In [1]:
```python
import math
import matplotlib as mpl
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from pprint import pprint
import re
import seaborn as sns
import time
import warnings

from sklearn.datasets import make_classification
from sklearn.decomposition import NMF
from sklearn.linear_model import LogisticRegression, LinearRegression
from sklearn.metrics import classification_report, make_scorer, accuracy_score, con
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import LabelEncoder
from sklearn.svm import SVC

from scipy.cluster.hierarchy import linkage, dendrogram
from pygam import LinearGAM
```

In [2]:
```python
data = pd.read_csv('https://raw.githubusercontent.com/obbrown1/Unsupervised_Final/m
```

Problem Statement

I am using the data set from the kaggle competition "ICR - Identifying Age-Related Conditions". My goal is to build a binary classifier to predict if a person "person has one or more of any of the three medical conditions (Class 1), or none of the three medical conditions (Class 0)." The training data consists of an Id column, a target column ('Class'), and 56 columns representating anonymized health characteristics. Due to the anonymous nature of the features, and the fact that I have no medical expertise, I will have to rely more heavily on unsupervised and domain-agnostic methods of analysis. I will create a number of different models, both supervised and unsupervised, and compare their performance, along the dimensions of training time, run time, and accuracy.

https://www.kaggle.com/competitions/icr-identify-age-related-conditions

EDA

Field Descriptions

- **Id**: Unique identifier for each observation.

- **AB-GL**: Fifty-six anonymized health characteristics. All are numeric except for EJ, which is categorical
- **Class**: A binary target: 1 indicates the subject has been diagnosed with one of the three conditions, 0 indicate

they have not.

EDA Procedure

- Inspect data.
- Check for missing or null values.
- Fix missing values by imputing column means.
- Encode EJ, transform letters to numbers.
- Normalize by column.
- Use hierarchical clustering to combine highly correlated columns.

In [3]:
```
data.info()
data.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 617 entries, 0 to 616
Data columns (total 58 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Id      617 non-null    object 
 1   AB      617 non-null    float64
 2   AF      617 non-null    float64
 3   AH      617 non-null    float64
 4   AM      617 non-null    float64
 5   AR      617 non-null    float64
 6   AX      617 non-null    float64
 7   AY      617 non-null    float64
 8   AZ      617 non-null    float64
 9   BC      617 non-null    float64
 10  BD      617 non-null    float64
 11  BN      617 non-null    float64
 12  BP      617 non-null    float64
 13  BQ      557 non-null    float64
 14  BR      617 non-null    float64
 15  BZ      617 non-null    float64
 16  CB      615 non-null    float64
 17  CC      614 non-null    float64
 18  CD      617 non-null    float64
 19  CF      617 non-null    float64
 20  CH      617 non-null    float64
 21  CL      617 non-null    float64
 22  CR      617 non-null    float64
 23  CS      617 non-null    float64
 24  CU      617 non-null    float64
 25  CW      617 non-null    float64
 26  DA      617 non-null    float64
 27  DE      617 non-null    float64
 28  DF      617 non-null    float64
 29  DH      617 non-null    float64
 30  DI      617 non-null    float64
 31  DL      617 non-null    float64
 32  DN      617 non-null    float64
 33  DU      616 non-null    float64
 34  DV      617 non-null    float64
 35  DY      617 non-null    float64
 36  EB      617 non-null    float64
 37  EE      617 non-null    float64
 38  EG      617 non-null    float64
 39  EH      617 non-null    float64
 40  EJ      617 non-null    object 
 41  EL      557 non-null    float64
 42  EP      617 non-null    float64
 43  EU      617 non-null    float64
 44  FC      616 non-null    float64
 45  FD      617 non-null    float64
 46  FE      617 non-null    float64
 47  FI      617 non-null    float64
 48  FL      616 non-null    float64
 49  FR      617 non-null    float64
 50  FS      615 non-null    float64
```

```
 51  GB       617 non-null    float64
 52  GE       617 non-null    float64
 53  GF       617 non-null    float64
 54  GH       617 non-null    float64
 55  GI       617 non-null    float64
 56  GL       616 non-null    float64
 57  Class    617 non-null    int64
dtypes: float64(55), int64(1), object(2)
memory usage: 279.7+ KB
```

Out[3]:

	Id	AB	AF	AH	AM	AR	AX	AY
0	000ff2bfdfe9	0.209377	3109.03329	85.200147	22.394407	8.138688	0.699861	0.025578
1	007255e47698	0.145282	978.76416	85.200147	36.968889	8.138688	3.632190	0.025578
2	013f2bd269f5	0.470030	2635.10654	85.200147	32.360553	8.138688	6.732840	0.025578
3	043ac50845d5	0.252107	3819.65177	120.201618	77.112203	8.138688	3.685344	0.025578
4	044fb8a146ec	0.380297	3733.04844	85.200147	14.103738	8.138688	3.942255	0.054810

5 rows × 58 columns

In [4]:

```
data = data.drop(['Id'], axis=1)

# Check for empty values
empty_columns = data.columns[data.isnull().all()]
print("Empty columns:", empty_columns)

# Check for null values
null_columns = data.columns[data.isnull().any()]
print("Null columns:", null_columns)

# Check for NaN values
nan_columns = data.columns[data.isna().any()]
print("NaN columns:", nan_columns)

# Check for missing values
missing_columns = data.columns[data.isnull().any() | data.isna().any()]
print("Missing value columns:", missing_columns)
```

```
Empty columns: Index([], dtype='object')
Null columns: Index(['BQ', 'CB', 'CC', 'DU', 'EL', 'FC', 'FL', 'FS', 'GL'], dtype='o
bject')
NaN columns: Index(['BQ', 'CB', 'CC', 'DU', 'EL', 'FC', 'FL', 'FS', 'GL'], dtype='ob
ject')
Missing value columns: Index(['BQ', 'CB', 'CC', 'DU', 'EL', 'FC', 'FL', 'FS', 'GL'],
dtype='object')
```

