NTNU – Trondheim
Norwegian University of
Science and Technology

Classifying filling rate of waste in containers using Machine Learning

Jonas G. Lien

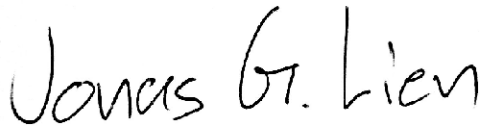
Master's Thesis for Mechanical Engineering
Supervisor: Amund Skavhaug
Submission date: December 2018

Norwegian University of Science and Technology
Department of Mechanical and Industrial Engineering

Preface

This is the concluding master's thesis of a five-year study programme in Mechanical Engineering at the Department of Mechanical and Industrial Engineering, NTNU. This thesis was written in the fall of 2018, unlike the norm of writing in the spring. This was done due to an increasing interest in computer science and its relation to industry, such that one more semester was spent on extra subjects in the computer science and cybernetics fields.

With the push for digitalization in industry, the last few years have been spent preparing accordingly. My studies have taken me on a journey from learning about electronics, hardware and microcontrollers, to industrial machinery, real-time software development, data processing, manufacturing technology and operations management. My pre-thesis specialization project was about Industry 4.0, industrial monitoring and data acquisition. Thus, it seemed fit and exciting for this master's thesis to be on the other side of the data - not collecting it, but making use of what is collected.



Jonas G. Lien - Oslo, december 2018

Acknowledgements

I would like to thank my supervisor, professor Amund Skavhaug, for providing me guidance through this master's thesis, both technically and motivationally. Even though I moved to another city while writing this thesis, prof. Skavhaug spared no expense in availability in spite of my disappearance off campus, and continued to offer guidance over the internet or via cell phone, even with great flexibility with regards to time and date.

I would also like to thank Veidekke for letting me use their data as basis for this thesis. Acando also has my great appreciation, with special thanks to Antonio Martellotta and Rune Larsen for providing the processed dataset used in this thesis, as well as intellectual sparring, guidance and great discussions.

Finally I would like to thank my dear partner, Torunn Vainio Gjøen, for always being there for me.

Summary

This thesis is doing exploratory work on a project between two companies, Veidekke and Acando. On Veidekke's construction site, there are containers filled with waste. They want Acando to automate the process of finding out when a container is full, and alert those responsible for picking up the full container and replacing it with an empty one. This was decided to be done by monitoring the containers by camera and use a supervised machine learning method called Convolutional Neural Networks (CNN), to classify the filling of waste and detect when the container is full.

Implementing a CNN places certain requirements. A large number of images is needed to train the network, and the images need to be labeled such that the CNN is trained to correctly classify new images. As this can prove costly, this thesis aims to find a solution that can bypass labeling, and use less images. This is done by using an unsupervised learning machine learning method called K-means, which groups data based on proximity. In order to group images, they need to be translated to a numerical representation. This is done in this thesis by applying six feature extraction methods on each image, and creating a feature vector of single number representations of each feature extraction.

Four different tests were conducted in order to measure to which degree the unsupervised learning method correctly classified the labeled images. A CNN was implemented, trained on the same dataset of images and tested for success of this method with regards to the same parameters. The filling rate of the container was divided into 5 classes, namely "0%", "25%", "50%", "75%" and "100%" full. The goal of the tests were to see how successful each method could correctly classify all classes.

It was found that the CNN had a success rate of 49.5% correct classification, whereas the unsupervised learning method scored 57.4%. The relatively low success rates were found to be due to too small of a dataset for the CNN and some human error in classification. For the unsupervised learning solution there were possibly unsuccessful feature representations, due to environmental noise. However, a way was found to automatically classify images with a drastically reduced amount of only 25 images, which could prove promising in case better feature representations or cleaner environment are used.

Sammendrag

I denne masteroppgaven gjøres eksperimentelt arbeid på et pågående prosjekt mellom to bedrifter, Veidekke og Acando. På Veidekkes byggeplasser er det mange avfallscontainere. Veidekke ønsker at Acando skal automatisere prosessen med å tømme fulle containere, hvor en ekstern bedrift henter en full container og etterlater en tom. Acando bestemte at dette skulle bli gjort ved å overvåke containerne med et kamera, og detektere når det ble fullt via en maskinlæringsmetode kalt Convolutional Neural Network (CNN).

Innføringen av et CNN medfører visse krav. En stor mengde bilder må manuelt klassifiseres for at CNN kan trenes opp til å detektere sitt gitte objektiv. Ettersom at dette kostbart i form av tid og penger, undersøker denne oppgaven muligheten for å finne en løsning som ikke krever manuell klassifisering og krever færre bilder. Dette er gjort via en ikke-veiledet maskinlæringsmetode kalt K-means, som grupperer data i henhold til datapunktenes euklidiske nærhet. For å kunne gruppere bildene via K-means, må bildene kunne representeres numerisk. Dette blir gjort ved å anvende Feature Extraction-metoder på bildene, for så å danne en vektor bestående av numeriske representasjoner av disse metodene.

Totalt fem forskjellige tester ble gjennomført i denne masteroppgaven. Fire av disse bestod i å undersøke ikke-veiledet maskinlæringsmetoders mulighet for automatisk klassifisering. Den femte testen bestod i å implementere en åpen kildekodeløsning for et CNN, for å se hvor godt denne kunne klassifisere dataen gitt for dette prosjektet. Datasettet av bilder brukt i denne oppgaven ble delt inn i fem klasser. Klassene representerte i hvilken grad en container var fylt, henholdsvis "0%", "25%", "50%", "75%" og "100%". Målet med denne oppgaven var å se hvor suksessfull de forskjellige metodene var i å korrekt klassifisere disse fyllingsgradene.

Det ble funnet at CNN-metoden hadde en suksessrate på 49.5% korrekt klassifisering, hvorav den beste ikke-veiledete maskinlæringsmetoden oppnådde 57.4% korrekt klassifisering. Den relativt lave suksessraten var funnet å grunne i at datasettet var for lite for CNN-metoden, kombinert med at den manuelle klassifiseringen trolig gjorde visse feilklassifiseringer. For ikke-veiledet læringsmetodene var det antatt at Feature Extraction-metodene ikke var helt suksessfulle til å gi gode numeriske representasjoner grunnet støy

i bildene. Likevel ble det funnet en måte å automatisk klassifisere fyllingsgraden i containerne, også med drastisk færre bilder sammenlignet med CNN, som virker lovende for videre arbeid dersom man forbedrer muligheten for å utelukke støy.

Contents

List of Figures

List of Tables

Abbreviations

ANN	=	Artificial Neural Network
CNN	=	Convolutional Neural Network
HOG	=	Histogram of Oriented Gradients
SIFT	=	Scale-Invariant Feature Transform
IDE	=	Integrated Development Environment
SURF	=	Speeded-Up Robust Features
BLOB	=	Binary Large OBjects
PCA	=	Principal Component Analysis
HLT	=	Hough Line Transform
LoG	=	Laplacian of Gaussian
DoG	=	Difference of Gaussians

1. Introduction

1.1. Introduction and formulation of problem

The purpose of this thesis came about through dialogue with two different Scandinavian companies, Veidekke and Acando. Veidekke is an entrepeneur- and real estate development company, and Acando is an IT consultancy company. It was decided that this thesis would do exploratory work related to one of Veidekke's and Acando's newly started projects. Their project consisted of Veidekke wanting to automate the process of emptying containers full of waste, specifically containers containing wood. This required that the containers needed to be monitored, and the agency responsible for emptying the containers were to be alerted when the containers became full. As this project had already begun before this thesis' assignment was formulated, certain decisions from Acando had already been made. These decisions were that the containers were to be monitored via a camera, and use machine learning as the tool to detect when the containers were full. More concretely, it was decided to use one of the currently most popular machine learning implementations for image classification and object detection, namely a Convolutional Neural Network (CNN).

In discussion with Acando, one problem of the implementation of a CNN arose, namely the labeling and training problem. A CNN needs to be trained in order to properly detect what it is programmed to look for. This training consists of using a, preferably large, set of images which has been labeled according to what the aim is to detect. If, for example, the aim of a CNN were to detect a cat in an image, it would need a large dataset of different images of cats for training of the neural network. Then, after training, one could run a prediction on a new image, and the CNN would output the likelihood that the image contains a cat. In this case, the labels would be cat or non-cat. When Acando began its work of training a CNN for Veidekke's project, they decided that the CNN ought to be categorized into five different labels/classes. These labels were to represent the state of the

container, namely to which extent the container was full. Thus, the images taken from the container were categorized into the labels "0%", "25%", "50%", "75%" and "100%". After approximately a month of monitoring the containers, 4802 unique images of containers with varying degree of waste filling had been gathered. The images were manually sorted into the most fitting class, based on the judgment of the person labeling the image. A dataset of 4802 labeled images was thus made, based on three cycles of the container from empty to full.

The incoming rate of new unique images of containers in various states was low, as there could go days before new waste was added to the container. The manual labeling could also prove tedious and had a time/cost. This raised the question of whether one could simplify or work around the labelling process. This problem would not reflect only on this specific task, but any task where one is interested in building a similar vision-based detection system with little data to begin with. With these two problems in mind; how to create a vision-based detection system with little data, and how to improve or avoid the need for human labeling, the path of this thesis began take shape. Some overarching questions emerged, and formed the basis of what was to be investigated in this thesis:

- How can the states of the container be detected without the need for human labeling?
- What could a viable non-supervisory labeling process look like?
- Would such an alternative outperform a CNN? How would they compare?

These questions will be kept in mind throughout this thesis and answered in conclusion of this thesis. To begin to address these questions, it is first looked at research done with similar problems in focus.

1.2. History and related work

As said by Chester and Ratasby, image classification research aims at finding representations of images that can be automatically used to categorize images into a finite set of classes [22]. This problem has been worked on for many decades, dating back to the 1960s, where the earliest applications of pattern recognition systems were implemented for character recognition in office automation related tasks [23]. Ever since, entire fields of

research have been immersed with this problem, ranging from the fields of computer vision with the development of feature extraction and object recognition, to other fields such as robotics, signal processing, mathematics, statistics, psychology, visualization/graphics and artificial intelligence via machine learning and the use of neural networks as means of image classification [24].

With the recent explosion of machine learning and neural network applications in particular, the different solutions to image classification are typically divided into two camps. These are called *supervised learning*, and *unsupervised learning*, and are two of the main branches within machine learning. Different types of feature extraction is at the heart of both methodologies, but their means of classification differ fundamentally. For supervised learning, classification is the output of the neural network, as a prediction based on how the trained neural network interprets the input. For unsupervised learning, the dataset to be analyzed is not labeled, thus not knowing directly what class a data point ought to be assigned to. The classification is instead decided solely based on the inherent composition and structure of the data, as a result of how the data is grouped.

However, being that grouping/clustering is the central means of classification in unsupervised learning, visual grouping in computer vision has proved to be a hard problem to solve. Methods that aim to extract features from images have been heavily researched, where edge- or contour detection has been a large focus as means to detect shapes and objects [25]. But with regards to analyzing data such as unlabeled images, do "objects" fall directly out of the statistics of the environment, or are they a more subjective, human-specific construct [26]? The idea of leveraging cluster IDs of images as means of learning representations or for classification has become a popular idea to investigate [27]. Clustering aims to group data together based on proximity in euclidean space, and one specific clustering method called K-means has shown dominant appearance in the field of classification based on grouping [27]. With the rise of data analysis with a seemingly large set of variables, clustering is also used to attempt to group data in a meaningful manner in high-dimensional space [28]. However, high-dimensional cluster analysis can have some disadvantages. The data is hard to visualize once one surpasses 3 dimensions, and points in high-dimensional space can become sparse, making it difficult for cluster analyses to provide good results [28]. This problem has often been worked around by using dimension-

ality reduction, allowing for high-dimensional data to be visualized if reduced down to 3 or less dimensions. Much used dimensionality reduction techniques are Factor Analysis or Principal Component Analysis (PCA) [29]. In the case of PCA, hyperplanes are created for projecting points in high-dimensional space down to the reduced subspace, while retaining a possibly high degree of variance from the data [29].

In recent years, supervised learning via the method of Convolutional Neural Networks (CNNs) have gained much attention and shown good results for image classification problems [30]. A CNN trained in a supervised manner via backpropagation dramatically improved the state of the art performance on a variety of Computer Vision tasks, such as image classification [30], [31] and detection [32]. Interestingly, it seems features learned by such networks often generalize to new datasets [33]: for example, the feature representation of a network trained for classification on a dataset of 14 million images, called ImageNet [34], also performs well on PASCAL VOC, containing approximately 21.700 images [35]. This paved the way for applications such as Inception V3, which is a CNN where the top layer of the neural network is retrained by the designer for its own specific task [10].

1.3. Ideas to be investigated

Summarizing the problems for the container project and relevant research in the fields of image classification can serve as a guide towards what could be interesting to investigate in this thesis. Veidekke and Acando have provided a dataset of 4802 labeled images. This is the result of how many unique images/how much valuable data that was possible to gather within approximately a month. In order to increase the size of the dataset, new images needs to be added and labeled manually. A convolutional neural network is shown through research to provide good results for object detection and image classification [34]. The development of feature extraction of images have made it possible for the designer to process an image in search of an object, and thus provide a descriptor of the image based on the result of the feature extraction. Unsupervised methods, such as K-means, is used to group data based on their proximity in euclidean space, and can serve as a classification mechanism on unlabeled data. Principal component analysis is a means of reducing the dimensions of a dataset, while retaining a high degree of the variance of the data.

This thesis thus aims to explore the dichotomy between supervised learning and unsupervised learning for its strengths and weaknesses when applied specifically to the task of automating recognition of a containers fill rate. This will be done by performing the following tasks:

- Search for feature extraction methods that fit this specific project.
- Creation of a feature vector for each image in the dataset, where the feature vector consists of representations of all feature extraction methods used.
- Use of unsupervised learning methods to test if classification/labeling can be performed on the feature vectors with satisfactory results.
- Implement a CNN by training on the dataset and evaluate its utility considering the size of the provided dataset.
- Comparison of the results of the different solutions.
- Comparison the utility of the different methods applied to this specific project.

1.4. Process of selecting methods and tests chosen

The selection process for the different methods was based on what seemed to fit this specific project best. Python was chosen as the language of implementation for all the tests and method implementations, due to its rich libraries and wide user community, as discussed on chapter 3. Open source projects in Python have made it easy to create a CNN for custom classification, and there is a wide selection of feature extraction methods and other image processing tools available. The feature extraction methods used in this thesis were chosen based on them being the methods that gave the best single numerical representations of the container filling problem, compared to other feature extraction methods found in various Python libraries, while testing by trial and error. One open source solution for creating a CNN, called "Inception V3", was chosen due to its relative ease of implementation and customizability. Regarding the unsupervised learning methods, there were two main branches to choose between. "Partitioned clustering" and "agglomerative clustering". Agglomerative clustering was discarded due to the nature of the clustering being counter-productive with regards to this thesis. With partitioned clustering one has

more control of how and where the data ought to be clustered. Two methods are presented in chapter 2, but only one - K-means - is selected for further use. This is due to the high computational requirements of the other method, "affinity propagation". With K-means, among other clustering methods, it is possible to decide how many clusters one wants the data to be grouped to. As this thesis both utilizes a labeled dataset and has pre-determined how many classes/labels there are to be selected, one can simply assign the number of clusters to equal the number of classes. However, unsupervised learning is most often used on unlabeled data, with the aim to explore how it is best to group the data. In this case, it might not be given how many clusters gives the best fit to the data. Methods used searching for the best fit for clustering go under the term *cluster validity*. Even though this will not be directly used in this thesis, theory for some of these methods are presented in chapter 2, in case future work based on this thesis should use unlabeled data with possibly undetermined amount of classes.

Six feature extraction methods were selected for testing the possibility of unsupervised classification. Five of these were methods obtained by different Python libraries, while one was developed by the author of this thesis. The success of unsupervised classification will be tested via the use of K-means, and will be judged on the basis of *correctness*. In this case, correctness means "to which degree does the points in a cluster have the same true label (assigned fill rate) as the cluster ID?". This is done by assigning each feature vector a new numerical representation for its true label. Instead of the assigned true label being "0%", "25%",..."100%", it is converted to "0","1",...,"4". Thus, a feature vector of an image that is for example labeled as 25% full, would look like $[fr_1, fr_2, fr_2, fr_3, fr_4, fr_5, fr_6, 1]$, where fr_1, fr_2... would be the feature representation by a single numerical value, for each respective feature extraction method, and "1" represents the true label of "25%" full. When creating clusters in K-means with the code used in this thesis, the cluster IDs range from "0" and upwards til "$k-1$" clusters. Thus, the *correctness* measures how many of the points within a given cluster has the same true label representation as the cluster ID. E.g., how many of the points within cluster 1 contain true label representation "1". This will be explained more in detail in chapter 4 - Tests and results.

Several different tests are run in order to see which gives the highest correctness. Clustering via K-means will tested in 6-dimensional space with all six feature representations,

and in 2-dimensional space with every combination of two of the six feature representations. This is done both to see if a high-dimensional cluster analysis would provide better results due to the use of more features, and in the case of the 2-dimensional cluster analyses to see if any of the features performed better than others. It will also be tested whether Principal Component Analysis can find hyperplanes which are better suited to cluster the projected points, than only clustering in "untouched" euclidean space. Due to the fact that clustering on feature extracted images is based on the inherent structure of the image, it will be tested whether one can get similar or satisfactory results with a drastically reduced dataset. The method that shows the best results (highest correctness), which is found to be the PCA-reduced K-means clustering, will be selected to be tested on a small subset of the complete dataset. In order to test if there is any validity to the claim that such a small subset can be used to gain similar results to the test with the complete dataset, a population proportion test is conducted right after.

To give a visual representation of the testing process for the unsupervised classification tests, figure 1.1 shows a flowchart practically representing the process.

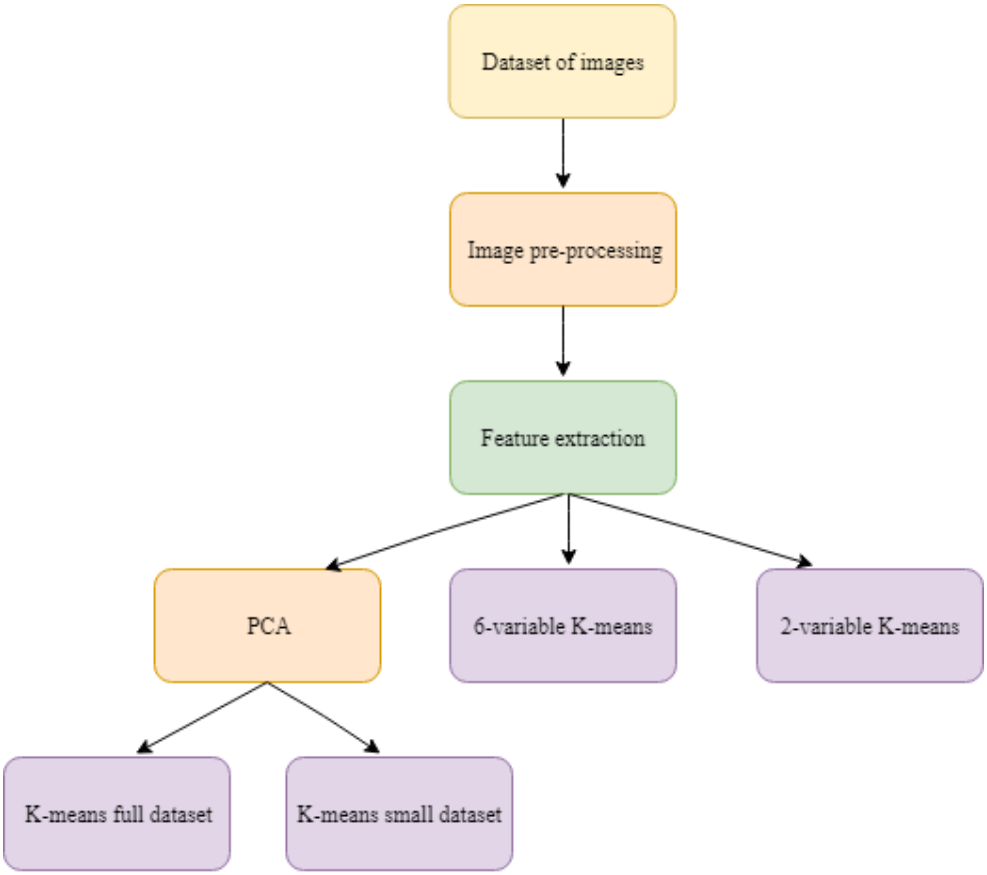


Figure 1.1.: Flowchart of process for unsupervised learning tests.

A CNN is trained on the dataset provided, using the open source "Inception V3" framework. Inception V3 works such that one trains a CNN on a dataset of labeled images, and afterwards runs predictions on one image at a time. The output of the prediction is the value of how probable it is that the image belongs to the different classes. For example, lets say an image of a roughly half full container is predicted by the network. The output could then be given as:

- 0%: 0.00
- 25%: 0.15
- 50%: 0.60
- 75%: 0.20
- 100%: 0.05

Due to the fact that the training set only spans 3 cylces from empty to full, a new and smaller dataset is given from Acando, which contains 250 new labeled images that are to be used as the test set for the CNN. Where normally a dataset is split in two, with typically 80% being used as training set and 20% is used as test set, images from new cycles are used as test set to reduce the chance of the neural network just recognizing the approximately very similar images it was trained on.

Chapter outline

Chapter 2

Chapter 2 portrays the theoretical foundation for the methods used in this thesis. The reader is introduced to image processing, various feature extraction methods and machine learning methods.

Chapter 3

Chapter 3 displays the tools used to implement the various methods used in testing, along with an account of why the different tools were selected.

Chapter 4

Chapter 4 presents the test setup, all tests conducted and the results from all tests.

Chapter 5

Chapter 5 portrays a discussion of the results, reasons for why and how the tests where conducted, discussion of the methods used and their underlying assumptions, and a comparison between results and the utility of the tests performed.

Chapter 6

Chapter 6 concludes the thesis, summarizes what has been done and presents recommendations for future work.

2. Theory

This chapter provides the theory for all methods used in this thesis, with the addition of cluster validity for potential further work. This chapter is roughly divided thematically in two parts. The first part regarding the image processing methods used in this thesis, and the second being the theory of the different machine learning and statistical methods used for the tests conducted.

2.1. Image pre-processing

A number of pre-processing steps can be applied to an image before it is used or manipulated further. For this project however, it was found that little pre-processing was needed, with additional popular pre-processing techniques providing worse results. Thus, only one type of filter will be used in pre-processing, namely the 2D convolution filter called Mean Blur.

2.1.1. Kernel Convolution - Mean Blur

Filtering an image is the process of mapping a new image based on the new values that are given for each pixel, or batch of pixels, after a *kernel* has run over it. This is done by taking a small grid, e.g. an 5×5 grid as the "mean blur kernel" shown beneath, and running it over the entire image.

$$K = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

This means that each pixel value in the selected 5×5 grid on the original image is multiplied by the value of the kernel element that is on top of it. When the kernel has filled up with all the element values, all values are added up and divided by the size of the kernel. This average value of the kernel is now the *new* pixel value at the position of the center of the kernel, on the new transformed image matrix. A visual example, albeit with a different kernel, shows the way a filtered image is created.

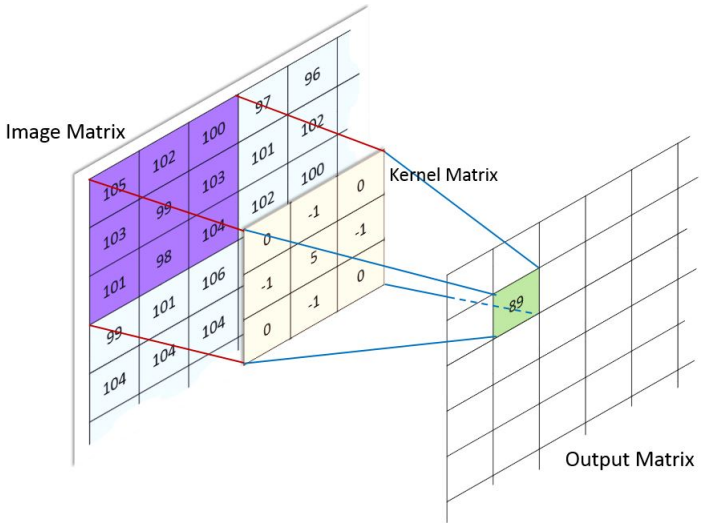


Figure 2.1.: Kernel convolution example [1].

2.2. Obtaining features from images

How to extract features of an image or how one can manipulate data in a matrix, such that changes to different pixel intensities can make a computer perform judgment whether something in an image is an object or not are the central questions to the fields of object detection and image processing. In the following sections, methods aiming to answer these questions will be investigated. The theory of each method is explained, but their utility and reasons for implementation are discussed in chapter 4 and 5.

2.2.1. Histogram of Oriented gradients

Histogram of Oriented Gradients (HOG) is a feature descriptor used mainly in computer vision for object detection and classification. It's aim is to detect robust features which are distinct from the background and/or noise. The idea is that the local object's appearance

which is investigated can be characterized rather well by the distribution of the local intensity gradients, or edge directions[36]. This is done by dividing the image window into small regions, called cells. For each cell, the horizontal and vertical gradients are calculated by filtering the image with two kernels. These are essentially Sobel operators with kernel size 1, and are represented as $g_x = [-1, 0, 1]$ and $g_y = [-1, 0, 1]^T$ for the horizontal- and vertical axes, respectively [37][2]. At every pixel, the gradient has a magnitude and a direction, found by the use of the Sobel operators. The gradient, g, and direction, Θ, are calculated as follows:

$$\begin{aligned} g &= \sqrt{g_x^2 + g_y^2} \\ \Theta &= arctan\frac{g_y}{g_x} \end{aligned} \tag{2.1}$$

For each cell, this translates to two matrices. A gradient magnitude matrix and a gradient direction matrix. From figure 2.2, one can see an example of how this would look[2].

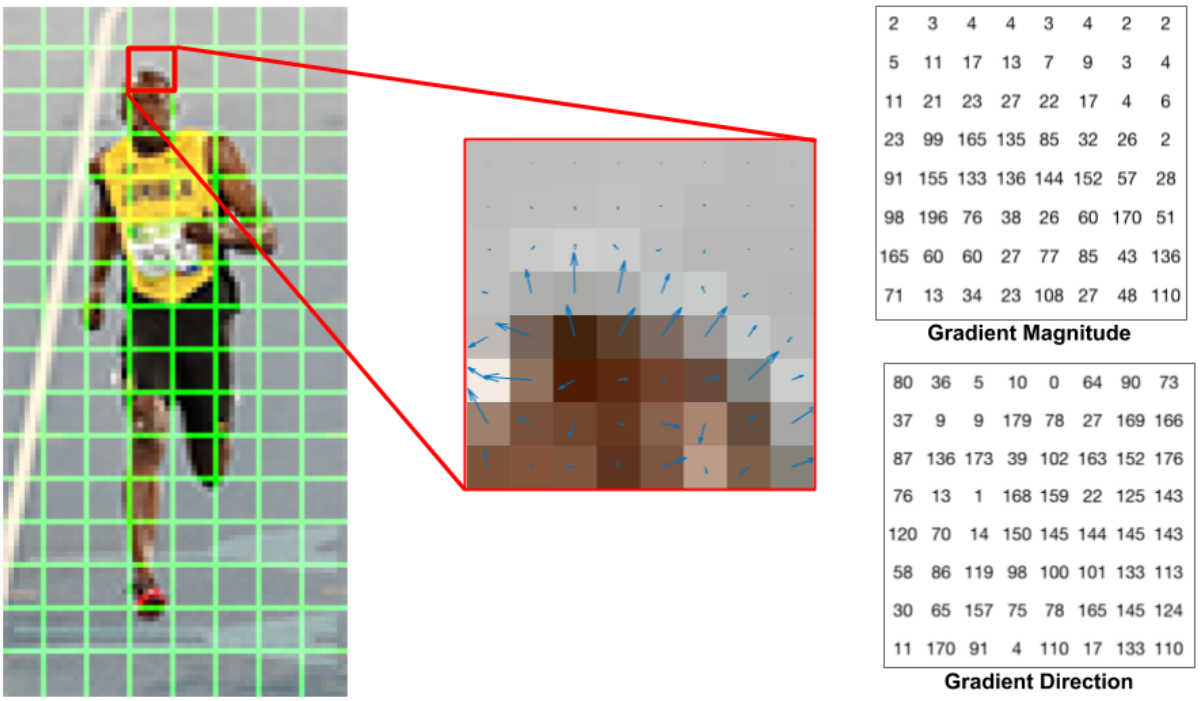


Figure 2.2.: HOG performed on an 8x8 cell[2].

Next, it is created a histogram of gradients in these cells. A bin is selected based on

the direction and a vote is made based on the magnitude. The histogram contains 9 bins corresponding to angles from 0, 20, 40 ... up to 160 degrees. Figure 2.3 shows the histogram is filled up and sorted based on its angles and magnitudes. In case a direction is between two possible angles, shown in red, it is evenly distributed between the two. The same goes for angles between 160 and 180. The histograms are then normalized, and possibly concatenated with the other cell's histograms if need be.

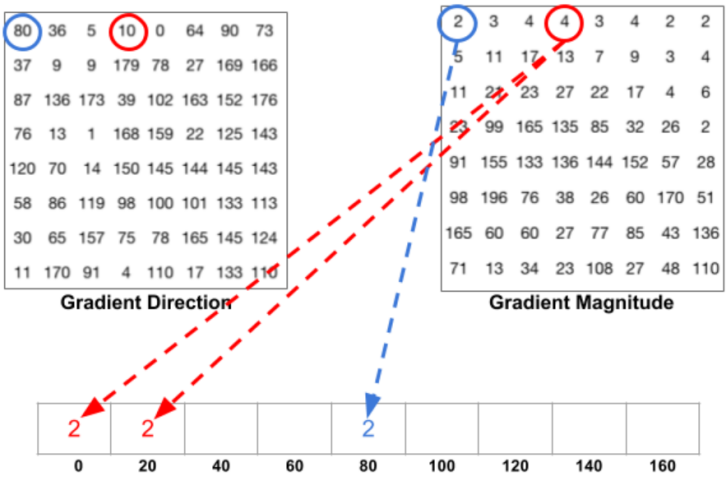


Figure 2.3.: Histogram voting/selection [2].

To apply the histograms to the original image for visual representation, each of the 9x1 normalized histograms of all 8x8 cells are added on top of the image, as shown in figure 2.4.

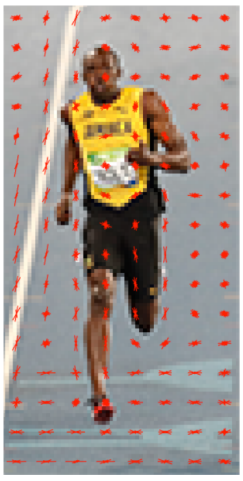
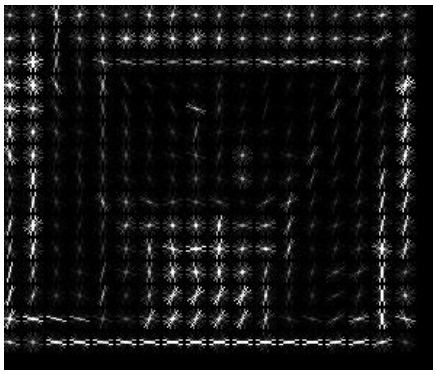


Figure 2.4.: HOG-matrix added on original image [2].

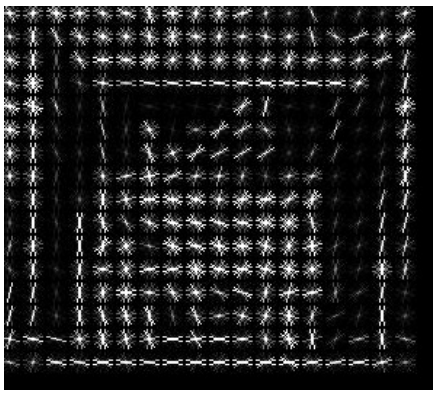
Applying this to two image test cases of the image dataset provided for this thesis, one can see the plots of the new HOG-matrices on the right side in figure 2.5.

(a) Almost empty container.



(b) HOG on almost empty container.

(c) Partially filled container.



(d) HOG on partially filled container.

Figure 2.5.: Showcase of HOG implemented on 2 images.

2.2.2. SIFT and SURF

A popular feature detection algorithm in image processing, is the **scale-invariant feature transform (SIFT)**[38]. It is used to recognize local keypoints in an image, invariant of the scale. This means, if one were to e.g. search for the same objects several images, this could prove troublesome if the images are not of the same size/scale or rotation. Not only is the SIFT-algorithm robust to rotation and scale, but it detects points of interest in an image rather well, and outperforms other methods such as Harris corner detection[39] when applied to this thesis' dataset. This algorithm, however, is not to be used as a means of comparison between pictures in this thesis, but as a tool for identifying keypoints in

an image. This is due to that the *amount* of objects/keypoints is what is of interest to this thesis. It is due to the hypothesis made that the more waste in a container, the more distinct keypoints are localized. A brief introduction is therefore given to the method as a whole, before focusing more primarily on the keypoint location.

When an image is scaled up, a corner might stop being registered as a corner if the filter window size is too small. So to detect larger corners, one needs larger windows. For SIFT, scale-space filtering is used. For this, a filter kernel which calculates the Laplacian of Gaussian (LoG) is typically used. The Laplacian of Gaussian is found for the image with different values of σ, where LoG acts as a *BLOB* (Binary Large OBjects) detector for blobs in various sizes due to change in σ. Examples of blobs are shown in figure 2.6.

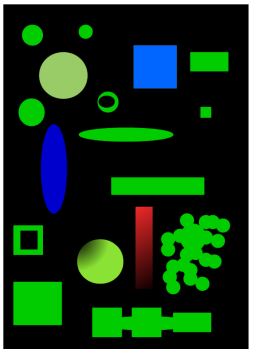


Figure 2.6.: Blobs of different shapes and sizes [3].

