**Charles University**

**Faculty of Mathematics and Physics**

**BACHELOR THESIS**

**2021 Tomáš Čelko**



**BACHELOR THESIS**

Tomáš Čelko

**Support for annotating and classifying particles detected by TimePix3**

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: RNDr. František Mráz, CSc.

Study programme: Computer Science

Specialization: IOI

Prague 2021

[Sample: Bound Sheet – a signed copy of the "master thesis assignment". **This assignment is NOT a part of the electronic version of the thesis. DO NOT SCAN**.]

I declare that I carried out this bachelor thesis independently and only with the cited sources, literature, and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…...... date............ signature

[Sample: Here you may thank whoever you wish (the supervisor of the thesis, the consultant, the person who lent the software, literature, etc.)]

Title: **Support for annotating and classifying particles detected by TimePix3**

Author: Tomáš Čelko

Department / Institute: Department of Software and Computer Science Education Supervisor of the master thesis: First and last name with titles, workplace

Abstract:

Hybrid pixel detectors like TimePix3 can capture gigabytes of data of various particles in a second. However, in ATLAS[] measurements a vast majority of these particles represent already well known particles. Distinguishing between the types of the particles is the first step in finding extraordinary particles. It is a non-trivial task often done by physicists.

Manual processing of the data with such proportions is inefficient. This is why we created a tool which enable physicists to first filter the clusters based on their properties. Apart from trivial properties of a cluster (eg. Pixel count or total energy) we used the methods like convex hull calculation, skeletonization and breadth-first search to analyze more complex features.

Using features of clusters we designed an application capable of training a neural network classifier according to the given parameters. These trained classifiers can be merged into a tree-resembling structure, offering a better utilization of unevenly distributed clusters in the training process. We trained such model and used it on the real ATLAS[] data.

As a result data was split into classes according to the prediction and a visualization tool was implemented to enable consequential manual analysis of the desired categories of clusters .

Keywords: Cluster classification Timepix3 feature

Contents

[Introduction / Preface 3](#_Toc72421572)

[1 Analysis 5](#_Toc72421573)

[1.1 The Medipix family and the TimePix3 detector 5](#_Toc72421574)

[1.2 Input format and calibration 6](#_Toc72421575)

[1.3 A little about neighboring 8](#_Toc72421576)

[1.4 Calculating the features 9](#_Toc72421577)

[1.5 Cluster visualization 12](#_Toc72421578)

[1.6 Classification 18](#_Toc72421579)

[2 Goals of the thesis 21](#_Toc72421580)

[3 Cluster processor applications 22](#_Toc72421581)

[3.1 Specification 22](#_Toc72421582)

[3.2 Cluster Viewer 23](#_Toc72421583)

[3.3 Cluster Filter 27](#_Toc72421584)

[3.4 Description Generator 28](#_Toc72421585)

[3.5 Classifier aplications 31](#_Toc72421586)

[4 Development documentation 36](#_Toc72421587)

[4.1 Main objects 37](#_Toc72421588)

[4.2 Cluster Viewer 37](#_Toc72421589)

[4.3 Filter and Description Generator 38](#_Toc72421590)

[4.4 Classifier aplications 41](#_Toc72421591)

[5 Experiment 43](#_Toc72421592)

[5.1 Classifier Parametrization 44](#_Toc72421593)

[5.2 K-Fold cross-validation of single-layered models 46](#_Toc72421594)

[5.3 Multi-layered classifier 47](#_Toc72421595)

[6 Epilogue / Conclusion 51](#_Toc72421596)

# Introduction / Preface

In nuclear physics, there have been many efforts to visualize and detect elementary particles. For this purpose, various detectors were invented. Several such detectors are now members of the so-called Medipix detector family. The most recent member of the family is called the TimePix3 detector. In terms of elementary particle detection, TimePix3 achieves state-of-art performance. During a specified timeframe, the detector captures a set of particle trajectories. However, in some cases, a single particle emits the secondary particles that can interact. Instead of detecting the particles one by one, we analyze groups of such particles – so-called clusters.

So far, there have not been many publications dealing with the processing, filtering, visualization, or classification of these clusters. In the filtering task, one needs to make sure the algorithm is fast because the size of the cluster dataset captured by TimePix3 detector can reach gigabytes of data over a short timeframe. The classification process can be challenging because the trajectory of the cluster depends on the angle at which the particle enters the field of the detector. Furthermore, the distribution of various types of clusters in the standard observation is usually very uneven. For instance, most of the data received from detectors like ATLAS [link] contains only simple traces consisting of a few pixels and does not provide much information for the analysis. This fact causes problems for many machine-learning-based approaches because the datasets of the rare and more complicated clusters often have a minimal size.

**Goals of the thesis**

Our main goal is to create a set of processing tools that would enable physicists to analyze the clusters and their properties. The input will be a collection of cluster from the Clusterer application []. Firstly we need to provide support for filtering the clusters based on their attributes. Secondly, we visualize them individually so that the users can see the cluster as a 2D and 3D image. Because the number of clusters in some datasets can be overwhelming, another goal is to make a tool, which could calculate the properties for the whole collection of clusters. The calculated properties of the clusters should enable to design a neural network-based classifier capable of classifying various clusters, which is our final goal. Eventually, the developed classifier could also detect extraordinary clusters, displaying exotic or even unseen particles.

**Thesis layout**

First, in the [Analysis] chapter, we will introduce TimePix3 detector as a member of the Medipix detector family. Then, we will go over the data format, and define various features of a cluster and the methods of their calculation. After we are familiar with the features, we discuss the classification task and its difficulties. When we are done with the analysis part, we specify our goals in a short chapter [Goals of the thesis]. Based on these goals we implement a solution which is further described in the [Cluster processor applications] chapter. This section contains elementary information about all of the applications that were created. For a more-detailed and technical details about the solution we created a separate chapter – [Development documentation]. Because the classification proces can be considered non-deterministic, we will provide a brief statistical analysis of the results in the section [Experiment].

# Analysis

In this chapter we discuss the detection of elementary particles with hybrid pixel detectors, as well as the tools and methods suitable for the analysis of the detector output. Firstly, we introduce the Medipix detector family, then we present the format of the data from the Timepix3 detector. We also introduce new types of neighboring relation for the pixels. Then we move to the visualization of the Timepix3 data and last but not least we discuss the classification methods for the groups of elementary particles called clusters.

## The Medipix family and the TimePix3 detector

It was the 1990's when some of the researchers from CERN came with an idea to transfer the devices, primarily developed for experiments in Large Hadron Collider(LHC) beyond particle physics. The first collaboration with such a goal started under the name Medipix1. It was the collaboration of the University of Freiburg, University of Glasgow, and Napoli and Pisa Universities together with CERN. So far, there have been four Medipix collaborations, each with its own specific goals. The chips developed in these collaborations are known as the members of the Medipix detector family.

The first member of the family was the Medipix1 chip (1997), consisting of 64 x 64 pixels acting similarly to an digital camera – counting the hits of elementary particles while the shutter is open. A few years later, the Medipix2 chip was developed, leading to the first Timepix chip (2006) being invented. This was the first chip that can be programmed to record one of the following properties:

* **Particle hit count:**  The number of hits detected by the pixel in one tick of the internal clock. (similar to Medipix1)
* **Time over the threshold:** Each pixel is assigned an energy threshold level(THL). When a charged particle approaches the pixel, the energy captured by the pixel rises. The time interval while the energy remains above the threshold we call the time over the threshold (ToT). This attribute is often measured as the number of ticks of a detector clock.
* **Time of the arrival:** – The absolute time since the start of measurement until the energy threshold level is reached we call the time of arrival (ToA).

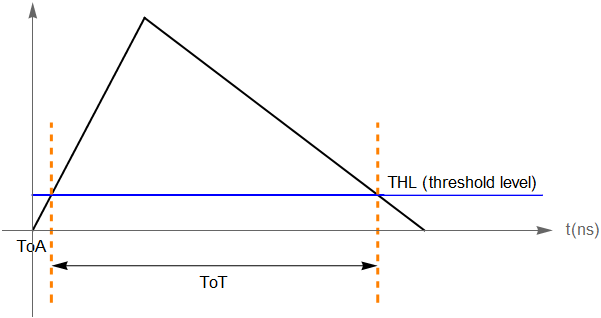


Figure 1.1 Time of arrival and time over the threshold visualization

In 2013 the new TimePix3 chip was introduced to the family. "Timepix3 is a general-purpose integrated circuit suitable for readout of both semiconductor detectors and gas-filled detectors. Compared to its predecessor Timepix the circuit has more functionality, better time resolution, and more advanced architecture for continuous sparse data readout with zero-suppression." [1] Zero-suppression means that there is no data output from the detector unless nonzero energy input is captured. This allows better efficiency in data collecting and storage but also in data analysis. The device utilizes a 256x256 pixel matrix where the size of each pixel is 55μm, achieving a time resolution of 1.56ns. The TimePix3 chip is nowadays used in the CERN LHC to detect sets of elementary particles, so-called clusters.

