TÉCNICO LISBOA

Automatic User Interface Generation

Gonçalo Correia de Matos

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Mário Alexandre Teles de Figueiredo
Eng. Hugo Miguel Ferrão Casal da Veiga

Examination Committee

Chairperson: Prof. José Alberto Rodrigues Pereira Sardinha
Supervisor: Prof. Mário Alexandre Teles de Figueiredo
Member of the Committee: Prof. Alexandre José Malheiro Bernardino

June 2022

To my beloved grandparents
Lisa and Zé, Roca and João

Acknowledgments

This dissertation would not have been possible without the support of many people to whom I would like to express my gratitude and appreciation.

First and foremost, I would like to thank my supervisors, Prof. Mário Figueiredo, Eng. Hugo Veiga, and Eng. João Lages for the perfectly balanced encouragement, guidance, and freedom throughout the thesis. You have provided me an unparalleled opportunity for growth and professional development. Being able to do research in this exciting field with your counseling, scientific feedback, and openness to new ideas has made this a truly enjoyable journey that I will never forget.

For a whole year I was supported by OutSystems, to which I want to thank the support, resources, and outstanding working conditions provided during my internship. It was thanks to the friendly and supportive working environment at OutSystems that I was able to continuously improve my work. I was lucky enough to find many people who provided valuable feedback, especially in the OutSystems AI team. I am grateful for the academic collaboration opportunity that OutSystems has yielded with Instituto Superior Técnico, mediated by Prof. Mário Figueiredo, and I hope this joint effort towards research work continues more vastly in future.

A special thanks goes out to my friends and colleagues who have offered their support and confidence during this journey, especially Nuno Calejo. This thesis was profoundly enriched by the amazing research work into the unknown that Nuno first conducted in this specific application of computer vision.

I also owe a deep debt of gratitude to every single person who took some of their time to help me with the crowdsourced task force of producing over fourteen thousand elements for the dataset. Your unique hand-drawn sketches were decisive for the success of this thesis. Thank you very much.

Nobody has been more important to me in this journey than the members of my family. I would like to thank my parents, sister, and brother, whose love, unrelenting encouragement, and continued support are with me in whatever I pursue. You have always stood by me through all my travails.

I also owe a deep debt of gratitude to my cousins, uncles, and aunts for helping me whenever they could, and always being a source of inspiration. Thank you for your love.

Last but not least, I would like to dedicate this thesis to my beloved grandparents Lisa and Zé, Roca and João, for their endless love, embodied integrity, and learning joy that have always emboldened me. Foremost in my mind is my paternal grandfather, Avô João, who passed away during the course of this thesis. Always and forever, your memory lives with me and all of us.

Resumo

Uma interface utilizador (IU) interativa e inteligível resulta sempre de um processo de desenvolvimento harmonioso. Corresponder às expetativas dos utilizadores relativamente à qualidade e funcionalidade de uma IU não depende apenas de princípios teóricos do desenho de interfaces, mas sim da qualidade de todas as fases de desenvolvimento que, tipicamente, são fastidiosas e requerem múltiplas iterações de tarefas monótonas e morosas. Uma das primeiras fases consiste na elaboração de um protótipo de baixa fidelidade. Essa fase é fulcral para o resultado final. A fim de tirar partido da agilidade e eficiência que os esboços conferem a todo o processo, esta tese propõe uma ferramenta baseada em aprendizagem automática que converte esboços feitos à mão de uma IU diretamente para código, permitindo assim gerar interfaces reais. A solução proposta consiste numa ferramenta de software que identifica os elementos desenhados utilizando visão computacional, avalia as respetivas posições e a hierarquia e, por fim, gera a IU pronta a ser utilizada. De todos os modelos testados, a melhor configuração obteve uma média de 98,7% de precisão média. Perante os resultados, concluiu-se que a ferramenta proposta pode ser usada como chave para a geração automática de interfaces e, assim, encurtar o ciclo de vida de desenvolvimento de sistemas. Permitir que programadores e designers avaliem os resultados dos esboços em tempo real não só torna a gestão de projetos mais eficiente como também reduz o tempo e os custos de colocação das aplicações no mercado, concebendo-se soluções robustas num período de tempo mais reduzido.

Palavras-chave: Inteligência artificial, Aprendizagem automática, Aprendizagem profunda, Visão computacional, Geração automática de código, Interface utilizador

Abstract

An interactive, straightforward user interface (UI) always derives from a seamless development process. Meeting user expectations for a high quality and functional UI goes way beyond respecting state-of-the-art design principles. It is the result of the whole design process, which tends to be unnecessarily laborious, requiring multiple iterations of unremarkable and time-consuming tasks. One of the first stages consists of drafting a prototype that is a schematic image of the screens. This stage is the key for the final result. Aiming to avail the agile and efficient experimentation afforded by hand-drawn sketches, this thesis introduces an automatic tool, based on machine learning, that converts hand-made UI sketches into code, and eventually generates the actual UI, maximizing the efficiency of the design process and greatly accelerating it. The proposed solution pipeline consists of a software tool that identifies the sketched elements using computer vision, evaluates both their position and hierarchy, and finally generates the corresponding UI, ready to be used. The top-performing testing fold setup scores 98.7% of mean average precision (mAP). Thus, this thesis can be used as a master key to automatically generate real-world interfaces in real time, hence shortening the System Development Life Cycle (SDLC) of software applications. Providing immediate feedback to developers and designers not only makes the project management process more efficient, but also reduces the time-to-market of applications, delivering meticulous and more substantial solutions in a shorter time.

Keywords: Artificial Intelligence, Machine Learning, Deep Learning, Computer Vision, Automatic Program Generation, User Interface

Contents

List of Tables

List of Figures

Nomenclature

AI	Artificial Intelligence.
AP	Average Precision.
CNN	Convolutional Neural Network.
COCO	Common Objects in Context.
CPU	Central Processing Unit.
CSS	Cascading Style Sheets.
CUDA	Compute Unified Device Architecture.
CV	Computer Vision.
DSL	Domain Specific Language.
FN	False Negative.
FP	False Positive.
GIoU	Generalized Intersection over Union.
GPU	Graphics Processing Unit.
HTML	HyperText Markup Language.
IDE	Integrated Development Environments.
IoU	Intersection over Union.
mAP	Mean Average Precision.
NN	Neural Network.
OCR	Optical Character Recognition.
OS	OutSystems.
ReLU	Rectified Linear Unit.
RPN	Region Proposal Network.

SDLC System Development Life Cycle.

SVM Support Vector Machine.

TN True Negative.

TP True Positive.

TS TypeScript.

UI User Interface.

UX User Experience.

VM Virtual Machine.

VOC Visual Object Challenge.

XML Extensible Markup Language.

YOLO You Only Look Once.

Chapter 1

Introduction

This Chapter provides an overview of this thesis by contextualizing its motivations and objectives. Section 1.1 describes the main challenges posed by the user interface prototyping process that motivated this thesis work. Section 1.2 is an overture of relevant related work, highlighting the limitations and challenges faced by recently implemented solutions. Section 1.3 presents the main contributions of this thesis work and the goals that the implemented solution aims to achieve. Finally, Section 1.4 outlines the structure of this document.

1.1 Motivation

Traditionally, building a user interface (UI) is known to be a tedious, prone-to-error and detail-driven task. Machine learning can be used to greatly accelerate front-end development by using more data and algorithms, but requiring less coding. So, building an automatic tool that could interpret a hand-made sketch and generate the actual UI would accelerate and improve the whole design process, providing the developer immediate feedback on what is being generated and allowing for changes to be made in real time.

A seamless, interactive, and straightforward design process would lead to a better user interface, benefiting both the developers and the users. This would also positively impact the System Development Life Cycle (SDLC) of software applications by reducing it. Further improving the manageability, objectivity, and control of projects, would ultimately reduce the time-to-market and the cost-to-market of applications, allowing developers and designers to deliver more accurate and tangible products in a shorter time.

1.2 Topic Overview

Automatic UI generation has recently been building momentum in software development, as more developers find themselves working on unnecessarily laborious, unremarkable, and time-consuming tasks that require multiple iterations and do not always lead to the very best result.

One of the first stages of building a UI consists of drafting a prototype that is a schematic image of the screens. For this purpose, the most common UI design prototyping solutions lead developers and designers to produce both hand-made and digital sketches. In order to accelerate this stage, some modern integrated development environments (IDEs), such as Apple Xcode, Google Android Studio, and Microsoft Visual Studio, provide built-in UI editors with standardized UI patterns and screen templates. However, these solutions do not provide a seamless, immediate, and easy way to generate the UI code based on the initial hand-made prototypes that often result from project brainstorming. That is because connecting the abstract hand-drawn sketches and the production of consistent UI code, which is a task usually performed by a specialized developer, is not linear. In fact, building a UI is more artistry and handiwork, than a methodical and scientific work. Depending on the developer, the ways of structuring the UI code can diverge a lot, even from the same desired output. This yields that establishing heuristics to perform such a task is not viable, leading to the use of machine learning for automatic UI generation.

While this is a new field of research, it has been propelling the creation of new companies, such as Uizard [1], and motivating new research and development projects in different companies, like OutSystems [2], teleportHQ [3], Microsoft [4], and Airbnb [5].

1.3 Objectives

This thesis proposes a tool that converts hand-drawn sketches into real world user interfaces, as depicted in Figure 1.1. In order to achieve this, the tool recognizes the representation of each UI element, infers their hierarchy and positions, and generates the corresponding user interface code.

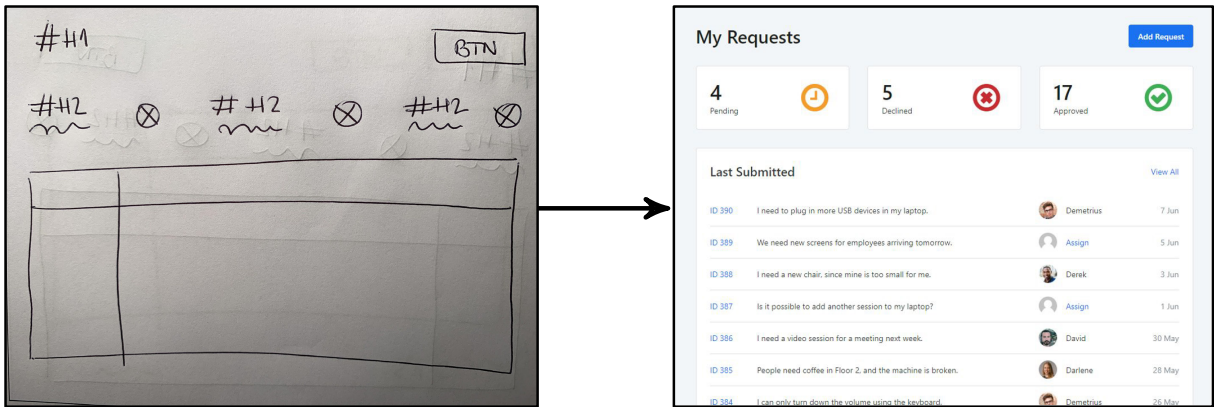


Figure 1.1: Conversion of a raw photo of a hand-drawn sketch (input) into a user interface (output).

While the ultimate objective of this tool is to maximize the efficiency of prototyping tasks and accelerate the design process, specific objectives were set for all the intermediate stages of the implemented solution pipeline, namely:

- Extract all relevant data from the input hand-drawn image.
  - Train a computer vision model to detect hand-drawn UI elements, which requires producing a dedicated dataset for the task at hand and conceiving the necessary tools for labeling and data augmentation.

  - Develop a pre-processing pipeline of steps to be applied to the input images before feeding the data to the network.

- Generate the code for the UI.
  - Analyze the extracted data to infer the hierarchical structure of the sketched UI elements in order to preserve the designed layout when generating the code.
  - Produce an agnostic structure with spatially grouped UI elements and correct potential errors in the trace set.
  - Generate the source code to be compiled and visualize the UI.

This thesis will cover how these objectives were pursued in each stage of the pipeline and how the complete tool can be used to generate the most commonly designed user interfaces.

1.4 Thesis Outline

Throughout the five chapters of this thesis, the implemented solution is described in detail along with the methodologies used. More specifically, Chapter 2 provides an overview of the background concepts on machine learning, deep learning, and methods used for image analysis. Chapter 3 focuses on the implemented solution, describing each stage of the pipeline in detail. Chapter 4 presents the results obtained. And, finally, concluding remarks are presented in Chapter 5, along with multiple future work possibilities.

Chapter 2

Background

Chapter 2 provides a brief top-down presentation of the core concepts that are essential to better understand the essence of the current state-of-the-art algorithms and tools used for object detection and image classification. It also covers the two main architectures for hand-drawn sketch analysis, by specifying the capabilities of the existing implementations and their limitations.

2.1 From Machine Learning to Deep Learning

The most common deep learning architectures used for computer vision and image analysis, such as convolutional neural networks (CNNs), which will be covered in detail in this chapter, are part of a broader family of machine learning methods. Thus, it is important to understand how they are related to each other.

2.1.1 Machine Learning

The core objective of a learning machine is to generalize from its experience [6]. Detecting patterns and adapting to new circumstances may not be possible using explicit instructions. This application of artificial intelligence relies on patterns and inference by developing algorithms that can help a program learn and identify patterns given a certain dataset [7].

In order to make predictions without being explicitly programmed to perform a task, machine learning algorithms build a mathematical model based on training data. Presently, machine learning algorithms are used for a wide variety of applications, including computer vision, many tasks of which would be infeasible using conventional algorithms [8].

2.1.2 Machine Learning Approaches

Depending on the problem to be solved, different types of machine learning algorithms can be used. The most important scenarios for this thesis work are:

- Supervised learning algorithms, which build a mathematical model from a pre-labelled dataset containing both the inputs and the corresponding desired outputs. The model is then used to predict new outputs for inputs that were not known before.

- Unsupervised learning algorithms, which do not use a labelled dataset for training. This approach builds a mathematical model just from the inputs by finding structure in the data (i.e., grouping or clustering data points).

- Semi-supervised learning algorithms, which fall in between the previous two cases, using datasets where a small part of the data is labeled and the rest is not. These algorithms pursue the idea that unlabeled data contains important information for the decision-making process [9].

2.1.3 Neural Networks

Neural networks (NNs) are a class of machine learning algorithms designed to recognize patterns [10]. These algorithms are inspired by the structure of the human brain and provide multiple ways of classifying and clustering data. The central unit of any neural network is called perceptron, which classifies an input vector by separating two different categories with an hyperplane. As described in Figure 2.1, the input of a perceptron is typically a feature vector x, which is then multiplied by weights w, added to a bias b, and passed through an activation function f, leading to the output y given by

$$y = f(w \cdot x + b), \tag{2.1}$$

where $w \cdot x$ denotes the inner product between vectors w and x.

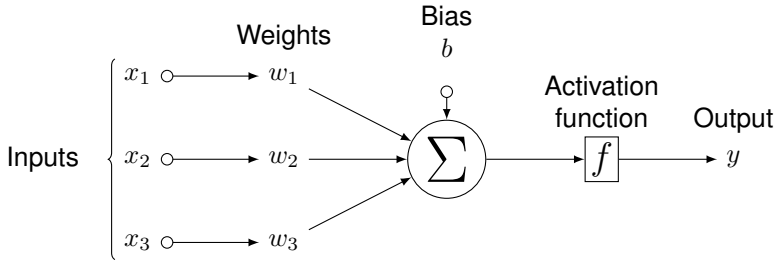


Figure 2.1: Schematic illustration of a neural network perceptron.

The combination of multiple perceptrons aiming to solve more complex problems leads to multilayer neural networks, which are typically grouped into three different types of layers: the input layer, the hidden layers, and the output layer. Figure 2.2 illustrates an example of a multilayer neural network.

The input layer restructures the data to be processed by the rest of the network. The hidden layers, between the input and output layers, perform most of the relevant computations, allowing the network to learn the features of the data using linear projections and activation functions. Finally, the output layer converts the output of the hidden layers into a meaningful output for the task in question.

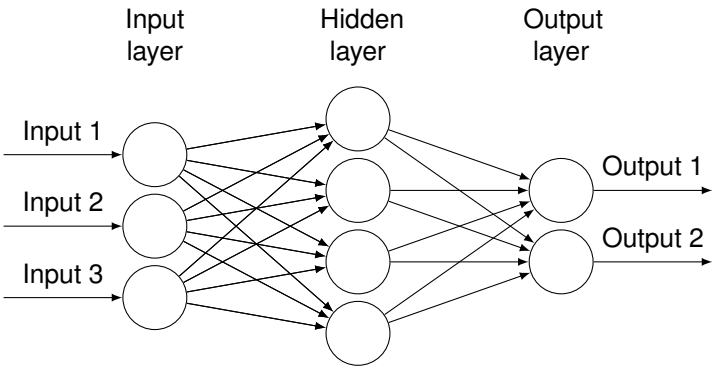


Figure 2.2: Example of a multilayer neural network.

2.1.4 Activation Functions

Neural networks are function-approximating models that can improve themselves with experience. In order to work effectively, NNs rely on activation functions to transform the values between each layer [11].

Rectified Linear Unit

The ReLU is among the most used activation functions; mathematically, it is defined as

$$ReLU(x) = \max(0, x). \tag{2.2}$$

The ReLU is the linear identity of any positive value, and zero for any negative value. This makes ReLU easy and fast to both compute and converge, considering that the slope does not plateau for a linear function.

Sigmoid Function

Another frequently used activation function is the sigmoid function, mathematically defined by

$$\sigma(x) = \frac{e^x}{1 + e^x}. \tag{2.3}$$

This activation function outputs a value between zero and one, making it especially convenient for problems involving probabilities.

Softmax Function

The softmax function normalizes an input vector of real values into a probability distribution with the same dimension of the input vector. Softmax is often used in neural networks to map the non-normalized output of a network into a probability distribution over predicted output classes (i.e., the vector components will sum up to one, and none of the values can be negative or greater than one).

The softmax function is defined as

$$(Softmax(x))_j = \frac{e^{x_j}}{\sum_i e^{x_i}}. \tag{2.4}$$

Loss Functions

Optimizing the parameters of a neural network consists of improving how well the neural network models the training data by minimizing the output result of a loss function that consecutively compares the target values with predicted values. Then, the hyperparameters are updated to minimize the average loss, formally given by

$$J(w^T, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}_i, y_i), \tag{2.5}$$

where L is the difference between the ground-truth y value and the $\hat{y}$ predicted value, considering the w^T weights and b biases that eventually minimize the value of J (average loss).

There are two main types of loss functions known as regression and classification loss functions, which are purposeful for the two main types of neural networks. Regression loss functions, like the mean squared error function, are used for regression neural networks, where given an input value the model predicts the corresponding output value. On the other hand, classification loss functions, like the cross-entropy function, are used in classification neural networks, where given an input value the neural network produces a vector of probabilities of the input belonging to a set of given categories, thus selecting the category with the highest probability.

Mean Squared Error

The mean squared error function computes the squared distances between a regression line and a given set of points. The squaring operation is necessary to both ignore the sign of the error and emphasize larger differences. In the context of neural networks, the distances represent the discrepancy between the output and target values. Formally,

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2. \tag{2.6}$$

Cross-Entropy

The cross-entropy is the average number of bits needed to encode data coming from a source with distribution p when using a model q [12]. Generally, considering a target or an underlying probability distribution p, and an approximation of the target distribution q, the cross-entropy of q from p is given by

$$H(p, q) = -\sum_{x \in \chi} p(x) \log q(x). \tag{2.7}$$

2.1.5 Backpropagation

The backpropagation algorithm is the classical learning mechanism of multilayer neural networks [13]. The first step of this algorithm is to initialize the weights and biases of the network, for example, with random values from a Gaussian distribution. Then, the input data is propagated forward through the network. After the network processes the data, the obtained output is compared with the desired output using the loss function, and the gradient, or error, is calculated. The gradient value is then propagated back through the network and the gradients of the loss function with respect to the parameters of the hidden layers are computed. Finally, the parameters of the network are updated based on the computation of its gradient and subtraction of a fraction of the gradient from the weights according to a specified learning rate. This update is repeated until some convergence criterion is satisfied.

2.1.6 Deep Learning

Different problems require different machine learning approaches. Deep learning is a subset of machine learning that uses neural networks to exploit the unknown structure of an input. Deep learning algorithms seek to discover good representations, often at multiple levels, with higher-level learned features defined in terms of lower-level features (i.e., complex representations are represented in terms of smaller representations). While extracting relevant features from an input with an unknown structure can be a great challenge, deep learning algorithms often solve the representation learning problem by representing complex concepts as multiple simpler concepts [14]. This is particularly useful for many different fields, including computer vision, natural language processing, bioinformatics, and many others. A deep learning model in the computer vision field may consist of a first visible layer that receives an input image, a subsequent set of hidden layers that identify multiple simpler concepts that are useful to establish relations between the data, and a final output layer that classifies the object found.

2.2 Convolutional Neural Networks

With the advancements in computer vision, it quickly became obvious that regular fully connected NNs (i.e., where each neuron is connected to all the neurons of the next layer) would not be the solution to perform all the necessary computations and avoid overfitting issues in a timely manner [15]. A CNN is formed by multiple layers of convolutional filters, alternated with subsampling filters, followed by fully connected layers [16]. The basis of the design of conventional CNNs, shown in Figure 2.3, was first introduced by LeCun *et al.* [17] to tackle the challenges posed by computer vision. More particularly, CNNs were created to solve a handwritten digit recognition problem.

When an input image is provided, a CNN does not know where the desired features are located in the image, so it runs through the whole image looking for every possible location, thus creating a filter that consists of a weighted average of all measurements, instead of the general matrix multiplication used by regular NNs. This process, represented in Figure 2.4, corresponds to a convolution.

When dealing with numerous convolutions on large kernel sizes and on large datasets within multiple

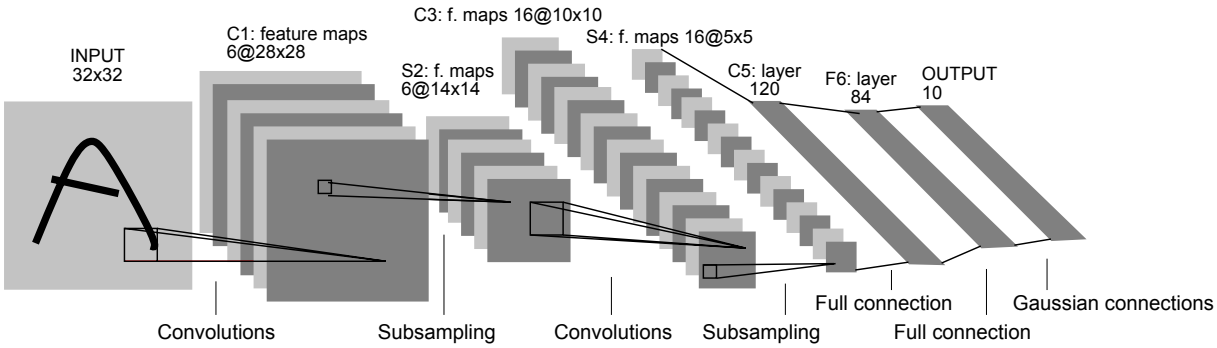


Figure 2.3: Illustration of the LeNet-5 CNN architecture, containing an input layer and six processing layers. Each plane is a feature map, i.e., a set of units whose weights are constrained to be identical [18].

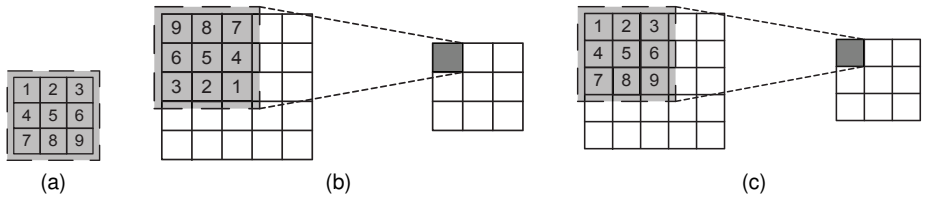


Figure 2.4: Illustration of a 2D discrete convolution operation: (a) example of a 3×3 convolution kernel, (b) convolution with kernel flipping, (c) convolution without kernel flipping [16].

training iterations, such as image analysis, replacing the convolution operation with cross-correlation reduces the computational time in both feedforward and backward propagations [16].

Formally, the general form of a 2D discrete cross-relation is

$$Y(x,y) = \sum_{u=0}^{K_x} \sum_{v=0}^{K_y} X(x+u, y+v) w(u,v), \tag{2.8}$$

where X is an input image, Y is the output image, w is the kernel, and K_x and K_y represent the width and height of the convolutional kernel, respectively. The difference between a 2D discrete convolution and a cross-relation is that the kernel weights are not flipped in the latter.

It is important to retain the three central stages of a CNN. The first stage consists of a layer that performs several convolution operations in parallel for feature extraction, which replaces the regular multiplication operations done in fully-connected NNs. The second stage is usually called “the detector stage” and corresponds to a feature map layer, where each linear activation function is run through nonlinear activation functions. Finally, the third stage consists of applying a pooling operation to downsample the features into feature maps and create a summarized version of the features detected in the input.

Convolutional Layers

Combining a set of convolutional filters forms a convolutional layer of the network. Because the filters used in the convolution operations are spatially invariant of the image, the number of free parameters needed is drastically reduced when compared to fully-connected NNs.

A convolutional layer of a CNN accepts a so-called tensor (i.e., a three-dimensional data cube) of

size $W_i \times H_i \times D_i$, and requires four hyperparameters: the number of filters K, the size of the filters F, the stride S, and the padding P. The output is a tensor of size $W_j \times H_j \times D_j$, where

$$W_j = \frac{W_i - F + 2P}{S} + 1, \quad H_j = \frac{H_i - F + 2P}{S} + 1, \quad \text{and} \quad D_j = K. \tag{2.9}$$

Nonlinear Transformation

The output of each convolutional layer is passed through a nonlinear transform, e.g., $ReLU(x)$, or the hyperbolic tangent function, which will map input values to perform nontrivial computations using a small set of nodes.

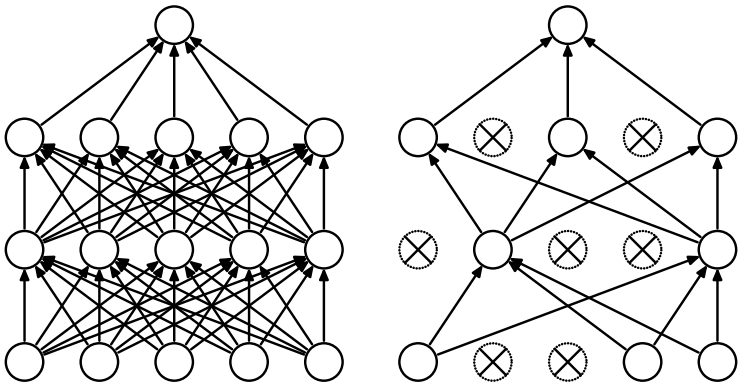
Pooling

Spatial pooling or downsampling techniques consist of grouping local features, such as per-pixel color measurements, in order to reduce the dimension of each feature, while retaining the most important information and avoiding overfitting. This technique is typically used to improve the robustness of objects against slight deformations, which is extremely relevant in computer vision [19]. Two main types of spatial pooling are applied: max pooling, which chooses the highest values from the selected pooling filter, and average pooling, which computes the average of the selected filters. It is important to retain that spatial pooling receives a tensor of size $W_i \times H_i \times D_i$, requires two hyperparameters (the spatial extent F and the stride S), and outputs a tensor of size $W_j \times H_j \times D_j$, where

$$W_j = \frac{W_i - F}{S} + 1, \quad H_j = \frac{H_i - F}{S} + 1, \quad \text{and} \quad D_j = D_i. \tag{2.10}$$

2.2.1 Training

After setting a CNN structure, learning its parameters (i.e., weights and biases) requires training [20], which typically consists of two phases. The first is a forward phase, where the input is passed through the network end-to-end, and the second is a backward phase, where gradients are backpropagated and the network weights are updated [21]. A supervised learning approach requires the CNN function to learn its parameters based on a provided set of input-output examples, which is called the dataset. Traditionally, training the network requires splitting the dataset into three different subsets: the training set, the validation set, and the test set. The training set is usually the largest one, as it is used to learn the weights and biases of the network. The validation set is used to verify which model and set of hyperparameters lead to the best results. Finally, the test set is used to assess the classification accuracy. In order to better train the network, a large number of labeled examples is required. Generally, during the training process, the network receives an example from the training set, performs all the necessary computations and, finally, compares the output of the network with the ground-truth label provided from the dataset, in order to update the parameters of the network. Assessing the difference between the predicted output and the correct label is crucial for improving the network. For that purpose, loss functions described in Equations (2.6) and (2.7) are an intuitive parameter to compute the difference



(a) Standard neural network (b) Dropout approach

Figure 2.5: Overview of the dropout model [22]. (a) A standard neural network with 2 hidden layers. (b) An example of a thinned network produced by applying dropout to the NN on the left. Crossed units ($\otimes$) have been dropped.

between the predicted output and the correct label. Updating the parameters is, then, an optimization problem for which we can use the backpropagation algorithm. Generally, the idea is to use the gradient of the loss function until some convergence criterion for a minimum value is satisfied. In order to use the backpropagation for every parameter, it is necessary to use the chain rule to compute the gradients.

Pre-processing

Building an effective network requires careful consideration of the input data format. Typically, image data input parameters are: total number of images, image height, image width, total number of channels, and total number of levels per pixel. It is important to assure that the data dimensions are of the same scale, in order to make the algorithm that receives the data more trainable. These parameters mean and standard deviation should only be computed over the training set and then applied consistently throughout the training, validation and test sets.

Regularization

Different model configurations are known to reduce overfitting, but require additional computational expense for training and maintaining different models. While CNNs avoid most overfitting issues of regular fully connected NNs [15], generalizing the network to unseen examples is still a great challenge. Some empirical techniques, known as regularization methods, are used to tackle this challenge. As an example, the dropout model can be used to simulate having a large number of different network architectures by randomly dropping out nodes during training [22].

2.2.2 From Image Classification to Object Detection

Image classification is an exemplary computer vision application that intends to appoint a label to a given input image from a pre-set array of categories. Before the deep learning era, most image classification methodologies were based on heuristics. Histograms of oriented gradient is a technique that counts the

occurrences of gradient orientation in localized portions of an image [23]. The detector window is tiled with a grid of overlapping blocks in which histograms of oriented gradient feature vectors are extracted. The combined vectors are fed to a linear SVM for object and non-object classification. The detection window used by this technique is scanned across the image at all positions and scales, and conventional non-maximum suppression is run on the output pyramid to detect object instances.

Object detection is an extension of image classification, consisting of detecting an object in an image and identifying its position and size in the image, apart from the usual image classification task [23]. However, object detection is one of the most challenging problems as it is prone to localization and classification errors. There are two main object detection approaches, which are reviewed next.

Region Proposals

One of the most common approaches is based on the technique of finding region proposals in order to localize objects. Despite having very good performance, this approach is computationally expensive due to having a large number of proposed regions [24].

The first research work to follow this approach is widely known by R-CNN [25], which proposed a method that uses selective search for extracting approximately 2000 regions from an image, instead of trying to classify a higher number of regions. These 2000 regions, called region proposals, are generated using a greedy algorithm to recursively combine similar regions into larger ones and produce the final candidate region proposals.

As depicted in Figure 2.6, the final candidates are warped into a square and fed into a CNN that produces a 4096-dimensional feature vector as output, thus acting as a feature extractor. Finally, the extracted features are fed into an support vector machine to classify the presence of the object within each candidate region proposal.

This algorithm also predicts four offset values to adjust and increase the precision of the object bounding box.

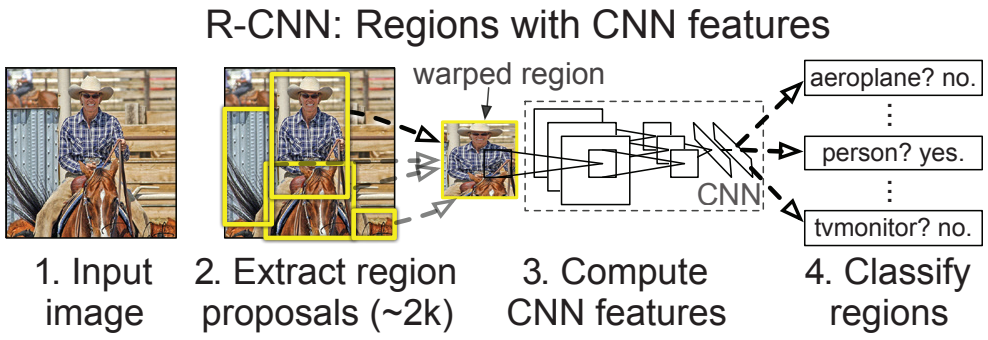


Figure 2.6: Overview of the proposed R-CNN object detection system [25], which (1) takes an input image, (2) extracts around 2000 bottom-up region proposals, (3) computes features for each proposal using a large CNN, and finally (4) classifies each region using class-specific linear SVMs.

However, R-CNN has some drawbacks, such as the time-consuming network training process to classify 2000 region proposals per image, which is aggravated by the static selective search that does not learn while selecting region proposals, ultimately leading to the generation of bad candidates. Also,

R-CNN cannot be used for real-time detections, as it takes approximately 47 seconds to test each single image.

There have been several iterations after R-CNN, aiming to develop a high-confidence region-based object detection framework using region proposals that boost up the classification performance with less computational burden, reducing processing time [26].

Fast R-CNN [27] is an iteration of R-CNN that propelled a similar yet faster object detection algorithm. This new approach consists of feeding the input image to the CNN to generate a convolutional feature map, instead of feeding the region proposals to the CNN. Regions of proposals are then identified from the convolutional feature map, warped into squares and reshaped into a fixed size using a region of interest pooling layer, so that they can be fed into a fully connected layer. Finally, from the region of interest feature vector, a softmax layer is used to predict the class of the proposed region and the offset values to adjust the bounding box, as shown in Figure 2.7.

