Product recognition on store shelves

Computer Vision 2017-2018

Marco Di Vincenzo

Summary

1. Introduction

The overall task for this project is to develop a system capable of, given a set of product models, recognizing them in store shelves pictures. This system could be used to help visually impaired people browsing groceries or to automatically detect low in stock or misplaced products. In order to keep complexity down, we are only considering cereal packages since their characteristic traits are all in the flat front side.
In the main topics of this report, only the general ideas and reasoning behind the proposed solution will be presented. For full implementation details and practical considerations, please refer to section 28 – Implementation details.

2. Goals

This system, given a picture of a shelf, must output for each recognized model their number of instances and for each instance their position and detected scale. The project will be divided in two main steps (plus one final challenge), of increasing complexity.
The first goal to reach is to successfully recognize models assuming only one instance is present at most, while the second one consists in recognizing an arbitrary number of instances. More precisely, the problem statement suggests to use local invariant features to accomplish the first step and the generalized Hough transform (in conjunction with local invariant features) to accomplish the second. Since no indication about the final challenge is provided, various approaches were tried but most of the focus went on the Hough transform.
No explicit real-time requirement is stated, so, throughout the whole project, precision has been preferred over speed, even though an effort has been made to keep everything as efficient as possible.

3. Setup

The project has been developed leveraging on the power of the OpenCV 3 library, and it has been written in C++ (standard 14). All of the code (including the test images) is available at the author's GitHub profile (https://github.com/euneirophrenia/CVProjectWork).
The images (models and test scenes) to work with were provided beforehand and consist of 27 models and 15 test scenes.

Every experiment has been carried out on a MacBook Pro 2013, 8 GB di RAM, 2.4 GHz i7.
In the following, some time measurements will be shown but those are relative to the before-mentioned architecture and may vary a bit on other systems, depending on the hardware.

4. Problem Analysis

Other than the explicit requirements stated in the problem description, there are other things to care about.
First of all, the models provided are acquired in a very heterogeneous way: there are some with a very high resolution (e.g. 1005 x 1500) and some with a very low resolution (e.g. 161 x 238). Moreover, there are many models that are very similar to each other since they are only flavor variations of the same "base" product. In facts, there are also two perfectly identical models (*9.jpg* and *23.jpg*), possibly due to some human mistake during acquisition.
This situation makes it potentially hard to detect some models thus requiring some attention.

The scene images, on the other hand, are a lot more homogeneous (excluding those used in the final challenge, which, of course, is supposed to be challenging). However, they are rather blurry, probably due to the camera used to acquire them, and they present some "distracting elements" such as price tags or other products not included in the models. It is also worthy to notice that sometimes, the models appear in the scene with some slight variation (e.g. cereal boxes with some bonus gift within advertised, while the advertisement is not shown in the model).

These problems call for a *preprocessing* step in which we can try to solve (or at least to ease) some of these hardships. Since the models are a given, we can ideally preprocess them once, "offline", and store the results for later usage. This means that we can ideally preprocess model heavily without impact on the run-time performances. Scenes, instead, even though were given to us just like the models, must probably be considered an input that in the actual system will be acquired at run time, so it is important that their preprocessing is as fast as possible.

All in all, we can derive from these problems the first approximated pipeline that will (hopefully) lead to a solution, as shown in *figure 1*, right here below.

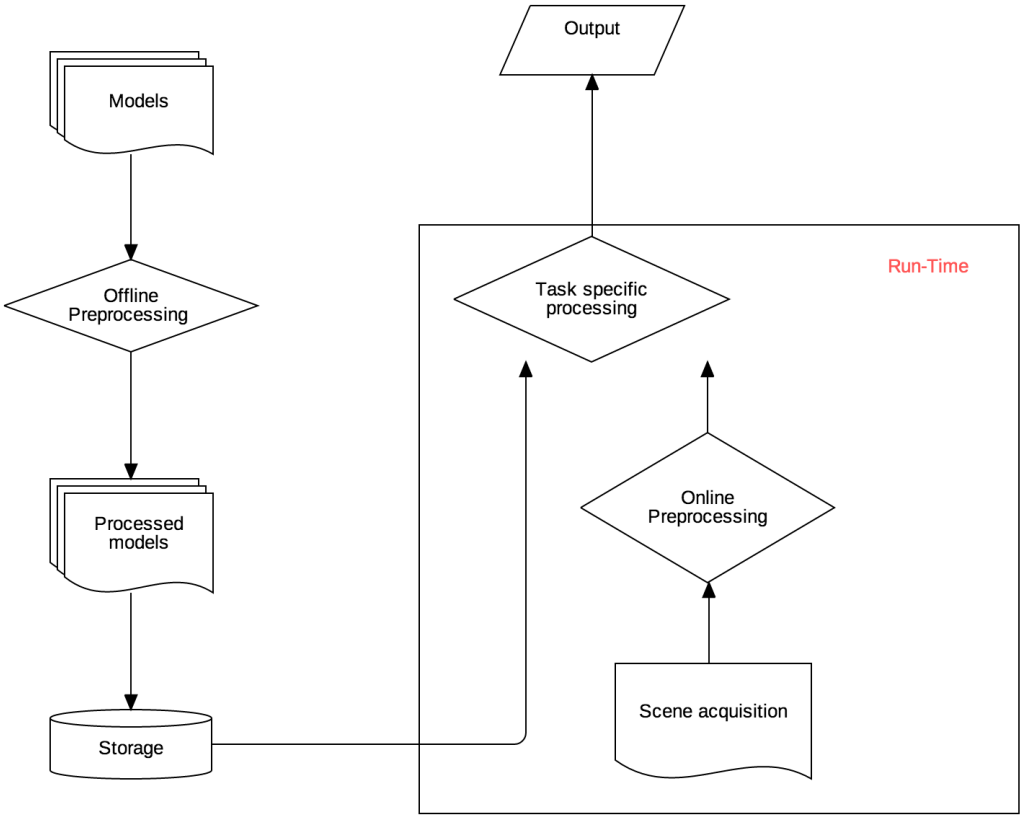


Figure 1 The general Pipeline

More details on the implementation of each steps will be provided soon, following the order of the problem tasks. As a general consideration, the problem text strongly hinted to the generalized Hough transform as a mean to achieve the final toughest goal. For this reason, every consideration has been carried out, tested and tuned with that particular approach in mind.

5. Offline preprocessing

During development of the project, this step has been thought about several times.

The only constant idea was to work on gray-scale images; partially because it is more efficient then working on color image but also because color is not a reliable information without proper lighting control.

Since the very beginning, it was clear that it was pointless to have 1000x1500 images when trying to match them in blurry scenes at a lower scale. This would be rather counterproductive since matching high resolution models to low resolution scenes would lead to a lot of spurious matches, potentially hiding other correct (low resolution) model

matches. So, the first thought was to scale every model to a common reference size, and to blur them accordingly. Soon enough, "magic numbers" started to appear in the project, representing choices such as the blurring kernel size or the reference scale. While fixed magic numbers may work smoothly in the two basic steps required, they fall *very* short in the final challenge, where models appear at a much lower scale. So, instead of pursuing a way in which we detect by hand one fine tuning of those magic numbers for each step, it has been decided to find a more general way to automatically compute those numbers, so that the overall system could be more consistent and robust with respect to scale changes. Surely, this didn't prevent the need for some tuning and also ended up leaving an impact on performances, however, as we will discuss soon, both the impact on performances and the amount of tuning needed are extremely low.

5.1 Preprocessing – Resizing the models

The proposed idea is the most obvious one: to scale every model down to the scale of the smallest one, while possibly preserving proportions (every model has in fact a different aspect ratio, it's best not to alter it). So, the first step was to determine the lowest scale model and this was done (arbitrarily) by finding the model with lowest area (measured as *width x height*). Before resizing, a Gaussian Filter is applied to denoise the image (and also to prevent artifacts while down-sampling, as signal theory teaches us). The size of the gaussian kernel was tuned by hand and kept rather small in size (3x3), with a σ computed based on the scaling factor (the higher the scale factor, the higher σ), such that if no scaling is involved (i.e. when handling the smallest model) no filtering is involved. Since some models are already at a low resolution, an excessive filtering may destroy important features making it hard to tell apart similar models. The kernel size choice was determined by looking at the effects induced on the performances of the classification task (i.e. on the accuracy), both in the base steps and in the final challenge.

After that, knowing the smallest model, we can choose one dimension (i.e. in this case the *height*) and scale it to match the one of the smallest model, scaling the other dimension (i.e. the *width*) to maintain the original aspect ratio.

However, given the really small size of the smallest model, in this way it was really hard to tell apart similar models, since some useful features (e.g. small writings on the boxes) got lost. So, while keeping the general idea, it was decided to settle for 1.5 times the smallest size. This measure is one of the very few hand-tuned features and was deemed a reasonable

tradeoff between the idea explained above and the need to preserve meaningful features: it is not high enough to introduce sensible artifacts (even when scaling up) but high enough to preserve useful details, across the models.

