Towards More Efficient Mobile UI Design: Automatic Code Generation from Hand-Drawn Sketches Using Deep Learning

Daniel Leivers

Master of Science in Artificial Intelligence
The University of Bath
2023

Towards More Efficient Mobile UI Design: Automatic Code Generation from Hand-Drawn Sketches Using Deep Learning

Submitted by: Daniel Leivers

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

In this dissertation we describe the process of designing and building an application to support the creation of mobile application user interface (UI) prototypes. While paper prototypes have generally been the most popular choice for the early stages of designing an app, it still takes time to implement those prototypes in code, we aim to speed up this process by generating the code as a user draws their prototype.

To solve this problem we developed an iPad app which allows a user to draw using the Apple Pencil. This presents us the opportunity to apply deep learning techniques on the device to recognise what the user has drawn, providing the user with near real-time feedback. In this dissertation we discuss the design, development, user testing and evaluation of the resulting app.

A video demonstration of the app is available on YouTube `https://www.youtube.com/watch?v=SKGdZ3H9eyY` and the source code can be accessed on GitHub `https://github.com/Otaku-Development/msc-final-project`.

Contents

List of Figures

List of Tables

Acknowledgements

This dissertation marks the culmination of an important academic journey, and I owe my gratitude to several key individuals for their support.

First, I extend my sincere appreciation to my supervisor, Dr. Zack Lyons, for their valuable feedback and encouragement throughout this process. Your guidance has been instrumental in the development of this work.

I also want to express my heartfelt thanks to my partner, Emma, for her understanding and unwavering support. Her diligent proofreading and dedicated care for our little boy, Ben, have been indispensable.

Chapter 1

Introduction

Good design and user experience (UX) often separates a successful mobile application (app) from an unsuccessful app. Since the Apple App Store launched in 2008, businesses compete to sell their software to users and the look and feel of an app often becomes the differentiating factor for a user choosing a particular app. Apple host yearly app design awards (Apple, 2023a) to reward well built apps and, given the amount of money that can be made on the App Store, businesses are willing to invest large sums in an attractive user interface (UI).

The process of designing an app is highly iterative (Figure 1.1), moving through paper based sketches, wireframes, higher fidelity mock ups and prototypes (both programmatic and paper) before achieving the final design with user testing occurring at multiple stages of the process. This is a lengthy process and may be repeated multiple times to evolve the UX and UI. Time restrictions sometimes means that user testing isn't undertaken for all design iterations and feedback from (potential) users can be missed.

Typically, designers use a design tool such as Figma, Sketch or Photoshop to produce a design document for developers to reference when implementing the app. The developers on Apple platforms generally use Xcode and its built in tools to create a functional implementation of the design. Apple's current recommendation for implementing user interfaces on iOS (and iPadOS and macOS) apps is SwiftUI.

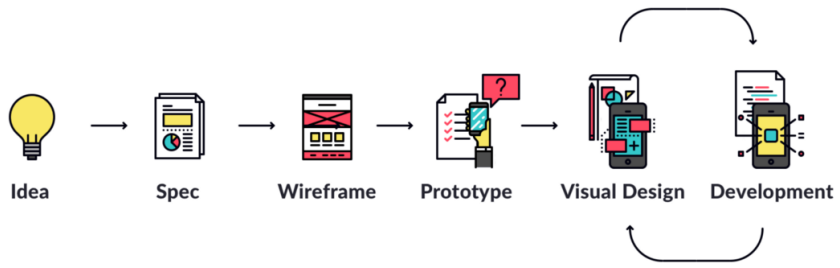


Figure 1.1: The app design process in broad strokes (Flarup, 2016)

Designing and implementing a mobile UX and UI is complex and time-consuming and requires a high level of expertise in design and coding. End-users who do not have coding expertise

often struggle to translate their UI designs into code, which can result in miscommunication and errors in the development process.

The cost of hiring the specialist designer skills required to build mobile apps is high, as shown by the salary information gathered by StackOverflow (2023), and creating an app is expensive (Dogtiev, 2023). Given that much of the development time for a mobile app is spent on coding the UI from a set of designs it's clear that there is potential for a large time and cost saving by reducing the manual work in this area (King, 2023).

This problem is particularly relevant for small businesses and individuals who cannot afford to hire a dedicated designer or developer and start ups who need to move quickly to prove their business. There is a clear need for an automated solution that enables users to create UI code from designs without requiring any coding knowledge and developers to translate UI designs into code more quickly.

Furthermore, Flora, Wang and Chande (2014) found that 50% of participants in their survey considered creating prototypes to be a top priority for creating "an excellent user experience" and making an app more successful.

Enabling the rapid development of UI code would vastly increase the speed at which usable prototypes can be created, speeding up the design iteration cycle (Figure 1.2) and reducing the time for feedback from potential users.

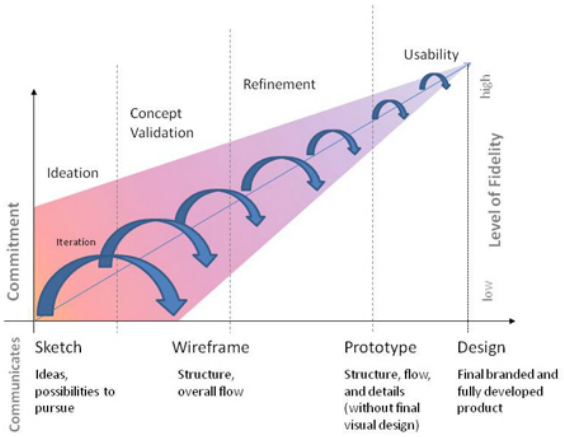


Figure 1.2: The sketch to design continuum (Lepore, 2010)

To address this problem, this project aims to develop an Artificial Intelligence (AI) system that can automatically generate SwiftUI code from hand-drawn sketches of mobile UI designs. The system will allow end-users to create sketches using an iPad app which then generates SwiftUI code that can be used (at least as a starting point) to implement their designs in a mobile app.

1.1 High fidelity vs. low fidelity

Each stage of the app development lifecycle requires different levels of designs. In the early stages of a project, rough or unpolished designs are created for use in ideation or prototyping (Babich, 2023). These are often referred to as "low fidelity" designs. Typically these designs are low in detail, black and white and created using pen and paper or digitally. The benefit of these low fidelity designs is that they are quick, easy and low cost to create, however they are

not always suitable for making design decisions as they lack enough detail to allow stakeholders to fully envision the final app (UXPin, 2021).

As a project progresses designs are refined further. Colour is added along with greater detail such as animations to create "high fidelity" designs. High fidelity designs are created digitally and are ideal for sharing with stakeholders as they aim to closely represent the final app design, however they are more expensive and time consuming to create than lower fidelity versions (UXPin, 2021).

1.2 SwiftUI

SwiftUI is a modern UI framework introduced by Apple in 2019 (Apple, 2023k) that enables developers to build user interfaces using a declarative syntax. SwiftUI offers several advantages over traditional UI frameworks, including improved code maintainability, ease of use, and a live preview feature that allows developers to see their changes inline in the Xcode editor. This makes SwiftUI a good choice for this project, as we can replicate the preview to show the generated code inline in our app and ensure that it accurately reflects the intended UI design.

SwiftUI is a subset of Swift, specifically aimed at declaring a user interface. While Swift can be compiled on other platforms such as Linux, various Apple frameworks such as UIKit or SwiftUI are only available on Apple platforms and as such can only be compiled against or linked to on these platforms.

1.3 Deep learning

Recent advances in deep learning have led to significant progress in automatic code generation from UI design sketches. For example, Beltramelli (2017) proposed a system that can generate code from hand-drawn sketches of UI designs using a combination of convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Similarly, Baulé et al. (2021) presented a deep learning-based approach for generating code from sketches of mobile apps created by end-users. Aşıroğlu et al. (2019) also propose an approach using CNNs for generating HTML from hand drawn mock up images and there are further examples that use higher fidelity designs like Bajammal, Mazinanian and Mesbah (2018).

These papers demonstrate the feasibility and potential of using deep learning techniques for automatic code generation from sketches of UI designs.

1.4 What are we doing differently?

Baulé et al. (2020) review the current state of code generation from images. Their research shows that iOS is an under-served area and that there doesn't seem to have been any efforts to "cut out the middleman" and draw designs directly on a device.

Baulé et al. (2021) focus on generating UI for a tool called App Inventor (MIT, 2022) which generates Android apps. However, the tool only recognises limited layouts and UI components.

Our project aims to allows users to generate UI code directly for the platform they are using (iOS) and to explore the recognition of additional layouts and components if possible. By

choosing SwiftUI we are outputting designs using the current industry standard format for iOS app development, rather than creating another intermediate tool.

Many previous approaches involve digitising a pen and paper sketch to generate code on a computer. Our proposed approach saves time for users by skipping the need to capture their paper sketches using a camera and providing immediate feedback on the generated code as it gets rendered inline. In addition, this approach potentially provides routes for further exploration such as running the AI model on device, generating UI during the drawing process or attempting to augment the user's drawing skills with suggestions.

1.5 Research question

How does the use of an iPad app for sketching mobile designs, which automatically generates code, impact the efficiency of rapid prototyping in UI/UX design, and in what ways does this method offer advancements over conventional prototyping techniques?

1.6 Problem conclusion

In this project, we will focus on developing an AI system that can generate SwiftUI code from hand-drawn (low fidelity) sketches of mobile UI designs in an effort to improve the efficiency of the prototyping phase of app development. We will use deep learning techniques to extract meaningful features from the sketches and generate corresponding code. As part of this we will develop an iPad app that allows end-users to create hand-drawn sketches of mobile UI designs using the Apple Pencil. These sketches will be used to generate SwiftUI code which is then presented back to the user as a rendered UI.

Chapter 2

Literature and Technology Survey

2.1 Introduction

We are not the first to notice that there is an opportunity to reduce the time taken to produce UI code from designs using machine learning. In recent years there have been several papers demonstrating approaches to this idea, but there are many variations on the details and the implementations.

Some approaches use hand drawn, paper-based sketches to generate HTML or App Inventor (MIT, 2022) code, others use high fidelity designs or app screenshots to generate Android code or a Domain Specific Language (DSL) which is then translated in to platform specific code.

However, few of these approaches utilise iOS, none have attempted SwiftUI and none allow sketching on the device that's being targeted (i.e. drawing on an iPad to produce code the iPad is able to run).

The approaches can be generally split into two broad stages; object detection and classification (for recognising where UI elements are and what classification they are), and code generation (where the output of the previous step is translated into code).

Note that there are multiple approaches named "Sketch2Code".

2.2 Mobile app design

The design for a mobile app is generally built up of UI components provided by the platform. Apple provides the Human Interface Guidelines (Apple, 2023e) which describe the basic components and how to use them in a consistent fashion.

Common UI elements include:

- image views
- (text) labels
- text fields
- (toggle) switches
- buttons

- progress bars
- activity indicators
- page control
- picker view

These atomic UI components are grouped in container views or layout hierarchies such as a vertical or horizontal layout, or a list in order to build up a screen design. It is these basic UI components and layouts that we will look to recognise.

2.3 Sketch input

The vast majority of papers approaching a similar problem of detecting UI from user sketches do so using paper-based sketches. The idea being that the user draws the UI using pen and paper, captures it (by camera) and uploads it to a tool on their computer. Carter and Hundhausen (2010) found the main mediums used for prototypes are art supplies because of the speed of creation and ease of creation/use, but they also cite the disadvantages of this approach as the lack of user interaction, the significant difference from the end result and the difficulty of digitising.

Most existing approaches to converting paper sketches to code suffer from some of these disadvantages and almost all require the user to somehow digitise the sketches (Carter and Hundhausen, 2010). Having to change mediums inevitably introduces a delay as a user must complete their drawing, take a picture of it and upload it to a system to generate the code. If the system gets something wrong their options are to either redo the work or edit the code. We suspect that this also introduces friction for the users as it is an additional step.

Few have deviated from this approach but one that deserves a mention is Doodle2App (Mohian and Csallner, 2020) who acknowledge that integrating freehand sketching directly into the system would bridge a gap in software development prototyping. As such, they allow users to sketch using the computer mouse as the input. While this brings everything together into one tool and enables users to get instant results, it is hard to see UI designers wanting to wrangle a mouse for their drawings. Wimmer, Untertrifaller and Grechenig (2020) recognise this issue of sketch input and feedback and their approach attempts to bring some sense of real-time feedback to the system by using a computer and USB webcam setup pointed at a white board. Drawings are sketched and the system detects changes (as long as they are visible to the webcam). However, this set up is a little cumbersome and they acknowledge that a graphics tablet and stylus may be a feasible area for future research.

2.4 Approaches for object detection and classification

In this section we cover both object detection (locating an object in an input image) and object classification (determining what UI component an object is). We have approached this by examining other authors approaches individually as they often vary in implementation enough to be of note, sometimes the process of object detection and object classification is not distinct and are part and parcel of the same process, sometimes they're not.

Various machine learning approaches for detecting UI elements and their spacial information

(layouts) have been attempted. Generally, object detection in this context means that when an image contains one or more elements, the system is able to reason about where those objects are, what is their position and their width and height.

Conversely, the object classification process consists of a system being presented with an image where the object to be classified fills the bounds of that image and there is no positional information as it is the only thing that exists in that image. The system then attempts to discern a class for the image from a known list of possible classifications.

REMAUI, a tool produced by Nguyen and Csallner (2015), aims to reverse engineer mobile app (Android) UI from a screenshot or a high fidelity design. They use a combination of Optical Character Recognition (OCR) via Tesseract, Computer Vision (CV) using OpenCV and merge the results, then then apply various heuristics to infer user interface elements and their layouts. This approach seems to handle layouts well, and is one of the few approaches that attempts to recognise lists, however the system only really deals with limited UI elements (image and text elements). Similarly Nikam et al. (2021) also aim to interpret screenshots in to code. They limit the UI elements they are training their model on to a mere four classes (text, text input, image, button) and use various preprocessing methods to detect components before using a Convolutional Neural Network (CNN) for classification.

Using deep learning methods, in particular a Convolutional Neural Network (CNN), is a popular approach especially for object detection. Aşıroğlu et al. (2019) take this approach in order to detect objects in sketches of web pages and output HTML. They pre-process the input image (greyscale and Gaussian Blur), before applying morphological transformations so as to bound objects in the image, crop the detected objects to their bounding box and then pass this object through the deep learning model to determine what classification a given object is, i.e. a button. They limit the UI elements they aim to recognise to textboxes (labels in our context), dropdowns, buttons and checkboxes.

Sketch2code, from Robinson (2019), compares a number of solutions including CV and CNN to generate single page websites from sketches. They chose to fine tune an Xception model pretrained on ImageNet. Their results indicate that the CNN based approach performed more favourably with users, however the CNN is only used for classification. The author applies various computer vision techniques for object detection.

A paper with a refreshingly different name, pix2code, by Beltramelli (2017) again uses a CNN approach to process a given image into a set of DSL tokens. Unlike the other authors they perform unsupervised learning by mapping an input image to a vector of DSL tokens. Liu, Hu and Shu (2018) mirror this approach for their paper on improving pix2code.

Doodle2App (Mohian and Csallner, 2020) presents a unique solution where object detection is partly handled by user input giving the application coordinates of the user's drawing (as they draw with a mouse) but also by the user having to indicate the end of drawing each individual UI element by pressing a key ("z"). This allows them to easily identify and separate drawings to be classified but introduces additional mental load for the user (which goes some way to losing the simplicity users may be used to from prototyping on paper). Each drawing can then be passed for classification as it's completed, with feedback given in a preview pane in quick succession.

Other authors have discovered that certain types of deep learning models are able to discern the object location in a larger image as well as the classification. Adefris, Habtie and Taye

(2022) evaluate three deep CNN based object detection models. Faster RCNN with Inception V2, SSD MobileNet v2, Faster RCNN with Inception ResNet v2. The benefit of using Faster RCNN (Faster Region-based Convolutional Neural Network) is that as well as outputting a class probability for a given component it also outputs a location.

Jain et al. (2019) present another solution (also called Sketch2code) which again uses a CNN, however they use ResNet and a Feature Pyramid Network to detect objects, determine their positions and classify them. Similarly Sketch2aia from Baulé et al. (2021) use a model called YOLOv3, which is from a family of "You only look once" models, designed to detect and classify objects in a single shot.

2.4.1 You only look once (YOLO)

Of the strategies employed by other authors tacking a similar problem (detecting UI elements in an image, locating their positions and classifying them) the "you only look once" set of models stands out as an interesting solution. This particular algorithm is fast - an image can be provided to the network and the result shows the location and classification for any objects found in the input. Conversely, other CNN based solutions simply provide classification and require a separate technique to perform detection.

Redmon et al. (2016) presented the original YOLO, claiming to be the first neural network to predict bounding boxes and classifications in a single pass. They also demonstrated performance benefits over other methods such as R-CNN. Since this original paper there have been many further iterations of YOLO including (but not limited to):

- YOLO
- YOLOv2
- YOLO9000
- YOLOv3
- YOLOv4
- YOLOv5
- YOLOv8

Jiang et al. (2022) compare some of the more popular versions of YOLO and find that YOLOv5 provides opportunity for enhancements of data via augmentation techniques such as mosaic, scaling and more. It also offers similar accuracy to YOLOv4 but with improved performance. Additionally previous versions of YOLO such as YOLOv4 use the Darknet framework Bochkovskiy (2023), YOLOv5 does not, instead it uses the PyTorch framework which can in turn be converted to run on Apple devices (Bove, 2023a).

There are examples of YOLOv5 in use in our problem domain, (Altinbas and Serif, 2022) found that YOLOv5 was able to outperform an SSD (Single-Shot Detector) algorithm when trained on the same dataset. Abdelhamid, Alotaibi and Mousa (2020) trained YOLOv5 on hand drawn sketches of UI elements and managed to achieve a classification and detection accuracy of over 98%. They also suggest that data augmentation techniques could be used to increase the data usage in training, without increasing the actual amount of training data.

2.5 Training data

Broadly the approaches for sourcing training data (for training the models referred to in Section 2.4) are grouped in to two main strategies; gathering sketches from volunteers or generating synthetic data. Each comes with its own trade offs.

While Nguyen and Csallner (2015) mostly used screenshots of apps for their data, they also manually created sketches of several apps based on screenshots they had taken from the top 100 apps on the App Store. However as their approach does not use any trained models this data was used for evaluation rather than training and so no labelling or metadata was needed.

Aşıroğlu et al. (2019) used a dataset of 186 samples of hand-drawn website designs, these images were provided from Microsoft AI Lab (Microsoft, 2019) for their Sketch2Code app. They then cropped these down in to each UI component, labelled them and used the resulting dataset to train their CNN.

Adefris, Habtie and Taye (2022) focused on collecting sketches for the most commonly used UI components, they determined these to be buttons, labels, textboxes, paragraphs, text areas, checkboxes, radio buttons, etc. In total they selected 14 components and decided on a common sketched notation to represent each (Figure 2.1), for example an image is represented as a box with a cross through it. This resulted in a dataset of 562 UI images, each of which containing numerous (labelled) UI components.

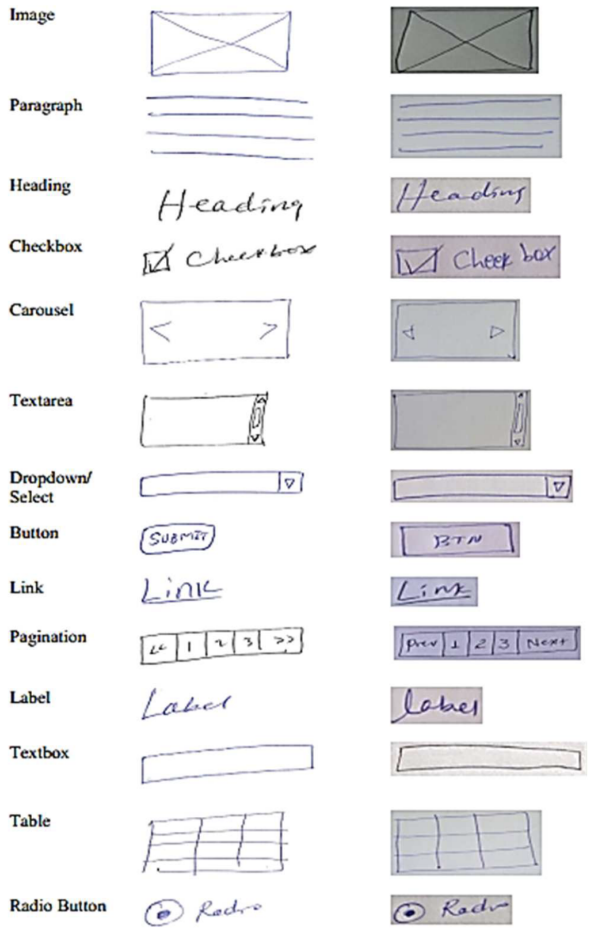


Figure 2.1: Low fidelity sketches of UI components (Adefris, Habtie and Taye, 2022)

Jain et al. (2019) supports ten UI classes and is trained using a collection of 149 hand drawn sketches, containing 2001 samples of elements. They note that performance could be improved by adding more diversity to the dataset.

Baulé et al. (2021) collected a number of screenshots of App Inventor (MIT, 2022) apps and had volunteers draw sketches of them. This resulted in 279 sketches of screens, each containing multiple UI elements which were manually labelled to identify the classification and location. Given their relatively small dataset the authors relied on transfer learning to augment a pretrained model with their own data.

Mohian and Csallner (2020) chose not to use flat images, instead proposing that the way the user draws (i.e. the order of the strokes) is valuable information that would be lost in an image. They leverage some of the dataset from Google's Quick, Draw! (Ha and Eck, 2017), which uses vector drawings to train a RNN to generate sketches. However they found there were relatively few overlapping classes for their use case. The use of a simple input device was advantageous to them in gathering additional data as they built a simple website to allow users to sketch and were able to employ Amazon's Mechanical Turk (Amazon, 2018) to collect 11,500 drawings. They were then able to use transfer learning to augment the pretrained model from Google with their own data.

Rather than sourcing hand drawn sketches of UI elements Robinson (2019) compiled a dataset by generating sketches of existing websites. 1,250 images from the dataset were used for training. The author noted that while the training performance was high the model did not fare as well with real data as there are unpredictable differences or artefacts in the generated sketches.

Another study by Beltramelli (2017) also hit the issue of no existing dataset. In their case they required access to screenshots and their resulting code (the code being written in their own DSL). To move past this issue they synthesised their own set of data by creating a UI generator. Overall their training set is 1,500 instances per platform and 250 instances for testing. Liu, Hu and Shu (2018) also use this dataset. Nikam et al. (2021) also took this approach, synthesising their own data set by generating small apps with different components and using screen shots of them as training data.

There have also been efforts to gather large datasets suitable for training detectors. Syn (Pandian, Suleri and Jarke, 2020) and SynZ (Pandian, Suleri and Jarke, 2021a) are two such collections, both presented by the same set of authors, they contain large numbers of synthetic sketches of UI elements with the goal of providing training data for other researchers. Their initial offering Syn has 125,000 low fidelity sketches which were synthesised from 5,917 hand drawn sketches by 350 participants gathered from UISketch (Pandian, Suleri and Jarke, 2021b). They quickly followed up their research with SynZ (Pandian, Suleri and Jarke, 2021a) which aims to address issues in the original dataset, namely that the sketches were not statistically similar to real-life UI screens. The SynZ dataset contains 175,377 UI sketches and was generated from a combination of user sketches and the RICO dataset (Deka et al., 2017).

Approaches relying on gathering hand drawn data from users generally all had relatively low numbers of sketches, many of these only had samples numbering in the hundreds. This can lead to detection or classification problems due to simply not enough variety of data to train a reliable model and as such many of these authors leaned on approaches such as transfer learning to augment an existing model rather than train one from scratch. Conversely, approaches relying on synthetic data may have thousands of samples, however the approaches

using synthetic data found that their trained model did not perform as well as expected when using non-synthetic data at inference time.

2.6 Approaches for code generation

Code generation is accomplished in a variety of ways in the papers we surveyed. However there are two main themes. The majority are programmatically generated in that the output of the detection and classification stage is iterated in some manner in order to output a code structure. Others take a machine learning approach of some sort, though this is much more complex.

Nguyen and Csallner (2015) do not share many details on how the export (code generation) step of REMAUI works, however we assume it is a programmatic method which uses the results of the previous steps (objects detected and their positions) to build out an android layout.

Aşıroğlu et al. (2019) constructed a "HTML builder" algorithm in order to convert their detected and classified objects into code. The algorithm itself is relatively straightforward and does not involved any machine learning components, and uses the coordinates derived from the object detection phase in order to position the UI elements.

Adefris, Habtie and Taye (2022) have created an algorithm for laying out the UI component code in a grid system, allowing it to be performed with a relatively simple combination of 'for' loops.