## Input format and calibration

Because the output of the TimePix3 chip over some timeframe can contain multiple clusters, we use the Clusterer application [4] to process the raw input from the detector and separate the clusters from each other. This means that instead of processing the data in the raw format, we use the data in the MM (clustered) format. The MM data format consists of three text files – ini, cl, and px file.

* Ini file contains the name of the measurement and the names of the cl and px files associated with the given measurement. By default, this file expects to find the cl and px file in the same directory where the ini file is located
* Cl file consists of the primary data about the cluster collection, where each line represents a single cluster. A line contains the first time of arrival of the cluster in ns (since the start of the measurement). This is followed by the size of a cluster and the number of the line in the px file (and the number of the byte) where the cluster starts.
* Px file includes data about each pixel in the cluster, namely and coordinates of the pixel, the time of arrival in ns, and the energy in keV.

|  |  |  |
| --- | --- | --- |
| **File** | **Format** | **Example** |
| ini | [Measurement (or any string ending with a newline char)] PxFile=[Relative path from the parent direcotry of .ini to px file]  ClFile=[Relative path from the parent direcotry of .ini to cl file] | Measurement 123 PxFile= Clusters\_px.txt ClFile= Clusters\_cl.txt |
| cl | [First ToA (float)] [Pixel Hit Count ( integer 0-232)] [LineOfStart in px file ( integer 0-232)] [Byte of start in px file (integer 0-232)] | 12345.647 100 5 30 |
| px | [x coordinate of the pixel ( integer 0-255)] [y coordinate of the pixel (integer 0-255)] [ToA (decimal)] [Energy (decimal)] | 123 128 15540 14.235 |

Table 1.1 Input MM file format

There are two kinds of MM formats, which only differ in the px file:

* ***Calibrated*** data have the structure as displayed in the chart [Table 1.1]
* ***Non-calibrated*** data are very similar to the calibrated ones, but instead of the energy attribute in .px file, they use the attribute

To determine whether a file is calibrated or non-calibrated, we can open the px file and look for the last column. If the decimal part of the values is zero in every pixel, we know we are dealing with non-calibrated data because the is measured as a number of ticks of the detector, which is an integral value. However, if the values have a nontrivial decimal part that indicates the data is already calibrated.

The calibration is the process of replacing the attribute with the corresponding energy. The energy deposited in the pixel is a function of the but also four other parameters of a pixel, denoted by the letters , , , and . These parameters are set during the measurement and are usually stored in separate text files as a 256x256 matrix of decimal numbers. It holds:

In the equation, *E* represents the energy deposited in the pixel, with *a*, *b*, *c,* and *t* being the calibration parameters.

## A little about neighboring

Neighboring of the pixels is an essential term when it comes to 2D image analysis. In general, two pixels are called neighbors if there is a relatively small distance between them. To be more specific, there are two known kinds of neighbors:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

* ***8-neighbors of a pixel p***

Figure 1.2 8-neighbours of a pixel p (green color)

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

* ***4-neighbors of a pixel p***

Figure 1.3 4-neighbours of a pixel p (green color)

And for future use, we will also introduce another kind of neighbors:

* ***Y-neighbors of a pixel p***

This kind of neighbors has two main variants:

Variant AVariant B

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Figure 1.4 Y-neighbours - variant A (green color)

Figure 1.5 Y-neighbours - variant B (green color)

For each variant, we also consider their symmetric alternatives to

belong to the same variant.

## Calculating the features

A cluster can be defined as a set of pixels, each containing information about its and coordinates, and energy. Apart from these explicit features, we also analyze the implicit features of the cluster like pixel count and the cluster’s total energy. This is the critical point of all our following work and also the main topic of this subchapter. We briefly discuss the features and the information they indicate about the cluster. The properties we can calculate range from straightforward to more sophisticated. The most important features are the following:

* ***Total, average and maximum energy, the standard deviation of energy,*** and ***low energy pixels count***provide information about the energy distribution of pixels in a cluster. The total energy of a cluster is defined as a sum of the energies of each pixel in the cluster. A pixel is considered to have low energy if its energy is less than 10keV.
* ***Pixel count*** attribute reflects the size of a cluster
* ***Width(diameter)*** and ***convexity*** attributes are both based on the convex hull of a cluster. Let denote the vertices of the convex hull of a cluster. Then the convexity is defined as follows:

The convexity of a cluster provides us the information about its shape because the more complicated clusters usually tend to have a concave shape. In contrast, the simple ones are often more convex. Width, also commonly know as the diameter, is defined as where *d* represents the distance between given vertices. For the function *d,* we chose to use the Euclidian distance in a plane: where is a pixel with the coordinates and .

* ***The standard deviation of the ToA*** captures the information about the timespan of a cluster. Another option would be to use the difference of the . This, however, has a disadvantage because in a cluster, there are often a few pixels that show extreme values of ToA. The standard deviation is considered to be "more resistant" to these outliers pixels and provides a reasonable estimate of the timespan of the cluster.
* ***Vertex count, Crosspoint count,*** and ***Branch Count*** reflect the possible number of particles in a cluster and the shapes of the trajectories in the cluster. A pixel is considered to be a vertex if it only has one neighboring pixel. Crosspoints are the pixels where the trajectories of different particles meet. These pixels we find as the ones with nonzero energy and with three or more 4-neighbors or 3 Y-neighbours with nonzero energy (both variants of Y-neighbors are possible). Crosspoints of a set are denoted by , and are displayed on the image (pink color) [Figure 1.6].

Figure 1.6 Crosspoints of a cluster (pink color)

Because we do the branch analysis iteratively we define the term branch found at the k-th iteration instead of a simple branch. A branch found at the *k*th iteration is a set of pixels defined as follows: Let *P* denote a set of pixels in a cluster *C*. Let denote a graph such that there exists a bijection *b* between *P* and *V*. Furthermore, for each pair of 8 - neighboring pixels in the cluster there is a corresponding edge in *.* Then we say is a branch found at the k-th iteration if and only if its corresponding graph is a path, has the maximal possible size, and at least one of the following conditions holds:

1. (base step of the definition), or
2. none of the pixels in are part of any of the previousbranches in (inductive step of the definition)

## Cluster visualization

The clusters we obtain from the Clusterer [] can be represented as a set of pixels, each with its own value of and energy. These pixels can then be visualized on a 256x256 bitmap, where a pixel is assigned a color according to the value of the pixels attribute. In this subchapter we introduce some methods for further cluster analysis based on its 2D image. These methods include skeletonization, z-coordinate calculation, finding the center of a cluster and branch analysis.

*Skeletonization*

Skeletonization of a binary image is defined as the thinning process that outputs a simpler version of the original image, the so-called skeleton. An important requirement for skeletonization is to have the image in binary format – each pixel is either white or black. Because the image of a cluster is not binary, we assign the value 1 to each pixel with nonzero energy and a value 0 to a pixel with zero energy. The skeleton should preserve the original shape of an image. In fact, the definition of skeletonization is not exact, which means various approaches can be used, each possibly outputting a unique skeleton.



Figure 1.7 Different skeletons of the letters B, N and Z

For this purpose, we decided to modify Zhan-Shuen's algorithm for thinning binary digital patterns [6] in the following way:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| 1 |  |  | |
| 2. |  |  | |
| 3 |  |  | |
| 4. |  |  | |
| 5. | If condition(S,p) and p.energy<t |  | |
| 6. |  |  | |
| 7. |  |  | |
| 8. |  |  | |
| 9. |  |  | |
| 10. |  |  | |
| 11. |  |  | |
| 12. |  |  | |
| 13. |  |  | |

Algorithm 1.1 Skeletonization of a cluster

|  |  |
| --- | --- |
|  | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | as displayed in the figure [Figure 1.8] |
|  |  |
|  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Table 1.2Description of functions and variables used in the Algorithm 1.

