chi_sq_87 7030 split .05 threshold

January 3, 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=0.3,
    random_state=10)

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

[ ]:
```
((8001, 87), (3429, 87), (8001, 1), (3429, 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([1.83569465e+01, 1.49131186e+01, 6.76330545e+02, 1.82566029e+01,
        1.01951447e+01, 4.65673053e+01, 2.09347136e+02, 4.89171359e+01,
                   nan, 7.53148104e+01, 1.46775301e+00, 6.63686807e+00,
        8.37957961e-01, 2.47247356e+01, 5.94476238e+00, 2.51351150e+01,
        3.71713513e-02, 2.36794645e+01, 2.64211661e+00, 8.88630671e-02,
        4.39801261e+02, 3.60708888e+01, 3.77932908e+01, 1.75021740e+01,
        4.15672811e+01, 2.29064686e+02, 1.69027267e+02, 2.97238119e+00,
        2.81803485e+00, 4.39725683e+01, 3.16099671e+02, 1.15925380e+02,
        1.56697452e+01, 2.89670197e+02, 1.37088747e+00, 8.04855493e+01,
        4.27714714e-05, 6.92468921e-01, 2.57606370e+01, 1.62824885e+01,
        3.13369774e-02, 7.40381614e-01, 4.00088512e+01, 3.37843028e+00,
        2.29738807e+01, 5.47738054e+00, 3.55418381e+01, 7.02022546e+00,
        1.50270090e+01, 1.47256699e+01, 1.90177663e+02, 6.22133100e+01,
        3.36869868e+01, 3.01080484e+01, 9.69700849e+01, 1.52524490e+02,
        6.64208432e+01, 1.13832855e+02, 2.94862463e+01,            nan,
        3.78250054e+00,            nan, 3.52276929e+01,            nan,
        4.42307114e+00, 1.08868779e+00, 9.09609687e+01, 9.28104235e+01,
                   nan, 1.60840341e+02, 8.40539808e+01,            nan,
        8.46181684e-01, 3.56702835e+01, 1.04026757e+02, 1.52258940e-01,
        8.18972365e-02, 2.79839251e+02, 2.08875098e+02, 1.26054243e+02,
        2.82248487e+01, 9.02094305e+00, 1.63151859e+02, 1.30478127e+01,
        1.15316457e+02, 1.97810797e+03, 4.08786108e+02]),
 array([1.83150007e-005, 1.12577404e-004, 4.19671063e-149, 1.93055574e-005,
        1.40810914e-003, 8.85250039e-012, 1.90679184e-047, 2.67007782e-012,
                    nan, 4.01337861e-018, 2.25700552e-001, 9.98893917e-003,
```

```
       3.59981449e-001, 6.61299807e-007, 1.47610811e-002, 5.34506319e-007,
       8.47116596e-001, 1.13790290e-006, 1.04064816e-001, 7.65627537e-001,
       1.19591294e-097, 1.90268172e-009, 7.86518052e-010, 2.86979347e-005,
       1.13883350e-010, 9.53500103e-052, 1.20677044e-038, 8.46971008e-002,
       9.32104587e-002, 3.33010414e-011, 1.02445647e-070, 4.93521036e-027,
       7.54210251e-005, 5.86702031e-065, 2.41659463e-001, 2.92838626e-019,
       9.94781880e-001, 4.05325876e-001, 3.86491028e-007, 5.45659008e-005,
       8.59490690e-001, 3.89538661e-001, 2.52814681e-010, 6.60549387e-002,
       1.64217471e-006, 1.92641060e-002, 2.49630827e-009, 8.05940820e-003,
       1.05983467e-004, 1.24341787e-004, 2.90771100e-043, 3.08197829e-015,
       6.47322788e-009, 4.08631781e-008, 7.03827331e-023, 4.86626742e-035,
       3.64230292e-016, 1.41768936e-026, 5.63149210e-008,             nan,
       5.17912077e-002,             nan, 2.93322715e-009,             nan,
       3.54561661e-002, 2.96762135e-001, 1.46534278e-021, 5.75512476e-022,
                   nan, 7.41390255e-037, 4.81448017e-020,             nan,
       3.57634770e-001, 2.33701849e-009, 1.99602186e-024, 6.96385914e-001,
       7.74742611e-001, 8.14003373e-063, 2.41707770e-047, 2.99192528e-029,
       1.08009014e-007, 2.66903653e-003, 2.31763854e-037, 3.03638806e-004,
       6.70897439e-027, 0.00000000e+000, 6.73535680e-091]))
```

[ ]:

```
#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                    1.831500e-05
length_hostname               1.125774e-04
ip                           4.196711e-149
nb_dots                       1.930556e-05
nb_hyphens                    1.408109e-03
nb_at                         8.852500e-12
nb_qm                         1.906792e-47
nb_and                        2.670078e-12
nb_or                                  NaN
nb_eq                         4.013379e-18
nb_underscore                 2.257006e-01
nb_tilde                      9.988939e-03
nb_percent                    3.599814e-01
nb_slash                      6.612998e-07
nb_star                       1.476108e-02
nb_colon                      5.345063e-07
nb_comma                      8.471166e-01
nb_semicolumn                 1.137903e-06
nb_dollar                     1.040648e-01
nb_space                      7.656275e-01
nb_www                        1.195913e-97
nb_com                        1.902682e-09
```

nb_dslash	7.865181e-10
http_in_path	2.869793e-05
https_token	1.138833e-10
ratio_digits_url	9.535001e-52
ratio_digits_host	1.206770e-38
punycode	8.469710e-02
port	9.321046e-02
tld_in_path	3.330104e-11
tld_in_subdomain	1.024456e-70
abnormal_subdomain	4.935210e-27
nb_subdomains	7.542103e-05
prefix_suffix	5.867020e-65
random_domain	2.416595e-01
shortening_service	2.928386e-19
path_extension	9.947819e-01
nb_redirection	4.053259e-01
nb_external_redirection	3.864910e-07
length_words_raw	5.456590e-05
char_repeat	8.594907e-01
shortest_words_raw	3.895387e-01
shortest_word_host	2.528147e-10
shortest_word_path	6.605494e-02
longest_words_raw	1.642175e-06
longest_word_host	1.926411e-02
longest_word_path	2.496308e-09
avg_words_raw	8.059408e-03
avg_word_host	1.059835e-04
avg_word_path	1.243418e-04
phish_hints	2.907711e-43
domain_in_brand	3.081978e-15
brand_in_subdomain	6.473228e-09
brand_in_path	4.086318e-08
suspecious_tld	7.038273e-23
statistical_report	4.866267e-35
nb_hyperlinks	3.642303e-16
ratio_intHyperlinks	1.417689e-26
ratio_extHyperlinks	5.631492e-08
ratio_nullHyperlinks	NaN
nb_extCSS	5.179121e-02
ratio_intRedirection	NaN
ratio_extRedirection	2.933227e-09
ratio_intErrors	NaN
ratio_extErrors	3.545617e-02
login_form	2.967621e-01
external_favicon	1.465343e-21
links_in_tags	5.755125e-22
submit_email	NaN

```
ratio_intMedia                7.413903e-37
ratio_extMedia                4.814480e-20
sfh                                    NaN
iframe                        3.576348e-01
popup_window                  2.337018e-09
safe_anchor                   1.996022e-24
onmouseover                   6.963859e-01
right_clic                    7.747426e-01
empty_title                   8.140034e-63
domain_in_title               2.417078e-47
domain_with_copyright         2.991925e-29
whois_registered_domain       1.080090e-07
domain_registration_length    2.669037e-03
domain_age                    2.317639e-37
web_traffic                   3.036388e-04
dns_record                    6.708974e-27
google_index                  0.000000e+00
page_rank                     6.735357e-91
dtype: float64
```

