Аннотация

В данной работе обсуждается целесообразность реализации и эффективность некоторых алгоритмов машинного обучения на языке программирования Go. Представлены сравнения скорости работы решения задач линейной алгебры на языках Go, C++, Python, реализованные с использованием широко известных и популярных библиотек: Gonum [1], Armadillo [2], Numpy [3]. Представлены сравнения скорости работы и оценки результатов алгоритмов машинного обучения (linear regression, PCA, DBSCAB, k-means, logistic regression) на языках Go, C++, Python, реализованные самостоятельно (Go) и с использованием готовых библиотек mlpack [4], sklearn [5] для C++ и Python соответственно. В заключении приводится вывод о целесообразности развития пакетов алгоритмов машинного обучения на языке программирования Go.
Исходный код можно найти на github:
https://github.com/bayesiangopher/bayesiangopher

Annotation

This paper discusses the feasibility and effectiveness of some machine learning algorithms in the Go programming language. The paper presents comparisons of the speed of solving linear algebra problems in Go, C++, Python languages, implemented using well-known and popular libraries: Gonum [1], Armadillo [2], Numpy [3]. The article presents comparisons of the speed and evaluation of the results of machine learning algorithms (linear regression, PCA, DBSCAB, k-means, logistic regression) in Go, C++, Python, implemented independently (Go) and using ready-made libraries mlpack [4], sklearn [5] for C++ and Python, respectively. In conclusion, the conclusion about the expediency of the development of machine learning algorithm packages in the Go programming language is given.

Source code can be found on github:
https://github.com/bayesiangopher/bayesiangopher

Содержание

Введение

Сегодня необходимость решения задач машинного обучения является одним из главных условий получения результата во множестве направлений человеческой деятельности: наука, медицина [6]

“…With the increase in antibiotic resistance, exploiting ML techniques is already proving quite powerful in identifying new antibacterial agents in a faster and potentially cheaper way
” [7],

экология [8], политика [9][10] и бизнес. А быстрое и корректное решения этих задач является ключом к успеху, увеличению роста скорости получения необходимых результатов и развитию новых технологий.

На данный момент большинство компаний и лабораторий используют Python, точнее его обертку над C/C++, для решения задач машинного обучения (см. рис. 1) [11]. Мы видим в этом несколько проблем, негативно сказывающихся на отрасли, основными из которых можно выделить:

1. Чистый код на Python будет медленным и ресурсоемким;
2. Будучи оберткой вокруг C/C++, Python лишает разработчиков и исследований возможности его расширения и оптимизации, а также профилирования и отлаживания своего кода;
3. Предоставляя настолько высокий уровень в библиотеках (sklearn, tensorflow, pytorch) разработчики и исследователи, в большинстве своем, избегают изучения тонкостей работы алгоритмов, что неизбежно ведет понижению уровня компетенции и в дальнейшем невозможности решать действительно сложные задачи, требующие творческого, нового подхода.

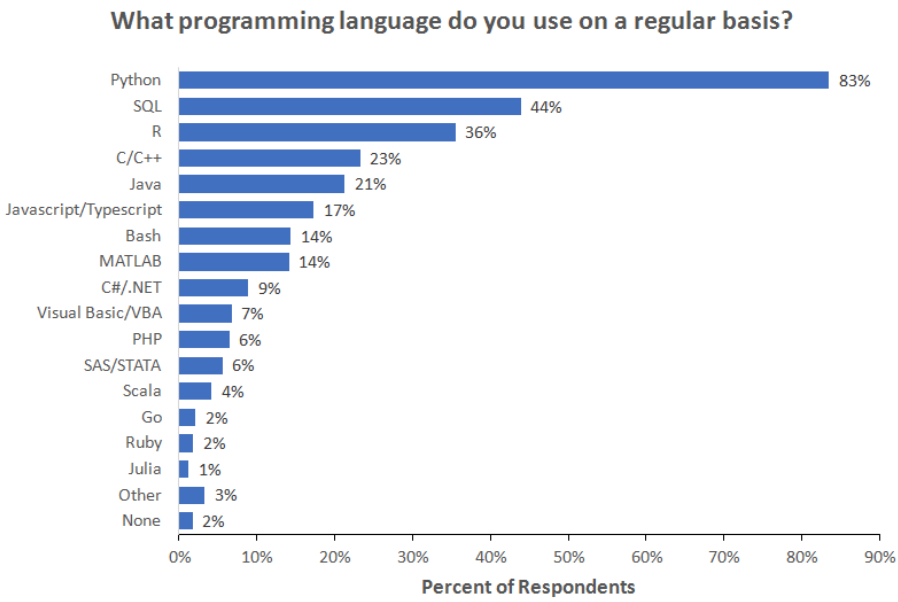


Рисунок 1 - Данные о результатах самых популярных языках с конкурса Kaggle 2018

В связи с этим, мы поставили перед собой задачу подобрать оптимальный, с нашей точки зрения, язык и начать, в рамках магистерской диссертации, разработку библиотеки для машинного обучения на выбранном языке программирования (далее ЯП) – Go.

Актуальность работы определяется растущей потребностью в специалистах по машинному обучению (+226% от 2018 года по данным Яндекс.Парктикума и аналитической службы HeadHunter [18]) которым необходимо обучаться, что сделать фундаментальным образом, строя этот процесс вокруг кода, а не теоретической математики, очень сложно, а также менять ЯП, в которых они компетентны, если это не Python/C/C++. Кроме того, использование нескольких ЯП в одном проекте, даже на микросервисной архитектуре, сильно усложняет задачу введения машинного обучения в сервисы реализованные на других ЯП, что вынуждает многие команды реализовывать проекты на Python вместо использования других, возможно более удачных для данной задачи, ЯП, либо отказываться от внедрения логик, связанных с машинным обучением.

В работе представлены результаты исследования скорости решения задач линейной алгебры, на которых базируется большинство алгоритмов машинного обучения, и, непосредственно, алгоритмов машинного обучения,

для определения возможности и целесообразности использования ЯП Go для решения задач данного класса.

В первой главе рассмотрен ЯП Go и приведено обоснование выбора, а также актуальности данной темы. В главе «Существующие инструменты машинного обучения» представлен короткий обзор основных инструментов с API для Python, C++ и обозначена главная, на наш взгляд, проблема таких инструментов, а также представлен обзор существующих решений на Go. В главах посвященных тестированию приводятся результаты сравнения скорости работы алгоритмов на вышеупомянутых языках с использованием описанных в предыдущей главе инструментов. Последняя глава посвящена анализу полученных результатов и планам по развитию нашей библиотеки.

Язык программирования Go

Разобьем языки на 2 группы: первая группа определяет сложность синтаксиса и стандартной библиотеки языка (easy и hard), вторая группа (simple и complex, где complex – это языки с большим набором “магии”) определяет объем “магии” в языке, т.е. неявных конструкций, реализация которых скрыта от пользователя, яркий пример – инструкция in в Python.

Go – язык идеологии simple и easy, что определяет простоту написания кода на нем, в отличие от hard языков, таких как C++, Java, Lisp и т.п., т.е. простоту и лаконичность синтаксиса, а концепция simple отличает его от языков вида complex отсутствием “магии”, например, отсутствует встроенная функция min() для поиска минимального элемента в срезе, что вынуждает разработчиков создавать собственные реализации подобного функционала и обращать внимание на сложность решения данной задачи и, возможно, использовать другие структуры данных такие как деревья.

Проект Go включает в себя сам язык, его инструментарий, стандартные библиотеки и последнее (по списку, но не по значению) – культуру радикальной простоты. Будучи одним из современных языков высокого уровня, Go обладает преимуществом ретроспективного анализа других языков, и это преимущество использовано в полной мере: в Go имеются сборка мусора, система пакетов, полноценные функции, лексическая область видимости, интерфейсы системных вызовов, и неизменяемые строки, текст в которых кодируется UTF-8 кодировкой, Go обладает гибкой и необычной возможностью параллельности, основанной на CSP: стеки переменного размера легких потоков (goroutines) Go изначально малы, чтобы создание одной go-подпрограммы было дешевым, а создание миллиона – практичным [12].

Основные плюсы языка Go:

1. Быстрая компиляция в машинный код [13] (Implementation);
2. Простой синтаксис [13] (Changes from C);
3. Строгая типизация [13] (Types);
4. Быстрая скорость работы;
5. Сборщик мусора [13] (Why do garbage collection? Won't it be too expensive?);
6. Отсутствие "магии";
7. WebAssembly [14];
8. Goroutines [13] (Design).

Большинство пунктов аргументировано ссылками на документацию, что касается седьмого пункта - технология WebAssembly позволяет компилировать высокоуровневые языки в бинарный формат инструкций, который можно развертывать в web, следовательно код, написанный на Go можно компилировать в клиентский код приложения.

Кроме того, нельзя не отметить низкий порог вхождения. Если предположить, что его можно определить через объем документации языка, то вот пример объема документации на некоторые языки:

- Go - ~150 страниц;
- Scala - ~ 190 страниц;
- ECMA-262 – ~580 страниц;
- C – ~550 страниц;
- Java – ~780 страниц;
- C++ - ~1300 страниц.

Важным плюсом Go является именно простота его синтаксиса и жестко зафиксированный стиль кода благодаря инструменту автоматического форматирования gofmt [15], так как огромный объем времени разработчик тратит именно на чтение чужого кода.

Для решения задач машинного обучения, кроме существования большой экосистемы и готовых инструментов, ключевым критерием выбора

языка является скорость работы (пункт 4), этот вопрос был нами исследован и в последующих главах представлены исчерпывающе результаты. В данной главе приведем небольшое сравнение скорости на задаче возведения в степень float64 числа 100 000 000 раз и вычитания основания, результаты для Go, C, Java:

Показатель степени 2.4:

Java: result: 1.053906e+24, during: 7432 ms
C: result: 1.053906e+24, during: 5544 ms
Go: result: 1.053906e+24, during: 8.716807708s

Показатель степени 2:

Java: result: 1.543194e+21, during: 630 ms
C: result: 1.543194e+21, during: 852 ms
Go: result: 1.543194e+21, during: 3.336549272s

Показатель степени 2, однако вместо стандартной функции Pow() использовалось просто умножение x*x:

Java: result: 1.543194e+21, during: 636 ms
C: result: 1.543194e+21, during: 340 ms
Go: result: 1.543194e+21, during: 115.491272ms

Результаты впечатляют.

Основываясь на перечисленных плюсах, был выбрал язык Go, кроме того, его растущая популярность в больших компаниях все больше заставляет обращать на него внимание еще больше [16].

Актуальность данной работы подтверждает недавно представленные TensorFlow API для ЯП Swift. Сами разработчики TensorFlow выделяют следующие плюсы программирования нейронных сетей на базе TensorFlow с использованием Swift [17]:

- использование лучших подходов в проектировании архитектуры API и написании методов, которые сформировались в результате развития индустрии;
- использование одной технологии для продуктового кода и для проектирования машинного обучения;
- лучшие возможности для обучения и привлечения новых разработчиком, в стек которых не входят C/C++ и Python;

Все эти пункты касаются и Go, кроме того, последний пункт особенно актуален, в то время как Swift используют, преимущественно, для мобильных приложений, Go стремительно набирает популярность и, не без преувеличения будет сказано, захватывает web, машинное обучение в котором особенно актуально для бизнеса и исследований.

Кроме того, учитывая, что нейронные сети и алгоритмы машинного обучения становятся "умнее", быстрее и легковеснее, а индустрия программирования сейчас очень много внимания концентрирует на безопасности личных данных, не в последнюю очередь из-за большого внимания правительства разных стран к этому вопросу, которые необходимо отправлять на серверы для обучения моделей, есть предпосылки к вынесению некоторого функционала на клиентскую часть мобильных и веб приложений, и именно тут вступает в игру WebAssembly.