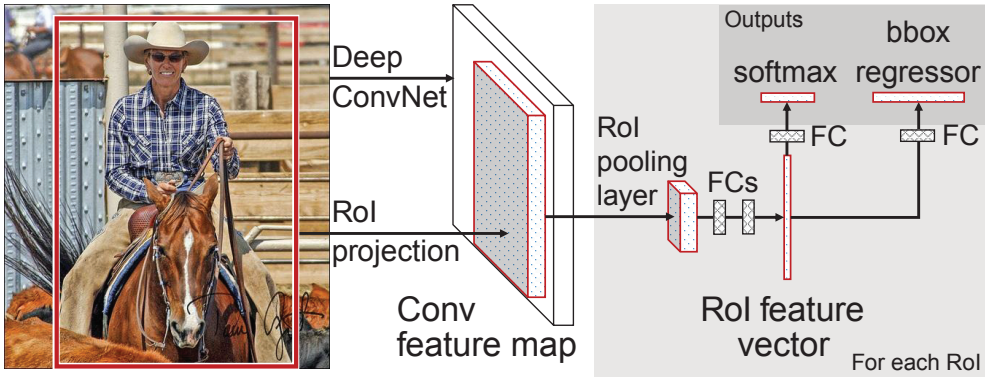


Figure 2.7: Illustration of the Fast R-CNN [27] architecture. A given input image and multiple regions of interest are input into a CNN. Each region of interest is pooled into a fixed-size feature map and mapped to a feature vector by fully connected layers. The network has two output vectors per region of interest: the softmax probabilities and the bounding-box regression offsets per each class. The architecture is trained end-to-end with a multi-task loss.

The main reason why Fast R-CNN is less time-consuming than R-CNN is because the convolution operation is done only once per image and a feature map is generated from it, rather than feeding 2000 region proposals to the CNN every time.

Even so, results show that including region proposals significantly slows down the algorithm [28]. Both R-CNN and Fast R-CNN use selective search to identify region proposals, thus affecting the performance of the network.

In order to tackle this bottleneck, a third important approach was introduced aiming to eliminate the selective search algorithm and let the network learn the region proposals, as shown in Figure 2.8.

Faster R-CNN is a similar approach to Fast R-CNN, yet consists of feeding an input image directly to a CNN, which provides a convolutional feature map. Then, instead of using the selective search algorithm on the feature map to identify region proposals, a separate network is used to predict the region proposals. At last, the predicted region proposals are reshaped using a region of interest pooling layer, which is then used to classify the image within the proposed region and predict the offset values

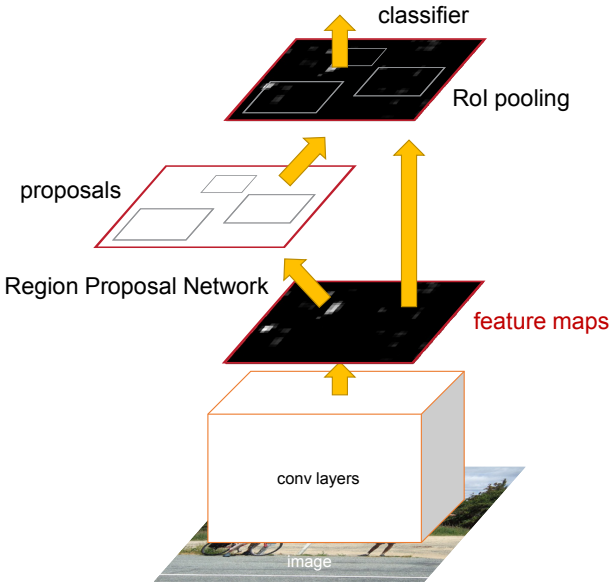


Figure 2.8: Overview of the Faster R-CNN [28] architecture. Faster R-CNN is a unified network for object detection composed of two modules. The first module is a deep fully convolutional network that proposes regions, and the second module is the Fast R-CNN detector [27] that uses the proposed regions.

for the bounding boxes.

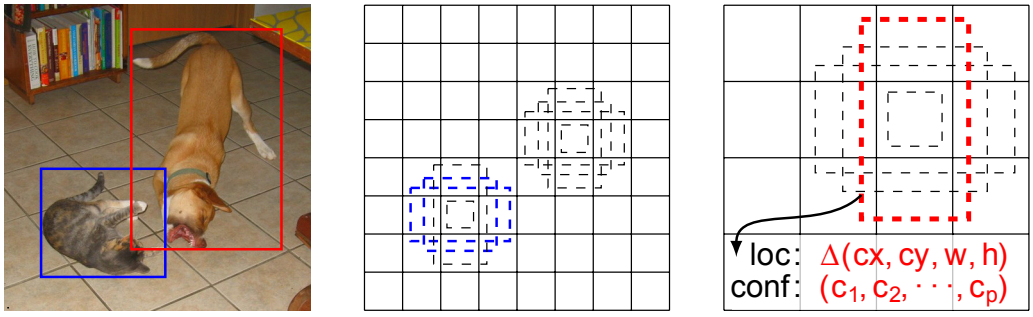
While Faster R-CNN was proven to be significantly more efficient in test speeds when compared with prior R-CNN approaches [27], it could only operate at 7 frames per second.

Therefore, this approach is still considered impracticable for real-time applications that require faster detections.

While all of these approaches increased detection speeds, these improvements came only at the cost of significantly decreased detection accuracy. Aiming to maintain high detection accuracy results, another important research work led to a new Single Shot MultiBox Detector (SSD).

The SSD approach was the first deep-network-based object detector that did not resample pixels or features for bounding box hypotheses and yet maintained the accuracy of approaches that do. It produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes, followed by a non-maximum suppression step to produce the final detections [29].

SSD only needs an input image with the ground-truth bounding boxes for each object. First, a small set of default boxes with different aspect ratios are evaluated at each location in several feature maps with different scales (e.g., 8×8 and 4×4, as depicted in Figure 2.9). Both the shape offsets and the confidences for all object categories are computed for each default box (e.g., $(c_1, c_2, \cdots, c_p)$). At training time, these default boxes are matched to the ground truth boxes and the model loss is a weighted sum between localization loss and confidence loss.



(a) Ground-truth boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 2.9: Demonstration of the SSD framework. In this example, two default boxes would be matched with the cat box and one with the dog box, which are treated as positives and the rest as negatives.

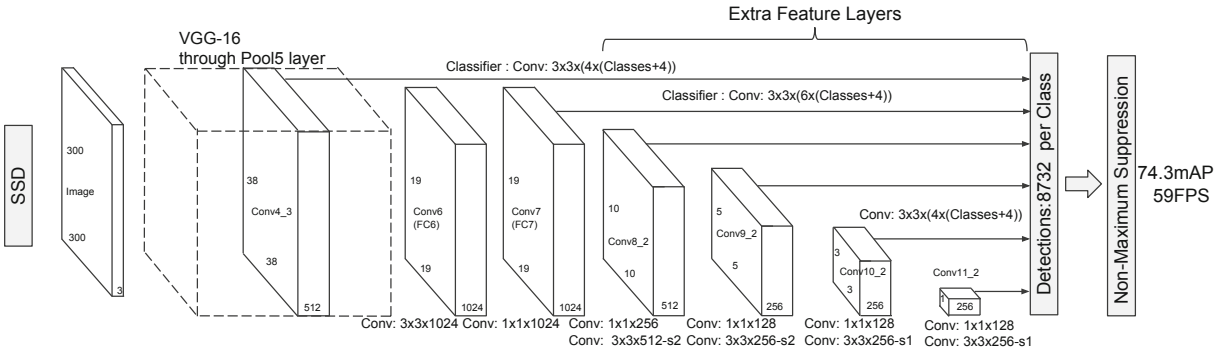


Figure 2.10: Overview of the SSD architecture [29], containing several feature layers in the end of a base network to predict the offsets of the default boxes with different scales and aspect ratios, as well as the respective confidences.

All in all, the SSD method is based on a feedforward convolutional network on which the early layers are based on a standard architecture used for high quality image classification, which is called the base network, and the final layers consist of an auxiliary structure, as depicted in Figure 2.10.

Regression

Another common approach to perform object detection is a one-step framework based in the regression task that consists of approximating input variables of a mapping function to a continuous output variable. The idea is to map the pixels of the image directly to bounding box coordinates and class probabilities. This approach achieves better performance when compared with frameworks based on region proposals, as it avoids several interdependent stages.

As an example, the You Only Look Once (YOLO) algorithm [30] frames object detection as a regression problem of spatially separated bounding boxes and associated class probabilities with a single network. Since the whole detection pipeline is a single network, it can be optimized end-to-end directly on detection performance. YOLO is considered a milestone in the development of target detection algorithms, such as RCNN, Faster-RCNN, and SSD, making it the most advanced real-time object detection model.

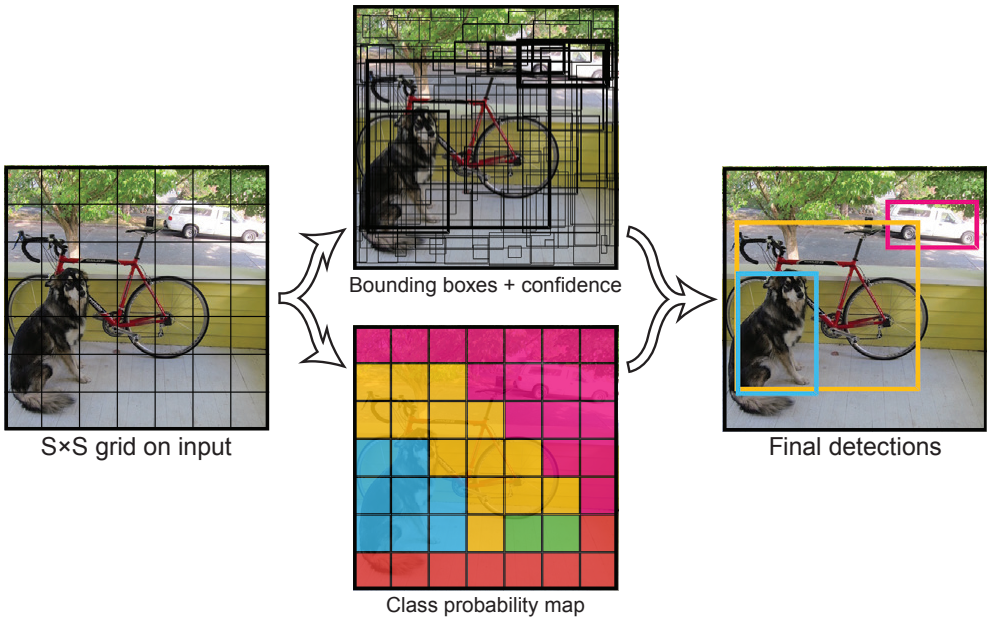


Figure 2.11: Illustration of a YOLO [30] model applying a 7×7 grid cell to an input image.

2.2.3 You Only Look Once

As aforementioned, YOLO is a real-time object detection algorithm that stands out from R-CNN by classifying and predicting bounding boxes directly from the image's pixels, instead of relying on region proposals. For common usage problems, YOLO integrates the object bounding box prediction and the object class judgment into a single neural network model, providing a significant speed improvement for the detection task.

In addition, a class probability is calculated from a set of conditional probabilities computed for each particular cell. Finally, the bounding boxes and their respective class probabilities are filtered to determine the final detections using a $S \times S \times (C + B \times 5)$ tensor that corresponds to the predictions of each cell of the grid, where C corresponds to class probabilities.

When provided an image, YOLO divides it into a $S \times S$ grid of cells (default 7×7, as depicted in Figure 2.11). Each grid cell predicts B bounding boxes, along with their respective position parameters, and C class confidence scores for the objects whose centers are located in it. All other cells disregard non-central parts of an object, even if seems to be visible across multiple cells.

The confidence score of each class reflects the probability of presence or absence of an object in bounding box. The confidence score is calculated by multiplying the presence probability of an object inside a cell and the intersection over union (IoU) of the object prediction box and the ground truth box. Because the probability of an object being inside a cell can vary between 0 and 1, the confidence score is close to 0 if an object does not exist in a cell.

Both during training and testing, YOLO can see the complete input image, thus making good use of context information while performing the detection and avoiding predicting nonexistent classes in the background.

Bounding boxes are the regions of interest (RoI) of the identified candidate objects, given by four

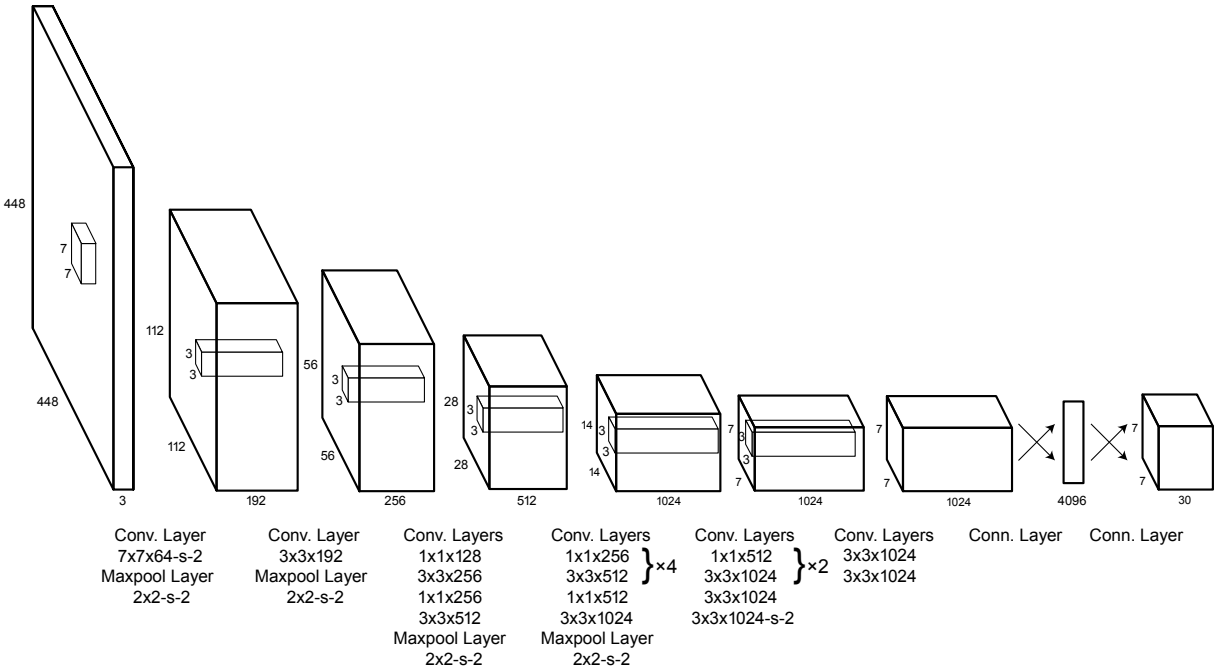


Figure 2.12: Illustration of the Fast R-CNN [27] architecture. A given input image and multiple regions of interest are input into a CNN. Each region of interest is pooled into a fixed-size feature map and mapped to a feature vector by fully connected layers. The network has two output vectors per region of interest: the softmax probabilities and the bounding-box regression offsets per each class. The architecture is trained end-to-end with a multi-task loss.

parameters (x, y, w, h), which correspond to the center coordinatex x and y, along with the box width and height, respectively. Combined with the confidence score aforementioned, each bounding box consists of five parameters.

Thus, the final layer output of YOLO is a tensor with the shape of $S \times S \times (5 \times B + C)$. For example, when evaluating the YOLO model for the COCO dataset, which contains 80 classes, and considering that each cell predicts 2 bounding boxes, the tensor output shape is $7 \times 7 \times (5 \times 2 + 80)$.

Because YOLO predicts multiple bounding boxes, it uses non-maximum suppression to ensure that only one box is detected per object, merging overlapping bounding boxes of the same object into a single one. This is achieved by discarding the boxes with low confidence scores and keeping the boxes with highest confidence scores. Finally, any of the remaining boxes that has an intersection over union (IoU) index higher than a certain threshold will also be discarded.

The YOLO model architecture, presented in Figure 2.12, namely the sequences of 1×1 and 3×3 convolutional layers, was inspired by the inception GoogLeNet model for image classification, which is mainly a combination of convolution and maxpooling layers that help reduce the features space from preceding layers. Nevertheless, the final layer uses a linear activation function instead of a Leaky Rectified Linear Unit (leaky ReLU) activation as all other layers.

Over time, YOLO has had several iterations and became faster and more reliable. Currently, there are five main versions of YOLO, including YOLOv1, YOLOv2, YOLOv3, YOLOv4, and YOLOv5.

The first iteration, YOLOv1, was developed on the basis of the R-CNN region proposals approach. As aforementioned, R-CNN uses a CNN for target detection and SVM for prediction classification, making

it computationally heavy and slow. However, the bounding boxes position detection and object classification accuracy values were high.

All in all, YOLOv1 tried to tackle the R-CNN drawbacks by only processing the input images once (thus the algorithm's name), extracting different features through multiple convolutional layers, and sharing convolution kernel parameters. This has improved image detection speeds, making it faster than previous detection models and, therefore, ideal for real-time applications. However, YOLOv1 had the disadvantage of recording a low accuracy score on position detections and ignoring small objects.

YOLOv2 [31] upgraded the YOLOv1 backbone network to use average pooling, softmax classification and an anchor prediction box. Also, a combined training method of target classification and detection is proposed. These improvements led to accuracy improvements, eespecially for the detection of small objects.

YOLOv3 [32] introduced some improvements over YOLOv2, increasing the depth of the network and to improve the model accuracy. Softmax classifiers were replaced with multiple logistic classifiers.

YOLOv4 [33] was presented in 2019, aiming to provide fast target detections that could be used in a real-world work environment. This iteration also used data enhancements and introduced the latest deep learning state-of-the-art activation functions, such as CutMix data enhancement, Swish and Mish activation functions.

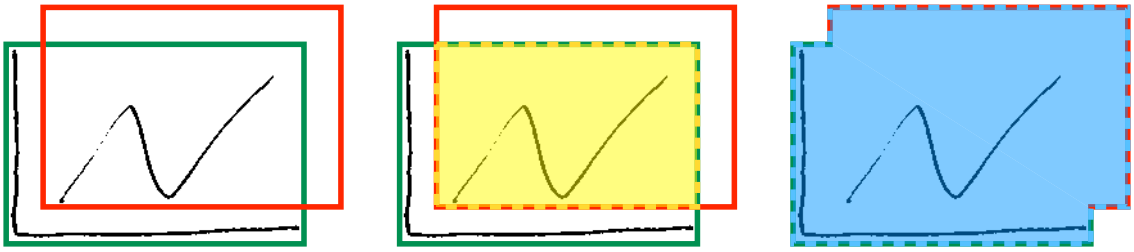
YOLOv5 [34] is currently the latest iteration of YOLO. This version introduced significant running speed improvements, with the fastest speed reaching 140 frames per second. At the same time, the size of YOLOv5 became smaller than previous iterations, with the weight file being nearly 90% lighter than the weights of YOLOv4, allowing YOLOv5 to be deployed on embedded devices. YOLOv5 also has a higher accuracy rate and even better capacities to identify small objects.

2.2.4 Evaluation metrics

Evaluating the performance of object detection algorithms requires a quantitative analysis of key performance metrics. The most commonly used evaluation metric for object detection algorithms is mean average precision (mAP), which relies on several important concepts which will be overviewed in this section.

Intersection over Union

Measuring the intersection over union (IoU), also known as Jaccard index, of a given detection requires considering two areas: the ground-truth bounding box (denoted by BB_{GT}) and the bounding box predicted by the model (denoted by BB_P), both illustrated in Figure 2.13.



(a) Chart ground-truth (green) and predicted (red) bounding boxes.

(b) Area of intersection (yellow).

(c) Area of union (blue).

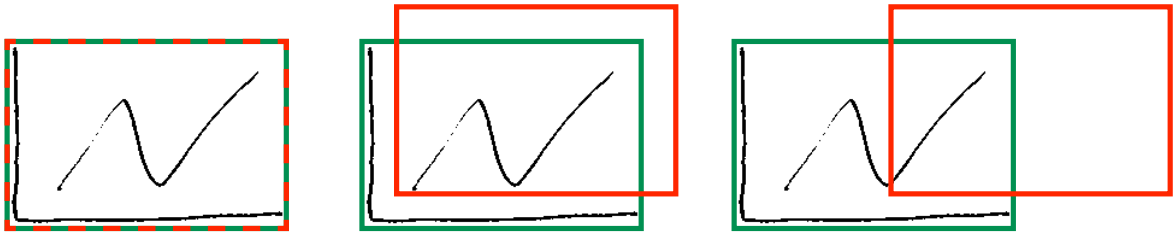
Figure 2.13: Illustration of the intersection and union areas needed to compute the IoU evaluation metric for a hand-drawn representation of a chart detection.

IoU is then calculated by dividing the intersection area of BB_{GT} and BB_P by the union area of BB_{GT} and BB_P, as follows [35]:

$$IoU = \frac{BB_{GT} \cap BB_P}{BB_{GT} \cup BB_P}. \tag{2.11}$$

True and false detections

Determining true and false detections is indispensable to assess the precision of a model. Applying an IoU threshold against a set of predicted and ground-truth bounding boxes is the most common approach to determine valid detections.



(a) Perfect detection, considering that the ground-truth (green) and predicted (red) bounding boxes are coincident ($IoU = 1$) .

(b) True detection, where the IoU score for the ground-truth and predicted bounding boxes is above the 50% threshold.

(c) False detection, where the intersection over union score for the ground-truth and predicted bounding boxes is below the predefined 50% threshold.

Figure 2.14: Illustration of a perfect detection, a true detection and a false detection for a chart element.

A detection is considered to be a true positive when the computed IoU index meets or exceeds the predefined threshold, generally of 50%.

Contrarily, a false positive detection indicates that the predicted bounding box had no associated ground-truth bounding box (e.g., an incorrect detection, where the predicted bounding box is excessively deviated from the ground-truth bounding box, thus not meeting the IoU threshold). False negative detections are also a possibility, meaning that a ground-truth bounding box is not associated to any predicted bounding box (e.g., no element is detected).

Evaluating the performance of a classification model requires correlating the instances that belong an actual class with the instances appointed for a predicted class.

A 2×2 confusion matrix, shown in Table 2.1, is a table layout that provides an intuitive visualization of all four kinds of detections (true positives, false negatives, and false positives, described earlier, along with true negatives, which correspond to any other detection of an empty background area).

Table 2.1: Binary confusion matrix using four kinds of results.

		Predicted value		
		Positive	Negative	Total
Ground-truth value	Positive	TP	FN	Actual positives
	Negative	FP	TN	Actual negatives
	Total	Positive predictions	Negative predictions	

Precision and Recall

While visualizing the results by comparing the predicted and ground-truth values is an intuitive way of evaluating the performance of a model, it is also worthwhile to evaluate two metrics that characterize any model. The metric that effectively describes the purity of our positive detections relatively to the ground-truth values, which means the total of predictions had a matching ground truth annotation, is Precision, described as:

$$Precision = \frac{TP}{TP + FP}. \tag{2.12}$$

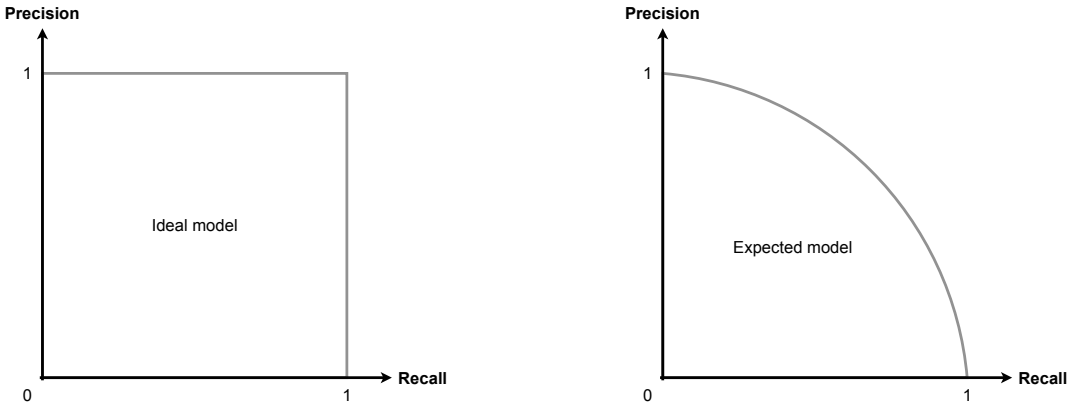
A perfect Precision score of 1.0 means that there is a high likelihood that a model prediction is a correct prediction. Nevertheless, the precision of a model does not describe the completeness of the positive predictions relatively to the ground-truth values. Evaluating how many actual objects were actually predicted accordingly to their ground-truth values requires using Recall, which is given by:

$$Recall = \frac{TP}{TP + FN}. \tag{2.13}$$

Likewise, a perfect Recall score of 1.0 means that a model will positively detect almost all value, whence an ideal model with both high Precision and Recall would be a perfect object detector, capable of correctly predicting all existing ground-truth values [36].

Although having a perfect model is not a realistic scenario, two pessimistic scenarios must be considered to better understand the correlation between both metrics. On one hand, having a high Recall value and a low Precision scenario implies that most ground-truth values have been detected, but also that most detections were incorrect (i.e., high number of false positives). On the other hand, having a high Precision value and a low Recall value yields that all predicted values were correct, but most ground-truth values have been missed (i.e., high number of false negatives).

Describing the correlation between both metrics results in a curve plot of precision and recall values for all the detections with different confidence scores predicted by the model, as shown in Figure 2.15.



(a) Ideal Precision-Recall curve with 1.0 values.

(b) Expected Precision-Recall curve.

Figure 2.15: Precision-Recall correlation curve plots for ideal and expected scenarios.

Average Precision (AP)

The correlation between Precision and Recall can also be described using AP [37]. The general definition of this metric is the area under the Precision-Recall curve, which can be calculated using:

$$AP = \int_0^1 P_{interpolated}(r)dr. \tag{2.14}$$

All in all, measuring Average Precision requires generating all prediction scores, converting those scores into class labels, calculating the confusion matrix with TP, FP, TN, and FN values, computing the precision and recall metrics, and finding the area under the precision-recall curve.

Precision and Recall are always between 0 and 1, so AP also ranges between 0 and 1. There are, however, three different approaches to compute AP, namely a 11-point, 40-point, and all-point interpolation method.

The 11-point interpolation method consists of plotting a Precision-Recall curve that summarizes the average precision values across a set of 11 different recall values of $R_{11} = \{0, 0.1, 0.2, \ldots, 1\}$. Similarly, the 40-point interpolation approach consists of computing recall points at 40 equally spaced points of $R_{40} = \{\frac{1}{40}, \frac{2}{40}, \frac{3}{40}, \ldots, 1\}$, providing a clearer evaluation of the model. For both methods, AP is given by:

$$AP_R = \frac{1}{\#R} \sum_{r \in R} P_{interpolated}(r), \quad \text{where} \quad P_{interpolated}(r) = \max_{r' \geq r} P(r'). \tag{2.15}$$

At each Recall level, each Precision value is replaced with the maximum Precision value to the right of each Recall level, meaning that the maximum precision value at recall value greater than equal to r is taken, rather than averaging over actual observed precision values through point r.

In the all-point interpolation method, the Precision *vs.* Recall curve is summarized by average precision values at all points instead of just eleven points.

$$AP_{all} = \sum_{n}(r_{n+1} - r_n)P_{interpolated}(r_{n+1}), \quad \text{where} \quad P_{interpolated}(r_{n+1}) = \max_{r' \geq r_{n+1}} P(r'). \tag{2.16}$$

In this case, instead of using the precision observed at only few points, the AP is now obtained by interpolating the precision at each level, taking the maximum precision whose recall value is greater or equal than r_{n+1}.

Mean Average Precision (mAP)

The mAP measures the accuracy of an object detector over all classes in a specific dataset. Usually, mAP is just the average of AP for each class, which is given by:

$$mAP = \frac{1}{n}\sum_{i=1}^{n} AP_i, \quad \text{for } n \text{ classes.} \tag{2.17}$$

However, the interpretation of AP and mAP varies in different contexts. For instance, under the COCO challenge evaluation, there is no difference between AP and mAP [38].

2.3 Dataset Generation

Common deep learning algorithms require labelled data in order to learn and generalize to unseen scenarios. One of the challenges of training neural networks is to collect a large labelled dataset, which is usually not available for many specific models. So, generating a large, varied, and realistic dataset tends to be a laborious task, especially for hand-drawn sketches, considering that large amounts of labelled data are not readily available.

This thesis work aims to develop a system that can recognize different hand-drawn UI elements and screen templates, which means that a dedicated dataset for this specific task had to be developed. To this end, there are different possible approaches.

2.3.1 Human-generated Dataset Approach

The first possible approach is to produce real hand-drawn sketches, manually scan them, and label one by one, leading to a dataset with examples that best resemble the final application. However, handing this task to a single person would be extremely laborious and unfit for the timeframe of the thesis work.

Several researchers believe that this is the best option, and so tried to tackle the time-consuming disadvantage by crowdsourcing the drawing, scanning, and labelling tasks [39]. Therefore, the only viable option to generate a large enough dataset is to challenge a community of volunteers and request contributions to the dataset.

2.3.2 Computer-generated Dataset Approaches

Another possible approach is to *sketchify* real user interfaces and, based on the source code, extract the necessary tags for the labelling task. This approach could be applied to this thesis work by building a program that replaces the UI elements in the source code with their respective hand-drawn representation, in order to generate samples that look like real human hand-drawn sketches, and then infer the label of each UI element by analyzing the original source code. Some researchers have committed to this approach for simpler scenarios where there is a single object in an image, as shown in Figure 2.16 [40].

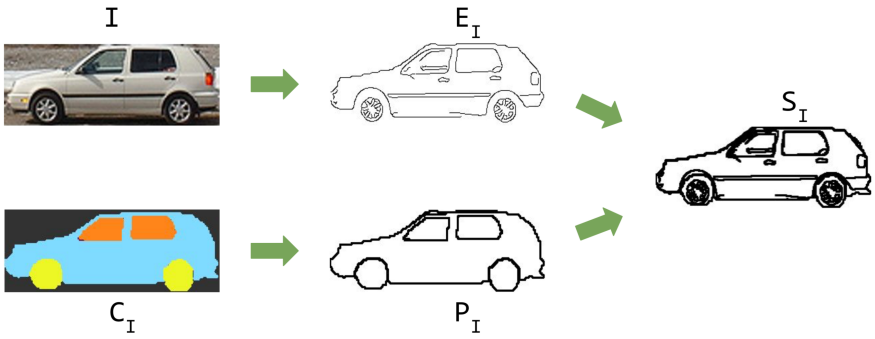


Figure 2.16: Illustration of the *sketchification* approach [40]. The edge image E_I, corresponding to the input photo I, is merged with the part and object contours P_I, derived from ground-truth label C_I, to obtain the final *sketchified* image S_I.

Other than *sketchifying* real user interfaces, a possible approach is to build a program that generates simulated hand-drawn UI sketches. This option has the great advantage of producing a scalable dataset, considering that a computer program can effortlessly generate different UI designs and the labelling task can be done instantly. On the other hand, it is relatively hard for a computer to simulate human hand-drawn sketches, due to their inherent abstract nature and variability.

2.3.3 Data Augmentation

Augmenting the training set by generating more samples has shown to improve object detection performance [41]. For both human- and computer-generated datasets, common techniques of data augmentation may be used to transform images, creating additional samples that are still realistic and can be used in the training process.

Usually, using data augmentation consists of performing simple transformations, but there are more ways to generate data by sampling images from models of the object to recognize. The most common data augmentation operations consist of geometric manipulations, as shown in Figure 2.17, that can be easily implemented.

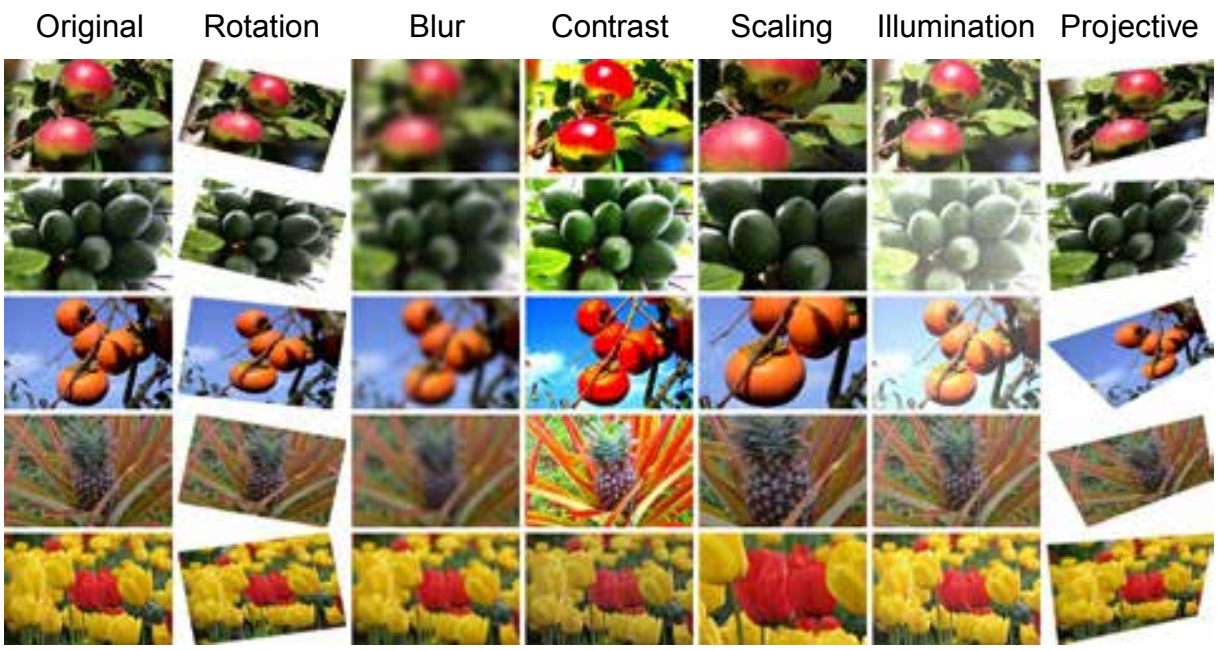


Figure 2.17: Effects of common data augmentation techniques on the AgrilPlant dataset [42].

Considering the variability intrinsic to hand-drawn sketches, morphological operations such as dilation and erosion can be effective to produce additional realistic samples of the same sketch by varying the stroke width, thus simulating the use of different pens.

However, it is important to retain that, in order to be impartial, the test set should only contain true hand-drawn sketches, since this will be the scenario on which the system will ultimately be used in real applications.

2.3.4 Morphological Operations

Nonlinear mathematical morphology is capable of providing more sophisticated image processing techniques through morphological operations, offering greater flexibility and better results than traditional data augmentation geometric manipulations [43]. However, optimizing the pipeline delay process and reducing the latency of morphology operators is still an active research field [44].

Morphological image operators take advantage of a wide range of algorithms for edge detection, noise removal, and image segmentation, producing image modifications based on the neighborhood of a pixel. For any given input image, a morphological operation consists of applying a structuring element that generates an output image of the same size, meaning that the value of each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbors.

Any morphological image processing technique is composed of two base operations, dilation and erosion, shown in Figure 2.18. The dilation operation increases forefront object boundaries, while the erosion operation increases background object boundaries. These operations are dual, meaning that dilating background objects is equivalent to erode foreground objects.

