# chi_sq_30 9010 split .05 threshold

January 2, 2023

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

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

[ ]:
```
# Importing the dataset
df = pd.read_csv('dataset_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=0.1,
    random_state=10)

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

[ ]:
```
((9949, 30), (1106, 30), (9949, 1), (1106, 1))
```

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

[ ]:
```
f_p_values
```

[ ]:
```
(array([2.95454443e+01, 5.69285338e+01, 6.48054923e+00, 5.19005883e+00,
        2.24367695e+00, 1.05380952e+03, 5.83898807e+02, 3.39307396e+03,
        3.43323680e+02, 7.10402361e-03, 2.19169758e+00, 3.28375268e+00,
        2.57362852e+02, 2.67851850e+03, 4.16627163e+02, 6.85589281e+02,
        9.97960542e-01, 6.39645595e+00, 2.83434016e+00, 2.37822783e+00,
        9.20405451e-02, 6.23476770e-03, 1.16281643e-03, 6.87102345e+01,
        1.79259515e+01, 6.19393719e+02, 8.66397569e+01, 2.26984316e+01,
        2.81600035e+00, 8.30737109e+00]),
 array([5.46208850e-008, 4.51940623e-014, 1.09061279e-002, 2.27164473e-002,
        1.34161615e-001, 3.61673986e-231, 5.32127479e-129, 0.00000000e+000,
        1.20515364e-076, 9.32829544e-001, 1.38756314e-001, 6.99687740e-002,
        6.44707289e-058, 0.00000000e+000, 1.32309547e-092, 4.06870428e-151,
        3.17804501e-001, 1.14348410e-002, 9.22686941e-002, 1.23037052e-001,
        7.61598878e-001, 9.37064005e-001, 9.72797333e-001, 1.14047515e-016,
        2.29668002e-005, 1.01302745e-136, 1.30203200e-020, 1.89522496e-006,
        9.33286991e-002, 3.94845012e-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      5.462088e-08
URLURL_Length                   4.519406e-14
Shortining_Service              1.090613e-02
having_At_Symbol                2.271645e-02
double_slash_redirecting        1.341616e-01
Prefix_Suffix                  3.616740e-231
having_Sub_Domain              5.321275e-129
SSLfinal_State                  0.000000e+00
Domain_registeration_length     1.205154e-76
Favicon                         9.328295e-01
port                            1.387563e-01
HTTPS_token                     6.996877e-02
Request_URL                     6.447073e-58
URL_of_Anchor                   0.000000e+00
Links_in_tags                   1.323095e-92
SFH                            4.068704e-151
Submitting_to_email             3.178045e-01
Abnormal_URL                    1.143484e-02
Redirect                        9.226869e-02
on_mouseover                    1.230371e-01
RightClick                      7.615989e-01
popUpWidnow                     9.370640e-01
Iframe                          9.727973e-01
```

```
age_of_domain                1.140475e-16
DNSRecord                    2.296680e-05
web_traffic                 1.013027e-136
Page_Rank                    1.302032e-20
Google_Index                 1.895225e-06
Links_pointing_to_page       9.332870e-02
Statistical_report           3.948450e-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                        9.727973e-01
popUpWidnow                   9.370640e-01
Favicon                       9.328295e-01
RightClick                    7.615989e-01
Submitting_to_email           3.178045e-01
port                          1.387563e-01
double_slash_redirecting      1.341616e-01
on_mouseover                  1.230371e-01
Links_pointing_to_page        9.332870e-02
Redirect                      9.226869e-02
HTTPS_token                   6.996877e-02
having_At_Symbol              2.271645e-02
Abnormal_URL                  1.143484e-02
Shortining_Service            1.090613e-02
Statistical_report            3.948450e-03
DNSRecord                     2.296680e-05
Google_Index                  1.895225e-06
having_IPhaving_IP_Address    5.462088e-08
URLURL_Length                 4.519406e-14
age_of_domain                 1.140475e-16
Page_Rank                     1.302032e-20
Request_URL                   6.447073e-58
Domain_registeration_length   1.205154e-76
Links_in_tags                 1.323095e-92
having_Sub_Domain            5.321275e-129
web_traffic                  1.013027e-136
SFH                          4.068704e-151
Prefix_Suffix                3.616740e-231
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))
```

[ ]: 11

[ ]:
```
drop_feature
```

[ ]:
```
{'Favicon',
 'HTTPS_token',
 'Iframe',
 'Links_pointing_to_page',
 'Redirect',
 '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 9949 samples.
Testing set has 1106 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': 10, 'max_iter': 2500, 'penalty': 'l2', 'solver': 'lbfgs'}
LogisticRegression(C=10, max_iter=2500)
0.9239128236757226
```

[ ]:
```
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)
```

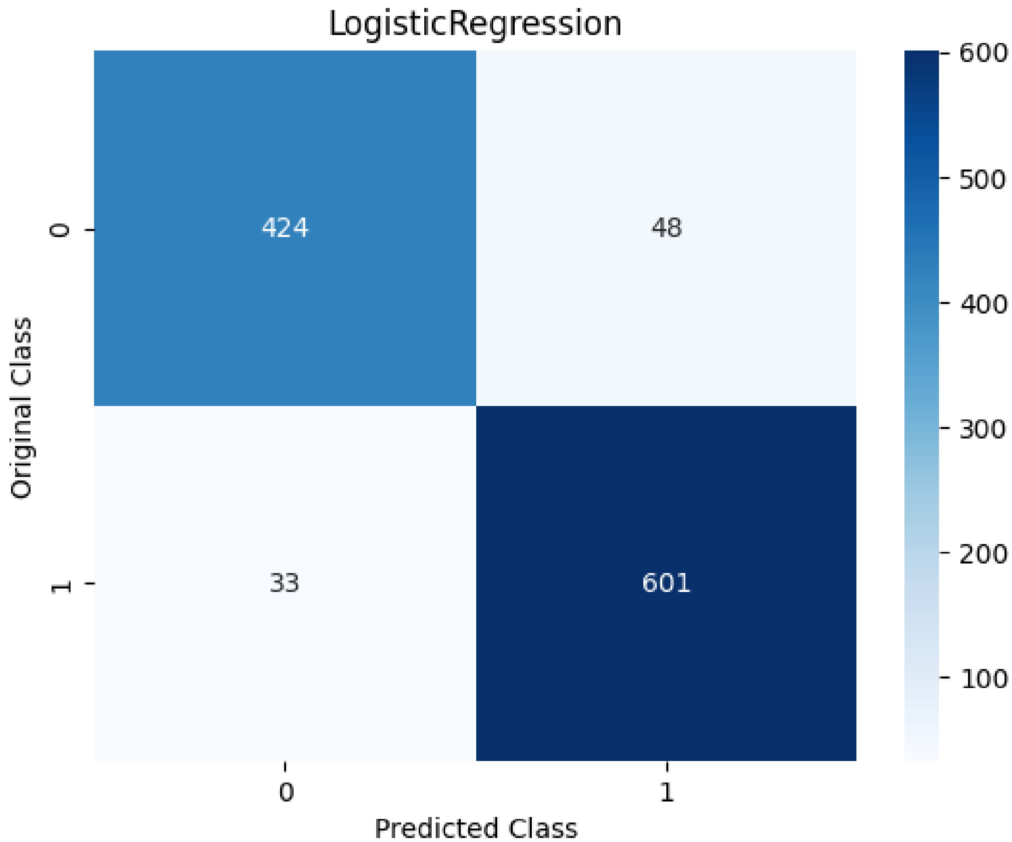
```
Accuracy of logr classifier :  92.6763110307414
```

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

