

correlation_target_label_87 7030 split .02 threshold

January 3, 2023

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

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

[ ]: # Importing the dataset
df = pd.read_csv('dataset_phishing.csv')
df.drop(['url'], axis=1, inplace=True)
#df.head(50)

[ ]: # if your dataset contains missing value, check which column has missing values
#df.isnull().sum()

[ ]: #df.dropna(inplace=True)

[ ]: from sklearn import preprocessing

col = [df.columns[-1]]

lab_en= preprocessing.LabelEncoder()

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

#df.head(50)
```

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

```
[ ]:
```

	Col	Correlation
0	status	1.000000e+00
1	google_index	7.311708e-01
2	page_rank	5.111371e-01
3	nb_www	4.434677e-01
4	ratio_digits_url	3.563946e-01
5	domain_in_title	3.428070e-01
6	nb_hyperlinks	3.426283e-01
7	phish_hints	3.353927e-01
8	domain_age	3.318891e-01
9	ip	3.216978e-01
10	nb_qm	2.943191e-01
11	length_url	2.485805e-01
12	ratio_intHyperlinks	2.439821e-01
13	nb_slash	2.422700e-01
14	length_hostname	2.383224e-01
15	nb_eq	2.333863e-01
16	ratio_digits_host	2.243349e-01
17	shortest_word_host	2.230840e-01
18	prefix_suffix	2.146807e-01
19	longest_word_path	2.127091e-01
20	tld_in_subdomain	2.088842e-01
21	empty_title	2.070428e-01
22	nb_dots	2.070288e-01
23	longest_words_raw	2.001466e-01
24	avg_word_path	1.972561e-01
25	avg_word_host	1.935017e-01
26	ratio_intMedia	1.933331e-01
27	length_words_raw	1.920105e-01
28	links_in_tags	1.844011e-01
29	safe_anchor	1.733973e-01
30	domain_with_copyright	1.730985e-01
31	nb_and	1.705464e-01
32	avg_words_raw	1.675637e-01
33	domain_registration_length	1.617188e-01
34	nb_com	1.562835e-01
35	ratio_extRedirection	1.508267e-01
36	external_favicon	1.465654e-01
37	statistical_report	1.439435e-01
38	nb_at	1.429146e-01
39	ratio_extMedia	1.404059e-01

40	abnormal_subdomain	1.281598e-01
41	longest_word_host	1.245156e-01
42	dns_record	1.221190e-01
43	https_token	1.146691e-01
44	nb_subdomains	1.128907e-01
45	suspicious_tld	1.100896e-01
46	shortening_service	1.061200e-01
47	nb_semicolumn	1.035541e-01
48	nb_hyphens	1.001075e-01
49	domain_in_brand	9.822216e-02
50	nb_colon	9.283531e-02
51	nb_extCSS	8.356663e-02
52	ratio_extHyperlinks	8.335725e-02
53	tld_in_path	7.914651e-02
54	shortest_word_path	7.436495e-02
55	nb_dslash	7.260234e-02
56	http_in_path	7.077624e-02
57	whois_registered_domain	6.697907e-02
58	brand_in_path	6.515575e-02
59	brand_in_subdomain	6.425702e-02
60	web_traffic	6.038772e-02
61	popup_window	5.760197e-02
62	nb_external_redirection	5.620994e-02
63	shortest_words_raw	3.936361e-02
64	nb_underscore	3.809134e-02
65	ratio_extErrors	3.470251e-02
66	nb_tilde	3.014233e-02
67	nb_percent	2.810129e-02
68	nb_star	2.646512e-02
69	nb_dollar	2.496206e-02
70	nb_redirection	2.440520e-02
71	random_domain	1.963062e-02
72	login_form	1.900010e-02
73	punycode	1.871039e-02
74	char_repeat	1.473217e-02
75	iframe	1.208332e-02
76	nb_comma	1.186465e-02
77	port	9.011116e-03
78	onmouseover	7.787061e-03
79	right_clic	4.680056e-03
80	nb_space	4.193222e-03
81	path_extension	5.592660e-17
82	nb_or	NaN
83	ratio_nullHyperlinks	NaN
84	ratio_intRedirection	NaN
85	ratio_intErrors	NaN
86	submit_email	NaN

```
[ ]: 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))
```

```
[ ]: 17
```

```
[ ]: corr_feature
```

```
[ ]: {'char_repeat',
      'iframe',
      'login_form',
      'nb_comma',
      'nb_or',
      'nb_space',
      'onmouseover',
      'path_extension',
      'port',
      'punycode',
      'random_domain',
      'ratio_intErrors',
      'ratio_intRedirection',
      'ratio_nullHyperlinks',
      'right_clic',
      'sfh',
      'submit_email'}
```

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

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

```
[ ]: 71
```

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

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

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

Count of Legitimate Websites = 5715

Count of Phishy Websites = 5715

```
[ ]: X = df.drop(['status'], axis=1, inplace=False)
      #X.head()
      #same work
      ##inplace true modifies the og data & does not return anything
      ##inplace false does not modify og data but returns something which we store in
      ↪ a var
      # X= df.drop(columns='Result')
      # X.head()
```

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

```
[ ]: y = df['status']
      y = pd.DataFrame(y)
      y.head()
```

```
[ ]:      status
      0      0
      1      1
      2      1
      3      0
      4      0
```

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

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

```
[ ]: ((8001, 70), (3429, 70), (8001, 1), (3429, 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 8001 samples.

Testing set has 3429 samples.

```
[ ]: from sklearn.preprocessing import MinMaxScaler
```

```

scaler= MinMaxScaler()

col_X_train = [X_train.columns[:]]

for c in col_X_train:
    X_train[c]= scaler.fit_transform(X_train[c])

#X_train.head(5)

```

```

[ ]: col_X_test = [X_test.columns[:]]

for c in col_X_test:
    X_test[c]= scaler.transform(X_test[c])

#X_test.head(5)

```

```

[ ]: from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# defining parameter range
param_grid = {'penalty' : ['l2'],
              'C' : [10, 20, 30], #0.1, 1, 10, 20,
              '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 15 candidates, totalling 150 fits
{'C': 20, 'max_iter': 2500, 'penalty': 'l2', 'solver': 'lbfgs'}
LogisticRegression(C=20, max_iter=2500)
0.9423824906367042

```

[ ]: logr_model = grid_logr.best_estimator_

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

```

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

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

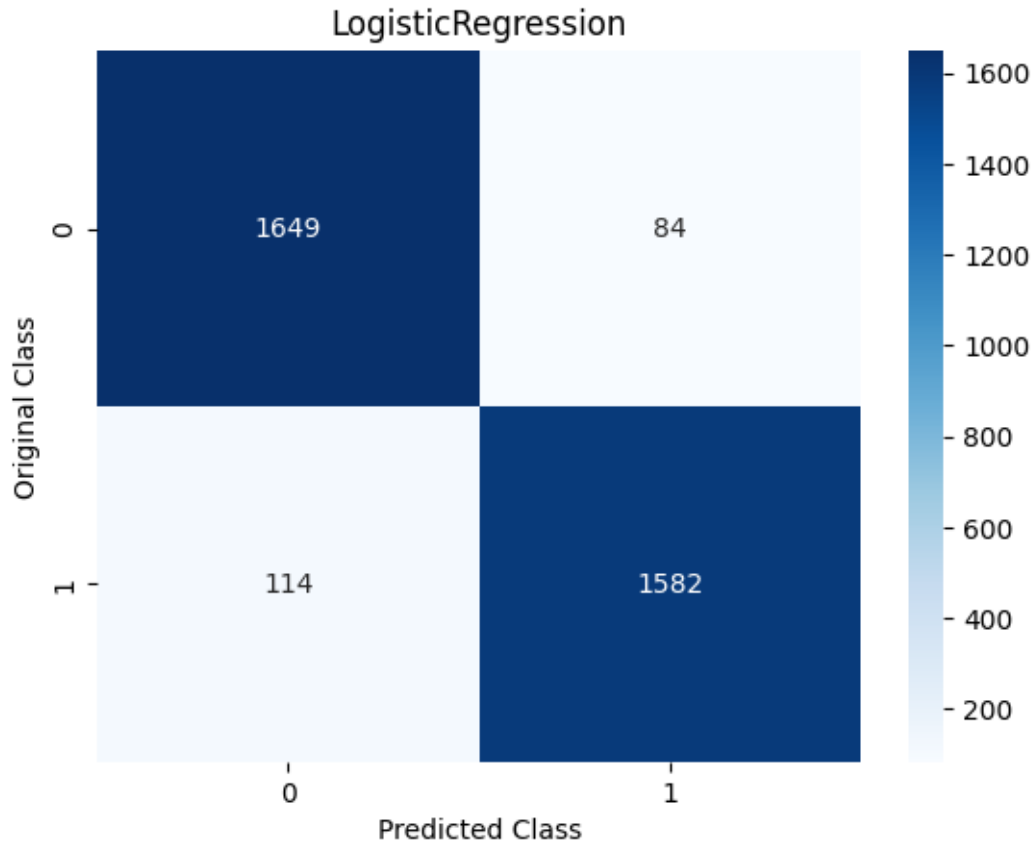
```
[ ]: print ("Accuracy of logr classifier : ", accuracy_score(y_test,   
↳logr_predict)*100)
```

Accuracy of logr classifier : 94.22572178477691

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

	precision	recall	f1-score	support
0	0.94	0.95	0.94	1733
1	0.95	0.93	0.94	1696
accuracy			0.94	3429
macro avg	0.94	0.94	0.94	3429
weighted avg	0.94	0.94	0.94	3429

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



