Сумський державний університет

Кафедра

Прикладної математики та моделювання складних систем

Курсова робота

Дисципліна

Машинне навчання

Студентка: Пороскун О. О.

Група: ПМ.м-21

Викладач: Харченко В. О.

Суми, Сумська область

2023

**ЗМІСТ**

[**Вступ** 3](#_Toc136792803)

[**Постановка задачі** 4](#_Toc136792804)

[**Теоретична частина** 5](#_Toc136792805)

[*Класифікація методом дерева прийняття рішень (decision tree)* 5](#_Toc136792806)

[*Класифікація методом k-найближчих сусiдiв (kNN - k-Nearest Neighbors)* 6](#_Toc136792807)

[*Класифікація методом опорних векторів (SVM)* 9](#_Toc136792808)

[*Зменшення розмiрностi методом головних компонент (PCA)* 11](#_Toc136792809)

[*Зменшення розмiрностi методом сингулярного розкладання (SVD)* 15](#_Toc136792810)

[*Кластеризацiя методом k-середнiх (k-Means)* 16](#_Toc136792811)

[*Кластеризацiя методом DBscan* 21](#_Toc136792812)

[*Пошук асоціативних правил з використанням методу Apriori* 24](#_Toc136792813)

[**Програмна реалізація** 28](#_Toc136792814)

[Метод дерева прийняття рішень (decision tree) 28](#_Toc136792815)

[Метод класифікації kNN 33](#_Toc136792816)

[Метод опорних векторів (SVM) 40](#_Toc136792817)

[Метод головних компонент (PCA) 51](#_Toc136792818)

[Редукування даних: метод SVD 60](#_Toc136792819)

[Метод k-середнiх (k-Means) 65](#_Toc136792820)

[Метод DBSCAN 79](#_Toc136792821)

[Асоціації та пошук правил: метод Apriori 86](#_Toc136792822)

[**Висновки** 92](#_Toc136792823)

# **Вступ**

Перш ніж говорити про навчання, почнемо з аналізу термінології.

Data Science - це загальне найменування дисциплін з вивчення даних, Machine Learning - це підрозділ Data Science, який займається розбудовою розумних моделей. Такі моделі можуть використовуватися для передбачення покупки товару користувачем, рекомендацій у соцмережах (рекомендаційні системи), розпізнавання зображень тощо.

Data Science спеціалісти займаються дослідженнями. В іноземних компаніях такій посаді відповідають позиціїresearch-інженерів — це переважно математики, які працюють з теоретичною частиною алгоритмів та досліджують різноманітні закономірності.

Machine Learning інженери, своєю чергою, займаються побудовою моделей з урахуванням отриманих даних. Але такий поділ існує лише теоретично чи лише деяких країнах.

В Україні Data Science та Machine Learning раніше використовувалися як слова-синоніми, зараз ці поняття вже починають розділяти. У наших реаліях вакансії, де необхідне знання Machine Learning, найчастіше називаються Data Scientist і навпаки. Тому, якщо ви хочете працювати з даними, вам слід вивчити і те, й інше.

*Класи застосування машинного навчання(МН)*

* Навчання з учителем (Supervised learning). Ми маємо чимало прикладів і правильні, еталонні, відповіді до кожного з них. Наприклад, історичні дані успішности студентів з минулих років.
* Навчання без учителя (Unsupervised learning). Це знаходження сенсу в даних без наявности конкретної, правильної, відповіді. Інакше кажучи, ми не знаємо, що шукаємо, але дуже сильно хочемо знайти.
* Навчання із закріпленням (Reinforcement learning). Це розширений варіант навчання з учителем, коли замість еталонних відповідей програма наприкінці отримує зворотний зв’язок. Наприклад, якщо ми вчимо машину грати в шахи, оцінка кожного можливого ходу є надто затратною й не завжди правильною, але наприкінці гри дуже легко визначити виграш або програш алгоритму.
* Перенесення навчання (Transfer learning). Навчаємо модель для однієї проблеми й використовуємо результат навчання для модифікації даних від іншої. Наприклад, фільтр, за допомогою якого ви робите селфі подібними до творінь Вінсента ван Гога та який домальовує заячі вуха у відеоконференції або змушує Обаму вимовляти ваші слова його ж голосом і з його мімікою.

Мета машиного навчання — передбачити результат за вхідними даними.

# **Постановка задачі**

Потрібно створити алгоритми або використати вбудовані функції на мові програмування Python за 8 темами машинного навчання:

1. класифікація методом дерева прийняття рішень (decision tree),
2. класифікація методом k-найближчих сусiдiв (kNN - k-Nearest Neighbors),
3. класифікація методом опорних векторів (SVM),
4. зменшення розмiрностi методом головних компонент (PCA),
5. зменшення розмiрностi методом сингулярного розкладання (SVD),
6. кластеризацiя методом k-середнiх (k-Means),
7. кластеризацiя методом DBscan,
8. пошук асоціативних правил з використанням методу Apriori.

# **Теоретична частина**

## *Класифікація методом дерева прийняття рішень (decision tree)*

Дерева рiшень Decision Tree є одним iз найбiльш ефективних iнструментiв iнтелектуального аналiзу даних та передбачуваної аналiтики, якi дозволяють вирiшувати завдання класифiкацiї та регресiї.

Оскільки правила в деревах рiшень виходять шляхом узагальнення безлічі окремих спостережень (навчальних прикладів), що описують предметну область, то за аналогією з відповідним методом логічного виведення їх називають індуктивними правилами, а процес навчання – iндукцiєю дерев рiшень.

У навчальнiй множинi для прикладiв має бути задане цiльове значення, оскiльки дерева рiшень є моделями, що будуються на основi навчання з учителем. При цьому якщо цiльова змiнна дискретна (мiтка класу), то модель називають деревом класифiкацiї, а якщо безперервна, то деревом регресiї.

*Структура дерева рiшень*

Введемо до розгляду основнi поняття, що використовуються в теорiї дерев рiшень.

Табл. 1 Основні поняття в теорiї дерев рiшень

|  |  |
| --- | --- |
| Назва | Опис |
| Об’єкт | Приклад, шаблон, спостереження |
| Атрибут | Ознака, незалежна змiнна, властивiсть |
| Цiльова змiнна | Залежна змiнна, мiтка класу |
| Вузол | Внутрiшнiй вузол дерева, вузол перевiрки |
| Кореневий вузол | Початковий вузол дерева рiшень |
| Лист | Кiнцевий вузол дерева, вузол рiшення |
| Вирiшальне правило | Умова у вузлi, перевiрка |

Власне, саме дерево рiшень — метод подання вирiшальних правил у iєрархiчнiй структурi, що складається з елементiв двох типiв – вузлiв (node) та листя (leaf). У вузлах знаходяться вирiшальнi правила i проводиться перевiрка вiдповiдностi прикладiв цього правила за яким-небудь атрибутом навчальної множини.

У найпростiшому випадку, в результатi перевiрки, множина прикладiв, що потрапили у вузол, розбивається на двi пiдмножини, в одну з яких потрапляють приклади, що задовольняють правило, а в iнше не задовольняють. Потiм до кожної пiдмножини знову застосовується правило i процедура рекурсивно повторюється доки буде досягнуто деяка умова зупинки алгоритму. В результатi в останньому вузлi перевiрка та розбиття не проводиться i вiн оголошується листом. Лист визначає рiшення для кожного прикладу, що в нього потрапив. Для дерева класифiкацiї це клас, що асоцiюється з вузлом, а для дерева регресiї модальний iнтервал цiльової змiнної, що вiдповiдає листу.

Таким чином, на вiдмiну вiд вузла, у листi мiститься не правило, а пiдмножина об’єктiв, що задовольняють всi правила гiлки, яка закiнчується цим листом. Очевидно, щоб потрапити в лист, приклад повинен задовольняти всi правила, що лежать на шляху до цього листа. Оскiльки шлях у деревi до кожного листа єдиний, то й кожен приклад може потрапити лише в один лист, що забезпечує єдинiсть рiшення.

*Основнi етапи побудови*

У ходi побудови дерева рiшень необхiдно вирiшити кiлька основних проблем, з кожною з яких пов’язаний вiдповiдний крок процесу навчання:

* Вибiр атрибута, за яким буде проводитися розбиття в даному вузлi (атрибута розбиття).
* Вибiр критерiю зупинення навчання.
* Вибiр методу вiдсiкання гiлок (спрощення).
* Оцiнка точностi збудованого дерева.

## *Класифікація методом k-найближчих сусiдiв (kNN - k-Nearest Neighbors)*

Типовим представником методiв класифiкацiї є метод k-найближчих сусiдiв k-nearest neighbors algorithm (KNN). Вiн використовує просту логiку: об’єкт вiдноситься до того ж класу, що бiльшiсть його найближчих сусiдiв.

Метод належить до класу непараметричних, тобто не вимагає припущень про те, з якого статистичного розподілу була сформована навчальна множина. Отже, класифікацiйні моделі, побудовані за допомогою методу KNN, також будуть непараметричними. Це означає, що структура моделi не визначається жорстко спочатку, а визначається даними.

Оскiльки ознаки, на основi яких проводиться класифiкацiя, можуть мати рiзну фiзичну природу i, вiдповiдно, дiапазони значень, для покращення результатiв класифiкацiї буде корисно виконати нормалiзацiю навчальних даних.

*Алгоритм*

Нехай є набiр даних, що складається з n спостережень Xi(i = 1, . . . , n) для кожного з яких заданий клас Cj (j = 1, . . . , m). Тодi на його основi може бути сформована навчальна множина, всi приклади якої являють собою пари Xi, Cj .

Алгоритм KNN можна роздiлити на двi простi фази: навчання та класифiкацiї. Пiд час навчання алгоритм просто запам’ятовує вектори ознак спостережень та його мiтки класiв (тобто приклади). Також задається параметр алгоритму k, який задає число “сусiдiв”, якi будуть використовуватися при класифiкацiї.

На фазi класифiкацiї пред’являється новий об’єкт, котрого мiтка класу не задана. Для нього визначаються k найближчих (у сенсi деякої метрики) попередньо класифiкованих спостережень. Потiм вибирається клас, якому належить бiльшiсть з k найближчих прикладiв-сусiдiв, i до цього ж класу вiдноситься об’єкт, що класифiкується. Пояснимо роботу алгоритму за допомогою рисунка.

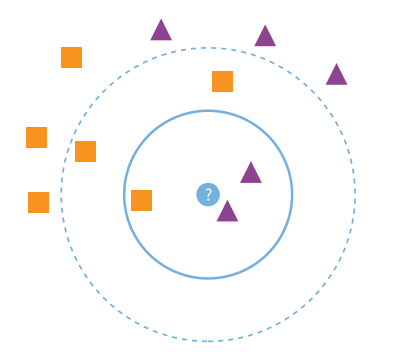


Рис. 1 Робота алгоритму KNN

Кружком представлений об’єкт, який потрiбно класифiкувати, вiднiсши до одного з двох класiв «трикутники» та «квадрати».

Якщо вибрати k = 3, то з трьох найближчих об’єктiв виявляться два «трикутниками» i один «квадратом». Вiдтак новому об’єкту буде надано клас «трикутник».

Якщо поставити k = 5, то з п’яти «сусiдiв» два будуть «трикутники» i три «квадрати», в результатi об’єкт, що класифiкується, буде розпiзнаний як «квадрат».

*Визначення класу нового об’єкту*

У найпростiшому випадку клас нового об’єкта може бути визначений простим вибором класу, що найчастiше зустрiчається серед k прикладiв. При цьому розраховується вiдстань вiд даного об’єкта до кожного об’єкта з навчальної вибiрки за формулою

де – i-та ознака об’єкта, що класифiкується – ознака класифікованих об’єктів.

Однак на практицi це не завжди вдале рiшення, наприклад, якщо частота появи для двох або бiльше класiв виявляється однаковою. I тут використовують деяку функцiю, з допомогою якої визначається клас, звану функцiєю поєднання (combination function). У звичайному випадку використовують так зване просте невиважене голосування (simple unweighted voting). При цьому передбачається, що всi k прикладiв мають однакове право «голосу» незалежно вiд вiдстань до об’єкта, що класифiкується. Однак, логiчно припустити, що чим далi приклад розташований вiд об’єкта, що класифiкується, в просторi ознак, тим нижче йозначимiсть для визначення класу. Тому для покращення результатiв класифiкацiї вводять зважування прикладiв залежно вiд їхньої вiддаленостi. I тут використовують зважене голосування (weighted voting).

В основi iдеї зваженого голосування лежить введення «штрафу» для класу, залежно вiд того, наскiльки приклади, що вiдносяться до нього, вiддаленi вiд класифiкованого об’єкта. Такий “штраф” представляється як сума величин, обернених квадрату вiдстаней вiд прикладу j-го класу до класифiкованого об’єкта (часто це значення називають показником близькостi):

де D – оператор обчислення вiдстанi, x – вектор ознак об’єкта, що класифiкується, aij – i-й приклад j-го класу.

Таким чином, «перемагає» той клас, для якого величина Qj виявиться найбiльшою. При цьому також знижується ймовiрнiсть того, що класи отримають однакову кiлькiсть голосiв.

*Областi застосування алгоритму*

Алгоритм KNN може застосовуватися практично у всiх завданнях

класифiкацiї, особливо у випадках, коли оцiнити параметри ймовiрнiсного розподiлу даних складно чи неможливо. Найбiльш типовими програмами алгоритму KNN є:

• класифiкацiя клiєнтiв (наприклад, за рiвнем лояльностi );

• медицина – класифiкацiя пацiєнтiв за медичними показниками;

• маркетинг – класифiкацiя товарiв за рiвнем популярностi i т. д.

*Переваги та недолiки алгоритму*

До *переваг* алгоритму можна вiднести.

• стiйкiсть до викидiв i аномальних значень, оскiльки ймовiрнiсть попадання записiв, що мiстять їх, до k-найближчих сусiдiв мала. Якщо ж це сталося, то вплив на голосування (особливо зважене) також, швидше за все, буде незначним, а отже, малим буде i вплив на результати класифiкацiї;

• програмна реалiзацiя алгоритму доволi проста;

• результати роботи алгоритму легко пiддаються iнтерпретацiї. Логiка роботи алгоритму зрозумiла експертам у рiзних галузях.

До *недолiкiв* алгоритму KNN можна вiднести:

• даний метод не створює будь-яких моделей, якi узагальнюють попереднiй досвiд, а iнтерес можуть становити й самi правила класифiкацiї;

• при класифiкацiї об’єкта використовуються всi доступнi данi, тому метод KNN є досить витратним у планi, особливо у разi великих обсягiв даних;

• висока трудомiсткiсть через необхiднiсть обчислення вiдстаней до всiх прикладiв;

• пiдвищенi вимоги до репрезентативностi вихiдних даних

## *Класифікація методом опорних векторів (SVM)*

Розглянемо метод опорних векторiв (англ. SVM, Support Vector Machine) для задач класифiкацiї. Буде представлено основну iдею алгоритму, виведення налаштування його ваг i розiбрано просту реалiзацiю своїми руками. На прикладi датасету Iris буде продемонстровано роботу написаного алгоритму з лiнiйно роздiленими-/нероздiлними даними у просторi R2. Додатково будуть озвученi плюси та мiнуси алгоритму, його модифiкацiї.

Завдання

Вирiшуватимемо завдання бiнарної (коли класiв всього два) класифiкацiї. Спочатку алгоритм тренується на об’єктах з навчальної вибiрки, котрим заздалегiдь вiдомi мiтки класiв. Далi вже навчений алгоритм передбачає мiтку класу для кожного об’єкта з вiдкладеної/тестової вибiрки. Мiтки класiв можуть набувати значень Y = {−1, +1}. Об’єкт – вектор з N ознаками X = (x1, x2, . . . , xn) у просторi Rn. При навчаннi алгоритм повинен побудувати функцiю F(X) = Y , яка приймає аргумент X – об’єкт з простору Rn i видає мiтку класу Y . Головна мета SVM як класифiкатора – знайти рiвняння роздiльної гiперплощини

w1x1 + w2x2 + . . . + wnxn + w0 = 0

в просторi Rn, яка роздiлила б два класи якимось оптимальним чином.

*Загальнi вiдомостi про алгоритм*

Для простоти розглянемо вибiрку об’єктiв з двома ознаками x1 та x2 якi належать до двох рiзних класiв. Iлюстрацiю наведено на рисунку 3.1. Загальний вид перетворення F об’єкта X на мiтку класуY :

F(X) = sign(wTX − b), де w = (w1, w2, . . . , wn), b = −w0.

Пiсля налаштування ваг алгоритму w та b (навчання), всi об’єкти, що потрапляють по одну сторону вiд побудованої гiперплощини, передбачатимуться як перший клас, а об’єкти, що потрапляють по iнший бiк – другий клас.

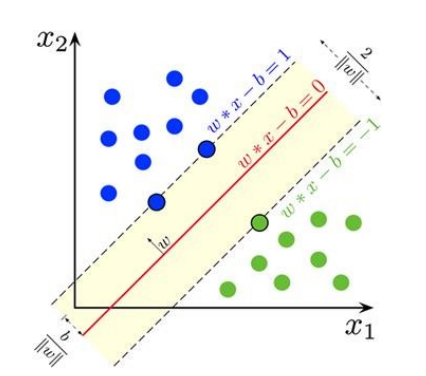


Рис. 2 Робота алгоритму SVM

Синi кружки – об’єкти першого класу; зеленi – об’єкти другого класу; кружки з чорним контуром – опорнi об’єкти; пряма оптимально роздiляє два класи; прямi , якi проходять через опорнi вектори x+ та x− визначають ширину роздiляючої полоси, як проекцiю вектора (x+−x−) на вектор нормалi до роздiляючої прямої (гiперплощини) . Усерединi функцiї стоїть лiнiйна комбiнацiя ознак об’єкта з вагами алгоритму, саме тому SVM вiдноситься до лiнiйних алгоритмiв. Роздiляючу гiперплощину можна побудувати рiзними способами, але в SVM ваги w i b налаштовуються таким чином, щоб об’єкти класiв лежали якнайдалi вiд роздiльної гiперплощини. Iншими словами, алгоритм максимiзує зазор (англ. margin) мiж гiперплощиною та об’єктами класiв, якi розташованi найближче до неї – опорнi вектори x+ та x−.

*Плюси, мiнуси та модифiкацiї*

Плюси:

* добре працює iз простором ознак великого розмiру;
* добре працює з даними невеликого обсягу;
* алгоритм максимiзує смугу, що роздiляє, яка, як подушка безпеки, дозволяє зменшити кiлькiсть помилок класифiкацiї;
* оскiльки алгоритм зводиться до розв’язання задачi квадратичного програмування у опуклiй областi, то таке завдання завжди має єдине рiшення (роздiльна гiперплощина з певними гiперпараметрами алгоритму завжди одна).

Мiнуси:

* довгий час навчання (для великих наборiв даних);
* нестiйкiсть до шуму: викиди у навчальних даних стають опорними об’єктами-порушниками i безпосередньо впливають на побудову роздiльної гiперплощини;
* не описанi загальнi методи побудови ядер i спрямовуючих просторiв, що найбiльш пiдходять для конкретного завдання у разi лiнiйної нероздiльностi класiв. Пiдбирати кориснi перетворення даних – мистецтво.

Модифiкацiї алгоритму:

* Метод релевантних векторiв (Revance Vector Machine, RVM)
* 1-norm SVM (LASSO SVM)
* Doubly Regularized SVM (ElasticNet SVM)
* Support Features Machine (SFM)
* Relevance Features Machine (RFM).

## *Зменшення розмiрностi методом головних компонент (PCA)*

В аналiзi даних, як i в будь-якому iншому аналiзi, часом буває незайвим створити спрощену модель, що максимально точно описує реальний стан справ. Часто буває так, що ознаки досить сильно залежать одна вiд одної та їх одночасна наявнiсть є надмiрною. Якщо одна ознака строго залежить вiд iншої, то знаючи одну, ми завжди знаємо й iншу. Але набагато частiше буває так, що ознаки залежать одна вiд одної не так строго i (що важливо!) не так очевидно. Але як з’ясувати, який саме набiр параметрiв добре описує наш набiр даних, але при цьому має невелику надмiрнiсть? Iншими словами, як зменшити розмiрнiсть простору, в якому “живуть” данi, втративши при цьому мiнiмум iнформацiї?

Способи вирiшення цього завдання називаються методами зменшення розмiрностi (dimensionality reduction). Метод головних компонент (principal components analysis, PCA) – один з них. Мета PCA – вилучення з цих даних потрiбної iнформацiї. Що є iнформацiєю, залежить вiд сутi завдання, що розв’язується. Данi можуть мiстити потрiбну нам iнформацiю, вони можуть бути надлишковими. Однак, у деяких випадках, iнформацiї в даних може не бути зовсiм. Розмiрнiсть даних — кiлькiсть зразкiв та змiнних -– має велике значення для успiшного видобутку iнформацiї. Зайвих даних немає — краще, коли їх багато, нiж мало.

Данi завжди (або майже завжди) мiстять у собi небажану складову, яка називається шумом. Природа цього шуму може бути рiзною, але, у багатьох випадках, шум – це та частина даних, яка не мiстить iнформацiї, що шукається. Що вважати шумом, а що iнформацiєю, завжди вирiшується з урахуванням поставлених цiлей та методiв, якi використовуються для її досягнення.

Шум i надмiрнiсть у даних обов’язково виявляють себе через кореляцiйнi зв’язки мiж змiнними. Похибки даних можуть призвести до появи не систематичних, а випадкових зв’язкiв мiж змiнними. Поняття ефективного рангу та прихованих, латентних змiнних, кiлькiсть яких дорiвнює цьому рангу, є найважливiшим поняттям у PCA.

Розглянемо простiший випадок, коли об’єкти описуються лише двома ознаками: x1 та x2. Такi данi легко зобразити на площинi (див. рисунок 3).

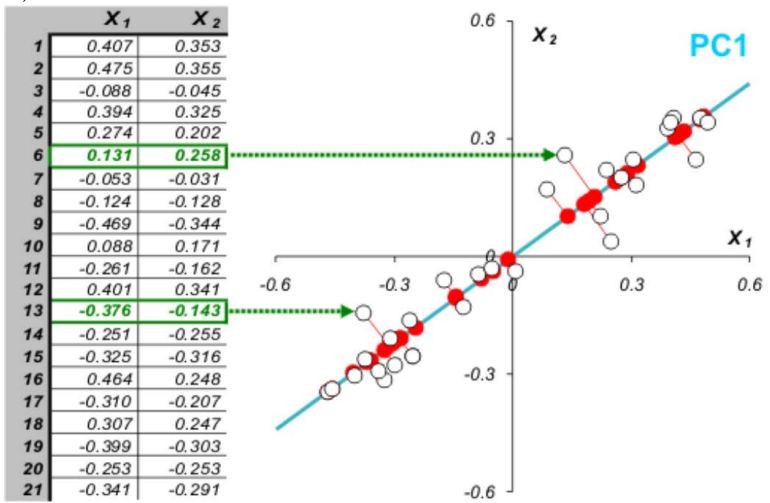


Рис. 3

Кожному рядку вихiдної таблицi (тобто зразку) вiдповiдає точка на площинi з вiдповiдними координатами. Вони позначенi порожнiми кружками. Проведемо через них пряму, так, щоб уздовж неї вiдбувалася максимальна змiна даних. На рисунку ця пряма видiлена блакитним кольором; вона називається першою головною компонентою PC1. Потiм проектуємо всi вихiднi точки на цю вiсь (червонi кружки). Якщо данi описанi не повнiстю (шум великий), то вибирається ще один напрямок (PC2) – перпендикулярне до першого, так щоб описати змiну, що залишилася в даних i т.д.

Таким чином, знаючи залежностi та їх силу, ми можемо виразити кiлька ознак через одну, злити докупи, так би мовити, i працювати вже з бiльш простою моделлю. Звичайно, уникнути втрат iнформацiї, швидше за все, не вдасться, але мiнiмiзувати її нам допоможе якраз метод PCA.

Отже, даний метод апроксимує n-розмiрну хмару спостережень до елiпсоїда (теж n-вимiрного), пiвосi якого i будуть майбутнiми головними компонентами. I при проекцiї даних на такi осi (зниженнi розмiрностi) зберiгається найбiльше iнформацiї.

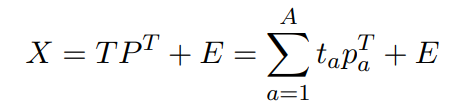
Зменшення розмiрностi даних

Метод основних компонентiв застосовується до даних, записаним у виглядi матрицi X -– прямокутної таблицi чисел розмiрнiстю I рядкiв i J стовпцiв. Традицiйно рядки цiєї матрицi називаються об’єктами. Вони нумеруються iндексом i = 1, . . . , I. Стовпцi називаються ознаками i вони нумеруються iндексом j = 1, . . . , J. У методi головних компонент використовуються новi, формальнi змiннi

ta (a = 1, . . . , A), що є лiнiйною комбiнацiєю вихiдних змiнних xj (j = 1, . . . , J):



За допомогою цих нових змiнних матриця X розкладається у добуток двох матриць T та P:



Матриця T називається матрицею рахункiв (scores). Її розмiрнiсть (I × A). Матриця P називається матрицею навантажень (loadings). Її розмiрнiсть (J × A). E – матриця залишкiв, розмiрнiстю (I × J).