```
              precision    recall  f1-score   support

           0       0.93      0.90      0.91       472
           1       0.93      0.95      0.94       634

    accuracy                           0.93      1106
   macro avg       0.93      0.92      0.92      1106
weighted avg       0.93      0.93      0.93      1106
```

[ ]:
```
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': 1}
KNeighborsClassifier(n_neighbors=1)
0.9551719361394498
```

```
[ ]: 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)
```

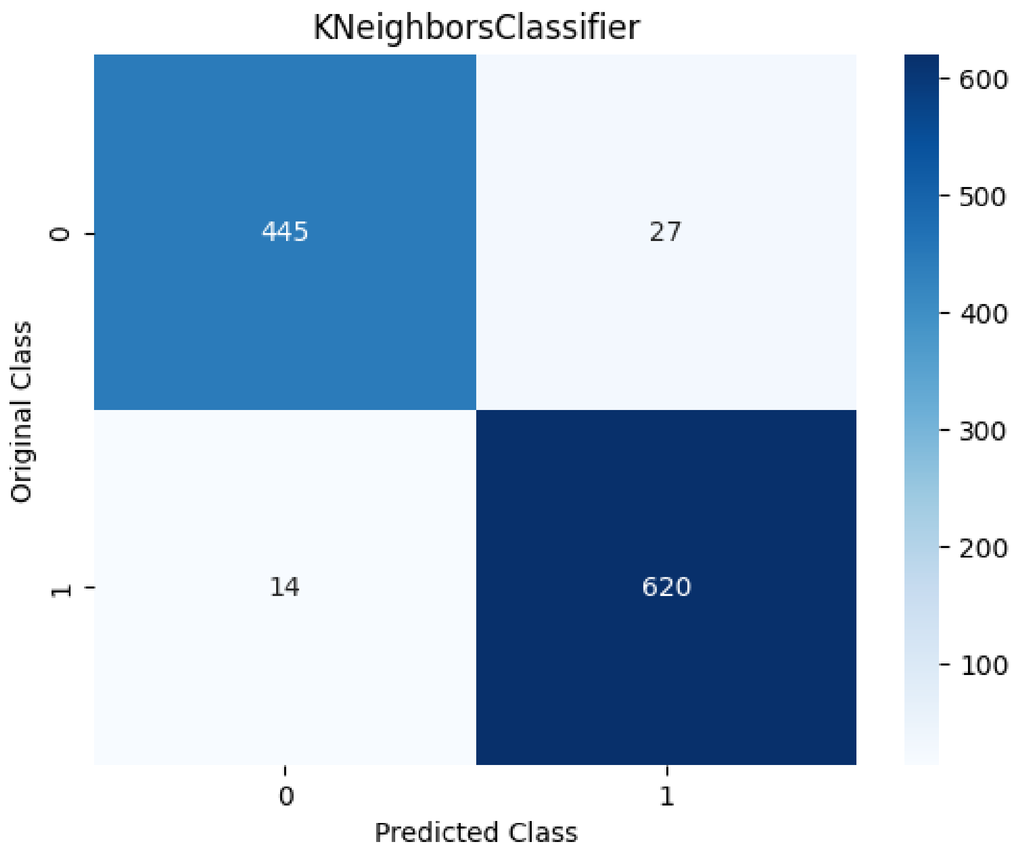
```
[ ]: print('The accuracy of knn Classifier is: ', 100.0 * accuracy_score(y_test,␣
 ↪knn_predict))
```

```
The accuracy of knn Classifier is:  96.29294755877035
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.94      0.96       472
           1       0.96      0.98      0.97       634

    accuracy                           0.96      1106
   macro avg       0.96      0.96      0.96      1106
weighted avg       0.96      0.96      0.96      1106
```

[ ]:
```
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': 1, 'gamma': 1, 'kernel': 'rbf'}
SVC(C=1, gamma=1)
0.9617048016743677
```

[ ]:
```
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))
```

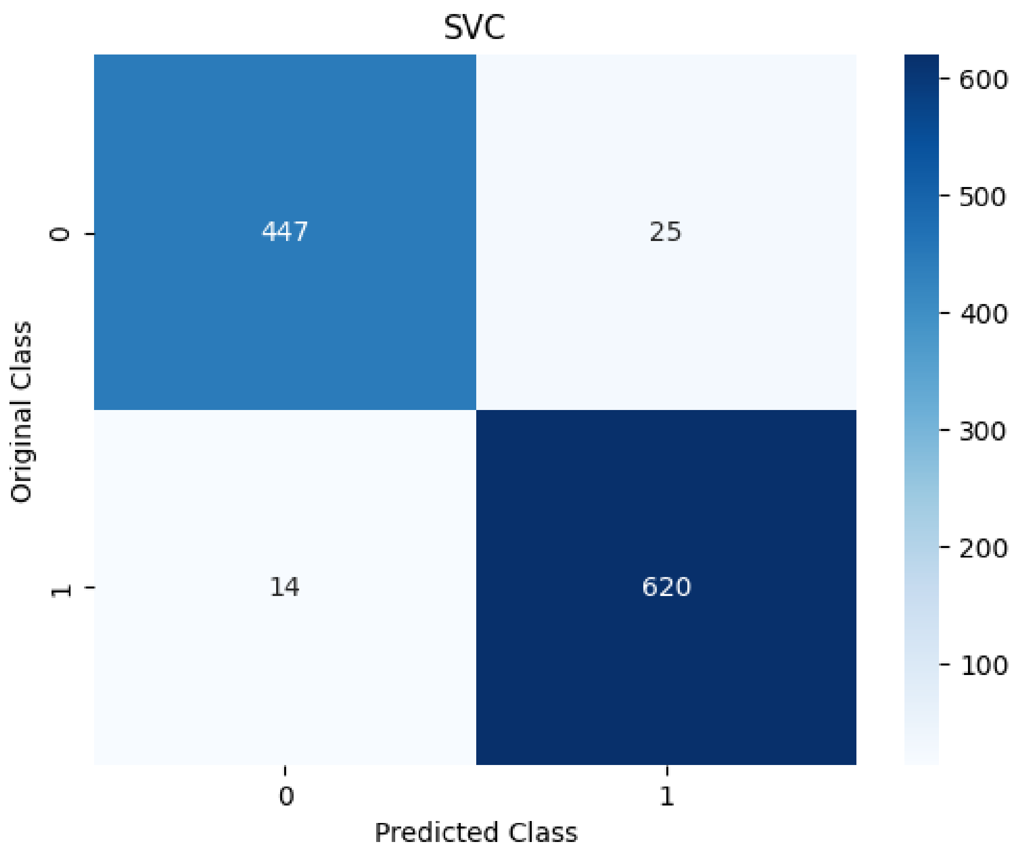
```
The accuracy of svc Classifier is:  96.47377938517178
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       472
           1       0.96      0.98      0.97       634

    accuracy                           0.96      1106
   macro avg       0.97      0.96      0.96      1106
weighted avg       0.96      0.96      0.96      1106
```

[ ]:

```
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)
0.9616039958343021
```

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

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

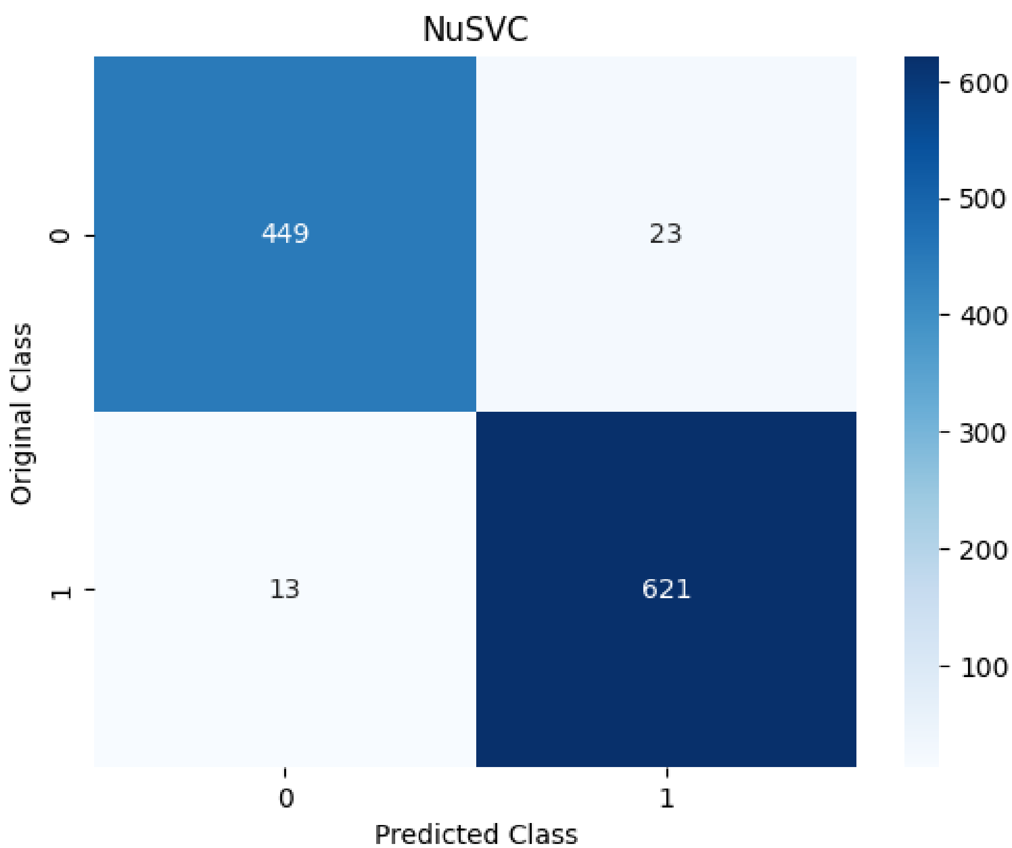
[ ]:
```
print('The accuracy of nusvc Classifier is: ', 100.0 * accuracy_score(y_test,␣
↪nusvc_predict))
```

```
The accuracy of nusvc Classifier is:  96.74502712477397
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       472
           1       0.96      0.98      0.97       634

    accuracy                           0.97      1106
   macro avg       0.97      0.97      0.97      1106
weighted avg       0.97      0.97      0.97      1106
```

[ ]:
```
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': 1, 'dual': False, 'loss': 'squared_hinge', 'penalty': 'l2', 'tol': 0.01}
LinearSVC(C=1, dual=False, tol=0.01)
0.924616644591165
```

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

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

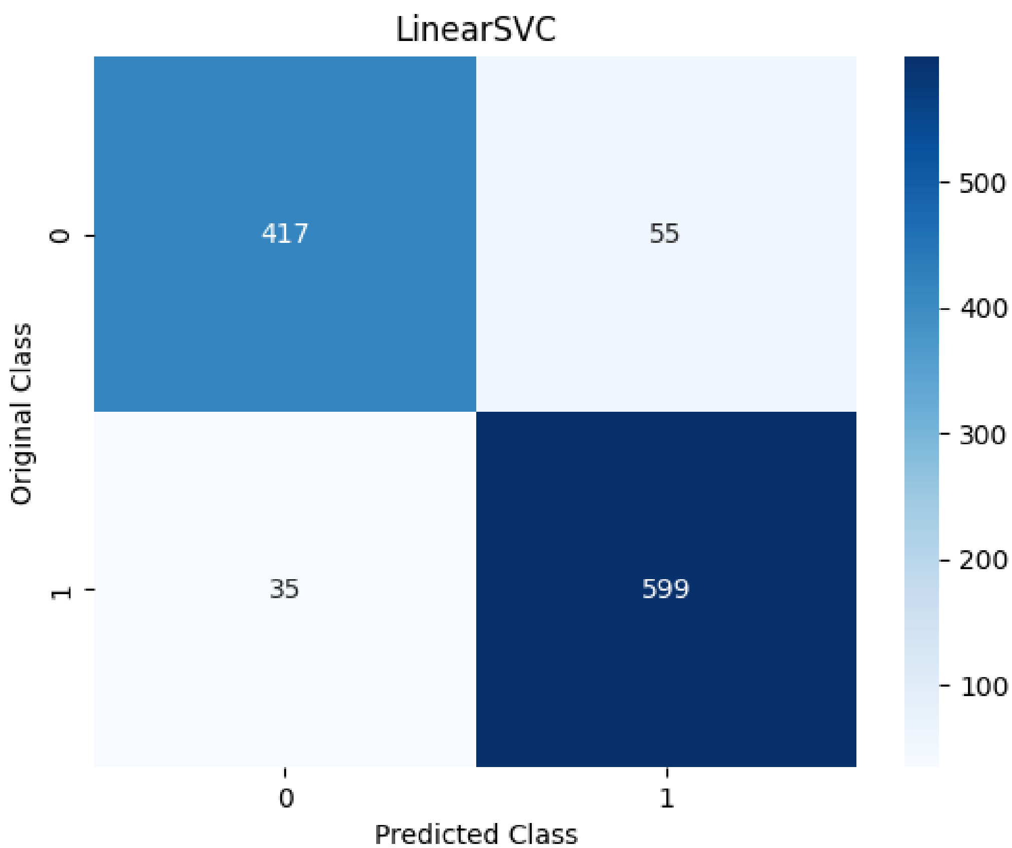
[ ]:
```
print('The accuracy of lsvc Classifier is: ', 100.0 * accuracy_score(y_test,␣
↪lsvc_predict))
```

```
The accuracy of lsvc Classifier is:  91.86256781193491
```

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

```
              precision    recall  f1-score   support

           0       0.92      0.88      0.90       472
           1       0.92      0.94      0.93       634

    accuracy                           0.92      1106
   macro avg       0.92      0.91      0.92      1106
weighted avg       0.92      0.92      0.92      1106
```

[ ]:
```
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': 100}
```

```
AdaBoostClassifier(n_estimators=100)
0.9330590578647767
```

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

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

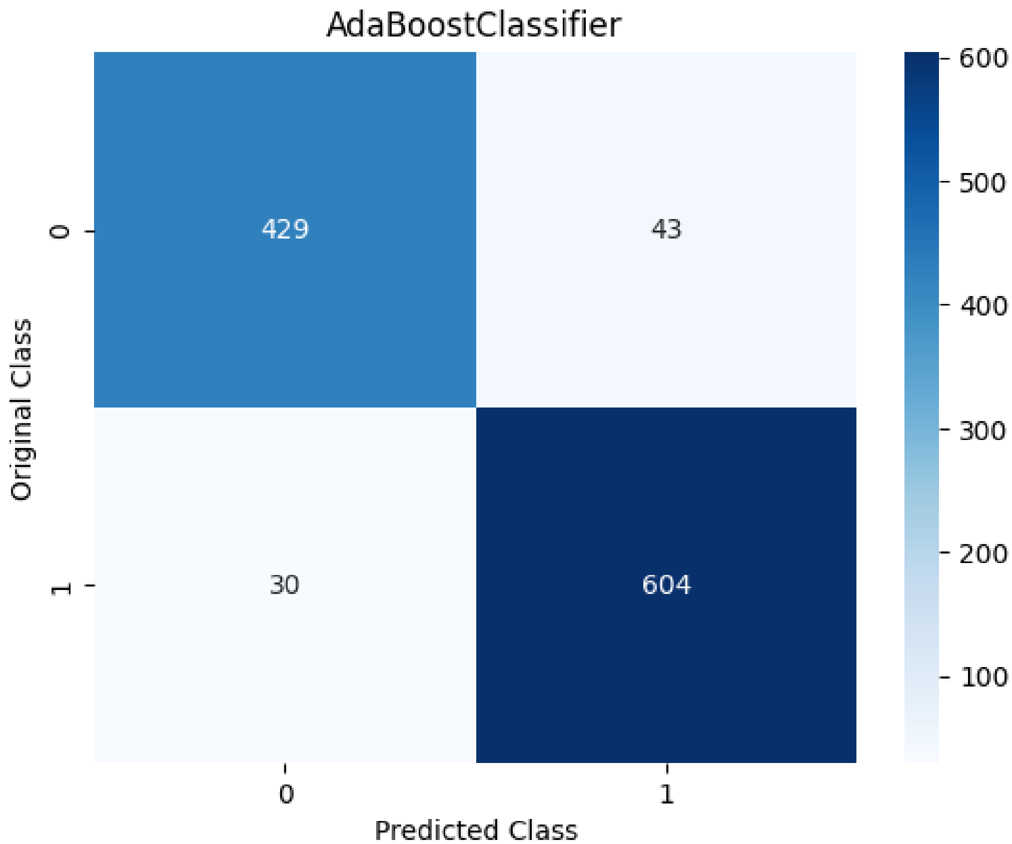
[ ]:
```
print('The accuracy of Ada Boost Classifier is: ', 100.0 *␣
  ↪accuracy_score(ada_predict,y_test))
```

