correlation_target _label_30 7030 spliit .02 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,.02)
     len(set(corr_feature))
```

```
[ ]: 5
```

```
[ ]: corr_feature
```

```
[ ]: {'Favicon', 'Iframe', '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)
```

```
[ ]: 26
```

```
[ ]: #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

[ ]:
```
# separate dataset into train and test
from cProfile import label
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=10)

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

[ ]:
```
((7738, 25), (3317, 25), (7738, 1), (3317, 1))
```

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

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

```
Training set has 7738 samples.
Testing set has 3317 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': 1, 'max_iter': 2500, 'penalty': 'l2', 'solver': 'lbfgs'}
LogisticRegression(C=1, max_iter=2500)
0.9286656237151139
```

```
[ ]: 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)
```

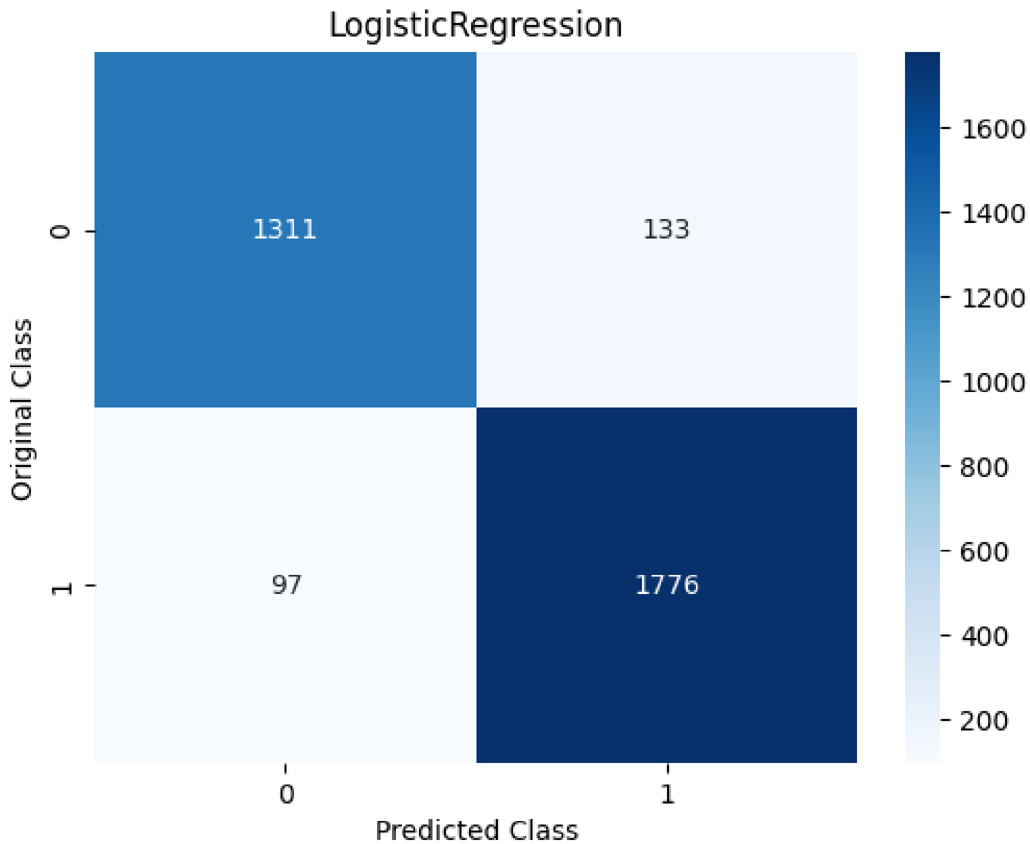
```
[ ]: print ("Accuracy of logr classifier : ", accuracy_score(y_test,␣
      ↪logr_predict)*100)
```

```
Accuracy of logr classifier :  93.0660235152246
```

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

```
              precision    recall  f1-score   support

           0       0.93      0.91      0.92      1444
           1       0.93      0.95      0.94      1873

    accuracy                           0.93      3317
   macro avg       0.93      0.93      0.93      3317
weighted avg       0.93      0.93      0.93      3317
```

```
[ ]: 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)
0.9590302221954798
```

[ ]:
```
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))
```

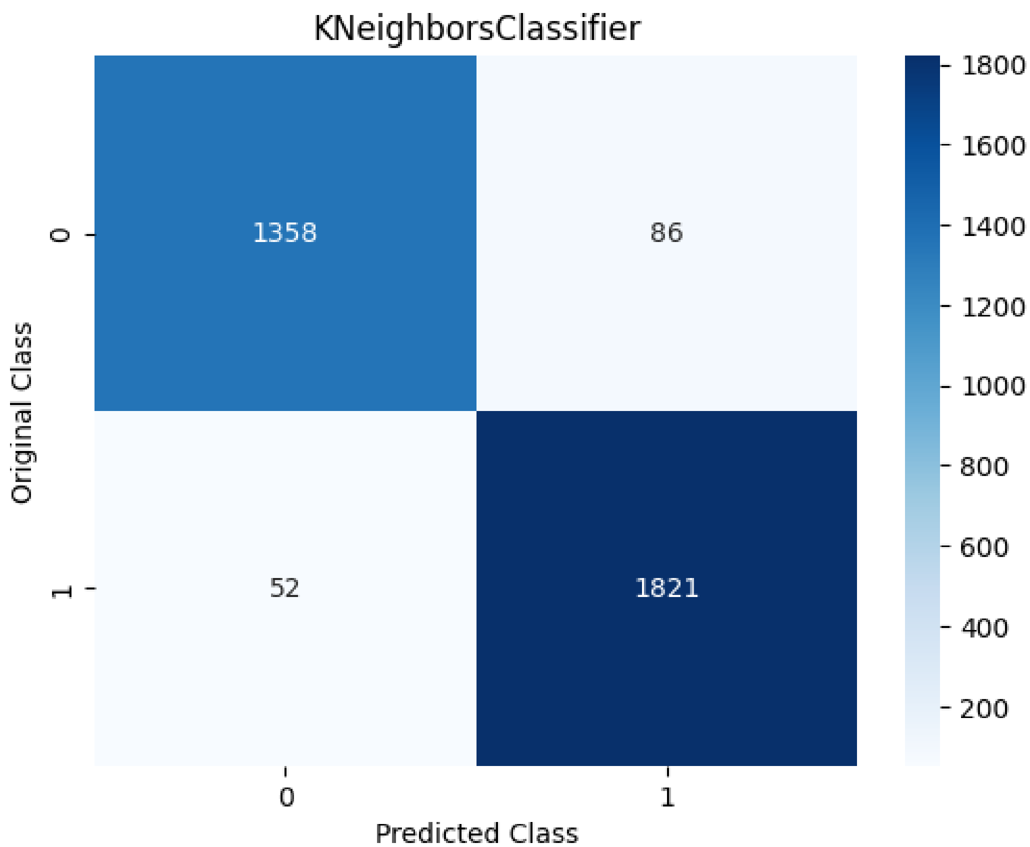
```
The accuracy of knn Classifier is:  95.83961410913476
```

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

```
              precision    recall  f1-score   support

           0       0.96      0.94      0.95      1444
           1       0.95      0.97      0.96      1873

    accuracy                           0.96      3317
   macro avg       0.96      0.96      0.96      3317
weighted avg       0.96      0.96      0.96      3317
```

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

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



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

# RocCurveDisplay.from_predictions(y_test,knn_y_pred_proba_positive)

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

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

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

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

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

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

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

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

