# IAPR: Final project - Chocolate Recognition

**Moodle group ID:** *39*
**Kaggle challenge:** `Classic` **Kaggle team name (exact):** "*LatruiteGauloise*"

**Author 1 (SCIPER):** *Léo Bruneau (341902)*
**Author 2 (SCIPER):** *Louis Pivron (329614)*
**Author 3 (SCIPER):** *Huckleberry Thums (345317)*

**Due date:** 21.05.2025 (11:59 pm)

## Key Submission Guidelines:

- **Before submitting your notebook, rerun it from scratch!** Go to: `Kernel` > `Restart & Run All`
- **Only groups of three will be accepted**, except in exceptional circumstances.

---

## Justification of Design Choices

### background selection

Looking at the test images, it was quite clear that the neutral background images would be easier to deal with compared to the rest of the dataset. So we decided to build first a very specific pipeline that would work well with neutral background images.

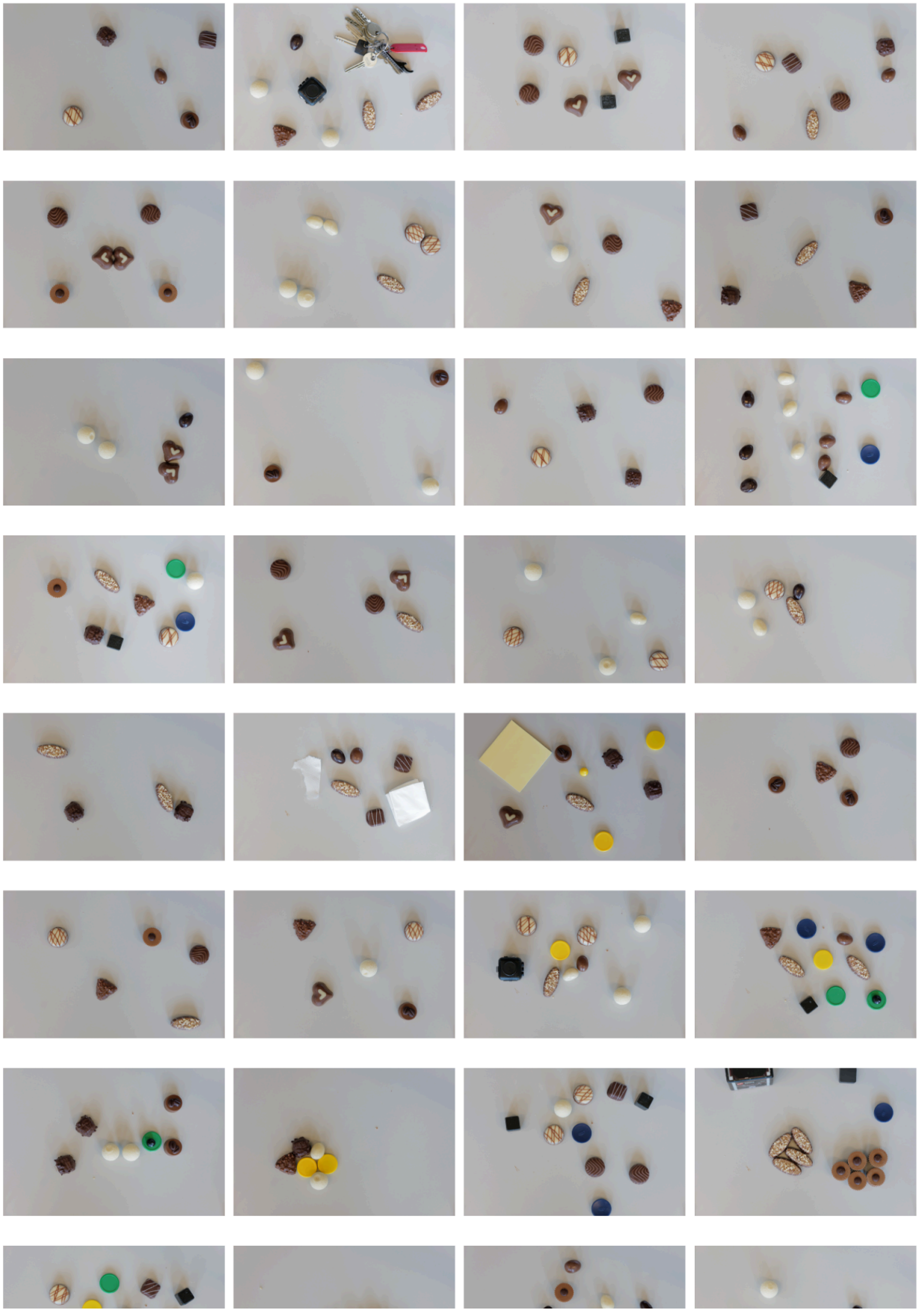
But before being able to process the neutral background images, we needed at way to create clusters of images based on features that would somehow relate to the background contents.

