Assessment 2

Task 1: Develop learning-based model(s) for Classification

In [1]:

```
# TensorFlow config to GPU
import tensorflow as tf
print(tf.__version__)

gpus = tf.config.list_physical_devices('GPU')
if gpus:
    tf.config.set_logical_device_configuration(
        gpus[0],
        [tf.config.LogicalDeviceConfiguration(memory_limit=15292)]
    )

logical_gpus = tf.config.list_logical_devices('GPU')
print(logical_gpus)
print(len(gpus), "Physical GPU,", len(logical_gpus), "Logical GPUs")



from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())

print()
print()

import tensorflow as tf
print("Num GPUs Available: ",
len(tf.config.list_physical_devices('GPU')))
```

```
2.6.0
[LogicalDevice(name='/device:GPU:0', device_type='GPU')]
1 Physical GPU, 1 Logical GPUs
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 8707722222192318641
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 16034824192
locality {
  bus_id: 1
  links {
  }
```

```
}
incarnation: 8088543449005111868
physical_device_desc: "device: 0, name: NVIDIA GeForce RTX 2060, pci bus id: 0000:0
1:00.0, compute capability: 7.5"
]


Num GPUs Available:  1
```

In [2]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')

from sklearn.metrics import precision_recall_curve
from sklearn.metrics import plot_precision_recall_curve
from sklearn.metrics import average_precision_score
from sklearn.metrics import auc
```

In [3]:

```
train_df = pd.read_csv('ECG_dataset/train.csv', header=None)
test_df = pd.read_csv('ECG_dataset/test.csv', header=None)
validation_df = pd.read_csv('ECG_dataset/validation.csv', header=None)
```

In [4]:

```
print(train_df.shape)
print(test_df.shape)
print(validation_df.shape)
```

```
(1081, 141)
(180, 141)
(541, 141)
```

We will be using train_df as our training data and validation_df as our validation data. We will be using test_df as our test data.

In [5]:

```
train_df.head()
```

Out[5]:

	0	1	2	3	4	5	6	7	8	
0	1	0.024133	0.016065	0.044639	0.031001	-0.009473	-0.042663	-0.077283	-0.091508	-0.04611
1	0	0.424380	0.344420	0.348130	0.340170	0.243370	0.241730	0.268780	0.273420	0.35644
2	0	1.529500	1.776600	1.936700	1.840200	1.800000	1.724900	1.405800	1.008800	0.72472

	0	1	2	3	4	5	6	7	8	
3	0	1.286500	1.049900	0.793600	0.473590	0.111730	-0.054857	-0.062095	-0.120750	-0.10301
4	1	-0.175400	-0.121920	-0.053532	-0.024293	0.022917	0.116440	0.187040	0.240710	0.31434

5 rows × 141 columns

In [6]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1081 entries, 0 to 1080
Columns: 141 entries, 0 to 140
dtypes: float64(140), int64(1)
memory usage: 1.2 MB
```

In [7]:
```
train_df.shape
```

Out[7]:
```
(1081, 141)
```

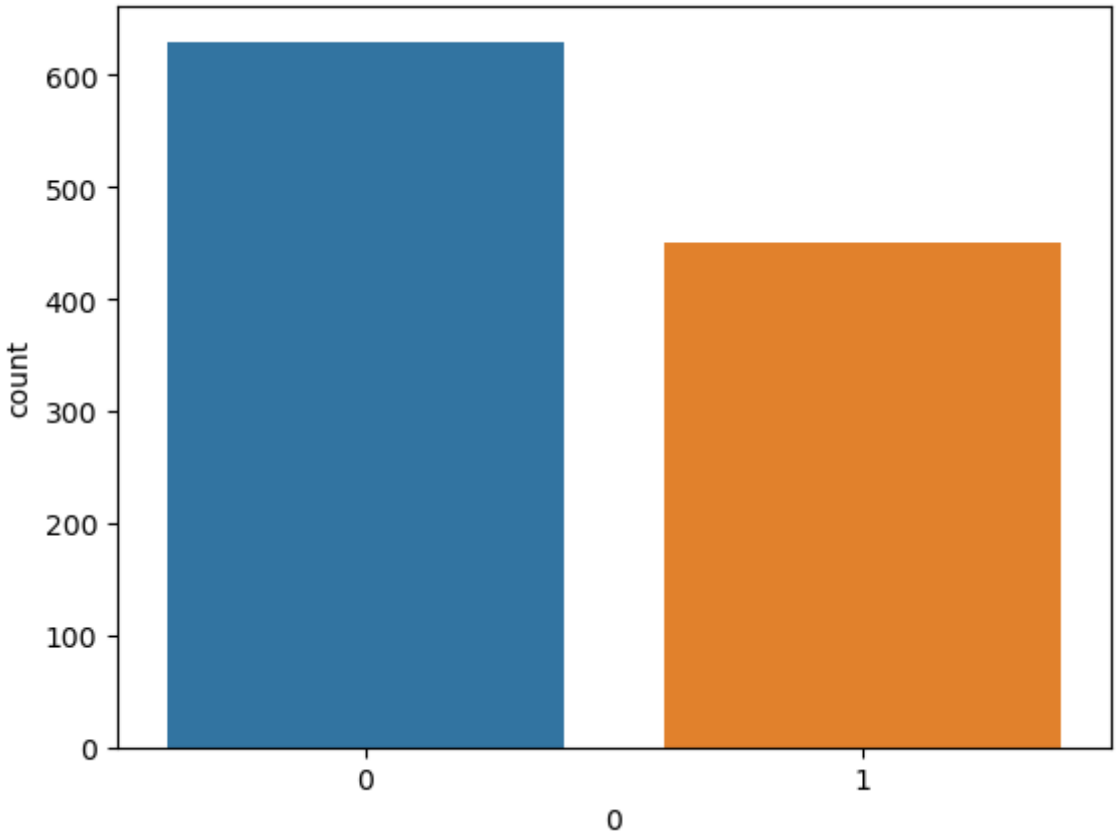
The first column of the data is the label. The remaining columns are the features. The features are the different ECG signals. The label is the class of the ECG signal. The classes are:

- 0 (not having a cardiovascular disease)
- 1 (having a cardiovascular disease)

Aim is to build a model that can predict the class of the ECG signal. This can be formulated as a binary classification problem.

In [9]:
```
# plot countplot for first column
sns.countplot(x=0, data=train_df)
```

Out[9]:
```
<AxesSubplot:xlabel='0', ylabel='count'>
```



In [10]:

```
#Showing the info of train and test data
print('##############Train data##############')
print(train_df.info())
print('##############Test data##############')
print(test_df.info())
#taking a sample of the test data (for reasons of plotting later)
sample = train_df.sample(25)
sampleX= sample.iloc[:,1:]
sampleY=sample.iloc[:,0]
print('##############Sample Info##############')
print(sampleX.info())
```

```
##############Train data##############
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1081 entries, 0 to 1080
Columns: 141 entries, 0 to 140
dtypes: float64(140), int64(1)
memory usage: 1.2 MB
None
##############Test data##############
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 180 entries, 0 to 179
Columns: 141 entries, 0 to 140
dtypes: float64(140), int64(1)
memory usage: 198.4 KB
None
##############Sample Info##############
<class 'pandas.core.frame.DataFrame'>
Int64Index: 25 entries, 10 to 327
Columns: 140 entries, 1 to 140
dtypes: float64(140)
```

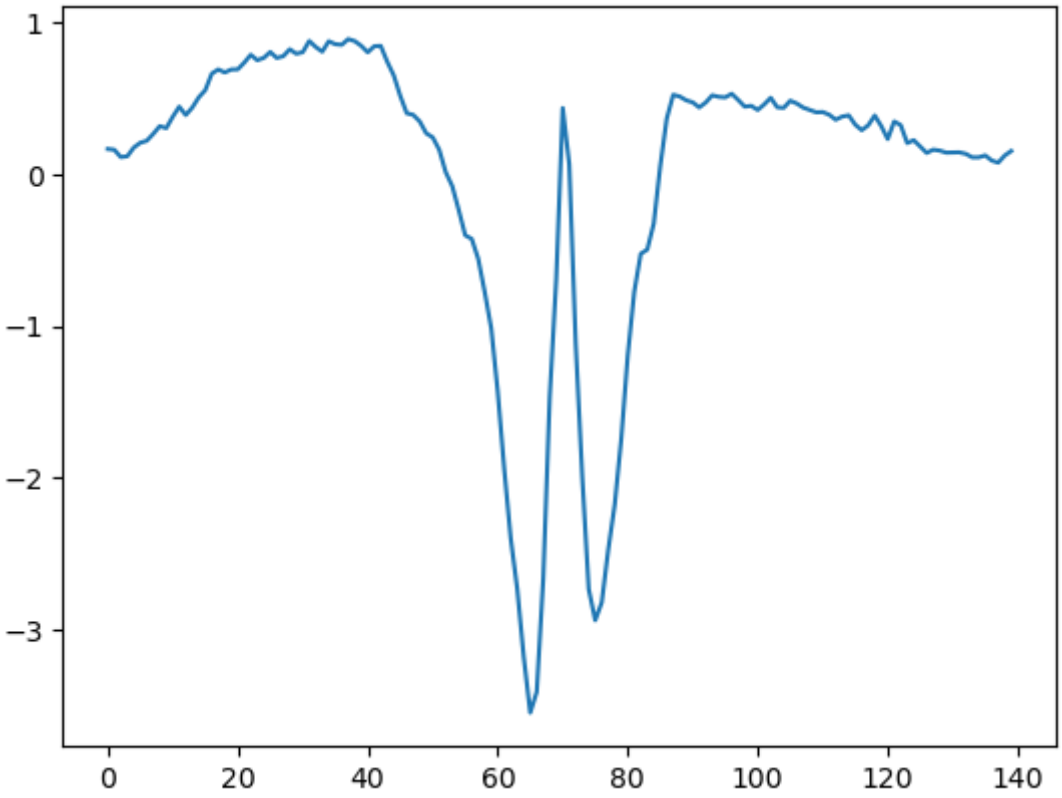
```
memory usage: 27.5 KB
None
```

In [11]:

```
import matplotlib.pyplot as plt
plt.plot(np.array(range(0,140)),sampleX.iloc[10])
```

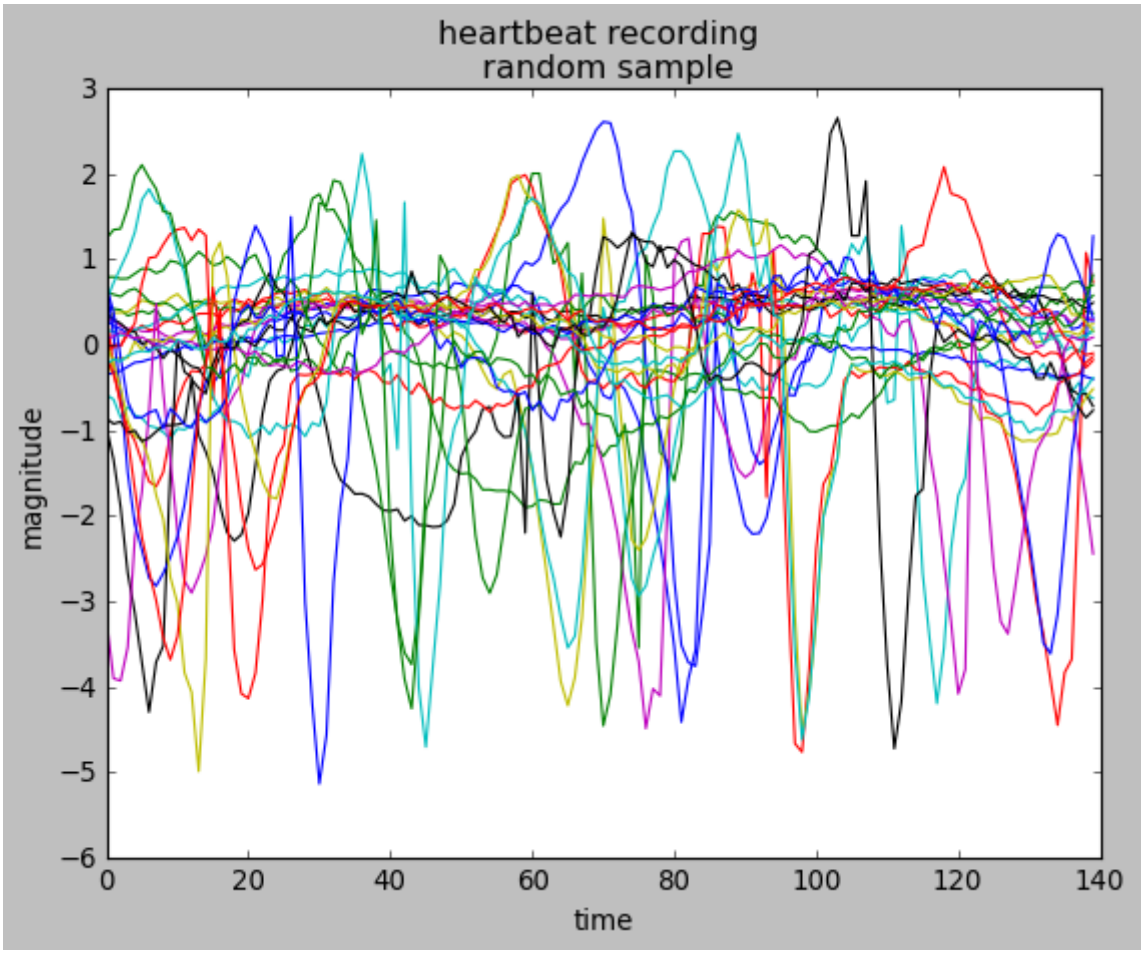
Out[11]:

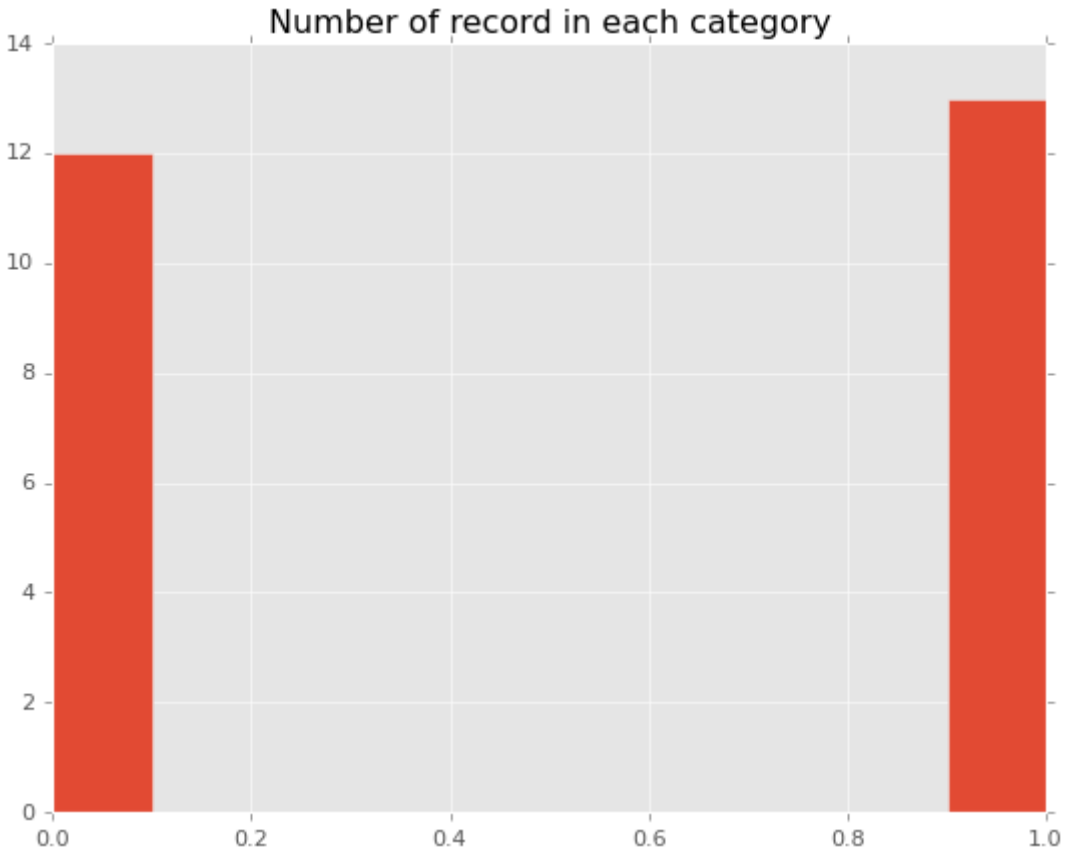
```
[<matplotlib.lines.Line2D at 0x21808f29d30>]
```



In [12]:

```
#ploting sample data information

plt.style.use('classic')
#ploting the samples
for index, row in sampleX.iterrows():
    plt.plot(np.array(range(0,140)), row)

plt.xlabel('time')
plt.ylabel('magnitude')
plt.title("heartbeat recording \n random sample")
plt.show()



plt.style.use('ggplot')
plt.title("Number of record in each category")
plt.hist(sample.iloc[:,0].transpose())
plt.show()
```





In [13]:

```
#number of labels for train and test
print("Train data")
print("Type\tCount")
print(train_df.iloc[:,0].value_counts())
```

```
print('####################################')
print("Test data")
print("Type\tCount")
print(test_df.iloc[:,0].value_counts())
```

```
Train data
Type    Count
0    630
1    451
Name: 0, dtype: int64
####################################
Test data
Type    Count
0    102
1     78
Name: 0, dtype: int64
```

The models that we will be using are:

- Naive Bayes Classifier
- K-Nearest Neighbors Classifier
- Ensemble Learning Classifier
- Support Vector Machine Classifier

Then we will compare the performance of the models and select the best model for the task.