Figure 1.8 Example of pixel p with on indices (0,1), (3,4), (5,6)

The whole skeletonization process consists of using the twice. First time with value set to 10 keV and then we repeat the process with set to infinity. As a result, the first iteration aims to remove the halo effect of the low energy pixels while still preserving the shape. The second iteration with behaves the same as the original version of Zhan-Shuen's algorithm. A different approach would be to filter the low-energy pixels right away, but that does not guarantee that the cluster remains connected, which is a problem for future branch analysis. Because the thinning process returns nontrivial data for bigger clusters, we optimized the algorithm for those (~70-1000 pixels).

*3D reconstruction*

Because the Timpix3 detector is capable of capturing the particle energy, we are able to reconstruct its trajectory in 2D. However, Timepix3 also stores data about the time of arrival which can be used to estimate the z-coordinate of the particle. To provide the user with a 3D trajectory image, we need to calculate the z-coordinate. The z coordinate is a function of the relative time of arrival , where  is the difference between the arrival time of the particular pixel and the minimum time of arrival in the entire cluster. It holds:

[1]

Parameters used are the following:

*  - depletion voltage that is the minimum voltage at which the bulk of the sensor is fully depleted
* - bias voltage, which is the amount of voltage that a detector needs in order order to function [13]
*  - electrical mobility of a hole
*  - thickness of the sensor

All parameters are specified at the beginning of the measurement and remain constant for the whole duration of the measurement.

*Center Finder*

When analyzing the shape of a cluster, it could be useful to know where the center of the event is. The center is usually the point with high energy located at the intersection of the visible trajectories (if there are any). If we manage to find this point correctly, we could then start analyzing each trajectory starting from the core pixel of a cluster event. However, this task proved quite challenging, especially for the complicated clusters, because there can be multiple points with the high energy level lying at the intersection of the trajectories. For this task, we proposed an algorithm based on the energy of the pixel and its surrounding pixels: Let C denote an arbitrary set of pixels and let  be the set of pixels surrounding the pixel (possibly where can be the Euclidian distance and ). Then, we define the weighted surrounding energy of a pixel with the weight as with being the energy of a neighboring pixel . For each pixel, the surrounding energy is used to calculate the center cost function.

Let represent the set of pixels of a cluster and let denote its skeletonized version. The center cost function is defined as follows:

In our algorithm, we set . Then the center pixel of a set of pixels is computed simply as the

*Branch Analyzer*

Before discussing the branch analysis, let us once again mention the definition from We say the branch  is a subbranch of the branch , if it starts in one of the crosspoints of branch b. To analyze the branches, we proposed the following algorithm:

|  |  |  |
| --- | --- | --- |
|  | | |
| 1. |  |  |
| 2. |  |  |
| 3. |  |  |
| 4. |  |  |
| 5. |  |  |
| 6. |  |  |
| 7 |  |  |
| 8. |  |  |
| 9. |  |  |
| 10. |  |  |
| 11. |  |  |
| 12. |  |  |
| 13. |  |  |
| 14. |  |  |

Algorithm 1.2 Branch analysis

|  |  |  |
| --- | --- | --- |
|  | | |
| 0. |  |  |
| 1. |  |  |
| 2. |  |  |
| 3. |  |  |
| 4. |  |  |
| 5. |  |  |
| 6. |  |  |
| 6. |  |  |
| 7. |  |  |
| 8. |  |  |

Algorithm 1.3 Get Branch function

|  |  |
| --- | --- |
|  | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Figure 1.9 variables and function used in branch analysis algorithms

## Classification

After we are able to visualize clusters and analyze their properties, we can start focusing on subsequent classification. Firstly we briefly analyze the work done in terms of cluster classification. Then, we discuss the problems and the choice of the classifier model. Another topic we examine is generation of the training data.

Classification of a cluster is a task where we are given a cluster and a set of possible classes. Based on the cluster, we should predict to which class the cluster belongs. Ideally, we also want to estimate how sure we are about our prediction and possibly return the result "unclassified" if we are not confident about the prediction.

So far, there have been a few attempts to classify clusters – one example being the work Detecting elementary particles with Timepix3 detector[]. In this work, the clusters were divided into classes based primarily on their shape. The categories used in the thesis were the following:

|  |  |
| --- | --- |
| * Dot | * Heavy Blob |
| * Long gamma | * Straight track |
| * Heavy track | * Curly track |

Table 1.3 Categories of clusters based on their shape

Even though this classification provides valuable information about the shape of a cluster, mapping these categories to the real particle examples is still nontrivial. For instance, one particle can have a dot shape when it traverses the detector perpendicularly. In contrast, if the same particle enters the detector's field in a direction parallel with the orientation of sensors, it could leave a straight or even a curly track, as shown in [Figure 1.10].

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Figure 1.10 Possible 2D visualizations of the electron

The categories we decided to use for classification were based on the training data we were provided by Mr. Declan Garvey, working at the Institute of Experimental and Applied Physics of the Czech technical university. The classes of the particles in the data were the following:

|  |  |
| --- | --- |
| * Electron | * Pion |
| * Muon | * Proton |
| * Helium | * Fragment |
| * Iron | * Lead |

Table 1.4 Types of the clusters for classification with their examples

Because it seemed to be very difficult to manually set the criteria for each class, we decided that a machine-learning-based approach could be a valid choice. We chose neural networks as these became very popular when it comes to solving complex problems, and in many tasks they achieve state-of-art performance. When fed with the data, the neural network model can learn from the data features until it reaches the maximum accuracy. There are many kinds of neural networks, but we narrowed the choice down to the two primary candidates – the convolutional neural network (CNN) and the multi-layered perceptron (MLP). The first candidate – the CNN – is widely used for the 2D image analysis, but it has a couple of drawbacks for our task. For instance, the cluster would have to be represented as a 2D image containing both the pixels with nonzero energy and also the ones carrying no energy. This means using CNN could be a little less efficient because CNN processes all of the given pixels (even though the zero energy pixels provide no information about the cluster). Considering the fact that we are able to calculate features and work with those instead of the whole set of pixels lead us to conclusion that we could use the feature-based MLP model.

*Training Data Generation*

Even though we had the data to train our model, the data were separated into files by particle type and angle of crossing the detector. This format can be great for viewing and browsing, but it is preferable to have one data source for training purposes. For that it is necessary to create an application that processes multiple files and generates a single training collection. When creating this application, we need to keep in mind that our dataset is likely have an uneven distribution of classes, so we need to add an option to spread the classes evenly.

# Goals of the thesis

Based on the obsevations from the Analysis chapter we formulated the main goals of the thesis:

* ***Cluster visualization*** – The commonly used MMFile format [Table 1.1] provides plenty of machine readable information about the clusters. However, for a physicist it can be difficult or even impossible to extract an intuitive overview about the cluster dataset from this format. This is why we decided to set the first goal to **provide an interactive cluster visualization tool, which would display various properties of a single cluster (like a 2D and 3D image of a cluster) while also displaying a brief overview of the whole dataset**.
* ***Cluster filtering*** - Because we expect the input data to mostly contain regular and uninteresting clusters, we might need to select nontrivial clusters. In order to do that **we must create an application for filtering based on the attributes of a cluster. This application should focuse on the performance as we expect it to process a large amount of input data.**
* ***Classification*** – Because of the lack of tools for cluster classification, **we would like to create an interface which would provide a support for the classification of clusters by using the methods of machine-learning**. In order to create such classifier tool, we set two additional subgoals:
  + ***Data preparation*** – Before we start with the classification, **we must create an application which would prepare the training data for the subsequent classification.**
  + ***Classifier training*** – Because there can be different kinds of cluster dataset in the future we would like not to only create a single tool aimed for the classification of the specific clusters. Instead, **we want to create a parametrized interface for classifier training, which would enable the users to train and customize the classifier models based on their needs and the specific dataset.**

# Cluster processor applications

## Specification

Based on the observations from the Analysis chapter, we decided to make a solution which consists of multiple applications – tools for viewing, analyzing and classifying the clusters. The reason why we split the solution into multiple applications is that this way it allows for a better extendability. When some developer comes with a better idea for classificator, he can still use the rest of our solution and adjust only the some part of the application. Another advantage of this approach is that each user can decide what functionality is required and download only the specific projects that cover the user’s requirement. All these tools are primarily made for physicists and could be divided into two categories:

* ***Interactive:*** 
  + ***ClusterViewer –*** an application for cluster visualization.The tool works interactively, which means that after user input to view or analyze a cluster, it responds immediately for the selected cluster. The application can display both 2D and 3D visualization of the cluster, and histograms of the cluster properties. Furthermore, the viewer can load a classifier classifier model of a user choice and use it on
* ***Non-interactive:***
  + ***ClusterFilterer –*** a project mostly for cluster filtering based on its features. User selects the desired interval for values of the attributes in UI and then the whole collection of clusters is processed at once.
  + ***DescriptionGenerator –*** a program which proceses multiple input files into one training dataset based on the preferred options selected in UI.
  + ***ClusterClassifier –*** a console application which takes trained classifier file and dataset of clusters in json format as an argument. The output of the application is classification of the whole dataset and splitting the data into multiple files based on the predicted class attribute.
  + ***ClusterClassifierUI –*** a user interface when it is possible to train your own classifier when provided classifier configuration parameters and the training data. Apart from training, user can also merge trained classifiers into multi-level classifiers.
  + ***ClusterTrainer –*** a console application that is capable of training a classifier after configuration parameters and training data are provided.