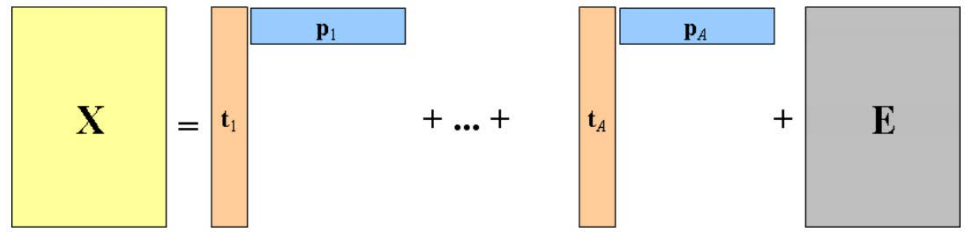


Рис. 4

Новi змiннi ta називаються головними компонентами (Principal Components), тому сам метод називається методом головних компонент (PCA). Число стовпцiв ta в матрицi T, i pa в матрицi P дорiвнює A, яке називається числом головних компонент (PC). Ця величина свiдомо менша вiд числа змiнних J i числа зразкiв I.

Важливою властивiстю PCA є ортогональнiсть (незалежнiсть) основних компонент. Тому матриця рахункiв T не перебудовується зi збiльшенням числа компонент, а до неї просто додається ще один стовпець – вiдповiдний новому напрямку. Теж вiдбувається i з матрицею навантажень P.

*Статистичнi основи*

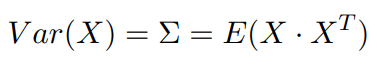
Для опису випадкової величини використовуються моменти. Потрiбнi нам – математичне очiкування та дисперсiя. Можна смiливо сказати, що математичне очiкування це “центр тяжiння” величини, а дисперсiя – її “розмiри” (розкид). Сам процес проектування на вектор нiяк не впливає на значення середнiх, тому що для мiнiмiзацiї втрат iнформацiї наш вектор має проходити через центр нашої вибiрки. Тому зазвичай проводять центрування вибiрки -– лiнiйно змiщуючи її так, щоб середнi значення ознак дорiвнювали 0. Оператор, зворотний зсуву дорiвнюватиме вектору початкових середнiх значень — вiн знадобиться вiдновлення вибiрки у вихiднiй розмiрностi. При цьому, дисперсiя залежить вiд порядкiв значень випадкової величини, тобто, є чутливою до масштабування.

Для опису форми випадкового вектора необхiдна матриця коварiацiї. У коварiацiйнiй матрицi по дiагоналi будуть дисперсiї ознак (оскiльки i = j), а в iнших комiрках — коварiацiї вiдповiдних пар ознак. А в силу симетричностi коварiацiї матриця теж буде симетрична. Коварiацiйна матриця є узагальненням дисперсiї на випадок багатовимiрних випадкових величин вона так само описує форму (розкид) випадкової величини, як i дисперсiя, яка для одновимiрної випадкової величини має вигляд матрицi розмiру 1×1, в якiй її єдиний член заданий формулою . Коварiацiйна матриця описує форму нашої випадкової величини (ознаки). Узагальнення дисперсiї на вищi розмiрностi – матриця коварiацiї, i цi два поняття еквiвалентнi. При проекцiї на вектор максимiзується дисперсiя проекцiї, при проекцiї на простiр великих порядкiв – вся її коварiацiйна матриця.

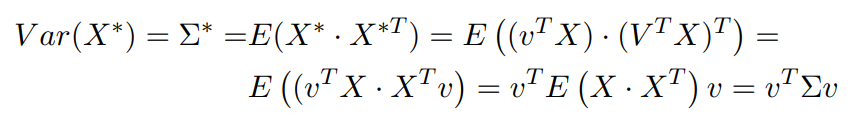
*Власнi значення та власнi вектори*

Тепер треба знайти такi вектори, при якому б максимiзувався розкид (дисперсiя) проекцiї вибiрки на них. Слiд зазначити, що узагальнення дисперсiї на вищi розмiрностi – матриця коварiацiй, i цi два поняття еквiвалентнi. При проекцiї на вектор максимiзується дисперсiя проекцiї, при проекцiї простору великих порядкiв – вся її коварiацiйна матриця.

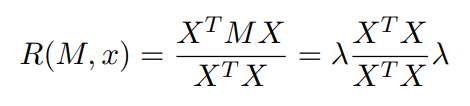
Отже, вiзьмемо одиничний вектор на який проектуватимемо наш випадковий вектор X. Тодi проекцiя на нього дорiвнюватиме vTX (vT – транспонування вектора v). Дисперсiя проекцiї на вектор вiдповiдно дорiвнює Var(vTX). Загалом у векторнiй формi (для центрованих величин) дисперсiя виражається так:



Вiдповiдно, дисперсiя проекцiї:



Таким чином, що дисперсiя максимiзується за умови максимального значення . Далi зручно використати спiввiдношення Релея, якi для коварiацiйних матриць M мають спецiальний випадок:



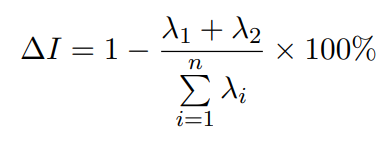
i вiдповiдно маємо



Остання формула є розкладанням матрицi на власнi вектори та значення: X є власним вектором, а λ – власним значенням. Кiлькiсть власних векторiв та значень дорiвнюють розмiру матрицi (i значення можуть повторюватися).

Таким чином, приходимо до висновку, що напрямок максимальної дисперсiї у проекцiї завжди збiгається з власним вектором, що має максимальне власне значення, яке дорiвнює величинi цiєї дисперсiї. Це справедливо також для проекцiй на бiльшу кiлькiсть вимiрювань – дисперсiя (коварiацiйна матриця) проекцiї на m-вимiрний простiр буде максимальною у напрямку m власних векторiв, що мають максимальнi власнi значення.

Розраховуємо кiлькiсть втраченої iнформацiї використовуючи власнi значення матрицi коварiацiй при зменшеннi розмiрностi з n ознак до двох:



## *Зменшення розмiрностi методом сингулярного розкладання (SVD)*

Сингулярним розкладанням будь-якої матриці 𝑋 розміру 𝑛 × 𝑑 є її подання у вигляді добутку трьох матриць: 𝑈, Σ, 𝑉:



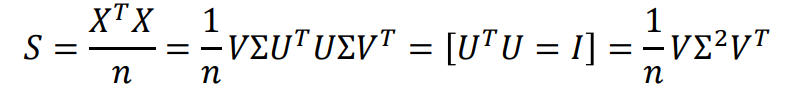
Є обмеження на 𝑈, Σ, 𝑉:

* 𝑈 має розміри 𝑛×𝑛, Σ має розмір 𝑛×𝑑, а 𝑉 має розмір 𝑑 × 𝑑.
* 𝑈 і 𝑉 – ортогональні матриці. Тобто 𝑈𝑇𝑈 = 𝐼 і 𝑉T𝑉 = 𝐼.
* Σ є діагональною матрицею: Σ = 𝑑𝑖𝑎𝑔(𝜎1, … , 𝜎𝑑) – сингулярні значення матриці 𝑋. Тобто всі елементи в Σ дорівнюють нулю, якщо вони не лежать на діагоналі. Крім того, діагональні елементи в Σ розташовані від найбільшого до найменшого.

Стовбці матриці 𝑈 – ліві сингулярні вектори. Стовбці матриці 𝑉 – праві сингулярні вектори.

Встановимо, як значення 𝜎1, … , 𝜎𝑑 діагональної матриці залежать від даних

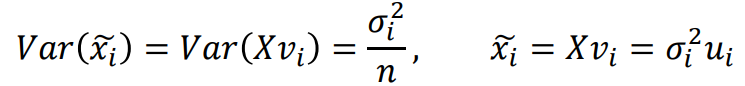
Знайдемо матрицю коваріації



Таким чином:

* Стовбці 𝑣1, … , 𝑣𝑑 матриці 𝑉 являють собою власний базис матриці коваріацій 𝑆;
* – власні значення матриці коваріацій 𝑆;
* Вектори 𝑣1, … , 𝑣𝑑 – головні (основні) компоненти (principal components) для матриці даних.

Для будь-якого вектору маємо



Властивість матриці Σ:



Отже:

* Перша головна компонента 𝑢1 характеризується тією властивістю, що має максимальну дисперсію серед всіх нормованих лінійних комбінацій стовбців матриці 𝑋;
* має мінімальну дисперсію.

Одна дуже корисна властивість SVD полягає в тому, що він завжди знаходить оптимальний набір факторів, які найкраще передбачають оцінки, відповідно до стандартної міри подібності до матриць (нормою Фробеніуса). Тобто якщо ми використовуємо SVD для пошуку факторів матриці, це найкращі фактори, які можна знайти. Ця властивість оптимальності означає, що нам не потрібно ставити питання, чи може інший набір чисел краще передбачати результати.

## *Кластеризацiя методом k-середнiх (k-Means)*

Алгоритми кластеризацiї дозволяють роздiлити об’єкти у випадку, коли класи заздалегiдь не зазначенi, а кластери мають бути сформованi за схожiстю елементiв. Тобто відбувається реалізація навчання без учителя.

Вiдмiнний приклад кластеризацiї — маркери на картах в iнтернетi. Коли ви шукаєте всi ресторани азiатської кухнi у великому мiстi, машинi доводиться групувати їх у кружечки з цифрою, iнакше браузер зависне в потугах намалювати мiльйон маркерiв.

Бiльш складнi приклади кластеризацiї можна згадати у програмах iPhoto або Google Photos, якi знаходять обличчя людей на фотографiях та групують їх у альбоми. Програма не знає як звуть ваших друзiв, але може вiдрiзнити їх за характерними рисами обличчя.

Кластеризацiя сьогоднi використовують для:

* Сегментацiя ринку (типiв покупцiв)
* Об’єднання близьких точок на картi
* Стиснення зображень
* Аналiз та розмiтки нових даних.
* Детектори аномальної поведiнки

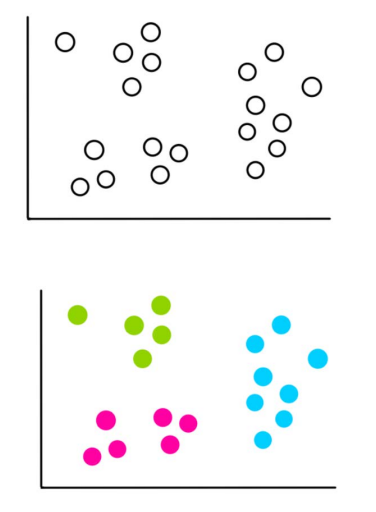
Популярнi алгоритми: Метод K-середнiх (K-means), Mean-Shift, DBSCAN.

*Метод k-середнiх (k-means Method)*

Даний метод належить до алгоритмiв неiєрахiчного подiлу (Partitioning algorithms), якi здiйснюють декомпозицiю набору даних, що складається з n спостережень, на k груп (кластерiв) iз заздалегiдь невiдомими параметрами. При цьому виконується пошук центроїдiв – максимально вiддалених один вiд одного центрiв згущень точок з мiнiмальним розкидом усерединi кожного кластера.

*Покрокова реалiзацiя методу*

Постановка задачi та попередня обробка даних.



Нехай iснує набiр даних з n = 19 значень, якi виглядають приблизно так:

Тепер припустимо, що ми знаємо, що цi данi розбиваються на три, вiдносно очевиднi категорiї i виглядатиме це так:

Рис. 5

Завдання – використовувати алгоритм кластеризацiї k-means, щоб зробити цю категоризацiю.

* Крок 1: Вибираємо кiлькiсть кластерiв, k.

Число кластерiв, якi ми хочемо розпiзнати, це i є k у k-means. У нашому випадку, тому що ми припустили, що всього 3 кластери, k = 3.

* Крок 2: Вибираємо до випадкових значень

Процес виявлення кластерiв ми починаємо з вибору трьох випадково обраних значень (не обов’язково, щоб вони були нашими даними). Цi точки зараз працюватимуть як центроїди або центри кластерiв, якi ми збираємося згрупувати:

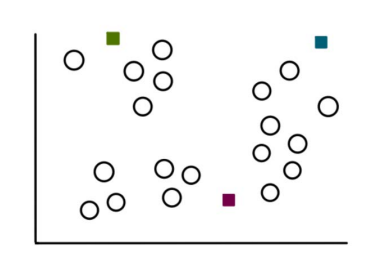


Рис. 6

* Крок 3: Створення кластерiв k

Щоб створити кластери, ми почнемо вимiрювати вiдстань мiж кожним значенням наших даних до кожного з трьох центроїдiв, а потiм додаємо його до найближчого кластера.

Для значення, взятого як приклад, вiдстанi виглядатимуть приблизно так:

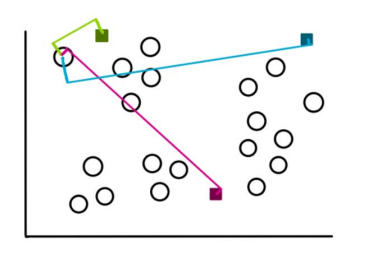
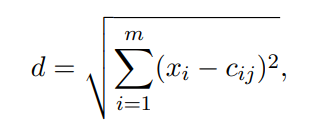


Рис. 7

Для розрахунку вiдстанi мiж об’єктами буде використовувати стандартне визначення декартової вiдстанi:



де xi – координата в m-вимiрному просторi (значення кожної ознаки); cij – координата кожного центроїда. У 2-мiрному просторi для знаходження вiдстанi мiж двома точками маємо:



Використовуючи цю формулу, ми повторюємо процес зi значеннями, що залишилися, пiсля цього кластери будуть виглядати наступним чином:

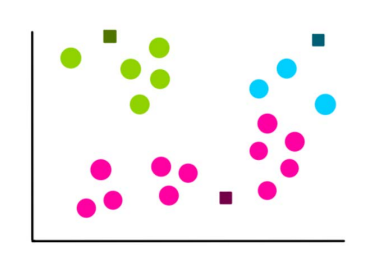


Рис. 8

* Крок 4: Обчислюємо новий центроїд кожного кластера

Тепер, коли ми маємо всi три кластери, ми знаходимо новi центроїди для кожного з них за формулою:



ckj – k-та координата j-того кластера;

xki – k-та координата i-того об’єкта, що вiднесений до jтого кластера;

nj – кiлькiсть об’єктiв у j-тому кластерi.

Так, новi центроїди матимуть вигляд:

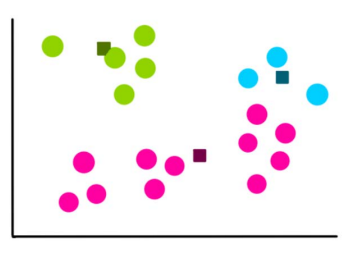
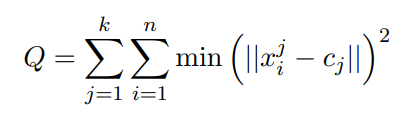


Рис. 9

Крок 5: Знову вiднесемо об’єкти до кожного з центроїдiв

Деякi об’єкти вiднесуться до iншого центроїду (кластеру). Розраховуємо суму квадратiв внутрiшньокластерних вiдстаней до центру кластера (within-cluster sum of squares, WCSS):



* Крок 6: повторюємо кроки 4-5

Кроки 4 i 5 повторюються до тих пiр, поки алгоритм не стабiлiзується, тобто до тих пiр, поки об’єкти не перестануть переходити вiд одного центроїду (кластера) до iншого. Говорячи формально, мета алгоритму – мiнiмiзувати функцiя втрат Q.

Давайте припустимо, що наступнi 4 iтерацiї виглядатимуть так:

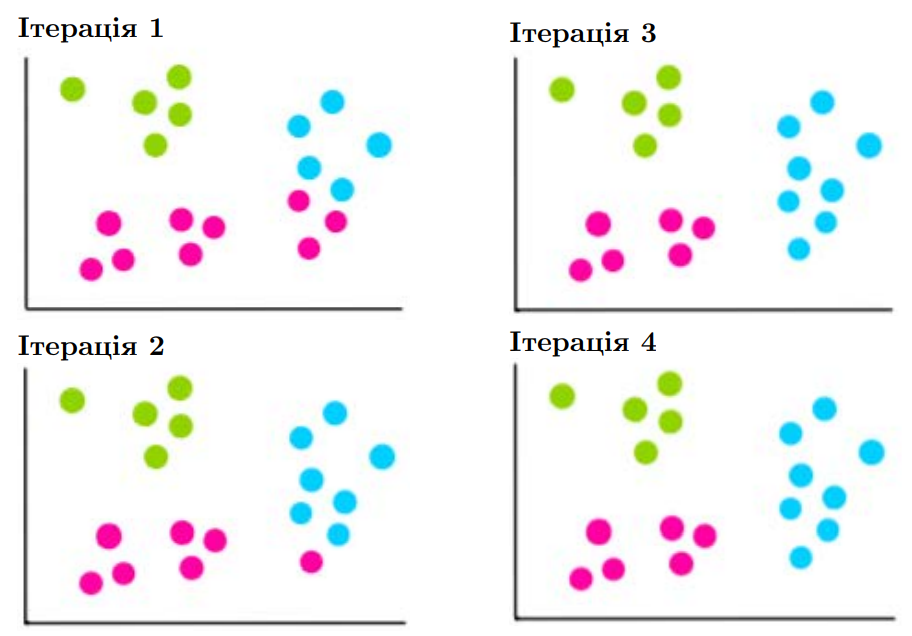
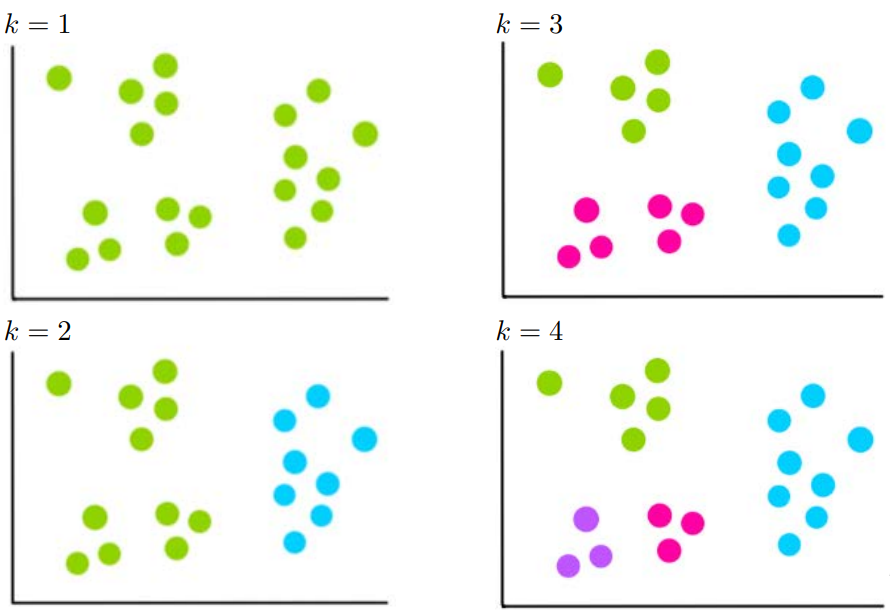


Рис. 10

На двох останнiх iтерацiях бачимо, що кластери не змiнюються. Це означає, що алгоритм зiйшовся i ми зупиняємо процес. Потiм ми вибираємо кластери iз найменшим WCSS. Це будуть кластери на останнiх 2 iтерацiях, тому вони стають нашими фiнальними кластерами.

*Вибiр кiлькостi кластерiв*

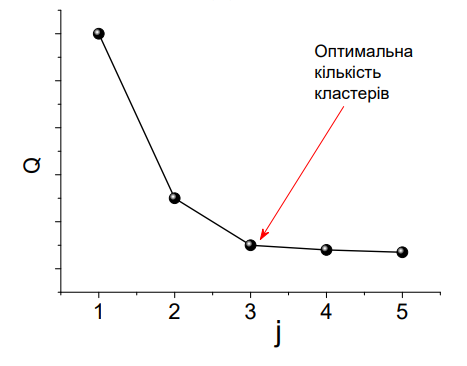
Один iз способiв вибору кiлькостi кластерiв є експертний метод – вибiр кiлькостi кластерiв залежатиме вiд знання про предметну область (domain knowledge).



При кластеризацiї методом k-середнiх кiлькiсть кластерiв найчастiше оцiнюють за допомогою «методу лiктя» (elbow method).

Вiн передбачає багаторазове циклiчне виконання алгоритму зi збiльшенням кiлькостi кластерiв, що вибираються.

Рис. 11

Для кожного вибору отримаємо мiнiмальне значення суму квадратiв внутрiшньокластерних вiдстаней Q. За результатами дослiджень будуємо графiк залежностi Q(j).

На рисунку 12 наведено типову залежнiсть Q(j). Як бачимо, пiсля того, як кiлькiсть кластерiв досягає трьох, сума квадратiв внутрiшньокластерних вiдстаней перестає суттєво зменшуватися.

Рис. 12

## *Кластеризацiя методом DBscan*

Маючи справу з просторовими кластерами рiзної щiльностi, розмiру та форми, може бути складно виявити групу точок. Завдання може бути ще складнiшим, якщо данi мiстять шум i викиди. Для роботи з великими просторовими базами даних був запропонований метод DBSCAN (Density-based spatial clustering of applications with noise, густинний алгоритм просторової кластеризацiї з присутнiстю шуму), який, як випливає з назви, оперує щiльнiстю даних.

На вхiд вiн просить вже знайому матрицю близькостi та два параметри – радiус epsilon-околицi та кiлькiсть сусiдiв. Так одразу i не зрозумiєш, як їх вибрати, причому тут щiльнiсть, i чому саме DBSCAN добре розправляється з шумовими даними. Без цього складно визначити межi його застосування. Можна видiлити три основнi причини використання алгоритму:

1. Вимагає мiнiмальних знань предметної галузi.
2. Вiн може виявити кластери довiльної форми.
3. Ефективний для великих баз даних, тобто коли розмiр вибiрки перевищує кiлька тисяч.

*Формальний пiдхiд*

Уведемо кiлька визначень. Нехай задана деяка симетрична функцiя вiдстанi та константи ϵ i m. Тодi:

1. Назвемо область E(x), для якої ∀y : ρ(x, y) ≤ ϵ, ϵ-околиця об’єкта x.

2. Кореневим об’єктом (core point) ступеня m називається об’єкт, ϵ-околиця якого мiстить щонайменше m об’єктiв: |E(x)| ≥ m.

3. Об’єкт p безпосередньо щiльно досягається з об’єкта q, якщо i – кореневий об’єкт.

4. Об’єкт щiльно-досяжний з об’єкта q, якщо ∃p1, p2 , …, pn, p1 = q, pn = p, такi що ∀i ∈ 1 . . . n − 1 : pi+1 безпосередньо щiльнодосяжний з pi.

Оберемо якийсь кореневий об’єкт p з датасету, позначимо його i помiстимо всiх його безпосередньо щiльно-досяжних сусiдiв у список обходу. Тепер для кожного об’єкту q зi списку: позначимо його, i якщо вiн теж є кореневим, додамо всiх його сусiдiв до списку обходу. Тривiально доводиться, що кластери помiчених точок, сформованi в ходi цього алгоритму максимальнi (тобто їх не можна розширити ще однiєю точкою, щоб задовольнялися умови) i зв’язнi в сенсi досяжностi. Звiдси випливає, що якщо ми обiйшли не всi точки, можна перезапустити обхiд з якогось iншого кореневого об’єкта, i новий кластер не поглине попереднiй.

*Налаштування параметрiв*

Iснують варiанти DBSCAN, якi здатнi пом’якшити цю проблему.

Iдея полягає у пiдстроюваннi ϵ у рiзних галузях по ходу роботи алгоритму. На жаль, зростає кiлькiсть параметрiв алгоритму.

Iснують евристики для вибору m та ϵ. Найчастiше застосовується такий метод та його варiацiї:

1. Обираємо m. Зазвичай використовуються значення вiд 3 до 9, чим неоднорiднiший очiкується датасет, i чим бiльший рiвень шуму, тим бiльшим слiд взяти m.
2. Обчислюємо середню вiдстань по m найближчим сусiдам кожної точки. Тобто, якщо m = 3, потрiбно вибрати трьох найближчих сусiдiв, скласти вiдстанi й подiлити втричi.
3. Сортуємо отриманi значення за зростанням та виводимо на екран.
4. Отримуємо зростаючу залежнiсть. Обираємо ϵ десь у смузi, де вiдбувається найсильнiший перегин. Чим бiльше ϵ, тим бiльшими вийдуть кластери, i тим менше їх буде.

У будь-якому випадку, головнi недолiки DBSCAN – нездатнiсть з’єднувати кластери через прорiзи, i, навпаки, здатнiсть пов’язувати рiзнi кластери через щiльно населенi перемички. Частково тому зi збiльшенням розмiрностi даних n пiдступний “удар у спину” завдає «прокляття» розмiрностi: що бiльше n, то бiльше мiсць, де можуть випадково виникнути отвори чи мости. Адекватна кiлькiсть точок даних N зростає експоненцiйно зi збiльшенням n.