We used an unsupervised method for this purpose since it yielded decent results and the need for a supervised approach was therefore not necessary.

In [26]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(20, 50))
plt.imshow(plt.imread('image_cluster.png'))
plt.axis('off')
plt.show()
```

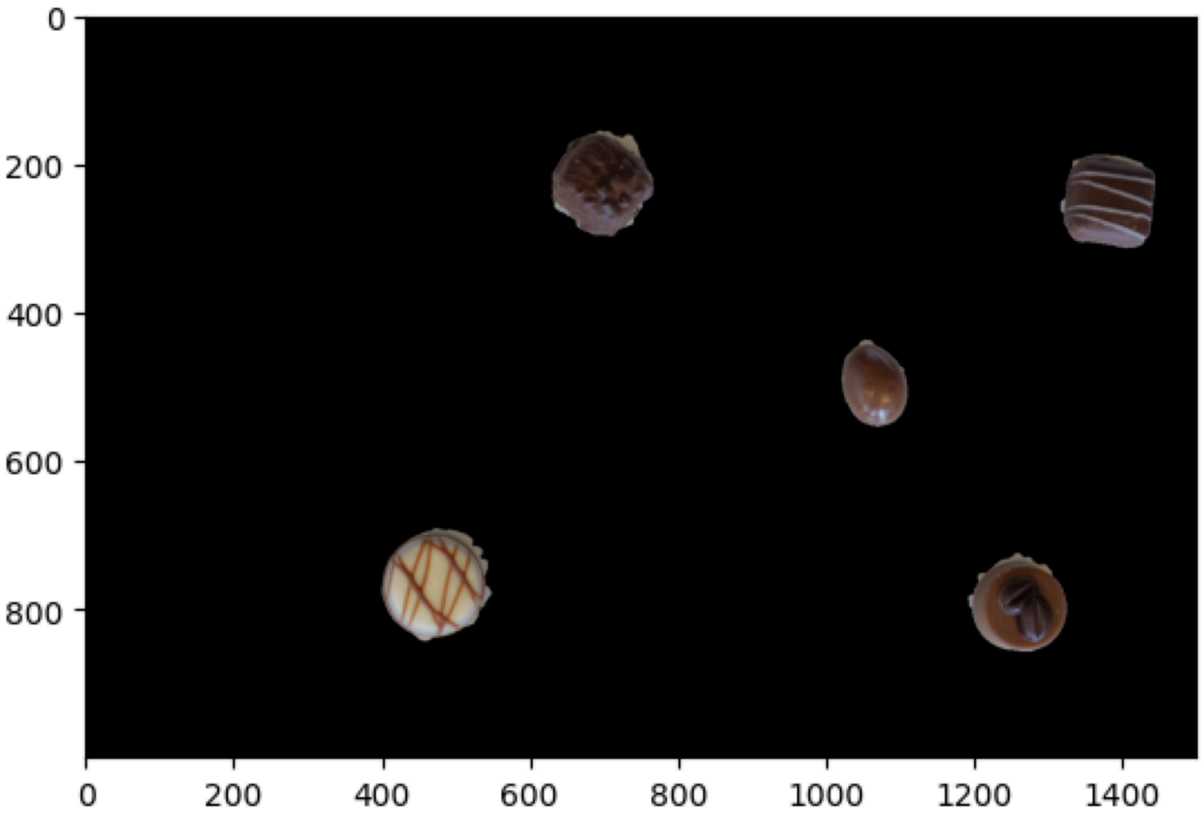
Images from Cluster 1



From the 60 neutral-background images of the test set, 46 were correctly clustered (14 being found in other clusters including complex backgrounds).

