ЛАБОРАТОРНА РОБОТА № 1

ПОПЕРЕДНЯ ОБРОБКА ТА КОНТРОЛЬОВАНА КЛАСИФІКАЦІЯ ДАНИХ

***Мета*:** використовуючи спеціалізовані бібліотеки та мову програмування Python дослідити попередню обробку та класифікацію даних**.**

Хід роботи:

Нормалізація даних:

```
# Нормалізація даних
data_normalized_l1 = preprocessing.normalize(input_data, norm='l1')
data_normalized_l2 = preprocessing.normalize(input_data, norm='l2')
print("\nl1 normalized data:\n", data_normalized_l1)
print("\nl2 normalized data:\n", data_normalized_l2)
```

```
l1 normalized data:
 [[ 0.45132743 -0.25663717  0.2920354 ]
 [-0.0794702   0.51655629 -0.40397351]
 [ 0.609375    0.0625      0.328125  ]
 [ 0.33640553 -0.4562212  -0.20737327]]

l2 normalized data:
 [[ 0.75765788 -0.43082507  0.49024922]
 [-0.12030718  0.78199664 -0.61156148]
 [ 0.87690281  0.08993875  0.47217844]
 [ 0.55734935 -0.75585734 -0.34357152]]
```

Рис.1.Результат нормалізації даних.

L1-нормалізація та L2-нормалізація - це два різних методи нормалізації даних, які використовуються для перетворення вхідних даних так, щоб вони мали одиничну норму (довжину) вектора.

L1-нормалізація (також відома як " манхеттенська нормалізація"):

Обчислює суму абсолютних значень кожного рядка в матриці даних, потім кожне значення в рядку ділиться на цю суму.

L2-нормалізація (також відома як " Евклідова нормалізація"):

Обчислює Евклідову норму (квадратний корінь з суми квадратів значень)

					ДУ «Житомирська політехніка».22.122.4.000 – Лр1		
Змн.	Арк.	№ докум.	Підпис	Дата			
Розроб.		Дяченко В.В.			Звіт з лабораторної роботи	Літ.	Арк. 1 / Аркушів
Перевір.		Голенко М.Ю.					
Керівник						ФІКТ Гр. КН-20-1(1)	
Н. контр.							
Зав. каф.							

кожного рядка в матриці даних, потім кожне значення в рядку ділиться на цю норму. Через що ми отримуємо різний результат при однакових вхідних даних. В нашому випадку абсолютні значення при L2-нормалізації – більші.

Завдання 1

```
[ ] import numpy as np
    from sklearn import preprocessing

[ ] # Надання позначок вхідних даних
    Input_labels = ['red', 'black', 'red', 'green', 'black', 'yellow', 'white']

[ ] # Створення кодувальника та встановлення відповідності
    # між мітками та числами
    encoder = preprocessing.LabelEncoder()
    encoder.fit(Input_labels)

    ▾ LabelEncoder
    LabelEncoder()
```

Рис.2. Результат створення кодувальника.

```
# Виведення відображення
print("\nLabel mapping:")
for i, item in enumerate(encoder.classes_):
  print(item, '-->', i)
```

```
Label mapping:
black --> 0
green --> 1
red --> 2
white --> 3
yellow --> 4
black --> 5
```

```
[ ] # перетворення міток за допомогою кодувальника
    test_labels = ['green', 'red', 'black']
    encoded_values = encoder.transform(test_labels)
    print("\nLabels =", test_labels )
    print("Encoded values =", list (encoded_values ) )
```

```
Labels = ['green', 'red', 'black']
Encoded values = [1, 2, 0]
```

Рис.3. Результат відображення та перетворення міток.

```
# Декодування набору чисел за допомогою декодера
encoded_values = [3, 0, 4, 1]
decoded_list = encoder.inverse_transform(encoded_values)
print("\nEncoded values =", encoded_values)
print("Decoded labels =", list (decoded_list ) )
```

```
Encoded values = [3, 0, 4, 1]
Decoded labels = ['white', 'black', 'yellow', 'green']
```

Рис.4. Результат декодування.

