Understanding Metrics of Classification

Tanvir Azhar

In this notebook, we make a clear understanding of the different metrics used to determine the performance of any classifier. We will calculate different metrics and see what it represents. We will calculate these metrics by defining our own functions, and also verify them by comparing with `scikit-learn's` library functions.

Lets start with importing libraries we will need in this notebook.

Importing libraries

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn import metrics
import matplotlib.pyplot as plt
import seaborn as sns
import itertools

np.random.seed(42)  # for reproducibility
sns.set(rc={"figure.figsize": (8, 8)})
sns.set_style("ticks")
```

Dataset

For comparing different metrics, we will use the Breast Cancer Dataset which is provided by sckit-learn in `dataset` module. This dataset is used for binary classification between two types of cancer.

```
# Load the dataset
data = load_breast_cancer()
print(data.DESCR)  # print short description
```

```
.. _breast_cancer_dataset:

Breast cancer wisconsin (diagnostic) dataset
--------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 569

    :Number of Attributes: 30 numeric, predictive attributes and the
class
```

```
    :Attribute Information:
        - radius (mean of distances from center to points on the
perimeter)
        - texture (standard deviation of gray-scale values)
        - perimeter
        - area
        - smoothness (local variation in radius lengths)
        - compactness (perimeter^2 / area - 1.0)
        - concavity (severity of concave portions of the contour)
        - concave points (number of concave portions of the contour)
        - symmetry
        - fractal dimension ("coastline approximation" - 1)

        The mean, standard error, and "worst" or largest (mean of the
three
        worst/largest values) of these features were computed for each
image,
        resulting in 30 features.  For instance, field 0 is Mean
Radius, field
        10 is Radius SE, field 20 is Worst Radius.

        - class:
                - WDBC-Malignant
                - WDBC-Benign

    :Summary Statistics:

    ===================================== ====== ======
                                           Min    Max
    ===================================== ====== ======
    radius (mean):                        6.981  28.11
    texture (mean):                       9.71   39.28
    perimeter (mean):                     43.79  188.5
    area (mean):                          143.5  2501.0
    smoothness (mean):                    0.053  0.163
    compactness (mean):                   0.019  0.345
    concavity (mean):                     0.0    0.427
    concave points (mean):                0.0    0.201
    symmetry (mean):                      0.106  0.304
    fractal dimension (mean):             0.05   0.097
    radius (standard error):              0.112  2.873
    texture (standard error):             0.36   4.885
    perimeter (standard error):           0.757  21.98
    area (standard error):                6.802  542.2
    smoothness (standard error):          0.002  0.031
    compactness (standard error):         0.002  0.135
    concavity (standard error):           0.0    0.396
    concave points (standard error):      0.0    0.053
    symmetry (standard error):            0.008  0.079
```

```
    fractal dimension (standard error):   0.001  0.03
    radius (worst):                       7.93   36.04
    texture (worst):                      12.02  49.54
    perimeter (worst):                    50.41  251.2
    area (worst):                         185.2  4254.0
    smoothness (worst):                   0.071  0.223
    compactness (worst):                  0.027  1.058
    concavity (worst):                    0.0    1.252
    concave points (worst):               0.0    0.291
    symmetry (worst):                     0.156  0.664
    fractal dimension (worst):            0.055  0.208
    ================================== ====== ======

    :Missing Attribute Values: None

    :Class Distribution: 212 - Malignant, 357 - Benign

    :Creator:  Dr. William H. Wolberg, W. Nick Street, Olvi L.
Mangasarian

    :Donor: Nick Street

    :Date: November, 1995

This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic)
datasets.
https://goo.gl/U2Uwz2

Features are computed from a digitized image of a fine needle
aspirate (FNA) of a breast mass.  They describe
characteristics of the cell nuclei present in the image.

Separating plane described above was obtained using
Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree
Construction Via Linear Programming." Proceedings of the 4th
Midwest Artificial Intelligence and Cognitive Science Society,
pp. 97-101, 1992], a classification method which uses linear
programming to construct a decision tree.  Relevant features
were selected using an exhaustive search in the space of 1-4
features and 1-3 separating planes.

The actual linear program used to obtain the separating plane
in the 3-dimensional space is that described in:
[K. P. Bennett and O. L. Mangasarian: "Robust Linear
Programming Discrimination of Two Linearly Inseparable Sets",
Optimization Methods and Software 1, 1992, 23-34].

This database is also available through the UW CS ftp server:

ftp ftp.cs.wisc.edu
```

```
cd math-prog/cpo-dataset/machine-learn/WDBC/

.. topic:: References

   - W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature
extraction
     for breast tumor diagnosis. IS&T/SPIE 1993 International
Symposium on
     Electronic Imaging: Science and Technology, volume 1905, pages
861-870,
     San Jose, CA, 1993.
   - O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer
diagnosis and
     prognosis via linear programming. Operations Research, 43(4),
pages 570-577,
     July-August 1995.
   - W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning
techniques
     to diagnose breast cancer from fine-needle aspirates. Cancer
Letters 77 (1994)
     163-171.
```

Let's see what are the targets classes i.e. types of cancer.

```
print(f"Types of cancer (targets) are {data.target_names}")
```

```
Types of cancer (targets) are ['malignant' 'benign']
```

```
X = data.data
X.shape
```

```
(569, 30)
```

What is the dimension of data? We see there are **569 examples** and each example has **30 features**. The **target** variable is binary (**0 and 1** for `malignant` and `benign`). We will consider `benign` as `positive` class, and `malignant` as `negative` class. We can understand it as: `0 for is_not_benign` `1 for is_benign`

```
X = data.data  # features
y = data.target  # labels
print(f"Shape of features is {X.shape}, and shape of target is
{y.shape}")
```

