correlation_target _label_30 8020 split .03 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)
```

```
[ ]: ##print(df.corr()['Result'].sort_values())
     ## correlation values of features with target label
     corr_col = abs(df.corr()['Result']).sort_values(ascending=False)
     corr_col = corr_col.rename_axis('Col').reset_index(name='Correlation')
     corr_col
```

```
[ ]:                           Col  Correlation
     0                       Result     1.000000
     1                SSLfinal_State     0.714741
     2                 URL_of_Anchor     0.692935
     3                 Prefix_Suffix     0.348606
     4                   web_traffic     0.346103
     5             having_Sub_Domain     0.298323
     6                   Request_URL     0.253372
     7                 Links_in_tags     0.248229
     8   Domain_registeration_length     0.225789
     9                           SFH     0.221419
     10                 Google_Index     0.128950
     11                age_of_domain     0.121496
     12                    Page_Rank     0.104645
     13    having_IPhaving_IP_Address     0.094160
     14           Statistical_report     0.079857
     15                    DNSRecord     0.075718
     16           Shortining_Service     0.067966
     17                 Abnormal_URL     0.060488
     18                URLURL_Length     0.057430
     19             having_At_Symbol     0.052948
     20                 on_mouseover     0.041838
     21                  HTTPS_token     0.039854
     22       double_slash_redirecting     0.038608
     23                         port     0.036419
     24         Links_pointing_to_page     0.032574
     25                     Redirect     0.020113
     26           Submitting_to_email     0.018249
     27                   RightClick     0.012653
     28                       Iframe     0.003394
     29                      Favicon     0.000280
     30                  popUpWidnow     0.000086
```

```
[ ]: def correlation (corr_col, threshold):
             corr_feature = set()
             for index, row in corr_col.iterrows():
                     if row['Correlation'] < threshold or np.
      ↪isnan(row['Correlation']):
                             corr_feature.add(row['Col'])
             return corr_feature
```

```
[ ]: corr_feature = correlation(corr_col,.03)
     len(set(corr_feature))
```

```
[ ]: 6
```

```
[ ]: corr_feature
```

```
[ ]: {'Favicon',
      'Iframe',
      'Redirect',
      'RightClick',
      'Submitting_to_email',
      'popUpWidnow'}
```

```
[ ]: df.drop(corr_feature, axis=1, inplace=True)
```

```
[ ]: # # Remove features having correlation coeff. between +/- 0.03
     # df.drop(['Favicon','Iframe','Redirect',
     #                  'popUpWidnow','RightClick','Submitting_to_email'], axis=1,␣
      ↪inplace=True)
```

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

```
[ ]: 25
```

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

```
[ ]: 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
```

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

```
[ ]: # #Using Pearson Correlation
     # plt.figure(figsize=(30,30))
     # corr = df.corr()
     # sns.heatmap(corr, annot=True, cmap=plt.cm.CMRmap_r)
     # plt.show()
```

```
[ ]: # # with the following function we can select highly correlated features
     # # it will remove the first feature that is correlated with anything other␣
      ↪feature
```

```
# def correlation(dataset, threshold):
#     col_corr = set()  # Set of all the names of correlated columns
#     corr_matrix = dataset.corr()
#     for i in range(len(corr_matrix.columns)):
#         for j in range(i):
#             if abs(corr_matrix.iloc[i, j]) > threshold: # we are interested␣
 ↪in absolute coeff value
#                 colname = corr_matrix.columns[i]  # getting the name of column
#                 col_corr.add(colname)
#     return col_corr
```

[ ]:
```
# corr_features = correlation(df, 0.8)
# len(set(corr_features))
```

[ ]:
```
# corr_features
```

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

[ ]:
```
#from sklearn import preprocessing



# col =df[df.columns[:]]

# lab_en= preprocessing.LabelEncoder()

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

# df.head()
```

[ ]:
```
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
```

[ ]:
```python
# 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, 24), (2211, 24), (8844, 1), (2211, 1))
```

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

[ ]:
```python
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.
```

[ ]:
```python
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': 1, 'max_iter': 2500, 'penalty': 'l2', 'solver': 'lbfgs'}
LogisticRegression(C=1, max_iter=2500)
0.9269570774854922
```

[ ]:
```
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)
```

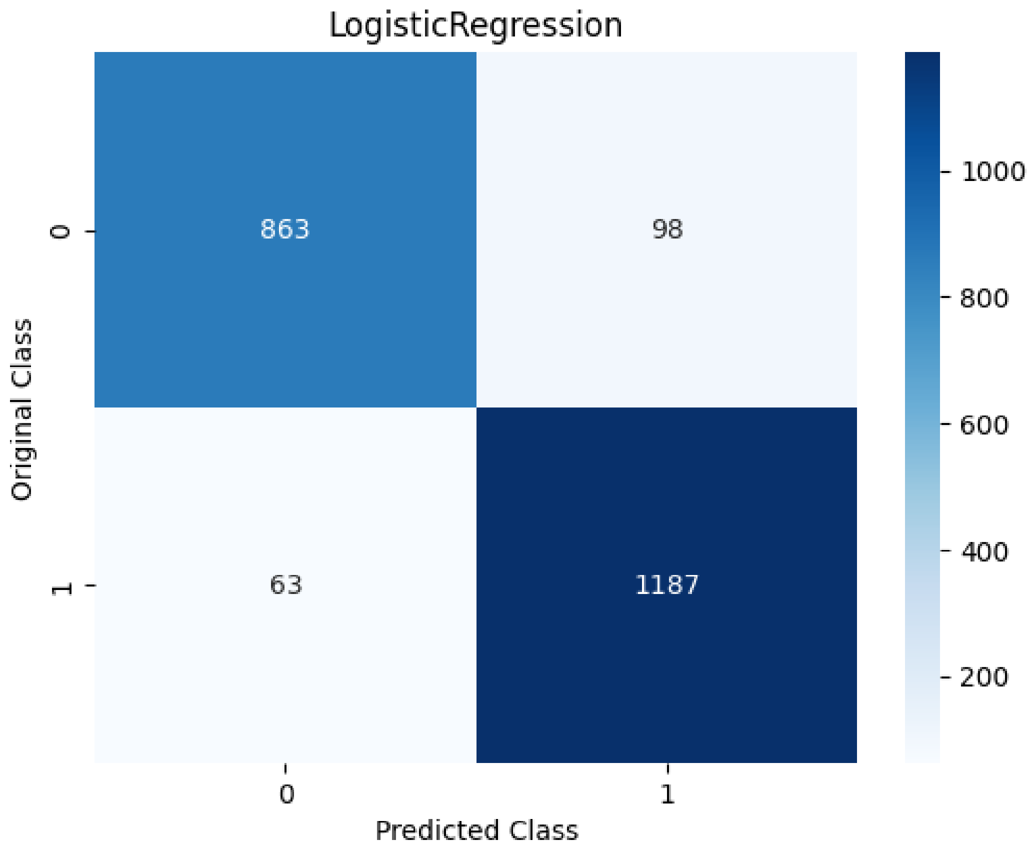
[ ]:
```
print ("Accuracy of logr classifier : ", accuracy_score(y_test,
 ↪logr_predict)*100)
```

```
Accuracy of logr classifier :  92.71822704658526
```

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

