Automatic User Interface Generation

Gonçalo Correia de Matos
goncalocmatos@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisbon, Portugal

June 2022

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, Image Analysis, Computer Vision, Automatic Program Generation, User Interface

1. Introduction

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.

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.

This new field of research has been propelling the creation of new companies, such as Uizard, and motivating new research and development projects in different companies, like teleportHQ, Microsoft, and Airbnb.

This thesis proposes a tool that converts hand-drawn sketches into real world user interfaces. 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.

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, making the tool more versatile and introducing important advancements across the board.

2. Background

2.1. Object Detection

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. The basis of de design of conventional CNNs was first introduced by LeCun *et al.* [4] to tackle the challenges posed by computer vision. More particularly, CNNs were created to solve a handwritten digit recognition problem, known as image classification. 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 [2].

One 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, as it avoids several interdependent stages.

2.2. You Only Look Once

The You Only Look Once (YOLO) algorithm [5] 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. Over time, YOLO has had several iterations and became faster and more reliable. Currently, there are five main versions of YOLO, from YOLOv1 to 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 were high.

YOLOv5 [3] 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, 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.

3. Implementation

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.

3.1. Screen Templates and User Interface Elements

The OutSystems UI framework includes a wide variety of adaptive and interactive UI elements that compose screen templates. 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.

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.

3.2. 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 and a smaller dataset using the standard representations proposed by teleportHQ, depicted in Figure 1.

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 sketches multiple times.

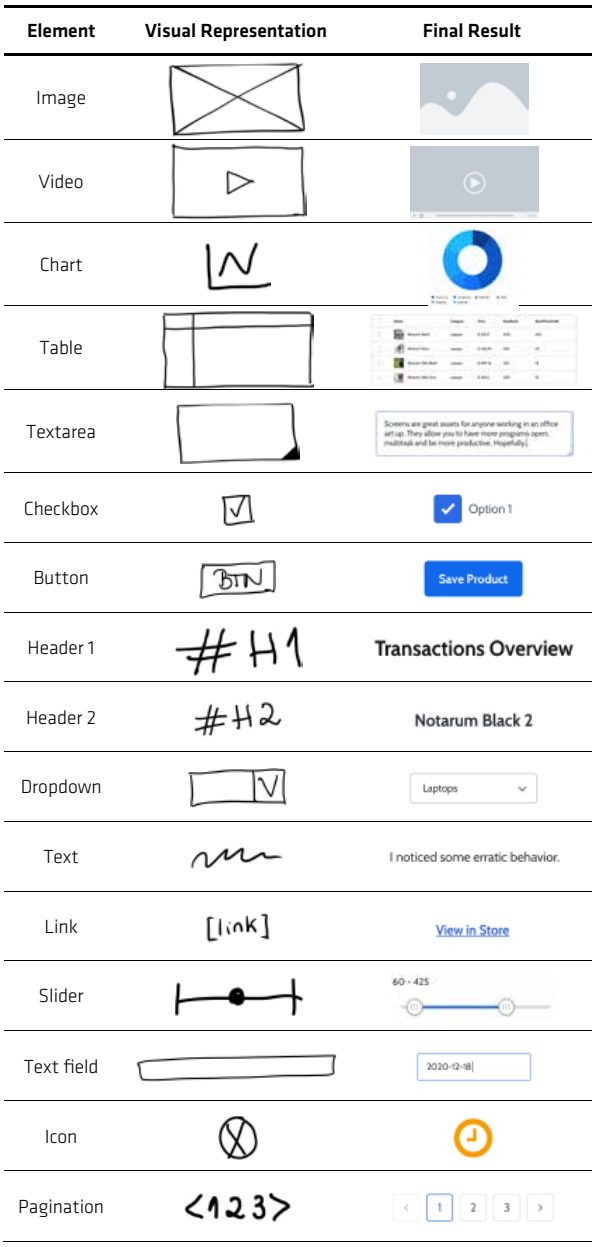


Figure 1: List of hand-drawn representations supported by our object detection model.

3.3. Human-generated Dataset

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 [1].

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.

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, in an attempt to remove noise and preserve a pure white background where the hand-drawn lines are black. 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.

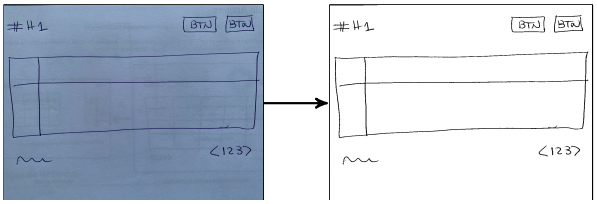


Figure 2: Resizing and binarization pre-processing results (photo file renamed from `IMG_1405.jpeg` to `BATCH_000002-USER_000005-SKETCH_000006.png`.

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 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.

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.

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`).

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 [6]. This tool provided an important insights 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, shown in Figure 3.