```
Shape of features is (569, 30), and shape of target is (569,)
```

Split the data

Since we shouldn't train and test our model with the same dataset, it is always a good idea to split the data in three parts - train data, test data, and validation data. We won't require

validation data here. We split the dataset into training and testing data, with 369 examples for training, and 200 examples for testing.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=200, random_state=42, stratify=y)

y_train[:10]

array([1, 1, 1, 1, 0, 1, 1, 1, 0, 1])

X_train[:10]

array([[1.387e+01, 2.070e+01, 8.977e+01, 5.848e+02, 9.578e-02, 1.018e-
01,
        3.688e-02, 2.369e-02, 1.620e-01, 6.688e-02, 2.720e-01,
1.047e+00,
        2.076e+00, 2.312e+01, 6.298e-03, 2.172e-02, 2.615e-02, 9.061e-
03,
        1.490e-02, 3.599e-03, 1.505e+01, 2.475e+01, 9.917e+01,
6.886e+02,
        1.264e-01, 2.037e-01, 1.377e-01, 6.845e-02, 2.249e-01, 8.492e-
02],
       [1.176e+01, 2.160e+01, 7.472e+01, 4.279e+02, 8.637e-02, 4.966e-
02,
        1.657e-02, 1.115e-02, 1.495e-01, 5.888e-02, 4.062e-01,
1.210e+00,
        2.635e+00, 2.847e+01, 5.857e-03, 9.758e-03, 1.168e-02, 7.445e-
03,
        2.406e-02, 1.769e-03, 1.298e+01, 2.572e+01, 8.298e+01,
5.165e+02,
        1.085e-01, 8.615e-02, 5.523e-02, 3.715e-02, 2.433e-01, 6.563e-
02],
       [1.495e+01, 1.877e+01, 9.784e+01, 6.895e+02, 8.138e-02, 1.167e-
01,
        9.050e-02, 3.562e-02, 1.744e-01, 6.493e-02, 4.220e-01,
1.909e+00,
        3.271e+00, 3.943e+01, 5.790e-03, 4.877e-02, 5.303e-02, 1.527e-
02,
        3.356e-02, 9.368e-03, 1.625e+01, 2.547e+01, 1.071e+02,
8.097e+02,
        9.970e-02, 2.521e-01, 2.500e-01, 8.405e-02, 2.852e-01, 9.218e-
02],
       [1.203e+01, 1.793e+01, 7.609e+01, 4.460e+02, 7.683e-02, 3.892e-
02,
        1.546e-03, 5.592e-03, 1.382e-01, 6.070e-02, 2.335e-01, 9.097e-
01,
        1.466e+00, 1.697e+01, 4.729e-03, 6.887e-03, 1.184e-03, 3.951e-
03,
        1.466e-02, 1.755e-03, 1.307e+01, 2.225e+01, 8.274e+01,
5.234e+02,
```

```
        1.013e-01, 7.390e-02, 7.732e-03, 2.796e-02, 2.171e-01, 7.037e-
02],
       [1.348e+01, 2.082e+01, 8.840e+01, 5.592e+02, 1.016e-01, 1.255e-
01,
        1.063e-01, 5.439e-02, 1.720e-01, 6.419e-02, 2.130e-01, 5.914e-
01,
        1.545e+00, 1.852e+01, 5.367e-03, 2.239e-02, 3.049e-02, 1.262e-
02,
        1.377e-02, 3.187e-03, 1.553e+01, 2.602e+01, 1.073e+02,
7.404e+02,
        1.610e-01, 4.225e-01, 5.030e-01, 2.258e-01, 2.807e-01, 1.071e-
01],
       [1.086e+01, 2.148e+01, 6.851e+01, 3.605e+02, 7.431e-02, 4.227e-
02,
        0.000e+00, 0.000e+00, 1.661e-01, 5.948e-02, 3.163e-01,
1.304e+00,
        2.115e+00, 2.067e+01, 9.579e-03, 1.104e-02, 0.000e+00,
0.000e+00,
        3.004e-02, 2.228e-03, 1.166e+01, 2.477e+01, 7.408e+01,
4.123e+02,
        1.001e-01, 7.348e-02, 0.000e+00, 0.000e+00, 2.458e-01, 6.592e-
02],
       [1.157e+01, 1.904e+01, 7.420e+01, 4.097e+02, 8.546e-02, 7.722e-
02,
        5.485e-02, 1.428e-02, 2.031e-01, 6.267e-02, 2.864e-01,
1.440e+00,
        2.206e+00, 2.030e+01, 7.278e-03, 2.047e-02, 4.447e-02, 8.799e-
03,
        1.868e-02, 3.339e-03, 1.307e+01, 2.698e+01, 8.643e+01,
5.205e+02,
        1.249e-01, 1.937e-01, 2.560e-01, 6.664e-02, 3.035e-01, 8.284e-
02],
       [1.094e+01, 1.859e+01, 7.039e+01, 3.700e+02, 1.004e-01, 7.460e-
02,
        4.944e-02, 2.932e-02, 1.486e-01, 6.615e-02, 3.796e-01,
1.743e+00,
        3.018e+00, 2.578e+01, 9.519e-03, 2.134e-02, 1.990e-02, 1.155e-
02,
        2.079e-02, 2.701e-03, 1.240e+01, 2.558e+01, 8.276e+01,
4.724e+02,
        1.363e-01, 1.644e-01, 1.412e-01, 7.887e-02, 2.251e-01, 7.732e-
02],
       [1.969e+01, 2.125e+01, 1.300e+02, 1.203e+03, 1.096e-01, 1.599e-
01,
        1.974e-01, 1.279e-01, 2.069e-01, 5.999e-02, 7.456e-01, 7.869e-
01,
        4.585e+00, 9.403e+01, 6.150e-03, 4.006e-02, 3.832e-02, 2.058e-
02,
        2.250e-02, 4.571e-03, 2.357e+01, 2.553e+01, 1.525e+02,
```

```
1.709e+03,
        1.444e-01, 4.245e-01, 4.504e-01, 2.430e-01, 3.613e-01, 8.758e-
02],
       [1.277e+01, 2.141e+01, 8.202e+01, 5.074e+02, 8.749e-02, 6.601e-
02,
        3.112e-02, 2.864e-02, 1.694e-01, 6.287e-02, 7.311e-01,
1.748e+00,
        5.118e+00, 5.365e+01, 4.571e-03, 1.790e-02, 2.176e-02, 1.757e-
02,
        3.373e-02, 5.875e-03, 1.375e+01, 2.350e+01, 8.904e+01,
5.795e+02,
        9.388e-02, 8.978e-02, 5.186e-02, 4.773e-02, 2.179e-01, 6.871e-
02]])
```

Training and predicting data

In this example, we will use scikit's Support Vector Machines classifier to predict whether its a benign cancer. `SVC` classifier is used from `sklearn.svm`. Ofcourse, we can try any of the other classifiers and compare accuracies.

```
classifier = svm.SVC(kernel='linear', C=1.0, probability=True,
verbose=True)
```

Next we fit/train the model on our *training dataset*. It trains quite fast since we are working with relatively small dataset.

```
classifier.fit(X_train, y_train)
```

```
[LibSVM]
```

```
SVC(kernel='linear', probability=True, verbose=True)
```

