

ЛАБОРАТОРНА РОБОТА № 1

ПОПЕРЕДНЯ ОБРОБКА ТА КОНТРОЛЬОВАНА КЛАСИФІКАЦІЯ ДАНИХ

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

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

Теоретичні відомості подані на лекціях. Також доцільно вивчити матеріал поданий в літературі:

Джоши Пратик. Искусственный интеллект с примерами на Python. : Пер. с англ. - СПб. : ООО "Диалектика", 2019. - 448 с. - Парал. тит. англ. ISBN 978-5-907114-41-8 (рус.)

Можна використовувати [Google Colab](https://colab.research.google.com/) або Jupiter Notebook.

2. ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ ТА МЕТОДИЧНІ РЕКОМЕНДАЦІЇ ДО ЙОГО ВИКОНАННЯ

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. Покажемо, як це робиться.

Створіть новий файл Python та імпортуйте такі пакети.

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

Визначимо деяку вибірку даних.

```
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]])
```

Розглянемо декілька різних методів попередньої обробки даних

2.1.1. Бінарізація

Цей 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.

Додамо наступні рядки до того ж файлу Python.

```
# Бінаризація даних
data_binarized =
preprocessing.Binarizer(threshold=2.1).transform(input_data)
print("\n Binarized data:\n", data_binarized)
```

Виконавши код, отримаємо результат

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

Як бачите, всі значення понад 2,1 примусово встановлюються рівними 1. Інші значення стають рівними 0.

2.1.2. Виключення середнього

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.

Додамо наступні рядки до того ж файлу Python, який використовувався в попередньому завданні 2.1.1.

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

Ці рядки коду відображають середнє значення і середньоквадратичне відхилення для вхідних даних.

Тепер виключимо середнє значення

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

Після виконання коду отримаємо:

```
BEFORE:
Mean = [ 3.775 -1.15 -1.3 ]
Std deviation = [3.12039661 6.36651396 4.0620192 ]

AFTER:
Mean = [1.11022302e-16 0.00000000e+00 2.77555756e-17]
Std deviation = [1. 1. 1.]
```

Неважко помітити, що середнє значення практично рівне 0, а стандартне відхилення - 1.

2.1.3. 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.

Додамо до того ж файлу Python наступні рядки.

```
# 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)
```

Після виконання коду отримаємо такий результат:

```
Min max scaled data:
[[0.74117647 0.39548023 1.          ]
 [0.          1.          0.          ]
 [0.6         0.5819209  0.87234043]
 [1.          0.          0.17021277]]
```

Кожен рядок відмасштабований таким чином, щоб максимальним значенням була б 1, а всі решта значень визначалися відносно неї.

2.1.4. 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 порівнянню з *L2-нормалізацією*, оскільки вона менш чутлива до викидів.

Дуже часто дані містять викиди, і з цим нічого не вдієш. Ми хочемо використовувати безпечні методи, 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*.

Додамо наступні рядки в той же файл Python.

```
# 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)
```

Виконавши код отримаємо наступні результати

```
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]]
```

Копії екрану з кодом програми та результатами занесіть у звіт (рис.1 звіту). Зробіть висновок чим відрізняються L1-нормалізація від L2-нормалізації

2.1.5. Кодування міток

Як правило, в процесі класифікації даних ми маємо справу з множиною міток (labels). Ними можуть бути слова, числа або інші об'єкти. Функції машинного навчання, що входять до бібліотеки *sklearn*, очікують, що мітки є числами. Тому, якщо мітки – це вже числа, ми можемо використовувати їх безпосередньо для того, щоб почати тренування. Однак, зазвичай, це не так. На практиці мітками служать слова, оскільки в такому вигляді вони краще всього сприймаються людиною. Ми позначаємо тренувальні дані словами, щоб полегшити відстеження відповідей. Для перетворення слів у числа необхідно використовувати *кодування*. Під *кодуванням міток* (label encoding) мається на увазі процес перетворення словесних міток на числову форму. Завдяки цьому алгоритми можуть оперувати нашими даними.

Створіть новий файл Python та імпортуйте такі пакети.

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

Визначимо мітки.

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

Створимо об'єкт кодування міток та навчимо його.

```
# Створення кодувальника та встановлення відповідності
# між мітками та числами
encoder = preprocessing.LabelEncoder()
encoder.fit(input_labels)
```

Виведемо відображення слів на числа.