In [8]:
```
# split the data into X_train and y_train, X_test and y_test
X_train = train_df.iloc[:, 1:]
y_train = train_df.iloc[:, 0]

X_valid = validation_df.iloc[:, 1:]
y_valid = validation_df.iloc[:, 0]

# Scale the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
```

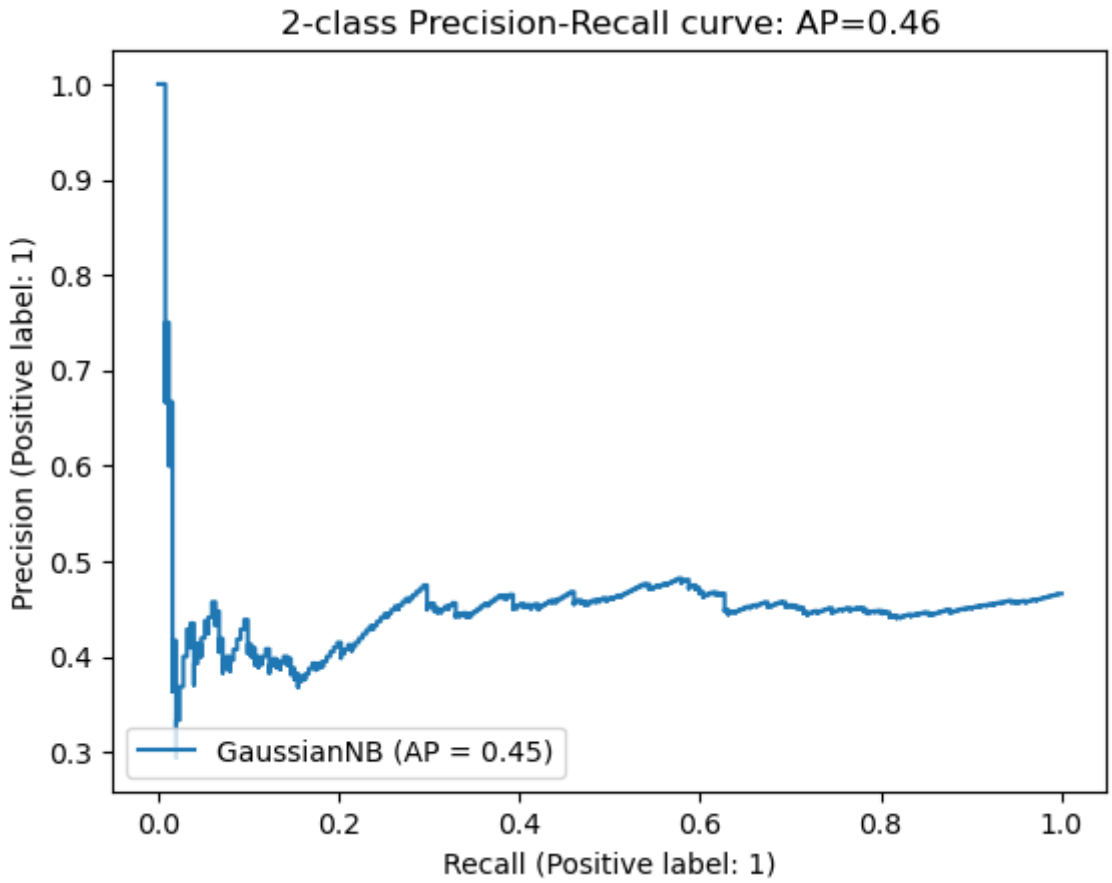
After splitting the data into training and validation sets, we need to scale the data. We will be using StandardScaler for this purpose.

1. Naive Bayes Classifier

In [66]:
```
# Naive Bayes
from sklearn.naive_bayes import GaussianNB



# Fit the model
gnb = GaussianNB()
gnb.fit(X_train, y_train)
```

```
# Predict the model
y_pred_gnb = gnb.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_gnb))
print("F1 score: ", f1_score(y_valid, y_pred_gnb, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_gnb))

# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(gnb, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_gnb)))

# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_gnb)
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.49353049907578556
F1 score:  0.4922838801304605
Confusion matrix:  [[159 130]
 [144 108]]
Average precision-recall score: 0.46
```



The naive Bayes classifier is a simple probabilistic classifier based on applying Bayes' theorem with strong (naive) independence assumptions between the features. A Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. A Naive Bayes classifier considers each of these features to contribute independently to the probability that this fruit is an apple, regardless of any possible correlations between the color, roundness, and diameter features.

Gaussian Naive Bayes is used when the data is normally distributed. It is assumed that the data follows a normal distribution. Inorder to use Gaussian Naive Bayes, we need to convert the data into a normal distribution.

This model gave an accuracy of 0.49 on the validation set which is not good.

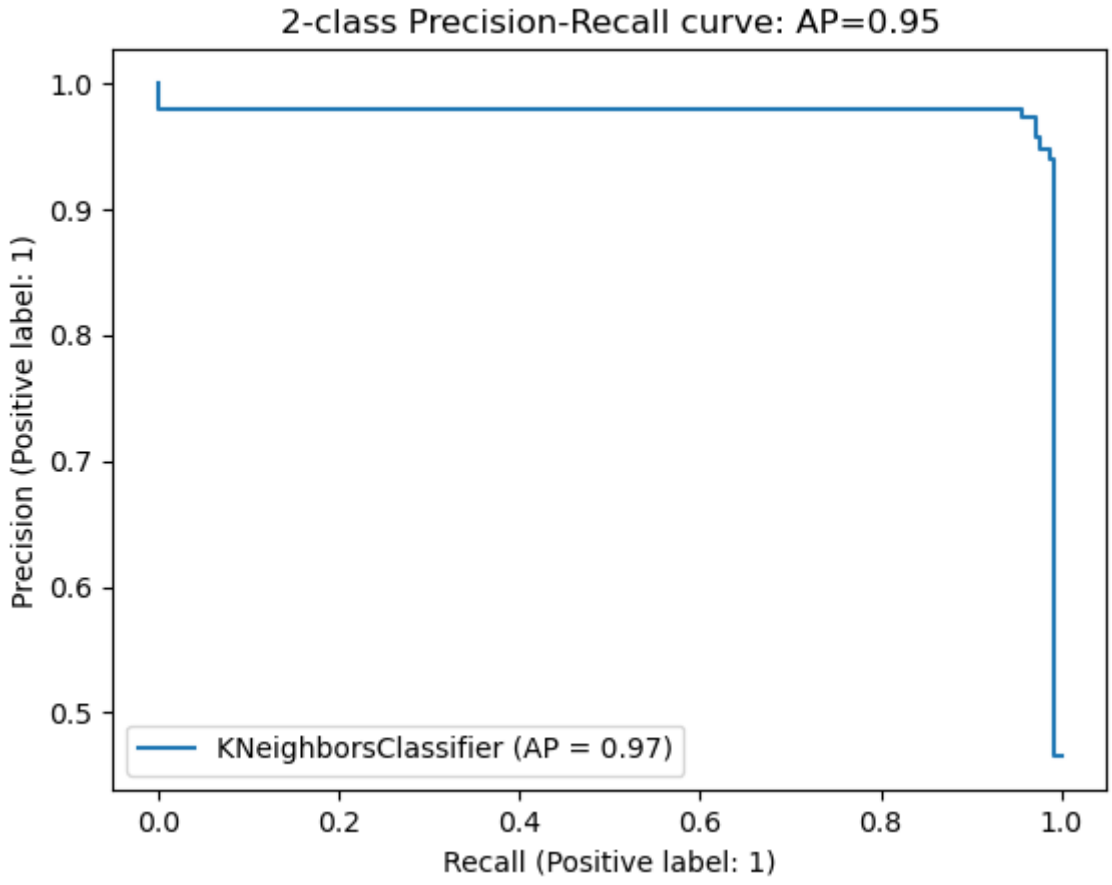
2. K-Nearest Neighbors Classifier

In [67]:

```
# KNN
from sklearn.neighbors import KNeighborsClassifier

# Fit the model
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Predict the model
y_pred_knn = knn.predict(X_valid)
```

```
# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_knn))
print("F1 score: ", f1_score(y_valid, y_pred_knn, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_knn))

# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(knn, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_knn)))

# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_knn)
print('Average precision-recall score: {0:0.2f}'.format(
      average_precision))
```

```
Accuracy:  0.9685767097966729
F1 score:  0.968593948287203
Confusion matrix:  [[278  11]
 [  6 246]]
Average precision-recall score: 0.95
```



KNN (K-Nearest Neighbors) is a simple algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions). KNN has been used in statistical estimation and pattern recognition already in the beginning of 1970's as a non-parametric technique. The algorithm is among the simplest of all machine learning algorithms.

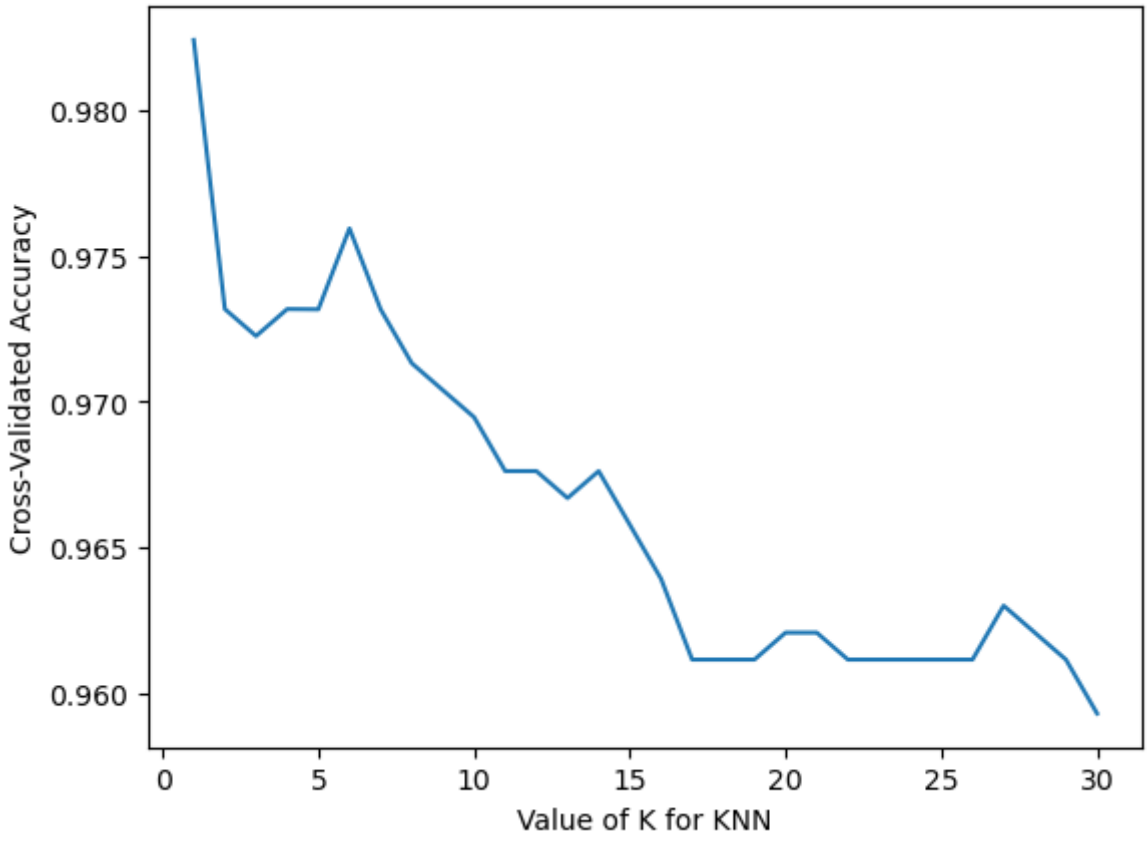
In KNN, the hyperparameter K is the number of nearest neighbors to be considered. The KNN algorithm assumes that similar things exist in close proximity. In other words, similar things are near to each other.

This k value is a hyperparameter that we need to tune. We will be using GridSearchCV to find the best value of k.

