МГУ им. Ломоносова
Факультет ВМК
Кафедра ММП

Изучение Python, NumPy

Задание выполнила:

студентка 3 курса Соболева Дарья

Москва
2016

Содержание

1 Описание задания

Задание представляет собой 8 задач. Для каждой задачи рассматривается несколько вариантов кода различной эффективности:

1. полностью векторизованный вариант
2. вариант без векторизации
3. наиболее читаемый способ или частично векторизованный вариант.

Полученные результаты для каждой из задач представлены в виде графиков зависимостей времени выполнения всех реализаций от размера входных данных, элементы которых, с целью обеспечения чистоты экспериментов, являются случайными числами.
Осуществляется несколько запусков каждого алгоритма, оценивается `timeit.best`. К каждой задаче присутствуют краткие выводы, сделанные на основании оценок полученных результатов.

2 Структура проекта

Работа была выполнена на языке `Python` версии `3.5`.

Каждая задача была оформлена в отдельном модуле. Так как в рамках одной задачи рассматривалось несколько вариантов кода, решено было объединить их в единый класс, с методами, соответствующими различным реализациям поставленной задачи.

Ко всем задачам присутствуют автоматические тесты, проверяющие совпадение результатов работы всех вариантов кода. Для каждой реализации проверяется корректность работы на матрицах различной размерности. В некоторых задачах используется иная идеалогия тестирования, о чем дополнительно оговорено в описании. Тесты используют встроенный в `Python` фреймворк `unittest`.

Проведение экспериментов осуществлялось в `Ipython Notebook`, с вынесением кратких комментариев к полученным результатам.

В конце работы была осуществлена проверка всех модулей на соответствие `style guide PEP 8`.

3 Эксперименты

3.1 Задача №1

Условие: подсчитать произведение ненулевых элементов на диагонали прямоугольной матрицы.

Доп. условие: полагаем, что у матрицы `X` есть на диагонали ненулевые элементы.

Прототипы используемых функций:

1. Полностью векторизованный вариант, использующий метод `numpy.diag(X)`

```
ProdNonZero.vect_version(X)
   TestProdNonZero.test_vect_version_X_0darray()
   TestProdNonZero.test_vect_version_X_1darray()
   TestProdNonZero.test_vect_version_X_2darray()
```

2. Полностью не векторизованный вариант

```
ProdNonZero.non_vect_version(X)
    TestProdNonZero.test_non_vect_version_X_0darray()
    TestProdNonZero.test_non_vect_version_X_1darray()
    TestProdNonZero.test_non_vect_version_X_2darray()
```

3. Вариант с частичной векторизацией

```
ProdNonZero.medium_version(X)
    TestProdNonZero.test_medium_version_X_0darray()
    TestProdNonZero.test_medium_version_X_1darray()
    TestProdNonZero.test_medium_version_X_2darray()
```

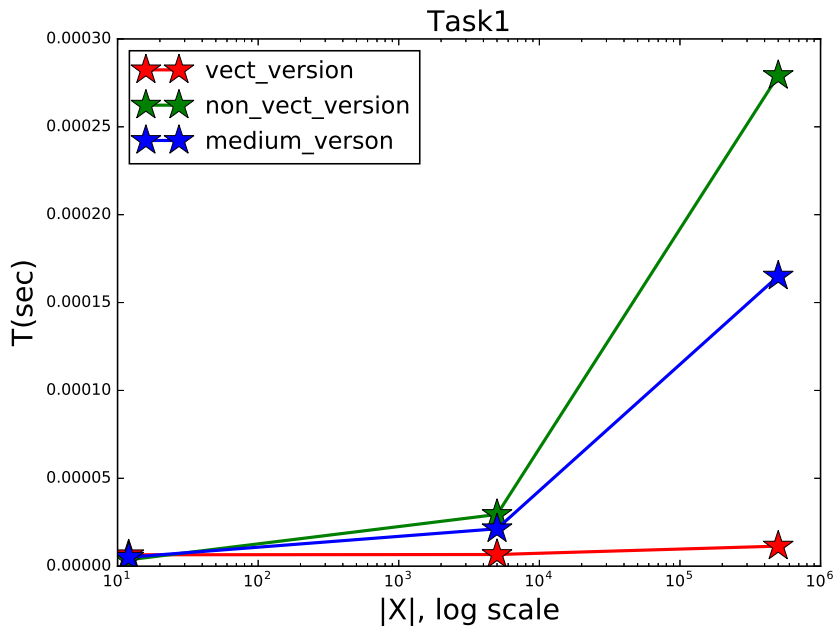

Рис. 1: Задача №1

Выводы: На графике 1 видно, что ускорение от векторизации ощутимо на матрицах большого размера.

