ЛАБОРАТОРНА РОБОТА № 1

***Тема: «*ПОПЕРЕДНЯ ОБРОБКА ТА КОНТРОЛЬОВАНА КЛАСИФІКАЦІЯ ДАНИХ***»*

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

**Хід роботи**

Посилання на програмнй код на Github:

<https://github.com/dengaevsky/Labs_AI/tree/main/lab1>

***Завдання 2.1.* Попередня обробка даних**

Як правило, при обробці ми маємо справу з великими обсягами необроблених вихідних даних. Алгоритми машинного навчання розраховані на те, що, перш ніж вони зможуть розпочати процес тренування, отримані дані будуть відформатовані певним чином. Щоб привести дані до форми, що прийнятна для алгоритмів машинного навчання, ми повинні попередньо підготувати їх і перетворити на потрібний формат.

**Розглянемо декілька різних методів попередньої обробки даних**

**Бінарізація —** процес застосовується в тих випадках, коли ми хочемо перетворити наші числові значення на булеві значення (0, 1). Скористаємося вбудованим методом для бінаризації вхідних даних, встановивши значення 2.1 як порогове. Всі значення понад 2.1 примусово встановлюються рівними 1. Інші значення стають рівними 0.

**Виключення середнього** — методика попередньої обробки даних, що зазвичай використовується в машинному навчанні. Як правило, із векторів ознак (feature vectors) доцільно виключати середні значення, щоб кожна ознака (feature) центрувалася на нулі. Це робиться з метою, виключити з розгляду зміщення значень у векторах ознак.

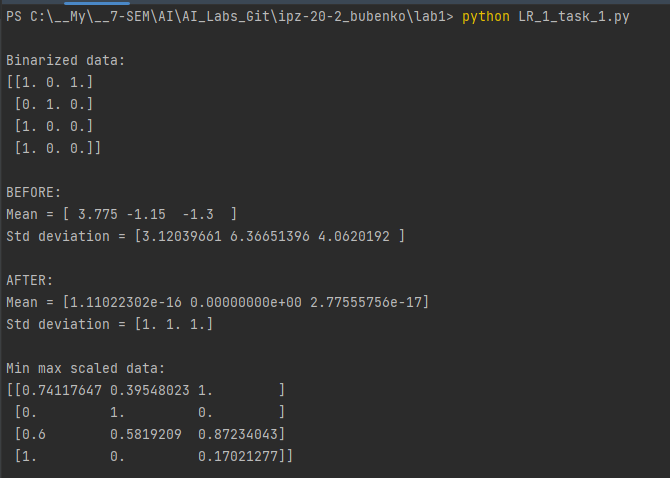
**Масштабування —** у нашому векторі ознак кожне значення може змінюватись у деяких випадкових межах. Тому дуже важливо масштабувати ознаки, щоб вони були рівним ігровим полем для тренування алгоритму машинного навчання. Ми не хочемо, щоб будь-яка з ознак могла набувати штучно великого або малого значення лише через природу вимірів. Кожен рядок відмасштабований таким чином, щоб максимальним значенням була б 1, а всі решта значень визначалися відносно неї.

**Нормалізація —** Процес нормалізації полягає у зміні значень у векторі ознак таким чином, щоб для їх вимірювання можна було використовувати одну загальну шкалу. У машинному навчанні використовують різні форми нормалізації. У найбільш поширених з них, значення змінюються так, щоб їх сума дорівнювала 1. **L1-нормалізація** використовує метод найменших абсолютних відхилень (Least Absolute Deviations), що забезпечує рівність 1 суми абсолютних значень в кожному ряду. **L2-нормалізація** використовує метод найменших квадратів, що забезпечує рівність 1 суми квадратів значень. Взагалі, техніка *L1-нормалізації* вважається більш надійною по порівняно з *L2-нормалізацією*, оскільки вона менш чутлива до викидів. Дуже часто дані містять викиди, і з цим нічого не вдієш. Ми хочемо використовувати безпечні методики, що дозволяють ігнорувати викиди у процесі обчислень. Якби ми вирішували завдання, в якому викиди грають важливу роль, то, ймовірно, найкращим вибором була б *L2-нормалізація*.

Лістинг програмного коду до завдань 2.1.1 – 2.1.4.:

import numpy as np  
from sklearn import preprocessing  
  
input\_data = np.array([[5.1, -2.9, 3.3],  
 [-1.2, 7.8, -6.1],  
 [3.9, 0.4, 2.1],  
 [7.3, -9.9, -4.5]])  
  
# Бінаризація даних  
data\_binarized = preprocessing.Binarizer(threshold=2.1).transform(input\_data)  
print(f"\nBinarized data:\n{data\_binarized}")  
  
# Виведення середнього значення та стандартного відхилення  
print("\nBEFORE: ")  
print(f"Mean = {input\_data.mean(axis=0)}")  
print(f"Std deviation = {input\_data.std(axis=0)}")  
  
# Виключення середнього  
data\_scaled = preprocessing.scale(input\_data)  
print("\nAFTER: ")  
print(f"Mean = {data\_scaled.mean(axis=0)}")  
print(f"Std deviation = {data\_scaled.std(axis=0)}")  
  
# Масштабування MinМax  
data\_scaled\_minmax = preprocessing.MinMaxScaler(feature\_range=(0, 1))  
data\_scaled\_minmax = data\_scaled\_minmax.fit\_transform(input\_data)  
print(f"\nMin max scaled data:\n{data\_scaled\_minmax}")  
  
# Нормалізація даних  
data\_normalized\_l1 = preprocessing.normalize(input\_data, norm='l1')  
data\_normalized\_l2 = preprocessing.normalize(input\_data, norm='l2')  
print(f"\nL1 normalized data:\n{data\_normalized\_l1}")  
print(f"\nL2 normalized data:\n{data\_normalized\_l2}")

Результат виконання програми:



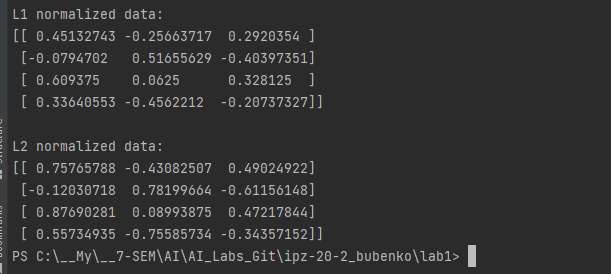


Рис. 1.1. – 1.2. Результат виконання пограми

***Висновок чим відрізняються L1-нормалізація від L2-нормалізацієї***

Головна відмінність полягає в тому, як вони враховують значення. L1-нормалізація використовує абсолютні значення, що дозволяє менш чутливо реагувати на великі відхилення. У той час, як L2-нормалізація використовує квадратичні значення, і це робить її більш чутливою до великих відхилень, оскільки вони впливають на суму квадратів значень значно сильніше.

Отже, L1-нормалізація менш чутлива до великих викидів, проте може втратити деяку інформацію про розподіл значень, тоді як L2-нормалізація більш точно враховує великі відхилення, але виявляє більшу чутливість до них.

**Кодування міток —** це процес перетворення категоріальних міток або текстових даних у числові значення. Це часто використовується в машинному навчанні, оскільки багато алгоритмів працюють тільки з числовими даними. Наприклад, якщо у вас є стовпець з категоріями "Червоний", "Синій" та "Зелений", то після кодування міток може з'явитися стовпець з числовими значеннями, наприклад "0", "1" та "2", де кожна цифра відповідає відповідній категорії. Це важливий крок у підготовці даних для багатьох моделей машинного навчання, оскільки дозволяє включити категоріальні дані в процес навчання.

Лістинг програмного коду до завдання 2.1.5:

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)  
  
# Виведення відображення  
print("\nLabel mapping: ")  
for i, item in enumerate(encoder.classes\_):  
 print(item, '-->', i)  
  
# Перетворення міток за допомогою кодувальника  
test\_labels = ['green', 'red', 'black']  
encoded\_values = encoder.transform(test\_labels)  
print(f"\nLabels = {test\_labels}")  
print(f"Encoded values = {list(encoded\_values)}")  
  
# Декодування набору чисел за допомогою декодера  
encoded\_values = [3, 0, 4, 1]  
decoded\_list = encoder.inverse\_transform(encoded\_values)  
print(f"\nEncoded values = {encoded\_values}")  
print(f"Decoded labels = {list(decoded\_list)}")

Результат виконання програми:

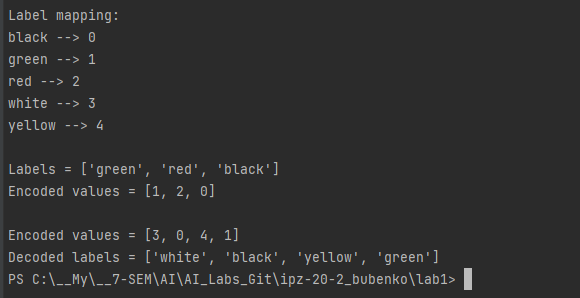


Рис. 2. Результат виконання програми

Отже, в результаті бачимо масиви: початковий масив числових значень та масив, де ці числа були декодовані назад у текстові мітки. Наприклад, "3" може бути декодовано назад у "green", і так далі. Як бачимо алгоритм працює коректно навіть при дублюванні міток (деякі значення зустрічаються більше 1 разу у початковому масиві).

***Завдання 2.2.* Попередня обробка нових даних**

У коді програми попереднього завдання поміняйте дані по рядках (значення змінної input\_data) на значення відповідно варіанту таблиці 1 та виконайте операції: Бінарізації, Виключення середнього, Масштабування, Нормалізації.

Варіант обирається відповідно номера за списком групи відповідно до таблиці 1.

Таблиця 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № варіанту | Значення змінної  input\_data | | | | | | | | | | | | Поріг бінаризації |
| 5. | -1.3 | 3.9 | 4.5 | -5.3 | -4.2 | -1.3 | 5.2 | -6.5 | -1.1 | -5.2 | 2.6 | -2.2 | 3.0 |

Лістнг програми:

import numpy as np  
from sklearn import preprocessing  
  
  
input\_data = np.array([[-1.3, 3.9, 4.5],  
 [-5.3, -4.2, -1.3],  
 [5.2, -6.5, -1.1],  
 [-5.2, 2.6, -2.2]])  
  
troubleshold = 3.0  
  
# Бінаризація даних  
data\_binarized = preprocessing.Binarizer(threshold=troubleshold).transform(input\_data)  
print(f"\nBinarized data:\n{data\_binarized}")  
  
# Виведення середнього значення та стандартного відхилення  
print("\nBEFORE: ")  
print(f"Mean = {input\_data.mean(axis=0)}")  
print(f"Std deviation = {input\_data.std(axis=0)}")  
  
# Виключення середнього  
data\_scaled = preprocessing.scale(input\_data)  
print("\nAFTER: ")  
print(f"Mean = {data\_scaled.mean(axis=0)}")  
print(f"Std deviation = {data\_scaled.std(axis=0)}")  
  
# Масштабування MinМax  
data\_scaled\_minmax = preprocessing.MinMaxScaler(feature\_range=(0, 1))  
data\_scaled\_minmax = data\_scaled\_minmax.fit\_transform(input\_data)  
print(f"\nMin max scaled data:\n{data\_scaled\_minmax}")  
  
# Нормалізація даних  
data\_normalized\_l1 = preprocessing.normalize(input\_data, norm='l1')  
data\_normalized\_l2 = preprocessing.normalize(input\_data, norm='l2')  
print(f"\nL1 normalized data:\n{data\_normalized\_l1}")  
print(f"\nL2 normalized data:\n{data\_normalized\_l2}")

