

## Evaluation using the BeFree corpus

### EUADR dataset

The EU-ADR dataset contains annotations on drugs, diseases, genes and proteins, and associations between them. In this study, we used only GDAs to evaluate the method. Each association is classified according to its level of certainty as positive association (PA), negative association (NA), speculative association (SA); or false association (FA). The EU-ADR corpus is based on 100 MEDLINE abstracts for each association set, and its annotation was conducted by three experts.

---

### imports

```
import tensorflow as tf
from keras.models import load_model
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras_tqdm import TQDMNotebookCallback
import numpy as np
from keras_tqdm import TQDMNotebookCallback
import nltk
import xml.etree.ElementTree as ET
import pandas as pd
import os
import string
from nltk.tokenize import TreebankWordTokenizer
from numpy.random import random_sample
import re
import pickle
from keras.layers import Embedding, Flatten, LSTM, GRU
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from keras.layers import Embedding, Flatten, LSTM
from keras.layers.convolutional import Conv2D, MaxPooling2D
from tensorflow.keras.utils import to_categorical
from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Activation, Input, merge, Conv1D, MaxPooling1D, GlobalMaxPooling1D, Convolution1D
from keras import regularizers
from sklearn.metrics import precision_recall_fscore_support
from sklearn.model_selection import StratifiedKFold
import matplotlib.pyplot as plt
from keras.layers import Concatenate, concatenate
from keras import backend as K
from keras.layers import multiply
from keras.layers import merge
from keras.layers.core import *
from keras.layers.recurrent import LSTM
from keras.models import *
```

### Define Callback functions to generate Measures

```
from keras import backend as K

def f1(y_true, y_pred):
    def recall(y_true, y_pred):
        true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
        possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
        recall = true_positives / (possible_positives + K.epsilon())
        return recall

    def precision(y_true, y_pred):
        true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
        predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
        precision = true_positives / (predicted_positives + K.epsilon())
        return precision
    precision = precision(y_true, y_pred)
```

```
recall = recall(y_true, y_pred)
return 2*((precision*recall)/(precision+recall+K.epsilon()))
```

## ✓ Experiments to reproduce the results of Table 8

### ✓ Load Prerocssed Data

```
with open('../data/pickles/befree_EUADR_2class_PubMed-and-PMC-w2v.pickle', 'rb') as handle:
```

```
W_train = pickle.load(handle)
print("W_train",len(W_train))
d1_train = pickle.load(handle)
print("d1_train",len(d1_train))
d2_train = pickle.load(handle)
print("d2_train",len(d2_train))
Y_train = pickle.load(handle)
print("Y_train",len(d2_train))
Tr_word_list = pickle.load(handle)
print("Tr_word_list",len(d2_train))
word_vectors = pickle.load(handle)
print("word_vectors",len(word_vectors))
word_dict = pickle.load(handle)
print("word_dict",len(word_dict))
d1_dict = pickle.load(handle)
print("d1_dict",len(d1_dict))
d2_dict = pickle.load(handle)
print("d2_dict",len(d2_dict))
label_dict = pickle.load(handle)
print("label_dict",len(label_dict))
MAX_SEQUENCE_LENGTH = pickle.load(handle)
print("MAX_SEQUENCE_LENGTH",MAX_SEQUENCE_LENGTH)
```

```
→ W_train 355
d1_train 355
d2_train 355
Y_train 355
Tr_word_list 355
word_vectors 1355
word_dict 1355
d1_dict 169
d2_dict 171
label_dict 4
MAX_SEQUENCE_LENGTH 102
```

### ✓ Create Position Embedding Vectors

```
import keras
from keras_pos_embd import TrigPosEmbedding

model = keras.models.Sequential()
model.add(TrigPosEmbedding(
    input_shape=(None,),
    output_dim=20, # The dimension of embeddings.
    mode=TrigPosEmbedding.MODE_EXPAND, # Use `expand` mode
    name='Pos-Embd',
))
model.compile('adam', keras.losses.mae, {})
model.summary()

d1_train_embedded=model.predict(d1_train)

d1_train_embedded.shape

d2_train_embedded=model.predict(d2_train)

d2_train_embedded.shape

→ Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
Pos-Embd (TrigPosEmbedding)	(None, None, 20)	0

=====  
 Total params: 0  
 Trainable params: 0  
 Non-trainable params: 0  
 =====  
 (355, 102, 20)

## ▼ Prepare Word Embedding Layer

