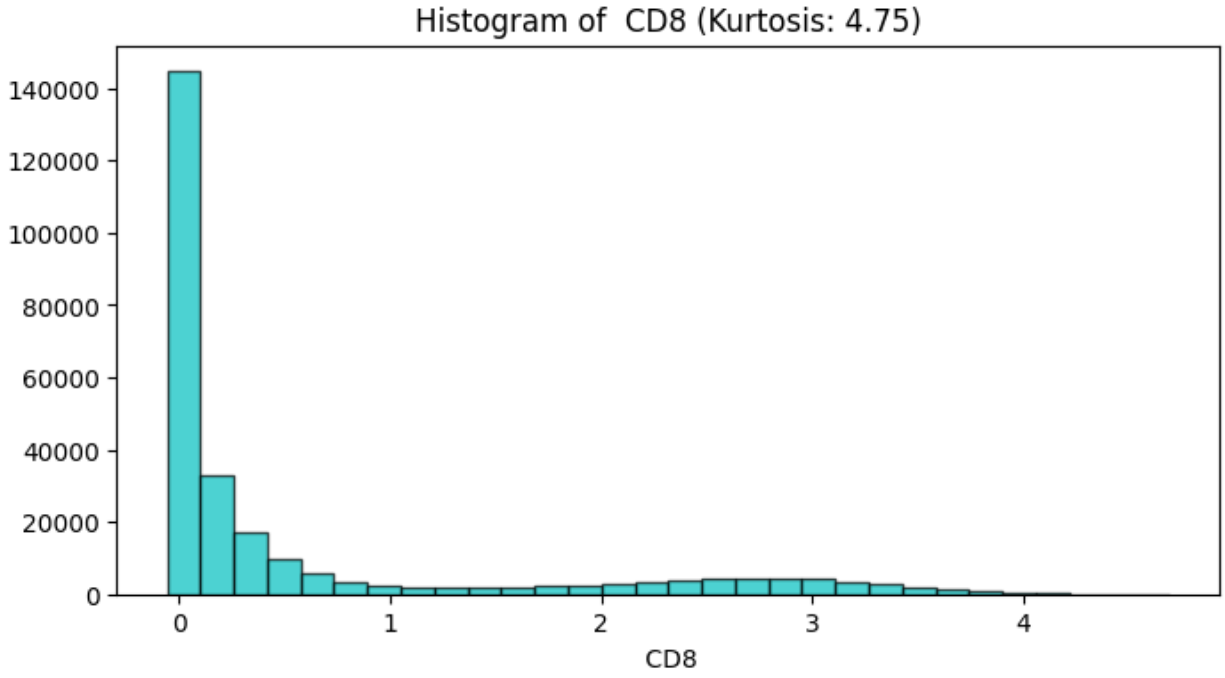

Histogram of CD133 (Kurtosis: 9.19)
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5
CD133

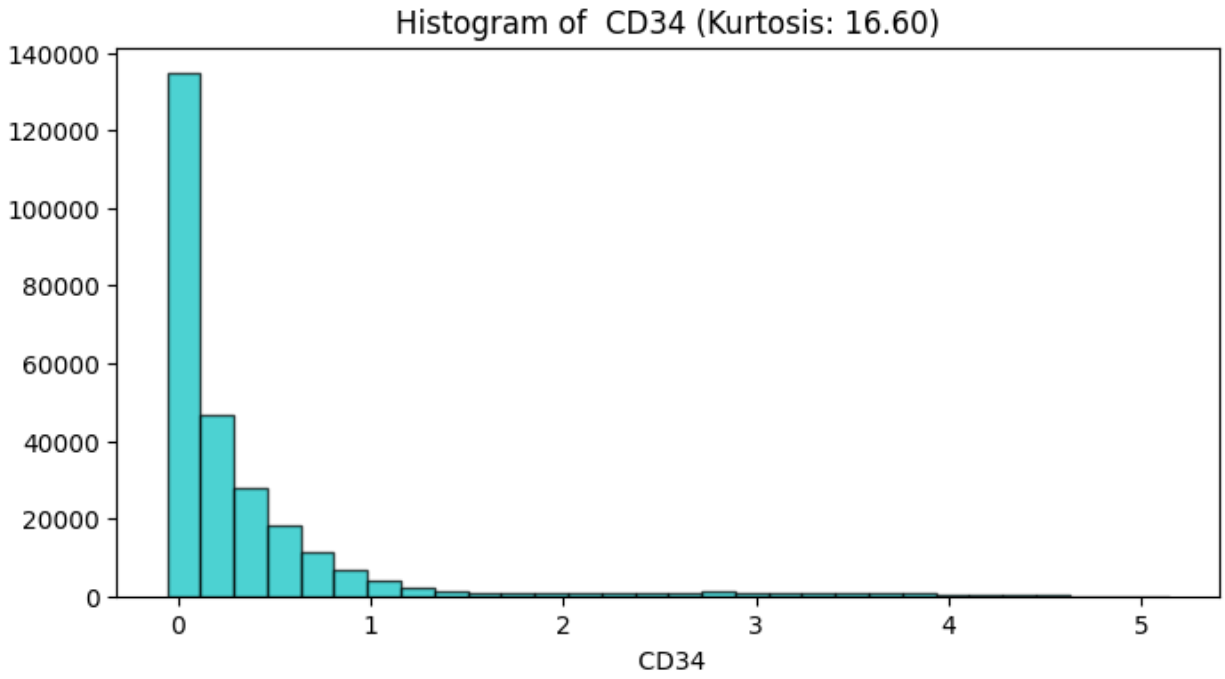

Histogram of CD19 (Kurtosis: 4.59)
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
CD19

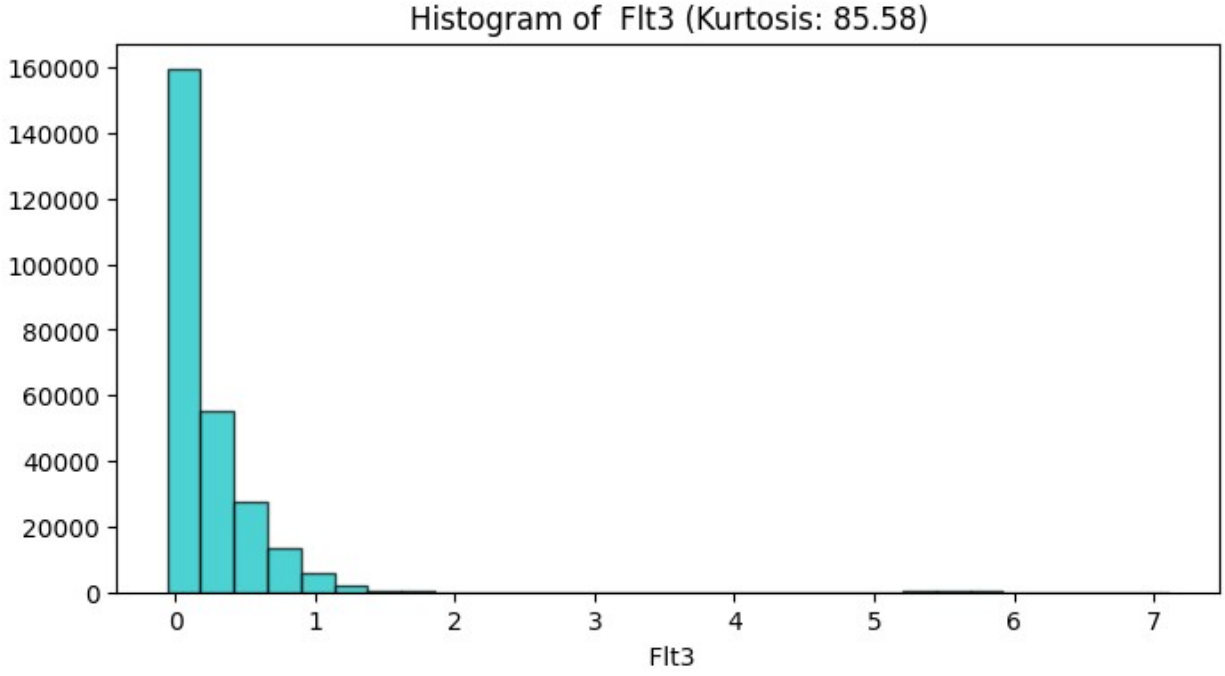

Histogram of CD22 (Kurtosis: 7.50)
160000
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
CD22

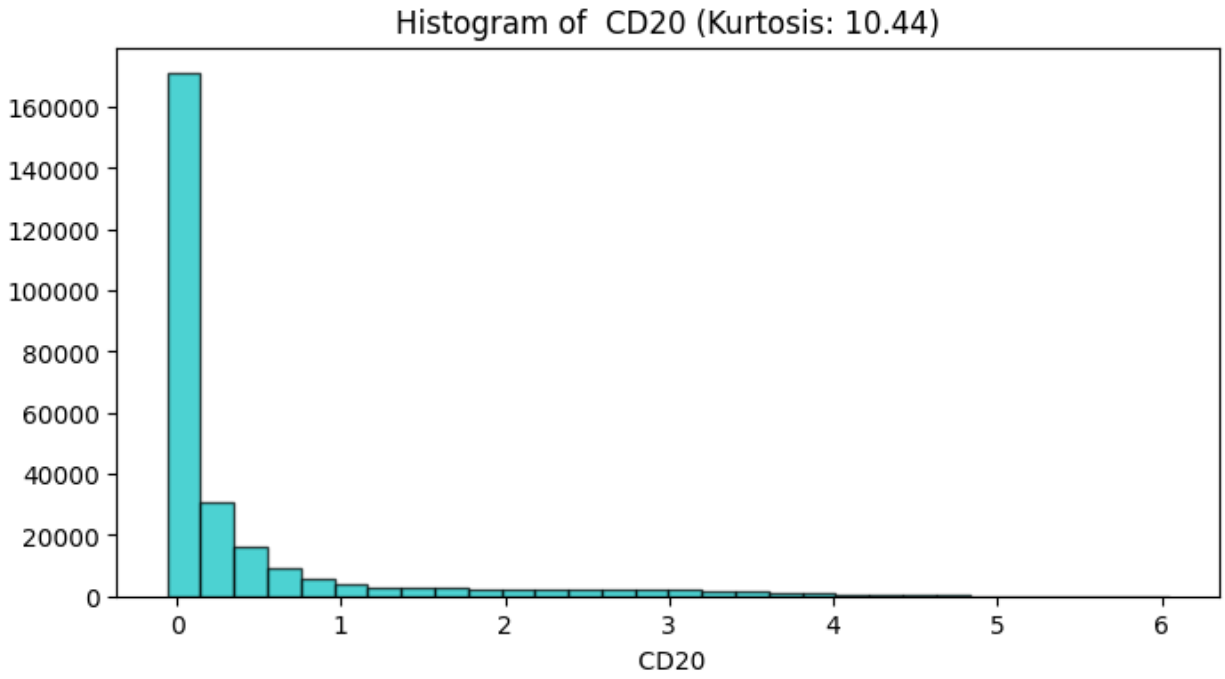

Histogram of CD11b (Kurtosis: 4.96)
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
CD11b

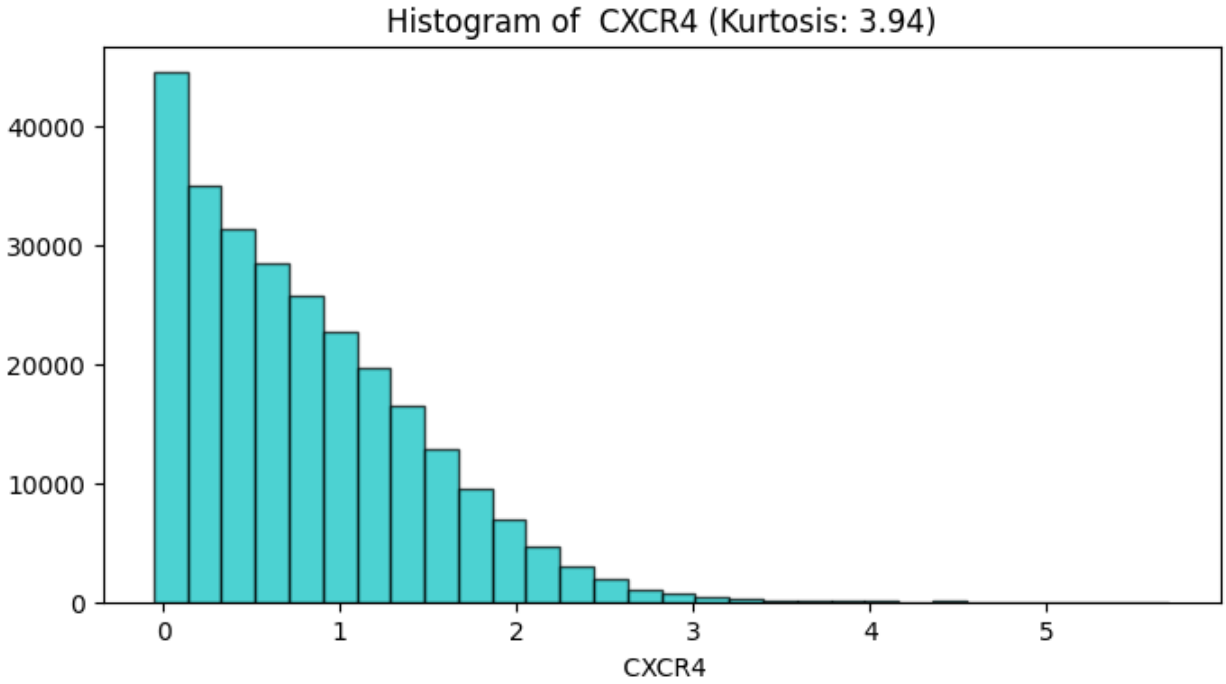

Histogram of CD4 (Kurtosis: 5.84)
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
CD4

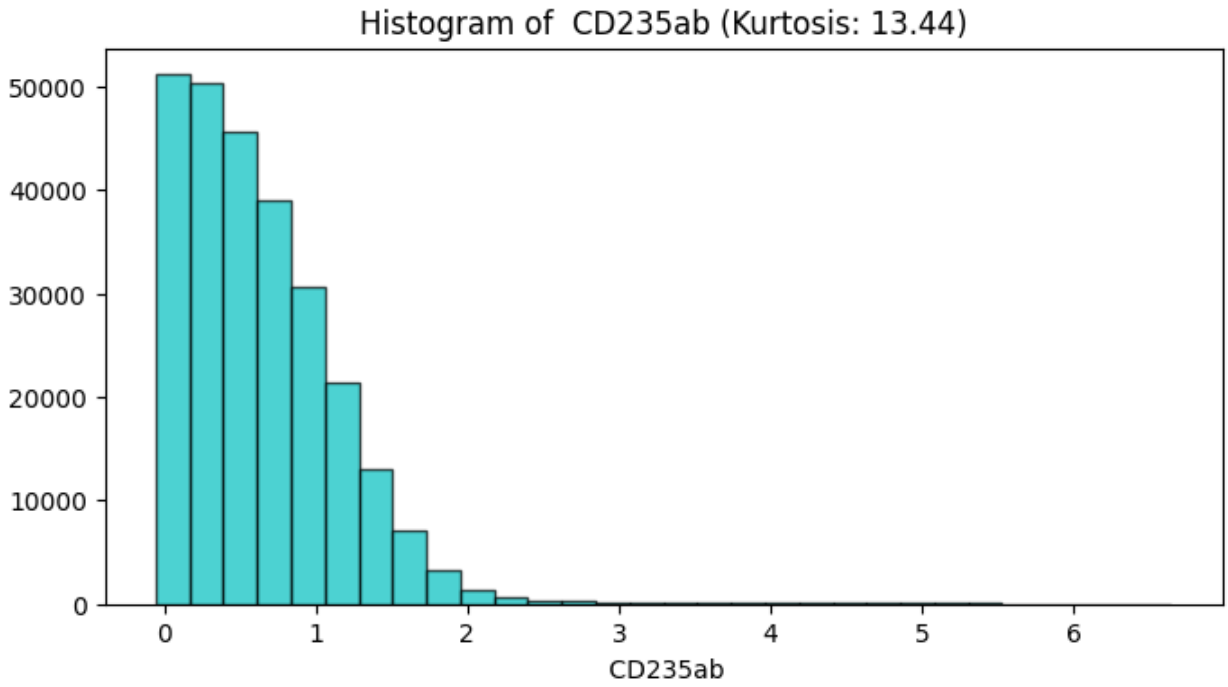

Histogram of CD8 (Kurtosis: 4.75)
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
CD8

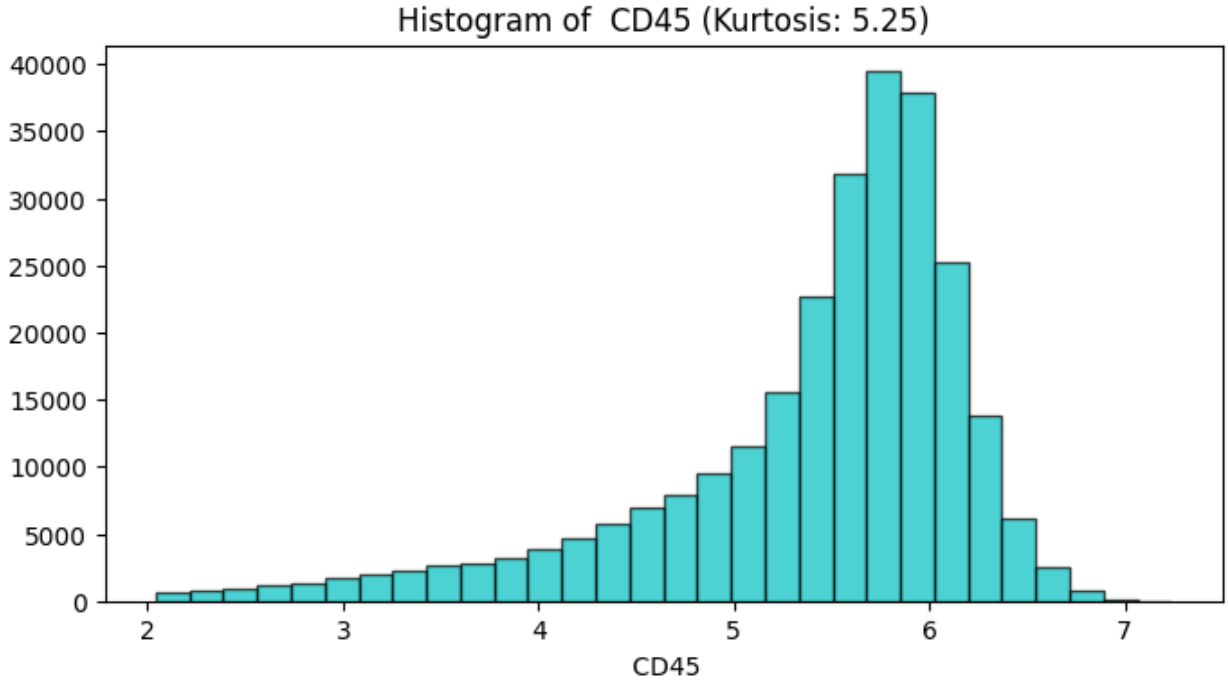

Histogram of CD34 (Kurtosis: 16.60)
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
CD34

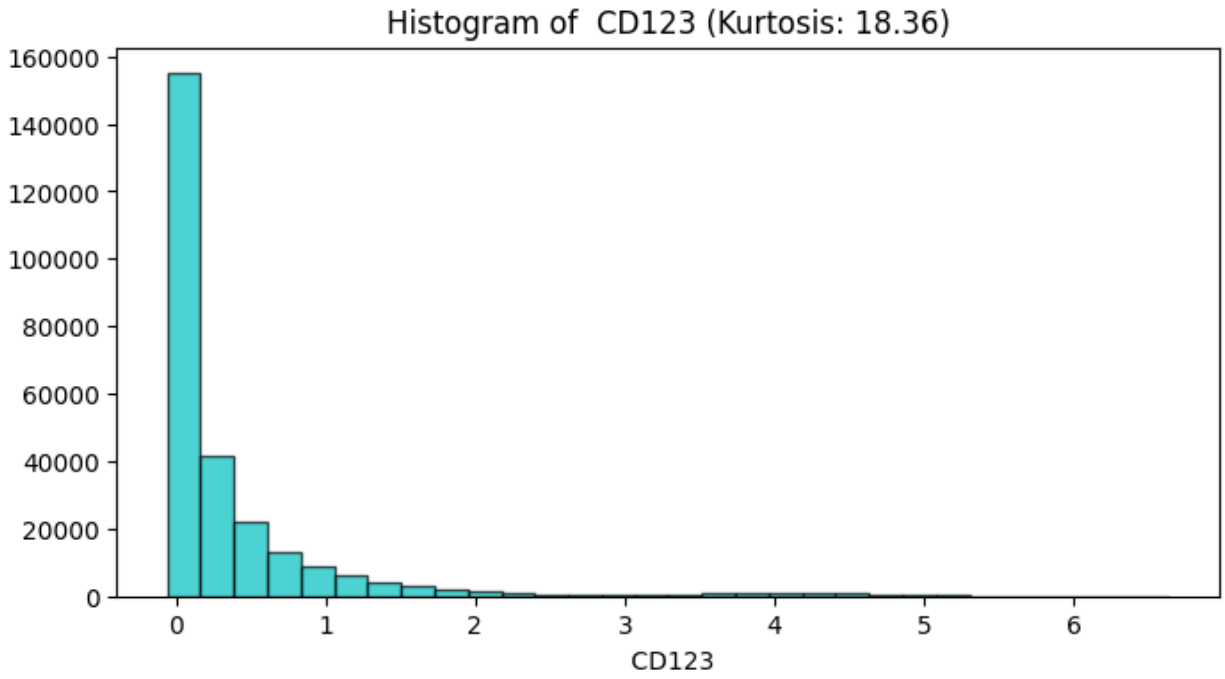

Histogram of Flt3 (Kurtosis: 85.58)
160000
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7
Flt3

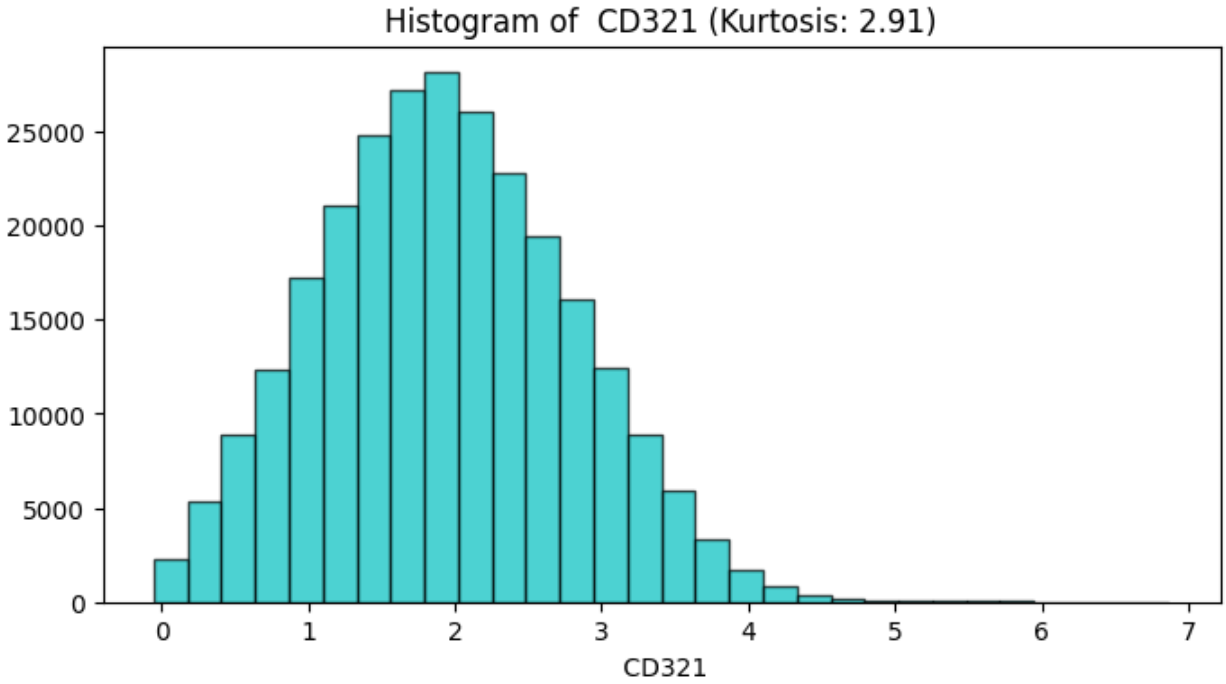

Histogram of CD20 (Kurtosis: 10.44)
160000
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
CD20

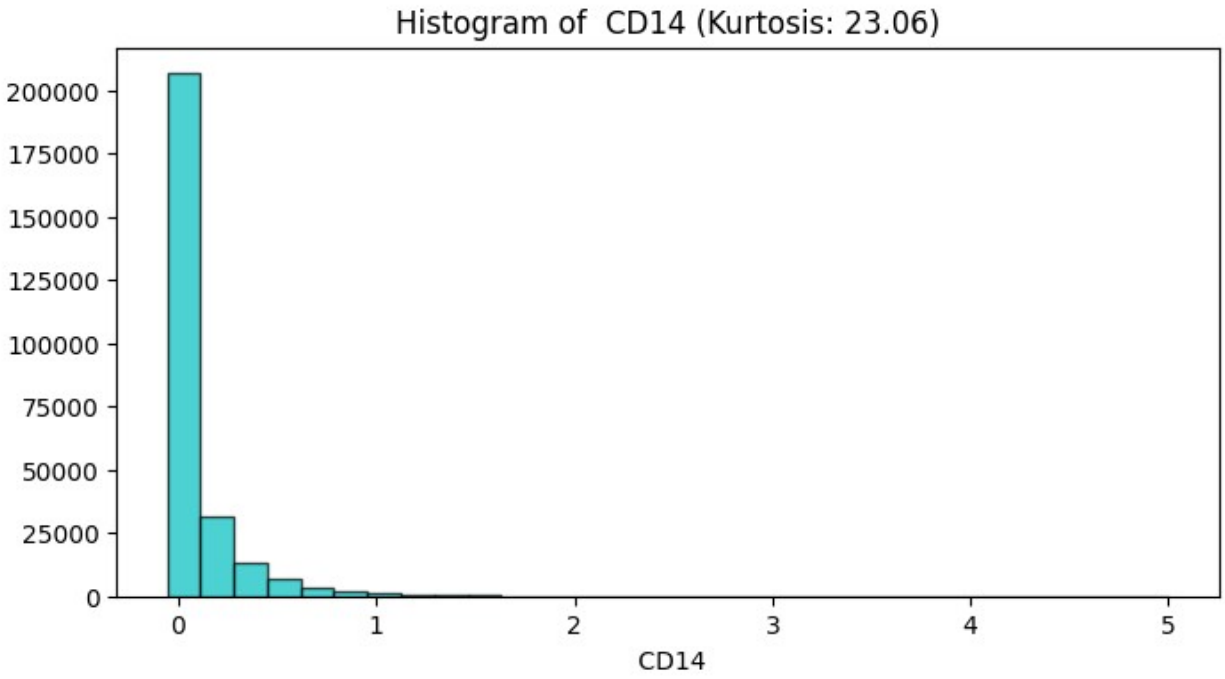

Histogram of CXCR4 (Kurtosis: 3.94)
40000
30000
20000
10000
0
0
1
2
3
4
5
CXCR4

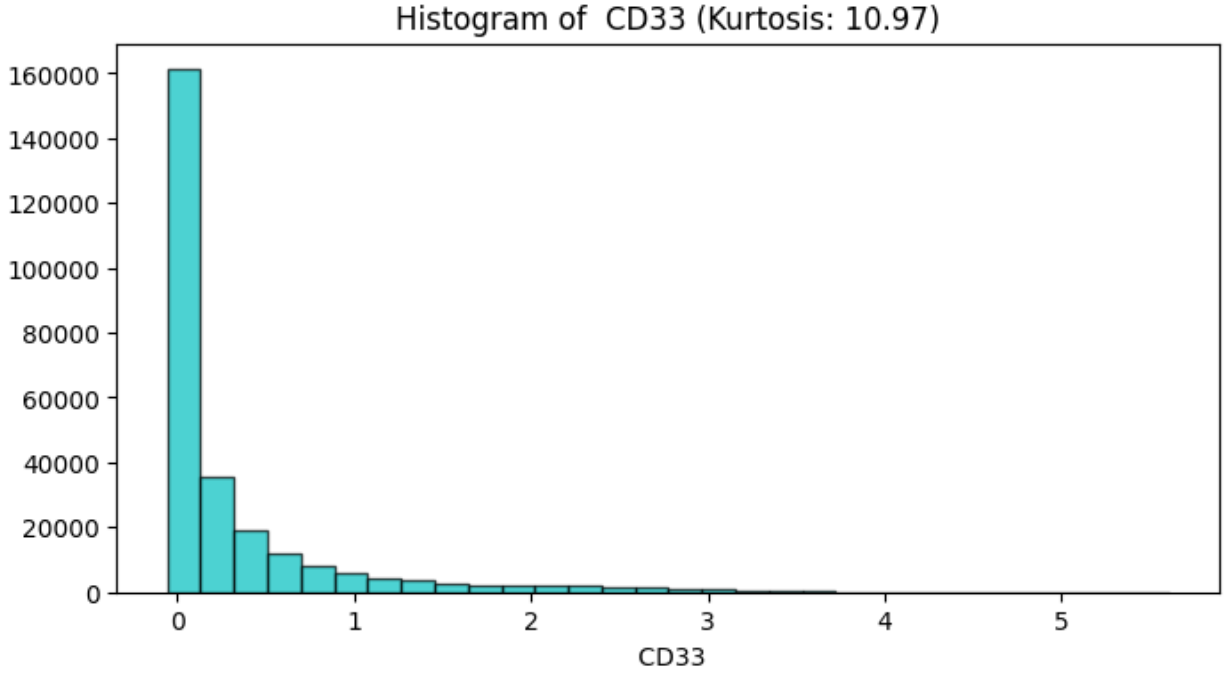

Histogram of CD235ab (Kurtosis: 13.44)
50000
40000
30000
20000
10000
0
0
1
2
3
4
5
6
CD235ab

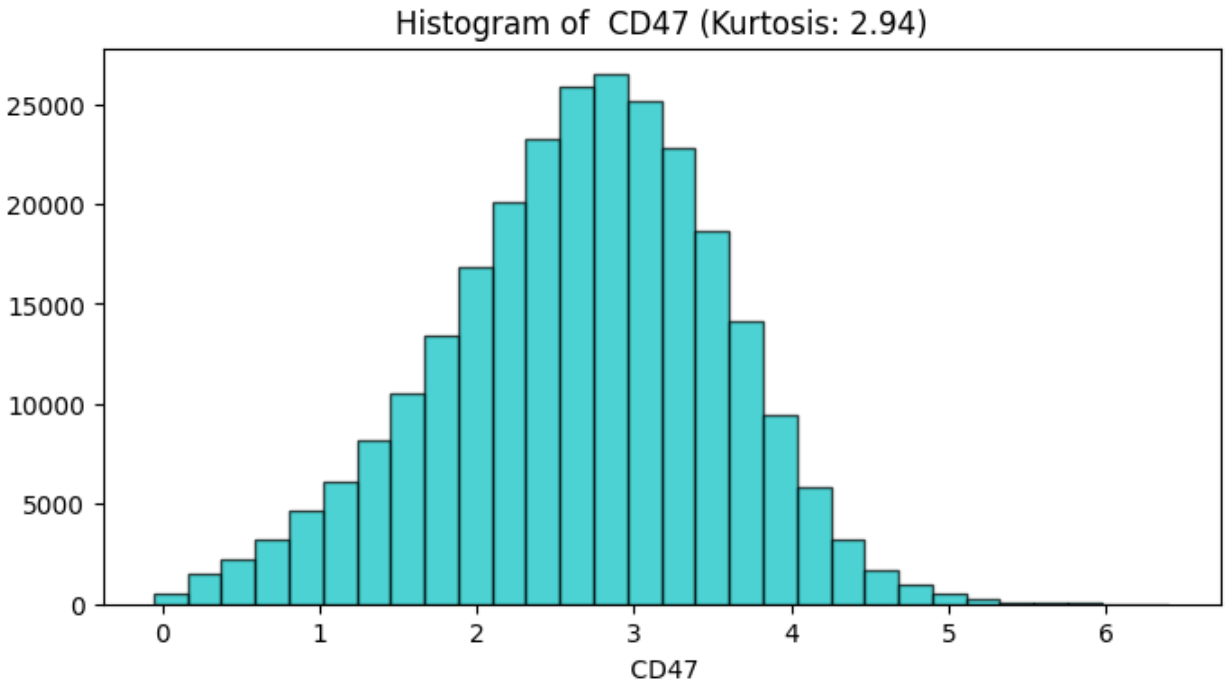

Histogram of CD45 (Kurtosis: 5.25)
40000
35000
30000
25000
20000
15000
10000
5000
0
2
3
4
5
6
7
CD45

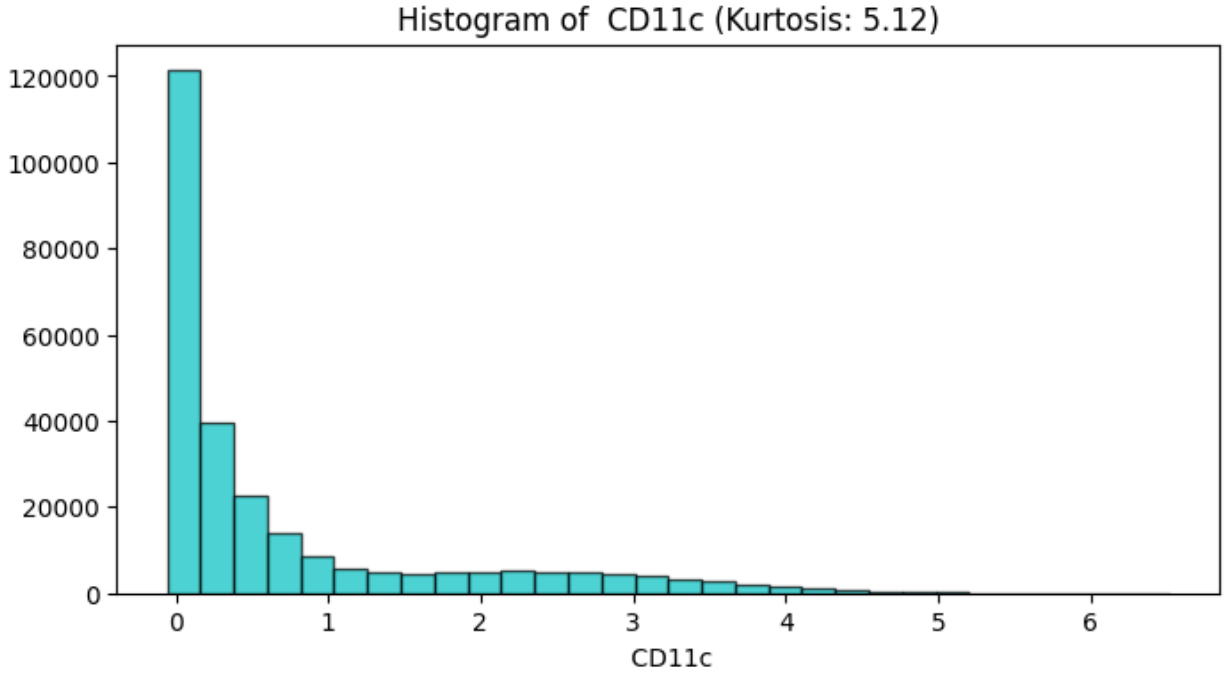

Histogram of CD123 (Kurtosis: 18.36)
160000
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
CD123

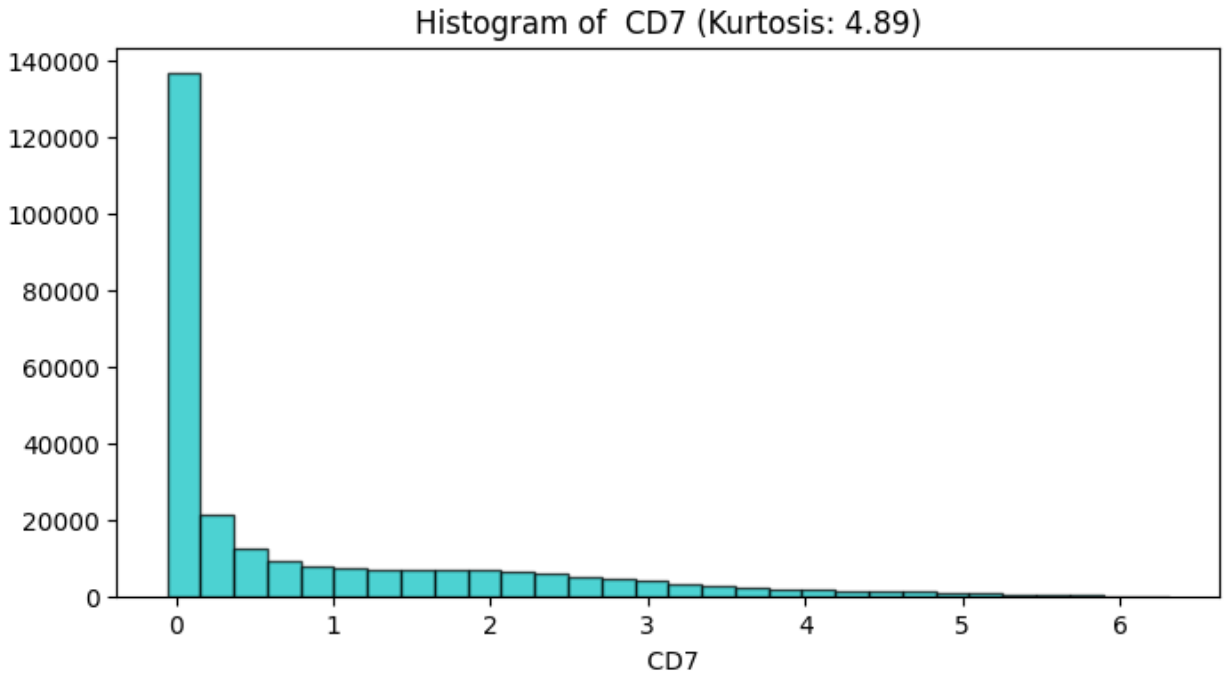

Histogram of CD321 (Kurtosis: 2.91)
25000
20000
15000
10000
5000
0
0
1
2
3
4
5
6
7
CD321