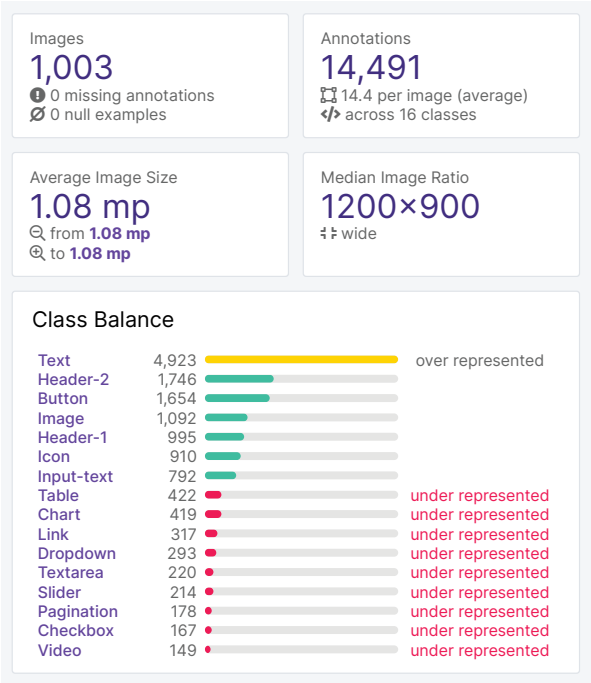


Figure 3: Illustration of the proposed approach for computer-generated sketches, where the elements of the *Admin Dashboard* screen template are replaced with their respective hand-drawn representation.

3.4. 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 with slight contextual adjustments.

The general idea of this approach, shown in Figure 4, is to take advantage of the cumbersome labeling task, by replacing the elements in the UI with their correct hand-drawn representation.

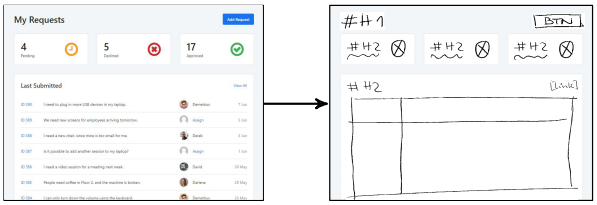


Figure 4: Illustration of the proposed approach for computer-generated sketches, where the elements of the *Admin Dashboard* screen template are replaced with their respective hand-drawn representation.

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 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.

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.

3.5. Object Detection Model

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.

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.

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.

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.

All in all, our model was 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 path `../data.yaml`.
- `--cfg`: configuration of our model described in a YAML model configuration file.
- `--name`: model name to be displayed and eventually saved.

3.6. 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, 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.

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.

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.

3.7. 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.

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

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`.

4. Results

This section 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. 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 [8].

A suitable number of training epochs was chosen to train our model with a custom dataset without exceeding Google Colab's usage quotas.

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. Each fold took 27 to 39 minutes to train.

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.

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).

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.

4.2. 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.

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 7 is representative of the element-wise performance of the trained networks. 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.

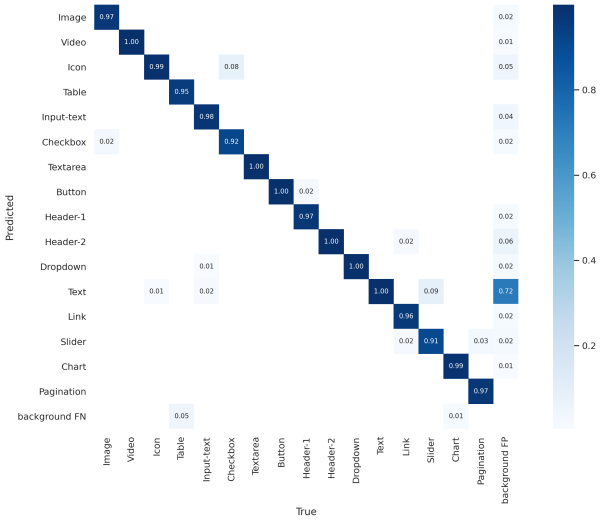
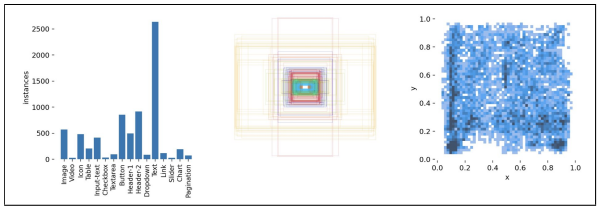
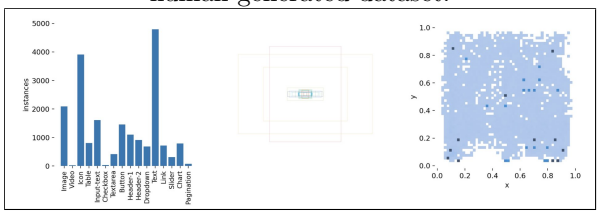


Figure 5: Confusion matrix for all 16 classes and background false negatives of fold 1.



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

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.

Table 1 presents the results for the 5-fold approach followed for the human-generated dataset. This table presents the accuracy results for the test set.

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) 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*).

4.3. 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.

Figure 6 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 and some classes are naturally more present in real-world user interfaces, sometimes appearing repeatedly in an application.

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 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.

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.

Table 1: 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

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.

