Отчет по заданию №1 Практикума на ЭВМ

Никишин Евгений, 317 группа

09.10.2015

Содержание

1 Введение

Данное задание направлено на освоение языка Python и системы научных вычислений NumPy. Задание состоит из 8 задач, при этом для каждой из них нужно написать векторизованный, невекторизованный варианты, а также третий вариант на усмотрение студента. Затем предлагается исследовать скорости работы написанных алгоритмов и, для некоторых задач, сравнить с готовыми реализациями в библиотеке SciPy

2 Задачи

2.1 Задача 1

Формулировка

Подсчитать произведение ненулевых элементов элементов на диагонали произвольной матрицы.

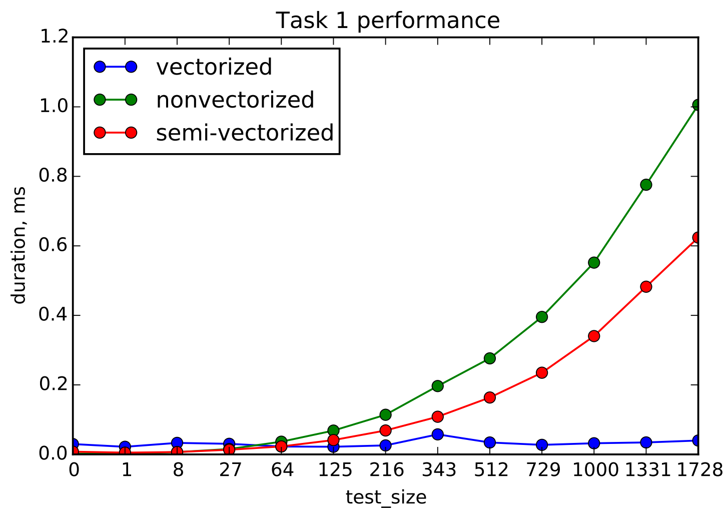
Описание решений

1. Векторизованный

```
def diag_nonzero_prod1(X):
    diag = np.diag(X)
    return np.prod(diag[diag != 0])
```

2. В невекторизованном варианте заводится словарь, в который добавляются ненулевые диагональные элементы, а затем считается их произведение. При этом цикл проходит от нуля до минимума из количества столбцов и строк.
3. В третьем варианте заводится переменная res = 1, и умножается циклом на каждый ненулевой элемент np.diag(X).

Сравнение результатов



Выводы

Как видно, в целом, векторизованный вариант работает лучше двух других. Однако при небольших размерах данных имеет смысл использовать несложные алгоритмы. Видно, что при маленьких размерах обыкновенное умножение работает быстрее, чем np.prod.

2.2 Задача 2

Формулировка

Даны матрица X и векторы одинаковой длины i и j. Построить вектор np.array(X[i[0], j[0]], . . . , X[i[N-1], j[N-1]]).

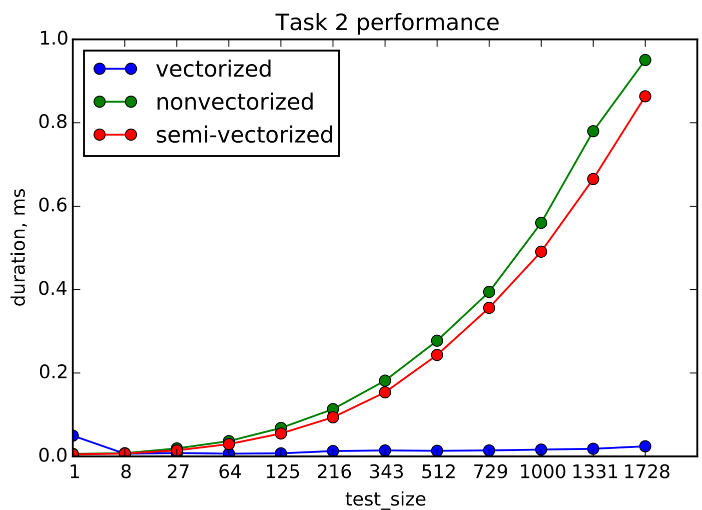
Описание решений

1. Векторизованный

```
def new_vector1(X, i, j):
    return X[i, j]
```

2. В невекторизованном варианте заводится словарь, в который циклом добавляются элементы X[i[k], j[k]].
3. Заводится массив размера i.shape[0], в который циклом добавляются необходимые элементы.

Сравнение результатов



Выводы

Снова векторизованный вариант справляется быстрее с задачей почти на всех, даже небольших размерах выборки.

