A Python Framework for Exhaustive Machine Learning Algorithms and Features Evaluations

Fabien Dubosson*, Stefano Bromuri,*† and Michael Schumacher*
*AISLab, HES-SO Valais//Wallis
†Management, Science and Technology,Open University of the Netherlands
{fabien.dubosson,michael.schumacher}@hevs.ch
stefano.bromuri@ou.nl

***Abstract*—Machine learning domain has grown quickly the last few years, in particular in the mobile eHealth domain. In the context of the *D1NAMO* project, we aimed to detect hypoglycemia on Type 1 diabetes patients by using their ECG, recorded with a sport-like chest belt. In order to know if the data contain enough information for this classification task, we needed to apply and evaluate machine learning algorithms on several kinds of features. We have built a Python toolbox for this reason. It is built on top of the scikit-learn toolbox and it allows evaluating a defined set of machine learning algorithms on a defined set of features extractors, taking care of applying good machine learning techniques such as cross-validation or parameters grid-search. The resulting framework can be used as a first analysis toolbox to investigate the potential of the data. It can also be used to fine-tune parameters of machine learning algorithms or parameters of features extractors. In this paper we explain the motivation of such a framework, we present its structure and we show a case study presenting negative results that we could quickly spot using our toolbox.**

***Keywords*-machine learning; python; framework; evaluation; features; grid search**

I. Introduction

Machine learning is becoming more and more used in all sorts of fields nowadays. The personal health field is also following this trend, partly due to the quantified-self movement that makes the number of mobile health devices – such as the Fitbit[1] or the Jawbone Up[2] – increase. These devices are generating a lot of data, such as continuous heart-rate signals or accelerometers output. Machine learning is useful for this analysis of the large amount of collected data and the extraction of knowledge from it.

The *D1NAMO* project aims at defining a personal health system [1] to monitor diabetes type 1 patients by using their physiological signals. In *D1NAMO* we want to use machine learning to achieve a non-invasive hypo/hyper-glycemia detection. Physiological signals are acquired by the Zephyr BioHarness 3[3], which is a sport-like chest belt. The use of machine learning permits to identify points of interest in the signal or to summarize the data.

[1]https://www.fitbit.com/
[2]https://jawbone.com/up
[3]http://www.zephyranywhere.com/products/bioharness-3

The traditional machine learning pipeline is usually composed of the following steps:

- Data acquisition
- Data pre-processing (formatting, cleaning,)
- Features selection and extraction
- Model selection and training
- Evaluation

The two most important steps are the *Features selection and extraction* and the *Model selection and training*. A careful selection of the features and model is important as it impacts highly the overall efficiency of the process. On the other hand, the *Data acquisition* is usually limited by the projects settings – such as the experiment setup and the devices used – making it difficult to change once the project has been started. The *Data pre-processing* is a needed step in order to keep relevant data in a usable format. All signal processing meant to be useful for the feature extraction, such as normalization for instance, are excluded from this definition. Hence, this step does not offer much place for improvement. Finally the *Evaluation* is used to measure, analyze and compare the efficiency of the algorithms.

The selection of the features and of the model can be achieved in two ways. First by using some domain-knowledge, like selecting the features that are known to contain the needed information and selecting the model that has proven to be successful on this kind of features. The alternative option is to select a set of features and a set of machine learning algorithms of interest, and to do an exhaustive evaluation on the cross product of these two sets.

This paper presents a machine learning framework that allows an exhaustive search over a set of features extractors and a set of machine learning algorithms. There are two main usages for it: firstly analyzing of the data in order to see the potential of the data to address the research problem, and secondly at the end adjusting the parameters of the classification algorithms and features extractors in order to maximize efficiency.

Most of the existing Python machine learning frameworks are tools that provide machine learning algorithms, with some differences in the proposed algorithms or in their implementations. Some of them are for instance written

in pure Python, some others are written in CPython [2], or some others are using *theano* [3], [4] – a CPU/GPU array processing framework – as backend. The framework presented in this paper differentiates from these ones as it is not offering machine learning algorithms, but it is based on them and offers a higher-level way to apply exhaustively a set of machine learning algorithms.