Существующие инструменты для машинного обучения

Большинство языков, использующихся для решения задач машинного обучения имеют ряд недостатков, которые, на наш взгляд, являются критичными и делают Go таким привлекательным для решения поставленной задачи:

1. Python: медленный при выполнении, требует очень много памяти из-за динамической типизации, плохо поддерживает парадигму параллельного выполнения (GIL);
2. Python как обертка над C/C++: слишком высокоуровневый, практически невозможно внести изменения в реализацию, однако быстрый, простой и предоставляет огромное количество готовых решений для использования;
3. C/C++: сложен для освоения, сложен для реализации, отсутствует сборщик мусора, однако очень быстрый;
4. JavaScript: прост для освоения и очень гибкий, небезопасный из-за динамической типизации и неявного преобразования типов, медленный;
5. Java: средней сложности для освоения, JVM (минус), немного медленный для вычислений.

Python

Стандартом для решения задач машинного обучения является Python как обертка над C/C++, однако, как мы уже обозначили выше, это несет в себе множество проблем, связанных с обучением, расширением и отладкой кода. Основные библиотеки для машинного обучения на Python:

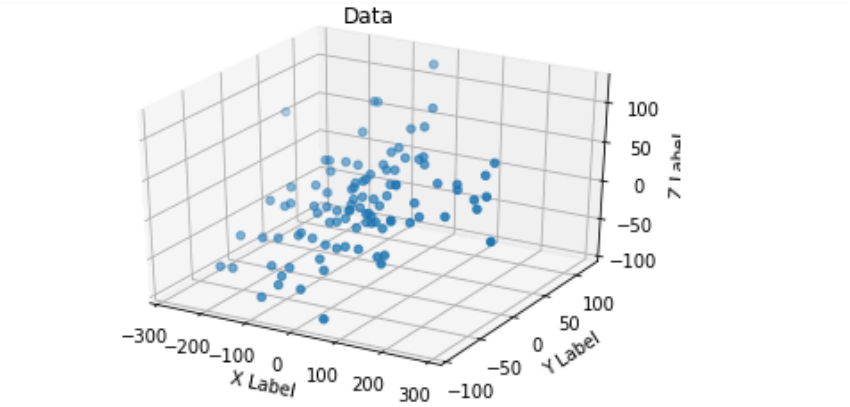
1. Scikit-learn [5];
2. TensorFlow [19];
3. Keras [20];
4. Caffe [21];
5. pyTorch [22];
6. xgboost [23].

Большинство из приведенных фрэймворков и библиотек решают вопросы связанные с проектированием и использованием нейронных сетей и вычислениями на GPU, в рамках данной работы мы не изучали эти вопросы полноценно, однако нам кажется, если Go показывает себя хорошо в решении задач машинного обучения, а использование его для вычислений на GPU возможно и развивается [24][25], то и в этой сфере он может показать хорошие результаты. Кроме того, для Go существует библиотека для облачного TPU API от Google [26], созданного именно для работы нейронных сетей. В дальнейшем развитии библиотеки планируется разработка API для проектирования нейронных сетей, кроме того, мы общались с создатель самой популярной библиотеки нейронных сетей для Go [25], и он приглашал нас в разработку.

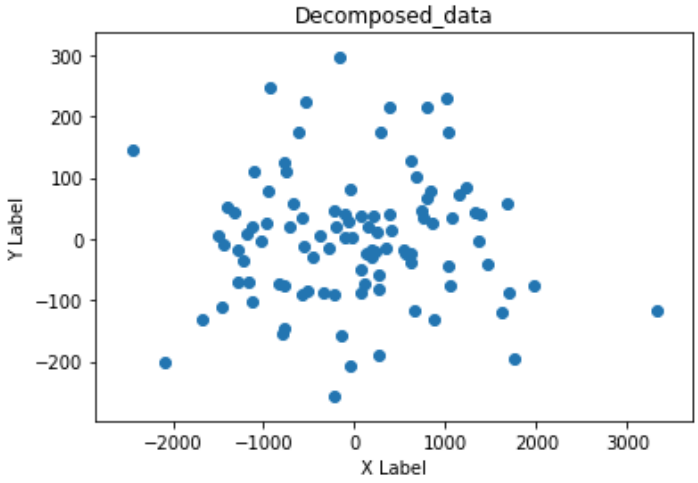
Основной библиотекой именно для машинного обучения является scikit-learn, имеющей практически все известные алгоритмы для обучения с полноценной и хорошей документацией, рассмотрим взаимодействие с ней подробнее на примере алгоритма понижения размерности PCA основанного на сингулярном (SVD) разложении матриц реализованном по паттерну LAPACK архитектуры на C [27]:

```
temp_0 = (np.random.randn(100) * 100).reshape(100,1)
temp_1 = (np.random.randn(100) * 37).reshape(100,1)
temp_2 = (np.random.randn(100) * 999).reshape(100,1)
X = np.hstack((temp_0, temp_1, temp_2))
show_data_3D(X, "Data")
print(f"Data:\n {X[0:10,:]}")
pca = PCA(n_components=2)
X_decomposed = pca.fit_transform(X)
show_data(X_decomposed, "Decomposed_data")
```

```
print(f"Decomposed data:\n {X_decomposed[0:10,:]}")
print(f"Var of decomposed data: {pca.explained_variance_ratio_}")
def show_data(data, title):
    fig, ax = plt.subplots()
    ax.use_sticky_edges = False
    ax.margins(0.07)
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.scatter(data[:,0], data[:,1])
    ax.set_title(f"{title}")
    plt.show()
def show_data_3D(data, title):
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.scatter(data[:,0],data[:,1],data[:,1])
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    ax.set_title(f"{title}")
    plt.show()
```




```
Data:
 [[ -50.68765872   -7.79871416 1147.82009815]
 [-126.61751408   29.90789632  782.23748467]
 [ 108.16393112   -6.31992555 -658.59985182]]
```




```
Decomposed data:
 [[1025.88664966  -45.26074529]
 [ 659.9701204  -115.85041552]
 [-779.60844085  123.59936831]]
Var of decomposed data: [0.98731961 0.01117224]
```

Достаточно просто и понятно, происходит и расчет декомпозиции и коэффициентов для полученных данных, однако, все вычисления строятся на библиотеки numpy, которая полностью скрывает реализацию алгоритмов за C, что делает невозможным отладку и дополнение кода. Последовательно проходя по коду к моменту решения SVD в пакете numpy/linalg/ мы доходим до вызова функций из скомпилированного кода C:

```
if compute_uv:
    if full_matrices:
        if m < n:
            gufunc = _umath_linalg.svd_m_f
        else:
            gufunc = _umath_linalg.svd_n_f
    else:
        if m < n:
            gufunc = _umath_linalg.svd_m_s
        else:
            gufunc = _umath_linalg.svd_n_s

    signature = 'D->DdD' if isComplexType(t) else 'd->ddd'
    u, s, vh = gufunc(a, signature=signature, extobj=extobj)
    u = u.astype(result_t, copy=False)
    s = s.astype(_realType(result_t), copy=False)
    vh = vh.astype(result_t, copy=False)
    return wrap(u), s, wrap(vh)
else:
    if m < n:
        gufunc = _umath_linalg.svd_m
    else:
        gufunc = _umath_linalg.svd_n
```

далее пройти и понять, что происходит, поставить брэйкпоинты для отладки не получается, так как работает скомпилированный код. На github [3] можно найти код до компиляции, в нем происходит вызов C который решает задачу.

Можно возразить, что реализация SVD разложения достаточно рутинная задача, которую реализовали в numpy максимально правильным способом и сомневаться в этом не стоит, но от ошибок никто не застрахован. В ходе выполнения этой работы мы столкнулись с проблемой SVD разложения на Gonum, которая выделяла память для 14 000 000 000 000 float64 для разложения матрицы 100 000 x 2, несмотря на то, что Gonum включена в X пакеты самого языка Go, а значит планируется в стандартную библиотеку. Именно благодаря возможности пройти по коду, отладить его,

мы нашли проблему и локально ее подправили, отписавшись разработчикам об этом, что, мы надеемся, поможет в дальнейшем ее решить.

Основная проблема использования scikit-learn кроется скорее в numpy, так как общую логику решения задачи можно понять по оберткам и коду методов классов алгоритмов машинного обучения, но совсем иначе дело обстоит с библиотеками и фреймворками для нейронных сетей, где вся логика и вычисления спрятаны в C/C++ код.

C/C++

C/C++ библиотеки написаны без оберток так как сам язык реализации является эталонным в вопросах скорости и работы с памятью, однако разработка и ее сложность на этом языке требует от специалистов очень хорошего уровня знаний и опыта работы не только в jupyter notebook, что, зачастую, является ключевой проблемой для бизнеса и исследований, где результат нужен как можно быстрее, а скорость можно увеличить добавив вычислительных мощностей. Кроме того, для библиотек на данных языках редко можно найти документацию сравнимую по удобству и полноте с документацией на Python, который считается эталонным примером документирования, и даже Go.

В нашей работе мы использовали библиотеку mlpack [4] основанную на библиотеки для вычислений armadillo [2].

Кроме этой библиотеки популярны:

1. Apach.SIGNA
2. TensorFlow
3. Caffe
4. Shogun

Go

Самой крупной библиотекой машинного обучения на Go является golearn [28], в которой реализованы:

- Алгоритмы кластеризации:
    - Expectation Maximization algorithm;
    - DBSCAN;
- Классификации:
    - KNN;
    - Bernoulli Naïve Bayes Classifier;
    - ID3;
- Алгоритмы регрессии:
    - Linear;
    - Logistic.

Всего 7 алгоритмов, при этом в проект 500 коммитов, 46 контрибьютеров и 6500 звезд на github. Количество реализованных алгоритмов очень мало, для сравнения, в scikit-learn реализовано более 100 моделей, поэтому реализация новых моделей и развитие Go в сфере машинного обучения актуальная и интересная задача.

Тестирования скорости решения задач линейной алгебры на языках Go, Python, C++

Алгоритмы машинного обучения строятся на решении задач линейной алгебры, поэтому для тестирования скорости работы Go в сравнении с Python, C++ были реализованы наборы тестов на основе библиотек Gonum, NumPy, Armadillo в разрезе набора двух типов задач:

1. Операции с векторами для размерностей 1024, 16384, 65535, 131072, 242144, 524288:
    a. Создание вектора случайных чисел [0, 100) (create);
    b. Масштабирования вектора (scale);
    c. Определение Евклидовой нормы вектора (l2_norm);
    d. Сумма векторов (add);
    e. Разница векторов (sub);
    f. Произведение векторов (dot).

2. Операции с матрицами для размерностей 32x32, 128x128, 256x256, 512x512, 1024x1024:
    a. Создание матрицы случайных числе [0, 100) (create);
    b. Масштабирование матрицы (scale);
    c. Транспонирование матрицы (transpose);
    d. Сумма матриц (add);
    e. Разница матриц (sub);
    f. Произведение матриц (dot);
    g. Нахождение определителя матрицы (det);
    h. Нахождение собственных значений и собственных векторов матрицы (eigens);
    i. Сингулярное разложение матрицы (svd);

j. Разложение Холецкого матрицы (cholesky).

Итоговые результаты для векторов и матрицы наибольшего размера (524288 и 1024x1024 соответственно) представлены в таблицах 1 и 2. Динамика скорости работы решения каждой задачи для каждой размерности представлена на графиках в Приложении Б.

Тестирование проводилось на виртуальной машине с характеристиками, указанными в Приложении А.

	Functions for vectors					
	create	scale	l2_norm	add	sub	dot
go	703,8264	1	4,223342	1,278452	1,632842	1
python	1	4,312027	2,921511	1	1	3,68021
c++	2,671569	1,578388	1	1,412641	1,752087	1,842717

Таблица 1 - результаты скорости работы языков go, python и C++ при операциях с векторами 524288x1

	Functions for matrices									
	create	scale	transpose	add	sub	dot	det	eigns	svd	cholesky
go	571,4286	3,424743	1	1,238684	1,136249	10,5678	9,089664	18,72961	24,82417	13,04874
python	1	2,302587	5179,423	1	1	2,009339	2,4411	1,306469	1,783467	6,062727
c++	1,602652	1	57724,67	1,300646	1,226381	1	1	1	1	1

Таблица 2 - результаты скорости работы языков go, python и C++ при операциях с матрицами 1024x1024

Gonum – библиотека с реализацией на чистом Go представляющая BLAS [29] стандарт для решения базовых векторных и матричных операций в LAPACK [30] архитектуре для решения задач нахождения собственных значений и векторов, сингулярного разложения и т.п.