Завдання 2

```
[1] import numpy as np
    from sklearn import preprocessing
```

```
[8] #task 1
    input_data = np.array([[-5.3,-8.9,3.0],
                           [2.9, 5.1,  -3.3],
                           [3.1, -2.8, -3.2],
                           [2.2, -1.4, 5.1]])
```

```
# Бінаризація даних
data_binarized = preprocessing.Binarizer(threshold=3.0).transform(input_data)
print("\n Binarized data:\n", data_binarized)
```

```
Binarized data:
[[0. 0. 0.]
[0. 1. 0.]
[1. 0. 0.]
[0. 0. 1.]]
```

Рис.5. Результат зміни початкових даних та їх бінаризації.

```
[10] # Виведення середнього значення та стандартного відхилення
     print("\nBEFORE: ")
     print("Mean =", input_data.mean(axis=0))
     print("Std deviation =", input_data.std(axis=0))

     BEFORE:
     Mean = [ 0.725 -2.     0.4  ]
     Std deviation = [3.49454933 4.97543968 3.72491611]

     #Виключення середнього
     data_scaled = preprocessing.scale(input_data)
     print("\nAFTER: ")
     print("Mean =", data_scaled.mean(axis=0))
     print("Std deviation =", data_scaled.std(axis=0))

     AFTER:
     Mean = [-2.77555756e-17 -2.42861287e-17  0.00000000e+00]
     Std deviation = [1. 1. 1.]
```

Рис.6. Результат виключення середнього.

```
# Масштабування MinMax
data_scaler_minmax = preprocessing.MinMaxScaler(feature_range=(0, 1))
data_scaled_minmax = data_scaler_minmax.fit_transform(input_data)
print("\nMin max scaled data:\n", data_scaled_minmax)

Min max scaled data:
 [[0.         0.         0.75      ]
 [0.97619048 1.         0.        ]
 [1.         0.43571429 0.01190476]
 [0.89285714 0.53571429 1.        ]]

# Нормалізація даних
data_normalized_l1 = preprocessing.normalize(input_data, norm='l1')
data_normalized_l2 = preprocessing.normalize(input_data, norm='l2')
print("\nl1 normalized data:\n", data_normalized_l1)
print("\nl2 normalized data:\n", data_normalized_l2)

l1 normalized data:
 [[-0.30813953 -0.51744186  0.1744186 ]
 [ 0.25663717  0.45132743 -0.2920354 ]
 [ 0.34065934 -0.30769231 -0.35164835]
 [ 0.25287356 -0.16091954  0.5862069 ]]

l2 normalized data:
 [[-0.49145755 -0.82527777  0.27818352]
 [ 0.43082507  0.75765788 -0.49024922]
 [ 0.58911518 -0.53210404 -0.6081189 ]
 [ 0.38407812 -0.24441335  0.89036291]]
```

Рис.7. Результат масштабування та нормалізації даних.

Завдання 3

```
[5]  import numpy as np
     from sklearn import linear_model
     import matplotlib.pyplot as plt
     from utilities import visualize_classifier

[6]  # Визначення зразка вхідних даних
     X = np.array([[3.1, 7.2], [4, 6.7], [2.9, 8], [5.1, 4.5],
                   [6, 5], [5.6, 5], [3.3, 0.4],
                   [3.9, 0.9], [2.8, 1],
                   [0.5, 3.4], [1, 4], [0.6, 4.9]])
     y = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3])

[7]  # Створення логістичного класифікатора
     classifier = linear_model.LogisticRegression(solver='liblinear',C=1)

[8]  # Тренування класифікатора
     classifier.fit(X, y)

     LogisticRegression
     LogisticRegression(C=1, solver='liblinear')
```

Рис.8. Результат створення та тренування логістичного класифікатора.

```
visualize_classifier(classifier, X, y)
```

