Universidade Federal do Paraná
Setor de Ciências Exatas
Departamento de Estatística
Programa de Especialização em *Data Science* e *Big Data*

Lucca Portes Cavalheiro

Analysis on Python Performance for Data Stream Mining

Curitiba
2020

Lucca Portes Cavalheiro

Analysis on Python Performance for Data Stream Mining

Monografia apresentada ao Programa de Especialização em Data Science e Big Data da Universidade Federal do Paraná como requisito parcial para a obtenção do grau de especialista.

Orientador: Marco Antônio Zanata Alves

Curitiba
2020

Analysis on Python Performance for Data Stream Mining

Lucca Portes Cavalheiro[1]
Marco Antônio Zanata Alves[2]
Jean Paul Barddal[3]

Abstract

Data stream mining is an essential task in today's scientific community. The most famous library for performing such a task in Python, Scikit-Multiflow, presents a severe performance problem, when compared to the library it was inspired on, MOA, written in Java. Python is an easy to use programming language, and its libraries implemented improves the user experience, however, with a performance cost. With the right tools, Python libraries can present performance comparable to low-level languages such as C/C++. This work performs a comparison of the implementation of methods from Scikit-Multiflow, with new implementations in low-level languages with a binding to Python. The results showed a significant improvement in the original performance of the library, while keeping the predictions and prediction results intact.
Keywords: High-performance Computing, Python, Data Stream Mining.

Resumo

A mineração de fluxos de dados é uma tarefa essencial na comunidade científica de hoje. A biblioteca mais famosa por executar tal tarefa em Python, Scikit-Multiflow, apresenta um grave problema de desempenho, em comparação com a biblioteca na qual foi inspirada, MOA, em Java. Python é uma linguagem de programação fácil de usar e suas bibliotecas melhoram a experiência do usuário, no entanto, com um custo para o desempenho. Com as ferramentas certas, as bibliotecas Python podem apresentar desempenho comparável a linguagens de baixo nível, como C / C ++. Este trabalho tem como objetivo realizar uma comparação da implementação de métodos do Scikit-Multiflow, com novas implementações em linguagens de baixo nível com uma interface em Python. Os resultados mostraram uma melhora significativa no desempenho original da biblioteca.
Palavras-chave: *Computação de Alto Desempenho, Python, Mineração de Fluxo de Dados.*

[1]Student from the Data Science & Big Data Specialization Course, lucca@ppgia.pucpr.br.
[2]Professor from Graduate Program in Informatics (PPGInf) – Federal University of Paraná (UFPR), mazalves@inf.ufpr.br.
[2]Professor from Graduate Program in Informatics (PPGInf) – Federal University of Paraná (UFPR), jean.barddal@ppgia.pucpr.br.

1 Introduction

Nowadays, data is a valuable resource. As time passes and the storage prices decrease, companies tend to accumulate more and more data for further analysis. In this scenario, machine learning comes as a valuable tool to extract knowledge from this enormous data mass. However, traditional machine learning techniques (also referred to as batch machine learning) are not suitable for big datasets that are made available over time. That is because most algorithms require to load the whole dataset into memory, which is very often not possible as the data is potentially unbounded.

Thus, data stream mining techniques were proposed to fill in this gap. Contrary to batch machine learning, these algorithms process the instances once at a time, updating their internal models as each instance arrives. It allows the computation of a potentially infinite amount of data. Thus, high speed and efficient use of memory are essential.

The scientific community currently has two main libraries available for applying data stream mining. One of them is Massive Online Analysis (MOA) [1], implemented in Java, which has been the default option for implementing and comparing methods in the past years. The other one is Scikit-Multiflow [2], more recent and implemented in Python.

Although Scikit-Multiflow is still in its early stages, we believe that it has excellent potential for community adherence since it uses Python implementation, which is considered by many a more beginner-friendly language than Java. However, when comparing the processing time of Scikit-Multiflow to MOA, the latter is significantly faster (approximately 71× faster), as shown in Table 1. We consider this an obstacle for Scikit-Multiflow to acquire more active users.

