Towards the Overcome of Performance Pitfalls in Data Stream Mining Tools

Lucca Portes Cavalheiro
Graduate Program in Informatics (PPGIa)
Pontifícia Universidade Católica do Paraná
Curitiba, Brazil
lucca@ppgia.pucpr.br

Marco Antonio Alves Zanata
Graduate Program in Informatics (PPGInf)
Federal University of Paraná
Curitiba, Brazil
mazalves@inf.ufpr.br

Jean Paul Barddal
Graduate Program in Informatics (PPGIa)
Pontifícia Universidade Católica do Paraná
Curitiba, Brazil
jean.barddal@ppgia.pucpr.br

***Abstract*—Data stream mining is an essential task in today's scientific community. It allows machine learning models to be updated over time as new data becomes available. Three pillars should be accounted for when selecting an appropriate algorithm for data stream mining: accuracy, processing time, and memory consumption. To develop and assess machine learning models in streaming scenarios, different tools have been developed, where the Massive Online Analysis, written in Java, and scikit-multiflow, written in Python, are in the spotlight. Despite the ease of use of both tools, neither are focused on performance, which puts in jeopardy the usage of the computational resources. In this paper, we show that with the right tools, Python libraries reach performance comparable to C/C++. More specifically, we show how optimized implementations in scikit-multiflow using low-level languages, i.e., C++, C++ with Intel Intrinsics, and Rust; with bindings to Python vastly overcome existing tools in computational resources usage while keeping predictive performance intact.**

I. 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 loading 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 or in small chunks, thus updating their internal models as new data becomes available. In seminal papers and books related to data stream mining, three main pillars are brought up for the development of novel techniques [1], [2]:

- Accuracy: algorithms should provide accurate responses over time. This is important as the data distribution may drift over time (concept drift), and machine learning models should adapt accordingly so that accuracy is not put in jeopardy.
- Processing time: algorithms should use a limited amount of time to process each instance. If the processing time scales up according to the arrival of new data, these are expected to be buffered, which may culminate in system crashes due to lack of RAM.
- Memory consumption: algorithms should be light-weighted in terms of memory consumption. Similarly to the processing time constraint, the size of models must not increase indefinitely as new data becomes available.

TABLE I
Time in seconds (s) for executing Naïve Bayes, under the Prequential validation procedure [2], with the SEA data generator [4] with 100,000 instances.

MOA	scikit-multiflow
0.2000	14.2303

To develop and assess machine learning models in streaming scenarios, different tools have been developed. The first tool is the Massive Online Analysis (MOA) [1], implemented in Java, which has been the default option for implementing and comparing methods in the past years. More recently, authors in [3] proposed scikit-multiflow, which is still in its early stages but depicts the potential for becoming the off-the-shelf solution for new researchers and practitioners in the data stream mining community. Despite the ease of use of both tools, neither are focused on performance, which puts in jeopardy the usage of the computational resources. For instance, when comparing the processing time of scikit-multiflow to MOA, the latter is significantly faster (approximately $71\times$ faster), as shown in Table I. We consider this an obstacle for scikit-multiflow to acquire more active users and to fill in the processing time and memory usage requirements, which are relevant when these models are to be deployed.

In this paper, we compare and analyze the original implementation in scikit-multiflow against three proposed implementations of the same part in the low-level languages C++ and Rust, with an interface for accessing the algorithm in Python. More specifically, we have reimplemented the Naive Bayes classification algorithm, the validation process Prequential [2], 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.

This paper is divided as follows. Section II brings forward related works on Python code optimization that is required to understand our proposal. Section III layers the scope of our code optimization, including classifiers, data generators, validation procedures, as well as details on how our results are conducted. Section IV discusses the results obtained. Finally, Section V concludes this work and states envisioned future works.

II. 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].

With few lines of code, 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. The code below it uses NumPy to achieve the same behavior. Table II present the execution time difference (in seconds). NumPy, in this case, performed approximately 7 times faster. These differences serve as a basis on why libraries such as NumExpr [8] take as input previously built arrays from Python or NumPy and compute array operations swiftly.

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 converts 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.

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

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

Fig. 1. Code generating random list using pure Python and NumPy

TABLE II
Time in seconds (s) for generating random list using pure Python and NumPy

Pure Python	NumPy
0.2274	0.0321

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 do 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 [1], 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 [2] library is an example of a Python library for parsing JSON written in Rust which benefits from 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, and finally to import this compiled version into another Python script. This method has caveats. For instance, in order to expect a significant increase in 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 [9], and Numba [10].