*Пiдсумок*

DBSCAN слiд використовувати, коли:

* Датасет є в мiру великий (N approx106). Навiть N ≈ 107 −108 якщо пiд рукою оптимiзована та розпаралена реалiзацiя.
* Наперед вiдома функцiя близькостi, симетрична, бажано, не дуже складна. KD-Tree оптимiзацiя часто працює тiльки з евклiдовою вiдстанню.
* Очiкуванi кластери складної форми: вкладенi та аномальнi кластери, згустки даних малої розмiрностi.
* Щiльнiсть меж мiж згустками менше щiльностi найменш щiльного кластера. Краще якщо кластери зовсiм вiдокремленi один вiд одного.
* Складнiсть елементiв датасета значення не має. Однак їх має бути достатньо, щоб не виникало сильних розривiв у густинi.
* Кiлькiсть елементiв у кластерi може змiнюватись як завгодно.
* Кiлькiсть викидiв значення не має (в розумних межах), якщо вони розсiянi у великому об’ємi.
* Кiлькiсть кластерiв значення немає.

*Сфери застосування*

DBSCAN має iсторiю успiшних застосувань. Наприклад, можна вiдзначити його використання у завданнях:

* Виявлення соцiальних гурткiв
* Сегментування зображень
* Моделювання поведiнки користувачiв веб-сайтiв
* Попередня обробка в задачi прогнозування погоди

## *Пошук асоціативних правил з використанням методу Apriori*

Асоціація шукає закономірності у потоці замовлень

Навчання асоціативним правилам або пошук асоціативних правил — це метод навчання машин на базі правил виявлення зв'язків, що цікавлять нас, між змінними у великій базі даних.

Метод пропонується для встановлення сильних правил, виявлених у базі даних за допомогою деяких заходів цікавості. Цей заснований на правилах підхід також генерує нові правила в міру аналізу додаткових даних. Кінцевою метою, виходячи з досить великого набору даних, допомогти машині імітувати виділення людських ознак і створити можливість знаходження абстрактних асоціацій з нових некласифікованих даних.

Сьогодні використовують для:

* Прогноз акцій та розпродажів
* Аналіз товарів, що купуються разом
* Розміщення товарів на полицях
* Аналіз патернів поведінки на веб-сайтах

Популярні алгоритми: Apriori, FP-growth.

*Алгоритм APriory – пошук в ширину*

Кроки в Apriory

1) На першій ітерації алгоритму кожен предмет приймається як кандидат набору з одного предмета. Алгоритм буде рахувати випадки появи кожного елемента.

2) Нехай буде якась мінімальна підтримка, min\_sup. Визначається набір 1- елементних наборів предметів, поява яких задовольняє мінімальну суму. Тільки тих кандидатів, для яких підтримка більша або дорівнює min\_sup, приймають на наступну ітерацію, а інших обрізають.

3) Далі виявляються часті набори предметів. Для цього на етапі об’єднання набір із двох елементів формується шляхом формування групи з 2 шляхом комбінування елементів між собою.

4) Кандидати з 2 елементів обрізаються з використанням порогового значення min-sup. Тепер у таблиці будуть дво-елементні набори з мінімальною сумою.

5) Наступна ітерація сформує три-елементні набори, використовуючи кроки об’єднання та обрізки. Ця ітерація буде слідувати властивості антимонотонності. Якщо всі підмножини з 2 елементів є частими, тоді надмножина буде частою, інакше вона буде обрізана.

6) Наступним кроком буде створення набору 4-елементів, з'єднавши 3-набір елементів із собою та обрізання, якщо його підмножина не відповідає критеріям min\_sup. Алгоритм зупиняється, коли досягається найчастіший набір елементів.



Рис. 13



Рис.14

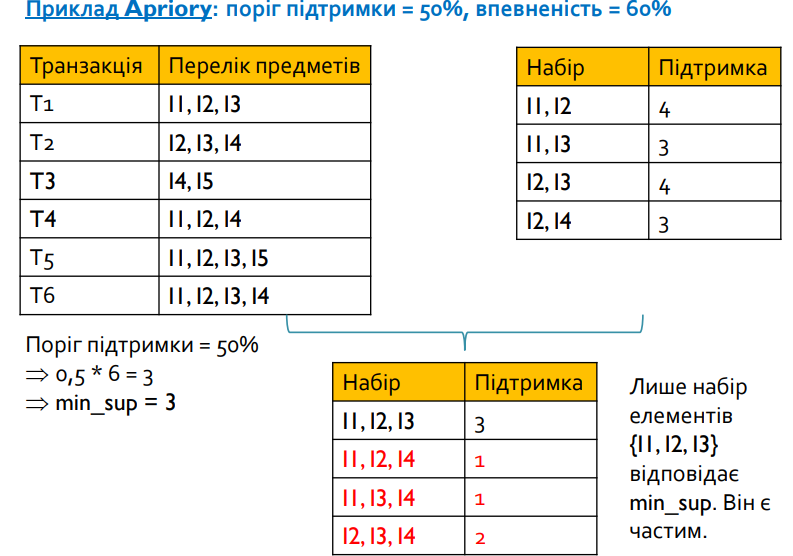


Рис.15



Рис. 16

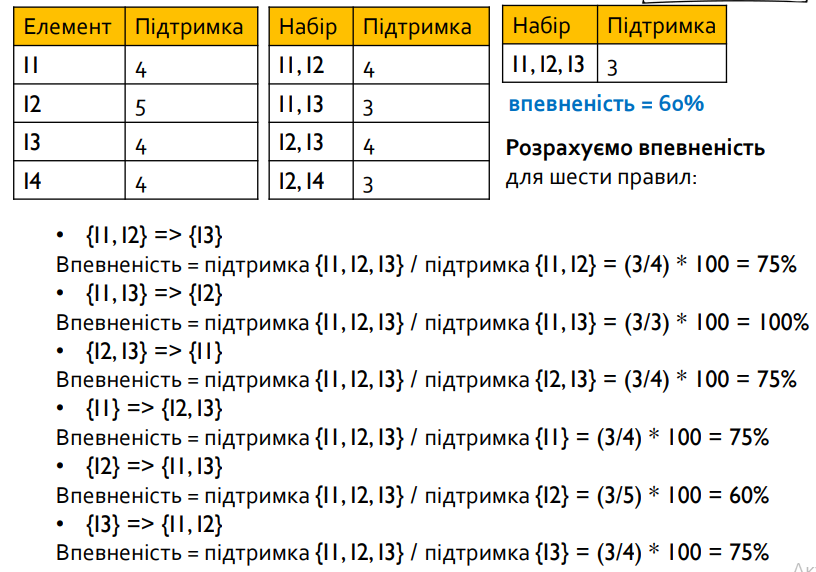


Рис. 17

# **Програмна реалізація**

Завдання були виконані у Jupiter Notebook. ¶

## Метод дерева прийняття рішень (decision tree)[¶](#%D0%9C%D0%B5%D1%82%D0%BE%D0%B4-%D0%B4%D)

Лабораторна робота 1¶

Task 1

Постановка задачі

1. Розділити всю вибірку на навчальну та тестову
2. Побудувати алгоритм навчання на навчальній вибірці з використанням вбудованого алгоритму DecisionTreeClassifier() з бібліотеки sklearn
3. Перевірити результати роботи на тестовій вибірці.
4. Вивести на екран дерево.
5. Оформити результати у вигляді звіту.

Розглянемо перший датасет[¶](" \l "%D0%A0%D0%BE%D0%B7%D0%B3%D0%BB%D1%8F%D0)

* Імпортуємо необхідні бібліотеки

In [1]:

import numpy as np

import pandas as pd

from matplotlib import pyplot as plt

In [2]:

from sklearn import tree

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import train\_test\_split

# Імпорт модуля метрик scikit-learn для розрахунку точності

from sklearn import metrics

from sklearn.tree import export\_graphviz

from io import StringIO

from IPython.display import Image

import graphviz

import pydotplus

In [3]:

!pip install graphviz

!pip install pydotplus

Requirement already satisfied: graphviz in c:\users\admin\anaconda3\lib\site-packages (0.20.1)

Requirement already satisfied: pydotplus in c:\users\admin\anaconda3\lib\site-packages (2.0.2)

Requirement already satisfied: pyparsing>=2.0.1 in c:\users\admin\anaconda3\lib\site-packages (from pydotplus) (3.0.9)

In [4]:

#conda install graphviz

* Будемо розглядати датасет **creditcard.csv**. Виведемо його у формі датафрейму.

In [5]:

data = pd.read\_csv('creditcard.csv')

In [6]:

data.head()

Out[6]:

|  | **Time** | **V1** | **V2** | **V3** | **V4** | **V5** | **V6** | **V7** | **V8** | **V9** | **...** | **V21** | **V22** | **V23** | **V24** | **V25** | **V26** | **V27** | **V28** | **Amount** | **Class** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0.0 | -1.359807 | -0.072781 | 2.536347 | 1.378155 | -0.338321 | 0.462388 | 0.239599 | 0.098698 | 0.363787 | ... | -0.018307 | 0.277838 | -0.110474 | 0.066928 | 0.128539 | -0.189115 | 0.133558 | -0.021053 | 149.62 | 0 |
| **1** | 0.0 | 1.191857 | 0.266151 | 0.166480 | 0.448154 | 0.060018 | -0.082361 | -0.078803 | 0.085102 | -0.255425 | ... | -0.225775 | -0.638672 | 0.101288 | -0.339846 | 0.167170 | 0.125895 | -0.008983 | 0.014724 | 2.69 | 0 |
| **2** | 1.0 | -1.358354 | -1.340163 | 1.773209 | 0.379780 | -0.503198 | 1.800499 | 0.791461 | 0.247676 | -1.514654 | ... | 0.247998 | 0.771679 | 0.909412 | -0.689281 | -0.327642 | -0.139097 | -0.055353 | -0.059752 | 378.66 | 0 |
| **3** | 1.0 | -0.966272 | -0.185226 | 1.792993 | -0.863291 | -0.010309 | 1.247203 | 0.237609 | 0.377436 | -1.387024 | ... | -0.108300 | 0.005274 | -0.190321 | -1.175575 | 0.647376 | -0.221929 | 0.062723 | 0.061458 | 123.50 | 0 |
| **4** | 2.0 | -1.158233 | 0.877737 | 1.548718 | 0.403034 | -0.407193 | 0.095921 | 0.592941 | -0.270533 | 0.817739 | ... | -0.009431 | 0.798278 | -0.137458 | 0.141267 | -0.206010 | 0.502292 | 0.219422 | 0.215153 | 69.99 | 0 |

5 rows × 31 columns

In [7]:

#data.info()

* Далі визначаємо ознаки та цільову змінну. Ділимо вибірку на навчальну і тестову. Виведемо їх розмірності.

In [8]:

X = data.drop(columns='Class', axis=1)

y = data['Class']

In [9]:

data['Class'].value\_counts()

Out[9]:

0 284315

1 492

Name: Class, dtype: int64

In [10]:

X\_train, X\_test, y\_train, y\_test = \

train\_test\_split(X, y, test\_size = 0.25, random\_state = 1) # 75% training and 25% test

print(X\_train.shape, X\_test.shape)

print(y\_train.shape, y\_test.shape)

(213605, 30) (71202, 30)

(213605,) (71202,)

* Створюємо об'єкт класифікатора дерева рішень. За замовчуванням у класифікатора стоїть критерій gini (default="gini"), тому можна не вказувати його. Максимальна глибина дерева буде 3.
* Створюємо класифікатор дерева прийняття рішень для тренувань.
* Робимо прогноз на тестовому наборі даних.

In [11]:

clf = DecisionTreeClassifier(random\_state=0, max\_depth=3)

clf = clf.fit(X\_train, y\_train)

y\_pred = clf.predict(X\_test)

* Перевіряємо результати роботи на тестовій вибірці за допомогою точності моделі (як часто класифікатор є правильним):

In [12]:

y\_pred = clf.predict(X\_test)

print("Оцінка тестового набору даних: ", metrics.accuracy\_score(y\_test, y\_pred))

Оцінка тестового набору даних: 0.9991713715906857

* Виводимо на екран дерево прийняття рішень:

In [13]:

plt.figure(figsize=(15,10))

tree.plot\_tree(clf.fit(X\_train, y\_train))

Out[13]:

[Text(0.5, 0.875, 'X[17] <= -2.789\ngini = 0.004\nsamples = 213605\nvalue = [213224, 381]'),

Text(0.25, 0.625, 'X[10] <= -2.129\ngini = 0.348\nsamples = 348\nvalue = [78, 270]'),

Text(0.125, 0.375, 'X[26] <= -0.225\ngini = 0.256\nsamples = 305\nvalue = [46, 259]'),

Text(0.0625, 0.125, 'gini = 0.442\nsamples = 103\nvalue = [34, 69]'),

Text(0.1875, 0.125, 'gini = 0.112\nsamples = 202\nvalue = [12, 190]'),

Text(0.375, 0.375, 'X[21] <= 0.304\ngini = 0.381\nsamples = 43\nvalue = [32, 11]'),

Text(0.3125, 0.125, 'gini = 0.124\nsamples = 30\nvalue = [28, 2]'),

Text(0.4375, 0.125, 'gini = 0.426\nsamples = 13\nvalue = [4, 9]'),

Text(0.75, 0.625, 'X[14] <= -8.092\ngini = 0.001\nsamples = 213257\nvalue = [213146, 111]'),

Text(0.625, 0.375, 'X[15] <= 1.423\ngini = 0.198\nsamples = 27\nvalue = [3, 24]'),

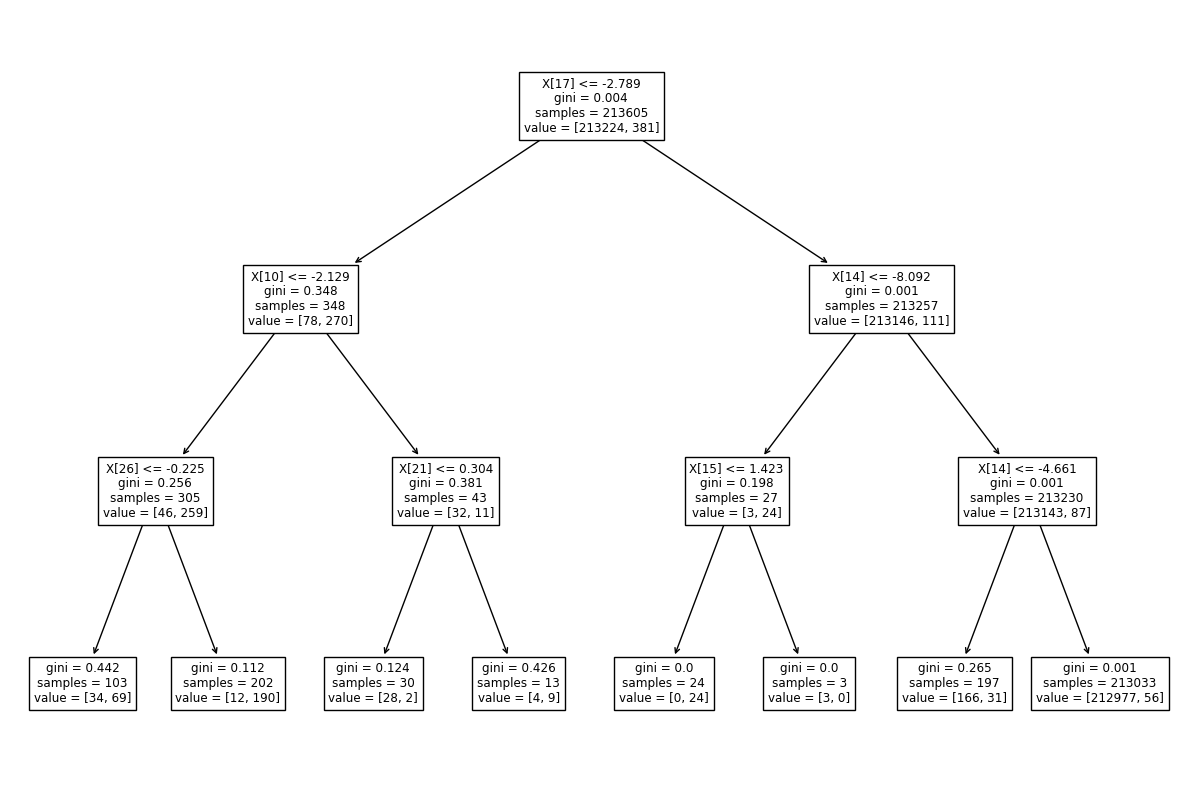
Text(0.5625, 0.125, 'gini = 0.0\nsamples = 24\nvalue = [0, 24]'),

Text(0.6875, 0.125, 'gini = 0.0\nsamples = 3\nvalue = [3, 0]'),

Text(0.875, 0.375, 'X[14] <= -4.661\ngini = 0.001\nsamples = 213230\nvalue = [213143, 87]'),

Text(0.8125, 0.125, 'gini = 0.265\nsamples = 197\nvalue = [166, 31]'),

Text(0.9375, 0.125, 'gini = 0.001\nsamples = 213033\nvalue = [212977, 56]')]



* Візуалізуємо дерево більш детально

In [14]:

feature\_cols\_credit = [i for i in X.columns]

dot\_data\_credit = StringIO()

export\_graphviz(clf, out\_file = dot\_data\_credit,

feature\_names = feature\_cols\_credit,

class\_names = ["0", "1"],

filled = True, rounded = True,

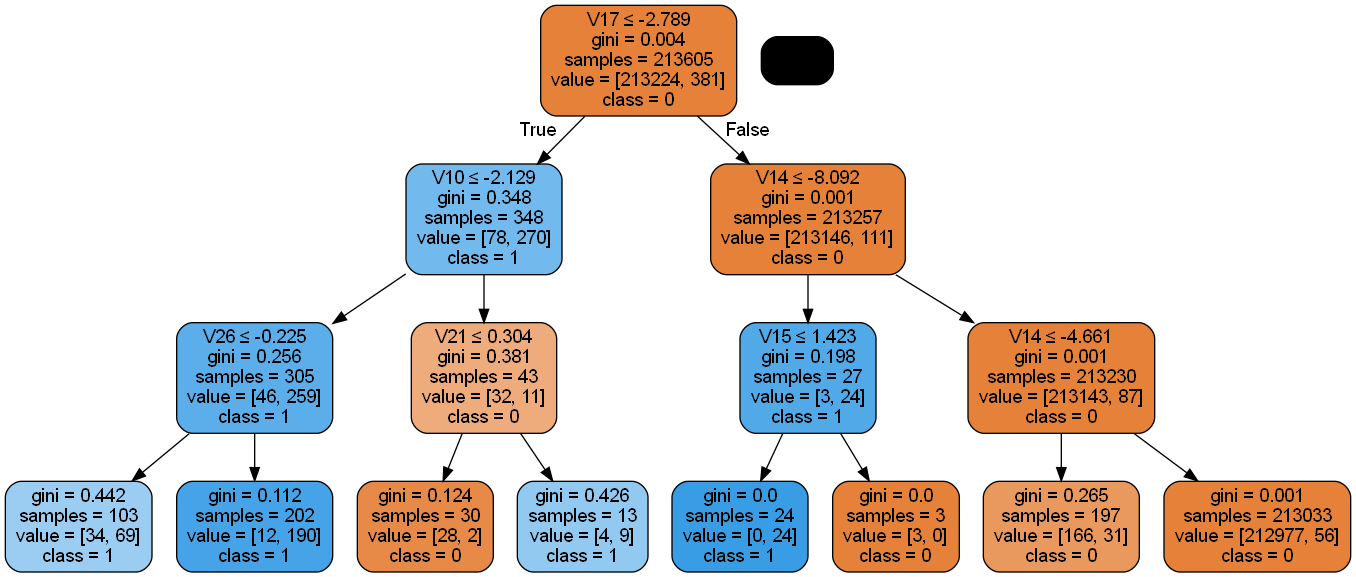
special\_characters = True)

graph\_credit = pydotplus.graph\_from\_dot\_data(dot\_data\_credit.getvalue())

graph\_credit.write\_png('lab1\_credit.png')

Image(graph\_credit.create\_png())

Out[14]:



''' %matplotlib inline feature\_cols\_credit = [i for i in X.columns] dot\_data\_credit = tree.export\_graphviz(clf, out\_file = None, feature\_names = feature\_cols\_credit, class\_names = ["0", "1"], filled = True, rounded = True, special\_characters = True) graph\_credit = graphviz.Source(dot\_data\_credit) graph\_credit.save('lab1\_credit.png') graph\_credit '''

Розглянемо тепер інший датасет[¶](" \l "%D0%A0%D0%BE%D0%B7%D0%B3%D0%BB%D1%8F%D0)

* Імпортуємо необхідні бібліотеки

In [15]:

from sklearn.datasets import load\_iris

from sklearn.tree import DecisionTreeClassifier

from sklearn.tree import export\_text

* Будемо розглядати датасет **iris**. Далі визначаємо ознаки та цільову змінну. Виведемо датасет у формі датафрейму

In [16]:

iris = load\_iris()

In [17]:

X2, y2 = iris.data, iris.target

In [18]:

# dfIris = pd.DataFrame(data = np.c\_[iris['data'], iris['target']], columns = iris['feature\_names'] + ['target'])

dfIris = pd.DataFrame(data = np.c\_[X2, y2], columns = iris['feature\_names'] + ['target'])

dfIris

Out[18]:

|  | **sepal length (cm)** | **sepal width (cm)** | **petal length (cm)** | **petal width (cm)** | **target** |
| --- | --- | --- | --- | --- | --- |
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | 0.0 |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | 0.0 |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | 0.0 |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | 0.0 |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | 0.0 |
| **...** | ... | ... | ... | ... | ... |
| **145** | 6.7 | 3.0 | 5.2 | 2.3 | 2.0 |
| **146** | 6.3 | 2.5 | 5.0 | 1.9 | 2.0 |
| **147** | 6.5 | 3.0 | 5.2 | 2.0 | 2.0 |
| **148** | 6.2 | 3.4 | 5.4 | 2.3 | 2.0 |
| **149** | 5.9 | 3.0 | 5.1 | 1.8 | 2.0 |

150 rows × 5 columns

Класи розглянутих об'єктів:

In [19]:

print(iris['target\_names'])

['setosa' 'versicolor' 'virginica']

In [20]:

print(dfIris['target'].unique())

[0. 1. 2.]

* Ділимо вибірку на навчальну і тестову. Виведемо їх розмірності.

In [21]:

X2\_train, X2\_test, y2\_train, y2\_test = \

train\_test\_split(X2, y2, test\_size = 0.25, random\_state = 1) # 75% training and 25% test

print(X2\_train.shape, X2\_test.shape)

print(y2\_train.shape, y2\_test.shape)

(112, 4) (38, 4)

(112,) (38,)

* Створюємо об'єкт класифікатора дерева рішень. За замовчуванням у класифікатора стоїть критерій gini (default="gini"), тому можна не вказувати його. Максимальна глибина дерева буде 3.
* Створюємо класифікатор дерева прийняття рішень для тренувань.
* Робимо прогноз на тестовому наборі даних.

In [22]:

clf2 = DecisionTreeClassifier(random\_state=0, max\_depth=3)

clf2 = clf2.fit(X2\_train, y2\_train)

y2\_pred = clf2.predict(X2\_test)

* Перевіряємо результати роботи на тестовій вибірці за допомогою точності моделі (як часто класифікатор є правильним):

In [23]:

y2\_pred = clf2.predict(X2\_test)

print("Оцінка тестового набору даних: ", metrics.accuracy\_score(y2\_test, y2\_pred))

Оцінка тестового набору даних: 0.9736842105263158

* Виводимо на екран дерево прийняття рішень:

In [24]:

#plt.figure(figsize=(10,9))

#tree.plot\_tree(clf2.fit(X2\_train, y2\_train))

#plt.title("Дерево прийняття рішень (decision tree)")

In [25]:

decision\_tree\_iris = clf2

tx = export\_text(decision\_tree\_iris, feature\_names=iris['feature\_names'])

print(tx)

|--- petal width (cm) <= 0.80

| |--- class: 0

|--- petal width (cm) > 0.80

| |--- petal width (cm) <= 1.65

| | |--- petal length (cm) <= 4.95

| | | |--- class: 1

| | |--- petal length (cm) > 4.95

| | | |--- class: 2

| |--- petal width (cm) > 1.65

| | |--- petal length (cm) <= 4.85

| | | |--- class: 2

| | |--- petal length (cm) > 4.85

| | | |--- class: 2

In [26]:

dot\_data = tree.export\_graphviz(clf2, out\_file = None,

feature\_names = iris.feature\_names,

class\_names = ["0", "1", "2"],

filled = True, rounded = True,

special\_characters = True)

graph = graphviz.Source(dot\_data)

#graph.save('lab1\_iris.png')

graph.render("lab1\_iris")

graph

Out[26]:

petal width (cm) ≤ 0.8 gini = 0.665 samples = 112 value = [37, 34, 41] class = 2 gini = 0.0 samples = 37 value = [37, 0, 0] class = 0 True petal width (cm) ≤ 1.65 gini = 0.496 samples = 75 value = [0, 34, 41] class = 2 False petal length (cm) ≤ 4.95 gini = 0.193 samples = 37 value = [0, 33, 4] class = 1 petal length (cm) ≤ 4.85 gini = 0.051 samples = 38 value = [0, 1, 37] class = 2 gini = 0.0 samples = 32 value = [0, 32, 0] class = 1 gini = 0.32 samples = 5 value = [0, 1, 4] class = 2 gini = 0.375 samples = 4 value = [0, 1, 3] class = 2 gini = 0.0 samples = 34 value = [0, 0, 34] class = 2

* Наступний варіант з назвами класів ірисів.

