



Головна астрономічна обсерваторія НАН України
Київ, Україна

Python: бібліотека scikit-learn для морфологічної класифікація
SDSS-галактик

PYTHON

Дар'я Добричева
daria@mao.kiev.ua

Максим Василенко
vasmax@mao.kiev.ua

Рекомендовано Вченою радою ГАО НАН України
від 24.12.2020, протокол № 16

Зміст

1. Вступ	1
1.1. Що таке машинне навчання	1
1.2. Типи машинного навчання	2
1.3. Вступ до задач машинного навчання	3
2. Експрес введення до Python	4
2.1. Елементарні операції	4
2.2. Аналіз даних з NumPy і Pandas	10
2.2.1 NumPy	10
2.2.2 Pandas	15
2.3. Matplotlib	28
2.4. Seaborn	33
3. Linear Regression	36
4. Logistic Regression	40
5. k-nearest neighbor algorithm	43
6. Decision Tree Classification	45
6.0.1 Decision Tree	45
6.0.2 Random Forests	46
7. Support Vector Machines	50
References	54

1. Вступ

Python є важливою мовою програмування в сучасній астрономії та широко використовується для різноманітних завдань, у тому числі для морфологічної класифікації галактик. У цьому посібнику ми дослідимо основні причини, чому Python важливий для цієї конкретної програми.

1. Простий у вивченні та використанні. Однією з найбільших переваг Python є те, що його відносно легко вивчити та використовувати. Синтаксис Python простий та інтуїтивно зрозумілий, що робить Python чудовою мовою для початківців. Це особливо важливо для астрономів, які можуть не мати достатнього досвіду програмування.
2. Велика та активна спільнота. Python має велику й активну спільноту користувачів і розробників. Це означає, що доступно багато ресурсів, які допоможуть вам вивчити мову, вирішити проблеми та розробити нові інструменти та програми. Це також означає, що існує багато бібліотек і пакетів, доступних спеціально для астрономічних програм.
3. Потужні засоби обробки та візуалізації даних. Python має чудові інструменти обробки та візуалізації даних, що робить його ідеальною мовою для аналізу та відображення астрономічних даних. За допомогою таких бібліотек, як NumPy, Pandas і Matplotlib, ви можете легко маніпулювати великими наборами даних і візуалізувати їх.
4. Доступ до астрономічних баз даних. Python забезпечує легкий доступ до астрономічних баз даних, таких як Sloan Digital Sky Survey (SDSS), архів космічного телескопа Хаббла (HST) і віртуальна обсерваторія (VO). Ці бази даних містять величезну кількість даних, які можна використовувати для дослідження та аналізу.
5. Наявність спеціалізованих пакетів для астрономічних застосувань. У Python є багато спеціалізованих пакетів для астрономічних програм, таких як Astropy і Scikit-learn, які можна використовувати для бінарної морфологічної класифікації галактик. Ці пакети надають спеціальні інструменти для статистичного аналізу.

Детектування мільйонів галактик сучасними цифровими оглядами неба, ставить виклик вченим щодо пошуку альтернативних методів обробки інформації у вигляді методів машинного навчання. Даний методичний посібник присвячений саме цій проблемі – застосуванню методів машинного навчання для визначення морфології галактик.

Морфологія галактик є одним з ключових елементів дослідження еволюції Всесвіту для позагалактичної астрономії та спостережної космології. Найбільш точним методом класифікації галактик, який застосовували дотепер астрономи, є візуальна класифікація. Однак головним його недоліком є залученням великої кількості робочої сили як висококваліфікованих фахівців, так і астрономів-аматорів, наприклад, у проєкті Galaxy Zoo¹. Сучасні дослідження галактик, або ті що заплановані на найближче майбутнє, (SDSS [1], [2], LSST, DES [3], KiDS тощо) здатні виявити сотні мільйонів галактик, що неможливо класифікувати вручну. Це збільшує інтерес до використання альтернативи, у вигляді методів машинного навчання (включаючи глибоке навчання) для автоматизованої морфологічної класифікації галактик.

Підсумовуючи, Python є важливою мовою для астрономів, і його популярність зросла в останні роки завдяки простоті використання, потужним інструментам обробки та візуалізації даних, а також спеціалізованим пакетам для астрономічних програм. Для бінарної морфологічної класифікації галактик Python надає необхідні інструменти для аналізу зображень, редукції даних і статистичного аналізу, необхідні для точної та ефективної класифікації.

1.1. Що таке машинне навчання

Як астрономи ми постійно маємо справу з величезними масивами даних, від зображень галактик і зірок до спектрів різноманітних астрономічних об'єктів. Здатність обробляти й аналізувати ці дані має вирішальне значення для нових відкриттів і вдосконалення нашого розуміння Всесвіту. Тут на допомогу приходять машинне навчання.

¹<http://data.galaxyzoo.org>

Машинне навчання – це галузь дослідження, яка зосереджена на розробці алгоритмів, які можуть навчатися на основі даних і робити прогнози або приймати рішення на основі цих знань. Це підмножина штучного інтелекту (ШІ) і передбачає використання статистичних і математичних моделей для аналізу та отримання інформації з даних.

Метою машинного навчання є розробка алгоритмів, які можуть автоматично визначати шаблони в даних і використовувати ці шаблони для прогнозування або виконання дій. Це робиться за допомогою навчальних даних, які є набором прикладів, які використовуються для навчання алгоритму машинного навчання, як робити прогнози або приймати рішення. Алгоритм вивчає навчальні дані, коригуючи свої внутрішні параметри, поки не зможе точно передбачити результати для нових, невидимих даних.

Існує багато типів алгоритмів машинного навчання, включаючи навчання з учителем, навчання без учителя та навчання з підкріпленням. У навчанні з учителем алгоритм надається з позначеними навчальними даними, де вже відомий правильний результат. Тоді алгоритм вчиться пов'язувати певні функції з певними результатами та може робити прогнози для нових, невидимих даних. Навчання без учителя, з іншого боку, передбачає навчання з немаркованих даних, де алгоритм повинен самостійно ідентифікувати шаблони або зв'язки. Навчання з підкріпленням передбачає навчання шляхом проб і помилок, коли алгоритм отримує зворотний зв'язок на основі дій, які він виконує, і відповідно коригує свою поведінку. В даному методичному посібнику ми приводимо приклад тільки навчання з учителем.

Машинне навчання є потужним інструментом для аналізу та прогнозування на основі складних наборів даних. Це дозволяє нам розробляти алгоритми, які можуть автоматично ідентифікувати шаблони та робити прогнози на основі цих шаблонів. У галузі астрономії машинне навчання має багато застосувань, зокрема аналіз зображень, класифікацію об'єктів і аналіз даних. Використовуючи алгоритми машинного навчання, ми можемо швидко й точно аналізувати великі набори даних, допомагаючи нам робити нові відкриття та поглиблювати наше розуміння Всесвіту.

Ми розглянемо основи машинного навчання, зокрема різні типи алгоритмів і методик, що використовуються в галузі астрономії. Ми також обговоримо важливість попередньої обробки даних і вибору функцій, а також те, як оцінити продуктивність моделі машинного навчання. Ознайомившись з даним посібником ви зможете добре розуміти основи машинного навчання та як його можна використовувати в галузі астрономії.

1.2. Типи машинного навчання

Навчання з учителем Навчання з учителем потребує набір даних (*dataset*), що вміщає приклади з вже відомими (лейблованими) значеннями $(x_i, y_i)_{i=1}^N$. В нашому прикладі вибірку галактик з вже визначеними морфологічними типами. Кожен елемент x_i серед N називається вектор ознак (*feature vector*). Вектор ознак – це вектор у якому кожен вимір $j = 1, \dots, D$ описує елемент вибірки. Це значення називається ознакою і позначається як $x^{(j)}$. Оскільки, кожен x_i у нашій вибірці представляє галактику, то перша ознака $x^{(1)}$ може містити, наприклад, показник кольору, $x^{(2)}$ – абсолютну зоряну величину, $x^{(3)}$ – відстань та ін. Для всіх галактик x вибірки значення j у векторній характеристиці завжди буде містити однаковий вид інформації. Наприклад, $x_i^{(2)}$ містить абсолютну зоряну величину для деякої галактики x_i , в той час як $x_k^{(2)}$ містить абсолютну зоряну величину для кожної галактики x_k , $i, k = 1, \dots, N$. Цільове значення, y_i може бути елементом, що належить до кінцевого набору класів $1, 2, \dots, C$, або реальне число, або більш складна структура, як вектор, матриця, дерево або графік.

Навчання без вчителя У навчанні без вчителя, набір даних – це сукупність елементів вибірки без лейблування $\{x_i\}_{i=1}^N$. Знову ж таки, x є функціональним вектором, а мета навчання без вчителя полягає у створенні моделі, яка сприймає векторний елемент x як вхідний або перетворює його в інший вектор або у значення, яке може бути використане для вирішення практичної проблеми. Наприклад, при кластеризації модель повертає ідентифікатор кластера для кожного вектора функцій у наборі даних. При зменшенні розмірності вихідна модель – це функціональний вектор, який має менше функцій, ніж вхід x ; у вихідному сигналі відхилення – це реальне число, яке вказує, наскільки відрізняється від “типового” прикладу в наборі даних.

Напіваавтоматичне навчання У навчанні набір даних містить як лейбловані, так і нелейбловані елементи вибірки. Зазвичай кількість нелейблованих елементів набагато перевищує кількість лейблованих прикладів. Мета напіваавтоматичного алгоритму, така ж, як і алгоритму навчання з учителем. Маємо надію, що використання багатьох нелейблованих елементів вибірки може допомогти алгоритму навчання знайти (можна сказати, “виробляти” чи “обчислювати”) кращу модель. Це може виглядати протиприродно, що навчання може отримати користь від додавання ще не маркованих прикладів. Схоже, ми додаємо ще більшої невизначеності проблемі. Однак, додаючи незазначені приклади, ви додаєте більше інформації про свою вибірку, більша вибірка краще відображає ймовірність розподілу даних. Теоретично алгоритм навчання повинен мати можливість використовувати цю додаткову інформацію.

Навчання з підкріпленням Навчання з підкріпленням – це розділ машинного навчання, де машина “живе” в середовищі і здатна сприймати стан цього середовища як вектор ознак. Машина може виконувати дії в будь-якому стані. Різні дії приносять різну винагороду, а також можуть перенести машину в інший стан навколишнього середовища. Метою алгоритму навчання посилення є вивчення політики. Політика – це функція (подібна до моделі в навчанні з учителем), яка приймає вектор властивості стану як вхідний і виводить оптимальну дію для виконання в цьому стані. Дія є оптимальною, якщо вона максимізує очікувану середню винагороду. Навчання з підкріпленням вирішує певний тип проблем, коли прийняття рішень є послідовним, а мета – довгостроковою, наприклад, в робототехніці, управлінні ресурсами чи логістиці.

1.3. Вступ до задач машинного навчання

Задачі машинного навчання розбиваються на наступні кроки:

1. Отримання даних
2. Підготовка даних
3. Вибір релевантних ознак і даних
4. Вибір алгоритму навчання
5. Тренування обраного алгоритму
6. Аналіз отриманих результатів

Звичайно, якщо на кроці 6 Ви не отримали якісних результатів, то слід повернутися на: крок 4, щоб спробувати інший алгоритм навчання крок 3, щоб вибрати інші ознаки для навчання алгоритму крок 2, щоб перепідготувати дані В цьому методичному посібнику ми будемо вводити дані вручну чи завантажувати з google drive. Підготовка даних полягатиме у видаленні пропущеної інформації та нормуванні параметрів. Підбір релевантних ознак буде здійснюватися шляхом поступового зменшення розмірності вхідних даних за рахунок видалення параметрів які не впливають на класифікацію. Ми розглянемо декілька алгоритмів машинного навчання з учителем (k-nearest neighbor algorithm, Logistic Regression, Random Forest, Support Vector Machines). Тренування буде здійснюватися після підбору оптимальних параметрів навчання. А аналіз полягає у визначенні точності, що ми зробимо різними способами.

2. Експрес введення до Python

В данній роботі ми будемо наводити приклади, працюючи в **GoogleColab**², або скорочено “Colab” – це безкоштовний сервіс від Google Research. Він дозволяє будь-кому працювати в середовищі з мовою Python (2/3) через браузер. Даний сервіс включає в себе послуги ноутбуків Jupyter, але не потребує налаштування, одночасно надаючи безкоштовний доступ до обчислювальних ресурсів, включаючи графічні процесори. Google Colab надає все необхідне для машинного навчання прямо в браузері, дає безкоштовний доступ до потужних GPU і TPU. Заздалегідь хочеться зауважити, що у нього є деякі обмеження, тому ви не зможете використовувати його для production, але за допомогою Google Colab ви можете легко проводити маніпуляції з даними і застосовувати різні методи машинного навчання.

2.1. Елементарні операції

Python використовує арифметичні операції для маніпулювання числами аналогічно іншим мовам програмування:

```
Додавання: +
Множення: *
Далення: /
Піднесення до степеня: **
Остача від ділення: %
Ціле від ділення: //
```

Для кращого сприйняття коду ми будемо використовувати коментарі. Коментарі – це анотації що знаходяться в самому коді і допомагають розібратися в написаному, але ігноруються при інтерпретації коду і не виконуються. Існує декілька варіантів закоментувати текст, ми можемо закоментувати стрічку або закоментувати частину коду. Для того щоб закоментувати стрічку використовується оператор ‘#’, таким чином код чи текст що стоїть після цього оператора буде ігноруватися. Якщо ж треба закоментувати частину коду чи якийсь текст, то можна використати встроєні подвійні або одинарні лапки. Код чи текст що стоїть такими операторами не буде виконуватися. Такі оператори будуть ігноруватися якщо вони стоять між одинарних лапок і будуть сприйматися як звичайне текстове значення. Але варто пам’ятати, що при використанні коментарів слід дотримуватися табуляції до якої чутлива мова програмування Python.

Приклад присвоєння чисел змінним та обчислення їх суми:

```
[In]: x,y = 2,3    # Присвоювати значення змінним можна перераховуючи їх через кому
      z = x + y    # Виконуємо додавання результат якого записуємо в нову змінну z
      z           # Виведення значення z
[Out]: 5
```

В середовищі ‘Colab’ можна виводити на екран змінні без використання функції print(), як це вже було показано в попередньому прикладі. Але допускаються два варіанти виводу:

```
[In]: x = 'hello' # Присвоєння змінній x значення текстового значення 'hello'
      x           # Вивід результату (допустимо в 'Colab')
[Out]: 'hello'
[In]: print(x)    # Вивід результату з використанням функції print()
[Out]: hello
```

Досить часто виникають ситуації, коли потрібно вивести рядок, підставивши в нього деякі значення, отримані в процесі виконання програми. Підстановку значень можна зробити за допомогою форматування рядків використовуючи метод format:

²<https://colab.research.google.com>

```
[In]: name = 'galaxies'
      num = 12
      print('The number of {} in the sample is {}'.format(name,num))
[Out]: The number of galaxies in the sample is 12
[In]: print('The number of {one} in the sample is {two}'.format(one=name,two=num))
[Out]: The number of galaxies in the sample is 12
```

Python працює з трьома основними типами послідовностей: *списки* (lists), *кортежі* (tuples) та *діапазон* (range).

