chi_sq_30 8020 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_30.csv')
df.drop(['index'], axis=1, inplace=True)
#df.head()
```

[ ]:
```
# 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[:]

lab_en= preprocessing.LabelEncoder()

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

#df.head(50)
```

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

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

```
Count of Legitimate Websites =  4898
Count of Phishy Websites =  6157
```

[ ]:
```
X = df.drop(['Result'], 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['Result']
y = pd.DataFrame(y)
y.head()
```

[ ]:

	Result
0	0
1	0
2	0
3	0
4	1

[ ]:
```
# 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=0.2,
    random_state=10)

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

[ ]:
```
((8844, 30), (2211, 30), (8844, 1), (2211, 1))
```

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

[ ]:
```
f_p_values
```

[ ]:
```
(array([2.64869869e+01, 5.72067876e+01, 5.91290630e+00, 4.49639532e+00,
        1.72043147e+00, 9.29090279e+02, 5.30714533e+02, 3.04760233e+03,
        3.13840567e+02, 1.46840183e-02, 1.77673975e+00, 2.54189780e+00,
        2.45503080e+02, 2.39284341e+03, 3.56017356e+02, 6.17716334e+02,
        7.02843949e-01, 5.15428896e+00, 4.59309783e+00, 2.03646612e+00,
        7.21769102e-02, 1.20494967e-02, 3.73189907e-03, 5.92825806e+01,
        1.61455855e+01, 5.45332425e+02, 7.23058731e+01, 1.79418393e+01,
        2.42125192e+00, 7.38279593e+00]),
 array([2.65319400e-007, 3.92314877e-014, 1.50303549e-002, 3.39663832e-002,
        1.89637515e-001, 4.65644194e-204, 1.97468784e-117, 0.00000000e+000,
        3.18126126e-070, 9.03550341e-001, 1.82550177e-001, 1.10861464e-001,
        2.48223964e-055, 0.00000000e+000, 2.07398946e-079, 2.34664611e-136,
        4.01829733e-001, 2.31890283e-002, 3.21009451e-002, 1.53566047e-001,
        7.88193177e-001, 9.12591625e-001, 9.51288113e-001, 1.36580212e-014,
        5.86551169e-005, 1.30432301e-120, 1.84298086e-017, 2.27758839e-005,
        1.19699235e-001, 6.58507192e-003]))
```

[ ]:
```
#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
```

[ ]:
```
having_IPhaving_IP_Address      2.653194e-07
URLURL_Length                   3.923149e-14
Shortining_Service              1.503035e-02
having_At_Symbol                3.396638e-02
double_slash_redirecting        1.896375e-01
Prefix_Suffix                  4.656442e-204
having_Sub_Domain              1.974688e-117
SSLfinal_State                  0.000000e+00
Domain_registeration_length     3.181261e-70
Favicon                         9.035503e-01
port                            1.825502e-01
HTTPS_token                     1.108615e-01
Request_URL                     2.482240e-55
URL_of_Anchor                   0.000000e+00
Links_in_tags                   2.073989e-79
SFH                            2.346646e-136
Submitting_to_email             4.018297e-01
Abnormal_URL                    2.318903e-02
Redirect                        3.210095e-02
on_mouseover                    1.535660e-01
RightClick                      7.881932e-01
popUpWidnow                     9.125916e-01
Iframe                          9.512881e-01
```

```
age_of_domain                    1.365802e-14
DNSRecord                        5.865512e-05
web_traffic                     1.304323e-120
Page_Rank                        1.842981e-17
Google_Index                     2.277588e-05
Links_pointing_to_page           1.196992e-01
Statistical_report               6.585072e-03
dtype: float64
```

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

[ ]:
```
Iframe                           9.512881e-01
popUpWidnow                      9.125916e-01
Favicon                          9.035503e-01
RightClick                       7.881932e-01
Submitting_to_email              4.018297e-01
double_slash_redirecting         1.896375e-01
port                             1.825502e-01
on_mouseover                     1.535660e-01
Links_pointing_to_page           1.196992e-01
HTTPS_token                      1.108615e-01
having_At_Symbol                 3.396638e-02
Redirect                         3.210095e-02
Abnormal_URL                     2.318903e-02
Shortining_Service               1.503035e-02
Statistical_report               6.585072e-03
DNSRecord                        5.865512e-05
Google_Index                     2.277588e-05
having_IPhaving_IP_Address       2.653194e-07
URLURL_Length                    3.923149e-14
age_of_domain                    1.365802e-14
Page_Rank                        1.842981e-17
Request_URL                      2.482240e-55
Domain_registeration_length      3.181261e-70
Links_in_tags                    2.073989e-79
having_Sub_Domain               1.974688e-117
web_traffic                     1.304323e-120
SFH                             2.346646e-136
Prefix_Suffix                   4.656442e-204
URL_of_Anchor                    0.000000e+00
SSLfinal_State                   0.000000e+00
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))
```

[ ]:
```
10
```

[ ]:
```
drop_feature
```

[ ]:
```
{'Favicon',
 'HTTPS_token',
 'Iframe',
 'Links_pointing_to_page',
 'RightClick',
 'Submitting_to_email',
 'double_slash_redirecting',
 'on_mouseover',
 'popUpWidnow',
 'port'}
```

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

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

[ ]:
```
31
```

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

```
Training set has 8844 samples.
Testing set has 2211 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)
0.9263919779124166
```

[ ]:
```
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)
```

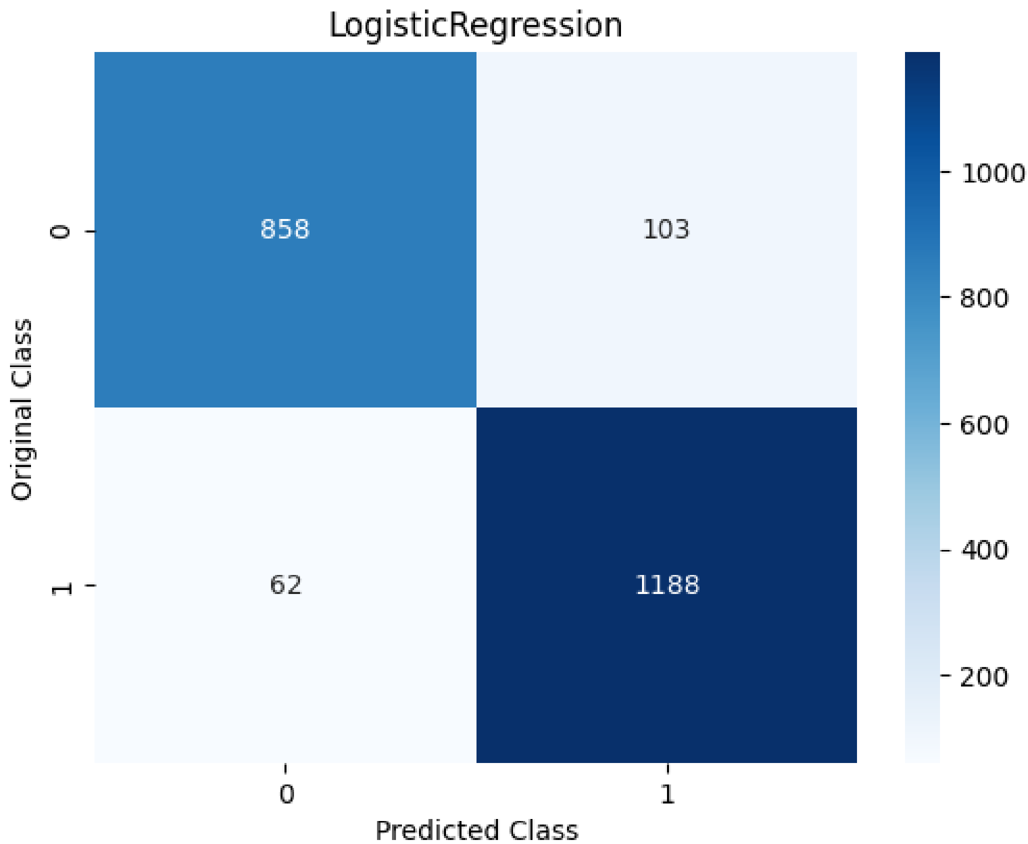
[ ]:
```
print ("Accuracy of logr classifier : ", accuracy_score(y_test,␣
↪logr_predict)*100)
```

```
Accuracy of logr classifier :  92.53731343283582
```