In [5]:

```
count = data.isna().any(axis=1).sum()
count / len(data)
```

Out[5]: 0.11183144246353323

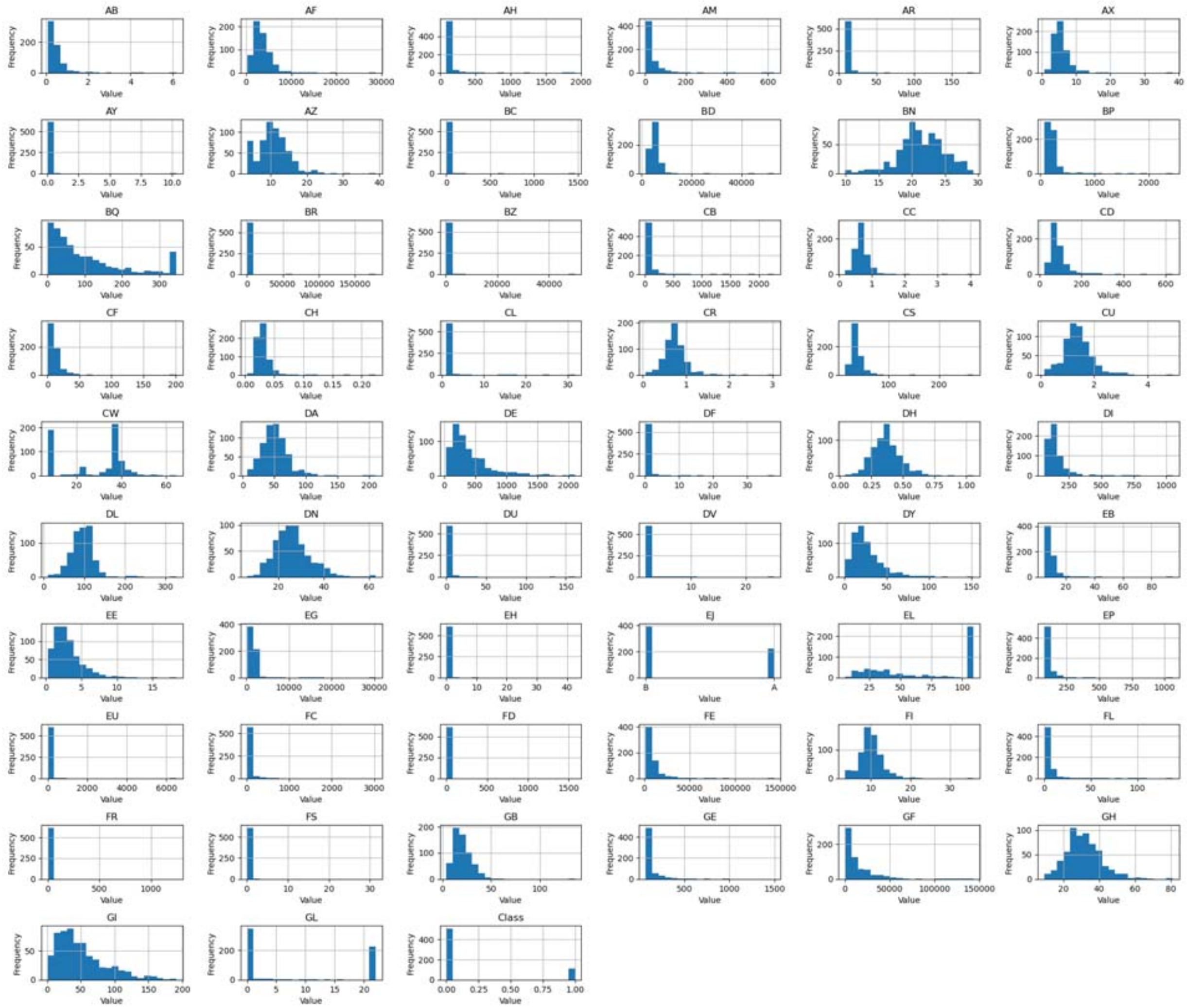
Missing Values

Rows with at least one missing value represent 11% of the total number of rows. Since this is already a small data set, instead of removing those rows, I will replace the missing values with the mean of their respective columns.

Data Visualization

In [6]:

```
fig, axes = plt.subplots(12, 6, figsize=(20, 20))
axes = axes.flatten()
num_cols = len(data.columns)

for i, column in enumerate(data.columns):
    ax = axes[i]
    data[column].hist(ax=ax, bins=20)
    ax.set_title(column)
    ax.set_xlabel('Value')
    ax.set_ylabel('Frequency')

if num_cols < len(axes):
    for j in range(num_cols, len(axes)):
        axes[j].set_visible(False)

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



Many of the features that appear to be very tightly clustered around a single value actually have extreme outliers that greatly expand the range of the data and therefore compress the bulk of it to one side of the distribution. I will replace the top and bottom 5% of values to correct with median values to correct for this.

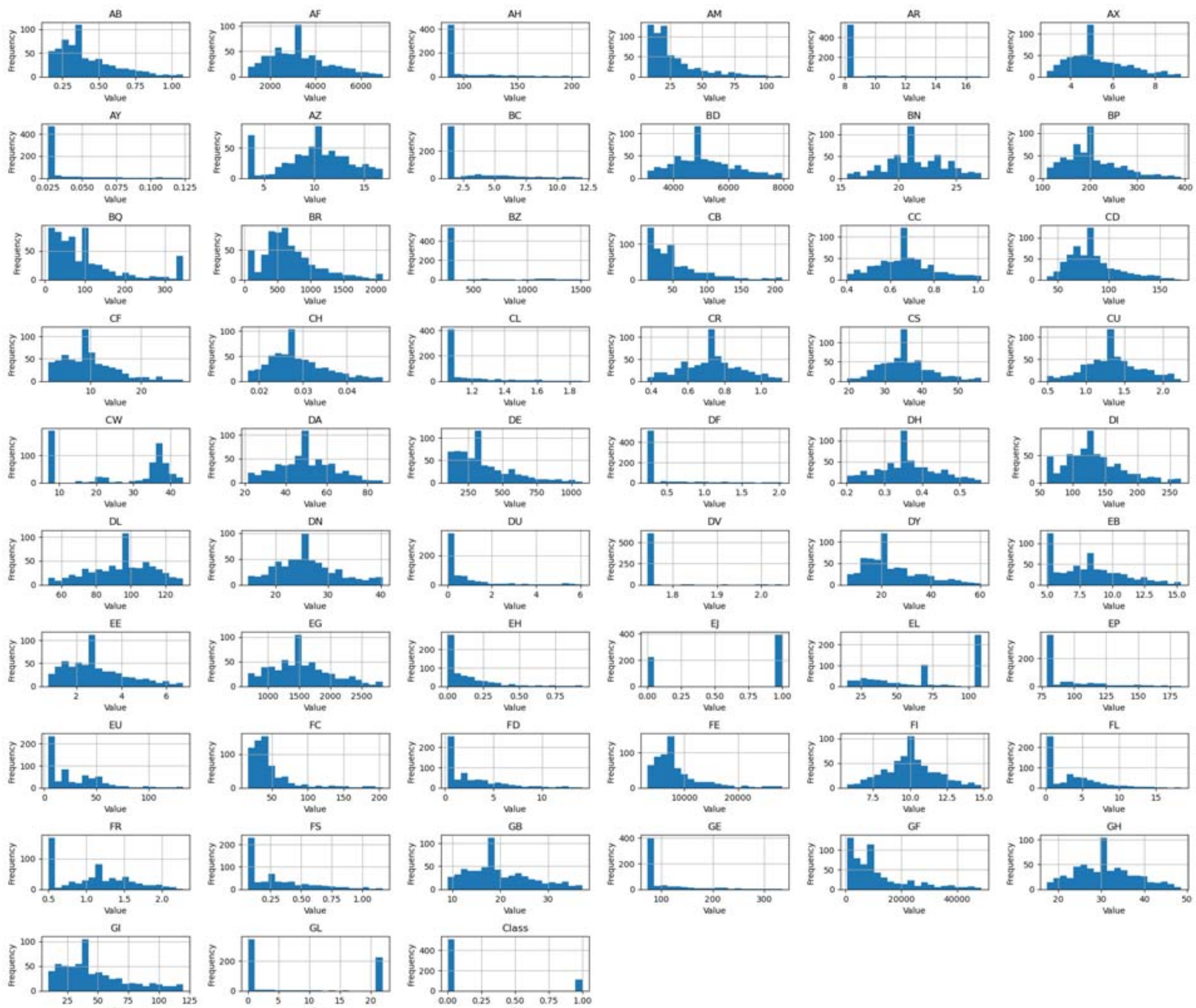
It may be the case that the outlying values are significant, but because the features are anonymized health measures, it is hard to say for certain whether they are significant or simply bad data.

In [7]:
```
le = LabelEncoder()
data['EJ'] = le.fit_transform(data['EJ'])
data = data.fillna(data.mean())
```

In [8]:
```
#replace outliers (top and bottom 5%) with column median
for column in data.columns:
    med = data[column].median()
    top = data[column].quantile(0.95)
    bottom = data[column].quantile(0.05)
    condition = (data[column] > top) | (data[column] < bottom)
    data.loc[condition, column] = med
```

In [9]:

```python
fig, axes = plt.subplots(12, 6, figsize=(20, 20))
axes = axes.flatten()
num_cols = len(data.columns)

for i, column in enumerate(data.columns):
    ax = axes[i]
    data[column].hist(ax=ax, bins=20)  # Adjust the number of bins as needed
    ax.set_title(column)
    ax.set_xlabel('Value')
    ax.set_ylabel('Frequency')

if num_cols < len(axes):
    for j in range(num_cols, len(axes)):
        axes[j].set_visible(False)

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



Data Labeling and Normalization

Columns have vastly different scales of values. For example, values in column 'GL' are on the order of 10^0, while values in column 'BD' can reach 10^4. In order to be able to compare them effectively without the columns with larger values swamping out those with smaller values, I will normalize the data.

In [10]:
```
min_max_df = pd.DataFrame({
                           'Minimum': data.min(),
                           'Maximum': data.max()})
print(min_max_df)
```

	Minimum	Maximum
AB	0.153828	1.076796
AF	1020.763260	6957.752890
AH	85.200147	209.852064
AM	7.166458	111.922483
AR	8.138688	17.027268
AX	2.870316	9.231078
AY	0.025578	0.123627
AZ	3.396778	16.845246
BC	1.229900	11.989768
BD	3042.040690	7954.676210
BN	15.536400	27.188700
BP	108.488727	392.658570
BQ	8.466250	344.644105
BR	51.216883	2095.573217
BZ	257.432377	1513.799323
CB	12.499760	210.133833
CC	0.402656	1.017413
CD	38.878912	171.197232
CF	1.651263	28.250890
CH	0.017512	0.048158
CL	1.050225	1.877675
CR	0.379650	1.113450
CS	19.516739	56.456274
CU	0.491013	2.239902
CW	7.030640	43.373768
DA	21.537880	87.429980
DE	101.257702	1078.791575
DF	0.238680	2.037906
DH	0.200876	0.557532
DI	60.232470	267.405360
DL	52.516640	129.943280
DN	14.760312	40.363960
DU	0.005518	6.076257
DV	1.743070	2.046100
DY	6.005804	60.644246
EB	4.926396	15.351696
EE	0.749766	6.735801
EG	683.012075	2835.627088
EH	0.003042	0.936936
EJ	0.000000	1.000000
EL	15.089217	109.125159
EP	78.526968	183.795001
EU	3.828384	132.899616
FC	16.005024	204.940848
FD	0.296850	14.141934
FE	3143.109390	28141.505950
FI	5.722494	14.852022
FL	0.173229	18.431146
FR	0.497060	2.276790
FS	0.067730	1.164956
GB	9.032242	37.077772
GE	72.611063	334.733511
GF	558.039294	48734.587490
GH	17.287766	48.618586
GI	9.364652	119.657156

```
GL         0.045692     21.978000
Class      0.000000      1.000000
```

In [11]:
```python
def normalize_df(df):
    for col in df.columns:
        df[col] = df[col] / np.max(df[col])
    return df

data_norm = normalize_df(data)
data_norm = data_norm.drop('Class', axis=1)
data_norm.head()
```