```
              precision    recall  f1-score   support

           0       0.93      0.90      0.91       961
           1       0.92      0.95      0.94      1250

    accuracy                           0.93      2211
   macro avg       0.93      0.92      0.93      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()
```

[ ]:
```
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)
0.9616696065649206
```

[ ]:
```
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))
```

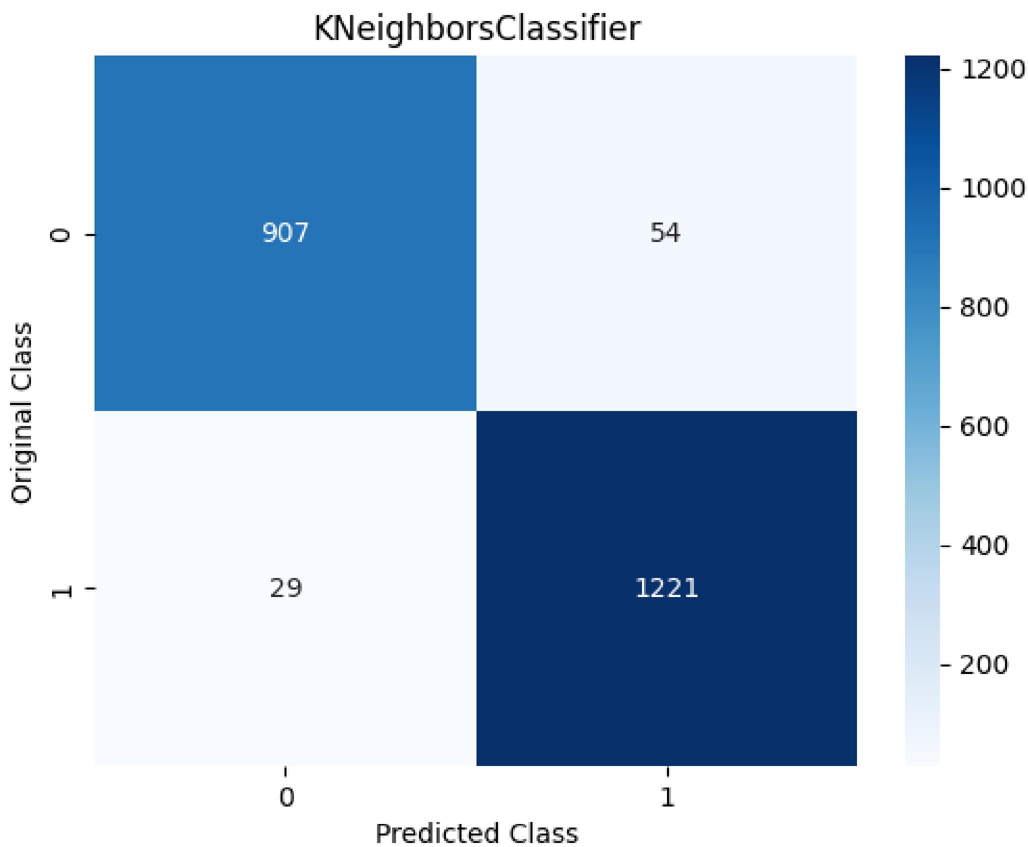
```
The accuracy of knn Classifier is:  96.24604251469923
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.94      0.96       961
           1       0.96      0.98      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, 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, 20],
                   'gamma': [1, 0.1, 0.01, 0.001],
                   '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 100 candidates, totalling 1000 fits
{'C': 10, 'gamma': 1, 'kernel': 'rbf'}
SVC(C=10, gamma=1)
0.9643832604749851
```

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

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

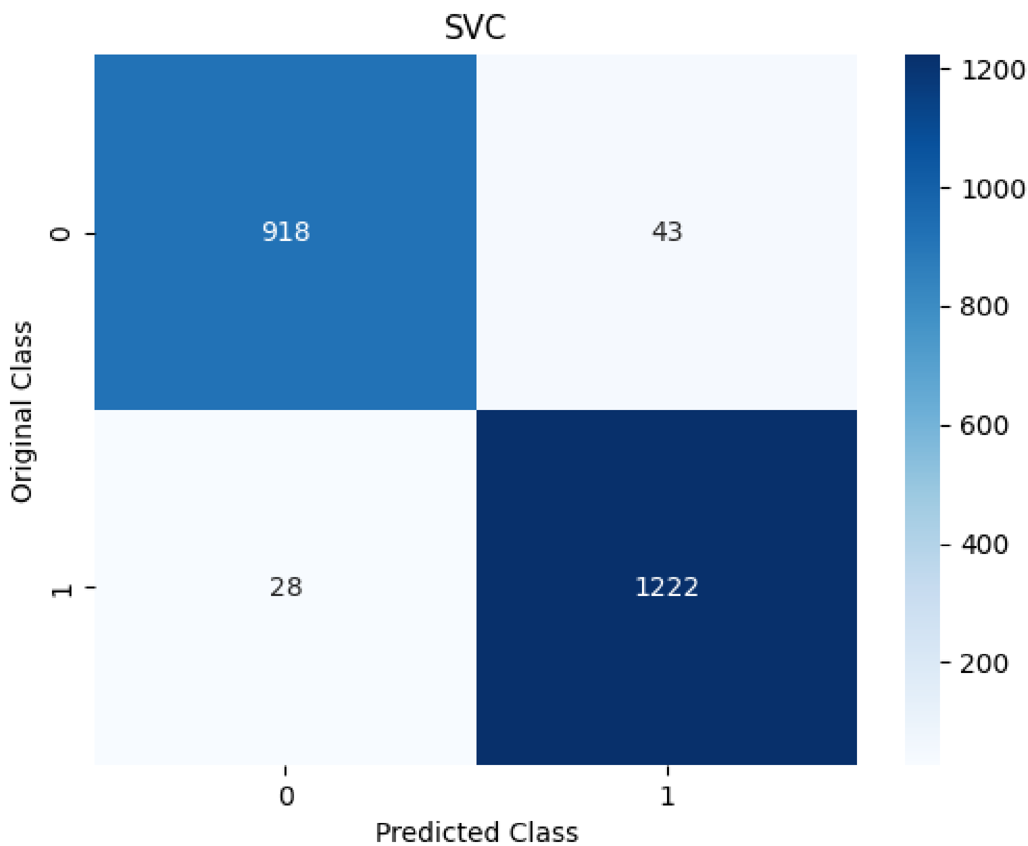
```
[ ]: print('The accuracy of svc Classifier is: ', 100.0 * accuracy_score(y_test,␣
      ↪svc_predict))
```

```
The accuracy of svc Classifier is:  96.78878335594754
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.96      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, 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, 0.001],
                   '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 32 candidates, totalling 320 fits
{'gamma': 1, 'kernel': 'rbf', 'nu': 0.1}
NuSVC(gamma=1, nu=0.1)
0.9643832604749851
```

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

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

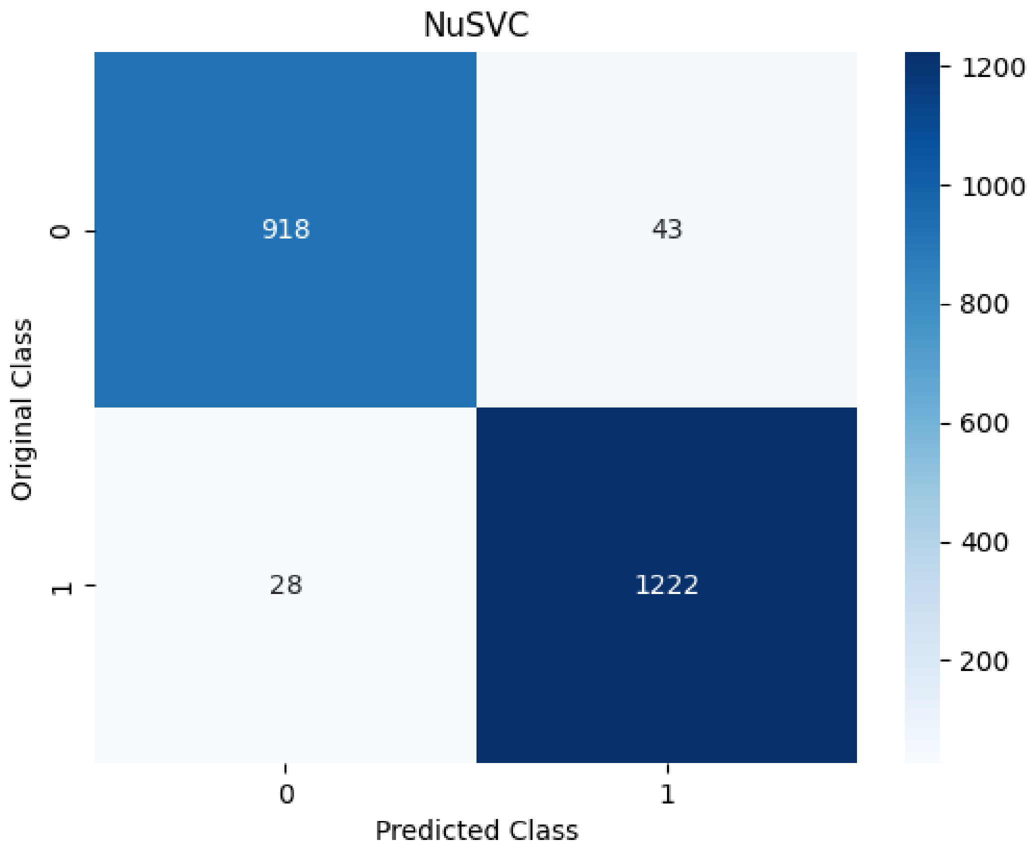
[ ]:
```
print('The accuracy of nusvc Classifier is: ', 100.0 * accuracy_score(y_test,
    nusvc_predict))
```