In [27]:

dot\_data2 = tree.export\_graphviz(clf2, out\_file = None,

feature\_names = iris.feature\_names,

class\_names = iris.target\_names,

filled = True, rounded = True,

special\_characters = True)

graph2 = graphviz.Source(dot\_data2)

#graph2.save('lab1\_iris\_classColors.jpg')

graph2.render("lab1\_iris\_classColors")

graph2

Out[27]:

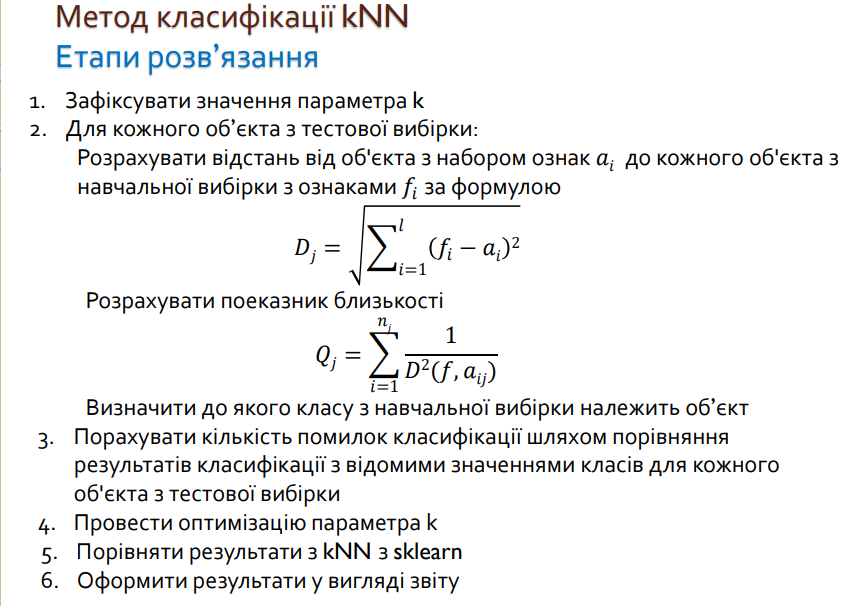
petal width (cm) ≤ 0.8 gini = 0.665 samples = 112 value = [37, 34, 41] class = virginica gini = 0.0 samples = 37 value = [37, 0, 0] class = setosa True petal width (cm) ≤ 1.65 gini = 0.496 samples = 75 value = [0, 34, 41] class = virginica False petal length (cm) ≤ 4.95 gini = 0.193 samples = 37 value = [0, 33, 4] class = versicolor petal length (cm) ≤ 4.85 gini = 0.051 samples = 38 value = [0, 1, 37] class = virginica gini = 0.0 samples = 32 value = [0, 32, 0] class = versicolor gini = 0.32 samples = 5 value = [0, 1, 4] class = virginica gini = 0.375 samples = 4 value = [0, 1, 3] class = virginica gini = 0.0 samples = 34 value = [0, 0, 34] class = virginica

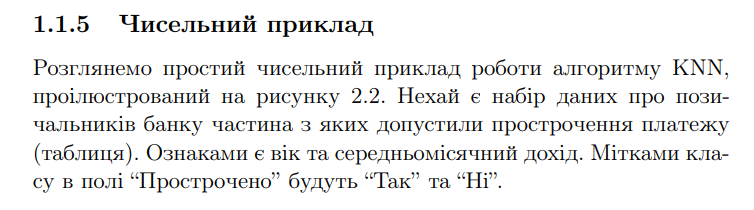
''' dot\_data = StringIO() export\_graphviz(clf2, out\_file = None, feature\_names = iris.feature\_names, class\_names = ["0", "1", "2"], filled = True, rounded = True, special\_characters = True) graph = pydotplus.graph\_from\_dot\_data(dot\_data.getvalue()) graph.write\_png('lab1\_iris.png') Image(graph.create\_png()) '''

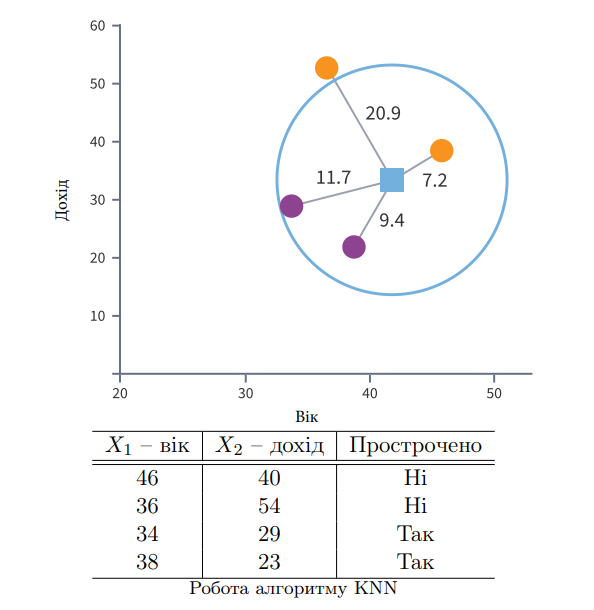
Машинне навчання[¶](" \l "%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D0)

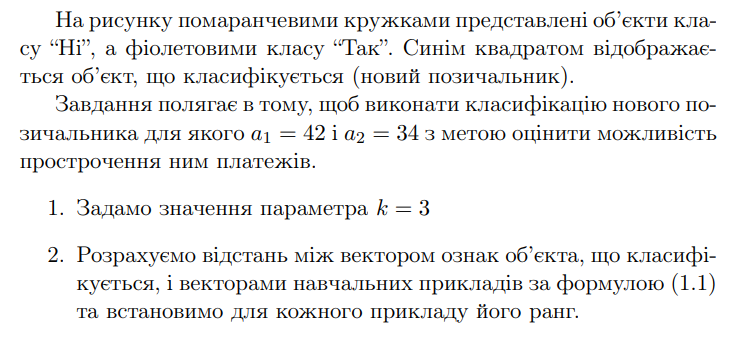
## Метод класифікації kNN[¶](" \l "%D0%9C%D0%B5%D1%82%D0%BE%D0%B4-%D0%B4%D)

Лабораторна робота 2¶











* Імпортуємо потрібні бібліотеки

In [1]:

# import warnings filter

from warnings import simplefilter

# ignore all future warnings

simplefilter(action='ignore', category=FutureWarning)

In [2]:

import numpy as np

import pandas as pd

import seaborn as sns

from matplotlib import pyplot as plt

In [3]:

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

* Розглянемо датасет з ірисами. Створимо датафрейм з нього.

In [4]:

iris = load\_iris()

X, y = iris.data, iris.target

In [5]:

# dfIris = pd.DataFrame(data = np.c\_[iris['data'], iris['target']], columns = iris['feature\_names'] + ['target'])

dfIris = pd.DataFrame(data = np.c\_[X, y], columns = iris['feature\_names'] + ['target'])

dfIris

Out[5]:

|  | **sepal length (cm)** | **sepal width (cm)** | **petal length (cm)** | **petal width (cm)** | **target** |
| --- | --- | --- | --- | --- | --- |
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | 0.0 |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | 0.0 |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | 0.0 |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | 0.0 |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | 0.0 |
| **...** | ... | ... | ... | ... | ... |
| **145** | 6.7 | 3.0 | 5.2 | 2.3 | 2.0 |
| **146** | 6.3 | 2.5 | 5.0 | 1.9 | 2.0 |
| **147** | 6.5 | 3.0 | 5.2 | 2.0 | 2.0 |
| **148** | 6.2 | 3.4 | 5.4 | 2.3 | 2.0 |
| **149** | 5.9 | 3.0 | 5.1 | 1.8 | 2.0 |

150 rows × 5 columns

In [6]:

print(iris['target\_names'])

['setosa' 'versicolor' 'virginica']

In [7]:

print(dfIris['target'].unique())

[0. 1. 2.]

* Покажемо на графіках пари відповідних параметрів і значення цільовох функції.

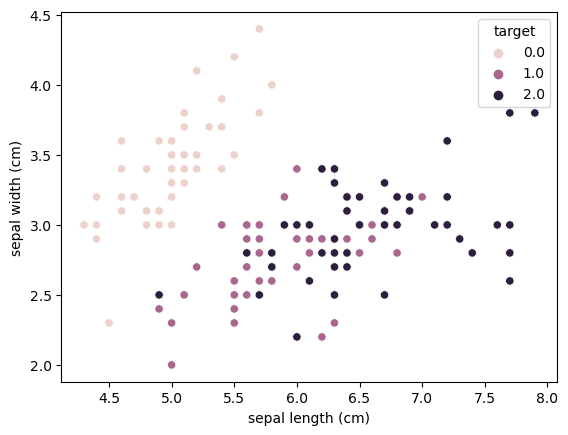
In [8]:

sns.scatterplot(x = dfIris['sepal length (cm)'], y = dfIris['sepal width (cm)'], hue = dfIris['target'])

#plt.legend(bbox\_to\_anchor=(1,1), loc="upper left")

Out[8]:

<AxesSubplot:xlabel='sepal length (cm)', ylabel='sepal width (cm)'>

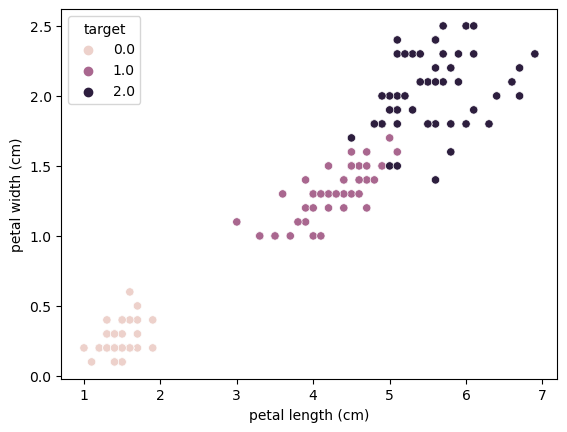


In [9]:

sns.scatterplot(x = dfIris['petal length (cm)'], y = dfIris['petal width (cm)'], hue = dfIris['target'])

Out[9]:

<AxesSubplot:xlabel='petal length (cm)', ylabel='petal width (cm)'>



* Розділимо вибірку на навчальну і тестову.

In [10]:

X\_train, X\_test, y\_train, y\_test = \

train\_test\_split(X, y, test\_size = 0.25, random\_state = 1) # 75% training and 25% test

#print(X\_train.shape, X\_test.shape)

#print(y\_train.shape, y\_test.shape)

* Далі потрібно знайти відстані за відповідними формулами. Також створимо ще навчальний і тестовий датафрейм.

In [11]:

# знаходимо відстані між одним тестовим об'ктом (j) і всіма навчальними (0,...,len(X\_train))

def fun\_distance(X\_train, X\_test, j):

n\_train = len(X\_train)

distances = [np.linalg.norm(X\_train[i] - X\_test[j]) for i in range(n\_train)]

return distances

In [12]:

def fun\_df(X\_train, X\_test, y\_train, y\_test):

print('df\_test(наприклад, перший):')

df\_test = pd.DataFrame(data = np.c\_[X\_test, y\_test], columns = ['x1','x2','x3','x4',] + ['y'])

display(df\_test.head(1))

print('\ndf\_train(перші 5 рядків):')

df\_train = pd.DataFrame(data = np.c\_[X\_train, y\_train], columns = ['x1','x2','x3','x4',] + ['y'])

display(df\_train.head())

return df\_test, df\_train

df\_test, df\_train = fun\_df(X\_train, X\_test, y\_train, y\_test)

df\_test(наприклад, перший):

|  | **x1** | **x2** | **x3** | **x4** | **y** |
| --- | --- | --- | --- | --- | --- |
| **0** | 5.8 | 4.0 | 1.2 | 0.2 | 0.0 |

df\_train(перші 5 рядків):

|  | **x1** | **x2** | **x3** | **x4** | **y** |
| --- | --- | --- | --- | --- | --- |
| **0** | 6.5 | 2.8 | 4.6 | 1.5 | 1.0 |
| **1** | 6.7 | 2.5 | 5.8 | 1.8 | 2.0 |
| **2** | 6.8 | 3.0 | 5.5 | 2.1 | 2.0 |
| **3** | 5.1 | 3.5 | 1.4 | 0.3 | 0.0 |
| **4** | 6.0 | 2.2 | 5.0 | 1.5 | 2.0 |

* Для проведення зваженого голосування розрахуємо показник близькостi **Q**.

In [13]:

def fun\_Q(df\_sort, clas):

y = df\_sort.query('`y` == @clas')

if y.shape[0] == 0:

Q = 0

else:

y['distances^2'] = (y['distances'])\*\*2

Q = 1 / y['distances^2']

Q = Q.sum()

#print('Q\_{0:.0f} = {1}'.format(clas, Q))

return clas, Q

* Тепер визначимо клас конкретного тестового об'єкта.

In [14]:

def fun\_Class(distances, df\_train, k):

# df\_test = pd.DataFrame(data = np.c\_[X\_test, y\_test], columns = ['x1','x2','x3','x4',] + ['y'])

# df\_train = pd.DataFrame(data = np.c\_[X\_train, y\_train], columns = ['x1','x2','x3','x4',] + ['y'])

df\_train['distances'] = distances

display(df\_train.head())

# df\_sort

df\_sort = df\_train.sort\_values(by=['distances'])

display(df\_sort.head())

print('\nk сусідів =',k)

df\_sort = df\_sort.head(k)

display(df\_sort.head())

classes = df\_train['y'].unique()

classes.sort()

#print('classes =', classes)

Q, Class = [], []

for i in classes:

cl, q = fun\_Q(df\_sort, clas=i)

Q.append(q)

Class.append(cl)

df\_new = pd.DataFrame(data = np.c\_[Class, Q], columns = ['class', 'Q'])

display(df\_new)

answer = df\_new[ df\_new['Q'] == df\_new['Q'].max() ]

answer = answer['class'].values

answer = int(answer[0])

namees\_class = iris['target\_names']

print('class prediction: ', answer, '-', namees\_class[answer])

return answer

* **Виконаємо всі дії за алгоритмом для кожного тестового об'єкта по черзі і продемонструємо їх.**

In [15]:

k\_neighbours = 3

y\_pred = []

for j\_test in range(len(X\_test)):

print("----", j\_test+1, " тестовий об`єкт ----")

distances = fun\_distance(X\_train, X\_test, j=j\_test)

y\_pr = fun\_Class(distances, df\_train, k=k\_neighbours)

y\_pred.append(y\_pr)

print("y\_test =", y\_test[j\_test], '\n\n')

---- 1 тестовий об`єкт ----

|  | **x1** | | **x2** | | **x3** | | **x4** | | **y** | | **distances** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 6.5 | | 2.8 | | 4.6 | | 1.5 | | 1.0 | | 3.896152 | |
| **1** | 6.7 | | 2.5 | | 5.8 | | 1.8 | | 2.0 | | 5.174940 | |
| **2** | 6.8 | | 3.0 | | 5.5 | | 2.1 | | 2.0 | | 4.909175 | |
| **3** | 5.1 | | 3.5 | | 1.4 | | 0.3 | | 0.0 | | 0.888819 | |
| **4** | 6.0 | | 2.2 | | 5.0 | | 1.5 | | 2.0 | | 4.405678 | |
|  | | **x1** | | **x2** | | **x3** | | **x4** | | **y** | | **distances** | |
| **62** | | 5.7 | | 4.4 | | 1.5 | | 0.4 | | 0.0 | | 0.547723 | |
| **8** | | 5.7 | | 3.8 | | 1.7 | | 0.3 | | 0.0 | | 0.556776 | |
| **42** | | 5.4 | | 3.7 | | 1.5 | | 0.2 | | 0.0 | | 0.583095 | |
| **13** | | 5.5 | | 3.5 | | 1.3 | | 0.2 | | 0.0 | | 0.591608 | |
| **44** | | 5.2 | | 4.1 | | 1.5 | | 0.1 | | 0.0 | | 0.685565 | |

k сусідів = 3

|  | | **x1** | **x2** | | **x3** | **x4** | | **y** | **distances** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **62** | | 5.7 | 4.4 | | 1.5 | 0.4 | | 0.0 | 0.547723 |
| **8** | | 5.7 | 3.8 | | 1.7 | 0.3 | | 0.0 | 0.556776 |
| **42** | | 5.4 | 3.7 | | 1.5 | 0.2 | | 0.0 | 0.583095 |
|  | **class** | | | **Q** | | |
| **0** | 0.0 | | | 9.500316 | | |
| **1** | 1.0 | | | 0.000000 | | |
| **2** | 2.0 | | | 0.000000 | | |

class prediction: 0 - setosa

y\_test = 0

---- 2 тестовий об`єкт ----

|  | **x1** | | **x2** | | **x3** | | **x4** | | **y** | | **distances** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 6.5 | | 2.8 | | 4.6 | | 1.5 | | 1.0 | | 2.184033 | |
| **1** | 6.7 | | 2.5 | | 5.8 | | 1.8 | | 2.0 | | 3.300000 | |
| **2** | 6.8 | | 3.0 | | 5.5 | | 2.1 | | 2.0 | | 3.223352 | |
| **3** | 5.1 | | 3.5 | | 1.4 | | 0.3 | | 0.0 | | 2.049390 | |
| **4** | 6.0 | | 2.2 | | 5.0 | | 1.5 | | 2.0 | | 2.249444 | |
|  | | **x1** | | **x2** | | **x3** | | **x4** | | **y** | | **distances** | |
| **91** | | 4.9 | | 2.4 | | 3.3 | | 1.0 | | 1.0 | | 0.387298 | |
| **39** | | 5.0 | | 2.3 | | 3.3 | | 1.0 | | 1.0 | | 0.387298 | |
| **88** | | 5.0 | | 2.0 | | 3.5 | | 1.0 | | 1.0 | | 0.721110 | |
| **106** | | 5.7 | | 2.6 | | 3.5 | | 1.0 | | 1.0 | | 0.793725 | |
| **87** | | 5.5 | | 2.4 | | 3.7 | | 1.0 | | 1.0 | | 0.818535 | |

k сусідів = 3

|  | | **x1** | **x2** | | **x3** | **x4** | | **y** | **distances** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **91** | | 4.9 | 2.4 | | 3.3 | 1.0 | | 1.0 | 0.387298 |
| **39** | | 5.0 | 2.3 | | 3.3 | 1.0 | | 1.0 | 0.387298 |
| **88** | | 5.0 | 2.0 | | 3.5 | 1.0 | | 1.0 | 0.721110 |
|  | **class** | | | **Q** | | |
| **0** | 0.0 | | | 0.00000 | | |
| **1** | 1.0 | | | 15.25641 | | |
| **2** | 2.0 | | | 0.00000 | | |

class prediction: 1 - versicolor

y\_test = 1

[…]

---- 38 тестовий об`єкт ----

|  | **x1** | | **x2** | | **x3** | | **x4** | | **y** | | **distances** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 6.5 | | 2.8 | | 4.6 | | 1.5 | | 1.0 | | 3.738984 | |
| **1** | 6.7 | | 2.5 | | 5.8 | | 1.8 | | 2.0 | | 4.998000 | |
| **2** | 6.8 | | 3.0 | | 5.5 | | 2.1 | | 2.0 | | 4.810405 | |
| **3** | 5.1 | | 3.5 | | 1.4 | | 0.3 | | 0.0 | | 0.173205 | |
| **4** | 6.0 | | 2.2 | | 5.0 | | 1.5 | | 2.0 | | 4.090232 | |
|  | | **x1** | | **x2** | | **x3** | | **x4** | | **y** | | **distances** | |
| **50** | | 5.2 | | 3.5 | | 1.5 | | 0.2 | | 0.0 | | 0.141421 | |
| **89** | | 5.1 | | 3.5 | | 1.4 | | 0.2 | | 0.0 | | 0.141421 | |
| **12** | | 5.1 | | 3.4 | | 1.5 | | 0.2 | | 0.0 | | 0.141421 | |
| **3** | | 5.1 | | 3.5 | | 1.4 | | 0.3 | | 0.0 | | 0.173205 | |
| **95** | | 5.0 | | 3.4 | | 1.5 | | 0.2 | | 0.0 | | 0.223607 | |

k сусідів = 3

|  | | **x1** | **x2** | | **x3** | | **x4** | **y** | **distances** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **50** | | 5.2 | 3.5 | | 1.5 | | 0.2 | 0.0 | 0.141421 |
| **89** | | 5.1 | 3.5 | | 1.4 | | 0.2 | 0.0 | 0.141421 |
| **12** | | 5.1 | 3.4 | | 1.5 | | 0.2 | 0.0 | 0.141421 |
|  | **class** | | | **Q** | |
| **0** | 0.0 | | | 150.0 | |
| **1** | 1.0 | | | 0.0 | |
| **2** | 2.0 | | | 0.0 | |

class prediction: 0 - setosa

y\_test = 0

* ***Знайдемо точність - відповідність прогнозованих та реальних даних.***

In [16]:

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy (точність):", accuracy)

Accuracy (точність): 1.0

**З використанням вбудованого алгоритму kNN з бібліотеки *sklearn* провести класифікацію об’єктів. Визначити кількість помилок.¶**

In [17]:

# Масштабування функцій за допомогою StandardScaler

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

In [18]:

knn = KNeighborsClassifier(n\_neighbors=3)

knn.fit(X\_train, y\_train)

Out[18]:

KNeighborsClassifier(n\_neighbors=3)

In [19]:

y\_pred = knn.predict(X\_test)

y\_pred

Out[19]:

array([0, 1, 1, 0, 2, 1, 2, 0, 0, 2, 1, 0, 2, 1, 1, 0, 1, 1, 0, 0, 1, 1,

1, 0, 2, 1, 0, 0, 1, 2, 1, 2, 1, 2, 2, 0, 1, 0])

In [20]:

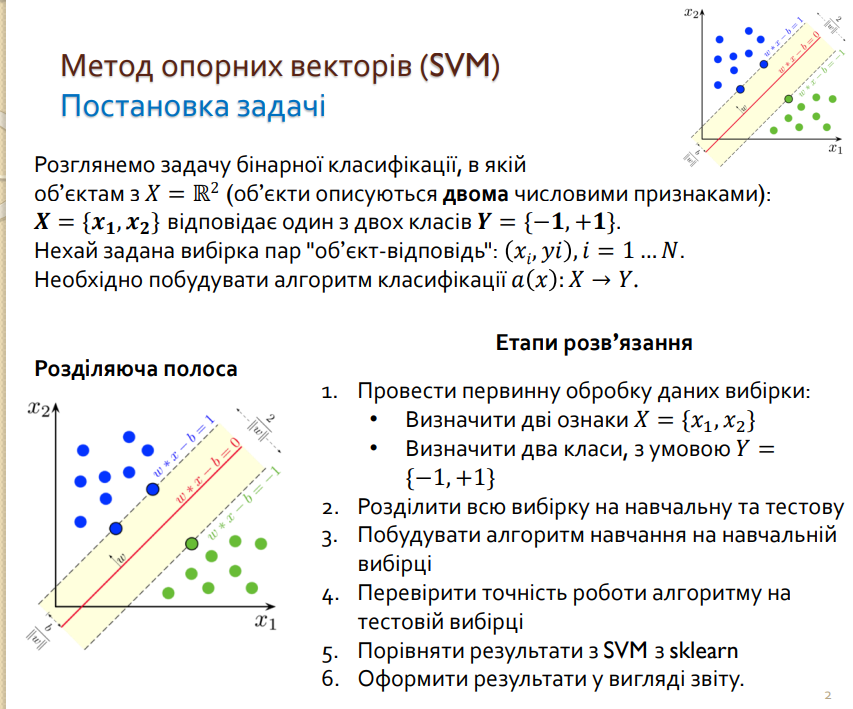
accuracy = accuracy\_score(y\_test, y\_pred)

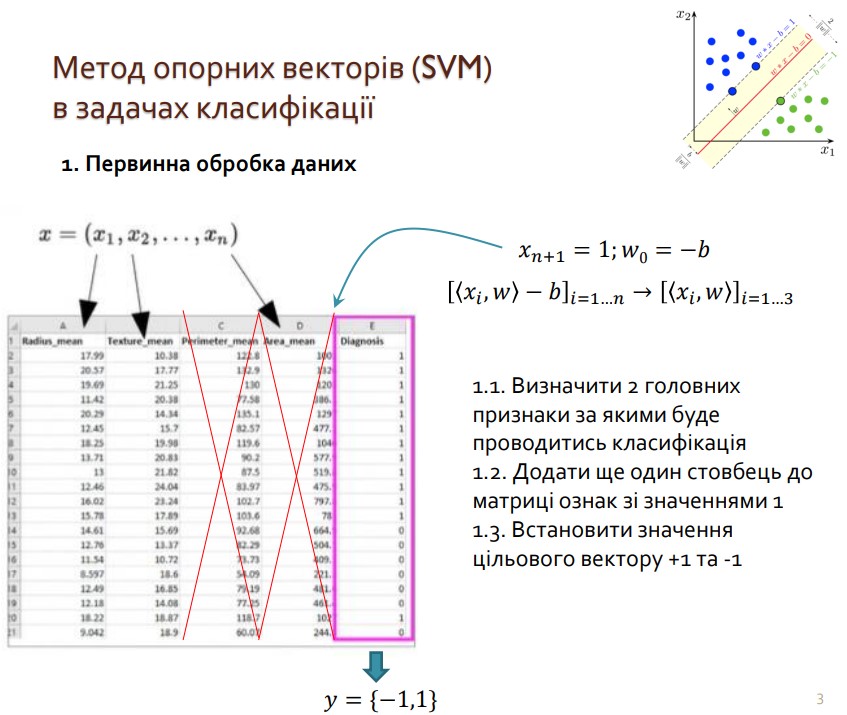
print("Accuracy (точність):", accuracy)