The framework also uses several packages from the Python ecosystem. The scikit-learn [5] project is a python framework offering machine learning algorithms implementation in Python with a coherent API [6]. Our framework is built on top of it for the machine learning part. There are also some alternatives such as *pylearn2* [7], *pyML* or *PyBrain*, but scikit-learn was chosen for its large set of algorithm and its coherent API. The part of the code dealing with data is built on top of *pandas* [8]. A large consensus in the Python community is to use *numpy* [9] as the building block for numerical computations, and most of the above-mentioned Python packages are built on it. This allows to easily share the matrices and data between all the packages.

The works that are probably the most related to the framework of this paper are the self-made scripts of researchers, which are already doing this kind of machine learning workflow. Our toolbox tries to provide an extendable abstraction of the data, and has already showed its usefulness when porting another dataset to the framework with ease: the same code for the algorithms and the features has been used without modifications on both datasets, the only work having been needed is the database wrappers

The rest of this paper starts with a presentation of the *D1NAMO* project in the section II. The following section, III, describes our toolbox and finally section IV is presenting its usage in a case study: we are presenting negative results concerning the problem of detecting hypoglycemia with only one lead ECG, and showing that by means of our framework we simplify the experimental set up allowing researchers to quickly obtain a set of results to decide how to further proceed with the data.

II. The *D1NAMO* project

Diabetes type 1 is an autoimmune disease that affects the insulin level of a patient. Once a patient is affected by diabetes type 1, the only possible treatment is insulin shots several times a day to keep the insulin level under control and keep the risk of hypoglycemia low [10]. Unfortunately, intensively controlled glycemia levels get patients to have a higher outcome from the perspective of microvascular and macrovascular complications of diabetes type 1, meaning that there exist a trade-off between limiting the amounts of hypoglycemia of the patient and limiting the occurrence of cardiovascular diseases later in the patient life.

Consequently, understanding the physiological counter-regulatory responses caused by hypoglycemia with respect to the usage of insulin would allow to improve the management

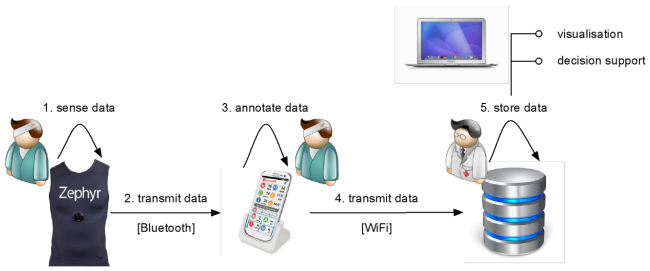

Figure 1. The *D1NAMO* platform

of hypoglycemic episodes. Also, observing the physiological values of a patient before, during and after occurrences of hypoglycemia would permit to have a better understanding of the phenomenon as well as to allow a non-invasive prediction of hypoglycemic episodes. Furthermore, being able to predict hypoglycemia given the level of activity of the patient during the day and the week would allow doctors to act preemptively toward the hypoglycemia. The use of the *D1NAMO* personal health system will allow to:

- Monitor the activity of the patient.
- Provide informative feedback to the patient and the doctor about the physical activities performed and their impact on diet and medication.
- Detecting symptoms of hypoglycemic attacks in order to provide early alerts to the doctors.
- The built models will permit to construct a platform helping patients and medical doctors to monitor the disease:

In the preliminary phase of the project, the Zephyr BioHarness 3 was chosen as device to acquire patient's physiological signals. This device monitors three variables: the Electrocardiogram (ECG), the breathing amplitude and 3D accelerations. It is also providing higher level aggregated information at a lower rate (1 per second or 1 per minute), such as the heart rate, the breathing rate, the posture, the activity level and some others. The setup of *D1NAMO* generates around 15 hours of data daily for each signal.

