chi_sq_87 9010 split .05 threshold

January 2, 2023

[ ]:
```
# Importing the packages
import sys
import numpy as np
np.set_printoptions(threshold=sys.maxsize)
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import sklearn
import random
from sklearn.metrics import␣
↪confusion_matrix,accuracy_score,classification_report,RocCurveDisplay,ConfusionMatrixDispla
```

[ ]:
```
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)
```

[ ]:
```
# Importing the dataset
df = pd.read_csv('dataset_phishing.csv')
df.drop(['url'], axis=1, inplace=True)
#df.head(50)
```

[ ]:
```
# if your dataset contains missing value, check which column has missing values
#df.isnull().sum()
```

[ ]:
```
#df.dropna(inplace=True)
```

[ ]:
```
from sklearn import preprocessing


col = [df.columns[-1]]

lab_en= preprocessing.LabelEncoder()

for c in col:
    df[c]= lab_en.fit_transform(df[c])

#df.head(50)
```

[ ]:
```
a=len(df[df.status==0])
b=len(df[df.status==1])
```

[ ]:
```
print("Count of Legitimate Websites = ", a)
print("Count of Phishy Websites = ", b)
```

```
Count of Legitimate Websites =  5715
Count of Phishy Websites =  5715
```

[ ]:
```
X = df.drop(['status'], axis=1, inplace=False)
#X.head()
#same work
##inplace true modifies the og data & does not return anything
##inplace false does not modify og data but returns something whoch we store in␣
↪a var
# X= df.drop(columns='Result')
# X.head()
```

[ ]:
```
#df.head()
```

[ ]:
```
y = df['status']
y = pd.DataFrame(y)
y.head()
```

[ ]:

	status
0	0
1	1
2	1
3	0
4	0

[ ]:
```
# separate dataset into train and test
from cProfile import label
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.1,
    random_state=10)

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[ ]:
```
((10287, 87), (1143, 87), (10287, 1), (1143, 1))
```

[ ]:
```
#X_test.head()
```

[ ]:
```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler= MinMaxScaler()

col_X_train = [X_train.columns[:]]

for c in col_X_train:
    X_train[c]= scaler.fit_transform(X_train[c])

#X_train.head(5)
```

[ ]:
```
col_X_test = [X_test.columns[:]]

for c in col_X_test:
    X_test[c]= scaler.transform(X_test[c])

#X_test.head(5)
```

[ ]:
```
#perform chi square test
from sklearn.feature_selection import chi2
f_p_values = chi2(X_train,y_train)
```

[ ]:
```
f_p_values
```

[ ]:
```
(array([2.49086941e+01, 1.91339460e+01, 8.97009539e+02, 2.43403940e+01,
        9.85647157e+00, 5.73948803e+01, 2.83457054e+02, 6.86570320e+01,
                   nan, 1.03651303e+02, 3.65915998e+00, 7.72492082e+00,
        1.63724029e+00, 3.09648967e+01, 8.02024528e+00, 3.24314357e+01,
        3.46090971e-02, 3.10790240e+01, 3.17468042e+00, 1.75299097e-02,
        5.59710793e+02, 4.69312899e+01, 5.21045343e+01, 2.39398201e+01,
        5.33293572e+01, 2.69755396e+02, 2.22997514e+02, 3.00759198e+00,
        1.51520552e+00, 6.15646214e+01, 4.19686819e+02, 1.56746719e+02,
        2.20365704e+01, 3.89288391e+02, 2.45616159e+00, 9.72153729e+01,
        3.19403679e-06, 7.43966713e-01, 3.20809811e+01, 2.24252558e+01,
        1.90190574e-01, 1.19848608e+00, 5.17682395e+01, 5.91156717e+00,
        1.80796886e+01, 7.01881806e+00, 2.83833213e+01, 8.95421273e+00,
        1.95910834e+01, 1.98195604e+01, 2.56743862e+02, 8.91888419e+01,
        4.61164104e+01, 3.94492445e+01, 1.30815312e+02, 1.99833674e+02,
        8.44119645e+01, 1.45470196e+02, 2.92454930e+01,            nan,
        5.00732025e+00,            nan, 5.09474832e+01,            nan,
        4.96325801e+00, 4.07851055e+00, 1.28869275e+02, 1.16519976e+02,
                   nan, 1.90581495e+02, 1.28196906e+02,            nan,
        2.55629024e+00, 3.77525805e+01, 1.32251694e+02, 8.83993374e-02,
        2.90792032e-01, 3.82790955e+02, 2.68947309e+02, 1.77967864e+02,
        4.61631093e+01, 1.17163926e+01, 2.12755623e+02, 1.29597864e+01,
        1.51304941e+02, 2.54269815e+03, 5.37638582e+02]),
 array([6.01107263e-007, 1.21858145e-005, 4.38442116e-197, 8.07283085e-007,
        1.69235373e-003, 3.56534141e-014, 1.32513553e-063, 1.17166165e-016,
                    nan, 2.41248633e-024, 5.57617402e-002, 5.44637484e-003,
```

```
       2.00704533e-001, 2.62737698e-008, 4.62573052e-003, 1.23472858e-008,
       8.52417154e-001, 2.47734112e-008, 7.47878410e-002, 8.94667468e-001,
       9.71697877e-124, 7.35195142e-012, 5.26232734e-013, 9.93944687e-007,
       2.82059660e-013, 1.28297977e-060, 2.00709991e-050, 8.28753242e-002,
       2.18346485e-001, 4.28448194e-015, 2.85505080e-093, 5.81427253e-036,
       2.67504758e-006, 1.18248521e-086, 1.17064540e-001, 6.21819997e-023,
       9.98574032e-001, 3.88393154e-001, 1.47877927e-008, 2.18482080e-006,
       6.62758052e-001, 2.73624471e-001, 6.24539431e-013, 1.50417845e-002,
       2.11849315e-005, 8.06574550e-003, 9.95190127e-008, 2.76830461e-003,
       9.59157935e-006, 8.51072671e-006, 8.79616539e-058, 3.58867374e-021,
       1.11431172e-011, 3.36697985e-010, 2.71749051e-030, 2.27053398e-045,
       4.01709482e-020, 1.69498400e-033, 6.37643694e-008,             nan,
       2.52403488e-002,             nan, 9.48705818e-013,             nan,
       2.58913819e-002, 4.34319278e-002, 7.24353892e-030, 3.65677363e-027,
                   nan, 2.37358184e-043, 1.01641916e-029,             nan,
       1.09856096e-001, 8.03104131e-010, 1.31802918e-030, 7.66222015e-001,
       5.89713424e-001, 3.07137070e-085, 1.92459886e-060, 1.34626829e-040,
       1.08806539e-011, 6.19519327e-004, 3.44100373e-048, 3.18253492e-004,
       8.98956614e-035, 0.00000000e+000, 6.15397965e-119]))
```

[ ]:
```
#The less the p_values the more important that feature is
p_values = pd.Series(f_p_values[1])
p_values.index = X_train.columns
p_values
```

[ ]:
```
length_url                  6.011073e-07
length_hostname             1.218581e-05
ip                         4.384421e-197
nb_dots                     8.072831e-07
nb_hyphens                  1.692354e-03
nb_at                       3.565341e-14
nb_qm                       1.325136e-63
nb_and                      1.171662e-16
nb_or                                NaN
nb_eq                       2.412486e-24
nb_underscore               5.576174e-02
nb_tilde                    5.446375e-03
nb_percent                  2.007045e-01
nb_slash                    2.627377e-08
nb_star                     4.625731e-03
nb_colon                    1.234729e-08
nb_comma                    8.524172e-01
nb_semicolumn               2.477341e-08
nb_dollar                   7.478784e-02
nb_space                    8.946675e-01
nb_www                     9.716979e-124
nb_com                      7.351951e-12
```

nb_dslash	5.262327e-13
http_in_path	9.939447e-07
https_token	2.820597e-13
ratio_digits_url	1.282980e-60
ratio_digits_host	2.007100e-50
punycode	8.287532e-02
port	2.183465e-01
tld_in_path	4.284482e-15
tld_in_subdomain	2.855051e-93
abnormal_subdomain	5.814273e-36
nb_subdomains	2.675048e-06
prefix_suffix	1.182485e-86
random_domain	1.170645e-01
shortening_service	6.218200e-23
path_extension	9.985740e-01
nb_redirection	3.883932e-01
nb_external_redirection	1.478779e-08
length_words_raw	2.184821e-06
char_repeat	6.627581e-01
shortest_words_raw	2.736245e-01
shortest_word_host	6.245394e-13
shortest_word_path	1.504178e-02
longest_words_raw	2.118493e-05
longest_word_host	8.065745e-03
longest_word_path	9.951901e-08
avg_words_raw	2.768305e-03
avg_word_host	9.591579e-06
avg_word_path	8.510727e-06
phish_hints	8.796165e-58
domain_in_brand	3.588674e-21
brand_in_subdomain	1.114312e-11
brand_in_path	3.366980e-10
suspecious_tld	2.717491e-30
statistical_report	2.270534e-45
nb_hyperlinks	4.017095e-20
ratio_intHyperlinks	1.694984e-33
ratio_extHyperlinks	6.376437e-08
ratio_nullHyperlinks	NaN
nb_extCSS	2.524035e-02
ratio_intRedirection	NaN
ratio_extRedirection	9.487058e-13
ratio_intErrors	NaN
ratio_extErrors	2.589138e-02
login_form	4.343193e-02
external_favicon	7.243539e-30
links_in_tags	3.656774e-27
submit_email	NaN

```
ratio_intMedia                  2.373582e-43
ratio_extMedia                  1.016419e-29
sfh                                      NaN
iframe                          1.098561e-01
popup_window                    8.031041e-10
safe_anchor                     1.318029e-30
onmouseover                     7.662220e-01
right_clic                      5.897134e-01
empty_title                     3.071371e-85
domain_in_title                 1.924599e-60
domain_with_copyright           1.346268e-40
whois_registered_domain         1.088065e-11
domain_registration_length      6.195193e-04
domain_age                      3.441004e-48
web_traffic                     3.182535e-04
dns_record                      8.989566e-35
google_index                    0.000000e+00
page_rank                      6.153980e-119
dtype: float64
```

[ ]:
```python
#sort p_values to check which feature has the lowest values
p_values = p_values.sort_values(ascending = False)
p_values
```

[ ]:
```
path_extension                  9.985740e-01
nb_space                        8.946675e-01
nb_comma                        8.524172e-01
onmouseover                     7.662220e-01
char_repeat                     6.627581e-01
right_clic                      5.897134e-01
nb_redirection                  3.883932e-01
shortest_words_raw              2.736245e-01
port                            2.183465e-01
nb_percent                      2.007045e-01
random_domain                   1.170645e-01
iframe                          1.098561e-01
punycode                        8.287532e-02
nb_dollar                       7.478784e-02
nb_underscore                   5.576174e-02
login_form                      4.343193e-02
ratio_extErrors                 2.589138e-02
nb_extCSS                       2.524035e-02
shortest_word_path              1.504178e-02
longest_word_host               8.065745e-03
nb_tilde                        5.446375e-03
nb_star                         4.625731e-03
avg_words_raw                   2.768305e-03
```

nb_hyphens	1.692354e-03
domain_registration_length	6.195193e-04
web_traffic	3.182535e-04
longest_words_raw	2.118493e-05
length_hostname	1.218581e-05
avg_word_host	9.591579e-06
avg_word_path	8.510727e-06
nb_subdomains	2.675048e-06
length_words_raw	2.184821e-06
http_in_path	9.939447e-07
nb_dots	8.072831e-07
length_url	6.011073e-07
longest_word_path	9.951901e-08
ratio_extHyperlinks	6.376437e-08
nb_slash	2.627377e-08
nb_semicolumn	2.477341e-08
nb_external_redirection	1.478779e-08
nb_colon	1.234729e-08
popup_window	8.031041e-10
brand_in_path	3.366980e-10
brand_in_subdomain	1.114312e-11
whois_registered_domain	1.088065e-11
nb_com	7.351951e-12
ratio_extRedirection	9.487058e-13
shortest_word_host	6.245394e-13
nb_dslash	5.262327e-13
https_token	2.820597e-13
nb_at	3.565341e-14
tld_in_path	4.284482e-15
nb_and	1.171662e-16
nb_hyperlinks	4.017095e-20
domain_in_brand	3.588674e-21
shortening_service	6.218200e-23
nb_eq	2.412486e-24
links_in_tags	3.656774e-27
ratio_extMedia	1.016419e-29
external_favicon	7.243539e-30
suspecious_tld	2.717491e-30
safe_anchor	1.318029e-30
ratio_intHyperlinks	1.694984e-33
dns_record	8.989566e-35
abnormal_subdomain	5.814273e-36
domain_with_copyright	1.346268e-40
ratio_intMedia	2.373582e-43
statistical_report	2.270534e-45
domain_age	3.441004e-48
ratio_digits_host	2.007100e-50

