# Product recognition on store shelves

Computer Vision 2017/2018

Marco Di Vincenzo

## 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.

## 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.

## 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 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, we will see some time measurements but those are relative to the before-mentioned architecture and may vary a bit on other systems, depending on the hardware.

## 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.

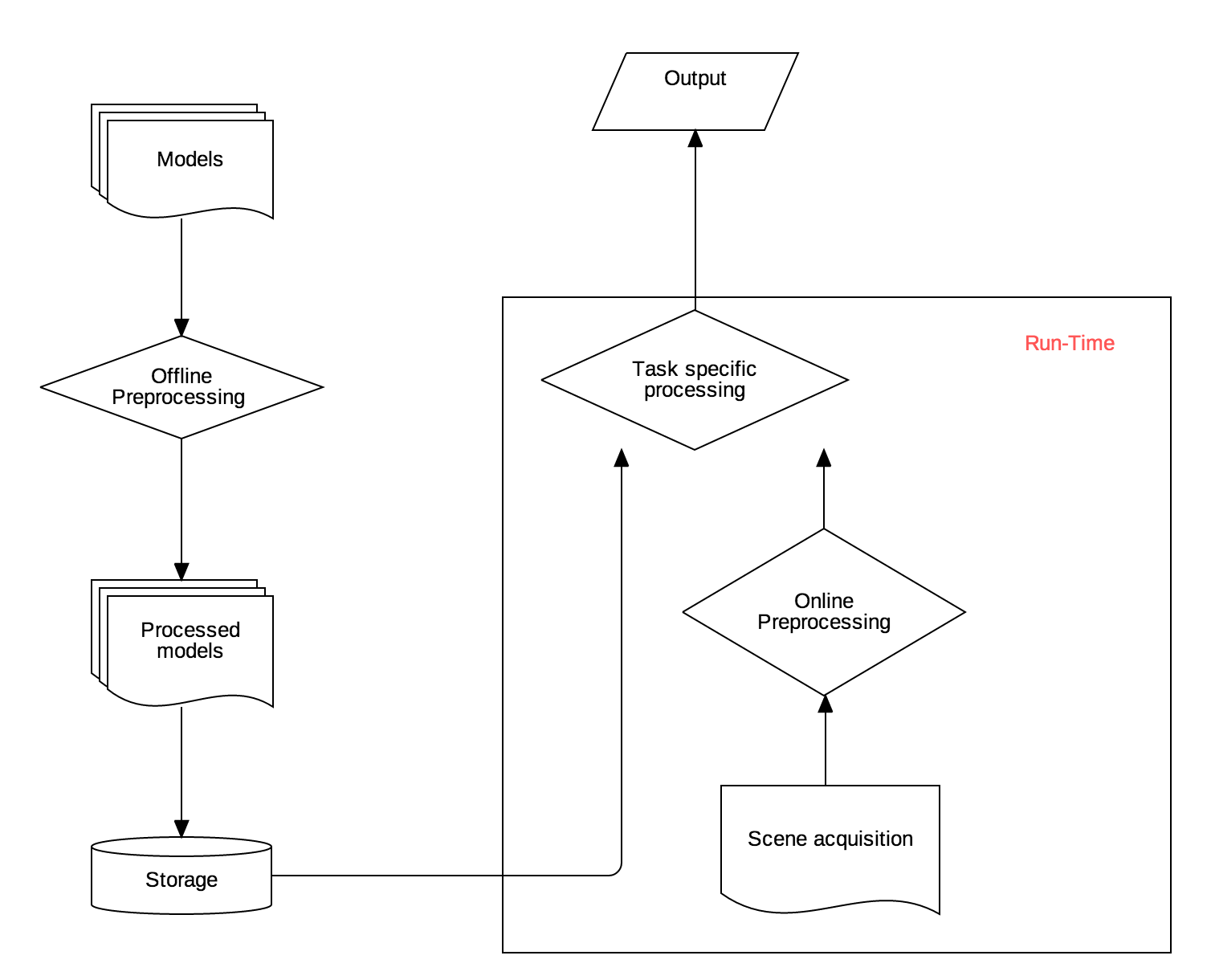


Figure 1 The general Pipeline

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.

## Offline preprocessing

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

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. 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 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.

### 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 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. The size choice was determined by looking at the effects induced on the performances of the classification task, 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 scaled it to match the one of the smallest model and scaled the other dimension (i.e. the *width*) to maintain the original aspect ratio.

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), 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 was deemed a bad idea as also the template matching approach was found incapable of providing better results than using the generalized Hough transform.

### Preprocessing – Image sharpening

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 sharpening, whereas in the final challenge sharpening introduces artifacts.

### 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, 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.

### 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. 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 features 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:

While the symmetric similarity

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 (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 soften the definition of “good match”. Instead of only considering a good match the ones stated by Lowe, the ones with a ratio above the threshold will also be spared if they involve a “confusing” couple, i.e. a couple whose similarity index is above a certain level (further details later, when discussing the various tasks’ logic).

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 matches, but we are assuming to be able to perform these operations once and for all “offline”.

## 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.

## Appendix

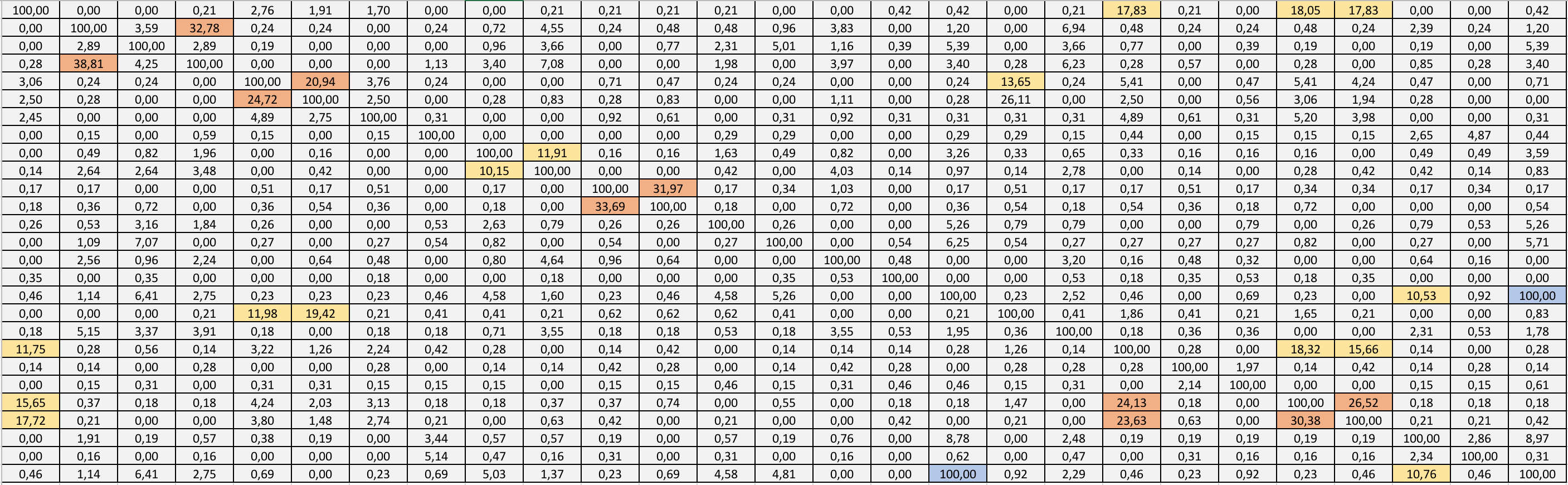


Table 1 Similarity table, scaled by 100 to represent a percentage