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

```
              precision    recall  f1-score   support

           0       0.93      0.89      0.91       961
           1       0.92      0.95      0.94      1250

    accuracy                           0.93      2211
   macro avg       0.93      0.92      0.92      2211
weighted avg       0.93      0.93      0.93      2211
```

[ ]:
```
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()
```

[ ]:
```python
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': 1}
KNeighborsClassifier(n_neighbors=1)
0.9543199887516935
```

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

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

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

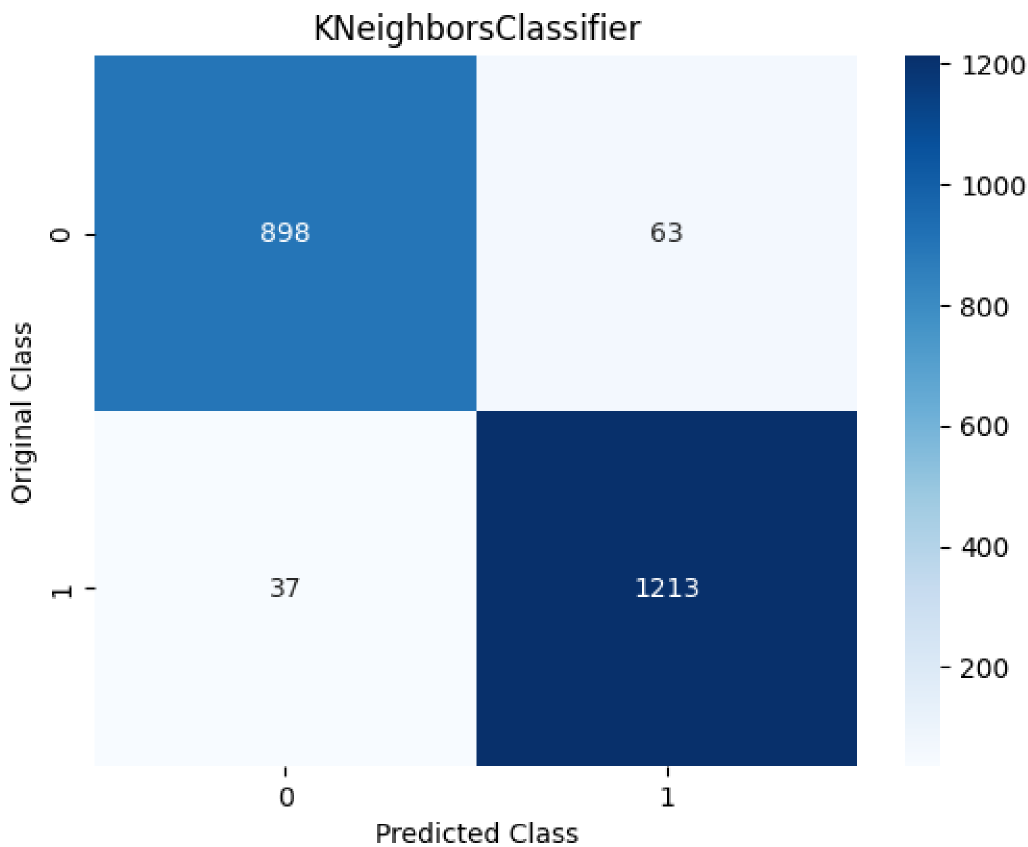
```
The accuracy of knn Classifier is:  95.47715965626413
```

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

```
              precision    recall  f1-score   support

           0       0.96      0.93      0.95       961
           1       0.95      0.97      0.96      1250

    accuracy                           0.95      2211
   macro avg       0.96      0.95      0.95      2211
weighted avg       0.95      0.95      0.95      2211
```

[ ]:
```python
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")
```

[ ]:
```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': 1, 'gamma': 1, 'kernel': 'rbf'}
SVC(C=1, gamma=1)
0.9596344300432037
```

[ ]:
```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)
```

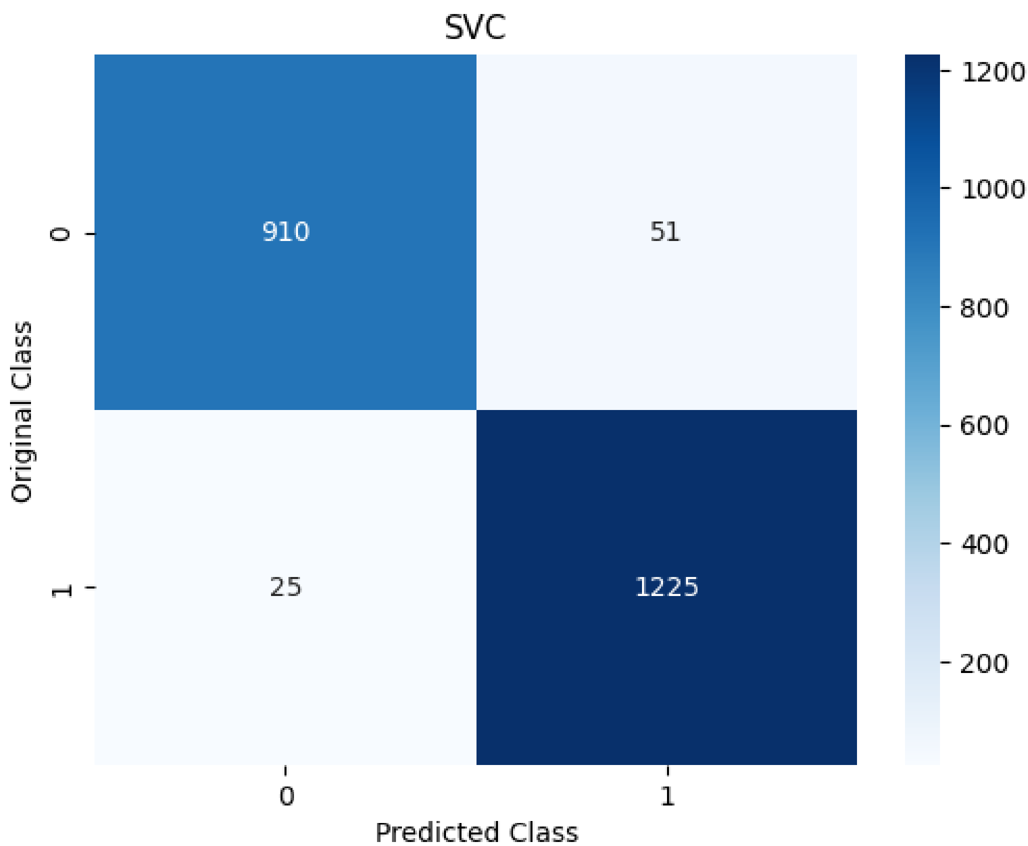
[ ]:
```python
print('The accuracy of svc Classifier is: ', 100.0 * accuracy_score(y_test,␣
 ↪svc_predict))
```

```
The accuracy of svc Classifier is:  96.56264133876074
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       961
           1       0.96      0.98      0.97      1250

    accuracy                           0.97      2211
   macro avg       0.97      0.96      0.96      2211
weighted avg       0.97      0.97      0.97      2211
```

```
[ ]: 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': 1, 'kernel': 'rbf', 'nu': 0.1}
NuSVC(gamma=1, nu=0.1)
0.9594079300559857
```