Robinson (2019) uses a recursive algorithm to build a HTML structure based on the bounding boxes detected in the CV stage.

As with others Sketch2Aia (Baulé et al., 2021) takes a programmatic approach consisting of several loops to build up a JSON output that can be used directly in the App Inventor (MIT, 2022) tool.

Nikam et al. (2021) also takes this programmatic approach. What sets them apart is that they choose to output code for use in React Native, a JavaScript cross platform mobile development framework. This allows them to output android and iOS applications which share a codebase. Similarly, Jain et al. (2019) output an intermediate JSON blob from their detection and classification step. This JSON is then used to generate code for iOS, android and web. This has the advantage of being able to add more UI parsers for more platforms.

The details on Doodle2App's (Mohian and Csallner, 2020) code generation are fairly scant but from the demonstration videos the detected UI elements appear to be laid out with absolute positioning in the preview pane – as in they top left pixel coordinates of the sketched item is used as the top left pixel coordinate of the output element. There are no attempts at aligning elements on a row or grid or similar, however the approach does support nesting elements inside a rectangular container.

Only a few of the approaches take a machine learning approach for the generation of code from the output of the detection and classification stage.

Beltramelli (2017) is one of the few to use a deep learning for the code generation as well as the initial object detection and classification. They use a Recurrent Neural Network (RNN), specifically Long Short-Term Memory (LSTM), to output a sequence of tokens for their own DSL. The resulting DSL code is then run through their own compiler and output for a given

platform. The advantage of using their own DSL here is that they can add more compilers in order to target more platforms and as such they're able to output code for android, iOS and web.

Liu, Hu and Shu (2018) built upon the work from Beltramelli (2017) by swapping out LSTM for Bi-directional LSTM (BLSTM) with the goal of improving code generation output and higher accuracy. The BLSTM allows for past and future contextual information to be used in the code generation.

2.7 Native vs. cross-platform development

When developing mobile apps one of the early decisions the developers must make is whether to build the app using native or cross-platform languages and tools.

2.7.1 Native development

Native development refers to using the languages and tools provided by the platform creators. For Android this typically refers to Kotlin or Java programming languages and the Android Studio (Google, 2023a) integrated development environment (IDE). For Apple's iOS (and macOS) devices this typically refers to Objective C Or Swift programming languages and the Xcode IDE. Further to this each platform has multiple options for user interface layout which include how different sized screens are handled, how rotation works and more - for iOS this generally means either using UIKit or SwiftUI. UIKit has been a staple of Apple development for a long time, many of Apple's own technologies and frameworks are available for use as UIKit components. SwiftUI however is newer and, as discussed in Section 1.2, is Apple's latest UI technology and is the recommended approach for new apps and also for their new headset the Apple Vision Pro (Apple, 2023d).

The benefits of native development are that the programmers are working with libraries and software developer kits provided by the platform creators, they can use the latest tooling as it is released and, as they're working directly with these tools, they have low computational overhead and can potentially extract high performance.

The largest downside to native development is that if your app is intended for both android and iOS app stores then you must create two apps using two sets of programming languages, tools and layout systems. This inevitably means creating the app for both platforms will require more resources, which may mean higher cost or more time, than creating the app for a single platform. Furthermore developers for these specific technologies may be more expensive, StackOverflow (2023) shows Objective C and Swift in the top 15 of the "top paying technologies".

2.7.2 Cross-platform development

Cross-platform development refers to building apps that can run on both platforms from a single codebase in an effort to save time and resources (and potentially reduce cost). There have been many cross-platform frameworks, some of which have fallen out of favour or are no longer supported, the current popular options include React Native, Flutter or Xamarin.

The advantage of cross-platform development is very clear, there is potential for developers to create one codebase and produce apps for both target platforms. They may also use

programming languages which are more common, and so less expensive to hire developers for, React Native for instance uses JavaScript which StackOverflow (2023) lists in the bottom 15 of their "top paying technologies".

However, there are potential disadvantages. Often cross-platform tooling only provides access to features that the tool developers have provided functionality for, and often this means that functionality must exist on both platforms. Which it doesn't always.

Generally cross-platform development uses another programming language such as JavaScript, Dart or C#. Sometimes, as is the case with React Native, the developers may find themselves also having to modify platform specific code - meaning a developer may find themselves working in JavaScript, Kotlin (and/or Java) and Swift (and/or Objective C) as well as all of the tooling that comes with the platforms (Meta, 2023a). This is especially the case if the developer needs to access a new UI element or feature that the cross-platform tooling doesn't yet provide access to.

Cross-platform tooling does not always use the native user interface elements, meaning that they provide their own version of a label or a button or a switch, etc. This can result in parts of the UI not feeling quite right due to differences in look and feel or behaviour (Dalmasso et al., 2013).

Further to this there is generally a performance overhead as something additional is needed to handle translating to the two platforms. For example, in React Native there is a JavaScript engine (Meta, 2023b) running to interpret the developer's JavaScript code and interact with the native tooling underneath. Authors such as Nawrocki et al. (2021) and Willocx, Vossaert and Naessens (2015) have measured the performance of native apps and cross-platform apps and generally the performance of native apps is better, as well as benefiting from faster start up time and lower memory footprint. The performance benefits of using native development on iOS were also noted by Delía et al. (2017) who found a remarkable difference compared with cross-platform approaches.

2.8 Performance evaluation

The successful performance of the various approaches we have referenced here varies greatly and is often difficult to quantify. Some authors such as Aşıroğlu et al. (2019) only discuss their training and validation accuracy (96% and 73% respectively) rather than attempting to measure how their overall approach performs with unseen user generated data.

Nguyen and Csallner (2015) leverage the fact that they are processing screenshots as an input and measure the similarity in the input image against a screenshot of their output app, and then take this further by attempting to compare the view hierarchies. They obtain reasonable results and discuss that their results may be better than measured due to differently sized transparent areas on bounding boxes of UI elements (with their output trimming much further). However they do not discuss what effect having unsupported UI elements in the input image had on the output or the measurements.

Other approaches to performance evaluation include user surveys, Adefris, Habtie and Taye (2022) used ten sketches and used their system to generate corresponding web pages. They then randomly selected 24 individuals in the IT industry and asked them to evaluate each of the generated pages given the input sketches. Though the sample size of individuals and data

is relatively small they do achieve good results with only two users claiming two designs did not match. Baulé et al. (2021) also take this same user survey approach with comparable results to Adefris, Habtie and Taye (2022). Given the highly subjective nature of design and how a sketch may look when translated in to a more precise output this seems a reasonable approach.

Robinson (2019) also evaluated performance with a user survey (as well as presenting other metrics for model performance) and this led them to the conclusion that the performance was not high enough to be used in production environments. This is further emphasised by Calò and Russis (2022) who use a similar dataset of artificial sketches and note that while the accuracy of predicting synthetic sketches is good ($\sim$90%) the performance with real sketches is significantly lower ($\sim$68-69%).

Beltramelli (2017) reports the error percentage in UI element classification per platform (iOS, Android and web), with web performing the best at $\sim$12%, and iOS and Android performing similarly at 22%. Their conclusion is that performance could be improved further by training a bigger model on significantly more data for more epochs. Liu, Hu and Shu (2018) show that the implementation of BLSTM can enhance the performance of pix2code but the same points remain around training data size.

Mohian and Csallner (2020) used several metrics to evaluate Doodle2App. These included measuring the average runtime to process and classify a (single) UI element as well as the average time for detecting an element to converting it to an Android application. Other metrics included comparisons to a competing product, Teleport (Brie, 2019), both comparing detection/classification time and classification accuracy (but only for overlapping classes). The latter however doesn't seem an entirely fair comparison as the Teleport model is aimed at detecting hand drawn elements on pen and paper and is being fed entirely computer created images (drawn by mouse) which will have differences such as no pen/pencil texture, more artificial lines (the difference between drawing with a mouse vs pencil is likely significant).

2.9 Literature review conclusion

All of the approaches discussed have various limitations that prevent them from being particularly useful in the real world. Many only support a limited set of UI elements, REMAUI (Nguyen and Csallner, 2015) being a prominent example as it only really supports binary classification of elements - label or image.

A key limitation is that many of the approaches ignore key layouts like "lists" (or 'UITableView' in iOS UIKit) but a huge number of apps rely on lists as their root screen to navigate to the detail of a given item (think of a mail app for email, the root screen is often the list in which you tap an individual item to navigate to the detail for a given email). From our research only Nguyen and Csallner (2015) seem to have attempted to handle a list-like layout.

Looking at the various methods used to obtain training data it becomes clear that while it may not seem that a large number of screenshots or input sketches are needed, those inputs contain multiple UI elements, for instance Adefris, Habtie and Taye (2022) collected "only" 562 UI images but that resulted in 11,152 UI components.

It is also interesting, if not entirely surprising, to see other authors struggle with the lack of data. Some turn to generating artificial sketches from finished websites or apps (Robinson,

2019). While this is a novel solution to hard to source data the resulting model doesn't perform as well with real data as it might have if the training dataset contained human drawn images. Similarly, others generate their own UI and matching code samples (Beltramelli, 2017) but find that this doesn't provide enough training data to achieve high performance levels. The symptom of this, as Baulé et al. (2021) found, is that users may have issues with mis-classification, size or layout of detected elements.

Another popular approach to dealing with the limited amount of training data is to restrict the number of UI components the systems are looking to recognise. With some authors only attempting to interpret images and text (presumably any UI element that is not text can then be recognised as an image or used as training data as an image).

Many solutions attempt to detect objects and classify them separately, typically by determining a bounding box for each detected element, cropping the image and then classifying that image. Approaches like Adefris, Habtie and Taye (2022), who use Faster R-CNN for both detection and classification, offer a clear advantage in terms of reducing steps and ensuring the classification model is referring to the exact same pixels as is being specified for the bounding box. Approaches that use YOLO models are an evolution of this approach, offering detection and classification in a single pass through the model in a faster time than R-CNN approaches.

There are also a variety of code generation methods on display in the papers reviewed, while many turn to relatively simple algorithms to iterate over the output of the detection/classification step, some attempt to use deep learning to find the relationships between visual elements and the resulting code. Beltramelli (2017) also suggests that Generative Adversarial Networks (GANs) may be a good fit for generating code from an input image in future research.

Regarding the choice of a programming language to support for code generation output, we have found that few authors deal with native iOS languages (Swift or Objective C). Most similar solutions either use cross-platform solutions (Baulé et al., 2021), Android (Nguyen and Csallner, 2015), or generate non-mobile layouts in HTML (Aşıroğlu et al., 2019). Exploring iOS code generation is the best choice for potential cost savings (given the cost of developers in this space (StackOverflow, 2023)), the lower quality of UI in cross-platform (Dalmasso et al., 2013) and the superior performance that native iOS development offers (Delía et al., 2017).

Surprisingly, almost all of the approaches we have investigated rely on using a paper based sketch, which must be digitised in some way, as the input. The nature of this process means it can only generate non-real-time code, as the the user has to upload an image each time they want to change the output. Only Mohian and Csallner (2020) and Wimmer, Untertrifaller and Grechenig (2020) present alternatives to this process. One relies on the user drawing using their mouse with occasional keyboard input to indicate the end of drawing a UI element - when this key press occurs the system generates a new output and the user can see feedback before they have finished the entire sketch. The other relies on a cumbersome hardware configuration and whiteboard which is far from portable or quick to set up. We believe that this is an area which could be improved upon.

Chapter 3

Requirements

3.1 Introduction

In this chapter we provide an overview of the requirements gathered and their source or reasoning. The requirements have been gathered from a number of sources, the main one being the literature review. As part of the literature review we evaluated similar solutions for generating code from sketches. This highlighted a number of shortcomings of some approaches as well as identifying new areas to explore.

3.2 Goals

Our literature review in Chapter 2 identified several opportunities for research given the current state of the art. We cannot attempt to tackle all of these, as such we have chosen the areas we believe we can have the biggest impact in. The key points we intend to expand upon are how the sketch is input in to the system, code generation for iOS (in particular SwiftUI) and combining detection and classification with a YOLO model trained on our own data.

Our high level goal is to attempt to improve on the prototyping phase of mobile app development. This phase is often done using pen and paper and we aim to replicate the speed, portability and flexibility of this technique using a digital alternative. We intend to do this by creating an app that allows a user to draw a sketch of a UI and the app will generate the code for the user to continue using for further development. While many authors investigating this area took the route of using an image of a paper sketch as input we intend to attempt to use sketches being drawn by the user, allowing for faster feedback and reducing the manual steps needed to use the system.

3.3 Detailed requirements

We have used MoSCoW analysis (Dawson, 2015, p.121) to prioritise the requirements according to:

- Must have - the project must include this
- Should have - the project should include this

- Could have - the project could include this
- Won't have - the project won't have this in this version but this could be considered for a future version

RQ1: The user must be able to draw on the device

Other approaches generally capture a pen and paper sketch using a camera and then upload the result to a computer for processing (as discussed in section 2.3). This is a cumbersome process as noted by Carter and Hundhausen (2010). Instead, our goal is to allow the user to draw directly on their device.

RQ2: The app must generate SwiftUI code

In the literature review we noted that very few approaches output code for Apple devices (see Section 2.6), and more specifically that no other approaches have attempted to output SwiftUI (Section 1.2). This requirement is important as Apple's recommended approach to developing apps on their platforms is now Swift and SwiftUI (as discussed in Section 2.7.1), so this choice will ensure the tool outputs code in the current industry standard language. It is also clear that the native languages are more expensive to implement so it makes sense to automate this area first.

RQ3: The user should be able to draw with a stylus

The experience should remain as close to drawing with pen and paper as possible, as Carter and Hundhausen (2010) observed this was the preferred method for prototyping. (Mohian and Csallner, 2020) noted that a potential future area of research would be to build a system which used a graphics tablet and stylus.

RQ4: The app should show its output as the user draws

The user needs to know the app has correctly interpreted their drawings whilst using it. Most other approaches we reviewed perform their output generation in an asynchronous manner. Capturing the user's drawings on their local device gives us an opportunity to attempt to recognise sketches in real time, if we can achieve a good level of performance. Most other approaches don't offer this opportunity as they require digitising a sketch after it is completed, except for Mohian and Csallner (2020) who also employs this strategy.

RQ5: Code generation should happen automatically as the user draws, it should not need user intervention

Mohian and Csallner (2020) had the closest approach to ours as they tried to move the whole prototyping and code generation process on to a single device. However, they required the user to indicate to the system that the user had completed drawing each element. While this makes it easier for the system to identify distinct drawings to use for detection it makes the drawing process less natural. (Mohian and Csallner, 2020) used a system which pointed a camera at a whiteboard, allowing for a more natural drawing experience but their system lacks the ease and portability of our approach and we intend to address some of their approach's limitations in this project. From our research in Chapter 2 this seems to be an under explored area but it will significantly enhance the user experience so will be a key focus of this project.

RQ6: The user should be able to draw, see results and run the generated code on the same device

One of the disadvantages of other similar systems is that the user must draw their sketch on pen and paper, capture it and upload it to a computer in order to produce generated code for them to use. Our goal is to streamline this process to replicate the simplicity of pen and paper. One way to close this gap in the software development process is to be able to handle it all on a single device.

RQ7: The user should be able to export the generated code

The value of generating code is that it potentially gives developers a head start, however the user of the app may not be a developer so it should be possible for the user to export the code for use on another device.

RQ8: The app could work "offline"

The prototyping phase can happen anywhere and users may be in a coffee shop or a meeting room with clients or other locations. Pen and paper works in all of these environments and doesn't need an internet connection. Ideally, we are aiming to match or improve upon the pen and paper experience, and being able to work without an internet connection is something we can potentially replicate.

RQ9: The app will not consider complex UI designs

The project will not consider the generation of code for complex UI designs that require advanced features such as animations, navigation or custom views. The focus will be on generating code for basic UI designs such as labels, buttons, and text fields for a single screen at a time.

3.4 Conclusion

These requirements were created to address the limitations of the existing approaches explored during the literature and technology survey (Chapter 2) and will form the basis of our design and implementation in the following chapters.

Chapter 4

Design

The chosen approach is to use the Apple iPad and Apple Pencil (a stylus) as an input device. This allows us to fulfil multiple requirements **RQ1**, **RQ3** and **RQ6**, whilst other devices such as a laptop would only fulfil some of these requirements. It also allows us to deploy on a portable device which aligns with our broad goal to replicate the advantages of pen and paper. If we can deploy the entire app (drawing, machine learning model, etc) on an iPad and not rely on a server (i.e. if the machine learning model were to be cloud based) then this allows for a level of portability other similar systems discussed in Chapter 2 could not reach. If our model can run fast enough it will also allow us the opportunity to give feedback to the user as they sketch, something very few other solutions have achieved. The iPad also allows running SwiftUI code in its Swift Playgrounds app, which we may be able to leverage to compile our own generated code. Furthermore, Apple platforms are some of the more expensive to develop mobile apps for (StackOverflow, 2023) so it makes sense to target time (and therefore cost) savings in this area. Finally, from our research (Chapter 2) no other author has attempted this approach entirely on an Apple device.

In our literature and technology survey we discovered that synthetic data sets used to train a model do not perform as well as may be desired when used with real user data (Section 2.5). Given our intention to capture sketches on a device no other solution seems to be using we may expect to see a similar issue if we were to use a synthetic data set, such as SynZ Pandian, Suleri and Jarke (2021a). There would also likely be issues if we were to use a dataset consisting of photos of paper-based sketches. For example, such a dataset may have artefacts we are unlikely to see when using an Apple Pencil including camera related issues like angle to the paper, orientation and lighting (Abdelhamid, Alotaibi and Mousa, 2020), perhaps even details in the pencil drawing itself such as the texture of the line made by graphite (a feature that Apple Pencil drawings will lack). While it may be that in preprocessing these datasets we could render much of these differences moot there could be parallels with Syn (Pandian, Suleri and Jarke, 2020) where the dataset they generated ended up more different to the target than desired. In addition, machine learning algorithms are very good at learning imperceivable details and bias from a dataset and there may be differences we are unable to detect and correct/remove.

As such we have elected to gather our own dataset on device, in an attempt to get the best model performance possible. While acknowledging we will likely have to deal with a smaller dataset and make use of approaches such as transfer learning. This also means we do not need to address issues like identifying the paper in an image and removing the background or

overlapping artefacts in the image (i.e. a pen left on top of the paper or other objects).

This project begins with no existing data on which to build a model so it touches on every stage of the data science pipeline. The pipeline has been described in several ways but all contain similar components. One described by Biswas, Wardat and Rajan (2022) is shown in Figure 4.1.

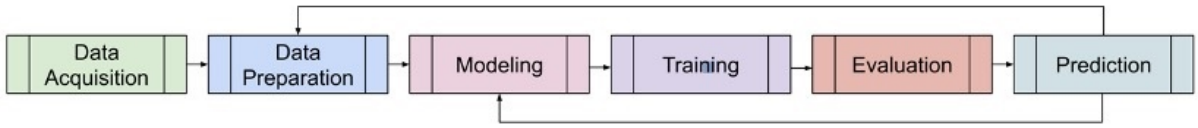


Figure 4.1: Data science pipeline overview (Biswas, Wardat and Rajan, 2022)

We are building three distinct components:

- Data capture tool
- Machine learning model
- Drawing app

These components align to different stages of the data science pipeline. The data capture tool will cover data acquisition as well as the data preparation. In training the machine learning model we will cover modelling, training and evaluation. Finally the data capture app will handle the prediction phase.

The following sections discuss our design for these components in more detail.

4.1 Data capture tool

4.1.1 Data acquisition and preparation

Most other authors gathered drawings on paper and then digitized them to use for training. These images were sourced either from other projects, as with Aşıroğlu et al. (2019), or gathered by the authors from volunteers as with Baulé et al. (2021).

Some authors did capture drawings digitally, such as Mohian and Csallner (2020) who leveraged data gathered by Google (Ha and Eck, 2017) using an online game as well as building their own online tool to gather additional digital drawings.

To capture sketches from users on device we will build an iPadOS app which allows the user to draw, using an Apple Pencil, on a given canvas area. To give users some idea of what to draw we will provide an example of an iPhone app UI from an existing app on the same screen and instruct them to attempt to make a "low fidelity" copy in their drawing. This is similar to Baulé et al. (2021) who showed screenshots of apps to their users to draw for training data.

As the user draws, our app will capture their sketch data. Once they have finished drawing the app will have a mode for the user or the instructor (the author) to label each UI component the user has drawn.

An image of the user drawing and the labelled data – consisting of bounding box coordinates, classification types and a cropped image of that element – are uploaded to cloud storage for later manipulation.

Given our unique input device we cannot employ tools like Mechanical Turk (Amazon, 2018) to gather additional images.

4.2 Training the model

4.2.1 Modeling

At this stage we have labelled user data (images of sketches with a set of bounding boxes and classifications for each UI element) in a data store. Given we are able to collect labelled data it makes sense to use supervised learning in order to create a model.

We have bounding box information from the user labelling/data capture tool which we could use to split up the data (i.e. crop the full size image to only be an image of the element in question) we will have situations where elements overlap. This is just the nature of sketching - humans are imprecise and the user is drawing a relatively small item in a relatively tight space - and to some extent this is the nature of UI design - some elements may be containers or overlap. Many of the authors in the literature review had to use a variety of strategies to attempt to detect and segment individual UI elements. For instance Aşıroğlu et al. (2019) greyscale and Gaussian blur the input image, apply morphological transformations and crop the detected objects to their bounding box to create something to pass to their deep learning model for classification. Others, like Baulé et al. (2021), use YOLO (Section 2.4.1) since algorithms of this type will detect and classify anything found in a given image. Of the approaches researched, the latter - which uses a YOLO algorithm - is most efficient (less pre-processing) and may deal with overlapping elements better, so we have elected to use YOLOv5 for detection and classification.

4.2.2 Training

We are collecting and building our own dataset so expect to have a limited amount of data. To manage this we will use transfer learning to augment an appropriately selected, pre-existing object detection model with our data.

To train our model we expect to need to experiment with various hyperparameters to achieve a good result. Altinbas and Serif (2022) suggest that data augmentation can be used to increase the amount of data that can be used while training. We intend to experiment with data augmentation techniques during training.

4.2.3 Evaluation

A small subset of our captured data will be set aside for evaluation. This data will not have been seen by the model (as it was not used in training) and as it is pre-labelled we can determine how successful at detection and classification our model is. This will give us some sense of how well the model is able to perform with unseen data. Though we acknowledge this will reduce our training data set size this is an essential step to be able to allow us to compare model versions.

4.3 Drawing app

4.3.1 Prediction

To evaluate the success of our model with a larger variety of data we will need to deploy it in our iPad app and test it with users. To successfully run the model on device in a reasonable time we ideally want to have our code run on the Apple Neural Engine (Orhon et al., 2022), as opposed to the CPU, as this is a specialised element of the Apple Silicon system-on-a-chip (SoC) which is designed for running ML models. However, to do this we will need to convert the model we trained in to a supported format but fortunately Apple provide a set of tools to accomplish this. The Apple Core ML Tools (Bove, 2023b) allow developers to convert models from PyTorch, Tensorflow, scikit-learn and several other libraries and frameworks. This reinforces our choice of using YOLOv5 to train our model as it is PyTorch based unlike some other YOLO frameworks such as Darknet (Bochkovskiy, 2023) which is C based (Solawetz, 2020).

4.3.2 Visualisation

The detected elements produced by the model will be processed by the app and presented to the user in a preview panel alongside the drawing panel. This gives the user immediate (or close to) feedback and allows them to visually see if the model has correctly interpreted their drawing.

We will also provide a view to show the code which has been generated as a result of the user drawing, as well as a way to export this to other coding tools to allow for compilation and modification.

Chapter 5

Implementation and Testing

In this chapter we discuss the detailed implementation of the system described in Chapter 4. Each component underwent various revisions as it was developed to ensure it meets the goals and requirements in Chapter 3.

The implementation consists of three distinct parts: the data capture tool, the model training and the drawing app.

- Data capture tool - this component is an iPad app which allows users to copy a provided image as a sketch. This sketch is stored as part of our training data.
- Model training - this component consists of Python code which manipulates our dataset and trains a machine learning model for detecting and classifying sketches of UI elements.
- Drawing app - the main output which combines elements of the other two components. It is an iPad app which allows users to sketch a UI which is passed through the machine learning model to detect and classify each element before processing them and displaying a preview along with generated SwiftUI code.

5.1 Data capture tool

The purpose of the data capture tool is to record, label and store user sketches to use this data for training later. The tool requires a significant engineering effort and includes a front-end iPadOS application written in Swift and SwiftUI as well as a back-end service created on Google's Firebase (Google, 2023d). Figure 5.1 shows the high level architecture of the application.