```
The accuracy of Ada Boost Classifier is:  93.3996383363472
```

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

```
              precision    recall  f1-score   support

           0       0.93      0.91      0.92       472
           1       0.93      0.95      0.94       634

    accuracy                           0.93      1106
   macro avg       0.93      0.93      0.93      1106
weighted avg       0.93      0.93      0.93      1106
```

[ ]:
```
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, …)
0.9642167578334326
```

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

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

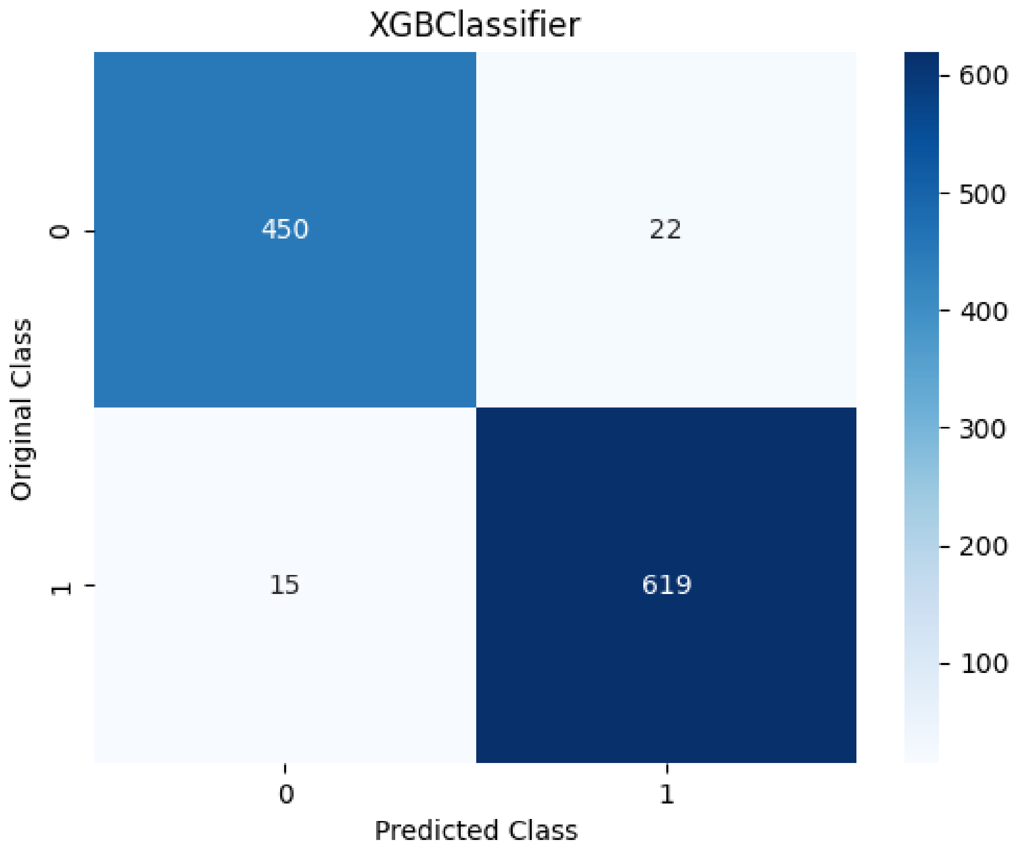
[ ]:
```
print('The accuracy of XGBoost Classifier is: ' , 100.0 *␣
↪accuracy_score(xgb_predict,y_test))
```

```
The accuracy of XGBoost Classifier is:  96.65461121157324
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       472
           1       0.97      0.98      0.97       634

    accuracy                           0.97      1106
   macro avg       0.97      0.96      0.97      1106
weighted avg       0.97      0.97      0.97      1106
```

[ ]:
```
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)
0.9622069098005117
```

```
[ ]: 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)
```

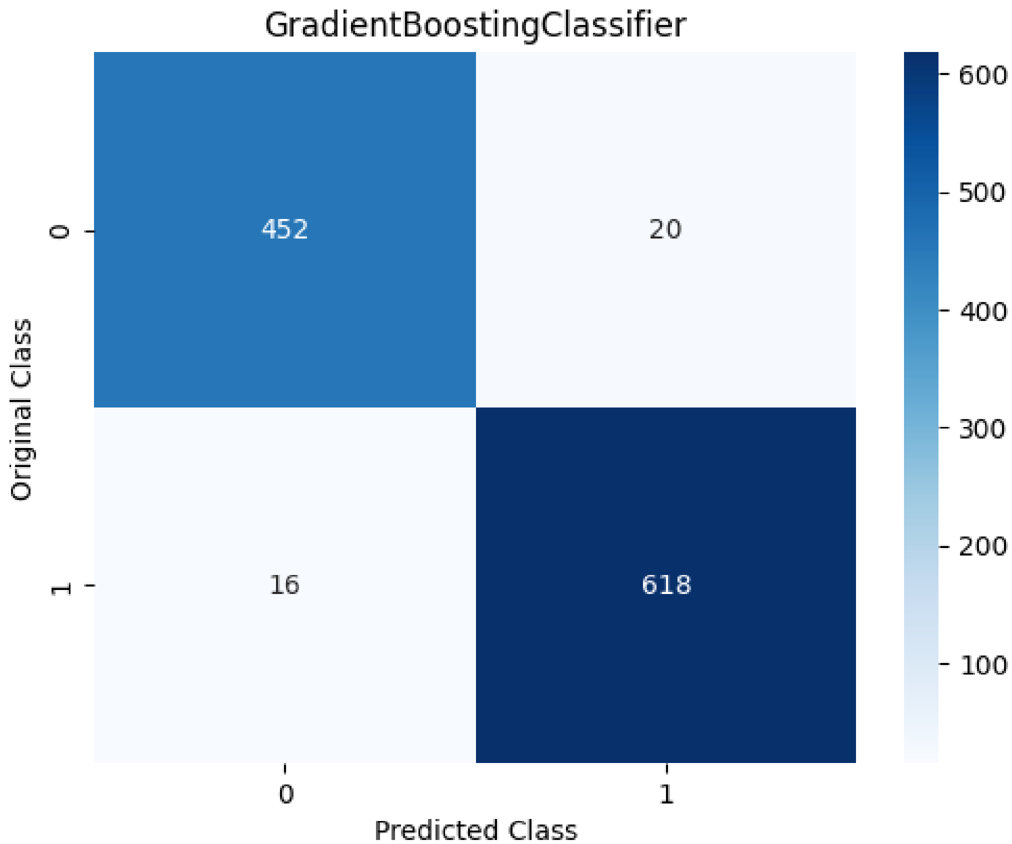
```
[ ]: print('The accuracy of GradientBoost Classifier is: ' , 100.0 *␣
     ↪accuracy_score(gbc_predict,y_test))
```

