Intelligent and autonomous vehicles

Athens - TA23

ENSTA ParisTech

Benjamin Lazard (Télécom ParisTech)
Nicolas Cadart (ENSTA ParisTech)

November 18, 2016

Contents

1 Processing of LIDAR data

1.1 Objective

Using data from a video camera and LIDAR scans, we want to detect the ground plane and the obstacles. The pratical questions for a driver would be : where is the road ? what is its curve ? Is it flat or bumpy ? Are there obstacles ? How close ?

1.2 Data type

In this exercise, we deal with 3 sets of {camera image, velodyne LIDAR cloud point, transformation matrices}. Each image is a 3D matrix : the image itself is made of 1242×375 pixels, each having RGB 1-byte values. The size of the cloud point `velo` varies, however it is roughly 10^5 points wide. `velo` is therefore quite a large vector, the lines of which are x, y, z cartesian coordinates of a scanned point.

1.3 Our work

1.3.1 Projection of LIDAR points on the camera plane

For each camera image corresponding to the driver's perspective, we overlaid the LIDAR points projected thanks to the successive application of the transformation matrices (these matrices are fixed parameters and can be determined using solely the position of the camera and LIDAR sensors on the car).

First, we extended `velo` by adding ones as a fourth dimension so as to perform both rotations and translations with only one matrix. Step 1 is a projection on the camera plane, step 2 is a rectification, step 3 is a normalization of these coordinates (since we have all points gathered in the same `rectified_camera` plane, we can consider that z=1).

1.3.2 Display of the results

Then, we removed the points that did not fit the camera angle (that were outside the 1242×375 pictures). Indeed the velodyne LIDAR is a 360-degree view sensor.

Eventually, we used a colorset to distinguish different areas on the image.

— The green points are the projected points of the LIDAR on the `camera_plane` corresponding to the upper half of the camera image. These points are basically obtained by the LIDAR whilst pointing at the sky, and we do not need them for the analysis of driving aleas.
— The red points form the ground. Assuming the exactitude of the positions of the camera and sensors and therefore of the transformation matrices, any asymmetry is (supposed to be) due to the road not being smooth.
— The blue points are either walls or detected potential obstacles. These last ones are obtained by setting the effective height of the camera, selecting only points "high enough" in the `rectified_camera_plane`. We found a satisfying value of $y = 1.62$.