(a) Original image. (b) Dilation operation. (c) Erosion operation.

Figure 2.18: Demonstration of the two base morphological operations.

For sketches, other techniques can also be used, for example, by applying Bezier pivot deformation (BPD), as shown in Figure 2.19, as well as morphological operations, namely dilation, erosion, small rotations, mirroring, rescaling, among others, depicted in Figure 2.18 [45].

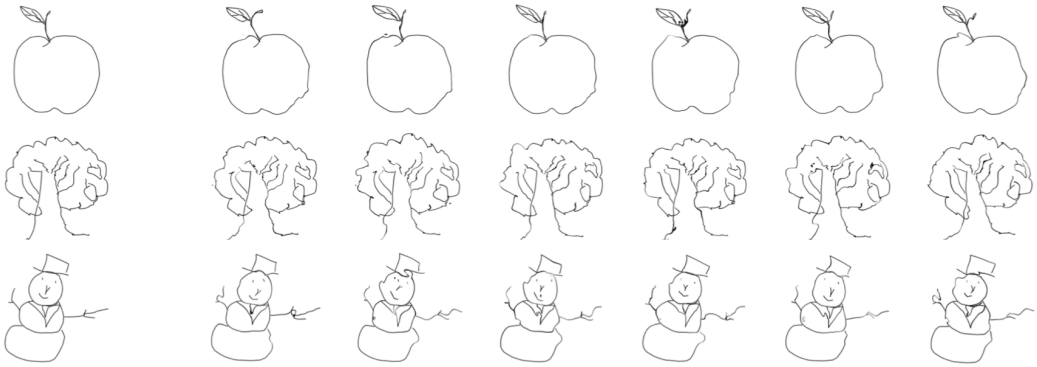


Figure 2.19: Examples of sketches generated by the proposed BPD approach. The left image of each row is the original hand-drawn sketch of the TU-Berlin dataset [46]. The other 6 samples are deformed sketches generated by BPD.

2.3.5 Cross-Validation

One of the most common techniques for model evaluation and model selection in machine learning practice is k-fold cross-validation [47], sometimes referred to as the train/test holdout method.

The k-fold cross-validation consists of going through training and validation stages in successive rounds for k times. The main idea is that each fold of a dataset must have the opportunity to be tested. In each round, the dataset is splitted into k folds, 1 being used as a validation set, and the remaining $(k-1)$ folds being merged and used as the training set, as shown in Figure 2.20 for a 5-fold cross-validation example.

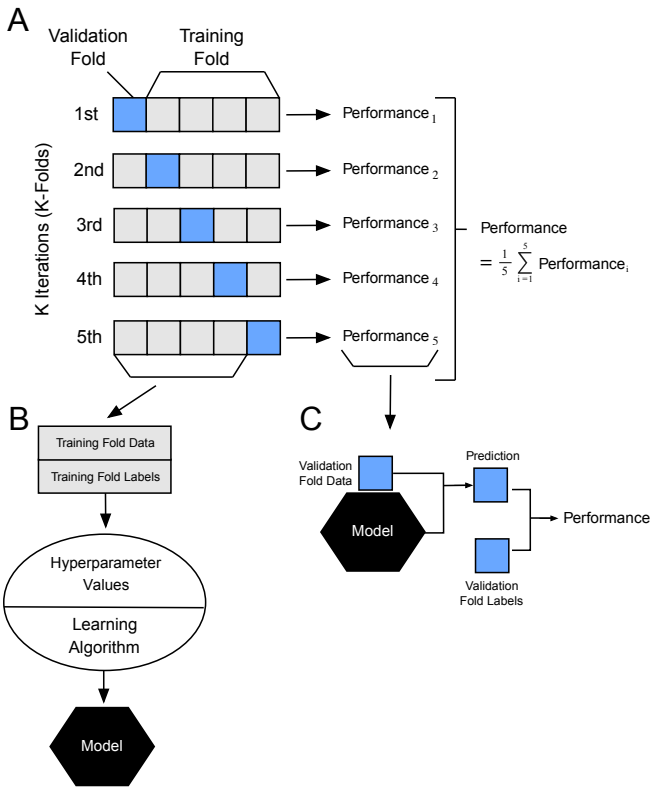


Figure 2.20: Illustration of the k-fold cross-validation procedure [48].

All in all, this approach consists of using a learning algorithm with fixed hyperparameters to fit the object detection model to the training $(k-1)$-folds in each iteration. Considering a 5-fold cross-validation scenario, this approach would result in five different models fit to different training subsets of the original dataset, yet partly overlapped by other folds training subsets and evaluated by non-overlapping validation subsets. The cross-validation performance is, then, the arithmetic mean over the k-fold performance estimations for the validation sets.

Although this process is computationally expensive, it can be useful for small datasets, where withholding data from the training set would be too wasteful. For a crowdsourced human-generated dataset, this is also a viable option, considering the crowdsourcing process limitations.

2.4 Program Generation

The perception of a need for domain-specific program generation is emerging [49]. So far, the main aims of program generation have been programming convenience and reliability. Program generation can substantially contribute to reduce the production cost and time-to-market of software development, while at the same time improve the quality and stability of a system.

Although the generation of computer programs is an active research field, program generation from visual inputs is still a nearly unexplored research area [50].

In this field, however, a straightforward approach is to use a domain specific language (DSL). The versatility of a DSL makes it easy to be compiled and, ultimately, generate the code [51]. In this thesis

context, program generation is inextricably linked to the image analysis stage of the system. Once the model outputs the drawing primitives, a domain-specific language (DSL) could take the outputs of the object detection algorithm (e.g., the UI elements position, size, and class) and generate an agnostic structure with all identified elements.

An illustrative example of a DSL approach describing a user interface image is represented in Figure 2.21, where, for instance, the footer is described as containing four different buttons and the first row of the UI is described as containing two objects: a label, and a switch.

(a) UI screenshot

```
stack {
   row {
      label, switch
   }
   row {
      label, btn-add
   }
   row {
      label, slider, label
   }
   row {
      img, label
   }
}
footer {
   btn-more, btn-contact, btn-search, btn-download
}
```

(b) DSL code describing the UI

Figure 2.21: Illustration of a user interface described in a markup-like DSL [50].

As long as the image analysis stage does not output wrong or insufficient information to generate the code, the simplicity of a DSL approach may lead to fewer mistakes. However, this is a narrow approach, limited to the size of the defined DSL. It will not be able to identify or correct errors that the image analysis stage may produce, nor infer features of the code to be generated.

A more sophisticated approach is to use a model that combines deep learning techniques and program synthesis [52]. This approach learns a convolutional neural network that proposes drawing primitives of an image and then learns a model that uses program synthesis techniques to recover a graphics program from that specification. The main advantage of this approach is that, with a graphics program in hand, it is possible to correct errors made by the network, measure similarity between drawings, and extrapolate drawings.

Chapter 3

Implementation

Sustained by the core concepts presented in Chapter 2, and motivated by the state-of-the-art algorithms, methodologies and tools described, a pipeline of multiple stages was developed to recognize hand-drawn UI elements, infer their hierarchy, and generate the corresponding user interface.

While the central stage of the implemented pipeline is the object detection model, training it required a large and varied dataset, leading to the development of multiple auxiliary stages. Generating the code of the resulting sketched interface also required conceiving dedicated stages for both outputting the hierarchy DSL and the code generation task.

All stages were developed following a modular approach and an agnostic implementation, allowing for agile modifications, upgrades, and continuous tests and improvements. Thus, the ultimate result is never compromised in any way, but rather constantly refined.

3.1 Dataset

The ultimate goal of the system is to identify different elements in a hand-drawn interface using computer vision. Similarly to other deep learning algorithms, computer vision models require large amounts of labelled data, which are not promptly available.

The abstract nature and high variability of hand-drawn sketches amplify the importance of having a large dataset covering the most diverse styles of hand-drawing possible [53]. While collecting large amounts of labeled hand-drawn sketches for a very specific use case can be a great challenge, combining both human- and computer-generated records lead to more realistic results.

Nevertheless, the essence of computer-generated records will always be human-generated samples, regardless of the approach chosen from the examples covered in Chapter 2.

The complete dataset used is the result of an iterative process, but the first and most important step was to define which UI elements would be considered. To do that, a detailed analysis to all screen templates included in the OutSystems UI framework [54] was conducted.

3.1.1 Screen Templates and User Interface Elements

The OutSystems UI framework includes a wide variety of adaptive and interactive UI elements that compose screen templates [55]. OutSystems has identified the most commonly designed user interfaces [54] and, using the 83 UI elements included in the OutSystems UI framework, set 17 different use cases that were materialized in ready-to-use screen templates, such as the *Bulk Actions* screen shown in Figure 3.1.

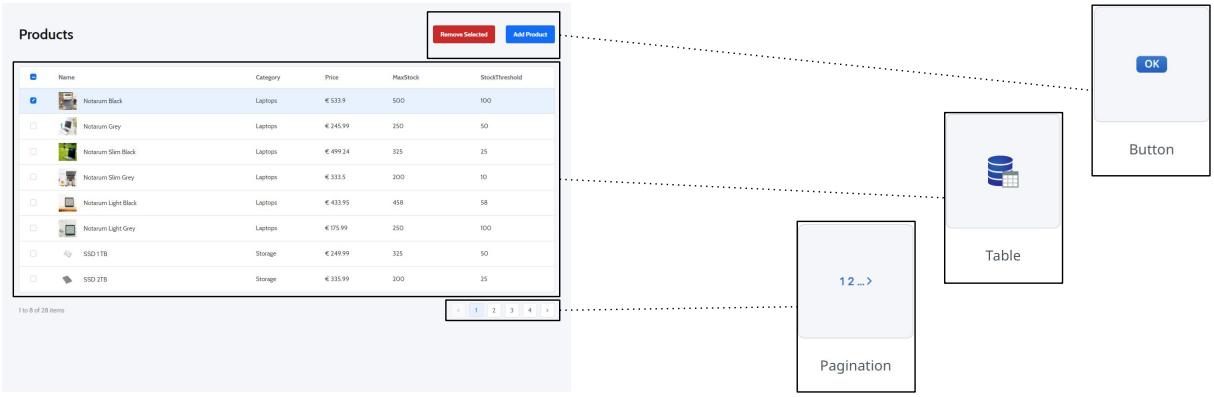


Figure 3.1: Breakdown of the *Bulk Actions* screen template into three illustrative user interface elements from the OutSystems UI framework.

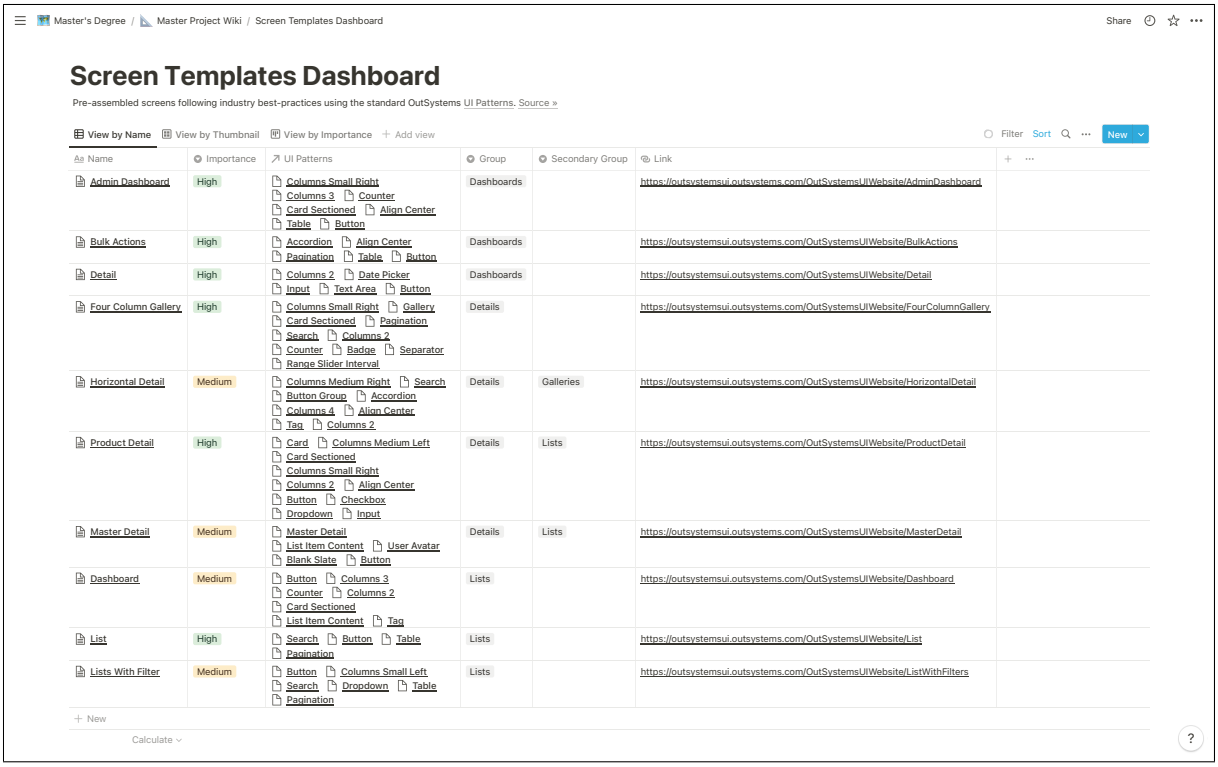
Deciding which UI patterns needed to be included in the dataset required a meticulous analysis of the complete OutSystems UI library, matching each screen template with all the respective contained UI elements.

This curation required not only an individual analysis of each screen template but also defining their importance, identifying the corresponding main category and subcategory, and linking them to the available OutSystems documentation page. The individual analysis of each UI patterns also required a categorization by group, a purposefulness rank, all the potentially redundant elements and all the screen templates using them.

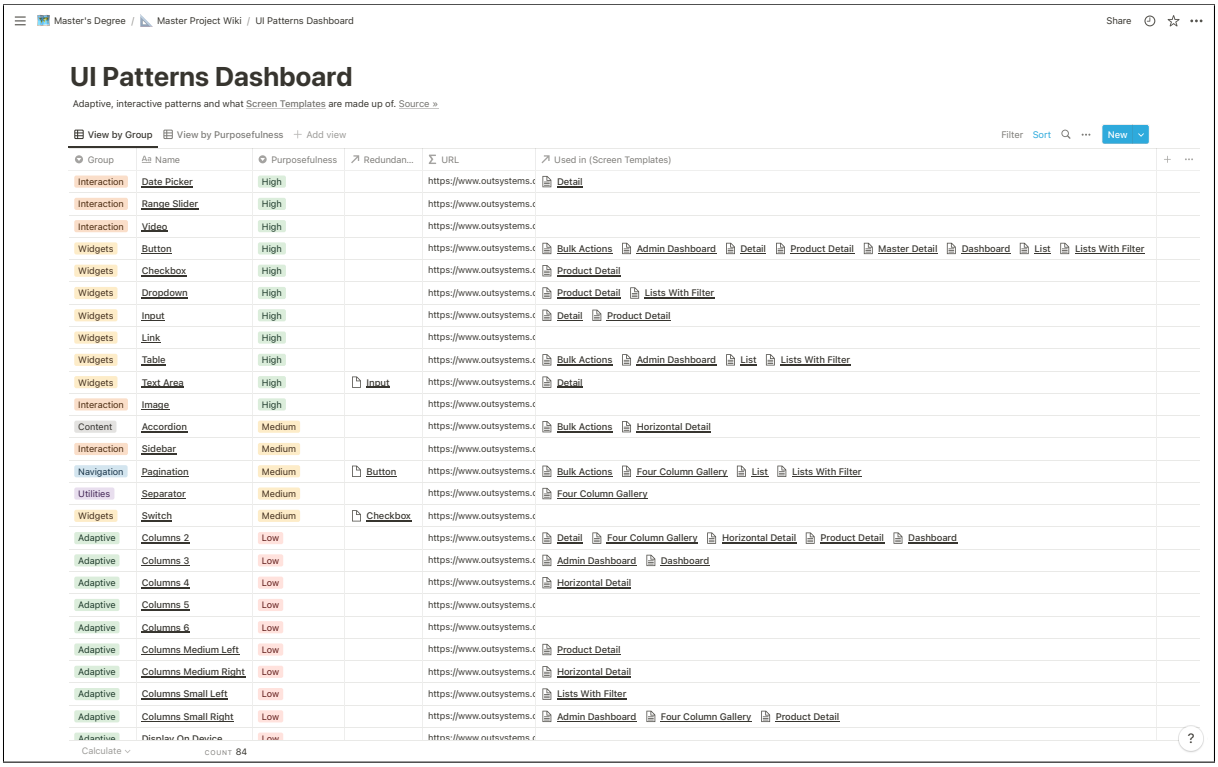
Centralizing all of this information led to the development of two linked dashboards, as shown in Figure 3.6, that could easily reflect the inextricable association between screen templates and UI patterns. These dashboards also served as a centralized repository to prepare future work on subsequent pipeline stages, namely extracting visual CSS attributes for the code generation stage.

After mapping all screen templates, a set of 16 prominent user interface elements was chosen based on the number of uses and number of ambiguous and/or redundant elements. These criteria were fine-tuned along with the dataset generation process progress, in order to achieve the best results.

While not all screen templates use all the 16 elements, they are simply different combinations of these, meaning that this set of elements represent the 16 most used elements that can coexist in a user interface in almost any imaginable way, amplifying the dataset reach.



(a) Screen templates dashboard screenshot.



(b) UI patterns dashboard screenshot.

Figure 3.2: Dashboards needed for the in-depth analysis of the OutSystems UI Framework.

3.1.2 Hand-drawn Representation of User Interface Elements

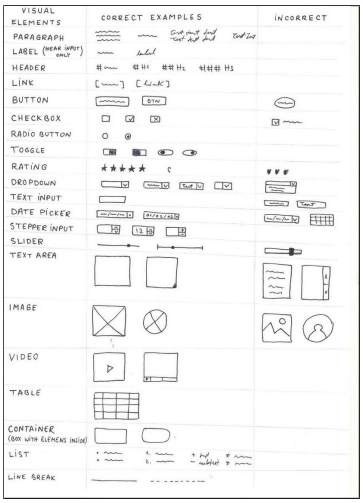
Following the curation process of the 83 UI patterns available in the OutSystems UI framework and having the final goal of this project in mind, it was necessary to establish a hand-drawn representation for each of the 16 relevant elements.

Setting a streamlined, intuitive, and distinct representation for each element was critically important. The high variability of hand-drawn sketches naturally requires a large dataset covering the most diverse styles of hand-drawing. Without restricting the number of admissible representations for each UI element, the complex challenge of training the model to correctly identify each element would only be more arduous.

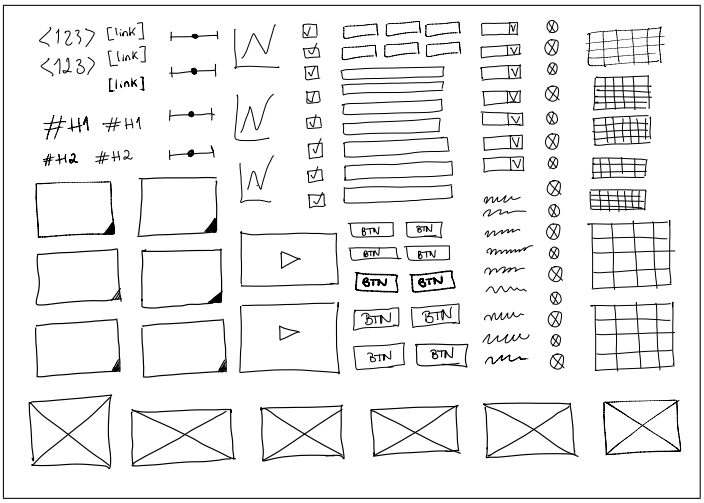
Having a final representations catalogue was critical before launching the laborious task of building a human-generated dataset. However, it was the result of an iterative process that occurred in parallel with the development of two other important stages of the pipeline: the automatic dataset generator tool and the object detection model.

While fine-tuning the hand-drawn representations of UI elements led to important improvements of the dataset generator tool, it has also benefited from a preliminary analysis of the testing results using a simpler implementations of YOLOv2 [30] and a smaller dataset using the standard representations proposed by teleportHQ [3], depicted in Figure 3.3.

Before launching the crowdsourced effort of producing a human-generated dataset, it was critically important to stabilize the hand-drawn representations of all UI elements, so that voluntary collaborators did not have to repeat their sketches multiple times.



(a) List of teleportHQ's standard representations [3].



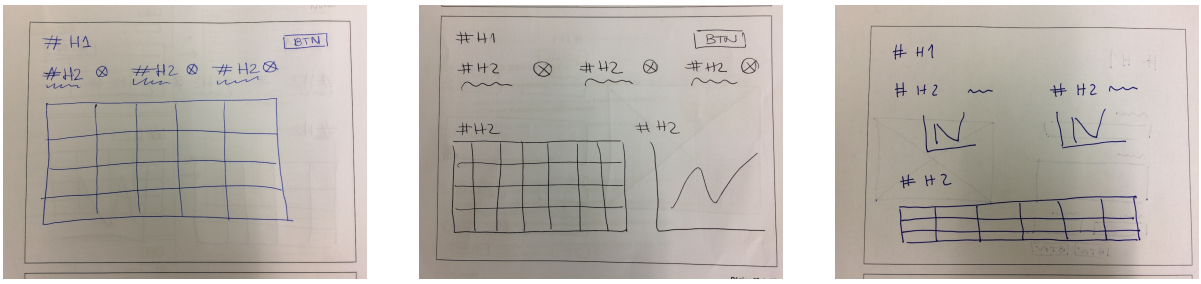
(b) Exemplary paper sheet of mass-produced hand-drawn representations to be used by the dataset generator tool.

Figure 3.3: Dashboards needed for the in-depth analysis of the OutSystems UI Framework.

One of the reasons why fine-tuning these representations led to improvements across multiple pipeline stages, like the computer-generated dataset tool, was because the need to quickly evaluate object detection results for different representations.

The *Table* UI element was one of the most challenging hand-drawn representations, requiring the highest number of iterations before launching the crowdsourced effort to produce the dataset.

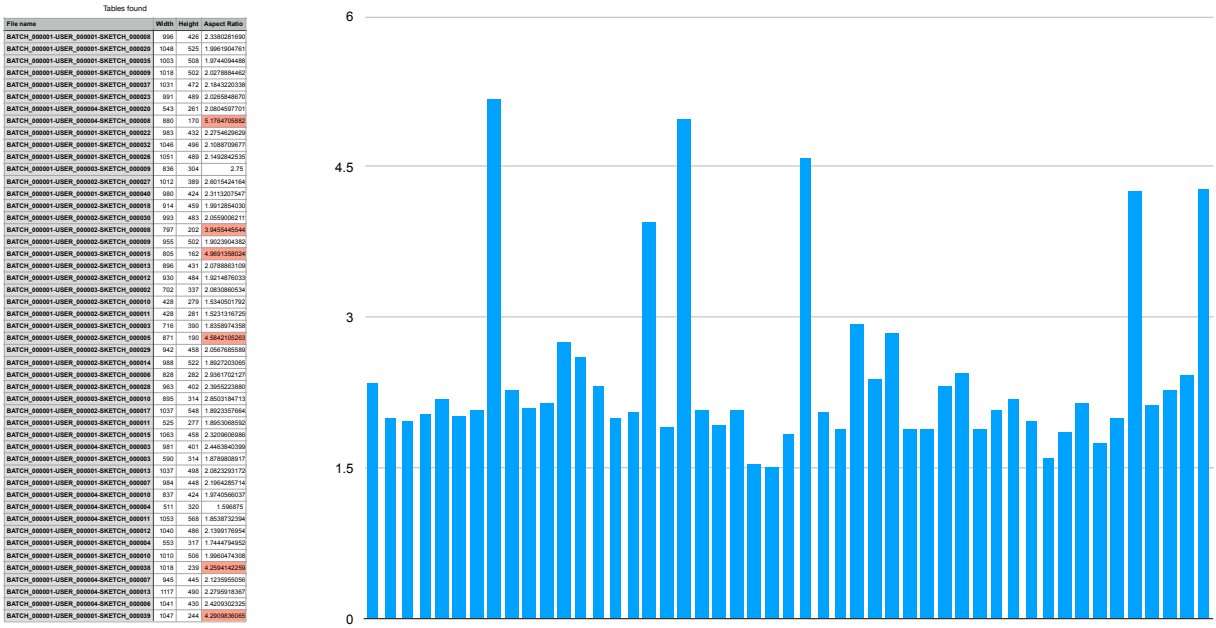
As shown in Figure 3.3, the first proposed representation was imported from teleportHQ's proposed standards list. While a simple 4×4 grid seemed to be the most straightforward way of representing a table, the first batch of users that produced sketches for the human-generated dataset proved how different contexts may lead to significantly different drawing, as shown in Figure 3.4.



(a) (b) (c)

Figure 3.4: Illustration of how different contributors have drawn different tables without noticing: (a) example of a 5×4 grid representation with rectangular-shaped cells, (b) a 7×4 grid representation with square-shaped cells, and (c) a 6×3 grid representation with irregular rectangular-shaped cells.

While the most appealing solution to streamline these inconsistencies seemed to be the inclusion of a warning in the instructions document, alerting contributors to respect the 4×4 grid, it became evident that this representation would still vary depending on each table aspect ratio, as shown in Figure 3.5.



Tables found

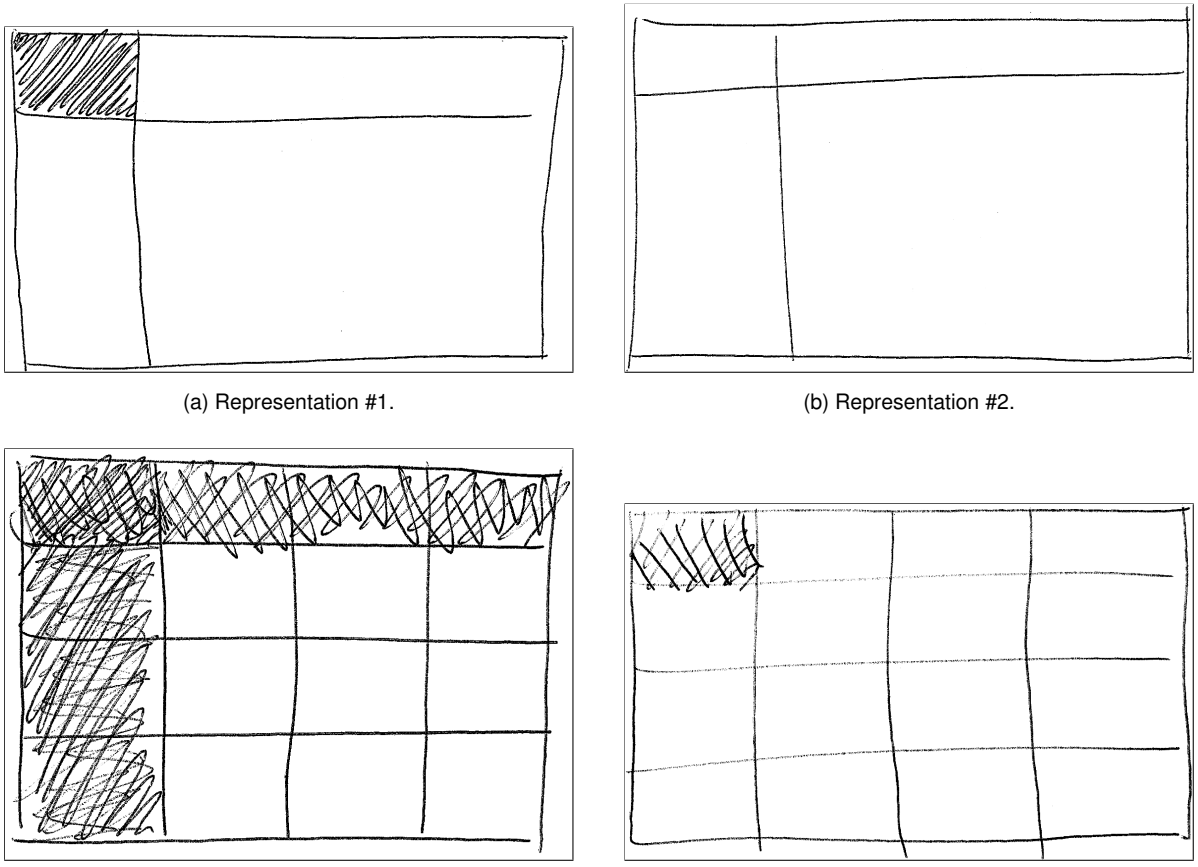
File name	Width	Height	Aspect Ratio
BATCH_000001-USER_000001-SKETCH_000008	996	426	2.338028169
BATCH_000001-USER_000001-SKETCH_000020	1048	525	1.996190476
BATCH_000001-USER_000001-SKETCH_000035	1033	508	1.974409449
BATCH_000001-USER_000001-SKETCH_000009	1018	502	2.027888446
BATCH_000001-USER_000001-SKETCH_000037	1031	472	2.184322034
BATCH_000001-USER_000001-SKETCH_000023	991	489	2.026584867
BATCH_000001-USER_000004-SKETCH_000020	543	261	2.080459770
BATCH_000001-USER_000004-SKETCH_000008	880	170	5.176470588
BATCH_000001-USER_000001-SKETCH_000022	983	432	2.275462963
BATCH_000001-USER_000001-SKETCH_000032	1046	496	2.108870967
BATCH_000001-USER_000001-SKETCH_000026	1051	489	2.149284253
BATCH_000001-USER_000003-SKETCH_000009	836	304	2.75
BATCH_000001-USER_000002-SKETCH_000027	1012	389	2.601542416
BATCH_000001-USER_000001-SKETCH_000040	980	424	2.311320754
BATCH_000001-USER_000002-SKETCH_000018	914	459	1.991285403
BATCH_000001-USER_000002-SKETCH_000030	993	483	2.055900621
BATCH_000001-USER_000002-SKETCH_000008	797	202	3.945544554
BATCH_000001-USER_000002-SKETCH_000009	955	502	1.902390438
BATCH_000001-USER_000003-SKETCH_000015	805	162	4.969135802
BATCH_000001-USER_000002-SKETCH_000013	896	431	2.078886310
BATCH_000001-USER_000002-SKETCH_000012	930	484	1.921487603
BATCH_000001-USER_000003-SKETCH_000002	702	337	2.083086053
BATCH_000001-USER_000002-SKETCH_000010	428	279	1.534050179
BATCH_000001-USER_000002-SKETCH_000011	428	281	1.523131672
BATCH_000001-USER_000003-SKETCH_000003	716	390	1.835897435
BATCH_000001-USER_000002-SKETCH_000005	871	190	4.584210526
BATCH_000001-USER_000002-SKETCH_000029	942	458	2.056768558
BATCH_000001-USER_000002-SKETCH_000014	988	522	1.892720306
BATCH_000001-USER_000003-SKETCH_000006	828	282	2.936170212
BATCH_000001-USER_000002-SKETCH_000028	963	402	2.395522388
BATCH_000001-USER_000003-SKETCH_000010	895	314	2.850318471
BATCH_000001-USER_000002-SKETCH_000017	1037	548	1.892335766
BATCH_000001-USER_000003-SKETCH_000011	525	277	1.895306859
BATCH_000001-USER_000001-SKETCH_000015	1063	458	2.320960698
BATCH_000001-USER_000004-SKETCH_000003	981	401	2.446384039
BATCH_000001-USER_000001-SKETCH_000003	590	314	1.878980891
BATCH_000001-USER_000001-SKETCH_000013	1037	498	2.082329317
BATCH_000001-USER_000001-SKETCH_000007	984	448	2.196428571
BATCH_000001-USER_000004-SKETCH_000010	837	424	1.974056603
BATCH_000001-USER_000004-SKETCH_000004	511	320	1.596875
BATCH_000001-USER_000004-SKETCH_000011	1053	568	1.853873239
BATCH_000001-USER_000001-SKETCH_000012	1040	486	2.139176954
BATCH_000001-USER_000001-SKETCH_000004	553	317	1.744479495
BATCH_000001-USER_000001-SKETCH_000010	1010	506	1.996047430
BATCH_000001-USER_000001-SKETCH_000038	1018	239	4.259414225
BATCH_000001-USER_000004-SKETCH_000007	945	445	2.123595506
BATCH_000001-USER_000004-SKETCH_000013	1117	490	2.279591836
BATCH_000001-USER_000004-SKETCH_000006	1041	430	2.420930232
BATCH_000001-USER_000001-SKETCH_000039	1047	244	4.290983606

Figure 3.5: Analysis of tables' aspect ratios across the sketches produced by the first batch of users.

Considering the available OutSystems UI framework screen templates, it turns out that tables serve different purposes across different user interfaces, ranging from a preponderant role with nearly full-screen sizes in dashboards, to significantly smaller instances with small previews of charts data.

These dimensional differences make users unconsciously draw more or less columns and rows. Therefore, several alternative hand-drawn representations were tested for the *Table* UI element, including a more complex shadowed design, in order to force users to be more careful while representing a table and inducing them to count how many rows and columns needed to be shadowed, as shown in Figure 3.6.

All the alternative representations led to more accurate object detection results and less inconsistencies among users. Nevertheless, as explained earlier, the hand-drawn representations must be simple and straightforward, avoiding any compromises during the informal sketching process to convey ideas. Having this priority in mind, Figure 3.6 b) representation was chosen.



(a) Representation #1.

(b) Representation #2.

(c) Representation #3.

(d) Representation #4.

Figure 3.6: Examples of alternative hand-drawn representations of the *Table* UI element.

In order to avoid repeating the collection of hand-drawn sketches among the first batch of dataset contributors for each new representation, a custom Python script was developed to make this process more seamless. All in all, this auxiliary script finds all drawn tables from the annotations file and overlaps them with an alternative representation sample from a mass-produced collection similar to the paper sheet shown in Figure 3.3.

After multiple iterations, the result was the intended streamlined, intuitive, and distinct catalogue of hand-drawn representations for each of the 16 chosen elements, as shown in the examples of Figure 3.7 and in page 2 of Appendix A.