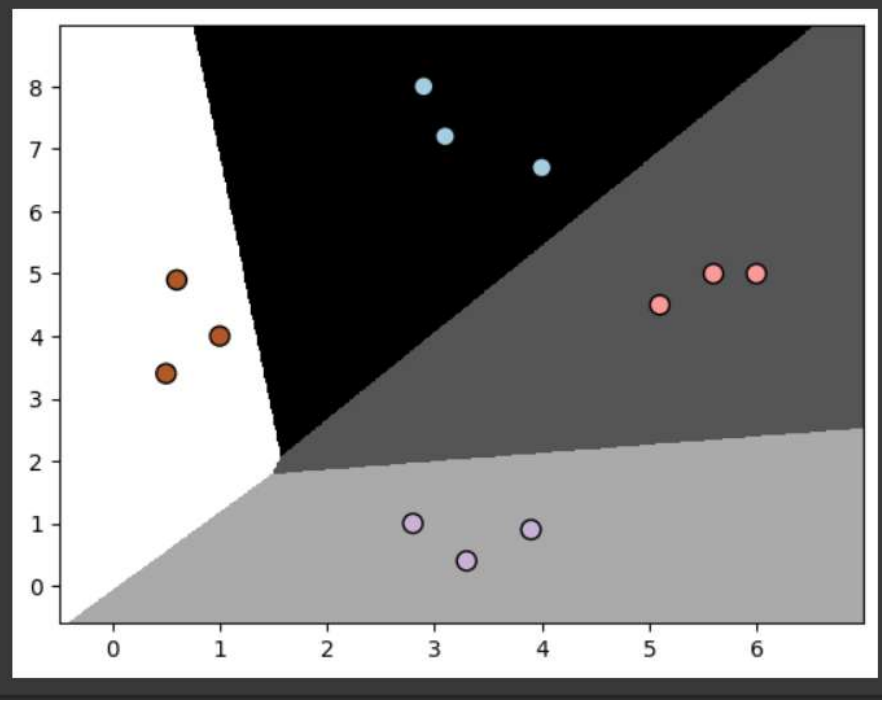


Рис.9. Візуалізація результата роботи класифікатора.

Завдання 4

```
[1] import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.naive_bayes import GaussianNB
    from sklearn.model_selection import train_test_split

    from utilities import visualize_classifier

    # Вхідний файл, який містить дані
    input_file = 'data_multivar_nb.txt'

[3] # Завантаження даних із вхідного файлу
    data = np.loadtxt(input_file, delimiter=',')
    X, y = data[:, :-1], data[:, -1]

[4] # Створення наївного байєсовського класифікатора
    classifier = GaussianNB()

[5] # Тренування класифікатора
    classifier.fit(X, y)

    ▾ GaussianNB
    GaussianNB()
```

Рис.10. Створення та тренування класифікатора.

```
# Прогнозування значень для тренувальних даних
y_pred = classifier.predict(X)

# Обчислення якості класифікатора
accuracy = 100.0 * (y == y_pred).sum() / X.shape[0]
print("Accuracy of Naive Bayes classifier =", round(accuracy, 2), "%")

Accuracy of Naive Bayes classifier = 99.75 %

# Візуалізація результатів роботи класифікатора
visualize_classifier(classifier, X, y)
```

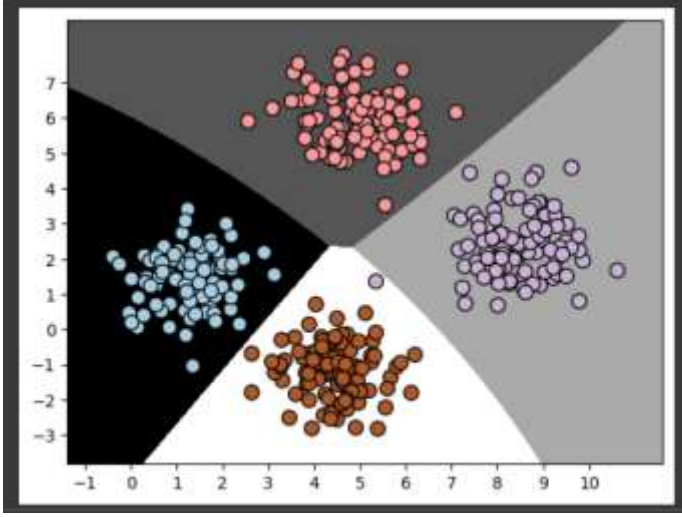


Рис.11. Прогнозування значень, обчислення якості та візуалізація результатів роботи створеного класифікатора.

```
# Розбивка даних на навчальний та тестовий набори
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=3)
classifier_new = GaussianNB()
classifier_new.fit(X_train, y_train)
y_test_pred = classifier_new.predict(X_test)

# Обчислення якості класифікатора
accuracy = 100.0 * (y_test == y_test_pred).sum() / X_test.shape[0]
print("Accuracy of the new classifier =", round(accuracy, 2), "%")

# Візуалізація роботи класифікатора
visualize_classifier(classifier_new, X_test, y_test)
```

```
Accuracy of the new classifier = 100.0 %
```