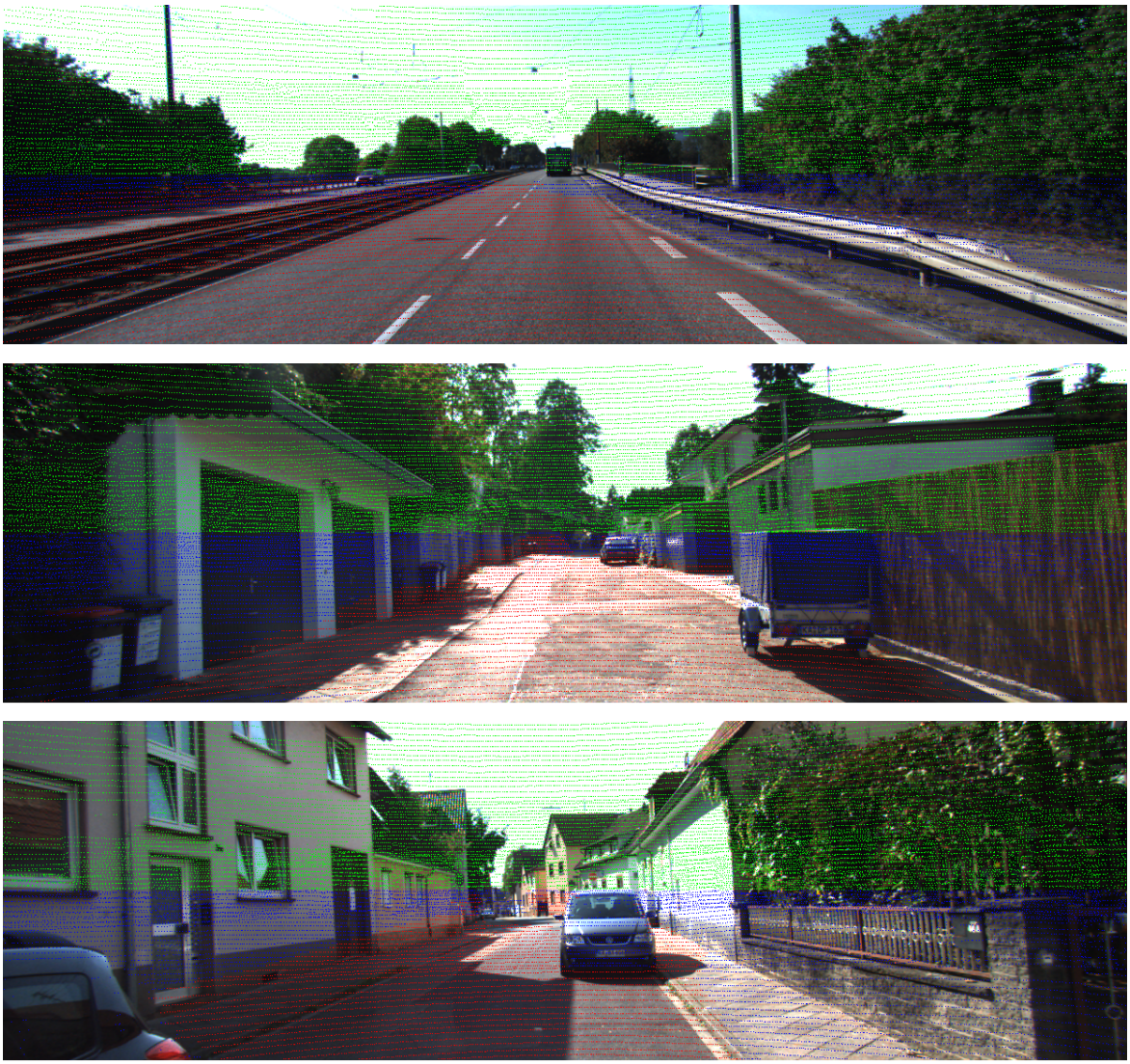


FIGURE 1 – LIDAR analysis of the situation

1.4 Conclusion

After going through this process, we are quite satisfied with the result obtained. However, one must recognize that our calculi lack precision : the kerb is only approximately detected, and on the expressway, the limit between the two directions of circulation is fuzzy. But obstacles are quite well spotted, and the general perspective and curving of the road is well analyzed.

2 Detection of traffic lights

2.1 Objective

The detection of traffic lights, other cars'lights, and roadsigns constitutes a major issue for autonomous cars. Using camera images, the objective here is to find points with a high power of light incident in red spectrum, as areas of interests to be further analysed.

2.2 Data type and general idea

(a) RGB image (b) Lab image

FIGURE 2 – Camera images

To do this, we have at our disposal the camera images into the RGB and $L \times a \times b$ channels. Our goal is to calculate a matrix F representing the level of red of each pixel. It is then possible to find and indicate the F maximum values area on our camera image, corresponding to potential red traffic lights.

2.3 Our work

2.3.1 Calculation of F

As usual, we will consider our Lab image as a simple 3D matrix, where each dimension represents the L, a or b channel. Therefore, the matrix F can be easily calculated by the formula $F_{ij} = L_{ij} * (a_{ij} + b_{ij})$.

2.3.2 Detection of the maximal values of F

We now have to write a function `detectMaxima` which finds the maximal values of F, or in other words, the brightest red pixels of the camera image. We proceed in an iterative fashion :

1. find and store the maximal value of F and its position
2. suppress a small area around it
3. repeat from 1. until a fixed number of iterations has been performed.