The platform we are developing for *D1NAMO* is composed of different entities. There is first the Zephyr BioHarness 3 sensor that is used to acquire physiological signals of the patient. This device is connected with Bluetooth to an Android application that collects the data continuously and stores it on the smartphone. Each five minutes (customizable), the application is sending the accumulated raw data to a server through a REST web service. The server analyzes the incoming data in order to detect abnormal events. It also stores the signals in a PostgreSQL database. Finally there is a web interface that allows to query the database, allowing the patient and his medical doctor to verify and analyze the patient's situation. Figure 1 offers a high-level picture of this solution.

In order to implement the intelligence in the server that

will detect the hypo- and hyper-glycemic events, some research has to be made to find which conjunction of machine learning and features brings the best results. To have relevant results, we started a framework to work with physiological signals. It has been created so it is possible to extend it to other time-series signals.

III. Presentation of the Framework

The platform can be seen as the equivalent of a series of Python scripts that are traditionally written by researchers when tackling a machine learning problem. There are parts for reading the data, applying some signal processing steps, some machine learning algorithms and outputting the results to files. The toolbox can be seen as a way to abstract some components of this process to allow the reuse of code and algorithms, while trying to apply some good machine learning practices such as cross-validation. It is also designed to avoid common mistakes, such as the models being trained on the testing set, or the features extracting knowledge from the testing data.

A. Libraries

The framework is built on top of several Python libraries that are widely adopted in the Python community. The base building block is *numpy* for all numerical computations. It is also used by most of the python packages, allowing an easy way to pass data along all of them.

In order to share the data among the whole framework, the *pandas*[4] package is used, offering a notion of `Dataframe` and a lot of functions to work on it. It is often compared to the `dataframe` of the *R* language, offering more or less equivalent features. `Dataframes` can be seen as two-dimensional arrays in which columns can be named. All sequences of data in the framework are passed as `Dataframes`, the columns being the different signals, the rows being the observations in the time-series.

The framework is using the *scikit-learn API* [6] for interacting with machine learning algorithms. This allows running most of the algorithms from the *scikit-learn* package, and to easily apply the parameters grid-search system that it provides. It is also possible to use other packages that offers wrappers for the *scikit-learn API*, such as for instance *keras*[5], a deep learning python package.

The plotting of results is made with *matplotlib* [11], a tool already used in the scientific community for creating plots. There is a packages called *seaborn*[6] built on top of it, which provides a higher-level interface for creating plots and customizing them.

B. Structure

The framework is composed of four Python sub-packages, as shown in Figure 2. The *core* package groups the central

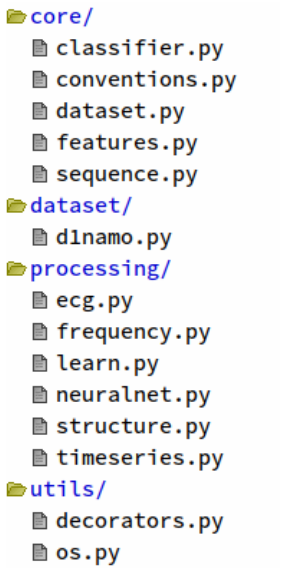

Figure 2. The structure of the framework

concepts of the framework. The naming conventions defined in `conventions.py` allow all parts of the code to know how to access a specific signal. Without a convention, some databases would name the columns "ECG" and some others "EcgWaveform". The convention defines a `Signal` enumeration providing for instance a `Signal.ECG` that any part of the code can use to refer to this signal. If a new type of signal needs to be supported, it should be added in the conventions. The `sequence.py` file provides an object representing a continuous observation of some physiological signals in a `Dataframe` format. It can be annotated with any key-value information, such as "patient=1203" or "class=hyper". The `dataset.py` file provides a Python object representing an entity grouping all sequences belonging to the same dataset. It also offers some filtering possibilities to access only sequences that have keywords that are fulfilling a given predicate. This is useful to access all sequences concerning a given patient, or corresponding to a given class. The `classifier.py` and `features.py` files both provide an abstract definition of what are the notions of `classifier` and `features extractors` in the framework. All algorithms that are children of these two classes can be used within the evaluations without extra work.