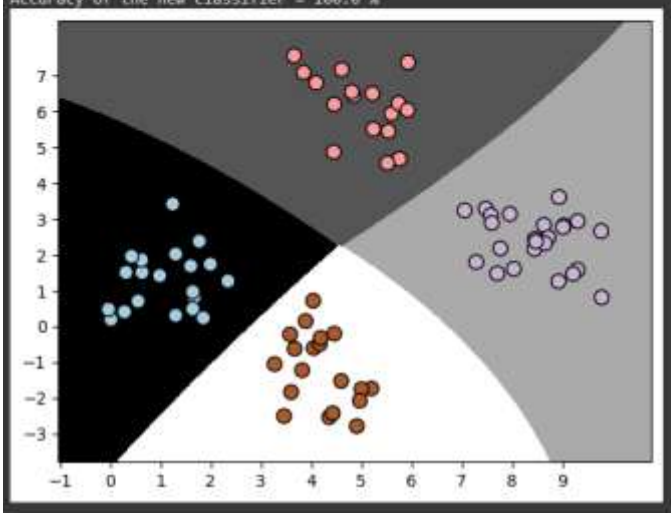
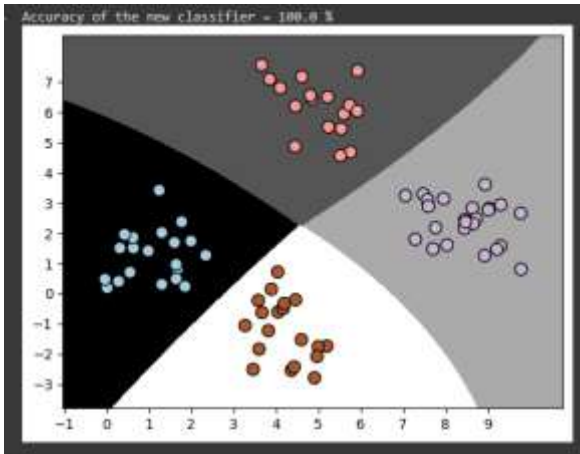


Рис.12. Результат розбивки набору даних, обчислення якості та візуалізація результатів роботи створеного класифікатора.

```
Accuracy of the new classifier = 100.0 %
```



```
num_folds = 3
accuracy_values = cross_val_score(classifier, X, y, scoring='accuracy', cv=num_folds)
print("Accuracy: " + str(round(100 * accuracy_values.mean(), 2)) + "%")
precision_values = cross_val_score(classifier, X, y, scoring='precision_weighted', cv=num_folds)
print("Precision: " + str(round(100 * precision_values.mean(), 2)) + "%")
recall_values = cross_val_score(classifier, X, y, scoring='recall_weighted', cv=num_folds)
print("Recall: " + str(round(100 * recall_values.mean(), 2)) + "%")
f1_values = cross_val_score(classifier, X, y, scoring='f1_weighted', cv=num_folds)
print("F1: " + str(round(100 * f1_values.mean(), 2)) + "%")
```

```
Accuracy: 99.75%
Precision: 99.76%
Recall: 99.75%
F1: 99.75%
```

Рис.13. Результат роботи класифікатора при повторному прогоні.

При повторному прогоні коду ми отримуємо той самий результат. Так відбувається через використання випадкового розподілу даних на навчальний та тестовий набори за допомогою функції train_test_split з фіксованим значенням random_state (3). Цей параметр встановлює початковий стан генератора випадкових чисел, і якщо він залишається незмінним між кількома запусками, то поділ даних буде однаковим, відповідно модель буде навчатися та тестуватися на одних і тих самих наборах даних.

Завдання 5

```
import pandas as pd
df = pd.read_csv('data_metrics.csv')
df.head()
```

	actual_label	model_RF	model_LR
0	1	0.639816	0.531904
1	0	0.490993	0.414496
2	1	0.623815	0.569883
3	1	0.506616	0.443674
4	0	0.418302	0.369532

```
[5] thresh = 0.5
df['predicted_RF'] = (df.model_RF >= 0.5).astype('int')
df['predicted_LR'] = (df.model_LR >= 0.5).astype('int')
df.head()
```

	actual_label	model_RF	model_LR	predicted_RF	predicted_LR
0	1	0.639816	0.531904	1	1
1	0	0.490993	0.414496	0	0
2	1	0.623815	0.569883	1	1
3	1	0.506616	0.443674	1	0

Рис.14. Завантаження набору даних та визначення граничного значення.

```
[7] def find_TP(y_true, y_pred):
        # Підраховує кількість true positives (y_true = 1, y_pred = 1)
        return sum((y_true == 1) & (y_pred == 1))

    def find_FN(y_true, y_pred):
        # Підраховує кількість false negatives (y_true = 1, y_pred = 0)
        return sum((y_true == 1) & (y_pred == 0))

    def find_FP(y_true, y_pred):
        # Підраховує кількість false positives (y_true = 0, y_pred = 1)
        return sum((y_true == 0) & (y_pred == 1))

    def find_TN(y_true, y_pred):
        # Підраховує кількість true negatives (y_true = 0, y_pred = 0)
        return sum((y_true == 0) & (y_pred == 0))
```

```
print('TP:',find_TP(df.actual_label.values,   df.predicted_RF.values))
print('FN:',find_FN(df.actual_label.values, df.predicted_RF.values))
print('FP:',find_FP(df.actual_label.values, df.predicted_RF.values))
print('TN:',find_TN(df.actual_label.values, df.predicted_RF.values))
```

```
TP: 5047
FN: 2832
FP: 2360
TN: 5519
```