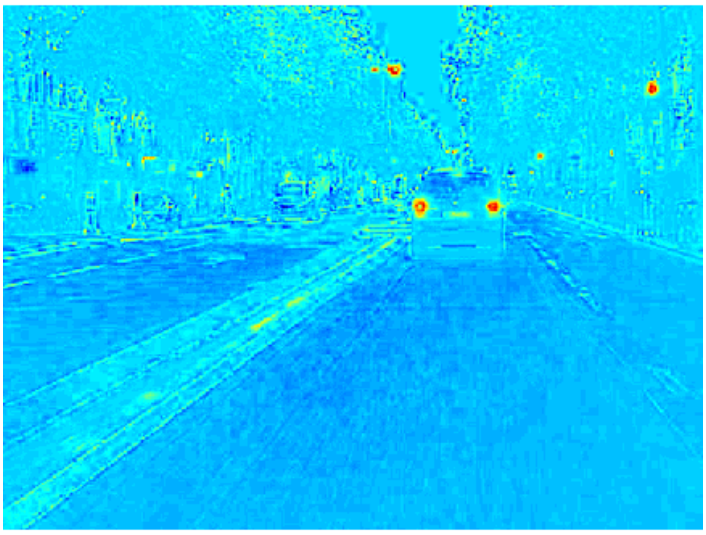


FIGURE 3 – F matrix showing the red pixels intensity

By "suppressing a small area around the maximum", we mean putting all the values of this area to F minimal value. This step is essential, and mustn't be forgotten. Indeed, as each max area is represented through numerous pixels, the algorithm just skips other pixels deemed to belong to the same object, which allows to identify other bright red bodies with minimum processing power.

Here is the Matlab code for this function :

```
% function that looks for the N strongest values in a 2D matrix F
% Return values:
%   x,y,maxVals - vectors with nrMaxima entries representing the x/y
%                 position and strength of each found maximum
% parameters:
%   F - a 2D matrix of doubles
%   nrMaxima - how many maxima shall be found?
%   boxSize - how large is the area around each found maximum that should
%             be erased?

function  [x,y,maxVals] = detectMaxima(F, nrMaxima, boxSize)

  x = zeros(1,nrMaxima) ;
  y = zeros(1,nrMaxima) ;
  maxVals = zeros(1,nrMaxima) ;

  for max_id=1:1:nrMaxima
      % find the maximum value and coordinates
      [max_value, max_coord] = max(F(:)) ;
      [max_row, max_col] = ind2sub(size(F),max_coord);
      x(max_id) = max_col ;
      y(max_id) = max_row ;
      maxVals(max_id) = max_value ;

      % erase the area around the maximum
      y_min = round( max(1,          max_row - boxSize/2));
      y_max = round( min(size(F,1), max_row + boxSize/2)) ;
```

```
      x_min = round( max(1,          max_col - boxSize/2));
      x_max = round( min(size(F,2), max_col + boxSize/2));
      F(y_min:y_max, x_min:x_max) = min(F(:)) ;
  end
end
```

2.3.3 Results of the algorithm

We are now able to display the results. We add a rectangle around each detected red area, indicating as well the red brightness. However, we can play on several parameters to obtain the best possible detection.

— **Influence of the number of maxima and the suppression box size :** Low suppression leads to the fact that all allowed number of maxima is wasted around a couple of sources and some stoplights or backlights might not be detected. High suppression `boxSize` value (about 30 px) combined with a substantial value of `nrMaxima` (about 10) ensures that all meaningful sources are not overlooked.

— **Influence of red brightness threshold :** It is also interesting to modify the red brightness threshold value in order to restrict what is displayed. On the one hand, a low value (∼10000) surely detects all the traffic lights. However, this isn't the best choice, as lots of other red objects (signs, cars,...) are detected as well, which will lead to unnecessary braking of our car... On the other hand, a high value (∼18000) leads to an algorithm making rare mistakes, but there is the risk that it misses some of the lights, which is unacceptable, as it may lead to an accident ! A good compromise is made for `thresholdRed` = 16000.