```
The accuracy of nusvc Classifier is:  96.78878335594754
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.96      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, 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)
0.9263919779124166
```

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

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

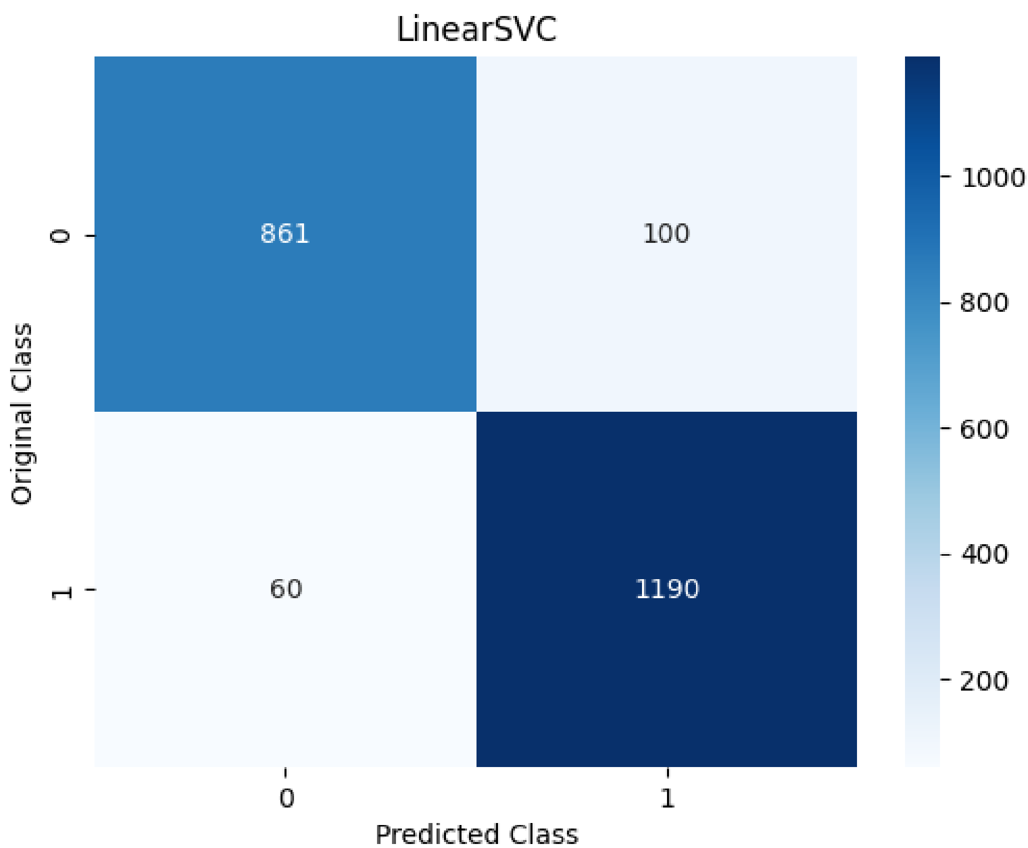
[ ]:
```
print('The accuracy of lsvc Classifier is: ', 100.0 * accuracy_score(y_test,␣
↪lsvc_predict))
```

```
The accuracy of lsvc Classifier is:  92.76345545002262
```

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

```
              precision    recall  f1-score   support

           0       0.93      0.90      0.91       961
           1       0.92      0.95      0.94      1250

    accuracy                           0.93      2211
   macro avg       0.93      0.92      0.93      2211
weighted avg       0.93      0.93      0.93      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)
0.9343069509420456
```

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

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

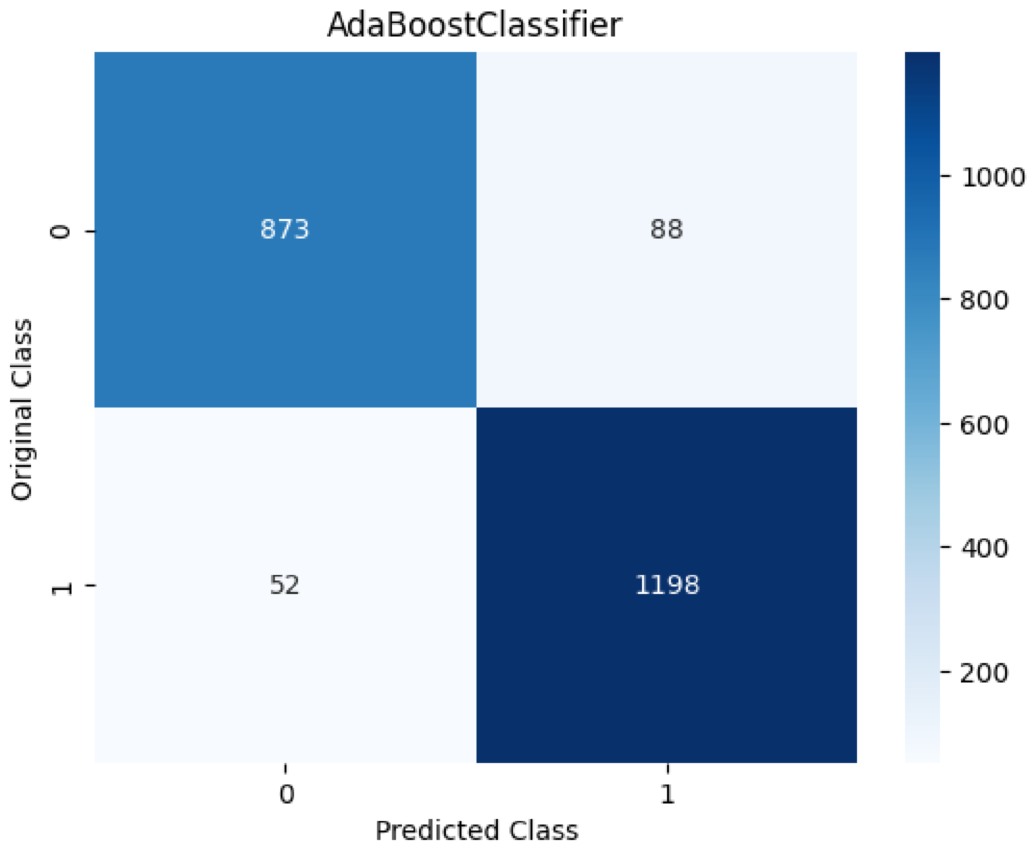
[ ]:
```
print('The accuracy of Ada Boost Classifier is: ', 100.0 *␣
↪accuracy_score(ada_predict,y_test))
```