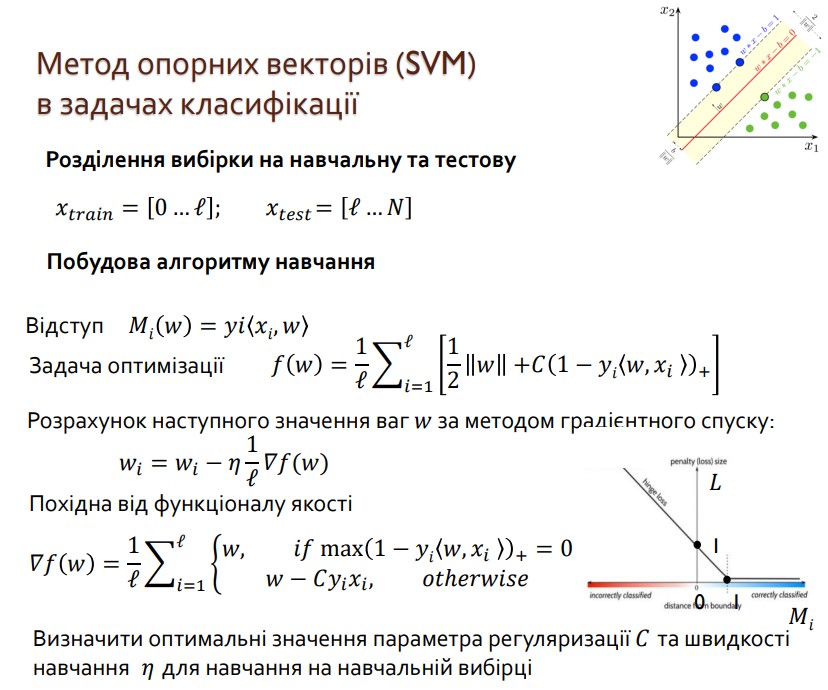
Accuracy (точність): 1.0

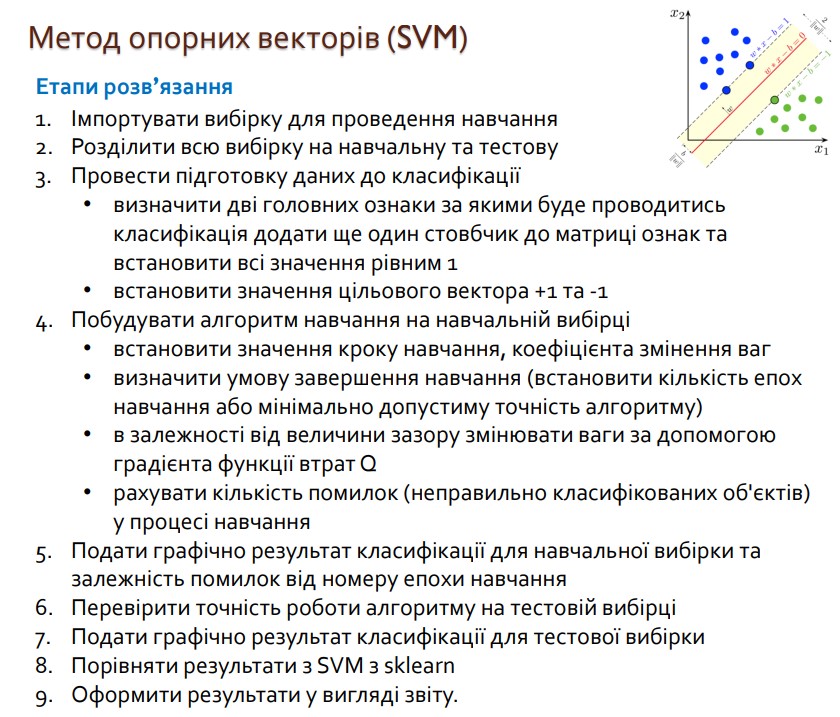
## Метод опорних векторів (SVM)[¶](" \l "%D0%9C%D0%B5%D1%82%D0%BE%D0%B4-%D0%BE%D)

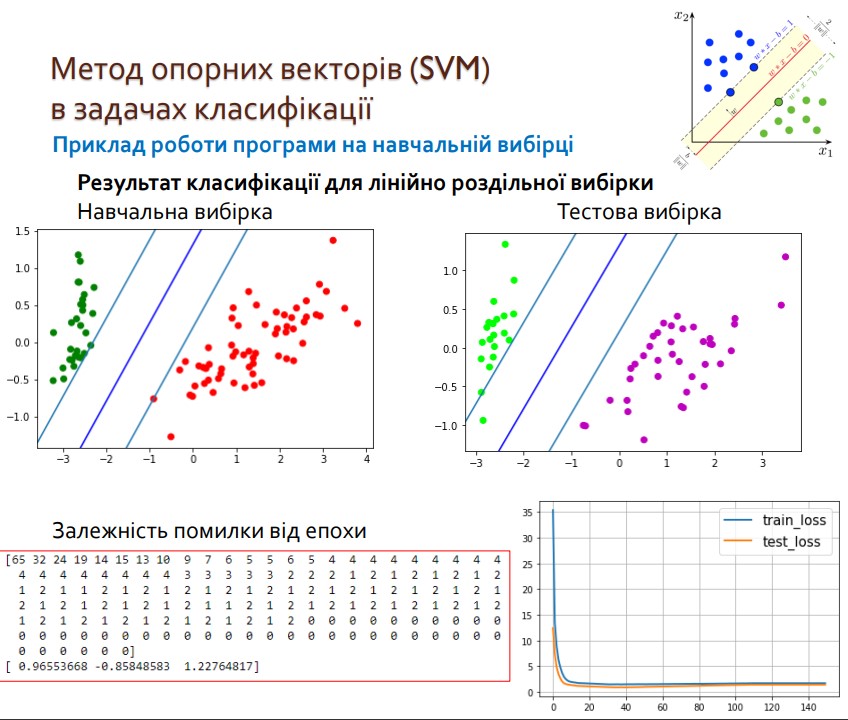
Лабораторна робота 3¶











* **Імпортуємо потрібні бібліотеки**:

In [1]:

import numpy as np

import pandas as pd

import warnings

warnings.filterwarnings('ignore')

import seaborn as sns

import matplotlib.pyplot as plt

import matplotlib.lines as mlines

plt.rcParams['figure.figsize'] = (8,6)

%matplotlib inline

from sklearn.datasets import load\_iris

from sklearn.decomposition import PCA

from sklearn.model\_selection import train\_test\_split

from sklearn import metrics

* **Визначаємо функції, що реалізовують Метод опорних векторів (svm):**

In [2]:

# функція для зображення лінії

def newline(p1, p2, color=None):

ax = plt.gca()

xmin, xmax = ax.get\_xbound()

if(p2[0] == p1[0]):

xmin = xmax = p1[0]

ymin, ymax = ax.get\_ybound()

else:

ymax = p1[1]+(p2[1]-p1[1])/(p2[0]-p1[0])\*(xmax-p1[0])

ymin = p1[1]+(p2[1]-p1[1])/(p2[0]-p1[0])\*(xmin-p1[0])

l = mlines.Line2D([xmin,xmax], [ymin,ymax], color=color)

ax.add\_line(l)

return l

In [3]:

def add\_bias\_feature(a):

a\_extended = np.zeros((a.shape[0],a.shape[1]+1))

a\_extended[:,:-1] = a

a\_extended[:,-1] = int(1)

return a\_extended

In [4]:

class CustomSVM(object):

\_\_class\_\_ = "CustomSVM"

\_\_doc\_\_ = """

Це реалізація алгоритму класифікації SVM (тільки для бінарної класифікації)

# Параметри

etha: float(default - 0.01)

Швидкість навчання, крок градієнта

alpha: float, (default - 0.1)

Параметр регуляризації в 0.5\*alpha\*||w||^2

epochs: int, (default - 200)

Кількість епох навчання

"""

def \_\_init\_\_(self, etha=0.01, alpha=0.1, epochs=200):

self.\_epochs = epochs

self.\_etha = etha

self.\_alpha = alpha

self.\_w = None

self.history\_w = []

self.train\_errors = None

self.val\_errors = None

self.train\_loss = None

self.val\_loss = None

def fit(self, X\_train, Y\_train, X\_val, Y\_val, verbose=False): #arrays: X; Y =-1,1

if len(set(Y\_train)) != 2 or len(set(Y\_val)) != 2:

raise ValueError("Кількість класів в Y не дорівнює 2!")

X\_train = add\_bias\_feature(X\_train)

X\_val = add\_bias\_feature(X\_val)

self.\_w = np.random.normal(loc=0, scale=0.05, size=X\_train.shape[1])

self.history\_w.append(self.\_w)

train\_errors = []

val\_errors = []

train\_loss\_epoch = []

val\_loss\_epoch = []

for epoch in range(self.\_epochs):

tr\_err = 0

val\_err = 0

tr\_loss = 0

val\_loss = 0

for i,x in enumerate(X\_train):

margin = Y\_train[i]\*np.dot(self.\_w,X\_train[i])

if margin >= 1: # классифицируем верно

self.\_w = self.\_w - self.\_etha\*self.\_alpha\*self.\_w/self.\_epochs

tr\_loss += self.soft\_margin\_loss(X\_train[i],Y\_train[i])

else: # классифицируем неверно или попадаем на полосу разделения при 0<m<1

self.\_w = self.\_w +\

self.\_etha\*(Y\_train[i]\*X\_train[i] - self.\_alpha\*self.\_w/self.\_epochs)

tr\_err += 1

tr\_loss += self.soft\_margin\_loss(X\_train[i],Y\_train[i])

self.history\_w.append(self.\_w)

for i,x in enumerate(X\_val):

val\_loss += self.soft\_margin\_loss(X\_val[i], Y\_val[i])

val\_err += (Y\_val[i]\*np.dot(self.\_w,X\_val[i])<1).astype(int)

if verbose:

print('epoch {}. Errors={}. Mean Hinge\_loss={}'\

.format(epoch,err,loss))

train\_errors.append(tr\_err)

val\_errors.append(val\_err)

train\_loss\_epoch.append(tr\_loss)

val\_loss\_epoch.append(val\_loss)

self.history\_w = np.array(self.history\_w)

self.train\_errors = np.array(train\_errors)

self.val\_errors = np.array(val\_errors)

self.train\_loss = np.array(train\_loss\_epoch)

self.val\_loss = np.array(val\_loss\_epoch)

def predict(self, X:np.array) -> np.array:

y\_pred = []

X\_extended = add\_bias\_feature(X)

for i in range(len(X\_extended)):

y\_pred.append(np.sign(np.dot(self.\_w,X\_extended[i])))

return np.array(y\_pred)

def hinge\_loss(self, x, y):

return max(0,1 - y\*np.dot(x, self.\_w))

def soft\_margin\_loss(self, x, y):

return self.hinge\_loss(x,y)+self.\_alpha\*np.dot(self.\_w, self.\_w)

* **Імпортуємо вибірку для проведення навчання**

In [5]:

iris = load\_iris()

X = iris.data

y = iris.target

In [6]:

print(np.shape(X))

print(np.shape(y))

(150, 4)

(150,)

In [7]:

dfIris = pd.DataFrame(data = np.c\_[X, y], columns = iris['feature\_names'] + ['target'])

dfIris

Out[7]:

|  | **sepal length (cm)** | **sepal width (cm)** | **petal length (cm)** | **petal width (cm)** | **target** |
| --- | --- | --- | --- | --- | --- |
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | 0.0 |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | 0.0 |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | 0.0 |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | 0.0 |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | 0.0 |
| **...** | ... | ... | ... | ... | ... |
| **145** | 6.7 | 3.0 | 5.2 | 2.3 | 2.0 |
| **146** | 6.3 | 2.5 | 5.0 | 1.9 | 2.0 |
| **147** | 6.5 | 3.0 | 5.2 | 2.0 | 2.0 |
| **148** | 6.2 | 3.4 | 5.4 | 2.3 | 2.0 |
| **149** | 5.9 | 3.0 | 5.1 | 1.8 | 2.0 |

150 rows × 5 columns

In [8]:

print(dfIris['target'].unique())

[0. 1. 2.]

* **Проводимо підготовку даних до класифікації:**

In [9]:

pca = PCA(n\_components=2)

X = pca.fit\_transform(X)

y = (y > 0).astype(int)\*2-1 # [0,1,2] --> [False,True,True] --> [0,1,1] --> [0,2,2] --> [-1,1,1]

In [10]:

dfIris2 = pd.DataFrame(data = np.c\_[X, y], columns = ['x1', 'x2'] + ['y'])

dfIris2

Out[10]:

|  | **x1** | **x2** | **y** |
| --- | --- | --- | --- |
| **0** | -2.684126 | 0.319397 | -1.0 |
| **1** | -2.714142 | -0.177001 | -1.0 |
| **2** | -2.888991 | -0.144949 | -1.0 |
| **3** | -2.745343 | -0.318299 | -1.0 |
| **4** | -2.728717 | 0.326755 | -1.0 |
| **...** | ... | ... | ... |
| **145** | 1.944110 | 0.187532 | 1.0 |
| **146** | 1.527167 | -0.375317 | 1.0 |
| **147** | 1.764346 | 0.078859 | 1.0 |
| **148** | 1.900942 | 0.116628 | 1.0 |
| **149** | 1.390189 | -0.282661 | 1.0 |

150 rows × 3 columns

In [11]:

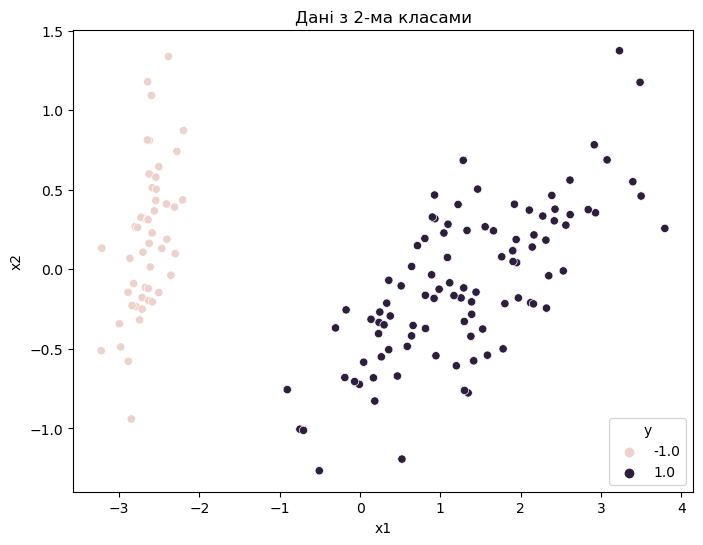
sns.scatterplot(x = dfIris2['x1'], y = dfIris2['x2'], hue = dfIris2['y'])

plt.title('Дані з 2-ма класами')

#plt.legend(bbox\_to\_anchor=(1,1), loc="upper left")

Out[11]:

Text(0.5, 1.0, 'Дані з 2-ма класами')



* **Розділяємо всю вибірку на навчальну та тестову. Навчаємо алгоритм на навчальній вибірці.**

In [12]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=2023)

In [13]:

# блок ініціалізації та навчання

svm = CustomSVM(etha=0.005, alpha=0.006, epochs=150)

svm.fit(X\_train, y\_train, X\_test, y\_test)

* **Подаємо графічно результат класифікації для навчальної вибірки та залежність помилок від номеру епохи навчання.**

In [14]:

df\_train = pd.DataFrame(data = np.c\_[X\_train, y\_train], columns = ['x1', 'x2'] + ['y'])

display(df\_train.head(5))

|  | **x1** | **x2** | **y** |
| --- | --- | --- | --- |
| **0** | 1.764346 | 0.078859 | 1.0 |
| **1** | -2.310256 | 0.391346 | -1.0 |
| **2** | -0.173925 | -0.254854 | 1.0 |
| **3** | 1.971531 | -0.179728 | 1.0 |
| **4** | 0.136429 | -0.314032 | 1.0 |

In [15]:

sns.scatterplot(x = df\_train['x1'], y = df\_train['x2'], hue = df\_train['y'])

# в w0\*x\_i[0]+w1\*x\_i[1]+w2\*1=0 по черзі

# підставляємо x\_i[0]=0, x\_i[1]=0

newline([0,-svm.\_w[2]/svm.\_w[1]],[-svm.\_w[2]/svm.\_w[0],0], 'blue')

# w0\*x\_i[0] + w1\*x\_i[1] + w2\*1 = 1

newline([0, 1 / svm.\_w[1]-svm.\_w[2]/svm.\_w[1]], [ 1 / svm.\_w[0]-svm.\_w[2]/svm.\_w[0], 0], 'pink')

# w0\*x\_i[0] + w1\*x\_i[1] + w2\*1 = -1

newline([0, -1 / svm.\_w[1]-svm.\_w[2]/svm.\_w[1]], [-1 / svm.\_w[0]-svm.\_w[2]/svm.\_w[0], 0], 'gold')

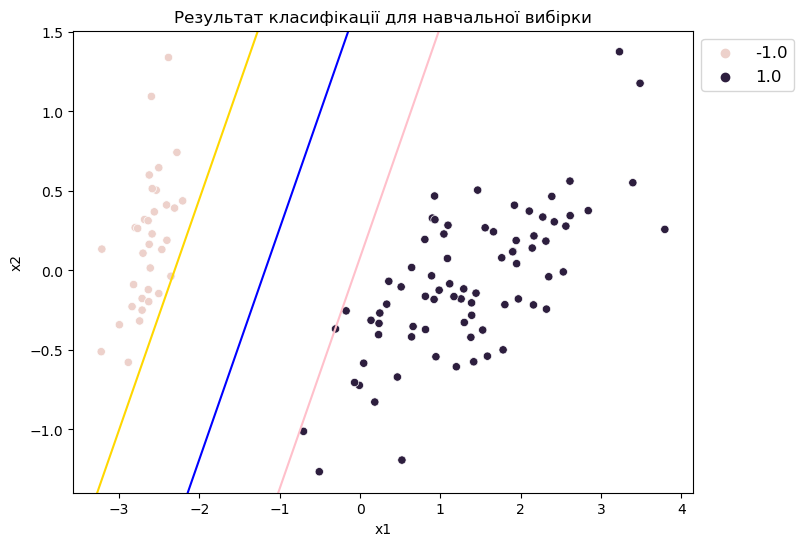
plt.xlabel("x1")

plt.ylabel("x2")

plt.title("Результат класифікації для навчальної вибірки" )

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

plt.show()



In [16]:

print('Кількість помилок у кожну епоху:\n', svm.train\_errors, '\n')

print('Значення ваг або коефіцієнтів для прямої w0\*x\_i[0]+w1\*x\_i[1]+w2=0 :\n', svm.\_w)

plt.plot(svm.train\_errors, linewidth=2, label='train')

plt.xlabel("Епоха")

plt.ylabel("Кількість помилок")

plt.title("Кількість помилок у кожну епоху на тренувальній вибірці" )

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

plt.show()

Кількість помилок у кожну епоху:

[84 36 26 18 15 14 13 8 8 5 5 5 6 4 4 4 4 4 4 4 4 4 4 4

4 4 4 4 4 3 3 3 1 2 1 1 2 1 1 1 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

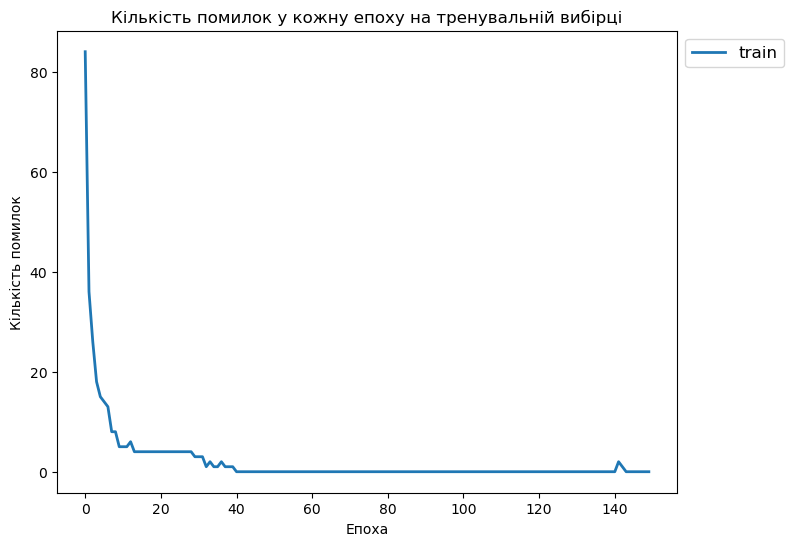
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 1 0

0 0 0 0 0 0]

Значення ваг або коефіцієнтів для прямої w0\*x\_i[0]+w1\*x\_i[1]+w2=0 :

[ 0.88713539 -0.61137468 1.0493689 ]



* **Перевіряємо точність роботи алгоритму на тестовій вибірці:**

In [17]:

# Прогнозуємо значення на тестовій вибірці

y\_predict = svm.predict(X\_test)

In [18]:

# Вимірюємо точність прогнозів

print("Accuracy:", metrics.accuracy\_score(y\_test, y\_predict))

#print("Accuracy score: %.3f" %metrics.accuracy\_score(y\_test, y\_predict))

Accuracy: 1.0

* **Подаємо графічно результат класифікації для тестової вибірки**

In [19]:

df\_test = pd.DataFrame(data = np.c\_[X\_test, y\_test], columns = ['x1', 'x2'] + ['y'])

display(df\_test.head(5))

|  | **x1** | **x2** | **y** |
| --- | --- | --- | --- |
| **0** | 2.123609 | -0.209729 | 1.0 |
| **1** | 0.714853 | 0.149056 | 1.0 |
| **2** | 0.299001 | -0.348898 | 1.0 |
| **3** | 2.932587 | 0.355500 | 1.0 |
| **4** | 1.332024 | 0.244441 | 1.0 |

In [20]:

sns.scatterplot(x = df\_test['x1'], y = df\_test['x2'], hue = df\_test['y'])

#d = {-1:'yellow', 1:'blue'}

#plt.scatter(X\_test[:,0], X\_test[:,1], c=[d[y] for y in y\_test], label = d)

# в w0\*x\_i[0] + w1\*x\_i[1] + w2\*1 = 0 по черзі

# підставляємо x\_i[0] = 0, x\_i[1] = 0

newline([0, -svm.\_w[2] / svm.\_w[1]], [-svm.\_w[2]/svm.\_w[0], 0], 'blue')

# w0\*x\_i[0] + w1\*x\_i[1] + w2\*1 = 1

newline([0, 1 / svm.\_w[1]-svm.\_w[2]/svm.\_w[1]], [ 1 / svm.\_w[0]-svm.\_w[2]/svm.\_w[0], 0], 'pink')

# w0\*x\_i[0] + w1\*x\_i[1] + w2\*1 = -1

newline([0, -1 / svm.\_w[1]-svm.\_w[2]/svm.\_w[1]], [-1 / svm.\_w[0]-svm.\_w[2]/svm.\_w[0], 0], 'gold')

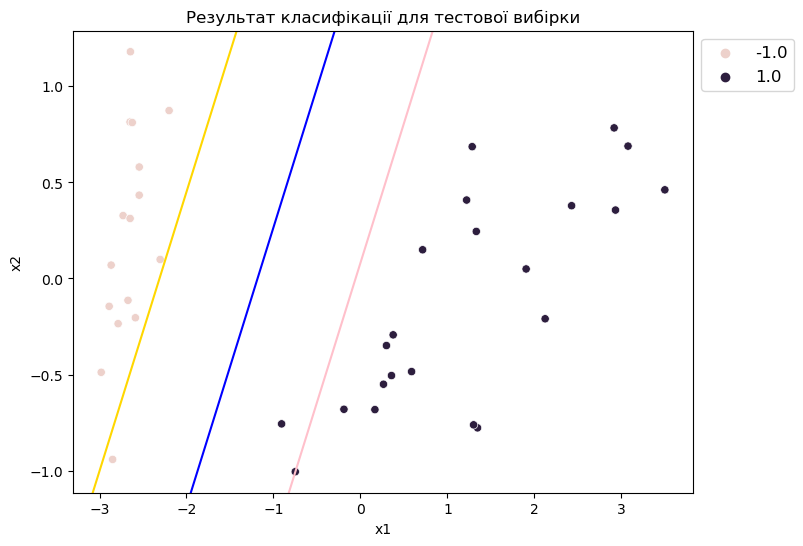
plt.xlabel("x1")

plt.ylabel("x2")

plt.title("Результат класифікації для тестової вибірки" )

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

plt.show()



* Побудуємо функції втрат навчальної та тестовоої вибірки.