The app itself starts with a user consent screen, if users agree to the terms presented then they are taken to the main screen (Figure 5.2). Here users are presented with a drawing panel on the left and a screenshot of an existing iPhone app on the right. The user's goal when using the app is to sketch their interpretation of a low fidelity version of the screenshot they are being shown (on the right of Figure 5.2). This process was inspired by Baulé et al. (2021), who gathered screenshots of apps and had volunteers draw paper sketches of them. Once the sketch is complete the tools along the top left of the screen are used to labelling the drawing data. When all UI elements have been labelled the user presses close and is prompted to save their work, at this point the users consent form, sketch image and associated data are uploaded to Firebase for storage.

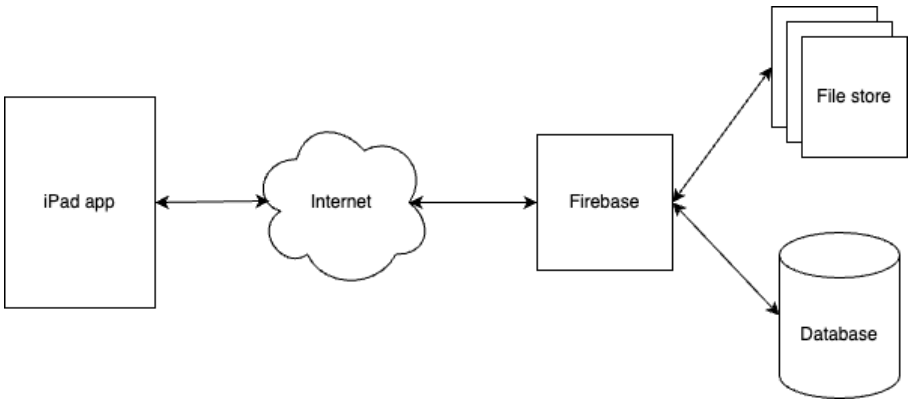


Figure 5.1: The architecture of the data capture tool

5.1.1 User consent

The data capture tool will record and store user sketches for training our model, so we must inform users of our intent to do this, what we will use the data for and request their permission to store the data they provide. The first screen the users see explains the purpose of the app and asks them to accept the terms and conditions of use, as shown in Figure 5.3. They then have the option to fill in a name for us to refer to them by (they are free to use pseudonyms or leave the field empty), and must consent to us collecting their data. If they disagree to us storing their data they cannot continue to the drawing screen and the form resets ready to be completed by another user). They must also agree or disagree to a debrief and are asked if they would like to participate in the evaluation or not. The only required field that prevents a user progressing to the drawing screen is the acceptance of the terms and conditions.

The data captured here is held in memory on the device, it is then uploaded and stored in Firebase if and when the user saves a sketch.

For ease of tracking (for the author) this screen shows a count of the current number of sketches stored in Firebase at the bottom left.

5.1.2 Example apps

To give users inspiration for which UI elements to draw and in what order or layout we show them an image of an existing app. These images were gathered from manual screenshots of apps featured in the Apple App Store charts or apps that we had access to the source code for. Overall just under two hundred screenshots were collected and uploaded to a Firebase Cloud Storage (Google, 2023b).

The images selected were chosen for their common UI elements or app design patterns like login or registration screens. The screens were all captured without any user data being displayed or entered, i.e. on a registration screen all fields were left empty.

Whenever the drawing screen is loaded the app fetches the listing of files from Firebase, selects a random file and downloads it. This image is then presented to the user on the preview panel of the drawing screen.

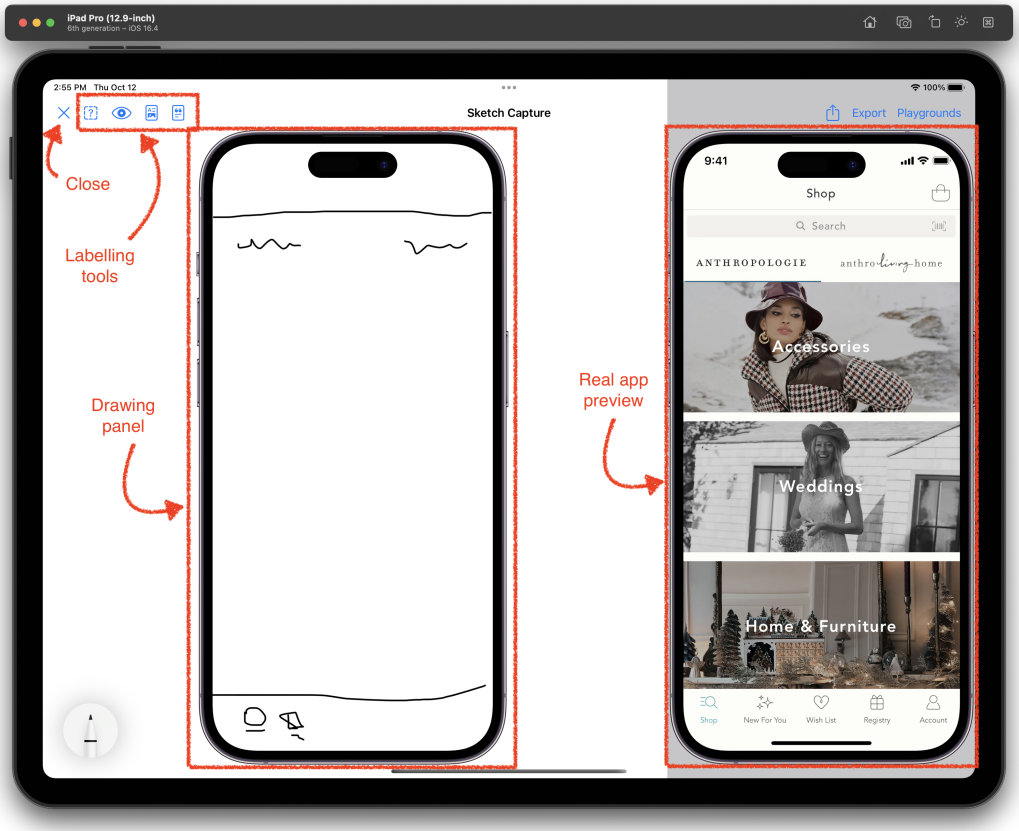


Figure 5.2: An annotated screenshot of the data capture tool user interface

5.1.3 Drawing

To enable us to capture data to use for training we had to create a tool to capture sketch information from potential users and volunteers. This tool needed to use the same drawing tools that the final application would use in order to ensure the model was trained on similar data with the same set of details the final tool uses - for instance if our data capture tool used thicker or rougher lines than our final tool, then we may see the confidence of results affected, or perhaps no results. This meant the drawing panel we created was to be used both for the data capture tool and the final drawing application.

Apple provides a View for UIKit called PKCanvasView for interacting with the Apple Pencil and it provides much of the drawing support. However, we used SwiftUI to create the app UI because Apple is pushing SwiftUI as being the best choice for app development on their platforms (as discussed in Section 2.7.1). Since PKCanvasView is not supported out of the box in SwiftUI, we created a wrapper called CanvasView to allow the app to use it as part of the SwiftUI interface. The resulting View gives the user the ability to draw and erase with the Apple Pencil and gives the app callbacks for every stroke (or line) the user makes.

5.1.4 Labelling

We are using a supervised learning approach and so, as discussed in 4.1.1, we must label the user data to train our model with the various classes we need to detect. To do this we provide a labelling mode which displays bounding boxes to the user over the top of their sketch. The

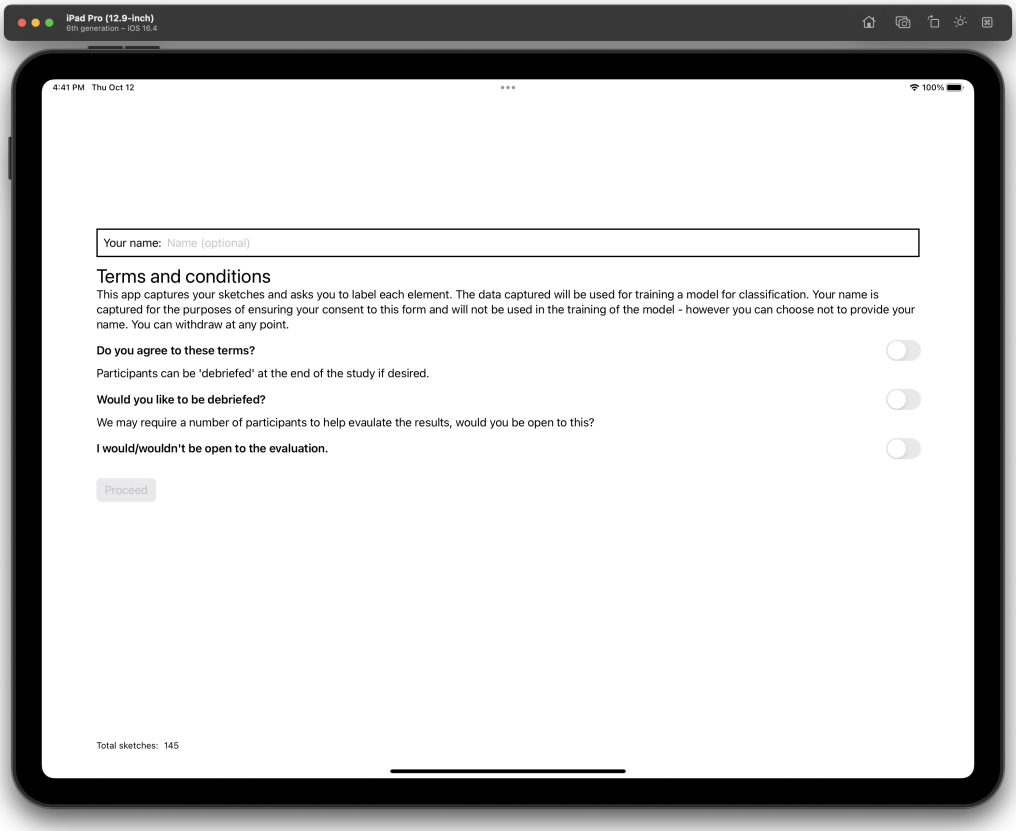


Figure 5.3: User consent screen

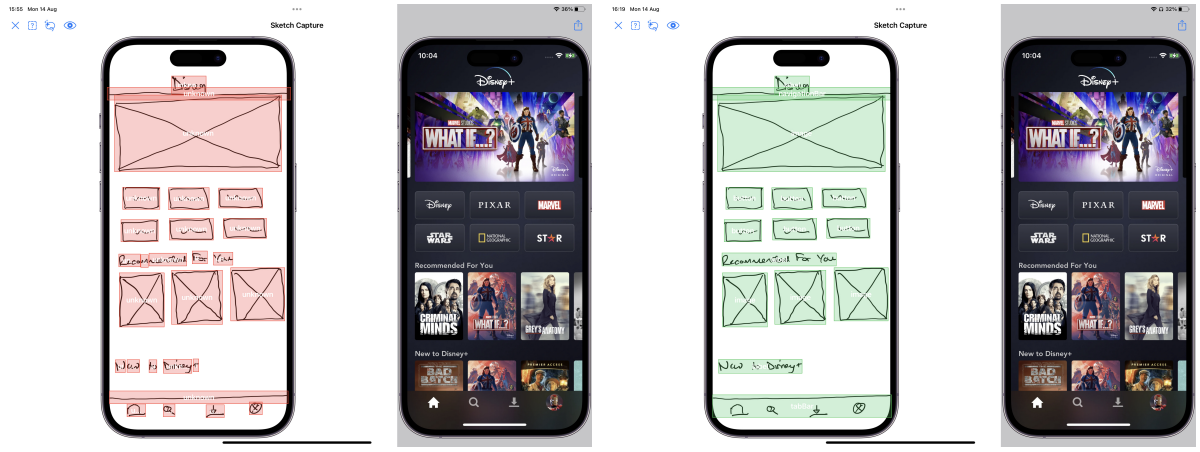
bounding boxes are initially created from the user stroke data captured by our drawing view - we have the x,y coordinates, width and height of each stroke.

Creating bounding boxes

Given any UI element drawn by the user may be made up of multiple strokes we attempt to group strokes together to try to guess what might be a UI element which needs classification. We achieve this by creating a bounding box for each stroke, then iterating over the list of boxes checking for any overlaps with any other boxes. When an overlap is found the two overlapping boxes are merged. The final list of bounding boxes contains no overlapping elements and is then displayed to the user.

Our initial bounding boxes often give the user a good starting point for labelling elements but it does not always find the correct grouping or bounds. Figure 5.4 shows the initially generated bounding boxes (Figure 5.5a) and the user corrected and labelled bounding boxes (Figure 5.5b). If these images are closely inspected we can see some differences, specifically the text "Recommended for you" in the centre of both images. It's clear to the observer that the sentence should be a single box on the UI so that it may be labelled as a single element for training.

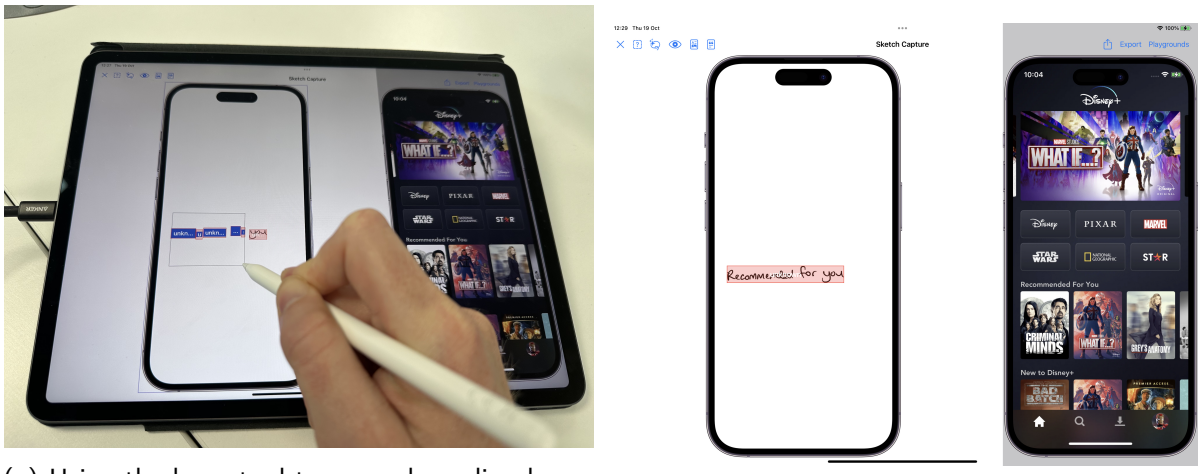
However, when we calculated the bounding boxes based on strokes and their overlap the words themselves did not overlap. In some cases letters in a word did not overlap resulting in several bounding boxes. To enable the user to label or classify this text as a single UI element we



(a) Automatically generated bounding boxes (b) User labelled bounding boxes

Figure 5.4: Generated bounding boxes versus user confirmed bounding boxes

provide a (lasso) tool that allows the user to group select bounding boxes and combine them. This can be seen in 5.5.

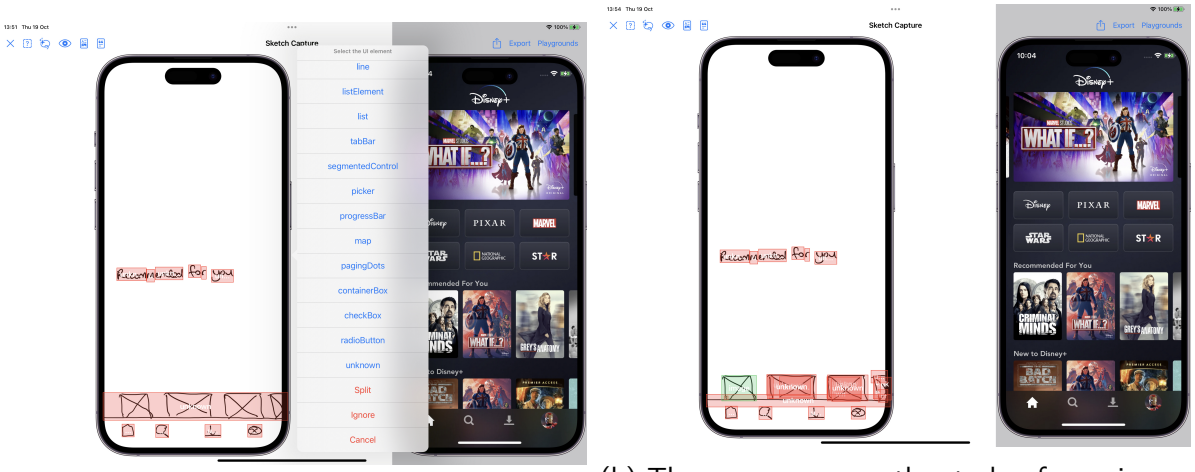


(a) Using the lasso tool to group bounding boxes together

(b) The resulting bounding box

Figure 5.5: Combining bounding boxes

Users could also find that due to their drawings inadvertently overlapping they have a starting bounding box that contains more that one separate element. We implemented a "split" function that the user can select which breaks a given bounding box down into its component strokes, with a single bounding box per stroke. The user can then use the lasso tool to group together only the strokes that are needed for each desired bounding box. Crucially, the lasso tool only selects "unclassified" bounding boxes, this means that a user can easily group strokes in to a bounding box, label it and then work on grouping the remaining strokes (which may overlap or be extremely close to the previously created box). This functionality is shown in Figure 5.6. Where a user needs to split up the tab bar and the images at the bottom of the screen, they simply tap the bounding box that needs operating on and a list of options are presented (Figure 5.6a). At the bottom of this list is the option to "Split", after selecting this the bounding boxes are reduced to their most granular to allow the user greater flexibility (Figure 5.6b).



(a) Wrongly grouped tab bar and the user options

(b) The user can group the strokes for an image together and label that element

Figure 5.6: Splitting bounding boxes

In addition to the grouping and splitting tools provided to users, the main function required of them is to label the UI elements in their sketch. When the user is satisfied a bounding box is correct they simply tap on it to bring up the list of options, this list (shown in Figure 5.6a and 5.7) contains an entry for any UI element we support (and some we might like to support) as well as the split and ignore functions (the ignore function simply excludes a bounding box from use). Once one of the labels has been selected the bounding box changes colour to green and displays its classification inside it (this can be seen on the image in the bottom left of Figure 5.7). This process is repeated until the user has labelled all the bounding boxes.

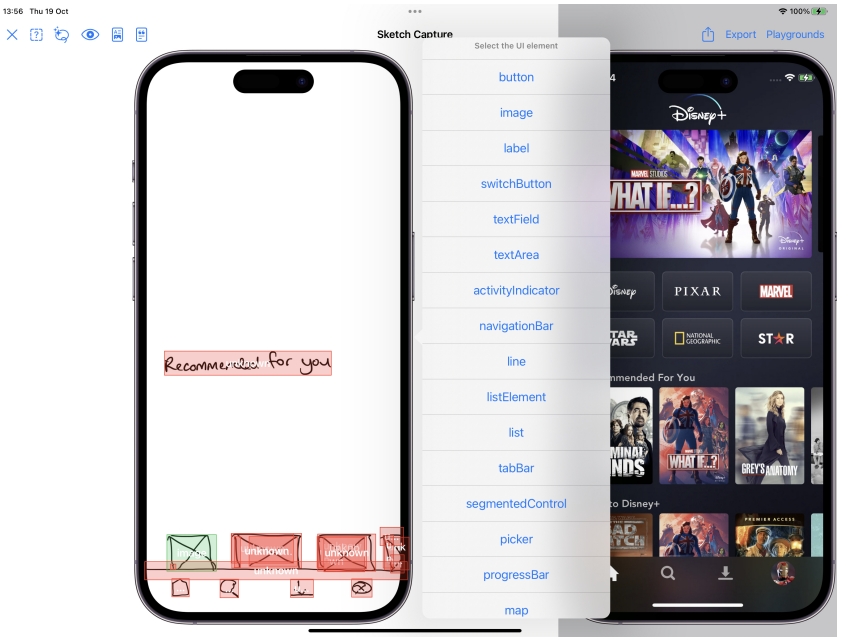
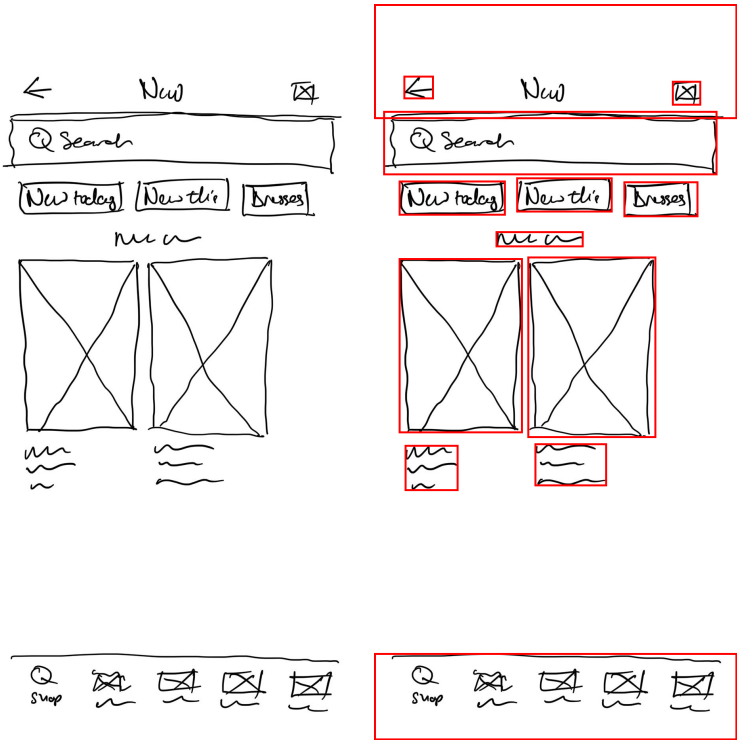


Figure 5.7: Bounding box labelling options

5.1.5 Post processing and storing

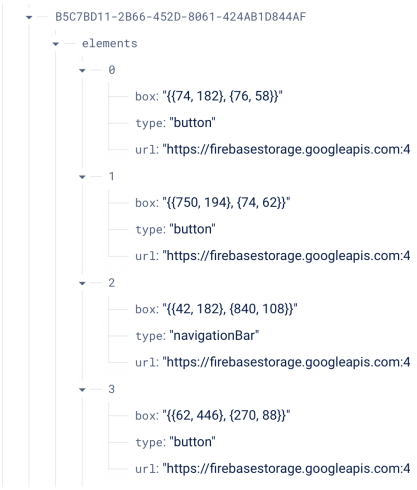
When the sketch is complete and all data has been sufficiently grouped and labelled we save the captured data for processing and training the model at a later time.

The app creates an image of the drawing area, capturing the user's sketch (Figure 5.8a), and uses the final list of bounding boxes to also create cropped images of a UI element defined by a bounding box. The complete image and individual images are uploaded to our Firebase Cloud Store. The raw data of the bounding boxes (which includes x, y, width, height and label) is stored along with the uploaded image paths (to link bounding boxes to their respective images) and the user consent form are stored in a Firebase Realtime Database (Figure 5.8c). To separate the data it is stored in a tree-like structure with the date at the root node, followed by a unique identifier for each sketch created on that date.



(a) The sketched image

(b) The labelled image



(c) The structure of data stored in the Firebase Realtime Database

Figure 5.8: An image drawn by a user, labelled and stored

5.2 Training the model

To train the model to detect and classify UI elements in a given image there are several stages we needed to move through. Each of the steps described have been created in a set of Jupyter Notebooks:

1. fetch_firebase_data—yolo.ipynb - includes fetching data, pre-processing, training and validation
2. yolo—to—coreml.ipynb - converts the model we created to CoreML

5.2.1 Data

From our data capture process we gathered a total of 145 drawings from 10 users. These drawings break down as shown in Table 5.1.

Class	Count
button	389
image	274
label	680
tabBar	57
navigationBar	46
segmentedControl	7
textField	46
switchButton	20

Table 5.1: The number of training data examples for each UI element class

5.2.2 Fetching and pre-processing

The first step is to retrieve our data from Firebase. Our notebook authenticates with Firebase, then iterates through the Realtime Database tree.

For each node (which represents a single screen sketch) we retrieve the coordinates of the bounding boxes and their classifications. These coordinates are originally created on the iPad in the coordinate system used by the device, where the origin is the top left of the screen and for each box the x and y represent the top left corner of the box, and all values are absolute values. The YOLO algorithm however expects a relative coordinate space; x and y represent the centre of a box and all values are relative between 0 and 1 (with 1 being the full height or width). So as part of our data retrieval we convert the coordinate from the iPad space to the YOLO coordinate space.