```
phish_hints                   8.796165e-58
domain_in_title               1.924599e-60
ratio_digits_url              1.282980e-60
nb_qm                         1.325136e-63
empty_title                   3.071371e-85
prefix_suffix                 1.182485e-86
tld_in_subdomain              2.855051e-93
page_rank                    6.153980e-119
nb_www                       9.716979e-124
ip                           4.384421e-197
google_index                  0.000000e+00
nb_or                                  NaN
ratio_nullHyperlinks                   NaN
ratio_intRedirection                   NaN
ratio_intErrors                        NaN
submit_email                           NaN
sfh                                    NaN
dtype: float64
```

[ ]:
```
def DropFeature (p_values, threshold):
        drop_feature = set()
        for index, values in p_values.items():
                if values > threshold or np.isnan(values):
                        drop_feature.add(index)
        return drop_feature
```

[ ]:
```
drop_feature = DropFeature(p_values,.05)
len(set(drop_feature))
```

[ ]: 21

[ ]:
```
drop_feature
```

[ ]:
```
{'char_repeat',
 'iframe',
 'nb_comma',
 'nb_dollar',
 'nb_or',
 'nb_percent',
 'nb_redirection',
 'nb_space',
 'nb_underscore',
 'onmouseover',
 'path_extension',
 'port',
 'punycode',
 'random_domain',
```

```
 'ratio_intErrors',
 'ratio_intRedirection',
 'ratio_nullHyperlinks',
 'right_clic',
 'sfh',
 'shortest_words_raw',
 'submit_email'}
```

[ ]:
```
X_train.drop(drop_feature, axis=1, inplace=True)
X_test.drop(drop_feature, axis=1, inplace=True)
```

[ ]:
```
len(X_train.columns)
```

[ ]: 66

[ ]:
```
len(X_test.columns)
```

[ ]: 66

[ ]:
```
print("Training set has {} samples.".format(X_train.shape[0]))
print("Testing set has {} samples.".format(X_test.shape[0]))
```

```
Training set has 10287 samples.
Testing set has 1143 samples.
```

[ ]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# defining parameter range
param_grid = {'penalty' : ['l2'],
             'C' : [0.1, 1, 10, 20, 30],
             'solver' : ['lbfgs','newton-cg','liblinear','sag','saga'],
             'max_iter' : [2500, 5000]}

grid_logr = GridSearchCV(LogisticRegression(), param_grid, refit = True, cv =
 ↪10, verbose = 3, n_jobs = -1)

# fitting the model for grid search
grid_logr.fit(X_train, y_train.values.ravel())

# print best parameter after tuning
print(grid_logr.best_params_)

# print how our model looks after hyper-parameter tuning
print(grid_logr.best_estimator_)
print(grid_logr.best_score_)
```

```
Fitting 10 folds for each of 50 candidates, totalling 500 fits
{'C': 30, 'max_iter': 2500, 'penalty': 'l2', 'solver': 'lbfgs'}
```

```
LogisticRegression(C=30, max_iter=2500)
0.9414791097094758
```

[ ]:
```python
logr_model = grid_logr.best_estimator_

# Performing training
#logr_model = logr.fit(X_train, y_train.values.ravel())
```

[ ]:
```python
logr_predict  =  logr_model.predict(X_test)
```

[ ]:
```python
# from sklearn.metrics import confusion_matrix,accuracy_score
# cm = confusion_matrix(y_test, dct_pred)
# ac = accuracy_score(y_test, dct_pred)
```

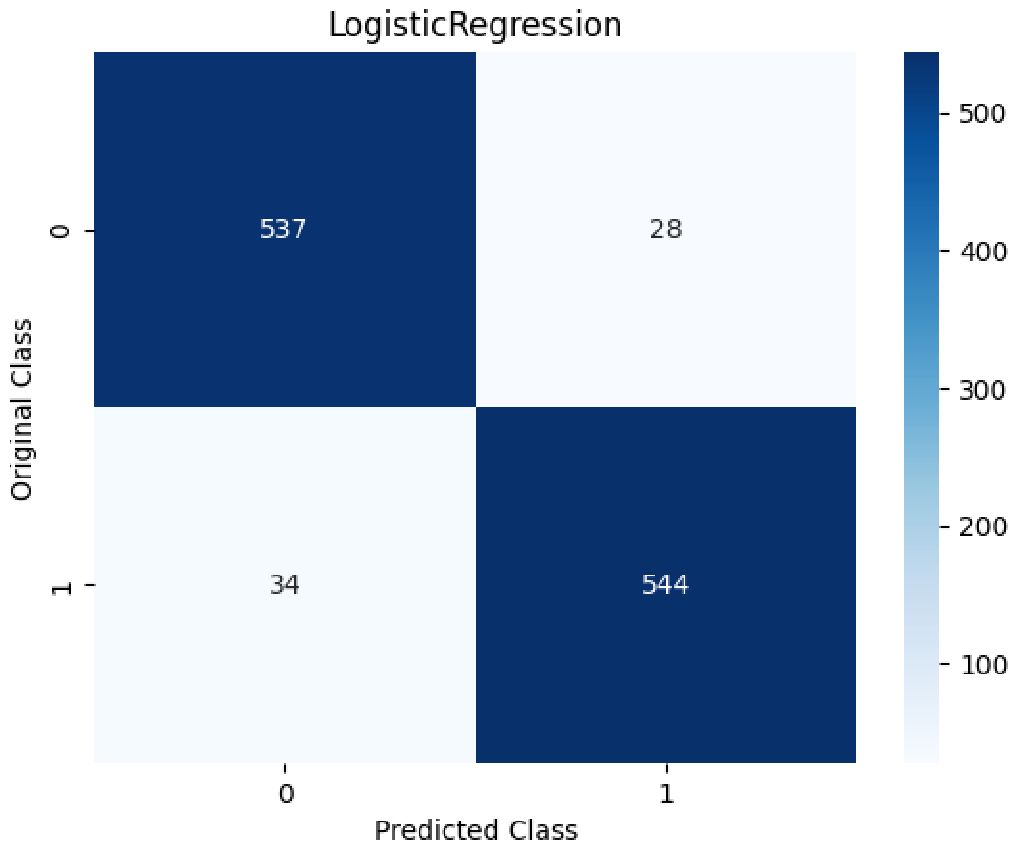
[ ]:
```python
print ("Accuracy of logr classifier : ", accuracy_score(y_test,␣
↪logr_predict)*100)
```

```
Accuracy of logr classifier :  94.57567804024497
```

[ ]:
```python
print(classification_report(y_test, logr_predict))
```

```
              precision    recall  f1-score   support

           0       0.94      0.95      0.95       565
           1       0.95      0.94      0.95       578

    accuracy                           0.95      1143
   macro avg       0.95      0.95      0.95      1143
weighted avg       0.95      0.95      0.95      1143
```

[ ]:
```python
sns.heatmap(confusion_matrix(y_test, logr_predict), annot=True, fmt='g',␣
↪cmap='Blues')
plt.title("LogisticRegression")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



[ ]:
```
# from sklearn.neighbors import KNeighborsClassifier

# #training_accuracy=[]
# test_accuracy=[]

# neighbors=range(1,10)
# ##values.ravel() converts vector y to flattened array
# for i in neighbors:
#     knn=KNeighborsClassifier(n_neighbors=i)
#     knn_model = knn.fit(X_train,y_train.values.ravel())
#     #training_accuracy.append(knn.score(X_train,y_train.values.ravel()))
#     test_accuracy.append(knn_model.score(X_test,y_test.values.ravel()))
```

[ ]:
```
# plt.plot(neighbors,test_accuracy,label="test accuracy")
# plt.ylabel("Accuracy")
# plt.xlabel("number of neighbors")
# plt.legend()
# plt.show()
```

```
[ ]: from sklearn.neighbors import KNeighborsClassifier

# defining parameter range
param_grid = {'n_neighbors': [1,2,3,4,5,6,7,8,9,10]}

grid_knn = GridSearchCV(KNeighborsClassifier(), param_grid, refit = True, cv =
 ↪10, verbose = 3, n_jobs = -1)

# fitting the model for grid search
grid_knn.fit(X_train, y_train.values.ravel())

# print best parameter after tuning
print(grid_knn.best_params_)

# print how our model looks after hyper-parameter tuning
print(grid_knn.best_estimator_)
print(grid_knn.best_score_)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits
{'n_neighbors': 5}
KNeighborsClassifier()
0.9254400592921993
```

```
[ ]: knn_model = grid_knn.best_estimator_
#knn_model = knn.fit(X_train,y_train.values.ravel())
```

```
[ ]: #print ("Accuracy of knn classifier: ", max(test_accuracy)*100)
knn_predict = knn_model.predict(X_test)
```