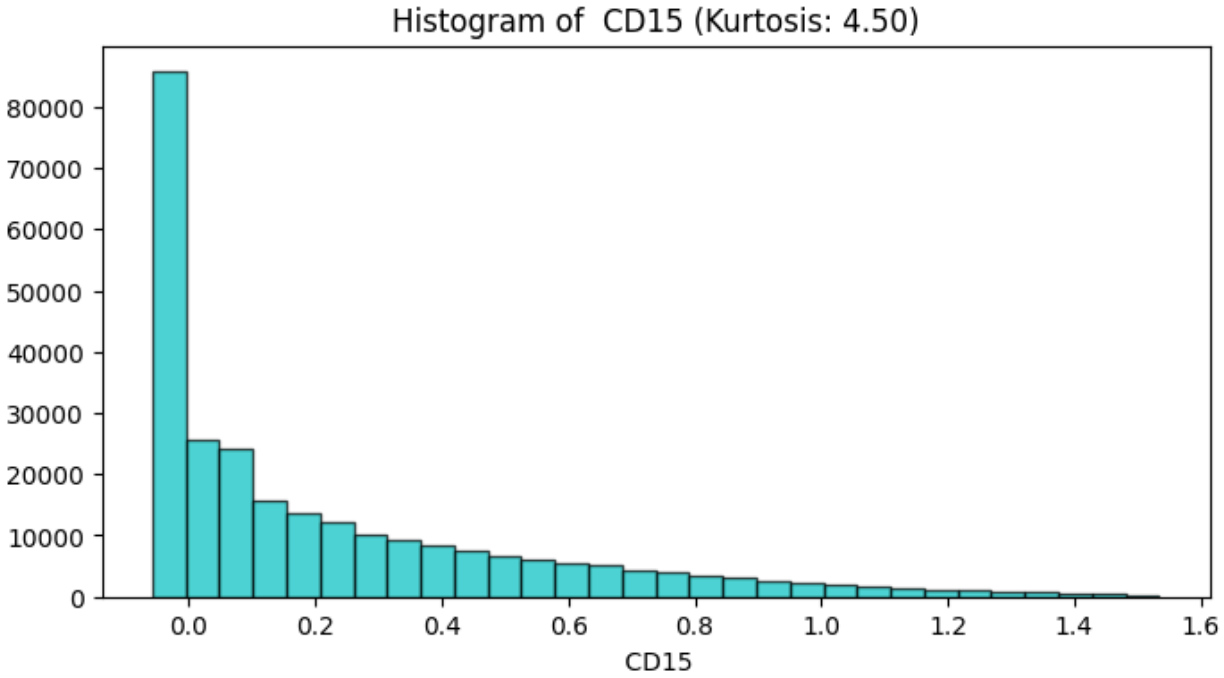

Histogram of CD14 (Kurtosis: 23.06)
200000
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5
CD14

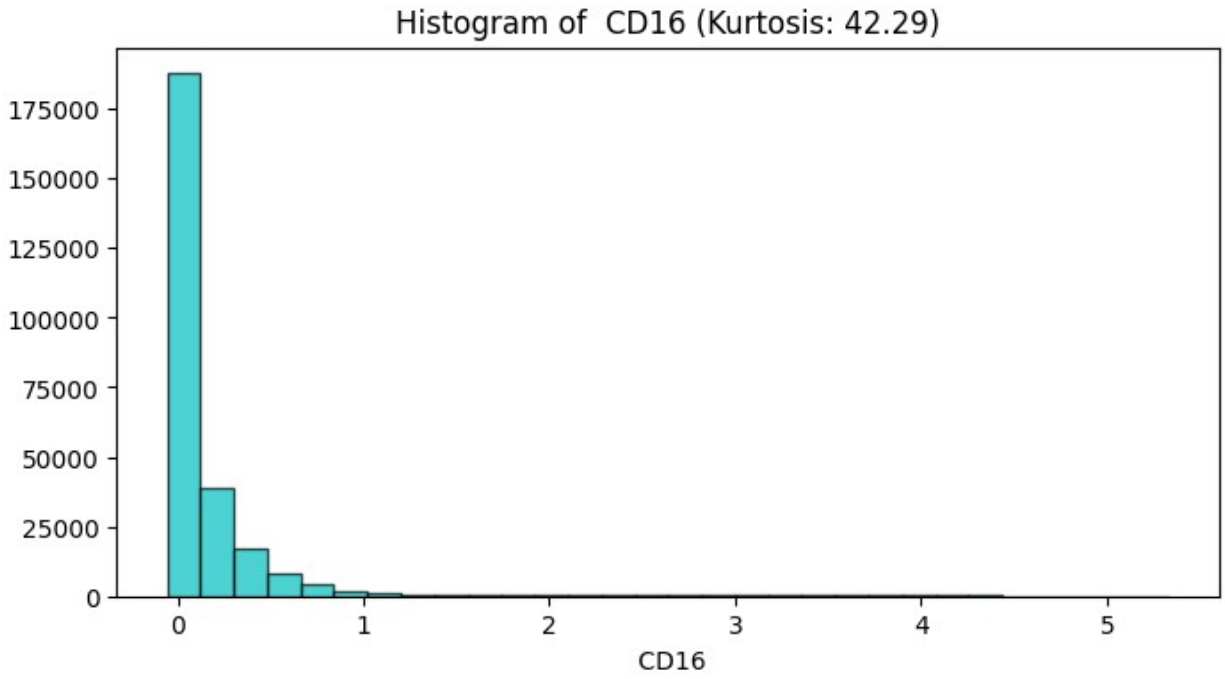

Histogram of CD33 (Kurtosis: 10.97)
160000
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
CD33

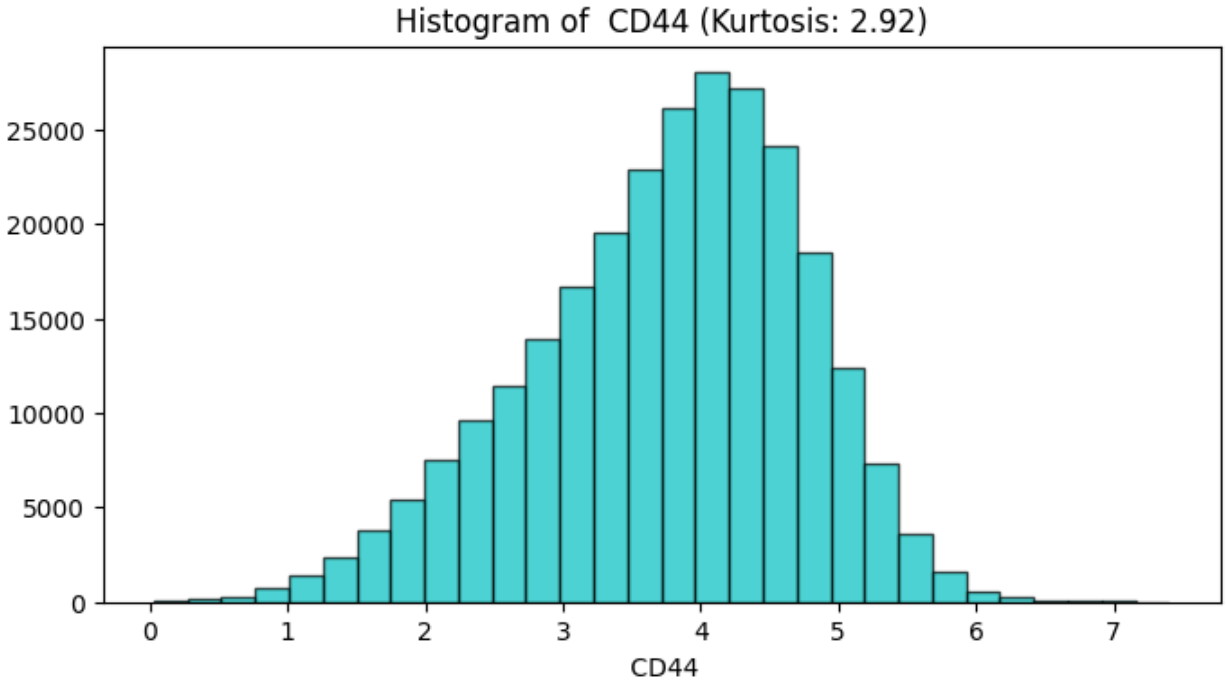

Histogram of CD47 (Kurtosis: 2.94)
25000
20000
15000
10000
5000
0
0
1
2
3
4
5
6
CD47

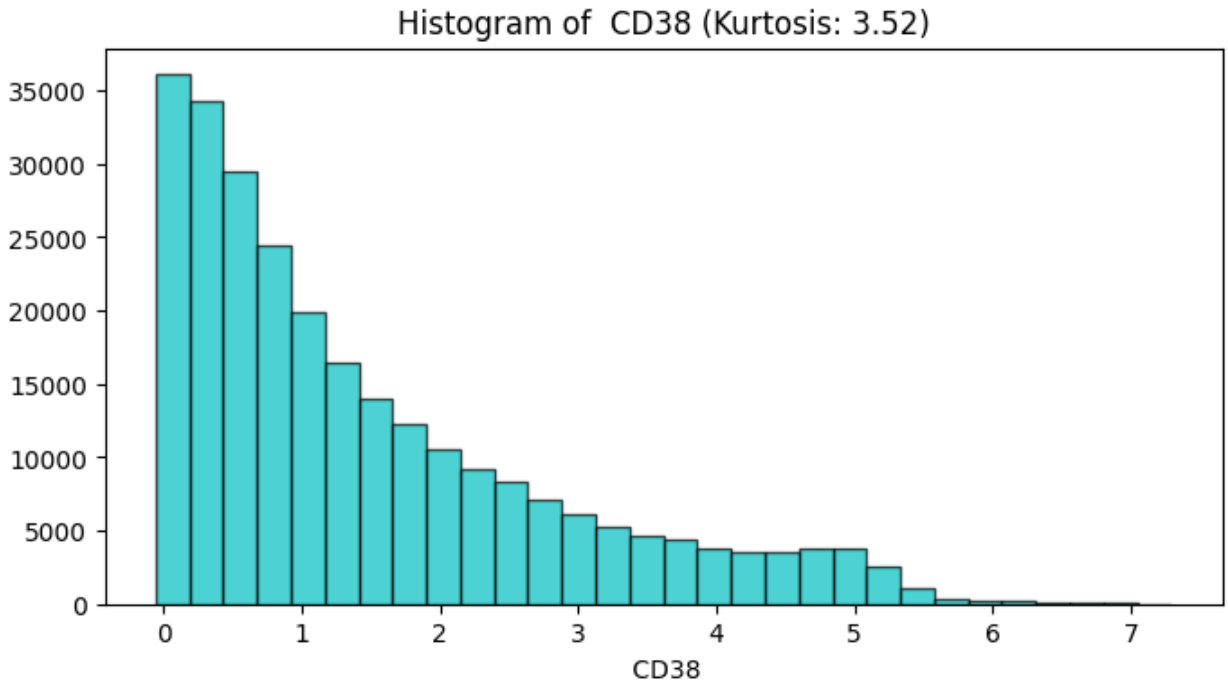

Histogram of CD11c (Kurtosis: 5.12)
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
CD11c

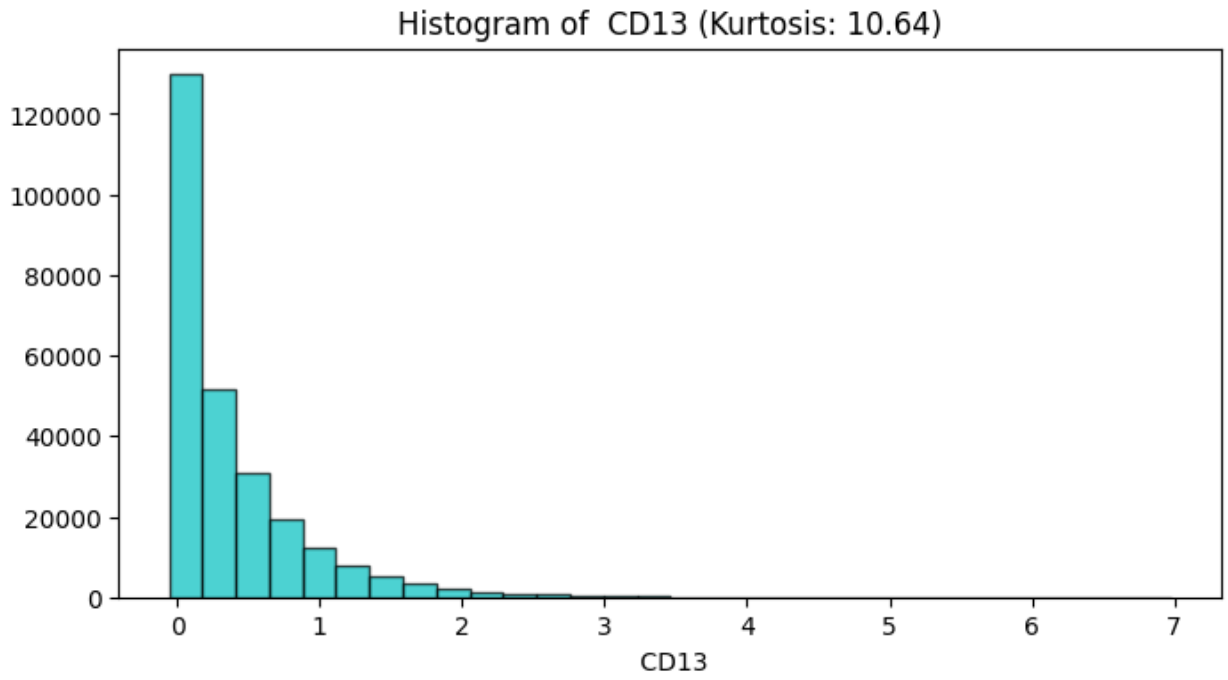

Histogram of CD7 (Kurtosis: 4.89)
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
CD7

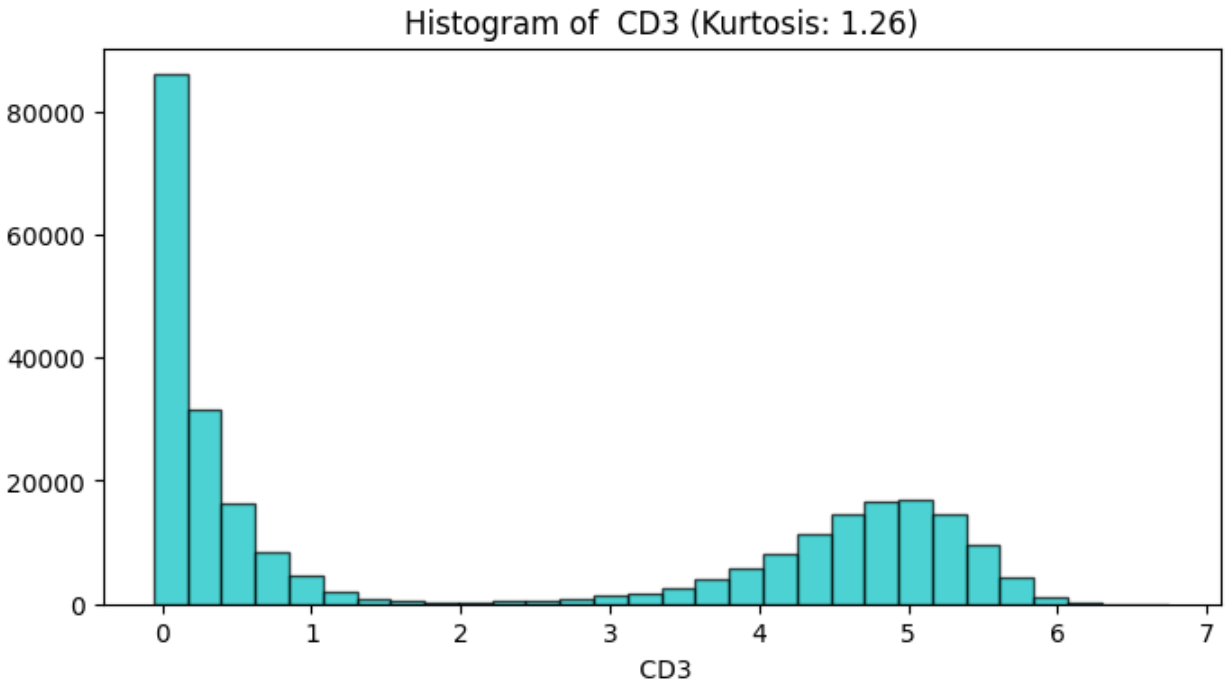

Histogram of CD15 (Kurtosis: 4.50)
80000
70000
60000
50000
40000
30000
20000
10000
0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
CD15

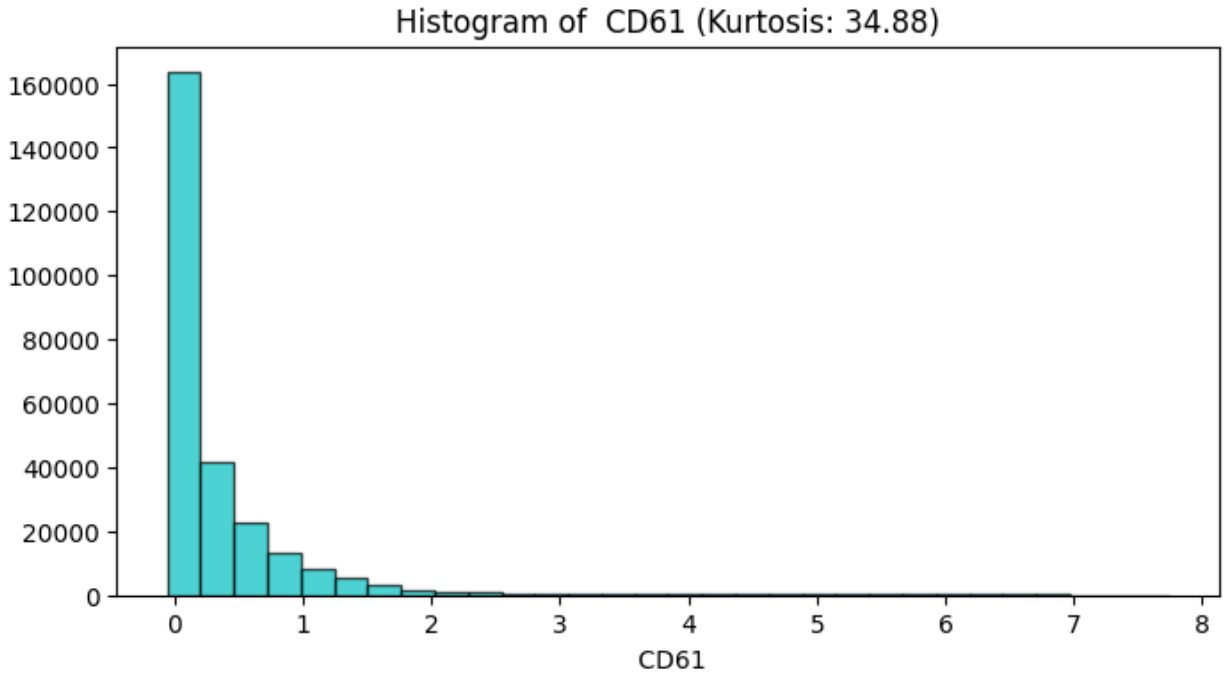

Histogram of CD16 (Kurtosis: 42.29)
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5
CD16

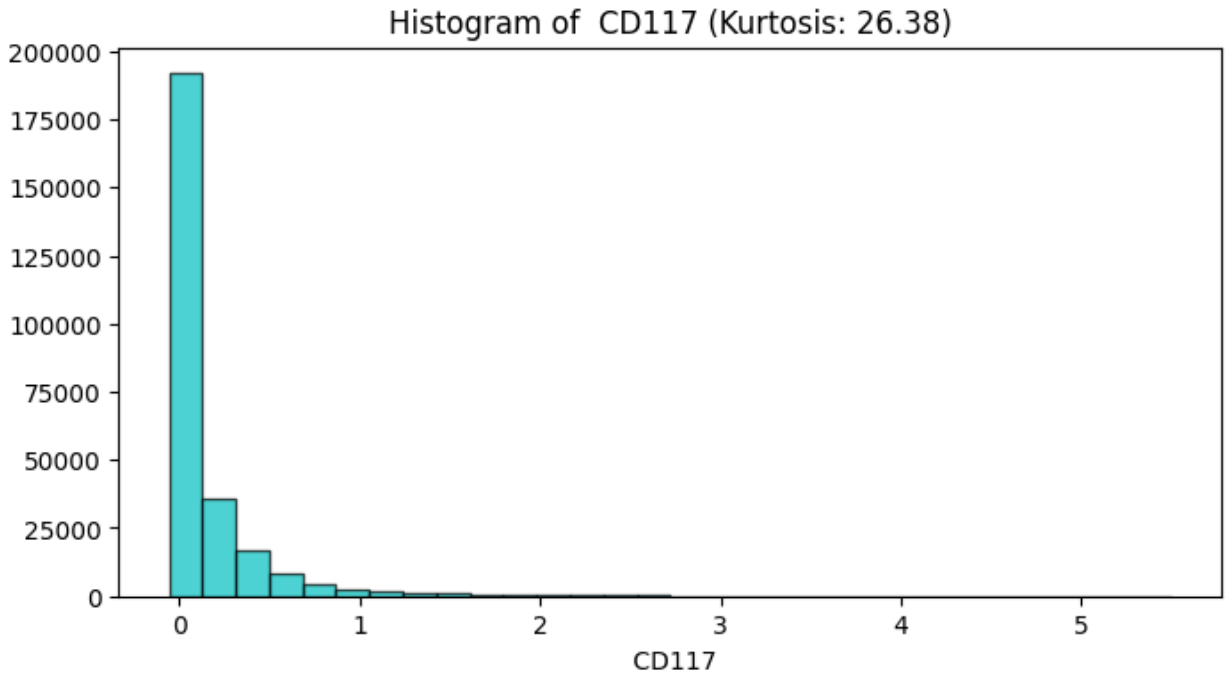

Histogram of CD44 (Kurtosis: 2.92)
25000
20000
15000
10000
5000
0
0
1
2
3
4
5
6
7
CD44

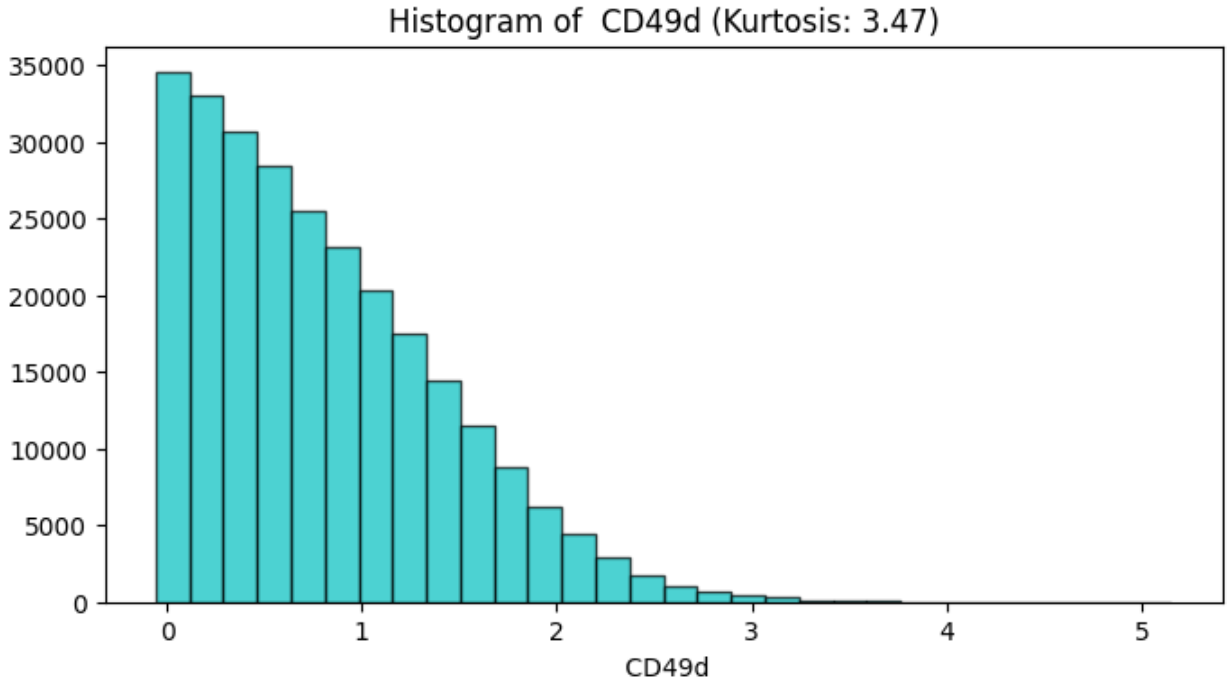

Histogram of CD38 (Kurtosis: 3.52)
35000
30000
25000
20000
15000
10000
5000
0
0
1
2
3
4
5
6
7
CD38

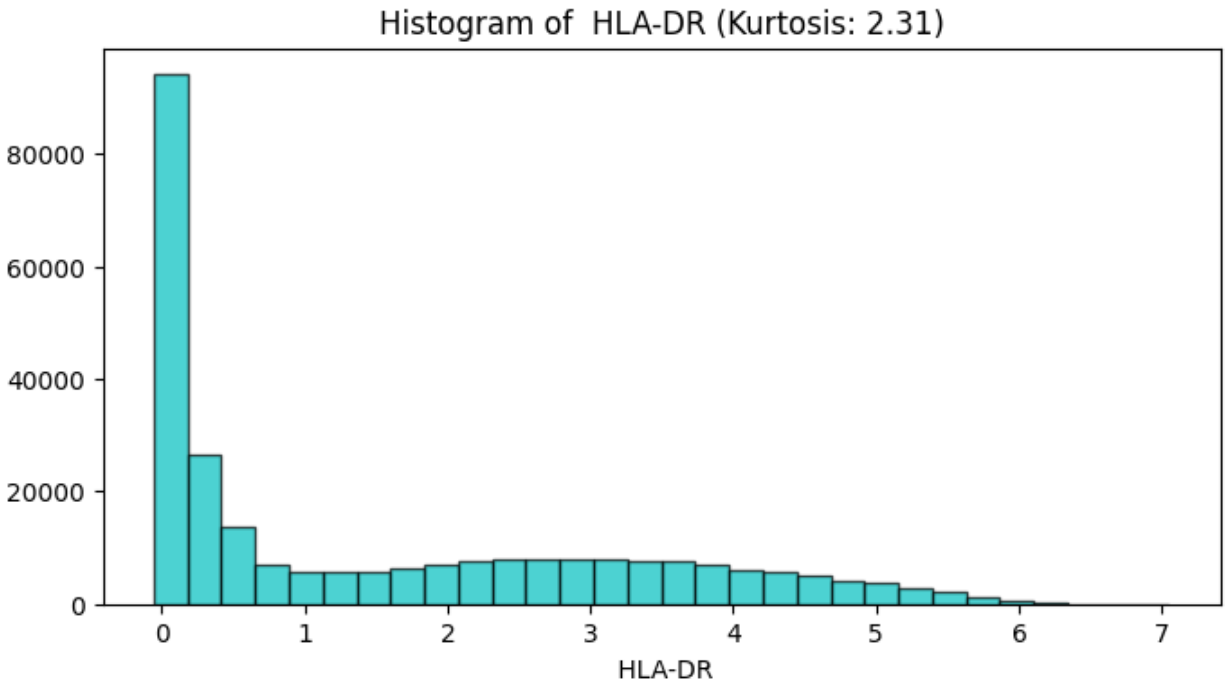

Histogram of CD13 (Kurtosis: 10.64)
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7
CD13

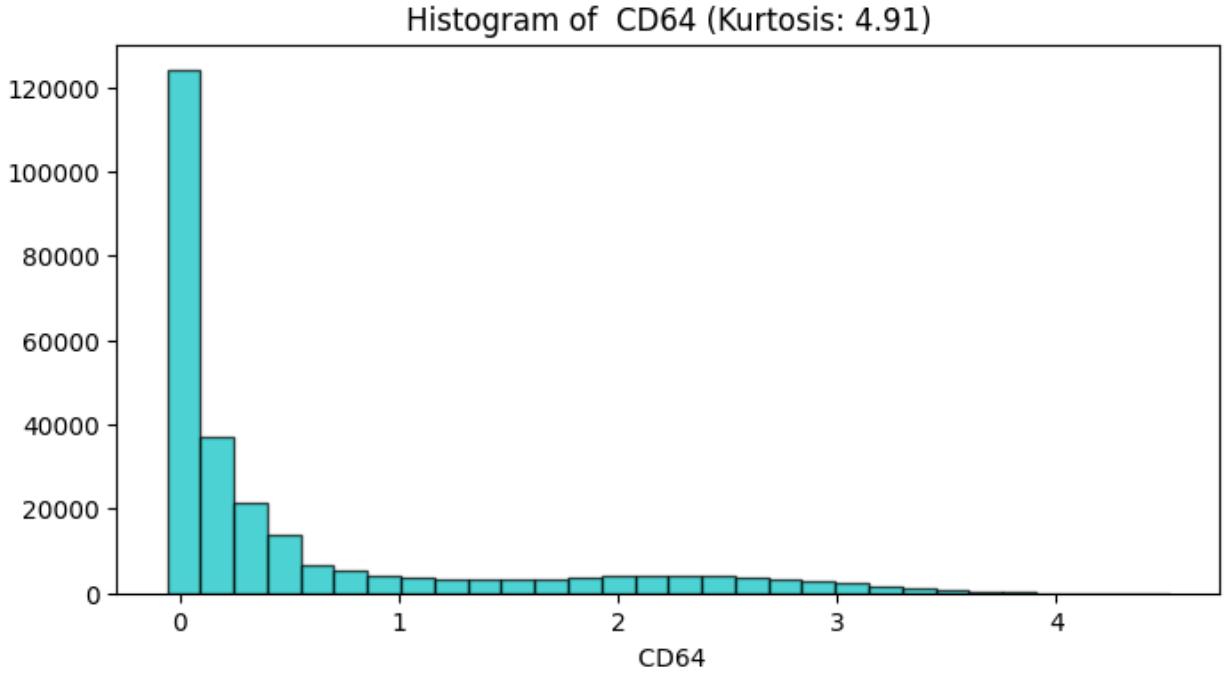

Histogram of CD3 (Kurtosis: 1.26)
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7
CD3

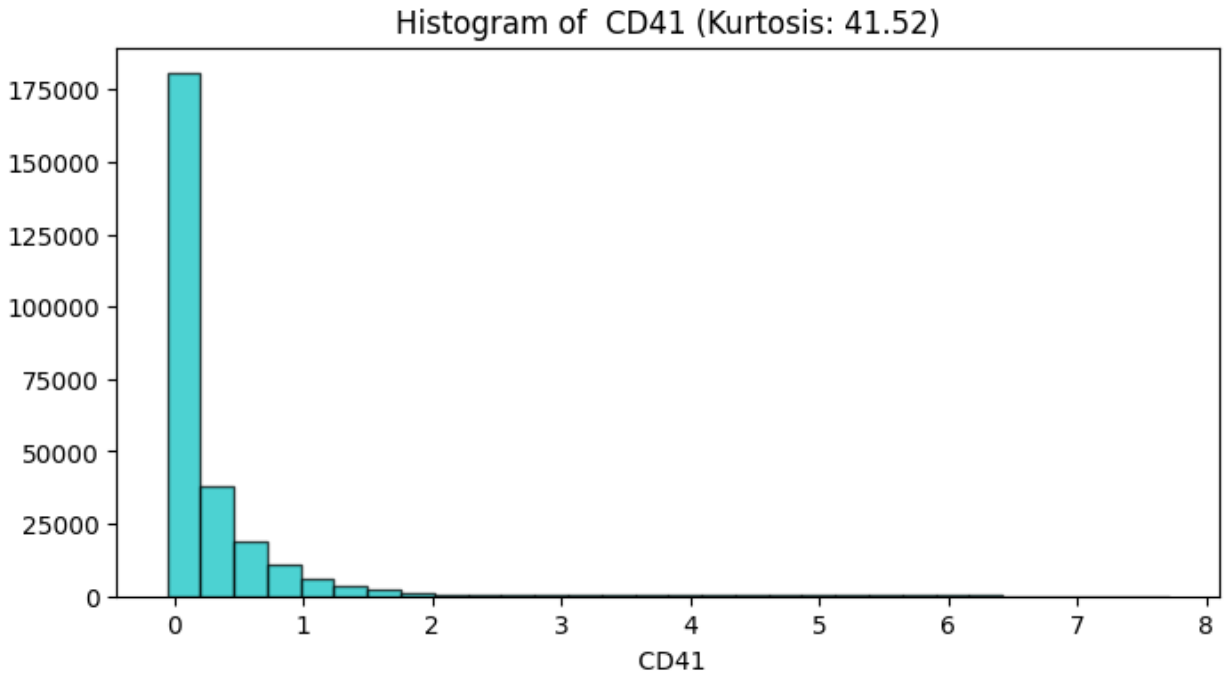

Histogram of CD61 (Kurtosis: 34.88)
160000
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7
8
CD61

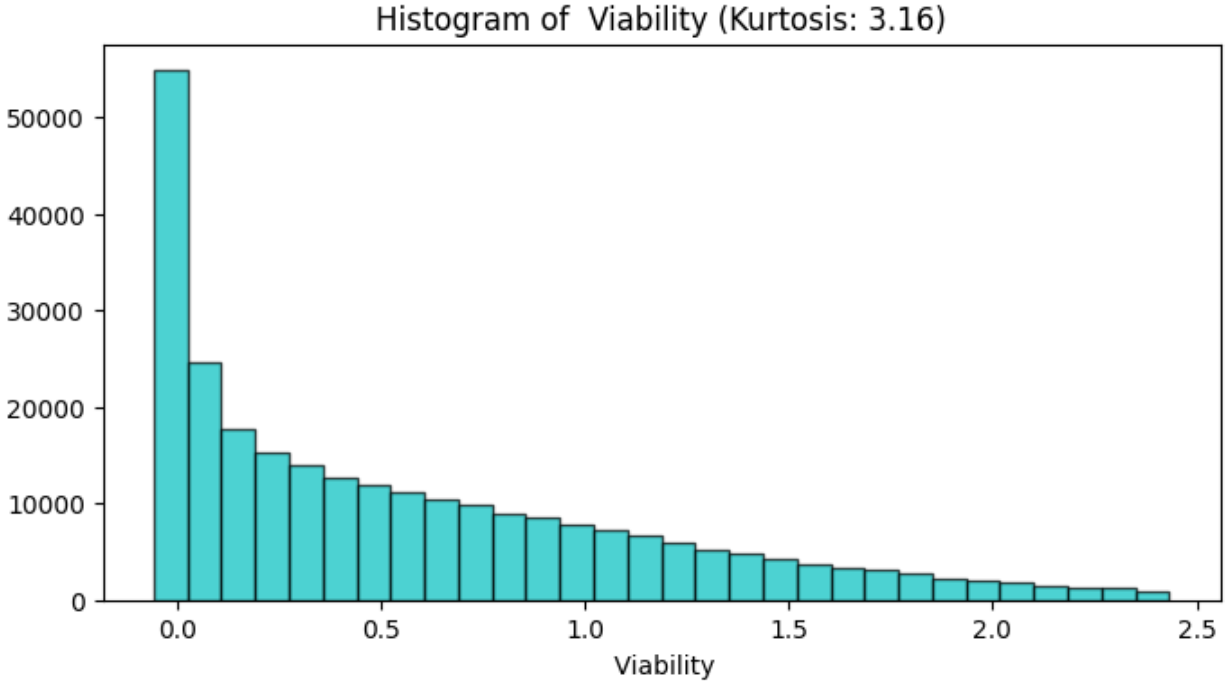

Histogram of CD117 (Kurtosis: 26.38)
200000
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5
CD117