Table 1: Time in second(s) for executing Naïve Bayes, under the evaluation Prequential, with the SEA data generator with 100,000 instances

MOA	Scikit-Multiflow
0.2000	14.2303

In this paper, we compare a part of the original implementation of Scikit-Multiflow with three proposed

implementations of the same part in the low-level languages C and Rust, with an interface for accessing the algorithm in Python. We have reimplemented the classification algorithm, Naïve Bayes, for streams, the validation process Prequential [3], and the data generators Streaming Ensemble Algorithm (SEA) [4] and Random Radial Basis Function (Random RBF). The source code for our implementation is made available at https://github.com/nbpaper/NaiveBayesImplementations.

2 Related work

Python is naturally not a fast programming language, especially when compared to low-level languages such as C and C++. However, this was not a requirement for Python. Its dynamic-typed system and abstractions of fairly complicated data structures prioritize easiness over speed. Nonetheless, this does not mean that there are not ways of using Python for high-performance computations.

The first and most used Python implementation for high performance is CPython, written in C, as the name indicates. That causes Python to bind to compiled libraries written in low-level languages. One of the most used libraries in the Python ecosystem is NumPy [5], a library for array processing that takes advantage of bindings. NumPy is based on two highly optimized libraries for array calculations written in C and Fortran: BLAS (Basic Linear Algebra Subprograms) [6] and LAPACK (Linear Algebra Package) [7].

```
# Pure Python
random_list = []
for _ in range(1000000):
    random_list.append(random.random())

# NumPy
random_list = np.random.rand(1000000)
```

Figure 1: Code generating random list using pure Python and NumPy

Table 2: Time in seconds for generating random list using pure Python and NumPy

Pure Python	NumPy
0.2274	0.0321

With few code lines, it is possible to demonstrate the difference in processing time from pure Python to NumPy. The first part of the code in Figure 1 generates a list of one million random numbers in pure Python. While at the bottom, the code uses Numpy to achieve the same behavior. Table 2 present the execution time difference (in seconds). NumPy, in this case, performed approximately 7× faster.

Another library that also takes advantage of low-level implementation is NumExpr [8]. It takes input previously built arrays from Python or NumPy and computes array operations with minimal overhead.

Most of these libraries implemented in C/C++ take advantage of Intel Intrinsics routines. These are assembly-coded functions that provide access to highly optimized vector operations. The compiler will convert these routines to SIMD (single instruction multiple data) instructions, which operate with the whole vector simultaneously, which is called vectorization. Besides, C/C++ compilers can automatically translate loops into vectorized code, which is called auto-vectorization.

There are several sets of Intel Intrinsics routines, and as CPUs are improved, more optimized functions are released as Instruction Set Architecture (ISA) extensions to vectorize the code. Modern CPUs maintain backward compatibility with most of the previously released functions. However, applications using newly released functions will not run in older CPUs. When publishing a library that uses Intrinsics, publishers must either specify CPU compatibility or provide multiple implementations, with new and old sets, chosen at compile time. Examples of ISA extensions are SSE (Streaming SIMD Extensions), SSE2, AVX (Advanced Vector Extensions), AVX2, and AVX 512.

There are initiatives in many compiled languages that allow writing code that can be bound to Python. One of these is PyO3 [9], a library written in Rust that allows bidirectional iteration, compiling Rust code so it can be imported from Python and calling Python code from within Rust programs. The orjson [10] library is an example of a Python library for parsing JSON written in Rust and using PyO3.

Another approach for increasing the Python code's speed is to transpile the Python code into C/C++ code, and subsequently, compile it into binary, then importing this compiled version into another Python script. This method has limitations. For example, in order to expect a significant increase in the speed, in many cases, the programmer should give up the dynamic typing convenience offered in Python. The most used tools of this kind are Cython [11] and Numba [12].