Самая высокая скорость работы наблюдается у векторизованного варианта.

3.2 Задача №2

Условие: дана матрица `X` и два вектора одинаковой длины `i` и `j`. Построить вектор `numpy.array([X[i[0], j[0]], X[i[1], j[1]], . . . , X[i[N-1], j[N-1]]])`.

Доп. условие: полагаем, что матрица `X` имеет соответствующие индексы.

Прототипы используемых функций:

1. Полностью векторизованный вариант с методом `numpy.hstack(i, j)`

```
MatrixFromVectors.vect_version_hstack(X, i, j)
    TestMatrixFromVectors.test_vect_version_hstack_0darray()
    TestMatrixFromVectors.test_vect_version_hstack_1darray()
```

2. Полностью векторизованный вариант с методом `numpy.concat(i, j)`

```
MatrixFromVectors.vect_version_concat(X, i, j)
    TestMatrixFromVectors.test_vect_version_concat_0darray()
    TestMatrixFromVectors.test_vect_version_concat_1darray()
```

3. Полностью векторизованный вариант, использующий транспонирование

```
MatrixFromVectors.vect_version_concat(X, i, j)
   TestMatrixFromVectors.test_vect_version_transpose_0darray()
   TestMatrixFromVectors.test_vect_version_transpose_1darray()
```

4. Полностью не векторизованный вариант

```
MatrixFromVectors.non_vect_version(X, i, j)
   TestMatrixFromVectors.test_non_vect_version_0darray()
   TestMatrixFromVectors.test_non_vect_version_1darray()
```

5. Самый интуитивно простой способ: `X[i, j]`

```
MatrixFromVectors.fast_version(X, i, j)
    TestMatrixFromVectors.test_fast_version_0darray()
    TestMatrixFromVectors.test_fast_version_1darray()
```

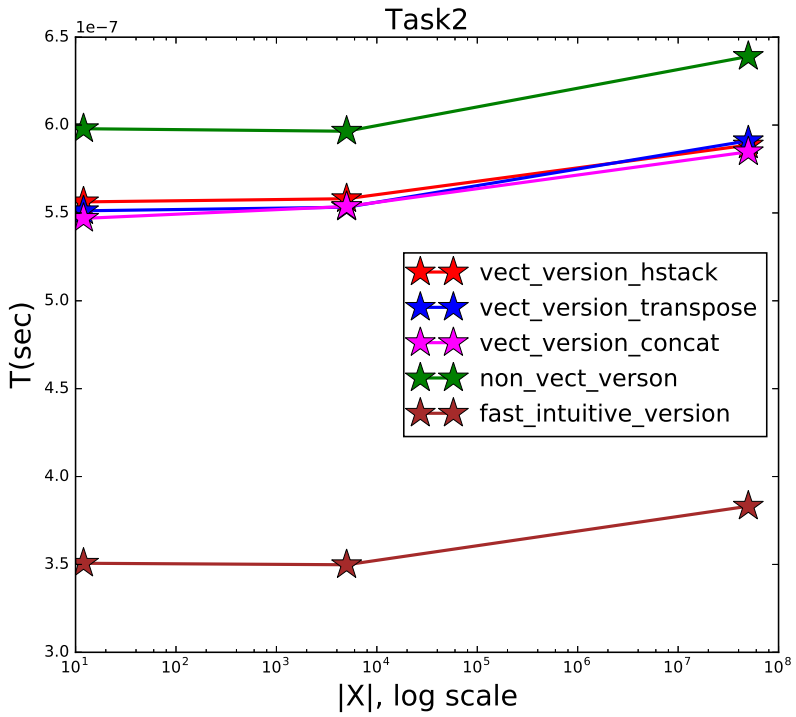

Рис. 2: Задача №2

Выводы: Оценивая график 2, приходим к выводу, что самым эффективным является интуитивно простой, основанный на свойствах `numpy.array` способ.