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

[ ]:
```
path_extension                9.947819e-01
char_repeat                   8.594907e-01
nb_comma                      8.471166e-01
right_clic                    7.747426e-01
nb_space                      7.656275e-01
onmouseover                   6.963859e-01
nb_redirection                4.053259e-01
shortest_words_raw            3.895387e-01
nb_percent                    3.599814e-01
iframe                        3.576348e-01
login_form                    2.967621e-01
random_domain                 2.416595e-01
nb_underscore                 2.257006e-01
nb_dollar                     1.040648e-01
port                          9.321046e-02
punycode                      8.469710e-02
shortest_word_path            6.605494e-02
nb_extCSS                     5.179121e-02
ratio_extErrors               3.545617e-02
longest_word_host             1.926411e-02
nb_star                       1.476108e-02
nb_tilde                      9.988939e-03
avg_words_raw                 8.059408e-03
```

domain_registration_length	2.669037e-03
nb_hyphens	1.408109e-03
web_traffic	3.036388e-04
avg_word_path	1.243418e-04
length_hostname	1.125774e-04
avg_word_host	1.059835e-04
nb_subdomains	7.542103e-05
length_words_raw	5.456590e-05
http_in_path	2.869793e-05
nb_dots	1.930556e-05
length_url	1.831500e-05
longest_words_raw	1.642175e-06
nb_semicolumn	1.137903e-06
nb_slash	6.612998e-07
nb_colon	5.345063e-07
nb_external_redirection	3.864910e-07
whois_registered_domain	1.080090e-07
ratio_extHyperlinks	5.631492e-08
brand_in_path	4.086318e-08
brand_in_subdomain	6.473228e-09
ratio_extRedirection	2.933227e-09
longest_word_path	2.496308e-09
popup_window	2.337018e-09
nb_com	1.902682e-09
nb_dslash	7.865181e-10
shortest_word_host	2.528147e-10
https_token	1.138833e-10
tld_in_path	3.330104e-11
nb_at	8.852500e-12
nb_and	2.670078e-12
domain_in_brand	3.081978e-15
nb_hyperlinks	3.642303e-16
nb_eq	4.013379e-18
shortening_service	2.928386e-19
ratio_extMedia	4.814480e-20
external_favicon	1.465343e-21
links_in_tags	5.755125e-22
suspecious_tld	7.038273e-23
safe_anchor	1.996022e-24
ratio_intHyperlinks	1.417689e-26
dns_record	6.708974e-27
abnormal_subdomain	4.935210e-27
domain_with_copyright	2.991925e-29
statistical_report	4.866267e-35
ratio_intMedia	7.413903e-37
domain_age	2.317639e-37
ratio_digits_host	1.206770e-38

```
phish_hints                2.907711e-43
domain_in_title            2.417078e-47
nb_qm                      1.906792e-47
ratio_digits_url           9.535001e-52
empty_title                8.140034e-63
prefix_suffix              5.867020e-65
tld_in_subdomain           1.024456e-70
page_rank                  6.735357e-91
nb_www                     1.195913e-97
ip                        4.196711e-149
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))
```

[ ]:
```
24
```

[ ]:
```
drop_feature
```

[ ]:
```
{'char_repeat',
 'iframe',
 'login_form',
 'nb_comma',
 'nb_dollar',
 'nb_extCSS',
 '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_word_path',
 '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)
```

[ ]:
```
63
```

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

[ ]:
```
63
```

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

```
Training set has 8001 samples.
Testing set has 3429 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)
0.9407573345817728
```

[ ]:
```
logr_model = grid_logr.best_estimator_

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

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

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

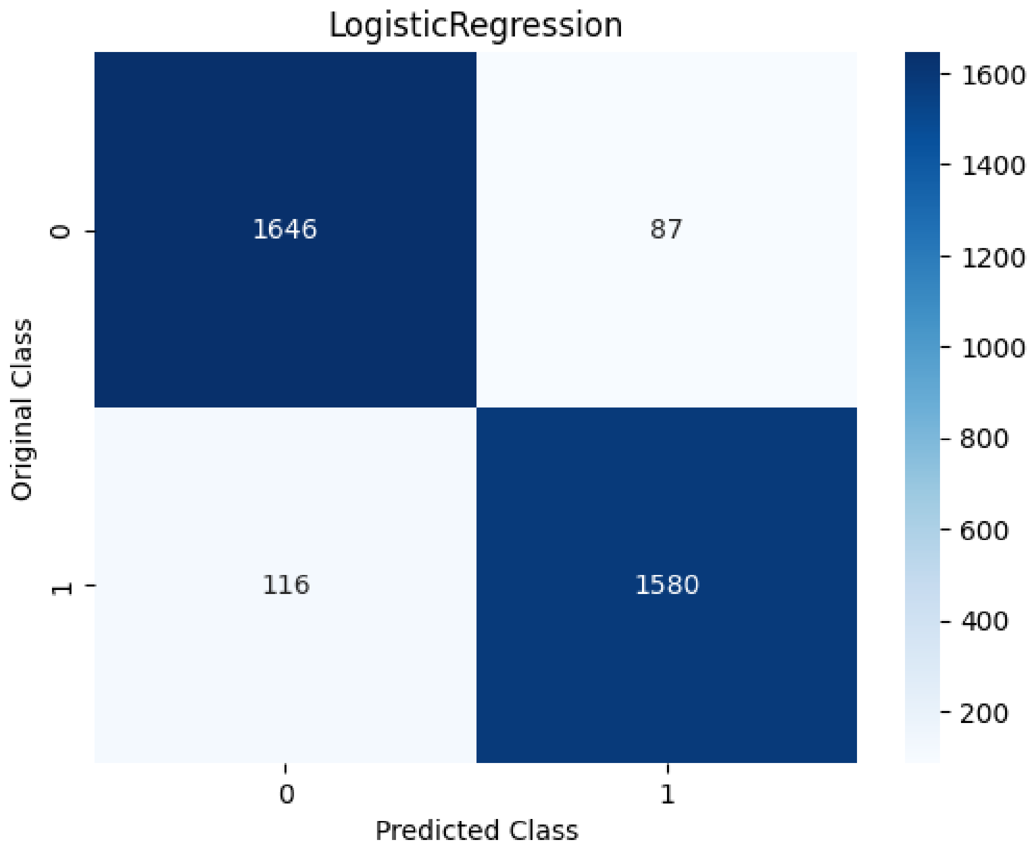
[ ]:
```
print ("Accuracy of logr classifier : ", accuracy_score(y_test,␣
↪logr_predict)*100)
```

```
Accuracy of logr classifier :  94.07990667833187
 94.07990667833187
```

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

```
              precision    recall  f1-score   support

           0       0.93      0.95      0.94      1733
           1       0.95      0.93      0.94      1696

    accuracy                           0.94      3429
   macro avg       0.94      0.94      0.94      3429
weighted avg       0.94      0.94      0.94      3429
```

[ ]:
```
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': 3}
KNeighborsClassifier(n_neighbors=3)
0.9241348314606741
```

```
[ ]: 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))
```

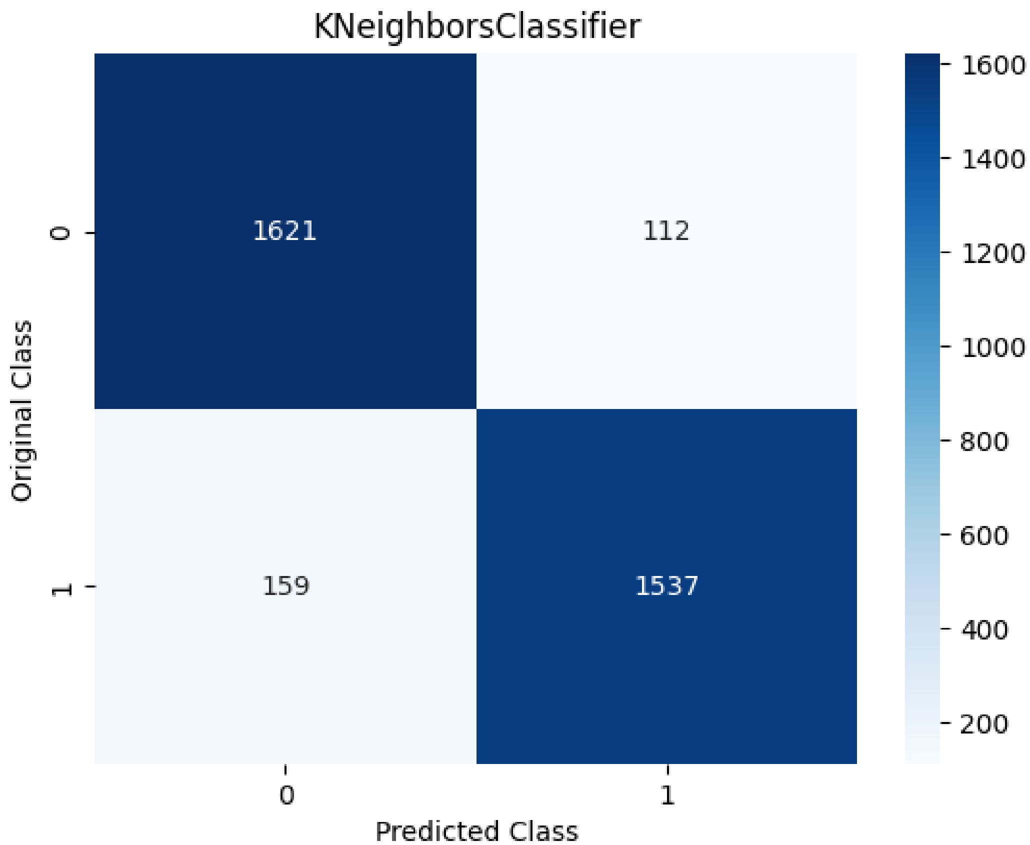
```
The accuracy of knn Classifier is:  92.0968212306795
```

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

