dHealth 2022
G. Schreier et al. (Eds.)

doi:10.3233/SHTI220369

Integration of Python Modules in a MATLAB-Based Predictive Analytics Toolset for Healthcare

Lukas HAIDER[a,b,1], Martin BAUMGARTNER[a,b], Dieter HAYN[a,c] and Guenter SCHREIER[a,b]
[a] *AIT Austrian Institute of Technology, Graz, Austria*
[b] *Institute of Neural Engineering, Graz University of Technology, Graz, Austria*
[c] *Ludwig Boltzmann Institute for Digital Health and Prevention, Salzburg, Austria*

Abstract. Background: Python and MATLAB both are common tools used for predictive modelling applications, not only in healthcare. In our predictive modelling group, both tools are widely used. None of the two tools is optimal for all tasks along the value chain of predictive modelling in healthcare. Objectives: The aim of this study was to explore different ways to extend our MATLAB-based toolset with Python functions. Methods: Pre-existing interfaces between MATLAB and Python have been evaluated and more comprehensive interfaces have been designed to exchange even complex data formats such as MATLAB tables. Results: The interfaces have successfully been implemented and they were validated in a Natural Language Processing scenario based on free-text notes from a telehealth services for heart failure patients. Conclusion: Integration of Python modules in our MATLAB toolset is possible. Further improvements especially in terms of performance, are required if large datasets need to be exchanged. A big advantage of our concept is that tabular data can be exchanged between MATLAB and Python without loss and the Python functions are called dynamically via the interface.

Keywords. MATLAB, Python, Predictive Analytics, Machine learning, Data exchange, Interfacing

1. Introduction

1.1. Background

With increasing digitalization and the use of telehealth systems and wearables in medicine, ever larger amounts of data are being generated. The processing and interpretation of these large amounts of data is difficult for humans. Therefore, predictive modelling and other artificial intelligence (AI) technologies are becoming more and more popular in healthcare [1] [2].

To create accurate predictive models, specialists from different disciplines and with different background are needed to contribute their knowledge during the model development. Furthermore, the presentation of the models must be understandable and comprehensible for humans. This requires interactive graphical tools that combine

[1] Corresponding Author: Lukas Haier, Institute of Neural Engineering, Graz University of Technology, AIT Austrian Institute of Technology, Reininghausstr. 13/1, 8020 Graz, Austria, E-Mail: L_haider@gmx.at

human intelligence with the computing power of computers to produce optimal results. [3] [4] [5]

1.2. Predictive Analytics Toolset for Healthcare (PATH)

In the past years, a 'Predictive Analytics Toolset for Healthcare' (PATH) has been developed at the AIT Austrian Institute of Technology [6]. This tool is based on MATLAB (MathWorks, Natick, US). PATH consists of various modules supporting the whole process from objective definition, data de-identification and model development to deployment. The relation of these modules to one another can be specified via a graph-based GUI in MATLAB. For data transfer between the modules, MATLAB tables and structures are used: Mainly, data are passed using MATLAB tables, while structures are used to exchange settings and metadata.

Additionally, several modules have been developed at AIT especially for deep leaning applications, which are based on Python (The Python Software Foundation, www.python.org). So far, researchers had to decide whether MATLAB or Python should be used for a specific research question, since no interface between the respective modules existed. One example of such a pre-existing Python module is a natural language processing (NLP) module for classifying telehealth notes into one or more categories [7].

1.3. State of the art

In the literature, six different ways to link MATLAB and Python are described. Already in 2009, a first MATLAB to Python compiler was published by Jurica and von Leeuwen [8]. In 2015, MATLAB introduced its own product called MATLAB Compiler SDK [9]. Both products have the capability to compile MATLAB code into Python code. This interface is suitable if MATLAB code should be integrated in Python. However, it wasn't useful for our interface because we needed the other direction, i.e., to integrate Python code into MATLAB.