На задачах, связанных с векторами (табл. 1), Go дает результаты лучше C++ и Python (масштабирование и перемножение), лучше C++, но хуже Python (суммирование и разница) и незначительно хуже C++ и Python при нахождении Евклидовой нормы, однако на создание вектора Go потребовалось в 700 раз больше времени чем Python, что свидетельствует о

недостаточно быстрой реализации функции math.rand(), можно предположить, что Go медленно выделяет память, но все функции библиотеки gonum реализованы так, что любая операция требует присвоения в выделенное место в памяти, так что в любом из тестов, кроме масштабирования, приходилось выделять память под новый вектор такой же длинны. Кроме того, при проектировании Go выделением памяти старались сделать максимально быстрой (три вида разных размеров аллокаторов, локальный кеш на P, оптимизация компилятора, инлайн и анализ эскейп последовательностей), поэтому проблема в реализации генерации случайных чисел.

На задачах, связанных с матрицами (табл. 2), Go дает результаты хуже, сумма, масштабирование и разница матриц требует одинаково времени на Go и на C++/Python, транспонирование в Go реализовано заменой характеристик кол-ва строк и столбцов в структуре, представляющей матрицу, данные же остаются неизменными, так как хранятся в срезе []float64. Но вот остальные алгоритмы: нахождение определителя, собственных чисел и векторов, сингулярное разложение и разложение Холецкого на Gonum занимают в 10-25 раз больше времени чем на C/C++, что является критичным, так как именно эти алгоритмы используются в алгоритмах машинного обучения. Мы предполагаем, что именно из-за способа хранения данных в структурах gonum возникает данная проблема, так как каждый элемент матрицы – это элемент в []float64 срезе, а векторизации, на которой базируется numpy, в gonum нет: вычисления происходят в двойных циклах.

Однако, учитывая, что равнение идет на numpy и armadillo, являющиеся основными библиотеками в своих языках для решения математических задач, которые развиваются уже много лет и привлекают внимание множества исследователей, улучшающих их, Gonum показывает хорошие результаты, предоставляет удобный API, так, например, выглядит инициализация матрицы:

```
 v := make([]float64, 12)
 for i := 0; i < 12; i++ {
  v[i] = float64(i)
 }
 A := mat.NewDense(3, 4, v)
```

Тут также показано, что данные матрицы хранятся в простом списке []float64, и свободно может быть изменен и оптимизирован именно под ваши цели.

Основные выводы на основании полученных результатов приведены в главе «Анализ полученных результатов».

Тестирования скорости решения задач машинного обучения на языках Go, Python, C++

В качестве базовых методов для тестирования скорости обучения в данной работе были рассмотрены следующие пять модели:

1. Линейная регрессия [];
2. Логистическая регрессия [];
3. DBSCAN [];
4. KMeans [];
5. Метод главных компонент (PCA []).

В качестве данных для обучения вышеупомянутых методов были рассмотрены следующие:

1. wwt_weather [31];
2. breast_canser_dataset [32];
3. (сгенерированный);
4. xclara [33];
5. boston_housing [34].

В отличие от библиотек на Python и C++, в которой линейная регрессия реализована при помощи методов семейства градиентного спуска, в Go был предложен подход с использованием SVD или QR разложения. Так как линейная регрессия сводится к решению метода наименьших квадратов, то задача может быть представлена в следующем виде:

$\min_w \|y - Xw\|^2$, где $y -$ вектор $(m \times 1)$ наблюдений;

$X -$ матрица $(m \times n)$регрессоров; $w -$ вектор $(n \times 1)$ весов.

$$(Xw - y)^T(Xw - y) \to \min_w.$$

$$X^T Xw = X^T y => w = (X^T X)^{-1} X^T y$$

В свою очередь матрица факторов может быть разложена при помощи SVD или QR разложения:

$X = UDV^T -$ сингулярнное разлоджение, где U и V унитарные матрицы порядка m и n, из левых и правых собственных векторов, соответственно, матрицы X;

$D -$ матрица $m \times n$ с собственными значениями X на главной диагонали.

Тогда решение будет представимо в виде:

$$w = (VD^T U^T UDV^T)^{-1} VD^T U^T y = VD^+ U^T,$$

где $D^+ -$ псевдообратная матрица.

Или $w = (R^T Q^T QR)^{-1} R^T Q^T y = (R^T R)^{-1} R^T Q^T y$, в случае разложения $X = QR$, где $Q -$ унитарная матрица порядка m; $R -$ верхняя треугольная матрица $m \times n$.

В силу плохой векторизации в библиотеке Gonum, из таблицы 3 видно, что метод реализованный на Go, в разы уступает реализации на Python, тем не менее обгоняя по скорости библиотеку mlpack.

Так как на данный момент, эталоном качества моделей машинного обучения служит Python, также были произведены сравнения точности полученный результатов (табл. 3).

lang	method	trainrows	traincols	time, ns	scores
go	linearregression	10000	2	198570927	0.943758
python	linearregression	10000	2	7272005	0.962096
cpp	linearregression	10000	2	787417803	0.934873

Таблица 3 - Результаты линейной регрессии для языков Go, Python, C++

Для реализации логистической регрессии был рассмотрен стохастический градиентный спуск, который разбивает обучающую выборку

на группы (batch) (алгоритм 1). Логистическая регрессия сводится к минимизации функционала $Q(w) = \sum_{i=1}^{n} \log(1 + exp(-(w, x_i)y_i)) \to min$

Алгоритм {1} Стохастический градиентный спуск.

---

1. Вход: Обучающая выборка X, темп обучения lr, темп забывания fr;
2. Выход: $w -$ вектор весов;
3. Инициализировать w и оценку функционала $\bar{Q} \coloneqq \frac{1}{n}\sum_{i=1}^{n} L_i(M)$, где $L_i(M) = \log(1 + e^M) -$ логорифмическая функция потерь
4. Пока не сойдутся w и/или $\bar{Q}$:
5. Случайным образом выбирать k объектов из X
6. $For\ j \coloneqq 0;\ j < k;\ j \mathrel{+}= 1$:
7. Вычислить потерю: $e_j \coloneqq L_j(w)$
8. Градиентный шаг: $w \coloneqq w - lr\nabla L_j(w)$
9. Оценить функционал $\bar{Q} \coloneqq fr * e_j + (1 - fr)\bar{Q}$
10. Конец

Данный подход аналогично обгоняет реализацию на C++ и уступает scikit-learn (табл. 4).

lang	Method	trainrows	traincols	time, ns	scores
python	logisticregression	499	30	29279112	0.842857
go	logisticregression	499	30	1273384054	0.885714
cpp	logisticregression	499	30	3168684502	0.857843

Таблица 4 - Результаты логистической регрессии для языков Go, Python, C++

В ходе тестирования было обнаружено, что алгоритм DBSCAN реализован не оптимально, по времени исполнения (табл. 5). Сам алгоритм может быть описан следующим образом [35]:

1. Выбираем $\varepsilon -$ радиус окрестности относительно некоторой точки и $minNeighbors -$ минимальное число точек, попадание которых в ε-окрестность некоторой точки определяет её как основную точку.

2. Ищем точки в в ε-окрестности каждой точки, выделяя основные точки с более чем $minNeighbors$ соседями.
3. Игнорируя неосновные точки, совершаем поиск связных компонент основных точек на графе соседей.
4. Определяем является ли неосновная точка шумом или достижимой ближайшему кластеру.

Правдивость результатов кластеризации методом DBSCAN была проведены графическим образом и продемонстрирована на графиках 1 и 2.

lang	method	trainrows	traincols	time, ns
go	dbscan	4000	2	2.50567E+11
python	dbscan	4000	2	83724093
cpp	dbscan	4000	2	196703739

Таблица 5 - Результаты DBSCAN для языков Go, Python, C++

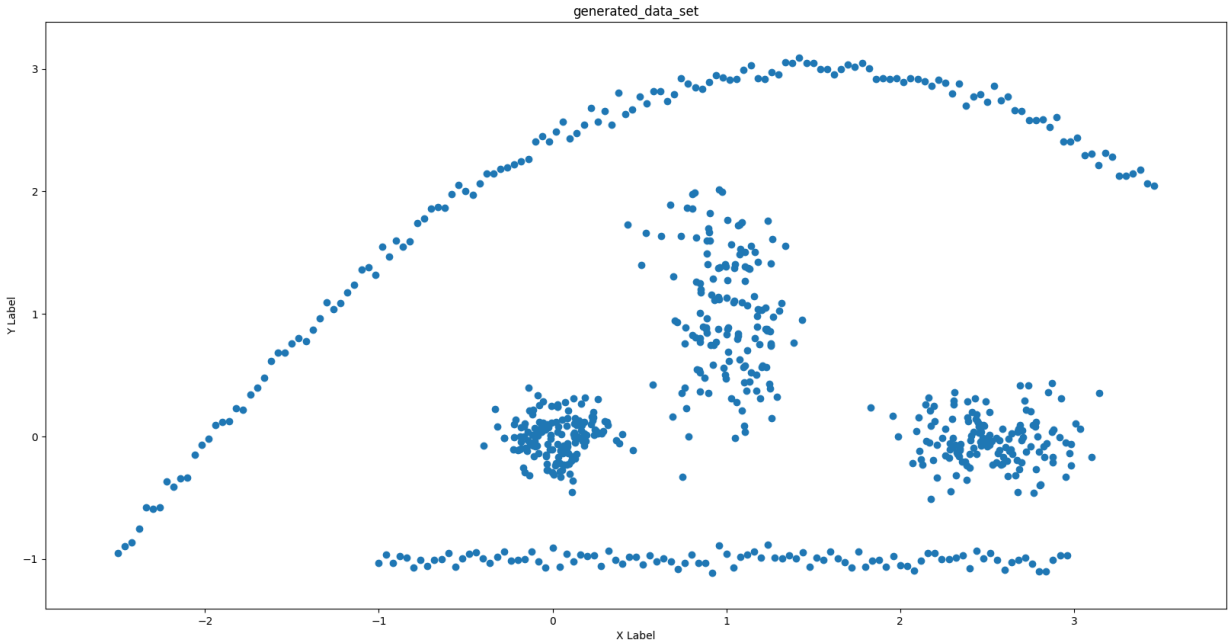


Рисунок 2 - Сгенерированные данные

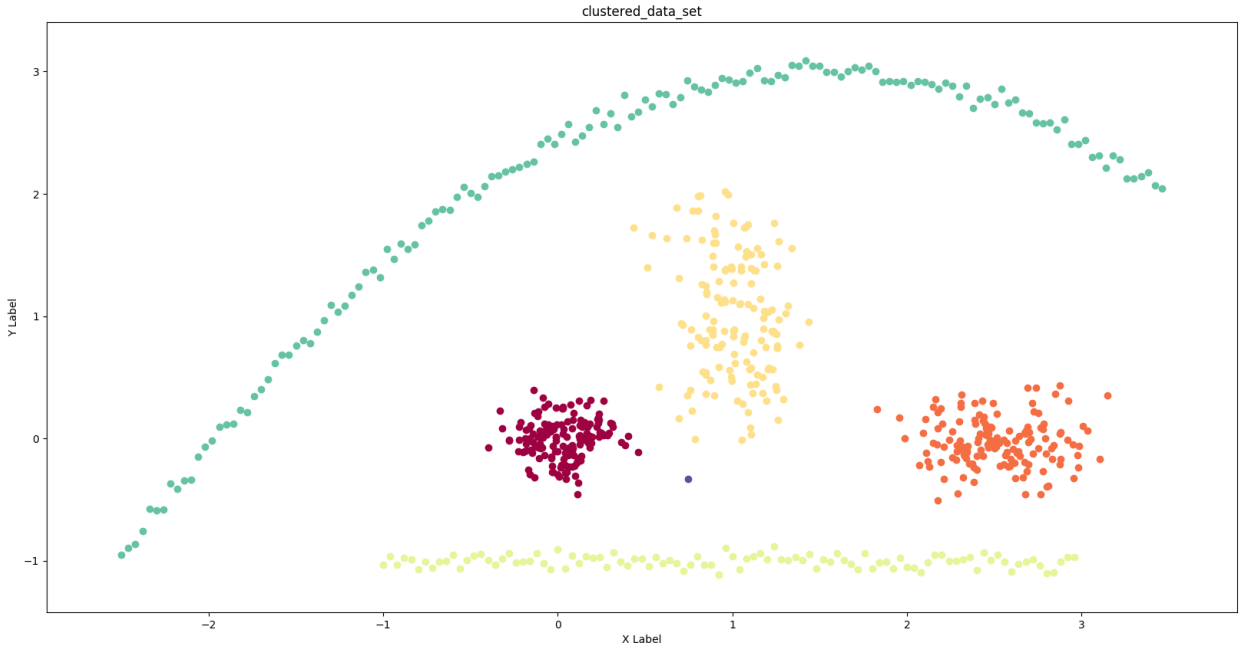


Рисунок 3 - Кластеризованные данные методом DBSCAN на Python и Go

Другой алгоритм кластеризации, который был рассмотрен – Kmeans.