```
[ ]: print('The accuracy of knn Classifier is: ', 100.0 * accuracy_score(y_test,
 ↪knn_predict))
```

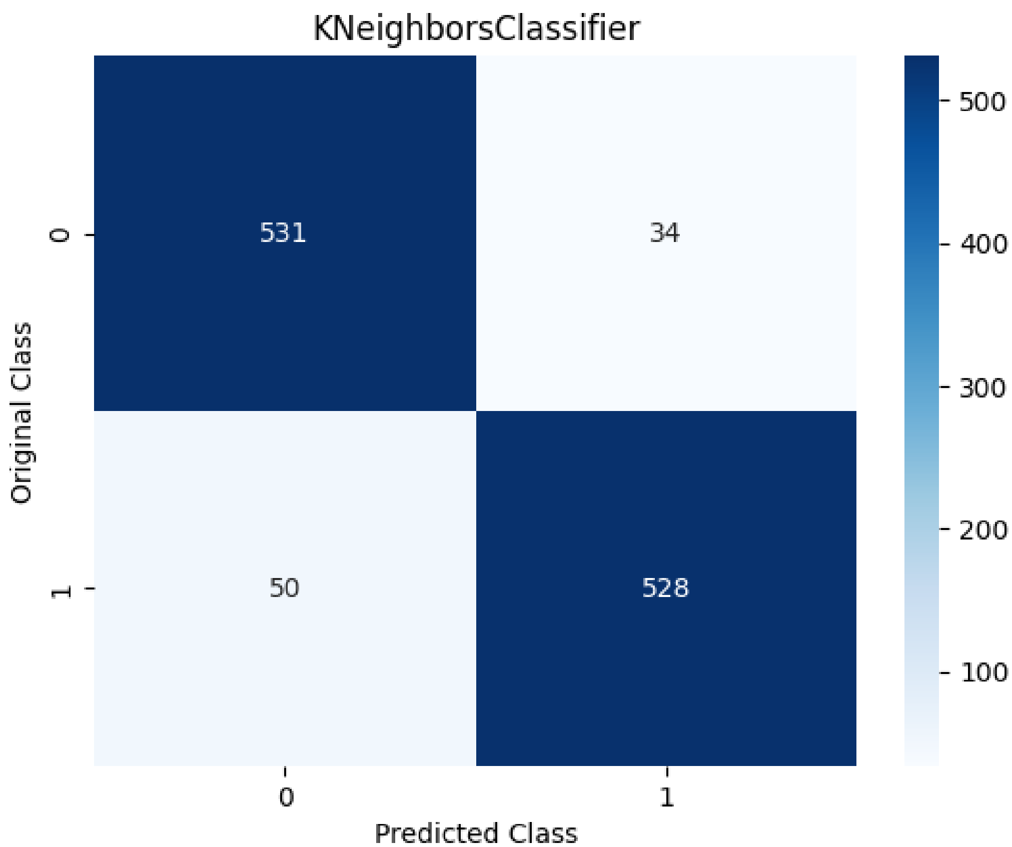
```
The accuracy of knn Classifier is:  92.6509186351706
```

```
[ ]: print(classification_report(y_test, knn_predict))
```

```
              precision    recall  f1-score   support

           0       0.91      0.94      0.93       565
           1       0.94      0.91      0.93       578

    accuracy                           0.93      1143
   macro avg       0.93      0.93      0.93      1143
weighted avg       0.93      0.93      0.93      1143
```

```
[ ]: sns.heatmap(confusion_matrix(y_test, knn_predict), annot=True, fmt='g',
 ↪cmap='Blues')
plt.title("KNeighborsClassifier")
```

```python
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



[ ]:

```python
# # here is the change
# knn_y_pred_proba = knn.predict_proba(X_test)
# knn_y_pred_proba_positive = knn_y_pred_proba[:, 1]

# RocCurveDisplay.from_predictions(y_test,knn_y_pred_proba_positive)

# fig, ax = plt.subplots()
# RocCurveDisplay.from_estimator(
#     logreg, X_test, y_test, ax = ax)

# logreg_y_decision = logreg.decision_function(X_test)
# metrics.RocCurveDisplay.
 ↪from_predictions(y_test,logreg_y_decision,ax=ax,name="logreg predictions")
```

[ ]:
```python
from sklearn.svm import SVC

# defining parameter range
param_grid = {'C': [0.1, 1, 10],
                        'gamma': [1, 0.1, 0.01],
                        'kernel': ['linear','poly', 'rbf', 'sigmoid']}

grid_svc = GridSearchCV(SVC(), param_grid, refit = True, cv = 10, verbose = 3,␣
 ↪n_jobs = -1)

# fitting the model for grid search
grid_svc.fit(X_train, y_train.values.ravel())

# print best parameter after tuning
print(grid_svc.best_params_)

# print how our model looks after hyper-parameter tuning
print(grid_svc.best_estimator_)
print(grid_svc.best_score_)
```

```
Fitting 10 folds for each of 36 candidates, totalling 360 fits
{'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
SVC(C=10, gamma=0.1)
0.9556720853989177
```

[ ]:
```python
svc_model = grid_svc.best_estimator_
#svc_model = svc.fit(X_train,y_train.values.ravel())
```

[ ]:
```python
svc_predict = svc_model.predict(X_test)
```

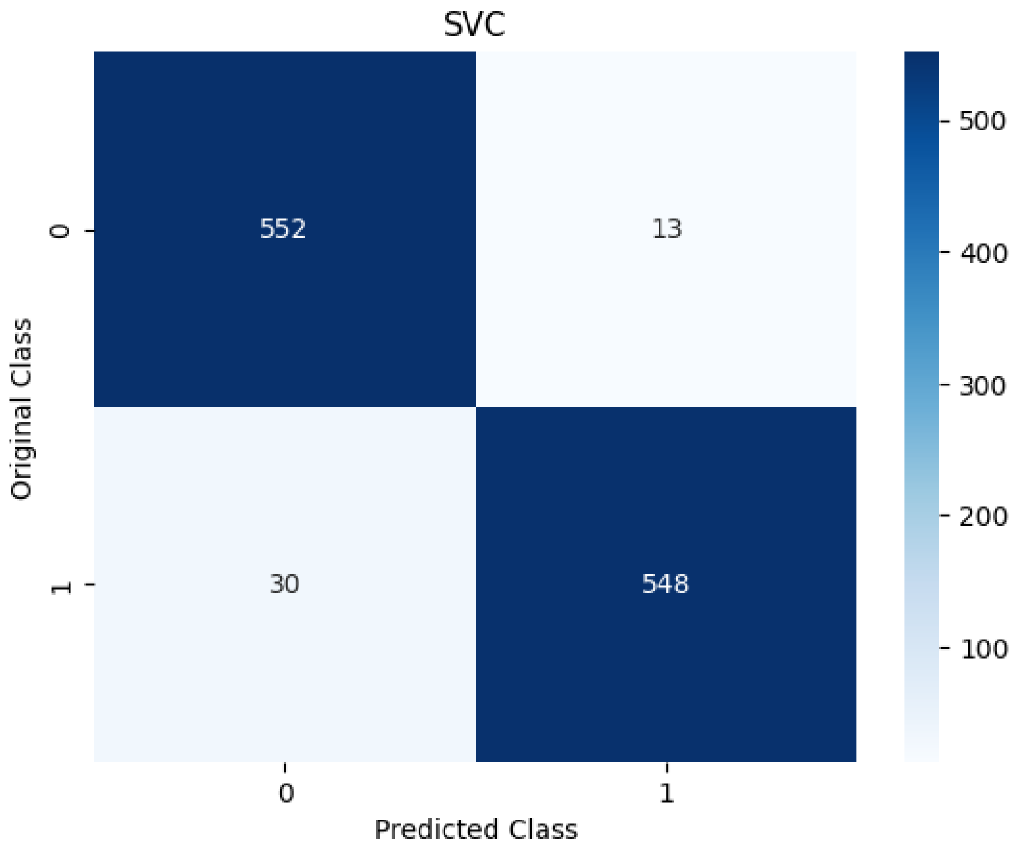
[ ]:
```python
print('The accuracy of svc Classifier is: ', 100.0 * accuracy_score(y_test,␣
 ↪svc_predict))
```

```
The accuracy of svc Classifier is:  96.23797025371829
```

[ ]:
```python
print(classification_report(y_test, svc_predict))
```

```
              precision    recall  f1-score   support

           0       0.95      0.98      0.96       565
           1       0.98      0.95      0.96       578

    accuracy                           0.96      1143
   macro avg       0.96      0.96      0.96      1143
weighted avg       0.96      0.96      0.96      1143
```

[ ]:
```
sns.heatmap(confusion_matrix(y_test, svc_predict), annot=True, fmt='g',␣
↪cmap='Blues')
plt.title("SVC")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



[ ]:
```
from sklearn.svm import NuSVC

# defining parameter range
param_grid = {'nu': [0.1, 0.5],
              'gamma': [1, 0.1, 0.01],
              'kernel': ['linear','poly', 'rbf', 'sigmoid']}

grid_nusvc = GridSearchCV(NuSVC(), param_grid, refit = True, verbose = 3, cv =␣
↪10, n_jobs = -1)

# fitting the model for grid search
grid_nusvc.fit(X_train, y_train.values.ravel())
```

```
# print best parameter after tuning
print(grid_nusvc.best_params_)

# print how our model looks after hyper-parameter tuning
print(grid_nusvc.best_estimator_)
print(grid_nusvc.best_score_)
```

```
Fitting 10 folds for each of 24 candidates, totalling 240 fits
{'gamma': 0.1, 'kernel': 'rbf', 'nu': 0.1}
NuSVC(gamma=0.1, nu=0.1)
0.957519010939562
```

[ ]:
```
nusvc_model = grid_nusvc.best_estimator_
#nusvc_model = nusvc.fit(X_train, y_train.values.ravel())
```

[ ]:
```
nusvc_predict = nusvc_model.predict(X_test)
```

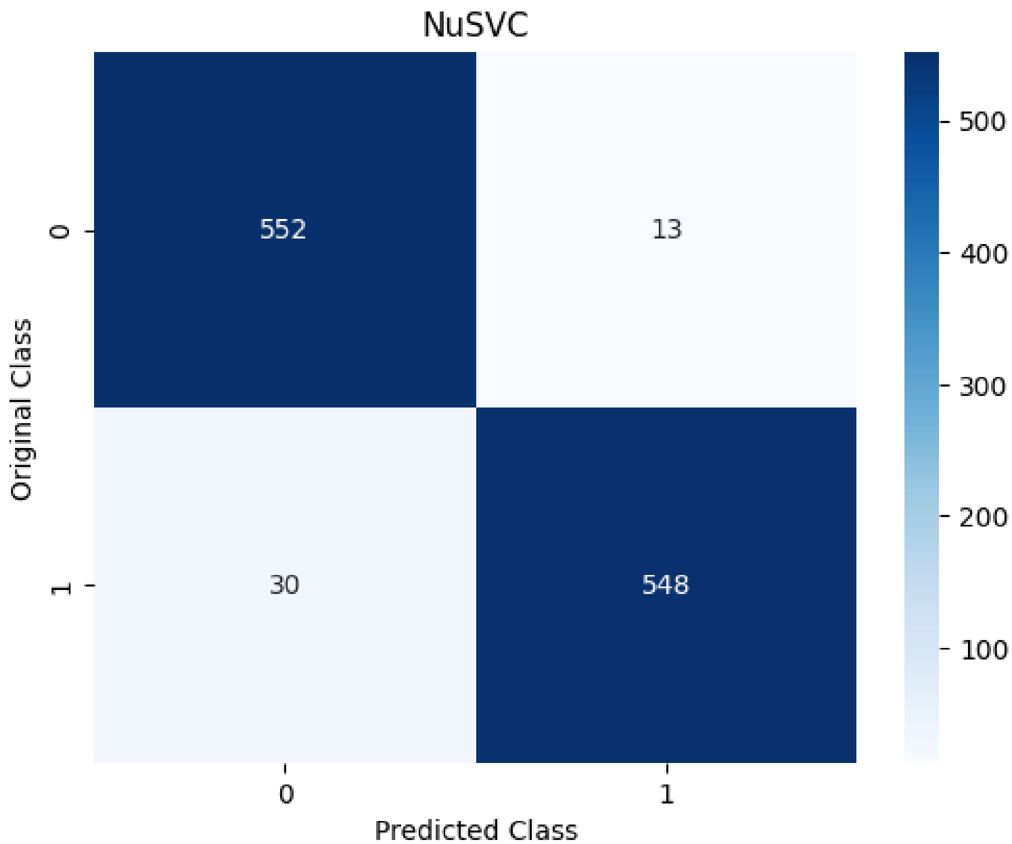
[ ]:
```
print('The accuracy of nusvc Classifier is: ', 100.0 * accuracy_score(y_test,␣
↪nusvc_predict))
```

