

correlation\_target\_label\_30\_9010\_split .03

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, 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_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_registration_length	0.225789
9	SFH	0.221419
10	Google_Index	0.128950
11	age_of_domain	0.121496
12	Page_Rank	0.104645
13	having_IPhaving_IP_Address	0.094160
14	Statistical_report	0.079857
15	DNSRecord	0.075718
16	Shortining_Service	0.067966
17	Abnormal_URL	0.060488
18	URLURL_Length	0.057430
19	having_At_Symbol	0.052948
20	on_mouseover	0.041838
21	HTTPS_token	0.039854
22	double_slash_redirecting	0.038608
23	port	0.036419
24	Links_pointing_to_page	0.032574
25	Redirect	0.020113
26	Submitting_to_email	0.018249
27	RightClick	0.012653
28	Iframe	0.003394
29	Favicon	0.000280
30	popUpWidnow	0.000086

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

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

[ ]: 6

[ ]: corr_feature

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

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

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

[ ]: len(df.columns)

[ ]: 25

[ ]: #df.head()

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

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

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

[ ]: # df.corr()

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

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

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

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

```
[ ]: # corr_features
```

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

```
[ ]: #from sklearn import preprocessing

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

# lab_en= preprocessing.LabelEncoder()

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

# df.head()
```

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

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

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

```
[ ]: Result
0      0
```

```
1      0
2      0
3      0
4      1
```

```
[ ]: # separate dataset into train and test
from cProfile import label
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.1,
    random_state=10)

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

```
[ ]: ((9949, 24), (1106, 24), (9949, 1), (1106, 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 9949 samples.

Testing set has 1106 samples.

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

# defining parameter range
param_grid = {'penalty' : ['l2'],
              'C' : [0.1, 1, 10, 20, 30],
              'solver' : ['lbfgs', 'newton-cg', 'liblinear', 'sag', 'saga'],
              'max_iter' : [2500, 5000]}

grid_logr = GridSearchCV(LogisticRegression(), param_grid, refit = True, cv = 10, verbose = 3, n_jobs = -1)

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

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

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

```
Fitting 10 folds for each of 50 candidates, totalling 500 fits
{'C': 1, 'max_iter': 2500, 'penalty': 'l2', 'solver': 'lbfgs'}
LogisticRegression(C=1, max_iter=2500)
0.9266265937332537
```

```
[ ]: logr_model = grid_logr.best_estimator_

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

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

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

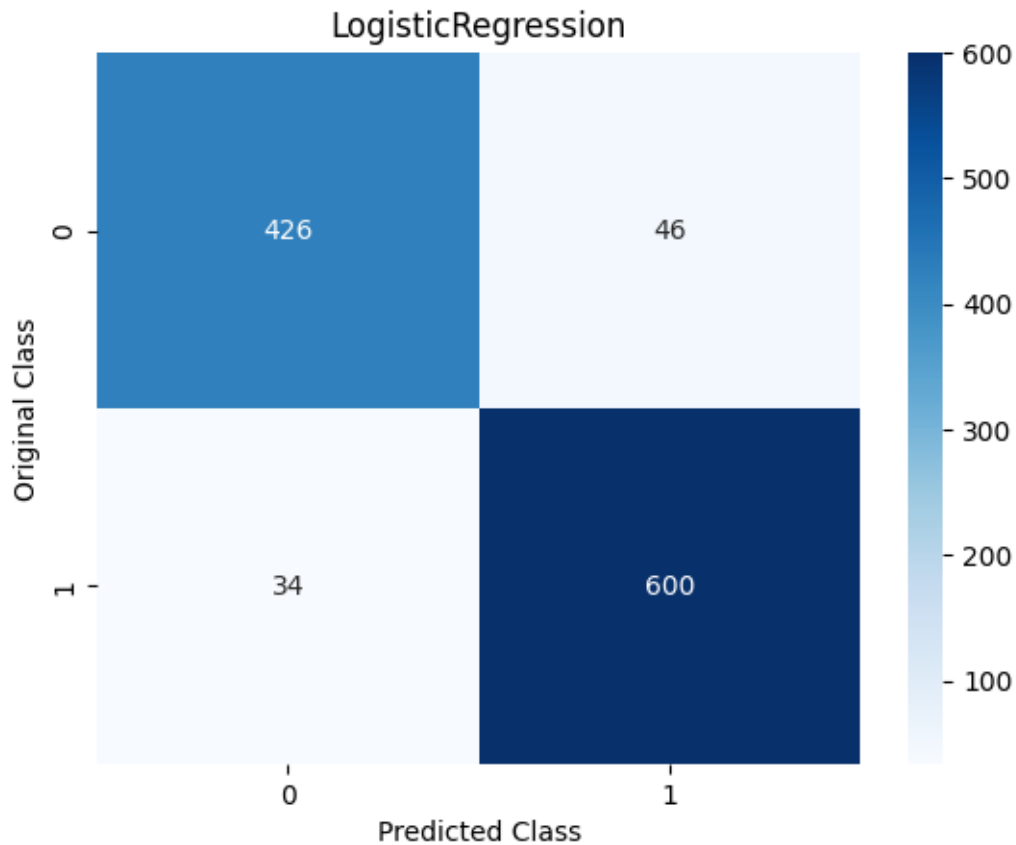
```
[ ]: print ("Accuracy of logr classifier : ", accuracy_score(y_test,
↳logr_predict)*100)
```

Accuracy of logr classifier : 92.76672694394213

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

	precision	recall	f1-score	support
0	0.93	0.90	0.91	472
1	0.93	0.95	0.94	634
accuracy			0.93	1106
macro avg	0.93	0.92	0.93	1106
weighted avg	0.93	0.93	0.93	1106

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



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

# #training_accuracy=[]
# test_accuracy=[]

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

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

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

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

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

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

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

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

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits
{'n_neighbors': 1}
KNeighborsClassifier(n_neighbors=1)
0.9614034963550144
```

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

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

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

```
The accuracy of knn Classifier is: 96.9258589511754
```

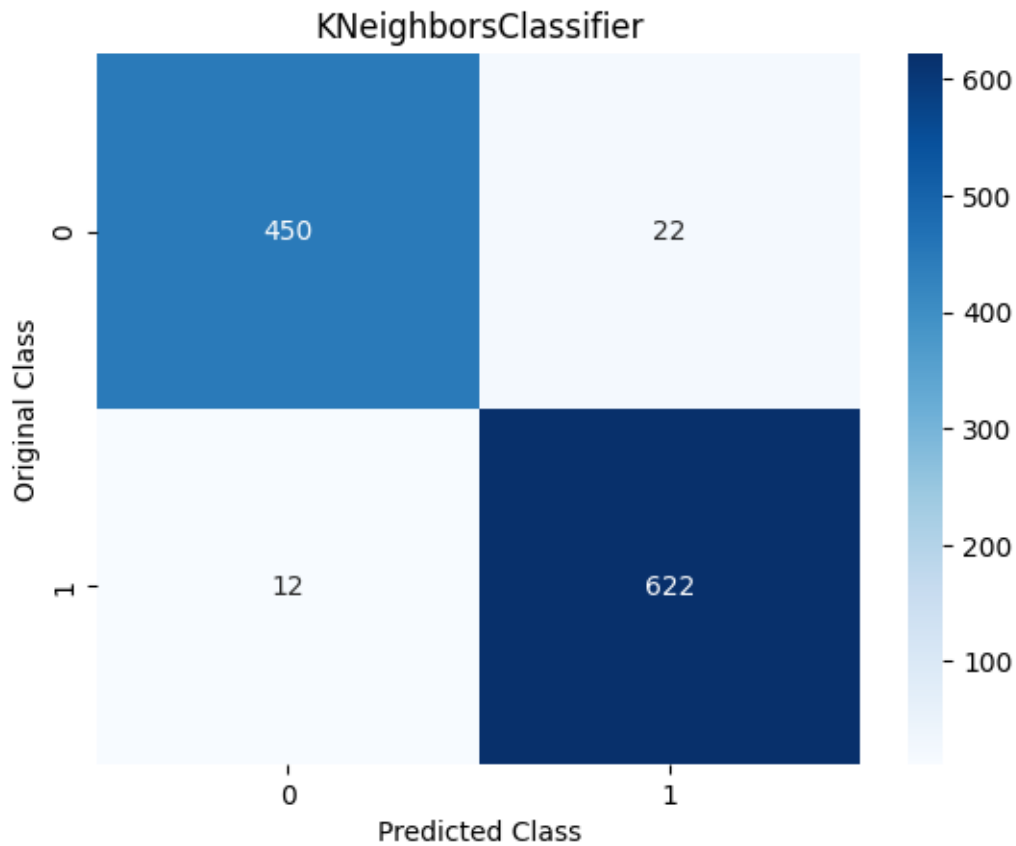
```
[ ]: print(classification_report(y_test, knn_predict))
```

	precision	recall	f1-score	support
0	0.97	0.95	0.96	472
1	0.97	0.98	0.97	634
accuracy			0.97	1106
macro avg	0.97	0.97	0.97	1106
weighted avg	0.97	0.97	0.97	1106