```
              precision    recall  f1-score   support

           0       0.91      0.94      0.92      1733
           1       0.93      0.91      0.92      1696

    accuracy                           0.92      3429
   macro avg       0.92      0.92      0.92      3429
weighted avg       0.92      0.92      0.92      3429
```

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

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



[ ]:
```
# # 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")
```

[ ]:
```
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)
0.9538804619225967
```

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

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

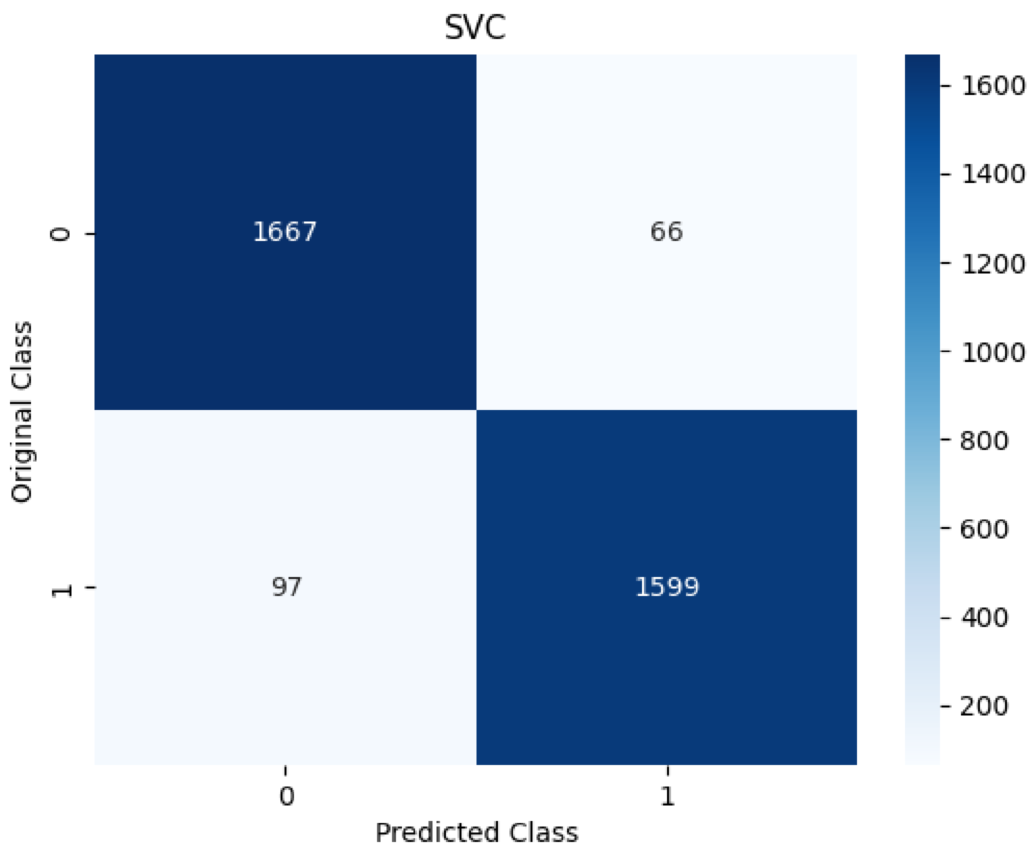
[ ]:
```
print('The accuracy of svc Classifier is: ', 100.0 * accuracy_score(y_test,␣
↪svc_predict))
```

```
The accuracy of svc Classifier is:  95.2464275298921
```

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

```
              precision    recall  f1-score   support

           0       0.95      0.96      0.95      1733
           1       0.96      0.94      0.95      1696

    accuracy                           0.95      3429
   macro avg       0.95      0.95      0.95      3429
weighted avg       0.95      0.95      0.95      3429
```

[ ]:
```
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())
```

```python
# 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': 0.01, 'kernel': 'rbf', 'nu': 0.1}
NuSVC(gamma=0.01, nu=0.1)
0.9541304619225967
```

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

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

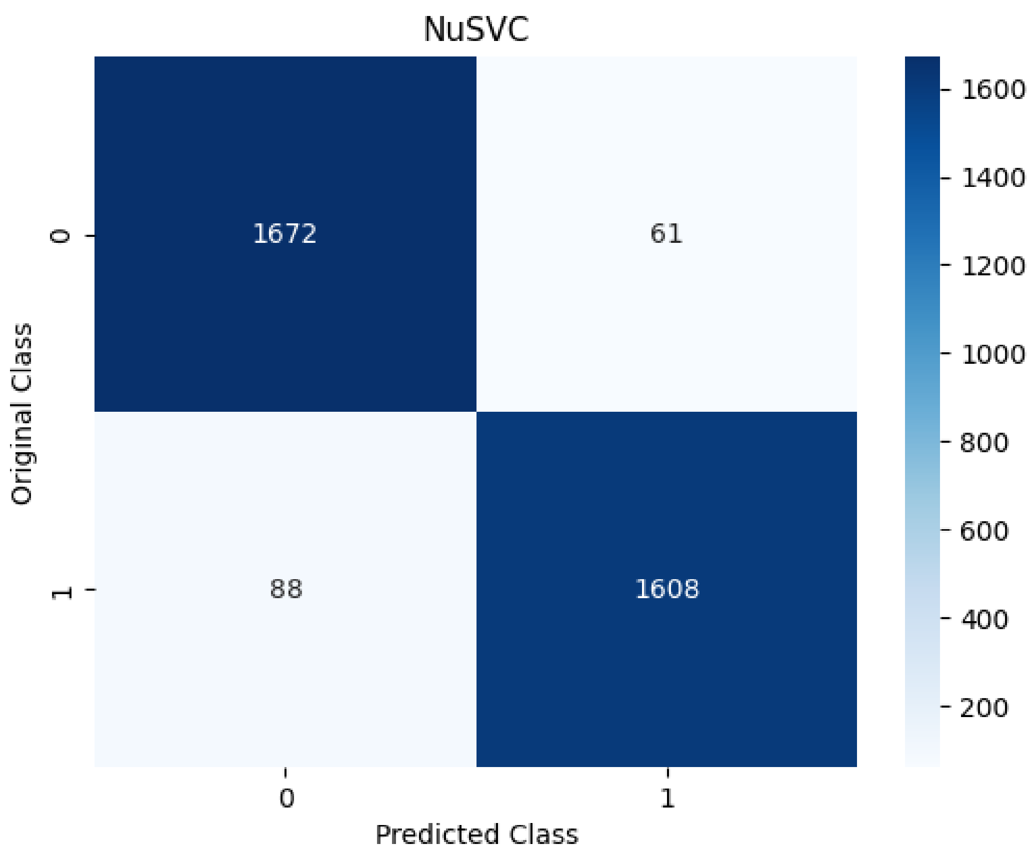
[ ]:
```python
print('The accuracy of nusvc Classifier is: ', 100.0 * accuracy_score(y_test,␣
↪nusvc_predict))
```

```
The accuracy of nusvc Classifier is:  95.65470982793818
```

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