Another idea that has been taken into account was to handle also the model preprocessing at run-time. While this may sound as a bad idea (and in facts it is with respect to execution time), this approach allowed to, in principle, scale the models to roughly match the size at which they appear in the scene (more on this esteem in the following) and this, in turn, allows us to use template matching techniques efficiently rather than local invariant features and the generalized Hough transform. Also, while using the SIFT paradigm to match local invariant features, having images already at the scale in which their most relevant features appear allow us to limit the octave number while still getting good results. However, empirical results show that this is a non-factor probably due to the really good OpenCV implementation of the SIFT paradigm: even limiting the octave number did not produce sensible improvements in computation time. Most probably, the little time saved was lost approximating the model scale in the scene.
Ultimately, the run-time preprocessing of models was deemed a bad idea as also the template matching approach was found incapable of providing better results than using local invariant features and the generalized Hough transform.

5.2 Preprocessing – Image sharpening and de-blurring

Due to the generally low resolution and blurry nature of the models (and the scenes, as well), an image sharpening approach has been tried. However, possibly due to some mistakes in the implementation, this approach did not lead to an improvement in classification precision. Most noticeably, the proposed solution properly classifies every single model in every single test scene (except the final challenge) even without de-blurring, whereas in the final challenge de-blurring introduces some artifacts, allowing for detection of some models instead of others.
The idea behind the implementation is rather simple: create a blurred image and subtract it from the original image. More precisely, if we did a 1:1 subtraction, the result would be mostly dark, so it is common practice to instead perform a weighted sum with the weights summing up to 1 (e.g. 1.5 for the original image, -0.5 for the blurred version). The results are generally good per se, but at the end of the day they were not able to improve the results and thus they were discarded.

Figure 2.1 A model before de-blurring

Figure 2.2 The same model after de-blurring

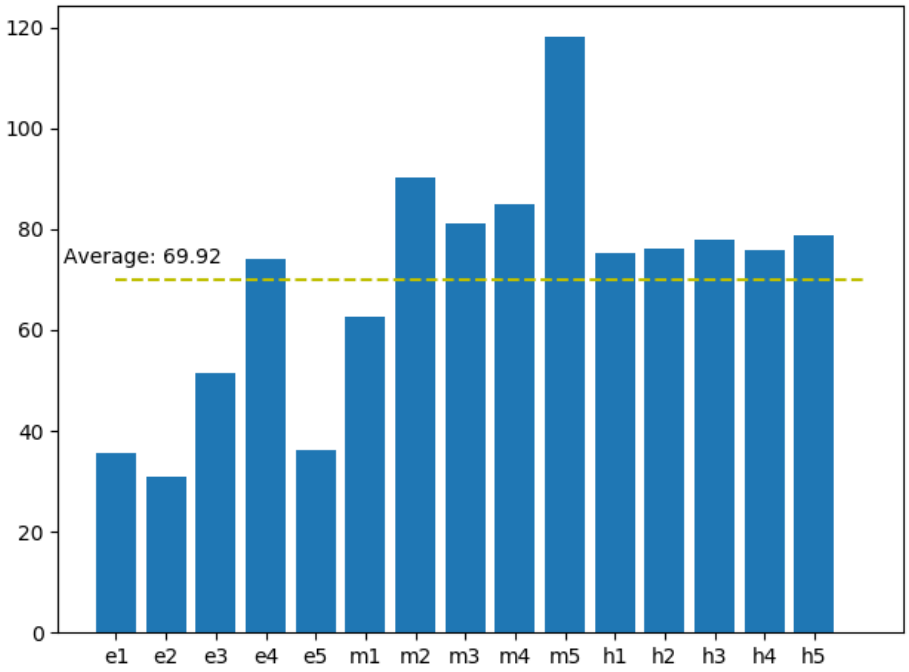
5.3 Preprocessing – Scale esteem

Albeit not used while preprocessing the models, an esteem of the products' scale in the scene was deemed useful during the matching phase (we will see why in the following). The key idea is to find the longest vertical line in the scene and use that as an esteem of the height of the models in the scene. This assumption is entirely based on the nature of the scenes, and it was deemed the most robust one. In facts, any rotation of the models around a vertical axis (which are the most likely to happen) will not change the measurement, since rotations don't alter lines parallel to their axes. On the other hand, rotations around an axis perpendicular to the image plane (i.e. a rotation that would make the model lean towards the camera or the opposite way) would change this measurement. However, we are only considering the *longest* vertical line and even in the unlikely event where all the products were skewed, the space between the shelf's planes may as well provide a useful vertical line.

In order to detect vertical lines, the OpenCV probabilistic implementation of the Hough transform has been used. More precisely, first we extract edges from the image, using the OpenCV implementation of the Canny's edge detector and then feed the edges to the probabilistic Hough line transform that will output starting and ending points of the detected

lines. A vertical line will be one whose starting and ending *x* coordinate is constant, allowing for an easy computation of the longest line.

Of course, this is something computed at run-time, but only computed once per scene throughout the whole execution, and it's computed fast enough, as shown in the graph below.



Graph 1 Average execution time in milliseconds for each scene scale approximation

The graph shows the distribution of the average execution time (in milliseconds) computed iterating the esteem for each image 20 times. The average of the averages is highlighted in yellow and it is roughly 70ms.

5.4 Preprocessing – Extracting local invariant features

Many alternative extractors and descriptors were considered. In particular, by experimental results, SIFT descriptors proved to be the most effective in terms of precision. SURF, ORB and RISK were all faster than SIFT but also less precise and for this reason, discarded. It is important to notice that in the first step, given the restrictive hypothesis, all of the above descriptors performed reasonably well. However, as the development pushed towards less constrained steps, SIFT was the clear winner.

Most noticeably, to further improve performance, it has been decided to opt for a variation of the SIFT paradigm: *RootSIFT*. As the name may suggests, it basically consists in extracting a square root from the SIFT descriptors, after a L-1 normalization. Everything else remains

totally unchanged. The reason behind, as explained in the proposing paper [1], is that Euclidean distance is not really optimal for comparing histograms (such as the SIFT descriptors) and often leads to inferior performances compared to using other measures (e.g. χ^2 or the Hellinger kernel). Basically, this simple correction of the descriptors makes the normal SIFT matching behave as if we were using a better distance measure, improving the matching precision without introducing any significant computational overhead. RootSIFT has been deployed in every step of the process and applied both to the models and to the scene. Even though the SIFT paradigm alone gives good results, as we can see here below, sometimes it is not enough. Notice how *model 11* (the middle one) gets wrongly located on top of *model 1* (the right-most one) when using SIFT only.

Figure 3 Scene e3 wrongly classified by using only SIFT

Figure 4 Scene e3 properly classified by RootSIFT

5.5 Preprocessing – Similarity detection

As anticipated before, there are many couples of models very similar to each other. However, a formal measurement of this similarity is needed. It seemed reasonable building a "similarity table", basically a *N x N* table in which each cell *(i, j)* contains a measure of how similar the i-th model is to the j-th model. Since all the project revolves around local invariant features, those have been used to compute those similarity measures as well. More precisely, after the rescaling phase, SIFT-compatible features are extracted from each model and then matched against each other accordingly. The ratio between the number of "good" matches found and the number of key-points was used as similarity measure, where a "good" match is considered such if its distance ratio to the second nearest neighbor is lower than a fixed threshold (as described in Lowe's paper, who tuned the threshold to 0.7 while 0.65 turned out to work best in this case). Notice that this way the similarity table is not necessarily symmetric, since similar models may have different number of relevant key-points. In practice, for each couple of models (*m1, m2*), the similarity score has been computed as follows:

$$sim(m1, m2) = \frac{good\ matches(m1, m2)}{number\ of\ keypoints\ in\ m1}$$

While the symmetric similarity

$$sim(m2, m1) = \frac{good\ matches(m1, m2)}{number\ of\ keypoints\ in\ m2}$$

Where the matches are obviously computed only once for both the indexes.
This creates a table with each entry ranging between 0 (for completely different models) and 1 (perfectly equal). Given the size of the table, it is not shown immediately here below, but rather in the appendix A (in the last pages). As can be seen, there are some models with well above 20% of their key-points matching (and also, as mentioned before, one couple of completely equal models, possibly due to human error), with the worst offenders reaching almost 39%. While these percentages do not translate directly, they give an idea of the underlying problem: if we matched keeping only the best promising match, we would discard a lot of other equally correct matches, potentially leading to classification as one model instead of the other one. Thus, later on in the process, this table will be used to improve the matching logic: for "easy to tell" models, a faster approach will be used, while for "hard to tell" models a more conservative (and thus slower) approach must be used. For this purpose,

in the following, “hard models” will be the ones that have at least one other model with which they scored a “high enough” similarity score, and “easy models” will be the others. An arbitrary (and rather conservative) threshold has been set to 0.1. While higher values might be used to achieve higher efficiency, a conservative threshold was preferred since no explicit time constraints were expressed.