Результат виконання програми:

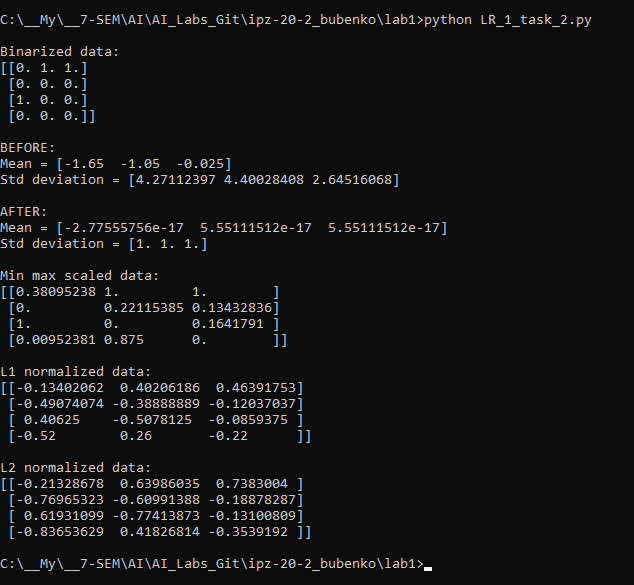


Рис. 3. Результат виконання програми

***Завдання 2.3.*** **Класифікація логістичною регресією або логістичний класифікатор**

**Логістична регресія (logistic regression)** - це методика, що використовується для пояснення відносин між вхідними та вихідними змінними. Вхідні змінні вважаються незалежними, вихідні – залежними. Залежна змінна може мати лише фіксований набір значень. Ці значення відповідають класам завдання класифікації. Метою є ідентифікація відносин між незалежними та залежними змінними за допомогою оцінки ймовірностей того, що та або інша залежна змінна відноситься до того чи іншого класу. За своєю природою логістична функція є сигмоїдою, що використовується для створення функцій з різними параметрами. Вона тісно пов’язана з аналізом даних на основі узагальненої лінійної моделі, у відповідності до якої робиться спроба підігнати пряму лінію до групи точок таким чином, щоб мінімізувати помилку. Замість лінійної регресії ми застосовуємо логістичну регресію. В дійсності сама по собі логістична регресія призначена не для класифікації даних, проте вона дозволяє спростити вирішення цього завдання.

Лістинг файлу *utilities.py*:

import numpy as np  
import matplotlib.pyplot as plt  
  
  
def visualize\_classifier(classifier, X, y):  
 # Define the minimum and maximum values for X and Y  
 # that will be used in the mesh grid  
 min\_x, max\_x = X[:, 0].min() - 1.0, X[:, 0].max() + 1.0  
 min\_y, max\_y = X[:, 1].min() - 1.0, X[:, 1].max() + 1.0  
  
 # Define the step size to use in plotting the mesh grid  
 mesh\_step\_size = 0.01  
  
 # Define the mesh grid of X and Y values  
 x\_vals, y\_vals = np.meshgrid(np.arange(min\_x, max\_x, mesh\_step\_size), np.arange(min\_y, max\_y, mesh\_step\_size))  
  
 # Run the classifier on the mesh grid  
 output = classifier.predict(np.c\_[x\_vals.ravel(), y\_vals.ravel()])  
  
 # Reshape the output array  
 output = output.reshape(x\_vals.shape)  
  
 # Create a plot  
 plt.figure()  
  
 # Choose a color scheme for the plot  
 plt.pcolormesh(x\_vals, y\_vals, output, cmap=plt.cm.gray)  
  
 # Overlay the training points on the plot  
 plt.scatter(X[:, 0], X[:, 1], c=y, s=75, edgecolors='black', linewidth=1, cmap=plt.cm.Paired)  
  
 # Specify the boundaries of the plot  
 plt.xlim(x\_vals.min(), x\_vals.max())  
 plt.ylim(y\_vals.min(), y\_vals.max())  
  
 # Specify the ticks on the X and Y axes  
 plt.xticks((np.arange(int(X[:, 0].min() - 1), int(X[:, 0].max() + 1), 1.0)))  
 plt.yticks((np.arange(int(X[:, 1].min() - 1), int(X[:, 1].max() + 1), 1.0)))  
  
 plt.show()

Лістинг власне програми:

import numpy as np  
from sklearn import linear\_model  
from utilities import visualize\_classifier  
  
# 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])  
  
# Створення логістичного класифікатора  
classifier = linear\_model.LogisticRegression(solver='liblinear', C=1)  
  
# Тренування класифікатора  
classifier.fit(X,y)  
visualize\_classifier(classifier, X, y)

Результат виконання програми:

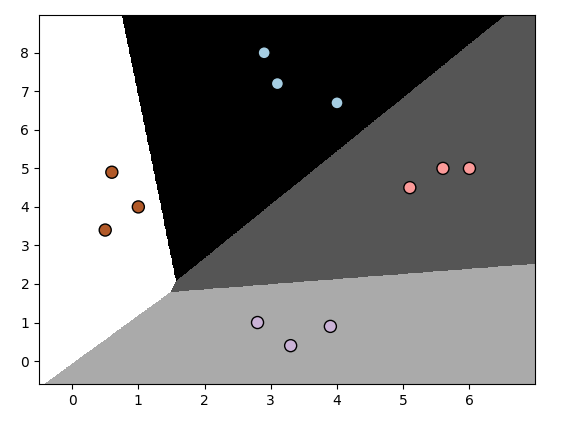


Рис. 4. Результат виконання програми

***Завдання 2.4.* Класифікація наївним байєсовським класифікатором**

**Наївний байєсовський класифікатор (Naive Bayes classifier)** — це простий класифікатор, заснований на використанні теореми Байєса, яка описує ймовірність події з урахуванням пов'язаних з нею умов. Такий класифікатор створюється за допомогою привласнення позначок класів екземплярам завдання. Останні представляються як векторів значень ознак. При цьому передбачається, що значення будь-якої заданої ознаки не залежить від значень інших ознак. Його припущення про незалежність ознак і становить наївну частину байєсовського класифікатора. Ми можемо оцінювати вплив будь-якої ознаки змінної класу незалежно від впливу інших ознак.