In [21]:

plt.plot(svm.train\_loss, linewidth=2, label='train\_loss')

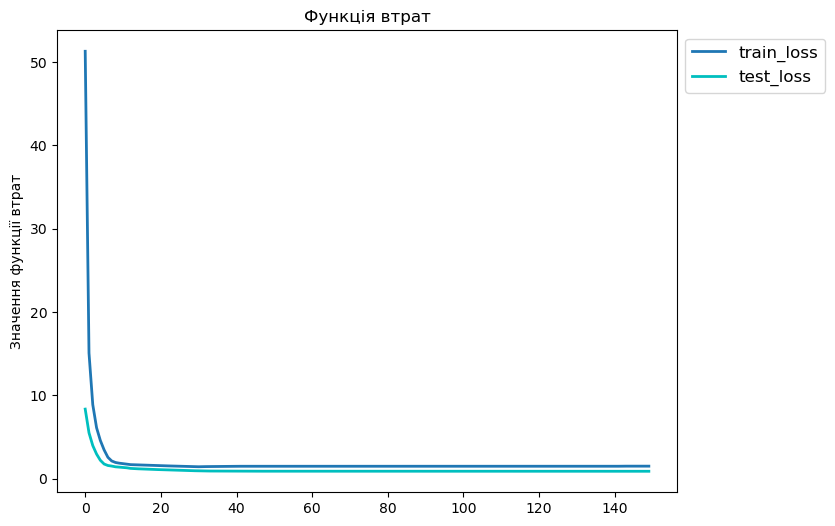
plt.plot(svm.val\_loss, linewidth=2, label='test\_loss', c = 'c')

plt.ylabel("Значення функції втрат")

plt.title("Функція втрат" )

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

plt.show()



* **Порівняємо результати з SVM з sklearn.**

In [22]:

#Import svm model

from sklearn import svm

# Створюємо svm Класифікатор

clf = svm.SVC(kernel='linear') # Linear Kernel

# Навчаємо модель за допомогою навчальних наборів

clf.fit(X\_train, y\_train)

# Прогнозуємо значення цільової змінної для тестового набору даних

y\_pred = clf.predict(X\_test)

In [23]:

# Вимірюємо точність прогнозів

print("Accuracy:", metrics.accuracy\_score(y\_test, y\_pred))

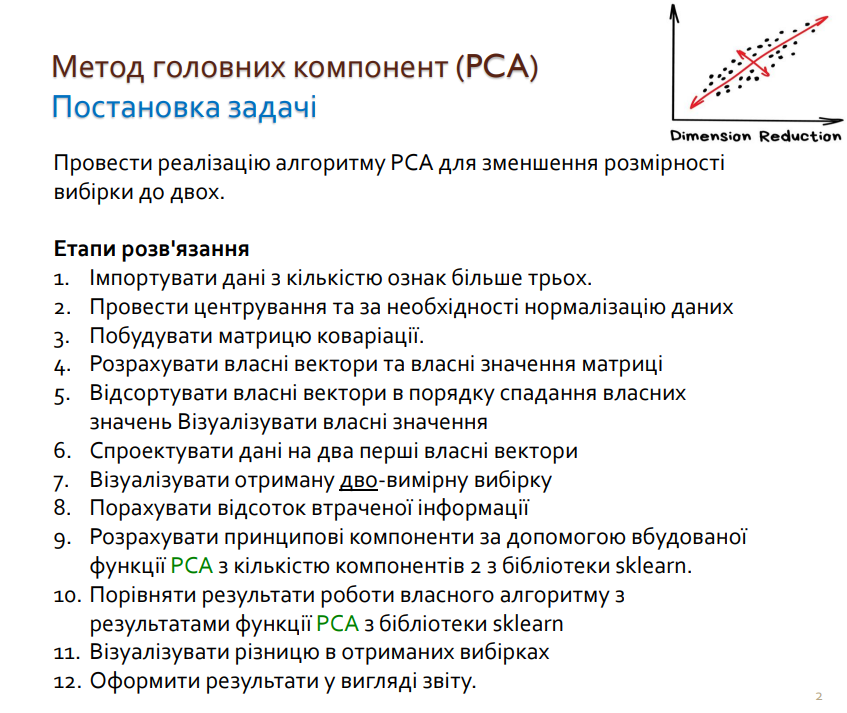
#print("Accuracy score: %.3f" %metrics.accuracy\_score(y\_test, y\_predict))

Accuracy: 1.0

Отже, результати співпадають.

## Метод головних компонент (PCA)[¶](#%D0%9C%D0%B5%D1%82%D0%BE%D0%B4-%D0%B3%D)

Лабораторна робота 4¶





* **Імпортуємо потрібні бібліотеки:**

In [1]:

#!pip install seaborn

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

%matplotlib inline

In [2]:

import warnings

warnings.filterwarnings('ignore')

In [3]:

from sklearn.datasets import load\_iris

from sklearn.decomposition import PCA

from sklearn.model\_selection import train\_test\_split

#from sklearn import metrics

**1. Імпортувати дані з кількістю ознак більше трьох.**

In [4]:

iris = load\_iris()

features = iris.data

target = iris.target

In [5]:

print(np.shape(features))

print(np.shape(target))

(150, 4)

(150,)

In [6]:

dfIris = pd.DataFrame(data = np.c\_[features, target], columns = iris['feature\_names'] + ['target'])

dfIris

Out[6]:

|  | **sepal length (cm)** | **sepal width (cm)** | **petal length (cm)** | **petal width (cm)** | **target** |
| --- | --- | --- | --- | --- | --- |
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | 0.0 |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | 0.0 |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | 0.0 |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | 0.0 |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | 0.0 |
| **...** | ... | ... | ... | ... | ... |
| **145** | 6.7 | 3.0 | 5.2 | 2.3 | 2.0 |
| **146** | 6.3 | 2.5 | 5.0 | 1.9 | 2.0 |
| **147** | 6.5 | 3.0 | 5.2 | 2.0 | 2.0 |
| **148** | 6.2 | 3.4 | 5.4 | 2.3 | 2.0 |
| **149** | 5.9 | 3.0 | 5.1 | 1.8 | 2.0 |

150 rows × 5 columns

In [7]:

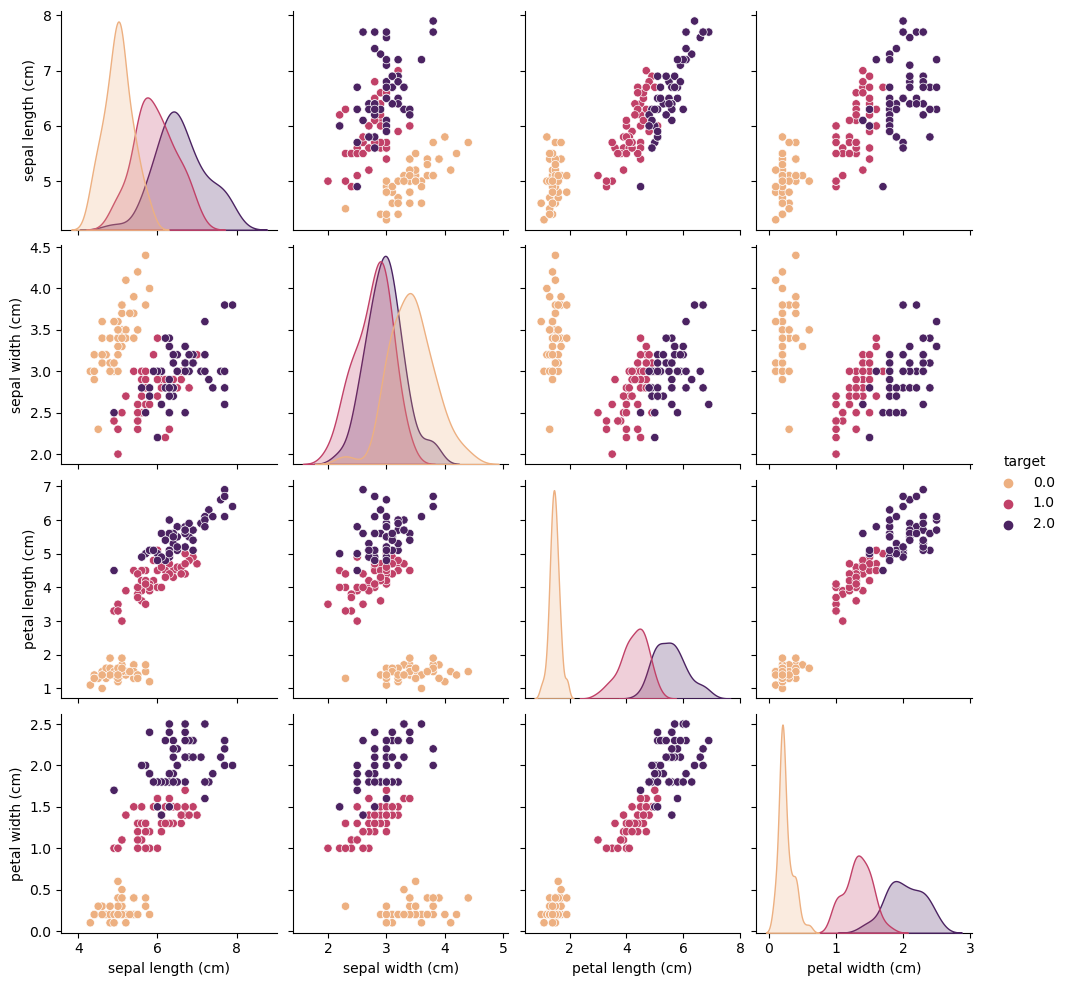
print(dfIris['target'].unique())

[0. 1. 2.]

In [8]:

sns.pairplot(dfIris, hue ='target', palette = "flare")

plt.show()



Оскiльки перший клас добре видiляється вiд iнших двох залишимо у вибiрцi лише данi для другого та третього класiв. Отримуємо вибiрку X = [ℓ×n], де ℓ – кiлькiсть об’єктiв, n – кiлькiсть ознак (для даного прикладу n = 4).

In [9]:

dfIris2 = dfIris.query("`target` != 0")

dfIris2

Out[9]:

|  | **sepal length (cm)** | **sepal width (cm)** | **petal length (cm)** | **petal width (cm)** | **target** |
| --- | --- | --- | --- | --- | --- |
| **50** | 7.0 | 3.2 | 4.7 | 1.4 | 1.0 |
| **51** | 6.4 | 3.2 | 4.5 | 1.5 | 1.0 |
| **52** | 6.9 | 3.1 | 4.9 | 1.5 | 1.0 |
| **53** | 5.5 | 2.3 | 4.0 | 1.3 | 1.0 |
| **54** | 6.5 | 2.8 | 4.6 | 1.5 | 1.0 |
| **...** | ... | ... | ... | ... | ... |
| **145** | 6.7 | 3.0 | 5.2 | 2.3 | 2.0 |
| **146** | 6.3 | 2.5 | 5.0 | 1.9 | 2.0 |
| **147** | 6.5 | 3.0 | 5.2 | 2.0 | 2.0 |
| **148** | 6.2 | 3.4 | 5.4 | 2.3 | 2.0 |
| **149** | 5.9 | 3.0 | 5.1 | 1.8 | 2.0 |

100 rows × 5 columns

In [10]:

dfIris2['target'].unique()

Out[10]:

array([1., 2.])

In [11]:

X = np.array((dfIris2.drop(columns='target')).values)

X[:5]

Out[11]:

array([[7. , 3.2, 4.7, 1.4],

[6.4, 3.2, 4.5, 1.5],

[6.9, 3.1, 4.9, 1.5],

[5.5, 2.3, 4. , 1.3],

[6.5, 2.8, 4.6, 1.5]])

In [12]:

#y = np.array((dfIris2.drop(columns=['sepal length (cm)','sepal width (cm)','petal length (cm)','petal width (cm)'])).values)

#y

**2. Провести центрування та за необхідності нормалізацію даних.**

In [13]:

from sklearn.preprocessing import StandardScaler

# масштабування даних

x\_scaled = StandardScaler().fit\_transform(X)

In [14]:

X[0]

Out[14]:

array([7. , 3.2, 4.7, 1.4])

In [15]:

x\_scaled[0]

Out[15]:

array([ 1.11900931, 0.99068792, -0.25077906, -0.65303909])

In [16]:

x\_matrix = np.array([x\_scaled[:, 0], x\_scaled[:, 1], x\_scaled[:, 2], x\_scaled[:, 3]])

**3. Побудувати матрицю коваріації.**

In [17]:

covmat = np.cov(x\_matrix)

covmat

Out[17]:

array([[1.01010101, 0.55944934, 0.83684713, 0.59970648],

[0.55944934, 1.01010101, 0.52505282, 0.57192176],

[0.83684713, 0.52505282, 1.01010101, 0.83166424],

[0.59970648, 0.57192176, 0.83166424, 1.01010101]])

In [18]:

np.shape(covmat)

Out[18]:

(4, 4)

**4. Розрахувати власні вектори та власні значення матриці.**

In [19]:

w, v = np.linalg.eig(covmat)

print('Власні значення (вектор):\n', w)

Власні значення (вектор):

[2.98778909 0.08206412 0.40983199 0.56071884]

In [20]:

print('Власні вектори(масив):\n', v)

Власні вектори(масив):

[[-0.5071303 -0.4623842 -0.69307315 0.22061124]

[-0.43474966 0.13624797 -0.05558682 -0.88844771]

[-0.54369524 0.74828559 0.01943532 0.37958719]

[-0.5081408 -0.45574777 0.71845806 0.13380926]]

Відповідність власних значень та векторів:

* w[0] відповідає 0-му стовпчику v (v[:,0]),
* w[1] відповідає 1-му стовпчику v (v[:,1]),
* w[i] відповідає i-му стовпчику v (v[:,i]).

In [21]:

w[0]

Out[21]:

2.9877890906474667

In [22]:

v[:,0]

Out[22]:

array([-0.5071303 , -0.43474966, -0.54369524, -0.5081408 ])

Перевірка $ Au = w\_1 u:$

In [23]:

u = v[:, 1]

print('u ', u)

w1 = w[1]

print('w1 ', w1, '\n')

print('𝐴𝑢 \t', np.dot(covmat, u))

print('𝑤1 𝑢 \t', w1\*u)

u [-0.4623842 0.13624797 0.74828559 -0.45574777]

w1 0.08206411695786031

𝐴𝑢 [-0.03794515 0.01118107 0.0614074 -0.03740054]

𝑤1 𝑢 [-0.03794515 0.01118107 0.0614074 -0.03740054]

**5. Відсортувати власні вектори в порядку спадання власних значень. Візуалізувати власні значення.**

In [24]:

data = [('1', w[0]),

('2', w[1]),

('3', w[2]),

('4', w[3])]

df = pd.DataFrame(data, columns =['№ ознаки', 'Власне значення'])

df

Out[24]:

|  | **№ ознаки** | **Власне значення** |
| --- | --- | --- |
| **0** | 1 | 2.987789 |
| **1** | 2 | 0.082064 |
| **2** | 3 | 0.409832 |
| **3** | 4 | 0.560719 |

In [25]:

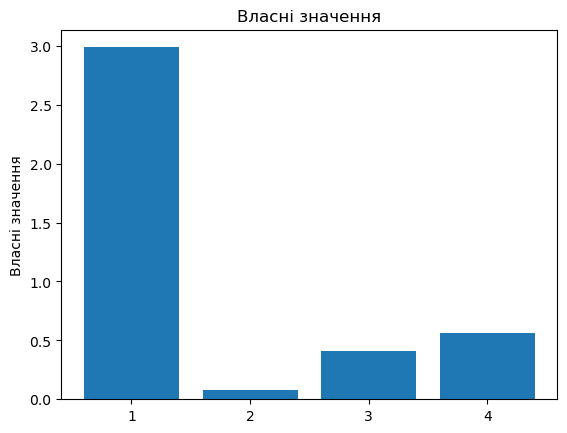
plt.bar(df['№ ознаки'], df['Власне значення'])

plt.ylabel('Власні значення')

plt.title('Власні значення')

Out[25]:

Text(0.5, 1.0, 'Власні значення')



In [26]:

df\_sort = df.sort\_values(by=['Власне значення'], ascending=False)

display(df\_sort)

plt.bar(df\_sort['№ ознаки'], df\_sort['Власне значення'])

plt.ylabel('Власні значення')

plt.title('Власні значення (відсортовані за спаданням)')

|  | **№ ознаки** | **Власне значення** |
| --- | --- | --- |
| **0** | 1 | 2.987789 |
| **3** | 4 | 0.560719 |
| **2** | 3 | 0.409832 |
| **1** | 2 | 0.082064 |

Out[26]:

Text(0.5, 1.0, 'Власні значення (відсортовані за спаданням)')



**6. Спроектувати дані на два перші власні вектори. Встановимо кiлькiсть головних компонент $nPC = 2$. Проектуємо данi на першi два власнi вектори матрицi коварiацiї, що характеризуються найбiльшими власними значеннями.**

Множимо вихiдну матрицюданих розмiру $[ℓ × n]$ на транспоновану матрицю двох власних векторiв розмiру $[n × nPC]$ для отримання нової матрицi даних розмiром $[ℓ × nPC]$. Вiзуалiзуємо отриманi данi.

In [27]:

np.shape(x\_scaled) # [ℓ × 𝑛]

Out[27]:

(100, 4)

In [28]:

index1, index2 = (df\_sort['№ ознаки'].head(2)).values

index1, index2 = int(index1), int(index2)

print(index1, index2)

1 4

In [29]:

eigenvectors = np.array([v[index1-1], v[index2-1]])

vT = eigenvectors.transpose()

print(np.shape(vT)) # [𝑛 × 𝑛𝑃𝐶]

print(vT)

(4, 2)

[[-0.5071303 -0.5081408 ]

[-0.4623842 -0.45574777]

[-0.69307315 0.71845806]

[ 0.22061124 0.13380926]]

In [30]:

x\_new = np.dot(x\_scaled, vT)

np.shape(x\_new) # [ℓ × 𝑛𝑃𝐶]

Out[30]:

(100, 2)

**7. Візуалізувати отриману дво-вимірну вибірку.**

In [31]:

lab = dfIris2['target'].unique()

plt.plot(x\_new, 'o', markersize = 5,

#markerfacecolor='white',

#markeredgecolor='gray',

markeredgewidth = 2, label=lab)

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.title('Проекція даних на два головних вектори')

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

Out[31]:

<matplotlib.legend.Legend at 0x220a26b86a0>



**8. Порахувати відсоток втраченої інформації, використовуючи власнi значення матрицi коварiацiй при зменшеннi розмiрностi з чотирьох ознак до двох:**

In [32]:

df\_sort

Out[32]:

|  | **№ ознаки** | **Власне значення** |
| --- | --- | --- |
| **0** | 1 | 2.987789 |
| **3** | 4 | 0.560719 |
| **2** | 3 | 0.409832 |
| **1** | 2 | 0.082064 |

In [33]:

l1 = df\_sort['Власне значення'].values[0]

l2 = df\_sort['Власне значення'].values[1]

print('l1 {:.6f}\nl2 {:.6f}'.format(l1, l2))

l\_sum = np.sum(df\_sort['Власне значення'])

delt\_I = 1 - ((l1+l2)/l\_sum)

delt\_I \*= 100

print('Відсоток втраченої інформації:\n{:.3} %'.format(delt\_I))

l1 2.987789

l2 0.560719

Відсоток втраченої інформації:

12.2 %

**9. Розрахувати принципові компоненти за допомогою вбудованої функції PCA з кількістю компонентів 2 з бібліотеки sklearn.**

In [34]:

pca = PCA(n\_components = 2)

XPCAreduced = pca.fit\_transform(np.transpose(x\_matrix))

**10. Порівняти результати роботи власного алгоритму з результатами функції PCA з бібліотеки sklearn.**

In [35]:

print ('Sklearn reduced X [:5]: \n', XPCAreduced[:5])

Sklearn reduced X [:5]:

[[ 0.53000157 -0.81588358]

[ 0.05648712 -1.0773472 ]

[ 0.5744019 -0.4569081 ]

[-2.38876911 0.74233881]

[-0.32567545 0.07569688]]

In [36]:

x\_new[:5]

Out[36]:

array([[-0.99582149, -1.28767501],

[-0.31350845, -0.96865302],

[-0.89581604, -0.86638657],

[ 1.95293489, 0.46302455],

[ 0.08385608, -0.40762341]])

In [37]:

print('x\_scaled \t', np.shape(x\_scaled)) # [ℓ × 𝑛]

print('x\_new \t', np.shape(x\_new)) # [ℓ × 𝑛𝑃𝐶]

print('XPCAreduced \t', np.shape(XPCAreduced)) # [ℓ × 𝑛𝑃𝐶]

x\_scaled (100, 4)

x\_new (100, 2)

XPCAreduced (100, 2)

**11. Візуалізувати різницю в отриманих вибірках.**

In [38]:

difference = (XPCAreduced-x\_new)

In [39]:

lab = dfIris2['target'].unique()

plt.plot(difference, 's', markersize = 5,

#markerfacecolor='white',

#markeredgecolor='gray',

markeredgewidth = 2, label=lab)

plt.xlabel('delt PC1')

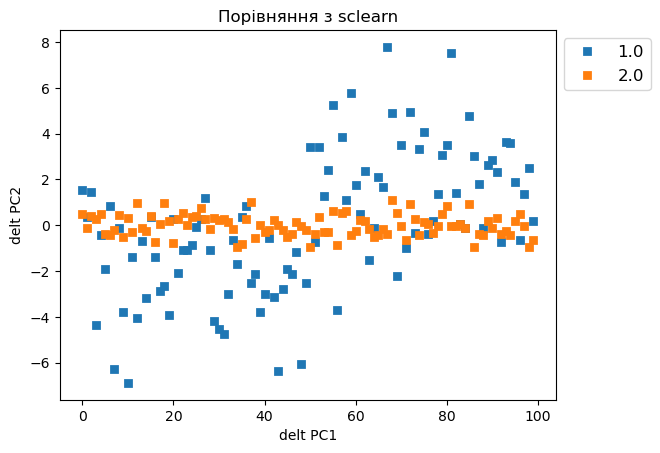
plt.ylabel('delt PC2')

plt.title('Порівняння з sclearn')

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

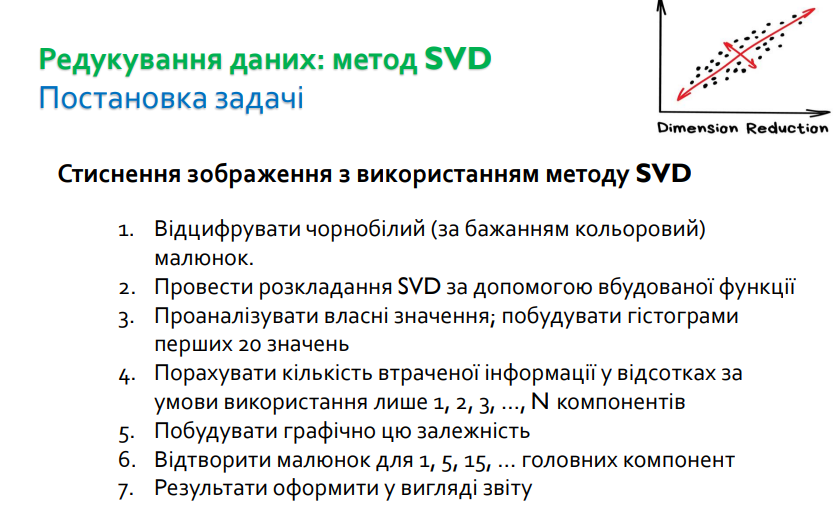
Out[39]:

<matplotlib.legend.Legend at 0x220a2745700>



## Редукування даних: метод SVD[¶](#%D0%A0%D0%B5%D0%B4%D1%83%D0%BA%D1%83%D0)

Лабораторна робота 5¶



In [1]:

!pip install opencv-python

!pip install imutils

Requirement already satisfied: opencv-python in c:\users\admin\anaconda3\lib\site-packages (4.7.0.72)

Requirement already satisfied: numpy>=1.17.0 in c:\users\admin\anaconda3\lib\site-packages (from opencv-python) (1.22.4)

Requirement already satisfied: imutils in c:\users\admin\anaconda3\lib\site-packages (0.5.4)

In [2]:

%matplotlib inline

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib.image as mpimg

from matplotlib.image import imread

import os

# Import opencv

import cv2

In [3]:

import warnings

warnings.filterwarnings('ignore')

* **1**

In [4]:

name\_img = 'beach.jpg'

img\_orig = mpimg.imread(name\_img)

imgplot = plt.imshow(img\_orig)



* **2-3**

In [5]:

def fun\_matrixs(name\_img, components):

# Завантажте вхідне зображення і перетвориіть в чорнобіле

img = cv2.imread(name\_img, 0)

original\_shape = img.shape

print('Розміри початкового зображення:', original\_shape)

U, S, V = np.linalg.svd(img)

print('\nРозміри матриць отриманих з np.linalg.svd(...):\n U.shape, S.shape, V.shape')

print(U.shape, S.shape, V.shape)

first = [min(np.shape(img))]

#comps = [first, 200, 50, 15, 5, 1]

comps = first + components

print('\nСписок числа компонент:', comps)

matr = []

for i in range(len(comps)):

low\_rank = U[:, :comps[i]] @ np.diag(S[:comps[i]]) @ V[:comps[i], :]