In [27]:

```
# using cross validation to find the best k
from sklearn.model_selection import cross_val_score



# search for an optimal value of K for KNN
k_range = range(1, 31)
k_scores = []
k_values = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train, cv=10,
scoring='accuracy')
    k_values.append(k)
    k_scores.append(scores.mean())
print(k_scores)

print()
print(max(k_scores))
optimal_k = k_values[k_scores.index(max(k_scores))]
print(optimal_k)

# plot the value of K for KNN (x-axis) versus the cross-validated
accuracy (y-axis)
plt.plot(k_range, k_scores)
plt.xlabel('Value of K for KNN')
plt.ylabel('Cross-Validated Accuracy')
plt.show()
```

```
[0.9824159021406726, 0.9731736323479444, 0.9722477064220184, 0.9731736323479444, 0.9
731651376146789, 0.9759429153924566, 0.9731651376146789, 0.971313285762827, 0.970387
359836901, 0.9694614339109752, 0.9676095820591233, 0.9676095820591233, 0.96668365613
31975, 0.9676095820591234, 0.9657662249405368, 0.963914373088685, 0.961136595310907
3, 0.9611365953109073, 0.9611365953109073, 0.9620625212368331, 0.9620625212368333,
0.9611365953109073, 0.9611365953109072, 0.9611365953109073, 0.9611365953109073, 0.96
11365953109073, 0.9629884471627591, 0.9620625212368331, 0.9611365953109072, 0.959284
7434590555]

0.9824159021406726
1
```

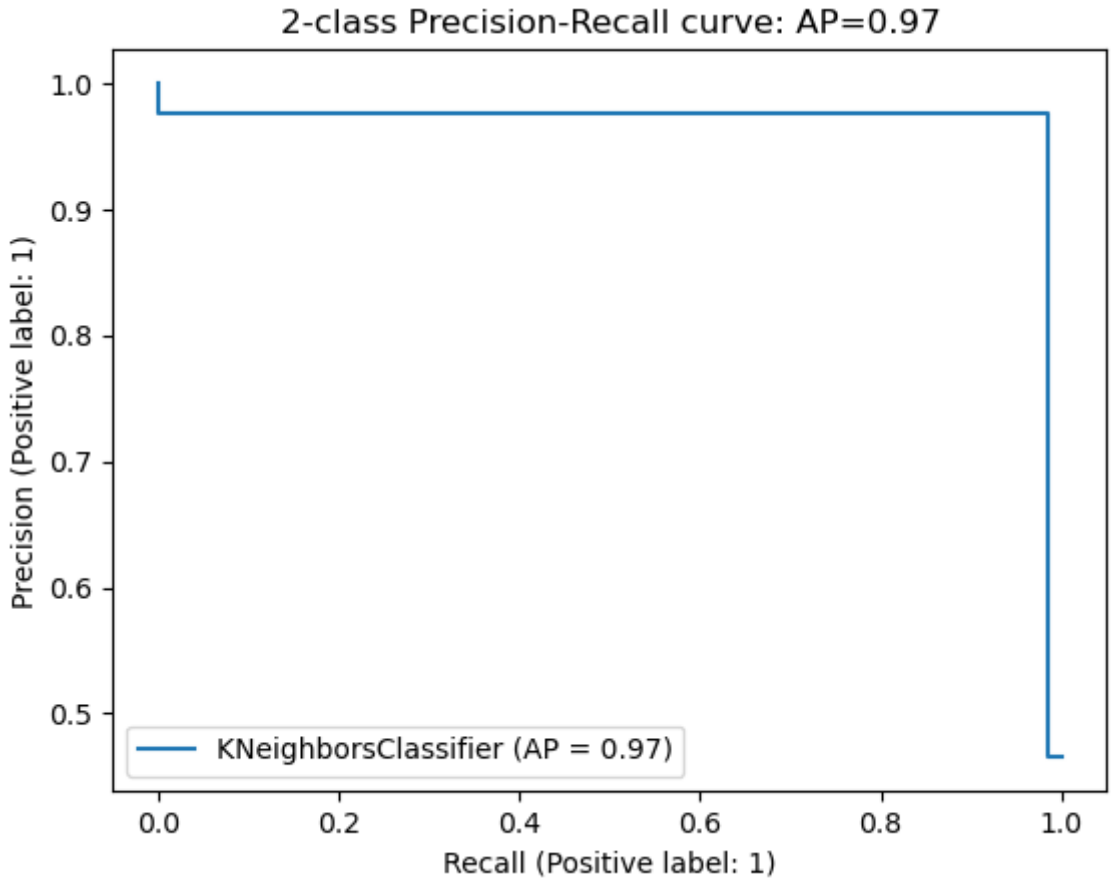


In [68]:

```
# Create a new KNN model with the optimal k
knn = KNeighborsClassifier(n_neighbors=optimal_k)

# Fit the model
knn.fit(X_train, y_train)

# Predict the model
y_pred_knn = knn.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_knn))
print("F1 score: ", f1_score(y_valid, y_pred_knn, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_knn))

# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(knn, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_knn)))

# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_knn)
```

```
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.9815157116451017
F1 score:  0.9815201510808478
Confusion matrix:  [[283   6]
 [  4 248]]
Average precision-recall score: 0.97
```



Cross validation helped us to find the best value of k. The best value of k is 1. This model gave an accuracy of 96% when k was set to 5. Now that we have found the best value of k, it gave an accuracy of 98% on the validation set.

3. Ensemble Learning Classifier

Ensemble learning is a machine learning paradigm where multiple models, such as classifiers or experts, are strategically generated and combined to solve a particular computational intelligence problem. Ensemble methods use multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone.

We will be using the following ensemble learning models:

- Bagging Classifier : A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

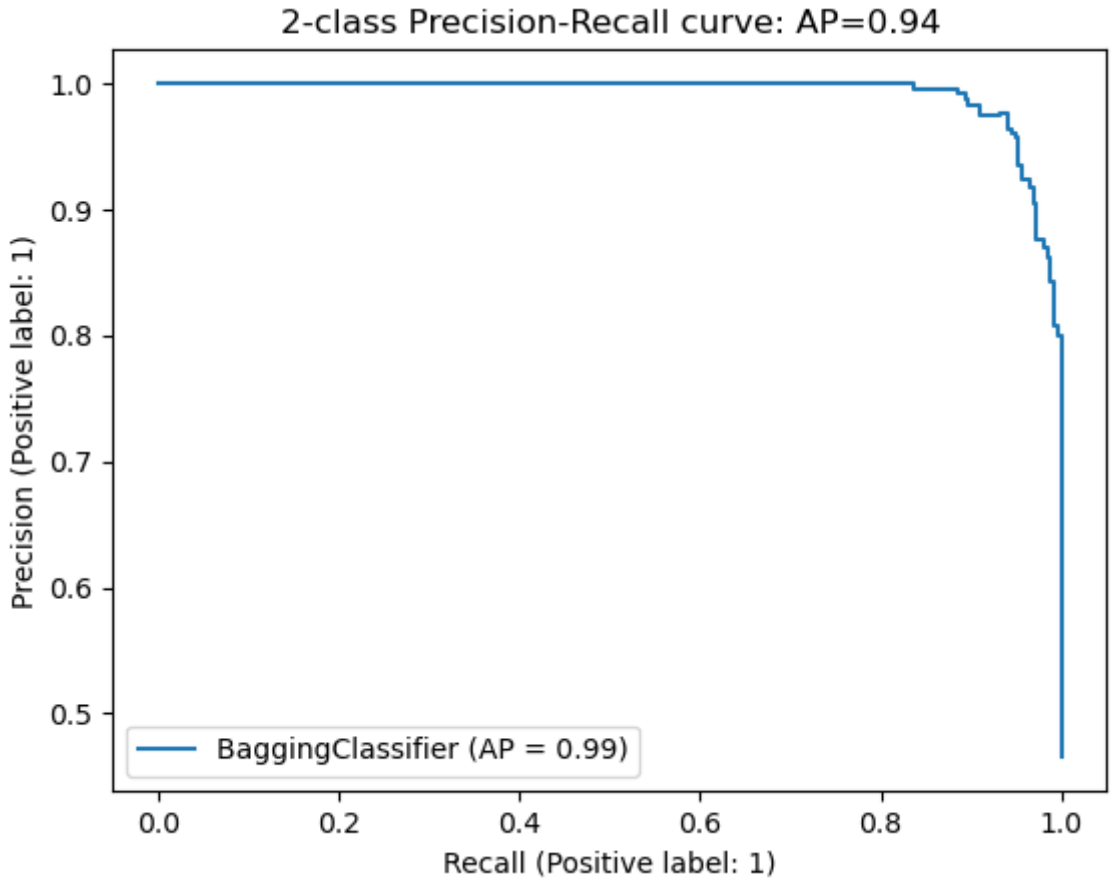
- Random Forest Classifier : A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if bootstrap=True (default).
- AdaBoost Classifier : AdaBoost (Adaptive Boosting) is a machine learning meta-algorithm formulated by Yoav Freund and Robert Schapire. It is a boosting algorithm that can be used to convert weak learners to strong ones. It is one of the most popular boosting algorithms. It is used in a variety of areas including computer vision and speech recognition.
- Gradient Boosting Classifier : Gradient Boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

In [69]:

```
# Bagging Classifier
from sklearn.ensemble import BaggingClassifier

# fit the model
bagging = BaggingClassifier(n_estimators=100, max_samples=0.5,
max_features=0.5)
bagging.fit(X_train, y_train)

# Predict the model
y_pred_bagging = bagging.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_bagging))
print("F1 score: ", f1_score(y_valid, y_pred_bagging,
average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_bagging))

# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(bagging, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_bagging)))

# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_bagging)
```

```
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.9611829944547134
F1 score:  0.9611276875860413
Confusion matrix:  [[283   6]
 [ 15 237]]
Average precision-recall score: 0.94
```



In [34]:

```
# implementing cross validation to find the best n_estimators,
max_samples and max_features using GridSearchCV
from sklearn.model_selection import GridSearchCV

# Create the parameter grid based on the results of random search
param_grid = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_samples': [0.5, 0.6, 0.7, 0.8, 0.9],
    'max_features': [0.5, 0.6, 0.7, 0.8, 0.9]
}

# Create a based model
bagging = BaggingClassifier()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator=bagging, param_grid=param_grid,
cv=3, n_jobs=-1, verbose=2)
```

```
# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# print the best parameters
print(grid_search.best_params_)
print(grid_search.best_score_)
print(grid_search.best_estimator_)
```

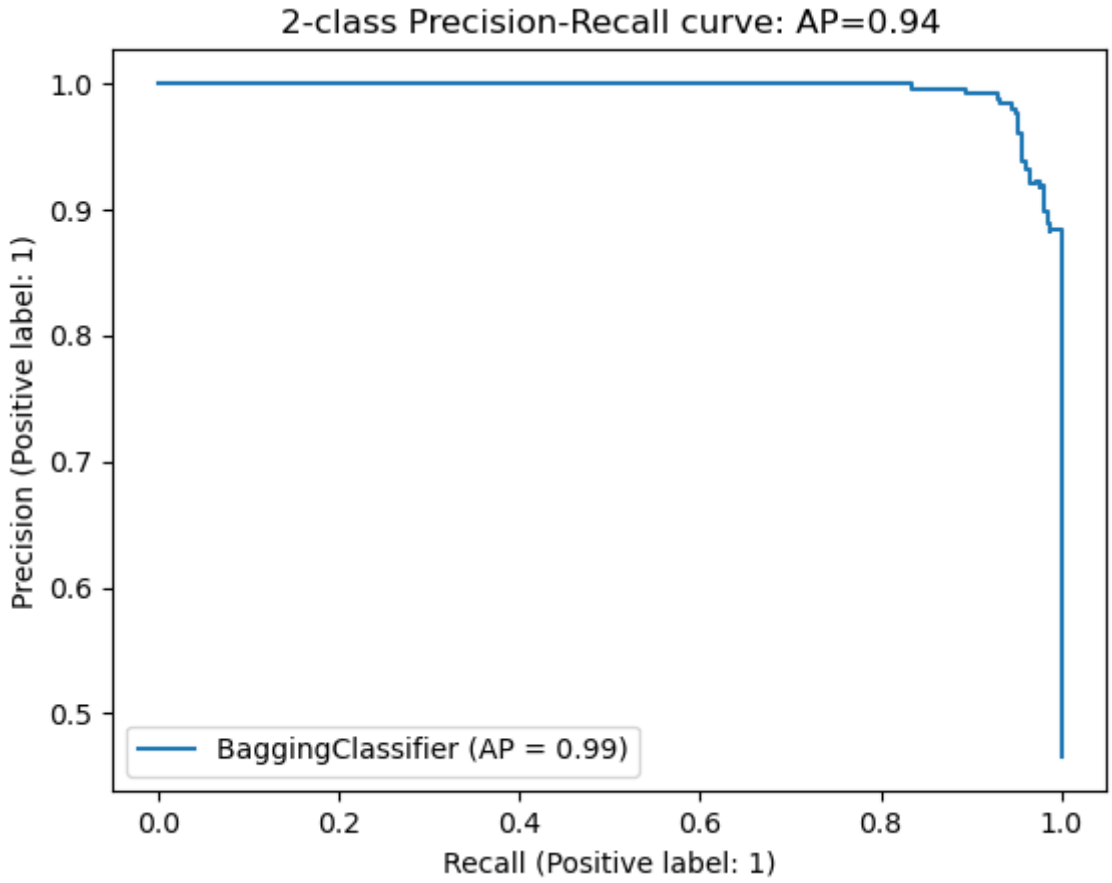
```
Fitting 3 folds for each of 125 candidates, totalling 375 fits
{'max_features': 0.6, 'max_samples': 0.9, 'n_estimators': 300}
0.9491022878834512
BaggingClassifier(max_features=0.6, max_samples=0.9, n_estimators=300)
Accuracy:  0.9648798521256932
F1 score:  0.9648545004581948
Confusion matrix:  [[282   7]
 [ 12 240]]
```

In [72]:

```
# Create a new Bagging Classifier model with the optimal parameters
bagging = BaggingClassifier(n_estimators=300,
                            max_samples=0.9,
                            max_features=0.6)

# Fit the model
bagging.fit(X_train, y_train)

# Predict the model
y_pred_bagging = bagging.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_bagging))
print("F1 score: ", f1_score(y_valid, y_pred_bagging,
average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_bagging))

# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(bagging, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_bagging)))

# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_bagging)
```

```
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.9630314232902033
F1 score:  0.9630105887658721
Confusion matrix:  [[281   8]
 [ 12 240]]
Average precision-recall score: 0.94
```



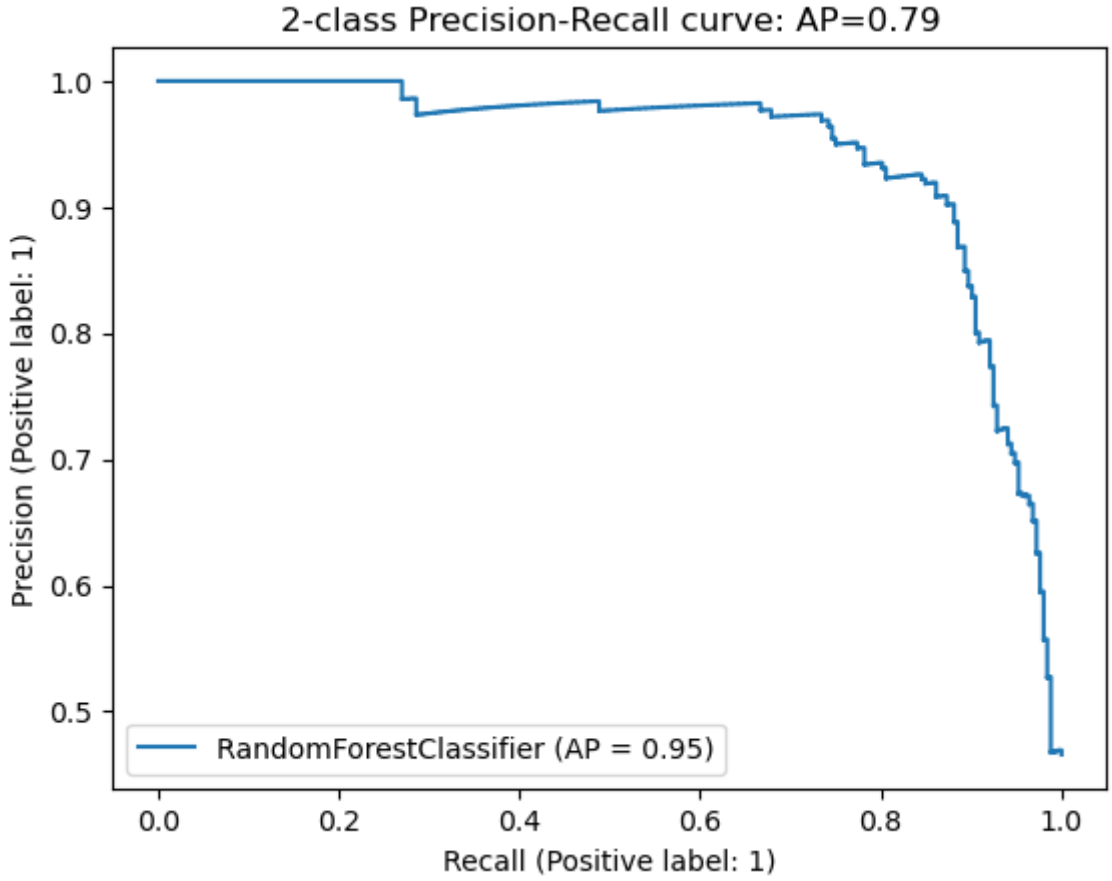
No big difference in the accuracy of the models while using cross validation with 96% accuracy.