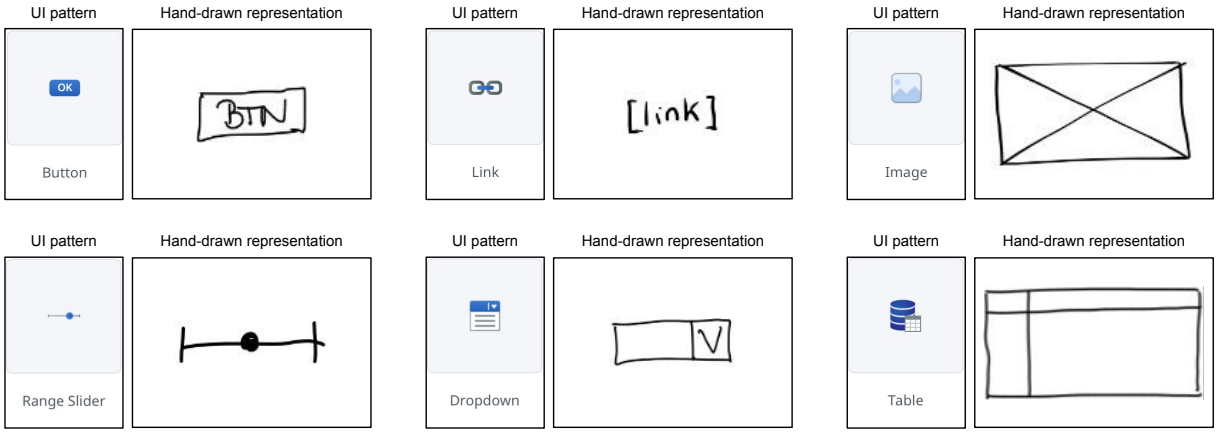


Figure 3.7: Illustrative standard hand-drawn representations of prominent UI elements, which will be used for human-generated elements.

3.1.3 Human-generated Dataset

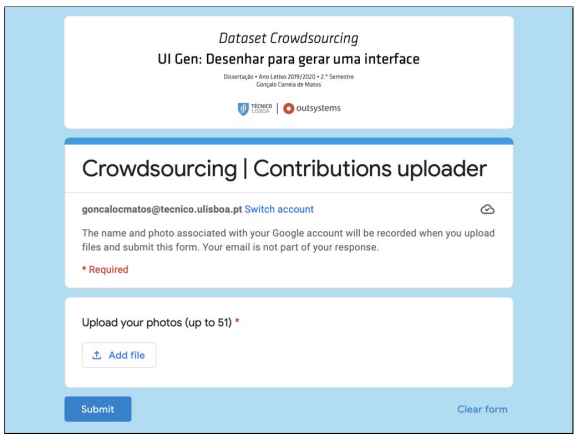
As aforementioned, one of the most difficult challenges posed by neural networks is collecting large amounts of relevant and labeled elements for the dataset. Considering that the final goal of this project is to convert any hand-made sketch into actual code, having a significant number of diverse hand-drawn sketches in the dataset is critically important. This means that the core of the dataset has to be made of real and diverse human-generated records.

While there are no particularly efficient methods to crowdsource the production of a human-generated dataset, we started by designing a straightforward contribution process to promote remote contributions. This was inspired by the previous in-person dataset generation sessions organized during the previous thesis work supported by OutSystems [2].

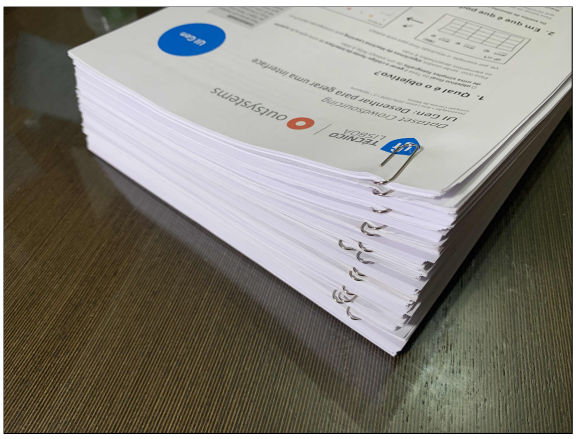
Appendix A contains the instructions document that was sent to the volunteers. The main purpose of this document was to concisely present the thesis work motivation, provide the list of correct representations for the supported UI elements, and illustrate 17 examples of valid user interfaces that could potentially be used as an inspiration for their own sketches. The document is written in European Portuguese, which was the native language of all participating volunteers.

The particular task of gathering a vast human-generated dataset required a considerable effort from 24 volunteers who signed up to draw up to 51 user interfaces, take photos of their sketches, and upload those pictures using the contributions uploader form shown in Figure 3.8.

For this thesis work, we required contributors to return their paper copies in case something went wrong with the photo-taking task that was requested. This cautious request proved to be appropriate, considering that the returned physical copies of hand-drawn sketches were used to fine-tune the pre-processing tool, as will be explained later on.



(a) Online contributions uploader.



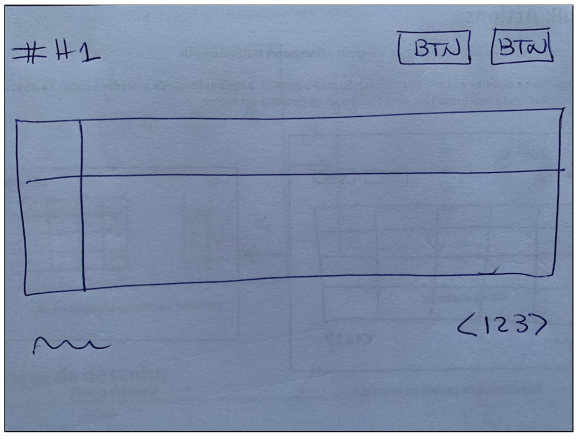
(b) Returned physical copies of all hand-drawn sketches.

Figure 3.8: Crowdsourced photo uploads and returned paper copies.

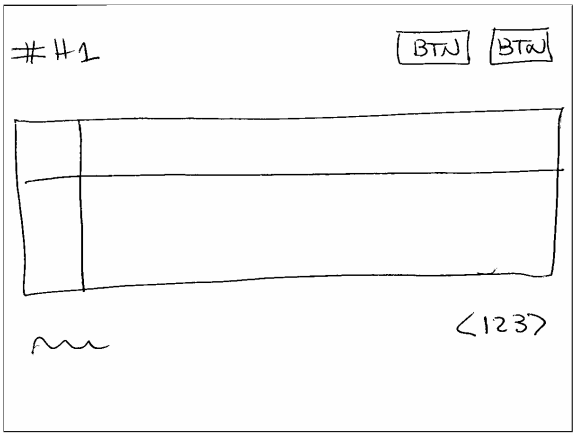
After receiving photos from all volunteers, we summarized all contributions by assigning a unique ID to each volunteer and counting the total of sketches uploaded. This summary was then used to organize our human-generated dataset. All uploaded images were renamed following the same hierarchical rational of batch, user ID, and sketch ID (e.g., `BATCH_000001-USER_000019-SKETCH_000037.png`). Streamlined, structured and clear filenames are crucial for many of the tools implemented throughout the pipeline.

Having the human-generated dataset organized by batch, user, and sketch, we proceeded with a sequence of two pre-processing steps before the labeling task. First, all photos were rescaled to a standard size of 1200 by 900 pixels, which corresponds to the average OutSystems UI screen template resolution.

Secondly, a binarization step was embedded into the pre-processing script using the method by Sauvola [56], in an attempt to remove noise and preserve a pure white background where the hand-drawn lines are black, as depicted in Figure 3.9.



(a) Sixth photo submitted by volunteer #5 using the contributions uploader (file name: `IMG_1405.jpeg`).



(b) Pre-processed photo after resizing and binarization (file name: `BATCH_000002-USER_000005-SKETCH_000006.png`).

Figure 3.9: Resizing and binarization pre-processing results.

Finally, after all contributions were organized and pre-processed, it was possible to proceed with the labeling task using a graphical image annotation tool called LabelImg [57], which is written in Python and uses Qt for its graphical interface, shown in Figure 3.10.

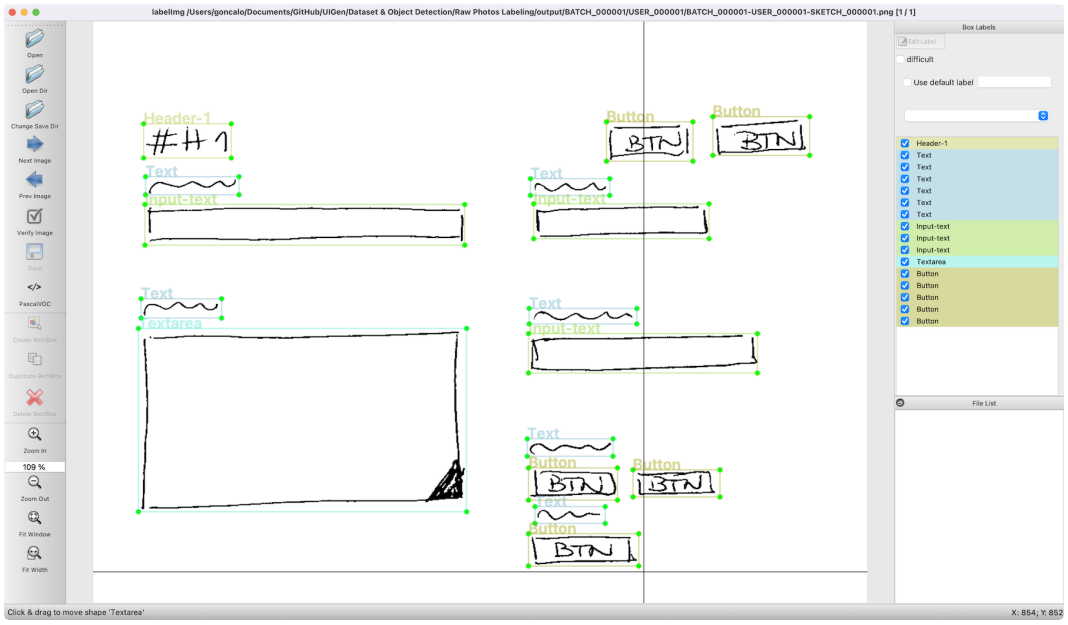


Figure 3.10: Screenshot of LabelImg, a graphical image annotation tool and label object bounding boxes in images [57].

Labeling a dataset using this tool consists of opening each image from the dataset, drawing a bounding box around each object, and selecting the corresponding class. After saving the annotated images, a XML file is created following the Pascal Visual Object Classes (VOC) format.

After the labeling task was completed, a summary of all volunteer contributions was prepared, containing the total amount of produced and labeled sketches per volunteer, adding up to over 1000 sketches.

Table 3.1: Summary of volunteer contributions per user during the crowdsourced process.

Volunteer ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Batch	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Sketches	40	30	16	24	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	24	21	9	28	32

However, the latest YOLOv5 version, used in this thesis, only supports the state-of-the-art YOLO annotations standard, which provides an individual text file per image with the same name corresponding to the intended image. The specifications of the YOLO format are as follows:

- each line in the annotations text file corresponds to a single object.
- each line follows the same attributes pattern: `class x_center y_center width height`.
- class names are not explicitly in the annotations, but rather an integer corresponding ID.
- `x_center`, `y_center`, `width`, `height` must be normalized values (ranging from 0 to 1).

BATCH_000002-USER_000005-SKETCH_000001.xml

```
<annotation>
        <folder>5</folder>
        <filename>BATCH_000002-USER_000005-SKETCH_000001.png</filename>
        <path>C:\Users\gom\Documents\GitHub\UIGen\Dataset\Raw Photos
Labeling\Preparation\5\BATCH_000002-USER_000005-SKETCH_000001.png</path>
        <source>
                <database>Unknown</database>
        </source>
        <size>
                <width>1200</width>
                <height>900</height>
                <depth>1</depth>
        </size>
        <segmented>0</segmented>
        <object>
                <name>Header-1</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>16</xmin>
                        <ymin>62</ymin>
                        <xmax>180</xmax>
                        <ymax>130</ymax>
                </bndbox>
        </object>
        <object>
                <name>Image</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
```

(a) Pascal VOC annotations XML file.

BATCH_000002-USER_000005-SKETCH_000001.txt

```
8 0.081667 0.106667 0.136667 0.075556
0 0.092083 0.283333 0.179167 0.186667
11 0.282083 0.222222 0.135833 0.046667
11 0.287083 0.301667 0.145833 0.038889
11 0.273750 0.435556 0.147500 0.048889
11 0.280417 0.494444 0.145833 0.042222
11 0.275833 0.631667 0.138333 0.050000
11 0.278750 0.696111 0.155833 0.041111
11 0.271250 0.823889 0.147500 0.038889
11 0.280000 0.883333 0.135000 0.040000
11 0.508750 0.420000 0.144167 0.042222
11 0.508333 0.487222 0.150000 0.043333
11 0.729583 0.216111 0.140833 0.052222
11 0.718333 0.282778 0.120000 0.043333
0 0.538333 0.268889 0.211667 0.175556
7 0.912917 0.250000 0.145833 0.088889
0 0.095417 0.485000 0.175833 0.190000
0 0.099583 0.672778 0.182500 0.174444
0 0.100833 0.863333 0.185000 0.173333
```

(b) YOLO annotations TXT file.

Figure 3.11: Side by side comparison of Pascal VOC and YOLO formats for the same labeling annotations.

In order to convert LabelImg Pascal VOC annotations into the YOLO format, we used a Python script that requires a `class.txt` file describing all classes and converts XML annotations exported by LabelImg (on which all bounding boxes are given by `<xmin>`, `<ymin>`, `<xmax>`, and `<ymax>`) into TXT files following the YOLO format (given by `x_center`, `y_center`, `width`, and `height`) using the following formulas:

$$x_{center} = \frac{((x_{max} + x_{min})/2)}{w} \quad \text{and} \quad y_{center} = \frac{((y_{max} + y_{min})/2)}{h}, \text{ and} \tag{3.1}$$

$$width = \frac{(x_{max} - x_{min})}{w} \quad \text{and} \quad height = \frac{y_{max} - y_{min}}{h}. \tag{3.2}$$

The YOLO format only requires a single easily parsable TXT file (with the same file name and in the same directory) defining all objects in an image, each object corresponding to a single line. After running this Python script, no images in the dataset are not modified, but rather a TXT file is generated with the corresponding normalized coordinates, as shown in Figures 3.11 and 3.12.

Figure 3.12: Visual interpretation of the YOLO format normalized coordinates [58].

After converting all annotations into the YOLO format, a review of the labeling process was conducted by inspecting each annotated sample using a tooled called Roboflow [59]. This tool provided an important insight for this thesis work using its *Dataset Health Check* feature, which shows how many elements of each class there are and provides an intuitive visualization of class balance or imbalance.

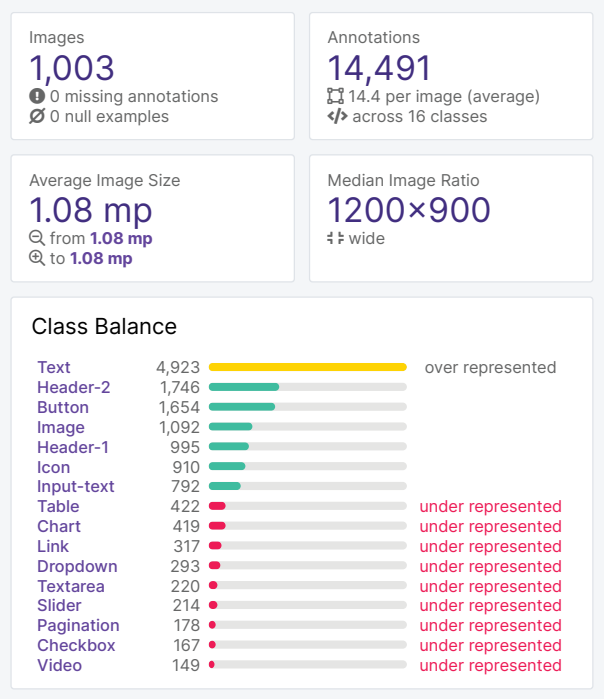


Figure 3.13: Roboflow *Dataset Health Check* results [59].

By analyzing the chart shown in Figure 3.13, several ideas emerged to balance the number of elements of each class in the dataset. Nevertheless, the evident imbalance was not unexpected, considering that some UI elements are knowingly predominant across all sketches. As an example, the *Text* UI element is represented multiple times in all screen templates, while the *Video* UI element is only present in the *Product Feature* screen template shown in Appendix A.

This chart was also relevant for the next object detection stage in the pipeline. Splitting the dataset without carefully analyzing the data we had could compromise the unbiased assessment of the training process performance.

While the most traditional division ratio tends to be around 70%, 20%, and 10% for train, validation, and test, respectively, our 1000-sample dataset is not large enough. As explained in Section 2.3.5, k-fold cross-validation is one possible approach when the dataset is not abundant.

For this human-generated dataset, we used 5-fold cross-validation to repeat the training and validation processes on different subsets of the complete dataset, thus avoiding biased evaluations of the object detection model performance on unseen data.

As depicted on Table 3.2, this method consists of splitting the dataset into 5 different groups and perform individual training processes using three subsets and the remaining two subsets are used for validating and testing the model's performance.

Table 3.2: Dataset partitions holdout process for the 5-fold approach.

		Dataset				
Iterations	**Fold 1**	Test	Val	Train	Train	Train
	Fold 2	Train	Test	Val	Train	Train
	Fold 3	Train	Train	Test	Val	Train
	Fold 4	Train	Train	Train	Test	Val
	Fold 5	Val	Train	Train	Train	Test

For the particular circumstances of this thesis work, instead of randomizing the samples that go into the five groups iteratively assigned to train, validation, and test subsets, we decided to organize the dataset into five pre-set categories, all with an equivalent number of sketches. This was mainly to avoid an overfitting scenario where the high variability of handwriting could contaminate the train process for the actual graphical representation of certain elements that include characters (e.g., *Button*, *Header-1*, etc.), as will be explained on Chapter 4.

All five style reflect the calligraphy letterform of all volunteers, namely the ones with a rounded cursive style, a sharp cursive style, a rounded script style, a sharp script style, and a hybrid style (i.e., the volunteer mixes cursive and script styles). The final arrangement of sketches is shown in Table 3.3.

Table 3.3: Distribution of volunteer contributions per calligraphy style.

Calligraphy style	User IDs
Category A	1, 5, 16, 17, 18, 23, 24
Category B	2, 7, 13, 15
Category C	9, 11, 12, 19
Category D	3, 4, 6, 8, 14
Category E	10, 20, 21, 22

Considering that we identified five major calligraphy styles and our cross-validation method consists of five folds, the distribution of sketches for the train, test, and validation subsets was intuitive, as shown in Table 3.4.

Table 3.4: Distribution of calligraphy categories per fold.

Fold #	Train subset	Validation subset	Test subset
Fold 1	C, D, E	B	A
Fold 2	A, D, E	C	B
Fold 3	A, B, E	D	C
Fold 4	A, B, C	E	D
Fold 5	B, C, D	A	E

3.1.4 Computer-generated Dataset

While producing a human-generated dataset is key to achieve the best object detection results, creating a larger and realistic dataset in a timely manner requires an automatic dataset generation tool. After the laborious task of collecting over 1000 unique hand-drawn sketches, and manually labeling over 14,000 elements one by one, a dataset generator was developed following the sketchification approach described in Section 2.16, with slight contextual adjustments.

The general idea of this approach, shown in Figure 3.14, is to take advantage of the cumbersome labeling task, by replacing the elements in the UI with their correct hand-drawn representation.

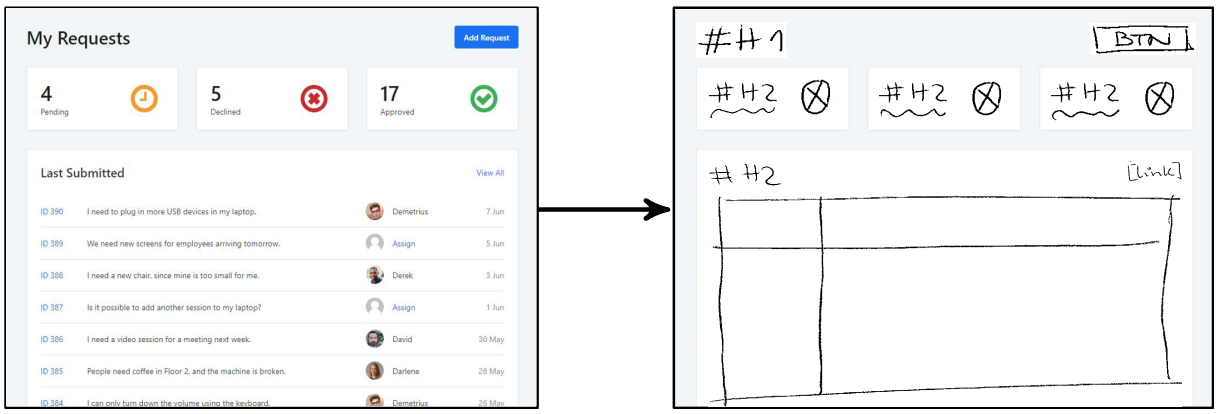


Figure 3.14: Illustration of the proposed approach for computer-generated sketches, where the elements of the *Admin Dashboard* screen template are replaced with their respective hand-drawn representation.

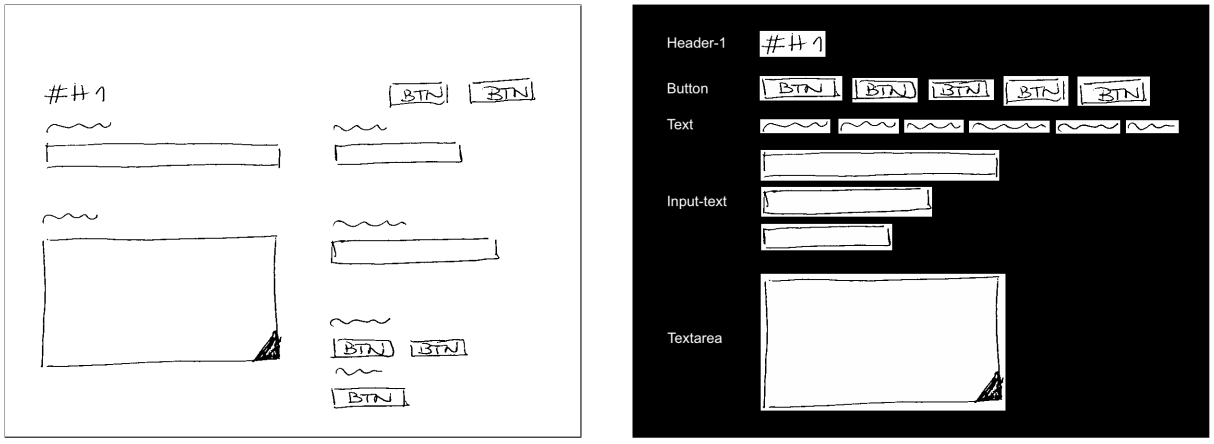
Eventually, the sketchification approach can also be combined with data augmentation techniques and morphological operations. The higher the number of labeled hand-drawn UI elements from the human-generated dataset, the higher the number of possible combinations in new computer-generated screen templates. By randomly combining different sources of UI elements in each screen template, a more distinct and realistic dataset can be generated.

After having a significant number of representations for each UI element, which were hand-drawn by different people, it is possible to massively generate realistic dataset samples. Table 3.5 shows the total of hand-drawn samples available for each of the 16 classes supported by our object detector.

Table 3.5: Total of hand-drawn representations cropped from the human-generated dataset per class.

Classes	Text	Header-2	Button	Image	Header-1	Icon	Input-text	Table	Chart	Link	Dropdown	Textarea	Slider	Pagination	Checkbox	Video
Representations	4,923	1,746	1,654	1,092	995	910	792	422	419	317	293	220	214	178	167	149

Therefore, our implementation of this dataset generator starts with an element cropper that extracts each labeled UI element from the human-generated dataset and organizes them into folders, each corresponding to a single class. Figure 3.17 shows the result of applying this Python script, extracting 1 samples of a *Header-1*, 5 samples of *Button*, 6 samples of *Text*, 3 samples of *Input-text*, and 1 sample of *Textarea*.



(a) Pre-processed original human-generated sketch.

(b) Extracted human hand-drawn UI elements.

Figure 3.15: Extracted hand-drawn UI elements from a pre-processed human-generated sketch.

The core task of our dataset generator tool is to produce a ready-to-train dataset, with realistic computer-generated sketches and their respective annotations in the YOLO format, with no need to perform extra steps after the generation.

Our tool replaces all UI elements in real user interfaces with an image of their respective hand-drawn representation, which was previously cropped out of the labeled human-generated dataset. This replacement process must assure that the real UI element is replaced by hand-drawn representation with the same size and in the same location.

In order to achieve this, we developed a dataset generator written in Python and JavaScript that uses Selenium [60] to automate the dataset generation by modifying the DOM structure of OutSystems UI screen templates, which are available as Web applications written in HTML. Nevertheless, not all UI elements are identifiable in the DOM structure by the same type of arguments, so we started by analyzing OutSystems UI [55] and gathered a small list of HTML and CSS attributes that allow our dataset to find all supported elements on the screen, as summarized in Table 3.6.

Depending on the attributes of each UI element, our dataset generator uses three JavaScript functions, namely `getElementsByClassName()`, `getElementsByTagName()`, and `getElementById()`, to retrieve their DOM objects. A fourth JavaScript function was used, `getBoundingClientRect()`, to log the original UI element size and position, shown in Figure 3.16, and use them for the image replacement.

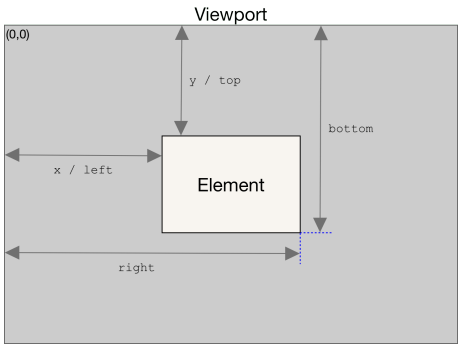


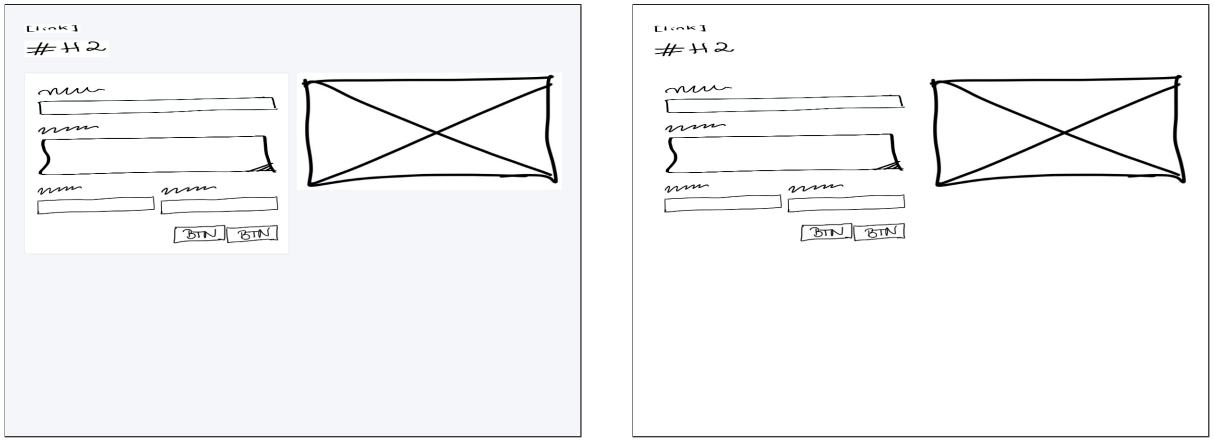
Figure 3.16: Attributes included in the DOMRect object returned by `getBoundingClientRect()` [61].

Table 3.6: Replaceable elements HTML and CSS attributes.

UI element	CSS Class	HTML Tag	HTML ID	HTML Type
Image	×	`<img>`	×	×
Video	×	`<video>`	×	×
Icon	`radius-round-i`	`<i>`	×	×
Table	`table`	×	×	×
Checkbox	`checkbox`	`<input>`	×	`checkbox`
Input-text	`search-input`	`<input>`	`PageTitle`	`text`
Textarea	×	`<textarea>`	×	×
Button	`btn`	`<button>`	`Button`	×
Header-1	`header-1`	`<h1>`	×	×
Header-2	`header-2`	`<h1>`	×	×
Dropdown	`dropdown-container`	×	×	×
Text	×	`<label>`	×	×
Link	×	`<a>`	×	×
Slider	`range-slider`	×	×	×
Pagination	`pagination-container`	×	×	×
Chart	`os-chart`	×	×	×

During the development process of our dataset generator tool, the pre-processing script developed during the human-generated dataset stage had to be embedded into the generator itself, but also rewritten to accommodate the particularities of a computer-generated dataset.

Instead of pre-processing images after being generated, we decided to implement a fixed set of CSS modifications that remove any container or background color that could compromise the final result, as shown in Figure 3.17. Hence, after all elements have been replaced by their respective hand-drawn representation, we proceed to implement slight modifications to the page CSS before screenshotting it for the dataset.



(a) UI after all elements have been replaced.

(b) UI after CSS is modified to remove colors.

Figure 3.17: Pre-processed computer-generated sketch after CSS color modifications.

3.2 Object Detection Model

All in all, our object detection model serves a simple role in the implemented pipeline: it receives a hand-drawn sketch as input and extracts all the features of the image, namely the class and position of the objects. Therefore, the object detection stage is the central stage of the implemented solution, considering that all further pipeline stages rely on its output.

Chapter 2 exhaustively addresses how the current state-of-the-art algorithms, methodologies, and tools could be used in this thesis work, also covering core background concepts and examples of related work. While region proposals are one of the most common approaches to localize objects and has very good performance, it requires multiple stages, such as generating region proposals, extracting features with a CNN, classifying, and generating bounding boxes, making it computationally expensive.

The You Only Look Once (YOLO) algorithm follows a one-step framework approach, which usually allows significantly more efficient computing times and maintains high accuracy levels.

This approach consists of feeding a given hand-drawn sketch into the YOLO network, which then outputs a set of bounding boxes coordinates associated with their respective class and confidence level for each detected object. From its first iteration, YOLOv1, which was based on Darknet and built into C, to its fifth and fastest iteration yet, YOLOv5, YOLO has improved its architecture, currently providing the highest detection accuracy and the fastest inference speed of all iterations, making it the ideal choice for our pipeline. In addition, YOLOv5 is written in Python instead of C, making the installation and integration processes easier from the official Ultralytics' GitHub repository [62].

However, this is only a baseline architecture that can be configured by researchers and customized to achieve the best results depending on their problems. By cloning and editing the YOLOv5 architecture configuration, researchers can add layers, remove blocks, customize image processing operations, optimize activation functions, and more.

Training and testing YOLOv5 for a breakthrough project requires a custom and labeled dataset, splitted to be used for train, test, and validation purposes. As covered in Section 3.1, we produced two datasets: a human-generated dataset with 1003 unique hand-drawn sketches, containing 14,491 UI elements labeled one by one, and a computer-generated dataset with 2,000 hand-drawn sketches, automatically labeled by our dataset generator tool.

All images are in `.png` format and their respective annotation files are plain `.txt` files. These files specify the location and size of the UI elements with labels in the corresponding image. In order to be compatible with YOLOv5, all data files were moved to their respective folder, depending on their train, test, or validation purpose, and then organized into two sub-folders according to the file type (images and labels), as shown in file tree structure depicted in Figure 3.18.

Our YOLOv5 implemented pipeline includes all necessary steps to build, train, and test our model. Essentially, it consists of five main steps implemented in Google Colaboratory that will be reviewed in this section: 1) install YOLO and import our dataset from the previous pipeline stage, 2) configure our custom model, 3) connecting Tensorboard and WandB to plot metrics, 4) train our model, and 5) validate and test our model's accuracy and perform detections on unseen samples for further qualitative analyses.

```
dataset/
├── train/
│   ├── images/
│   └── labels/
├── test/
│   ├── images/
│   └── labels/
├── valid/
│   ├── images/
│   └── labels/
└── data.yaml
```

Figure 3.18: Dataset file tree structure preparation for YOLOv5.

Google Colaboratory is an online Integrated Development Environment (IDE) that supports academic research and learning on AI. Colab provides a code environment similar to Jupyter Notebook, and supports Graphics Processing Unit (GPU) acceleration. It also supports the most important libraries for deep learning research work, such as PyTorch, TensorFlow, Keras, and OpenCV.

Because machine learning tasks and deep learning algorithms require good hardware processing power (usually based on GPU), most desktop computers are not ideal to train a model. However, the Colab's GPU acceleration (Tesla T4 architecture) is an undeniable offer, considering that these are some of the highest performing GPUs.

3.2.1 Installing YOLO and Importing a Dataset

The first step in our object detection Colab consists of installing YOLOv5 and several Python dependencies and libraries for matrix operations, plotting, and file handling, which are listed in a `requirements.txt` file in the YOLOv5 directory, as shown in Figure 3.19.

Figure 3.19: Cloning Ultralytics' YOLOv5 repository and importing our dataset.

Considering that Colab is an isolated environment that runs in the cloud, the dataset must be uploaded before training the model. After uploading the dataset to Google Drive, Colab needs to import it. This first step includes all the necessary code to mount Google Drive and unzip the dataset folders.

3.2.2 Configuring a Custom Object Detection Model

After downloading the dataset to the isolated Colaboratory environment, custom changes have to be made to YOLOv5, namely, adjusting the number of classes being used, considering that we are training a custom object detector. Figure 3.20 shows how some of these adjustments were made.

Figure 3.20: Customizing YOLOv5 model configuration properties.

We use the YOLOv5 backbone, which is summarized in Table 3.7 and mainly includes Cross Stage Partial Network (CSPNet) [63] and Focus modules.

Table 3.7: YOLOv5 backbone architecture summary.

Number	Module	Filter	Size
1x	Focus	64	3x3
1x	Convolutional	128	3x3/2
3x	CSP	128	1x1
1x	Convolutional	256	3x3/2
9x	CSP	256	1x1
1x	Convolutional	512	3x3/2
9x	CSP	512	1x1
1x	Convolutional	1024	3x3/2
1x	SPP	1024	1x1
3x	CSP	1024	1x1

The CSPNet module is applied to first split the feature map of the beginning layer into two branches and then unite them through a hierarchical structure, reducing the calculation amount while ensuring accuracy. The Focus module is meant to perform a slicing operation of the feature map. The remaining 3x3/2 convolution layers are mainly for downsampling.

YOLOv5 also comes with standard hyperparameters for training with the COCO dataset that do not apply to our custom object detector.

Therefore, all standard augmentation operations, such as image hue, saturation, and value augmentation, as well as image rotation, translation, scale, shear, perspective, flip up-down, flip left-right, mosaic effect, and mix-up, were disabled, as summarized in Table 3.8. We kept other hyperparameters values from YOLOv5 repository, such as initial learning rate of 0.01, box loss weight of 0.05 and class loss gain of 0.5. We use Stochastic Gradient Descent (SGD) as optimization method.