## Cluster Viewer

In this subchapter, we will discuss all the features of the application for cluster visualization – ClusterViewer. This application contains tools to display and analyze the properties of a cluster. The features range from a simple 2D image to a 3D image and the analysis of the particle class.

*Input*

Input for the ClusterViewer application is a single ini file in the MM file format [Table 1.1]. This is selected via the user interface using either the dialog window or by typing the path to the file. Another option is to load a file in JSON file format, which is an array of objects (clusters). This file format is an output of the DescriptionGenerator application and can also be an output of the ClassifierForClusters program. Each of the objects in the array must contain three specific keys in order to be successfully loaded. These are:

* ClFile – a relative path to the cluster cl file, starting from the directory of the application which created the file (it is recommended to have the ClusterViewer project in the same directory as the application which created the file)
* PxFile – a relative path to the cluster px file (similar to ClFile)
* ClIndex – index of the given cluster in the cl file

Once the file is successfully loaded, we can proceed to the other tools.

*2D view*

To get a better overview of the cluster dataset, we decided to create a 2D image of the cluster's energy. The primary purpose of the image is to visualize the cluster as a bitmap, assigning colors based on the energy of the pixels. The viewer contains a 2D image that is represented as a 256x256 bitmap. In the cases where the same pixel is hit multiple times in a cluster, we decided to display the hit with the highest deposited energy. Because the low-energy pixels prevail in most clusters, we chose to map energies to color space logarithmically, which seemed to distinguish the energies without the need for a wide range of colors. For a more complicated cluster, this could also distinguish between halo pixels and the “real” particle trajectory.Users can navigate through the currently loaded collection by clicking Previous and Next buttons or typing out the index and clicking the Find button. Using the mouse wheel is also possible to zoom in the cluster based on the position of the cursor.

Figure 3.1 Cluster visualization indicating the halo pixels (yellow)

*Collection Histogram*

Another feature of the application is computing and presenting the Collection Histogram. That is the histogram of the currently loaded collection of clusters, representing the distribution of clusters with respect to PixelCount. The default option will display the histogram based on the cluster’s pixel count, but the program is easily extendable to accept any function of the cluster that returns a numeric value.

Figure 3.2 Histogram of the clusters with respect to their pixel count

*Pixel Histogram*

Pixel Histogram works similarly to the Collection Histogram, except for the fact that it depicts the histogram of the pixels in the currently loaded cluster. The default displayed property of the pixel is its energy. This histogram could be helpful when deciding the class of a given cluster, because similar classes tend to have a similar energy distribution.



Figure 3.3 Histogram of pixels in a single cluster based on the energy of each pixel

*3D Visualization*

Based on the ToA feature of the cluster, we calculated the coordinate according to the formula for the dependency of the ToA from the coordinate of the pixel. This way, we transformed two-dimensional points into three-dimensional, which we then showed visually as a 3D scatter plot. For a better viewing experience, we added an option to rotate the image around the x and y axes.



Figure 3.4 2D visualization of the cluster with linear trajectory

Figure 3.5 3D visualization of the cluster with linear trajectory



Figure 3.6 3D visualization of the cluster with multiple branches

Figure 3.7 2D visualization of the cluster with multiple branches

*Skeletonizer*

The application can also apply skeletonization on a given cluster. The differences in the output of the original skeletonization algorithm and our modification can be seen in the figures [Figure 3.8][Figure 3.9][Figure 3.10].



Figure 3.8 Skeletonized cluster (modified Zhan Shuen's algorithm – [])

Figure 3.9 Skeletonized cluster (original Zhan Shuen's algorithm)

Figure 3.10 Orginal cluster

*Branch Finder*

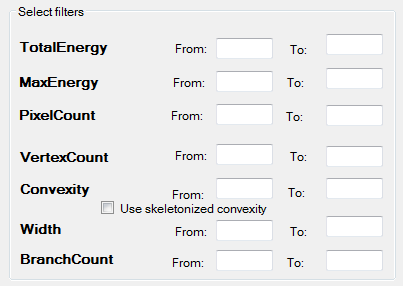
One of the more sophisticated features of the ClusterViewer application is the BranchAnalyzer. Its main task is to analyze the given cluster and search for the possible trajectories of various particles in the cluster. These trajectories - branches are then distinguished by their colors. A branch is considered to be the main branch if it starts in the center of a cluster. Each branch can also have its subbranches, which are denoted by the lighter version of their parent‘s branch color. For branch analysis, we use the algorithm based on the breadth-first search on the cluster‘s skeleton [Algorithm 1.2]. For the example of the branch analysis on a nontrivial cluster, see figures [Figure 3.11] and [Figure 3.12].

**

Figure 3.11 Cluster after branch analysis, each branch with its distinct color (pink points are the cross points where the branches split)

Figure 3.12 Original cluster before branch analysis

## Cluster Filter

After we created ClusterViewer application, we noticed that a vast majority of the clusters only has a few pixels. In order to find some non-trivial clusters we created ClusterFilter application. It consists of a simple user interface, where a user selects the ini file and the upper and lower bound for the attributes of a cluster, see [Table 3.1]. After that only the clusters which fit into the interval (lower bound, upperbound) are selected and written into a new cl file. During the process, px file is not copied. Instead, the former px file is used so this newly created cl file only provides “a new view” into the original px file.

In order to speed up the computation, we implemented the lazy evaluation, which means that after some filtering condition for the cluster fails, the rest is not evaluated. Attributes are calculated in order from the easiest to calculate to the most expensive. This implies that setting some easy-to-compute attribute to a narrow interval could greatly decrease the computation time, so we recommend to do so any time it is possible.

Table 3.1 User interface of the filterer - setting the lower and upper bound for filters

## Description Generator

The DescriptionGenerator is the application that was made to prepare the data (the features of a cluster) for the consequential training of the machine-learning-based classifier model. The application requires the data to be in data in the MM (clustered) format [Table 1.1]. We also made the algorithm extendable to more features without requiring a lot of programming.

*Input selection*

 First, a user selects and loads one or more ini files via the dialog window. All of the currently loaded files are displayed in the UI as shown in [Figure 3.13].

Figure 3.13 Loaded ini files into the DescriptionGenerator

All loaded ini files this way we call the class partitions or simply partitions. A partition can be assigned a class name, which is an editable field in the UI. These partitions are later grouped based on the class name to form the classes. The class names are compared as strings, so one needs to keep in mind the names are case sensitive. Apart from choosing input files, a user must also select the name for the output file.

*Feature selection*

After the files are selected, a user can choose which attributes will be calculated from the list. The algorithm is designed only to calculate the features that are selected. Even though more features might provide more information about the cluster, adding features could also increase the algorithm's time complexity and, therefore, the duration of the whole calculation.

We calculate features for each cluster on the input and store it into a single file in the JSON file format. For bigger clusters, the feature calculation complexities can be split into four categories according to the computational complexity:

1. Expensive

* BranchCount
* Branches

1. Less expensive

Figure 3.14 Check boxes with cluster features in the Cluster Filter application

* Width
* Convexity
* RelativeHaloSize

1. Inexpensive

* TotalEnergy, MaxEnergy, AverageEnergy
* VertextCount, CrosspointCount, RelLowEnergyPixels
* StdOfArrival, StdOfEnergy

1. Cheap

* Class, PxFile, ClFile
* PixelCount, ClIndex

Note - The complexity factor should be considered only for a large amount of the input data.