Рис.15. Результат визначення власних функцій для перевірки confusion_matrix.

```
import numpy as np
def find_conf_matrix_values(y_true,y_pred):
    # calculate TP, FN, FP, TN
    TP = find_TP(y_true,y_pred)
    FN = find_FN(y_true,y_pred)
    FP = find_FP(y_true,y_pred)
    TN = find_TN(y_true,y_pred)
    return TP,FN,FP,TN
def diachenko_confusion_matrix(y_true, y_pred):
    TP,FN,FP,TN = find_conf_matrix_values(y_true,y_pred)
    return np.array([[TN,FP],[FN,TP]])
```

```
diachenko_confusion_matrix(df.actual_label.values, df.predicted_RF.values)
```

```
array([[5519, 2360],
       [2832, 5047]])
```

Рис.16. Результат написання функції diachenko_confusion_matrix та перевірка роботи створеної функції.

```
from sklearn.metrics import accuracy_score
accuracy_score(df.actual_label.values, df.predicted_RF.values)
```

```
0.6705165630156111
```

```
def diachenko_accuracy_score(y_true, y_pred):
    TN, FP, FN, TP = confusion_matrix(y_true, y_pred).ravel()
    accuracy = (TP + TN) / (TP + TN + FP + FN)

    return accuracy
assert diachenko_accuracy_score(df.actual_label.values, df.predicted_LR.values) == accuracy_score(df.actual
print('Accuracy RF: %.3f'%(diachenko_accuracy_score(df.actual_label.values, df.predicted_RF.values)))
print('Accuracy LR: %.3f'%(diachenko_accuracy_score(df.actual_label.values, df.predicted_LR.values)))
```

```
Accuracy RF: 0.671
Accuracy LR: 0.616
```

Рис.17. Результат визначення власної ф-ції accurancy_score.

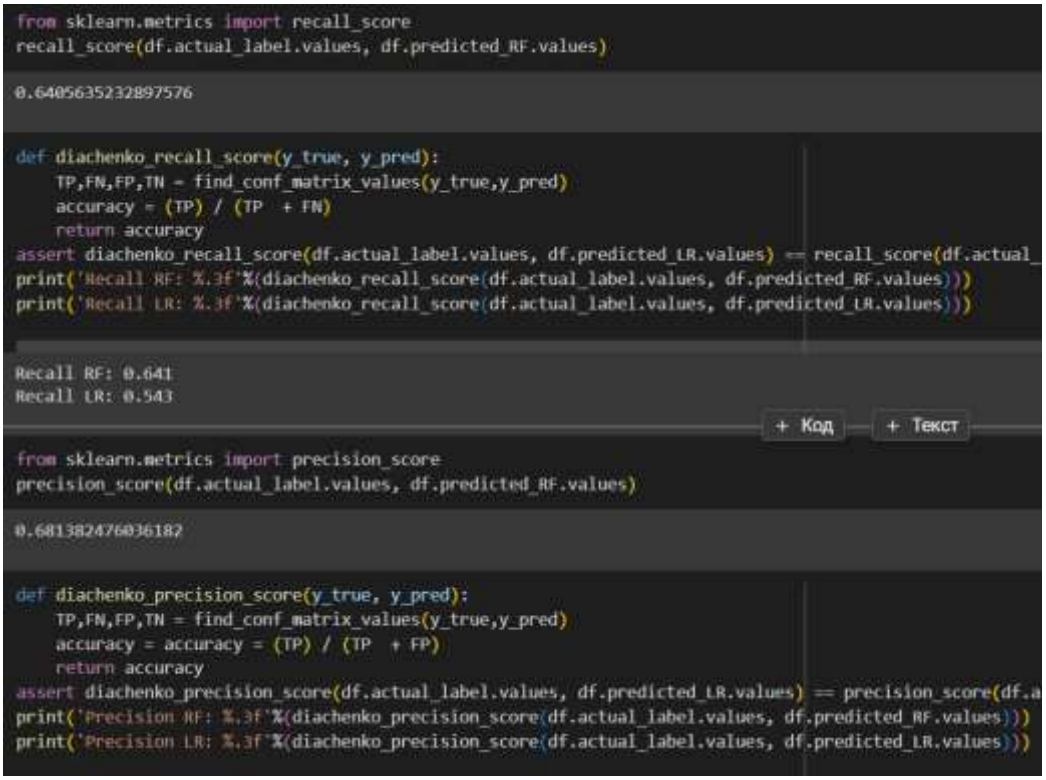


Рис.18. Результат визначення власної ф-ції recall_score та precision_score.