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

Перетворимо набір випадково впорядкованих міток, щоб перевірити роботу кодувальника.

```
# 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 ) )
```

Декодуємо випадковий набір чисел.

```
# 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 ) )
```

Після виконання цього коду у вікні терміналу повинна відобразитися інформація. Проаналізуйте цю інформацію.

Копії екрану з кодом програми та результатами занесіть у звіт (рис. 2 звіту)

Програмний код збережіть під назвою LR_1_task_1.py

Завдання 2.2. Попередня обробка нових даних

У кодї 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.

Копії екрану з кодом програми та результатами занесіть у звіт (рис. 3 звіту)

Таблиця 1

№ варіанту	Значення змінної <code>input_data</code>												Поріг бінарізації
1.	4.3	-9.9	-3.5	-2.9	4.1	3.3	-2.2	8.8	-6.1	3.9	1.4	2.2	2.2
2.	4.1	-5.9	-3.5	-1.9	4.6	3.9	-4.2	6.8	6.3	3.9	3.4	1.2	3.2
3.	1.3	-3.9	6.5	-4.9	-2.2	1.3	2.2	6.5	-6.1	-5.4	-1.4	2.2	1.1
4.	-5.3	-8.9	3.0	2.9	5.1	-3.3	3.1	-2.8	-3.2	2.2	-1.4	5.1	3.0
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
6.	2.3	-1.6	6.1	-2.4	-1.2	4.3	3.2	5.5	-6.1	-4.4	1.4	-1.2	2.1
7.	1.3	3.9	6.2	4.9	2.2	-4.3	-2.2	6.5	4.1	-5.2	-3.4	-5.2	2.0
8.	4.6	9.9	-3.5	-2.9	4.1	3.3	-2.2	8.8	-6.1	3.9	1.4	2.2	2.2
9.	4.1	-5.9	3.3	6.9	4.6	3.9	-4.2	3.8	2.3	3.9	3.4	1.2	3.2
10.	1.3	-3.9	6.5	-4.9	-2.2	1.3	2.2	6.5	-6.1	3.4	-3.4	-2.2	1.2
11.	-5.3	-8.9	3.0	2.9	5.1	-3.3	3.1	-2.8	-3.2	2.2	-1.4	5.1	2.0
12.	-1.3	3.9	4.5	-5.3	-4.2	3.3	-5.2	-6.5	-1.1	-5.2	2.6	-2.2	1.8
13.	-2.3	-1.6	-6.1	-2.4	-1.2	4.3	3.2	3.1	6.1	-4.4	1.4	-1.2	2.1
14.	-1.3	3.9	6.2	-4.9	2.2	-4.3	-2.2	6.5	4.1	-5.2	-3.4	-5.2	2.2
15.	-2.3	3.9	-4.5	-5.3	-4.2	-1.3	5.2	-6.5	-1.1	-5.2	2.6	-2.2	3.0
16.	-3.3	-1.6	6.1	2.4	-1.2	4.3	-3.2	5.5	-6.1	-4.4	1.4	-1.2	2.1
17.	1.3	3.9	6.2	4.9	2.2	-4.3	-2.6	6.5	4.1	-5.2	-3.4	-5.2	2.0
18.	4.6	3.9	-3.5	-2.9	4.1	3.3	2.2	8.8	-6.1	3.9	1.4	2.2	2.2
19.	-4.1	-5.5	3.3	6.9	4.6	3.9	-4.2	3.8	2.3	3.9	3.4	-1.2	3.2
20.	6.3	-3.9	6.5	-4.9	-2.2	1.3	2.2	6.5	-6.1	-3.4	5.2	-1.2	1.2
21.	-5.3	-8.9	3.0	2.9	5.1	-3.3	3.1	-2.8	-3.2	4.2	-1.4	6.1	2.0
22.	-1.6	3.9	4.5	-4.3	4.2	3.3	-5.2	-6.5	5.1	-5.2	2.6	-2.2	3.8
23.	2.5	-1.6	-6.1	-2.4	-1.2	4.3	3.2	3.1	6.1	-4.4	1.4	-1.2	2.5
24.	-4.3	3.3	-6.2	4.9	5.2	-5.3	-4.2	6.5	4.4	-3.2	-3.4	6.1	2.2
25.	-3.3	-1.6	6.1	2.4	-1.2	4.3	-3.2	5.5	-5.3	-4.4	1.4	-1.2	2.1
26.	1.3	3.9	6.2	4.9	2.2	-4.3	-2.6	6.5	4.6	-5.2	-3.4	-5.2	2.7
27.	4.6	3.9	-3.5	-2.9	4.1	3.3	2.2	8.8	-4.1	3.9	2.4	4.2	2.2
28.	-4.1	-5.5	3.3	6.9	4.6	3.9	-4.2	3.8	2.3	3.9	3.4	-1.2	3.0
29.	1.6	3.1	6.5	-4.9	-2.2	1.3	2.2	6.5	-6.1	-3.4	5.2	-3.2	3.2
30.	-5.3	-8.9	4.2	2.9	-5.0	-3.3	3.1	-2.8	-3.2	4.2	-1.4	6.1	1.5