*Distribution selection*

Before processing, we need to select the distribution of the input data in the output. For that, we set the last parameters:

* ***Class distribution*** (even / proportional): When choosing even, each of the classes has a probability of to be the next class to process. If the option proportional is chosen, each class has a probability of being the next to process proportional to its size: where is the size of its cl file.
* ***Partition processing order*** (serial / parallel): Let represent the ordered partitions of a single class (based on the input order). After selecting serial, each class can proceed to the next partition only after the previous partition in the given class was processed (the whole partition, then the whole partition and so on). Suppose the user chooses parallel option, and 𝑖-th partition was processed in the previous iteration. In that case, then the next partition to process is the -th partition (when then reset to 0 to once again loop through the partitions).

Note: this order is only the order of partitions **inside** the class, keep in mind that each class is distributed by the previous parameter - Class distribution.

* ***Ending condition*** (first class / last class / first partition): This condition sets when the processing should terminate. By choosing first class option, the algorithm ends when any of the classes are fully processed. When last class is selected, the program waits for all classes to be processed before terminating. If the user chooses the first partition option, the program finishes as soon as any of the class partitions is depleted.
* ***Align class:*** When we set the optional parameter - align class - to the name of some of the classes in the loaded files, then the termination of the algorithm is only based on partitions of this align class and on the ending condition. All the other classes are meanwhile processed but with a single adjustment: When a partition of the non-align class is fully processed, it is not removed from the list, but it is only reset and will be processed once again. This means only the align class is being depleted while the others are repeatedly looping over their data.

Even though the first three parameters might actually seem useful at first glance, one might wonder about the usefulness of the align class parameter. When we want to achieve even class distribution, we have two possible options:

* Downsampling[]
* Upsampling[]

One can notice that when choosing even class distribution (with an ending condition on first-class) and no align class, this is an example of downsampling. In contrast, if the align class is set, that can be considered an implementation of the upsampling. After all these necessary steps are completed, the user can click the Process button, which starts the processing and writing the output in a JSON text file, see [Figure 3.15].

Figure 3.15 An example of calculated features for a single cluster

## Classifier aplications

When we have the training data prepared by the Description Generator application, we can start focusing on developing the tools for cluster classification. For this purpose we designed three applications:

* ClassifierUI
* ClassifierTrainer
* ClassifierForClusters

Each of these applications uses a common type of neural network classifier model – a Multi-layered perceptron (MLP). [] This type of model has many learning parameters that can affect how quickly the model learns and how robust the model is.

*NN attributes and learning parameters*

|  |  |
| --- | --- |
| *name* | [any string] - used as a **unique** classifier identifier |
| *validAttributes* | [array of ClusterAttibute object] - sets the attributes of a cluster used during the learning process |
| *layerSizes* | [array of positive integers] - represents the sizes of the hidden layers |
| *activationFunction* | [“relu”, “sigmoid”] - sets one of the two currently supported activation functions |
| *learningAlgorithm* | [“leven-marq”, “backProp”] - chooses either the Backpropagation or Levenberg-Marqadt learning algorithm |
| *epochSize* | [positive integer] – the number of samples used in a single training epoch |
| *learningRate\** | [real number between 0 and 1] – the base step size in the gradient descent learning algorithm [] (lower values = slow learning, higher values = possibly no convergence) |
| *momentum\** | [real number between 0 and 1] - a learning parameter which displays how quickly the step size inceases after successful iteration of gradient descent [] |
| *evaluationMultiClasses\** | [array of pairs ([string], [array of strings])] – maps the prediction classes in the array to a new class name for the evaluation process (for example, if our classifier predicts classes , and , we might want to analyze how well the classifier did at splitting class from the rest (using the accuracy)), so we map [(,[]),(, []) |
| *usedTrainDataSize\** | [a real number between 0 and 1] – the proportion of data used for training (remaining data are used for testing) |
| *printInterval\** | [positive integer] – the frequency of printing the training error of the neural network to the console |

Table 3.2 attributes and learning parameters of the classifier model stored in a config file (json),

\* = field is not obligatory

*Classifier UI and Classifier Trainer*

The ClassifierUI is an application that aims to provide two main functionalities – classifier training and classifier merging. ClassifierTrainer is a console application that also provides classifier training (precisely the same as the ClassifierUI) but offers the option to run the learning process from the script without manual interaction. Because the training process is the same in the console application as it is in the UI, we will not discuss it separately.

***Classifier Training*** –The application trains a new classifier model based on the training data and the learning parameters data provided by the user. Some parameters like the minimal accuracy can be set directly via the UI, but most of the learning parameters are set in a JSON config file, which must be loaded into the UI before learning starts. Optionally we can also load an already trained classifier model, and if done so, this model will be trained instead of creating a new MLP. By clicking Train Classifier we start the learning process. Then, the data is randomly split into training and test set according to the learning parameter (by default 90% training, 10% test). Information about the error rate on the currently processed epoch is displayed in the console window. When the training is completed, the model is tested on the test data. To present the quality of the classifier, we used the confusion matrix[] – a matrix where its element is the number of clusters that belong to the -th class and were labeled by the classifier as the -th class. So ideally, we want to have the most cluster examples on the diagonal of the matrix. The result of the training process is what we call a simple classifier, and it is stored in two files with the suffixes .csf and .csf\_support. These two files should be in the same directory and preserve the same name at all times (they only differ in suffixes).

***Classifier merging*** – When we have multiple simple classifiers, we might need to concatenate their results in a single classifier with a tree-like structure. This can be especially useful, because as displayed in [Figure 3.16] when we have classes with little training data (Classes A and B), we can create a root classifier trained on an evenly distributed dataset by downsampling the larger classes (Class C and D). However, downsampling means not using some part of the training data, which could negatively affect the model's accuracy. Because it is possible to have more than one layer, we can train another classifier on the larger dataset (of Classes C and D), which enables us to:

* Use most of the training data for learning
* Preserve the even distribution of classes among each level of the tree, which is preferred by many machine-learning based classifiers

Figure 3.16 Example of a multilayered classifier with two layers (Root classifier and Classifier 1)

Via the UI, it is possible to create a multi-layered classifier with a maximum of 4 levels, and on each level, there can be a single classifier at most. This is because it can be quite complex to build any tree via the UI, and also we expect this version to be sufficient for most cases. With a little programming, the user can build any classifier tree by simply connecting the nodes.

To merge classifiers into a multi-layered classifier user imports the trained model, and if it is not the final layer, the user also specifies the split class, which is the name of the class that the classifier on the next layer will further process. This means if we input simple classifiers for merging, we should also input corresponding split classes. Then the name of the output is selected, and when the merging is completed, two output files are created - .csf and .csf\_support files.

*Classifier For Clusters*

The ClassifierForClusters is a console application. It can classify the given collection and provide an output based on its parameters when it is provided a trained classifier model and the cluster data in json format. There are three possible types of the output:

* ***Frequencies:*** The application prints the key-value pairs of the name of the class and its frequency in the data
* ***Classes:*** The program outputs a single JSON file for each of the classes, which contains the clusters that were predicted to belong to the particular category.
* ***Specials:*** When selected this option, the program's output is similar to the Classes option but, moreover, outputs a file where all the unclassified clusters that match specific criteria (potentially rare clusters) are stored. Changing these criteria is simple but requires a slight modification of the source code.

# Development documentation

In this chapter, we will discuss the technical part of the solution ClusterProcessor. We will have a brief look into the implementation and the main objects and methods used in the solution. This section is primarily aimed for the developers who would like to find out how our solution works and those who plan to extend this solution in any way.

*Technology*

The framework we decided to use for the solution is .NET Framework and Windows Forms. The reason for that is simply because my primary programming language is C#, and I consider Windows Forms to be a reasonable choice for creating Graphical User Interface. However, WinForms does not support all features we required, so we need the following external libraries:

* Chart Director [11] – a 3D graphical plotting library
* Newtonsoft.Net[] – a library (NuGet package) used for serialization and deserialization from the JSON file format
* Accord.Net[] – a machine-learning framework (NuGet package) we use for the classification of the clusters

*Solution structure*

The whole solution represents a set of tools for cluster processing. The solution is structured into multiple separate projects mostly for two reasons:

1. Users can choose which parts of the solution are desired and use only those.
2. This way the whole solution can be easily extended by modifying only one project a reusing the rest of the solution.

