ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

PROJECT

on

Computer Vision and Image Processing M

Gamma Ray Astrophysics

Luca Bonfiglioli, Antonio Grasso

Anno Accademico 2018/2019

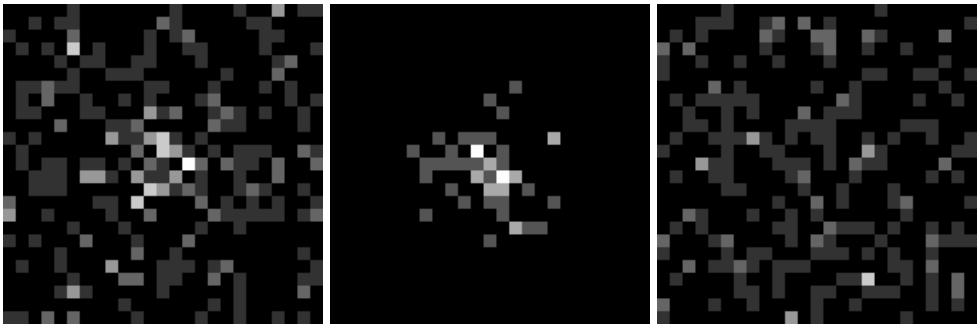
Contents

Chapter 1

Problem

The main task is to analyse astronomical images taken with gamma-ray telescopes of the **Cherenkov Telescope Array (CTA)** to find gamma-ray sources into the images. In particular, we need to develop a computer vision software using the **Python** programming language. This software should be able to determine whether an image contains a gamma-ray source or not. In the former case, it should also be able to locate the gamma-ray source position into the image.

These images are in the **Flexible Image Transport System (FITS)** format, which consists of one or more Header and Data Units (HDUs), where the first HDU is called primary HDU, or primary array. In our case, the primary array contains a 2-D array of pixels, which can be seen as the 2D-histogram of the image. The FITS headers may contain image metadata.



(a) Source and background (b) Source (c) Background noise

Figure 1.1: Simulated maps showing a 25x25 pixel region

Sky maps are corrupted by noise, which may constitute an issue when trying to detect the presence and location of a gamma-ray source. **Noise** is present in two main forms:

- Some photons emitted by a source are detected from a slightly different angle, this results in pixel displacement around a source, making it appear like a blob of high intensity pixels, scattered across a small region, as shown in figure 1.1b;
- Some photons come from directions which do not correspond to any emitting source (background noise), as shown in figure 1.1c.

The background noise is not completely uniform, but it may have some peaks and lows and if the source is weak enough it may become indistinguishable from those background noise peaks.

Chapter 2

Solution

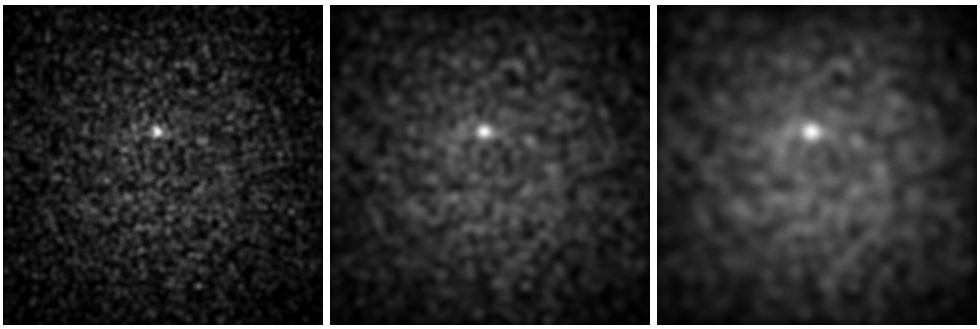
Starting from a raw image, we observe that the region of the image corresponding to the source contains a higher density of pixels with greater intensity than the background ones. However, not all the pixels belonging to this region carry high intensity values: high intensity pixels appear scattered and mixed with low intensity ones.