Now save the prediction results both as probability and as classes. **`y_preds`** is a **1D vector** of one of `{0, 1}` values, denoting predictions as malignant and benign, respectively. **`y_proba`** is a **2D vector**, where for each example, it contains a vector of length 2, `[prob. of malignant, prob. of benign]`

```
y_preds = classifier.predict(X_test)
y_proba = classifier.predict_proba(X_test)

print(y_preds)
```

```
[1 0 1 1 1 0 1 0 1 1 1 1 1 1 0 1 1 1 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1
1 1
 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 1 1 1 0 0 1 1 0 0 1 0 1
0 0
 1 1 1 1 1 0 1 1 1 1 0 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 1 1 0 1 0 1
0 1
 1 0 0 1 1 0 1 1 0 1 0 1 1 0 1 1 1 1 1 1 1 0 1 0 1 0 0 1 1 0 1 0 1 0 1
```

```
1 1
 1 1 0 0 0 1 1 1 1 1 0 1 0 1 0 0 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 0 0 0 1
1 0
 0 1 1 0 0 0 1 1 0 1 0 1 1 1 1]
```

```
print(y_proba)
```

```
[9.96972624e-01 8.13664441e-02 9.99991677e-01 9.68093925e-01
 9.99999072e-01 4.54132111e-02 8.03882974e-01 2.84795425e-03
 7.28905839e-01 6.90721596e-01 9.93326132e-01 8.30878631e-01
 9.99996329e-01 9.26147269e-01 9.02829328e-01 6.90841228e-03
 9.96388752e-01 9.88084937e-01 9.99999189e-01 1.12727360e-01
 9.21705236e-01 5.49672567e-01 1.35902753e-01 9.80909671e-01
 2.54752914e-01 1.21386513e-02 9.76246816e-01 6.63683734e-05
 6.37522729e-06 4.73538747e-02 1.20073583e-02 9.49481736e-01
 9.77798898e-01 7.17704642e-01 9.28702027e-01 9.83059195e-01
 1.00000000e+00 6.83766802e-01 1.99385948e-03 9.99999460e-01
 7.83125389e-01 9.85576088e-01 9.92488973e-01 9.64853732e-01
 9.86450934e-01 9.35995895e-01 9.97375872e-01 8.75016573e-01
 9.92737883e-01 8.55429292e-01 9.92755991e-01 9.55709893e-01
 9.30349646e-01 8.38595705e-01 7.87411570e-01 2.08152930e-04
 9.89863823e-01 1.27251341e-02 2.86689276e-02 2.40110896e-03
 9.79530625e-01 9.87370186e-01 9.86028468e-01 8.63879733e-05
 4.11089629e-01 9.74126646e-01 9.88886518e-01 8.50358138e-05
 4.32902943e-07 8.70549366e-01 9.97811356e-02 9.59832128e-01
 4.00438253e-01 1.15602523e-04 9.75994958e-01 9.59449950e-01
 9.21331427e-01 9.80789828e-01 9.92234983e-01 1.01056153e-03
 9.52420060e-01 6.29956933e-01 9.90402010e-01 7.13682268e-01
 2.13465252e-01 7.82613258e-01 5.07039070e-06 1.54884367e-04
 9.95380571e-01 9.95530603e-01 9.86921478e-01 9.95414544e-01
 9.53468005e-01 2.64675340e-07 8.19239432e-07 1.09390467e-02
 9.54845030e-01 6.84917161e-01 9.99999390e-01 9.74533035e-01
 6.70090606e-01 9.91837445e-01 1.04603326e-03 9.99983251e-01
 9.91062829e-01 1.35289659e-01 9.95551438e-01 7.36863850e-02
 9.99998383e-01 3.14855246e-04 9.99999697e-01 9.77684287e-01
 1.00000010e-07 1.41278126e-06 9.91402778e-01 9.55579894e-01
 2.88919350e-01 9.78499237e-01 9.55809206e-01 1.24975040e-01
 9.59805347e-01 4.48111821e-06 9.94341035e-01 9.33824127e-01
 2.22609198e-03 9.91451673e-01 7.98227751e-01 9.91386008e-01
 9.96258870e-01 9.39262265e-01 9.27253989e-01 9.17695311e-01
 4.07297538e-04 8.57306684e-01 8.39921854e-02 9.99988381e-01
 1.64686285e-05 1.90744213e-01 9.94017125e-01 9.92402499e-01
 2.94686292e-02 9.99994354e-01 1.15008339e-05 9.69511927e-01
 1.00000010e-07 9.22948185e-01 7.60457907e-01 9.77442725e-01
 9.60469220e-01 9.49711303e-01 5.51682638e-01 1.80118247e-04
 2.41393278e-02 9.55641023e-01 9.93311953e-01 9.84970048e-01
 9.57997991e-01 9.68204467e-01 3.32754880e-05 9.88655313e-01
 1.00000010e-07 7.94292803e-01 2.56546982e-02 4.63356614e-03
 9.86015580e-01 9.99991038e-01 4.01599634e-03 9.11694900e-01
 9.96262892e-01 9.05578447e-01 2.44651039e-04 1.53168485e-02
```

```
 9.63460063e-01 8.70313909e-01 2.06848642e-02 9.34147880e-01
 9.86558877e-01 8.99263981e-01 9.99985937e-01 1.11826582e-04
 1.56215563e-05 2.13294993e-02 9.96858076e-01 9.51353393e-01
 6.09602731e-02 1.00000010e-07 9.70250680e-01 9.99997036e-01
 1.84154632e-01 1.00000010e-07 1.00000010e-07 7.98650395e-01
 9.99998398e-01 1.30988273e-03 9.99983850e-01 3.91110041e-04
 9.96589822e-01 9.99990679e-01 7.30197181e-01 9.99988903e-01]
```

We need to reshape `y_proba` to a 1D vector denoting the probability of having `benign` cancer.

```
y_proba = y_proba[:,1].reshape((y_proba.shape[0],))

y_proba[:5], y_preds[:5], y_test[:5]
```

```
(array([0.99697262, 0.08136644, 0.99999168, 0.96809392, 0.99999907]),
 array([1, 0, 1, 1, 1]),
 array([1, 0, 1, 1, 1]))
```

Confusion Matrix

Let us calculate confusion matrix of the predictions. It is implemented in Scikit-learn's `sklearn.metrics.confusion_matrix`.

```
conf = metrics.confusion_matrix(y_test, y_preds)
conf
```

```
array([[ 68,   7],
       [  2, 123]])
```

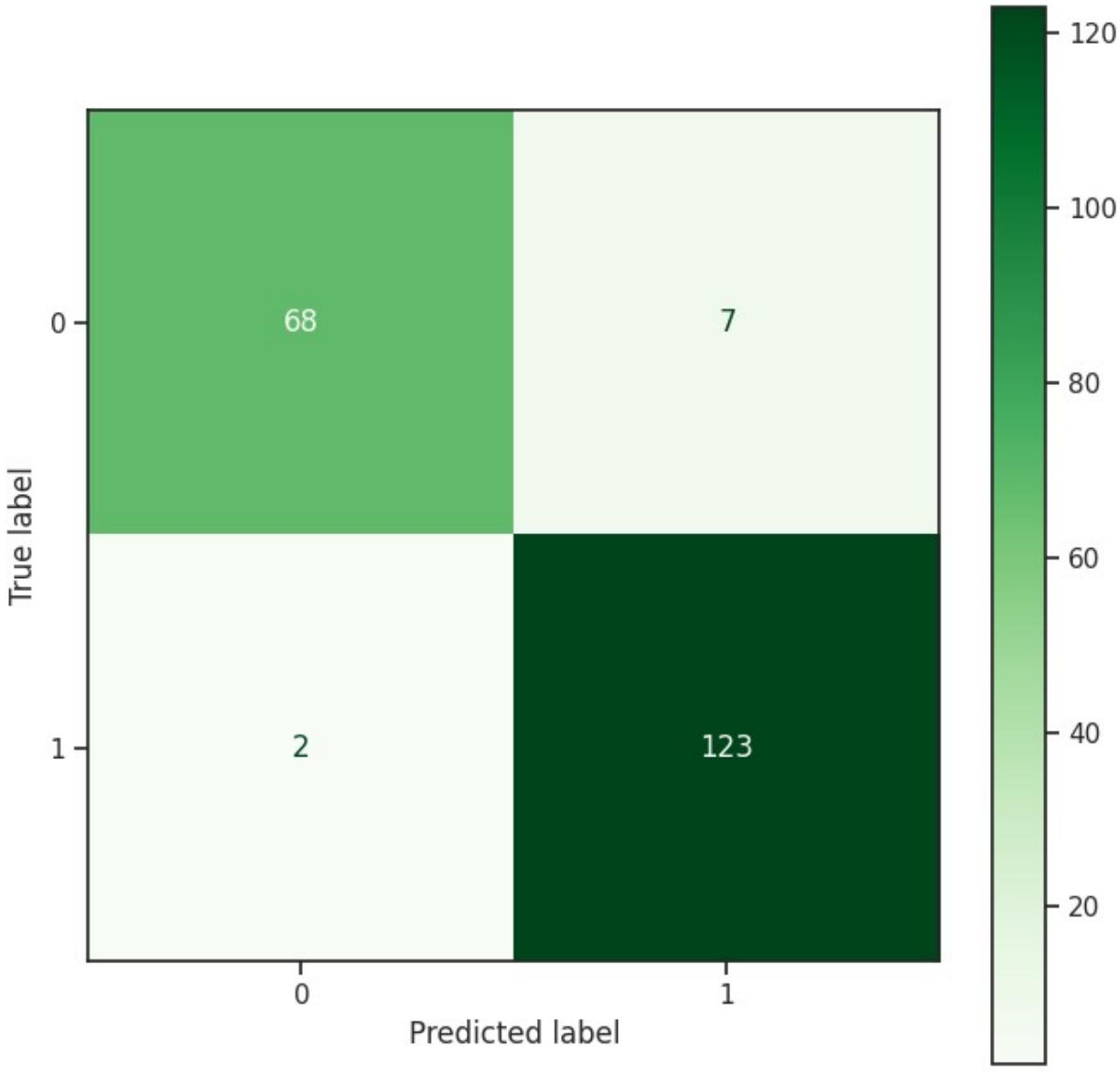
We can also implement our own confusion matrix. Here's my implementation.

```
def get_confusion_matrix(y_true, y_pred):
    n_classes = len(np.unique(y_true))
    conf = np.zeros((n_classes, n_classes))
    for actual, pred in zip(y_true, y_pred):
        conf[int(actual)][int(pred)] += 1
    return conf.astype('int')

conf = get_confusion_matrix(y_test, y_preds)
conf
```

```
array([[ 68,   7],
       [  2, 123]])
```

```
from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_estimator(classifier,X_test,y_test,cmap=pl
t.cm.Greens)
```