```
Fitting 10 folds for each of 36 candidates, totalling 360 fits
{'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
SVC(C=10, gamma=0.1)
0.9639441285504645
```

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

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

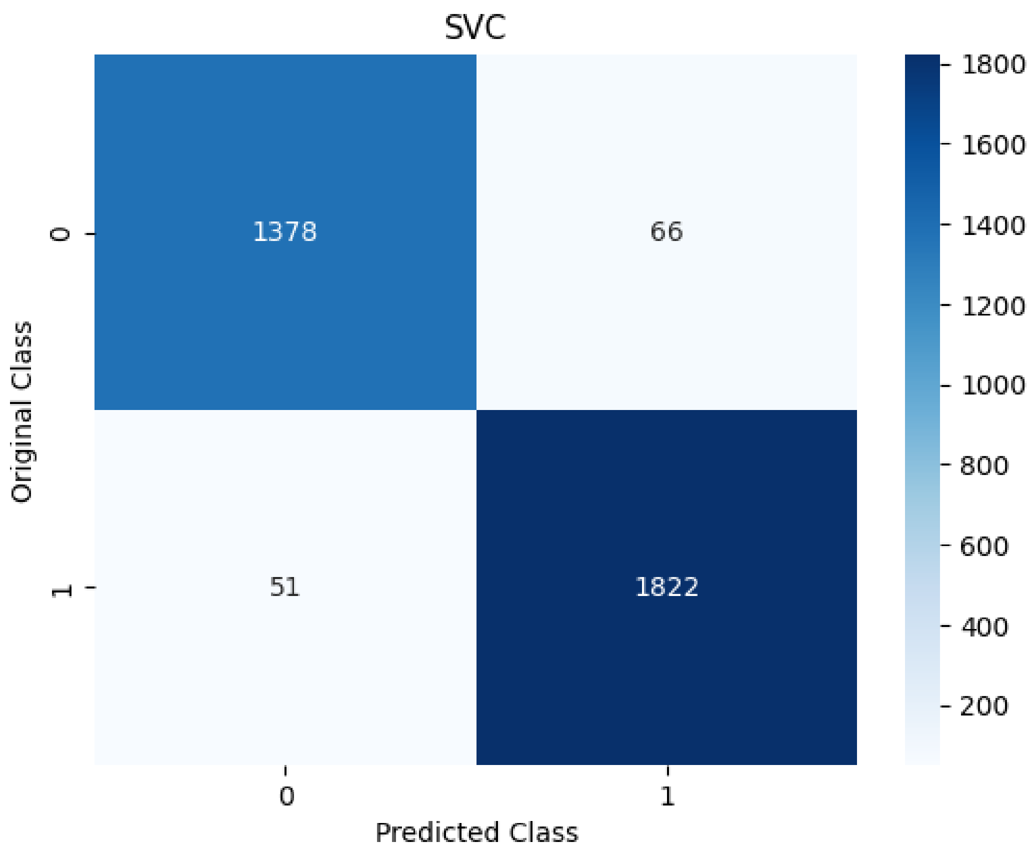
```
[ ]: print('The accuracy of svc Classifier is: ', 100.0 * accuracy_score(y_test,␣
      ↪svc_predict))
```

```
The accuracy of svc Classifier is:  96.47271630991861
```

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

```
              precision    recall  f1-score   support

           0       0.96      0.95      0.96      1444
           1       0.97      0.97      0.97      1873

    accuracy                           0.96      3317
   macro avg       0.96      0.96      0.96      3317
weighted avg       0.96      0.96      0.96      3317
```

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



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

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

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

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

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

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

```
Fitting 10 folds for each of 24 candidates, totalling 240 fits
{'gamma': 0.1, 'kernel': 'rbf', 'nu': 0.1}
NuSVC(gamma=0.1, nu=0.1)
0.9627811707131182
```

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

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

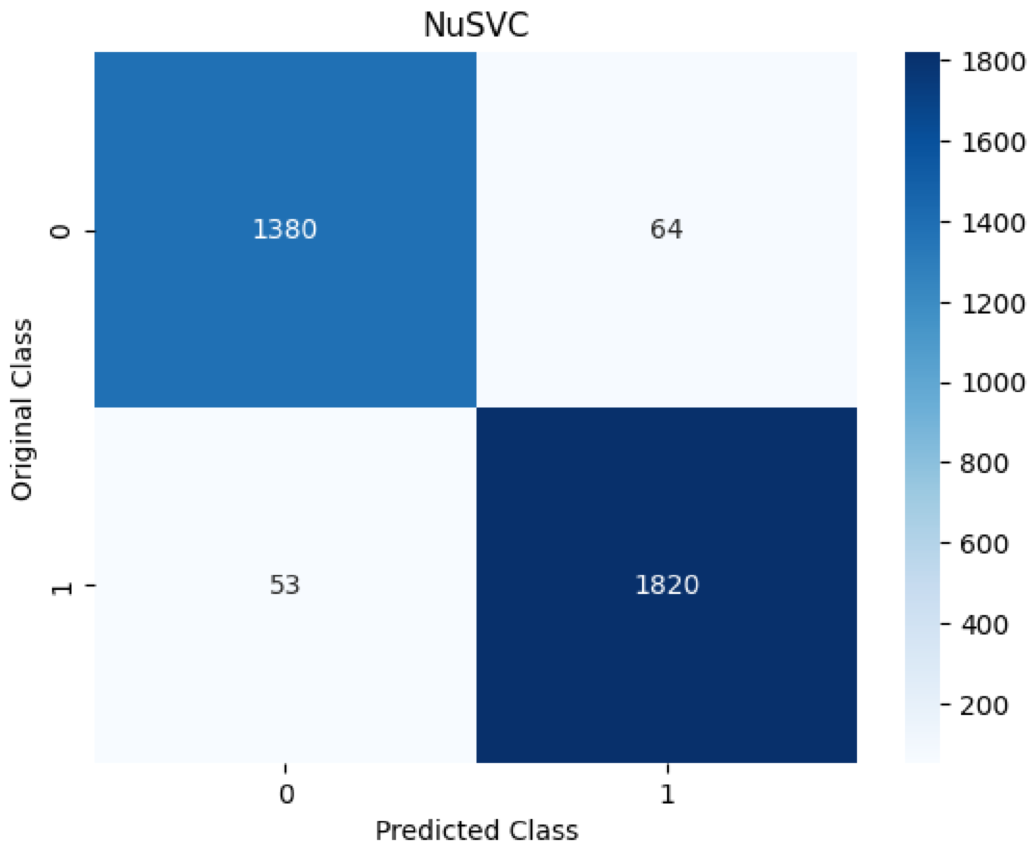
[ ]:
```
print('The accuracy of nusvc Classifier is: ', 100.0 * accuracy_score(y_test,␣
 ↪nusvc_predict))
```

```
The accuracy of nusvc Classifier is:  96.47271630991861
```

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

```
              precision    recall  f1-score   support

           0       0.96      0.96      0.96      1444
           1       0.97      0.97      0.97      1873

    accuracy                           0.96      3317
   macro avg       0.96      0.96      0.96      3317
weighted avg       0.96      0.96      0.96      3317
```

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



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

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

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

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

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

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

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