FIGURE 4 – Detection of traffic lights

2.4 Conclusion

The detection of traffic lights by this simple algorithm based on the analysis of a Lab image is efficient, although some overdetection or unspotted lights might occur. The influence of several factors such as red threshold value, number of maxima or suppression box size was considered in order to establish the best combination from the position of object detection and processing power.

3 Image processing by gradient filter

3.1 Objective

The first step to object recognition is object detection. We have already performed color pattern recognition, and now we want to perform basic shape pre-treatement.

3.2 Data type

Here, we deal with 640×480 px black & white images taken by the camera in the driver's perspective. We want to perform image processing operations to detect the contours, and therefore just need the average pixel variation, independently from the colors of the shapes.

3.3 Our work

First, we apply a gradient filter on the x dimension, then on the y dimension. This operation corresponds basically to an averaging of color change, to keep only quick pixel variations in the image (which define the outline of an object). N levels of averaging are tested, according to the following leveling equation :

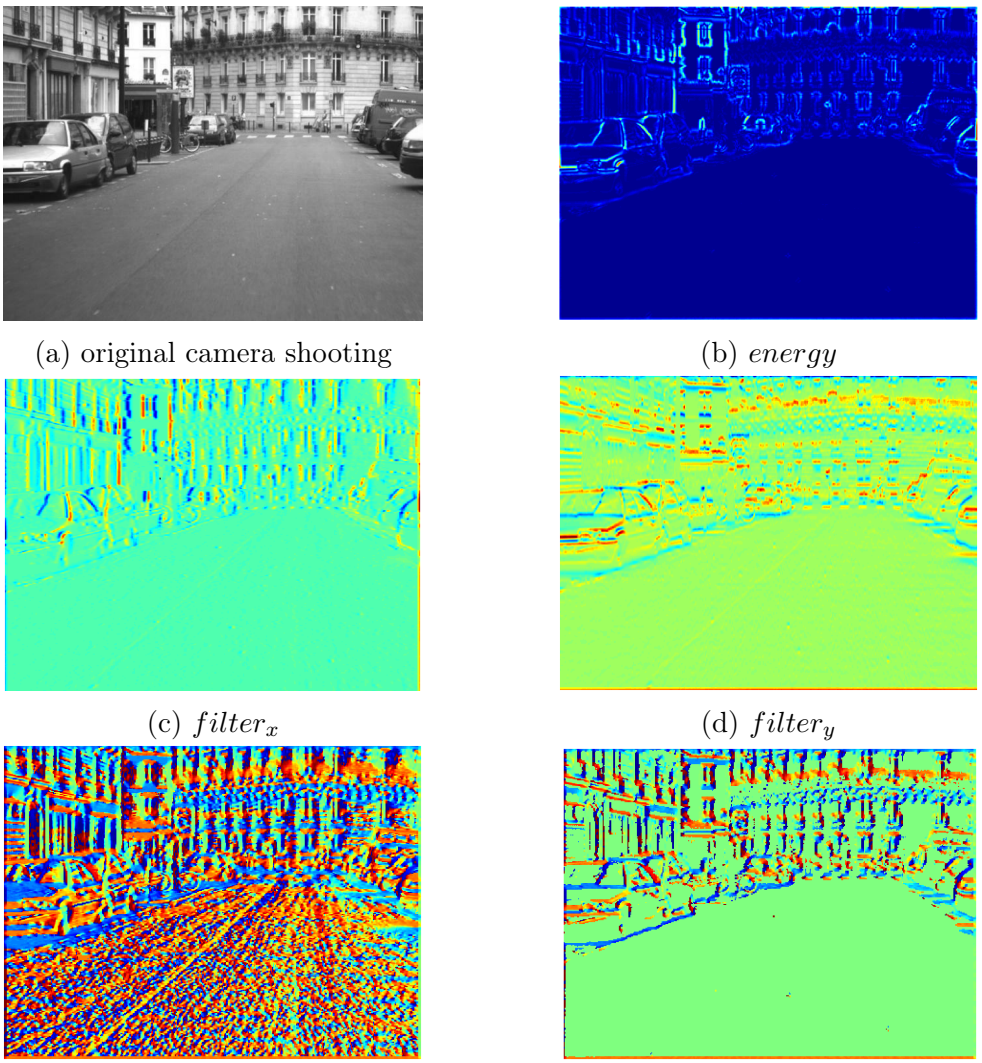
$$y_n = \sum_{k=0}^{\lfloor N/2 \rfloor - 1} x_{n-k} - \sum_{k=\lfloor N/2 \rfloor}^{N} x_{n-k}$$