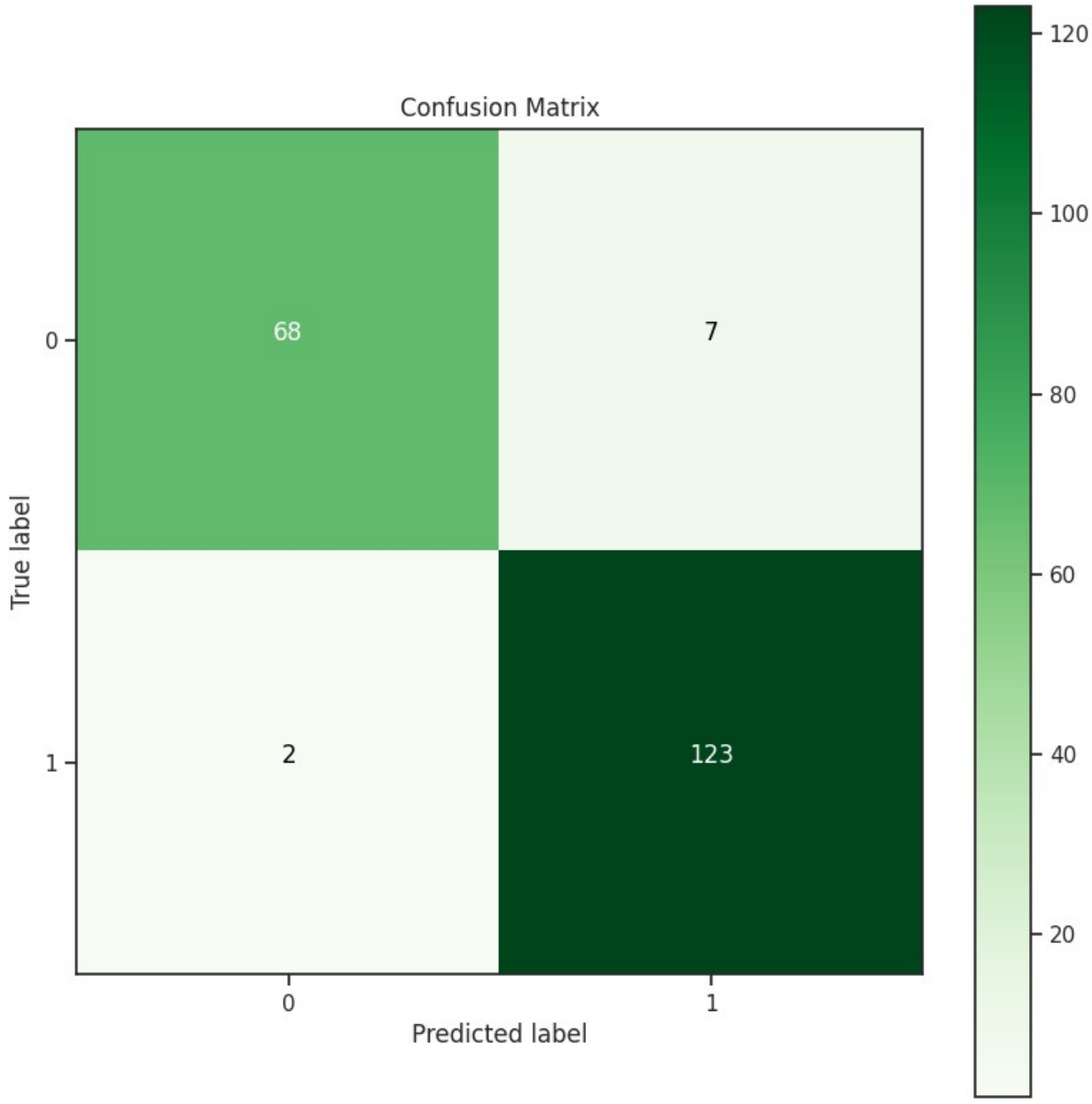
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7ee4cd8cc430>
```




```
classes = [0, 1]
# plot confusion matrix
plt.imshow(conf, interpolation='nearest', cmap=plt.cm.Greens)
plt.title("Confusion Matrix")
plt.colorbar()
tick_marks = np.arange(len(classes))
plt.xticks(tick_marks, classes)
plt.yticks(tick_marks, classes)

fmt = 'd'
```

```
thresh = conf.max() / 2.
for i, j in itertools.product(range(conf.shape[0]),
range(conf.shape[1])):
    plt.text(j, i, format(conf[i, j], fmt),
             horizontalalignment="center",
             color="white" if conf[i, j] > thresh else "black")

plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')
```

```
Text(0.5, 114.24999999999993, 'Predicted label')
```



From the confusion matrix, we can see the number of examples predicted correct by our classifier, for both classes seperately. We can get the numbers of `True Positives, True Negatives, False Positives, and False Negatives` from this confusion matrix. Lets store these terms in some variables.

```
# from the confusion matrix
TP = true_pos = 123
TN = true_neg = 68
FP = false_pos = 7
FN = false_neg = 2
```

Some basic metrics

Now, we will calculate some basic metrics from these four values. We will need a dictionary to store these metrics. Lets create a dictionary `results`.

```
results = {}
```

Accuracy

number of examples correctly predicted / total number of examples

```
metric = "ACC"
results[metric] = (TP + TN) / (TP + TN + FP + FN)
print(f"{metric} is {results[metric]: .3f}")
```

```
ACC is  0.955
```

```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_preds))
```

```
              precision    recall  f1-score   support
```

```
           0       0.97      0.91      0.94        75
           1       0.95      0.98      0.96       125

    accuracy                           0.95       200
   macro avg       0.96      0.95      0.95       200
weighted avg       0.96      0.95      0.95       200
```

True Positive Rate

number of samples actually and predicted as `Positive` / total number of samples actually `Positive` Also called **Sensitivity or Recall**.

```
# Sensitivity or Recall
metric = "TPR"
results[metric] = TP / (TP + FN)
print(f"{metric} is {results[metric]: .3f}")
```

```
TPR is  0.984
```

True Negative Rate

number of samples actually and predicted as `Negative` / total number of samples actually `Negative` Also called **Specificity**.

```
# Specificity
metric = "TNR"
results[metric] = TN / (TN + FP)
print(f"{metric} is {results[metric]: .3f}")
```

```
TNR is  0.907
```

Positive Predictive Value

number of samples actually and predicted as `Positive` / total number of samples predicted as `Positive` Also called **Precision**.

```
# Precision
metric = "PPV"
results[metric] = TP / (TP + FP)
print(f"{metric} is {results[metric]: .3f}")
```

```
PPV is  0.946
```

Negative Predictive Value

number of samples actually and predicted as `Negative` / total number of samples predicted as

`Negative`

```
metric = "NPV"
results[metric] = TN / (TN + FN)
print(f"{metric} is {results[metric]: .3f}")
```

```
NPV is  0.971
```

F1 score

Harmonic Mean of Precision and Recall.