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



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



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

# RocCurveDisplay.from_predictions(y_test,knn_y_pred_proba_positive)

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

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

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

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

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

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

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

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

Fitting 10 folds for each of 36 candidates, totalling 360 fits  
{'C': 10, 'gamma': 1, 'kernel': 'rbf'}  
SVC(C=10, gamma=1)  
0.9660264097145689

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

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

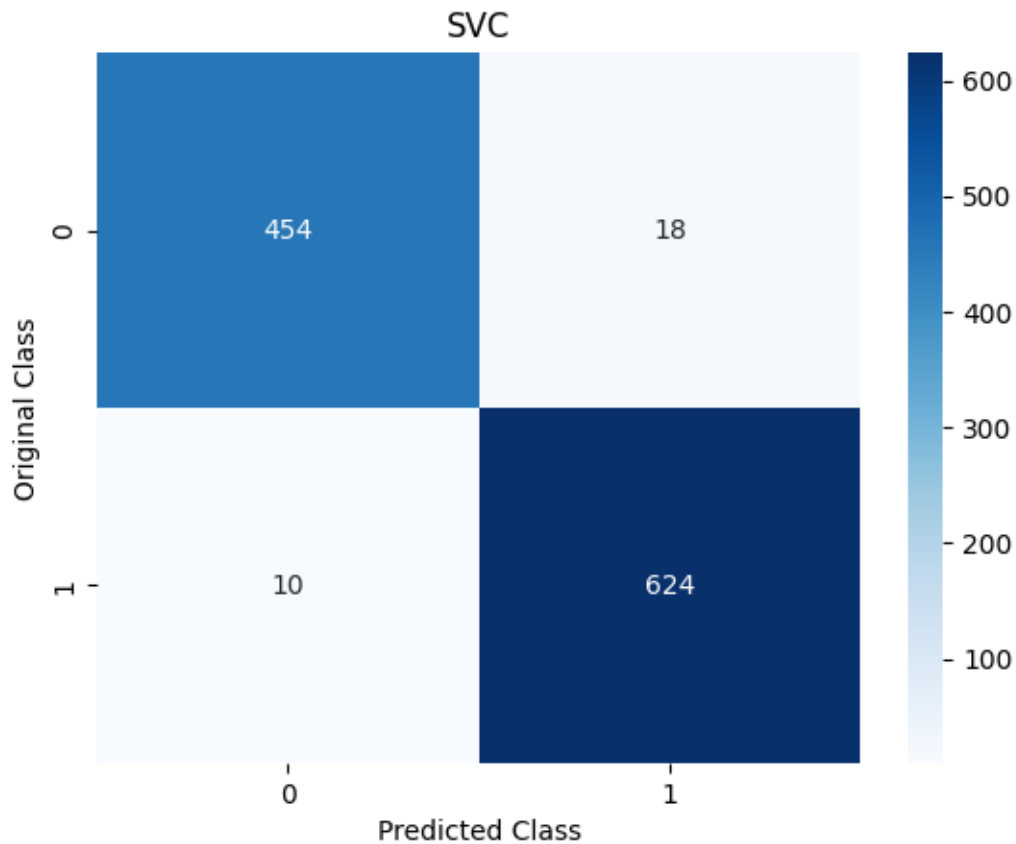
```
[ ]: print('The accuracy of svc Classifier is: ', 100.0 * accuracy_score(y_test,
↪svc_predict))
```

The accuracy of svc Classifier is: 97.46835443037975

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

	precision	recall	f1-score	support
0	0.98	0.96	0.97	472
1	0.97	0.98	0.98	634
accuracy			0.97	1106
macro avg	0.98	0.97	0.97	1106
weighted avg	0.97	0.97	0.97	1106

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



```
[ ]: from sklearn.svm import NuSVC  
  
# defining parameter range  
param_grid = {'nu': [0.1, 0.5],  
              'gamma': [1, 0.1, 0.01],  
              'kernel': ['linear', 'poly', 'rbf', 'sigmoid']}  
  
grid_nusvc = GridSearchCV(NuSVC(), param_grid, refit = True, verbose = 3, cv =  
    10, n_jobs = -1)  
  
# fitting the model for grid search  
grid_nusvc.fit(X_train, y_train.values.ravel())
```

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

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

```
Fitting 10 folds for each of 24 candidates, totalling 240 fits
{'gamma': 1, 'kernel': 'rbf', 'nu': 0.1}
NuSVC(gamma=1, nu=0.1)
0.9661269122271315
```

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

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

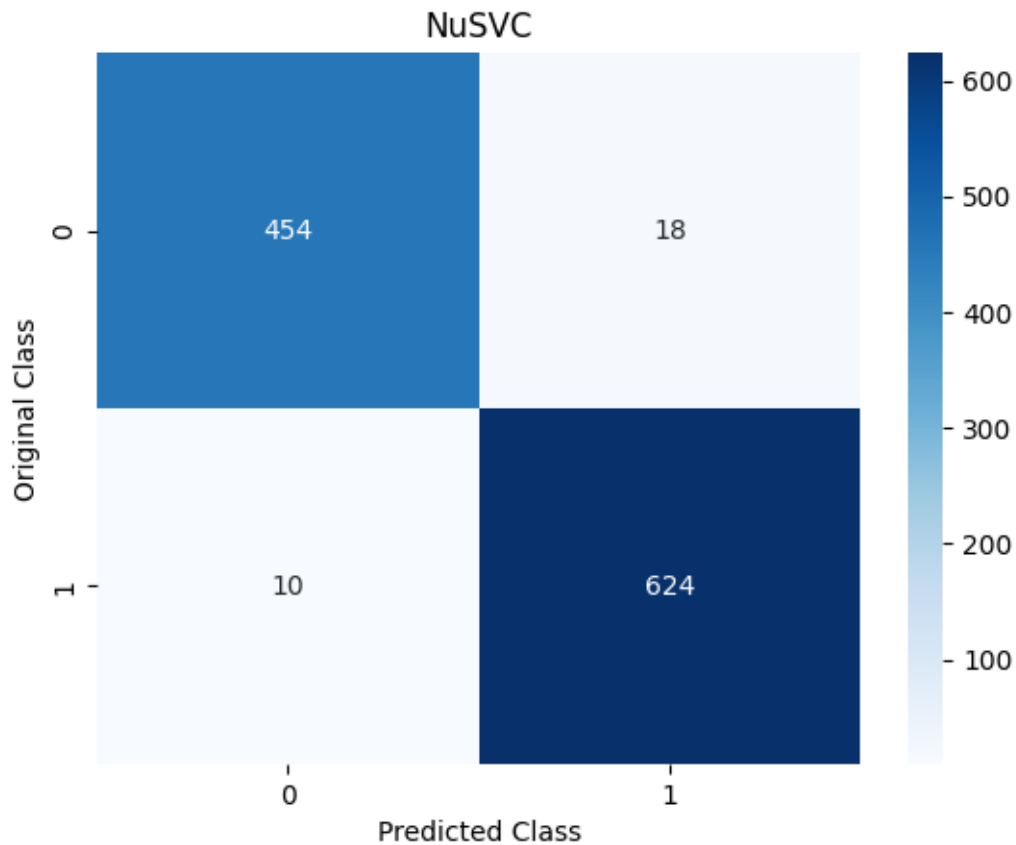
```
[ ]: print('The accuracy of nusvc Classifier is: ', 100.0 * accuracy_score(y_test,
↪nusvc_predict))
```

```
The accuracy of nusvc Classifier is: 97.46835443037975
```