3.3 Задача №3

Условие: даны два вектора `x` и `y`. Проверить, задают ли они одно и то же мультимножество.

Прототипы используемых функций:

1. Полностью векторизованный вариант

```
MultiSet.vect_version(x, y)
   TestMultiSet.test_vect_version_0darray()
   TestMultiSet.test_vect_version_1darray()
```

2. Полностью векторизованный вариант, использующий `numpy.unique`

```
MultiSet.another_vect_version(x, y)
   TestMultiSet.test_another_vect_version_0darray()
   TestMultiSet.test_another_vect_version_1darray()
```

3. Полностью не векторизованный вариант, используюищий `collections.Counter`

```
MultiSet.non_vect_version(x, y)
   TestMultiSet.test_non_vect_version_0darray()
   TestMultiSet.test_non_vect_version_1darray()
```

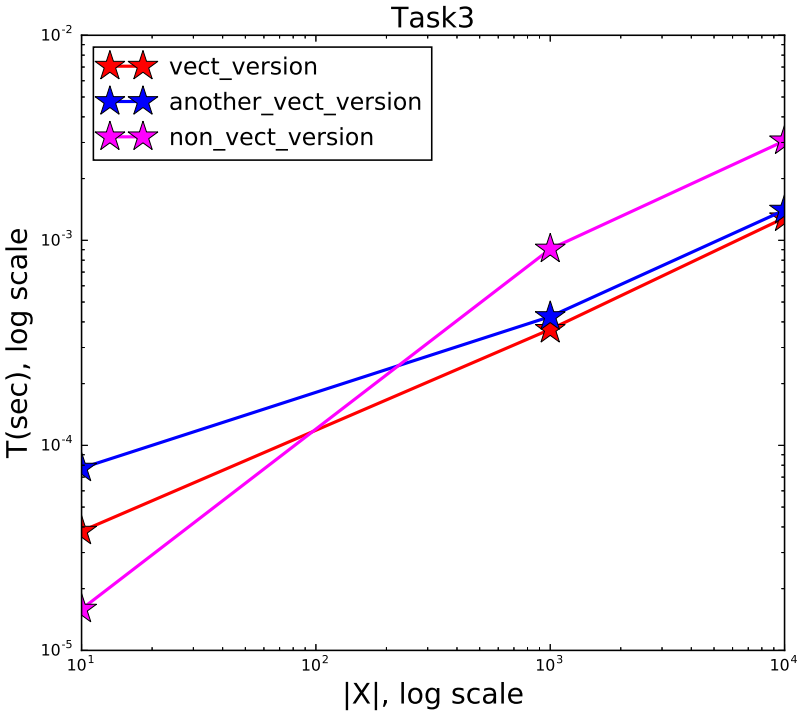

Рис. 3: Задача №3

Выводы: Очевидно, что на больших данных предпочтение отдается векторизованным способам. Даже эффективно реализованные библиотечные функции не могут конкурировать. Что и демонстрирует нам график 3.

3.4 Задача №4

Условие: найти максимальный элемент в векторе `x` среди элементов, перед которыми стоит нулевой.

Доп. условие: полагаем, что нулевые элементы присутствуют, а также не стоят на последнем месте в векторе.

Прототипы используемых функций:

1. Полностью векторизованный вариант с методом `numpy.roll`

```
MaxAfterZero.vect_version(x)
   TestMaxAfterZero.test_vect_version()
   TestMaxAfterZero.test_vect_version_zeros()
```

2. Метод, использующий `map-filter-reduce`

```
MaxAfterZero.smart_version(x)
   TestMaxAfterZero.test_smart_version()
   TestMaxAfterZero.test_smart_version_zeros()
```

3. Полностью не векторизованный вариант

```
MaxAfterZero.non_vect_version(x)
   TestMaxAfterZero.test_non_vect_version()
   TestMaxAfterZero.test_non_vect_version_zeros()
```

В качестве входных данных, использующихся для тестирования каждой из реализаций, подаются вектор, содержащий хотя бы один ненулевой элемент, а также вектор, содержащий только нулевые элементы.