which can be written as a polynomial filter, with a numerator $[1 \dots 1 \ -1 \dots -1]$. The mathematics behind this is Z-Fourier Transform.

Then we calculate the energy as the euclidian norm : $energy = \sqrt{filter_x^2 + filter_y^2}$. In practice, we calculate $energy^2$, and then power it by 0.7 (a value supposed to erase the small gaps, thus highlighting the biggest).

Another way to combine shape detection in both dimensions is to calculate the overall "angle detection" using the ratio of these filters (to combine the shape detection in both dimensions), and accentuate the biggest discrepancies thus detected using `atan` function. It can be filtered once more keeping only the values of the matrix where the *energy* is above a threshold − say for example quantile 70%.

Here is a demonstration of those filters.



(a) original camera shooting

(b) *energy*

(c) $filter_x$

(d) $filter_y$

(e) *angles_raw*

(f) filtered angles

FIGURE 5 – gradient filters

3.4 Conclusion

The shapes are correctly approximated. For smart cars application, one must notice that the building might be over-contrasted, however, the street has been correctly detected as a free area, and the surrounding cars are probably ready for machine learning analysis, since they have been reduced to a few strokes.

4 Introduction to machine learning

4.1 Objective

To distinguish the various road signs, and the categories of surrounding objects (pedestrians, cars, trucks, trees,...), we must use machine learning tool. The objective of the fourth exercise is simply to discover the underlying topics such as gradient descent algorithm, training set vs test set discrepancy, and over-prediction.

4.2 Data type and general idea

The context is a bit abstract here. The idea is that given a variable x, which is an observation (like a measurement by one of the cars' sensors), we want to predict the category y to which it corresponds. For example, from $x = picture\ of\ an\ object$, determine $y \in \{stepmother\ crossing, \quad stop\ sign, \quad car, \quad palm\ tree\}$. As any human could tell the category of x (except if you really hate your stepmother), we assume that it is no subjective judging, and that there is a function f such that $f(x) = y$. The goal of machine learning is to approximate f.

To do so, we consider 2 sets of data.

— The 1^{st} one : `training` is not very large, and is used for training. We have a vector $X_{training} = (x_1, ..., x_n)$ and we know its category : $Y_{training} = (y_1, ..., y_n)$. Based on this information, we want to approximate f as a polynomial function $\hat{f}$. The degree of $\hat{f}$ is called complexity C. $\hat{f}(x) = \sum_{i=0}^{C} w_i\, x^i$ is computed by minimizing the quadratic mean error $E = \frac{1}{N} \sum_{l=1}^{n} \left(y_l - \hat{f}\,(x_l) \right)^2$ thanks to the gradient descent algorithm, over `nr_iterations` on the set.

— The 2^{nd} one : `test`, is a set $\{X_{test}, Y_{test}\}$ different from the previous one (and much larger). We want to test $\hat{f}$ computed with the previous set on X_{test}, and check whether the result $Y_{predicted}$ is close to human perception Y_{test}.

4.3 Our work

After building our own sets of data, we have to proceed as described in the previous section. The model used is linear : $y_i = 2\,x_i + 1 + noise$. In reality, the noise is a random parameter expected to be small compared to x_i in general, and corresponding to wrong sensor measurements, or lack of precision of the analysis...

4.3.1 Influence of the model complexity on the approximation error

Let us try different $\hat{f}$, changing the complexity, with $n = 40$ and `nr_iteration` = 100 (please run the program for detailed results, the procedure and plotting are automatized).