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

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

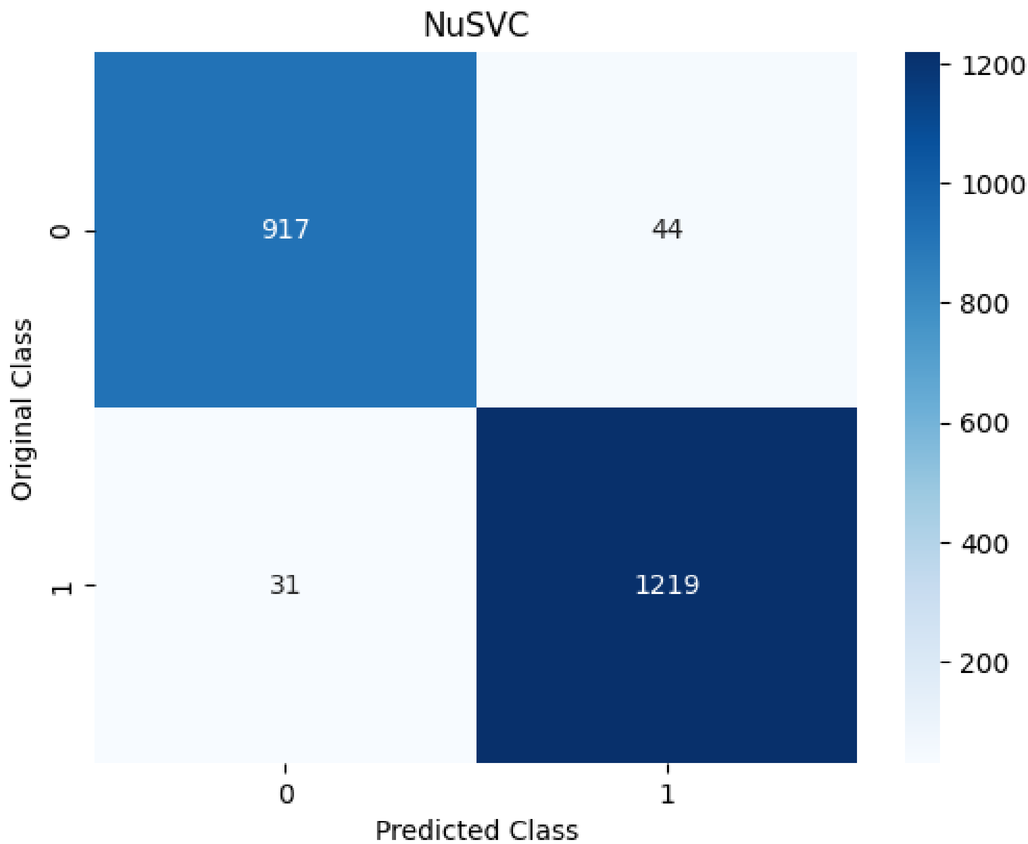
[ ]:
```
print('The accuracy of nusvc Classifier is: ', 100.0 * accuracy_score(y_test,␣
↪nusvc_predict))
```

```
The accuracy of nusvc Classifier is:  96.6078697421981
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       961
           1       0.97      0.98      0.97      1250

    accuracy                           0.97      2211
   macro avg       0.97      0.96      0.97      2211
weighted avg       0.97      0.97      0.97      2211
```

[ ]:
```
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': 10, 'dual': False, 'loss': 'squared_hinge', 'penalty': 'l2', 'tol': 0.01}
LinearSVC(C=10, dual=False, tol=0.01)
0.9256007618171127
```

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

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

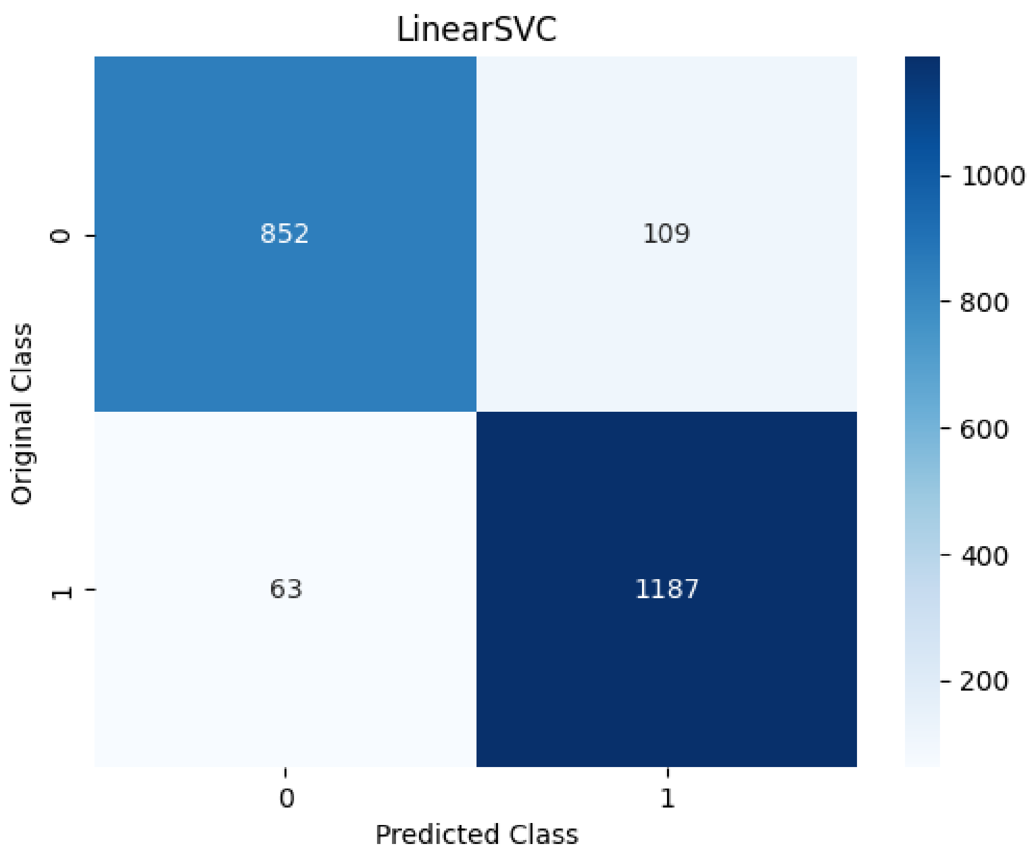
[ ]:
```
print('The accuracy of lsvc Classifier is: ', 100.0 * accuracy_score(y_test,
 ↪lsvc_predict))
```

```
The accuracy of lsvc Classifier is:  92.22071460877432
```

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

```
              precision    recall  f1-score   support

           0       0.93      0.89      0.91       961
           1       0.92      0.95      0.93      1250

    accuracy                           0.92      2211
   macro avg       0.92      0.92      0.92      2211
weighted avg       0.92      0.92      0.92      2211
```

[ ]:
```
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)
0.9352119283176112
```