We then programmatically create a text file for each sketch containing a row for every bounding box, consisting of the classification (as a number) and the converted coordinate information. A sample of this can be seen in Figure 5.9.

Along with each text file the corresponding sketch image is downloaded. It is stored in Firebase Cloud Store as a `.png` file as the raw sketch image is comprised of line drawings on a transparent background. For training we must remove the transparency from this image so it creates a white image with the same dimensions, copies the sketch information over the top and then saves the resulting image as a `.jpg` file.

Exceptions to the rule

For some UI elements we were able to apply some additional pre-processing in an attempt to normalise some of the data. When converting bounding boxes specified by users in the iPad screen coordinate space to YOLO coordinate space there are some elements where we already know the expected size or position, so we took this opportunity to override them here. Specifically the `navigationBar` and `tabBar` will always be the full width of the screen (and so we replace `width` with `1`). In addition, we know that a `navigationBar` is only ever positioned at the very top of the screen, so we convert the centre Y position (taking in to account the height)

Classification	Centre X	Centre Y	Width	Height
1	0.553610503	0.353225806	0.568927789	0.252688172
2	0.507658643	0.559139784	0.595185995	0.047311827
2	0.515317286	0.681182795	0.557986870	0.091397849
0	0.497811816	0.890322580	0.671772428	0.096774193

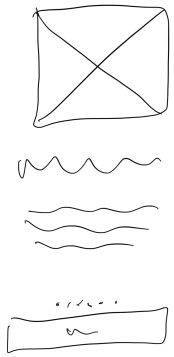


Figure 5.9: The bounding box data (top) for the sample sketch (bottom)

to sit against the top bounds of the image. Similarly, a tabBar only occurs at the bottom of the screen and so we recalculate the Y coordinate to position it at the bottom.

5.2.3 Training

With the training data downloaded and correctly formatted we split our data amongst three folders; train, validation, test.

First we shuffled the data (to avoid always training our model with the same data in the same order, or always putting the same data in to the same folders) and then we allocated them between the three folders with the highest amount allocated to 'train'.

Given our relatively small dataset, and the success seen by other authors such as Baulé et al. (2021), we have chosen to take advantage of transfer learning. This allows us to take an existing model, freeze some of the weights in the existing layers and only train the remaining layers at the end of the model with our new data. Only changing these last layers allows us to swap out the final detection layer, where classification occurs, with our own layer which contains only our classes (and not the classes the original model was created to detect). Initially we focused on freezing the backbone of the network and only training the head, for the YOLOv5s model this was the first 10 layers.

YOLOv5 is distributed with several model sizes all trained on the COCO dataset (Lin et al., 2014). As we were intending to run on an iPad, which could be considered a low powered device and we would like fast feedback from the model, we elected to use yolov5s as our starting point because it's a relatively small model at 7.2 million parameters, Jocher (2023).

Later in the project we experimented with freezing fewer layers and data augmentation with the goal of improving accuracy.

Data augmentation

As our dataset is comparatively small one technique to artificially increase the amount of training samples is to employ data augmentation. There are multiple strategies for data augmentation but the approaches that seemed to fit our purposes best were mixup, mosaic and copy-paste.

Mosaic data augmentation

This technique creates new training data by stitching together four different images from the training dataset at different scales. The bounding boxes from the input images are carried across, scaled and re-positioned to account for the new scale and location. A sample of this new training data can be seen in Figure 5.10.

There are several advantages to this approach including; low computational overhead due to only rearranging existing data, and aiding multi-scale detection as objects are being introduced at additional scales.

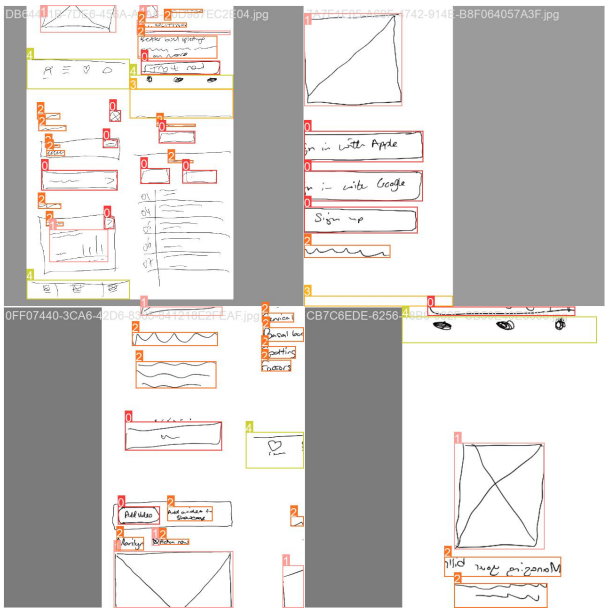


Figure 5.10: Example of our training data with mosaic data augmentation applied

Mixup data augmentation

In mixup, two images are superimposed with different coefficient ratios and then the associated labels adjusted and superimposed (Zhang et al., 2017). For each pair of input images mixup creates a pair of mixed up images. It can also lead to overlapping features and noisy input images. However, it also has a low computational overhead and for our purposes may help create new data containing UI elements which are close together or overlapping - which will often occur in real use. Because elements can become combined their labels become "soft labels" which are a blend of the two.

A sample of this approach, when combined with mosaic, can be seen in Figure 5.10.

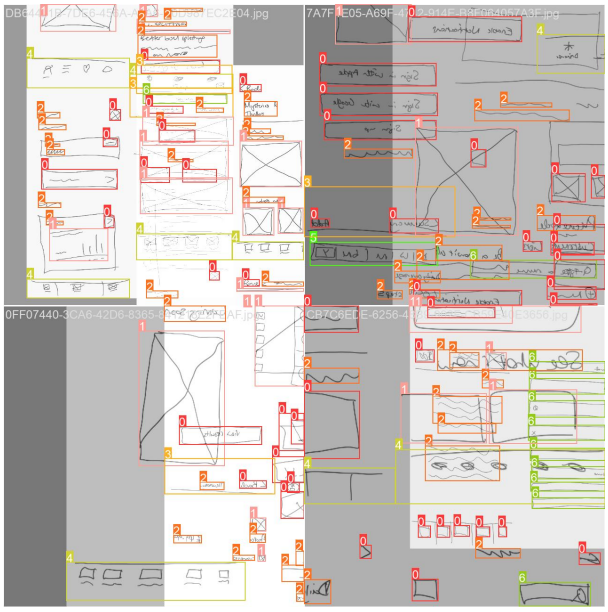


Figure 5.11: Example of our training data with mosaic and mixup data augmentation applied

Copy-paste data augmentation

Copy-paste data augmentation is where a region of a source image is selected and copied into another image. This may include some or all of a labelled element, and if so that labelling is carried across or adjusted if it is only part of the labelled element. This has the advantage of training the model to deal with occlusion (as there may be only part of an element) as well as introducing items in more contexts than may occur in the original training data.

Training results

The results of these experiments can be seen in the F1 curves shown in Figure 5.12, we chose to use F1 graphs for comparison of the accuracy of various experiments as it offers a value computed using the precision and recall. Each graph shows the overall F1 score (bold blue line) and the key states the score and what level of confidence achieved (for instance in Figure 5.12a "all classes 0.71 at 0.429" indicates the F1 score of 0.71 was achieved at a confidence of 0.429).

We can see in Figure 5.12a that even without additional data augmentation (mosaic augmentation is used in all of our experiments) and with a relatively small amount of data we can achieve some reasonable results. However, this was more of a starting point and we then tweaked various hyperparameters to achieve better precision and recall.

In experiment 22 (Figure 5.12b) we added a high percentage (0.8) of mixup data augmentation which actually resulted in the same F1 score but at a worse precision. Both experiment 21 and 22 were trained over 300 epochs and it may be that by introducing mixup augmentation we need more iterations to achieve a good result or our value was too high and this led to noisy data to our detriment.

We put this to the test in experiment 29 (Figure 5.12c) by keeping mixup augmentation high, also introducing some copy-paste data augmentation and increasing the number of epochs to 600. This resulted in a slightly higher confidence but a slightly lower F1 score overall.

In experiment 32 (Figure 5.12d) we removed the copy-paste augmentation (our theory being that the effect of copy-paste is likely negligible if we have a high value for mixup as they are such similar techniques), kept a high percentage of mixup augmentation and chose to only freeze 8 layers instead of 10. Because we decided to freeze fewer layers we anticipated needing to train for longer (as we are forcing the model to learn more values than with 10 frozen layers) and so we ran it for 1000 epochs. This resulted in a higher F1 score and a higher confidence than our initial experiment (Figure 5.12a).

For experiment 34 (Figure 5.12e) we kept the number of frozen layers at 8, reduced our mixup value significantly (down to 0.5) and added a very small value for copy-paste. This was partially in an attempt to verify our earlier thought that too high a mixup value may introduce too much noise in the data. As with previous attempts with smaller numbers of frozen layers we trained for 1000 epochs. The results show that we kept the same F1 score as our previous experiment but achieved a higher level of confidence.

To see plainly the effect of freezing more or fewer layers we re-ran experiment 34 but with the original 10 frozen layers (which was originally chosen as this is the whole backbone of the model). With all other parameters kept the same the experiment resulted in the same F1 score as the previous experiment but with a slightly lower confidence (Figure 5.12f). This is to be expected since we trained fewer layers of our model.

Our best performing experiment was experiment 34, it shares the highest F1 score of our experiments but it has a higher confidence level which means we should be able to see the same level of performance when in use but with a higher confidence threshold. This would indicate that it is a slightly more reliable model overall.

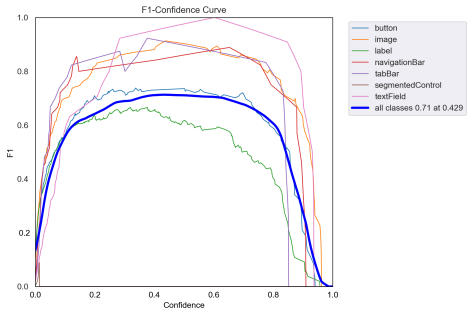
There are trends in Figure 5.12e that apply to all of our experiments. For our individual class performance we can see simply from the shape of the lines that some classes clearly have more data to train on than others. Some lines are contain long straight segments that have quite angular bends (see textField and navigationBar in Figure 5.12e), indicating a class with less examples. Other lines such as button or **label** show much more granularity, or more frequent changes of direction at smaller intervals, indicating there is more training data causing the values to adjust. Conversely the line indicating the segmentedControl class is almost impossible to find on any of the graphs in Figure 5.12, there is simply not enough data here to train on (in fact there are only 7 examples of a segmentedControl in the training data, see Table 5.1).

5.2.4 Converting to CoreML

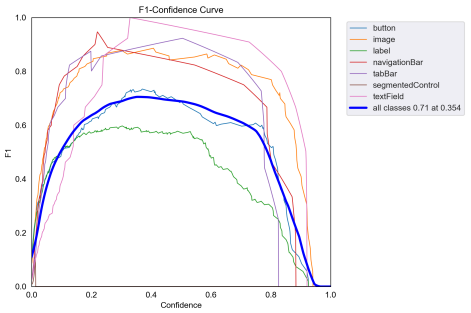
The file format required to run a model on an iOS or iPadOS device is a CoreML Package (.mlpackage). Apple provide documentation on how to convert a PyTorch model to their format (Bove, 2023a), and the authors of YOLOv5 also provide a python script which converts a given YOLOv5 model to CoreML, however there are a couple of problems.

The first is that the code provided by the YOLOv5 authors produces a .mlmodel file, rather than the newer .mlpackage. While this is not a significant problem, the other problem is more significant so we addressed this as part of that solution.

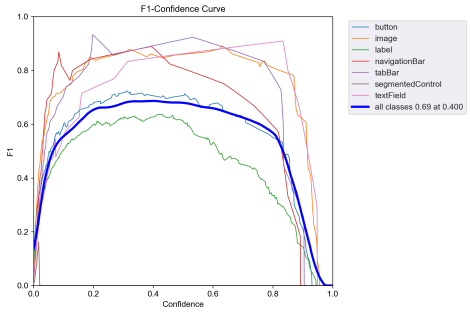
The second, more significant issue is that the CoreML model exported from YOLOv5 does not include a Non-Maximum Suppression (NMS) layer. Whether this was an oversight or the intention was that users would implement NMS separately is unknown.



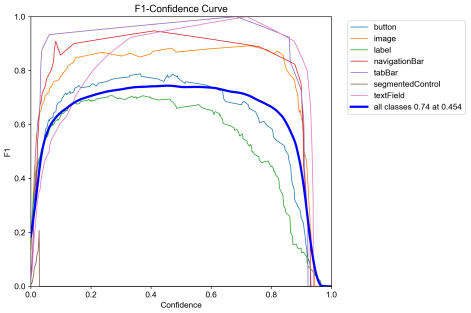
(a) Experiment 21



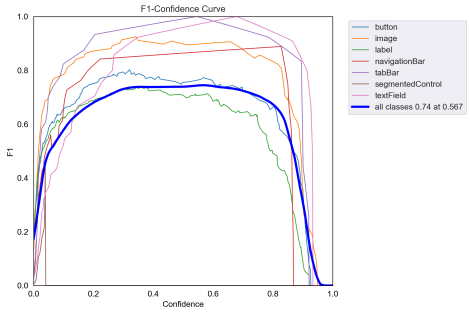
(b) Experiment 22



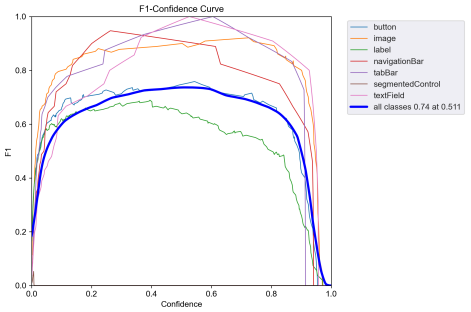
(c) Experiment 29



(d) Experiment 32



(e) Experiment 34



(f) Experiment 35

Figure 5.12: Experiment results

5.2.5 Non-maximum suppression (NMS)

Many computer vision tasks rely on NMS algorithms (Neubeck and Gool, 2006) for filtering out multiple detections. Object detection algorithms, like YOLOv5 and YOLO9000, use anchor boxes (Redmon and Farhadi, 2017) which is a concept introduced by Ren et al. (2016) for Faster R-CNN. These anchor boxes are predefined shapes which represent possible different sizes of objects, this is preferable over a single sized bounding box as objects can have varying size and shape, anchor boxes accommodate this, and is also a faster way to produce region proposals for detection. However, this also means that a single occurrence of an object will have several matching boxes and there may not be a good fit for all examples.

This approach also means there are a fixed number of possible boxes which can be detected in an image. The original YOLO approach used grid cell proposals for detections and this produces a fixed 98 detection boxes per input image (Redmon et al., 2016), YOLO9000 uses an anchor system and produces over a thousand (Redmon and Farhadi, 2017).

No matter the number of boxes the same problem remains so we needed to intelligently determine which boxes (if any) are most likely to be correct in order for the rest of our app to act on that information. This is often achieved using non-maximum suppression and Figure 5.13 shows an example of this.

Figure 5.13: Example object detection output, and after applying NMS (Prakash, 2021)

While we could have implemented NMS in the mobile code base in native Swift code, this would require processing over a thousand possible detections on the iPad CPU. This would likely be a relatively slow process and would potentially endanger our performance goals of near real-time detection of elements - especially as all user input is also handled on the CPU. Instead we chose to create our own version of the YOLOv5 export script which adds an NMS layer to the resulting CoreML model. One benefit of this is that the code is run on the iPad's GPU which is much more suited to this task and should perform more rapidly than the CPU.

This involved creating a non-maximum suppression model which takes the output of our trained model as its input, and itself outputs confidence and coordinates. We then built a pipeline of the two models, which outputs the classification, confidence and coordinates above a given threshold, instead of the full result of possible detections of every possible shape.

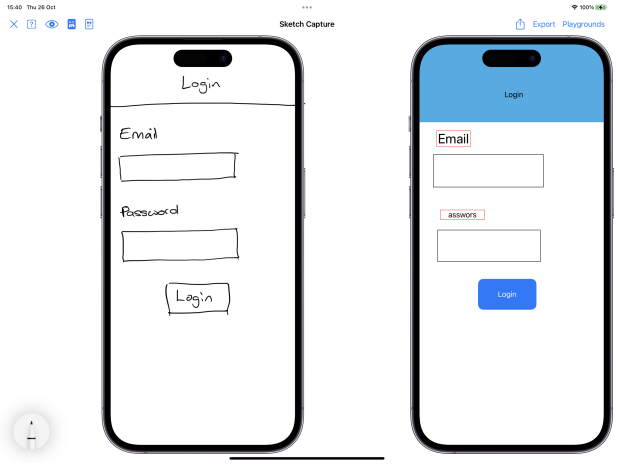
The bonus to creating our own version of the output code was that we were also able to modify it to output a `.mlpackage` file and keep pace with Apple's CoreML requirements.

The final pipeline containing both our trained model and the NMS model is shown in Figure B.1.

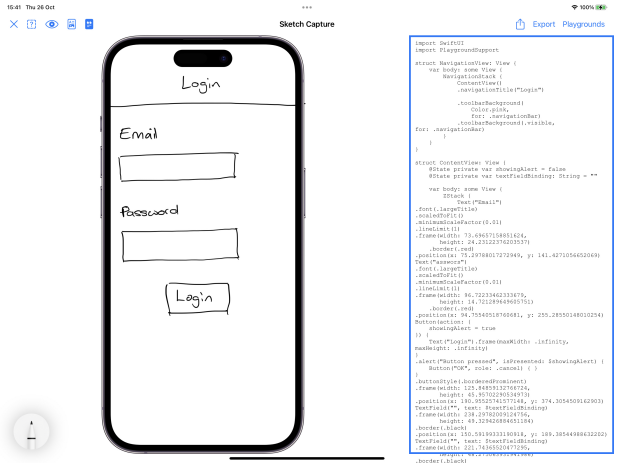
5.3 Drawing app

The drawing app itself was intended to have a similar UI to the data capture app to allow us to reuse much of the user interface code (in particular the drawing code). Although we refer to the drawing app and data capture app as separate apps throughout this document we built both within the same app. This approach allows us to continue to gather more training data while also allowing testing of the drawing and generation features. To access the drawing screens we added a "Draw" button to the user consent screen (5.1.1). The user is presented with a drawing area and a preview area (which is similar in layout to Figure 5.2 but with a new preview panel on the right hand side), as they draw the preview area is populated with the results of our trained model.

Sharing code with the training app screens also allows us to bringing some of the debug tools (i.e. the eye icon at the top left allows us to see the raw model output, without post processing,



(a) The drawing app in "UI preview" mode



(b) The drawing app in "code preview" mode

Figure 5.14: The drawing app layout

on top of our drawing) we created for data capture in to the drawing app. Some of our user interview participants found this particularly useful (6.3.1).

The drawing app has two modes controlled by buttons at the top left of the screen: UI preview mode (Figure 5.14a) and code preview mode (Figure 5.14b). In either mode, as the user draws their desired UI on the left hand panel the right hand panel automatically updates with the results of our trained model (either as UI elements being drawn or by code being generated), this happens in near real-time (our model generates results in around a tenth of a second). A video demonstration of the app is available on YouTube `https://www.youtube.com/watch?v=SKGdZ3H9eyY` (Leivers, 2023b).

5.3.1 Making predictions

As our app's main input is the user drawing using the Apple Pencil, it is even more important than in other apps that we maintain a responsive user experience. If the user is attempting to draw and the device resources are overloaded the user will see a lag in their drawing with the Pencil/user moving further ahead of the rendered line as the device struggles to catch up with the drawing.

To prevent this happening we maintain a low memory footprint and move intensive work off to background threads (where possible). In iOS and iPadOS the main thread is responsible for receiving user input and drawing to the screen, as such, we want to keep intensive work off the main thread as much as possible. While our machine learning model will be run on the GPU, rather than the CPU, the memory pool is shared between the two (Apple, 2023c) so keeping on top of memory usage is crucial to maintaining a responsive app experience.

When a user is drawing a line the app does nothing until the Pencil leaves the screen (i.e. the line being drawn is finished), at this point we know the drawing has changed and the user has either finished or will continue drawing. We use this point to trigger our model and the next steps are shown in 5.1.

Listing 5.1: Queuing drawing changes for inference

```
func queueRecognition(canvasView: PKCanvasView) {
    currentDetectionWorkItem?.cancel()
    let newWorkItem = recognizeWithWorkItem(canvasView: canvasView)
    currentDetectionWorkItem = newWorkItem
    DispatchQueue.global().async(execute: newWorkItem)
}
```

The first step is to cancel any currently running inference task (this would be any ongoing inference task from the last line that was drawn, the user may have drawn a new line before the previous output was processed). We then create a new work item with a reference to the function to be called (this function is passed a reference to the drawing canvas), and assign this new work item to a variable to allow us to cancel it in future if needed. The work item is then triggered to run on a background thread.

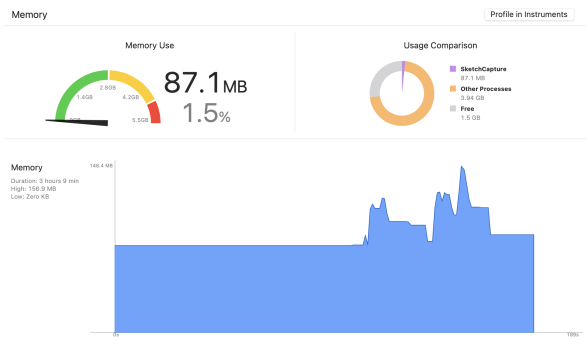
This approach ensures we are only triggering a single inference task at any point in time, saving as much wasted processing time as we can and keeping the required memory low.

The function call recognizeWithWorkItem(canvasView: canvasView), is where we capture an image of the drawing canvas on a white background (in the same way as when pre-processing our data for training in 5.2.2), we pass this image to our trained model and await the results for processing.

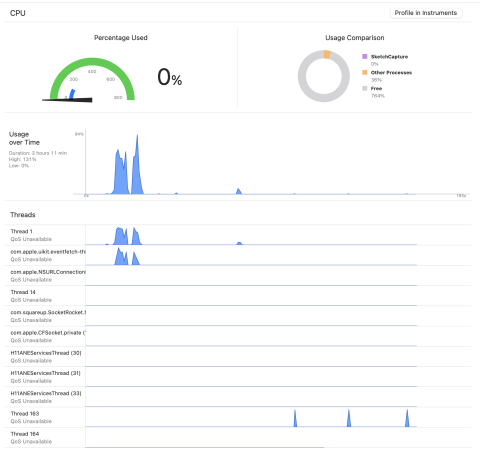
Our app successfully maintains a low memory footprint (generally under 100MB, see Figure 5.15a) and while CPU usage does spike occasionally (Figure 5.15b) it is very manageable and generally no slow down or lag is perceptible by users. Some of these spikes in CPU usage are simply due to the model being loaded in the memory the first time it is used. This manifests in a small delay on the first few detection when in use. This can further be seen in the timings in Table A.5 where the first few detections are significantly slower, at a little under 3 seconds, before dropping down to under 200 milliseconds. We suspect we could reduce the impact of this on the user by calling the model with a blank image as soon as the screen is shown or finding another method of "warming up" the model.

We recently came across documentation from Apple (2023h) which allows a callback when loading of the model is complete, which may be an avenue to explore in future. The iPad device used was a iPad Pro (12.9-inch) (4th generation), at the time of writing the latest iPad Pro is the iPad Pro (6th generation).

If our call to the model for inference is not cancelled it will output a list of bounding boxes representing a detection and classification for each (i.e. this is the output of the non-maximum



(a) Drawing app memory usage



(b) Drawing app CPU usage

Figure 5.15: The resource usage of the drawing app

suppression model). We process these bounding boxes both to create a UI preview and generate SwiftUI code for export.

The processing of the bounding boxes first involves postprocessing the model output. The detections are received from the model as an array of Apple's VNRecognizedObjectObservation objects, we convert these to our own BoundingBox class which contains helper functions and additional attributes.

We then look for overlapping detections in non-container types (i.e. a navigation bar may contain a title and some buttons so there is an overlap implicit in this type) and group them in a list (i.e. a pair of overlapping elements would be one element in the list). We then iterate over each group of overlapping elements. If the elements overlap more than a specified heuristic (i.e. 85%) then we treat this as an artefact of the model output and we expand the bounding box with the highest confidence value to the size of both boxes and keep that. If the overlaps are less then the heuristic we only keep the bounding box with the highest confidence value and simply ignore the other box.

The resulting bounding boxes are then further processed with attributes of the UI element they have been classified kept in mind. This is achieved by filtering the set of results by each classification, applying our post processing and adding the results to a final list and then repeating for each class.