Алгоритм {2} KMeans.

1. Вход: X – набор точек, k – количество кластеров;
2. Выход: $C - k$ центройдов;
3. Инициализировать C случайным образом из выборки X.
4. Пока не сойдётся:
5. $For\ x_i\ in\ X$:
6. Поиск ближайшего центройда c_j: $arg\min_j Dist(x_i, c_j)$, $Dist$ – растояние (пример: $L2$) между x_i и c_j
7. Определение точки x_i к кластеру j.
8. $For\ c_j\ in\ C$:
9. Переопределение центройдов, как среднее арифметическое всех точек x_i принадлежащих кластеру j.
10. Конец

Из результатов, представленных ниже в таблице (таблица 4) можно сделать вывод, что небольшое, по сравнению с другими методами отличие по времени продолжает свидетельствовать о проблеме векторизации в Go. Результаты полученные для Go и Python совпадают (график 3).

lang	method	trainrows	traincols	time, ns
python	kmeans	2499	2	85376787
go	kmeans	2499	2	141433581
cpp	kmeans	2499	2	16683604

Таблица 6 - Результаты KMeans для языков Go, Python, C++

Рисунок 4 Кластеризованные данные xclara методом KMeans на Python и Go.

Несмотря на разочаровывающее отставание по времени для сингулярного (SVD) разложения (табл. 2), метод главных компонент был реализован таким образом, чтобы нивелировать эту проблему – нам удалось добиться увеличения скорости выполнения (табл. 7). Данный метод отбирает главные признаки следующим образом:

Имеем матрицы старых признаков F (размерности $l \times n$) и новых G (размерности $l \times m$, где $m \leq n$). Также введем матрицу U (размерности $n \times m$) линейного преобразования признаков, которая приближает $\tilde{F} = GU^T$ к исходным признакам F. Т.е. требуется найти новые признаки G и преобразования U, решая следующую задачу минимизации:

$$\left\|\tilde{F} - F\right\| = \|GU^T - F\| \to \min_{G,U}, \text{ где выбор } G \text{ и } U \text{ следует из теоремы 1.}$$

Теорема 1.

Если $m \leq rank(F)$, то минимум $\left\|\tilde{F} - F\right\|$ достигается, когда $U -$ это собственные вектора матрицы $F^T F$, соответствующие m максимальным собственным значениям, а матрица $G = FU$.

lang	method	trainrows	traincols	time, ns	scores
go	pca	506	14	697774	0.8045
python	pca	506	14	4770803	0.8045
cpp	pca	506	14	170397	0.8045

Таблица 7 - Результаты PCA для языков Go, Python, C++

Общие характеристики скорости для приведенных методов машинного обучения представлены в таблице 8.

	dbscan	linearregression	pca	logisticregression	kmeans
go	2992.8	27.3	4.1	43.5	8.5
python	1	1	28	1	5.1
cpp	2.6	108.3	1	108.2	1

Таблица 8 - Результаты скорости обучения Go, Python, C++.

Анализ полученных результатов

Результаты, полученные в ходе проведенного исследования, помогли выявить основные потенциальные проблемы для развития Go в сфере машинного обучения, а также понять нам, как разработчикам, на что стоит сделать акценты и какие вопросы проработать внимательнее и качественнее.

Основной проблемой в реализации алгоритмов линейной алгебры является работа с каждым элементом матрицы в один момент времени, что значительно замедляет скорость работы, а учитывая возможности многопоточного решения задач на Go, необходимой доработкой алгоритмов является векторизация вычислений.

В реализации алгоритмов машинного обучения худшие результаты показал DBSCAN, так как кластеризация по этому алгоритму подразумевает нахождение расстояний между всеми элементами в указанной окрестности, что ведет к обходу всех элементов из каждого элемента, т.е. наибольшей сложности O(n^2), однако алгоритм можно значительно оптимизировать с использованием k-d и ball деревьев, однако это ведет к квадратичному увеличению расхода памяти. Алгоритм будет доработан и оптимизирован.

Остальные алгоритмы показали хорошие результаты, но над оптимизацией логистической регрессии, добавлением SAG (стохастического среднего градиента), и оптимизацией линейной регрессии путем улучшения логики алгоритмов линейной алгебры работы далее запланированы.

Выводы

Go как инструмент для машинного обучения

Несмотря на отставание в скорости работы от C/C++ и Python, мы пришли к выводу, основываясь на полученных данных, опыте проектирования моделей машинного обучения и использования готовых библиотек на всех трех языках, что Go – ЯП отлично подходящий для машинного обучения благодаря всем перечисленным в главе «Язык программирования Go» плюсам:

1. Быстрая компиляция в машинный код –
   Разрабатывая код для тестирования на mlpack этот пункт был критичным, так как малейшее изменения в коде приводило к необходимости компилировать его заново, что занимало от 10 до 30 секунд;
2. Простой синтаксис –
   Несмотря на строгую типизацию, отсутствие предопределенных аргументов функций и try/except блоков, синтаксис Go простой, лаконичный, легкий для освоения и программирования, а новая концепции ООП, основанная на структурах и определенных для них методов, а также тип интерфейсов, дают возможность посмотреть на привычные задачи под другим углом и проектировать систему на новых абстракциях и сущностях;
3. Строгая типизация –
   Строгая типизация = увеличение скорости и повышение уровня надежности систем, что наиболее актуально в задачах машинного обучения;

4. Быстрая скорость работы –

   Go нельзя назвать самым быстрым языком, однако во многих алгоритмах его скорость была на уровне C/C++ и NumPy;
5. Сборщик мусора;
6. Отсутствие “магии” –

   Действительно важный плюс, так как многие методы в работе, как поиск минимального элемента в списке или удаление элемента из списка, приходится реализовывать в виде функций самостоятельно, но каждая такая реализация заставляет разработчика задуматься об оптимальности принятого им решения;
7. WebAssembly;
8. Goroutines –

   Как потенциально отличный способ векторизации вычислений.

Основная проблема, с которой придется столкнуться при реализации алгоритмов машинного обучения на Go – это достаточно слабая базовая библиотека матриц gonum, однако ее реализацию всегда можно дополнить и улучшить, так как она написана на чистом Go.

Траектория развития нашей библиотеки

На данный момент в библиотеке реализованы пять алгоритмов:

- Supervised learning:
  - Linear regression;
  - Logistic regression;
- Unsupervised learning:
  - K-means (clustering);
  - DBSCAN (clustering);
- Data processing:
  - PCA (decomposition).

Кроме того, исходный код всего нашего исследования и тестирования выложен в открытый доступ на github (https://github.com/bayesiangopher/bayesiangopher).

Идею развития Go как нового языка для машинного обучения мы видим перспективной, поэтому библиотека будет развивать уже не в рамках магистерской диссертации.

В дальнейшем планируется реализовать следующие алгоритмы, наработки по которым уже имеются:

- KNN (k-nearest neighbors);
- Lasso regression;
- Ridge regression;
- ElasticNet regression;
- GPR (Gaussian Process Regression);
- t-SNE;
- ID3 tree decision algorithm;
- C4.5 tree decision algorithm.

Архитектура API будет и в дальнейшим развиваться подобно scikit-

learn, так как ее реализация нам кажется наиболее удобной для использования.

Кроме того, для улучшения скорости работы и уменьшения ошибок в результатах, мы планируем разработку пакета helpers с функционалом методов оптимизации, структур данных и более тонкой реализацией алгоритмов линейной алгебры, основанной на векторизации вычислений с применением goroutines, а также пакета core для работы с разными форматами данных, их предобработки и построения графиков.

Заключение

В ходе работы были исследованы библиотеки gonum, numpy, armadillo, scikit-learn, mlpack использующиеся в решении задач, связанных с машинным обучением. Целью исследования была сравнительная оценка скорости работы базовых алгоритмов линейной алгебры, реализованных в gonum и python/C/C++ библиотеках, а также создания первой итерации библиотеки для машинного обучения bayesiangopher на Go и сравнение скорости обучения, ошибок полученных результатов с результатами mlpack и scikit-learn.

Полученные результаты однозначно подтверждают целесообразность реализации и развития решений на Go для машинного обучения в силу относительной быстроты языка, скорости компиляции, простоты синтаксиса и растущей популярности ЯП Go.

Основной проблемой, выявленной в ходе работы, является недостаточно грамотная реализация матричного функционала библиотеки gonum, а именно долгое создание новых структур-матриц и медленная скорость работы алгоритмов разложения (SVD, Cholesky decomposition, QR) и нахождения собственных значений и векторов матрицы, которые лежат в основе многих алгоритмов машинного обучения. Эту проблему, по нашей оценке, можно решить введением многопоточной векторизации вычислений, которая возможна на Go в его стандартном дистрибутиве.

Список используемой литературы и источников:

[1] https://github.com/gonum ;

[2] http://arma.sourceforge.net ;

[3] https://www.numpy.org ;

[4] https://www.mlpack.org ;

[5] https://scikit-learn.org/stable/ ;

[6] Clancey, J. W. & Shortliffe, E. H. *Readings in Medical Artificial Intelligence: The First Decade* Ch. 1 (Addison Wesley, 1984);

[7] Fjell, C. D. et al. *J. Med. Chem.* 52, 2006–2015 (2009);

[8] Olden, J. D., Lawler, J. J. & Poff, N. L. Machine learning methods without tears: A primer for ecologists. Q. Rev. Biol. 83, 171–193 (2008);

[9] Pierson, E., Simoiu, C., Overgoor, J., Corbett-Davies, S., Ramachandran, V., Phillips, C., and Goel, S. (2017). "A large-scale Analysis of Racial Disparities in Police Stops across the United States." arXiv preprint arXiv:1706.05678;

[10] C. A. Hidalgo, B. Klinger, A.-L. Barab´asi, R. Hausmann. "The Product Space Condi- tions the Development of Nations." Science 317.5837 (2007): 482-487;

[11] https://www.kaggle.com/kaggle/kaggle-survey-2018 ;

[12] Язык программирования Go. : Пер. с англ. – М. : ООО "И.Д. Вильямс", 2018. – 432 с. : ил. – Парал. тит. англ. ISBN 978-5-8459-2051-5 (рус.);

[13] https://golang.org/doc/faq ;

[14] https://github.com/golang/go/wiki/WebAssembly ;

[15] https://golang.org/cmd/gofmt/ ;

[16] https://habr.com/ru/company/mailru/blog/446914/ ;

[17] https://github.com/tensorflow/swift/blob/master/docs/WhySwiftForTensorFlow.md ;

[18] https://yandex.ru/company/researches/2019/it-jobs ;

[19] https://www.tensorflow.org ;

[20] https://keras.io ;

[21] https://caffe.berkeleyvision.org ;

[22] https://pytorch.org ;

[23] https://xgboost.readthedocs.io/en/latest/ ;

[24] https://godoc.org/github.com/mumax/3/cuda ;

[25] https://godoc.org/gorgonia.org/cu ;

[26] https://godoc.org/google.golang.org/api/tpu/v1alpha1 ;

[27] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html ;

[28] https://github.com/sjwhitworth/golearn ;

[29] http://www.netlib.org/blas/ ;

[30] http://www.netlib.org/lapack/ ;

[31] https://www.kaggle.com/smid80/weatherww2 ;

[32] https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Original) ;

[33] https://www.kaggle.com/hdriss/xclara ;

[34] https://www.kaggle.com/c/boston-housing ;

[35] http://www.ccs.neu.edu/home/vip/teach/DMcourse/2_cluster_EM_mixt/notes_slides/revisitofrevisitDBSCAN.pdf ;

Приложение А

Характеристики компьютера.