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

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

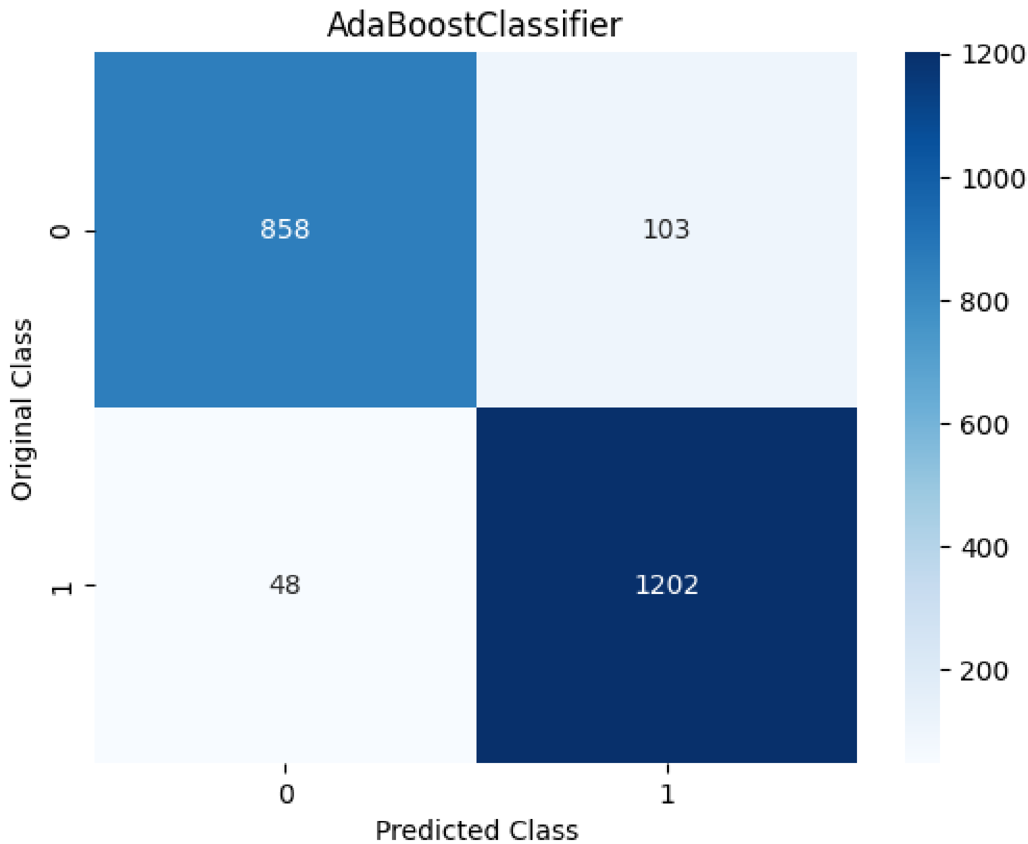
[ ]:
```
print('The accuracy of Ada Boost Classifier is: ', 100.0 *␣
↪accuracy_score(ada_predict,y_test))
```

```
The accuracy of Ada Boost Classifier is:  93.17051108095885
```

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

```
              precision    recall  f1-score   support

           0       0.95      0.89      0.92       961
           1       0.92      0.96      0.94      1250

    accuracy                           0.93      2211
   macro avg       0.93      0.93      0.93      2211
weighted avg       0.93      0.93      0.93      2211
```

[ ]:
```
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': 250}
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=250,
              n_jobs=0, num_parallel_tree=1, predictor='auto', random_state=0,
              reg_alpha=0, reg_lambda=1, …)
0.9642703939463658
```

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

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

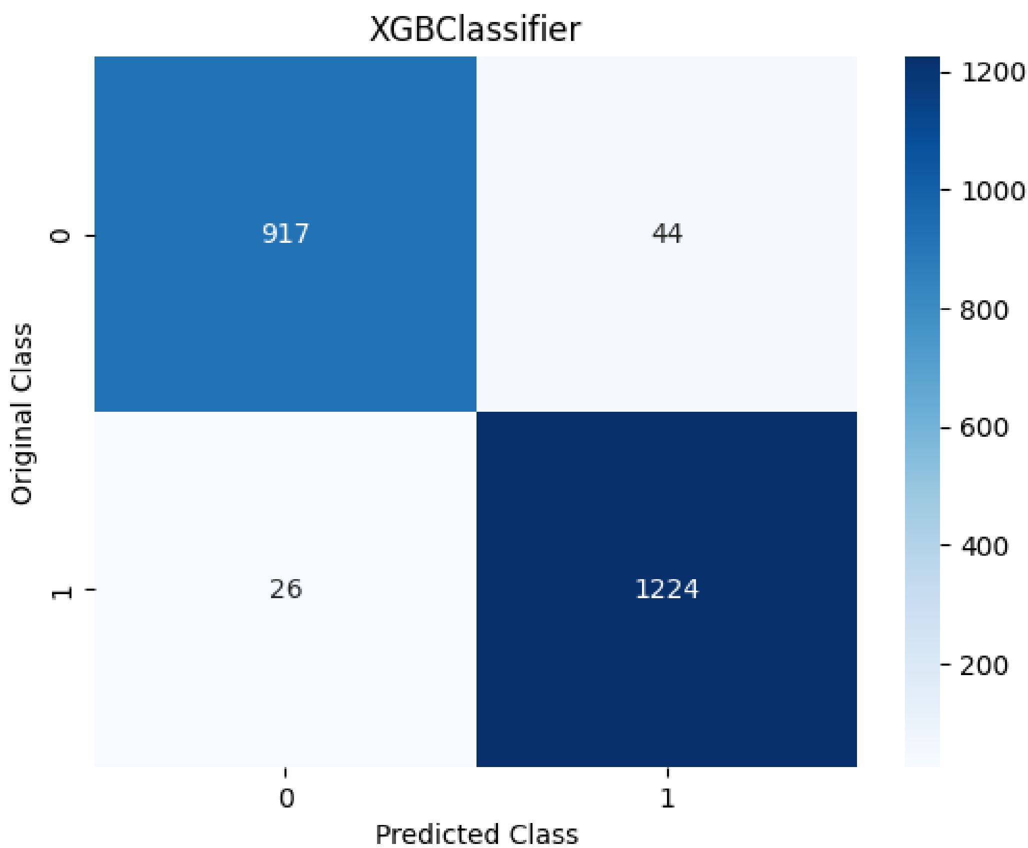
[ ]:
```
print('The accuracy of XGBoost Classifier is: ' , 100.0 *␣
 ↪accuracy_score(xgb_predict,y_test))
```

```
The accuracy of XGBoost Classifier is:  96.8340117593849
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       961
           1       0.97      0.98      0.97      1250

    accuracy                           0.97      2211
   macro avg       0.97      0.97      0.97      2211
weighted avg       0.97      0.97      0.97      2211
```

[ ]:
```
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': 1, 'n_estimators': 250}
GradientBoostingClassifier(learning_rate=1, n_estimators=250)
0.9595213078712579
```

[ ]:
```
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)
```

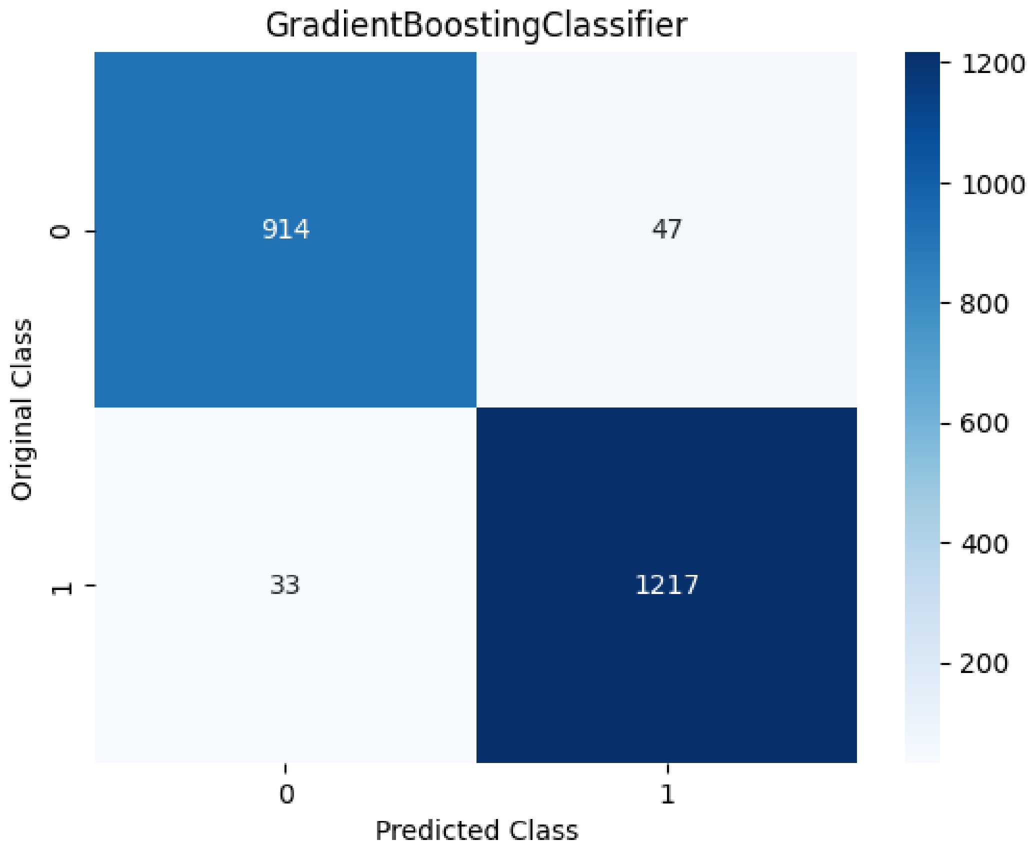
[ ]:
```
print('The accuracy of GradientBoost Classifier is: ' , 100.0 *␣
↪accuracy_score(gbc_predict,y_test))
```

```
The accuracy of GradientBoost Classifier is:  96.38172772501132
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       961
           1       0.96      0.97      0.97      1250

    accuracy                           0.96      2211
   macro avg       0.96      0.96      0.96      2211
weighted avg       0.96      0.96      0.96      2211
```