Most of the popular Python-based software amongst the scientific community uses one or more of the previously explained approaches. For example, the data processing library Pandas [13], and the machine learning framework Scikit-Learn [14] use NumPy extensively in its internal calculations, as well as Cython in critical parts.

Many recently published papers also take advantage of these tools. For example, the library pyts [15], focused on time series classification, uses Numpy and Numba in its implementation. Cornac [16], a library for recommender systems, on the other hand, uses Cython (instead of Numba) and Numpy.

3 Optimization Analysis

As stated in the introduction, we re-implemented specific parts of the original Scikit-Multiflow code to

obtain improved performance. In this work, we focus on the Naïve Bayes algorithm implementation as a proof of concept.

3.1 Naïve Bayes

The Naïve Bayes is a classification algorithm based on the Bayes Theorem, given in Equation (1). This theorem calculates the conditional probability of A happening, given that B happened. Applying to a classification problem, the probability computed by Naive Bayes is of the set of features x belonging to the class y. This is computed for each of the classes.

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)} \tag{1}$$

3.2 SEA Generator

The SEA generator generates data by creating a random feature vector with tree elements $\{x_1, x_2, x_3\}$, but only two of them ($\{x_1, x_2\}$) contribute to classifying the instance. The definition of the target feature follows a linear threshold. If $x_1 + x_2 > \theta$. The value of θ is variable and changing it in the middle of an algorithm execution creates a concept drift.

3.3 Random RBF Generator

The Random RBF generator works by drawing normally distributed samples around previously created centroids. It works by initially generating n centroids with a random standard deviation associated. When an instance needs to be generated, a centroid is randomly chosen and the attributes for the instance are drawn from a normal distribution with the standard deviation of the centroid. The class of the instance is the same as of the centroid.

3.4 Prequential

The Prequential validation process combines and interacts data generation with the classifier learning and testing processes. Its idea is reasonably straightforward, where for each instance that arrives from the stream (generator), the algorithm uses it first to test the model and then train it. This order is essential because it guarantees that the model is only evaluated against instances never trained before.

Prequential works with integrated performance analyzers that track how well the learning task is at a given moment. The traditional metrics shown as output of Prequential are accuracy and kappa. In this sense, `n_wait` is a vital parameter to define after the interval of how many instances the selected metrics will be evaluated.

3.5 Challenges and Motivation

Although Scikit-Multiflow uses NumPy in its code, many of its inner usages are suboptimal. An example is given in Figure 2, where the matrix `y_proba` is iterated row by row and the `argmax` function is applied to each row, with its output appended on the variable `predictions`. However, with a simple call to a NumPy function, with the parameter `axis = 1`, as shown in the bottom part of the same Figure, NumPy performs the same computation, yet, in an optimized fashion. In order to illustrate this, Table 3 compares the execution time on both pieces of code using as input a matrix of shape (1000000×20) with random numbers ranging from zero to one. The speedup was of approximately 19 times.

Table 3: Time in seconds for applying the argmax function in a random matrix using pure Python and NumPy

Pure Python	NumPy
2.1245	0.1109

```
# Adapted from Scikit-Multiflow
preds = []
for i in y_proba:
    class_val = numpy.argmax(i)
    preds.append(class_val)

# Optimal NumPy code
preds = numpy.argmax(y_proba, axis=1)
```

Figure 2: Code adapted from Scikit-Multiflow and its NumPy version.

Nonetheless, as discussed in the previous sections, the Prequential process works with a single instance at a time, and this represents a problem for code optimization. Libraries such as NumPy have overhead when initializing their arrays as the memory must be allocated all at once to perform fast array operations. This overhead is negligible when dealing with arrays of a reasonable size. However, creating a NumPy array with unitary size per instance presents a significant overhead in the execution.

Because of this characteristic, trying to optimize Scikit-Multiflow's code by transcribing pure Python code to use NumPy or NumExpr functions or correcting NumPy uses such as in Figure 2, may not be the wisest solution. That is why we choose to perform low-level implementations of the selected parts.