```
The accuracy of nusvc Classifier is:  96.23797025371829
```

[ ]:
```
print(classification_report(y_test, nusvc_predict))
```

```
              precision    recall  f1-score   support

           0       0.95      0.98      0.96       565
           1       0.98      0.95      0.96       578

    accuracy                           0.96      1143
   macro avg       0.96      0.96      0.96      1143
weighted avg       0.96      0.96      0.96      1143
```

[ ]:
```
sns.heatmap(confusion_matrix(y_test, nusvc_predict), annot=True, fmt='g',␣
↪cmap='Blues')
plt.title("NuSVC")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```




```
[ ]: from sklearn.svm import LinearSVC

# defining parameter range
param_grid = {'C': [0.1, 1, 10, 20, 30],
                        'penalty': ['l1','l2'],
                        'loss': ['squared_hinge'],
                        'dual': [False],
                        'tol': [.1,.01,.001]}

grid_lsvc = GridSearchCV(LinearSVC(), param_grid, refit = True, verbose = 3, cv␣
 ↪= 10, n_jobs = -1)

# fitting the model for grid search
grid_lsvc.fit(X_train, y_train.values.ravel())

# print best parameter after tuning
print(grid_lsvc.best_params_)

# print how our model looks after hyper-parameter tuning
```

```
print(grid_lsvc.best_estimator_)
print(grid_lsvc.best_score_)
```

```
Fitting 10 folds for each of 30 candidates, totalling 300 fits
{'C': 30, 'dual': False, 'loss': 'squared_hinge', 'penalty': 'l1', 'tol': 0.001}
LinearSVC(C=30, dual=False, penalty='l1', tol=0.001)
0.9426457631412765
```

[ ]:
```
lsvc_model = grid_lsvc.best_estimator_
#lsvc_model = lsvc.fit(X_train, y_train.values.ravel())
```

[ ]:
```
lsvc_predict = lsvc_model.predict(X_test)
```

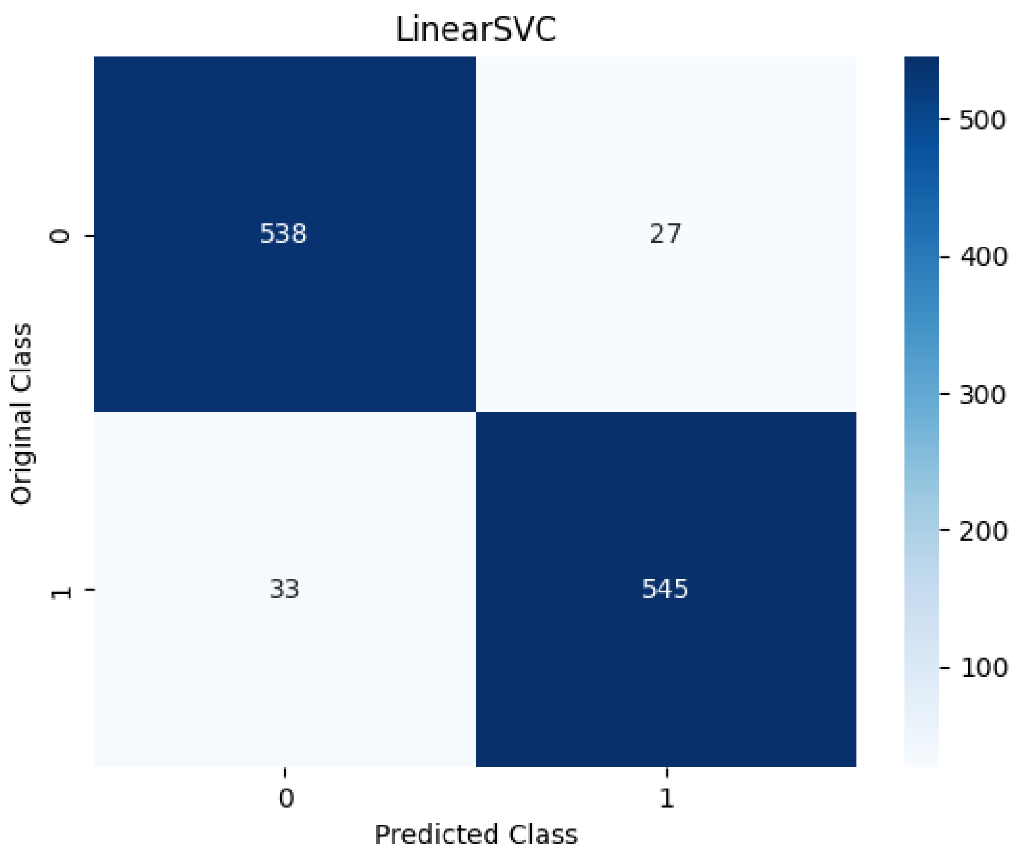
[ ]:
```
print('The accuracy of lsvc Classifier is: ', 100.0 * accuracy_score(y_test,␣
 ↪lsvc_predict))
```

```
The accuracy of lsvc Classifier is:  94.750656167979
```

[ ]:
```
print(classification_report(y_test, lsvc_predict))
```

```
              precision    recall  f1-score   support

           0       0.94      0.95      0.95       565
           1       0.95      0.94      0.95       578

    accuracy                           0.95      1143
   macro avg       0.95      0.95      0.95      1143
weighted avg       0.95      0.95      0.95      1143
```

[ ]:
```
sns.heatmap(confusion_matrix(y_test, lsvc_predict), annot=True, fmt='g',␣
 ↪cmap='Blues')
plt.title("LinearSVC")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



[ ]:
```python
from sklearn.ensemble import AdaBoostClassifier

# defining parameter range
param_grid = {'n_estimators': [40,50,100,200,300]}

grid_ada = GridSearchCV(AdaBoostClassifier(), param_grid, refit = True, verbose␣
↪= 3, cv = 10, n_jobs = -1)

# fitting the model for grid search
grid_ada.fit(X_train, y_train.values.ravel())

# print best parameter after tuning
print(grid_ada.best_params_)

# print how our model looks after hyper-parameter tuning
print(grid_ada.best_estimator_)
print(grid_ada.best_score_)
```

```
Fitting 10 folds for each of 5 candidates, totalling 50 fits
{'n_estimators': 300}
```

```
AdaBoostClassifier(n_estimators=300)
0.9557692671287524
```

[ ]:
```
ada_model = grid_ada.best_estimator_
#ada_model = ada.fit(X_train,y_train.values.ravel())
```

[ ]:
```
ada_predict = ada_model.predict(X_test)
```

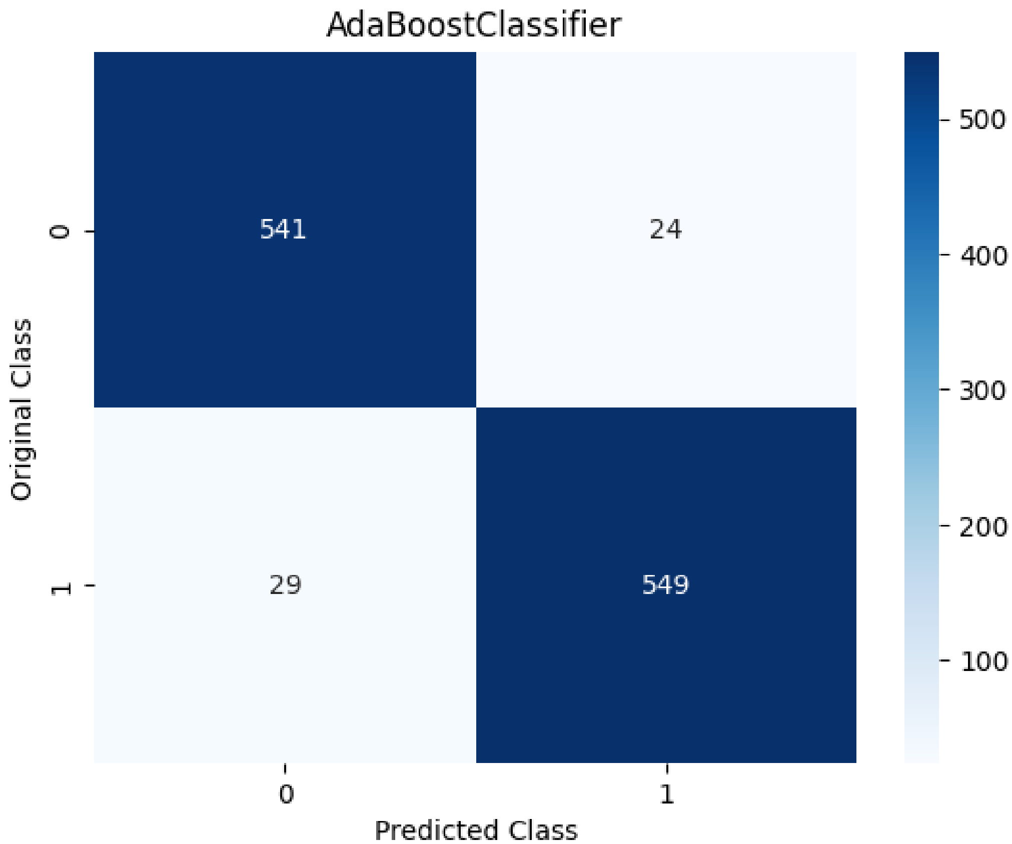
[ ]:
```
print('The accuracy of Ada Boost Classifier is: ', 100.0 *␣
    ↪accuracy_score(ada_predict,y_test))
```

```
The accuracy of Ada Boost Classifier is:  95.36307961504812
```

[ ]:
```
print(classification_report(y_test, ada_predict))
```

```
              precision    recall  f1-score   support

           0       0.95      0.96      0.95       565
           1       0.96      0.95      0.95       578

    accuracy                           0.95      1143
   macro avg       0.95      0.95      0.95      1143
weighted avg       0.95      0.95      0.95      1143
```

[ ]:
```
sns.heatmap(confusion_matrix(y_test, ada_predict), annot=True, fmt='g',␣
    ↪cmap='Blues')
plt.title("AdaBoostClassifier")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



