Coke bottle defect recognition

# Task description

Given are 141 images, each image containing 3 bottles of coke. The bottles may have defects, that need to be recognized by an algorithm.

These defects may be the following:

1. The bottle is not present
2. The bottle is missing its cap
3. The bottle is missing its label, or it is in the wrong position
4. The level of the fluid is too low
5. The level of the fluid is too high
6. All the reasonable combination of the defects above

Implement an algorithm that can tell the defects of the bottles.

# Approaches

For the classification I used two approaches, which require the preprocessing of the images. The two algorithms are decision tree learning and multilayer perceptron.

The former is a simple algorithm, can work well with small datasets. The latter is a type of artificial neural network, which is a universal approach for classification purposes002E

# Technologies

## Python

## OpenCV

## Tensorflow

## Sklearn

# Image processing/features

Before doing any processing, I split each image into three pieces, each containing only one bottle. This can be done by cropping the images with static pixel addresses, because the bottles are approximately in the same area of the images.

After splitting the images, I divide the data extraction into two parts based on the color of the things to find. The cap and the label have bright colors and the coke is quite dark. This distinction is useful when creating binary images.

## Cap and label

### Features

For these properties, I decided to draw rectangles around the cap and label.

The rectangles’ area and position can tell, if the bottle has defects regarding the cap and label. The position can firstly help determine, which rectangle should be the label, and which should be the cap, secondly a leaning label should have lower center coordinates. Also, an unwanted rectangle will result in a false negative, which is preferred over false positive.

The features are:

* Cap
  + Area of the rectangle
  + x and y coordinates of the center of the rectangle
* Label
  + Area of the rectangle
  + x and y coordinates of the center of the rectangle

### Algorithm

Before trying to find any of the features, distorting the image is a must, because it will remove every text, pixel errors and will lead to a smooth picture.

As a first step I wanted to get rid of all the white areas on the picture, which might influence the thresholding later.

**Erosion** is a great function to delete details of an image and as I only want to roughly remove the bright parts, that may interfere with the future work, there’s no need to **delate** the image right now.

I found, that erosion with a 5x5 (from now on every erosion is done with a kernel this size) kernel, repeating it 7 times gives a good picture for this job. Converting the image into greyscale and thresholding with a relatively high (130) value, “findContours()” will work great.

After finding the bright spot(s) they are painted black by drawing an approximated, filled polygon on the image. The approximation function used is “approxPolyDP()”, which uses Ramer–Douglas–Peucker algorithm. This step is already drawing onto the original image.

The second part of the algorithm also starts with erosion, but to restore the size of the areas I’m searching for, after 10 iterations of erosion 5 dilation is performed.

To be able to work with the image and find the important parts the image is converted to greyscale, then a threshold is performed with a relatively low value (50). Given the previous step, the bright areas are now black, they won’t interfere with the current low threshold.

The binary image is then blurred with a normalized box filter (35x35). Finding the contours now might still result in more, than two shapes. If so, I gave the blurring another try with a smaller kernel.

Now the rectangles may be drawn. I used the “boundingRect()” function, which doesn’t consider the rotation of the contours. I choose this over the “minAreaRect()”, since the rotation never changed in any of the cases and it’s easier to handle a rectangle, whose sides are parallel with the axes.

The resulting rectangles’ areas and center point is calculated, and the following logic will determine the final features:

* If a rectangle’s area is smaller, than a specific value (based on trials it’s 3000), it is considered a false recognition.
* If no rectangles are found, zeros are set to all feature values.
* If only one rectangle is found, based on the center of the shapes, the other is set to zero.
* If two rectangles are found, they are ordered based on their center.
* If there are still more, than two, the first two options are chosen to be the correct ones.

## Fluid level

### Features

This solution is like the previous one in a way, that the fluid will be surrounded by a rectangle. The feature will be now the “y” coordinate of the top-left corner of this box. The main reason why the other properties of the rectangle are not used is, that the label’s state will influence a lot of these properties. Take the area as an example. If the label is missing, the area of this shape will increase, which could result in a “too much fluid”, however the level is right, or the “right amount” even though it’s low.

### Algorithm

Because the color to find the contours for is close to black, there’s no need to blur the texts completely, the label, cap and texts will be much brighter, than the coke.

Applying erosion and dilation with the opposite amount of iterations as previously (5-10 instead of 10-5), thresholding the greyscale image gives a good approximation where the fluid level should be.

After this, I apply a blur with a 25x25 kernel. This seems improve the results.

The result will be:

* If a rectangle’s area is smaller, than 500, it is considered a false recognition.
* If no rectangles are found, the image’s height is returned (the higher the value, the lower the fluid level).
* The first “good” rectangle’s top-left y coordinate is returned.

# Machine learning

## Classes/Labels

|  |  |  |
| --- | --- | --- |
| Value | Class | Samples |
| 0 | OK | 312 |
| 1 | No bottle | 12 |
| 2 | No cap | 18 |
| 3 | Wrong label | 48 |
| 4 | Low fluid level | 13 |
| 5 | High fluid level | 12 |
| 6 | No cap and label | 3 |
| 7 | No cap, low fluid | 1 |
| 8 | No cap, high fluid | 0 |
| 9 | Wrong label, low fluid | 3 |
| 10 | Wrong label, high fluid | 1 |
| 11 | No cap, wrong label, low fluid | 0 |
| 12 | No cap, wrong label, high fluid | 0 |

After classifying all the images, it can be noted, that most of these states have low number of samples, or no sample at all, which means it’s almost impossible to work with them.

One solution would be to merge some of the more complex states with the less complex, more frequent ones. Another, in my opinion better, is to separate the problems into two different models: Fluid recognition and cap and label recognition.

# Cap and label classes:

|  |  |  |
| --- | --- | --- |
| Value | Class | Samples |
| 0 | OK | 337 |
| 1 | No bottle | 12 |
| 2 | No cap | 19 |
| 3 | Wrong label | 52 |
| 4 | No cap and label | 3 |

#### Fluid classes

|  |  |  |
| --- | --- | --- |
| Value | Class | Samples |
| 0 | OK | 341 |
| 1 | Low fluid level | 29 |
| 2 | High fluid level | 53 |

## Learning

As mentioned earlier, the two techniques are Multilayer perceptron and Decision tree learning. With the splitting of the two different problems and with better datasets, I’ve created two similar models for both problems.

The MLP for the cap and label problem has 6 inputs and 5 outputs. The “values” of the classes area transformed into “**one hot**” arrays. This is done by setting the nth element of an array to one and leave the rest 0, n is the value of the class. The array’s size will be ***max(class values) + 1****.* The resulting array will represent the expected outputs states of the neural network.

For the fluid recognition the same can be said, except it has 1 input and 3 outputs.

The decision trees will have the same properties, but they only need the features and classes and don’t need the one hot translation.

### MLP

I want to list the properties for the MLP.

* Neural network model  
  By this property, I mean the number of neurons and hidden layers in the neural network.
* Epochs  
  How many times the same data set is taught to the network
* Learning rate  
  This value tells how big steps the optimizer will perform in order to reach an optimal state. If this value is too high, or low, it may cause the algorithm to never find an optimal state.  
  OR: How much the values are allowed to change in each step.
* Randomize biases  
  Biases are part of a neuron’s calculations.
* Cost function  
  Softmax with cross entropy
* Optimizer  
  Adam and Gradient Decent
* Training/Testing ration  
  Percentage of the original dataset used for training and testing.

# Results

*See Data.xlsx for the results*.