Програмний код збережіть під назвою `LR_1_task_2.py`

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. Зважаючи на її простість, логістичну регресію часто задіюють у машинному навчанні. Розглянемо приклад застосування логістичної регресії до створення класифікатора. Перш ніж продовжити, переконайтеся, що у вашій системі встановлено пакет Tkinter. У разі потреби ви зможете знайти його за адресою <https://docs.python.org/2/library/tkinter.html>.

Створіть новий файл Python та імпортуйте наведені нижче пакети.

Ми імпортуватимемо одну з функцій, що містяться у файлі `utilities.py`. Ці функції не є обов'язковими для проведення класифікації, але вони дозволяють наглядно показати, що ж відбувається при класифікації (візуалізувати функції та результати). Нижче ми познайомимось із цією функцією більш детально, а поки що просто імпортуйте її. (перепишіть файл у директорію вашого проекту).

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

Визначимо зразок вхідних даних за допомогою двовимірних векторів і відповідних міток.

```
# 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])
```

Ми тренуватимемо класифікатор, використовуючи ці позначені дані.
Створимо об'єкт логістичного класифікатора.

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

Навчимо класифікатор, використовуючи певні дані.

```
# Тренування класифікатора
classifier.fit(X, y)
```

Візуалізуємо результати роботи класифікатора, відстеживши межі

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

Копії екрану з кодом програми та результатами занесіть у звіт (рис. 4 звіту)

Програмний код збережіть під назвою LR_1_task_3.py

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. Наприклад, ми можемо вважати тварину гепардом, якщо воно має плямисту шкіру, чотири лапи та хвіст і розвиває швидкість, рівну приблизно 70 миль на годину. У разі використання наївного байєсівського класифікатора вважається, що кожна з ознак робить незалежний внесок у кінцевий результат, що оцінює ймовірність те, що тварина є гепардом. Ми не обтяжуватимемо себе розглядом кореляції між малюнком шкіри, кількістю лап, наявністю хвоста та швидкістю переміщення.

Займемося створенням наївного байєсівського класифікатора. Створіть новий файл Python та імпоруйте такі пакети.

```
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
```

Як джерело даних ми будемо використовувати файл data_multivar_nb.txt, кожен рядок якого містить значення розділені комою.

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

Завантажимо дані із цього файлу.

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

Створимо екземпляр наївного байєсовського класифікатора. У даному випадку ми будемо використовувати гаусівський наївний байєсівський класифікатор, в якому передбачається, що значення, які асоціюються з кожним класом, дотримуються закону розподілу Гауса.

```
# Створення наївного байєсовського класифікатора
classifier = GaussianNB()
```

Навчимо класифікатор, використовуючи тренувальні дані.

```
# Тренування класифікатора
classifier.fit(X, y)
```

Запустимо класифікатор на тренувальних даних та спрогнозуємо результати.

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

Обчислимо якість (ассураcy)¹ класифікатора, порівнявши передбачені значення з істинними мітками, а потім візуалізуємо результат.

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

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

Копії екрану з кодом програми та результатами занесіть у звіт (рис. 5 звіту)

Попередній метод обчислення якості класифікатора не є надійним. Нам потрібно виконати перехресну перевірку, щоб не використовувати ті самі тренувальні дані при тестуванні.