3.6 Experiments

For the re-implementation, the languages chosen were C++ (using structures from its standard library), C++ (using Intel Intrinsics AVX for a faster array processing), and Rust (using the ndarray library [17]). All of them provide and were called using a Python interface. The option for C++ was because of its widespread adoption

in the Python community for developing libraries that require fast processing. Rust was also selected to compare a classical language's speed, commonly known for its performance, versus a more modern language, which also targets performance.

Figure 3 provides a comparison between two sums of arrays in order to exemplify the code difference between C++ with its standard library versus C++ with Intel Intrinsics. The functions are very similar, the basic difference lies in comparing lines 6 and 15. While the code using the standard library (line 5) iterates over the vectors of the type `double` and individually sum its elements, the code using AVX use the function `_mm256_add_pd` to sum vectors of type `__m256d`. This type is an array that can store up to 32 bytes of memory, in this case, it is used for storing 4 `doubles`. All elements in one `__m256d` array are summed at the same time on with the function `_mm256_add_pd`, there are similar functions for other types of operations.

```
1   // Using STD
2   vector<double> vec_sum(
3     vector<double> v1, vector<double> v2) {
4       vector<double> ret(v1.size());
5       for (auto i = 0; i < ret.size(); i++)
6         ret[i] = v1[i] + v2[i];
7       return ret;
8   }
9
10  // Using AVX
11  vector<__m256d> vec_sum(
12    vector<__m256d> v1, vector<__m256d> v2) {
13      vector<__m256d> ret(v1.size());
14      for (auto i = 0; i < ret.size(); i++)
15        ret[i] = _mm256_add_pd(v1[i], v2[i]);
16      return ret;
17  }
```

Figure 3: Comparison of vector addition using C++ standard library versus C++ with AVX.

We did not merely translate the algorithms, from Python, into the proposed languages. We also redesigned them with a focus on the performance. Some features present in the original implementation of Scikit-Multiflow were left out, such as performance metric computers other than the accuracy and support for multiple classifiers in Prequential.

We executed the experiments on a Linux Ubuntu 18.04.1 with Intel Core i5-3570 (Ivy Bridge) 3.40GHz with four cores as CPU (although all the experiments use a single thread only), 8 GB of main memory. We executed two different sets of experiments, for accessing the impact on speed by increasing the number of instances and features on each experiment.

For the experiments regarding the instance number, we used the SEA generator. The number of instances generated at each experiment varied from 20 to 100020, with a step of 5000. We set the `n_wait` parameter as 1%, 5%, and 10% of the total number of instances. For analyzing the features number, we used the Random RBF generator, because its design allows us to define the number of features generated at each instance. The number of features generated in this set varied from 2 to 102, with a step of 5. The number of instances and the `n_wait` parameter were set to 50000 and 2500 respectively. The Prequential process was executed ten times for each implementation and each experiment, plus the original Scikit-Multiflow and MOA implementation. The time was measured (in seconds) for each execution. In the end, the average of the time values was computed.

As memory efficiency is also an important part of data stream mining, we also performed a memory usage analysis of all the methods. For this test, a single execution of Prequential was used with the SEA Generator with 100,000 instances and $n_wait = 5000$.

4 Results and Discussion

4.1 Processing Time

The results regarding the instance number increase for `n_wait = 1%` can be seen in Figure 4, y axis displays in logarithmic scale the execution time (in seconds), x axis presents the number of instances. All of the proposed implementations performed very similarly. It is evident the difference between any of them versus the original implementation of Scikit-Multiflow. All of our implementations were also faster than the Java implementation on MOA.