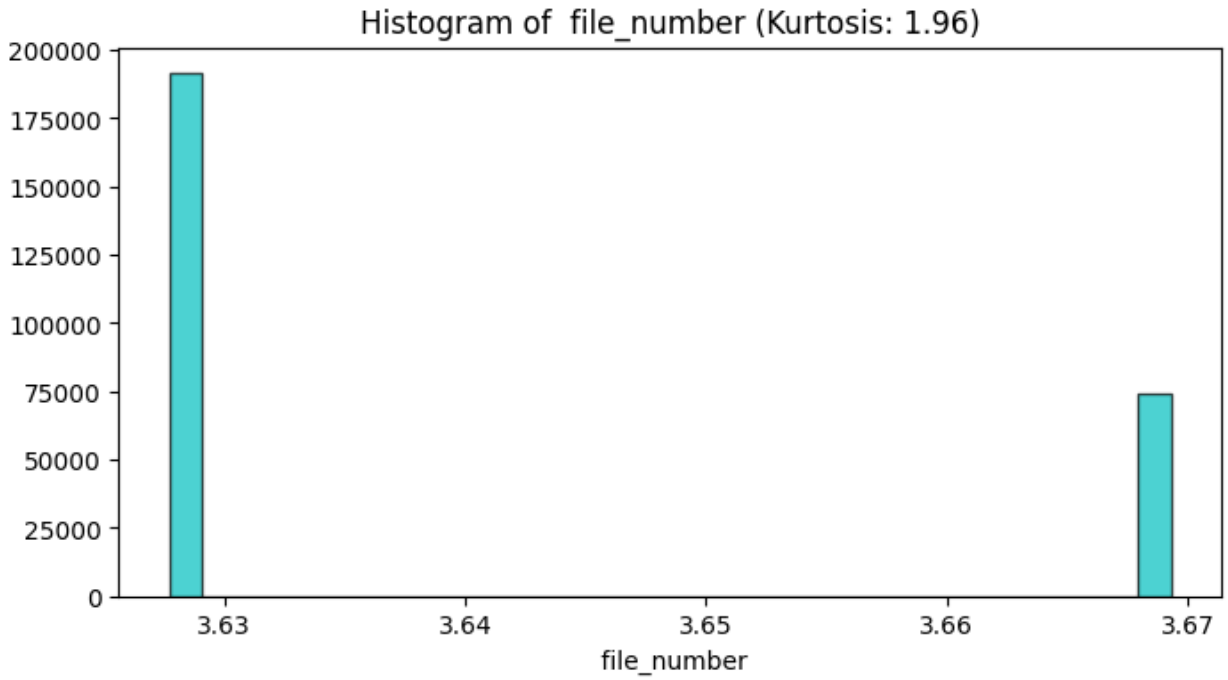

Histogram of CD49d (Kurtosis: 3.47)
35000
30000
25000
20000
15000
10000
5000
0
0
1
2
3
4
5
CD49d

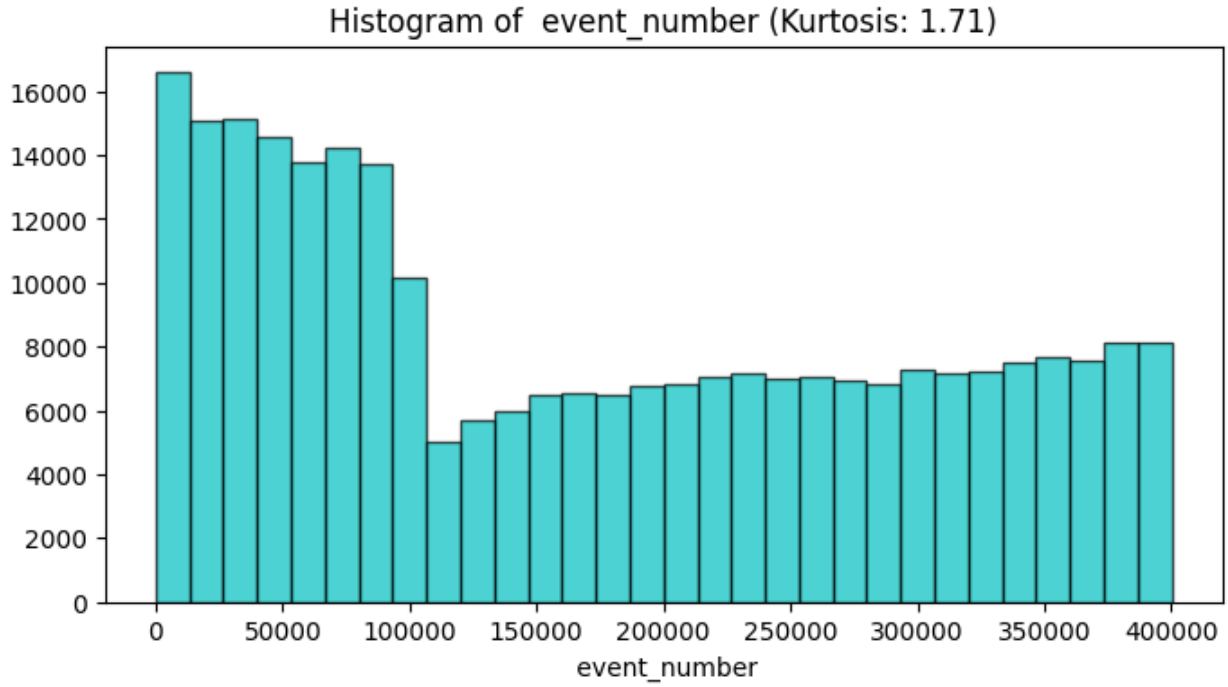

Histogram of HLA-DR (Kurtosis: 2.31)
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7
HLA-DR

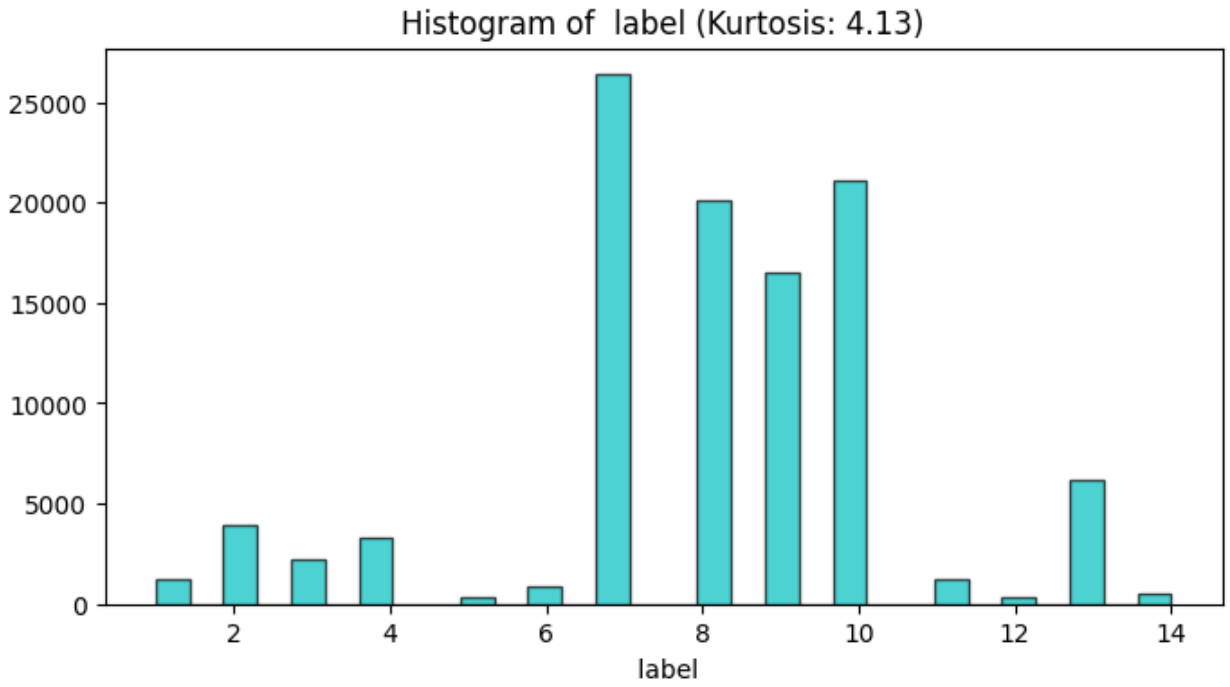

Histogram of CD64 (Kurtosis: 4.91)
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
CD64

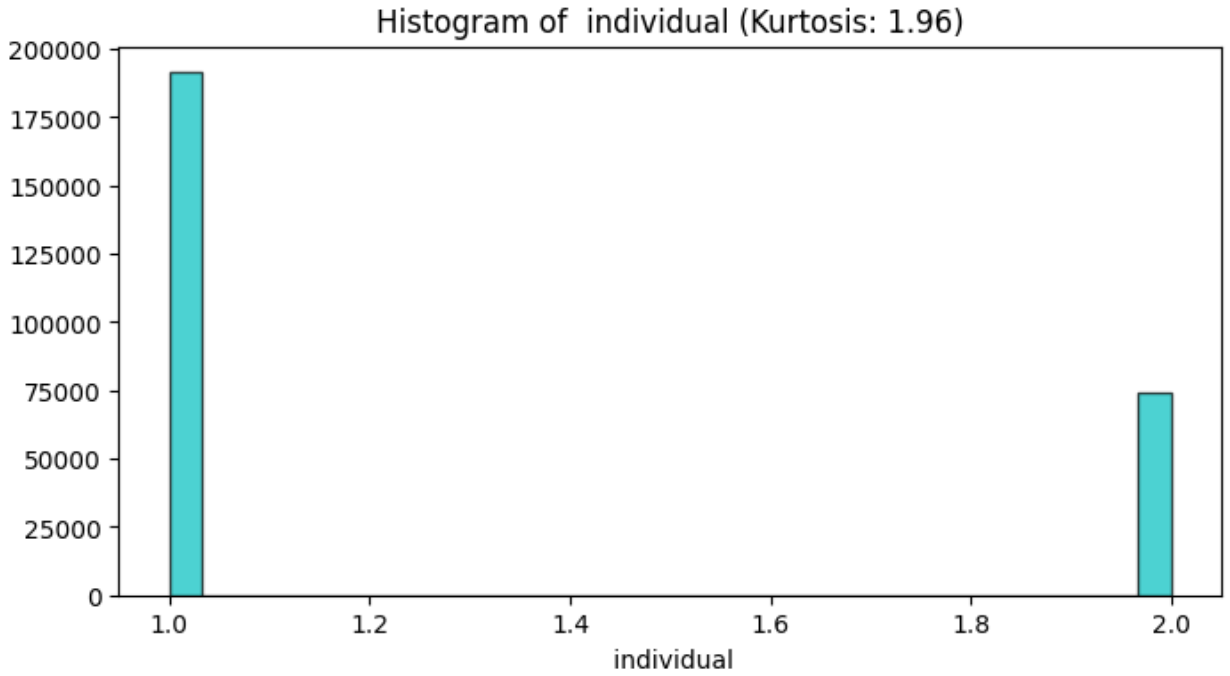

Histogram of CD41 (Kurtosis: 41.52)
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5
6
7
8
CD41

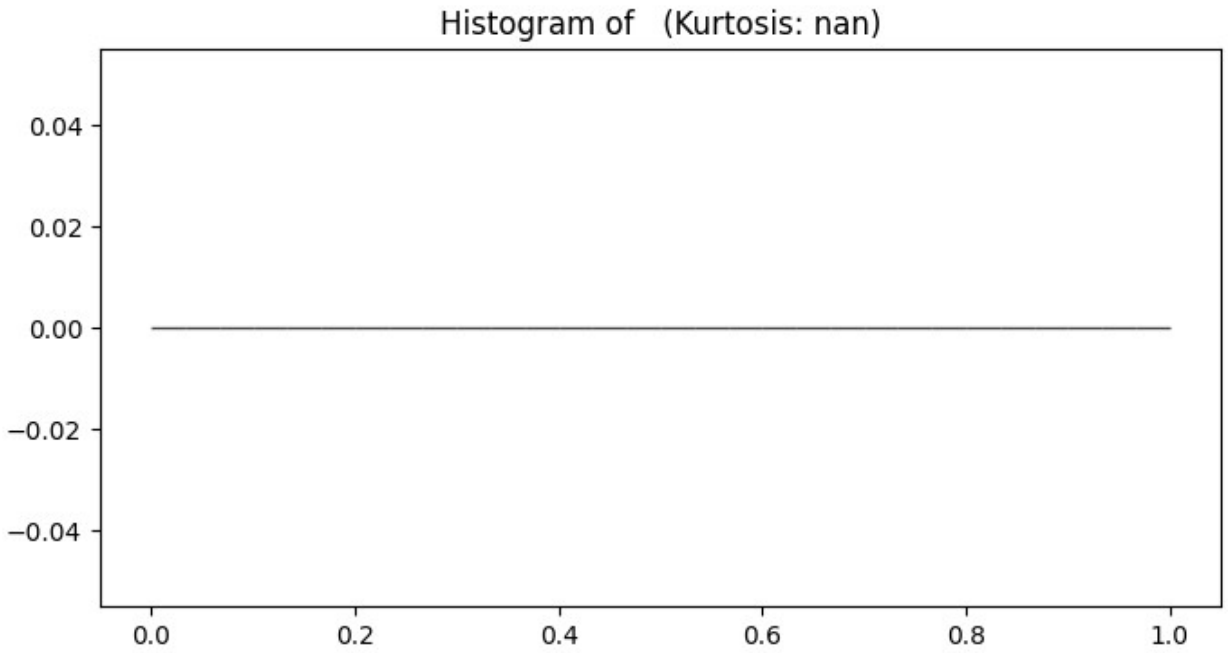

Histogram of Viability (Kurtosis: 3.16)
50000
40000
30000
20000
10000
0
0.0
0.5
1.0
1.5
2.0
2.5
Viability

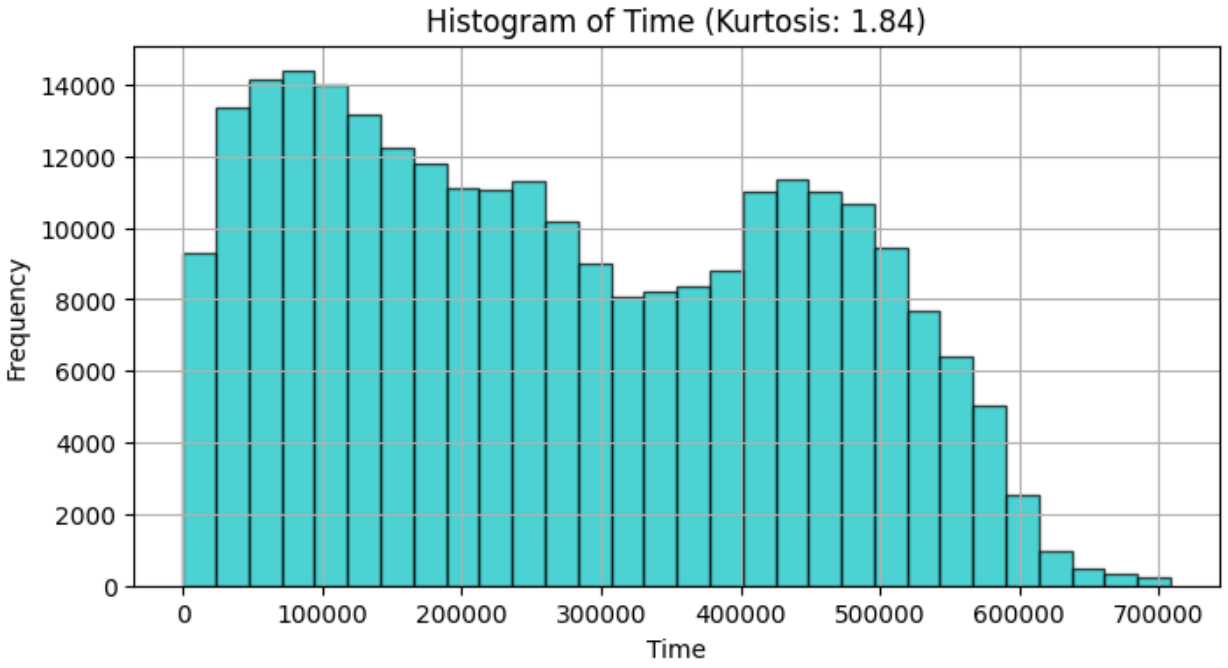

Histogram of file_number (Kurtosis: 1.96)
200000
175000
150000
125000
100000
75000
50000
25000
0
3.63
3.64
3.65
3.66
3.67
file_number

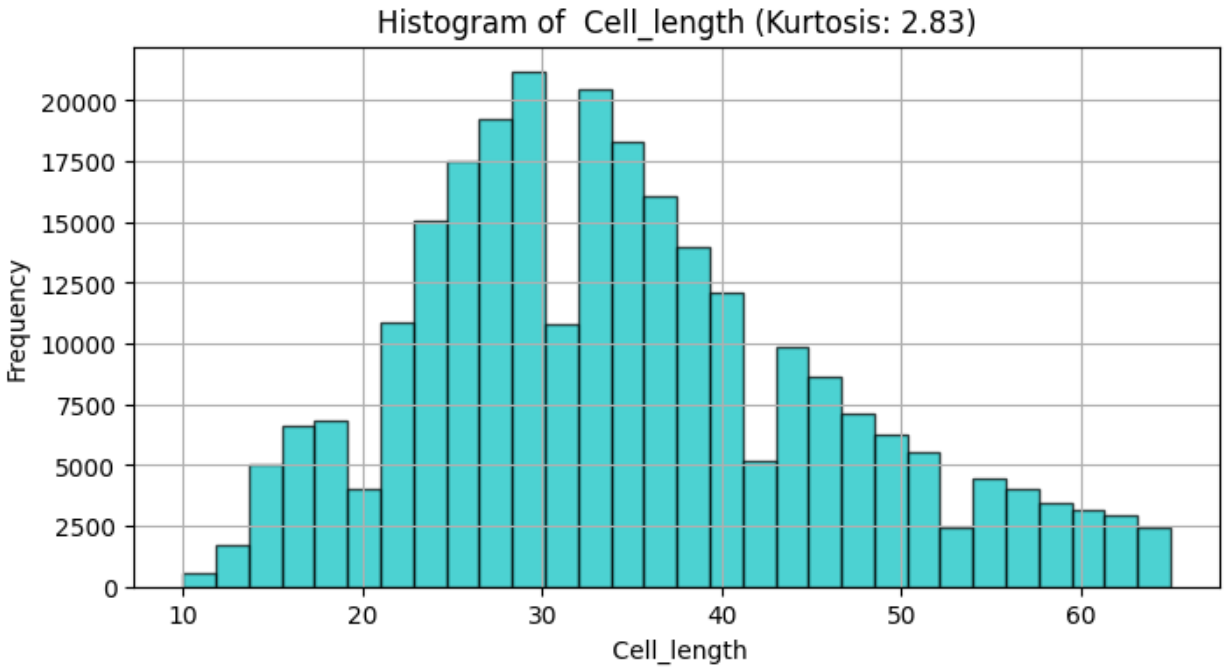

Histogram of event_number (Kurtosis: 1.71)
16000
14000
12000
10000
8000
6000
4000
2000
0
0
50000
100000
150000
200000
250000
300000
350000
400000
event_number

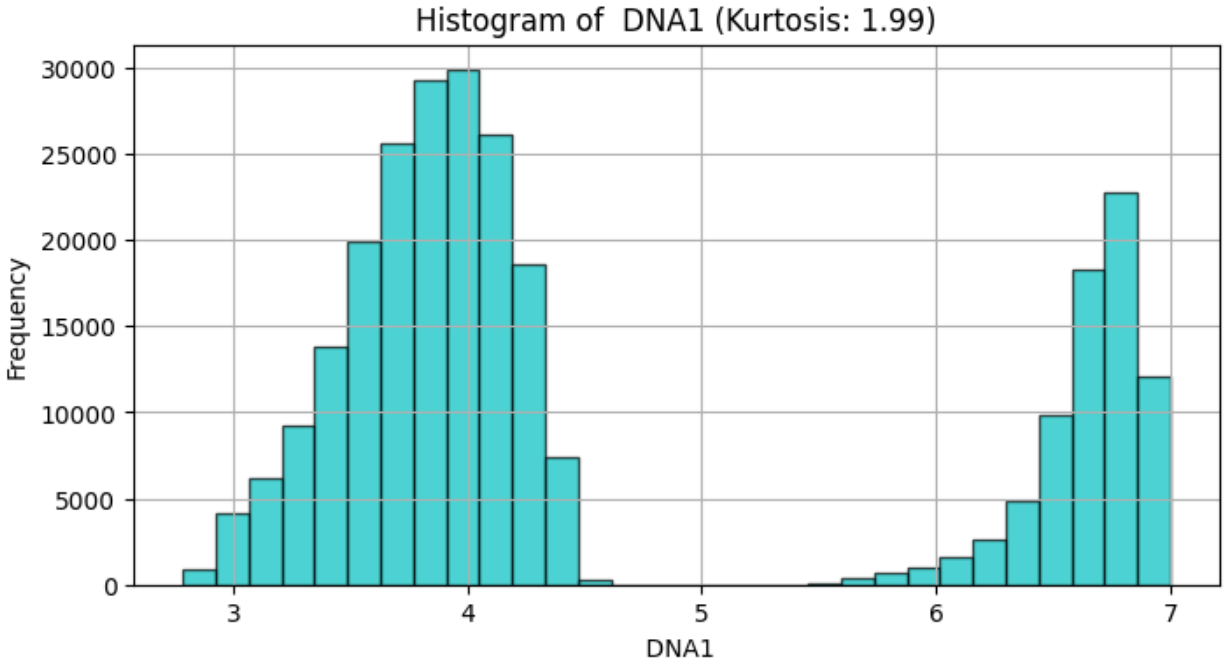

Histogram of label (Kurtosis: 4.13)
25000
20000
15000
10000
5000
0
2
4
6
8
10
12
14
label

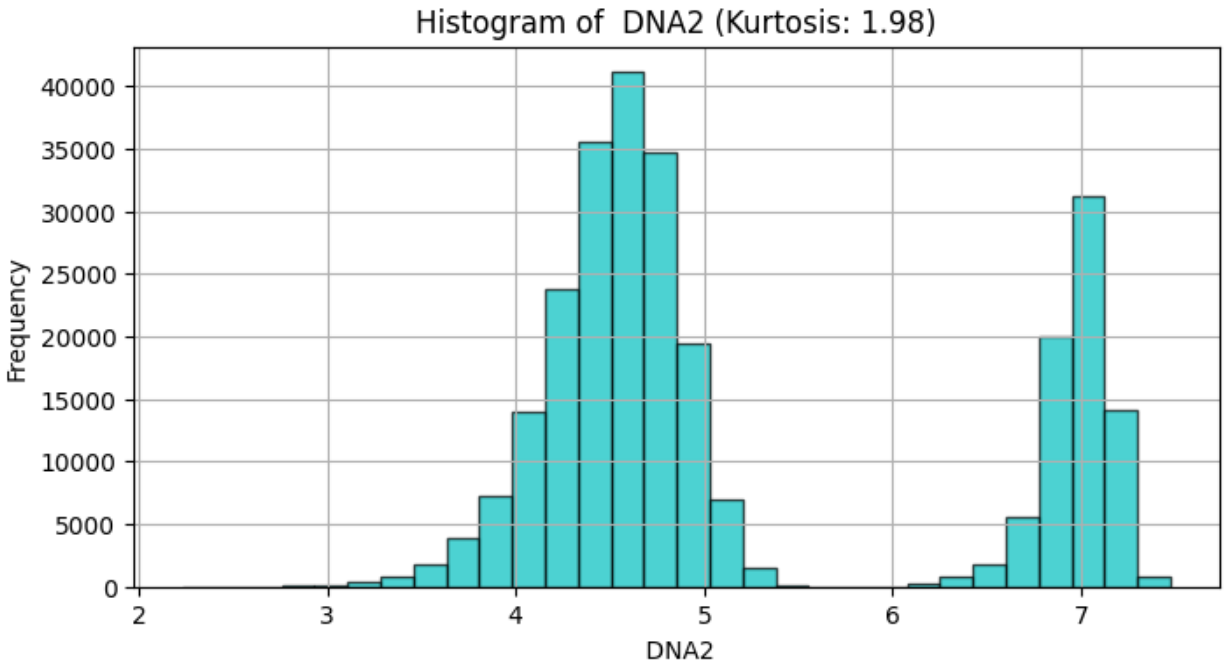

Histogram of individual (Kurtosis: 1.96)
200000
175000
150000
125000
100000
75000
50000
25000
0
1.0
1.2
1.4
1.6
1.8
2.0
individual

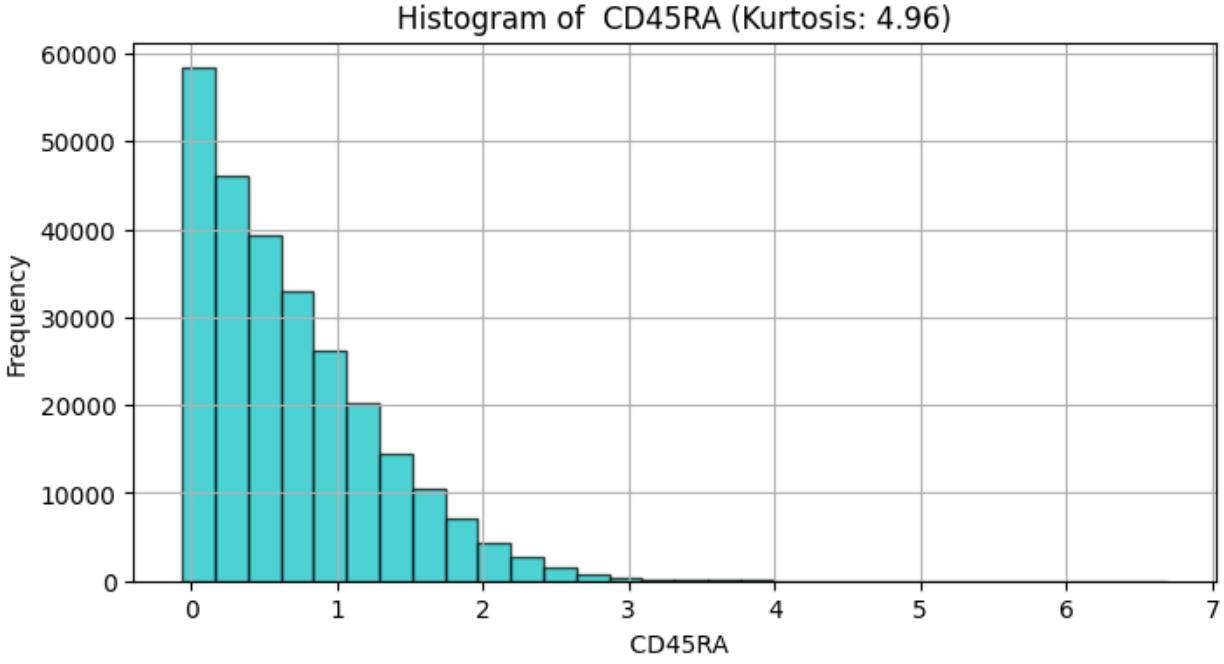

Histogram of (Kurtosis: nan)
0.04
0.02
0.00
−0.02
−0.04
0.0
0.2
0.4
0.6
0.8
1.0

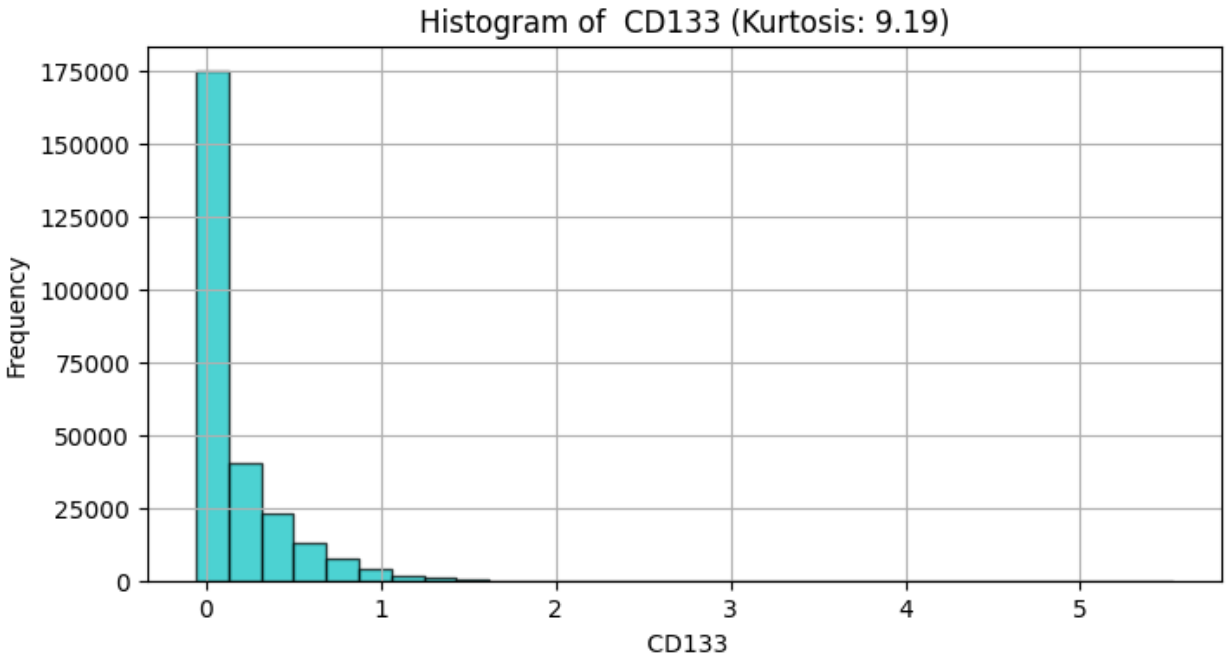

Histogram of Time (Kurtosis: 1.84)
Frequency
14000
12000
10000
8000
6000
4000
2000
0
0
100000
200000
300000
400000
500000
600000
700000
Time

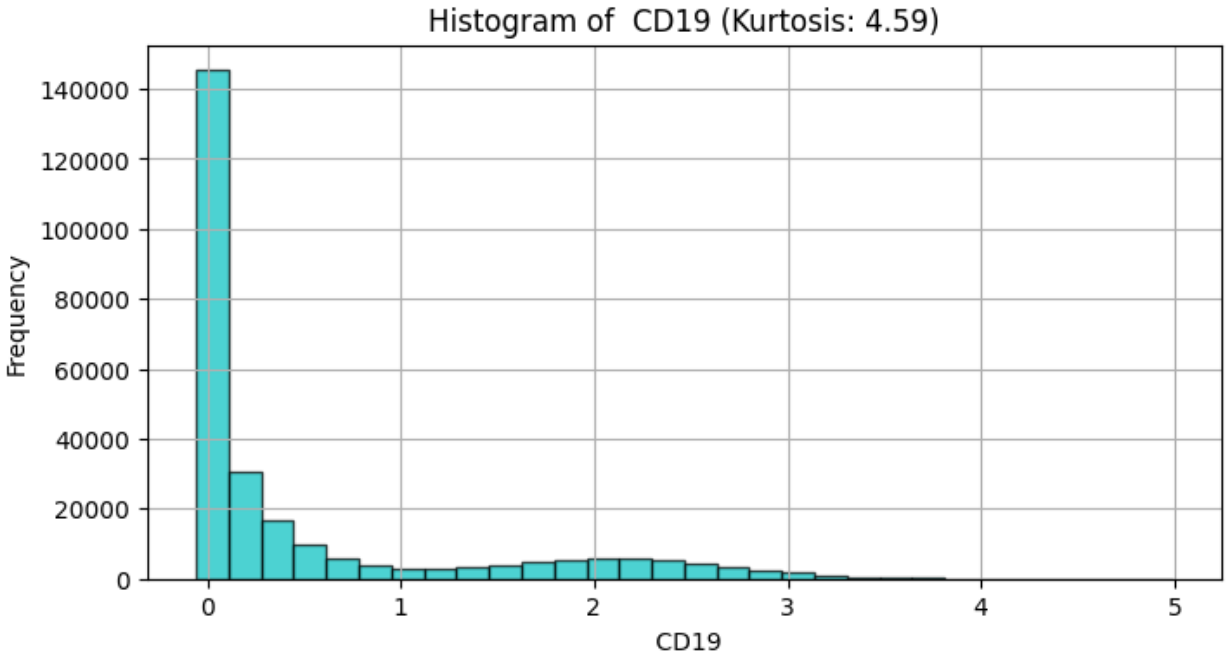

Histogram of Cell_length (Kurtosis: 2.83)
Frequency
Cell_length
0
2500
5000
7500
10000
12500
15000
17500
20000
10
20
30
40
50
60

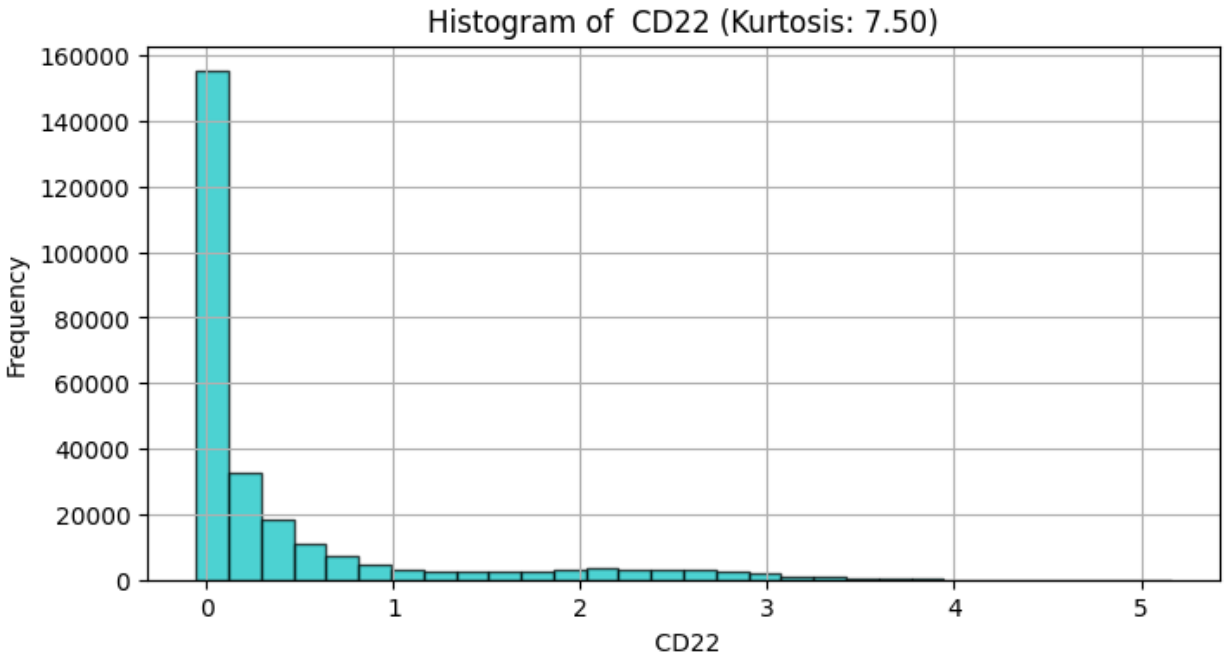

Histogram of DNA1 (Kurtosis: 1.99)
Frequency
DNA1
0
5000
10000
15000
20000
25000
30000
3
4
5
6
7

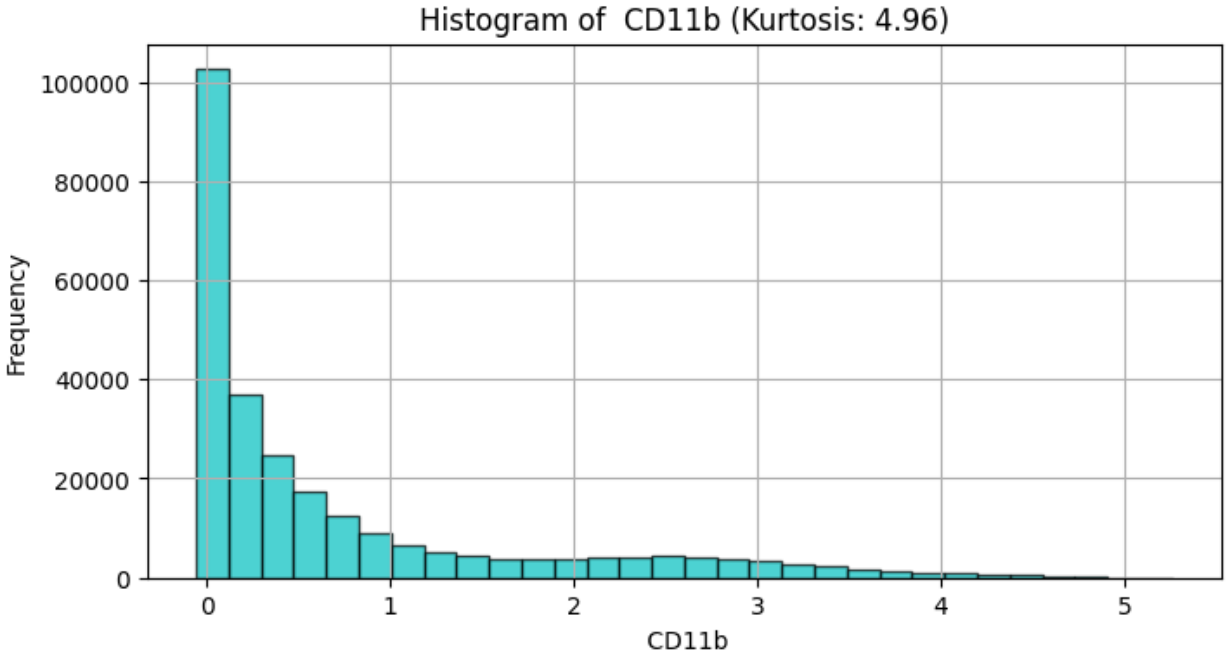

Histogram of DNA2 (Kurtosis: 1.98)
Frequency
DNA2
0
5000
10000
15000
20000
25000
30000
35000
40000
2
3
4
5
6
7