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

	precision	recall	f1-score	support
0	0.98	0.96	0.97	472
1	0.97	0.98	0.98	634
accuracy			0.97	1106
macro avg	0.98	0.97	0.97	1106
weighted avg	0.97	0.97	0.97	1106

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



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

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

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

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

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

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

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

Fitting 10 folds for each of 30 candidates, totalling 300 fits  
{'C': 10, 'dual': False, 'loss': 'squared\_hinge', 'penalty': 'l2', 'tol': 0.001}  
LinearSVC(C=10, dual=False, tol=0.001)  
0.9270286037835049

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

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

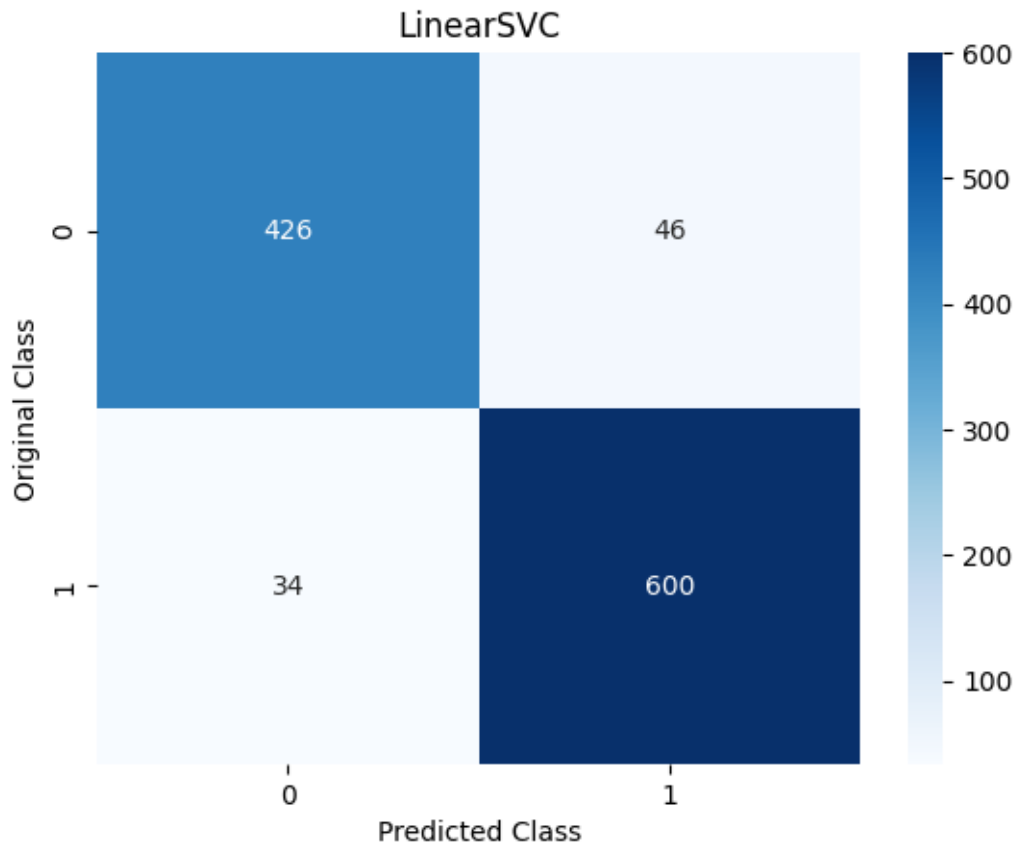
```
[ ]: print('The accuracy of lsvc Classifier is: ', 100.0 * accuracy_score(y_test,
      ↪lsvc_predict))
```

The accuracy of lsvc Classifier is: 92.76672694394213

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

	precision	recall	f1-score	support
0	0.93	0.90	0.91	472
1	0.93	0.95	0.94	634
accuracy			0.93	1106
macro avg	0.93	0.92	0.93	1106
weighted avg	0.93	0.93	0.93	1106

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



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

# defining parameter range
param_grid = {'n_estimators': [40,50,100,200,300]}

grid_ada = GridSearchCV(AdaBoostClassifier(), param_grid, refit = True, verbose=
    ↪ 3, cv = 10, n_jobs = -1)

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

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

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

Fitting 10 folds for each of 5 candidates, totalling 50 fits  
 {'n\_estimators': 100}

```
AdaBoostClassifier(n_estimators=100)
0.9353704134353862
```

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

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

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

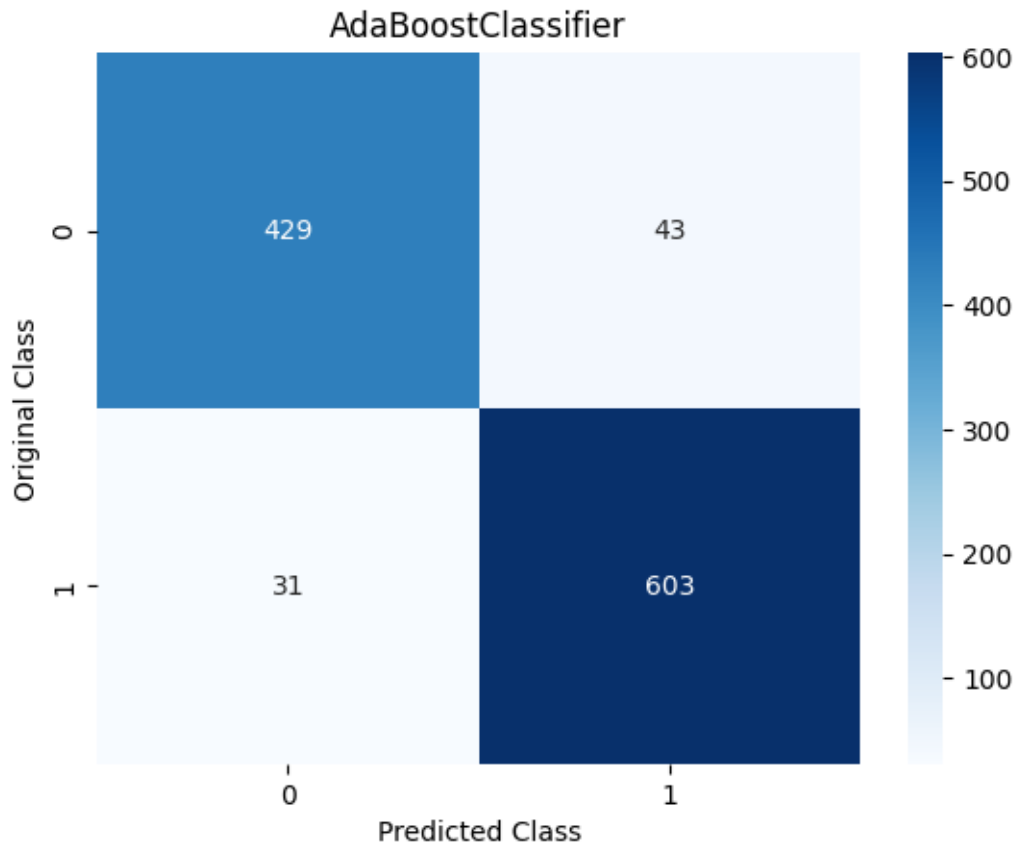
The accuracy of Ada Boost Classifier is: 93.30922242314648

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

	precision	recall	f1-score	support
0	0.93	0.91	0.92	472
1	0.93	0.95	0.94	634
accuracy			0.93	1106
macro avg	0.93	0.93	0.93	1106
weighted avg	0.93	0.93	0.93	1106

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





```
[ ]: from xgboost import XGBClassifier

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

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

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

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

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

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

Fitting 10 folds for each of 15 candidates, totalling 150 fits