In [73]:

```
# Random Forest
from sklearn.ensemble import RandomForestClassifier

# fit the model
rf = RandomForestClassifier(n_estimators=100, max_depth=2,
random_state=0)
rf.fit(X_train, y_train)

# Predict the model
y_pred_rf = rf.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_rf))
print("F1 score: ", f1_score(y_valid, y_pred_rf, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_rf))
```

```
# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(rf, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_rf)))
# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_rf)
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.8188539741219963
F1 score:  0.8110658237266908
Confusion matrix:  [[286   3]
 [ 95 157]]
Average precision-recall score: 0.79
```



In [39]:

```
# implementing cross validation to find the best n_estimators, max_depth
and random_state using GridSearchCV
from sklearn.model_selection import GridSearchCV

# Create the parameter grid based on the results of random search
param_grid = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [2, 3, 4, 5, 6, 7, 8, 9, 10],
    'random_state': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
}

# Create a based model
rf = RandomForestClassifier()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=3,
n_jobs=-1, verbose=2)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# print the best parameters
print(grid_search.best_params_)
print(grid_search.best_score_)
print(grid_search.best_estimator_)
```

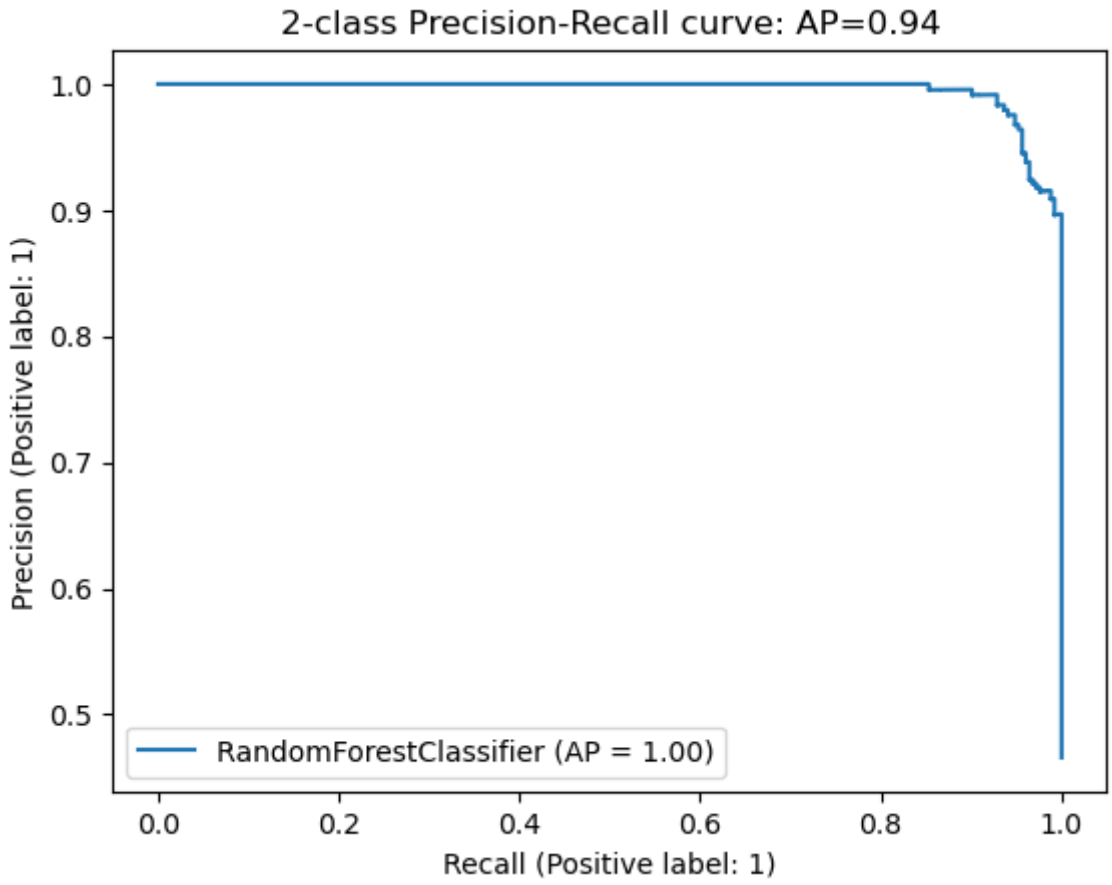
```
Fitting 3 folds for each of 450 candidates, totalling 1350 fits
{'max_depth': 10, 'n_estimators': 300, 'random_state': 0}
0.9528008618036319
RandomForestClassifier(max_depth=10, n_estimators=300, random_state=0)
```

In [74]:

```
# Create a new Random Forest model with the optimal parameters
rf = RandomForestClassifier(n_estimators=300,
                            max_depth=10,
                            random_state=0)

# Fit the model
rf.fit(X_train, y_train)

# Predict the model
y_pred_rf = rf.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_rf))
print("F1 score: ", f1_score(y_valid, y_pred_rf, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_rf))

# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(rf, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_rf)))
```

```
# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_rf)
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.9630314232902033
F1 score:  0.9630105887658721
Confusion matrix:  [[281   8]
 [ 12 240]]
Average precision-recall score: 0.94
```



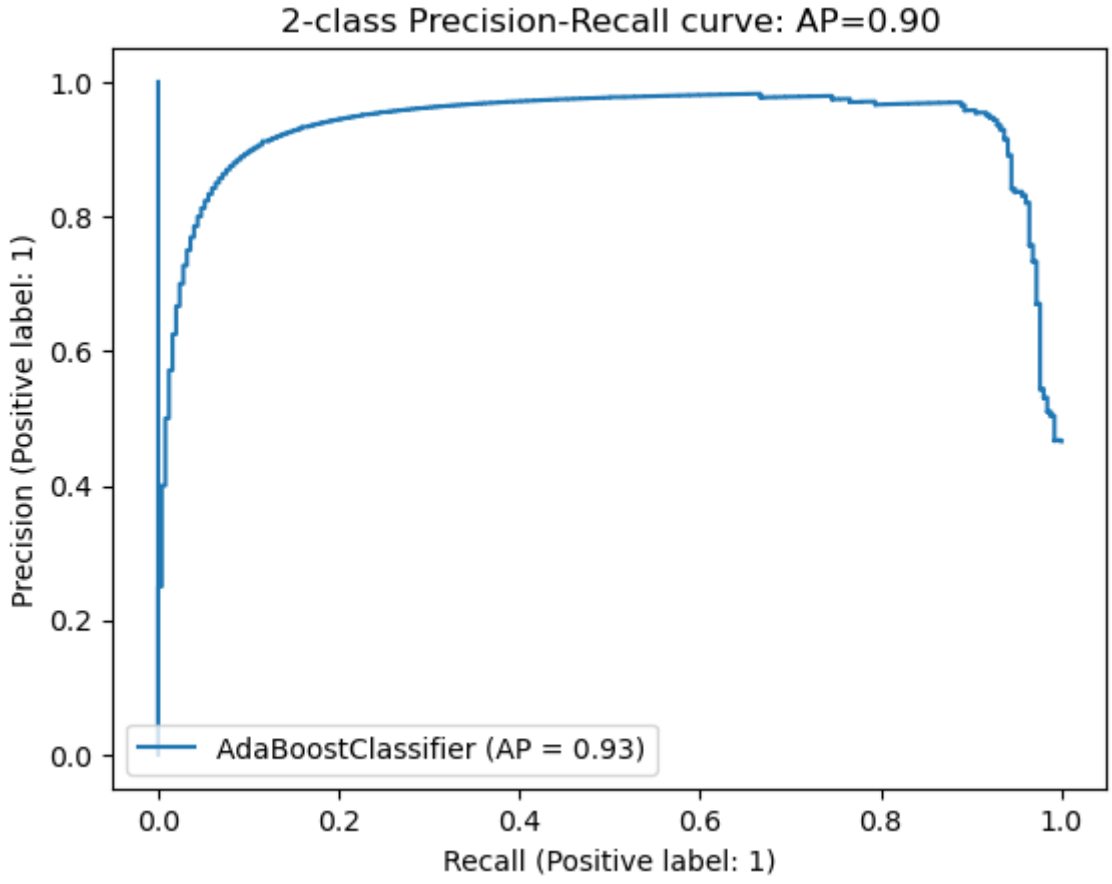
Random Forest Classifier is an extension of bagging classifier. It is a very powerful model. It gave an accuracy of 96% on the validation set which is an improvement over the previous model without cross validation.

In [75]:

```
# AdaBoost
from sklearn.ensemble import AdaBoostClassifier

# fit the model
ada = AdaBoostClassifier(n_estimators=100, random_state=0)
ada.fit(X_train, y_train)

# Predict the model
y_pred_ada = ada.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_ada))
```

```
print("F1 score: ", f1_score(y_valid, y_pred_ada, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_ada))

# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(ada, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_ada)))
# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_ada)
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.9353049907578558
F1 score:  0.935327626521316
Confusion matrix:  [[270  19]
 [ 16 236]]
Average precision-recall score: 0.90
```



In [76]:

```
# implementing cross validation to find the best n_estimators and
random_state using GridSearchCV
from sklearn.model_selection import GridSearchCV

# Create the parameter grid based on the results of random search
param_grid = {
```

```
    'n_estimators': [100, 200, 300, 400, 500],
    'random_state': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}

# Create a based model
ada = AdaBoostClassifier()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator=ada, param_grid=param_grid, cv=3,
n_jobs=-1, verbose=2)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# print the best parameters
print(grid_search.best_params_)
print(grid_search.best_score_)
print(grid_search.best_estimator_)
```

```
Fitting 3 folds for each of 50 candidates, totalling 150 fits
{'n_estimators': 400, 'random_state': 0}
0.9306094182825485
AdaBoostClassifier(n_estimators=400, random_state=0)
```

In [45]:

```
print(grid_search.best_estimator_)
```

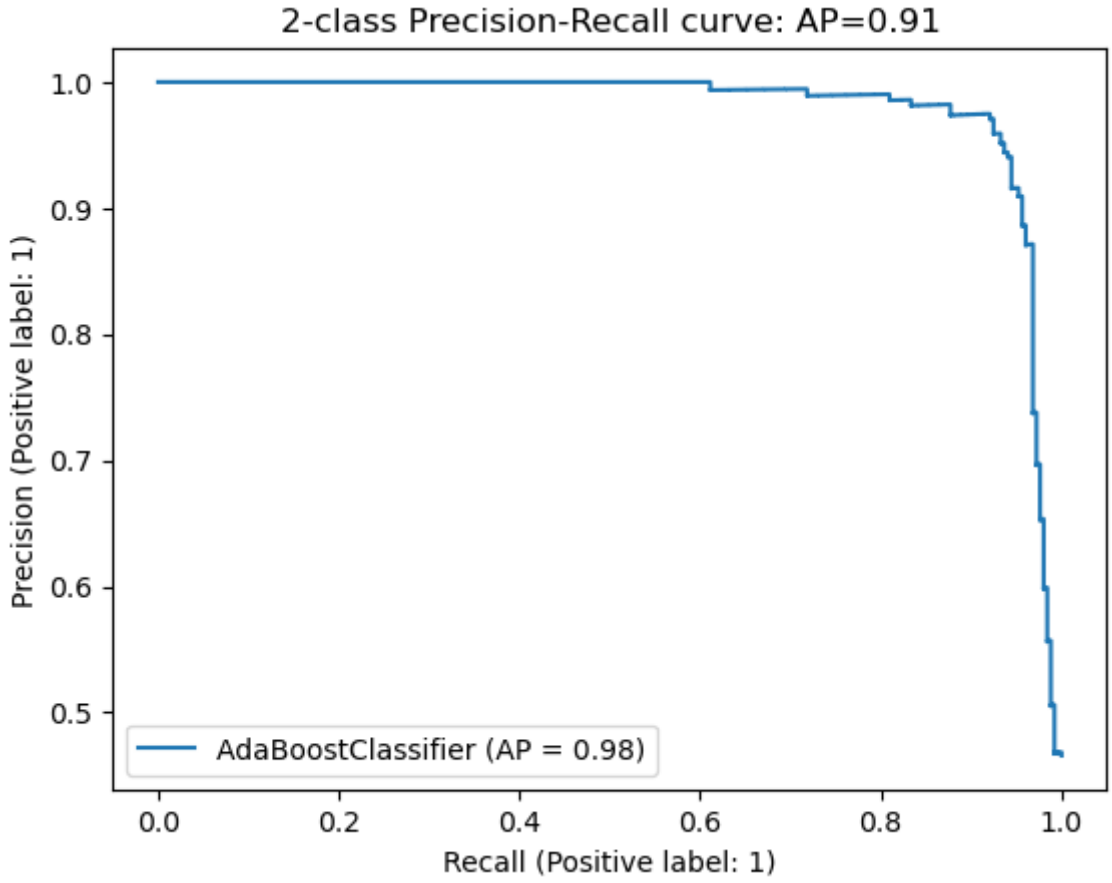
```
AdaBoostClassifier(n_estimators=400, random_state=0)
```

In [77]:

```
# Create a new AdaBoost model with the optimal parameters
ada = AdaBoostClassifier(n_estimators=400,
                         random_state=0)

# Fit the model
ada.fit(X_train, y_train)

# Predict the model
y_pred_ada = ada.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_ada))
print("F1 score: ", f1_score(y_valid, y_pred_ada, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_ada))

# Precision and recall and Area under the curv
```

```
disp = plot_precision_recall_curve(ada, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_ada)))
# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_ada)
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.944547134935305
F1 score:  0.9445322794487168
Confusion matrix:  [[275  14]
 [ 16 236]]
Average precision-recall score: 0.91
```



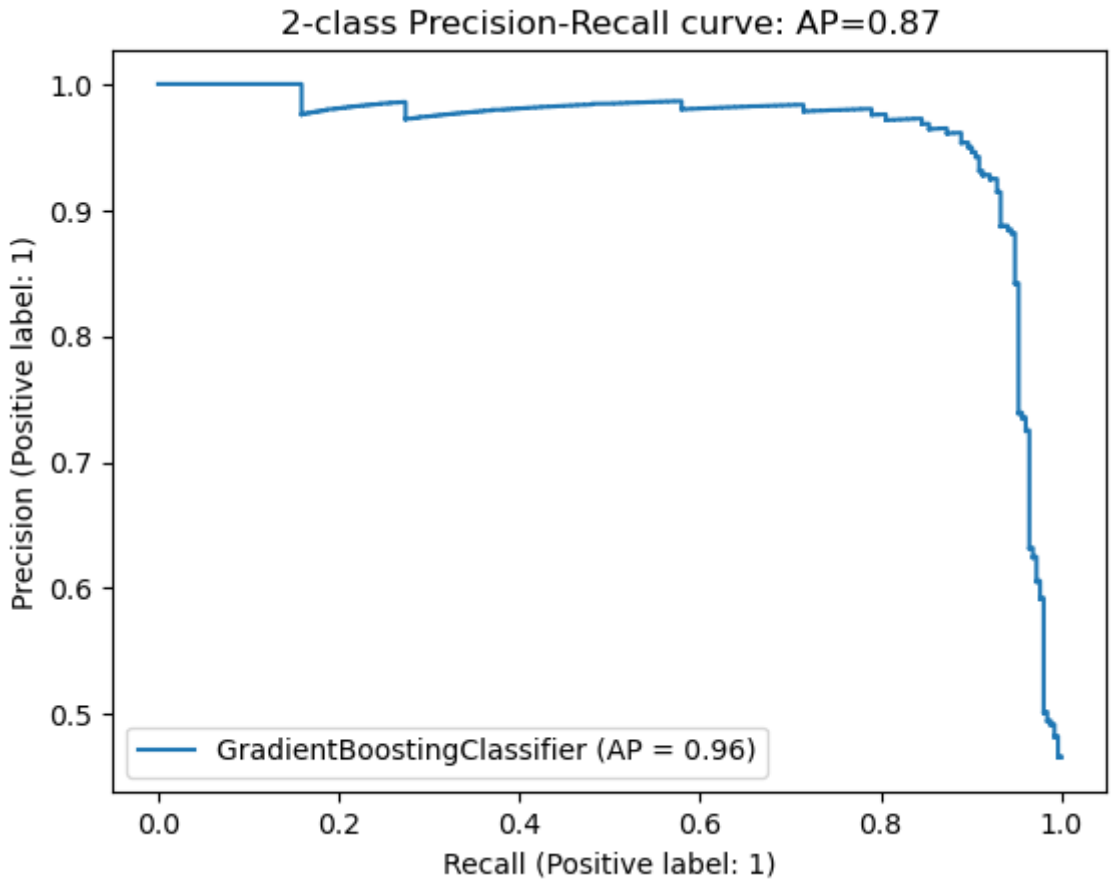
AdaBoost Classifier is a boosting classifier. It is a very powerful model. It gave an accuracy of 94% on the validation set.