```
The accuracy of Ada Boost Classifier is:  93.66802351876979
```

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

```
              precision    recall  f1-score   support

           0       0.94      0.91      0.93       961
           1       0.93      0.96      0.94      1250

    accuracy                           0.94      2211
   macro avg       0.94      0.93      0.94      2211
weighted avg       0.94      0.94      0.94      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, …)
0.9719584835237874
```

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

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

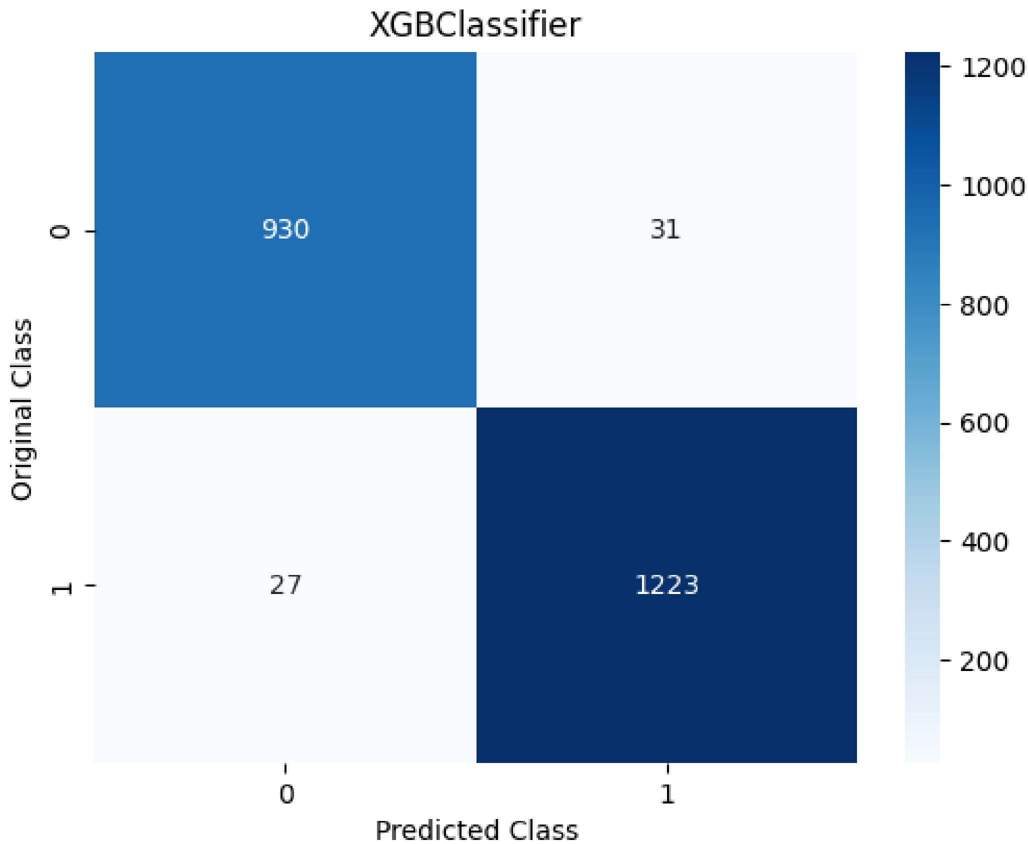
[ ]:
```
print('The accuracy of XGBoost Classifier is: ' , 100.0 *␣
 ↪accuracy_score(xgb_predict,y_test))
```

```
The accuracy of XGBoost Classifier is:  97.3767526006332
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.97      0.97       961
           1       0.98      0.98      0.98      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)
0.9698117186900836
```

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

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

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

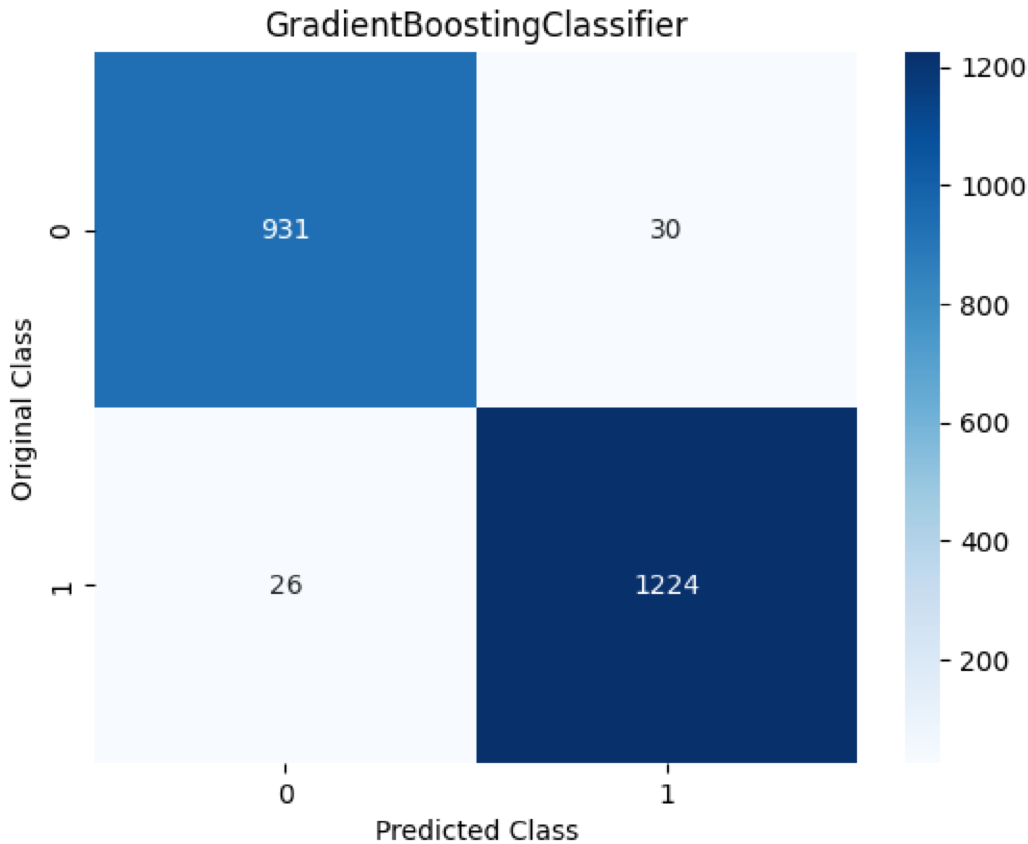
[ ]:
```
print('The accuracy of GradientBoost Classifier is: ' , 100.0 *␣
 ↪accuracy_score(gbc_predict,y_test))
```