```
{'gamma': 0.1, '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, ...)
```

0.9734640000808874

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

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

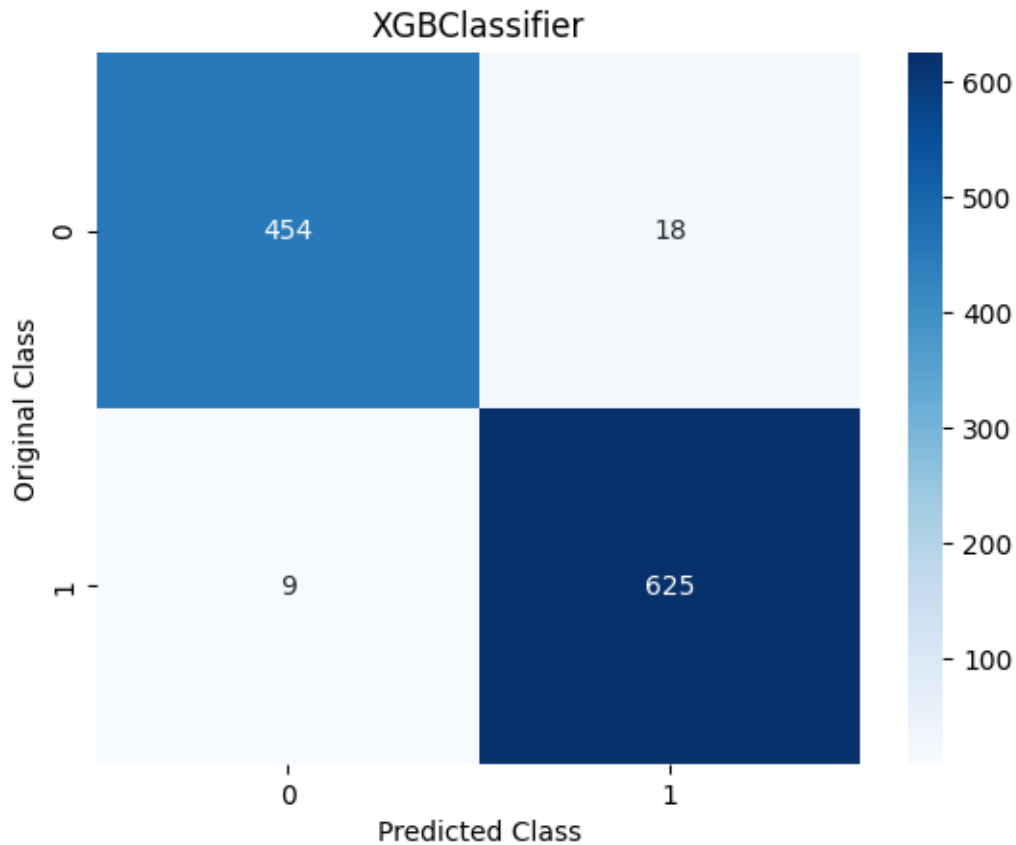
```
[ ]: print('The accuracy of XGBoost Classifier is: ', 100.0 *
      ↪accuracy_score(xgb_predict,y_test))
```

The accuracy of XGBoost Classifier is: 97.55877034358048

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

	precision	recall	f1-score	support
0	0.98	0.96	0.97	472
1	0.97	0.99	0.98	634
accuracy			0.98	1106
macro avg	0.98	0.97	0.97	1106
weighted avg	0.98	0.98	0.98	1106

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



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

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

grid_gbc = GridSearchCV(GradientBoostingClassifier(), param_grid, refit = True,
    verbose = 3, cv = 10, n_jobs = -1)

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

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

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

```
print(grid_gbc.best_score_)
```

Fitting 10 folds for each of 15 candidates, totalling 150 fits  
{'learning\_rate': 1, 'n\_estimators': 200}  
GradientBoostingClassifier(learning\_rate=1, n\_estimators=200)  
0.9688399745204898

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

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

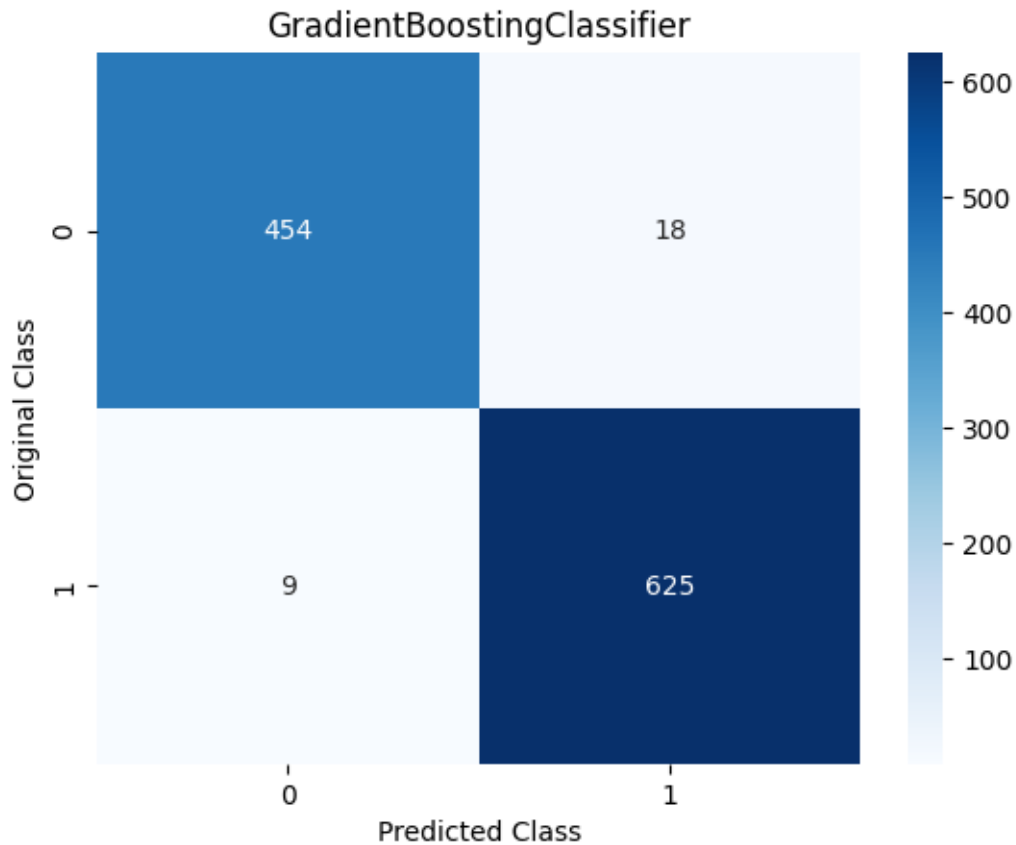
```
[ ]: print('The accuracy of GradientBoost Classifier is: ', 100.0 *  
      ↪accuracy_score(gbc_predict,y_test))
```

The accuracy of GradientBoost Classifier is: 97.55877034358048

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

	precision	recall	f1-score	support
0	0.98	0.96	0.97	472
1	0.97	0.99	0.98	634
accuracy			0.98	1106
macro avg	0.98	0.97	0.97	1106
weighted avg	0.98	0.98	0.98	1106

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



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

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

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

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

# defining parameter range
param_grid = {
    "base_estimator": [DecisionTreeClassifier()],
    "n_estimators": [50,100,150,200,250]
```

```

}

grid_bag = GridSearchCV(BaggingClassifier(), param_grid, refit = True, verbose_
↳ = 3, cv = 10, n_jobs = -1)

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

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

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

```

```

Fitting 10 folds for each of 5 candidates, totalling 50 fits
{'base_estimator': DecisionTreeClassifier(), 'n_estimators': 150}
BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=150)
0.9688401767388249

```

```

[ ]: 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: 97.64918625678119