```
Fitting 10 folds for each of 30 candidates, totalling 300 fits
{'C': 10, 'dual': False, 'loss': 'squared_hinge', 'penalty': 'l2', 'tol': 0.001}
LinearSVC(C=10, dual=False, tol=0.001)
0.9285367590280493
```

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

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

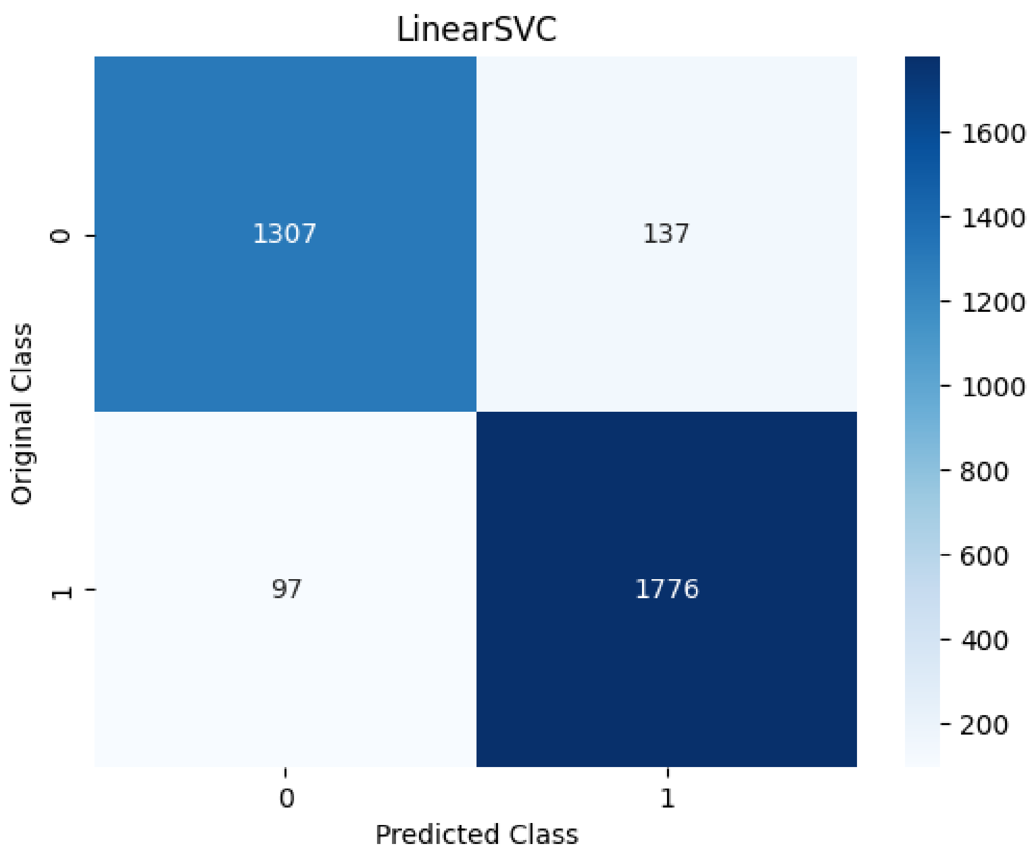
[ ]:
```
print('The accuracy of lsvc Classifier is: ', 100.0 * accuracy_score(y_test,␣
 ↪lsvc_predict))
```

```
The accuracy of lsvc Classifier is:  92.94543261983719
```

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

```
              precision    recall  f1-score   support

           0       0.93      0.91      0.92      1444
           1       0.93      0.95      0.94      1873

    accuracy                           0.93      3317
   macro avg       0.93      0.93      0.93      3317
weighted avg       0.93      0.93      0.93      3317
```

[ ]:
```
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)
0.9369356946826184
```

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

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

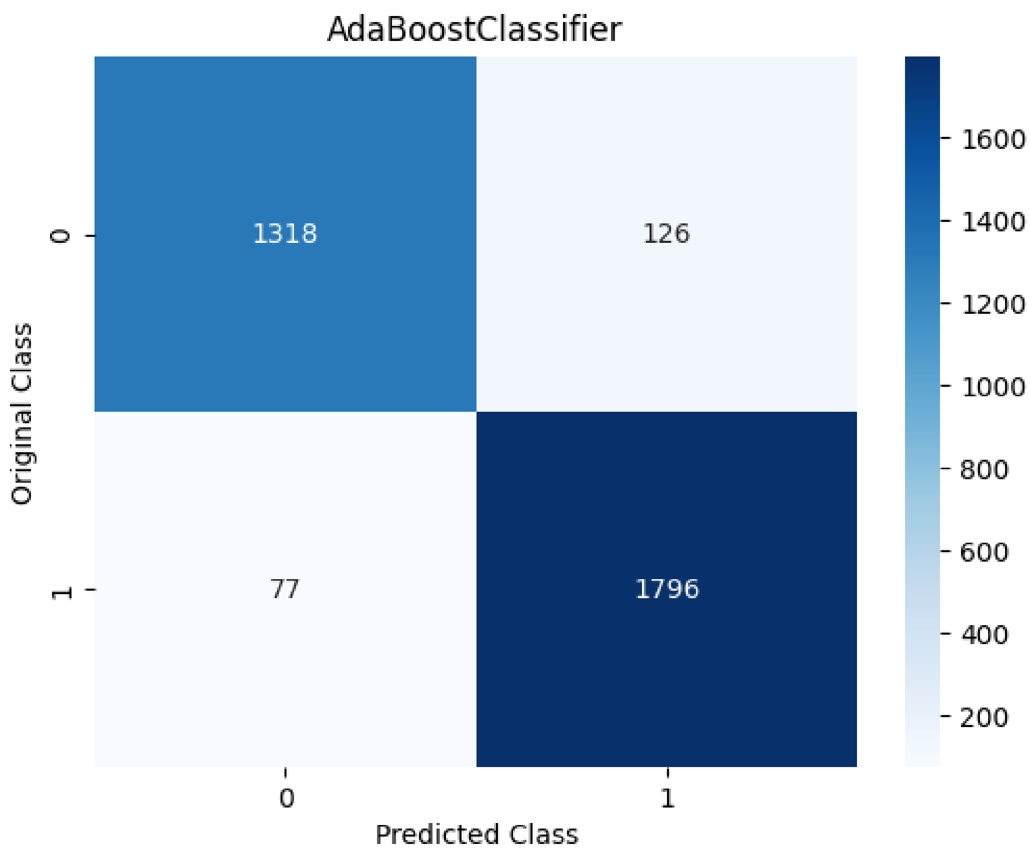
[ ]:
```python
print('The accuracy of Ada Boost Classifier is: ', 100.0 *␣
  ↪accuracy_score(ada_predict,y_test))
```

```
The accuracy of Ada Boost Classifier is:  93.88001205908954
```

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

```
              precision    recall  f1-score   support

           0       0.94      0.91      0.93      1444
           1       0.93      0.96      0.95      1873

    accuracy                           0.94      3317
   macro avg       0.94      0.94      0.94      3317
weighted avg       0.94      0.94      0.94      3317
```

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




```
[ ]: from xgboost import XGBClassifier


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

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

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

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

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

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

```
Fitting 10 folds for each of 15 candidates, totalling 150 fits
{'gamma': 0.1, 'n_estimators': 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=0.1, gpu_id=-1, grow_policy='depthwise',
              importance_type=None, interaction_constraints='',
              learning_rate=0.300000012, max_bin=256, max_cat_to_onehot=4,
              max_delta_step=0, max_depth=6, max_leaves=0, min_child_weight=1,
              missing=nan, monotone_constraints='()', n_estimators=250,
              n_jobs=0, num_parallel_tree=1, predictor='auto', random_state=0,
              reg_alpha=0, reg_lambda=1, …)
0.9706616391053349
```