Рис.19. Результат визначення власної ф-ції f1_score.

```
print('scores with threshold = 0.5')
print('Accuracy RF: %.3f'%(diachenko_accuracy_score(df.actual_label.values, df.predicted_RF.values)))
print('Recall RF: %.3f'%(diachenko_recall_score(df.actual_label.values, df.predicted_RF.values)))
print('Precision RF: %.3f'%(diachenko_precision_score(df.actual_label.values, df.predicted_RF.values)))
print('F1 RF: %.3f'%(diachenko_f1_score(df.actual_label.values, df.predicted_RF.values)))
print('')
print('scores with threshold = 0.25')
print('Accuracy RF: %.3f'%(diachenko_accuracy_score(df.actual_label.values, (df.model_RF >= 0.25).astype('int').values)))
print('Recall RF: %.3f'%(diachenko_recall_score(df.actual_label.values, (df.model_RF >= 0.25).astype('int').values)))
print('Precision RF: %.3f'%(diachenko_precision_score(df.actual_label.values, (df.model_RF >= 0.25).astype('int').values)))
print('F1 RF: %.3f'%(diachenko_f1_score(df.actual_label.values, (df.model_RF >= 0.25).astype('int').values)))
```

```
scores with threshold = 0.5
Accuracy RF: 0.671
Recall RF: 0.641
Precision RF: 0.681
F1 RF: 0.660

scores with threshold = 0.25
Accuracy RF: 0.502
Recall RF: 1.000
Precision RF: 0.501
F1 RF: 0.668
```

Рис.20. Результати при різних порогах.

Зниження порогу до 0.25 призвело до збільшення полності (Recall), але зменшило точність (Precision) з 0.681 до 0.501. Це означає, що модель розпізнала всі позитивні приклади, але також зробила багато помилкових позитивних класифікацій. Поріг 0.5 в свою чергу має вищу точність та дає більш збалансований F1-показник.

```
plt.plot(fpr_RF, tpr_RF,'r-',label = 'RF')
plt.plot(fpr_LR,tpr_LR,'b-', label= 'LR')
plt.plot([0,1],[0,1],'k-',label='random')
plt.plot([0,0,1,1],[0,1,1,1],'g-',label='perfect')
plt.legend()
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

```
Text(0, 0.5, 'True Positive Rate')
```

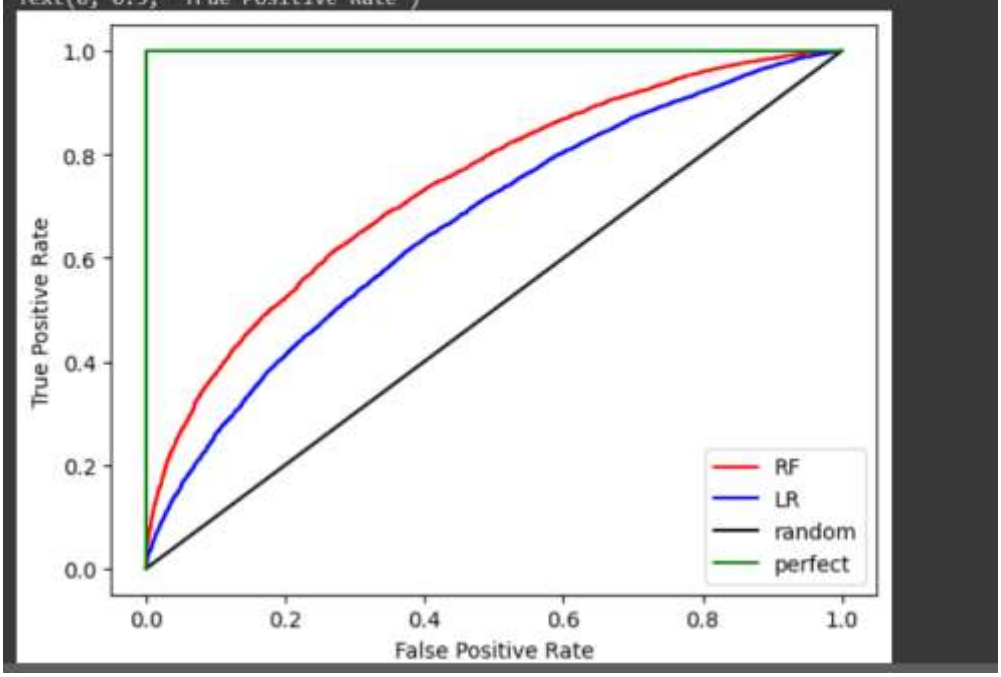


Рис.21. Результат побудови ROC кривої для кожної моделі.