Списки є невід’ємна частина мови Python. Список дозволяє додавати або видаляти елементи, і список автоматично налаштовує сховище резервних копій, яке містить ці елементи, виділяючи або звільняючи пам’ять. Списки Python можуть містити довільні елементи - “все” є об’єктом у Python, включаючи функції. Тому ви можете поєднувати та поєднувати різні типи даних і зберігати їх усі в одному списку. Це може бути потужною функцією, але недоліком є те, що підтримка декількох типів даних одночасно означає, що дані, як правило, менш щільно упаковані. І як результат, вся конструкція займає більше місця³.

```
[In]: [1,2,3]                # Створюємо список з використанням квадратних дужок
[Out]: [1, 2, 3]
[In]: ['hi',1,[1,2]]         # Список може містити елементи різних типів даних
[Out]: ['hi', 1, [1, 2]]
[In]: my_list = ['a','b','c'] # Створюємо список з назвою my_list
[In]: p=[]                   # Також можна створювати пусті списки
      p
[Out]: []
```

Додавання елементів до списку є досить поширеним завданням у Python, тому мова надає багато методів та операторів, які можуть допомогти вам у цій операції. Одним з таких методів є **.append()**. За допомогою **.append()** ви можете додавати елементи в кінець існуючого оригінального списку, не створюючи новий список. Ви також можете використовувати **.append()** у циклі **for** для програмного заповнення списків.

```
[In]: my_list.append('d')    # Додаємо елемент 'd' в кінець списку
      my_list
[Out]: ['a', 'b', 'c', 'd']
[In]: my_list[1]             # Викликаємо другий елемент my_list
[Out]: 'b'
[In]: my_list[1:]            # Викликаємо всі елементи починаючи з другого (включно)
[Out]: ['b', 'c', 'd']
[In]: my_list[:1]           # Викликаємо всі елементи до другого (виключно)
[Out]: ['a']
[In]: my_list[0] = 'NEW'     # Змінюємо перший елемент в списку
      my_list
[Out]: ['NEW', 'b', 'c', 'd']
[In]: nest = [1,2,3,[4,5,['target']]]
      nest[3]                # Викликаємо третій елемент в nest, це вкладений список
[Out]: [4, 5, ['target']]
[In]: nest[3][2][0]          # Звертаємось до елементів у вкладених списках
[Out]: 'target'
```

Метод Python **reverse()** змінює порядок списку, а метод **sort()** сортує список:

```
[In]: a = [4, 1, 2, 3]
      a.reverse()            # Змінюємо порядок елементів списку на зворотній
      a
[Out]: [3, 2, 1, 4]
```

³<https://docs.python.org/3.6/tutorial/datastructures.html#tuples-and-sequences>

```
[In]: a.sort()          # Сортуємо елементи списку на зворотні
      a
[Out]: [1, 2, 3, 4]
```

Коли є потреба користуватись послідовністю яку не можна змінити, на допомогу прийдуть **кортежі**⁴. Слово **кортеж** взяли з математики, де воно означає впорядковану та скінченну сукупність елементів. Кількість елементів в кортежі визначає його довжину. Якщо ми маємо кортеж, наприклад, з п'яти елементів, то в ньому завжди будуть саме ці п'ять елементи. Елементи кортежу розділяються комою та заключені в круглі дужки:

```
[In]: k = (1, 'p', 2, 'hi', 7) # Створюємо кортеж елементи з різних типів даних
      k
[Out]: (1, 'p', 2, 'hi', 7)
[In]: k[0]                     # Звертаємось до елементів першого елементу масиву
[Out]: 1
[In]: v = (10, 11, k)          # Елементом кортежу можуть бути інші кортежі
      v
[Out]: (10, 11, (1, 'p', 2, 'hi', 7))
```

Діапазон представляє з себе незмінну послідовність чисел і зазвичай використовується для циклу for.

```
[In]: list(range(5))           # Створюємо діапазон від 0 до 5
[Out]: [0, 1, 2, 3, 4]
[In]: list(range(1, 11))       # Створюємо діапазон від 1 до 11
[Out]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[In]: list(range(0, 30, 4))     # Створюємо діапазон від 0 до 30 з кроком 4 починаючи з 0
[Out]: [0, 4, 8, 12, 16, 20, 24, 28]
[In]: list(range(0, -10, -1))   # Також допускається рух і взоротному напрямку
[Out]: [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
[In]: list(range(0))            # Також можливі і пусті діапазони
[Out]: []
```

Корисним типом даних, вбудованим в Python, є **словники** (dict). Філологічний словник має дві частини: слово, яке визначається; та його визначення. Словники Python, зберігають колекцію об'єктів. Однак замість того, щоб зберігати об'єкти в послідовності, словники зберігають інформацію в парах даних, які називаються: ключ-значення. Ключ унікальний, а значення можуть як повторюватись так і містити різноманітні типи даних. Тобто кожен об'єкт у словнику має дві частини: ключ і значення. Ось невеликий приклад використання словника:

```
[In]: areas = {'Daria': 'astrophysics',
               'Max': 'radio astronomy',
               'Lara': 'astrometry',
               'Alex': 'celestial mechanics',
               'Viki': 'observational astronomy'}
      areas
[Out]: {'Alex': 'celestial mechanics',
       'Daria': 'astrophysics',
       'Lara': 'astrometry',
       'Max': 'radio astronomy',
       'Viki': 'observational astronomy'}
[In]: areas['Daria']
[Out]: 'astrophysics'
```

⁴<https://docs.python.org/3/library/stdtypes.html#tuple>

Як і в інших мовах програмування, в Python присутній логічний тип даних 'bool', цей тип даних може приймати два допустимі значення істини чи хибності, тобто 'True' чи 'False', 'True'-тотожне будь-якому не нульовому значенню, а 'False' відповідно тотожне нулю. Цей тип даних є результатом логічних операторів і часто застосовується для перевірки і в циклах.

Приклади використання логічних операторів порівняння:

```
[In]: 1 > 2                # Строго більше
[Out]: False
[In]: 1 >= 1              # Більше або дорівнює
[Out]: True
[In]: 1 != 4              # Не дорівнює
[Out]: True
[In]: 1 == 1              # Строго дорівнює
[Out]: True
[In]: 'Pryvit' == 'Buvai' # Порівнювати можна будь-які типи даних
[Out]: False
[In]: (1 > 2) and (2 < 3)  # Перевірка декількох умов одночасно
[Out]: False
[In]: (1 == 2) or (2 == 3) or (4 == 4) # Перевірка хоча б однієї з умов
[Out]: True
```

На різних етапах роботи часто доводиться використовувати перевірки щоб обирати подальший шлях виконання програми.

Приклади використання **if**, **elif**, **else**:

```
[In]: if 1 < 2:             # Операція виконується,
    print('Привіт')         # якщо виконується умова
[Out]: Привіт
[In]: if 1 > 2:
    print('Галактика')
    else:                   # Виконується тільки в тому випадку,
    print('Зоря')           # якщо попередні умови не виконались
[Out]: Зоря
[In]: if 1 == 2:
    print('Меркурій')
    elif 2 == 3:            # Об'єднання декількох перевірок в одну
    print('Венера')
    else:                   # Виконується, якщо не спрацювали всі 'elif' та 'if'
    print('Земля')
[Out]: Земля
```

Якщо потрібно виконати одну і ту саму операцію багато разів ви можете використати цикли. **Цикл** – це блок коду, який повторюється знову і знову або задану кількість разів, або до тих пір, поки не буде виконана задана умова. Python, як і більшість мов програмування, в своєму арсеналі має два типи циклів: цикл **for** (for loop) та цикл **while** (while loop).

Цикл **for** в мові програмування Python призначений для перебору елементів структур даних і деяких інших об'єктів. Що значить перебір елементів? Наприклад, у нас є список, що складається з ряду елементів. Спочатку беремо з нього перший елемент, потім другий, потім третій і так далі. З кожним елементом ми виконуємо одні й ті ж дії в тілі for. Нам не потрібно витягати елементи по їх індексами і піклуатись, на якому з них список закінчується. Цикл for сам перебере і визначить кінець. Цикл for має дві основні частини:

- оператор for починається з ключового слова for, після якого стоїть двокрапкою (:);
- тіло циклу містить код, який повинен виконуватися на кожному кроці циклу, і має відступи з чотирьох пробілів.

Давайте розглянемо приклад:

```
[In]: seq = [1,2,3,4,5]
      for item in seq:          # Виконується для кожного елемента
          print(item)
[Out]: 1
       2
       3
       4
       5
[In]: for jelly in seq:         # В середині циклу змінна jelly щоразу приймає
      print(jelly+jelly)        # значення відповідного елемента
[Out]: 2
       4
       6
       8
       10
```

Цикл *while* повторює виконання коду до тих пір поки якась умова відповідає значенню 'True'. Кожен цикл *while* має дві частини:

- оператор *while* починається з ключового слова *while*, за яким слід умова, і закінчується двокрапкою (:);
- тіло циклу містить код, який повторюється на кожному кроці циклу. Кожен рядок має відступи чотирьох пробілів.

Коли цикл *while* виконується, Python оцінює умову тесту та визначає, чи є вона істинна чи хибна. Якщо умова тесту є істинною, продовжується виконання коду у тілі циклу. В іншому випадку код у тілі пропускається, а решта програми виконується.

```
[In]: i = 1
      while i < 5:              # Виконується допоки умова True
          print('i is: {}'.format(i))
          i = i+1
[Out]: i is: 1
       i is: 2
       i is: 3
       i is: 4
```

List comprehensions – це спосіб складання списків, за допомогою якого ми можемо переписати цикл *for* всього в один рядок коду:

```
[In]: x = [1,2,3,4]
      out = []
      for item in x:           # До списку 'out' щоразу додається результат
          out.append(item**2)  # піднесення до степеня елемента зі списку 'x'
          print(out)
[Out]: [1, 4, 9, 16]
[In]: [item**2 for item in x]  # Наведене вище можна записати в короткій формі
[Out]: [1, 4, 9, 16]
```

Слід зауважити, що циклів необхідно уникати в тому випадку коли операцію можна виконати за допомогою влаштованих алгоритмів та індексацію, мова про яку піде далі, оскільки виконання циклу в Python може займати багато часу. Особливо варто уникати вкладених циклів.

Коли ви будете писати програми, ви можете виявити, що вам доведеться використовувати однакові рядки коду неодноразово. Або, можливо, вам потрібно кілька разів обчислити одну і ту ж формулу з різними значеннями. У вас може виникнути спокуса скопіювати та вставити подібний код до інших частин вашої програми та модифікувати його за потреби, але є більш

раціональне рішення. Повторюваний код може бути важким для підтримки. Якщо ви виявите помилку в коді, який був скопійований та вставлений всюди, вам доведеться застосувати виправлення скрізь, де був скопійований код. Це багато роботи, і ви можете пропустити місце. Для уникнення таких повторень потрібно визначати та використовувати власні *функції*, що будуть викликатись, коли вам потрібно повторно використовувати потрібний код в новому місці.

Кожна функція складається з двох частин:

- сигнатура функції визначає назву функції та будь-які вхідні дані, які вона очікує;
- тіло функції містить код, який запускається кожного разу, коли використовується функція.

Всі змінні які визначаються всередині функції, залишаються і можуть бути викликані тільки всередині тіла цієї функції. Приклад використання:

```
[In]: def square(x):      # Визначення функції та вхідні дані 'x'
      return x**2        # Результат виконання, що повертається в головну частину
      out = square(2)
      print(out)
[Out]: 4
```

Lambda-функція в Python – це анонімна функція або функція, що не має імені, а отже вона займає не більше одного рядка. Кожна *lambda*-функція має три основні частини:

- ключове слово `lambda`;
- параметри або пов'язані змінні;
- тіло функції.

```
[In]: adder = lambda x, y: x + y      # Визначаємо lambda-функцію
      adder(1,2)                      # Застосовуємо lambda-функцію
[Out]: 3
```

Цей код можна переписати в звичному вигляді, але він буде займати більше місця:

```
[In]: def adder(x, y):
      return x + y
      adder(1,2)
[Out]: 3
```

Мова програмування Python містить деякі корисні методи, що вбудовані в неї і не потребують підключення сторонніх бібліотек. Приклади використання деяких методів:

```
[In]: st = 'hello my name is Galileo'
      st.lower()                                # Приведення до нижнього регістру
[Out]: 'hello my name is galileo'
[In]: st.upper()                                # Приведення до верхнього регістру
[Out]: 'HELLO MY NAME IS GALILEO'
[In]: st.split()                                # Розділення стрічки на слова
[Out]: ['hello', 'my', 'name', 'is', 'Galileo'] # Результат - список елементів
[In]: tweet = 'Hello I am Galileo! #Astronomer'
      tweet.split('#')                          # Рміна маркера розділення
[Out]: ['Hello I am Galileo! ', 'Astronomer']
[In]: tweet.split('#')[1]                      # Звернення до елемента не визначаючи
[Out]: 'Astronomer'                            # результат розділення
[In]: d={'key1': 'item1', 'key2': 'item2'}
      d.keys()                                  # Виведення списку ключових слів словника
[Out]: dict_keys(['key2', 'key1'])
```

```
[In]: d.items()                                # Окремо виведення списку ключів і значень
[Out]: dict_items([('key2', 'item2'), ('key1', 'item1')])
[In]: lst = [1,2,3]
      lst.pop()                                # Вилучає елемент зі списку
[Out]: 3                                       # та повертає його
[In]: lst
[Out]: [1, 2]
[In]: 'x' in [1,2,3]                           # Можна перевірити наявність
[Out]: False                                 # елемента в списку
[In]: 'x' in ['x','y','z']
[Out]: True
```

2.2. Аналіз даних з NumPy і Pandas

Науковці витрачають велику кількість часу на очищення наборів та обробку даних, зводячи їх до вигляду який можна використовувати. Багато дослідників стверджують, що початкові етапи отримання та очищення даних складають 80 % роботи. Отже, якщо ви будете працювати з великими масивами даних, важливо вміти працювати з неупорядкованими даними, незалежно від того, чи це означає відсутність значень, невідповідне форматування або неправильні записи. Python, за замовчуванням, не має жодної з цих функцій, за винятком деяких основних математичних операцій, які можуть мати справу лише зі змінною, а не з масивом або матрицею. **NumPy** і **Pandas** - це два потужні пакети Python, які дозволяють ефективно використовувати мову в наукових цілях. Ми допоможемо розібратись, як привести данні у відповідну форму за допомогою бібліотеки NumPy і Pandas.

2.2.1 NumPy

NumPy – це один з основних пакетів для наукових обчислень в Python. Він містить функціональні можливості для роботи з багатовимірними масивами, високорівневими математичними функціями. При необхідності можна використовувати формалізм лінійної алгебри, не змінюючи попередньо масиви NumPy. Більше того, масиви можна динамічно змінювати за розміром. Замість того, щоб створювати новий масив, коли ви хочете позбутися певних елементів, ви можете застосувати до нього маску. Створення масивів NumPy важливо, коли ви працюєте з іншими бібліотеками Python, які покладаються на них, такими як SciPy, Pandas, Matplotlib, scikit-learn тощо. NumPy підходить для створення та роботи з масивами, оскільки пропонує корисні процедури, забезпечує підвищення продуктивності та дозволяє писати стислий код.

Для того, щоб використовувати бібліотеку NumPy, нам потрібно імпортувати її в нашу програму. За домовленістю, модуль numpy імпортується під псевдонімом np, наступним чином:

```
[In]: import numpy as np
```

Приклад створення масиву з використанням NumPy:

```
[In]: my_list = [1,2,3]
      my_list
[Out]: [1, 2, 3]
[In]: np.array(my_list)
[Out]: array([1, 2, 3])
[In]: my_matrix = [[1,2,3],[4,5,6],[7,8,9]]
      my_matrix
[Out]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[In]: np.array(my_matrix)
[Out]: array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

У числових обчисленнях дуже часто потрібні масиви з рівномірно розміщеними значеннями між початковим і кінцевим. NumPy надає подібну функцію для створення таких масивів: **np.arange**. Функція має три аргументи, де перші два аргументи є початковим і кінцевим значеннями, а третій – кроком:

```
[In]: np.arange(0,10)                                # За замовчуванням крок одиничний
[Out]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
[In]: np.arange(0,11,2)
[Out]: array([ 0,  2,  4,  6,  8, 10])
```

numpy.zeros() або **np.zeros** Функція Python використовується для створення матриці, повної нулів, **np.ones()** – повної одиниць. В Python **numpy.zeros()** можна використовувати, коли ви ініціалізуєте ваги під час першої ітерації в статистичних завданнях. Розглянемо такий приклад:

```
[In]: np.zeros(3)
[Out]: array([ 0.,  0.,  0.])
[In]: np.zeros((3,3))
[Out]: array([[ 0.,  0.,  0.],
               [ 0.,  0.,  0.],
               [ 0.,  0.,  0.]])
[In]: np.ones(3)
[Out]: array([ 1.,  1.,  1.])
[In]: np.ones((3,3))
[Out]: array([[ 1.,  1.,  1.],
               [ 1.,  1.,  1.],
               [ 1.,  1.,  1.]])
```

Функція **linspace()** повертає одновимірний масив із зазначеної кількості елементів, значення яких рівномірно розподілені усередині заданого інтервалу:

```
[In]: np.linspace(0,10,3)
[Out]: array([ 0.,  5., 10.])
[In]: np.linspace(0,10,50)
[Out]: array([ 0.          ,  0.20408163,  0.40816327,  0.6122449 ,
               0.81632653,  1.02040816,  1.2244898 ,  1.42857143,
               1.63265306,  1.83673469,  2.04081633,  2.24489796,
               2.44897959,  2.65306122,  2.85714286,  3.06122449,
               3.26530612,  3.46938776,  3.67346939,  3.87755102,
               4.08163265,  4.28571429,  4.48979592,  4.69387755,
               4.89795918,  5.10204082,  5.30612245,  5.51020408,
               5.71428571,  5.91836735,  6.12244898,  6.32653061,
               6.53061224,  6.73469388,  6.93877551,  7.14285714,
               7.34693878,  7.55102041,  7.75510204,  7.95918367,
               8.16326531,  8.36734694,  8.57142857,  8.7755102 ,
               8.97959184,  9.18367347,  9.3877551 ,  9.59183673,
               9.79591837, 10.          ])
```

Функція **eye()** використовується для створення двувимірної одиничної матриці:

```
[In]: np.eye(4)
[Out]: array([[ 1.,  0.,  0.,  0.],
               [ 0.,  1.,  0.,  0.],
               [ 0.,  0.,  1.,  0.],
               [ 0.,  0.,  0.,  1.]])
```

np.random.rand() дана функція створює масив вказаної форми та заповнює його випадковими числами з плаваючою точкою, які рівномірно розподіляються в інтервалі [0, 1):

```
[In]: np.random.rand(2)                # Повертає вектор з двох
[Out]: array([ 0.11570539,  0.35279769]) # випадкових елементів
[In]: np.random.rand(5,5)              # Повертає двувимірний масив
                                           # однорідно розподілених елементів
[Out]: array([[ 0.66660768,  0.87589888,  0.12421056,  0.65074126,  0.60260888],
               [ 0.70027668,  0.85572434,  0.8464595 ,  0.2735416 ,  0.10955384],
               [ 0.0670566 ,  0.83267738,  0.9082729 ,  0.58249129,  0.12305748],
               [ 0.27948423,  0.66422017,  0.95639833,  0.34238788,  0.9578872 ],
               [ 0.72155386,  0.3035422 ,  0.85249683,  0.30414307,  0.79718816]])
```

np.random.randn() ця функція створює масив зазначеної форми і заповнює його випадковими числами з плаваючою точкою, які обрані з одновимірного нормального (Гаусовим) розподілу із середнім значенням рівним 0 і дисперсією рівною 1:

```
[In]: np.random.randn(2)               # Повертає вектор з двох випадкових елементів
[Out]: array([-0.27954018,  0.90078368])
[In]: np.random.randn(5,5)             # Повертає двувимірний масив з випадковими
                                           # нормально розподіленими елементами
[Out]: array([[ 0.70154515,  0.22441999,  1.33563186,  0.82872577, -0.28247509],
               [ 0.64489788,  0.61815094, -0.81693168, -0.30102424, -0.29030574],
               [ 0.8695976 ,  0.413755 ,  2.20047208,  0.17955692, -0.82159344],
               [ 0.59264235,  1.29869894, -1.18870241,  0.11590888, -0.09181687],
               [-0.96924265, -1.62888685, -2.05787102, -0.29705576,  0.68915542]])
```

За допомогою функції ***randint*** можна створити масиви з випадкових цілих чисел:

```
[In]: np.random.randint(1,100)         # Повертає випадкове ціле число в інтервалі
[Out]: 44                               # від 1 до 100
[In]: np.random.randint(1,100,10)      # Повертає 10 випадково розподілених цілих чисел
                                           # в інтервалі від 1 до 100
[Out]: array([13, 64, 27, 63, 46, 68, 92, 10, 58, 24])
```

Функція ***reshape()*** змінює форму масиву без зміни його даних за умови однакової кількості елементів в початковому і кінцевому масивах. Наведемо приклад, спочатку створивши масив, а потім змінимо його форму за допомогою ***reshape()***:

```
[In]: arr = np.arange(25)
      arr
[Out]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
              17, 18, 19, 20, 21, 22, 23, 24])
[In]: arr.reshape(5,5)                 # Перетворює масив 'arr' довжиною в 25
                                           # елементів в двувимірний масив 5x5
[Out]: array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14],
               [15, 16, 17, 18, 19],
               [20, 21, 22, 23, 24]])
[In]: ranarr = np.random.randint(0,50,10)
      ranarr
[Out]: array([ 7, 35, 47, 19, 18, 46, 27, 41, 11, 42])
[In]: ranarr.reshape(2,5)              # Перетворює масив 'arr' довжиною в 10
                                           # елементів в двувимірний масив 2x5
[Out]: array([[ 7, 35, 47, 19, 18],
               [46, 27, 41, 11, 42]])
```

За допомогою вбудованих в Python функцій ***min()*** та ***max()***, можна знайти мінімальне або максимальне значення елемента списку. Функції ***argmin()*** та ***argmax()*** повертають індекси мінімального або максимального елемента списку:

```
[In]: ranarr
[Out]: array([10, 12, 41, 17, 49,  2, 46,  3, 19, 39])
[In]: ranarr.max()          # Повертає максимальний елемент з 'ranarr'
[Out]: 49
[In]: ranarr.argmax()       # Повертає індекс максимального елементу з 'ranarr'
[Out]: 4
[In]: ranarr.min()          # Повертає мінімальний елемент з 'ranarr'
[Out]: 2
[In]: ranarr.argmin()        # Повертає значення мінімального елементу з 'ranarr'
[Out]: 5
```

Індексація (Indexing) та **нарізка** (Selection) - це дві найпоширеніші операції, з якими вам потрібно бути знайомими під час роботи з масивами NumPy. Ви будете використовувати їх, коли працюватимете з підмножиною масиву. Наведемо приклади індексування та нарізки одновимірних масивів:

```
[In]: arr = np.arange(0,11)      # Створюємо numpy масив з одиничним кроком
      arr                        # від 0 до 11
[Out]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
[In]: arr[8]                     # Обираємо елементи за аналогією зі списками
[Out]: 8
[In]: arr[-2]                    # Можна також використовувати зворотню індексацію
[Out]: 9
[In]: arr[1:5]                   # Вибираємо елементи з першого по п'ятий (не включно)
[Out]: array([1, 2, 3, 4])
[In]: arr[0:5]=100               # Перевизначаємо елементи масиву з першого по п'ятий
      arr
[Out]: array([100, 100, 100, 100, 100,  5,  6,  7,  8,  9, 10])
[In]: arr = np.arange(0,11)
      slice_of_arr = arr[0:6]     # Також ми можемо вирізати частину масиву
      slice_of_arr
[Out]: array([0, 1, 2, 3, 4, 5])
[In]: slice_of_arr[:]=99          # Переозначити всі елементи, або їх частину.
      slice_of_arr
[Out]: array([99, 99, 99, 99, 99, 99])
[In]: arr
[Out]: array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
[In]: arr_copy = arr.copy()       # Таким чином можна створити копію масиву
      arr_copy
[Out]: array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
```

Якщо ви працюєте з даними, вам частіше доведеться стикатися з двовимірними масивами чим з одновимірними. Для створення багатовимірних масивів ми використовуємо подвійні дужки. Індексацію можна використовувати для виділення якоїсь конкретної частини такого масиву для роботи з нею. Приклади індексування та нарізки двовимірних масивів:

```
[In]: arr_2d = np.array([[5,10,15],[20,25,30],[35,40,45]])
      arr_2d
[Out]: array([[ 5, 10, 15],
              [20, 25, 30],
              [35, 40, 45]])
[In]: arr_2d[1]                 # Виділяємо стрічку з індексом '1'
[Out]: array([20, 25, 30])
[In]: arr_2d[1,:]               # Альтернативна форма вибору стрічки з індексом '1'
[Out]: array([20, 25, 30])
[In]: arr_2d[1][0]              # Елемент з індексом '0' зі стрічки з індексом '1'
[Out]: 20
```

```
[In]: arr_2d[1,0]          # Альтернативна форма вибору елемента
[Out]: 20
[In]: arr_2d[:2,1:]        # Обидаємо всі стрічки до другої (не включно)
                                # і всі стовбчики після першого (включно)
[Out]: array([[10, 15],
              [25, 30]])
```

Fancy Indexing – це спосіб вибору елементів з багатовимірного масиву одночасно шляхом передачі масиву індексів. Цей спосіб є коорисним, оскільки на відміну від індексації з його допомогою ми можемо обрати елементи які не стоять підряд:

```
[In]: arr2d = np.zeros((5,5))          # arr2d.shape[1] повертає кількість стовбчиків
    for i in range(arr2d.shape[1]):    # (індекс '1') в двовимірному масиві 'arr2d'
        arr2d[i] = i                  # Заповнюємо стрічки номерами їх індексів
    arr2d
[Out]: array([[0., 0., 0., 0., 0.],
              [1., 1., 1., 1., 1.],
              [2., 2., 2., 2., 2.],
              [3., 3., 3., 3., 3.],
              [4., 4., 4., 4., 4.]])
[In]: arr2d[[2,4]]                  # Одночасно обираємо дві стрічки з масиву
                                # з індексами '2' і '4'
[Out]: array([[2., 2., 2., 2., 2.],
              [4., 4., 4., 4., 4.]])
```

Подібно до арифметичних операцій, ми можемо застосовувати будь-який оператор порівняння до масиву NumPy. Він застосовуватиметься до кожного елемента масиву, а новий масив NumPy Array буде створений з відповідними значеннями True або False в тих місцях де умова була вмонана або ні, такі масиви ще називаються масками.

```
[In]: arr = np.arange(1,11)
    arr
[Out]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
[In]: bool_arr=arr > 4          # Визначаємо елементи, що задовольняють умові
    bool_arr                    # і присвоюємо їх змінній
[Out]: array([False, False, False, False,  True,
              True,  True,  True,  True,  True], dtype=bool)
[In]: arr[bool_arr]            # Використання маски для виділення елементів,
                                # що відповідали умові порівняння
[Out]: array([ 5,  6,  7,  8,  9, 10])
[In]: arr[arr>2]               # Альтернативний варіант використання масок
[Out]: array([ 3,  4,  5,  6,  7,  8,  9, 10])
```

Ми можемо виконувати з NumPy Array основні математичні операції, за умови якщо масиви однакової форми, не використовуючи індексацію чи циклів:

```
[In]: arr = np.arange(1,11)
    arr + arr
[Out]: array([2, 4, 6, 8, 10, 12, 14, 16, 18, 20])
[In]: arr * arr
[Out]: array([1, 4, 9, 16, 25, 36, 49, 64, 81, 100])
[In]: arr - arr
[Out]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
[In]: arr/arr
[Out]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
[In]: 1/arr
```

```
[Out]: array([1., 0.5, 0.33333333, 0.25, 0.2,
             0.16666667, 0.14285714, 0.125, 0.11111111, 0.1])
[In]: arr**3
[Out]: array([1, 8, 27, 64, 125, 216, 343, 512, 729, 1000])
```

NumPy надає можливість застосування функцій, які охоплюють широкий спектр операцій, що застосовуються одночасно до всіх елементів масиву. Наведемо приклади деяких з них як (корінь, експонента, максимальне значення, синус, логарифм натуральний):

```
[In]: np.sqrt(arr)
[Out]: array([1., 1.41421356, 1.73205081, 2., 2.23606798,
             2.44948974, 2.64575131, 2.82842712, 3., 3.16227766])
[In]: np.exp(arr)
[Out]: array([2.71828183e+00, 7.38905610e+00, 2.00855369e+01, 5.45981500e+01,
             1.48413159e+02, 4.03428793e+02, 1.09663316e+03, 2.98095799e+03,
             8.10308393e+03, 2.20264658e+04])
[In]: np.max(arr)
[Out]: 10
[In]: np.sin(arr)
[Out]: array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 , -0.95892427,
             -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849, -0.54402111])
[In]: np.log(arr)
[Out]: array([0.,          0.69314718, 1.09861229, 1.38629436, 1.60943791,
             1.79175947, 1.94591015, 2.07944154, 2.19722458, 2.30258509])
```

Більш детально про можливості бібліотеки NumPy ви можете ознайомитися на сайті за посиланням (<https://numpy.org/doc/stable/>)

2.2.2 Pandas

Pandas – це ще один важливий пакет для наукових обчислень в Python. Він застосовується для маніпулювання даними та їхнього аналізу, зокрема, роботи з чисельними таблицями та часовими рядами. Особливо, якщо ви працюєте з Python і шукаєте щось більш потужне, ніж Excel. Pandas використовує швидкі та гнучкі структури даних, покликані зробити роботу з реляційними або лейблованими даними легкою та інтуїтивно зрозумілою.

Для того, щоб використовувати бібліотеку Pandas, нам потрібно імпортувати її в нашу програму. За домовленістю, модуль pandas імпортується під псевдонімом `pd`, наступним чином:

```
[In]: import numpy as np
      import pandas as pd
```

Series – перший основний тип даних, про який ми дізнаємося для pandas. Ми вже імпортували Pandas тепер дослідимо об'єкт Series.

Series дуже схожа на масив NumPy (насправді він будується поверх об'єкта масиву NumPy). Що відрізняє масив NumPy від Series, так це те, що Series може мати мітки осей, тобто він може бути проіндексований міткою, а не просто розташуванням числа. Він також може містити будь-який довільний об'єкт Python.

Ви можете перетворити список, масив NumPy або словник у серію. Вивчимо цю концепцію на кількох прикладах:

```
[In]: labels = ['Галактика', 'Зоря', 'Спам']
      my_list = [10, 20, 30]
      arr = np.array([10, 20, 30])
      d = {'Галактика': 10, 'Зоря': 20, 'Спам': 30}
      pd.Series(data=my_list)          # використовуючи список
[Out]: 0    10
      1    20
      2    30
```

```

dtype: int64
[In]: pd.Series(data=my_list,index=labels)
[Out]: Галактика    10
      Зоря         20
      Спам         30
dtype: int64
[In]: pd.Series(my_list,labels)
[Out]: Галактика    10
      Зоря         20
      Спам         30
dtype: int64
[In]: pd.Series(arr)                                # використовуючи масив NumPy
[Out]: 0     10
      1     20
      2     30
dtype: int64
[In]: pd.Series(arr,labels)
[Out]: Галактика    10
      Зоря         20
      Спам         30
dtype: int64
[In]: pd.Series(d)                                # використовуючи словник
[Out]: Галактика    10
      Зоря         20
      Спам         30
dtype: int64

```

Series Pandas може вміщувати різні типи об'єктів:

```

[In]: pd.Series(data=labels)
[Out]: 0     Галактика
      1     Зоря
      2     Спам
dtype: object
[In]: pd.Series([sum,print,len])
[Out]: 0     <built-in function sum>
      1     <built-in function print>
      2     <built-in function len>
dtype: object

```

Ключем до використання Series є розуміння її індексу. Pandas використовує ці назви індексів або цифри, дозволяючи швидко шукати інформацію (працює як хеш-таблиця або словник).

Давайте подивимось кілька прикладів того, як отримати інформацію із Series. Створимо два серії, ser1 та ser2:

```

[In]: ser1 = pd.Series([1,2,3,4],index = ['WISEA', 'SDSS', '2MASS', 'LEDA'])
      ser1
[Out]: WISEA     1
      SDSS     2
      2MASS    3
      LEDA     4
dtype: int64
[In]: ser2 = pd.Series([1,2,5,4],index = ['WISEA', 'SDSS', '2MASS', 'LEDA'])
      ser2
[Out]: WISEA     1
      SDSS     2

```

```

2MASS      3
2dFGRS     4
dtype: int64
[In]: ser1['WISEA']
[Out]: 1

```

Також арифметичні операції можна виконувати з урахуванням індексу:

```

[In]: ser1 + ser2
[Out]: 2MASS      8.0
      2dFGRS     NaN
      LEDA       NaN
      SDSS       4.0
      WISEA      2.0
      dtype: float64

```

DataFrames є робочим коником Pandas. Ми можемо уявляти DataFrame як купу об'єктів Series, об'єднаних для спільного використання одного і того ж індексу. Давайме використаємо Pandas, щоб дослідити цю тему!

```

[In]: from numpy.random import randn
      np.random.seed(100)
      df = pd.DataFrame(randn(5,4),index='A B C D E'.split(),
      columns='Ra Dec z Mag'.split())
      df
[Out]:

```

	Ra	Dec	z	Mag
A	-1.749765	0.342680	1.153036	-0.252436
B	0.981321	0.514219	0.221180	-1.070043
C	-0.189496	0.255001	-0.458027	0.435163
D	-0.583595	0.816847	0.672721	-0.104411
E	-0.531280	1.029733	-0.438136	-1.118318

Давайме розглянемо різні методи виділення даних із DataFrame:

```

[In]: df['Ra']
[Out]: A    -1.749765
      B     0.981321
      C    -0.189496
      D    -0.583595
      E    -0.531280
      Name: Ra, dtype: float64

```

```

[In]: df[['z', 'Mag']]
[Out]:

```

	z	Mag
A	1.153036	-0.252436
B	0.221180	-1.070043
C	-0.458027	0.435163
D	0.672721	-0.104411
E	-0.438136	-1.118318

```

[In]: df.Ra [Out]: A -1.749765 B 0.981321 C -0.189496 D -0.583595 E -0.531280 Name: Ra,
dtype: float64 [In]: type(df['Dec']) [Out]: pandas.core.series.Series

```

Приклад створення нового стовбчика шляхом додавання двох інших:

```

[In]: df['new'] = df['Ra'] + df['Dec']
[In]: df
[Out]:

```

	Ra	Dec	z	Mag	new
A	-1.749765	0.342680	1.153036	-0.252436	-1.407085
B	0.981321	0.514219	0.221180	-1.070043	1.495540
C	-0.189496	0.255001	-0.458027	0.435163	0.065506
D	-0.583595	0.816847	0.672721	-0.104411	0.233252
E	-0.531280	1.029733	-0.438136	-1.118318	0.498452

Приклад видалення стовбчика:

```
[In]: df.drop('new',axis=1)
OR
[In]: df.drop('new',axis=1, inplace=True)
[Out]:
```

	Ra	Dec	z	Mag
A	-1.749765	0.342680	1.153036	-0.252436
B	0.981321	0.514219	0.221180	-1.070043
C	-0.189496	0.255001	-0.458027	0.435163
D	-0.583595	0.816847	0.672721	-0.104411
E	-0.531280	1.029733	-0.438136	-1.118318

Приклад виділення рядка:

```
[In]: df.loc['A']
[Out]: Ra      -1.540616
      Dec      2.046714
      z       -1.396999
      Mag     -1.097172
      Name: A, dtype: float64
[In]: df.iloc[2]
[Out]: Ra      0.894598
      Dec      0.759693
      z       -1.497720
      Mag     -1.193886
      Name: C, dtype: float64
[In]: df.loc['A','Dec']
[Out]: 2.0467139684821385
[In]: df.loc[['A','B'],['Ra','Dec']]
[Out]:
```

	Ra	Dec
A	-1.749765	0.342680
B	0.981321	0.514219

Давайте подивимося, як вибрати рядки на основі деяких умов у Pandas DataFrame за допомогою арифметичних операторів '>', '=', '<=', '!=', 'in'.