This is by far the heaviest part of the preprocessing, since it takes a lot of time to extract all the key-points, compute features and perform all the $\frac{N\,x\,(N-1)}{2}$ matches, but we are assuming to be able to perform these operations once and for all “offline”. Moreover, given the previous (qualitative) definition of “easy” and “hard” models, we don’t actually need to compute the full table: for a given model it is enough to find one other similar model to label it “hard”, no need to test any further after that.

6. Step A – Multiple Product Detection

As anticipated before, this step's goal is to deploy local invariant features to recognize different products in the same scene, under the hypothesis that one instance at most is present for each product. Once local invariant features are extracted, this is quite straightforwardly solved by using OpenCV built-in matching functions. Most noticeably, there are two ways to perform matches according to OpenCV documentation: between a single model and a single scene or between one model and multiple scenes. Clearly, there is no implicit knowledge within the matching functions about what a model is nor what a scene is, so the second approach can be easily used to match between one scene and multiple models. However, in this case, a little attention must be paid. As a general rule, in the project, good matches have been selected according to the *second nearest neighbor distance ratio* test, as suggested by Lowe, in his famous paper [2]. In this case, given the presence of very similar models, while matching against multiple models at once, it is highly probable for the second nearest neighbor to be rather close. Thus, a simple test between those distances will work only on the so-called "easy models", while on the "hard models" it would lead to a loss of (potentially) good matches. While this loss is by no means significant if the overall number of matches is high enough (i.e. if the scene has high enough resolution), it becomes really impactful as the scene resolution lowers. It is also important to notice that now we are working under the assumption that each model figures at most *once* in a given scene. This makes it easier to tell apart similar models if they both appear in the same scene, since their matches will not hide each other, even when matching multiple models at once.

For instance, consider the following result obtained by matching multiple models at once (ignoring similarity):

Figure 5 Scene e3 after matching multiple models at once

In this case, even if the central model and the right-most model are very similar (and actually, for several reasons, are the hardest couple to tell apart), even matching several models at once does not lead to a loss of classification accuracy, since matches won't hide each other. On the other hand, if only one of those two was in the scene, matches would have hidden one the other, as in this example:

Figure 6 Scene e1 after classifying with multiple models at once

As we can observe, the (now) right-most model (*model 11*) has been erroneously classified as *model 1*. This happens for several reasons. First of all, it is important to notice that there is an extra element in the model as it is in the scene compared to as it was provided (the extra advertisement on the bottom-right corner), plus the distinctive writing ("cioccolato e nocciole") being moved on the opposite side (to accommodate room for the advertisement), coincidentally ending up exactly in the same place where *model 1* has its own writing. All of these things, combined with the fact that *model 11* came at one of the lowest resolution (while *model 1* at one of the highest) make the matches of model *1* overlap with many of those of model *11*. At this point, some algorithms involving colors were taken into consideration, however color was deemed too unreliable, and subject to occlusions, variations of the models and variations of lighting. Instead, it has been decided to use the similarity information to refine the above discussed method and to split the problem in two:

the detection of “easy models” (by matching multiple models at once) and the detection of “hard models” by matching one model at a time.

In particular, once all the matches are collected for each “hard” model, a way to solve conflicts has been devised. First of all, only the models with a reasonably high number of matches are kept. After that, a position is computed for each of those models and for the models *close* to each other (i.e. for those whose distance is below a threshold) only the one with the highest number of matches is kept.

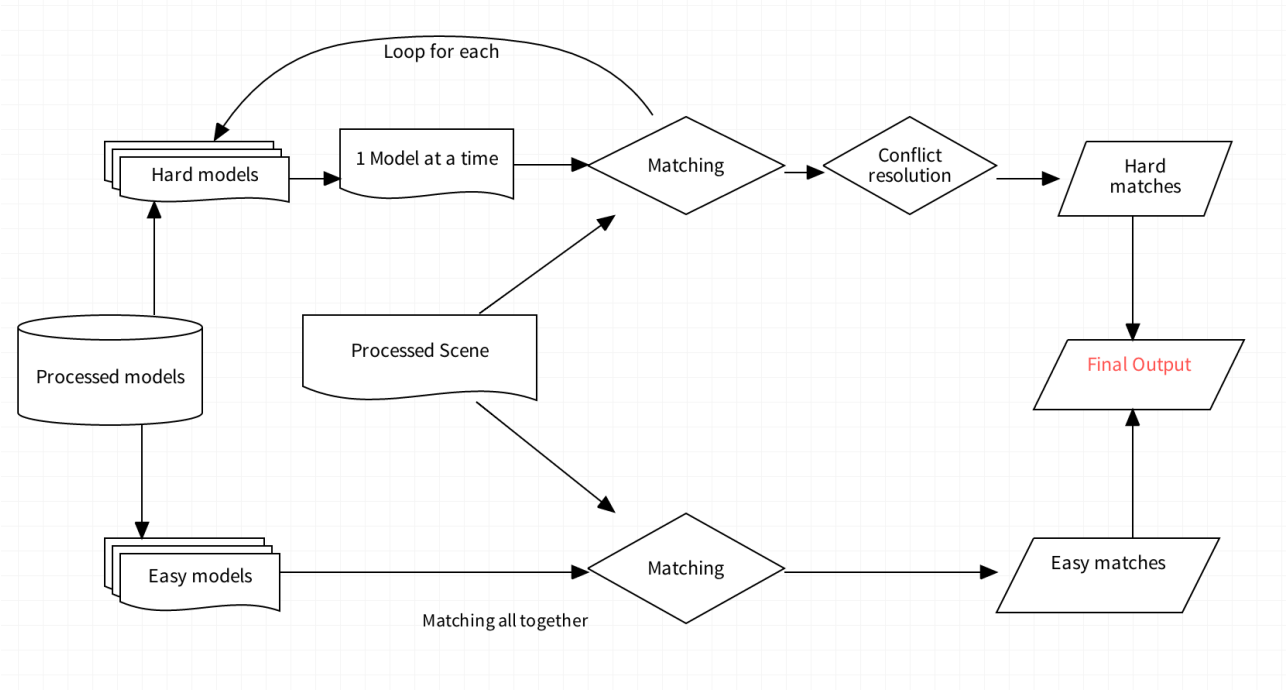
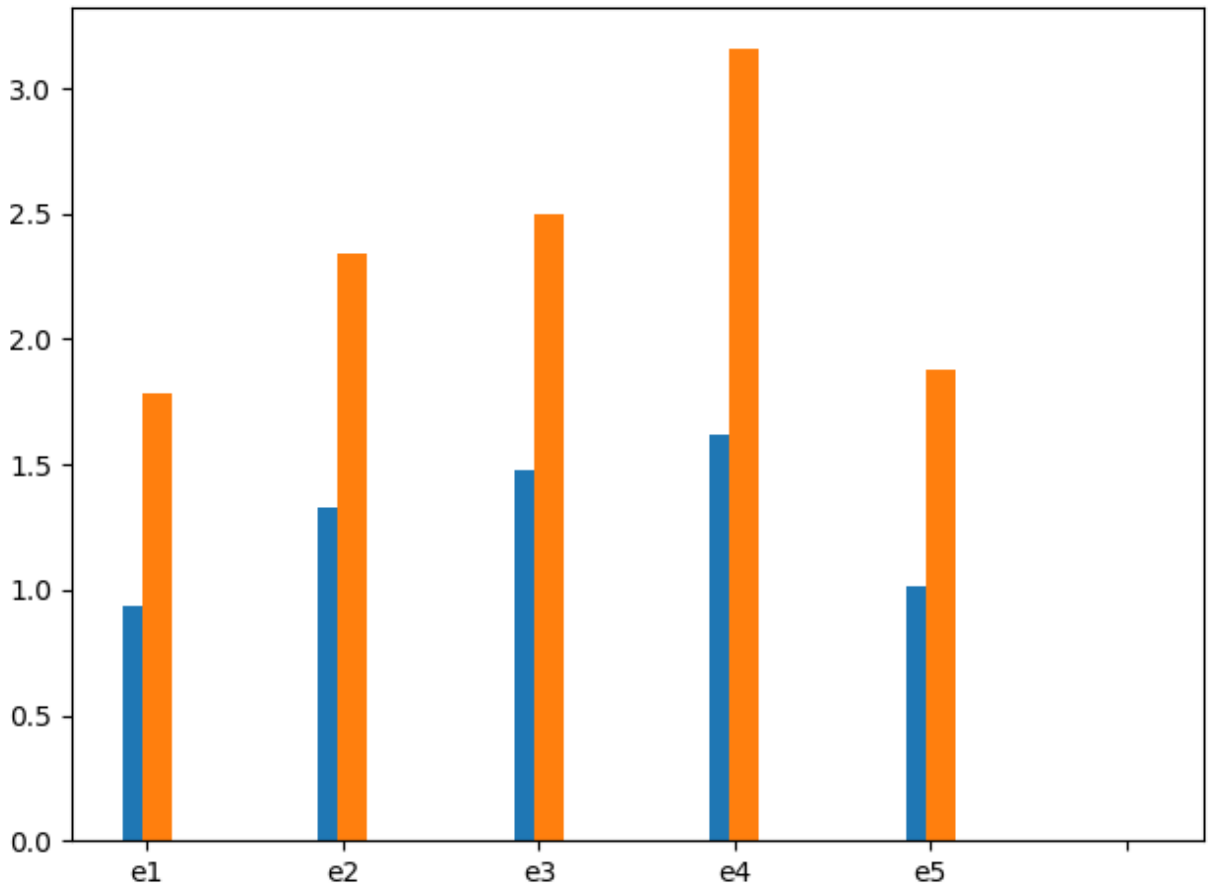