```
metric = "F1"
results[metric] = 2 / (1 / results["PPV"] + 1 / results["TPR"])
print(f"{metric} is {results[metric]: .3f}")
```

```
F1 is  0.965
```

Matthew's correlation coefficient

Matthew's coefficient range between `[-1, 1]`. `0` usually means totally random predictions. `1` means a perfect classifier, while a negative value (`[-1, 0)`) suggests a negative correlation between predictions and actual values. Here' the formual for MCC

```
metric = "MCC"
num = TP * TN - FP * FN
```

```
den = ((TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)) ** 0.5
results[metric] = num / den
print(f"{metric} is {results[metric]: .3f}")
```

```
MCC is  0.904
```

Comparing these calculated metrics

Let's check if these values match with the values calculated from `scikit-learn`'s functions.

```
print(f"Calculated and Actual Accuracy:
{results['ACC']: .3f}, {metrics.accuracy_score(y_test,
y_preds): .3f}")
print(f"Calculated and Actual Precision score:
{results['PPV']: .3f}, {metrics.precision_score(y_test,
y_preds): .3f}")
print(f"Calculated and Actual Recall score:
{results['TPR']: .3f}, {metrics.recall_score(y_test, y_preds): .3f}")
print(f"Calculated and Actual F1 score:
{results['F1']: .3f}, {metrics.f1_score(y_test, y_preds): .3f}")
print(f"Calculated and Actual Matthew's correlation coefficient:
{results['MCC']: .3f}, {metrics.matthews_corrcoef(y_test,
y_preds): .3f}")
```

```
Calculated and Actual Accuracy:                          0.955,
0.955
Calculated and Actual Precision score:                   0.946,
0.946
Calculated and Actual Recall score:                      0.984,
0.984
Calculated and Actual F1 score:                          0.965,
0.965
Calculated and Actual Matthew's correlation coefficient:  0.904,
0.904
```

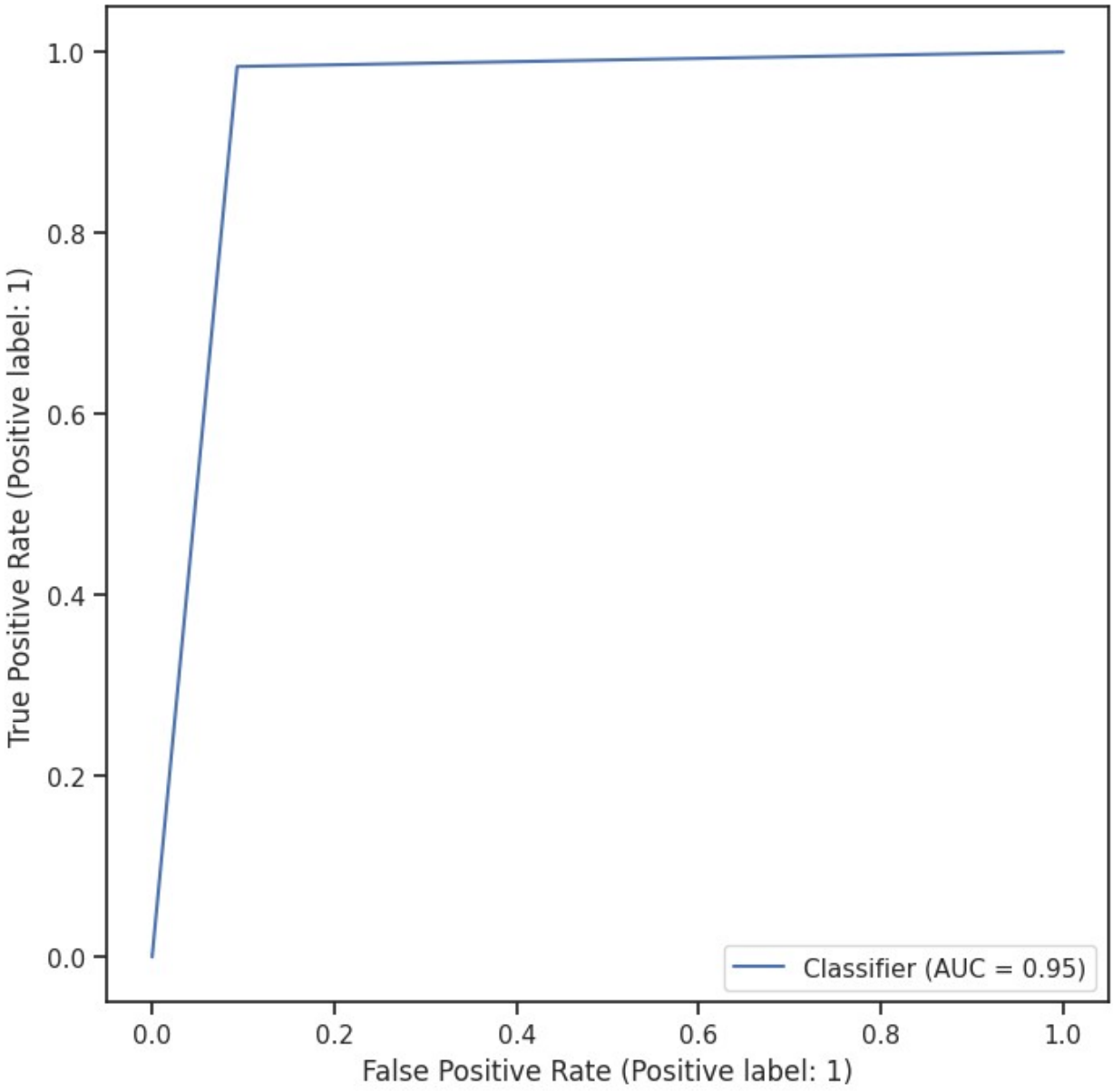
ROC curve (Receiver Operating Characteristic curve)

A receiver operating characteristic curve, i.e. **ROC curve**, is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The ROC curve is created by plotting the **True Positive Rate (TPR)** against the **False Positive Rate (FPR)** at various threshold settings. Let's plot the ROC curve for the Breast Cancer Dataset.

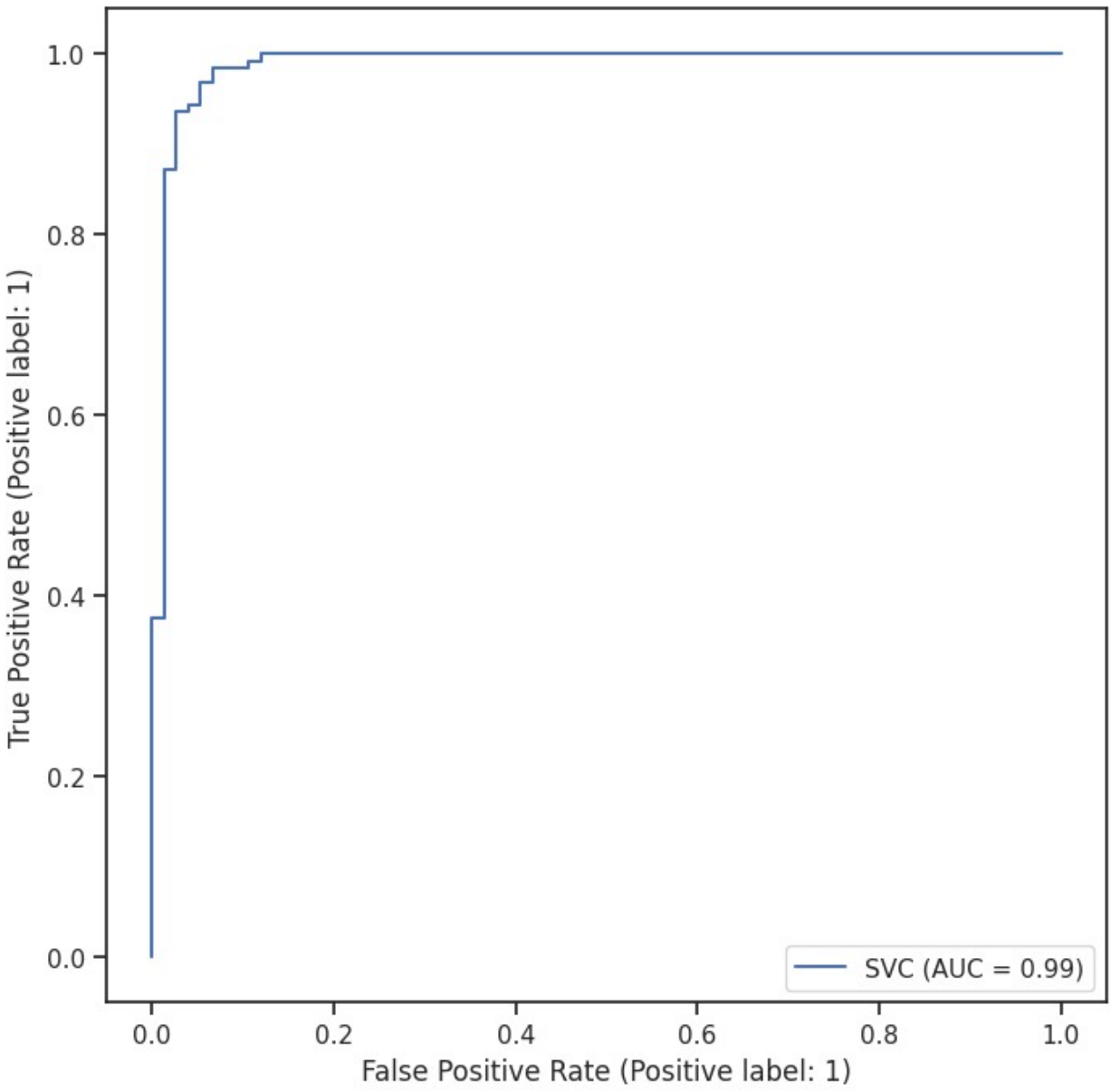
First we will create an ROC curve by calculating values manually at intervaled **thresholds**, then we will compare our results with the `scikit-learn`'s built-in `metrics.roc_curve`.

