In [22]:
```
import pandas as pd
import numpy as np
import sklearn
import matplotlib.pyplot as plt
```

In [23]:
```
data = pd.read_csv('Thyroid_Diff.csv')
```

In [24]:
```
data.head()
```

Out[24]:

	Age	Gender	Smoking	Hx Smoking	Hx Radiothreapy	Thyroid Function	Physical Examination	Adenopathy	Patholog
0	27	F	No	No	No	Euthyroid	Single nodular goiter-left	No	Micropapillar
1	34	F	No	Yes	No	Euthyroid	Multinodular goiter	No	Micropapillar
2	30	F	No	No	No	Euthyroid	Single nodular goiter-right	No	Micropapillar
3	62	F	No	No	No	Euthyroid	Single nodular goiter-right	No	Micropapillar
4	62	F	No	No	No	Euthyroid	Multinodular goiter	No	Micropapillar

Data Preprocessing

Data Cleaning

In [25]:
```
data.info()
#No null values found
#Preprocess object into integer/float
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 383 entries, 0 to 382
Data columns (total 17 columns):
 #   Column               Non-Null Count  Dtype 
---  ------               --------------  ----- 
 0   Age                  383 non-null    int64 
 1   Gender               383 non-null    object
 2   Smoking              383 non-null    object
 3   Hx Smoking           383 non-null    object
 4   Hx Radiothreapy      383 non-null    object
 5   Thyroid Function     383 non-null    object
 6   Physical Examination 383 non-null    object
 7   Adenopathy           383 non-null    object
 8   Pathology            383 non-null    object
 9   Focality             383 non-null    object
 10  Risk                 383 non-null    object
 11  T                    383 non-null    object
 12  N                    383 non-null    object
 13  M                    383 non-null    object
 14  Stage                383 non-null    object
 15  Response             383 non-null    object
 16  Recurred             383 non-null    object
dtypes: int64(1), object(16)
memory usage: 51.0+ KB
```

In [26]:

```
data.isnull().values.any()
```

Out[26]:

```
False
```

In [27]:

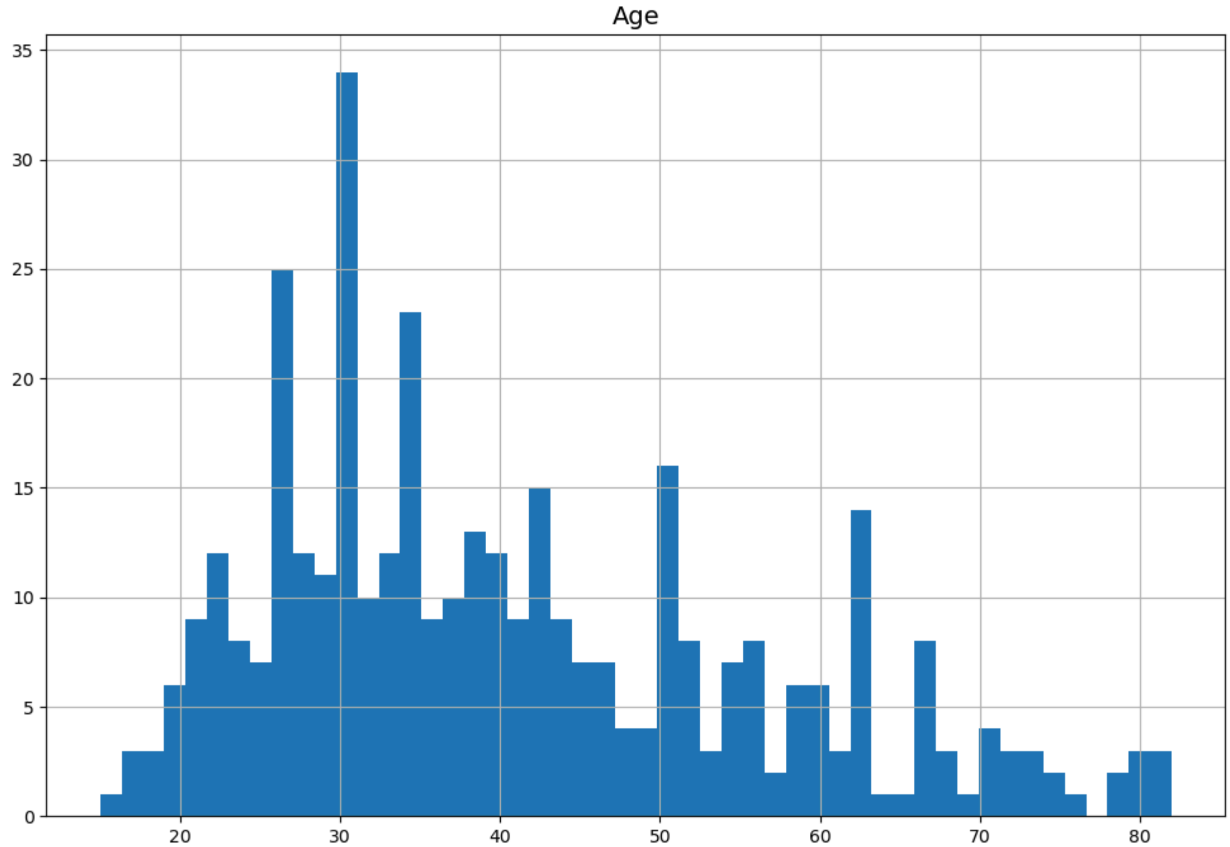
```
data.describe().T
```

Out[27]:

	count	mean	std	min	25%	50%	75%	max
Age	383.0	40.866841	15.134494	15.0	29.0	37.0	51.0	82.0

In [28]:

```
# extra code – the next 5 lines define the default font sizes
plt.rc('font', size=14)
plt.rc('axes', labelsize=14, titlesize=14)
plt.rc('legend', fontsize=14)
plt.rc('xtick', labelsize=10)
plt.rc('ytick', labelsize=10)

data.hist(bins=50, figsize=(12,8))
plt.show()
```



In [29]:
```
# Replace values
data['Recurred'] = data['Recurred'].replace({'Yes': '1', 'No': '0'})

# Write the modified DataFrame back to CSV
data.to_csv('modified_thyroid.csv', index=False)
```

In [30]:
```
df = pd.read_csv('modified_thyroid.csv')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 383 entries, 0 to 382
Data columns (total 17 columns):
 #   Column                Non-Null Count  Dtype 
---  ------                --------------  ----- 
 0   Age                   383 non-null    int64 
 1   Gender                383 non-null    object
 2   Smoking               383 non-null    object
 3   Hx Smoking            383 non-null    object
 4   Hx Radiothreapy       383 non-null    object
 5   Thyroid Function      383 non-null    object
 6   Physical Examination  383 non-null    object
 7   Adenopathy            383 non-null    object
 8   Pathology             383 non-null    object
 9   Focality              383 non-null    object
 10  Risk                  383 non-null    object
 11  T                     383 non-null    object
 12  N                     383 non-null    object
 13  M                     383 non-null    object
 14  Stage                 383 non-null    object
 15  Response              383 non-null    object
 16  Recurred              383 non-null    int64 
dtypes: int64(2), object(15)
memory usage: 51.0+ KB
```

In [31]:
```
recurrence= df.drop('Recurred', axis=1)
recurrence_labels = df[['Recurred']].copy()
```

In [32]:
```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='median') #impute using the median value
```

In [33]:
```
#Select the numerical varibles
recurrence_num = recurrence.select_dtypes(include=[np.number])
```

In [34]:
```
imputer.fit(recurrence_num)
```

Out[34]:
```
▾ SimpleImputer
SimpleImputer(strategy='median')
```

In [35]:
```
#show the median values for each variable
imputer.statistics_
```

Out[35]:
```
array([37.])
```

In [36]:
```
X = imputer.transform(recurrence_num)
```

In [37]:
```
imputer.feature_names_in_
```

Out[37]:
```
array(['Age'], dtype=object)
```

In [38]:
```
from sklearn import set_config

set_config(transform_output='pandas') #scikit-learn >= 1.2
```

In [39]:
```
#Dropping outliers
from sklearn.ensemble import IsolationForest

isolation_forest = IsolationForest(random_state=42)
outlier_pred = isolation_forest.fit_predict(X)
```

In [40]:
```
pd.Series(outlier_pred).value_counts()
```

Out[40]:
```
 1    232
-1    151
Name: count, dtype: int64
```

Encode Features

In [41]:
```
recurrence_cat = recurrence[['Gender', 'Smoking', 'Hx Smoking', 'Hx Radiothreapy', 'Th
                             'Physical Examination', 'Adenopathy', 'Pathology', 'Foca
                             'Stage', 'Response']]
recurrence_cat.head()
```

Out[41]:

	Gender	Smoking	Hx Smoking	Hx Radiothreapy	Thyroid Function	Physical Examination	Adenopathy	Pathology	Fo
0	F	No	No	No	Euthyroid	Single nodular goiter-left	No	Micropapillary	
1	F	No	Yes	No	Euthyroid	Multinodular goiter	No	Micropapillary	
2	F	No	No	No	Euthyroid	Single nodular goiter-right	No	Micropapillary	
3	F	No	No	No	Euthyroid	Single nodular goiter-right	No	Micropapillary	
4	F	No	No	No	Euthyroid	Multinodular goiter	No	Micropapillary	N

In [42]:
```
recurrence_cat_oneHot = recurrence[['Thyroid Function', 'Physical Examination', 'Patho
recurrence_cat_ordinal = recurrence[['Gender', 'Smoking', 'Hx Smoking', 'Hx Radiothrea
```

In [43]:
```
from sklearn.preprocessing import OneHotEncoder, LabelEncoder

cat_encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
recurrence_cat_1hot = cat_encoder.fit_transform(recurrence_cat_oneHot)
recurrence_cat_1hot
```

Out[43]:

	Thyroid Function_Clinical Hyperthyroidism	Thyroid Function_Clinical Hypothyroidism	Thyroid Function_Euthyroid	Thyroid Function_Subclinical Hyperthyroidism	Thyroid Function_Subclinical Hypothyroidism
0	0.0	0.0	1.0	0.0	0.0
1	0.0	0.0	1.0	0.0	0.0
2	0.0	0.0	1.0	0.0	0.0
3	0.0	0.0	1.0	0.0	0.0
4	0.0	0.0	1.0	0.0	0.0
...	...	...	...	...	...
378	0.0	0.0	1.0	0.0	0.0
379	0.0	0.0	1.0	0.0	0.0
380	0.0	0.0	1.0	0.0	0.0
381	1.0	0.0	0.0	0.0	0.0
382	0.0	0.0	1.0	0.0	0.0

383 rows × 46 columns

In [44]:

```
recurrence_cat_ordinal
```

Out[44]:

	Gender	Smoking	Hx Smoking	Hx Radiothreapy
0	F	No	No	No
1	F	No	Yes	No
2	F	No	No	No
3	F	No	No	No
4	F	No	No	No
...	...	...	...	...
378	M	Yes	Yes	Yes
379	M	Yes	No	Yes
380	M	Yes	Yes	No
381	M	Yes	Yes	Yes
382	M	Yes	No	No

383 rows × 4 columns

Encode Target

In [45]:

```
recurrence_tar_ordinal = df[['Recurred']]
```

In [46]:
```
recurrence_tar_ordinal
```

Out[46]:

	Recurred
0	0
1	0
2	0
3	0
4	0
...	...
378	1
379	1
380	1
381	1
382	1

383 rows × 1 columns

In [47]:
```
cat_encoder.categories_
```

Out[47]:
```
[array(['Clinical Hyperthyroidism', 'Clinical Hypothyroidism', 'Euthyroid',
        'Subclinical Hyperthyroidism', 'Subclinical Hypothyroidism'],
       dtype=object),
 array(['Diffuse goiter', 'Multinodular goiter', 'Normal',
        'Single nodular goiter-left', 'Single nodular goiter-right'],
       dtype=object),
 array(['Follicular', 'Hurthel cell', 'Micropapillary', 'Papillary'],
       dtype=object),
 array(['Multi-Focal', 'Uni-Focal'], dtype=object),
 array(['High', 'Intermediate', 'Low'], dtype=object),
 array(['T1a', 'T1b', 'T2', 'T3a', 'T3b', 'T4a', 'T4b'], dtype=object),
 array(['N0', 'N1a', 'N1b'], dtype=object),
 array(['M0', 'M1'], dtype=object),
 array(['I', 'II', 'III', 'IVA', 'IVB'], dtype=object),
 array(['Biochemical Incomplete', 'Excellent', 'Indeterminate',
        'Structural Incomplete'], dtype=object),
 array(['Bilateral', 'Extensive', 'Left', 'No', 'Posterior', 'Right'],
       dtype=object)]
```

Feature Scaling

In [48]:
```
from sklearn.preprocessing import StandardScaler
std_scaler = StandardScaler()
recurrence_scaled = std_scaler.fit_transform(recurrence_num)
```

In [49]:
```
recurrence_num.columns
```

Out[49]:
```
Index(['Age'], dtype='object')
```

Transformation Pipeline

In [50]:
```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy='median')),
    ('standardize', StandardScaler())
])
```

In [51]:
```
set_config(display='diagram')
num_pipeline
```

Out[51]:

Pipeline

SimpleImputer

StandardScaler

In [52]:
```
recurrence_prepared = num_pipeline.fit_transform(recurrence_num)
recurrence_prepared.head()
```

Out[52]:

	Age
0	-0.917439
1	-0.454315
2	-0.718957
3	1.398184
4	1.398184

In [53]:
```
recurrence_prepared.describe()
```

Out[53]:

	Age
count	3.830000e+02
mean	-1.855203e-17
std	1.001308e+00
min	-1.711367e+00
25%	-7.851180e-01
50%	-2.558327e-01
75%	6.704165e-01
max	2.721397e+00

In [54]:
```
recurrence_cat_oneHot.columns
```

Out[54]:
```
Index(['Thyroid Function', 'Physical Examination', 'Pathology', 'Focality',
       'Risk', 'T', 'N', 'M', 'Stage', 'Response', 'Adenopathy'],
      dtype='object')
```

In [55]:
```
recurrence_cat_ordinal.columns
```

Out[55]:
```
Index(['Gender', 'Smoking', 'Hx Smoking', 'Hx Radiothreapy'], dtype='object')
```

In [56]:
```
recurrence_tar_ordinal.columns
```

Out[56]:
```
Index(['Recurred'], dtype='object')
```

In [57]:
```
recurrence_num.columns
```

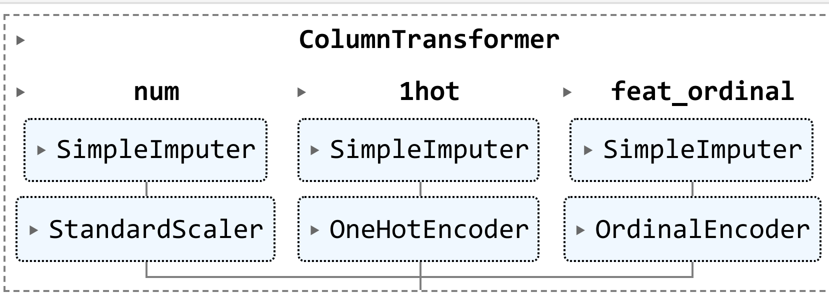
Out[57]:
```
Index(['Age'], dtype='object')
```

In [58]:
```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder


num_features = ['Age']
cat_features_1hot = ['Thyroid Function', 'Physical Examination', 'Adenopathy', 'Pathol
       'Risk', 'T', 'N', 'M', 'Stage', 'Response']
cat_features_ordinal = ['Gender', 'Smoking', 'Hx Smoking', 'Hx Radiothreapy']

OneHot_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy='most_frequent')),
    ('1hot_encoder', OneHotEncoder(sparse_output=False, handle_unknown='ignore'))
])

Ordinal_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy='most_frequent')),
    ('ordinal_encoder', OrdinalEncoder())
])
```

In [59]:
```
f_preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_features),
    ("1hot", OneHot_pipeline, cat_features_1hot),
    ("feat_ordinal", Ordinal_pipeline, cat_features_ordinal)
    ])
```

In [60]:
```
f_preprocessing
```

Out[60]:



In [61]:
```
recurrence.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 383 entries, 0 to 382
Data columns (total 16 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Age                   383 non-null    int64
 1   Gender                383 non-null    object
 2   Smoking               383 non-null    object
 3   Hx Smoking            383 non-null    object
 4   Hx Radiothreapy       383 non-null    object
 5   Thyroid Function      383 non-null    object
 6   Physical Examination  383 non-null    object
 7   Adenopathy            383 non-null    object
 8   Pathology             383 non-null    object
 9   Focality              383 non-null    object
 10  Risk                  383 non-null    object
 11  T                     383 non-null    object
 12  N                     383 non-null    object
 13  M                     383 non-null    object
 14  Stage                 383 non-null    object
 15  Response              383 non-null    object
dtypes: int64(1), object(15)
memory usage: 48.0+ KB
```

In [62]:
```
recurrence_prepared = f_preprocessing.fit_transform(recurrence)
```

In [63]:
```
recurrence_prepared.head()
```

Out[63]:

	num__Age	1hot__Thyroid Function_Clinical Hyperthyroidism	1hot__Thyroid Function_Clinical Hypothyroidism	1hot__Thyroid Function_Euthyroid	1hot__Thyroid Function_Subclinical Hyperthyroidism	1hot_ Function_S Hypoth
0	-0.917439	0.0	0.0	1.0	0.0	
1	-0.454315	0.0	0.0	1.0	0.0	
2	-0.718957	0.0	0.0	1.0	0.0	
3	1.398184	0.0	0.0	1.0	0.0	
4	1.398184	0.0	0.0	1.0	0.0	

5 rows × 51 columns

SMOTE Oversampling

In [64]:
```
X = recurrence_prepared
Y = recurrence_labels
```

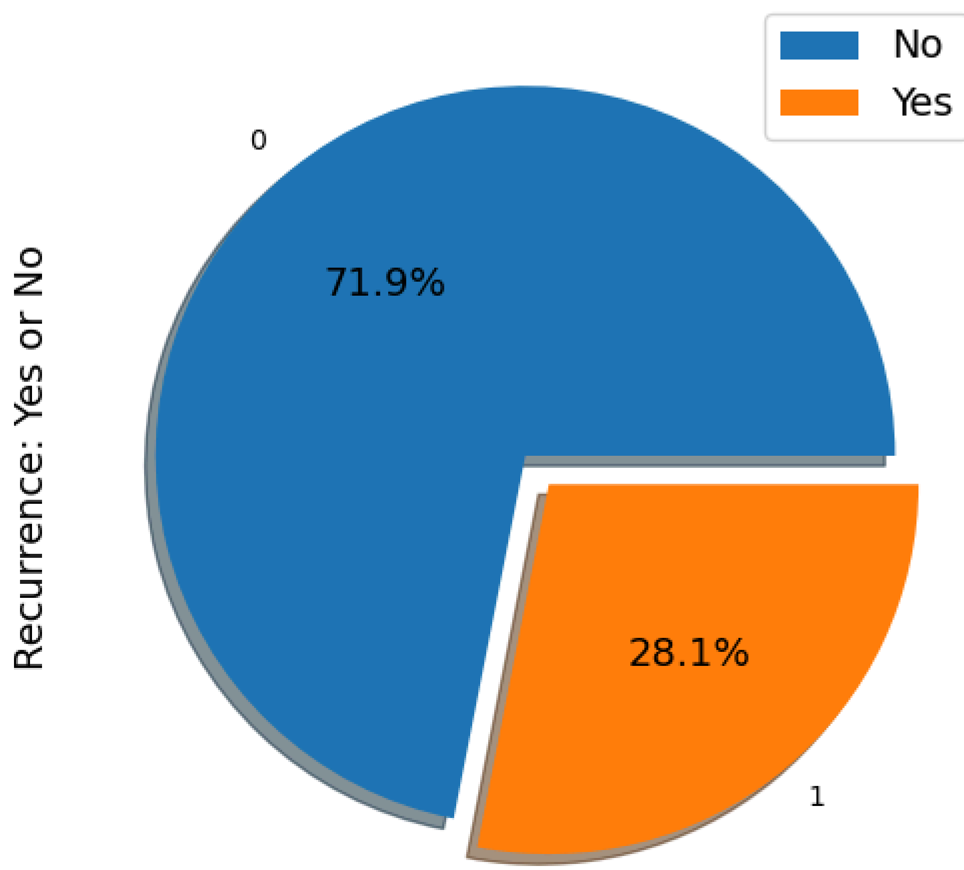
In [65]:
```
from sklearn.model_selection import train_test_split
# Split data into train and test set

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, stratify = Y,
```

In [66]:
```
from imblearn.over_sampling import SMOTE
smote = SMOTE(sampling_strategy="minority")
X_train_SMOTE, y_train_SMOTE = smote.fit_resample(X_train, Y_train)
```

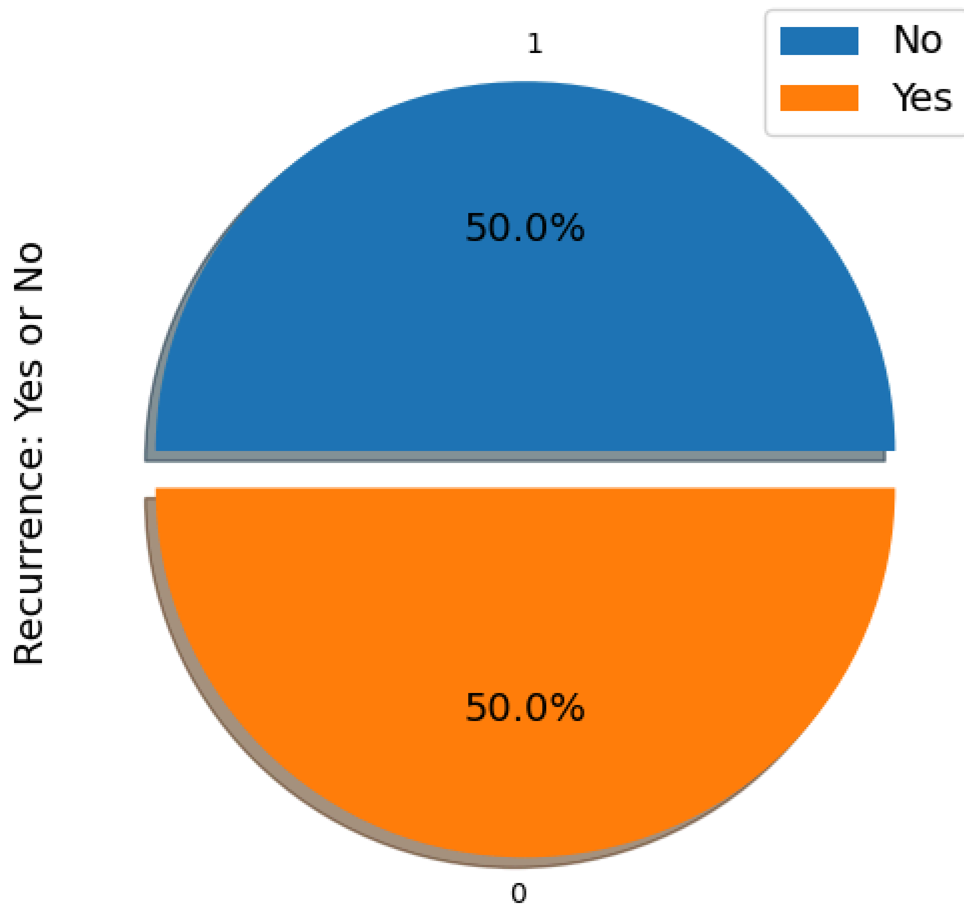
```
X_train_SMOTE.shape, y_train_SMOTE.shape
```