The *dataset* package is a logical entity to group the different wrappers that read a specific dataset on the disk. For now only *D1NAMO* is supported, and the `d1namo.py` file provides the wrapper that is reads the data from the files in the format used by the framework. Adding the support for a new dataset consists creating a new Python file and making it to read the data into `Sequences` objects, then grouping them inside a `Dataset` object. Thanks to this structure, it

[4]http://pandas.pydata.org/
[5]http://keras.io/
[6]https://stanford.edu/~mwaskom/software/seaborn/

is possible to use the new dataset with already existing code and algorithms.

The *processing* package provides functions and algorithms to work on the data. They are grouped by domains, such as *ecg*, *frequency*, *neural network*, *machine learning*, and so on. All functions in these files are working with `Sequence` objects (or `Datasets` more generally). This allows to use the already implemented functionalities and algorithms on new datasets easily, as all data are sharing the same structure. Functionalities should still check the conformity of the given data (like the existence of a specific signal for instance) as not all datasets provide the same set of signals.

Finally the *utils* package offers some unrelated helper functions. For now it has an `os` module that wraps some operating system functions in an easier application program interface (API). The `decorators.py` file provides some Python decorators that offer an easy way to add behaviours to functions, such as `@debug` that makes an interactive prompt to appear if a bug arises during the execution of the function, `@skip` that makes a given function to be skipped, or even `@pre` and `@post` to check for pre- and post-conditions.

C. Functionalities

The framework is then completed by a `run.py` script that uses the different components to run the evaluations. The pseudo-code of its main loop is the following:

```
data = load(dataset)
for classifier in classifiers:
  for extractor in features_extractors:
    for cv in cross_validations:
      train, test = split_train_test(cv, data)
      train_feat, test_feats = extractor(train, test)
      best_model = grid_search(classifier, train_feat)
      predicted = best_model(test_feat)
      scores.append(classifier,
                    extractor,
                    metrics(predicted, test))
```

The script first load the data with the Python object that has been created for the given dataset. Then it iterates through all the machine learning algorithms and features extractors given lists, in order to apply each algorithm on top of each extracted features. For this it first does a cross-validation loop, in which a parameters grid-search will be applied to select the best model. Once the model has been selected, the score of the current cross-validated iteration is kept in an list for being used in the reporting at the end.

The extraction of the features is not done at the level of the first loop. This would offer much better performances as it would be done only once for all classifiers, but doing it after the train/test split permits to avoid a typical machine learning error for some kind of features: if the features extraction includes some knowledge about the data, it should be done *only* on the train set. For instance if the feature is the distance from an observation to three centroids of the dataset, using the testing data to compute the centroids would already include some knowledge about them in the features, biasing the results.

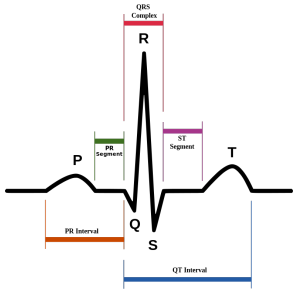

Figure 3. *PQRST* points of an ECG beat.

The builtin `random` library, as well as `numpy.random`, allows to set a seed for initializing the random number generator. Setting the seed at the beginning of an experiment permits to reproduce the results in a deterministic way. Being able to reproduce experiments is important for doing scientific research and also from the software development perspective. It allows to run again an experiment to debug it or to identify what was happening. This is only possible as long as there is no concurrency involved with thread of multiprocessing, because the order in which algorithms are accessing random number would change between subsequent runs.

The script supports some command-line flags, such as:

- `--debug` to get an interactive prompt in case of error. It simplifies the debugging as it is possible to directly inspect variables.
- `--cv n` to set the number of cross-validation to `n`.
- `--gs n` to set the number of cross-validation inside the parameters grid-search to `n`.
- `--seed n` to set the seed of the random number generator.