Розіб'ємо дані на навчальний та тестовий набори. Відповідно до значення параметра `test_size`, зазначеного в рядку коду нижче, ми віднесемо 80% даних до тренування, а 20% - до тестування. Потім ми виконаємо тренування найпростішим байєсовським класифікатором на цих даних.

```
# Розбивка даних на навчальний та тестовий набори
X_train, X_test, y_train, y_test =
train_test_split.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)
```

Обчислимо якість класифікатора і візуалізуємо результати.

```
# 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)
```

Скористаємося вбудованими функціями для обчислення якості (акурасу), точності (precision) 2 та повноти (recall) 3 класифікатора на підставі потрійної перехрестної перевірки.

```
num_folds = 3
accuracy_values = train_test_split.cross_val_score(classifier,
X, y, scoring='accuracy', cv=num_folds)
print("Accuracy: " + str(round(100 * accuracy_values.mean(), 2))
+ "%")

precision_values = train_test_split.cross_val_score(classifier,
X, y, scoring='precision_weighted', cv=num_folds)
print("Precision: " + str(round(100 * precision_values.mean(),
2)) + "%")

recall_values = train_test_split.cross_val_score(classifier,
X, y, scoring='recall_weighted', cv=num_folds)
print("Recall: " + str(round(100 * recall_values.mean(), 2)) +
"%")

f1_values = train_test_split.cross_val_score(classifier,
```

```
X, y, scoring='f1_weighted', cv=num_folds)
print("F1: " + str(round(100 * f1_values.mean(), 2)) + "%")
```

Виконавши цей код, ви отримаєте для першого тренувального прогону зображення. На ньому показані межі, отримані за допомогою класифікатора.

Копії екрану з кодом програми та результатами занесіть у звіт (рис. 6 звіту)

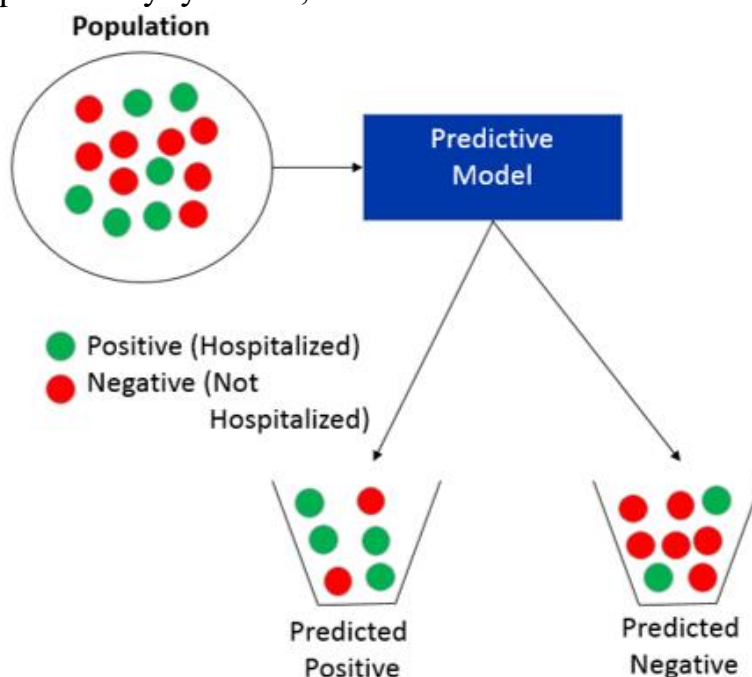
Зробіть ще один прогін та зображення результатів класифікації занесіть у звіт. (рис. 7 звіту)

Порівняйте між собою результати висновок запишіть у звіт

Програмний код збережіть під назвою LR_1_task_4.py

Завдання 2.5. Вивчити метрики якості класифікації

Класифікація полягає у спробі передбачити, з якого класу надходить конкретна вибірка із популяції. Наприклад, якщо ми намагаємося передбачити, чи буде конкретного пацієнта повторно госпіталізовано, то можливі два класи: госпіталізований (позитивний) і не госпіталізований (негативний). Потім модель класифікації намагається передбачити, чи кожен пацієнт буде госпіталізований чи не госпіталізований. Іншими словами, класифікація просто намагається передбачити, який сегмент (прогнозований позитивний або прогнозований негативний) має бути розміщений конкретною вибіркою із сукупності, як показано нижче.



Коли ви тренуєте свою прогностичну класифікаційну модель, ви захочете оцінити, наскільки вона хороша. Цікаво, що є безліч різних способів оцінки продуктивності. Більшість дослідників даних, які використовують Python для прогнозного моделювання використовують пакет scikit-learn. Scikit-learn Python. Він містить багато вбудованих функцій для аналізу продуктивності моделей. 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)

Ми напишемо наші власні функції з нуля, припускаючи двокласову класифікацію.

