**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**

**Язык программирования Go**

**Существующие инструменты машинного обучения**

**Python**

**C++**

**Go**

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

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

Анализ полученных результатов

Задачи линейной алгебры

Задачи машинного обучения

Выводы

Go как инструмент для машинного обучения

Траектория развития нашей библиотеки

Заключение

Список литературы

Приложения

**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.

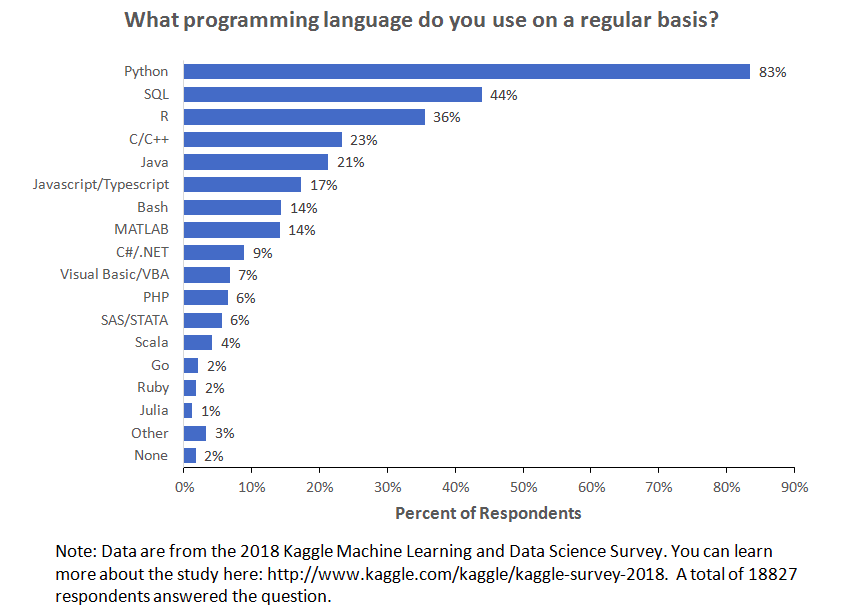


Рисунок 1 - Данные о результатах самых популярных языках с конкурса Kaggle 2018

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, С++ 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. Предпоследняя 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 списке, 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].

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

Основную проблему ЯП Go можно заметить уже сейчас, однако, 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 нейронных сетей;

- 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, 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 в реализации 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()

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, происходит вызов С 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 х 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++**

С/С++ 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. Поэтому миру машинного обучения Go необходимы энтузиасты для создания и развития сообщества.

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

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:
   1. Создание вектора случайных чисел [0, 100) (create);
   2. Масштабирования вектора (scale);
   3. Определение Евклидовой нормы вектора (l2\_norm);
   4. Сумма векторов (add);
   5. Разница векторов (sub);
   6. Произведение векторов (dot).
2. Операции с матрицами для размерностей 32х32, 128х128, 256х256, 512х512, 1024х1024:
   1. Создание матрицы случайных числе [0, 100) (create);
   2. Масштабирование матрицы (scale);
   3. Транспонирование матрицы (transpose);
   4. Сумма матрицы (add);
   5. Разница матрицы (sub);
   6. Произведение матрицы (dot);
   7. Нахождение определителя матрицы (det);
   8. Нахождение собственных значений и собственных векторов матрицы (eigens);
   9. Сингулярное разложение матрицы (svd);
   10. Разложение Холецкого матрицы (cholesky).

Итоговые результаты для векторов и матрицы наибольшего размера

(524288 и 1024х1024 соответственно) представлены в таблицах 1 и 2. Динамика скорости работы решения каждой задачи для каждой размерности представлена на графиках в Приложении Б.

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



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

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

Основные выводы и анализ результатов буду приведены в главе «Анализ полученных результатов».

**Список литературы**

[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> ;

**Приложение А**

Характеристики компьютера.

OS: Ubuntu 18.04;

CPU: см. рис. А1;

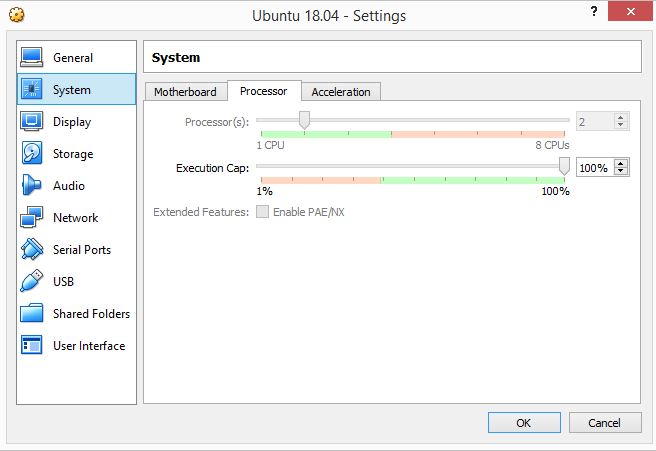
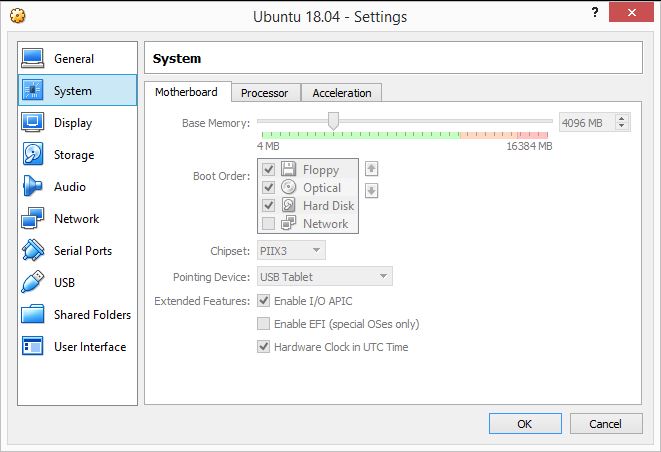
RAM: см. рис. А2;****