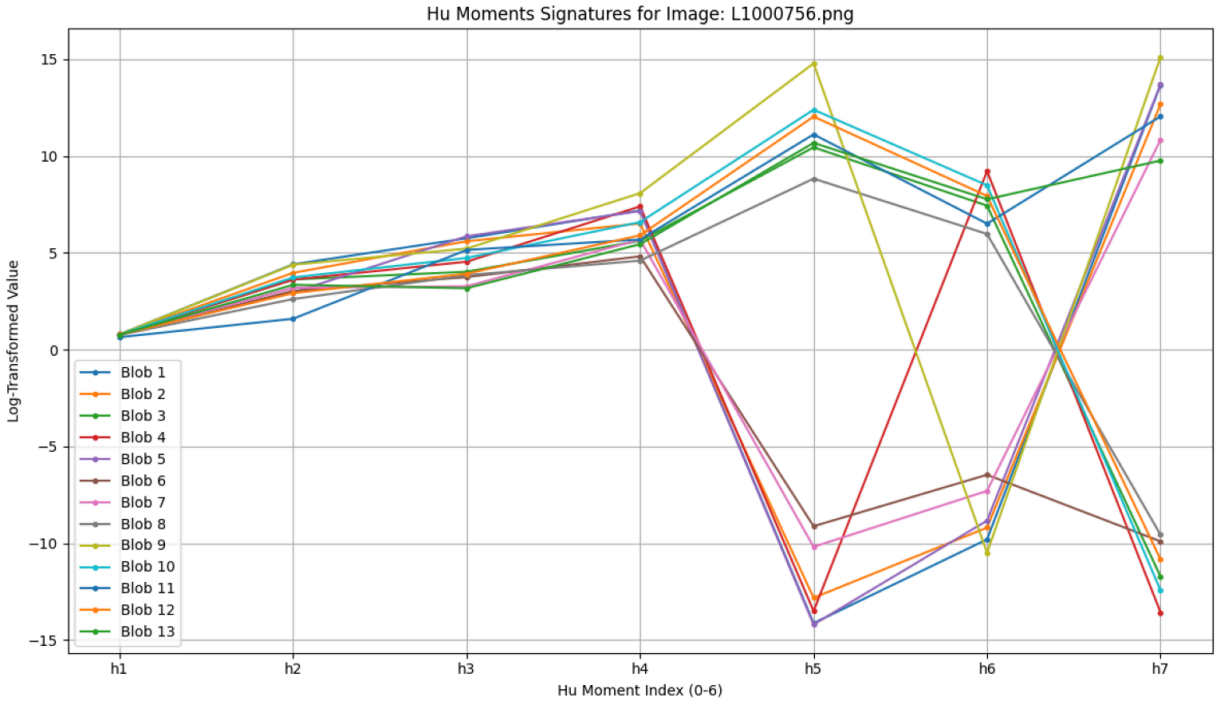
Then we performed object detection on the clustered neutral background images. We obtained segmentation masks that were then used to recover shape features from the segmented chocolates.

In [27]:
```
plt.figure(figsize=(20, 50))
plt.imshow(plt.imread('image_1.png'))
plt.axis('off')
plt.show()
```

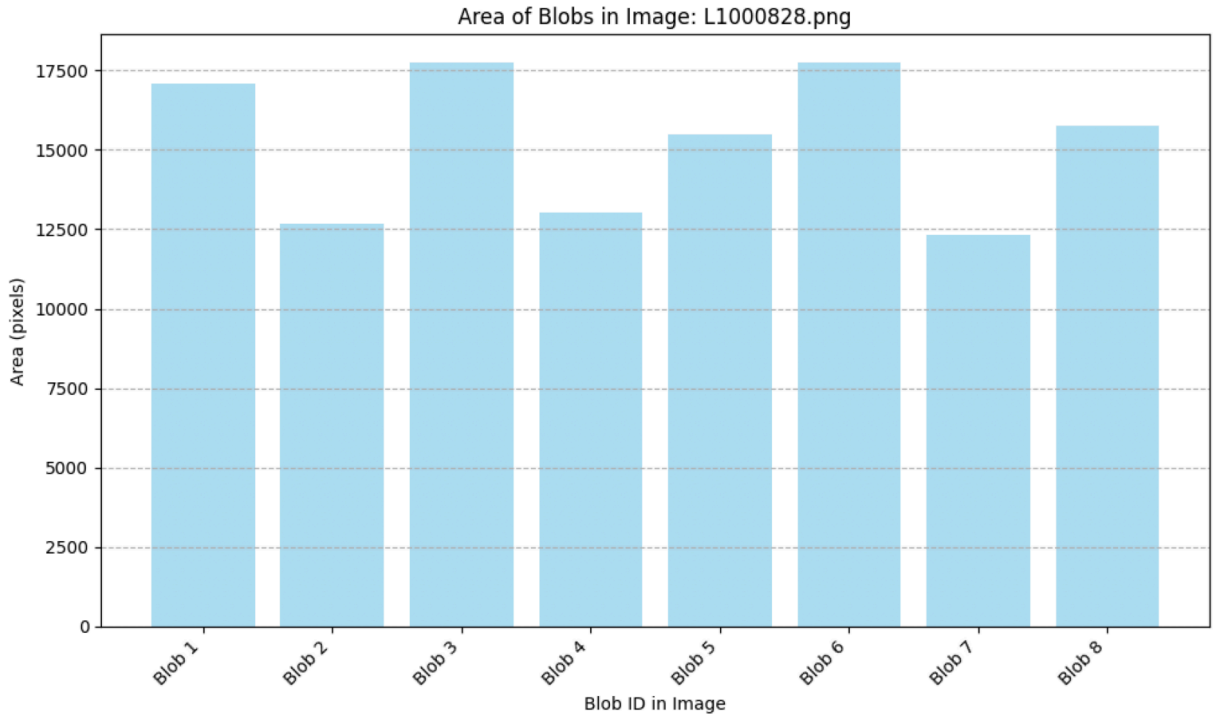



Since we had much information on the shape of the chocolates from the segmentation, we used a set of features from skimage.measure.regionprops such as area, elongation, perimeter as well as 7 hue moments.