```
The accuracy of GradientBoost Classifier is:  97.46720940750791
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.97      0.97       961
           1       0.98      0.98      0.98      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, 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': 150}
BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=150)
0.9687929800342561
```

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

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

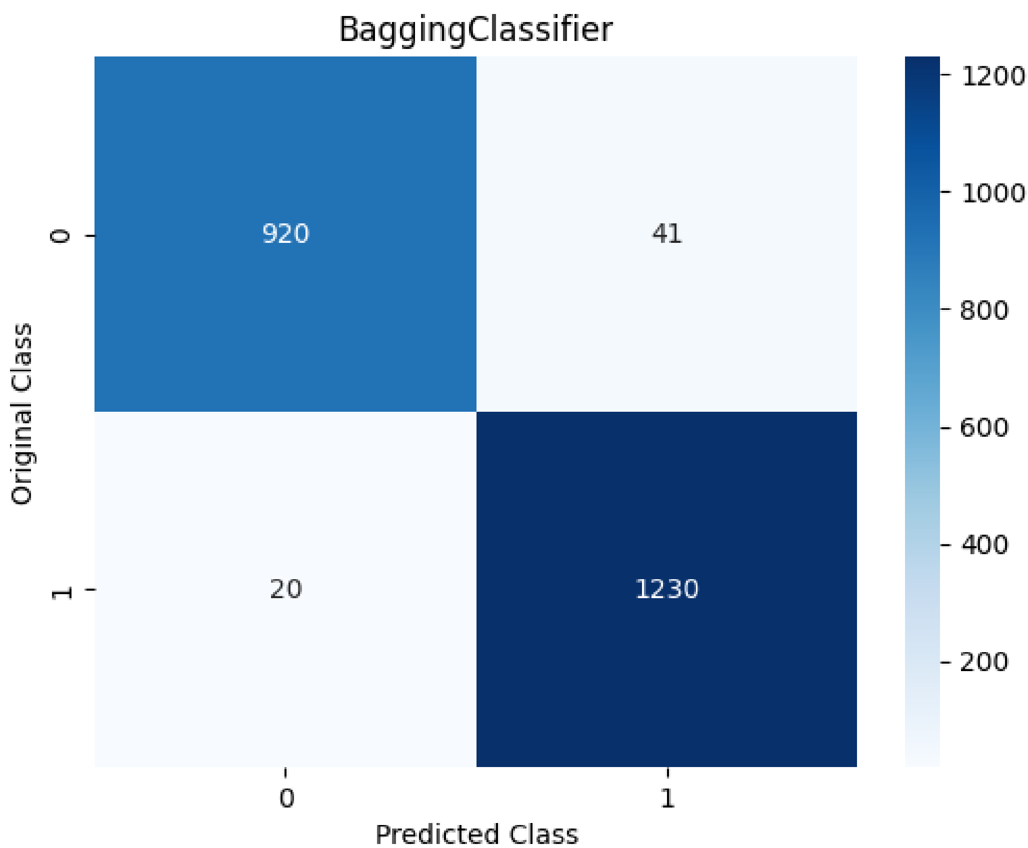
[ ]:
```
print('The accuracy of Bagging Classifier is: ' , 100.0 *␣
↪accuracy_score(y_test, bag_predict))
```

```
The accuracy of Bagging Classifier is:  97.24106739032112
```

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

```
              precision    recall  f1-score   support

           0       0.98      0.96      0.97       961
           1       0.97      0.98      0.98      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, bag_predict), annot=True, fmt='g',␣
↪cmap='Blues')
plt.title("BaggingClassifier")
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



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

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

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

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

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

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

```
Fitting 10 folds for each of 5 candidates, totalling 50 fits
{'n_estimators': 100}
RandomForestClassifier()
0.9709412787279188
```

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

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

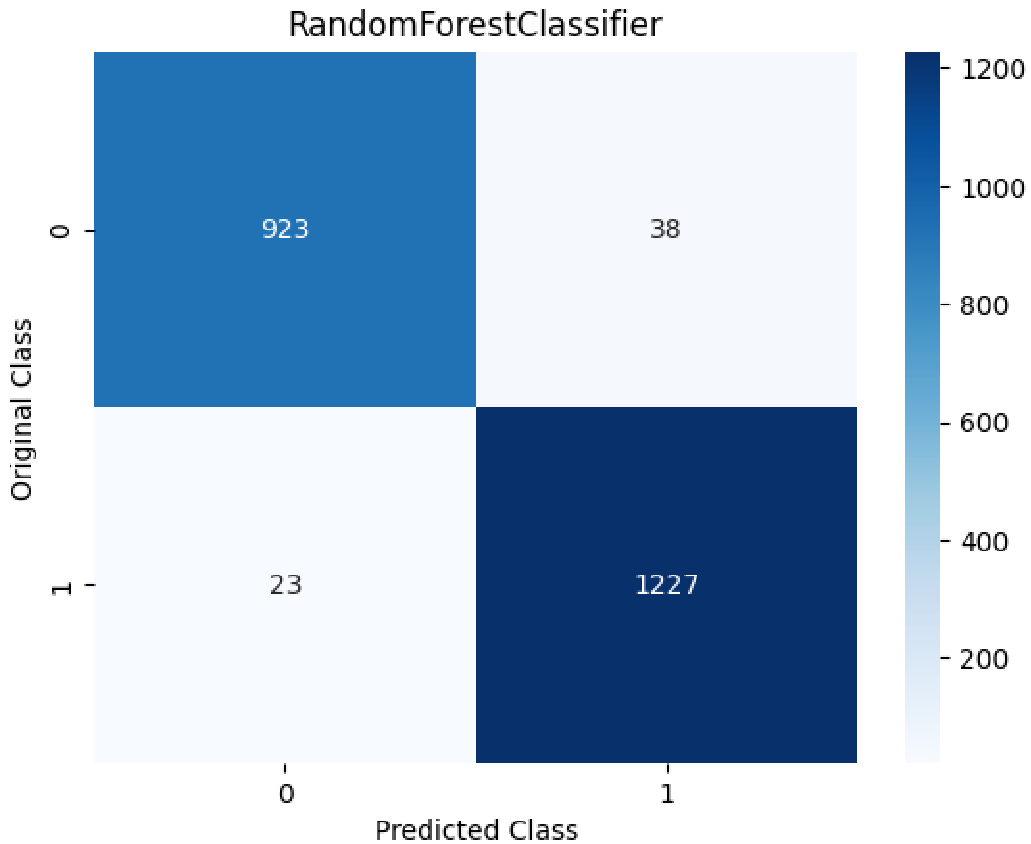
[ ]:
```
print('The accuracy of RandomForest Classifier is: ' , 100.0 *␣
 ↪accuracy_score(rfc_predict,y_test))
```

```
The accuracy of RandomForest Classifier is:  97.24106739032112
```