```
The accuracy of GradientBoost Classifier is:  96.74502712477397
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96       472
           1       0.97      0.97      0.97       634

    accuracy                           0.97      1106
   macro avg       0.97      0.97      0.97      1106
weighted avg       0.97      0.97      0.97      1106
```

```
[ ]: 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)
0.9625088217748703
```

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

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

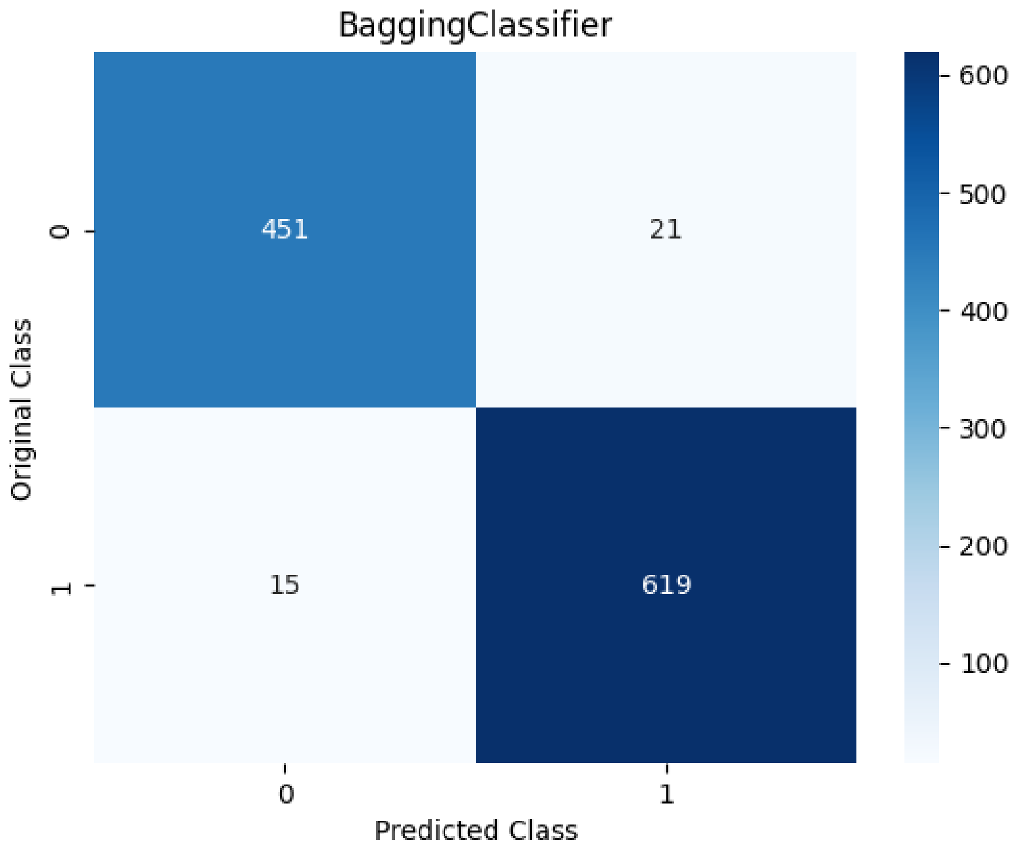
[ ]:
```
print('The accuracy of Bagging Classifier is: ' , 100.0 *
 ↪accuracy_score(y_test, bag_predict))
```

```
The accuracy of Bagging Classifier is:  96.74502712477397
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96       472
           1       0.97      0.98      0.97       634

    accuracy                           0.97      1106
   macro avg       0.97      0.97      0.97      1106
weighted avg       0.97      0.97      0.97      1106
```

[ ]:
```
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': 50}
RandomForestClassifier(n_estimators=50)
0.9643175636734982
```

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

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

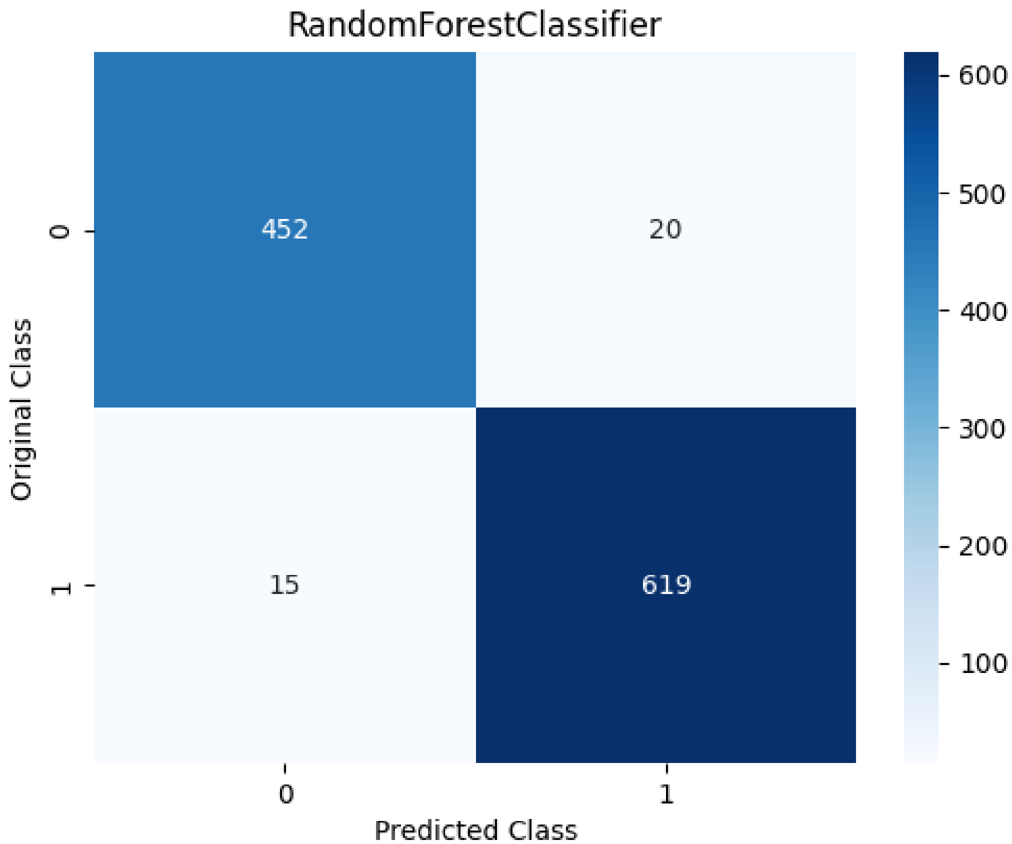
[ ]:
```
print('The accuracy of RandomForest Classifier is: ' , 100.0 *␣
↪accuracy_score(rfc_predict,y_test))
```