```
              precision    recall  f1-score   support

           0       0.95      0.96      0.96      1733
           1       0.96      0.95      0.96      1696

    accuracy                           0.96      3429
   macro avg       0.96      0.96      0.96      3429
weighted avg       0.96      0.96      0.96      3429
```

[ ]:
```python
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': 20, 'dual': False, 'loss': 'squared_hinge', 'penalty': 'l1', 'tol': 0.001}
LinearSVC(C=20, dual=False, penalty='l1', tol=0.001)
0.9408823345817726
```

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

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

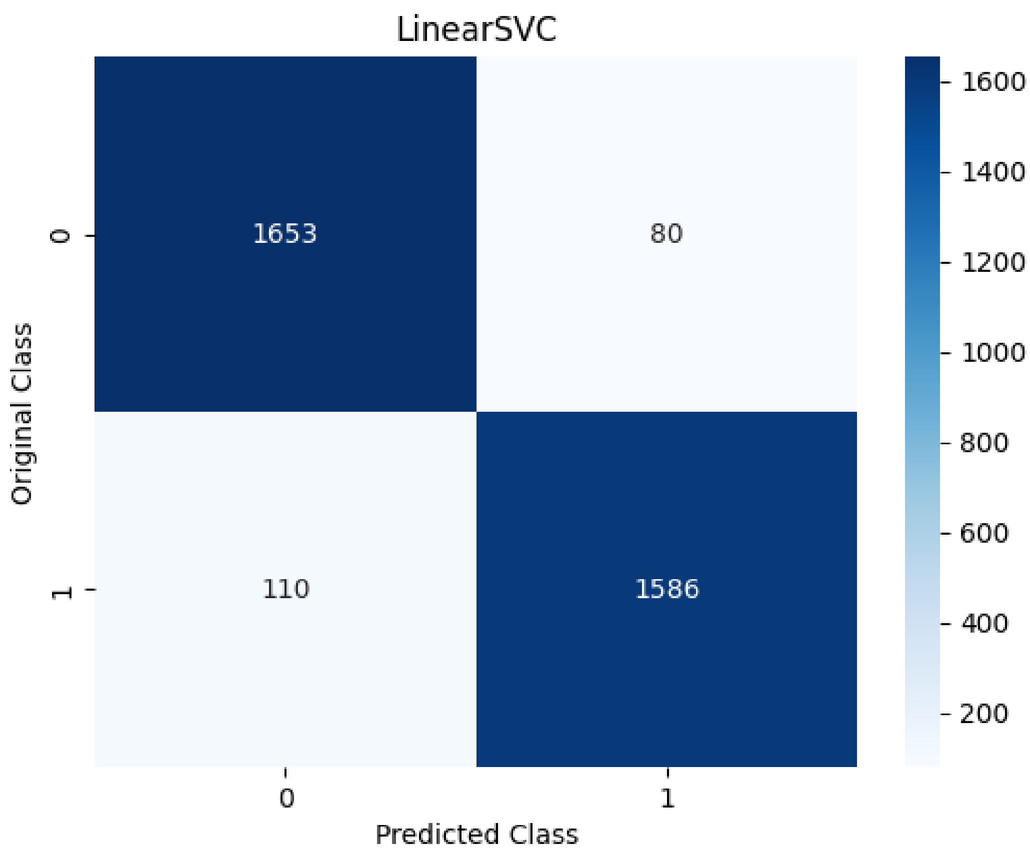
[ ]:
```
print('The accuracy of lsvc Classifier is: ', 100.0 * accuracy_score(y_test,␣
↪lsvc_predict))
```

```
The accuracy of lsvc Classifier is:  94.45902595508895
```

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

```
              precision    recall  f1-score   support

           0       0.94      0.95      0.95      1733
           1       0.95      0.94      0.94      1696

    accuracy                           0.94      3429
   macro avg       0.94      0.94      0.94      3429
weighted avg       0.94      0.94      0.94      3429
```

[ ]:
```
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()
```




```
[ ]: 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)
0.9546310861423221
```

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

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

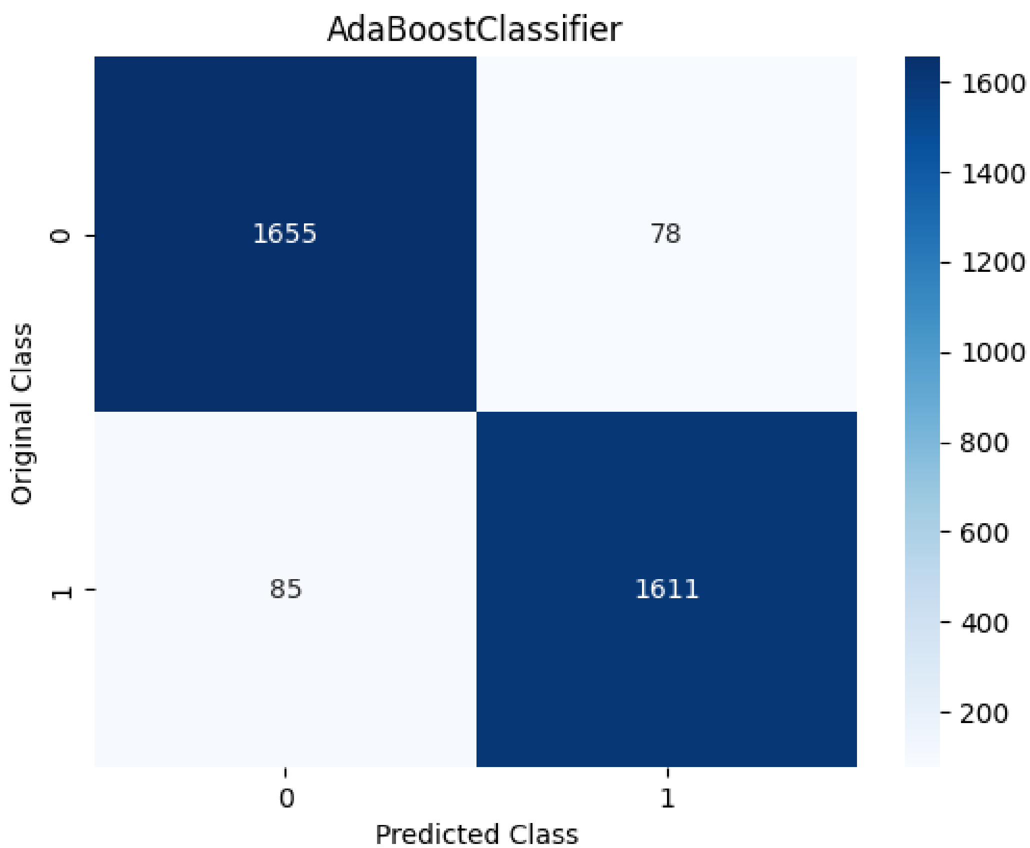
[ ]:
```
print('The accuracy of Ada Boost Classifier is: ', 100.0 *␣
↪accuracy_score(ada_predict,y_test))
```

```
The accuracy of Ada Boost Classifier is:  95.2464275298921
```

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

```
              precision    recall  f1-score   support

           0       0.95      0.95      0.95      1733
           1       0.95      0.95      0.95      1696

    accuracy                           0.95      3429
   macro avg       0.95      0.95      0.95      3429
weighted avg       0.95      0.95      0.95      3429
```

[ ]:
```
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': 0.01, 'n_estimators': 150}
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=0.01, 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=150,
              n_jobs=0, num_parallel_tree=1, predictor='auto', random_state=0,
              reg_alpha=0, reg_lambda=1, …)
0.9695029650436953
```

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

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

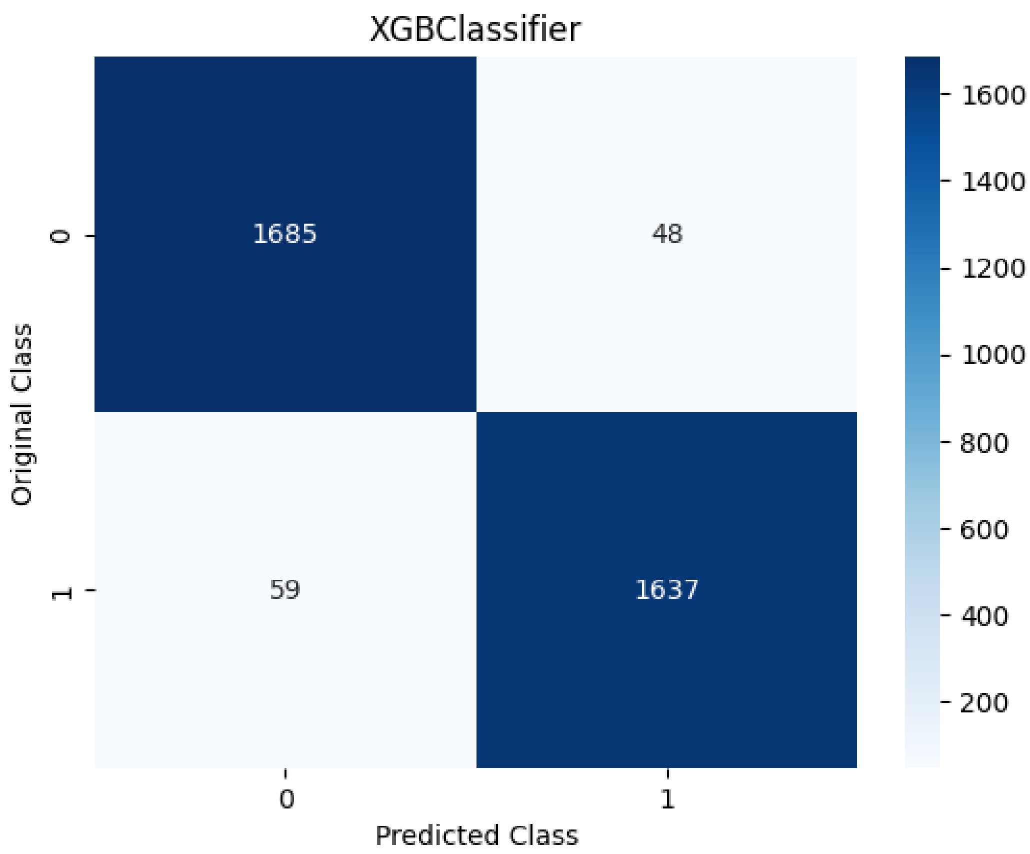
[ ]:
```
print('The accuracy of XGBoost Classifier is: ' , 100.0 *␣
 ↪accuracy_score(xgb_predict,y_test))
```