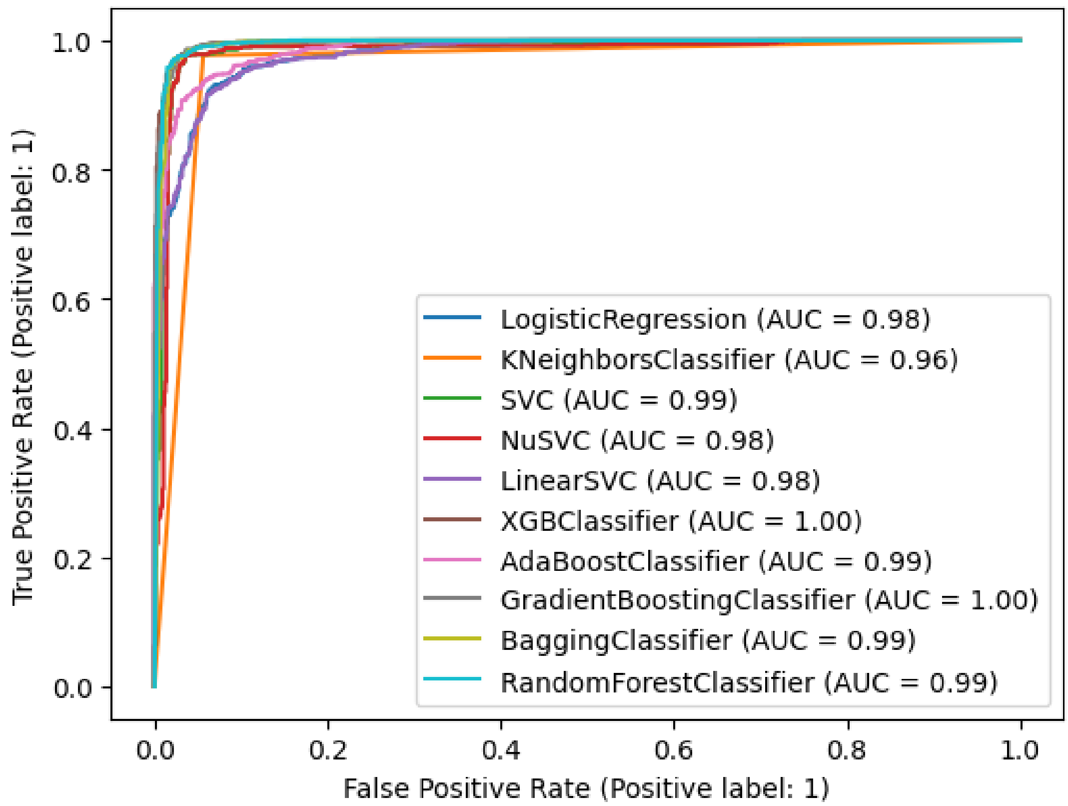
[ ]:
```
print(classification_report(y_test, rfc_predict))
```

```
              precision    recall  f1-score   support

           0       0.98      0.96      0.97       961
           1       0.97      0.98      0.98      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_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_4 (LSTM)               (None, 100)               50000

 batch_normalization_4 (Batc  (None, 100)              400
 hNormalization)

 dense_16 (Dense)            (None, 50)                5050
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_4 (LSTM)               (None, 100)               50000

 batch_normalization_4 (Batc  (None, 100)              400
 hNormalization)

 dense_16 (Dense)            (None, 50)                5050

 dense_17 (Dense)            (None, 25)                1275

 dense_18 (Dense)            (None, 10)                260

 dense_19 (Dense)            (None, 1)                 11

=================================================================
Total params: 56,996
Trainable params: 56,796
Non-trainable params: 200
_________________________________________________________________
Epoch 1/100
222/222 [==============================] - 4s 7ms/step - loss: 0.2457 -
accuracy: 0.9071 - val_loss: 0.3533 - val_accuracy: 0.9079
Epoch 2/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1710 -
```

```
accuracy: 0.9278 - val_loss: 0.1759 - val_accuracy: 0.9469
Epoch 3/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1559 -
accuracy: 0.9343 - val_loss: 0.1349 - val_accuracy: 0.9469
Epoch 4/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1386 -
accuracy: 0.9415 - val_loss: 0.1289 - val_accuracy: 0.9469
Epoch 5/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1279 -
accuracy: 0.9470 - val_loss: 0.1352 - val_accuracy: 0.9525
Epoch 6/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1224 -
accuracy: 0.9495 - val_loss: 0.1259 - val_accuracy: 0.9429
Epoch 7/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1114 -
accuracy: 0.9532 - val_loss: 0.1129 - val_accuracy: 0.9542
Epoch 8/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1040 -
accuracy: 0.9575 - val_loss: 0.1420 - val_accuracy: 0.9469
Epoch 9/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1007 -
accuracy: 0.9563 - val_loss: 0.1173 - val_accuracy: 0.9542
Epoch 10/100
222/222 [==============================] - 1s 4ms/step - loss: 0.1024 -
accuracy: 0.9562 - val_loss: 0.1273 - val_accuracy: 0.9463
Epoch 11/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0877 -
accuracy: 0.9634 - val_loss: 0.1199 - val_accuracy: 0.9508
Epoch 12/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0878 -
accuracy: 0.9616 - val_loss: 0.1222 - val_accuracy: 0.9486
Epoch 13/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0842 -
accuracy: 0.9644 - val_loss: 0.1173 - val_accuracy: 0.9531
Epoch 14/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0798 -
accuracy: 0.9661 - val_loss: 0.1125 - val_accuracy: 0.9536
Epoch 15/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0821 -
accuracy: 0.9654 - val_loss: 0.1110 - val_accuracy: 0.9559
Epoch 16/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0839 -
accuracy: 0.9630 - val_loss: 0.1142 - val_accuracy: 0.9559
Epoch 17/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0792 -
accuracy: 0.9669 - val_loss: 0.0996 - val_accuracy: 0.9644
Epoch 18/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0732 -
```

```
accuracy: 0.9665 - val_loss: 0.1255 - val_accuracy: 0.9548
Epoch 19/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0762 -
accuracy: 0.9683 - val_loss: 0.1214 - val_accuracy: 0.9525
Epoch 20/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0675 -
accuracy: 0.9713 - val_loss: 0.1071 - val_accuracy: 0.9536
Epoch 21/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0756 -
accuracy: 0.9689 - val_loss: 0.1262 - val_accuracy: 0.9491
Epoch 22/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0637 -
accuracy: 0.9737 - val_loss: 0.1041 - val_accuracy: 0.9610
Epoch 23/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0679 -
accuracy: 0.9714 - val_loss: 0.1084 - val_accuracy: 0.9582
Epoch 24/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0661 -
accuracy: 0.9716 - val_loss: 0.1021 - val_accuracy: 0.9633
Epoch 25/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0627 -
accuracy: 0.9729 - val_loss: 0.1234 - val_accuracy: 0.9559
Epoch 26/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0654 -
accuracy: 0.9726 - val_loss: 0.1377 - val_accuracy: 0.9469
Epoch 27/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0758 -
accuracy: 0.9693 - val_loss: 0.1316 - val_accuracy: 0.9599
Epoch 28/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0772 -
accuracy: 0.9686 - val_loss: 0.1245 - val_accuracy: 0.9536
Epoch 29/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0594 -
accuracy: 0.9755 - val_loss: 0.1023 - val_accuracy: 0.9599
Epoch 30/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0572 -
accuracy: 0.9743 - val_loss: 0.1318 - val_accuracy: 0.9559
Epoch 31/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0520 -
accuracy: 0.9771 - val_loss: 0.1493 - val_accuracy: 0.9508
Epoch 32/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0577 -
accuracy: 0.9760 - val_loss: 0.1092 - val_accuracy: 0.9621
Epoch 33/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0632 -
accuracy: 0.9717 - val_loss: 0.1170 - val_accuracy: 0.9616
Epoch 34/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0595 -
```

```
accuracy: 0.9750 - val_loss: 0.1095 - val_accuracy: 0.9593
Epoch 35/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0624 -
accuracy: 0.9740 - val_loss: 0.1227 - val_accuracy: 0.9582
Epoch 36/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0546 -
accuracy: 0.9755 - val_loss: 0.1416 - val_accuracy: 0.9610
Epoch 37/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0547 -
accuracy: 0.9750 - val_loss: 0.1216 - val_accuracy: 0.9593
Epoch 38/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0561 -
accuracy: 0.9760 - val_loss: 0.1206 - val_accuracy: 0.9627
Epoch 39/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0558 -
accuracy: 0.9754 - val_loss: 0.1275 - val_accuracy: 0.9559
Epoch 40/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0516 -
accuracy: 0.9767 - val_loss: 0.1223 - val_accuracy: 0.9593
Epoch 41/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0517 -
accuracy: 0.9777 - val_loss: 0.1157 - val_accuracy: 0.9661
Epoch 42/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0509 -
accuracy: 0.9792 - val_loss: 0.1238 - val_accuracy: 0.9604
Epoch 43/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0525 -
accuracy: 0.9794 - val_loss: 0.1248 - val_accuracy: 0.9570
Epoch 44/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0784 -
accuracy: 0.9740 - val_loss: 0.1317 - val_accuracy: 0.9587
Epoch 45/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0480 -
accuracy: 0.9780 - val_loss: 0.1234 - val_accuracy: 0.9627
Epoch 46/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0518 -
accuracy: 0.9778 - val_loss: 0.1194 - val_accuracy: 0.9650
Epoch 47/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0484 -
accuracy: 0.9804 - val_loss: 0.1251 - val_accuracy: 0.9678
Epoch 48/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0536 -
accuracy: 0.9758 - val_loss: 0.1125 - val_accuracy: 0.9655
Epoch 49/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0485 -
accuracy: 0.9782 - val_loss: 0.1282 - val_accuracy: 0.9621
Epoch 50/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0469 -
```

```
accuracy: 0.9796 - val_loss: 0.1250 - val_accuracy: 0.9593
Epoch 51/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0534 -
accuracy: 0.9758 - val_loss: 0.1275 - val_accuracy: 0.9582
Epoch 52/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0604 -
accuracy: 0.9741 - val_loss: 0.1304 - val_accuracy: 0.9559
Epoch 53/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0472 -
accuracy: 0.9782 - val_loss: 0.1121 - val_accuracy: 0.9644
Epoch 54/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0476 -
accuracy: 0.9802 - val_loss: 0.1144 - val_accuracy: 0.9655
Epoch 55/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0440 -
accuracy: 0.9802 - val_loss: 0.1129 - val_accuracy: 0.9695
Epoch 56/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0453 -
accuracy: 0.9819 - val_loss: 0.1278 - val_accuracy: 0.9582
Epoch 57/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0503 -
accuracy: 0.9808 - val_loss: 0.1188 - val_accuracy: 0.9655
Epoch 58/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0439 -
accuracy: 0.9811 - val_loss: 0.1326 - val_accuracy: 0.9655
Epoch 59/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0416 -
accuracy: 0.9820 - val_loss: 0.1213 - val_accuracy: 0.9616
Epoch 60/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0414 -
accuracy: 0.9829 - val_loss: 0.1278 - val_accuracy: 0.9610
Epoch 61/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0424 -
accuracy: 0.9806 - val_loss: 0.1398 - val_accuracy: 0.9650
Epoch 62/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0466 -
accuracy: 0.9792 - val_loss: 0.1251 - val_accuracy: 0.9638
Epoch 63/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0415 -
accuracy: 0.9825 - val_loss: 0.1221 - val_accuracy: 0.9650
Epoch 64/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0413 -
accuracy: 0.9809 - val_loss: 0.1361 - val_accuracy: 0.9576
Epoch 65/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0472 -
accuracy: 0.9795 - val_loss: 0.1539 - val_accuracy: 0.9599
Epoch 66/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0610 -
```

```
accuracy: 0.9743 - val_loss: 0.1394 - val_accuracy: 0.9650
Epoch 67/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0434 -
accuracy: 0.9806 - val_loss: 0.1442 - val_accuracy: 0.9599
Epoch 68/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0384 -
accuracy: 0.9826 - val_loss: 0.1381 - val_accuracy: 0.9638
Epoch 69/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0429 -
accuracy: 0.9813 - val_loss: 0.1343 - val_accuracy: 0.9621
Epoch 70/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0388 -
accuracy: 0.9823 - val_loss: 0.1356 - val_accuracy: 0.9644
Epoch 71/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0455 -
accuracy: 0.9792 - val_loss: 0.1320 - val_accuracy: 0.9565
Epoch 72/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0442 -
accuracy: 0.9791 - val_loss: 0.1324 - val_accuracy: 0.9638
Epoch 73/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0489 -
accuracy: 0.9796 - val_loss: 0.1458 - val_accuracy: 0.9644
Epoch 74/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0481 -
accuracy: 0.9811 - val_loss: 0.1368 - val_accuracy: 0.9610
Epoch 75/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0403 -
accuracy: 0.9823 - val_loss: 0.1293 - val_accuracy: 0.9666
Epoch 76/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0370 -
accuracy: 0.9837 - val_loss: 0.1403 - val_accuracy: 0.9570
Epoch 77/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0391 -
accuracy: 0.9836 - val_loss: 0.1371 - val_accuracy: 0.9672
Epoch 78/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0357 -
accuracy: 0.9839 - val_loss: 0.1410 - val_accuracy: 0.9655
Epoch 79/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0366 -
accuracy: 0.9829 - val_loss: 0.1540 - val_accuracy: 0.9627
Epoch 80/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0402 -
accuracy: 0.9811 - val_loss: 0.1591 - val_accuracy: 0.9536
Epoch 81/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0412 -
accuracy: 0.9832 - val_loss: 0.1405 - val_accuracy: 0.9633
Epoch 82/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0442 -
```