Postprocessing UI element detection

For bounding boxes representing either a tabBar or a navigationBar we check the number of detections, there can only be one of each class in a screen so if we have multiple results of each we must decide which ones to keep. We again achieve this by using the detection confidence, the highest value is kept and the bounding boxes of the other detections are merged resulting in a single tabBar or navigationBar that has the classification, confidence and other attributes of the highest confidence result but the bounds of the union of all detected tabBars or navigationBars.

For the navigationBar we further check for any other detected elements of any classification which have their bounds inside the navigationBar. Of these detections we then look for the

left-most item, the centre-most item and the right-most item, everything else is discarded. While any of these may not be present if they are we use them to infer the configuration of the navigationBar:

- Left-most item - if present this indicates the navigationBar should have a back button. Apple (2023b) states that best practice is to use the standard back button, which when used in the "Standard title" size (which we replicate here) is always on the left in left-to-right languages (it's reversed for right-to-left languages). Note that we only look for presence or absence, though what is drawn may make the model's confidence higher or lower (for instance if the user has drawn an actual back button that may increase the chances of detecting the whole navigation bar).
- Centre-most item - While we assume the navigationBar has a title and default it to the text "title", if this item is present then we attempt to recognize the text the user has written (see section 5.3.1) and use that in place of "title".
- Right-most item - If present this indicates the navigationBar should have a right bar button. If something is drawn here we always show a "Done" button regardless of what the user sketched.

The various combinations of user drawings and their effects on the navigationBar output can be seen in Figure 5.16. This information is then stored in the bounding box object that represents the navigationBar.

(a) Title only

(b) Title and back button

(c) Right bar button only

(d) Back and right bar buttons

(e) Back button, title and right bar button

Figure 5.16: How the user's drawing affects the resulting navigation bar

Any bounding boxes representing **label** or button classifications are passed to handwriting recognition (see section 5.3.1) to attempt to extract any text that may be present. Users can represent text either by writing the text our or using one or more wavy lines (which is generally accepted to represent placeholder text in prototypes). If any text is present it is stored in the bounding box object for later use.

Handwriting recognition

Handwriting recognition or text detection was never explicitly one of our goals, while we were absolutely needing to detect a **label** UI element in a user's drawing, which may take the form of a wavy line or a piece of handwritten text, we were not necessarily expecting to attempt to determine the content of that writing. However, during development it became clear that being able to extract written text and use it in the generated previews would enhance the user experience.

Rather than training an additional model in the task of handwriting recognition we decided to use something "off the shelf", specifically Apple provides a text recognition capability as part of its iOS and iPadOS Vision framework (Apple, 2023j).

To use the built in text recognition we crop the sketch image down to just the bounding box of the desired component. That image is then passed to a VNRecognizeTextRequest object on a background thread and the results returned when complete.

The results are not always completely accurate. The Apple model may well not be trained on handwriting at this scale or any number of other reasons and so the images you see in this report may show incorrect text. It was decided that having this feature, even if not fully accurate, was better at demonstrating how this prototype application could work than if the feature was not present.

5.3.2 Preview generation

Initially we had hoped to find a route to generate a SwiftUI string and interpret it at run-time to form the preview UI to the user. Unfortunately while SwiftUI code can be encoded as a string it is still fixed at compile time and cannot be swapped out in quite the way we initially intended. However, given we support a defined set of UI elements and are providing a tool for low fidelity prototypes (and not production quality screens) we were able to create a layout engine which creates a preview UI from the bounding boxes our model outputs.

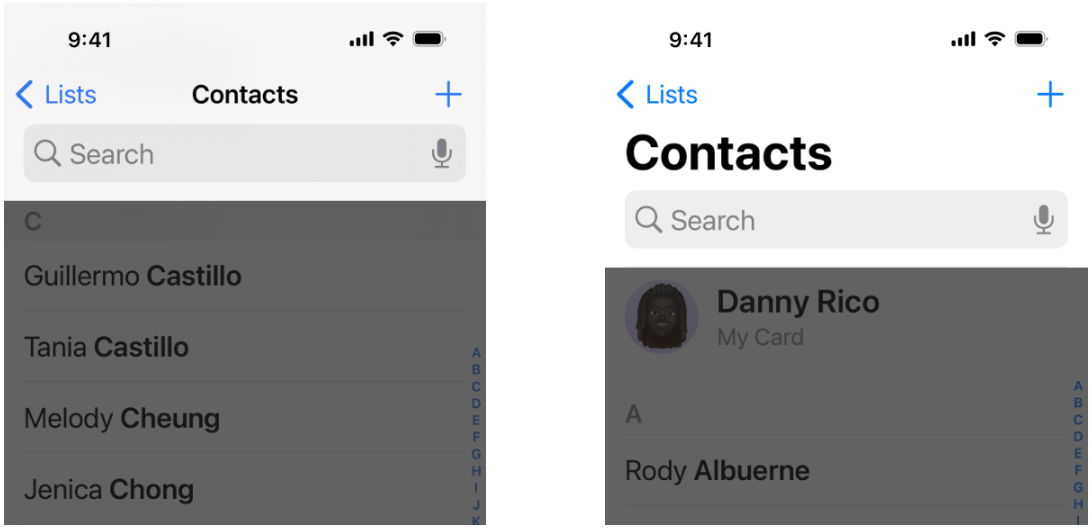
To create the UI preview we iterate over the list of bounding boxes (which contain everything needed thanks to the process described in 5.3.1) in a similar fashion to Robinson (2019), for each box we check the classification and pass it to a matching function for display. Each classification has its own function responsible for the rendering of that particular user element type, this allows us control over some specific tweaks.

Navigation bar

The navigation bar is a top level element in SwiftUI, it must sit at the top of the view hierarchy on the current screen. Unfortunately as our drawing app screen is written in SwiftUI it was extremely challenging to try to force a NavigationBar to render in child view in SwiftUI. We didn't quite find a solution we felt was of a high enough quality, instead we decided to create our own custom SwiftUI view which replicates the layout of a NavigationBar.

When creating our own NavigationBar we had to decide on a style to replicate, Apple's SwiftUI NavigationBar uses the "large style" by default (Figure 5.17b). However, very few of the designs we gathered for data capture (5.1.2) outside of Apple's own apps use this style, instead they generally use the "standard style" (Figure 5.17a) or their own high fidelity design equivalent. Our implementation of a NavigationBar can be seen in Figure 5.16.

The bounding box representation of a NavigationBar includes whether the bar has a back button or not, the text to show in the title (if any) and whether or not a right bar button is present. If the bounding box does not indicate any title text we simply default to "title" and always show a title, if it does include it then we use the included text.



(a) Standard style (b) Large style

Figure 5.17: Apple's navigation bar styles

Tab bar

Similar to the navigation bar (5.3.2) tab bar is an element which must wrap its contents as children and does not easily scale down to fit inside another container so we created our own custom SwiftUI view to represent a tab bar.

Label/Text

When we position a text element on the screen we use the detected text from the postprocessing stage (5.3.1) if present and scale the font to fit the text in to the detected element size. If there is no detected text (which may occur if the user draws wavy lines to represent a block of text) then the box is filled with multiple lines of *Lorem Ipsum* at a fixed font size until the display size of the box is full. Each of these situations can be seen in Figure 5.18.

Button

Positioning a button element is relatively simple. We use the bounding box as the size of the button itself and any text detected from postprocessing (similar to a label) is used as the button title at a fixed font size. If there is no text detected then a placeholder string is used as the title. The button is added as an actual SwiftUI button and is fully interactive, triggering an alert when pressed.

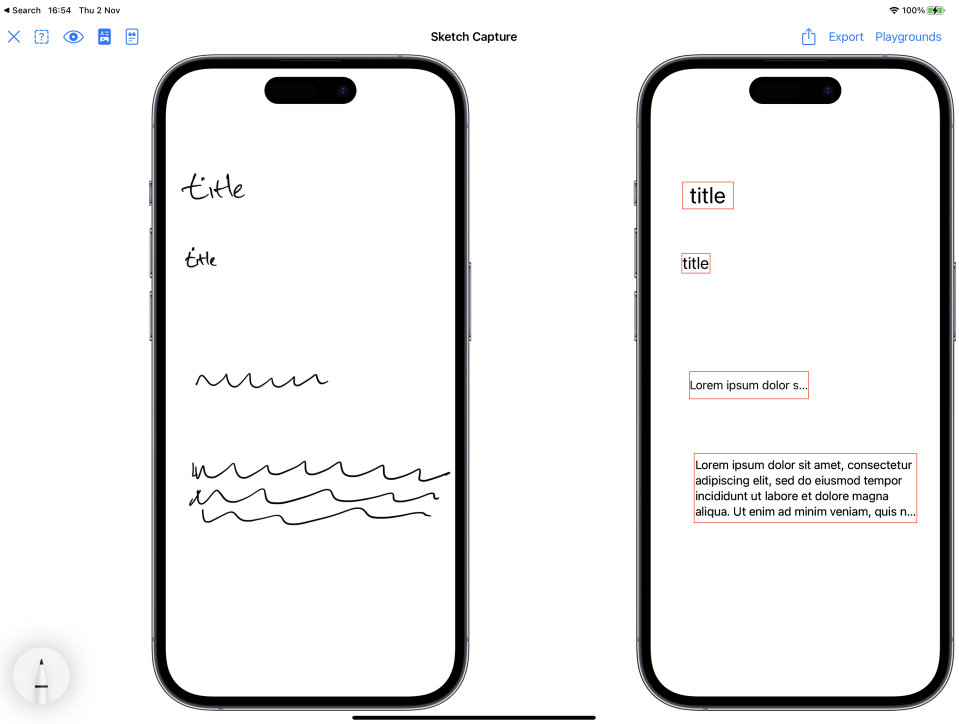


Figure 5.18: Examples of how labels are displayed in the preview panel

5.3.3 Code generation

As with the preview generation (5.3.2), and similarly to many of the approaches discussed in Section 2.6, we use the post processed bounding boxes generated by our model to generate our code.

How UI elements are represented in SwiftUI varies depending on the element itself so we sometimes have to generate significantly different code depending on the element detected. The biggest changes come from navigation based elements (NavigationBar and TabBar) as they essentially act as containers and must have the other UI elements as their children. For instance if we compare the code for displaying a simple label, 5.2, with the code for displaying that same label in a tab bar, 5.3.

Listing 5.2: Rendering a label

```
struct ContentView: View {
    var body: some View {
        Text("Test")
    }
}
```

Listing 5.3: Rendering a label in a tab

```
struct TabContainerView: View {
    var body: some View {
        TabView {
            ContentView()
            .tabItem {
                Label("Menu", systemImage: "list.dash")
            }
```

```
            ContentView()
            .tabItem {
                Label("Order", systemImage: "square.and.pencil")
            }
        }
    }
}

struct ContentView: View {
    var body: some View {
        Text("Test")
    }
}
```

When creating UI previews we handled tab bars and navigation bars by creating our own representations of them. However for our output code we create "native" UI elements as per Apple's SwiftUI documentation (Apple, 2023k).

To translate the bounding boxes into SwiftUI and maintain this hierarchy we baked in a set of template strings for each UI element we support. Each of these strings is then tweaked with the information detected by the model and our post processing, including position, size, text, etc. More primitive UI elements such as buttons, labels and images are built up in to a ContentView, similar to 5.2. We then check for navigation elements and, if present, we reference our ContentView inside them.

For buttons we include an action that triggers an alert to appear onscreen when the user presses them, thus creating an interactive user interface. Similarly text fields can be tapped on, the keyboard appears and text can be added.

This generated code is recreated every time the model outputs new bounding boxes. This allows the user to select the "code preview" mode in the app (5.14b) and visibly see the code change as drawings are added or erased.

5.3.4 Performance

We refer to performance throughout this dissertation and in most cases we are referring to the accuracy of the model or some other measure of the ability to correctly detect and classify elements. However, in this subsection we discuss the speed of the model and our code generation. How quickly does it run? Is an iPad a device that is capable of delivering feedback quickly to the user? One of our early concerns was that the iPad itself would not actually have enough resources to run our model and that we would need to access a more capable computer via a web server or other mechanism to get the results of the user's drawing.

When training early versions of our model we were pleased to find we could achieve a good level of performance entirely on the iPad. Table A.5 shows the timings captured while drawing the sketch shown in Figure 5.19. We captured two main values; the time taken to detect and classify elements and the full time taken from user sketching to code generation. While the former is a subset of the latter its useful to have a sense of the complete time taken as that is what the user experiences.

The time to detect and classify elements averages 0.66 seconds and the time to detect, classify and generate code averaging 0.8 seconds. Whilst this is not "real-time", which would be hard to achieve given the iPad Pro screen refreshing at 120 frames per second (Apple, 2017) which results in rendering a frame every 8.33 milliseconds, it is fast enough to provide the user a sense of response or feedback. We consider this nearly real-time and occasionally refer to the performance as being "near real-time" in this dissertation.

One of systems which also captured sketches digitally, Mohian and Csallner (2020), measured the timings on their approach. Their average time to detect, classify and output Android app code was 526ms on their development machine and 94ms on an Amazon EC2 runtime. While this is faster than our own timings we must bear in mind that the hardware being used by Mohian and Csallner (2020) (a 16 GB RAM 64-bit Windows 10 machine with a 2.20 GHz Intel i7-8750H CPU and an AMD64 Ubuntu 16.04.5 Amazon EC2 t2.micro instance) is significantly more powerful than a 4th generation iPad Pro.

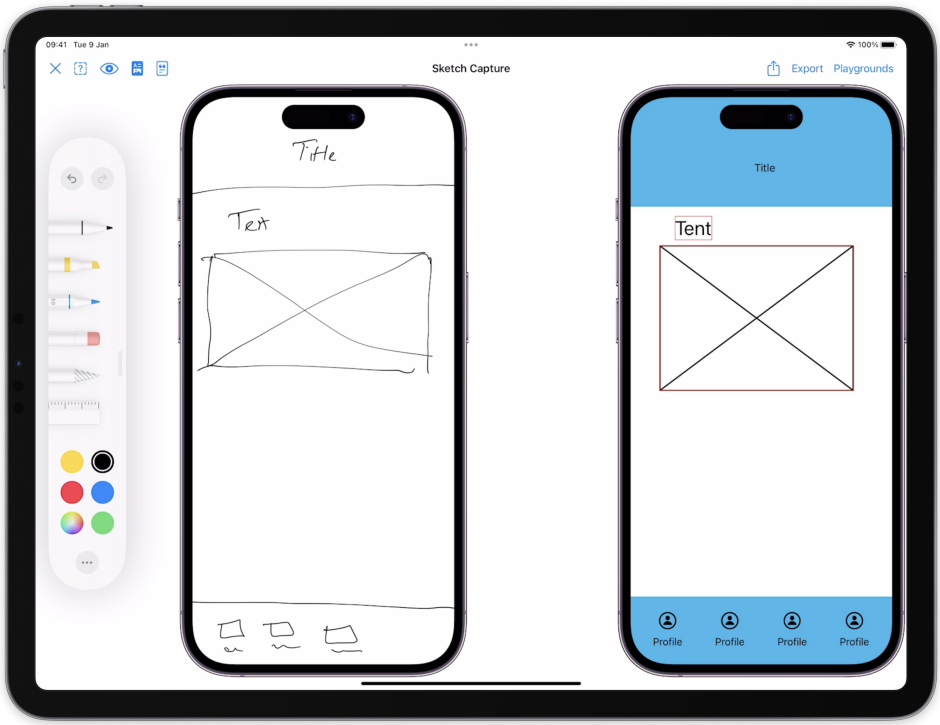


Figure 5.19: A sketch drawn to gather timings from the model running

5.3.5 Exporting

One of our goals is to create code that is then useful as a starting point for developers intending to take the generated prototype further, or even for users to be able to tweak on device to achieve their goals (for example styling). To achieve this the users need to be able to export the code so that it can be used in other applications.

The two main applications that are likely to be useful to our target audience are Xcode (Apple, 2023l) and Swift Playgrounds (Apple, 2023g).

Xcode is an application exclusively for Apple Mac computers and is generally used for building macOS, iOS and iPadOS apps. To allow our users to open their generated code in Xcode we

will have to provide a way of exporting the code to a file and then sharing that code in a way that their Apple Mac can access.

Swift Playgrounds is an iPadOS app, it allows users to run code on their iPad in a sandbox and while its capabilities are somewhat limited (Chin, 2021) it does support compiling and running SwiftUI code.

Luckily both applications share a file format they are both capable of opening - `.playground`.

The `.playground` file format

Unfortunately Apple have not published any details of the `.playground` file format or structure. Presumably they don't expect third parties to be exporting files in this format. We spent some time attempting to replicate the file format and discovered that it is little more than a zipped up collection of Swift files and project files (which could be found in any Xcode project), however it must contain a "specific type identifier" in order for the Swift Playgrounds app to identify the file. We resorted to reverse engineering the Swift Playgrounds app itself in order to retrieve the type identifier, this process was documented in a blog post (Leivers, 2023a).

With this information we were then able to save our generated code to either the iPad local filesystem or to the user's iCloud Drive (Apple, 2023f) as shown in Figure 5.20b. Saving to either of these enables Swift Playgrounds, running on the same device, to open the file and run the code (Figure 5.20c). Saving to iCloud Drive provides a convenient way for the user to save the code and open it on their Apple Mac computer in Xcode (Figure 5.20d) from which the user can run the code on an iOS device or the iOS Simulator (Figure 5.20e.

Making the workflow on iPad more seamless

While both applications are certainly useful to our users, Swift Playgrounds potentially offers the most seamless experience. The user can sketch a design in our app, see the results, export the code to Swift Playgrounds and tweak it all on the same device. To make this process as seamless as possible we wanted to be able to open the Swift Playgrounds app from a button press in our own application (rather than the user having to leave the app to browse the home screen and look for the Swift Playgrounds app icon). As with the `.playground` file format, Apple has not published any documentation about how this might be achieved. However, as we were already extracting information from the Swift Playgrounds application we were also able to locate the "custom URL scheme" used by the application.

A "custom URL scheme" is a unique URL an app registers with the operating system (OS), when this URL is opened from another application the OS then knows which app to send that request to. This can be seen with common URLs such as `https://` and `mailto://`, when these are actioned in iOS or iPadOS they trigger the Safari and Mail apps respectively. We were able to discover that Swift Playgrounds registers the URL scheme `x-com-apple-playgrounds://` and so we added a button at the top right of the screen called "Playgrounds" (Figure 5.20a) using the following code:

```
Link("Playgrounds", destination: URL(string:
    "x-com-apple-playgrounds://")!)
```

When a user taps this button it triggers the OS to open Swift Playgrounds. The intention is that the user can save their project out with the "Export" button, then press the "Playgrounds"

button to launch straight in to Swift Playgrounds with as little interruption to their workflow as possible.

5.4 Conclusion

To be able to build the drawing app we had to create several components first. These included a data capture app running on the iPad which was used to collect sketches from users and store them on a server for use in training. This data capture app contained tools for labelling the user data as well as for debugging.

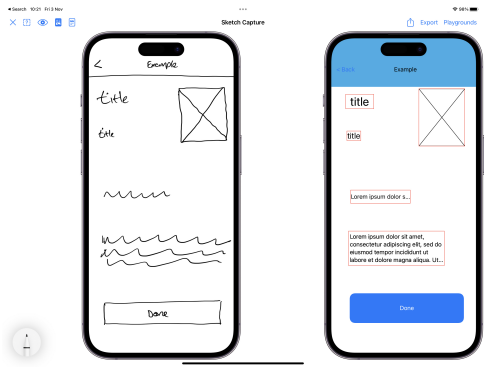
Once we had collected enough data we were able to preprocess it and then use it to train a model. We would repeat this process several times to experiment with hyperparameters and data augmentation techniques to try and improve our detection and classification accuracy. We also found we needed to add a NMS model to our pipeline in order to filter a large number of possible predictions down to only the ones which have a high enough probability to be worth using.

Having converted our model to a CoreML format we were able to then use it in our drawing app. In developing the app we reused parts of our data capture app codebase, such as the drawing canvas, and built a preview panel to display the system's interpretation of the user's drawing as well as the generated code.

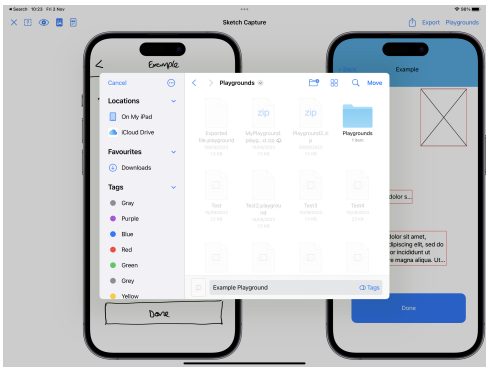
In this phase we paid particular attention to the usage of system resources, keeping memory usage as light as possible and attempting to keep usage off the main thread. This allowed us to keep drawing smooth and responsive while still generating responses to the users drawing frequently enough (whenever a stroke was completed). We were also able to post-process the predictions from the model, allowing us to combine our knowledge of the convention for some UI elements with the predictions - for example a NavigationBar can only appear once per screen and only ever at the top edge.

In lieu of being able to compile code directly in our app (which isn't permitted by Apple), we built a method to export our generated code and allowed users to open it in the Apple Swift Playgrounds app (which is allowed to compile Swift code). This involved some reverse engineering (see Section 5.3.5) but provides a reasonably seamless experience for users.

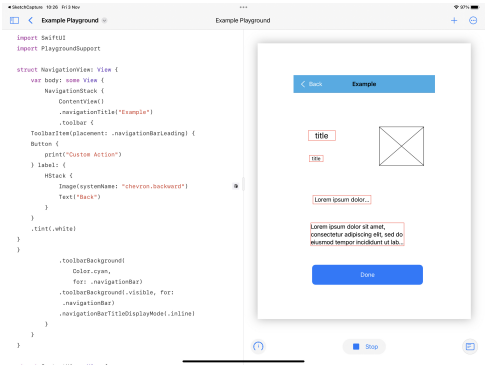
In the next chapter we discuss the results of our app with users.



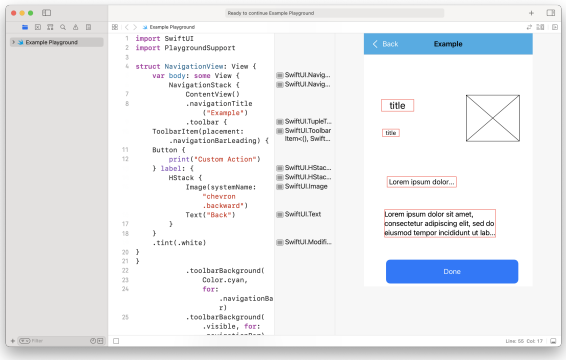
(a) The drawing we will export



(b) Saving our exported code



(c) Our exported code running in Swift Playgrounds on the iPad



(d) Our exported code running in Xcode on an Apple Mac computer

(e) Our exported code running in the iOS Simulator

Figure 5.20: Exporting the code

Chapter 6

Results

6.1 Introduction

In Section 2.8 we found that many authors conducted a user survey to evaluate the performance of their system.

This chapter presents the findings of own user interviews where participants were presented with a demonstration of the drawing app and then asked to try creating a screen using the app. This is in contrast to Adefris, Habtie and Taye (2022) where users were shown output from their app and asked questions, we thought it would be valuable to get hands on feedback. There were six participants in total, all of whom work in the field of mobile app development in roles including developers, UI designers, UX designers and product managers. Due to the breadth of roles some questions chosen are more appropriate to designers and some more appropriate to those who work with programming languages. Some participants were involved in the creation of training data and some had no prior exposure to the project.

6.2 Questions

Questions to participants:

1. Rate the overall usability of the prototype. (Out of 5)
2. Are there specific features or functions that you found easy or challenging to use?
3. Do you have any suggestions for improving the user interface for better usability?
4. Were you able to achieve your goals efficiently using the prototype?
5. Did the prototype help in quickly transitioning from sketching to building UIs?
6. To what extent did the prototype improve your productivity in the design process?
7. How seamlessly would the prototype fit into your existing design workflow?
8. Does the prototype complement or disrupt your current design process?
9. Do you have any suggestions for making it more compatible with your workflow?

10. Does the prototype accelerate the prototyping phase compared to traditional pencil-and-paper methods?
11. Was there any time saved or efficiency gained from using the prototype?
12. Evaluate the quality of the code generated by the prototype.
13. Do you have any feedback on the generated code's clarity, correctness, and/or adherence to best practices?
14. Can you customize the generated code to meet your specific design requirements?
15. Does the prototype offers enough flexibility to accommodate various design styles and preferences?
16. Did you encounter any challenges while using the prototype?
17. Would you recommend the prototype to others?
18. Do you have any recommendations on how the prototype can be enhanced or expanded?
19. If you have experience with other prototyping or UI design tools, could you compare the prototype to those tools in terms of advantages and disadvantages?