Another published way is to use a built-in interface like the Python environment ("*pyenv*") in MATLAB or MATLAB Engine in Python [9] [10] [11]. Both interfaces are provided by MALTAB and have the capability to call Python functions in MATLAB or MATLAB functions in Python. This is a very interesting way of interfacing for our use case. However, the exchange of MATLAB *tables* is not supported.

Another published option is exchanging data via files. Different file types can be used, depending on the specific requirements of the interface. For example, Keshavarz and Mojra in [12] use Python-files and txt-files to exchange data. MathWorks published a solution based on parquet–files for data exchange, with column orientated data like *tables* [9]. Furthermore, it is also possible to use MAT-files for this task, since there are Python modules like *SciPy* [13] and *hdf5storage* [14] supporting this original MATLAB file format. However, metadata saved in the MATLAB *table properties* are not supported by these approaches.

MATLAB also supports the use of the Open Neural Network Exchange (ONNX) format to use Python based frameworks in MATLAB [9]. However, this approach does not support data processing in MATLAB and Python together.

Another published way is to use a module called *mlabwrap* [15]. This module has the capability to use MATLAB in Python. It was published by Bednar in 2009 [16] and uses a similar way like the MATLAB Engine in Python [9]. Since *mlabwrap* isn't

updated since 2011, it did not support some of the newly developed MATLAB data formats and, therefore, it was not applicable for our interface.

Summing up all the approaches described above, two types of data transfer were identified:

1. Using "direct" interfacing with built-in functions
2. Using files (e.g., MAT-files) stored on the disc for data exchange

However, each of these options comes with certain drawbacks:

1. using "direct" interfacing: converting non-supported data types into supported data types, computing time, only a few Python versions are supported by MATLAB
2. using "mat-files": disk-space, time for saving and loading, different "mat-file" formats like ('-v6','-v7' or 'v7.3') etc.

No matter which of the two approaches is used, the transferred data types must be supported. Our literature research revealed:

- Supported data types: scalar structures, doubles, 1-dimentional cell array, 1-dimentional char array
- Unsupported data types: Struct arrays, tables, time stamps, string arrays [9]

In Python, *pandas* is a common module in data science. This module includes a class called *DataFrame* which is similar to MALTAB *tables* [17]. Therefore, this class seemed promising for our interface.

1.4. Objectives

The aim of this work was to explore ways to integrate Python modules into PATH. For this purpose, different concepts should be developed to call Python functions from MATLAB and to exchange data including MATLAB *tables* and *structures* between MALTAB and Python. Beyond feasibility, the concept should also be efficient in terms of the computation time needed to exchange data, compared to the computing time of the function. The concepts should be validated by integrating a pre-existing, Python-based natural language processing (NLP) module [7] into PATH, based on a telehealth dataset for heart failure patients [18].

2. Methods

2.1. Interface implementation

For the implementation of the interface, both alternatives described in chapter 1.3. were used, i.e., 1. 'direct' interfacing with built-in functions and 2. „mat-files" for data exchange. In the first concept, a data stream was passed while in the second concept files were used to exchange data between MATLAB and Python.

On the MATLAB side, data types not supported for exchange between MATLAB and Python were converted to supported data types. *Tables* and *timetables* were transferred to *structs* prior exchange with Python. Therefore, MATLAB provides two built-in functions called *table2struc* and *struct2table*. While these functions supported the exchange of table content, metadata stored in *table properties* were lost. Therefore, such metadata were transformed to a structure *tableMetadata*, which was stored in a nested structure called *dataExchangeStruct* together with the actual data. For translation of this structure into a *table* including *table properties,* another algorithm was developed.

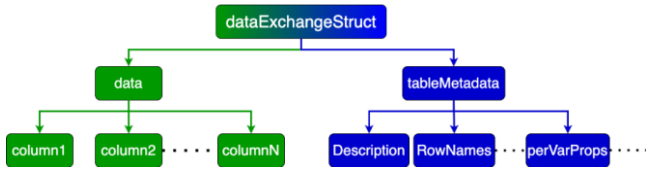

Figure 1. Design of the three-level structure to exchange table between MATLAB and Python

In Fig. 1, the general design of the data is shown. The green part represents the actual data, while the blue part contains the metadata of the *table properties*. In the end, a three-layer nested structure was the most-suited application for data exchange.