Зверніть увагу, що вам потрібно буде заповнити частини, позначені як # your code here

Завантажте зразок набору даних, який має фактичні мітки (actual_label) та ймовірності прогнозування для двох моделей (model_RF та model_LR). Тут ймовірність - це можливість бути 1-м класом.

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

Вони матимуть такий вигляд

	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

У більшості проектів ви визначатимете граничне значення, щоб визначити, які ймовірності прогнозування позначені як прогнозований позитивний або передбачений негативний результат. А поки що давайте припустимо, що поріг дорівнює 0,5. Давайте додамо два додаткові стовпці, які перетворюють ймовірності на передбачені мітки.

```
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
4	0	0.418302	0.369532	0	0

Матриця помилок (confusion_matrix)

Матриця неточностей (confusion matrix) – це таблиця, що використовується для опису ефективності класифікатора. Зазвичай вона витягується з тестового набору даних, для якого відомі базові справжні значення.

Ми аналізуємо результати віднесення до кожного класу та визначаємо частку невірно віднесених класів. У процесі конструювання вищезгаданої таблиці ми маємо справу з кількома ключовими метриками, що грають дуже важливу роль у машинному навчанні.

Для нашої задачі, враховуючи фактичну мітку та прогнозовану мітку, перше, що ми можемо зробити, це розділити наші вибірки на 4 сегменти:

Істинно позитивний – фактичний = 1, прогнозований = 1

Хибний позитивний – фактичний = 0, прогнозований = 1

Невірно негативний - фактичний = 1, прогнозований = 0

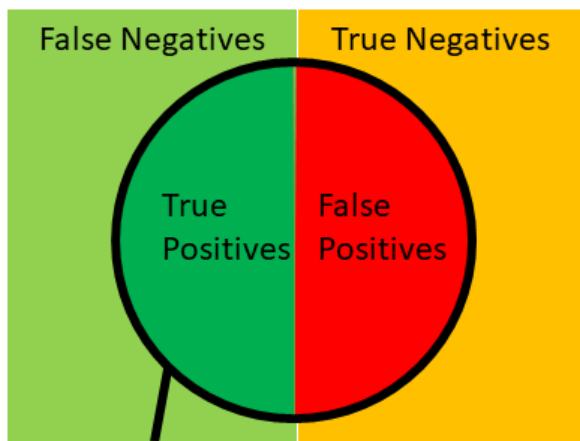
Істинно негативний - фактичний = 0, прогнозований = 0

Ці сегменти можуть бути представлені наступним зображенням

(джерело

https://en.wikipedia.org/wiki/Precision_and_recall#/media/File:Precisionrecall.svg

) і ми будемо посилатися на це зображення в багатьох розрахунках нижче.



Predicted Positive

Ці сегменти можна подати і у вигляді матриці чи таблиці

Confusion Matrix		Predicted	
		Negative	Positive
Actual	Negative	True Negative	False Positive
	Positive	False Negative	True Positive

Ми можемо отримати матрицю помилок (у вигляді масиву 2x2) з scikit-learn, яка приймає як вхідні дані фактичні мітки та передбачені мітки

```
from sklearn.metrics import
confusion_matrixconfusion_matrix(df.actual_label.values,
df.predicted RF.values)
```

```
array([[5519, 2360],
       [2832, 5047]], dtype=int64)
```

де було 5047 істинних позитивних результатів, 2360 помилкових позитивних результатів, 2832 помилкових негативних та 5519 істинних негативних. **Визначте ваші власні функції для перевірки confusion_matrix, Зверніть увагу, що тут заповнено перший елемент, а вам потрібно заповнити решту 3.**

УВАГА! При написанні ваших власних функцій в коді ТУТ І ДАЛІ замість my в ваших функціях my_confusion_matrix повинно стояти ваше прізвище англ..мовою! Наприклад:

```
def ivanov_confusion_matrix(y_true, y_pred):
```

```
def find_TP(y_true, y_pred):
    # counts the number of true positives (y_true = 1, y_pred =
    1)
    return sum((y_true == 1) & (y_pred == 1))
def find_FN(y_true, y_pred):
```

```

    # counts the number of false negatives (y_true = 1, y_pred =
0)
    return # your code here
def find_FP(y_true, y_pred):
    # counts the number of false positives (y_true = 0, y_pred =
1)
    return # your code here
def find_TN(y_true, y_pred):
    # counts the number of true negatives (y_true = 0, y_pred =
0)
    return # your code here

```

Перевірте свої результати на відповідність