```

EMBEDDING_DIM=word_vectors.shape[1]
embedding_matrix=word_vectors
print(EMBEDDING_DIM)
print(len(embedding_matrix))

param='binary'

def create_embedding_layer(l2_reg=0.01,use_pretrained=True,is_trainable=False):

    if use_pretrained:
        return Embedding(len(word_dict) , EMBEDDING_DIM,weights=[embedding_matrix],input_length=MAX_SEQUENCE_LENGTH,trainable=is_trainable,

    else:
        return Embedding(len(word_dict) , EMBEDDING_DIM,input_length=MAX_SEQUENCE_LENGTH)
  
```

↔ 200  
1355

## ▼ Prepare Attention Mechanism

```

INPUT_DIM = 2
TIME_STEPS = MAX_SEQUENCE_LENGTH

def attentionNew(inputs):
    inputs = Lambda(lambda x: tf.keras.backend.sigmoid(x))(inputs)
    input_dim = int(inputs.shape[2])
    a = Permute((2, 1))(inputs)
    a = Dense(TIME_STEPS, activation='softmax')(a)
    a_probs = Permute((2, 1))(a)
    output_attention_mul = multiply([inputs, a_probs])
    output_attention_mul = Lambda(lambda x: tf.keras.backend.sigmoid(x))(output_attention_mul)
    return output_attention_mul
  
```

## ▼ Create the Model

```

from keras.optimizers import Adam
def build_model_positionAttention():
    sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
    embedding_layer = create_embedding_layer(use_pretrained=True, is_trainable=False)
    embedded_sequences = embedding_layer(sequence_input)

    pos_embedd_1 = Input(shape=(MAX_SEQUENCE_LENGTH, 20), dtype='float32')
    pos_embedd_2 = Input(shape=(MAX_SEQUENCE_LENGTH, 20), dtype='float32')

    # Concatenate embeddings and position encodings
    embedded_sequences = concatenate([embedded_sequences, pos_embedd_1, pos_embedd_2])

    # Convolutional layers
    x = Conv1D(256, 5, activation='relu')(embedded_sequences)
    x = MaxPooling1D(3)(x)
    x = Dropout(0.4)(x)
    x = Conv1D(128, 3, activation='relu')(x)
    x = MaxPooling1D(3)(x)
    x = Dropout(0.4)(x)
  
```

```

conv_sequence_7 = GlobalMaxPooling1D()(x) # Shape: (None, 64)

# Bidirectional RNN layers
forward = GRU(100, return_sequences=True, recurrent_dropout=0.1)(embedded_sequences)
backward = GRU(100, return_sequences=True, go_backwards=True, recurrent_dropout=0.05)(embedded_sequences)
lstm_gru_sequence = concatenate([forward, backward], axis=-1) # Shape: (None, MAX_SEQUENCE_LENGTH, 200)

# Apply attention mechanism
attention_output = attentionNew(lstm_gru_sequence) # Shape: (None, MAX_SEQUENCE_LENGTH, 200)
attention_pooled = GlobalMaxPooling1D()(attention_output) # Shape: (None, 200)

# Merge CNN and attention-enhanced RNN outputs
merge = concatenate([conv_sequence_7, attention_pooled]) # Shape: (None, 264)

# Fully connected layers
x = Dropout(0.4)(merge)
x = Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.05))(x)
x = Dropout(0.4)(x)
preds = Dense(2, activation='softmax')(x)
optimizer = Adam(learning_rate=0.001)
model = Model(inputs=[sequence_input, pos_embedd_1, pos_embedd_2], outputs=preds)

model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['acc', f1])