```
[ ]: # from sklearn.neighbors import KNeighborsClassifier

# #training_accuracy=[]
# test_accuracy=[]

# neighbors=range(1,10)
# ##values.ravel() converts vector y to flattened array
# for i in neighbors:
#     knn=KNeighborsClassifier(n_neighbors=i)
#     knn_model = knn.fit(X_train,y_train.values.ravel())
#     #training_accuracy.append(knn.score(X_train,y_train.values.ravel()))
#     test_accuracy.append(knn_model.score(X_test,y_test.values.ravel()))
```

```
[ ]: # plt.plot(neighbors,test_accuracy,label="test accuracy")
# plt.ylabel("Accuracy")
# plt.xlabel("number of neighbors")
# plt.legend()
# plt.show()
```



```
[ ]: from sklearn.neighbors import KNeighborsClassifier

# defining parameter range
param_grid = {'n_neighbors': [1,2,3,4,5,6,7,8,9,10]}

grid_knn = GridSearchCV(KNeighborsClassifier(), param_grid, refit = True, cv = 10, verbose = 3, n_jobs = -1)

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

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

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

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits
{'n_neighbors': 3}
KNeighborsClassifier(n_neighbors=3)
0.9250101435705368
```

```
[ ]: 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))
```

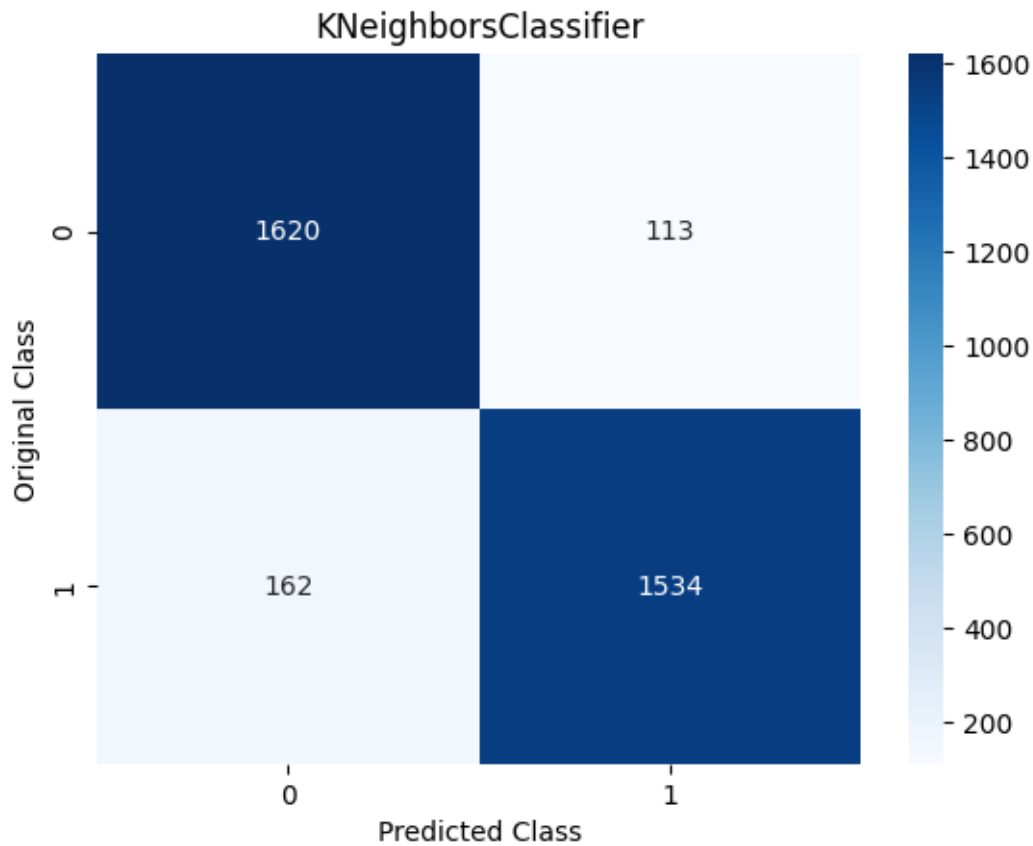
```
The accuracy of knn Classifier is: 91.98016914552348
```

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

	precision	recall	f1-score	support
0	0.91	0.93	0.92	1733
1	0.93	0.90	0.92	1696
accuracy			0.92	3429
macro avg	0.92	0.92	0.92	3429
weighted avg	0.92	0.92	0.92	3429

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

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



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

# RocCurveDisplay.from_predictions(y_test,knn_y_pred_proba_positive)

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

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

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

# defining parameter range
param_grid = {'C': [1, 10], #0.1, 1, 10
              'gamma': [1, 0.1], #
              'kernel': ['rbf']} # '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 4 candidates, totalling 40 fits
{'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
SVC(C=10, gamma=0.1)
0.9552549937578029

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

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

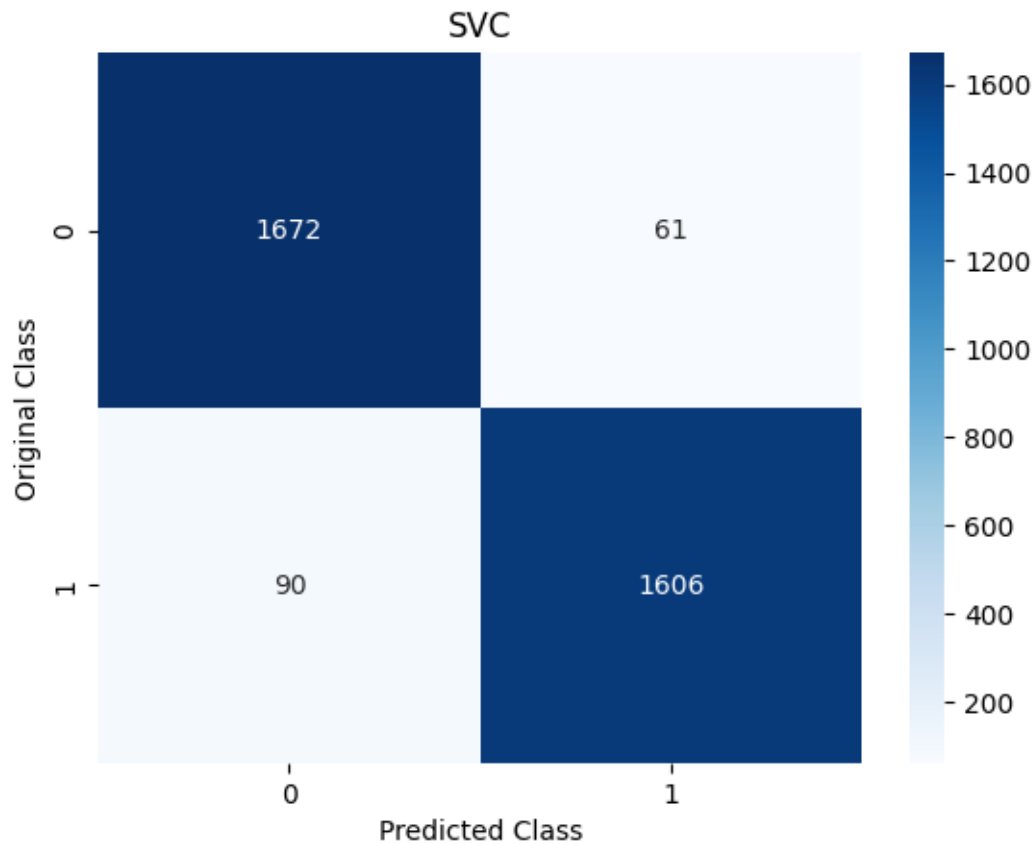
```
[ ]: print('The accuracy of svc Classifier is: ', 100.0 * accuracy_score(y_test,
↪svc_predict))
```

The accuracy of svc Classifier is: 95.59638378536016

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

	precision	recall	f1-score	support
0	0.95	0.96	0.96	1733
1	0.96	0.95	0.96	1696
accuracy			0.96	3429
macro avg	0.96	0.96	0.96	3429
weighted avg	0.96	0.96	0.96	3429