```
The accuracy of RandomForest Classifier is:  96.83544303797468
```

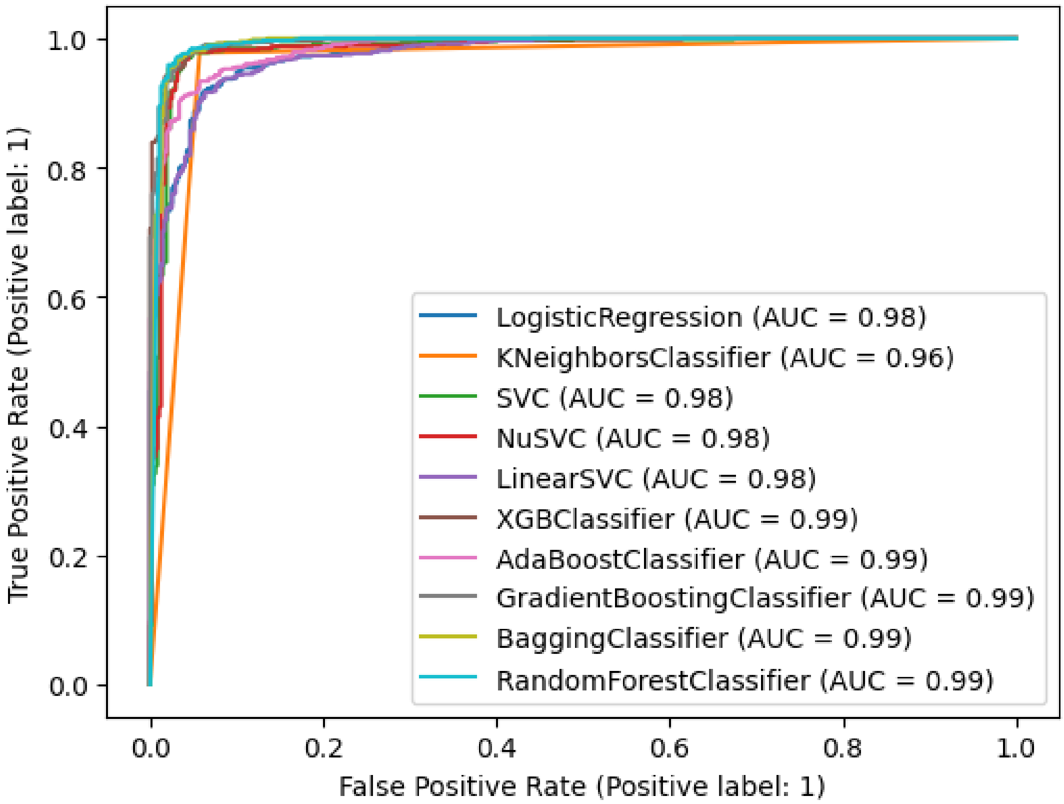
[ ]:
```
print(classification_report(y_test, rfc_predict))
```

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96       472
           1       0.97      0.98      0.97       634

    accuracy                           0.97      1106
   macro avg       0.97      0.97      0.97      1106
weighted avg       0.97      0.97      0.97      1106
```