Figure 7 The overall combined approach

This approach to resolve conflict is somewhat similar to (and actually inspired by) the generalized Hough transform. In facts, each match “votes” for the presence of its corresponding model and only the most voted models are kept. Also, models without enough votes are discarded.
As we can see, this time there’s no trouble telling apart models, no matter the scene.

Figure 8 Scene e1 properly classified

In terms of performance, matching multiple models at once is several times faster than matching one at a time. Clearly there are less function calls and context switches, but the main point is that matching multiple models returns less matches (for the reasons discussed before, some "otherwise good" matches shadow each other) and this lifts a lot of weight from all the following computation. In facts, if only the heavy "1-at-a-time" matching version was used, the execution time would average above 2 seconds per scene. Instead, while using a combination of the two approaches, the average computation time drops by half, as shown in the graph below, while still giving perfect results. The blue values are the one obtained matching easy models all at once and then matching hard models one by one solving conflicts while the orange one only applies the "one by one" approach for all the models.



Graph 2 Comparison of 1-by-1 approach and the compound approach (time expressed in seconds)

The test was limited to only these five scenes because this approach does not work "as is" with multiple instances of an object (which is the case of the other scenes), for which a complete version of the generalized Hough transform will be used. Results for the "fast-only" (matching all the models at once) version are not showed since it does not always produce correct results, while the other two do.

Notice that performances are heavily affected by the definition of "hard models" (i.e. by the choice of the similarity threshold: the higher the threshold, the less hard models there will be, thus leading to improvement in computation time. On the other hand, too high of a threshold may lead to a loss of accuracy.

In appendix B, all the final experimental results (i.e. the classified scenes) are reported in detail.

7. Step B – Multiple Instances Detection

Now we must lift the single instance hypothesis. This carries some consequences: now matches can also be diluted across the various instances. If we matched from a model to the scene, always using the second nearest neighbor rule as before, we would have every instance of the model compete against each other. Luckily enough, this can be solved easily enough by "inverting" the direction of the match (i.e. matching from the scene to the model).

As the problem text hinted, the generalized Hough transform has been deployed to handle the detection of multiple instances. More precisely, for each model image an "Hough model" has been extracted during the preprocessing phase. Each Hough model consists of a look-up table (i.e. an array) which associates to each key-point a vector (in the geometrical meaning) pointing towards the barycenter of all the features. In other words, after extracting all the key-points and features, the barycenter is computed as the average position of all the key-points:

$$barycenter = \frac{\sum_{i=1}^{N} keypoint_i}{N}$$

And after that, the Hough model for the i-th key-point as:

$$\boldsymbol{hough}_i = barycenter - keypoint_i$$

Notice that, conventionally, the difference of two points denotes a vector, as the bold font is trying to highlight.

At run-time, features will be extracted from the scene and matched against models. For each matching key-point, an estimate of the supposed barycenter will be computed, by adding the matching vector in the previously built Hough model, after a proper scale and rotation correction. If enough "evidence" (i.e. enough matches hinting to the same barycenter) is collected, we will say that the object has been found.

The general idea is basically the same explained during the first step (step A, Figure 7): models will be matched either all at once or one by one, according to their similarity values and after that, conflicts will be solved.

Compared to what has already been discussed, there's an added problem: due to a lot of factors such as quantization errors, noise and slight variations of models inside the scene, not all the votes agree on one point. In facts, even with only one instance in the scene, only a

small fraction of the matches of any given model agrees on one point, the others are spread in a seemingly random area around. So, there are two possible solutions: either to straight up ignore anything except local maxima or to perform some clustering, in order to somehow collapse "near enough" votes. If we ignore everything except local maxima, we can still get somewhat precise results only if there are a lot of matches to begin with. If the scene resolution is low and / or the scene contains a lot of models, there are not so many good matches available for each model. In this scenario, ignoring everything except local maxima is a big waste of (potentially) useful information. For this reason, it has been decided to pursue the clustering way. The general scheme for the solution proposed, thus, is the following.

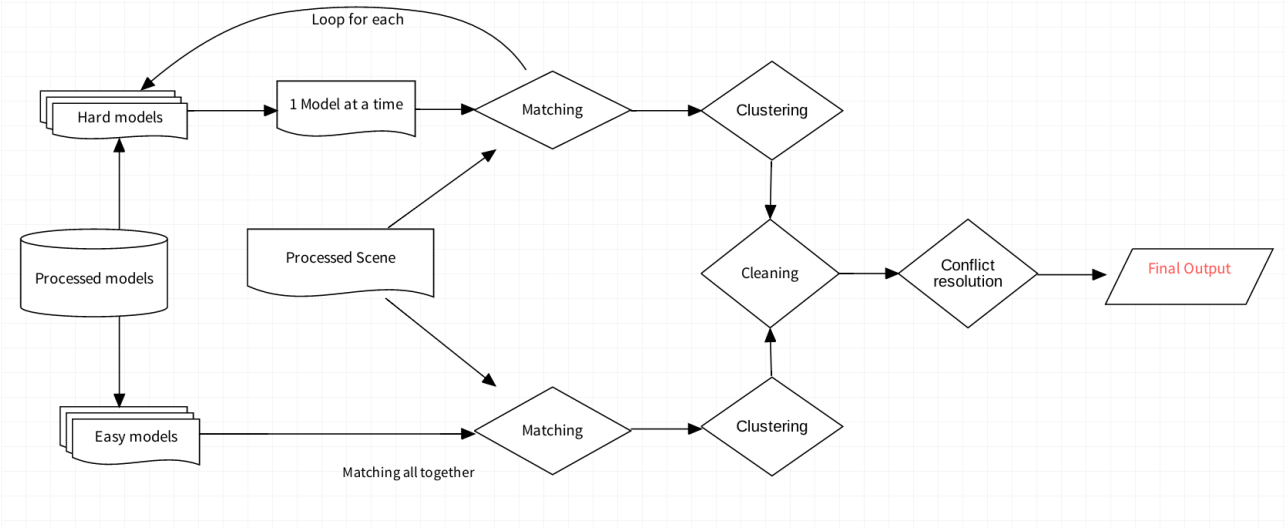


Figure 9 General scheme revised

7.1 Step B – Clustering

Several different ideas where tried regarding how votes should be clustered.

The very first idea tried was to store votes in a symbolic image and leverage on a connected components detection algorithm to fetch "blobs" (i.e. clusters) of connected "pixels" (i.e. votes). However, since those techniques work on binary images, another scan of the whole image would have been necessary after that to locate the center of each "blob" and their actual "area" (i.e. the sum of the clustered votes). Moreover, there is no actual guarantee that the votes would form a connected blob, thus requiring additional smoothing. In facts, by plotting the distribution of the votes, we get results similar to the ones shown in figure here below. As can be seen, matches tend to form a seemingly random cloud around an ideal center, but there is no guarantee of continuity.

Figure 10 Vote distribution for scene e3 and model 11

All in all, that approach was deemed redundant and for this reason another solution was developed. The proposed approach follows two simple rules, whenever a new vote is cast:

1. if there is a previous one within a *small* distance, join them together.
2. Otherwise, store it as is.

In this context, a quantity directly computed from the estimated scene scale (see paragraph 5.3 for more details) is used as a measure of "small distance". In order to keep time performances in check, in the actual implementation we *don't* actually compute N distances each time a vote is cast, since that would be a huge waste of time (see section 9 – Implementation details for more information about this).
When two votes are joint together, they create a new "vote" representing them both with the following properties:

- It has a "value" equal to the sum of the two original votes' values (a freshly added vote has value 1)
- It has a final position equal to a weighted average of the original positions (with the values being the weights).