Out[66]: ((440, 51), (440, 1))

In [67]:
```
import matplotlib.pyplot as plt
#Helper function for data distribution
#Visualize proportion of Terminated Vs. Active
def show_recurrence_distrib(data):
  count = ""
  if isinstance(data, pd.DataFrame):
    count = data['Recurred'].value_counts()
  else:
    count = data.value_counts()


  count.plot(kind="pie", explode= [0, 0.1], figsize=(6,6), autopct = "%1.1f%%", shadow
  plt.ylabel("Recurrence: Yes or No")
  plt.legend(["No", "Yes"])
  plt.show()

#Visualize the proportion of borrowers
show_recurrence_distrib(Y_train)
```



In [68]:
```
show_recurrence_distrib(y_train_SMOTE)
```



Supervised Learning

Logistic Regression

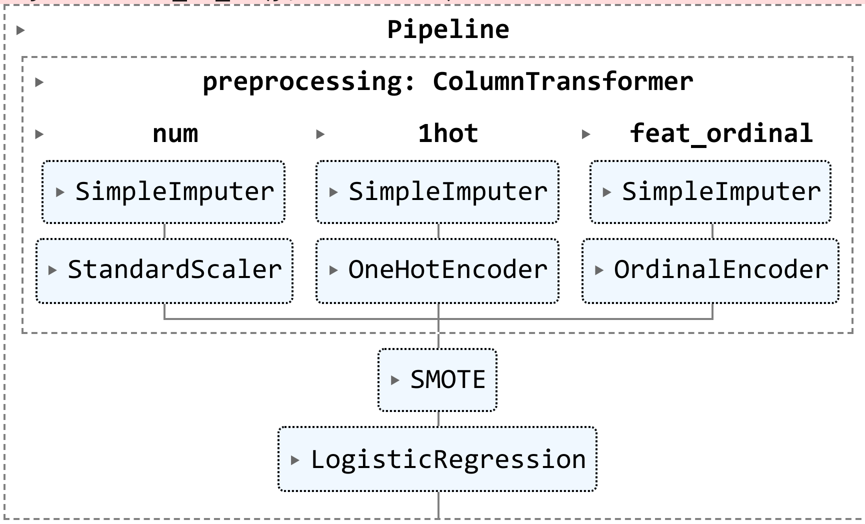
Logistic Regression with SMOTE

In [52]:
```
from imblearn.pipeline import Pipeline as imbpipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix


# Define the logistic regression pipeline with SMOTE
logReg_s = imbpipeline(steps=[
    ["preprocessing", f_preprocessing],
    ["SMOTE", SMOTE(random_state=42, sampling_strategy='minority')],
    ["logistic", LogisticRegression(solver='saga', random_state=42, C=1, penalty='l1',
])

X = recurrence
Y = recurrence_labels
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, stratify=Y, r
```

In [53]:
```
logReg_s.fit(X_train, Y_train)
```

```
C:\Users\CKY\anaconda3\Lib\site-packages\sklearn\utils\validation.py:1143: DataConver
sionWarning: A column-vector y was passed when a 1d array was expected. Please change
the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

Out[53]:



In [257…

```python
y_pred = logReg_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)
P = np.sum(Y_test.values)
N = len(Y_test) - P

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Logistic Regression with SMOTE")
print("-------------------------------------")
print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Logistic Regression with SMOTE
-------------------------------------
Precision: 1.0
Accuracy: 0.974025974025974
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.0
F1-score: 0.9523809523809523
```

Logistic Regression with SMOTE and Stratified K-Folds

In [259…

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

import warnings
from sklearn.exceptions import DataConversionWarning

# Suppress DataConversionWarning
warnings.filterwarnings(action='ignore', category=DataConversionWarning)

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
target = df.loc[:, 'Recurred']
print("Logistic Regression with SMOTE - Stratified K-Folds")
print("-------------------------------------")
# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = logReg_s.predict(X_test)

    # Print evaluation metrics
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
for train_index, test_index in skf.split(df, target):
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
Logistic Regression with SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.961038961038961
Fold 1 Precision: 1.0
Fold 1 Recall: 0.8636363636363636
Fold 1 F1 Score: 0.9268292682926829

Fold 2 Accuracy: 0.935064935064935
Fold 2 Precision: 0.9473684210526315
Fold 2 Recall: 0.8181818181818182
Fold 2 F1 Score: 0.8780487804878049

Fold 3 Accuracy: 0.948051948051948
Fold 3 Precision: 1.0
Fold 3 Recall: 0.8181818181818182
Fold 3 F1 Score: 0.9

Fold 4 Accuracy: 0.868421052631579
Fold 4 Precision: 0.6896551724137931
Fold 4 Recall: 0.9523809523809523
Fold 4 F1 Score: 0.7999999999999999

Fold 5 Accuracy: 0.7763157894736842
Fold 5 Precision: 0.5526315789473685
Fold 5 Recall: 1.0
Fold 5 F1 Score: 0.711864406779661
```

Logistic Regression with SMOTE - Grid Search

In [261…

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Define the parameter grid
param_grid = {
    'logistic__C': [0.1, 1, 10],  # Regularization parameter
    'logistic__penalty': ['l1', 'l2']  # Penalty term
}


# Perform grid search
grid_search = GridSearchCV(logReg_s, param_grid, cv=5, scoring='f1')
grid_search.fit(X_train, Y_train)

print("Logistic Regression with SMOTE - Grid Search")
print("-------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best F1-score on validation data:", grid_search.best_score_)

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

# Compute confusion matrix and other metrics
confusion_matrix_result = confusion_matrix(Y_test, y_pred)
tn, fp, fn, tp = confusion_matrix_result.ravel()
```

```
precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N
f1 = 2 * (precision * recall) / (precision + recall)

# Print metrics
print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Logistic Regression with SMOTE - Grid Search
-------------------------------------
Best parameters found: {'logistic__C': 1, 'logistic__penalty': 'l1'}
Best F1-score on validation data: 0.9325421396628826
Precision: 1.0
Accuracy: 0.974025974025974
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.0
F1-score: 0.9523809523809523
```

Logistic Regression with SMOTE and PCA

In [263…

```
from sklearn.decomposition import PCA

logReg_s = imbpipeline(steps=[
    ["preprocessing", f_preprocessing],
    ["pca", PCA(n_components=0.95)],  # Specify the desired explained variance ratio
    ["SMOTE", SMOTE(random_state=42, sampling_strategy='minority')],
    ["logistic", LogisticRegression(solver='saga', random_state=42, C=1, penalty='l1',
])

logReg_s.fit(X_train, Y_train)

y_pred = logReg_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)
P = np.sum(Y_test.values)
N = len(Y_test) - P

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Logistic Regression with SMOTE - PCA")
print("-------------------------------------")

print("Precision:", precision)
```

```
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Logistic Regression with SMOTE - PCA
-------------------------------------
Precision: 1.0
Accuracy: 0.974025974025974
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.0
F1-score: 0.9523809523809523
```

Logistic Regression without SMOTE

In [54]:

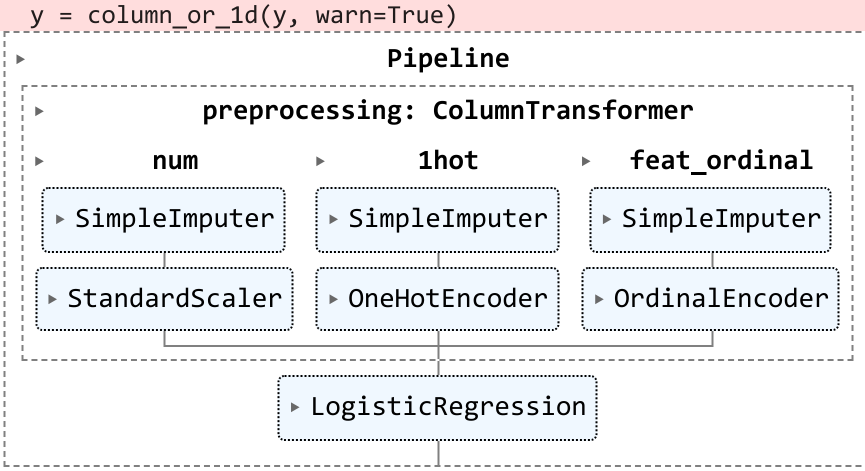
```
from imblearn.pipeline import Pipeline as imbpipeline
from sklearn.linear_model import LogisticRegression


# Define the logistic regression pipeline with SMOTE
logReg = imbpipeline(steps=[
    ["preprocessing", f_preprocessing],
    ["logistic", LogisticRegression(solver='saga', random_state=42, C=10, penalty='l2'
])

X = recurrence
Y = recurrence_labels
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, stratify=Y, r
```

In [55]:

```
logReg.fit(X_train, Y_train)
```

```
C:\Users\CKY\anaconda3\Lib\site-packages\sklearn\utils\validation.py:1143: DataConver
sionWarning: A column-vector y was passed when a 1d array was expected. Please change
the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

Out[55]:



In [308…

```
y_pred = logReg.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)
P = np.sum(Y_test.values)
N = len(Y_test) - P
```

```
tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Logistic Regression without SMOTE")
print("------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Logistic Regression without SMOTE
------------------------------------
Precision: 0.9523809523809523
Accuracy: 0.961038961038961
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.01818181818181818
F1-score: 0.9302325581395349
```

Logistic Regression without SMOTE - Stratified K-Folds

In [309…

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
target = df.loc[:, 'Recurred']

print("Logistic Regression without SMOTE - Stratified K-Folds")
print("------------------------------------")

# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = logReg.predict(X_test)

    # Print evaluation metrics
```

```
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
for train_index, test_index in skf.split(df, target):
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
Logistic Regression without SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.948051948051948
Fold 1 Precision: 1.0
Fold 1 Recall: 0.8181818181818182
Fold 1 F1 Score: 0.9

Fold 2 Accuracy: 0.922077922077922
Fold 2 Precision: 0.9
Fold 2 Recall: 0.8181818181818182
Fold 2 F1 Score: 0.8571428571428572

Fold 3 Accuracy: 0.948051948051948
Fold 3 Precision: 1.0
Fold 3 Recall: 0.8181818181818182
Fold 3 F1 Score: 0.9

Fold 4 Accuracy: 0.8289473684210527
Fold 4 Precision: 0.6666666666666666
Fold 4 Recall: 0.7619047619047619
Fold 4 F1 Score: 0.7111111111111111