On the Python side, a specific class was developed which contained all algorithms to prepare the data and store them as pandas *DataFrame*. Metadata were stored as *dictionary*. This class contained a method for importing files or variables provided by MATLAB as described above. Also, a function was developed which coded Python data in the same format to send back data from Python to MATLAB.

The exact workflow of calling the Python function is shown in the sequence diagram plotted in Fig. 2.

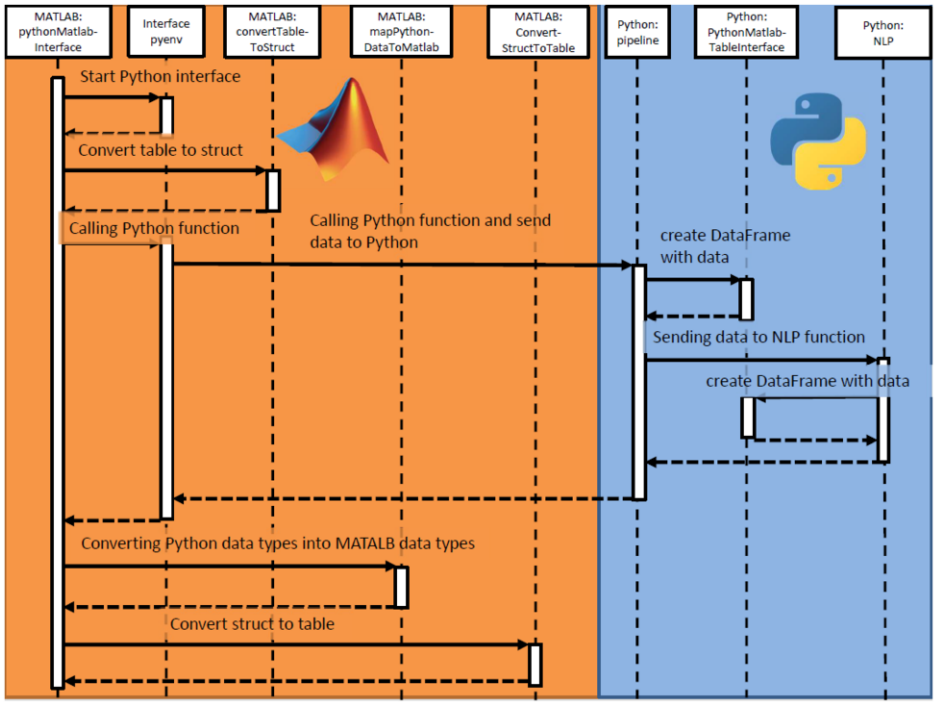

Figure 2. Sequence diagram of the interface. Orange: MATLAB code; Blue: Python code

The interface is started by calling the newly developed function "*pythonMatlabInterface*". First, the interface checked if the Python environment is already running. If not, it was started using the MATLAB command *pyenv*. Next, the data stored in tables were converted to the nested structure *dataExchangeStruct* shown in Fig 1. Then, the Python function was called using the command *py.NLP* and the data were passed to the Python function. There, the structure *dataExchangeStruct* was divided

into the *table* data, stored in the *pandas DataFrame,* and the *metadata*, stored as *dictionary*. Next the data were passed to the NLP function and where the actual data processing took place in Python. The results were stored into a new *DataFrame* using the developed "*PythonMatlabTableInterface*". Then, the data stream created within this class was sent back to MATLAB, where all Python data types were mapped into MALTAB datatypes and the *dataExchangeStruct* was converted to a MATLAB *table*.

There is also the option to use "mat-files" for data exchange instead of the Python environment. Therefore, the same functions as shown in Fig 2were used. Instead of calling the Python function in MATLAB, the data were saved to a 'MAT-file'. Thereafter, the "mat-files" were loaded in Python using the developed class. For returning data from Python to MATLAB, "mat-files" were saved in Python and loaded in MATLAB. The default version of the "mat-file" was the so-called version '-v6'. If necessary, data could be saved and loaded also in all other versions.