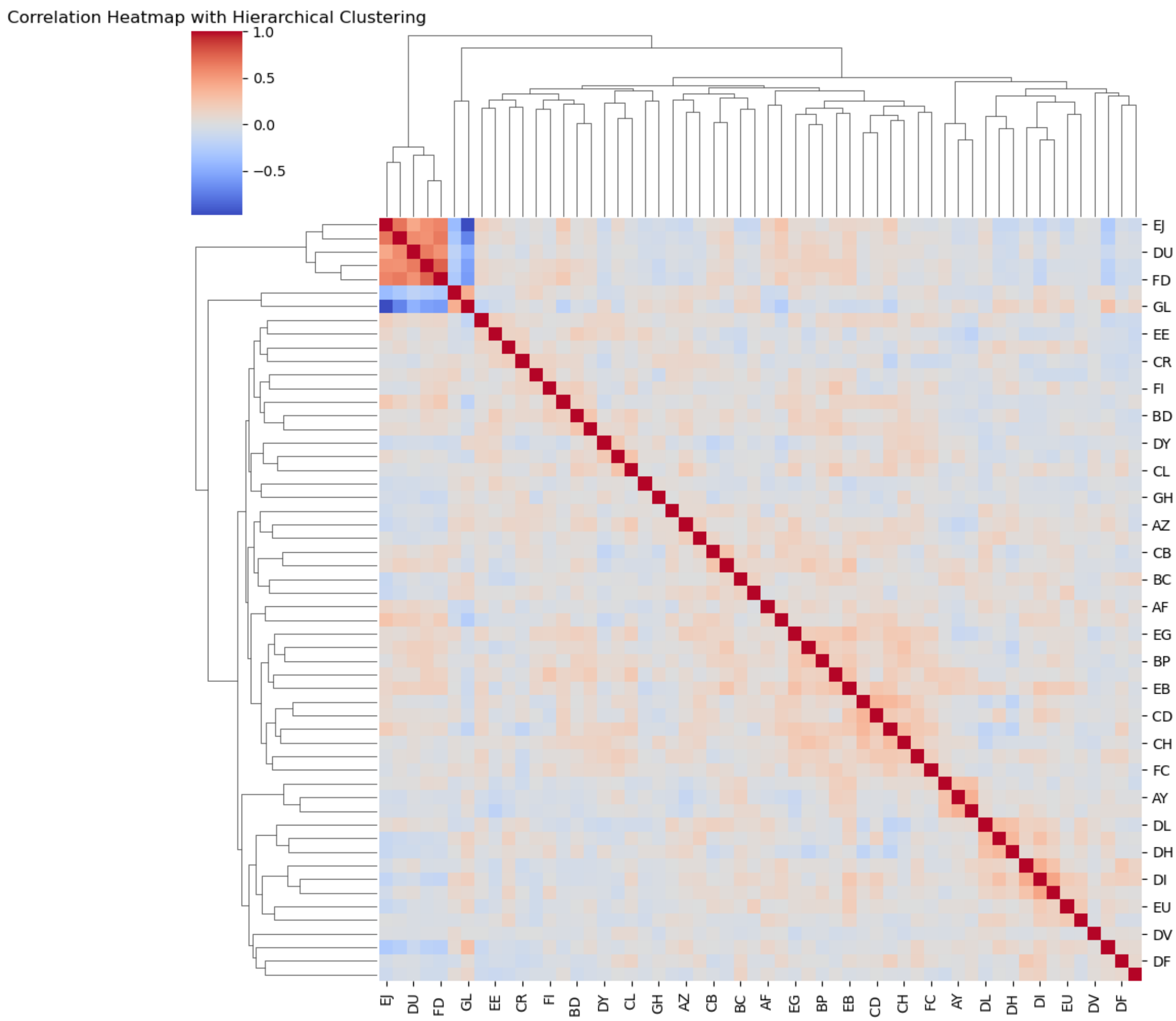
Out[11]:

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	0.194444	[illegible]	0.406001	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	0.406001	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.102579
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.102579
[illegible]	[illegible]	[illegible]	0.406001	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.102579

5 rows × 56 columns

In [12]:
```python
correlation_matrix = data_norm.corr()

# Create the hierarchical clustering heatmap
plt.figure(figsize=(10, 8))
heatmap = sns.clustermap(correlation_matrix, annot=False, cmap="coolwarm")
plt.title("Correlation Heatmap with Hierarchical Clustering")
plt.show()
```

```
<Figure size 1000x800 with 0 Axes>
```



Correlations

A group of roughly seven columns are highly correlated with eachother (positively and negatively). I will combine these, and any resulting columns, into a single resulting column to minimize the effects of multiple correlations.

In [13]:
```
#Combine correlated columns
corr_lim = 0.25
df_corr = data_norm.copy()
mcorr = 1
i = 0

while mcorr > corr_lim: # and i < 10:
    correlation_matrix = df_corr.corr()
    max_corr = np.abs(correlation_matrix.unstack().sort_values(ascending=False))
    column1, column2 = max_corr[max_corr != 1].index[0]
    mcorr = correlation_matrix.loc[column1, column2]
    #print("The two most correlated columns are:", column1, "and", column2)
    #print("Correlation coefficient:", mcorr, '\n')
    col_name = column1 + '_' + column2
    df_corr[col_name] = (np.abs(df_corr[column1]) + np.abs(df_corr[column2])) /2
    df_corr = df_corr.drop([column1, column2], axis=1)
```

```
    i = i + 1

print(df_corr.columns, '\n')
```

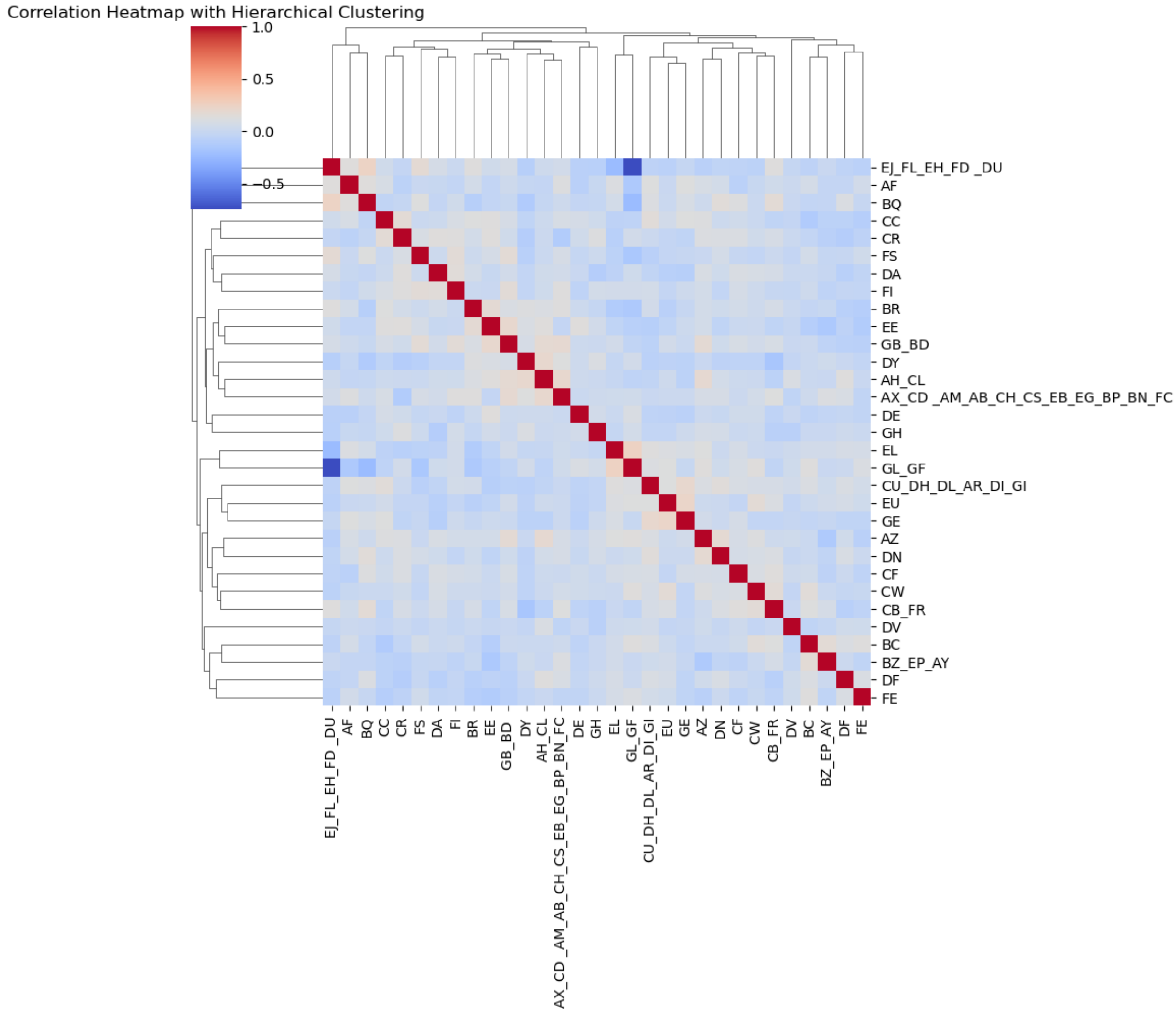
```
Index(['AF', 'AZ', 'BC', 'BQ', 'BR', 'CC', 'CF', 'CR', 'CW ', 'DA', 'DE', 'DF',
       'DN', 'DV', 'DY', 'EE', 'EL', 'EU', 'FE', 'FI', 'FS', 'GE', 'GH',
       'EJ_FL_EH_FD _DU', 'GL_GF', 'BZ_EP_AY', 'AH_CL', 'CB_FR',
       'AX_CD _AM_AB_CH_CS_EB_EG_BP_BN_FC', 'GB_BD ', 'CU_DH_DL_AR_DI_GI'],
      dtype='object')
```