In [28]:
```
plt.figure(figsize=(20, 50))
plt.imshow(plt.imread('hu_moments_signatures_L1000756.png'))
plt.axis('off')
plt.show()
```




In [29]:
```
plt.figure(figsize=(20, 50))
plt.imshow(plt.imread('area_per_blob_L1000828.png'))
plt.axis('off')
plt.show()
```




We used also information about the color of each segmented chocolate by comparing the HSV histograms of each segmented region of the original image

with the 13 references segmentations. For that purpose we also segmented properly the 13 reference chocolates and calculated their respective HSV histograms.

Finally we measured the similarity between both histograms using the bhattacharyya distance and kept the five smallest distances as features for training a supervised model.

We then trained different machine learning models to learn from the previous features. We therefore manually labeled the detected blobs from all neutral-background images into 13 + 1 classes to also classify non-chocolates objects that were segmented.

Among kNN, RandomForest and SVM models, a tuned SVM yielded the best classification results (see last section for quantitative results).

So overall this approach yielded satisfying results for neutral-background images but struggled to generalise to other backgrounds

## General pipeline

One of the core parts of our pipeline is the sliding window detector. We came about this idea after having tried multiple classical object detection and segmentation methods. For example, we tried using region growing and contour detection, but these methods did not perform well on images with non-uniform backgrounds. Due to the clutter and the variety of backgrounds, it was difficult to foresee using contour or region based methods.

Using a sliding window is much more robust to these variations. The idea is to take a window of a fixed size and slide it over the image with a fixed stride. For each position of the window, we compute the histogram of the pixels inside the window. This histogram is then compared to a set of histograms that we have pre-computed for each reference chocolate. The best-matching histogram is then used to determine the likelihood of the window containing a chocolate.

This procedure leaves us with a heatmap showing the likelihood of each pixel being part of a chocolate. By thresholding, we can obtain a binary mask of the detections. The detections are then in the form of 'blobs' in the heatmap. To detect these blobs we use edge detection through the Laplacian of Gaussian (LoG) method. This method has input parameters that allow us to control the size of the blobs we want to detect, and how many by specifying a threshold.

After the blobs have been detected, we can use the bounding boxes of these blobs

to crop the original image and obtain the chocolate candidates. From these patches, we can then compute the features we want to use for classification.

## Features

In our project we use a variety of features. Since we had trouble dealing with noisy backgrounds, our method does not rely on segmentations of the chocolates. So, the features we compute do not include any shape or contour based features. Instead, we use a combination of texture features, color features, and some basic statistics. Below we list the features we compute:

**Color based features**: color statistics such as means and standard deviations in multiple color spaces (RGB, LAB) and color histograms in these same color spaces.
**Texture based features**: Local Binary Patterns (LBP), Haralick GLCM features, and Gabor energy.

Note that these features were computed on subdivisions of the segmentated reference chocolates and on the extracted patches of the train/test images. Subdivisions were done by splitting the masked reference images into a certain number of patches. The number of patches depends on the size of the reference chocolate.

## Classification

The basic idea of the classification is to train an SVM classifier (which was found to be the best performer amongts other solutions such as random forests and gradient boosted trees) on feature vectors extracted from image patches which, thanks to the segmentation pipeline, have been extracted from the training set images.

# Technical Description

## Reference Image Processing

Before segmenting and extracting features from the reference chocolates, we perform a series of preprocessing steps to isolate each chocolate in its image. Since each reference image contains a single chocolate on a relatively uniform background, segmentation can be approached with simple but effective methods.

First, images are downsampled by a factor of four to reduce computational cost during edge detection and morphological operations. Edge detection is then

performed using the Canny algorithm, which includes Gaussian smoothing to suppress noise and enhance true edges. This is what `compute_masks` does.

For most reference chocolates, the Canny edges capture the chocolate contours well. However, in cases like Jelly White and Comtesse, the chocolates are low-contrast and similar in color to the background, making Canny alone insufficient. To address this, we apply the Hough ellipse transform to fit an ellipse to the detected edge points. This yields an approximate contour of the chocolate even when edges are faint or incomplete. The best ellipse is selected based on the accumulator score, and its perimeter is rasterized back into an edge mask.

