

Python in the enterprise

Machine-learning-based silicon sensor quality evaluation

Dawid Gerstel, Marcin Lis, Michał Labuz, Konrad Zuchniak

September 14, 2016

1 Introduction

The goal was to implement software for classifying silicon sensors as either good or bad ones. The k-nearest neighbours algorithm was to be used to assess a sensor's response based upon a few attributes in the so-called feature space. These, in turn, were to be obtained in a preprocessing from sensor's response histograms in the ROOT framework. This report summarises the project functionality, evolution, methodology used and results obtained.

2 Managing the project

At the beginning, the tasks were assigned to the authors of this report as follows:

task	person responsible
Analysis of the sensor's response histograms	all
Choosing specific ML algorithm	all
Histogramming the measurements in the feature space	Dawid G.
Extracting feature space attributes from the histograms	Dawid G.
ML algorithm implementation	Michał Ł. & Konrad Z.
Testing accuracy of the algorithm	all
Implementing graphical user interface	Marcin Ł.
Results analysis	all
Functionality testing and suggesting improvements	all

Table 1: Summary of the tasks assigned to the team members at the project's start

Naturally, the team did not strictly respect the above task division as oftentimes authors asked and helped each other rather spontaneously. The team used git repository and the project is accessible at <https://github.com/dgerstel/PITE-Sensor-Eval>.

3 Project overview – its evolution and dataflow

The input data come from the real silicon sensors of the LHCb's Vertex Locator. There are 42 sensors in total and each one has 2048 channels at which noise (with pedestal subtracted) had been observed producing 2-dimensional histograms as in the Fig. 1. We had been supplied with these data (`.../data/165526/VELODQM_165526_2015-10-12_18.05.29_NZS.root`) before the project started. Then, we created a ROOT macro `.../code/preprocessData.C` that scanned over the ROOT file, extracted the 2-dimensional distributions and projected them onto y-axis building 1-dimensional sensor noise distributions that are irrespective of its channels. Fig. 2 shows an exemplary such plot. Both types of histograms are saved in `.../fig/control_plots/`.

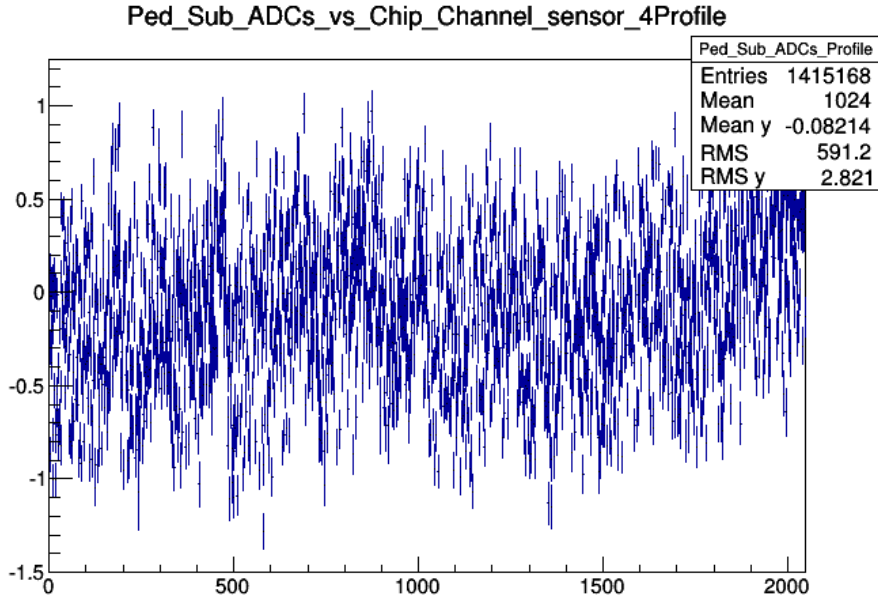


Figure 1: Mean noise after pedestal subtraction (y-axis) for subsequent channels (x-axis).

