Challenges and Issues of Modern Science, 3 (2024). ISSN: 3083-5704
URL: https://cims.fti.dp.ua/j/article/view/215
PURL: https://purl.org/cims/2403.017

Векторизація обчислень для оптимізації коду на мові програмування Python

Олексій Земляний, Олег Байбуз

Purpose. The purpose of this study is to explore vectorization as an engineering technique to improve the performance and readability of Python code, particularly in data processing tasks. We aim to demonstrate the benefits of vectorization through practical examples involving the handling of missing data. **Design / Method / Approach.** To achieve the research goals, we performed a comparative analysis between loop-based and vectorized implementations. In particular, two versions of the function were developed to detect columns containing a specified number of missing values in the data set. These implementations were tested on two real-world datasets. We compared execution time and code readability. **Findings.** The findings showed that vectorization resulted in substantial performance improvements, reducing execution time by hundreds of times compared to traditional loop-based methods. Additionally, the vectorized code was more compact, leading to greater readability and ease of maintenance. **Theoretical Implications.** Vectorization provides a higher level of abstraction for performing operations on data structures. This allows developers to focus on algorithmic logic rather than managing iterative control structures, contributing to broader discussions on optimizing computational efficiency in Python. **Practical Implications**. For data engineers and analysts, vectorization represents a highly effective solution for optimizing Python code. It significantly accelerates data-intensive tasks, such as missing data imputation, data analysis, and machine learning, making it an essential tool for enhancing productivity in data-driven environments. **Originality / Value**. This study presents a practical approach to optimizing Python code through vectorization. It is valuable for professionals seeking to improve efficiency in their workflows. **Research Limitations / Future Research**. The limitation of this study is that it focuses on a single approach - vectorization. Future research should investigate the use of vectorization in conjunction with Just-In-Time (JIT) compilation using tools such as Numba to further improve Python performance. **Paper Type.** Practitioner Paper.

Keywords:

vectorization, Python optimization, data processing, performance improvement, code readability, software development, missing data handling

Contributor Details:

Oleksii Zemlianyi, PhD student, Researcher, Oles Honchar Dnipro National University: Dnipro, UA, zemlyanoy.aleksey@gmail.com

Oleh Baibuz, Prof., Dr.Sc., Oles Honchar Dnipro National University: Dnipro, UA, olegbaybuz68@gmail.com

Python є однією з найпопулярніших мов програмування, особливо у сферах аналізу даних, машинного навчання та наукових обчислень. Проте, незважаючи на свою зручність і універсальність, Python не завжди демонструє високу продуктивність, особливо коли йдеться про великі обсяги обчислень. Одним із найпотужніших методів підвищення швидкодії Python-коду є векторизація – техніка, що дозволяє виконувати операції над цілими масивами даних без використання циклів.

Мета і завдання

Метою цієї роботи є дослідити застосування векторизації для підвищення продуктивності та читабельності власного Python-коду та продемонструвати її переваги на прикладах.

Матеріали і методи

Векторизація – це інженерна техніка, за допомогою якої можна здійснювати обчислення над масивами або векторами даних за допомогою спеціалізованих функцій, що виконують операції над цілими структурами даних одночасно без використання циклів *for* (Turner-Trauring, 2023). В Python векторизацію виконують за допомогою бібліотек, написаних на C або Rust, наприклад, NumPy, Pandas чи Numba, які надають оптимізовані функції для роботи з масивами даних.

Переваги векторизації наступні:

1. Продуктивність. Операції на масивах обробляються на нижчому рівні (часто в машинному коді), що робить їх швидшими.

2. Стислість коду. Векторизовані операції дозволяють уникати вкладених циклів та роблять код чистішим.

3. Оптимізація пам'яті. Векторизовані операції зменшують кількість звернень до пам'яті.

Використання готових векторизованих рішень з бібліотек

Спочатку розглянемо готові векторизовані рішення, що надаються бібліотеками такими, як NumPy чи Pandas. Ці бібліотеки надають операції та інженерну техніку застосування загальних операцій для всього контейнеру даних одночасно. Порівняємо використання циклів та векторизацію з точки зору прискорення часу обчислень та підвищення компактності та читабельності коду на прикладах власного Python-коду при розробці програмного забезпечення імпутування пропусків у даних (Zemlianyi & Baibuz, 2024).

