Informatik 11
Embedded Software

RWTH AACHEN UNIVERSITY

Documentation

Novelty Detection Analysis System: Manual

Jan van Essen

Aachen, 18. Juli 2023

1 Installation and Start-Up

This section discusses the initial installation and setup of the software.

If you just want to use the software (instead of developing it), you can just start the mdas.exe and everything should work without any further set up.
If you want to develop the software, read the following installation instructions.
There are two options: You can either download and install the required python version and the required packages or you can use the provided conda environment to start the program.

1.1 Install the required python version and packages

The software is compatible with the following python versions: 3.6, 3.7, 3.8. Please download and install one of these versions first (`https://www.python.org/downloads/`).
If you already have another (incompatible) python version installed, please be careful which version you add to the PATH, and make sure that you start the NDAS using the compatible python version. In this case, it may be easier to use anaconda instead of installing different python versions at the same time (see section 1.2)
The main requirements arise from the use of Qt for the graphical user interface. Consequently, it is necessary to install PyQt5 as Python bindings for Qt5. Without these Python bindings it is not possible to run the software. Also, PyQtGraph is a main component of the software, as this graph library is used for visualization and plotting of the data. The analysis system at the time of this documentation has the following requirements:

```
pyqtgraph==0.11.1
numpy>=1.18.4
hickle==4.0.4
wfdb==3.2.0
scipy==1.5.3
pandas==1.0.3
PyQt5==5.15.4
PyYAML==5.4.1
qtwidgets==0.18
bs4==0.0.1
```

```
lxml==4.6.3
seaborn==0.11.1
scikit-learn==0.24.2
kneed == 0.7.0
humanfriendly==10.0
mysql==0.0.3
mysql-connector-python==8.0.28
paramiko==2.10.3
```

Listing 1.1: requirements.txt: Current requirements for the analysis software.

The software has been tested with the indicated versions, but may work fine with newer software versions of the listed packages. This is subject to testing. The current requirements can always be found in requirements.txt.

The installation of these packages can be done with the Python Package installer. For this, the requirements.txt can be read directly into `pip` and the required packages are installed automatically via the following command:

```
pip install -r PATH/TO/requirements.txt
```

Listing 1.2: Installation of requirements from the requirements.txt.

Another option is to just run the script install_dependencies.bat. It checks wether a compatible python version and all necessary packages are installed. Missing packages are installed automatically if needed.
The software has a start-up check. If not all required packages are installed, this is shown to the user in an error message.

No further configuration is necessary to start the software. The graphical user interface can be started by running `python ndas.py` from the console.

1.2 Start the program using Anaconda

If you don't want to install a compatible python version directly (e.g. if you already have a newer python version installed), you can use the provided anaconda environment to start the program. To do so, please follow these steps:

1. Download and install Anaconda (`https://www.anaconda.com/products/distribution`).
2. Open the anaconda powershell and navigate into the project folder (<your local path>\novelty-detection-analysis-system \NDAS)
3. Execute the following command: **conda env create -f ndas_environment.yml**

4. Execute the following command: **conda activate ndas_environment**
5. After this, you should be able to start the program using **python ndas.py**.

To deactivate the environment, execute the command **conda deactivate ndas_environment**.

2 Configuration

The analysis software has a configuration file in the config folder. In this config.yml file, various settings can be made, such as the dark mode or other visual settings. In the following these settings are presented.

The configuration file uses YAML[1] and thus has a fixed format which must be adhered to so that the file can still be read by the software. Most of the settings are self-explanatory. The settings are sorted by extension, because the respective extension gets the corresponding configurations in the initialization process during start-up. Additional labels can be defined that can be selected for annotation by adding strings to the labels section:

```yaml
annotation:
  labels:
    - "Condition"
    - "Sensor"
    - "Unknown"
```

Listing 2.1: List of available labels for annotation.

The stored physiological value ranges are also present in the configuration file and are read out by the `physiologicallimits` extension. Here, a lower limit and an upper limit must be specified for each physiological data type. The assignment is done via the identifiers of the table columns, therefore aliases can be specified. Columns that have these aliases or the main identifier of the physiological value can thus be identified and the value limit assigned. Below is the entry for the temperature:

```yaml
physiologicalinfo:
  temperature:
    low: 30
    high: 40
    aliases: ["temp", "t", "c"]
  ...
```

Listing 2.2: Physiological info for temperature.

The darkmode can also be activated via the configuration file. Further information can be found in the file.

[1] https://en.wikipedia.org/wiki/YAML

3 Adding new Algorithms

The analysis system can be extended with further algorithms. For this purpose, an autoloader was implemented, which automatically loads the algorithms available in the corresponding folder at program start up. Algorithms can then be selected via the graphical user interface without further configuration requirements.