Correlations pt. 2

The resulting correlations matrix shows that there is only a single pair of strongly correlated columns remaining, and correlations are down across the board. This should help reduce noise and allow the various models to more easily identify the salient features in the data.

In [14]:
```
correlation_matrix = df_corr.corr()

# Create the hierarchical clustering heatmap
plt.figure(figsize=(10, 8))
heatmap = sns.clustermap(correlation_matrix, annot=False, cmap="coolwarm")
plt.title("Correlation Heatmap with Hierarchical Clustering")
plt.show()
```

```
<Figure size 1000x800 with 0 Axes>
```



In [15]:
```
#Create train and test data sets from normalized data and data frame with combined
Xn = data_norm
yn = data['Class']
X_train_n, X_test_n, y_train_n, y_test_n = train_test_split(Xn, yn, test_size=0.2,

X = df_corr
y = data['Class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

Modeling - Non-negative Matrix Factorization (NMF)

The first model I will use will be non-negative matrix factorization with a logistic regression classifier. This technique should be helpful in reducing the highly-dimensioned dataset into smaller and more manageable factorized matrices. I will use hyperparameter tuning to identify the best parameters for this model. I suspect that the model will perform better on the data with combined correlated columns, but I will also test it on the normalized data as well.

In [16]:
```python
#hyperparameter tuning
def hyperparameter_tuning_NMF(X_train, y_train):
    warnings.filterwarnings("ignore")

    pipeline = Pipeline([
        ('nmf', NMF()),
        ('classifier', LogisticRegression(solver='lbfgs', max_iter=500))
    ])

    param_grid = {
        'nmf__n_components': [2,5,10],
        'nmf__solver': ['mu'],
        'nmf__beta_loss': ['frobenius', 'kullback-leibler'],
        'nmf__l1_ratio': [0, 0.5, 1]
    }

    scoring = make_scorer(accuracy_score)
    grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring=scoring)
    grid_search.fit(X_train, y_train)

    warnings.filterwarnings("default")

    return (grid_search.best_estimator_, grid_search.best_params_)
```

NMF - Combined Columns

In [17]:
```python
#create and fit model - combined correlated columns.
st = time.time()
(best_model, best_params) = hyperparameter_tuning_NMF(X_train, y_train)
best_params = {key: [value] for key, value in best_params.items()}
nmf_pst = time.time() - st
print('Parameter search time: ', nmf_pst)
```

```
Parameter search time:  7.195476770401001
```

In [18]:
```python
pd.DataFrame(best_params)
```

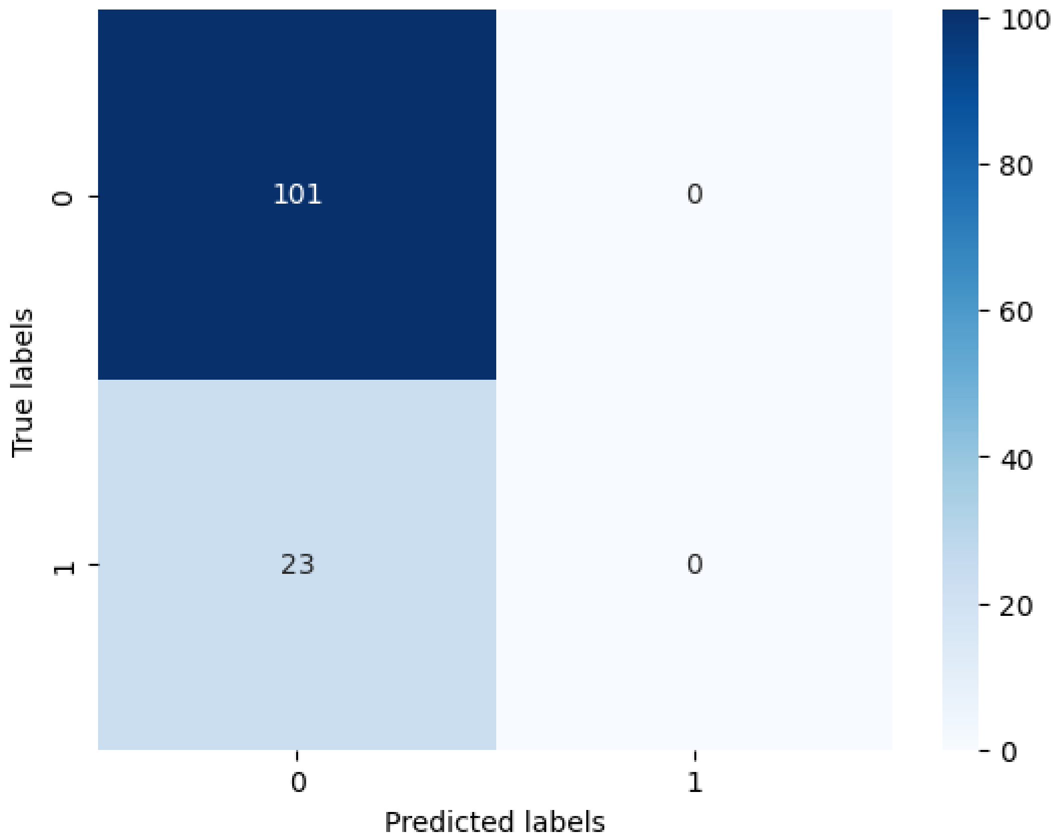
Out[18]:

	nmf__beta_loss	nmf__l1_ratio	nmf__n_components	nmf__solver
0	frobenius	0	2	mu

In [19]:
```python
st = time.time()
pred = best_model.predict(X_test)
nmf_pt = time.time() - st
nmf_acc = accuracy_score(y_test, pred)
print('NMF accuracy: ', nmf_acc)
print('Prediction time: ', nmf_pt)