Fold 5 Accuracy: 0.7368421052631579
Fold 5 Precision: 0.5121951219512195
Fold 5 Recall: 1.0
Fold 5 F1 Score: 0.6774193548387097
```

Logistic Regression without SMOTE - Grid Search

In [310…

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Define the parameter grid
param_grid = {
    'logistic__C': [0.1, 1, 10],  # Regularization parameter
    'logistic__penalty': ['l1', 'l2']  # Penalty term
}


# Perform grid search
grid_search = GridSearchCV(logReg, param_grid, cv=5, scoring='f1')
grid_search.fit(X_train, Y_train)

print("Logistic Regression without SMOTE - Grid Search")
```

```
print("-------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best F1-score on validation data:", grid_search.best_score_)

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

# Compute confusion matrix and other metrics
confusion_matrix_result = confusion_matrix(Y_test, y_pred)
tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N
f1 = 2 * (precision * recall) / (precision + recall)

# Print metrics
print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Logistic Regression without SMOTE - Grid Search
-------------------------------------
Best parameters found: {'logistic__C': 10, 'logistic__penalty': 'l2'}
Best F1-score on validation data: 0.9404634581105169
Precision: 0.9523809523809523
Accuracy: 0.961038961038961
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.01818181818181818
F1-score: 0.9302325581395349
```

Logistic Regression without SMOTE, with PCA

In [311…

```
from sklearn.decomposition import PCA

logReg = imbpipeline(steps=[
    ["preprocessing", f_preprocessing],
    ["pca", PCA(n_components=0.95)],  # Specify the desired explained variance ratio
    ["logistic", LogisticRegression(solver='saga', random_state=42, C=10, penalty='l2'
])

logReg.fit(X_train, Y_train)

y_pred = logReg.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)
P = np.sum(Y_test.values)
N = len(Y_test) - P

tn, fp, fn, tp = confusion_matrix_result.ravel()
```

```
precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Logistic Regression without SMOTE - PCA")
print("--------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

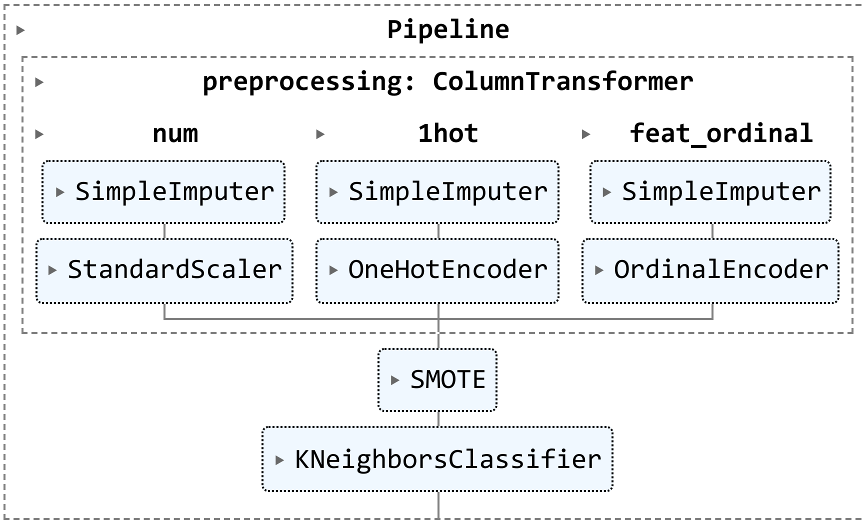
```
Logistic Regression without SMOTE - PCA
-------------------------------------
Precision: 0.95
Accuracy: 0.948051948051948
Recall: 0.8636363636363636
True positive rate: 0.8636363636363636
False positive rate: 0.01818181818181818
F1-score: 0.9047619047619048
```

KNN

with SMOTE

In [56]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix

knn_classifier_s = imbpipeline(steps = [
                                    ["preprocessing", f_preprocessing],
                                ["SMOTE", SMOTE(random_state=42, sampling_strategy='mi
                                ['knn', KNeighborsClassifier(n_neighbors=2, weights='u

knn_classifier_s
```

Out[56]:

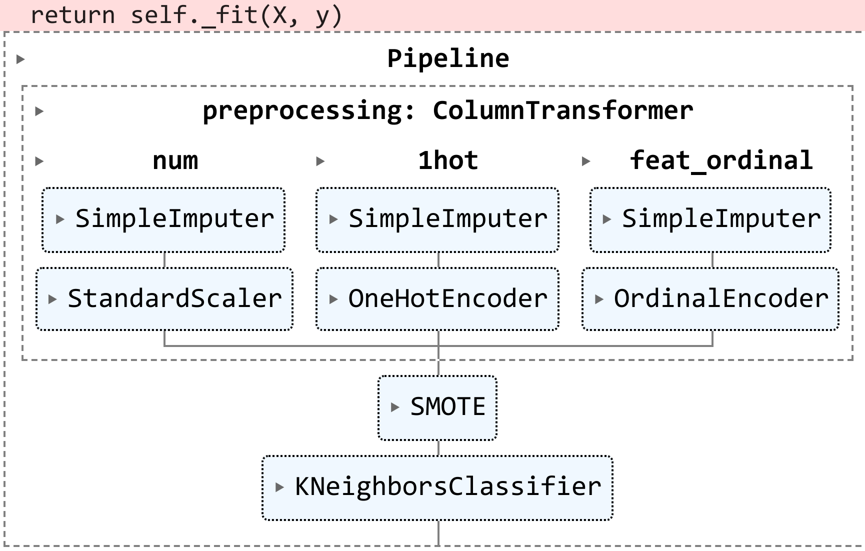


In [57]:

```
X = recurrence
Y = recurrence_labels
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, stratify=Y, r

knn_classifier_s.fit(X_train, Y_train)
```

```
C:\Users\CKY\anaconda3\Lib\site-packages\sklearn\neighbors\_classification.py:215: Da
taConversionWarning: A column-vector y was passed when a 1d array was expected. Pleas
e change the shape of y to (n_samples,), for example using ravel().
  return self._fit(X, y)
```

Out[57]:



In [347…

```
y_pred = knn_classifier_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)
P = np.sum(Y_test.values)
N = len(Y_test) - P

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N
```

```
f1 = 2 * (precision * recall) / (precision + recall)

print("KNN with SMOTE")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
KNN with SMOTE
-------------------------------------
Precision: 0.9523809523809523
Accuracy: 0.961038961038961
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.01818181818181818
F1-score: 0.9302325581395349
```

KNN with SMOTE - Stratified-K Folds

In [348…

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
target = df.loc[:, 'Recurred']

print("KNN with SMOTE - Stratified K-Folds")
print("-------------------------------------")

# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = knn_classifier_s.predict(X_test)

    # Print evaluation metrics
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
```

```
for train_index, test_index in skf.split(df, target):
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
KNN with SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.8831168831168831
Fold 1 Precision: 1.0
Fold 1 Recall: 0.5909090909090909
Fold 1 F1 Score: 0.7428571428571429

Fold 2 Accuracy: 0.8961038961038961
Fold 2 Precision: 0.8888888888888888
Fold 2 Recall: 0.7272727272727273
Fold 2 F1 Score: 0.7999999999999999

Fold 3 Accuracy: 0.9090909090909091
Fold 3 Precision: 1.0
Fold 3 Recall: 0.6818181818181818
Fold 3 F1 Score: 0.8108108108108109

Fold 4 Accuracy: 0.8421052631578947
Fold 4 Precision: 0.6551724137931034
Fold 4 Recall: 0.9047619047619048
Fold 4 F1 Score: 0.7599999999999999

Fold 5 Accuracy: 0.7105263157894737
Fold 5 Precision: 0.4878048780487805
Fold 5 Recall: 0.9523809523809523
Fold 5 F1 Score: 0.6451612903225806
```

KNN with SMOTE - Grid Search

In [349…

```
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier

# Define the parameter grid
param_grid = {
    'knn__n_neighbors': [1,2,3,4,5,6,7,8,9],  # Test different number of neighbors
    'knn__weights': ['uniform', 'distance']  # Test different weighting schemes
}


# Perform grid search
grid_search = GridSearchCV(knn_classifier_s, param_grid, cv=5, scoring='f1')
grid_search.fit(X_train, Y_train)


print("KNN with SMOTE - Grid Search")
print("--------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best F1-score on validation data:", grid_search.best_score_)

# Evaluate the best model on test data
```

```
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

# Compute confusion matrix and other metrics
confusion_matrix_result = confusion_matrix(Y_test, y_pred)
tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N
f1 = 2 * (precision * recall) / (precision + recall)

# Print metrics
print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
KNN with SMOTE - Grid Search
-------------------------------------
Best parameters found: {'knn__n_neighbors': 2, 'knn__weights': 'uniform'}
Best F1-score on validation data: 0.8582983193277312
Precision: 0.9523809523809523
Accuracy: 0.961038961038961
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.01818181818181818
F1-score: 0.9302325581395349
```

KNN with SMOTE and PCA

In [358…

```
from sklearn.decomposition import PCA

knn_classifier_s = imbpipeline(steps = [
                                    ["preprocessing", f_preprocessing],
                                ["pca", PCA(n_components=0.95)],
                                ["SMOTE", SMOTE(random_state=42, sampling_strategy='mi
                                ['knn', KNeighborsClassifier(n_neighbors=3, weights='u

knn_classifier_s.fit(X_train, Y_train)

y_pred = knn_classifier_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)
P = np.sum(Y_test.values)
N = len(Y_test) - P

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N
```

```
f1 = 2 * (precision * recall) / (precision + recall)

print("KNN with SMOTE - PCA")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

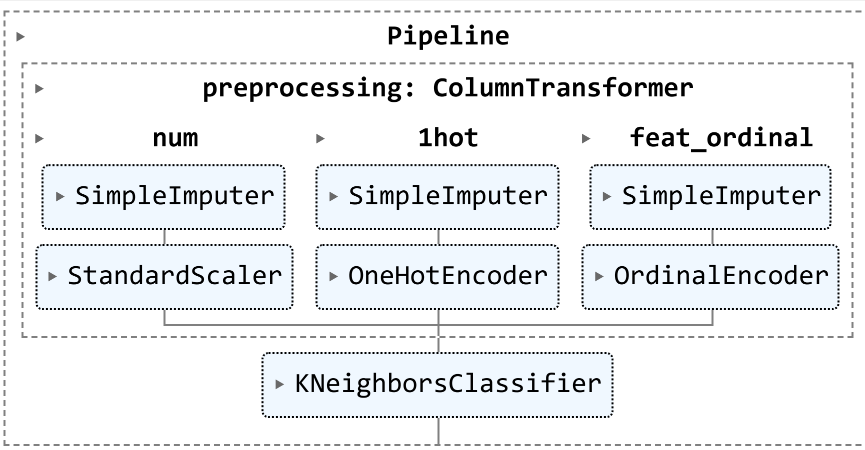
```
KNN with SMOTE - PCA
-------------------------------------
Precision: 0.875
Accuracy: 0.948051948051948
Recall: 0.9545454545454546
True positive rate: 0.9545454545454546
False positive rate: 0.05454545454545454
F1-score: 0.9130434782608695
```

KNN without SMOTE

In [58]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix

knn_classifier = imbpipeline(steps = [
                                    ["preprocessing", f_preprocessing],
                                ['knn', KNeighborsClassifier(n_neighbors=4, weights='d

knn_classifier
```

Out[58]:

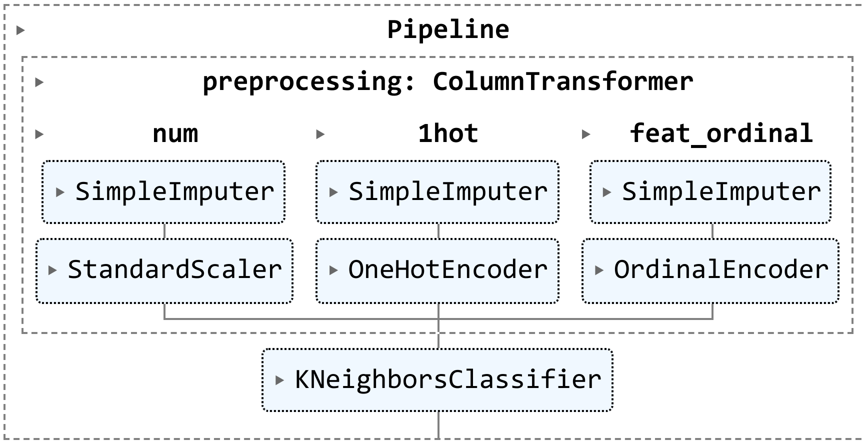


In [59]:

```
X = recurrence
Y = recurrence_labels
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, stratify=Y, r

knn_classifier.fit(X_train, Y_train)
```

```
C:\Users\CKY\anaconda3\Lib\site-packages\sklearn\neighbors\_classification.py:215: Da
taConversionWarning: A column-vector y was passed when a 1d array was expected. Pleas
e change the shape of y to (n_samples,), for example using ravel().
  return self._fit(X, y)
```

Out[59]:



In [368…

```
y_pred = knn_classifier.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)
P = np.sum(Y_test.values)
N = len(Y_test) - P

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("KNN without SMOTE")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
KNN without SMOTE
-------------------------------------
Precision: 0.9090909090909091
Accuracy: 0.948051948051948
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.03636363636363636
F1-score: 0.9090909090909091
```

KNN without SMOTE - Stratified-K folds

In [369…

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
```

```
target = df.loc[:, 'Recurred']

print("KNN without SMOTE - Stratified K-Folds")
print("-------------------------------------")

# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = knn_classifier.predict(X_test)

    # Print evaluation metrics
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
for train_index, test_index in skf.split(df, target):
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
KNN without SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.8701298701298701
Fold 1 Precision: 1.0
Fold 1 Recall: 0.5454545454545454
Fold 1 F1 Score: 0.7058823529411764

Fold 2 Accuracy: 0.8571428571428571
Fold 2 Precision: 0.9230769230769231
Fold 2 Recall: 0.5454545454545454
Fold 2 F1 Score: 0.6857142857142856

Fold 3 Accuracy: 0.9090909090909091
Fold 3 Precision: 1.0
Fold 3 Recall: 0.6818181818181818
Fold 3 F1 Score: 0.8108108108108109

Fold 4 Accuracy: 0.8421052631578947
Fold 4 Precision: 0.6551724137931034
Fold 4 Recall: 0.9047619047619048
Fold 4 F1 Score: 0.7599999999999999

Fold 5 Accuracy: 0.75
Fold 5 Precision: 0.5263157894736842
Fold 5 Recall: 0.9523809523809523
Fold 5 F1 Score: 0.6779661016949152
```

KNN without SMOTE - Grid Search

In [370…

```python
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier

# Define the parameter grid
param_grid = {
    'knn__n_neighbors': [1,2,3,4,5,6,7,8,9],  # Test different number of neighbors
    'knn__weights': ['uniform', 'distance']  # Test different weighting schemes
}


# Perform grid search
grid_search = GridSearchCV(knn_classifier, param_grid, cv=5, scoring='f1')
grid_search.fit(X_train, Y_train)


print("KNN without  SMOTE - Grid Search")
print("-------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best F1-score on validation data:", grid_search.best_score_)

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

# Compute confusion matrix and other metrics
confusion_matrix_result = confusion_matrix(Y_test, y_pred)
tn, fp, fn, tp = confusion_matrix_result.ravel()
```

```
precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N
f1 = 2 * (precision * recall) / (precision + recall)

# Print metrics
print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
KNN without  SMOTE - Grid Search
-------------------------------------
Best parameters found: {'knn__n_neighbors': 4, 'knn__weights': 'distance'}
Best F1-score on validation data: 0.866474583563833
Precision: 0.9090909090909091
Accuracy: 0.948051948051948
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.03636363636363636
F1-score: 0.9090909090909091
```

KNN without SMOTE, with PCA

In [376…

```
from sklearn.decomposition import PCA

knn_classifier = imbpipeline(steps = [
                                    ["preprocessing", f_preprocessing],
                                ["pca", PCA(n_components=0.95)],
                                ['knn', KNeighborsClassifier(n_neighbors=4, weights='d

knn_classifier.fit(X_train, Y_train)

y_pred = knn_classifier.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("KNN without SMOTE - PCA")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
```

```
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
KNN without SMOTE - PCA
-------------------------------------
Precision: 0.9523809523809523
Accuracy: 0.961038961038961
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.01818181818181818
F1-score: 0.9302325581395349
```

Decision Tree

with SMOTE

In [419…

```
from sklearn.tree import DecisionTreeClassifier

# Define your pipeline
decision_tree_s = imbpipeline([
    ['preprocessing', f_preprocessing],
    ['smote', SMOTE(random_state=42, sampling_strategy='minority')],
    ['decision_tree', DecisionTreeClassifier(criterion='entropy', max_depth=10, min_sa
])
```

In [420…

```
X = recurrence
Y = recurrence_labels
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, stratify=Y, r
```

In [421…

```
# Train the model
decision_tree_s.fit(X_train, Y_train)
# Predictions on the test set
y_pred = decision_tree_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Decision Tree with SMOTE")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Decision Tree with SMOTE
-------------------------------------
Precision: 1.0
Accuracy: 0.961038961038961
Recall: 0.8636363636363636
True positive rate: 0.8636363636363636
False positive rate: 0.0
F1-score: 0.9268292682926829
```

Decision Tree with SMOTE - Stratified K Folds

In [422…

```python
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
target = df.loc[:, 'Recurred']

print("Decision Tree with SMOTE - Stratified K-Folds")
print("-------------------------------------")

# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = decision_tree_s.predict(X_test)

    # Print evaluation metrics
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
for train_index, test_index in skf.split(df, target):
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
Decision Tree with SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.961038961038961
Fold 1 Precision: 1.0
Fold 1 Recall: 0.8636363636363636
Fold 1 F1 Score: 0.9268292682926829

Fold 2 Accuracy: 0.922077922077922
Fold 2 Precision: 0.9444444444444444
Fold 2 Recall: 0.7727272727272727
Fold 2 F1 Score: 0.85

Fold 3 Accuracy: 0.935064935064935
Fold 3 Precision: 0.9473684210526315
Fold 3 Recall: 0.8181818181818182
Fold 3 F1 Score: 0.8780487804878049

Fold 4 Accuracy: 0.8421052631578947
Fold 4 Precision: 0.6551724137931034
Fold 4 Recall: 0.9047619047619048
Fold 4 F1 Score: 0.7599999999999999

Fold 5 Accuracy: 0.7105263157894737
Fold 5 Precision: 0.48717948717948717
Fold 5 Recall: 0.9047619047619048
Fold 5 F1 Score: 0.6333333333333333
```

Decision Tree with SMOTE - Grid Search

In [423…

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    'decision_tree__max_depth': [None, 10, 20],
    'decision_tree__min_samples_split': [2, 5, 10],
    'decision_tree__min_samples_leaf': [1, 2, 4]
}

# Perform grid search
grid_search = GridSearchCV(decision_tree_s, param_grid, cv=5, scoring='f1')
grid_search.fit(X_train, Y_train)

print("Decision Tree with SMOTE - Grid Search")
print("--------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best F1-score on validation data:", grid_search.best_score_)

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

# Compute confusion matrix and other metrics
confusion_matrix_result = confusion_matrix(Y_test, y_pred)
# Calculate TP, FP, TN, FN, precision, accuracy, recall, true positive rate, false pos
# Print metrics
```

```
Decision Tree with SMOTE - Grid Search
-------------------------------------
Best parameters found: {'decision_tree__max_depth': 20, 'decision_tree__min_samples_l
eaf': 4, 'decision_tree__min_samples_split': 10}
Best F1-score on validation data: 0.8969971205265324
```

Decision Tree with SMOTE and PCA

In [425…

```python
from sklearn.decomposition import PCA

decision_tree_s = imbpipeline([
    ['preprocessing', f_preprocessing],
    ["pca", PCA(n_components=0.95)],
    ['smote', SMOTE(random_state=42, sampling_strategy='minority')],
    ['classifier', DecisionTreeClassifier(criterion='entropy', max_depth=10, min_sampl
])

decision_tree_s.fit(X_train, Y_train)

y_pred = decision_tree_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Decision Tree with SMOTE - PCA")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Decision Tree with SMOTE - PCA
-------------------------------------
Precision: 0.7916666666666666
Accuracy: 0.8961038961038961
Recall: 0.8636363636363636
True positive rate: 0.8636363636363636
False positive rate: 0.09090909090909091
F1-score: 0.8260869565217391
```

Decision Tree without SMOTE

In [459…

```python
from sklearn.tree import DecisionTreeClassifier

# Define your pipeline
decision_tree = imbpipeline([
```

```
    ['preprocessing', f_preprocessing],
    ['classifier', DecisionTreeClassifier(criterion='entropy', min_samples_leaf=1, mir
])
```

In [460…

```
X = recurrence
Y = recurrence_labels
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, stratify=Y, r
```

In [465…

```
# Train the model
decision_tree.fit(X_train, Y_train)
# Predictions on the test set
y_pred = decision_tree.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Decision Tree without SMOTE")
print("--------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Decision Tree without SMOTE
-------------------------------------
Precision: 0.95
Accuracy: 0.948051948051948
Recall: 0.8636363636363636
True positive rate: 0.8636363636363636
False positive rate: 0.01818181818181818
F1-score: 0.9047619047619048
```

Decision Tree without SMOTE - Stratified K Folds

In [466…

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
target = df.loc[:, 'Recurred']

print("Decision Tree wit0hout SMOTE - Stratified K-Folds")
print("--------------------------------------")
```

```python
# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = decision_tree.predict(X_test)

    # Print evaluation metrics
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
for train_index, test_index in skf.split(df, target):
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
Decision Tree wit0hout SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.961038961038961
Fold 1 Precision: 1.0
Fold 1 Recall: 0.8636363636363636
Fold 1 F1 Score: 0.9268292682926829

Fold 2 Accuracy: 0.935064935064935
Fold 2 Precision: 0.9473684210526315
Fold 2 Recall: 0.8181818181818182
Fold 2 F1 Score: 0.8780487804878049

Fold 3 Accuracy: 0.935064935064935
Fold 3 Precision: 0.9473684210526315
Fold 3 Recall: 0.8181818181818182
Fold 3 F1 Score: 0.8780487804878049

Fold 4 Accuracy: 0.9473684210526315
Fold 4 Precision: 0.9047619047619048
Fold 4 Recall: 0.9047619047619048
Fold 4 F1 Score: 0.9047619047619048

Fold 5 Accuracy: 0.7368421052631579
Fold 5 Precision: 0.5128205128205128
Fold 5 Recall: 0.9523809523809523
Fold 5 F1 Score: 0.6666666666666666
```

Decision Tree without SMOTE - Grid Search

In [475…

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    'classifier__max_depth': [None, 10, 20],
    'classifier__min_samples_split': [2, 5, 10],
    'classifier__min_samples_leaf': [1, 2, 4]
}

# Perform grid search
grid_search = GridSearchCV(decision_tree, param_grid, cv=5, scoring='f1')
grid_search.fit(X_train, Y_train)

print("Decision Tree without SMOTE - Grid Search")
print("-------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best F1-score on validation data:", grid_search.best_score_)

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

# Compute confusion matrix and other metrics
confusion_matrix_result = confusion_matrix(Y_test, y_pred)
# Calculate TP, FP, TN, FN, precision, accuracy, recall, true positive rate, false pos
# Print metrics
```

```
Decision Tree without SMOTE - Grid Search
-------------------------------------
Best parameters found: {'classifier__max_depth': 20, 'classifier__min_samples_leaf': 
4, 'classifier__min_samples_split': 10}
Best F1-score on validation data: 0.8189851616322205
```

Decision Tree without SMOTE, with PCA

In [473…