```
[In]: df
[Out]:
```

	Ra	Dec	z	Mag
A	-1.749765	0.342680	1.153036	-0.252436
B	0.981321	0.514219	0.221180	-1.070043
C	-0.189496	0.255001	-0.458027	0.435163
D	-0.583595	0.816847	0.672721	-0.104411
E	-0.531280	1.029733	-0.438136	-1.118318

```
[In]: df>0
[Out]:
```

	Ra	Dec	z	Mag
A	False	True	True	False
B	True	True	True	False
C	False	True	False	True
D	False	True	True	False
E	False	True	False	False

```
[In]: df[df>0]
[Out]:
```

	Ra	Dec	z	Mag
A	NaN	0.342680	1.153036	NaN
B	0.981321	0.514219	0.221180	NaN
C	NaN	0.255001	NaN	0.435163
D	NaN	0.816847	0.672721	NaN
E	NaN	1.029733	NaN	NaN

```
[In]: df[df['Ra']>0]
[Out]:
```

	Ra	Dec	z	Mag
B	0.981321	0.514219	0.221180	-1.070043

```
[In]: df[df['Ra']>0]['Dec']
[Out]: B    0.514219
      Name: Dec, dtype: float64
```

```
[In]: df[df['Ra']>0][['z','Mag']]
[Out]:
```

	z	Mag
B	0.22118	-1.070043

Для двох умов ви можете використовувати & з дужками:

```
[In]: df[(df['Dec']>0) & (df['z'] > 1)]
[Out]:
```

	Ra	Dec	z	Mag
A	-1.749765	0.342680	1.153036	-0.252436

Давайте обговоримо ще деякі особливості індексації, включаючи скидання індексу або встановлення іншого. Ми також наведемо приклади що до ієрархії індексів!

```
[In]: df.reset_index()
[Out]:
```

index		Ra	Dec	z	Mag
0	A	-1.749765	0.342680	1.153036	-0.252436
1	B	0.981321	0.514219	0.221180	-1.070043
2	C	-0.189496	0.255001	-0.458027	0.435163
3	D	-0.583595	0.816847	0.672721	-0.104411
4	E	-0.531280	1.029733	-0.438136	-1.118318

Давайте розглянемо як працювати з Multi-Index, спочатку ми створимо короткий приклад того, як би виглядав Multi-Index DataFrame:

```
[In]: outside = ['Star1','Star1','Star1','Star2','Star2','Star2']
      inside = [1,2,3,1,2,3]
      hier_index = list(zip(outside,inside))
      hier_index = pd.MultiIndex.from_tuples(hier_index)
      hier_index
```

```
[Out]: MultiIndex([('Star1', 1),
                  ('Star1', 2),
                  ('Star1', 3),
                  ('Star2', 1),
                  ('Star2', 2),
                  ('Star2', 3)],

[In]: df = pd.DataFrame(np.random.randn(6,2),index=hier_index,columns=['A','B'])
df
[Out]:
```

		A	B
Star1	1	1.618982	1.541605
	2	-0.251879	-0.842436
	3	0.184519	0.937082
Star2	1	0.731000	1.361556
	2	-0.326238	0.055676
	3	0.222400	-1.443217

Тепер покажемо, як це індексувати! Для ієрархії індексів ми використовуємо `df.loc []`, якщо це було на осі стовпців, ви б просто використовували нотацію звичайних дужок `df []`. Виклик одного рівня індексу повертає підкадр даних:

```
[In]: df.loc['Star1']
[Out]:
```

	A	B
1	1.618982	1.541605
2	-0.251879	-0.842436
3	0.184519	0.937082

```
[In]: df.loc['Star1'].loc[1]
[Out]: A    1.618982
       B    1.541605
       Name: 1, dtype: float64
[In]: df.index.names
[Out]: FrozenList([None, None])
[In]: df.index.names = ['Group','Num']
[In]: df
[Out]:
```

Group	Num	A	B
Star1	1	1.618982	1.541605
	2	-0.251879	-0.842436
	3	0.184519	0.937082
Star2	1	0.731000	1.361556
	2	-0.326238	0.055676
	3	0.222400	-1.443217

```
[In]: df.xs('Star1')
[Out]:
```

	A	B
Num		
1	1.618982	1.541605
2	-0.251879	-0.842436
3	0.184519	0.937082

Цей метод дозволяє заглядати глибше в групи і одержувати інформацію по підгрупі в кожній групі.

```
[In]: df.xs(['Star1',1])
[Out]: A    1.618982
       B    1.541605
       Name: (Star1, 1), dtype: float64
```

Дані не завжди бувають повними. Іноді трапляються пошкоджені файли або таблиці з пропущеними значеннями. Їх можна викинути чи замінити на серене значення по рядку чи по стовбчику.

```
[In]: import numpy as np
import pandas as pd
df = pd.DataFrame({'A':[1, 2, np.nan],
                  'B':[3, np.nan, np.nan],
                  'C':[4, 5, 3]})

df
[Out]:
```

	A	B	C
0	1.0	3.0	4
1	2.0	NaN	5
2	NaN	NaN	3

Якщо ми просто використаємо команду що викидає всі пропущені значення, то ми отримаємо тільки ті рядки в яких всі данні присутні.

```
[In]: df.dropna()
[Out]:
```

	A	B	C
0	1.0	3.0	4

Але якщо вам необхідно виділити колонки що є повністю заповненні, а не рядки, то потрібно вказати вісь.

```
[In]: df.dropna(axis=1)
[Out]:
```

	C
0	4
1	5
2	3

Якщо ж ви хочете відкинути тільки ті рядки чи колонки кількість пропущених значень певного значення, ви можете використати параметр `thresh`.

```
[In]: df.dropna(thresh=2)
[Out]:
```

	A	B	C
0	1.0	3.0	4
1	2.0	NaN	5

Також ми можемо замінити наші пропущені значення на будь-яке інше значення. Наприклад на слово "Good".

```
[In]: df.fillna(value='Good')
[Out]:
```

	A	B	C
0	1.0	3.0	4
1	2.0	Good	5
2	Good	Good	3

Ми також можемо обирати в якому саме стовбчику ми хочемо замінити значення і на яке. Наприклад в даному випадку ми замінюємо всі відсутні значення в стовбчику на середнє значення даних з цього стовбчика.

```
[In]: df['A'].fillna(value=df['A'].mean())
[Out]: 0    1.0
       1    2.0
       2    1.5
       Name: A, dtype: float64
```

Групування дозволяє групувати разом рядки базуючись на колонках виконуючи певні операції над ними (сумування, визначення середнього і тд.).

```
[In]: import pandas as pd
      data = {'Telescop': ['Star', 'Star', 'Planet', 'Planet', 'Galaxy', 'Galaxy'],
              'Name': ['Sun', 'Betelgeuse', 'Earth', 'Jupiter', 'Milky Way', 'Andromeda'],
              'Parameres': [222, 133, 444, 556, 668, 887]}
      df = pd.DataFrame(data)
      df
```

```
[Out]:
```

	Telescop	Name	Parameres
0	Star	Sun	222
1	Star	Betelgeuse	133
2	Planet	Earth	444
3	Planet	Jupiter	556
4	Galaxy	Milky Way	668
5	Galaxy	Andromeda	887

Ми можемо зберегти групу, як об'єкт, а потім проводити над нею різні операції.

```
[In]: df.groupby('Telescop')
[Out]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f6ad1cd6c50>
```

Наприклад, ми можемо визначити середнє значення в групі.

```
[In]: by_comp = df.groupby("Telescop")
      by_comp.mean()
[Out]:
```

	Parameres
Telescop	
Galaxy	777.5
Planet	500.0
Star	177.5

Що еквівалентно виразу:

```
[In]: df.groupby('Telescop').mean()
[Out]:
```

	Parameres
Telescop	
Galaxy	777.5
Planet	500.0
Star	177.5

Також ми можемо використовувати інші операції:
Середньоквадратичне відхилення

```
[In]: by_comp.std()
[Out]:
```

Parameres**Telescop**

Galaxy	154.856385
Planet	79.195959
Star	62.932504

Мінімальне значення

[In]: by_comp.min()

[Out]:

	Name	Parameres
Telescop		
Galaxy	Andromeda	668
Planet	Earth	444
Star	Betelgeuse	133

Максимальне значення

[In]: by_comp.max()

[Out]:

	Name	Parameres
Telescop		
Galaxy	Milky Way	887
Planet	Jupiter	556
Star	Sun	222

Кількість елементів

[In]: by_comp.count()

[Out]:

	Name	Parameres
Telescop		
Galaxy	2	2
Planet	2	2
Star	2	2

Опис групи

[In]: by_comp.describe()

[Out]:

	Parameres							
	count	mean	std	min	25%	50%	75%	max
Telescop								
Galaxy	2.0	777.5	154.856385	668.0	722.75	777.5	832.25	887.0
Planet	2.0	500.0	79.195959	444.0	472.00	500.0	528.00	556.0
Star	2.0	177.5	62.932504	133.0	155.25	177.5	199.75	222.0

Також ми можемо змінювати форму, щоб замінити рядки стовбчики і навпаки.

[In]: by_comp.describe().transpose()

[Out]:

	Telescop	Galaxy	Planet	Star
Parameres				
count	2.000000	2.000000	2.000000	2.000000
mean	777.500000	500.000000	177.500000	
std	154.856385	79.195959	62.932504	
min	668.000000	444.000000	133.000000	
25%	722.750000	472.000000	155.250000	
50%	777.500000	500.000000	177.500000	
75%	832.250000	528.000000	199.750000	
max	887.000000	556.000000	222.000000	

А також можемо отримати інформацію по кожній підгрупі в групі.

```
[In]: by_comp.describe().transpose()['Star']
[Out]: Parameres  count      2.000000
        mean      177.500000
        std       62.932504
        min      133.000000
        25%      155.250000
        50%      177.500000
        75%      199.750000
        max      222.000000
        Name: Star, dtype: float64
```

Іноді доводиться об'єднувати декілька таблиць в одну, для цього в pandas передбачені певні методи. Для початку, створимо декілька таблиць

```
[In]: df1 = pd.DataFrame({'S': ['Star0', 'Star1', 'Star2', 'Star3'],
                          'P': ['Planet0', 'Planet1', 'Planet2', 'Planet3'],
                          'G': ['Galaxy0', 'Galaxy1', 'Galaxy2', 'Galaxy3'],
                          'T': ['Telescop0', 'Telescop1', 'Telescop2', 'Telescop3']},
                          index=[0, 1, 2, 3])
df2 = pd.DataFrame({'S': ['Star4', 'Star5', 'Star6', 'Star7'],
                    'P': ['Planet4', 'Planet5', 'Planet6', 'Planet7'],
                    'G': ['Galaxy4', 'Galaxy5', 'Galaxy6', 'Galaxy7'],
                    'T': ['Telescop4', 'Telescop5', 'Telescop6', 'Telescop7']},
                    index=[4, 5, 6, 7])
df3 = pd.DataFrame({'S': ['Star8', 'Star9', 'Star10', 'Star11'],
                    'P': ['Planet8', 'Planet9', 'Planet10', 'Planet11'],
                    'G': ['Galaxy8', 'Galaxy9', 'Galaxy10', 'Galaxy11'],
                    'T': ['Telescop8', 'Telescop9', 'Telescop10', 'Telescop11']},
                    index=[8, 9, 10, 11])
```

Таблиця 1

```
[In]: df1
[Out]:
```

	S	P	G	T
0	Star0	Planet0	Galaxy0	Telescop0
1	Star1	Planet1	Galaxy1	Telescop1
2	Star2	Planet2	Galaxy2	Telescop2
3	Star3	Planet3	Galaxy3	Telescop3

Таблиця 2

```
[In]: df2
[Out]:
```

	S	P	G	T
4	Star4	Planet4	Galaxy4	Telescop4
5	Star5	Planet5	Galaxy5	Telescop5
6	Star6	Planet6	Galaxy6	Telescop6
7	Star7	Planet7	Galaxy7	Telescop7

Таблиця 3

```
[In]: df3
[Out]:
```

	S	P	G	T
8	Star8	Planet8	Galaxy8	Telescop8
9	Star9	Planet9	Galaxy9	Telescop9
10	Star10	Planet10	Galaxy10	Telescop10
11	Star11	Planet11	Galaxy11	Telescop11

Concatenation - об'єднує DataFrames. Майте на увазі, що розміри повинні збігатися вздовж осі, по якій ви об'єднуєте. Ви можете передавати цій функції список DataFrames, які ви хочете об'єднати.

```
[In]: pd.concat([df1,df2,df3])
[Out]:
```

	S	P	G	T
0	Star0	Planet0	Galaxy0	Telescop0
1	Star1	Planet1	Galaxy1	Telescop1
2	Star2	Planet2	Galaxy2	Telescop2
3	Star3	Planet3	Galaxy3	Telescop3
4	Star4	Planet4	Galaxy4	Telescop4
5	Star5	Planet5	Galaxy5	Telescop5
6	Star6	Planet6	Galaxy6	Telescop6
7	Star7	Planet7	Galaxy7	Telescop7
8	Star8	Planet8	Galaxy8	Telescop8
9	Star9	Planet9	Galaxy9	Telescop9
10	Star10	Planet10	Galaxy10	Telescop10
11	Star11	Planet11	Galaxy11	Telescop11

Merge – злиття, також використовується для об'єднання двох таблиць, наприклад, якщо необхідно додати додаткові особливості, ми можемо це зробити базуючись на якійсь колонці що присутня в обох таблицях, скажімо це може бути номер об'єкту в каталозі. Для прикладу створимо ще пару таблиць.

```
[In]: first = pd.DataFrame({'Parameters1': ['P0', 'P0', 'P1', 'P2'],
                             'Parameters2': ['P0', 'P1', 'P0', 'P1'],
                             'A': ['Star0', 'Star1', 'Star2', 'Star3'],
                             'B': ['Planet0', 'Planet1', 'Planet2', 'Planet3']})

second = pd.DataFrame({'Parameters1': ['P0', 'P1', 'P1', 'P2'],
                       'Parameters2': ['P0', 'P0', 'P0', 'P0'],
                       'C': ['Galaxy0', 'Galaxy1', 'Galaxy2', 'Galaxy03'],
                       'D': ['Telescop0', 'Telescop1', 'Telescop2', 'Telescop3']})
```

Тут ми об'єднуємо дві таблиці таким чином щоб вони мали спільні колонки 'Parameters1' та 'Parameters2'.

```
[In]: pd.merge(first, second, on=['Parameters1', 'Parameters2'])
[Out]:
```

	Parameters1	Parameters2	A	B	C	D
0	P0	P0	Star0	Planet0	Galaxy0	Telescop0
1	P1	P0	Star2	Planet2	Galaxy1	Telescop1
2	P1	P0	Star2	Planet2	Galaxy2	Telescop2

Також цей метод містить такий параметр об'єднання як `how` який може приймати значення 'inner', 'outer', 'left', 'right'. Але при використанні цього параметра треба бути обережним з пропумками які можуть виникнути.

```
[In]: pd.merge(first, second, how='outer', on=['Parameters1', 'Parameters2'])
[Out]:
```

	Parameters1	Parameters2	A	B	C	D
0	P0	P0	Star0	Planet0	Galaxy0	Telescop0
1	P0	P1	Star1	Planet1	NaN	NaN
2	P1	P0	Star2	Planet2	Galaxy1	Telescop1
3	P1	P0	Star2	Planet2	Galaxy2	Telescop2
4	P2	P1	Star3	Planet3	NaN	NaN
5	P2	P0	NaN	NaN	Galaxy03	Telescop3

```
[In]: pd.merge(first, second, how='right', on=['Parameters1', 'Parameters2'])
[Out]:
```

	Parameters1	Parameters2	A	B	C	D
0	P0	P0	Star0	Planet0	Galaxy0	Telescop0
1	P1	P0	Star2	Planet2	Galaxy1	Telescop1
2	P1	P0	Star2	Planet2	Galaxy2	Telescop2
3	P2	P0	NaN	NaN	Galaxy03	Telescop3

```
[In]: pd.merge(first, second, how='left', on=['Parameters1', 'Parameters2'])
[Out]:
```

	Parameters1	Parameters2	A	B	C	D
0	P0	P0	Star0	Planet0	Galaxy0	Telescop0
1	P0	P1	Star1	Planet1	NaN	NaN
2	P1	P0	Star2	Planet2	Galaxy1	Telescop1
3	P1	P0	Star2	Planet2	Galaxy2	Telescop2
4	P2	P1	Star3	Planet3	NaN	NaN

Joining-приєднання, є зручним методом для об'єднання стовбчиків двох по-різному проіндексованих DataFrames в один. Для демонстрації знову створимо дві таблиці даних.

```
[In]: first = pd.DataFrame({'S': ['Star0', 'Star1', 'Star2'],
                             'P': ['Planet0', 'Planet1', 'Planet2']},
                             index=['P0', 'P1', 'P2'])

second = pd.DataFrame({'G': ['Galaxy0', 'Galaxy2', 'Galaxy3'],
                        'T': ['Telescop0', 'Telescop2', 'Telescop3']},
                        index=['P0', 'P2', 'P3'])

[In]: first.join(second)
[Out]:
```

	S	P	G	T
P0	Star0	Planet0	Galaxy0	Telescop0
P1	Star1	Planet1	NaN	NaN
P2	Star2	Planet2	Galaxy2	Telescop2

Цей метод також містить параметр 'how'.