[ ]:
```
from xgboost import XGBClassifier


# defining parameter range
param_grid = {
    "gamma": [.01, .1, .5],
    "n_estimators": [50,100,150,200,250]
}

grid_xgb = GridSearchCV(XGBClassifier(), param_grid, refit = True, verbose = 3,␣
↪cv = 10, n_jobs = -1)

# fitting the model for grid search
grid_xgb.fit(X_train, y_train.values.ravel())

# print best parameter after tuning
print(grid_xgb.best_params_)

# print how our model looks after hyper-parameter tuning
```

```
print(grid_xgb.best_estimator_)
print(grid_xgb.best_score_)
```

```
Fitting 10 folds for each of 15 candidates, totalling 150 fits
{'gamma': 0.1, 'n_estimators': 200}
XGBClassifier(base_score=0.5, booster='gbtree', callbacks=None,
              colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
              early_stopping_rounds=None, enable_categorical=False,
              eval_metric=None, gamma=0.1, gpu_id=-1, grow_policy='depthwise',
              importance_type=None, interaction_constraints='',
              learning_rate=0.300000012, max_bin=256, max_cat_to_onehot=4,
              max_delta_step=0, max_depth=6, max_leaves=0, min_child_weight=1,
              missing=nan, monotone_constraints='()', n_estimators=200,
              n_jobs=0, num_parallel_tree=1, predictor='auto', random_state=0,
              reg_alpha=0, reg_lambda=1, …)
0.970836216643411
```

[ ]:
```
xgb_model = grid_xgb.best_estimator_
#xgb_model = xgb.fit(X_train,y_train)
```

[ ]:
```
xgb_predict=xgb_model.predict(X_test)
```

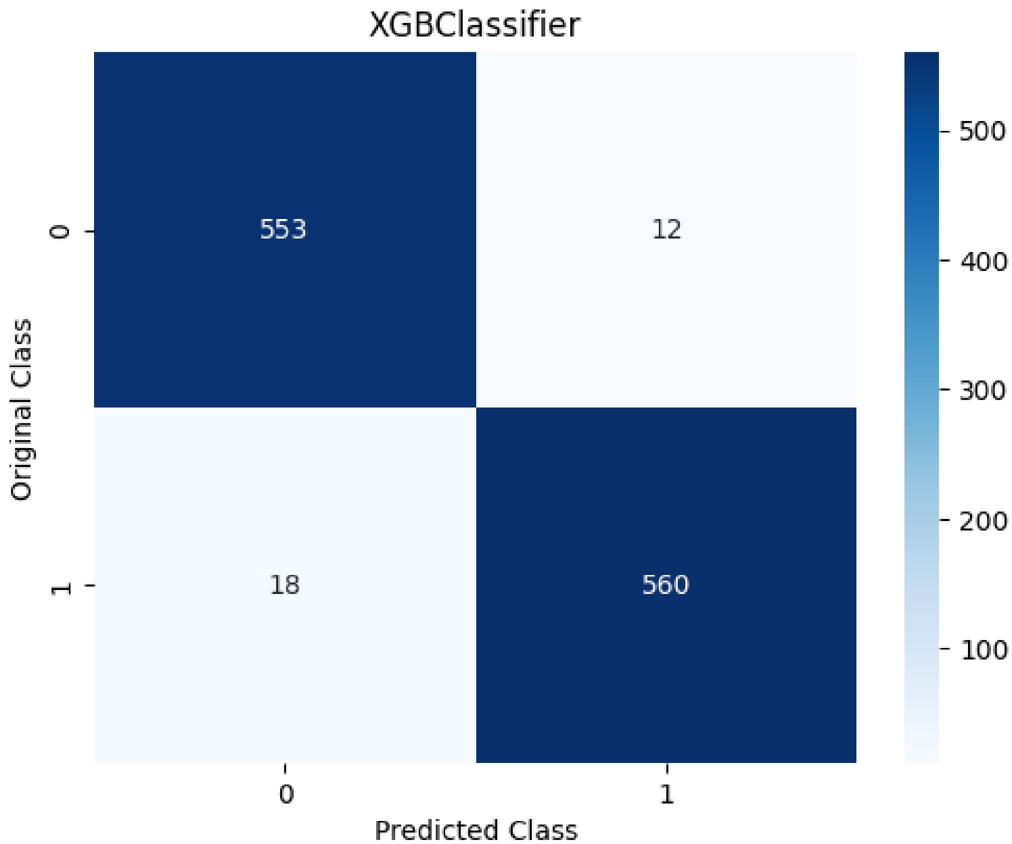
[ ]:
```
print('The accuracy of XGBoost Classifier is: ' , 100.0 *␣
↪accuracy_score(xgb_predict,y_test))
```

```
The accuracy of XGBoost Classifier is:  97.3753280839895
```

[ ]:
```
print(classification_report(y_test, xgb_predict))
```

```
              precision    recall  f1-score   support

           0       0.97      0.98      0.97       565
           1       0.98      0.97      0.97       578

    accuracy                           0.97      1143
   macro avg       0.97      0.97      0.97      1143
weighted avg       0.97      0.97      0.97      1143
```

[ ]:
```
sns.heatmap(confusion_matrix(y_test, xgb_predict), annot=True, fmt='g',␣
↪cmap='Blues')
plt.title("XGBClassifier")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



[ ]:
```python
from sklearn.ensemble import GradientBoostingClassifier

# defining parameter range
param_grid = {
    "learning_rate": [.1,.5,1],
    "n_estimators": [50,100,150,200,250]
}

grid_gbc = GridSearchCV(GradientBoostingClassifier(), param_grid, refit = True, verbose = 3, cv = 10, n_jobs = -1)

# fitting the model for grid search
grid_gbc.fit(X_train, y_train.values.ravel())

# print best parameter after tuning
print(grid_gbc.best_params_)

# print how our model looks after hyper-parameter tuning
print(grid_gbc.best_estimator_)
```

```
print(grid_gbc.best_score_)
```

```
Fitting 10 folds for each of 15 candidates, totalling 150 fits
{'learning_rate': 0.5, 'n_estimators': 250}
GradientBoostingClassifier(learning_rate=0.5, n_estimators=250)
0.968405917119488
```

[ ]:
```
gbc_model = grid_gbc.best_estimator_
#gbc_model = gbc.fit(X_train,y_train.values.ravel())

#clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
#    max_depth=1, random_state=0).fit(X_train, y_train)
#clf.score(X_test, y_test)
```

[ ]:
```
gbc_predict = gbc_model.predict(X_test)
```

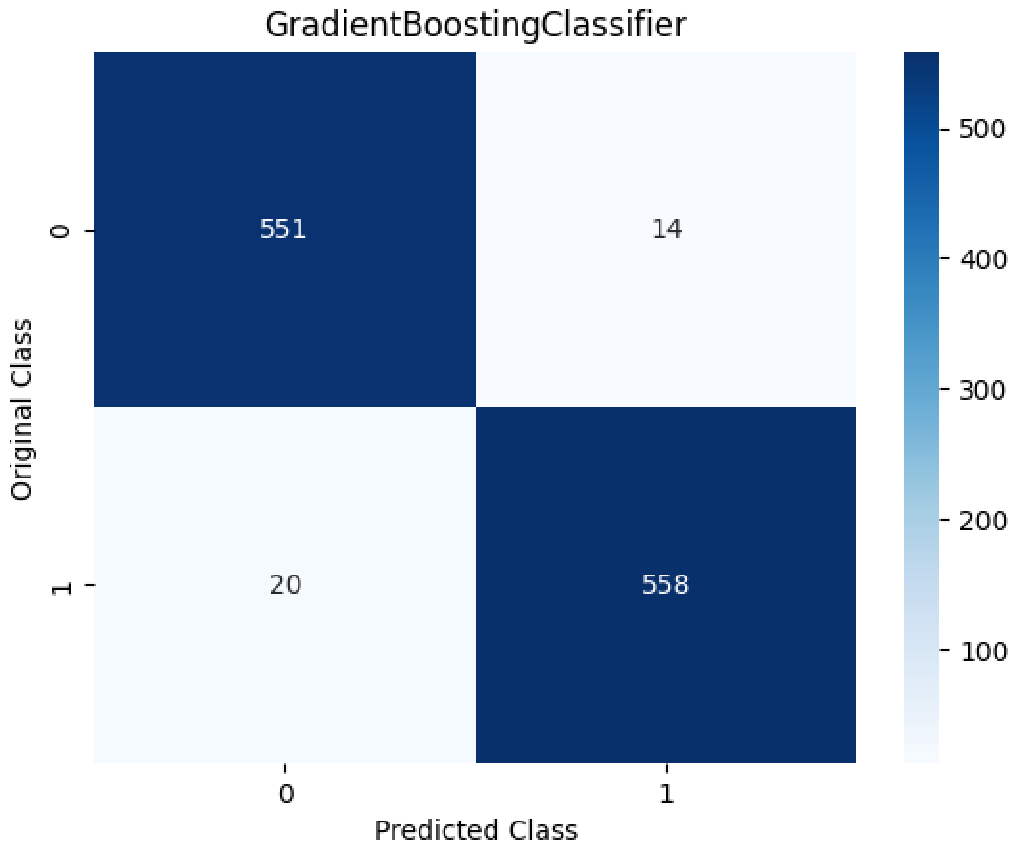
[ ]:
```
print('The accuracy of GradientBoost Classifier is: ' , 100.0 *␣
↪accuracy_score(gbc_predict,y_test))
```

```
The accuracy of GradientBoost Classifier is:  97.02537182852143
```

[ ]:
```
print(classification_report(y_test, gbc_predict))
```

```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       565
           1       0.98      0.97      0.97       578

    accuracy                           0.97      1143
   macro avg       0.97      0.97      0.97      1143
weighted avg       0.97      0.97      0.97      1143
```

[ ]:
```
sns.heatmap(confusion_matrix(y_test, gbc_predict), annot=True, fmt='g',␣
↪cmap='Blues')
plt.title("GradientBoostingClassifier")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



[ ]:
```
# gbc_model.get_params().keys()
```

[ ]:
```
# import inspect
# import sklearn
# import xgboost

# models = [xgboost.XGBClassifier]
# for m in models:
#     hyperparams = inspect.signature(m.__init__)
#     print(hyperparams)
# #or
# xgb_model.get_params().keys()
```

[ ]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

# defining parameter range
param_grid = {
    "base_estimator": [DecisionTreeClassifier()],
    "n_estimators": [50,100,150,200,250]
```

```
}

grid_bag = GridSearchCV(BaggingClassifier(), param_grid, refit = True, verbose␣
 ↪= 3, cv = 10, n_jobs = -1)