In [78]:

```
# Gradient Boosting
from sklearn.ensemble import GradientBoostingClassifier

# fit the model
gb = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
max_depth=1, random_state=0)
gb.fit(X_train, y_train)

# Predict the model
```

```
y_pred_gb = gb.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_gb))
print("F1 score: ", f1_score(y_valid, y_pred_gb, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_gb))

# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(gb, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_gb)))
# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_gb)
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.922365988909427
F1 score:  0.9224277045432434
Confusion matrix:  [[264  25]
 [ 17 235]]
Average precision-recall score: 0.87
```



In [48]:

```
# implementing cross validation to find the best n_estimators,
learning_rate, max_depth and random_state using GridSearchCV
```

```
from sklearn.model_selection import GridSearchCV

# Create the parameter grid based on the results of random search
param_grid = {
    'n_estimators': [100, 200, 300, 400, 500],
    'learning_rate': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
    'max_depth': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'random_state': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}

# Create a based model
gb = GradientBoostingClassifier()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator=gb, param_grid=param_grid, cv=3,
n_jobs=-1, verbose=2)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# print the best parameters
print(grid_search.best_params_)
print(grid_search.best_score_)
print(grid_search.best_estimator_)
```

```
Fitting 3 folds for each of 5000 candidates, totalling 15000 fits
{'learning_rate': 0.5, 'max_depth': 7, 'n_estimators': 100, 'random_state': 2}
0.9583538524674259
GradientBoostingClassifier(learning_rate=0.5, max_depth=7, random_state=2)
```

In [49]:

```
print(grid_search.best_estimator_)
# GradientBoostingClassifier(learning_rate=0.5, max_depth=7,
random_state=2)
```

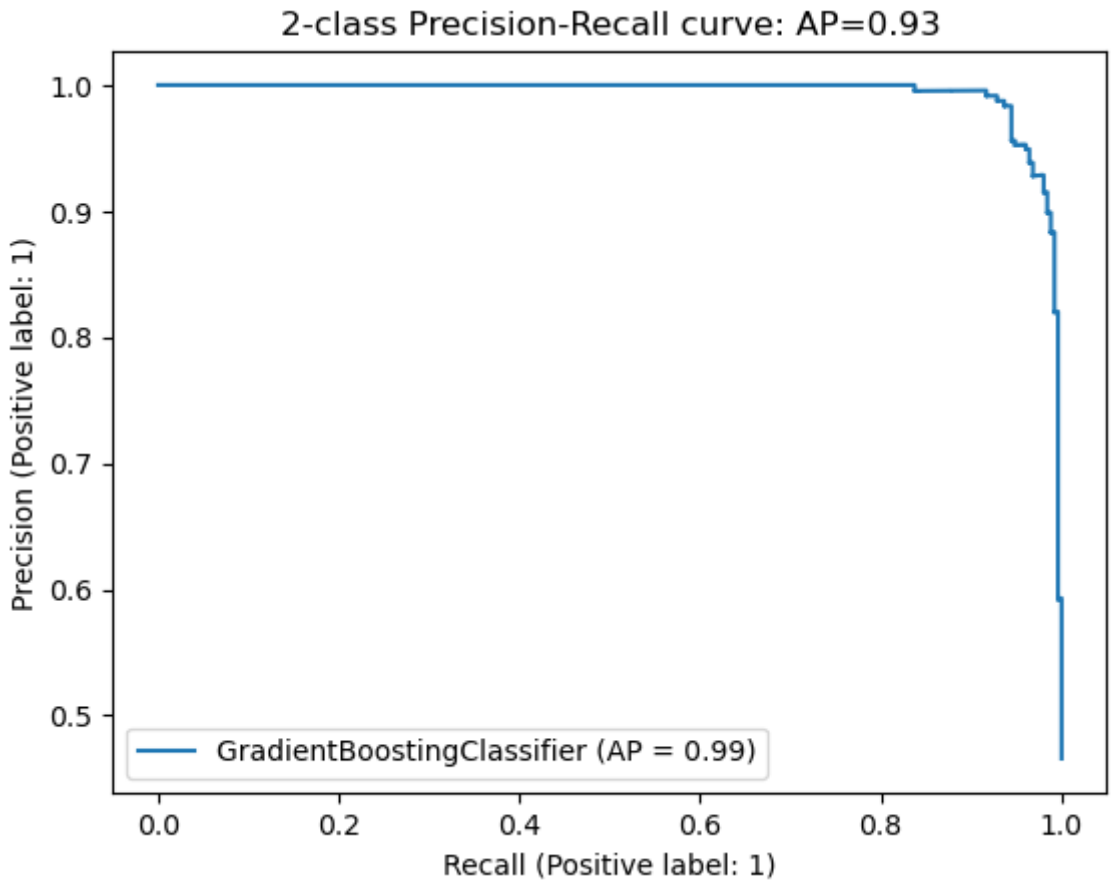
```
GradientBoostingClassifier(learning_rate=0.5, max_depth=7, random_state=2)
```

In [79]:

```
# Create a new Gradient Boosting model with the optimal parameters
gb = GradientBoostingClassifier(n_estimators=100,
                                learning_rate=0.5,
                                max_depth=7,
                                random_state=2)

# Fit the model
gb.fit(X_train, y_train)
```

```
# Predict the model
y_pred_gb = gb.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_gb))
print("F1 score: ", f1_score(y_valid, y_pred_gb, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_gb))

# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(gb, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                   'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_gb)))
# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_gb)
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.955637707948244
F1 score:  0.9556258235589734
Confusion matrix:  [[278  11]
 [ 13 239]]
Average precision-recall score: 0.93
```



Gradient Boosting Classifier is a boosting classifier. It is a very powerful model. It gave an

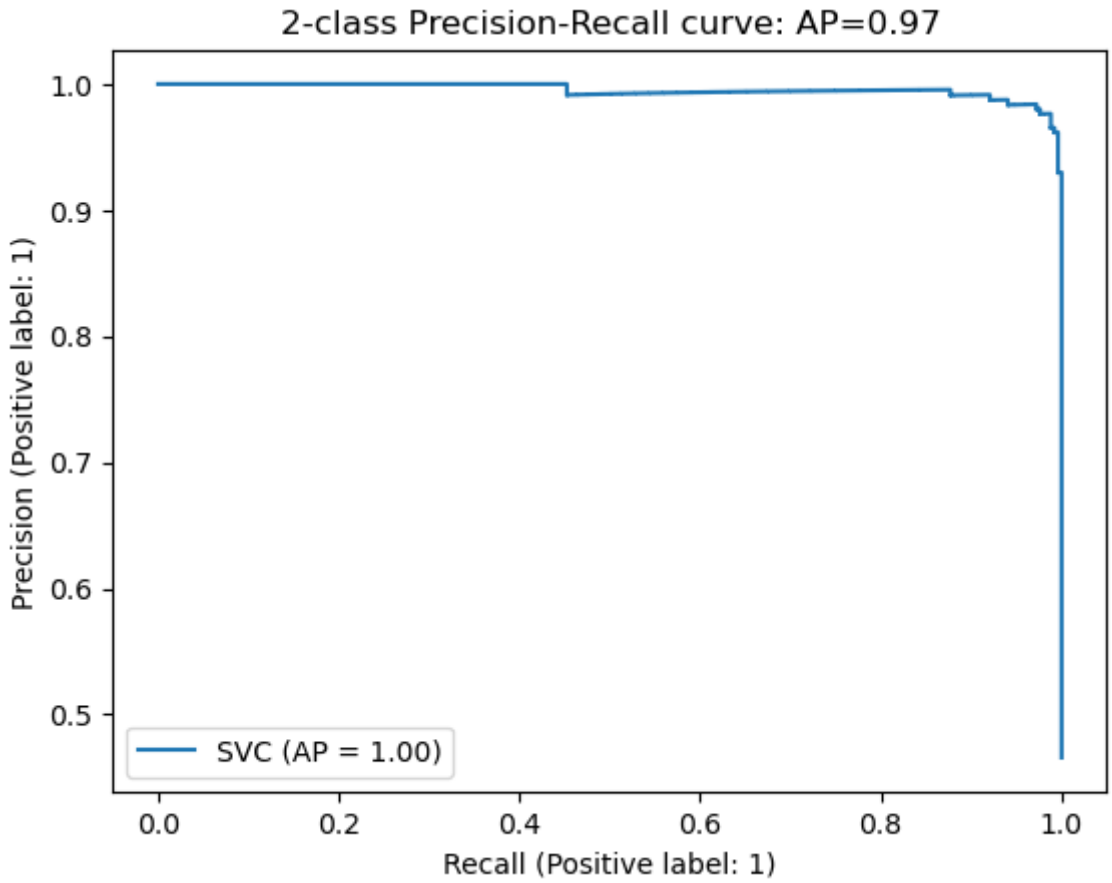
accuracy of 95% on the validation set which is an improvement over the previous model without cross validation (92%).

4. Support Vector Machine Classifier

In [80]:

```
# SVM Classifier
from sklearn.svm import SVC

# fit the model
svm = SVC(gamma='auto')
svm.fit(X_train, y_train)

# Predict the model
y_pred_svm = svm.predict(X_valid)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_valid, y_pred_svm))
print("F1 score: ", f1_score(y_valid, y_pred_svm, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_valid, y_pred_svm))

# Precision and recall and Area under the curv

disp = plot_precision_recall_curve(svm, X_valid, y_valid)
disp.ax_.set_title('2-class Precision-Recall curve: '
                           'AP=
{0:0.2f}'.format(average_precision_score(y_valid, y_pred_svm)))
# Area under the curve
average_precision = average_precision_score(y_valid, y_pred_svm)
print('Average precision-recall score: {0:0.2f}'.format(
        average_precision))
```

```
Accuracy:  0.9815157116451017
F1 score:  0.9815201510808478
Confusion matrix:  [[283   6]
 [  4 248]]
Average precision-recall score: 0.97
```



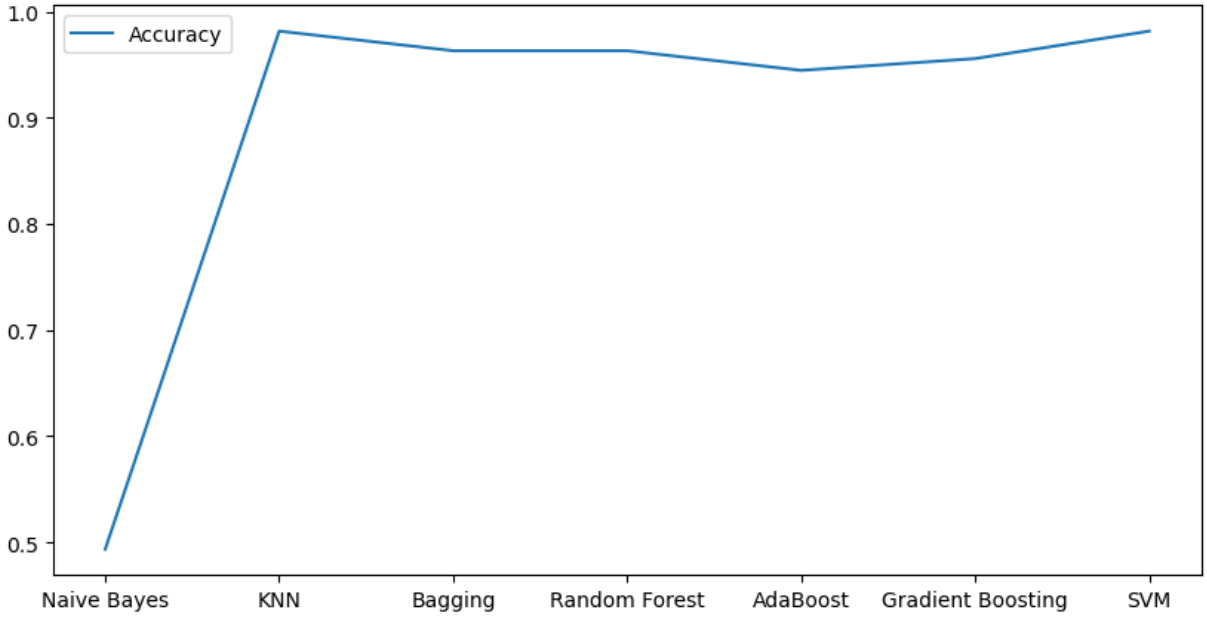
SVM (Support Vector Machine) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiate the two classes very well.

And from this, we can see that this model gave an accuracy of 98% on the validation set.

5. Comparison of the models

In [81]:
```
models = ['Naive Bayes', 'KNN', 'Bagging', 'Random Forest', 'AdaBoost',
'Gradient Boosting', 'SVM']
accuracy = [accuracy_score(y_valid, y_pred_gnb), accuracy_score(y_valid,
y_pred_knn), accuracy_score(y_valid, y_pred_bagging),
accuracy_score(y_valid, y_pred_rf), accuracy_score(y_valid, y_pred_ada),
accuracy_score(y_valid, y_pred_gb), accuracy_score(y_valid, y_pred_svm)]
```

In [82]:
```
plt.figure(figsize=(10,5))
plt.plot(models, accuracy, label='Accuracy')
plt.legend()
plt.show()
```



In [83]:
```
accuracy
```

Out[83]:
```
[0.49353049907578556,
 0.9815157116451017,
 0.9630314232902033,
 0.9630314232902033,
 0.944547134935305,
 0.955637707948244,
 0.9815157116451017]
```

Based on the accuracy of the models on validation data, K-Nearest Neighbors Classifier and SVM, both gave an accuracy of 98% which is impressive.