```
[In]: first.join(second, how='outer')
[Out]:
```

	S	P	G	T
P0	Star0	Planet0	Galaxy0	Telescop0
P1	Star1	Planet1	NaN	NaN
P2	Star2	Planet2	Galaxy2	Telescop2
P3	NaN	NaN	Galaxy3	Telescop3

До даних можна застосовувати певні операції, приклади яких ми наведемо нижче. Для цього ми створимо новий набір даних.

```
[In]:import pandas as pd
      df = pd.DataFrame({'col1':[1,2,3,4],
                        'col2':[555,666,777,444],
                        'col3':['Sun','Planet','Star','Galaxy']})

      df.head()
[Out]:
```

	col1	col2	col3
0	1	555	Sun
1	2	666	Planet
2	3	777	Star
3	4	444	Galaxy

Ця операція дозволяє отримати масив значень без повторів другої колонки.

```
[In]: df['col2'].unique()
[Out]: array([555, 666, 777, 444])
```

Ця операція дозволяє отримати кількість унікальних значень без повторів другої колонки.

```
[In]: df['col2'].nunique()
[Out]: 4
```

Ця операція рахує кількість посторів для кожного значення.

```
[In]: df['col2'].value_counts()
[Out]: 444    1
       555    1
       666    1
       777    1
       Name: col2, dtype: int64
```

Також використовуючі маски, ми можемо обирати ті значення з таблиці які відповідають заданим умовам. Наприклад, ті рядки, для яких значення в першій колонці більше 2, а в другій, тотожне 444.

```
[In]: newdf = df[(df['col1']>2) & (df['col2']==444)]
      newdf
[Out]:
```

	col1	col2	col3
3	4	444	Galaxy

Також одним з найпотужніших методів в pandas є застосування функцій до масивів даних. Для демонстрації визначимо функцію times2 і застосуємо її до першого стовбчика.

```
[In]: def times2(x):
      return x*2
      df['col1'].apply(times2)
[Out]: 0    2
       1    4
       2    6
       3    8
       Name: col1, dtype: int64
```

Визначення довжини елементів третього стовбчика

```
[In]: df['col3'].apply(len)
[Out]: 0    3
       1    6
       2    4
       3    6
```

```
Name: col3, dtype: int64
```

Сума всіх елементів першого стовбчика:

```
[In]: df['col1'].sum()
[Out]: 10
```

Видалення першого стовбчика:

```
[In]: del df['col1']
      df
[Out]:
```

	col2	col3
0	555	Sun
1	666	Planet
2	777	Star
3	444	Galaxy

Повертає список заголовків:

```
[In]: df.columns
[Out]: Index(['col2', 'col3'], dtype='object')
```

Сортування таблиці за другим стовбчиком:

```
[In]: df.sort_values(by='col2')
[Out]:
```

	col2	col3
3	444	Galaxy
0	555	Sun
1	666	Planet
2	777	Star

Перевірка на наявність пустих значень:

```
[In]: df.isnull()
[Out]:
```

	col2	col3
3	False	False
0	False	False
1	False	False
2	False	False

2.3. Matplotlib

Візуалізація даних може дати вам важливу інформацію для їх подальшого аналізу. Matplotlib є стандартним пакетом для побудови двовимірної та тривимірної графіки за допомогою Python. Його створив Джон Хантер. Для того, щоб завантажити Matplotlib необхідно завантажити підмодуль pyplot пакета matplotlib.

```
[In]: import matplotlib.pyplot as plt
```

Давайте розглянемо простий приклад з використанням двох масивів numpy. Ви також можете використовувати списки, але, швидше за все, ви будете передавати масиви numpy або стовпці pandas (які, по суті, також поведуться як масиви).

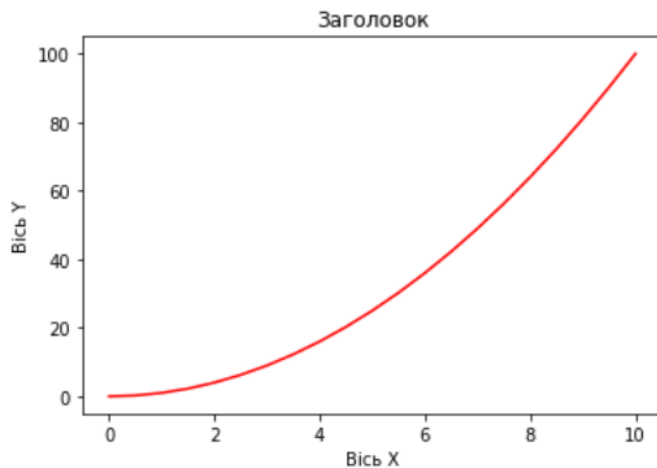
```
[In]: import numpy as np
      x = np.linspace(0, 10, 21)
      y = x ** 2
```

Виведемо масив x та y :

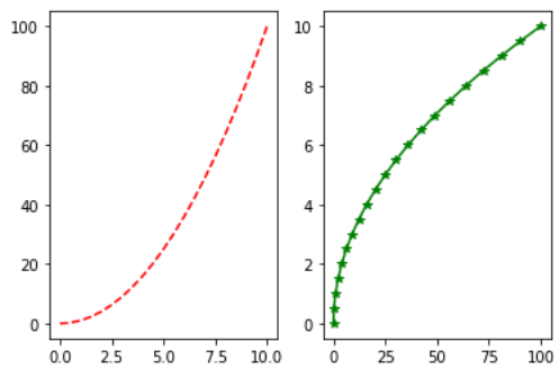
```
[In]: x
[Out]: array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
            5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5, 10. ])
[In]: y
[Out]: array([ 0. ,  0.25,  1. ,  2.25,  4. ,  6.25,  9. , 12.25,
            16. , 20.25, 25. , 30.25, 36. , 42.25, 49. , 56.25,
            64. , 72.25, 81. , 90.25, 100. ])
```

Наразі ми маємо два масиви і можемо створити дуже простий лінійний графік:

```
[In]: plt.plot(x, y, 'r')          # 'r' - червоний колір
      plt.xlabel('Вісь X')
      plt.ylabel('Вісь Y')
      plt.title('Заголовок')
      plt.show()
[Out]:
```



```
[In]: # plt.subplot(N-рядків, N-колонок, порядковий номер графіка)
      plt.subplot(1,2,1)
      plt.plot(x, y, 'r--')
      plt.subplot(1,2,2)
      plt.plot(y, x, 'g*-');
[Out]:
```



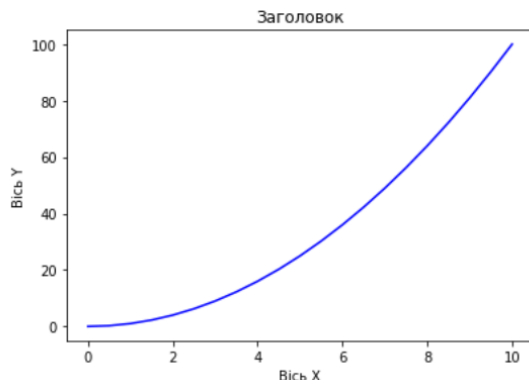
При використанні Matplotlib для побудови графіків доступні два типи інтерфейсів, перший ми з вами використали - це інтерфейс `pyplot`, другий - це об'єктно-орієнтований інтерфейс. Перший варіант будує графіки на поточній площині і осях, а от другий вимагає ініціювання нової площини та осей при побудові нового графіка. Не можна сказати що якийсь з варіантів гірший чи кращий, потрібно звертати увагу на поставлену задачу.

Більшість термінів які потрібно знати вам для використання Matplotlib інтуїтивно зрозумілі та відображені на малюнку. Figure - це остаточне зображення, яке може містити 1 або більше осей (axes). Axes (осі) представляють собою окремий графік (plot). Як тільки ви зрозумієте, що це таке і як отримати до них доступ через об'єктно-орієнтована API, інша частина процесу стане на свої місця.

```
[In]: # Створюємо новий Figure (пуста конва)
fig = plt.figure()
# Додаємо осі до малюнка
axes = fig.add_axes([0.1, 0.1, 0.8, 0.8])
# зліва, знизу, ширина, висота (діапазон від 0 до 1)

# Нанесимо сам графік на цей набір осей
axes.plot(x, y, 'b')
axes.set_xlabel('Вісь X')
axes.set_ylabel('Вісь Y')
axes.set_title('Заголовок')
```

[Out]:



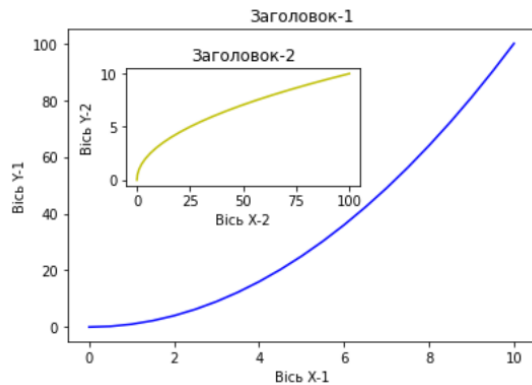
```
[In]: # Створюємо новий Figure (пуста конва)
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # головні осі
axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # вбудовані осі

# Опис основного графіка
axes1.plot(x, y, 'b')
axes1.set_xlabel('Вісь X-1')
axes1.set_ylabel('Вісь Y-1')
axes1.set_title('Заголовок-1')

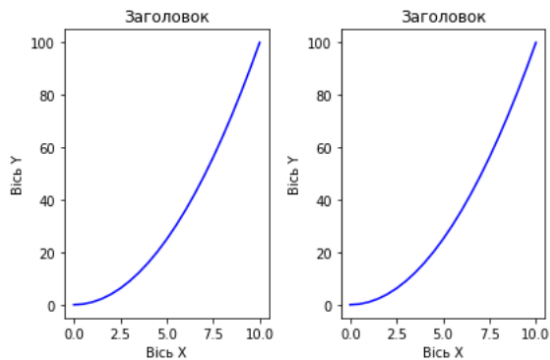
# Опис вбудованого графіка
axes2.plot(y, x, 'y')
axes2.set_xlabel('Вісь X-2')
axes2.set_ylabel('Вісь Y-2')
axes2.set_title('Заголовок-2');
```

[Out]:



```
[In]: # Створюємо нову порожню конву з можливістю розмістити два графіка
fig, axes = plt.subplots(nrows=1, ncols=2)
# Використовуємо об'єкт axes для додавання матеріалу і огортаємо це в цикл for
for ax in axes:
    ax.plot(x, y, 'b')
    ax.set_xlabel('Вісь X')
    ax.set_ylabel('Вісь Y')
    ax.set_title('Заголовок')
# Виводимо зображення
fig
plt.tight_layout()
```

[Out]:



Існують різні варіації розміщення легенди:

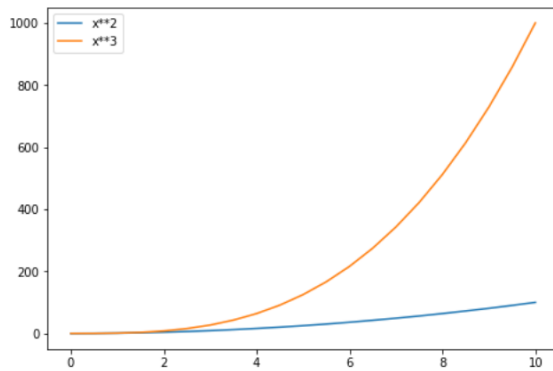
- верхній правий кут - `ax.legend(loc=1)`
- верхній лівий кут - `ax.legend(loc=2)`
- нижній лівий кут - `ax.legend(loc=3)`
- нижній правий кут - `ax.legend(loc=4)`
- коли `matplotlib` сам вирішить оптимальне місце розташування - `ax.legend(loc=0)`

```
[In]: fig = plt.figure()

ax = fig.add_axes([0,0,1,1])

ax.plot(x, x**2, label="x**2")
ax.plot(x, x**3, label="x**3")
ax.legend(loc=0)
```

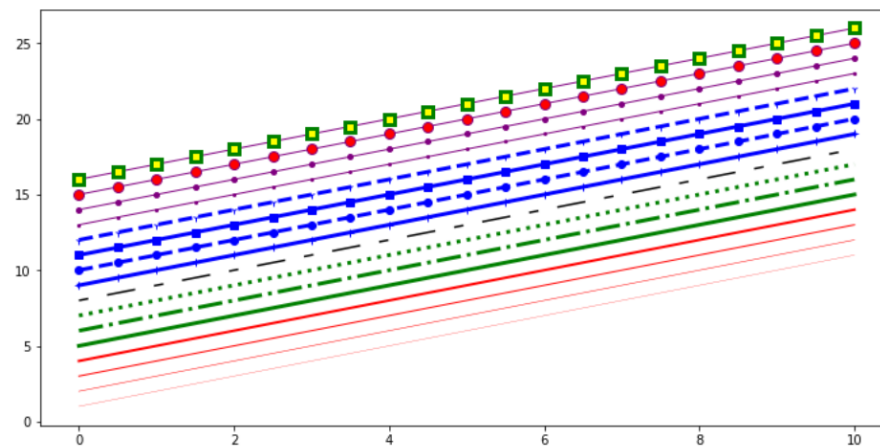
[Out]:



```
[In]: fig, ax = plt.subplots(figsize=(12,6))
```

```
ax.plot(x, x+1, color="red", linewidth=0.25)
ax.plot(x, x+2, color="red", linewidth=0.50)
ax.plot(x, x+3, color="red", linewidth=1.00)
ax.plot(x, x+4, color="red", linewidth=2.00)
# можливі варіанти типу лінії '-', '--', '-.', ':', 'steps'
ax.plot(x, x+5, color="green", lw=3, linestyle='--')
ax.plot(x, x+6, color="green", lw=3, ls='-.')
ax.plot(x, x+7, color="green", lw=3, ls=':')
# тип
line, = ax.plot(x, x+8, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # формат: довжина рядка, довжина простору, ...
# можливі символи для маркера: маркер = '+', 'o', '*', 's',
# ',', ' ', '1', '2', '3', '4', ...
ax.plot(x, x+ 9, color="blue", lw=3, ls='-', marker='+')
ax.plot(x, x+10, color="blue", lw=3, ls='--', marker='o')
ax.plot(x, x+11, color="blue", lw=3, ls='-', marker='s')
ax.plot(x, x+12, color="blue", lw=3, ls='--', marker='1')
# розмір і колір маркера
ax.plot(x, x+13, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, x+14, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersize=8,
        markerfacecolor="red")
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=3, markeredgecolor="green");
```

[Out]:



- <http://www.matplotlib.org> - Головна сторінка самопроєкту matplotlib, на ви знайдете посилання на багато інших прикладів його використання.
- <https://github.com/matplotlib/matplotlib> - Вихідний код matplotlib.
- <http://matplotlib.org/gallery.html> - Дуже рекомендую скористатись даним посиланням, що містить велику галерею з різними типами сюжетів, які може створити з matplotlib.
- <http://www.loria.fr/~rougier/teaching/matplotlib> - Гарний підручник з matplotlib.

2.4. Seaborn

Дослідники витрачають велику кількість часу на очищення даних і приведення їх до формату з яким можна працювати далі для вирішення своїх задач. Багато хто стверджує, що етап отримання та очищення даних складають 80% роботи.

Якщо ви тільки починаєте працювати з даними або плануєте, важливо знати що ви стикнетесь з безладними даними, незалежно від того, чи це означає відсутність значень, невідповідне форматування, неправильні записи або безглузді викиди.

В даному методичному посібнику ми покажемо роботу вже з репрезентативною вибіркою галактик, очищеною від дублікатів, зірок, та з візуально визначеними морфологічними типами.

Для цього файл треба попередньо завантажити на Google Disk і далі можна опрацювати його за допомогою Google Colab. Для завантаження файлів ми будемо використовувати Google drive.

```
[In]: import numpy as np
import pandas as pd
from google.colab import drive
drive.mount("/content/gdrive/")
df = pd.read_csv('/content/gdrive/My Drive/6163_morphology.csv')
```

Щоб швидко проаналізувати зміст прочитаного файла зручно використовувати:

- функцію *head()*, яка повертає заголовки DataFrame;

```
[In]: df.head()
```

- функцію *info()* для друку стислого резюме DataFrame, включаючи тип dindex та dtype типів стовпців, ненульові значення та використання пам'яті.

```
[In]: df.info()
```

- функцію *describe()* для обчислення деяких статистичних даних, таких як проценти, середнє та стандартне значення числових значень Series або DataFrame також числові та об'єктні ряди, та набори стовпців DataFrame змішаних типів даних.

```
[In]: df.describe()
```

В pandas ви можете швидко отримати назви заголовків вашої таблиці.

```
[In]: df.columns
[Out]: Index(['n', 'Ra', 'dec', 'Mu', 'Mg', 'Mr', 'Mi', 'Mz', 'Mu-Mr', 'Mg-Mi',
            'Mr-Mz', 'R50/R90', 'z', 'Type'],
            dtype='object')
```

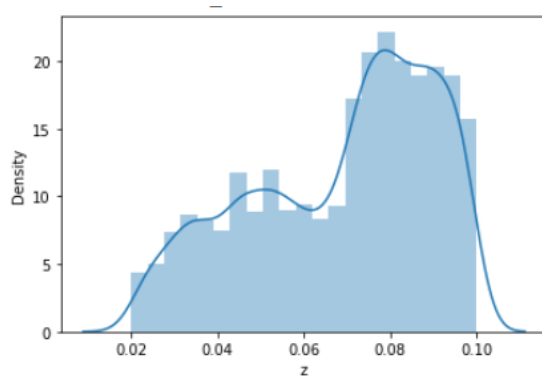
При аналізі даних часто буває корисним побудова графіків одного і того ж типу, наприклад, графіків розподілу, які містять різні підмножини з цих даних. При такому підході виходить багато графіків, але всі вони, в сукупності, дають хороше уявлення про особливості в даних - допомагають визначити напрям в якому потрібно досліджувати. Seaborn - бібліотека на основі matplotlib, що містить функції, які як раз і дозволяють будувати кілька графіків (областей Axes) на одному зображенні (області Figure). Одержуваний в результаті багатосегментний графік в точності відповідає структурі даних.