Once edges are obtained, either directly from Canny or via the fitted ellipse, we apply a morphological closing operation using a 7×7 kernel to seal any small gaps in the contour. We then fill internal holes to obtain a solid binary region, and apply erosion to shrink the mask slightly, ensuring that background pixels near the boundary are excluded.

The result is a clean binary mask isolating the chocolate, which we use for two purposes: to extract visual features from the chocolate region, and to compute global color histograms that serve as templates for the sliding window detector.

## Sliding Window

The sliding window detector is implemented in our `sliding_window_compare` function, which systematically scans the image with a fixed-size window (default: 64×64 pixels) and stride (default: 16 pixels). At each position, a patch is extracted and its color histogram is computed using the RGB channels, with histogram parameters tuned for sufficient granularity (16 bins per channel). We use a smoothed histogram via a Gaussian kernel to improve robustness to noise and small variations.

This patch histogram is then compared to all reference chocolate histograms using the Bhattacharyya distance, a metric that quantifies similarity between probability distributions. The closest match is selected, and the inverse distance ($1 -$ distance) is used as a similarity score. These scores are stored in a heatmap aligned with the image grid, representing local similarity to known chocolate types.

This heatmap serves as the input for the next stage of the pipeline, where we apply blob detection via the Laplacian of Gaussian (LoG) to isolate likely chocolate candidates.

## Blob Detection

The `compute_blobs` function identifies chocolate candidates by detecting local maxima in the heatmap produced by the sliding window detector. It uses the Laplacian of Gaussian (LoG) method via skimage.feature.blob_log, which is well-suited for detecting roughly circular blobs at multiple scales. The parameters min_sigma, max_sigma, and thr control the minimum and maximum expected blob size and the detection threshold, respectively.

Once raw blobs are detected, their scale (σ) is converted to an approximate radius $R=2\sigma R=2\sigma$. However, LoG often produces multiple detections for a single object, especially when objects are large or span several overlapping regions. To address this, the function clusters nearby blobs using DBSCAN, treating detections within avg_R × 1.2 pixels of each other as belonging to the same chocolate candidate. For each cluster, the center is computed as the average of member centers, and the radius is set based on the selected merge_policy: the average radius, the maximum, or a fixed value (avg_R).

The final output is a list of candidate detections, each represented as a circle (y, x, R) in heatmap coordinates, which can be mapped back to the original image for cropping or further analysis.

## Patch Extraction

The `extract_crops` function extracts square patches from the original images, centered on the detected chocolate blobs. It takes as input a list of images, a list of blob detections for each image, and parameters defining the crop size, sliding window size, and stride.

For each blob, the function first converts the blob coordinates from heatmap space to image space. This is done by scaling the coordinates by the stride and offsetting by half the window size to recover the center position in the original image. A square region of size crop_size × crop_size is then extracted around this center.

To ensure consistency in input dimensions, the function pads the crop with zeros if it would otherwise extend beyond the image boundary or be smaller than the target size. This guarantees that all extracted patches are uniform in shape and suitable for downstream classification.

The output is a list of lists of crops, where each sublist corresponds to the set of chocolate candidate regions extracted from a single image.

## Training Data Labeling

With the segmentation pipeline crops containing chocolates were extracted from the training dataset images and subsequently underwent manual labelling. Thus, each extracted patch was given one of 14 labels, i.e. 13 for the different chocolate classes and an additional 1 class encoding non-chocolates image patches

## Feature Extraction

At this point in the pipeline we have both the reference chocolate masks and labeled training patches. For the reference images, we subdivide the masks into patches of size 64x64 and compute the features on these patches. This is in a sense data augmentation, since we can use the same chocolate to compute multiple feature vectors and thus increasing the number of training samples.

Considering all reference patches and training patches, this gives us a matrix of features of size 821 x 818.

Using all features gives us feature vectors of length 818. Not all of these features may be useful, so we use PCA to reduce the dimensionality of the feature vectors. The number of components is set such that 95% of the variance is explained, which is a common practice. Before doing so it is necessary to standardize the features, since they are on different scales. We use the StandardScaler from sklearn to do this. The PCA is done using the PCA class from sklearn. The PCA is fitted on the training data (which includes the reference data) and then used to transform the future test data. This is done to ensure that the test data is transformed in the same way as the training data. The same can be said for the StandardScaler.