```
from sklearn.metrics import RocCurveDisplay

RocCurveDisplay.from_predictions(y_test, y_preds)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7ee50c91d780>
```




```
from sklearn.metrics import RocCurveDisplay
RocCurveDisplay.from_estimator(classifier,X_test,y_test)

<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7ee50c91d5d0>
```

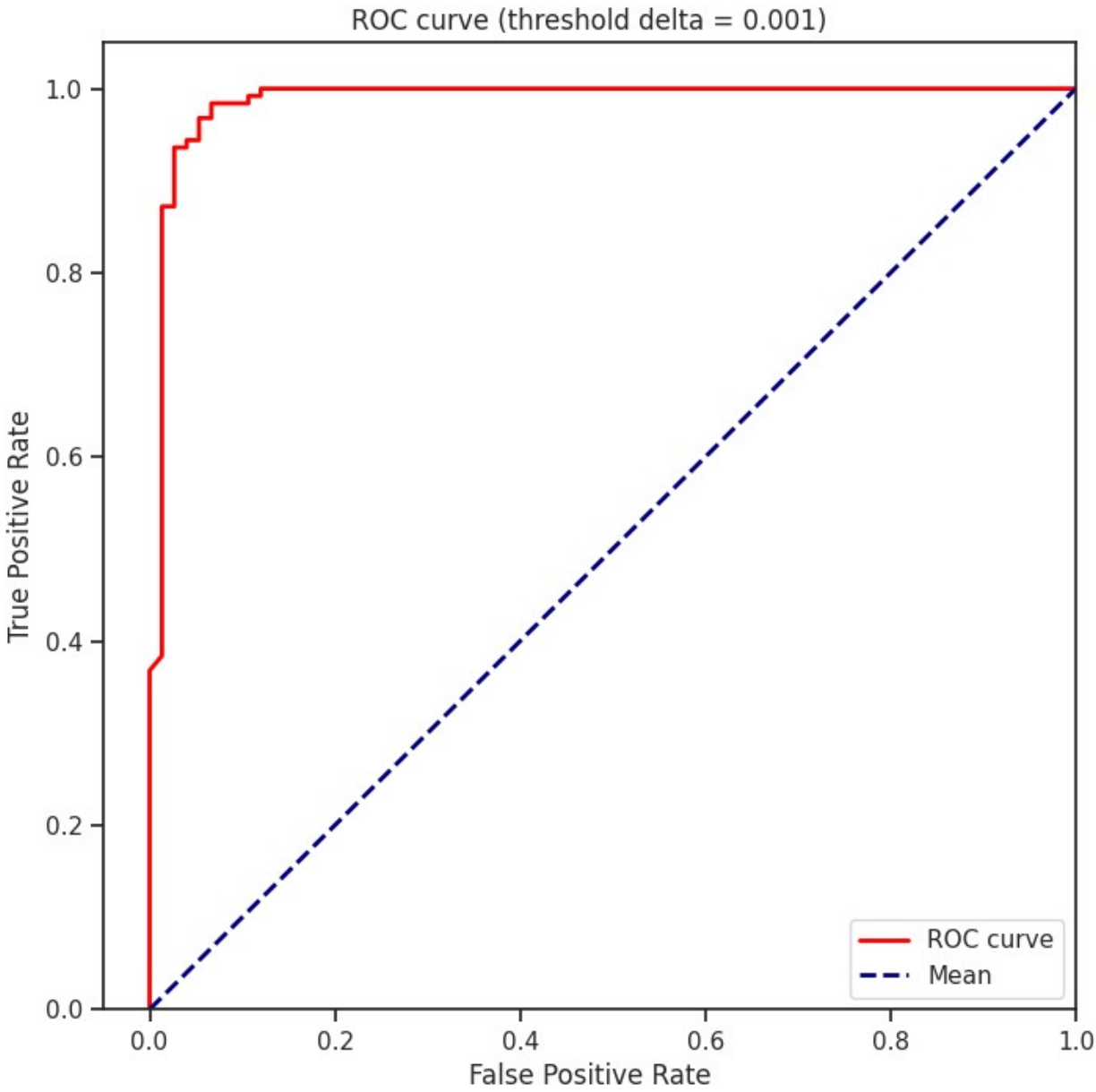



```
def get_roc_curve(y_test, y_proba, delta=0.1):
    """
    Return the True Positive Rates (TPRs), False Positive Rates
(FPRs),
    and the threshold values, seperated by delta.
    """
    thresh = list(np.arange(0, 1, delta)) + [1]
    TPRs = []
    FPRs = []
    y_pred = np.empty(y_proba.shape)
    for th in thresh:
        y_pred[y_proba < th] = 0
        y_pred[y_proba >= th] = 1
```

```
        # confusion matrix from the function we defined
        (TN, FP), (FN, TP) = get_confusion_matrix(y_test, y_pred)

        TPR = TP / (TP + FN)  # sensitivity
        FPR = FP / (FP + TN)  # 1 - specificity
        TPRs.append(TPR)
        FPRs.append(FPR)
    return FPRs, TPRs, thresh
```

Here, `delta` is the difference between each of the thresholds at which ROC curve is calculated.