# fitting the model for grid search
grid_bag.fit(X_train, y_train.values.ravel())

# print best parameter after tuning
print(grid_bag.best_params_)

# print how our model looks after hyper-parameter tuning
print(grid_bag.best_estimator_)
print(grid_bag.best_score_)
```

```
Fitting 10 folds for each of 5 candidates, totalling 50 fits
{'base_estimator': DecisionTreeClassifier(), 'n_estimators': 100}
BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=100)
0.9590744858254585
```

[ ]:
```
bag_model = grid_bag.best_estimator_
#bag_model = bag.fit(X_train, y_train.values.ravel())
```

[ ]:
```
bag_predict = bag_model.predict(X_test)
```

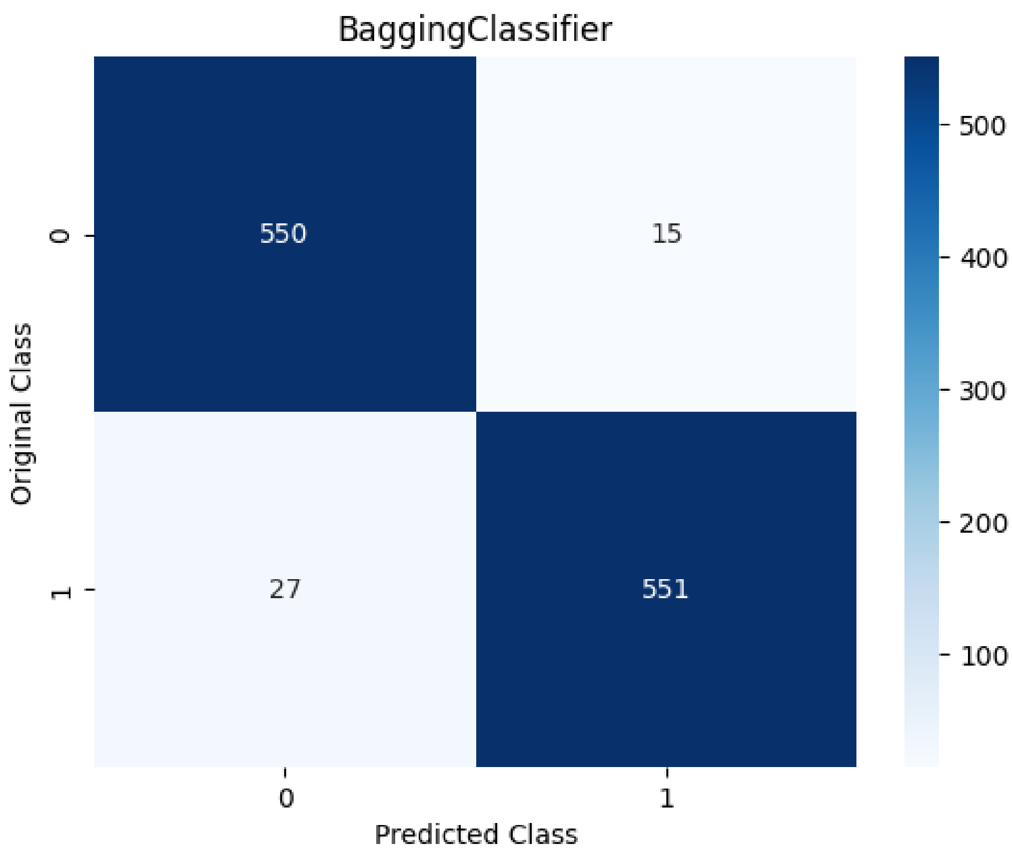
[ ]:
```
print('The accuracy of Bagging Classifier is: ' , 100.0 *␣
 ↪accuracy_score(y_test, bag_predict))
```

```
The accuracy of Bagging Classifier is:  96.3254593175853
```

[ ]:
```
print(classification_report(y_test, bag_predict))
```

```
              precision    recall  f1-score   support

           0       0.95      0.97      0.96       565
           1       0.97      0.95      0.96       578

    accuracy                           0.96      1143
   macro avg       0.96      0.96      0.96      1143
weighted avg       0.96      0.96      0.96      1143
```

[ ]:
```
sns.heatmap(confusion_matrix(y_test, bag_predict), annot=True, fmt='g',␣
 ↪cmap='Blues')
plt.title("BaggingClassifier")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



[ ]:
```
from sklearn.ensemble import RandomForestClassifier

# defining parameter range
param_grid = {
    "n_estimators": [50,100,150,200,250]
}

grid_rfc = GridSearchCV(RandomForestClassifier(), param_grid, refit = True,␣
 ↪verbose = 3, cv = 10, n_jobs = -1)

# fitting the model for grid search
grid_rfc.fit(X_train, y_train.values.ravel())

# print best parameter after tuning
print(grid_rfc.best_params_)

# print how our model looks after hyper-parameter tuning
print(grid_rfc.best_estimator_)
print(grid_rfc.best_score_)
```

```
Fitting 10 folds for each of 5 candidates, totalling 50 fits
{'n_estimators': 100}
RandomForestClassifier()
0.9658784358657304
```

[ ]:
```
rfc_model = grid_rfc.best_estimator_
#rfc_model = rfc.fit(X_train,y_train.values.ravel())
```

[ ]:
```
rfc_predict = rfc_model.predict(X_test)
```

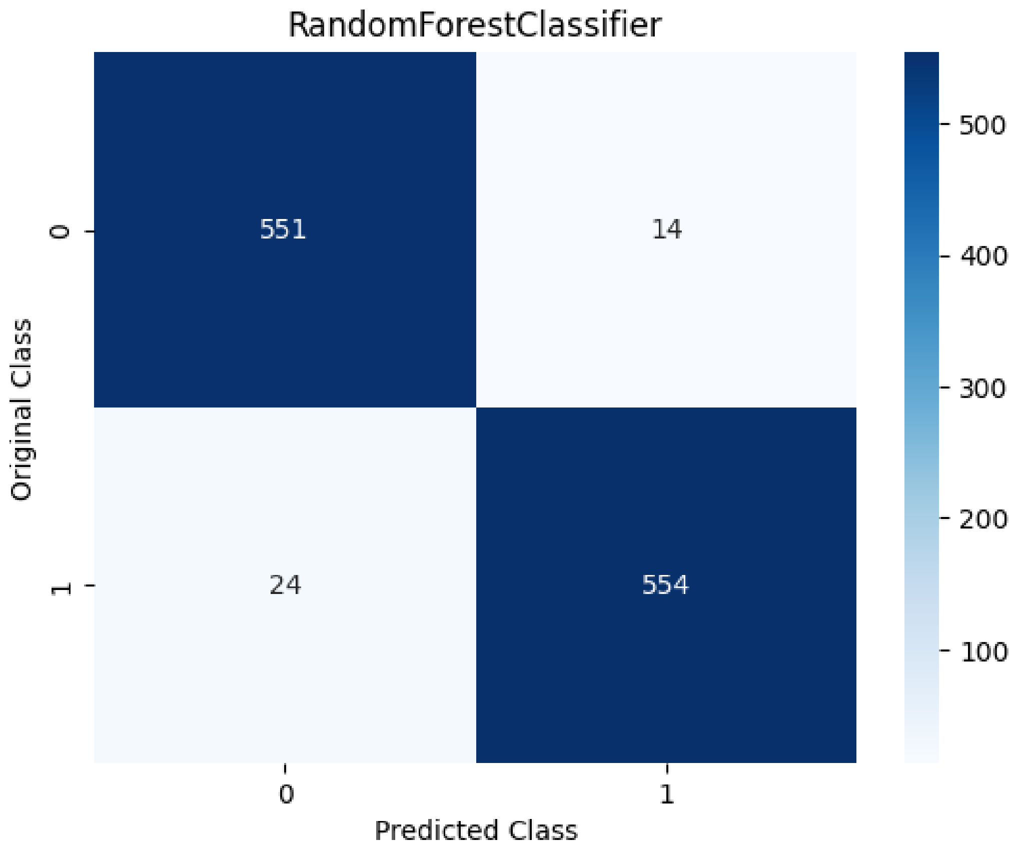
[ ]:
```
print('The accuracy of RandomForest Classifier is: ' , 100.0 *␣
 ↪accuracy_score(rfc_predict,y_test))
```

```
The accuracy of RandomForest Classifier is:  96.67541557305337
```

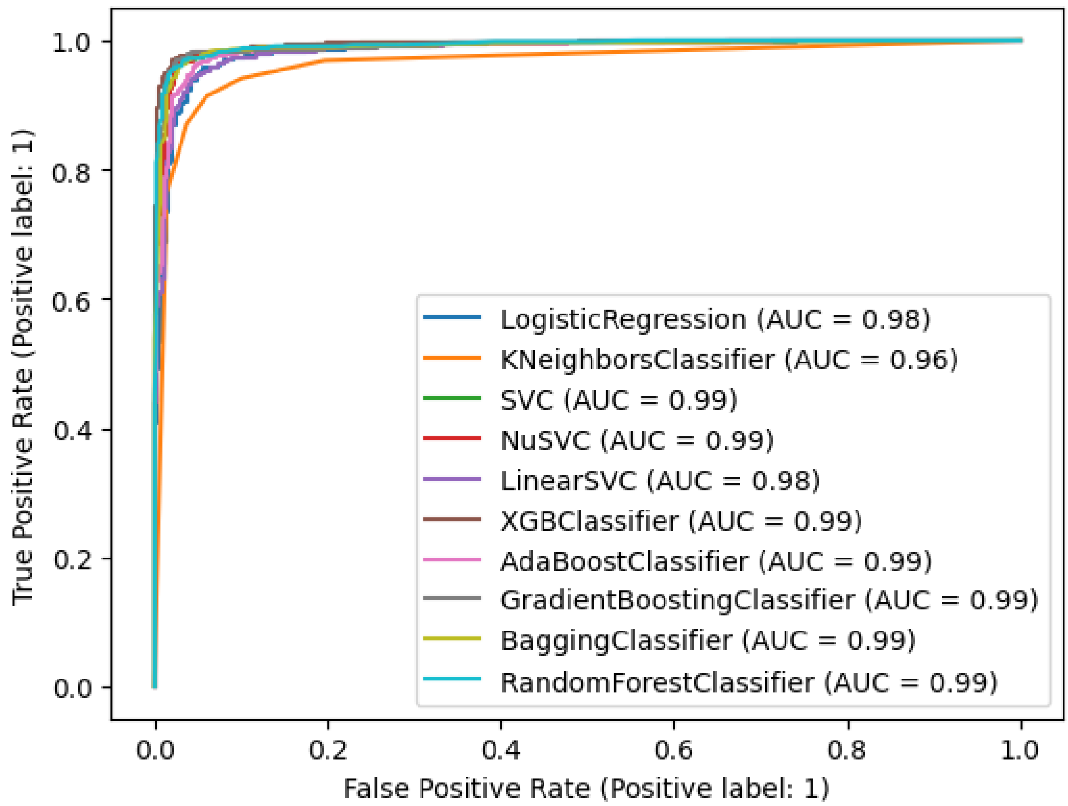
[ ]:
```
print(classification_report(y_test, rfc_predict))
```

```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       565
           1       0.98      0.96      0.97       578

    accuracy                           0.97      1143
   macro avg       0.97      0.97      0.97      1143
weighted avg       0.97      0.97      0.97      1143
```

[ ]:
```
sns.heatmap(confusion_matrix(y_test, rfc_predict), annot=True, fmt='g',␣
 ↪cmap='Blues')
plt.title("RandomForestClassifier")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



[ ]:
```
estimators =␣
  ↪[logr_model,knn_model,svc_model,nusvc_model,lsvc_model,xgb_model,ada_model,gbc_model,bag_mo

for estimator in estimators:
    RocCurveDisplay.from_estimator(estimator,X_test,y_test,ax=plt.gca())
```



[ ]:
```
import tensorflow as tf
#from tensorflow.keras.datasets import imdb
from keras.layers import Embedding, Dense, LSTM, BatchNormalization
from keras.losses import BinaryCrossentropy
from keras.models import Sequential
from keras.optimizers import Adam
#from tensorflow.keras.preprocessing.sequence import pad_sequences