IV. Case Study

We present in this section our case study. Some scientific papers showed that ECG has information, which allows to predict hypoglycemia events. ECG measures the electrical activity of the heart and our sensor is specifically acquiring an `I`-lead ECG signal. This ECG is usually characterized by the *QRS* points and the *P* and *T* waves, as shown in the Figure 3. Several publications [12]–[14] showed a decrease in the *QT* interval during hypoglycemia periods with a medical setup. Being a sport-like chest belt, the contact of the electrode with the skin is sensitive to movements and therefore the quality of the signal is less good than an ECG acquired with medical devices. In order to verify if the ECG can be used with our setup to detect hypoglycemia, we need to test different features and different algorithms.

The *D1NAMO* dataset consists of two datasets, one acquired on healthy patients, and one acquired on diabetic patients with nearly the same setup. The data acquisition

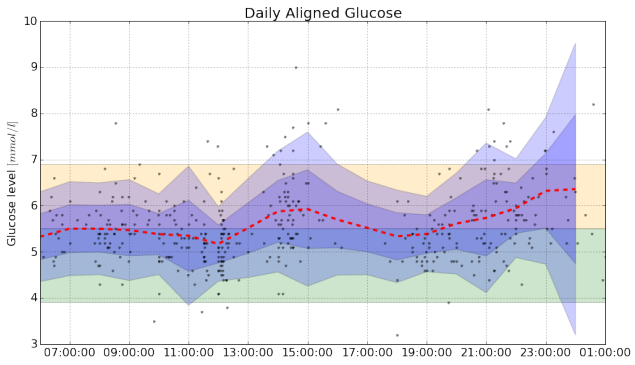

Figure 4. The daily *D1NAMO* glucose measurements

on the diabetic patients is not yet finished, we decided in the meantime to prepare our algorithms on the dataset of healthy people, which is already complete.

The healthy people dataset has been acquired on 20 patients, wearing the sensor for four consecutive days. The setups for healthy patients and diabetic patients are slightly different, as the glucose measurement is done with a continuous glucose monitoring for type 1 diabetic patients. Healthy people have measured their glucose level with a Bayer Contour XT six times a day. This difference would have implied some code change in a traditional workflow, but with our framework only the code dealing with the database needs to be created in order to reuse the existing code with exactly the same algorithms/features.

For the purpose of this study, we extracted 355 sequences of two minutes ECG around each glucose measurements of healthy people. All glucose measurements of all patients are shown in the Figure 4, the dotted red line being the mean, the blue areas the 75th and 95th percentile. We annotated all points being bigger than 6.9 as "hyper"-glycemia, all points being below 4.2 as "hypo"-glycemia, and all the others as "normal". Taking a 2 minutes signal around each glucose measurement represents 309 "normal" sequences, 38 "hyper" and 8 "hypo".

For this case study we implemented 3 types of features extractor that contains the *QT* interval information in different forms:

- Histograms of *QT* interval
- Histograms Derivative of the *QT* interval
- 10 bins Histogram of polynomial coefficient of *PQRST* fitting

The histograms are used with 2, 5, 10, 50 and 100 bins each, the polynomial fitting within orders from 1 to 8. All this together represents 18 different kinds of features. We also choose to use three machine learning Algorithms:

- K-Nearest neighbors (KNN)
- Support Vector Machine (SVM)
- Random Forest (RF)

The KNN is selected for its simplicity as a learning algorithm. The SVM and RF because a large-scale study [15] ranked these as the best algorithms and showed that both of these algorithms are averaging more than 90% of the best scores among a lot of classifiers in a large variety of classification tasks. They will thus give a good indication of what is possible to achieve with machine learning.