```
from sklearn.decomposition import PCA

decision_tree = imbpipeline([
    ['preprocessing', f_preprocessing],
    ["pca", PCA(n_components=0.98)],
    ['classifier', DecisionTreeClassifier(criterion='entropy', min_samples_leaf=1, mir
])

decision_tree.fit(X_train, Y_train)

y_pred = decision_tree.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N
```

```
f1 = 2 * (precision * recall) / (precision + recall)

print("Decision Tree without SMOTE - PCA")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Decision Tree without SMOTE - PCA
-------------------------------------
Precision: 0.782608695652174
Accuracy: 0.8831168831168831
Recall: 0.8181818181818182
True positive rate: 0.8181818181818182
False positive rate: 0.09090909090909091
F1-score: 0.8
```

Regression Tree

with SMOTE

In [477…

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error


# Define your pipeline
regression_s = imbpipeline([
    ['preprocessing', f_preprocessing],
    ['smote', SMOTE(random_state=42, sampling_strategy='minority')],
    ['regressor', DecisionTreeRegressor()] # Regression tree
])

# Train the model
regression_s.fit(X_train, Y_train)

y_pred = regression_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Regression Tree with SMOTE")
print("-------------------------------------")
```

```
print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Regression Tree with SMOTE
-------------------------------------
Precision: 0.9523809523809523
Accuracy: 0.961038961038961
Recall: 0.9090909090909091
True positive rate: 0.9090909090909091
False positive rate: 0.01818181818181818
F1-score: 0.9302325581395349
```

Regression Tree with SMOTE - Stratified K Folds

In [483…

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
target = df.loc[:, 'Recurred']

print("Regression Tree with SMOTE - Stratified K-Folds")
print("-------------------------------------")

# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = regression_s.predict(X_test)

    # Print evaluation metrics
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
for train_index, test_index in skf.split(df, target):
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
Regression Tree with SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.8441558441558441
Fold 1 Precision: 1.0
Fold 1 Recall: 0.45454545454545453
Fold 1 F1 Score: 0.625

Fold 2 Accuracy: 0.8311688311688312
Fold 2 Precision: 0.7142857142857143
Fold 2 Recall: 0.6818181818181818
Fold 2 F1 Score: 0.6976744186046512

Fold 3 Accuracy: 0.8961038961038961
Fold 3 Precision: 0.8888888888888888
Fold 3 Recall: 0.7272727272727273
Fold 3 F1 Score: 0.7999999999999999

Fold 4 Accuracy: 0.7105263157894737
Fold 4 Precision: 0.4838709677419355
Fold 4 Recall: 0.7142857142857143
Fold 4 F1 Score: 0.5769230769230769

Fold 5 Accuracy: 0.7763157894736842
Fold 5 Precision: 0.5555555555555556
Fold 5 Recall: 0.9523809523809523
Fold 5 F1 Score: 0.7017543859649122
```

Regression Tree with SMOTE - Grid Search

In [485…

```python
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor

# Define the parameter grid
param_grid = {
    'regressor__max_depth': [None, 10, 20],
    'regressor__min_samples_split': [2, 5, 10],
    'regressor__min_samples_leaf': [1, 2, 4]
}

# Perform grid search
grid_search = GridSearchCV(regression_s, param_grid, cv=5, scoring='neg_mean_squared_e
grid_search.fit(X_train, Y_train)

print("Regression Tree with SMOTE - Grid Search")
print("-------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best negative mean squared error on validation data:", grid_search.best_score_)

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

# Compute evaluation metrics
mse = mean_squared_error(Y_test, y_pred)
print("Mean Squared Error on test data:", mse)
```

```
Regression Tree with SMOTE - Grid Search
-------------------------------------
Best parameters found: {'regressor__max_depth': 10, 'regressor__min_samples_leaf': 4,
'regressor__min_samples_split': 10}
Best negative mean squared error on validation data: -0.10672136360468938
Mean Squared Error on test data: 0.06757976591309925
```

Regression Tree with SMOTE and PCA

In [479…

```python
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error


# Define your pipeline
regression_s = imbpipeline([
    ['preprocessing', f_preprocessing],
    ["pca", PCA(n_components=0.98)],
    ['smote', SMOTE(random_state=42, sampling_strategy='minority')],
    ['regressor', DecisionTreeRegressor()] # Regression tree
])

# Train the model
regression_s.fit(X_train, Y_train)

y_pred = regression_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Regression Tree with SMOTE - PCA")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Regression Tree with SMOTE - PCA
-------------------------------------
Precision: 0.8095238095238095
Accuracy: 0.8831168831168831
Recall: 0.7727272727272727
True positive rate: 0.7727272727272727
False positive rate: 0.07272727272727272
F1-score: 0.7906976744186046
```

without SMOTE

In [186…

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error


# Define your pipeline
regression = imbpipeline([
    ['preprocessing', f_preprocessing],
    ['regressor', DecisionTreeRegressor()] # Regression tree
])

# Train the model
regression.fit(X_train, Y_train)

y_pred = regression.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Precision: 0.9473684210526315
Accuracy: 0.935064935064935
Recall: 0.8181818181818182
True positive rate: 0.8181818181818182
False positive rate: 0.01818181818181818
F1-score: 0.8780487804878049
```

without SMOTE, with PCA

In [187…

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error


# Define your pipeline
regression = imbpipeline([
    ['preprocessing', f_preprocessing],
    ["pca", PCA(n_components=0.95)],
    ['regressor', DecisionTreeRegressor()] # Regression tree
])

# Train the model
regression.fit(X_train, Y_train)

y_pred = regression.predict(X_test)
```

```
confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Precision: 0.85
Accuracy: 0.8961038961038961
Recall: 0.7727272727272727
True positive rate: 0.7727272727272727
False positive rate: 0.05454545454545454
F1-score: 0.8095238095238095
```

Naive Bayes

with SMOTE

In [533…

```
from sklearn.naive_bayes import BernoulliNB

# Create a pipeline with preprocessing, SMOTE, scaling (if necessary), and Bernoulli N
naive_s = imbpipeline([
    ['preprocessing', f_preprocessing],
    ['smote', SMOTE(random_state=42, sampling_strategy='minority')],
    ['naive_bayes', BernoulliNB(alpha=0.1)]  # Bernoulli Naive Bayes classifier
])

# Fit the model
naive_s.fit(X_train, Y_train)

# Predictions on the test set
y_pred = naive_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)
```

```
print("Naive Bayes with SMOTE")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Naive Bayes with SMOTE
-------------------------------------
Precision: 0.9411764705882353
Accuracy: 0.948051948051948
Recall: 0.8421052631578947
True positive rate: 0.7272727272727273
False positive rate: 0.01818181818181818
F1-score: 0.8888888888888888
```

Naive Bayes with SMOTE - Stratified K Folds

In [519…

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
target = df.loc[:, 'Recurred']

print("Naive Bayes with SMOTE - Stratified K-Folds")
print("-------------------------------------")

# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = naive_s.predict(X_test)

    # Print evaluation metrics
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
for train_index, test_index in skf.split(df, target):
```

```
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
Naive Bayes with SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.8701298701298701
Fold 1 Precision: 0.9285714285714286
Fold 1 Recall: 0.5909090909090909
Fold 1 F1 Score: 0.7222222222222223

Fold 2 Accuracy: 0.9090909090909091
Fold 2 Precision: 1.0
Fold 2 Recall: 0.6818181818181818
Fold 2 F1 Score: 0.8108108108108109

Fold 3 Accuracy: 0.974025974025974
Fold 3 Precision: 1.0
Fold 3 Recall: 0.9090909090909091
Fold 3 F1 Score: 0.9523809523809523

Fold 4 Accuracy: 0.8289473684210527
Fold 4 Precision: 0.625
Fold 4 Recall: 0.9523809523809523
Fold 4 F1 Score: 0.7547169811320755

Fold 5 Accuracy: 0.7236842105263158
Fold 5 Precision: 0.5
Fold 5 Recall: 1.0
Fold 5 F1 Score: 0.6666666666666666
```

Naive Bayes with SMOTE - Grid Search

In [520…

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    'naive_bayes__alpha': [0.1, 0.5, 1.0],    # Smoothing parameter
    'naive_bayes__binarize': [0.0, 0.5, 1.0]   # Binarization threshold
}

# Perform grid search
grid_search = GridSearchCV(naive_s, param_grid, cv=5, scoring='f1')
grid_search.fit(X_train, Y_train)

print("Naive Bayes with SMOTE - Grid Search")
print("-------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best F1-score on validation data:", grid_search.best_score_)

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

# Compute confusion matrix and other metrics
```

```
confusion_matrix_result = confusion_matrix(Y_test, y_pred)
# Calculate TP, FP, TN, FN, precision, accuracy, recall, true positive rate, false pos
# Print metrics
```

```
Naive Bayes with SMOTE - Grid Search
-------------------------------------
Best parameters found: {'naive_bayes__alpha': 0.1, 'naive_bayes__binarize': 0.0}
Best F1-score on validation data: 0.8285456885456887
```

Naive Bayes with SMOTE - PCA

In [527…

```
from sklearn.decomposition import PCA

naive_s = imbpipeline([
    ['preprocessing', f_preprocessing],
    ["pca", PCA(n_components=0.98)],
    ['smote', SMOTE(random_state=42, sampling_strategy='minority')],
    ['naive_bayes', BernoulliNB(alpha=0.1)]  # Bernoulli Naive Bayes classifier
])


naive_s.fit(X_train, Y_train)

y_pred = naive_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Naive Bayes with SMOTE - PCA")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Naive Bayes with SMOTE - PCA
-------------------------------------
Precision: 0.7272727272727273
Accuracy: 0.8831168831168831
Recall: 0.8421052631578947
True positive rate: 0.7272727272727273
False positive rate: 0.10909090909090909
F1-score: 0.7804878048780488
```

Naive Bayes without SMOTE

In [522…

```
from sklearn.naive_bayes import BernoulliNB

# Create a pipeline with preprocessing, SMOTE, scaling (if necessary), and Bernoulli N
naive = imbpipeline([
    ['preprocessing', f_preprocessing],
    ['naive_bayes', BernoulliNB(alpha=0.1)]  # Bernoulli Naive Bayes classifier
])

# Fit the model
naive.fit(X_train, Y_train)

# Predictions on the test set
y_pred = naive.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("Naive Bayes without SMOTE")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Naive Bayes without SMOTE
-------------------------------------
Precision: 0.9411764705882353
Accuracy: 0.948051948051948
Recall: 0.8421052631578947
True positive rate: 0.7272727272727273
False positive rate: 0.01818181818181818
F1-score: 0.8888888888888888
```

Naive Bayes without SMOTE - Stratified K Folds

In [523…

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
target = df.loc[:, 'Recurred']

print("Naive Bayes without SMOTE - Stratified K-Folds")
print("-------------------------------------")
```

```python
# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = naive.predict(X_test)

    # Print evaluation metrics
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
for train_index, test_index in skf.split(df, target):
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
Naive Bayes without SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.8701298701298701
Fold 1 Precision: 0.9285714285714286
Fold 1 Recall: 0.5909090909090909
Fold 1 F1 Score: 0.7222222222222223

Fold 2 Accuracy: 0.9090909090909091
Fold 2 Precision: 1.0
Fold 2 Recall: 0.6818181818181818
Fold 2 F1 Score: 0.8108108108108109

Fold 3 Accuracy: 0.974025974025974
Fold 3 Precision: 1.0
Fold 3 Recall: 0.9090909090909091
Fold 3 F1 Score: 0.9523809523809523

Fold 4 Accuracy: 0.8289473684210527
Fold 4 Precision: 0.625
Fold 4 Recall: 0.9523809523809523
Fold 4 F1 Score: 0.7547169811320755

Fold 5 Accuracy: 0.6973684210526315
Fold 5 Precision: 0.4772727272727273
Fold 5 Recall: 1.0
Fold 5 F1 Score: 0.6461538461538462
```