```
[ ]: 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], #.5
              'gamma': [.1], #1,.01
              'kernel': ['rbf']} # '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 1 candidates, totalling 10 fits
{'gamma': 0.1, 'kernel': 'rbf', 'nu': 0.1}
NuSVC(gamma=0.1, nu=0.1)
0.9541303058676656
```

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

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

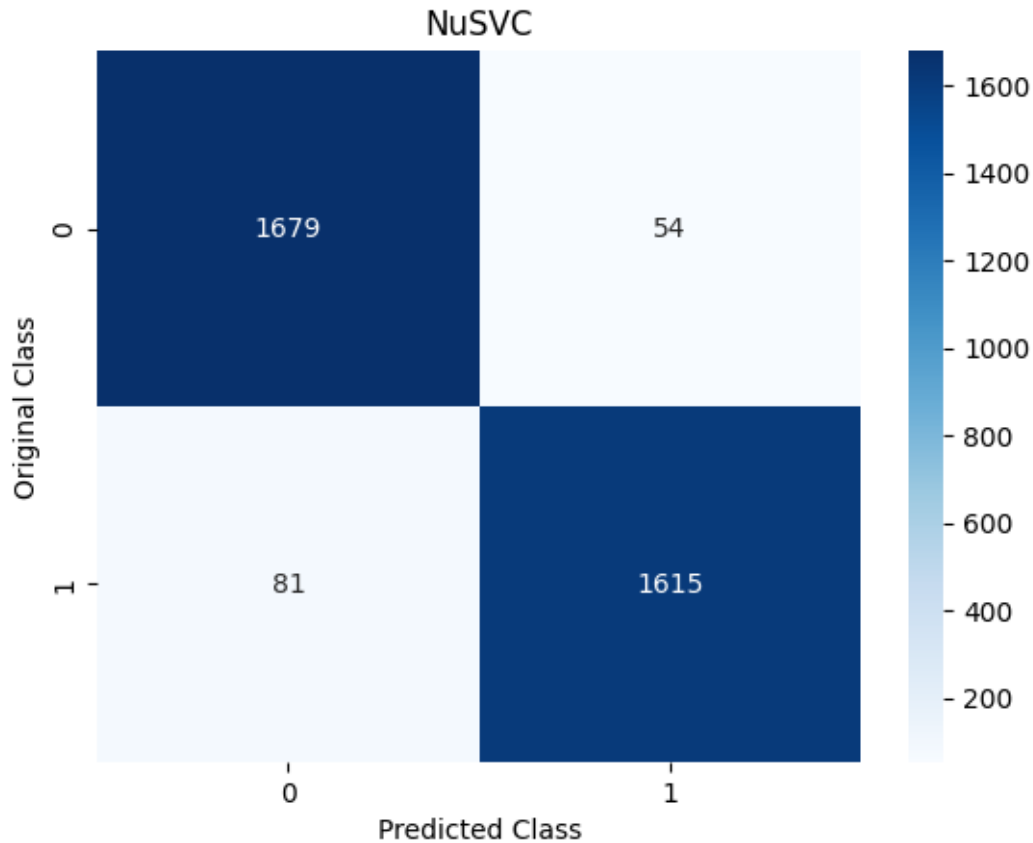
```
[ ]: print('The accuracy of nusvc Classifier is: ', 100.0 * accuracy_score(y_test,
↪nusvc_predict))
```

```
The accuracy of nusvc Classifier is: 96.06299212598425
```

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

	precision	recall	f1-score	support
0	0.95	0.97	0.96	1733
1	0.97	0.95	0.96	1696
accuracy			0.96	3429
macro avg	0.96	0.96	0.96	3429
weighted avg	0.96	0.96	0.96	3429

```
[ ]: sns.heatmap(confusion_matrix(y_test, 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': 'l1', 'tol': 0.001}
LinearSVC(C=10, dual=False, penalty='l1', tol=0.001)
0.9431320224719102

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

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

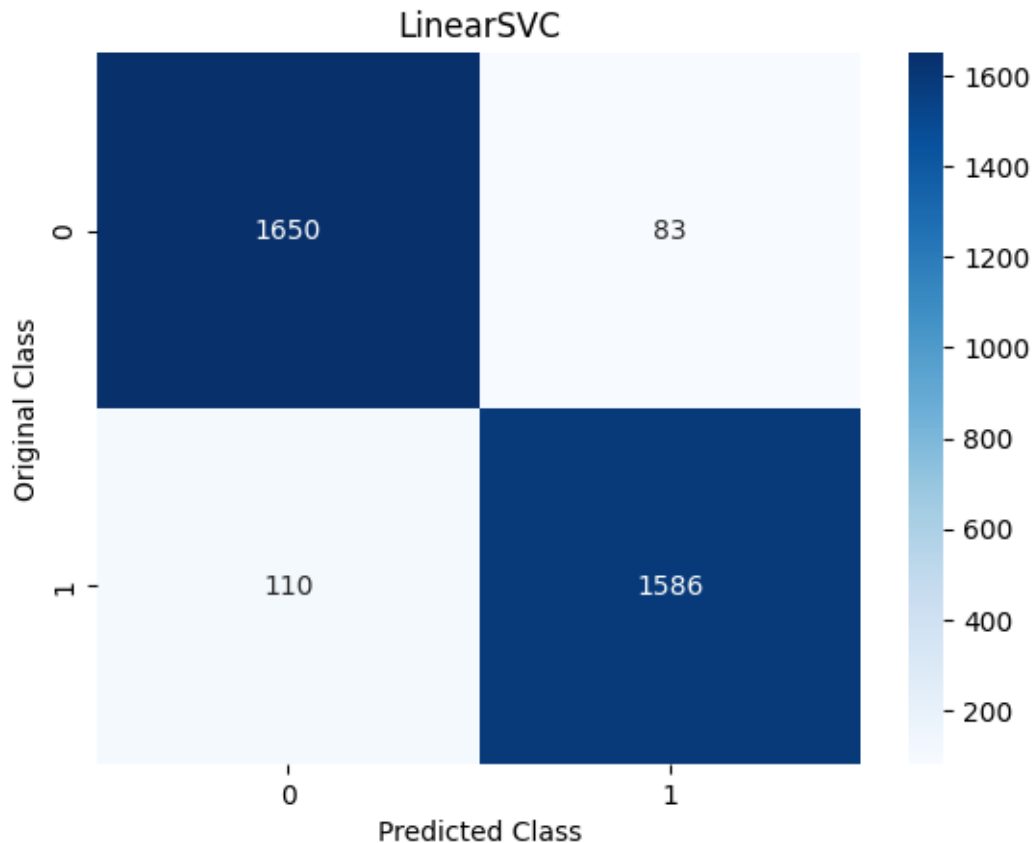
```
[ ]: print('The accuracy of lsvc Classifier is: ', 100.0 * accuracy_score(y_test,
      ↪lsvc_predict))
```

The accuracy of lsvc Classifier is: 94.37153689122194

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

	precision	recall	f1-score	support
0	0.94	0.95	0.94	1733
1	0.95	0.94	0.94	1696
accuracy			0.94	3429
macro avg	0.94	0.94	0.94	3429
weighted avg	0.94	0.94	0.94	3429

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



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

# defining parameter range
param_grid = {'n_estimators': [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 3 candidates, totalling 30 fits
 {'n_estimators': 300}


```
AdaBoostClassifier(n_estimators=300)
0.9561309300873908
```