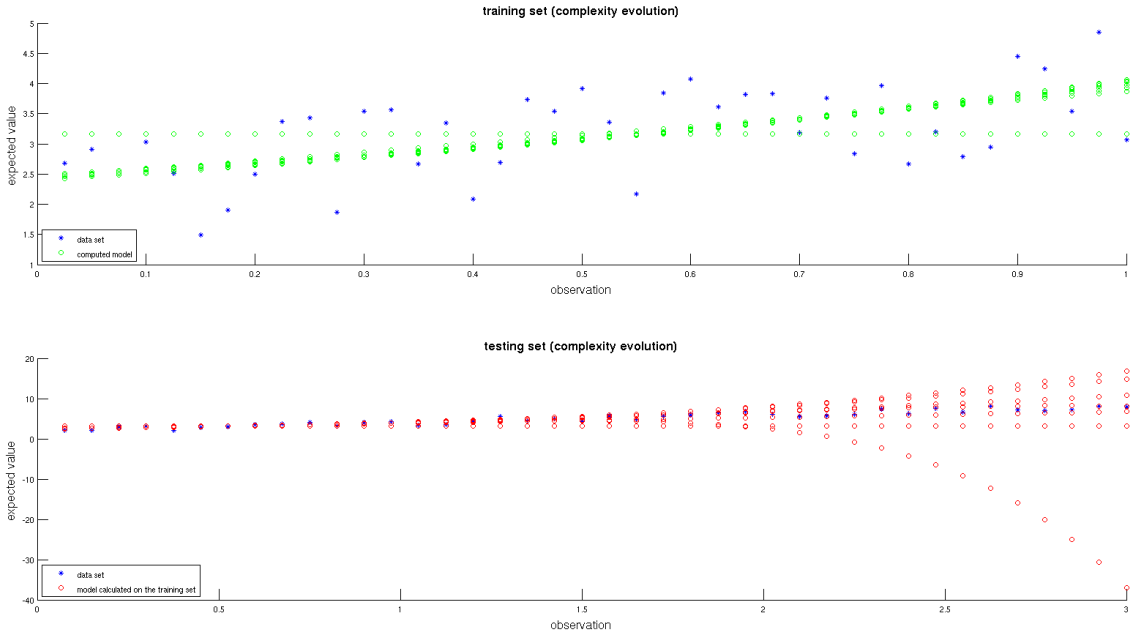


FIGURE 6 – impact of complexity $\in [\![0, 6]\!]$

First thing noticeable is that the result in the training set is almost always satisfactory. However, as there is a lot of noise there, the model that minimizes training error is not the one expected : complexity = 6 is the best parameter in the training set (whereas expected complexity = 1 since the model is linear)... In fact, the more we enable the use of a poles for f, the more it will fit to the set of points. However, the generalization for complexity = 6 is totally irrelevant. Indeed, an approximation is always valid in a specific range. Therefore, as the test set gets further than the training values, the coefficients related to high powers get to be insignificant, increasing sharply the error. As expected, the minimum error is obtained with a complexity = 1, agreeing with the linear model. Moreover, we see that the gap between training set and test set is bigger when x_{test} gets very different from $x_{training}$.That shows just how careful we have to be while choosing the training set. We have to make it cover most possible real-life x.

4.3.2 Influence of the accuracy of the gradient descent on the approximation error

Then let us test the impact of `nr_iterations` on $\hat{f}$. If you ran our program, this graph should display as well on figure 2 in Matlab.