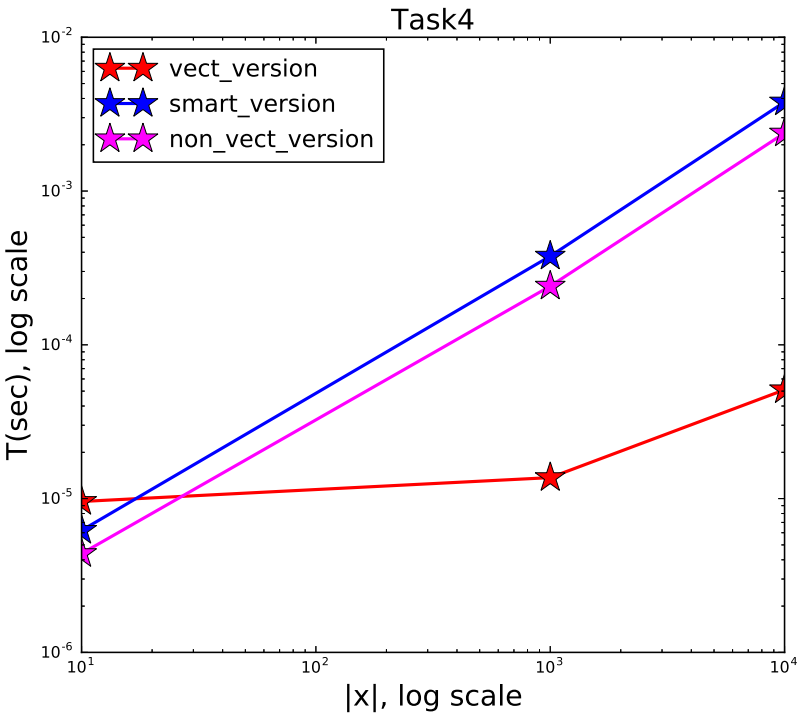

Рис. 4: Задача №4

Выводы: На графике 4 наблюдаем очень схожее поведение «умного» и по-другому реализованного, но также невекторизованного способа. Самой эффективной по-прежнему остается векторизация.

3.5 Задача №5

Условие: 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)`.

Доп. условие: полагаем, что на вход были поданы матрица и вектор указанных размеров.

Прототипы используемых функций:

1. Полностью векторизованный вариант, использующий подход `numpy.broadcast`

```
ConvertImage.vect_version(img, weights)
    TestConvertImage.test_vect_version()
```

2. Вариант с частичной векторизацией

```
ConvertImage.medium_version(img, weights)
    TestConvertImage.test_medium_version()
```

3. Полностью не векторизованный вариант

```
ConvertImage.non_vect_version(img, weights)
    TestConvertImage.test_non_vect_version()
```

Корректность каждой из реализаций проверяется на изображениях, сохраненных при помощи функции `numpy.save`.

(1) (2)