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

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

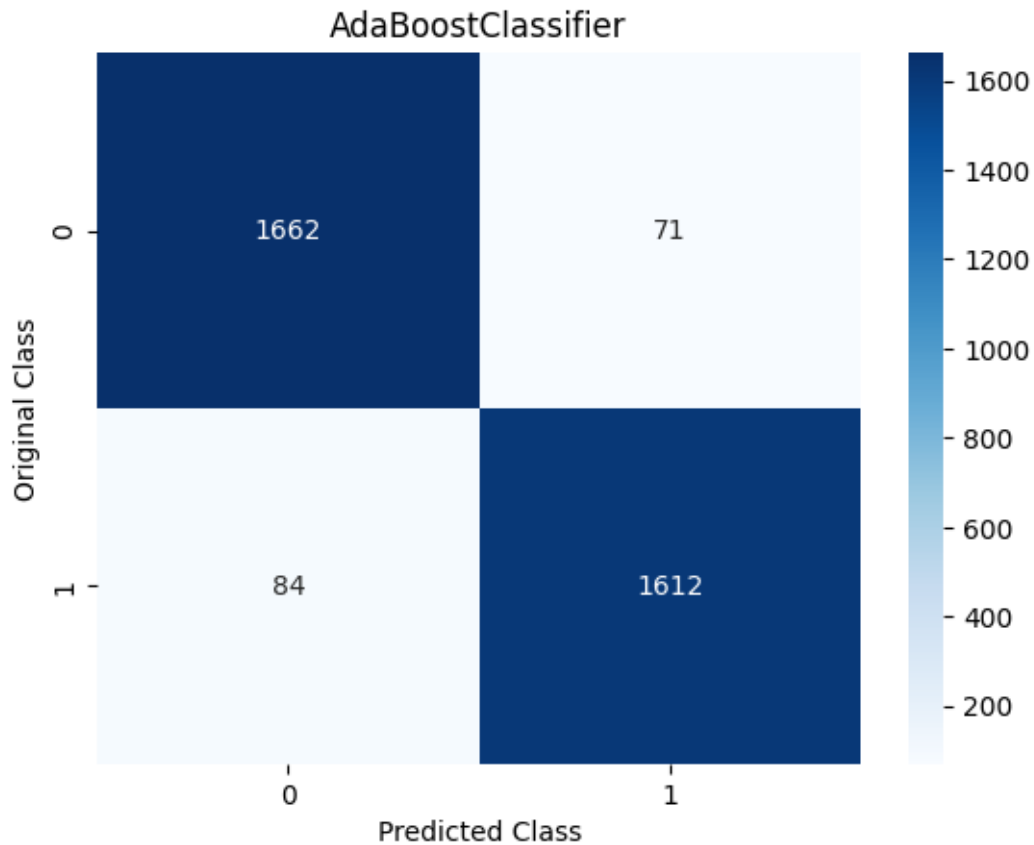
```
[ ]: print('The accuracy of Ada Boost Classifier is: ', 100.0 *  
      ↪accuracy_score(ada_predict,y_test))
```

The accuracy of Ada Boost Classifier is: 95.47973170020414

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

	precision	recall	f1-score	support
0	0.95	0.96	0.96	1733
1	0.96	0.95	0.95	1696
accuracy			0.95	3429
macro avg	0.95	0.95	0.95	3429
weighted avg	0.95	0.95	0.95	3429

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



```
[ ]: from xgboost import XGBClassifier

# defining parameter range
param_grid = {
    "gamma": [.01, .1, .5], #
    "n_estimators": [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 9 candidates, totalling 90 fits

```
{'gamma': 0.1, 'n_estimators': 150}
```

```
XGBClassifier(base_score=0.5, booster='gbtree', callbacks=None,
              colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
              early_stopping_rounds=None, enable_categorical=False,
              eval_metric=None, gamma=0.1, 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=150,
              n_jobs=0, num_parallel_tree=1, predictor='auto', random_state=0,
              reg_alpha=0, reg_lambda=1, ...)
```

0.9683781210986269

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

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

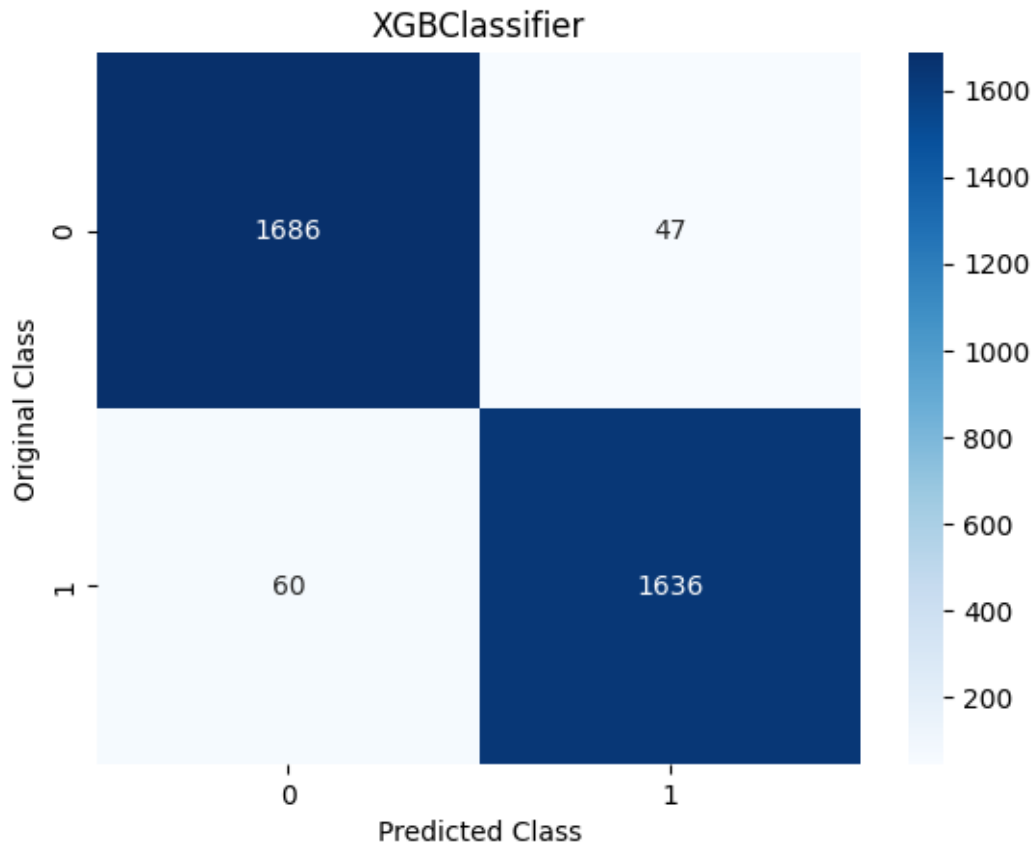
```
[ ]: print('The accuracy of XGBoost Classifier is: ', 100.0 *
      ↪accuracy_score(xgb_predict,y_test))
```

The accuracy of XGBoost Classifier is: 96.87955672207642

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

	precision	recall	f1-score	support
0	0.97	0.97	0.97	1733
1	0.97	0.96	0.97	1696
accuracy			0.97	3429
macro avg	0.97	0.97	0.97	3429
weighted avg	0.97	0.97	0.97	3429

```
[ ]: sns.heatmap(confusion_matrix(y_test, 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": [.5], #.1,.5,1
    "n_estimators": [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 3 candidates, totalling 30 fits
{'learning_rate': 0.5, 'n_estimators': 250}
GradientBoostingClassifier(learning_rate=0.5, n_estimators=250)
0.9672537453183521

```
[ ]: gbc_model = grid_gbc.best_estimator_  
      #gbc_model = gbc.fit(X_train,y_train.values.ravel())  
  
      #clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,  
      #    max_depth=1, random_state=0).fit(X_train, y_train)  
      #clf.score(X_test, y_test)
```