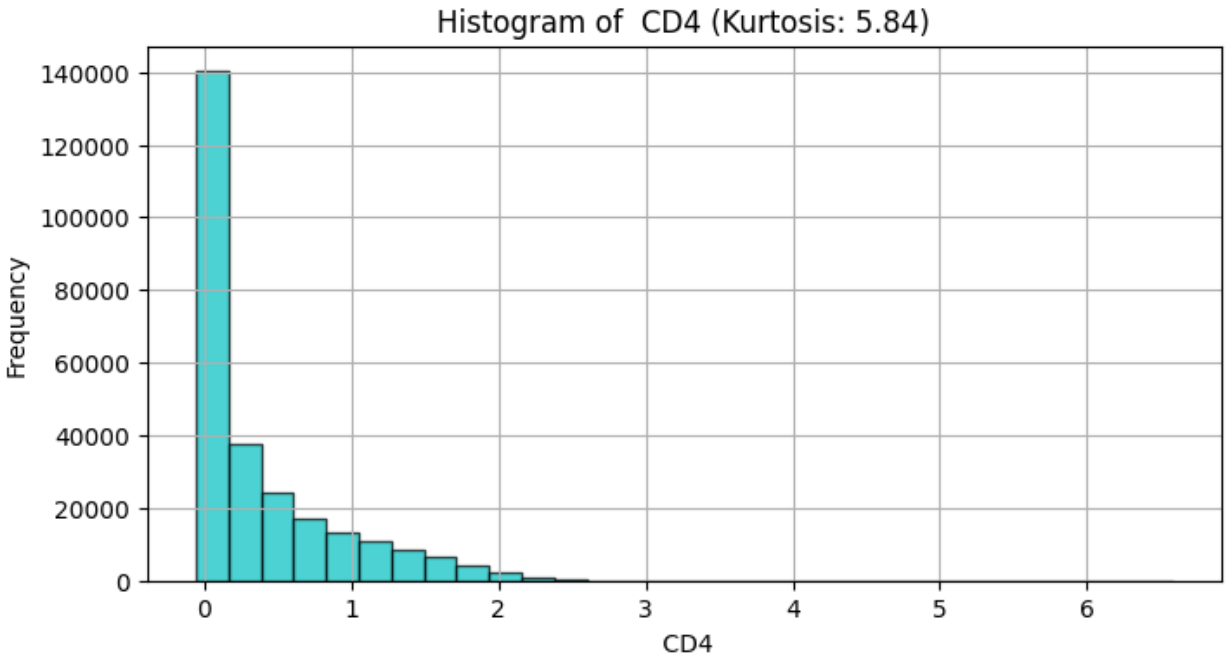

Histogram of CD45RA (Kurtosis: 4.96)
Frequency
CD45RA
0
10000
20000
30000
40000
50000
60000
0
1
2
3
4
5
6
7

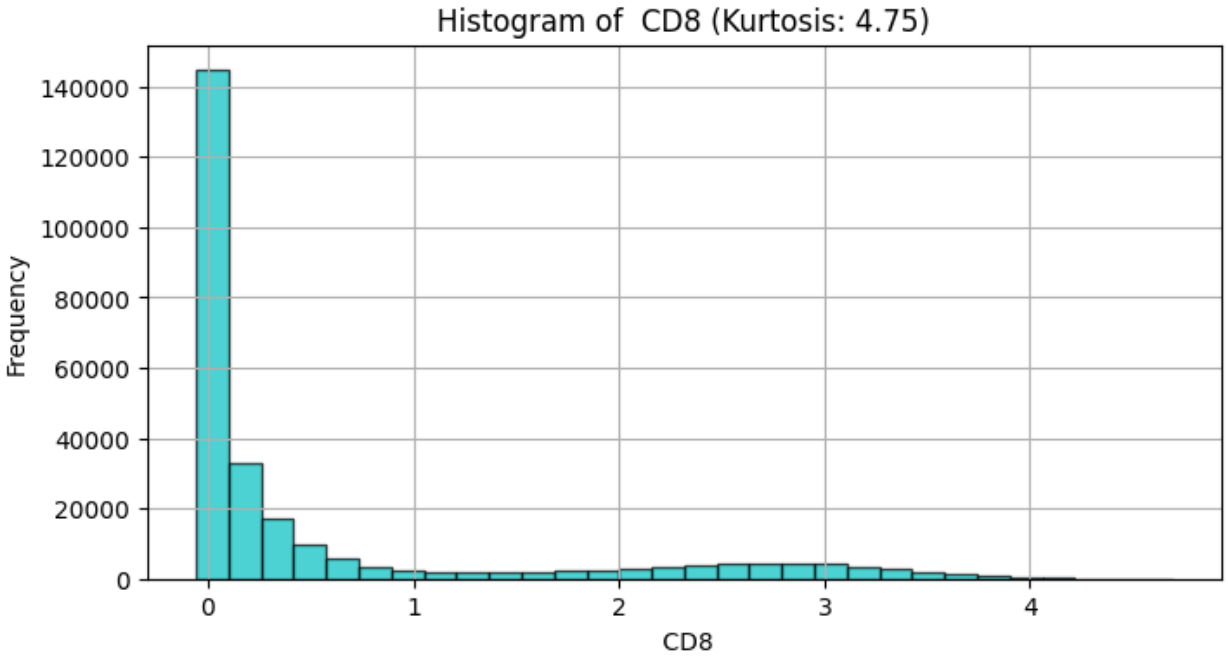

Histogram of CD133 (Kurtosis: 9.19)
Frequency
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5
CD133

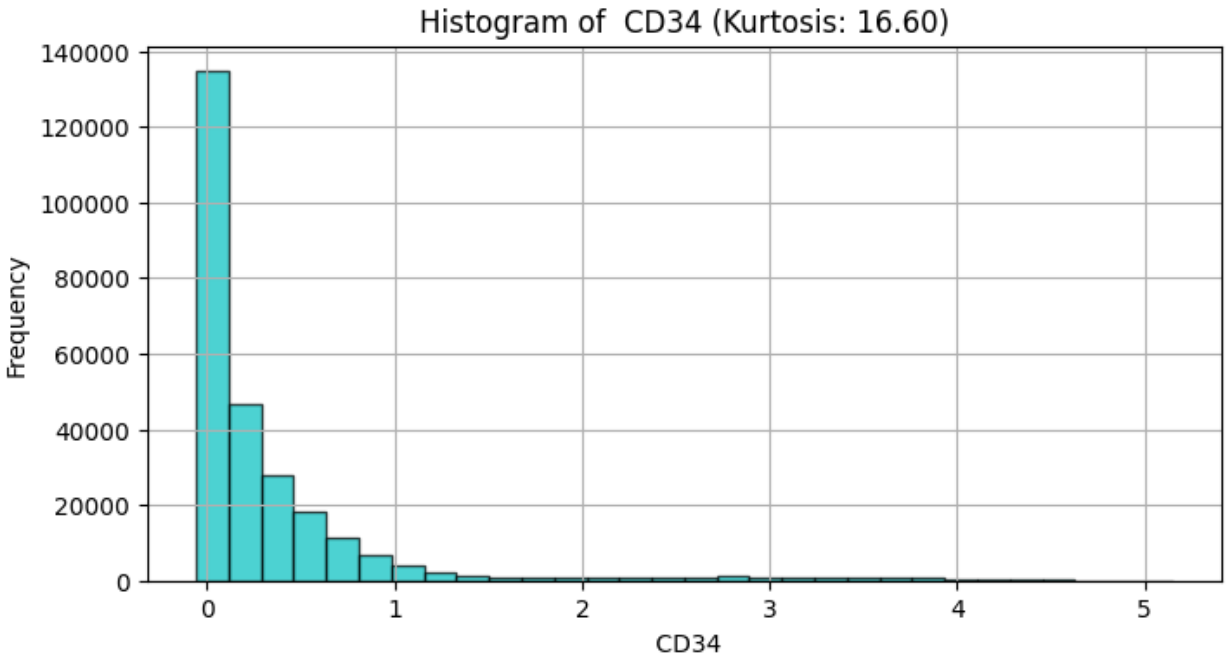

Histogram of CD19 (Kurtosis: 4.59)
Frequency
140000
120000
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
CD19

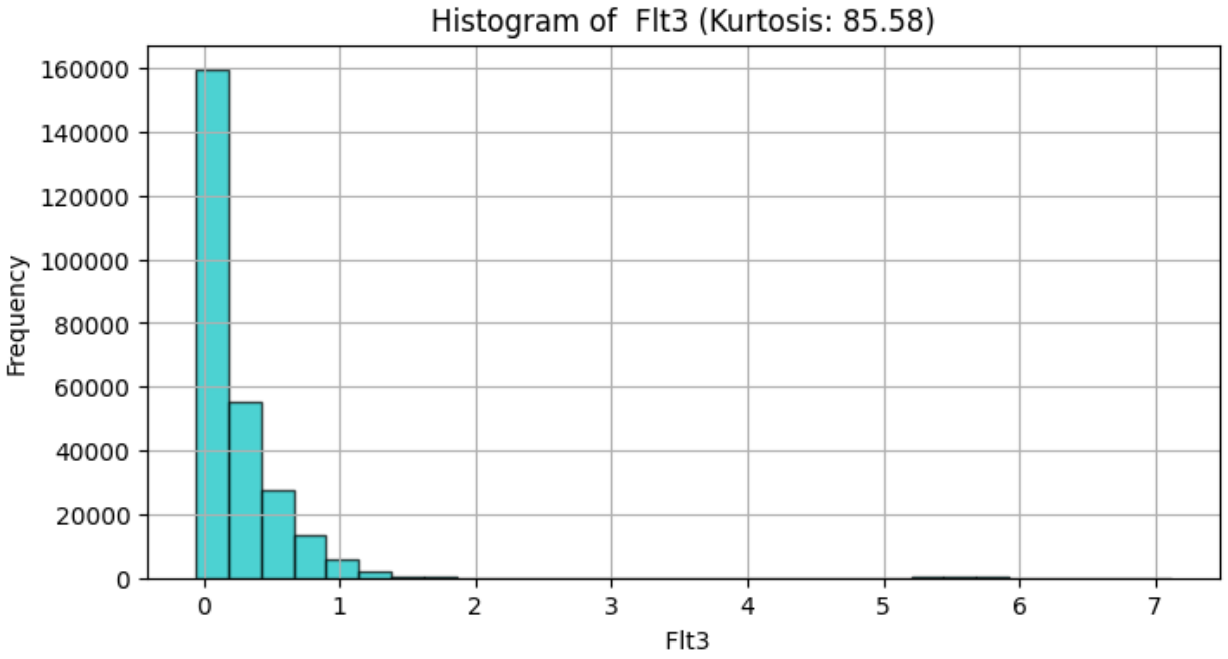

Histogram of CD22 (Kurtosis: 7.50)
Frequency
CD22
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5

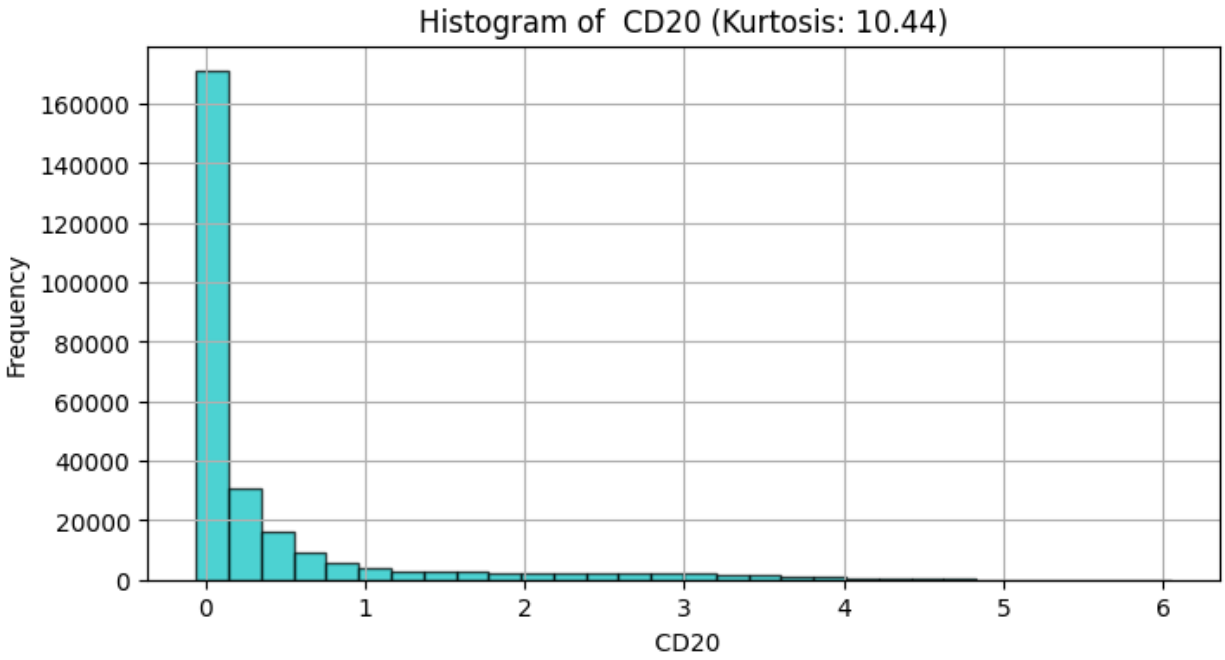

Histogram of CD11b (Kurtosis: 4.96)
Frequency
CD11b
0
20000
40000
60000
80000
100000
0
1
2
3
4
5

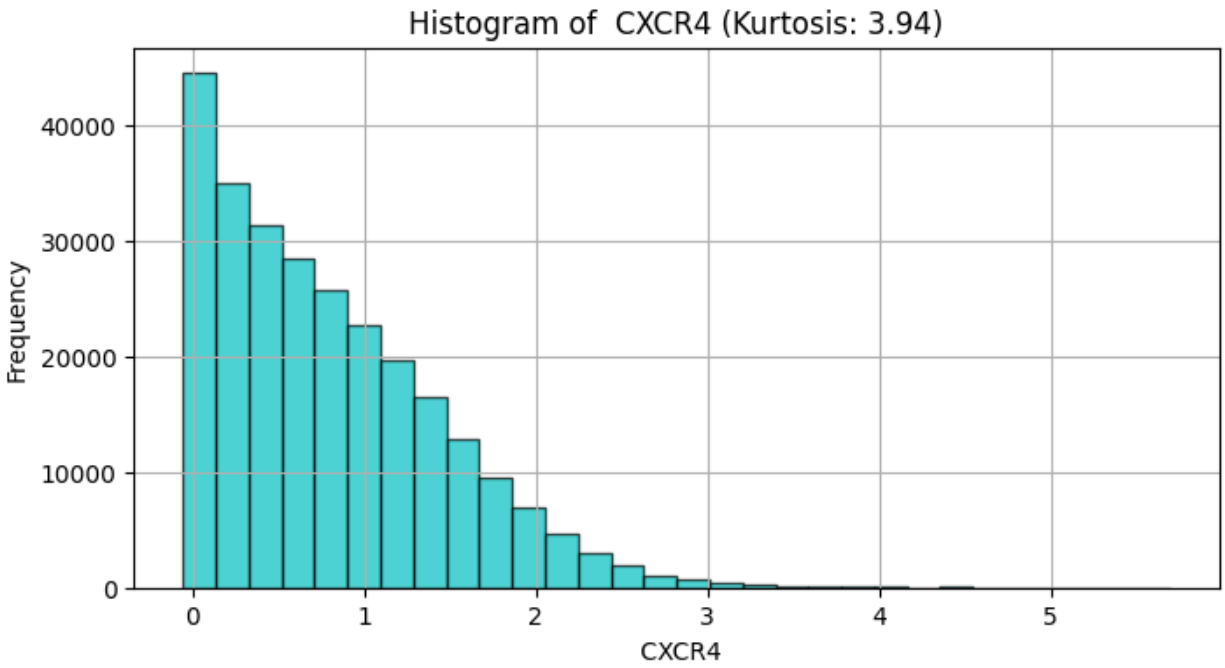

Histogram of CD4 (Kurtosis: 5.84)
Frequency
CD4
0
20000
40000
60000
80000
100000
120000
140000
0
1
2
3
4
5
6

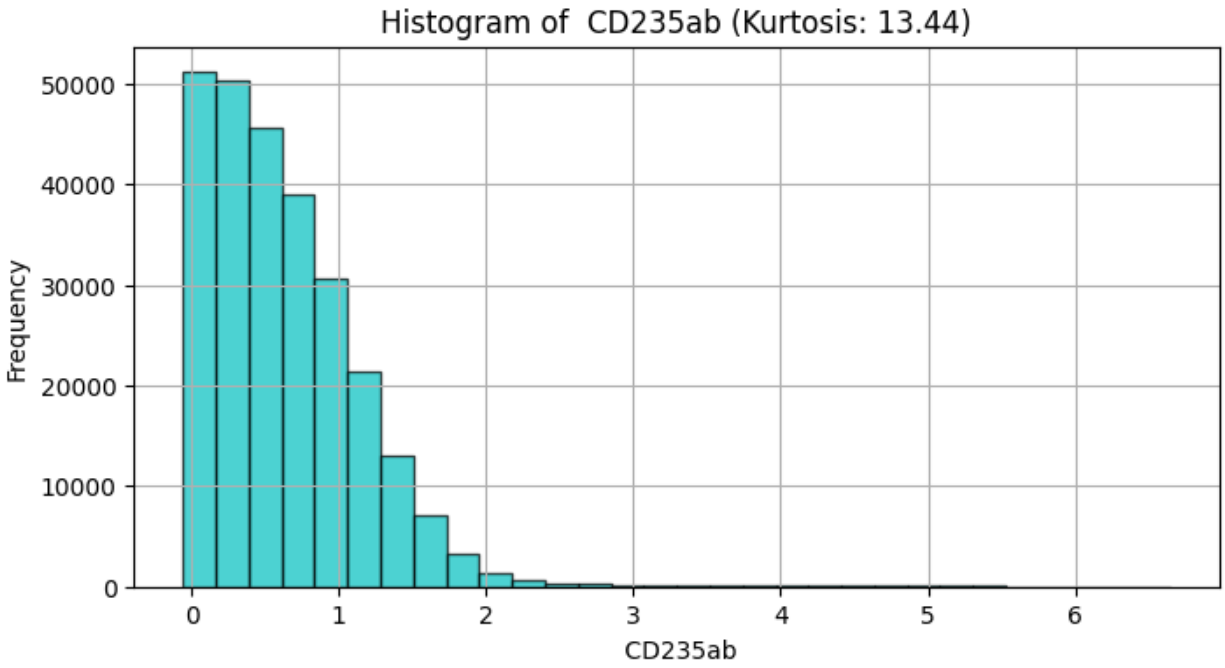

Histogram of CD8 (Kurtosis: 4.75)
Frequency
CD8
0
20000
40000
60000
80000
100000
120000
140000
0
1
2
3
4

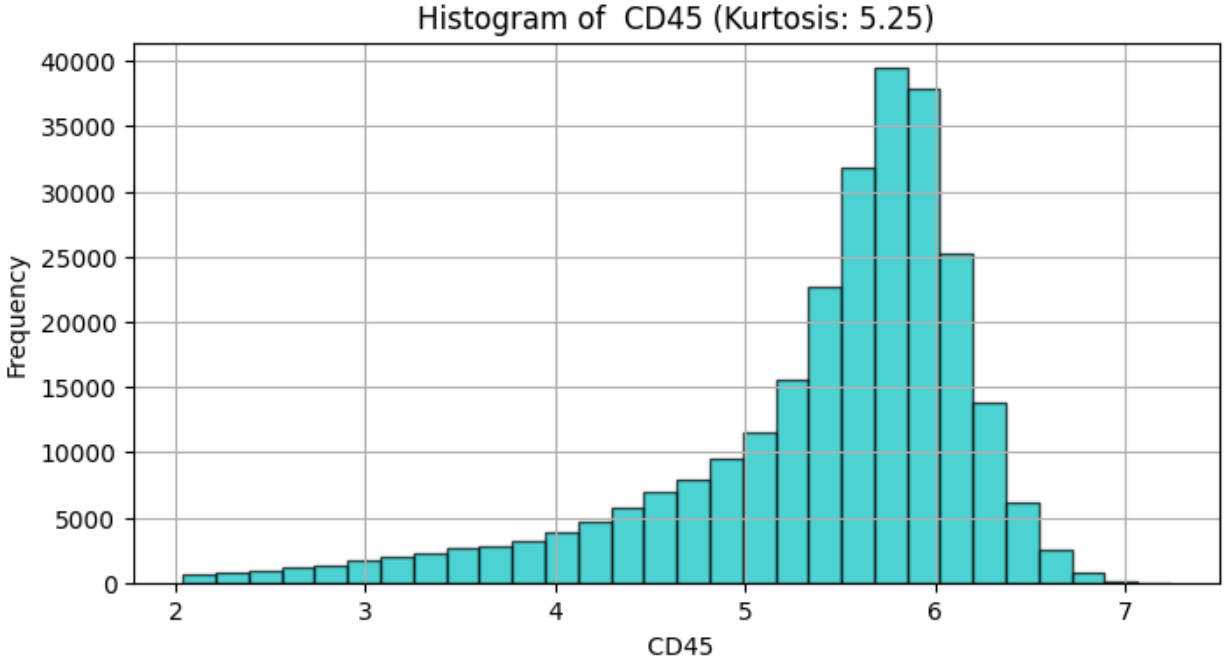

Histogram of CD34 (Kurtosis: 16.60)
Frequency
CD34
0
20000
40000
60000
80000
100000
120000
140000
0
1
2
3
4
5

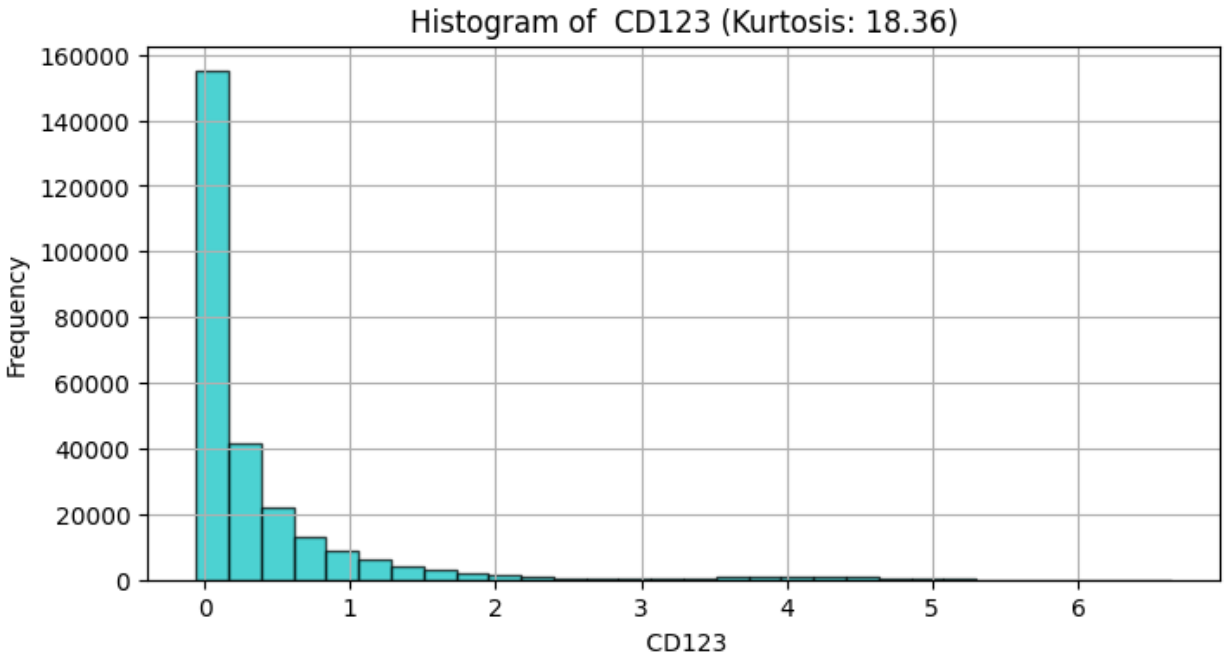

Histogram of Flt3 (Kurtosis: 85.58)
Frequency
Flt3
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6
7

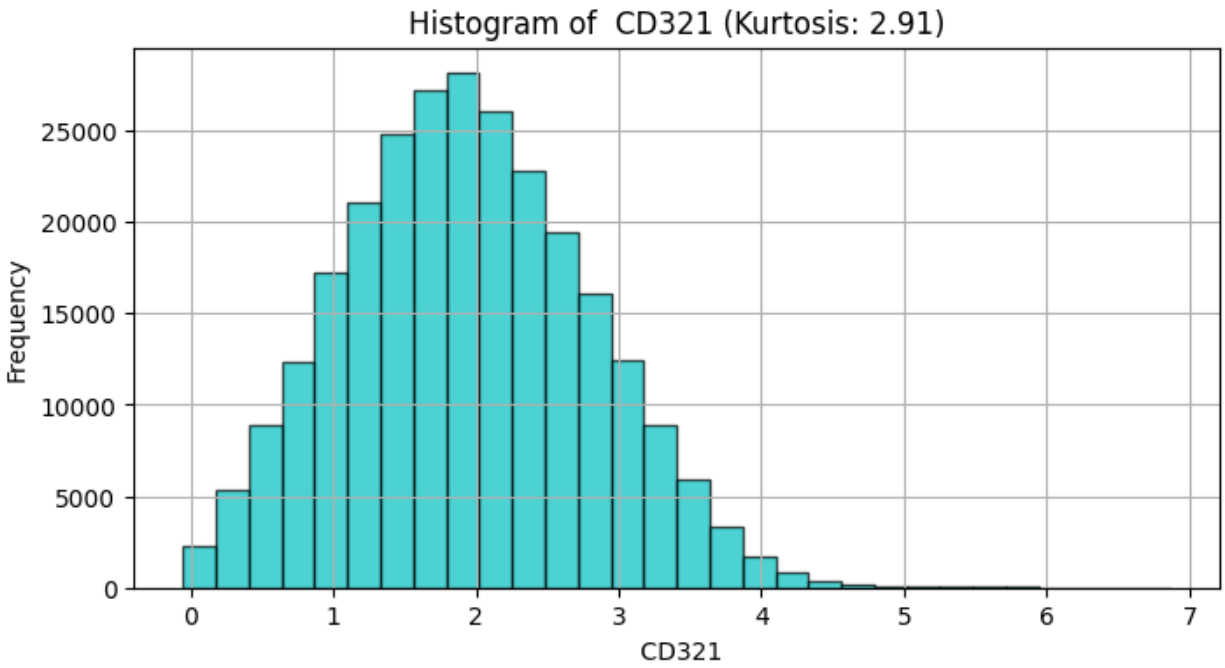

Histogram of CD20 (Kurtosis: 10.44)
Frequency
CD20
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6

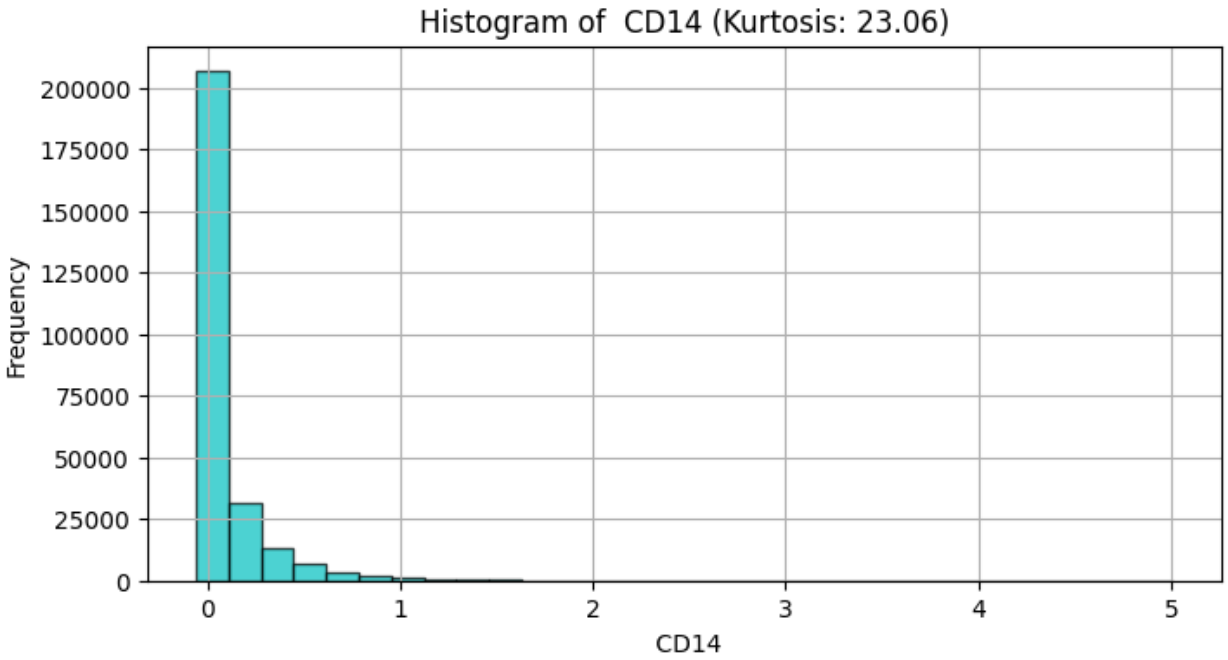

Histogram of CXCR4 (Kurtosis: 3.94)
Frequency
CXCR4
0
10000
20000
30000
40000
0
1
2
3
4
5

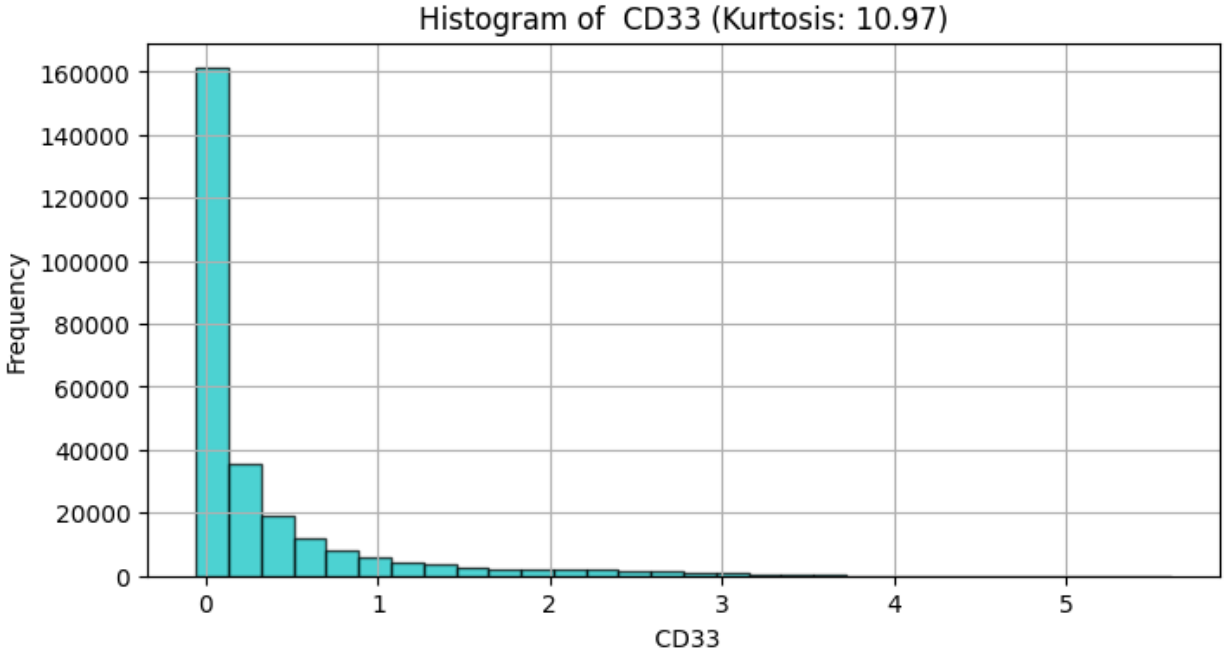

Histogram of CD235ab (Kurtosis: 13.44)
Frequency
CD235ab
0
10000
20000
30000
40000
50000
0
1
2
3
4
5
6

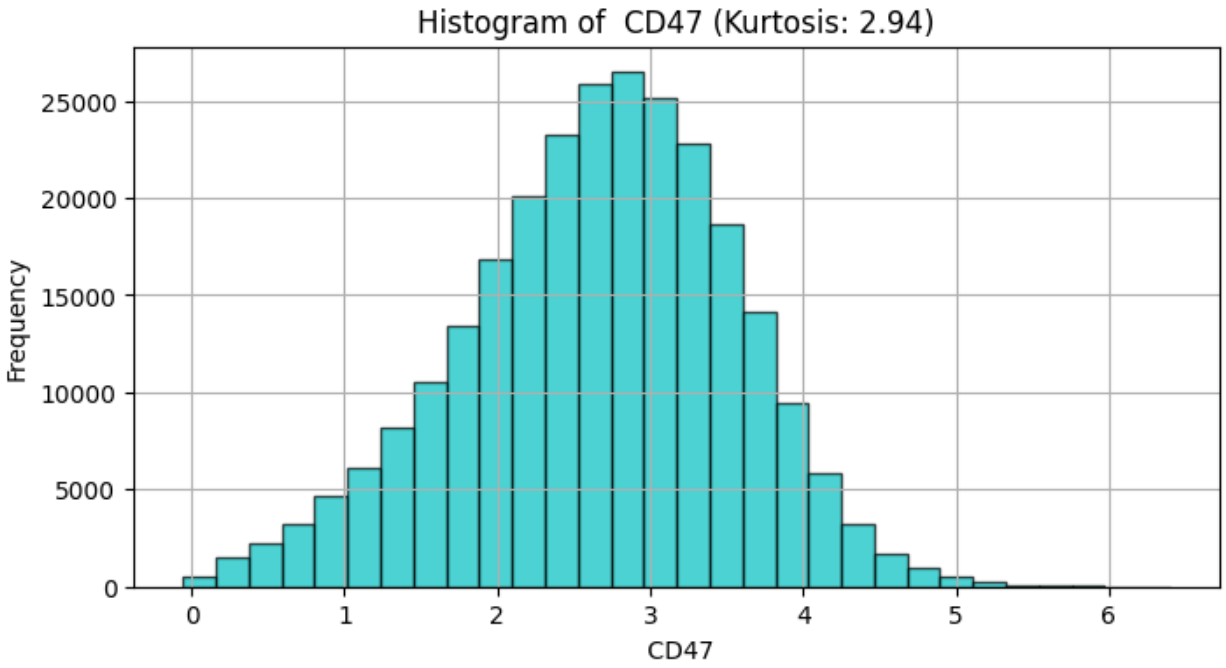

Histogram of CD45 (Kurtosis: 5.25)
Frequency
CD45
0
5000
10000
15000
20000
25000
30000
35000
40000
2
3
4
5
6
7

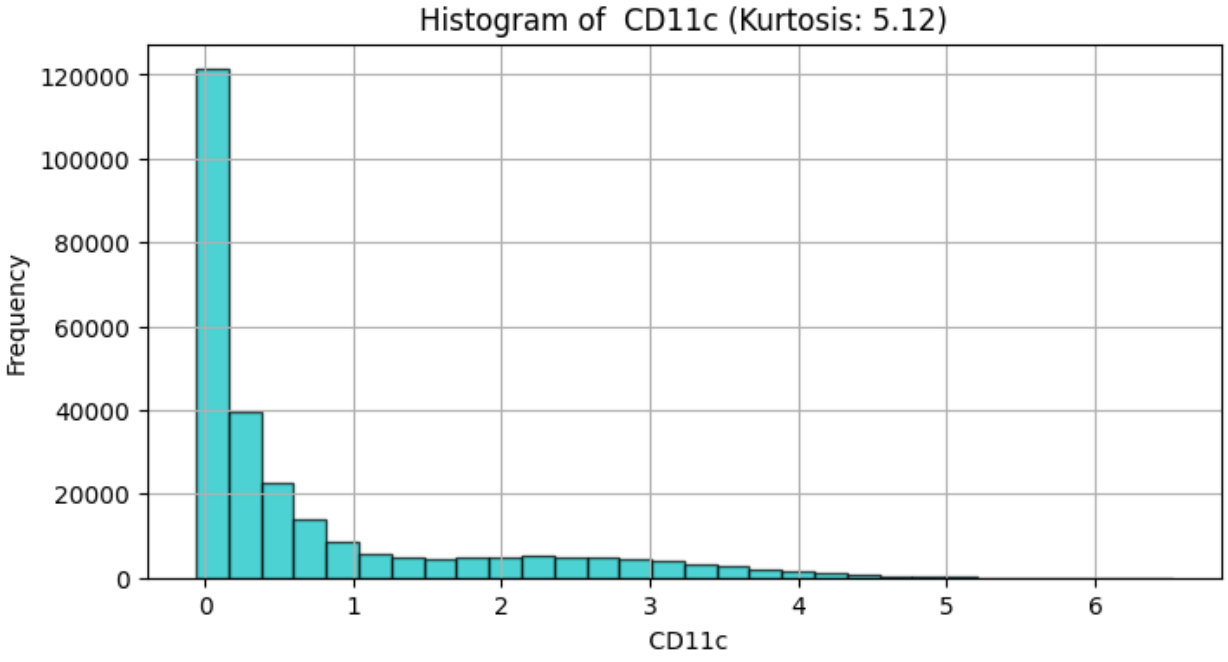

Histogram of CD123 (Kurtosis: 18.36)
Frequency
CD123
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6