```
[In]: import seaborn as sns
```

За допомогою *distplot* можна побудувати розподіл однієї величини, наприклад:

```
[In]: sns.distplot(df['z'])
```

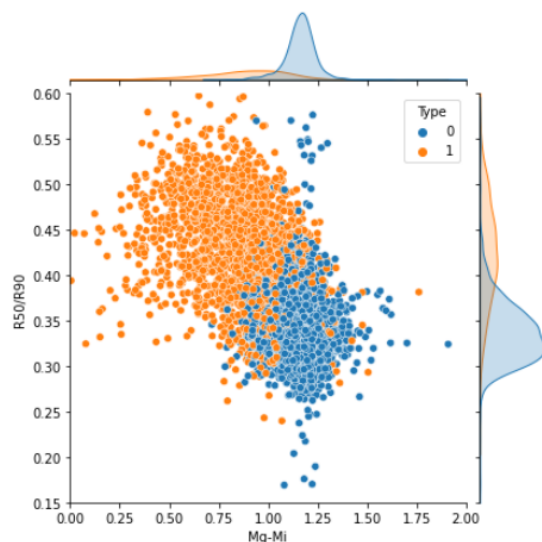
```
[Out]:
```



Для побудови розподілу двох величин допоможе функція *jointplot()*. Вона має параметр *kind* який може набувати таких значень:

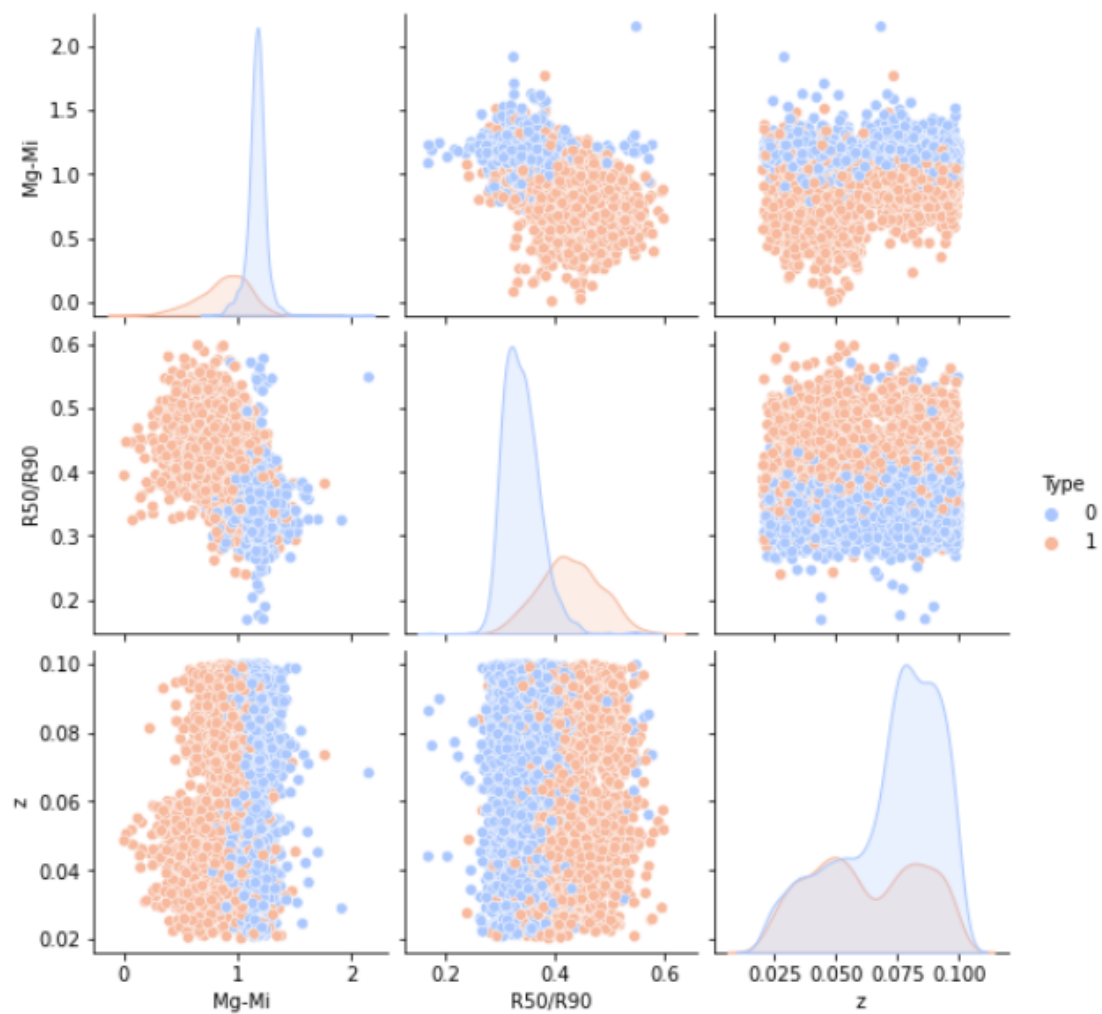
```
[In]: sns.jointplot(x='Mg-Mi', y='R50/R90', data=df, kind='scatter',
                    xlim=(0, 2), ylim=(0.15, 0.6), hue='Type')
```

```
[Out]:
```



```
[In]: sns.pairplot(df[['Mg-Mi', 'R50/R90', 'z', 'Type']],
                    hue='Type', palette='coolwarm')
```

```
[Out]:
```



3. Linear Regression

```
[In]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
[In]: from google.colab import drive
drive.mount("/content/gdrive/")
df = pd.read_csv('/content/gdrive/My Drive/6163_morphology.csv')
```

```
[In]: df.head()
```

```
[Out]:
```

	n	Ra	dec	Mu	Mg	Mr	Mi	Mz	Mu-Mr	Mg-Mi	Mr-Mz	R50/R90	z	Type
0	14300	198.807794	44.407360	-19.465374	-20.775398	-21.382226	-21.684707	-21.973335	1.916852	0.909308	0.591109	0.441292	0.035448	1
1	30643	207.508440	19.370078	-20.304267	-22.027810	-22.736804	-23.084715	-23.370877	2.432537	1.056905	0.634073	0.286919	0.062382	0
2	427	113.910060	45.553409	-18.628109	-20.511564	-21.333896	-21.725232	-22.054910	2.705788	1.213667	0.721014	0.311150	0.081059	0
3	31072	217.459165	25.678422	-18.943683	-20.232374	-20.775293	-21.075923	-21.311882	1.831609	0.843549	0.536589	0.497877	0.094052	1
4	43203	227.902793	6.181272	-18.668989	-20.430259	-21.091646	-21.411507	-21.651996	2.422656	0.981248	0.560351	0.343024	0.081744	0

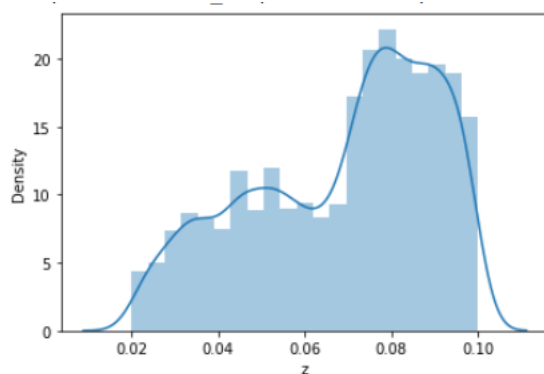
```
[In]: df.info()
```

```
[Out]:
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6163 entries, 0 to 6162
Data columns (total 14 columns):
#   Column      Non-Null Count  Dtype
---  -
0    n           6163 non-null   int64
1    Ra          6163 non-null   float64
2    dec         6163 non-null   float64
3    Mu          6163 non-null   float64
4    Mg          6163 non-null   float64
5    Mr          6163 non-null   float64
6    Mi          6163 non-null   float64
7    Mz          6163 non-null   float64
8    Mu-Mr       6163 non-null   float64
9    Mg-Mi       6163 non-null   float64
10   Mr-Mz       6163 non-null   float64
11   R50/R90     6163 non-null   float64
12   z           6163 non-null   float64
13   Type        6163 non-null   int64
dtypes: float64(12), int64(2)
memory usage: 674.2 KB
```

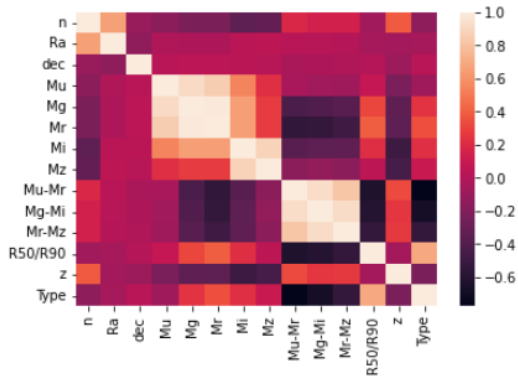
```
[In]: sns.distplot(df['z'])
```

```
[Out]:
```



```
[In]: sns.heatmap(df.corr())
```

```
[Out]:
```



Навчання моделі лінійної регресії

Давайте тепер почнемо тренувати модель лінійної регресії! Нам потрібно буде спочатку розділити наші дані на масив X, який містить дані для тренування, та масив y з цільовою змінною, в даному випадку стовпець червоне зміщення. Для визначення червоного зміщення нам важливі фотометричні параметри, тому ми будемо використовувати тільки ці дані, бо, наприклад, ID галактики в цьому випадку нам не потрібно.

```
[In]: X = df[['Ra', 'dec', 'Mu', 'Mg', 'Mr', 'Mi', 'Mz',
              'Mu-Mr', 'Mg-Mi', 'Mr-Mz', 'R50/R90' ]]
      y = df['z']
```

```
[In]: from sklearn.model_selection import train_test_split
```

```
[In]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=101)
```

```
[In]: from sklearn.linear_model import LinearRegression
```

```
[In]: lm = LinearRegression()
```

```
[In]: lm.fit(X_train,y_train)
```

```
[In]: print(lm.intercept_)
```

```
[In]: coeff_df = pd.DataFrame(lm.coef_,X.columns,columns=['Coefficient'])
      coeff_df
```

```
[Out]:
```

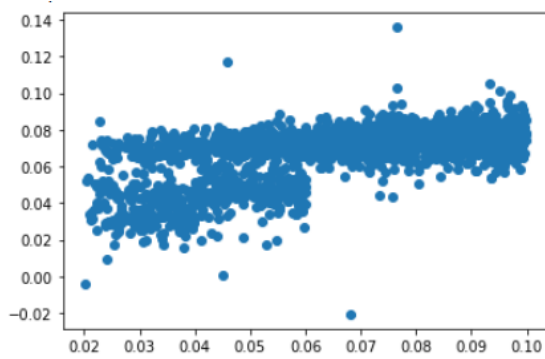
	Coefficient
Ra	-0.000033
dec	-0.000119
Mu	-0.099948
Mg	-0.054868
Mr	0.152820
Mi	0.002377
Mz	-0.003670
Mu-Mr	0.144174
Mg-Mi	-0.074028
Mr-Mz	0.051565
R50/R90	0.045522

Передбачення з нашої моделі

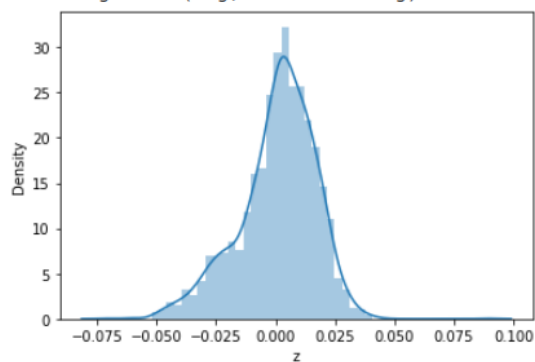
Давайте подивимось передбачення з нашої тестової вибірки і подивимось, наскільки добре вийшло передбачити!

```
[In]: predictions = lm.predict(X_test)
```

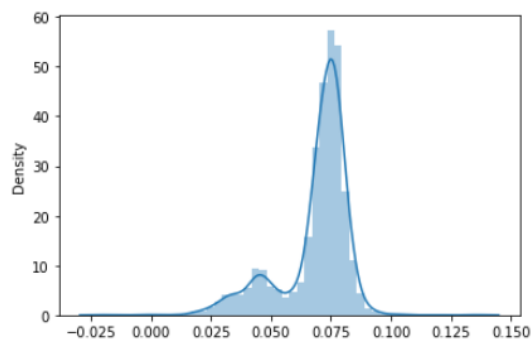
```
[In]: plt.scatter(y_test, predictions)
[Out]:
```



```
[In]: sns.distplot((y_test-predictions), bins=50)
[Out]:
```



```
[In]: sns.distplot((predictions), bins=50)
[Out]:
```



Regression Evaluation Metrics

Ось три загальні метрики оцінки для регресійних проблем:

Середня абсолютна похибка Mean Absolute Error (MAE) є середнім значенням абсолютної помилки:

$$\frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

Середньоквадратична похибка Mean Squared Error (MSE) є середнім значенням квадратичних помилок:

$$\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Корінь квадратний з середньоквадратичної похибки Root Mean Squared Error (RMSE) - це квадратний корінь із середнього значення квадратичних помилок:

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

Порівняння цих показників:

- MAE найлегше зрозуміти, оскільки це середня помилка.
- MSE є більш популярним, ніж MAE, тому що MSE “карає” більші помилки, що, як правило, корисно в реальному світі.
- RMSE навіть більш популярний, ніж MSE, оскільки RMSE можна інтерпретувати в одиницях “y”.

Все це функції втрат (loss functions), тому що ми хочемо їх мінімізувати.

```
[In]: from sklearn import metrics
```

```
[In]: print('MAE:', metrics.mean_absolute_error(y_test, predictions))  
      print('MSE:', metrics.mean_squared_error(y_test, predictions))  
      print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test, predictions)))
```

```
[Out]: MAE: 0.012562983775938855  
      MSE: 0.0002606046068894442  
      RMSE: 0.016143252673778113
```

```
[In]: sum(df['z'])
```

```
[Out]: 424.26112300000005
```

4. Logistic Regression

Логістична регресія, незважаючи на назву, представляє з себе модель для задачі класифікації, а не регресії. Логістичну модель дуже просто реалізувати і вона добре працює для лінійно розділених класів і є потужним алгоритмом для задач лінійної і бінарної класифікації. Її також можна розширити на багатокласову класифікацію. Введемо формулу відношення шансів, тобто шансів на користь окремо взятої події. В нашому випадку це належність галактики до певного типу. Данне відношення можна записати як: $p/(1-p)$, тут p - ймовірність того що подія відбудеться. Під подією ми розуміємо належність до певного класу. Введемо функцію логарифму відношення шансів (*logit*).

$$\text{logit}(p) = \log \frac{p}{1-p} \quad (1)$$

Ця функція приймає вхідні значення в діапазоні від 0 до 1 і трансформує їх по всьому діапазону дійсних чисел, які можна використовувати для визначення зв'язків між значенням ознак(параметрів) і логарифмом співвідношення вірогідностей:

$$\text{logit}(p(y=1|x)) = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^m w_ix_i = \mathbf{w}^T \mathbf{x} \quad (2)$$

$p(y=1|x)$ - це умовна вірогідність, що окремий зразок належить до класу 1 при наявності в нього ознак $\mathbf{x}$. В машинному навчанні нас насправді цікавить передбачення ймовірності, тобто зворотня функція, таку функцію називають логістичною(звідки і походить назва), а іноді просто сигмоїд через її схожість на латинську літеру S.

$$\phi(z) = \frac{1}{1 + e^{-z}} \quad (3)$$

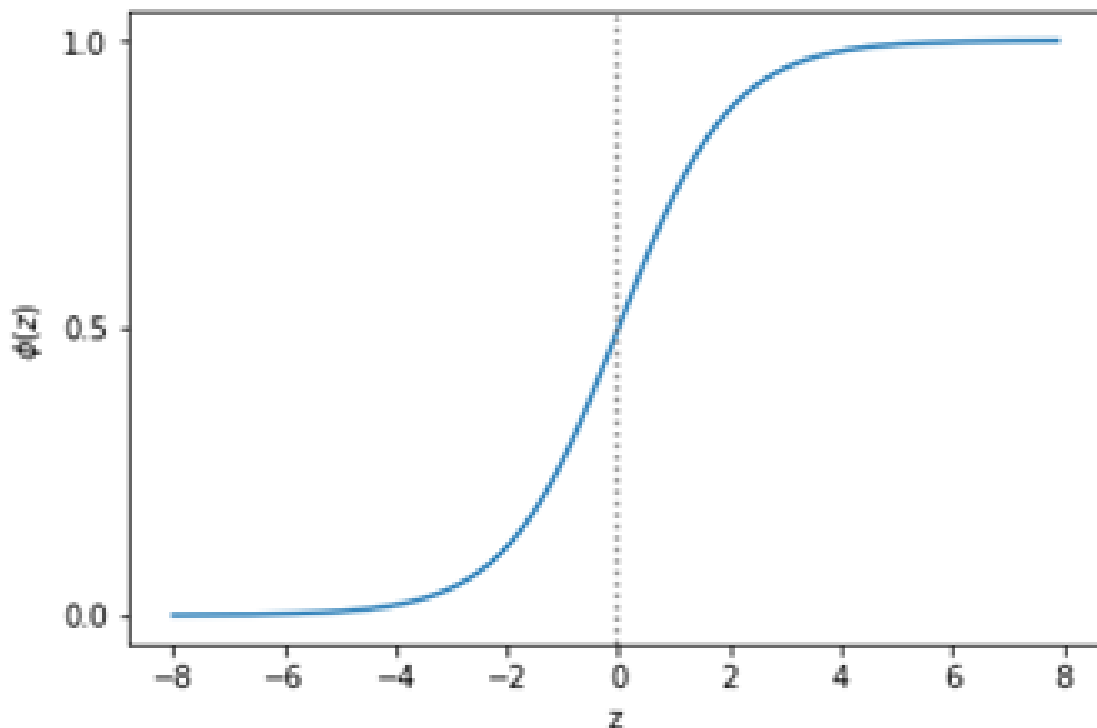


Рис. 1: Вигляд логістичної функції

Тут z – це вхідні данні подані у вигляді лінійної комбінації ознак і вагових коефіцієнтів для зразку, які можна означити як

$$z = w^T x = w_0 x_0 + w_1 x_1 + \dots + w_m x_m$$

Задача регресії полягає в підборі вагових коефіцієнтів $w(m)$ так, щоб вони якомога точно передбачали значення для тренувальної вибірки. Наша ж задача це підібрати такі вагові коефіцієнти щоб вони якомога точно передбачали значення на тестовій вибірці. Таким чином значення з сигмоїдальної функції буде визначати вірогідність приналежності об'єкта до визначеного класу при наявності в нього параметрів x на відкаліброваних регресією параметричних вагах w . Для бінарної класифікації за значенням вірогідності для об'єктів далі використовуються квантизатор у вигляді [Рашка]:

$$\hat{y} = \begin{cases} 1 & \text{якщо } \phi(z) > 0.5 \\ 0 & \text{в іншому випадку} \end{cases} \quad (4)$$

У всіх інших методах, які будуть давати нам значення ймовірності ми також будемо застосовувати цей квантизатор.