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

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

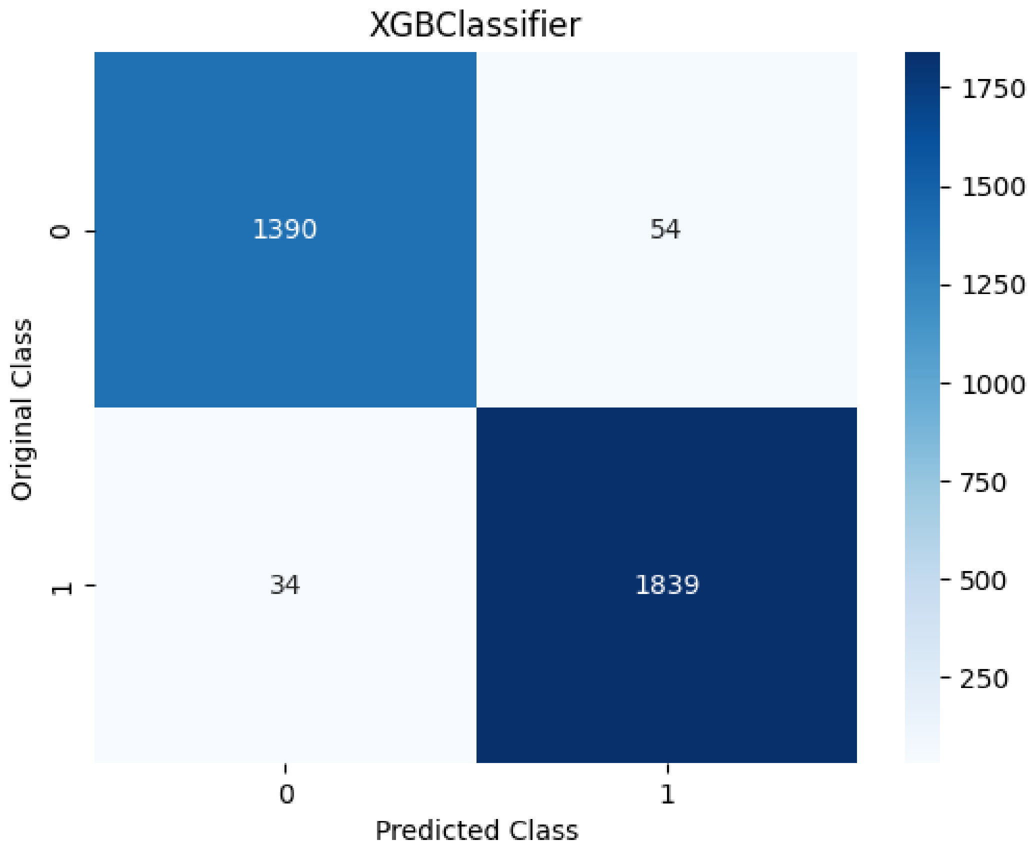
[ ]:
```
print('The accuracy of XGBoost Classifier is: ' , 100.0 *␣
 ↪accuracy_score(xgb_predict,y_test))
```

```
The accuracy of XGBoost Classifier is:  97.34700030147724
```

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

```
              precision    recall  f1-score   support

           0       0.98      0.96      0.97      1444
           1       0.97      0.98      0.98      1873

    accuracy                           0.97      3317
   macro avg       0.97      0.97      0.97      3317
weighted avg       0.97      0.97      0.97      3317
```

[ ]:
```
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': 150}
GradientBoostingClassifier(learning_rate=1, n_estimators=150)
0.9671725984536238
```

[ ]:
```
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)
```

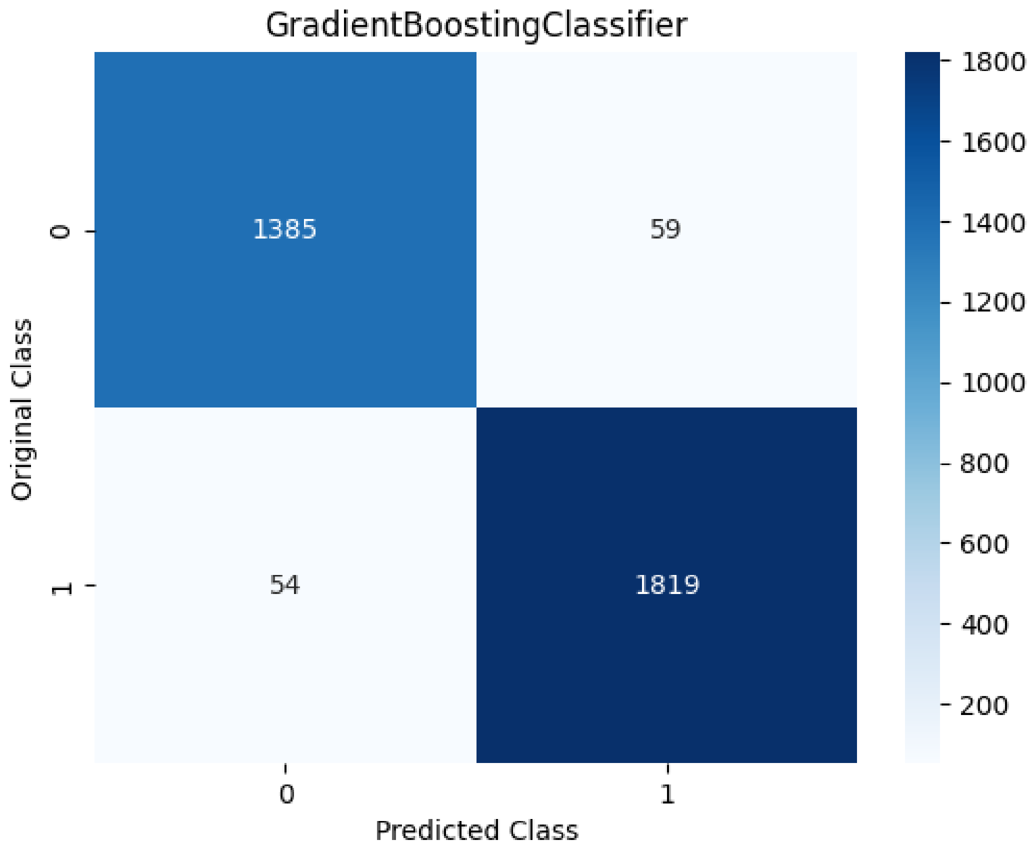
[ ]:
```
print('The accuracy of GradientBoost Classifier is: ' , 100.0 *␣
↪accuracy_score(gbc_predict,y_test))
```

```
The accuracy of GradientBoost Classifier is:  96.593307205306
```

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

```
              precision    recall  f1-score   support

           0       0.96      0.96      0.96      1444
           1       0.97      0.97      0.97      1873

    accuracy                           0.97      3317
   macro avg       0.97      0.97      0.97      3317
weighted avg       0.97      0.97      0.97      3317
```

[ ]:
```
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)
0.9652341125384838
```