[ ]:
```
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]
```

```python
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_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_2 (LSTM)               (None, 100)               48000

 batch_normalization_2 (Batc  (None, 100)              400
 hNormalization)

 dense_8 (Dense)             (None, 50)                5050

 dense_9 (Dense)             (None, 25)                1275
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_2 (LSTM)               (None, 100)               48000

 batch_normalization_2 (Batc  (None, 100)              400
 hNormalization)

 dense_8 (Dense)             (None, 50)                5050

 dense_9 (Dense)             (None, 25)                1275

 dense_10 (Dense)            (None, 10)                260

 dense_11 (Dense)            (None, 1)                 11

=================================================================
Total params: 54,996
Trainable params: 54,796
Non-trainable params: 200
_________________________________________________________________
Epoch 1/100
249/249 [==============================] - 4s 6ms/step - loss: 0.2307 -
accuracy: 0.9118 - val_loss: 0.3389 - val_accuracy: 0.9387
```

```
Epoch 2/100
249/249 [==============================] - 1s 5ms/step - loss: 0.1739 -
accuracy: 0.9273 - val_loss: 0.1605 - val_accuracy: 0.9447
Epoch 3/100
249/249 [==============================] - 1s 5ms/step - loss: 0.1549 -
accuracy: 0.9358 - val_loss: 0.1293 - val_accuracy: 0.9437
Epoch 4/100
249/249 [==============================] - 1s 4ms/step - loss: 0.1445 -
accuracy: 0.9382 - val_loss: 0.1344 - val_accuracy: 0.9447
Epoch 5/100
249/249 [==============================] - 1s 4ms/step - loss: 0.1371 -
accuracy: 0.9422 - val_loss: 0.1220 - val_accuracy: 0.9482
Epoch 6/100
249/249 [==============================] - 1s 4ms/step - loss: 0.1275 -
accuracy: 0.9460 - val_loss: 0.1169 - val_accuracy: 0.9503
Epoch 7/100
249/249 [==============================] - 1s 4ms/step - loss: 0.1224 -
accuracy: 0.9489 - val_loss: 0.1202 - val_accuracy: 0.9503
Epoch 8/100
249/249 [==============================] - 1s 4ms/step - loss: 0.1182 -
accuracy: 0.9506 - val_loss: 0.1154 - val_accuracy: 0.9543
Epoch 9/100
249/249 [==============================] - 1s 4ms/step - loss: 0.1150 -
accuracy: 0.9506 - val_loss: 0.1184 - val_accuracy: 0.9523
Epoch 10/100
249/249 [==============================] - 1s 4ms/step - loss: 0.1095 -
accuracy: 0.9556 - val_loss: 0.1120 - val_accuracy: 0.9528
Epoch 11/100
249/249 [==============================] - 1s 4ms/step - loss: 0.1041 -
accuracy: 0.9554 - val_loss: 0.1174 - val_accuracy: 0.9533
Epoch 12/100
249/249 [==============================] - 1s 4ms/step - loss: 0.1027 -
accuracy: 0.9570 - val_loss: 0.1086 - val_accuracy: 0.9523
Epoch 13/100
249/249 [==============================] - 1s 5ms/step - loss: 0.1041 -
accuracy: 0.9567 - val_loss: 0.1154 - val_accuracy: 0.9457
Epoch 14/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0986 -
accuracy: 0.9588 - val_loss: 0.1106 - val_accuracy: 0.9553
Epoch 15/100
249/249 [==============================] - 1s 5ms/step - loss: 0.0958 -
accuracy: 0.9593 - val_loss: 0.1135 - val_accuracy: 0.9533
Epoch 16/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0941 -
accuracy: 0.9603 - val_loss: 0.1097 - val_accuracy: 0.9563
Epoch 17/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0936 -
accuracy: 0.9587 - val_loss: 0.1126 - val_accuracy: 0.9563
```

```
Epoch 18/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0885 -
accuracy: 0.9619 - val_loss: 0.1025 - val_accuracy: 0.9583
Epoch 19/100
249/249 [==============================] - 1s 5ms/step - loss: 0.0875 -
accuracy: 0.9621 - val_loss: 0.1054 - val_accuracy: 0.9568
Epoch 20/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0850 -
accuracy: 0.9619 - val_loss: 0.0981 - val_accuracy: 0.9608
Epoch 21/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0841 -
accuracy: 0.9642 - val_loss: 0.1072 - val_accuracy: 0.9568
Epoch 22/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0812 -
accuracy: 0.9644 - val_loss: 0.1203 - val_accuracy: 0.9573
Epoch 23/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0810 -
accuracy: 0.9663 - val_loss: 0.1092 - val_accuracy: 0.9583
Epoch 24/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0808 -
accuracy: 0.9646 - val_loss: 0.1190 - val_accuracy: 0.9563
Epoch 25/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0783 -
accuracy: 0.9656 - val_loss: 0.1120 - val_accuracy: 0.9608
Epoch 26/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0750 -
accuracy: 0.9667 - val_loss: 0.1068 - val_accuracy: 0.9613
Epoch 27/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0730 -
accuracy: 0.9673 - val_loss: 0.1080 - val_accuracy: 0.9578
Epoch 28/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0724 -
accuracy: 0.9681 - val_loss: 0.1273 - val_accuracy: 0.9523
Epoch 29/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0702 -
accuracy: 0.9683 - val_loss: 0.1055 - val_accuracy: 0.9583
Epoch 30/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0699 -
accuracy: 0.9685 - val_loss: 0.1160 - val_accuracy: 0.9593
Epoch 31/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0693 -
accuracy: 0.9690 - val_loss: 0.1142 - val_accuracy: 0.9578
Epoch 32/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0695 -
accuracy: 0.9706 - val_loss: 0.1146 - val_accuracy: 0.9548
Epoch 33/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0695 -
accuracy: 0.9696 - val_loss: 0.1130 - val_accuracy: 0.9618
```

```
Epoch 34/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0672 -
accuracy: 0.9703 - val_loss: 0.1137 - val_accuracy: 0.9608
Epoch 35/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0658 -
accuracy: 0.9707 - val_loss: 0.1183 - val_accuracy: 0.9563
Epoch 36/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0683 -
accuracy: 0.9686 - val_loss: 0.1067 - val_accuracy: 0.9613
Epoch 37/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0658 -
accuracy: 0.9714 - val_loss: 0.1128 - val_accuracy: 0.9583
Epoch 38/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0682 -
accuracy: 0.9702 - val_loss: 0.1305 - val_accuracy: 0.9538
Epoch 39/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0692 -
accuracy: 0.9697 - val_loss: 0.1096 - val_accuracy: 0.9603
Epoch 40/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0624 -
accuracy: 0.9712 - val_loss: 0.1094 - val_accuracy: 0.9553
Epoch 41/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0659 -
accuracy: 0.9714 - val_loss: 0.1091 - val_accuracy: 0.9603
Epoch 42/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0606 -
accuracy: 0.9726 - val_loss: 0.1157 - val_accuracy: 0.9608
Epoch 43/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0617 -
accuracy: 0.9736 - val_loss: 0.1210 - val_accuracy: 0.9588
Epoch 44/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0595 -
accuracy: 0.9732 - val_loss: 0.1351 - val_accuracy: 0.9578
Epoch 45/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0626 -
accuracy: 0.9725 - val_loss: 0.1234 - val_accuracy: 0.9608
Epoch 46/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0630 -
accuracy: 0.9720 - val_loss: 0.1094 - val_accuracy: 0.9608
Epoch 47/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0595 -
accuracy: 0.9744 - val_loss: 0.1094 - val_accuracy: 0.9578
Epoch 48/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0582 -
accuracy: 0.9735 - val_loss: 0.1154 - val_accuracy: 0.9568
Epoch 49/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0598 -
accuracy: 0.9732 - val_loss: 0.1136 - val_accuracy: 0.9623
```

```
Epoch 50/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0588 -
accuracy: 0.9732 - val_loss: 0.1121 - val_accuracy: 0.9648
Epoch 51/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0599 -
accuracy: 0.9734 - val_loss: 0.1124 - val_accuracy: 0.9618
Epoch 52/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0561 -
accuracy: 0.9735 - val_loss: 0.1165 - val_accuracy: 0.9608
Epoch 53/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0610 -
accuracy: 0.9742 - val_loss: 0.1159 - val_accuracy: 0.9613
Epoch 54/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0556 -
accuracy: 0.9761 - val_loss: 0.1159 - val_accuracy: 0.9593
Epoch 55/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0578 -
accuracy: 0.9744 - val_loss: 0.1069 - val_accuracy: 0.9613
Epoch 56/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0547 -
accuracy: 0.9745 - val_loss: 0.1092 - val_accuracy: 0.9598
Epoch 57/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0536 -
accuracy: 0.9758 - val_loss: 0.1223 - val_accuracy: 0.9608
Epoch 58/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0506 -
accuracy: 0.9771 - val_loss: 0.1267 - val_accuracy: 0.9628
Epoch 59/100
249/249 [==============================] - 1s 5ms/step - loss: 0.0522 -
accuracy: 0.9764 - val_loss: 0.1220 - val_accuracy: 0.9663
Epoch 60/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0519 -
accuracy: 0.9774 - val_loss: 0.1112 - val_accuracy: 0.9623
Epoch 61/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0519 -
accuracy: 0.9769 - val_loss: 0.1375 - val_accuracy: 0.9623
Epoch 62/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0581 -
accuracy: 0.9742 - val_loss: 0.1410 - val_accuracy: 0.9578
Epoch 63/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0561 -
accuracy: 0.9735 - val_loss: 0.1165 - val_accuracy: 0.9613
Epoch 64/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0543 -
accuracy: 0.9763 - val_loss: 0.1278 - val_accuracy: 0.9608
Epoch 65/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0523 -
accuracy: 0.9744 - val_loss: 0.1167 - val_accuracy: 0.9578
```

```
Epoch 66/100
249/249 [==============================] - 1s 5ms/step - loss: 0.0516 -
accuracy: 0.9783 - val_loss: 0.1178 - val_accuracy: 0.9608
Epoch 67/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0492 -
accuracy: 0.9776 - val_loss: 0.1278 - val_accuracy: 0.9618
Epoch 68/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0496 -
accuracy: 0.9770 - val_loss: 0.1227 - val_accuracy: 0.9633
Epoch 69/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0554 -
accuracy: 0.9746 - val_loss: 0.1302 - val_accuracy: 0.9648
Epoch 70/100
249/249 [==============================] - 1s 5ms/step - loss: 0.0534 -
accuracy: 0.9759 - val_loss: 0.1109 - val_accuracy: 0.9653
Epoch 71/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0505 -
accuracy: 0.9764 - val_loss: 0.1197 - val_accuracy: 0.9623
Epoch 72/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0497 -
accuracy: 0.9760 - val_loss: 0.1224 - val_accuracy: 0.9628
Epoch 73/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0518 -
accuracy: 0.9773 - val_loss: 0.1334 - val_accuracy: 0.9603
Epoch 74/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0507 -
accuracy: 0.9766 - val_loss: 0.1117 - val_accuracy: 0.9638
Epoch 75/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0475 -
accuracy: 0.9776 - val_loss: 0.1209 - val_accuracy: 0.9643
Epoch 76/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0493 -
accuracy: 0.9785 - val_loss: 0.1409 - val_accuracy: 0.9618
Epoch 77/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0486 -
accuracy: 0.9770 - val_loss: 0.1264 - val_accuracy: 0.9623
Epoch 78/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0502 -
accuracy: 0.9763 - val_loss: 0.1176 - val_accuracy: 0.9608
Epoch 79/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0466 -
accuracy: 0.9773 - val_loss: 0.1271 - val_accuracy: 0.9593
Epoch 80/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0486 -
accuracy: 0.9765 - val_loss: 0.1248 - val_accuracy: 0.9638
Epoch 81/100
249/249 [==============================] - 1s 5ms/step - loss: 0.0508 -
accuracy: 0.9773 - val_loss: 0.1257 - val_accuracy: 0.9633
```