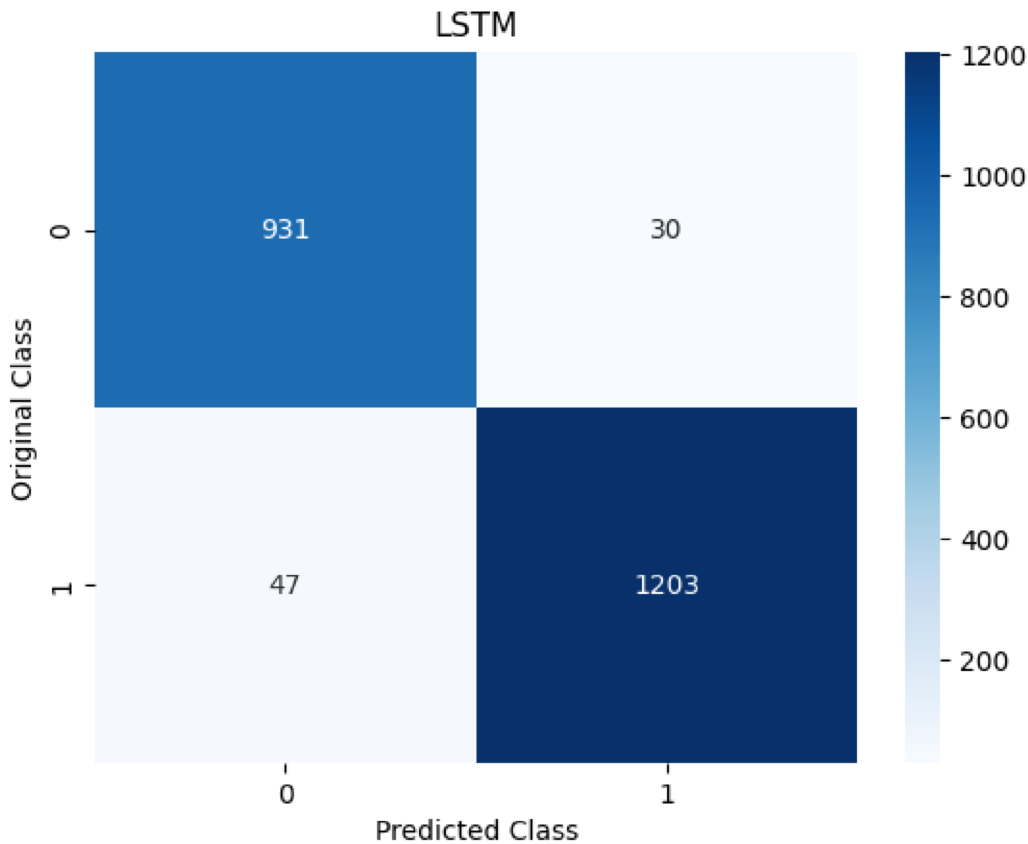
```
accuracy: 0.9808 - val_loss: 0.1490 - val_accuracy: 0.9661
Epoch 83/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0444 -
accuracy: 0.9812 - val_loss: 0.1567 - val_accuracy: 0.9655
Epoch 84/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0423 -
accuracy: 0.9820 - val_loss: 0.1450 - val_accuracy: 0.9638
Epoch 85/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0405 -
accuracy: 0.9832 - val_loss: 0.1568 - val_accuracy: 0.9616
Epoch 86/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0398 -
accuracy: 0.9813 - val_loss: 0.1421 - val_accuracy: 0.9650
Epoch 87/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0381 -
accuracy: 0.9825 - val_loss: 0.1500 - val_accuracy: 0.9638
Epoch 88/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0398 -
accuracy: 0.9832 - val_loss: 0.1650 - val_accuracy: 0.9650
Epoch 89/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0400 -
accuracy: 0.9823 - val_loss: 0.1682 - val_accuracy: 0.9633
Epoch 90/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0380 -
accuracy: 0.9840 - val_loss: 0.1534 - val_accuracy: 0.9638
Epoch 91/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0400 -
accuracy: 0.9826 - val_loss: 0.1521 - val_accuracy: 0.9655
Epoch 92/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0394 -
accuracy: 0.9833 - val_loss: 0.1313 - val_accuracy: 0.9604
Epoch 93/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0386 -
accuracy: 0.9823 - val_loss: 0.1475 - val_accuracy: 0.9638
Epoch 94/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0334 -
accuracy: 0.9860 - val_loss: 0.1698 - val_accuracy: 0.9661
Epoch 95/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0402 -
accuracy: 0.9832 - val_loss: 0.1484 - val_accuracy: 0.9604
Epoch 96/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0343 -
accuracy: 0.9845 - val_loss: 0.1466 - val_accuracy: 0.9587
Epoch 97/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0442 -
accuracy: 0.9813 - val_loss: 0.1205 - val_accuracy: 0.9582
Epoch 98/100
222/222 [==============================] - 1s 5ms/step - loss: 0.0394 -
```

```
accuracy: 0.9829 - val_loss: 0.1369 - val_accuracy: 0.9655
Epoch 99/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0419 -
accuracy: 0.9815 - val_loss: 0.1418 - val_accuracy: 0.9633
Epoch 100/100
222/222 [==============================] - 1s 4ms/step - loss: 0.0371 -
accuracy: 0.9835 - val_loss: 0.1332 - val_accuracy: 0.9616
Test results - Loss: 0.10944759845733643 - Accuracy: 96.517413854599%
```