However, due to LoG being a little costly to perform [38], SIFT uses an approximation called Difference of Gaussians (DoG). Keypoints of the image are found as maxima or minima of the DoG, given by:

$$D(x, y, \sigma) = L(x, y, k_i\sigma) - L(x, y, k_j\sigma) \tag{2.2}$$

Where $L(x, y, k\sigma)$ is the convolution of the original image $I(x, y)$ with the Gaussian blur $G(x, y, k\sigma)$, such that $L(x, y, k\sigma) = G(x, y, k\sigma) * I(x, y)$, with the Gaussian blur represented as

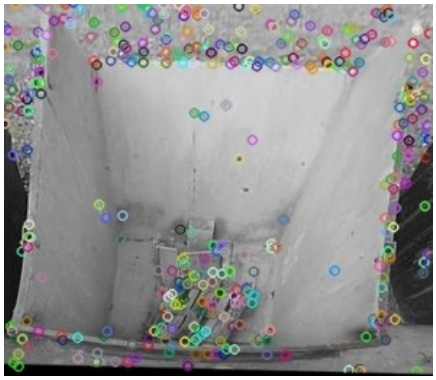
$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}. \tag{2.3}$$

Once the DoG is found, the local extrema are searched over scale and space. This provides potentially too inaccurate results, such that Tayolor series expansion of scale

space is used to get a more accurate location of extrema.

As explained in the beginning of this section, the purpose of implementing this algorithm is not for recognition of the same objects in several images. The full scale of SIFT's utility is therefore not given here, as the main objective is already reached. Shown in figure 2.7, two containers of various levels of waste are displayed with applied SIFT on the right.

(a) Almost empty container.



(b) SIFT on almost empty container.

(c) Partially filled container.

(d) SIFT on partially filled container.

Figure 2.7.: Showcase of SIFT implemented on 2 images.

Speeded-Up Robust Features (SURF)[4] is a similar feature detector to SIFT, implemented slightly different. Instead of approximating the LoG via DoG, like SIFT, it approximates LoG via different convolutional kernels called box filters [4]. The approximation is shown in figure 2.8, where on the left side is the LoG and on the right is the box filter approximation.

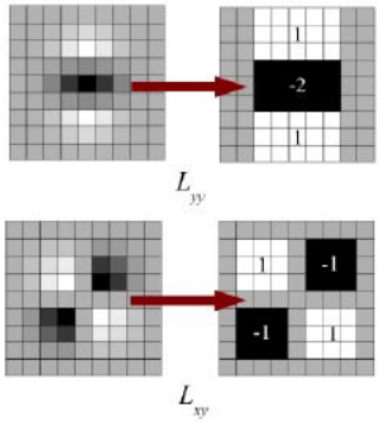
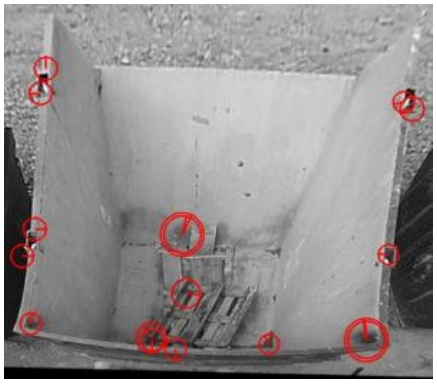


Figure 2.8.: Right side representing box filter approximations [4].

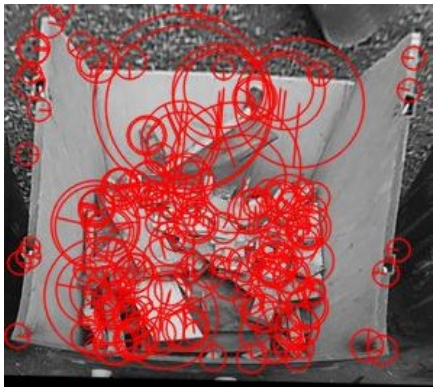
The reason for this is to save computational time, at the cost of a slight decrease in accuracy compared to SIFT [4]. The use case for this thesis is however the same, and SURF is being used as another way to represent the amount of waste with the amount of keypoints located. The similar test images as the previous feature detectors, can be shown for SURF in figure 2.9.

(a) Almost empty container.



(b) SURF on almost empty container.

(c) Partially filled container.



(d) SURF on partially filled container.

Figure 2.9.: Showcase of SURF implemented on 2 images.

2.2.3. Hough Line Transform

The waste in the containers of this thesis is wood, typically in the form of planks. A plank generally has the shape of a rectangle, with well-defined lines. One common problem of edge detectors is that some edges along a line might not be detected. This can cause a noisy interpretation of an image, in cases where a line seems apparent but only a few edges are detected. The Hough Line Transformation (HLT)[40] is therefore a solution to this problem, where lines are attributed to a series of certain edges based on a weight. This weight chooses the best fit line in case several edges are detected in close proximity, with several possible potential lines.

Let $r = xcos\Theta + ycos\Theta$ be the radius between the origin and an edge detected point of

interest, where the origin is any corner of an image, typically bottom left. For each point in two-dimensional space, the set of all possible straight lines through that point corresponds to a sinusoidal curve in the (r, Θ) plane which is unique to that point. A set of two or more points that form a straight line will then produce sinusoids which cross at (r, Θ) for that line, thus giving a distance and an angle for where the line ought to pass through, as shown in figure 2.10. The length of the line is then set to span between the two points with greatest distance between them on the same line.

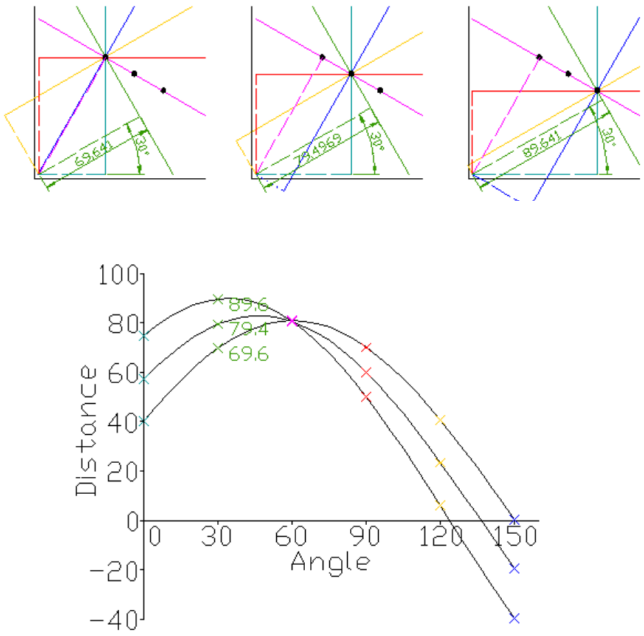
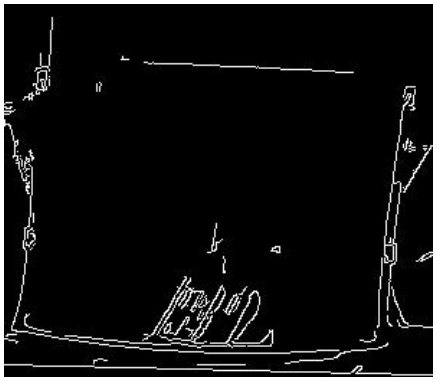


Figure 2.10.: Finding Hough lines [5].

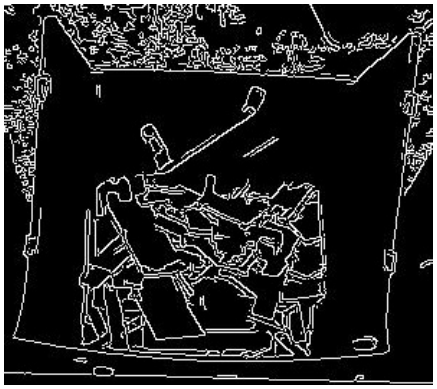
Implementing this transform on two example images from the dataset, the HLT transform matrix is plotted as an image shown on the right side of figure 2.11.

(a) Almost empty container.



(b) HLT on almost empty container.

(c) Partially filled container.



(d) HLT on partially filled container.

Figure 2.11.: Showcase of Hough implemented on 2 images.

2.2.4. Hough-distance

This is an algorithm that has been developed by the author of this thesis, with the purpose of obtaining a new feature representation based on the Hough Line Transform. When looking at the HLT-transformed images, some suppositions can be made. It seems that the more waste in the container, the more lines are created in the transformed image. The transformed image has 8-bit resolution, meaning each pixel intensity ranges between $[0, 255]$. However, due to the nature of the transformed image, and as one can see, the pixel intensity is only *either* 0 or 255. Black or white. Thus, a new descriptor was made for this thesis, where an algorithm runs on the output matrix of the HLT.

As one can see from the transformed images, there are greater regions of black in the image containing the least waste. An algorithm is created which acts as an accumulator.

Each row of the matrix is iteratively selected. For each row, a counter counts for how many pixels (elements in the row) it can travel before registering a white pixel. This count is given an exponential weight of 2, to reward the algorithm with a higher count the longer it takes between registering a white pixel (a new Hough Line), and is reset the moment it registers a white pixel. The amount of black pixels registered before a white pixel is detected can be though of as a batch. At the end of each row, the value of the batches for that row are added together and sent to the total accumulator. At last, the total count is divided by 100,000, a number found by trial and error to downsize the total count to an integer that ranges approximately between [0,5]. This is done due to the fact that principal component analysis is performed on all feature representations later in this thesis, and it is not wanted that this dimension blows up in euclidean space compared to the other features when performing the dimensionality reduction. In order for ease of reference in later use, the algorithm was given a name. For the sake of simplicity, the algorithm is named the "Hough-distance", and is shown beneath.

```
def Hough_dist(imageArray):
    totalCount = 0
    for i in range(len(imageArray)):
        AccumulatedCount = 0
        tempCount = 0
        for j in range(len(imageArray)):
            if imageArray[i][j] == 255:
                AccumulatedCount += tempCount ** 2
                tempCount = 0
            else:
                tempCount += 1
        totalCount += AccumulatedCount
        AccumulatedCount = 0
    return totalCount/100000
```

This algorithm can approximately be described mathematically as:

$$C = \sum_{i}^{rows} \sum_{j} b_j^2 \tag{2.4}$$

Where b_j^2 represents the weighted count for each batch of black pixels counted between two white pixels.

2.2.5. Histogram Threshold

A histogram of each pixel intensity ranging from [0,255] for the image is created. A threshold value, e.g. the value that is the median of the histogram of the image matrix, divides the image. Every pixel with intensity value lower than the threshold value is selected to be black (pixel intensity 0), and every pixel intensity which equals or is greater than the threshold value is set to be white (pixel intensity 255). Implemented on the two example images, this is shown beneath in figure 2.12.

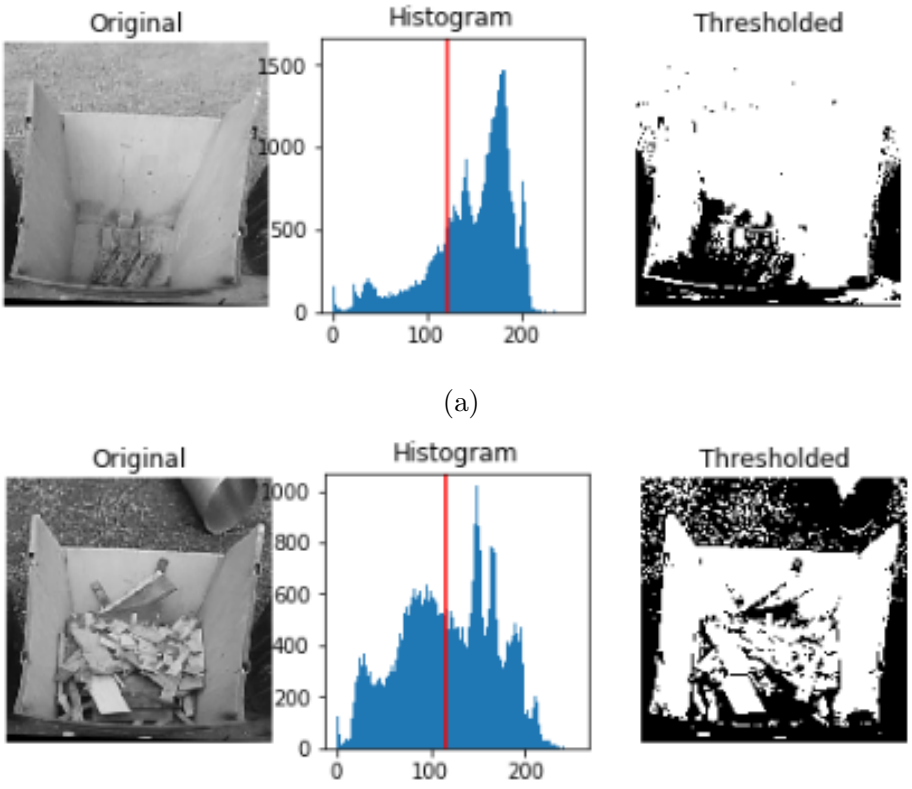


Figure 2.12.: Histogram Threshold implemented on two containers.

2.3. Machine learning branches

Machine learning is a branch within Artificial Intelligence that have grown immensely in the later years [41]. Within machine learning, it branches further. However, two main branches commonly divide the field machine learning. The *predictive* or *supervised* learning, and

the *descriptive* or *unsupervised* learning.

2.3.1. Supervised learning

The goal of supervised learning is to obtain a mapping from the inputs $\boldsymbol{x}$ to the outputs y, given a set of input-output pairs. This set is called a training set, typically represented as D. It is mathematically represented as

$$D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N} \tag{2.5}$$

Where N is the number of training examples.

The inputs $\boldsymbol{x}$ are called features, attributes or covariates, and are D-dimensional vectors of numbers representing the features one wants to investigate, such as e.g. the height and weight of a person, the colors of a car, etc. These inputs are often stored in a N x D matrix, X, called the design matrix.

2.3.2. Unsupervised learning

'Descriptive' or 'Unsupervised learning' is the second main type of machine learning after supervised learning. The goal is to find patterns in the data given. In unsupervised learning, inputs of the training set are given as

$$D = \{(\boldsymbol{x}_i)\}_{i=1}^{N} \tag{2.6}$$

where one can see there is no response variable y to classify the results. This branch of machine learning is also fittingly called "knowledge discovery". It is typically a less well-defined approach as one does not know what types of patterns one is looking for, and there is no obvious metric to use for determining the error of the approach. Unsupervised learning is also often referred to as "Cluster Analysis", as the purpose of most methods within unsupervised learning is to group, or cluster, data based on similarity. Cluster analysis has been applied to a variety of fields, such as medicine, socialogy, psychiatry, anthropology, criminology, archeology, geology, geography, market research, economics and engineering, to mention a few [42].

2.3.3. Reinforcement learning

Reinforcement learning is the third main branch of machine learning and is typically the least used. Its main approach is learning how to act or behave given reward or punishment signals. Reinforcement learning will not be used in this paper, so it will therefore not be discussed any further.

2.4. Supervised learning

There are many methods under the branch of supervised learning. Some of the most used methods are: Support Vector Machines, linear regression, logistic regression, linear discriminant analysis, decision trees, k-nearest neighbor algorithm and neural networks [43]. There are also many different variants of neural networks. However, they are all based on the foundation of what is called the Artificial Neural Network (ANN), and the different variants of neural networks have additional features built on top or around the ANN. Since a neural network variant called Convolutional Neural Network has been used in this thesis, it is first explained how an ANN works, followed by an explanation of a CNN.

2.4.1. Artificial Neural Network

An artificial neural network is a computing system which is inspired by the biological neural networks of the brains of humans and animals. Mimicking the neurons and synapses of the brain, the ANN is based on the same model. The ANN consists of layers of connected nodes, where the nodes mimic the neurons of the brain, and the connections between the nodes, called the weights, mimic the synapses. The neural network consists of an input and an output layer, with optional "hidden layers" in between, as shown in figure 2.13. The goal of this neural network is to serve an estimated value from the output layer, based on what is given to the input layer and how the weights between the layers are set.

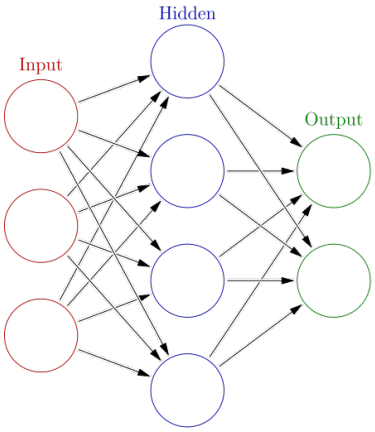


Figure 2.13.: Example architecture of a neural network [6].

One typical example of an ANN, is to predict housing prices. In this case, the input layer would consist of perhaps 5 nodes, where each node could present one feature of the house. E.g., number of bedrooms, square feet of the house, zip code, proximity to schools and proximity to city center. The neural network then predicts the housing price at the output layer based on the weights connected between the layers. This is done via forward propagation, where a linear combination of the nodes is formed based on the size of each individual weight connected to the node. One can either decide the weights between nodes in the layers oneself, or use what is called backpropagation. What makes a neural network a part of machine *learning*, is namely the use of backpropagation such that the neural network optimizes the weights between nodes in the different layers. How a neural network is built and how it works is thus explained more in detail.

Let $\hat{y}$ be the output of a node and x be the input feature. The output of a node in a network is thus given by:

$$\hat{y} = w \times x + b \tag{2.7}$$

Where w is the weight assigned to the node and b is a bias value. As the neural network is in fact a *network*, there are generally several inputs, several nodes and often several layers. Lets first focus on how several inputs affect one node. Now let $\hat{y}$ still be the output of the node, but let $\mathbf{x}$ be an input vector $[x_1, x_2, ..., x_n]$ and $\mathbf{w}$ be a vector of the weights $[w_1, w_2, ..., w_n]$ corresponding respectively to the inputs. Then one has the output $\hat{y}$ of a

single node given as:

$$\hat{y} = \mathbf{w}^T\mathbf{x} + b \tag{2.8}$$

In the case of 3 inputs, this would look like figure 2.14.

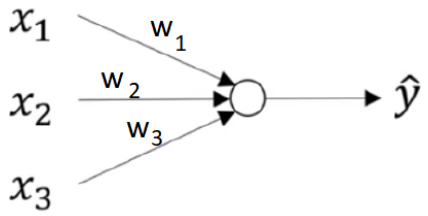


Figure 2.14.: One node with 3 inputs

In the brain, a neuron and its synapses get bigger relative to how much they are used, which also indicates their utility. In a neural network, a node has to achieve a successful result on an *activation function* in order to carry to the next layers, or for the "neuron to fire". This activation function outputs a value a, based on the linear combination of inputs into the activation function. Many different activation functions exist. Two of the most used are either the sigmoid function or the rectified linear unit (ReLU) function. For these activation functions, a threshold value is set such that if $a > threshold$, the node fires to the next layer. If not, the node stays dormant and outputs 0. Before presenting these two activation functions, some slight changes are made to reduce the use of symbols.

Let $\mathbf{z} = \mathbf{w}^T\mathbf{x} + b$, such that $\mathbf{z} = [z_1, z_2, ..., z_n]$ for nodes $[1, 2, ..., n]$ in the same layer. As previously stated, two of the most used activation functions are the sigmoid function, or the ReLU function. They are represented as:

$$\begin{aligned} sigmoid : \sigma(z) &= \frac{1}{1 + e^{-z}} \\ ReLU : R(z) &= max(0, z) \end{aligned} \tag{2.9}$$

A visual representation is shown in figure 2.15, with the sigmoid function on the left and the ReLU function on the right. The threshold for the sigmoid function is typically set as 0.5, where the threshold for ReLU is >0.

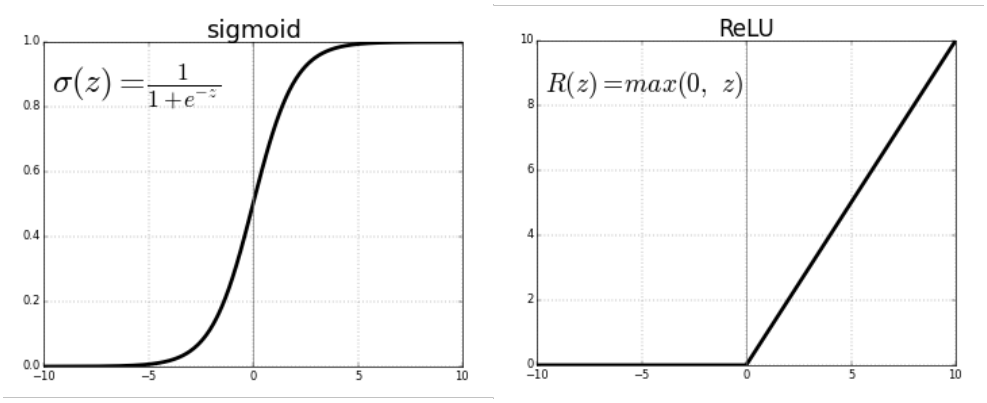


Figure 2.15.: Plots of sigmoid and ReLU function.

Let $a = \sigma(z)$ be the output of the activation function $\sigma(z)$. Now a is the output of node z. When a neural network reaches its end, the output layer, $\hat{y} = a$. For a input layer with 3 inputs, no hidden layer and a single node in the output layer, this would look much like figure 2.14, but to give a visual representation of the new symbols it is shown in figure 2.16.

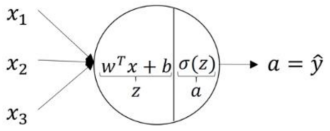


Figure 2.16.: Activation function output [7].

At this point the terminology is almost complete. However, since a neural network has the capability of being several layers deep, annotations is needed to keep track. Let $[l]$ be the annotation for the layer in question, starting at the first hidden layer and ending at

the output layer. The forward propagation of the network is thus described as:

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{w}^{[1]T}\mathbf{x} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= \sigma(\mathbf{z}^{[1]}), \\
\mathbf{z}^{[2]} &= \mathbf{w}^{[2]T}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \\
\mathbf{a}^{[2]} &= \sigma(\mathbf{z}^{[2]}), \\
&... \\
\mathbf{z}^{[l]} &= \mathbf{w}^{[l]T}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \\
\hat{y} &= \sigma(\mathbf{z}^{[l]})
\end{aligned} \tag{2.10}$$

A visual example of this is shown in figure 2.17, with one hidden layer.

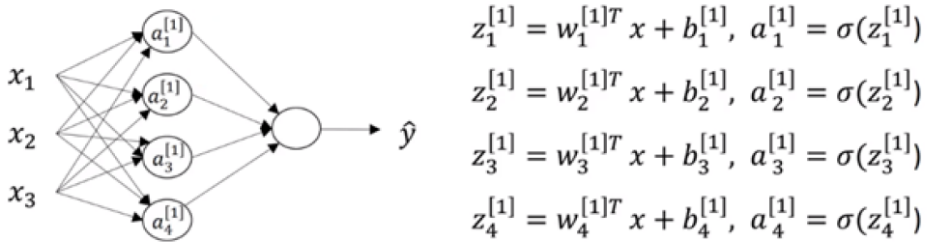


Figure 2.17.: A neural network with one hidden layer [7].

Until this point, it has only been shown how to calculate the forward pass of the neural network. What makes the neural network able to "learn" and train, is the feat of back-propagation. First, an error function needs to be introduced in order to evaluate how well the predicted outcome $\hat{y}$ scored relative to the true label y. This is also called the loss function, or the cost function. One of the most common error functions is the logistic loss:

$$L = (y \times log(\hat{y}) + (1 - y) \times log(1 - \hat{y})) \tag{2.11}$$

The goal of backpropagaton and training of a neural network is to minimize the error function. This is done by finding the weights and biases which minimize the error function. Finding this is done by what is called **gradient descent**. Let $E(w, b)$ denote the error function with regards to the weights and biases. When minimizing the error function, the aim is to find the minimum of said function. This is done by finding the steepest gradients

for w and b through partial derivation of the error function $E(w, b)$ at each layer. The goal is to update the parameters w and b through iterations, in order to find the best fitting weights and biases that minimize the error function, thus increasing the correctness of the prediction of the neural network. This is done by implementing something called *learning rate* (LR), which constitutes how large of a step down the direction of the gradients of the parameters to take. Thus, parameters w and b are updated as:

$$\begin{aligned} w_i^{[l]} &= w_i^{[l]} - LR \times \Big(\frac{\partial E(w,b)}{\partial w}\Big)_i^{[l]} \\ b_i^{[l]} &= b_i^{[l]} - LR \times \Big(\frac{\partial E(w,b)}{\partial b}\Big)_i^{[l]} \end{aligned} \tag{2.12}$$

This is calculated iteratively backwards through the network, utilizing the chain rule when calculating towards the beginning of the network. It is important to note that the learning step should not be designed to be too high, such that it overreaches the minima, nor too low such that the learning of the neural network is slow and takes too long. When starting the neural network, the weights and biases needs to be initialized. This is typically done by the weights being assigned random values, and the biases set to zero.

2.4.2. Convolutional Neural Network

The theory and examples of the ANNs presented in the previous section, were what is called a "fully connected neural network". This means that every node in each layer are all connected to every node in the next layer. A digital image is a matrix of pixel intensities, or 3 matrices in the case of a colored image, each matrix corresponding to one of the basic colors. If a fully connected neural network were to be implemented on a raw image, each pixel would correspond to one node. Depending on the resolution of the image, the amount of parameters needed to be calculated quickly become immense. Taking the example of a small grayscale image with a resolution of only 32×32 pixels, the input feature vector would contain 1024 elements. If the hidden layer were to have e.g. 7200 nodes, one would end up with 7,372,800 distinct parameters. One could argue that reducing the number of neurons in the hidden layer would help with reducing the number of distinct parameters. However, this might adversely affect the performance of classification. Therefore, it is common to keep the number of neurons in the first hidden layer high [44].

In order to address the parameters issue, one could hypothetically rearrange the neurons in the hidden layer into blocks of neurons. Assume, instead of a layer containing 7200×1 neurons, one could rearrange it into 50 blocks of 12×12 neurons. One then assumes that the pixels that are in proximity to one another are highly correlated, and pixels far away are not. E.g. that pixel (2,2) is more correlated to pixel (0,0) than pixel (32,32). Assume that neuron (0,0) in each block is intended to extract information around pixel (3,3) Likewise, neuron (11,11) in all blocks are intended to extract information from pixel (29,29) in the image. Due to the low correlation between pixels far away from each other, neuron (0,0) only needs information from pixel (3,3) and its surrounding pixels to gain information from this region. Thus, one can connect each neuron in each block to a certain region on the image, and extract information by applying a filter over said region. The filters would be convolutional filters/kernels explained in section 2.1.1, and operate in the same manner. However, the filters may change, depending on what feautre the CNN is programmed to detect. If the filter was set to be 5×5, the number of parameters would be reduced to $5 \times 5 \times 50 \times 12 \times 12 = 180,000$, thus obtaining a substantial reduction of parameters needed compared to the fully connected layer. To further reduce parameters, one could assume that all neurons in the same block share the same weights, which is called *weight sharing*. Thus, one is down to $5 \times 5 \times 50 = 1250$ weights between the hidden layer and the image, giving a 99.98% reduction in parameters compared to the fully connected layer, in this specific example.

The objective of the convolution layer is to extract the high-level features such as edges, colors, gradient orientation, etc., from the input image. The next layer is the pooling layer, which is responsible for reducing the spatial size of the convolved features. This also is done to decrease the computational load for processing the neural network. There are two types of pooling: Max pooling and average pooling. Max pooling returns the maximum value of the region that the kernel is covering, where the average pooling returns the average value of said region. Max pooling acts as a noise suppressant, as it extracts the most extreme features, and is therefore the most commonly used [9].

A deep CNN can have several convolution and pooling layers in order to accentuate features and gain accuracy. However, this might come at a cost. "Overtraining" the neural network can make it perform better on the training data, but might end up giving poorer

results on the test data, as the model is *overfitting* during training. Finally, one or more fully connected layers is added to learn non-linear combinations of the high-level features from the previous layers and classify at the end. An example of a CNN can be shown in figure 2.18, with two convolution and max pooling layers, one fully connected activation layer and one last fully connected classification layer.

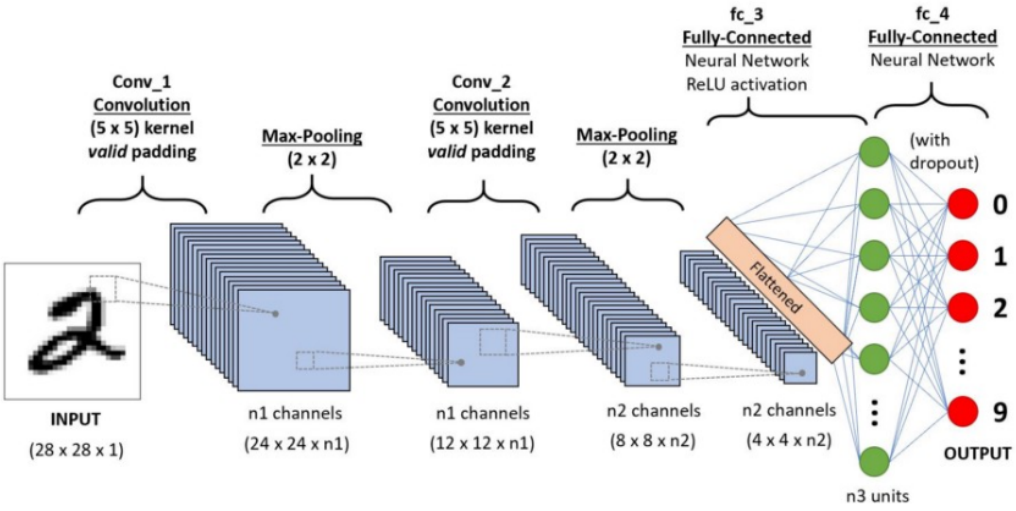


Figure 2.18.: Example of a CNN structure [8].

2.4.3. Inception V3

Inception V3 is a CNN designed by Google Brain Team, which is a machine intelligence team focused on deep learning [45]. Inception's purpose was to challenge the idea that just stacking convolution layers deeper and deeper would improve performance. As previously explained, just adding convolution layers is prone to overfitting and becomes increasingly computationally expensive for every layer added. The solution implemented in Inception was to utilize multiple filters of different sizes at the same level, thus going "wider" rather than deeper [9]. As shown in figure 2.19, three different filters and max pooling is performed on the same layer, concatenating the result at the end of the layer.

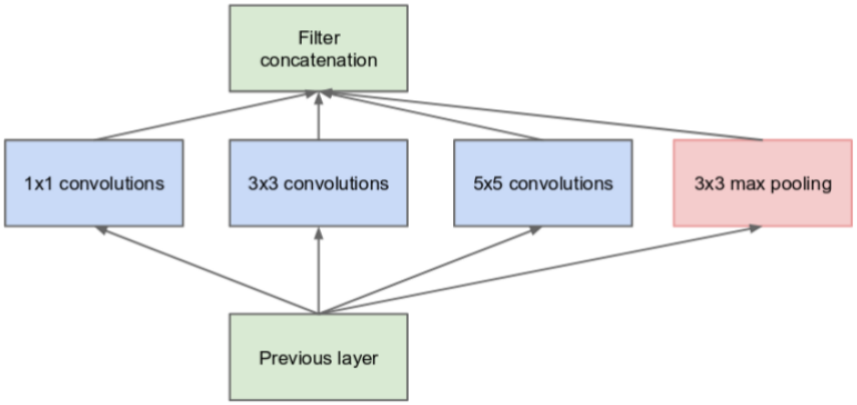


Figure 2.19.: First Inception module [9].

This was the filter configuration for the first version of Inception - Inception V1. When developing this CNN further, it was decided that factorizing a 5×5 convolution to two 3×3 convolution operations would improve computational speed, as a 5×5 convolution is 2.78 times more expensive than a 3×3 convolution [9]. In fact, a reduction in computational time is found for any filter of size $n \times n$ when reduced to a combination of $1 \times n$ and $n \times 1$ convolutions. A combination of a 1×3 convolution followed by a 3×1 convolution was found to be 33% cheaper than a 3×3 convolution. Thus, a new convolution layer was implemented for Inception V2 and V3[9], shown in figure 2.20.

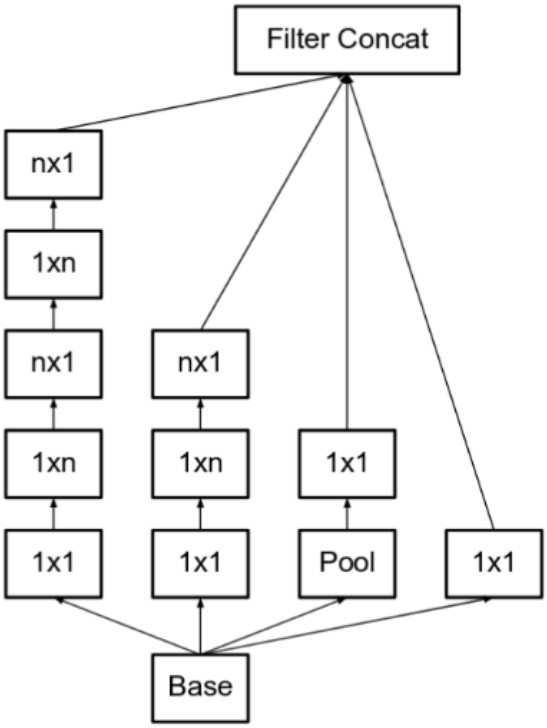


Figure 2.20.: First Inception module [9].

At the end of the network, the classification layer consist of the *softmax function*. The softmax function takes the output vector from the previous layer, and normalizes it into a probability distribution. This is done such that in case some vector elements from the output vector from the previous layer were negative, these would now become non-negative, and all elements are normalized to the range between 0 and 1. Thus, the non-normalized output of the previous layer is transformed to a probability distribution over the predicted output classes. The softmax function is given for a K-dimensional vector $\mathbf{z}$ as:

$$\begin{aligned} \sigma : \mathbb{R}^K \rightarrow \left\{ \sigma \in \mathbb{R}^K | \sigma_i > 0, \sum_{i=1}^{K} \sigma_i = 1 \right\}, \ \ where \\ \sigma(\mathbf{z}_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \ \ for \ \ j = 1, ..., K. \end{aligned} \tag{2.13}$$

At last, one can see the 23-layer deep structure of Inception V3's CNN, shown in figure 2.21.