```
The accuracy of XGBoost Classifier is:  96.87955672207642
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      1733
           1       0.97      0.97      0.97      1696

    accuracy                           0.97      3429
   macro avg       0.97      0.97      0.97      3429
weighted avg       0.97      0.97      0.97      3429
```

[ ]:
```
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()
```



[ ]:
```
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)
0.9680031210986266
```

[ ]:
```
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)
```

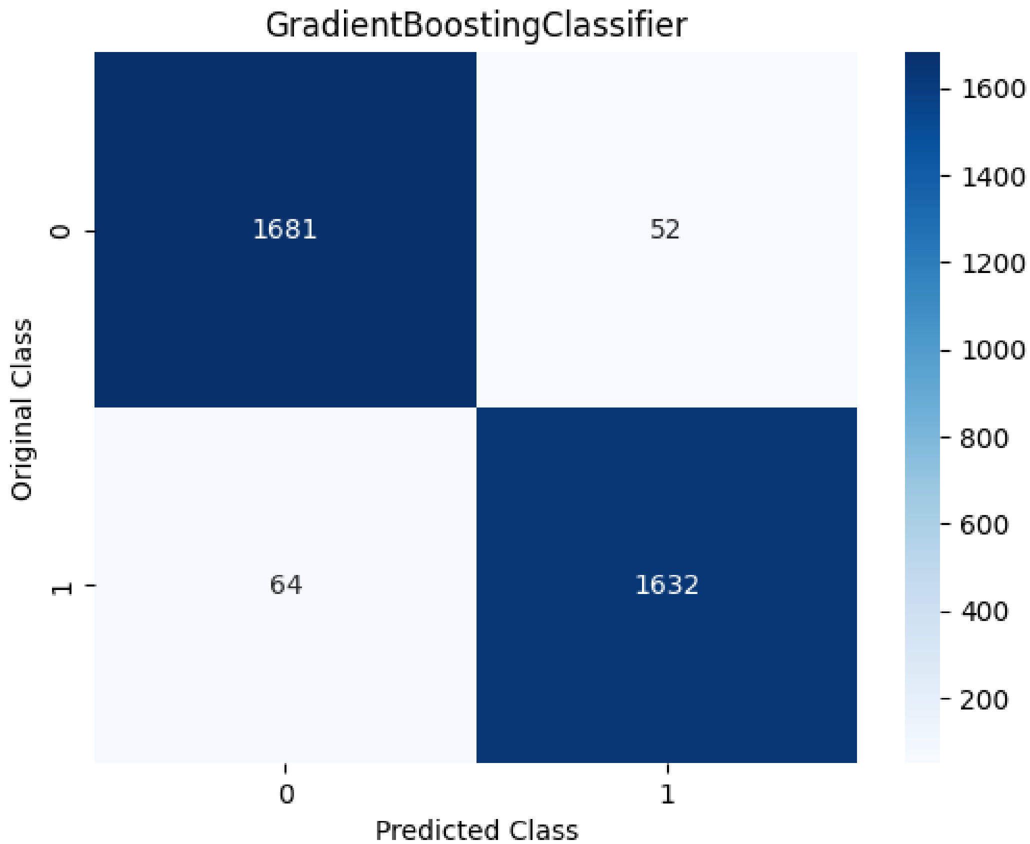
[ ]:
```
print('The accuracy of GradientBoost Classifier is: ' , 100.0 *␣
 ↪accuracy_score(gbc_predict,y_test))
```

```
The accuracy of GradientBoost Classifier is:  96.61708953047535
```

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

```
              precision    recall  f1-score   support

           0       0.96      0.97      0.97      1733
           1       0.97      0.96      0.97      1696

    accuracy                           0.97      3429
   macro avg       0.97      0.97      0.97      3429
weighted avg       0.97      0.97      0.97      3429
```

[ ]:
```
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': 200}
BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=200)
0.9576295255930087
```

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

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

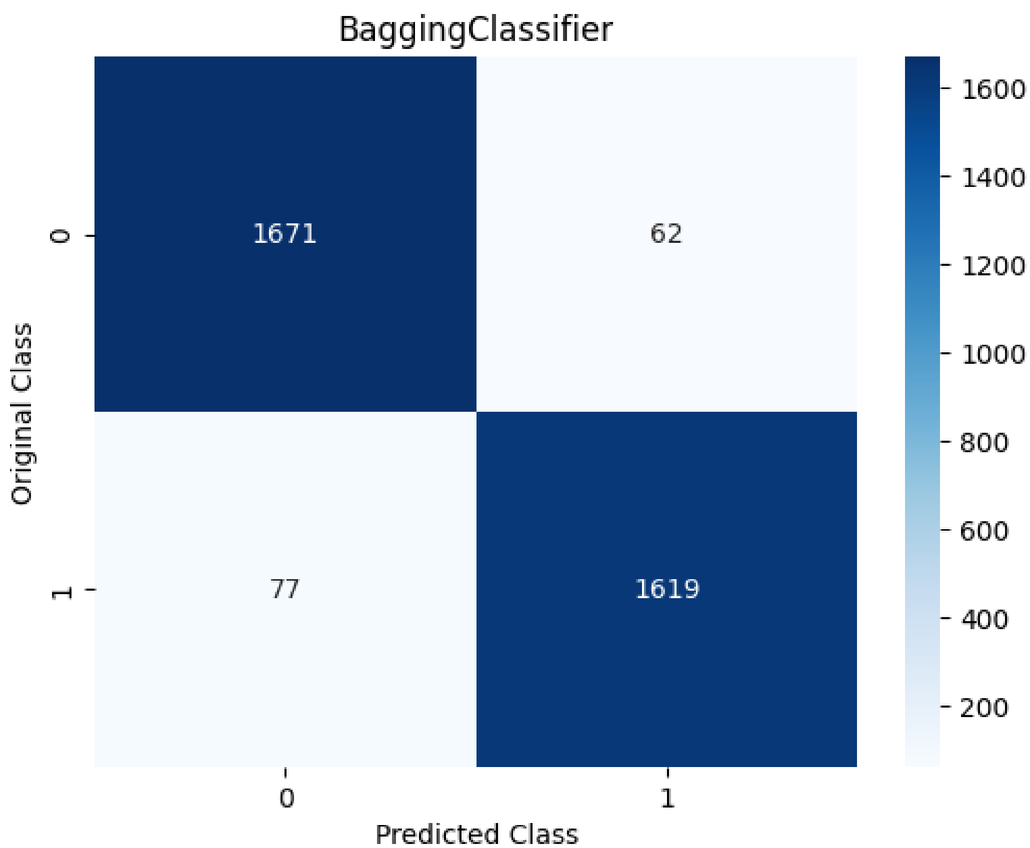
[ ]:
```
print('The accuracy of Bagging Classifier is: ' , 100.0 *
    accuracy_score(y_test, bag_predict))
```

```
The accuracy of Bagging Classifier is:  95.94634004082823
```

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