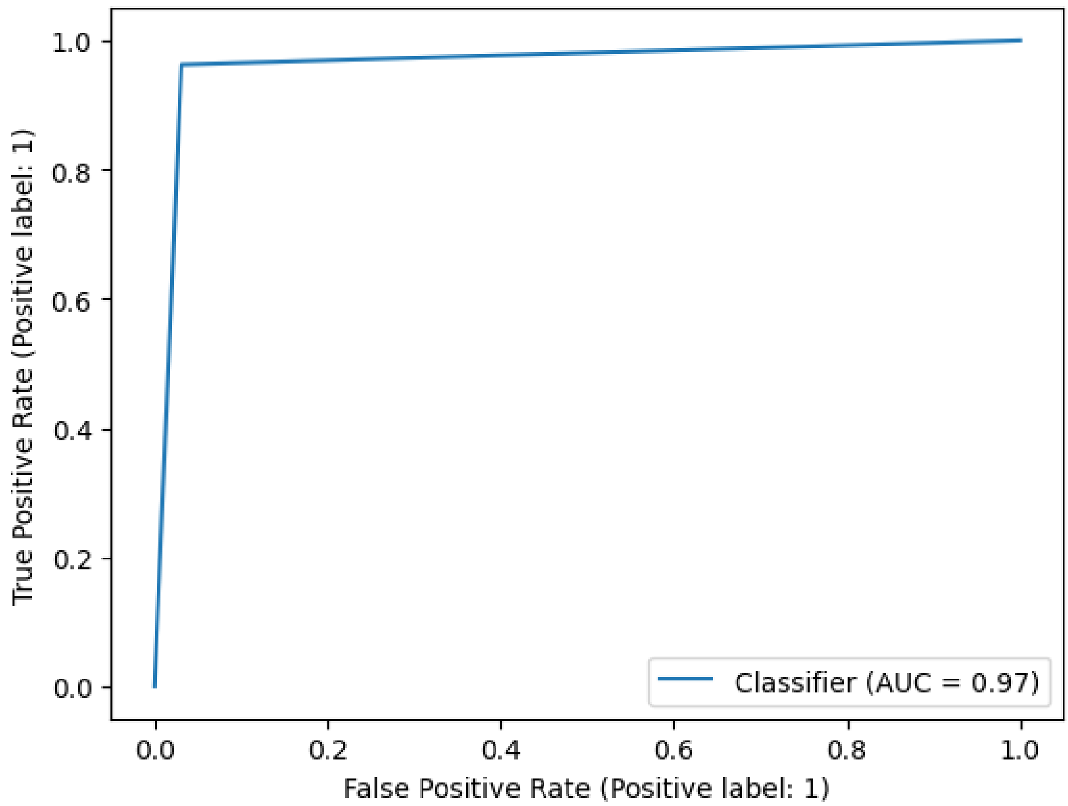
[ ]:
```
lstm_predict_proba = lstm_model.predict(X_test_reshape, batch_size=32)
lstm_predict_class = (lstm_predict_proba > 0.5).astype("int32")
print(classification_report(y_test, lstm_predict_class))
```

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

           0       0.95      0.97      0.96       961
           1       0.98      0.96      0.97      1250

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

[ ]:
```
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()
```



[ ]:
```
# print("Trade off between true positive rate and false positive rate")
# from sklearn.metrics import roc_curve
# fpr, tpr, _ = roc_curve(y_test, lstm_predict_class)
# plt.plot(fpr, tpr)
# plt.title('ROC curve')
# plt.xlabel('false positive rate')
# plt.ylabel('true positive rate')
# plt.xlim(0,)
# plt.ylim(0,)
# plt.show()
```

[ ]:
```
# from sklearn.metrics import roc_curve
# fpr, tpr, thresh = roc_curve(y_test, lstm_predict_class)
```

[ ]:
```
# # plot roc curves
# plt.plot(fpr, tpr, linestyle='--',color='orange', label='LSTM')

# # title
# plt.title('ROC curve')
# # x label
# plt.xlabel('False Positive Rate')
```

```
# # y label
# plt.ylabel('True Positive rate')

# plt.legend(loc='best')
# plt.savefig('ROC',dpi=300)
# plt.show()
```

[ ]:
```
# from keras.layers import Flatten
# model = Sequential([
#     Flatten(input_shape=(len(X_test.columns),)),
#     Dense(16, activation=tf.nn.relu),
#         Dense(16, activation=tf.nn.relu),
#     Dense(1, activation=tf.nn.sigmoid),
# ])

# model.compile(optimizer='adam',
#               loss='binary_crossentropy',
#               metrics=['accuracy'])

# model.fit(X_train, y_train, epochs=50, batch_size=1)

# test_loss, test_acc = model.evaluate(X_test, y_test)
# print('Test accuracy:', test_acc)
```

[ ]:
```
# model_pred = model.predict(X_test, batch_size=64)
# model_pred = (model_pred > 0.5).astype(int).reshape(-1,)
# print(classification_report(y_test, model_pred))
```

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

[ ]:
```
# tensorflow\python\keras\engine\sequential.py:455: UserWarning: model.
↪predict_classes() is deprecated and will be removed after 2021-01-01. Please␣
↪use instead:* np.argmax(model.predict(x), axis=-1), if your model does␣
↪multi-class classification (e.g. if it uses a softmax last-layer activation).
↪* (model.predict(x) > 0.5).astype("int32"), if your model does binary␣
↪classification (e.g. if it uses a sigmoid last-layer activation).
```