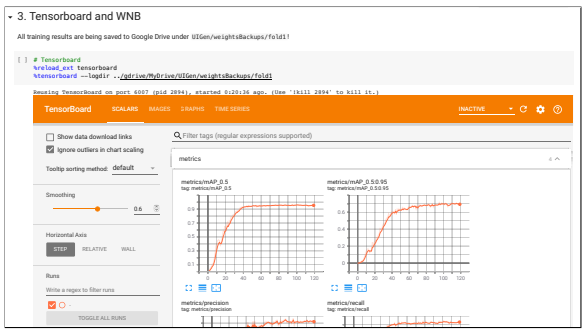
Table 3.8: Summary of our YOLOv5 customized hyperparameters.

Hyperparameters	Values	Descriptions
lr0	0.01	Initial learning rate (SGD= 1^{-2}, Adam = 1^{-3})
lrf	0.2	Final OneCycleLR learning rate (lr0 * lrf)
momentum	0.937	SGD momentum/Adam beta1
weight_decay	0.0005	Optimizer weight decay 5^{-4}
warmup_epochs	3.0	Warmup epoch
warmup_momentum	0.8	Warmup initial momentum
warmup_bias_lr	0.1	Warmup initial bias lr
box	0.05	Box loss gain
cls	0.5	Cls loss gain
cls_pw	1.0	Cls BCELoss positive weight
obj	1.0	Obj loss gain (scale with pixels)
obj_pw	1.0	Obj BCELoss positive weight
iou_t	0.20	IoU training threshold
anchor_t	4.0	Anchor-multiple threshold
anchors	0	Anchors per output grid (0 to ignore)
fl_gamma	0.0	Focal loss gamma (efficientDet default gamma = 1.5)
hsv_h	0.0	Image HSV-Hue augmentation (fraction)
hsv_s	0.0	Image HSV-Saturation augmentation (fraction)
hsv_v	0.0	Image HSV-Value augmentation (fraction)
degrees	5.0	Image rotation (+/- deg)
translate	0.05	Image translation (+/- fraction)
scale	0.02	Image scale (+/- gain)
shear	0.0	Image shear (+/- deg)
perspective	0.00001	Image perspective (+/- fraction), range 0-0.001
flipud	0.0	Image flip up-down (probability)
fliplr	0.0	Image flip left-right (probability)
mosaic	0.5	Image mosaic (probability)
mixup	0.0	Image mixup (probability)

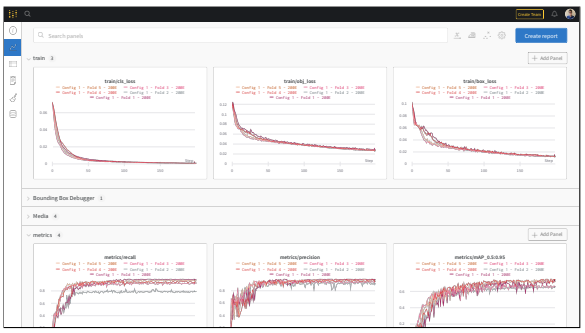
3.2.3 Connecting Tensorboard and WandB

Fine-tuning our model and analyzing our results required using two different visualization tools that helped us control train, validation, and test on the go. For this end, Tensorboard and WandB were used.

While Tensorboard provides great insights right into Google Colab, WandB allowed us to analyze and compare different sessions in a more versatile way, thus benefiting the performance metrics analysis in Chapter 4. Figure 3.21 shows how both tools were used by our model.



(a) Tensorboard built into Google Colab.



(b) WandB dashboard for five runs.

Figure 3.21: Comparison between Tensorboard and WandB for visualizing performance metrics.

3.2.4 Training a Custom Object Detection Model

After customizing YOLOv5 and connecting two visualization tools, the train implementation was quite simple, requiring a single command line with several arguments.

Figure 3.22: YOLOv5 train command line.

All in all, the model will be trained by compiling and running the train.py file according with the following configurable arguments:

- Image size: 1200 (width).
- Rectangular images: True.
- Batch size: 16.
- Epochs: the number of training iterations.
- `--data`: dataset description path `../data.yaml`.
- `--cfg`: configuration of our model described in a YAML model configuration file.
- `--name`: model name to be displayed and eventually saved.

3.2.5 Validating and Testing a Custom Object Detection Model

After finishing the training process, trained weights are saved and can be used to validate the model accuracy for new and unseen samples of our dataset, namely the validation and test sets. Using the commands shown in Figure 3.23, the `val.py` script will be compiled to export three key performance metrics: Precision (P), Recall (R), and two Mean Average Precision (mAP) values over different IoU thresholds (up to 0.5 and from 0.5 to 0.95).

Figure 3.23: Validating YOLOv5 using the validation and test sets.

After the validation process and quantitative metrics are saved, it is important to perform a qualitative evaluation by printing the predicted bounding boxes that cover the detected UI elements. As shown in Figure 3.24, this script draws all bounding boxes onto each image from the validation and test sets. Then, we display the results in the Colab user interface using IPython display. All the images are saved in the same folder containing the results from the training phase.

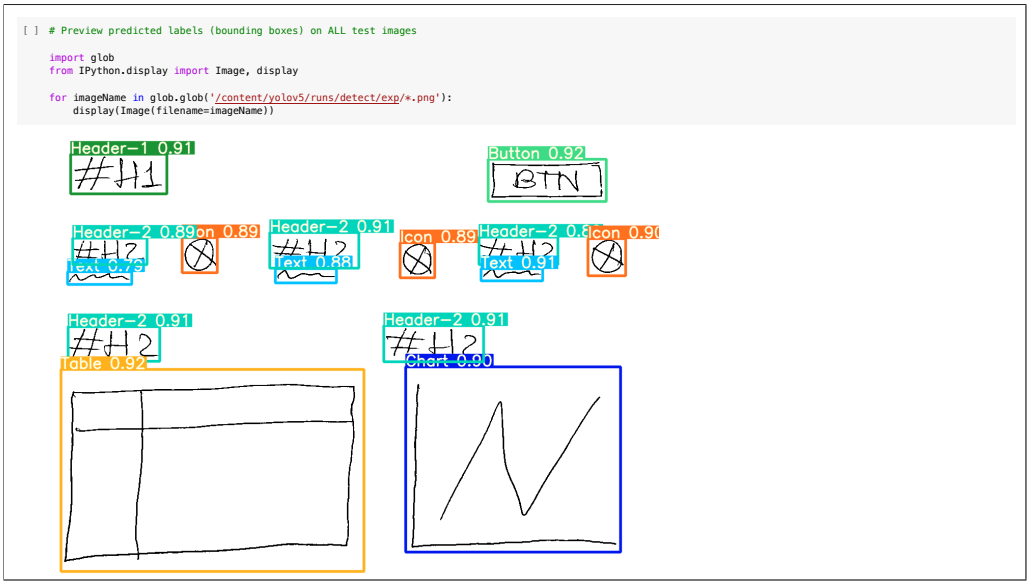


Figure 3.24: Detecting UI elements and printing the predicted bounding boxes.

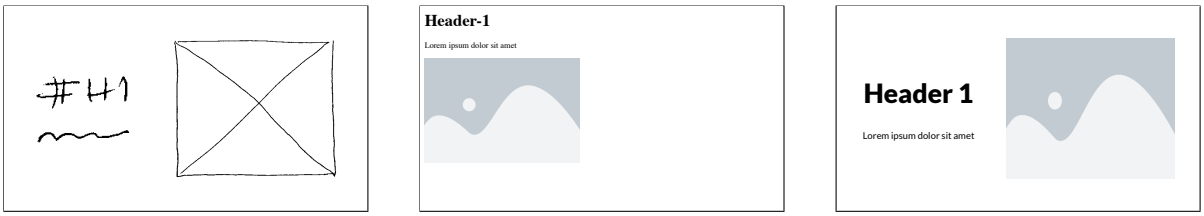
3.3 Spatial Grouping Algorithm

The ultimate goal of the implemented pipeline is to generate the code of a sketched interface that respects the principles of the OutSystems UI framework [64], a low-code framework for web and mobile applications.

A flexible and effective approach is to leverage the established web standards for creating user interfaces. With HTML and CSS, it is possible to build any layout for an app, making it responsive and user-friendly. User interfaces created with HTML and CSS look usually sharper than their counterparts thanks to the specialized rasterization engines of modern web browsers. This approach is especially important not only because our model supports media UI elements such as images, videos, and charts, but also because styling the generated app according to OutSystems UI framework requires us to match the established design of each UI element.

Once the model outputs the detection results containing classes and positions of all elements in a sketch, generating the corresponding UI requires a code generation step that transforms the drawing primitives into a purposeful and spatially organized mock-up. A simple approach would be to straightly generate each UI element as a floating HTML element. However, this approach would compromise the appearance and overall functionality of the generated web app.

The nature of HTML and CSS formatting entails several visual deformations due to the lack of a hierarchical structure in the object detection model output. In order to display the UI elements correctly, for instance a *Header-1* and an *Image* side by side, they first need to be embedded into a container, so that CSS properties can be changed to position elements accordingly. Figure 3.25 shows how the same correct drawing primitives outputted by the object detection model can lead to different results if HTML and CSS are not used properly.



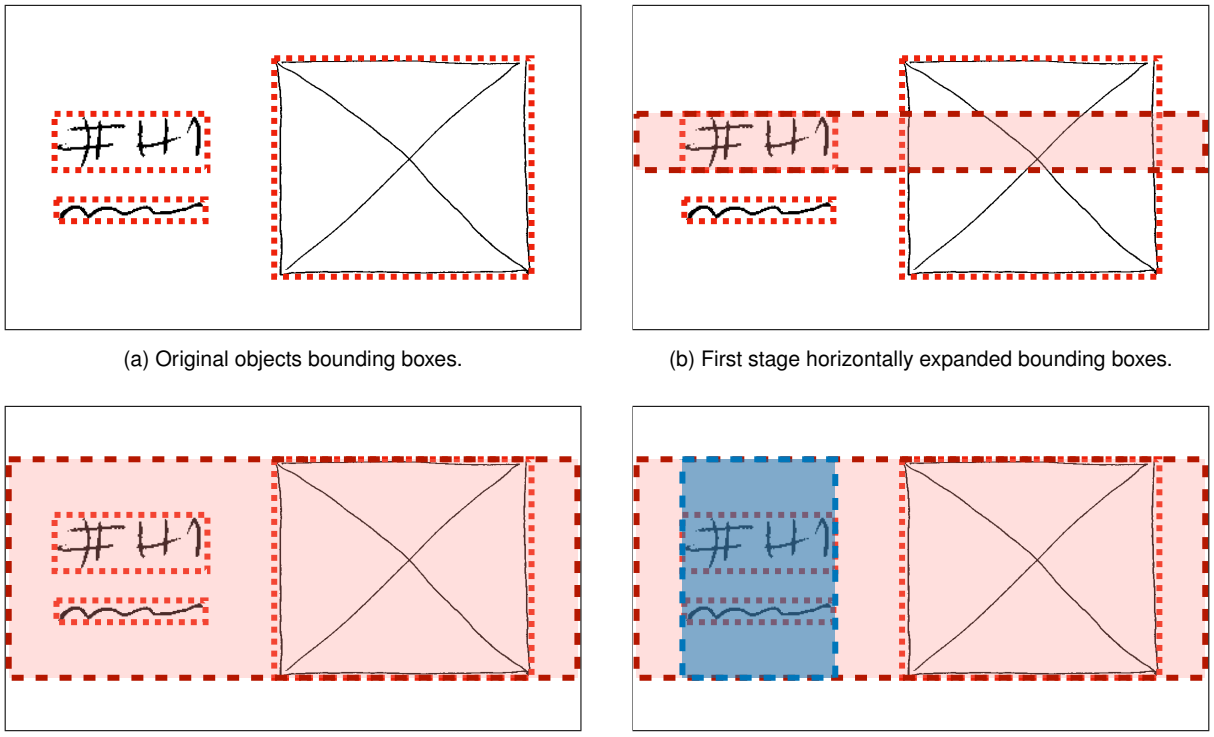
(a) Original sketch. (b) Native HTML-exclusive generation. (c) HTML and CSS-based generation.

Figure 3.25: Illustration of how native HTML elements without CSS can compromise the generation of a UI from the correct object detection results.

The four main steps of our spatial grouping algorithm are illustrated in Figure 3.26. This algorithm starts by sorting the sketched UI elements from left to right, top to bottom, and then proceeds with two main phases.

The first phase consists of testing horizontal intersections with edge-to-edge bounding boxes. If no intersection is possible the edge-to-edge group is closed, we conclude that there are no elements side by side and proceed to check with all other UI elements. In case an intersection is found, then we create a group and try to intersect it as a whole with other horizontally aligned elements, if any.

During the second phase of this algorithm, we proceed to test vertical intersections within major horizontal groups, so that vertically aligned elements can coexist properly. This is evaluated by expanding each element's bounding box from top to bottom and intersecting it with all other elements in the horizontal group. If no intersection is found, then we close the group, save it and proceed to find groups with the next UI elements. Otherwise, we adjust the CSS `flex` property to align the elements within the identified group.



(a) Original objects bounding boxes.

(b) First stage horizontally expanded bounding boxes.

(c) Final group of all horizontally aligned objects.

(d) Final stage vertically expanded bounding box.

Figure 3.26: Illustration of the spatial grouping algorithm: (a) Evaluating the original objects' bounding boxes. (b) Horizontally expanding and intersecting the first element's bounding box edge to edge. (c) Evaluating the final group of all horizontally aligned elements. (d) Finding vertically aligned elements within the horizontal group by expanding and intersecting objects' bounding boxes top to bottom.

Considering that previewing and hard coding all possible combinations of elements positions would be infeasible, we followed an agnostic approach and implemented a versatile spatial grouping algorithm that focuses on passing its hierarchical inference to the code generation stage.

Therefore, this stage outputs a domain-specific language (DSL) containing an agnostic and hierarchical structure with all identified elements that will be used by the code generator. Despite the increased complexity of this approach, it does have some significant advantages, such as the versatility of a DSL and the ability to correct potential errors in the trace set, modelling similarity between drawings, but also being able to extrapolate figures based on the source code.

In fact, using a DSL to output the hierarchical structure allows us to plug in all pipeline stages implemented so far into any code generator or other tools, no matter the goal or programming language it supports. Parsing the arguments and the structure of the DSL is enough to make sure that all previous pipeline stages outputs are preserved and ready to be used.

3.4 Code Generation

The last stage of this thesis is a code generator, which will allow us to render the sketched user interface after going through all pipeline stages. As covered in the previous section, once our object detection model outputs the drawing primitives and a hierarchy is inferred by our spatial grouping algorithm, a domain-specific language (DSL) will be generated. Listing 3.1 shows an input example for a simple UI that contains a single *Header-1* element.

```
export default {
  children: [
    {
      id: 0,
      top: 0,
      left: 0,
      width: 168.0,
      height: 40.0,
      type: "Header-1",
      value: "Header-1"
    }
  ]
  type: "root",
  parsingDirection: "vertical",
  top: 0,
  left: 0,
  width: 1200,
  height: 900,
  id: "0",
  passes: 0
}
```

Listing 3.1: Input example of a single major container of 1200×900, containing a single Header-1 element, which was exported by our spatial grouping algorithm.

Our code generator, written in React and Typescript, parses the DSL and generates the same structure following teleportHQ's User Interface Definition Language (UIDL) [65].

This UIDL is also a universal format that can describe all the possible scenarios for a given user interface, thus allowing us to generate the same user interface with various tools and frameworks, transition technologies without effort, and provide programmatic manipulation. All in all, it is a human-readable JSON document, which is supported natively by most programming languages.

The first building block of the UIDL structure is called a `UIDLNode`, which serves as a root for further nodes. Depending on each element's purpose, the node may be `static`, `dynamic`, `element`, `conditional`, `repeat`, `slot`, and `nested-style`. Table 3.9 shows the attributes supported by each type of node.

Table 3.9: Summary of UIDL keys supported per node type [65].

	Root Node	Children	Attribute	Style	Conditional References	Repeat
`static` value	×	×	×	×		×
`dynamic` reference	×	×	×	×	×	×
`element` node	×	×				
`conditional` node	×	×				
`repeat` node	×	×				
`slot` node		×				

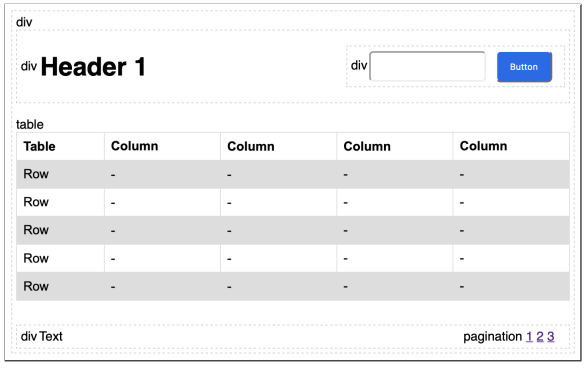
As the UIDL is being traversed, the nodes are interpreted and translated into lines of code, so a `static` node becomes plain text (e.g., the *Text* UI element) and `element` nodes become HTML tags. The content of each node in the UIDL represents all the information that the node holds. Listing 3.2 shows a code snippet of the UIDL generated from the DSL on Listing 3.1.

```
>style: Object (...)
 fontFamily: "'Cabin', sans-serif"
 flexDirection: "row"
 justifyContent: "space-between"
 alignItems: "center"
 padding: "5px"

>children: Array(1)
    >children: Array(1)
        >0: Object
         type: "element"
        >content: Object
         elementType: "h1"
        >style: Object (...)
         color: "black"
         width: "168px"
```

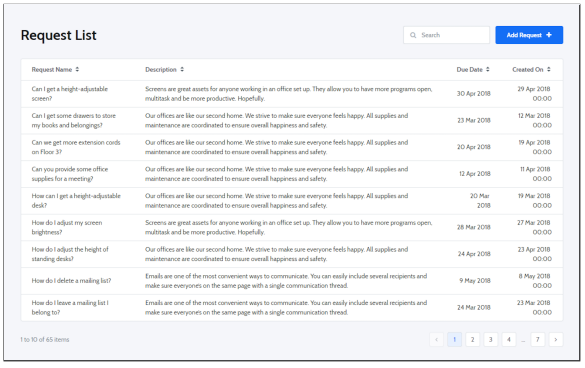
Listing 3.2: Generated UIDL for a user interface containing a single Header-1 element.

One of the key aspects when generating a user interface is to keep a consistent graphical appearance that matches OutSystems UI framework principles. Therefore, our UIDL generator imports the corresponding CSS style attributes of each element to the corresponding `style` key, which is supported by both `static` and `dynamic` nodes.

Finally, the UIDL is plugged into teleportHQ's React code generator, and a web application is rendered. This is the last step of the implemented pipeline, reflecting the result of early pipeline stages as a whole.



(a) Generated user interface using React.



(b) Real screen template of OutSystems UI.

Figure 3.27: Comparison of a (a) screenshot of a React web-generated app, with the detected containers marked with a dotted line and generic placeholders inside each UI element, with an (b) image of a screen template using the OutSystems UI Framework CSS that matches the same UI elements and page structure.

3.5 Complete Pipeline

Having described each stage inner workings individually, it is important to explain how they will be chained. The complete pipeline of the proposed solution, represented in Figure 3.28, shows that the system takes a hand-drawn UI sketch as input, which is pre-processed in the first stage of the algorithm. Then, in the second stage, the UI sketch is passed through the object detection algorithm and, in the third stage, the spatial grouping algorithm generates an agnostic structure of the sketch. Finally, using that information, a code generator produces the sketch corresponding source code.

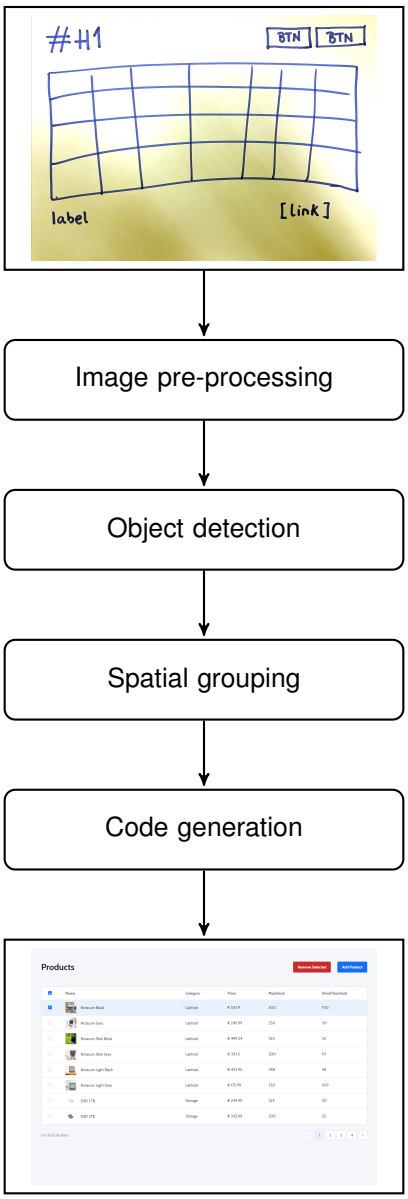


Figure 3.28: Proposed solution complete pipeline.

Chapter 4

Results

This chapter exhibits the overall performance of the implemented pipeline, focusing on the object detection performance metrics. The analysis of the results covers different experiments with our YOLOv5 model for human- and computer-generated datasets and also dives into the impact of early pipeline stages on the object detection results.

Both datasets are structured to be used for train, test, and validation using YOLOv5, as explained in Chapter 3. Google Colab provides access to powerful GPUs, which is critically important to accelerate the train, so we decided to implement our model in Colab, following the existing Ultralytics notebook [62].

A suitable number of training epochs was chosen to train our model with a custom dataset without exceeding Google Colab's usage quotas. Figure 4.1 shows that each fold took 27 to 39 minutes to train.

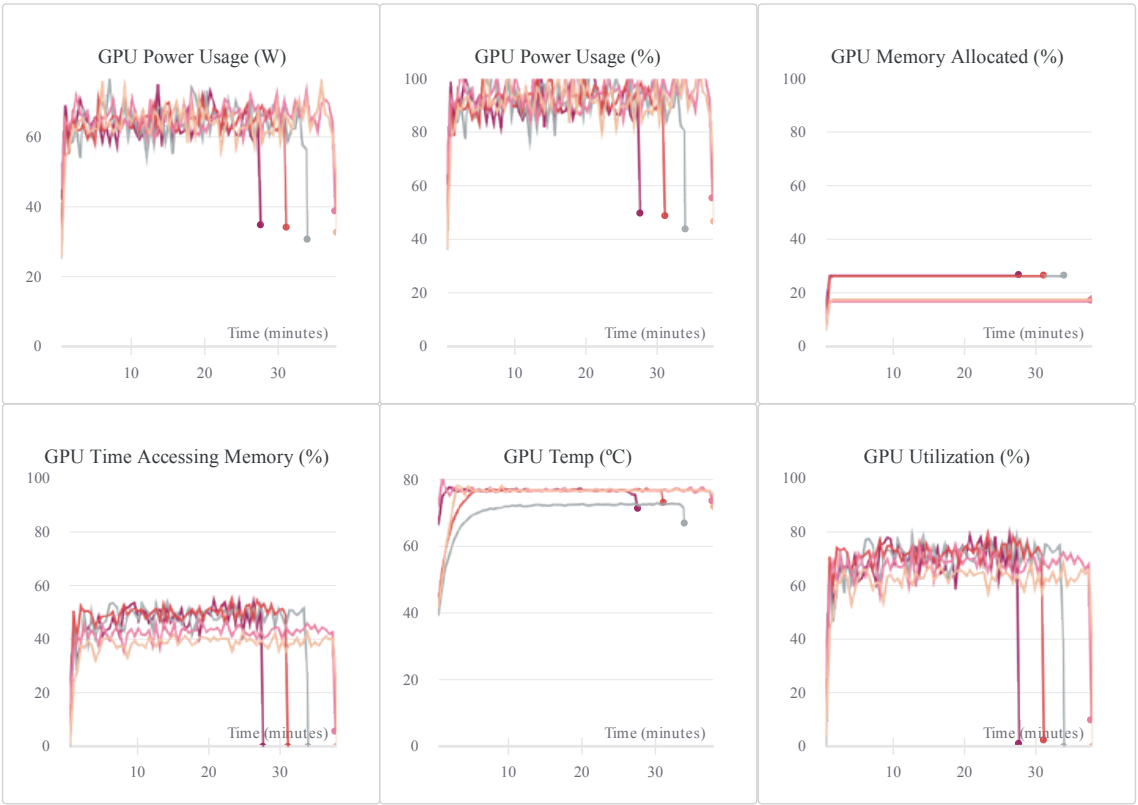


Figure 4.1: Google Colab hardware usage data and total train time for the human-generated dataset.

Due to the training time constraints imposed, the parameters of training the YOLOv5 model were limited to an image size of 1200 pixels, a batch size of 30 samples, and a total of 200 epochs. The hardware usage data presented in Figure 4.1 corresponds to the training conditions described in Chapter 3 and it was plotted using WandB [66].

In order to provide a fair comparison between the performance of different folds of the human-generated dataset, the splitting task followed the same consistent approach, focusing on the importance of keeping the most diverse styles of handwriting and hand-drawing apart, thus avoiding overfitting scenarios.

The detailed and individual performance results for each fold of the human-generated dataset are provided in Appendix B, while the average of all folds is presented in this chapter. As for the computer-generated dataset, a straightforward 60%, 20%, and 20% split of samples was assigned for train, test, and validation, respectively.

Accordingly, the following sections focus on quantitative and qualitative object detection results, thus providing a better understanding of the final results and a reliable perception of our model's performance for both human- and computer-generated datasets.

4.1 Quantitative Results

A quantitative analysis of key performance metrics is crucial to evaluate our model's performance. The most relevant and commonly used evaluation metric for object detection algorithms is mean average precision (mAP), which relies on several important concepts that were overview in Chapter 2.

In this section, we present the results for both the validation and the test sets of our human- and computer-generated datasets. For the 5-fold cross validation approach we present the accuracy results for the average of all folds.

We considered three different types of loss from YOLO: box loss, objectness loss and classification loss. The box loss represents how well the model can locate the center of a UI element and how well the predicted bounding box covers the entire hand-drawn representation. Objectness loss is a measure of the probability that an object exists in a proposed region of interest. If the objectivity is high, it means that the image window is likely to contain a UI element. Finally, classification loss evaluates how well the model predicts the correct class of the sketched UI elements.

We used early stopping to select the best weights. All bounding box losses are calculated by mean square loss, and the classification loss is calculated by cross-entropy loss. The x-axis represents epochs in all figures and the y-axis corresponds to the title of each sub-figure.

4.1.1 Human-generated Dataset

For the five folds of our human-generated dataset, the model improved swiftly in terms of precision, recall, and mean average precision before plateauing after about 200 epochs. The box, objectness and classification losses of the validation data also showed a rapid decline until around epoch 200.

Figure 4.2 presents the plots of box loss, objectness loss, classification loss, precision, recall, and mean average precision over the 200 training epochs of our human-generated dataset first fold.

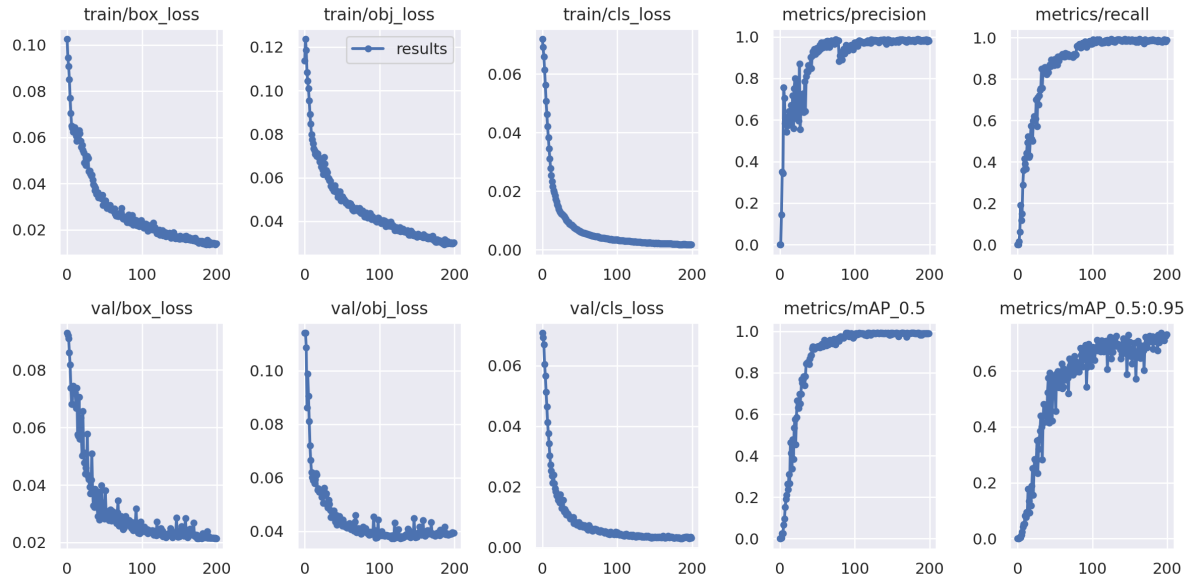


Figure 4.2: Performance plots for the human-generated dataset fold 1 model train losses, validation losses, and all evaluation metrics.

These plots were generated within Google Colab and were monitored during the train of the model using Tensorboard. However, as explained in Subsection 3.2.3, WandB was also used to compare different sessions in a more versatile way, allowing us to compare the behavior of different folds, as shown Figure 4.3.

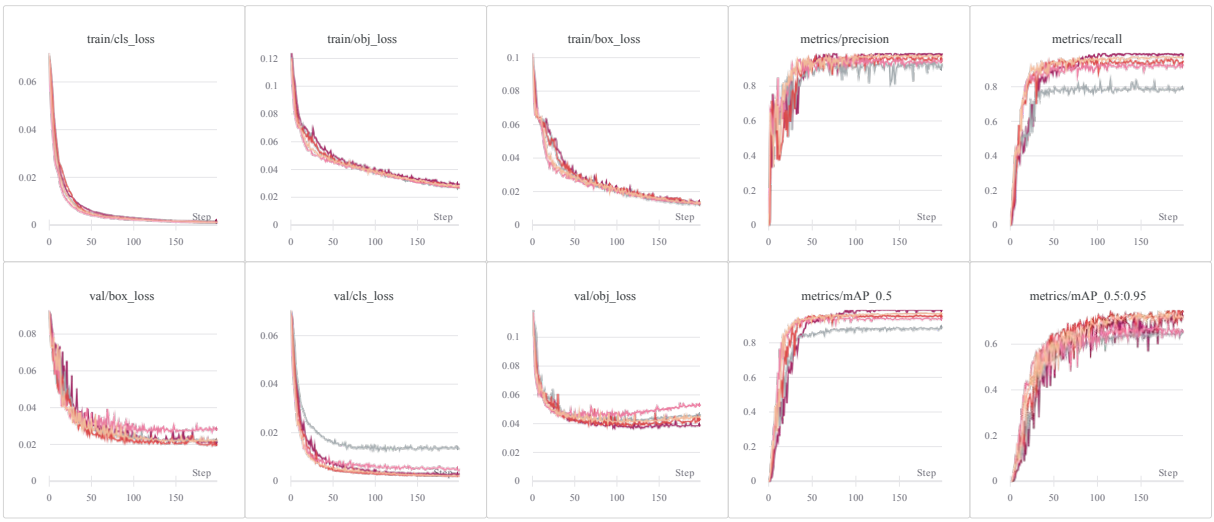


Figure 4.3: Overlapped plots of train loss, validation loss, and performance metrics of all five folds.

Besides analyzing the overall results for all folds, it is important to discuss the element-wise performance of the model to understand which elements are leading to better results or hurting the performance. The confusion matrix shown in Figure 4.4 is representative of the element-wise performance of the trained networks.

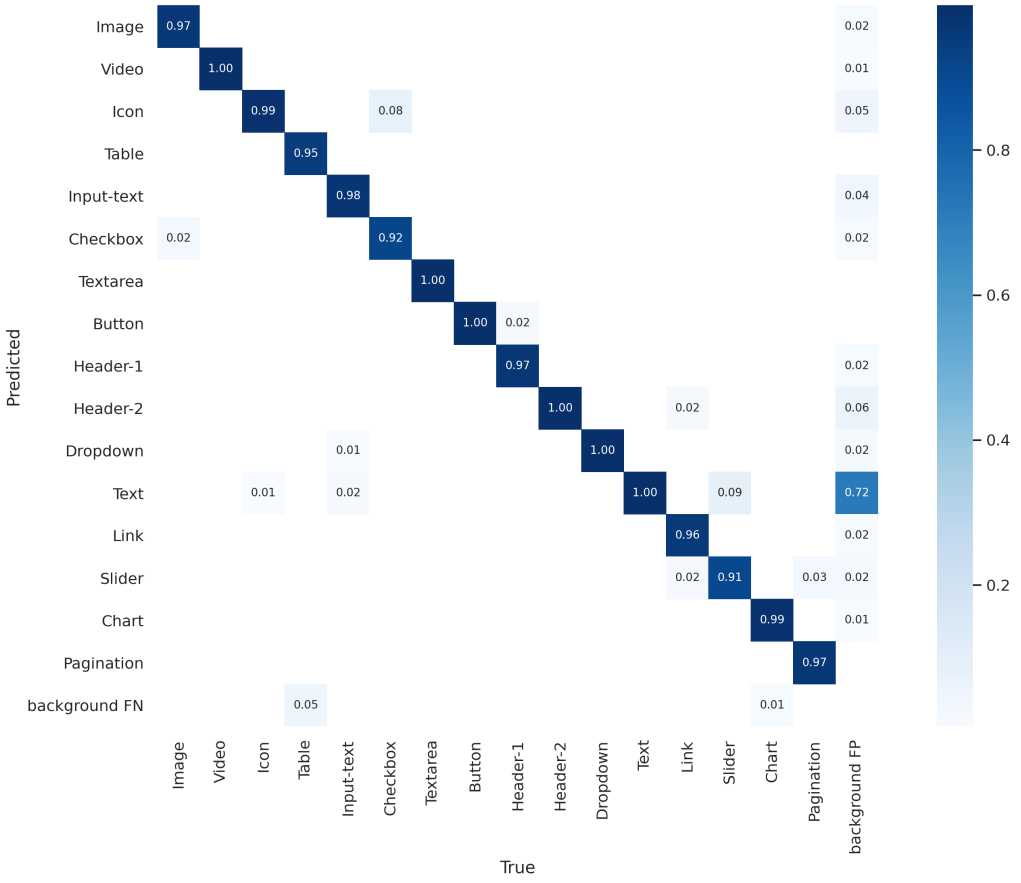


Figure 4.4: Confusion matrix for all 16 classes and background false negatives of fold 1.