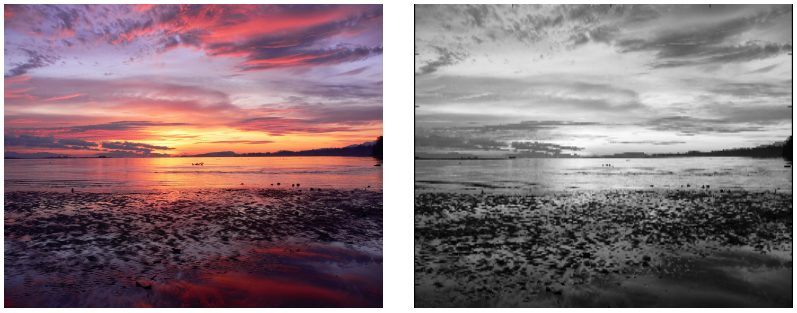

Рис. 5: Пример работы алгоритмов

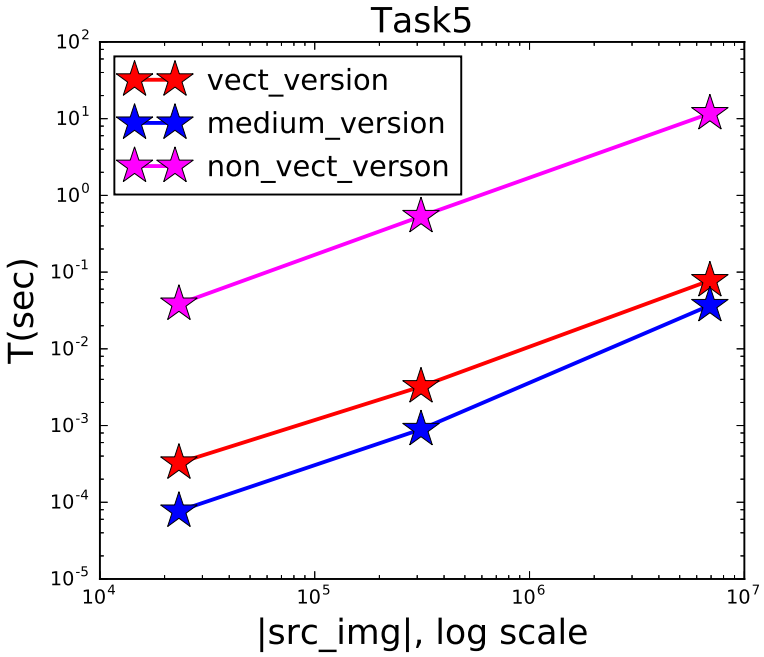

Рис. 6: Задача №5

Выводы: график 6 показал, что наиболее эффективным является способ с частичной векторизацией.

3.6 Задача №6

Условие: реализовать кодирование длин серий (`Run-length-encoding`).
Дан вектор x. Необходимо вернуть кортеж из двух векторов одинаковой длины. Первый содержит числа, а второй - сколько раз их нужно повторить.

Доп. условие: считаем, что вектор не пустой и в нем нет констант типа `None`.

Прототипы используемых функций:

1. Полностью векторизованный вариант с методами
   `numpy.flatnonzero, numpy.isclose, numpy.diff`

   ```
   RunLengthEncode.vect_version(x)
      TestRunLengthEncode.test_vect_version_0darray()
      TestRunLengthEncode.test_vect_version_1darray()
   ```

2. Метод, использующий `itertools.groupby`

   ```
   RunLengthEncode.version_groupby(x)
      TestRunLengthEncode.test_groupby_version_0darray()
      TestRunLengthEncode.test_groupby_version_1darray()
   ```

3. Полностью не векторизованный вариант

   ```
   RunLengthEncode.non_vect_version(x)
      TestRunLengthEncode.test_non_vect_version_0darray()
      TestRunLengthEncode.test_non_vect_version_1darray()
   ```