```python
from sklearn.metrics import roc_auc_score
auc_RF = roc_auc_score(df.actual_label.values, df.model_RF.values)
auc_LR = roc_auc_score(df.actual_label.values, df.model_LR.values)
print('AUC RF:%.3f'% auc_RF)
print('AUC LR:%.3f'% auc_LR)
```

```
AUC RF:0.738
AUC LR:0.666
```

Рис.22. Метрика площі під кривою для аналізу продуктивності.

```python
import matplotlib.pyplot as plt
plt.plot(fpr_RF, tpr_RF,'r-',label = 'RF AUC: %.3f'%auc_RF)
plt.plot(fpr_LR,tpr_LR,'b-', label= 'LR AUC: %.3f'%auc_LR)
plt.plot([0,1],[0,1],'k-',label='random')
plt.plot([0,0,1,1],[0,1,1,1],'g-',label='perfect')
plt.legend()
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

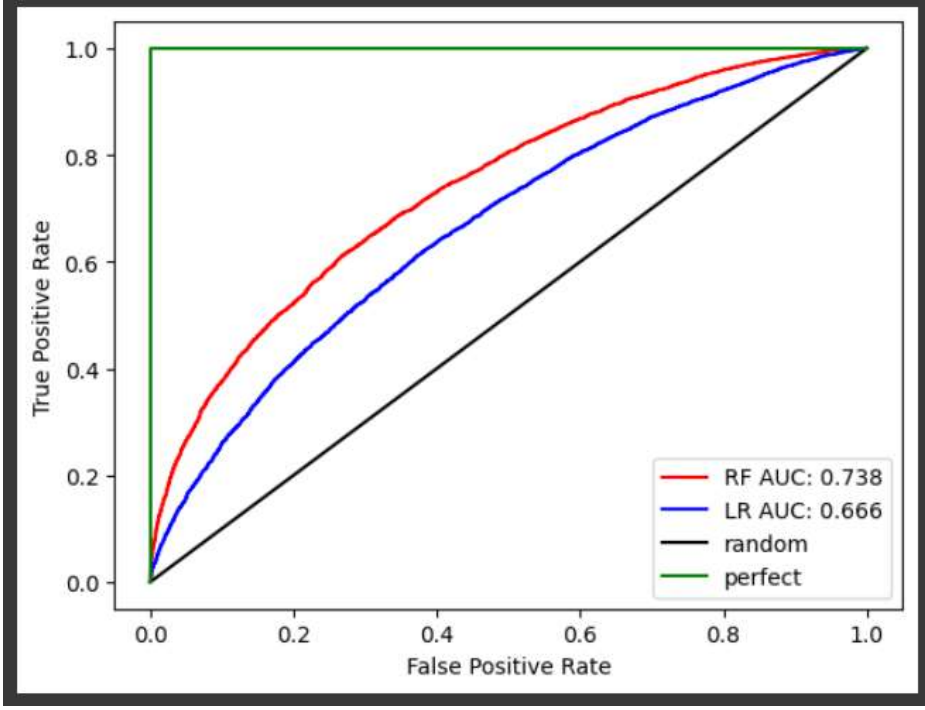


Рис.21. Результат побудови ROC кривої для кожної моделі з AUC у легенді.

Зазвичай, більше значення AUC вказує на кращу якість класифікатора. Таким чином, у нашому випадку модель випадковий ліс (RF) має кращу продуктивність, оскільки значення AUC більше. Також при порозі 0.5, модель RF має кращу точність, полнотніть та F1-показник порівняно з моделлю логістичної регресії (LR).