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

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

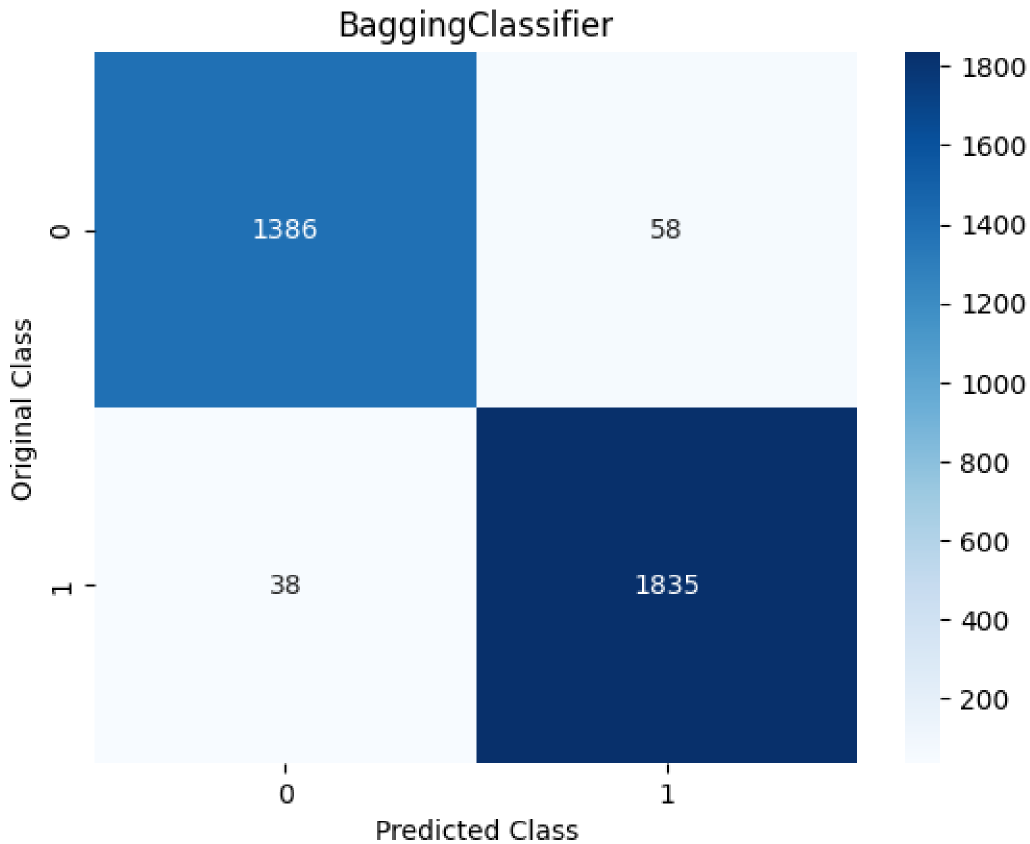
[ ]:
```
print('The accuracy of Bagging Classifier is: ' , 100.0 *␣
↪accuracy_score(y_test, bag_predict))
```

```
The accuracy of Bagging Classifier is:  97.10581851070245
```

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

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.97      1444
           1       0.97      0.98      0.97      1873

    accuracy                           0.97      3317
   macro avg       0.97      0.97      0.97      3317
weighted avg       0.97      0.97      0.97      3317
```

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



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

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

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

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

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

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

```
Fitting 10 folds for each of 5 candidates, totalling 50 fits
{'n_estimators': 250}
RandomForestClassifier(n_estimators=250)
0.9685939542237867
```

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

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

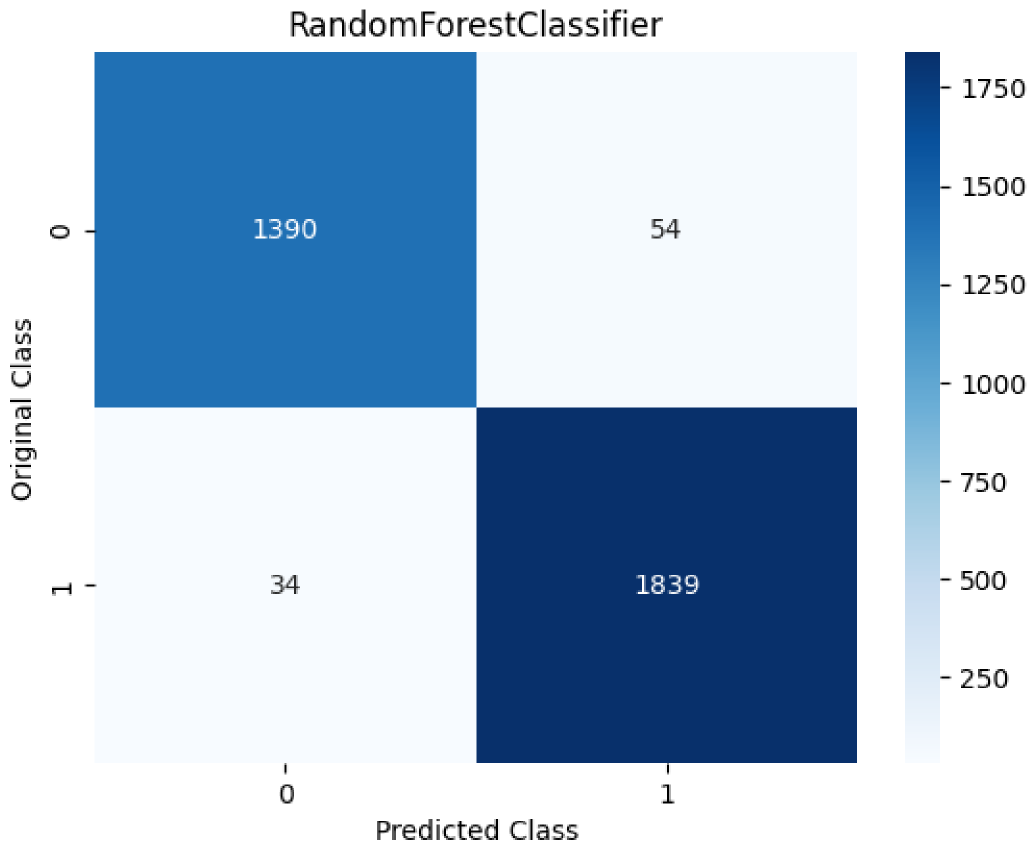
[ ]:
```
print('The accuracy of RandomForest Classifier is: ' , 100.0 *␣
 ↪accuracy_score(rfc_predict,y_test))
```