```
              precision    recall  f1-score   support

           0       0.96      0.96      0.96      1733
           1       0.96      0.95      0.96      1696

    accuracy                           0.96      3429
   macro avg       0.96      0.96      0.96      3429
weighted avg       0.96      0.96      0.96      3429
```

[ ]:
```
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': 150}
RandomForestClassifier(n_estimators=150)
0.9645040574282149
```

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

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

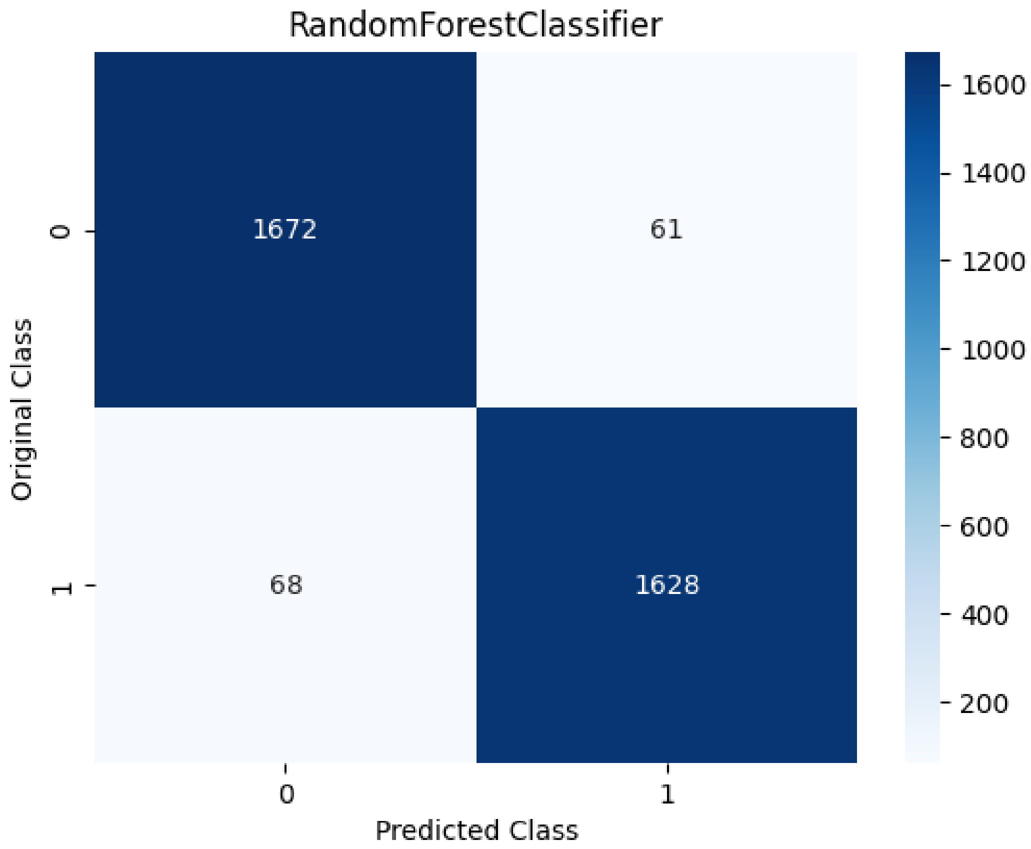
[ ]:
```
print('The accuracy of RandomForest Classifier is: ' , 100.0 *␣
↪accuracy_score(rfc_predict,y_test))
```

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

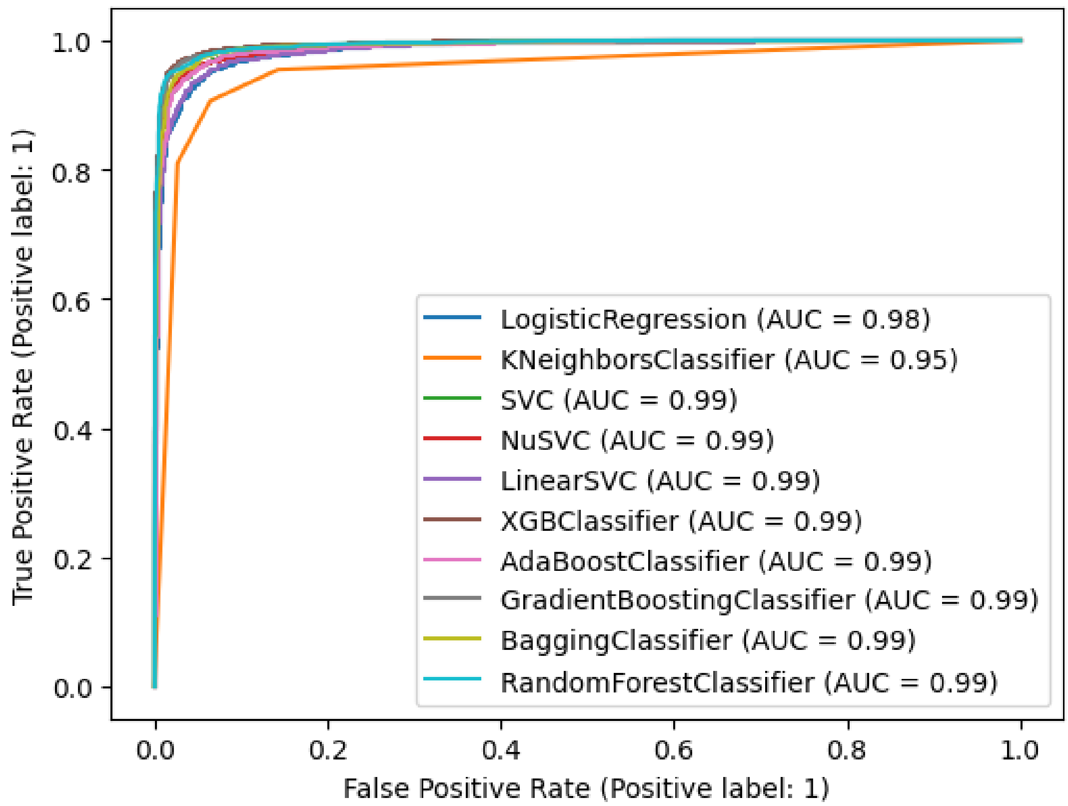
[ ]:
```
print(classification_report(y_test, rfc_predict))
```

```
              precision    recall  f1-score   support

           0       0.96      0.96      0.96      1733
           1       0.96      0.96      0.96      1696

    accuracy                           0.96      3429
   macro avg       0.96      0.96      0.96      3429
weighted avg       0.96      0.96      0.96      3429
```

[ ]:
```
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())
```



[ ]:
```python
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)               65600

 batch_normalization (BatchN  (None, 100)              400
 ormalization)


_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 100)               65600

 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: 72,596
Trainable params: 72,396
Non-trainable params: 200
_________________________________________________________________
Epoch 1/100
200/200 [==============================] - 3s 7ms/step - loss: 0.2308 -
accuracy: 0.9114 - val_loss: 0.4555 - val_accuracy: 0.8857
Epoch 2/100
200/200 [==============================] - 1s 4ms/step - loss: 0.1608 -
accuracy: 0.9383 - val_loss: 0.2597 - val_accuracy: 0.9313
```

```
Epoch 3/100
200/200 [==============================] - 1s 4ms/step - loss: 0.1476 -
accuracy: 0.9447 - val_loss: 0.1707 - val_accuracy: 0.9363
Epoch 4/100
200/200 [==============================] - 1s 4ms/step - loss: 0.1309 -
accuracy: 0.9489 - val_loss: 0.1570 - val_accuracy: 0.9394
Epoch 5/100
200/200 [==============================] - 1s 4ms/step - loss: 0.1320 -
accuracy: 0.9481 - val_loss: 0.1555 - val_accuracy: 0.9375
Epoch 6/100
200/200 [==============================] - 1s 4ms/step - loss: 0.1168 -
accuracy: 0.9564 - val_loss: 0.1565 - val_accuracy: 0.9463
Epoch 7/100
200/200 [==============================] - 1s 4ms/step - loss: 0.1144 -
accuracy: 0.9555 - val_loss: 0.1541 - val_accuracy: 0.9438
Epoch 8/100
200/200 [==============================] - 1s 4ms/step - loss: 0.1008 -
accuracy: 0.9633 - val_loss: 0.1875 - val_accuracy: 0.9269
Epoch 9/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0978 -
accuracy: 0.9633 - val_loss: 0.1482 - val_accuracy: 0.9469
Epoch 10/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0972 -
accuracy: 0.9644 - val_loss: 0.1729 - val_accuracy: 0.9444
Epoch 11/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0868 -
accuracy: 0.9683 - val_loss: 0.1705 - val_accuracy: 0.9419
Epoch 12/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0823 -
accuracy: 0.9683 - val_loss: 0.1720 - val_accuracy: 0.9444
Epoch 13/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0807 -
accuracy: 0.9695 - val_loss: 0.1669 - val_accuracy: 0.9507
Epoch 14/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0751 -
accuracy: 0.9728 - val_loss: 0.1681 - val_accuracy: 0.9463
Epoch 15/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0693 -
accuracy: 0.9728 - val_loss: 0.1679 - val_accuracy: 0.9463
Epoch 16/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0693 -
accuracy: 0.9745 - val_loss: 0.1724 - val_accuracy: 0.9482
Epoch 17/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0713 -
accuracy: 0.9745 - val_loss: 0.1617 - val_accuracy: 0.9488
Epoch 18/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0545 -
accuracy: 0.9798 - val_loss: 0.1809 - val_accuracy: 0.9482
```