```

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))

```

Давайте напишемо функцію, яка обчислить всі чотири сегменти для нас, та іншу функцію для дублювання `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 my_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]])

```

Перевірте відповідність результатів з

```

my_confusion_matrix(df.actual_label.values,
df.predicted_RF.values)

```

Замість того, щоб порівнювати вручну, давайте перевіримо, що наші функції працюють, використовуючи вбудовані `Pythonassert` і `Numpy'sarray_equal` функції

```

assert
np.array_equal(my_confusion_matrix(df.actual_label.values,


```

```
df.predicted_RF.values),
confusion_matrix(df.actual_label.values, df.predicted_RF.values)
), 'my_confusion_matrix() is not correct for RF'assert
np.array_equal(my_confusion_matrix(df.actual_label.values,
df.predicted_LR.values),confusion_matrix(df.actual_label.values,
df.predicted_LR.values) ), 'my_confusion_matrix() is not correct
for LR'
```

Зважаючи на ці чотири сегменти (TP, FP, FN, TN), ми можемо обчислити багато інших показників продуктивності.

accuracy_score

Найбільш поширеним показником для класифікації є акуратність, тобто частка вибірок, що правильно спрогнозовані, як показано нижче:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = \frac{\text{Fraction predicted correctly}}{\text{Total samples}}$$


Ми можемо отримати оцінку точності з scikit-learn, яка приймає як вхідні дані фактичні мітки та прогнозовані мітки.

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

Ваша відповідь повинна бути 0.6705165630156111


Визначте свою власну функцію, яка дублює accuracy_score, використовуючи формулу вище.

```
def my_accuracy_score(y_true, y_pred):
    # calculates the fraction of samples
    TP, FN, FP, TN = find_conf_matrix_values(y_true, y_pred)
    return # your code hereassert
my_accuracy_score(df.actual_label.values,
df.predicted_RF.values) ==
accuracy_score(df.actual_label.values, df.predicted_RF.values),
'my_accuracy_score failed on
assert my_accuracy_score(df.actual_label.values,
df.predicted_LR.values) ==
accuracy_score(df.actual_label.values, df.predicted_LR.values),
'my_accuracy_score failed on LR'
print('Accuracy RF:
%.3f'%(my_accuracy_score(df.actual_label.values,
df.predicted_RF.values)))
```

```
print('Accuracy LR:
%.3f'%(my_accuracy_score(df.actual_label.values,
df.predicted_LR.values)))
```

Використовуючи акуратність як показник продуктивності, можна зробити висновок, що модель RF є більш точною (0,67), ніж модель LR (0,62). Так що ми маємо зупинитися тут і сказати, що модель RF – найкраща модель? Ні! Акуратність не завжди є найкращою метрикою для оцінки моделей класифікації. Наприклад, припустимо, що ми намагаємося передбачити те, що відбувається лише 1 раз із 100 разів. Ми могли б побудувати модель, яка матиме точність 99%, сказавши, що подія ніколи не відбувалася. Тим не менш, ми ловимо 0% подій, які нас цікавлять. Показник 0% тут є ще одним показником продуктивності, відомий як **recall_score**:

recall_score (також відомий як чутливість) є частка позитивних подій, які ви правильно передбачили, як показано нижче:

$$\text{Recall (Sensitivity)} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{Fraction of positives predicted correctly}}{\text{Total positives}}$$


Ми можемо отримати оцінку точності з scikit-learn, яка приймає як вхідні дані фактичні мітки та прогнозовані мітки.

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

Визначте власну функцію, яка дублює recall_score, використовуючи формулу вище.


```
def my_recall_score(y_true, y_pred):
    # calculates the fraction of positive samples predicted
    correctly
    TP, FN, FP, TN = find_conf_matrix_values(y_true, y_pred)
    return # your code here
assert
my_recall_score(df.actual_label.values, df.predicted_RF.values)
== recall_score(df.actual_label.values, df.predicted_RF.values),
'my_accuracy_score failed on RF'
assert my_recall_score(df.actual_label.values,
df.predicted_LR.values) == recall_score(df.actual_label.values,
df.predicted_LR.values), 'my_accuracy_score failed on LR'
print('Recall RF: %.3f'%(my_recall_score(df.actual_label.values,
df.predicted_RF.values)))
```

```
print('Recall LR: %.3f'%(my_recall_score(df.actual_label.values,
df.predicted_LR.values))))
```

Один із способів підвищити повноту – збільшити кількість вибірок, які ви визначаєте як прогнозовані позитивні, шляхом зниження порога для прогнозованих позитивних результатів. На жаль, це також збільшить кількість хибних спрацьовувань. Інший показник продуктивності, названий точністю, враховує це.

precision_score

Точність - це частка очікуваних позитивних подій, які є позитивними, як показано нижче:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{Fraction of predicted positives that are actually positive}}{\text{Total predicted positives}}$$


Ми можемо отримати оцінку точності з scikit-learn, яка приймає як вхідні дані фактичні мітки та прогнозовані мітки.