```
The accuracy of RandomForest Classifier is:  97.34700030147724
```

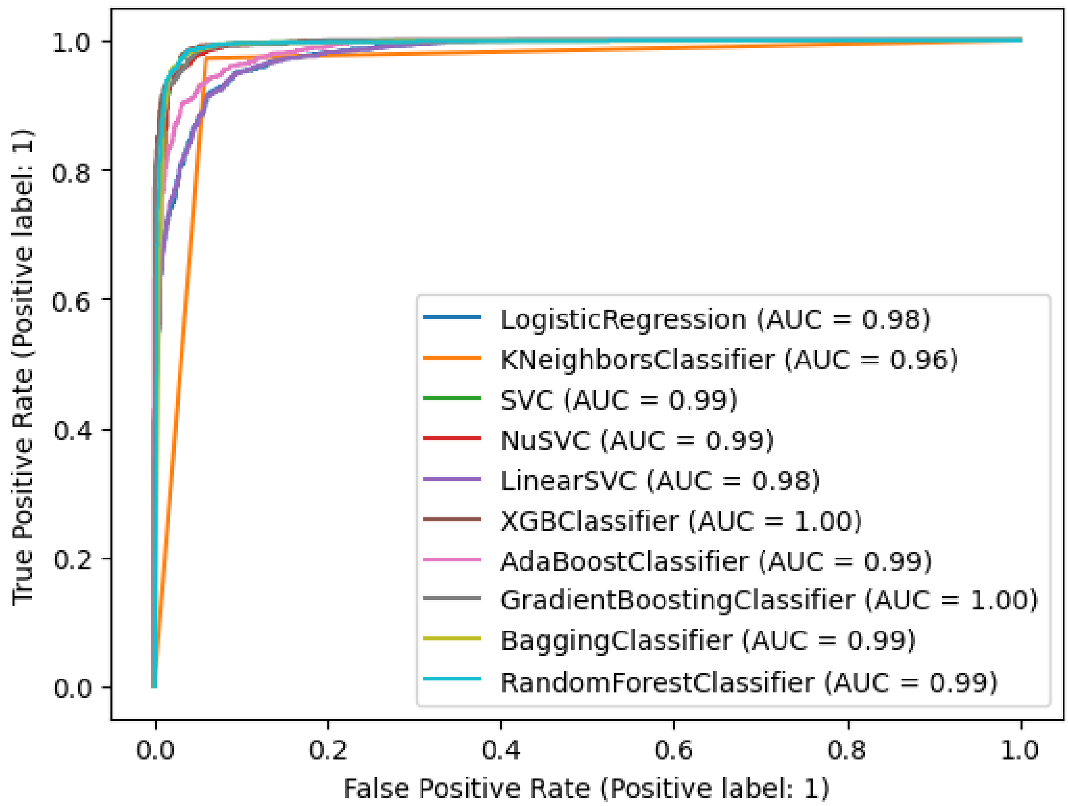
[ ]:
```
print(classification_report(y_test, rfc_predict))
```

```
              precision    recall  f1-score   support

           0       0.98      0.96      0.97      1444
           1       0.97      0.98      0.98      1873