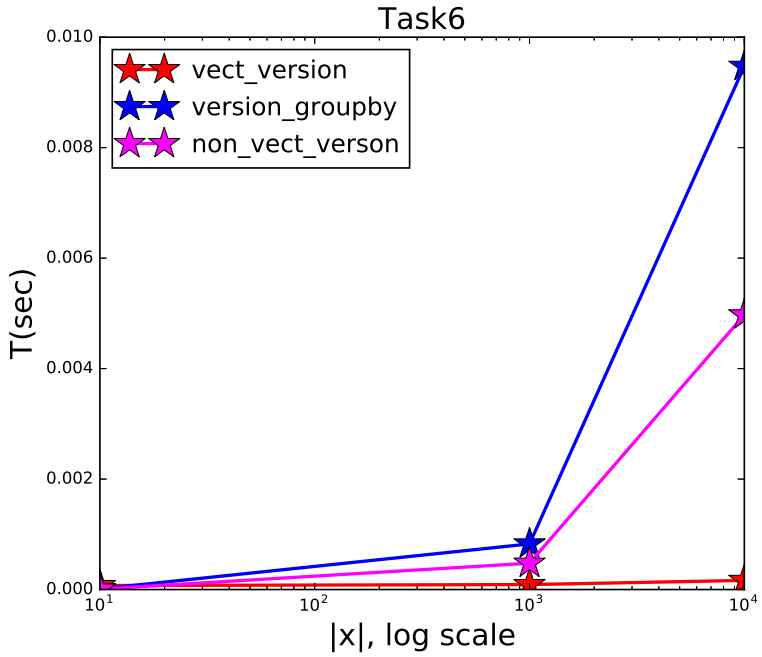

Рис. 7: Задача №6

Выводы: график 7 показал, что векторизованный способ снова является самым эффективным, в то время как метод, использующий `collections.groupby` плох даже по сравнению с невекторизованным.

3.7 Задача №7

Условие: даны две выборки объектов - `X` и `Y`. Вычислить матрицу евклидовых расстояний между объектами. Сравнить с функцией `scipy.spatial.distance.cdist`.

Доп. условие: полагаем, что `X` и `Y` имеют одинаковое число признаков.

Прототипы используемых функций:

1. Полностью векторизованный вариант, использующий подход `numpy.broadcast`

```
EuclideanDist.vect_version(X, Y)
   TestEuclideanDist.test_vect_version_0darray()
   TestEuclideanDist.test_vect_version_1darray()
   TestEuclideanDist.test_vect_version_2darray()
```

2. Метод, использующий `scipy.distance.euclidean`

```
EuclideanDist.version_scipy(X, Y)
   TestEuclideanDist.test_version_scipy_0darray()
   TestEuclideanDist.test_version_scipy_1darray()
   TestEuclideanDist.test_version_scipy_2darray()
```

3. Полностью не векторизованный вариант

```
EuclideanDist.non_vect_version(X, Y)
   TestEuclideanDist.test_non_vect_version_0darray()
   TestEuclideanDist.test_non_vect_version_1darray()
   TestEuclideanDist.test_non_vect_version_2darray()
```

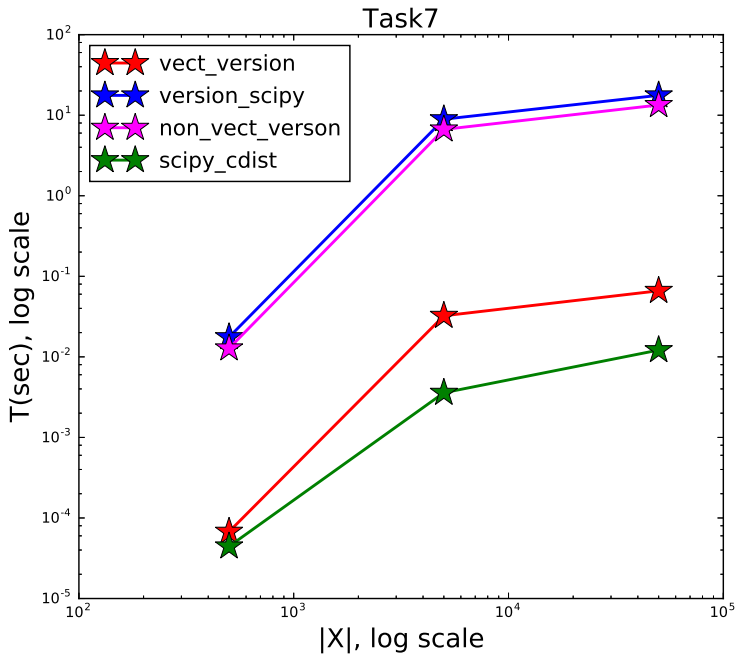

Рис. 8: Задача №7

Выводы: график 8 является ярким примером того, как удобно использовать эффективно реализованные библиотечные функции. Следует отметить, что векторизованный способ достаточно близок по скорости работы с рассмотренной библиотечной функцией.

3.8 Задача №8

Условие: 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)`. Разрешается использовать библиотечные функции для подсчета определителя матрицы, а также обратной матрицы, в том числе в невекторизованном варианте. Сравнить с `scipy.stats.multivariate_normal(m,C).logpdf(X)` как по скорости работы, так и по точности вычислений.