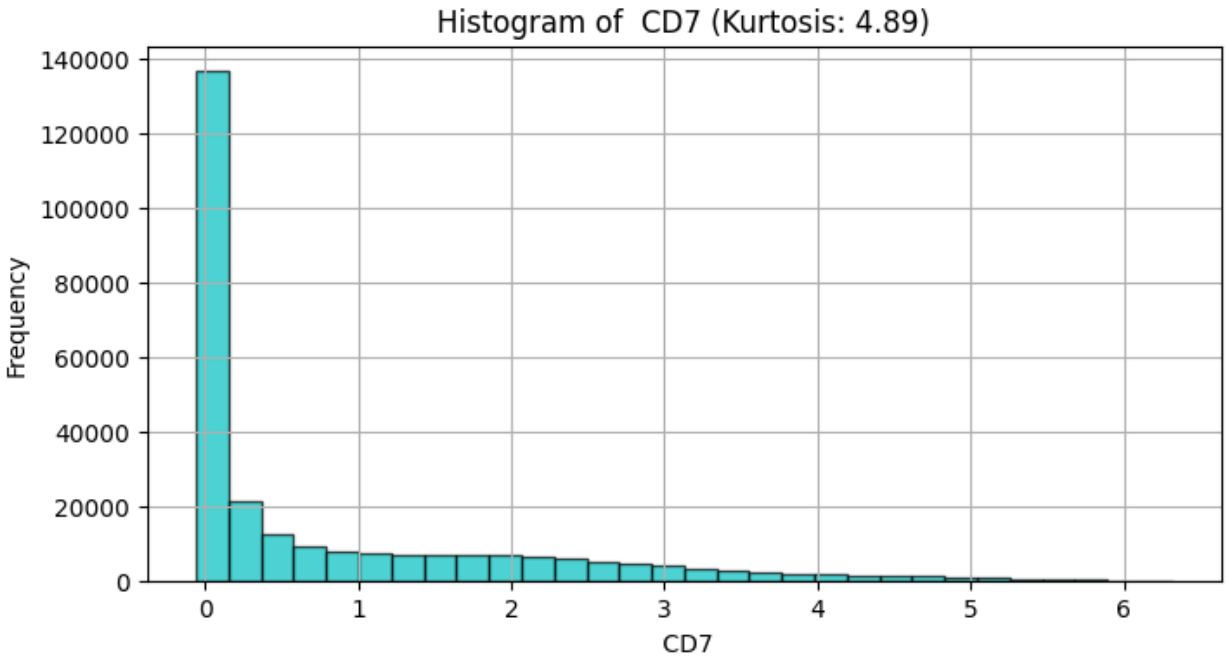

Histogram of CD321 (Kurtosis: 2.91)
Frequency
CD321
0
5000
10000
15000
20000
25000
0
1
2
3
4
5
6
7

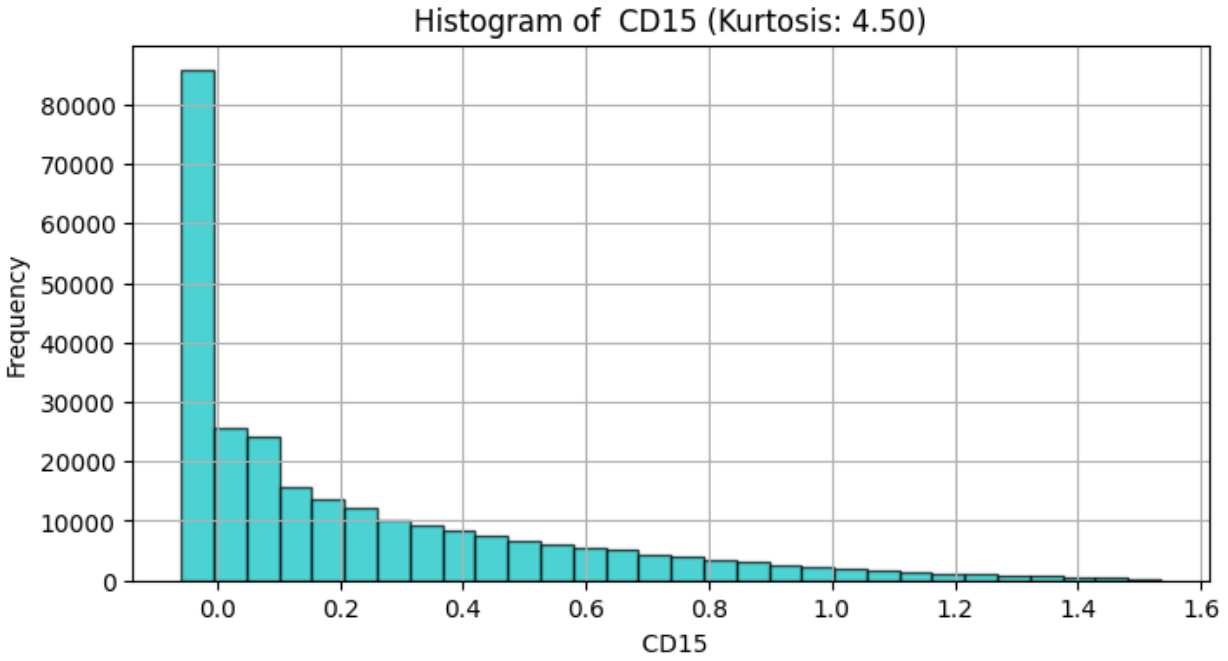

Histogram of CD14 (Kurtosis: 23.06)
Frequency
CD14
0
25000
50000
75000
100000
125000
150000
175000
200000
0
1
2
3
4
5

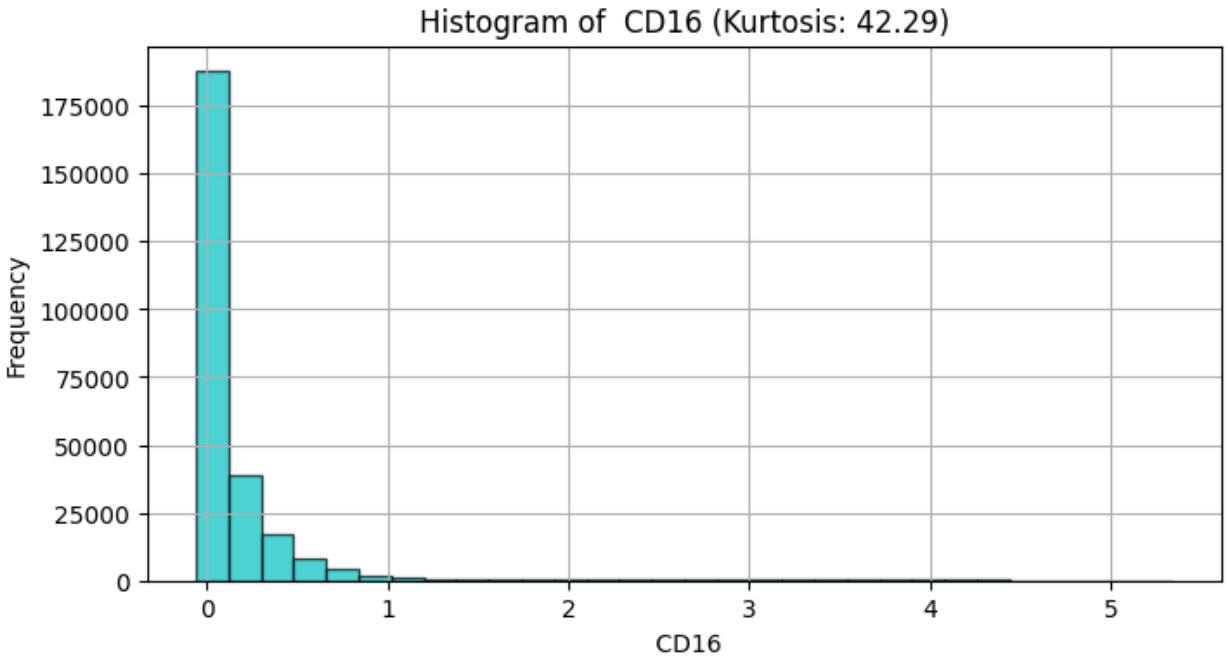

Histogram of CD33 (Kurtosis: 10.97)
Frequency
CD33
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5

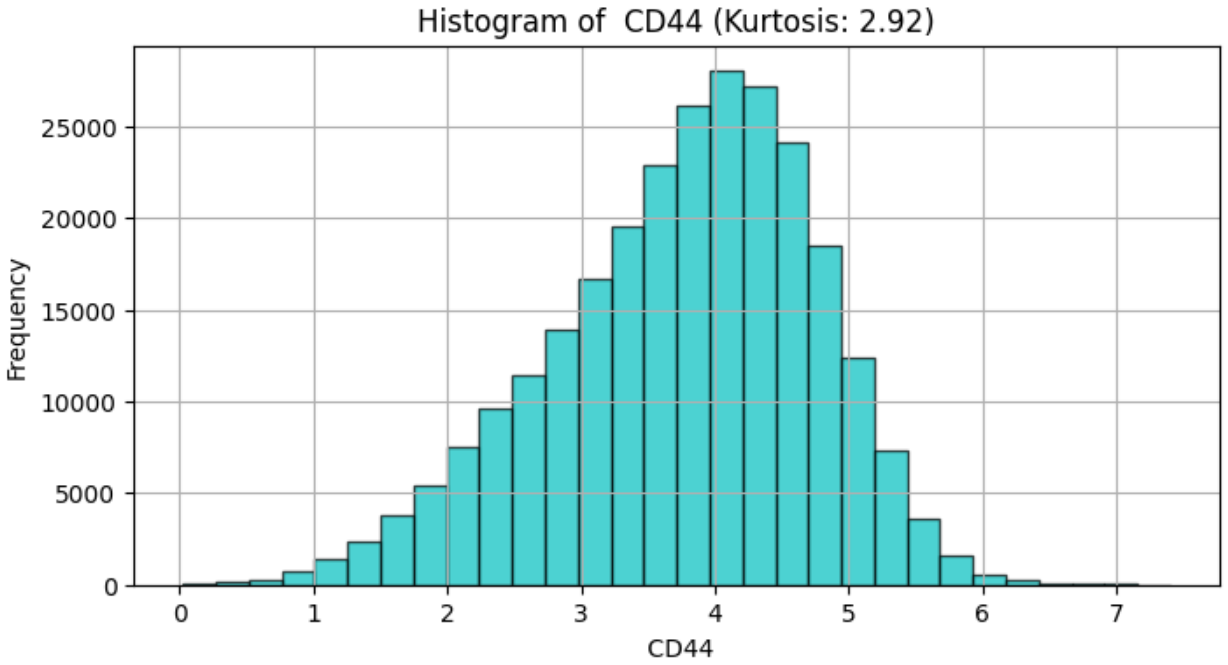

Histogram of CD47 (Kurtosis: 2.94)
Frequency
CD47
0
5000
10000
15000
20000
25000
0
1
2
3
4
5
6

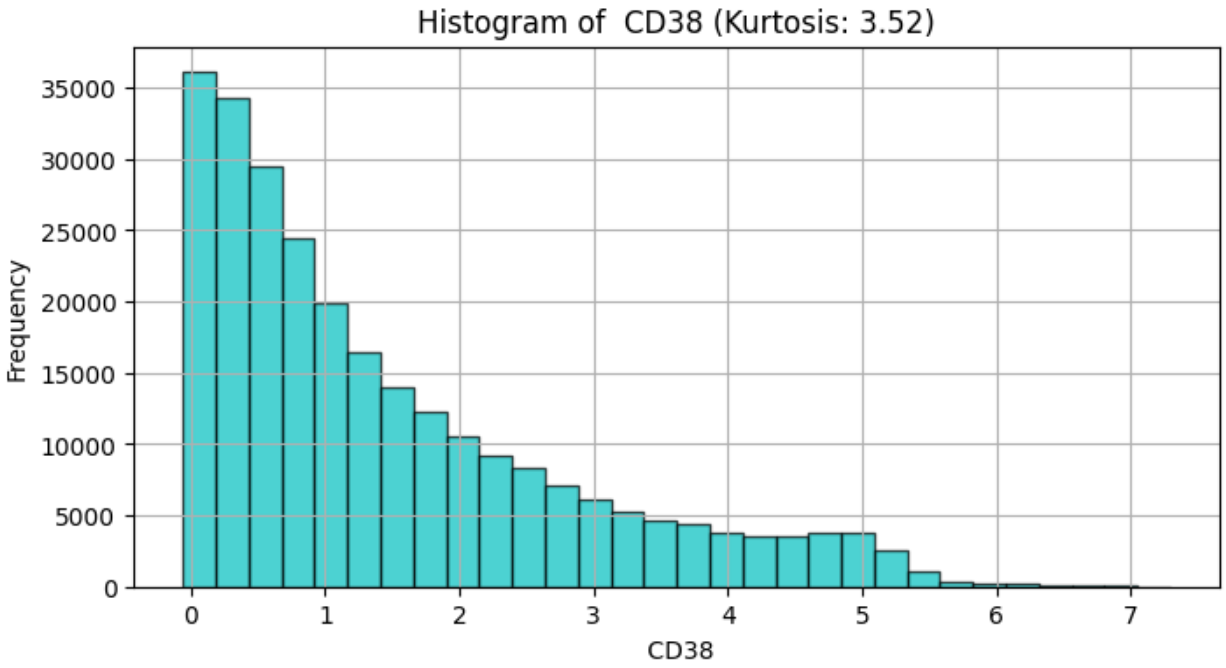

Histogram of CD11c (Kurtosis: 5.12)
Frequency
CD11c
0
20000
40000
60000
80000
100000
120000
0
1
2
3
4
5
6

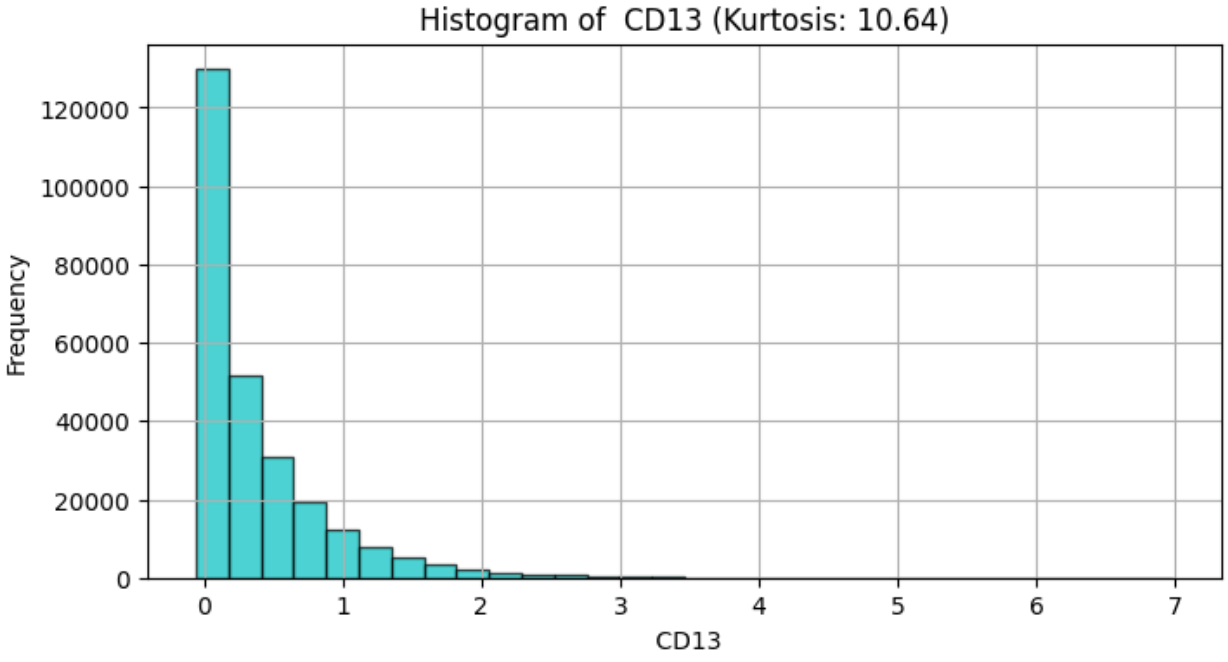

Histogram of CD7 (Kurtosis: 4.89)
Frequency
CD7
0
20000
40000
60000
80000
100000
120000
140000
0
1
2
3
4
5
6

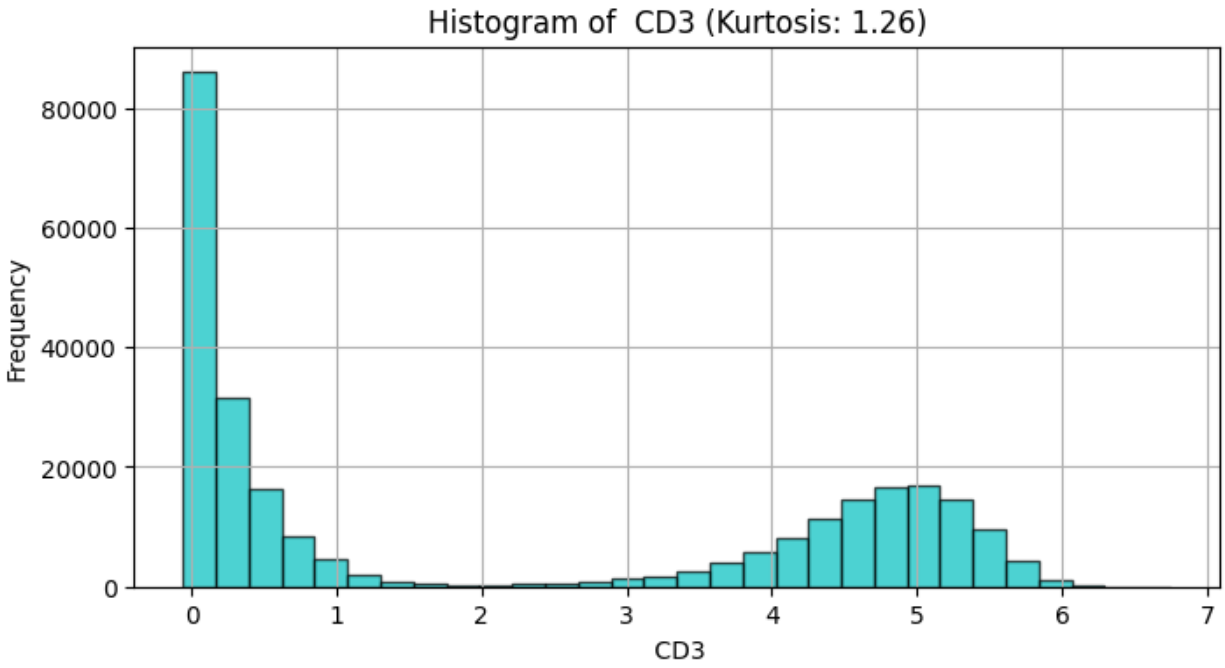

Histogram of CD15 (Kurtosis: 4.50)
Frequency
CD15
0
10000
20000
30000
40000
50000
60000
70000
80000
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

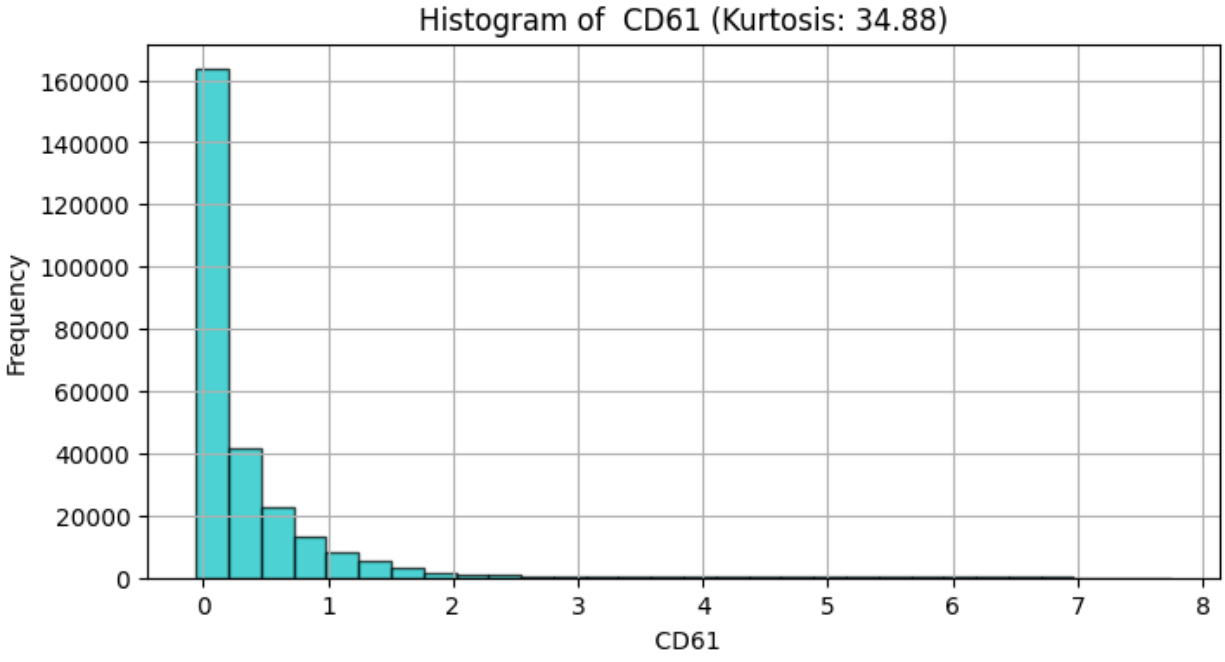

Histogram of CD16 (Kurtosis: 42.29)
Frequency
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5
CD16

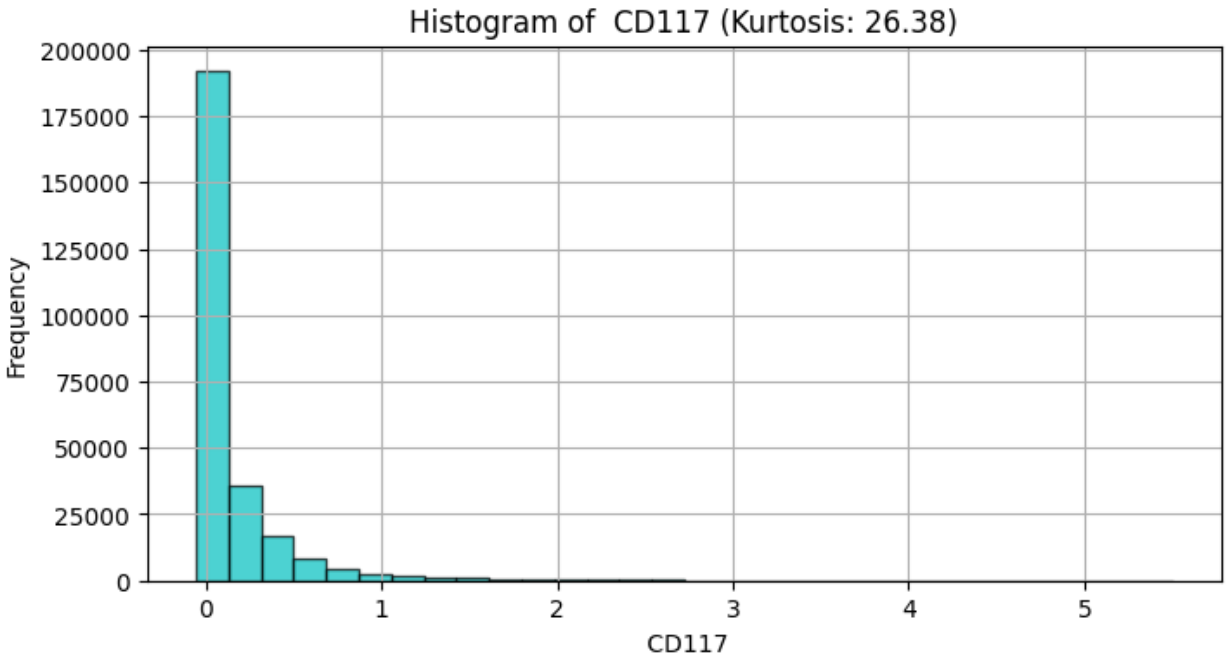

Histogram of CD44 (Kurtosis: 2.92)
Frequency
25000
20000
15000
10000
5000
0
0
1
2
3
4
5
6
7
CD44

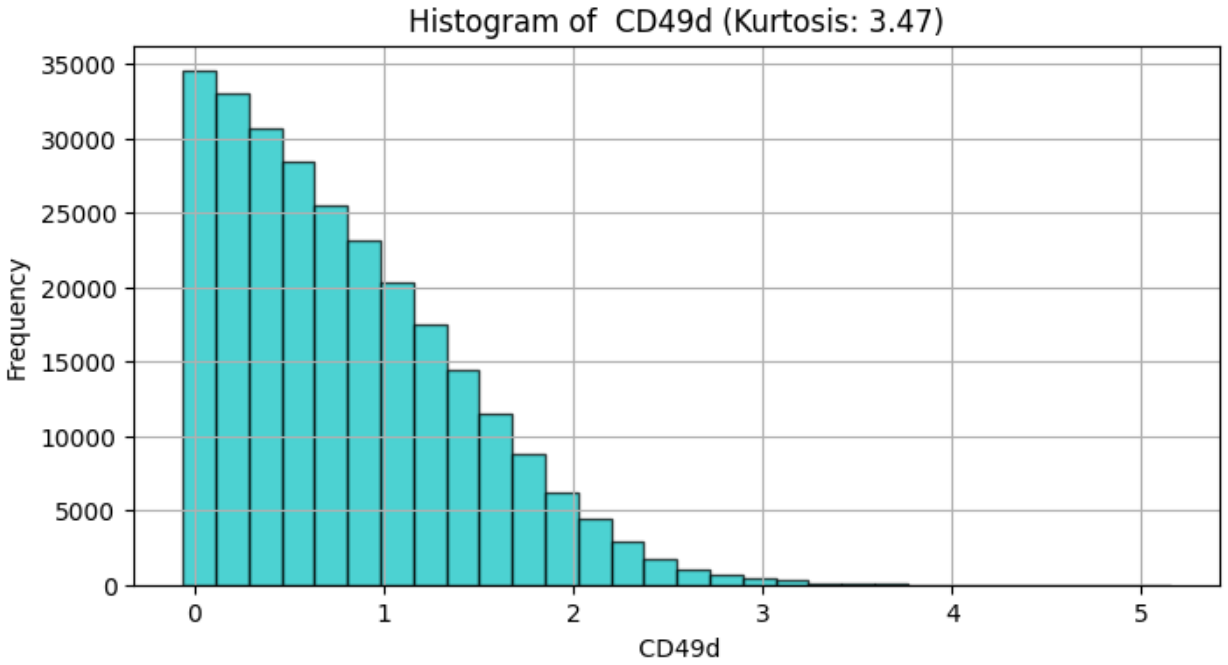

Histogram of CD38 (Kurtosis: 3.52)
Frequency
CD38
0
5000
10000
15000
20000
25000
30000
35000
0
1
2
3
4
5
6
7

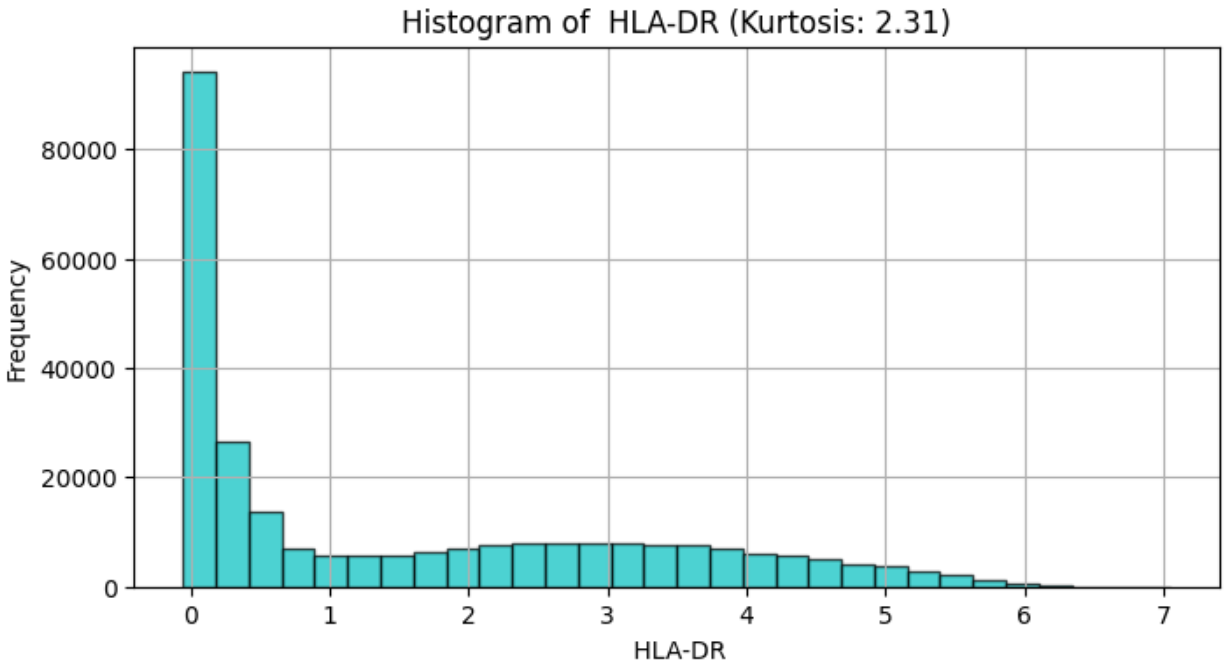

Histogram of CD13 (Kurtosis: 10.64)
Frequency
CD13
0
20000
40000
60000
80000
100000
120000
0
1
2
3
4
5
6
7

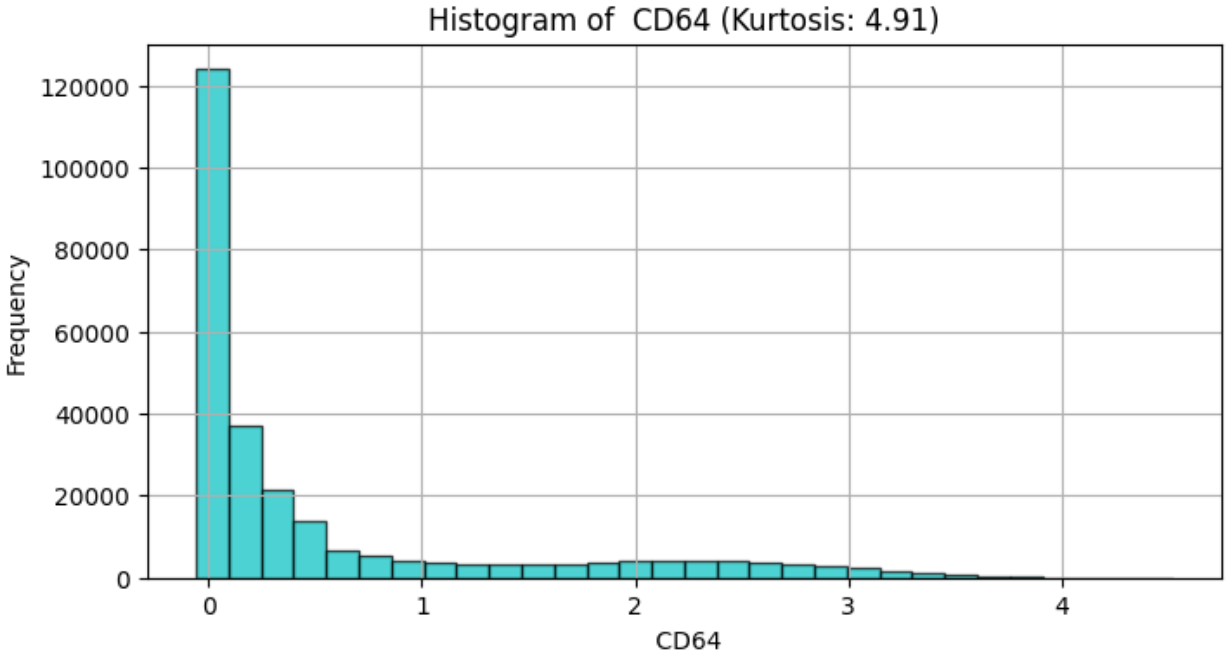

Histogram of CD3 (Kurtosis: 1.26)
Frequency
CD3
0
20000
40000
60000
80000
0
1
2
3
4
5
6
7

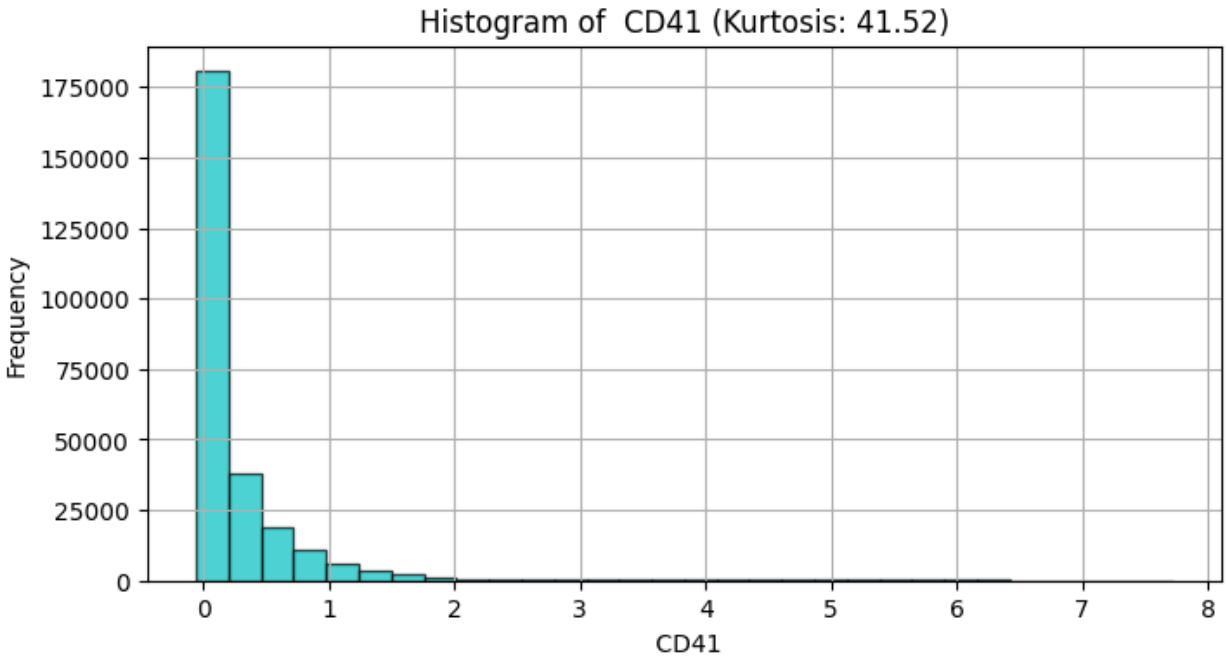

Histogram of CD61 (Kurtosis: 34.88)
Frequency
CD61
0
20000
40000
60000
80000
100000
120000
140000
160000
0
1
2
3
4
5
6
7
8

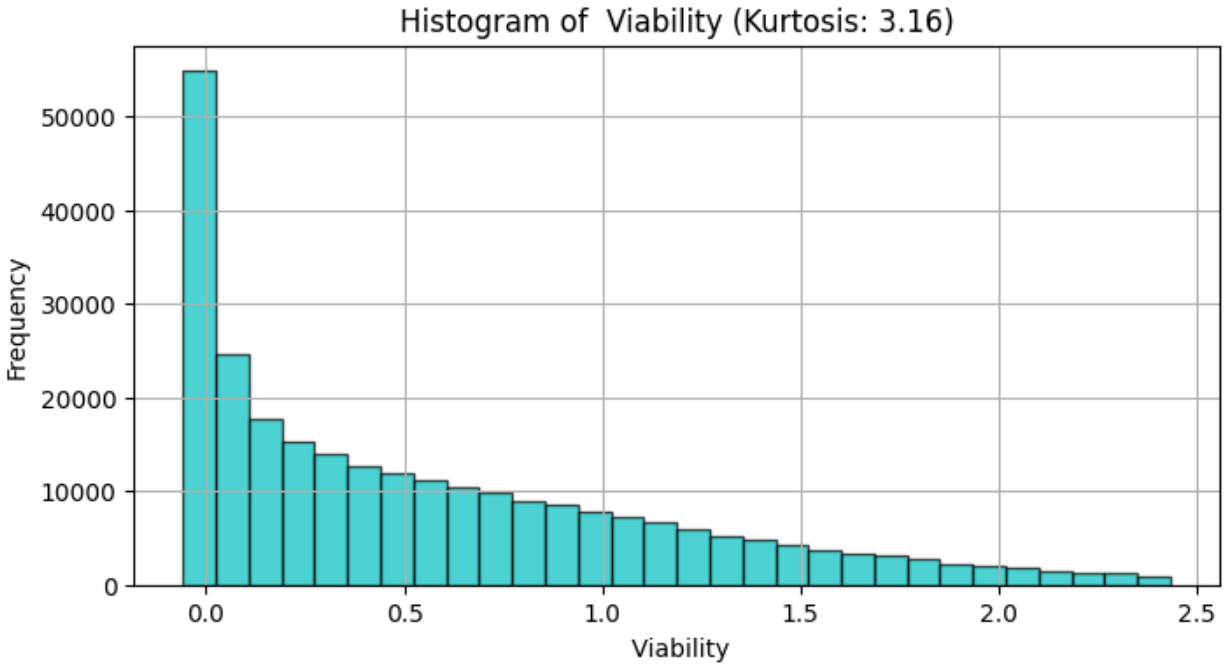

Histogram of CD117 (Kurtosis: 26.38)
Frequency
CD117
200000
175000
150000
125000
100000
75000
50000
25000
0
0
1
2
3
4
5