One of the major limitations of this approach is that it suffers if there is a lot of noise. If there are a lot of bad votes being cast, they will corrupt the results. However, this effect can be kept in check by keeping small the maximum distance to trigger the join. Notice that this

approach is only a slight generalization of the other alternative to clustering (the "ignore everything except local maxima" approach). In facts, if we use 0 as maximum distance to trigger the join, the two approaches are the same (only perfect matches will pile up and thus only local maxima will emerge). More precisely, due to the sensibility to noise, it has been found best to first cast votes and join them only within a very small window (e.g. 10x10 pixels) and only after that to join votes on a larger scale. This is equivalent to initially account only for very small quantization errors (since every computation is done with floating point coordinates, which of course need to be translated into pixel coordinates) and only at a second time, after all votes have been cast and some foundation for stability has been built, to account for major variations.

After all of these operations are performed, we have a list of aggregated votes for each model. Within those lists, there are still a lot of spurious matches and (most likely) also erroneous classifications. For this reason, we still need to perform a *cleaning* step and a *conflict resolution* step (in a very similar way to what we discussed in section 6 – step A).

7.2 Step B – Cleaning

In step A, this was done rather easily by simply setting a fixed threshold for the minimum number of votes needed. In this step, an effort has been made to be a little bit more general and robust to variation. Thus, another two-phase approach has been devised, consisting first of a "*relative filtering*" and after that an "*absolute filtering*".

Depending on the particular scene settings, there will be some models that will be easier to detect compared to others: there will be unlucky models in shady places, or with a lot of similar models present and so on. So, there *will* be a lot of variation between the number of votes that each kind of model can pile up, making it so that finding a single threshold capable of cleaning up everything is impossible. However, for any given model, there will be a lot less variation: the different instances of the same model in a given scene will most likely all get a similar number of votes. Hence the idea behind the so-called "*relative filtering*": filter out every cluster with total number of votes below a fixed (ideally small) percentage of the highest number of votes for that model. Everything below that threshold will most likely be noise, spurious matches, maybe due to some distractors in the scene that for some reason looked similar to the model.

The highest number has been chosen instead of the mean in order to be more robust to noise. The cluster with the highest number of votes for a model is the less likely to be an

error, so it is the best-looking candidate for a reference. The average, instead, especially with a low number of instances of the same model in the scene, is highly affected by noise and spurious matches, making it less than ideal.
A threshold has been set to 40%. While lower values may be used to be more "forgiving" and account for more variation between instances in the same scene, this value performed reasonably well.

Relative filtering alone is not enough. In facts, if a model is totally not in the scene, it may still get some spurious matches. If all the matches found for a model are noise, relative filtering will not be able to filter them out since, at the very least, the best cluster is always kept.
So, some overall reasoning is still needed. As mentioned before, finding a global good threshold is quite not possible and it gets increasingly hard as the number of models in the scene grows. However, it is possible to carefully clean out *very* spurious results (i.e. clusters with only 1-2 votes) by setting an extremely low global threshold. In facts, since at the end of the day we need to compute a bounding box, we could as well filter out anything below 4 votes, since those clusters will not make it possible to estimate a homography, thus making it impossible to derive the bounding box. This is the idea behind the so-called *absolute filtering*. However, instead of simply filtering out everything below 4 votes, it has been decided to compute a threshold from the global average number of votes. This allows to filter clearly bad matches (e.g. clusters with 1-2 votes) while still (potentially) keeping those even below 4 as they may still carry some useful information, especially in the final challenge, where for every model there is a really low number of matches to work with. For this reason, a threshold has been set as:

$$global\ threshold = 1 + \lfloor 0.1 * (global\ average\ number\ of\ votes) \rfloor$$

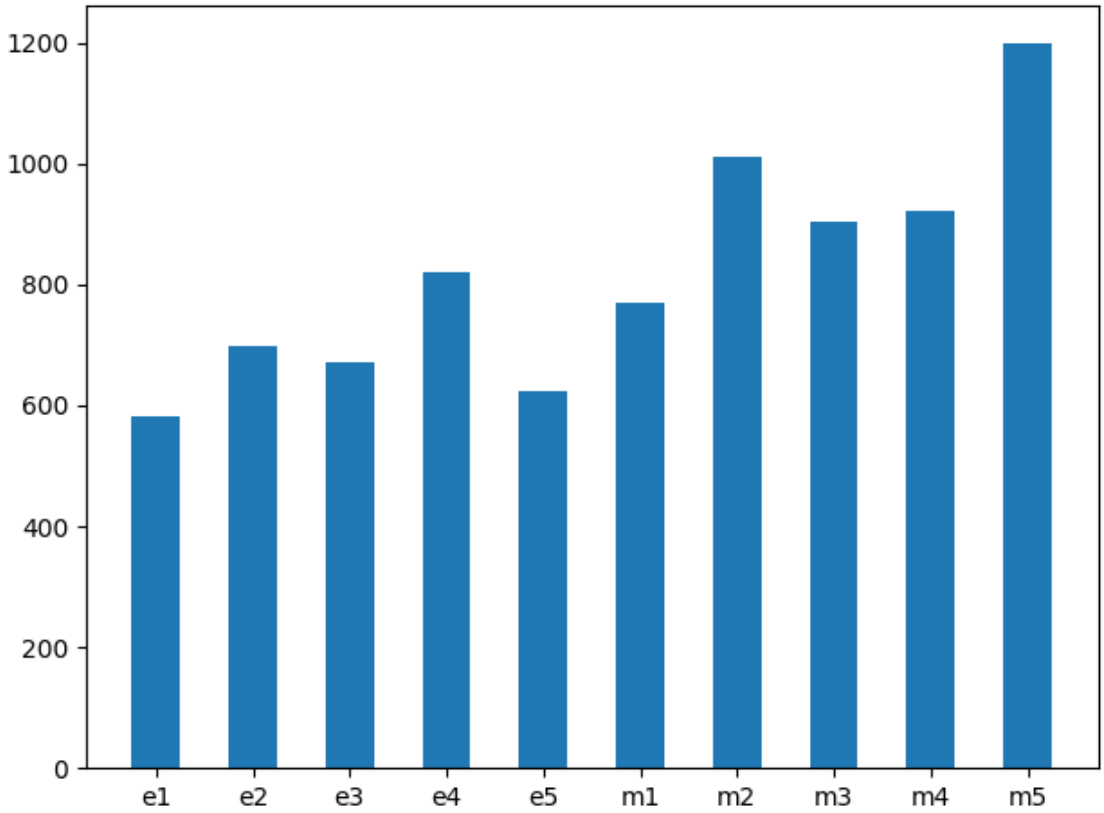
Where the constant "plus 1" makes it so that surely everything with only 1 vote gets canceled out, even when the global average is below 10 votes.

7.3 Step B – Conflict resolution

Conflict resolution is handled in a very similar way to what was discussed during step A. Since we're treating a shelf, where occlusions between models, if present at all, are very marginal, if two clusters belonging to two different models are too close to each other, probably one of them is wrong. So, only the one with the greatest number of votes is kept. Again, a quantity computed from the estimated scene scale has been used as a threshold for the distance between clusters.

7.4 Step B – Time results

The approach proposed properly classifies every model in every given test scene (*except* for the ones of the final challenge). The graph below shows the execution time (in milliseconds) needed to perform the classification task for the various test scenes. Notice that the actual total run time is a little bit higher since it also involves loading and processing the scene.



Graph 3 Time results of GHT classification on various scenes

Also notice that these results are lower than the ones obtained in step A (on the same scenes e1 to e5), mostly because of many little optimizations deployed in this step that simply were not taken into account in step A. More details on optimization are provided in section 9.

8. Step C – Whole Shelf Challenge

This is by far the most difficult task. In facts, not only there are multiple instances of each model in the scenes but also there is a large number of models overall, making each one appearing at a rather low scale.

As it was anticipated before, an effort has been made in order to try and generalize and abstract as much as possible from the scene scale, so that the approach discussed in section 7 (the generalized Hough transform) would still work without any further tuning.

In facts, applying directly the approach discussed before produces some results, which however are very far from perfect. More precisely, while a certain number of models is actually properly detected, there are still several misclassified instances (or not classified at all) and several poor bounding boxes.

Figure 11 scene h1

Figure 12 Scene h2

As can be seen, there are indeed some good results but also a lot of failures. This is due to the fact that at this resolution there is only a small number of good matches, thus making it really hard to properly handle. Also, this issue cannot be easily solved by, let's say, using a more forgiving threshold for the ratio between nearest neighbors' distances. While that solution does indeed increase the number of matches, it also lowers their quality, not leading all in all to major improvements: some models now wrongly classified would coincidentally be properly classified while some now properly classified would be wrongly classified. In facts, the results displayed here use a threshold of 0.9 in order to keep a lot of matches, as opposed to the 0.65 that was enough to properly classify everything until step B.

So, clearly there is faulty logic within the reasoning proposed, or, even if the logic was correct, the overall idea is not sufficient.