return model

```

```

model = build_model_positionAttention()
model.summary()

```

Model: "model\_2"

Layer (type)	Output Shape	Param #	Connected to
input_4 (InputLayer)	(None, 102)	0	
embedding_2 (Embedding)	(None, 102, 200)	271000	input_4[0][0]
input_5 (InputLayer)	(None, 102, 20)	0	
input_6 (InputLayer)	(None, 102, 20)	0	
concatenate_4 (Concatenate)	(None, 102, 240)	0	embedding_2[0][0] input_5[0][0] input_6[0][0]
gru_3 (GRU)	(None, 102, 100)	102300	concatenate_4[0][0]
gru_4 (GRU)	(None, 102, 100)	102300	concatenate_4[0][0]
concatenate_5 (Concatenate)	(None, 102, 200)	0	gru_3[0][0] gru_4[0][0]
conv1d_3 (Conv1D)	(None, 98, 256)	307456	concatenate_4[0][0]
lambda_3 (Lambda)	(None, 102, 200)	0	concatenate_5[0][0]
max_pooling1d_3 (MaxPooling1D)	(None, 32, 256)	0	conv1d_3[0][0]
permute_3 (Permute)	(None, 200, 102)	0	lambda_3[0][0]
dropout_5 (Dropout)	(None, 32, 256)	0	max_pooling1d_3[0][0]
dense_4 (Dense)	(None, 200, 102)	10506	permute_3[0][0]
conv1d_4 (Conv1D)	(None, 30, 128)	98432	dropout_5[0][0]
permute_4 (Permute)	(None, 102, 200)	0	dense_4[0][0]
max_pooling1d_4 (MaxPooling1D)	(None, 10, 128)	0	conv1d_4[0][0]
multiply_2 (Multiply)	(None, 102, 200)	0	lambda_3[0][0] permute_4[0][0]
dropout_6 (Dropout)	(None, 10, 128)	0	max_pooling1d_4[0][0]
lambda_4 (Lambda)	(None, 102, 200)	0	multiply_2[0][0]
global_max_pooling1d_3 (GlobalM	(None, 128)	0	dropout_6[0][0]
global_max_pooling1d_4 (GlobalM	(None, 200)	0	lambda_4[0][0]

concatenate_6 (Concatenate)	(None, 328)	0	global_max_pooling1d_3[0][0] global_max_pooling1d_4[0][0]
dropout_7 (Dropout)	(None, 328)	0	concatenate_6[0][0]
dense_5 (Dense)	(None, 256)	84224	dropout_7[0][0]

## ▼ Run the Evaluation using 10 fold Cross Validation

```
validation_split_rate=0.1
skf = StratifiedKFold(n_splits=5, random_state=None)
Y = [np.argmax(y, axis=None, out=None) for y in Y_train]
#print(len(Y))
all_histories=[]
for trI, teI in skf.split(W_train,Y):
    train_index =trI
    test_index  =teI
trainRate = (len(train_index)/len(Y))*100
testRate = (len(test_index)/len(Y))*100
print ("TrainRate:{:.2f}% testRate:{:.2f}% validation:{:.2f}% ".format(trainRate,testRate, trainRate*validation_split_rate))
#print(train_index, test_index)
X_train, X_test = W_train[train_index], W_train[test_index]
pos_train1, pos_test1 = d1_train_embedded[train_index], d1_train_embedded[test_index]
pos_train2, pos_test2 = d2_train_embedded[train_index], d2_train_embedded[test_index]
y_train, y_test = Y_train[train_index], Y_train[test_index]
```

```
# # Saving the training data split as a pickle file
# training_data = {
#     'X_train': X_train,
#     'pos_train1': pos_train1,
#     'pos_train2': pos_train2,
#     'y_train': y_train
# }
```

```
# with open('training_data.pkl', 'wb') as f:
#     pickle.dump(training_data, f)
```

```
# # Saving the testing data split as a pickle file
# testing_data = {
#     'X_test': X_test,
#     'pos_test1': pos_test1,
#     'pos_test2': pos_test2,
#     'y_test': y_test
# }
```

```
# with open('testing_data.pkl', 'wb') as f:
#     pickle.dump(testing_data, f)
```

 TrainRate:80.00% testRate:20.00% validation:8.00%

```
# Load the training data from the pickle file
with open('training_data.pkl', 'rb') as f:
    train_data = pickle.load(f)
```

```
# Load the testing data from the pickle file
with open('testing_data.pkl', 'rb') as f:
    test_data = pickle.load(f)
```

```
# Extract data from the loaded dictionaries
X_train = train_data['X_train']
pos_train1 = train_data['pos_train1']
pos_train2 = train_data['pos_train2']
y_train = train_data['y_train']
```


```
X_test = test_data['X_test']
```

```
pos_test1 = test_data['pos_test1']
pos_test2 = test_data['pos_test2']
y_test = test_data['y_test']
```

```
print(X_train.shape)
```

 (284, 102)

```
epochs =70
batch_size =32
validation_split_rate=0.1
history=model.fit([X_train,pos_train1,pos_train2], y_train,validation_split=validation_split_rate ,epochs=epochs, batch_size=batch_size,vert
```

 Train on 255 samples, validate on 29 samples