The raw results can be seen in Appendix A.1.

6.3 Themes

The analysis of the participants' answers revealed several similar or overlapping comments. From the raw results data (Appendix A.1) we counted occurrences of comments or requests that were common in each interview. Once we had a list of comments that were mentioned by multiple participants we looked at which of these overlapped or could be grouped (for instance a participant might mention that they thought they needed an onboarding or instructions screen and another participant might have mentioned they did not know where to start so these have an overlap or common theme). These identified groupings or themes are discussed in detail below.

6.3.1 Theme 1: Detection and classification

Of the participants, half found that the detection of their sketches was not as accurate as they desired. Participant A commented that "Hand drawing is pretty intuitive, the recognition can be a bit off sometimes" and Participant F cited "detection issues" as being a concern. Some participants found that the model either did not detect the object they had drawn, detected the wrong type of element or drew the element at the wrong size.

Participants who encountered fewer or no detection issues like Participant C said "Very easy to draw diagrams". Participant E agreed with this, saying "drawing was easy and responsive. Generation of prototype was quick to pick up what I had drawn".

However, Participant D said that they "didn't know what components it would or wouldn't recognise, or what the limitations were". They also went on to suggest we could "expose the debug view of what the AI is seeing to the user" to help see what might be the problem. The debug view is shown in Figure 6.1, when enabled this shows the bounding boxes generated by

the model along with the classification and the confidence (i.e. the raw output of the NMS 5.2.5). This shows the model output before we post process it, helping to identify bugs/issues with out post processing code or highlight incorrect classification.

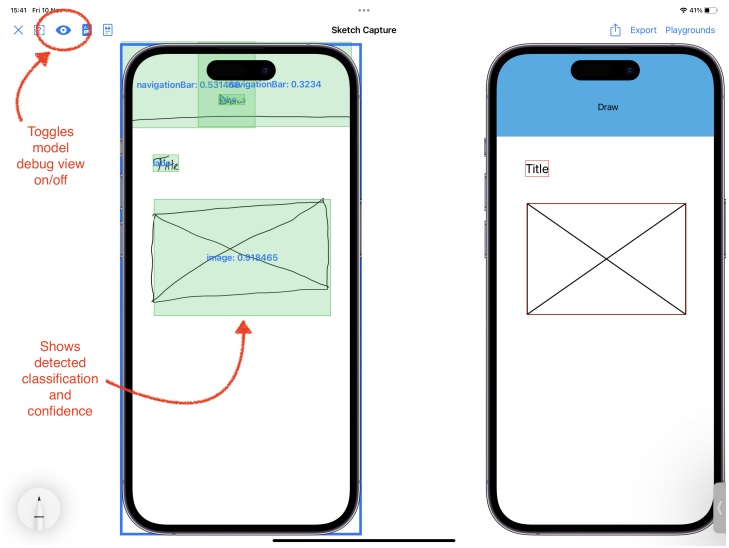


Figure 6.1: The drawing app with the model debug view enabled.

Overall even the participants who encountered an issue could see the potential in the tool, going on to have a generally favourable opinion of the app. Every participant stated they would recommend the app to other people to try.

6.3.2 Theme 2: Correcting

While some participants were very familiar with the iPad and the Apple Pencil and knew how the stock toolbar (which contains an undo button, redo button and an eraser tool) worked, others had little or no experience with this. If things went wrong with detection they were unsure how to correct either their own mistakes or the app's mistakes.

Participant A was more familiar with the iPad and Apple Pencil combination and experienced some detection or classification issues. They stated that "object recognition and not having the option to classify or correct the model yourself is the biggest pain point. You have to just rub it out and try again.". However, other participants like Participant D struggled, "when it did things wrong I didn't know how to correct it".

Other participants did not notice the eraser tool but found the undo function and Participant F was one such user who summarised "because of the detection issues the undo function plays such a crucial role".

6.3.3 Theme 3: Onboarding

Half of the participants suggested there was a need for an onboarding screen to be shown to the user before they started drawing for the first time. Because the app design - when empty or free of existing sketches - is simply two frames of an iPhone side by side they were not sure where to draw without being given some instruction upfront.

Participant B suggested that we "split the screen better to be clearer on what is the sketching space and what is the preview space, Figma does this well.".

Some participants like Participant F had a different reason for suggesting an onboarding screen as they were looking for a guide to what could be drawn. While referring to the demo from the author they stated "I liked how you showed me what you could draw, that's how I would have drawn an image but some proper onboarding might be good". This echoes Participant D's comment from Section 6.3.1 that they "didn't know what components it would or wouldn't recognise, or what the limitations were".

The other half of participants had been involved in the data capture stage so the layout of the app was certainly more familiar - the drawing app was designed to have a very similar user interface to the data capture app (Figure 5.2) as discussed in Section 5.3. This experience had perhaps also given them some familiarity with what the capabilities of the drawing app might be.

6.3.4 Theme 4: Drawing and responsiveness

There was strong positive feedback from half of the participants on the ease of drawing and the responsiveness of the app. Participant E mentioned that "drawing was easy and responsive" and Participant C echoed this saying it offered "Very easy to draw diagrams".

The only real counter to this was Participant B who commented that "there's a bit of a delay when generating/recognition", however we speculate whether this was simply the initial loading of the model which we discuss in Section 5.3.1.

6.3.5 Theme 5: Transitioning to code

When asked whether the app helped the participants quickly transition from sketching to building UIs, five of the six users stated that it it did. One of the designers, Participant F, stated "given the code output it's actually really useful as I can't write any SwiftUI right now so this gives me a starting point".

While no designers, nor product managers, felt comfortable giving responses to questions about the quality of the generated code, all participants felt they could edit the code to tweak the output. This is a significant statement as our experience would generally suggest that non-developers are usually reluctant to edit code. We speculate that the tool generating the code from sketches makes it feel reproducible to the participants, that if they broke something they could just draw it again and quickly replace it.

Of the participants who did feel comfortable commenting on the code quality, they were generally happy. "Although it generates absolutely positioned code it gives a good starting point for a junior developer to be given the code and run with it." said Participant A. Participant C agreed "whilst not perfect, it is human readable making it easy to make tweaks to fix it.", as did Participant E who stated "it's basic but does the job".

6.3.6 Theme 6: Theming

Participant E stated it "would be good to add colours and other design elements (e.g. chosen images)" and they were not alone as others expressed a desire to set the colour that elements are generated in. Participant A suggesting that "having support to add colours from the client's brand in quickly would be a massive selling point".

We don't necessarily consider it a failing of the app that it doesn't offer this ability, after all we are only attempting to generate low fidelity designs, rather that participants are perhaps keen to have the app generate slightly higher fidelity designs and maybe find more use-cases where this tool might be appropriate. Participant A also suggests "for prototyping and getting stakeholders to understand it would save hours of work.".

Participant B, who is a designer, seemed to understand this distinction, stating it "doesn't offer enough flexibility for brands or if you wanted to do high fidelity, but it offers enough flexibility for wireframes".

6.3.7 Theme 7: Other tools and user workflows

When asked to compare the app with other prototyping tools, four of the participants were unable to make a comparison. Participant F said "it's not a comparison I can make because there's just nothing like it, it's kind of like no-code swift", similarly Participant A stated " not similar enough, other design tools are things like Adobe XD or Figma which offer a different feature set".

Many of the users (Participants B,C and E) suggested it would be useful to export the prototypes to other design tools like Figma or Sketch. While this is out of scope of what we could achieve it is interesting to see the participants suggesting ways to fit the tool in to their current workflows.

We discuss what this may mean further in Section 7.5.

6.4 Conclusion

While we purposely included some questions that we expected to only be answered by developers and some only by designers, we were pleased to get a crossover with some participants. For designers, the app's ability to generate code gave them a starting point which they felt they could replicate if they broke something resulting in greater confidence to experiment with the code and a willingness to "figure it out". For developers, because the goal isn't to create a "high fidelity" prototype, they felt more open to experimenting with designs and more confident to show their results to colleagues or stakeholders where they might not have done with traditional tools as they may have felt their drawing or design abilities were not good enough.

Overall, all participants regarded the app favourably, despite any issues they may have faced. The most common issue was around detection and classification with some participants having issues with the generated results of their drawing being incorrect.

We must also acknowledge that this is a small sample size, future work should look to survey a larger group in order gather more feedback enabling us to draw stronger conclusions.

Chapter 7

Discussion and critical reflection

In this chapter we reflect on the components of the system that we have built, how users interacted with them and their feedback on it. We consider what parts of the process were a success and what parts were less successful.

7.1 Data capture tool

The data capture tool worked well and was a convenient way to record user sample data. As it was built on an iPad we were able to conduct data gathering sessions with users at multiple locations without being restricted to a computer in an office or a bulky laptop. However, this also meant that sketches had to be gathered in person. While we knew a number of potential users who own Apple Pencils and iPads, which could have allowed for distributing the application wider and recruiting volunteers via the internet, the need for the author to be involved in the labelling process made this unviable.

Our Firebase cloud solution proved to be a convenient way to store and pass the data back to another machine for training. There was no need to have the iPad available to do this. We were able to use Python to create a script to download the data and pre-process it prior to training without any real complications. This proved to be a convenient, flexible approach that we would consider using in future projects.

The main issue with the data capture tool was that it was more difficult to gather as much data as we hoped and not for the reasons we might have expected.

While the data capture tool we created assists greatly in allowing for sketching, grouping/splitting drawings into bounding boxes and classifying each element and all these tools work well, the actual selection/splitting/combining/labelling process takes time on every sketch. It also had to either be explained to the current user or the device had to be passed back to the author for labelling and then passed back to the user again for more sketches. All of this took time and potentially limited the amount of sketches we could gather in a session with a user.

In addition to this we started to notice that different users label the sketches in different ways which led to inconsistencies in the data. For example, when labelling a navigationBar (which potentially contains a title, back button and a right hand bar button) should the elements contained within it be labelled separately or as part of the navigationBar? Do we want the model to recognize the whole combination? If not then do we only have a horizontal line to

give the model as training? We decided that the navigationBar should be the whole area drawn by the user, including any buttons or labels they added to it, but in general the answer to this is that we should simply choose one approach and keep that approach consistent in the labelled data. This also highlighted another issue around the labelling of data as some users of the data capture tool may not know the difference between different elements. For instance, they may have confused a navigationBar with a tabBar. Both of these scenarios result in the same issue of incorrectly or inconsistently labelled data. However it is difficult to know this up front and even more difficult to communicate all of these caveats and nuances to users if we give them the task of labelling the sketches themselves. This led to more onus on the author to handle the labelling of user sketches and having to be present in person for every data gathering session.

One future solution to this may be to create a set of comprehensive onboarding and instruction screens to explain how to label data correctly, though this would likely be a turn off for users. Another solution may be to separate out the labelling from the sketching, allow users to draw their assigned screen, it gets uploaded to the server and then the author uses the tool download these unprocessed sketches to split and label the data asynchronously.

We also found that this inconsistent labelling was a problem with some of our early captured data and retrospectively changing this was difficult. If a simple classification needed changing this was a simple but tedious process (as browsing our tree structure in Firebase to manually change data is time consuming). However if the bounds of an element needed changing this became extremely difficult as we do not have a user interface for this, instead we had to calculate values by hand. When the latter case occurred it was easier to delete this data and capture more instead of attempting to edit it.

In an effort to make it easy for users to get started with creating training data we display sample screenshots of an existing app alongside the drawing canvas. This is an evolution of the approach taken by Baulé et al. (2021) who present their volunteers with screenshots of apps and then ask them to draw them on paper. The user then copies that user interface as a low fidelity sketch. We did not encounter users who were stuck on what to draw, they were all able to just get going, as the interface was mostly just "drawing" there was little barrier to entry. Different apps, however, treat their design in different ways. Sometimes a design has buttons which look identical to labels, i.e. there's no styling or perhaps a colour difference between a label and a button, and this makes it hard for a model to learn the difference between a label and a button. Particularly if pre-processing involves greyscaling the image or otherwise removing the colour. This happens with many UI element types, a button may be just text, underlined text (similar to a hyperlink) or a fully designed, bordered button with a fill colour or gradient – should these all be labelled as simply a button? In future versions we could consider separating out more types, for our button example we might have "bordered button", "underlined button", and so on.

Our approach of showing users a screenshot of an existing app had another drawback - we ended up with an inconsistent number of examples of UI elements drawn by the user. UI elements such as navigationBar or tabBar only occur once on a given screen, however elements like buttons or text labels may occur multiple times in the screenshot. There is also no guarantee a screenshot contains an example of each type of UI element we want to support, in fact most do not and so some classes naturally resulted in having fewer examples and some resulted in having more. This can be seen in the count of elements in table 5.1 where we have many examples of a button but very few examples of a segmentedControl. One way we could

mitigate this in future might be to show users specific UI element types and ask them to draw them multiple times (in addition to copying low fidelity versions of screenshots).

While creating the data capture app we built in a number of tools including combining and splitting tools which functioned well for allowing the author to easily separate and group sketches/strokes into UI elements and then label them with a classification. We also created a debug tool which shows the model's current predictions (see Figure 6.1). It was a useful indicator for whether the current user's drawings were able to be detected by a working version of the model - i.e. if they can't be detected that may be a good indicator that their training data is valuable to include in the next round of training. However, it turned out to be useful in more ways than we intended. It was originally expected to be a convenient way to verify what the model would classify drawings as in the data capture tool (as we produced many different versions of models) but it could eventually become the building blocks of something that would help speed up the labelling process. In the drawing tool it became a useful way to work out what was happening "internally", almost like a way to debug the user's drawing.

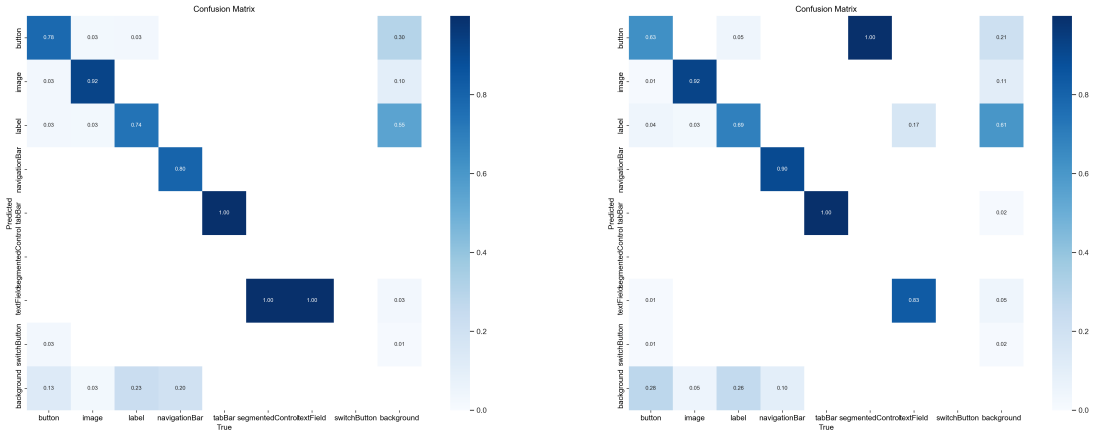
7.2 Training the model

We were able to train a model which was capable of recognizing sketches reasonably well with a relatively small dataset. While it may not recognize as many classes as well as we might have liked, it does recognise popular classes like label, button, image, navigation bar, tab bar well.

Similar to authors like Beltramelli (2017) and Liu, Hu and Shu (2018), the main issue for training stems from the amount of data we have. While we had a reasonable number of examples of common classes like "image", "label" or "button" others, like "segmented control", have very few examples. This is a common issue in our problem domain, Altinbas and Serif (2022) share a graph of the count of the different UI elements in their dataset and they find that the count of label, button, icon and image examples far outweigh the count of other elements.

Additionally, there is a chance those few examples may not be included in the training set - they could be in the test or validation set when we split all of our images prior to training. Obviously the majority end up in the training bucket but we cannot guarantee examples of infrequently occurring classes will end up there. For classes with fewer examples we are more likely to see the model confuse them with other elements or not be able to detect them at all. We can see an example of both of these scenarios in Figure 7.1, here we show the confusion matrices for experiments 34 and 35 in which we can see that the segmentedControl element in one experiment is confused with a textField but in the other is confused with a button. In the same figure we can also see that neither experiment is able to detect a switchButton. This lack of data drove us to only really be able to support more common UI elements, this is common amongst other solutions in our literature review (Chapter 2) with some authors only supporting binary classification (Nguyen and Csallner, 2015). Although this is not ideal it did not seem to be a huge issue while testing with users as they mostly wanted to draw the more common elements. They had more issues with detection where the model would sometimes detect the correct element and sometimes not. We feel more data would have helped correct this issue as well.

While we may not have had as much data as we might have anticipated, the data augmentation techniques we experimented with (Section 5.2.3) successfully improved the accuracy of the

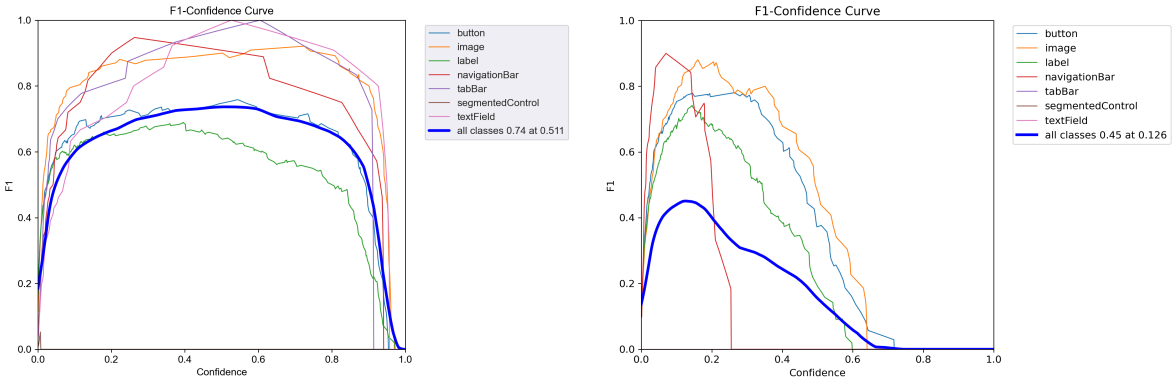


(a) Confusion matrix (experiment 34) (b) Confusion matrix (experiment 35)

Figure 7.1: Confusion matrices from experiments 34 and 35

model when compared to the default training with no additional augmentation.

We also observed that training took a longer period of time than was perhaps expected. In particular when training more layers (when fewer layers were frozen) we frequently encountered training times greater than six hours on an Apple MacBook Pro (Apple Silicon M2 Pro CPU with 32GB memory). With further investigation we discovered an issue with YOLOv5 which results in it only training on the CPU when on an Apple Silicon machine. We time-boxed an investigation into this issue and were unable to find a solution. Rather than spend further time on it we simply managed training times by running overnight or in the background when performing other tasks. As an experiment we were able to train a model using YOLOv8 on the GPU, this actually resulted in a similar training time to our maximum YOLOv5 training time (despite only running for around 180 epochs, where it auto-stopped due to lack of accuracy improvement) and produced a vastly less useful model - we can see in Figure 7.2 that the YOLOv8 model has a lower confidence and lower F1 score for all classes.



(a) F1 curve using yolov5 (experiment 35) (b) F1 curve using yolov8

Figure 7.2: F1 curves from different versions of YOLO

We did not spend any additional time tweaking hyperparameters for our YOLOv8 experiment but we were able to get reasonable results with YOLOv5 using the default setup, so while we might expect to be able to improve the results a little for YOLOv8 we doubt that changing hyperparameters would lead to vastly different results. Instead the differences between the

two YOLO approaches are likely to be the reason for the differing outcomes. YOLOv8 is very similar to YOLOv5 in that it is made up of a Feature Pyramid Network consisting of a backbone, neck and head, however the make up of these components is different (Selcuk and Serif, 2023), they contain different sizes of convolutional layers and different modules. In addition, and perhaps most significantly, YOLOv8 does not use anchor boxes, YOLOv5 starts with a predefined set of anchor boxes which it learns from the input data, instead YOLOv8 tries to predict them. Our theory is that this results in YOLOv8 requiring more data than we were able to provide for it to perform well. While we were not able to experiment with this theory Selcuk and Serif (2023) were able to achieve a high level of accuracy when training YOLOv8 with 4543 UI images (the same dataset as Selcuk and Serif (2023)) which is considerably more than from our 145 images.

While training time was perhaps not as fast as might have been expected we were able to manage it and this allowed us to experiment with variations on the model and changes to hyperparameters or different splits of the data amongst training/test/validation. This would have been more difficult if the training times were much more significant.

Another distraction we did not anticipate is that when training the model, we had expected the process to export a file which we could convert to CoreML format and immediately get working in our app. Unfortunately that was not the case and a fair amount of additional work was required to identify the issues and work up a solution, specifically around need for adding in non-maximum suppression (discussed in section 5.2.5).

Overall we feel the model we created was as accurate as it could be for the amount of data available and it performed well enough to show users the potential of the system (6.3.1).

7.3 Drawing app

As highlighted by many of our users, the drawing app would benefit from either a better UI design or some introduction/onboarding screens informing users where to draw and where to expect their preview output (see Section 6.3.3).

In terms of the design we suspect this is due to us initially presenting the user with two images of an iPhone frame which are indistinguishable. Looking at Figure 5.18, if the sketch area and preview area are empty they look the same.

While the design should be relatively simple to improve for clarity, we suspect some of the points made by users referring to capability and what components can be recognised, would be resolved by improving our model. The obvious route to an improved model would be to gather a greater quantity of data.

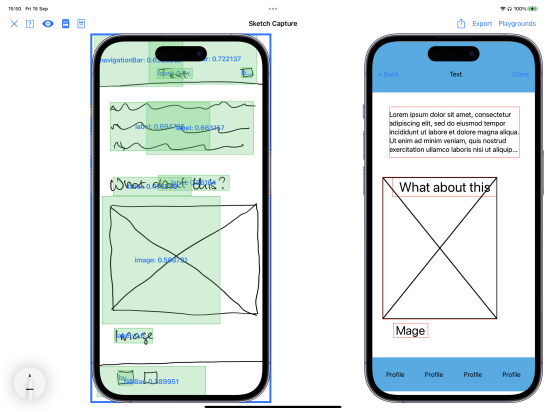
Despite this, all of the users seemed to find the drawing process intuitive and understood how it was meant to work from the outset (Section 6.3.4).

During the development of the drawing app we encountered a few issues that we attempted to address or manage. As we created subsequent versions of the model (either because of collecting more data or to experiment with hyper parameters) and added them to our app, we found that our thresholds for detection of elements changed. For some elements like **label**, button, image this was not an issue as we take a default threshold built in to the model (this value is 0.25). Conversely we apply additional post processing to elements like navigationBar and one of these post processing steps is to filter out any occurrences with confidences lower

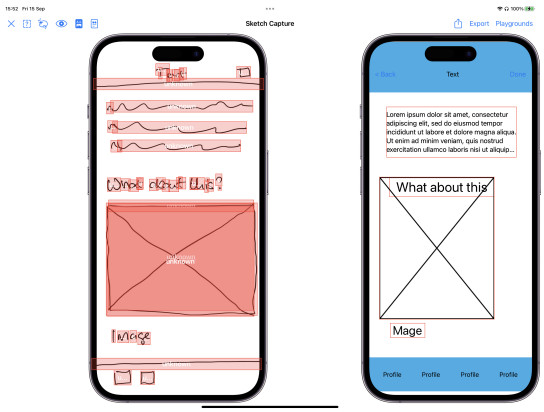
than a specific value (for some models we found 0.45 to be appropriate). This part of our process ensured that only a single navigationBar could occur, and was mostly successful, but we encountered issues as different model versions needed this value increased or decreased. This step could be improved by enhancing other post processing steps - we already only take results close to or at the top of the screen, but this threshold could be tightened - as well as training our model with more navigationBar samples. We could also address the YOLOv5 anchors discussed in this section.

Sometimes our output elements do not align to the original drawing's bounds or they overlap with other elements where they should not. This can be seen in Figure 7.3 where we can see the debug tool (in green, Figure 7.3a) is different to the drawing bounds (in red, Figure 7.3b) and is also showing different bounds to the output for both the image and the tab bar.