2.2. Performance test

We have tested the performance of all approaches to find out, how much time is needed to transfer the computation between MATLAB and Python, depending on the amount of data. Therefore, four different performance tests were conducted:

1. We compared the time for saving tables with the build-in MATLAB functions 'save' and 'load' with the time for saving and loading tables with the developed interface in MATLAB. A table of size 1,000 MB, containing 113,000 rows and 1,000 columns, was used. Eight columns were of type datetime, four columns were cells of characters, the other columns were of type double. Different versions for saving files in MATLAB, i.e., '-v6' '-v7' and '-v7.3', were tested.
2. We analysed the time to convert a table into the developed structure and to convert this structure back into a table in MATLAB. The columns corresponded to those in test 1, while different numbers of rows were tested.
3. The time for loading and saving "mat-files" in Python was examined. "mat-files" of format '-v6', '-v7' and '-v7.3' as generated in test 1 were used (1,000 MB table, 113,000 rows, 1,000 columns).
4. The processing time of the data-stream from MATLAB to Python and back was analysed. The data stream was generated from a table, which was converted form the "mat-file" in test 1.

Each test was repeated 50 times and mean and standard deviation were calculated.

2.3. Pilot application in a real-world scenario with telehealth free-text annotations

While the tests described in 2.2 focussed on technical interoperability, pragmatic interoperability was tested by applying the interface on pre-existing Python and MATLAB tools in a pilot application. Therefore, the feasibility of the developed interface was evaluated in a pilot application with real world telehealth data. Therefore, a dataset with free text notes from "HerzMobil Tirol" [18] was used. "HerzMobil Tirol" is a telehealth program where patients record their vital parameters (e.g., blood pressure, heart rate, ...) each day and submit those data to a backend service. Additionally, patients and healthcare professionals can enter free text notes, e.g., concerning subjective wellbeing, therapy, diagnosis, education and training, technical problems, etc. Classification of these notes via natural language processing (NLP) was already explored by Wiesmüller et al. in a previous work [7]. The NLP module developed by Wiesmüller

was implemented as a Python script. This script performed all operations by loading the data and saving the classes per free-text note to a result file.

This pre-existing script was included in a PATH process. The results obtained by calling the script from MATLAB and transferring all data from MATLAB to Python and transferring the results back from Python to MATLAB were compared to the results obtained by the original Python script.

3. Results

3.1. Performance test

We evaluated the two options of the interface as described above and compared the results of MATLAB / Python only on a PC with following hardware specifications:16GB RAM and a processor Intel® Core™ i7-4610M CPU @ 3.00GHz. The software specifications are: Windows 10 Enterprise, MATLAB R2021b, Python 3.8, Python IDE: PyCharm Community Edition 2021.1.3

Tab. 1 shows the storage and loading time of tables and the standard derivation σ with the built-in MATLAB function and the developed algorithm. '-v6', '-v7' and '-v7.3' here stands for the "mat-files" saving versions defined by MATLAB. There you can see, that the developed algorithm had similar saving and loading times.

Table 1: Performance of the developed functions compared to the built-in MATLAB functions for saving and loading different "mat-file" versions (mean ± standard deviation [s])

MAT – file version	'-v6'	'-v7'	'-v7.3'
MATLAB: save	6.222 ± 1.245	25.622 ± 1.861	65.409 ± 4.887
saveTableForPython	5.408 ± 0.392	27.309 ±1.862	67.442 ±5.420
MATLAB load	4.248 ±0.192	8.445 ± 0.183	32.712 ± 3.234
loadTableFromPython	4.786 ± 0.201	11.285 ± 0.301	25.417 ±2.151

Tab. 2 shows the time needed to convert data of different sizes between tables and the newly developed data structure. It could be shown that the developed algorithm is verry fast and the converting time is short compared to the saving time shown in Table 1