OS: Ubuntu 18.04;

CPU: см. рис. А1;

RAM: см. рис. А2;

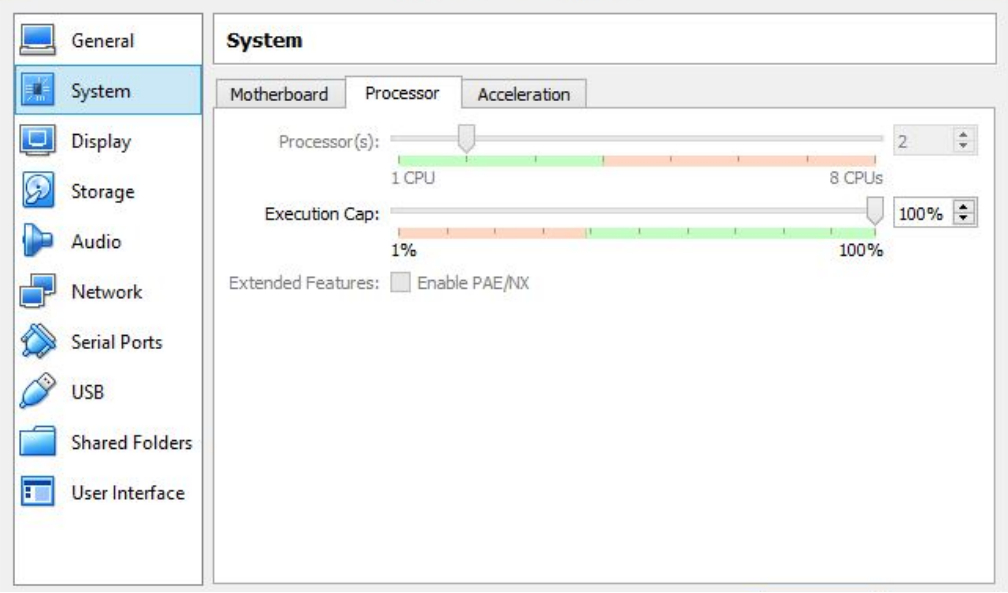


Рисунок А1 - Характеристики выделенного ресурса CPU для виртуальной машины

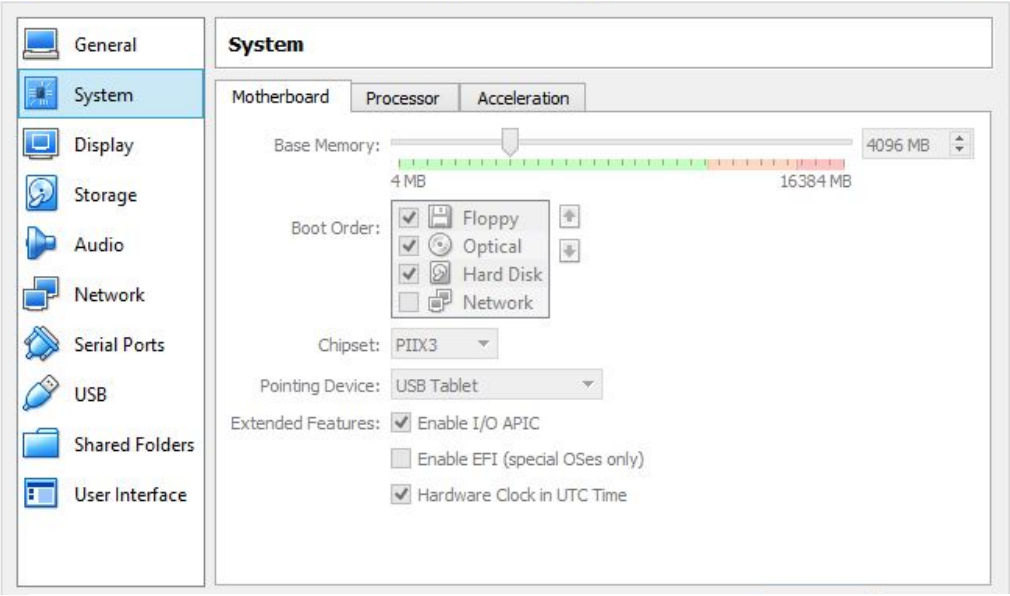
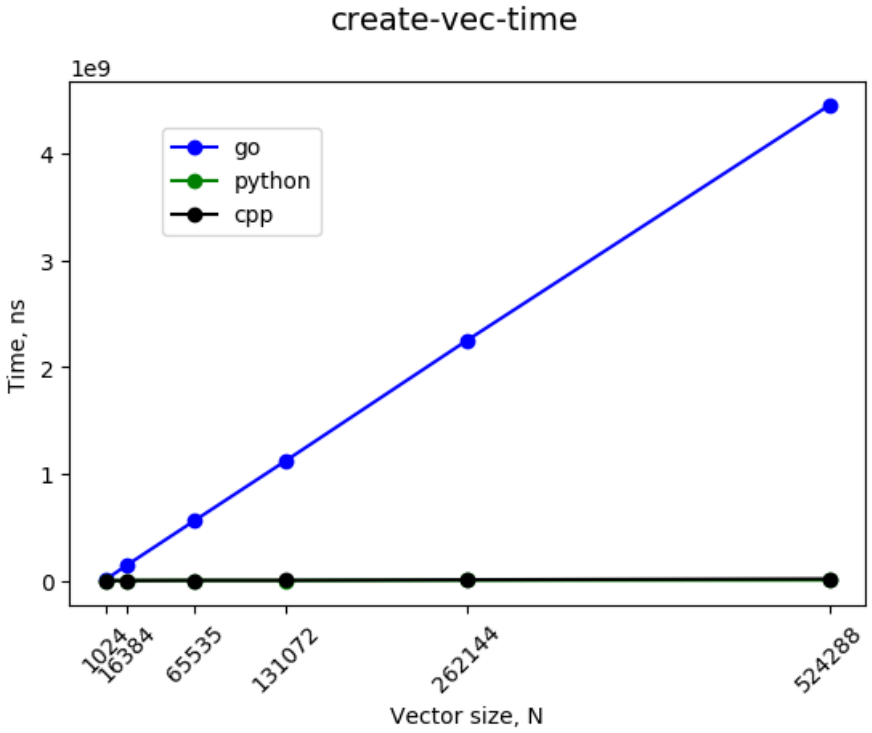


Рисунок А2 - Характеристики выделенной RAM для виртуальной машины

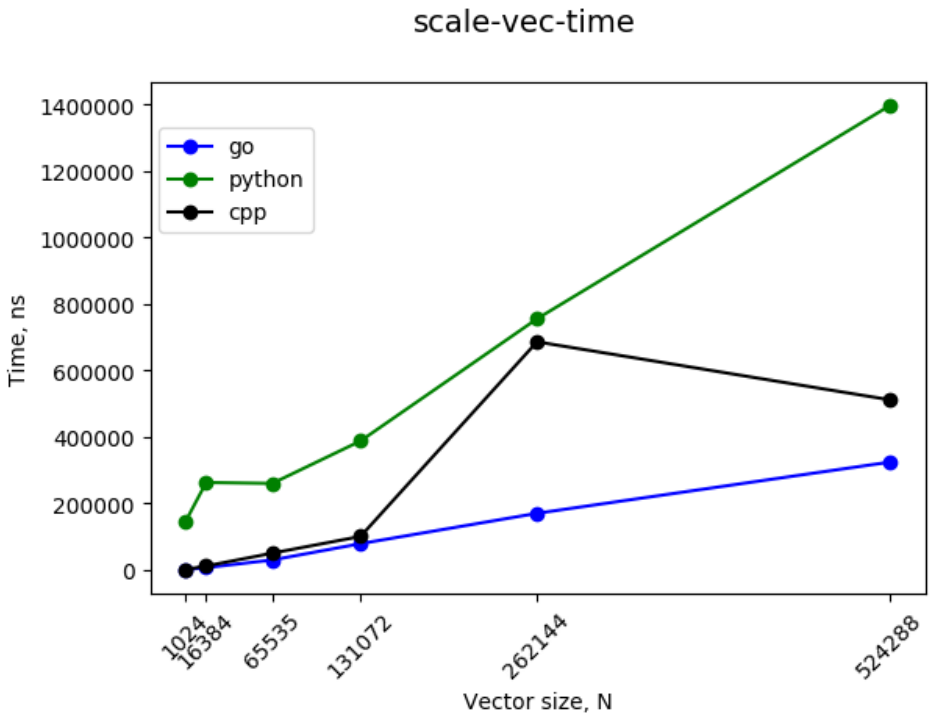
Приложение Б

Вектора:



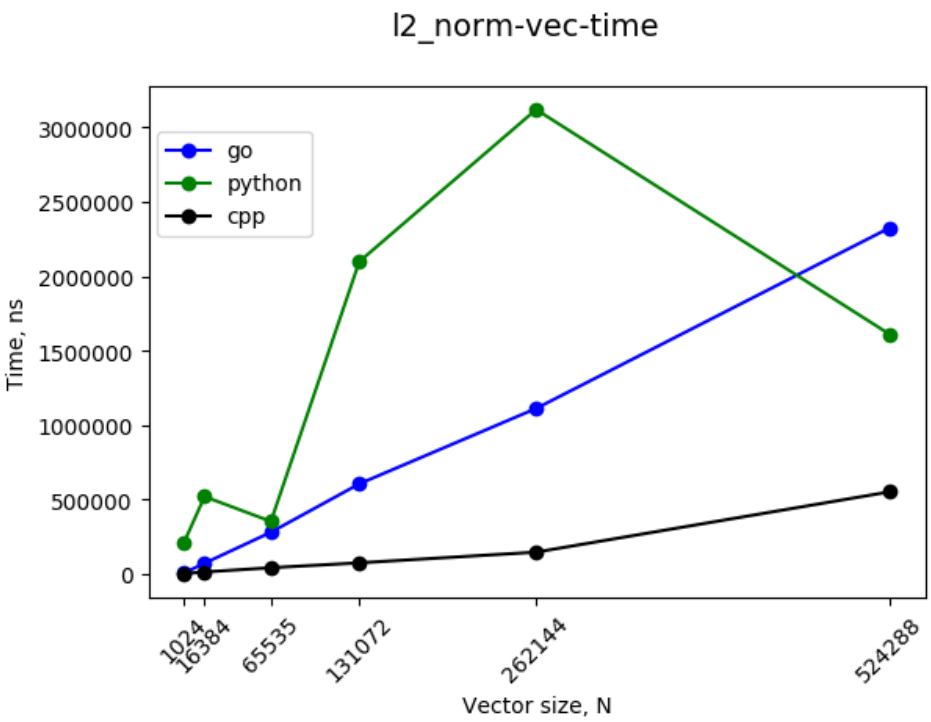
lang	shape	time, ns
go	1024	8661255
go	16384	1,44E+08
go	65535	5,63E+08
go	131072	1,12E+09
go	262144	2,25E+09
go	524288	4,45E+09
python	1024	911903
python	16384	1581478
python	65535	2843141
python	131072	1750493
python	262144	3292870
python	524288	6321454
cpp	1024	106617
cpp	16384	983036
cpp	65535	2859410
cpp	131072	3540560
cpp	262144	8771150
cpp	524288	16888200