2.3 Задача 3

Формулировка

Необходимо проверить, задают ли два вектора одинаковые мультимножества.

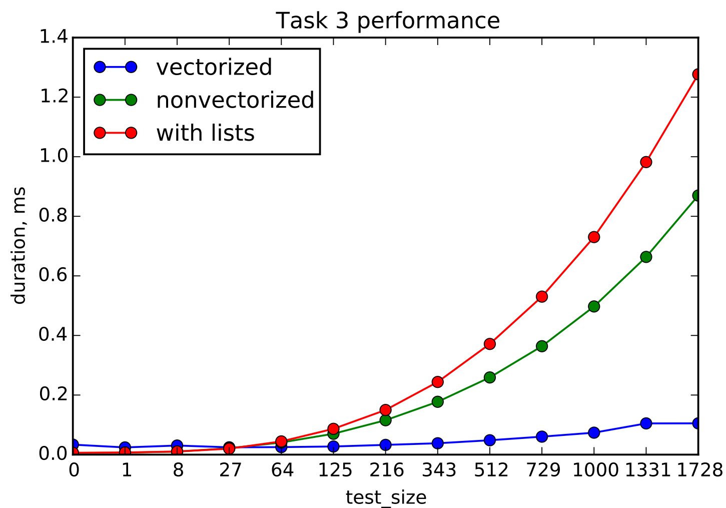
Описание решений

1. Векторизованный

```
def check_multisets1(x, y):
    return np.array_equal(np.sort(x), np.sort(y))
```

2. В невекторизованном варианте заводятся 2 словаря, содержащие все элементы векторов и их количество, а затем проверяется их равенство.
3. В третьем варианте векторы конвертируются в списки, которые, в свою очередь, сортируются и сравниваются.

Сравнение результатов



Выводы

В данной задаче (как, впрочем, и во многих других) векторизованный вариант — самый простой и естественный, а другие — лишь ненужное усложнение, поэтому и имеем такие результаты.

2.4 Задача 4

Формулировка

В векторе найти максимальный элемент, следующий за нулевым.

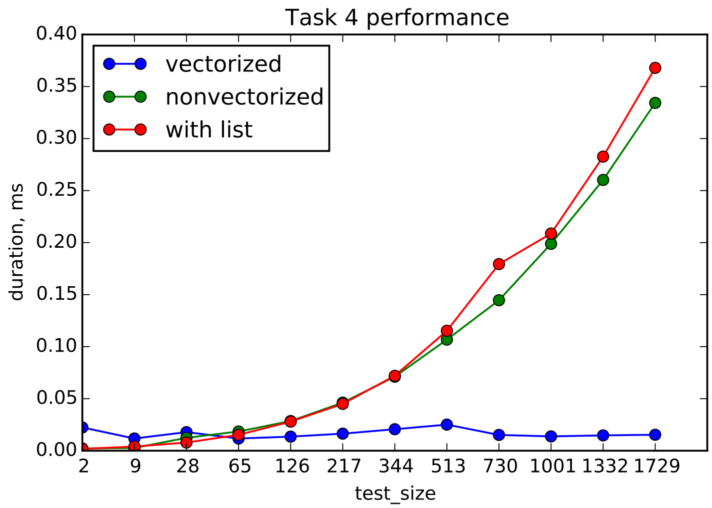
Описание решений

1. Векторизованный

```
def max_after_zero1(x):
    return np.max(x[np.where(x[:-1:] == 0)[0] + 1])
```

Получаем набор индексов нулей, прибавляем 1, ищем максимум элементов с такими индексами. В данном алгоритме сразу видно недостаток: при пустом x будет выдана ошибка. Но так как оговорено, что все объекты непустые (а также для того, чтобы код был в одну строку), такая проверка отсутствует.
2. Невекторизованный вариант: при нахождении нулевого элемента сравниваем с максимумом последующий элемент.
3. Заводится список всех элементов, следующих за нулем, и находится максимум.

Сравнение результатов



Выводы

Как видно, векторизованный вариант не слишком очевидный и не слишком простой, поэтому имеет смысл использовать простые модели на векторах длины меньше 50.

2.5 Задача 5

Формулировка

Дан трёхмерный массив, содержащий изображение, размера (height, width, numChannels), а также вектор длины numChannels. Сложить каналы изображения с указанными весами, и вернуть результат в виде матрицы размера (height, width).