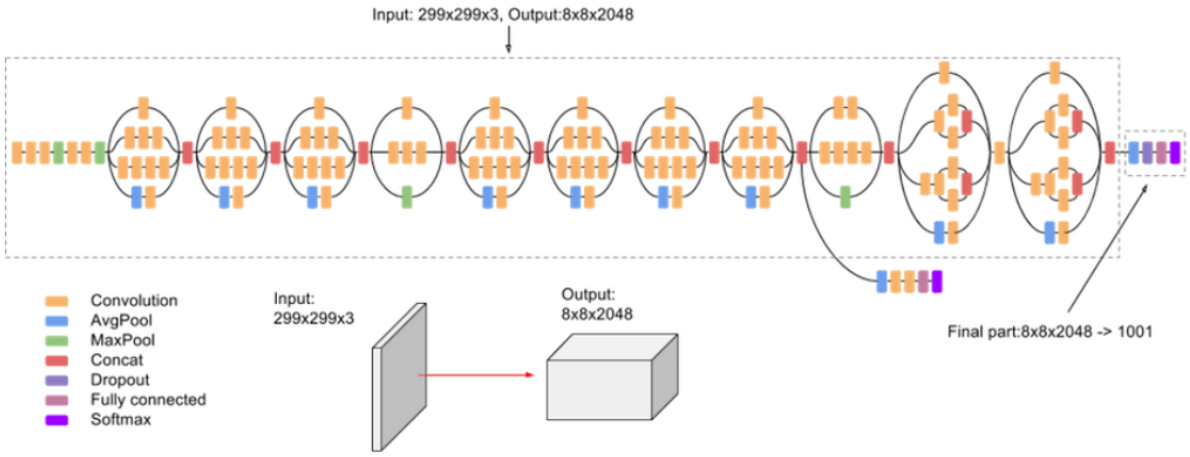


Figure 2.21.: Complete structure of Inception V3 [10].

2.5. Unsupervised learning

The purpose of unsupervised learning, or perhaps more suitably called cluster analysis, is to divide data into groups that form some meaning or utility for the interpreter. The interpreter could be a person looking at the data, or another program using the clustered data as classification for further processing or modeling. It has been widely used in many fields, such as biology, psychology or other social sciences. However, the biggest rise in usage has been in the later years within the fields of machine learning, pattern recognition, data mining, and statistics [46].

Cluster analysis is the study of techniques in which the goal is finding the most representative cluster prototypes. Data is grouped based only on information found in the data, which describes the objects and their similarities. The goal is for data to be grouped up with as much similarity (often proximity) to similar objects, meaning it adheres to the same group, and as much dissimilarity to other groups/clusters.

In figure 2.22, one is visually introduced to the results of clustering with different amount of clusters chosen. there are 20 data points more or less scattered on a plot. One can loosely see that the data points tend to appear in two groups. A good estimation would then be to assign a clustering algorithm with two clusters. This can be seen in (b) in the figure. However, when one decides to implement a clustering algorithm, one is often required to

a priori set the number of clusters one wants. For example in the algorithm "K-Means", which will be addressed later, one is required to specify the number of clusters for the dataset. One can then see from the figure that (c), with four clusters, or (d), with six clusters, tend to make small clusters right next to each other. At some point the amount of clusters will group data with such close proximity to each other that one cannot with good conscience state that these data belong in different groups. This is addressed by the use of cluster validity, which is explained later in this chapter.

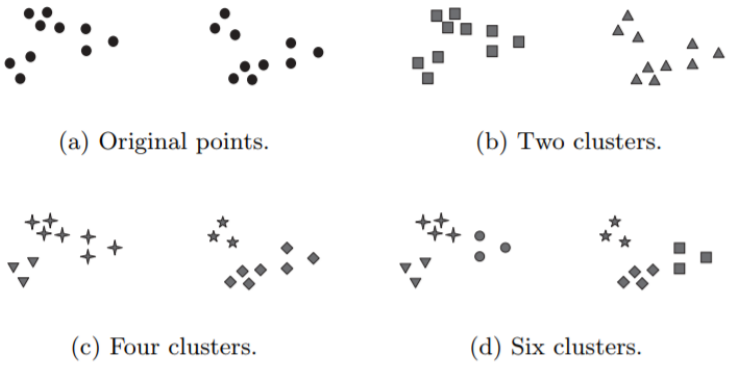


Figure 2.22.: Varying amounts of clusters [11]

2.5.1. Types of clustering

It is commonly said to be main two branches within clustering applications [28]. These are called *hierarchical* (flat/nested) clustering and *partitioned* (unnested) clustering. **A partitioned clustering** is simply the division of data objects into groups (clusters) which are non-overlapping. Figure 2.22 was an example of this. Thus, it is important that each data point adheres to one, and only one, cluster. With **hierarchical clustering**, each data point is iteratively branched together with its closest, or most similar, data point. It is typically displayed in what is called a cluster dendrogram, as shown in figure 2.23. Here one can see sub-clusters and their relation.

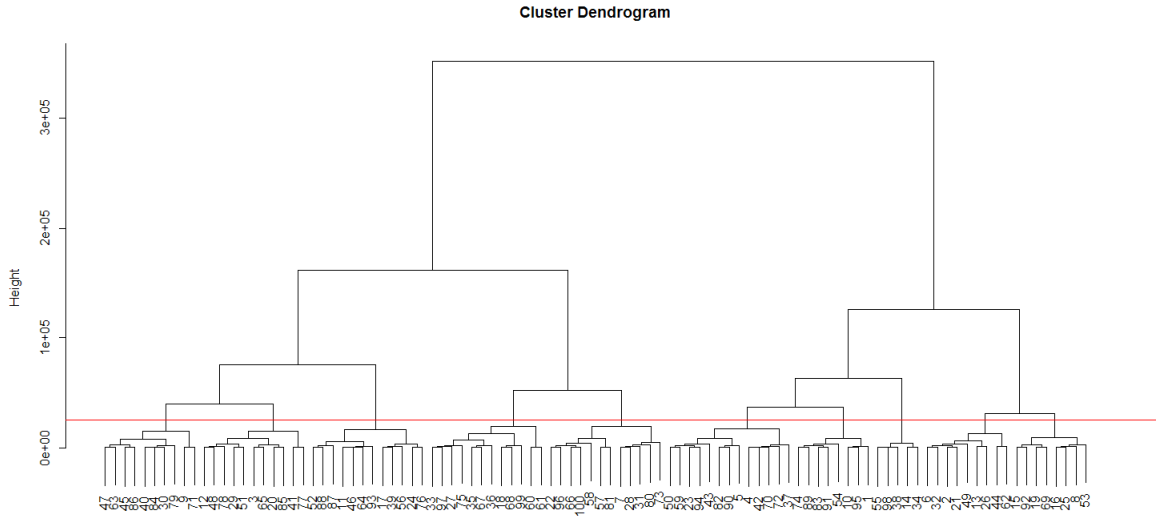


Figure 2.23.: Cluster dendrogram [12]

Exclusive vs Overlapping vs Fuzzy

Data points in clusters are typically defined within one of three states. Exclusive, overlapping or fuzzy. When a data point is exclusively assigned to a cluster, it is only a member of that cluster. When a cluster is overlapping, or non-exclusive, a data point can belong to several clusters. An example of this could be that a student belongs to both a cluster of enrolled students, but also to a cluster of employees, given he for example works as a student assistant. In a 2D-graph this would typically be displayed as a point somewhere between the center of the two clusters. Fuzzy clustering is similar to non-exclusive clustering. However, each data point is given a weight between 0 and 1 of how much it belongs to each given cluster. A constraint for ensuring that the clustering is coherent, is that the sum of weights for each data point must equal 1.

2.5.2. Similarity and dissimilarity

In cluster analysis the inputs differ based on what type of analysis is to be done. As partitioned clustering is a similarity-based clustering one uses what is called a $N \times N$ dissimilarity matrix, or distance matrix as means of calculating the similarity or dissimilarity between points. In hierarchical clustering one uses a $N \times D$ feature matrix, or design ma-

trix. Similarity and dissimilarity with regards to distance measure, are just the opposites of each other. The closer one point is to another, the more similar they are. The further they are apart, the more dissimilar. Since this thesis focuses on the use of partitioned clustering, some dissimilarity measures are introduced.

Measuring similarity/dissimilarity

A dissimilarity matrix $\mathbf{D}$ is a matrix where $d_{i,i} = 0$ and $d_{i,j} \geq 0$ is a measure of the "distance" between objects i and j.

The mathematical definition of dissimilarity between objects in terms of the dissimilarity of their attributes are:

$$\Delta(\mathbf{X}_i, \mathbf{X}_{i'}) = \sum_{j=i}^{D} \Delta_j(x_{ij}, x_{i'j}) \tag{2.14}$$

where some common dissimilarity functions are:

Squared (Euclidean) distance

$$\Delta_j(x_{ij}, x_{i'j}) = (x_{ij} - x_{i'j})^2 \tag{2.15}$$

This is the squared distance of the line between two points in euclidean space. Squared distance strongly emphasizes large differences, due to its exponential nature. One can also apply cubed distance or bigger exponentials to affect the sensitivity of distance between points even further. However, squared distance is most often sufficient and is the dissimilarity measure most often used.

City block distance

$$\Delta_j(x_{ij}, x_{i'j}) = |(x_{ij} - x_{i'j})| \tag{2.16}$$

City block distance is also called l_1 distance or Manhattan distance, and is found by computing how many rows and columns one have to move in order to get from x_i to $x_{i'}$, like maneuvering from one city block to another.

Correlation coefficient

if $\mathbf{X}_i$ is a vector, typically from a time-series of real-valued data, the correlation coefficient is commonly used. However, if the data is standardized, one has that

$$corr[\mathbf{X}_i, \mathbf{X}_{i'}] = \sum_j x_{ij} x_{i'j}. \tag{2.17}$$

Through a series of calculation-steps and similarities, as shown in [47], one arrives at

$$\sum (x_{ij} - x_{i'j})^2 = 2(1 - corr[\mathbf{X}_i, \mathbf{X}_{i'}]). \tag{2.18}$$

Therefore, as one can see, clustering based on correlation of standardized data is equal to clustering based on the squared distance [47].

Categorical variables

In cases where categorical values are measured, like grouping cars by their color, one can assign a distance 1 if the features are different, and conversely 0 if not. The sum of all the categorical then becomes:

$$\Delta(\mathbf{X}_i, \mathbf{X}_{i'}) = \sum_{j=1}^{D} (x_{ij} \neq x_{i'j}) \tag{2.19}$$

This is also known as the Hamming distance.

Other measuring methods worth mentioning could be the Jaccard Similarity or the Cosine Correlation, which are widely used in item similarity measurements of recommendation systems. An example of this could be an online movie- and TV-show provider trying to best recommend new and/or similar items to any given user. However, the similarity measures already stated are the ones most used in clustering problems and will thus be used in this thesis.

2.6. Partitioned clusters

2.6.1. K-Means algorithm

For unsupervised learning, the K-means algorithm is one of the most known and used clustering algorithms [13]. The K-means algorithm clusters an unlabeled data set into K

clusters, where K is chosen by the programmer. The idea is to map a series of points into a group or cluster, based on how effectively the points are assigned to each cluster. The K-means algorithm operates such that each point is assigned its cluster based on the shortest path to the closest cluster, using squared distance as the distance measure. The steps of the algorithm can be explained as follows. Take an unlabeled data set, and randomly assign the K chosen cluster centroids amongst the data. Then calculate the squared distance of each data point in the data set and assign membership to the closest cluster for each point. Then take the mean of all the points in each cluster, and assign that as the new center point for the cluster centroid. Then repeat the cluster assignment and cluster centroid alignment until the cluster centroids movement converge, ultimately stopping.

This is represented mathematically as follows. Let

$$X = \{x_1, x_2, ..., x_n\} \in \mathbb{R}^n \tag{2.20}$$

be a $D \times n$ matrix, where D is the number of points in the data set and n is the number of variables, or dimensions, of the data set to be investigated. Thus, let

$$C = \{c_k, k = 1, ..., K\} \in \mathbb{R}^k \tag{2.21}$$

be the set of indexes for which cluster each example x_i is assigned. An example of this could be that x_3 is assigned to cluster $k = 2$, such that $c_3 = 2$.

Let μ_k be the mean of cluster c_k. The squared error between x_i and μ_k for cluster c_k is found through the objective function $J(c_k)$.

$$J(c_k) = \sum_{x_i \in c_k} ||x_i - \mu_k||^2 \tag{2.22}$$

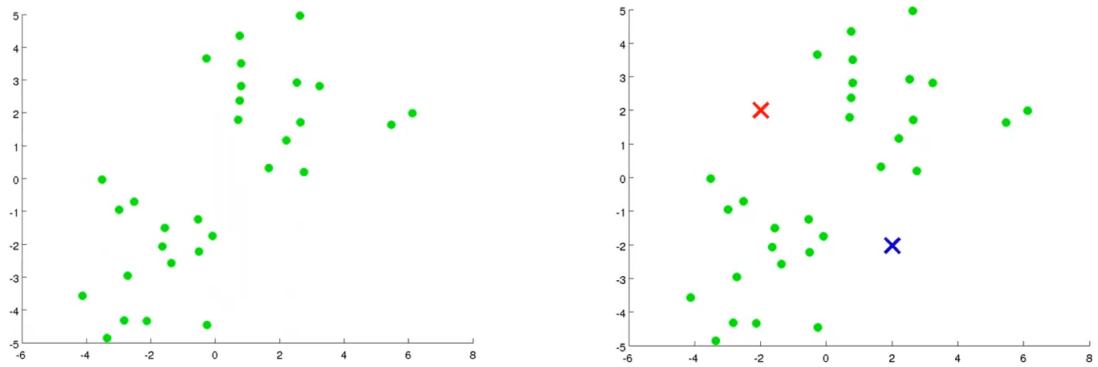
The goal of K-means is to minimize the sum of the squared error over *all* K clusters. The overall objective function thus becomes

$$J(C) = \sum_{k=1}^{K} \sum_{x_i \in c_k} ||x_i - \mu_k||^2 \tag{2.23}$$

And is minimized through

$$min_{c_1,...,c_k,\mu_1,...,\mu_k} J(C). \tag{2.24}$$

Even though the K-means algorithm is now described verbally and mathematically, it is best accompanied by an example visually. Shown in the same steps as written on the beginning of this section, figure 2.24 displays an unlabeled data set and the random initialization of K=2 clusters.

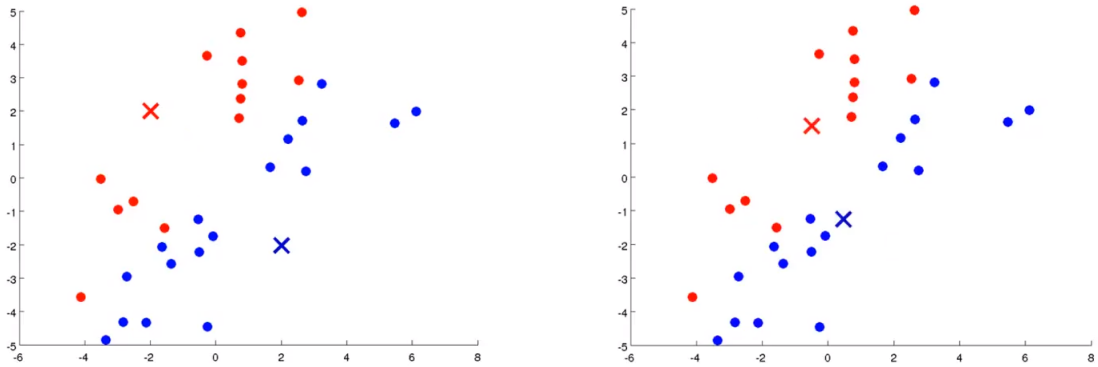


(a) Unlabeled data set

(b) K=2 random initialization

Figure 2.24.: K-means initialization [13].

Then the cluster assignment step is performer, where membership to each cluster is assigned through minimizing distance to each cluster, as shown in 2.25a. Thus, the new cluster centroid is calculated and assigned, as shown in 2.25b.



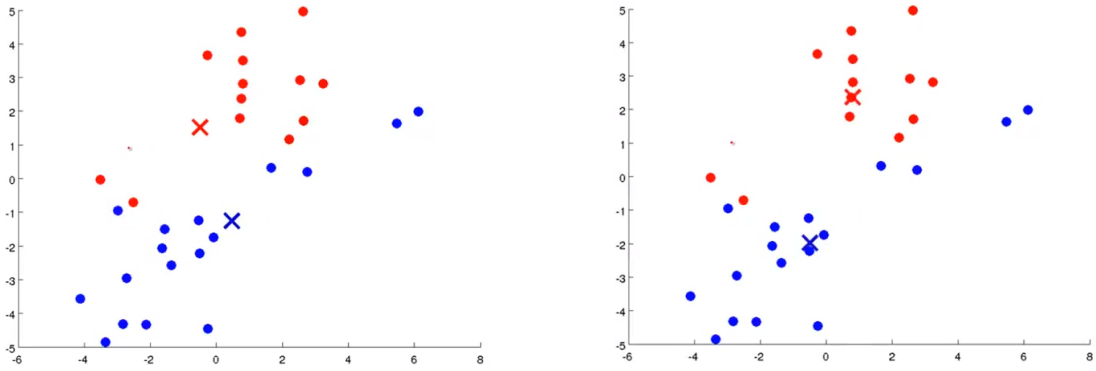
(a) Cluster assignment

(b) Centroid adjustment

Figure 2.25.: First round of K-means [13].

Now the algorithm is in its iterative state, and 4.3 shows the second step where the same

action as in 2.25 is performed.



(a) Cluster assignment (b) Centroid adjustment

Figure 2.26.: The two steps of one iteration of K-means [13].

At last one arrives at the end when no new points are assigned to the cluster and thus the cluster centroid won't change, as seen in 2.27.

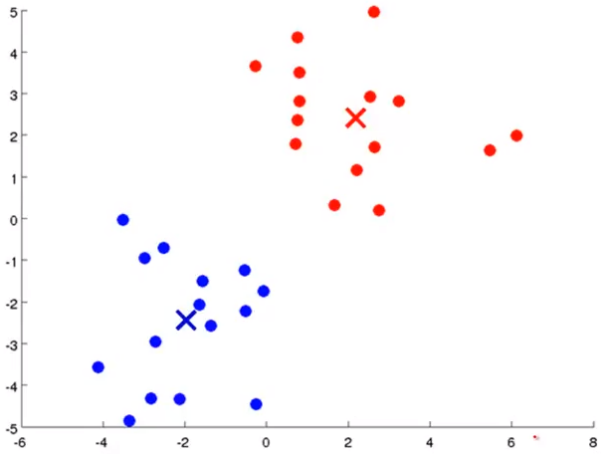


Figure 2.27.: Completed algorithm [13].

As explained in the beginning of this chapter, choosing the right K number of clusters can be hard. Unlike supervised learning, one has no measure of accuracy of an unlabeled data set. However, what is being done, is minimizing the squared error between points and their assigned cluster. Instead of measuring accuracy, the correlation between the data points and cluster centroids can be measured in various manners. The most used of

these are either the Elbow method, the Silhouette method or the Gap statistic method [15], which is presented later in this chapter.

2.6.2. Affinity propagation

Deciding the number of clusters can be a difficult task if one has little knowledge or understanding of data to be investigated. As a consequence of this, methods have been proposed to have the number of clusters figured out autonomously by the computer. Affinity Propagation is one of these proposed solutions. It shares some similarities with K-Means, but is structured quite differently.

In Affinity Propagation, the data points can be thought of as all being in a network, where each individual point communicates with each other. The purpose of this communication is to decide which data point is going to act as an *exemplar*, the center point of a cluster. Every data point is therefore collectively determining which data point should act as exemplar through each iteration of the algorithm [14]. The message passing between points are stored in two matrices. The *responsibility matrix* and the *availability matrix*. The responsibility matrix R reflects how well-suited a point k is to be an exemplar for point i through $r(i, k)$. The availability matrix A reflects how appropriate it would be for a point i to choose point k as its exemplar through $a(i, k)$. The message passing of availability and responsibility is visually represented in figure 2.28.

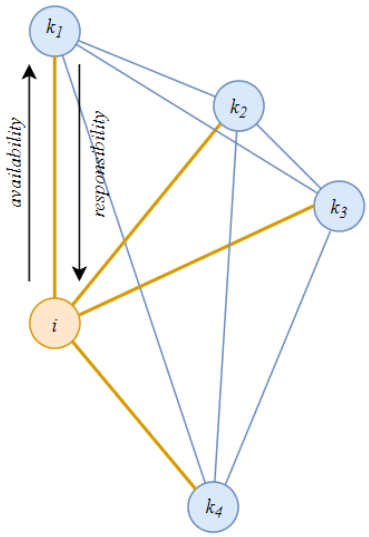


Figure 2.28.: Visual representation of message passing between points[14].

Similarity

The first messages sent per iteration are the responsibilities. These values are based on a similarity function s. Affinity propagation uses the negative squared distance as similarity measure. This is then stored in the similarity matrix S.

$$s(i,k) = -||x_i - x_k||^2 \in S \tag{2.25}$$

Responsibility

The responsibility messages are defined as:

$$r(i,k) \leftarrow s(i.k) - max_{k's.t.k' \neq k}\{a(i,k') + s(i,k')\} \tag{2.26}$$

this can be implemented through a nested for loop where one iterates over every row i and determines the $max(A+S)$ for every index not equal to i or k.

Availability

For all points which isn't on the diagonal of A, that being all messages going from one data point to all others, the update equals the responsibility that point k assigns to itself

and the sums of responsibilities that all other data points assign to k.

$$a(i,k) \leftarrow min\{0, r(k,k) + \sum_{i's.t.i' \notin \{i,k\}} max\{0, r(i',k)\} \tag{2.27}$$

For the points on the diagonal of matrix A, which are the availability values a that a data point sends to themselves, the message values equals to the sum of all positive responsibility values sent to the current data point.

$$a(k,k) \leftarrow \sum_{i' \neq k} max\{0, r(i',k)\} \tag{2.28}$$

Exemplars

The final exemplars are then chosen to be the maximum value of $A + B$.

$$exemplar(i,k) = max\{a(i',k) + b(i',k)\} \tag{2.29}$$

Clustering

In order to commence the clustering of the data, the implementer needs to decide the number of iterations, a damping factor and the preference. The preference is set to the diagonal of the matrix S. The preference value indicates how strongly a data point thinks of itself as a potential exemplar. If this value is initialized at 0 and remains unmodified, not clustering will occur as every data point believes itself should be an exemplar. One common solution is to set the preference to the median of S. However, this opens up for a propensity to split clusters as the iteration continues for long. If this is not wanted, increasing the preference to a higher negative value than the median(S) will often help.

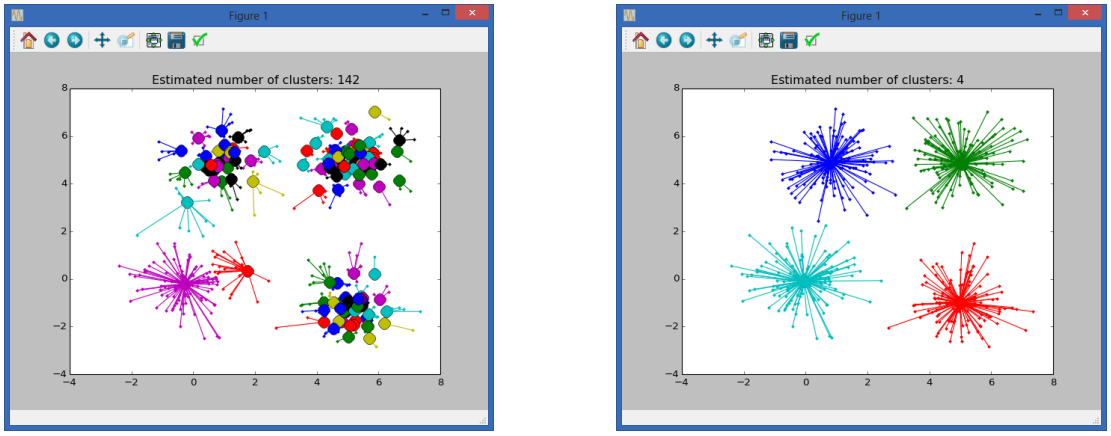


Figure 2.29.: Reduction of clusters through iterations [14].

Due to the nested for-loop implementation of the Affinity Propagation, it has a complexity of $O(n^2)$, such that the increase of data points greatly affect computational time.

2.7. Cluster validity

When working with unlabeled data and little to no information as to how the data might best be grouped, testing of *cluster validity* is performed. If the number of clusters is not directly relevant to the task at hand, cluster validity methods aims to find the amount of clusters that describes the best fit for the data investigated. Following are some validity methods for partitioned clusters.

2.7.1. Elbow method

The goal of the K-means method was to minimize the squared error between points and their assigned cluster. The elbow method is a means of looking at the total sum of squared error relative to the number of clusters K. As one can see in 2.30, the sum of squared errors decreases as the number of clusters increases. What the elbow method aims to describe, is that when there is a significant bend from vertical to horizontal on the graph (like the elbow of an arm), this might represent the correct number of clusters for the given data set. This is because at this point, there has been significant drops in squared errors for the clusters. But, after this point, it can seem unreasonable that it is a good choice to further divide data that are already in close proximity to each other in new clusters. An example

is shown in figure 2.30, where the elbow is thought to be at $k = 4$ clusters.

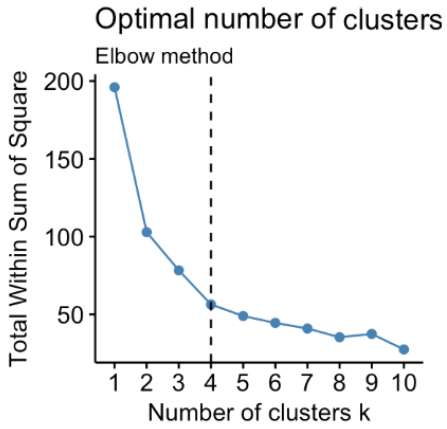
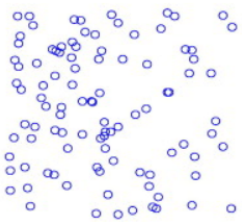
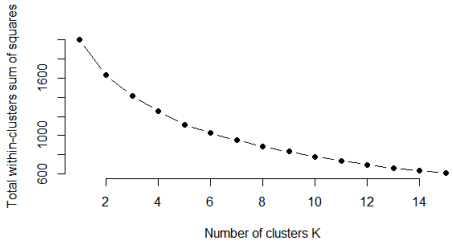


Figure 2.30.: Example of Elbow method dictates k=4 clusters [15].

The Elbow method will however not always display an elbow in the graph. If one has a data set with uniformly dispersed points as in 2.31a, it is more likely for the Elbow method graph to display itself as in 2.31b. Here one can see that there is no well-defined elbow, and thus it is hard to justify which, if any, amount of clusters will successfully divide the data in any sensible manner.



(a) Disperse points



(b) Elbow method without an elbow

Figure 2.31.: Bad case for Elbow method [15].

2.7.2. Silhouette method

The Silhouette method is another method to measure the quality of clustering. It determines how well each data point lies within its own cluster. A good score is indicated by a

high average silhouette score. A silhouette value is a measure of how similar a data point is to its own cluster, called cohesion, compared to the other clusters, namely separation. The value ranges from -1 to +1, where a high value indicates that the data point is a good fit for its own cluster and a poor fit for any of the other possible clusters. Thus, a high average silhouette score signifies that all the points are well fitted for their clusters and a bad fit for other clusters.

Mathematically, the Silhouette method can be described as

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}} \tag{2.30}$$

Where $a(i)$ is the average distance between point i and all other data within the same cluster and $b(i)$ is the smallest average distance of point i to all points in any other cluster which point i do not contain a membership. Written a bit more algorithmicly minded, one has:

$$s(i) = \left\{ \begin{array}{cc} \text{1-a(i)/b(i),} & \text{a(i) <b(i)} \\ \text{0,} & \text{a(i)=b(i)} \\ \text{b(i)/a(i)-1,} & \text{a(i) >b(i)} \end{array} \right\}, for -1 \leq s(i) \leq 1$$

Figure 2.32 is an illustration of how one might pick the correct number of clusters based on the silhouette method.

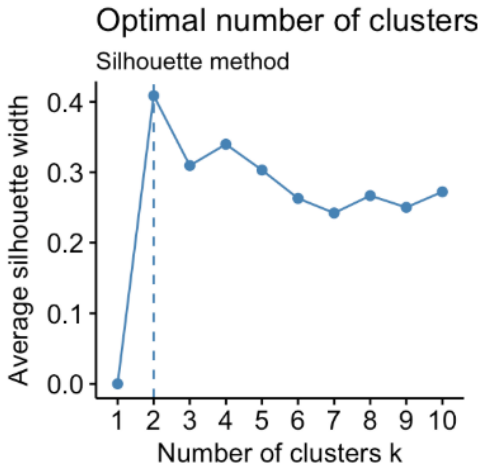


Figure 2.32.: Silhouette method [15].

2.7.3. Gap statistic

The idea of the gap statistic is to take the total within intra-cluster variation for different values of k and compare them with their expected values under a null reference distribution of the data. The estimate of the optimal clusters will then be the value for which $logW_k$ falls the farthest below the reference curve, thus maximizing the gap statistic [48]. This means that the variation within clusters is much smaller than the randomly distributed points of the reference distribution.

Let D_k represents the sum of intra-cluster distances between the points in a given cluster C_k, containing n_k points.

$$\begin{aligned} D_k &= \sum_{x_i \in C_k} \sum_{x_j \in C_k} ||x_i - x_j||^2 \\ &= 2n_k \sum_{x_i \in C_k} ||x_i - \mu_k||^2 \end{aligned} \tag{2.31}$$

Adding up the normalized intra-cluster sum of squares then gives the measure of the clusters compactness, called W_k.

$$W_k = \sum_{k=1}^{K} \frac{1}{2n_k} D_k \tag{2.32}$$

Plotting the difference in W_k for a different number of clusters would then give the plot that is used for the elbow method. However, the gap statistic takes the log of W_k and compares it with a null reference distribution of the data. In other words, a distribution with no obvious clustering. The estimate for the optimal number of clusters K is then the value for which difference between $E_n^*\{logW_k\}$, being the null reference distribution compactness, and $logW_k$ is the largest. This is formalized as

$$Gap_n(k) = E_n^*\{logW_k\} - logW_k \tag{2.33}$$

The reference datasets are often generated by sampling uniformly from the original dataset's bounding box, as seen in figure 2.33a, top right picture. To obtain the estimate $E_n^*\{logW_k\}$, one needs to compute the average of B copies $logW_k^*$ for $B = 10$. Each of these are created with a Monte Carlo sample from the reference distribution, with a

standard deviation $sd(k)$. When accounting for the simulation error, one gets the quantity s_k, represented as:

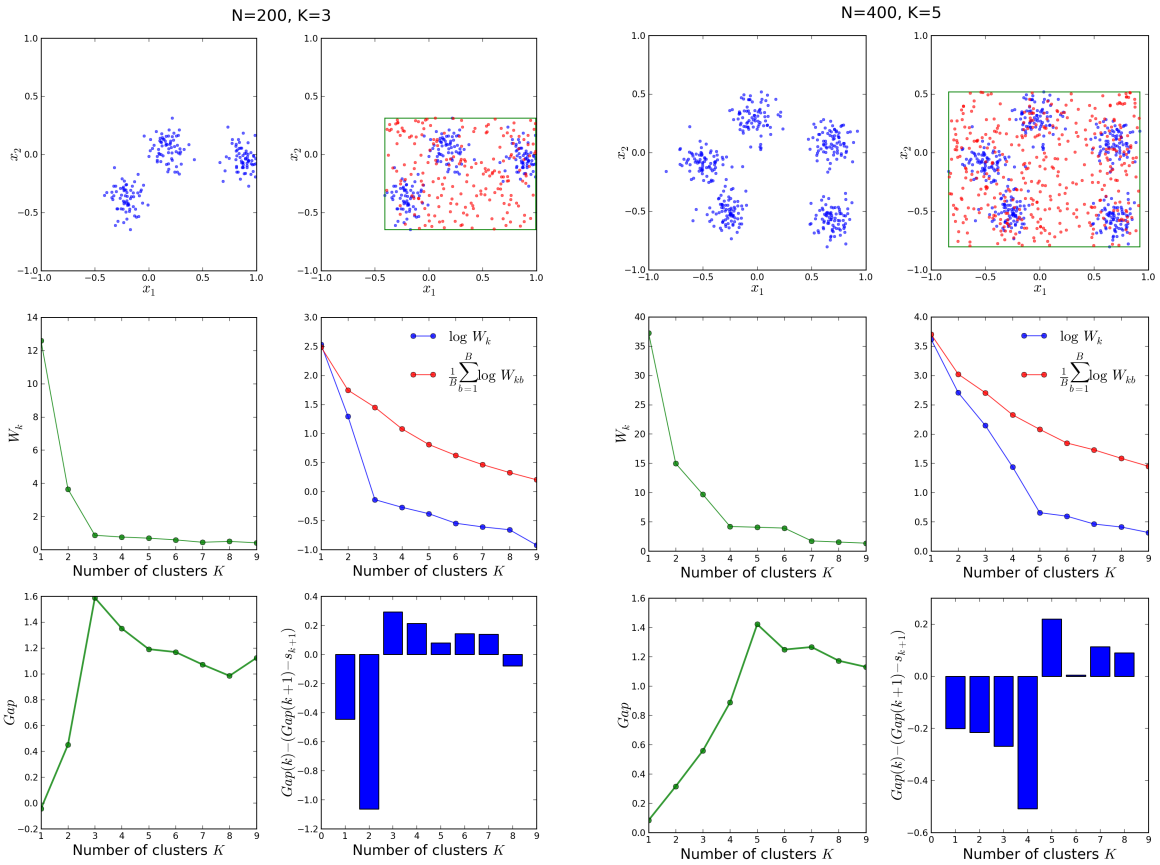
$$s_k = \sqrt{1 + 1/B} \times sd(k) \tag{2.34}$$

Finally, the optimal number of K clusters is the smallest k that fulfills

$$Gap(k) \geq Gap(k+1) - s_{k+1}. \tag{2.35}$$

As this is quite abstract and not so straight forward to understand through mere reading, some following figures will hopefully guide the reader towards better understanding. In figure 2.33a one can see three clusters in the top left plot. The top right plot takes the boundary around around the dataset, and creates a null reference distribution. In the middle left one can see the cluster compactness relative to the number of clusters K, which is the visual aid one uses when determining the elbow method. The middle right takes the logarithm and average logarithm of the compactness measure relative to clusters K. The bottom left plot thus shows the gap, $Gap_n(k) = E_n^*\{logW_k\} - logW_k$ for each cluster K. A high score in this measure indicates probable optimums for the amount of clusters K. Finally, in the lower right plot the entire gap statistic, $Gap(k) \geq Gap(k+1) - s_{k+1}.$, is measured for amount of clusters K. Here one can see that Gap(3) at K = 3 clusters is the one to fulfill the equation.

A similar example is seen in figure 2.33b, with 400 data points and 5 clusters. It shows similar results, and seems to work well.