The box, objectness and classification losses of the validation data also showed a rapid decline until around epoch 200.

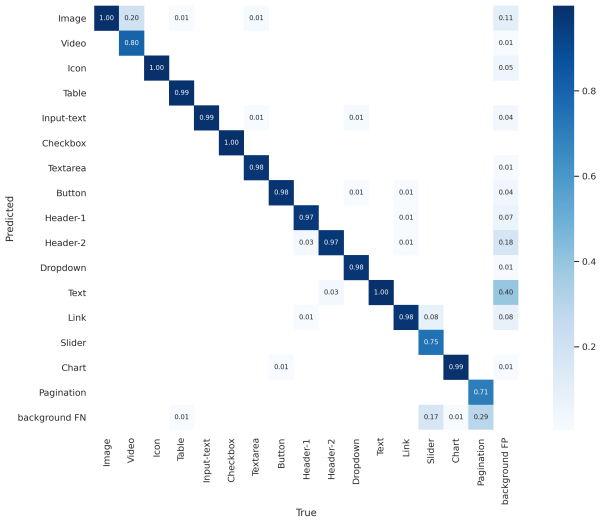


Figure 7: Confusion matrix for the model trained with a computer-generated dataset and tested with the human-generated dataset.

Table 2 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.

4.4. 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.

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 Figure 8 this was done by comparing the ground-truth labels printed onto the input images and the predicted labels.

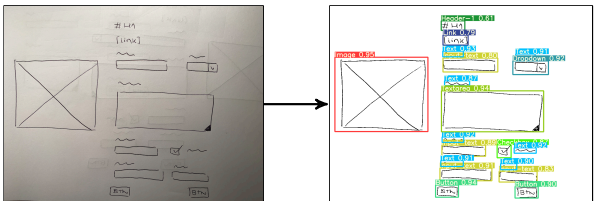


Figure 8: Example of an accurate detection performed by the model.

The qualitative analysis of our object detection results show 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.

Table 2: 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

This low confidence score is due to the aspect ratio variability of some elements in hand-drawn sketches that 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. The example shown in Figure 9 presents a lower confidence score for the *Image* element, but it is still high enough to surpass the 0.5 threshold.

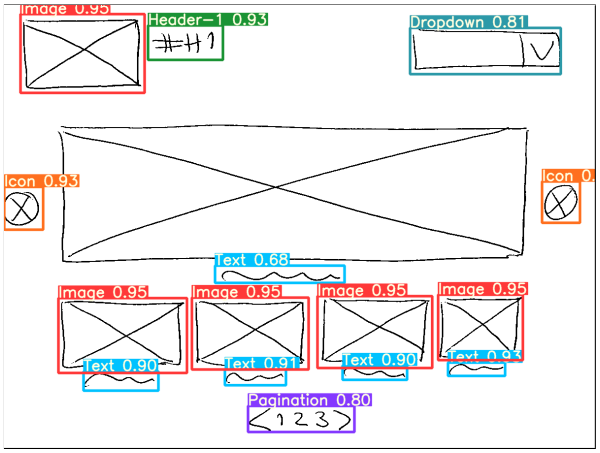


Figure 9: Example of an unusually wide *Image* element that is not above the 0.5 threshold.

Figure 10 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.

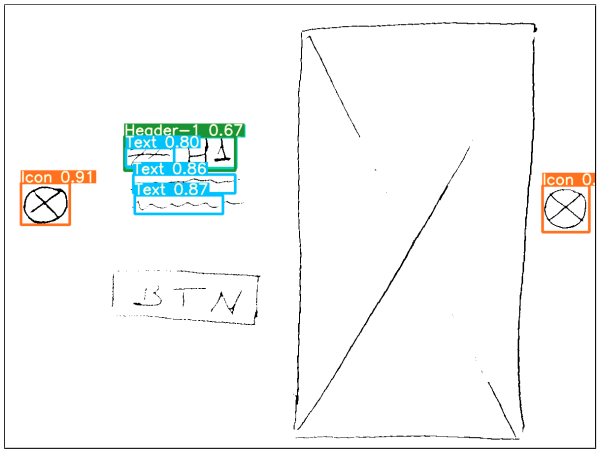


Figure 10: Example of two missing detections of a *Button* and an *Image* due to pre-processing issues.

This issue is due to the discrepancy of light, brilliance, shades, and saturation across different users' photos. In this specific case, the the original photos was 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.

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 [7], but these techniques were only successful in a limited small group of images that did not present too much noise.

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.

We propose 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.

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.

Automatic UI generation has been building momentum in software development and continues to drive start-ups focused in this field of research. However, it is far from mature. 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; and implementing an optical character recognition system to avoid static placeholders (e.g., buttons action text, search field suggestion placeholders, text labels, headers, etc.).

Acknowledgements

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.

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. A special thanks goes out to my family, friends, and colleagues who have offered their support and confidence during this journey.

References

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

[2] 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.

[3] 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.

[4] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[5] 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.

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

[7] H. M. Sharifipour, B. Yousefi, and X. P. V. Maldague. Skeletonization and reconstruction based on graph morphological transformations. *CoRR*, abs/2009.07970, 2020.

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