Figure 4.1 Dependency structure of the ClusterProcessor solution

Cluster Viewer

Cluster Calculator

Classifier

Trainer

ClassifierForClusters

Descr Generator

Cluster Filter

ClassifierUI

Cluster Experiment

The projects in the solution can be divided into three layers according to their dependencies, as shown in [Figure 4.1]

## Main objects

C# as a programming language is well designed for object-oriented programming, and that is also the approach we chose to use. Each class represents an entity, and these entities communicate with each other. This way, we tried to design our program. In this section, we would like to introduce a few main objects shared across the whole solution.

**PixelPoint** – an object for a single pixel, which stores its and coordinates, energy, and the time of arrival

**Cluster** – a class which stores all of the necessary data for a single cluster, including an array of its pixels

**ClusterAttribute** – an enum attribute that can be computed and used as training data for the classifier

**DefaultAttirbuteCalculator** – an object which can calculate the values of given cluster attributes when provided the cluster object

## Cluster Viewer

Even though the whole cluster viewer project consists mostly of the event handlers for the application's buttons, it uses some features from the ClusterCalculator, which are worth noting from the software engineering perspective.

*Browsing Cluster Collection*

Browsing of the collection of clusters is done via the buttons. Without the chance of loading the data into operating memory, we need to search for a specific cluster sequentially each time we click previous. There is an option of creating a mapping table (index of a cluster, byte offset of a cluster) and then use.Seek() method (which might be faster but could also consume a significant amount of memory). Another approach would be to use caching of the recently viewed clusters. We tested the browsing on the files with large cl files and did not notice any significant issue (browsing is done only on the cl file, which is usually significantly smaller than the px file). Because of that, we decided not to try optimizing browsing any further. Browsing is implemented by ClusterInfoCollection :IEnumerable.

*Skeletonization*

Skeletonization is the process of finding a skeleton of a binary image. For this purpose, we proposed Algorithm 1.1. As the thinning process returns nontrivial data for bigger clusters, we will optimize the algorithm for these (> 100 pixels). The key is to use a well-fitting data structure. Throughout the algorithm, we perform two operations most often. The first one is.Contains(), which means 'does the specific pixel have a nonzero energy ?' and the second one is.Delete() with is setting the energy of a specific pixel to 0. An ideal option for these operations seems to be an array of 256x256 pixels, which can effectively do both these operations in constant time. The problem is that this approach would lead to huge performance issues as most of the clusters are smaller than 100 pixels, and the overhead of the array with 65 536 items would take much time. Using the list collection List<T> from the standard library, we get rid of this problem. On the other hand.Contains() and .Delete() take linear time to execute. That is why we chose to use HashSet<T>, which has similar memory usage as List<T> but both.Contains() and .Delete() are done in a constant time. In case we would need another performance boost, the whole algorithm can be almost trivially parallelized.

## Filter and Description Generator

Both ClusterFilter and DescriptionGenerator calculate the features for many clusters, so we decided to analyze them together. The applications run on a specific collection for a potentially long time, so performance seems to be a good factor we should focus on and optimize.

*Complexity*

The iteration process reuses ClusterInfoCollection, and as we need to process every item in the collection, there is not much to be optimized. Let  denote the size of the px file and the size of the cl file. The iteration process has a linear time complexity with respect to . The calculation of an attribute value depends on the particular attribute. Asymptotic time complexities with respect to the number of pixels of a cluster is shown in Table 4.1.

|  |  |
| --- | --- |
| ATTRIBUTE | ASYMPTOTIC TIME COMPLEXITY |
| PixelCount, ClIndex, PxFile, ClFile, Class | : All of these attributes are present when loading the cluster – no computational overhead |
| TotalEnergy, MaxEnergy, AverageEnergy, RelLowEnergyPixels, StdOfEnergy, StdOfArrival | : Calculation of these properties only requires an iteration over each pixel in the cluster. Therefore we obtain the linear complexity. Because this is also the lower bound, we can state that the complexity |
| Width, RelativeHaloSize, Convexity, VertexCount, CrossPointCount | : Because all of these attributes require skeletonization of the cluster, we need to analyze Algorithm 1.1. Each iteration consists of pixels and there can be at most iterations, where is the thickness of the cluster. By analyzing the worst-case scenario (cluster has a circular shape), we can notice that . When we combine these observations we obtain the resulting upper bound. (Note: Convexity and Width attribute also require convex hull calculation, which is done in , which doesn’t affect the asymptotic complexity) |
| BranchCount, Branches | originally, after update: When we look at Algorithm 1.2, we can notice the following:   * The time complexity of the breadth-first search[] is linear * The branch count could be at most linear with respect to the number of pixels.   Thus, we can estimate the worst-case complexity by . In contrast, we usually do not need to analyze the cluster much further if we already have hundreds of its longest branches. Because of that, we set the upper bound for branch count to be 40. We also set the depth of the calculation, so the algorithm ignores the branches that are nested more than four times. Even though this restriction doesn’t provide us a complete branch analysis, in most cases, it is sufficient to provide a significant performance boost to. (Note: From the real (non-asymptotic) observed calculation time, it is still the most expensive calculation.) |

Table 4.1 Time complexity of the feature calculation

To sum up, the total complexity of the filtering process is when taking the updated branch analysis into account.

*Adding new features*

To add a new feature to DescriptionGenerator, a user needs to follow these steps:

1. Add the name of the attribute to the enum ClusterAttribute to the ClusterCalculator project.
2. Add a case to the switch section of the CalcAttributes method of the DefaultAttributeCalculator with your attribute and its calculation and store the value to the dictionary attributePairs to an index attribute. (attributePairs[attribute] = calculated\_value)
3. In the DescriptionGenerator project, update the UI – add a checkbox to the other attributes with the same name as the enum you added in step 1 (because the string-enum conversions are done automatically).

## Classifier aplications

In this chapter, we discuss the process of cluster classification from a developer point of view. We introduce some fundamental objects used in the classification process. Then we also mention a simple guild how to build your own classifier.

*Classifier objects*

**Interface IClassifier** –an interface that provides required methods for a classifier model (.Classify())

**Interface ILearnableClassifier : IClassifier –** an interface that extends the IClassifier by adding support for training (.Train() method)

**Class NNClassifier : ILearnableClassifier** – a basic classifier based on a multi-layered perceptron NN

**Class MultiLevelClassifier : IClassifier –** a classifier created by merging more NNClassifiers

**Class ClassPrediction –** an object that represents the prediction calculated by the classifier, apart from the name of the predicted class, also contains the probabilities of how confident the classifier is about each class (implemented via the softmax[] function)

*Building your tree classifier*

When you want to build a MultiLevelClassifier you first need to create a NNClassifier, which can be done two ways:

1. From the ClassifierUI
2. Using a console application ClassifierTrainer

After we have all of the NNClassifiers prepared, we can start with merging. There is a merging option in the ClassifierUI, which only allows a single classifier on each level. In case you want to build any classifier tree, it can be done by following these steps (an example of how we merged our classifier can be seen in the .FromDefault() in the class MultiLevelClassifier):

1. Create new instances of NNClassifier and call.LoadFromFile() and create a new instance of MultiLevelClassifier
2. Wrap the simple classifiers into the class ClassifierNode and the root classifier into the ClassifierRoot
3. Connect the nodes starting from the bottom of the tree by calling parentNode.Descendants.Add(splitClass, sonNode)
4. After all of the previous steps are done, set the root by calling the method of the MultiLevelClassifier - SetRoot(classifierRoot)

*Serialization*

Each classifier that is created can also be stored in a file for future use. This process is called serialization and is done via the Newtonsoft library[]. When we were implementing the serialization of both NNClassifier and MultiLevelClassifier, we needed to set all of the property modifiers to the public so they can be visible for the serializer. Another problem is that the network itself (from the Accord.Net[] library), cannot be serialized using Newtonsoft[] library because of compatibility issues. That is the main reason why as a result, each classifier is stored in two files (.csf and .csf\_support). The .csf file contains a serialized Network by the serialization tools of the Accord.Net[] library. In contrast, the .csf\_support file contains the whole NNClassifier object except for the network itself (serialized by the Newtonsoft library[]).