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

Визначте власну функцію, яка дублює precision_score, використовуючи формулу вище.

```
def my_precision_score(y_true, y_pred):
    # calculates the fraction of predicted positives samples
    that are actually positive
    TP,FN,FP,TN = find_conf_matrix_values(y_true,y_pred)
    return # your code here
assert
my_precision_score(df.actual_label.values,
df.predicted_RF.values) ==
precision_score(df.actual_label.values, df.predicted_RF.values),
'my_accuracy_score failed on RF'
assert my_precision_score(df.actual_label.values,
df.predicted_LR.values) ==
precision_score(df.actual_label.values, df.predicted_LR.values),
'my_accuracy_score failed on LR'
print('Precision RF:
%.3f'%(my_precision_score(df.actual_label.values,
df.predicted_RF.values))))
```

```
print('Precision LR:
%.3f'%(my_precision_score(df.actual_label.values,
df.predicted_LR.values))))
```

В цьому випадку, схоже, що RF модель краще як за повнотою, так і по точності. Але що б ви зробили, якби одна модель краща за повнотою, а інша була точнішою. Один метод, який використовують у цьому випадку, називають рахунком F1.

f1_score. Оцінка f1 є гармонійним середнім значенням повноти та точності, з більш високою оцінкою як краща модель. Оцінка f1 розраховується за такою формулою:

$$F1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = \frac{2 * (precision * recall)}{precision + recall}$$

Ми можемо отримати оцінку f1 з scikit-learn, яка приймає як вхідні дані фактичні мітки та прогнозовані мітки

```
from sklearn.metrics import
f1_scoref1_score(df.actual_label.values, df.predicted_RF.values)
```

Визначте власну функцію, яка дублює f1_score, використовуючи формулу вище.

```
def my_f1_score(y_true, y_pred):
    # calculates the F1 score
    recall = my_recall_score(y_true,y_pred)
    precision = my_precision_score(y_true,y_pred)
    return # your code hereassert
my_f1_score(df.actual_label.values, df.predicted_RF.values) ==
f1_score(df.actual_label.values, df.predicted_RF.values),
'my_accuracy_score failed on RF'
assert my_f1_score(df.actual_label.values,
df.predicted_LR.values) == f1_score(df.actual_label.values,
df.predicted_LR.values), 'my_accuracy_score failed on LR'
print('F1 RF: %.3f'%(my_f1_score(df.actual_label.values,
df.predicted_RF.values))))
print('F1 LR: %.3f'%(my_f1_score(df.actual_label.values,
df.predicted_LR.values))))
```

До цих пір ми припускали, що ми визначили поріг 0,5 для вибору зразків, які прогнозуються як позитивні. Якщо ми змінимо цей поріг, показники продуктивності зміняться. Як показано нижче:

```
print('scores with threshold = 0.5')
```

```

print('Accuracy RF:
%.3f'%(my_accuracy_score(df.actual_label.values,
df.predicted_RF.values)))
print('Recall RF: %.3f'%(my_recall_score(df.actual_label.values,
df.predicted_RF.values)))
print('Precision RF:
%.3f'%(my_precision_score(df.actual_label.values,
df.predicted_RF.values)))
print('F1 RF: %.3f'%(my_f1_score(df.actual_label.values,
df.predicted_RF.values)))
print('')
print('scores with threshold = 0.25')
print('Accuracy RF:
%.3f'%(my_accuracy_score(df.actual_label.values, (df.model_RF >=
0.25).astype('int').values)))
print('Recall RF: %.3f'%(my_recall_score(df.actual_label.values,
(df.model_RF >= 0.25).astype('int').values)))
print('Precision RF:
%.3f'%(my_precision_score(df.actual_label.values, (df.model_RF
>= 0.25).astype('int').values)))
print('F1 RF: %.3f'%(my_f1_score(df.actual_label.values,
(df.model_RF >= 0.25).astype('int').values)))

```

Порівняйте результати для різних порогів та зробіть висновки.