Рисунок A1 - Характеристики выделенного ресурса CPU для виртуальной машины

Рисунок A2 - Характеристики выделенной RAM для виртуальной машины

**Приложение Б**

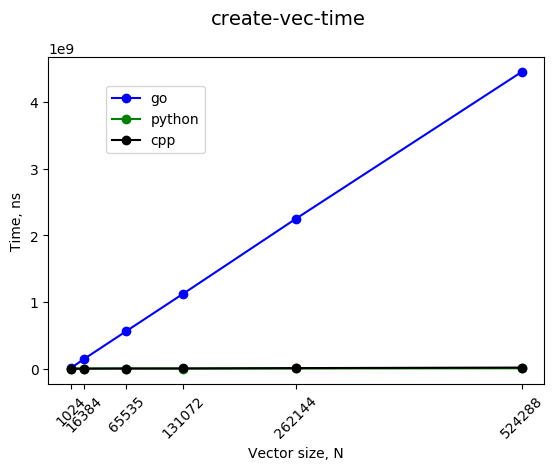




Рисунок Б1 - результаты скорости работы функции по созданию векторов float64

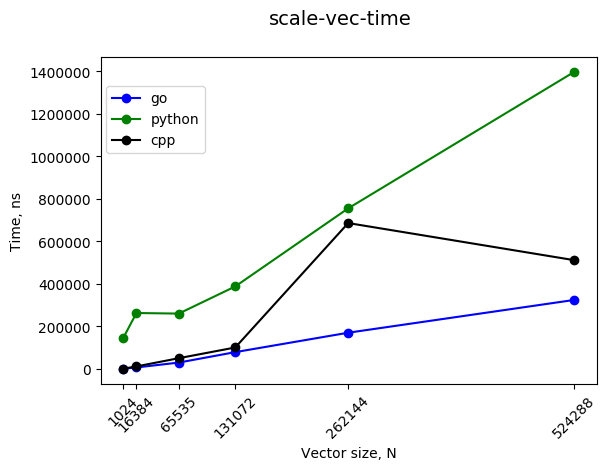


Рисунок Б2 - результаты скорости работы функции по масштабированию векторов float64

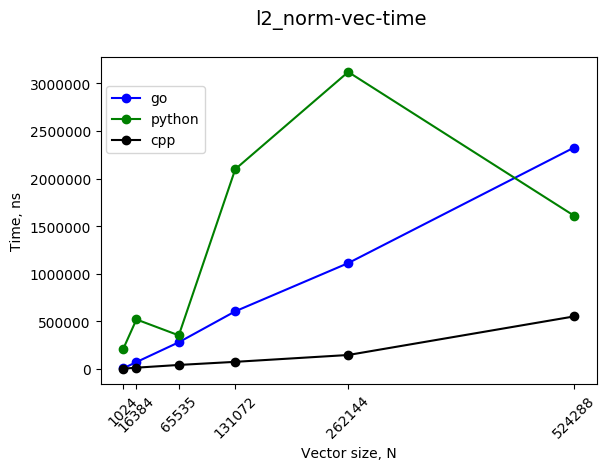


Рисунок Б3- результаты скорости работы функции по нахождению Евклидовой нормы векторов float64



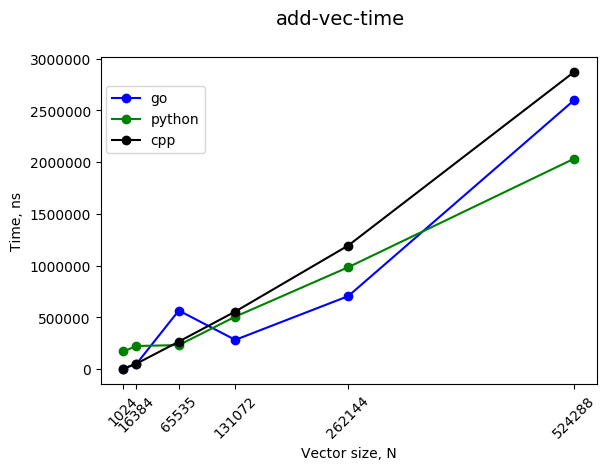


Рисунок Б4 - результаты скорости работы функции по нахождению суммы векторов float64

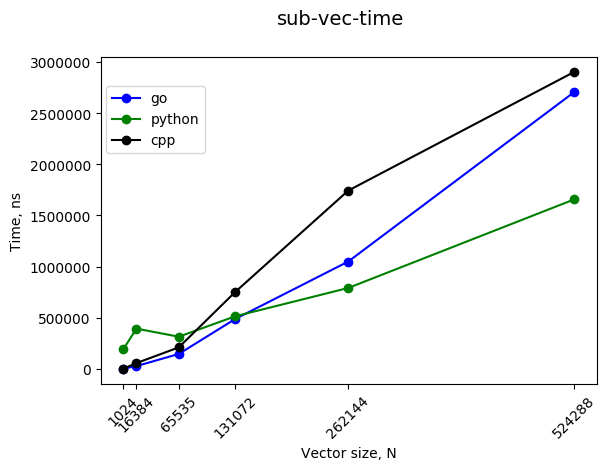


Рисунок Б5- результаты скорости работы функции по нахождению разницы векторов float64