The confusion matrix shows that most elements have excellent performance, considering that the predictions are correct between 91% and 100%. This specific visualization was extremely useful for fine-tuning UI elements representations. The overall scenario was significantly different when representations were not distinctive enough.

One of the issues that was identified earlier was related with images noise after the pre-processing binarization operation. Sometimes, little scratches or even shadows from the back-page are visible and binarized as *Text* elements.

This means that our model predicts a significant amount of *Text* elements where, in fact, there are no elements drawn at all (nearly 72% of image background false positives). The detailed analysis of this issue is mainly qualitative, so we will cover it in the next section.

Tables 4.1 and 4.2 present the results for the 5-fold approach followed for the human-generated dataset. These tables present the accuracy results for both the validation and test set, respectively.

We focused on three key performance metrics: precision, recall, and two mean average precision (mAP) values over different IoU thresholds (up to 0.5 and from 0.5 to 0.95). We also included how many samples of the 1,003 human-generated samples were present in the validation and test set (considering that they represent around one-fifth of the dataset, as explained in Section 3.1.3) and how many samples of each UI element are present in those images. In spite of the overall good results, it is clear that underrepresented elements show inferior results (e.g. *Slider* and *Checkbox*).

Table 4.1: Average 5-fold cross validation results for the human-generated validation set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	219	243	0.9402	0.975	0.990	0.856
Video	219	30	0.9594	0.7962	0.8528	0.649
Icon	219	200	0.959	0.9898	0.992	0.6526
Table	219	92	0.9818	0.8168	0.966	0.694
Input-text	219	174	0.888	0.9852	0.958	0.7386
Checkbox	219	33	0.919	0.8518	0.969	0.665
Textarea	219	47	0.987	0.9192	0.9502	0.769
Button	219	363	0.9838	0.9974	0.993	0.7734
Header-1	219	218	0.987	0.968	0.990	0.665
Header-2	219	386	0.954	0.9836	0.984	0.663
Dropdown	219	63	0.9706	0.8874	0.926	0.733
Text	219	1086	0.9558	0.996	0.995	0.636
Link	219	68	0.9332	0.9374	0.978	0.715
Slider	219	44	0.9142	0.59676	0.723	0.5162
Chart	219	88	0.947	0.976	0.9822	0.7638
Pagination	219	38	0.9136	0.821	0.8382	0.55
All Classes	219	3173	0.9496	0.9062	0.943	0.6898

Table 4.2: Average 5-fold cross validation results for the human-generated test set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	201	218	0.9568	0.9782	0.988	0.837
Video	201	30	0.9182	0.7096	0.814	0.584
Icon	201	182	0.965	0.9908	0.990	0.650
Table	201	84	0.99	0.8234	0.966	0.679
Input-text	201	158	0.899	0.9806	0.9534	0.715
Checkbox	201	33	0.878	0.9274	0.960	0.635
Textarea	201	44	0.9754	0.9168	0.9558	0.771
Button	201	331	0.9908	0.9976	0.994	0.776
Header-1	201	199	0.977	0.9644	0.9884	0.628
Header-2	201	349	0.9566	0.984	0.984	0.6316
Dropdown	201	59	0.928	0.874	0.9094	0.6974
Text	201	985	0.955	0.9878	0.990	0.613
Link	201	63	0.885	0.9846	0.982	0.709
Slider	201	43	0.8164	0.5533	0.714	0.507
Chart	201	84	0.928	0.9792	0.982	0.7494
Pagination	201	36	0.9152	0.8156	0.850	0.5394
All Classes	201	2898	0.933	0.9042	0.939	0.6704

4.1.2 Computer-generated Dataset Results

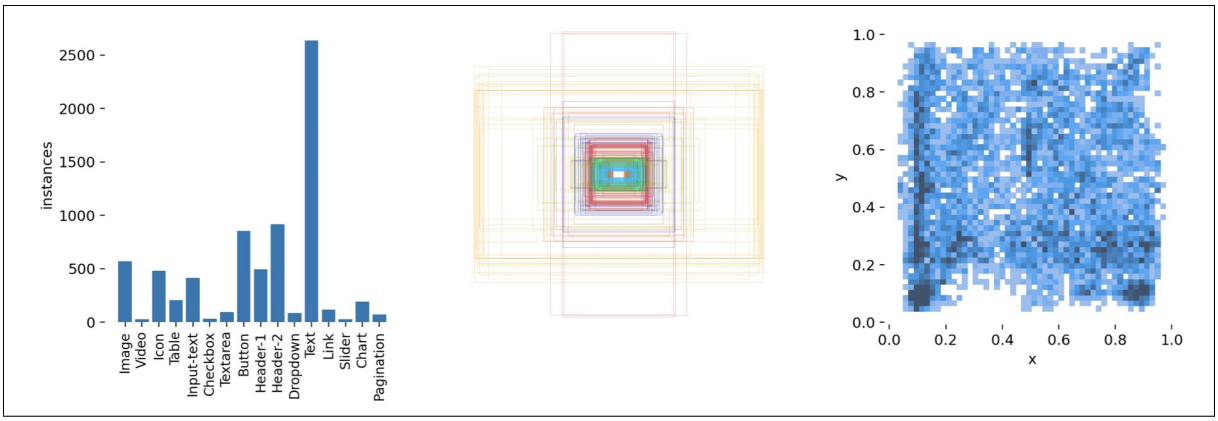
The main issues with the human-generated dataset results are the pen scratches identified as *Text* elements and the underrepresented classes, like the *Slider* and *Checkbox* elements, that are more rare in the dataset.

The dataset generator tool conceived for this thesis has many possible applications, but we focused on overcoming the main drawbacks of the human-generated dataset by using it to create a better equilibrium between element classes and perform more data augmentation operations with elements cropped from the original dataset.

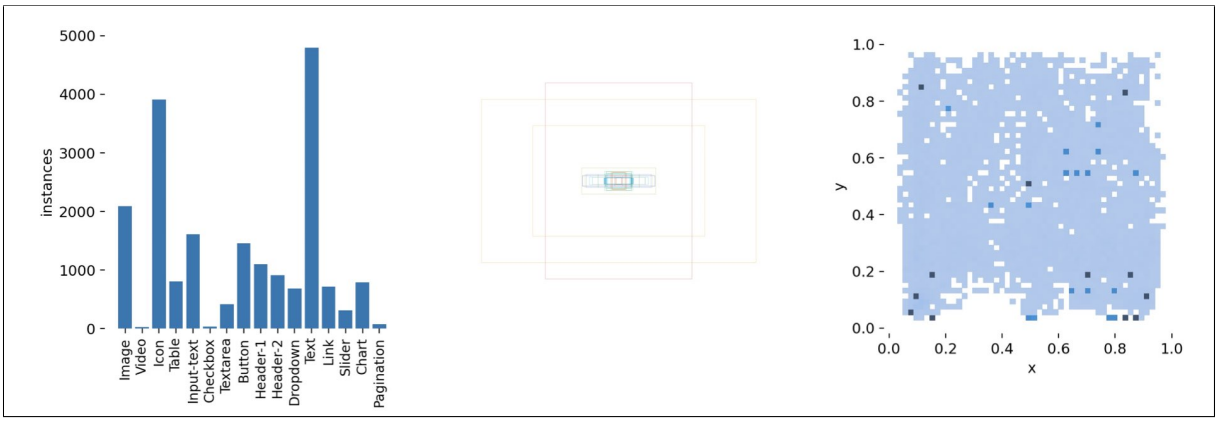
Figure 4.5 shows the main differences between human- and computer-generated datasets when it comes to the distribution of the number of images per class, the distribution of bounding boxes aspect ratios, and the distribution of bounding boxes center coordinates (x, y) for both datasets.

The computer-generated dataset allowed us to create a more even distribution of elements, thus duplicating the human-generated dataset size to a total of 2000 samples.

The total of elements per class, however, was not duplicated across the board, as we used OutSystems UI screen templates to create these samples, as covered in Chapter 3, and some classes are naturally more present in real-world user interfaces, sometimes appearing repeatedly in an application.



(a) Class image count, bounding boxes aspect ratios, and center distribution for the human-generated dataset.



(b) Class image count and bounding boxes aspect ratios, and center distribution for the computer-generated dataset.

Figure 4.5: Distribution of the number of images per class, bounding boxes aspect ratios, and bounding box center coordinates (x, y) for both datasets.

Besides adjusting the number of represented classes, the use of a dataset generator based on screen templates compromised the diversity of bounding boxes aspect ratios and their distribution.

The confusion matrix in Figure 4.6 shows that the issue with background false detections as *Text* elements reduced by over 55%, as the computer-generated samples do not include as much noise as real human hand-drawn samples.

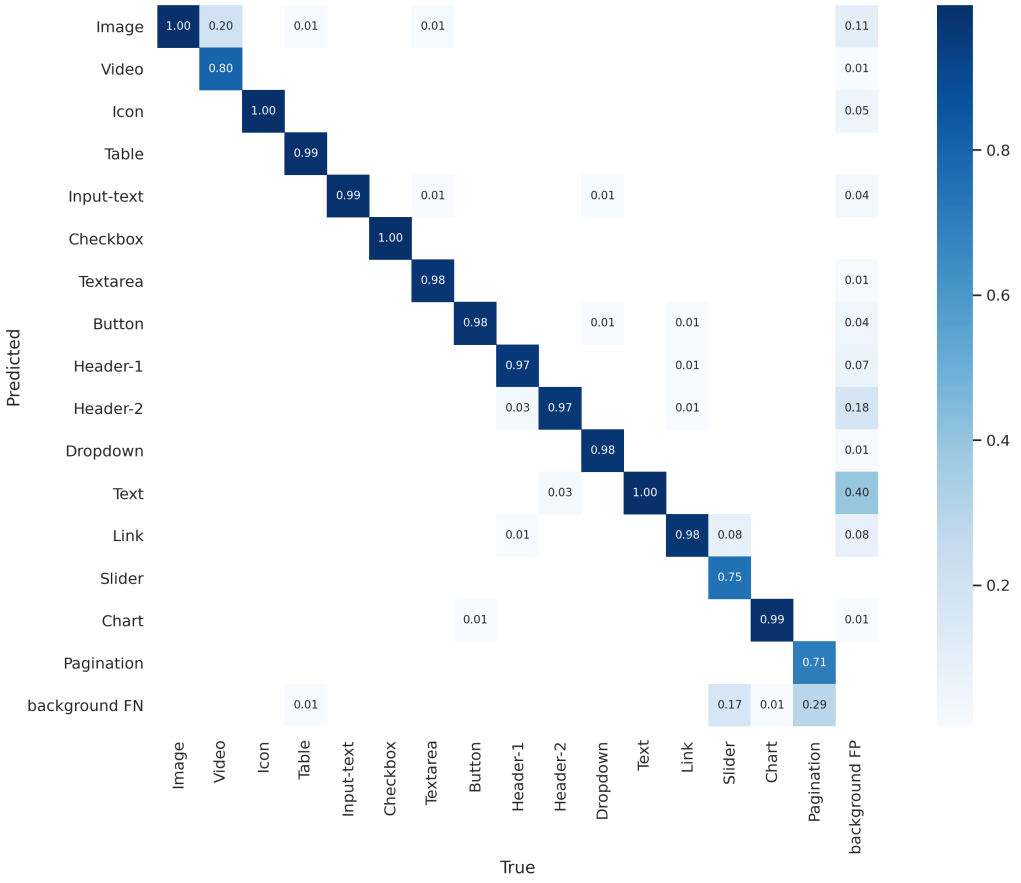


Figure 4.6: Confusion matrix for the model trained with a computer-generated dataset and tested with the human-generated dataset.

This was an important breakthrough to our model, but the overall accuracy was still not perfect and the results suggest that the class equilibrium need to be improved. All underrepresented classes, like the *Pagination*, *Slider*, and *Video* UI elements, still have inferior results to over-represented classes.

While data augmentation operations were used to increase the representations and worked brilliantly for some, they are not as effective for some representations. As an example, the straightforward representation of the *Icon* element can be easily augmented with several data augmentation operations without compromising its legibility. However, more meticulous representations that include text and small details are more difficult to augment.

The solution to improve the results for both datasets is, thus, collect more human-generated samples and manually label them. It is important to retain that the computer-generated dataset benefits from a larger human-generated dataset in double, considering that the human-generated dataset is used to feed the repository of hand-drawn representations that will be used by our dataset generator.

As for the detailed plots of evaluation metrics, this model also improved swiftly in terms of precision, recall, and mean average precision before plateauing after about 200 epochs.

The box, objectness and classification losses of the validation data also showed a rapid decline until around epoch 200. Figure 4.7 presents the detailed plots of losses, precision, recall, and mean average precision over the 200 training epochs of our computer-generated dataset.

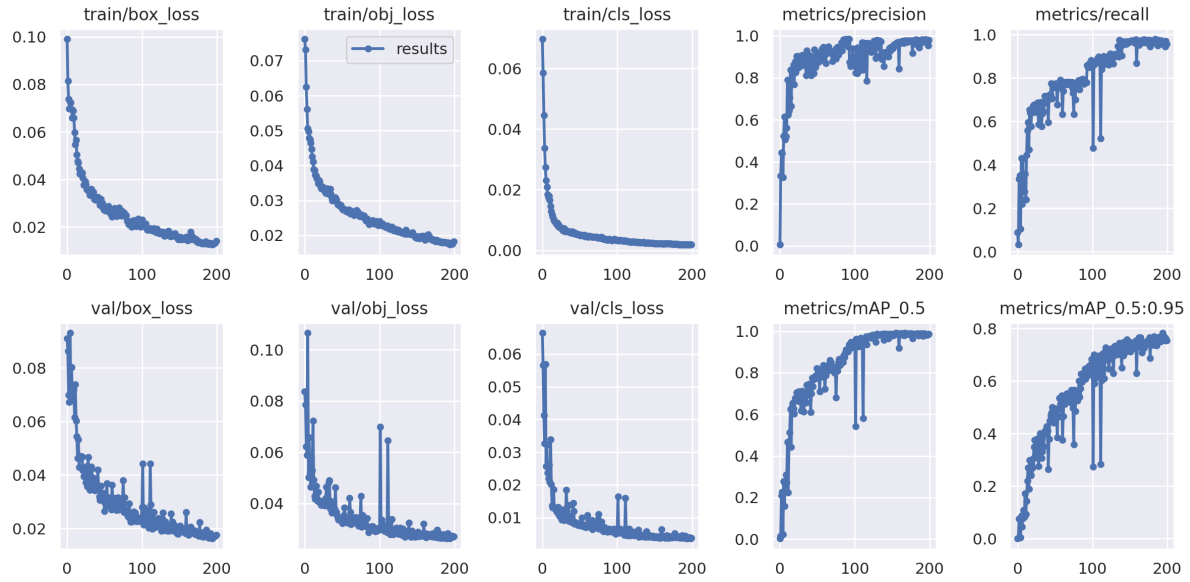


Figure 4.7: Performance plots for the computer-generated dataset model train losses, validation losses, and all evaluation metrics.

Table 4.3 presents the performance metrics of precision, recall, and two mAP values over different IoU thresholds (up to 0.5 and from 0.5 to 0.95) for this model trained with computer-generated samples and tested with human-generated sketches.

Table 4.3: Detection results for the test set for the human-generated test set.

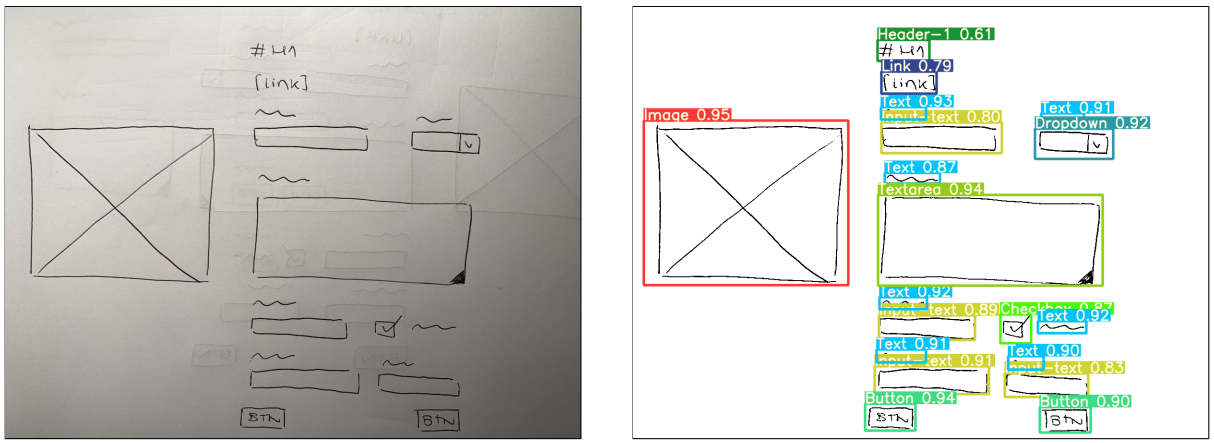
Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	400	529	0.971	1	0.995	0.877
Video	400	10	0.964	0.9	0.986	0.689
Icon	400	1088	0.996	1	0.996	0.85
Table	400	184	1	0.979	0.995	0.855
Input-text	400	366	0.987	0.992	0.993	0.798
Checkbox	400	12	0.993	1	0.995	0.759
Textarea	400	137	0.996	0.993	0.995	0.868
Button	400	459	0.994	0.989	0.995	0.813
Header-1	400	299	0.979	0.983	0.992	0.769
Header-2	400	368	0.953	1	0.994	0.671
Dropdown	400	135	0.993	0.993	0.995	0.756
Text	400	1568	0.964	0.998	0.996	0.665
Link	400	146	0.934	1	0.995	0.821
Slider	400	12	1	0.804	0.938	0.562
Chart	400	169	0.973	0.994	0.991	0.66
Pagination	400	24	1	0.667	0.932	0.642
All Classes	400	5506	0.981	0.956	0.986	0.753

4.2 Qualitative Results

Evaluating the performance of our model required more than discussing and analyzing YOLO's quantitative metrics. In fact, qualitative results were just as important to find opportunities for improving our model, fine-tune our elements representations, adjust the model hyperparameters, and modify our data augmentation stages.

Throughout the implementation of this thesis, a continuous qualitative evaluation was pursued to better perceive the performance of our object detector in real-world scenarios. This section covers how this qualitative analysis was conducted, namely the most frequently identified issues and how they were addressed.

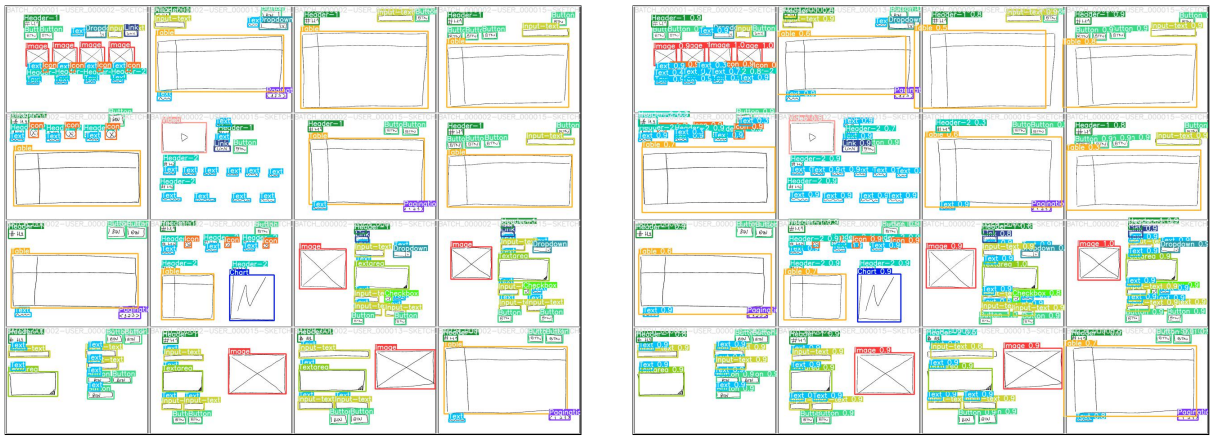
As described in Section 3.2, the approach followed to evaluate the qualitative results consisted of exporting the predicted labels by printing the model primitives as bounding boxes onto with their corresponding class onto the image. As shown in Figures 4.8 and 4.9, this was done individually and in bulk, by comparing the ground-truth labels printed onto the input images and the predicted labels.



(a) Original photo took and uploaded by the user.

(b) Object detection model output.

Figure 4.8: Example of an accurate detection performed by the model.



(a) Mosaic with ground-truth labels.

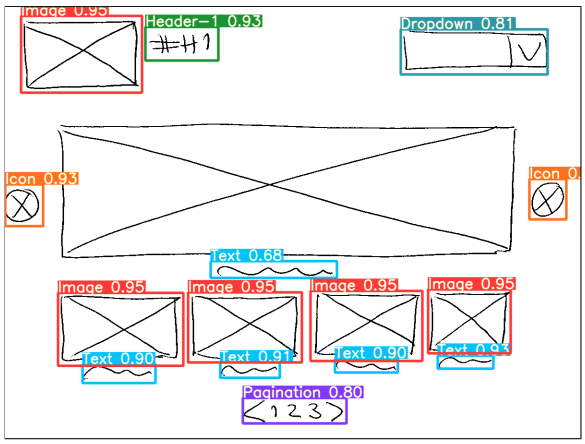
(b) Mosaic with corresponding predicted labels.

Figure 4.9: Example of ground-truth and predicted labels qualitative evaluation in bulk.

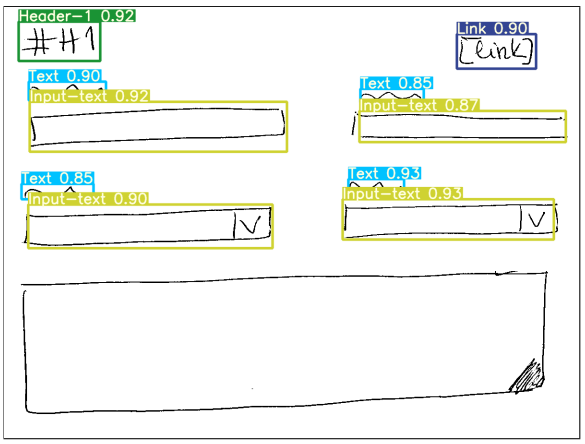
The qualitative analysis of our object detection results shows that there are two important issues to be considered. The first is related to the aspect ratio of the represented UI elements and the second is related to the pre-processing stage of the implemented pipeline. Figure 4.10 illustrates the first and most common issue of our detections. The *Image*, *Textarea*, and *Table* elements drawn in the first three sketches are not labeled due to the low confidence score of the detections being below 0.5.

This low confidence score is due to the aspect ratio variability of these three elements in the depicted hand-drawn sketches, which are not similar to any OutSystems UI screen template. Although it may seem straightforward to match a very wide representation of an element with a regular one, it requires more training over more diverse sketches and, thus, a larger dataset.

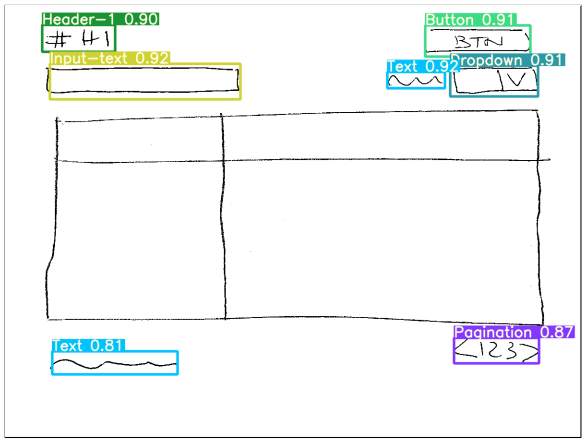
Also, the implementation followed for the human-generated dataset split aggravates this specific issue. While it was important to isolate users with different calligraphy styles to avoid overfitting, we also ended up segregating the sketches of volunteers that invented their own user interfaces from the ground up and the volunteers that were inspired by OutSystems UI screen templates. The fourth example shown in Figure 4.10 presents a lower confidence score for the *Table* element, but it is still high enough to surpass the 0.5 threshold. This slightly higher score is due to the fact that this sketch was inspired by the *Transactions Dashboard* screen template, making this specific aspect ratio more common.



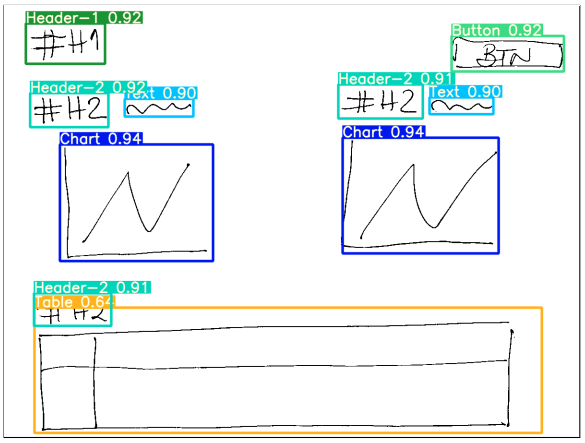
(a) Example of an unusually wide *Image* element.



(b) Example of an unusually wide *Textarea* element.



(c) Example of a deformed *Table* element.



(d) Example of an unusually wide *Table* element.

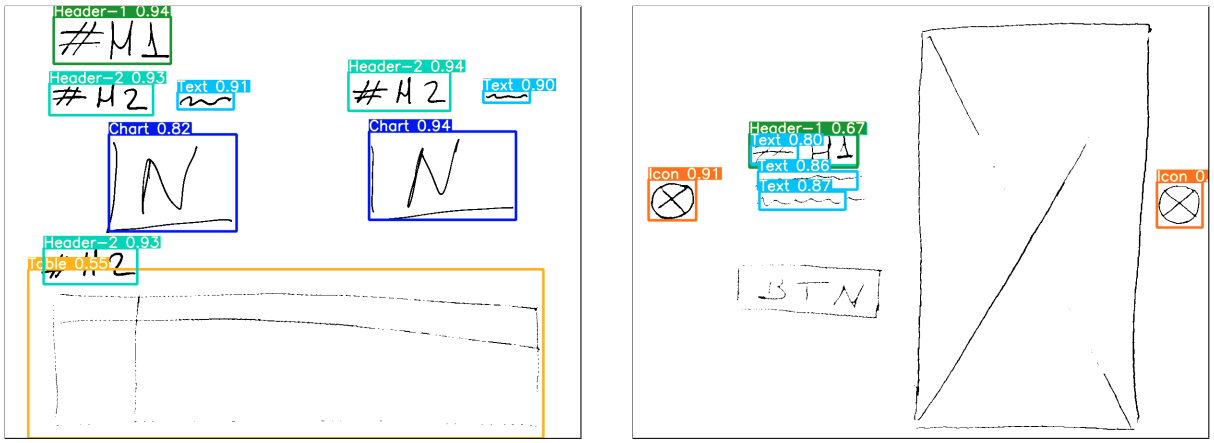
Figure 4.10: Examples of unusually wide and deformed elements in sketches.

Figure 4.11 shows the second issue of some of our detections, which is related with the low quality of the input image. The first example shows a low confidence score for the *Table* element and the second example shows that the *Button* and *Table* elements are almost imperceptible, making the confidence score below the 0.5 threshold.

This issue is due to the discrepancy of light, brilliance, shades, and saturation across different users' photos. In these specific cases, the original photos were shaded, meaning that a significant image brightness decrease from the center to the corners was present, therefore influencing the image quality by creating unwanted dark or shaded edges that the pre-processing algorithm is then unable to binarize correctly.

A significant variation of colors over the imaging field may occur due to the camera used by the user having a small sensor, which was demonstrating by analyzing the exchangeable image file format metadata of the uploaded photos.

In order to achieve better results, several approaches were tested during the implementation of our pipeline, namely skeletonization and reconstruction based on graph morphological transformations [67], but these techniques were only successful in a limited small group of images that did not present too much noise.



(a) Poor performance detecting a *Table*.

(b) No detection of a *Button* and an *Image*.

Figure 4.11: Example of detection issues due to pre-processing binarization.

Chapter 5

Conclusions

This thesis was developed on the premise of conducting a successful pioneering research project to introduce a novel approach which explores how the state-of-the-art computer vision algorithms can be used for code generation from hand-drawn UI sketches.

The structure of the chapters in this thesis was conceived to answer the pre-set aims of reviewing background literature, exploring the state-of-the-art methodologies to follow, proposing a solution for the task at hand, and finally analyze the obtained results.

These concluding remarks provide an overture to the achievements and main contributions of this thesis, but also yield a reflection on the effectiveness of the proposed solution, aiming to encourage future improvements and possible directions, in order to pursue the full potential of this thesis.

5.1 Achievements

This thesis proposes an automatic tool, based on machine learning, that converts hand-drawn UI sketches into code and eventually generates the actual UI, hence availing the agile experimentation afforded by hand-made sketches.

During the literature and related work review, it became clear that this thesis could introduce a novel, broader approach to the task at hand, by supporting the largest number of different UI elements possible and converting the hand-drawn sketches straight into the real-world UI, preparing all the intermediate steps to work seamlessly.

More specifically, the proposed solution pipeline consists of a software tool that identifies the sketched elements using computer vision, evaluates their hierarchy, and finally generates the corresponding UI.

The proposed solution maximizes the efficiency of the design process and consequently shortens the SDLC of software applications, which not only improves the manageability, objectivity, and control of projects, but also reduces time-to-market and cost-to-market of applications, delivering meticulous and more substantial solutions in a shorter time.

In order to achieve the ultimate objectives of this thesis, significant milestones were accomplished throughout the multiple stages described in Chapter 3, namely:

- Extracting all relevant data from the input hand-drawn image, using a computer vision model to detect hand-drawn UI elements, which required:
  - Disambiguating UI elements representations, merging redundant elements and setting a unique pattern for all elements included in the OutSystems UI Framework [55].
  - Producing a dedicated human-generated dataset with over fourteen thousand UI elements for the task at hand, by organizing a major crowdsourced effort and manually labeling the contributions.
  - Developing a dataset generator tool inspired by the *sketchifying* approach [40], taking advantage of data augmentation techniques to further expand the number of samples in the dataset.
  - Conceiving a pre-processing pipeline of steps to be applied to the input images before feeding the data to the network.
  - Creating a new dataset splitter that could accommodate the specific file formats and absolute coordinates systems of all stages, in order to generate the train, test, and validation sets from the complete dataset.
  - Programming a straightforward cloud-based pipeline to evaluate the results of YOLOv5 [34] for train, test and validation.
- Converting the computer vision model primitives into a real-world UI, by generating its source code, which required:
  - Establishing heuristics to predict groups of elements and implementing clustering techniques, such as quadtrees for image processing, mesh generation, and collision detection.
  - Developing a tool to infer the hierarchical structure of the sketched UI elements from the model primitives, in order to preserve the designed layout when generating the code.
  - Producing files with the agnostic structure of spatially grouped UI elements to be interpreted while the UI is rendered.
  - Creating a code generation pipeline to ultimately compile and render the source code produced from the computer vision model primitives and the inferred structure, aiming to visualize the final UI.

A paper was produced during the UI elements representations disambiguation study, covering the analysis of results that was performed to disambiguate and create unique hand-drawn representations, which will be submitted to a future conference.

Considering the different programming languages used and the diverse inner workings of each stage, another important achievement of this thesis was finding a way to integrate all intermediate steps, by matching the produced outputs with the expected inputs. This was accomplished by following a language-agnostic API rationale throughout the thesis, which will allow future research projects in this area to modify or easily upgrade any stage, without compromising the dataset produced or any methods developed.

5.2 Future Work

Automatic UI generation has been building momentum in software development and continues to drive start-ups focused on this field of research. However, it is far from mature.

As more developers and designers find themselves working on laborious, unremarkable, and time-consuming tasks that require multiple iterations and do not always lead to the best result, it becomes more evident that this field of research will continue to expand its amplitude.

Having highlighted the most substantial achievements of this thesis, it is important to retain that further improvements are possible. Some ideas, which emerged from the development of the proposed solution, might encourage future pioneering research projects in this field, namely:

- Evolving the YOLOv5 hyperparameters to control the training, avoid overfitting, and find optimal values, hence improving the computer vision results.
- Testing and analyzing the results of YOLOv5 for video frames coming from a continuous stream, in order to take full advantage of real-time automatic UI generation.
- Improving the heuristics to better infer the hierarchical structure of hand-drawn UI elements from absolute positions identified by the computer vision model.
- Exploring more code generators and APIs that take an agnostic hierarchy of spatially arranged elements to generate a final UI.
- Implementing an optical character recognition system to avoid static placeholders (e.g. buttons action text, search field suggestion placeholders, text labels, headers, etc.).

Bibliography

[1] Uizard. Uizard. https://uizard.io/, 2019. [Online; accessed October 23, 2019].

[2] N. M. C. Alves. From mockup to ui. Master's thesis, Instituto Superior Técnico, January 2019.

[3] teleportHQ. *The Second Version of Our Vision API*. https://teleporthq.io/blog/new-vision-api, 2019. [Online; accessed October 13, 2019].

[4] Microsoft. *Sketch2Code*. https://github.com/Microsoft/ailab/tree/master/Sketch2Code/, 2018. [Online; accessed October 23, 2019].

[5] Airbnb. *Sketching Interfaces*. https://airbnb.design/sketching-interfaces/, 2019. [Online; accessed October 23, 2019].

[6] C. M. Bishop and N. M. Nasrabadi. *Pattern Recognition and Machine Learning*. *J. Electronic Imaging*, 16(4):049901, 2007. doi: 10.1117/1.2819119. URL https://doi.org/10.1117/1.2819119.

[7] E. Alpaydin. *Introduction to machine learning*. Adaptive computation and machine learning. MIT Press, 2004. ISBN 978-0-262-01211-9.

[8] T. M. Mitchell. Machine Learning*, International Edition*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997. ISBN 0071154671. URL http://www.worldcat.org/oclc/61321007.

[9] X. Zhu, Z. Ghahramani, and J. D. Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*, pages 912–919, 2003. URL http://www.aaai.org/Library/ICML/2003/icml03-118.php.

[10] M. Anthony and P. L. Bartlett. *Neural Network Learning - Theoretical Foundations*. Cambridge University Press, 2002. ISBN 978-0-521-57353-5. URL http://www.cambridge.org/gb/knowledge/isbn/item1154061/?site_locale=en_GB.

[11] M. Manavazhahan. *A Study of Activation Functions for Neural Networks*. *Computer Science and Computer Engineering Undergraduate Honors Theses*, 2017.

[12] K. P. Murphy. *Machine learning - a probabilistic perspective*. Adaptive computation and machine learning series. MIT Press, 2012. ISBN 0262018020.