```
delta = 0.001
FPRs, TPRs, _ = get_roc_curve(y_test, y_proba, delta)

# Plot the ROC curve
plt.plot(FPRs, TPRs, color='red',
         lw=2, label='ROC curve')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',
label="Mean")
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f'ROC curve (threshold delta = {delta})')
plt.legend(loc="lower right")
plt.show()
```



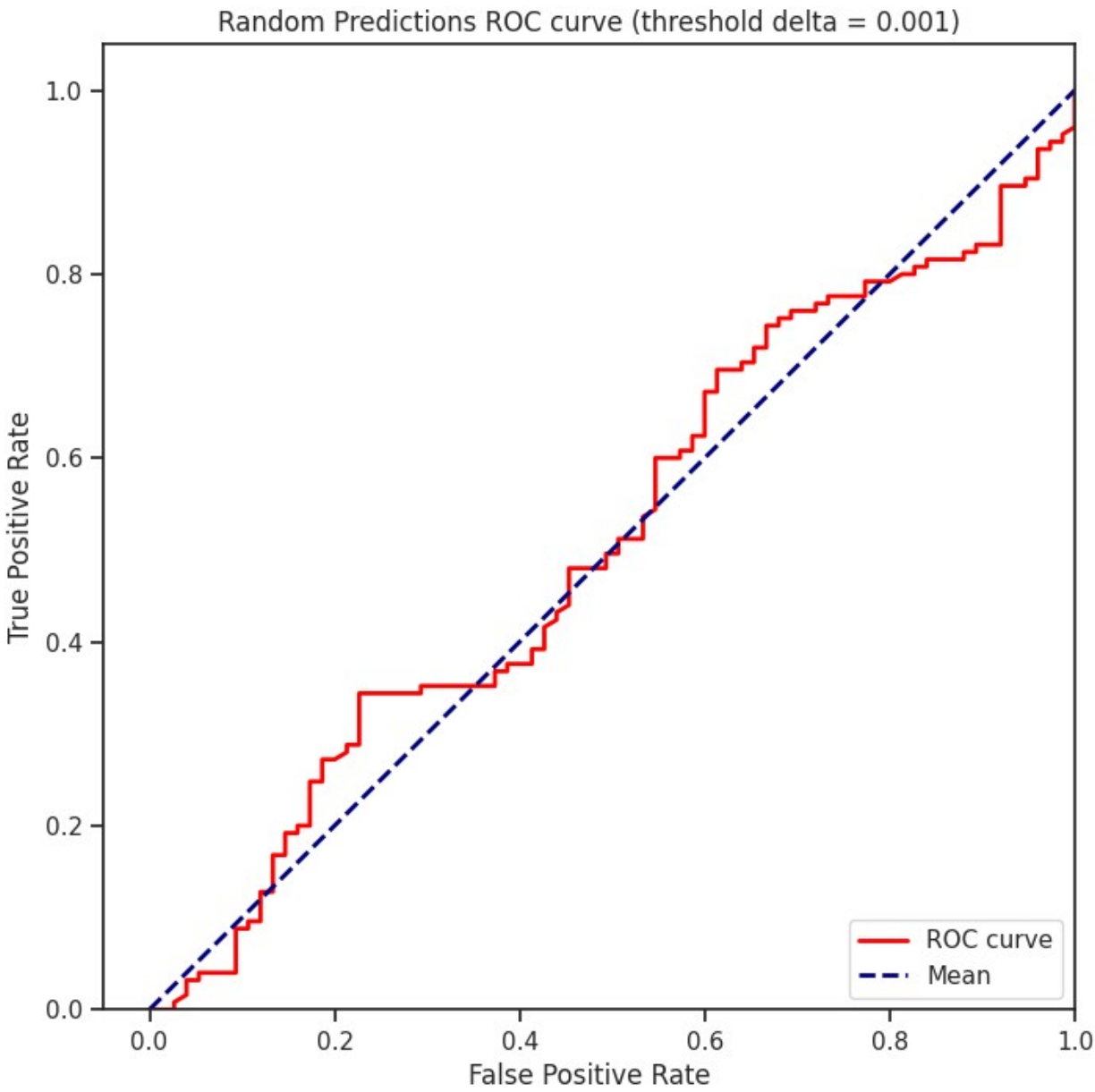
The peak towards left-most corner means near perfect classifier, while random prediction will have the curve as the blue dotted straight line. This ROC curve tells us that our model is nearly perfect classifier, with high accuracy :)

Let's calculate an ROC curve with random predictions, and plot it to see the difference.

```
# create random predictions
rand_proba = np.random.random(size=(y_proba.shape))
rand_proba[:5]  # 0.5 probability of being 0 or 1
```

```
array([0.79654299, 0.18343479, 0.779691  , 0.59685016, 0.44583275])
```

```
FPRs, TPRs, _ = get_roc_curve(y_test, rand_proba, delta)  # passing
random preds