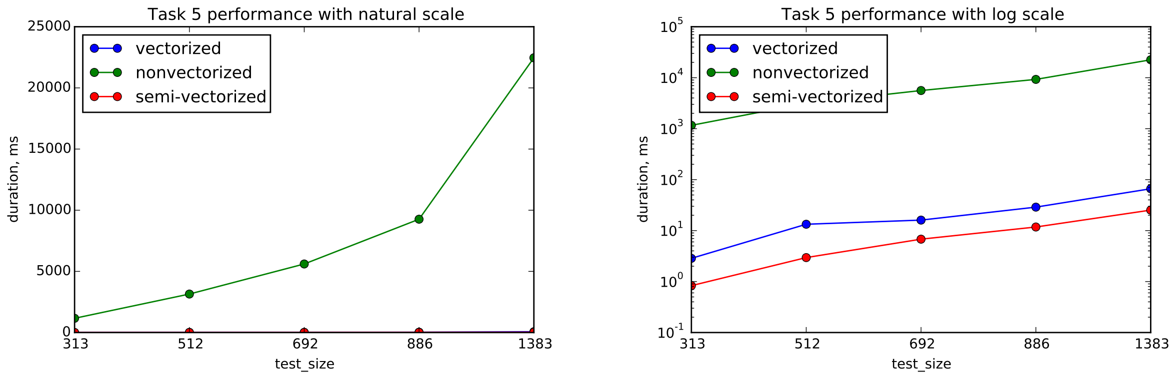
Описание решений

1. Векторизованный

```
def weighted_array_sum1(X, weights):
    return np.average(X, axis=2, weights=weights)
```

2. Невекторизованный — тройной цикл по всему изображению и всем каналам.
3. Полувекторизованный — цикл только по каналам.

Сравнение результатов



а) б)

Рис. 1: Зависимость времени работы алгоритмов в натуральной (линейной) и логарифмической шкале.

Выводы

Полностью невекторизованный вариант работает чудовищно долго. В линейной шкале графики полу- и векторизованного варианта почти что совпадают с осью x. Однако заметно, что для 3-х каналов вариант алгоритма с циклом по всего лишь трем каналам работает хоть и не намного, но быстрее полностью векторизованного варианта.

2.6 Задача 6

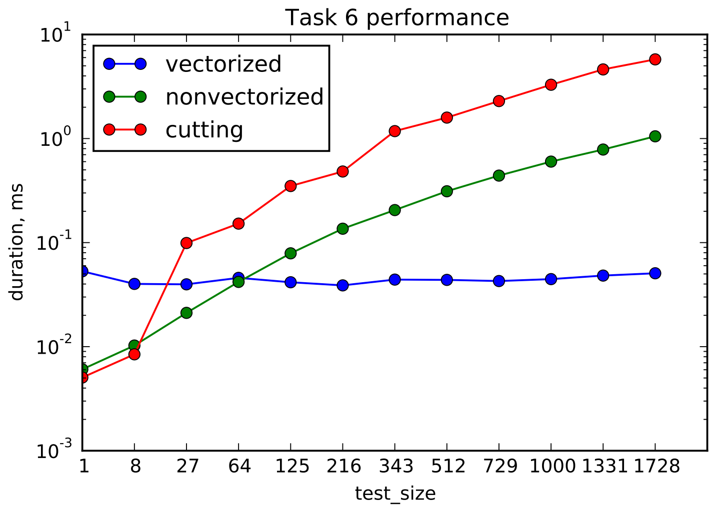
Формулировка

Необходимо реализовать Run-length encoding для произвольного вектора.

Описание решений

1. Векторизованный вариант: находим индексы, в которых соседние элементы отличаются, вектор элементов с такими индексами плюс последний — то, что ищем. Количество же определяется разницами между полученными индексами.
2. В невекторизованном варианте пробегаем по вектору, пока соседние элементы равны, увеличиваем счетчик на единицу, как только стали не равны, в список значений добавляем предыдущий элемент, а в количественный список текущее значение счетчика. Счетчик устанавливаем равным 1.
3. В третьем варианте "урезаются"последовательно равные элементы с добавлением в список их количества.

Сравнение результатов



Выводы

Все алгоритмы достаточно сложные, но, как всегда, асимптотически побеждает векторизованный вариант.

2.7 Задача 7

Формулировка

Даны две выборки объектов - X и Y. Вычислить матрицу евклидовых расстояний между объектами. Сравнить с SciPy реализацией.