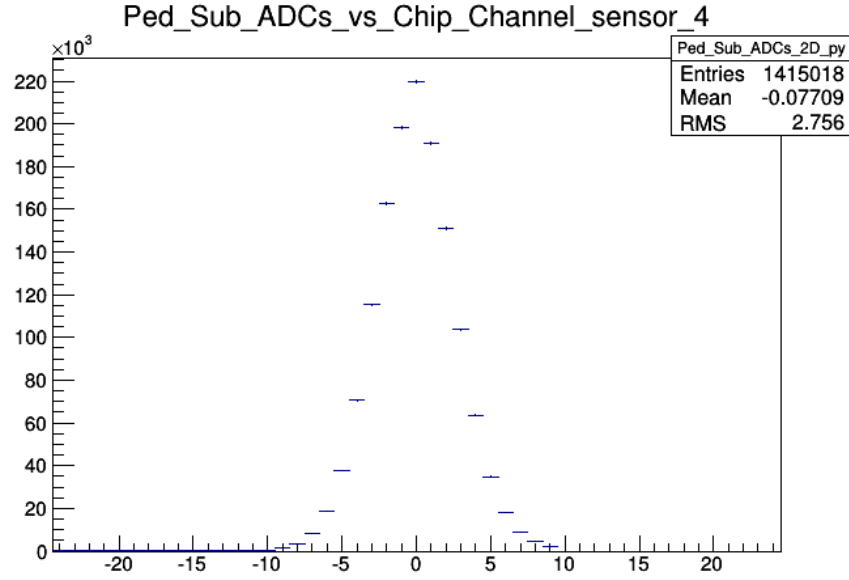


Figure 2: Frequency distribution of mean noise (x-axis).

Then, from these distributions four parameters were extracted, namely: mean, rms, skewness and kurtosis. These comprised the attributes for a machine learning algorithm. For any supervised learning it is mandatory to provide an adequate class label. We decided to approach classification as binary (either good or bad) with a perspective to develop a continuous and probabilistic classification to either category later on (if at all). The resulting data are stored in `.../data/sensorData.dat`.

After a brief analysis by eye we agreed that all 42 sensors are acceptable and we attributed them class '1' (good).

Next, because we were supplied with only good sensors – we had to simulate response of low-quality ones by shifting attributes of the 'good' ones. It is done by a function in `FakeData.py` ... The simulated low-quality sensor results are stored in files with prefix `.../data/danefake*`.

Then, we implemented k-nearest neighbours classifier (`.../code/Classifier.py`), taught the algorithm using the real sensor and simulated samples and, finally, we tested the classification accuracy. To this end, we used k-fold cross-validation described in the next section.

4 Methodology

Below we describe classes and functions used that come from the scikit-learn library¹.

`KNeighborsClassifier` is a class for classification based on nearest neighbours and it provides instance-based learning, as it does not attempt to construct a general internal model, but simply stores instances of the training data. Class constructor takes parameters as below:

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, weights='
    uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski',
    metric_params=None, n_jobs=1, **kwargs)
```

In this small project we used only a few of them:

- `n_neighbors` is the number of nearest neighbors which should be taken under consideration;
- `weights` parameter has two choices: 'uniform' and 'distance'. For the 'uniform' weight, each of the k neighbours has equal vote whatever its distance from the target point is. 'distance' weights points by the inverse of their distance. `weight` may also be a user defined function;
- `algorithm` parameter lets user to choose which algorithm should be used to compute the nearest neighbours; 'auto' means that class will attempt to decide the most appropriate one itself;
- `metric` decides how distances are calculated in parametric space. A general formulation of distance metric is 'minkowski' distance.

`KNeighborsClassifier` methods used in this project:

- `KNeighborsClassifier.fit(X,y)` fits the model using `X` as training data and `y` as target values, where `X` is an array-like object;
- `KNeighborsClassifier.predict(X)` predicts the class labels for the provided data, where `X` is an array-like object;
- `KNeighborsClassifier.predict_proba(X)` returns probability estimates for the test data `X`, where `X` is an array-like object.

To evaluate classification accuracy we used k-fold cross-validation, which is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set. It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice. The scikit-learn library for cross-validation provides class named `cross_validation` which has 2 methods which we used in the project:

- `cross_validation.ShuffleSplit` random permutation cross-validation iterator. Does not guarantee that all folds will be different. However, this is still very likely for sizeable datasets;
- `cross_validation.cross_val_score` evaluates score by cross-validation.

5 Analysis

With the application there are two different types of analyses possible to perform. First of all, we may look at histograms of sensor's noise parameters, like for instance mean and skewness distributions that are depicted in Fig. 3 and Fig. 4, respectively. The second type allows for evaluating knn algorithm output. Fig. ?? represents classification accuracy as a function of number of neighbours for sample split according to various train-to-test-ratios. This example shows that for splitting the sample into equal halves for training and test we obtained highest – 100% – accuracy.

¹<http://scikit-learn.org/stable/>

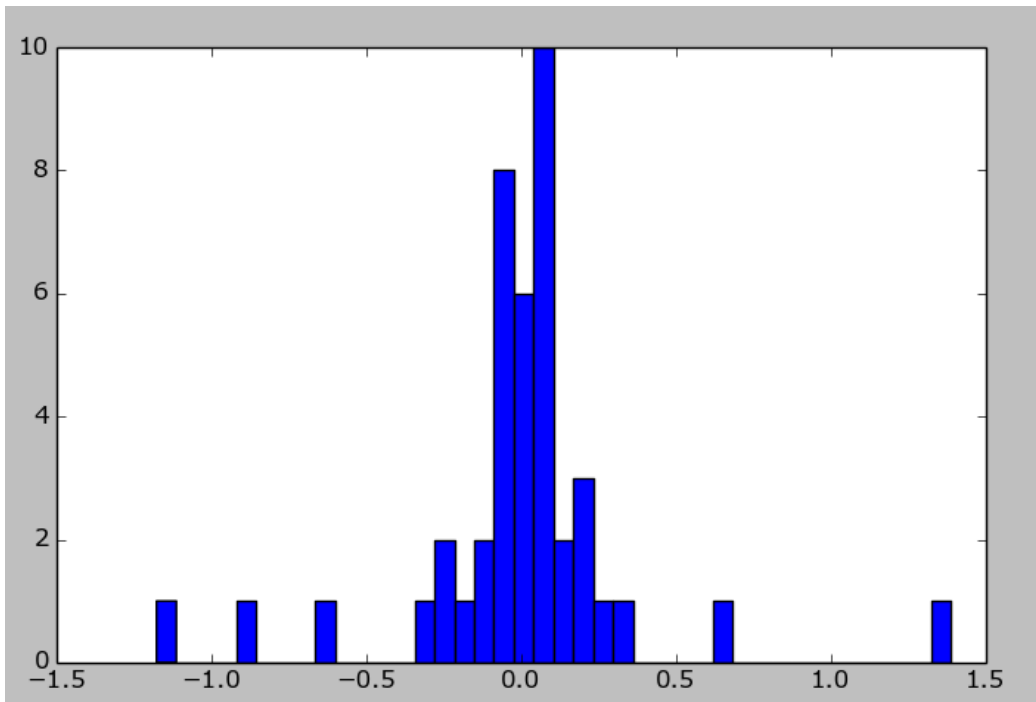


Figure 3: Average parameter histogram.

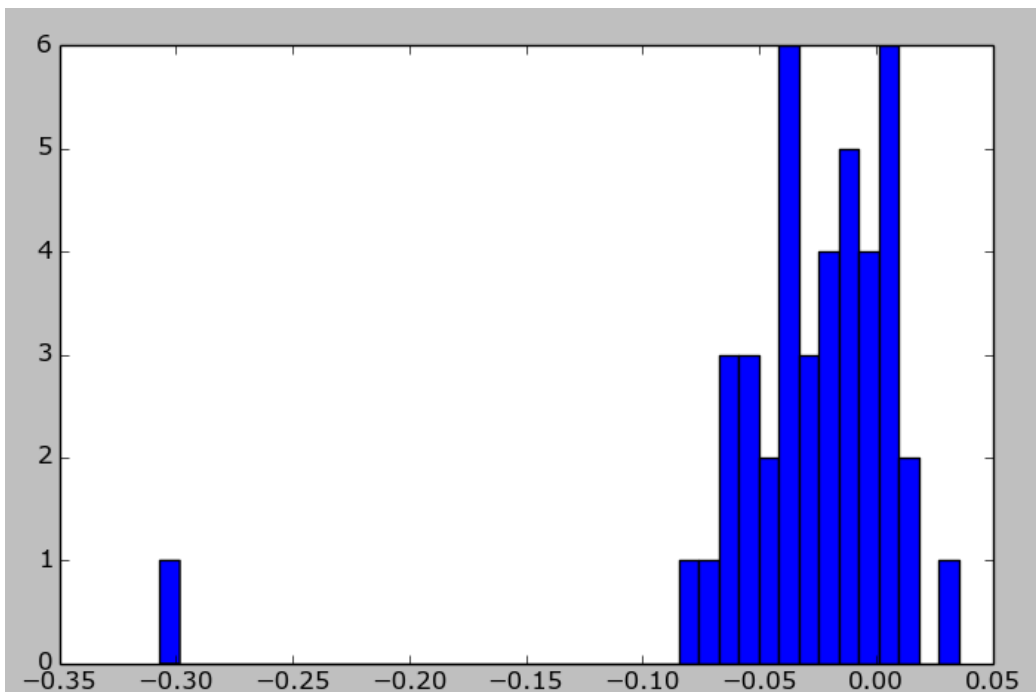


Figure 4: Skewness parameter histogram.

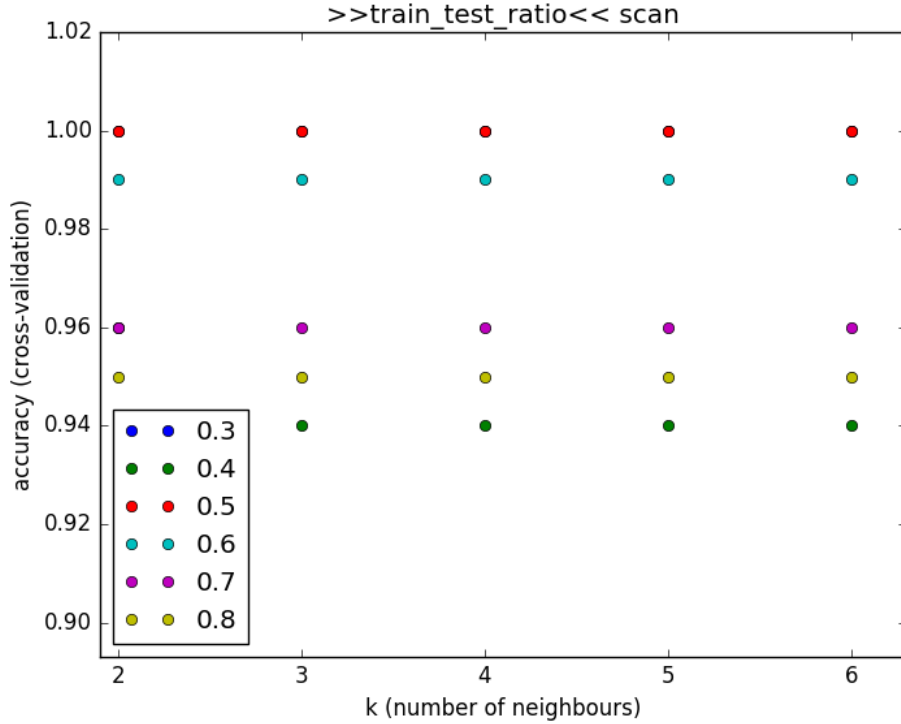


Figure 5: An exemplary analysis plot showing accuracy as a function of number of neighbours for various train-to-test ratios.