2.1 Gaussian filtering

The first step to accomplish is denoising the raw image: this can be done by applying a **Gaussian filter**. Before applying it, we need to set σ and the size of the kernel s. Since the interval $[-3\sigma, 3\sigma]$ captures 99% of the area of the Gaussian function, we can assign to s the closest odd integer value to 6σ. We decided to bind the kernel size to a minimum of 5.

$$s(\sigma) = \max\left(2\left\lfloor\frac{\lfloor 6\sigma \rfloor}{2}\right\rfloor + 1, 5\right) \tag{2.1}$$

We now need to choose the value of σ in order to make it easier for us to detect the source: this depends on its shape and on the background noise in its neighbourhood. Thus, given a chosen σ, a single filtering may not be enough to detect the source, because that specific value may not be appropriate for that particular image. Therefore, we choose to apply multiple filters with increasing values of σ.
Starting from an image A filtered with σ_a we want to obtain an image B filtered



(a) Image smoothed with $\sigma = 1.5$ (b) Image smoothed with $\sigma = 2.5$ (c) Image smoothed with $\sigma = 3.5$

Figure 2.1: Image smoothed with different values of σ

with σ_b: if $\sigma_b > \sigma_a$, this can be easily done by filtering A with $\sigma_c = \sqrt{\sigma_b^2 - \sigma_a^2}$. According to this formula, $\sigma_c < \sigma_b$ this means that $s(\sigma_c) < s(\sigma_b)$, so filtering with σ_c is less computationally expensive than filtering with σ_b.
Once chosen a range of n values $[\sigma_1, ..., \sigma_n]$, we can avoid filtering the original image n times with σ_i, $i = 1, ..., n$, by instead applying n consecutive filters with $\tilde{\sigma}_i$, $i = 1, ..., n$ where:

$$\tilde{\sigma}_i = \begin{cases} \sigma_1 & i = 1 \\ \sqrt{\sigma_i^2 - \sigma_{i-1}^2} & i > 1 \end{cases} \tag{2.2}$$

By increasing σ, as shown in figure 2.1, the source becomes more compact, its shape more circular and the pixel with the highest intensity tends to be around the center of this region.

2.2 Local maxima extraction

If the source is present, for each filtered image we observe that there exists a local maximum which is a good approximation of the source location: the lower the σ, the better the approximation.
Thus, we need to extract local maxima from each image: to do so, we iterate on each pixel checking whether it is greater than all its neighbouring pixels or not.

The maxima are then stored into a list as tuples $\langle\langle x, y\rangle, i\rangle$ where $\langle x, y\rangle$ are the maxima coordinates and i is their intensity.
They are then sorted by decreasing values of intensity, because we know that if the source is present the corresponding local maximum will be among the most intense ones.
By increasing σ from σ_i to σ_{i+1} at each step, we expect some maxima found at the step $i + 1$ to remain within a small range from their position at step i. We want these corresponding maxima to be considered as a single maximum throughout all the iterations.

2.3 Useful data structures

For this purpose, we now introduce two simple data structures: PointData and PointDataList.

2.3.1 Pointdata

It lets us store values related to a specific point in the image.
Attributes:

- *original*: a pair of coordinates initialised to a given tuple;
- *latest*: another pair of coordinates, initially equal to *original*;
- *r*: a numeric value initialised to a given positive value;
- *values*: a list of numeric values, initially empty.

Methods:

- *set(c, v, i)*: it takes as input a pair of coordinates *c*, a value *v* and an index *i*, setting *latest* to *c* and the i-th element of *values* to the maximum value between *c* and the previously stored value.
- *compatible(c)*: it takes as input a pair of coordinates *c* and it returns a Boolean value that is true if the Euclidean distance between *c* and *latest* is less than *r*.

Example

We initialise PointData with coordinates (100, 100) and $r = 5$:

original	**latest**	**r**	**values**
(100, 100)	(100, 100)	5	∅

We perform the set operation with the coordinates **(101, 100)**, the value **30** and the index **0**:

original	**latest**	**r**	**values**
(100, 100)	**(101, 100)**	5	**30**

We perform another set operation with the coordinates **(101, 102)**, the value **15** and the index **4**:

original	**latest**	**r**	**values**				
(100, 100)	**(101, 102)**	5	30	0	0	0	**15**

The parameters *original* and *r* are intended to be read-only.

2.3.2 PointDataList

It is a collection of PointData.

Attributes:

- *points*: a list of PointData, initially empty;
- *r*: a numeric value initialised to a given positive value.

Methods:

- *set(c, v, i)*: it takes as input the same parameters *c*, *v*, *i* as those of the set operation of PointData. This operation checks whether there is at least one element of points compatible with the coordinates *c* or not: in the former case it performs the set operation with the parameters *c*, *v*, *i* on the closest

compatible PointData; in the latter it adds a new PointData to points and it performs the set operation on it.