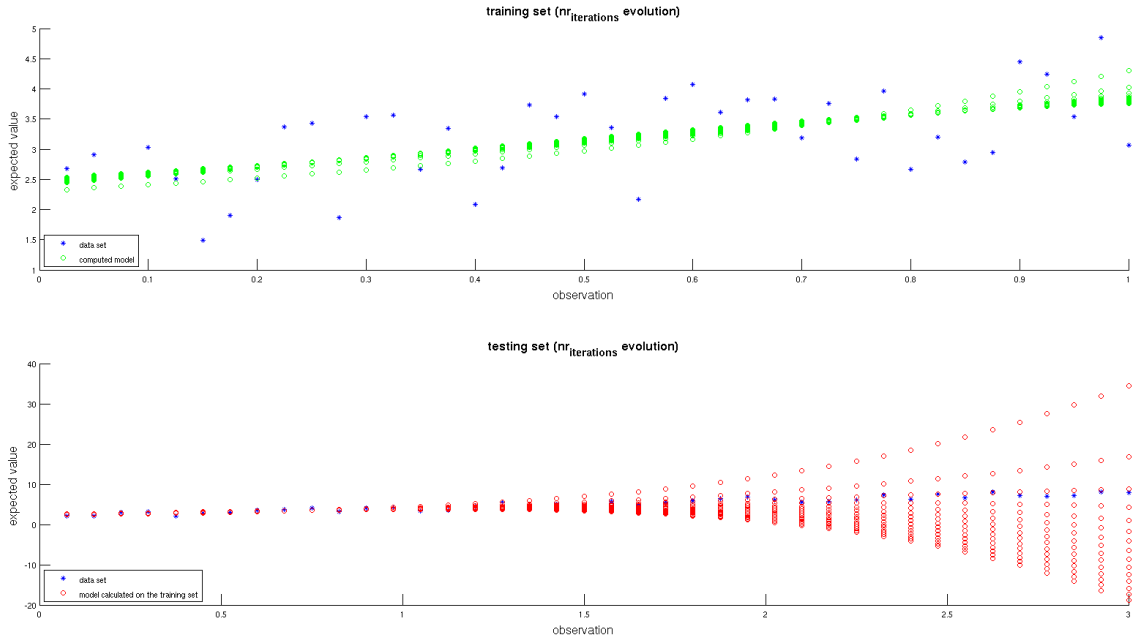


FIGURE 7 – impact of nr_iterations $\in [\![50, 750]\!]$

We can see here that as `nr_iterations` grows, past a certain threshold `nr_iterations` $= 150 \pm 50$, the training error decreases, and the testing error increases. This phenomenon is known as over-predicting, and shows that it is no use trying to nullify the training error, because otherwise, instead of cleverly comparing the test set with the training set, we rather perform simple picking from the training set, with weird behavior if a test set element cannot be found in the training set.

5 Path finding with Dijkstra's algorithm

5.1 Objective

After our autonomous car has been equipped with all its sensors and is able to move avoiding the obstacles, we need to know how to get to the destination the fastest way. In this exercise, we will implement the Dijkstra's algorithm. The aim of this algorithm, used by the GPS, is to determine the best path to join a *start* point to a *stop* point, minimizing one criterion (distance, time, fuel, money...).

5.2 Data type

We are given a map where 17 nodes and 22 edges between these nodes are depicted along with their respective costs. The aim is to find the path with lowest cost from Paris (Node 1) to Cologne (Node 17).

5.3 Our work

5.3.1 Construction of the cost matrix

The first step is to represent the map as a matrix, where `costMatrix(a,b)` represents the cost value of the edge from node `a` to node `b`. If these two nodes are not linked, then we put the cost of this edge to infinity. Moreover, we suppose here that the cost to go from `a` to `b` is the same as the cost of going from `b` to `a`. As a result, `costMatrix` is a symmetric matrix. However, putting different costs for the two directions could be interesting, representing for example a traffic jam on one side of the road.

5.3.2 Frontier calculation

This is the most tricky part of the algorithm. To find the best path, we visit each node, and compute the best and shortest way to reach it, until we've tested all the nodes.

We create `S` (the set of visited nodes initialized as `start`), `dist` (the lengths of the shortest paths from `start` to each node) and `prev` (the list of penultimates nodes in the optimal path from `start` to each node).

At each iteration of the algorithm, we need `S` to sprawl, and `dist` and `prev` to update, until all nodes get visited. This is done through the calculation of the `frontier`.