```
Epoch 1/70
255/255 [=====] - 2s 7ms/step - loss: 593.5716 - acc: 0.5137 - f1: 0.5139 - val_loss: 590.9819 - val_acc: 0.
Epoch 2/70
255/255 [=====] - 1s 3ms/step - loss: 590.0521 - acc: 0.5255 - f1: 0.5252 - val_loss: 588.0590 - val_acc: 0.
Epoch 3/70
255/255 [=====] - 1s 3ms/step - loss: 587.1259 - acc: 0.5922 - f1: 0.5916 - val_loss: 585.4492 - val_acc: 0.
Epoch 4/70
255/255 [=====] - 1s 3ms/step - loss: 584.6716 - acc: 0.5647 - f1: 0.5655 - val_loss: 583.1022 - val_acc: 0.
Epoch 5/70
255/255 [=====] - 1s 3ms/step - loss: 582.6348 - acc: 0.6078 - f1: 0.6077 - val_loss: 581.4643 - val_acc: 0.
Epoch 6/70
255/255 [=====] - 1s 3ms/step - loss: 580.9631 - acc: 0.6157 - f1: 0.6157 - val_loss: 579.8991 - val_acc: 0.
Epoch 7/70
255/255 [=====] - 1s 3ms/step - loss: 579.6339 - acc: 0.6157 - f1: 0.6159 - val_loss: 578.7748 - val_acc: 0.
Epoch 8/70
255/255 [=====] - 1s 4ms/step - loss: 578.5844 - acc: 0.6588 - f1: 0.6584 - val_loss: 577.8549 - val_acc: 0.
Epoch 9/70
255/255 [=====] - 1s 4ms/step - loss: 577.7364 - acc: 0.6549 - f1: 0.6549 - val_loss: 577.0925 - val_acc: 0.
Epoch 10/70
255/255 [=====] - 1s 4ms/step - loss: 577.0823 - acc: 0.6510 - f1: 0.6510 - val_loss: 576.4881 - val_acc: 0.
Epoch 11/70
255/255 [=====] - 1s 3ms/step - loss: 576.5616 - acc: 0.6627 - f1: 0.6627 - val_loss: 576.1151 - val_acc: 0.
Epoch 12/70
255/255 [=====] - 1s 3ms/step - loss: 576.1274 - acc: 0.6706 - f1: 0.6707 - val_loss: 575.7488 - val_acc: 0.
Epoch 13/70
255/255 [=====] - 1s 3ms/step - loss: 575.8197 - acc: 0.6863 - f1: 0.6864 - val_loss: 575.3936 - val_acc: 0.
Epoch 14/70
255/255 [=====] - 1s 3ms/step - loss: 575.5617 - acc: 0.6706 - f1: 0.6705 - val_loss: 575.2310 - val_acc: 0.
Epoch 15/70
255/255 [=====] - 1s 3ms/step - loss: 575.3317 - acc: 0.6980 - f1: 0.6977 - val_loss: 574.9706 - val_acc: 0.
Epoch 16/70
255/255 [=====] - 1s 3ms/step - loss: 575.1542 - acc: 0.6667 - f1: 0.6660 - val_loss: 575.0431 - val_acc: 0.
Epoch 17/70
255/255 [=====] - 1s 3ms/step - loss: 575.0291 - acc: 0.6588 - f1: 0.6590 - val_loss: 574.6978 - val_acc: 0.
Epoch 18/70
255/255 [=====] - 1s 3ms/step - loss: 574.8964 - acc: 0.6980 - f1: 0.6982 - val_loss: 574.7197 - val_acc: 0.
Epoch 19/70
255/255 [=====] - 1s 3ms/step - loss: 574.7648 - acc: 0.7098 - f1: 0.7090 - val_loss: 574.6496 - val_acc: 0.
Epoch 20/70
255/255 [=====] - 1s 3ms/step - loss: 574.7043 - acc: 0.6745 - f1: 0.6749 - val_loss: 574.4107 - val_acc: 0.
Epoch 21/70
255/255 [=====] - 1s 3ms/step - loss: 574.6216 - acc: 0.7490 - f1: 0.7494 - val_loss: 574.6079 - val_acc: 0.
Epoch 22/70
255/255 [=====] - 1s 3ms/step - loss: 574.5677 - acc: 0.7373 - f1: 0.7374 - val_loss: 574.4766 - val_acc: 0.
Epoch 23/70
255/255 [=====] - 1s 3ms/step - loss: 574.4445 - acc: 0.7765 - f1: 0.7767 - val_loss: 574.3586 - val_acc: 0.
Epoch 24/70
255/255 [=====] - 1s 3ms/step - loss: 574.4054 - acc: 0.7804 - f1: 0.7807 - val_loss: 574.3237 - val_acc: 0.
Epoch 25/70
255/255 [=====] - 1s 3ms/step - loss: 574.3577 - acc: 0.7412 - f1: 0.7417 - val_loss: 574.2224 - val_acc: 0.
Epoch 26/70
255/255 [=====] - 1s 3ms/step - loss: 574.3245 - acc: 0.7843 - f1: 0.7841 - val_loss: 574.4989 - val_acc: 0.
Epoch 27/70
255/255 [=====] - 1s 3ms/step - loss: 574.3015 - acc: 0.7725 - f1: 0.7729 - val_loss: 574.3706 - val_acc: 0.
Epoch 28/70
255/255 [=====] - 1s 3ms/step - loss: 574.2547 - acc: 0.8000 - f1: 0.7996 - val_loss: 574.3383 - val_acc: 0.
```