- *get_point(c)*: it takes as input a pair of coordinates *c* returning the closest compatible element of points if any is present, nothing otherwise.

Example

Here we have a PointDataList already initialised with 2 PointData and some values:

original	**latest**	**r**	**values**				
(100, 100)	(101, 102)	5	30	0	0	0	15
(120, 100)	(121, 99)	5	0	2	15	20	0

We perform the set operation with the coordinates **(50, 80)**, the value **18** and the index **3**. Since there is not any PointData compatible with (50, 80), a new one will be created and added to the structure:

original	**latest**	**r**	**values**				
(100, 100)	(101, 102)	5	30	0	0	0	15
(120, 100)	(121, 99)	5	0	2	15	20	0
(50, 80)	**(50, 80)**	**5**	**0**	**0**	**0**	**18**	**0**

We perform the set operation with the coordinates **(100, 101)**, the value **30** and the index **5**. (100, 101) is compatible with (101, 102), so it will perform the set operation on the first PointData:

original	**latest**	**r**	**values**					
(100, 100)	**(100, 101)**	5	30	0	0	0	15	**30**
(120, 100)	(121, 99)	5	0	2	15	20	0	0
(50, 80)	(50, 80)	5	0	0	0	18	0	0

2.4 Features

For each filtered image we want now to compute a set of features for some of the maxima and store the computed values inside different PointDataList structures, one for each feature.
Every feature is associated with the method *update_feature(maxima, i, point_data_list)*, where *feature* corresponds to the feature name: it takes as input the sorted list of the maxima *maxima*, the index *i* corresponding to the i-th consecutive filtering of the image and the PointDataList *point_data_list* to update with the computed values. All the features must provide higher values for maxima which are more likely to correspond to the source and vice versa.
s We have thought of two main features so far:

- **intensity**: the intensity of the maximum under consideration. This feature is very significative and carries a lot of information about the presence of the source.
- **isolatedness**: the sum of the distances between a maximum and the k nearest ones. Generally speaking, the source tends to be more isolated than the rest of the maxima, but the information provided by this feature is often useless.

We now show two graphs computed on a sky map image. In each graph, each line represents the maxima trend for a specific measure across the sigma range:

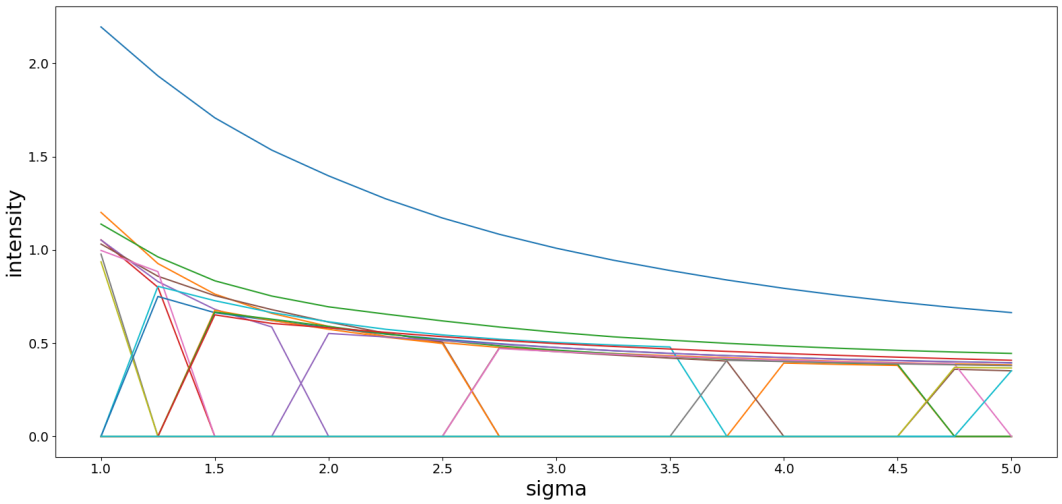


Figure 2.2: **Intensity** graph: we can notice a blue line significantly higher than all the others, corresponding to the source