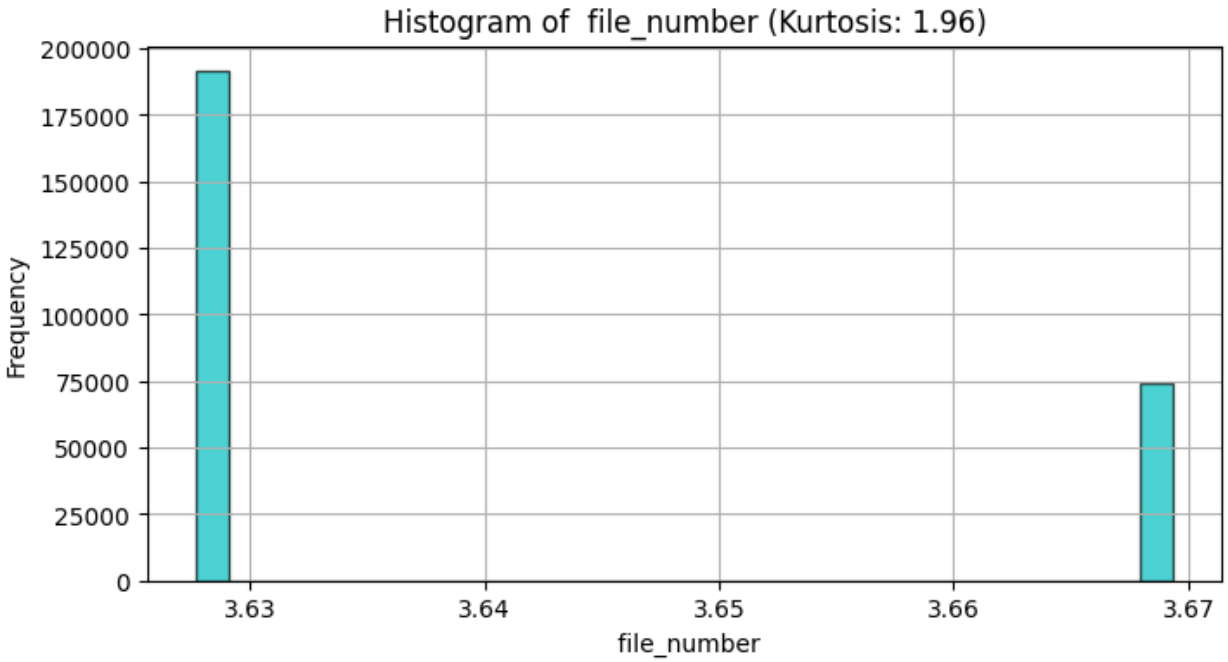

Histogram of CD49d (Kurtosis: 3.47)
Frequency
CD49d
35000
30000
25000
20000
15000
10000
5000
0
0
1
2
3
4
5

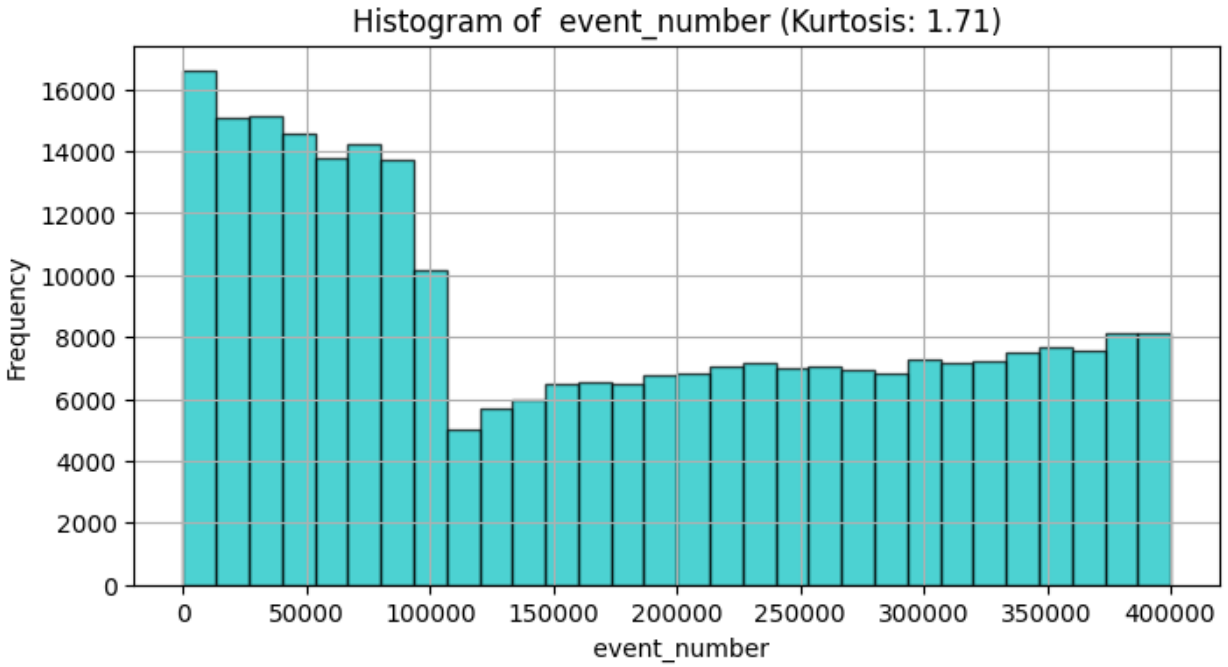

Histogram of HLA-DR (Kurtosis: 2.31)
Frequency
HLA-DR
0
20000
40000
60000
80000
0
1
2
3
4
5
6
7

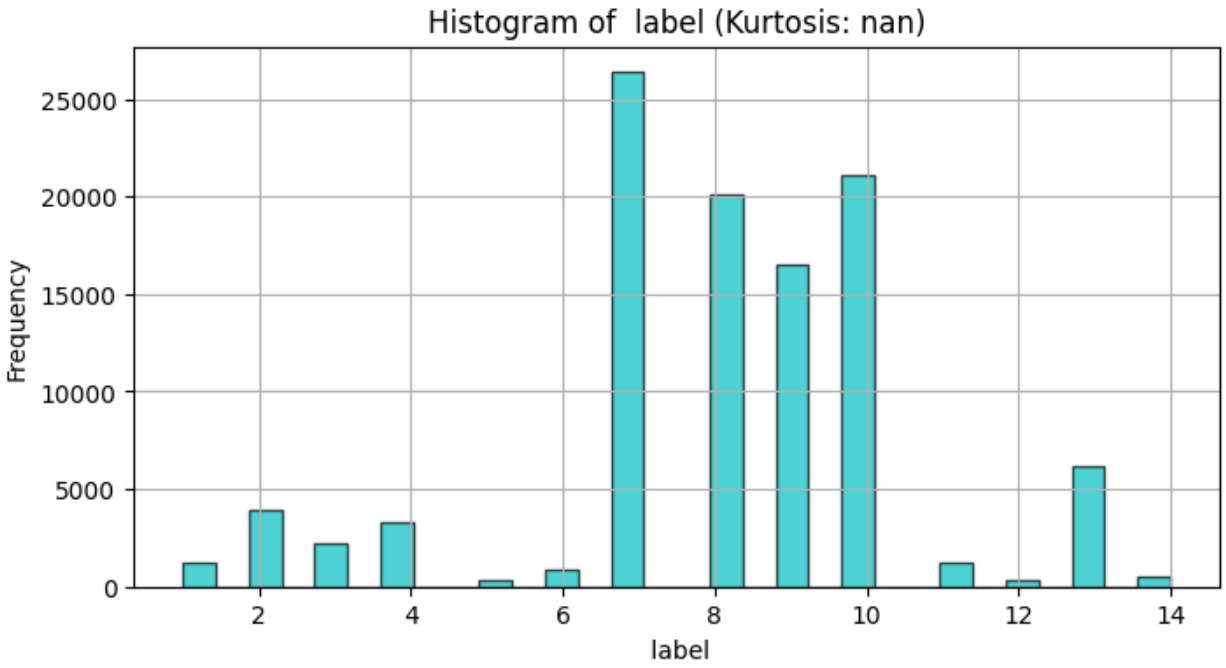

Histogram of CD64 (Kurtosis: 4.91)
Frequency
CD64
0
20000
40000
60000
80000
100000
120000
0
1
2
3
4

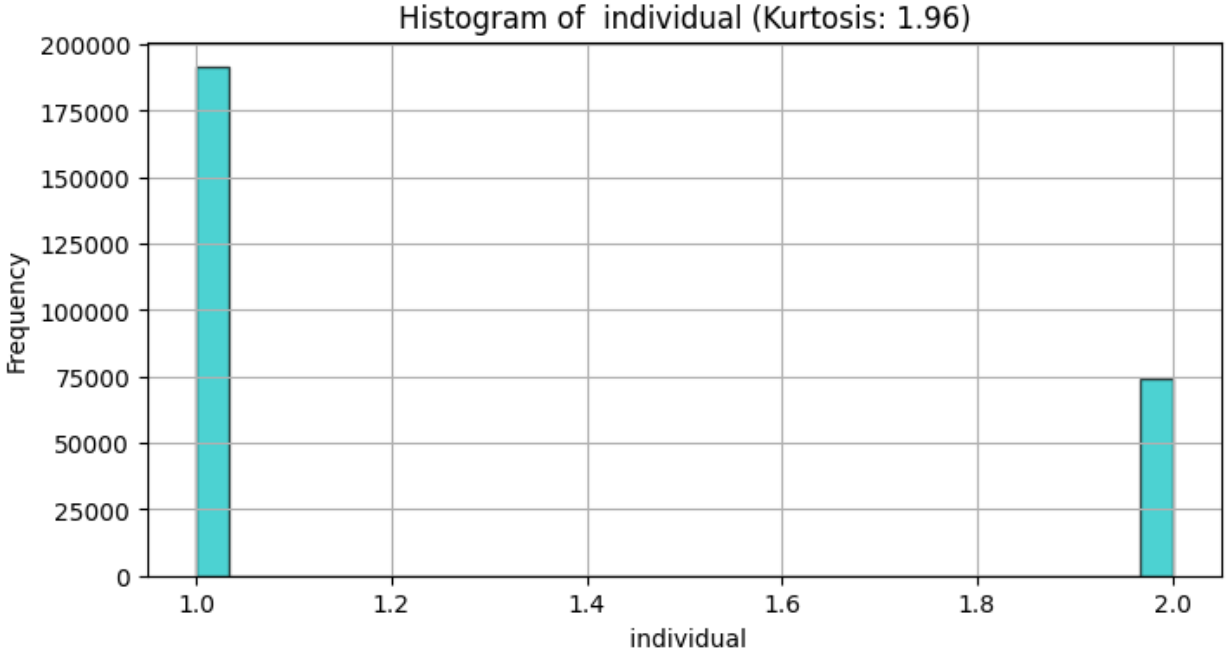

Histogram of CD41 (Kurtosis: 41.52)
Frequency
CD41
0
25000
50000
75000
100000
125000
150000
175000
0
1
2
3
4
5
6
7
8

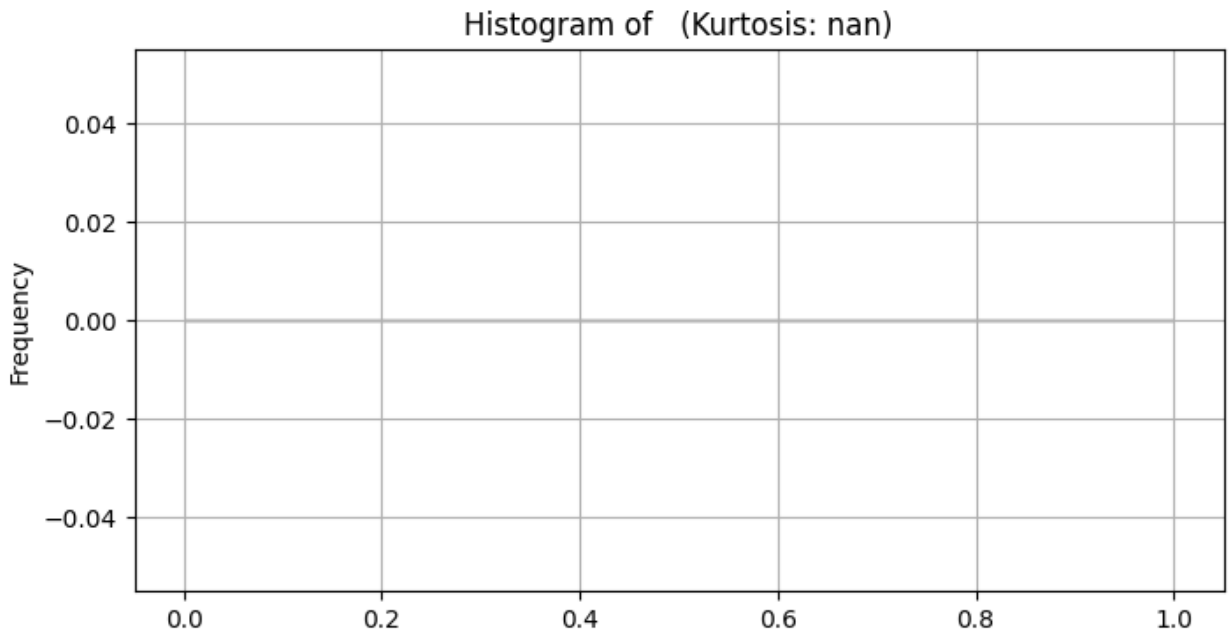

Histogram of Viability (Kurtosis: 3.16)
Frequency
Viability
0
10000
20000
30000
40000
50000
0.0
0.5
1.0
1.5
2.0
2.5


Histogram of file_number (Kurtosis: 1.96)
Frequency
file_number
200000
175000
150000
125000
100000
75000
50000
25000
0
3.63
3.64
3.65
3.66
3.67


Histogram of event_number (Kurtosis: 1.71)
Frequency
event_number
16000
14000
12000
10000
8000
6000
4000
2000
0
0
50000
100000
150000
200000
250000
300000
350000
400000


Histogram of label (Kurtosis: nan)
Frequency
label
0
5000
10000
15000
20000
25000
2
4
6
8
10
12
14


Histogram of individual (Kurtosis: 1.96)
Frequency
individual
0
25000
50000
75000
100000
125000
150000
175000
200000
1.0
1.2
1.4
1.6
1.8
2.0



Loading the MNIST Dataset and computing the t-sne

```
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,n_iter=1000, verbose=1)
tsne_results = tsne.fit_transform(train_images_flat)
```

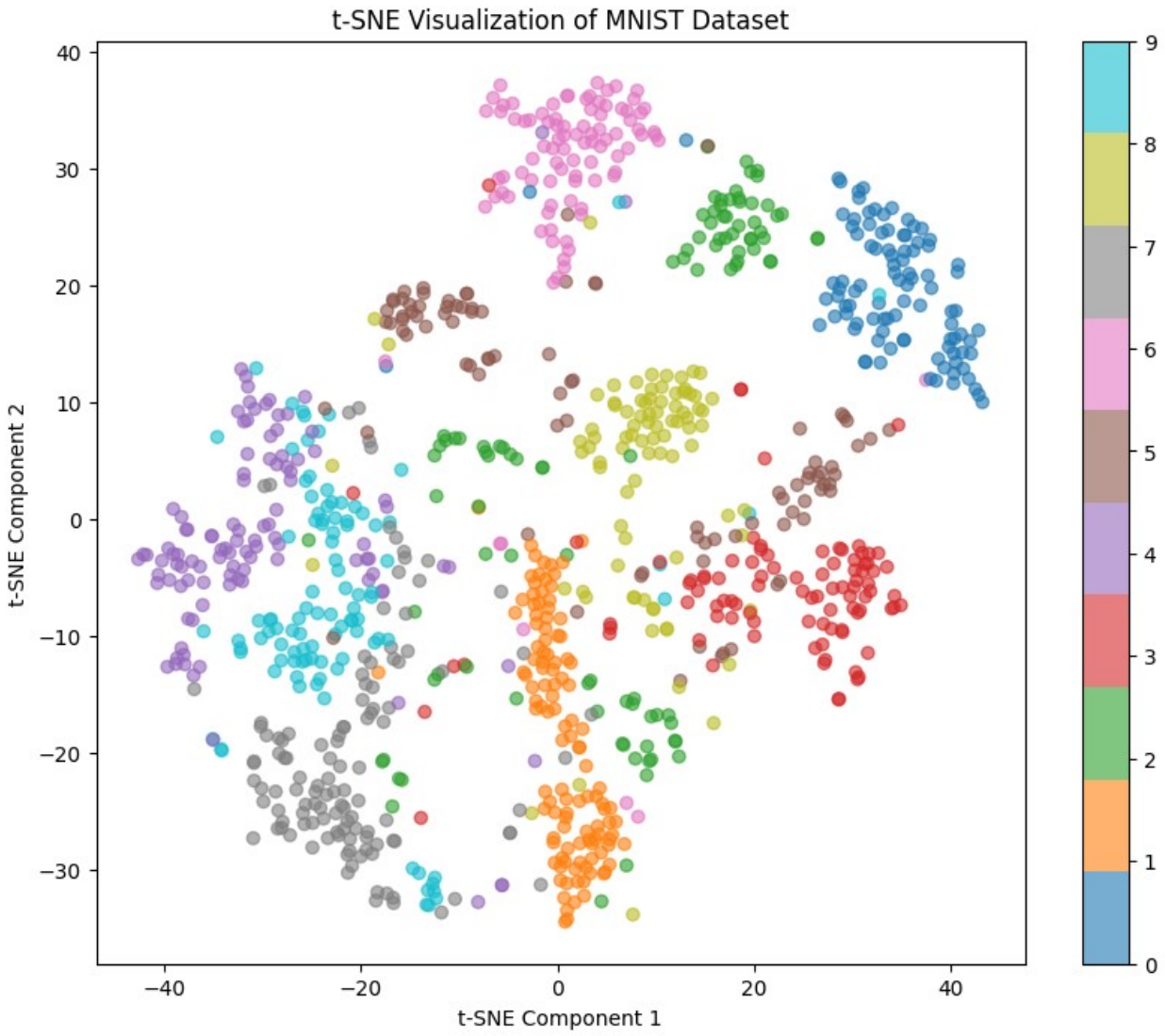
```
/usr/local/lib/python3.10/dist-packages/sklearn/manifold/
_t_sne.py:1162: FutureWarning: 'n_iter' was renamed to 'max_iter' in
version 1.5 and will be removed in 1.7.
  warnings.warn(

[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 1000 samples in 0.002s...
[t-SNE] Computed neighbors for 1000 samples in 0.086s...
[t-SNE] Computed conditional probabilities for sample 1000 / 1000
[t-SNE] Mean sigma: 2.756635
[t-SNE] KL divergence after 250 iterations with early exaggeration:
69.076538
[t-SNE] KL divergence after 1000 iterations: 0.892835
```

Ploting the t-SNE for MNIST Dataset

```
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]

# Initialize and fit t-SNE  (This was missing in the original code)
tsne = TSNE(n_components=2, random_state=42, perplexity=30,
n_iter=1000, verbose=1)
#train_images_tsne = tsne.fit_transform(train_images_flat)
train_images_tsne = tsne.fit_transform(train_images_flat)
exclude_columns = ['Event', 'Time', 'Cell_length', 'file_number',
'event_number', 'label', 'individual']
# 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

/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 0.238s...
[t-SNE] Computed conditional probabilities for sample 1000 / 1000
[t-SNE] Mean sigma: 2.756635
[t-SNE] KL divergence after 250 iterations with early exaggeration: 69.076538
[t-SNE] KL divergence after 1000 iterations: 0.892835
```



Dimensionality reduction

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import numpy as np # Import numpy

# Initialize the MinMaxScaler
scaler = MinMaxScaler()

# Replace empty strings or spaces with NaN
df = df.replace(r'^\s*$', np.nan, regex=True)  # Use np.nan directly

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

# Standardize all columns in the dataset (using the imputed data)
```

```
df[:] = scaler.fit_transform(df)

# Check the standardized data
print(df.head())
```

```
<ipython-input-15-ddf1a2b997e5>:9: FutureWarning: Downcasting behavior
in `replace` is deprecated and will be removed in a future version. To
retain the old behavior, explicitly call
`result.infer_objects(copy=False)`. To opt-in to the future behavior,
set `pd.set_option('future.no_silent_downcasting', True)`
  df = df.replace(r'^\s*$', np.nan, regex=True)  # Use np.nan directly
/usr/local/lib/python3.10/dist-packages/sklearn/utils/_array_api.py:69
5: RuntimeWarning: All-NaN slice encountered
  return xp.asarray(numpy.nanmin(X, axis=axis))
/usr/local/lib/python3.10/dist-packages/sklearn/utils/_array_api.py:71
2: RuntimeWarning: All-NaN slice encountered
  return xp.asarray(numpy.nanmax(X, axis=axis))
<ipython-input-15-ddf1a2b997e5>:16: FutureWarning: Setting an item of
incompatible dtype is deprecated and will raise in a future error of
pandas. Value '[0.21818182 0.45454545 0.4        ... 0.63636364
0.56363636 0.52727273]' has dtype incompatible with int64, please
explicitly cast to a compatible dtype first.
  df[:] = scaler.fit_transform(df)
<ipython-input-15-ddf1a2b997e5>:16: FutureWarning: Setting an item of
incompatible dtype is deprecated and will raise in a future error of
pandas. Value '[0.00076479 0.00135962 0.0043113  ... 0.25665128
0.25667877 0.25669127]' has dtype incompatible with int64, please
explicitly cast to a compatible dtype first.
  df[:] = scaler.fit_transform(df)
<ipython-input-15-ddf1a2b997e5>:16: FutureWarning: Setting an item of
incompatible dtype is deprecated and will raise in a future error of
pandas. Value '[1.         0.         0.22222222 ... 0.44444444
0.66666667 0.55555556]' has dtype incompatible with int64, please
explicitly cast to a compatible dtype first.
  df[:] = scaler.fit_transform(df)
```

```
       Time   Cell_length      DNA1      DNA2    CD45RA     CD133
CD19  \
0  0.003799      0.218182  0.380681  0.454713  0.032599  0.005102
0.010180
1  0.005271      0.454545  0.368682  0.492802  0.112418  0.003545
0.008208
2  0.009899      0.400000  0.249642  0.410614  0.097929  0.004631
0.026137
3  0.010017      0.345455  0.348592  0.495353  0.072765  0.005455
0.008008
4  0.010865      0.272727  0.282425  0.433546  0.007186  0.004974
0.027438

       CD22     CD11b       CD4  ...    HLA-DR      CD64      CD41  \
```

```
0  0.023713  0.009222  0.063461  ...  0.242242  0.011544  0.007238
1  0.025250  0.162862  0.003363  ...  0.077290  0.044364  0.119109
2  0.002753  0.010595  0.007376  ...  0.192154  0.010323  0.006151
3  0.002543  0.148887  0.005827  ...  0.027916  0.009779  0.004141
4  0.106009  0.219187  0.091949  ...  0.035906  0.029363  0.002283

    Viability   file_number   event_number   label   individual
label
0    0.283583           0.0       0.000765     0.0            0 NaN
1.000000
1    0.248639           0.0       0.001360     0.0            0 NaN
0.000000
2    0.281539           0.0       0.004311     0.0            0 NaN
0.222222
3    0.012628           0.0       0.004411     0.0            0 NaN
0.222222
4    0.136999           0.0       0.005074     0.0            0 NaN
0.888889

[5 rows x 43 columns]
```

2D plot for PCA

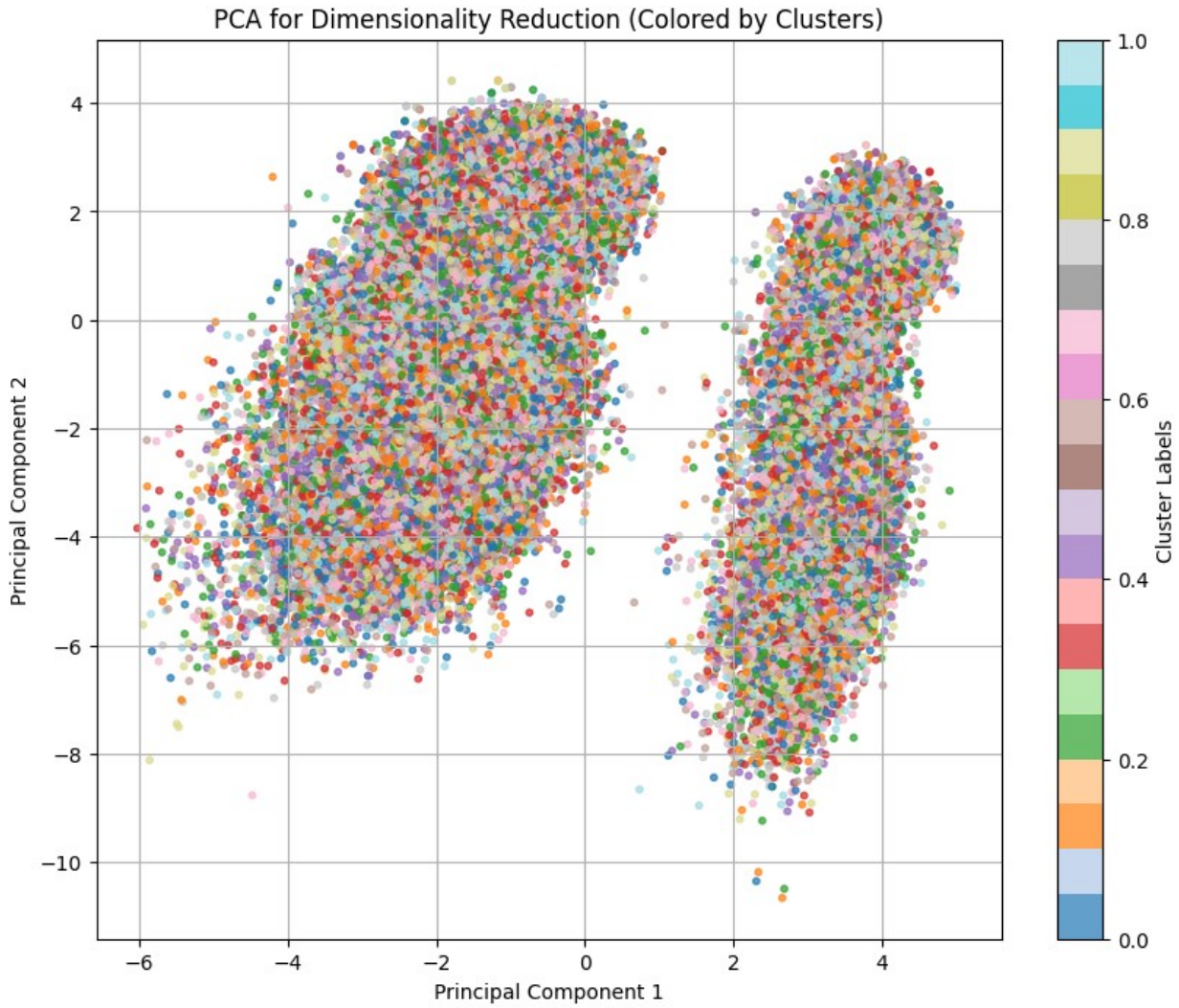
```
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import numpy as np
print(df.columns)

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(['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', ' ',
'label'],
      dtype='object')
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/impute/_base.py:598:
UserWarning: Skipping features without any observed values: [' ']. At
least one non-missing value is needed for imputation with
strategy='mean'.
  warnings.warn(
```

	Standard deviation	Proportion of Variance	Cumulative Proportion
PC1	2.401951	0.144234	0.144234
PC2	2.346199	0.137616	0.281849
PC3	1.961911	0.096227	0.378076
PC4	1.643769	0.067549	0.445626



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
```

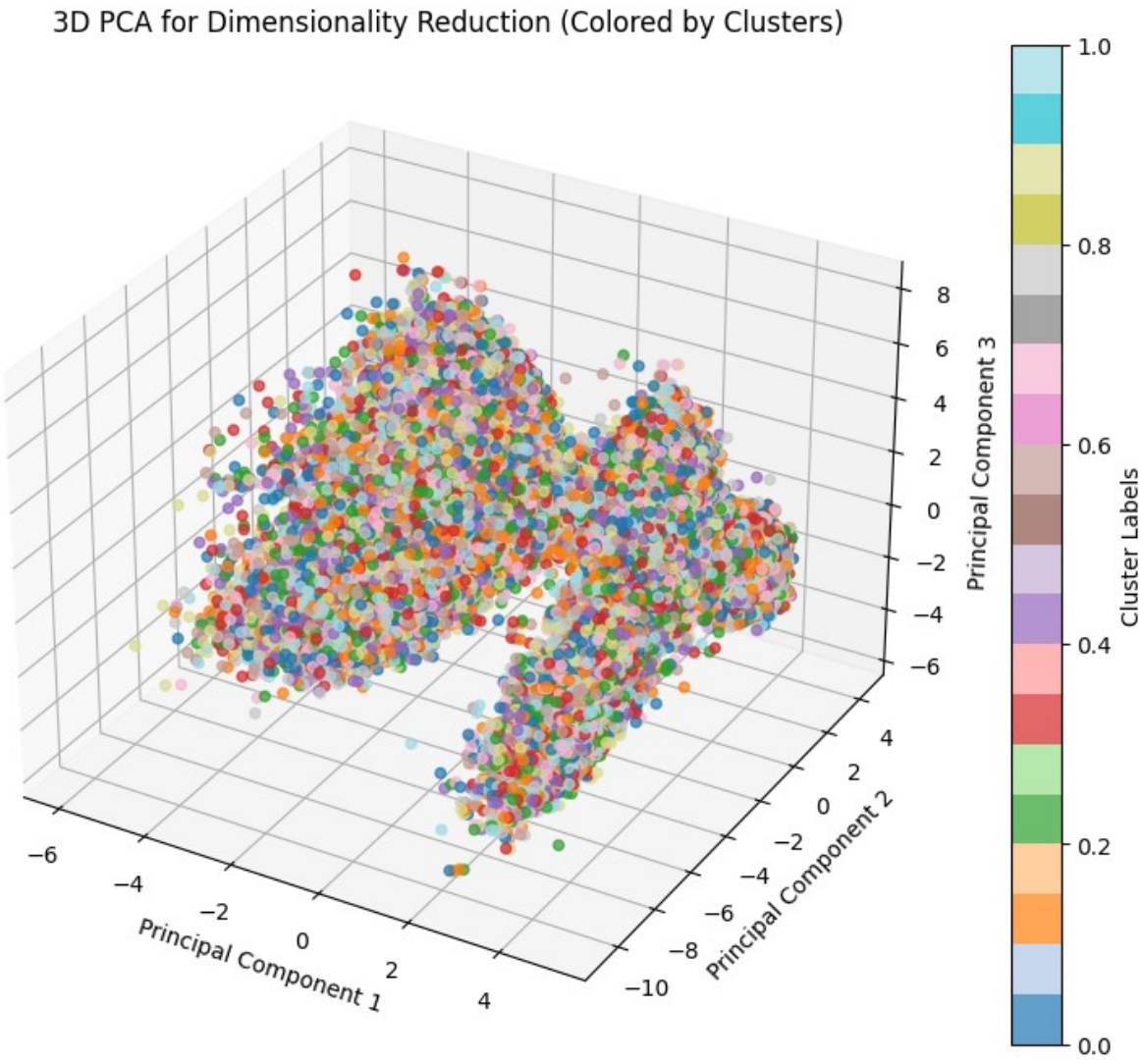
```
columns_to_drop = ['Event', 'Time', 'Cell_length', 'file_number',
'event_number', 'label', 'individual']
# Replace with actual column names if different, e.g.,
# columns_to_drop = ['event', 'time', 'cell_length', 'file_number',
'event_number', 'label', 'individual']

# 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: ['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', ' ', 'label']
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/impute/_base.py:598:
UserWarning: Skipping features without any observed values: [' ']. At
least one non-missing value is needed for imputation with
strategy='mean'.
  warnings.warn(
```

	Standard deviation	Proportion of Variance	Cumulative Proportion
PC1	2.401951	0.144234	0.144234
PC2	2.346199	0.137616	0.281849
PC3	1.961911	0.096227	0.378076
PC4	1.643769	0.067549	0.445626



t-SNE

```
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(['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')
```

```
41
```

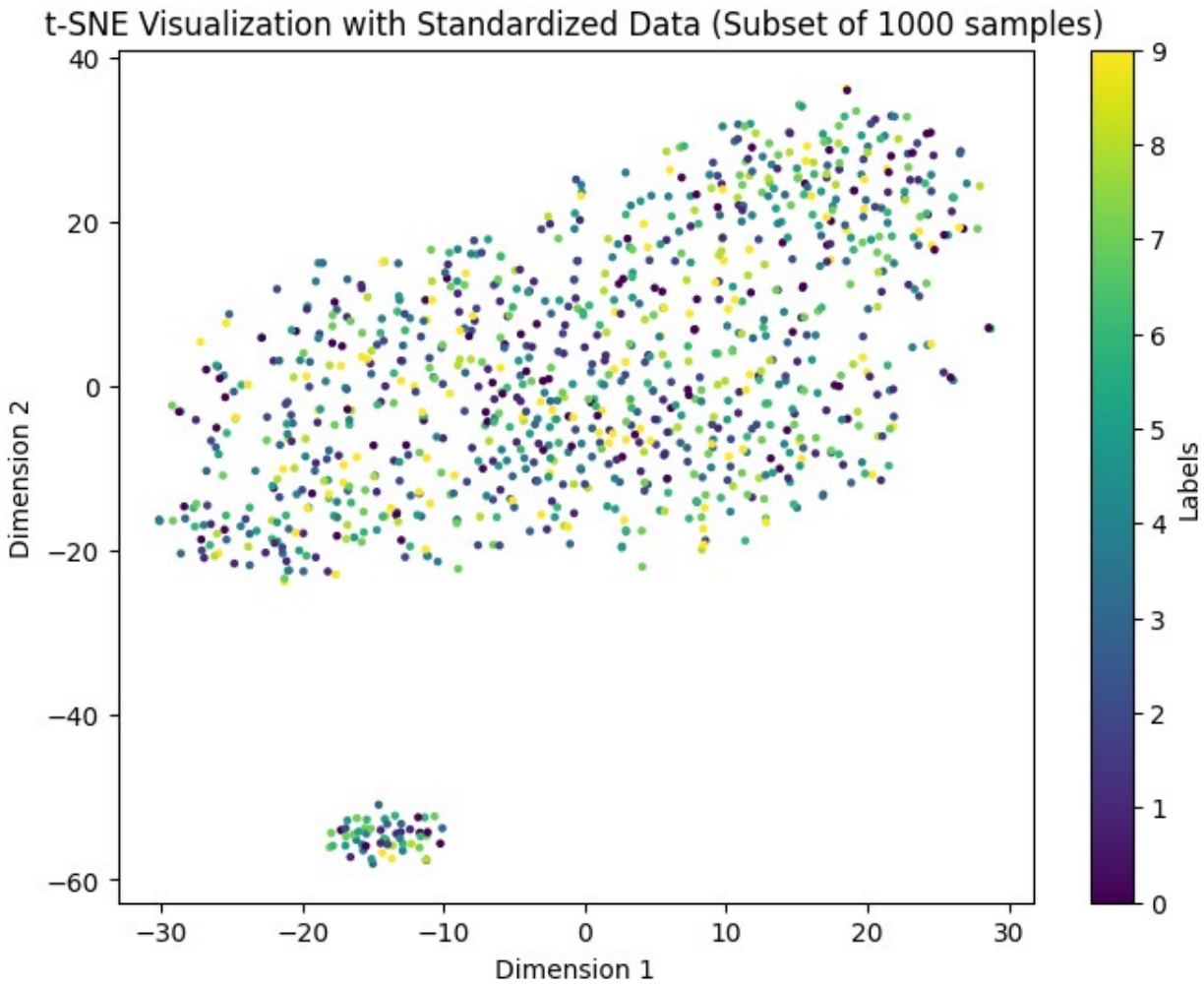
```
import tensorflow as tf
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import numpy as np

# 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()
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 1000 samples in 0.001s...
[t-SNE] Computed neighbors for 1000 samples in 0.105s...
[t-SNE] Computed conditional probabilities for sample 1000 / 1000
[t-SNE] Mean sigma: 1.384183
[t-SNE] KL divergence after 250 iterations with early exaggeration:
67.696228
[t-SNE] KL divergence after 1000 iterations: 1.612201
```