*Selecting uncommon clusters*

One of the ultimate goals of the classification could be finding the uncommon clusters*.* Implementation of the selection of such clusters is implemented directly in the classification. ClassPrediction contains a method CalcConfidence(), which calculates the certainty of the prediction based on the output neuron values from the network. This method serves to detect unclassified clusters. During the classification, a method CheckSpecialClusters() is called, which takes the unclassified cluster and decides if it can be considered an extraordinary cluster. Currently, the implementation of this method only selects the clusters with nontrivial branch count and pixel count. Both CalcConfidence() and CheckSpecialClusters() are marked as virtual so that they could be potentially reimplemented simply by inheritance. This is also where I see the biggest chance for improvement and extension of the program.

# Experiment

During the work on the thesis, we created a few classifier models which should be tested on real datasets. For that, we designed a couple of scenarios where we compare our approach with the alternatives and analyze the results.

In all of our experiments we will use the base sample dataset of clusters as displayed in [Table 1.4]. To perform all of the experiments we prepared the following single-layered types of models:

1. - divides particles to lead particles and the rest
2. - separates fragments, helium and iron from each other and the rest
3. - separates low energy electrons from protons and from the rest
4. - splits particles into 4 categories – electrons, muons, pions and an artificial category elPi0, which represent electrons and pions which traversed the detector at the zero degrees angle (these we consider to be very difficult to split, because both categories only contain very few pixels).
5. - splits clusters into all categories directly (including the artificial category elPi0)

Using the single-layered classifiers and given training data distribution (the clusters with similar frequency in the training dataset are on the same level) we also constructed the multi-layered classifier as displayed in the hierarchy [Figure 5.1].

Figure 5.1 Multi-level classifier structure

In order to measure the quality of a classifier we define the accuracy of a classifier on the dataset as . []

## Classifier Parametrization

Each single-layered classifier that is trained via our solution has multiple parameters as shown in [Table 3.2]. In this experiment we will try to modify the values of some of these parameters and compare the results. As a base set of parameters we consider the file fragCalibration.json, so we will modify the classifier . Because there are many parameters to possibly modify and training of multiple classifiers on a large datasets is time consuming, we chose only a few parameters to modify:

* Learning rate
* Momentum
* Number of layers of the neural network

For each of these properties of classifier we decided to try 6 scenarios (every scenario was tested on 10 classifiers) :

1. ***Lower learning rate and momentum*** 
   * learning rate: 0.1
   * momentum: 0.1

Result - average normalized accuracy of the model: 0.86.

Interpretation: First we need to note that the dataset we use for training of this network is  large (almost 500 000 clusters). This means that despite low values of learning rate and momentum (slow learning), the classifier had enough time to converge to a reasonable accuracy.

1. ***Original learning rate and momentum***
   * learning rate: 0.5
   * momentum: 0.5

Result - an average normalized accuracy: 0.90

Interpretation: These values of parameters showed the best convergence speed and accuracy. This is the reason why we use these values as the defaults in our experiments.

1. ***Higher learning rate and momentum***
   * learning rate: 1
   * momentum: 1

Result – an average normalized accuracy: 0

Interpretation: When we set the values of learning parameters too high, we can notice that a divergence occures, which causes the classifier provide unusable results.

1. ***Smaller number of hidden layers***
   * A single hidden layer with only one neuron

Result – an average normalized accuracy: 0.67

Interpretation: Using a smaller network usually implies faster learning, but small networks do not perform very well on more complex problems.

1. ***Former number of hidden layers***
   * Two hidden layers with 13 neurons each

Result – an average normalized accuracy: 0.90

Interpretation: This configuration seems to provide a classifier with the best accuracy for our task. We can observe that the network is large enough, to capture the various relationships of the features and small enough to learn reasonably fast.

1. ***Higher number of hidden layers***
   * Three hidden layers with 13 neurons each

Result – an average normalized accuracy: 0.25

Interpretation: We can notice that the accuracy is really low. This could be because the network is too deep for our task and our chosen activation function (sigmoid function[]) and therefore it learns very slowly.

## K-Fold cross-validation of single-layered models

Because the training of the classifier is, in general, a non-deterministic procedure, we should also test the training method and how often the classifier training is successful. Another property we might be interested in is the variance of these results, which is precisely what we test in the experiment.

One of the most commonly used methods for training evaluation is the -fold cross-validation[]. There, the data is randomly split into partitions where partitions are used for training and a single part is for testing. We performed a 6-fold cross-validation of the dataset and obtained results (accuracies) as shown in the table below [Table 5.1].

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Val\_run\classifier |  |  |  |  |
| 1. validation | 0.987 | 0.921 | 0.951 | 0.705 |
| 2. validation | 0.979 | 0.922 | 0.950 | 0.747 |
| 3. validation | 0.981 | 0.912 | 0.957 | 0.733 |
| 4. validation | 0.990 | 0.938 | 0.938 | 0.740 |
| 5. validation | 0.977 | 0.938 | 0.943 | 0.741 |
| 6. validation | 0.986 | 0.900 | 0.959 | 0.740 |
| Mean | 0.983 | 0.922 | 0.950 | 0.734 |
| Variance | 0.000026 | 0.00452 | 0.000065 | 0.000226 |

Table 5.1 Results of the cross validation of a single layered model

The results of the 6-fold cross-validation indicate, that our classifiers appear to provide a stable performance (relatively low variance). We can also notice, that the categories electron/muon/pion have lower accuracies which could mean that these categories are very difficult, if not impossible to classify correctly with high accuracy.

## Multi-layered classifier

The concept of a multi-layered classifier is something we use in this thesis frequently. But how does it fare in comparison with a single-layered classifier on the real data?

In order to compare these two classifiers we trained 10 single-layered models and 10 multi-layered classifiers on the similar amount of data and compared the results on the test dataset.

The best model accuracy was 0.61 (mean 0.52) , while the best accuracy of the model reached 0.82 (mean 0.80). We also estimated the variance of to be 0.0038 while the variance of was only 0.0001. Confusion matrices of these models can be seen in the tables [Table 5.2] and [Table 5.3]. In the table it is visible that the particle distribution in the test set is very uneven. However, this is not a problem, because we can normalize each value by dividing it by the sum of its row. After rerunning the experiment and computing the normalized confusion matrix we obtain the best relative accuracy of 0.85 for model and 0.59 for model.

Based on the results of an experiment we conclude that the accuracy of the multi-layered classifier was significantly higher than the single-layered model with both classifiers using similar architectures of the neural networks. One reason why there is such a difference could be the fact that multi-layered models do not suffer from downsampling (and upsampling) as much as single-layered models. Another reason could be the scaling – each single-layered classifier scales its inputs (linear scaling []) into the range (0,1) which means having a type of a cluster with very high energy causes the the scaling to scale most of the other cluster energies very close to 0, which makes them difficult to separate. In contrast, multi-layered classifier we bulit first processes the bigger cluster types (first layer) and then proceeds to smaller clusters which causes the scaled values to be more evenly spread in the (0,1) interval. An important note – by no means we state that multi-level architecture of the classifier is always better than single-layered architecture. It might depend on the type of the classifier, but in our case the differences were significant.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| True\Pred | lead | frag | he | fe | prot | low\_e | elPi0 | pion | muon | electr | uncl |
| lead | 283 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| frag | 17 | 362 | 31 | 200 | 64 | 0 | 0 | 0 | 0 | 0 | 10 |
| he | 1 | 1537 | 2321 | 4392 | 830 | 3 | 17 | 0 | 0 | 1 | 206 |
| fe | 3 | 2 | 0 | 1769 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| prot | 0 | 550 | 126 | 94 | 31011 | 152 | 2860 | 67 | 408 | 414 | 315 |
| low\_e | 0 | 0 | 0 | 0 | 83 | 1525 | 1291 | 28 | 29 | 37 | 7 |
| elPi0 | 0 | 20 | 0 | 0 | 721 | 43 | 4640 | 78 | 130 | 278 | 221 |
| pion | 3 | 10 | 0 | 0 | 326 | 28 | 1233 | 4106 | 238 | 2835 | 90 |
| muon | 11 | 102 | 11 | 0 | 678 | 11 | 624 | 30 | 1303 | 126 | 104 |
| electr | 46 | 390 | 0 | 0 | 2806 | 80 | 3301 | 1235 | 2527 | 4167 | 448 |
| uncl | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 5.2 Confusion matrix of the model with the best accuracy in the experiment (uncl represents unclassified clusters)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| True\Pred | lead | frag | he | fe | prot | low\_e | elPi0 | pion | muon | electr | uncl |
| lead | 287 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| frag | 22 | 609 | 44 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 8 |
| he | 1 | 1295 | 7899 | 1 | 37 | 8 | 0 | 2 | 0 | 0 | 65 |
| fe | 2 | 9 | 0 | 1749 | 0 | 0 | 0 | 0 | 0 | 0 | 14 |
| prot | 1 | 173 | 1528 | 0 | 33111 | 236 | 139 | 102 | 426 | 207 | 74 |
| low\_e | 0 | 0 | 0 | 0 | 36 | 2771 | 14 | 16 | 40 | 16 | 107 |
| elPi0 | 0 | 0 | 0 | 0 | 176 | 51 | 7886 | 36 | 406 | 402 | 43 |
| pion | 0 | 0 | 3 | 0 | 55 | 186 | 37 | 5051 | 97 | 41 | 30 |
| muon | 0 | 0 | 0 | 0 | 71 | 38 | 29 | 112 | 2469 | 264 | 17 |
| electr | 0 | 0 | 0 | 0 | 681 | 180 | 3002 | 406 | 3543 | 7050 | 138 |
| uncl | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 5.3 Confusion matrix of the model with the best accuracy in the experiment (uncl represents unclassified clusters)