Naive Bayes without SMOTE -Grid Search

In [526…

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    'naive_bayes__alpha': [0.1, 0.5, 1.0],    # Smoothing parameter
    'naive_bayes__binarize': [0.0, 0.5, 1.0]   # Binarization threshold
}

# Perform grid search
grid_search = GridSearchCV(naive, param_grid, cv=5, scoring='f1')
grid_search.fit(X_train, Y_train)

print("Naive Bayes without SMOTE - Grid Search")
print("-------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best F1-score on validation data:", grid_search.best_score_)

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

# Compute confusion matrix and other metrics
confusion_matrix_result = confusion_matrix(Y_test, y_pred)
# Calculate TP, FP, TN, FN, precision, accuracy, recall, true positive rate, false pos
# Print metrics
```

```
Naive Bayes without SMOTE - Grid Search
-------------------------------------
Best parameters found: {'naive_bayes__alpha': 0.1, 'naive_bayes__binarize': 0.0}
Best F1-score on validation data: 0.8223035327686491
```

Naive Bayes without SMOTE - PCA

In [531…

```
from sklearn.decomposition import PCA

naive = imbpipeline([
    ['preprocessing', f_preprocessing],
    ["pca", PCA(n_components=0.98)],
    ['naive_bayes', BernoulliNB(alpha=0.1)]
])


naive_s.fit(X_train, Y_train)

y_pred = naive_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N
```

```
f1 = 2 * (precision * recall) / (precision + recall)

print("Naive Bayes without SMOTE - PCA")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
Naive Bayes without SMOTE - PCA
-------------------------------------
Precision: 0.7272727272727273
Accuracy: 0.8831168831168831
Recall: 0.8421052631578947
True positive rate: 0.7272727272727273
False positive rate: 0.10909090909090909
F1-score: 0.7804878048780488
```

LDA

with SMOTE

In [534…

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda_s = imbpipeline([
            ['preprocessing', f_preprocessing],
    ['smote', SMOTE(random_state=42, sampling_strategy='minority')],
    ['lda', LinearDiscriminantAnalysis()]  # LDA classifier
])

# Train the model on the SMOTE training data
lda_s.fit(X_train, Y_train)

# Predictions on the test set
y_pred = lda_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("LDA with SMOTE")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
```

```
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
LDA with SMOTE
-------------------------------------
Precision: 1.0
Accuracy: 0.974025974025974
Recall: 0.8947368421052632
True positive rate: 0.7727272727272727
False positive rate: 0.0
F1-score: 0.9444444444444444
```

LDA with SMOTE - Stratified K Folds

In [536…

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
target = df.loc[:, 'Recurred']

print("LDA with SMOTE - Stratified K-Folds")
print("-------------------------------------")

# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = lda_s.predict(X_test)

    # Print evaluation metrics
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
for train_index, test_index in skf.split(df, target):
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
LDA with SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.961038961038961
Fold 1 Precision: 1.0
Fold 1 Recall: 0.8636363636363636
Fold 1 F1 Score: 0.9268292682926829

Fold 2 Accuracy: 0.935064935064935
Fold 2 Precision: 0.9473684210526315
Fold 2 Recall: 0.8181818181818182
Fold 2 F1 Score: 0.8780487804878049

Fold 3 Accuracy: 0.8961038961038961
Fold 3 Precision: 0.8181818181818182
Fold 3 Recall: 0.8181818181818182
Fold 3 F1 Score: 0.8181818181818182

Fold 4 Accuracy: 0.7631578947368421
Fold 4 Precision: 0.6153846153846154
Fold 4 Recall: 0.38095238095238093
Fold 4 F1 Score: 0.47058823529411764

Fold 5 Accuracy: 0.8157894736842105
Fold 5 Precision: 0.6
Fold 5 Recall: 1.0
Fold 5 F1 Score: 0.7499999999999999
```

LDA with SMOTE - Grid Search

In [545…

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    'lda__n_components': [None, 1, min(X_train.shape[1], len(np.unique(Y_train))) - 1]
}


# Perform grid search
grid_search = GridSearchCV(lda_s, param_grid, cv=5, scoring='f1')
grid_search.fit(X_train, Y_train)

print("LDA with SMOTE - Grid Search")
print("-------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best F1-score on validation data:", grid_search.best_score_)

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

# Compute confusion matrix and other metrics
confusion_matrix_result = confusion_matrix(Y_test, y_pred)
# Calculate TP, FP, TN, FN, precision, accuracy, recall, true positive rate, false pos
# Print metrics
```

```
LDA with SMOTE - Grid Search
-------------------------------------
Best parameters found: {'lda__n_components': None}
Best F1-score on validation data: 0.8911976911976913
```

LDA with SMOTE and PCA

In [544…

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda_s = imbpipeline([
            ['preprocessing', f_preprocessing],
    ["pca", PCA(n_components=0.95)],
    ['smote', SMOTE(random_state=42, sampling_strategy='minority')],
    ['lda', LinearDiscriminantAnalysis()]  # LDA classifier
])

# Train the model on the SMOTE training data
lda_s.fit(X_train, Y_train)

# Predictions on the test set
y_pred = lda_s.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("LDA with SMOTE - PCA")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
LDA with SMOTE - PCA
-------------------------------------
Precision: 1.0
Accuracy: 0.974025974025974
Recall: 0.8947368421052632
True positive rate: 0.7727272727272727
False positive rate: 0.0
F1-score: 0.9444444444444444
```

LDA without SMOTE

In [546…

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = imbpipeline([
```

```
            ['preprocessing', f_preprocessing],
    ['lda', LinearDiscriminantAnalysis()]  # LDA classifier
])

# Train the model on the SMOTE training data
lda.fit(X_train, Y_train)

# Predictions on the test set
y_pred = lda.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("LDA without SMOTE")
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

```
LDA without SMOTE
-------------------------------------
Precision: 1.0
Accuracy: 0.961038961038961
Recall: 0.8421052631578947
True positive rate: 0.7272727272727273
False positive rate: 0.0
F1-score: 0.9142857142857143
```

LDA without SMOTE - Stratified K Folds

In [548…

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

skf = StratifiedKFold(n_splits=5)

# Tell the function which column we are going to use as the target
# Use loc() function to extract the data of that column
target = df.loc[:, 'Recurred']

print("LDA without SMOTE - Stratified K-Folds")
print("-------------------------------------")

# Define a function to train the model and evaluate on each fold
def train_model(train, test, fold_no):
    X = recurrence
    y = recurrence_labels
```

```
    X_train = X.loc[train]
    y_train = y.loc[train]
    X_test = X.loc[test]
    y_test = y.loc[test]

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

    # Make predictions
    y_pred = lda.predict(X_test)

    # Print evaluation metrics
    print('Fold', str(fold_no), 'Accuracy:', accuracy_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Precision:', precision_score(y_test, y_pred))
    print('Fold', str(fold_no), 'Recall:', recall_score(y_test, y_pred))
    print('Fold', str(fold_no), 'F1 Score:', f1_score(y_test, y_pred))
    print()

# Perform cross-validation
fold_no = 1
for train_index, test_index in skf.split(df, target):
    # Use the indices provided by split function to extract the corresponding
    # train data & test data
    train_model(train_index, test_index, fold_no)
    fold_no += 1
```

```
LDA without SMOTE - Stratified K-Folds
-------------------------------------
Fold 1 Accuracy: 0.961038961038961
Fold 1 Precision: 1.0
Fold 1 Recall: 0.8636363636363636
Fold 1 F1 Score: 0.9268292682926829

Fold 2 Accuracy: 0.948051948051948
Fold 2 Precision: 1.0
Fold 2 Recall: 0.8181818181818182
Fold 2 F1 Score: 0.9

Fold 3 Accuracy: 0.922077922077922
Fold 3 Precision: 1.0
Fold 3 Recall: 0.7272727272727273
Fold 3 F1 Score: 0.8421052631578948

Fold 4 Accuracy: 0.9605263157894737
Fold 4 Precision: 0.9090909090909091
Fold 4 Recall: 0.9523809523809523
Fold 4 F1 Score: 0.9302325581395349

Fold 5 Accuracy: 0.75
Fold 5 Precision: 0.525
Fold 5 Recall: 1.0
Fold 5 F1 Score: 0.6885245901639345
```

LDA without SMOTE - Grid Search

In [551…

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
```

```
param_grid = {
    'lda__n_components': [None, 1, min(X_train.shape[1], len(np.unique(Y_train))) - 1]
}


# Perform grid search
grid_search = GridSearchCV(lda, param_grid, cv=5, scoring='f1')
grid_search.fit(X_train, Y_train)

print("LDA without SMOTE - Grid Search")
print("--------------------------------------")

# Print best parameters and best score
print("Best parameters found:", grid_search.best_params_)
print("Best F1-score on validation data:", grid_search.best_score_)

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

# Compute confusion matrix and other metrics
confusion_matrix_result = confusion_matrix(Y_test, y_pred)
# Calculate TP, FP, TN, FN, precision, accuracy, recall, true positive rate, false pos
# Print metrics
```

```
LDA without SMOTE - Grid Search
-------------------------------------
Best parameters found: {'lda__n_components': None}
Best F1-score on validation data: 0.9090467276710161
```

LDA without SMOTE, with PCA

In [552…

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = imbpipeline([
            ['preprocessing', f_preprocessing],
    ["pca", PCA(n_components=0.95)],
    ['lda', LinearDiscriminantAnalysis()]  # LDA classifier
])

# Train the model on the SMOTE training data
lda.fit(X_train, Y_train)

# Predictions on the test set
y_pred = lda.predict(X_test)

confusion_matrix_result = confusion_matrix(Y_test, y_pred)

tn, fp, fn, tp = confusion_matrix_result.ravel()

precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
true_positive_rate = tp / P
false_positive_rate = fp / N

f1 = 2 * (precision * recall) / (precision + recall)

print("LDA without SMOTE - PCA")
```

```
print("-------------------------------------")

print("Precision:", precision)
print("Accuracy:", accuracy)
print("Recall:", recall)
print("True positive rate:", true_positive_rate)
print("False positive rate:", false_positive_rate)
print("F1-score:", f1)
```