Autoendoders

Binary masking

```
import numpy as np

# Sample data (replace with your actual data)
data = np.array([10, 25, 5, 18, 30])

# Define a threshold for the mask
threshold = 20

# Create the binary mask
mask = data > threshold  # Creates a mask where values above the
threshold are True

# Apply the mask to the data
masked_data = data[mask]  # Selects elements where the mask is True

# Print the results
```

```
print("Original data:", data)
print("Mask:", mask)
print("Masked data:", masked_data)
```

```
Original data: [10 25  5 18 30]
Mask: [False  True False False  True]
Masked data: [25 30]
```

```
import numpy as np

# Sample data (replace with your actual data)
x = np.array([10, 25, 5])

# Define probabilities for each element of x [1, 0, 1]
p_m = np.array([1, 0, 1])

# Create the binary mask using binomial distribution
mask = np.random.binomial(1, p_m, x.shape)

# Apply the mask to the data
masked_data = x * mask # Apply mask by element-wise multiplication

# Print the results
print("Original data (x):", x)
print("Mask:", mask)
print("Masked data:", masked_data)
```

```
Original data (x): [10 25  5]
Mask: [1 0 1]
Masked data: [10  0  5]
```

```
import numpy as np
import pandas as pd

#Example DataFrame (replace with your actual dataset)

data=pd.DataFrame({
    'Cell_length': np.random.randint(1, 100, 10),
    'DNA1': np.random.rand(10),
    'DNA2': np.random.rand(10),
    'CD45RA': np.random.rand(10),
    'CD133': np.random.rand(10),
    'label': np.random.choice([0, 1], 10) # Binary label column
})

# Convert DataFrame to a NumPy array for easier manipulation

data_np = data.values

# Set the masking probability p_m 0.5 # probability of keeping a value
(set to 1)
```

```
p_m=0.5

#Generate binary mask with np.random.binomial

# This creates a mask with 1s and es, where 1 appears with probability
p_m

mask=np.random.binomial(1, p_m, data_np.shape)

#Apply the binary mask to the data

masked_data = data_np*mask

masked_data_df = pd.DataFrame (masked_data, columns=data.columns)
#Convent back to DataFrame if needed
print("Original Data: \n", data)

print("\Binary Mask: \n", mask)

print("\nMasked Data:\n", masked_data_df)
```

```
Original Data:
    Cell_length      DNA1      DNA2    CD45RA     CD133  label
0            37  0.833478  0.734158  0.727948  0.261915      1
1            15  0.712300  0.192982  0.667822  0.052265      1
2            12  0.749902  0.274659  0.404928  0.917866      0
3            63  0.625738  0.352232  0.135970  0.251107      0
4            46  0.960928  0.445247  0.847065  0.849814      0
5            80  0.260756  0.248592  0.166385  0.964399      1
6            56  0.913324  0.592140  0.620640  0.287783      0
7            31  0.395527  0.128589  0.854315  0.125333      1
8            40  0.425765  0.877414  0.231974  0.557664      1
9            27  0.879432  0.256707  0.672520  0.004837      1
\Binary Mask:
 [[1 1 0 0 1 1]
 [0 1 1 0 0 0]
 [1 0 0 1 1 0]
 [1 1 0 0 0 0]
 [1 0 0 0 0 0]
 [1 0 1 0 0 1]
 [0 1 1 0 0 0]
 [1 1 1 1 0 0]
 [1 1 1 1 1 1]
 [0 1 1 1 1 0]]

Masked Data:
    Cell_length      DNA1      DNA2    CD45RA     CD133  label
0          37.0  0.833478  0.000000  0.000000  0.261915    1.0
1           0.0  0.712300  0.192982  0.000000  0.000000    0.0
2          12.0  0.000000  0.000000  0.404928  0.917866    0.0
```

```
3          63.0  0.625738  0.000000  0.000000  0.000000    0.0
4          46.0  0.000000  0.000000  0.000000  0.000000    0.0
5          80.0  0.000000  0.248592  0.000000  0.000000    1.0
6           0.0  0.913324  0.592140  0.000000  0.000000    0.0
7          31.0  0.395527  0.128589  0.854315  0.000000    0.0
8          40.0  0.425765  0.877414  0.231974  0.557664    1.0
9           0.0  0.879432  0.256707  0.672520  0.004837    0.0
```

Randomly shuffle the column values in the dataframes

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

# Sample DataFrame
data = {
    'A': [1, 2, 3, 4, 5],
    'B': ['p', 'q', 'r', 's', 't'],
    'C': [10, 20, 30, 40, 50]
}

df = pd.DataFrame(data)

# Shuffling
shuffled_df = df.apply(lambda x: np.random.permutation(x))

print("Original DataFrame:")
print(df)

print("\nShuffled DataFrame:")
print(shuffled_df)
```

```
Original DataFrame:
   A  B   C
0  1  p  10
1  2  q  20
2  3  r  30
3  4  s  40
4  5  t  50

Shuffled DataFrame:
   A  B   C
0  4  q  50
1  5  r  20
2  3  t  40
3  2  p  30
4  1  s  10
```

x * (1-m) + x_shuffled * m = x_corrupted

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

# Sample DataFrame
data = {
    'A': [1, 2, 3, 4, 5],
    'B': ['p', 'q', 'r', 's', 't'],
    'C': [10, 20, 30, 40, 50]
}

df = pd.DataFrame(data)

# Create corrupted DataFrame and store the shuffled DataFrame
corrupted_df = df.copy()
shuffled_df = df.copy()
for col in df.columns:
    # Shuffle column values and store them in shuffled_df
    shuffled_values = np.random.permutation(df[col].values)
    shuffled_df[col] = shuffled_values  # Assign shuffled values to shuffled_df
    # Create a mask for the column
    mask = np.random.rand(len(df)) < 0.5
    # Apply corruption using the mask and shuffled values
    corrupted_df.loc[mask, col] = shuffled_values[mask]

print("Original DataFrame:")
print(df)

print("\nShuffled DataFrame:")
print(shuffled_df)

print("\nCorrupted DataFrame:")
print(corrupted_df)
```

```
Original DataFrame:
   A  B   C
0  1  p  10
1  2  q  20
2  3  r  30
3  4  s  40
4  5  t  50

Shuffled DataFrame:
   A  B   C
0  2  t  20
1  5  s  40
2  1  q  10
3  4  r  30
4  3  p  50
```

```
Corrupted DataFrame:
   A  B   C
0  2  t  20
1  5  s  40
2  3  q  30
3  4  s  40
4  3  t  50
```

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

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

# Sample DataFrame
data = {
    'A': [1, 2, 3, 4, 5],
    'B': ['p', 'q', 'r', 's', 't'],
    'C': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)

# Apply the function to the sample DataFrame
df_corrupted = corrupt_data(df, corruption_prob=0.3)  # 30% corruption
probability

# Print the original and corrupted DataFrames
print("Original DataFrame:")
print(df)
print("\nCorrupted DataFrame:")
print(df_corrupted)
```

```
Original DataFrame:
   A  B   C
0  1  p  10
1  2  q  20
2  3  r  30
3  4  s  40
4  5  t  50
```

```
Corrupted DataFrame:
   A  B   C
0  3  p  10
1  1  q  30
2  3  r  30
3  2  s  40
4  5  q  50
```

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

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

# Sample DataFrame
data = {
    'A': [1, 2, 3, 4, 5],
    'B': ['p', 'q', 'r', 's', 't'],
    'C': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)

# Apply the function to the sample DataFrame
df_corrupted = corrupt_data(df, corruption_prob=0.3)  # 30% corruption
probability
# Generate the mask matrix
mask_new = 1 * (df != df_corrupted)

# Print the mask matrix
print("\nMask Matrix (mask_new):")
print(mask_new)
# Print the original and corrupted DataFrames
print("Original DataFrame:")
print(df)
print("\nCorrupted DataFrame:")
print(df_corrupted)
```

```
Mask Matrix (mask_new):
   A  B  C
```

```
0  0  0  0
1  0  0  0
2  0  0  0
3  0  0  0
4  0  1  0
Original DataFrame:
   A  B   C
0  1  p  10
1  2  q  20
2  3  r  30
3  4  s  40
4  5  t  50

Corrupted DataFrame:
   A  B   C
0  1  p  10
1  2  q  20
2  3  r  30
3  4  s  40
4  5  r  50
```

Apply the function on corruption technique on dataset

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

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

# Sample DataFrame
data = {
    'A': [1, 2, 3, 4, 5],
    'B': ['p', 'q', 'r', 's', 't'],
    'C': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)

# Apply the function to the sample DataFrame
df_corrupted = corrupt_data(df, corruption_prob=0.3)  # 30% corruption
probability
```

```
# Get the indices of labeled (corrupted) and unlabeled (uncorrupted)
data
labeled_indices = mask_new.any(axis=1)
unlabeled_indices = ~labeled_indices  # Invert the labeled indices

# Create the labeled and unlabeled datasets
x_labeled = df[labeled_indices]
y_labeled = mask_new[labeled_indices]  # Using mask_new as labels
x_unlabeled = df[unlabeled_indices]
y_unlabeled = mask_new[unlabeled_indices]  # Unlabeled data has mask
values of 0

# Print the shapes of the datasets
print("Shape of x_labeled:", x_labeled.shape)
print("Shape of y_labeled:", y_labeled.shape)
print("Shape of x_unlabeled:", x_unlabeled.shape)
print("Shape of y_unlabeled:", y_unlabeled.shape)
```

```
Shape of x_labeled: (1, 3)
Shape of y_labeled: (1, 3)
Shape of x_unlabeled: (4, 3)
Shape of y_unlabeled: (4, 3)
```

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

Data corruption

mask_new = 1*(data_filtered!=data_corrupted)

```
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

# Sample DataFrame
data = {
    'A': [1, 2, 3, 4, 5],
    'B': ['p', 'q', 'r', 's', 't'],
```

```
    'C': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)

# Apply the function to the sample DataFrame
df_corrupted = corrupt_data(df, corruption_prob=0.3)  # 30% corruption
probability

# Generate the mask matrix before using it
mask_new = 1 * (df != df_corrupted) # This line was missing, causing
the error

# Get the indices of labeled (corrupted) and unlabeled (uncorrupted)
data
labeled_indices = mask_new.any(axis=1)
unlabeled_indices = ~labeled_indices  # Invert the labeled indices

# Create the labeled and unlabeled datasets
x_labeled = df[labeled_indices]
y_labeled = mask_new[labeled_indices]  # Using mask_new as labels
x_unlabeled = df[unlabeled_indices]
y_unlabeled = mask_new[unlabeled_indices]  # Unlabeled data has mask
values of 0

# 1. Split into labeled and unlabeled data
labeled_indices = (df != df_corrupted).any(axis=1) # Comparing df and
df_corrupted
unlabeled_indices = ~labeled_indices

x_labeled = df[labeled_indices]
y_labeled = (df != df_corrupted)[labeled_indices].astype(int) # Create
labels (1 for corrupted, 0 for not corrupted)
x_unlabeled = df[unlabeled_indices]
y_unlabeled = (df != df_corrupted)[unlabeled_indices].astype(int) #
Labels for unlabeled data (all 0)

# 2. Split labeled data into train and test sets (70/30 ratio)
x_train, x_test, y_train, y_test = train_test_split(
    x_labeled, y_labeled, test_size=0.3, random_state=42
)
# Print the datasets
print("\nLabeled Data (x_labeled):")
print(x_labeled)
print("\nLabels for Labeled Data (y_labeled):")
print(y_labeled)
print("\nUnlabeled Data (x_unlabeled):")
print(x_unlabeled)
print("\nLabels for Unlabeled Data (y_unlabeled):")
print(y_unlabeled)
```

```
print("\nTraining Data (x_train):")
print(x_train)
print("\nTraining Labels (y_train):")
print(y_train)
print("\nTesting Data (x_test):")
print(x_test)
print("\nTesting Labels (y_test):")
print(y_test)
```

```
Labeled Data (x_labeled):
   A  B   C
1  2  q  20
4  5  t  50

Labels for Labeled Data (y_labeled):
   A  B  C
1  0  1  0
4  1  0  0

Unlabeled Data (x_unlabeled):
   A  B   C
0  1  p  10
2  3  r  30
3  4  s  40

Labels for Unlabeled Data (y_unlabeled):
   A  B  C
0  0  0  0
2  0  0  0
3  0  0  0

Training Data (x_train):
   A  B   C
1  2  q  20

Training Labels (y_train):
   A  B  C
1  0  1  0

Testing Data (x_test):
   A  B   C
4  5  t  50

Testing Labels (y_test):
   A  B  C
4  1  0  0
```

```
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 =
pd.read_csv('/content/drive/MyDrive/data/Levine_32dim.csv')

# 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)
```

```
<ipython-input-8-3347b35315d5>:18: DtypeWarning: Columns (39) have
mixed types. Specify dtype option on import or set low_memory=False.
```

```
  df_levine =
pd.read_csv('/content/drive/MyDrive/data/Levine_32dim.csv')

Shape of x_train_levine: (185938, 42)
Shape of y_train_levine: (185938, 42)
Shape of x_test_levine: (79688, 42)
Shape of y_test_levine: (79688, 42)

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:

        mask = np.random.rand(len(df)) < corruption_prob

        shuffled_values = np.random.permutation(df[col].values)

        corrupted_df.loc[mask, col] = shuffled_values[mask]

    return corrupted_df


df = pd.read_csv(r'/content/drive/MyDrive/Levine_32dim.csv')

df_corrupted = corrupt_data(df, corruption_prob=0.3)


mask_new = 1 * (df != df_corrupted)

labeled_indices = mask_new.any(axis=1)
unlabeled_indices = ~labeled_indices

x_labeled = df[labeled_indices]
y_labeled = mask_new[labeled_indices].astype(int)
x_unlabeled = df[unlabeled_indices]
y_unlabeled = mask_new[unlabeled_indices].astype(int)

x_train, x_test, y_train, y_test = train_test_split(
    x_labeled, y_labeled, test_size=0.3, random_state=42
)

print("\nLabeled Data (x_labeled):")
print(x_labeled)
print("\nLabels for Labeled Data (y_labeled):")
print(y_labeled)
print("\nUnlabeled Data (x_unlabeled):")
print(x_unlabeled)
print("\nLabels for Unlabeled Data (y_unlabeled):")
```

```
print(y_unlabeled)

print("\nTraining Data (x_train):")
print(x_train)
print("\nTraining Labels (y_train):")
print(y_train)
print("\nTesting Data (x_test):")
print(x_test)
print("\nTesting Labels (y_test):")
print(y_test)
```

```
<ipython-input-9-09660811e86f>:18: DtypeWarning: Columns (39) have
mixed types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv(r'/content/drive/MyDrive/Levine_32dim.csv')


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

            CD19      CD22     CD11b       CD4  ...     CD49d    HLA-
DR  \
0      -0.006696  0.066388 -0.009184  0.363602  ...  0.853505
1.664480
1      -0.016654  0.074409  0.808031 -0.035424  ...  0.197818
0.491592
2       0.073855 -0.042977 -0.001881 -0.008781  ...  2.586670
1.308337
```

```
3      -0.017661 -0.044072  0.733698 -0.019066  ...  1.338669
0.140523
4       0.080423  0.495791  1.107627  0.552746  ...  0.180924
0.197332
...          ...       ...       ...       ...  ...       ...       ..
.
265621  3.127012  2.389596  0.212047  0.003287  ...  0.069388
3.550516
265622 -0.055620 -0.007261  0.063395  0.145304  ...  0.533736
0.123758
265623 -0.008864 -0.035158 -0.041845  0.970120  ...  1.269464
0.047215
265624 -0.026111 -0.030837 -0.034641  1.597189  ... -0.055912
0.501536
265625 -0.048786  0.073983 -0.031787  0.078800  ...  0.101955
6.200001

            CD64      CD41   Viability   file_number   event_number
label  \
0      -0.005376 -0.001961    0.648429      3.627711            307
1
1       0.144814  0.868014    0.561384      3.627711            545
1
2      -0.010961 -0.010413    0.643337      3.627711           1726
1
3      -0.013449 -0.026039   -0.026523      3.627711           1766
1
4       0.076167 -0.040488    0.283287      3.627711           2031
1
...          ...       ...         ...           ...            ...
...
265621  0.147588 -0.043806    0.144479      3.669327         102685
NaN
265622 -0.042495 -0.027971    0.236957      3.669327         102686
NaN
265623 -0.008000 -0.025811   -0.003500      3.669327         102690
NaN
265624  0.053884 -0.042602    0.107206      3.669327         102701
NaN
265625  0.296877  0.192786    0.620872      3.669327         102706
NaN

         individual
0                 1
1                 1
2                 1
3                 1
4                 1
...             ... ..
```

```
265621            2
265622            2
265623            2
265624            2
265625            2

[265625 rows x 42 columns]

Labels for Labeled Data (y_labeled):
        Time   Cell_length   DNA1   DNA2   CD45RA   CD133   CD19
CD22  \
0          0             0      0      0        1       0      0
0
1          0             1      0      1        0       0      1
1
2          0             0      1      1        0       0      0
0
3          0             1      0      0        0       0      0
0
4          1             0      0      1        0       1      0
0
...      ...           ...    ...    ...      ...     ...    ...    ..
.
265621     0             0      0      0        0       1      0
0
265622     0             0      0      0        1       1      1
0
265623     0             0      0      1        1       1      0
1
265624     0             0      0      0        1       0      0
0
265625     0             1      0      1        0       1      1
1

         CD11b   CD4  ...   CD49d   HLA-DR   CD64   CD41
Viability  \
0            0     1  ...       0        0      0      0           0

1            0     1  ...       1        0      0      1           0

2            0     0  ...       1        0      0      0           0

3            0     0  ...       1        0      0      0           1

4            1     0  ...       1        1      0      1           1

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

265621       0     0  ...       0        0      0      0           0
```

```
265622       0     1  ...       0        0      1      0           0

265623       0     0  ...       0        0      0      0           0

265624       0     1  ...       0        0      0      1           0

265625       0     0  ...       0        1      0      0           0


         file_number   event_number   label   individual
0                  0              0       0           0  0
1                  1              0       0           0  0
2                  1              0       1           0  0
3                  0              0       0           0  0
4                  0              0       0           0  0
...              ...            ...     ...         ... ..
265621             0              0       0           0  0
265622             1              0       0           0  0
265623             0              1       0           1  0
265624             1              0       0           0  0
265625             0              0       0           0  0

[265625 rows x 42 columns]

Unlabeled Data (x_unlabeled):
            Time   Cell_length      DNA1      DNA2    CD45RA     CD133
\
161285  246888.0            27  3.810219  4.470415  1.550783  0.166561


            CD19      CD22     CD11b       CD4  ...     CD49d    HLA-
DR  \
161285  1.890609  2.099512  0.057983  0.189265  ...  0.371446
3.449781

            CD64      CD41   Viability   file_number   event_number
label  \
161285 -0.010887  0.252074     0.89409      3.627711         217110
NaN

         individual
161285            1

[1 rows x 42 columns]

Labels for Unlabeled Data (y_unlabeled):
        Time   Cell_length   DNA1   DNA2   CD45RA   CD133   CD19
CD22  \
161285     0             0      0      0        0       0      0
0
```

```
         CD11b   CD4  ...   CD49d   HLA-DR   CD64   CD41
Viability  \
161285       0     0  ...       0        0      0      0           0


         file_number   event_number   label   individual
161285             0              0       0            0  0

[1 rows x 42 columns]

Training Data (x_train):
               Time   Cell_length      DNA1      DNA2    CD45RA
CD133  \
112323   40778.0000            20  4.038789  4.766040  0.358112 -
0.018525
250246  426049.0000            38  6.813376  7.015943  0.850273 -
0.044988
72182   567753.0000            29  3.630845  4.471507  0.797058
0.310841
50911   505487.0000            22  3.221981  4.528588  0.427810 -
0.046120
22336   424718.0000            30  3.959151  4.539984  0.739531 -
0.006289
...             ...           ...       ...       ...       ...
...
119879   69026.0000            35  3.896394  4.568209  0.238602 -
0.022931
259179  518654.4375            57  6.092913  6.812010  1.166134 -
0.050529
131932  113944.0000            64  3.943022  4.647186  1.926321
0.094668
146867  179533.0000            37  3.056461  3.595415  0.289241
0.163690
121958   76701.0000            43  3.245267  4.109544  1.506159 -
0.012223

            CD19      CD22     CD11b       CD4  ...     CD49d    HLA-
DR  \
112323 -0.015670 -0.038216 -0.006305  0.095079  ...  0.357806
0.160120
250246 -0.041293 -0.023683 -0.039434  0.690446  ...  0.102453 -
0.052398
72182   1.526767 -0.015747  0.784267 -0.008252  ...  2.275365
2.919452
50911   2.706543  3.028495  0.162550  0.094481  ...  0.210280
2.638860
22336   0.038441 -0.008545  0.073549  0.471662  ...  0.767908 -
0.040116
...          ...       ...       ...       ...  ...       ...       ..
.
```

```
119879  0.063364  1.254473  0.831690  0.713964  ...  0.333200
0.441514
259179  1.004339  2.447712  0.174300 -0.032400  ...  0.795383
5.519310
131932  0.203043  0.388187  0.414550 -0.016618  ...  0.896863
0.850431
146867 -0.018320  0.833612 -0.018161 -0.037270  ...  0.442390
0.624523
121958  0.152900  0.884522  1.232073  0.336815  ...  1.173544
1.117928

            CD64      CD41   Viability   file_number   event_number
label  \
112323 -0.040943 -0.005514    0.295370      3.627711          32062
NaN
250246 -0.025210 -0.050443   -0.015361      3.669327          63017
NaN
72182   0.021405 -0.022959    0.053688      3.627711         393504
13
50911  -0.014304  0.007113    0.184492      3.627711         374300
9
22336   0.229680 -0.014554    0.121214      3.627711         333161
7
...          ...       ...         ...           ...            ...
...
119879  0.538406  0.375899    1.523324      3.627711          66344
NaN
259179  0.175531  0.343033   -0.054672      3.669327          84606
NaN
131932  0.238235 -0.032165    1.563885      3.627711         119016
NaN
146867  0.284806 -0.007662    1.423528      3.627711         170997
NaN
121958  0.733002  0.060496    1.630995      3.627711          75822
NaN

         individual
112323            1
250246            2
72182             1
50911             1
22336             1
...             ... ..
119879            1
259179            2
131932            1
146867            1
121958            1

[185937 rows x 42 columns]
```

```
Training Labels (y_train):
        Time   Cell_length   DNA1   DNA2   CD45RA   CD133   CD19
CD22  \
112323     0             1      0      1        1       0      0
0
250246     0             0      0      1        0       0      0
0
72182      0             0      1      1        0       1      0
0
50911      0             0      0      0        1       0      0
0
22336      0             1      1      0        1       0      1
0
...      ...           ...    ...    ...      ...     ...    ...    ..
.
119879     0             0      0      0        0       1      0
0
259179     1             0      0      0        1       0      0
0
131932     0             0      0      0        0       0      0
0
146867     1             1      0      1        0       0      0
0
121958     0             0      0      0        0       1      0
0

          CD11b   CD4  ...   CD49d   HLA-DR   CD64   CD41
Viability  \
112323        0     0  ...       0        0      0      0           1

250246        0     0  ...       0        0      0      0           0

72182         0     0  ...       0        0      0      1           0

50911         1     0  ...       0        0      0      0           0

22336         1     0  ...       1        0      1      1           0

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

119879        0     1  ...       0        1      0      1           0

259179        1     0  ...       0        1      1      0           0

131932        0     0  ...       1        0      0      0           0

146867        0     1  ...       0        0      1      0           0

121958        0     1  ...       0        0      0      0           0
```

```
        file_number   event_number   label   individual
112323            0              0       0           0  0
250246            0              0       0           1  0
72182             0              0       0           1  0
50911             0              0       1           0  0
22336             0              0       0           0  0
...             ...            ...     ...         ... ..
119879            0              1       1           0  0
259179            0              0       0           1  0
131932            0              1       0           0  0
146867            0              0       0           0  0
121958            0              0       0           0  0

[185937 rows x 42 columns]

Testing Data (x_test):
            Time   Cell_length      DNA1      DNA2    CD45RA     CD133
\
159744  239502.0            25  3.875798  4.877994  0.064975 -0.032015

253785  461849.0            30  6.947580  7.073083 -0.022401 -0.018339

219101  549569.0            26  3.880525  4.793793 -0.021495  0.437048

137512  137581.0            29  3.752332  4.651581  1.261561 -0.020296

116835   57715.0            35  3.839168  4.179122  1.132873  0.403402

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

70534   254998.0            32  3.931968  4.841372  1.181050  0.159801

86718   149705.0            30  6.379300  6.665539  1.725482 -0.042390

87369   193971.0            42  6.354186  6.832409  0.648614 -0.020578

95453     1946.0            59  6.684025  7.125886 -0.007148 -0.017551

194416  401346.0            33  4.329600  4.787856  0.695976  0.717245


             CD19      CD22     CD11b       CD4  ...     CD49d    HLA-
DR  \
159744 -0.031299 -0.020087  0.306332  1.226449  ...  0.523893 -
0.010695
253785 -0.025362 -0.014267  1.950981 -0.041442  ...  0.849110
0.030091
219101  1.053145  0.610920 -0.036768  0.629747  ...  0.694252
3.324835
```

```
137512 -0.015791 -0.037131 -0.005347  0.603409  ...  1.467767
4.860878
116835 -0.015633 -0.000987  0.051322  1.446739  ...  0.578184
0.109987
...          ...       ...       ...       ...  ...       ...       ..
.
70534   1.224687  0.129718  0.131855 -0.013561  ...  1.165955
3.509306
86718  -0.014799 -0.007869  0.993733  0.126719  ...  0.062300 -
0.010911
87369  -0.013731 -0.011855 -0.005835 -0.025524  ... -0.008457 -
0.030210
95453  -0.028611  0.090126  2.875104  0.040365  ...  0.742402
0.871192
194416  0.133931 -0.014450  1.991794 -0.023792  ...  0.168983
2.990170

            CD64      CD41   Viability   file_number   event_number
label  \
159744  0.228701 -0.001603    0.167130      3.627711         212318
NaN
253785 -0.013564 -0.001851   -0.023146      3.669327          71287
NaN
219101  0.102830 -0.020054    0.460168      3.627711         388974
NaN
137512  0.014485  0.144562    1.538764      3.627711         139933
NaN
116835  0.160224  0.666484    0.765420      3.627711          52386
NaN
...          ...       ...         ...           ...            ...
...
70534   0.200088  0.727026    0.454695      3.627711         222580
13
86718  -0.018017 -0.027270   -0.017355      3.669327          33251
8
87369  -0.022345 -0.034298    0.395601      3.669327          42031
8
95453   2.352954  0.200624    0.398390      3.669327            224
10
194416  2.348327  2.246444    0.973870      3.627711         318426
NaN

         individual
159744            1
253785            2
219101            1
137512            1
116835            1
...             ... ..
```

```
70534              1
86718              2
87369              2
95453              2
194416             1

[79688 rows x 42 columns]

Testing Labels (y_test):
        Time   Cell_length   DNA1   DNA2   CD45RA   CD133   CD19
CD22  \
159744     1             0      0      0        0       0      0
1
253785     1             0      0      0        1       1      0
0
219101     0             1      0      0        0       1      0
0
137512     0             0      1      0        0       0      0
0
116835     0             1      0      0        0       1      1
0
...      ...           ...    ...    ...      ...     ...    ...    ..
.
70534      0             0      1      0        1       0      0
0
86718      0             0      0      0        0       0      0
0
87369      0             1      0      0        0       1      0
0
95453      0             0      0      1        0       0      0
0
194416     0             0      0      1        0       0      0
1

         CD11b   CD4  ...   CD49d   HLA-DR   CD64   CD41
Viability  \
159744       0     0  ...       0        0      0      0           0

253785       0     1  ...       0        0      1      0           1

219101       0     0  ...       0        1      0      1           1

137512       0     0  ...       0        0      0      0           0

116835       0     0  ...       0        1      1      1           0

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

70534        0     0  ...       0        0      0      1           0
```

```
86718        1     1  ...        0         1      0      1            0

87369        1     0  ...        0         0      0      0            1

95453        1     0  ...        0         0      0      0            0

194416       0     1  ...        1         0      0      0            0


         file_number   event_number   label   individual
159744             0              1       0            0  0
253785             0              1       0            0  0
219101             0              0       0            0  0
137512             0              0       0            0  0
116835             0              0       0            0  0
...              ...            ...     ...          ... ..
70534              0              0       0            0  0
86718              1              0       0            0  0
87369              0              0       0            0  0
95453              0              0       1            0  0
194416             0              0       0            1  0

[79688 rows x 42 columns]
```

```
import pandas as pd
import numpy as np

# Load the CSV file into the DataFrame first
df = pd.read_csv('/content/drive/MyDrive/data/Levine_32dim.csv')
print(df.head())

# Check if 'label' column exists. If not, handle it accordingly.
if 'label' in df.columns:
  label_column = 'label' # Assign 'label' to label_column if it exists
else:
  print(f"Label column 'label' not found in the dataset. Using index
instead.")
  label_column = df.index

label_column = 'label' if 'label' in df.columns else None

# If the label column is None (i.e. it wasn't found)
if label_column is None:
    print(f"Label column 'label' not found in the dataset. Using index
instead.")
    # If we want to proceed with index as 'label', we can create a new
column:
    df['label'] = df.index
    label_column = 'label'  # Update label_column to reflect the new
column
```

```
else:
    print("Label column found in the dataset.")
```

```
<ipython-input-11-69e063aee4e3>:5: DtypeWarning: Columns (39) have
mixed types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv('/content/drive/MyDrive/data/Levine_32dim.csv')
```

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

       CD22     CD11b       CD4  ...     CD49d    HLA-DR      CD64
CD41  \
0  0.066388 -0.009184  0.363602  ...  0.853505  1.664480 -0.005376 -
0.001961
1  0.074409  0.808031 -0.035424  ...  0.197818  0.491592  0.144814
0.868014
2 -0.042977 -0.001881 -0.008781  ...  2.586670  1.308337 -0.010961 -
0.010413
3 -0.044072  0.733698 -0.019066  ...  1.338669  0.140523 -0.013449 -
0.026039
4  0.495791  1.107627  0.552746  ...  0.180924  0.197332  0.076167 -
0.040488

    Viability   file_number   event_number   label   individual
0    0.648429      3.627711            307       1            1
1    0.561384      3.627711            545       1            1
2    0.643337      3.627711           1726       1            1
3   -0.026523      3.627711           1766       1            1
4    0.283287      3.627711           2031       1            1

[5 rows x 42 columns]
Label column 'label' not found in the dataset. Using index instead.
Label column 'label' not found in the dataset. Using index instead.
```

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

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

label_column = 'label'

# Assuming df is your original DataFrame:
df = pd.read_csv('/content/drive/MyDrive/data/Levine_32dim.csv')

# Apply corruption
df_corrupted = corrupt_data(df, corruption_prob=0.3)

# Generate mask_new
mask_new = 1 * (df != df_corrupted)

# Filter based on mask_new (if necessary)
data_filtered = df[mask_new.any(axis=1)] # Example filtering

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

```
<ipython-input-3-232301ef4ece>:19: DtypeWarning: Columns (39) have
mixed types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv('/content/drive/MyDrive/data/Levine_32dim.csv')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-3-232301ef4ece> in <cell line: 31>()
     29
     30 # Separate features (X) and labels (y) from the labeled data
---> 31 x = data_filtered.drop(columns=[label_column])
     32 y = data_filtered[label_column]
     33

/usr/local/lib/python3.10/dist-packages/pandas/core/frame.py in drop(self, labels, axis, index, columns, level, inplace, errors)
   5579                 weight  1.0     0.8
   5580         """
-> 5581         return super().drop(
   5582             labels=labels,
   5583             axis=axis,

/usr/local/lib/python3.10/dist-packages/pandas/core/generic.py in drop(self, labels, axis, index, columns, level, inplace, errors)
   4786         for axis, labels in axes.items():
   4787             if labels is not None:
-> 4788                 obj = obj._drop_axis(labels, axis, level=level, errors=errors)
   4789
   4790         if inplace:

/usr/local/lib/python3.10/dist-packages/pandas/core/generic.py in _drop_axis(self, labels, axis, level, errors, only_slice)
   4828                 new_axis = axis.drop(labels, level=level, errors=errors)
   4829             else:
-> 4830                 new_axis = axis.drop(labels, errors=errors)
   4831             indexer = axis.get_indexer(new_axis)
   4832

/usr/local/lib/python3.10/dist-packages/pandas/core/indexes/base.py in drop(self, labels, errors)
   7068         if mask.any():
   7069             if errors != "ignore":
-> 7070                 raise KeyError(f"{labels[mask].tolist()} not found in axis")
   7071             indexer = indexer[~mask]
   7072         return self.delete(indexer)

KeyError: "['label'] not found in axis"
```

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

```
label_column = 'label'

# Create or load your DataFrame here
# Replace 'your_data.csv' with the actual path to your data file if
loading from a file
# new_df = pd.read_csv('your_data.csv')
# Or create a DataFrame from other sources, e.g.,:
df = pd.read_csv(r'/content/drive/MyDrive/Levine_32dim.csv')

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

```
Label column 'label' not found in the dataset.
```

```
<ipython-input-7-a8924857c7c6>:10: DtypeWarning: Columns (39) have
mixed types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv(r'/content/drive/MyDrive/Levine_32dim.csv')
```

```
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):
   col1 col2
0     1    a

Test features (X_test):
   col1 col2
1     2    b

Training labels (y_train):
0    0.0
Name: label, dtype: float64

Test labels (y_test):
1    1.0
Name: label, dtype: float64

Shape of X_train: (1, 2)
Shape of X_test: (1, 2)
Shape of y_train: (1,)
Shape of y_test: (1,)
shape of X (2, 2)
shape of y (2,)
```

implement logistic regression and xgboost using sklearn library for your x_train, y_train data and test it on x_test data. return the ouptuts probabilities

calculate the logistic regression and xgboost loss values for the encoded data from the encoder

```
# Import necessary libraries
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
import numpy as np

# Generate a synthetic dataset for demonstration
x, y = make_classification(n_samples=100, n_features=10,
random_state=42)
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=42)
```

```
def logistic_regression_with_outputs(x_train, y_train, x_test):
    """
    Implements Logistic Regression and returns output probabilities.

    Args:
        x_train, y_train: Training data.
        x_test: Testing data.

    Returns:
        y_pred_prob: Predicted probabilities for x_test.
    """
    # Create and train the Logistic Regression model
    model = LogisticRegression()
    model.fit(x_train, y_train)

    # Predict probabilities on the test data
    y_pred_prob = model.predict_proba(x_test)

    # Print the output probabilities
    print("Logistic Regression Output Probabilities:")
    print(y_pred_prob)

    return y_pred_prob

def xgboost_with_outputs(x_train, y_train, x_test):
    """
    Implements XGBoost and returns output probabilities.

    Args:
        x_train, y_train: Training data.
        x_test: Testing data.

    Returns:
        y_pred_prob: Predicted probabilities for x_test.
    """
    # Create and train the XGBoost model
    model = XGBClassifier(use_label_encoder=False,
eval_metric='logloss')
    model.fit(x_train, y_train)

    # Predict probabilities on the test data
    y_pred_prob = model.predict_proba(x_test)

    # Print the output probabilities
    print("\nXGBoost Output Probabilities:")
    print(y_pred_prob)

    return y_pred_prob
```

```python
# Call the Logistic Regression function and print results
y_pred_prob_logistic = logistic_regression_with_outputs(x_train,
y_train, x_test)
print("\nFinal Logistic Regression Probabilities:")
print(y_pred_prob_logistic)

# Call the XGBoost function and print results
y_pred_prob_xgboost = xgboost_with_outputs(x_train, y_train, x_test)
print("\nFinal XGBoost Probabilities:")
print(y_pred_prob_xgboost)
```

```
Logistic Regression Output Probabilities:
[[9.04675251e-01 9.53247494e-02]
 [7.97965075e-02 9.20203492e-01]
 [1.33736145e-02 9.86626386e-01]
 [7.86373395e-02 9.21362661e-01]
 [2.07691091e-02 9.79230891e-01]
 [9.72546031e-01 2.74539693e-02]
 [9.78171609e-01 2.18283908e-02]
 [2.54443533e-01 7.45556467e-01]
 [2.14117582e-01 7.85882418e-01]
 [1.65396820e-01 8.34603180e-01]
 [9.88737978e-01 1.12620223e-02]
 [2.80587665e-03 9.97194123e-01]
 [7.85824805e-04 9.99214175e-01]
 [9.99130522e-01 8.69477913e-04]
 [5.13601693e-03 9.94863983e-01]
 [9.67299512e-01 3.27004880e-02]
 [3.91255210e-02 9.60874479e-01]
 [9.82741573e-01 1.72584274e-02]
 [9.69083770e-01 3.09162297e-02]
 [9.97698675e-01 2.30132477e-03]]

Final Logistic Regression Probabilities:
[[9.04675251e-01 9.53247494e-02]
 [7.97965075e-02 9.20203492e-01]
 [1.33736145e-02 9.86626386e-01]
 [7.86373395e-02 9.21362661e-01]
 [2.07691091e-02 9.79230891e-01]
 [9.72546031e-01 2.74539693e-02]
 [9.78171609e-01 2.18283908e-02]
 [2.54443533e-01 7.45556467e-01]
 [2.14117582e-01 7.85882418e-01]
 [1.65396820e-01 8.34603180e-01]
 [9.88737978e-01 1.12620223e-02]
 [2.80587665e-03 9.97194123e-01]
 [7.85824805e-04 9.99214175e-01]
 [9.99130522e-01 8.69477913e-04]
 [5.13601693e-03 9.94863983e-01]
 [9.67299512e-01 3.27004880e-02]
```

```
 [3.91255210e-02 9.60874479e-01]
 [9.82741573e-01 1.72584274e-02]
 [9.69083770e-01 3.09162297e-02]
 [9.97698675e-01 2.30132477e-03]]

XGBoost Output Probabilities:
[[0.973202   0.02679799]
 [0.05494148 0.9450585 ]
 [0.03825974 0.96174026]
 [0.0221805  0.9778195 ]
 [0.00801504 0.99198496]
 [0.98203075 0.01796927]
 [0.97422457 0.02577544]
 [0.03004247 0.96995753]
 [0.02741474 0.97258526]
 [0.02611899 0.973881  ]
 [0.99399126 0.00600873]
 [0.09624082 0.9037592 ]
 [0.04554039 0.9544596 ]
 [0.9983915  0.00160852]
 [0.00383204 0.99616796]
 [0.9877788  0.0122212 ]
 [0.15245807 0.8475419 ]
 [0.92679    0.07320997]
 [0.98324716 0.01675281]
 [0.9970123  0.00298767]]

Final XGBoost Probabilities:
[[0.973202   0.02679799]
 [0.05494148 0.9450585 ]
 [0.03825974 0.96174026]
 [0.0221805  0.9778195 ]
 [0.00801504 0.99198496]
 [0.98203075 0.01796927]
 [0.97422457 0.02577544]
 [0.03004247 0.96995753]
 [0.02741474 0.97258526]
 [0.02611899 0.973881  ]
 [0.99399126 0.00600873]
 [0.09624082 0.9037592 ]
 [0.04554039 0.9544596 ]
 [0.9983915  0.00160852]
 [0.00383204 0.99616796]
 [0.9877788  0.0122212 ]
 [0.15245807 0.8475419 ]
 [0.92679    0.07320997]
 [0.98324716 0.01675281]
 [0.9970123  0.00298767]]
```

```
/usr/local/lib/python3.10/dist-packages/xgboost/core.py:158:
UserWarning: [14:30:22] WARNING: /workspace/src/learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
```

Calculating the Logistic Regression Log Loss and XGBoost Log Loss

```
# Import necessary libraries
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, log_loss # Import log_loss
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
import numpy as np

# Generate a synthetic dataset for demonstration
x, y = make_classification(n_samples=100, n_features=10,
random_state=42)
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=42)

def logistic_regression_with_outputs(x_train, y_train, x_test,
y_test): # Added y_test
    """
    Implements Logistic Regression, calculates and prints log loss.