```
import matplotlib.pyplot as plt
```

```
# Training & Validation accuracy
train_loss = history.history['loss']
val_loss = history.history['val_loss']
train_acc = history.history['acc']
val_acc = history.history['val_acc']
```

```

epochs = len(train_loss)

xc = range(epochs)

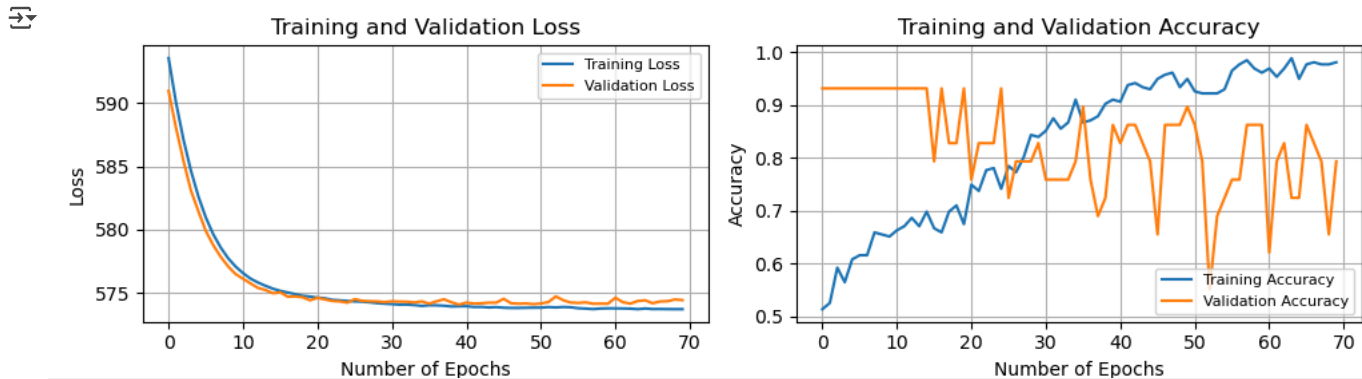
plt.figure(figsize=(10, 3))

# Loss subplot
plt.subplot(1, 2, 1)
plt.plot(xc, train_loss, label='Training Loss')
plt.plot(xc, val_loss, label='Validation Loss')
plt.xlabel('Number of Epochs', fontsize=10)
plt.ylabel('Loss', fontsize=10)
plt.title('Training and Validation Loss', fontsize=12)
plt.legend(fontsize=8)
plt.grid(True)

# Accuracy subplot
plt.subplot(1, 2, 2)
plt.plot(xc, train_acc, label='Training Accuracy')
plt.plot(xc, val_acc, label='Validation Accuracy')
plt.xlabel('Number of Epochs', fontsize=10)
plt.ylabel('Accuracy', fontsize=10)
plt.title('Training and Validation Accuracy', fontsize=12)
plt.legend(fontsize=8, loc='lower right') # Change position to lower right
plt.grid(True)

plt.tight_layout()
plt.show()

```



```

predicted = np.argmax(model.predict([X_test,pos_test1,pos_test2]), axis=1)
y_test_to_label= np.argmax(y_test, axis=1)
prec, reca, fscore, sup = precision_recall_fscore_support(y_test_to_label, predicted, average=param)

```

```

# Generate the classification report as a dictionary
report_dict = classification_report(y_test_to_label, predicted, output_dict=True)