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

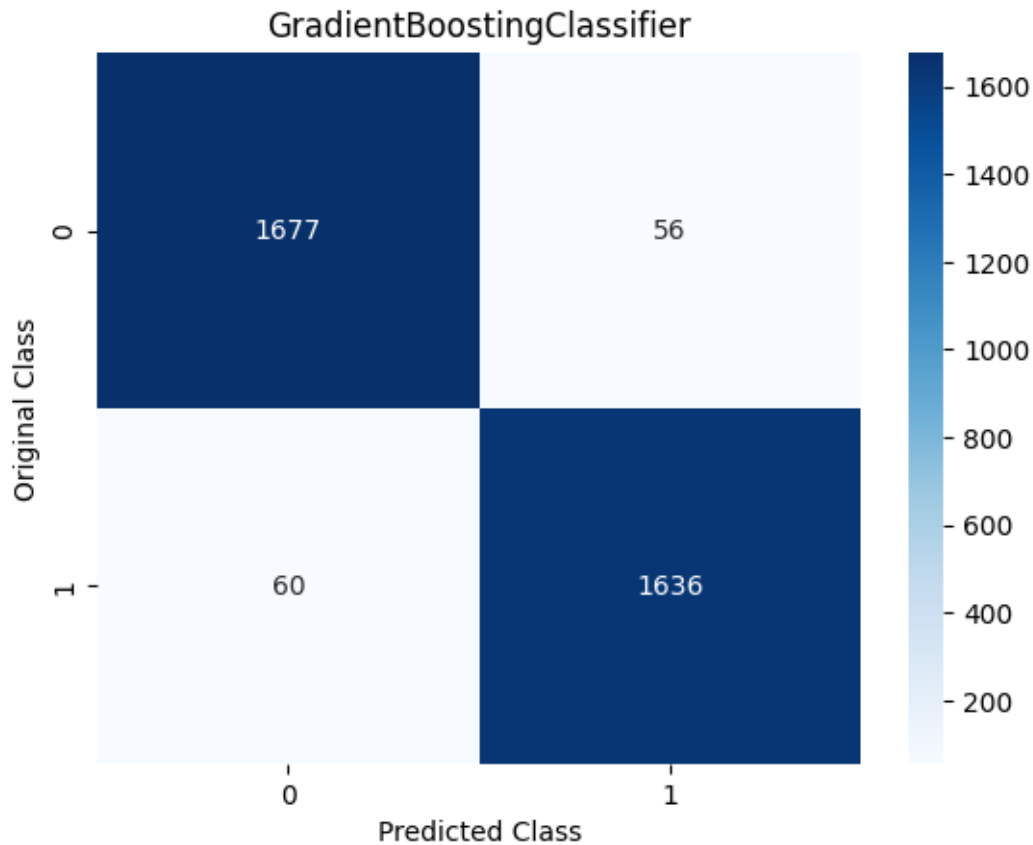
```
[ ]: print('The accuracy of GradientBoost Classifier is: ', 100.0 *  
      ↪accuracy_score(gbc_predict,y_test))
```

The accuracy of GradientBoost Classifier is: 96.61708953047535

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

	precision	recall	f1-score	support
0	0.97	0.97	0.97	1733
1	0.97	0.96	0.97	1696
accuracy			0.97	3429
macro avg	0.97	0.97	0.97	3429
weighted avg	0.97	0.97	0.97	3429

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



```
[ ]: # gbc_model.get_params().keys()
```

```
[ ]: # import inspect
# import sklearn
# import xgboost

# models = [xgboost.XGBClassifier]
# for m in models:
#     hyperparams = inspect.signature(m.__init__)
#     print(hyperparams)
# #or
# xgb_model.get_params().keys()
```

```
[ ]: from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

# defining parameter range
param_grid = {
    "base_estimator": [DecisionTreeClassifier()],
    "n_estimators": [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 3 candidates, totalling 30 fits
{'base_estimator': DecisionTreeClassifier(), 'n_estimators': 250}
BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=250)
0.9570049937578027

```

```

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

```

```

[ ]: bag_predict = bag_model.predict(X_test)

```

```

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

```

```

The accuracy of Bagging Classifier is: 95.88801399825022

```

```

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

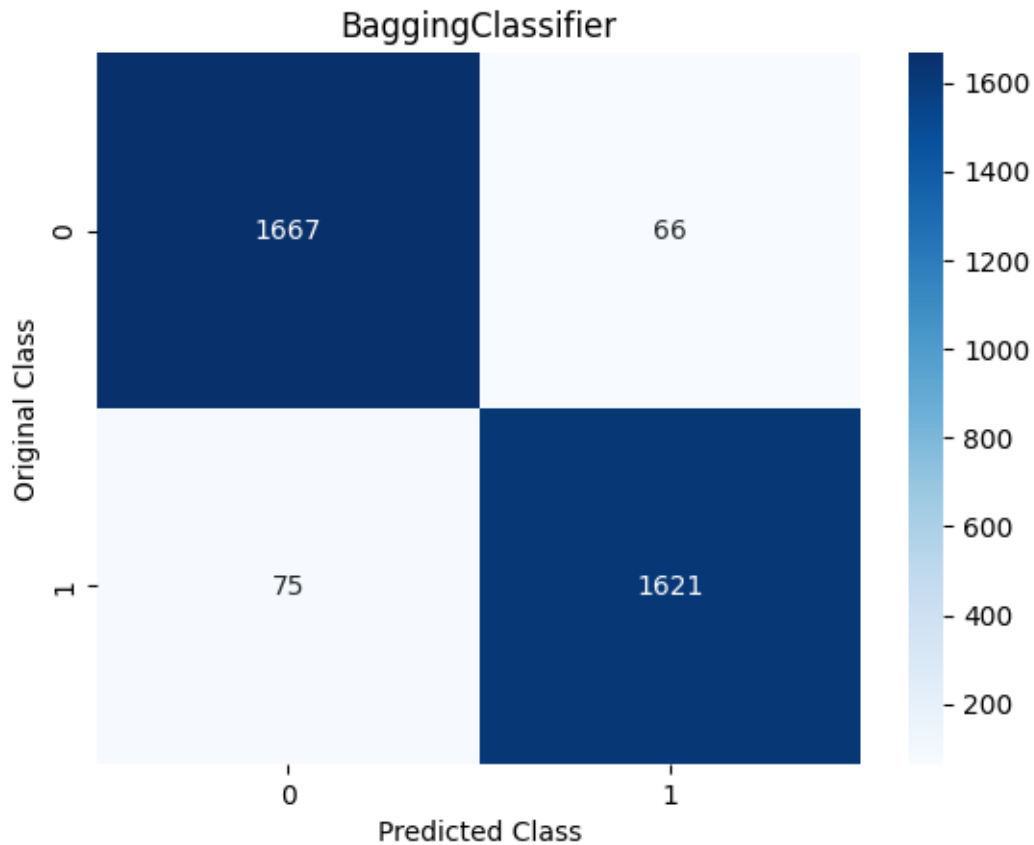
```

	precision	recall	f1-score	support
0	0.96	0.96	0.96	1733
1	0.96	0.96	0.96	1696
accuracy			0.96	3429
macro avg	0.96	0.96	0.96	3429
weighted avg	0.96	0.96	0.96	3429

```

[ ]: sns.heatmap(confusion_matrix(y_test, 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]
}

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 4 candidates, totalling 40 fits
{'n_estimators': 200}
RandomForestClassifier(n_estimators=200)
0.9647545255930087
```