# Plot the ROC curve
plt.plot(FPRs, TPRs, color='red',
         lw=2, label='ROC curve')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',
label="Mean")
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f'Random Predictions ROC curve (threshold delta = {delta})')
plt.legend(loc="lower right")
plt.show()
```

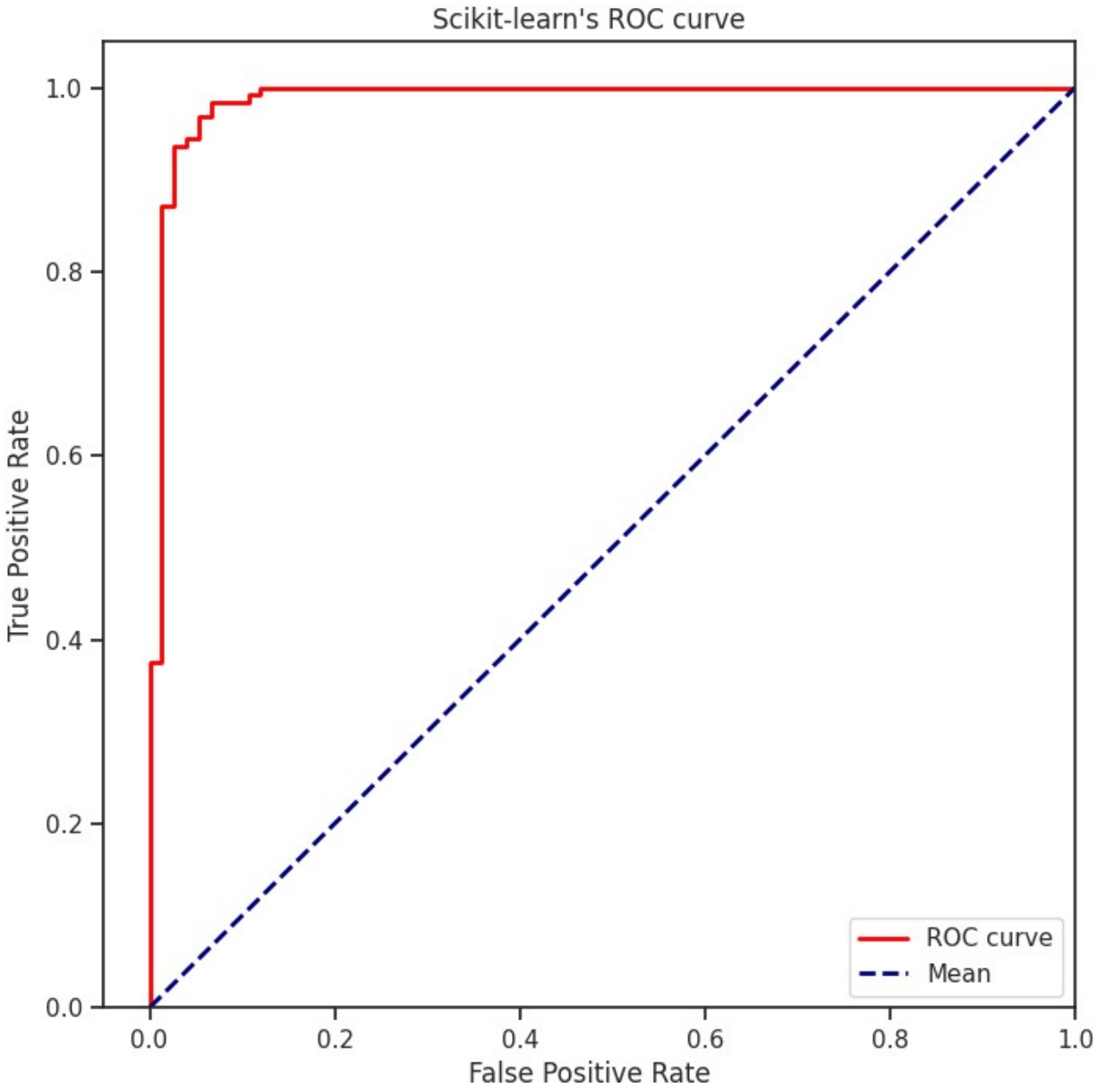


As we can see from the above plot, random predictions give the ROC curve nearly at the mean. Contrast it with the previous curve, how our model has top left peak, due to its superior performance.

Now, let's see how these curve's are similiar to the ROC curve with scikit's implementation.

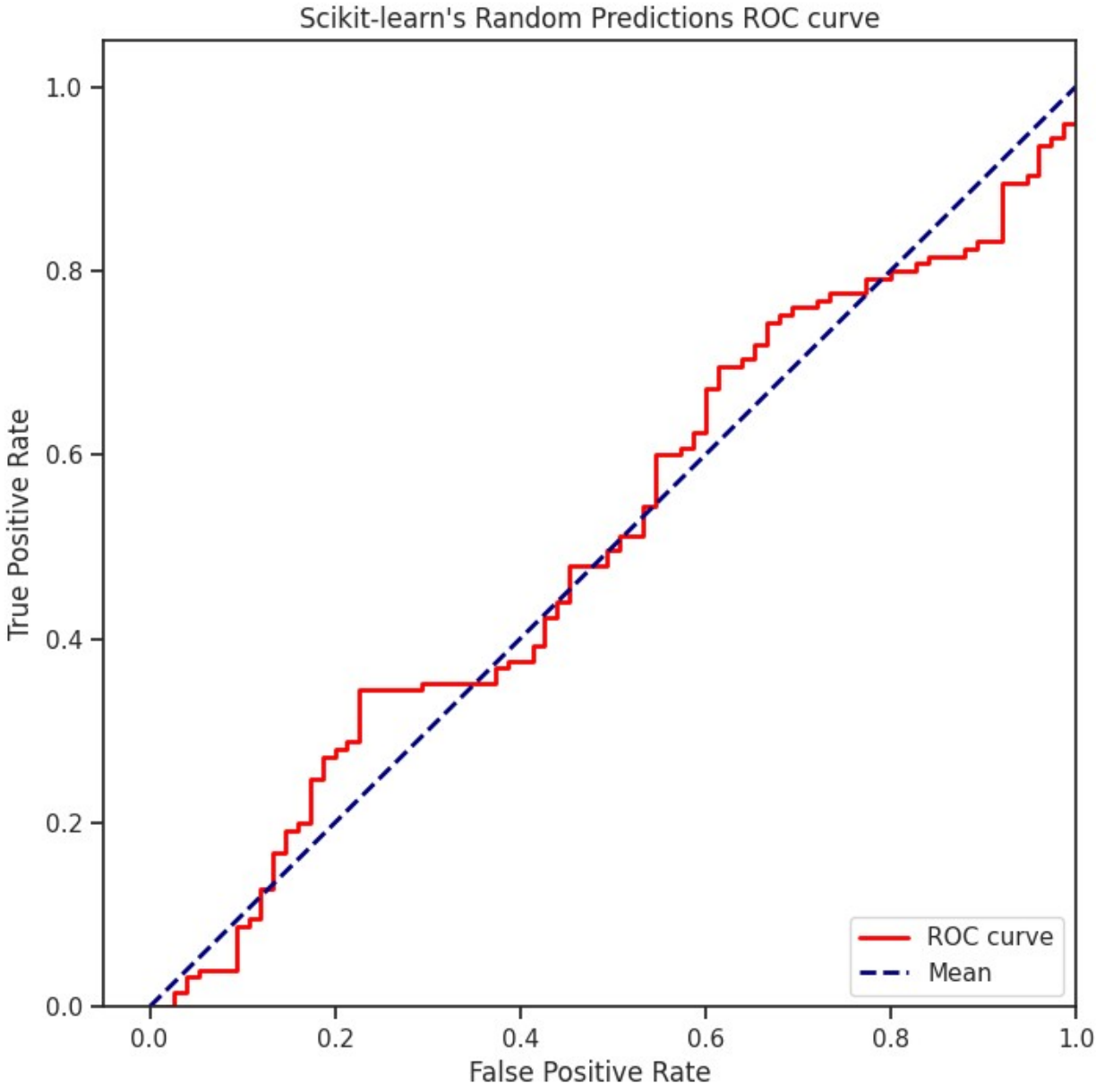
```
FPRs, TPRs, _ = metrics.roc_curve(y_test, y_proba)

# Plot the ROC curve
plt.plot(FPRs, TPRs, color='red',
         lw=2, label='ROC curve')
```

```
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',
label="Mean")
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title("Scikit-learn's ROC curve")
plt.legend(loc="lower right")
plt.show()
```




```
FPRs, TPRs, _ = metrics.roc_curve(y_test, rand_proba)  # passing
random preds
```

```
# Plot the ROC curve
plt.plot(FPRs, TPRs, color='red',
         lw=2, label='ROC curve')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',
label="Mean")
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title("Scikit-learn's Random Predictions ROC curve")
plt.legend(loc="lower right")
plt.show()
```



ROC-AUC score

Now, that we know about ROC curve, what it represents, **AUC score** is very easy to understand. AUC stands for **Area under the Curve**, which is nothing but the area under the ROC curve formed by the predictions. As we saw, a totally random prediction will have AUC score `0.5`, while a perfect classifier will have AUC score of `1`.

Lets check the AUC score of our model.

```
auc_score = metrics.roc_auc_score(y_test, y_proba)
print(f"Scikit's ROC-AUC score of SVC model is {auc_score: .4f}")
```

```
Scikit's ROC-AUC score of SVC model is  0.9872
```

We can also calculate the **ROC-AUC score** by summing up the areas under each observation of FPRs and TPRs.

```
def get_roc_auc_score(y_test, y_proba):

    # use the function get_roc_curve that we created.
    FPRs, TPRs, _ = get_roc_curve(y_test, y_proba)
    FPRs.reverse()
    TPRs.reverse()
    x1, y1 = FPRs[0], TPRs[0]
    auc = 0.0
    prev = 0.0
    # cumulative differences in x-axis
    diffs = [FPRs[i] - FPRs[i-1] for i in range(1, len(FPRs))]
    for x, y in zip(diffs, TPRs[1:]):
        auc += (x * y1)  # area of rectangle
        auc += (x * (y - y1)/2)  # area of triangle formed (if any)
        y1 = y
    return auc

auc_score = get_roc_auc_score(y_test, y_proba)
print(f"Our ROC-AUC score of SVC model is {auc_score: .4f}")
```

```
Our ROC-AUC score of SVC model is  0.9874
```

This is a good ROC-AUC score as we expected. (Also pretty close to Scikit's implementation). Lets try the ROC-AUC score of random predictions.

```
auc_score = metrics.roc_auc_score(y_test, y_proba)
print(f"Scikit's ROC-AUC score of random predictions is:
{auc_score: .4f}")
auc_score = get_roc_auc_score(y_test, rand_proba)
print(f"Our ROC-AUC score of random predictions is:
{auc_score: .4f}")
```

```
Scikit's ROC-AUC score of random predictions is:  0.9872
Our ROC-AUC score of random predictions is:       0.5080
```

As expected, it is close to `0.5`. (The minor differences are due to different granularity of thresholds chosen by us, and in Scikit's implementation.)