## Classification

After the main pipeline for detecting and extracting chocolate crops from images is completed, it is then used on the training set images. All extracted patches are then combined with retrieved (manually) from the reference images in order to put together a larger training dataset for the downstream classifier. With the help of an auxiliary "labeller" dash application, all crops are labelled with one of 14 different labels (i.e. the 13 different chocolate classes with an additional one for out of distribution samples). Subsequently, one feature vector is extracted from each crop. All vectors are put together (with a np.vstack maneuver) in a 821 x 818 matrix which is then standardized and reduced with PCA. An SVM classifier is then trained and finetuned with 5-fold CV.

# Quantitative and Qualitative Analysis

## Quantitative Results

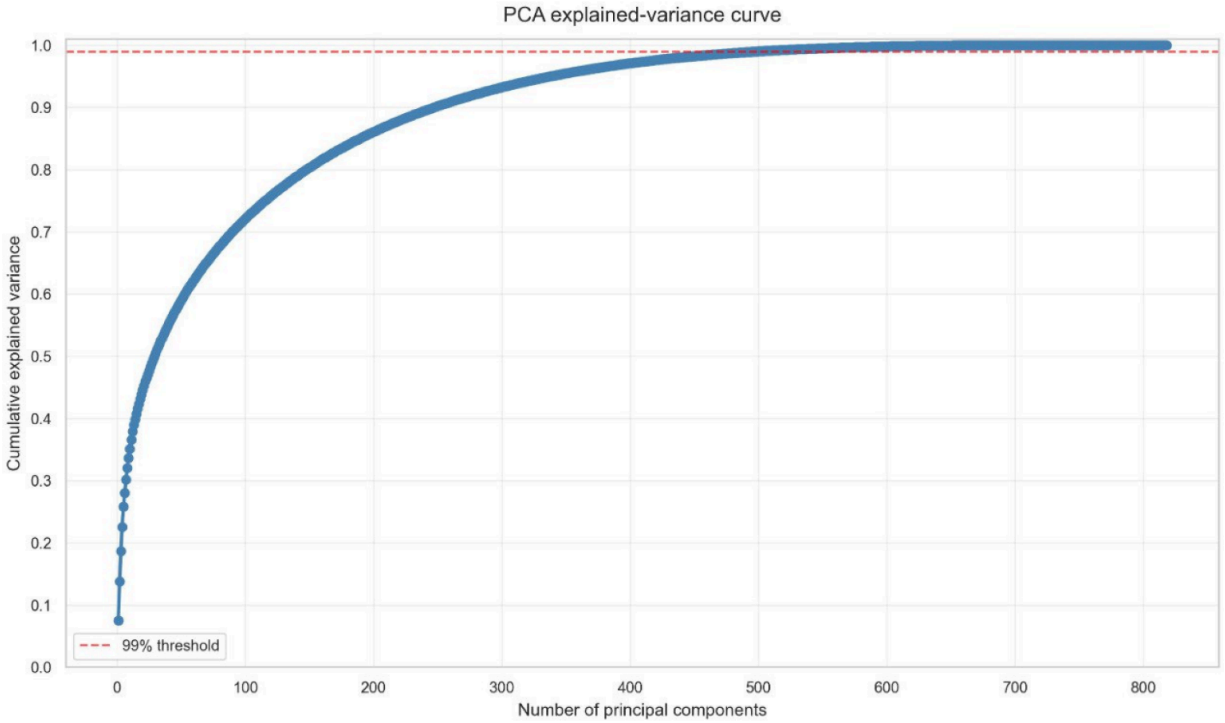
### Main Results

The results of our pipeline and classification are in general very satisfactory considering we only use classical methods. We obtain a best F1-score of 0.85 after having optimized the various hyperparameters of the pipeline. As for the results on the public test set (those on Kaggle), we obtain a best F1 score of 0.73575 (with the same parameters as the ones used for the train set). We noticed that for some sets of parameters, the F1 score on the train set was higher than previous ones, but the F1 score on the public test set was lower. This is a sign of overfitting, which is something less present when using classical methods than when using deep learning methods.

#### Other Results

Below is a plot of the cumulative explained variance as a function of the number of components. This shows that we can reduce the dimensionality of the feature vectors to 99% of the variance with less than 500 components. This is a good result, since it means that we can reduce the dimensionality of the feature vectors significantly without losing too much information.