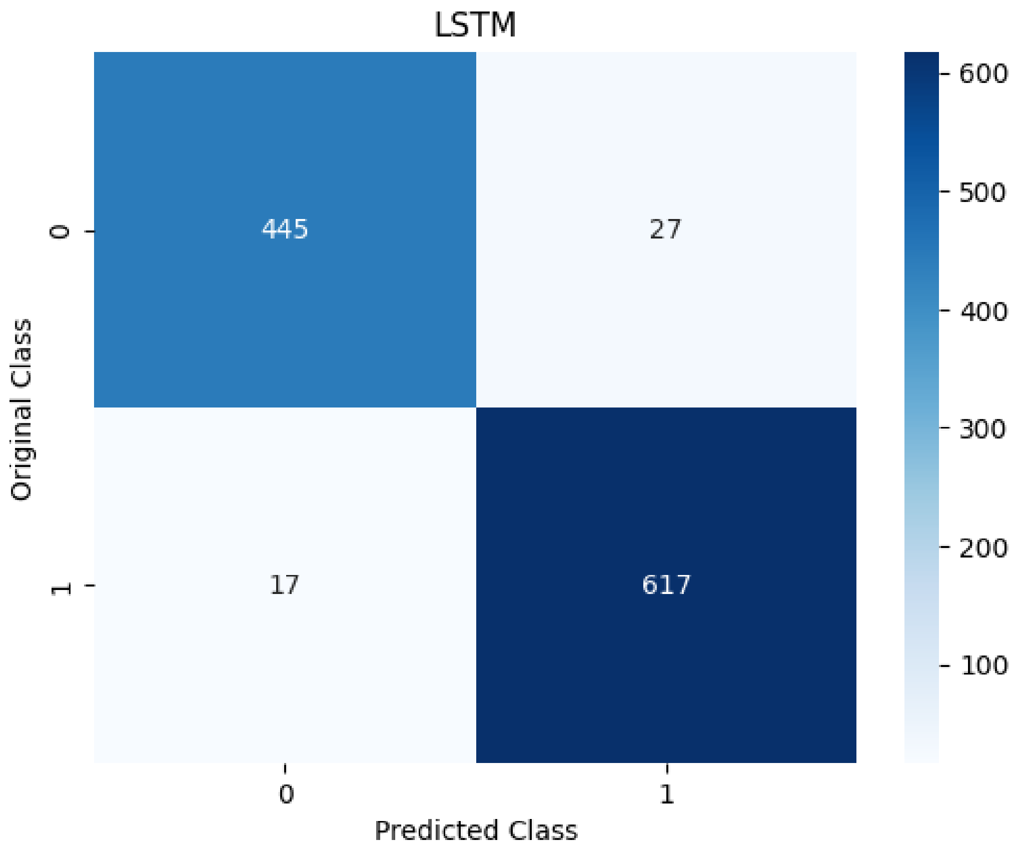
```
Epoch 82/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0501 -
accuracy: 0.9763 - val_loss: 0.1236 - val_accuracy: 0.9613
Epoch 83/100
249/249 [==============================] - 1s 5ms/step - loss: 0.0482 -
accuracy: 0.9773 - val_loss: 0.1340 - val_accuracy: 0.9613
Epoch 84/100
249/249 [==============================] - 1s 5ms/step - loss: 0.0468 -
accuracy: 0.9783 - val_loss: 0.1389 - val_accuracy: 0.9593
Epoch 85/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0501 -
accuracy: 0.9768 - val_loss: 0.1433 - val_accuracy: 0.9583
Epoch 86/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0463 -
accuracy: 0.9776 - val_loss: 0.1319 - val_accuracy: 0.9628
Epoch 87/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0435 -
accuracy: 0.9799 - val_loss: 0.1170 - val_accuracy: 0.9593
Epoch 88/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0444 -
accuracy: 0.9799 - val_loss: 0.1281 - val_accuracy: 0.9598
Epoch 89/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0443 -
accuracy: 0.9785 - val_loss: 0.1297 - val_accuracy: 0.9623
Epoch 90/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0441 -
accuracy: 0.9781 - val_loss: 0.1321 - val_accuracy: 0.9618
Epoch 91/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0520 -
accuracy: 0.9755 - val_loss: 0.1391 - val_accuracy: 0.9633
Epoch 92/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0413 -
accuracy: 0.9803 - val_loss: 0.1518 - val_accuracy: 0.9608
Epoch 93/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0440 -
accuracy: 0.9778 - val_loss: 0.1303 - val_accuracy: 0.9638
Epoch 94/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0477 -
accuracy: 0.9768 - val_loss: 0.1466 - val_accuracy: 0.9658
Epoch 95/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0438 -
accuracy: 0.9786 - val_loss: 0.1415 - val_accuracy: 0.9628
Epoch 96/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0535 -
accuracy: 0.9768 - val_loss: 0.1520 - val_accuracy: 0.9588
Epoch 97/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0482 -
accuracy: 0.9774 - val_loss: 0.1477 - val_accuracy: 0.9608
```

```
Epoch 98/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0482 -
accuracy: 0.9788 - val_loss: 0.1267 - val_accuracy: 0.9628
Epoch 99/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0471 -
accuracy: 0.9774 - val_loss: 0.1453 - val_accuracy: 0.9608
Epoch 100/100
249/249 [==============================] - 1s 4ms/step - loss: 0.0425 -
accuracy: 0.9798 - val_loss: 0.1459 - val_accuracy: 0.9643
Test results - Loss: 0.1807340830564499 - Accuracy: 96.02169990539551%
```

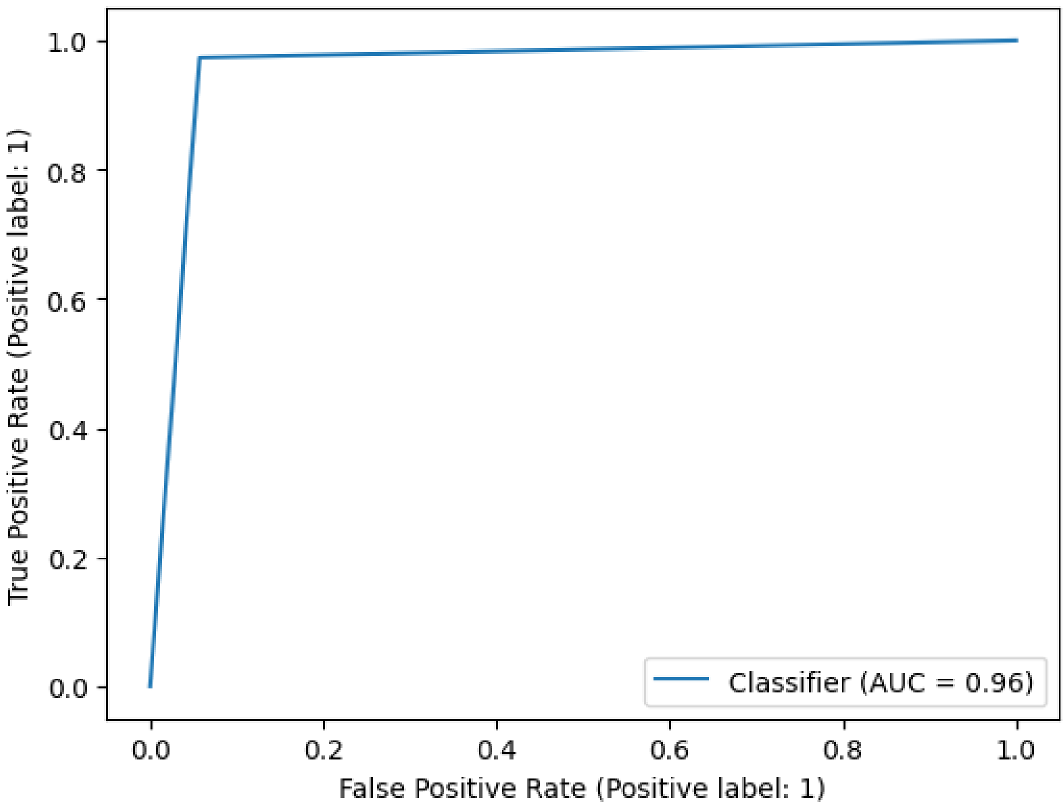
[ ]:
```
lstm_predict_proba = lstm_model.predict(X_test_reshape, batch_size=32)
lstm_predict_class = (lstm_predict_proba > 0.5).astype("int32")
print(classification_report(y_test, lstm_predict_class))
```

```
35/35 [==============================] - 1s 2ms/step
              precision    recall  f1-score   support

           0       0.96      0.94      0.95       472
           1       0.96      0.97      0.97       634

    accuracy                           0.96      1106
   macro avg       0.96      0.96      0.96      1106
weighted avg       0.96      0.96      0.96      1106
```

[ ]:
```
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)