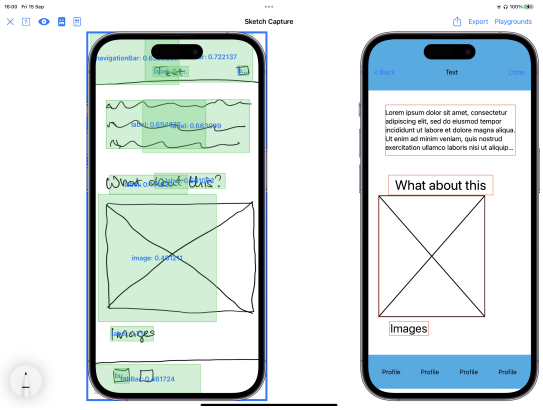
Furthermore, we can also see that the output image height is incorrect in Figures 7.3 and 7.3b. However, a small change to a nearby element (the bottom "Images" label - we added an "s") causes the model to re-run and detect the height of the image more successfully (Figure 7.3c). While this is not an issue for the tab bar (as our post processing will force it to be full width as this is the only way a tab bar can occur), we cannot apply this logic to an image which can be any size.



(a) Debug tool bounds



(b) Drawing bounds



(c) Output image height changed

Figure 7.3: A drawing showing the difference between model output, debug mode and actual drawing bounds

There are a number of causes of this issue. The first is that our post processing could be improved further. Because we are able to receive the drawing, perform the model inference and

draw the output all on a single device we could use the knowledge of the drawing's bounding box information in our post processing to better align the output with the intended size.

The other cause of the mis-positioning issue is due to how YOLOv5 locates objects. It uses anchor boxes (as discussed in Section 5.2.5). These anchor boxes are a fixed size and created as part of the training process. As with the rest of the model training they are dependent on the variety and quantity of training data. It may be that our data did not produce anchor boxes that are a good fit for real world data, or simply that anchor boxes are not a good match for our type of data - UI elements like image's can be any size after all, so attempting to bound them to fixed anchor boxes may be inappropriate.

Overall the resulting app was able to recognise sketches reasonably well. While some of the participants in our user interviews had issues with the app correctly detecting and generating what they expected (Section 6.3.1), they were happy to erase or undo their drawings and try again to get the results they wanted (Section 6.3.2). Furthermore, most users found the drawing intuitive (Section 6.3.4) and were impressed with its abilities and possibilities.

7.4 Code generation and exporting

The app successfully generates SwiftUI code which can be exported and compiled by Apple's compiler. We have not seen any instances where the user has managed to draw something which generated code that could not be compiled in the latest version of the drawing app.

The code generated uses absolute positioning for UI elements, this has the disadvantage that the UI it represents will look correct on the screen size on which is was drawn but on other screens of smaller or larger size it will not scale up or down. While this is fine for prototyping, when building the "real" product the code generated would need to use a layout system that allows for varying screen sizes. However this did not really seem to be an issue for users (Section 6.3.5).

Being able to export a file and open it in Apple's Playgrounds app on the same device and subsequently show that the code compiles is very convenient. Users generally seemed happy with this transition from generating code via drawing to then be able to run and further edit the code (see Section 6.3.5).

While the ideal solution would have been to compile the generated code in app, there is no Swift compiler available to run on iPad OS (in third party apps) and so compromises had to be made. However, our solution is close enough to this ideal and didn't seem to be an issue for any of our users (see Chapter 6).

7.5 Compatibility with users' workflows

The user survey revealed several interesting points around the workflow in use by our participants. Many were keen to be able to "theme" the output of the app (Section 6.3.6), with the goal of being able to present prototypes to clients in their brand colours. This is certainly an interesting idea and it could be argued this starts to push the output of the app from low fidelity towards more higher fidelity designs. It also may indicate that the survey participants are thinking about how this could fit into and enhance their workflow, rather than dismissing it.

The other interesting area for discussion relates to the tools used by the participants (Section 6.3.7). Many named tools such as Figma or Adobe XD, which are generally used for high fidelity design or high fidelity prototypes. We would speculate that the participants are generally working with pre-existing apps where the design language has already been established and so paper prototypes perhaps make less sense at this stage. This may be an interesting area to explore in future, specifically as to whether we could incorporate high fidelity UI component designs (i.e. a branded buttons, which may be a step beyond "theming") to start outputting code which more closely resembles high fidelity prototypes and offers greater commercial value for the tool.

This may suggest that the tool we built doesn't currently fit the user's existing workflow. There may be several reasons for this, as Participant F noted there is nothing similar available currently so it may require a change of workflow, or perhaps it does not quite fit in its current state as it is not a production level application. It may also be that the users surveyed are not currently involved in prototyping for a new app, instead they may be evolving existing apps with an established design (for which Adobe XD etc would make it fast to drag and drop components together for new screen designs - though there's currently no mobile app code exported from this). Or it could simply be that our survey size was too small, involving too few users to get a conclusive overview of how this tool might fit into various design workflows.

7.6 Conclusion

In this chapter we reflected on the components we built and the feedback we received from the user survey. The app successfully allows users to sketch out a UI and automatically generate working SwiftUI code, albeit sometimes with some detection or classification issues. We believe many of these issues could be lessened or resolved by improving our model. As the project progressed and we captured more data we created further iterations of the model and steadily improved detection. If we continue this process over a longer period of time we believe we could achieve better detection and classification accuracy in the future.

Our participants also noted that the UI design of our drawing app was not as clear as they desired in terms of where they should start, what they should do or what the capabilities are. We should look to update this in future versions to make it more intuitive and add an onboarding screen to introduce users to the app's features.

A key area of interest is how the app fits into a user's existing workflow. While the sample size of our survey was small we hoped there may be designers who would find the app a better fit for their workflow. It would be interesting to explore this further for future versions. Overall, we were encouraged to find that most participants were impressed with the app and could see potential value in using the tool.

Chapter 8

Conclusion

The goal of this project (1.5) was to create an application that could generate code from hand-drawn sketches of mobile UI designs in an effort to improve the prototyping phase of mobile app software development. In our literature and technology survey (Chapter 2) we found that iOS was an under-explored area in the research, and more specifically that SwiftUI was completely unexplored as an output format. We also found that the reason UI prototyping is often performed with pen and paper is the ease of use compared with other approaches. With that in mind we decided to create an iPad application that used the Apple Pencil as input (Chapters 4 and 5), attempting to replicate the ease of using pen and paper whilst also providing cutting edge features like SwiftUI code generation. We then undertook a user survey to gather feedback on our approach (Chapter 6).

In Section 1.5 we proposed a research question, asking how the use of an iPad app for sketching mobile designs and automatically generating code might impact the efficiency of rapid prototyping in UI/UX design, and in what ways does this method offer advancements over conventional prototyping techniques? We believe it can aid the efficiency of rapid prototyping, and whilst our user survey (Chapter 6) did not show our tool slotting easily into participants' existing workflows it did spark enthusiasm amongst the group and gave non-developers the confidence to edit code. If we were able to see similar results with a larger sample group that would be a significant positive result and show huge potential for increasing the efficiency of rapid prototyping as designers and other non-developers would perhaps be able to create working, native prototypes of new app designs without utilising expensive developer expertise. Furthermore, our iPad app offers up several advantages over conventional prototyping tools (such as pencil and paper) which include being fully digital, generating code from sketches automatically and the ability to share this code.

8.1 Contributions

While the project is inspired by existing works it makes a number of developments and contributions to the field.

Firstly, most existing research or software projects of a similar nature (see Section 2.3) rely on the use of a computer such as Baulé et al. (2021) where images of sketches are captured with a mobile device and then use a web tool to perform the rest of the functions. Most take the approach of processing a photo of a user sketch and some allow the user to sketch using

the mouse or other input device (Mohian and Csallner, 2020). Our project improves on these approaches by using an iPad and Pencil which are smaller than a full size computer and nearly as portable as a notebook and pen. Furthermore, our app can run entirely offline allowing users many of the same freedoms that a pen and paper approach might provide. Whilst it is not clear whether similar projects required an internet connection or not, we would probably assume that most that rely on a web tool require a network connection. It is hard to argue with the portability that our app provides.

We believe that our approach is the first such project to generate SwiftUI code, and while other research has looked to generate iOS code it is certainly less prominent than Android and is often cross platform code, such as AppInventor used by Baulé et al. (2021), rather than what might be considered "native" code (Swift or Objective C).

Whilst our training dataset is gathered on the device we are targeting, other research has generally focused on either gathering (or reusing) pen and paper sketches (Baulé et al., 2021) or generating sketch-like images from other images (Robinson, 2019) or other synthetic data sets like Syn (Pandian, Suleri and Jarke, 2020). This removes some element of doubt as to whether there are issues with our data that affect performance from the perspective of "are our sketches realistic enough" or other faults we might be forced to question if we had used generated or artificial sketches (Calò and Russis, 2022). We might also have questioned the similarity of line data (thickness/texture/etc as discussed in Chapter 4) if we had attempted to train using paper-based sketches.

Our model provides near real-time feedback (Section 5.3.4) to users as they sketch and our app allows the user to draw naturally as they would with pen and paper with the model continuing to respond as they draw further. In our technology survey we were only able to find limited examples of other research where a response from the system is provided as the user draws (Mohian and Csallner, 2020) and even then it required the user to indicate they had drawn an element (instead of the system figuring it out), which - unlike our app - offers quite a jarring requirement for the user compared with pen and paper. The only other example we were able to find was from Wimmer, Untertrifaller and Grechenig (2020) who use a computer and webcam mounted to a whiteboard to capture sketches of web page layouts and generate them as the user draws. While they achieve a similar goal of allowing users to draw naturally and receive feedback with previews in real-time, their setup is more cumbersome and less portable than our approach. They do not mention the time taken for their system to generate an output, making it difficult to compare whether their claim of "real-time" is any faster or slower than ours.

As far as we have found, no other approach allows the entire user experience to occur on a single device without an internet connection (see Section 8.2 for our caveat), matching the versatility of pen and paper prototyping. Mohian and Csallner (2020) were the closest to this, creating a website that allowed users to draw with a mouse and marking each drawn element as complete for processing and repeating until the full drawing was complete. However, it relies on running in a web browser with an internet connection. Our app allows the user to draw, see a live preview, generate code and compile it or tweak it (in Swift Playgrounds) all on a single device and all offline. While pen and paper has the advantages of ease of use and portability, it lacks polish and cannot generate code.

Other approaches such as Robinson (2019) may require that the user digitises sketches, uploading them to online services and, if they target Apple platforms, may further require

code to be exported to other devices (Swift and SwiftUI code can only be compiled on Apple systems). We believe ours is the first approach to use a mobile device to perform the whole process. While other approaches could potentially deploy their solution on a laptop (which could perhaps be considered a mobile device of sorts) the majority of them would still be gathering sketch data via camera input or transferring from a phone to then move on to the computer. This is important as it is another example of where we are trying to reduce the differences between paper based prototyping and digital prototyping systems, if we can match the portability and speed of paper then we are on the right track. Wimmer, Untertrifaller and Grechenig (2020) have similar ambitions to build a system that allows authors to avoid the manual digitisation of sketches. Their system successfully captures user sketches in real-time from a whiteboard, however they acknowledge in their limitations that a smartphone app or a graphics tablet approach would potentially provide a better setup. Theoretically, there would be nothing to stop us also allowing touch input (rather than only Apple Pencil) for drawing and deploying the very same app on an iPhone, it would still work and likely with comparable performance to the iPad Pro we have tested on.

Our data capture process is also unique. The majority of approaches, as mentioned in Section 2.5, use drawings gathered from volunteers on paper. A few, such as Mohian and Csallner (2020), used sketch data captured digitally. We take this a step further with our system. We created a data capture iPad app which allows the user to draw sketches (see Section 5.1). The user is provided with a random screenshot to make a low fidelity copy of (an evolution of Baulé et al. (2021)). We provide various tools for the user to then label the elements of the drawing (including splitting and grouping) and when complete, the drawing and its metadata is saved to our Firebase server ready for use in training.

8.2 Comparison to our original requirements

In this section we compare the app we built to the requirements we originally created in Chapter 3. The app successfully met all the requirements.

Requirements **RQ1** and **RQ3** specified that the user must be able to draw on the device and that the user should be able to draw with a stylus. Using the Apple Pencil and iPad achieves these requirements as it allows users to draw using the Pencil (stylus) directly on to the iPad.

As discussed in section 5.3.4, our model performs in near real-time, and provides the user with prompt feedback. This certainly achieves requirement **RQ4**.

The app captures the user drawing and passes it to our trained model every time the Pencil leaves the screen. Most user interface elements take at least one stroke to draw so this results in the model running at least once for each element the user is drawing. This results in the app generating code automatically as needed, fulfilling requirement **RQ5**.

Requirement **RQ2** specifies we must generate SwiftUI code. We discuss our generated code in Section 5.3.3 and 7.4, it successfully generates SwiftUI code that can be independently compiled in Apple's apps and tools.

All of the above fulfilled requirements run on the iPad device without the user having to context switch or move to a computer. To compile their generated SwiftUI code they can use the Apple's iPad app "Swift Playgrounds" (Apple, 2023g) and we even added a shortcut button in our app to open the Playgrounds app for them. We consider this to have fulfilled

requirement **RQ6**.

We ended up building both the data capture app and drawing app within one app and allow the user to choose where to go from the introduction screen. Prior to the introduction screen there is an initial loading screen that requires an internet connection as it fetches a screenshot for use in the data capture app. However, the drawing app portion has no need for an internet connection. Once a user is past the initial screen the WiFi can be disabled and the drawing app will still function. It is not a significant piece of work to separate the drawing app properly from the data capture app, however as this was not one of the main goals of the project this remains a piece of future work. In keeping with these goals we specified requirement **RQ8** as a "could" in the MoSCoW analysis and we feel we have accomplished this. The app could indeed work "offline" with a little additional development time (and can currently be used offline with a little planning - loading to the initial screen with a connection).

Our final requirement, **RQ9**, referred to not considering complex UI designs, navigation or animation as part of our project but we have built a solid foundation upon which we could begin to build out these more complex areas in future.

8.3 Future work

Throughout this project we have encountered areas for potential future research or development.

8.3.1 Improvements to the data capture tool

Assuming a future version of the app would likely need to gather more data we would intend to update the data capture tool in an effort to speed up the labelling steps discussed in Section 7.1. One option for speeding up this stage would be to incorporate the latest version of the model and use it to do a first pass at grouping and labelling the sketches. The user can then confirm or correct those labels and upload the results and this would significantly speed up the data capture process.

8.3.2 Model training

There are various opportunities for us to gather more data from users and subsequently update our model to improve classification and detection and in future introduce more UI element types.

The first step would be to introduce a way for the user to mark wrongly detected (or undetected) elements with their correct classifications (and potentially locations). This extra input data could be immediately reflected in the preview and code output panels but can also be used as input data along with a capture of their sketch.

One way to use this data may be to explore leveraging Apple's updatable CoreML model feature (Apple, 2023i) to personalise the on device model with the user's individual sketching style. While this would not help improve the general model for other users it would keep the user's sketch data private to their own device.

Another route would be to upload these corrections (with permission from the user) to our own server in the same format our data capture app uses. These extra examples can then be used to expand our data set and produce improved models for future updates of the app.

In our implementation we were able to detect and classify a number of popular UI elements, however there are more elements that we did not attempt to detect (mostly due to lack of data). We also have elements which under perform or are infrequently detected compared to more popular elements like image or label. Further data capture could be undertaken to gather more examples of the less common UI elements in an effort to train a more accurate and reliable model.

There are also opportunities for us to experiment with more models or algorithms. Expanding our dataset would allow us to test our theory that YOLOv8 requires more data to train an equivalent model and determine whether or not it's anchor free approach is superior to YOLOv5 for our purposes (Section 7.2). Though it is worth noting that Selcuk and Serif (2023) compared YOLOv8 to YOLOv5 and found only marginal improvements in accuracy. Another route to attempting to improve our anchor box accuracy may be training with YOLOv5-MGC (Cheng et al., 2022). This uses K-means++ to generate anchor regions instead of the K-means algorithm used by Ultralytics YOLOv5 in an effort to make them more suitable to a UI element dataset, they also tweak the models backbone to try and improve detection and classification accuracy.

A completely alternate approach may be to capture and use the vector data we have from the users' drawings for training a new model in a similar manner to Mohian and Csallner (2020). It may be that the additional data such as drawing order etc can produce a more accurate model than training with simply image data alone.

8.3.3 Detection and classification

From the user's drawing we have location information about each of their individual UI elements and we then pass the whole image to the trained model which generates its own set of location information for the detected elements. As we discussed in Section 7.3, the output position or size is sometimes wrong, we could look to correct this bounding box data when we think we have better information in the original sketch.

Participants in our user survey (Appendix A.1) commented that they would normally design to a grid, with set margins and other such fixed sizing. They suggested it would be beneficial for the app to output with this in mind, aligning items where appropriate and matching the ideal spacing of the designers grid.

It was highlighted in Section 2.9 that many apps rely on a list view of sorts, however detecting list or table views was something we did not have time to address. This would be another interesting area to explore as we noted in Chapter 2 that very few other approaches had attempted to tackle this layout either. Taking this a step further we could also attempt to detect and output other types of screen layout in an effort to move away from absolute positioning in our output code (see Section 7.4).

8.3.4 Other platforms and languages

One of the goals of this project was to be able to accomplish every task the user may need, including compilation, on a single device. However, it may be useful to the user to also offer to output languages which cannot be compiled on the device as well. For instance, we could output Android code in the form of either XML layouts (Google, 2023e) or Jetpack Compose (Google, 2023c) code, the latter being very similar to SwiftUI. While we could not compile the

code on device it may be useful for users to be able to export for additional platforms without additional coding knowledge.

Similarly, we could also look to integrate cross-platform languages. A more complex and integrated solution could be created with React Native. We could potentially embed a React Native view in the app instead of our current preview panel. This could be populated with our own generated JavaScript and be able to run and update live code from our app without the user having to leave to another app to compile (as happens currently). While we had specific reasons for choosing SwiftUI (see Chapters 2 and 3) this may be another interesting avenue to pursue.

Investigating adapting the UI of our drawing app for deployment on an iPhone would be a reasonably simple exercise in terms of design and development. The interesting area to explore further would be how easily a user can draw a UI on a device which is the same size as the expected output. Would our model still detect sketches drawn using a finger instead of an Apple Pencil? Can the user draw in a detailed enough manner in this way? There may be potential solutions such as allowing the user to zoom in and out to make smaller drawings but these likely also introduce new challengers. However, having the ability to create prototypes in this way on a phone would be a compelling offering.

8.3.5 Extending code generation

There are also several opportunities we could explore to extend the functionality of code generated beyond adding more UI elements. It would not be a huge leap to allow multiple screen sketches and have users draw a line from a button to another screen to indicate navigation. This could then be output in the code.

Another major part of many mobile apps is scrolling. There are several options for supporting this. One of the easier routes would be to allow the user to extend the sketchable area downwards to create a longer screen. Then when code is generated if the element's positions run off the bottom of the screen it is an indication that the screen should scroll and this can be accounted for.

With more time it would be interesting to experiment with machine learning based approaches to code generation, such as from Beltramelli (2017) and Liu, Hu and Shu (2018), as opposed to our programmatic approach.

8.3.6 High fidelity design output

While our project focused on generating low fidelity designs several participants in our survey (Appendix A.1) suggested high fidelity output would be nice to have too. This would be an interesting area to explore, specifically whether generative AI techniques such as Generative Adversarial Networks (GAN) could be employed to produce a high fidelity design based off a low fidelity sketch, as has been suggested by Beltramelli (2017).

8.3.7 General improvements

Many of the following improvements were considered out of scope for this project but would be valuable additions for a production version of this application:

- Adding support for accessibility to the generated code.

- Adding support for coloured elements, for instance if a sketch is drawn in blue then the outputted element should be blue.
- Adding support for theming (providing a number of brand colours), once a theme is set all elements would be generated in the theme colours.
- Support for different sub-types of UI elements, for example an underlined button, a label button rather than only the standard button we currently support.
- Update the design of the drawing app to make its usage clearer to new users.

Bibliography

Abdelhamid, A.A., Alotaibi, S.R. and Mousa, A., 2020. Deep learning-based prototyping of android GUI from hand-drawn mockups. *Iet software* [Online], 14(7), pp.816–824. Available from: https://doi.org/10.1049/iet-sen.2019.0378.

Adefris, B.B., Habtie, A.B. and Taye, Y.G., 2022. Automatic Code Generation From Low Fidelity Graphical User Interface Sketches Using Deep Learning. *2022 international conference on information and communication technology for development for africa (ict4da)* [Online], 00, pp.1–6. Available from: https://doi.org/10.1109/ict4da56482.2022.9971204.

Altinbas, M.D. and Serif, T., 2022. GUI Element Detection from Mobile UI Images Using YOLOv5. *Lecture notes in computer science* [Online], pp.32–45. Available from: https://doi.org/10.1007/978-3-031-14391-5_3.

Amazon, 2018. Amazon Mechanical Turk [Online]. Available from: https://www.mturk.com/ [Accessed 2023-12-6].

Apple, 2017. iPad Pro, in 10.5-inch and 12.9-inch models, introduces the world's most advanced display and breakthrough performance [Online]. Available from: https://www.apple.com/uk/newsroom/2017/06/ipad-pro-10-5-and-12-9-inch-models-introduces-worlds-most-advanced-display-breakthrough-performance/ [Accessed 2023-11-30].

Apple, 2023a. Apple Design Awards [Online]. Available from: https://developer.apple.com/design/awards/ [Accessed 2023-4-16].

Apple, 2023b. Apple Human Interface Guidelines - Navigation bars [Online]. Available from: https://developer.apple.com/design/human-interface-guidelines/navigation-bars [Accessed 2023-10-26].

Apple, 2023c. Choosing a Resource Storage Mode for Apple GPUs [Online]. Available from: https://developer.apple.com/documentation/metal/resource_fundamentals/choosing_a_resource_storage_mode_for_apple_gpus [Accessed 2023-10-27].

Apple, 2023d. Discover visionOS [Online]. Available from: https://developer.apple.com/visionos/ [Accessed 2023-12-6].

Apple, 2023e. Human Interface Guidelines [Online]. Available from: https://developer.apple.com/design/human-interface-guidelines/guidelines/overview/ [Accessed 2023-4-16].

Apple, 2023f. iCloud Drive [Online]. Available from: https://www.icloud.com/iclouddrive [Accessed 2023-11-3].

Apple, 2023g. Learn to code with Swift Playgrounds [Online]. Available from: https://developer.apple.com/swift-playgrounds/ [Accessed 2023-11-3].

Apple, 2023h. load(contentsOf:configuration:completionHandler:). Available from: https://developer.apple.com/documentation/coreml/mlmodel/3600218-load [Accessed 2023-12-12].

Apple, 2023i. Personalizing a Model with On-Device Updates [Online]. Available from: https://developer.apple.com/documentation/coreml/model_personalization/personalizing_a_model_with_on-device_updates [Accessed 2023-12-7].

Apple, 2023j. Recognizing Text in Images [Online]. Available from: https://developer.apple.com/documentation/vision/recognizing_text_in_images [Accessed 2023-10-27].

Apple, 2023k. SwiftUI [Online]. Available from: https://developer.apple.com/xcode/swiftui/ [Accessed 2023-11-2].

Apple, 2023l. Xcode 15 [Online]. Available from: https://developer.apple.com/xcode/ [Accessed 2023-11-3].

Aşıroğlu, B., Mete, B.R., Yıldız, E., Nalçakan, Y., Sezen, A., Dağtekin, M. and Ensari, T., 2019. Automatic HTML Code Generation from Mock-Up Images Using Machine Learning Techniques. *2019 scientific meeting on electrical-electronics & biomedical engineering and computer science (ebbt)* [Online], 00, pp.1–4. Available from: https://doi.org/10.1109/ebbt.2019.8741736.

Babich, N., 2023. Low fidelity vs. high fidelity: the differences between design prototypes. Available from: https://webflow.com/blog/low-vs-high-fidelity#::text=Fidelity%20can%20vary%20in%20content,possible%20to%20the%20final%20design. [Accessed 2023-11-24].

Bajammal, M., Mazinanian, D. and Mesbah, A., 2018. Generating reusable web components from mockups. *Proceedings of the 33rd acm/ieee international conference on automated software engineering* [Online], 00, pp.601–611. Available from: https://doi.org/10.1145/3238147.3238194.

Baulé, D., Wangenheim, C.G.v., Wangenheim, A. and Hauck, J.C.R., 2020. Recent Progress in Automated Code Generation from GUI Images Using Machine Learning Techniques. *Jucs - journal of universal computer science* [Online], 26(9), pp.1095–1127. Available from: https://doi.org/10.3897/jucs.2020.058.

Baulé, D., Wangenheim, C.G.v., Wangenheim, A.v., Hauck, J.C.R. and Júnior, E.C.V., 2021. Automatic code generation from sketches of mobile applications in end-user development using Deep Learning. *arxiv* [Online]. 2103.05704, Available from: https://doi.org/10.48550/arxiv.2103.05704.

Beltramelli, T., 2017. pix2code: Generating Code from a Graphical User Interface Screenshot. *arxiv* [Online]. 1705.07962, Available from: https://doi.org/10.48550/arxiv.1705.07962.