Лістинг програми:

import numpy as np  
from sklearn.naive\_bayes import GaussianNB  
from utilities import visualize\_classifier  
  
input\_file = 'data\_multivar\_nb.txt'  
  
data = np.loadtxt(input\_file, delimiter=',')  
X, y = data[:, :-1], data[:, -1]  
  
classifier = GaussianNB()  
classifier.fit(X, y)  
  
y\_pred = classifier.predict(X)  
accuracy = 100.0 \* (y == y\_pred).sum() / X.shape[0]  
  
print(accuracy)  
print("Accuracy of the Naive Bayes classifier =", round(accuracy, 2), "%")  
  
visualize\_classifier(classifier, X, y)

Результат виконання програми:

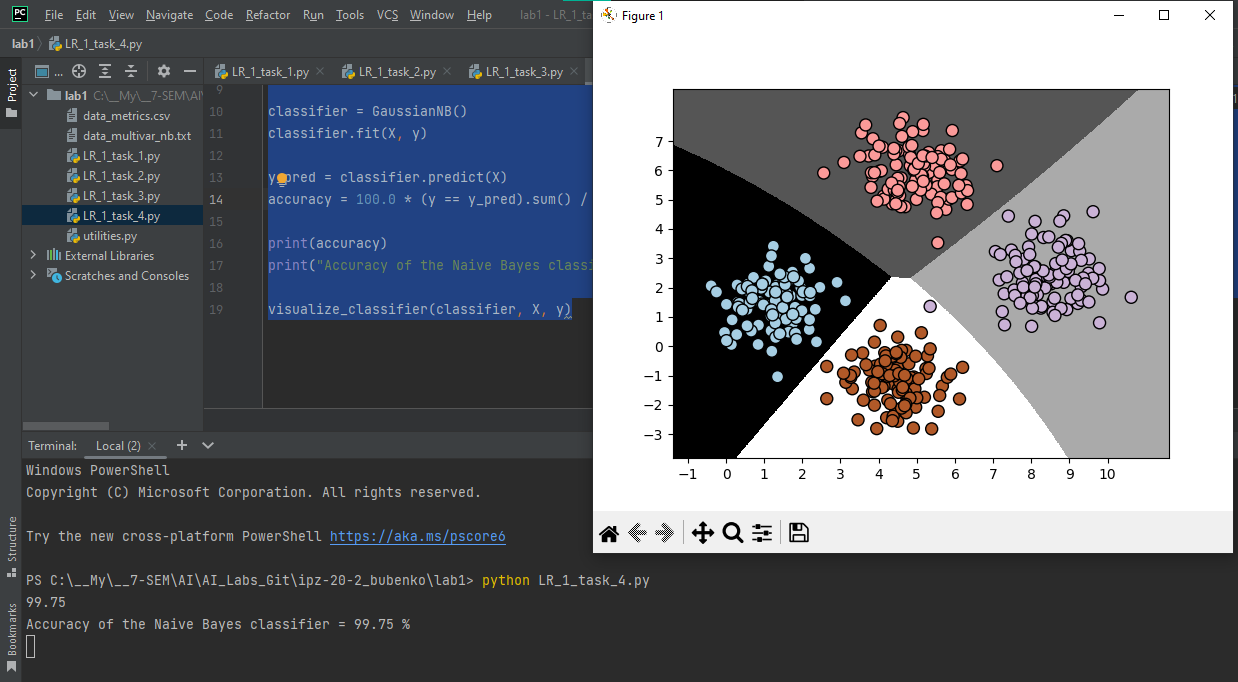




Рис. 5. Результат виконання програми

Лістинг модифікованої програми:

import numpy as np  
from sklearn.naive\_bayes import GaussianNB  
from sklearn.model\_selection import train\_test\_split, cross\_val\_score  
from utilities import visualize\_classifier  
  
input\_file = 'data\_multivar\_nb.txt'  
  
data = np.loadtxt(input\_file, delimiter = ',')  
  
X, y = data[:, :-1], data[:, -1]  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state = 3)  
  
classifier = GaussianNB()  
classifier.fit(X\_train, y\_train)  
  
y\_test\_pred = classifier.predict(X\_test)  
accuracy = 100.0 \* (y\_test == y\_test\_pred).sum() / X\_test.shape[0]  
  
print(accuracy)  
print("Accuracy of the new Naive Bayes classifier =", round(accuracy, 2), "%")  
  
visualize\_classifier(classifier, X\_test, y\_test)  
  
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)) + "%")

Результат виконання програми:

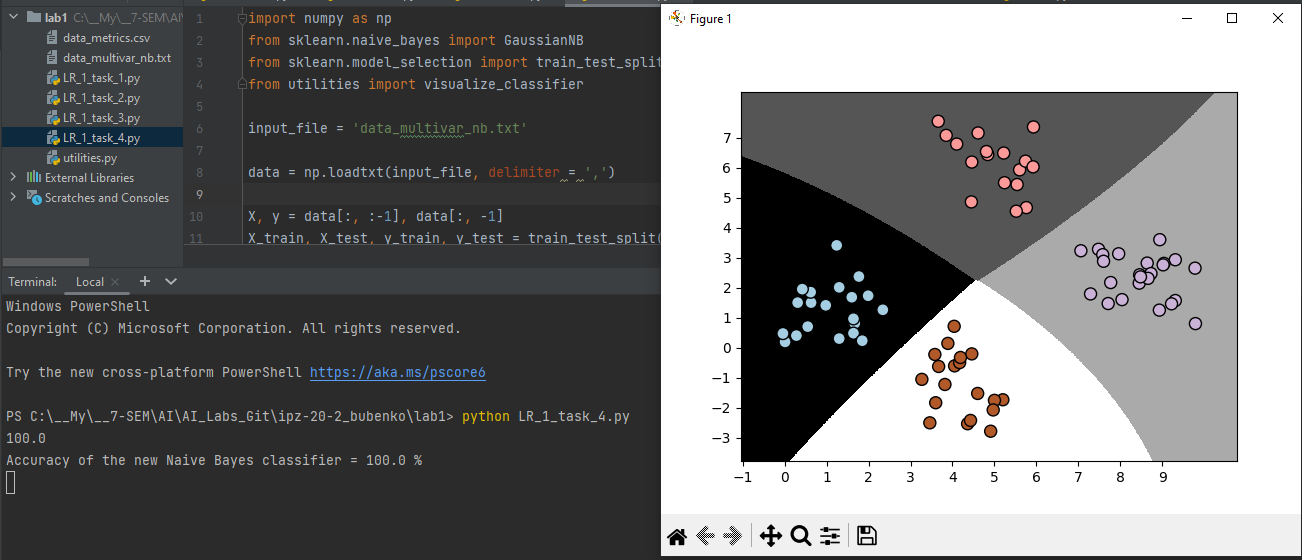
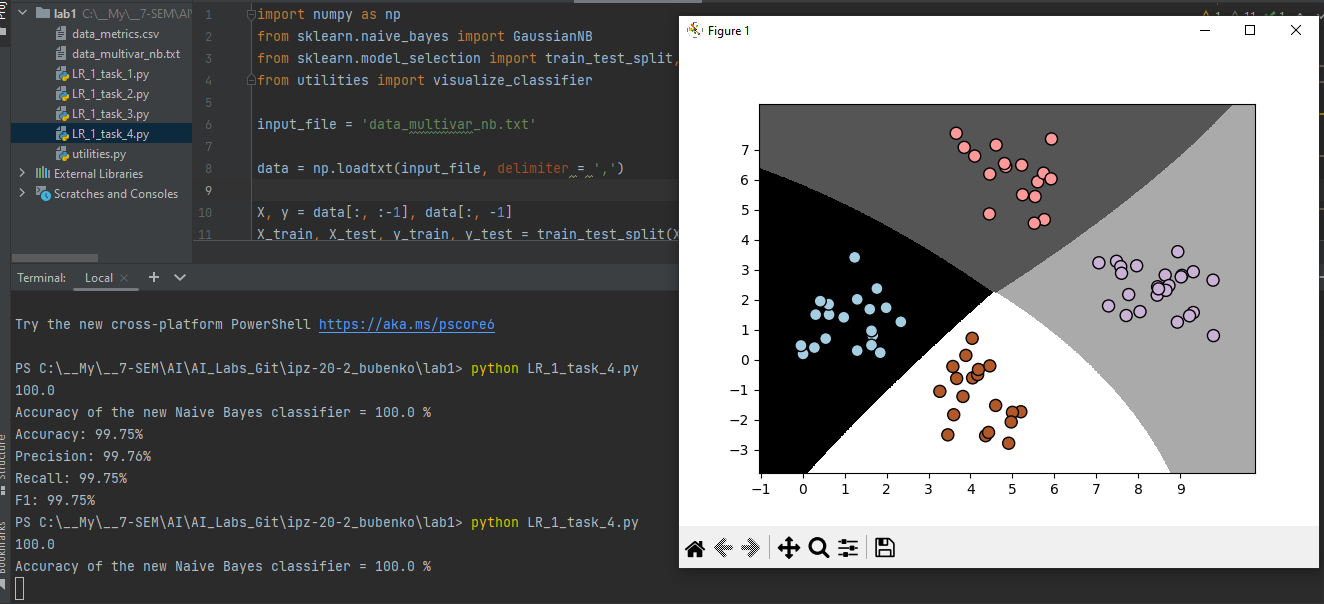




Рис. 6. Результат виконання програми



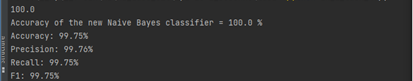


Рис. 7. Результат виконання програми (другий прогін)

***Порівняння результатів***

У першому випадку точність 99.75%. У другому випадку, після того як ми виконали перехресну перевірку та розбили дані на тестові та тренувальні точність підвищилась до 100%.Отже, перший метод є ненадійним.

***Завдання 2.5.* Вивчити метрики якості класифікації**

Класифікація полягає у спробі передбачити, з якого класу надходить конкретна вибірка із популяції.

У цьому завданні ми розглянемо деякі з цих метрик і напишемо власні функції з нуля, щоб зрозуміти математику, що лежить в основі деяких з них.

Запрограмуємо наступні показники зі sklearn.metrics:

confusion\_matrix – матриця помилок (або матриця неточностей чи плутанини);

accuracy\_score – акуратність (з англ. може перекласти як точність, але не плутайте бо то інший показник)

recall\_score – повнота

precision\_score – точність

f1\_score – F-міра

roc\_curve – ROC-крива, крива робочих характеристик (англ. Receiver Operating Characteristics curve).

roc\_auc\_score – вимір площі під ROC-кривою (англ. Area Under the Curve - AUC). (ROC-AUC)