Рисунок Б1 - результаты скорости работы функции по созданию векторов float64



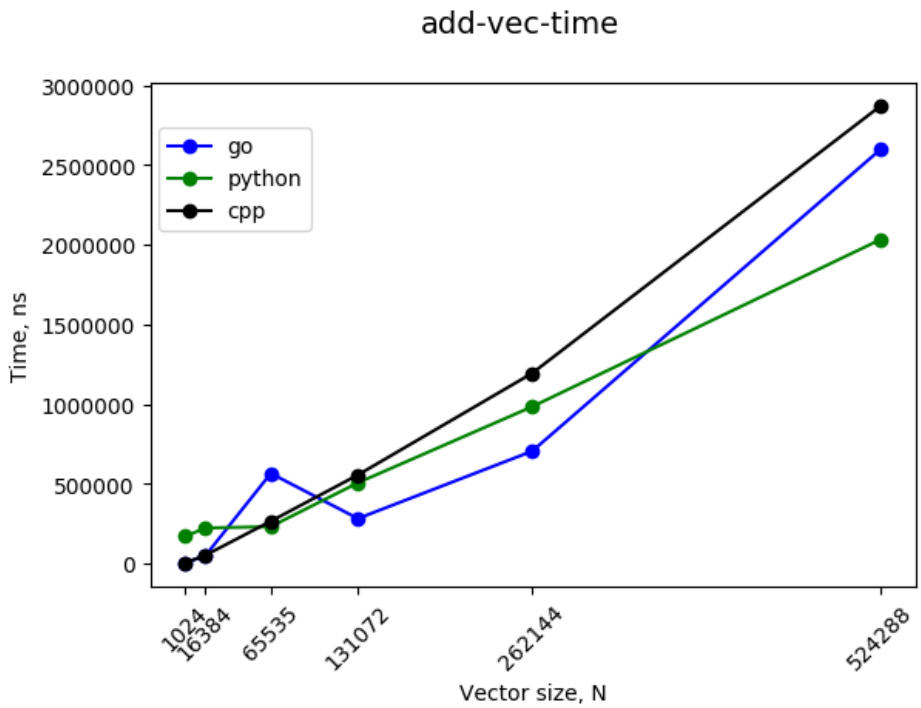
lang	shape	time, ns
go	1024	614
go	16384	6342
go	65535	29346
go	131072	78432
go	262144	169732
go	524288	323776
python	1024	144553
python	16384	262427
python	65535	259637
python	131072	387382
python	262144	754666
python	524288	1396131
cpp	1024	556
cpp	16384	11224
cpp	65535	49889
cpp	131072	100246
cpp	262144	685471
cpp	524288	511044

Рисунок Б2 - результаты скорости работы функции по масштабированию векторов float64



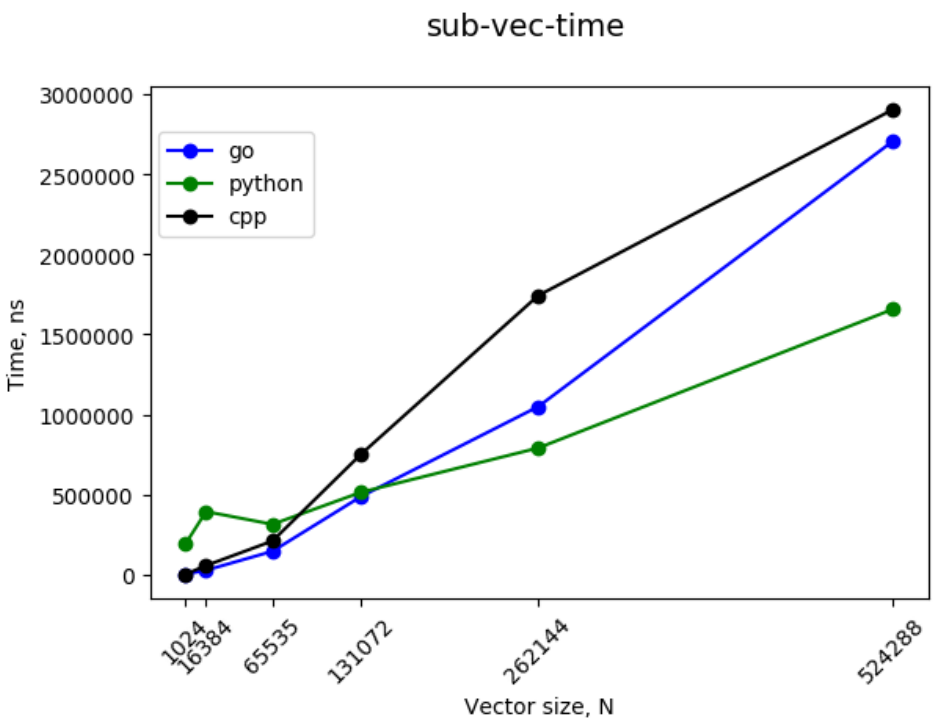
lang	shape	time, ns
go	1024	4982
go	16384	70843
go	65535	278969
go	131072	604497
go	262144	1111401
go	524288	2326339
python	1024	208377
python	16384	518894
python	65535	351643
python	131072	2098751
python	262144	3121113
python	524288	1609253
cpp	1024	1197
cpp	16384	11900
cpp	65535	40440
cpp	131072	73205
cpp	262144	145151
cpp	524288	550829

Рисунок Б3- результаты скорости работы функции по нахождению Евклидовой нормы векторов float64



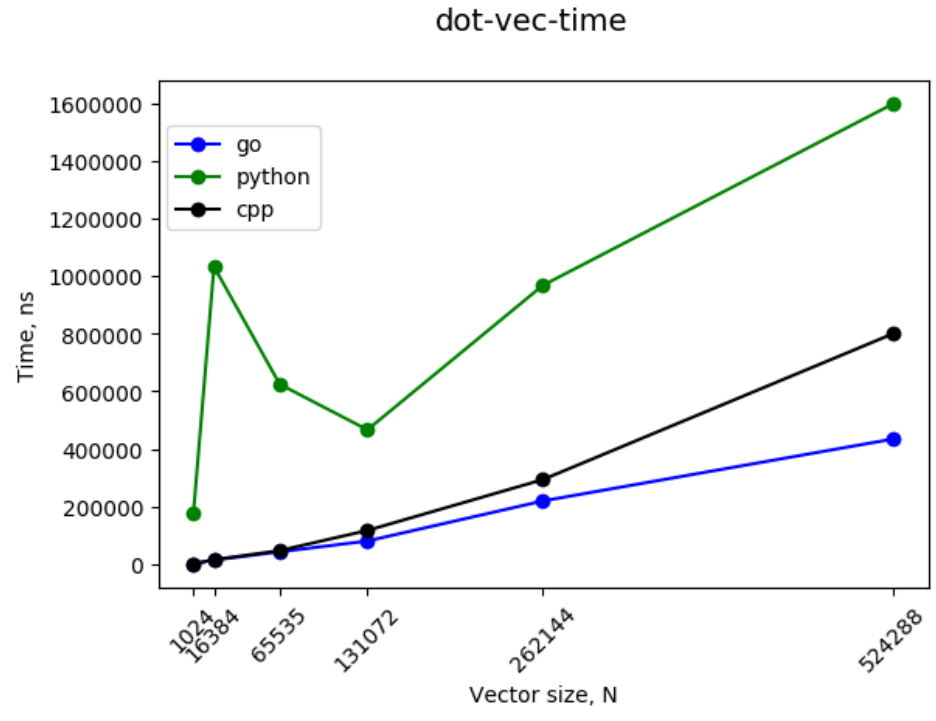
lang	shape	time, ns
go	1024	2760
go	16384	48647
go	65535	564649
go	131072	281028
go	262144	703934
go	524288	2598112
python	1024	169062
python	16384	221800
python	65535	230574
python	131072	505375
python	262144	983142
python	524288	2032232
cpp	1024	2576
cpp	16384	50905
cpp	65535	264428
cpp	131072	554781
cpp	262144	1192909
cpp	524288	2870815

Рисунок Б4 - результаты скорости работы функции по нахождению суммы векторов float64



lang	shape	time, ns
go	1024	2838
go	16384	27760
go	65535	146319
go	131072	488314
go	262144	1049547
go	524288	2705361
python	1024	190114
python	16384	394225
python	65535	314712
python	131072	514912
python	262144	791335
python	524288	1656842
cpp	1024	2329
cpp	16384	57070
cpp	65535	210613
cpp	131072	752619
cpp	262144	1742937
cpp	524288	2902931

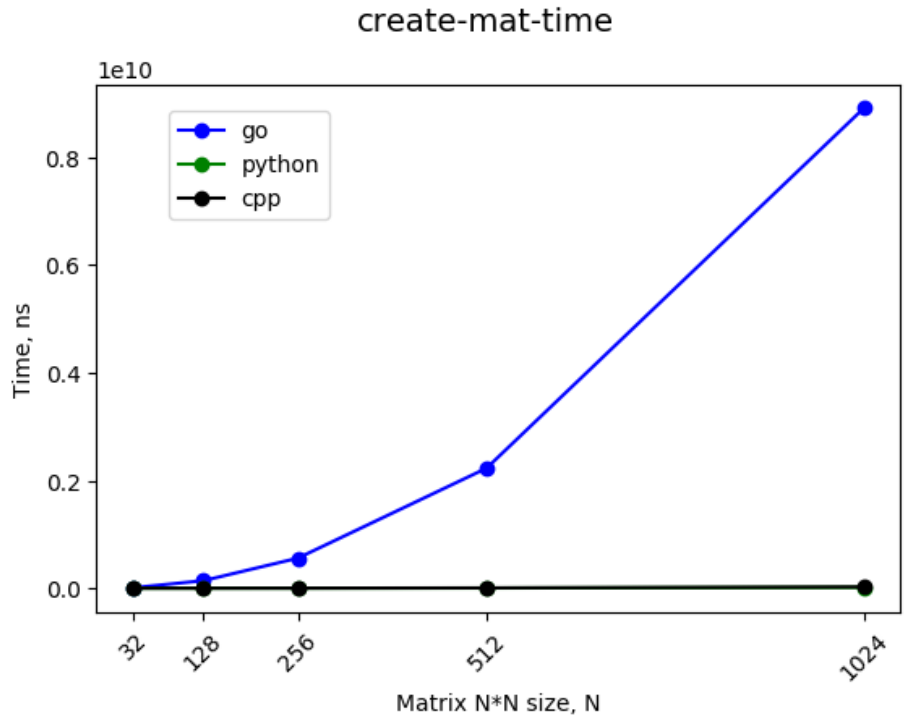
Рисунок Б5- результаты скорости работы функции по нахождению разницы векторов float64



lang	shape	time, ns
go	1024	1141
go	16384	14090
go	65535	42666
go	131072	80137
go	262144	219202
go	524288	434344
python	1024	177240
python	16384	1032423
python	65535	624251
python	131072	466299
python	262144	967383
python	524288	1598477
cpp	1024	854
cpp	16384	15240
cpp	65535	46951
cpp	131072	116821
cpp	262144	293242
cpp	524288	800373