#print(np.shape(low\_rank))

matr.append(low\_rank)

return matr

In [6]:

components0 = [200, 100, 50, 25, 15, 10, 5, 4, 3, 2, 1]

matr0 = fun\_matrixs(name\_img, components0)

print(np.shape(matr0))

Розміри початкового зображення: (638, 960)

Розміри матриць отриманих з np.linalg.svd(...):

U.shape, S.shape, V.shape

(638, 638) (638,) (960, 960)

Список числа компонент: [638, 200, 100, 50, 25, 15, 10, 5, 4, 3, 2, 1]

(12, 638, 960)

In [7]:

matr0\_ = matr0[0]

covmat = np.cov(matr0\_)

w, v = np.linalg.eig(covmat)

In [8]:

d2 = {'Власні значення': w[:20]}

df1 = pd.DataFrame(d2)

display(df1.head())

|  | **Власні значення** |
| --- | --- |
| **0** | 310432.929103 |
| **1** | 116198.452046 |
| **2** | 71619.792069 |
| **3** | 21235.600645 |
| **4** | 16093.473605 |

In [9]:

numbers = np.arange(20)

plt.bar(numbers, height=df1['Власні значення'])

plt.xticks(numbers)

plt.title('Гістограма перших 20 власних значень')

plt.xlabel('Номер власного вектору')

plt.ylabel('Власні значення коваріаційної матриці')

Out[9]:

Text(0, 0.5, 'Власні значення коваріаційної матриці')



* **4**

In [10]:

def plot\_components(name\_img):

# Завантажте вхідне зображення і перетвориіть в чорнобіле

img = cv2.imread(name\_img, 0)

original\_shape = img.shape

print('Розміри початкового зображення:', original\_shape)

# Отримання svd

U, S, V = np.linalg.svd(img)

print('\nРозміри матриць отриманих з np.linalg.svd(...):\n U.shape, S.shape, V.shape')

print(U.shape, S.shape, V.shape)

# зображення з різною кількістю компонентів

first = min(np.shape(img))

components = [first, 200, 50, 15, 5, 4, 3, 2, 1]

print('\nСписок числа компонент:', components, '\n')

plt.figure(figsize = (16, 8))

for i in range(len(components)):

low\_rank = U[:, :components[i]] @ np.diag(S[:components[i]]) @ V[:components[i], :]

return components

components = plot\_components(name\_img)

Розміри початкового зображення: (638, 960)

Розміри матриць отриманих з np.linalg.svd(...):

U.shape, S.shape, V.shape

(638, 638) (638,) (960, 960)

Список числа компонент: [638, 200, 50, 15, 5, 4, 3, 2, 1]

<Figure size 1600x800 with 0 Axes>

In [11]:

x = components[::-1]

percentage = [(1 - x[i]/x[-1])\*100 for i in range(len(x))]

print('Втрати інформації')

for i in percentage:

print(round(i,3), '%')

Втрати інформації

99.843 %

99.687 %

99.53 %

99.373 %

99.216 %

97.649 %

92.163 %

68.652 %

0.0 %

* **5**

In [12]:

plt.figure(figsize = (6, 5))

plt.plot(x, percentage, 'o--', label = 'Втрати інформації %')

plt.title('Втрати інформації %')

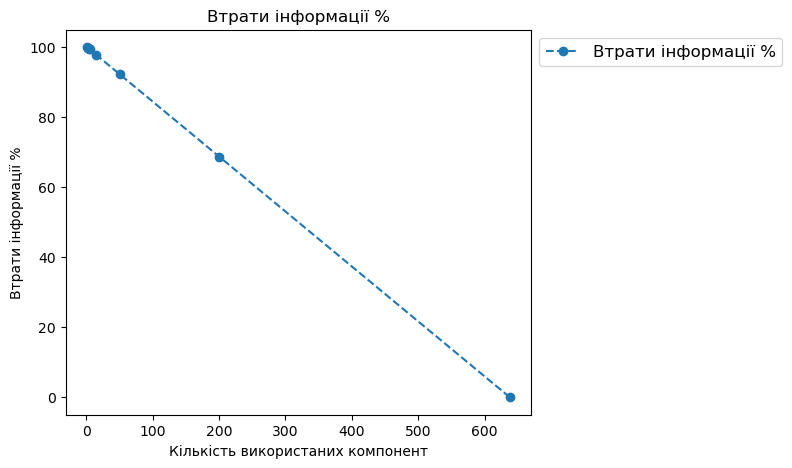
plt.xlabel('Кількість використаних компонент')

plt.ylabel('Втрати інформації %')

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

Out[12]:

<matplotlib.legend.Legend at 0x1d82b7c14c0>



* **6**

In [13]:

def plot\_n\_comp(name\_img):

# Завантажте вхідне зображення і перетвориіть в чорнобіле

img = cv2.imread(name\_img, 0)

original\_shape = img.shape

print('Розміри початкового зображення:', original\_shape)

# Отримання svd

U, S, V = np.linalg.svd(img)

print('\nРозміри матриць отриманих з np.linalg.svd(...):\n U.shape, S.shape, V.shape')

print(U.shape, S.shape, V.shape)

# зображення з різною кількістю компонентів

first = min(np.shape(img))

components = [first, 300, 75, 15, 5, 1]

print('\nСписок числа компонент:', components, '\n')

plt.figure(figsize = (16, 8))

for i in range(len(components)):

low\_rank = U[:, :components[i]] @ np.diag(S[:components[i]]) @ V[:components[i], :]

if(i == 0):

plt.subplot(2, 3, i+1), plt.imshow(low\_rank, cmap = 'gray'), plt.axis('off'), \

plt.title("Початкове зображення з n\_components =" + str(components[i]))

else:

plt.subplot(2, 3, i+1), plt.imshow(low\_rank, cmap = 'gray'), plt.axis('off'), \

plt.title("n\_components =" + str(components[i]))

#matr.append(low\_rank)

#print()

plot\_n\_comp(name\_img)

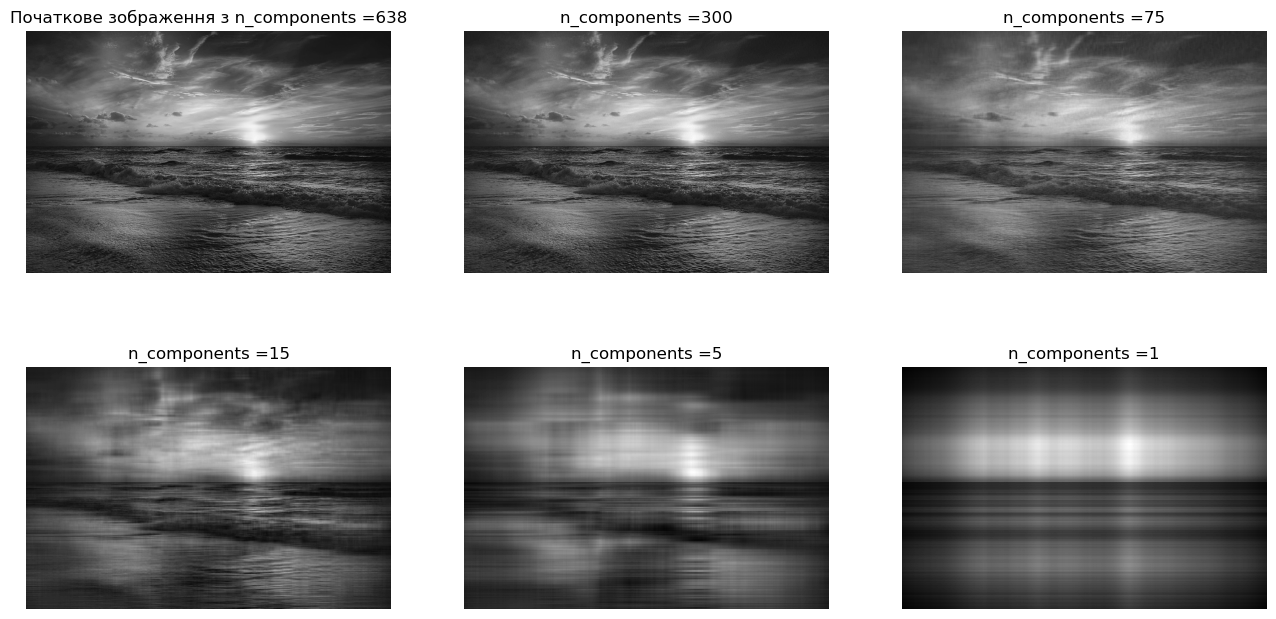
Розміри початкового зображення: (638, 960)

Розміри матриць отриманих з np.linalg.svd(...):

U.shape, S.shape, V.shape

(638, 638) (638,) (960, 960)

Список числа компонент: [638, 300, 75, 15, 5, 1]



## Метод k-середнiх (k-Means)[¶](" \l "%D0%9C%D0%B5%D1%82%D0%BE%D0%B4-k-%D1%81)

Лабораторна робота 6¶



In [1]:

import warnings

warnings.filterwarnings('ignore')

In [2]:

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

%matplotlib inline

1. Iмпортувати/згенерувати розмiченi/нерозмiченi данi

In [3]:

blobs = pd.read\_csv('kmeans\_blobs.csv')

colnames = list(blobs.columns[1:-1])

blobs.head()

Out[3]:

|  | **ID** | **x** | **y** | **cluster** |
| --- | --- | --- | --- | --- |
| **0** | 0 | 24.412 | 32.932 | 2 |
| **1** | 1 | 35.190 | 12.189 | 1 |
| **2** | 2 | 26.288 | 41.718 | 2 |
| **3** | 3 | 0.376 | 15.506 | 0 |
| **4** | 4 | 26.116 | 3.963 | 1 |

In [4]:

customcmap = ListedColormap(["crimson", "mediumblue", "darkmagenta"])

fig, ax = plt.subplots(figsize=(8, 6))

plt.scatter(x=blobs['x'], y=blobs['y'], s=50,

c=blobs['cluster'].astype('category'),

cmap = customcmap)

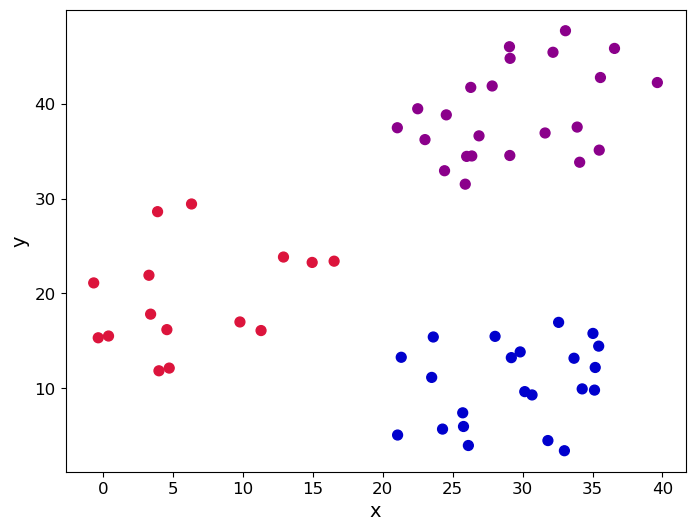
ax.set\_xlabel(r'x', fontsize=14)

ax.set\_ylabel(r'y', fontsize=14)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.show()



3. Провести кластеризацiю методом k-means при фiксованому значеннi кiлькостi кластерiв.

4. Вiзуалiзувати отриманi результати.

Кроки 1 і 2 - Визначення та ініціювання центроїдів.

In [5]:

def initiate\_centroids(k, dset):

'''

Вибрати k точок даних як центроїди

k: кількість центроїдів

dset: pandas dataframe

'''

centroids = dset.sample(k)

return centroids

np.random.seed(42)

k=3

df = blobs[['x','y']]

centroids = initiate\_centroids(k, df)

centroids

Out[5]:

|  | **x** | **y** |
| --- | --- | --- |
| **0** | 24.412 | 32.932 |
| **5** | 25.893 | 31.515 |
| **36** | 26.878 | 36.609 |

Крок 3 - Розрахуйте відстань.

In [6]:

def rsserr(a,b):

# Обчислити квадратний корінь з суми похибок. a та b - numpy масиви

return np.square(np.sum((a-b)\*\*2))

Крок 4 - Призначте центроїди.

In [7]:

def centroid\_assignation(dset, centroids):

'''

Маючи датафрейм `dset` і набір `centroids`, ми призначаємо кожній

точці даних у `dset` призначаємо центроїд.

- dset - pandas датафрейм зі спостереженнями

- centroids - фрейм даних з центроїдами

'''

k = centroids.shape[0]

n = dset.shape[0]

assignation = []

assign\_errors = []

for obs in range(n):

# Похибка оцінки

all\_errors = np.array([])

for centroid in range(k):

err = rsserr(centroids.iloc[centroid, :], dset.iloc[obs,:])

all\_errors = np.append(all\_errors, err)

# найближчий центроїд і помилку

nearest\_centroid = np.where(all\_errors==np.amin(all\_errors))[0].tolist()[0]

nearest\_centroid\_error = np.amin(all\_errors)

assignation.append(nearest\_centroid)

assign\_errors.append(nearest\_centroid\_error)

return assignation, assign\_errors

In [8]:

df['centroid'], df['error'] = centroid\_assignation(df, centroids)

df.head()

Out[8]:

|  | **x** | **y** | **centroid** | **error** |
| --- | --- | --- | --- | --- |
| **0** | 24.412 | 32.932 | 0 | 0.000000 |
| **1** | 35.190 | 12.189 | 1 | 211534.211314 |
| **2** | 26.288 | 41.718 | 2 | 699.601495 |
| **3** | 0.376 | 15.506 | 0 | 776856.744109 |
| **4** | 26.116 | 3.963 | 1 | 576327.599678 |

In [9]:

fig, ax = plt.subplots(figsize=(8, 6))

plt.scatter(df.iloc[:,0], df.iloc[:,1], marker = 'o',

c=df['centroid'].astype('category'),

cmap = customcmap, s=80, alpha=0.5)

plt.scatter(centroids.iloc[:,0], centroids.iloc[:,1],

marker = 's', s=200, c=[0, 1, 2],

cmap = customcmap)

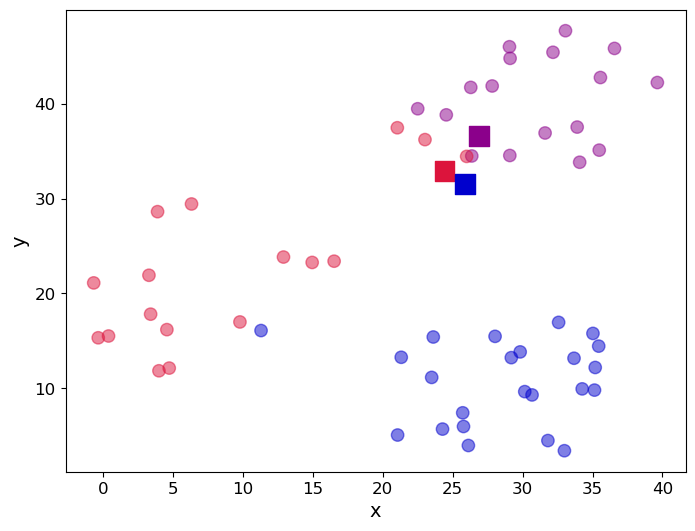
ax.set\_xlabel(r'x', fontsize=14)

ax.set\_ylabel(r'y', fontsize=14)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.show()



In [10]:

print("Сумарна помилка is {0:.2f}".format(df['error'].sum()))

Сумарна помилка is 11927659.01

Крок 5 - Оновлення розташування центроїда.

In [11]:

centroids = df.groupby('centroid').agg('mean').loc[:, colnames].reset\_index(drop = True)

centroids

Out[11]:

|  | **x** | **y** |
| --- | --- | --- |
| **0** | 9.889444 | 23.242611 |
| **1** | 28.435750 | 11.546250 |
| **2** | 30.759333 | 40.311167 |

In [12]:

fig, ax = plt.subplots(figsize=(8, 6))

plt.scatter(df.iloc[:,0], df.iloc[:,1], marker = 'o',

c=df['centroid'].astype('category'),

cmap = customcmap, s=80, alpha=0.5)

plt.scatter(centroids.iloc[:,0], centroids.iloc[:,1],

marker = 's', s=200,

c=[0, 1, 2], cmap = customcmap)

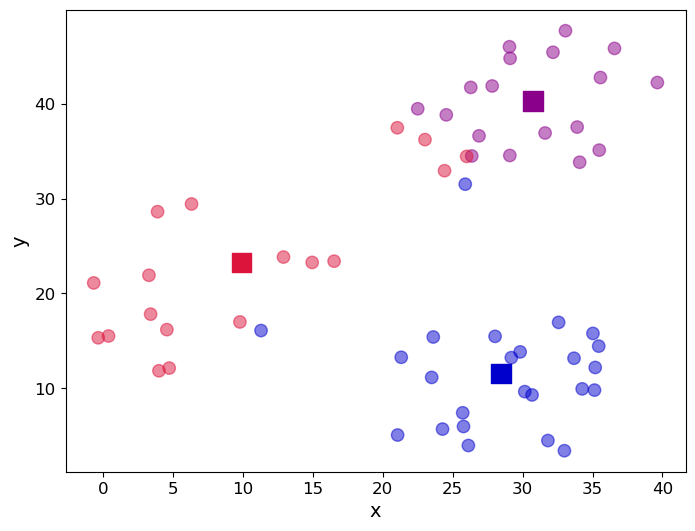
ax.set\_xlabel(r'x', fontsize=14)

ax.set\_ylabel(r'y', fontsize=14)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.show()



Крок 6 - Повторіть кроки 3-5.

In [13]:

def kmeans(dset, k=2, tol=1e-4):

'''

K-means реалізовано для набору

`dset`: датафрей зі спостереженнями

`k`: кількість кластерів, за замовчуванням k=2

`tol`: допуск=1E-4

'''

# Давайте працювати в копії, щоб не зіпсувати оригінал

working\_dset = dset.copy()

# Визначимо деякі змінні для зберігання помилки,

# сигналу зупинки та лічильник для ітерацій

err = []

goahead = True

j = 0

# Крок 2: Ініціюйте кластери, визначивши центроїди

centroids = initiate\_centroids(k, dset)

while(goahead):

# Крок 3 і 4 - Призначення центроїдів і розрахунок похибки

working\_dset['centroid'], j\_err = centroid\_assignation(working\_dset, centroids)

err.append(sum(j\_err))

# Крок 5 - Оновлення положення центроїда

centroids = working\_dset.groupby('centroid').agg('mean').reset\_index(drop = True)

# Крок 6 - Перезапустіть ітерацію

if j>0:

# Чи є похибка меншою за допуск (1E-4)

if err[j-1]-err[j]<=tol:

goahead = False

j+=1

working\_dset['centroid'], j\_err = centroid\_assignation(working\_dset, centroids)

centroids = working\_dset.groupby('centroid').agg('mean').reset\_index(drop = True)

return working\_dset['centroid'], j\_err, centroids

In [14]:

np.random.seed(42)

df['centroid'], df['error'], centroids = kmeans(df[['x','y']], 3)

df.head()

Out[14]:

|  | **x** | **y** | **centroid** | **error** |
| --- | --- | --- | --- | --- |
| **0** | 24.412 | 32.932 | 2 | 3767.568743 |
| **1** | 35.190 | 12.189 | 1 | 1399.889001 |
| **2** | 26.288 | 41.718 | 2 | 262.961097 |
| **3** | 0.376 | 15.506 | 0 | 2683.086425 |
| **4** | 26.116 | 3.963 | 1 | 2723.650198 |

In [15]:

centroids

Out[15]:

|  | **x** | **y** |
| --- | --- | --- |
| **0** | 6.322867 | 19.559800 |
| **1** | 29.330864 | 10.432409 |
| **2** | 29.304957 | 39.050783 |

In [16]:

fig, ax = plt.subplots(figsize=(8, 6))

plt.scatter(df.iloc[:,0], df.iloc[:,1], marker = 'o',

c=df['centroid'].astype('category'),

cmap = customcmap, s=80, alpha=0.5)

plt.scatter(centroids.iloc[:,0], centroids.iloc[:,1],

marker = 's', s=200, c=[0, 1, 2],

cmap = customcmap)

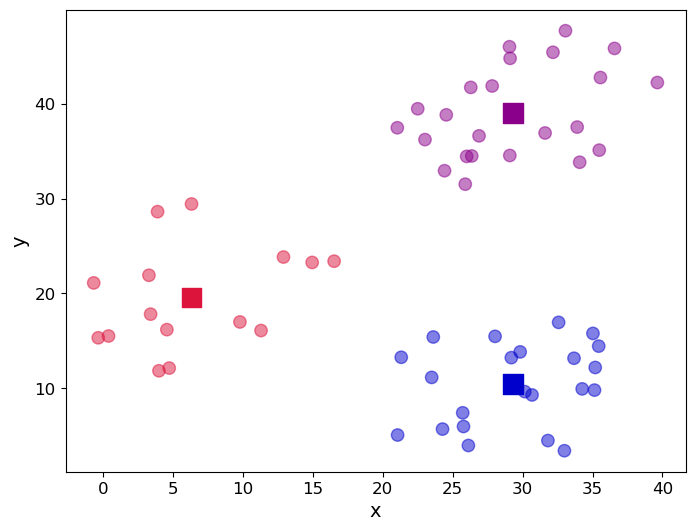
ax.set\_xlabel(r'x', fontsize=14)

ax.set\_ylabel(r'y', fontsize=14)

plt.xticks(fontsize=12)

plt.yticks(fontsize=12)

plt.show()



Отримали 3 групи.У цьому конкретному прикладі дані такі, що різниця між групами є чіткою. Однак, нам може не так пощастити в кожному випадку. Тому питання про те, скільки груп існує, залишається відкритим. Ми можемо використати графік, який допоможе нам мінімізувати похибку, подивившись на запуск алгоритму з послідовністю і шукати "лікоть" на графіку, який вказує на достатню кількість кластерів для використання:

5. Встановити оптимальну кiлькiсть кластерiв з використанням методiв “метод лiктя” / “метод силуету”. Порiвняти отриманi результатами з розмiченими даними (за наявностi).

In [17]:

n = 10

err\_total = []

df\_elbow = blobs[['x','y']]

for i in range(n):

working\_dset, my\_errs, centroids = kmeans(df\_elbow, i+1)

err\_total.append(sum(my\_errs))

fig, ax = plt.subplots(figsize=(7, 4))

plt.plot(range(1,n+1), err\_total, linewidth=3, marker='o')

ax.set\_xlabel(r'Кількість кластерів', fontsize=14)

ax.set\_ylabel(r'Загальна кількість помилок', fontsize=14)

plt.xticks(range(1, n+1), fontsize=12)

plt.yticks(fontsize=12)

plt.show()



Тепер ми можемо застосувати "правило ліктя", яке є евристичним прийомом, що допоможе нам визначити кількість кластерів. Якщо уявити лінію, зображену вище, як руку, то "лікоть" - це точка перегину. У цьому випадку "лікоть" знаходиться між 2 і 4 кластерами, що вказує нам на те, що вибір 3 є правильним.

Порiвняти отриманi результатами з розмiченими даними (за наявностi).