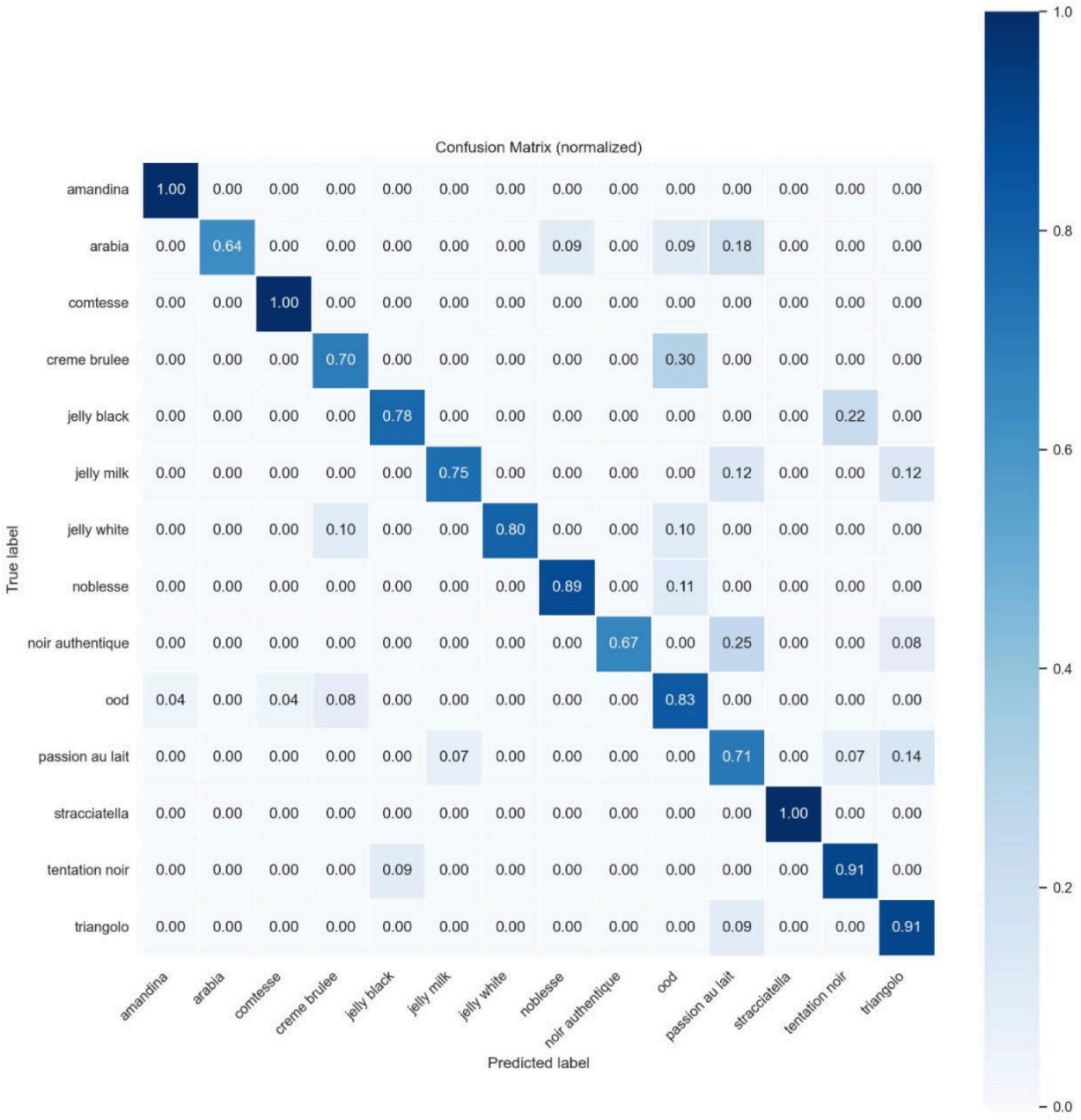
In [30]:
```
plt.figure(figsize=(20, 50))
plt.imshow(plt.imread('PCA.jpg'))
plt.axis('off')
plt.show()
```




Here is the plot of the confusion matrix obtained for the training data (which, we should remind, includes the reference data). We can see that the model is able to classify most of the chocolates quite well (which can be seen as values close to 1 on the diagonal), with some exceptions. We can see that Creme Brulee is often classified as being Out-Of-Distribution (OOD), actually 30% of the time. This chocolate is quite different from the others, which is probably the cause of this. Though this is the worst result.

We would like to point out the fact that the passion au lait chocolates are not well classified (relatively to the others). We saw this more intuitively in the heatmaps: for this chocolate, the blobs were often located on the edges of the chocolate.

In [31]:
```
plt.figure(figsize=(20, 50))
plt.imshow(plt.imread('matrix.jpg'))
plt.axis('off')
plt.show()
```




# Qualitative Results

## Reference Image Segmentation

Below we show the results of the segmentation on the reference images, after applying the Canny edge detector and the Hough ellipse transform when necessary, as well as the morphological operations.