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 [11], and the machine learning framework scikit-learn [12] 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 [13], focused on time series classification, uses Numpy and Numba in its implementation. Cornac [14], a library for recommender systems, on the other hand, uses Cython (instead of Numba) and Numpy.

III. The Scope of our Optimization Analysis

As stated in the introduction, we re-implemented specific parts of the original scikit-multiflow code to obtain improved performance. In the following sections, we provide a background on the classification model, data generator, and validation process used in this optimization analysis. Finally, we bring insights on why this optimization is required given the existing scikit-multiflow implementation.

[1]https://github.com/PyO3/pyo3
[2]https://github.com/ijl/orjson

A. 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 it 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.

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

B. 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, i.e., if $x_1 + x_2 > \theta$ then $y = 1$ and $y = 0$, otherwise. The value of θ is variable and changing it in the middle of an algorithm execution synthesizes a concept drift.

C. 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 is 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.

D. Prequential

The Prequential validation process combines and interacts data generation with the classifier learning and testing processes. In prequential, 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 the interval (number of instances) assumed between evaluation metrics computation.

E. Challenges and Motivation

Even though 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 III 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 is of approximately 19 times.

TABLE III
TIME IN SECONDS (s) FOR APPLYING THE ARGMAX FUNCTION IN A RANDOM MATRIX USING PURE PYTHON AND NUMPY.

Pure Python	NumPy
2.1245s	0.1109s

```
# 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)
```

Fig. 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 overheads 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 a 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.

F. Experiments Setup

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[3]). 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 similar, but 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, which in this case is used for storing 4 `doubles`. All elements

[3]https://github.com/rust-ndarray/ndarray

```
// Using STD
vector<double> vec_sum(
  vector<double> v1, vector<double> v2) {
    vector<double> ret(v1.size());
    for (auto i = 0; i < ret.size(); i++)
      ret[i] = v1[i] + v2[i];
    return ret;
}

// Using AVX
vector<__m256d> vec_sum(
  vector<__m256d> v1, vector<__m256d> v2) {
    vector<__m256d> ret(v1.size());
    for (auto i = 0; i < ret.size(); i++)
      ret[i] = _mm256_add_pd(v1[i], v2[i]);
    return ret;
}
```

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

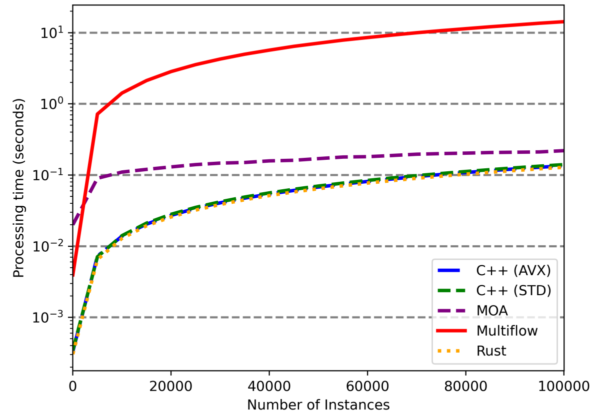

Fig. 4. Comparison of processing time between proposed implementations and original scikit-multiflow and MOA functions regarding number of instances.

in one __m256d array are summed at the same time with the function _mm256_add_pd.

We did not merely translate the algorithms from Python into the proposed languages. We also redesigned them with a focus on performance. Some features available 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. No other jobs were executing during experimentation. 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 number of features, we used the Random RBF generator because its design allows us to set up the number of features generated. 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. The time was measured (in seconds) for each execution, and the results reported are averages obtained across the runs.

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 $n\_wait = 5000$.

IV. Results and Discussion

This section reports and discusses the results obtained in terms of processing time and memory consumption.

TABLE IV
MEAN TIME IN SECOND(S) FOR RUNNING THE EXPERIMENTS REGARDING INSTANCES.

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

A. Processing Time

The results regarding the instance number increase for n_wait = 1% can be seen in Figure 4, where the y axis displays in logarithmic scale the execution time (in seconds) and the x axis presents the number of instances. All of the proposed optimized implementations performed similarly, and it is evident that the proposed implementations are significantly faster than the original scikit-multiflow and Java's implementation provided in MOA.