    Args:
        x_train, y_train: Training data.
        x_test, y_test: Testing data.

    Returns:
        y_pred_prob: Predicted probabilities for x_test.
    """
    model = LogisticRegression()
    model.fit(x_train, y_train)
    y_pred_prob = model.predict_proba(x_test)

    # Calculate and print log loss
    lr_log_loss = log_loss(y_test, y_pred_prob)
    print(f"Logistic Regression Log Loss: {lr_log_loss}")

    return y_pred_prob

def xgboost_with_outputs(x_train, y_train, x_test, y_test): # Added
y_test
    """
    Implements XGBoost, calculates and prints log loss.

    Args:
```

```
        x_train, y_train: Training data.
        x_test, y_test: Testing data.

    Returns:
        y_pred_prob: Predicted probabilities for x_test.
    """
    model = XGBClassifier()
    model.fit(x_train, y_train)
    y_pred_prob = model.predict_proba(x_test)

    # Calculate and print log loss
    xgb_log_loss = log_loss(y_test, y_pred_prob)
    print(f"XGBoost Log Loss: {xgb_log_loss}")

    return y_pred_prob

# Example usage:
logistic_regression_probs = logistic_regression_with_outputs(x_train,
y_train, x_test, y_test) # Pass y_test
xgboost_probs = xgboost_with_outputs(x_train, y_train, x_test, y_test)
# Pass y_test
```

```
Logistic Regression Log Loss: 0.060509459503793404
XGBoost Log Loss: 0.03575391160039887
```

The log loss along with there Predicted Probabilities

```
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier  # Import XGBClassifier
from sklearn.metrics import log_loss
import numpy as np

# Define the logistic regression function
def Logistic(x_train, y_train, x_test, y_test):
     # Check and reshape y_train if needed
    if len(y_train.shape) > 1:
        y_train = y_train.ravel()

    # Define and fit the logistic regression model
    model = LogisticRegression(random_state=42, max_iter=1000)
    model.fit(x_train, y_train)

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

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

    return y_test_hat, loss
```

```python
# Define the XGBoost function
def XGBoost(x_train, y_train, x_test, y_test):
    # Check and reshape y_train if needed
    if len(y_train.shape) > 1:
        y_train = y_train.ravel()

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

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

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

    return y_test_hat, loss

# Get predicted probabilities and log loss for Logistic Regression
y_test_probabilities_logistic, loss_value_logistic = Logistic(x_train,
y_train, x_test, y_test)

# Get predicted probabilities and log loss for XGBoost
y_test_probabilities_xgboost, loss_value_xgboost = XGBoost(x_train,
y_train, x_test, y_test)

# Print results
print("Logistic Regression:")
print("Predicted probabilities:\n", y_test_probabilities_logistic)
print("\nLog Loss:", loss_value_logistic)

print("\nXGBoost:")
print("Predicted probabilities:\n", y_test_probabilities_xgboost)
print("\nLog Loss:", loss_value_xgboost)
```

```
Logistic Regression:
Predicted probabilities:
 [[9.04675251e-01 9.53247494e-02]
 [7.97965075e-02 9.20203492e-01]
 [1.33736145e-02 9.86626386e-01]
 [7.86373395e-02 9.21362661e-01]
 [2.07691091e-02 9.79230891e-01]
 [9.72546031e-01 2.74539693e-02]
 [9.78171609e-01 2.18283908e-02]
 [2.54443533e-01 7.45556467e-01]
 [2.14117582e-01 7.85882418e-01]
 [1.65396820e-01 8.34603180e-01]
 [9.88737978e-01 1.12620223e-02]
```

```
 [2.80587665e-03 9.97194123e-01]
 [7.85824805e-04 9.99214175e-01]
 [9.99130522e-01 8.69477913e-04]
 [5.13601693e-03 9.94863983e-01]
 [9.67299512e-01 3.27004880e-02]
 [3.91255210e-02 9.60874479e-01]
 [9.82741573e-01 1.72584274e-02]
 [9.69083770e-01 3.09162297e-02]
 [9.97698675e-01 2.30132477e-03]]

Log Loss: 0.060509459503793404

XGBoost:
Predicted probabilities:
 [[0.973202   0.02679799]
 [0.05494148 0.9450585 ]
 [0.03825974 0.96174026]
 [0.0221805  0.9778195 ]
 [0.00801504 0.99198496]
 [0.98203075 0.01796927]
 [0.97422457 0.02577544]
 [0.03004247 0.96995753]
 [0.02741474 0.97258526]
 [0.02611899 0.973881  ]
 [0.99399126 0.00600873]
 [0.09624082 0.9037592 ]
 [0.04554039 0.9544596 ]
 [0.9983915  0.00160852]
 [0.00383204 0.99616796]
 [0.9877788  0.0122212 ]
 [0.15245807 0.8475419 ]
 [0.92679    0.07320997]
 [0.98324716 0.01675281]
 [0.9970123  0.00298767]]

Log Loss: 0.03575391160039887

/usr/local/lib/python3.10/dist-packages/xgboost/core.py:158:
UserWarning: [13:23:30] WARNING: /workspace/src/learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
```

Create a function called as self_supervised, x_unlabeled dataset,masking probability,alpha,parameters

Encoder part of auto encoder use Keras

SIR'S CODE

```
from keras.layers import Input, Dense

from keras.models import Model

from keras import models

import numpy as np

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

  #model creation.
  #Defining an encoder.
  #Auto encoder sturcuter-> corrupted input -> encoder ---> latent
space> decoder.
  # working on the encoder part and extracting the latent space
  #creating a fully connecting network with the number of neurons in
the first layer equal to the number features present in the dataset.
  #input layer will be of size 37.

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

#output1--->mask estimation
output1=Dense(int(dimension), activation='sigmoid',
name='mask_estination')(h)

#output2--->feature estimation
output2=Dense(int(dimension),activation='sigmoid',
name='feature_estination')(h)

#relu function negates any value that is less than sigmoid function
takes any input and converts it to the range of 0 to 1.
# input ---> output 1
#|
#|---> output 2
model=Model(inputs=input_layer, outputs=[output1,output2])

model.compile(optimizer = "rmsprop", loss="mask_estimation':
'binary_crossentropy', 'feature_estimation": "mean_squared_error'}},
loss_weights: ('mask_estimation':1, 'feature_estimation': alpha}

corruption_binary_mask=binary_mask(p_m,x_unlabeled) # returns binary
mask with dimesnions equal to that that of x_unlabeled

x_unlabeled_corrupted=corruption(corruption_binary_mask, x_unlabeled)
```

```
# return corrupted unlabeled dataset

model.fit(x_unlabeled_corrupted, {'mask':m_label,
'feature':x_unlabeled}, epochs=epochs, batch_size = batch_size)

name_of_layer=model.layers[1].name

layer_output = model.get_layer(name_of_layer).output

encoder=moddels.Model(inputs=model.input,outputs=layer_output)

return encoder

x_unlab x_unlabeled_scaled
p_m = 0.3
alpha = 2.0
parameters = ('batch_size: 128',
              'epochs': 50
              }

self_supervised(x_unlab,p_m,alpha, paramets)
```

.....

```
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 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - loss: 3.2148
Epoch 2/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 136ms/step - loss: 3.1657
Epoch 3/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 3.1305
Epoch 4/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - loss: 3.1016
Epoch 5/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step - loss: 3.0764
Epoch 6/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step - loss: 3.0538
```

```
Epoch 7/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step - loss: 3.0330
Epoch 8/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - loss: 3.0136
Epoch 9/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - loss: 2.9953
Epoch 10/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - loss: 2.9779
Epoch 11/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.9613
Epoch 12/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 2.9454
Epoch 13/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step - loss: 2.9301
Epoch 14/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - loss: 2.9152
Epoch 15/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.9009
Epoch 16/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 2.8869
Epoch 17/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 2.8733
Epoch 18/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.8601
Epoch 19/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.8472
Epoch 20/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.8346
Epoch 21/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.8222
Epoch 22/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 2.8102
Epoch 23/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - loss: 2.7984
Epoch 24/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.7869
Epoch 25/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.7757
Epoch 26/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - loss: 2.7646
Epoch 27/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.7538
Epoch 28/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.7432
Epoch 29/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - loss: 2.7328
Epoch 30/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step - loss: 2.7227
Epoch 31/50
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step - loss: 2.7128
Epoch 32/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.7031
Epoch 33/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step - loss: 2.6936
Epoch 34/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - loss: 2.6842
Epoch 35/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - loss: 2.6750
Epoch 36/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.6660
Epoch 37/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - loss: 2.6572
Epoch 38/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - loss: 2.6486
Epoch 39/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 2.6401
Epoch 40/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.6317
Epoch 41/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.6235
Epoch 42/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - loss: 2.6155
Epoch 43/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 2.6076
Epoch 44/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - loss: 2.5999
Epoch 45/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - loss: 2.5922
Epoch 46/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.5847
Epoch 47/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - loss: 2.5773
Epoch 48/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - loss: 2.5700
Epoch 49/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - loss: 2.5629
Epoch 50/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.5559

Model: "functional_2"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┃ Connected to                    ┃                        ┃               ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer_1                   │ (None, 37)             │             0 │
│ -                               │                        │               │
```

```
│ (InputLayer)                │                           │
                          │
├─────────────────────────────┼───────────────────────────┼──────────────────
──────────┤
│ dense_1 (Dense)             │ (None, 64)                │           2,432
│ input_layer_1[0][0]         │
├─────────────────────────────┼───────────────────────────┼──────────────────
──────────┤
│ mask_estimation (Dense)     │ (None, 37)                │           2,405
│ dense_1[0][0]               │
├─────────────────────────────┼───────────────────────────┼──────────────────
──────────┤
│ feature_estimation          │ (None, 37)                │           2,405
│ dense_1[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': [3.2148494720458984, 3.1656508445739746,
3.1305243968963623, 3.101625680923462, 3.076428174972534,
3.053760290145874, 3.0329673290252686, 3.0135788917541504,
2.995291233062744, 2.9779343605041504, 2.96134877204895,
2.9454054832458496, 2.930050849914551, 2.91523814201355,
2.900876998901367, 2.8869009017944336, 2.8733301162719727,
2.860090970993042, 2.8471617698669434, 2.8345530033111572,
2.8222227096557617, 2.810180902481079, 2.798426866531372,
2.7869369983673096, 2.7756659984588623, 2.764626979827881,
2.7538084983825684, 2.7432100772857666, 2.7328436374664307,
2.722712516784668, 2.71279239654541, 2.703068256378174,
2.6935575008392334, 2.6842174530029297, 2.675023078918457,
2.666018009185791, 2.657200813293457, 2.6485555171966553,
2.6400771141052246, 2.6317408084869385, 2.623544692993164,
2.615511894226074, 2.607628583908081, 2.599869966506958,
2.5922176837921143, 2.5846939086914062, 2.577296257019043,
2.570037603378296, 2.5628952980041504, 2.5558645725250244]}

x_unlab = x_unlabeled_scaled
p_m = 0.3
alpha = 2.0
parameters = {'batch_size': 128,
              'epochs': 50,
              'learning_rate': 0.001,  # Add learning_rate to the
```

```
dictionary
              'hidden': 64            # Add hidden to the dictionary
              }

self_supervised(x_unlab,p_m,alpha, parameters)

Epoch 1/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 948ms/step - loss: 3.3020
Epoch 2/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 3.2506
Epoch 3/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 3.2140
Epoch 4/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step - loss: 3.1839
Epoch 5/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step - loss: 3.1577
Epoch 6/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - loss: 3.1343
Epoch 7/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 3.1126
Epoch 8/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 3.0925
Epoch 9/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 3.0734
Epoch 10/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step - loss: 3.0554
Epoch 11/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 3.0382
Epoch 12/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 3.0216
Epoch 13/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - loss: 3.0057
Epoch 14/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step - loss: 2.9902
Epoch 15/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - loss: 2.9752
Epoch 16/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - loss: 2.9607
Epoch 17/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - loss: 2.9465
Epoch 18/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - loss: 2.9327
Epoch 19/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.9192
Epoch 20/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - loss: 2.9059
Epoch 21/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.8930
Epoch 22/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.8803
```

```
Epoch 23/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.8679
Epoch 24/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.8558
Epoch 25/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 2.8438
Epoch 26/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 2.8321
Epoch 27/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - loss: 2.8206
Epoch 28/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.8094
Epoch 29/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.7983
Epoch 30/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.7874
Epoch 31/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - loss: 2.7767
Epoch 32/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.7662
Epoch 33/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.7559
Epoch 34/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step - loss: 2.7458
Epoch 35/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.7359
Epoch 36/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - loss: 2.7261
Epoch 37/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 2.7165
Epoch 38/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - loss: 2.7071
Epoch 39/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 44ms/step - loss: 2.6978
Epoch 40/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.6886
Epoch 41/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 2.6797
Epoch 42/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 2.6709
Epoch 43/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 2.6622
Epoch 44/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 2.6537
Epoch 45/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - loss: 2.6453
Epoch 46/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - loss: 2.6371
Epoch 47/50
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.6290
Epoch 48/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step - loss: 2.6210
Epoch 49/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.6132
Epoch 50/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 2.6055

(<Functional name=functional_5, built=True>,
 <Functional name=functional_4, built=True>,
 <keras.src.callbacks.history.History at 0x7b4f85e2a470>)
```

```python
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)

    # New mask label for the additional model fitting
    mask_label = corruption(corruption_binary_mask, x_unlabeled)

    # Training the model with corrupted data and mask label
    history = model.fit(
        x_unlabeled_corrupted,
        {'mask_estimation': mask_label, '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)
```

```python
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 ━━━━━━━━━━━━━━━━━━━━ 1s 998ms/step - loss: 3.2226
Epoch 2/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - loss: 3.1528
Epoch 3/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 3.1029
Epoch 4/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - loss: 3.0617
Epoch 5/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 3.0255
Epoch 6/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - loss: 2.9927
Epoch 7/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.9622
Epoch 8/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - loss: 2.9335
Epoch 9/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.9063
Epoch 10/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.8802
Epoch 11/50
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.8551
Epoch 12/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 2.8307
Epoch 13/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.8070
Epoch 14/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - loss: 2.7838
Epoch 15/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - loss: 2.7612
Epoch 16/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.7391
Epoch 17/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.7174
Epoch 18/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.6960
Epoch 19/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 2.6750
Epoch 20/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.6543
Epoch 21/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - loss: 2.6340
Epoch 22/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.6139
Epoch 23/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.5941
Epoch 24/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.5745
Epoch 25/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step - loss: 2.5551
Epoch 26/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.5360
Epoch 27/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step - loss: 2.5169
Epoch 28/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - loss: 2.4981
Epoch 29/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - loss: 2.4793
Epoch 30/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 2.4607
Epoch 31/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step - loss: 2.4423
Epoch 32/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 2.4240
Epoch 33/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 2.4058
Epoch 34/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.3879
Epoch 35/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.3701
```

```
Epoch 36/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - loss: 2.3525
Epoch 37/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 2.3350
Epoch 38/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 2.3177
Epoch 39/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - loss: 2.3004
Epoch 40/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 2.2834
Epoch 41/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - loss: 2.2664
Epoch 42/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step - loss: 2.2495
Epoch 43/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.2327
Epoch 44/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - loss: 2.2161
Epoch 45/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.1996
Epoch 46/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.1833
Epoch 47/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 2.1670
Epoch 48/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.1509
Epoch 49/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 2.1348
Epoch 50/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.1189

Model: "functional"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape                           ┃                        Param # ┃
┃ Connected to                    ┃                                        ┃                                ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
┃ input_layer (InputLayer)        ┃ (None, 37)                             ┃                              0 ┃
┃ -                               ┃                                        ┃                                ┃
┣━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┫
┃ dense (Dense)                   ┃ (None, 64)                             ┃                          2,432 ┃
┃ input_layer[0][0]               ┃                                        ┃                                ┃
┣━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┫
┃ mask_estimation (Dense)         ┃ (None, 37)                             ┃                          2,405 ┃
┃ dense[0][0]                     ┃                                        ┃                                ┃
┗━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┻━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┻━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┛
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ feature_estimation_dense[0][0]  ┃ (None, 37)                             ┃         2,405 ┃
┃ (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': [3.222633123397827, 3.1528258323669434,
3.1029114723205566, 3.0616893768310547, 3.0255210399627686,
2.9926841259002686, 2.9622225761413574, 2.9335362911224365,
2.906329870223999, 2.8802483081817627, 2.8550803661346436,
2.830705165863037, 2.806973934173584, 2.783829927444458,
2.761218309402466, 2.7390799522399902, 2.7173633575439453,
2.6960089206695557, 2.674996852874756, 2.6543076038360596,
2.63395619392395, 2.6139020919799805, 2.5940771102905273,
2.574489116668701, 2.5551273822784424, 2.5359578132629395,
2.5169436931610107, 2.498098611831665, 2.4793426990509033,
2.46071195602417, 2.442277431488037, 2.4239892959594727,
2.40584397315979, 2.3878679275512695, 2.370082139968872,
2.3524653911590576, 2.3350043296813965, 2.3176522254943848,
2.3004403114318848, 2.283350944519043, 2.266359567642212,
2.2494912147521973, 2.23274564743042, 2.2161221504211426,
2.1996233463287354, 2.1832637786865234, 2.1670212745666504,
2.15088152885437, 2.134843111038208, 2.1188976764678955]}
```

```python
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, p_m, x.shape)

# Updated corruption function as per the user's provided code
def corruption(mask, data):
    """
    Apply the mask to shuffle data across samples for corruption.
```

```
    Returns the new mask and the corrupted data.
    """
    num_samples, num_features = data.shape
    shuffled_data = np.zeros([num_samples, num_features])

    # Shuffle each feature across samples
    for feature_idx in range(num_features):
        shuffled_indices = np.random.permutation(num_samples)
        shuffled_data[:, feature_idx] = data[shuffled_indices,
feature_idx]

    # Create corrupted data based on the mask
    data_corrupted = data * (1 - mask) + shuffled_data * mask

    # Create new mask indicating which data points were corrupted
    mask_new = (data != data_corrupted).astype(int)

    return mask_new, data_corrupted

# 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')(h)

    # Output 2 --> feature estimation
    output2 = Dense(dimension, activation='sigmoid', name='feature')
(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': "binary_crossentropy", 'feature':
```

```
'mean_squared_error'},
        loss_weights={'mask': 1.0, 'feature': float(alpha)}
    )

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

    # Corrupt the unlabeled data and obtain new mask
    mask_label, x_unlabeled_corrupted =
corruption(corruption_binary_mask, x_unlabeled)

    # Training the model with corrupted data and mask label
    history = model.fit(
        x_unlabeled_corrupted,
        {'mask': mask_label, 'feature': 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 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)
```

```python
# Print the model summary
model.summary()

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

```
Epoch 1/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - loss: 3.3030
Epoch 2/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step - loss: 3.2488
Epoch 3/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step - loss: 3.2105
Epoch 4/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step - loss: 3.1793
Epoch 5/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - loss: 3.1523
Epoch 6/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step - loss: 3.1282
Epoch 7/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - loss: 3.1061
Epoch 8/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 3.0857
Epoch 9/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 3.0665
Epoch 10/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 3.0484
Epoch 11/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 130ms/step - loss: 3.0311
Epoch 12/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step - loss: 3.0146
Epoch 13/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - loss: 2.9988
Epoch 14/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 2.9836
Epoch 15/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step - loss: 2.9688
Epoch 16/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.9546
Epoch 17/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - loss: 2.9408
Epoch 18/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step - loss: 2.9273
Epoch 19/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 132ms/step - loss: 2.9143
Epoch 20/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 2.9015
Epoch 21/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - loss: 2.8891
Epoch 22/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.8770
```

```
Epoch 23/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 2.8652
Epoch 24/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.8537
Epoch 25/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 2.8424
Epoch 26/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - loss: 2.8314
Epoch 27/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step - loss: 2.8207
Epoch 28/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.8101
Epoch 29/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step - loss: 2.7998
Epoch 30/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 133ms/step - loss: 2.7897
Epoch 31/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step - loss: 2.7798
Epoch 32/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.7701
Epoch 33/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 2.7607
Epoch 34/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.7514
Epoch 35/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - loss: 2.7423
Epoch 36/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step - loss: 2.7335
Epoch 37/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - loss: 2.7248
Epoch 38/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 131ms/step - loss: 2.7163
Epoch 39/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - loss: 2.7080
Epoch 40/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.6998
Epoch 41/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.6918
Epoch 42/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step - loss: 2.6840
Epoch 43/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step - loss: 2.6763
Epoch 44/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step - loss: 2.6687
Epoch 45/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step - loss: 2.6613
Epoch 46/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - loss: 2.6540
Epoch 47/50
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 2.6469
Epoch 48/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 133ms/step - loss: 2.6399
Epoch 49/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step - loss: 2.6330
Epoch 50/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step - loss: 2.6263

Model: "functional_2"
```

Layer (type) Connected to	Output Shape	Param #
input_layer_1 (InputLayer) -	(None, 37)	0
dense_1 (Dense) input_layer_1[0][0]	(None, 64)	2,432
mask (Dense) dense_1[0][0]	(None, 37)	2,405
feature (Dense) dense_1[0][0]	(None, 37)	2,405

```
 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': [3.3030221462249756, 3.2487828731536865,
3.2104978561401367, 3.179304599761963, 3.1523351669311523,
3.128208875656128, 3.10614013671875, 3.0856823921203613,
3.066506862640381, 3.048368453979492, 3.031118392944336,
3.014636754989624, 2.9988183975219727, 2.983574867248535,
2.96884822845459, 2.954620122909546, 2.9407901763916016,
2.927337884902954, 2.914262294769287, 2.901540756225586,
2.889132499694824, 2.877002239227295, 2.8651914596557617,
```

```
2.853667974472046, 2.842421293258667, 2.831423759460449,
2.8206636905670166, 2.810091972351074, 2.7997756004333496,
2.7896735668182373, 2.779776096343994, 2.7701077461242676,
2.7606589794158936, 2.7514045238494873, 2.742347002029419,
2.733474016189575, 2.7247812747955322, 2.716279983520508,
2.707951307296753, 2.699798822402954, 2.6918160915374756,
2.683981418609619, 2.6762819290161133, 2.6687111854553223,
2.6612977981567383, 2.6540160179138184, 2.646883010864258,
2.6398801803588867, 2.63301944732666, 2.626258134841919]}
```

Save the Encoder Model

SIR'S CODE

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

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

# check how well logistic regression and xgboost works on the encoded
data

X_train_scaled_encoded = encoder.predict(X_train_scaled)
X_test_scaled_encoded = encoder.predict(X_test_scaled)

y_encoded = logistic(X_train_scaled_encoded, y_train,
X_test_scaled_encoded)

# compute log loss for y_encoded and y_test
print(log_loss(y_test, y_encoded))

# do the sam xgboost
y_encoded_xgb = xgboost(X_train_scaled_encoded, y_train,
X_test_scaled_encoded)

# compute log loss for y_encoded_xgb and y_test
print (log_loss(y_test,y_encpded_xgb))
```

.....

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

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

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

```
import numpy as np

# Load or create your training and testing data here
# Replace these with your actual data loading or creation steps
# Example using random data:
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=1000, n_features=37,
n_informative=10, n_classes=2, random_state=42)
x_train, x_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)


# Check the shape of x_train and X_test
print("x_train shape:", x_train.shape)
print("X_test shape:", x_test.shape)

# Check the expected input shape of the encoder model
print("Expected encoder model input shape:",
encoder_model.input_shape)

# Reshape or adjust features if necessary
required_features = encoder_model.input_shape[1]
if x_train.shape[1] != required_features or x_test.shape[1] !=
required_features:
    print(f"Adjusting feature count to match encoder model's expected
input shape ({required_features} features).")

    # If your data has fewer features, you need to apply appropriate
feature engineering
    # or consider retraining the encoder with the correct number of
features.
    # For demonstration, here's a simple approach to add dummy
features if needed:

    def adjust_features(data, required_features):
        current_features = data.shape[1]
        if current_features < required_features:
            additional_columns = required_features - current_features
            return np.hstack([data, np.zeros((data.shape[0],
additional_columns))])
        elif current_features > required_features:
            return data[:, :required_features]
        else:
            return data

    x_train = adjust_features(x_train, required_features)
    x_test = adjust_features(x_test, required_features)  # Adjust
x_test as well
```

```
# Encode the features
X_train_scaled_encoded = encoder_model.predict(x_train)
X_test_scaled_encoded = encoder_model.predict(x_test)

# Ensure no NaNs in encoded data
if np.isnan(X_train_scaled_encoded).any() or
np.isnan(X_test_scaled_encoded).any():
    print("Error: NaN values detected in encoded features. Please
check the preprocessing steps.")
else:
    # Logistic Regression Model
    logistic_model = LogisticRegression(max_iter=2000)
    logistic_model.fit(X_train_scaled_encoded, y_train)

    # Predict probabilities for the test set
    y_encoded = logistic_model.predict_proba(X_test_scaled_encoded)

    # Check if shapes match for log loss calculation
    if y_encoded.shape[1] == len(np.unique(y_test)):
        print("Logistic Regression Log Loss:", log_loss(y_test,
y_encoded))
    else:
        print(f"Error: Mismatched shapes between y_test and y_encoded.
y_test unique classes: {len(np.unique(y_test))}, y_encoded columns:
{y_encoded.shape[1]}")

    # XGBoost Model
    xgb_model = XGBClassifier(eval_metric='mlogloss')
    xgb_model.fit(X_train_scaled_encoded, y_train)
    y_encoded_xgb = xgb
```

```
x_train shape: (800, 37)
X_test shape: (200, 37)
Expected encoder model input shape: (None, 41)
Adjusting feature count to match encoder model's expected input shape
(41 features).
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
Error: NaN values detected in encoded features. Please check the
preprocessing steps.
```

SIR'S CODE

Semi supervised predict the labels for unlabeled inputs, model ,train and semi supervised

```
# purpose of the code we have written till now is to check how well
the model will perform on our encoded data
#next part semi supervised predict the labels for unlabeled inputs.
```

```
# model, train, semi supervised

def model(input_dimension, hidden_dimension, label_dimension,
activation = tf.nn.relu):
  #inputs takes input dimension as arguement
  Inputs = tf.keras.Input(shape = input_dimension, name='model_input')
  x = layers. Dense (hidden_dimension, activation activation, name =
'model dense_layer_1') (inputs) # dense layer 1
  x = layers.Dense(hidden_dimension, activation activation, name =
'model_dense_layer_2')(x) dense layer 2
  y_logit layers.Dense(label dimension, activation = None, name = '
model_logit_output')(x) # logit output
  y layers.Activation('softmax', name 'model_output') (y_logit) actual
prediction model = models. Model(inputs inputs, outputs = [y_logit,yl,
name+"model") # model creation
  return model

def train (feature_batch, label batch, unlabeled_feature_batch, model,
beta, supv_loss_fn, optimizer):
  with tf.GradientTape() as tape:
    y_logit, y = model(feature_batch, training = True) # getting
outputs for labeled data
    y_loss = supv_loss_fn(label_batch, y_logit) # calculating
supervised loss function for labeled data
    unlabeled_y_logit, unlabeled_y = model(unlabeled_feature_batch,
training = True) # getting outputs for unlabeled data
    unlabeled_y_loss =
tf.reduce.mean(tf.nn.moments(unlabeled_y_logit , axes = 0)[1]) # loss
function for unlabeled data
    # unsupervised loss function calculates the mean and variance of
the outputs and will penalize if the variance is high i.e, it will try
to
    # reduce the variance of the output
    total_loss = y_loss + beta * unlabeled_y_loss # loss formula. Beta
is a hyperparameter i.e, you have to enter your own value for this
  grads = tape.gradient(total_loss, model.trainable_weights) #
calculating gradiennts or by how much the weights need to be changed
  optimizer.apply_gradients(zip(grads, model.trainable_weights)) #
making the changes to the weights

  return total_loss

def semi_supervised(x_train,y_train, x_unlabaled, x_test, parameters,
mask_probability, K, beta):
  #parameters is a dictionary
  hidden_dimension = parameter ['hidden_dimension']
  act_fn = tf.nn.relu
  batch_size = parameters ['batch_size']
  epochs = parameters ['epochs']
  input_dimension=x_train.shape[1]
```

```
  label dimension=y_train.shape [1]
  if y_train.ndim 1 or y train.shape[1]==1:
    total_classes = len(np.unique(y_train)) # calculates the total
number of classes
    classes_dimension=total_classes
    class_mapping={ label idx for idx, label in
enumerate(unique_classes)} = maps each unique class label in y train
to an interger index
    y_train_mapped=np.vectorize(class_mapping.get) (y_train) # using
np.vectorize to apply class mapping function each element in y train
this
    #maps each original label to its integer index
    # for ex in our case If the classes are from 1-15 they will be
mapped to 8 14 where 1-0, 2-1, 3->2 and so on
  else:
    print("y_train is not of dimension. Please check your input
arguement for y_train")
    class_mapping = None
    classes_dimension y_train.shape[1]
    y_train_mapped = y_train

  #splitting the training data into training and validation sets
  index=np.random.permutation(x_train.shape[0])
  train_index=index: int(len(idx)*0.9)]
  valid_index=index [int(len(idx)*0.9):]

  #getting training sets
  splitted_train_x = x_train [train_index,:]
  splitted_train_y=y_train_mapped [train_index]

  #getting validation sets
  splitted_valid_x = x_train [valid_index,:]
  splitted_valid_y = y_train_mapped [valid_index]

  #data encoding
  encoder = call your encoder here
  x_valid_encoded encoder.predict(splitted_valid_x)
  x_test_encoded encoder.predict(x_test)

  # initialize the model
  Model model(input_dimension (encoder.output_shape (1),,
hidden_dimension hidden_dimension, label_dimension label_dimension)
  optimizer optimimizers.Adam()
  supv_loss_fn = tf.keras. losses.Categorical Crossentropy(from_logits
= True)

  #training loop
  for epoch in range(total_epochs):
    #sending a batch of labeled data
    batch_index = np.random.choice(len(x_train), batch_size,
```

```
replace=False)
    batch_x = x_train [batch_index]
    batch_y = y_train [batch_index]
    batch_x_encoded encoder.predict(batch_x)

    #sennding a batch on unlabeld data
    batch_unlabeled_index = np.random.choice(len(x_unlabeled),
batch_size, replace=False)
    batch_unlabeled_x = x_unlabeled [batch_unlabeled_index]
    batch_unlabled_x_shuffled = []
    for rep in range(K):
      mask_batch_unlabeled = mask_generator(mask_probability,
batch_unlabeled_x) # generate mask for batch of unabeled data
      unlabeled_shuffled_temp = corruption (mask_batch,
batch_unlabeled_x) # get shuffled batch data

unlabeled_shuffled_temp_encoded=encoder.predict(unlabeled_shuffled_tem
p) # encode the shuffled data

batch_unlabled_x_shuffled.append(unlabeled_shuffled_temp_encoded)
append the encoded data to the original unlabeled shuffled array

batch_unlabled_x_shuffled=np.concatenate(batch_unlabled_x_shuffled,
axis=0) # this line is outside the for loop
    #make sure you add the all the values to shuffled data

# purpose of the code we have written till now is to check how well
the model will perform on our encoded data
#next part semi supervised predict the labels for unlabeled inputs.
# model, train, semi supervised

def model(input_dimension, hidden_dimension, label_dimension,
activation = tf.nn.relu):
  #inputs takes input dimension as arguement
  Inputs = tf.keras.Input(shape = input_dimension, name='model_input')
  x = layers. Dense (hidden_dimension, activation activation, name =
'model dense_layer_1') (inputs) # dense layer 1
  x = layers.Dense(hidden_dimension, activation activation, name =
'model_dense_layer_2')(x) dense layer 2
  y_logit layers.Dense(label dimension, activation = None, name = '
model_logit_output')(x) # logit output
  y layers.Activation('softmax', name 'model_output') (y_logit) actual
prediction model = models. Model(inputs inputs, outputs = [y_logit,yl,
name+"model") # model creation
  return model

def train (feature_batch, label batch, unlabeled_feature_batch, model,
beta, supv_loss_fn, optimizer):
  with tf.GradientTape() as tape:
    y_logit, y = model(feature_batch, training = True) # getting
```

```
outputs for labeled data
    y_loss = supv_loss_fn(label_batch, y_logit) # calculating
supervised loss function for labeled data
    unlabeled_y_logit, unlabeled_y = model(unlabeled_feature_batch,
training = True) # getting outputs for unlabeled data
    unlabeled_y_loss =
tf.reduce.mean(tf.nn.moments(unlabeled_y_logit , axes = 0)[1]) # loss
function for unlabeled data
    # unsupervised loss function calculates the mean and variance of
the outputs and will penalize if the variance is high i.e, it will try
to
    # reduce the variance of the output
    total_loss = y_loss + beta * unlabeled_y_loss # loss formula. Beta
is a hyperparameter i.e, you have to enter your own value for this
  grads = tape.gradient(total_loss, model.trainable_weights) #
calculating gradiennts or by how much the weights need to be changed
  optimizer.apply_gradients(zip(grads, model.trainable_weights)) #
making the changes to the weights

  return total_loss

def semi_supervised(x_train, y_train, x_unlabeled, x_test, parameters,
mask_probability, K, beta):
    # parameters is a dictionary
    hidden_dimension = parameters['hidden_dimension']
    act_fn = tf.nn.relu
    batch_size = parameters['batch_size']
    epochs = parameters['epochs']
    input_dimension = x_train.shape[1]
    label_dimension = y_train.shape[1]

    if y_train.ndim == 1 or y_train.shape[1] == 1:
        total_classes = len(np.unique(y_train))  # calculates the
total number of classes

        classes_dimension = total_classes

        class_mapping = {label: idx for idx, label in
enumerate(unique_classes)}  # maps each unique class label in y_train
to an integer index
        y_train_mapped = np.vectorize(class_mapping.get)(y_train)  #
using np.vectorize to apply class_mapping function to each element in
y_train. This maps each original label to its integer index
        # for example, if the classes are from 1-15 ---> they will be
mapped to 0-14 where 1 -> 0, 2 -> 1, 3 -> 2 and so on

    else:
      print("y_train is not an dimension check the input argument")
      class_mapping = None
      classes_dimension = y_train.shape[1]
```

```
      y_train_mapped = y_train
   # Splitting the training data into training (90%) and validation
(10%) sets
    index = np.random.permutation(x_train.shape[0])  # Shuffle indices
    train_index = index[:int(len(index) * 0.9)]      # First 90% for
training
    valid_index = index[int(len(index) * 0.9):]      # Remaining 10%
for validation

# Creating the training and validation sets
    splitted_train_x = x_train[train_index, :]
    splitted_train_y = y_train_mapped[train_index]
    splitted_valid_x = x_train[valid_index, :]
    splitted_valid_y = y_train_mapped[valid_index]

# Display the shapes to verify the split
    print("Training set (X):", splitted_train_x.shape)
    print("Training set (y):", splitted_train_y.shape)
    print("Validation set (X):", splitted_valid_x.shape)
    print("Validation set (y):", splitted_valid_y.shape)
    # data encoding
    encoder = # call your encoder here

    x_valid_encoded = encoder.predict(splitted_valid_x)
    x_test_encoded = encoder.predict(x_test)

# initialize the model
    Model = model(input_dimension = (encoder.output_shape[1],),
hidden_dimension = hidden_dimension, label_dimension =
label_dimension)
    optimizer = optimizers.Adam()
    supv_loss_fn =
tf.keras.losses.CategoricalCrossentropy(from_logits=True)

# training loop
    for epoch in range(total_epochs):
    # sending a batch of labeled data
        batch_index = np.random.choice(len(x_train), batch_size,
replace=False)
        batch_x = x_train[batch_index]
        batch_y = y_train[batch_index]
        batch_x_encoded = encoder.predict(batch_x)

    #sennding a batch on unlabeld data
    batch_unlabeled_index = np.random.choice(len(x_unlabeled),
batch_size, replace=False)
    batch_unlabeled_x = x_unlabeled [batch_unlabeled_index]
    batch_unlabled_x_shuffled = []
    for rep in range(K):
      mask_batch_unlabeled = mask_generator(mask_probability,
```

```
batch_unlabeled_x) # generate mask for batch of unabeled data
      unlabeled_shuffled_temp = corruption (mask_batch,
batch_unlabeled_x) # get shuffled batch data

unlabeled_shuffled_temp_encoded=encoder.predict(unlabeled_shuffled_tem
p) # encode the shuffled data

batch_unlabled_x_shuffled.append(unlabeled_shuffled_temp_encoded)
append the encoded data to the original unlabeled shuffled array

batch_unlabled_x_shuffled=np.concatenate(batch_unlabled_x_shuffled,
axis=0) # this line is outside the for loop
    #make sure you add the all the values to shuffled data
```