    accuracy                           0.97      3317
   macro avg       0.97      0.97      0.97      3317
weighted avg       0.97      0.97      0.97      3317
```

[ ]:
```
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
```

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

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

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_9 (LSTM)               (None, 100)               50400

 batch_normalization_9 (Batc  (None, 100)              400
 hNormalization)

 dense_36 (Dense)            (None, 50)                5050

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_9 (LSTM)               (None, 100)               50400

 batch_normalization_9 (Batc  (None, 100)              400
 hNormalization)

 dense_36 (Dense)            (None, 50)                5050

 dense_37 (Dense)            (None, 25)                1275

 dense_38 (Dense)            (None, 10)                260

 dense_39 (Dense)            (None, 1)                 11

=================================================================
Total params: 57,396
Trainable params: 57,196
Non-trainable params: 200
_________________________________________________________________
Epoch 1/100
194/194 [==============================] - 4s 7ms/step - loss: 0.2365 -
accuracy: 0.9039 - val_loss: 0.4272 - val_accuracy: 0.8766
Epoch 2/100
```

```
194/194 [==============================] - 1s 4ms/step - loss: 0.1672 -
accuracy: 0.9312 - val_loss: 0.2204 - val_accuracy: 0.9360
Epoch 3/100
194/194 [==============================] - 1s 4ms/step - loss: 0.1440 -
accuracy: 0.9375 - val_loss: 0.1433 - val_accuracy: 0.9490
Epoch 4/100
194/194 [==============================] - 1s 4ms/step - loss: 0.1364 -
accuracy: 0.9425 - val_loss: 0.1248 - val_accuracy: 0.9516
Epoch 5/100
194/194 [==============================] - 1s 4ms/step - loss: 0.1269 -
accuracy: 0.9446 - val_loss: 0.1192 - val_accuracy: 0.9432
Epoch 6/100
194/194 [==============================] - 1s 4ms/step - loss: 0.1211 -
accuracy: 0.9491 - val_loss: 0.1283 - val_accuracy: 0.9483
Epoch 7/100
194/194 [==============================] - 1s 4ms/step - loss: 0.1088 -
accuracy: 0.9536 - val_loss: 0.1304 - val_accuracy: 0.9490
Epoch 8/100
194/194 [==============================] - 1s 5ms/step - loss: 0.1073 -
accuracy: 0.9546 - val_loss: 0.1217 - val_accuracy: 0.9528
Epoch 9/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0966 -
accuracy: 0.9603 - val_loss: 0.1258 - val_accuracy: 0.9535
Epoch 10/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0966 -
accuracy: 0.9590 - val_loss: 0.1415 - val_accuracy: 0.9464
Epoch 11/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0949 -
accuracy: 0.9606 - val_loss: 0.1463 - val_accuracy: 0.9444
Epoch 12/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0886 -
accuracy: 0.9606 - val_loss: 0.1087 - val_accuracy: 0.9535
Epoch 13/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0829 -
accuracy: 0.9641 - val_loss: 0.1131 - val_accuracy: 0.9516
Epoch 14/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0746 -
accuracy: 0.9687 - val_loss: 0.1092 - val_accuracy: 0.9548
Epoch 15/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0827 -
accuracy: 0.9674 - val_loss: 0.1152 - val_accuracy: 0.9503
Epoch 16/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0758 -
accuracy: 0.9691 - val_loss: 0.1261 - val_accuracy: 0.9554
Epoch 17/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0723 -
accuracy: 0.9724 - val_loss: 0.1229 - val_accuracy: 0.9522
Epoch 18/100
```

```
194/194 [==============================] - 1s 5ms/step - loss: 0.0721 -
accuracy: 0.9691 - val_loss: 0.0975 - val_accuracy: 0.9599
Epoch 19/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0616 -
accuracy: 0.9745 - val_loss: 0.1474 - val_accuracy: 0.9373
Epoch 20/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0686 -
accuracy: 0.9714 - val_loss: 0.1269 - val_accuracy: 0.9593
Epoch 21/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0633 -
accuracy: 0.9738 - val_loss: 0.1258 - val_accuracy: 0.9522
Epoch 22/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0652 -
accuracy: 0.9738 - val_loss: 0.1253 - val_accuracy: 0.9567
Epoch 23/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0590 -
accuracy: 0.9751 - val_loss: 0.1324 - val_accuracy: 0.9561
Epoch 24/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0663 -
accuracy: 0.9722 - val_loss: 0.1056 - val_accuracy: 0.9612
Epoch 25/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0591 -
accuracy: 0.9756 - val_loss: 0.1054 - val_accuracy: 0.9651
Epoch 26/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0610 -
accuracy: 0.9751 - val_loss: 0.1064 - val_accuracy: 0.9587
Epoch 27/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0598 -
accuracy: 0.9742 - val_loss: 0.1224 - val_accuracy: 0.9567
Epoch 28/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0542 -
accuracy: 0.9774 - val_loss: 0.1226 - val_accuracy: 0.9535
Epoch 29/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0559 -
accuracy: 0.9764 - val_loss: 0.1234 - val_accuracy: 0.9574
Epoch 30/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0503 -
accuracy: 0.9805 - val_loss: 0.1094 - val_accuracy: 0.9638
Epoch 31/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0503 -
accuracy: 0.9785 - val_loss: 0.1284 - val_accuracy: 0.9528
Epoch 32/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0501 -
accuracy: 0.9774 - val_loss: 0.1178 - val_accuracy: 0.9599
Epoch 33/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0507 -
accuracy: 0.9805 - val_loss: 0.1389 - val_accuracy: 0.9612
Epoch 34/100
```

```
194/194 [==============================] - 1s 5ms/step - loss: 0.0529 -
accuracy: 0.9761 - val_loss: 0.1266 - val_accuracy: 0.9587
Epoch 35/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0514 -
accuracy: 0.9780 - val_loss: 0.1275 - val_accuracy: 0.9587
Epoch 36/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0480 -
accuracy: 0.9813 - val_loss: 0.1372 - val_accuracy: 0.9593
Epoch 37/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0436 -
accuracy: 0.9795 - val_loss: 0.1508 - val_accuracy: 0.9561
Epoch 38/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0508 -
accuracy: 0.9780 - val_loss: 0.1128 - val_accuracy: 0.9599
Epoch 39/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0499 -
accuracy: 0.9806 - val_loss: 0.1536 - val_accuracy: 0.9567
Epoch 40/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0474 -
accuracy: 0.9817 - val_loss: 0.1311 - val_accuracy: 0.9567
Epoch 41/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0410 -
accuracy: 0.9847 - val_loss: 0.1426 - val_accuracy: 0.9599
Epoch 42/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0459 -
accuracy: 0.9817 - val_loss: 0.1222 - val_accuracy: 0.9599
Epoch 43/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0400 -
accuracy: 0.9838 - val_loss: 0.1475 - val_accuracy: 0.9496
Epoch 44/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0426 -
accuracy: 0.9813 - val_loss: 0.1377 - val_accuracy: 0.9567
Epoch 45/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0446 -
accuracy: 0.9822 - val_loss: 0.1171 - val_accuracy: 0.9625
Epoch 46/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0389 -
accuracy: 0.9830 - val_loss: 0.1396 - val_accuracy: 0.9574
Epoch 47/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0365 -
accuracy: 0.9838 - val_loss: 0.1365 - val_accuracy: 0.9632
Epoch 48/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0404 -
accuracy: 0.9821 - val_loss: 0.1366 - val_accuracy: 0.9606
Epoch 49/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0378 -
accuracy: 0.9835 - val_loss: 0.1461 - val_accuracy: 0.9561
Epoch 50/100
```

```
194/194 [==============================] - 1s 4ms/step - loss: 0.0477 -
accuracy: 0.9811 - val_loss: 0.1344 - val_accuracy: 0.9561
Epoch 51/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0384 -
accuracy: 0.9840 - val_loss: 0.1383 - val_accuracy: 0.9632
Epoch 52/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0390 -
accuracy: 0.9822 - val_loss: 0.1977 - val_accuracy: 0.9580
Epoch 53/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0425 -
accuracy: 0.9834 - val_loss: 0.1350 - val_accuracy: 0.9574
Epoch 54/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0388 -
accuracy: 0.9821 - val_loss: 0.1536 - val_accuracy: 0.9580
Epoch 55/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0396 -
accuracy: 0.9834 - val_loss: 0.1405 - val_accuracy: 0.9587
Epoch 56/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0410 -
accuracy: 0.9811 - val_loss: 0.1372 - val_accuracy: 0.9599
Epoch 57/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0416 -
accuracy: 0.9801 - val_loss: 0.1592 - val_accuracy: 0.9541
Epoch 58/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0360 -
accuracy: 0.9853 - val_loss: 0.1306 - val_accuracy: 0.9651
Epoch 59/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0368 -
accuracy: 0.9850 - val_loss: 0.1342 - val_accuracy: 0.9625
Epoch 60/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0355 -
accuracy: 0.9843 - val_loss: 0.1365 - val_accuracy: 0.9619
Epoch 61/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0367 -
accuracy: 0.9850 - val_loss: 0.1549 - val_accuracy: 0.9593
Epoch 62/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0365 -
accuracy: 0.9827 - val_loss: 0.1575 - val_accuracy: 0.9548
Epoch 63/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0346 -
accuracy: 0.9869 - val_loss: 0.1386 - val_accuracy: 0.9587
Epoch 64/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0343 -
accuracy: 0.9850 - val_loss: 0.1653 - val_accuracy: 0.9561
Epoch 65/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0369 -
accuracy: 0.9851 - val_loss: 0.1484 - val_accuracy: 0.9574
Epoch 66/100
```

```
194/194 [==============================] - 1s 4ms/step - loss: 0.0395 -
accuracy: 0.9838 - val_loss: 0.1577 - val_accuracy: 0.9606
Epoch 67/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0333 -
accuracy: 0.9848 - val_loss: 0.1290 - val_accuracy: 0.9632
Epoch 68/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0332 -
accuracy: 0.9859 - val_loss: 0.1462 - val_accuracy: 0.9599
Epoch 69/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0332 -
accuracy: 0.9853 - val_loss: 0.1580 - val_accuracy: 0.9574
Epoch 70/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0329 -
accuracy: 0.9845 - val_loss: 0.1715 - val_accuracy: 0.9541
Epoch 71/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0331 -
accuracy: 0.9853 - val_loss: 0.1599 - val_accuracy: 0.9632
Epoch 72/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0363 -
accuracy: 0.9840 - val_loss: 0.1460 - val_accuracy: 0.9625
Epoch 73/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0326 -
accuracy: 0.9864 - val_loss: 0.1688 - val_accuracy: 0.9574
Epoch 74/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0347 -
accuracy: 0.9840 - val_loss: 0.1681 - val_accuracy: 0.9606
Epoch 75/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0358 -
accuracy: 0.9847 - val_loss: 0.1347 - val_accuracy: 0.9593
Epoch 76/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0316 -
accuracy: 0.9869 - val_loss: 0.1742 - val_accuracy: 0.9509
Epoch 77/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0354 -
accuracy: 0.9843 - val_loss: 0.1409 - val_accuracy: 0.9658
Epoch 78/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0414 -
accuracy: 0.9834 - val_loss: 0.1666 - val_accuracy: 0.9599
Epoch 79/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0327 -
accuracy: 0.9858 - val_loss: 0.1476 - val_accuracy: 0.9625
Epoch 80/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0303 -
accuracy: 0.9868 - val_loss: 0.1625 - val_accuracy: 0.9587
Epoch 81/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0294 -
accuracy: 0.9869 - val_loss: 0.1450 - val_accuracy: 0.9612
Epoch 82/100
```