Biswas, S., Wardat, M. and Rajan, H., 2022. The art and practice of data science pipelines. *Proceedings of the 44th international conference on software engineering* [Online], pp.2091–2103. 2112.01590, Available from: https://doi.org/10.1145/3510003.3510057.

Bochkovskiy, A., 2023. Yolo v4, v3 and v2 for Windows and Linux [Online]. Available from: `https://github.com/AlexeyAB/darknet` [Accessed 2023-10-12].

Bove, T., 2023a. PyTorch Conversion Workflow [Online]. Available from: `https://apple.github.io/coremltools/docs-guides/source/convert-pytorch-workflow.html` [Accessed 2023-10-22].

Bove, T., 2023b. What Is Core ML Tools? [Online]. Available from: `https://apple.github.io/coremltools/docs-guides/source/overview-coremltools.html` [Accessed 2023-10-12].

Brie, P., 2019. The Second Version of Our Vision API [Online]. Available from: `https://teleporthq.io/blog/new-vision-api` [Accessed 2023-9-22].

Calò, T. and Russis, L.D., 2022. Style-Aware Sketch-to-Code Conversion for the Web. *Companion of the 2022 acm sigchi symposium on engineering interactive computing systems* [Online], pp.44–47. Available from: `https://doi.org/10.1145/3531706.3536462`.

Carter, A.S. and Hundhausen, C.D., 2010. How is User Interface Prototyping Really Done in Practice? A Survey of User Interface Designers. *2010 ieee symposium on visual languages and human-centric computing* [Online], pp.207–211. Available from: `https://doi.org/10.1109/vlhcc.2010.36`.

Cheng, J., Tan, D., Zhang, T., Wei, A. and Chen, J., 2022. YOLOv5-MGC: GUI Element Identification for Mobile Applications Based on Improved YOLOv5. *Mobile information systems* [Online], 2022, pp.1–9. Available from: `https://doi.org/10.1155/2022/8900734`.

Chin, M., 2021. Apple says you can build apps on an iPad now, but devs say the reality is trickier. Available from: `https://www.theverge.com/2021/6/15/22534902/ipad-pro-apple-swift-playgrounds-4-wwdc-2021` [Accessed 2023-11-3].

Dalmasso, I., Datta, S.K., Bonnet, C. and Nikaein, N., 2013. Survey, Comparison and Evaluation of Cross Platform Mobile Application Development Tools. *2013 9th international wireless communications and mobile computing conference (iwcmc)* [Online], 1, pp.323–328. Available from: `https://doi.org/10.1109/iwcmc.2013.6583580`.

Dawson, C.W., 2015. *Projects in Computing and Information Systems*. 3rd ed. Pearson Education Limited.

Deka, B., Huang, Z., Franzen, C., Hibschman, J., Afergan, D., Li, Y., Nichols, J. and Kumar, R., 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. *Proceedings of the 30th annual acm symposium on user interface software and technology* [Online], pp.845–854. Available from: `https://doi.org/10.1145/3126594.3126651`.

Delía, L., Galdamez, N., Corbalan, L., Pesado, P. and Thomas, P., 2017. Approaches to mobile application development: Comparative performance analysis [Online]. *2017 computing conference*. pp.652–659. Available from: `https://doi.org/10.1109/SAI.2017.8252165`.

Dogtiev, A., 2023. App Development Cost. Available from: `https://www.businessofapps.com/app-developers/research/app-development-cost/` [Accessed 2023-11-24].

Flarup, M., 2016. What You Should Know About The App Design Process [Online]. Avail-

able from: https://www.smashingmagazine.com/2016/11/what-everyone-should-know-about-the-process-behind-app-design/ [Accessed 2023-4-16].

Flora, H.K., Wang, X. and Chande, S.V., 2014. An Investigation into Mobile Application Development Processes: Challenges and Best Practices. *International journal of modern education and computer science* [Online], 6(6), pp.1–9. Available from: https://doi.org/10.5815/ijmecs.2014.06.01.

Google, 2023a. Android Studio [Online]. Available from: https://developer.android.com/studio [Accessed 2023-12-5].

Google, 2023b. Cloud Storage for Firebase [Online]. Available from: https://firebase.google.com/docs/storage [Accessed 2023-10-12].

Google, 2023c. Compose layout basics [Online]. Available from: https://developer.android.com/jetpack/compose/layouts/basics [Accessed 2023-12-7].

Google, 2023d. Firebase [Online]. Available from: https://firebase.google.com/ [Accessed 2023-10-12].

Google, 2023e. Layouts in Views [Online]. Available from: https://developer.android.com/develop/ui/views/layout/declaring-layout [Accessed 2023-12-7].

Ha, D. and Eck, D., 2017. A Neural Representation of Sketch Drawings. *arxiv* [Online]. 1704.03477, Available from: https://doi.org/10.48550/arxiv.1704.03477.

Jain, V., Agrawal, P., Banga, S., Kapoor, R. and Gulyani, S., 2019. Sketch2Code: Transformation of Sketches to UI in Real-time Using Deep Neural Network. *arxiv* [Online]. 1910.08930, Available from: https://doi.org/10.48550/arxiv.1910.08930.

Jiang, P., Ergu, D., Liu, F., Cai, Y. and Ma, B., 2022. A Review of Yolo Algorithm Developments. *Procedia computer science* [Online], 199, pp.1066–1073. Available from: https://doi.org/10.1016/j.procs.2022.01.135.

Jocher, G., 2023. YOLOv5 Pretrained Checkpoints [Online]. Available from: https://github.com/ultralytics/yolov5#pretrained-checkpoints [Accessed 2023-10-12].

King, M., 2023. App Development Process. Available from: https://www.businessofapps.com/app-developers/research/app-development-process/ [Accessed 2023-11-24].

Leivers, D., 2023a. Experiments with Swift Playgrounds and FileDocument (or, how to save your project to an Apple FileDocument type) [Online]. Available from: https://medium.com/@sofaracing/experiments-with-swift-playgrounds-and-filedocument-or-how-to-save-your-project-to-an-apple-5347a2f4c94 [Accessed 2023-11-3].

Leivers, D., 2023b. Near real-time UI code generation on iPad [Online]. Available from: https://www.youtube.com/watch?v=SKGdZ3H9eyY [Accessed 2023-11-16].

Lepore, T., 2010. Sketches and Wireframes and Prototypes! Oh My! Creating Your Own Magical Wizard Experience [Online]. Available from: https://www.uxmatters.com/mt/archives/2010/05/sketches-and-wireframes-and-prototypes-oh-my-creating-your-own-magical-wizard-experience.php [Accessed 2023-4-10].

Lin, T.Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C.L. and Dollár, P., 2014. Microsoft COCO: Common Objects in Context. *arxiv* [Online]. 1405.0312, Available from: https://doi.org/10.48550/arxiv.1405.0312.

Liu, Y., Hu, Q. and Shu, K., 2018. Improving pix2code based Bi-directional LSTM. *2018 ieee international conference on automation, electronics and electrical engineering (auteee)* [Online], 00, pp.220–223. Available from: https://doi.org/10.1109/auteee.2018.8720784.

Meta, 2023a. Communication between native and React Native [Online]. Available from: https://reactnative.dev/docs/communication-ios [Accessed 2023-12-5].

Meta, 2023b. JavaScript Environment [Online]. Available from: https://reactnative.dev/docs/javascript-environment [Accessed 2023-12-5].

Microsoft, 2019. Sketch2Code (Documentation) [Online]. Available from: https://github.com/microsoft/ailab/tree/master/Sketch2Code [Accessed 2023-4-16].

MIT, 2022. MIT App Inventor [Online]. Available from: https://appinventor.mit.edu [Accessed 2023-3-28].

Mohian, S. and Csallner, C., 2020. Doodle2App. *Proceedings of the ieee/acm 7th international conference on mobile software engineering and systems* [Online], pp.81–84. Available from: https://doi.org/10.1145/3387905.3388607.

Nawrocki, P., Wrona, K., Marczak, M. and Sniezynski, B., 2021. A Comparison of Native and Cross-Platform Frameworks for Mobile Applications. *Computer* [Online], 54(3), pp.18–27. Available from: https://doi.org/10.1109/mc.2020.2983893.

Neubeck, A. and Gool, L.V., 2006. Efficient Non-Maximum Suppression. *18th international conference on pattern recognition (icpr'06)* [Online], 3, pp.850–855. Available from: https://doi.org/10.1109/icpr.2006.479.

Nguyen, T.A. and Csallner, C., 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). *2015 30th ieee/acm international conference on automated software engineering (ase)* [Online], pp.248–259. Available from: https://doi.org/10.1109/ase.2015.32.

Nikam, C., Keshervani, R., Shah, S. and Aghav, J., 2021. Code Generation from Images Using Neural Networks. *Algorithms for intelligent systems* [Online], pp.149–160. Available from: https://doi.org/10.1007/978-981-16-3802-2_12.

Orhon, A., Wadhwa, A., Kim, Y., Rossi, F. and Jagadeesh, V., 2022. Deploying Transformers on the Apple Neural Engine [Online]. Available from: https://machinelearning.apple.com/research/neural-engine-transformers [Accessed 2023-10-12].

Pandian, V.P.S., Suleri, S. and Jarke, M., 2020. Syn: Synthetic Dataset for Training UI Element Detector From Lo-Fi Sketches. *Proceedings of the 25th international conference on intelligent user interfaces companion* [Online], pp.79–80. Available from: https://doi.org/10.1145/3379336.3381498.

Pandian, V.P.S., Suleri, S. and Jarke, M., 2021a. SynZ: Enhanced Synthetic Dataset for Training UI Element Detectors. *26th international conference on intelligent user interfaces* [Online], pp.67–69. Available from: https://doi.org/10.1145/3397482.3450725.

Pandian, V.P.S., Suleri, S. and Jarke, P.D.M., 2021b. UISketch: A Large-Scale Dataset of UI Element Sketches. *Proceedings of the 2021 chi conference on human factors in computing systems* [Online], pp.1–14. Available from: https://doi.org/10.1145/3411764.3445784.

Prakash, J., 2021. Non Maximum Suppression: Theory and Implementation in PyTorch [Online]. Available from: https://learnopencv.com/non-maximum-suppression-theory-and-implementation-in-pytorch/ [Accessed 2023-10-22].

Redmon, J., Divvala, S., Girshick, R. and Farhadi, A., 2016. You Only Look Once: Unified, Real-Time Object Detection. *2016 ieee conference on computer vision and pattern recognition (cvpr)* [Online], pp.779–788. Available from: https://doi.org/10.1109/cvpr.2016.91.

Redmon, J. and Farhadi, A., 2017. YOLO9000: Better, Faster, Stronger. *2017 ieee conference on computer vision and pattern recognition (cvpr)* [Online], pp.6517–6525. Available from: https://doi.org/10.1109/cvpr.2017.690.

Ren, S., He, K., Girshick, R. and Sun, J., 2016. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *Ieee transactions on pattern analysis and machine intelligence* [Online], 39(6), pp.1137–1149. Available from: https://doi.org/10.1109/tpami.2016.2577031.

Robinson, A., 2019. Sketch2code: Generating a website from a paper mockup. *arxiv* [Online]. 1905.13750, Available from: https://doi.org/10.48550/arxiv.1905.13750.

Selcuk, B. and Serif, T., 2023. A Comparison of YOLOv5 and YOLOv8 in the Context of Mobile UI Detection. *Lecture notes in computer science* [Online], pp.161–174. Available from: https://doi.org/10.1007/978-3-031-39764-6_11.

Solawetz, J., 2020. What is YOLOv5? A Guide for Beginners. [Online]. Available from: https://blog.roboflow.com/yolov5-improvements-and-evaluation/#origin-of-yolov5-an-extension-of-yolov3-pytorch [Accessed 2023-10-12].

StackOverflow, 2023. 2023 Developer Survey [Online]. Available from: https://survey.stackoverflow.co/2023/#section-top-paying-technologies-top-paying-technologies [Accessed 2023-11-24].

UXPin, 2021. High-Fidelity Prototyping vs Low-Fidelity Prototypes: Which to Choose When? [Online]. Available from: https://www.uxpin.com/studio/blog/high-fidelity-prototyping-low-fidelity-difference/ [Accessed 2023-11-24].

Willocx, M., Vossaert, J. and Naessens, V., 2015. A quantitative assessment of performance in mobile app development tools [Online]. *2015 ieee international conference on mobile services*. pp.454–461. Available from: https://doi.org/10.1109/MobServ.2015.68.

Wimmer, C., Untertrifaller, A. and Grechenig, T., 2020. SketchingInterfaces: A Tool for Automatically Generating High-Fidelity User Interface Mockups from Hand-Drawn Sketches. *32nd australian conference on human-computer interaction* [Online], pp.538–545. Available from: https://doi.org/10.1145/3441000.3441015.

Zhang, H., Cisse, M., Dauphin, Y.N. and Lopez-Paz, D., 2017. mixup: Beyond Empirical Risk Minimization. *arxiv* [Online]. 1710.09412, Available from: https://doi.org/10.48550/arxiv.1710.09412.

Appendix A

Raw Results Output

A.1 User survey results

The results of the user survey we conducted (see Chapter 6) are presented in the tables below.

Participant	Role	Question 1	Question 2	Question 3	Question 4	Question 5
A	Developer	5	Hand drawing is pretty intuitive, the recognition can be a bit off sometimes	Being able to correct the model would be handy. Once classified it would be good to be able to move the sketched elements around to reposition	Partly, the model sometimes can't determine what I've drawn	Yes
B	Designer	4	There's a bit of a delay when generating/recognition, the colour choices (i.e. the nav bar being blue) was unexpected, wouldn't know which panel to draw on immediately, needs the intro	Split the screen better to be clearer on what is the sketching space and what is the preview space, Figma does this well. Could also import images of sketches.	Can't really user test with it but in a meeting this would be a good way to show/communicate to stakeholders on the fly	Yes
C	Developer	3.5	Very easy to draw diagrams	A way to add certain common shapes / view types and make them drag and drop would make it easier for myself as I can't draw to save my life. It would be nice to be able to extend the length of a drawing to show that content should scroll	It worked well for and mostly correctly identified what I was attempting to draw. The option to choose between SwiftUI and UIKit would make it more effective for me.	Not yet as I don't use SwiftUI, if it was UIKit then yes it would.

D	Product manager	3	Didn't know what components it would or wouldn't recognise, what the limitations were. When it did things wrong I didn't know how to correct it.	I would have some sort of onboarding and expose the debug view of what the AI is seeing to the user	No	No
E	Developer	5	Drawing was easy and responsive. Generation of prototype was quick to pick up what I had drawn.		It got most of what I added to the screen correct and created valid code	It definitely gives a good starting point for the code to work from

F	UX designer	2	Because of the detection issues the undo function plays a crucial role, and the proximity thing where it struggles with elements drawn too close together. I liked how you showed me what you could draw, that's how I would have drawn an image but some proper onboarding might be good.	More prominent undo button, proper onboarding screen(s). Styles of things like, say, a checkbox and you could change to a switch instead of a checkbox. And then do the same sort of thing but maybe there's, like, a single gesture to draw one of those things and then you can then just somehow toggle them into different modes. Grid alignment as well. I wanted to change pen styles as well and do things with that like typographic hierarchy. As well as drag and pinch and zoom on elements that were already drawn.	Not really but it could do if it was more reliable.	Kind of, given the code output it's actually really useful as I can't write any SwiftUI right now so this gives me a starting point.

Participant	Question 6	Question 7	Question 8	Question 9	Question 10
A	It would massively help you to quickly wireframe and present to people a fully formed UI.	N/A	N/A	N/A	Yes
B	I'd have to use it properly in the design process to give an accurate answer, but a finished version would probably speed me up. More in the early stages of a project for testing quick ideas and communicating with stakeholders or in a workshop.	It's a nice way to encourage people to get involved in drawing in workshops as they won't be judged on their drawing skills.	It has the potential to complement.	Would want a way to share a prototype quickly with users for testing.	It has the potential to potential to but not currently. Does have the potential to encourage more people to input in the early stages.
C		Being able to quickly get a code-able design to experiment with without having to first create the a design via Sketch/Figma.	It helps for prototyping ideas, as it reduces the feedback cycle.	Exporting more file formats would be interesting, Sketch/Figma/PNGs so someone other than a developer can see the design.	Compared to pencil and paper this is much faster.
D	Kinda no	We tend to prototype straight into Figma and rarely do paper drawing	Neither		

E	With more accuracy then the productivity gains could be large	It would be easy to implement into the existing flow due to the ability to export the generated code	Definitely complements	Being able to import something from Figma or Sketch for example would be great	Yes, I can see this tool providing more real-world and achievable designs and would identify potential design limitations earlier
F	Kinda	I do a lot on whiteboards, if it could do that with a big undo button it would be amazing	Neither		We don't really do pen and paper

Participant	Question 11	Question 12	Question 13	Question 14	Question 15
A	For prototyping and getting stakeholders to understand it would save hours of work.	Although it generates absolutely positioned code it gives a good starting point for a junior developer to be given the code and run with it.	Could do with more white space/indenting. Code itself is fine. Might be nice to have some comments on positioning so you can relate it back to the preview/sketch as the code output doesn't match the order of the sketch (left to right or top to bottom) so it can be hard to identify which code block is which element	Yes	There could be better support for theming the UI. Having support to add colours from the client's brand in quickly would be a massive selling point.
B		Not a developer	Not a developer but could figure some bits out	Could edit basic things like words but would probably change things like words by editing the drawing.	Doesn't offer enough flexibility for brands or if you wanted to do high fidelity, but it offers enough flexibility for wire-frames
C	Being able to prototype quickly has greatly improved efficiency.	Whilst not perfect, it is human readable making it easy to make tweaks to fix it.		Yes fine-tuning is possible as the code is easily readable	It's limited to fairly simple designs currently, which works well for prototyping but not for more involved screens.
D				I think I could work out some things to change	

E	Hard to say at this stage, but the possibilities are significant	It's basic but does the job. Would be good to add colours and other design elements (images)	The code is basic but is runnable, valid code that forms a basis to iterate from	Yes, absolutely for the reasons mentioned in the "Design Quality" section	It picked up a number of different controls and placement so there is a good amount of flexibility. Additional native controls would be useful (such as switches, sliders)
F	Not so much			Yeah I could figure it out	Yeah it gives me a staring point

Participant	Question 16	Question 17	Question 18	Question 19
A	Object recognition and not having the option to classify or correct the model yourself is the biggest pain point. You have to just rub it out and try again.	Yes, I would say this is a really good alpha product and demonstrates what the app is trying to do.	Theming, i.e. specifying default colours or fonts. Classifying items yourself. The ability to add multiple screens and link them together so that a button navigates to another screen. Being able to move the drawn UI elements around.	They're not similar enough, other design tools are things like Adobe XD or Figma which offer a different feature set.

B	Just the onboarding of figuring out where to draw initially.	Yeah it's cool, it's in a place where I could use it for early stage design bits.		Figma is my normal tool and you'd have to create the whole thing in Figma whereas this you can just create a sketch. Figma however gives a huge amount of control of what the end result looks like and configuring what happens between screens, it's getting swanky. Easy to share a Figma design with users but it's not as quick to make. Protopie is another but that needs quite a lot of specific knowledge and is hard to adopt, but you can put logic in to it, you can do motion and swanky stuff but it's quite niche and hard to learn and slow to build. You export to Protopie from Figma and you have to be very careful about layer names whereas most designers don't bother naming layers as there's loads of them. Marvel lets you take a picture of a sketch and turn it in to a clickable prototype but you have to manually label the areas in the image.
C	My lack of ability to draw what I wanted didn't always give the AI the chance to understand what I wanted, some prebuilt drag and drop elements might help with this.	Yes for quickly prototyping ideas	Additional "assistance" when drawing (drag and drop pre-drawn elements). Exporting more file types: PNG/Sketch/Figma Support for UIKit	I often use sketch to prototype designs. Sketch allows for more detailed designs but this allows me to instantly get usable code to prototype with.
D		Yes		

E	Some elements were not identified correctly and there is no option to tell the program that it is incorrect and needs to "try again"!	It's an impressive prototype to be able to do what it does, and so quickly. It would need more work to become fully adoptable but I would definitely recommend it.	As above, it needs to be more flexible to identify more components and more advanced layouts (being able to arrange into a stack view for example)	There's nothing like it
F	Detection was flaky and drawing elements close together didn't really work			It's not a comparison I can make because there's just nothing like it, it's kind of like no-code swift.

A.2 Model performance

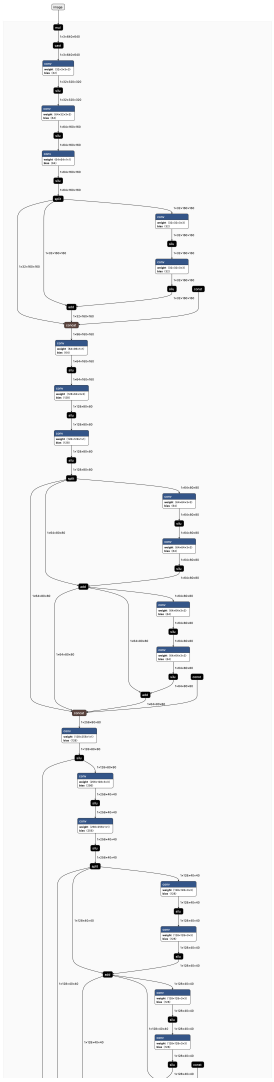
The following table lists the time taken for the model to detect and classify elements on a given image and the total time including generation of the code and UI elements.

Run	Detection/classification	Detection/classification and generation
1	2.200407	2.683552
2	2.222996	2.698845
3	2.249525	2.716664
4	2.283714	2.725142
5	2.306578	2.740880
6	2.335309	2.757163
7	2.359915	2.776879
8	0.122166	0.142193
9	0.131363	0.151537
10	0.126699	0.147279
11	0.105334	0.137893
12	0.113050	0.133771
13	0.117333	0.158697
14	0.116052	0.122958
15	0.108991	0.148219
16	0.120250	0.158703
17	0.129776	0.172703
18	0.113283	0.153022
19	0.114700	0.153700
20	0.126755	0.166771
21	0.132006	0.170730
22	0.116697	0.156618
23	0.130657	0.161833
24	0.120297	0.170574
25	0.129638	0.205070
26	0.116375	0.179115
27	0.118080	0.162460
28	0.133685	0.172663

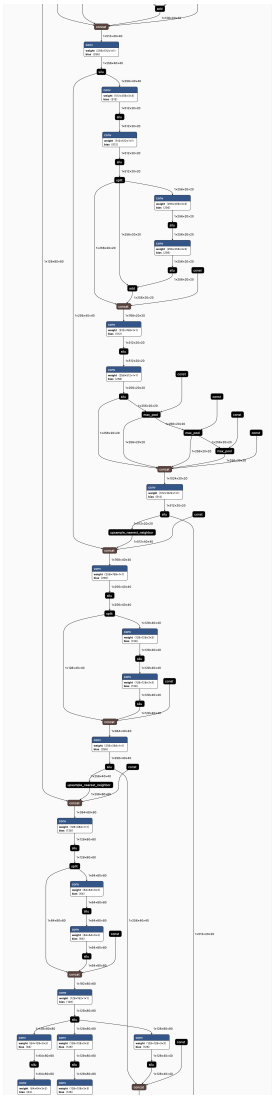
Table A.5: The time taken in seconds to run our trained model and the time taken to run our model and generate code.

Appendix B

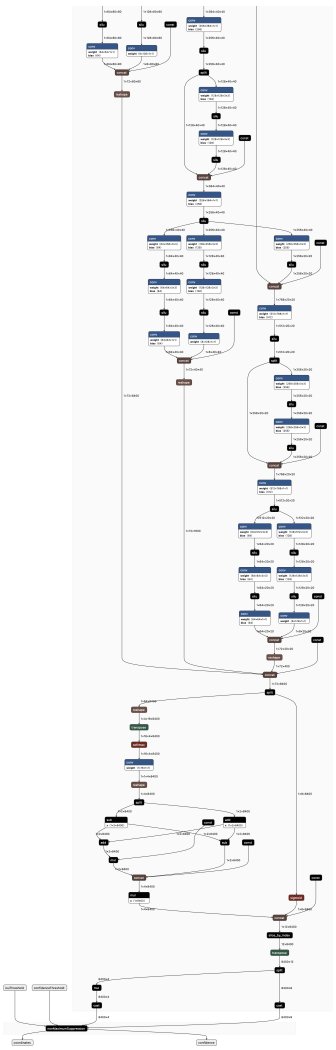
Model



(a) The start of the pipeline



(b) The middle of the pipeline (continuing from the bottom of B.1a)



(c) The end of the pipeline (continuing from the bottom of B.1b)

Figure B.1: The full pipeline containing the trained model and the NMS model at the end.