Розглянемо дві реалізації функції, що виконують різними способами одну і ту саму дію: знаходять в наборі даних імена стовпців, які містять задану кількість пропусків у даних. В лістингу 1а наведено реалізацію традиційним підходом із застосуванням циклів. В лістингу 1б ця функція переписана із застосуванням векторизації обчислень.

Лістинг 1а – Реалізація з використанням циклів (Джерело: Автори)

```
def get_col_names_with_count_of_nan_is_n_loop(data, n=1):
    df = data.copy()
    # Додаємо стовпець 'count', де кожен елемент –
    # кількість пропусків (NaN) у рядку
    counts = []
    for i in range(len(df)):
        count_nan = 0
        for col in df.columns:
            if pd.isnull(df.iloc[i][col]):
                count_nan += 1
        counts.append(count_nan)
    df['count'] = counts
    # Додаємо стовпець 'nans', де кожен елемент –
    # рядок із назвами стовпців, що містять NaN у цьому рядку
    nans = []
    for i in range(len(df)):
        nan_columns = []
        for col in df.columns:
            if pd.isnull(df.iloc[i][col]):
                nan_columns.append(col)
        nans.append(",".join(nan_columns))
    df['nans'] = nans
    # Створюємо список унікальних назв стовпців,
    # які мають рівно n пропусків
    d = []
    for i in range(len(df)):
        if df.iloc[i]['count'] == n:
            nan_columns = df.iloc[i]['nans']
            if nan_columns:
                d.extend(nan_columns.split(","))
    d = np.unique(d)
    return d
```

Лістинг 1б – Реалізація з використанням векторизації (Джерело: Автори)

```
def get_col_names_with_count_of_nan_is_n_vect(data, n=1):
    df = data.copy()
    # Додаємо стовпець 'count', де кожен елемент –
    # кількість пропусків (NaN) у рядку
    df['count'] = df.isnull().sum(axis=1)
    # Додаємо стовпець 'nans', де кожен елемент –
    # рядок із назвами стовпців, що містять NaN у цьому рядку
    df['nans'] = (df.isnull() @ (df.columns + ",")).str[:-1]
    # Створюємо список унікальних назв стовпців,
    # які мають рівно n пропусків
    d = df[df['count'] == n]['nans'].tolist()
    d = np.unique(d)
    return d
```

Результати використання векторизації

Порівняємо лістинги 1а та 1б візуально. Бачимо, що код лістингу 1б значно компактніше, що робить його більш читабельним. Крім того, виконаємо порівняння швидкодії цих методів. Для цього візьмемо набори даних, які ми

використовували у своїх попередніх дослідженнях, UCI Heart Disease Data, UCI HDD (Janosi et al., 1988) та Framingham Heart Study, Framingham FHS (NHLBI, 2024). Для усунення зсуву у результаті експеримент повторено 5 разів.

В таблиці 1 наведено розміри наборів даних та середній час виконання методів.

Таблиця 1 – Порівняння швидкодії двох реалізацій методів (Джерело: Автори)

Набір даних	Кількість рядків	Кількість стовпців	Час виконання	
			Метод з використанням циклів	Метод з векторизацією обчислень
UCI HDD	920	16	2,039572	0,003774
FRAMINGHAM FHS	4240	18	10,327844	0,005967

Швидкість виконання для набору даних UCI HDD збільшилась у 540 разів з використанням векторизації обчислень, а для набору FRAMINGHAM FHS – у 1730 разів.

Має сенс також розглянути у лістингу 1б наступний рядок

df['nans'] = (df.isnull() @ (df.columns + ",")).str[:-1]

Тут використовується матрична операція @ з Pandas для створення рядків з назвами стовпців, що містять значення NaN у кожному рядку DataFrame. Розберемо, що робить цей код крок за кроком.

1. df.isnull():

Ця функція повертає булевий DataFrame того ж розміру, що й df, де кожен елемент буде True, якщо відповідний елемент у df є NaN, і False — якщо ні.

df	A	B	C
0	1	3	NaN
1	NaN	4	NaN
2	2	NaN	5

df.isnull()	A	B	C
0	False	False	True
1	True	False	True
2	False	True	False

Рисунок 1 – Вихідний DataFrame df (ліворуч) та df.isnull() (праворуч) (Джерело: Автори)

2. df.columns + ",":

Створюється масив з назвами всіх стовпців, до яких додається кома. В нашому прикладі повернеться

Index(['A,', 'B,', 'C,'], dtype='object')

3. df.isnull() @ (df.columns + ","):