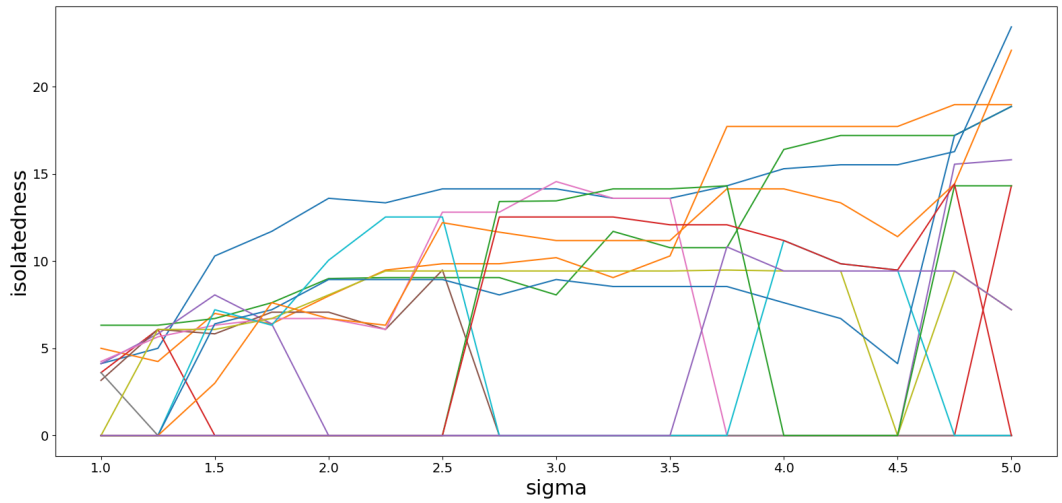


Figure 2.3: **Isolatedness** graph: we can notice a blue line significantly higher than all the others in the sigma range [1.5, 2.5], corresponding to the source

2.5 Election algorithm

After computing all the features described above for each maximum, we end up with a fully populated PointDataList for every feature (i.e. intensity, isolatedness). We now proceed with an election algorithm to find if there is a point which is more likely than the others to correspond to the source, among all the points belonging to the union of all the PointDataList. For every feature and for every step ($i = 1, ..., n$), we consider the two highest values v_1, v_2 ($v_1 \geq v_2$) for that particular step:

- if $v_1 > kv_2$, v_1 gets w votes, where k and w are two positive constants depending on the feature;
- otherwise, the same number of votes w is assigned to a special candidate that represents the absence of the source.

There are two possible outcomes:

1. The best scoring candidate is associated with a pair of coordinates which corresponds to the source;
2. The best scoring candidate is the one representing the absence of the source, meaning that there is no source in the image.

2.6 Configuration file

Along with the code, we provide a configuration file to set the most relevant parameters of the whole algorithm described so far. The configuration file is a JSON file that contains the following main parameters:

- **dir**: a string representing the path containing the skymap files that the algorithm should analyse;
- **sigma_array**: a list of values [σ_1, ... , σ_n] corresponding to the σ_i, $i = 1, ..., n$ values needed for filtering the images;
- **active_measures**: a list of strings representing the features that the algorithm takes into account. Each feature is also associated to a parameter corresponding to the name feature (e.g. intensity, isolatedness).

Each of these parameters contains the lists of specific parameters needed for that particular feature. Among these specific parameters there always must be four compulsory ones:

- **vote_thresh**: the value of k in the section "Election algorithm"
- **vote_weigth**: the value of w in the section "Election algorithm"
- **dist_thresh**: the value of r for the PointDataList corresponding to this feature
- **method**: the name of the method that computes the feature, that method will be called automatically if the parameter **active_measures** contains the name of this feature.

Chapter 3

Tests and Results

3.1 ctools

We tested our algorithm on sky maps simulated using **ctools**, a software package developed for the scientific analysis of Cherenkov Telescope Array (CTA) data ([1]). Before generating a sky map, we must generate an **event list** including photon events from astrophysical sources and background events from an instrumental background model ([2]). To do so, we must use the **ctobssim** tool, providing the following inputs:

- the Right Ascension of CTA pointing ra and the Declination of CTA pointing dec, in degrees;
- the radius of the simulation region rad, in degrees;
- a time interval defined by $tmin$ and $tmax$, in UTC format;
- an energy interval definedy by $emin$ and $emax$, in TeV;
- an instrumental response function irf;
- an input XML model $inmodel$: this is a source and background model definition file. We can set both the Right Ascension and the Declination of the source together with its flow and we can also set the background model.