Лістинг програми:

import pandas as pd  
import numpy as np  
from sklearn.metrics import accuracy\_score, confusion\_matrix, recall\_score, precision\_score, f1\_score, roc\_curve, \  
 roc\_auc\_score  
import matplotlib.pyplot as plt  
  
df = pd.read\_csv('data\_metrics.csv')  
df.head()  
  
df['predicted\_RF'] = (df.model\_RF >= 0.5).astype('int')  
df['predicted\_LR'] = (df.model\_LR >= 0.5).astype('int')  
df.head()  
  
confusion\_matrix(df.actual\_label.values, df.predicted\_RF.values)  
  
  
def find\_TP(y\_true, y\_pred):  
 return sum((y\_true == 1) & (y\_pred == 1))  
  
  
def find\_FN(y\_true, y\_pred):  
 return sum((y\_true == 1) & (y\_pred == 0))  
  
  
def find\_FP(y\_true, y\_pred):  
 return sum((y\_true == 0) & (y\_pred == 1))  
  
  
def find\_TN(y\_true, y\_pred):  
 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))  
  
  
def find\_conf\_matrix\_values(y\_true, y\_pred):  
 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 gaevsky\_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]])  
  
  
gaevsky\_confusion\_matrix(df.actual\_label.values, df.predicted\_RF.values)  
  
assert np.array\_equal(gaevsky\_confusion\_matrix(df.actual\_label.values, df.predicted\_RF.values),  
 confusion\_matrix(df.actual\_label.values, df.predicted\_RF.values))  
  
  
def gaevsky\_accuracy\_score(y\_true, y\_pred):  
 TP, FN, FP, TN = find\_conf\_matrix\_values(y\_true, y\_pred)  
 return (TP + TN) / (TP + FP + TN + FN)  
  
  
assert gaevsky\_accuracy\_score(df.actual\_label.values, df.predicted\_RF.values) == accuracy\_score(df.actual\_label.values,  
 df.predicted\_RF.values), 'gaevsky\_accuracy\_score failed on RF'  
  
assert gaevsky\_accuracy\_score(df.actual\_label.values, df.predicted\_LR.values) == accuracy\_score(df.actual\_label.values,  
 df.predicted\_LR.values), 'gaevsky\_accuracy\_score failed on LR'  
  
print('Accuracy RF: %.3f' % (gaevsky\_accuracy\_score(df.actual\_label.values, df.predicted\_RF.values)))  
print('Accuracy LR: %.3f' % (gaevsky\_accuracy\_score(df.actual\_label.values, df.predicted\_LR.values)))  
  
accuracy\_score(df.actual\_label.values, df.predicted\_RF.values)  
recall\_score(df.actual\_label.values, df.predicted\_RF.values)  
  
  
def gaevsky\_recall\_score(y\_true, y\_pred):  
 TP, FN, FP, TN = find\_conf\_matrix\_values(y\_true, y\_pred)  
 return TP / (TP + FN)  
  
  
assert gaevsky\_recall\_score(df.actual\_label.values, df.predicted\_RF.values) == recall\_score(df.actual\_label.values,  
 df.predicted\_RF.values), 'gaevsky\_recall\_score failed on RF'  
  
assert gaevsky\_recall\_score(df.actual\_label.values, df.predicted\_LR.values) == recall\_score(df.actual\_label.values,  
 df.predicted\_LR.values), 'gaevsky\_recall\_score failed on LR'  
  
print('Recall RF: %.3f' % (gaevsky\_recall\_score(df.actual\_label.values, df.predicted\_RF.values)))  
print('Recall LR: %.3f' % (gaevsky\_recall\_score(df.actual\_label.values, df.predicted\_LR.values)))  
  
precision\_score(df.actual\_label.values, df.predicted\_RF.values)  
  
  
def gaevsky\_precision\_score(y\_true, y\_pred):  
 TP, FN, FP, TN = find\_conf\_matrix\_values(y\_true, y\_pred)  
 return TP / (TP + FP)  
  
  
assert gaevsky\_precision\_score(df.actual\_label.values, df.predicted\_RF.values) == precision\_score(  
 df.actual\_label.values, df.predicted\_RF.values), 'gaevsky\_precision\_score failed on RF'  
  
assert gaevsky\_precision\_score(df.actual\_label.values, df.predicted\_LR.values) == precision\_score(  
 df.actual\_label.values, df.predicted\_LR.values), 'gaevsky\_precision\_score failed on LR'  
  
print('Precision RF: %.3f' % (gaevsky\_precision\_score(df.actual\_label.values, df.predicted\_RF.values)))  
print('Precision LR: %.3f' % (gaevsky\_precision\_score(df.actual\_label.values, df.predicted\_LR.values)))  
  
f1\_score(df.actual\_label.values, df.predicted\_RF.values)  
  
  
def gaevsky\_f1\_score(y\_true, y\_pred):  
 TP, FN, FP, TN = find\_conf\_matrix\_values(y\_true, y\_pred)  
 recall = gaevsky\_recall\_score(y\_true, y\_pred)  
 precision = gaevsky\_precision\_score(y\_true, y\_pred)  
 return 2 \* (recall \* precision) / (recall + precision)  
  
  
assert gaevsky\_f1\_score(df.actual\_label.values, df.predicted\_RF.values) == f1\_score(df.actual\_label.values,  
 df.predicted\_RF.values), 'gaevsky\_f1\_score failed on RF'  
assert gaevsky\_f1\_score(df.actual\_label.values, df.predicted\_LR.values) == f1\_score(df.actual\_label.values,  
 df.predicted\_LR.values), 'gaevsky\_f1\_score failed on LR'  
  
print('F1 RF: %.3f' % (gaevsky\_f1\_score(df.actual\_label.values, df.predicted\_RF.values)))  
print('F1 LR: %.3f' % (gaevsky\_f1\_score(df.actual\_label.values, df.predicted\_LR.values)))  
  