Оператор @ – це оператор матричного множення з Pandas, який в нашому випадку використовується для створення рядка з назвами стовпців, де значення є NaN в рядку DataFrame. По суті це буде матричне множення

булевого DataFrame на рядок з назвами стовпців. Кожен True у рядку DataFrame помножується на відповідну назву стовпця (з комою), і результат з'єднується в один рядок для кожного рядка DataFrame. В нашому прикладі результат буде наступний.

```
0      C,
1   A, C,
2      B,
```

Рисунок 2 – Результат операції df.isnull() @ (df.columns + ",") (Джерело: Автори)

Це означає, що в першому рядку тільки стовпець C містить значення NaN, у другому — стовпці A і C, а в третьому — стовпець B.

4. .str[:-1]

Ця операція видалить останню кому.

Таким чином, використовуючи швидкі векторизовані операції з Pandas, цей код створює новий стовпець, де для кожного рядка зберігається рядок із назвами всіх стовпців, що містять NaN у цьому рядку, розділені комами.

Векторизація власних функцій

Існує також техніка перетворення власних функцій на векторизовані за допомогою *np.vectorize*. Це інструмент з бібліотеки NumPy, який дозволяє застосувати функцію до кожного елемента масиву. Це часто виглядає як векторизація, проте слід розуміти, що *np.vectorize* є, по суті, лише обгорткою навколо циклу *for*, що дозволяє викликати функцію для кожного елемента масиву, не використовуючи явний цикл. Ми не отримаємо приросту швидкодії, але покращимо компактність та читабельність коду.

Фрагмент власного коду з використанням *np.vectorize* наведено у лістингу 2.

Лістинг 2 – Фрагмент коду з np.vectorize (Джерело: Автори)

```
Y_test = algreg.predict(X_test)
# correct answers
d = data[i].value_counts().to_dict()
if measure == 'value':
      Y_test = np.vectorize(
                          lambda x: correct_value(d, x))(Y_test)
else:
      Y_test = np.vectorize(
                         lambda x: correct_value_weight(d,
x))(Y_test)
df.loc[df[i].isna(), i] = Y_test
```

Ця техніка використовується для того, щоб позбутись циклів для застосування корегування значення одразу для всіх елементів стовпця DataFrame.

Таким чином, можна зробити наступні висновки щодо *np.vectorize.*

1. Зручність: np.vectorize робить код простішим, оскільки не потрібно писати явні цикли.

2. Не підвищує швидкодію: попри те, що функція називається «векторизованою», насправді вона не виконує обчислення в паралелі або в оптимізованій формі, як це робить векторизація за допомогою NumPy. np.vectorize – це, по суті, обгортка над циклом, яка не забезпечує значного прискорення обчислень.

3. Застосування: Використовується в ситуаціях, коли потрібно легко застосувати довільну функцію до масиву, але не для оптимізації швидкодії.

Висновки

Ми дослідили використання векторизації обчислень з двох точок зору: для підвищення швидкодії та для покращення компактності та читабельності коду на прикладах власного коду, що написано для рішення задачі імпутування пропусків у даних. Виявлено значні переваги цієї інженерної техніки як для підвищення швидкодії, так і для покращення читабельності коду.

Таким чином, векторизація є потужним інженерним рішенням для оптимізації коду на Python, особливо при роботі з великими обсягами даних та складними обчисленнями. Вона дозволяє значно зменшити час виконання програм і робить код більш чистим і зрозумілим. Використання бібліотек, таких як NumPy та Pandas, до-зволяє ефективно реалізовувати векторизовані операції в Python, що є критично важливим у сучасній науці про дані та інженерних задачах.

Посилання

Turner-Trauring, I. (2023, January). *How vectorization speeds up your Python code*. Hyphenated Enterprises LLC. https://pythonspeed.com/articles/vectorization-python/

Zemlianyi, O., & Baibuz, O. (2024). Методи імпутування пропусків у даних про ішемічну хворобу серця. *System Technologies, 2*(151), 33–49. https://doi.org/10.34185/1562-9945-2-151-2024-04

Janosi, A., Steinbrunn, W., Pfisterer, M., & Detrano, R. (1988). *Heart Disease*. UCI Machine Learning Repository. https://doi.org/10.24432/C52P4X

NHLBI. (2024). *Framingham Heart Study-Cohort (FHS-Cohort)*. National Heart, Lung, and Blood Institute. https://biolincc.nhlbi.nih.gov/studies/framcohort/