6 Summary

Python in the enterprise, as the name suggests, is a subject oriented on utilising Python as a tool for delivering a product of, usually, a team enterprise, whether for business or scientific purposes. The latter motivation characterised this project. Its completion let us develop both – hard and soft skills. Apart from practising and reviewing the programming language itself, the following traits of team work we would like to emphasise. First of all, we had to come up with clear division of responsibilities and organisation of working time. Team work made us realise how important reliable code documentation and control of project progress is. Additionally we understood purpose of using revision control system, like in our case *git*. Some functions in our code were written, at least in part, by all of us. Thanks to that we could compare the way we think, observe and compare various programming styles and learn something new.

As far as hard skills are concerned, we had a chance to get to know basics of Machine Learning in Python. Particularly well we familiarised ourselves with k nearest neighbours algorithm and `KNeighborsClassifier` class from scikit-learn module. To understand classification based on KNN we had to read thoroughly documentation of the class and learn its methods and attributes.

Finally, we managed to complete all the plans and built models that help distinguish low and high quality silicon sensors based on their noise analysis and machine learning, what is an evidence of our good work organisation.

A Application guide

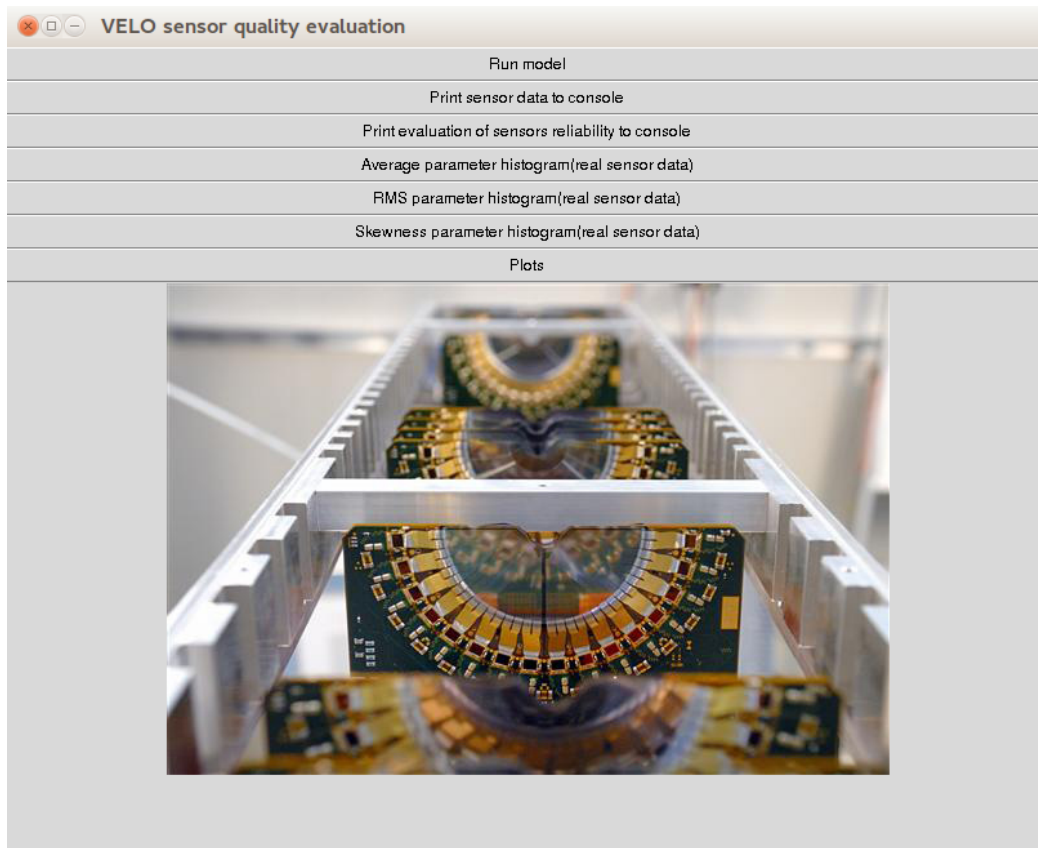


Figure 6: Application GUI after start

After starting the application with invoking `python __init__.py` from the directory `.../code` and clicking **Start** button the above screen pops up.

- **Run model** – Enables to run KNN classification of all data from `.../data` directory; it also allows to generate simulated data.
- **Print sensor data to console** - Displays content of sensor data file with all its parameters and class label.
- **print evaluation of sensors reliable to console** - shows output in console and saves it to the log files.
- **Average/RMS/Skewness parameter histogram** - displays histograms of corresponding parameters of a real sensor.
- **Plots** - Contains analysis plots of accuracy as a function of number of neighbours for various parameter values.

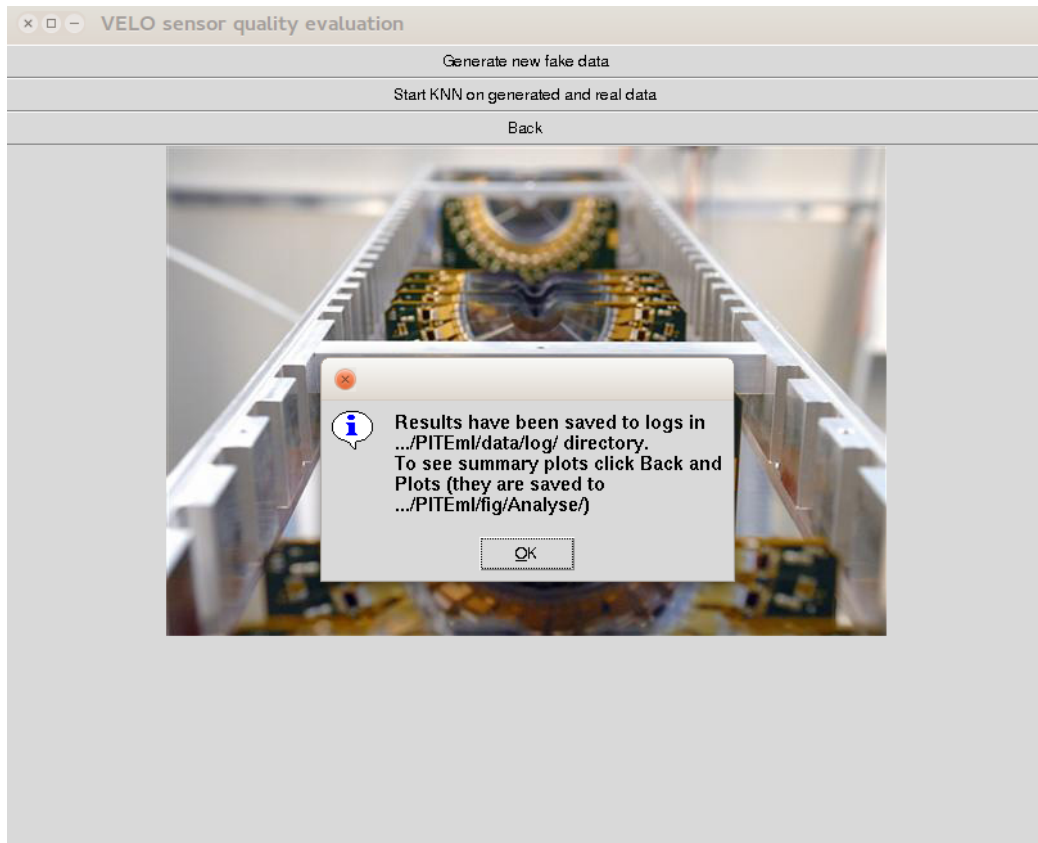


Figure 7: Application GUI after choosing Run model from the initial window.

- **Generate new fake data** - Generates new simulated sensor data of departure from real sensor parameters as chosen by user.
- **Start KNN on real and generated data** - Launches KNN algorithm and produces output in log files as well as plots.

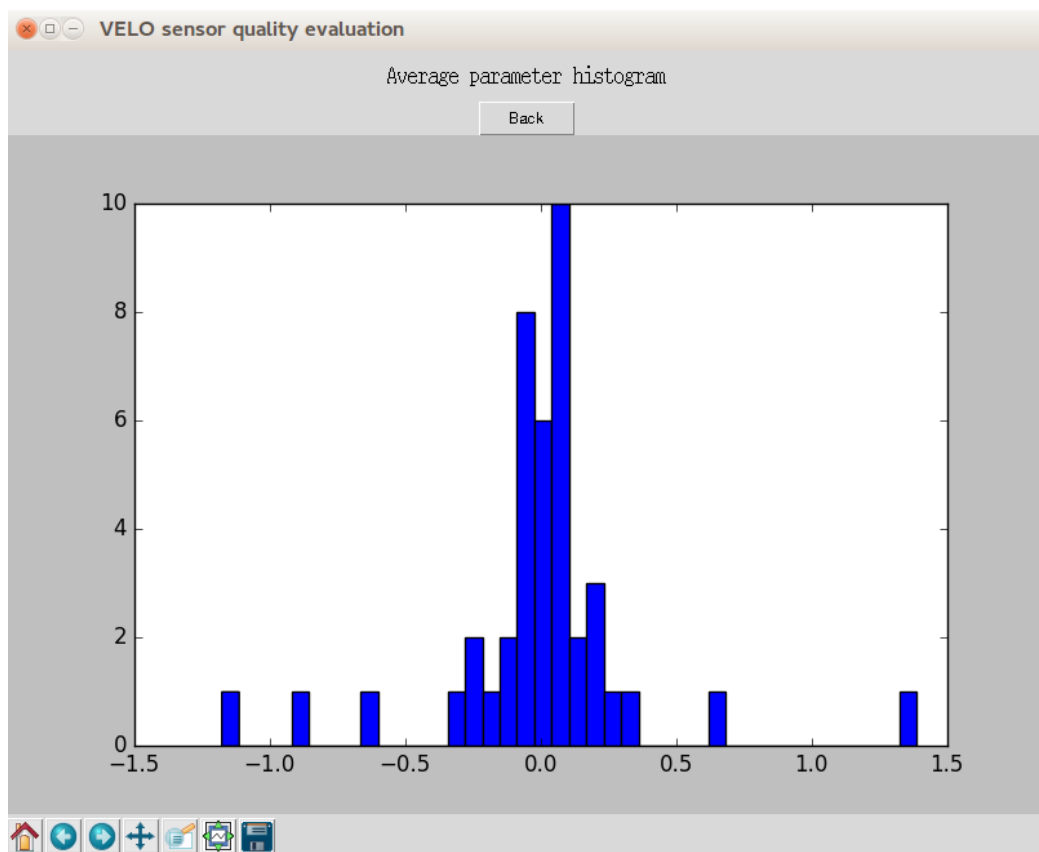


Figure 8: An example of a real sensor histogram.