Як же оцінювати модель, якщо ми не вибрали поріг? Одним із найпоширеніших методів є використання кривої робочих характеристик приймача (ROC).

roc_curve та roc_auc_score

Криві ROC ДУЖЕ допомагають зрозуміти баланс між істинно позитивними показниками та хибнопозитивними показниками. Sci-kit learn має вбудовані функції для кривих ROC та їх аналізу. Входи в ці функції (roc_curve та roc_auc_score) фактичні мітки та прогнозовані ймовірності (не прогнозовані мітки). І та й інша roc_curve, а також roc_auc_score обидві складні функції, тому ми не будемо писати ці функції з нуля. Натомість ми покажемо, як використовувати функції Sci-Kit Learn і пояснимо ключові моменти. Давайте почнемо з використання roc_curve.

```

from sklearn.metrics import roc_curve
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)

```

Функція roc_curve повертає три списки:

пороги = всі унікальні ймовірності передбачення в порядку спадання

fpr = хибнопозитивний показник ($FP/(FP+TN)$) для кожного порогу

tpr = істинно позитивна швидкість ($TP/(TP+FN)$) для кожного порогу.

```
thresholds_RF
array([0.93052053, 0.82363091, 0.82354671, ..., 0.25654616, 0.25587275,
       0.17142947])
```

```
fpr_RF
array([0.          , 0.          , 0.          , ..., 0.9941617, 0.9941617,
       1.          ])
```

```
tpr_RF
array([1.26919660e-04, 5.33062571e-03, 5.58446503e-03, ...,
       9.99873080e-01, 1.00000000e+00, 1.00000000e+00])
```

Побудуйте криву ROC для кожної моделі, як показано нижче.

```
import matplotlib.pyplot as pltplt.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')
plt.show()
```

Рисунок занесіть у звіт

Є кілька речей, які ми можемо спостерігати із цієї фігури:

- модель, яка випадково вгадує мітку, призведе до чорної лінії, і нам необхідно мати модель із кривою над цією чорною лінією.
- ROC- крива, яка знаходиться далі від чорної лінії, краще, тому RF (червона) виглядає краще, ніж LR (синя)
- Хоча це не видно безпосередньо, високий поріг призводить до точки у лівому нижньому кутку, а низький поріг призводить до точки у верхньому правому куті. Це означає, що, зменшуючи поріг, ви отримуєте вищий TPR за рахунок вищого FPR.

Для аналізу продуктивності ми використовуватимемо метрику площі під кривою.

```
from sklearn.metrics import roc_auc_scoreauc_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)
```

Як ви зможете побачити, площа під кривою для моделі RF (AUC = 0,738) краще, ніж LR (AUC = 0,666). Коли будете криву ROC, доцільно додавати AUC до легенди, як показано нижче.

```
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()
```

Рисунок занесіть у звіт. Зробіть висновки яка з двох моделей (RF та LR) краща і чому. Висновки занесіть у звіт.

У прогнозуючій аналітиці при виборі між двома чи декількома моделями важливо вибрати одну метрику продуктивності. Ви можете вибирати з множини (акуратність, повнота, точність, f1-оцінка, AUC тощо). Зрештою, ви повинні використовувати метрику продуктивності, яка найбільше підходить для аналізованого бізнес-завдання. Багато дослідників даних вважають за краще використовувати AUC для аналізу продуктивності кожної моделі, оскільки він не вимагає вибору порога і допомагає збалансувати істинний позитивний показник і хибнопозитивний показник.

Програмний код збережіть під назвою LR_1_task_5.py

Завдання 2.6. Розробіть програму класифікації даних в файлі data_multivar_nb.txt за допомогою машини опорних векторів (Support Vector Machine - SVM). Розрахуйте показники якості класифікації. Порівняйте їх з показниками найвісного байєсівського класифікатора. Зробіть висновки яку модель класифікації краще обрати і чому.

Код програми та результати занесіть у звіт.

Програмний код збережіть під назвою LR_1_task_6.py

Коди комітати на GitHub. У кожному звіті повинно бути посилання на GitHub.

**Назвіть бланк звіту СШІ-ЛР-1-NNN-XXXXX.doc
де NNN – позначення групи
XXXXX – позначення прізвища студента.**

Переконвертуйте файл звіту в СШІ-ЛР-1-NNN-XXXXX.pdf