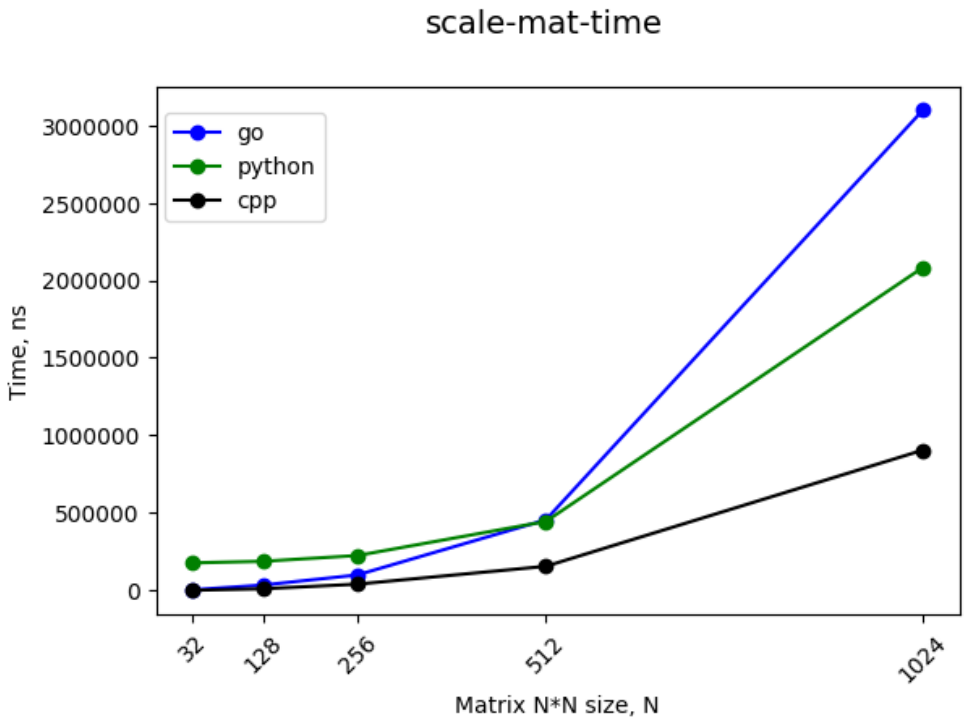
Рисунок Б6- результаты скорости работы функции по нахождению произведения векторов float64

Матрицы:



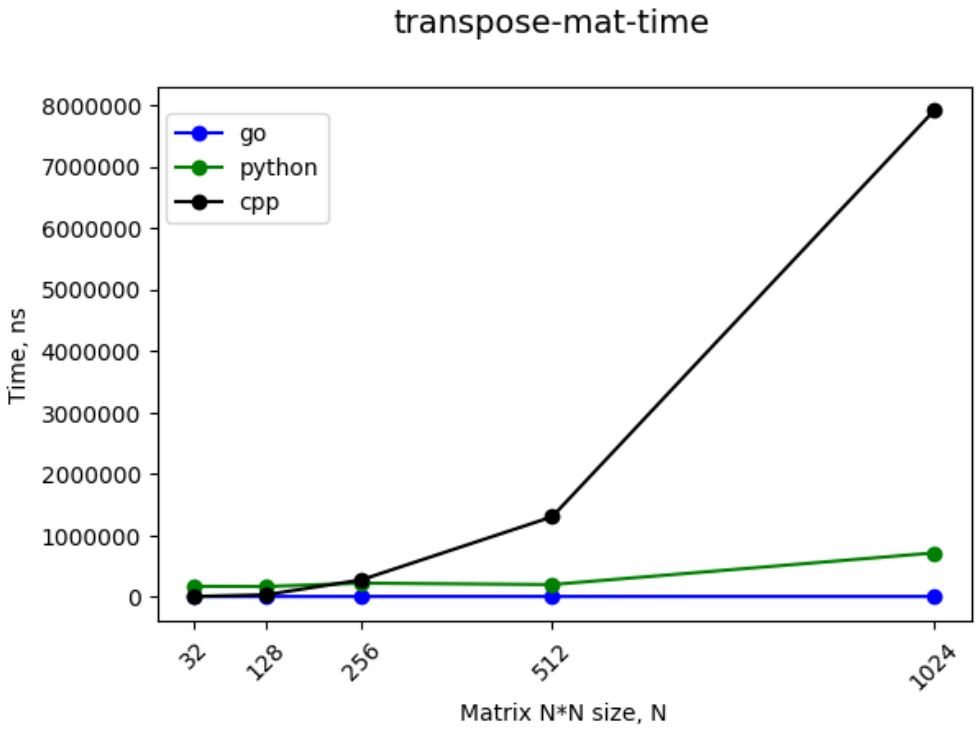
lang	shape	time, ns
go	32	8698334
go	128	1,41E+08
go	256	5,6E+08
go	512	2,23E+09
go	1024	8,92E+09
python	32	214028
python	128	455355
python	256	1159048
python	512	3833818
python	1024	15602135
cpp	32	26568
cpp	128	399149
cpp	256	1521470
cpp	512	6183400
cpp	1024	25004800

Рисунок Б9- результаты скорости работы функции по созданию матриц float64



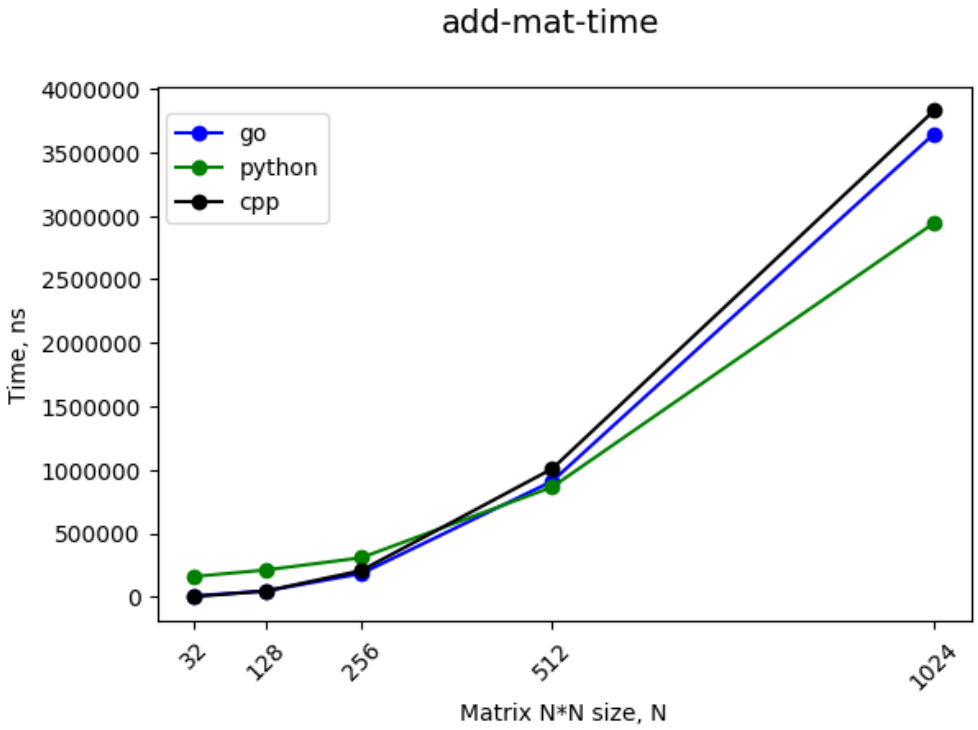
lang	shape	time, ns
go	32	3017
go	128	34093
go	256	99194
go	512	454819
go	1024	3097242
python	32	177073
python	128	187206
python	256	224328
python	512	444388
python	1024	2082395
cpp	32	463
cpp	128	9969
cpp	256	39775
cpp	512	155012
cpp	1024	904372

Рисунок Б8- результаты скорости работы функции по масштабированию матриц float64



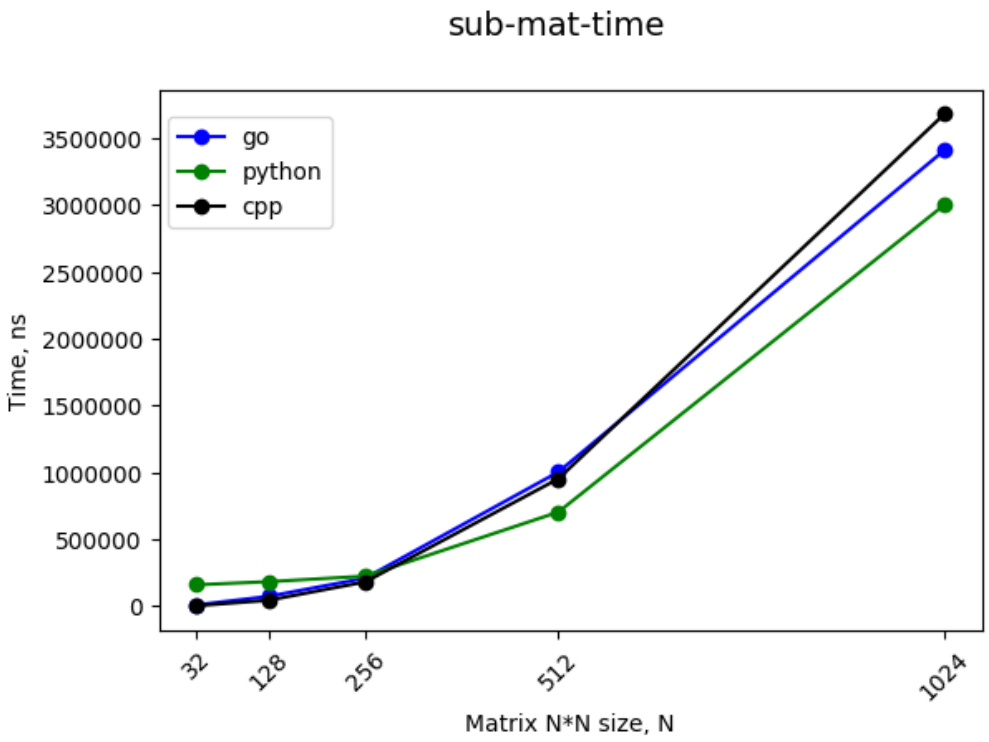
lang	shape	time, ns
go	32	127
go	128	127
go	256	97
go	512	141
go	1024	137
python	32	165939
python	128	162839
python	256	217533
python	512	194549
python	1024	709581
cpp	32	422
cpp	128	27857
cpp	256	270404
cpp	512	1298130
cpp	1024	7908280

Рисунок Б9- результаты скорости работы функции по транспонированию матриц float64



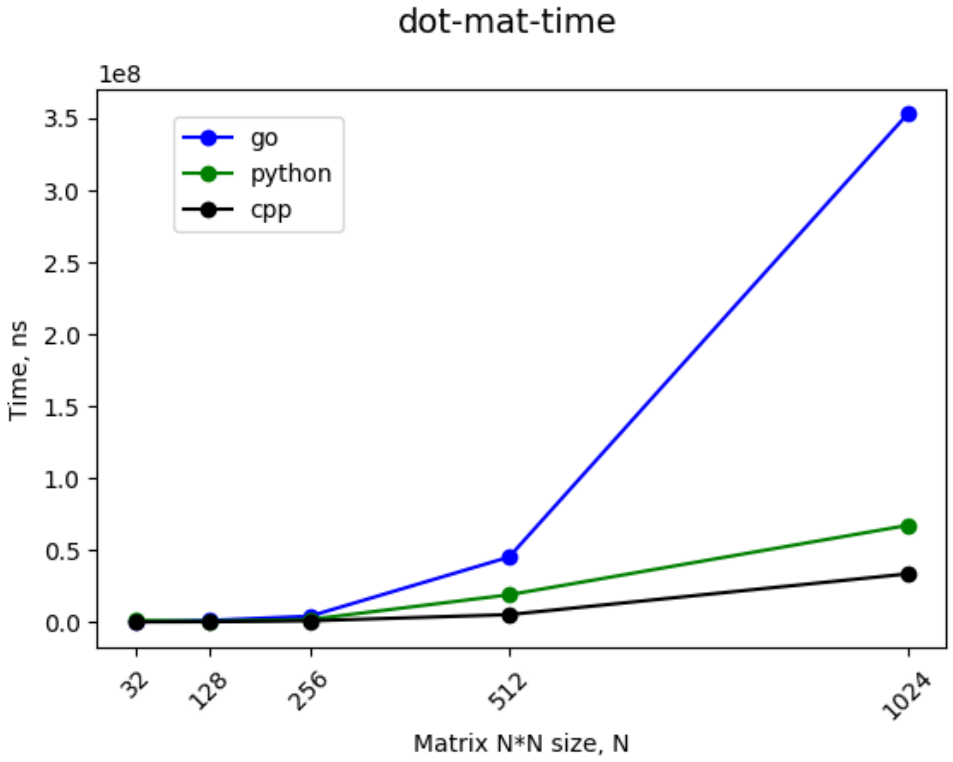
lang	shape	time, ns
go	32	6539
go	128	45388
go	256	182354
go	512	909578
go	1024	3646055
python	32	159573
python	128	210762
python	256	306940
python	512	863981
python	1024	2943491
cpp	32	1941
cpp	128	44632
cpp	256	206242
cpp	512	1007800
cpp	1024	3828440