The dataset is loaded in the right format using the Python object that should be written independently for each dataset:

```
dataset = D1Dataset(D1PATH)
```

The definition of features extractors and machine learning algorithms to use is done by defining an array with all elements:

```
classifiers = [
  KNN(),
  RandomForest(),
  SVM()]
extractors = [
  algo(bins)
  for algo in [QTFeatureExtractor,
               QTDerivativeFeatureExtractor]
  for bins in [2, 5, 10, 50, 100]]
extractors += [
  PolynomialPQRSTFeaturesExtractor(x)
  for x in [1, 2, 3, 4, 5, 6, 7, 8]]
```

Once all these elements are defined, it is possible to use the `run_evaluations` function that will take care of applying the different features extractors and machine learning algorithms on the dataset. This function takes also care of running a cross-validation. The generic side of this function is possible thanks to the defined *APIs* and formats gluing the different parts of the framework together. The *prior* score is to compute the prior probability on the dataset in order to plot it in the report:

```
evaluations = run_evaluations(dataset,
                              classifiers,
                              extractors)
prior = compute_prior_scores(dataset)
report(evaluations, prior)
```

Running the scripts logs the output both to the screen and to a file named with the current timestamp in order to keep track of the previous runs. It also does the same in another file for the reported errors. The output also contains the initial random number generator seed, allowing to reproduce the experiment afterward in case of need. The output looks as follows:

```
SEED: 833596236
PRIOR: 87.04%

=====================================================
Random Forest Classifier
Polynomial PQRST features extractor with 3 deg, 10 bins
-----------------------------------------------------
Classification report

             precision    recall  f1-score   support

       hypo       0.00      0.00      0.00         8
      hyper       0.00      0.00      0.00        38
```

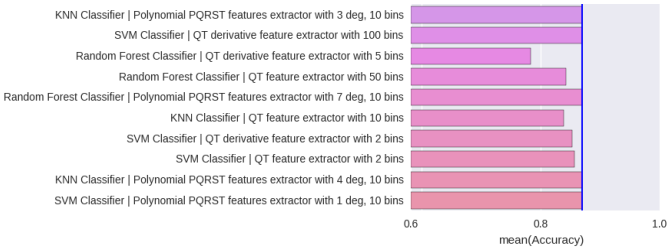

Figure 5. Part of the overview plot generated by the framework

```
     normal       0.87      1.00      0.93       309

avg / total       0.76      0.87      0.81       355

-------------------------------------------------------
Confusion matrix

[[  0   0   8]
 [  0   0  38]
 [  0   0 309]]
=======================================================
```

This is only the result of the first combination of classifiers and features extractors. The output contains which classifiers and features extractors were used, a classification report with some metrics (class by class and in average), and finally the confusion matrix. The rows and columns of the confusion matrix are the same as in the classification report. The full log contains one entry for each combination of classifiers and features extractors. At the end of the log there is a table, which summarizes the average accuracy, precision and recall of each one of them. This table is also written in a *CSV* file for an possible treatment later. The report generates a plot offering an easy to read overview of all combinations performance, as shown in Figure 5. In this plot, the blue line represents the prior probability, and each line represents an evaluated combination.

The full plot shows that no feature (whatever the selected machine learning algorithm is) outreaches the prior probability of the dataset. The evaluation with the Polynomial features extractors achieved the score of the prior by classifying all classes as "normal", the majority class. For the rest of the features, the higher the features dimension are (histogram of 50, 100), the better it seems to work. On this classification task with these particular features, the *RF* algorithm in general is less efficient than the *KNN*, which in turns is less efficient in general than the *SVM*. The small number of evaluations, 54 in this case study, does not allow to infer any general conclusion about the classifiers.

The fact that no setup allows to do better than the prior, in conjunction of the results of [15], lets us quickly know that the probability of having better results by playing with machine learning algorithms and/or parameters is fairly small, and that our features will probably not allow to classify hypo/hyper/normo-glycemia in the form as they currently are. Instead, other ways of improving the classification has to be used, such as applying better signal pre-processing to improve *ECGs*, by using accelerometers for instance.

V. Discussion

This publication is presenting a Python framework that offers a way to apply and compare machine learning algorithms and features extractors on top of physiological data. It is also taking care at applying good machine learning practices such as cross-validation and parameters grid-search, and ensures that some mistakes are not appearing, such as the classification labels kept in the testing data, or features including some knowledge about testing data.