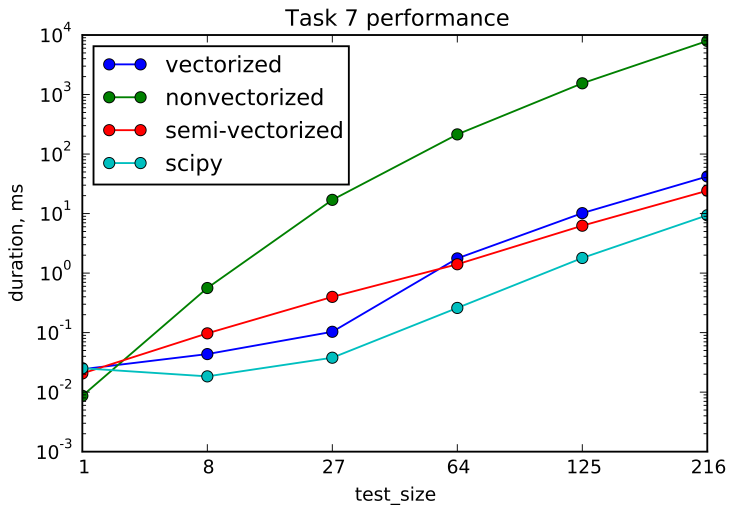
Описание решений

1. Векторизованный (с использованием broadcasting)

```
def object_dist1(X, Y):
    return np.sqrt(np.sum((X[:, :, np.newaxis] -
    Y.T[np.newaxis, :, :]) ** 2, axis=1))
```

2. Тройной цикл
3. Третий вариант — полувекторизованный, тоже с использованием broadcasting. Находятся расстояния между всей матрицей X и отдельной строкой Y. При этом в результирующую матрицу ответы записываются построчно.

Сравнение результатов



Выводы

С циклами выходит все колоссально долго (обратите внимание на логарифмическую шкалу). Остальные же реализации работают примерно одинаково, но готовая реализация все же чуточку лучше.

2.8 Задача 8

Формулировка

Реализовать функцию вычисления логарифма плотности многомерного нормального распределения. Входные параметры: точки X, размер (N, D), мат. ожидание m, вектор длины D, матрица ковариаций C, размер (D, D). Плотность многомерного нормального распределения имеет формулу:

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{n/2}|C|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-m)C^{-1}(\mathbf{x}-m)^T}$$

Описание решений

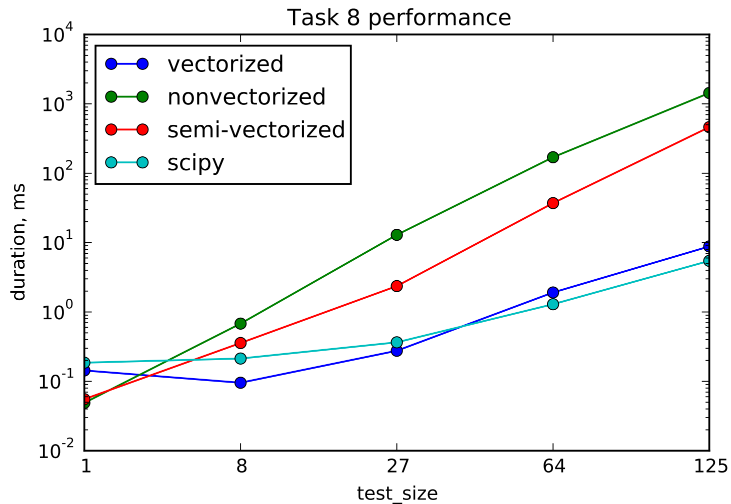
1. Векторизованный

```
def function1(X, m, C):
    return np.diag(np.log(np.exp(-0.5 * np.dot(np.dot((X-m),
    np.linalg.inv(C)), (X-m).T)) / ((2*np.pi) **
    (X.shape[1]/2.0) * (np.linalg.det(C))**0.5)))
```

2. Делается всё то же самое, только все векторные вычисления (кроме подсчета обратной матрицы ковариации и ее определителя) заменены на вычисления через циклы.

3. Полувекторизованный — невекторизованный вариант с векторными вычислениями произведений матриц.

Сравнение результатов



Выводы

Как и ожидалось, векторизованный вариант лучше всех, содержащих невекторизованные операции, но чуть хуже встроенной реализации.

3 Глобальные выводы

Как можно заметить, в серьезных задачах следует использовать векторизацию всегда, когда возможно (по крайней мере, судя по данным задачам). Это объясняется тем фактом, что циклы в языке Python работают значительно дольше, чем циклы в языке C++, на котором реализовано ядро NumPy и Scipy. Однако если гарантировано, что функция будет работать с данными небольших размеров, имеет смысл задуматься над простой, быстрой по написанию реализацией. Также написание невекторизованных функций полезно для демонстрации быстроты работы векторизованных вариантов.