`frontier` will always be a vector, where `frontier(i)` is the total cost from the start node to that node `i`. If that node has already been visited, or isn't directly linked to a node which has been, its frontier value is set to infinity. To calculate it, we proceed in an iterative fashion :

1. Set `frontier` values to infinity.
2. Select a visited point (named `node`) (*i.e.* from `S`)
3. If this `node` is linked to a point (named `dest_node`) that hasn't been visited yet (*i.e.* $\notin$ `S`) , we add this accessible `dest_node` to the frontier by indicating the cost of the shortest path to reach it from the start node

4. We update the vector `prev`, for this one node (if the shortest way to reach `dest_node` is `node`, and not another node from `S`)
5. If `S` has not been completely reviewed, go back to step 2. with another `node`

5.3.3 Selection of the closest frontier node and updating

In each loop of the algorithm, once frontier is computed, we select the node from the frontier which is the closest to the start node, and we set it to visited (*i.e.* add it to `S`). Eventually, we update the `dist` vector which represent the distance of the shortest path to this "closest `dest_node`" from the start.

Then, if all nodes have not been visited yet, we loop again through all this steps (calculation of the frontier, selection of the best node, update of the best previous point and the shortest cost to each point).

5.3.4 Extraction of the shortest path

Now that we have the lowest cost to reach the stop node (`dist(stop)`), we need to determine the path. To do so, we use recursively the vector `prev` from the stop node, until we reach from the `start` : what is the previous "best point" to reach the stop node ? And the one before ? And before ? ... We now know the best path to join the start and stop nodes, and the length of it.

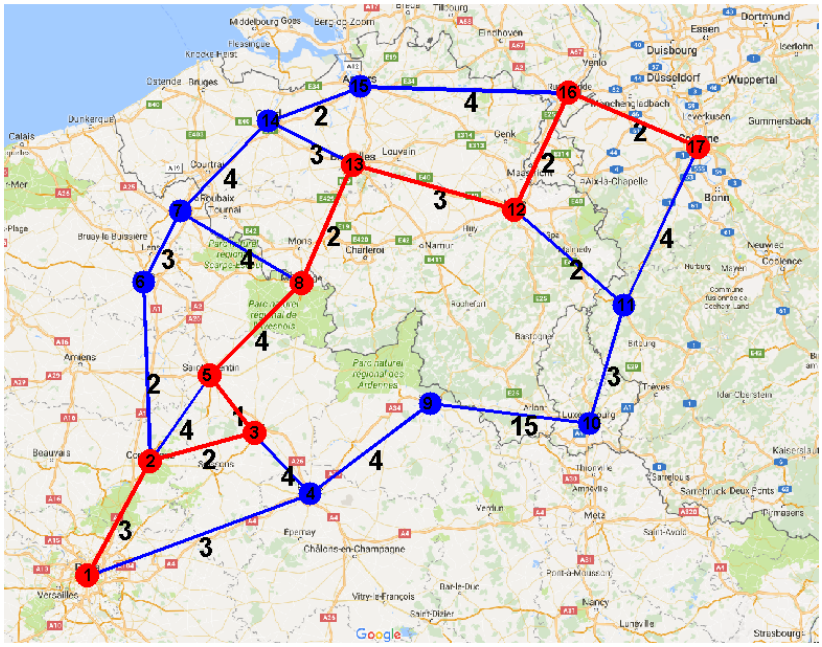


FIGURE 8 – Path found with a traffic jam between node 9 and 10, length = 19

5.4 Conclusion

This algorithm works well. However, we can notice that if two shortest path are possible (which was the case here), it will only display one of the two.

As defined here, the problem is also easily adjustable. For example, a lowered traffic can be seen as an increase of the cost of the edge in the affected area.

We can also impose a stage at our journey. To do so in our route calculation, we just need to run twice the algorithm, from the start node to the stage node, then from the stage node to the stop node.

Lastly, this algorithm can be used to solve many other different problems of optimization, such as finding the best manufacturing strategy, where a node could be the type of good or a mode of production, and the cost edge would be the price of it.