Заметка: можно рассматривать матрицу `X` как матрицу объекты-признаки (`N` обьектов, `D` признаков). Формула, использованная для расчетов логарифма плотности многомерного нормального распределения:

$$\ln p(x, m, \Sigma) = -\frac{D}{2}\ln(2\pi) - \frac{1}{2}\ln det(\Sigma) - \frac{1}{2}(x-m)^T\Sigma^{-1}(x-m)$$

$$p(x, m, \Sigma) = \frac{1}{\sqrt{(2\pi)^D det(\Sigma)}} e^{-\frac{1}{2}(x-m)^T\Sigma^{-1}(x-m)}$$

Доп. условие: полагаем, что матрица `C` симметрична и положительно определена.

Прототипы используемых функций:

1. Полностью векторизованный вариант

```
MultiLogNormDensity.vect_version(X, m, C)
   TestMultiLogNormDensity.test_vect_version()
```

2. Полностью векторизованный способ (более интуитивный)

```
MultiLogNormDensity.another_vect_version(X, m, C)
   TestMultiLogNormDensity.test_another_vect_version()
```

3. Полностью не векторизованный вариант

```
MultiLogNormDensity.non_vect_version(X, m, C)
   TestMultiLogNormDensity.test_non_vect_version()
```

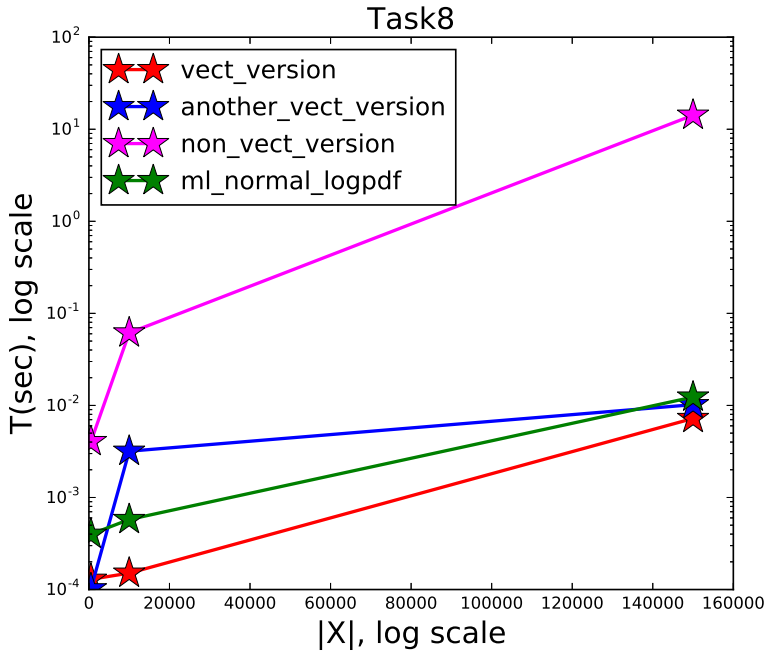

Рис. 9: Задача №8

Выводы: На графике 9 видно, что наиболее эффективным является векторизованный способ, даже по сравнению с библиотечной функцией `ml_normal_logpdf`. Необходимо уточнить, что в виду ограничений по времени, было проведено небольшое количество запусков каждого из алгоритмов, поэтому оценки могут быть смещенными.

4 Выводы

На основании результатов, полученных в ходе проведения экспериментов, установили несомненную пользу от проведения векторизации при работе с данными больших размеров. Кроме того, в некоторых задачах осуществлялось сравнение каждой реализации с существующими библиотечными функциями. Было выявлено несколько случаев, в которых библиотечные функции работали значительно быстрее (Задача №7). Однако, гораздо более типичной ситуацией являлась наиболее эффективная работа векторной реализации по сравнению со всеми рассмотренными функциями. А значит, в данной ситуации, достаточно написать простой, быстро работающий векторизованный способ. Важно отметить, что осуществление автоматической проверки корректности написанных функций очень помогло написанию правильных, работающих на данных разной размерности программ.