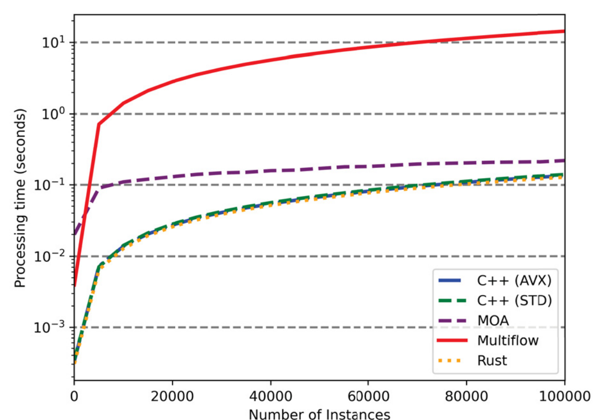

Figure 4: Comparison of processing time between proposed implementations and original Scikit-Multiflow and MOA functions regarding number of instances

Table 4 shows the average processing time of the executions for 20, 50020 and 100020 instances. Once again, is clear the improvement provided by the implementations in low-level languages. With 100020 instances, the implementations in C++, C++ with AVX and Rust outperformed the original implementation by roughly 102×, 105× and 111×. The implementation in MOA was 65× faster.

When comparing only with MOA (again with 100020 instances), the C++ implementations without and with

Intel intrinsics were 1.57× and 1.62× respectively. The Rust implementation was 1.71×.

Table 4: Mean time in second(s) for running the experiments regarding instances

Instances	Multiflow	MOA	C++	C++ AVX	Rust
20	0.0039	0.0199	0.0003	0.0003	0.0003
50020	7.1149	0.1699	0.0700	0.0677	0.0642
100020	14.2713	0.2200	0.1400	0.1350	0.1284

Figure 5 shows the results for the experiments concerning the increase of features. y axis displays in logarithmic scale the execution time (in seconds), x axis presents the number of features. It is clear that there is a point in the features number, where the vectorization provided by AVX starts to be better than the C++ implementation with STL. The vectorization was performed on features level, and it has a cost, so this behavior was expected. The gains of vectorization are better seen as the array sizes are increased, making the cost of vectorization setup negligible.

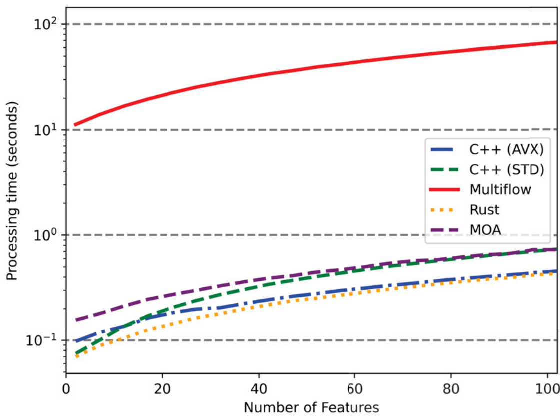

Figure 5: Comparison of processing time between proposed implementations and original Scikit-Multiflow and MOA functions regarding number of features

As for the direct comparison of the times, Table 5 shows the average processing times for 2, 52, and 102 features. Compared to Scikit-Multiflow with 102 features, the C++ (STD) implementation was 92× better, using Intel Intrinsics the number was 147×. Rust was again the best performant, running at 155× faster. The MOA implementation was 92×.

Comparing our implementations with MOA (with 102 features), C++ With Intel intrinsics and Rust performed 1.60× faster and 1.68× faster respectively. The C++ implementation with its standard library virtually tied with MOA.

4.2 Memory Usage

Figure 6 shows the memory usage for all the implementations over time. x axis is the processing

Table 5: Mean time in second(s) for running the experiments regarding features

Features	Multiflow	MOA	C++	C++ AVX	Rust
2	11.2802	0.1559	0.0748	0.0976	0.0697
52	39.5166	0.4470	0.4053	0.2788	0.2519
102	67.2595	0.7300	0.7298	0.4551	0.4329

time normalized from 0 to 1, this was done for the sake of clarity, so all of the curves would end at the same point. y axis is the memory usage in MB. It is evident that all of the proposed implementations used considerably less memory than MOA and Scikit-Multiflow. But it is worth noting that even though Scikit-Multiflow used more memory than MOA, as it is shown in Table 6, it presented a more consistent usage over time. The observed behavior for MOA in Figure 6, with the memory consumption considerably increasing as new instances arrive, is not sustainable for an enormous quantity of instances.