In [85]:
```
# Testing these models on test data
X_test = test_df.iloc[:, 1:]
y_test = test_df.iloc[:, 0]

# Scale the data
# scaler = StandardScaler()
# X_test = scaler.fit_transform(X_test)

# Naive Bayes
print("Naive Bayes")
y_pred_gnb = gnb.predict(X_test)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_test, y_pred_gnb))
print("F1 score: ", f1_score(y_test, y_pred_gnb, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_test, y_pred_gnb))
print("**************************************")
# KNN
print("KNN")
```

```
y_pred_knn = knn.predict(X_test)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_test, y_pred_knn))
print("F1 score: ", f1_score(y_test, y_pred_knn, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_test, y_pred_knn))
print("**************************************")
# Bagging
print("Bagging")
y_pred_bagging = bagging.predict(X_test)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_test, y_pred_bagging))
print("F1 score: ", f1_score(y_test, y_pred_bagging,
average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_test, y_pred_bagging))
print("**************************************")
# Random Forest
print("Random Forest")
y_pred_rf = rf.predict(X_test)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_test, y_pred_rf))
print("F1 score: ", f1_score(y_test, y_pred_rf, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_test, y_pred_rf))
print("**************************************")
# AdaBoost
print("AdaBoost")
y_pred_ada = ada.predict(X_test)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_test, y_pred_ada))
print("F1 score: ", f1_score(y_test, y_pred_ada, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_test, y_pred_ada))
print("**************************************")
# Gradient Boosting
print("Gradient Boosting")
y_pred_gb = gb.predict(X_test)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_test, y_pred_gb))
```

```
print("F1 score: ", f1_score(y_test, y_pred_gb, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_test, y_pred_gb))
print("**************************************")
# SVM
print("SVM")
y_pred_svm = svm.predict(X_test)

# Evaluate the model
print("Accuracy: ", accuracy_score(y_test, y_pred_svm))
print("F1 score: ", f1_score(y_test, y_pred_svm, average='weighted'))
print("Confusion matrix: ", confusion_matrix(y_test, y_pred_svm))
print("**************************************")
```

```
Naive Bayes
Accuracy:  0.5277777777777778
F1 score:  0.5265700483091788
Confusion matrix:  [[61 41]
 [44 34]]
**************************************
KNN
Accuracy:  0.9833333333333333
F1 score:  0.9833453609864768
Confusion matrix:  [[100   2]
 [  1  77]]
**************************************
Bagging
Accuracy:  0.9777777777777777
F1 score:  0.977741352498634
Confusion matrix:  [[101   1]
 [  3  75]]
**************************************
Random Forest
Accuracy:  0.9833333333333333
F1 score:  0.9833202202989771
Confusion matrix:  [[101   1]
 [  2  76]]
**************************************
AdaBoost
Accuracy:  0.9611111111111111
F1 score:  0.9611391756351123
Confusion matrix:  [[98  4]
 [ 3 75]]
**************************************
Gradient Boosting
Accuracy:  0.9888888888888889
F1 score:  0.9888888888888889
Confusion matrix:  [[101   1]
 [  1  77]]
**************************************
SVM
Accuracy:  0.9777777777777777
F1 score:  0.9778084137527677
Confusion matrix:  [[99  3]
 [ 1 77]]
**************************************
```

In [86]:

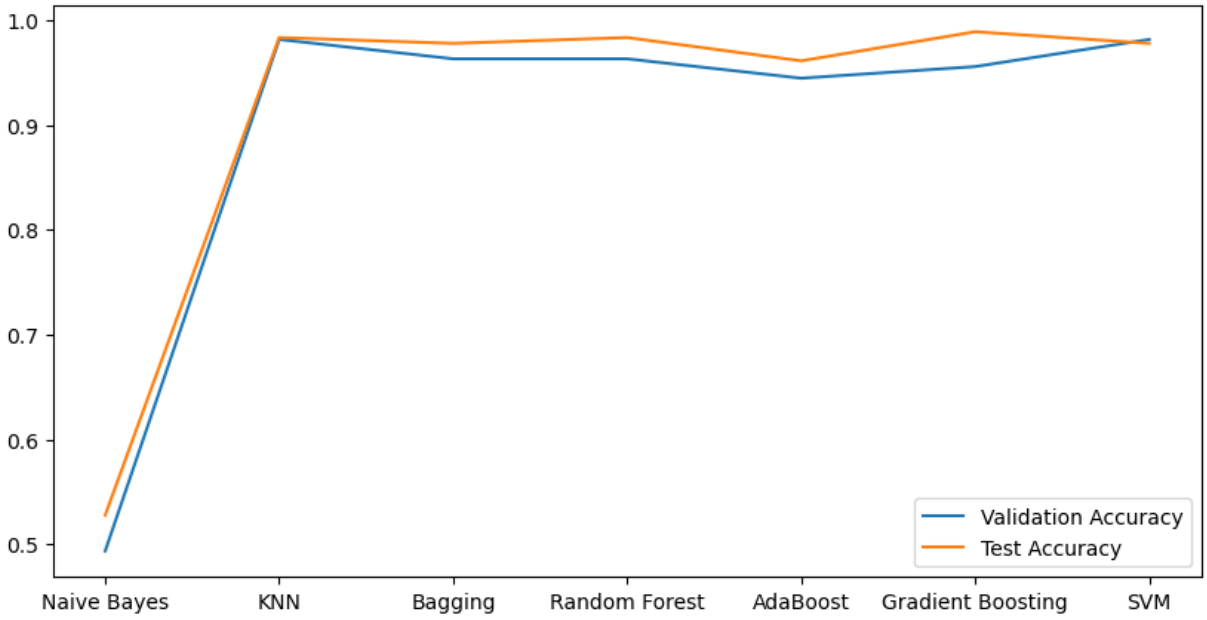
```
test_accuracy = [accuracy_score(y_test, y_pred_gnb),
accuracy_score(y_test, y_pred_knn), accuracy_score(y_test,
y_pred_bagging), accuracy_score(y_test, y_pred_rf),
```

```
accuracy_score(y_test, y_pred_ada), accuracy_score(y_test, y_pred_gb),
accuracy_score(y_test, y_pred_svm)]
```

In [87]:
```
test_accuracy
```

Out[87]:
```
[0.5277777777777778,
 0.9833333333333333,
 0.9777777777777777,
 0.9833333333333333,
 0.9611111111111111,
 0.9888888888888889,
 0.9777777777777777]
```

In [88]:
```
plt.figure(figsize=(10,5))
plt.plot(models, accuracy, label='Validation Accuracy')
plt.plot(models, test_accuracy, label='Test Accuracy')
plt.legend()
plt.show()
```



But, Gradient Boosting seems to produce better results than K-Nearest Neighbors Classifier and SVM on Testing Data. It gave an accuracy of 98% on the test set.

Its hard to say which model is better. Gradient Boosting Classifier and SVM both gave an accuracy of 98% on the validation set. But, Gradient Boosting Classifier gave an accuracy of 98% on the test set. So, we will be using Gradient Boosting Classifier for the task.

Assessment 2

Task 2: Develop recurrent neural network(s) for sequence-to-sequence classification

In [1]:

```
# TensorFlow config to GPU
import tensorflow as tf
print(tf.__version__)

gpus = tf.config.list_physical_devices('GPU')
if gpus:
    tf.config.set_logical_device_configuration(
        gpus[0],
        [tf.config.LogicalDeviceConfiguration(memory_limit=15292)]
    )

logical_gpus = tf.config.list_logical_devices('GPU')
print(logical_gpus)
print(len(gpus), "Physical GPU,", len(logical_gpus), "Logical GPUs")



from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())

print()
print()

import tensorflow as tf
print("Num GPUs Available: ",
len(tf.config.list_physical_devices('GPU')))
```

```
2.6.0
[LogicalDevice(name='/device:GPU:0', device_type='GPU')]
1 Physical GPU, 1 Logical GPUs
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 2201299975258163592
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 16034824192
locality {
  bus_id: 1
  links {
  }
```

```
}
incarnation: 3233997999066038929
physical_device_desc: "device: 0, name: NVIDIA GeForce RTX 2060, pci bus id: 0000:0
1:00.0, compute capability: 7.5"
]


Num GPUs Available:  1
```

In [2]:

```python
#loading environment
import pandas as pd
import numpy as np
''' Data visualisation'''
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
''' Scikit-Learn'''
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, RobustScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import cross_validate, cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn import set_config

set_config(display='diagram')
from sklearn.compose import ColumnTransformer
from sklearn.compose import make_column_selector
from sklearn.metrics import confusion_matrix
''' Imbalanced Classes'''
import imblearn
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline
''' Tensorflow Keras'''
from tensorflow import keras
from tensorflow.keras import models
from tensorflow.keras import Sequential, layers
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras import regularizers
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers.schedules import ExponentialDecay
from keras.utils import np_utils
from tensorflow.keras.utils import to_categorical
from keras.preprocessing import image
```

```
from os import listdir
from os.path import isdir, join
import warnings
warnings.filterwarnings('ignore')
```

In [46]:
```
# Set Pandas options to display more columns
pd.options.display.max_columns=200

# Read in the data csv
df=pd.read_csv('ECG_dataset/trainval.csv', encoding='utf-8',
header=None)

# Show a snaphsot of data
df
```

Out[46]:

	0	1	2	3	4	5	6	7	8	
0	1	0.024133	0.016065	0.044639	0.031001	-0.009473	-0.042663	-0.077283	-0.091508	-0.04
1	0	0.424380	0.344420	0.348130	0.340170	0.243370	0.241730	0.268780	0.273420	0.35
2	0	1.529500	1.776600	1.936700	1.840200	1.800000	1.724900	1.405800	1.008800	0.72
3	0	1.286500	1.049900	0.793600	0.473590	0.111730	-0.054857	-0.062095	-0.120750	-0.10
4	1	-0.175400	-0.121920	-0.053532	-0.024293	0.022917	0.116440	0.187040	0.240710	0.31
...	...	...	...	...	...	...	...	...	...	
1617	1	0.429370	0.603980	0.663720	0.501780	0.323160	0.296740	0.354540	0.537550	0.44
1618	0	-0.552610	-0.382620	-0.331440	-0.240030	-0.227290	-0.208440	-0.146600	-0.050338	-0.05
1619	0	1.716600	1.884900	1.808500	1.750200	1.541700	1.352600	1.107600	0.884450	0.53
1620	0	1.120500	1.227900	1.311500	1.814600	1.954700	1.891200	1.792000	1.437400	1.29
1621	1	1.453000	1.084100	1.098700	1.135100	1.466200	1.482600	0.970650	0.205070	-0.83

1622 rows × 141 columns

The first column of the data is the label. The remaining columns are the features. The features are the different ECG signals. The label is the class of the ECG signal. The classes are:

- 0 (not having a cardiovascular disease)
- 1 (having a cardiovascular disease)

Aim is to build a model that can predict the class of the ECG signal using RNNs.

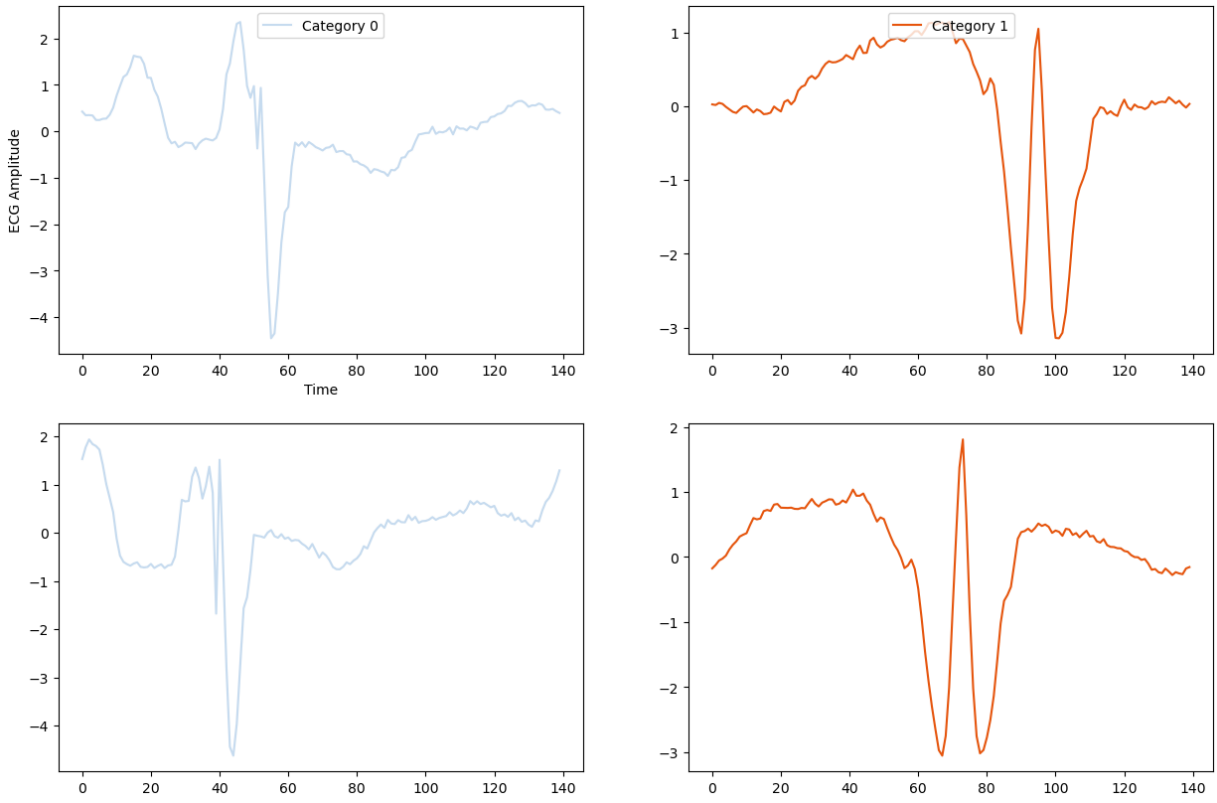
In [47]:
```
X = df.iloc[:, 1:]
# convert X to numpy array
X = X.to_numpy().tolist()
X[0][0]
```

Out[47]: 0.024133

In [48]:
```
y = df.iloc[:, 0]
y = y.to_numpy().tolist()
y[0]
```

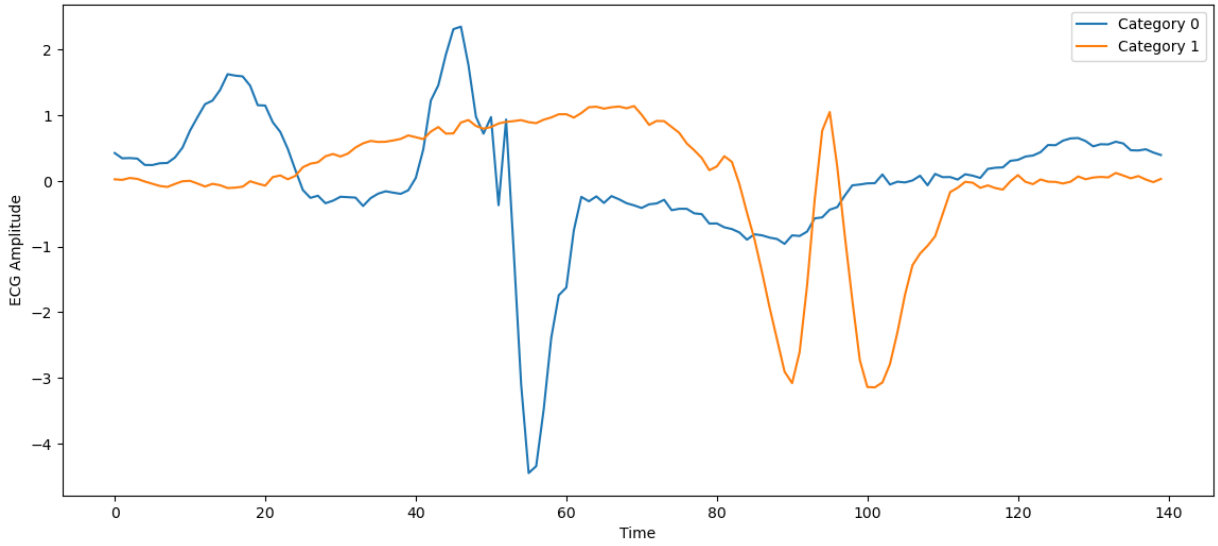
Out[48]: 1

In [49]:
```
# visualizing data
from matplotlib.cm import get_cmap
name = 'tab20c'
cmap = get_cmap(name)
colors_list = cmap.colors

fix, axs = plt.subplots(2, 2, figsize=(15,10))
for i in range(2):
    for j in range(2):
        idx_C = np.argwhere(np.array(y) == j)
        axs[i,j].plot(X[idx_C[i][0]], label=f'Category {j}',
c=colors_list[j+3])
        if i == 0:
            axs[i,j].legend(loc='upper center')
            if j ==0:
                axs[i,j].set_xlabel('Time')
                axs[i,j].set_ylabel('ECG Amplitude')
```