[ ]:
```
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': 250}
BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=250)
0.9618955952654856
```

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

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

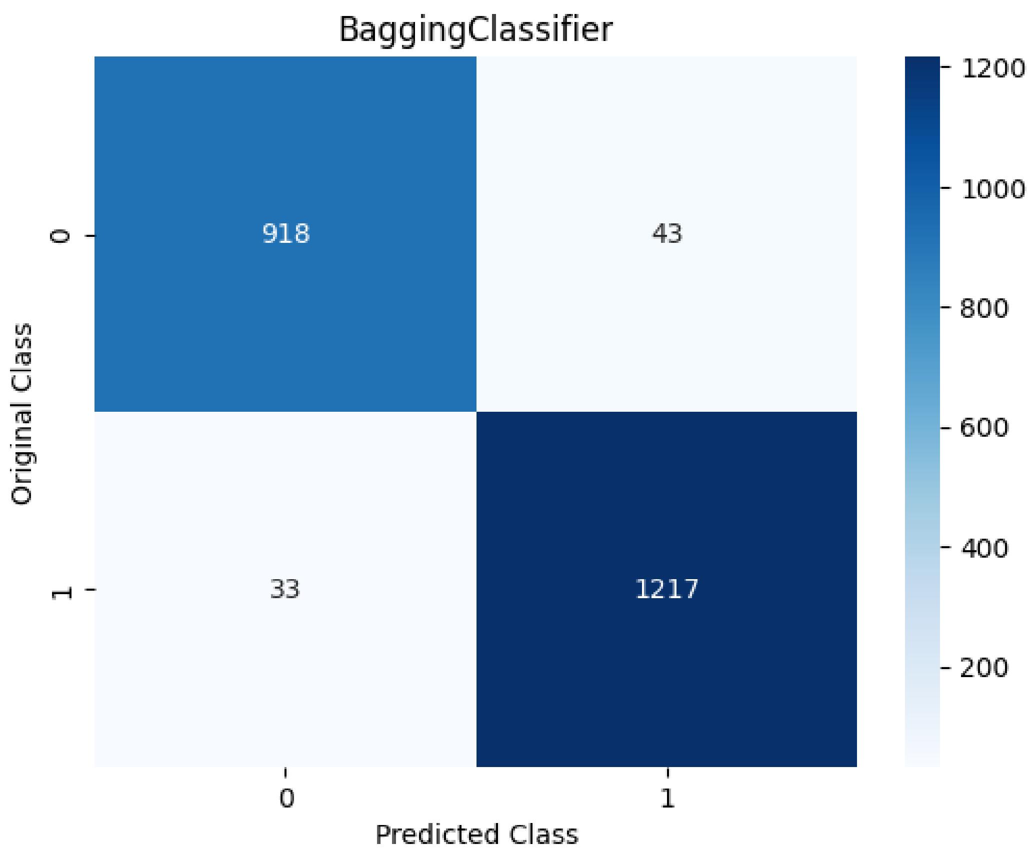
[ ]:
```
print('The accuracy of Bagging Classifier is: ' , 100.0 *␣
↪accuracy_score(y_test, bag_predict))
```

```
The accuracy of Bagging Classifier is:  96.56264133876074
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96       961
           1       0.97      0.97      0.97      1250

    accuracy                           0.97      2211
   macro avg       0.97      0.96      0.96      2211
weighted avg       0.97      0.97      0.97      2211
```

[ ]:
```
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': 250}
RandomForestClassifier(n_estimators=250)
0.9649491269780401
```

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

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

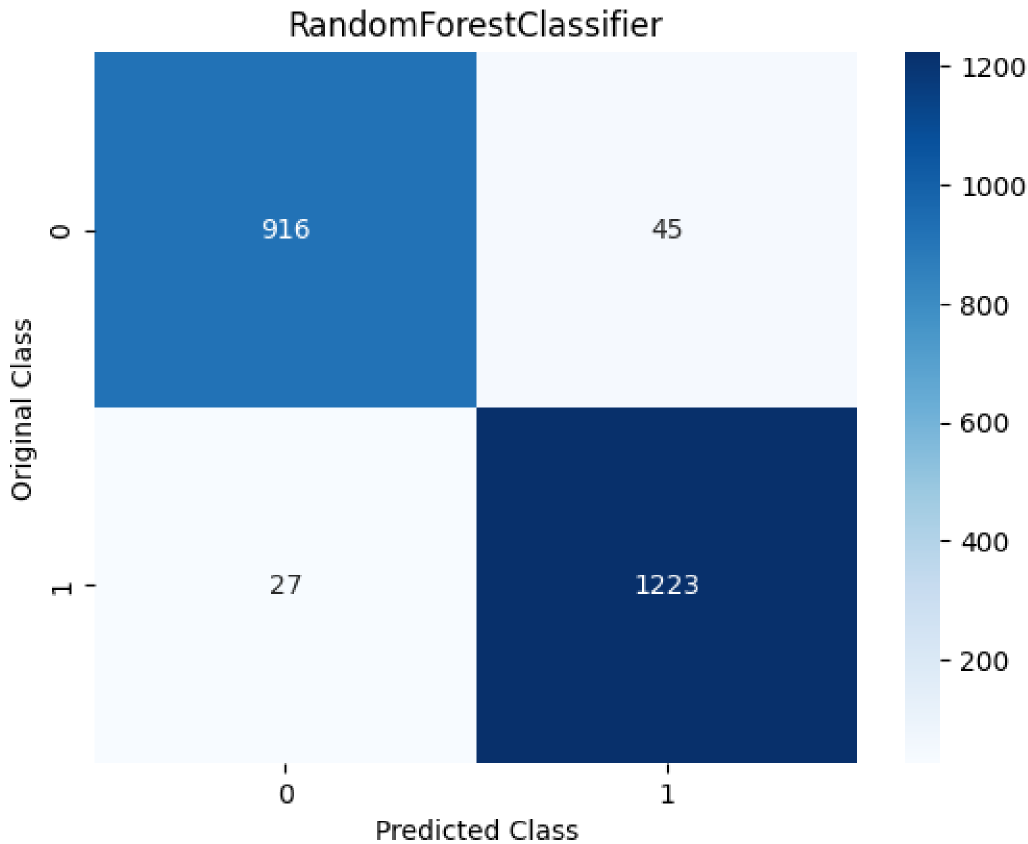
[ ]:
```
print('The accuracy of RandomForest Classifier is: ' , 100.0 *␣
 ↪accuracy_score(rfc_predict,y_test))
```

```
The accuracy of RandomForest Classifier is:  96.74355495251018
```