print('\nscores with threshold = 0.5')  
print('Accuracy RF: %.3f' % (gaevsky\_accuracy\_score(df.actual\_label.values, df.predicted\_RF.values)))  
print('Recall RF: %.3f' % (gaevsky\_recall\_score(df.actual\_label.values, df.predicted\_RF.values)))  
print('Precision RF: %.3f' % (gaevsky\_precision\_score(df.actual\_label.values, df.predicted\_RF.values)))  
print('F1 RF: %.3f' % (gaevsky\_f1\_score(df.actual\_label.values, df.predicted\_RF.values)))  
print('Accuracy LR: %.3f' % (gaevsky\_accuracy\_score(df.actual\_label.values, df.predicted\_LR.values)))  
print('Recall LR: %.3f' % (gaevsky\_recall\_score(df.actual\_label.values, df.predicted\_LR.values)))  
print('Precision LR: %.3f' % (gaevsky\_precision\_score(df.actual\_label.values, df.predicted\_LR.values)))  
print('F1 LR: %.3f' % (gaevsky\_f1\_score(df.actual\_label.values, df.predicted\_LR.values)))  
print('')  
  
print('scores with threshold = 0.25')  
print(  
 'Accuracy RF: %.3f' % (gaevsky\_accuracy\_score(df.actual\_label.values, (df.model\_RF >= 0.25).astype('int').values)))  
print('Recall RF: %.3f' % (gaevsky\_recall\_score(df.actual\_label.values, (df.model\_RF >= 0.25).astype('int').values)))  
print('Precision RF: %.3f' % (  
 gaevsky\_precision\_score(df.actual\_label.values, (df.model\_RF >= 0.25).astype('int').values)))  
print('F1 RF: %.3f' % (gaevsky\_f1\_score(df.actual\_label.values, (df.model\_RF >= 0.25).astype('int').values)))  
print(  
 'Accuracy LR: %.3f' % (gaevsky\_accuracy\_score(df.actual\_label.values, (df.model\_LR >= 0.25).astype('int').values)))  
print('Recall LR: %.3f' % (gaevsky\_recall\_score(df.actual\_label.values, (df.model\_LR >= 0.25).astype('int').values)))  
print('Precision LR: %.3f' % (  
 gaevsky\_precision\_score(df.actual\_label.values, (df.model\_LR >= 0.25).astype('int').values)))  
print('F1 LR: %.3f' % (gaevsky\_f1\_score(df.actual\_label.values, (df.model\_LR >= 0.25).astype('int').values)))  
  
fpr\_RF, tpr\_RF, thresholds\_RF = roc\_curve(df.actual\_label.values, df.model\_RF.values)  
fpr\_LR, tpr\_LR, thresholds\_LR = roc\_curve(df.actual\_label.values, df.model\_LR.values)  
  
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)  
  
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()

Результат виконання програми:

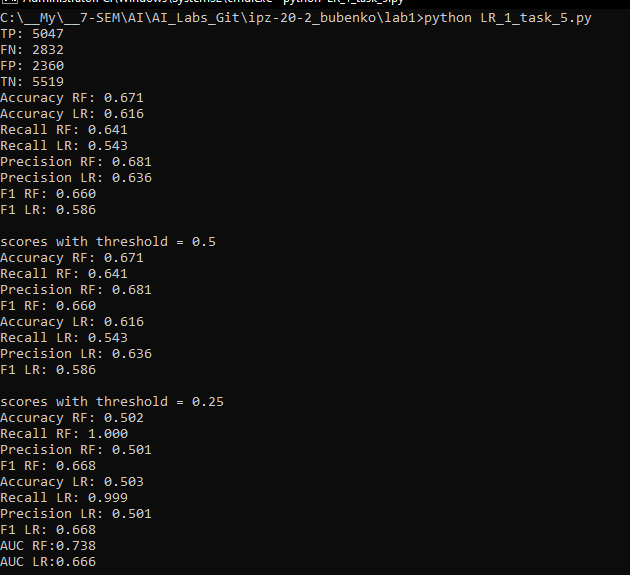


Рис. 8. Результат виконання програми

***Висновок щодо результатів для різних порогів (0.5 та 0.25)***

Отже, зі зібльшенням порогу, значення міри F1 зменшується. Зниження порогу до 0.25 призводить до збільшення кількості виявлених позитивних випадків, що впливає на збільшення значення Recall, але при цьому знижує точність моделі, яка відображається у вищому значенні False Positives та зменшенні Precision.

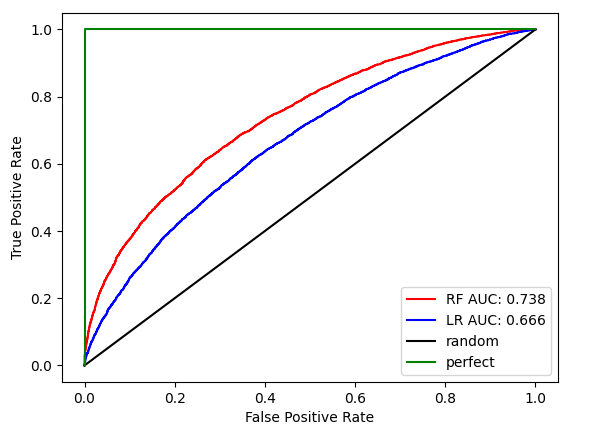


Рис. 9. Результат виконання програми (ROC-крива)

***Висновок щодо того, яка з двох моделей (LR чи RF) краща і чому***

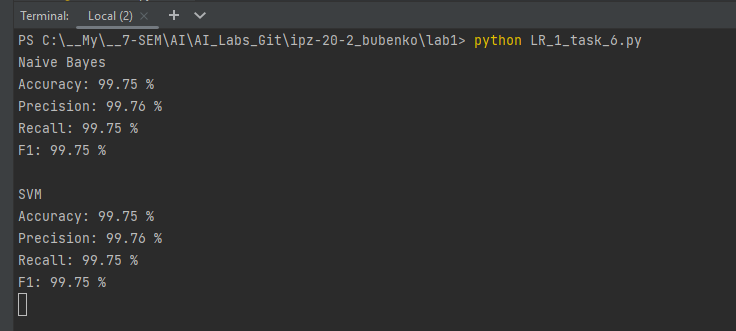
Отже, проаналізувавши графік, можна помітити, що RF модель має більшу точність, аніж LR модель. RF має кращу здатність розділяти класи та забезпечує більш високу продуктивність в даній задачі класифікації. Але можуть бути ситуації, коли LR матиме переваги перед RF, тому важливо враховувати складність моделі.