```

```

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

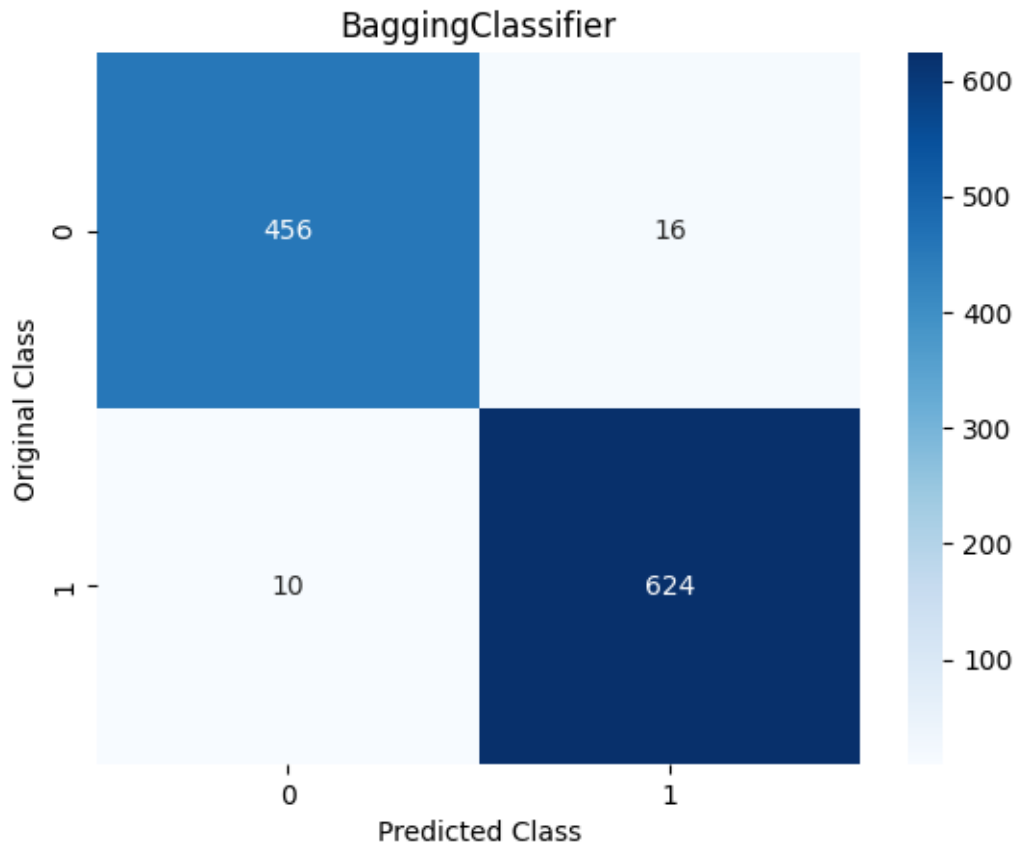
```

	precision	recall	f1-score	support
0	0.98	0.97	0.97	472
1	0.97	0.98	0.98	634
accuracy			0.98	1106
macro avg	0.98	0.98	0.98	1106
weighted avg	0.98	0.98	0.98	1106

```

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

```



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

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

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

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

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

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

```
Fitting 10 folds for each of 5 candidates, totalling 50 fits
{'n_estimators': 250}
RandomForestClassifier(n_estimators=250)
0.9704488235948354
```

```
[ ]: 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))
```

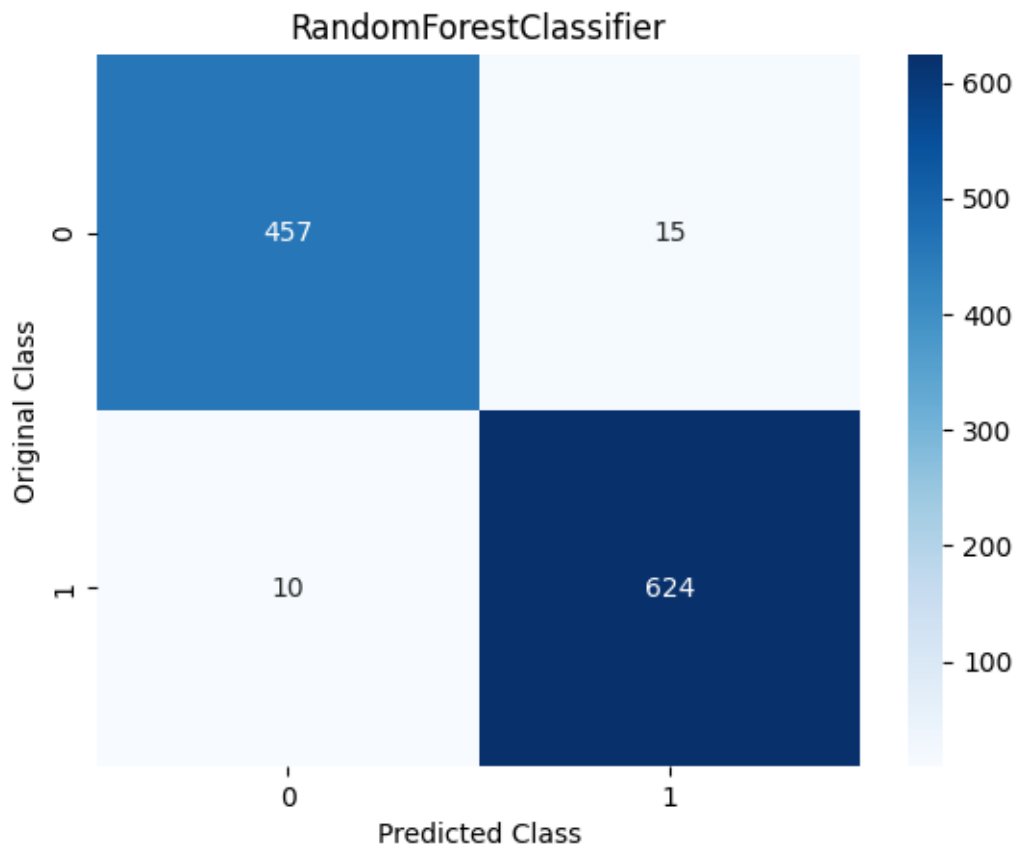
The accuracy of RandomForest Classifier is: 97.73960216998192

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

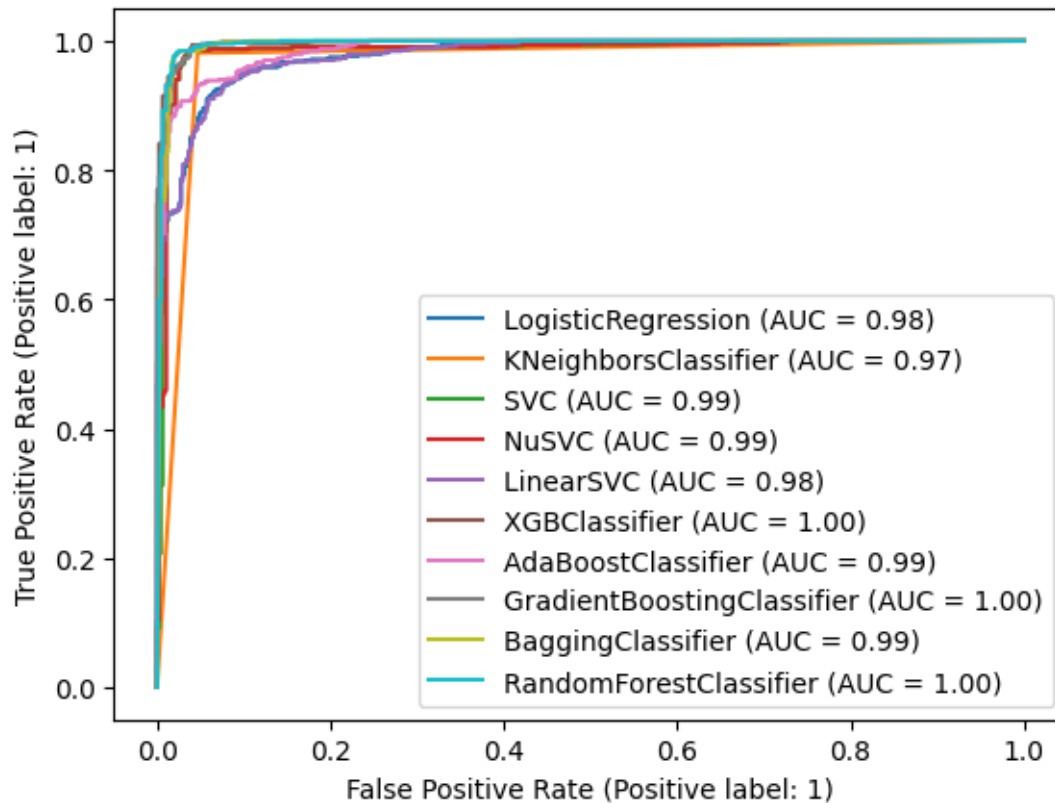
	precision	recall	f1-score	support
0	0.98	0.97	0.97	472
1	0.98	0.98	0.98	634
accuracy			0.98	1106
macro avg	0.98	0.98	0.98	1106
weighted avg	0.98	0.98	0.98	1106