Here below, the remaining results for this step are reported.

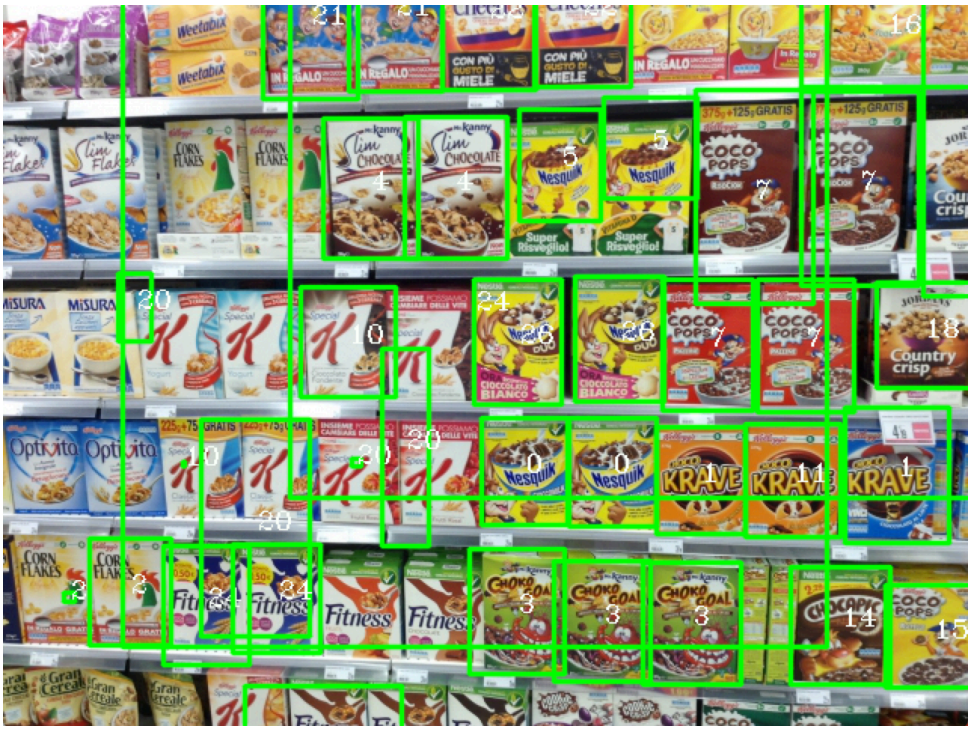


Figure 13 Scene h3

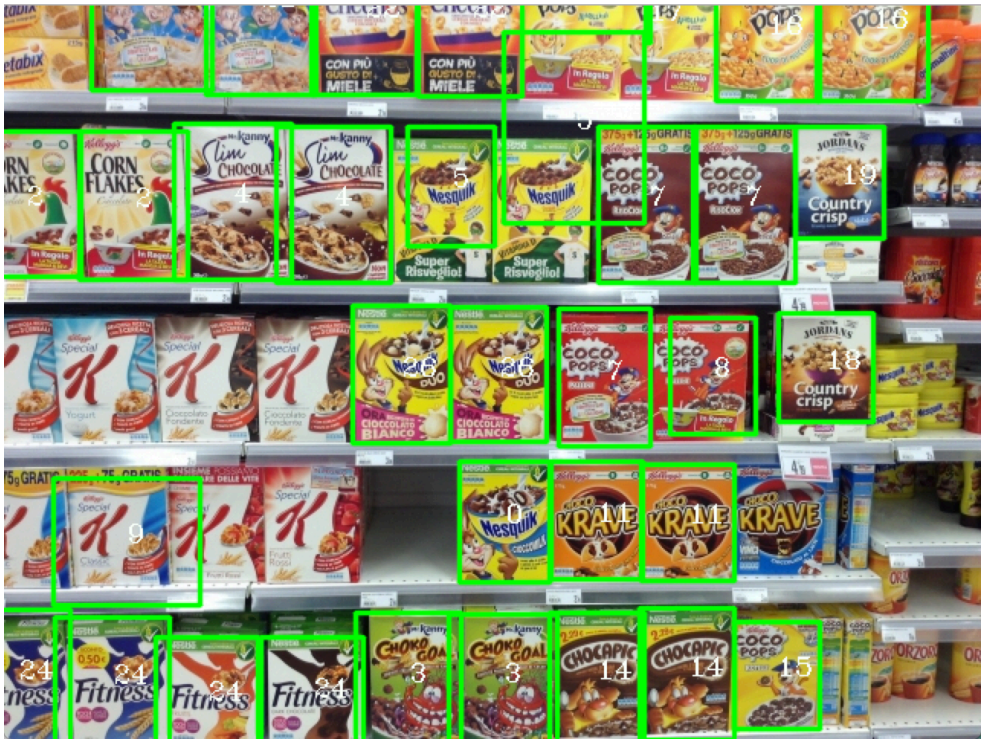


Figure 14 Scene h4

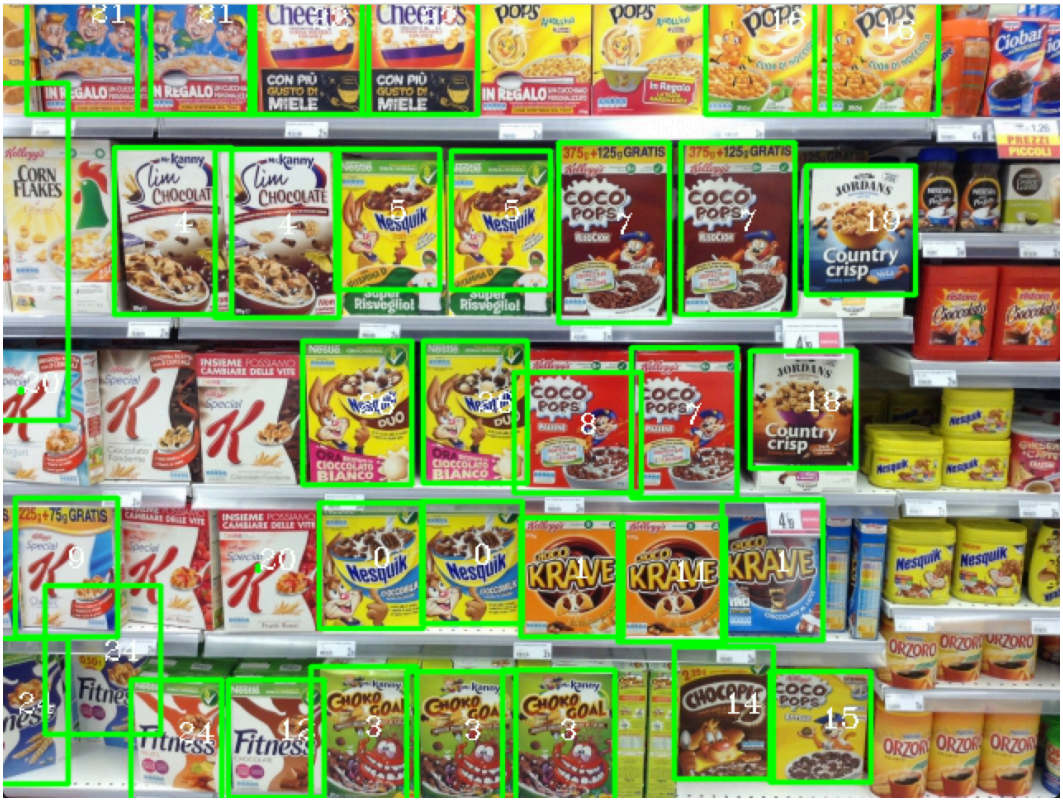


Figure 15 Scene h5

As bad as it may sound, no better results than the ones showed here were achieved.
In facts, probably due to my very limited knowledge, no other feasible solutions were found other than using the generalized Hough transform.
Most probably, there is something cleverer, some more intelligent analysis that can be carried out on the scenes to preprocess them in a smarter way and eliminate much more problems than what is done here. Some ideas that were thought of, but for various reason discarded, are briefly summed up in section 10 – improvements and developments.

9. Implementation details and overall optimizations

Throughout the development of the entirety of the project, an effort has been made to keep everything readable and performant. For this purpose, some convenient data structures were created.

9.1 General optimizations

During all the development, every single functionality built relied on pointers to particular data structures (more details in the following paragraphs).
Pointers were chosen over regular values for two fundamental reasons. First of all, C (and C++) pass arguments to functions *by copy*. This means that every time we invoke a function on a normal structure, such as a `cv::Mat` or a `std::vector`, it gets fully copied. The obvious solution is, indeed, to adopt pointers since copying a pointer is several times faster than copying the full pointed data structure. The other main reason pointers were chosen is because we needed changes and operations to be permanent on the models: it would have been way too slow to create a new processed image and return for each function call. Pointers, clearly, allow to modify the passed value "in place".