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

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

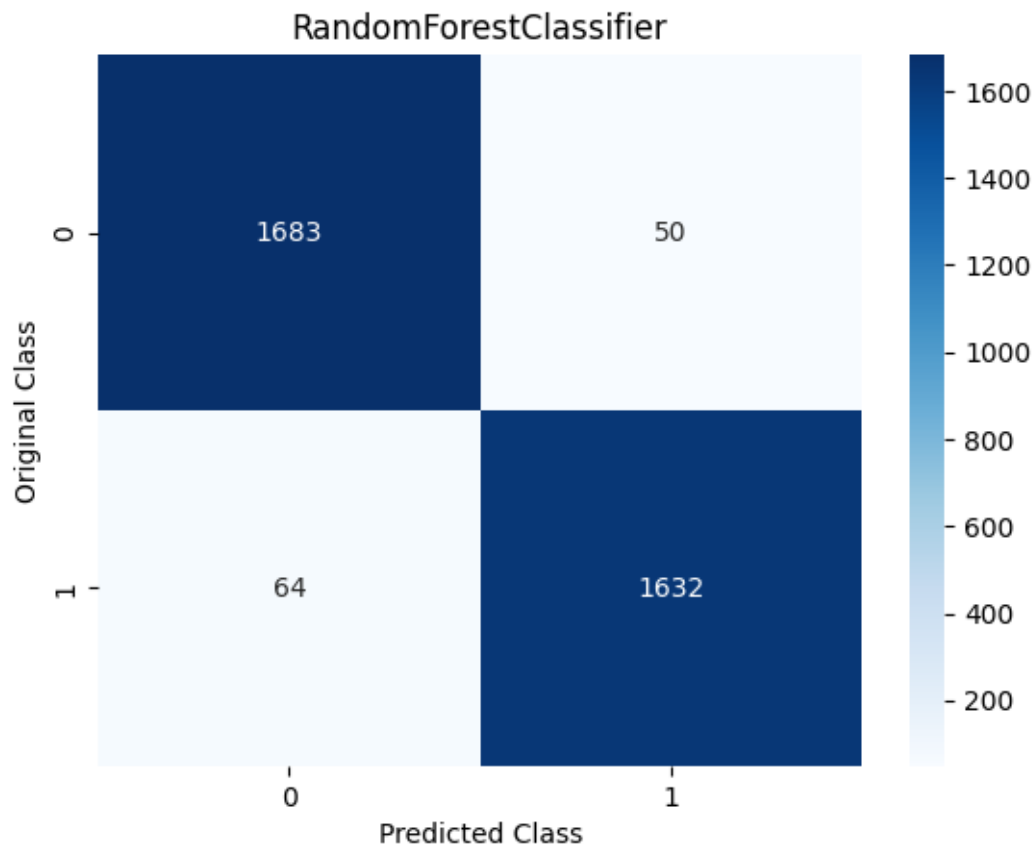
```
[ ]: print('The accuracy of RandomForest Classifier is: ' , 100.0 *
      ↪accuracy_score(rfc_predict,y_test))
```

The accuracy of RandomForest Classifier is: 96.67541557305337

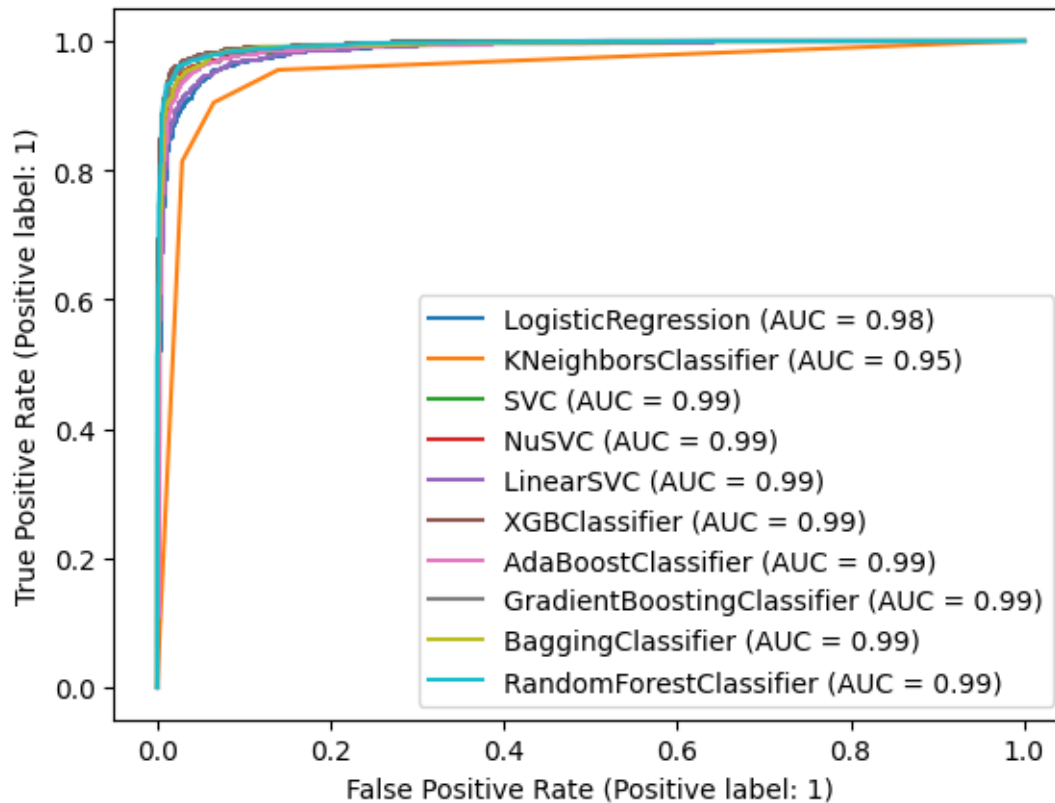
```
[ ]: print(classification_report(y_test, rfc_predict))
```