Table IV shows the average processing time of the executions for 20, 50020 and 100020 instances. As before, the improvement provided by the implementations in low-level languages is clear. With 100020 instances, the implementations in C++, C++ with AVX, and Rust outperformed the original implementation by roughly 102, 105, and 111 times, respectively, while MOA's implementation was 65 times faster.

When comparing the proposed implementations solely against MOA (again with 100020 instances), the C++ implementations without and with Intel intrinsics were 1.57 and 1.62 times faster, respectively. We also observe that the Rust implementation was 1.71 times faster.

Figure 5 shows the results for the experiments concerning the increase of features, where the y axis displays in logarithmic scale the execution time (in seconds) and the x axis presents the number of features. It is clear that there is an inflection point in which the number of features that causes the AVX vectorization to outperform the C++ implementation with STL. The vectorization was performed on the features level, and it has a cost, so this behavior was expected. More specifically, the improvements caused by vectorization are higher when array sizes are increased, rendering the cost of vectorization setup negligible.

As for the direct comparison of the processing times, Table V shows the average processing times for 2, 52, and 102 features. Compared to scikit-multiflow with 102 features, the C++ (STD) implementation was 92 times faster, while using Intel Intrinsics the number was 147 times. Rust was again the best performant choice, running 155 times faster. On the other hand, MOA's implementation was 92 times faster than scikit-multiflow's original code.

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

B. Memory Usage

Figure 6 shows the memory usage for all the implementations over time, where the x axis denotes the processing time normalized from 0 to 1 and the y axis is the memory usage in megabytes (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 VI, it presented a more consistent usage over time. The observed behavior for MOA in Figure VI, i.e., the memory consumption considerably increasing as new instances arrive, is not sustainable for an enormous and potentially quantity of instances.

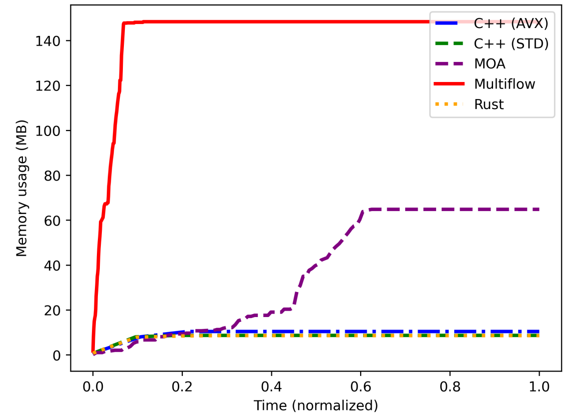

Fig. 6. Comparison of memory usage between proposed implementations and original scikit-multiflow and MOA.

TABLE VI
MAXIMUM MEMORY USAGE (MB) FOR RUNNING PREQUENTIAL WITH 100,000 INSTANCES GENERATED BY THE SEA GENERATOR.

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

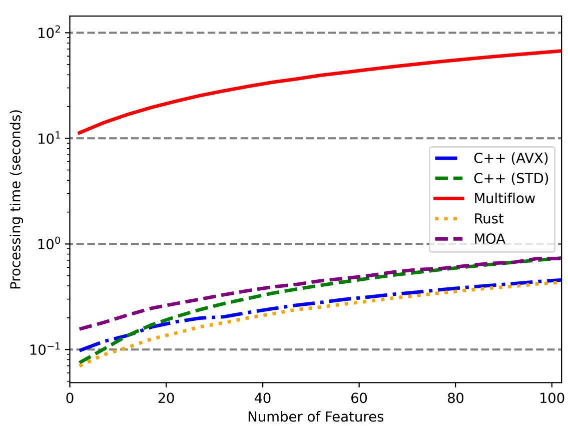

Fig. 5. Comparison of processing time between proposed implementations and original scikit-multiflow and MOA functions regarding number of features.

TABLE V
MEAN TIME IN SECONDS (s) FOR RUNNING THE EXPERIMENTS REGARDING FEATURES

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

V. CONCLUSIONS AND FUTURE WORK

In this paper, we conclude that a high-performance and memory-efficient Python library for mining streams of data is achievable. All the low-level implementations proposed outperformed the original scikit-multiflow implementation written in Python and Massive Online Analysis (MOA) implementation in Java.

The least performant of the proposed implementations 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 in 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 a larger feature array. Rust also presented a good speedup because, besides being also an extremely low-level language, its internal libraries also abstract types 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 contributing to 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, we believe that this is a gap that should be addressed in future works and implementations so that processing time and memory consumption constraints are not violated when models are deployed.

REFERENCES

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

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

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

[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] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.