Moreover, every "core" function (i.e. functions that for some reason are called several times during the execution) has been marked as `inline`. This qualifier tells the compiler to actually resolve the function call at compile-time (rather than at run-time) and to *inline* it, in a similar fashion to how the preprocessor handles macros. The only downsides of this practice are the increase in compile time and in size of the actual executable produced, but in this case, both those side effects are in check as compilation time (for all it matters) is around 6 seconds and the executable size around 1.1 MB.

9.2 Implementation – RichImage abstraction

As the number of models and data related to models grew (key-points, features, Hough models and so on) the various structures started losing cohesion. In facts, there were lots of vectors, vectors of vectors and so on, only glued together by indexes. In order to keep everything coherent, the class *RichImage* was created.
At its core, it wraps a normal `cv::Mat` but also keeps track of the image extracted features, key-points, path to the image and offers methods to perform basic tasks, such as computing the Hough model, estimating the scale or improving the image.

As anticipated, all the other structures and functions are built on top of *pointers* to RichImages.

9.3 Implementation – Blob abstraction

In spite of the name, the *Blob* structure is what has been addressed to as a "cluster" in the previous discussion. This abstraction was deemed necessary because when we deploy the generalized Hough transform and handle all the votes, we must also keep track of the matches corresponding to those votes in order to later retrieve the estimated bounding box, position and size of the detected model. In the very first iterations of the project, this was not taken into account and, instead, an effort was made to try and reconstruct that information from the estimated average scale of the votes. However, that was simply dumb and soon discarded, in favor of the Blob abstraction.

9.4 Implementation – Developing the GHT

In order to ease the development of the generalized Hough transform, a particular structure has been created: the so-called `VotingMatrix`. This data structure handles the vote casting, cleaning and conflict resolution while trying to keep everything as fast as possible.

As a first approximation, it encapsulates a normal `cv::Mat` which stores the actual votes, and whenever a new vote is cast, it automatically finds any other vote within the specified collapsing distance and, if necessary, it collapse them. However, in order not to compute a huge quantity of distances every time, the votes are handled slightly differently.

The actual `Blob` structures (incapsulating votes) are stored in vectors (one `std::vector` for each model), so that, for later uses, one could directly access only the meaningful positions instead of performing several scans of a symbolic image. In order to not compute distances, for each model is also stored a *mask*. These masks are actual `cv::Mat` in which each "pixel" is either -1 or the index of the nearest `Blob` in the corresponding model's vector. In this way, whenever a vote is cast, we can simply access the model's mask and see if there was already a `Blob` within the specified distance. If there was, simply collapse them, otherwise the new vote is encapsulated in a new `Blob` and added to the corresponding vector. Finally, the mask is updated by setting all "pixels" within a patch of the specified size around the vote to point to the freshly added `Blob.` This saves a lot of actual computation time and works decently well only under the assumption that, at this point, the collapsing distance is rather small (e.g. 10 pixels). Too high values proved to be "too greedy" in case of erroneous votes, leading to several spurious clusters. In facts, there is no guarantee, when

collapsing immediately at full distance, that if two votes are "close" one of them is good. If the original one is actually bad, the other gets collapsed onto the bad one (reinforcing the error) while also potentially punishing potential subsequent good votes. Instead, when collapsing within a small and limited window, errors will still reinforce each other, but they will not interfere with the good votes that will eventually emerge and "correct" the errors by either absorbing them in the following full-distance collapse phase or in the cleaning phase.

The cleaning phase, as mentioned in section 7.2, is divided in two phases: relative and absolute filtering. For practical usage, in spite of the diagram showed in Figure 9, it was found best to perform the conflict resolution between the two cleaning phases (i.e. after the relative filtering but before the absolute) as shown here below. As anticipated before, the absolute filtering suffers from errors, especially with a low number of instances, so resolving as many of them as possible beforehand is not a bad idea. Plus, if we performed it before the conflict resolution, we might still have needed to perform it afterwards to catch some Blobs that could have just barely survived the first sweep (thanks to other errors). During these phases (cleaning and conflict resolution) distances are actually computed. While this may sound bad (and it could actually be improved), it is important to notice that at this point there is a rather limited number of Blobs (i.e. clusters) and the number only decreases while moving on. So, all in all, as seen in Graph 3, the impact on performances is rather low.

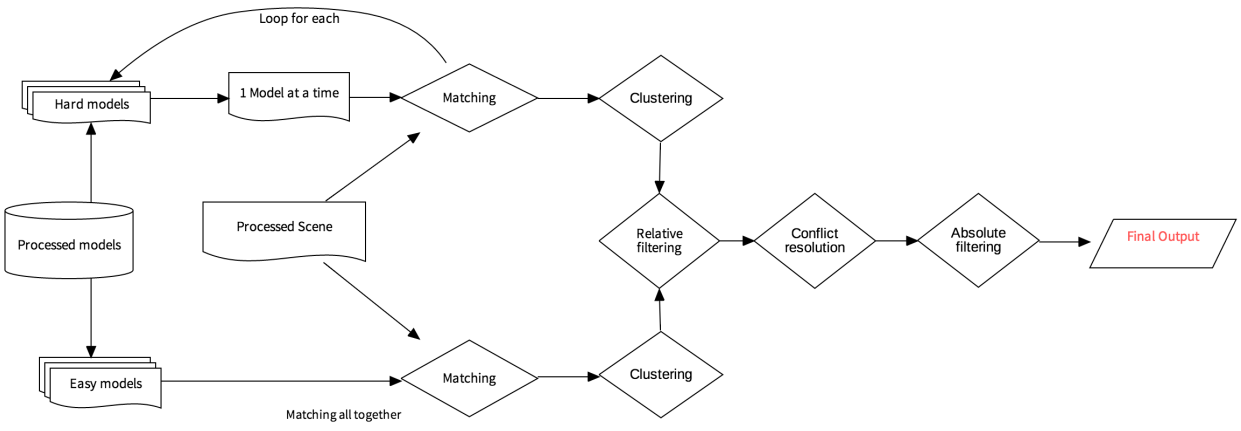


Figure 16 Actual implemented scheme for step B

10. Possible improvements and future developments

Many other ideas were taken into account during the development, but for various reasons they were all discarded. Nonetheless, they could be as well the starting point for some future improvements.

10.1 Convolutional Neural Networks

When confronted with the task of object detection in noisy environments, neural networks are the state of the art. In this case, unfortunately, they were immediately excluded due to the extremely low number of samples to work with. With such a limited number of samples, it would be extremely hard to refine a model capable of enough generalization (i.e. not falling into overfitting) with high enough performances. Plus, the example provided are not "labeled", which means to either turn to unsupervised learning or to manually label the samples. A very interesting idea, but ultimately not really feasible with this number of samples.

10.2 Color space analysis

While, during the development, working with gray scale images was the most obvious choice, there are also other interesting color spaces that might be worth exploring. The idea behind is that in this particular scenario (handling cereal boxes on a store shelf), color carries a valuable information. However, in the common RGB space, color is unreliable since too subject to variations due to lighting and occlusions. Nonetheless, other color spaces may prove useful by, at least, providing some degree of lighting invariance.

For instance, HSV (hue – saturation – value), and many others, can be efficiently used to operate color-based segmentation, as described in the corresponding OpenCV tutorial [3].

However, I was not able to think about any robust approach to actually accomplish improvements.

10.3 Genetic algorithms

One of the most beautiful natural metaphors deployed in artificial intelligence, this family of algorithms are particularly suited for all those situations in which there is an unknown relation between the input of the problem (i.e. the models) and the output quality (i.e. the precision of the classification). Genetic algorithms are capable of automatically evolving a solution starting only from the inputs, the base operators on them and a measure of the quality of the output. In this case, any solution would be a tree describing the solution algorithm structure: starting from the leaves (the inputs), each node would be one operation (e.g. a filter, a binary morphology operator and so on) which takes the node's children as parameters. The nodes, branches and leaves connection would be the "genome" of the solution. This is a very similar concept to what is done in convolutional neural networks, where we allow the network to learn the proper convolutions. Here, we allow the machine to learn arbitrarily complex algorithms while only providing the "base ingredients" we deem the most fit along with the rules to combine them. As beautiful as it sounds, it also an *extremely* slow task which often requires several days even when accelerating computation on GPUs. Furthermore, it requires a measure of the quality of a solution, much like neural networks require labeled data.
All in all, this was deemed too time consuming and probably not necessary, but totally worth a try in a future.

10.4 Optical Character Recognition

The idea is taken from what us, humans, do when we have to recognize products. We probably do most of the work "at first glance" by recognizing colors or logos, but the absolute proof comes from reading the labels. Unfortunately, I was not able to clean the scene enough to extract only those edges concerning writings.