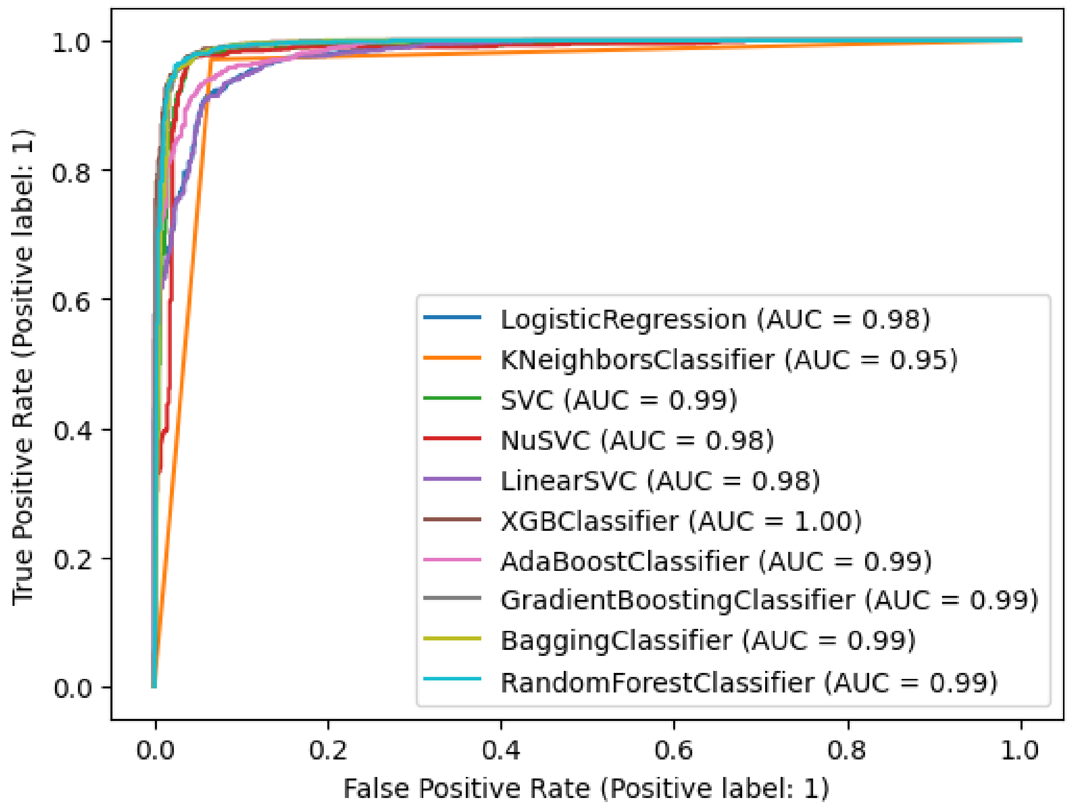
[ ]:
```
print(classification_report(y_test, rfc_predict))
```

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       961
           1       0.96      0.98      0.97      1250

    accuracy                           0.97      2211
   macro avg       0.97      0.97      0.97      2211
weighted avg       0.97      0.97      0.97      2211
```

[ ]:
```
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
```

```
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_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_1 (LSTM)               (None, 100)               48400

 batch_normalization_1 (Batc  (None, 100)              400
 hNormalization)

 dense_4 (Dense)             (None, 50)                5050


_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_1 (LSTM)               (None, 100)               48400

 batch_normalization_1 (Batc  (None, 100)              400
 hNormalization)

 dense_4 (Dense)             (None, 50)                5050

 dense_5 (Dense)             (None, 25)                1275

 dense_6 (Dense)             (None, 10)                260

 dense_7 (Dense)             (None, 1)                 11

=================================================================
Total params: 55,396
Trainable params: 55,196
Non-trainable params: 200
_________________________________________________________________
Epoch 1/100
222/222 [==============================] - 4s 7ms/step - loss: 0.2218 -
accuracy: 0.9122 - val_loss: 0.4066 - val_accuracy: 0.9361
Epoch 2/100
```

```
222/222 [==============================] - 1s 4ms/step - loss: 0.1709 -
accuracy: 0.9300 - val_loss: 0.2155 - val_accuracy: 0.9412
Epoch 3/100
222/222 [==============================] - 1s 5ms/step - loss: 0.1539 -
accuracy: 0.9361 - val_loss: 0.1365 - val_accuracy: 0.9457
Epoch 4/100
222/222 [==============================] - 1s 5ms/step - loss: 0.1482 -
accuracy: 0.9350 - val_loss: 0.1257 - val_accuracy: 0.9508
Epoch 5/100
222/222 [==============================] - 1s 5ms/step - loss: 0.1359 -
accuracy: 0.9433 - val_loss: 0.1259 - val_accuracy: 0.9457
Epoch 6/100
222/222 [==============================] - 1s 5ms/step - loss: 0.1313 -
accuracy: 0.9435 - val_loss: 0.1293 - val_accuracy: 0.9446
Epoch 7/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1208 -
accuracy: 0.9490 - val_loss: 0.1149 - val_accuracy: 0.9548
Epoch 8/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1172 -
accuracy: 0.9504 - val_loss: 0.1166 - val_accuracy: 0.9435
Epoch 9/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1090 -
accuracy: 0.9562 - val_loss: 0.1142 - val_accuracy: 0.9536
Epoch 10/100
222/222 [==============================] - 1s 5ms/step - loss: 0.1081 -
accuracy: 0.9525 - val_loss: 0.1089 - val_accuracy: 0.9548
Epoch 11/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1036 -
accuracy: 0.9549 - val_loss: 0.1271 - val_accuracy: 0.9474
Epoch 12/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1025 -
accuracy: 0.9563 - val_loss: 0.1149 - val_accuracy: 0.9559
Epoch 13/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1111 -
accuracy: 0.9553 - val_loss: 0.1226 - val_accuracy: 0.9542
Epoch 14/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0925 -
accuracy: 0.9582 - val_loss: 0.1187 - val_accuracy: 0.9491
Epoch 15/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0948 -
accuracy: 0.9577 - val_loss: 0.1197 - val_accuracy: 0.9508
Epoch 16/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0847 -
accuracy: 0.9624 - val_loss: 0.1126 - val_accuracy: 0.9548
Epoch 17/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0882 -
accuracy: 0.9637 - val_loss: 0.1087 - val_accuracy: 0.9633
Epoch 18/100
```

```
222/222 [==============================] - 1s 4ms/step - loss: 0.0874 -
accuracy: 0.9610 - val_loss: 0.1235 - val_accuracy: 0.9570
Epoch 19/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0901 -
accuracy: 0.9613 - val_loss: 0.1148 - val_accuracy: 0.9514
Epoch 20/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0860 -
accuracy: 0.9627 - val_loss: 0.1178 - val_accuracy: 0.9525
Epoch 21/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0785 -
accuracy: 0.9651 - val_loss: 0.1207 - val_accuracy: 0.9542
Epoch 22/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0817 -
accuracy: 0.9628 - val_loss: 0.1109 - val_accuracy: 0.9570
Epoch 23/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0758 -
accuracy: 0.9668 - val_loss: 0.1074 - val_accuracy: 0.9553
Epoch 24/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0763 -
accuracy: 0.9657 - val_loss: 0.1091 - val_accuracy: 0.9565
Epoch 25/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0741 -
accuracy: 0.9676 - val_loss: 0.1108 - val_accuracy: 0.9582
Epoch 26/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0776 -
accuracy: 0.9655 - val_loss: 0.1210 - val_accuracy: 0.9520
Epoch 27/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0779 -
accuracy: 0.9654 - val_loss: 0.1092 - val_accuracy: 0.9576
Epoch 28/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0701 -
accuracy: 0.9695 - val_loss: 0.1134 - val_accuracy: 0.9520
Epoch 29/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0720 -
accuracy: 0.9672 - val_loss: 0.1118 - val_accuracy: 0.9599
Epoch 30/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0666 -
accuracy: 0.9696 - val_loss: 0.1142 - val_accuracy: 0.9616
Epoch 31/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0694 -
accuracy: 0.9700 - val_loss: 0.1250 - val_accuracy: 0.9553
Epoch 32/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0678 -
accuracy: 0.9705 - val_loss: 0.1273 - val_accuracy: 0.9514
Epoch 33/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0674 -
accuracy: 0.9705 - val_loss: 0.1299 - val_accuracy: 0.9604
Epoch 34/100
```