To implement a new algorithm, a new class has to be created which inherits from the BaseDetector superclass. The algorithm class has to implement a `detect()` method that takes a pandas dataframe and returns a dictionary. The return values are structured as follows: The returned dictionary contains multiple other dictionaries, depending on the number of dimensions. The column name is used as identifier and therefore as key in the dictionary. The column-specific dictionaries contain the detected class for each individual data point of the column, as visualized below:

```
total_results =
{
 columnID1 = {timeID1 : c1t1class, timeID2: c1t2class...},
 columnID2 = {timeID1  : c2t1class, ...},
 ...
}
```

Listing 3.1: Structure of the return dictionary.

The available classes for the detection process are depicted below:

Tabelle 3.1: Return values for algorithms.

Value	Name	Color (Hex triplet)
-2	Ignored	Grey (#a9a9a9)
-1	Training	Turquoise (#00afbb)
0	Normal	Blue (#4e81bd)
1	Anomalous (Tier-I)	Red (#fc4e08)
2	Anomalous (Tier-II)	Yellow (#e7b800)

The following is a sample implementation of an algorithm. In this minimal example, every second data value is classified as an anomaly:

```python
from ndas.algorithms.basedetector import BaseDetector

class ExampleDetector(BaseDetector):

  def __init__(self, *args, **kwargs):
    super(ExampleDetector, self).__init__(*args, **kwargs)

  def detect(self, datasets, **kwargs) -> dict:
    total_results = {}

    for column in datasets.columns[1:]:
      column_results = {}

      i = 0
      for index, row in datasets[[datasets.columns[0], column]].iterrows():
          if i == 0:
              column_results[row[datasets.columns[0]]] = 1
              i = 1
          else:
              column_results[row[datasets.columns[0]]] = 0
              i = 0
      total_results[column] = column_results
    return total_results
```

Listing 3.2: example.py: Minimal example for an implemented algorithm.

When creating new algorithms, auxiliary functions can be used, which have been implemented in the superclass and are therefore directly available. Via `register_parameter` additional parameters can be registered, which are queried in the GUI before running the algorithm. Via `signal_error` an error message can be sent to the GUI, which is displayed to the user in a messagebox window. With `signal_percentage` the current percentage status for the progress bar can be sent. Additional plots can be submitted to the GUI using `signal_add_plot`, while additional line plots can be added to existing plots using `signal_add_line` and `signal_add_infinite_line`. A string can be sent to the logger via `log`. Via `get_physiological_information` the physiological value ranges can be requested from the config file for passed identifiers (e.g. temperature). Further information about the auxiliary functions can be found in the BaseDetector class.

4 Data format

In this section, I will describe the data format which is used to save the current state into a file.
The data is stored into a dictionary, which contains the following elements:

- **patientinformation** - information about the database where the patient comes from and its ID, if available. This is used for the functionality to load additional labels from a different file for the same patient, to check weither the two files represents really the same patient.
- **data** - a dictionary which contains the plot data. It consists of the following elements:
  - *dataframe* - This is a two dimensional numpy.ndarry which contains the actual plot data. The fist column represents the time line (i.e. the X axis of the plot), and the remaining columns represents the data for every parameter that are uses as Y values for the plot.
  - *dataframe_labels* - These are the titles for every column of the previous array.
  - *data_slider_start* - If the user sliced the data, the value of the left control element of the slider is stored here.
  - *dater_slicer_end* - If the user sliced the data, the value of the right control element of the slider is stored here.
  - *imputed_dataframe* - If the user used the imputation functionality, the new imputed data are stored here into a numpy.ndarray. The dataframe array is overwritten by this array as soon as the user presses the „apply results onto loaded data set“ button.
  - *mask_dataframe* - TBD
- **Novelties** - This is a dictionary, which contains information which point was marked as a novelty, if the user executed a novelty detection algorithm. Every key of the dictionary is a plotname and every value is a list which is as long as the according plot contains points. A one means, that the according point is marked as a novelty, and a zero means the opposite, i.e. the point is not marked as a novelty.

- **Labels** - This is an array which contains the information about every label which the user added via the annotation view. Every element of the array is a dictionary which consists information about the point coordinates, the plotname and the label itself.

When saving the data, this dictionary is serialized using pickle and then written into a file. Originally, the hickle package was used to serialize the dictionary, but this caused compatibility issues between the compiled NDAS version and the script version for whatever reasons. To avoid this, we switched to pickle. Older saved files should nevertheless still be loadable, because if loading using pickle fails, the NDAS tries to load the file using hickle, to maintain backward compatibility.