*Best classifier*

Apart from the classifiers we trained for the experiments, we also repeated the process multiple times, iterating over the training set and this way got a model with better results than the models trained in the experiments. When we run this classifier on the test dataset and normalize the confusion matrix, we reached the accuracy of 0.875 (87.5%). The normalized confusion matrix for this classifier is displayed in the table[]. We also decided to save this model for future use - it is stored in the file bestClassifier.csf which is a part of the presented solution.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| True\Pred | lead | frag | he | fe | prot | low\_e | elPi0 | pion | muon | electr | uncl |
| lead | 0.921 | 0.034 | 0 | 0.044 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| frag | 0.004 | 0.888 | 0.102 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.004 |
| he | 0 | 0.054 | 0.94 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.002 |
| fe | 0 | 0.006 | 0 | 0.976 | 0 | 0 | 0 | 0 | 0 | 0 | 0.016 |
| prot | 0 | 0.004 | 0.022 | 0 | 0.933 | 0.006 | 0.006 | 0.01 | 0.004 | 0.009 | 0.001 |
| low\_e | 0 | 0 | 0 | 0 | 0.005 | 0.957 | 0.012 | 0.011 | 0.004 | 0.008 | 0 |
| elPi0 | 0 | 0 | 0.001 | 0 | 0.032 | 0.032 | 0.897 | 0.007 | 0.029 | 0.015 | 0.003 |
| pion | 0 | 0 | 0 | 0 | 0.007 | 0.006 | 0.002 | 0.821 | 0.044 | 0.109 | 0.007 |
| muon | 0 | 0 | 0 | 0 | 0.011 | 0.011 | 0.024 | 0.006 | 0.841 | 0.102 | 0.003 |
| electr | 0.003 | 0 | 0 | 0 | 0.016 | 0.011 | 0.026 | 0.163 | 0.198 | 0.571 | 0.008 |
| uncl | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 5.4 Normalized confusion matrix of the best trained model

# Epilogue / Conclusion

The main focus of the thesis was to implement a set of tools for processing the output clusters from the TimePix3 chip in the MM data format[]. We had to create an interactive tool for cluster visualization and non-interactive tools for cluster filtering, training data generation and cluster classification with an option for training new classifiers according to the needs of the user. We consider all of these goals to be completed.

**Attribute calculation**

As one of the essential tasks in our project was analyzing the clusters, we suggested some attributes of a cluster attributes of a cluster. These attributes were designed to provide more compact represnation of clusters and allow for a simpler distinction between various types of clusters. Calculation of some of these features rely on an image skeleton where we suggested a slight modification of the Zhan-Shuen’s algorithm [Algorithm 1.1]. Arguably the most interesting part of the feature calculation was the branch analysis were we introduced a new algorithm based on the breadth-first-search [Algorithm 1.2].

**Visualization**

We created a project for cluster visualization which provides 2D and 3D image representation of a cluster. Apart from that, the application also contains histogram of the pixels in currently viewed cluster as well as the histogram of the whole cluster collection. By using feature calculation we are also able to show cluster’s attributes to the user. It is also possible to load a trained classifier into the viewer and use the classifier directly inside the viewer for prediction of the cluster‘s class.

**Classification**

One of the most challenging tasks in the whole thesis was cluster classification. Firstly, we created a tool for feature based training data generation in JSON[] format. A new type of multi-layered classifier architecture was proposed, created by connecting the results of multiple classifiers into a tree-like structure. Then, we implemented an application which is capable of training a simple neural network classifier based on the parameters selected by the user that could reflect the expected distribution of the cluster classes in a dataset. These classifiers can then be merged into a multi-layered classifier. To present the functionality of the training application we created a model for classification of the real cluster dataset from the ATLAS[] detector.

**Future research**

Because each tool is implemented by a separate application, it should not be difficult to extend our solution in many ways. As one of the most promising extensions we see experimenting with different cluster features and examining their internal dependencies to provide the best possible results. Another important extension of our application could be reimplementation of the function for selecting rare unclassified clusters based on the new features like the number of branches with length (or energy) above a given threshold.

[Sample: Bibliography is prepared in accordance with applicable standards. A recommended citation standard for the master thesis is ISO 690. Names of journals can be abbreviated, but only in a codified form. All used sources must be properly cited.]

**Bibliography**

1. **Meduna, Lukáš.** Detecting elementary particles with Timepix3 detector. *Digitálny repozitář Univerzity Karlovy.* [Online] 2009. https://dspace.cuni.cz/bitstream/handle/20.500.11956/106938/120331567.pdf?sequence=1&isAllowed=y.

2. CERN - Timepix. [Online] [Cited: 4 1, 2021.] https://kt.cern/technologies/timepix3.

1. TimePix3 detector: <https://kt.cern/technologies/timepix3>
2. 3D reconstruction: <https://link.springer.com/article/10.1140/epjc/s10052-017-4993-4>
3. Clusterer and Mr.Meduna’s Work: <http://medunalukas.cz/utef/>
4. Mr.Meduna’s Bachelor Thesis(MM data format): <https://dspace.cuni.cz/handle/20.500.11956/106938?locale-attribute=cs>
5. 3D plotting library: <https://www.advsofteng.com/index.html>
6. Skeletonization: <http://agcggs680.pbworks.com/f/Zhan-Suen_algorithm.pdf>
7. Skeletonization <https://core.ac.uk/download/pdf/82669939.pdf>
8. Cluster Processor project repository
9. Skeletonization <https://www.researchgate.net/publication/308822048_A_new_thinning_algorithm_for_binary_images>
10. Accord.net: <http://accord-framework.net/>
11. Chart director library: <https://www.advsofteng.com/>
12. Skeletonization image <https://www.researchgate.net/publication/220273912_K3M_A_universal_algorithm_for_image_skeletonization_and_a_review_of_thinning_techniques>
13. <https://indico.cern.ch/event/124392/contributions/1339904/attachments/74582/106976/IntroSilicon.pdf>

**List of Tables**

[Table 1.1 Input MM file format 7](#_Toc72421597)

[Table 1.2Description of functions and variables used in the Algorithm 1. 14](#_Toc72421598)

[Table 1.3 Categories of clusters based on their shape 18](#_Toc72421599)

[Table 1.4 Types of the clusters for classification with their examples 20](#_Toc72421600)

[Table 3.1 User interface of the filterer - setting the lower and upper bound for filters 27](file:///D:\source\repos\Celko2020\ClusterProcessor.docx#_Toc72421601)

[Table 3.2 attributes and learning parameters of the classifier model stored in a config file (json), 32](#_Toc72421602)

[Table 4.1 Time complexity of the feature calculation 40](#_Toc72421603)

[Table 5.1 Results of the cross validation of a single layered model 47](#_Toc72421604)

[Table 5.2 Confusion matrix of the model with the best accuracy in the experiment (uncl represents unclassified clusters) 49](#_Toc72421605)

[Table 5.3 Confusion matrix of the model with the best accuracy in the experiment (uncl represents unclassified clusters) 49](#_Toc72421606)

[Table 5.4 Normalized confusion matrix of the best trained model 50](#_Toc72421607)

**List of Abbreviations**[Sample: Attachments to the master thesis, if any (various additions such as programme extracts, diagrams, etc.). Each attachment must be referred to at least once from one's own text of the thesis. Attachments are numbered.]

**Attachments**

[Sample: Hard back cover of the master thesis **- not a part of the electronic version**.]