```
222/222 [==============================] - 1s 4ms/step - loss: 0.0649 -
accuracy: 0.9703 - val_loss: 0.1229 - val_accuracy: 0.9542
Epoch 35/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0803 -
accuracy: 0.9642 - val_loss: 0.1144 - val_accuracy: 0.9621
Epoch 36/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0656 -
accuracy: 0.9709 - val_loss: 0.1267 - val_accuracy: 0.9559
Epoch 37/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0631 -
accuracy: 0.9737 - val_loss: 0.1219 - val_accuracy: 0.9633
Epoch 38/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0625 -
accuracy: 0.9719 - val_loss: 0.1286 - val_accuracy: 0.9542
Epoch 39/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0635 -
accuracy: 0.9727 - val_loss: 0.1287 - val_accuracy: 0.9587
Epoch 40/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0618 -
accuracy: 0.9730 - val_loss: 0.1234 - val_accuracy: 0.9616
Epoch 41/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0597 -
accuracy: 0.9733 - val_loss: 0.1288 - val_accuracy: 0.9548
Epoch 42/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0615 -
accuracy: 0.9714 - val_loss: 0.1320 - val_accuracy: 0.9604
Epoch 43/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0766 -
accuracy: 0.9657 - val_loss: 0.1201 - val_accuracy: 0.9553
Epoch 44/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0647 -
accuracy: 0.9709 - val_loss: 0.1309 - val_accuracy: 0.9582
Epoch 45/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0618 -
accuracy: 0.9713 - val_loss: 0.1273 - val_accuracy: 0.9565
Epoch 46/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0553 -
accuracy: 0.9747 - val_loss: 0.1435 - val_accuracy: 0.9491
Epoch 47/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0561 -
accuracy: 0.9748 - val_loss: 0.1391 - val_accuracy: 0.9593
Epoch 48/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0560 -
accuracy: 0.9760 - val_loss: 0.1329 - val_accuracy: 0.9565
Epoch 49/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0556 -
accuracy: 0.9757 - val_loss: 0.1235 - val_accuracy: 0.9582
Epoch 50/100
```

```
222/222 [==============================] - 1s 4ms/step - loss: 0.0551 -
accuracy: 0.9747 - val_loss: 0.1369 - val_accuracy: 0.9548
Epoch 51/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0558 -
accuracy: 0.9761 - val_loss: 0.1199 - val_accuracy: 0.9576
Epoch 52/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0508 -
accuracy: 0.9770 - val_loss: 0.1392 - val_accuracy: 0.9604
Epoch 53/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0543 -
accuracy: 0.9763 - val_loss: 0.1217 - val_accuracy: 0.9638
Epoch 54/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0592 -
accuracy: 0.9755 - val_loss: 0.1453 - val_accuracy: 0.9520
Epoch 55/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0518 -
accuracy: 0.9755 - val_loss: 0.1679 - val_accuracy: 0.9373
Epoch 56/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0542 -
accuracy: 0.9753 - val_loss: 0.1510 - val_accuracy: 0.9587
Epoch 57/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0621 -
accuracy: 0.9731 - val_loss: 0.1277 - val_accuracy: 0.9587
Epoch 58/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0526 -
accuracy: 0.9764 - val_loss: 0.1262 - val_accuracy: 0.9593
Epoch 59/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0521 -
accuracy: 0.9770 - val_loss: 0.1396 - val_accuracy: 0.9570
Epoch 60/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0530 -
accuracy: 0.9755 - val_loss: 0.1351 - val_accuracy: 0.9604
Epoch 61/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0503 -
accuracy: 0.9760 - val_loss: 0.1362 - val_accuracy: 0.9582
Epoch 62/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0523 -
accuracy: 0.9764 - val_loss: 0.1494 - val_accuracy: 0.9644
Epoch 63/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0470 -
accuracy: 0.9768 - val_loss: 0.1661 - val_accuracy: 0.9531
Epoch 64/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0473 -
accuracy: 0.9778 - val_loss: 0.1439 - val_accuracy: 0.9559
Epoch 65/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0600 -
accuracy: 0.9743 - val_loss: 0.1463 - val_accuracy: 0.9576
Epoch 66/100
```