# Model configuration
additional_metrics = ['accuracy']
batch_size = 32
#embedding_output_dims = (X_train.shape[1])
loss_function = BinaryCrossentropy()
#max_sequence_length = (X_train.shape[1])
#num_distinct_words = (X_train.shape[1])
number_of_epochs = 100
optimizer = Adam()
validation_split = 0.20
verbosity_mode = 1

# reshape from [samples, features] into [samples, timesteps, features]
```

```
timesteps = 1
X_train_reshape = X_train.values.ravel().reshape(X_train.shape[0],timesteps,␣
↪X_train.shape[1])
X_test_reshape = X_test.values.ravel().reshape(X_test.shape[0],timesteps,␣
↪X_test.shape[1])

# Disable eager execution
#tf.compat.v1.disable_eager_execution()

# Load dataset
# (x_train, y_train), (x_test, y_test) = imdb.
↪load_data(num_words=num_distinct_words)
# print(x_train.shape)
# print(x_test.shape)

# Pad all sequences
# padded_inputs = pad_sequences(X_train, maxlen=max_sequence_length, value = 0.
↪0) # 0.0 because it corresponds with <PAD>
# padded_inputs_test = pad_sequences(X_test, maxlen=max_sequence_length, value␣
↪= 0.0) # 0.0 because it corresponds with <PAD>

# Define the Keras model
def build_model_lstm():
    model = Sequential()
    #model.add(Embedding(num_distinct_words, embedding_output_dims,␣
↪input_length=max_sequence_length))
    model.add(LSTM(100, input_shape = (timesteps,X_train_reshape.shape[2])))
    model.add(BatchNormalization())
    model.add(Dense(50, activation='relu'))
    model.add(Dense(25, activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    # Compile the model
    model.compile(optimizer=optimizer, loss=loss_function,␣
↪metrics=additional_metrics)
    return model

#from keras.wrappers.scikit_learn import KerasClassifier
lstm_model = build_model_lstm()
# Give a summary
lstm_model.summary()

# Train the model
```

```python
history = lstm_model.fit(X_train_reshape, y_train.values.ravel(),␣
↪batch_size=batch_size, epochs=number_of_epochs, verbose=verbosity_mode,␣
↪validation_split=validation_split)

# Test the model after training
#lstm_predict = lstm_model.predict(X_test_reshape)
test_results = lstm_model.evaluate(X_test_reshape, y_test.values.ravel(),␣
↪verbose=False)
print(f'Test results - Loss: {test_results[0]} - Accuracy:␣
↪{100*test_results[1]}%')
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 100)               66800

 batch_normalization (BatchN  (None, 100)              400
 ormalization)


_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 100)               66800

 batch_normalization (BatchN  (None, 100)              400
 ormalization)

 dense (Dense)               (None, 50)                5050

 dense_1 (Dense)             (None, 25)                1275

 dense_2 (Dense)             (None, 10)                260

 dense_3 (Dense)             (None, 1)                 11

=================================================================
Total params: 73,796
Trainable params: 73,596
Non-trainable params: 200
_________________________________________________________________
Epoch 1/100
258/258 [==============================] - 4s 7ms/step - loss: 0.2286 -
accuracy: 0.9121 - val_loss: 0.3887 - val_accuracy: 0.9349
Epoch 2/100
258/258 [==============================] - 1s 5ms/step - loss: 0.1511 -
accuracy: 0.9441 - val_loss: 0.1841 - val_accuracy: 0.9407
```

```
Epoch 3/100
258/258 [==============================] - 1s 4ms/step - loss: 0.1319 -
accuracy: 0.9508 - val_loss: 0.1690 - val_accuracy: 0.9397
Epoch 4/100
258/258 [==============================] - 1s 4ms/step - loss: 0.1230 -
accuracy: 0.9546 - val_loss: 0.1488 - val_accuracy: 0.9524
Epoch 5/100
258/258 [==============================] - 1s 4ms/step - loss: 0.1123 -
accuracy: 0.9583 - val_loss: 0.1498 - val_accuracy: 0.9475
Epoch 6/100
258/258 [==============================] - 1s 5ms/step - loss: 0.1057 -
accuracy: 0.9615 - val_loss: 0.1989 - val_accuracy: 0.9329
Epoch 7/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0982 -
accuracy: 0.9643 - val_loss: 0.1459 - val_accuracy: 0.9514
Epoch 8/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0908 -
accuracy: 0.9656 - val_loss: 0.1375 - val_accuracy: 0.9466
Epoch 9/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0845 -
accuracy: 0.9676 - val_loss: 0.1422 - val_accuracy: 0.9534
Epoch 10/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0873 -
accuracy: 0.9648 - val_loss: 0.1556 - val_accuracy: 0.9451
Epoch 11/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0768 -
accuracy: 0.9691 - val_loss: 0.1449 - val_accuracy: 0.9558
Epoch 12/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0745 -
accuracy: 0.9719 - val_loss: 0.1456 - val_accuracy: 0.9529
Epoch 13/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0726 -
accuracy: 0.9744 - val_loss: 0.1561 - val_accuracy: 0.9558
Epoch 14/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0678 -
accuracy: 0.9729 - val_loss: 0.1509 - val_accuracy: 0.9514
Epoch 15/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0611 -
accuracy: 0.9768 - val_loss: 0.1631 - val_accuracy: 0.9543
Epoch 16/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0600 -
accuracy: 0.9779 - val_loss: 0.1718 - val_accuracy: 0.9500
Epoch 17/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0566 -
accuracy: 0.9782 - val_loss: 0.1644 - val_accuracy: 0.9485
Epoch 18/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0572 -
accuracy: 0.9796 - val_loss: 0.1512 - val_accuracy: 0.9577
```

```
Epoch 19/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0534 -
accuracy: 0.9797 - val_loss: 0.1964 - val_accuracy: 0.9475
Epoch 20/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0561 -
accuracy: 0.9785 - val_loss: 0.1631 - val_accuracy: 0.9490
Epoch 21/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0515 -
accuracy: 0.9801 - val_loss: 0.1624 - val_accuracy: 0.9519
Epoch 22/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0437 -
accuracy: 0.9842 - val_loss: 0.1993 - val_accuracy: 0.9456
Epoch 23/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0438 -
accuracy: 0.9832 - val_loss: 0.1865 - val_accuracy: 0.9490
Epoch 24/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0425 -
accuracy: 0.9825 - val_loss: 0.1903 - val_accuracy: 0.9495
Epoch 25/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0429 -
accuracy: 0.9832 - val_loss: 0.1965 - val_accuracy: 0.9466
Epoch 26/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0441 -
accuracy: 0.9846 - val_loss: 0.2047 - val_accuracy: 0.9466
Epoch 27/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0383 -
accuracy: 0.9842 - val_loss: 0.1889 - val_accuracy: 0.9495
Epoch 28/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0355 -
accuracy: 0.9855 - val_loss: 0.1907 - val_accuracy: 0.9534
Epoch 29/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0330 -
accuracy: 0.9871 - val_loss: 0.2027 - val_accuracy: 0.9485
Epoch 30/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0372 -
accuracy: 0.9861 - val_loss: 0.2101 - val_accuracy: 0.9461
Epoch 31/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0306 -
accuracy: 0.9888 - val_loss: 0.2160 - val_accuracy: 0.9490
Epoch 32/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0374 -
accuracy: 0.9844 - val_loss: 0.2086 - val_accuracy: 0.9534
Epoch 33/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0350 -
accuracy: 0.9865 - val_loss: 0.1967 - val_accuracy: 0.9543
Epoch 34/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0323 -
accuracy: 0.9888 - val_loss: 0.2198 - val_accuracy: 0.9500
```

```
Epoch 35/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0265 -
accuracy: 0.9903 - val_loss: 0.2471 - val_accuracy: 0.9495
Epoch 36/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0286 -
accuracy: 0.9892 - val_loss: 0.2141 - val_accuracy: 0.9480
Epoch 37/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0328 -
accuracy: 0.9878 - val_loss: 0.2374 - val_accuracy: 0.9451
Epoch 38/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0234 -
accuracy: 0.9913 - val_loss: 0.2337 - val_accuracy: 0.9509
Epoch 39/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0258 -
accuracy: 0.9905 - val_loss: 0.2415 - val_accuracy: 0.9495
Epoch 40/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0234 -
accuracy: 0.9909 - val_loss: 0.2356 - val_accuracy: 0.9504
Epoch 41/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0180 -
accuracy: 0.9934 - val_loss: 0.2622 - val_accuracy: 0.9495
Epoch 42/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0202 -
accuracy: 0.9930 - val_loss: 0.2554 - val_accuracy: 0.9431
Epoch 43/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0254 -
accuracy: 0.9908 - val_loss: 0.2624 - val_accuracy: 0.9490
Epoch 44/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0236 -
accuracy: 0.9913 - val_loss: 0.2721 - val_accuracy: 0.9456
Epoch 45/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0207 -
accuracy: 0.9920 - val_loss: 0.2646 - val_accuracy: 0.9466
Epoch 46/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0219 -
accuracy: 0.9917 - val_loss: 0.2685 - val_accuracy: 0.9480
Epoch 47/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0201 -
accuracy: 0.9937 - val_loss: 0.2672 - val_accuracy: 0.9461
Epoch 48/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0184 -
accuracy: 0.9922 - val_loss: 0.2542 - val_accuracy: 0.9466
Epoch 49/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0141 -
accuracy: 0.9955 - val_loss: 0.2878 - val_accuracy: 0.9524
Epoch 50/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0250 -
accuracy: 0.9920 - val_loss: 0.2518 - val_accuracy: 0.9436
```

```
Epoch 51/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0161 -
accuracy: 0.9944 - val_loss: 0.2310 - val_accuracy: 0.9519
Epoch 52/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0196 -
accuracy: 0.9936 - val_loss: 0.2829 - val_accuracy: 0.9466
Epoch 53/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0210 -
accuracy: 0.9933 - val_loss: 0.2660 - val_accuracy: 0.9466
Epoch 54/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0134 -
accuracy: 0.9948 - val_loss: 0.2975 - val_accuracy: 0.9470
Epoch 55/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0149 -
accuracy: 0.9956 - val_loss: 0.2955 - val_accuracy: 0.9470
Epoch 56/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0236 -
accuracy: 0.9909 - val_loss: 0.2901 - val_accuracy: 0.9431
Epoch 57/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0159 -
accuracy: 0.9940 - val_loss: 0.3400 - val_accuracy: 0.9422
Epoch 58/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0159 -
accuracy: 0.9950 - val_loss: 0.2853 - val_accuracy: 0.9500
Epoch 59/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0255 -
accuracy: 0.9908 - val_loss: 0.2943 - val_accuracy: 0.9456
Epoch 60/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0199 -
accuracy: 0.9926 - val_loss: 0.2594 - val_accuracy: 0.9572
Epoch 61/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0147 -
accuracy: 0.9944 - val_loss: 0.2710 - val_accuracy: 0.9504
Epoch 62/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0129 -
accuracy: 0.9957 - val_loss: 0.2971 - val_accuracy: 0.9504
Epoch 63/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0130 -
accuracy: 0.9960 - val_loss: 0.3000 - val_accuracy: 0.9490
Epoch 64/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0083 -
accuracy: 0.9968 - val_loss: 0.3075 - val_accuracy: 0.9495
Epoch 65/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0182 -
accuracy: 0.9940 - val_loss: 0.2898 - val_accuracy: 0.9480
Epoch 66/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0125 -
accuracy: 0.9955 - val_loss: 0.2960 - val_accuracy: 0.9514
```