Table 2: Time to convert data of different size between a MATLAB table and a struct (mean ± standard deviation [s])

Size	500 MB	1,000 MB	1,500 MB	2,000 MB
convertTableToStruct	0.153 ± 0.040	0.393 ± 0.083	0.747 ± 0.056	1.028 ± 0.103
convertStructToTable	0.136 ± 0.021	0.369 ± 0.079	0.714 ± 0.057	1.005 ± 0.098

Tab. 3 shows the time that it takes to load or save the "mat-files" stored in version '-v6' and 'v7' in Python. Saving the data takes four times longer than loading the data from a "mat-file". This algorithm is not as fast as the MATLAB functions shown in Table 1. Tab. 4 shows the time required to process the data flow in Python. This algorithm is clearly slower than the MATLAB algorithm.

Table 3: Time to load and save different "mat-file" versions of files with the size of 1,000 MB in Python using the developed Python class (mean ± standard deviation [s])

MAT – file version	'-v6'	'-v7'	'-v7.3'
Load MATLAB table	25.606 ± 0.737	28.451 ± 0.457	-
Save MATLAB table	100.952 ± 0.782	101.205 ± 0.412	-

Table 4 Process data from and to data stream in Python (size: 1000 MB)

	Mean ± standard deviation [s]
Store data from MATLAB data stream into Python class	45.266 ± 1.440
Create data stream from MATALB	51.224 ± 4.664

3.2. Pilot application in a real-world scenario with telehealth free-text annotations

We applied the developed interface to a predictive modelling scenario which combined pre-existing modules coded in MATLAB and Python, respectively. Since only functions (not scripts) can be called by PATH, the pre-existing NLP script was wrapped into a Python function. For the whole workflow, the file containing all free text notes was loaded as a *table* in MATLAB and transferred to the Python function via the newly developed interface. The Python module classified all free text notes to one or more classes and generated a Python *dictionary*. This *dictionary* was then transferred back to MATLAB via the interface. The whole process for calling the NLP function in Python was successfully tested within PATH. The Python call did not significantly increase the overall processing time and further processing of the results sent back to MATLB was possible without any restrictions.

4. Discussion

We have successfully developed an interface between MATLAB and Python which supports the exchange of all required data formats, including *tables.* As can be observed in Tab 1, the developed algorithm is similarly fast as the built-in MATLAB functions. Therefore, this way is well suited to store *tables* in a form that can be read in Python. The time for storing and loading data depends significantly on the "mat-file" version. Version '-v6' and '-v7' are binary files developed by MathWorks. Different time between version '-v6' and '-v7' are due to compression in '-v7'. Files of version '-v7.3' are based on the HDF5 standard. Conversion to HDF5 and compression leads to the significantly increased time losses for this format. However, variables larger than 2GB can only be stored in '-v7.3' in MATLAB.

Comparing the loading and saving time of MATLAB to the times in Python (Tab. 1 and Tab. 3) shows that Python has a loading time that is up to 5 times longer and a saving time that is up to 20 times slower. This is due to the fact, that the exact structure of the MAT file is not known. Therefore, the *SciPy* algorithm was developed by trial and error, which exact structure is included in a "mat-files". Also, only the '-v6' and '-v7' versions are listed in Table 3. This is due to the fact, that the module for storing and loading files of version '-v7.3' has a very bad performance and the times for 1,000 MB were not measurable. Therefore, it is currently not possible to use files stored in version '-v7.3' in our interface.

When comparing Tab. 2 and 4, it can be seen, that the conversion of a *table* into the newly developed *structure* in MATLAB is very fast. The time depends on the size and the content of the *table*. The more columns with *datetimes* and *cells of characters* are contained, the longer the conversion takes. Converting this *structure* to a *pandas DataFrame* in Python takes much longer than the conversion in MATLAB.