ax = plt.subplot()
cm_nmf = confusion_matrix(y_test, pred)
sns.heatmap(cm_nmf, annot=True, cmap='Blues', fmt='d', xticklabels = ['0', '1'], yt
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
```

```
NMF accuracy:  0.8145161290322581
Prediction time:  0.007996082305908203
```

Out[19]: `Text(50.722222222222214, 0.5, 'True labels')`



NMF - Normalized

In [20]:
```
#create and fit model
st = time.time()
(best_model_n, best_params_n) = hyperparameter_tuning_NMF(X_train_n, y_train_n)
best_params_n = {key: [value] for key, value in best_params_n.items()}
nmf_pst2 = time.time() - st
print('Parameter search time: ', nmf_pst2)
```

```
Parameter search time:  9.594046831130981
```

In [21]:
```
pd.DataFrame(best_params_n)
```

Out[21]:

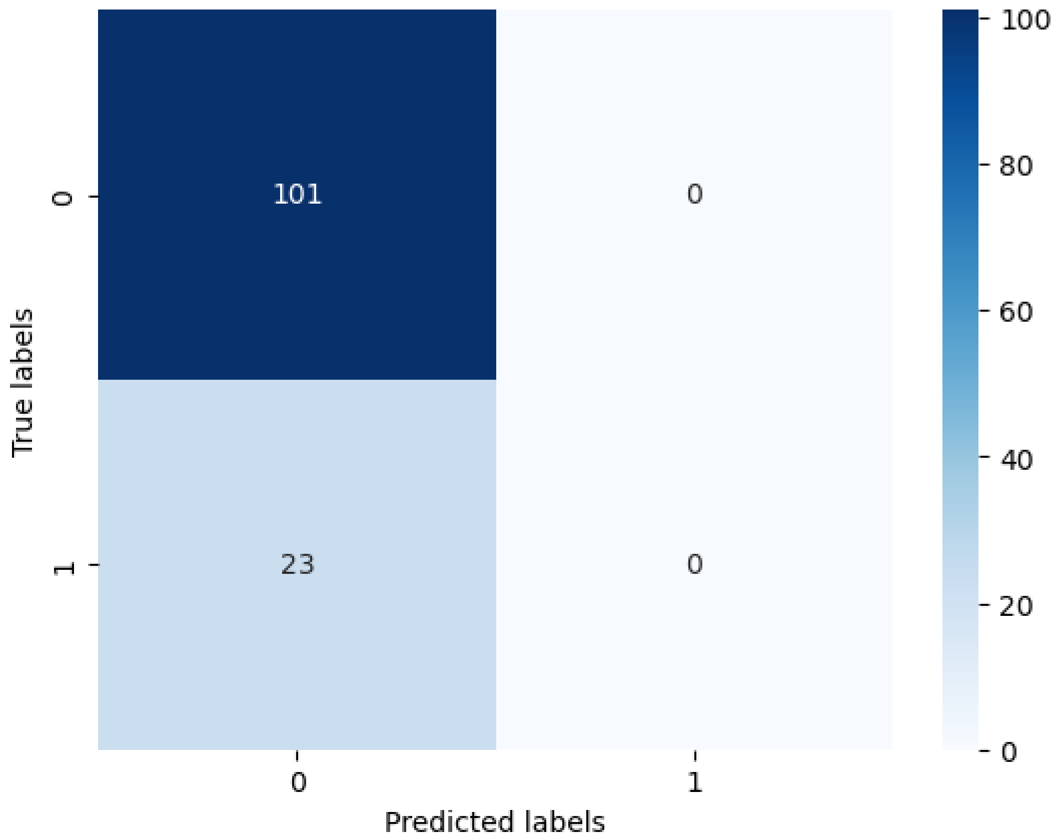
	nmf__beta_loss	nmf__l1_ratio	nmf__n_components	nmf__solver
0	frobenius	0	2	mu

In [22]:
```
st = time.time()
pred2 = best_model_n.predict(X_test_n)
nmf_pt2 = time.time() - st
nmf_acc2 = accuracy_score(y_test_n, pred2)
print('NMF accuracy: ', nmf_acc2)
print('Prediction time: ', nmf_pt2)
```

```
ax2 = plt.subplot()
cm_nmf = confusion_matrix(y_test_n, pred2)
sns.heatmap(cm_nmf, annot=True, cmap='Blues', fmt='d', xticklabels = ['0', '1'], yt
ax2.set_xlabel('Predicted labels')
ax2.set_ylabel('True labels')
```