In [18]:

blobs = pd.read\_csv('kmeans\_blobs.csv')

blobs.head()

Out[18]:

|  | **ID** | **x** | **y** | **cluster** |
| --- | --- | --- | --- | --- |
| **0** | 0 | 24.412 | 32.932 | 2 |
| **1** | 1 | 35.190 | 12.189 | 1 |
| **2** | 2 | 26.288 | 41.718 | 2 |
| **3** | 3 | 0.376 | 15.506 | 0 |
| **4** | 4 | 26.116 | 3.963 | 1 |

In [19]:

sns.scatterplot(x = blobs['x'], y = blobs['y'], hue = blobs['cluster'])

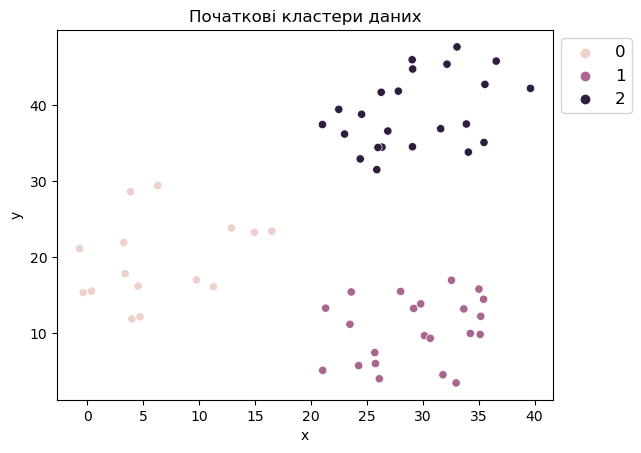
plt.xlabel("x")

plt.ylabel("y")

plt.title("Початкові кластери даних" )

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

plt.show()



6. Порiвняти результати з k-means з sklearn

In [20]:

from sklearn.cluster import KMeans

In [21]:

data = pd.read\_csv('kmeans\_blobs.csv')

data.head()

Out[21]:

|  | **ID** | **x** | **y** | **cluster** |
| --- | --- | --- | --- | --- |
| **0** | 0 | 24.412 | 32.932 | 2 |
| **1** | 1 | 35.190 | 12.189 | 1 |
| **2** | 2 | 26.288 | 41.718 | 2 |
| **3** | 3 | 0.376 | 15.506 | 0 |
| **4** | 4 | 26.116 | 3.963 | 1 |

In [22]:

data\_xy = data[['x', 'y']]

data\_xy.head()

Out[22]:

|  | **x** | **y** |
| --- | --- | --- |
| **0** | 24.412 | 32.932 |
| **1** | 35.190 | 12.189 |
| **2** | 26.288 | 41.718 |
| **3** | 0.376 | 15.506 |
| **4** | 26.116 | 3.963 |

In [23]:

inertias = []

n=10

for i in range(1,n+1):

kmeans = KMeans(n\_clusters=i)

kmeans.fit(data\_xy)

inertias.append(kmeans.inertia\_)

plt.plot(range(1,11), inertias, marker='o')

plt.title('Метод ліктя')

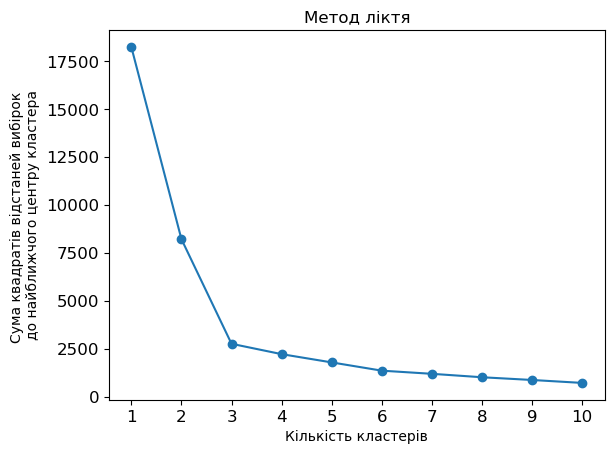
plt.xlabel('Кількість кластерів')

plt.ylabel('Сума квадратів відстаней вибірок \nдо найближчого центру кластера')

plt.xticks(range(1, n+1), fontsize=12)

plt.yticks(fontsize=12)

plt.show()



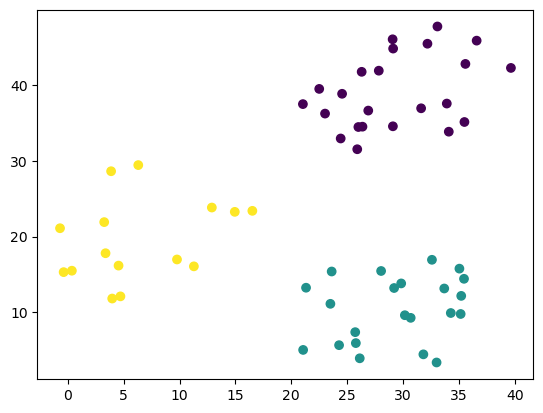
In [24]:

kmeans = KMeans(n\_clusters=3)

kmeans.fit(data\_xy)

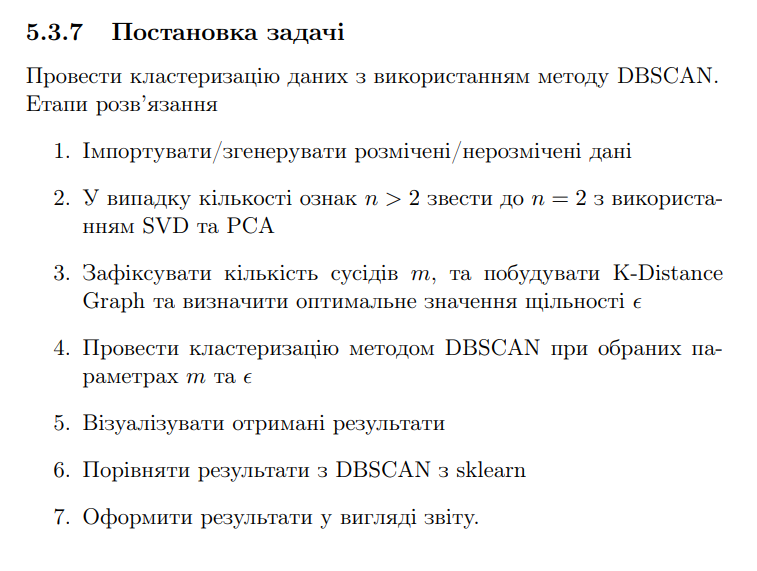
plt.scatter(data\_xy['x'], data\_xy['y'], c=kmeans.labels\_)

plt.show()



## Метод DBSCAN[¶](#%D0%9C%D0%B5%D1%82%D0%BE%D0%B4-DBSCAN)

Лабораторна робота 7¶



In [1]:

import math

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

In [2]:

from sklearn.preprocessing import normalize

from sklearn.cluster import DBSCAN

from scipy.spatial import distance

from sklearn.neighbors import NearestNeighbors

1

In [3]:

# генеруємо дані

def make\_circle(r, n, noise = 30, seed = 1234):

np.random.seed(seed)

return [(math.cos(2\*math.pi/n\*x)\*r+np.random.normal(-noise,noise), math.sin(2\*math.pi/n\*x)\*r+np.random.normal(-noise,noise)) for x in range(1,n+1)]

small\_circle = make\_circle(100, 300, 10)

medium\_circle = make\_circle(300, 700, 20)

big\_circle = make\_circle(500, 1000, 30)

noise = [(np.random.randint(-600,600),np.random.randint(-600,600)) for i in range(300)]

In [4]:

def arrray\_to\_df(arr, i):

df = pd.DataFrame(arr)

df['cluster'] = str(i)

return df

data = [arrray\_to\_df(arr, i) for i, arr in enumerate([small\_circle, medium\_circle, big\_circle, noise])]

data = pd.concat(data)

data.columns = ['x', 'y', 'cluster']

In [5]:

plt.scatter(data['x'], data['y'], label='data')

plt.xlabel('x')

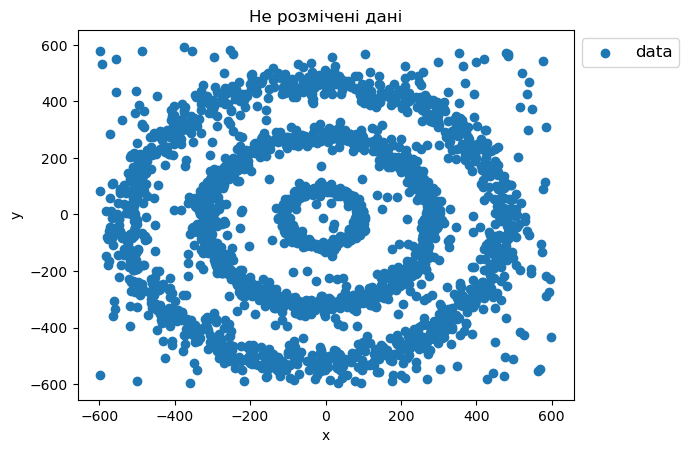
plt.ylabel('y')

plt.title("Не розмічені дані")

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

Out[5]:

<matplotlib.legend.Legend at 0x135e08a15b0>



In [6]:

plt.rcParams['figure.figsize'] = [8,6]

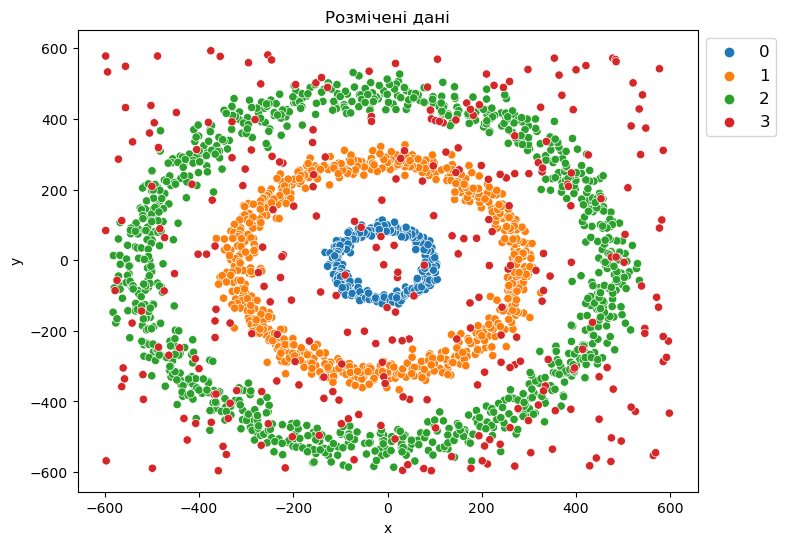
sns.scatterplot(data = data, x = 'x',y = 'y',hue = 'cluster')

plt.title("Розмічені дані")

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

Out[6]:

<matplotlib.legend.Legend at 0x135e141f1f0>



3

In [7]:

k = 2

data\_nn = data.copy()[['x', 'y']]

nearest\_neighbors = NearestNeighbors(n\_neighbors=k)

neighbors = nearest\_neighbors.fit(data\_nn)

distances, indices = neighbors.kneighbors(data\_nn)

distances = np.sort(distances, axis=0)

distances = distances[:,1]

i = np.arange(len(distances))

sns.lineplot(x = i, y = distances)

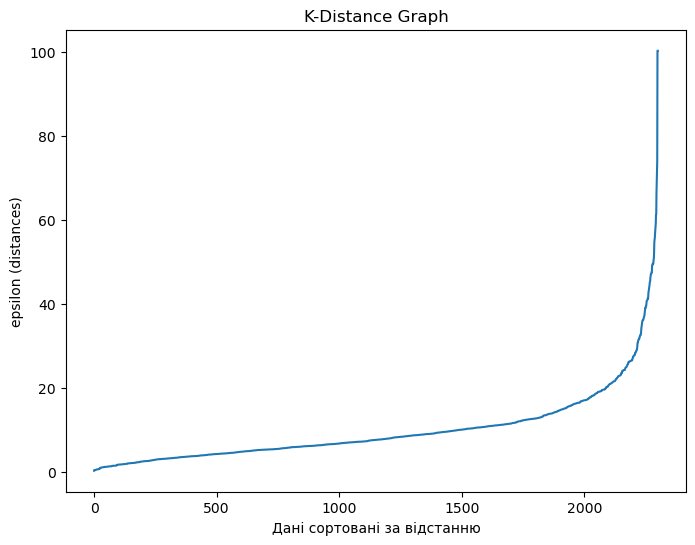
plt.xlabel("Дані сортовані за відстанню")

plt.ylabel("epsilon (distances)")

plt.title('K-Distance Graph')

Out[7]:

Text(0.5, 1.0, 'K-Distance Graph')



4

In [8]:

class dbscan:

def \_\_init\_\_(self, epsilon = None, min\_samples = None, distance = 'euclidean', normalize = False):

self.epsilon = epsilon

self.min\_samples = min\_samples

self.distance = distance

self.normalize = normalize

def find\_distance(self, x, type = 'euclidean'):

"""

Finds distance between numpy arrays.

"""

return distance.squareform(distance.pdist(x, type))

def normalization(self, x):

return (x-np.min(x))/(np.max(x) - np.min(x))

def find\_neighbors(self, x):

return np.where(x <= self.epsilon)[0]

def expand\_cluster(self, neighbors, x, cluster, labels):

# Iterate over each neighbor

for neighbor in neighbors:

# Check that is not assigned

if labels[neighbor] == 0:

# Find neighbors

neighbor\_neighbors = self.find\_neighbors(x[neighbor])

# Check if is core

if len(neighbor\_neighbors) >= self.min\_samples:

labels[neighbor] = cluster

# For each neighbor in neighbors, expand cluster

labels = self.expand\_cluster(neighbor\_neighbors, x, cluster, labels)

return labels

def fit(self, x):

"""

Given a reference point and comparison points and a distance function, returns the index of the neighbors.

"""

# Do normalization

if self.normalize:

x = self.normalization(x)

# Find distance

dist\_matrix = self.find\_distance(x, self.distance)

# Initialize cluster

cluster = 1

n\_obs = x.shape[0]

labels = np.zeros(n\_obs)

for i in range(n\_obs):

# If value not assigned

if labels[i] == 0:

# Find neighbors

neighbors = self.find\_neighbors(dist\_matrix[i])

# Check if neighbors > min\_samples (self included as neighbor)

if len(neighbors) > self.min\_samples:

# If observation is not assigned --> Assign to cluster

if labels[i] == 0:

labels[i] = cluster

# Expand cluster on neighbors

labels = self.expand\_cluster(neighbors, dist\_matrix, cluster, labels)

# Go to next cluster

cluster = cluster + 1

return labels

In [9]:

data['dbscan\_custom'] = dbscan(epsilon=32, min\_samples=5).fit(data[['x', 'y']].to\_numpy())

5

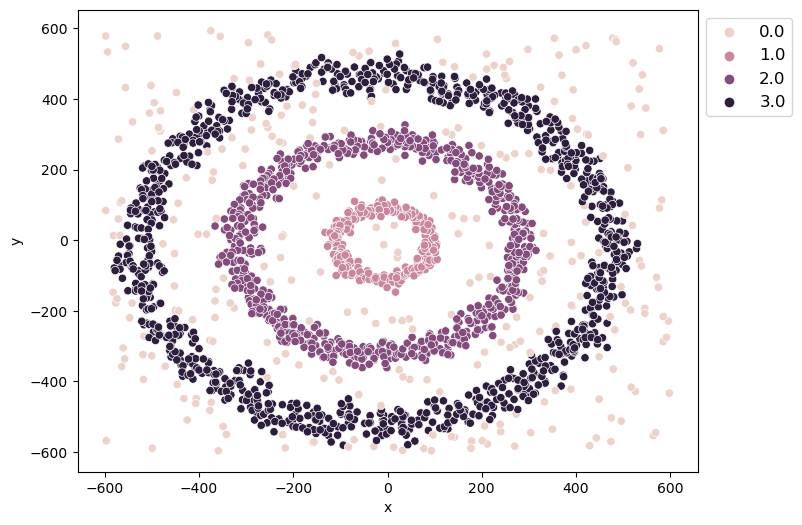
In [10]:

sns.scatterplot(data = data, x = 'x', y = 'y', hue = 'dbscan\_custom')

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

Out[10]:

<matplotlib.legend.Legend at 0x135e10e8bb0>



6

In [11]:

data2 = data.copy()

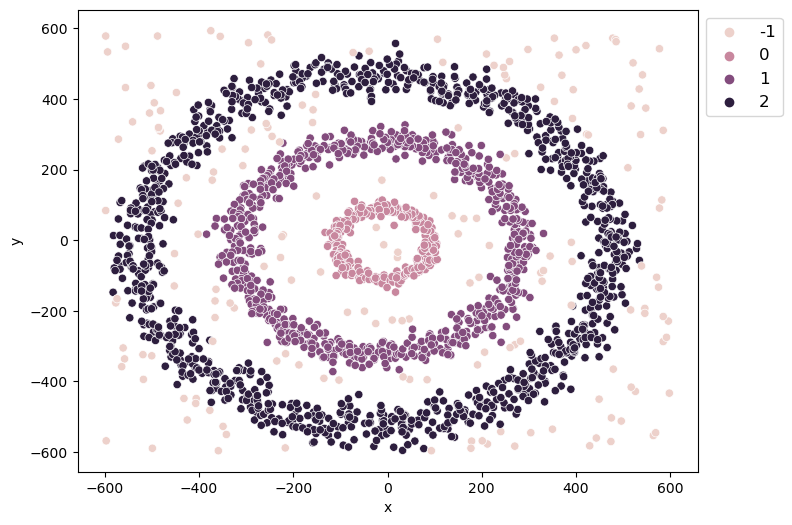
data2['dbscan2'] = DBSCAN(eps=32, min\_samples=5).fit\_predict(data[['x', 'y']])

sns.scatterplot(data = data2, x = 'x', y = 'y', hue = 'dbscan2')

plt.legend(bbox\_to\_anchor=(1,1), loc="upper left", prop={'size': 12})

Out[11]:

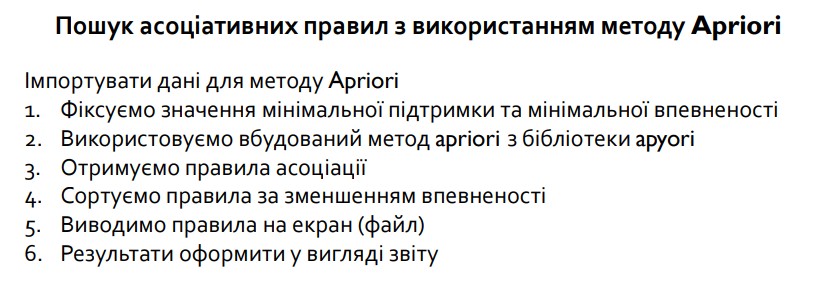
<matplotlib.legend.Legend at 0x135e1119880>

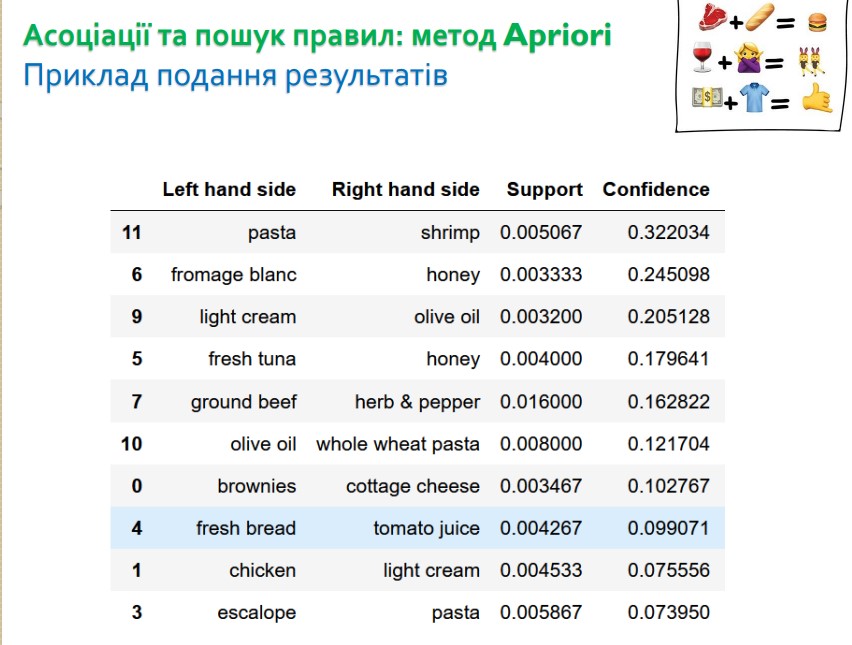


Отже, результати ідентичні.

## Асоціації та пошук правил: метод Apriori[¶](" \l "%D0%90%D1%81%D0%BE%D1%86%D1%96%D0%B0%D1)

Лабораторна робота 8¶





In [1]:

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from apyori import apriori

In [2]:

!pip install apyori

Requirement already satisfied: apyori in c:\users\admin\anaconda3\lib\site-packages (1.1.2)

In [3]:

dataset = pd.read\_csv("Market\_Basket\_Optimisation.csv")

dataset

Out[3]:

|  | **shrimp** | **almonds** | **avocado** | **vegetables mix** | **green grapes** | **whole weat flour** | **yams** | **cottage cheese** | **energy drink** | **tomato juice** | **low fat yogurt** | **green tea** | **honey** | **salad** | **mineral water** | **salmon** | **antioxydant juice** | **frozen smoothie** | **spinach** | **olive oil** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | burgers | meatballs | eggs | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **1** | chutney | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **2** | turkey | avocado | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **3** | mineral water | milk | energy bar | whole wheat rice | green tea | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **4** | low fat yogurt | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **7495** | butter | light mayo | fresh bread | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **7496** | burgers | frozen vegetables | eggs | french fries | magazines | green tea | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **7497** | chicken | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **7498** | escalope | green tea | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **7499** | eggs | frozen smoothie | yogurt cake | low fat yogurt | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

7500 rows × 20 columns

In [4]:

len(dataset)

Out[4]:

7500

In [5]:

data\_new = dataset.drop(range(3500,len(dataset)))

data\_new

Out[5]:

|  | **shrimp** | **almonds** | **avocado** | **vegetables mix** | **green grapes** | **whole weat flour** | **yams** | **cottage cheese** | **energy drink** | **tomato juice** | **low fat yogurt** | **green tea** | **honey** | **salad** | **mineral water** | **salmon** | **antioxydant juice** | **frozen smoothie** | **spinach** | **olive oil** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | burgers | meatballs | eggs | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **1** | chutney | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **2** | turkey | avocado | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **3** | mineral water | milk | energy bar | whole wheat rice | green tea | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **4** | low fat yogurt | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **3495** | burgers | salmon | vegetables mix | carrots | green tea | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **3496** | tomatoes | rice | oil | green tea | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **3497** | herb & pepper | ground beef | salmon | spinach | honey | protein bar | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **3498** | chocolate | red wine | tomato sauce | butter | chicken | salmon | french fries | salt | mint | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **3499** | cookies | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

3500 rows × 20 columns

In [6]:

transactions = []

for i in range(0, len(data\_new)):

transactions.append([str(data\_new.values[i,j]) for j in range(0,20)])

In [7]:

rules = apriori(transactions = transactions, min\_support = 0.003, min\_cinfidence = 0.2, \

min\_lift = 3, min\_length = 2, max\_length = 2)

In [8]:

results = list(rules)

#results

In [9]:

with open("file\_results.txt", 'w') as f:

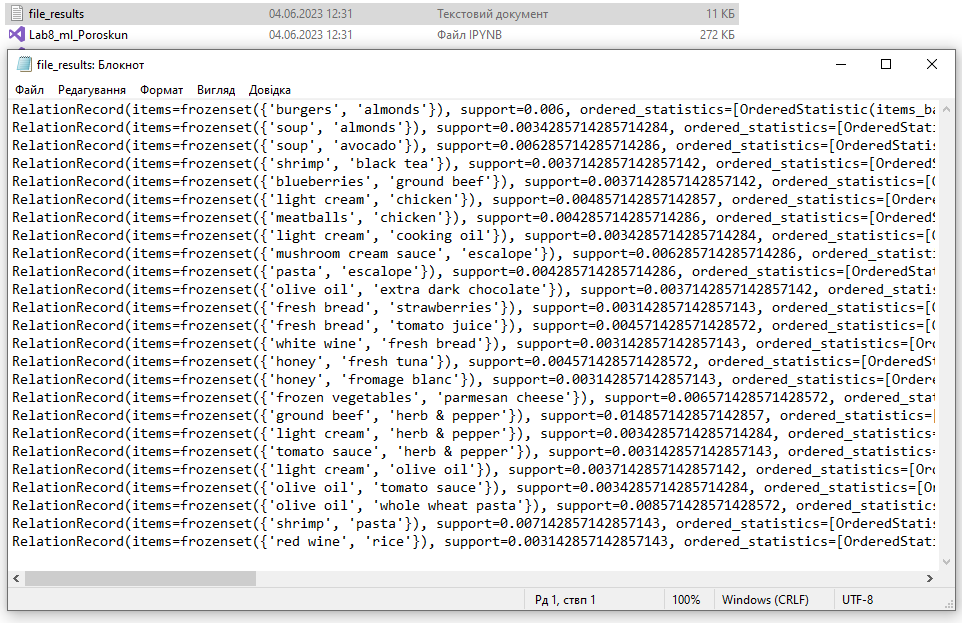
for s in results:

f.write(str(s) + '\n')

with open("file\_results.txt", 'r') as f:

results\_from\_file = [line.rstrip('\n') for line in f]

#print(results\_from\_file)



In [10]:

def inspect(results):

lhs =[tuple(result[2][0][0])[0] for result in results]

rhs =[tuple(result[2][0][1])[0] for result in results]

supports =[result[1] for result in results]

confidences =[result[2][0][2] for result in results]

lifts =[result[2][0][3] for result in results]

return list (zip(lhs, rhs, supports, confidences, lifts))

In [11]:

resultsinDataFrame = pd.DataFrame(inspect(results), \

columns = ["Left hand side", "Right hand side", "Support", "Confidence", "Lift"])

#resultsinDataFrame

In [12]:

resultsinDataFrame.drop('Lift', axis = 1).nlargest(n = 10, columns = "Confidence")

Out[12]:

|  | **Left hand side** | **Right hand side** | **Support** | **Confidence** |
| --- | --- | --- | --- | --- |
| **23** | pasta | shrimp | 0.007143 | 0.423729 |
| **4** | blueberries | ground beef | 0.003714 | 0.371429 |
| **0** | almonds | burgers | 0.006000 | 0.300000 |
| **10** | extra dark chocolate | olive oil | 0.003714 | 0.295455 |
| **3** | black tea | shrimp | 0.003714 | 0.282609 |
| **15** | fromage blanc | honey | 0.003143 | 0.275000 |
| **20** | light cream | olive oil | 0.003714 | 0.209677 |
| **14** | fresh tuna | honey | 0.004571 | 0.192771 |
| **2** | avocado | soup | 0.006286 | 0.174603 |
| **1** | almonds | soup | 0.003429 | 0.171429 |

# **Висновки**

У ході виконання роботи було проведено аналіз 8 тем машинного навчання, завданнями яких була класифікація, кластеризація, зменшення розмірності даних та пошук асоціативних правил. Частина алгоритмів була розроблена і результати були подібні до вже вбудованих методів. Тому вони розроблені правильно. Також були розглянуті результати виконання вбудованих функцій.