[13] C. Goller and A. Küchler. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of International Conference on Neural Networks (ICNN'96), Washington, DC, USA, June 3-6, 1996*, pages 347–352, 1996. doi: 10.1109/ICNN.1996.548916. URL https://doi.org/10.1109/ICNN.1996.548916.

[14] L. Deng and D. Yu. Deep learning: Methods and applications. *Foundations and Trends in Signal Processing*, 7(3-4):197–387, 2014. doi: 10.1561/2000000039. URL https://doi.org/10.1561/2000000039.

[15] H. Unger, K. Kyamakya, and J. Kacprzyk, editors. *Autonomous Systems: Developments and Trends*, volume 391 of *Studies in Computational Intelligence*. Springer, 2012. ISBN 978-3-642-24805-4. doi: 10.1007/978-3-642-24806-1. URL https://doi.org/10.1007/978-3-642-24806-1.

[16] S. S. Liew, M. Khalil-Hani, F. Radzi, and R. Bakhteri. Gender classification: A convolutional neural network approach. *Turkish Journal of Electrical Engineering and Computer Sciences*, 24:1248–1264, 03 2016. doi: 10.3906/elk-1311-58.

[17] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–44, 05 2015. doi: 10.1038/nature14539.

[18] Y. LeCun, S. Chopra, M. Ranzato, and F. J. Huang. Energy-based models in document recognition and computer vision. In *9th International Conference on Document Analysis and Recognition (ICDAR 2007), 23-26 September, Curitiba, Paraná, Brazil*, pages 337–341. IEEE Computer Society, 2007. doi: 10.1109/ICDAR.2007.4378728. URL https://doi.org/10.1109/ICDAR.2007.4378728.

[19] Y. Boureau, N. L. Roux, F. R. Bach, J. Ponce, and Y. LeCun. Ask the locals: Multi-way local pooling for image recognition. In *IEEE International Conference on Computer Vision, ICCV 2011, Barcelona, Spain, November 6-13, 2011*, pages 2651–2658, 2011. doi: 10.1109/ICCV.2011.6126555. URL https://doi.org/10.1109/ICCV.2011.6126555.

[20] I. J. Goodfellow, Y. Bengio, and A. C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN 978-0-262-03561-3. URL http://www.deeplearningbook.org/.

[21] T. Liu, S. Fang, Y. Zhao, P. Wang, and J. Zhang. Implementation of training convolutional neural networks. *CoRR*, abs/1506.01195, 2015. URL http://arxiv.org/abs/1506.01195.

[22] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014. URL http://dl.acm.org/citation.cfm?id=2670313.

[23] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005), 20-26*

June 2005, San Diego, CA, USA, pages 886–893, 2005. doi: 10.1109/CVPR.2005.177. URL https://doi.org/10.1109/CVPR.2005.177.

[24] J. H. Bappy and A. K. Roy-Chowdhury. CNN based region proposals for efficient object detection. In *2016 IEEE International Conference on Image Processing, ICIP 2016, Phoenix, AZ, USA, September 25-28, 2016*, pages 3658–3662, 2016. doi: 10.1109/ICIP.2016.7533042. URL https://doi.org/10.1109/ICIP.2016.7533042.

[25] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. URL http://arxiv.org/abs/1311.2524.

[26] P. Dong and W. Wang. Better region proposals for pedestrian detection with R-CNN. In *2016 Visual Communications and Image Processing, VCIP 2016, Chengdu, China, November 27-30, 2016*, pages 1–4, 2016. doi: 10.1109/VCIP.2016.7805452. URL https://doi.org/10.1109/VCIP.2016.7805452.

[27] R. B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015. URL http://arxiv.org/abs/1504.08083.

[28] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015. URL http://arxiv.org/abs/1506.01497.

[29] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg. SSD: single shot multibox detector. In B. Leibe, J. Matas, N. Sebe, and M. Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I*, volume 9905 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2016. doi: 10.1007/978-3-319-46448-0_2. URL https://doi.org/10.1007/978-3-319-46448-0_2.

[30] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015. URL http://arxiv.org/abs/1506.02640.

[31] J. Redmon and A. Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016. URL http://arxiv.org/abs/1612.08242.

[32] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018. URL http://arxiv.org/abs/1804.02767.

[33] A. Bochkovskiy, C. Wang, and H. M. Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020. URL https://arxiv.org/abs/2004.10934.

[34] G. Jocher, A. Stoken, J. Borovec, NanoCode012, ChristopherSTAN, L. Changyu, Laughing, tkianai, yxNONG, A. Hogan, lorenzomammana, AlexWang1900, A. Chaurasia, L. Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, Durgesh, F. Ingham, Frederik, Guilhen, A. Colmagro, H. Ye, Jacobsolawetz, J. Poznanski, J. Fang, J. Kim, K. Doan, and L. Y. . ultralytics/yolov5: v4.0 -

nn.SiLU() activations, Weights & Biases logging, PyTorch Hub integration, Jan. 2021. URL https://doi.org/10.5281/zenodo.4418161.

[35] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. D. Reid, and S. Savarese. Generalized intersection over union: A metric and a loss for bounding box regression. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 658–666. Computer Vision Foundation / IEEE, 2019. doi: 10.1109/CVPR.2019.00075. URL http://openaccess.thecvf.com/content_CVPR_2019/html/Rezatofighi_Generalized_Intersection_Over_Union_A_Metric_and_a_Loss_for_CVPR_2019_paper.html.

[36] R. Padilla, S. L. Netto, and E. A. B. da Silva. A survey on performance metrics for object-detection algorithms. In *2020 International Conference on Systems, Signals and Image Processing, IWSSIP 2020, Niterói, Brazil, July 1-3, 2020*, pages 237–242. IEEE, 2020. doi: 10.1109/IWSSIP48289.2020.9145130. URL https://doi.org/10.1109/IWSSIP48289.2020.9145130.

[37] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill Book Company, 1984. ISBN 0-07-054484-0.

[38] M. Everingham, L. V. Gool, C. K. I. Williams, J. M. Winn, and A. Zisserman. The pascal visual object classes (VOC) challenge. *Int. J. Comput. Vis.*, 88(2):303–338, 2010. doi: 10.1007/s11263-009-0275-4. URL https://doi.org/10.1007/s11263-009-0275-4.

[39] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2017. doi: 10.1145/3065386. URL http://doi.acm.org/10.1145/3065386.

[40] R. K. Sarvadevabhatla, I. Dwivedi, A. Biswas, S. Manocha, and V. B. R. Sketchparse: Towards rich descriptions for poorly drawn sketches using multi-task hierarchical deep networks. In *Proceedings of the 2017 ACM on Multimedia Conference, MM 2017, Mountain View, CA, USA, October 23-27, 2017*, pages 10–18, 2017. doi: 10.1145/3123266.3123270. URL https://doi.org/10.1145/3123266.3123270.

[41] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1409.1556.

[42] P. Pawara, E. Okafor, L. Schomaker, and M. A. Wiering. Data augmentation for plant classification. In J. Blanc-Talon, R. Penne, W. Philips, D. C. Popescu, and P. Scheunders, editors, *Advanced Concepts for Intelligent Vision Systems - 18th International Conference, ACIVS 2017, Antwerp, Belgium, September 18-21, 2017, Proceedings*, volume 10617 of *Lecture Notes in Computer Science*, pages 615–626. Springer, 2017. doi: 10.1007/978-3-319-70353-4_52. URL https://doi.org/10.1007/978-3-319-70353-4_52.

[43] R. M. Gibson, A. Ahmadinia, S. G. McMeekin, N. C. Strang, and G. Morison. A reconfigurable real-time morphological system for augmented vision. *EURASIP J. Adv. Signal Process.*, 2013:134, 2013. doi: 10.1186/1687-6180-2013-134. URL https://doi.org/10.1186/1687-6180-2013-134.

[44] M. Holzer, F. Schumacher, T. Greiner, and W. Rosenstiel. Optimized hardware architecture of a smart camera with novel cyclic image line storage structures for morphological raster scan image processing. In *2012 IEEE International Conference on Emerging Signal Processing Applications, ESPA 2012, Las Vegas, NV, USA, January 12-14, 2012*, pages 83–86. IEEE, 2012. doi: 10.1109/ESPA.2012.6152451. URL https://doi.org/10.1109/ESPA.2012.6152451.

[45] Y. Zheng, H. Yao, X. Sun, S. Zhang, S. Zhao, and F. Porikli. Sketch-specific data augmentation for freehand sketch recognition. *CoRR*, abs/1910.06038, 2019. URL http://arxiv.org/abs/1910.06038.

[46] M. Eitz, J. Hays, and M. Alexa. How do humans sketch objects? *ACM Trans. Graph.*, 31(4):44:1–44:10, 2012. doi: 10.1145/2185520.2185540. URL https://doi.org/10.1145/2185520.2185540.

[47] S. Bethu, B. S. Babu, K. Madhavi, and P. G. Krishna. Algorithm selection and model evaluation in application design using machine learning. In S. Boumerdassi, É. Renault, and P. Mühlethaler, editors, *Machine Learning for Networking - Second IFIP TC 6 International Conference, MLN 2019, Paris, France, December 3-5, 2019, Revised Selected Papers*, volume 12081 of *Lecture Notes in Computer Science*, pages 175–195. Springer, 2019. doi: 10.1007/978-3-030-45778-5_12. URL https://doi.org/10.1007/978-3-030-45778-5_12.

[48] S. Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *CoRR*, abs/1811.12808, 2018. URL http://arxiv.org/abs/1811.12808.

[49] C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors. *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, 2004. Springer. ISBN 3-540-22119-0. doi: 10.1007/b98156. URL https://doi.org/10.1007/b98156.

[50] T. Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *CoRR*, abs/1705.07962, 2017. URL http://arxiv.org/abs/1705.07962.

[51] M. Fowler. *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley, 2011. ISBN 978-0-321-71294-3. URL http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0321712943,00.html.

[52] K. Ellis, D. Ritchie, A. Solar-Lezama, and J. B. Tenenbaum. Learning to infer graphics programs from hand-drawn images. *CoRR*, abs/1707.09627, 2017. URL http://arxiv.org/abs/1707.09627.

[53] P. Koiran and E. D. Sontag. Vapnik-chervonenkis dimension of recurrent neural networks. *Discrete Applied Mathematics*, 86(1):63–79, 1998. doi: 10.1016/S0166-218X(98)00014-6. URL https://doi.org/10.1016/S0166-218X(98)00014-6.

[54] OutSystems. *OutSystems UI Framework Screen Templates Support Documentation*. https://outsystemsui.outsystems.com/OutSystemsUIWebsite/ScreenOverview, 2019. [Online; accessed December 18, 2019].

[55] OutSystems. *OutSystems UI Framework Patterns Support Documentation*. https://outsystemsui.outsystems.com/OutSystemsUIWebsite/PatternOverview, 2019. [Online; accessed December 18, 2019].

[56] J. Sauvola and M. Pietikäinen. Adaptive document image binarization. *Pattern Recognition*, 33 (2):225–236, 2000. ISSN 0031-3203. doi: https://doi.org/10.1016/S0031-3203(99)00055-2. URL https://www.sciencedirect.com/science/article/pii/S0031320399000552.

[57] LabelImg. Labelimg. https://github.com/tzutalin/labelImg, 2020. [Online; accessed November 3, 2020].

[58] Ultralytics. *YOLOv5 wiki with tutorials, environments, and the current repository status.* https://github.com/ultralytics/yolov5/wiki, 2019. [Online; accessed November 3, 2020].

[59] Roboflow. *Label Training Images and Export To Any Format*. https://roboflow.com/annotate, 2019. [Online; accessed October 13, 2019].

[60] S. F. Conservancy. *Selenium: Automating Web Browsers*. https://www.selenium.dev/about/, 2019. [Online; accessed April 23, 2020].

[61] M. Corporation. Mdn web docs: Element.getboundingclientrect. https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect, 2019. [Online; accessed October 23, 2019].

[62] Ultralytics. *A family of object detection architectures and models.* https://ultralytics.com/yolov5, 2019. [Online; accessed October 13, 2019].

[63] C. Wang, H. M. Liao, I. Yeh, Y. Wu, P. Chen, and J. Hsieh. Cspnet: A new backbone that can enhance learning capability of CNN. *CoRR*, abs/1911.11929, 2019. URL http://arxiv.org/abs/1911.11929.

[64] OutSystems. *OutSystems UI Framework*. https://outsystemsui.outsystems.com/, 2019. [Online; accessed December 18, 2019].

[65] teleportHQ. *User Interface Definition Language*. https://docs.teleporthq.io/uidl/, 2019. [Online; accessed October 13, 2019].

[66] L. Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

[67] H. M. Sharifipour, B. Yousefi, and X. P. V. Maldague. Skeletonization and reconstruction based on graph morphological transformations. *CoRR*, abs/2009.07970, 2020. URL https://arxiv.org/abs/2009.07970.

Appendix A

Crowdsourcing Instructions

This Appendix contains the instructions document that was sent to the volunteers that contributed with unique hand-drawn sketches for the human generated dataset described in Chapter 3. The main purpose of this document was to concisely present the thesis work motivation, provide the list of correct representations for the supported UI elements, and illustrate 17 examples of valid user interfaces that could potentially be used as an inspiration for their own sketches. The document is written in European Portuguese, which was the native language of all participating volunteers.

TÉCNICO LISBOA | outsystems

Dataset Crowdsourcing

UI Gen: Desenhar para gerar uma interface

Dissertação • Ano Letivo 2019/2020 • 2.º Semestre

Gonçalo Correia de Matos

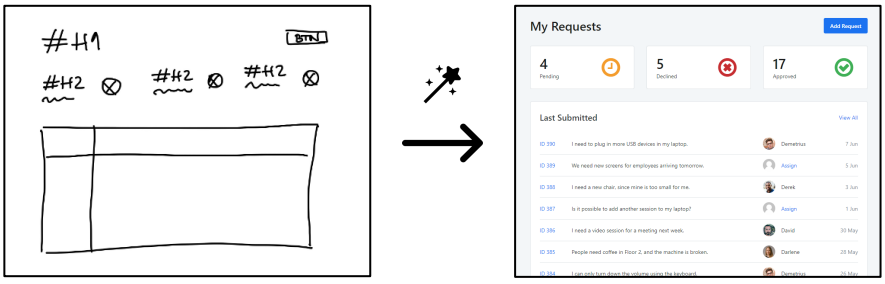
UI Gen
VERSÃO DIGITAL

1. Qual é o objetivo?

O **objetivo final** da Tese é **gerar o código-fonte da interface** de uma aplicação **a partir de uma simples fotografia** de um esboço feito à mão.

Para isso, serão usados **algoritmos de Machine Learning** que conseguirão identificar os elementos desenhados à mão.

Vê, por exemplo, o resultado final para este esboço:



CURIOSIDADE
Para tornar um modelo de Object Detection eficaz, é necessário treiná-lo com um *dataset* que reúna a maior variedade de estilos possível.

2. Em que é que podes ajudar?

Os estilos de desenho variam significativamente de pessoa para pessoa.

Para tornar os algoritmos de Machine Learning eficazes, **é necessário treiná-los** para que reconheçam a **maior diversidade de estilos possível**.

Preciso da tua ajuda para reunir um *dataset* grande, **variado** e representativo dos vários estilos de desenho.

NOTA IMPORTANTE
A representação de cada elemento tem de ser inequívoca. Por isso, deve corresponder ao padrão de representação de elementos adotado.

3. Como deves proceder?

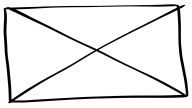
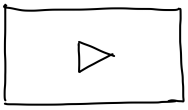
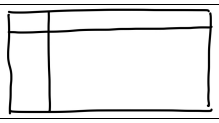
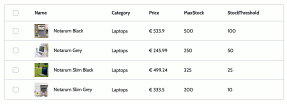
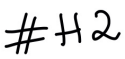
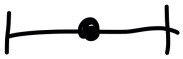
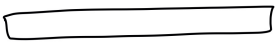
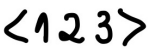
Só tens de **copiar 1 a 3 vezes cada uma das interfaces** para uma folha de papel e tirar uma fotografia. Para isso, deves:

1. **Desenhar** todas as "**Interfaces a desenhar**", das páginas 3 a 7, numa folha de papel, respeitando as representações dos elementos.
2. **Fotografar** com o telemóvel cada uma das interfaces que desenhaste, **sem cortar nenhum elemento** da interface.
3. **Enviar-me** as fotografias dos teus desenhos através do **Google Forms** disponível em **bit.ly/uigoncalo**. Podes submeter até 10 fotografias de cada vez.

INFORMAÇÃO
Se desenhares mais de uma interface por folha, certifica-te de que envias uma fotografia individual para cada.

Padrão de representação dos elementos

Por haver inúmeras formas de representar elementos, foi necessário definir um padrão de representação que **permite identificá-los facilmente**. De preferência, **usa cada elemento pelo menos uma vez** entre todos os teus esboços!

Elemento	Padrão de representação	Resultado final
Imagem		
Vídeo		
Gráfico		
Tabela		
Área de texto		Screens are great assets for anyone working in an office set up. They allow you to have more programs open, multitask and be more productive. Hopefully.
Checkbox		Option 1
Botão		Save Product
Título de nível 1		**Transcations Overview**
Título de nível 2		**Notarum Black 2**
Dropdown		Laptops
Texto		I noticed some erratic behavior.
Link		View in Store
Slider		60 - 425
Campo de texto		2020-12-18
Ícone		
Paginação		‹ 1 2 3 ›

CHECKLIST

Não esquecer...

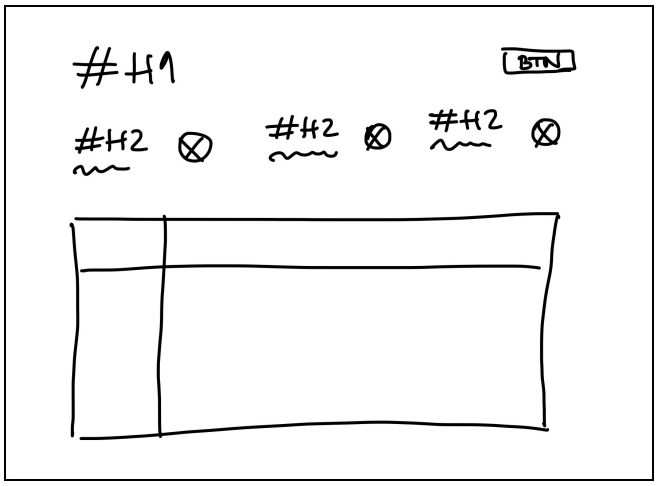
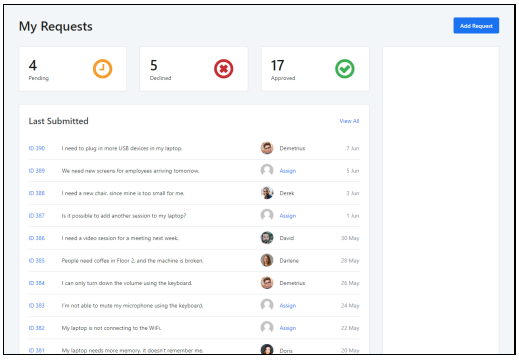
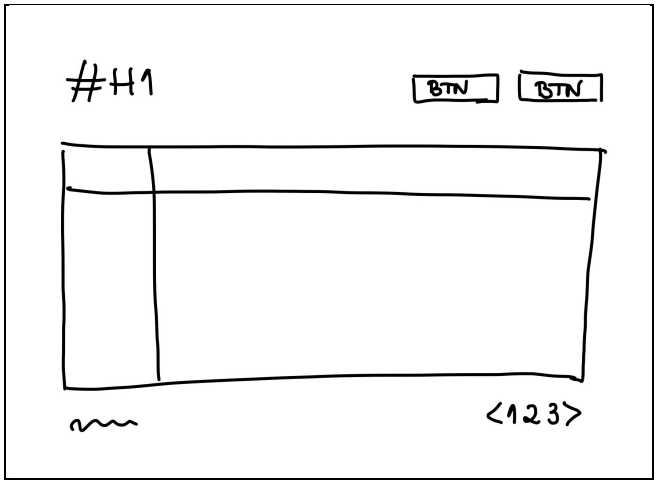
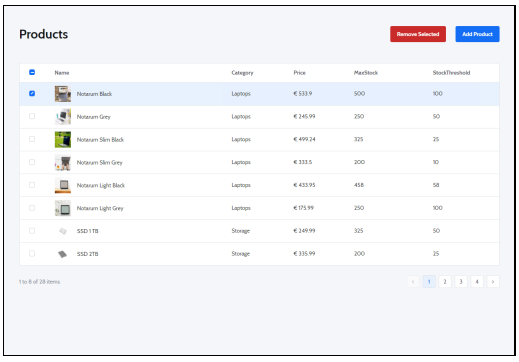
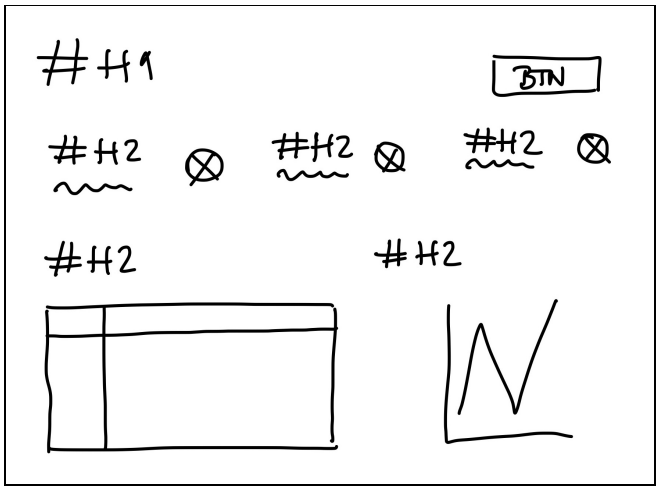
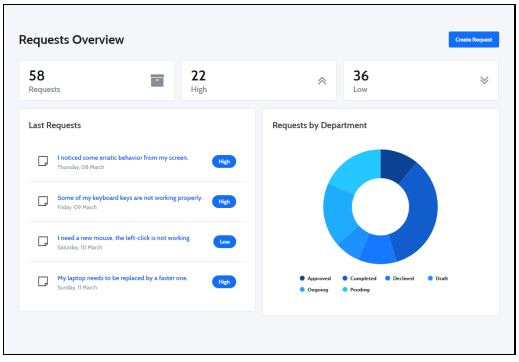
Podes usar esta *checklist* para verificares se estás a usar todos os elementos.

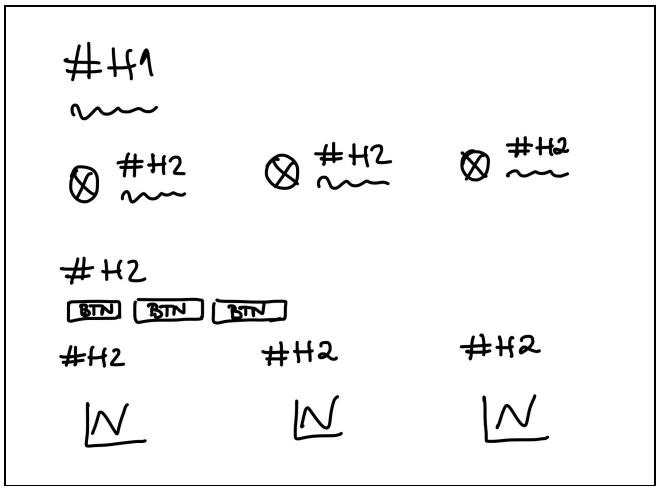
- ◯ Imagem
- ◯ Vídeo
- ◯ Gráfico
- ◯ Tabela
- ◯ Área de texto
- ◯ *Checkbox*
- ◯ Botão
- ◯ Título de nível 1
- ◯ Título de nível 2
- ◯ *Dropdown*
- ◯ Texto
- ◯ *Link*
- ◯ *Slider*
- ◯ Campo de texto
- ◯ Ícone
- ◯ Paginação

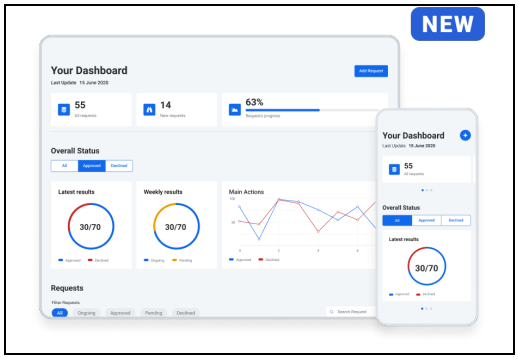
Interfaces a desenhar

A **framework OutSystems UI** inclui 17 **modelos de interfaces** prontos a usar nas aplicações. Estas miniaturas ilustram a representação desenhada de cada um desses modelos, respeitando os padrões de representação dos elementos incluídos em cada interface, conforme descrito na página anterior.

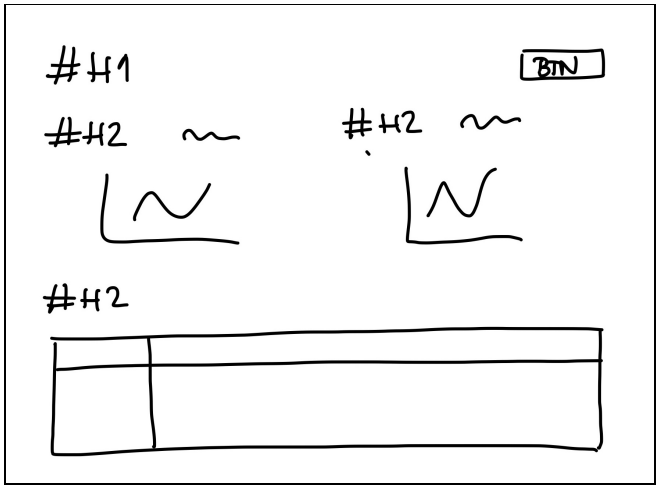
RECOMENDAÇÃO
Para **simplificar e acelerar** a tarefa de contribuição para o *dataset*, **a melhor forma de colaborar** é copiar para uma folha cada um destes exemplos, pelo menos, **uma vez**. Podes desenhar várias vezes por folha, mantendo a proporção.

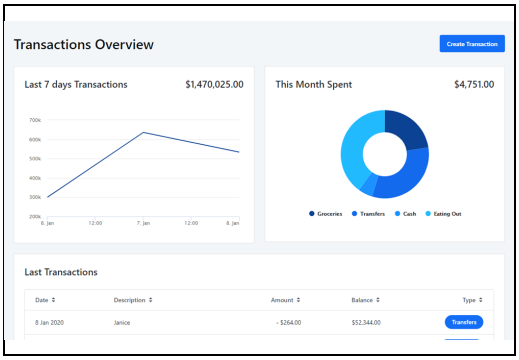
Representações desenhadas	Interfaces geradas
	 *Admin Dashboard*
	 *Bulk Actions*
	 *Dashboard*