	precision	recall	f1-score	support
0	0.96	0.97	0.97	1733
1	0.97	0.96	0.97	1696
accuracy			0.97	3429
macro avg	0.97	0.97	0.97	3429
weighted avg	0.97	0.97	0.97	3429

```
[ ]: sns.heatmap(confusion_matrix(y_test, 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_7"

Layer (type)	Output Shape	Param #
lstm_7 (LSTM)	(None, 100)	68400
batch_normalization_7 (Batch Normalization)	(None, 100)	400
dense_28 (Dense)	(None, 50)	5050
lstm_7 (LSTM)	(None, 100)	68400
batch_normalization_7 (Batch Normalization)	(None, 100)	400
dense_28 (Dense)	(None, 50)	5050
dense_29 (Dense)	(None, 25)	1275
dense_30 (Dense)	(None, 10)	260
dense_31 (Dense)	(None, 1)	11

Total params: 75,396
 Trainable params: 75,196
 Non-trainable params: 200

Epoch 1/100
 200/200 [=====] - 5s 7ms/step - loss: 0.2676 -
 accuracy: 0.8920 - val_loss: 0.4294 - val_accuracy: 0.9244
 Epoch 2/100
 200/200 [=====] - 1s 4ms/step - loss: 0.1532 -

accuracy: 0.9433 - val_loss: 0.2506 - val_accuracy: 0.9394
Epoch 3/100
200/200 [=====] - 1s 4ms/step - loss: 0.1408 -
accuracy: 0.9491 - val_loss: 0.1687 - val_accuracy: 0.9413
Epoch 4/100
200/200 [=====] - 1s 4ms/step - loss: 0.1271 -
accuracy: 0.9545 - val_loss: 0.1527 - val_accuracy: 0.9450
Epoch 5/100
200/200 [=====] - 1s 4ms/step - loss: 0.1249 -
accuracy: 0.9513 - val_loss: 0.1573 - val_accuracy: 0.9457
Epoch 6/100
200/200 [=====] - 1s 4ms/step - loss: 0.1138 -
accuracy: 0.9592 - val_loss: 0.1566 - val_accuracy: 0.9463
Epoch 7/100
200/200 [=====] - 1s 4ms/step - loss: 0.1012 -
accuracy: 0.9614 - val_loss: 0.1820 - val_accuracy: 0.9325
Epoch 8/100
200/200 [=====] - 1s 4ms/step - loss: 0.0986 -
accuracy: 0.9619 - val_loss: 0.1878 - val_accuracy: 0.9438
Epoch 9/100
200/200 [=====] - 1s 4ms/step - loss: 0.0946 -
accuracy: 0.9655 - val_loss: 0.1649 - val_accuracy: 0.9469
Epoch 10/100
200/200 [=====] - 1s 4ms/step - loss: 0.0881 -
accuracy: 0.9680 - val_loss: 0.1615 - val_accuracy: 0.9538
Epoch 11/100
200/200 [=====] - 1s 4ms/step - loss: 0.0810 -
accuracy: 0.9706 - val_loss: 0.1657 - val_accuracy: 0.9519
Epoch 12/100
200/200 [=====] - 1s 4ms/step - loss: 0.0844 -
accuracy: 0.9688 - val_loss: 0.2158 - val_accuracy: 0.9332
Epoch 13/100
200/200 [=====] - 1s 4ms/step - loss: 0.0865 -
accuracy: 0.9675 - val_loss: 0.1742 - val_accuracy: 0.9500
Epoch 14/100
200/200 [=====] - 1s 4ms/step - loss: 0.0712 -
accuracy: 0.9745 - val_loss: 0.2043 - val_accuracy: 0.9413
Epoch 15/100
200/200 [=====] - 1s 4ms/step - loss: 0.0668 -
accuracy: 0.9755 - val_loss: 0.1765 - val_accuracy: 0.9507
Epoch 16/100
200/200 [=====] - 1s 4ms/step - loss: 0.0599 -
accuracy: 0.9780 - val_loss: 0.1773 - val_accuracy: 0.9463
Epoch 17/100
200/200 [=====] - 1s 4ms/step - loss: 0.0626 -
accuracy: 0.9758 - val_loss: 0.1878 - val_accuracy: 0.9444
Epoch 18/100
200/200 [=====] - 1s 4ms/step - loss: 0.0562 -

accuracy: 0.9783 - val_loss: 0.1975 - val_accuracy: 0.9438
Epoch 19/100
200/200 [=====] - 1s 4ms/step - loss: 0.0517 -
accuracy: 0.9800 - val_loss: 0.2071 - val_accuracy: 0.9375
Epoch 20/100
200/200 [=====] - 1s 4ms/step - loss: 0.0527 -
accuracy: 0.9795 - val_loss: 0.2045 - val_accuracy: 0.9482
Epoch 21/100
200/200 [=====] - 1s 5ms/step - loss: 0.0552 -
accuracy: 0.9789 - val_loss: 0.2104 - val_accuracy: 0.9450
Epoch 22/100
200/200 [=====] - 1s 5ms/step - loss: 0.0505 -
accuracy: 0.9794 - val_loss: 0.2047 - val_accuracy: 0.9413
Epoch 23/100
200/200 [=====] - 1s 6ms/step - loss: 0.0393 -
accuracy: 0.9858 - val_loss: 0.2579 - val_accuracy: 0.9375
Epoch 24/100
200/200 [=====] - 1s 5ms/step - loss: 0.0447 -
accuracy: 0.9825 - val_loss: 0.2165 - val_accuracy: 0.9463
Epoch 25/100
200/200 [=====] - 1s 7ms/step - loss: 0.0337 -
accuracy: 0.9867 - val_loss: 0.2362 - val_accuracy: 0.9450
Epoch 26/100
200/200 [=====] - 1s 5ms/step - loss: 0.0353 -
accuracy: 0.9864 - val_loss: 0.2335 - val_accuracy: 0.9419
Epoch 27/100
200/200 [=====] - 1s 5ms/step - loss: 0.0385 -
accuracy: 0.9852 - val_loss: 0.2517 - val_accuracy: 0.9394
Epoch 28/100
200/200 [=====] - 1s 6ms/step - loss: 0.0313 -
accuracy: 0.9877 - val_loss: 0.2369 - val_accuracy: 0.9469
Epoch 29/100
200/200 [=====] - 1s 5ms/step - loss: 0.0298 -
accuracy: 0.9895 - val_loss: 0.2609 - val_accuracy: 0.9444
Epoch 30/100
200/200 [=====] - 1s 5ms/step - loss: 0.0271 -
accuracy: 0.9887 - val_loss: 0.2459 - val_accuracy: 0.9457
Epoch 31/100
200/200 [=====] - 1s 5ms/step - loss: 0.0298 -
accuracy: 0.9891 - val_loss: 0.2607 - val_accuracy: 0.9475
Epoch 32/100
200/200 [=====] - 1s 4ms/step - loss: 0.0269 -
accuracy: 0.9905 - val_loss: 0.2726 - val_accuracy: 0.9425
Epoch 33/100
200/200 [=====] - 1s 4ms/step - loss: 0.0337 -
accuracy: 0.9869 - val_loss: 0.2987 - val_accuracy: 0.9369
Epoch 34/100
200/200 [=====] - 1s 5ms/step - loss: 0.0283 -

accuracy: 0.9895 - val_loss: 0.2523 - val_accuracy: 0.9444
 Epoch 35/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0271 -
 accuracy: 0.9894 - val_loss: 0.2673 - val_accuracy: 0.9394
 Epoch 36/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0221 -
 accuracy: 0.9919 - val_loss: 0.2753 - val_accuracy: 0.9382
 Epoch 37/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0251 -
 accuracy: 0.9908 - val_loss: 0.2845 - val_accuracy: 0.9413
 Epoch 38/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0211 -
 accuracy: 0.9927 - val_loss: 0.2752 - val_accuracy: 0.9438
 Epoch 39/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0255 -
 accuracy: 0.9898 - val_loss: 0.2737 - val_accuracy: 0.9444
 Epoch 40/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0203 -
 accuracy: 0.9928 - val_loss: 0.2878 - val_accuracy: 0.9407
 Epoch 41/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0153 -
 accuracy: 0.9941 - val_loss: 0.2910 - val_accuracy: 0.9413
 Epoch 42/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0218 -
 accuracy: 0.9911 - val_loss: 0.3178 - val_accuracy: 0.9382
 Epoch 43/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0225 -
 accuracy: 0.9923 - val_loss: 0.3329 - val_accuracy: 0.9438
 Epoch 44/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0238 -
 accuracy: 0.9917 - val_loss: 0.3298 - val_accuracy: 0.9388
 Epoch 45/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0181 -
 accuracy: 0.9944 - val_loss: 0.3058 - val_accuracy: 0.9457
 Epoch 46/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0147 -
 accuracy: 0.9944 - val_loss: 0.3404 - val_accuracy: 0.9419
 Epoch 47/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0132 -
 accuracy: 0.9959 - val_loss: 0.3298 - val_accuracy: 0.9432
 Epoch 48/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0165 -
 accuracy: 0.9944 - val_loss: 0.3525 - val_accuracy: 0.9432
 Epoch 49/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0125 -
 accuracy: 0.9959 - val_loss: 0.3324 - val_accuracy: 0.9407
 Epoch 50/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0171 -

accuracy: 0.9939 - val_loss: 0.3208 - val_accuracy: 0.9444
 Epoch 51/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0253 -
 accuracy: 0.9914 - val_loss: 0.3187 - val_accuracy: 0.9325
 Epoch 52/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0231 -
 accuracy: 0.9933 - val_loss: 0.3388 - val_accuracy: 0.9344
 Epoch 53/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0167 -
 accuracy: 0.9942 - val_loss: 0.3228 - val_accuracy: 0.9432
 Epoch 54/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0112 -
 accuracy: 0.9959 - val_loss: 0.3938 - val_accuracy: 0.9338
 Epoch 55/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0110 -
 accuracy: 0.9967 - val_loss: 0.3442 - val_accuracy: 0.9463
 Epoch 56/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0102 -
 accuracy: 0.9958 - val_loss: 0.3891 - val_accuracy: 0.9394
 Epoch 57/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0119 -
 accuracy: 0.9952 - val_loss: 0.3883 - val_accuracy: 0.9394
 Epoch 58/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0193 -
 accuracy: 0.9941 - val_loss: 0.3652 - val_accuracy: 0.9413
 Epoch 59/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0093 -
 accuracy: 0.9966 - val_loss: 0.3594 - val_accuracy: 0.9344
 Epoch 60/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0206 -
 accuracy: 0.9927 - val_loss: 0.3807 - val_accuracy: 0.9325
 Epoch 61/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0141 -
 accuracy: 0.9944 - val_loss: 0.3485 - val_accuracy: 0.9400
 Epoch 62/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0188 -
 accuracy: 0.9934 - val_loss: 0.3983 - val_accuracy: 0.9325
 Epoch 63/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0149 -
 accuracy: 0.9947 - val_loss: 0.3244 - val_accuracy: 0.9425
 Epoch 64/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0077 -
 accuracy: 0.9973 - val_loss: 0.3696 - val_accuracy: 0.9413
 Epoch 65/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0128 -
 accuracy: 0.9955 - val_loss: 0.3322 - val_accuracy: 0.9419
 Epoch 66/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0127 -

accuracy: 0.9952 - val_loss: 0.3567 - val_accuracy: 0.9413
 Epoch 67/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0120 -
 accuracy: 0.9948 - val_loss: 0.3417 - val_accuracy: 0.9388
 Epoch 68/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0109 -
 accuracy: 0.9953 - val_loss: 0.3484 - val_accuracy: 0.9413
 Epoch 69/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0155 -
 accuracy: 0.9953 - val_loss: 0.3411 - val_accuracy: 0.9425
 Epoch 70/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0140 -
 accuracy: 0.9947 - val_loss: 0.3101 - val_accuracy: 0.9469
 Epoch 71/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0085 -
 accuracy: 0.9970 - val_loss: 0.3577 - val_accuracy: 0.9444
 Epoch 72/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0104 -
 accuracy: 0.9967 - val_loss: 0.3211 - val_accuracy: 0.9419
 Epoch 73/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0078 -
 accuracy: 0.9975 - val_loss: 0.3928 - val_accuracy: 0.9457
 Epoch 74/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0126 -
 accuracy: 0.9955 - val_loss: 0.3784 - val_accuracy: 0.9425
 Epoch 75/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0141 -
 accuracy: 0.9942 - val_loss: 0.3604 - val_accuracy: 0.9425
 Epoch 76/100
 200/200 [=====] - 1s 6ms/step - loss: 0.0124 -
 accuracy: 0.9962 - val_loss: 0.3799 - val_accuracy: 0.9419
 Epoch 77/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0083 -
 accuracy: 0.9969 - val_loss: 0.3428 - val_accuracy: 0.9463
 Epoch 78/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0137 -
 accuracy: 0.9955 - val_loss: 0.3506 - val_accuracy: 0.9369
 Epoch 79/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0117 -
 accuracy: 0.9969 - val_loss: 0.3616 - val_accuracy: 0.9413
 Epoch 80/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0086 -
 accuracy: 0.9970 - val_loss: 0.3473 - val_accuracy: 0.9450
 Epoch 81/100
 200/200 [=====] - 1s 4ms/step - loss: 0.0047 -
 accuracy: 0.9981 - val_loss: 0.3666 - val_accuracy: 0.9438
 Epoch 82/100
 200/200 [=====] - 1s 5ms/step - loss: 0.0051 -