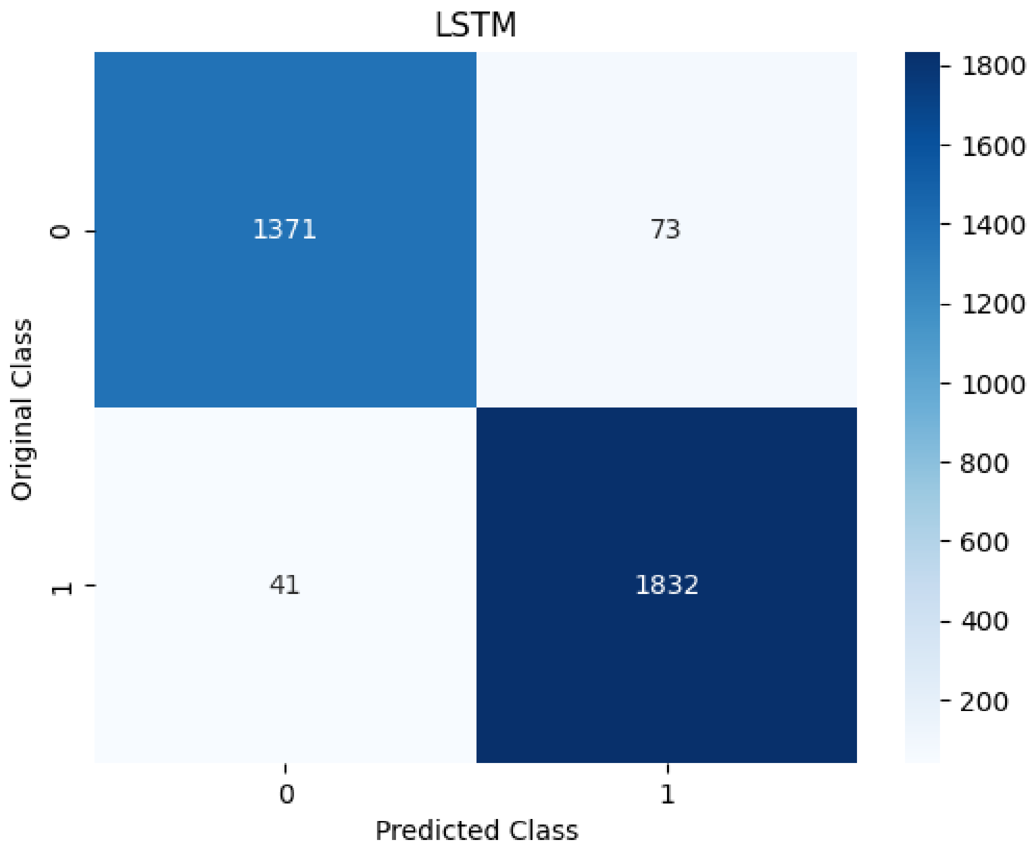
```
194/194 [==============================] - 1s 4ms/step - loss: 0.0309 -
accuracy: 0.9876 - val_loss: 0.1867 - val_accuracy: 0.9599
Epoch 83/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0339 -
accuracy: 0.9845 - val_loss: 0.1659 - val_accuracy: 0.9606
Epoch 84/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0323 -
accuracy: 0.9872 - val_loss: 0.1697 - val_accuracy: 0.9612
Epoch 85/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0291 -
accuracy: 0.9872 - val_loss: 0.1647 - val_accuracy: 0.9593
Epoch 86/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0330 -
accuracy: 0.9858 - val_loss: 0.1592 - val_accuracy: 0.9612
Epoch 87/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0330 -
accuracy: 0.9851 - val_loss: 0.1510 - val_accuracy: 0.9606
Epoch 88/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0416 -
accuracy: 0.9832 - val_loss: 0.1763 - val_accuracy: 0.9599
Epoch 89/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0343 -
accuracy: 0.9847 - val_loss: 0.1455 - val_accuracy: 0.9625
Epoch 90/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0291 -
accuracy: 0.9864 - val_loss: 0.1667 - val_accuracy: 0.9632
Epoch 91/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0280 -
accuracy: 0.9861 - val_loss: 0.1655 - val_accuracy: 0.9587
Epoch 92/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0295 -
accuracy: 0.9866 - val_loss: 0.1694 - val_accuracy: 0.9574
Epoch 93/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0307 -
accuracy: 0.9871 - val_loss: 0.1576 - val_accuracy: 0.9574
Epoch 94/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0290 -
accuracy: 0.9871 - val_loss: 0.1652 - val_accuracy: 0.9587
Epoch 95/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0316 -
accuracy: 0.9864 - val_loss: 0.1628 - val_accuracy: 0.9625
Epoch 96/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0343 -
accuracy: 0.9847 - val_loss: 0.1409 - val_accuracy: 0.9619
Epoch 97/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0327 -
accuracy: 0.9853 - val_loss: 0.1536 - val_accuracy: 0.9619
Epoch 98/100
```

```
194/194 [==============================] - 1s 4ms/step - loss: 0.0292 -
accuracy: 0.9859 - val_loss: 0.1656 - val_accuracy: 0.9606
Epoch 99/100
194/194 [==============================] - 1s 4ms/step - loss: 0.0326 -
accuracy: 0.9864 - val_loss: 0.1619 - val_accuracy: 0.9638
Epoch 100/100
194/194 [==============================] - 1s 5ms/step - loss: 0.0285 -
accuracy: 0.9871 - val_loss: 0.1705 - val_accuracy: 0.9580
Test results - Loss: 0.12258728593587875 - Accuracy: 96.56316041946411%
```

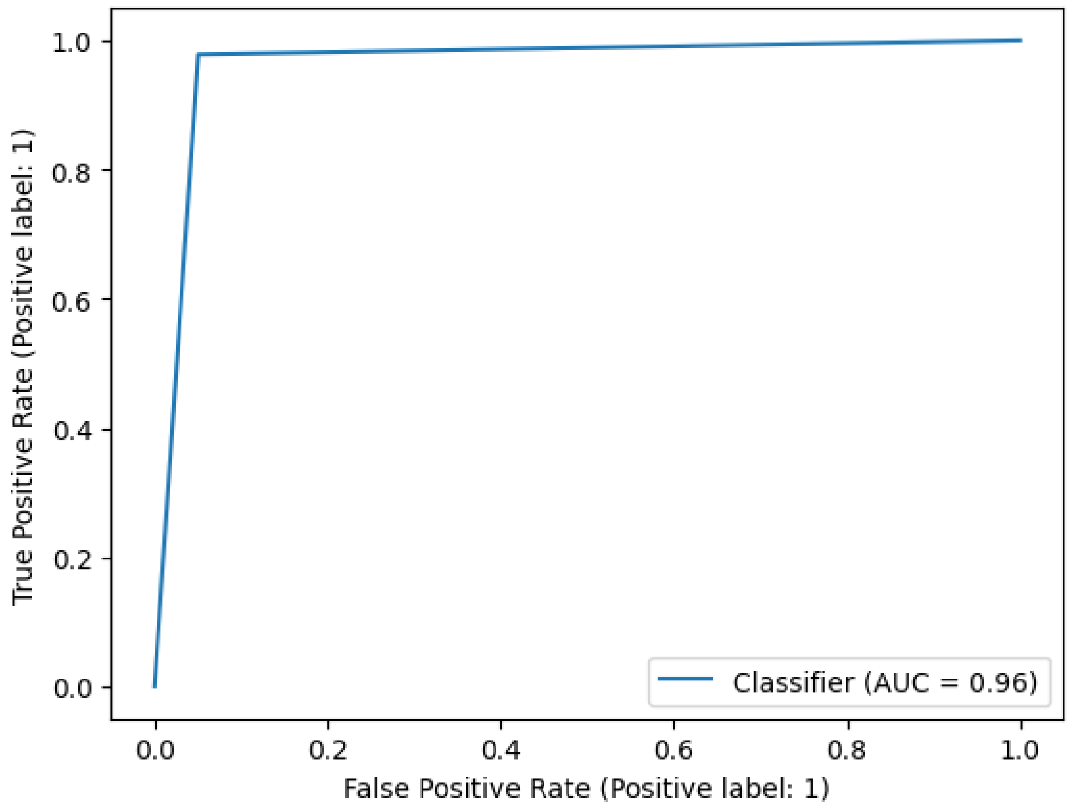
[ ]:
```
lstm_predict_proba = lstm_model.predict(X_test_reshape, batch_size=32)
lstm_predict_class = (lstm_predict_proba > 0.5).astype("int32")
print(classification_report(y_test, lstm_predict_class))
```

```
104/104 [==============================] - 1s 2ms/step
              precision    recall  f1-score   support

           0       0.97      0.95      0.96      1444
           1       0.96      0.98      0.97      1873

    accuracy                           0.97      3317
   macro avg       0.97      0.96      0.96      3317
weighted avg       0.97      0.97      0.97      3317
```

[ ]:
```
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).
```