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





```
[ ]: estimators =  
    ↳ [logr_model, knn_model, svc_model, nusvc_model, lsvc_model, xgb_model, ada_model, gbc_model, bag_mo  
  
for estimator in estimators:  
    RocCurveDisplay.from_estimator(estimator, X_test, y_test, ax=plt.gca())
```



```
[ ]: import tensorflow as tf
      #from tensorflow.keras.datasets import imdb
      from keras.layers import Embedding, Dense, LSTM, BatchNormalization
      from keras.losses import BinaryCrossentropy
      from keras.models import Sequential
      from keras.optimizers import Adam
      #from tensorflow.keras.preprocessing.sequence import pad_sequences

      # Model configuration
      additional_metrics = ['accuracy']
      batch_size = 32
      #embedding_output_dims = (X_train.shape[1])
      loss_function = BinaryCrossentropy()
      #max_sequence_length = (X_train.shape[1])
      #num_distinct_words = (X_train.shape[1])
      number_of_epochs = 100
      optimizer = Adam()
      validation_split = 0.20
      verbosity_mode = 1

      # reshape from [samples, features] into [samples, timesteps, features]
```

```

timesteps = 1
X_train_reshape = X_train.values.ravel().reshape(X_train.shape[0],timesteps,
↳X_train.shape[1])
X_test_reshape = X_test.values.ravel().reshape(X_test.shape[0],timesteps,
↳X_test.shape[1])

# Disable eager execution
#tf.compat.v1.disable_eager_execution()

# Load dataset
# (x_train, y_train), (x_test, y_test) = imdb.
↳load_data(num_words=num_distinct_words)
# print(x_train.shape)
# print(x_test.shape)

# Pad all sequences
# padded_inputs = pad_sequences(X_train, maxlen=max_sequence_length, value = 0.
↳0) # 0.0 because it corresponds with <PAD>
# padded_inputs_test = pad_sequences(X_test, maxlen=max_sequence_length, value
↳= 0.0) # 0.0 because it corresponds with <PAD>

# Define the Keras model
def build_model_lstm():
    model = Sequential()
    #model.add(Embedding(num_distinct_words, embedding_output_dims,
↳input_length=max_sequence_length))
    model.add(LSTM(100, input_shape = (timesteps,X_train_reshape.shape[2])))
    model.add(BatchNormalization())
    model.add(Dense(50, activation='relu'))
    model.add(Dense(25, activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    # Compile the model
    model.compile(optimizer=optimizer, loss=loss_function,
↳metrics=additional_metrics)
    return model

#from keras.wrappers.scikit_learn import KerasClassifier
lstm_model = build_model_lstm()
# Give a summary
lstm_model.summary()

# Train the model

```

```

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

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

```

Model: "sequential\_2"

Layer (type)	Output Shape	Param #
lstm_2 (LSTM)	(None, 100)	50000
batch_normalization_2 (Batch Normalization)	(None, 100)	400

Layer (type)	Output Shape	Param #
lstm_2 (LSTM)	(None, 100)	50000
batch_normalization_2 (Batch Normalization)	(None, 100)	400
dense_8 (Dense)	(None, 50)	5050
dense_9 (Dense)	(None, 25)	1275
dense_10 (Dense)	(None, 10)	260
dense_11 (Dense)	(None, 1)	11

```

=====
Total params: 56,996
Trainable params: 56,796
Non-trainable params: 200

```

```

-----
Epoch 1/100
249/249 [=====] - 4s 7ms/step - loss: 0.4093 -
accuracy: 0.8862 - val_loss: 0.4898 - val_accuracy: 0.9412
Epoch 2/100
249/249 [=====] - 1s 5ms/step - loss: 0.3350 -
accuracy: 0.9327 - val_loss: 0.3219 - val_accuracy: 0.9447

```

Epoch 3/100  
249/249 [=====] - 1s 4ms/step - loss: 0.2501 -  
accuracy: 0.9334 - val\_loss: 0.1426 - val\_accuracy: 0.9472  
Epoch 4/100  
249/249 [=====] - 1s 4ms/step - loss: 0.1375 -  
accuracy: 0.9448 - val\_loss: 0.1327 - val\_accuracy: 0.9482  
Epoch 5/100  
249/249 [=====] - 1s 4ms/step - loss: 0.1270 -  
accuracy: 0.9472 - val\_loss: 0.1217 - val\_accuracy: 0.9452  
Epoch 6/100  
249/249 [=====] - 1s 4ms/step - loss: 0.1183 -  
accuracy: 0.9521 - val\_loss: 0.1532 - val\_accuracy: 0.9467  
Epoch 7/100  
249/249 [=====] - 1s 4ms/step - loss: 0.1094 -  
accuracy: 0.9534 - val\_loss: 0.1145 - val\_accuracy: 0.9487  
Epoch 8/100  
249/249 [=====] - 1s 4ms/step - loss: 0.1026 -  
accuracy: 0.9593 - val\_loss: 0.1204 - val\_accuracy: 0.9528  
Epoch 9/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0959 -  
accuracy: 0.9593 - val\_loss: 0.1154 - val\_accuracy: 0.9568  
Epoch 10/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0935 -  
accuracy: 0.9612 - val\_loss: 0.1074 - val\_accuracy: 0.9583  
Epoch 11/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0840 -  
accuracy: 0.9672 - val\_loss: 0.1071 - val\_accuracy: 0.9573  
Epoch 12/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0839 -  
accuracy: 0.9634 - val\_loss: 0.1243 - val\_accuracy: 0.9558  
Epoch 13/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0826 -  
accuracy: 0.9646 - val\_loss: 0.1116 - val\_accuracy: 0.9588  
Epoch 14/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0839 -  
accuracy: 0.9648 - val\_loss: 0.1082 - val\_accuracy: 0.9628  
Epoch 15/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0765 -  
accuracy: 0.9691 - val\_loss: 0.1139 - val\_accuracy: 0.9563  
Epoch 16/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0737 -  
accuracy: 0.9681 - val\_loss: 0.1115 - val\_accuracy: 0.9633  
Epoch 17/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0770 -  
accuracy: 0.9676 - val\_loss: 0.1199 - val\_accuracy: 0.9588  
Epoch 18/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0748 -  
accuracy: 0.9688 - val\_loss: 0.1160 - val\_accuracy: 0.9598

Epoch 19/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0715 -  
accuracy: 0.9702 - val\_loss: 0.1082 - val\_accuracy: 0.9603  
Epoch 20/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0714 -  
accuracy: 0.9688 - val\_loss: 0.1021 - val\_accuracy: 0.9643  
Epoch 21/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0652 -  
accuracy: 0.9735 - val\_loss: 0.1071 - val\_accuracy: 0.9608  
Epoch 22/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0657 -  
accuracy: 0.9727 - val\_loss: 0.1025 - val\_accuracy: 0.9623  
Epoch 23/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0616 -  
accuracy: 0.9746 - val\_loss: 0.1092 - val\_accuracy: 0.9588  
Epoch 24/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0615 -  
accuracy: 0.9741 - val\_loss: 0.1046 - val\_accuracy: 0.9578  
Epoch 25/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0606 -  
accuracy: 0.9751 - val\_loss: 0.1166 - val\_accuracy: 0.9588  
Epoch 26/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0611 -  
accuracy: 0.9736 - val\_loss: 0.1284 - val\_accuracy: 0.9583  
Epoch 27/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0578 -  
accuracy: 0.9750 - val\_loss: 0.1117 - val\_accuracy: 0.9623  
Epoch 28/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0642 -  
accuracy: 0.9737 - val\_loss: 0.1144 - val\_accuracy: 0.9643  
Epoch 29/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0576 -  
accuracy: 0.9751 - val\_loss: 0.0931 - val\_accuracy: 0.9643  
Epoch 30/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0574 -  
accuracy: 0.9739 - val\_loss: 0.1084 - val\_accuracy: 0.9653  
Epoch 31/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0585 -  
accuracy: 0.9763 - val\_loss: 0.1179 - val\_accuracy: 0.9598  
Epoch 32/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0560 -  
accuracy: 0.9773 - val\_loss: 0.1029 - val\_accuracy: 0.9638  
Epoch 33/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0564 -  
accuracy: 0.9747 - val\_loss: 0.1000 - val\_accuracy: 0.9709  
Epoch 34/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0512 -  
accuracy: 0.9786 - val\_loss: 0.1032 - val\_accuracy: 0.9663