```
NMF accuracy:  0.8145161290322581
Prediction time:  0.011999130249023438
```

Out[22]: `Text(50.722222222222214, 0.5, 'True labels')`



NMF - Discussion

The NMF model performed significantly better on the normalized data than it did on the combined-columns data. This indicates that combining the correlated columns did not do much to help the analysis, and may have instead introduced some distortions to make the model less accurate.

The model achieved a good degree of accuracy, 90%, and upon looking at the confusion matrices, we can see that it had a similar number of type I and type II errors.

Modeling - Multilayer Perceptron (MLP)

The next model I will use will be a multilayer perceptron (MLP). This is an unsupervised learning method that creates a network of artificial neurons arranged in layers that take in information from the previous layer, apply weights and an activation function to that input, and then send their output to the next layer. This model may be well suited to the data because unsupervised algorithms have an advantage with unsupervised features. I cannot rely on domain knowledge to help me engineer features, so I must instead rely on the algorithm to detect meaningul patterns in the data.

That being said, there is also an increased danger of overfitting with an MLP, as this dataset is relatively small and the hyperparameters could be tuned very finely until the desired result is achieved. Overfitting the model on the training data would make it less accurate when used on different datasets.

I will train and fit an MLP model on both the normalized and combined-columns data, as with the NMF models.

MLP - Combined Columns

In [23]:
```
mlp = MLPClassifier()
warnings.filterwarnings("ignore")

param_grid = {
    'hidden_layer_sizes': [(50,), (100,), (50, 50), (100, 50)],
    'activation': ['relu', 'tanh'],
    'solver': ['sgd', 'adam'],
    'learning_rate': ['constant', 'adaptive'],
}

grid_search = GridSearchCV(mlp, param_grid, cv=5)
st = time.time()
grid_search.fit(X_train, y_train)
mlp_pst = time.time() - st
print('Parameter search time: ', mlp_pst)
```

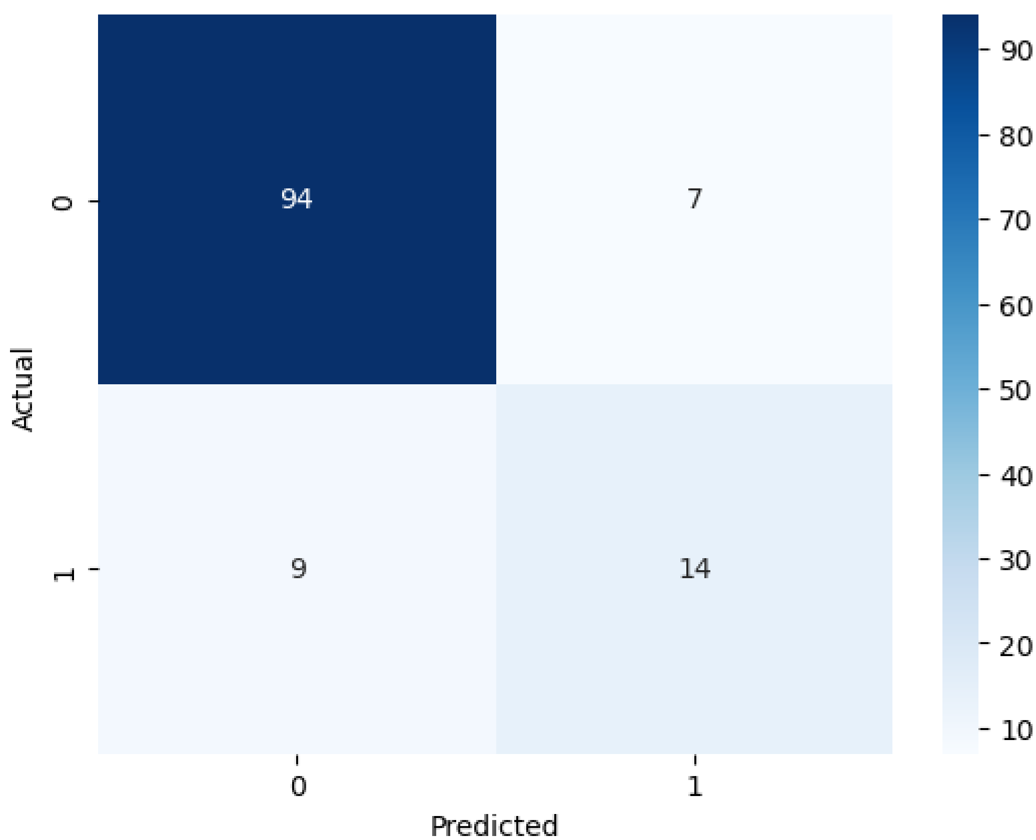
```
Parameter search time:  74.97149324417114
```

In [24]:
```
# Get the best parameters and score
best_params = grid_search.best_params_
best_score = grid_search.best_score_

print("Best Parameters:", best_params)
print("Best Score:", best_score)

# Evaluate the best model on the test data
best_model = grid_search.best_estimator_
st = time.time()
y_pred = best_model.predict(X_test)
mlp_pt = time.time() - st

print("Classification Report:")
c = classification_report(y_test, y_pred)
print(c)
mlp_acc = float(classification_report(y_test, y_pred, output_dict=True)['accuracy']

warnings.filterwarnings("default")
```

```
Best Parameters: {'activation': 'relu', 'hidden_layer_sizes': (50, 50), 'learning_ra
te': 'constant', 'solver': 'adam'}
Best Score: 0.8682539682539682
Classification Report:
              precision    recall  f1-score   support

         0.0       0.91      0.93      0.92       101
         1.0       0.67      0.61      0.64        23

    accuracy                           0.87       124
   macro avg       0.79      0.77      0.78       124
weighted avg       0.87      0.87      0.87       124
```

In [25]:

```python
# Create a confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Display the confusion matrix using a heatmap
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```



MLP - Normalized

In [26]:

```python
mlp_n = MLPClassifier()
warnings.filterwarnings("ignore")

# Define the parameter grid for hyperparameter tuning
```

```
param_grid = {
    'hidden_layer_sizes': [(50,), (100,), (50, 50), (100, 50)],
    'activation': ['relu', 'tanh'],
    'solver': ['sgd', 'adam'],
    'learning_rate': ['constant', 'adaptive'],
}

# Perform grid search using cross-validation
grid_search_n = GridSearchCV(mlp_n, param_grid, cv=5)
st = time.time()
# Fit the grid search to the training data
grid_search_n.fit(X_train_n, y_train_n)
mlp_pst2 = time.time() - st
print('MLP parameter search time: ', mlp_pst2)
```

```
MLP parameter search time:  85.22196817398071
```

In [27]:

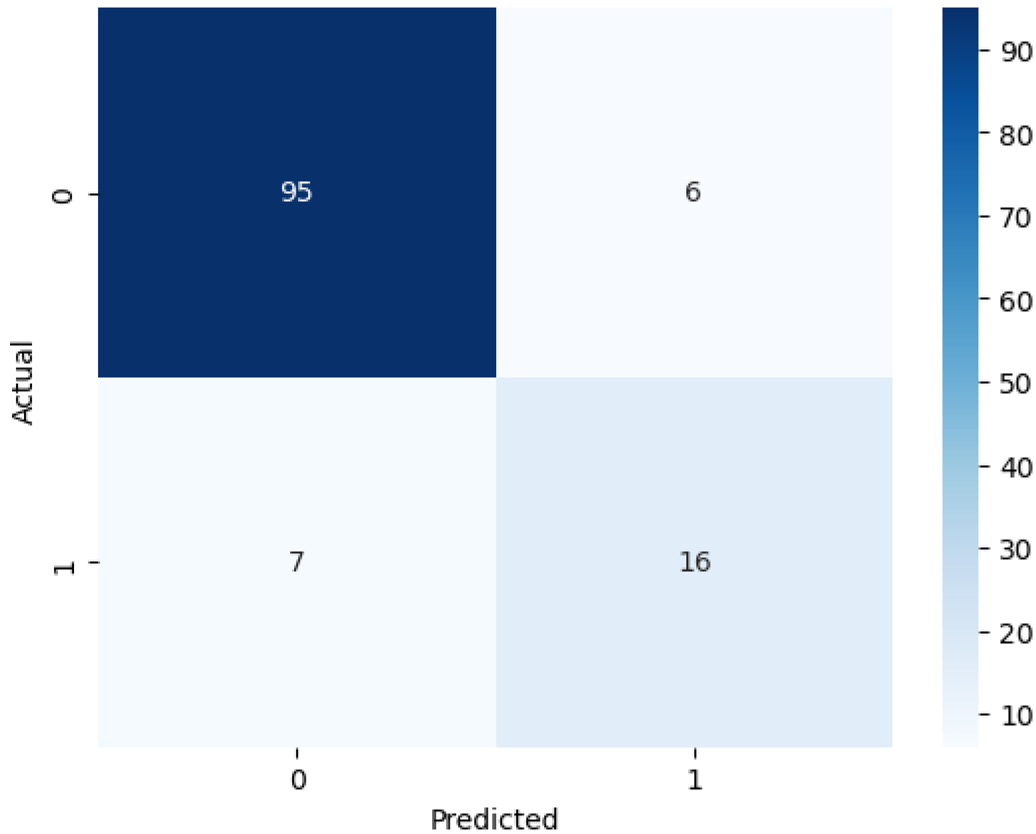
```
# Get the best parameters and score
best_params_n = grid_search_n.best_params_
best_score_n = grid_search_n.best_score_

print("Best Parameters:", best_params_n)
print("Best Score:", best_score_n)

# Evaluate the best model on the test data
best_model_n = grid_search_n.best_estimator_
st = time.time()
y_pred_n = best_model_n.predict(X_test_n)
mlp_pt2 = time.time() - st

print("Classification Report:")
c2 = classification_report(y_test_n, y_pred_n)
print(c2)
mlp_acc2 = float(classification_report(y_test_n, y_pred_n, output_dict=True)['accur

warnings.filterwarnings("default")
```

```
Best Parameters: {'activation': 'relu', 'hidden_layer_sizes': (100,), 'learning_rat
e': 'constant', 'solver': 'adam'}
Best Score: 0.8782106782106782
Classification Report:
              precision    recall  f1-score   support

         0.0       0.93      0.94      0.94       101
         1.0       0.73      0.70      0.71        23

    accuracy                           0.90       124
   macro avg       0.83      0.82      0.82       124
weighted avg       0.89      0.90      0.89       124
```

In [28]:

```
# Create a confusion matrix
cm = confusion_matrix(y_test_n, y_pred_n)

# Display the confusion matrix using a heatmap
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel('Predicted')
```

```
plt.ylabel('Actual')
plt.show()
```



MLP - Discussion

As with the NMF model, the MLP model performed better on the normalized than on the combined-columns data. This is further evidence that combining the correlated columns did not do much to help the analysis.

The model achieved a degree of accuracy similar to that of the NMF model, classifying 90% of the data points correctly, and had a similar distribution of type I and type II errors.

Modeling - Linear Models

Finally, I will test a linear model and generalised additive model on the combined columns and normalized data. I predict that these will perform more poorly then both the NMF and the MLP models, as it is unlikely that the 'Class' target variable is linearly dependent on any combination of the feature columns. However, these models will be useful as a comparison to the more advanced models, and serve as a benchmark to see how much the other, more sophisticated models improve over this baseline.

Modeling - Linear Regression

In [29]:
```
model = LinearRegression()

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

# Make predictions on the test set
y_pred = model.predict(X_test)
```

In [30]:
```
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared Score:", r2)
```

```
Mean Squared Error: 0.12281989919345072
R-squared Score: 0.18705175635019455
```

In [31]:
```
model = LinearRegression()

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

# Make predictions on the test set
y_pred = model.predict(X_test)
```

In [32]:
```
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared Score:", r2)
```

```
Mean Squared Error: 0.12281989919345072
R-squared Score: 0.18705175635019455
```

Modeling - GAM

In [33]:
```
model = LinearGAM().fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Calculate the mean squared error
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared Score:", r2)
```

```
Mean Squared Error: 0.15354219970058333
R-squared Score: -0.016299983898479864
```

In [34]:
```
results = pd.DataFrame({'Model': ['NMF - Comb', 'NMF - Norm', 'MLP - Comb', 'MLP -
                        'Training Time': [nmf_pst, nmf_pst2, mlp_pst, mlp_pst2],
```

```
                     'Prediction Time': [nmf_pt, nmf_pt2, mlp_pt, mlp_pt2],
                     'Accuracy': [nmf_acc, nmf_acc2, mlp_acc, mlp_acc2]
                    })

results
```

Out[34]:

	Model	Training Time	Prediction Time	Accuracy
0	NMF - Comb	7.195477	0.007996	0.814516
1	NMF - Norm	9.594047	0.011999	0.814516
2	MLP - Comb	74.971493	0.009003	0.870968
3	MLP - Norm	85.221968	0.013598	0.895161

In [ ]:

Discussion

The NMF model and the MLP model performed equally well on both the combined-column and normalized data. The MLP models had significantly longer training times, although that is due in part to having more hyperparameter combinations to consider. This was not a problem on this relatively small dataset, but with larger datasets the MLP training time could become prohibitive, and lead to less chance of finding the ideal hyperparameters.

Both models had fast, essentially negligible prediction times. This means that both could be used in scenarios in which real-time predictions were required.

Finally, while the models can't be compared directly, it is clear that both the NMF and MLP models performed better, and explained more of the variance in the data, than either the linear regression model or the GAM model. A small upfront investment in training time lead to significantly better results.

Data files and this notebook can be found here:
https://github.com/obbrown1/Unsupervised_Final