(a) N=200, K=3

(b) N=400, K=5

Figure 2.33.: Gap Statistic examples [16].

2.8. Dimensionality reduction

Dimensionality reduction is used in an array of applications, and by almost all scientific disciplines [29]. Working with data in high-dimensional space can present various problems. These could be computational power/time, visualization of data or wanting to figure out the similarity in a large amount of interdepentant variables. A possible solution to this is the use of dimensionality reduction. Reducing the amount of data to a smaller subspace, while still containing a big part of the data's variance can be done through several dimensionality reduction methods. Some of the most common methods are Decision Trees, Factor Analysis or Principal Component Analysis [49]. In this thesis, Principal Component Analysis (PCA) is used to reduce the data to the 2-dimensional plane in order to visualize spread of the

data being investigated, and to see whether they group up in a meaningful manner. More of this will be explained later, but first it is explained how principal component analysis is calculated and used.

2.8.1. Principal Component Analysis

The goal of principal component analysis (PCA) is to maximize the variance of a linear combination of the variables, such that one can determine which variables represents most of the variation in a data set. If one has a large set of variables, each representing a dimension in the data, the PCA maps the projection of the points from the old variables to a best fit line, called a PCA-dimension. This is one of the main methods for dimensionality reduction, and can be implemented in various ways. It is usually explained via an eigen-decomposition of the covariance matrix of the data, but Singular Value Decomposition (SVD) can also be used. Since this thesis uses an open software application of PCA by means of SVD, an account of how PCA is obtained via SVD is described as follows.

Let $\boldsymbol{X}$ be the data matrix of size $n \times p$, where n is the number of samples and p is the number of variables. It is centered by having column means subtracted. The $p \times p$ covariance matrix $\boldsymbol{C}$ is given by

$$\boldsymbol{C} = \frac{\boldsymbol{X}^T \boldsymbol{X}}{n-1}. \tag{2.36}$$

It is a symmetric matrix and can thus be diagonalized:

$$\boldsymbol{C} = \boldsymbol{V}\boldsymbol{L}\boldsymbol{V}^T \tag{2.37}$$

where $\boldsymbol{V}$ is a matrix of eigenvectors and $\boldsymbol{L}$ is a diagonal matrix with eigenvalues λ_i in decreasing order on the diagonal. These eigenvectors are the principal axes, or principal directions, of the data. The projections of the data onto the principal axes are the principal components. The $j-th$ principal component is given by the $j-th$ column of $\boldsymbol{XV}$, and the coordinates of the $i-th$ data point in the new principal component-space are given by the $i-th$ row of $\boldsymbol{XV}$. Thus singular value decomposition can be performed on $\boldsymbol{X}$ to obtain the decomposition:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{2.38}$$

where $\boldsymbol{U}$ is the unitary matrix, and $\boldsymbol{\Sigma}$ is the diagonal matrix of singular values s_i. From this, one gets:

$$\begin{aligned} \boldsymbol{C} &= \frac{\boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T}{n-1} \\ &= \boldsymbol{V}\frac{\Sigma^2}{n-1}\boldsymbol{V}^T. \end{aligned} \tag{2.39}$$

The right singular vectors $\boldsymbol{V}$ are the principal directions, and singular values are related to the eigenvalues of the covariance matrix via $\lambda_i = \frac{s_i^2}{n-1}$. The principal components are then given by:

$$\begin{aligned} \boldsymbol{X}\boldsymbol{V} &= \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\boldsymbol{V} \\ &= \boldsymbol{U}\boldsymbol{\Sigma} \end{aligned} \tag{2.40}$$

Where the principal components are given by the columns of $\boldsymbol{U}\boldsymbol{\Sigma}$.

2.8.2. The Curse of Dimensionality

When working with high-dimensional data, it is noteworthy to comment the term coined by Richard E. Bellman [50], "the curse of dimensionality". It refers to the fact that when dimensionality of the data investigated increases, the *volume* of the space increases so fast such that the data becomes sparse within this space. This sparsity can be problematic for any method that requires statistical significance. In order to obtain a statistically reliable result, the amount of data needed to support the result often grows exponentially with the dimensionality [50]. Also, organizing data by group adherence can become difficult in high-dimensional space, as all objects can become sparse and dissimilar [51]. This is to be taken into account for later chapters when applying the K-means algorithm on the dataset to cluster into groups, as well as serves the point for using the dimensionality reduction tool of principal component analysis.

3. Tools for implementation

As the theory of the different methods used in this thesis was presented in the previous chapter, this chapter focuses on how these methods can be implemented and used. The reasoning for which tools were chosen, and why, is presented in this chapter.

3.1. Choosing The Programming language

When deciding what programming language to use when implementing machine learning methods, there are several factors that need to be taken into account. Pro's and con's of several languages must be weighed. As of 2018, there is quite a selection of suitable languages for machine learning. Since several languages have well-developed packages and libraries for machine learning, all of the languages which do not are automatically left out of the equation. A short-list of the currently most popular languages for machine learning are [52] [53]:

- Python
- R
- Matlab
- Julia

The first thing to evaluate when considering which language to use, might be which is the most popular. This is because it is easy to assume that the most popular language most likely has the largest community, largest support, and perhaps the best tools in the market for what one might need. In figure 3.1 one can see the Google Trend searches [17], how popular each term has been on Google's search engine since 2012. However, even

though Python sticks out as the most popular of the Google searches, a little deeper dive into each language will follow, to ensure the correct choice of tools for this thesis.

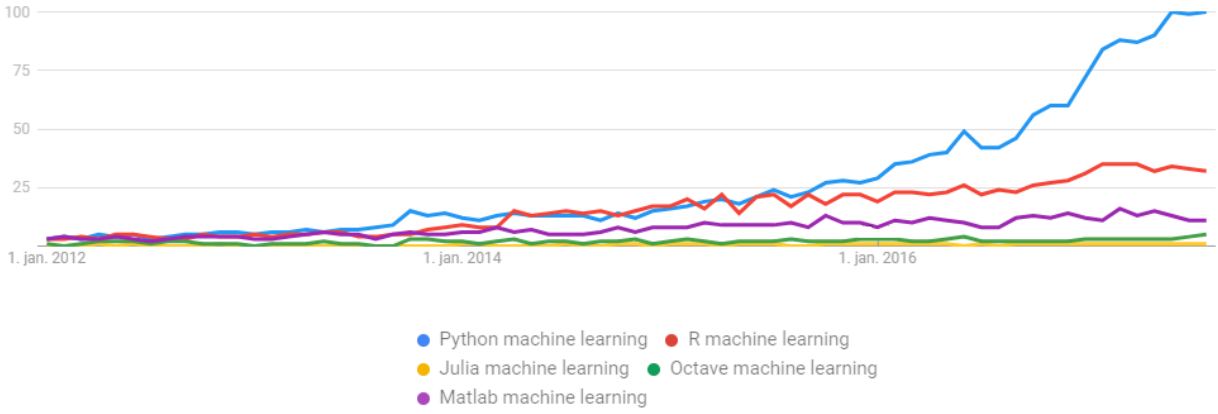


Figure 3.1.: Popularity of searches on Google [17].

3.1.1. R

R is a free, open source software environment and programming language, used most often for statistical computing and graphical display of its output [54]. It is popular for its vast number of machine learning algorithms implemented as third party packages or libraries. Used in large scale by academics in statistical fields, giving access to some state-of-the-art methods and community maturity [55]. It is an interpreted language, and not a compiled language. A compiled language converts the code into machine-code once, and thus the executable program will run in the same manner until a new program is compiled after some potential changes in the code. This is not the case with interpreted languages, where the interpreter executes the program directly, translating each statement in the code into a sequence of one or more subroutines, and then into machine code. Compiled programs thus generally run faster than interpreted ones, because interpreted programs must be reduced to machine instructions at run-time. This could prove to not be beneficial when working with large sets of data. However, since this thesis will only do a finite amount of analysis and computations, the speed is not of big importance. If one were to implement an online machine learning program, this point would be of greater importance. Some other difficulties of R are that documentation is deemed by the community to be too abstract and difficult, and the language in general having low scalability [52].

3.1.2. Matlab

Matlab is a programming environment also used in large scale by academics, but largely by industry as well. It's main strengths are its ease of use for beginners who have little knowledge of programming, but are in need of computational assistance. This could be everything from structural engineering and finite element analysis, to advanced mathematics and analytics. Matlab's syntax is also known to be quite easy, and attempts to mimic the manner in which science is done by hand to easily translate to the program.

One of the main downsides for the choice of Matlab, is its cost. It is a commercial software, which eliminates some subset of the user base. In turn this also makes the community smaller, and possibly makes it harder to find solutions to given problems when one gets stuck.

3.1.3. Julia

Julia is a programming language that was created to cater to the needs of high-performance numerical analysis and computational science. It's goal is possibly the most optimistic and far-reaching on the list, and is stated as follows [56]:

> "We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled."

With these goals in mind, it almost becomes difficult not to choose Julia for this assignment. If all that is promised is given, Julia would conquer the world of analytics and computational science. However, no matter the promise of a language or environment, the user base is always of importance. If the language is not used, there is little help to get when one needs help. Even though it has been under development since 2009, release of version 1.0 saw the light of day on august 9th, 2018 [57]. It is thus considered a very fresh language and will take some time to reach maturity.

3.1.4. Python

As of 2018, Python is one of the fastest growing programming languages. Shown in figure 3.2 [58], queries regarding Python has risen tremendously on the developer community website Stack Overflow, the world's largest developer community [59]. There are many reasons why Python has risen to popularity. It has an easy-to-learn syntax, while still not sacrificing performance, making it a good trade-off with regards to complexity and performance [60]. It is open source, and has a large community. This in turn makes for a rich set of libraries, good documentation and good possibilities for help available.

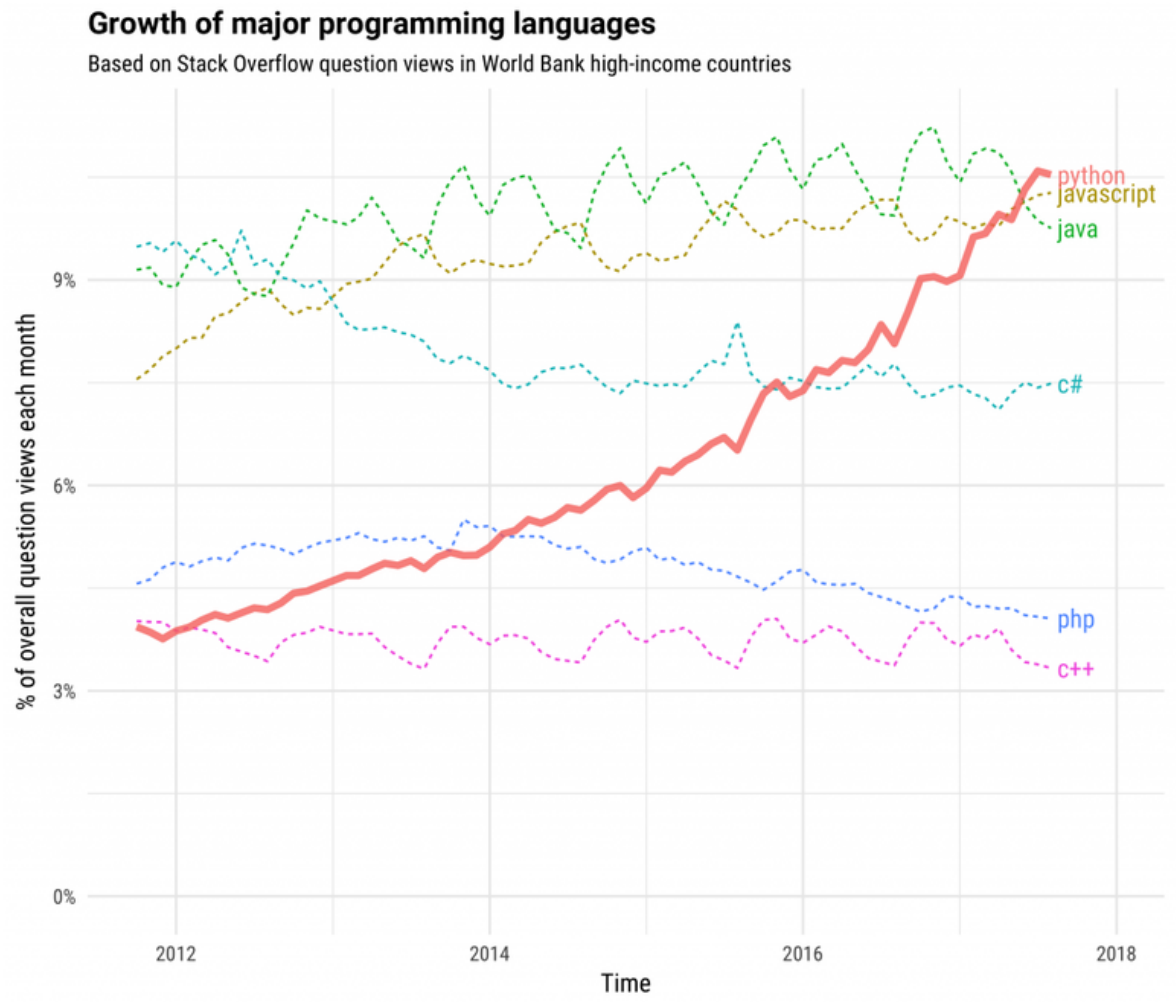


Figure 3.2.: Python's popularity on Stack Overflow [18].

There could be numerous reasons why exactly Python seems to be dominating as lan-

guage of choice for machine learning, but it can be thought that the combination of its ease-of-use and well-developed machine learning libraries is what has won people over. For this reason, and its large community, Python was chosen as the language of choice for this thesis. Its tools and libraries will be further discussed in the following chapters.

3.2. Python's data science and machine learning tools

Being that Python is an open source language steadily gaining popularity, several useful libraries for machine learning and other analytic purposes have emerged. Some of the most popular and well-developed libraries are NumPy, Pandas, and Matplotlib.

NumPy is one of the fundamental packages/extensions to Python for scientific computing. This is due to its features being:

- A powerful N-dimensional array object.
- Sophisticated broadcasting functions.
- Tools for integrating C/C++ and Fortran code.
- quick computing of linear algebra, Fourier transforms and random number capabilities.

A useful feature of the array object, is the runtime computation time of matrix multiplication. A regular list in Python are arrays of pointers to objects, with per-element dynamic type checking. A NumPy array is solely of homogeneous type and its operations are implemented in C, avoiding the general cost of loops in Python, making it much quicker. This can be seen in figure 3.3 [19], where a matrix dot-product is performed a thousand times in three different manners and has its runtime performance measured.

```
import timeit
normal_py_sec = timeit.timeit('sum(x*x for x in xrange(1000))',
                              number=10000)
naive_np_sec = timeit.timeit('sum(na*na)',
                             setup="import numpy as np; na=np.
arange(1000)",
                             number=10000)
good_np_sec = timeit.timeit('na.dot(na)',
                            setup="import numpy as np; na=np.
arange(1000)",
                            number=10000)

print("Normal Python: %f sec"%normal_py_sec)
print("Naive NumPy: %f sec"%naive_np_sec)
print("Good NumPy: %f sec"%good_np_sec)

Normal Python: 1.157467 sec
Naive NumPy: 4.061293 sec
Good NumPy: 0.033419 sec
```

Figure 3.3.: Run-time of NumPy compared to normal Python loop [19].

a NumPy array "na" is used via the NumPy .dot()-function, here called "Good NumPy". The .dot()-function is a dot product of two arrays, which in this case is the same array, "na". It can be seen that the runtime of "Good NumPy" is approximately 35 times faster than that of "Normal Python", using a standard programming implementation of matrix dot-product. "Naive NumPy" can be excluded due to not being very relevant for this point.

Pandas is a package for fast, flexible and expressive data structures, made to work with relational or labeled data. It is well suited for many different kinds of data, for example:

- Tabular data, like a SQL table or Excel spreadsheet.
- Ordered and unordered time series data.
- Arbitrary matrix data with row and column labels

Pandas is built on top of NumPy and is thus intended to integrate well within Pythons scientific computing environment. This allows Pandas to work well at data manipulation of large data sets, be it ordered or unordered data. Pandas is fast, as many of the low-level

algorithmic bits have been tweaked through Cython, the compiler for C extensions made for Python [61].

Matplotlib is an open source Python 2D plotting library, available with various printout formats and interactive environments. It is used to generate plots, histograms, power spectra, bar charts, scatterplots, etc. It is made to seamlessly interact with other Python-based libraries for numerical analysis, such as those previously mentioned (NumPy, Pandas).

3.2.1. Anaconda

Anaconda is a Python distribution containing the Python run-time environment, several integrated development environments and over 1,400 popular data science packages. In figure 3.4 one can see a snippet of what the distribution contains, with mainly its focus on data science libraries [20]. It also contains, amongst others, Scikit-learn, TensorFlow and Keras, which are popular packages with data science and machine learning functions and utilities built on top of the underlying packages mentioned in the previous section, mainly NumPy and/or SciPy, Pandas and Matplotlib.

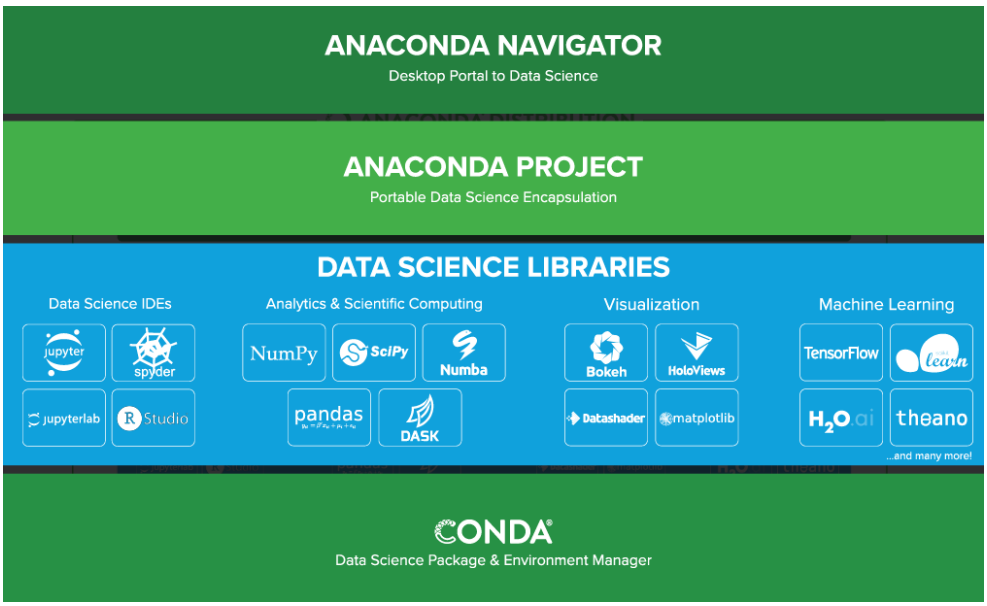


Figure 3.4.: Contents of the Anaconda Distribution [20].

Scikit-learn offers a lot of functions and algorithms for machine learning, and is one of

the packages most used in this thesis. It offers algorithms for classification, such as support vector machines, nearest neighbors and random forest. Algorithms for regression, such as ridge regression. Clustering algorithms, such as k-Means, spectral clustering and affinity propagation, which will be shown in section X. It also offers algorithms for dimensionality reduction, such as PCA, and utilities for preprocessing of the data.

The advantage of using the Anaconda distribution is its direct integration of all available libraries and various choice of integrated developing environments (IDE's), such as "spyder", "jupyterlab", "Jupyter notebook" and "psyplot-gui" [20]. The IDE that is used in this thesis, is the web-based computing notebook environment Jupyter notebook.

3.2.2. Jupyter notebook

The Jupyter notebook is an open-source web application and IDE that makes it easy to prototype and develop Python scripts and programs in the web-browser. When installed via the Anaconda framework, all of Anaconda's libraries are readily implementable. Figure 3.5 shows the graphical layout of how the Jupyter notebook looks like and work [21]. One can easily create new modules in the application, which can run dependently or independently of other modules. The notebook can be posted online and shared with others, for easy access to whatever one has created and wants to show others.

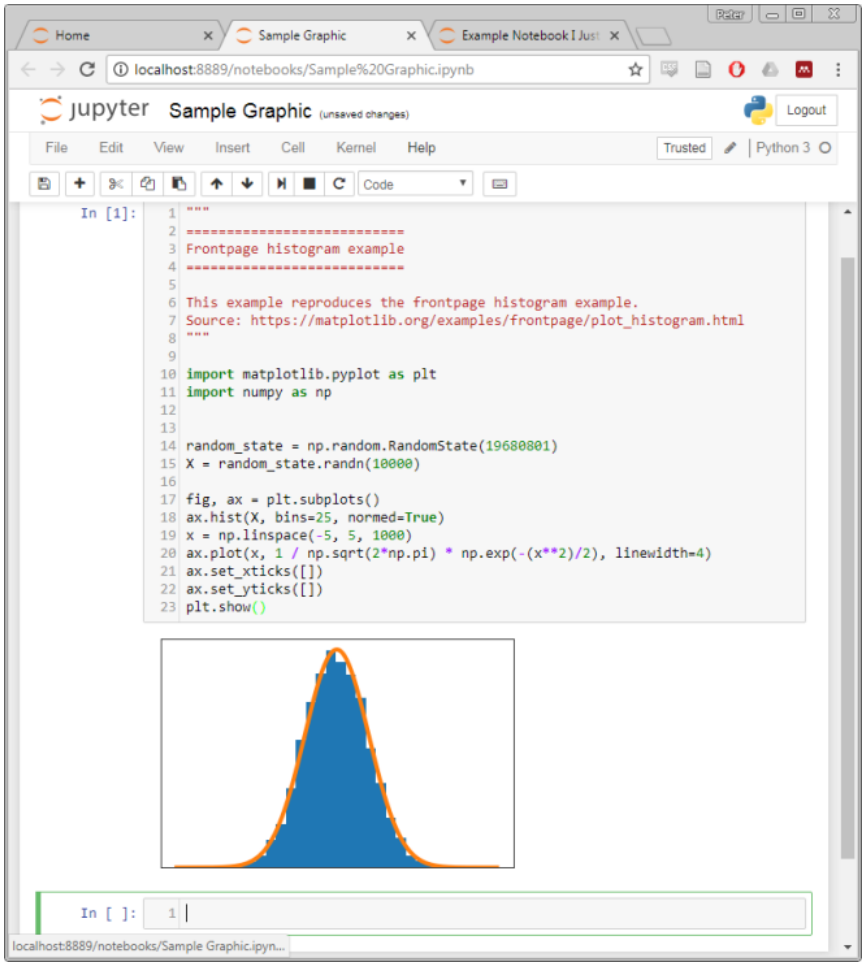


Figure 3.5.: Example layout of Jupyter [21].

It is due to the ease of implementation and display that Jupyter notebook has been chosen as IDE for this thesis. In appendix A one can see code written for this thesis, written in Python on Jupyter notebook.

4. Tests and results

With the theory explained, it is time to explain how and what the tests to be done are. This chapter will first give a short introduction into what tests are to be done, followed by how they are performed and the results they yield. Interpretation and reflection regarding the tests and results will mostly be done in the next chapter, with a conclusion drawn in the last chapter.

4.1. Data receival and test setup

Figure 5.1a illustrates the scenery of how and where the data is obtained. A camera is attached to a post, providing surveillance over the container beneath it. Images are taken of the container once a minute, throughout the day. The camera is connected via bluetooth to a portable field-router, which in turn is connected to the internet via 4G. The images are sent to Acando, where they manually perform the labeling, and assort the images into their respective folders, as explained in the introduction. This is done in batches, as no employee sits in real-time and labels every image coming in once a minute.

(a) Camera setup.

(b) Full picture from camera.

(c) Region of image that is cropped.

(d) Cropped and transformed.

Figure 4.1.: Overview of how data gets collected.

As stated in the introduction, labeled images from the first month of surveillance were the basis of the dataset given for this thesis. The dataset consists of 4802 images, distributed over the 5 labels "0%", "25%", "50%", "75%" and "100%", which represent the filling rate of wood waste in the container. As the dataset was handed over for this thesis, and the tools for implementation mentioned in the previous chapter were selected, the tests could begin to be made. Since Jupyter notebook was the selected framework for the work of this thesis, it serves as the hub and tool for most of the methods and data processing in this thesis. Everything is written in Python, and executed within the notebook. The complete code is found in appendix A. The code can be broken down in mainly three sections:

- Import of images and image pre-processing.
- Feature extraction from images.

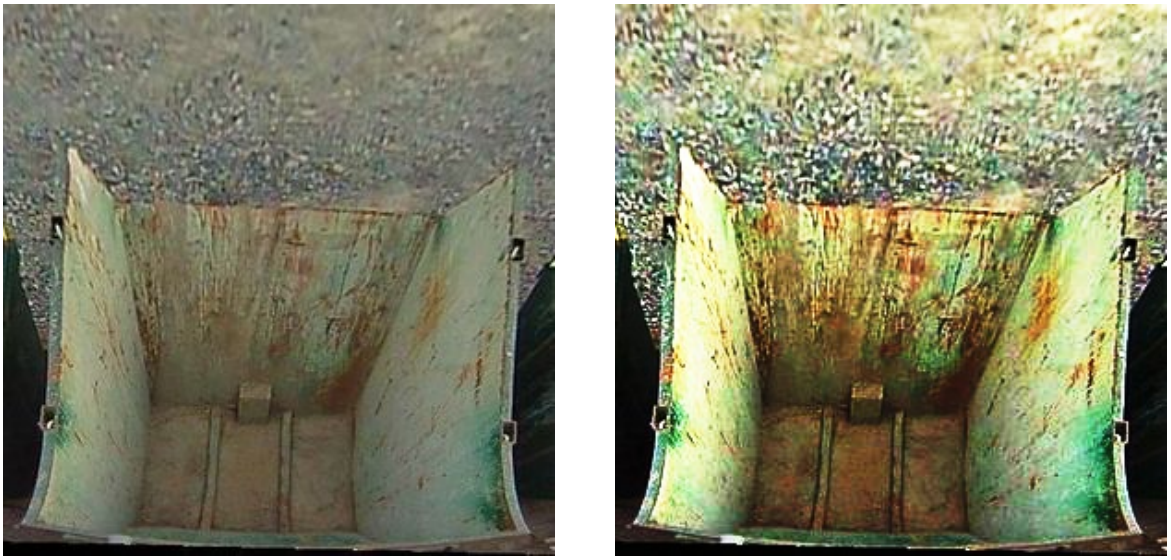
- Principal component analysis and K-means clustering of extracted features.

For testing of the convolutional neural network, the training of the CNN is done in the shell by running the python-file "retrain.py", and tested by looping the single image predictor "label_image.py" over the test set. The code for these two Python-files are found in appendix B and C, respectively.

4.2. Pre-processing of data

4.2.1. Image pre-processing

Typical image pre-processing techniques are contrast correction and noise reduction. When an image is displayed with a narrow color or brightness range, it can be visually hard to see the contents of the image and the image can be in need of contrast correction. This is typically done either through normalization or histogram equalization. It was experimented with contrast correction as a means of pre-processing in this thesis, but it was found to be counter intuitive. Due to the containers and/or the surrounding environment being noisy (with e.g. rust spots on the container walls), this pre-processing tool made such noise *even more* noticeable, as can be shown in figure 4.2.



(a) Original image. (b) Image after histogram equalization.

Figure 4.2.: Difference with or without histogram equalization.

Feature extraction methods are more successful with the less noise prevalent, so this pre-processing step was therefore not applied. With regards to noise reduction, a convolutional 2D-filter is applied, which effectively blurs the image. This smooths out spots and soft corners, making the feature extraction methods react less to noise and have a higher threshold before e.g. detecting edges.

4.2.2. Feature extraction

The tests in this chapter will be based on the data within the *feature vectors* analyzed. For every image in the provided dataset, six feature extraction methods have been performed. These are the six feature extraction methods described in chapter 2 - Theory. However, the output of these different feature extraction methods were not uniform, and not necessarily compatible to be successfully implemented in K-means as is. Why this is, is covered more in detail in chapter 5 - Discussion. However, each feature extraction needed to be represented in such a manner that it could be purposefully tested via K-means. In order to do this, it was decided to make a numerical representation of each feature extraction, such that each image produced a feature vector containing six single number elements. The following list explains how and why the respective feature extraction was represented in that specific manner:

- *Histogram of Oriented Gradients*: The output of the HOG-function has the same dimension as the input image. When the output-matrix is plotted, one can see the histograms of oriented gradients mapped onto a black substrate, as seen in chapter 2 - Theory. The gradients grew large when the change in texture in the region of the image was high. Thus, the uniform regions of the image, such as the walls of the container, tended to create little to no gradient histograms in said regions. The result would be a much greater portion of the image being black. Therefore it was decided that a numerical representation of the fill rate of the container could be the *mean pixel intensity* of the transformed image. The expected outcome would then be that full containers gave a high mean intensity due to the presence of many large white gradient histograms, with the opposite being true for empty containers.

- *Hough Line Transform*: As the output of the Hough Line Transform is similar to that of the HOG-function, the same assumption and representation is made. Instead

of creating gradient histograms, the Hough Line Transform maps the discovered lines onto a black substrate, resulting in a 2D-matrix representing the transformed image. As seen on the images of the Hough transform (REF), the wood waste is detected quite successfully and thus allow for the assumption of more wood waste correlating with higher mean intensity. Thus, *mean pixel intensity* is also the numerical representation of the Hough Line Transform.

- *Hough-distance*: This algorithm takes the HLT-transformed image as input, and iterates over every row of the transformed image. It counts batches of regions of black in each row, and gives exponential weight to the size of the batch (batch size to the power of two), in order to reward long stretches of black. The weighted batch sizes for all rows are accumulated, and this accumulation is the output of the algorithm. The algorithm was made under the assumption that containers with a low fill rate would have longer stretches of black, outputting a larger accumulated count for little waste in the container. Thus, the numerical representation for fill rate in this method is the *accumulated count* made by the algorithm.

- *SIFT*: The SIFT method detects a number of unique points. These points are stored in a vector, representing the location of the different points. The assumption for this method is that the more wood waste in the container, the more unique points are located. Thus, the numerical representation for this method, is the *number of points* located, found via the size of the vector mentioned.

- *SURF*: Similar to SIFT, but locating the keypoints via approximation of Laplacian of Gaussian by box filters instead of calculating the Differences og Gaussians. However, the same numerical representation is used – the *number of points* located.

- *Histogram Threshold*: In this case, it is assumed that the color difference between the container and the waste will set the threshold between the two and thus differentieting between them. As the output matrix is equal in form to that of HOG and Hough Line Transform, the *mean pixel intensity* of the transformed image is used.

Each feature extraction method is now have a numerical representation, such that a feature vector of six elements/feature representations is obtained, and unsupervised learning tests can commence.

4.3. Tests and results

In this section, five different tests are conducted, where one of them is expanded upon. It has been established that all 4802 images have gone through image pre-processing and are each allotted a feature vector and its original label. Thus, there are 4802 labeled feature vectors, where each vector consists of six elements. Each element is a numerical representation of the feature transformation done on the image. These 4802 feature vectors are the inputs of each of the three tests. The five tests are:

1. K-means clustering on all six dimensions, using the complete dataset of 4802 feature vectors.

2. K-means clustering on two dimensions, being the two principal components reduced from six dimensions via principal component analysis. Also using the complete dataset of 4802 feature vectors.

3. K-means clustering on two dimensions, being the two principal components reduced from six dimensions via principal component analysis. However, testing on only 25 images. Following is a population proportion-test to investigate if there is any statistical validity for the claim of only needing 25 images to get approximately equal results as with complete dataset.

4. K-means clustering on two dimensions, using the complete dataset of 4802 feature vectors. The two dimensions are all combinations of selecting two of the six different dimensions of feature transformation.

5. A convolutional neural network trained on the complete dataset, and tested on a new and smaller test set.

After all tests are conducted, a table of results for all tests is presented at the end of this chapter.

4.3.1. 6-dimensional K-means clustering on full dataset

Even though the complete code is presented in appendix A, it was thought useful to share the algorithm of the K-means. It is after all applied to most of the tests, with minor tweaks in between. To give a brief explanation of the code, it runs as follows:

The number of desired clusters, K, is set to be 5. This is to mimic grouping the data into five labels. If the position of the cluster center does not change more than the given tolerance, convergence is assumed and algorithm is ended. Max iterations is the amount of iterations the algorithm runs before ending, meaning if convergence is not reached within this point. This could mean several equal local minima with a spatial difference larger than given tolerance. In this case, the cluster centroid of the 500th iteration is chosen. After the K-means algorithm is complete, each data point (single instance of a feature vector) is given a cluster adherence. Then the function "cluster_correctness()" runs over the dataset and measures to which degree the cluster adherence matches the original label. The result of this is what's called *correctness*.