```
Epoch 19/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0596 -
accuracy: 0.9773 - val_loss: 0.1892 - val_accuracy: 0.9488
Epoch 20/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0571 -
accuracy: 0.9809 - val_loss: 0.1786 - val_accuracy: 0.9494
Epoch 21/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0500 -
accuracy: 0.9811 - val_loss: 0.2059 - val_accuracy: 0.9488
Epoch 22/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0470 -
accuracy: 0.9802 - val_loss: 0.2067 - val_accuracy: 0.9463
Epoch 23/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0479 -
accuracy: 0.9814 - val_loss: 0.1982 - val_accuracy: 0.9438
Epoch 24/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0428 -
accuracy: 0.9853 - val_loss: 0.2246 - val_accuracy: 0.9407
Epoch 25/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0397 -
accuracy: 0.9859 - val_loss: 0.2422 - val_accuracy: 0.9369
Epoch 26/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0489 -
accuracy: 0.9822 - val_loss: 0.2066 - val_accuracy: 0.9444
Epoch 27/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0362 -
accuracy: 0.9883 - val_loss: 0.2038 - val_accuracy: 0.9457
Epoch 28/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0317 -
accuracy: 0.9889 - val_loss: 0.2083 - val_accuracy: 0.9538
Epoch 29/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0397 -
accuracy: 0.9847 - val_loss: 0.2259 - val_accuracy: 0.9425
Epoch 30/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0361 -
accuracy: 0.9861 - val_loss: 0.2467 - val_accuracy: 0.9482
Epoch 31/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0390 -
accuracy: 0.9853 - val_loss: 0.2239 - val_accuracy: 0.9450
Epoch 32/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0298 -
accuracy: 0.9898 - val_loss: 0.2355 - val_accuracy: 0.9444
Epoch 33/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0279 -
accuracy: 0.9900 - val_loss: 0.2434 - val_accuracy: 0.9457
Epoch 34/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0255 -
accuracy: 0.9908 - val_loss: 0.2640 - val_accuracy: 0.9432
```

```
Epoch 35/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0274 -
accuracy: 0.9895 - val_loss: 0.2579 - val_accuracy: 0.9444
Epoch 36/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0274 -
accuracy: 0.9886 - val_loss: 0.2768 - val_accuracy: 0.9450
Epoch 37/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0305 -
accuracy: 0.9891 - val_loss: 0.2717 - val_accuracy: 0.9469
Epoch 38/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0242 -
accuracy: 0.9900 - val_loss: 0.2795 - val_accuracy: 0.9444
Epoch 39/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0271 -
accuracy: 0.9905 - val_loss: 0.2602 - val_accuracy: 0.9444
Epoch 40/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0252 -
accuracy: 0.9891 - val_loss: 0.3124 - val_accuracy: 0.9388
Epoch 41/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0265 -
accuracy: 0.9892 - val_loss: 0.3089 - val_accuracy: 0.9469
Epoch 42/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0221 -
accuracy: 0.9919 - val_loss: 0.2918 - val_accuracy: 0.9457
Epoch 43/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0155 -
accuracy: 0.9948 - val_loss: 0.2804 - val_accuracy: 0.9438
Epoch 44/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0233 -
accuracy: 0.9905 - val_loss: 0.3000 - val_accuracy: 0.9444
Epoch 45/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0210 -
accuracy: 0.9919 - val_loss: 0.2950 - val_accuracy: 0.9444
Epoch 46/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0210 -
accuracy: 0.9937 - val_loss: 0.2967 - val_accuracy: 0.9432
Epoch 47/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0300 -
accuracy: 0.9884 - val_loss: 0.3055 - val_accuracy: 0.9507
Epoch 48/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0233 -
accuracy: 0.9908 - val_loss: 0.3070 - val_accuracy: 0.9419
Epoch 49/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0228 -
accuracy: 0.9914 - val_loss: 0.3000 - val_accuracy: 0.9438
Epoch 50/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0166 -
accuracy: 0.9941 - val_loss: 0.2960 - val_accuracy: 0.9488
```

```
Epoch 51/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0141 -
accuracy: 0.9944 - val_loss: 0.3209 - val_accuracy: 0.9463
Epoch 52/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0194 -
accuracy: 0.9934 - val_loss: 0.2967 - val_accuracy: 0.9507
Epoch 53/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0159 -
accuracy: 0.9936 - val_loss: 0.3419 - val_accuracy: 0.9444
Epoch 54/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0138 -
accuracy: 0.9944 - val_loss: 0.3737 - val_accuracy: 0.9375
Epoch 55/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0211 -
accuracy: 0.9920 - val_loss: 0.3517 - val_accuracy: 0.9450
Epoch 56/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0260 -
accuracy: 0.9908 - val_loss: 0.3528 - val_accuracy: 0.9475
Epoch 57/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0208 -
accuracy: 0.9925 - val_loss: 0.3251 - val_accuracy: 0.9469
Epoch 58/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0141 -
accuracy: 0.9955 - val_loss: 0.2952 - val_accuracy: 0.9463
Epoch 59/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0231 -
accuracy: 0.9927 - val_loss: 0.2986 - val_accuracy: 0.9513
Epoch 60/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0105 -
accuracy: 0.9966 - val_loss: 0.3037 - val_accuracy: 0.9482
Epoch 61/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0104 -
accuracy: 0.9962 - val_loss: 0.3408 - val_accuracy: 0.9482
Epoch 62/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0098 -
accuracy: 0.9967 - val_loss: 0.3495 - val_accuracy: 0.9444
Epoch 63/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0105 -
accuracy: 0.9959 - val_loss: 0.3585 - val_accuracy: 0.9519
Epoch 64/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0165 -
accuracy: 0.9941 - val_loss: 0.3367 - val_accuracy: 0.9450
Epoch 65/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0164 -
accuracy: 0.9939 - val_loss: 0.3568 - val_accuracy: 0.9432
Epoch 66/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0143 -
accuracy: 0.9950 - val_loss: 0.3604 - val_accuracy: 0.9400
```