```

accuracy: 0.9984 - val_loss: 0.3437 - val_accuracy: 0.9475
Epoch 83/100
200/200 [=====] - 1s 4ms/step - loss: 0.0094 -
accuracy: 0.9969 - val_loss: 0.3590 - val_accuracy: 0.9432
Epoch 84/100
200/200 [=====] - 1s 5ms/step - loss: 0.0111 -
accuracy: 0.9961 - val_loss: 0.3388 - val_accuracy: 0.9444
Epoch 85/100
200/200 [=====] - 1s 5ms/step - loss: 0.0138 -
accuracy: 0.9959 - val_loss: 0.3418 - val_accuracy: 0.9432
Epoch 86/100
200/200 [=====] - 1s 5ms/step - loss: 0.0206 -
accuracy: 0.9947 - val_loss: 0.3257 - val_accuracy: 0.9375
Epoch 87/100
200/200 [=====] - 1s 4ms/step - loss: 0.0077 -
accuracy: 0.9969 - val_loss: 0.3360 - val_accuracy: 0.9457
Epoch 88/100
200/200 [=====] - 1s 4ms/step - loss: 0.0060 -
accuracy: 0.9981 - val_loss: 0.3484 - val_accuracy: 0.9463
Epoch 89/100
200/200 [=====] - 1s 4ms/step - loss: 0.0102 -
accuracy: 0.9961 - val_loss: 0.3951 - val_accuracy: 0.9475
Epoch 90/100
200/200 [=====] - 1s 5ms/step - loss: 0.0080 -
accuracy: 0.9970 - val_loss: 0.3805 - val_accuracy: 0.9394
Epoch 91/100
200/200 [=====] - 1s 5ms/step - loss: 0.0111 -
accuracy: 0.9962 - val_loss: 0.4051 - val_accuracy: 0.9413
Epoch 92/100
200/200 [=====] - 1s 4ms/step - loss: 0.0128 -
accuracy: 0.9966 - val_loss: 0.3285 - val_accuracy: 0.9438
Epoch 93/100
200/200 [=====] - 1s 4ms/step - loss: 0.0086 -
accuracy: 0.9975 - val_loss: 0.3508 - val_accuracy: 0.9457
Epoch 94/100
200/200 [=====] - 1s 5ms/step - loss: 0.0177 -
accuracy: 0.9944 - val_loss: 0.3363 - val_accuracy: 0.9425
Epoch 95/100
200/200 [=====] - 1s 5ms/step - loss: 0.0114 -
accuracy: 0.9953 - val_loss: 0.3332 - val_accuracy: 0.9413
Epoch 96/100
200/200 [=====] - 1s 5ms/step - loss: 0.0131 -
accuracy: 0.9956 - val_loss: 0.3520 - val_accuracy: 0.9394
Epoch 97/100
200/200 [=====] - 1s 4ms/step - loss: 0.0033 -
accuracy: 0.9992 - val_loss: 0.3399 - val_accuracy: 0.9400
Epoch 98/100
200/200 [=====] - 1s 4ms/step - loss: 0.0057 -

```

```

accuracy: 0.9980 - val_loss: 0.3543 - val_accuracy: 0.9394
Epoch 99/100
200/200 [=====] - 1s 5ms/step - loss: 0.0037 -
accuracy: 0.9986 - val_loss: 0.3636 - val_accuracy: 0.9432
Epoch 100/100
200/200 [=====] - 1s 5ms/step - loss: 0.0020 -
accuracy: 0.9997 - val_loss: 0.3848 - val_accuracy: 0.9438
Test results - Loss: 0.299445241689682 - Accuracy: 96.09215259552002%

```

```

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

```

```

108/108 [=====] - 1s 2ms/step

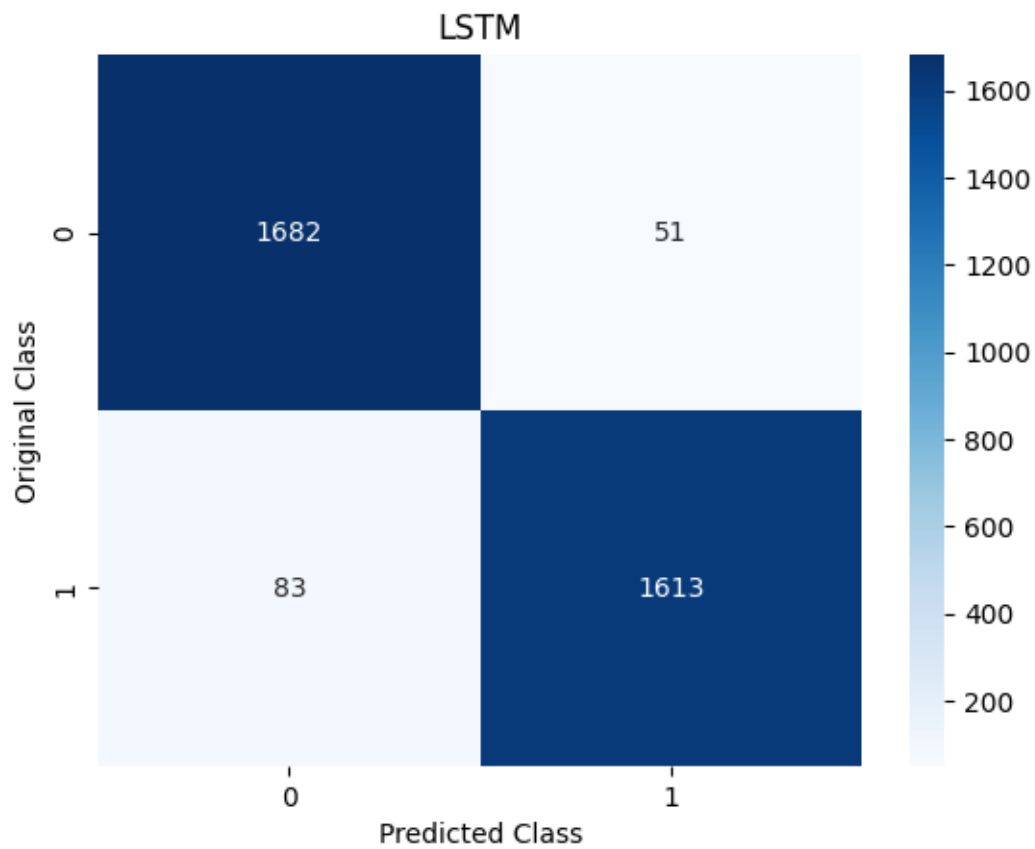
```

	precision	recall	f1-score	support
0	0.95	0.97	0.96	1733
1	0.97	0.95	0.96	1696
accuracy			0.96	3429
macro avg	0.96	0.96	0.96	3429
weighted avg	0.96	0.96	0.96	3429

```

[ ]: sns.heatmap(confusion_matrix(y_test, 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()
```