If, e.g. 5 feature vectors out of a dataset with size 10 has the same cluster value as label, the correctness would equal $5/10 = 0.5$. The cluster centroids initial positions are assigned to equal the position of a given data points/feature vector. These locations are decided to be distributed evenly, by being placed on $1/k * size(dataset) * i$ for $i = \{1, 2, ..., k\}$. An example of this would be that for a dataset with size 10, the cluster centroids positions are initialized at position [dataset[2], dataset[4], dataset[6], dataset[8], dataset[10]]. Here it is important to point out that the dataset needs to be sorted in order to receive the correct cluster centroid number. However, this is just important when comparing to *labels* in order to measure correctness, and not when performed on an unlabeled dataset. As explained in the chapter 2 - Theory, the clusters purpose is to minimize the distance of the points within them. Thus random cluster centroid initialization can work on an unlabeled dataset, but measuring correctness on a labeled dataset will probably not perform well. What has now been explained, is shown in python code beneath. The code is influenced by open source code [**?**] and adjusted by the author.

```
class K_Means:
    def __init__(self, k =5, tolerance = 0.0001, max_iterations = 500):
        self.k = k
        self.tolerance = tolerance
        self.max_iterations = max_iterations
        self.labelClasses = {}

    def fit(self, data):

        self.centroids = {}
```

```
11          centroidInterval = len(data)/k
12
13          #initialize the centroids evenly throughout the dataset
14          for i in range(self.k):
15              self.centroids[i] = data[centroidInterval*i]
16
17          #begin iterations
18          for i in range(self.max_iterations):
19              self.classes = {}
20              for i in range(self.k):
21                  self.classes[i] = []
22
23              #find the distance between the point and cluster;
24        #choose the nearest centroid
25              for features in data:
26                  distances = [np.linalg.norm(features - self.centroids[
    centroid])
27           for centroid in self.centroids]
28                  classification = distances.index(min(distances))
29                  self.classes[classification].append(features)
30
31              previous = dict(self.centroids)
32
33              #average the cluster datapoints to re-calculate the centroids
34              for classification in self.classes:
35                  self.centroids[classification] = np.average(
36           self.classes[classification], axis = 0)
37
38              isOptimal = True
39
40              for centroid in self.centroids:
41
42                  original_centroid = previous[centroid]
43                  curr = self.centroids[centroid]
44
45                  if np.sum((curr - original_centroid)/original_centroid *
    100.0)
46               > self.tolerance:
47                      isOptimal = False
48
```

```
            #break out of the main loop if the results are optimal, ie.
 #the centroids don't change their positions (more than given tolerance)
            if isOptimal:
                self.labelClasses = self.classes
                break


    def pred(self, data):
        distances = [np.linalg.norm(data - self.centroids[centroid])
          for centroid in self.centroids]
        classification = distances.index(min(distances))
        return classification

    def get_class(self):
        return self.labelClasses

def cluster_correctness(cluster_class):
    match = 0
    not_match = 0
    c = cluster_class

    for x in range(0,5):
        tempClusterArr = c[x]
        tempClusterArr = np.array(tempClusterArr)
        for i in range(len(tempClusterArr)):
            if tempClusterArr[i][6] == x:
                match += 1
            else:
                not_match += 1

    return match/(match+not_match)

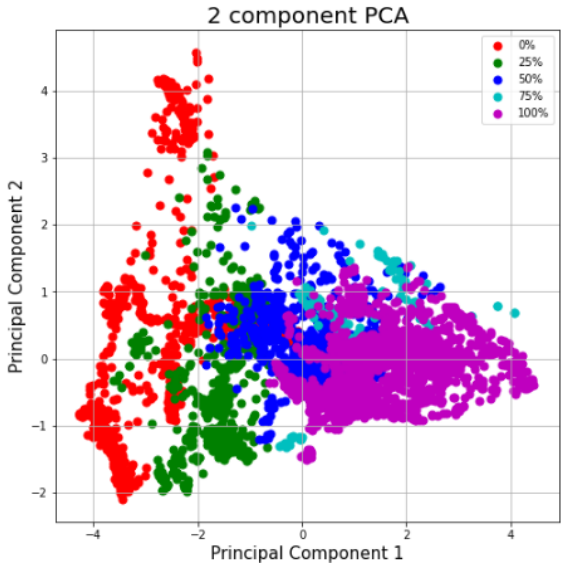

def main(df):

    X = df.values #returns an array of feature vectors from a DataFrame (df)
    km = K_Means(5)
    km.fit(X)
    cluster_class = km.get_class()
    cluster_correctness(cluster_class)
```

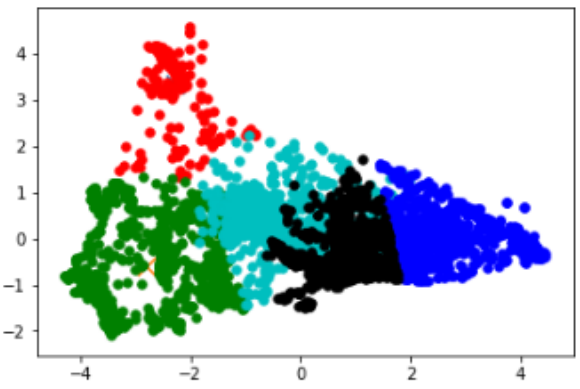
Running this on the complete dataset of 4802 feature vectors yielded *correctness* = 0.448.

4.3.2. K-means clustering on PCA-reduced set

This test is run with the same K-means algorithm and correctness measure, however the dataset is reduced to two dimensions, down from six. These two dimensions being the two principal axes, with a new set of feature vectors, [principal component 1, principal component 2]. Due to being only two dimensions, the results can be visualized. In figure 4.3 one can see a plot of the PCA-reduced dataset on the two principal axes on the left side, with colors representing their *original label*. That means that e.g. the purple points represents the images that were labeled as 100% full by Acando. On the right is a plot of the result of K-means ran on the PCA-reduced dataset. Here it is *very* important to state that the colors on the K-means plot are chosen *at random*, and do *not* correlate with the label colors of the PCA-plot. It is just a visual aid to see the groupings/clusters of K-means algorithms result.



(a) Plot of PCA with original labels



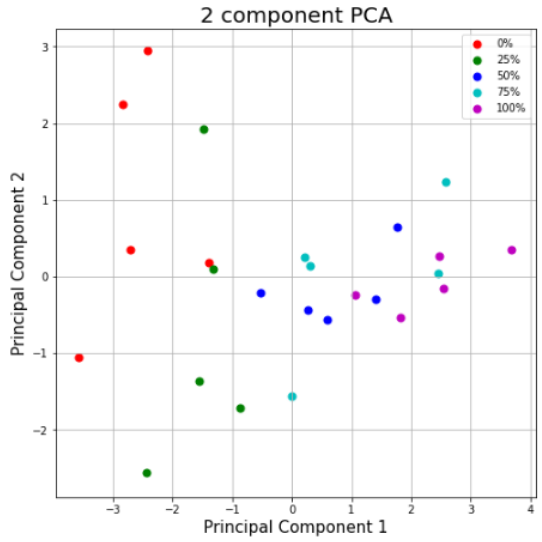
(b) K-means clustering of 2-component PCA

Figure 4.3.: Test of K-means applied on PCA-reduced dataset.

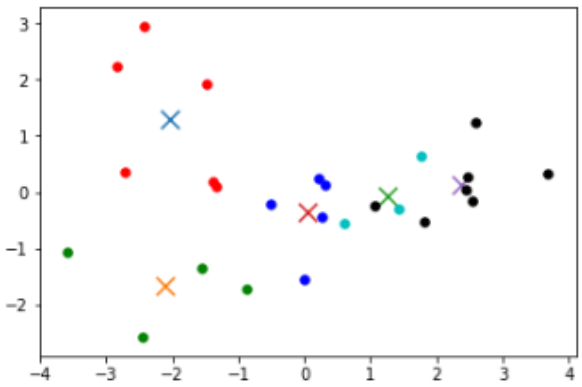
The result of this test yielded *correctness* = 0.572.

4.3.3. K-means clustering on PCA-reduced set with a small subset of images

One of the main interests of applying unsupervised learning methods, was not only to see whether labeled datasets could be dispensed with, but also to see whether approximately the same results can be obtained with a drastically reduced dataset than what is needed for supervised learning methods. A small example test of this is conducted and shown beneath, in figure 4.4. The downsized test was chosen to be done on the PCA-reduced dataset, as this both yielded a better result when tested on the complete dataset and has the possibility of being visualized with being two dimensions.



(a) Plot of PCA with original labels



(b) K-means clustering of 2-component PCA

Figure 4.4.: K-means applied on PCA-reduced dataset of size 25.

Here one can see only 25 images that has been feature transformed, PCA-reduced and clustered via K-means. The distribution of points bare similarity with that of the complete dataset, with just fewer and more sparse data points. In this particular example, the test yielded $correctness = 17/25 = 0.68$.

As this seems like a promising representative for having to use much less images to obtain approximately the same results, and even scored quite a lot better than the test of the complete dataset. This could of course be a lucky coincidence, and therefore needs to

be tested more in order determine if this claim could be valid. A test set of 30 samples were created, as a compromise between the size of the complete dataset, and the possibility to obtain inference as to whether this claim carries any statistical significance. Each sample consists of the same procedure as shown above, in figure 4.4. One sample represents 5 images per label, a total of 25 images per sample. All images in this test was chosen randomly from the complete dataset.

A population proportion-test is conducted on the following basis: The probability that the population proportion $\hat{p}$ resides within two standard deviations of the true population proportion p equals the probability that the true population proportion resides within two standard deviations of the estimated proportion population $\hat{p}$ with 95% confidence. For this to be true, the following criteria needs to be met:

- The data's individual observation have to be obtained from a simple random sample of the population of interest.
- The data's individual observations have to display normality. This can be verified mathematically with the following definition: Let n be the sample size of the population and $\hat{p}$ be its sample proportion. If $n\hat{p} \geq 10$ and $n(1-\hat{p}) \geq 10$, the data's individual observations portray normality.
- The data's individual observations need to be independent of each other. This is verified by letting N be the size of the true population, where one needs $N \geq 10n$.

The claim $n\hat{p} \geq 10$ and $n(1-\hat{p}) \geq 10$ for this test equals $30 \times 0.624 \geq 10$ and $30 \times (1 - 0.624) \geq 10$, which both hold true. For $N \geq 10n$, let N equals the size of the complete dataset divided by the amount of images needed to make a single sample, $N = 4802/25 = 192$. This, however, fails the test of independence where $N \geq 10n$ does not hold true. This is sadly due to the size of the dataset given for this thesis is not large enough. However, even though one can not verify the results of this test on this basis, it will still be performed in order to lay the foundation for further research to be able to test this claim with a larger dataset, such that the results can be valid. It is therefore *assumed* that the data displays independence for this test onwards, in order to complete it.

Let $\hat{p}$ be the mean of the correctness of the 30 samples, where it was found that $\hat{p} = 0.624$.

Let $n = 30$ and confidence $C = 0.95$. Since the test is two-sided, the z*-statistic is found by:

$$z^* = \frac{1 - C}{2} = 0.025. \tag{4.1}$$

From a table of standard normal probabilities, it is found that the Z-value at $Z(0.025) = -1.96$ and $Z(1 - 0.025) = 1.96$.

The standard error of the sample distribution is given as:

$$\begin{aligned} SE_{\hat{p}} &= \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \\ &= \sqrt{\frac{0.624(1-0.624)}{30}} \\ &= 0.087 \end{aligned} \tag{4.2}$$

Then one can say, with 95% confidence, that the true population proportion p lies within

$$\begin{aligned} & [\hat{p} - 1.96 \times SE_{\hat{p}}, \hat{p} + 1.96 \times SE_{\hat{p}}] \\ = \; & [\hat{p} - 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}, \hat{p} + 1.96 \times \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}] \\ = \; & [0.624 - 1.96 \times 0.087, 0.624 + 1.96 \times 0.087] \\ = \; & [0.453, 0.794]. \end{aligned} \tag{4.3}$$

This means that real correctness lies in the range between $[0.453, 0.794]$ with 95% confidence, when tested with 30 samples of 25 images per sample with 5 images per label. The real correctness was measured as 0.574 when using the complete dataset, so the claim of obtaining similar results with a drastically reduced dataset might be plausible, but will be discussed further in section 5 - Discussion.

4.3.4. Clustering on variations of two variables

Until this point, the previous tests have all used the full feature vector of six different elements in their analyses. In this section, a two-dimensional K-means will be performed on all combinations of two elements. This amounts to 15 different tests, as visualized by table

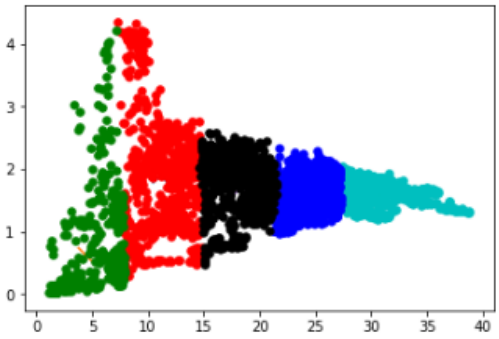
4.1. The "o"'s represent one single test done between their respective feature transform in the vertical and horizontal list of feature transformations. The "x"'s represent either the case where the vertical and horisontal feature transformations are the same (the diagonal line), or if that test has already been done.

	Hough	**Hough-dist**	**HOG**	**Hist.thresh.**	**SIFT**	**SURF**
Hough	x	o	o	o	o	o
Hough-dist	x	x	o	o	o	o
HOG	x	x	x	o	o	o
Hist.thresh.	x	x	x	x	o	o
SIFT	x	x	x	x	x	o

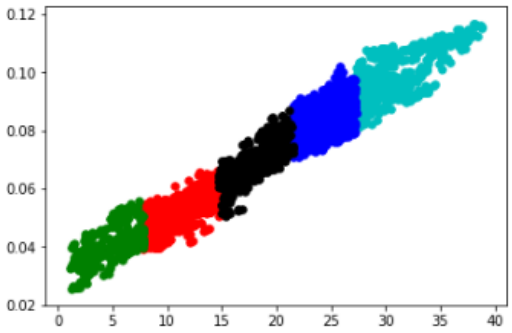
Table 4.1.: Table of elements to be matched and tested.

Following are the plots of all 15 tests. The caption of each plot represents two important factors. It displays which two elements of the feature vector are tested together, and their respective X- and Y-axis. Taking the first plot as example: "Hough Hough-dist" refers to the two elements of the complete 6-dimensional feature vector that represent the "Hough Line Transform", and the author-developed "Hough-distance" algorithm. Since Hough is written first, it represents the X-axis, and Hough-dist represents the Y-axis. This convention applies for all of the 15 plots. The results are presented in table 4.2 at the end of the chapter, sorted by correctness-score from high to low.

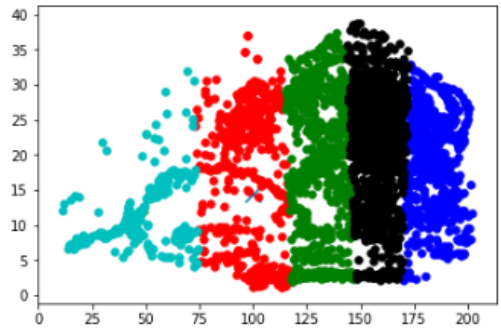
These tests are done in order to investigate whether one can get good results with just two of the feature representations, and/or to see whether some feature representations seem to give better results than others. It is interesting to see how the formation of the data points occur, and make any judgment whether this affect the K-means algorithm's and its correctness-score. The plots of the K-means for all two-variable combinations is shown beneath in figure 4.5 and 4.6. However, discussion regarding the plots and their correctness score is done in chapter 5 - Discussion.



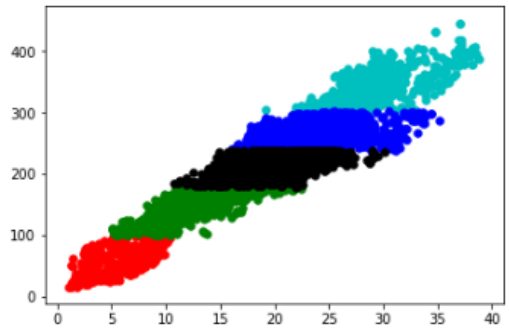
(a) Hough & Hough-dist



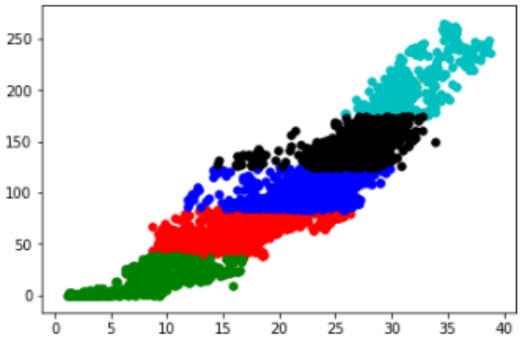
(b) Hough & HOG



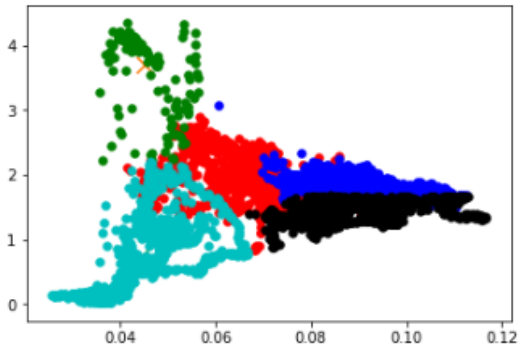
(c) Histogram threshold & Hough



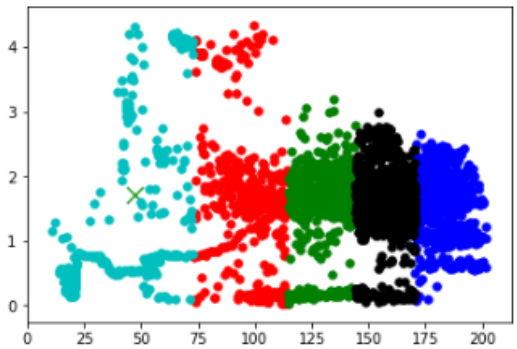
(d) Hough & SIFT



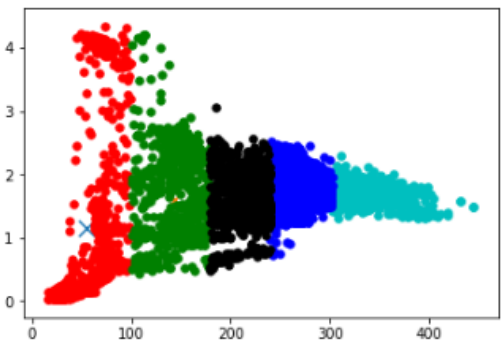
(e) Hough & SURF



(f) HOG & Hough-dist

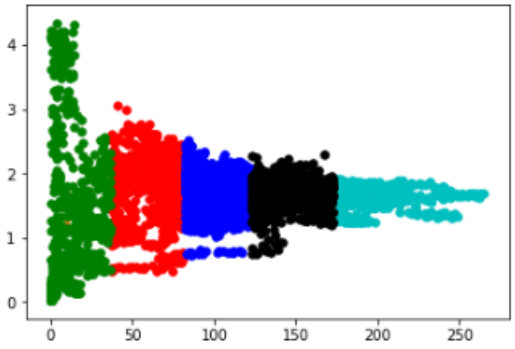


(g) Histogram threshold & Hough-dist

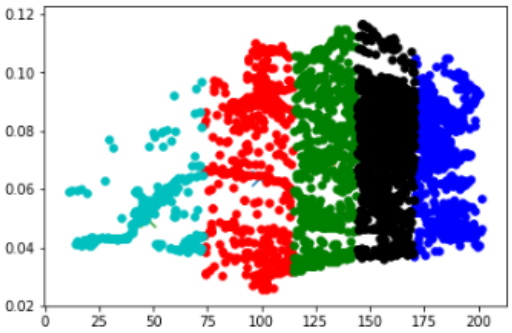


(h) SIFT & Hough-dist

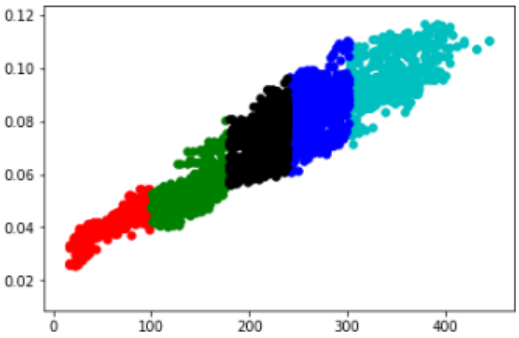
Figure 4.5.: First collection of 2-variable K-means-tests.



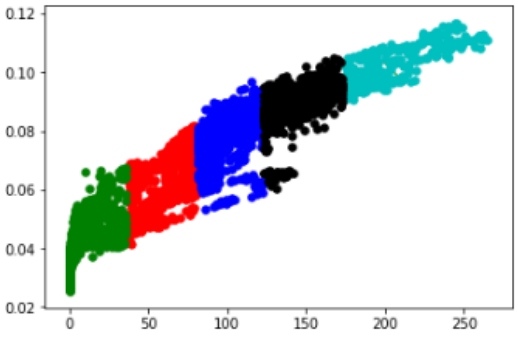
(a) SURF & Hough-dist



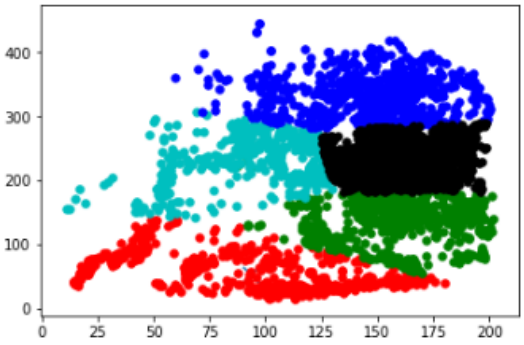
(b) Histogram threshold & HOG



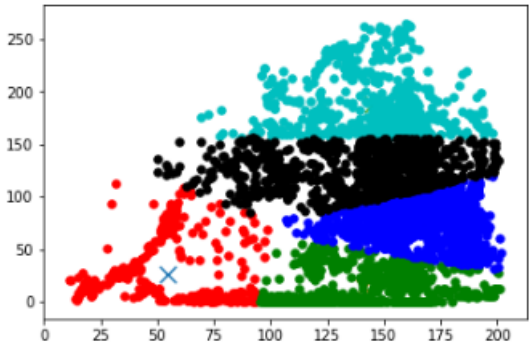
(c) SIFT & HOG



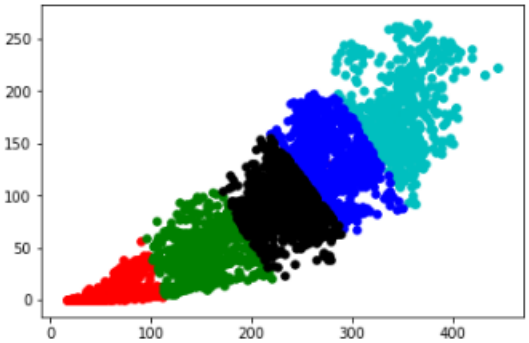
(d) SURF & HOG



(e) Histogram threshold & SIFT



(f) Histogram threshold & SURF



(g) SIFT & SURF

Figure 4.6.: Second collection of 2-variable K-means-tests.

4.3.5. Convolutional Neural Network via Inception V3

Even though most of the work for this thesis has focused on unsupervised learning, one of the goals of this thesis was to compare the results of the unsupervised learning methods to one of the most established image recognition methods in machine learning, namely that of a convolutional neural network (CITE). Since Acando had already chosen a design for its CNN, the open software solution Inception V3, it was natural to use the same such that the result comparisons of all methods would be of interest to Veidekke and Acando.

It is common practice, when training and testing a neural network, to split the dataset in two - a training set and a test set. This is typically split up in 80/20, where 80% of the original dataset is allocated to the training set, and the remaining 20% is for the test set. However, a different approach is chosen for this test. The full dataset will be used for training, and a new dataset is being used for testing. As mentioned in the introduction, this dataset consists of three cycles of waste for the containers. The new dataset is 250 labeled images from other, more recent waste cycles. This was done for two reasons. One reason being that the goal of implementing this technology is to detect each time the container becomes full. Therefore, there will always be variations in how the waste appears in the containers, and the containers might also change appearance (e.g. next container may for example be more rusty than the previous one). Since this is the real life criterion by which success will be measured, it seemed more reasonable to test this way. The second reason is that the CNN might be biased against its test set if the training set and test set are too similar. Given that there are only 3 cycles of waste that has been captured, it can be assumed that the CNN might be overfitting with regards to these 3 cycles, and give unlikely high scores for a test set from the same cycles.

The CNN is trained on the original dataset of 4802 images. The code for the open source Inception V3 training program "retrain.py" is shown in appendix B. The code for Inception V3's prediction program "label_image.py" is shown in appendix C. Following is an example of how a test/prediction is run, with 4 different images per label. With Inception V3, a prediction is run on a single image, outputting its estimate of what percentage chance each label is. The first test set of $4(images) \times 5(labels)$, is just to give a visual representation of Inception V3 operates and is to be interpreted. This is shown beneath, in figures 4.7, 4.8, 4.9, 4.10 and 4.11.

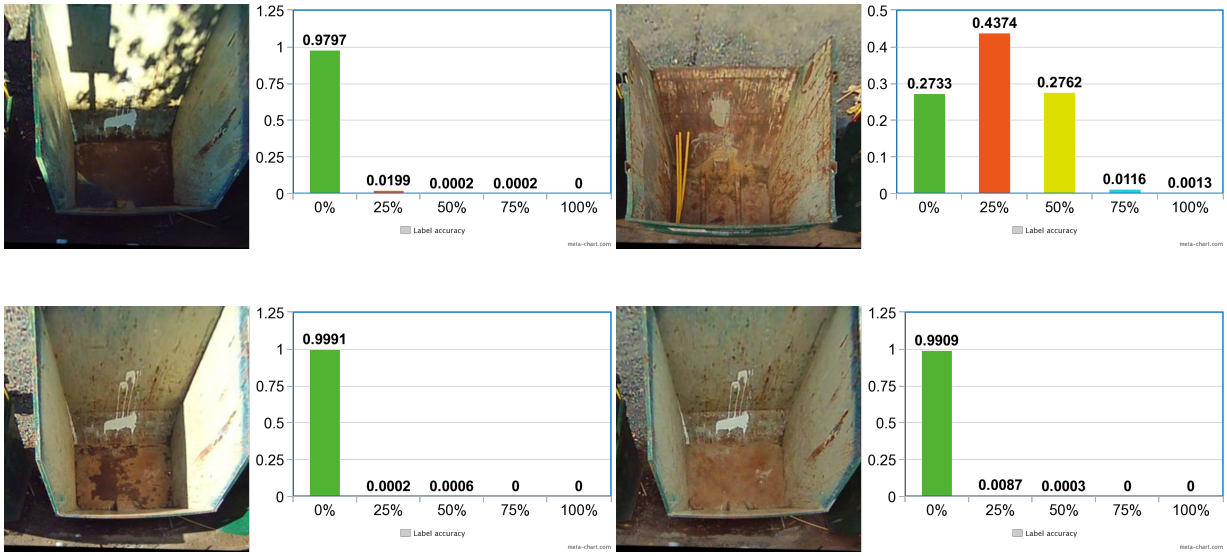


Figure 4.7.: 4 empty containers

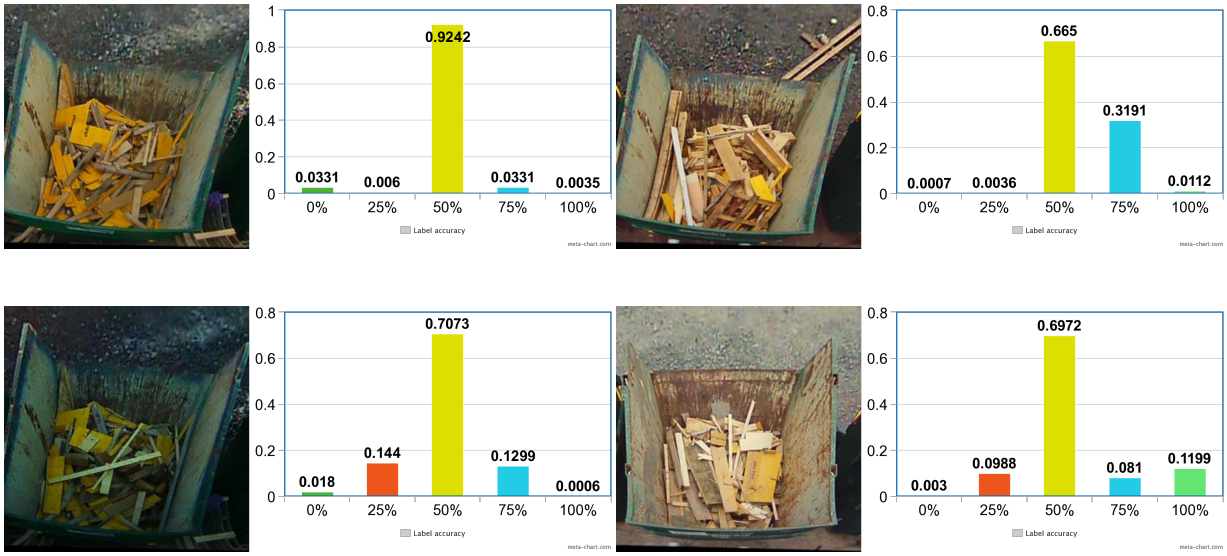


Figure 4.8.: 4 containers at 25%

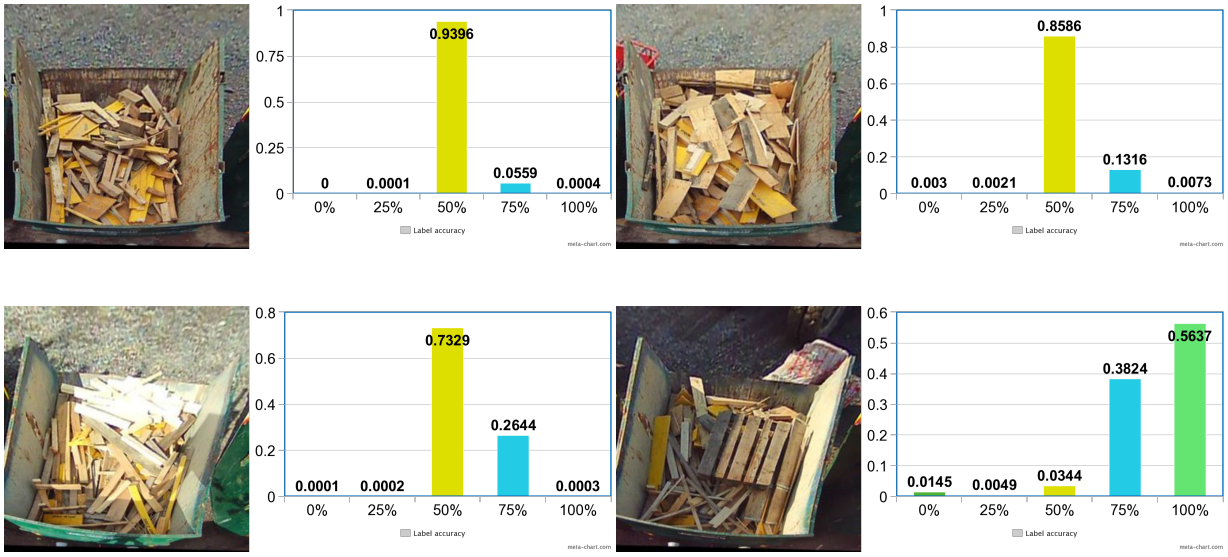


Figure 4.9.: 4 containers at 50%

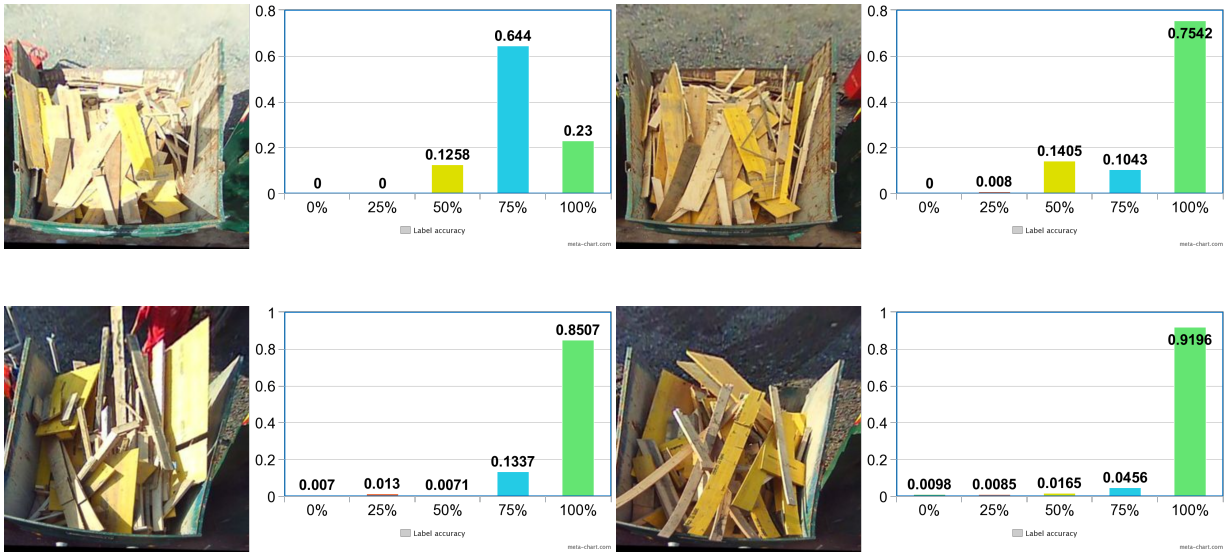


Figure 4.10.: 4 containers at 75%

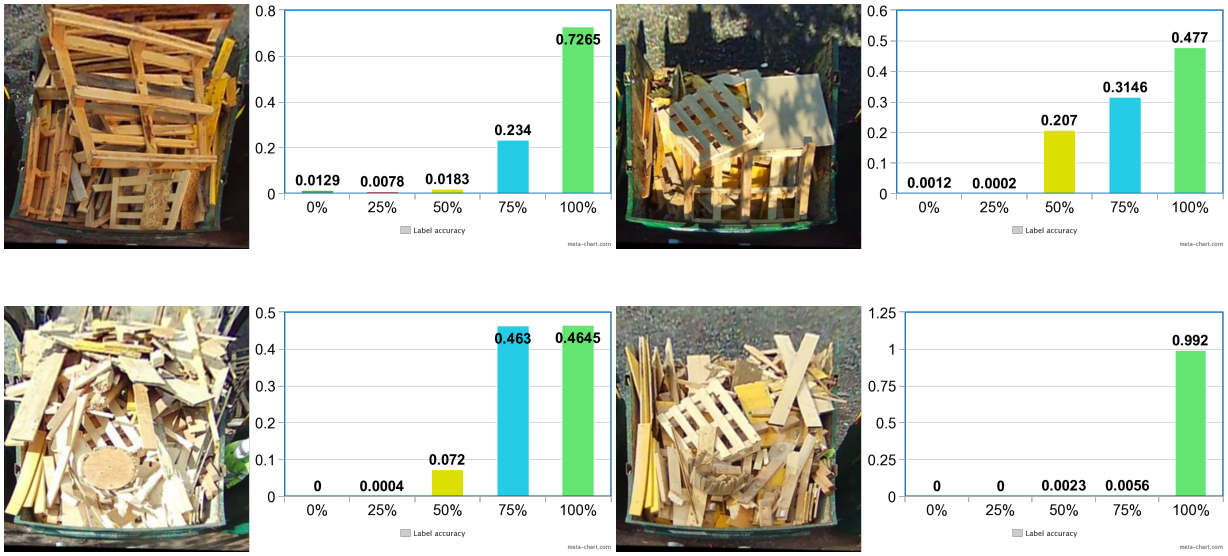


Figure 4.11.: 4 containers at 100%

These individual images' prediction score is averaged *per label* with respect to its true labels. This means e.g. that for the last figure with 4 images, the predictions estimate of "100%" is added up and averaged on the size of the sample (divided by 4). This way the correctness is measured, similarly to that of the unsupervised learning methods. For this small testset, correctness *per label* is shown in figure 4.12.