# Create a new dictionary to hold the formatted values
formatted_report_dict = {}

# Iterate over the items in the report dictionary
for key, value in report_dict.items():
    if isinstance(value, dict):
        # Format the nested dictionary values
        formatted_report_dict[key] = {sub_key: f"{sub_value:.4f}" for sub_key, sub_value in value.items()}
    else:
        # Format the top-level dictionary values
        formatted_report_dict[key] = f"{value:.4f}"

# Create a string representation of the formatted dictionary
formatted_report_str = classification_report(y_test_to_label, predicted, digits=4)

# Print the formatted classification report
print(formatted_report_str)

```

```

print(" Precision:{:.2f}% Recall:{:.2f}% Fscore:{:.2f}% ".format(prec*100, reca*100, fscore*100))

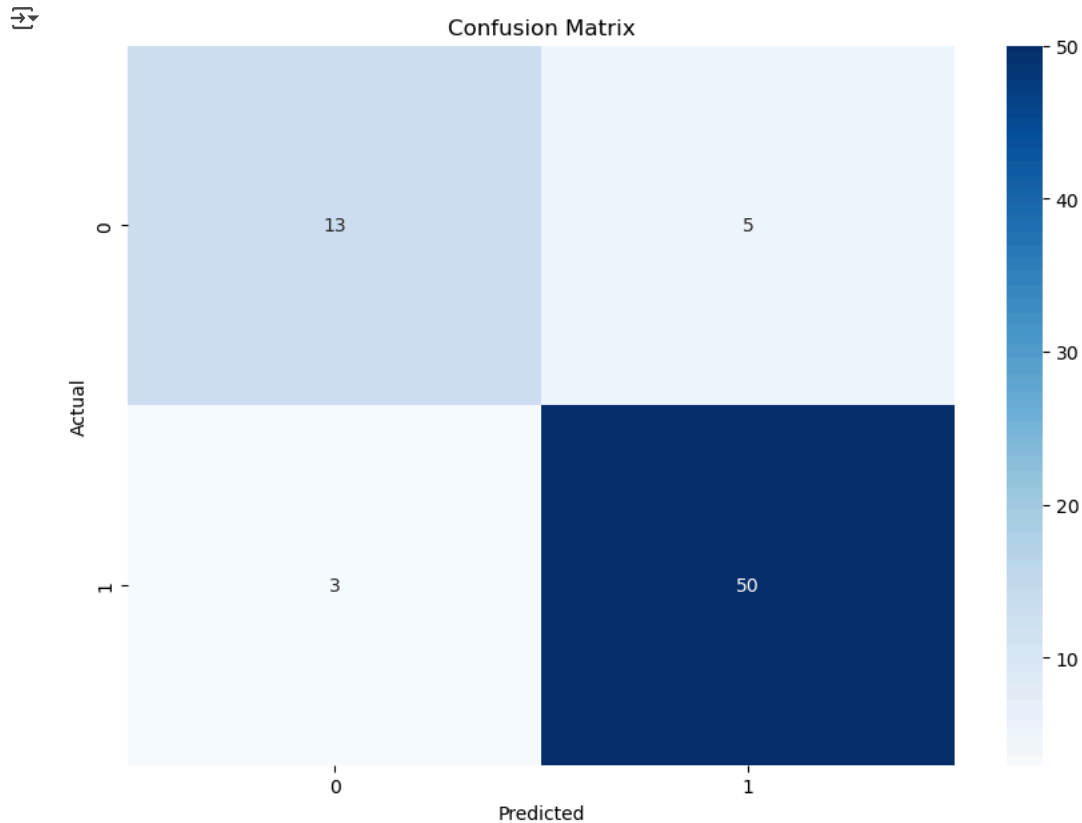
```

	precision	recall	f1-score	support
0	0.8125	0.7222	0.7647	18
1	0.9091	0.9434	0.9259	53
accuracy			0.8873	71
macro avg	0.8608	0.8328	0.8453	71
weighted avg	0.8846	0.8873	0.8851	71

Precision:90.91% Recall:94.34% Fscore:92.59%

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report, confusion_matrix, precision_recall_fscore_support
from sklearn.model_selection import StratifiedKFold
# Calculate and visualize the confusion matrix
cm = confusion_matrix(y_test_to_label, predicted)
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['0', '1'], yticklabels=['0', '1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()

# Print precision, recall, and f-score
prec, reca, fscore, sup = precision_recall_fscore_support(y_test_to_label, predicted, average=param)
print(" Precision:{:.2f}% Recall:{:.2f}% Fscore:{:.2f}% ".format(prec*100, reca*100, fscore*100))
```



Start coding or [generate](#) with AI.