Future updates of the algorithm will focus on the improvement of performance for converting the structure to a *pandas DataFrame* in Python. Additionally, ways to exchange data with more than 2 GB need to be explored (e.g., by splitting up such large

variables into multiple smaller files). Similarly, the developed interface currently always loads the whole amount of data, which may lead to limitations of the memory available.

The developed interface extends the development of new modules in PATH. Thus, new modules, e.g. for the evaluation of "HerzMobil Tirol", can now be developed in Python. This will in the future support the inclusion of new (Python) developers in our PATH developer team. Additionally, it will help us to evaluate state-of-the art approaches, no matter whether they were developed in Python or in MATLAB environments.

Finally, the presented concept of combining MATLAB and Python is useful to tackle similar challenges related to exchanging data stored in *tables* without losses and help to improve the interoperability between MATLAB and Python based data processing and AI approaches in general.

Acknowledgement

This work was performed in the context of the d4HealthTirol project, which is funded by the Land Tirol.

References

[1] D. V. Dimitrov, Medical Internet of Things and Big Data in Healthcare, *Healthc Inform Res.* **22**(3) (2016), 156–163.
[2] Dash, S., Shakyawar, S.K., Sharma, M. et al., Big data in healthcare: management, analysis and future prospects, *J Big Data* **6** (54) (2019)
[3] D. Keim et al., Mastering The Information Age – Solving Problems with Visual Analytics., (2010).
[4] D. Keim et al. , *Visual Analytics: Definition, Process, and Challenges*, in Information Visualization. Springer-Verlag, Berlin Heidelberg, 2008, pp. 154-175.
[5] S. Liu et al, Towards better analysis of machine learning models: A visual analytics perspective, *Visual Informatics*, (2017), **1**(1) 48-56.
[6] D. Hayn et al., Predictive analytics for data driven decision support in health and care, *it - Information Technology,* **60**(4) (2018)183-194.
[7] F. Wiesmüller et al., Natural Language Processing for Free-Text Classification in Telehealth Services: Differences Between Diabetes and Heart Failure Applications, *Studies in health technology and informatics* **279** (2021). 157- 164.
[8] P. Jurica and C. van Leeuwen, OMPC: an open-source MATLAB-to-Python compiler, *Frontiers in NEUROINFORMATICS*. **3**(5) (2009), 1-9.
[9] MathWorks, MathWorks, https://de.mathworks.com/products/matlab/matlab-and-python.html. last access: 02 Februar 2022.
[10] I. Pill et al., SIMULATE: A Toolset for Fault Injection and Mutation Testing of Sumulink Models, in *IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops*, Chicago, IL, USA, 2016 pp. 168-173.
[11] Y. Koush et al., OpenNFT: An open-source Python/Matlab framework for real-time fMRI neurofeedback training based on activity, connectivity and multivariate pattern analysis, *NeuroImage,* **156** (2017), 489-503.
[12] M. Keshavarz and A. Mojra, "Geometrical features assessment of liver's tumor with application of artificial neural network evolved by imperialist competitive algorithm," *International Journal for Numerical Methods in Biomedical Engineering,* **31**(5) (2015), 1-19
[13] The SciPy community, SciPy, https://docs.scipy.org/doc/scipy/tutorial/io.html., last access: 3.2.2022.
[14] F. Nordsiek, hdf5storage, https://pythonhosted.org/hdf5storage/information.html#matlab-mat-v7-3-file-support. last access: 3.2.2022.
[15] A. Schmolck and V. Rathod, mlabwrap v1.1, http://mlabwrap.sourceforge.net/. last accessed 3.2.2022.
[16] J. A. Bednar, Topographica: building and analyzing map-level simulations for Python, C/C++, MATLAB, NEST, or NEURON components, *Frontiers in Neuriinformatics,* **3** (2009), 1-9.
[17] The pandas development team, pandas, https://pandas.pydata.org/docs/index.html. last access 3.2.2022.
[18] A. Von der Heidt et al., HerzMobil Tirol network: rationale for and design of a collaborative heart failure disease management program in Austria, *Wiener klinische Wochenschrift,* **126** (2014),734-741.