Epoch 35/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0493 -  
accuracy: 0.9780 - val\_loss: 0.1325 - val\_accuracy: 0.9618  
Epoch 36/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0499 -  
accuracy: 0.9785 - val\_loss: 0.1188 - val\_accuracy: 0.9648  
Epoch 37/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0550 -  
accuracy: 0.9758 - val\_loss: 0.1072 - val\_accuracy: 0.9663  
Epoch 38/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0532 -  
accuracy: 0.9776 - val\_loss: 0.1075 - val\_accuracy: 0.9623  
Epoch 39/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0508 -  
accuracy: 0.9770 - val\_loss: 0.1202 - val\_accuracy: 0.9653  
Epoch 40/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0477 -  
accuracy: 0.9790 - val\_loss: 0.0954 - val\_accuracy: 0.9658  
Epoch 41/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0490 -  
accuracy: 0.9795 - val\_loss: 0.1109 - val\_accuracy: 0.9673  
Epoch 42/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0495 -  
accuracy: 0.9795 - val\_loss: 0.1256 - val\_accuracy: 0.9618  
Epoch 43/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0460 -  
accuracy: 0.9810 - val\_loss: 0.1151 - val\_accuracy: 0.9673  
Epoch 44/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0463 -  
accuracy: 0.9803 - val\_loss: 0.0995 - val\_accuracy: 0.9673  
Epoch 45/100  
249/249 [=====] - 1s 6ms/step - loss: 0.0456 -  
accuracy: 0.9786 - val\_loss: 0.1326 - val\_accuracy: 0.9683  
Epoch 46/100  
249/249 [=====] - 2s 7ms/step - loss: 0.0487 -  
accuracy: 0.9786 - val\_loss: 0.1063 - val\_accuracy: 0.9643  
Epoch 47/100  
249/249 [=====] - 2s 6ms/step - loss: 0.0439 -  
accuracy: 0.9828 - val\_loss: 0.1193 - val\_accuracy: 0.9648  
Epoch 48/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0427 -  
accuracy: 0.9814 - val\_loss: 0.1086 - val\_accuracy: 0.9648  
Epoch 49/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0417 -  
accuracy: 0.9824 - val\_loss: 0.1193 - val\_accuracy: 0.9663  
Epoch 50/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0461 -  
accuracy: 0.9810 - val\_loss: 0.0979 - val\_accuracy: 0.9643

Epoch 51/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0389 -  
accuracy: 0.9833 - val\_loss: 0.1049 - val\_accuracy: 0.9698  
Epoch 52/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0453 -  
accuracy: 0.9808 - val\_loss: 0.1320 - val\_accuracy: 0.9668  
Epoch 53/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0427 -  
accuracy: 0.9809 - val\_loss: 0.1053 - val\_accuracy: 0.9653  
Epoch 54/100  
249/249 [=====] - 1s 6ms/step - loss: 0.0436 -  
accuracy: 0.9820 - val\_loss: 0.1137 - val\_accuracy: 0.9668  
Epoch 55/100  
249/249 [=====] - 2s 7ms/step - loss: 0.0436 -  
accuracy: 0.9822 - val\_loss: 0.1267 - val\_accuracy: 0.9658  
Epoch 56/100  
249/249 [=====] - 2s 8ms/step - loss: 0.0435 -  
accuracy: 0.9814 - val\_loss: 0.1215 - val\_accuracy: 0.9633  
Epoch 57/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0411 -  
accuracy: 0.9829 - val\_loss: 0.0982 - val\_accuracy: 0.9688  
Epoch 58/100  
249/249 [=====] - 1s 6ms/step - loss: 0.0405 -  
accuracy: 0.9817 - val\_loss: 0.1177 - val\_accuracy: 0.9658  
Epoch 59/100  
249/249 [=====] - 2s 7ms/step - loss: 0.0442 -  
accuracy: 0.9795 - val\_loss: 0.1265 - val\_accuracy: 0.9673  
Epoch 60/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0391 -  
accuracy: 0.9827 - val\_loss: 0.1074 - val\_accuracy: 0.9678  
Epoch 61/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0401 -  
accuracy: 0.9828 - val\_loss: 0.1217 - val\_accuracy: 0.9663  
Epoch 62/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0420 -  
accuracy: 0.9796 - val\_loss: 0.1369 - val\_accuracy: 0.9658  
Epoch 63/100  
249/249 [=====] - 2s 7ms/step - loss: 0.0401 -  
accuracy: 0.9823 - val\_loss: 0.1068 - val\_accuracy: 0.9693  
Epoch 64/100  
249/249 [=====] - 2s 8ms/step - loss: 0.0396 -  
accuracy: 0.9829 - val\_loss: 0.1225 - val\_accuracy: 0.9688  
Epoch 65/100  
249/249 [=====] - 1s 6ms/step - loss: 0.0382 -  
accuracy: 0.9829 - val\_loss: 0.1124 - val\_accuracy: 0.9663  
Epoch 66/100  
249/249 [=====] - 2s 7ms/step - loss: 0.0402 -  
accuracy: 0.9833 - val\_loss: 0.1465 - val\_accuracy: 0.9623



Epoch 67/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0426 -  
accuracy: 0.9809 - val\_loss: 0.1638 - val\_accuracy: 0.9618  
Epoch 68/100  
249/249 [=====] - 2s 7ms/step - loss: 0.0428 -  
accuracy: 0.9812 - val\_loss: 0.1276 - val\_accuracy: 0.9678  
Epoch 69/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0412 -  
accuracy: 0.9820 - val\_loss: 0.1206 - val\_accuracy: 0.9663  
Epoch 70/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0368 -  
accuracy: 0.9835 - val\_loss: 0.1167 - val\_accuracy: 0.9698  
Epoch 71/100  
249/249 [=====] - 1s 6ms/step - loss: 0.0348 -  
accuracy: 0.9840 - val\_loss: 0.1220 - val\_accuracy: 0.9623  
Epoch 72/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0364 -  
accuracy: 0.9833 - val\_loss: 0.1547 - val\_accuracy: 0.9648  
Epoch 73/100  
249/249 [=====] - 1s 6ms/step - loss: 0.0397 -  
accuracy: 0.9827 - val\_loss: 0.1179 - val\_accuracy: 0.9633  
Epoch 74/100  
249/249 [=====] - 1s 6ms/step - loss: 0.0369 -  
accuracy: 0.9835 - val\_loss: 0.1220 - val\_accuracy: 0.9658  
Epoch 75/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0368 -  
accuracy: 0.9838 - val\_loss: 0.1062 - val\_accuracy: 0.9658  
Epoch 76/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0374 -  
accuracy: 0.9842 - val\_loss: 0.1726 - val\_accuracy: 0.9668  
Epoch 77/100  
249/249 [=====] - 2s 6ms/step - loss: 0.0391 -  
accuracy: 0.9830 - val\_loss: 0.1312 - val\_accuracy: 0.9683  
Epoch 78/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0387 -  
accuracy: 0.9839 - val\_loss: 0.1318 - val\_accuracy: 0.9643  
Epoch 79/100  
249/249 [=====] - 2s 7ms/step - loss: 0.0388 -  
accuracy: 0.9834 - val\_loss: 0.1149 - val\_accuracy: 0.9683  
Epoch 80/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0365 -  
accuracy: 0.9832 - val\_loss: 0.1352 - val\_accuracy: 0.9688  
Epoch 81/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0404 -  
accuracy: 0.9817 - val\_loss: 0.1259 - val\_accuracy: 0.9628  
Epoch 82/100  
249/249 [=====] - 2s 7ms/step - loss: 0.0366 -  
accuracy: 0.9840 - val\_loss: 0.1294 - val\_accuracy: 0.9638