Рисунок Б10- результаты скорости работы функции по сложению матриц float64



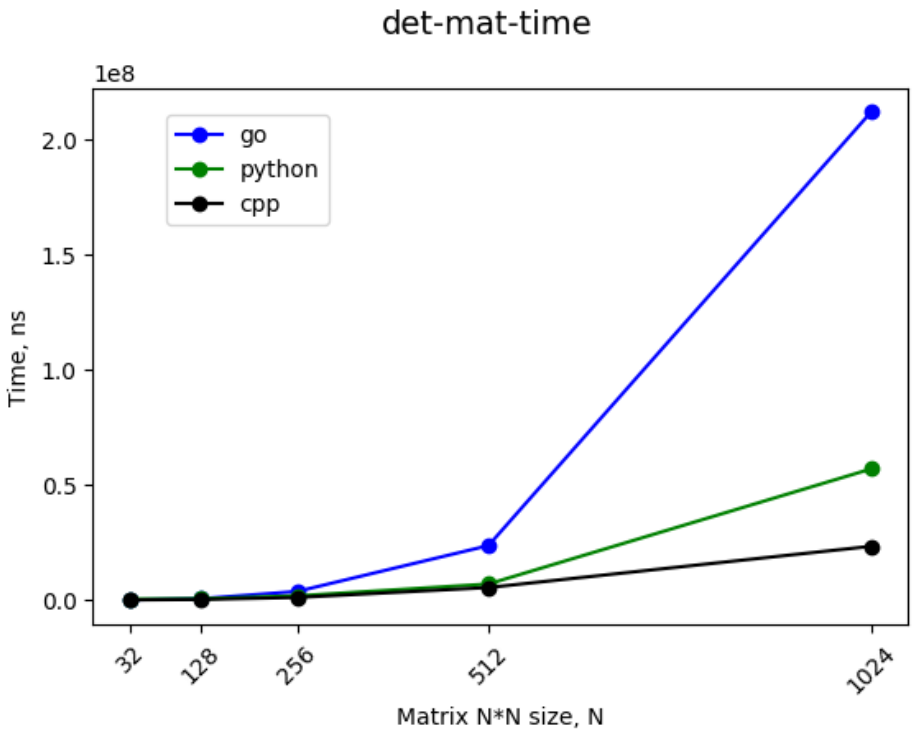
lang	shape	time, ns
go	32	7991
go	128	72269
go	256	208673
go	512	1003075
go	1024	3412210
python	32	158929
python	128	182247
python	256	223660
python	512	703263
python	1024	3003048
cpp	32	1954
cpp	128	43254
cpp	256	180560
cpp	512	952769
cpp	1024	3682880

Рисунок Б11- результаты скорости работы функции по вычитанию матриц float64



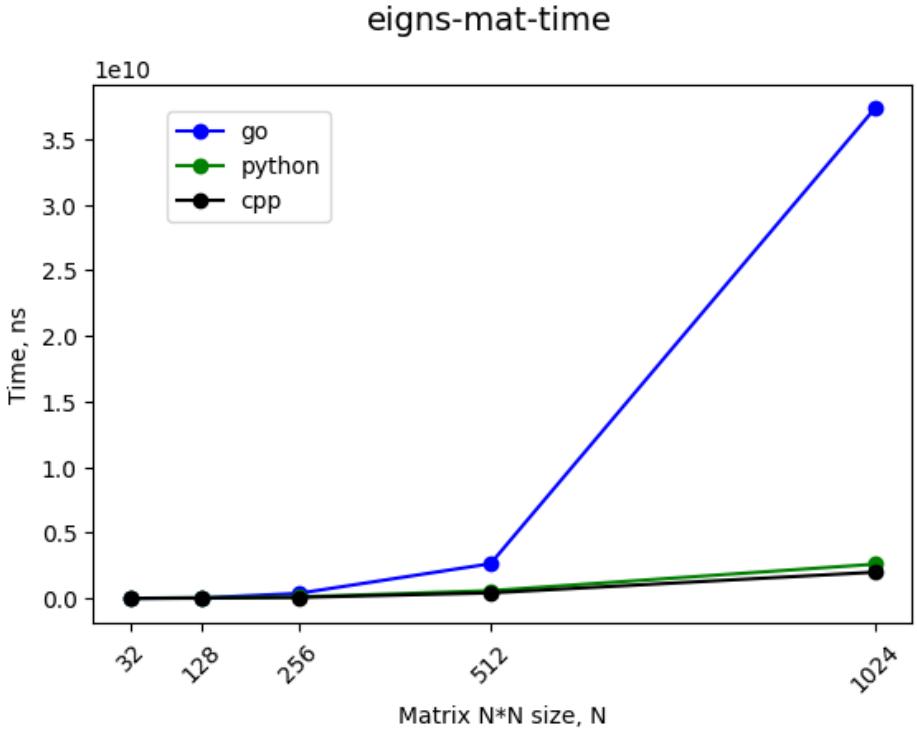
lang	shape	time, ns
go	32	25466
go	128	1066183
go	256	3977946
go	512	45035437
go	1024	3,53E+08
python	32	1090574
python	128	532579
python	256	1688671
python	512	18842649
python	1024	67030549
cpp	32	10130
cpp	128	154804
cpp	256	760255
cpp	512	5067860
cpp	1024	33359500

Рисунок Б12- результаты скорости работы функции по перемножению матриц float64



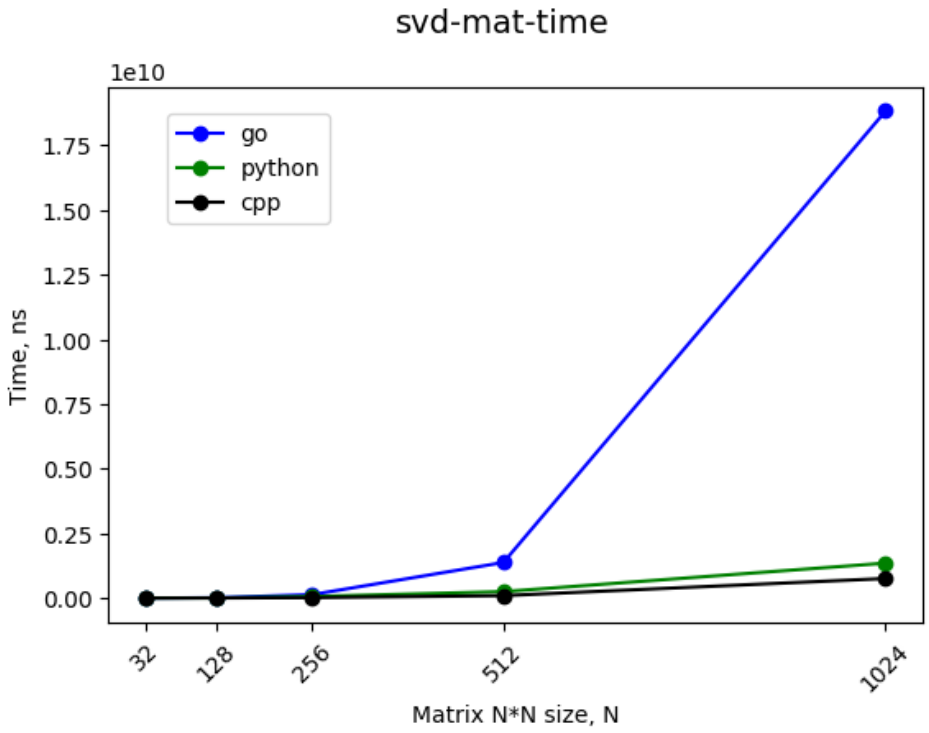
lang	shape	time, ns
go	32	32538
go	128	594995
go	256	3692489
go	512	23619786
go	1024	2,12E+08
python	32	400805
python	128	560355
python	256	1792812
python	512	6930208
python	1024	56903028
cpp	32	21214
cpp	128	231320
cpp	256	1085460
cpp	512	5395480
cpp	1024	23310400

Рисунок Б13- результаты скорости работы функции по нахождению определителя матриц float64



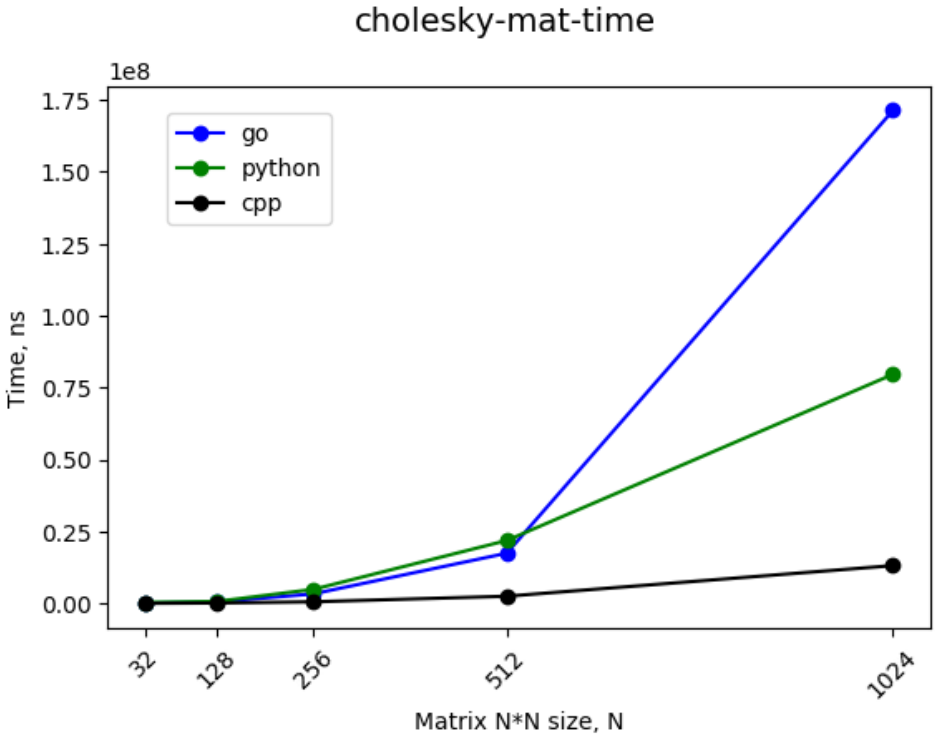
lang	shape	time, ns
go	32	571472
go	128	34075002
go	256	3,77E+08
go	512	2,64E+09
go	1024	3,73E+10
python	32	6821918
python	128	49999570
python	256	1,07E+08
python	512	5,82E+08
python	1024	2,6E+09
cpp	32	394971
cpp	128	14624700
cpp	256	72975200
cpp	512	4,14E+08
cpp	1024	1,99E+09

Рисунок Б14- результаты скорости работы функции по нахождению собственных значений и собственных векторов матриц float64



lang	shape	time, ns
go	32	530146
go	128	15096694
go	256	1,36E+08
go	512	1,38E+09
go	1024	1,88E+10
python	32	1977300
python	128	10826873
python	256	61424612
python	512	2,51E+08
python	1024	1,35E+09
cpp	32	546553
cpp	128	5283880
cpp	256	19740200
cpp	512	96201800
cpp	1024	7,59E+08

Рисунок Б15- результаты скорости работы функции по нахождению сингулярного разложения матриц float64



lang	shape	time, ns
go	32	47247
go	128	482698
go	256	3313047
go	512	17441307
go	1024	1,71E+08
python	32	344562
python	128	693964
python	256	4888033
python	512	21859121
python	1024	79603600
cpp	32	8365
cpp	128	133332
cpp	256	553182
cpp	512	2485000
cpp	1024	13130000

Рисунок Б16- результаты скорости работы функции по нахождению разложения Холецкого матриц float64