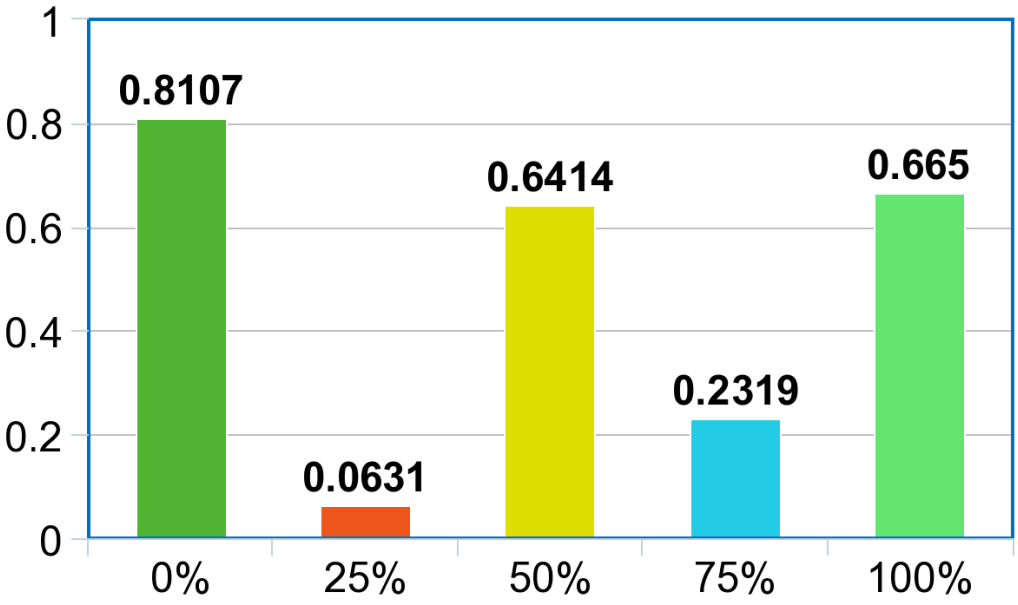


Figure 4.12.: Correctness per label

Averaging each score per label, one gets the total correctness for this test:

$$\frac{0.8107 + 0.0631 + 0.6414 + 0.2319 + 0.6650}{5} = 0.482. \tag{4.4}$$

For the real test set with 250 images, the test is run in the exact same manner as the example test set shown above. However, a simple for-loop was performed in the shell such that all 250 images could be run without manually running prediction on each image. The results of the complete test set with 250 images amounted to $correctness = 0.495$. Finally, the following table, table 4.2, displays all the results of all the tests conducted in this chapter.

4.3.6. Results

Results	
Method	Correctness
6-variable K-means	0.448
2-variable K-means on PCA-reduced set	0.574
Population proportion-test	0.624
CNN	0.495
2-variable K-means with Hist.thresh & SIFT	0.547
2-variable K-means with SIFT & SURF	0.488
2-variable K-means with Hough & SIFT	0.455
2-variable K-means with Hough-dist & SIFT	0.451
2-variable K-means with HOG & SIFT	0.441
2-variable K-means with Hough & Hist.thresh.	0.358
2-variable K-means with Hist.thresh & SURF	0.347
2-variable K-means with HOG & Hist.thresh.	0.344
2-variable K-means with Hough-dist & Hist.thresh.	0.344
2-variable K-means with Hough-dist & HOG	0.329
2-variable K-means with HOG & SURF	0.227
2-variable K-means with Hough-dist & SURF	0.223
2-variable K-means with Hough & SURF	0.221
2-variable K-means with Hough & Hough-dist	0.151
2-variable K-means with Hough & HOG	0.151

Table 4.2.: Results from all tests

5. Discussion

This chapter consists of a discussion about the results of the tests conducted in the previous chapter, methods used in this thesis, and how it was implemented. First, the results of the tests will be discussed, followed by a discussion about the choice of methods, which assumptions were made for using the different methods, and its implications on this thesis. At last, it will be discussed whether the findings of this thesis has any utility outside of this specific project.

5.1. Discussion of results

Five different tests were conducted in chapter 4 - Tests and results. To reiterate the tests conducted, the five tests were:

1. Testing of K-means using the full feature vector, clustering in six-dimensional space with the complete dataset of 4802 images.
2. Testing of K-means on a PCA-reduced set from full feature vector to two principal component variables with the complete dataset.
3. Testing of K-means on a PCA-reduced set from full feature vector to two principal component variables with a subset of the dataset, containing 25 images. For statistical validity, a population proportion test was conducted.
4. Testing of K-means using two of the six variables of the feature vector in every combination.
5. Testing a CNN which was on trained on the complete dataset, spanning 3 waste cycles, and using a test set of 250 images from new waste cycles.

Four of the five tests (1-4) were tests with regards to unsupervised learning methods. The last test was performed with a CNN, a supervised learning method. Following is the discussion of the results for all of the different tests.

5.1.1. Using the full feature vector - Test 1 & 2

- Testing of K-means using the full feature vector, clustering in six-dimensional space with the complete dataset of 4802 images.

- Testing of K-means on a PCA-reduced set from full feature vector to two principal component variables with the complete dataset.

From the list in the previous section, tests 1 and 2 were only different on the basis of whether or not the there had been performed a dimensionality reduction via PCA on the dataset. In test 1, K-means was performed in 6-dimensional space and yielded correctness = 0.448. With a PCA-reduced dataset in test 2, the K-means analysis yielded correctness = 0.574. The PCA-reduced test scored highest of the two tests, and even highest of all the tests using the complete dataset. Why this is the case, might be explained via the curse of dimensionality. When dimensionality increases, the overall volume increases, making the points within the space more sparse. As K-means measures distances in euclidean space, the increase in high-dimensional volume might affect the success of clustering in 6-dimensional space for that respective test. The PCA-reduced test also scored better than any of the tests using a combination of two feature vector variables as well. One possible conclusion to draw from this is that using more feature extraction methods, but reducing the dimensionality of the feature vector, provides both better feature representation *and* the possibility of more successful clustering in 2-dimensional space.

In chapter 1 - Introduction, one of the tasks for this thesis was:

Use of unsupervised learning methods to test if classification/labeling can be performed on the feature vectors with satisfactory results.

An unsupervised learning method for image classification has been found, but it ought to be discussed whether these results can be described as "satisfactory". The reason for implementing a monitoring and detection system on the containers, is to automate and reduce time for alerting the responsible party to pick up a full container in exchange for an empty one. The results of the test does *not* directly reflect this goal. If this were the

goal of the thesis, one could look at figure 4.3a in chapter 4 - Tests and results, showing the plot of the PCA with original labels. From this plot, one can see overlap of the classes "50%", "75%" and "100%". However, there is a clear distinction between, and practically no overlap of, the class "100%" and the classes "25%" and "0%". An unsupervised classification and distinction between empty and full containers could therefore be assumed to be successful and could be satisfactory. On the other side, not being able to fully distinguish between a half-full and a full container might not be thought of as satisfactory, as the cost of swapping containers every time they are half-full might prove costly.

However, what this thesis *is* trying to achieve, is to see whether all classes could be correctly classified without supervision/manual labeling in a satisfactory manner relative to a CNN. Interestingly, the unsupervised methods seem to classify *better* than the CNN *at this stage of implementation.*

5.1.2. Using a small subset of images - Test 3

- Testing of K-means on a PCA-reduced set from full feature vector to two principal component variables with a subset of the dataset, containing 25 images. For statistical validity, a population proportion test was conducted.

With K-means performed on the dimensionality-reduced complete dataset via PCA providing the highest score, this method was chosen to experiment with whether one could obtain a similar correctness score with a drastically reduced number of images from the dataset. It was first tried on a subset of only 25 images, yielding a correctness score of 0.68. However, since this was only one instance, this could be due to luck. Therefore a population proportion test was conducted in order to give some statistical validity to the claim of using few images could yield similar results.

When taking 30 samples of 25 images with 5 images per label, it was found that the mean correctness of the 30 samples were 0.624. This was higher than the PCA-reduced test using the complete dataset. But, given it is a small sample, this is assumed to be statistical deviation. However, the population proportion test stated with 95% confidence that the real correctness (being 0.574 from the PCA-reduced full dataset) was within the range [0.453, 0.794]. 0.574 is within this range, so the claim of obtaining similar results with a

drastically reduced dataset could be correct. However, the size of the dataset restricted the population proportion test to obtain the criterion of independence, namely that $N \geq 10n$. Since $N = 4802/25 = 192$ and $10n = 10 \times 25 = 250$, true independence could not be stated. However, it is not far from obtaining said criterion. It is therefore believed that if this test were to be replicated with a larger dataset, approximately similar results would occur.

5.1.3. Every two-variable combination - Test 4

- Testing of K-means using two of the six variables of the feature vector in every combination.

These tests provided a series of plots (figure 4.5 and 4.6), namely one for each of the 15 combinations, with its respective results provided in the table of results, table 4.2. With regards to the plots, one could see an indicator that the more sparsely the points were distributed in the plots, the harder it was for K-means to create good clusters, resulting in worse scores. This could be exemplified by figure 4.5f or figure 4.6f. However, this was not a strong correlation, such that more testing with different datasets is advised.

With regards to the table of results, one could see one important factor. All of the highest scoring 2-variable combinations contained the SIFT feature representation as one of its variables. This could make the argument for SIFT being a strong feature extraction method for this specific project.

5.1.4. Using a CNN - Test 5

- Testing a CNN which was on trained on the complete dataset, spanning 3 waste cycles, and using a test set of 250 images from new waste cycles.

The CNN-test scored a correctness of 0.495 when trained on the complete dataset and tested on the new dataset of 250 images. Considering it is claimed to be the state of the art solution to object detection [44], this might seem like a poor result. However, the test with 250 images showed similar results to that of the test with 20 images that was made as an example in chapter 4 - Tests and results. From that example, one could see that

registering the state "25%" and "75%" scored particularly low. One reason for this could be that the labeling done on the training set and test set were somewhat different. Since this was labeled by eye sight, it could be the case that what was labeled as "25%" in the test set was more similar to what was being labeled as "50%" in the training set, as can be seen on the images of the example test in figure **??**. One could thus argue that averaging the correctness of all states is unfair when the correctness of state "100%" seem to be higher. This could be seen as unfair due to the reason that this CNN was implemented primarily to detect when a container is *full* in order to alert those responsible of picking up and emptying the container. However, this was not the goal of the thesis. One of the goals stated in the introduction was to investigate how successful the different methods were in *correctly identifying each class/filling rate*. Therefore, the total correctness was seen as a fitting measure.

Another reason for possibly obtaining comparatively low correctness scores on the CNN, is due to the size of the dataset. State of the art CNNs with results upwards of 95% correct identification are typically trained with a much larger dataset, at times surpassing millions of labeled images [8]. Considering the dataset given for this thesis contained just below 5000 images, it can be assumed the neural network simply didn't have enough data for training in order to provide good predictions.

5.2. Discussion of methods

With the results of the tests discussed, this part will focus on the methods used in this thesis and the assumptions made, justifying their use.

5.2.1. Feature extraction - representations and assumptions.

Identifying filling rate through feature representation from feature extraction methods is the basis for attempting to classify the containers filling rate through unsupervised learning methods. One of the tasks presented in chapter 1 - Introduction was:

> *Search for feature extraction methods that fit this specific project.*

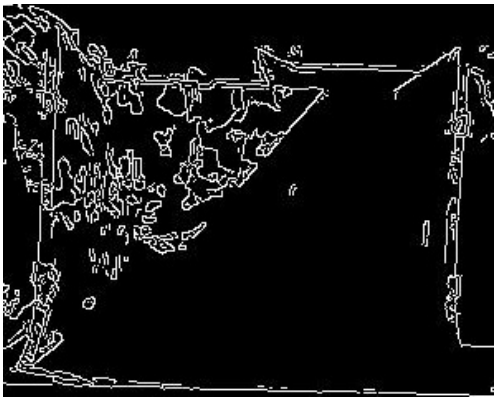
As Python was chosen as the language of implementation due to its rich machine learning libraries, the search for feature extraction methods that fit this specific method was conducted on a trial and error basis. Methods were found in publicly available libraries, like "*openCV*" or as open source on Github. Five of the six feature extractions were methods found online that seemed to perform well in this specific project, and one was developed by the author of this thesis. However, how does a feature extraction method "fit this specific project"? The goal of finding these methods, were to see if some types of features detected could tell anything about filling rate of the waste in the containers. After experimentation of different feature methods, some ideas came about as how the fill rate of a container could be represented. Two key observations laid the foundation for the search and use of given feature extraction methods. One observation was that the walls of the container were a lot more prevalent and showing when the container contained little waste. The other observation was that the walls had much more coherent texture with regards to color and edges, opposed to the wood waste. This became especially noticeable through experimentation with the five imported feature extraction methods.

However, finding feature extraction methods that could potentially describe the fill rate of waste in containers, was rapidly followed by the problem of *how* this should be described. As one of the goals were automatically classify the filling rates by cluster analysis of the features, the features needed to be able to be grouped in euclidean space, in a manner that was eligible for the use of e.g. K-means. The output of some of the feature extraction methods, such as Hough Line Transform and Histogram of Oriented Gradients, were a 2-dimensional matrix at the same size of the input image. As an input image was typically $299 \times 299 = 89,4901$ pixels, using the the output of a Hough Line Transform in K-means clustering, being a 2D matrix of size 299×299, would mean that one would search for similarity in terms of euclidean distance in 89,401-dimensional space. At this point it is safe to assume that one would suffer from the curse of dimensionality, and this type of clustering would likely not bear fruit.

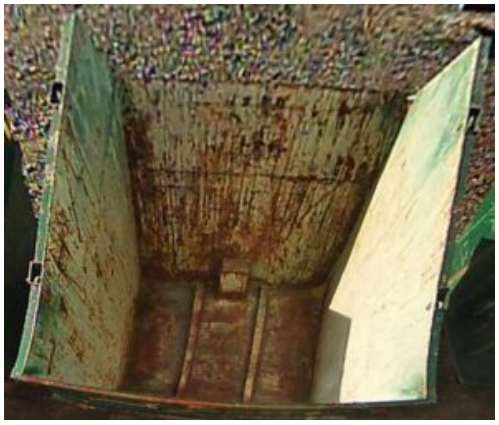
Thus, it was thought of a way that could still represent what the features were showing, but in a manner that was compatible with the clustering operation in order automatically classify the data. If the result of each feature extraction method could be represented as a single number, then one could investigate clustering with comparatively few dimensions.

However, the assumption that a single numerical representation of a feature extraction method could be a good descriptor is debatable. E.g. taking the mean pixel intensity of a feature transformed image, is highly reliant on the correctness of the assumption that the container wall would register as black and the waste as white. This is the point where noise and environment greatly affects this assumption. One container could have spotless walls, not triggering any of the feature detection methods and provide a black substrate of that region. Another container could be rife with rust or various spots, unintentionally triggering edge detection. Another problem is weather and time. Shadows might appear on the containers at various times throughout the day, again triggering certain feature detection methods and not registering as an empty wall. The differing illumination of the container by the sun could also affect these methods in a similar manner. These elements of noise was detected when working with the provided dataset. Thus, a container could have the same fill rate when looked at by eye sight, but have distinct differences in the numerical values of the feature vectors for two images taken with different environmental occurrences. This is exemplified in figure 5.1 shown beneath, where these environmental factors triggers in the Hough Line Transform. This is thought to be the main reason for the variance/spread of data points which contains the same true label in the tests, seen either on the PCA-plots or 2-variable test plots in chapter 4 - Tests and results.

(a) Shadow on container.



(b) Hough Line Transform on shadows.



(c) Very rusty container.

(d) Hough Line Transform on rust spots.

Figure 5.1.: Effects of environmental noise.

5.2.2. Unsupervised methods as means of classification

The overarching goal was to see whether clusters formed by unsupervised learning could work as a labeling mechanism. Would data points containing the same true label/fill rate also be grouped together in cluster analysis? If so, would cluster adherence be a sufficient labeling mechanism for data that is unlabeled and thus discard the need for manual labeling? At first, different unsupervised learning methods were considered. Agglomerative clustering was discarded on the basis of how the clustering algorithm worked. The bottom-up structure of agglomerative clustering made too few assumptions on how the data ought to be grouped, making it seemingly incompatible with this project. For partitioned clustering, affinity propagation was discarded on the basis of computational cost and K-means

was selected as best fit.

K-means works by clustering data based on the data's similarity. The similarity measure used in the testing, also being the most commonly used, was the euclidean squared distance between points. The K-means method was thus selected as the method for classifying based on cluster adherence. This was found to be quite possible, however highly reliant on the feature representations it performing cluster analysis on. It is therefore assumed that K-means as method for unsupervised classification is highly plausible, given good feature representations, able to distinctly differ in value based on its fill rate, is fed into the cluster analysis.

5.2.3. Comparison and utility of the different solutions.

Looking at the table of results, many of the tests results were in close proximity of one another. It is seen that for testing on the complete dataset, both the supervised and the best unsupervised solutions achieved approximately similar results. The CNN scored a correctness of 0.495 while the PCA-reduced K-means scored a correctness of 0.574. From a strictly numerical perspective, two observations and statements can be made. First being that both solutions seem to achieve approximately equal results from what is measured from the tests. If one were to choose a solution based on solely this statistic, the unsupervised learning solution would be at a slight advantage. The other observation is that neither solution would give those whom implements the solution high confidence regarding correctly classifying filling rate/classes. With a 40-50% error rate, a direct implementation of this could be assumed to not please a customer.

In order to say anything regarding the utility of the different methods with regards to this project, one has to distinguish between the utility of the different methods for this specific project *at this specific time in the project*, and for the future of this specific project. It has already been discussed that at this specific time, neither solution could be assumed to be satisfactory for a customer and readily available for implementation with such a big portion of the dataset not correctly classified. Finding this provides some key insights. In the case of supervised learning, there were two potential problems found. One being that the size of the dataset simply might have been too small to successfully train the CNN. The other was that one has to be careful during training, being meticulous with correctly labeling the

different classes. It was observed that the test set would often mistake a container of being 25% full with being 50%, and so on. For this specific project, it could mean the CNN could register a 100% full container too early, having the suppliers of the containers pick up and swap with an empty container unnecessarily often. In case of unsupervised learning, there also seemed to be two potential main problems. The first one, being perhaps the biggest, is the problem of feature detection and correct representation. Having to figure out a way to represent the states using different feature extraction methods proved difficult, and was assumed to be heavily influenced by environmental factors.

In a world where noise is always present, finding ways to suppress it can be difficult. In object detection in images, this is particularly difficult. However, this might prove the utility of using many different feature extraction methods and representations. If one type of feature extraction method is particularly prone to some type of environmental noise, another method might not be, thus reducing the potential error.

Another potential problem of unsupervised learning as means of classification, is the uncertainty of whether or not the data fall into the correct cluster. With noisy images, the assumptions of feature representation for the different feature extraction methods used in this thesis can fall short. A noisy empty container might register equal feature representation numbers as a container with much higher fill rate. The clustering algorithms do not take this into consideration, and is assumed to be the main reason for somewhat low correctness score and the overlap of classes.

What utility will the methods investigated provide for the future of the project of automating the container pick up process? And can this knowledge be transferred for something outside this specific project? Based on the knowledge of CNNs, it is safe to assume that obtaining more labeled images and further training the neural network will improve its accuracy of correctly detecting classes/filling rates. Even the erroneous labeling of classes done by the human designer is thought to even out at as the training set and test set increases. If one assumes that the human that performs the labeling classifies images with a normal distribution, the correct mean of the given class would eventually become prominent, and the neural network would adjust its weights accordingly. However, it is found out that implementing such a technology requires time and a sufficient amount of data. If time and data-requirements are not critical, implementing a CNN for classification and

detection can provide good results. This allows for a potentially wide range of use with regards to classification and detection in images, when constraints to time and data are low. In the opposite case, where constraints to time and data are high, the unsupervised learning methods provide certain advantages. Time spent on labeling is not needed, and the amount of data can possibly be drastically lower, opposed to implementing a CNN. However, this demands highly functioning feature representations, which can be hard to implement. Implementation of classification and detection via unsupervised learning methods is therefore advised for domains where noise is not a large factor, or highly prevalent, where as the neural network seem to obtain the important features through rigorous training and adjust its parameters to downsize the weights relating to noise. However, the feature representations made for this thesis can be assumed to work for any process where a *thing* with rough texture grows in size relative to a somewhat sterile environment, if the methods of this thesis is used for another project. However, some feature extraction and -representation tweaking ought to be expected.

6. Conclusion

6.1. Conclusion

This thesis is an investigation of whether unsupervised learning methods could be used as means of classifying filling rate of waste within a waste container by feature extractions from images of the container, and see how it compared to the use of a supervised machine learning method, namely a convolutional neural network (CNN). One of the main challenges of implementing a CNN, are the resources needed to provide a labeled dataset in order to train the CNN. The dataset needs to be sufficiently large in order for the CNN to be trained properly for making correct predictions, and the dataset needs to be labeled in order for the CNN to understand what it is training towards. This can be time-consuming and tedious work.

In order to bypass the labeling scheme, an unsupervised learning method called K-means was used to classify waste levels. K-means is a clustering algorithm which groups data based on their proximity in euclidean space. In order to successfully group images at various waste levels in euclidean space, a way to translate images into numeric representations was necessary. This was done by performing various feature extraction methods, and represent the output of these methods by a single number. For three of the feature extraction methods used (Histogram of Oriented Gradients, Hough Line Transform and Histogram Threshold), the mean pixel intensity of the transformed images would represent the amount of waste in the container. For two of the methods (Scale-Invariant Feature Transform and Speeded-Up Robust Features), the amount of keypoint localizations made by the feature extraction methods were the number that would represent the waste level. For the last method (Hough-distance), an algorithm was created for this thesis, based on the output of the Hough Line Transform. Put simply, this algorithm was made to search for how long a counter could travel through each row of the Hough Line-transformed image without

detecting indicators of waste. The goal for this algorithm was the higher the counter reached, the less waste was detected, thus also being a numeric representation of waste level. These six feature extraction methods were used on each image in the dataset provided, and a feature vector was created for each image. This feature vector consisted of six elements, which were the six numeric representations of the feature extractions.

Four types of tests were conducted in order to investigate clustering on the feature vectors as means of classifying waste levels. One test performed a six-dimensional K-means analysis with the full feature vector on the complete dataset. Another test performed K-means in two dimensions on all possible variations of two of the six feature vector elements. A third test performed Principal Component Analysis on the feature vectors, reducing the dimensionality from six to two, while retaining a large degree of the variance from the set of complete feature vectors. Then K-means was performed on the projected points within the principal axes. The fourth test was to see whether drastically reducing the size of the dataset would provide similar results compared to using the complete dataset. The method that scored highest of the three previous tests were chosen as the method to test with a reduced dataset. To assert statistical validity of the claim that using a small dataset yields similar results, a population proportion test was also conducted. These four tests provided three main findings.

- Performing PCA on the feature vectors provided the highest score for correct classification when using K-means. Clustering of projected points on a hyperplane made by PCA seemed to be the best way to both utilize the effect of using several feature representations, and group points in euclidean space without suffering from the effects of the curse of dimensionality.

- When using a small subset of only 25 images out of the 4802 images in the complete dataset, approximately similar score was achieved when using the small subset of images compared to the complete dataset. This provides the possibility of implementing an image classification system with drastically reduced need of images compared to a CNN.

- When testing all two-variable combinations of the six feature representations, it showed that SIFT appeared in all of the highest scoring tests, providing the assumption that this is a good feature descriptor to use for this specific project.

This thesis aimed to compare the unsupervised learning solution to a supervised learning solution. An open source solution for creating a custom CNN, called Inception V3, was implemented. It was trained on the dataset of 4802 images, and trained on a new set of 250 images in order to see its success in classifying labels. The test scores of both supervised and unsupervised solutions in and of themselves were not particularly high. The CNN correctly classified classes at a success rate of 49.5%, and the best unsupervised learning method tested on the dataset of 4802 images resulted in a success rate of 57.4%. In order to achieve better scores, it was assumed that the CNN needed more data to train on, and the unsupervised learning method needed better feature representation descriptors.

6.2. Further work

Most of the work for this thesis revolved around finding feature representations to successfully classify waste levels by the use of K-means. It was shown that the feature extraction methods were particularly prone to give bad results in the presence of environmental noise such as shadows and rust spots. It would be interesting to keep working on finding better feature representations. The amount of keypoints localized from SIFT seemed to correlate well with true labels, whereas the mean pixel intensity from Hough Line Transform and HOG were not particularly successful. Another potential feature descriptor based on the Hough Line Transform has come to mind at the final stages of this thesis. As it is seen in figure 5.1 in chapter 5, the Hough Line Transform is prone to detecting noise on the container. However, the noise such as rust spots or shadow spots are mostly a collection of really small dots on the transformed image. When planks are detected by the Hough Line Transform, their lines are generally much longer. One potential way of discarding noise could then be to create an algorithm which only accepts Hough Lines over a certain length, such that only the planks and the container is detected. A rough estimator could then potentially be to count the amount these lines, and possibly combine it with the Hough-distance algorithm to distinguish between many lines in the bottom of the container and when the container is full.

It is also thought of a way to create an unsupervised learning solution that could apply for any similar type of image classification problem, where it is created a bank of feature extraction methods and descriptors, and experiment on the data at hand. When features

and descriptors that provide a good fit for the data, these are selected as the feature vectors and are classified through a combination of PCA and K-means.

With regards to further noise reduction, it is advised to experiment with the use of "autoencoders". A neural network is made to learn efficient data encodings in an unsupervised manner, meaning one is reducing an input while retaining its features and extracting the input back to full size while eliminating noise. This could potentially be a good preprocessing step before implementing feature extractions and feature representations. This was not experimented with in this thesis due to time constraints.

Potentially, the combination of both unsupervised learning and supervised learning could make the process of automating container exchange less time consuming. If unsupervised learning is used to classify images and splitting them into labeled groups, the designer could go through each labeled and simply remove each image that was incorrectly placed in that group. Then time spent on labeling could be reduced, and the newly sorted labeled groups could be fed into a CNN.

Other sources of inputs are also advised to investigate, instead of images. The use of a distance measure, with infra red or sound waves, could also provide information of the containers filling rate. This could be combined with K-means for automatic classification, or a threshold value could be set for when the container was full, if that there were the only objective. However, these technologies were considered by Acando when they began their project, and monitoring and classification via camera and supervised learning was chosen on the basis of cost and their field of expertise. This is the reason why any other technologies has not been considered in this thesis.

Lastly, it is advised to implement a way of automatically detecting the container walls from the raw image, and crop it accordingly. Noise such as gravel from the ground surrounding the container was picked up by several feature extraction methods, adding uncertainty to the numerical feature representations. A cleaner result with regards to the feature representations is expected with a better crop of the container.

Bibliography

[1] Machine Learning Guru Collective. Image filtering. http://machinelearninguru.com/computer_vision/basics/convolution/image_convolution_1.html.

[2] Satya Mallick. Histogram of oriented gradients. https://www.learnopencv.com/histogram-of-oriented-gradients/.

[3] OpenCV Collective. Blob detection opencv. http://shoebill.info/images/b/blob-detection-opencv.html.

[4] Robert Fisher. Laplacian/laplacian of gaussian. https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm.

[5] Debao Zhou. Machine vision and vision based control. http://www.d.umn.edu/~dzhou/ME4060/ME%204060%20Chapter%2014%20Edge%20detection.pdf.

[6] Jose Portilla. A beginner's guide to neural networks in python and scikit learn 0.18. https://www.springboard.com/blog/beginners-guide-neural-network-in-python-scikit-learn-0-18/.

[7] A. Ng. Neural networks and deep learning. https://www.coursera.org/learn/neural-networks-deep-learning.

[8] Google Brain Team collective. Inception v3 model. https://bit.ly/2R0YalH.

[9] Google Net research collevtive. A simple guide to the versions of the inception network. https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202.

[10] Tensorflow collective. Image recognition via inception v3. https://www.tensorflow.org/tutorials/images/image_recognition, addendum =.

[11] Pang-Ning Tan. *Introduction to data mining*, chapter 8, page 491. Always learning. Pearson, Harlow, pearson new international edition. edition, 2014.

[12] Wikipedia Commons. Hierarchical clustering dendrogram. https://commons.wikimedia.org/wiki/File:Hybrid_clustering_case_study_dendogram.png, addendum =.

[13] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651 – 666, 2010. Award winning papers from the 19th International Conference on Pattern Recognition (ICPR).

[14] Ritchie Vink. Algorithm breakdown: Affinity propagation. https://www.ritchievink.com/blog/2018/05/18/algorithm-breakdown-affinity-propagation/.

[15] Alboukadel Kassambara. Determining the optimal number of clusters: 3 must know methods. http://www.sthda.com/english/articles/29-cluster-validation-essentials/96-determining-the-optimal-number-of-clusters-3-must-know-methods/#elbow-method.

[16] The Data Science Lab. Finding the k in k-means clustering. https://datasciencelab.wordpress.com/tag/gap-statistic/.

[17] Jonas Lien. Google trends - comparing languages for machine learning. https://trends.google.com/trends/explore?date=2012-01-01%202017-08-31&q=Python%20machine%20learning,R%20machine%20learning,Octave%20machine%20learning,Matlab%20machine%20learning,Julia%20machine%20learning.

[18] Nick Heath. Which is the fastest growing programming language? hint, it's not javascript. https://www.techrepublic.com/article/which-is-the-fastest-growing-programming-language-hint-its-not-javascript/.

[19] Willi Richert and Luis Pedro Coelho. *Building Machine Learning Systems with Python*, chapter 1, page 13. PACKT, 2016.

[20] Anaconda. Anaconda distribution. https://www.anaconda.com/distribution/.

[21] Peter Storm. Jupyter notebook: A.

[22] Joel Ratasby Uzi Chester. Machine learning for image classification and clustering using universal distance measure. https://www.ariel.ac.il/sites/ratsaby/Publications/PDF/sisap13.pdf.

[23] L. G. Roberts. *Pattern Recognition With An Adaptive Network*, pages 66–70. IRE International Convention Record, 1960.

[24] John K. Tsotsos Alexander Andreopoulos. 50 years of object recognition: Directions forward. page 2. Department of Computer Science and Engineering Centre for Vision Research, York University,.

[25] John Canny. A computational approach to edge detection. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, PAMI-8, 1986.

[26] Dillip Krishnan Edward H. Adelson Phillip Isola, Daniel Zoran. Learning visual groups from co-occurencs in space and time.

[27] Dhruv Batra Jianwei Yang, Devi Parikh. Joint unsupervised learning of deep representations and image clusters.

[28] Levent Ertöz Michael Steinbach and Vipin Kumar. The challenges of clustering high dimensional data. https://www-users.cs.umn.edu/~ertoz/papers/clustering_chapter.pdf, addendum =.

[29] Lynne J. Williams Hervé Abdi. Principal component analysis. https://www.utdallas.edu/~herve/abdi-awPCA2010.pdf.

[30] G. E. Hinton A. Krizhevsky, I. Sutskever. Imagenet: classification with deep convolutional neural networks. *NIPS*, pages 1106–1114, 2012.

[31] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *ECCV*, 2014.

[32] T. Darrell J. Malik R. Girshick, J. Donahue. Rich feature hierarchies for accurate object detection and semantic segmentation. *CVPR*, 2014.

[33] Jost Tobias Springenberg Martin Riedmiller Thomas Brox Alexey Dosovitskiy, Philipp Fischer. Discriminative unsupervised feature learning with exemplar convolutional neural networks.

[34] R. Socher L.-J. Li K. Li L. Fei-Fei J. Deng, W. Dong. Imagenet: A large-scale hierarchical image database. *CVPR*, 2009.

[35] C. K. Williams-J. Winn A. Zisserman M. Everingham, L. Gool. The pascal visual object classes (voc) challenge. *IJCV*, 88(2):303–338, 2010.

[36] Bill Triggs Navneet Dalal. Histograms of oriented gradients for human detection.

[37] Wikipedia contributors. Sobel operator. `https://en.wikipedia.org/wiki/Sobel_operator`.

[38] OpenCV collective. Introduction to sift (scale-invariant feature transform). `https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html`.

[39] M. Stephens C. Harris. A combined corner and edge detector. `http://www.bmva.org/bmvc/1988/avc-88-023.pdf`.

[40] Peter E. Hart Richard O. Duda. Use of the hough transformation to detect lines and curves in pictures.

[41] Bernard Marr. What is the difference between artificial intelligence and machine learning? `https://www.forbes.com/sites/bernardmarr/2016/12/06/what-is-the-difference-between-artificial-intelligence-and-machine-learning/`.

[42] Alvin C. Rencher. *Methods of multivariate analysis*, chapter 15, page 502. Wiley Series in Probability and Statistics. Wiley, Hoboken, N.J., 3rd ed. edition, 2012.

[43] Collective authors of Wikipedia. Supervised learning.

[44] Hamed Habibi Aghdam and Elnaz Jahani Heravi. *Convolutional Neural Networks*, pages 85–130. Springer International Publishing, Cham, 2017.

[45] Google Brain Team collective. Inception v3 model. `https://www.kaggle.com/google-brain/inception-v3`.

[46] Pang-Ning Tan. *Introduction to data mining*, chapter 8, page 487. Always learning. Pearson, Harlow, pearson new international edition. edition, 2014.

[47] Michael R. Berthold and Frank H¨oppner. On clustering time series using euclidean distance and pearson correlation.

[48] Trevor Hastie Robert Tibshirani, Guenther Walther. Estimating the number of clusters in a data set via the gap statistic. *J. R. Statist. Soc. B (2001)*, 63(2):411 – 423, 2001. Standford University, USA.

[49] Sunil Ray. What are the common methods to perform dimension reduction? `https://www.analyticsvidhya.com/blog/2015/07/dimension-reduction-methods/`.

[50] Wikipedia contributors. The curse of dimensionality. `https://en.wikipedia.org/wiki/Curse_of_dimensionality`.

[51] Damien François Michel Verleysen. The curse of dimensionality in data mining and time series prediction. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.2646&rep=rep1&type=pdf`.

[52] Rachit Kumar Agrawa. Machine learning programming languages — which is the best and why. `https://medium.com/@UdacityINDIA/machine-learning-programming-languages-why-is-the-best-and-why-56f9f370cb99`.