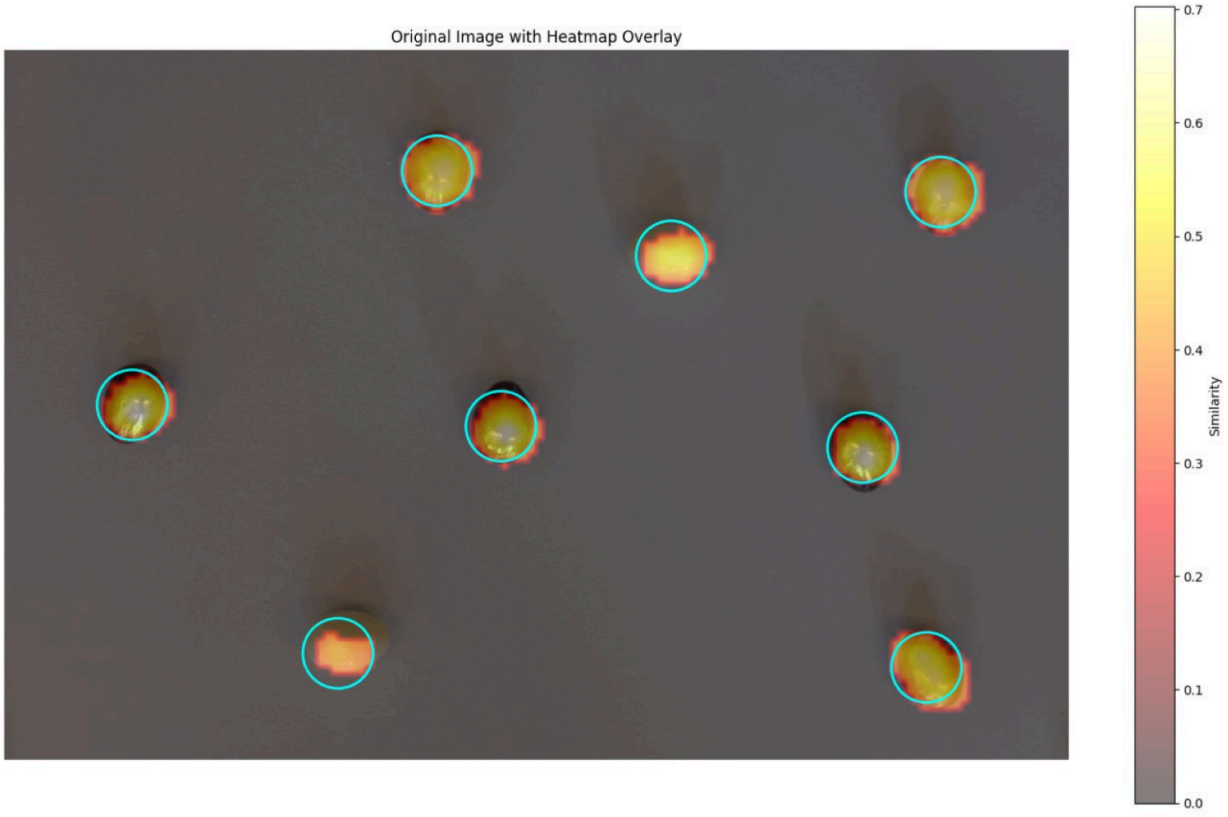
## Sliding Window Results: Heatmaps

Below are some examples of nicely performing heatmaps.

The image on the left is for a uniform background while the one on the right is for a noisy background.

We can see that in both cases the histogram matching is able to produce clean heatmaps and the blob detection works perfectly.

In [32]:
```
plt.figure(figsize=(20, 50))
plt.imshow(plt.imread('heatmap_neutral.jpg'))
plt.axis('off')
plt.show()
```




Below is an examples of poorly performing heatmaps on the two background types.

The complex background makes it harder to detect the chocolates. This can be attributed to the fact that the sliding window is based off the color histograms. So if parts of the background are similar in color to the reference chocolates, the sliding window method will introduce false positives. In general we observed that these false positives showed up with a smaller intensity than the true positives, which is why we used a threshold to filter them out. Though this is not a perfect solution, it does help to reduce the number of false positives. Another problem stems from the fact that the sliding window is smaller than the chocolate size. So it is possible that the window detects the same chocolate multiple times. This is why we used the DBSCAN clustering method to group the detections. This is not perfect either, especially in the cases when the chocolates are very close to each other. In this

case, the DBSCAN will group them together and we will only get one detection instead of two.

In [33]:
```
plt.figure(figsize=(20, 50))
plt.imshow(plt.imread('heatmap_flowers_bad.jpg'))
plt.axis('off')
plt.show()
```


## Patch Extraction

In [34]:
```
plt.figure(figsize=(20, 50))
plt.imshow(plt.imread('patch.jpg'))
plt.axis('off')
plt.show()
```