Epoch 83/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0359 -  
accuracy: 0.9834 - val\_loss: 0.1328 - val\_accuracy: 0.9673  
Epoch 84/100  
249/249 [=====] - 2s 8ms/step - loss: 0.0331 -  
accuracy: 0.9845 - val\_loss: 0.1364 - val\_accuracy: 0.9673  
Epoch 85/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0355 -  
accuracy: 0.9839 - val\_loss: 0.1236 - val\_accuracy: 0.9663  
Epoch 86/100  
249/249 [=====] - 2s 6ms/step - loss: 0.0361 -  
accuracy: 0.9839 - val\_loss: 0.1222 - val\_accuracy: 0.9714  
Epoch 87/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0380 -  
accuracy: 0.9834 - val\_loss: 0.1412 - val\_accuracy: 0.9668  
Epoch 88/100  
249/249 [=====] - 1s 6ms/step - loss: 0.0383 -  
accuracy: 0.9833 - val\_loss: 0.1396 - val\_accuracy: 0.9658  
Epoch 89/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0356 -  
accuracy: 0.9835 - val\_loss: 0.1509 - val\_accuracy: 0.9678  
Epoch 90/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0331 -  
accuracy: 0.9861 - val\_loss: 0.1656 - val\_accuracy: 0.9643  
Epoch 91/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0325 -  
accuracy: 0.9839 - val\_loss: 0.1511 - val\_accuracy: 0.9653  
Epoch 92/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0362 -  
accuracy: 0.9830 - val\_loss: 0.1374 - val\_accuracy: 0.9688  
Epoch 93/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0371 -  
accuracy: 0.9850 - val\_loss: 0.1251 - val\_accuracy: 0.9623  
Epoch 94/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0365 -  
accuracy: 0.9829 - val\_loss: 0.1184 - val\_accuracy: 0.9658  
Epoch 95/100  
249/249 [=====] - 1s 5ms/step - loss: 0.0339 -  
accuracy: 0.9856 - val\_loss: 0.1439 - val\_accuracy: 0.9658  
Epoch 96/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0351 -  
accuracy: 0.9843 - val\_loss: 0.1363 - val\_accuracy: 0.9658  
Epoch 97/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0354 -  
accuracy: 0.9833 - val\_loss: 0.1348 - val\_accuracy: 0.9643  
Epoch 98/100  
249/249 [=====] - 1s 4ms/step - loss: 0.0345 -  
accuracy: 0.9845 - val\_loss: 0.1220 - val\_accuracy: 0.9688

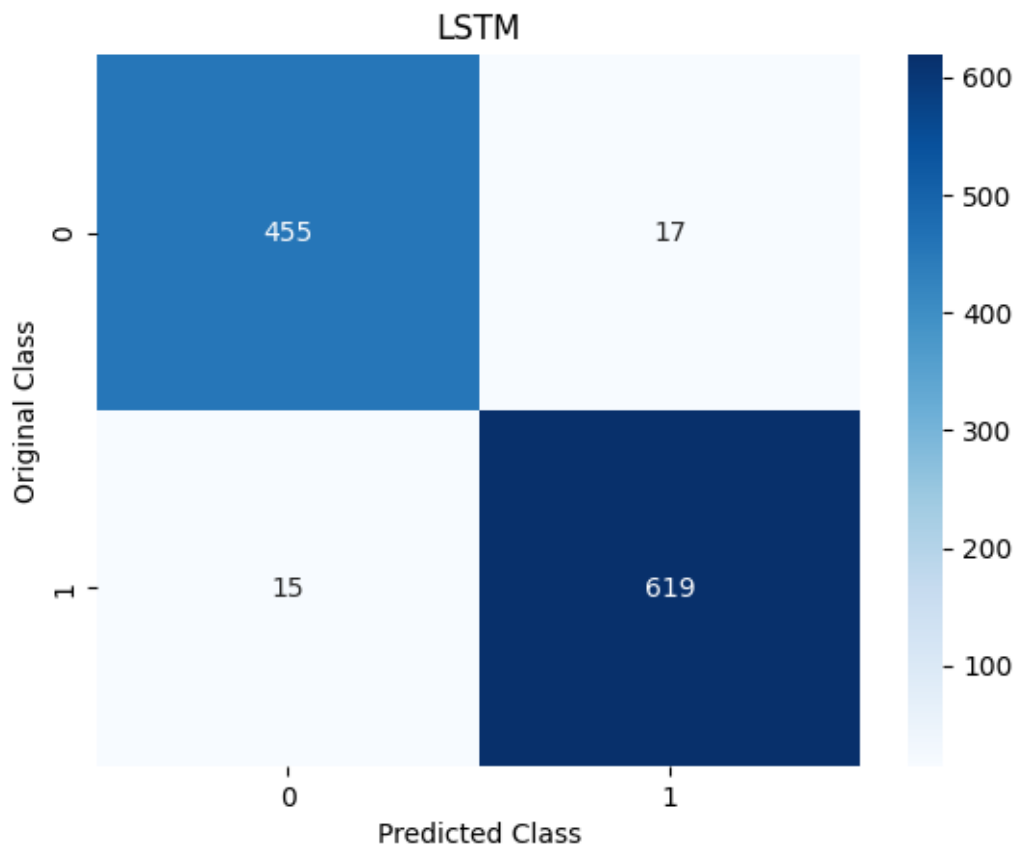
```
Epoch 99/100
249/249 [=====] - 1s 4ms/step - loss: 0.0318 -
accuracy: 0.9839 - val_loss: 0.1359 - val_accuracy: 0.9673
Epoch 100/100
249/249 [=====] - 1s 4ms/step - loss: 0.0360 -
accuracy: 0.9847 - val_loss: 0.1525 - val_accuracy: 0.9648
Test results - Loss: 0.1282627433538437 - Accuracy: 97.10668921470642%
```

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

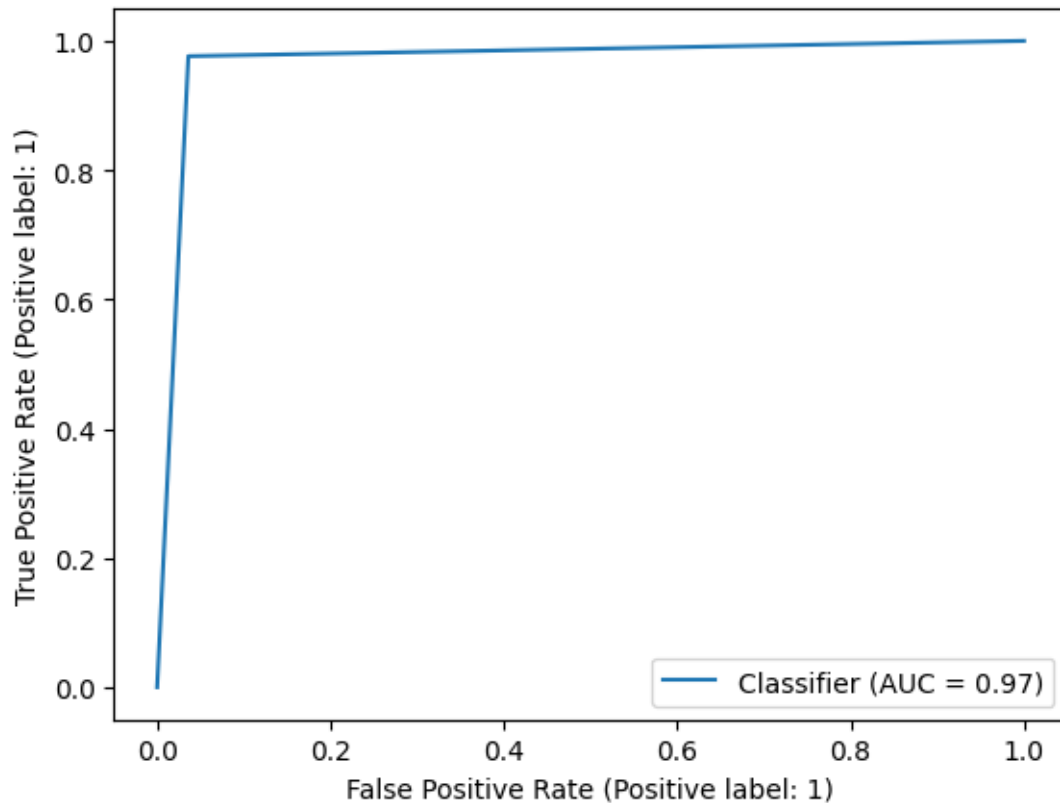
```
35/35 [=====] - 1s 2ms/step
```

	precision	recall	f1-score	support
0	0.97	0.96	0.97	472
1	0.97	0.98	0.97	634
accuracy			0.97	1106
macro avg	0.97	0.97	0.97	1106
weighted avg	0.97	0.97	0.97	1106

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