[53] George Anadiotis. The best programming language for data science and machine learning. `https://www.zdnet.com/article/the-best-programming-language-for-data-science-and-machine-learning/`.

[54] Wikipedia. R (programming language). `https://en.wikipedia.org/wiki/R_(programming_language)`.

[55] Jason Brownlee. What is r. `https://machinelearningmastery.com/what-is-r/`.

[56] Viral Shah Jeff Bezanson, Stefan Karpinski and Alan Edelman. Why we created julia. `https://julialang.org/blog/2012/02/why-we-created-julia`.

[57] Alex Arslan. Julia version 1.0.0. `https://github.com/JuliaLang/julia/releases/tag/v1.0.0`.

[58] David Robinson. The incredible growth of python. https://stackoverflow.blog/2017/09/06/incredible-growth-python/.

[59] Joel Spolsky and Jeff Atwood. About stack overflow. https://stackoverflow.com/company.

[60] Justin Stoltzfus. Why is python so popular in machine learning? https://www.techopedia.com/why-is-python-so-popular-in-machine-learning/7/32881.

[61] Justin Stoltzfus. pandas: powerful python data analysis toolkit. http://pandas.pydata.org/pandas-docs/stable/.

A. Code by author

```
#Imports

import numpy as np
import cv2
import matplotlib
from matplotlib import pyplot as plt
from sklearn import cluster
from skimage.feature import hog
from skimage import data, exposure, io, img_as_float, img_as_ubyte
from mpl_toolkits.mplot3d import Axes3D
from scipy import ndimage as ndi
from skimage.feature import shape_index
from skimage.draw import circle
from skimage.filters import threshold_otsu
import pandas as pd
import os
import glob
import re
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
import math
```

```
#Feature extraction methods

#SIFT method

def SIFT(path):
    img = img_or_path(path)
    gray= cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    sift = cv2.xfeatures2d.SIFT_create()
    kp = sift.detect(gray,None)
    num = len(kp)

    return num

#SURF method

def SURF(path):
    if isinstance(path, str):
        img = cv2.imread(path,0)
    else:
        img = path
    surf = cv2.xfeatures2d.SURF_create(2000)
    kp, des = surf.detectAndCompute(img,None)
    num = len(kp)
    return num

#Hough line transform

def Hough(path):
    img = img_or_path(path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    edges = cv2.Canny(gray, 75, 150)

    meanEdges = np.asarray(edges)
    num = np.mean(meanEdges)

    return num
```

```
72 def Hough_dist(path):
73     img = img_or_path(path)
74     gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
75     edges = cv2.Canny(gray, 75, 150)
76     edgy = np.asarray(edges)
77     totalCount = 0
78 
79     for i in range(len(edgy)):
80         AccumulatedCount = 0
81         tempCount = 0
82         for j in range(len(edgy)):
83             if edgy[i][j] == 255:
84                 AccumulatedCount += tempCount ** 1.5
85                 tempCount = 0
86             else:
87                 tempCount += 1
88         totalCount += AccumulatedCount
89         AccumulatedCount = 0
90     return totalCount/100000
91 
92 #Histogram thresholding
93 
94 def Hist_thresh(path):
95     if isinstance(path, str):
96         image = io.imread(path,as_gray=True)
97     else:
98         img = path
99         image = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
100    thresh = threshold_otsu(image)
101    for i in range(len(image)):
102        for j in range(len(image[i])):
103            if image[i][j] < thresh:
104                image[i][j] = 0
105            else:
106                image[i][j] = 255
107 
108    num = np.mean(image)
109    return num
110 
111
```

```
112 #HOG function:
113
114 def HOG(path):
115     image = img_or_path(path)
116     fd, hog_image = hog(image, orientations=8, pixels_per_cell=(16, 16),
    cells_per_block=(1, 1), visualize=True, multichannel=True)
117
118     hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0,
    10))
119     num = np.mean(hog_image_rescaled)
120
121     return num
122
123
124 #Help-function deciding whether input is image or a path to an image
125
126 def img_or_path(obj):
127     if isinstance(obj, str):
128         img = cv2.imread(obj)
129     else:
130         img = obj
131     return img
132
133
134 #Image cropping & preprocessing:
135
136 def Image_crop(path):
137     img = img_or_path(path)
138     crop_img = img[80:290, 20:280]
139     return crop_img
140
141 def Image_preproc(path):
142     img = img_or_path(path)
143     crop_img = Image_crop(img) #Cropping image
144     kernel = np.ones((5,5),np.float32)/25
145     dst = cv2.filter2D(crop_img,-1,kernel)
146     return dst
147
148
149
```

```
150 #Perform all feature transformations on an image and return its descriptive
        value
151
152 def array_func(path):
153     arr = []
154     thresh = Hist_thresh(path)
155     arr.append(thresh)
156     hough = Hough(path)
157     arr.append(hough)
158     sift = SIFT(path)
159     arr.append(sift)
160     surf = SURF(path)
161     arr.append(surf)
162     hog = HOG(path)
163     arr.append(hog)
164     houghDist = Hough_dist(path)
165     arr.append(houghDist)
166
167     return arr
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
```

```
189
190 #Calling preprocessing and feature transformations of all images in a folder
191
192 def folder_preproc(folderPath):
193
194     numbers = re.compile(r'(\d+)')
195     def numericalSort(value):
196         parts = numbers.split(value)
197         parts[1::2] = map(int, parts[1::2])
198         return parts
199
200     img_dir = folderPath
201     data_path = os.path.join(img_dir,'*g')
202     files = glob.glob(data_path)
203     arr = []
204     for f1 in sorted(files, key=numericalSort):
205
206         img_pre = Image_preproc(f1)
207         rows = array_func(img_pre)
208         arr.append(rows)
209
210     mat = np.matrix(arr)
211     df = pd.DataFrame(mat, columns=['Threshold','Hough','SIFT','SURF', 'HOG'
    , 'Hough-dist'])
212     return df
213
214 #Getting image preprocessed and feature-transformed images labeled and
    united in a single dataframe
215
216 def label_dataframe(folderPath, label):
217
218     df = folder_preproc(folderPath)
219     z = []
220     for i in range(len(df)):
221         z.append(label)
222     y = pd.DataFrame(z, columns=['target'])
223
224     labeledDf = pd.concat([df, y], axis=1)
225
226     return labeledDf
```

```
227
228 #Gathering all feature extracted values and concatenating into single
    DataFrame
229
230 def final_DFs_strLabel():
231     df1 = label_dataframe("path-to-images-labeled-0%","0%")
232     df2 = label_dataframe("path-to-images-labeled-25%","25%")
233     df3 = label_dataframe("path-to-images-labeled-50%","50%")
234     df4 = label_dataframe("path-to-images-labeled-75%","75%")
235     df5 = label_dataframe("path-to-images-labeled-100%","100%")
236
237     fin1 = pd.concat([df1, df2], ignore_index=True)
238     fin2 = pd.concat([fin1, df3], ignore_index=True)
239     fin3 = pd.concat([fin2, df4], ignore_index=True)
240     finalDF = pd.concat([fin3, df5], ignore_index=True)
241     return finalDF
242
243 def final_DFs_intLabel():
244     df1 = label_dataframe("path-to-images-labeled-0%",0)
245     df2 = label_dataframe("path-to-images-labeled-25%",1)
246     df3 = label_dataframe("path-to-images-labeled-50%",2)
247     df4 = label_dataframe("path-to-images-labeled-75%",3)
248     df5 = label_dataframe("path-to-images-labeled-100%",4)
249
250     fin1 = pd.concat([df1, df2], ignore_index=True)
251     fin2 = pd.concat([fin1, df3], ignore_index=True)
252     fin3 = pd.concat([fin2, df4], ignore_index=True)
253     finalDF = pd.concat([fin3, df5], ignore_index=True)
254     return finalDF
255
256
257
258
259
260
261
262
263
264
265
```

```
#Calculating Principal Component Analysis and storing in a dataframe.

def calculate_PCA(intOrStr):

    df = intOrStr
    features = ['Threshold', 'Hough', 'SIFT', 'SURF', 'HOG', 'Hough-dist']
    # Separating out the features
    x = df.loc[:, features].values
    # Separating out the target
    y = df.loc[:,['target']].values
    # Standardizing the features
    x = StandardScaler().fit_transform(x)

    pca = PCA(n_components=2)
    principalComponents = pca.fit_transform(x)
    principalDf = pd.DataFrame(data = principalComponents
                 , columns = ['principal component 1', 'principal component
   2'])

    finalDf = pd.concat([principalDf, df[['target']]], axis = 1)
    print(pd.DataFrame(pca.components_,columns=['Threshold', 'Hough', 'SIFT'
   , 'SURF', 'HOG', 'Hough-dist'],index = ['PC-1','PC-2']))
    print(pca.explained_variance_ratio_)

    return finalDf
```

```
304 #PCA plotting
305
306 def plot_PCA(dataframe):
307
308     finalDf = dataframe
309
310     PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
311       svd_solver='auto', tol=0.0, whiten=False)
312
313     fig = plt.figure(figsize = (8,8))
314     ax = fig.add_subplot(1,1,1)
315     ax.set_xlabel('Principal Component 1', fontsize = 15)
316     ax.set_ylabel('Principal Component 2', fontsize = 15)
317     ax.set_title('2 component PCA', fontsize = 20)
318     targets = ['0%','25%','50%','75%','100%']
319     colors = ['r', 'g', 'b', 'c', 'm']
320     for target, color in zip(targets,colors):
321         indicesToKeep = finalDf['target'] == target
322         ax.scatter(finalDf.loc[indicesToKeep, 'principal component 1']
323                    , finalDf.loc[indicesToKeep, 'principal component 2']
324                    , c = color
325                    , s = 50)
326     ax.legend(targets)
327     ax.grid()
328
329
330 #K-means clustering and measuring correctness
331
332 class K_Means:
333     def __init__(self, k =5, tolerance = 0.0001, max_iterations = 500):
334         self.k = k
335         self.tolerance = tolerance
336         self.max_iterations = max_iterations
337         self.labelClasses = {}
338
339     def fit(self, data):
340
341         self.centroids = {}
342         centroidInterval = len(data)/k
343
```

```
344         #initialize the centroids evenly throughout the dataset
345         for i in range(self.k):
346             self.centroids[i] = data[centroidInterval*i]
347 
348         #begin iterations
349         for i in range(self.max_iterations):
350             self.classes = {}
351             for i in range(self.k):
352                 self.classes[i] = []
353 
354             #find the distance between the point and cluster; choose the
    nearest centroid
355             for features in data:
356                 distances = [np.linalg.norm(features - self.centroids[
    centroid]) for centroid in self.centroids]
357                 classification = distances.index(min(distances))
358                 self.classes[classification].append(features)
359 
360             previous = dict(self.centroids)
361 
362             #average the cluster datapoints to re-calculate the centroids
363             for classification in self.classes:
364                 self.centroids[classification] = np.average(self.classes[
    classification], axis = 0)
365 
366             isOptimal = True
367 
368             for centroid in self.centroids:
369 
370                 original_centroid = previous[centroid]
371                 curr = self.centroids[centroid]
372 
373                 if np.sum((curr - original_centroid)/original_centroid *
    100.0) > self.tolerance:
374                     isOptimal = False
375 
376             #break out of the main loop if the results are optimal, ie. the
    centroids don't change their positions much(more than our tolerance)
377             if isOptimal:
378                 self.labelClasses = self.classes
```

```
                break


    def pred(self, data):
        distances = [np.linalg.norm(data - self.centroids[centroid]) for
    centroid in self.centroids]
        classification = distances.index(min(distances))
        return classification

    def get_class(self):
        return self.classez

def cluster_correctness(cluster_class):
    match = 0
    not_match = 0
    c = cluster_class

    for x in range(0,5):
        tempClusterArr = c[x]
        tempClusterArr = np.array(tempClusterArr)
        for i in range(len(tempClusterArr)):
            if tempClusterArr[i][6] == x:
                match += 1
            else:
                not_match += 1

    return match/(match+not_match)


def run_Kmeans(df):

    X = df.values #returns a numpy array of feature vectors
    km = K_Means4(5)
    km.fit(X)
    cluster_class = km.get_class()
    cluster_correctness(cluster_class)
```

```
418 #PCA matching compared to labeled set
419
420 def PCA_match(PCA_set):
421     p = PCA_set #Dataframe with columns "principal component 1" and "
    principal component 2"
422     minmax = p.loc[:,'principal component 1'].values
423     minVal = min(minmax)
424     maxVal = max(minmax)
425     valRange = (max(minmax)-min(minmax))/5
426     match = 0
427     not_match = 0
428     for i in range(0,1):
429         subset = p[p.target==i]
430         subset2 = subset.loc[:,'principal component 1'].values
431         for j in range(len(subset2)):
432             if ((subset2[j] > (minVal)) &  (subset2[j] < (minVal+(valRange))
    )):
433                 match += 1
434             else:
435                 not_match += 1
436     for i in range(1,4):
437         subset = p[p.target==i]
438         subset2 = subset.loc[:,'principal component 1'].values
439         for j in range(len(subset2)):
440             if (subset2[j] > (maxVal+(valRange*i))) & (subset2[j] < (maxVal
    +(valRange*(i+1)))):
441                 match += 1
442             else:
443                 not_match += 1
444
445     for i in range(4,5):
446         subset = p[p.target==i]
447         subset2 = subset.loc[:,'principal component 1'].values
448         for j in range(len(subset2)):
449             if subset2[j]>(minVal+(valRange*3)):
450                 match += 1
451             else:
452                 not_match += 1
453     return match/(match+not_match)
```

B. retrain.py

```
# Copyright 2015 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================

# NOTICE: This work was derived from tensorflow/examples/image_retraining
# and modified to use TensorFlow Hub modules.

# pylint: disable=line-too-long
r"""Simple transfer learning with image modules from TensorFlow Hub.

This example shows how to train an image classifier based on any
TensorFlow Hub module that computes image feature vectors. By default,
it uses the feature vectors computed by Inception V3 trained on ImageNet.
See https://github.com/tensorflow/hub/blob/master/docs/modules/image.md
for more options.

The top layer receives as input a 2048-dimensional vector (assuming
Inception V3) for each image. We train a softmax layer on top of this
representation. If the softmax layer contains N labels, this corresponds
```

```
to learning N + 2048*N model parameters for the biases and weights.

Here's an example, which assumes you have a folder containing class-named
subfolders, each full of images for each label. The example folder flower_photos
should have a structure like this:

~/flower_photos/daisy/photo1.jpg
~/flower_photos/daisy/photo2.jpg
...
~/flower_photos/rose/anotherphoto77.jpg
...
~/flower_photos/sunflower/somepicture.jpg

The subfolder names are important, since they define what label is applied to
each image, but the filenames themselves don't matter. (For a working example,
download http://download.tensorflow.org/example_images/flower_photos.tgz
and run  tar xzf flower_photos.tgz  to unpack it.)

Once your images are prepared, and you have pip-installed tensorflow-hub and
a sufficiently recent version of tensorflow, you can run the training with a
command like this:

```bash
python retrain.py --image_dir ~/flower_photos
```

You can replace the image_dir argument with any folder containing subfolders of
images. The label for each image is taken from the name of the subfolder it's
in.

This produces a new model file that can be loaded and run by any TensorFlow
program, for example the tensorflow/examples/label_image sample code.

By default this script will use the highly accurate, but comparatively large and
```

```
slow Inception V3 model architecture. It's recommended that you start with this
to validate that you have gathered good training data, but if you want to deploy
on resource-limited platforms, you can try the `--tfhub_module` flag with a
Mobilenet model. For more information on Mobilenet, see
https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html

For example:

Run floating-point version of Mobilenet:

```bash
python retrain.py --image_dir ~/flower_photos \
    --tfhub_module https://tfhub.dev/google/imagenet/mobilenet_v1_100_224/feature_vector/1
```

Run Mobilenet, instrumented for quantization:

```bash
python retrain.py --image_dir ~/flower_photos/ \
    --tfhub_module https://tfhub.dev/google/imagenet/mobilenet_v1_100_224/quantops/feature_vector/1
```

These instrumented models can be converted to fully quantized mobile models via
TensorFlow Lite.

There are different Mobilenet models to choose from, with a variety of file
size and latency options.
  - The first number can be '100', '075', '050', or '025' to control the number
    of neurons (activations of hidden layers); the number of weights (and hence
    to some extent the file size and speed) shrinks with the square of that
    fraction.
  - The second number is the input image size. You can choose '224', '192',
```

```
    '160', or '128', with smaller sizes giving faster speeds.

To use with TensorBoard:

By default, this script will log summaries to /tmp/retrain_logs directory

Visualize the summaries with this command:

tensorboard --logdir /tmp/retrain_logs

To use with Tensorflow Serving, run this tool with --saved_model_dir set
to some increasingly numbered export location under the model base path, e.g
    .:

```bash
python retrain.py (... other args as before ...) \
    --saved_model_dir=/tmp/saved_models/$(date +%s)/
tensorflow_model_server --port=9000 --model_name=my_image_classifier \
    --model_base_path=/tmp/saved_models/
```
"""
pylint: enable=line-too-long

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import collections
from datetime import datetime
import hashlib
import os.path
import random
import re
import sys

import numpy as np
import tensorflow as tf
import tensorflow_hub as hub