```
[In]: import pandas as pd
      from sklearn.model_selection import train_test_split

      X = df.drop(columns=['n', 'Ra', 'dec', 'Type'])
      y = df['Type']
      X = (X-X.min())/(X.max()-X.min())

      X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3, random_state=42)
[In]: from sklearn.linear_model import LogisticRegression
[In]: logmodel = LogisticRegression(class_weight='balanced',max_iter=1000)
      logmodel.fit(X_train,y_train)
[Out]: LogisticRegression(C=1.0, class_weight='balanced', dual=False,
      fit_intercept=True, intercept_scaling=1, l1_ratio=None,
      max_iter=1000, multi_class='auto', n_jobs=None, penalty='l2',
      random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
      warm_start=False)
[In]: predictions = logmodel.predict(X_test)
[In]: from sklearn.metrics import classification_report
[In]: print(classification_report(y_test,predictions))
[Out]:
```

	precision	recall	f1-score	support
0	0.95	0.96	0.95	1268
1	0.91	0.88	0.90	581
accuracy			0.94	1849
macro avg	0.93	0.92	0.93	1849
weighted avg	0.94	0.94	0.94	1849

```
[In]: param_grid = { 'C': [0.01,0.1,1,10,100,1000]}
      from sklearn.model_selection import GridSearchCV
      grid = GridSearchCV(LogisticRegression(class_weight='balanced',
      max_iter=1000),param_grid,refit=True,verbose=3)
      grid.fit(X_train,y_train)
[Out]:
```

```
[CV] C=1000 .....
[CV] ..... C=1000, score=0.940, total= 0.6s
[CV] C=1000 .....
[CV] ..... C=1000, score=0.948, total= 0.4s
[CV] C=1000 .....
[CV] ..... C=1000, score=0.963, total= 0.6s
[Parallel(n_jobs=1)]: Done 30 out of 30 | elapsed: 7.9s finished
GridSearchCV(cv=None, error_score=nan,
             estimator=LogisticRegression(C=1.0, class_weight='balanced',
                                           dual=False, fit_intercept=True,
                                           intercept_scaling=1, l1_ratio=None,
                                           max_iter=1000, multi_class='auto',
                                           n_jobs=None, penalty='l2',
                                           random_state=None, solver='lbfgs',
                                           tol=0.0001, verbose=0,
                                           warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=3)

[In]: grid.best_params_
[Out]: {'C': 10}
[In]: grid.best_estimator_
[Out]: LogisticRegression(C=10, class_weight='balanced', dual=False,
                          fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                          max_iter=1000, multi_class='auto', n_jobs=None, penalty='l2',
                          random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                          warm_start=False)
[In]: logmodel = LogisticRegression(**grid.best_params_,class_weight='balanced',max_iter=1000)
      logmodel.fit(X_train,y_train)
[Out]: LogisticRegression(C=10, class_weight='balanced', dual=False,
                          fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                          max_iter=1000, multi_class='auto', n_jobs=None, penalty='l2',
                          random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                          warm_start=False)
[In]: from sklearn.metrics import classification_report,confusion_matrix
      print(confusion_matrix(y_test,pred))
[Out]: [[1246  22] [ 86 495]]
[In]: print(classification_report(y_test,predictions))
[Out]:
```

	precision	recall	f1-score	support
0	0.95	0.96	0.95	1268
1	0.91	0.88	0.90	581
accuracy			0.94	1849
macro avg	0.93	0.92	0.93	1849
weighted avg	0.94	0.94	0.94	1849

5. k-nearest neighbor algorithm

Класифікатор на основі К-найближчих сусідів -це приклад найпростішого алгоритму машинного навчання. Він також є хорошим прикладом “лінивого” алгоритму через те що створює функції що розділяє класи, а запам’ятовує положення об’єктів тренувальної вибірки в гіперпросторі параметрів. Недоліком даного методу є те що його продуктивність лінійно залежить від розміру тренувальної вибірки. Для реалізації цього методу достатньо обрати число К-кількості сусідів, метрику відстані, знайти К найближчих сусідів в цій метриці та присвоїти об’єкту клас найбільшого числа з цих сусідів з тренувальної вибірки. Сусідам може присвоюватись статистична вага, наприклад $1/d$, де d - це відстань в просторі параметрів. Зважаючи на це, цей метор також чутливий до нормалізації, оскільки всі параметри мають давати однаковий вклад в оцінку відстані. Пошук числа К важливий, оскільки він дає змогу описати модель уникаючи перенавчання та недонавчання. [Рашка] Конкретний вигляд для визначення відстані залежить від метрики простору параметрів вибірки.

$$d_{i,j} = \sqrt[p]{\sum_k |x_k^i - x_k^j|^p} \quad (5)$$

Тут p – залежить від метрики (якщо $p=2$, то маємо Евклідов простір) До недоліків методу можна віднести ще залежність моделі від метрики і складність в підборі статистичних вігів. [Краснянський].

```
[In]: import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

[In]: from sklearn.model_selection import train_test_split

X = df.drop(columns=['n', 'Ra', 'dec', 'Type'])
y = df['Type']
X = (X-X.min())/(X.max()-X.min())

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3, random_state=42)
[In]: from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(X_train,y_train)
[Out]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
metric_params=None, n_jobs=None, n_neighbors=5, p=2,
weights='uniform')
[In]: pred = knn.predict(X_test)
[In]: from sklearn.metrics import classification_report,confusion_matrix
print(confusion_matrix(y_test,pred))
[Out]: [[1246  22] [ 86 495]]
[In]: print(classification_report(y_test,pred))
[Out]:
```

	precision	recall	f1-score	support
0	0.94	0.98	0.96	1268
1	0.96	0.85	0.90	581
accuracy			0.94	1849
macro avg	0.95	0.92	0.93	1849
weighted avg	0.94	0.94	0.94	1849

```
[In]: param_grid = { 'n_neighbors': [1,2,3,4,5,10,15]}
      from sklearn.model_selection import GridSearchCV
      grid = GridSearchCV(KNeighborsClassifier(),param_grid,refit=True,verbose=3)
      grid.fit(X_train,y_train)
```

```
[Out]:
```

```
[CV] n_neighbors=15 .....
[CV] ..... n_neighbors=15, score=0.924, total= 0.0s
[CV] n_neighbors=15 .....
[CV] ..... n_neighbors=15, score=0.928, total= 0.0s
[CV] n_neighbors=15 .....
[CV] ..... n_neighbors=15, score=0.926, total= 0.0s
[Parallel(n_jobs=1)]: Done 35 out of 35 | elapsed: 1.4s finished
GridSearchCV(cv=None, error_score=nan,
             estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
             metric='minkowski',
             metric_params=None, n_jobs=None,
             n_neighbors=5, p=2,
             weights='uniform'),
             iid='deprecated', n_jobs=None,
             param_grid={'n_neighbors': [1, 2, 3, 4, 5, 10, 15]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=3)
```

```
[In]: grid.best_params_
```

```
[Out]: {'n_neighbors': 5}
```

```
[In]: grid.best_estimator_
```

```
[Out]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                             metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                             weights='uniform')
```

```
[In]: knn = KNeighborsClassifier(**grid.best_params_)
```

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

```
[Out]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                             metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                             weights='uniform')
```

```
[In]: from sklearn.metrics import classification_report,confusion_matrix
      print(confusion_matrix(y_test,pred))
```

```
[Out]: [[1246  22] [ 86 495]]
```

```
[In]: print(classification_report(y_test,pred))
```

```
[Out]:
```

	precision	recall	f1-score	support
0	0.94	0.98	0.96	1268
1	0.96	0.85	0.90	581
accuracy			0.94	1849
macro avg	0.95	0.92	0.93	1849
weighted avg	0.94	0.94	0.94	1849

6. Decision Tree Classification

6.0..1 Decision Tree

Метод редева рішень часто використовують в задачах класифікації. В ньому ймовірність галактики належати тому чи іншому класу визначається через розподіл галактик за певними ознаками. Рішення приймаються на основі наявності у данного об'єкта тих чи інших ознак. У кожного рішення є свої наслідки які мають ймовірнісний характер. Дерево рішень будується в вигляді дерева-графів, яке складається з гілок, вузлів і листя. Гілки з'єднують між собою вузли в яких приймається рішення, листя – остаточний результат класифікації. Процес прийняття рішень послідовний і суть метода полягає в визначенню варіантів відсікання і визначенню які рішення варто прийняти в кожній вершині.

```
[In]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
"""імпортуємо з sklearn.model_selection функцію train_test_split
для розбиття вибірки на тренувальну і валідаційну"""

X = df.drop(columns=['n', 'Ra', 'dec', 'Type'])
"""з df викидаємо колонки 'n', 'Ra', 'dec', 'Type', оскільки вони
гарантовано не пов'язані з типом галактик і 'Type'"""
y = df['Type']
"""записуємо в змінну 'y' стовпчик 'Type' з df, як підбому змінну"""
X = (X-X.min())/(X.max()-X.min())
"""нормуємо дані щоб всі змінні вносили однаковий вклад в навчання"""

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)
"""розбиваємо нашу початкову вибірку на дві, тренувальну і валідаційну
з відповідним значенням цільових змінних в пропорції 2/5 попередньо
перемішуючи данні з параметром рандомізації random_state=42"""

from sklearn.tree import DecisionTreeClassifier
"""імпортуємо DecisionTreeClassifier з sklearn.tree"""

dtree = DecisionTreeClassifier(max_depth=3)
"""визначаємо наш класифікатор з параметром глибини рівному трьом"""

dtree.fit(X_train,y_train)
"""навчаємо нашу модель приймаючи X_train як набір параметрів,
а y_train, як цільове значення"""
[Out]: DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                             max_depth=3, max_features=None, max_leaf_nodes=None,
                             min_impurity_decrease=0.0, min_impurity_split=None,
                             min_samples_leaf=1, min_samples_split=2,
                             min_weight_fraction_leaf=0.0, presort='deprecated',
                             random_state=None, splitter='best')

[In]: predictions = dtree.predict(X_test)
"""передбачаємо значення типу для валідаційної(тестової) вибірки"""

from IPython.display import Image
from sklearn.externals.six import StringIO
from sklearn.tree import export_graphviz
import pydot
```

```

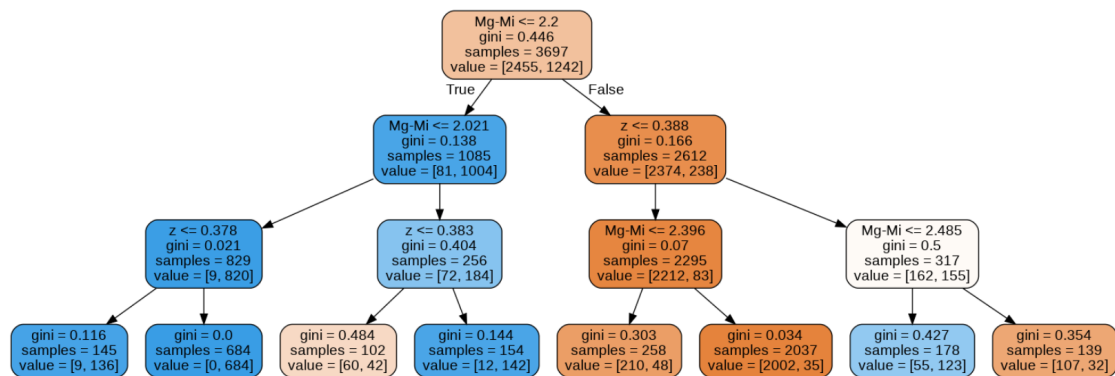
features = list(df.drop(columns='n', 'Ra', 'dec', 'Type'))
features
"""імпортуємо функції для візуалізації дерева-графа,
та записуємо наші зашоловки, як список стрічок"""

[Out]: ['Mu', 'Mg', 'Mr', 'Mi', 'Mz', 'Mu-Mr', 'Mg-Mi', 'Mr-Mz', 'R50/R90', 'z']
[In]: dot_data = StringIO()
      export_graphviz(dtree, out_file=dot_data, feature_names=features, filled=True, rounded=True)

      graph = pydot.graph_from_dot_data(dot_data.getvalue())
      Image(graph[0].create_png())

      """візуалізуємо нашу модель dtree"""
[Out]:

```



```

[In]: from sklearn.metrics import classification_report, confusion_matrix
      """імпортуємо функції для визначення точності нашої моделі"""
      print(classification_report(y_test, predictions))
      """виводимо на екран звіт перевірки точності передбачених параметрів,
      та реальних значень для валідаційної вибірки"""
[Out]:

```

	precision	recall	f1-score	support
0	0.92	0.96	0.94	1692
1	0.91	0.81	0.86	774
accuracy			0.92	2466
macro avg	0.91	0.89	0.90	2466
weighted avg	0.92	0.92	0.91	2466

```

[In]: print(confusion_matrix(y_test, predictions))# виводить на екран матрицю похибок
[Out]: [[1632  60] [ 148 626]]

```

6.0..2 Random Forests

Випадкові ліси – це сукупністю дерев, що приймають рішення, де кожне дерево дещо відрізняється від інших. Ідея, що стоїть за випадковим лісом, полягає в тому, що кожне дерево

може виконати порівняно хорошу роботу з прогнозування, але, швидше за все, деякі з дерев претренуються. Якщо ми побудуємо багато дерев та застосуємо усереднення їх результатів, це зменшить перетренування, зберігаючи передбачувальну силу дерев.

Алгоритм Випадковий ліс працює наступним чином. Маючи тренувальну вибірку, ми створюємо B випадкових вибірок S_b (для кожного $b = 1, \dots, B$) будуємо f_b модель дерева рішень, використовуючи кожну вибірку S_b як навчальну. Для вибірки S_b деякого b , робимо вибірку із заміною. Це означає, що ми починаємо з порожньої вибірки, а потім вибираємо приклад для навчання із навчального набору і ставимо його точну копію S_b , зберігаючи оригінальний приклад у вихідному навчальному наборі. Ми продовжуємо підбирати приклади навчання, поки $S_b = N$.

Після тренування у нас є дерева рішень. Прогноз для нового прикладу x отримується як середнє значення B прогнозів:

$$y \leftarrow \hat{f}(x) \frac{1}{B} \sum_{b=1}^B f_b(x) \quad (6)$$

y –морфологічний тип галактики, прийняття більшості голосів у разі класифікації.

Випадковий ліс використовує модифікований алгоритм навчання дерев, який перевіряє точність при кожному розбитті в процесі навчання. Це необхідно щоб уникнути співвідношення дерев: якщо одна або кілька особливостей є дуже впливовими для цільового значення, ці функції будуть вибрані для розділення прикладів на багатьох деревах. Це призведе до появи багатьох співвідносних дерев у нашому "лісі". Зв'язані прогнози не можуть допомогти підвищити точність прогнозування. Основною причиною кращої ефективності створення моделі є те, що "хороші" моделі, ймовірно, погоджуються між собою в прогнозі, тоді як "погані" моделі, ймовірно, не погоджуються між собою на однакових підвибірках.

Випадковий ліс – один з найбільш широко використовуваних алгоритмів навчання ансамблю. Ефективність полягає в тому, що, використовуючи кілька зразків вихідного набору даних, ми зменшуємо дисперсію остаточної моделі. Перенавчання трапляється тоді, коли наша модель намагається пояснити невеликі зміни в наборі даних. Оскільки наш набір даних є лише невеликою вибіркою для кожного дерева, а результат усереднюється, то перенавчання вдається уникнути. Якщо б нам не пощастило, як відібрали вибірку нашого навчального набору, то він міг би містити небажані (але неминучі) артефакти: шум, викиди, надмірно або недостатньо представлені приклади. Створюючи кілька випадкових зразків із заміною нашого навчального набору, ми таким чином зменшуємо дію цих артефактів.

З бібліотеки `sklearn` підключаємо `GridSearchCV` – для визначення оптимального значенню параметра тренування