In addition, if we want to generate different event samples in subsequent executions, we can provide a different $seed$ for each execution.

Once we have generated the event list, we can proceed generating the sky map using the **ctskymap** tool, providing the following inputs:

- an input event list $inobs$;
- the Right Ascension of the image centre $xref$ and the Declination of the image centre $yref$, in degrees;
- the projection method $proj$ and the coordinate system $coordsys$;
- the pixel size $binsz$, the size of the Right Ascension axis $nxpix$ and the size of the Declination axis $nypix$, in degrees;
- the energy interval defined by $emin$ and $emax$ within events are considered, in TeV.

3.2 Sky maps generation

We leveraged on the ctools library to implement two main functions for the automatic sky maps generation:

- **generate_src_data(start_model, flow, n, start_coords, radius, start_seed)**: it generates sky maps with one source, according to the following parameters:
  - **start_model**: source and background model XML file;
  - **flow**: flow value used for the simulations;
  - **n**: number of sky maps to generate;
  - **start_coords**: the coordinates the source will be placed around;
  - **radius**: the maximum distance between start_coords and the source;
  - **start_seed**: a positive integer used as a seed for the simulation.
- **generate_bkg_only_data(bkg_only_data, n, start_seed)**: it generates sky maps with only the background, according to the following parameters:
  - **bkg_only_model**: background model XML file;
  - **n**: number of sky maps to generate;
  - **start_seed**: a positive integer used as a seed for the simulation.

We generated two different sets of sky maps using the two previous methods:

1. The first set of **1000** sky maps was generated starting from an XML model of **both the source and the background**;
2. The second set of **1000** sky maps was generated starting from an XML model of **only the background**, with the purpose to simulate those images which do not contain any gamma-ray source.

As for the former case, we generated 1000 XML models with **source flow** equal to **2** and **source position** randomly set **within 1° from** the position **(221, 46)**.
As for the latter case, we simply used the same XML model, with the only definition of the background model.
The *XML models* were the only variable input parameter of **ctobssim** in the event list generation process together with the $\boldsymbol{seed}$, different for every execution. The other parameters were set according to the following scheme:

- $\boldsymbol{ra} = 221$ and $\boldsymbol{dec} = 46$;
- $\boldsymbol{rad} = 5$;
- $\boldsymbol{tmin}$ = 2020-01-01T00:00:00 and $\boldsymbol{tmin}$ = 2020-01-01T00:15:00;
- $\boldsymbol{emin} = 0.1$ and $\boldsymbol{emax} = 100$.

Sky maps were then generated using the following parameters for **ctskymap**:

- $\boldsymbol{xref} = 221$ and $\boldsymbol{yref} = 46$;
- $\boldsymbol{binsz} = 0.02$;
- $\boldsymbol{nxpix} = 200$ and $\boldsymbol{nypix} = 200$;
- $\boldsymbol{emin} = 0.1$ and $\boldsymbol{emax} = 100$.

3.3 Results

After generating the two sets of sky maps previously mentioned, we tested the algorithm over them. Different executions were needed to tune the configuration file parameters in order to **minimize** both the number of background only sky maps the algorithm found a gamma-ray source in (**false positives**) and the number of source and background sky maps the algorithm did not found the gamma-ray source in (**false negatives**). Here is the resulting confusion matrix:

		Prediction		
		Positive	Negative	Total
Label	Positive	997	3	1000
	Negative	3	997	1000
	Total	1000	1000	2000

Table 3.1: Confusion matrix

We can observe that both the **false positives rate** and the **false negatives rate** are equal to 0.3%.

3.4 Conclusions

Although we achieved a low error on both the false positives and the false negatives, from the purely scientific point of view we should prioritize the minimization of false positives rather than false negatives. Further parameters tuning and local analysis may be deployed to achieve better results.
Another problem concerns the election algorithm mentioned in the section 2.5: in order to make the algorithm work properly, v_1/v_2 should be scale-invariant for all the features. As for the intensity values, their ratios are clearly not scale-invariant, since values tend asymptotically to the mean of all the pixels in the image (Gaussian smooth with $\sigma \to +\infty$). Making those ratios scale-invariant across the σ range would be a clear improvement.

Bibliography

[1] http://cta.irap.omp.eu/ctools/index.html

[2] http://cta.irap.omp.eu/ctools/users/reference_manual/ctobssim.html#ctobssim