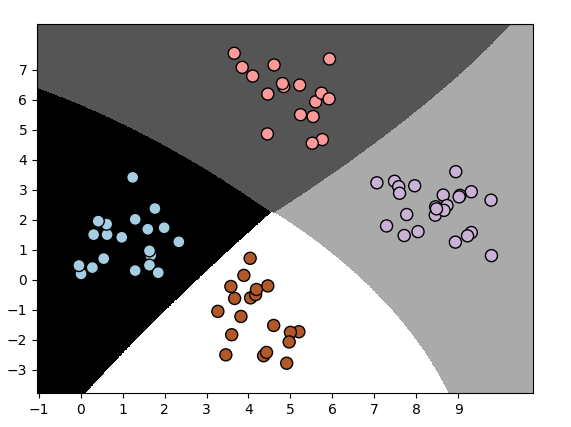
***Завдання 2.6.* Розробіть програму класифікації даних в файлі data\_multivar\_nb.txt за допомогою машини опорних векторів (Support Vector Machine - SVМ). Розрахуйте показники якості класифікації. Порівняйте їх з показниками наївного байєсівського класифікатора. Зробіть висновки яку модель класифікації краще обрати і чому.**

Лістинг програми:

from sklearn.svm import SVC  
from sklearn.naive\_bayes import GaussianNB  
import numpy as np  
from sklearn.model\_selection import train\_test\_split, cross\_val\_score  
from utilities import visualize\_classifier  
  
  
def evaluate\_classifier(classifier, X, y, num\_folds=3):  
 accuracy\_values = cross\_val\_score(classifier, X, y, scoring='accuracy', cv=num\_folds)  
 precision\_values = cross\_val\_score(classifier, X, y, scoring='precision\_weighted', cv=num\_folds)  
 recall\_values = cross\_val\_score(classifier, X, y, scoring='recall\_weighted', cv=num\_folds)  
 f1\_values = cross\_val\_score(classifier, X, y, scoring='f1\_weighted', cv=num\_folds)  
  
 accuracy\_mean = round(100 \* accuracy\_values.mean(), 2)  
 precision\_mean = round(100 \* precision\_values.mean(), 2)  
 recall\_mean = round(100 \* recall\_values.mean(), 2)  
 f1\_mean = round(100 \* f1\_values.mean(), 2)  
  
 return accuracy\_mean, precision\_mean, recall\_mean, f1\_mean  
  
  
input\_file = 'data\_multivar\_nb.txt'  
data = np.loadtxt(input\_file, delimiter=',')  
X, y = data[:, :-1], data[:, -1]  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=3)  
  
classifier\_svm = SVC()  
classifier\_svm.fit(X\_train, y\_train)  
  
classifier\_nb = GaussianNB()  
classifier\_nb.fit(X\_train, y\_train)  
  
num\_folds = 3  
  
print('Naive Bayes')  
nb\_accuracy, nb\_precision, nb\_recall, nb\_f1 = evaluate\_classifier(classifier\_nb, X, y, num\_folds)  
print("Accuracy:", nb\_accuracy, "%")  
print("Precision:", nb\_precision, "%")  
print("Recall:", nb\_recall, "%")  
print("F1:", nb\_f1, "%")  
  
print('\nSVM')  
svm\_accuracy, svm\_precision, svm\_recall, svm\_f1 = evaluate\_classifier(classifier\_svm, X, y, num\_folds)  
print("Accuracy:", svm\_accuracy, "%")  
print("Precision:", svm\_precision, "%")  
print("Recall:", svm\_recall, "%")  
print("F1:", svm\_f1, "%")  
  
visualize\_classifier(classifier\_nb, X\_test, y\_test)  
visualize\_classifier(classifier\_svm, X\_test, y\_test)

Результат виконання програми:





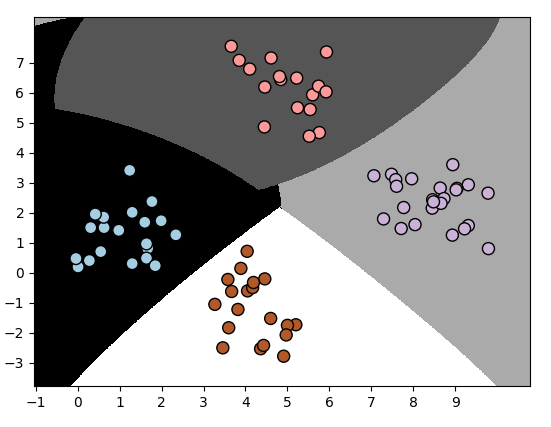


Рис. 10. - 12. Результат виконання програми

***Висновки про те, яку модель класифікації краще обрати і чому***

У нашому прикладі показнки SVM та наївного байєсівського класифікатора є однаковими, тому в нашому випадку можна обрати як першу, так і другу модель класифікації.

Проте, вибір моделі залежить від багатьох факторів, таких як розмір вибірки, складність проблеми та бажані метрики. SVM зазвичай працює добре в задачах з багатьма ознаками, коли є чітка границя між класами. Головним недоліком методу є те, що він підходить тільки до розв'язання завдань з двома класами. Наївний баєсівський класифікаторможе бути дуже ефективним в простих класифікаційних завданнях та в тих випадках, коли взаємозв'язки між ознаками досить прості.

***Висновок:*** у ході виконання лабораторної роботи я, використовуючи спеціалізовані бібліотеки та мову програмування Python, дослідив попередню обробку та класифікацію даних, зокрема, L1-нормалізацію та L2-нормалізацію, методи класифікації та різницю між моделями RF і LR.