```
[In]: from sklearn.ensemble import RandomForestClassifier
      """імпортуємо з sklearn.ensembles класифікатор RandomForestClassifier"""

      rfc = RandomForestClassifier()
      """визначаємо RandomForestClassifier як rfc з стандартними вхідними параметрами"""
      rfc.fit(X_train, y_train)
[Out]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                               criterion='gini', max_depth=None, max_features='auto',
                               max_leaf_nodes=None, max_samples=None,
                               min_impurity_decrease=0.0, min_impurity_split=None,
                               min_samples_leaf=1, min_samples_split=2,
                               min_weight_fraction_leaf=0.0, n_estimators=100,
                               n_jobs=None, oob_score=False, random_state=None,
                               verbose=0, warm_start=False)

[In]: rfc_pred = rfc.predict(X_test)
      print(classification_report(y_test, rfc_pred))
[Out]:
```

Стандартні параметри далеко не завжди є оптимальними, тому не зважаючи на задовільний результат ми можемо змінювати параметри моделі так, щоб максимізувати точність.

	precision	recall	f1-score	support
0	0.95	0.97	0.96	1692
1	0.93	0.88	0.90	774
accuracy			0.94	2466
macro avg	0.94	0.93	0.93	2466
weighted avg	0.94	0.94	0.94	2466

```
[In]: param_grid = {'max_depth': [3, 7, 11, 15], 'n_estimators': [1,10,100,500]}
      """створюємо словник з параметрами серед яких ми хочемо визначити найкращу комбінацію"""
      from sklearn.model_selection import GridSearchCV
      """імпортуємо GridSearchCV з sklearn.model_selection"""
      grid = GridSearchCV(RandomForestClassifier(),param_grid,refit=True,verbose=3)
      """визначаємо параметри GridSearchCV передаючи їй модель і словник параметрів"""
      grid.fit(X_train,y_train)
      """визначаємо найкращу комбінацію параметрів"""
```

```
[Out]:
```

```
[CV] max_depth=15, n_estimators=500 .....
[CV] ..... max_depth=15, n_estimators=500, score=0.954, total= 4.0s
[CV] max_depth=15, n_estimators=500 .....
[CV] ..... max_depth=15, n_estimators=500, score=0.950, total= 4.1s
[CV] max_depth=15, n_estimators=500 .....
[CV] ..... max_depth=15, n_estimators=500, score=0.951, total= 4.1s
[Parallel(n_jobs=1)]: Done 80 out of 80 | elapsed: 1.4min finished
GridSearchCV(cv=None, error_score=nan,
              estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
              class_weight=None,
              criterion='gini', max_depth=None,
              max_features='auto',
              max_leaf_nodes=None,
              max_samples=None,
              min_impurity_decrease=0.0,
              min_impurity_split=None,
              min_samples_leaf=1,
              min_samples_split=2,
              min_weight_fraction_leaf=0.0,
              n_estimators=100, n_jobs=None,
              oob_score=False,
              random_state=None, verbose=0,
              warm_start=False),
              iid='deprecated', n_jobs=None,
              param_grid={'max_depth': [3, 7, 11, 15],
              'n_estimators': [1, 10, 100, 500]},
              pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
              scoring=None, verbose=3)
```

```
[In]: grid.best_params_ # виводить на екран найкращі параметри
```

```
[Out]: {'max_depth': 7, 'n_estimators': 500}
```

```
[In]: grid.best_estimator_ # виводить на екран модель з найкращими параметрами
```

```
[Out]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
              criterion='gini', max_depth=7, max_features='auto',
              max_leaf_nodes=None, max_samples=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=500,
              n_jobs=None, oob_score=False, random_state=None,
              verbose=0, warm_start=False)
```

```
[In]: rfc = RandomForestClassifier(**grid.best_params_)
```

```
      """визначаємо модель з найкращими параметрами"""
```

```
rfc.fit(X_train, y_train)
[Out]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                             criterion='gini', max_depth=7, max_features='auto',
                             max_leaf_nodes=None, max_samples=None,
                             min_impurity_decrease=0.0, min_impurity_split=None,
                             min_samples_leaf=1, min_samples_split=2,
                             min_weight_fraction_leaf=0.0, n_estimators=500,
                             n_jobs=None, oob_score=False, random_state=None,
                             verbose=0, warm_start=False)
[In]: rfc_pred = rfc.predict(X_test)
      print(classification_report(y_test, rfc_pred))
[Out]:
```

	precision	recall	f1-score	support
0	0.95	0.98	0.96	1692
1	0.94	0.88	0.91	774
accuracy			0.94	2466
macro avg	0.94	0.93	0.93	2466
weighted avg	0.94	0.94	0.94	2466

7. Support Vector Machines

Як і метод логістичної регресії метод опорних векторів являє собою алгоритм лінійної класифікації. Такі моделі не модулюють сам клас, а шукають приму, криву чи площину що розділяє класи в багатовимірному просторі. Лінійний класифікатор намагається провести пряму що максимально відділяє один клас від іншого, але проблема полягає в тому що існує більш як одна така площина яка ідеально розділяє два класи. В методі опорних векторів рішенням пропонується проведення не лінії нульової ширини а лінії яка характеризується певною шириною (відступом) що визначається як відстань до найближчих представників всіх класів. В цьому методі обирається така лінія що максимізує цей відступ. [Вандер]

У машинному навчанні межа, що розділяє приклади різних класів, називається межею рішення.

Рівняння гіперплощини задається двома параметрами, реальним значенням w тієї ж розмірності, що і наш вхідний ознака x , і реальним числом b таким:

$$wx - b = 0, \quad (7)$$

де вираз wx означає $w_0x_0 + w_1x_1 + \dots + w_mx_m$, і m - кількість вимірів функціонального вектора x .

Тепер передбачувана мітка для деякого вхідного вектора x вводиться так:

$$y = \text{sign}(wx - b), \quad (8)$$

Для опису моделі вводиться поняття позитивної і негативної гіперплощини що є паралельні до площини рішень і на яких лежать найближчі об'єкти до цієї межі:

$$w^T x_{pos} + b = 1, w^T x_{neg} + b = -1 \quad (9)$$

Віднімемо ці два вирази між собою і нормуємо на довжину вектора коефіцієнтів кутових нахилів

$$\|w\| = \sqrt{\sum_{i=1}^m w_i^2} \quad (10)$$

Де m - це розмірність простору параметрів. Ми отримуємо

$$\frac{w^T(x_{pos} - x_{neg})}{\|w\|} = \frac{2}{\|w\|} \quad (11)$$

Ліву частину тепер можна інтерпретувати як відстань між двома площинами. Тепер метод зводиться пошуку площини для якої $\frac{2}{\|w\|}$ буде максимальним з обмеженнями на максимізацію правильно класифікованих об'єктів по різні сторони від цієї площини.

Щоб модель правильно спрогнозувала мітки наших прикладів кожен приклад задається парою (x_i, y_i) , де x_i - вектор властивостей прикладу i , y_i - його мітка, яка приймає значення або -1 , або $+1$. Отже, обмеження природно:

$$wx_i - b \geq +1 \text{ if } y_i = +1, wx_i - b \leq -1 \text{ if } y_i = -1., \quad (12)$$

Після визначення параметрів w і b ми можемо класифікувати об'єкти просто перевіряючи знак функції y . [Рашка] Враховуючи що вибір даної прямої на пряму залежить від норми вектора w метод чуттєвий до нормування вхідних даних щоб кожен з параметрів вносив однаковий вклад в визначення відстані.

Недоліком моделі є те що її параметри важко інтерпретувати [Краснянский]. Logistic Regression і SVC за замовчуванням ці дві моделі використовують L2 регуляцію. Для цих методів параметр C визначає степінь регуляції, пропорційне оберненому середньому відхиленню ймовірності передбачення моделі від реального класу. Коли значення C високе Logistic Regression і SVC намагаються підігнати моделі до навчальним даних якомога точніше, тоді як при низьких C моделі намагаються шукати вектор коефіцієнтів близький до нуля. Тому при низьких значеннях алгоритм намагається відповісти більшості точок даних, тоді як при малих значеннях C збільшує вклад кожної окремої точки. [Мюллер]

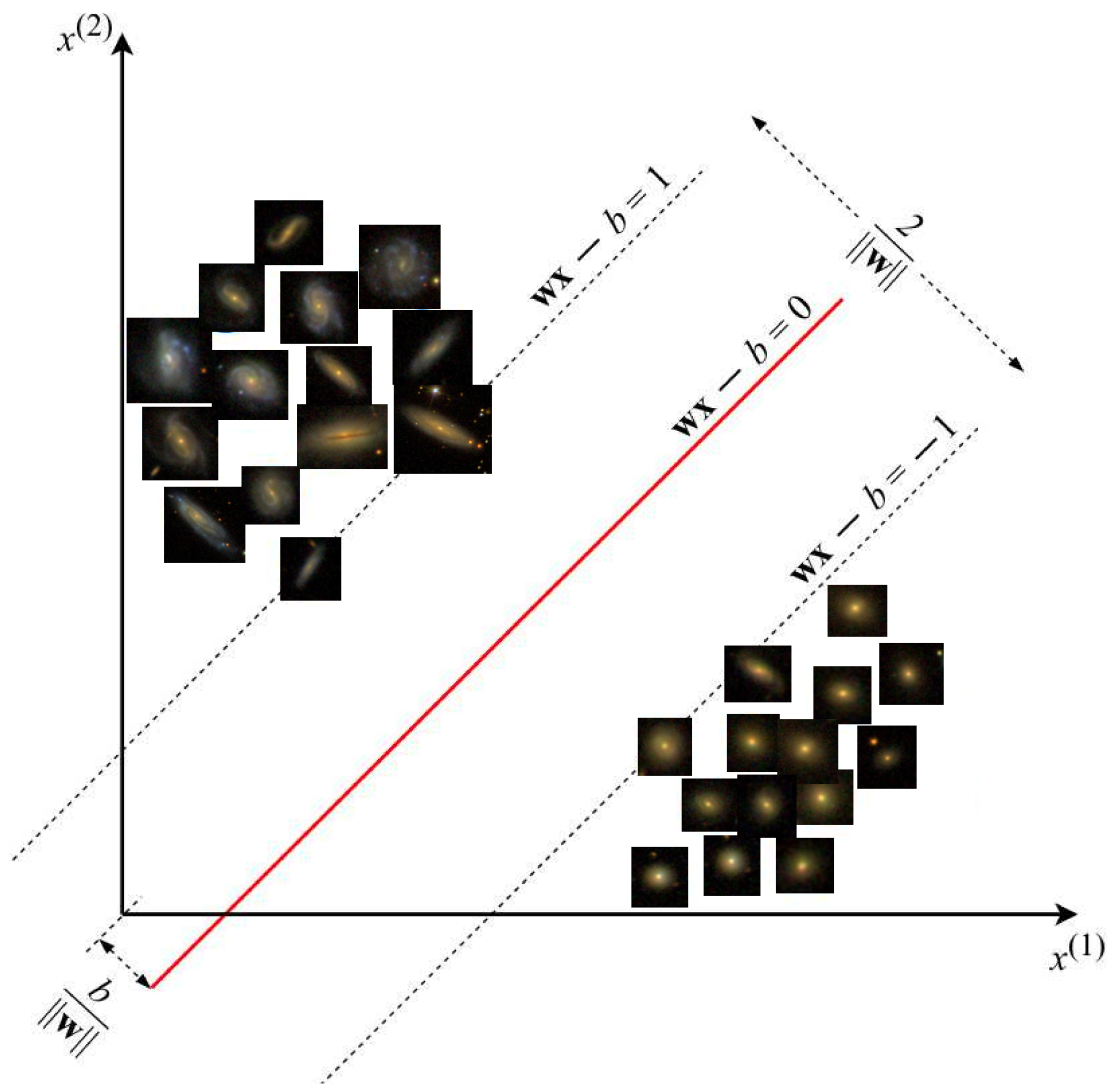


Рис. 2: Приклад моделі SVM для двовимірних функціональних векторів.

```
[In]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
[In]: from sklearn.model_selection import train_test_split

X = df.drop(columns=['n', 'Ra', 'dec', 'Type'])
y = df['Type']
X = (X - X.min()) / (X.max() - X.min())

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
[In]: from sklearn.svm import SVC
model = SVC()
model.fit(X_train, y_train)
[Out]: SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
max_iter=-1, probability=False, random_state=None, shrinking=True,
tol=0.001, verbose=False)
```

```
[In]: from sklearn.metrics import classification_report, confusion_matrix
      predictions = model.predict(X_test)
[In]: print(confusion_matrix(y_test, predictions))
[Out]: [[1267   1] [ 238 343]]
[In]: print(classification_report(y_test, predictions))
[Out]:
```

	precision	recall	f1-score	support
0	0.84	1.00	0.91	1268
1	1.00	0.59	0.74	581
accuracy			0.87	1849
macro avg	0.92	0.79	0.83	1849
weighted avg	0.89	0.87	0.86	1849

```
[In]: param_grid = {'C': [1, 10, 100, 1000], 'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
                    'kernel': ['rbf']}
      from sklearn.model_selection import GridSearchCV
      grid = GridSearchCV(SVC(class_weight='balanced'), param_grid, refit=True, verbose=3)
      grid.fit(X_train, y_train)
[Out]:
```

```
[CV] C=1000, gamma=0.0001, kernel=rbf .....
[CV] ... C=1000, gamma=0.0001, kernel=rbf, score=0.941, total= 0.2s
[CV] C=1000, gamma=0.0001, kernel=rbf .....
[CV] ... C=1000, gamma=0.0001, kernel=rbf, score=0.939, total= 0.2s
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed: 22.3s finished
GridSearchCV(cv=None, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [1, 10, 100, 1000],
                          'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
                          'kernel': ['rbf']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=3)
```

```
[In]: grid.best_params_
[Out]: {'C': 100, 'gamma': 0.1, 'kernel': 'rbf'}
[In]: grid.best_estimator_
[Out]: SVC(C=100, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
          decision_function_shape='ovr', degree=3, gamma=0.1, kernel='rbf',
          max_iter=-1, probability=False, random_state=None, shrinking=True,
          tol=0.001, verbose=False)
[In]: model = SVC(**grid.best_params_, class_weight='balanced', probability=True)
      model.fit(X_train, y_train)
[Out]: SVC(C=100, break_ties=False, cache_size=200, class_weight='balanced', coef0=0.0,
          decision_function_shape='ovr', degree=3, gamma=0.1, kernel='rbf',
          max_iter=-1, probability=True, random_state=None, shrinking=True, tol=0.001,
          verbose=False)
[In]: predictions = model.predict(X_test)
[In]: print(confusion_matrix(y_test, predictions))
[Out]: [[1212  56] [ 43 538]]
```

```
[In]: print(classification_report(y_test, predictions))
```

```
[Out]:
```

	precision	recall	f1-score	support
0	0.97	0.96	0.96	1268
1	0.91	0.93	0.92	581
accuracy			0.95	1849
macro avg	0.94	0.94	0.94	1849
weighted avg	0.95	0.95	0.95	1849

Література

- [1] D. J. Eisenstein, D. H. Weinberg та E. e. a. Agol, ?SDSS-III: Massive Spectroscopic Surveys of the Distant Universe, the Milky Way, and Extra-Solar Planetary Systems,? *aj*, т. 142, № 3, 72, с. 72, вер. 2011. DOI: [10.1088/0004-6256/142/3/72](https://doi.org/10.1088/0004-6256/142/3/72). arXiv: [1101.1529](https://arxiv.org/abs/1101.1529) [[astro-ph.IM](#)].
- [2] M. R. Blanton, M. A. Bershadsky та B. e. a. Abolfathi, ?Sloan Digital Sky Survey IV: Mapping the Milky Way, Nearby Galaxies, and the Distant Universe,? *aj*, т. 154, 28, с. 28, лип. 2017. DOI: [10.3847/1538-3881/aa7567](https://doi.org/10.3847/1538-3881/aa7567). arXiv: [1703.00052](https://arxiv.org/abs/1703.00052).
- [3] Dark Energy Survey Collaboration, T. Abbott, F. B. Abdalla, J. Aleksić, S. Allam та A. e. a. Amara, ?The Dark Energy Survey: more than dark energy - an overview,? *mnras*, т. 460, с. 1270—1299, серп. 2016. DOI: [10.1093/mnras/stw641](https://doi.org/10.1093/mnras/stw641). arXiv: [1601.00329](https://arxiv.org/abs/1601.00329).