```
Epoch 67/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0118 -
accuracy: 0.9952 - val_loss: 0.3776 - val_accuracy: 0.9369
Epoch 68/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0217 -
accuracy: 0.9930 - val_loss: 0.3652 - val_accuracy: 0.9444
Epoch 69/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0133 -
accuracy: 0.9948 - val_loss: 0.3219 - val_accuracy: 0.9457
Epoch 70/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0127 -
accuracy: 0.9955 - val_loss: 0.3721 - val_accuracy: 0.9432
Epoch 71/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0106 -
accuracy: 0.9964 - val_loss: 0.3627 - val_accuracy: 0.9425
Epoch 72/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0152 -
accuracy: 0.9944 - val_loss: 0.3949 - val_accuracy: 0.9382
Epoch 73/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0121 -
accuracy: 0.9961 - val_loss: 0.3501 - val_accuracy: 0.9457
Epoch 74/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0086 -
accuracy: 0.9973 - val_loss: 0.3629 - val_accuracy: 0.9463
Epoch 75/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0118 -
accuracy: 0.9964 - val_loss: 0.3715 - val_accuracy: 0.9482
Epoch 76/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0104 -
accuracy: 0.9959 - val_loss: 0.3713 - val_accuracy: 0.9438
Epoch 77/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0128 -
accuracy: 0.9958 - val_loss: 0.3764 - val_accuracy: 0.9432
Epoch 78/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0149 -
accuracy: 0.9948 - val_loss: 0.3517 - val_accuracy: 0.9550
Epoch 79/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0154 -
accuracy: 0.9948 - val_loss: 0.3685 - val_accuracy: 0.9475
Epoch 80/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0136 -
accuracy: 0.9950 - val_loss: 0.3733 - val_accuracy: 0.9469
Epoch 81/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0158 -
accuracy: 0.9947 - val_loss: 0.3906 - val_accuracy: 0.9425
Epoch 82/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0102 -
accuracy: 0.9962 - val_loss: 0.3616 - val_accuracy: 0.9444
```

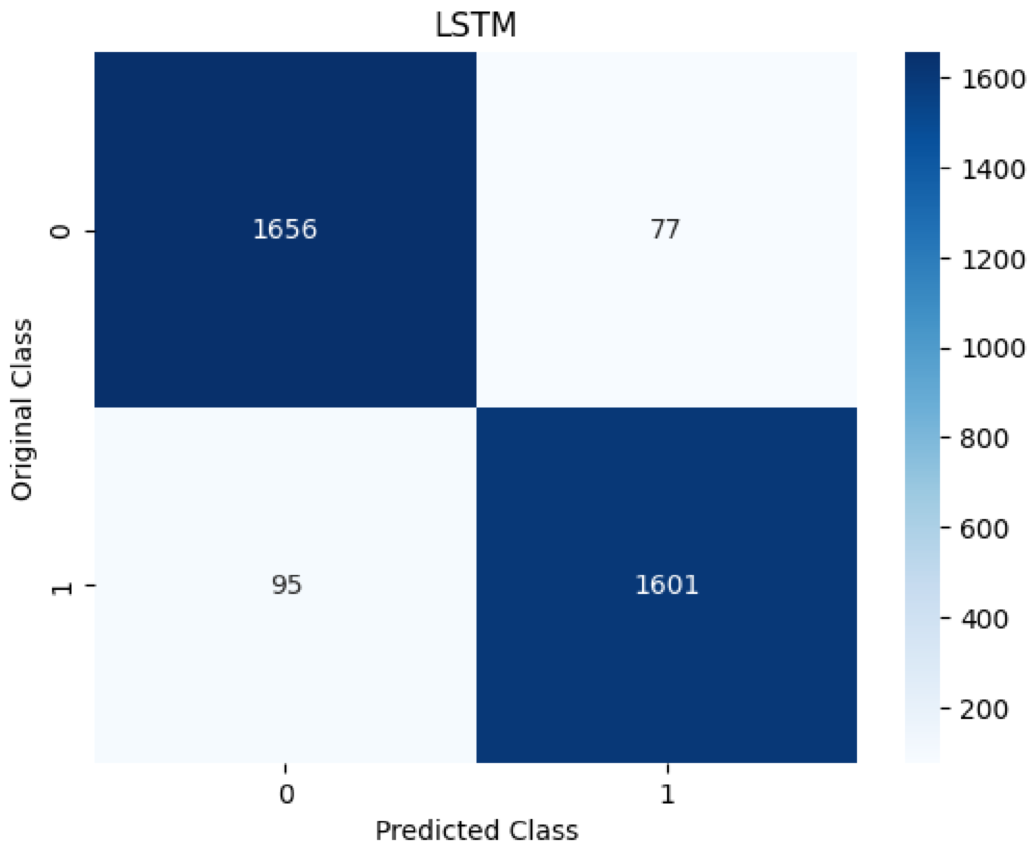
```
Epoch 83/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0119 -
accuracy: 0.9962 - val_loss: 0.3441 - val_accuracy: 0.9500
Epoch 84/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0096 -
accuracy: 0.9961 - val_loss: 0.3904 - val_accuracy: 0.9444
Epoch 85/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0089 -
accuracy: 0.9961 - val_loss: 0.3752 - val_accuracy: 0.9469
Epoch 86/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0040 -
accuracy: 0.9987 - val_loss: 0.4067 - val_accuracy: 0.9482
Epoch 87/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0083 -
accuracy: 0.9975 - val_loss: 0.3870 - val_accuracy: 0.9507
Epoch 88/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0165 -
accuracy: 0.9950 - val_loss: 0.3700 - val_accuracy: 0.9475
Epoch 89/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0086 -
accuracy: 0.9967 - val_loss: 0.3902 - val_accuracy: 0.9457
Epoch 90/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0120 -
accuracy: 0.9959 - val_loss: 0.3878 - val_accuracy: 0.9450
Epoch 91/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0118 -
accuracy: 0.9967 - val_loss: 0.4074 - val_accuracy: 0.9444
Epoch 92/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0085 -
accuracy: 0.9970 - val_loss: 0.3879 - val_accuracy: 0.9488
Epoch 93/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0096 -
accuracy: 0.9959 - val_loss: 0.4371 - val_accuracy: 0.9488
Epoch 94/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0121 -
accuracy: 0.9959 - val_loss: 0.4165 - val_accuracy: 0.9457
Epoch 95/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0051 -
accuracy: 0.9984 - val_loss: 0.4208 - val_accuracy: 0.9457
Epoch 96/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0083 -
accuracy: 0.9964 - val_loss: 0.4265 - val_accuracy: 0.9463
Epoch 97/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0078 -
accuracy: 0.9970 - val_loss: 0.4157 - val_accuracy: 0.9482
Epoch 98/100
200/200 [==============================] - 1s 5ms/step - loss: 0.0130 -
accuracy: 0.9961 - val_loss: 0.4469 - val_accuracy: 0.9388
```

```
Epoch 99/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0078 -
accuracy: 0.9975 - val_loss: 0.4458 - val_accuracy: 0.9400
Epoch 100/100
200/200 [==============================] - 1s 4ms/step - loss: 0.0073 -
accuracy: 0.9980 - val_loss: 0.4259 - val_accuracy: 0.9438
Test results - Loss: 0.35732439160346985 - Accuracy: 94.98395919799805%
```

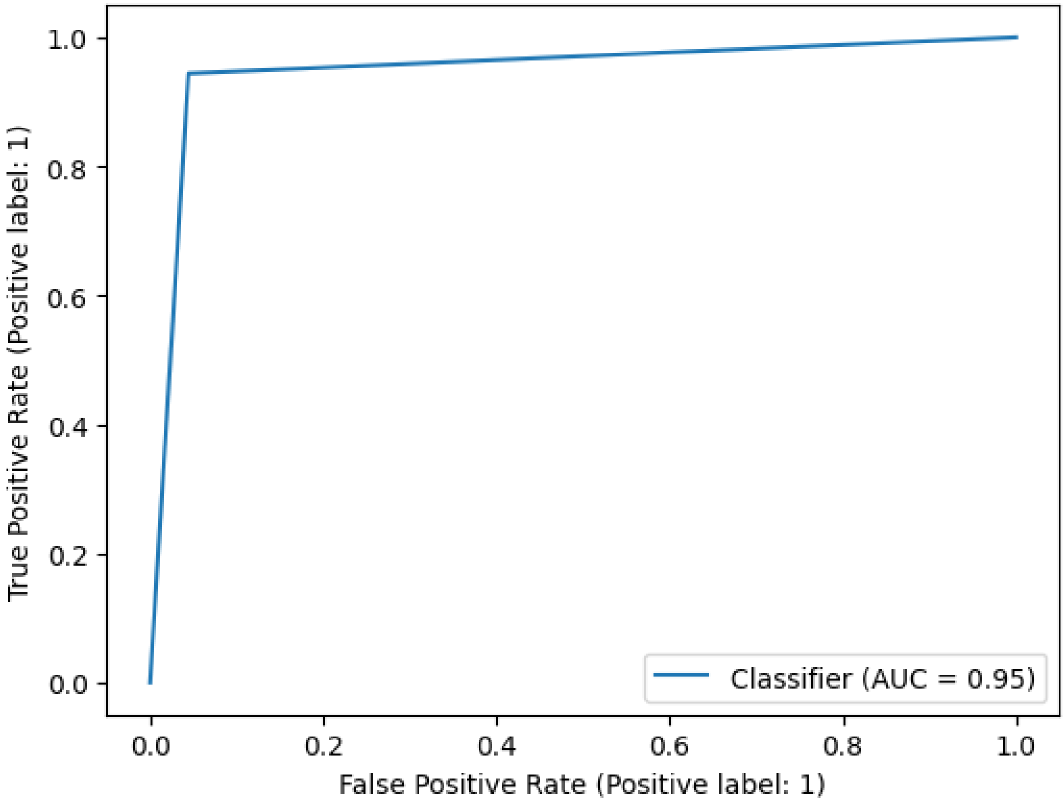
[ ]:
```
lstm_predict_proba = lstm_model.predict(X_test_reshape, batch_size=32)
lstm_predict_class = (lstm_predict_proba > 0.5).astype("int32")
print(classification_report(y_test, lstm_predict_class))
```

```
108/108 [==============================] - 1s 2ms/step
              precision    recall  f1-score   support

           0       0.95      0.96      0.95      1733
           1       0.95      0.94      0.95      1696

    accuracy                           0.95      3429
   macro avg       0.95      0.95      0.95      3429
weighted avg       0.95      0.95      0.95      3429
```

[ ]:
```
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 = 0.95)