In [50]:

```
#stacked visualization
plt.figure(figsize=(14, 6))
for i in range(2):
    idx_C = np.argwhere(np.array(y) == i)
    plt.plot(X[idx_C[0][0]], label=f'Category {i}')

plt.legend(loc='best')
plt.xlabel('Time')
plt.ylabel('ECG Amplitude')
plt.show()
```



In [51]:

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
X_pad = pad_sequences(X, dtype='float32', padding='post', value=-1)
```

```
print(X_pad.shape)
print(X_pad[0][0])
```

```
(1622, 140)
0.024133
```

In [52]:
```
#reshaping X
X_pad= np.expand_dims(X_pad, axis=-1)
```

In [53]:
```
#classes imbalance
np.unique(y, return_counts=True)
```

Out[53]:
```
(array([0, 1]), array([919, 703], dtype=int64))
```

In [54]:
```
# to categorical
y_cat = to_categorical(y)
y_cat.shape
```

Out[54]:
```
(1622, 2)
```

In [55]:
```
np.unique(y_cat, return_counts=True)
```

Out[55]:
```
(array([0., 1.], dtype=float32), array([1622, 1622], dtype=int64))
```

In [56]:
```
#Splitting data
X_train, X_test, y_train, y_test= train_test_split(X_pad, y_cat,
test_size=0.2, random_state=42)
```

In [14]:
```
#baseline model
sum_ = np.sum(y_train, axis=0)
predicted_category = np.argmax(sum_)

good_prediction = np.sum(y_test, axis=0)[predicted_category]
baseline_result = good_prediction/len(y_test)

print(f'Baseline accuracy: {baseline_result}')
```

```
Baseline accuracy: 0.5692307692307692
```

In [15]:
```
model = Sequential()
model.add(layers.Masking(mask_value=-1., input_shape=(140,1)))
model.add(layers.GRU(units=20, activation="tanh", input_shape=(140,1)))
model.add(layers.Dense(50, activation='relu'))
model.add(layers.Dropout(0.3))
model.add(layers.Dense(2, activation='softmax'))
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
masking (Masking)            (None, 140, 1)            0
_________________________________________________________________
gru (GRU)                    (None, 20)                1380
_________________________________________________________________
dense (Dense)                (None, 50)                1050
_________________________________________________________________
dropout (Dropout)            (None, 50)                0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 102
=================================================================
Total params: 2,532
Trainable params: 2,532
Non-trainable params: 0
_________________________________________________________________
```

In [16]:

```
es = EarlyStopping(patience=5, restore_best_weights=True,
monitor='val_loss')
#compile model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
#fit model
model.fit(X_train, y_train,
          epochs=100,
          batch_size=128,
          verbose=1,
          callbacks = [es],
          validation_split=0.2)
#evaluate model
model.evaluate(X_test, y_test)
```

```
Epoch 1/100
9/9 [==============================] - 9s 282ms/step - loss: 0.6880 - accuracy: 0.55
45 - val_loss: 0.6823 - val_accuracy: 0.6115
Epoch 2/100
9/9 [==============================] - 1s 19ms/step - loss: 0.6834 - accuracy: 0.555
4 - val_loss: 0.6751 - val_accuracy: 0.6115
Epoch 3/100
9/9 [==============================] - 1s 62ms/step - loss: 0.6822 - accuracy: 0.566
1 - val_loss: 0.6746 - val_accuracy: 0.6115
Epoch 4/100
9/9 [==============================] - 1s 66ms/step - loss: 0.6807 - accuracy: 0.568
9 - val_loss: 0.6701 - val_accuracy: 0.6115
Epoch 5/100
9/9 [==============================] - 1s 37ms/step - loss: 0.6776 - accuracy: 0.547
7 - val_loss: 0.6691 - val_accuracy: 0.6115
Epoch 6/100
9/9 [==============================] - 1s 63ms/step - loss: 0.6779 - accuracy: 0.562
2 - val_loss: 0.6656 - val_accuracy: 0.6115
Epoch 7/100
9/9 [==============================] - 0s 61ms/step - loss: 0.6771 - accuracy: 0.552
6 - val_loss: 0.6663 - val_accuracy: 0.6115
```

```
Epoch 8/100
9/9 [==============================] - 1s 72ms/step - loss: 0.6750 - accuracy: 0.552
6 - val_loss: 0.6652 - val_accuracy: 0.6231
Epoch 9/100
9/9 [==============================] - 1s 69ms/step - loss: 0.6721 - accuracy: 0.580
5 - val_loss: 0.6636 - val_accuracy: 0.6038
Epoch 10/100
9/9 [==============================] - 1s 75ms/step - loss: 0.6721 - accuracy: 0.552
6 - val_loss: 0.6642 - val_accuracy: 0.5885
Epoch 11/100
9/9 [==============================] - 1s 65ms/step - loss: 0.6722 - accuracy: 0.568
9 - val_loss: 0.6639 - val_accuracy: 0.5885
Epoch 12/100
9/9 [==============================] - 1s 68ms/step - loss: 0.6693 - accuracy: 0.576
7 - val_loss: 0.6595 - val_accuracy: 0.5692
Epoch 13/100
9/9 [==============================] - 1s 18ms/step - loss: 0.6661 - accuracy: 0.585
3 - val_loss: 0.6571 - val_accuracy: 0.5846
Epoch 14/100
9/9 [==============================] - 1s 71ms/step - loss: 0.6673 - accuracy: 0.577
6 - val_loss: 0.6528 - val_accuracy: 0.6154
Epoch 15/100
9/9 [==============================] - 1s 64ms/step - loss: 0.6666 - accuracy: 0.565
1 - val_loss: 0.6532 - val_accuracy: 0.5808
Epoch 16/100
9/9 [==============================] - 1s 64ms/step - loss: 0.6630 - accuracy: 0.581
5 - val_loss: 0.6541 - val_accuracy: 0.5769
Epoch 17/100
9/9 [==============================] - 1s 17ms/step - loss: 0.6630 - accuracy: 0.567
0 - val_loss: 0.6500 - val_accuracy: 0.5577
Epoch 18/100
9/9 [==============================] - 1s 65ms/step - loss: 0.6585 - accuracy: 0.613
3 - val_loss: 0.6506 - val_accuracy: 0.5731
Epoch 19/100
9/9 [==============================] - 1s 64ms/step - loss: 0.6584 - accuracy: 0.593
1 - val_loss: 0.6525 - val_accuracy: 0.5731
Epoch 20/100
9/9 [==============================] - 1s 18ms/step - loss: 0.6568 - accuracy: 0.598
8 - val_loss: 0.6496 - val_accuracy: 0.5808
Epoch 21/100
9/9 [==============================] - 1s 18ms/step - loss: 0.6588 - accuracy: 0.592
1 - val_loss: 0.6478 - val_accuracy: 0.5885
Epoch 22/100
9/9 [==============================] - 1s 64ms/step - loss: 0.6556 - accuracy: 0.595
9 - val_loss: 0.6466 - val_accuracy: 0.5769
Epoch 23/100
9/9 [==============================] - 1s 63ms/step - loss: 0.6551 - accuracy: 0.597
9 - val_loss: 0.6396 - val_accuracy: 0.5615
Epoch 24/100
9/9 [==============================] - 1s 68ms/step - loss: 0.6469 - accuracy: 0.611
4 - val_loss: 0.6402 - val_accuracy: 0.5731
Epoch 25/100
9/9 [==============================] - 1s 63ms/step - loss: 0.6510 - accuracy: 0.589
2 - val_loss: 0.6359 - val_accuracy: 0.5769
Epoch 26/100
9/9 [==============================] - 1s 74ms/step - loss: 0.6511 - accuracy: 0.601
7 - val_loss: 0.6397 - val_accuracy: 0.5923
Epoch 27/100
9/9 [==============================] - 1s 77ms/step - loss: 0.6470 - accuracy: 0.617
2 - val_loss: 0.6402 - val_accuracy: 0.5731
Epoch 28/100
9/9 [==============================] - 1s 72ms/step - loss: 0.6478 - accuracy: 0.604
6 - val_loss: 0.6361 - val_accuracy: 0.5962
Epoch 29/100
9/9 [==============================] - 1s 75ms/step - loss: 0.6445 - accuracy: 0.600
8 - val_loss: 0.6368 - val_accuracy: 0.6038
Epoch 30/100
9/9 [==============================] - 1s 74ms/step - loss: 0.6429 - accuracy: 0.609
5 - val_loss: 0.6473 - val_accuracy: 0.5846
```

```
11/11 [==============================] - 0s 7ms/step - loss: 0.6419 - accuracy: 0.59
08
```

Out[16]:
```
[0.6418614387512207, 0.5907692313194275]
```

In [17]:
```
testLoss, testAcc = model.evaluate(X_test, y_test)
print('Test accuracy:', testAcc)
print('Test loss:', testLoss)
```

```
11/11 [==============================] - 0s 9ms/step - loss: 0.6419 - accuracy: 0.59
08
Test accuracy: 0.5907692313194275
Test loss: 0.6418614387512207
```

In [18]:
```
#predict
y_pred = model.predict(X_test)
```

In [19]:
```
#predicted classes
pd.DataFrame(y_pred).mean().sort_values()
```

Out[19]:
```
1    0.442444
0    0.557556
dtype: float32
```

In [20]:
```
#actual classes count
pd.value_counts(y,normalize=True, ascending=True)
```

Out[20]:
```
1    0.433416
0    0.566584
dtype: float64
```

In [27]:
```
# confusion matrix
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)
confusion_matrix(y_true_classes, y_pred_classes)
```

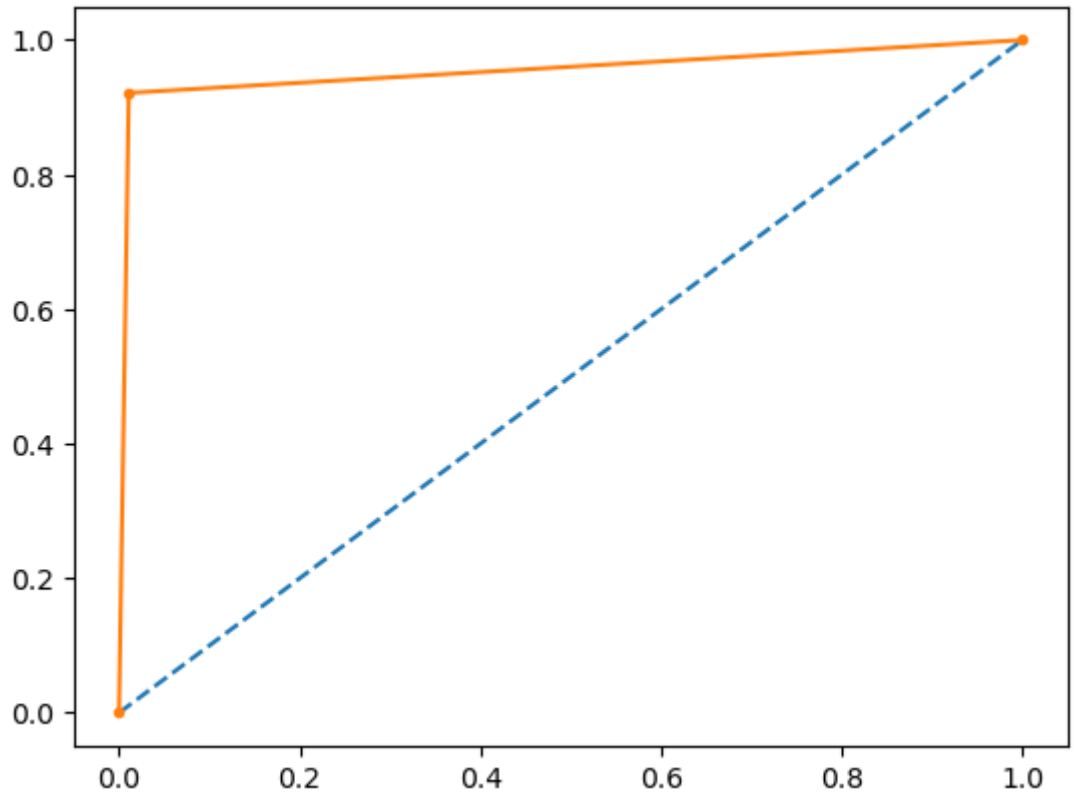
Out[27]:
```
array([[183,   2],
       [ 11, 129]], dtype=int64)
```

In [35]:
```
# Precision, Recall, F1-score, accuracy, and AUC
from sklearn.metrics import classification_report


print(classification_report(y_true_classes, y_pred_classes))


# AUC
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve


# calculate AUC
auc = roc_auc_score(y_true_classes, y_pred_classes)
```

```
print('AUC: %.3f' % auc)
# calculate roc curve
fpr, tpr, thresholds = roc_curve(y_true_classes, y_pred_classes)
# plot no skill
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.')
# show the plot
plt.show()
```

```
              precision    recall  f1-score   support

           0       0.94      0.99      0.97       185
           1       0.98      0.92      0.95       140

    accuracy                           0.96       325
   macro avg       0.96      0.96      0.96       325
weighted avg       0.96      0.96      0.96       325
```

```
AUC: 0.955
```



In [29]:

```
# plot RNN model
from tensorflow.keras.utils import plot_model
plot_model(model, to_file='model_plot.png', show_shapes=True,
show_layer_names=True)
```

```
('You must install pydot (`pip install pydot`) and install graphviz (see instruction
s at https://graphviz.gitlab.io/download/) ', 'for plot_model/model_to_dot to wor
k.')
```

In [21]:

```
# Using LSTM
```

```
model = Sequential()
model.add(layers.Masking(mask_value=-1., input_shape=(140,1)))
model.add(layers.LSTM(units=20, activation="tanh", input_shape=(140,1)))
model.add(layers.Dense(50, activation='relu'))
model.add(layers.Dropout(0.3))
model.add(layers.Dense(2, activation='softmax'))

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
masking_1 (Masking)          (None, 140, 1)            0
_________________________________________________________________
lstm (LSTM)                  (None, 20)                1760
_________________________________________________________________
dense_2 (Dense)              (None, 50)                1050
_________________________________________________________________
dropout_1 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 102
=================================================================
Total params: 2,912
Trainable params: 2,912
Non-trainable params: 0
_________________________________________________________________
```

In [22]:

```
es = EarlyStopping(patience=5, restore_best_weights=True,
monitor='val_loss')
#compile model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
#fit model
model.fit(X_train, y_train,
          epochs=100,
          batch_size=128,
          verbose=1,
          callbacks = [es],
          validation_split=0.2)
#evaluate model
model.evaluate(X_test, y_test)
```

```
Epoch 1/100
9/9 [==============================] - 8s 276ms/step - loss: 0.6867 - accuracy: 0.54
68 - val_loss: 0.6764 - val_accuracy: 0.6077
Epoch 2/100
9/9 [==============================] - 1s 26ms/step - loss: 0.6775 - accuracy: 0.556
4 - val_loss: 0.6739 - val_accuracy: 0.6654
Epoch 3/100
9/9 [==============================] - 1s 91ms/step - loss: 0.6770 - accuracy: 0.573
8 - val_loss: 0.6699 - val_accuracy: 0.6115
Epoch 4/100
9/9 [==============================] - 1s 84ms/step - loss: 0.6705 - accuracy: 0.592
```