10.5 Better optimizations

Even without any other ideas, I personally feel like there is probably a lot more that someone with actual knowledge of C++ could improve in the proposed code. Due to my very limited experience with the language, I probably missed some common practice or done some common mistake. All in all, time performances were mildly satisfying so I did not feel the strong need for harsh optimization, but for deployment on an actual production-ready system more work is surely needed.

Appendix A – Similarity table

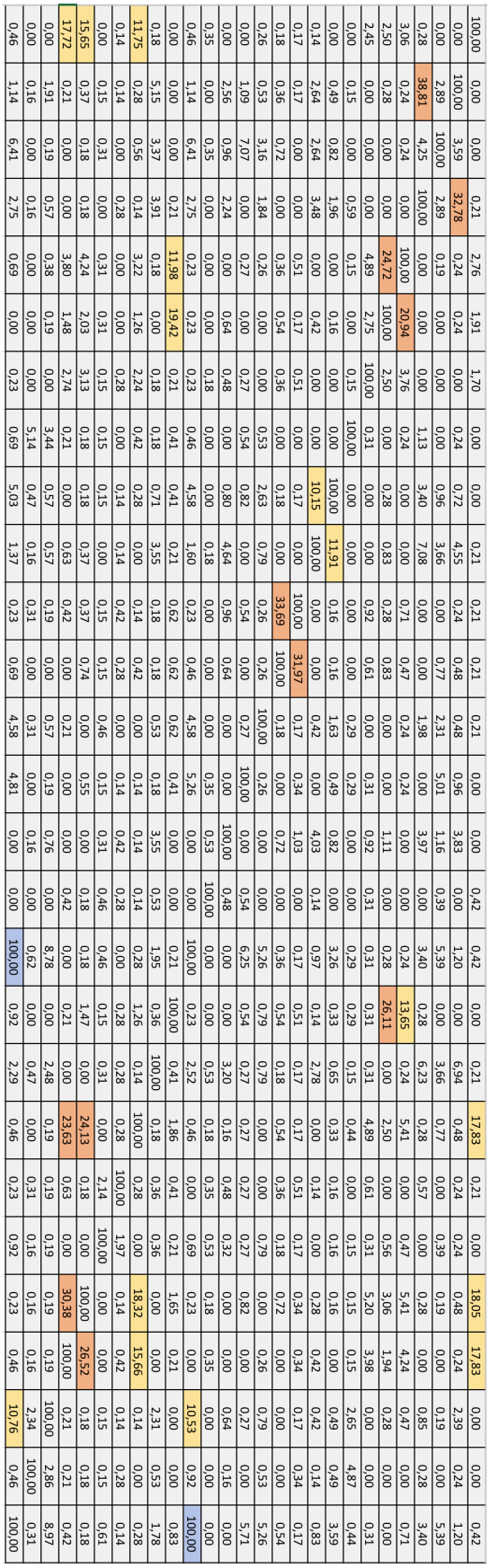


Table 1 Similarity table, scaled by 100 to represent a percentage

Appendix B – Step A full classification outputs

As anticipated, the scenes are all properly classified.

Figure 17 Scene e1

Figure 18 Scene e2

Figure 19 Scene e3

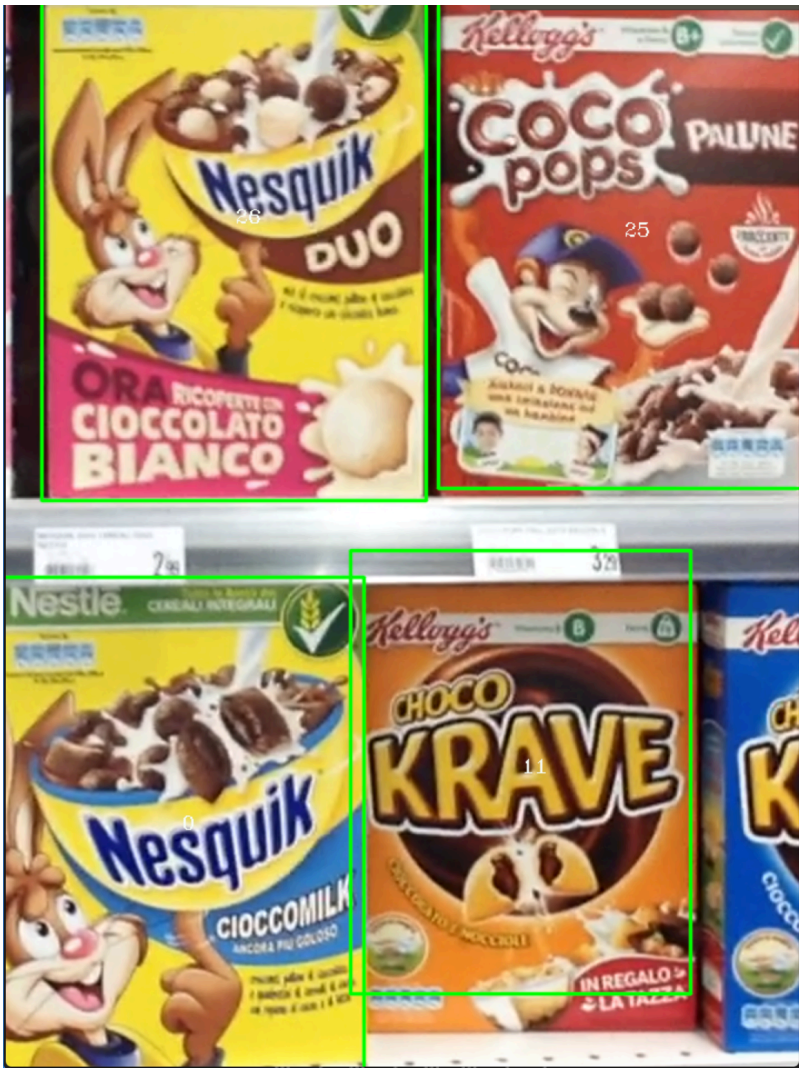


Figure 20 Scene e4

Figure 21 Scene e5

The bounding boxes for *model 1* and *model 11* are slightly wrong, but please notice that their model is actually different from how they figure in the scene (in the model there is not the advertisement), so an higher number of matches would be needed to correct this effect.

Appendix C – Step B full classification outputs

Even in this case, the proposed solution properly classifies every image (including the ones relative to step A, but excluding, as we saw, the ones in the final challenge).

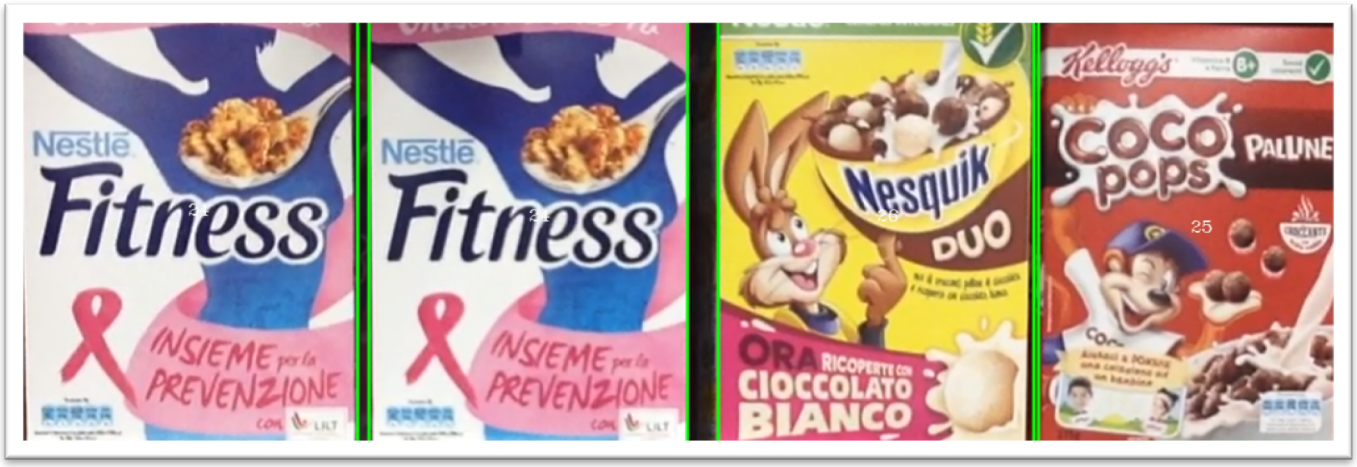


Figure 22 Scene m1

Figure 23 Scene m2

Figure 24 Scene m3

Figure 25 Scene m4

Figure 26 Scene m5

Bibliography and Sitography

[1] *Three things everyone should know to improve object retrieval*, Relja Arandjelovi´c, Andrew Zisserman, 2012

[2] *Distinctive Image Features from Scale-Invariant Keypoints*, David G. Lowe, 2004

[3] *https://www.learnopencv.com/color-spaces-in-opencv-cpp-python/*