```

```
135 FLAGS = None
136
137 MAX_NUM_IMAGES_PER_CLASS = 2 ** 27 - 1 # ~134M
138
139 # The location where variable checkpoints will be stored.
140 CHECKPOINT_NAME = '/tmp/_retrain_checkpoint'
141
142 # A module is understood as instrumented for quantization with TF-Lite
143 # if it contains any of these ops.
144 FAKE_QUANT_OPS = ('FakeQuantWithMinMaxVars',
145   'FakeQuantWithMinMaxVarsPerChannel')
146
147
148 def create_image_lists(image_dir, testing_percentage, validation_percentage)
    :
149   """Builds a list of training images from the file system.
150
151   Analyzes the sub folders in the image directory, splits them into stable
152   training, testing, and validation sets, and returns a data structure
153   describing the lists of images for each label and their paths.
154
155   Args:
156   image_dir: String path to a folder containing subfolders of images.
157   testing_percentage: Integer percentage of the images to reserve for
     tests.
158   validation_percentage: Integer percentage of images reserved for
     validation.
159
160   Returns:
161   An OrderedDict containing an entry for each label subfolder, with images
162   split into training, testing, and validation sets within each label.
163   The order of items defines the class indices.
164   """
165   if not tf.gfile.Exists(image_dir):
166   tf.logging.error("Image directory '" + image_dir + "' not found.")
167   return None
168   result = collections.OrderedDict()
169   sub_dirs = sorted(x[0] for x in tf.gfile.Walk(image_dir))
170   # The root directory comes first, so skip it.
171   is_root_dir = True
```

```
172  for sub_dir in sub_dirs:
173    if is_root_dir:
174      is_root_dir = False
175      continue
176    extensions = sorted(set(os.path.normcase(ext)  # Smash case on Windows.
177                            for ext in ['JPEG', 'JPG', 'jpeg', 'jpg']))
178    file_list = []
179    dir_name = os.path.basename(sub_dir)
180    if dir_name == image_dir:
181      continue
182    tf.logging.info("Looking for images in '" + dir_name + "'")
183    for extension in extensions:
184      file_glob = os.path.join(image_dir, dir_name, '*.' + extension)
185      file_list.extend(tf.gfile.Glob(file_glob))
186    if not file_list:
187      tf.logging.warning('No files found')
188      continue
189    if len(file_list) < 20:
190      tf.logging.warning(
191          'WARNING: Folder has less than 20 images, which may cause issues.'
    )
192    elif len(file_list) > MAX_NUM_IMAGES_PER_CLASS:
193      tf.logging.warning(
194          'WARNING: Folder {} has more than {} images. Some images will '
195          'never be selected.'.format(dir_name, MAX_NUM_IMAGES_PER_CLASS))
196    label_name = re.sub(r'[^a-z0-9]+', ' ', dir_name.lower())
197    training_images = []
198    testing_images = []
199    validation_images = []
200    for file_name in file_list:
201      base_name = os.path.basename(file_name)
202      # We want to ignore anything after '_nohash_' in the file name when
203      # deciding which set to put an image in, the data set creator has a
    way of
204      # grouping photos that are close variations of each other. For example
205      # this is used in the plant disease data set to group multiple
    pictures of
206      # the same leaf.
207      hash_name = re.sub(r'_nohash_.*$', '', file_name)
208      # This looks a bit magical, but we need to decide whether this file
```

```
    should
209       # go into the training, testing, or validation sets, and we want to
    keep
210       # existing files in the same set even if more files are subsequently
211       # added.
212       # To do that, we need a stable way of deciding based on just the file
    name
213       # itself, so we do a hash of that and then use that to generate a
214       # probability value that we use to assign it.
215       hash_name_hashed = hashlib.sha1(tf.compat.as_bytes(hash_name)).
    hexdigest()
216       percentage_hash = ((int(hash_name_hashed, 16) %
217                           (MAX_NUM_IMAGES_PER_CLASS + 1)) *
218                          (100.0 / MAX_NUM_IMAGES_PER_CLASS))
219       if percentage_hash < validation_percentage:
220         validation_images.append(base_name)
221       elif percentage_hash < (testing_percentage + validation_percentage):
222         testing_images.append(base_name)
223       else:
224         training_images.append(base_name)
225     result[label_name] = {
226         'dir': dir_name,
227         'training': training_images,
228         'testing': testing_images,
229         'validation': validation_images,
230     }
231   return result
232
233
234 def get_image_path(image_lists, label_name, index, image_dir, category):
235   """Returns a path to an image for a label at the given index.
236
237   Args:
238     image_lists: OrderedDict of training images for each label.
239     label_name: Label string we want to get an image for.
240     index: Int offset of the image we want. This will be moduloed by the
241     available number of images for the label, so it can be arbitrarily large
    .
242     image_dir: Root folder string of the subfolders containing the training
243     images.
```

```
244     category: Name string of set to pull images from - training, testing, or
245     validation.
246
247   Returns:
248     File system path string to an image that meets the requested parameters.
249
250   """
251   if label_name not in image_lists:
252     tf.logging.fatal('Label does not exist %s.', label_name)
253   label_lists = image_lists[label_name]
254   if category not in label_lists:
255     tf.logging.fatal('Category does not exist %s.', category)
256   category_list = label_lists[category]
257   if not category_list:
258     tf.logging.fatal('Label %s has no images in the category %s.',
259                      label_name, category)
260   mod_index = index % len(category_list)
261   base_name = category_list[mod_index]
262   sub_dir = label_lists['dir']
263   full_path = os.path.join(image_dir, sub_dir, base_name)
264   return full_path
265
266
267 def get_bottleneck_path(image_lists, label_name, index, bottleneck_dir,
268                         category, module_name):
269   """Returns a path to a bottleneck file for a label at the given index.
270
271   Args:
272     image_lists: OrderedDict of training images for each label.
273     label_name: Label string we want to get an image for.
274     index: Integer offset of the image we want. This will be moduloed by the
275     available number of images for the label, so it can be arbitrarily large
      .
276     bottleneck_dir: Folder string holding cached files of bottleneck values.
277     category: Name string of set to pull images from - training, testing, or
278     validation.
279     module_name: The name of the image module being used.
280
281   Returns:
282     File system path string to an image that meets the requested parameters.
```

```
  """
  module_name = (module_name.replace('://', '~')  # URL scheme.
                 .replace('/', '~')  # URL and Unix paths.
                 .replace(':', '~').replace('\\', '~'))  # Windows paths.
  return get_image_path(image_lists, label_name, index, bottleneck_dir,
                        category) + '_' + module_name + '.txt'


def create_module_graph(module_spec):
  """Creates a graph and loads Hub Module into it.

  Args:
    module_spec: the hub.ModuleSpec for the image module being used.

  Returns:
    graph: the tf.Graph that was created.
    bottleneck_tensor: the bottleneck values output by the module.
    resized_input_tensor: the input images, resized as expected by the
    module.
    wants_quantization: a boolean, whether the module has been instrumented
      with fake quantization ops.
  """
  height, width = hub.get_expected_image_size(module_spec)
  with tf.Graph().as_default() as graph:
    resized_input_tensor = tf.placeholder(tf.float32, [None, height, width,
    3])
    m = hub.Module(module_spec)
    bottleneck_tensor = m(resized_input_tensor)
    wants_quantization = any(node.op in FAKE_QUANT_OPS
                             for node in graph.as_graph_def().node)
  return graph, bottleneck_tensor, resized_input_tensor, wants_quantization


def run_bottleneck_on_image(sess, image_data, image_data_tensor,
                            decoded_image_tensor, resized_input_tensor,
                            bottleneck_tensor):
  """Runs inference on an image to extract the 'bottleneck' summary layer.

  Args:
    sess: Current active TensorFlow Session.
```

```
321     image_data: String of raw JPEG data.
322     image_data_tensor: Input data layer in the graph.
323     decoded_image_tensor: Output of initial image resizing and preprocessing
        .
324     resized_input_tensor: The input node of the recognition graph.
325     bottleneck_tensor: Layer before the final softmax.
326
327   Returns:
328     Numpy array of bottleneck values.
329   """
330   # First decode the JPEG image, resize it, and rescale the pixel values.
331   resized_input_values = sess.run(decoded_image_tensor,
332                                   {image_data_tensor: image_data})
333   # Then run it through the recognition network.
334   bottleneck_values = sess.run(bottleneck_tensor,
335                                {resized_input_tensor: resized_input_values})
336   bottleneck_values = np.squeeze(bottleneck_values)
337   return bottleneck_values
338
339
340 def ensure_dir_exists(dir_name):
341   """Makes sure the folder exists on disk.
342
343   Args:
344     dir_name: Path string to the folder we want to create.
345   """
346   if not os.path.exists(dir_name):
347     os.makedirs(dir_name)
348
349
350 def create_bottleneck_file(bottleneck_path, image_lists, label_name, index,
351                            image_dir, category, sess, jpeg_data_tensor,
352                            decoded_image_tensor, resized_input_tensor,
353                            bottleneck_tensor):
354   """Create a single bottleneck file."""
355   tf.logging.info('Creating bottleneck at ' + bottleneck_path)
356   image_path = get_image_path(image_lists, label_name, index,
357                               image_dir, category)
358   if not tf.gfile.Exists(image_path):
359     tf.logging.fatal('File does not exist %s', image_path)
```

```
360   image_data = tf.gfile.FastGFile(image_path, 'rb').read()
361   try:
362     bottleneck_values = run_bottleneck_on_image(
363         sess, image_data, jpeg_data_tensor, decoded_image_tensor,
364         resized_input_tensor, bottleneck_tensor)
365   except Exception as e:
366     raise RuntimeError('Error during processing file %s (%s)' % (image_path,
367                                                                  str(e)))
368   bottleneck_string = ','.join(str(x) for x in bottleneck_values)
369   with open(bottleneck_path, 'w') as bottleneck_file:
370     bottleneck_file.write(bottleneck_string)
371
372
373 def get_or_create_bottleneck(sess, image_lists, label_name, index, image_dir
      ,
374                              category, bottleneck_dir, jpeg_data_tensor,
375                              decoded_image_tensor, resized_input_tensor,
376                              bottleneck_tensor, module_name):
377   """Retrieves or calculates bottleneck values for an image.
378
379   If a cached version of the bottleneck data exists on-disk, return that,
380   otherwise calculate the data and save it to disk for future use.
381
382   Args:
383     sess: The current active TensorFlow Session.
384     image_lists: OrderedDict of training images for each label.
385     label_name: Label string we want to get an image for.
386     index: Integer offset of the image we want. This will be modulo-ed by
       the
387     available number of images for the label, so it can be arbitrarily large
       .
388     image_dir: Root folder string of the subfolders containing the training
389     images.
390     category: Name string of which set to pull images from - training,
       testing,
391     or validation.
392     bottleneck_dir: Folder string holding cached files of bottleneck values.
393     jpeg_data_tensor: The tensor to feed loaded jpeg data into.
394     decoded_image_tensor: The output of decoding and resizing the image.
395     resized_input_tensor: The input node of the recognition graph.
```

```
396     bottleneck_tensor: The output tensor for the bottleneck values.
397     module_name: The name of the image module being used.
398 
399   Returns:
400     Numpy array of values produced by the bottleneck layer for the image.
401   """
402   label_lists = image_lists[label_name]
403   sub_dir = label_lists['dir']
404   sub_dir_path = os.path.join(bottleneck_dir, sub_dir)
405   ensure_dir_exists(sub_dir_path)
406   bottleneck_path = get_bottleneck_path(image_lists, label_name, index,
407                                         bottleneck_dir, category,
    module_name)
408   if not os.path.exists(bottleneck_path):
409     create_bottleneck_file(bottleneck_path, image_lists, label_name, index,
410                            image_dir, category, sess, jpeg_data_tensor,
411                            decoded_image_tensor, resized_input_tensor,
412                            bottleneck_tensor)
413   with open(bottleneck_path, 'r') as bottleneck_file:
414     bottleneck_string = bottleneck_file.read()
415   did_hit_error = False
416   try:
417     bottleneck_values = [float(x) for x in bottleneck_string.split(',')]
418   except ValueError:
419     tf.logging.warning('Invalid float found, recreating bottleneck')
420     did_hit_error = True
421   if did_hit_error:
422     create_bottleneck_file(bottleneck_path, image_lists, label_name, index,
423                            image_dir, category, sess, jpeg_data_tensor,
424                            decoded_image_tensor, resized_input_tensor,
425                            bottleneck_tensor)
426     with open(bottleneck_path, 'r') as bottleneck_file:
427       bottleneck_string = bottleneck_file.read()
428     # Allow exceptions to propagate here, since they shouldn't happen after
    a
429     # fresh creation
430     bottleneck_values = [float(x) for x in bottleneck_string.split(',')]
431   return bottleneck_values
432 
433 
```

```
434 def cache_bottlenecks(sess, image_lists, image_dir, bottleneck_dir,
435                       jpeg_data_tensor, decoded_image_tensor,
436                       resized_input_tensor, bottleneck_tensor, module_name):
437   """Ensures all the training, testing, and validation bottlenecks are
      cached.
438
439   Because we're likely to read the same image multiple times (if there are
      no
440   distortions applied during training) it can speed things up a lot if we
441   calculate the bottleneck layer values once for each image during
442   preprocessing, and then just read those cached values repeatedly during
443   training. Here we go through all the images we've found, calculate those
444   values, and save them off.
445
446   Args:
447     sess: The current active TensorFlow Session.
448     image_lists: OrderedDict of training images for each label.
449     image_dir: Root folder string of the subfolders containing the training
450     images.
451     bottleneck_dir: Folder string holding cached files of bottleneck values.
452     jpeg_data_tensor: Input tensor for jpeg data from file.
453     decoded_image_tensor: The output of decoding and resizing the image.
454     resized_input_tensor: The input node of the recognition graph.
455     bottleneck_tensor: The penultimate output layer of the graph.
456     module_name: The name of the image module being used.
457
458   Returns:
459     Nothing.
460   """
461   how_many_bottlenecks = 0
462   ensure_dir_exists(bottleneck_dir)
463   for label_name, label_lists in image_lists.items():
464     for category in ['training', 'testing', 'validation']:
465       category_list = label_lists[category]
466       for index, unused_base_name in enumerate(category_list):
467         get_or_create_bottleneck(
468             sess, image_lists, label_name, index, image_dir, category,
469             bottleneck_dir, jpeg_data_tensor, decoded_image_tensor,
470             resized_input_tensor, bottleneck_tensor, module_name)
471
```

```
472         how_many_bottlenecks += 1
473         if how_many_bottlenecks % 100 == 0:
474           tf.logging.info(
475               str(how_many_bottlenecks) + ' bottleneck files created.')
476
477
478 def get_random_cached_bottlenecks(sess, image_lists, how_many, category,
479                                   bottleneck_dir, image_dir,
    jpeg_data_tensor,
480                                   decoded_image_tensor, resized_input_tensor
    ,
481                                   bottleneck_tensor, module_name):
482   """Retrieves bottleneck values for cached images.
483
484   If no distortions are being applied, this function can retrieve the cached
485   bottleneck values directly from disk for images. It picks a random set of
486   images from the specified category.
487
488   Args:
489     sess: Current TensorFlow Session.
490     image_lists: OrderedDict of training images for each label.
491     how_many: If positive, a random sample of this size will be chosen.
492     If negative, all bottlenecks will be retrieved.
493     category: Name string of which set to pull from - training, testing, or
494     validation.
495     bottleneck_dir: Folder string holding cached files of bottleneck values.
496     image_dir: Root folder string of the subfolders containing the training
497     images.
498     jpeg_data_tensor: The layer to feed jpeg image data into.
499     decoded_image_tensor: The output of decoding and resizing the image.
500     resized_input_tensor: The input node of the recognition graph.
501     bottleneck_tensor: The bottleneck output layer of the CNN graph.
502     module_name: The name of the image module being used.
503
504   Returns:
505     List of bottleneck arrays, their corresponding ground truths, and the
506     relevant filenames.
507   """
508   class_count = len(image_lists.keys())
509   bottlenecks = []
```

```
  ground_truths = []
  filenames = []
  if how_many >= 0:
    # Retrieve a random sample of bottlenecks.
    for unused_i in range(how_many):
      label_index = random.randrange(class_count)
      label_name = list(image_lists.keys())[label_index]
      image_index = random.randrange(MAX_NUM_IMAGES_PER_CLASS + 1)
      image_name = get_image_path(image_lists, label_name, image_index,
                                  image_dir, category)
      bottleneck = get_or_create_bottleneck(
          sess, image_lists, label_name, image_index, image_dir, category,
          bottleneck_dir, jpeg_data_tensor, decoded_image_tensor,
          resized_input_tensor, bottleneck_tensor, module_name)
      bottlenecks.append(bottleneck)
      ground_truths.append(label_index)
      filenames.append(image_name)
  else:
    # Retrieve all bottlenecks.
    for label_index, label_name in enumerate(image_lists.keys()):
      for image_index, image_name in enumerate(
          image_lists[label_name][category]):
        image_name = get_image_path(image_lists, label_name, image_index,
                                    image_dir, category)
        bottleneck = get_or_create_bottleneck(
            sess, image_lists, label_name, image_index, image_dir, category,
            bottleneck_dir, jpeg_data_tensor, decoded_image_tensor,
            resized_input_tensor, bottleneck_tensor, module_name)
        bottlenecks.append(bottleneck)
        ground_truths.append(label_index)
        filenames.append(image_name)
  return bottlenecks, ground_truths, filenames


def get_random_distorted_bottlenecks(
    sess, image_lists, how_many, category, image_dir, input_jpeg_tensor,
    distorted_image, resized_input_tensor, bottleneck_tensor):
  """Retrieves bottleneck values for training images, after distortions.

  If we're training with distortions like crops, scales, or flips, we have
```

```
  to
recalculate the full model for every image, and so we can't use cached
bottleneck values. Instead we find random images for the requested
  category,
run them through the distortion graph, and then the full graph to get the
bottleneck results for each.

Args:
  sess: Current TensorFlow Session.
  image_lists: OrderedDict of training images for each label.
  how_many: The integer number of bottleneck values to return.
  category: Name string of which set of images to fetch - training,
  testing,
  or validation.
  image_dir: Root folder string of the subfolders containing the training
  images.
  input_jpeg_tensor: The input layer we feed the image data to.
  distorted_image: The output node of the distortion graph.
  resized_input_tensor: The input node of the recognition graph.
  bottleneck_tensor: The bottleneck output layer of the CNN graph.

Returns:
  List of bottleneck arrays and their corresponding ground truths.
"""
class_count = len(image_lists.keys())
bottlenecks = []
ground_truths = []
for unused_i in range(how_many):
  label_index = random.randrange(class_count)
  label_name = list(image_lists.keys())[label_index]
  image_index = random.randrange(MAX_NUM_IMAGES_PER_CLASS + 1)
  image_path = get_image_path(image_lists, label_name, image_index,
  image_dir,
                              category)
  if not tf.gfile.Exists(image_path):
    tf.logging.fatal('File does not exist %s', image_path)
  jpeg_data = tf.gfile.FastGFile(image_path, 'rb').read()
  # Note that we materialize the distorted_image_data as a numpy array
  before
  # sending running inference on the image. This involves 2 memory copies
```

```
    and
585     # might be optimized in other implementations.
586     distorted_image_data = sess.run(distorted_image,
587                                     {input_jpeg_tensor: jpeg_data})
588     bottleneck_values = sess.run(bottleneck_tensor,
589                                  {resized_input_tensor: distorted_image_data
    })
590     bottleneck_values = np.squeeze(bottleneck_values)
591     bottlenecks.append(bottleneck_values)
592     ground_truths.append(label_index)
593   return bottlenecks, ground_truths
594
595
596 def should_distort_images(flip_left_right, random_crop, random_scale,
597                           random_brightness):
598   """Whether any distortions are enabled, from the input flags.
599
600   Args:
601     flip_left_right: Boolean whether to randomly mirror images horizontally.
602     random_crop: Integer percentage setting the total margin used around the
603     crop box.
604     random_scale: Integer percentage of how much to vary the scale by.
605     random_brightness: Integer range to randomly multiply the pixel values
    by.
606
607   Returns:
608     Boolean value indicating whether any distortions should be applied.
609   """
610   return (flip_left_right or (random_crop != 0) or (random_scale != 0) or
611           (random_brightness != 0))
612
613
614 def add_input_distortions(flip_left_right, random_crop, random_scale,
615                           random_brightness, module_spec):
616   """Creates the operations to apply the specified distortions.
617
618   During training it can help to improve the results if we run the images
619   through simple distortions like crops, scales, and flips. These reflect
    the
620   kind of variations we expect in the real world, and so can help train the
```

```
621 model to cope with natural data more effectively. Here we take the
     supplied
622 parameters and construct a network of operations to apply them to an image
    .
623
624 Cropping
625 ~~~~~~~~
626
627 Cropping is done by placing a bounding box at a random position in the
     full
628 image. The cropping parameter controls the size of that box relative to
     the
629 input image. If it's zero, then the box is the same size as the input and
     no
630 cropping is performed. If the value is 50%, then the crop box will be half
     the
631 width and height of the input. In a diagram it looks like this:
632
633 <        width         >
634 +----------------------+
635 |                      |
636 |   width - crop%      |
637 |    <      >          |
638 |    +------+          |
639 |    |      |          |
640 |    |      |          |
641 |    |      |          |
642 |    +------+          |
643 |                      |
644 |                      |
645 +----------------------+
646
647 Scaling
648 ~~~~~~~
649
650 Scaling is a lot like cropping, except that the bounding box is always
651 centered and its size varies randomly within the given range. For example
     if
652 the scale percentage is zero, then the bounding box is the same size as
     the
```

```
653   input and no scaling is applied. If it's 50%, then the bounding box will
       be in
654   a random range between half the width and height and full size.
655
656   Args:
657     flip_left_right: Boolean whether to randomly mirror images horizontally.
658     random_crop: Integer percentage setting the total margin used around the
659     crop box.
660     random_scale: Integer percentage of how much to vary the scale by.
661     random_brightness: Integer range to randomly multiply the pixel values
       by.
662     graph.
663     module_spec: The hub.ModuleSpec for the image module being used.
664
665   Returns:
666     The jpeg input layer and the distorted result tensor.
667   """
668   input_height, input_width = hub.get_expected_image_size(module_spec)
669   input_depth = hub.get_num_image_channels(module_spec)
670   jpeg_data = tf.placeholder(tf.string, name='DistortJPGInput')
671   decoded_image = tf.image.decode_jpeg(jpeg_data, channels=input_depth)
672   # Convert from full range of uint8 to range [0,1] of float32.
673   decoded_image_as_float = tf.image.convert_image_dtype(decoded_image,
674                                                         tf.float32)
675   decoded_image_4d = tf.expand_dims(decoded_image_as_float, 0)
676   margin_scale = 1.0 + (random_crop / 100.0)
677   resize_scale = 1.0 + (random_scale / 100.0)
678   margin_scale_value = tf.constant(margin_scale)
679   resize_scale_value = tf.random_uniform(shape=[],
680                                          minval=1.0,
681                                          maxval=resize_scale)
682   scale_value = tf.multiply(margin_scale_value, resize_scale_value)
683   precrop_width = tf.multiply(scale_value, input_width)
684   precrop_height = tf.multiply(scale_value, input_height)
685   precrop_shape = tf.stack([precrop_height, precrop_width])
686   precrop_shape_as_int = tf.cast(precrop_shape, dtype=tf.int32)
687   precropped_image = tf.image.resize_bilinear(decoded_image_4d,
688                                               precrop_shape_as_int)
689   precropped_image_3d = tf.squeeze(precropped_image, axis=[0])
690   cropped_image = tf.random_crop(precropped_image_3d,
```

```
691                                     [input_height, input_width, input_depth])
692   if flip_left_right:
693     flipped_image = tf.image.random_flip_left_right(cropped_image)
694   else:
695     flipped_image = cropped_image
696   brightness_min = 1.0 - (random_brightness / 100.0)
697   brightness_max = 1.0 + (random_brightness / 100.0)
698   brightness_value = tf.random_uniform(shape=[],
699                                        minval=brightness_min,
700                                        maxval=brightness_max)
701   brightened_image = tf.multiply(flipped_image, brightness_value)
702   distort_result = tf.expand_dims(brightened_image, 0, name='DistortResult')
703   return jpeg_data, distort_result
704
705
706 def variable_summaries(var):
707   """Attach a lot of summaries to a Tensor (for TensorBoard visualization).
      """
708   with tf.name_scope('summaries'):
709     mean = tf.reduce_mean(var)
710     tf.summary.scalar('mean', mean)
711     with tf.name_scope('stddev'):
712       stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
713     tf.summary.scalar('stddev', stddev)
714     tf.summary.scalar('max', tf.reduce_max(var))
715     tf.summary.scalar('min', tf.reduce_min(var))
716     tf.summary.histogram('histogram', var)
717
718
719 def add_final_retrain_ops(class_count, final_tensor_name, bottleneck_tensor,
720                           quantize_layer, is_training):
721   """Adds a new softmax and fully-connected layer for training and eval.
722
723   We need to retrain the top layer to identify our new classes, so this
      function
724   adds the right operations to the graph, along with some variables to hold
      the
725   weights, and then sets up all the gradients for the backward pass.
726
727   The set up for the softmax and fully-connected layers is based on:
```

```
728   https://www.tensorflow.org/tutorials/mnist/beginners/index.html
729
730   Args:
731     class_count: Integer of how many categories of things we're trying to
732         recognize.
733     final_tensor_name: Name string for the new final node that produces
      results.
734     bottleneck_tensor: The output of the main CNN graph.
735     quantize_layer: Boolean, specifying whether the newly added layer should
      be
736         instrumented for quantization with TF-Lite.
737     is_training: Boolean, specifying whether the newly add layer is for
      training
738         or eval.
739
740   Returns:
741     The tensors for the training and cross entropy results, and tensors for
      the
742     bottleneck input and ground truth input.
743   """
744   batch_size, bottleneck_tensor_size = bottleneck_tensor.get_shape().as_list
      ()
745   assert batch_size is None, 'We want to work with arbitrary batch size.'
746   with tf.name_scope('input'):
747     bottleneck_input = tf.placeholder_with_default(
748         bottleneck_tensor,
749         shape=[batch_size, bottleneck_tensor_size],
750         name='BottleneckInputPlaceholder')
751
752     ground_truth_input = tf.placeholder(
753         tf.int64, [batch_size], name='GroundTruthInput')
754
755   # Organizing the following ops so they are easier to see in TensorBoard.
756   layer_name = 'final_retrain_ops'
757   with tf.name_scope(layer_name):
758     with tf.name_scope('weights'):
759       initial_value = tf.truncated_normal(
760           [bottleneck_tensor_size, class_count], stddev=0.001)
761       layer_weights = tf.Variable(initial_value, name='final_weights')
762       variable_summaries(layer_weights)
```

```
763
764     with tf.name_scope('biases'):
765       layer_biases = tf.Variable(tf.zeros([class_count]), name='final_biases
        ')
766       variable_summaries(layer_biases)
767
768     with tf.name_scope('Wx_plus_b'):
769       logits = tf.matmul(bottleneck_input, layer_weights) + layer_biases
770       tf.summary.histogram('pre_activations', logits)
771
772   final_tensor = tf.nn.softmax(logits, name=final_tensor_name)
773
774   # The tf.contrib.quantize functions rewrite the graph in place for
775   # quantization. The imported model graph has already been rewritten, so
        upon
776   # calling these rewrites, only the newly added final layer will be
777   # transformed.
778   if quantize_layer:
779     if is_training:
780       tf.contrib.quantize.create_training_graph()
781     else:
782       tf.contrib.quantize.create_eval_graph()
783
784   tf.summary.histogram('activations', final_tensor)
785
786   # If this is an eval graph, we don't need to add loss ops or an optimizer.
787   if not is_training:
788     return None, None, bottleneck_input, ground_truth_input, final_tensor
789
790   with tf.name_scope('cross_entropy'):
791     cross_entropy_mean = tf.losses.sparse_softmax_cross_entropy(
792         labels=ground_truth_input, logits=logits)
793
794   tf.summary.scalar('cross_entropy', cross_entropy_mean)
795
796   with tf.name_scope('train'):
797     optimizer = tf.train.GradientDescentOptimizer(FLAGS.learning_rate)
798     train_step = optimizer.minimize(cross_entropy_mean)
799
800   return (train_step, cross_entropy_mean, bottleneck_input,
```

```
    ground_truth_input,
801           final_tensor)
802
803
804 def add_evaluation_step(result_tensor, ground_truth_tensor):
805   """Inserts the operations we need to evaluate the accuracy of our results.
806
807   Args:
808     result_tensor: The new final node that produces results.
809     ground_truth_tensor: The node we feed ground truth data
810     into.
811
812   Returns:
813     Tuple of (evaluation step, prediction).
814   """
815   with tf.name_scope('accuracy'):
816     with tf.name_scope('correct_prediction'):
817       prediction = tf.argmax(result_tensor, 1)
818       correct_prediction = tf.equal(prediction, ground_truth_tensor)
819     with tf.name_scope('accuracy'):
820       evaluation_step = tf.reduce_mean(tf.cast(correct_prediction, tf.
    float32))
821   tf.summary.scalar('accuracy', evaluation_step)
822   return evaluation_step, prediction
823
824
825 def run_final_eval(train_session, module_spec, class_count, image_lists,
826                    jpeg_data_tensor, decoded_image_tensor,
827                    resized_image_tensor, bottleneck_tensor):
828   """Runs a final evaluation on an eval graph using the test data set.
829
830   Args:
831     train_session: Session for the train graph with the tensors below.
832     module_spec: The hub.ModuleSpec for the image module being used.
833     class_count: Number of classes
834     image_lists: OrderedDict of training images for each label.
835     jpeg_data_tensor: The layer to feed jpeg image data into.
836     decoded_image_tensor: The output of decoding and resizing the image.
837     resized_image_tensor: The input node of the recognition graph.
838     bottleneck_tensor: The bottleneck output layer of the CNN graph.
```

```
  """
  test_bottlenecks, test_ground_truth, test_filenames = (
      get_random_cached_bottlenecks(train_session, image_lists,
                                    FLAGS.test_batch_size,
                                    'testing', FLAGS.bottleneck_dir,
                                    FLAGS.image_dir, jpeg_data_tensor,
                                    decoded_image_tensor,
   resized_image_tensor,
                                    bottleneck_tensor, FLAGS.tfhub_module))

  (eval_session, _, bottleneck_input, ground_truth_input, evaluation_step,
   prediction) = build_eval_session(module_spec, class_count)
  test_accuracy, predictions = eval_session.run(
      [evaluation_step, prediction],
      feed_dict={
          bottleneck_input: test_bottlenecks,
          ground_truth_input: test_ground_truth
      })
  tf.logging.info('Final test accuracy = %.1f%% (N=%d)' %
                  (test_accuracy * 100, len(test_bottlenecks)))

  if FLAGS.print_misclassified_test_images:
    tf.logging.info('=== MISCLASSIFIED TEST IMAGES ===')
    for i, test_filename in enumerate(test_filenames):
      if predictions[i] != test_ground_truth[i]:
        tf.logging.info('%70s  %s' % (test_filename,
                                      list(image_lists.keys())[predictions[i
   ]]))


def build_eval_session(module_spec, class_count):
  """Builds an restored eval session without train operations for exporting.

  Args:
    module_spec: The hub.ModuleSpec for the image module being used.
    class_count: Number of classes

  Returns:
    Eval session containing the restored eval graph.
    The bottleneck input, ground truth, eval step, and prediction tensors.
```

```
  """
  # If quantized, we need to create the correct eval graph for exporting.
  eval_graph, bottleneck_tensor, resized_input_tensor, wants_quantization =
    (
      create_module_graph(module_spec))

  eval_sess = tf.Session(graph=eval_graph)
  with eval_graph.as_default():
    # Add the new layer for exporting.
    (_, _, bottleneck_input,
     ground_truth_input, final_tensor) = add_final_retrain_ops(
         class_count, FLAGS.final_tensor_name, bottleneck_tensor,
         wants_quantization, is_training=False)

    # Now we need to restore the values from the training graph to the eval
    # graph.
    tf.train.Saver().restore(eval_sess, CHECKPOINT_NAME)

    evaluation_step, prediction = add_evaluation_step(final_tensor,
                                                      ground_truth_input)

  return (eval_sess, resized_input_tensor, bottleneck_input,
    ground_truth_input,
          evaluation_step, prediction)


def save_graph_to_file(graph_file_name, module_spec, class_count):
  """Saves an graph to file, creating a valid quantized one if necessary."""
  sess, _, _, _, _, _ = build_eval_session(module_spec, class_count)
  graph = sess.graph

  output_graph_def = tf.graph_util.convert_variables_to_constants(
      sess, graph.as_graph_def(), [FLAGS.final_tensor_name])

  with tf.gfile.FastGFile(graph_file_name, 'wb') as f:
    f.write(output_graph_def.SerializeToString())


def prepare_file_system():
  # Set up the directory we'll write summaries to for TensorBoard
```

```
  if tf.gfile.Exists(FLAGS.summaries_dir):
    tf.gfile.DeleteRecursively(FLAGS.summaries_dir)
  tf.gfile.MakeDirs(FLAGS.summaries_dir)
  if FLAGS.intermediate_store_frequency > 0:
    ensure_dir_exists(FLAGS.intermediate_output_graphs_dir)
  return


def add_jpeg_decoding(module_spec):
  """Adds operations that perform JPEG decoding and resizing to the graph..

  Args:
    module_spec: The hub.ModuleSpec for the image module being used.

  Returns:
    Tensors for the node to feed JPEG data into, and the output of the
      preprocessing steps.
  """
  input_height, input_width = hub.get_expected_image_size(module_spec)
  input_depth = hub.get_num_image_channels(module_spec)
  jpeg_data = tf.placeholder(tf.string, name='DecodeJPGInput')
  decoded_image = tf.image.decode_jpeg(jpeg_data, channels=input_depth)
  # Convert from full range of uint8 to range [0,1] of float32.
  decoded_image_as_float = tf.image.convert_image_dtype(decoded_image,
                                                        tf.float32)
  decoded_image_4d = tf.expand_dims(decoded_image_as_float, 0)
  resize_shape = tf.stack([input_height, input_width])
  resize_shape_as_int = tf.cast(resize_shape, dtype=tf.int32)
  resized_image = tf.image.resize_bilinear(decoded_image_4d,
                                           resize_shape_as_int)
  return jpeg_data, resized_image


def export_model(module_spec, class_count, saved_model_dir):
  """Exports model for serving.

  Args:
    module_spec: The hub.ModuleSpec for the image module being used.
    class_count: The number of classes.
    saved_model_dir: Directory in which to save exported model and variables
```

```
    .
  """
  # The SavedModel should hold the eval graph.
  sess, in_image, _, _, _, _ = build_eval_session(module_spec, class_count)
  with sess.graph.as_default() as graph:
    tf.saved_model.simple_save(
        sess,
        saved_model_dir,
        inputs={'image': in_image},
        outputs={'prediction': graph.get_tensor_by_name('final_result:0')},
        legacy_init_op=tf.group(tf.tables_initializer(), name='
    legacy_init_op')
    )


def main(_):
  # Needed to make sure the logging output is visible.
  # See https://github.com/tensorflow/tensorflow/issues/3047
  tf.logging.set_verbosity(tf.logging.INFO)

  if not FLAGS.image_dir:
    tf.logging.error('Must set flag --image_dir.')
    return -1

  # Prepare necessary directories that can be used during training
  prepare_file_system()

  # Look at the folder structure, and create lists of all the images.
  image_lists = create_image_lists(FLAGS.image_dir, FLAGS.testing_percentage
    ,
                                   FLAGS.validation_percentage)
  class_count = len(image_lists.keys())
  if class_count == 0:
    tf.logging.error('No valid folders of images found at ' + FLAGS.
    image_dir)
    return -1
  if class_count == 1:
    tf.logging.error('Only one valid folder of images found at ' +
                     FLAGS.image_dir +
                     ' - multiple classes are needed for classification.')
```

```
    return -1

# See if the command-line flags mean we're applying any distortions.
do_distort_images = should_distort_images(
    FLAGS.flip_left_right, FLAGS.random_crop, FLAGS.random_scale,
    FLAGS.random_brightness)

# Set up the pre-trained graph.
module_spec = hub.load_module_spec(FLAGS.tfhub_module)
graph, bottleneck_tensor, resized_image_tensor, wants_quantization = (
    create_module_graph(module_spec))

# Add the new layer that we'll be training.
with graph.as_default():
  (train_step, cross_entropy, bottleneck_input,
   ground_truth_input, final_tensor) = add_final_retrain_ops(
       class_count, FLAGS.final_tensor_name, bottleneck_tensor,
       wants_quantization, is_training=True)

with tf.Session(graph=graph) as sess:
  # Initialize all weights: for the module to their pretrained values,
  # and for the newly added retraining layer to random initial values.
  init = tf.global_variables_initializer()
  sess.run(init)

  # Set up the image decoding sub-graph.
  jpeg_data_tensor, decoded_image_tensor = add_jpeg_decoding(module_spec)

  if do_distort_images:
    # We will be applying distortions, so set up the operations we'll need
  .
    (distorted_jpeg_data_tensor,
     distorted_image_tensor) = add_input_distortions(
         FLAGS.flip_left_right, FLAGS.random_crop, FLAGS.random_scale,
         FLAGS.random_brightness, module_spec)
  else:
    # We'll make sure we've calculated the 'bottleneck' image summaries
  and
    # cached them on disk.
    cache_bottlenecks(sess, image_lists, FLAGS.image_dir,
```

```
                          FLAGS.bottleneck_dir, jpeg_data_tensor,
                          decoded_image_tensor, resized_image_tensor,
                          bottleneck_tensor, FLAGS.tfhub_module)

  # Create the operations we need to evaluate the accuracy of our new
layer.
  evaluation_step, _ = add_evaluation_step(final_tensor,
ground_truth_input)

  # Merge all the summaries and write them out to the summaries_dir
  merged = tf.summary.merge_all()
  train_writer = tf.summary.FileWriter(FLAGS.summaries_dir + '/train',
                                       sess.graph)

  validation_writer = tf.summary.FileWriter(
      FLAGS.summaries_dir + '/validation')

  # Create a train saver that is used to restore values into an eval graph
  # when exporting models.
  train_saver = tf.train.Saver()

  # Run the training for as many cycles as requested on the command line.
  for i in range(FLAGS.how_many_training_steps):
    # Get a batch of input bottleneck values, either calculated fresh
every
    # time with distortions applied, or from the cache stored on disk.
    if do_distort_images:
      (train_bottlenecks,
       train_ground_truth) = get_random_distorted_bottlenecks(
          sess, image_lists, FLAGS.train_batch_size, 'training',
          FLAGS.image_dir, distorted_jpeg_data_tensor,
          distorted_image_tensor, resized_image_tensor, bottleneck_tensor
)
    else:
      (train_bottlenecks,
       train_ground_truth, _) = get_random_cached_bottlenecks(
          sess, image_lists, FLAGS.train_batch_size, 'training',
          FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor,
          decoded_image_tensor, resized_image_tensor, bottleneck_tensor,
          FLAGS.tfhub_module)
```

```
1065      # Feed the bottlenecks and ground truth into the graph, and run a
      training
1066      # step. Capture training summaries for TensorBoard with the `merged`
      op.
1067      train_summary, _ = sess.run(
1068          [merged, train_step],
1069          feed_dict={bottleneck_input: train_bottlenecks,
1070                     ground_truth_input: train_ground_truth})
1071      train_writer.add_summary(train_summary, i)
1072
1073      # Every so often, print out how well the graph is training.
1074      is_last_step = (i + 1 == FLAGS.how_many_training_steps)
1075      if (i % FLAGS.eval_step_interval) == 0 or is_last_step:
1076        train_accuracy, cross_entropy_value = sess.run(
1077            [evaluation_step, cross_entropy],
1078            feed_dict={bottleneck_input: train_bottlenecks,
1079                       ground_truth_input: train_ground_truth})
1080        tf.logging.info('%s: Step %d: Train accuracy = %.1f%%' %
1081                        (datetime.now(), i, train_accuracy * 100))
1082        tf.logging.info('%s: Step %d: Cross entropy = %f' %
1083                        (datetime.now(), i, cross_entropy_value))
1084        # TODO: Make this use an eval graph, to avoid quantization
1085        # moving averages being updated by the validation set, though in
1086        # practice this makes a negligable difference.
1087        validation_bottlenecks, validation_ground_truth, _ = (
1088            get_random_cached_bottlenecks(
1089                sess, image_lists, FLAGS.validation_batch_size, 'validation'
      ,
1090                FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor,
1091                decoded_image_tensor, resized_image_tensor,
      bottleneck_tensor,
1092                FLAGS.tfhub_module))
1093        # Run a validation step and capture training summaries for
      TensorBoard
1094        # with the `merged` op.
1095        validation_summary, validation_accuracy = sess.run(
1096            [merged, evaluation_step],
1097            feed_dict={bottleneck_input: validation_bottlenecks,
1098                       ground_truth_input: validation_ground_truth})
1099        validation_writer.add_summary(validation_summary, i)
```

```
1100        tf.logging.info('%s: Step %d: Validation accuracy = %.1f%% (N=%d)' %
1101                        (datetime.now(), i, validation_accuracy * 100,
1102                         len(validation_bottlenecks)))
1103
1104      # Store intermediate results
1105      intermediate_frequency = FLAGS.intermediate_store_frequency
1106
1107      if (intermediate_frequency > 0 and (i % intermediate_frequency == 0)
1108          and i > 0):
1109        # If we want to do an intermediate save, save a checkpoint of the
    train
1110        # graph, to restore into the eval graph.
1111        train_saver.save(sess, CHECKPOINT_NAME)
1112        intermediate_file_name = (FLAGS.intermediate_output_graphs_dir +
1113                                  'intermediate_' + str(i) + '.pb')
1114        tf.logging.info('Save intermediate result to : ' +
1115                        intermediate_file_name)
1116        save_graph_to_file(intermediate_file_name, module_spec,
1117                           class_count)
1118
1119    # After training is complete, force one last save of the train
    checkpoint.
1120    train_saver.save(sess, CHECKPOINT_NAME)
1121
1122    # We've completed all our training, so run a final test evaluation on
1123    # some new images we haven't used before.
1124    run_final_eval(sess, module_spec, class_count, image_lists,
1125                   jpeg_data_tensor, decoded_image_tensor,
    resized_image_tensor,
1126                   bottleneck_tensor)
1127
1128    # Write out the trained graph and labels with the weights stored as
1129    # constants.
1130    tf.logging.info('Save final result to : ' + FLAGS.output_graph)
1131    if wants_quantization:
1132      tf.logging.info('The model is instrumented for quantization with TF-
    Lite')
1133    save_graph_to_file(FLAGS.output_graph, module_spec, class_count)
1134    with tf.gfile.FastGFile(FLAGS.output_labels, 'w') as f:
1135      f.write('\n'.join(image_lists.keys()) + '\n')
```

```
1136
1137     if FLAGS.saved_model_dir:
1138       export_model(module_spec, class_count, FLAGS.saved_model_dir)
1139
1140
1141 if __name__ == '__main__':
1142   parser = argparse.ArgumentParser()
1143   parser.add_argument(
1144       '--image_dir',
1145       type=str,
1146       default='',
1147       help='Path to folders of labeled images.'
1148   )
1149   parser.add_argument(
1150       '--output_graph',
1151       type=str,
1152       default='/tmp/output_graph.pb',
1153       help='Where to save the trained graph.'
1154   )
1155   parser.add_argument(
1156       '--intermediate_output_graphs_dir',
1157       type=str,
1158       default='/tmp/intermediate_graph/',
1159       help='Where to save the intermediate graphs.'
1160   )
1161   parser.add_argument(
1162       '--intermediate_store_frequency',
1163       type=int,
1164       default=0,
1165       help="""\
1166          How many steps to store intermediate graph. If "0" then will not
1167          store.\
1168       """
1169   )
1170   parser.add_argument(
1171       '--output_labels',
1172       type=str,
1173       default='/tmp/output_labels.txt',
1174       help='Where to save the trained graph\'s labels.'
1175   )
```

```
1176  parser.add_argument(
1177      '--summaries_dir',
1178      type=str,
1179      default='/tmp/retrain_logs',
1180      help='Where to save summary logs for TensorBoard.'
1181  )
1182  parser.add_argument(
1183      '--how_many_training_steps',
1184      type=int,
1185      default=4000,
1186      help='How many training steps to run before ending.'
1187  )
1188  parser.add_argument(
1189      '--learning_rate',
1190      type=float,
1191      default=0.01,
1192      help='How large a learning rate to use when training.'
1193  )
1194  parser.add_argument(
1195      '--testing_percentage',
1196      type=int,
1197      default=10,
1198      help='What percentage of images to use as a test set.'
1199  )
1200  parser.add_argument(
1201      '--validation_percentage',
1202      type=int,
1203      default=10,
1204      help='What percentage of images to use as a validation set.'
1205  )
1206  parser.add_argument(
1207      '--eval_step_interval',
1208      type=int,
1209      default=10,
1210      help='How often to evaluate the training results.'
1211  )
1212  parser.add_argument(
1213      '--train_batch_size',
1214      type=int,
1215      default=100,
```

```
1216        help='How many images to train on at a time.'
1217    )
1218    parser.add_argument(
1219        '--test_batch_size',
1220        type=int,
1221        default=-1,
1222        help="""\
1223        How many images to test on. This test set is only used once, to
     evaluate
1224        the final accuracy of the model after training completes.
1225        A value of -1 causes the entire test set to be used, which leads to
     more
1226        stable results across runs.\
1227        """
1228    )
1229    parser.add_argument(
1230        '--validation_batch_size',
1231        type=int,
1232        default=100,
1233        help="""\
1234        How many images to use in an evaluation batch. This validation set is
1235        used much more often than the test set, and is an early indicator of
     how
1236        accurate the model is during training.
1237        A value of -1 causes the entire validation set to be used, which leads
      to
1238        more stable results across training iterations, but may be slower on
     large
1239        training sets.\
1240        """
1241    )
1242    parser.add_argument(
1243        '--print_misclassified_test_images',
1244        default=False,
1245        help="""\
1246        Whether to print out a list of all misclassified test images.\
1247        """,
1248        action='store_true'
1249    )
1250    parser.add_argument(
```

```
1251        '--bottleneck_dir',
1252        type=str,
1253        default='/tmp/bottleneck',
1254        help='Path to cache bottleneck layer values as files.'
1255    )
1256    parser.add_argument(
1257        '--final_tensor_name',
1258        type=str,
1259        default='final_result',
1260        help="""\
1261        The name of the output classification layer in the retrained graph.\
1262        """
1263    )
1264    parser.add_argument(
1265        '--flip_left_right',
1266        default=False,
1267        help="""\
1268        Whether to randomly flip half of the training images horizontally.\
1269        """,
1270        action='store_true'
1271    )
1272    parser.add_argument(
1273        '--random_crop',
1274        type=int,
1275        default=0,
1276        help="""\
1277        A percentage determining how much of a margin to randomly crop off the
1278        training images.\
1279        """
1280    )
1281    parser.add_argument(
1282        '--random_scale',
1283        type=int,
1284        default=0,
1285        help="""\
1286        A percentage determining how much to randomly scale up the size of the
1287        training images by.\
1288        """
1289    )
1290    parser.add_argument(
```

```
      '--random_brightness',
      type=int,
      default=0,
      help="""\
      A percentage determining how much to randomly multiply the training image
      input pixels up or down by.\
      """
  )
  parser.add_argument(
      '--tfhub_module',
      type=str,
      default=(
          'https://tfhub.dev/google/imagenet/inception_v3/feature_vector/1'),
      help="""\
      Which TensorFlow Hub module to use.
      See https://github.com/tensorflow/hub/blob/master/docs/modules/image.md
      for some publicly available ones.\
      """)
  parser.add_argument(
      '--saved_model_dir',
      type=str,
      default='',
      help='Where to save the exported graph.')
  FLAGS, unparsed = parser.parse_known_args()
  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

C. label_image.py

```
# Copyright 2017 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse

import numpy as np
import tensorflow as tf


def load_graph(model_file):
  graph = tf.Graph()
  graph_def = tf.GraphDef()

```

```
  with open(model_file, "rb") as f:
    graph_def.ParseFromString(f.read())
  with graph.as_default():
    tf.import_graph_def(graph_def)

  return graph


def read_tensor_from_image_file(file_name,
                                input_height=299,
                                input_width=299,
                                input_mean=0,
                                input_std=255):
  input_name = "file_reader"
  output_name = "normalized"
  file_reader = tf.read_file(file_name, input_name)
  if file_name.endswith(".png"):
    image_reader = tf.image.decode_png(
        file_reader, channels=3, name="png_reader")
  elif file_name.endswith(".gif"):
    image_reader = tf.squeeze(
        tf.image.decode_gif(file_reader, name="gif_reader"))
  elif file_name.endswith(".bmp"):
    image_reader = tf.image.decode_bmp(file_reader, name="bmp_reader")
  else:
    image_reader = tf.image.decode_jpeg(
        file_reader, channels=3, name="jpeg_reader")
  float_caster = tf.cast(image_reader, tf.float32)
  dims_expander = tf.expand_dims(float_caster, 0)
  resized = tf.image.resize_bilinear(dims_expander, [input_height,
   input_width])
  normalized = tf.divide(tf.subtract(resized, [input_mean]), [input_std])
  sess = tf.Session()
  result = sess.run(normalized)

  return result


def load_labels(label_file):
  label = []
```

```
69    proto_as_ascii_lines = tf.gfile.GFile(label_file).readlines()
70    for l in proto_as_ascii_lines:
71      label.append(l.rstrip())
72    return label
73
74
75  if __name__ == "__main__":
76    file_name = "tensorflow/examples/label_image/data/grace_hopper.jpg"
77    model_file = \
78      "tensorflow/examples/label_image/data/inception_v3_2016_08_28_frozen.pb"
79    label_file = "tensorflow/examples/label_image/data/imagenet_slim_labels.
       txt"
80    input_height = 299
81    input_width = 299
82    input_mean = 0
83    input_std = 255
84    input_layer = "input"
85    output_layer = "InceptionV3/Predictions/Reshape_1"
86
87    parser = argparse.ArgumentParser()
88    parser.add_argument("--image", help="image to be processed")
89    parser.add_argument("--graph", help="graph/model to be executed")
90    parser.add_argument("--labels", help="name of file containing labels")
91    parser.add_argument("--input_height", type=int, help="input height")
92    parser.add_argument("--input_width", type=int, help="input width")
93    parser.add_argument("--input_mean", type=int, help="input mean")
94    parser.add_argument("--input_std", type=int, help="input std")
95    parser.add_argument("--input_layer", help="name of input layer")
96    parser.add_argument("--output_layer", help="name of output layer")
97    args = parser.parse_args()
98
99    if args.graph:
100     model_file = args.graph
101   if args.image:
102     file_name = args.image
103   if args.labels:
104     label_file = args.labels
105   if args.input_height:
106     input_height = args.input_height
107   if args.input_width:
```

```
    input_width = args.input_width
  if args.input_mean:
    input_mean = args.input_mean
  if args.input_std:
    input_std = args.input_std
  if args.input_layer:
    input_layer = args.input_layer
  if args.output_layer:
    output_layer = args.output_layer

  graph = load_graph(model_file)
  t = read_tensor_from_image_file(
      file_name,
      input_height=input_height,
      input_width=input_width,
      input_mean=input_mean,
      input_std=input_std)

  input_name = "import/" + input_layer
  output_name = "import/" + output_layer
  input_operation = graph.get_operation_by_name(input_name)
  output_operation = graph.get_operation_by_name(output_name)

  with tf.Session(graph=graph) as sess:
    results = sess.run(output_operation.outputs[0], {
        input_operation.outputs[0]: t
    })
  results = np.squeeze(results)

  top_k = results.argsort()[-5:][::-1]
  labels = load_labels(label_file)
  for i in top_k:
    print(labels[i], results[i])
```