```
1 - val_loss: 0.6619 - val_accuracy: 0.6308
Epoch 5/100
9/9 [==============================] - 1s 67ms/step - loss: 0.6669 - accuracy: 0.569
9 - val_loss: 0.6586 - val_accuracy: 0.6538
Epoch 6/100
9/9 [==============================] - 1s 61ms/step - loss: 0.6636 - accuracy: 0.581
5 - val_loss: 0.6522 - val_accuracy: 0.6346
Epoch 7/100
9/9 [==============================] - 0s 60ms/step - loss: 0.6554 - accuracy: 0.604
6 - val_loss: 0.6420 - val_accuracy: 0.6500
Epoch 8/100
9/9 [==============================] - 1s 65ms/step - loss: 0.6418 - accuracy: 0.617
2 - val_loss: 0.6339 - val_accuracy: 0.6462
Epoch 9/100
9/9 [==============================] - 1s 63ms/step - loss: 0.6349 - accuracy: 0.635
5 - val_loss: 0.6134 - val_accuracy: 0.6846
Epoch 10/100
9/9 [==============================] - 1s 24ms/step - loss: 0.6134 - accuracy: 0.633
6 - val_loss: 0.5795 - val_accuracy: 0.7154
Epoch 11/100
9/9 [==============================] - 1s 60ms/step - loss: 0.5724 - accuracy: 0.730
0 - val_loss: 0.4300 - val_accuracy: 0.8808
Epoch 12/100
9/9 [==============================] - 1s 67ms/step - loss: 0.4421 - accuracy: 0.871
7 - val_loss: 0.4164 - val_accuracy: 0.8577
Epoch 13/100
9/9 [==============================] - 1s 66ms/step - loss: 0.3794 - accuracy: 0.901
6 - val_loss: 0.2728 - val_accuracy: 0.9385
Epoch 14/100
9/9 [==============================] - 1s 73ms/step - loss: 0.3325 - accuracy: 0.911
3 - val_loss: 0.2470 - val_accuracy: 0.9462
Epoch 15/100
9/9 [==============================] - 0s 56ms/step - loss: 0.2961 - accuracy: 0.926
7 - val_loss: 0.2191 - val_accuracy: 0.9500
Epoch 16/100
9/9 [==============================] - 1s 22ms/step - loss: 0.2661 - accuracy: 0.928
6 - val_loss: 0.3418 - val_accuracy: 0.8538
Epoch 17/100
9/9 [==============================] - 1s 28ms/step - loss: 0.2826 - accuracy: 0.920
9 - val_loss: 0.2088 - val_accuracy: 0.9423
Epoch 18/100
9/9 [==============================] - 1s 76ms/step - loss: 0.2511 - accuracy: 0.936
4 - val_loss: 0.1850 - val_accuracy: 0.9462
Epoch 19/100
9/9 [==============================] - 1s 79ms/step - loss: 0.2343 - accuracy: 0.934
4 - val_loss: 0.1678 - val_accuracy: 0.9500
Epoch 20/100
9/9 [==============================] - 1s 72ms/step - loss: 0.2453 - accuracy: 0.934
4 - val_loss: 0.1745 - val_accuracy: 0.9615
Epoch 21/100
9/9 [==============================] - 1s 68ms/step - loss: 0.2221 - accuracy: 0.940
2 - val_loss: 0.1573 - val_accuracy: 0.9577
Epoch 22/100
9/9 [==============================] - 1s 63ms/step - loss: 0.2160 - accuracy: 0.937
3 - val_loss: 0.1428 - val_accuracy: 0.9654
Epoch 23/100
9/9 [==============================] - 1s 72ms/step - loss: 0.2071 - accuracy: 0.946
0 - val_loss: 0.1942 - val_accuracy: 0.9538
Epoch 24/100
9/9 [==============================] - 1s 76ms/step - loss: 0.2227 - accuracy: 0.942
1 - val_loss: 0.1299 - val_accuracy: 0.9654
Epoch 25/100
9/9 [==============================] - 1s 68ms/step - loss: 0.2242 - accuracy: 0.938
3 - val_loss: 0.1993 - val_accuracy: 0.9462
Epoch 26/100
9/9 [==============================] - 1s 73ms/step - loss: 0.2114 - accuracy: 0.946
0 - val_loss: 0.1218 - val_accuracy: 0.9692
Epoch 27/100
9/9 [==============================] - 1s 79ms/step - loss: 0.2092 - accuracy: 0.945
```

```
0 - val_loss: 0.1253 - val_accuracy: 0.9654
Epoch 28/100
9/9 [==============================] - 1s 85ms/step - loss: 0.1983 - accuracy: 0.947
0 - val_loss: 0.1375 - val_accuracy: 0.9577
Epoch 29/100
9/9 [==============================] - 1s 81ms/step - loss: 0.2092 - accuracy: 0.942
1 - val_loss: 0.1770 - val_accuracy: 0.9423
Epoch 30/100
9/9 [==============================] - 1s 71ms/step - loss: 0.1914 - accuracy: 0.948
9 - val_loss: 0.1183 - val_accuracy: 0.9731
Epoch 31/100
9/9 [==============================] - 1s 73ms/step - loss: 0.1907 - accuracy: 0.947
9 - val_loss: 0.1882 - val_accuracy: 0.9308
Epoch 32/100
9/9 [==============================] - 1s 76ms/step - loss: 0.1873 - accuracy: 0.951
8 - val_loss: 0.1059 - val_accuracy: 0.9731
Epoch 33/100
9/9 [==============================] - 1s 82ms/step - loss: 0.1816 - accuracy: 0.952
7 - val_loss: 0.1426 - val_accuracy: 0.9577
Epoch 34/100
9/9 [==============================] - 1s 81ms/step - loss: 0.1807 - accuracy: 0.954
7 - val_loss: 0.1014 - val_accuracy: 0.9769
Epoch 35/100
9/9 [==============================] - 1s 21ms/step - loss: 0.1756 - accuracy: 0.956
6 - val_loss: 0.0989 - val_accuracy: 0.9769
Epoch 36/100
9/9 [==============================] - 1s 68ms/step - loss: 0.1700 - accuracy: 0.956
6 - val_loss: 0.0880 - val_accuracy: 0.9846
Epoch 37/100
9/9 [==============================] - 1s 71ms/step - loss: 0.1726 - accuracy: 0.954
7 - val_loss: 0.0852 - val_accuracy: 0.9808
Epoch 38/100
9/9 [==============================] - 1s 66ms/step - loss: 0.1708 - accuracy: 0.957
6 - val_loss: 0.0857 - val_accuracy: 0.9808
Epoch 39/100
9/9 [==============================] - 1s 63ms/step - loss: 0.1644 - accuracy: 0.957
6 - val_loss: 0.1168 - val_accuracy: 0.9615
Epoch 40/100
9/9 [==============================] - 0s 59ms/step - loss: 0.1788 - accuracy: 0.955
6 - val_loss: 0.0873 - val_accuracy: 0.9846
Epoch 41/100
9/9 [==============================] - 1s 19ms/step - loss: 0.1647 - accuracy: 0.961
4 - val_loss: 0.0857 - val_accuracy: 0.9808
Epoch 42/100
9/9 [==============================] - 1s 74ms/step - loss: 0.1622 - accuracy: 0.956
6 - val_loss: 0.1746 - val_accuracy: 0.9500
11/11 [==============================] - 0s 10ms/step - loss: 0.1407 - accuracy: 0.9
600
```

Out[22]:
```
[0.14067725837230682, 0.9599999785423279]
```

In [23]:
```
testLoss, testAcc = model.evaluate(X_test, y_test)
print('Test accuracy:', testAcc)
print('Test loss:', testLoss)
```

```
11/11 [==============================] - 0s 9ms/step - loss: 0.1407 - accuracy: 0.96
00
Test accuracy: 0.9599999785423279
Test loss: 0.14067725837230682
```

In [26]:
```
#predict
y_pred = model.predict(X_test)


#predicted classes
```

```
print(pd.DataFrame(y_pred).mean().sort_values())
print()
#actual classes count
print(pd.value_counts(y,normalize=True, ascending=True))
```

```
1    0.422169
0    0.577831
dtype: float32

1    0.433416
0    0.566584
dtype: float64
```

In [30]:

```
# confusion matrix
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)

confusion_matrix(y_true_classes, y_pred_classes)
```

Out[30]:

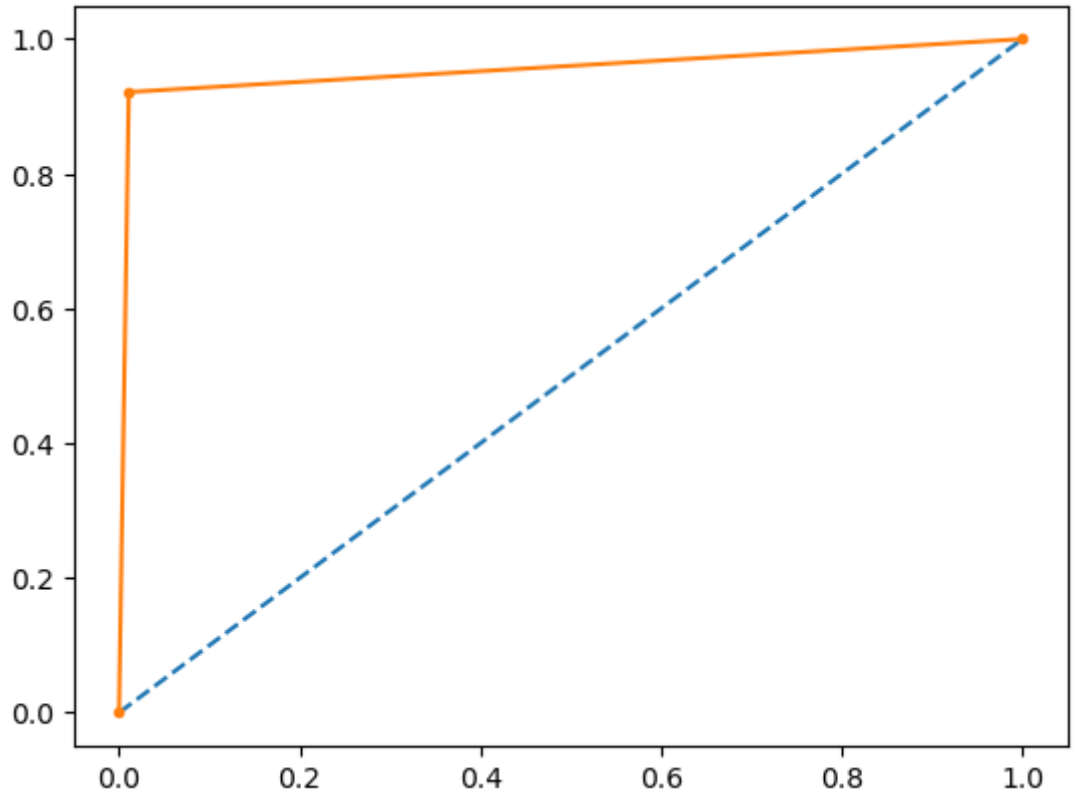
```
array([[183,   2],
       [ 11, 129]], dtype=int64)
```

In [34]:

```
# Precision, Recall, F1-score, accuracy, and AUC
from sklearn.metrics import classification_report

print(classification_report(y_true_classes, y_pred_classes))

# AUC
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve

# calculate AUC
auc = roc_auc_score(y_true_classes, y_pred_classes)
print('AUC: %.3f' % auc)
# calculate roc curve
fpr, tpr, thresholds = roc_curve(y_true_classes, y_pred_classes)
# plot no skill
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.')
# show the plot
plt.show()
```

```
              precision    recall  f1-score   support

           0       0.94      0.99      0.97       185
           1       0.98      0.92      0.95       140

    accuracy                           0.96       325
```

```
   macro avg       0.96      0.96      0.96       325
weighted avg       0.96      0.96      0.96       325

AUC: 0.955
```



In [31]:

```python
# plot RNN model
from tensorflow.keras.utils import plot_model
plot_model(model, to_file='model_plot.png', show_shapes=True,
show_layer_names=True)
```

```
('You must install pydot (`pip install pydot`) and install graphviz (see instruction
s at https://graphviz.gitlab.io/download/) ', 'for plot_model/model_to_dot to wor
k.')
```

From both models (LSMT and GRU):

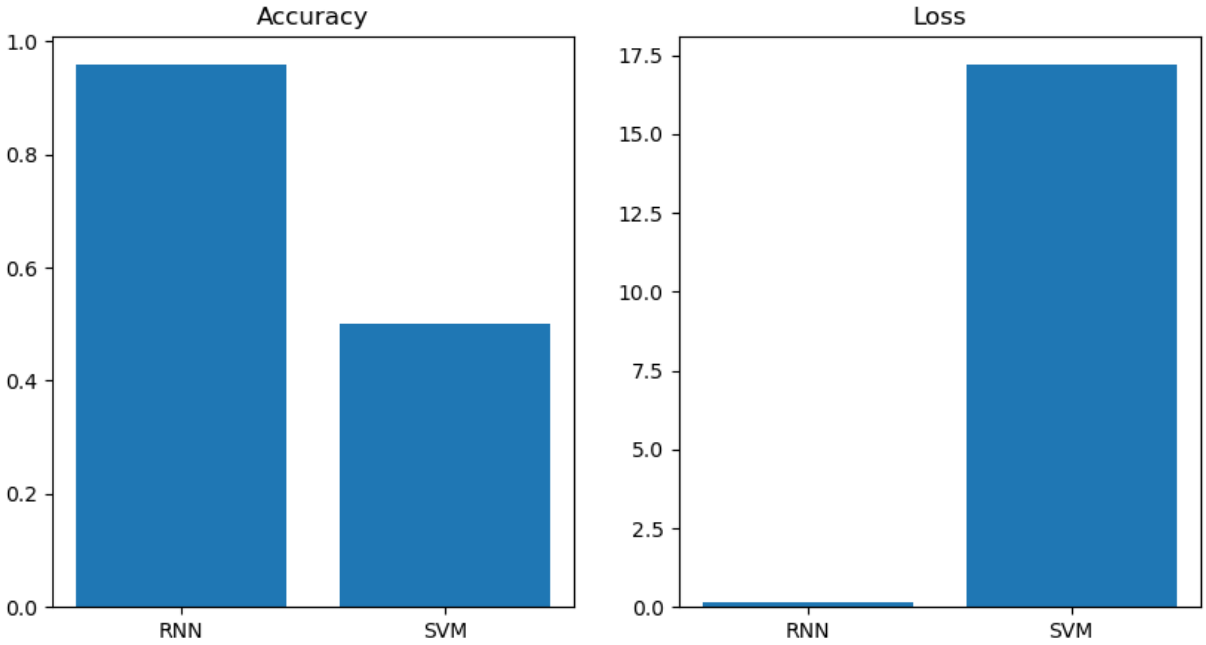
- The accuracy of the model on the test set is higher in the LSTM model than in the GRU model.
- The loss of the model on the test set is lower in the LSTM model than in the GRU model.

In [63]:

```python
# Comparing the RNN with SVM
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

X = df.iloc[:, 1:]
# convert X to numpy array
# X = X.to_numpy().tolist()
# X[0][0]

y = df.iloc[:, 0]
```

```
# y = y.to_numpy().tolist()
# y[0]

# split data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# SVM
svm = SVC(kernel='linear', C=1.0, random_state=0)
svm.fit(X_train, y_train)
y_pred = svm.predict(X_test)
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))

# calculate loss in SVM
from sklearn.metrics import log_loss
svm_loss = log_loss(y_test, y_pred)
print('SVM loss: %.2f' % svm_loss)

# plotting accuracy and loss between RNN and SVM
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.bar(['RNN','SVM'],[testAcc,accuracy_score(y_test, y_pred)])
plt.title('Accuracy')
plt.subplot(1,2,2)
plt.bar(['RNN','SVM'],[testLoss,svm_loss])
plt.title('Loss')
plt.show()
```

```
Accuracy: 0.50
SVM loss: 17.22
```



From the above graphs, we can see that the RNN is better than the SVM. The RNN has a higher accuracy and lower loss than the SVM. The RNN is also faster than the SVM.