```
Epoch 67/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0127 -
accuracy: 0.9949 - val_loss: 0.3255 - val_accuracy: 0.9461
Epoch 68/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0123 -
accuracy: 0.9945 - val_loss: 0.3165 - val_accuracy: 0.9490
Epoch 69/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0156 -
accuracy: 0.9943 - val_loss: 0.3274 - val_accuracy: 0.9524
Epoch 70/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0106 -
accuracy: 0.9964 - val_loss: 0.3143 - val_accuracy: 0.9509
Epoch 71/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0152 -
accuracy: 0.9950 - val_loss: 0.3157 - val_accuracy: 0.9470
Epoch 72/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0112 -
accuracy: 0.9961 - val_loss: 0.3338 - val_accuracy: 0.9485
Epoch 73/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0122 -
accuracy: 0.9964 - val_loss: 0.3216 - val_accuracy: 0.9509
Epoch 74/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0088 -
accuracy: 0.9973 - val_loss: 0.3258 - val_accuracy: 0.9495
Epoch 75/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0099 -
accuracy: 0.9965 - val_loss: 0.3381 - val_accuracy: 0.9490
Epoch 76/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0113 -
accuracy: 0.9964 - val_loss: 0.3591 - val_accuracy: 0.9519
Epoch 77/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0125 -
accuracy: 0.9953 - val_loss: 0.3495 - val_accuracy: 0.9548
Epoch 78/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0130 -
accuracy: 0.9944 - val_loss: 0.3476 - val_accuracy: 0.9475
Epoch 79/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0118 -
accuracy: 0.9954 - val_loss: 0.3303 - val_accuracy: 0.9500
Epoch 80/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0135 -
accuracy: 0.9961 - val_loss: 0.3298 - val_accuracy: 0.9509
Epoch 81/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0120 -
accuracy: 0.9959 - val_loss: 0.3401 - val_accuracy: 0.9461
Epoch 82/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0169 -
accuracy: 0.9951 - val_loss: 0.3563 - val_accuracy: 0.9485
```

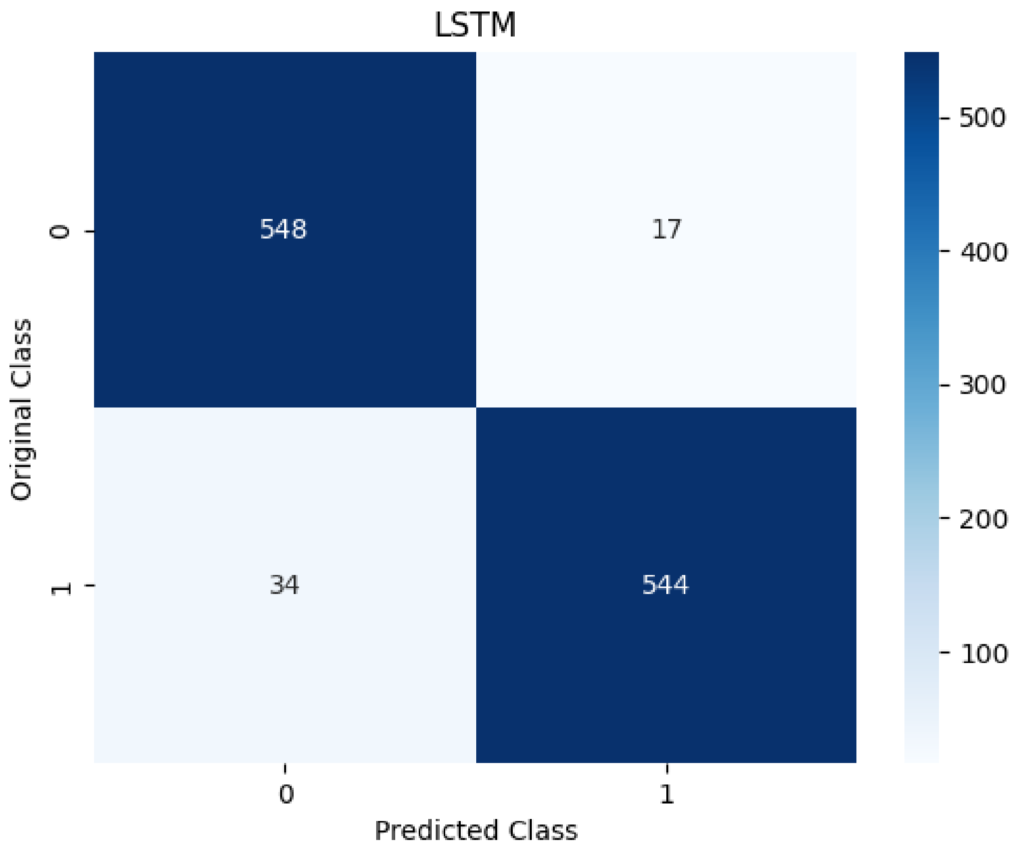
```
Epoch 83/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0175 -
accuracy: 0.9939 - val_loss: 0.3405 - val_accuracy: 0.9466
Epoch 84/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0128 -
accuracy: 0.9945 - val_loss: 0.3154 - val_accuracy: 0.9509
Epoch 85/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0099 -
accuracy: 0.9970 - val_loss: 0.3274 - val_accuracy: 0.9466
Epoch 86/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0104 -
accuracy: 0.9965 - val_loss: 0.3269 - val_accuracy: 0.9534
Epoch 87/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0087 -
accuracy: 0.9970 - val_loss: 0.3469 - val_accuracy: 0.9519
Epoch 88/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0092 -
accuracy: 0.9962 - val_loss: 0.3134 - val_accuracy: 0.9485
Epoch 89/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0094 -
accuracy: 0.9965 - val_loss: 0.3712 - val_accuracy: 0.9495
Epoch 90/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0105 -
accuracy: 0.9965 - val_loss: 0.3487 - val_accuracy: 0.9461
Epoch 91/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0116 -
accuracy: 0.9971 - val_loss: 0.3312 - val_accuracy: 0.9495
Epoch 92/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0149 -
accuracy: 0.9947 - val_loss: 0.3384 - val_accuracy: 0.9514
Epoch 93/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0069 -
accuracy: 0.9973 - val_loss: 0.3420 - val_accuracy: 0.9466
Epoch 94/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0069 -
accuracy: 0.9981 - val_loss: 0.3791 - val_accuracy: 0.9446
Epoch 95/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0124 -
accuracy: 0.9957 - val_loss: 0.3510 - val_accuracy: 0.9495
Epoch 96/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0208 -
accuracy: 0.9925 - val_loss: 0.3500 - val_accuracy: 0.9470
Epoch 97/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0095 -
accuracy: 0.9968 - val_loss: 0.3619 - val_accuracy: 0.9548
Epoch 98/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0084 -
accuracy: 0.9973 - val_loss: 0.3322 - val_accuracy: 0.9509
```

```
Epoch 99/100
258/258 [==============================] - 1s 5ms/step - loss: 0.0060 -
accuracy: 0.9982 - val_loss: 0.3435 - val_accuracy: 0.9543
Epoch 100/100
258/258 [==============================] - 1s 4ms/step - loss: 0.0091 -
accuracy: 0.9968 - val_loss: 0.3387 - val_accuracy: 0.9524
Test results - Loss: 0.295855313539505 - Accuracy: 95.53805589675903%
```

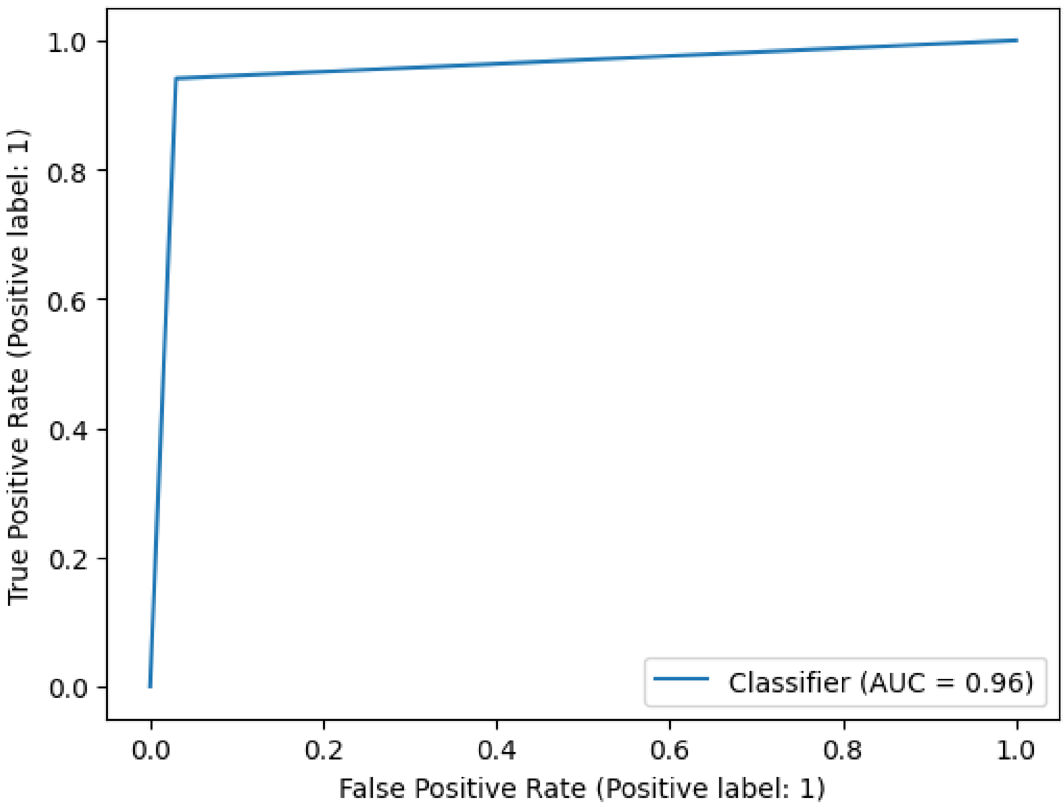
[ ]:
```
lstm_predict_proba = lstm_model.predict(X_test_reshape, batch_size=32)
lstm_predict_class = (lstm_predict_proba > 0.5).astype("int32")
print(classification_report(y_test, lstm_predict_class))
```

```
36/36 [==============================] - 1s 2ms/step
              precision    recall  f1-score   support

           0       0.94      0.97      0.96       565
           1       0.97      0.94      0.96       578

    accuracy                           0.96      1143
   macro avg       0.96      0.96      0.96      1143
weighted avg       0.96      0.96      0.96      1143
```

[ ]:
```
sns.heatmap(confusion_matrix(y_test, lstm_predict_class), annot=True, fmt='g',␣
 ↪cmap='Blues')
plt.title("LSTM")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



[ ]:
```
RocCurveDisplay.from_predictions(y_test,lstm_predict_class)
plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate (Positive label: 1)
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate (Positive label: 1)
Classifier (AUC = 0.96)