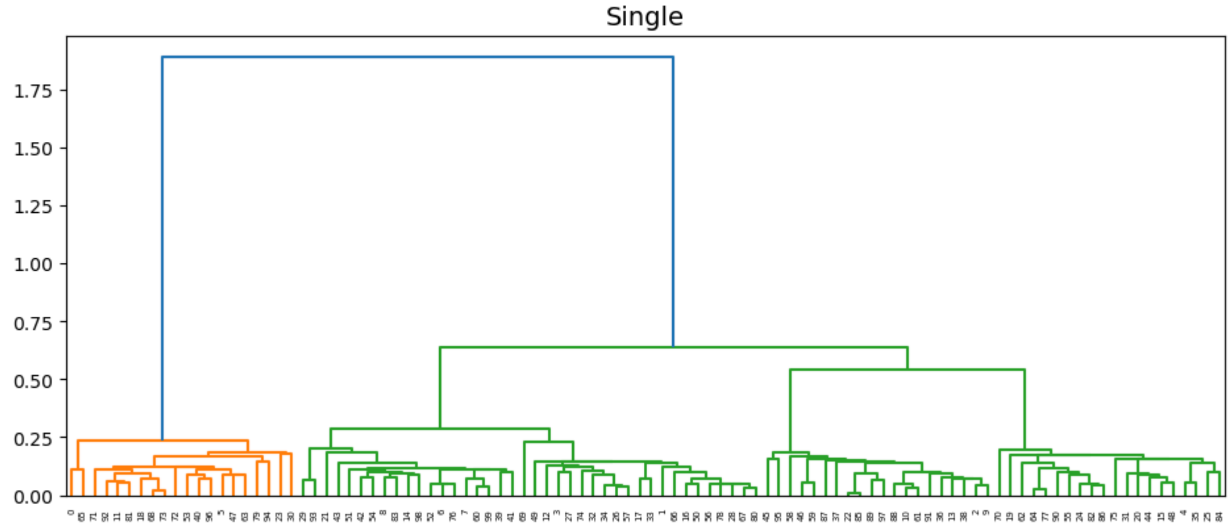
```
LDA without SMOTE - PCA
-------------------------------------
Precision: 1.0
Accuracy: 0.974025974025974
Recall: 0.8947368421052632
True positive rate: 0.7727272727272727
False positive rate: 0.0
F1-score: 0.9444444444444444
```

Unsupervised Learning

Hierachical Clustering

In [195…

```
from sklearn.datasets import make_blobs
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from scipy.cluster.hierarchy import linkage, dendrogram
import matplotlib.pyplot as plt

# Generate synthetic data
X, _ = make_blobs(n_samples=100, centers=5, random_state=42)

# Define the pipeline (although not necessary for hierarchical clustering)
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Scale features if necessary
])

# Fit the pipeline
X1 = pipeline.fit_transform(X)

# Perform hierarchical clustering
Z1 = linkage(X1, method='single', metric='euclidean')
#Z2 = linkage(X1, method='complete', metric='euclidean')
#Z3 = linkage(X1, method='average', metric='euclidean')
#Z4 = linkage(X1, method='ward', metric='euclidean')

plt.figure(figsize=(25, 10))
plt.subplot(2,2,1), dendrogram(Z1), plt.title('Single')
#plt.subplot(3,2,2), dendrogram(Z2), plt.title('Complete')
#plt.subplot(2,2,3), dendrogram(Z3), plt.title('Average')
#plt.subplot(2,2,4), dendrogram(Z4), plt.title('Ward')
plt.show()
```



In [70]:
```
from scipy.cluster.hierarchy import fcluster

f1 = fcluster(Z1, 2, criterion='maxclust')

print(f"Clusters: {f1}")
```

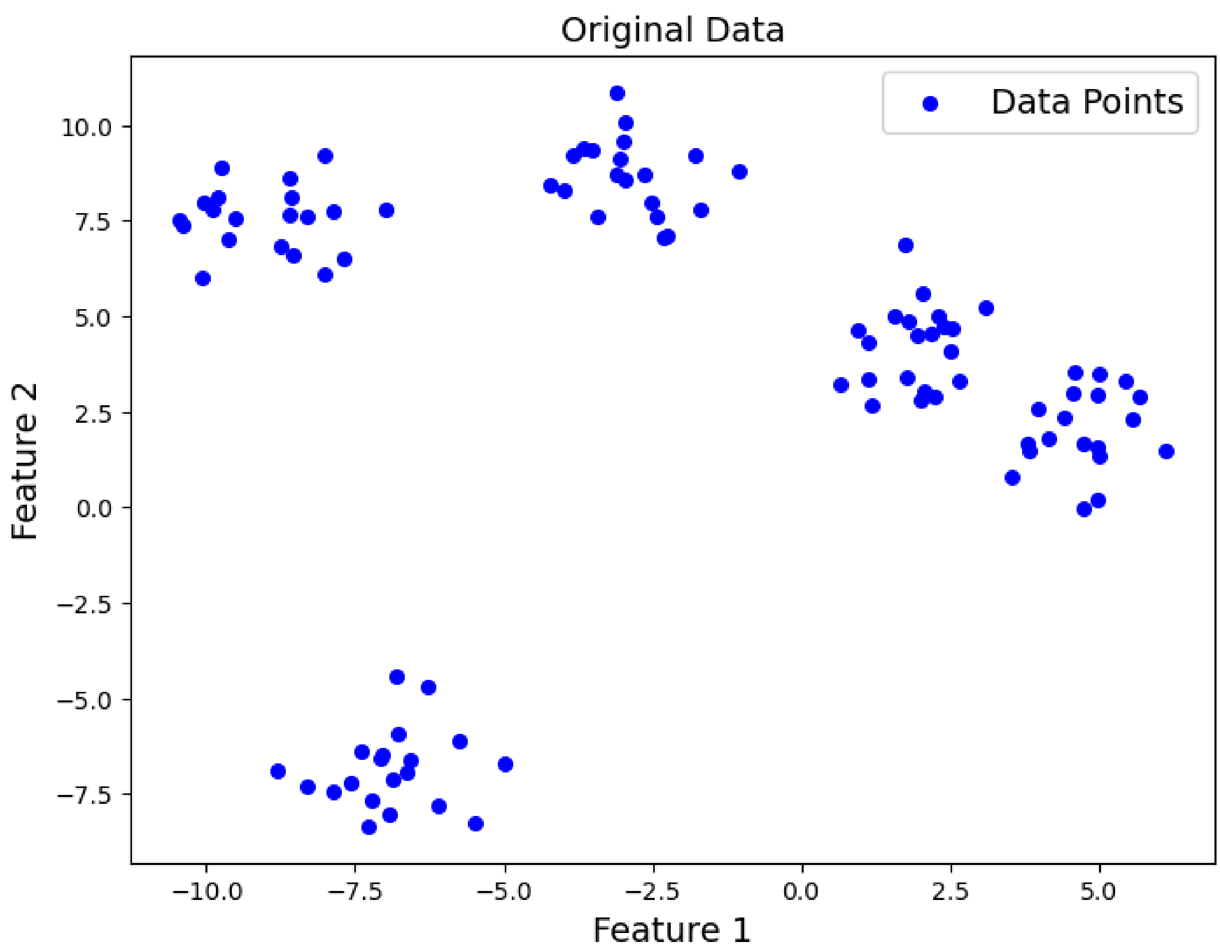
```
Clusters: [1 2 2 2 2 1 2 2 2 2 2 1 2 2 2 2 2 2 1 2 2 2 2 1 2 2 2 2 2 2 1 2 2 2 2 2 2
 2 2 2 1 2 2 2 2 2 2 1 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 1 2 1 2 2 1 2 2 1 1 1
 2 2 2 2 2 1 2 1 2 2 2 2 2 2 2 2 2 2 1 2 1 2 1 2 2 2]
```

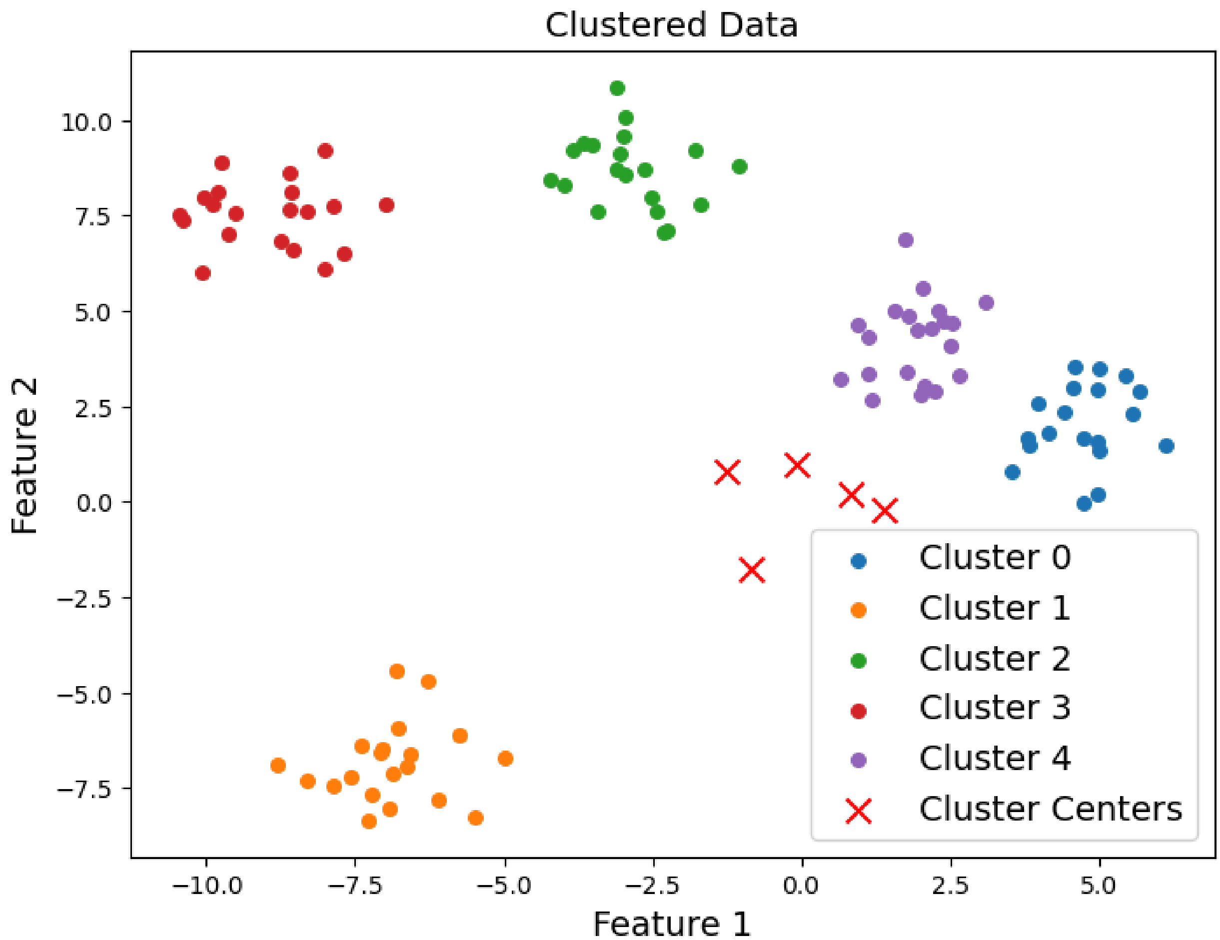
K-Means Clustering

In [71]:
```
from sklearn.datasets import make_blobs
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Generate synthetic data
X, _ = make_blobs(n_samples=100, centers=5, random_state=42)

# Define the pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('kmeans', KMeans(n_clusters=5, random_state=42))
])

# Fit the pipeline
pipeline.fit(X)

# Obtain cluster labels
cluster_labels = pipeline.predict(X)

# Plot the original data
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c='blue', s=30, label='Data Points')
plt.title('Original Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

```
# Plot the clustered data
plt.figure(figsize=(8, 6))
for cluster in range(5):
    plt.scatter(X[cluster_labels == cluster, 0], X[cluster_labels == cluster, 1], s=3(
plt.scatter(pipeline.named_steps['kmeans'].cluster_centers_[:, 0],
            pipeline.named_steps['kmeans'].cluster_centers_[:, 1],
            c='red', s=100, marker='x', label='Cluster Centers')
plt.title('Clustered Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

```
C:\Users\CKY\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarnin
g: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value
of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\CKY\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1382: UserWarnin
g: KMeans is known to have a memory leak on Windows with MKL, when there are less chu
nks than available threads. You can avoid it by setting the environment variable OMP_
NUM_THREADS=1.
  warnings.warn(
```




Clustered Data
Feature 2
Feature 1
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
Cluster 0
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster Centers

```
In [ ]:
```