```
222/222 [==============================] - 1s 4ms/step - loss: 0.0474 -
accuracy: 0.9772 - val_loss: 0.1519 - val_accuracy: 0.9570
Epoch 67/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0513 -
accuracy: 0.9754 - val_loss: 0.1301 - val_accuracy: 0.9644
Epoch 68/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0475 -
accuracy: 0.9794 - val_loss: 0.1460 - val_accuracy: 0.9627
Epoch 69/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0634 -
accuracy: 0.9726 - val_loss: 0.1459 - val_accuracy: 0.9627
Epoch 70/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0516 -
accuracy: 0.9758 - val_loss: 0.1443 - val_accuracy: 0.9542
Epoch 71/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0515 -
accuracy: 0.9771 - val_loss: 0.1387 - val_accuracy: 0.9593
Epoch 72/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0462 -
accuracy: 0.9772 - val_loss: 0.1343 - val_accuracy: 0.9604
Epoch 73/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0449 -
accuracy: 0.9789 - val_loss: 0.1574 - val_accuracy: 0.9621
Epoch 74/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0449 -
accuracy: 0.9775 - val_loss: 0.1413 - val_accuracy: 0.9576
Epoch 75/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0452 -
accuracy: 0.9789 - val_loss: 0.1421 - val_accuracy: 0.9644
Epoch 76/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0512 -
accuracy: 0.9758 - val_loss: 0.1476 - val_accuracy: 0.9621
Epoch 77/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0446 -
accuracy: 0.9794 - val_loss: 0.1603 - val_accuracy: 0.9587
Epoch 78/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0520 -
accuracy: 0.9744 - val_loss: 0.1360 - val_accuracy: 0.9644
Epoch 79/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0455 -
accuracy: 0.9774 - val_loss: 0.1647 - val_accuracy: 0.9525
Epoch 80/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0479 -
accuracy: 0.9764 - val_loss: 0.1319 - val_accuracy: 0.9627
Epoch 81/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0431 -
accuracy: 0.9791 - val_loss: 0.1436 - val_accuracy: 0.9638
Epoch 82/100
```

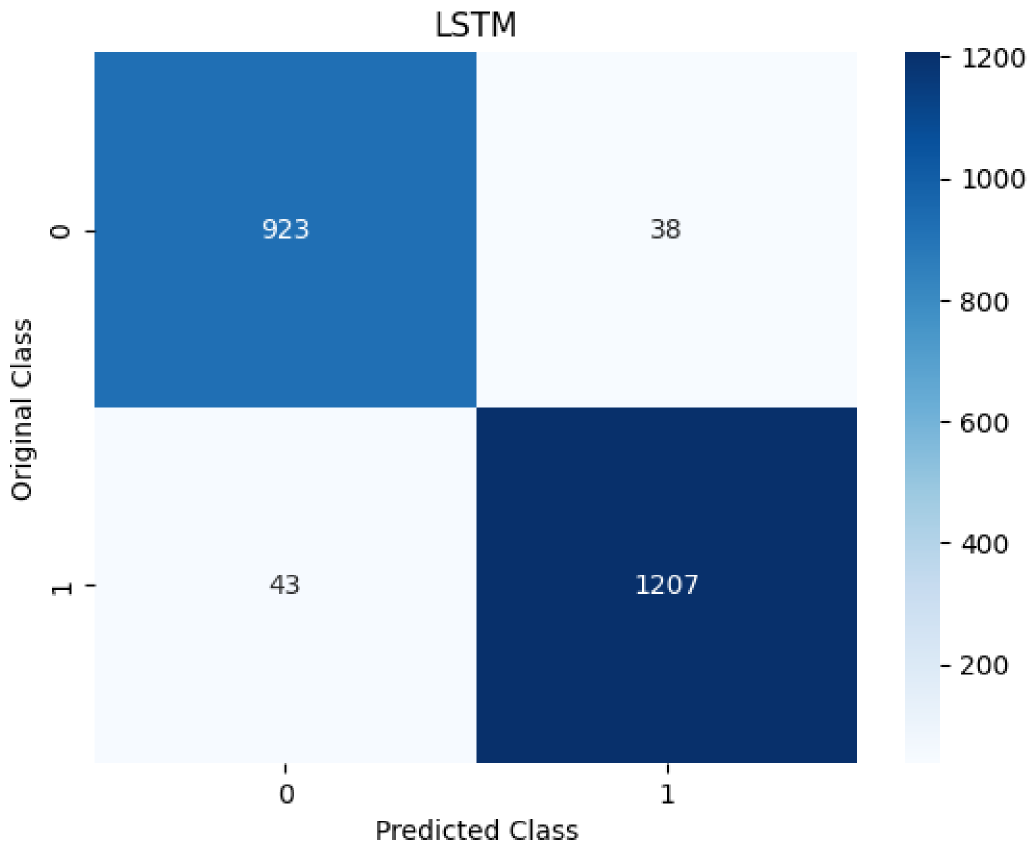
```
222/222 [==============================] - 1s 5ms/step - loss: 0.0429 -
accuracy: 0.9787 - val_loss: 0.1488 - val_accuracy: 0.9610
Epoch 83/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0443 -
accuracy: 0.9789 - val_loss: 0.1496 - val_accuracy: 0.9655
Epoch 84/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0500 -
accuracy: 0.9768 - val_loss: 0.1400 - val_accuracy: 0.9650
Epoch 85/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0454 -
accuracy: 0.9805 - val_loss: 0.1358 - val_accuracy: 0.9621
Epoch 86/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0620 -
accuracy: 0.9755 - val_loss: 0.1458 - val_accuracy: 0.9599
Epoch 87/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0560 -
accuracy: 0.9767 - val_loss: 0.1300 - val_accuracy: 0.9599
Epoch 88/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0441 -
accuracy: 0.9796 - val_loss: 0.1548 - val_accuracy: 0.9565
Epoch 89/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0417 -
accuracy: 0.9801 - val_loss: 0.1537 - val_accuracy: 0.9593
Epoch 90/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0428 -
accuracy: 0.9796 - val_loss: 0.1495 - val_accuracy: 0.9576
Epoch 91/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0434 -
accuracy: 0.9788 - val_loss: 0.1456 - val_accuracy: 0.9576
Epoch 92/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0576 -
accuracy: 0.9760 - val_loss: 0.1433 - val_accuracy: 0.9604
Epoch 93/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0459 -
accuracy: 0.9794 - val_loss: 0.1371 - val_accuracy: 0.9621
Epoch 94/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0408 -
accuracy: 0.9812 - val_loss: 0.1584 - val_accuracy: 0.9610
Epoch 95/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0564 -
accuracy: 0.9753 - val_loss: 0.1301 - val_accuracy: 0.9621
Epoch 96/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0561 -
accuracy: 0.9754 - val_loss: 0.1418 - val_accuracy: 0.9593
Epoch 97/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0421 -
accuracy: 0.9802 - val_loss: 0.1461 - val_accuracy: 0.9650
Epoch 98/100
```

```
222/222 [==============================] - 1s 4ms/step - loss: 0.0411 -
accuracy: 0.9796 - val_loss: 0.1630 - val_accuracy: 0.9559
Epoch 99/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0419 -
accuracy: 0.9801 - val_loss: 0.1494 - val_accuracy: 0.9627
Epoch 100/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0428 -
accuracy: 0.9791 - val_loss: 0.1512 - val_accuracy: 0.9587
Test results - Loss: 0.12548722326755524 - Accuracy: 96.33650183677673%
```

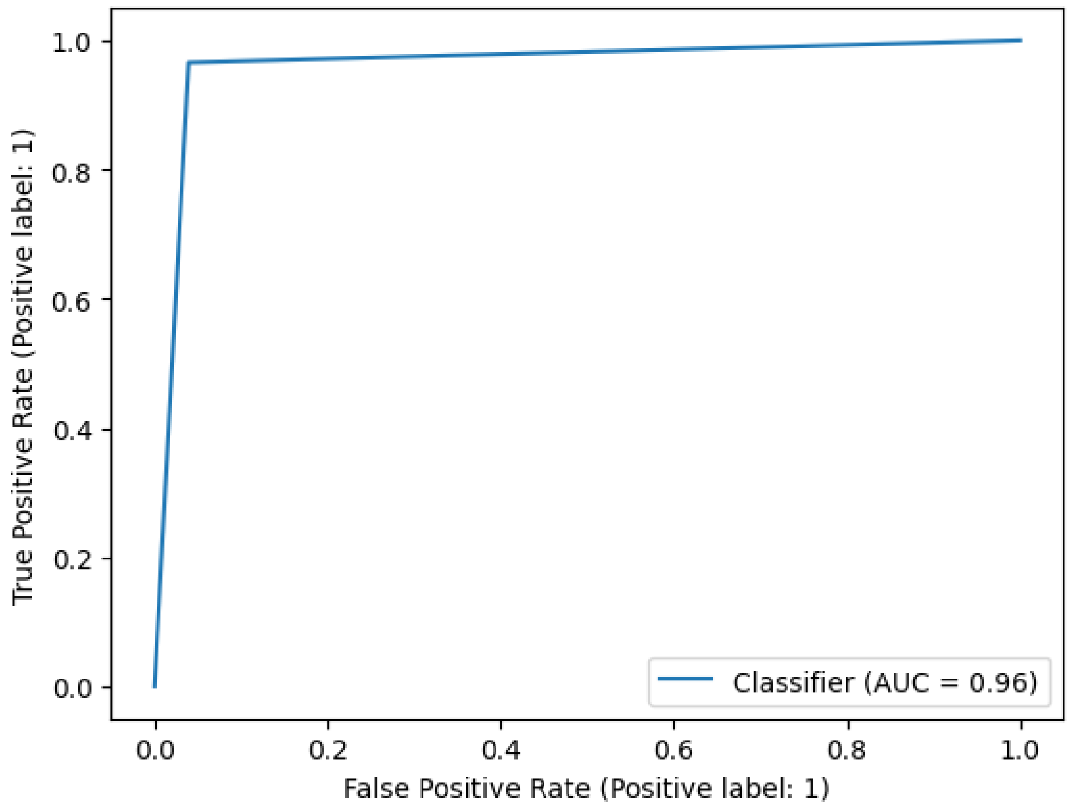
[ ]:
```python
lstm_predict_proba = lstm_model.predict(X_test_reshape, batch_size=32)
lstm_predict_class = (lstm_predict_proba > 0.5).astype("int32")
print(classification_report(y_test, lstm_predict_class))
```

```
70/70 [==============================] - 1s 2ms/step
              precision    recall  f1-score   support

           0       0.96      0.96      0.96       961
           1       0.97      0.97      0.97      1250

    accuracy                           0.96      2211
   macro avg       0.96      0.96      0.96      2211
weighted avg       0.96      0.96      0.96      2211
```

[ ]:
```python
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)
Classifier (AUC = 0.96)
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate (Positive label: 1)