Conceptually this toolbox does not differ from what scientists are used to do when they are working on a research problem including machine learning, which means writing hand-made scripts. But by abstracting the notion of a Datasets, by defining naming conventions for signals, by selecting an *API* for machine learning methods call and by choosing a common format for all data, this framework offers the possibility to write reusable machine learning tasks in a way that allows to apply exhaustive search over a set of algorithms and features. This also permits to write reusable code.

The presented case study shows that this framework may be used as a first analysis toolbox to get an idea about how well features are working. Later it can also be used to fine-tune some parameters of the algorithms or of the features.

In the future the framework can be extended to support more types of signals. This can be done easily by simply defining their names in the conventions. The real support for signals comes from the functions that are able to treat them. Extending the set of algorithms that support signals will be done while working on a dataset that requires them. The framework is missing unit testing for now. A refactoring and code cleanup of the framework would be needed before going further in releasing it under an open-source license. Some extensions of this framework would be useful, such as a support for parameters configuration files that would allow to define the classifiers, features, parameters space, without the need of coding them. This will also simplify the management of different setups.

Acknowledgment

This research has been financed by the *Nano-Tera.ch* initiative through the *D1NAMO* project.

References

[1] S. Bromuri, S. Puricel, R. Schumann, J. Krampf, J. Ruiz, and M. Schumacher, "An expert personal health system to monitor patients affected by gestational diabetes mellitus: A feasibility study," *in: Journal of Ambient Intelligence and Smart Environments, Ios Press*, 2015.

[2] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science and Engineering*, vol. 13.2, pp. 31–39, 2011.

[3] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio, "Theano: new features and speed improvements," Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[4] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Jun. 2010, oral Presentation.

[5] 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*, vol. 12, pp. 2825–2830, 2011.

[6] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[7] I. J. Goodfellow, D. Warde-Farley, P. Lamblin, V. Dumoulin, M. Mirza, R. Pascanu, J. Bergstra, F. Bastien, and Y. Bengio, "Pylearn2: a machine learning research library," *arXiv preprint arXiv:1308.4214*, 2013. [Online]. Available: http://arxiv.org/abs/1308.4214

[8] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 51 – 56.

[9] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[10] C. B. Smith, P. Choudhary, A. Pernet, D. Hopkins, and S. A. Amiel, "Hypoglycemia unawareness is associated with reduced adherence to therapeutic decisions in patients with type 1 diabetes evidence from a clinical audit," *Diabetes Care*, vol. 32, no. 7, pp. 1196–1198, 2009.

[11] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[12] T. F. Christensen, L. Tarnow, J. Randløv, L. Kristensen, J. Struijk, E. Eldrup, and O. K. Hejlesen, "Qt interval prolongation during spontaneous episodes of hypoglycaemia in type 1 diabetes: the impact of heart rate correction," *Diabetologia*, vol. 53, no. 9, pp. 2036–2041, 2010.

[13] T. Christensen, I. Lewinsky, L. Kristensen, J. Randlov, J. Poulsen, E. Eldrup, C. Pater, O. K. Hejlesen, and J. Struijk, "Qt interval prolongation during rapid fall in blood glucose in type i diabetes," in *Computers in Cardiology, 2007*. IEEE, 2007, pp. 345–348.

[14] G. Gruden, S. Giunti, F. Barutta, N. Chaturvedi, D. R. Witte, M. Tricarico, J. H. Fuller, P. C. Perin, and G. Bruno, "Qtc interval prolongation is independently associated with severe hypoglycemic attacks in type 1 diabetes from the eurodiab iddm complications study," *Diabetes care*, vol. 35, no. 1, pp. 125–127, 2012.

[15] M. Längkvist, L. Karlsson, and A. Loutfi, "A review of unsupervised feature learning and deep learning for time-series modeling," *Pattern Recognition Letters*, vol. 42, pp. 11–24, 2014.