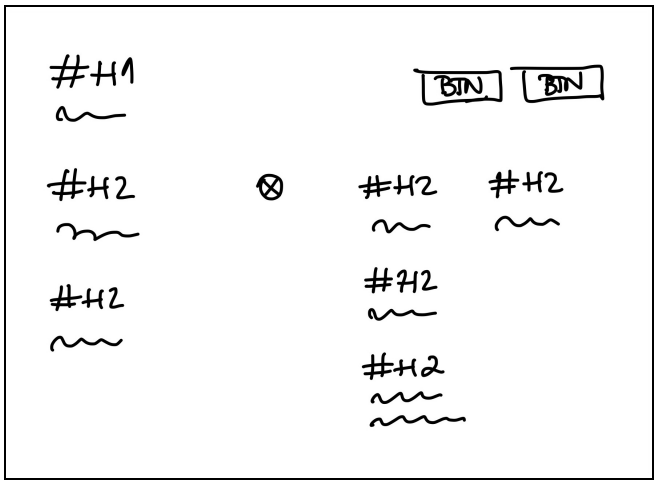


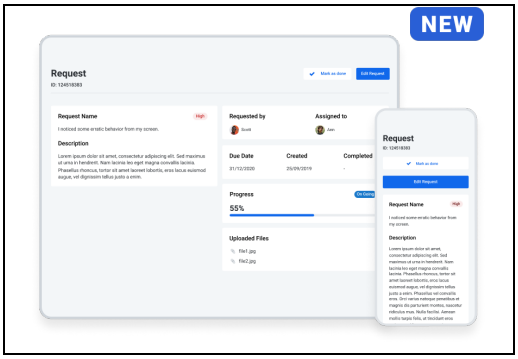
Requests Management



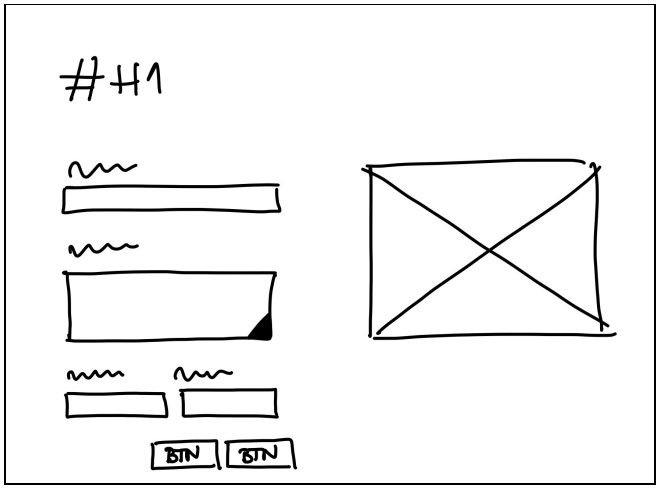


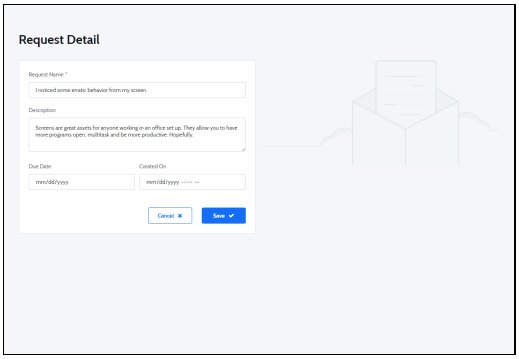
Transactions Dashboard



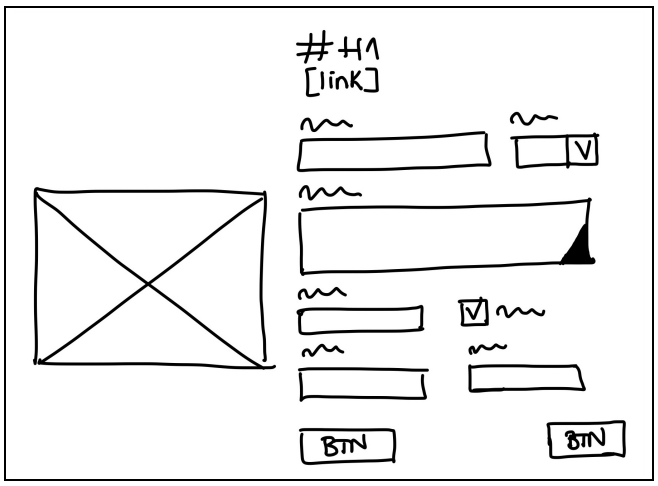


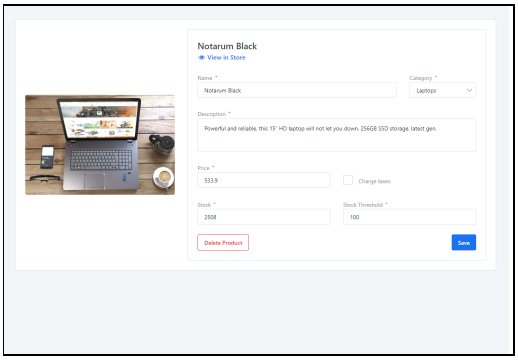
Request Detail



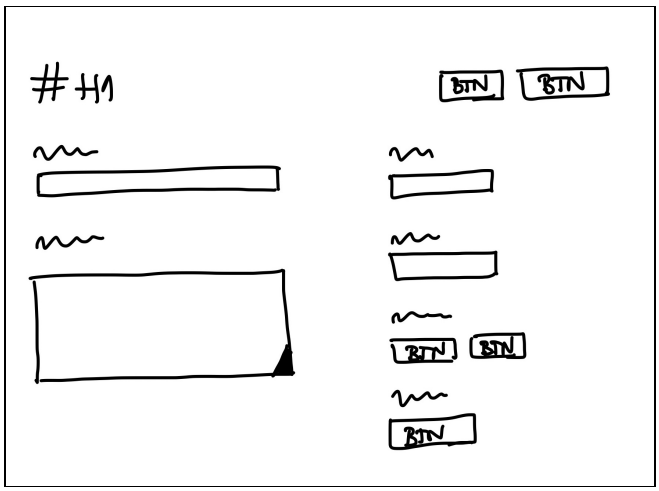


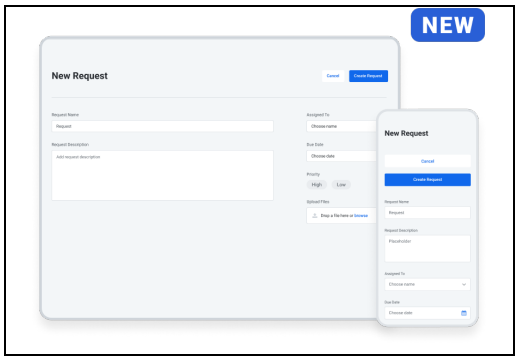
Detail



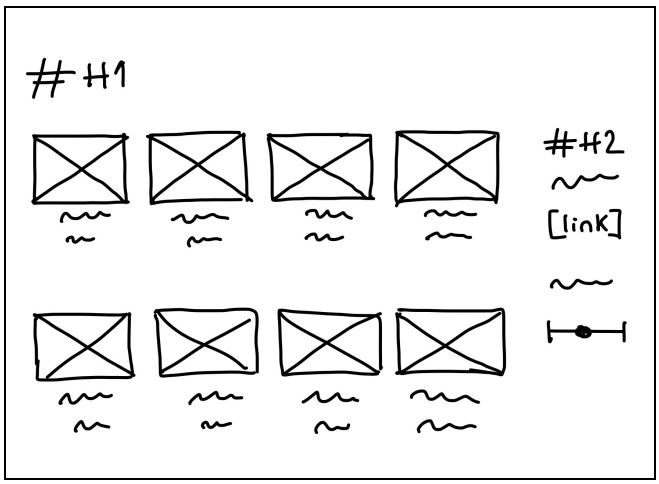


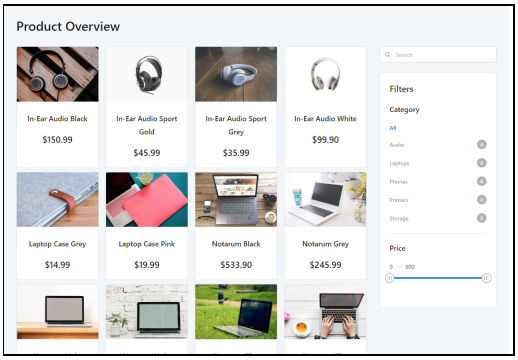
Product Detail



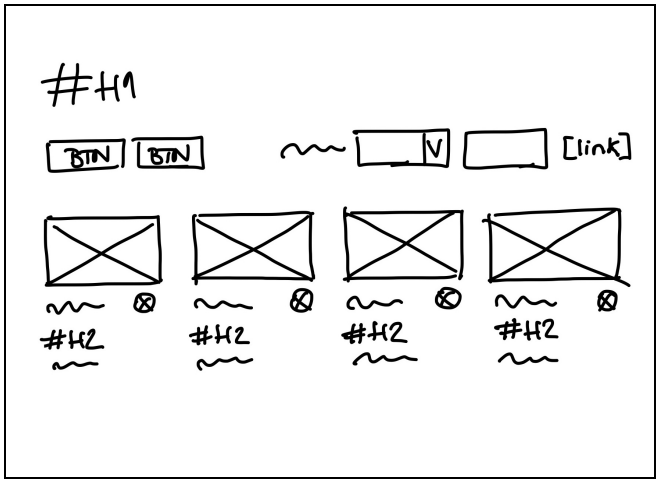


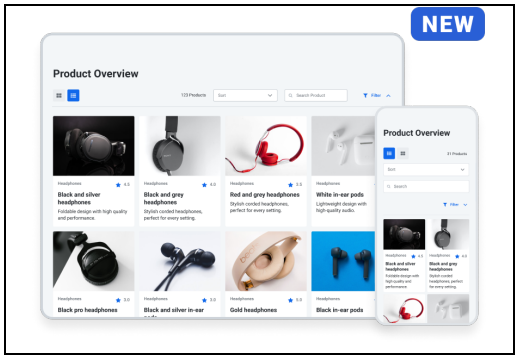
Request Creation



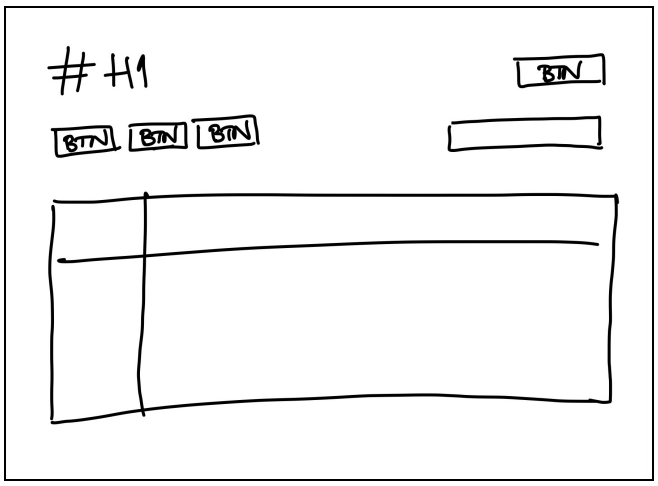


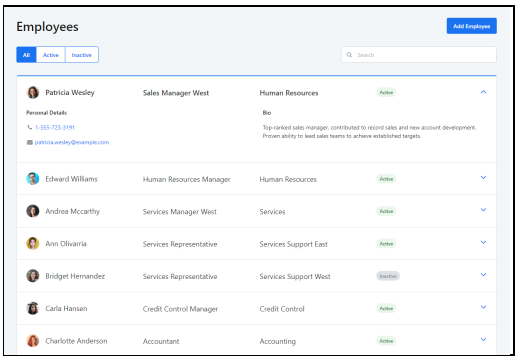
Four Column Gallery



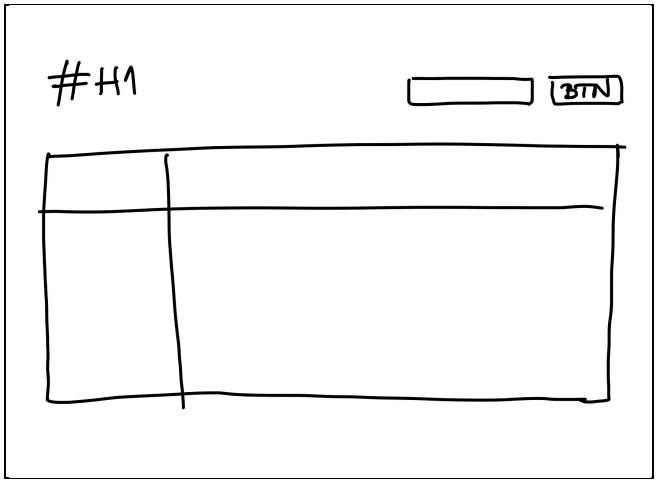


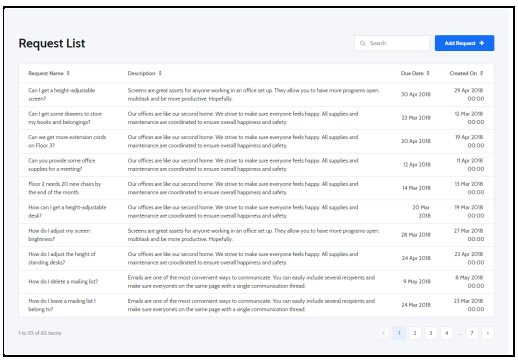
Product Catalog



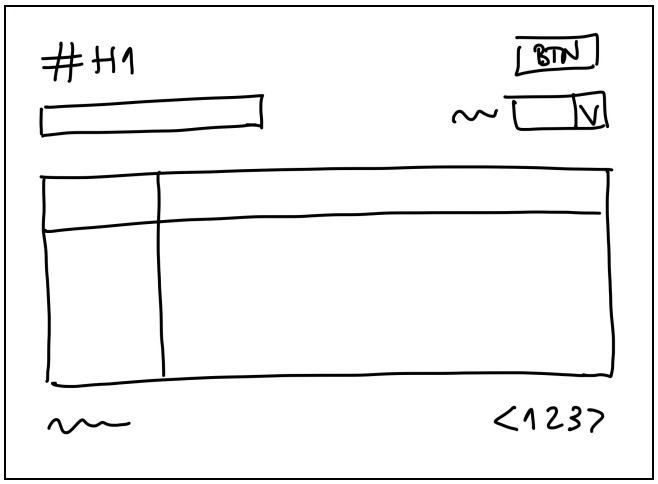


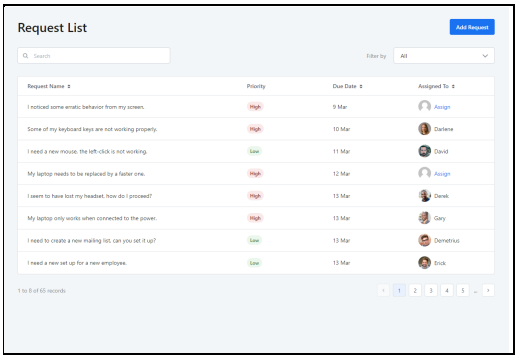
Horizontal Detail



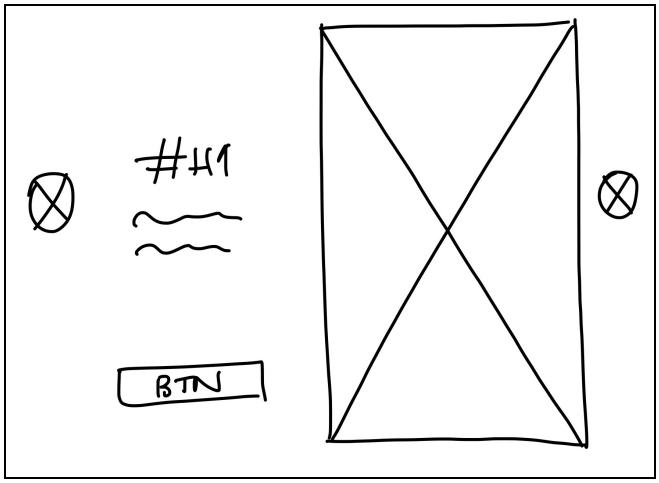


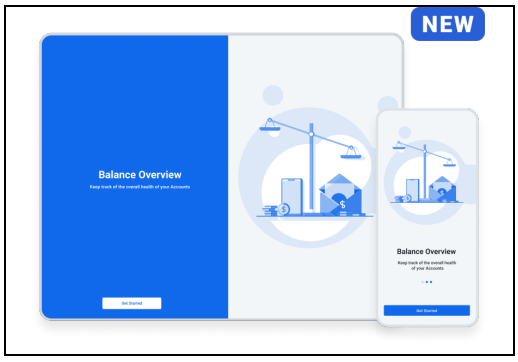
List



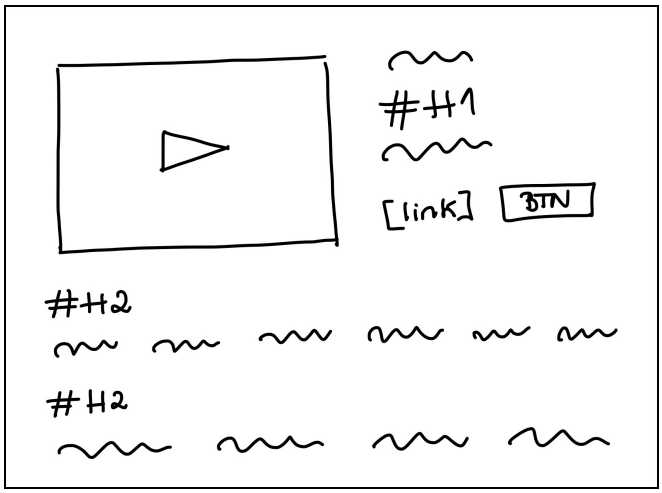


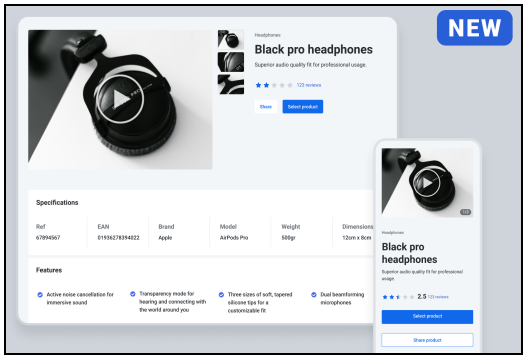
List with Filters





Onboarding Animation





Product Feature

Appendix B

Object Detection Results

This Appendix contains the results obtained from the YOLOv5-based object detection model described in Chapter 3. The tables included in this Appendix refer to different models, trained and testes with different train, test, and validation sets, according to the 5-fold approach followed. The presented values reflect the precision, recall and mAP score at different IoU thresholds (for scores greater than 0.5 and for scores from 0.5 to 0.95, i.e. `mAP@.5` and `mAP@.5:.95`).

Table B.1: Object detection results for fold 1 human-generated test set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	304	295	0.879	0.959	0.971	0.849
Video	304	111	0.962	0.228	0.588	0.378
Icon	304	241	0.944	1	0.995	0.631
Table	304	133	0.972	0.78	0.961	0.693
Input-text	304	213	0.651	0.958	0.806	0.581
Checkbox	304	123	1	0.806	0.992	0.709
Textarea	304	87	0.96	0.827	0.894	0.688
Button	304	439	0.976	1	0.994	0.766
Header-1	304	301	0.997	0.96	0.987	0.555
Header-2	304	463	0.96	1	0.984	0.617
Dropdown	304	171	0.901	0.428	0.604	0.412
Text	304	1217	0.983	0.999	0.996	0.616
Link	304	156	0.775	0.992	0.974	0.712
Slider	304	173	1	0.0115	0.403	0.227
Chart	304	159	0.813	0.962	0.975	0.768
Pagination	304	83	0.96	0.878	0.936	0.59
All Classes	304	4365	0.921	0.799	0.879	0.612

Table B.2: Object detection results for fold 1 human-generated validation set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	200	229	0.993	1	0.995	0.89
Video	200	10	0.946	0.9	0.966	0.778
Icon	200	188	0.996	1	0.995	0.622
Table	200	84	1	0.849	0.967	0.621
Input-text	200	166	0.971	0.997	0.989	0.76
Checkbox	200	12	0.937	1	0.995	0.649
Textarea	200	37	0.982	0.946	0.975	0.822
Button	200	359	0.996	0.989	0.989	0.74
Header-1	200	199	0.995	0.925	0.987	0.664
Header-2	200	368	0.92	1	0.983	0.635
Dropdown	200	35	0.985	1	0.995	0.821
Text	200	1068	0.889	0.998	0.996	0.588
Link	200	46	0.88	1	0.993	0.698
Slider	200	12	1	0.333	0.533	0.31
Chart	200	69	0.937	1	0.96	0.762
Pagination	200	24	0.973	1	0.995	0.708
All Classes	200	2906	0.962	0.934	0.957	0.692

Table B.3: Object detection results for fold 2 human-generated test set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	102	102	0.953	0.98	0.985	0.795
Video	102	7	0.731	0.714	0.797	0.5
Icon	102	101	0.939	0.97	0.977	0.622
Table	102	43	1	0.899	0.976	0.656
Input-text	102	84	0.881	1	0.992	0.752
Checkbox	102	10	1	0.999	0.995	0.674
Textarea	102	22	0.969	0.864	0.97	0.773
Button	102	168	0.993	1	0.996	0.764
Header-1	102	96	0.946	0.99	0.994	0.654
Header-2	102	183	0.94	0.967	0.984	0.599
Dropdown	102	16	0.915	1	0.995	0.767
Text	102	515	0.921	0.95	0.968	0.516
Link	102	21	0.836	0.973	0.989	0.686
Slider	102	7	0.418	0.119	0.394	0.301
Chart	102	49	0.906	0.959	0.969	0.714
Pagination	102	13	0.769	0.692	0.787	0.527
All Classes	102	1437	0.882	0.88	0.923	0.644

Table B.4: Object detection results for fold 2 human-generated validation set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	304	295	0.751	0.959	0.972	0.853
Video	304	111	0.876	0.192	0.388	0.232
Icon	304	241	0.866	0.996	0.995	0.64
Table	304	133	0.924	0.94	0.959	0.689
Input-text	304	213	0.503	0.986	0.832	0.617
Checkbox	304	123	1	0.342	0.937	0.582
Textarea	304	87	0.96	0.819	0.89	0.721
Button	304	439	0.94	0.998	0.993	0.774
Header-1	304	301	0.983	0.987	0.993	0.575
Header-2	304	463	0.947	0.994	0.995	0.636
Dropdown	304	171	1	0.5	0.661	0.494
Text	304	1217	0.95	0.998	0.995	0.653
Link	304	156	0.891	0.941	0.976	0.678
Slider	304	173	1	0.0148	0.301	0.181
Chart	304	159	0.858	0.951	0.983	0.789
Pagination	304	83	0.818	0.928	0.944	0.632
All Classes	304	4365	0.892	0.784	0.863	0.609

Table B.5: Object detection results for fold 3 human-generated test set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	193	225	0.957	0.99	0.993	0.848
Video	193	9	1	0.889	0.92	0.732
Icon	193	189	0.975	0.984	0.987	0.705
Table	193	79	1	0.808	0.963	0.712
Input-text	193	163	0.982	0.99	0.987	0.771
Checkbox	193	10	0.838	1	0.995	0.733
Textarea	193	38	1	0.92	0.945	0.778
Button	193	330	0.998	1	0.996	0.802
Header-1	193	193	0.964	0.964	0.991	0.691
Header-2	193	365	0.945	0.962	0.965	0.708
Dropdown	193	36	0.911	1	0.995	0.763
Text	193	1024	0.965	0.997	0.994	0.656
Link	193	45	0.906	0.978	0.975	0.733
Slider	193	11	0.836	1	0.988	0.804
Chart	193	70	0.966	0.986	0.994	0.74
Pagination	193	23	0.913	0.565	0.562	0.338
All Classes	193	2810	0.947	0.94	0.953	0.72

Table B.6: Object detection results for fold 3 human-generated validation set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	102	102	0.971	0.98	0.994	0.809
Video	102	7	0.794	0.857	0.856	0.611
Icon	102	101	0.965	0.941	0.981	0.569
Table	102	43	1	0.635	0.978	0.71
Input-text	102	84	0.984	0.976	0.987	0.746
Checkbox	102	10	0.74	1	0.995	0.727
Textarea	102	22	1	0.848	0.896	0.77
Button	102	168	0.995	0.994	0.995	0.778
Header-1	102	96	0.93	0.835	0.946	0.587
Header-2	102	183	0.994	0.954	0.987	0.648
Dropdown	102	16	0.895	1	0.995	0.758
Text	102	515	0.975	0.891	0.95	0.552
Link	102	21	0.841	0.952	0.972	0.712
Slider	102	7	0.621	1	0.718	0.534
Chart	102	49	0.985	0.939	0.973	0.749
Pagination	102	13	0.94	0.692	0.801	0.556
All Classes	102	1437	0.915	0.906	0.939	0.676

Table B.7: Object detection results for fold 4 human-generated test set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	204	241	1	0.962	0.994	0.827
Video	204	12	0.928	0.917	0.952	0.657
Icon	204	191	0.967	1	0.995	0.623
Table	204	83	0.978	0.88	0.986	0.692
Input-text	204	166	0.988	0.969	0.992	0.723
Checkbox	204	12	0.588	0.832	0.823	0.523
Textarea	204	36	0.977	1	0.995	0.809
Button	204	358	0.987	1	0.993	0.771
Header-1	204	206	0.985	0.976	0.98	0.623
Header-2	204	367	0.99	0.992	0.996	0.594
Dropdown	204	35	0.921	0.971	0.96	0.75
Text	204	1099	0.964	0.995	0.995	0.641
Link	204	49	0.92	0.98	0.976	0.713
Slider	204	11	0.829	0.636	0.789	0.495
Chart	204	72	1	0.989	0.995	0.784
Pagination	204	35	0.943	0.943	0.972	0.651
All Classes	204	2973	0.935	0.94	0.962	0.68

Table B.8: Object detection results for fold 4 human-generated validation set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	193	225	1	0.963	0.995	0.848
Video	193	9	0.983	1	0.995	0.682
Icon	193	189	0.971	0.974	0.992	0.727
Table	193	79	1	0.692	0.951	0.677
Input-text	193	163	0.988	0.982	0.986	0.766
Checkbox	193	10	0.939	1	0.995	0.738
Textarea	193	38	1	0.911	0.946	0.734
Button	193	330	0.997	1	0.996	0.773
Header-1	193	193	0.995	0.989	0.995	0.743
Header-2	193	365	0.97	0.967	0.981	0.711
Dropdown	193	36	0.987	1	0.995	0.812
Text	193	1024	0.988	0.993	0.992	0.658
Link	193	45	1	0.829	0.972	0.743
Slider	193	11	0.96	0.909	0.967	0.665
Chart	193	70	0.982	0.971	0.979	0.759
Pagination	193	23	0.864	0.652	0.7	0.38
All Classes	193	2810	0.977	0.927	0.965	0.713

Table B.9: Object detection results for fold 5 human-generated test set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	200	229	0.995	1	0.995	0.868
Video	200	10	0.97	0.8	0.813	0.653
Icon	200	188	1	1	0.995	0.671
Table	200	84	1	0.75	0.946	0.643
Input-text	200	166	0.994	0.986	0.99	0.747
Checkbox	200	12	0.964	1	0.995	0.536
Textarea	200	37	0.971	0.973	0.975	0.805
Button	200	359	1	0.988	0.989	0.778
Header-1	200	199	0.995	0.932	0.99	0.617
Header-2	200	368	0.948	0.999	0.992	0.64
Dropdown	200	35	0.991	0.971	0.993	0.795
Text	200	1068	0.944	0.998	0.996	0.636
Link	200	46	0.988	1	0.995	0.703
Slider	200	12	0.999	1	0.995	0.709
Chart	200	69	0.955	1	0.977	0.741
Pagination	200	24	0.991	1	0.995	0.591
All Classes	200	2906	0.982	0.962	0.977	0.696

Table B.10: Object detection results for fold 5 human-generated validation set.

Classes	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
Image	204	241	1	0.963	0.994	0.841
Video	204	12	0.992	1	0.995	0.821
Icon	204	191	0.986	0.995	0.991	0.569
Table	204	83	0.985	0.795	0.991	0.772
Input-text	204	166	0.994	0.971	0.995	0.779
Checkbox	204	12	0.881	0.917	0.925	0.621
Textarea	204	36	0.993	1	0.995	0.788
Button	204	358	0.988	1	0.989	0.778
Header-1	204	206	0.999	0.976	0.983	0.653
Header-2	204	367	0.988	0.995	0.996	0.623
Dropdown	204	35	0.97	0.937	0.983	0.773
Text	204	1099	0.987	0.995	0.996	0.626
Link	204	49	0.989	0.939	0.976	0.721
Slider	204	11	0.775	0.727	0.826	0.621
Chart	204	72	0.994	0.972	0.995	0.769
Pagination	204	35	1	0.96	0.99	0.692
All Classes	204	2973	0.97	0.946	0.976	0.715

Appendix C

Complete Pipeline Execution Example

This Appendix contains a complete pipeline execution example, presenting the inputs and outputs of all intermediate stages, hence demonstrating how this thesis integrates all stages seamless, even though different programming languages had to be used.

The diverse inner workings of each stage are also shown. This was accomplished by following a language-agnostic API rationale throughout the thesis, which will allow future research projects in this area to modify or easily upgrade any stage, without compromising the dataset produced or any methods developed.

Considering that the ultimate objective of this thesis is to convert a hand-drawn UI sketch straight into code, and eventually generate a real-world UI, observing Sections C.1 and C.7 illustrate the input and output of the implemented solution pipeline, respectively.

C.1 Original Hand-drawn Sketch Example

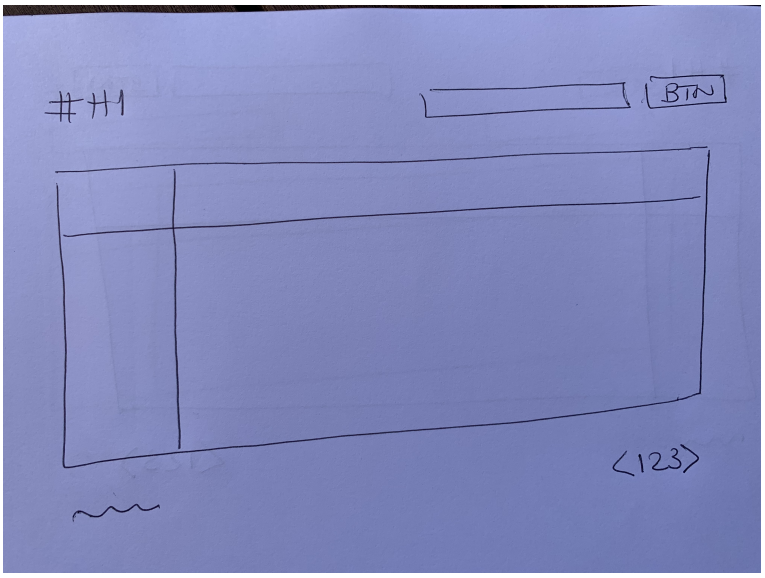


Figure C.1: Raw photo of a hand-drawn sketch taken with a smartphone camera (RGB color space, 4032 by 3024 pixels resolution, and JPEG file weighting 2.4 MB).

C.2 Pre-processed Example Result

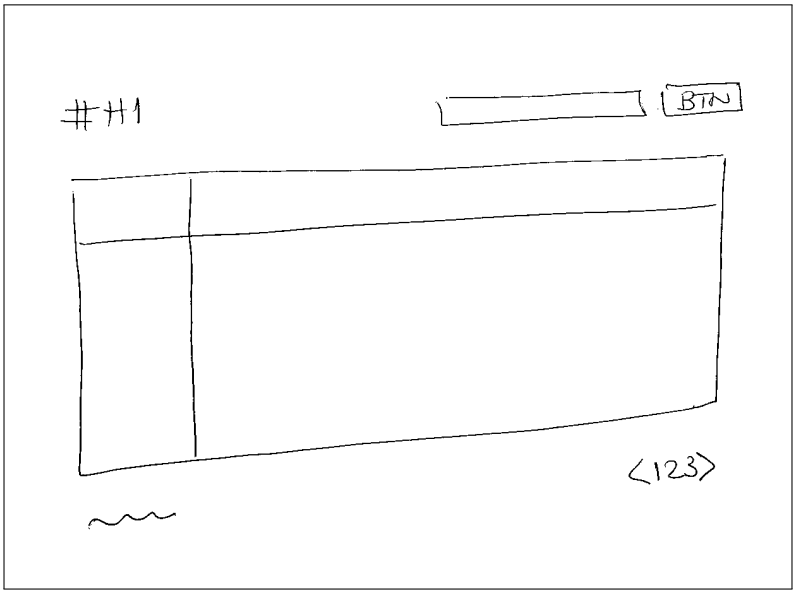
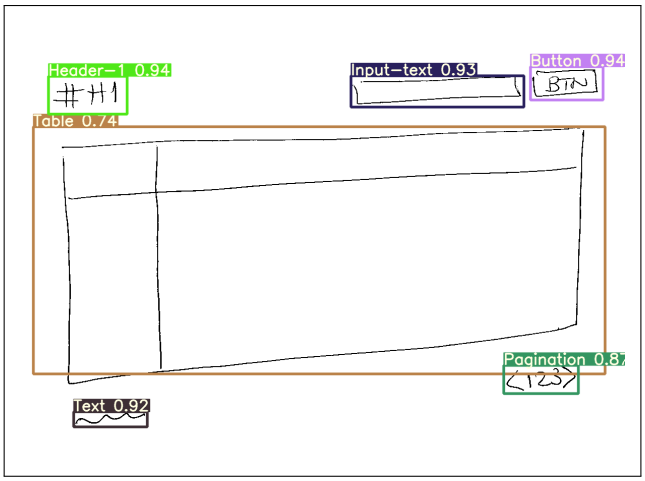


Figure C.2: Pre-processed image corresponding to the binarized, resized, and masked version of the original raw photo of a hand-drawn sketch taken with a smartphone camera (binary color space, 1200 by 900 pixels resolution, and PNG file weighting 7 KB).

C.3 Object Detection Example Result



(a)

```
8 0.112917 0.179444 0.129167 0.081111
4 0.686667 0.168333 0.288333 0.072222
7 0.901667 0.152222 0.118333 0.080000
3 0.502500 0.532778 0.875000 0.578889
15 0.862917 0.806111 0.120833 0.070000
11 0.149167 0.893333 0.123333 0.037778
```

(b)

Figure C.3: Pre-processed image corresponding to the binarized, resized, and masked version of the original raw photo of a hand-drawn sketch taken with a smartphone camera (binary color space, 1200 by 900 pixels resolution, and PNG file weighting 7 KB).

C.4 Spatial Grouping Example Result

```
export default {
  children: [
    {
      children: [
        {
          id: 0,
          top: 0,
          left: 0,
          width: 168.0,
          height: 40.0,
          type: "Header-1",
          value: "Header-1"
        },
        {
          children: [
            {
              id: 0,
              top: 0,
              left: 0,
              width: 179.0,
              height: 40.0,
              type: "Input-text",
              value: "Input-text"
            },
            {
              id: 0,
              top: 0,
              left: 0,
              width: 150.0,
              height: 40.0,
              type: "Button",
              value: "Button"
            }
          ],
          type: "generatedContainer",
          parsingDirection: "horizontal",
          top: 102,
          left: 49,
          width: 500,
          height: 102,
          id: "dd6ca2a5-98dd-4c3d-b148-e9ec77a2984b",
          passes: 0,
          section: true
        }
      ],
      type: "generatedContainer",
      parsingDirection: "horizontal",
      top: 102,
      left: 49,
      width: 500,
      height: 102,
      id: "dd6ca2a5-98dd-4c3d-b148-e9ec77a2984b",
      passes: 0,
      section: true
    },
    {
```

```
      id: 0,
      top: 0,
      left: 0,
      width: 1120.0,
      height: 532.0,
      type: "Table",
      value: "Table"
    },
    {
      children: [
        {
          id: 0,
          top: 0,
          left: 0,
          width: 228.0,
          height: 21.0,
          type: "Text",
          value: "Text"
        },
        {
          id: 0,
          top: 0,
          left: 0,
          width: 150.0,
          height: 40.0,
          type: "Pagination",
          value: "Pagination"
        }
      ],
      type: "generatedContainer",
      parsingDirection: "horizontal",
      top: 102,
      left: 49,
      width: 500,
      height: 102,
      id: "dd6ca2a5-98dd-4c3d-b148-e9ec77a2984b",
      passes: 0,
      section: true
    }
  ],
  type: "root",
  parsingDirection: "vertical",
  top: 0,
  left: 0,
  width: 1200,
  height: 900,
  id: "0",
  passes: 0
};
```

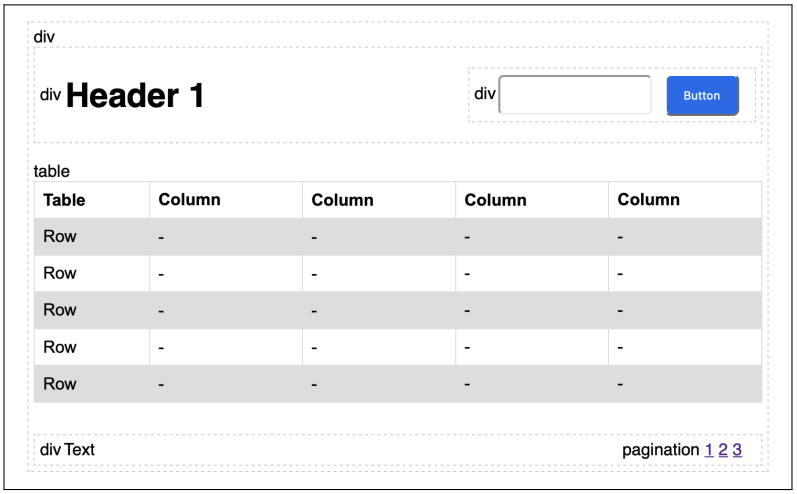
C.5 Agnostic UIDL Example Generation

```
{type: "element", content: Object}
 elementType: "div"

>style: Object (...)
 fontFamily: "'Cabin', sans-serif"
 flexDirection: "row"
 justifyContent: "space-between"
 alignItems: "center"
 padding: "5px"

>children: Array(3)
    >children: Array(2)
        >0: Object
         type: "element"
        >content: Object
         elementType: "h1"
        >style: Object (...)
         color: "black"
         width: "50px"

        >children: Array(2)
            >0: Object
             type: "element"
            >content: Object
             elementType: "input"
            >style: Object (...)
             height: "38px"
             borderRadius: "5px"

            >1: Object
             type: "element"
            >content: Object
             elementType: "button"
            >style: Object (...)
             height: "40"
             borderRadius: "6px"

    >children: Array(1)
        >0: Object
         type: "element"
        >content: Object
         elementType: "table"
        >style: Object (...)
         width: "100%"
         display: "block"

    >children: Array(2)
        >0: Object
         type: "static"
         content: "Text"

        >1: Object
         type: "element"
        >content: Object
         elementType: "pagination"
        >style: Object (...)
```

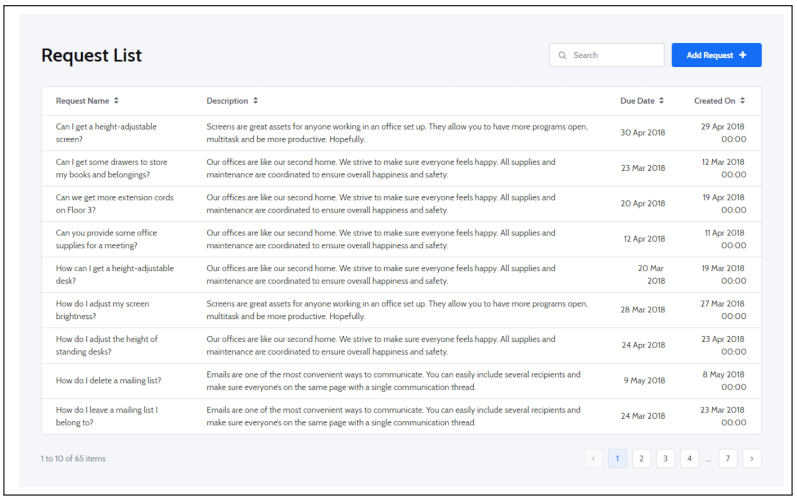
C.6 Code Generation Example Result

```
<div style="flex-direction: row; justify-content: space-between; align-items: center; border: 1px dashed rgb(204, 204, 204); outline-offset: -1px; padding: 5px;">
    div

    <div style="display: flex; flex-direction: row; justify-content: space-between; align-items: center; border: 1px dashed rgb(204, 204, 204); outline-offset: -1px; padding: 5px;">
        div

        <h1 style="color: black; width: 253px;">Header 1</h1>
        <div style="flex-direction: row; justify-content: space-between; align-items: center; border: 1px dashed rgb(204, 204, 204); outline-offset: -1px; padding: 5px;">
            div

            <input style="height: 38px; border-top-left-radius: 5px; border-top-right-radius: 5px; border-bottom-right-radius: 5px; border-bottom-left-radius: 5px;"></input>
            <button style="color: white; background-color: rgb(16, 104, 235); cursor: pointer; line-height: 1; height: 40px; display: inline-flex; margin: 0px 10px; padding: 0px 16px; border-top-left-radius: 6px; border-top-right-radius: 6px; border-bottom-right-radius: 6px; border-bottom-left-radius: 6px;">Button</button>
        </div>
    </div>

    table
    <table style="border-collapse: collapse; width: 100%;">
        <style>td, th { border: 1px solid #dddddd; text-align: left; padding: 8px; } tr:nth-child(even) { background-color: #dddddd; }</style>
        <tr>
            <th>Table</th>
            <th>Column</th> (...)
        </tr>
        <tr>
            <td>Row</td>
            <td>-</td> (...)
        </tr>
        (...)
    </table>

    <div style="display: flex; flex-direction: row; justify-content: space-between; align-items: center; border: 1px dashed rgb(204, 204, 204); outline-offset: -1px; padding: 5px;">
        div
        <span>Text</span>
        <div>
            pagination
            <a>1</a>
            <a>2</a>
            <a>3</a>
        </div>
    </div>
</div>
```

C.7 Web App Example Result



(a)



(b)

Figure C.4: Comparison of a (a) screenshot of a React web-generated app, with the detected containers marked with a dotted line and generic placeholders inside each UI element, with an (b) image of a screen template using the OutSystems UI Framework CSS that matches the same UI elements and page structure.