Завдання 6

```python
# Розбиття на навчальний та тестовий набори
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=3)

# Ініціалізуємо модель SVM
svm_model = SVC(kernel='linear')  # Використовуємо лінійне ядро для спрощення прикладу

# Тренуємо модель на навчальних даних
svm_model.fit(X_train, y_train)

# Проводимо класифікацію на тестових даних
y_pred = svm_model.predict(X_test)

classification_rep = classification_report(y_test, y_pred)
# Обчислення якості класифікатора
accuracy = 100.0 * (y_test == y_pred).sum() / X_test.shape[0]
print("Accuracy of the new classifier =", round(accuracy, 2), "%")
# Виводимо результати
print("Classification Report:\n", classification_rep)
# Візуалізація роботи класифікатора
visualize_classifier(classifier_new, X_test, y_test)
```

Рис.22. Класифікація даних в файлі за допомогою SVM.

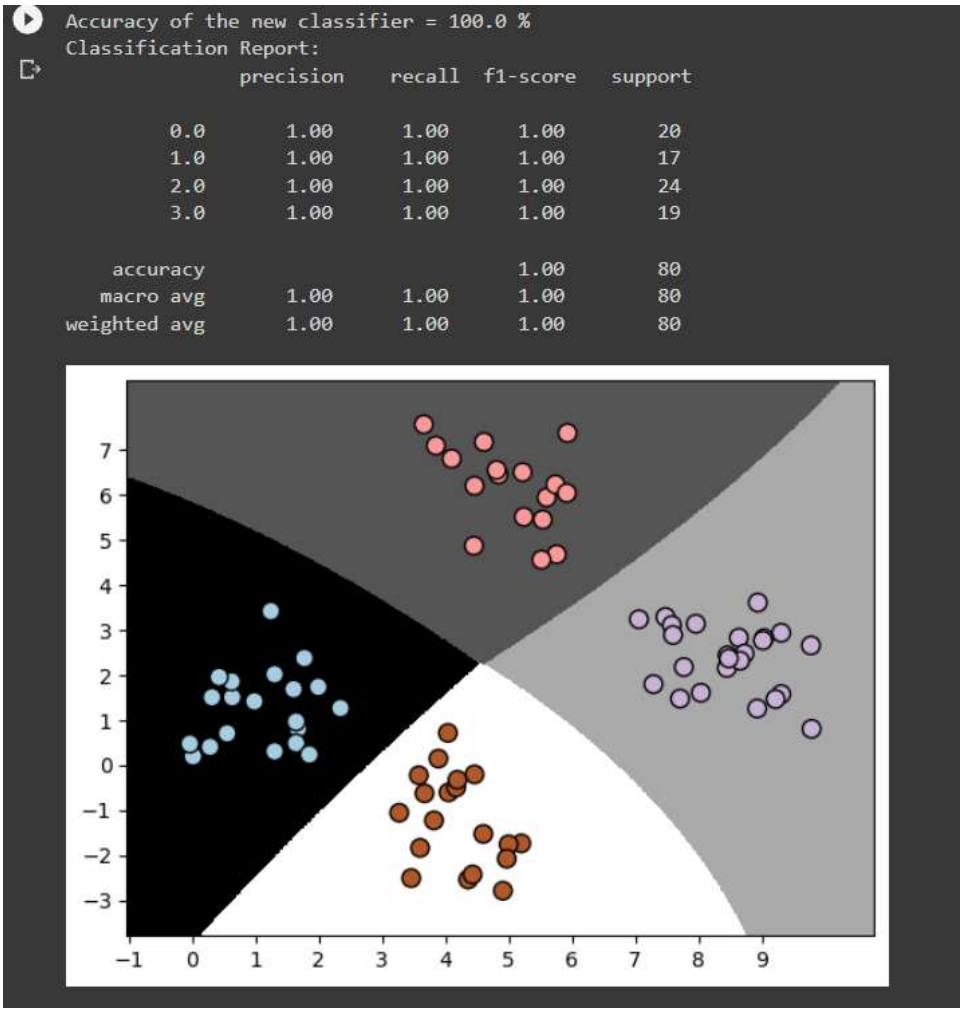


Рис.23. Результат роботи.

На основі наданих результатів, обидві моделі, баєсівський класифікатор і метод опорних векторів (SVM), досягли ідеальної точності (Accuracy = 100%) при класифікації того самого набору даних. Взагалі слід враховувати, що байєсівський класифікатор (Gaussian Naive Bayes) вважається простішим та менш обчислювально витратним методом, а метод опорних векторів (SVM) робить більш складні обчислення і може працювати краще, коли дані мають складну структуру.

У нашому ж випадку обидві моделі класифікації гарно виконують поставлену задачу за короткий проміжок часу і подальший вибір буде залежати від поставленої задачі.