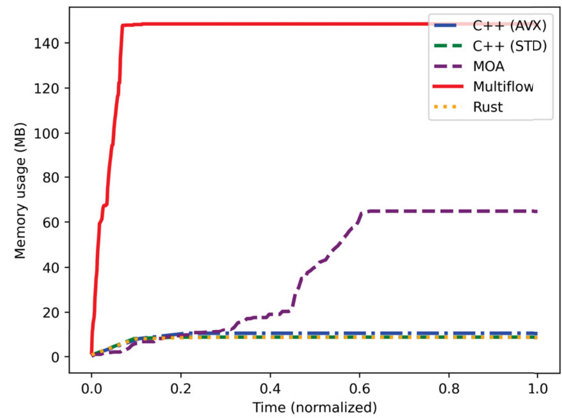

Figure 6: Comparison of memory usage between proposed implementations and original Scikit-Multiflow and MOA.

Table 6: Maximum memory usage (MB) for running Prequential with 100,000 instances generated by the SEA generator.

Multiflow	MOA	C++	C++ AVX	Rust
148.8085	63.8750	8.7343	10.4531	10.0039

5 Conclusions and Future Work

We concluded by our analysis that a high-performance and memory-efficient Python library for mining streams of data is achievable. All the low-level implementations proposed easily outperformed the original Scikit-Multiflow implementation.

The least performant of the implementations proposed by this project was with C++ using elements from its

standard library. This was an expected result because these elements provide a high level of abstraction for dealing with data structures. This provides great help for dealing with complicated algorithms easily, but it comes with a performance cost, which can be observed at the results.

The C++ implementation with Intel Intrinsics and the Rust implementations were very close, with Rust performing slightly better. The result for C++ with AVX was expected because the AVX routines are very efficient in vector processing. The explicit vectorization had better performance with larger features array. Rust also presented a good speedup because, besides being also an extremely low-level language, its internal libraries also abstracts some kind of vectorization. This result is interesting because it showed that Rust has good potential for serving as a backend for Python libraries. As Rust is a more recent language, it provides some facilities related to memory management that C++ does not provide. While also contributes to a faster coding time.

As for the comparison with MOA, all of our implementations outperformed it regarding the number of instances and features. Moreover, since MOA is open-source and widely adopted by the scientific community, it is expected that common algorithms, such as Naive Bayes, are implemented in a reasonably optimized way. Thus, we expect that there is still a gap in speed improvement in our implementations.

References

[1] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *Journal of Machine Learning Research*, 11(52), 2010.

[2] Jacob Montiel, Jesse Read, Albert Bifet, and Talel Abdessalem. Scikit-multiflow: A multi-output streaming framework. *Journal of Machine Learning Research*, 19(72), 2018.

[3] João Gama, Raquel Sebastião, and Pedro Rodrigues. On evaluating stream learning algorithms. *Machine Learning*, 90, 10 2013.

[4] Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. 07 2001.

[5] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2), 2011.

[6] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2), 2002.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, third edition, 1999.

[8] Robert McLeod, Francesc Alted, Antonio Valentino, Gaëtan de Menten, et al. pydata/numexpr: Numexpr v2.6.9, December 2018.

[9] PyO3. Pyo3/pyo3.

[10] Ijl. ijl/orjson.

[11] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2), 2011.

[12] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.

[13] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, 2010.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2011.

[15] Johann Faouzi and Hicham Janati. pyts: A python package for time series classification. *Journal of Machine Learning Research*, 21(46), 2020.

[16] Aghiles Salah, Quoc-Tuan Truong, and Hady W. Lauw. Cornac: A comparative framework for multimodal recommender systems. *Journal of Machine Learning Research*, 21(95), 2020.

[17] Rust-Ndarray. rust-ndarray/ndarray.